Object-Oriented Application Framework for Statistical Process Control in Manufacturing Domain

Liu Hong Song

2004

Acknowledgement

I would like to thank to my supervisor, Assoc. Prof. Dr. Lee Sai Peck, for giving me the opportunity to carry out this project. I am very grateful to her guidance, advice and suggestion throughout the project.

Thanks to my project partner, Mr. Thin Siew Khim, for his cooperation and help during the development of this project. A special thanks to my friend, Miss Lee Poh Chan, for her encouragement and enduring moral support. Not forgetting also my gratitude to my good friend, Mr. Pang Jen Sen, for his continuous support.

Finally, thanks in advance to those who review this work.

Liu Hong Song

Contents

List of Figures

List of Tables

Abstract

A continuing challenge for software developers is to design and develop efficient and cost-effective software implementation. Designing and developing new manufacturing applications is even time consuming and expensive due to the nature of manufacturing domain, which is diverse and complex. Using an object-oriented application framework is a promising means for alleviating this cost. An object-oriented application framework is a reusable software component providing large-scale reuse, which could include reuse of analysis, design and implementation in order to develop software faster to market.

The main objective of this research is to provide a solution by developing an object-oriented manufacturing application framework in the domain of Statistical Process Control (SPC), with a set of integrated reusable components, which can be customised to suit specific SPC manufacturing applications. This application framework provides a solution for developers to produce specific manufacturing applications by making use of existing integrated reusable components rather than developing these applications from scratch. Based on the structure and behaviour of various SPC applications, a generic model of SPC is proposed by following the concept of object-oriented technology, software component technology and design pattern. This dissertation will describe in detail the architecture and design in the development of the application framework for statistical process control in the manufacturing domain.

Chapter 1

Introduction

One of the main challenges contemporary organisations increasingly face is generated by the complexity of software development. While computing power and network bandwidth have increased dramatically over the past decade, the design and implementation of complex software remain expensive and error-prone (Fayad & Schmidt 1997). The time between two releases of a software product is diminishing, yet the development cycles of the software are still too long, which often result in software products that do not address business problems adequately. One of the main causes is that much of the cost and effort are generated by the continuous rediscovery and reinvention of core concepts and components. As a result, a flexible, extensible and maintainable software is needed to solve these problems.

These problems have motivated researchers and software developers to start adopting the approach of systematic software reuse in the development of software. Important progress towards software reuse was made with the emergence of the object-oriented (OO) paradigm. The use of OO programming (OOP) in conjunction with class libraries promises to lead to software design meeting the above requirements. The OOP mechanisms like inheritance, polymorphism and dynamic binding, enable the development of more

flexible, easily maintainable, and less error-prone software. As a result, the software industry has moved towards embracing object-oriented technology because of its potential to significantly increase developers' productivity and encourage innovation through reuse.

Large-scale software reuse is proven to be able to improve quality and reduce cost and time-to-market in software development (Henry & Faller 1995). Still, all the OO techniques provide reuse only at the level of individual and small components. The more complex problem of reuse at the level of large components that can be adapted for individual applications was not addressed by the OO paradigm itself. Object-oriented application frameworks (OOAF), large OO structures that can be tailored for specific applications, carry the OO paradigm further by providing infrastructure and flexibility for deploying OO technology and enabling reuse at a larger granularity (Bosch, Molin, Mattson & Bengtson 1997). An OOAF, which is an extensible set of object-oriented reusable components that are well integrated to execute well-defined sets of computing behaviour on a certain application domain, is a good answer to a dramatic improvement in software development.

1.1 Motivations

This research is motivated by several problems and needs in the manufacturing application systems development and advancement of the application framework. Manufacturing entails an extensive range of subject areas related to the entire lifecycle of products, from development through production and beyond to product support. As a result, manufacturing systems must be flexible, reliable, and scalable in order to meet the rapidly changing needs of today's enterprises.

Statistical Process Control, is a tool that businesses and industries use to

achieve quality in their products. Universally, businesses and industries use mathematics and statistical measurements to solve problems. There is an increasing demand for managers and workers who understand and are able to apply SPC methods. SPC is a well-proven technique used to monitor the performance of a process, which then provides the basis for achieving continuing improvements in product quality and productivity. SPC using a related set of statistically based tools for monitoring, analysing and effecting improvements in any process. Each SPC facility has specific processing requirements that need to be accommodated in order to successfully operate. Therefore, application framework technology provides a promising approach to building such systems. This research is to develop a statistical process control (SPC) application framework that provides a set of integrated reusable components, which are flexible, customisable and easy to reuse for adaptation of specific SPC applications.

1.2 Problem Statement

Designing an application framework with the integration of the statistical process control (SPC) domain is not an easy task. A good application framework requires good scope definition. If the scope is too broad, the framework will require a great deal of program code to create a specific application or it will require an enormous amount of code to build the framework to a level of complexity where it can handle all the cases without resorting to writing program code. If the scope is too narrow, the framework will not be applicable to enough systems to warrant interest.

The SPC domain in manufacturing involves a lot of applications, including graph plotting, analysis and calculation. After having done a survey and study of the domain, these applications are divided into two groups, which

are the variable control charts and attribute control charts. These two groups of applications consist of most of the basic applications and requirements of SPC systems in the manufacturing field. Variable control charts measure the quantitative measurements of a particular characteristic of a product and plot the results. Attribute control charts measure whether or not a product of a particular system or process meets a certain criteria. An attribute control chart plots the number of outputs meeting this criteria rather than quantitative data as in the variable control charts.

The application framework in this research project should allow software developers to use built-in facilities to develop a new SPC application. This is achieved by designing the framework based on an open design with a layered architecture, allowing the user to extend the framework's functionality. With this application framework, a different SPC application can be developed by simply reusing appropriate reusable components of the application framework. The reusable components are actually the generic components provided by the application framework that can be used to develop a similar but non-identical type of application to suit the specific needs.

In order to let the developer customise and configure the application framework easily, wizards are created to provide step-by-step customisation of the generic component via an Integrated Development Environment (IDE) (Lee, Thin & Liu 2001*a*). The variation and extensible points of the application framework that are used to exploit variability of features are mapped in the GUI (graphical user interface) provided by the IDE. In this way, the developers can customise or extend a variation or extensible point which they selected through a wizard associated to that variation point.

1.3 Objectives of Research

Based on the problem statement discussed in the previous section, the SPC application framework developed in this research project has several objectives as stated below:

- To develop an OO application framework in the domain of manufacturing statistical process control. The development of the application framework is based on the common application systems of the SPC, which are widely used in quality control in manufacturing.
- To provide a good software architecture by designing a layered system architecture that is suitable for the proposed SPC application framework.
- To shorten the time for developing SPC application systems through reusing the framework architecture by interfacing with framework components in GUI form. The application framework should be easy to use and users only need to learn a few functions. In order to achieve this objective, wizards in the IDE are developed for the application framework to provide step-by-step assistance in framework component selection and configuration to a default application.

1.4 Scope of Research

The construction of a SPC application system basically involves implementing a reusable control chart component in the framework for the purpose of later adaption or configuration by developers. At the highest level, a generic control chart component is provided by the application framework to support a variety of control charts in one of its variability points meant for adaption according to the specific needs of the developer, who is also referred to as

reuser interchangeably in various parts of this dissertation. This SPC application framework provides two categories of control charts. They are *variable control charts* and *attribute control charts* which are used for quality control work to control manufacturing process. Variable control charts include *X-Bar Range* chart, *Range* chart, *Sigma* chart and *X-Bar Sigma* chart while attribute control charts include *P* chart, *NP* chart and *C* chart.

1.5 Research Significance

There are several advantages to developing application systems by using application framework approach. The stable interfaces provided by application frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmers' productivity, as well as enhancing the quality, performance, reliability and interoperability of software.

The architecture and design of the application framework is well designed for reuse. Much of the functionality of SPC applications has been made available in the application framework. Therefore, similar applications with some exceptional, specific different requirements can be built by using the application framework as a starting point, rather than developing everything from scratch. As a result, time for development becomes shorter and this will lead to a better cost saving. In addition, specific applications developed from the application framework are more reliable and the efficiency of development will be increased since the application framework has been carefully designed

and tested.

One of the primary benefits of an OO application framework is its intrinsic modularity. An application framework enhances modularity by encapsulating volatile implementation details behind stable interface. Framework modularity helps improve software quality by localising the impact of design and implementation changes. This localisation reduces the effort required to understand and maintain existing software.

1.6 Chapters Organisation

The thesis is organised into the following chapters:

Chapter 2 is a literature review of some related topics such as object-oriented technology, systematic software reuse, component based systems and existing application frameworks. This chapter also contains a brief overview of the manufacturing domain.

Chapter 3 describes the applications model of the SPC application framework and its architecture which is based on the layered model. This chapter also presents the methodology for developing the application framework.

Chapter 4 focuses on the analysis and design phases of development of the SPC application framework. This chapter shows the use case model, analysis model, high-level design model, detailed design model, class hierarchy and design pattern of the SPC application framewrok.

Chapter 5 focuses on the implementation of the application framework. This chapter describes also the customisation of the application framework by using the wizard tools provided.

Chapter 6 is the conclusion, summary and discussion of the SPC application framework which is developed within this research project.

Chapter 2

Literature Review

Reuse of software has been a goal of software engineering for decades. With the emergence of OO techniques, important progress towards achieving this goal was made possible. An OO application framework (OOAF) is essentially the design of a set of objects that collaborate to carry out a set of responsibilities in a particular problem domain. Such frameworks have attracted attention from many researchers and software engineers for their ability to facilitate the reuse of larger components.

The literature review described in this chapter provides the background and foundation of the OO application framework, which is the basis for this research project. The main goal of this chapter is to facilitate the development and analysis of OOAF by presenting their main underlying concepts along with the main advantages and disadvantages associated with their use. Section 2.1 is related to identify the need for software reuse in this context. Section 2.2 briefly introduces the OO paradigm, as frameworks rely heavily on its mechanisms such as inheritance, object composition and polymorphism. In section 2.3, OOAF is introduced with its definition and classifications and some of the existing frameworks are also reviewed in this section. The internal components and the iterative process of building a framework with its main

phases is then described in the next two sections. Section 2.6 points out some consequences of using frameworks, highlighting some of their strengths and weaknesses. The last section briefly discusses the applicability of OOAF to the SPC problem domain.

2.1 Software Reuse

Software is difficult to develop, modify and maintain. Most software are delivered late and over budget. Even in this age of informational and technological advancement, software developers still have to create software systems from scratch due to the lack of reusable components. As such, a software company can have a competitive advantage over its competitors by delivering products faster and being able to adapt quickly to new changes by using systematic reuse strategy (Jacobson, Griss & Jonsson 1997).

It is proven that a higher level of abstraction gives a greater potential for reuse (Coad & Yourdon 1990). There is a consensus that there are many opportunities for reuse that span the entire life cycle of the products in a systems development environment. Reuse is the use of existing artifacts of software development such as design, specifications, code, documentation and test plans (Booch 1994*b*), elsewhere within a project or on other projects. Since software development schedules, estimates and costs are heavily influenced by the amount of new code that has to be designed and developed, if software development is on the critical path of a project, reusing of the existing design, specifications and code can have a significant positive impact on costs and schedules for a project.

A significant portion of the study concentrates on systematic reuse in which organisations design software to be reusable. For example, in two of the divisions of Hewlett Packard (HP) (Lim Wayne C. 1994), reuse is a

critical ingredient in achieving productivity and quality objectives. Available design, specification and code are used several times, the accumulated fixes in each use results in a higher quality product. The reuse technology provides components that can be easily connected to develop a new system. The developer does not have to know how the component is implemented but only needs to know how to use it. The resulting system will be efficient, easy to maintain and more reliable. Most work in software reuse has addressed composition technology (Biggerstaff & Richter 1989), where components are considered to be largely atomic and ideally unchanged when reuse, although some adaptation may be required. The components are the building blocks used in constructing and deriving a new software system.

Component-based software development 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 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. Reuse of components is more productive than reuse of code (Khajenoori & Linton 1994) since reuse of higher-level units involves less application or implementation specific constraints.

Software reuse is one of the most important issues for improving the productivity of software development processes. In the development process of reusable software, there are different starting points for building reusable component, each with emphasis on other activities: based on the generalization of several similar applications, based on the reengineering of legacy

software or based on models (Philippow & Riebisch 2001). All approaches require a smooth cooperation of stakeholders, whose roles vary in deifferent development phases. These roles are domain experts, software engineers and end users.

2.2 Object-Oriented Programming

The main obstacles encountered in traditional procedural programming languages such as difficulty in extending and specialising functionality, difficulty in factoring out common functionality for reuse, barrier to interoperability, maintenance overhead (Taligent Inc. 1998), have forced the software community to look for new approaches to software programming.

The OO paradigm presents new techniques to face the challenge of building large-scale programs. It originates with Simula, which was initially dedicated to solving simulation (model building) problems. Since then, OO technology has been exploited in a wide range of applications including databases, operating systems, distributed computing and user interfaces. The main benefits of the object-oriented approach as presented in Abadi and Cardelli (Abadi & Cardelli 1996) are:

- **The analogy between software models and physical models**. The analogy with a physical system model has proved to be useful in the process of developing a software model. It makes the analysis of the problem more efficient.

- T**he resilience of the software models.** Unlike the approach advocated by procedural languages, which emphasises the use of algorithms and procedures, the design of OO systems emphasises the binding of data structures with the methods to operate on the data. The idea is to design

object classes that correspond to the essential features of a problem. Algorithms, factored in methods and encapsulated in objects, form natural data abstraction boundaries. The main consequence of encapsulation is that it helps focus on modelling the system structure rather than trying to fit a problem to the procedural approach of a computer language.

- **The reusability of the components of the software model.** Objects are naturally organised into hierarchies during analysis, design and implementation and these encourages the reuse of methods and data that are located higher in the hierarchy. Furthermore, this property generates all the other advantages associated with software reuse including low maintenance overhead, high productivity, etc.

One of the main advantages of the OO paradigm is that it promotes the reusability of software components. Researchers (Johnson & Foote 1988) (Johnson & Russo 1991) (Opdyke 1992) have identified attributes of object-oriented languages as described below that promote reusable software:

- **Data abstraction** refers to the property of objects to encapsulate both state and behaviour. The only way to interact with an object and to determine an object state is by its behaviour. Thus, data abstraction encourages modular systems that are easy to understand.

- **Inheritance** is the sharing of attributes between a class and its subclasses (Abadi & Cardelli 1996). It promotes code reuse, since code shared by several classes can be placed in their common superclass to be inherited and reused. The programmer can define in this case a new class by choosing a closely related class as its superclass and describing the difference between the old and new classes. This style of programming is called programming-by-difference (Johnson & Foote 1988).

Another advantage of inheritance is that it provides a way to organise and classify classes, since sibling classes are usually related.

- **Polymorphism** is defined by Booch (Booch 1994*a*) as: "A concept in type-theory, according to which a name (such as variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways." In other words, a method can be invoked on an object without knowing the object's exact type. Because it works with a wider range of attributes, it is easier to reuse a polymorphic method than one that is not polymorphic. For example, the expression a+b will invoke different methods depending upon the class of the object in variable a. Operator "+" in this case is overriden in each class. Polymorphism allows an object to interact with other different objects as long as they have the same interface. It simplifies the definition of client objects, decouples objects from each other and allows them to vary their relationships to each other at runtime (Gamma, Helm, Johnson & Vlissides 1995).

Object-oriented languages have introduced a significant revolution in programming techniques. However, even if the programming job is made easier as the work is performed now at a higher level of abstraction with objects and class libraries, the programmer is still responsible for providing the structure and the flow control of the application. Therefore, reusability is achieved mainly at the class level and only rarely at a higher level (e.g. structural level). OOAF, as it will be seen in the next sections, carry the OO paradigm further by aiming at reusing software at a larger scale.

2.3 Application Frameworks

2.3.1 Framework Definitions and Concepts

Most authors agree that an framework is a reusable software architecture comprising both design and code but no generally accepted definition of a framework and its constituent parts exist. A widely accepted definition of framework comes from Ralph E. Johnson (Johnson & Foote 1988) of the University of Illinois:

"A framework is a set of classes that embodies an abstract design for solutions to a family of related problems and supports reuses at a larger granularity than classes"

The key terms in the definition of framework are:

- **Set of classes**. The set of classes refers to a number of object-oriented classes corresponding to the essential features of a problem domain.

- **Design**. The design of an application defines the overall structure of an application, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control.

- **Abstract design**. An abstract design is a design in which some of the components (classes) are abstract. Abstract classes define methods in terms of a few undefined methods that have to be implemented by the subclasses.

- **Solutions for a family of related problems**. The solutions to a family of related problems usually have common elements. They can belong to particular business units (such as data processing or cellular units) or application domains (such as user interfaces or real-time avionics).

- **Reuse**. Software reuse is the use of existing assets in some form within the software product development process. More than just code, assets are products and by-products of the software development life cycle and include software components, test suites, designs and documentation.

- **Granularity**. Granularity in this definition refers to the level of reuse. Low granularity reuse is the reuse of components while higher granularity reuse refers to design or analysis reuse.

Jacobson (Jacobson et al. 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 a foundation to an application development that provides a domain system with basic architecture of classes packed in software components and design of subsystems. A framework usually defines the overall structure of all the applications derived from it, their partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. The developer is only responsible for customising the framework to a particular application. This consists mainly of extending the abstract classes provided by the framework.

Application frameworks are skeletons that define the basic design of an application. They provide code fragments as the foundation of the application. Moreover, frameworks are much more than mere class libraries. They consist of large sets of code components and include the glue that lets the various classes cooperate. In other words, frameworks are sets of abstract classes with predefined interfaces and problem-adopted event handling. They are more customisable than most components and have a more complex interface. Through the use of interfaces and abstract classes, a framework imposes order and structure to an application. This order and structure allows the developer

to concentrate on solving the mission critical aspects of the project rather than worrying about the glue that holds it all together (Ahamed, Pezewski & Pezewski 2004).

A framework is a partial design and implementation for an application in a given problem domain. The core framework design comprises both abstract and concrete classes in the domain. The concrete classes in the framework are intended to be invisible to the framework user. An abstract class is either intended to invisible to the framework user or intended to be subclassed by the framework user. The core framework design describes the typical software architecture for applications in the domain.

However, the core framework design has to be accompanied with additional classes to be more usable. These additional classes form a number of class libraries, referred to as framework internal increments. Two common categories of internal increments that may be associated with a core framework design are the following:

- Subclass representing common realisation of the concepts captured by the superclasses. For example, an abstract superclass 'Device' may have a number of concrete subclasses that represent real-world devices commonly used in the domain captured by the framework.

- A collection of subclasses or classes representing the specifications for a complete instantiation of the framework in a particular context. For example, a graphical user interface framework may provide a collection of classes for a framework instantiation in the context provided by Windows 95.

The framework can be seen as generative since it is intended to be used as the foundation for the development of a number of applications in the application domain captured by the framework. This is in contrast to the normal way of

developing an object-oriented application. The difference in the development process is outlined in Figure 2.1.

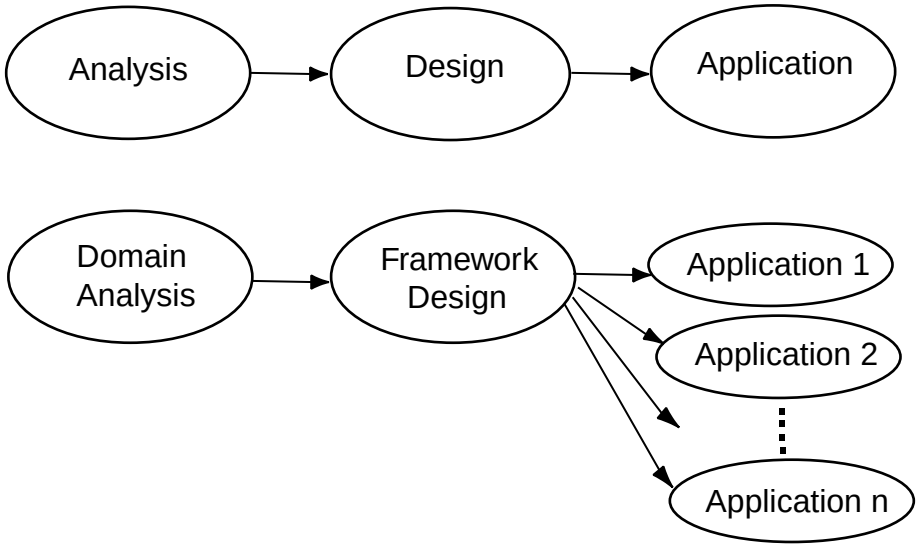

Figure 2.1: Tradition Versus Framework-Based Application Development

Object-Oriented Application Framework (OOAF) represents the scaling up of the fundamental principle of OO programming, which is inheritance and dynamic binding. By having these features, a new class can be implemented by providing only what is different in this compared to one which already exists. With OOAF, the same principle is applied to the whole applications or subsystems, so that the effort to develop a new application is proportional to the difference in functionality between the particular application and that in the framework.

In contrast to traditional approaches to software reuse, which are built on the paradigm of libraries containing a large number of small building-blocks, OOAF allows the highest common abstration level among a number of similar products or applications to be captured in terms of general concepts and structures. This creates a generic design that can be instantiated for each existing product as well as for products to be developed and marketed in the future.

Frameworks are ideally suited for capturing the commonalities in a prod-

uct family. The bulk of the functionality can be captured in the framework, which is maintained as a single system. Each product is an instantiation of the framework, where the amount of unique code is proportional to only the amount of specific features in that product and not to its total complexity.

An OOAF can be seen as a class library which is built on a systematic and extensive use of polymorphism or dynamic binding. OOAF grows out of the observation that inheritance and late-binding polymorphism are powerful abstraction mechanisms, and programs that are expressed in an OO programming language can be reused by incrementally adapting them to different needs. Among the earliest example of OO application frameworks was the Smalltalk Model-View-Controller framework.

2.3.2 Object-oriented Application Framework and Other Concepts

In (Firesmith 1994), the following is used as a definition:

A framework is a significant collection of collaborating classes that capture both small-scale patterns and major mechanisms that implement common requirements and design in a specific application domain.

Based on this definition, an object-oriented framework can be defined as:

A architecture designed for maximum reuse, represented as a collective set of abstract and concrete classes, encapsulated potential behaviour for subclass specialisations.

Based on the definition, there are some differences between object-oriented framework with other concepts such as:

- an object-oriented design pattern
- a pattern language
- a class library

- an ordinary object-oriented application

An object-oriented design pattern differs from a framework in three ways (Gamma et al. 1995):

- The design patterns are more abstract than a framework. Frameworks are embodied in code. This is not the case for design patterns, where only examples of the design patterns are embodied in code. The design patterns also describe the intent, trade-offs and consequences of a design, which is not the case for framework.

- Design patterns are smaller architectures than frameworks. A framework can contain in a number of design patterns, but the opposite is never possible. Thus, the design patterns have no major impact of the application's architecture.

- Frameworks are more specialised than design patterns. Frameworks are always related to a specific application domain, whereas design patterns are general and can be applied in any application domain.

A pattern language differs from frameworks in the way that a pattern language describes how to make a design where an object-oriented framework is a design. Describing in another way, a pattern language instructs us how to do it, while a framework does it for us.

A class library is a set of related classes that provides general-purpose functionality. The functionality typically covered by class libraries are, for example, collection classes (lists, stacks, sets, etc) and IO-handling. The difference between a class library and a framework is the degree of reuse and its impact on the architecture of the application.The framework has a major impact on the architecture of the framework-based application developed. A class in a class library is reused individually and a class in a framework is

reused together with other classses in the framework to solve a specific instance of a certain problem.

An object-oriented application differs from a framework in the way that the application describes a complete executable program that satisfies a requirement specification. This is contrast to the framework that captures the functionality of the application, but is not executable because it does not cover the behaviour in the specific application case.

2.3.3 Classification of Frameworks

There are many types of frameworks on the market, ranging from low-level frameworks that provide basic system software services such as communication, printing, and file systems support, to very specialised high-level frameworks for user interface or multimedia software components. Although the underlying principles are largely independent of the domains to which they are applied, a classification of frameworks by their scope is sometimes useful (Fayad & Schmidt 1997).

System infrastructure frameworks. Their primary use is to simplify the development of portable and efficient system infrastructure including operating systems, communication frameworks and frameworks for user interface. Being used internally within the organisation, they are not typically sold to customers directly.

Middleware integration or support frameworks. Their primary use is to integrate distributed applications and components. They are designed to enhance the ability of software to be modularised, reused and easily extended. Examples of middleware frameworks include object request broker (ORB) frameworks, message-oriented middleware and transactional databases.

Enterprise application or domain frameworks. Their primary use is to support the development of end-user applications and products directly and

therefore represent the base of enterprise business activities. They address different types of applications in a broad application domain such as telecommunications, avionics, manufacturing, education and financial engineering. In spite of the cost of development and/or purchase, enterprise frameworks can provide a substantial return on investment since they support the development of end-user applications and products directly.

Another important classification is to consider the techniques used to extend a framework. From this perspective, frameworks range along a continuum between the two extremes as follows:

White-box or architecture-driving frameworks rely heavily on OO features such as inheritance and dynamic binding. The framework is extended either by inheriting from framework base classes or by overriding pre-defined hook methods (Fayad & Schmidt 1997). A white-box framework defines interfaces for components that can be plugged into it via object composition. However, the difficulty of using white-box frameworks resides in the fact that they require in-depth understanding of the classes to be extended. Another weakness, specific to subclassing in general, is the dependence among methods: e.g. overriding one operation might require overriding another and so on. Subclassing can lead in this case to an explosion of classes.

Black-box or data-driven frameworks are structured using object composition and delegation rather than inheritance. They emphasise dynamic object relationships rather than static class relationships. A new functionality can be added to a framework by composing existing objects in new ways to reflect the behaviour of an application. The user in this case does not have to know the framework in-depth details, but only how to use existing objects and combine them. Black-box frameworks are therefore generally easier to use than white-box frameworks. On the other hand, black-box frameworks are more difficult to develop since their interfaces and hooks have to anticipate a wider

range of potential use cases. Due to their predefined flexibility, black-box frameworks are more rigid in the domain supported. Heavy use of object composition can also make the designs harder to understand. Nevertheless, many framework experts expect an increasing popularity of black-box frameworks, as developers become more familiar with techniques and patterns for factoring out common interfaces and components.

These two categories presented above are extreme cases because in practice a framework hardly ever is pure white-box or black-box, or has only called or calling components. In general, in a framework, inheritance is combined with object composition.

2.3.4 Review of Existing Frameworks

An application framework is a reusable, "semi-complete" application that can be specialised to produce custom applications (Johnson & Foote 1988). In contrast to earlier OO reuse techniques based on class libraries, frameworks are targeted for particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or manufacturing application). Frameworks like MacApp, ET++, Interviews, ACE, Microsoft's MFC and DCOM, JavaSoft's RMI and EJB (Enterprise Java Bean) and implementations of OMG's CORBA play an increasingly important role in contemporary software development. Some of the first frameworks were developed to support GUI applications, include Apple's MacApp and Next's NextStep in the 1980s. Others of the better-known framework projects were Taligent's CommonPoint developed by Apple and IBM, and Microsoft Foundation Classes (MFC) by Microsoft.

2.3.4.1 Model-View-Controller

Model-View-Controller (MVC) is a design pattern for building user interfaces developed using the Smalltalk programming environment. The MVC paradigm is a way of breaking an application or even a piece of an application interface into three parts: the model, the view, and the controller. The model is the underlying logical representation, the view is the visual representation, and the controller specifies how to handle the user's input. When a model changes, it notifies all views that depend on it. A view uses a controller to specify its response mechanism. For instance, the controller determines what action to take when receiving input from the keyboard. This separation of state and presentation allows for two very powerful features.

1. Multiple views based on the same model is consistent. For instance, a set of data can be presented in both table form and chart form. As the data model is being updated, the model notifies both views and gives each an opportunity to update itself.

2. Modification or creation of the views can be done without affecting the underlying model since models specify nothing about presentation.

2.3.4.2 Common Object Request Broker Architecture

Common Object Request Broker Architecture (CORBA) is the standard distributed object framework developed by the Object Management Group (OMG) consortium. It is a common object request broker architecture. ORB (Object Request Broker) is the middleware that establishes the client-server relationships between objects.

Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results.

The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

In ORB or CORBA, a client is defined as any code which invokes an operation on a distributed object, while server is a program that contains the implementations of one or more IDL (interface definition language) interfaces.

2.3.4.3 Visual Component Framework

The Visual Component Framework (VCF) is a C++ framework that is created to provide a simple-to-use cross platform GUI framework, with many of the advanced design features of Java and Java's Swing, and Borland's Visual Component Library. The framework is divided into three main sections: Foundation Kit, Graphics Kit, and Application Kit.

The Foundation Kit provides all of the base classes for the rest of the framework, as well as support for events, event listeners, properties, and basic streaming capabilities. The VCF makes use of C++'s Standard Template Library for its collection classes, and has a number of template based classes for the reuse purposes. The Graphics Kit provides a core set of easy-to-use 2D graphics classes, modelled heavily after the design of Java's Graphics2D architecture. The Application Kit provides the classes used in the GUI portion of the framework. This includes basics like windows, components, standard widgets (combo boxes, list boxes, trees, etc), common dialogs, a basic layout manager, drag-and-drop, and standard windowing events.

2.3.4.4 Verifiable Embedded Real-Time Application Framework

Verifiable Embedded Real-Time Application Framework (VERTAF) is proposed for embedded real-time application development, with the aim of reducing design errors and increasing design productivity (Hsiung, Lee, See, Fu & Chen 2002). The development of VERTAF is based on the integration of object-oriented technology, software component technology and formal verification technology. This framework consists of five basic software components: *Implanter*, *Modeler*, *Scheduler*, *Verifier* and *Generator*. When a software application is designed, objects specific to the application are identified. The real-time and embedding constraints are specified within these application objects. These application objects are then transformed into standard uniform process models. Based on these models, the application codes are generated. Different level of reuse, including object-level and component-level, increased design productivity and decreased overall design effort and time.

2.4 Framework Components

Conceptually, most of the authors (Demeyer, Meijler, Nierstrasz & Steyaert 1997) consider that there are two main components of a framework:

- **Framework contracts**. The common functionality in a specific domain is captured in the framework contracts. They formalise exactly which parts of the framework are to be reused. Thus, framework contracts impose a common structure for all the applications that use the same framework. The implementation and functionality of contracts are usually hidden from the user. However, because they form the skeleton of all applications, they have a direct impact on the performance and correctness of an application.

- **Hot spots**. A variable aspect of an application domain is called a hot spot (Schmid 1997). A framework is tailored for a specific application by implementing its hot spots according to the specific functionality of the application. Thus, different applications will differ from each other with regard to at least one hot spot. A hot spot allows a user to insert an application-specific class or subsystem. This can be done either by selecting the class or subsystem from a set of predefined classes supplied with a black-box framework, or by extending the abstract classes as in a white-box framework case as shown in Figure 2.2. In a black-box, the user has only to choose a class or a subsystem from the set supplied by the framework while in a white-box system the user has to actually build the class or the subsystem to be used by the framework.

Hot spots provide the mechanism for extending a framework and therefore their design has a big impact in the usability of the framework in general, especially in its flexibility and variability.

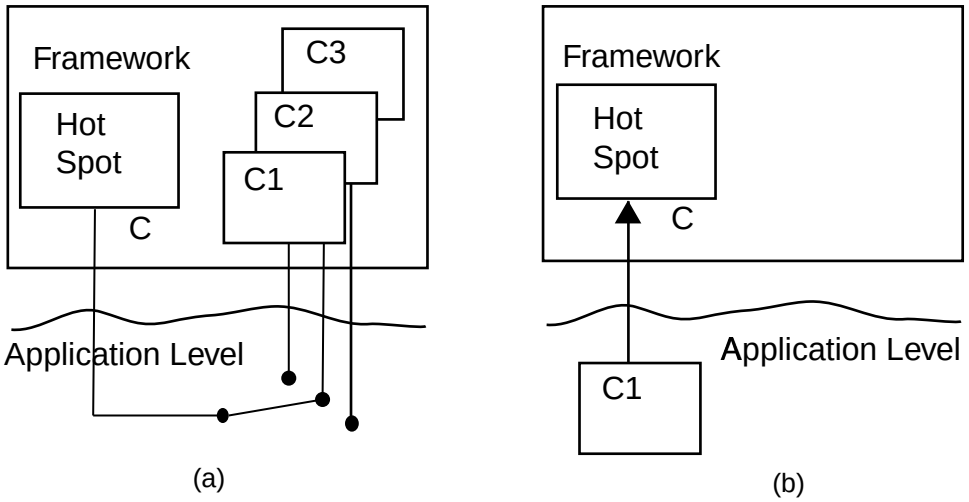

Figure 2.2: The Hot-Spot Mechanism: (a) in a Black-Box (b) in a White-Box

2.4.1 Hot-Spots

Hot spots have a big impact in the reusability and flexibility of a framework and therefore it is worthwhile to explore them in more details. Schmid (Schmid 1997) suggests that the variability required from a hot spot can be classified by the following characteristics:

- The common responsibility (C in Figure 2.2) that generalises the different alternatives.
- The different alternatives that realise C.
- The kind of variability required. This variability can be considered for example in alternatives with a common interface but different implementations, or alternatives with uniform service over different structures and so on.
- The multiplicity that gives the number and structuring of the alternatives that may be bound to a hot spot. It is directly related to the previous characteristic in the sense that usually the kind of variability dictates the number and structure of the alternatives.
- The binding time represents the point of time at which an alternative is selected. This time is either the time of creating an application or the run time. In the first case the application developer realises the binding while in the second case it is the end user responsibility to do it either once or repeatedly.

Structurally, a hot spot is typically composed of a base class and a number of subclasses as illustrated in Figure 2.3:

- An abstract base class, which defines the interface for common responsibilities.

- Concrete derived classes, which implement application specific alternatives.
- Possibly additional classes and relationships.

In the case of a white box framework, the application programmer has to implement the derived concrete classes. In contrast, a black box framework provides all these concrete classes and the user is responsible for choosing the appropriate ones and combining them to obtain the functionality required by the application.

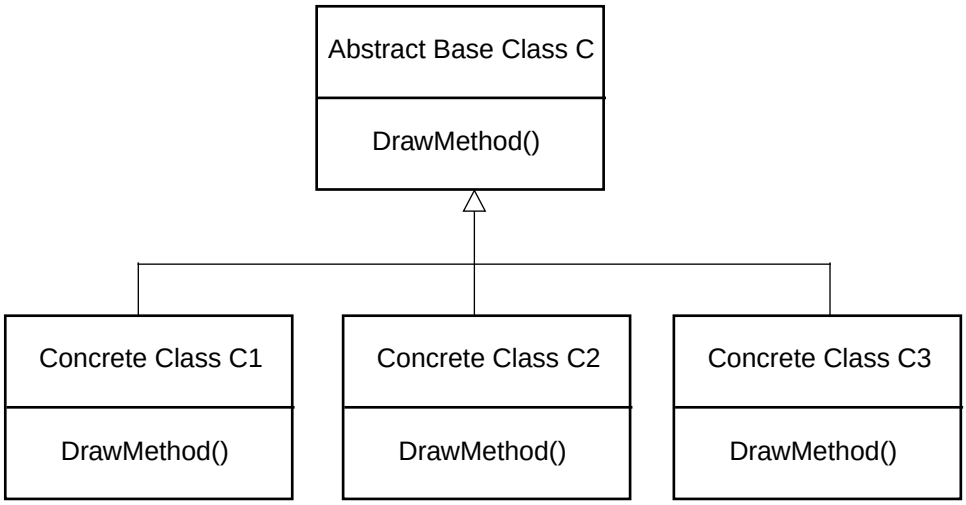

Figure 2.3: Example of Hot Spot Subsystem.

A hot spot usually contains a polymorphic reference typed with the base class. The user binds the hot spot by setting the reference to a subclass object of the base class. This object can be from a prefabricated set supplied with the framework in black-box case or can be built by the application developer by extending the base class. Methods in the base class are specialised in a child class and every call to such a method will be dynamically bound, via its reference, to the subclass method executed. Therefore, a hot spot subsystem introduces variability that is usually transparent to the remainder of the framework.

2.4.2 Abstract Classes

In a framework, the major design issues are made explicitly by means of *abstract classes*. Abstract classes form the skeleton of an OOAF. Furthermore the framework must be reused in such a way that the major design choices are respected. In general terms, an abstract class is a class that is only partially implemented. Before making use of the abstract class it must be made concrete by implementing the missing details. Conventionally only classes with abstract methods are called abstract class.

An abstract class is actually a class that implements one set of methods, called the *template methods* and another set of unimplemented methods, called *abstract methods* (or *virtual methods*). Instance of classes with abstract methods cannot be used, since their implementation is incomplete. The different kinds of methods can be defined as below.

- An abstract method is a method that has no implementation, and is formally declared as such.
- An template method is a method that has an implementation but that calls either directly or indirectly an abstract method. Thus, a method that calls another template method is itself a template method, since it will indirectly call an abstract method.
- A concrete method is a method that has an implementation and that does not rely on abstract or template methods.

An abstract method can be made concrete in a subclass by overriding it with a concrete method. The template methods are reused through inheritance.

2.5 Different Aspects of Reuse

There are several aspects of reuse strategy that need to be considered (Crnkovic & Larsson 2000).

2.5.1 Component Generality and Efficiency

Reusable components of an application framework must be sufficiently general to cover the different aspects of their use. At the same time, they must be concrete and simple enough to server a particular requirement in an efficient way. Developing a reusable component requires three to four rimes more resources than developing a component, which serves a particular case. The fact that the requirements of the components are usually incomplete and not well understood brings an additional level of complexity. In general, requirements for generality and efficiency at the same time lead to the implementation of several variants of components which can be used on a different abstraction level.

2.5.2 Evolution of Functional Requirements

The development of reusable components would be easier if the functional requirements did not evolve during the time of the development. New requirements for the components need to be redefined if there are new requirements for the products. The more reusable a component is, the more demands are placed on it.

2.5.3 Migration Between Different Platforms

The reason for the migration of platform can be customer requirement change to run the product in a specific platform or general trends in the growing popularity of certain operating systems which might support newer, better or

cheaper hardware. As an important part of the reuse concept was to keep the high-level components unchanged and encapsulate the differences between operating systems in low-level components.

2.5.4 Compatibility

A component can be replaced easily or added in new parts of a system if it is compatible with its previous version. This is an important factor for successful reusability. Compatibility issues are relatively simple when changes introduced in the products are of maintenance and improvement nature. More complicated problems occur when new changes introduced in a reusable component eliminate the compatibility, such as, there is a need to use an additional software to manage both versions.

2.6 Advantages and Drawbacks of Using Frameworks

2.6.1 Advantages

The popularity of frameworks is justified by their strengths in enabling the reuse of software components, design and analysis.

The main advantage of frameworks is that they capture the programming expertise necessary to solve a particular class of problems. Programmers use frameworks to obtain such problem-solving expertise without having to develop it independently. Frameworks allow reuse not only at the coding level but also, more important, at the design level and even at the analysis level (Johnson & Russo 1991).

Once a framework is understood, the development time for an application should drop. Besides, the modelling and the design are easier since frame-

works offer guidance in developing the application. Besides, it is easier to maintain several applications built on top of a framework than the same applications built independently. This is a consequence of the fact that a framework imposes a similar design structure and functionality on all its applications in contrast with the design of the independent applications.

As the framework is used, it is also tested more often and more bugs and errors are reported and solved. This makes applications built using stable frameworks more reliable compared to those developed from scratch. Moreover, expertise is embodied in a framework, problems are solved once and the solution is used consistently. This enables software developers to concentrate on their particular problem domain and rely on the framework to provide consistent services.

2.6.2 Drawbacks

There are a number of challenges that must be addressed in order to employ frameworks effectively (Fayad & Schmidt 1997):

Developing reusable frameworks for complex application domains is a very difficult task. Very often only expert developers posses the skills required to produce frameworks successfully. The learning curve to use an OO framework typically requires more effort depending on the complexity of the framework. Hands-on mentoring and training courses are required for improving the learning process.

As applications become more and more complex, they will be increasingly based on the integration of multiple frameworks, class libraries and existing components. Since many earlier generation frameworks were designed only for a specific problem domain, difficulties may be encountered in integration with other frameworks specialised for other domains.

Mechanisms that frameworks rely on, such as dynamic binding, employ

additional levels of indirection. The use of dynamic binding improves the generality and flexibility of the framework but reduces efficiency. This also applies to other OO specific operations such as dynamic creation and deletion of objects, virtual methods, etc.

Currently the lack of accepted standards for designing, implementing, documenting, and adapting frameworks impedes them from being truly effective across multiple application domains. A framework developed in one language cannot be used for applications that use other languages. It is important for companies and developers to work with standards organisations and middleware vendors to ensure that the emerging specifications support true interoperability and define features that meet their software needs.

2.7 Concluding Remarks

Object-oriented application framework presents a proven software design and implementation to develop reusable assets in software industry. The common problems and core domain concepts for statistical process control in manufacturing domain are captured in the development of this application framework. This helps to reduce the software development time. Software developers can focus on solving the business domain problems instead of programming problems concerning the architecture and design foundation.

SPC in manufacturing domain consists of much commonality and variability. Moreover, this domain is a complex domain and the environment is rapidly changing over time. It is a suitable domain for application framework development.

Chapter 3

An Architectural View of Object-Oriented SPC Application Framework

Modern large-scale information systems are very complex, impacted by constantly changing standards, and combine distributed computing, many systems and platform. This scenario cannot be avoided but can be managed with a good software architecture. Software architecture affects how software is designed and structured and therefore substantially influences the characteristics of the software systems. Choosing the right architecture is one of the most important decisions a software engineering business can make. Software architecture is important to maintain the integrity of system. This is to avoid patchwork of uncoordinated fixes in the software development and maintenance. A well-articulated software architecture is also key to managing the complexity of software systems, allowing large organisations to work on parts in parellel. A good software architecture allows the component and application systems to evolve gracefully over time. The architecture must be defined and described in such a way that it can be easily modified and improved as

new and changed requirements are implemented.

3.1 Layered Architecture

Layered architecture is a kind of software architecture that organises software in layers, where each layer is built on top of another more general layer. A layer can be defined as a set of subsystems or systems with the same degree of generality. Upper layers are more application specific, lower are more general. One of the advantages of the layered architecture is that it divides the complexity of different types of software components into manageable and relevant layers. This type of architecture allows software to be organised in layers according to application specificity. The further down the layers, the more general the componets. The software engineering processes can be then customised to develop and evolve components with different levels of specificity.

The object-oriented SPC application framework is built based on the layered software system as shown in Figure 3.1, which is adapted from (Jacobson et al. 1997). Each layer in the system is constructed from an apporopriate set of components drawn from several component systems in lower layers. Application systems can be built from component systems. Component systems may in turn be built from other component systems in lower layers. This layered system is composed of application systems at the top and component systems underneath. The design is made in such a way that the application framework can be built from the lower layers which are well constructed for reuse.

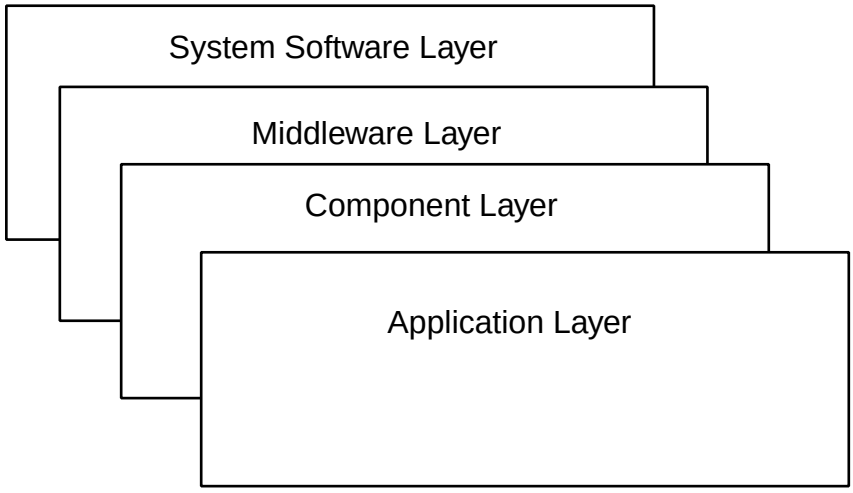

Figure 3.1: A Layered Architecture System

3.1.1 Application Systems

In the layered architecture, the first layer is called application layer. This layer contains different types of application systems for a specific types of domain. This application system is the final software system that one user can use and feel it directly. This application system offers a coherent set of use cases to end user. Application systems may interoperate directly or indirectly through some interfaces or services provided by the system in the lower layers.

3.1.2 Component Systems

The next layer, component systems layer contains the component systems that offer the use cases and object components to the developers to build application systems. This layer is the layer to be reused to develop a specific application system based on the reuse business's needs. Due to this reason, the component systems in this layer are made to be generic. Application developers can reuse these generic components to build a specific application by extending or modifying the generic components which are provided by the component systems. This layer is built on top of the lower layer, which is middleware layer.

3.1.3 Middleware Layer

While objects and components defined in a traditional programming language, such as C++ which provides capabilities, these are limited to executing in one fixed environment and bound to a language. The emergence of standardised platforms for interoperable distributed object computing is an important step towards globalised enterprise information systems.

The middleware layer offers component systems which are providing utility classes and platform independent services such as distributed object computing in heterogeneous environment. Examples to these middleware component systems are CORBA and Java. This layer also contains component systems for interfaces to database management systems, operating system services, etc. These components are used by application developers so that they can focus to build business specific components and application systems.

3.1.4 System Software Layer

The bottom layer is system software layer. This layer contains the software for the computing and networking infrastructure, such as operating system and interfaces to specifc hardware. Well-known operating systems are Windows, Linux, Unix and Macintosh.

3.2 Statistical Process Control Application Layered Architecture

The layered architecture of the SPC application framework contains four layers, which are the Application Layer, Component Layer, Middleware Layer and System Software Layer. All component systems in each layer of the layered architecture are interoperating with each other within the same layer and

with some dependencies between different layers. In the application layer, the application systems that can be constructed from the framework are known beforehand. This layer offers a coherent set of applications, which can be developed from the application framework. In the component layer, the component systems that are needed to construct a particular application system are also known beforehand. The middleware layer contains component systems for interfaces to database management systems, platform-independent operating system services and distribution service. The system software layer contains the software for the computing and networking infrastructure, such as operating systems and TCP/IP networking protocol.

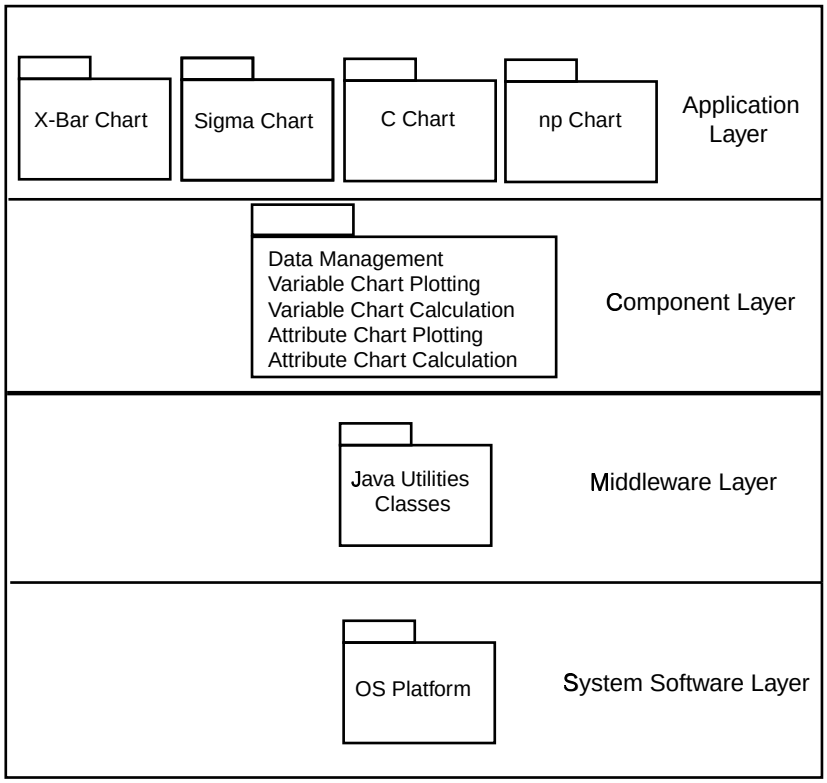

Figure 3.2: Statistical Process Control Application Framework Layered Architecture

3.2.1 Application Layer of SPC Application Framework

Statistical Process Control consists several measurement application systems that are widely used to appraising and monitoring quality performance and are the key ingredients to successful applications of quality control. Quality control relies on the continuous monitoring of quality of the input and output of the processes producing products and services. For example, a control chart is used to monitor the quality of the products in the production line. The application systems consist of several types of control charts such as *x-bar chart*, *sigma chart*, *range chart*, *p chart*, *np chart* and *c chart*, which can be categoried as variable control chart and attribute control chart respectively. These application systems can be built from the lower layer of the layered architecture. For instance, to build the attribute control chart application system, the components in the lower layer can be imported and customised. Each type of the control chart contains the same common functionalities such as data management, chart plotting management and chart calculation management as shown in Figure 3.2.

3.2.2 Component Layer of SPC Application Framework

This is the most important part of the layered system to enable the reusability of the application framework to happen. Basically this layer provides several generic components that can be reused to build the specific SPC application systems in the upper layer. These components can interoperate with each other in order to form the required infrastructure of the SPC application framework. This layer provides some general and common behaviours for the application systems to reuse without developing the system from scratch. While variation points are captured to identify the potential place to be customised to suit the specific requirements.

3.2.3 Middleware Layer of SPC Application Framework

The middleware layer of SPC application framework uses Java as the middleware system. Java provides several useful utility classes and service packages that are being used for this application framework across the domain, for example, Swing, AWT, which provides the utility classes for Graphic User Interfaces (GUI), while Remote Method Innovation (RMI) provides the server and client communication facilities. This enables the application framework to become a distributed and multi-tier system.

3.2.4 System Software Layer of SPC Application Framework

Java programming language is used for the implementation of SPC application framework. Since Java is a platform independent programming language, this application framework can be deployed in several platforms such as Windows, Linux and Unix, provided that particular system has the Java Virtual Machine (JVM) installed. TCP/IP network protocol is used to handle the communication between hardware and the application frmework.

3.3 Development Methodology

The methodology used in this project is based the integration of some existing concepts of Object-Oriented Software Engineering (OOSE) modelling notation and method, which makes it easy to compose reusable components (Jacobson et al. 1997, Lam 1997, Bellinzona, Fugini & Pernici 1995, Chan & Lammers 1997). It is a systematic technique to establish the "right" system architecture to help managers and engineers control the complexity of the system. The processes in the project can be grouped into two major cate-

gories: Domain Engineering (DE) and Application Engineering (AE). DE can be divided into Application Architecture Engineering (AAE) and Component Engineering (CE), each is shown as an ellipse in Figure 3.3. Application Architecture Engineering (AAE) is a process to determine from the company and end users, the way to decompose the overall standardised set of applications into a suite of application systems and supporting component systems. The process architects the Layered Architecture System which consists of facades, interfaces of the subsystems and component systems that support the complete architecture of related applications. Component Engineering (CE) is a process to design and construct reusable components and package them into component systems. The sources of this process are the results from the AAE and requirements of the company and end users. The goal is a consistent model that explicitly expresses commonality and variability across the suite of applications that will reuse these components.

AAE starts from the study of domains selection through to framework implementation and as the way to decompose the overall standardised set of applications into a suite of application systems and supporting component systems as shown in Figure 3.4. The process architects the Layered Architecture System which consists of the interfaces of the subsystems and component systems that support the complete architecture of related applications. There are several iterative stages of the process described as follows:

- Study of domains selection: Conduct a study for some selected domains and select one that is critical to the success of the research. The manufacturing domain is selected due to its complexity and diversity in applications, which is believed a good candidate for the research.

- Survey of existing domain systems: The survey lays the foundation of knowledge about the domain. In the survey, the requirements of

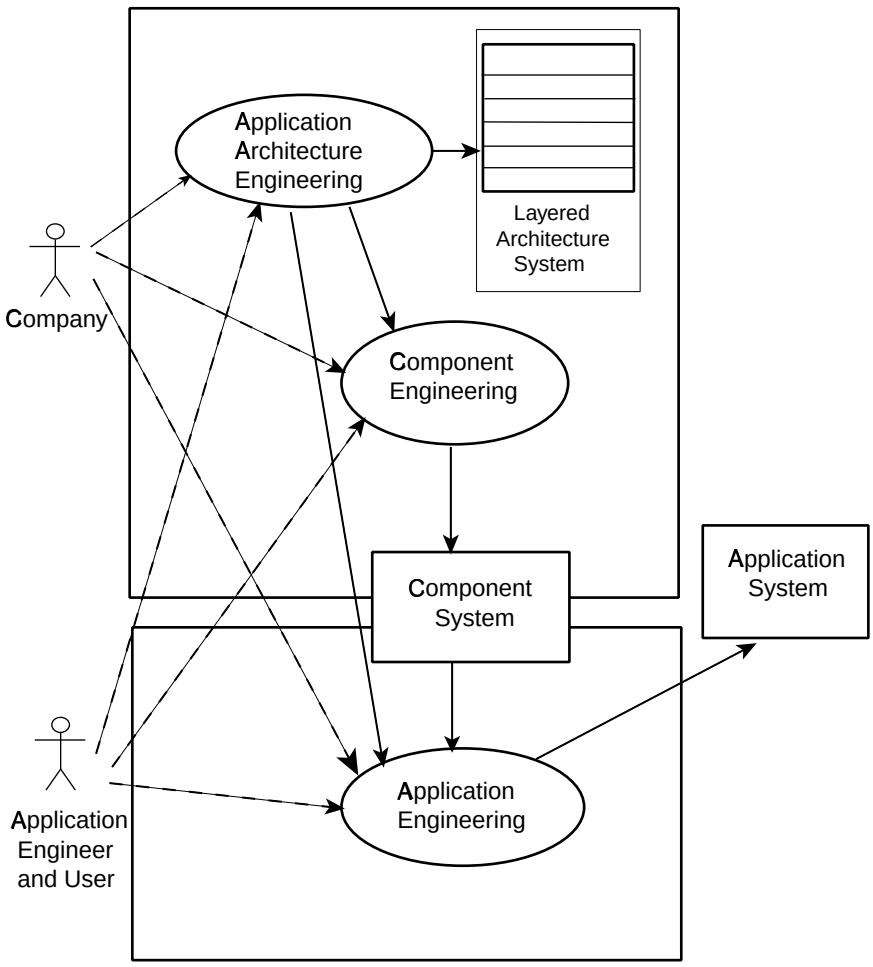

Figure 3.3: The Process Flow of the Project

each domain system's external actors, business processes and models are captured. Functionalities of the manufacturing domain are identified in terms of variable functionality and non-variable functionality. Non-variable functionality provides good candidates for generic component systems. The main aim is to identify the candidates for application and components systems for constructing high-level domain models.

- Define high-level domain models: Generic domain models on the selected domain are defined. These models are defined and updated during the AAE process for enhancement and to reflect new information represented.

- Identify patterns and components: Patterns and generic components are identified across the domain to provide a high level abstraction for the domain models. The high-level domain models serve as the foundation for producing a prototype design model that defines the layered architecture system in terms of application systems and component systems.

- Define design standards: Design standards are defined to provide guidelines on the definition of classes, inheritance hierarchies and rules for defining structural relationships between classes and components.

- Define domain object models: Static and dynamic domain models are defined by doing a detailed study on the domain models generated from the previous phase. Both static (structural) and dynamic (behavioural) aspects of the domain are modelled in detail.

- Coherence checking and evaluation: Coherence checking and evaluation will be done on the static domain model and dynamic domain model to identify any inconsistency and inefficiency.

- Framework detailed design: Develop a detailed design and architecture of the associated application framework by taking into account its implementation in a client-server environment. The detailed design model, which contains several design patterns as well as a library of class inheritance hierarchy used by various component systems, is built based on the layered architecture system. The library includes concrete classes and abstract classes with variation points. Most attributes and methods of each class are also defined in this phase.

- Framework implementation: The application and component systems will be implemented as defined in the detailed design of the layered architecture system.

- Testing, integration and evaluation: To discover any errors or problems in the implementation as a whole.

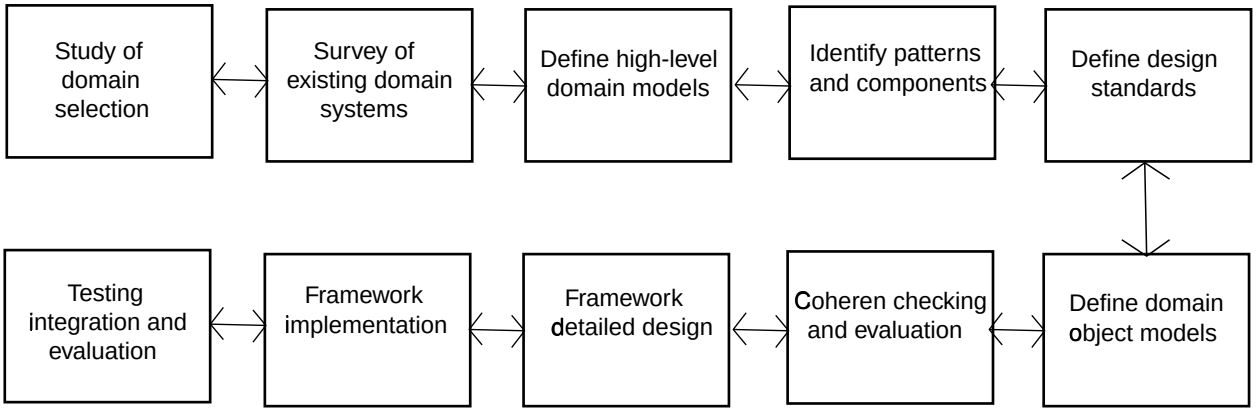

Figure 3.4: Stages of Application Architecture Engineering (AAE) Process

3.4 Summary

Software architecture defines the static organisation of software into subsystems, interconnected through interfaces, and defines how those nodes interact. A software architecture conforms to one or several styles. The subsystems

are partitioned according to generality into groups called layers. These are arranged on a page with the layers running from top to bottom, with the most specific one at the top and the most general at the bottom. An ironclad rule is that a layer cannot use anything above that layer. This guarantees loose coupling. The result is that a subsystem in one layer can be changed or even removed without impacting lower layers. Since the lower a layer is, the harder it is to change and the more dependencies it is involved in.

The statistical process control application framework is organised in the layered architecture, which contains four layers. They are application layer, component layer, middleware layer and system software layer. In the application layer an application system is a system delivered to customers outside of the reuse business. While in the component layer, a component system provides a set of compatible components used to engineer another application or component system. The middleware layer provides the utility classes and services such as distribution facility, interfaces to database management system, etc. The last layer, system software layer which contains the services for the computing and networking infrastructure and interfaces to specific hardware devices.

By using this layer architecture the benefits gain are lower life cycle cost, faster time to market, more robust and lower defects software is developed.

The methodology used in the research is referred to as Application Architecture Engineering (AAE). There are several iterative stages of the process. AAE starts from study of domains selection, survey of existing domain systems, define high-level domain models, identify patterns and components, define design standards, define domain object models, coherence checking and evaluation framework detailed design, framework implementation and through tesing, integration and evaluation.

Chapter 4

Analysis and Design of Object-Oriented SPC Application Framework

This research is based on an Object-Oriented Domain Engineering (OODE) method to be defined within the application framework of the project. The OODE development life cycle contains several main phases. These phases are OO Domain Analysis (OODA), OO Domain Design (OODD) and OO Application Framework Implementation. The various activities or processes of the OODE method will finally lead to a generic domain model which spans a general application, and which an OO application framework will be developed for the rapid assembly of applications. The OODE development is iterative and incremental.

In this research, Unified Modelling Language (UML) which can be used to analyse and design object-oriented software developement requirements is being used. Several types of diagrams such as use case diagram, class diagram, collaboration diagram, are being used to show the overview and the activities of the entire framework in the SPC domain.

4.1 OODA on SPC Domain Model

OODA is the process of discovering and recording the commonalities and variabilities of the system in the domain. In the first phase of the OODE life cycle, i.e. OO Domain Analysis, processes are carried out either sequentially or in an iterative manner. A study has been conducted for the selection of appropriate domains that are critical to the success of the project (Ahamed et al. 2004). Besides this, study on existing related OO domain systems concerning their external actors and processes as well as their structural and behavioral aspects has been conducted as well. The following activity is on defining generic domain models with object-orientation on the selected domain. The model is refined and updated during the OODE process for enhancement and to reflect new information represented. In this stage, patterns and other generic components will be identified across the domain in order to provide a higher level of abstraction for the domain models.

OODA includes the process to create a set of reusable components, which is being used in the development of systems in the domain. The SPC domain has activities of plotting chart or statistical calculation for different types of measurement, it can be variable or attribute measurement, depending on the usage. These activities are common across the systems and can be implementated as framework components.

4.1.1 Domain Analysis of SPC

Quality control relies on the continual monitoring of the inputs and output of the processes producing the products and services. SPC is the application of statistical techniques to ensure satisfactory quality as shown in Figure 4.1.

The input of the system is various types of quality specification and process metrics needed by the organisation. The basic procedure for determining

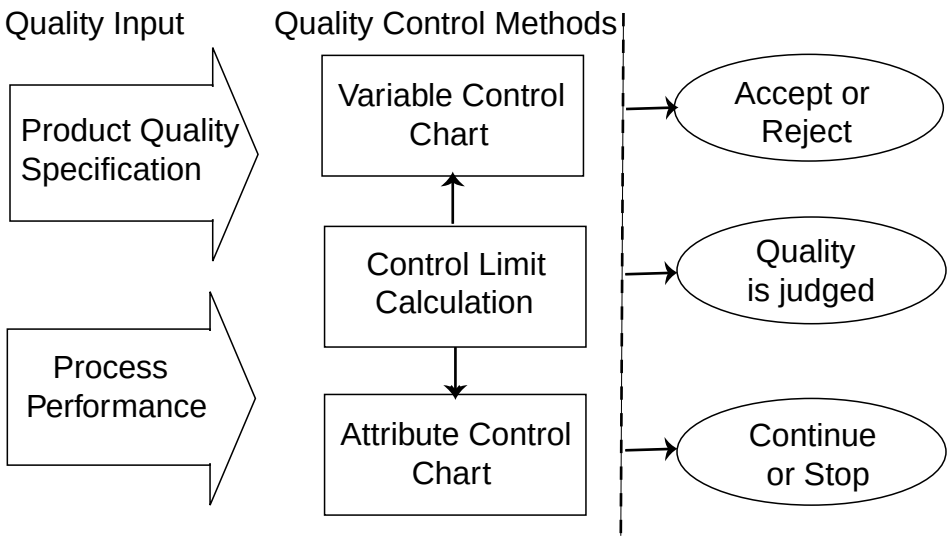

Figure 4.1: The Basic Structure of a Statistical Process Control System

whether a process is in control is, firstly, a random sample is taken from the product or service, then the quality characteristic is measured. If the sample measurement is found to be outside the *upper control limit* (UCL) or the *lower control limit* (LCL), the process is checked to determine the assignable cause of variation. By using the methods prvided by SPC such as control chart, the quality of the product is judged.

In SPC, tools called control charts are used primarily to prevent or detect production of defective products (finished goods, assemblies, components) or service. The primary objective is to make timely decisions about whether or not a particular process is in statistical control. In any process there will be variation from one measurement or unit produced to the next. If the process is out of statistical control, the process is stopped and inspection is carried out to find the root cause. There are two basic types of control charts, control charts for variables and control charts for attributes. Variable control charts are used to control a measurable characteristic, whereas attributes control charts control a countable characteristic.

The development of the object-oriented application framework is based on the common application systems of the SPC, which is widely used in quality

control in the manufacturing. The construction of the control chart application system basically involves implementing a reusable statistical process control component in the framework for the purpose of adaption or configuration. A generic statistical process control component will be provided by the application framework to support a variety of control charts such as *R-chart*, *X-chart*, *n-chart*, etc, in one of its variability points meant for adaptation to the specific needs of the reuser.

4.1.2 Use Case Modelling of SPC Application Framework

Use case modelling is used to capture reusable requirements of the SPC system. By using a use case model, the transition of the domain knowledge from a domain expert to a software architect become easier and simple. The use case diagram provides a high-level abstraction of the overview of the entire system. This is needed before one can start to define the design and details of the classes. Usually use case modelling is the first step to take in object modelling activities and will start off the software developement work.

A use case model defines the features or functionalities of a system should provide for the users. In the use case model, users of the system are called actors. A distinct role to interact with the system is assigned to each actor. Each way an actor uses the system is a distinct use case. Each use case defines a set of interactions with the system. The use case model is used as an important input to the later phases of modelling such as analysis and design models, which describe how the system is architected and designed.

The use case component model in SPC application framework is shown in Figure 4.2 and Figure 4.3. All of the use cases shown are generic use cases for SPC application framework. The use case model can be divided into four modules, which are variable chart calculation, variable chart plotting, attribute chart calculation and attribute chart plotting. There are three generic use cases

in variable chart calculation module. They are *variable inspection data handling*, *variable basic statistic calculation* and *variable chart limit calculation*. The actor of the use cases is *quality controller. Variable inspection data handling* is to capture and format the input data from the actor. The formatted data is calculated by various types of statistical processes and methods by *variable basic statistical calculation* use case. *Variable chart limit calculation* is to handle the calculated and formatted data from previous steps and performs the control limit calculation. The calculation is based on several control chart statistical algorithms. This data is needed for the plotting of the selection of control chart, such as *X-Bar chart*.

The variable chart plotting module is mainly used to plot the types of control chart by the selection of the user. In this module, there are two generic use cases. They are *variable chart data handling* and *variable chart plotting*. The actor involved here is called *plotter*. *Variable chart data handling* use case is to handle the input data of the graph plotting and display the output data to the graph. While *variable chart plotting* will display out the data in the control chart with the calculated control limits (upper control limit, centerline and lower control limit) as a reference to indicate the extremes of the acceptable behaviour of a process. This chart is very useful to monitor the quality control of a process in the aspect of process average, range and standard deviation to control the process variability or dispersion in the manufacturing process.

The attribute chart calculation module consists of three use cases. They are *nonconforming data handling*, *fraction nonforming calculation and attribute chart limit calculation.* The actor in this module is called *quality controller*. Nonconforming data handling use case is needed to prepare an interface to handle the raw data input by the user and make the data ready for *fraction nonconforming calculation* use case to calculate. *Attribute chart limit calculation* use case will do the calculation for the control limit data which is

used to plot graphs.

The attribute chart plotting module has two use cases, *attribute chart data handling* and *attribute chart plotting*. The actor for this use case module is called *plotter*. *Attribute chart data handling* use case is to handle the data and format it to be ready to plot. *Attribute chart plotting* use case plots the data into a graph by using previous calculated data.

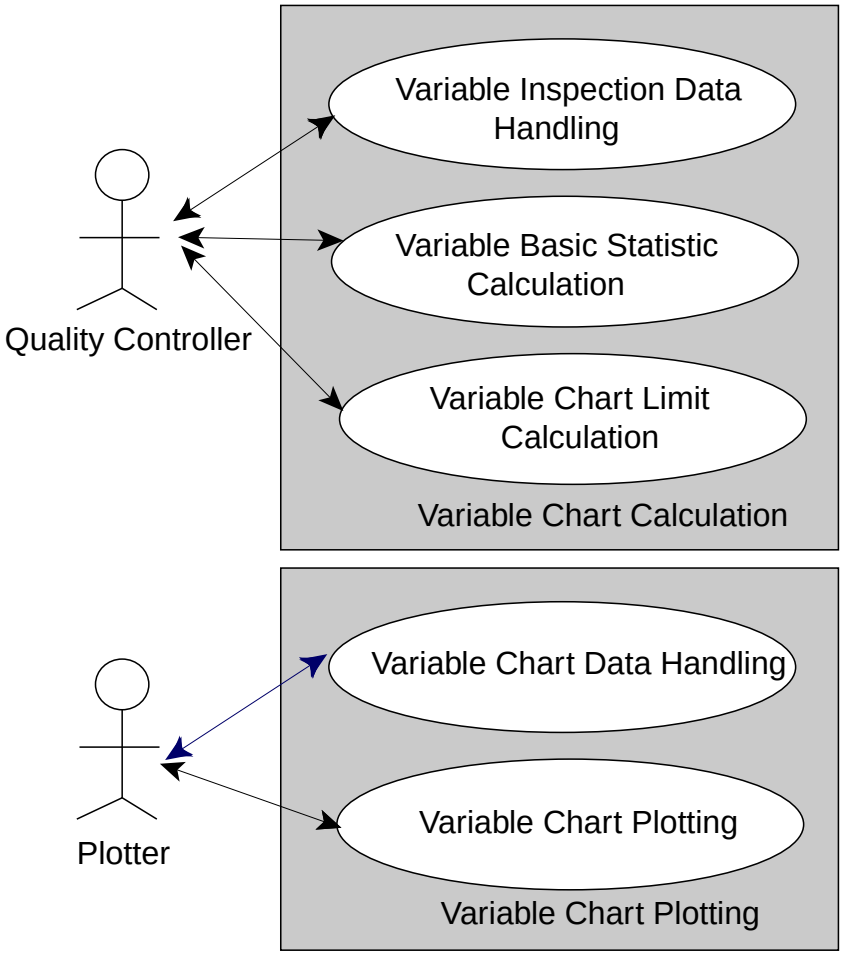

Figure 4.2: Use Case Component Model of Statistical Process Control Application Framework for Variable Control Chart

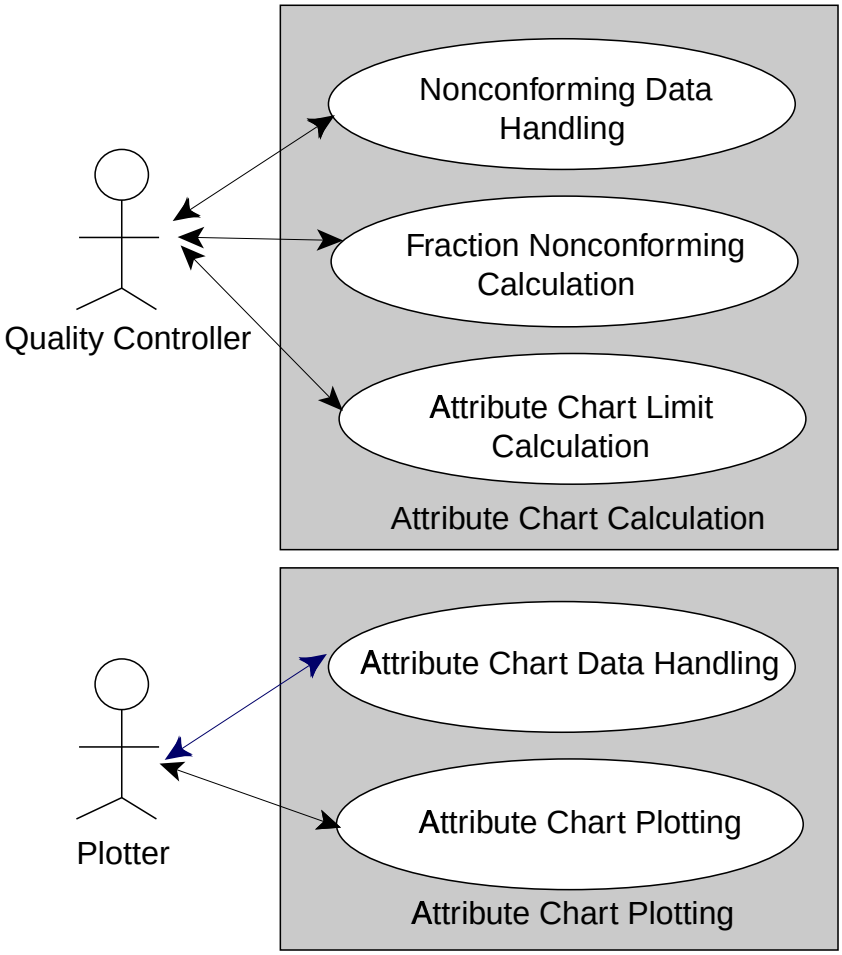

Figure 4.3: Use Case Component Model of Statistical Process Control Application Framework for Attribute Control Chart

4.1.3 Analysis Component Modelling of SPC Application Framework

The analysis component model is a high-level abstraction of the system implementation. Analysis objects interact in performing the use case instances. Each object plays one or more roles in at least one use case instance. The objects that perform the use case instance are presented in collaboration or sequence diagrams which are prepared for each individual use case, or for each group of use case.

The analysis component model of SPC Application Framework is presented using the boundary-control-entity (BCE) diagram as shown in Figure 4.4. There are several kinds of analysis type stereotypes: boundary, entity control classes. The boundary component handles the communication between the system and its surrounding. The control component performs use case specific behaviour. The entity component is in general a long-lived object in the system. Entity types are generic and often used to model business objects, which are dealt within many use cases. The variation point are used to define varying responsibilities in the analysis component model. This is the point where extension and customisation can be done in the component system.

The analysis component model of SPC Application Framewrok is developed from the use case component model in the previous section. The analysis model is divided into variable chart calculation, variable chart plotting, attribute chart calculation and attribute chart plotting modules as shown in Figure 4.5, Figure 4.6, Figure 4.7 and Figure 4.8. The variation point shown in the analysis model is the place for the object component to be extended and customised. For example, *variable centerline controller* control component can be customised to different logic thus create completely different types of

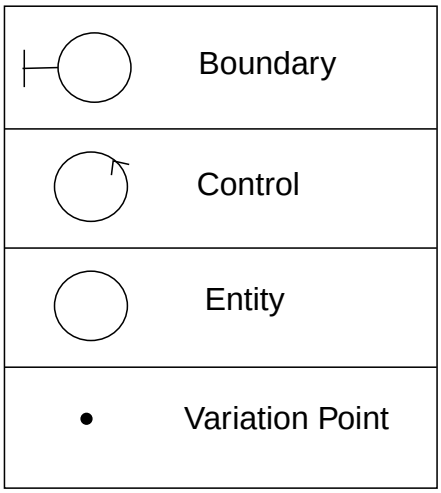

Figure 4.4: BCE Diagram used in Analysis Model

calculation and algorithms for the variable chart calculation module.

The mapping of the analysis component model is per use case component model. For instance, *Variable Chart Plotting* use case of the variable chart plotting module can be traced to the analysis component model as shown in Figure 4.9.

4.2 Object-Oriented Domain Design on SPC Domain Model

The following phase of the OODE life cycle is started by defining OODE design standards to provide guidelines on the definition of classes, inheritance hiearchies and rules for defining structural relationships between classes.

The high-level design model of SPC system is shown in Figure 4.10. The design component model is presented in the form of layered architecture which consists of application system layer and business-specific layer. The model shows the interoperability of the component systems and the application systems in form of layered architecture. Application layers consist of variable chart calculation, variable chart, attribute chart calculation,

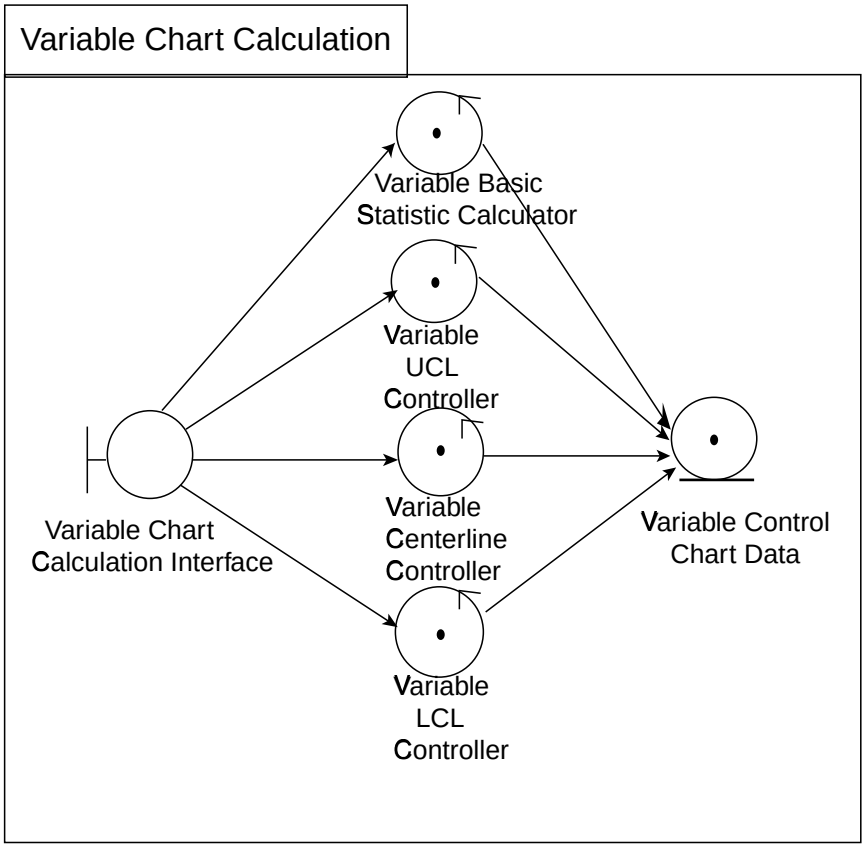

Figure 4.5: Variable Chart Calculation Module

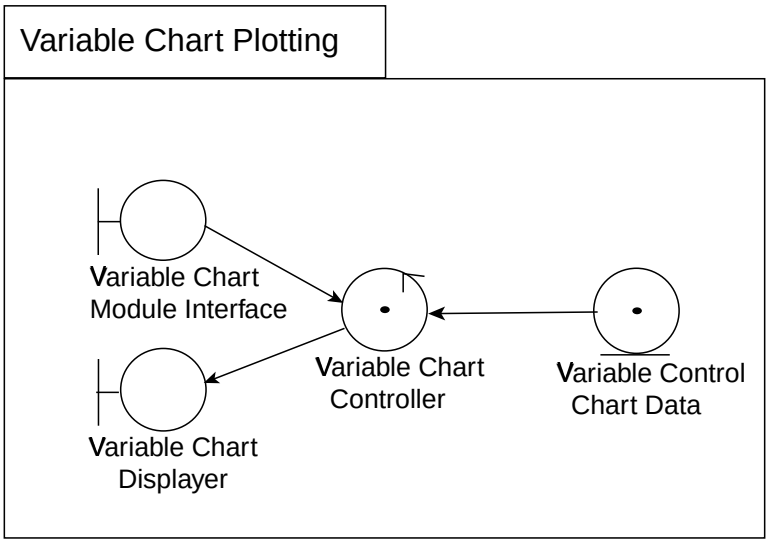

Figure 4.6: Variable Chart Plotting Module

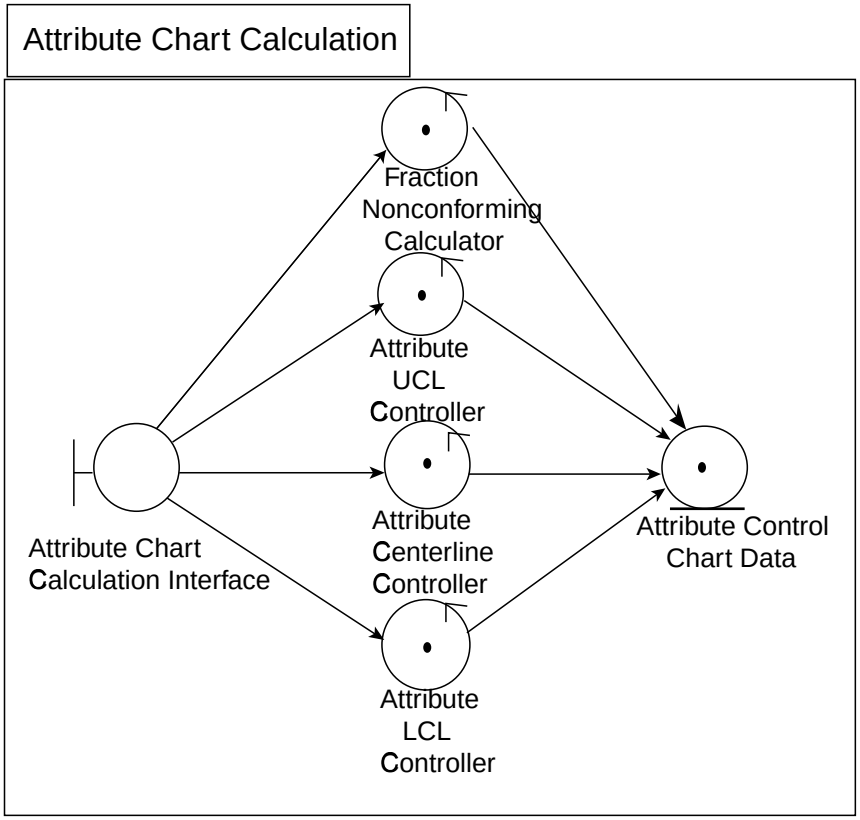

Figure 4.7: Attribute Chart Calculation Module

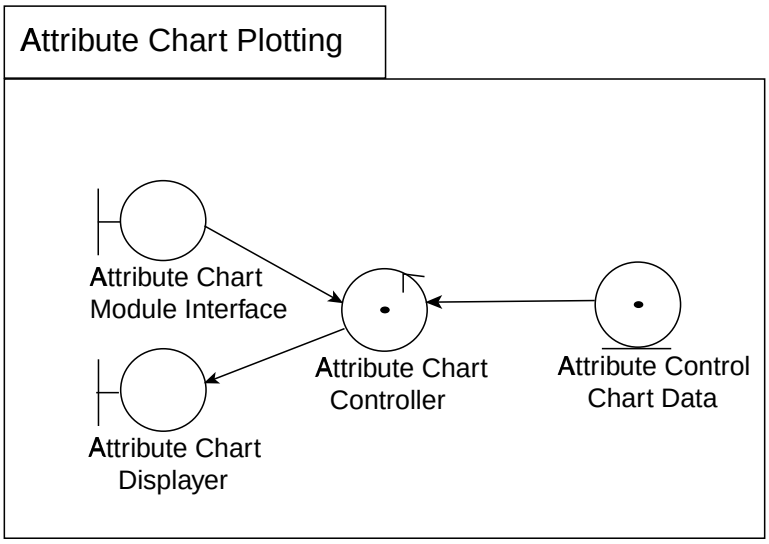

Figure 4.8: Attribute Chart Plotting Module

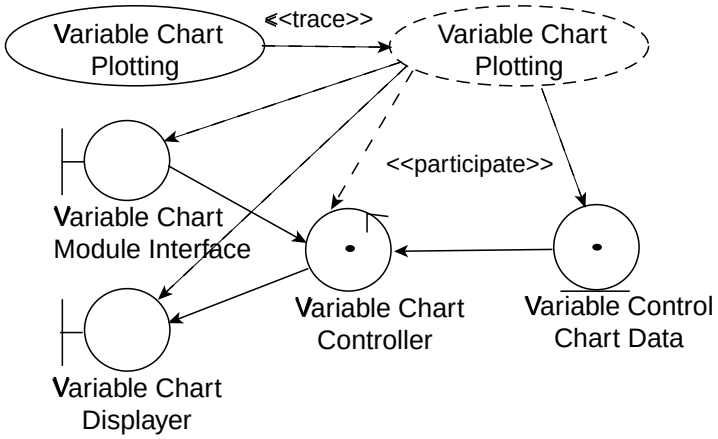

Figure 4.9: Traceability of Use Case Component Model to Analysis Component Model

attribute chart and database management application systems. These application systems can be developed by reusing the reusable component systems in the business specific layer. Component systems which are provided in the business specific layer are variable chart calculation, variable chart, attribute chart calculation, attribute chart and database management component systems. Within these reusable component systems there are special packages called facades.

A facade is defined as a packaged subset of components or references to components which is selected from a component system. Each facade acts as a kind of public interface to the component system (Jacobson et al. 1997). Facades encapsulate the internals of the component system in order to minimise dependencies and changes as the component system evolves. This public access allows the developer to reuse those parts of the component system that have been chosen to be available for reuse. From Figure 4.10, the variable centerline controller facade of variable chart calculation component system can be extended and customised to allow the developer to define a new method to calculate the centerline for variable chart for a selected types of chart.

The interoperability of component systems and application systems is shown in Figure 4.11. These component systems are interacting with the other

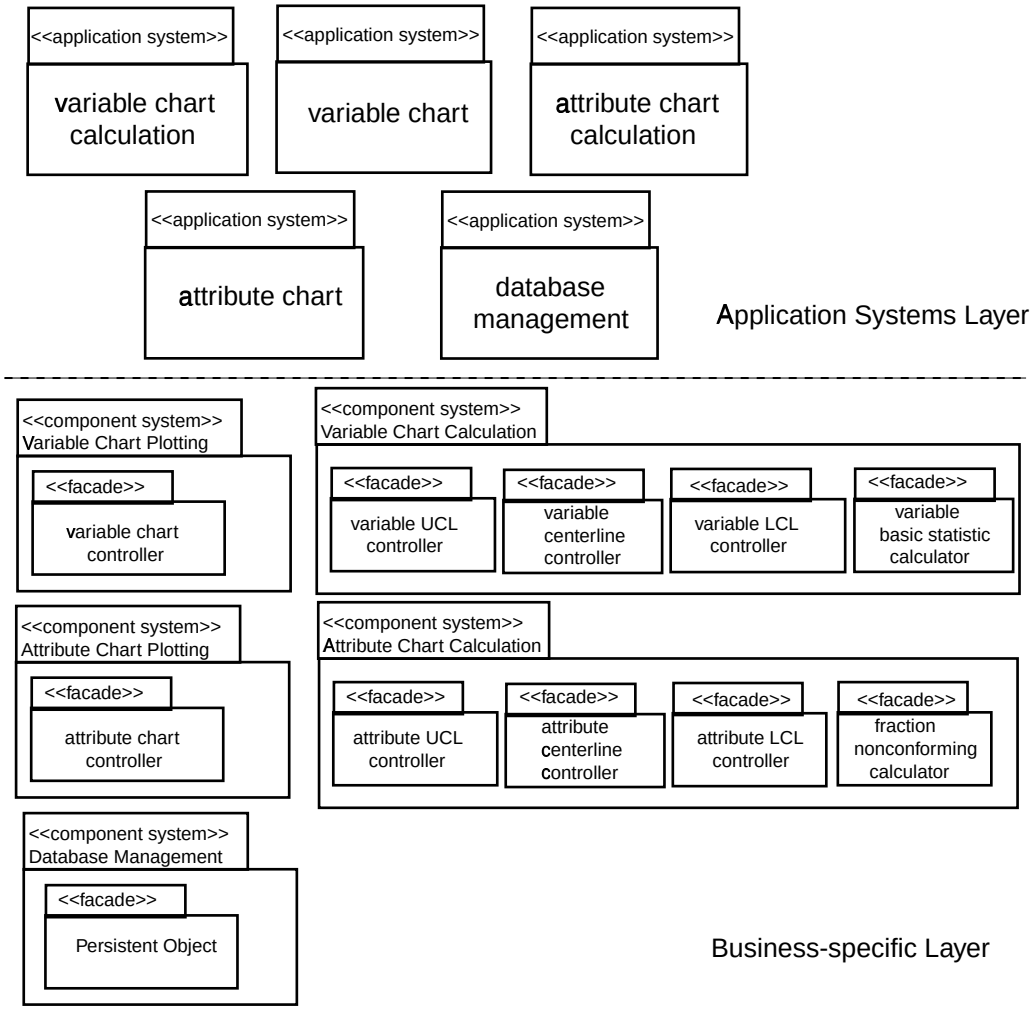

Figure 4.10: Detailed Facade View of SPC Design Component Model

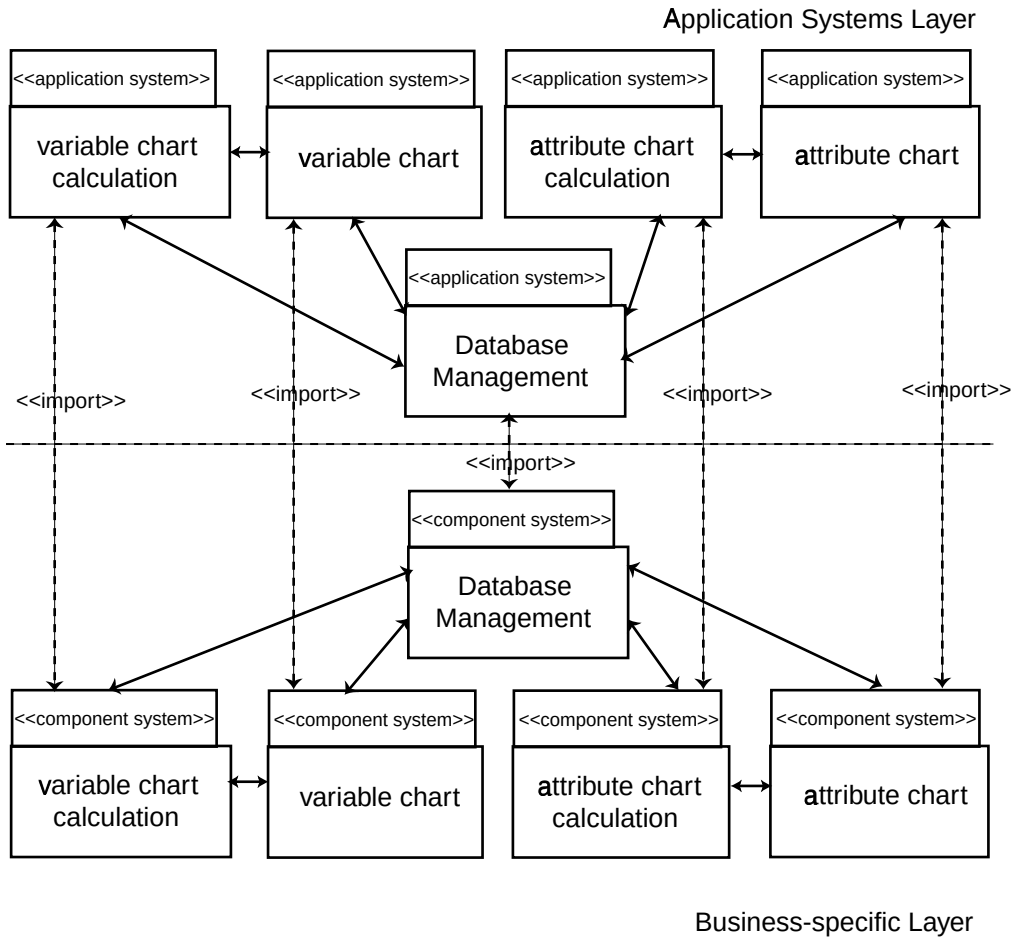

Figure 4.11: Inheritance View of SPC Design Component Model

component systems within business-specific layer. This is to ensure the application systems built from these component systems are well integrated. Variable chart plotting component system is interacting with variable chart calculation component system while attribute chart plotting component system is interacting with attribute chart calculation component system. Database management component system is handling all the entity components as defined in analysis component model. The entity components or persistent classes developed in SPC application framework are variable control chart data and attribute control chart data. These classes can be extended and customised by using this database management component system. Basically this component system is interacting with the database through several simple querries.

4.3 Class Hierarchy

Figure 4.12 shows the inheritance hierarchy of persistance object for SPC application framework. The top level of the hierarchy is *persistence object. Persistence object* is provided by persistence broker (Khor 2000) which translates objects to a few records and save them into the database, and translates records to objects when retrieving from the database. By inheriting the persistance class, *Mitem* obtains the persistence capability. *Mitem* is the abbreviation of manufacturing item, which is used to represent a material or item in production line of factory. The classes that inherit from *Mitem* are *variable control chart data* and *attribute control chart data.* Due to the fact that these objects need to be stored in the database, the identifier of the objects must be unique.

Figure 4.13 shows the inheritance hierarchy of component classes for component system in SPC application framework. The base class is *Module.* Subclasses are *variable chart calculation module*, *variable chart plotting*

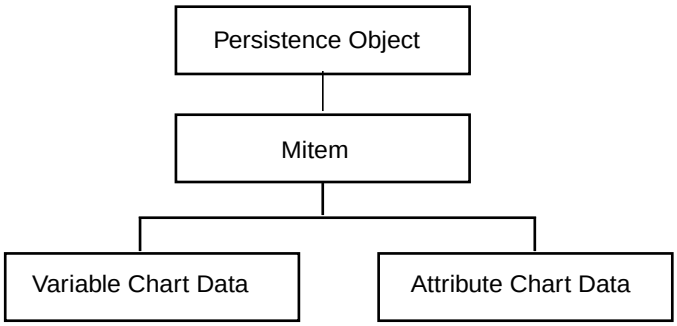

Figure 4.12: Inheritance Hierarchy of Persistance Objects

module, *attribute chart calculation module* and *attribute chart plotting module*. Base class *Module* provides the default interface for the integration of the application systems which are developed from component systems to the whole system. All of the component systems in business-specific layer must inherit from *Module* class.

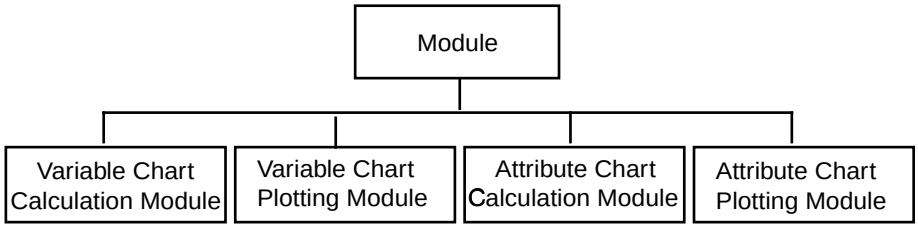

Figure 4.13: Inheritance Hierarchy of Component Classes

Figure 4.14 and 4.15 shows the inheritance hierarchy of variable chart and attribute chart. The classes that inherit from *variable chart* class are *range chart*, *sigma chart*, x *bar range chart* and *x bar sigma chart*. The classes that inherit from *attribute chart* class are *c chart*, *np chart* and *p chart*. *Variable chart* class and *attribute chart* class provide the basic functionalities to plot a variable chart or attribute chart. A new type of variable chart or attribute chart can be developed by inheriting *variable chart* class or *attribute chart* class as superclass.

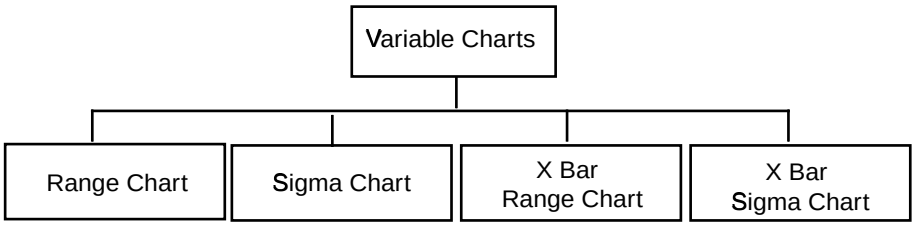

Figure 4.14: Inheritance Hierarchy of Variable Charts

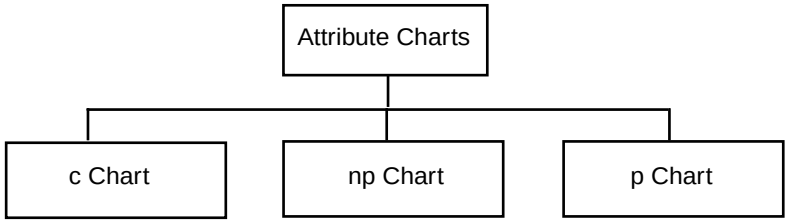

Figure 4.15: Inheritance Hierarchy of Attribute Charts

4.4 Design Pattern

Design patterns are recognised, named solutions to common design problems. The use of the most commonly referenced design patterns promote adaptable and reusable application framework. When a system evolves, changes to code involving design patterns will consist of creating new concrete classes that are extensions or subclasses of previously existing classes.

The design pattern used in the development of SPC application framework is based on the Template Method pattern as defined in (Larman 1997). This pattern is the core of the SPC application framework implementation. A skeleton of an algorithm with its varying and unvarying portion of code is defined in the superclass. The template method invokes other methods, which may be overriden in an extended class or a subclass. The subclass can override the varying methods. In the varying method, a new version method which has unique and different behaviour from the superclass can be defined at the point of variability.

Figure 4.16 shows the design pattern of variable chart plotting component system. Superclass *Charts* has a template method called *Chart*. In the tem-

plate method there is an abstract method called *plotChart.* This method is overridden in the extended concrete class, *RangeChart* and *SigmaChart*. The abstract method is redefined to plot different types of chart.

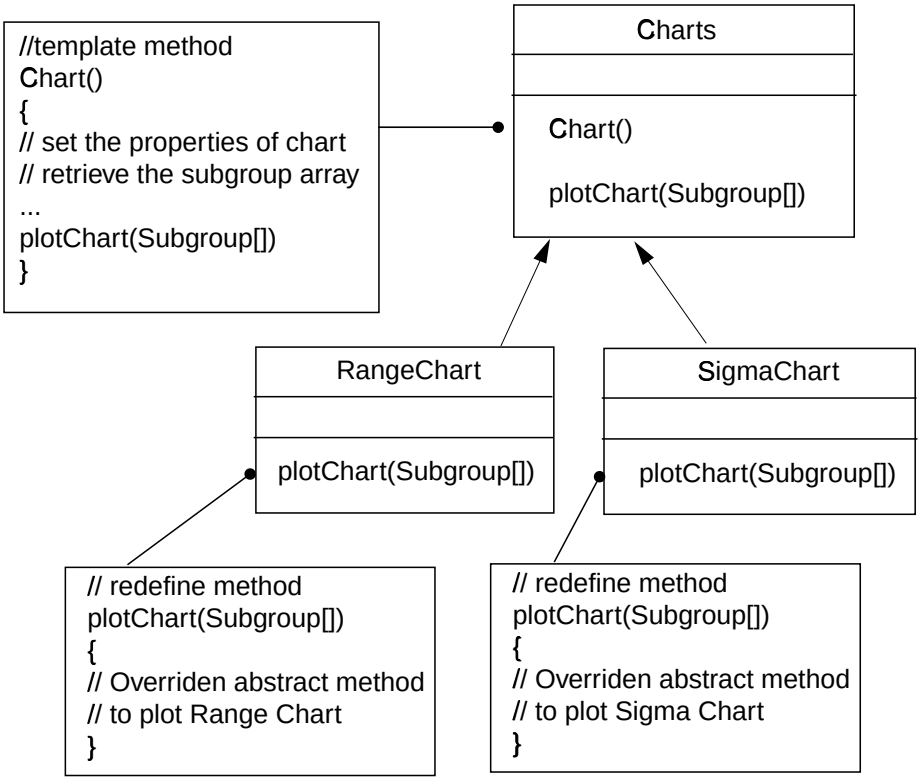

Figure 4.16: Design Pattern of SPC application Framework

4.5 Summary

In the development of the SPC application framework, the use cases and associated information identified in the analysis are the highest level of system strategy. UML is used throughout the project lifecycle to capture and communicate analysis and design decisions.

The development life cycle of SPC application framework begins with domain analysis to develop a use case model and analysis component model. In the domain analysis, a set of reusable component systems and component systems with the commonality and variation points has been identified. The use

case model is mapped to different component systems in the layered architecture. The analysis component model which is a high-level static structure helps to define the domain layered architecture for the application framework.

From the use case model and anaylsis component model, a high-level design component model is defined in a layered architecture system. This layered architecture system consists of application systems in the application system layer and component systems in the business-specific layer. Inheritance hierarchy and the template methods shows the relationships and the design pattern used in the SPC application framework.

Chapter 5

Implementation of Object-Oriented SPC Application Framework

Following from the detailed design phase, the next phase is related to implementing the application framework concerned. Towards the end of the implementation, a ready-to-assemble model will be defined so as to provide a guideline on the usage of the framework during delivery. This model will introduce a ready-to-assemble approach to the application engineers to rapidly assemble and customise an OO application framework for multiple OO applications within the general application domain of SPC application framework for different customers. Due to the iterative nature of the OODE method, it is possible that any recovery of errors might lead to the need to go back to the previous phases. The automated application framework will eventually be integrated and combined to form an overall application framework architecture.

5.1 Implementation Classes

Java programming language provided by Sun Microsystem is used in the implementation of SPC application framework. Java is a very powerful object-oriented programming language. The Java Development Kit (JDK) version 1.2.2. is used. JDK 1.2.2 contains the enhancements for Swing, AWT, Java2D API and a lot of utility classes.

The implementation of template method in this application framework is shown in Figure 5.1. The inheritance key word in Java is *extends*. Abstract class *Charts* has a template method called *chart* and an abstract method *plotChart*. The *RangeChart* class extends the *Charts* class and implements the abstract method *plotChart* to perform the algorithm to plot a range chart.

The *Charts* class implemented in the SPC application framework supports bar chart and line chart currently. *Charts* class coordinates several objects to achieve its aim of being able to draw a chart on a Java 2D graphics device: a *Title*, a *Legend*, a *Plot* and a *DataSource*. The *Plot* class in turn manages a horizontal axis and a vertical axis of the chart. *Title* class displays a description for a chart. Chart *Legend* class shows the names and visual representations of the series that are plotted in a chart. A *Plot* is a class that controls the visual representation of data. *DataSource* class contains some methods that perform various useful functions relating to data sources.

5.2 Application Framework Customisations

An SPC application framework can be very difficult to be adopted and reuse due to its complexity and size. Due to this fact, to reuse the SPC application framework, *wizard* tools that assist developers in choosing and customising the component systems in SPC application framework is integrated into an IDE which has been developed (Lee, Thin & Liu 2001*b*). By using the *wizard*

```
package moos.module.vChart;

abstract public class Charts {

  public void chart() {
  // set the properties of chart
  // create menu bar and chart properties
  // retrieve the subgroup array to plot chart
  }
}

abstract public JChart plotChart(Subgroup[] sub);
```

```
package moos.module.vChart;

public class RangeChart extends Charts {

  public JChart plotChart(Subgroup[] sub) {

  // implementation of the range chart

  }
}
```

Figure 5.1: Implementation of Template Method

to develop a new application, developers have a faster way to reuse the SPC application framework, and more importantly, to benefit from the application framework which shortens software development life cycle.

The *wizard* leads the developer through several steps which consist of module selections and feature customisations to develop a new working application without hassle. A *wizard* has a step-by-step user interface for the developer to follow to customise and extend the component system visually. There are four wizards being developed and integrated in the IDE. They are Variable Control Chart Calculation Wizard, Variable Chart Wizard, Attribute Control Chart Calculation Wizard and Attribute Chart Wizard.

Before creating any new module from the *wizard*, a project has to be created from the IDE as shown in Figure 5.2. There are four modules available to be customised. They are *Variable Chart Calculation Module*, *Variable Chart Plotting Module*, *Attribute Chart Calculation Module* and *Attribute Chart Plotting Module*.

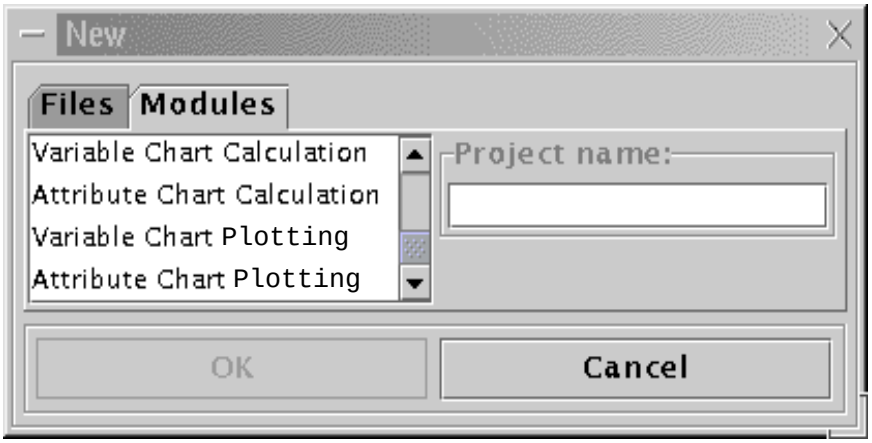

Figure 5.2: Creating New Module

5.2.1 Variable Control Chart Calculation Wizard

For the variable control chart calculation wizard, the first step to customise is to select and view the persistent class. Figure 5.3 shows that there is only one

persistent class which is variable control chart item provided for the customisation. The “View” button in the dialog box is to view the list of attributes with type and name of the persistent class as shown in Figure 5.4. By clicking the “Next” button, wizard will move forward to the next step of customisation.

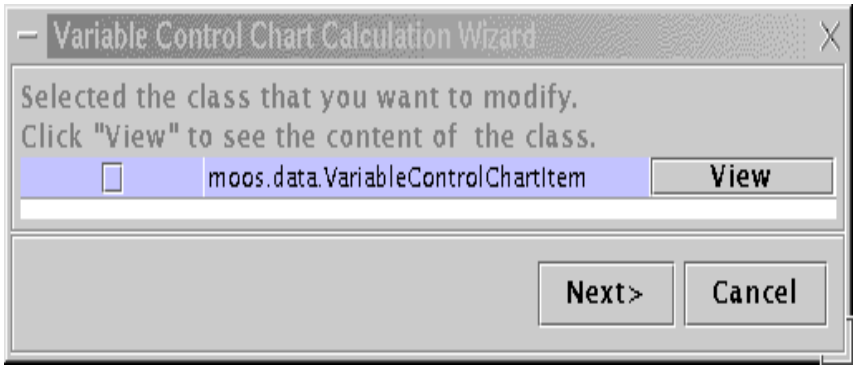

Figure 5.3: Class Selection of Variable Control Chart Calculation Wizard

moos.data.VariableControlChartItem

Data Type	Name
java.lang.String	Material_Code
double	Sigma_Upper_Control_Limit
java.lang.String	Description
double	XbarRange_Upper_Control_Limit
double	Sigma_Centerline
double	XbarSigma_Centerline
double	XbarRange_Lower_Control_Limit
double	XbarSigma_Upper_Control_Limit
double	Sigma_Lower_Control_Limit
double	Range_Centerline
double	XbarRange_Centerline
java.lang.String	priKey_Item_ID
int	subCount
double	Range_Upper_Control_Limit
double	Range_Lower_Control_Limit
double	XbarSigma_Lower_Control_Limit

Figure 5.4: Attributes List for class VariableControlChartItem

The next dialog box contains a “Add” button to add new attributes to the persistent class and a “Remove” button to remove an attribute as shown in

Figure 5.5. There is a textfield to show the extended new persistent class name and the list of new attributes added. For instance, three new attributes added are MR_Upper_Control_Limit, MR_Centerline, MR_Lower_Control_Limit of type double.

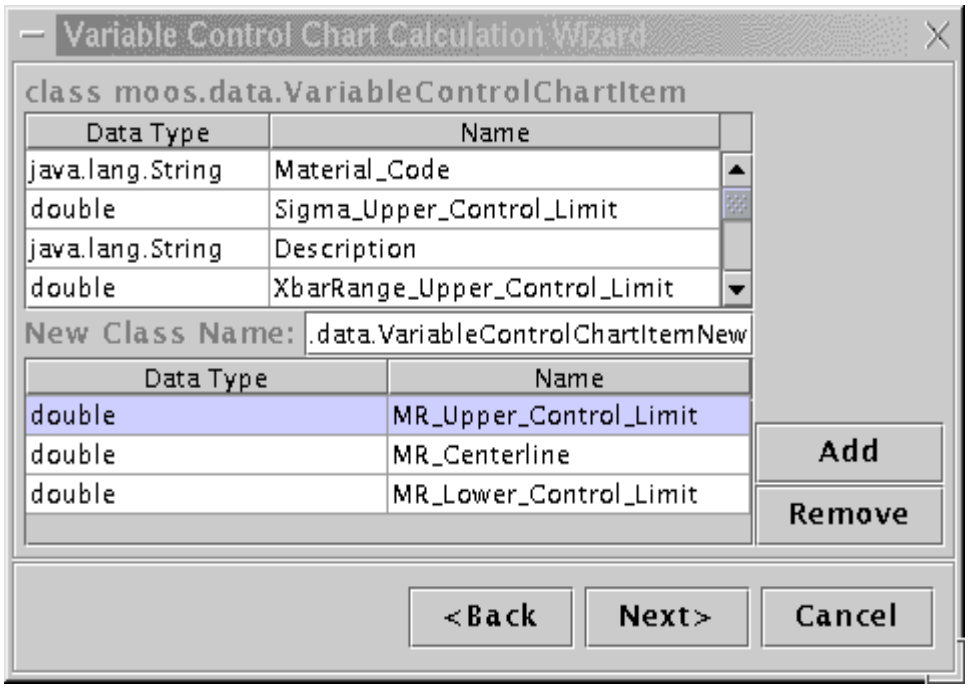

Figure 5.5: Variable Control Chart Calculation Wizard Customisation

After adding the new attributes, by clicking the “Next” button, the wizard will show information of the extended persistent class with the list of new attributes as shown in Figure 5.6. The next step is to customise the display of variable control chart calculation module. A new field can be added to the table in Subgroup class for calculation. The wizard is shown in Figure 5.7. By clicking the “OK” button, the wizard will retrieve all the information from the input through the previous steps and generate all the necessary source files and codes.

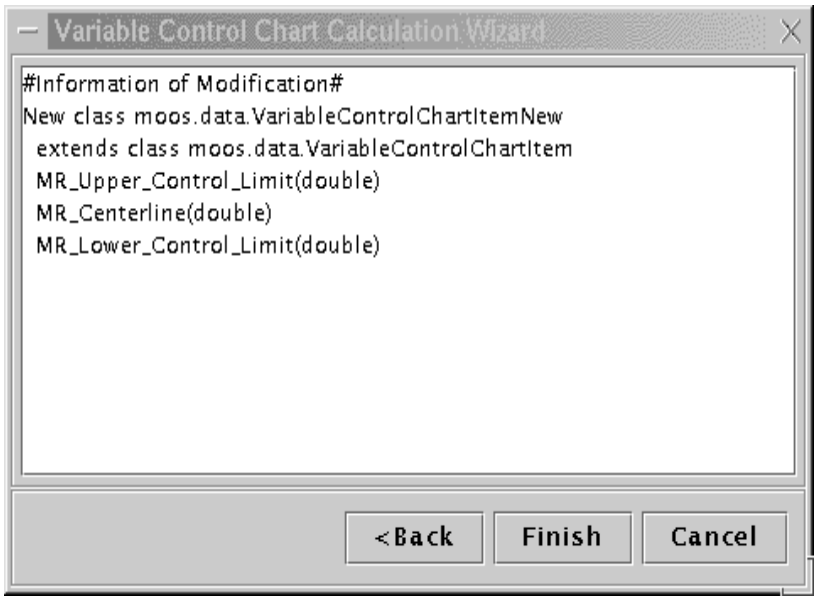

Figure 5.6: New Persistent Class with New Attributes List

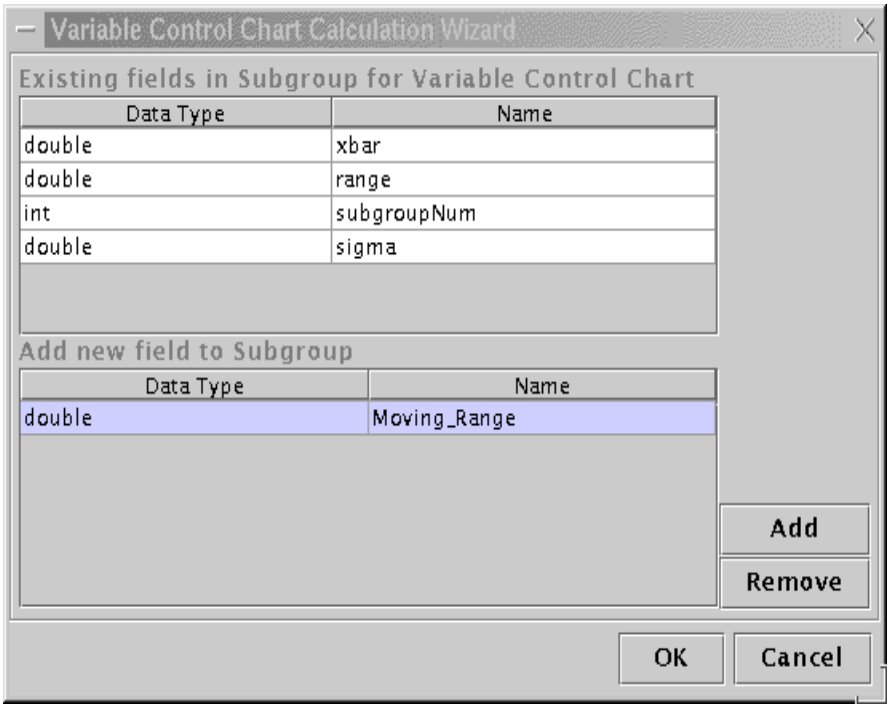

Figure 5.7: New Field for Subgroup

5.2.2 Variable Chart Wizard

Variable chart wizard is used to customise the Variable Chart Plotting Module for the developer to add new types of chart. Figure 5.8 shows the variable chart wizard customisation dialog box. This dialog box shows a list of existing variable charts. By clicking the “Add” button, a new type of chart can be added to the variable chart plotting module. The chart name and the chart class name have to be specified. After providing the necessary information to the wizard, on clicking “Next” button, a summary of the variable chart added to the wizard is shown in a dialog box as shown in Figure 5.9. The wizard at this stage is able to generate the code and the source files based on the input from the previous steps. This is achieved by clicking on the “Finish” button.

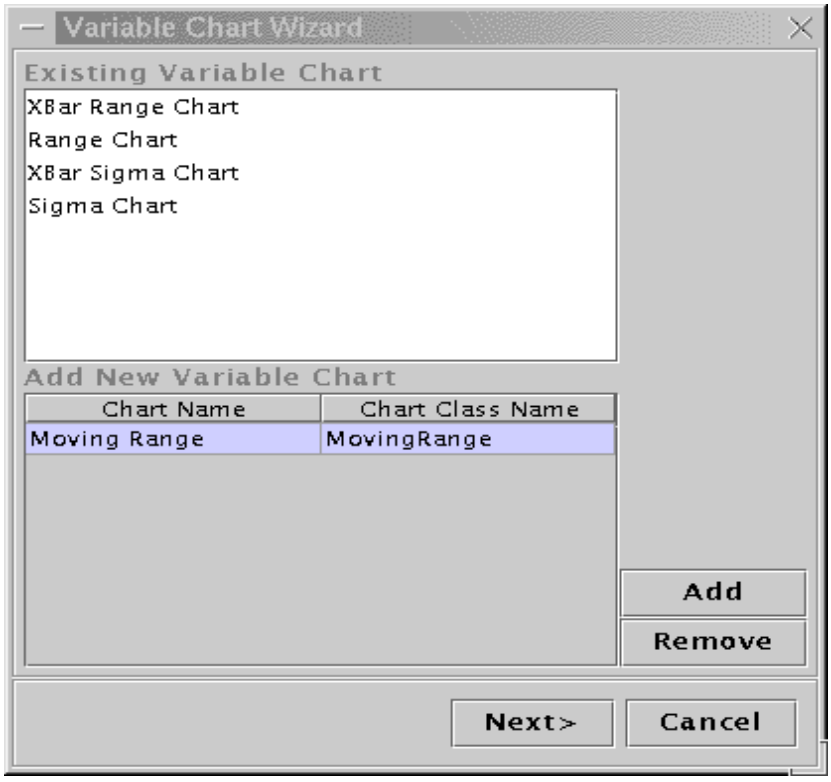

Figure 5.8: Variable Chart Wizard

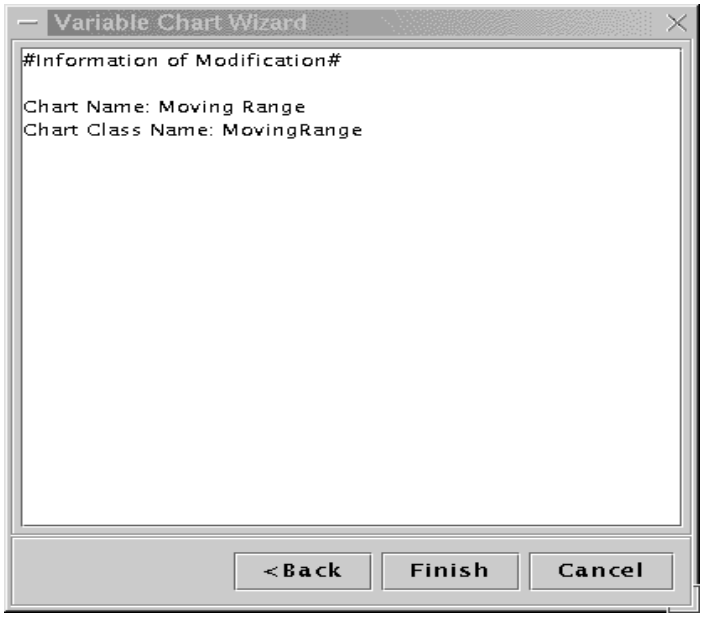

Figure 5.9: Variable Chart Wizard Summary

5.2.3 Attribute Control Chart Calculation Wizard

Attribute control chart calculation wizard is used to customise the calculation of attribute control chart. By using this wizard, the developer can add new types of attributes and define a new algorithm for calculation. Figure 5.10 and Figure 5.11 show the persistent class and the complete list of attributes in the class.

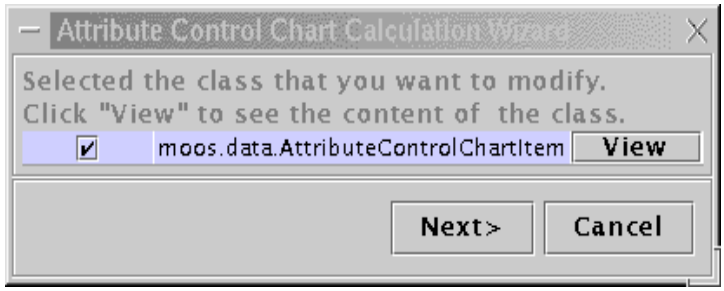

Figure 5.10: Class Selection of Attribute Control Chart Calculation Wizard

The next step of the wizard is to add new atributes to the attribute control chart calculation module as shown in Figure 5.12. The “Add” button is to add a new attribute to the persistent class and the “Remove” button is to remove an attribute. After adding a new attribute, the new attribute will appear

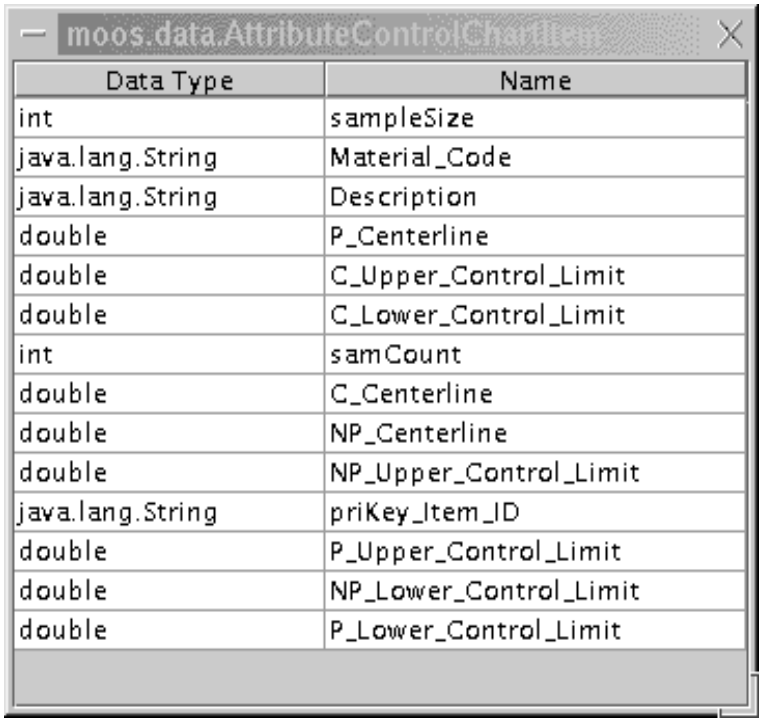

Figure 5.11: Attributes List for class AttributeControlChartItem

at the column which shows the data type and the attribute name. There is a textfield to show the new extended persistent class name. For instance, three new attributes are added which are U_Upper_Control_Limit, U_Centerline, U_Lower_Control_Limit of type double. After adding the attributes to the wizard, the next dialog box will show the summary for the result of the previous customisation as shown in Figure 5.13.

The next step of the wizard to customise the attribute control chart calculation is to add a new column to display the result of the calculation or a new column for the end user to input data in the table. A new column can be added to the table for Sample class for this purpose. As shown in Figure 5.14, the wizard dialog box is used to add a new field for Sample class. By clicking on the “OK” button, the wizard will retrieve all the information from the input through the previous steps and generate all the necessary source files and codes.

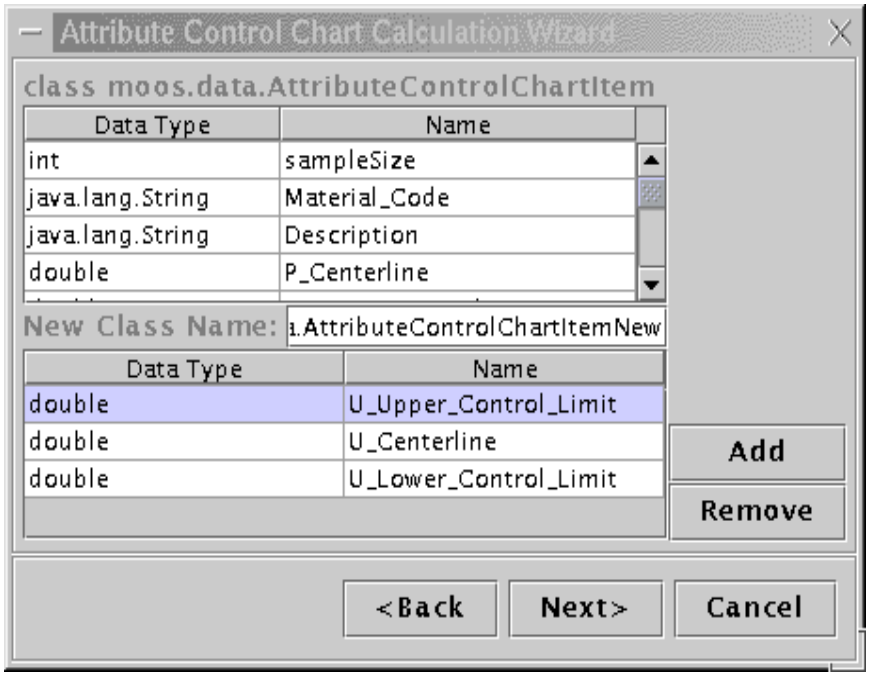

Figure 5.12: Attribute Control Chart Calculation Wizard

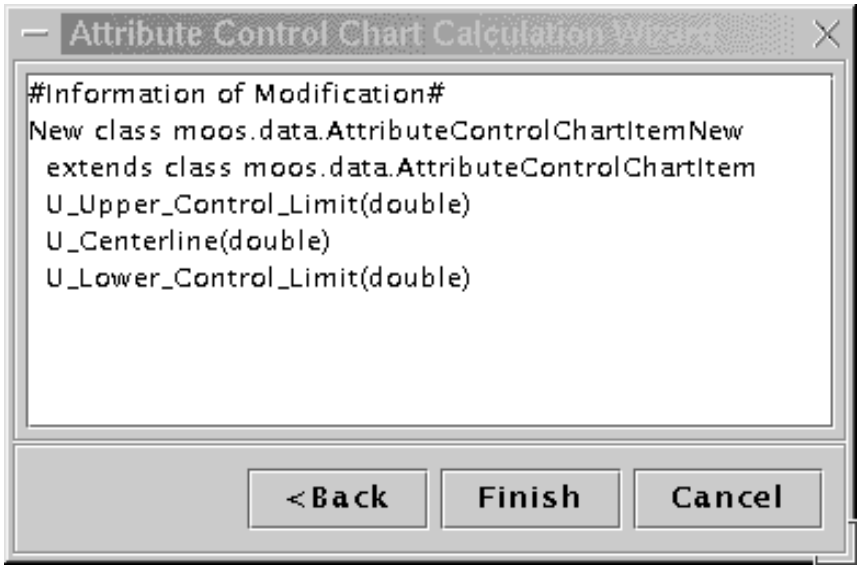

Figure 5.13: Attribute Control Chart Calculation Wizard Summary

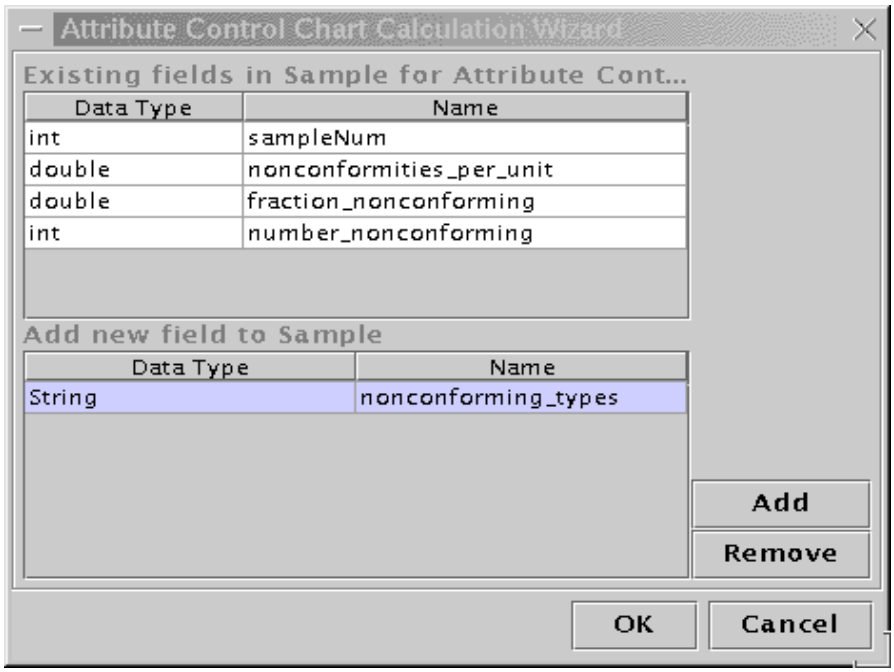

Figure 5.14: New Column for Sample Class

5.2.4 Attribute Chart Wizard

Attribute chart wizard is used to customise the Attribute Chart Plotting Module to add new types of attribute chart into the SPC application framework. Figure 5.15 shows the attribute chart wizard's customisation dialog box which has a list of existing attribute charts. By clicking on the "Add" button and specifing the chart name and the chart class name, a new type of chart is added. After providing the necessary information to the wizard, by clicking on "Next" button, a summary of the attribute chart added to the wizard is shown in a dialog box as shown in Figure 5.16. The last step of the wizard is to click on the "Finish" button. This will generate the code and the source files based on the input from the previous steps.

5.3 Customisation Procedure of Wizards

The procedure of building a new application on top of the existing component systems is discussed in this section. Wizards developed in this research

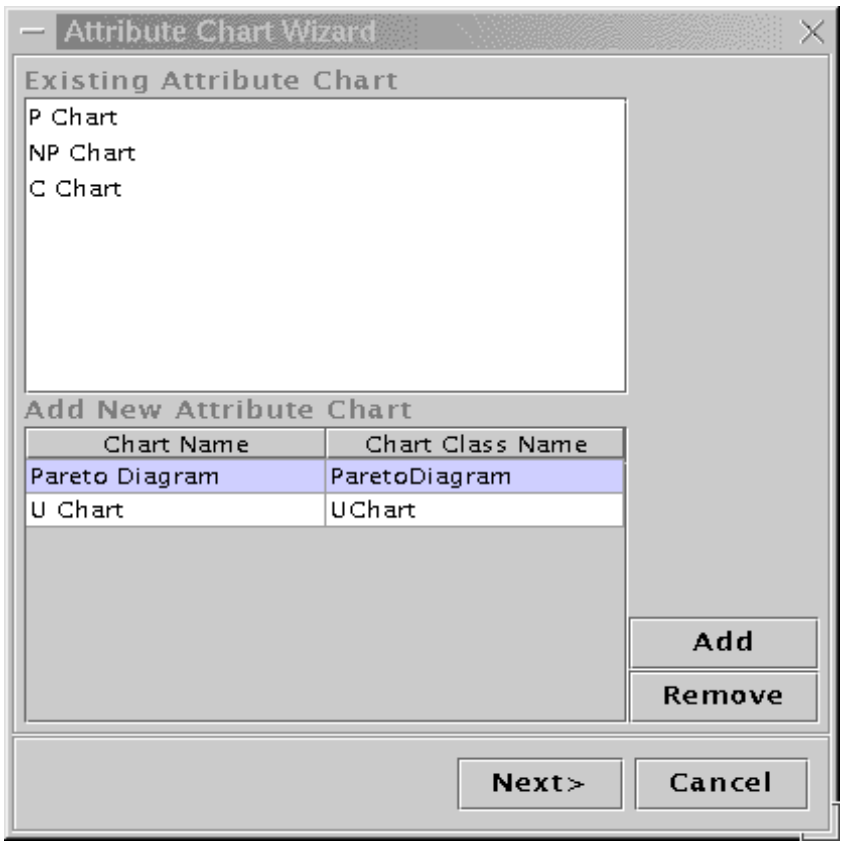

Figure 5.15: Attribute Chart Wizard

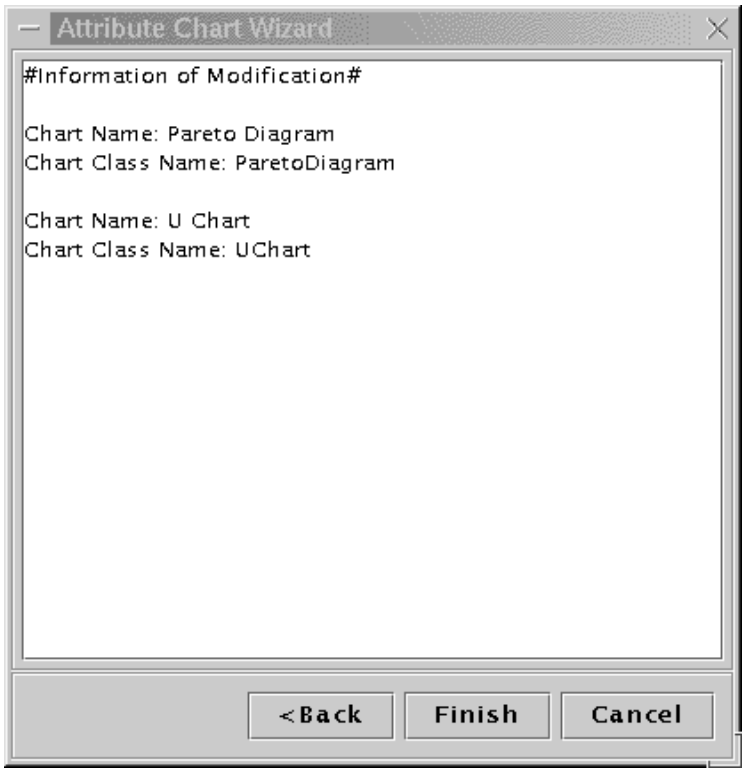

Figure 5.16: Attribute Chart Wizard Summary

project provide a simple way of customisation from the application framework. By performing the customisation of the application framework, the stability and interaction between component systems of the application framework, a reliable way of reusing one or more component systems can be determined. For more details, please refer to Appendix C for a case study.

The customisation consists of adding two new types of chart which are *U chart* and *Pareto diagram* to the existing application systems. This can be achieved by adding additional fields with a particular data type to a persistent class and adding a new calculation algorithm to the atttibute chart calculation module and attribute chart plotting module.

For the atttibute chart calculation module, the persistent classes involved are *moos.data.AttributeControlChartItem* and *moos.data.Sample*. The customisation of these classes is accomplished by creating a new subclass which is inheriting from the base class by adding new attributes to the new class. The new class for the *moos.data.AttributeControlChartItem* is *moos.data.AttributeControlChartItemNew*. The new attributes added are U_Upper_Control_Limit of data type double, U_Centerline of data type double and U_Lower_Control_Limit of data type double. These three attributes represent the three control limits which are upper bound, centerline and lower bound of a *U chart*. The new class for the *moos.data.Sample* is *moos.data.SampleNew.* The new attribute added is nonconforming_types of data type string. This attribute is the column for the user to input the type of nonconfoming.

The wizard steps of customisation of the attribute chart calculation module can be referred from Figure 5.10, Figure 5.11, Figure 5.12 and Figure 5.13. The customisation through the wizard of the attribute chart plotting module is shown in Figure 5.14, Figure 5.15 and Figure 5.16. The charts added are *U chart* and *Pareto diagram*. These two charts are new classes extended from the attribute chart base class. In oder to plot a *U chart*, we need the at-

tributes added for the class *moos.data.AttributeControlChartItemNew* which are U_Upper_Control_Limit, U_Centerline and U_Lower_Control_Limit. While for the Pareto diagram, the attribute needed is nonconforming_types for the class *moos.data.SampleNew.* These attributes are defined for the atttibute chart calculation module. From Figure 5.17, these attributes have been added to the atttribute chart module application.

The plotting of the chart is handled by the atttribute chart plotting module. Thus, to add and plot a new chart, two modules are involved in this case. However, this can be easily done by copying the project JAR files from the atttibute chart calculation module project folder to the atttibute chart plotting module project folder. The project JAR file is ready after the compilation of the project for that particular module in IDE. The project JAR file is a Java binary file that contains the compiled and linked classes. From Figure 5.18, *U chart* and *Pareto diagram* have been added to the application for the user to select and to plot. The output charts are shown in Figure 5.19 and Figure 5.20.

5.4 Summary

SPC application framework has provided several reusable component systems for the developer to reuse. The channel to customise and modify component systems is through the wizards tool. Wizards are developed to redefine, customise and extend the existing component systems to meet the user's requirement. It is a simple, powerful, easy-to-use tool providing a convenient way to develop a SPC application from the framework. Through several steps and clicks on the mouse, files and code are generated and ready for the developer to deploy the changes. Four wizards are provided, which are Variable Control Chart Calculation Wizard, Variable Chart Wizard, Attribute Control Chart

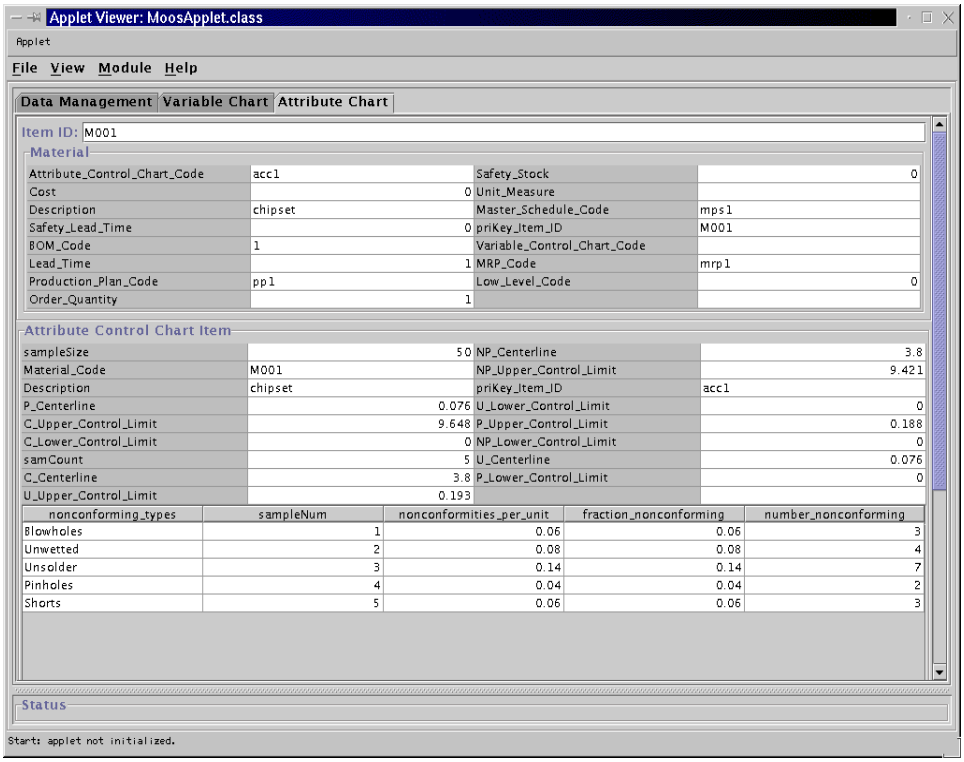

Figure 5.17: Attribute Chart Module Application After Customisation

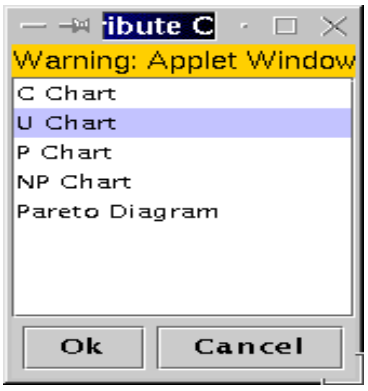

Figure 5.18: Attribute Chart Selection Window After Customisation

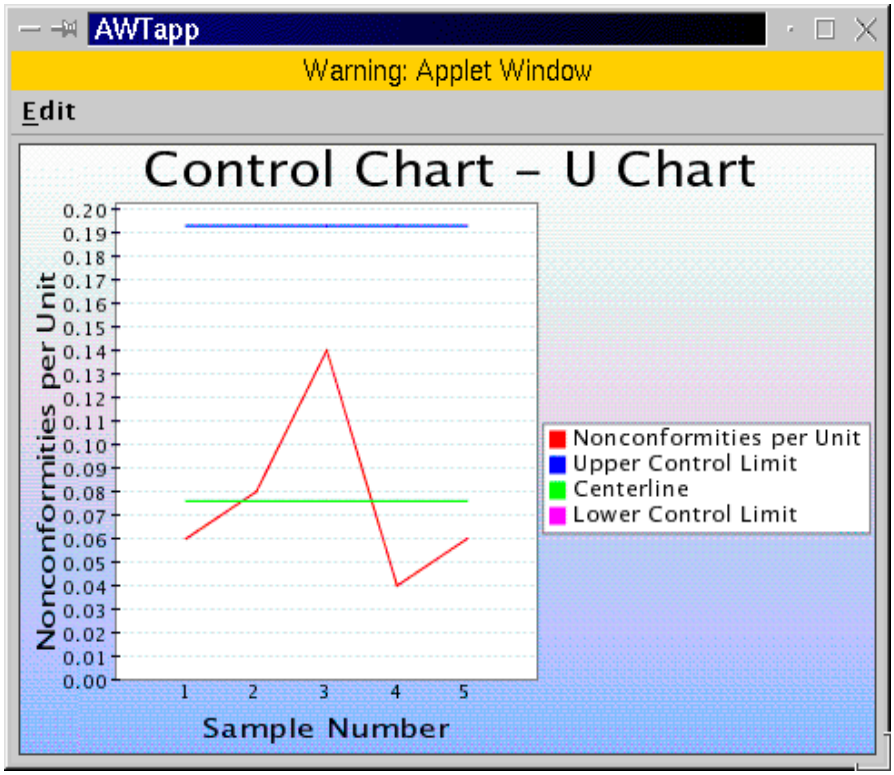

Figure 5.19: U Chart

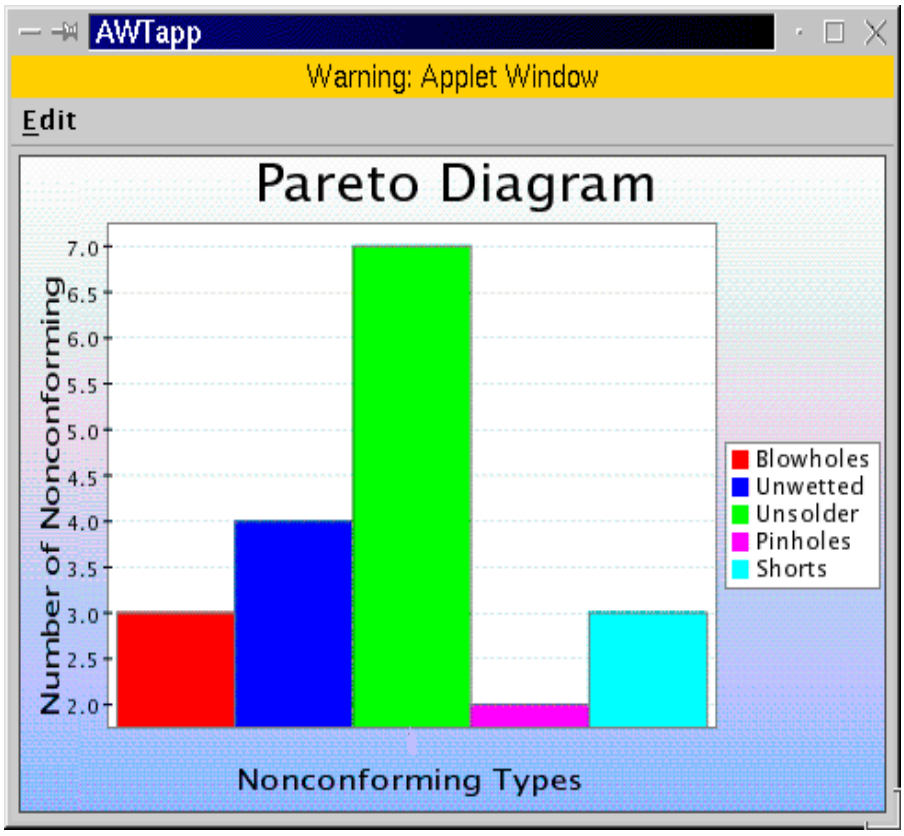

Figure 5.20: Pareto Diagram

Calculation Wizard, and Attribute Chart Wizard.

Chapter 6

Conclusion

The SPC application framework in the domain of manufacturing has been successfully developed in offering generic and reusable components and achieving the objectives in this research project. The functionalities provided by the SPC application framework are runnable and can be easily adapted and customised to build new SPC applications according to the specific needs of the developer. For more details, please refer to Appendix A for installation guide and Appendix B for user guide.

This research project presents an SPC application framework in the domain of manufacturing based on the Object-Oriented Techonology (OOT), systematic software reuse, layered architecture and most importantly the approach of application framework. A layered architecture is an architecture style that simplifies system evolution and enables substantial reuse of component systems. SPC application framework's approach focuses on building a system in layered, modular, object-oriented style which greatly enhances large-scale systematic reuse. This makes its infrastructure to be easily reused by the developer to build more specific applications in the domain of SPC in the manufacturing environment.

The layered architecture is an architecture that organises a software sys-

tem in layers, where each layer is built on top of another more general layer. By separating operations of a system from the user interface, it produces a more maintainable application with higher cohesion and less coupling. With SPC application framework, the application developer can focus on solving the business domain problem rather than programming problem which is related to architecture foundation matter.

SPC application framework is extendable and customisable. It consists of most of the reusable components for SPC applications such as variable control chart calculation component, variable control chart plotting component, attribute control chart calculation component and attribute control chart plotting component. These components can be modified and customised to develop more specific SPC applications in manufacturing domain.

Based on this project, a few papers of the research work have been published in local journals as well as international conferences (Lee, Thin & Liu 1999, Lee, Thin & Liu 2000*b*, Lee, Thin & Liu 2000*a*, Lee et al. 2001*a*, Lee et al. 2001*b*).

6.1 Strength of Research

The main strength of the SPC application framework is that the analysis, design, and implementation time for complex SPC applications in manufacturing systems are greatly reduced by reusing the provided business analysis, design and component systems. SPC application framework can be modified and customised easily to suit a particular requirement change. This is applying to the domain in order to let the manufacturer stay competitive in the market and eventually fulfill the ultimate goal of reducing time-to-market and produce a standard quality of software.

SPC application framework provides the ways for the developer to both

use the built-in functionality and modify or extend that functionality. Typically, developers can use SPC application framework's built-in functionalities directly without any modification and changes. If modification is needed, developers can extend and modify the functionalities by deriving new classes and overriding member functions from the application framework. SPC application framework minimises the amount of code needed to implement similar applications. This is true since the common abstractions for the applications are captured in the application framework, which reduces the new code to be developed.

In order to shorten the time to customise or develop new applications from the application framework, wizard tool is integrated to provide step-by-step instructions to assist software developers. The wizard provides a convenient way and an easy-to-use GUI to customise the application framework.

The selection of Java as the development language has provided a lot of features to SPC application framework such as portability, platform-independence and a common language for remote object invocation by networking communication. A lot of utility classes are provided in Java. These classes can be utilised and easily integrated into the application framework to provide a more powerful application framework.

6.2 Limitation of Research

The development of SPC application framework is an iterative process that can be refined. The application framework has to be changed from time to time to become more stable and attainable. Software development is iterative. This is important to evolve the application framework to become more mature and reliable. The iteration starts from recapturing and redefining the requirements and more importantly the commonality and variability across

the domain.

Applications built using SPC application framework are easier to maintain because assuming that the framework is correct, only the extensions that the applications introduce have to be maintained. However, the application framework will evolve over time and thus applications must evolve with it. The optimum of the framework design is not yet attainable at this early framework development iteration life cycle. The initial expected variations in a reusable component system can become a common requirement later that warrants inclusion into the framework component system design.

In order to achieve a more significant degree of success in software reusability for SPC applications in the domain of manufacturing, some other applications in statistical quality control (SQC) for manufacturing need to be added, for instance acceptance sampling, which consists of various standards of sampling plans such as single sampling, double sampling and unit sequential sampling. This will make the application framework a more comprehensive solution for manufacturing environment.

In SPC, there are a number of different control charts, each of which performs best for a particular kind of data for different situations. Control charts developed in the SPC application framework only support line and bar charts in the current implementation.

6.3 Future Work

Many factors can trigger the iteration of the software development life cycle, such as changing of requirement from the customer and feedback from the application developers. The evolution of the application framework from each iteration, makes the architecture and design more stable and reliable. The redefinition of commonality and variability of the application framework which

may consist of modification of the variation points or common requirement makes the problem domain to be addressed more accurately and correctly.

The domain of the SPC application framework can be extended to add component for other SQC applications. By developing these new reusable components within SPC application framework, the issues of integration with other component systems can be minimised.

SPC application framework can be further enhanced by adding more types charts, such as pie chart, boxplot and scatter diagram. Besides this, the input data capturing can be automated instead of key in manually by the user. In the manufacturing environment, data is captured by machines or robots. The automation of the data handling will enable the data analysis to be done in real-time mode.

6.4 Concluding Remarks

SPC application framework is modelled and designed in a layered system architecture which offers high-level reusability compared to a black-box framework and class library code reuse. The understanding of the business model and reuse process become simplified by developing a stable layered system architecture that captures the framework design abstractions into package components with interfaces and facades. SPC application framework is designed to accommodate variation points, customisations and assembly of new applications which can be developed easily by utilising the SPC wizard tools. The conclusion from the research project is, SPC application framework can result in rapid application system development and provides flexibility to customise SPC applications.

Appendix A

Installation Guide

A.1 System Requirement

These are the hardware requirements to install the application framework:

- For personal computer, the minimum requirement is Pentium II 550.
- Memory requires a minimum of 128 Megabyte RAM.
- 10 Gigabyte hardisk space is recommended for the data storage. The space required to install is at least than 20 Megabtye.

These are the software requirements to install the application framework on a desktop:

- An operating system that support Java such as UNIX, Windows, Linux, etc.
- Sun Microsystem Java Development Kit 1.2.2 (JDK 1.2.2) installed on the system.
- A database that supports Java Database Connectivity (JDBC) such as MySql.
- A web server that supports Java Applet

A.2 Installation

SPC application framework is developed in Java. All of the binary and class files needed to run the system is packaged in different folders, which are *demo*, *ide*, *server* and *applet*. In order to run the applications by using a web browser, the *applet* folder has to be copied to the HTML folder of the web server. The other folders can be copied to any directory.

Demo folder contains some examples of the application modules such as AttributeChartCalculation module, VariableChartCalculation module, attributeChart module and variableChart module. These application modules are created by using the wizard tool which is included in the IDE. The *ide* folder contains the IDE which is used to customise SPC application framewrok and create new applications. *Applet* and *server* folders contain all the developed Java class files of the SPC application framework.

Appendix B

SPC Application Framework Server and Applet User Guide

SPC application framework is an object-oriented application framework developed for statistical process control applications development in manufacturing environment. SPC application framework is divided into three main components, server, applet and IDE (Integrated Development Environment). Server provides the interface to the database and is used for applet to connect to the database. The Applet component has all of the core applications of the statistical process control applications. IDE is the development tool used to create new application modules and to customise the existing application modules from the applet and server components. This appendix focuses on the user guide for the server and applet components. The customisation of the application framework will be discussed in the next appendix through an example.

B.1 The Server Environment of SPC Application Framework

The Server component has a persistence broker that provides the object persistence services to the user application while shielding the user from the details of the storage mechanism used (Khor 2000). It maintains the integrity of the data in the database. The main usage is to save the data from the applet to a relational database and retrieve the records when received a request from the applet.

To run the server of SPC application framewrok, the user can execute a bash file called *run* from the console in Linux or a *run.bat* file if the platform is Windows. The file is located at the server directory. Basically, the file is used to execute a Java command, as shown below:

java -classpath MoosServer.jar <connection string> <user> <password>

The connection string is used to specify the connection to the type of database, database ip address and the database name. The format is:

jdbc:<database type>://<ip address>:<database name>

User and *password* are used for authentication to connect to the database.

The server provides a GUI for the user to specify the database type, IP address of the database, database name, user name and the password as shown in Figure B.1.

B.2 The Applet Environment of SPC Application Framework

Before the SPC application framework's applet component can be executed, the server has to be launched first in order to establish the connection to the database. The application can be run from a web browser, such as Netscape

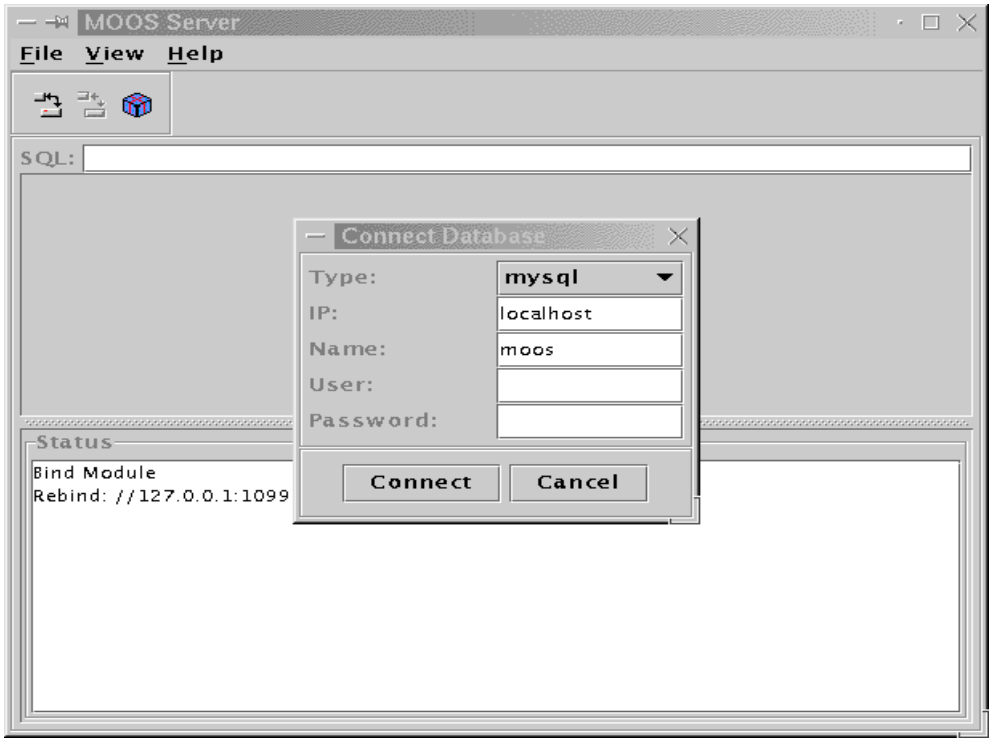

Figure B.1: SPC Application Framework Server

or Internet Explorer by entering the URL from the web browser. The URL is set according to the user's web browser setting. For instance:

http://127.0.0.1/applet/SPCApplet.html

After the applet is launched, the SPC application framework server will be connected automatically. There is a configuration file called *ClientModule.cfg* in *applet* folder. The purpose of the file is to configure the number of application modules to be loaded when the applet is launched. The file contains the applet module and the module class names. By default, there are five applet modules that will be loaded during an applet startup. These modules are DatabaseModule, VariableChartModule, AttributeChartModule, VChartModule and AChartModule as shown below.

DatabaseModule = moos.module.database.DatabaseModule

VariableChartModule = moos.module.variableChart.VariableChartModule

AttributeChartModule = moos.module.attributeChart.AttributeChartModule

VChartModule = moos.module.vChart.VChartModule

AChartModule = moos.module.aChart.AChartModule

Appendix C

Case Study : Variable Control Chart Customisation

The case study of customising and modifying the SPC application framework by using the IDE wizard tool is demonstrated in this appendix. The customisation includes the steps from using the wizards in the IDE to develop a new application until the steps to plug-in the newly developed application to the SPC applet and server.

C.1 Modification of the Variable Control Chart

The objective of the customisation is to add a new chart to the existing variable control charts in SPC application framework. The variable control charts provided in the SPC application framework are *X-Bar* chart, *Sigma* chart, *X-Bar Range* chart and *X-Bar Sigma* chart. The new chart to be added is called *Moving Range* chart. To add this new chart, two persistent classes and base class of chart need to be modified. The new persistent classes are used to keep the calculated data to the database based on the new algorithm, while chart class is used plot the new chart. This customisation will involve two

modules. *Variable Chart Calculation Module* is used for the persistent class modification while *Variable Chart Plotting Module* is used for the plotting chart operation.

The name of the persistent classes need to be modified are *moos.data.Subgroup* and *moos.data.VariableControlChartItem*. The base class of the chart is *moos.module.vChart.Charts*. The customisation of the persistent class and chart class can be done by extending or inheriting the base class and adding new attributes to the new class. The new class name of the *moos.data.Subgroup* is *moos.data.NewSubgroup* and the added attribute is Moving_Range of data type double. The new class of the *moos.data.VariableControlChartItem* is *moos.data.VariableControlChartItemNew* and there are three new attributes of data type double added. They are *MR_Upper_Control_Limit*, *MR_Centerline* and *MR_Lower_Control_Limit*. The new class name of the *moos.module.vChart.Charts* is *moos.module.vChart.MRChart*. A new attribute "*Moving Range*" of string data type is added.

C.2 Using IDE Wizard Tool for the Customisation

The previous section has discussed about the changes that have to be made to the modules. This section is focusing on the module customisation by using the IDE wizard tool. To run the IDE wizard tool, the user has to execute a run script provided in *ide* folder. The first step is to modify the *Variable Chart Calculation Module* by creating a new project by selecting the **File->New** from the menu bar in the IDE. As shown in Figure C.1, a "New" dialog box will appear. Select the *Variable Chart Calculation Module* and enter the project name. After that, click on the "OK" button. The project file will be saved at the location chosen by the user.

The variable chart calculation wizard will appear after clicking on the

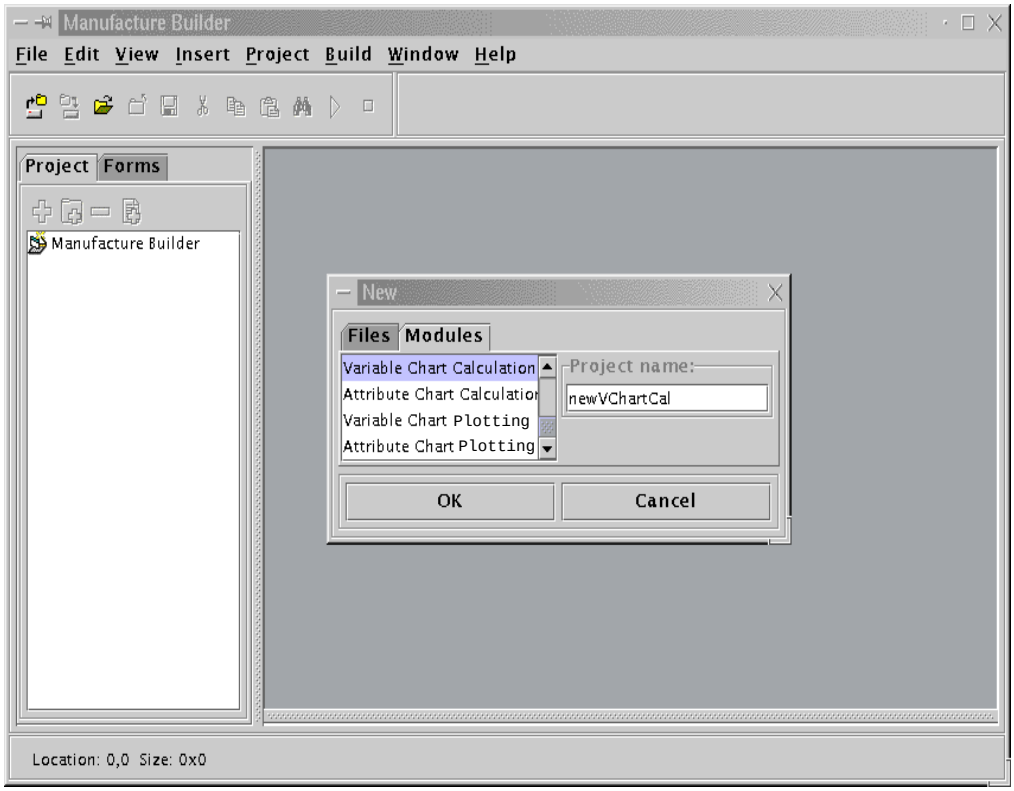

Figure C.1: Variable Chart Calculation Module New Dialog Box

"OK" button. As mentioned earlier, the persistent classes involved in this module's customisation are *moos.data.VariableControlChartItem* and *moos.data.NewSubgroup*. The following menu as shown in Figure C.2 requires the developer to add new attributes to the wizard for customisation. The "Add" button is used for this purpose. For this demonstration, three attributes of double data type are added. They are *MR_Upper_Control_Limit*, *MR_Centerline* and *MR_Lower_Control_Limit*. By clicking on "Next" button, the summary of the attributes is generated for reference as shown in Figure C.3. The new class name is *moos.data.VariableControlChartItemNew* now. It has three additional attributes if compared to the original base class.

The next step is to customise class *moos.data.Subgroup* after clicking on the finish button on the summary for class *moos.data.VariableControlChartItemNew*. As shown in Figure C.4, a new attribute called *Moving_Range* of double data type is added for the new class *moos.data.NewSubgroup* which is extended from class *moos.data.Subgroup*.

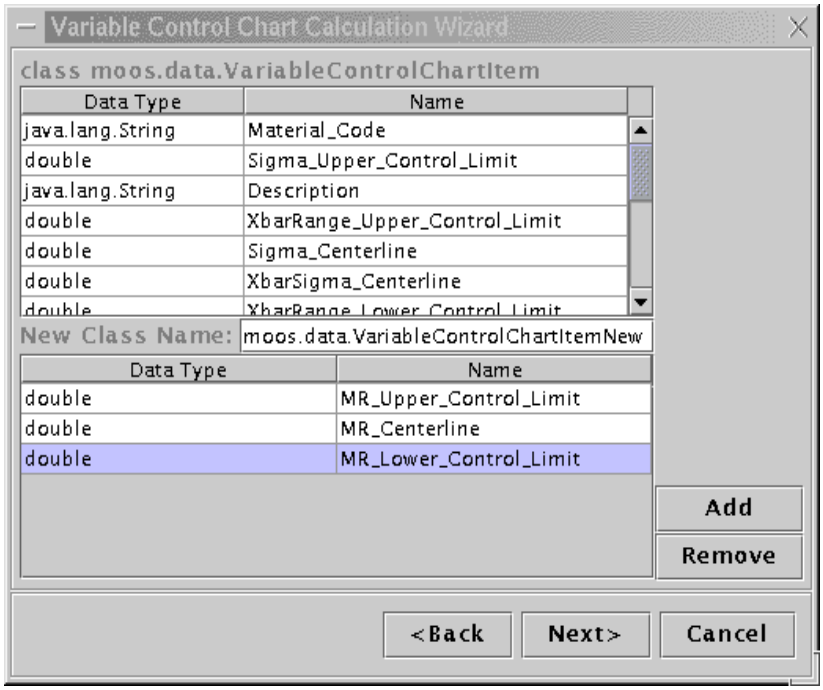

Figure C.2: Customisation of Class *moos.data.VariableControlChartItem*

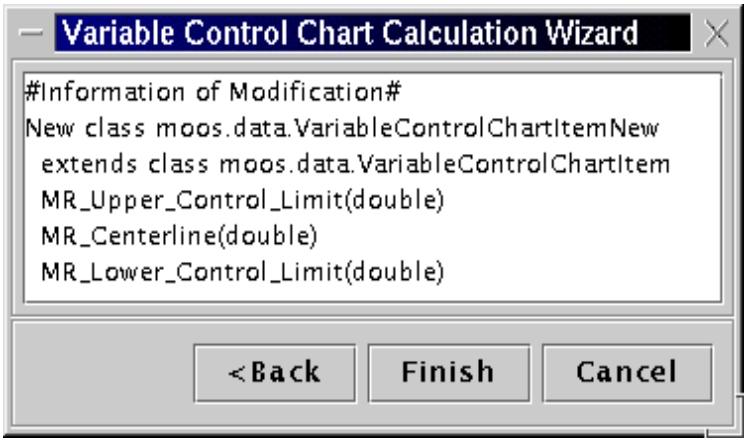

Figure C.3: Summary of Class *moos.data.VariableControlChartItemNew*

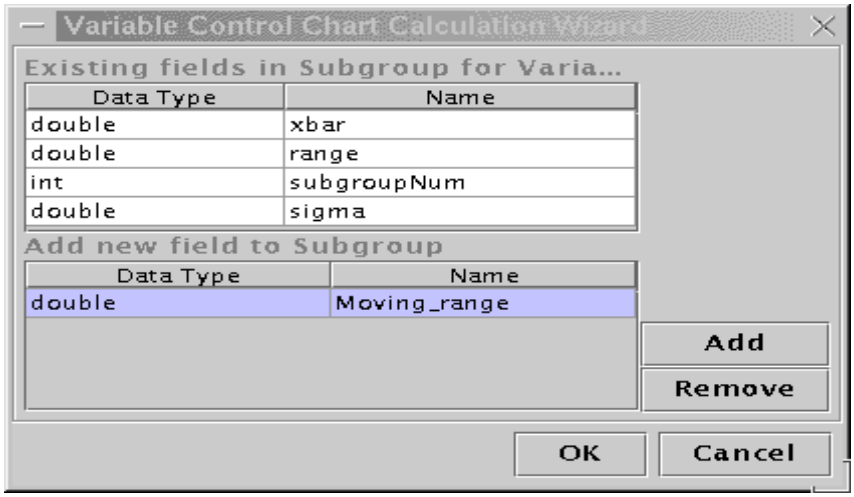

Figure C.4: Customisation of Class *moos.data.Subgroup*

After the customisation of the persistent classes, the IDE will generate the code based on the customisation. There are three files being generated as shown in Figure C.5. These files are *NewSubgroup.java*, *NewVariableControlChartController.java* and *VariableControlChartItemNew.java*. *NewSubgroup.java* and *VariableControlChartItemNew.java* are the new classes inherited from the persistent classes. *NewVariableControlChartController.java* provides the functions for the calculation of the three additional attributes for the class *VariableControlChartItemNew.* Developers can customise the functions to add their algorithm for the calculation. The source code for *VariableControlChartItemNew.java* is shown in Table C.1. The source code for *NewSubgroup.java* is shown in Table C.2. The source code for *NewVariableControlChartController.java* is shown in Table C.3.

The final steps for the customisation of *Variable Chart Calculation Module* is compilation. From the IDE menu, select **Build->Compile Project** menu. The compilation is using Java compiler provided by Sun Microsystem. After a clean compilation, select the **Build->Build Project** menu to create the project **jar** file. After this, plug-in the changes to the applet and server of SPC application framework. The steps for plug-in will be discussed

later.

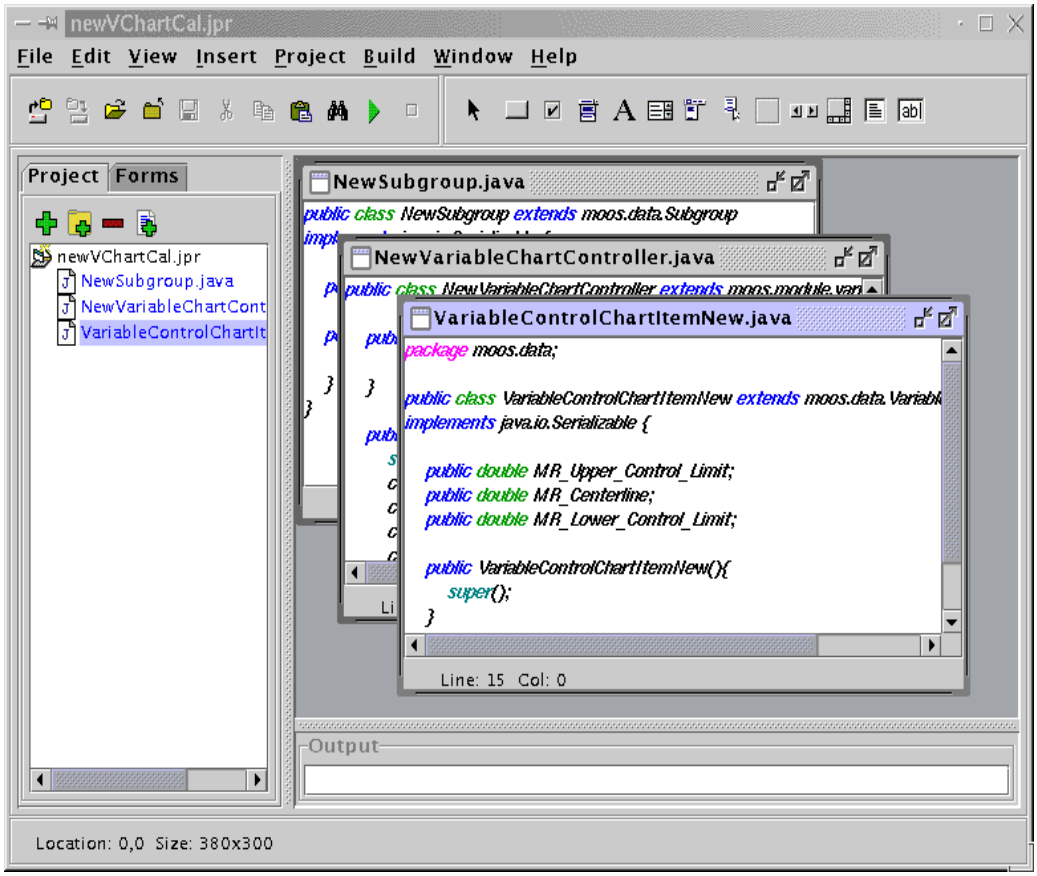

Figure C.5: File Generated from Variable Control Chart Wizard

```
package moos.data;

public class VariableControlChartItemNew extends moos.data.VariableControlChartItem
implements java.io.Serializable {

   public double MR_Upper_Control_Limit;
   public double MR_Centerline;
   public double MR_Lower_Control_Limit;

   public VariableControlChartItemNew(){
      super();
   }

}
```

Table C.1: Source Code for VariableControlChartItemNew.java

After the customisation of the *Variable Chart Calculation Module*, the

```
public class NewSubgroup extends moos.data.Subgroup
implements java.io.Serializable {

    public double Moving_Range;

    public NewSubgroup(){

    }
}
```

Table C.2: Source Code for NewSubgroup.java

new attributes and the calculation of these attributes has been implemented. The following module to be customised for chart plotting is *Variable Chart Plotting Module*. The first step for the customisation is to create a new project from the IDE by selecting the **File->New**. A "New" dialog box will appear as shown in Figure C.6.

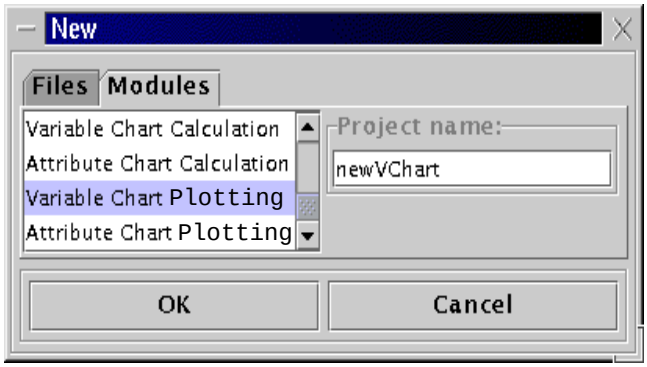

Figure C.6: Variable Chart Plotting Module New Dialog Box

After selecting the module and entering the project name, the varible chart plotting wizard will appear to customise the chart plotting as shown in Figure C.7. Click on the "Add" button to enter the chart name and chart class name in the customisation dialog box . The summary of the chart being added will appear after clicking on the "Next" button as shown in Figure C.8.

After adding the new chart, a new class called "MovingRange" which is inherited from chart base class is created. A Java source file called *MovingRange.java* is generated by the variable chart wizard as shown in Figure C.9.

```
public class NewVariableChartController extends
moos.module.variableChart.VariableChartController {

   public NewVariableChartController(){

   }

   public void calculation(moos.data.VariableControlChartItem vccitem){
      super.calculation(vccitem);
      calculationMoving_range(vccitem);
      calculationMR_Upper_Control_Limit(vccitem);
      calculationMR_Centerline(vccitem);
      calculationMR_Lower_Control_Limit(vccitem);
   }

   public void calculationMoving_range(moos.data.VariableControlChartItem vccitem){
      // write the code to calculate the field here
   }



    public void calculationMR_Upper_Control_Limit(moos.data.VariableControlChartItem
vccitem){
      // write the code to calculate the field here
   }

   public void calculationMR_Centerline(moos.data.VariableControlChartItem vccitem){
      // write the code to calculate the field here
   }

    public void calculationMR_Lower_Control_Limit(moos.data.VariableControlChartItem
vccitem){
      // write the code to calculate the field here
   }

}
```

Table C.3: Source Code for NewVariableControlChartController.java

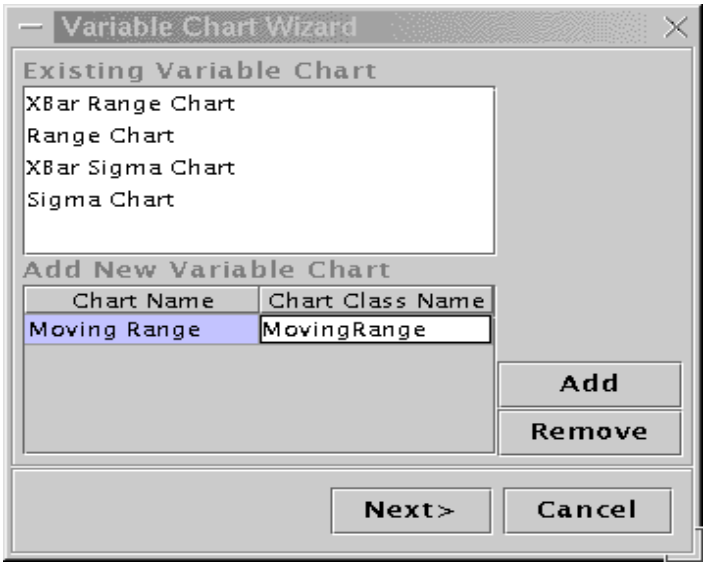

Figure C.7: Variable Chart Plotting Module Customisation

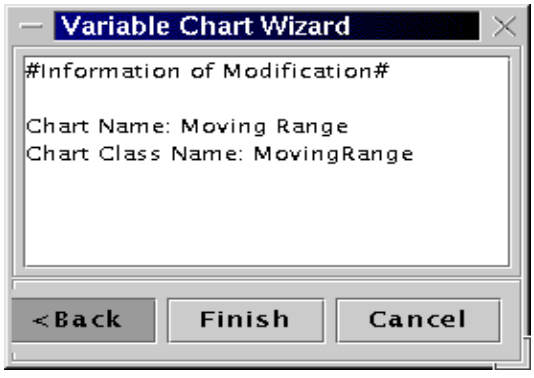

Figure C.8: Summary of Variable Chart Plotting Module Customisation

The source code generated is shown in Table C.4.

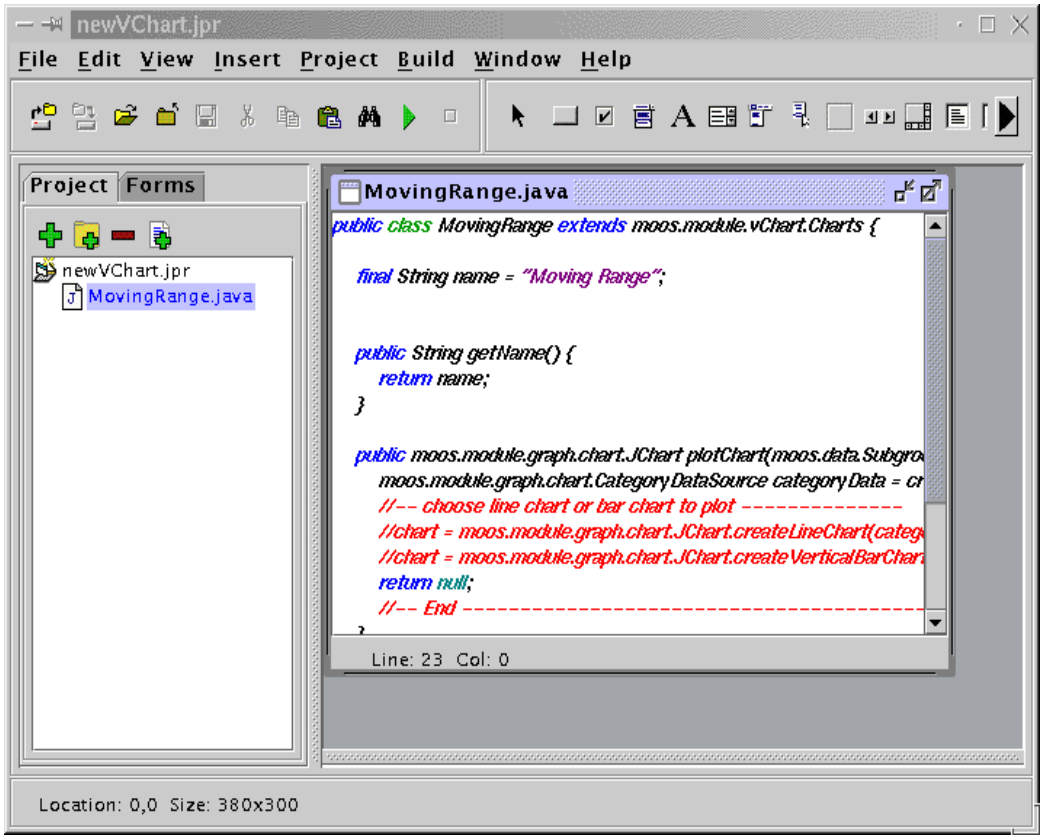

Figure C.9: File Generated from Variable Chart Wizard

The new attributes added in the customisation of *Variable Chart Calculation Module* is needed to plot the new MR chart. Thus, for the compilation to be successful, the project jar file from *Variable Chart Calculation Module* needs to be copied to this project.

After the customisation of the modules, an additional chart is added to the application framework. The new chart is *MR Chart.* As shown in the applet in Figure C.10, three new attributes, MR_Centerline, MR_Upper_Control_Limit and MR_Lower_Control_Limit appear at the Variable Control Chart Item field. From the module menu, the user can select the chart to plot after entering the data to the table below Variable Control Chart Item field. The chart selection as shown in Figure C.11 has the MR Chart which has been added. After the selection of chart, the chart is displayed as shown in Figure C.12.

```
public class MovingRange extends moos.module.vChart.Charts {

   final String name = "Moving Range";

   public String getName() {
      return name;
   }

   public moos.module.graph.chart.JChart plotChart(moos.data.Subgroup[] sub){
        moos.module.graph.chart.CategoryDataSource categoryData = createCategoryData-
Source(sub);
      //– choose line chart or bar chart to plot ————————
      //chart = moos.module.graph.chart.JChart.createLineChart(categoryData);
      //chart = moos.module.graph.chart.JChart.createVerticalBarChart(categoryData);
      return null;
      //– End ——————————————————————
   }
            public moos.module.graph.chart.CategoryDataSource createCategoryData-
Source(moos.data.Subgroup[] sub) {

      return null;
   }
}
```

Table C.4: Source Code for MovingRange.java

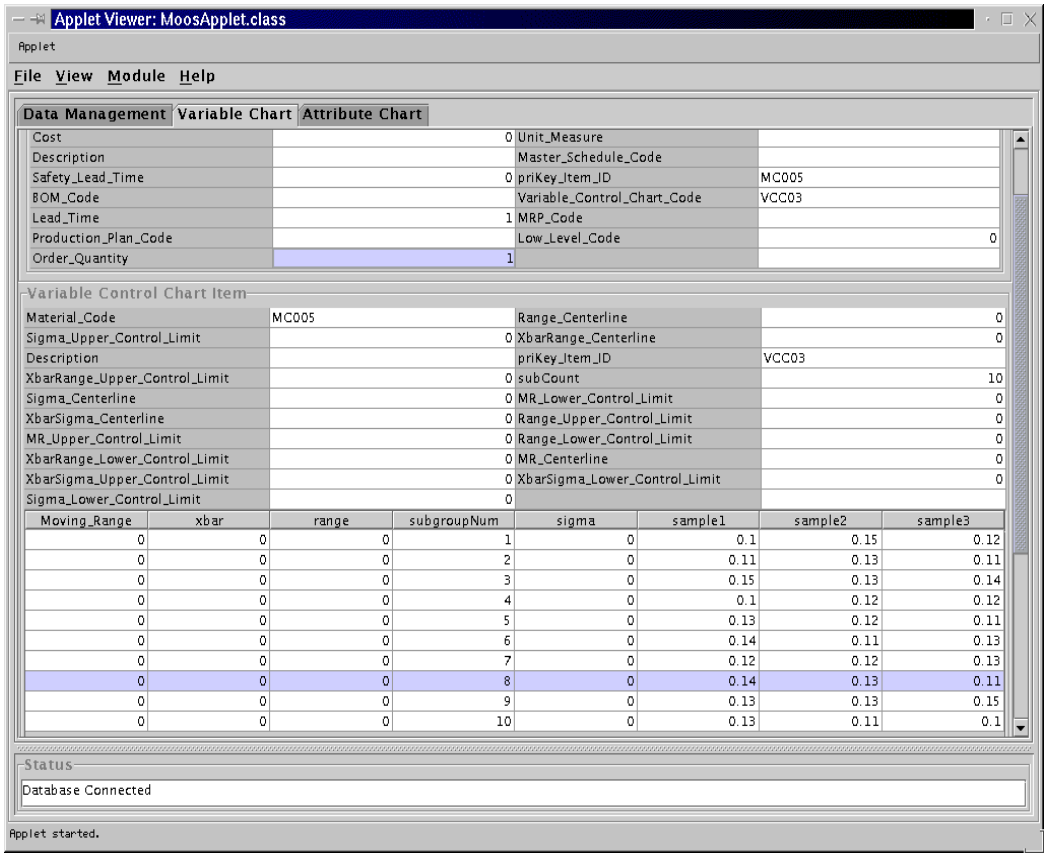

Figure C.10: Three New Attributes at the SPC Applet

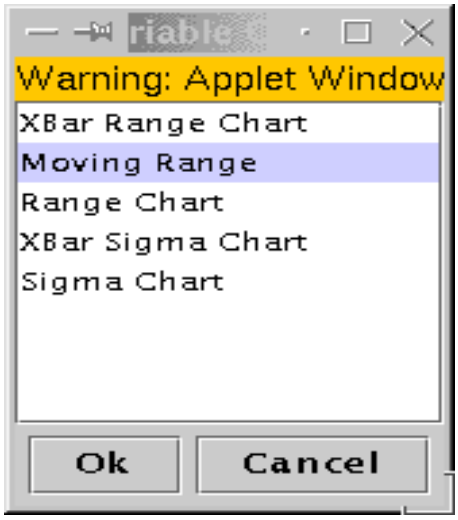

Figure C.11: Variable Chart Selection

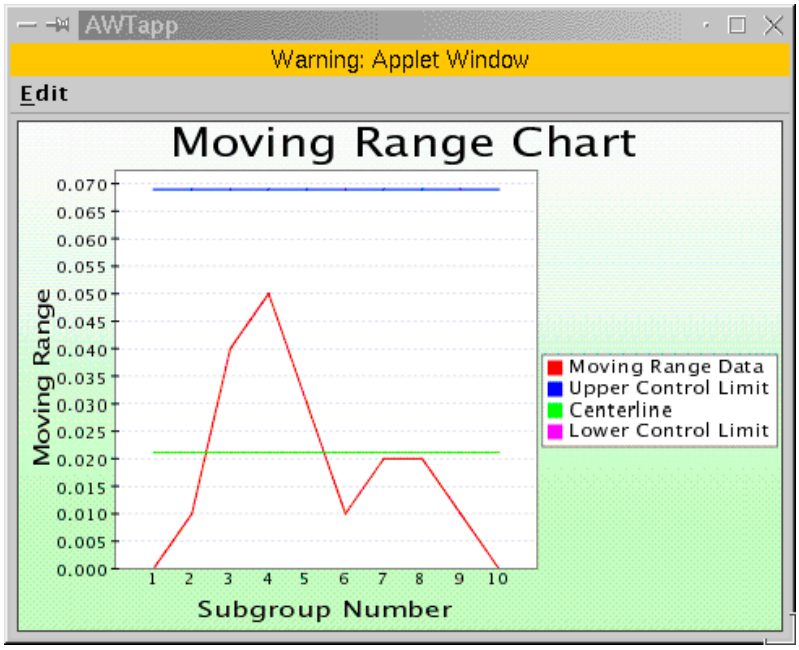

Figure C.12: Plotting of MR Chart

C.3 Plug-in to SPC Application Framework

There are two project jar files from the previous customisation that need to be plugged in to the application framework. The name of each project jar file follows the project name. They are newVChartCal.jar and newVChart.jar which are located at the project folder in the *ide* folder. Copy these two project jar files to the *applet* directory in Web server root directory. In order to load the correct plug-in, a configuration file called ClientModule.cfg and a html file called SPCApplet.html in the *applet* folder need to be modified.

Modification on the ClientModule.cfg:

Change this:

VariableChartModule = moos.module.variableChart.VariableChartModule

VChartModule = moos.module.vChart.VChartModule

to:

VariableChartModule = VariableChartModule

VChartModule = VChartModule

Modification on the SPCApplet.html:

Change the entry in the html to:

<PARAM NAME="archive" VALUE="MoosApplet.jar, newVChartCal.jar, newVChart.jar">

.....

archive="MoosApplet.jar, newVChartCal.jar, newVChart.jar"

.....

For the server plug-in, copy the two project jar files to server folder and modify the run script to bring up server.

The run script is now:

java -classpath MoosServer.jar:newVChartCal.jar:newVChart.jar <connection string> <user> <password>

References

Abadi, M. & Cardelli, L. (1996), 'A theory of objects'. Springer.

Ahamed, S. I., Pezewski, A. & Pezewski, A. (2004), Towards framework selection criteria and suitability for an application framework, *in* 'Information Technology: Coding and Computing, 2004', Vol. 1, Proceedings. ITCC 2004. Internation Conference, pp. 424–428.

Bellinzona, R., Fugini, M. G. & Pernici, B. (1995), 'Reusing specifications in OO applications', *IEEE Software* **12**(2), 65–75.

Biggerstaff, T. & Richter, C. (1989), 'Reusability framework, assessment and directions'. ACM Press.

Booch, G. (1994*a*), *Object-oriented Analysis and Design*, Menlo Park Ca., Addison-Wesley.

Booch, G. (1994*b*), *Object-oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Company, Inc.

Bosch, J., Molin, P., Mattson, M. & Bengtson, P. (1997), Object-oriented frameworks - problems & experiences, *in* 'Proceedings of the 23rd International Conference in Technology of Object-Oriented Languages and Systems, TOOLS '97 USA', Santa Barbara, California, pp. 203–214.

Chan, S. M. & Lammers, T. L. (1997), Creating a distributed factory object architecture, *in* 'Proceedings of the 1st Enterprise Distributed Object

Computing (EDOC 1997) Workshop, IEEE Computer Society Press', Los Alamitos, CA, pp. 282–290.

Chiang, C.-C. (2003), Development of reusable component through the use of adapters, *in* 'System Sciences, 2003', Proceedings of the 36th Annual Hawaii International Conference, pp. 319–328.

Coad, P. & Yourdon, E. (1990), 'Object-oriented analysis'. Englewood Cliffs, New Jersey, Yourdon Press.

Crnkovic, I. & Larsson, M. (2000), A case study: Demands on component-based development, *in* 'Software Engineering, 2000', Proceedings of the 2000 International Conference, pp. 23–31.

Demeyer, S., Meijler, T., Nierstrasz, O. & Steyaert, P. (1997), 'Design guidelines for 'tailorable' frameworks', *Commnications of the ACM* **40**(10), 60–64.

Fayad, E. M. & Schmidt, D. C. (1997), 'Object-oriented application frameworks', *Communications of the ACM* **40**(10), 32–38.

Firesmith, D. (1994), 'Frameworks: the golden path to object nirvana', *Journal of Object-Oriented Programming* **7**(8), 6–8.

Gamma, E., Helm, R., Johnson, H. & Vlissides, J. (1995), *Design Patterns - Elements of Reusable Object-Oriented Software*, Reading Mass., Addisson-Wesley.

Greenfield, J. & Short, K. (2003), Software factories: assembling applications with patterns, model, frameworks and tools, *in* 'Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, language, and applications', Anaheim, CA, USA, pp. 16–27.

Hennicker, R. & Koch, N. (2001), *System design of web application with UML*, Idea Publishing Group.

Henry, E. & Faller, B. (1995), 'Large-scale industrial reuse to reduce cost and cycle time', *IEEE Software* **12**(5), 47–53.

Hsiung, P.-A., Lee, T.-Y., See, W.-B., Fu, J.-M. & Chen, S.-J. (2002), Vertaf: An object-oriented application framework for embedded real-time systems, *in* 'Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002)', Proceedings. Fifth IEEE International Symposium, pp. 322–329.

Jacobson, I., Griss, M. & Jonsson, P. (1997), *Software Reuse Architecture, Process and Organization for Business Success*, Addison Wesley Longman Limited.

Johnson, R. & Foote, B. (1988), 'Designing reusable classes', *The Journal of Object-Oriented Programming* **1**(2), 22–35.

Johnson, R. & Russo, V. (1991), 'Reusing object-oriented designs'. University of Illinois tech report UIUCDCS 91-1696.

Khajenoori, S. & Linton, D. (1994), 'Enhancing software reusability through effective use of the essential modelling approach', *Integrated Manufacturing Systems* **36**(8), 495–501.

Khor, C. H. (2000), 'Persistence framework for multiple legacy databases of patient information systems'. Master thesis, Faculty of Computer Science And Information Technology, University of Malaya.

Kotani, M. & Gakuin, A. (2003), Enterprise software portfolio management:: An IT decision-making framework, *in* 'Information Science + IT Education Conference', Pori, Finland, pp. 257–263.

Lam, W. (1997), 'The development of very high-level components for software engineering', *In, IASTED International Conference Software Engineering* pp. 49–52.

Larman, C. (1997), *Applying UML and Pattern*, Prentice Hall, Inc.

Lee, S. P., Thin, S. K. & Liu, H. S. (1999), Object-oriented application framework on manufacturing domain, *in* 'International Conference on Information Technology', Sunwary Pyramid Exhibition, Convention Centre.

Lee, S. P., Thin, S. K. & Liu, H. S. (2000*a*), 'Object-oriented application framework on manufacturing domain', *Malaysian Journal of Computer Science* **13**(1), 56–64.

Lee, S. P., Thin, S. K. & Liu, H. S. (2000*b*), Object-oriented manufacturing application framework, *in* 'Proceedings of 34th International Conference and Exhibition on Technology of Object-oriented Languages and Systems (TOOLS-USA 2000), IEEE Computer Society Press', Santa Barbara, USA, pp. 253–262.

Lee, S. P., Thin, S. K. & Liu, H. S. (2001*a*), EMAF: An enterprise manufacturing framework integrated environment, *in* 'The Pacific Asia Conference on Information Technology 2001', Information Technology for e-Strategy, Yonsei University.

Lee, S. P., Thin, S. K. & Liu, H. S. (2001*b*), An enterprise manufacturing framework integrated environment for manufacturing application development, *in* 'Proceeding of RM6K7 IRPA Research Seminar', University of Malaya, pp. 174–177.

Lim Wayne C. (1994), 'Effects of reuse on quality, productivity and economics', *IEEE Software* **11**(5), 23–30.

Morel, B. & Alexander, P. (2003), Automating component adaption for reuse, *in* 'Automated Software Engineering, 2003', Proceedings. 18th IEEE Internation Conference, pp. 142–151.

Opdyke, W. F. (1992), 'Refactoring object-oriented frameworks'. PhD thesis, University of Illinois at Urbana-Champaign.

Philippow, I. & Riebisch, M. (2001), Systematic definition of reusable architectures, *in* 'Engineering of Computer Based System, 2001. ECBS 2001', Proceedings. Eighth Annual IEEE Internation Conference and Workshop, pp. 128–135.

Probst, S., Essmayr, W. & Weippl, E. (2002), Reusable component for developing security-aware application, *in* 'Computer Security Application Conference, 2002', Proceedings. 18th Annual, pp. 239–248.

Schmid, H. A. (1997), 'Systematic framework design by generalization', *Communication of the ACM* **40**(10), 48–51.

Shirland, L. E. (1993), *Statistical Quality Control with Microcomputer Applications*, John Wiley & Son, Inc.

Taligent Inc. (1998), 'Leveraging object-oriented frameworks', *A Taligent White Paper* .

(Abadi & Cardelli 1996, Biggerstaff & Richter 1989, Booch 1994*b*, Booch 1994*a*, Bosch et al. 1997, Coad & Yourdon 1990, Larman 1997, Demeyer et al. 1997, Henry & Faller 1995, Fayad & Schmidt 1997, Firesmith 1994, Gamma et al. 1995, Hennicker & Koch 2001, Jacobson et al. 1997, Greenfield & Short 2003, Johnson & Foote 1988, Johnson & Russo 1991, Khajenoori & Linton 1994, Khor 2000, Shirland 1993, Lee et al. 1999, Lee et al. 2000*b*, Lee et al. 2000*a*, Lee et al. 2001*a*, Lee et al. 2001*b*, Lim Wayne C. 1994, Kotani & Gakuin 2003, Opdyke 1992, Bellinzona et al. 1995, Chan & Lammers 1997, Schmid 1997, Taligent Inc. 1998, Lam 1997, Philippow & Riebisch 2001, Ahamed et al. 2004, Hsiung et al. 2002, Crnkovic & Larsson 2000, Morel & Alexander 2003, Chiang 2003, Probst, Essmayr & Weippl 2002)