Chapter 2  

Literature Review

The basis for this research project on an object-oriented library systems domain application framework draws its beginning from the extensive research carried out on previous work in the area of software reuse. The relationship to previous work done by methodologists and object-oriented software developers contributes to the foundation for this research project. There are important precursors that motivate the development of an object-oriented library systems domain application framework, such as to provide a significant degree of software reusability and reliability as compared to traditional software reuse of class library and the emphasis of a stable layered architectural development for the problem domain. The analysis and design of a domain-specific application framework highlighted the use of an object-oriented modeling language. The literature review described in this chapter provides the background and foundation of a proposal of the current research project which is presented as the second last section of this chapter.

2.1 Application Frameworks

2.1.1 Background and Definitions

Technically, an application framework is a set of classes that cooperate closely with each other and together embody a reusable design for a general category of related problems (Budd, 1997). The framework dictates the overall structure of the applications within the same domain. It describes how responsibilities are partitioned
between various software components and how these components must interact with each other.

(Gamma, 1994) offers a similar definition with regards to framework, i.e. "a framework is a set of cooperating classes that make up a reusable design for a specific class of software". The features of the cooperating classes have to be generic and without creating a full fledged application, so that they can later be customised. A framework is not a finished software product, rather it is a solid base of design that eventually helps in building an application. The application developer who utilises a framework has to understand the design of the framework in order to customise it to the specific application needs.

Before we can develop a framework, we need to understand its features. Therefore, it is necessary to draw from current development or expertise related to framework development. A widely accepted definition of framework in relation to design features comes from (Johnson, 1988), who claims "a framework is a set of classes that embodies an abstract design for solutions to a family of related problems". Therefore, we can define "a framework as a collection of cooperating classes that are put together as a reusable library" that exhibits a certain infrastructure features to serve common business problems.

We can acknowledge that design pattern is used to develop a framework. A framework predefines some guidelines or design parameters that application developers can recognise as patterns to help develop applications that are customised according to their needs. The inclusion of design patterns methodology can assist framework development by providing systematic design techniques. We
should design a framework that is adaptable to other related application development in order to prove itself useful and reusable.

In traditional approaches towards software reuse, class libraries are developed from a large number of small classes or methods without any architecture for a business domain. Object-oriented reuse implemented by framework development allows the highest common abstraction level among similarly related applications to be captured in terms of general concepts and structures of the business domain. An object-oriented (OO) framework can be seen as a class library which is built on a systematic and extensive use of polymorphism and inheritance towards achieving a common solution of an architectural business domain. Each application developed from a framework only needs to incorporate the additional and specific functionality as expected by the application developers. New, specific and concrete classes can be integrated on top of a framework which is made up of a group of generic classes that serve common purposes of the problem domain among related applications. The amount of unique code is proportional only to the amount of specific features in the application end product.

Frameworks are ideally suited for capturing the commonalities in a package or product family. The bulk of the business functionality can be captured in the framework which is maintained as a single system. Only a very generic, adaptable framework can last for future projects. If a framework gets too specific, it has to be redesigned to cater to other application development needs. The initial design of a reusable software has to be adaptable to as many project developments. Genericity is vital. Yet, if the design is too simplistic then it would not provide many useful functions.
(Jacobson, 1997) defines a framework as an abstract subsystem with a small architecture that offers an incomplete template for systems within a particular domain. A framework serves as a foundation to an application development that provides a domain system with basic architecture of classes packed in software components and designs of subsystems. The framework dictates the architecture of the application. The specific software design and development are left to application developers who will develop specific needs of an application end product. A framework is useful for a sizeable system that involves complex architecture and multiple packages of software components that are integrated together. By developing an initial small architecture, framework developers can concentrate on a subsystem framework at a time and eventually integrate together multiple subsystem frameworks for a business domain.

2.1.2 Collaborating Classes

An application framework on the library domain incorporates several entity types and the framework relies on individual or composite entity types for identification. Framework users will have to customise the behaviour of a framework by using different combination of objects. A framework dictates the called objects and objects are passed into the framework to be processed by the methods, and as such, it determines how the objects can be combined. The relationship between the domain classes of the framework is very important. According to (D'Souza, 1998), a framework is a collection of collaborating adaptable classes. No doubt that the classes of a framework are purposely kept simple and consistent to the problem domain so that they can be adapted to different needs of more specific applications.
to be developed. An application consists of collaborating classes from interacting components comprising the framework. An object-oriented framework is often characterised as a set of abstract and concrete classes that collaborate to provide the skeleton of an implementation for an application. Before concrete classes are implemented and objects instantiated, these classes will derive the abstract class or implement a common interface.

Application developers, who do not use the default application implementation that comes with the framework, can plug-in customised classes with specialised functions that exist in the form of software components. The new classes would override or overload the default implementation offered by the framework classes (D'Souza, 1998).

For example, a framework for item rental business domain can be used to build a video rental application, a property rental application, a resource library application or other related applications. All these applications share the common software components of the framework such as data store, retrieval of item and entity types of the domain classes.

2.1.3 Class Library Versus Framework

A class library is a set of utility classes or component classes that assist in the development of applications. Class libraries usually have no specific business aims to serve a system development. A class library is like a large-grained software component without thoughts on the problem domain. A compiler usually comes with a class library. Additionally, there are third-party libraries that can be linked with an application. The classes in a class library are often related by inheritance (Lewis, 1998). The Java Application Programming Interface (API) is a good example of a
hierarchical set of class libraries to assist software development. The specific functions offered by a class library has to be inherited or imported. The Java language is a tree of class inheritance from the base root class Object.

A question needs to be asked about the difference between a class library and a framework. A class library acts as a generic solution while a framework is more specific to a problem domain but sometimes they share some behaviour and try to imitate some functions. These overlapping functions are the results of a range of continuum with the class library at one end and the specific, sophisticated framework at the other end.

There are important differences between using a framework and a class library. Firstly, the class library is actually a traditional reuse pattern that shares common pieces of code across different classes by including them during initial declaration.

A class library does not have to be specially catered to any problem domain. The concern is mainly on the programming tool. Although related methods are grouped together in a separate class header, the packaging of class components is not carried out. A traditional class library usually concentrates on low-level implementation.

An application framework is more concerned with high-level design and implementation that concentrate on the business logic of the problem domain. Traditionally, a class library that contains procedural methods does not offer a domain-specific solution. It cannot solve a problem-specific domain because it is difficult to provide a default behaviour to the problem domain in a library (IBM, 1998). A traditional class library consists of class methods that have generic and widespread use in application development without being specific to a business area,
hence, not domain-specific. Thus, the class library does not provide the domain-specific class interface to create the framework architecture.

By having some class interfaces, we can inherit, customise and implement the framework classes to build our application. Once the framework classes are understood by the application developers, they can integrate their specific class implementations to the framework class, thereby extending and specifying the framework's behaviour.

Reusable objects and classes assign to logical groups or namespaces in a class library have created a perception of building a framework that is domain-specific or category-specific such as a business framework or Graphical User Interface (GUI) framework respectively.

2.1.4 Constituents of Framework

2.1.4.1 Software Components

A software component is a logical grouping of classes that can make up a subsystem. A software component performs a specific task and can be combined with other components. By representing itself as a subsystem, a software component is part of a larger system.

In most cases, components combine together to work as a complete system. Component-based development is the basis of building software applications from common foundation components, which as a whole, could form a framework. A framework is made up of components. Components are made up of business rules, application functionality, data or resources that are encapsulated to allow reuse in
multiple applications (Hurwitz, 1997). Components should be able to reuse across multiple applications to see its significant success.

There are 2 kinds of software components, which are fine-grained components and large-grained components. The fine-grained components are software components designed for general-purpose use. GUI controls, networking components, and spreadsheet utility controls are all examples of fine-grain components. The large-grained components are designed to solve a problem in a specific domain or for a particular type of product. An example of large-grain components used is a business domain application framework. Components, by themselves, are not very useful. For a software component to be useful, it must be fitted with other components into an application or a framework environment for it to perform as a working system. A software component is self-contained which is an important feature of software reusability and portability into different environment systems. A useful software component must have a well-defined interface which acts as a recognisable signature for integration with other components and can become a convenience to identify and use it. A lower-level term definition of a component is actually a grouping of related classes that interoperate and share common interfaces. Another component has to interact through this interface to invoke particular methods from the classes. The granularity of a component can range from a simple class to a whole system (Jacobson, 1997).

Application developers normally concentrate more on solving business logic of applications instead of technical details. By having component or framework reusability at their dispense, they can do more business solving. Presently, software development has advanced from object reusability to component reusability.
An assembly of components in the same domain also is a feature to be considered during framework development. To be truly effective in reusability, components should be portable and interoperable across applications (Jolin, 1996). Consider a library transaction component implementation that handles everything from the user interface data entry to the database update. If this activity were implemented as a self-contained entity that included the code from all the different classes of varying levels, we will create a vertical component. A vertical component is an entity that implements a business process (Finch, 1998). Commonly referred to as the "black box", a vertical component can be defined in terms of its input, business rules and output. A black box component is specific in its function. However, an application framework should be set out to achieve higher-level reuse that interoperate between system components in a domain-specific architecture.

Similar functional requirements between applications in a problem domain should be extracted into common components. These components are of horizontal type. Horizontal components of a framework are architecture specific. For example, a Java component acting as the front-end graphical user interface, the database as another component, middleware component and the business logic component at the back-end, all possibly on different architectural platforms. A good solution for code reuse is to combine vertical components with horizontal components. The vertical component provides functional benefits and the horizontal component provides architectural benefits. The processing in the horizontal component spans across a vertical component at a single architectural platform. We have business requirements incorporated into vertical components that span from the front-end, database, middleware and back-end. Hence, an application framework will incorporate both types of components which will have the features of a multi-tier architecture and
business requirements. A multi-tier architecture is comprised of a middleware, client and server. The client machine will usually handle the data display and request through its user-interface part of the application. The server side will usually perform the business logic and data management tasks. The middleware will provide utility classes for language implementation such as Java Collection framework classes and Component Object Request Broker Architecture (CORBA) (OMG, 1996) that marshals objects across network. CORBA facilities and services have become a vendor interoperability standard that ensures software components on different architectures communicate with one another.

By extending from the horizontal and vertical component model, we can move all related abstractions of business rules component and user interface component into an integrated framework. Most widgets for user interface are put into a class library and business rules are put into a knowledge-based repository.

Objects from the horizontal component will process on the business rules objects which the vertical component depends. As the end result, we can build horizontal components that query data structures and business rules associated with the application's vertical components (Finch, 1998).

![Diagram of Software Component Layers](image)

**Figure 2.1: Software Component Layers (Finch, 1998)**

Figure 2.1 shows software component layers comprised of horizontal layers such as a business rules repository, a widget's class library and processing objects. By
combining these horizontal components into the higher-level and lower-level part of vertical components, we can achieve reusability of software components if we lay them out in a layered pattern.

The interoperability between components is a key feature of compatible public interfaces, customised properties and generated events. A packaged component can be distributed more easily since it can interoperate with compatible components as recognisable grouping of classes.

**Figure 2.2: Types of Partitioned Business Components (Andersen, 2000)**

According to Figure 2.2 from (Andersen, 2000), business components can be made up of two types, such as entity-centric and process-centric. The entity-centric component represents entities in the business domain and encapsulates information, behaviour and rules that effect the entities. Examples of entity-centric components for
the library domain include borrower, item and title. A borrower entity component would encapsulate personal information that the business or library needs, type of borrower and rules that apply.

The process-centric component represents significant business processes or the type of work that needs to be done. This component encapsulates behaviour, rules and information associated with the process. For example, a "Loan" component will process a loan according to information such as type of borrower, item and rules that apply such as the loan period. The process component, however, is not void of entities but more process-based. The data model package in this library domain application framework is made up of more entity-centric than process-centric classes that model the business domain. The processing of a loan or reservation type is fully implemented in the process-centric components such as the controller component of Model, View and Controller (MVC) design pattern (Burbeck, 1992). MVC is about a recognisable pattern that alerts a client when the state of a server is changed. The user interface component is controlled by user events generated from the windows interface. The user interface controls are grouped into the View component of MVC.

Renown framework examples adopting this design pattern are used to support the standard functionality required by all GUI applications for a particular programming language and operating system like the Apple MacApp, Microsoft MFC and Borland OWL system. Major software companies have produced graphical application frameworks to assist software developers in building object-oriented applications.
We can consider the process-centric components as in control of business processes and functions. While the entity-centric components are data and entity-based, they are controlled by process-centric components that dictate the flow of a business process by requesting data and services in a specific sequence according to specific business rules (Andersen, 2000). The entity-centric component often provides the data and services required by process-centric components.

### 2.1.4.2 Robust Components

A good framework has robust component systems that are quickly adapted to the different needs of the application developers. The users of a framework can pick and choose the subsystems and components if they are flexible and well-designed. As mentioned earlier, a framework consists of interacting subsystems and components which form the workproducts of a business system domain.

Component Systems Engineering (CSE) defined by Jacobson (Jacobson, 1997) concentrates on building and packaging a component with the purpose of reusability. This component acts as a smaller framework that can serve different areas of needs if it is built in generic ways and easily configurable. Good packaging will produce a black-box framework that can be reused as it is. Documentation is important as part of a good component package to help developers understand the framework reusability.

The first step in CSE is capturing requirements and focusing on variability. From use cases, we can capture these requirements for one or more components. A use case is a sequence of transactions performed by a system to serve a end-user
(Jacobson, 1997). By identifying the commonalities and variabilities among related application systems in the same domain, the component development must meet the changing needs. Primarily, the robust components have to capture the common requirements and also provide an extension to the variable requirements so that they can serve as smaller frameworks to be reused over again.

2.1.5 Other Types of Frameworks

2.1.5.1 Architectural Framework

A foundation of software layers constructs an information technology (IT) architecture or provides an architectural framework that is suitable to the business strategy. An architectural framework is a software tool which can be used for developing a broad range of different architectures (OpenGroup, 2000). TOGAF from OpenGroup is such an architectural framework developed to assist software architects build a customised architecture of their IT business. An architectural framework can be used to design, evaluate and build the right architecture for the business organisation. Currently in the software market, Java 2 Enterprise Edition (J2EE) is a significant contribution to business enterprise in developing the right architecture for their IT network resources. A good IT business architecture provides benefits of efficient IT operation, lowers software development cost and most importantly, improves interoperability and network management. J2EE and CORBA can be considered as powerful architecture frameworks that make software components reusable on different platforms and usually resided in the middleware layer of a distributed software architecture.
2.1.5.2 Technical Framework

An utility package such as Java Collection Framework provided in Java programming language is a kind of technical framework that conveniently assist software development with flexible data structures and algorithms encapsulated in abstract classes, utility methods and classes. Unlike an architectural framework which resolves to remain language independent for interoperability between different platform systems, a technical framework is language-based. Other examples, such as JavaBeans, ActiveX or COM are software components in a language dependent architecture.

2.2 Identification of Domain's Functions

From surveys and interviews with the domain experts, we can analyse and identify the core solutions and business logic of the problem domain. The business logic would provide us with an abstraction of domain's functions to develop the framework.

The easiest way to identify business core functions is by using a bottom-up approach (Adair, 1998). By examining existing domain solutions, we can abstract the business logic and find the common solutions shared among related applications within the same domain. If we look carefully at the business process, we can produce the logical abstractions and identify the business entities before mapping out the types, classes, their relationships and hierarchies.
Some framework developers would draw from their past experience and previous project designs in the problem domain to build the framework. They identify the abstractions and begin designing the framework.

As suggested in (Adair, 1998), those people who are not experts in the problem domain or have not developed any applications for it, should examine existing applications and consider writing an application in the domain. Another important point is to learn from the domain expert to come up with the first use case analysis (Jacobson, 1997). A use case is a typical interaction scenario between a user and a computer system. In Unified Modelling Language (UML) methodology, a user is known as an actor and a computer system consists of multiple use cases. Use case analysis would factor out the common classes and identify the abstracted business logic and solutions with names.

In a library domain, for example, the abstraction would be activities about items borrowed, returned or reserved. These activities are identified as the functions of a data model which are usually separated from the database functions. The database functions adopted for the system can differ according to implementation preferences, for example, software developers can choose to use a relational database or object-oriented database management system.

The number of items borrowed, returned or reserved for each borrower can vary. Hence, there are two fundamental and important forms of abstractions, such as entity abstractions and action abstractions, as suggested by Seidewitz (Seidewitz, 1986). An entity abstraction is about an object that exists within a model of the problem domain. The author uses UML class diagram to show a static structure of concepts, types, and classes. Concepts show how software designers think about the problem.
domain, the types show interfaces of software components and classes show implementation of software components.

A group of objects designed with respective generalised set of operations collaborating in a UML interaction diagram, are considered action abstractions. An interaction diagram shows how several objects collaborate in a single use case.

2.2.1 Common Features

From the abstractions of use cases, we can draw out the common features of a problem domain. These common features are the core requirements of a framework development. Common features are termed as "Commonality" by Jacobson (Jacobson, 1997), which is described as the set of features or properties of a component, component systems or application systems that are the same or common between systems. The commonality can be exploited by building a set of common reusable components. Application developers who use the library systems domain application framework can depend on the common requirements and features of a library system from the software components of the framework. Hence, software reuse is an advantage for faster, reliable and cost-effective development especially in terms of a larger scale reuse, such as from the software components of a framework.

2.2.2 Variable Features

The different mechanisms of delivering a featured activity on a use case is known as a variation point in use case modelling. A use case can have one or more
variation points. A framework will shift the responsibility of the variation points to other specific set of classes external to the framework such as a plug component. This set of classes have to be created by application developers as derived classes or associated classes. The framework classes will have abstract methods so that they can be redefined in subclasses defined by application developers. These methods should be common to all related cases in the problem domain. In summary, a framework defers the variation points to subclasses.

A significant part of building a framework is to distant the variation points from the common attributes and methods. These variation points are implemented as customised features at the lower level by application developers and are derived from an abstract component of a framework.

![Diagram](image)

**Figure 2.3: Extension of Variation Point.**

For example as shown in Figure 2.3, a use case of borrowing an item is a common requirement in a library systems domain application framework, however, variations in some application requirements such as imposition of a rental fee or otherwise is anticipated. This anticipation is defined as a variation point in a use case which is displayed with a dot. The implementation of the different requirements are extended from the use case or component via the stereotype «extend>> into other
use case or component. A stereotype is a modelling element that extends the semantics of a metamodel (Jacobson, 1997). There are different extension mechanisms available for implementation, such as inheritance, delegation, template parameters and script generators, which will ensure that the business component remains robust for future use. Inheritance is a common mechanism in object-oriented design and programming that allows more specific attributes and methods to be included in derived classes besides inherited common base classes. Delegation allows the extension of a method to a server class to fulfil the operation of a client class. A template parameter allows a basic generic type to be defined with a specific data type during parameter referencing. A script generator based on a scripting language allows multiple dynamic types creations.

Application systems with varying needs should build lower-level specialised components that import attributes and basic operations from the higher-level common business components.

2.3 Design Mechanisms

The concept of "Model-View-Controller" (MVC) which has its origin from Smalltalk, is a type of design pattern. MVC assists in developing a framework because it segregates the model of the domain from the application's controlling functions that are built on top of the model. The view is simply the appearance to the user of a complete application. MVC is an example of design mechanism of how classes collaborate to achieve a solution. The design mechanism represents a level of reuse that is higher than the reuse of individual classes (Booch, 1994). The dependency mechanism of model classes in MVC is a design decision that can be found in almost
every domain. The domain model is a representation of the framework while the controller consists of most of the application functions that an application software developer caters to the needs of a specific user.

A "Model" is actually the data model underlying the object while the "Controller" determines how a user's interactions and events cause the data in the model to change. The "View" accesses the data from the "Model" and specifies how the data is drawn on the screen with the aid of graphical icons and custom controls. The MVC is a proven concept that divides a hugely complex problem into the three areas of tasks that should be handled separately and eventually coupled as a cohesive unit of an application.

Since a framework development is influenced by individual functions and graphical interfaces which are decided by the software developers, we can decouple as much as possible and build a model, controller and view that act as separate units with some cohesion.

Once the domain model components are accomplished, application developers can easily and independently develop the controller and view components. The model components which constitute the framework should not be overly redesigned because most application systems developed on the same framework components will have to be redesigned to fit the component interfaces.

Besides the data model, the framework is also a representation of data structure, access methods, persistency and relationships among the classes involved. A good object-oriented decomposition and interface between these three subsystems, model, controller and view, will assist development effort in scaling up software, modification or enhancement of software, and make reusability of a complex software
system more practical. By having a clear separation and encapsulation of the underlying data model classes in packages or components, a reworking of a confined class would not cause a rippling effect to other classes and encourages reusability of software code available in classes.

Most of the time, software requirements by end-users require different views and variations in functions of a domain but would not expect the software developers to build a drastically different domain application framework. Programmers usually subclass from the data model abstractions of a framework and customise the design and implementation of the eventual application. An object-oriented (OO) framework can better provide this kind of reusable solutions, once a model, controller and view concept is practised in building a framework.

How the model actually decides to store the data is assisted by the controller's functions, which in most cases, are incorporated with database functionality to ensure persistency. Additionally, the model might not contain data at all but rather acts as a proxy for data access from a persistent data store.

Some guidelines are necessary to framework builders. The following are three suggested guidelines:

- Derive a framework from existing business problems and solutions. Obviously, a major recurring, problem-solving effort requires a helpful assistance in the form of a framework that accelerates any software development effort.

- Develop a small-scaled and domain-focused framework which concentrates on specific type of problems to be solved for the domain. This brings the reference to a library systems domain application framework.
• Frameworks should be built on experiences that share common abstracted problems to solve generalised problems and leave the specific problems to application developers.

Obviously, a framework is a continuous effort that is improved with iterative process driven by client participation and prototyping. A framework should be a product that provides documentation and support as it will help in understanding and maintenance. These helpful guidelines go a long way to get people started with a framework that is easy to learn and use. The framework should have a short learning curve.

Although a framework can be built from a non-OO programming language, object-orientation would be resourceful and reusable from its features of classes and objects that make the framework design and code reusable. A class acts like a building block that developers use, extend or customised in software development.

The blocks of code can be substituted with minimal impact on the well-designed infrastructure of a framework. Thus, the framework can avoid any major redesign which would require substantial effort, and instead, software developers can concentrate on the specific user requirements which are actually the incorporated building blocks.

The good distinguished features of OO languages that prove very productive to software development effort are inheritance and polymorphism. Users can customise the behaviour of the framework by deriving new classes from the framework and overriding member functions. Obviously, all these distinguished features prove to be worthy use of OO languages. Architecture-driven frameworks rely on inheritance for
customisation and polymorphism for adaptation which are acted out by overriding methods.

2.4 Subsystems via Unified Modeling Language (UML) Package

According to UML design notation, a "package" is a group of related classes put together with a common interface that identifies software components and divides them into subsystems. To put it simply, a framework consists of classes in components. Analysis and design classes put into packaged components will assist in a framework development, as they divide a large and complex system into subsystems that are better managed as reusable components in identifiable packages that can be developed concurrently.

![Diagram](image)

**Figure 2.4: Interacting Components**

In Figure 2.4, an actor or user will interact with the software system by entering input messages to classes in the software components. The component in the subsystem package can have dependencies with other software component within the package to carry out a business function. When the process to the message is over, the output message is delivered back to the user.
Based on the UML package diagram, we can draw a close comparison between the package construct and the component which appear in the UML deployment diagram as physical module of code.

Classes in Java programming language can be grouped together to form the package concept as that exists in UML. However, analysis and design of component classes should not be influenced by the language of implementation, particularly in a framework development, which is meant to be generic and reusable over the business domain that can have different architectural platforms. A package can be easily identified with a <<stereotype>> tag to extend the metamodel semantic (Jacobson, 1997). A software component is encapsulated as it is considered a black box that offers a set of related services without making the source code available.

According to UML package diagram, we can show the dependencies between classes of different packages. The low-level application classes in the controller package have to be linked by dependencies with the domain classes and user interface classes. The stereotype tags will help us recognise the interface and purpose of the packages.
Figure 2.5: Package Diagram

In Figure 2.5, the package diagram represents the superordinate system which is a high-level design of subsystems such as the model package, controller package and view package. The model package is identified with the domain stereotype tag that interacts with the controller package identified with the application stereotype tag, and which interacts with the view package identified with the user interface (UI) stereotype tag. The association between the controller package and the user interface package is a form of communication to exchange messages. The controller package has a dependency relationship with the model package which is consistent with the domain.

The business logic and data model classes would be put into the model package. The application class functions are put into the controller package. The user interface classes of the produced application system would be in the view package.

By putting software components into packages, it is effective in reusing them because application developers can search and identify the software packages. We can identify a package that is named and arranged in a common hierarchy of packages, this is especially useful in a large software project. They only need to
import a specific package that has components which export specific interfaces or facades. A facade is a packaged subset of components, or references to components, selected from the component system (Jacobson, 1997). A component system offers packaged components with one or more facades that behave as a system product for software reusability. Each facade provides public access to only those parts of the component system that have been chosen to be available for reuse.

The attributes of a good designed package are that it should have one or more facades, descriptions, class and method identification signatures, together with good documentation. A package can be uniquely identified by the above-mentioned attributes and extended by parameterisation, templates and script generation mechanisms. The rightly identified relationships and associations of these attributes among packages is a first and major step towards better accuracy of systems design.

2.5 Survey of Library Systems Domain

A survey was conducted via interviews with library personnel from a local primary school, a secondary school, a college, a university and several public libraries. The primary and secondary schools impose their own system of loan and return of library items. The college and university library systems function on their own computerised system which has more services such as database search, retrieval, records of loan, reservation, etc. and other details such as borrowers, items and details of each copy of item. Fines are also recorded and rated. The types of items offered to borrowers from the younger age group in the primary and secondary schools are limited to books and magazines. The college and university offer a more
variety of items to be borrowed such as books, magazines, audio, video, CD and others. A survey was carried out involving the following parties:

- The primary school: SRK. Raja Mahadi, Taman Goodwood, Klang, Selangor, Malaysia.
- The secondary school: SM Raja Mahadi, Taman Goodwood, Klang, Selangor, Malaysia.
- The college: Kolej Damansara Utama, Petaling Jaya, Selangor, Malaysia.
- The university: University Malaya, Kuala Lumpur, Malaysia.
- Perpustakaan Awam, bandaraya Shah Alam, Selangor, Malaysia.

The input from the survey is used for the preparation of the proposal of the current research project to be described in the next section.

2.6 Research Proposal: Development of an OO Application Framework for Library Systems Domain

Software reusability is a significant advantage to software development as compared to development from scratch. The benefits can be in terms of quality and quantity. Software components that are reused are more reliable as they have been developed and tested through several projects. The time taken to develop the end product is shorter. For the library systems domain, the advantage of an application framework is the reusability of the appropriate software components across related applications, such as library systems in primary schools, secondary schools, colleges, universities, public libraries and commercial rental centres. A library
systems domain application framework is modelled and designed as component systems that are placed into a layered system architecture which can be reused and adapted in library application systems development, either in a single or distributed system environment.

How are these software components, which are part of the architecture framework, designed and developed? What about the problem domain analysis such as the library systems domain? For some well-structured and theoretically founded domain, we can identify the architecture and its components. However, there are certain domains without any clear structure and with a complex problem that has to rely on the domain expert to relate the requirements specifications into use case models. High-level domain analysis and abstraction design model can define the activities and requirements of a library systems domain use case into detailed design models for subsystems development.

2.6.1 Research Methodology

Based on the research's main objective to develop an object-oriented application framework, an object-oriented modelling tool is used for system analysis and design. Unified Modelling Language (UML) is the selected choice of tool to perform pre-implementation research of analysis and design on the library systems domain. The library systems domain is the preferred domain because the eventual application framework will prove its usefulness to related library systems which may be for a primary school, a secondary school, a college, a university or any commercial rental centre that offers similar activities such as rental, reservations and fees imposition on the items which may be books, magazines, CD, videos, etc.
The first methodological step is to classify the library related problems into abstract activities as part of the use case model. The use case model should capture the requirements abstractions and actors of the problem domain and trace each abstract use case.

The second methodological step is to model the library systems domain application framework according to Application Family Engineering (AFE) which is a business use case or software engineering process that develops new versions of application systems by reusing component systems (Jacobson, 1997). AFE process develops and maintains the overall layered system architecture so that software reuse is achievable. The first modelling feature of AFE is called the superordinate model which has a high level overview of library subsystems corresponding to an application system and laid out on the different layers of system architecture.

After analysing the library model subsystems and the layered architecture, we design the superordinate model which is defined in terms of subsystems organised into layers with facades between them. The high-level analysis and design component model constructs are, for example, packages, facades, stereotypes, use case and variation points. A use case model is mapped to an analysis model and a high-level design model for tracing the key abstraction of library activities.

Subsystem components are designed to group related object types. Reusable designs of the library object model in relation to a design pattern such as "Model-View-Controller" offers significant benefits to recognisable and practical designs. The design of data model, controller and user interface packages is a distinguished feature of MVC design pattern that would identify each subsystem.
Each of the library subsystems will be designed with UML low-level modelling diagrams as part of the library systems domain model. The class diagram is a low-level static diagram that captures the domain classes which are the major actors or objects that play an active role in the domain data model. The dynamic diagram captures the states and events that take place on the business application system.

The objects' states are important details that persist onto a data storage to be maintained for every object type. However, the type of data storage should be independent from the framework design and implementation. The choice of database left us to consider one that is compatible, resourceful and advanced technology. The object-oriented database is a type of database that can be potentially promising to software reusability because of it's direct implementation in an object-oriented programming language instead of interfacing a database connectivity protocol, such as a relational database. The object-oriented database implementation closely maps object-oriented concepts, such as aggregation and association of relationships among object types. The features of an object-oriented database will be implemented and tested to display the persisted state and attributes of an object, as for example, an item which may be available, reserved or loaned.

The library systems domain application framework is implemented as a set of component systems which comprised of generic abstract classes for the purpose of reusability at a high-level across related applications development.

The advanced features of a framework can be quickly learned if an application framework featuring a helpful wizard is included to assist application developers. The wizard is to help developers generate a working application on-the-fly or provide a customisation feature to produce selected components or packages chosen by the application developers.
The research includes a wizard development and implementation classes for testing the reusability of the library component systems. Hence, the end result is a faster developed and workable library system application. The details of the development methodology will be discussed in Chapter 3, Chapter 4 and Chapter 5.

Figure 2.6: Software Reuse Life Cycle Model

Based on Figure 2.6, a software reuse life cycle model is produced as part of the research methodology to assist in an overview of the development of an object-oriented application framework for library systems domain. The iterative life cycle is an important activity to ensure that incremental analysis and design are
adopted to achieve and increase the reusability of software components. At the
domain analysis step, components analysis activity is carried out, followed by the
subsystem design step with components design activity. After iterations between the
components analysis, components design and components implementation, only
then is components implementation into subsystem is feasible; otherwise, a
component is iterated over the component life cycle again. After successful
subsystem implementation, various subsystems are integrated into a complete
system.

Besides adopting the software iterative approach within the proposed
framework's software development life cycle, the final framework developed can
undergo further enhancement in its post-installation stage by going through a
separate iterative life cycle, taking the developed framework system and the project
experience as input. As depicted in Figure 2.6 with bold arrows, when there are minor
implementation changes, iteration between maintenance and post-installation often
happens. However, when there are many required system changes that would affect
software reusability, there would be the next incremental and iterative life cycle to
improve on the domain analysis and eventual design model. The commonality and
variability of library activities can change and will determine the degree of component
reusability.

The primary research method adopted was conducting interviews with
librarians and teachers cum librarians on the library systems available in the
environments of a primary school, a secondary school, a private college and a public
university. These diverse library systems offer the different perspectives of the most
simplest systems to the most advanced systems in operation.
The latest object-oriented methodology, UML, was adopted in modeling the analysis and design of the library systems domain. One of the most important secondary research adopted was references made into "Software Reuse, Architecture Process and Organisation for Business Process" (Jacobson, 1997). From a secondary research done on the Internet, one of the most advanced and developed object-oriented systems domain application framework was discovered and offers significant insights into a well-publicised material called "San Francisco Business Domain Framework", belonging to IBM (IBM, 1998).

A major proportion of the research effort was carried out in studying the Java language and POET database implementation details. All these relevant research resources have contributed to this research project.

2.6.2 Research Constraints

This research development of a library systems domain application framework is confined to an independent platform, Windows 95, and platform-independent programming language, Java. As Java can run on other platforms without recompilation, this library systems domain application framework is not required to be tested on other platforms.

Other types of databases, such as RDBMS and database servers, are not utilised to implement the application systems developed from the framework. Although the application system implementation is quite different for different types of databases, the framework still remains the same.

The functional aspects of the library systems domain that are not covered in the application framework are, such as database administration which may need reporting of log data, optimisation and configuration to ensure smooth data lookup,
updates and addition. The limited features of the demo version of POET object-oriented database used has only a local communication, which means it can only be developed on a single computer. The limited features of the demo version has no networking services. As such, the database is not fully platform independent but the library application systems developed can be deployed to other computer by first performing the installation of native database system drivers and libraries.

2.7 Summary

An application framework offers a software development approach that is beneficial to a business domain which would reuse it as a featured architecture consistent with the needs of the intended application development. Common features of the domain can be found in the framework and this would help application developers save time in application development. An application framework is a tested system that can be treated as a fine-grained class library with a planned architecture, technical and business decisions that are consistent with the preferred business domain. The framework would exist as software packaged components and can be implemented into overlapping layers of a system architecture.

The library domain functions are identified from existing libraries and this identification helps design the generic features of the application framework. Documented design patterns also assist in identifying the different subsystems of the framework. Variable features of a business domain are isolated as variation points in a UML model. UML as a modelling language would help identify each subsystem as a package of components.