What Works and What Does Not: An Analysis of Application Frameworks Technology

Wusheng Zhang
Victoria University, Australia

Mik Kim
RMIT University, Australia

Abstract

Advocates of application frameworks claim that this technology is one of the most promising, supporting large-scale reuse, increased productivity and quality, and reduced cost of software development. A number of its advocates suggest that the next decade will be a major challenge for the development and deployment of this technology. This study investigates the theory and practice of application frameworks technology to evaluate what works and what does not in systems development. The evaluation is based on quality criteria developed by the authors. The result of the study suggests that application frameworks technology does support large-scale reuse by incorporating other existing reuse techniques such as design patterns, class libraries and components. It also shows that the methodological support pertaining to building and implementing application frameworks is inadequate. Furthermore, it indicates that application frameworks technology may increase the quality of software in terms of correctness and reusability with some penalty factors but there is no guarantee of increasing the extendability and interoperability of software systems. There are still obstacles that restrict the potential benefits claimed by the proponents of application frameworks.

Keywords: *application frameworks technology, systems development, evaluation*

Introduction

Software development markets expect developers or development companies to deliver quality products at an affordable price within a required time frame. Developers and management alike are looking for technologies that can be used to increase the productivity and quality of software products. Mature engineering disciplines such as automobile design, have proven that reuse is the best way to increase the quality and productivity of products. However, despite the efforts of decades-long research the result of software reuse is still limited to code or class reuse (also known as small-scale), and developers are still 'reinventing the wheel'. Application framework is a technology anchored in this situation to promote reuse in terms of not only the code or class but also the module and architecture (also known as large-scale) of the reusable software artefacts to increase software productivity and quality. The notion of application frameworks appeared at the end of the 1980s. MacApp is one of the first user interface application framework designed specifically for implementing Macintosh applications in later 80s (Fayad, 2000b). Application frameworks has become a popular research topic during the 1990s. Numerous frameworks have been defined including domain independent frameworks such as Java Swing, Microsoft Foundation Class (MFC), graphical editors such as HotDraw and domain specific frameworks such as IBM's San Francisco framework.

The purpose of the study was to investigate the theory and practice of application frameworks technology to evaluate this technology in relation to the quality of software developed from application frameworks. In this paper the authors first discuss definitions and various classifications of application frameworks. Next are discussed the theoretical foundations of application framework technology including object technology and other reuse technologies that have an important role in the development of application frameworks. Then, the quality criteria constructed by the authors will be introduced to evaluate application frameworks more systematically. After that the results of the evaluation will be illustrated.

What is an Application Framework?

The common sense of the word of framework appears to be "a skeleton of another structure", which has been well adopted into the context of modern information systems development. Booch, Rumbaugh and Jacobson (1999) define a framework as "an architectural pattern that provides an extensible template for an application within a domain". In this context a framework is essentially a design skeleton that allows systems developers to create part of a system in the first place, and add design details when necessary. Johnson (1997) states that the definitions of frameworks vary, but the one used most is that "a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact". Another common definition is "a framework is the skeleton of an application that can be customized by an application developer". The former concerns the structure of a framework while the latter describes the purpose of the framework. Lewis (1995) argues that a framework is more than a class hierarchy. Fayad (1997) claims that a framework is a reusable, 'semi-complete' application that can be specialised to produce custom applications. Zamir (1999) defines "an object-oriented framework as the reusable design of a system or subsystem implemented through a collection of concrete and abstract class and their collaborations. The concrete classes provide the reusable components, while the design provides the context in which they are used." The concepts of frameworks and application frameworks are often used interchangeably in the context of systems development.

Although the definitions by different researchers vary, some of them are more abstract and concerned more with the analysis and design phase, while others are more interested in the design and development phase. The different emphases does not conflict each other but rather enrich and enlighten further research issues related to the field of application frameworks technology. An application framework initially is a semi-completed application with architectural structure, which can be implemented and customised by application developers to develop application software. Application frameworks design can be bottom-up and pattern driven or top-down and target driven (Schmid 1995, Szyperski1997 and Fayad 2000). The bottom-up design works well where an application framework domain is already well understood. Starting from proven patterns and working one's way up has the advantage of avoiding idiosyncratic solutions in the small, problematic solutions that should be replaced by application of an established pattern. A top-down and target driven approach is preferable where an application framework domain has not yet been sufficiently explored but where the target domain to be served by the framework is well understood.

Classifications of Application Frameworks

The application frameworks can be domain independent such as a graphical user interface (GUI) framework, or domain dependent or specific such as a CIM framework. Here, the word *domain* refers to business areas being applied for implementing the application frameworks. For example, if the application frameworks are used in a domain from the financial sector the application frameworks are domain specific to the financial application, or if the application frameworks are used in a domain of the manufacturing sector the application frameworks are domain specific to the manufacturing application. They can also be classified according to the scope, reuse perspective, the control aspect and the development process of application frameworks. **Figure 1** shows A summary of different classifications

used in current literature. It illustrates classifications from different perspectives. According to the scope of application frameworks, Fayad (2000) proposes to classify this into three categories namely: system infrastructure frameworks such as graphical user interface (GUI) and Microsoft Foundation Class (MFC); Middleware integration frameworks such as BEST and JAWS; and Enterprise

From the perspective of	Classifications
Domain	Domain independent
	Domain dependent or specific
Scope	Infrastructure frameworks
	Middleware integration frameworks
	Enterprise application frameworks
Reuse	Whitebox
	Blackbox
	Greybox
Control	Callable framework
	Calling framework
Development process	Analysis framework
	Design framework
	Implementation framework

Figure 1: The classification of frameworks

application frameworks such as SEMATECH CIM, OSEEFA and PRM. He claims (1999a) that the application frameworks are generally domain specific applications such as computer-integrated manufacturing frameworks, distributed systems, networking and telecommunications, or multimedia collaborative work environments.

From the perspective of reuse the application frameworks can be classified into whitebox, blackbox, and greybox frameworks (Szyperski 1997, Fayad, 2000). A whitebox application framework is a framework customised by subclassing existing framework classes and providing concrete implementations. To implement a whitebox framework, application developers use more inheritance and polymorphism. Application specific functionality is expressed by inheritance and new implementations. Implementation inheritance tends to require knowledge of the superclasses' implementations. In the last few years the application frameworks researchers are more interested to develop blackbox application frameworks which rely more on composition rather than inheritance. In the blackbox frameworks approach, the extendability of the framework is achieved by defining an interface for components that can be plugged into the framework using composition. Object composition is based on forwarding rather than delegation, merely relying on the interfaces of the involved objects. In a blackbox framework (Fayad 2000), an application developer selects from the set of subclasses provided by the framework as the blackbox components and binds it to the hot spot (plug in point). Thus, the developer may create an application without programming, merely by selecting, configuring, and parameterising framework components. A greybox approach is a combination of both the whitebox and the blackbox frameworks.

In view of taking control, the application frameworks can be classified as callable frameworks and calling frameworks (Fayad 2000). A callable framework allows the application to retain the thread of control and provides services when the application calls the frameworks. A calling framework provides a control loop that calls application-provided code at appropriate times. From a development process perspective the application frameworks can be divided into analysis frameworks, design frameworks, and implementation frameworks. The analysis frameworks typically focus on analysis level constructs, without making any commitment. They are typically the product of domain analysis. Most current application frameworks are either a design framework or an implementation framework. Largely the application frameworks are domain specific such as a financial application framework or a manufacturing framework. An application framework domain is a set of rules and roles and their semantic models codified in the framework itself. It provides a generic incomplete solution to a set of

similar problems within an application domain. Fayad (2000) states that an application framework embodies generalised expertise in the domain based on analysis and synthesis of a wide range of specific solutions. He argues that analysis and synthesis of a wide range of specific solutions will help to understand a design of the proposed application framework. It is shown that the research community has more understanding in some domains such as financial, manufacturing, communication and networks and social welfare than others (Eliens 2000, Fayad 2000).

Object Oriented Technology in Application Frameworks

The development of the application frameworks research is related to the development of object technology although there is no evidence that the framework technology is exclusive to object technology. However, the majority of the researchers in the area of application frameworks and most current application frameworks are object-oriented. Object-oriented technology is one of the fastest growing technologies of the last two decades promising better quality, productivity and interoperability through software reuse. Coad and Yourdon (1990) define "an object is an abstraction of something in a problem domain, reflecting the capabilities of the system to keep information about it, interact with it or both". In that sense objects are used to model an understanding of the application domain, which concerns the system and abstraction. Deitel (2003) defines "Object technology is a packaging scheme that facilitates the creation of meaningful software units". He explains that these units are usually large and focused on particular application areas and most of them can be reused (Deitel 2003). For example, there are data objects, time objects, audio objects, video objects, file objects, record objects and so on.

Iterative and incremental development approaches adopted in object-oriented technology have been the main development methodology supporting the development of application frameworks (Fayad, 1999a). Although the research of application frameworks is not exclusive to the object-oriented community, object-oriented technology has been the main driving force in the area of application frameworks. Advocates of application frameworks claim that the technology is one of the most promising technologies supporting large-scale reuse, increasing the productivity and quality, and reducing the cost of software development. Fayad (2000b) suggests that the primary benefit of object-oriented application frameworks stems from the modularity, reusability, extendability, and inversion of control they provide to developers. Many researchers and academics (i.e., Lewis et al, 1995, Eliens 2000, Fayad & Johnson, 2000, Due 2002) have argued that a major challenge for the next decade will be to develop and deploy application frameworks that operate in areas such as finance, medical care, insurance and telecommunication and networking. On the contrary, one survey (Cockburn 1997) shows that object-oriented approaches at frameworks development have failed more often than they have succeeded.

The central idea of object-oriented technology subsumes abstraction, modularity, encapsulation, inheritance and polymorphism - concepts that, on the face of it, lend themselves to reuse. The notable development of the technology consists of a comprehensive set of object-oriented modelling methods for analysis, design, and implementation, designed to realise the concepts mentioned above. Object-oriented technology has led to the development of patterns, components and application frameworks and object-oriented concepts have been applied in the process of developing and implementing application frameworks. Fayad (1999a, 1999b, 2000) stresses that frameworks build upon object-oriented concepts, which provides a conceptual base for more complex programming constructs and reusable implementation structures for large systems application. Eliens (2000) states that an object oriented approach will pay off when we have arrived at stable abstractions from which we have good implementations that may be reused for a variety of other applications. Accordingly, it can be said that application frameworks is a technology aimed to achieve large-scale reuse by applying object-oriented concepts. In the following sections some of the object-oriented concepts and principles will be discussed in relation to application frameworks and systems development.

Abstraction

Abstraction is one of the principal concepts of object-oriented technology and aims to reduce details required for implementing software systems. Microsoft Encarta Dictionary (2001) defines abstraction as

"to develop a line of thought from a concrete reality to a general principle or an intellectual idea; a concept or term that does not refer to a concrete object but that denotes a quality, an emotion, or an idea." A closer working definition defined by Graham (2001) is that "representing the essential features of something without including background or inessential detail." It stresses separation of the essential features and details. Abstraction is a powerful tool available to software developers and most of modern object oriented languages support the notion. For example, in the Jade language, a pure object-oriented development environment has an abstract class called *object*, which can be inherited by application developers to add their own classes. An abstract class, such as the *object class,* denoted with no instances, is often used to represent abstract concept, whose concrete subclass may add its structure and behaviour by implementing its abstract method. Within an inheritance hierarchy, it is likely that some of the topmost classes may contain features whose definitions are differed from the subclasses. In other words, there are no implementation details for these features within the super class. This type of class is subsequently known as an abstract class. Szyperski (1997) states that an abstract class is a class that cannot be instantiated, that is, no object can be a direct instance of an abstract class. An abstract class can have unimplemented methods/abstract methods. Concrete classes inheriting from an abstract class have to implement all such abstract methods. An ideal abstraction should encapsulate all the essential properties of an object, including data and processes. The main benefit of an abstraction is the design expertise embodied in it, ready for reuse (Szyperski 1997). Application frameworks are designed for the purpose of supporting large-scale reuse, therefore abstraction is a built-in notion in the application frameworks development paradigm.

Generalisation and specialisation

Generalisation describes the logical relationship between elements that share some characteristics or say it describes the grouping of objects that have a common set of properties and operations. Fowler (1997) defines generalisation as a taxonomic relationship between a more general element and a more specific element that is fully consistent with the general element and that adds additional information. Specialisation is the refinement of an abstraction by adding additional features. Generalisation and specialisation hierarchy is one of the most powerful tools of abstraction used in object-oriented modelling, which allows representing taxonomic relationships among classes (Bruel, 2002). The relationship between generalisation and specialisation allows us not only to classify objects, but also to use the generalisation and specialisation hierarchy. An object oriented approach uses generalisation and specialisation techniques to realise abstraction. An application framework is a skeleton of the structure for a system, and the classes within the framework contain a common set of properties and operations for the domain area described by the framework. It is a generic solution for a bushiness domain. An application developed by implementing an application framework is a specialisation of the framework in which the application developers specialise the classes in their intended applications by inheritance or composition.

Modularity

Zamir (1999) defines a model as a distinctively named and addressable element of software used as a building block for the physical structure of a system, and modularity as the characteristic of a system decomposed into a collection of cohesive and loosely coupled modules, typically a goal of systems analysis and design. Modularity has been the principle for many matured engineering disciplines. The importance of modularity has been emphasised in many of the writings of software theoreticians. Meyer (1988) and Graham (2001) state that a good model should have decomposability (- refers to the software engineering and project management requirement where systems be decomposable into manageable chunks so they can be changed more easily and so that individuals or teams can be assigned to coherent work packages), composability (- refers to the property of modules to be freely combined even in systems for which they were not developed), understandability (- helps people to comprehend a system by looking at its parts prior to gaining an understanding of the whole), continuity (- in a system implies both that small changes made to it will only result in small changes in its behaviour, and that small changes in the specification will require changes to only a few modules) and protection (- the criterion of modular protection insists that exception and error conditions either remain confined to the

module in which they occur or propagate to only a few other closely related modules). Fayad (1999a) states that modularity is one of the main benefits that application frameworks can offer to application developers. He argues that application frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. He (Fayad 2000) also suggests that extensive data coupling in a whitebox framework may break sound modularity and therefore, encourage people moving towards blackbox frameworks or greybox frameworks.

Encapsulation

This is one of the important concepts and the mechanisms to support the need for software reuse and security. Zamir (1999) defines encapsulation as the mechanism by which related data and procedures are bound together within an object. In effect, an object is a software capsule that functions as a blackbox, responding to messages from other objects and dispatching messages of its own in ways that do not reveal its internal structure. Encapsulation is the practice of hiding the data structure that represents the internal state of an object from access by any other than the public methods of that object. This can ensure that objects cannot change the internal state of other objects in unexpected ways, minimising the complexity of putting together modules of code from different sources. This is a programming facility used in object-oriented programming practice. Encapsulation is the technique for packaging the information in such a way as to hide what should be hidden and make the visible what is intended to be visible. The use of encapsulation is a powerful means of maintaining control over an object's data and state. It allows an object to determine whether and how data may be changed. This makes it possible to modify or enhance an object's implementation while keeping its exposed interfaces consistent, preventing backward-compatibility problems as the programs develop. Encapsulation promotes modularity, meaning that the object must be regarded as a building block of a complex system. Once a proper modularisation has been achieved, the implementer of the object may postpone any final decisions concerning the implementation at will. Application frameworks relies on the capability of encapsulation, in which the framework can hide the internal structures but allows application developers to use the functions defined via interfaces.

Polymorphism

Graham (2001) defines polymorphism as the ability to use the same expression to denote different operations. Many modern programming languages support polymorphic behaviour. Sometimes polymorphism is referred to as dynamic binding or runtime binding of function calls. Object-oriented programming languages derive most their power from inheritance and runtime binding of function calls. Application frameworks, especially whitebox frameworks, rely on polymorphism (dynamic binding) and inheritance to allow application developers to implement the framework.

Most current application frameworks apply object-oriented concepts and principles. Many notable projects in the application frameworks area are based on object-oriented technology such as San Francisco, OSEFO and SEMATECH CIM. Schmid (1996) argues that the primary benefit of an object oriented approach for application frameworks stems from the emphasis on modularity and extendability by encapsulating volatile implementation details behind stable interface and enhancing software reuse. Application frameworks is built upon the objects technology that is more likely to provide a conceptual base for complex programming constructs and reusable implementation structures. Additionally, object oriented technology provides the mechanisms needed for application frameworks such as inheritance, encapsulation and polymorphism.

Other Reuse Techniques and Application Frameworks

Application frameworks is a reuse technology aimed at large-scale reuse and it has a close relationship with other reuse techniques used in software engineering. An application framework can be seen as a collection of components, a generic solution for a class of problems, a frame of mind for solving problems and a set of architectural constraints. It integrates and concretises a number of patterns to a degree required to ensure proper interleaving and interaction of participants involved. An application

framework can also be seen as a kind of library, which provides reusable objects for applications. But in contrast to ordinary software class libraries, frameworks may at times take over control when the application runs. From a reuse perspective the application frameworks technology is closely related to other reuse techniques. Application frameworks use those reuse techniques to achieve the goal of large-scale reuse. As the reuse techniques have an important role in developing frameworks technology, understanding the specific technology is a stepping-stone for grounding evaluation criteria. Following, the foundational techniques (i.e., architecture, class libraries, patterns, and components) are explained:

Architecture

Software architecture is the foundation of system construction. Graham (2001) points out that software architecture deals with abstraction, with composition and decomposition, and also with style and aesthetics. Bass (1998) describes the software architecture of a program or computing system as the structure or structures of the systems, which comprise software components, the externally visible properties of those components and the relationships among them. Szyperski (1997) depicts system architecture as a means to capture an overall generic approach that makes it more likely that concrete systems following the architecture will be understandable, maintainable, evolvable, and economic. It is this integrating principle, covering technology and market that links software architecture to its great role model and justifies its name. Despite the different concentration of the definitions, software architecture is about an overview of a system. Generally speaking, software architecture can be seen as a set of rules, guidelines, interfaces, and conventions used to define how components and applications communicate and interoperate with each other. Recent software development experience has shown that sound software architecture for the software systems is necessary as software systems are more complex than before. Szyperski (1997) stresses that architecture prescribes proper frameworks for all involved mechanisms, limiting the degree of freedom to curb variations and enable cooperation. Architecture needs to be based on the principal considerations of overall functionality, performance, reliability, and security. Software engineers have learnt from practice such that architecture is needed in any system if they seek for guiding rules for design and implementation.

Architecture needs to create simultaneously the basis for independence and cooperation of systems. Independence of the systems aspect is required to enable multiple sources of solution parts. Cooperation between these otherwise independent aspects is essential in any no-trivial architecture. System architecture is the structure of a software system which provides a platform for application developers to build the system. It may be as concrete as providing detailed implementation requirements, to as abstract as giving a generic idea of how the system should be implemented. Application frameworks technology promises reuse of not only the frameworks source codes, but also more importantly, architecture (Fayad 1999a). A standardisation structure allows a significant reduction of the size and complexity of codes that application developers have to write.

Class libraries

These are a set of reusable classes, often defined as part of the implementation or design environment (Zamir 1999). Many programming languages have some ready usable classes embedded and available to application developers especially in visual development such as VB Studio.Net and J2EE. Class libraries in general offer static inheritance facilities but frameworks are more likely to support dynamic, run time binding facilities. Application frameworks defines ‘semi-complete’ applications that embody domain specific object structures and functionality. It can be viewed as extensions to object oriented class libraries. In contrast, class libraries provide a smaller granularity of reuse. For example, class library components like classes for strings, complex numbers and arrays are typically low-level and more domain-independent. Fayad (2000b) states that class libraries are typically passive and frameworks are active and exhibit ‘inversion of control’ at runtime.

Patterns

Classes and interaction structure of object-oriented designs may become fairly complex, and consequently difficult to develop and understand, which has led the study and development of patterns.

Design patterns are standard solutions to recurring problems, named to help people discuss them easily and think about design. Design patterns can be used as a micro-architecture that applies to a cross-domain design problem such as linked list and other classical data structure design. A design pattern describes a concrete solution to an architectural problem that might arise in a specific context. The solution proposed by the patterns is typically a way of structuring a cluster of objects and their interaction (Brugali et al. 2000). Schmid (1995) states that the repetitive use of design patterns created an overall architecture though each design pattern represents a micro architecture. He argues that design patterns give a better performance with more concrete guidance on how to realise a framework. Patterns are abstract, therefore they are not ready-made pluggable-solutions. They are most often represented in object-oriented development by commonly recurring arrangements of classes and the structural and dynamic connections between them. Graham (2001) argues that patterns are most useful because they provide a language for designers to communicate in. In particular, design patterns have proven their value in structuring the variable parts, called hot spots (allowing plug in software artefacts) of a framework (Pee, 1994). Fayad (2000) defines patterns as a conceptual solution to a recurring problem. Schmid (1995) argues that design patterns are an excellent means to describe the details of object and class interactions but they are not suited to give an overall picture. Design patterns are reusable architecture, object template, or design rule that has been shown to address a particular issue in an application domain (Zamir 1999). Most design patterns come either as a static description of a recurring pattern of architectural elements or as a rule to apply dynamically for when and how to apply the pattern. The majority of software patterns produced to date have been design patterns at various levels of abstraction but Fowler (1997, Graham 2001) introduces the idea of analysis patterns as opposed to design patterns. Fowler's patterns are reusable fragments of an object-oriented specification model generic enough to be applicable across a number of specific application domains.

Both patterns and frameworks facilitate reuse by capturing successful software development strategies. The primary difference is that frameworks focuses on reuse of concrete designs, algorithms, and implementations in a particular programming language. In contrast, patterns focus on reuse of abstract designs and software architectures. Frameworks can be viewed as a reification of families of design patterns. Likewise, design patterns can be viewed as the micro architectural elements of frameworks that document and motivate the semantics of frameworks in an effective way (Fayad, 2000b). Design patterns have been used extensively in developing application frameworks. Many researchers (Schmid 1995, Fayad 1999a, Fayad 2000) have suggested using as many patterns as possible for developing application frameworks because the abstractness and design expertise are embedded in patterns.

Components

Szyperski (1997) points out that component technology is standalone, which has gone beyond object orientation. He defines software components as binary units of independent production, acquisition, and deployment that interact to form a functioning system. In this definition a software component is best thought as a unit with well-defined interfaces that has explicit context dependencies. He explains that insisting on independence and binary form are essential to allow multiple independent vendors and robust integration. Components are not just a big object. Eliens (2000) notes that components usually consist of a collection of objects that provide additional functionality that allows components to interact together. Szyperski (1997) states that a component is a unit of independent deployment, a unit of third party composition, and it has no persistent state. By contrast, an object is a unit of instantiation, which has a unique identity, it has state, which can be a persistent state, and an object encapsulates its state and behaviour. A component is likely to come to life through objects and therefore would normally consist of one or more classes or immutable prototype objects.

Component and application frameworks have a close relationship. Components in a framework provide a generic architectural skeleton for a family of related applications, and complete applications could be composed by inheriting from and/or instantiating framework components. A component has well-specified functionalities with standard interface and behaviours, and a concrete implementation of an area of the system. Atkinson (2002) states that there are two types of relationship between component

instances that are important at runtime. The first is composition, which captures the idea that one component is a part of another. The key aspects of the composition relationship are:

1. Composite objects are responsible for the creation and destruction of their parts
2. The parts of a composite object take their identity from their composite object; and
3. Composition is transitive.

The other one is the client/server relationship. A client/sever relationship between two component instances defines a contract between them. For components to be independently deployable, their granularity and mutual dependencies have to be carefully controlled from the outset. Many application frameworks use Common Object Request Broker Architecture (CORBA) to increase the interoperability among each part of the framework. CORBA, a big component essentially has three parts: a set of invocation interfaces, the Object Request Broker (ORB), and a set of object adapters. For invocation interfaces and object adapters to work, two essential requirements need to be met. First, all object interfaces need to be described in a common language. Second, all languages used must have bindings to the common language (Szyperski 1997). Fayad (2000b) states that frameworks can be used to develop components. Equally, components can be used in blackbox frameworks.

The Evaluation of Application Frameworks Technology

The literature survey indicates that building application frameworks is hard and implementing application frameworks is as hard as building application frameworks (Fayad 1999, Fayad 1999b, Fayad 2000, Lewis 1995, Pree et al 2000), and that building and implementing application frameworks still need more methodological support (Fayad 1999a, 1999b, 2000). According to a survey (Fayad, 2000) the minimum time spent in developing an application framework was 0.5 person month and the maximum time to develop an application framework was 1000 person months. The average time to develop an application framework was about 21 person months. An application framework conventionally consists of the core classes of an application, and one has to understand the basic architecture of a particular application type to be able to specialise the framework (Pree et al, 2000). Using an application framework may simplify application developers' life since a framework provides generic solutions for a particular application domain. However, average learning time is a big factor in establishing the cost of the final application. The application developers have to understand what solutions the framework provides, and to comply with the rules imposed by the framework. Current literatures also indicate that application frameworks lack standards. For example, there is a suggestion that reusable components and frameworks must be accumulated in a standardised format (Chen 1999). Most researchers agree that the classification structure of an application framework must be appropriate and manageable. Application developers will have difficulties with understanding the framework if the structure of the framework is not clear and standardised.

Based on the literature analysis, this study proposes quality criteria to evaluate the quality aspects of application frameworks. The main purpose of proposing the quality criteria is to evaluate application frameworks more systematically. The quality criteria consist of four elements including correctness, extendability, reusability and interoperability drawn from various studies concerning software development and evaluation (i.e., Meyer 1988, Graham 2001 and Paul 2002). Correctness denotes that output is true and meets the specification correctly within the application domain. Correctness is one of the most important quality characteristics of software systems. In the context of software systems, correctness implies that the applications should reach certain requirements defined by users. Extendability means that applications should be easy to evolve and extend as requirements alter. Extendability is essential to ensure timely modification and enhancement of services and features (Schmidt, 1996). Technology evolution is even faster than before and the systems developed today must meet the challenge of tomorrow. It is vital that the systems developed today can be extended when user requirements change. Reusability denotes that applications should be built into reusable modules. Reusability is essential to leverage the domain knowledge of expert developers to avoid re-developing

and revalidating common solutions to recurring requirements and software challenges. It is one of the proven ways to increase product quality as the reusable modules of the software can be tested before release. Interoperability: means that applications should be readily compatible with other systems. Internet, distributed systems and networks development have made the information systems more complex than before. It is often required for a system to communicate with other systems or integrate with legacy systems.

The Results of the Evaluation

Correctness

This is the single most important quality aspect for any software systems from a systems testing point of view. McConnell (1993) finds that industry average experiences are about 15 to 50 errors per 1000 lines of codes, the application division at Microsoft experiences about 10 to 20 defects per 1000 lines of new developed codes during in-house testing and 0.5 defects per 1000 lines of codes in released product. So the reduction of the lines of code written by application developers may be one way to reduce the potential errors of applications. From the perspective of implementing application frameworks the applications developers who use application frameworks will reduce the new lines of codes required because the application framework itself is a semi-completed application. If the framework is well tested then the correctness of the application building upon the framework will increase because the application developers write less code. In other words, the application developers could reuse the codes and the structure of the framework. It is possible to reduce the potential errors caused by application developers if the lines of codes requested for an application is reduced. However, the initial cost for development of the application frameworks would be high (Fayad, 2000) because of the complex nature of developing and implementing frameworks. Fayad (2000d) also argues that a framework can produce higher quality because of the demands of a wide customer base and the fact that commercial frameworks will have successfully completed lengthy beta software programs. Frameworks technology uses class libraries, design patterns and components, which are well tested. Thus, the use of the technology will potentially increase the correctness of applications, which was built upon class libraries, patterns and components.

Extendability

Objects technology promotes extendability by utilising the concepts of abstract, inheritance, encapsulation and polymorphism. Application frameworks supports extendability by providing hot spots that allow applications to extend their stable interfaces (Fayad, 1997). However, with excessive data coupling (i.e., high inheritance coupling in whitebox frameworks approach breaks the modularity principle) the framework loses its flexibility and it is difficult to combine with other frameworks. Furthermore, it has been noticed that updating components is difficult and problematic (Fayad, 2000) for applications developed by implementing frameworks. It is not easy to achieve low coupling in practice although an idealised framework component should have clean interfaces, be cohesive and have little data coupling.

Reusability

Application frameworks technology promotes large-scale reuse through the architecture, the module and the code. Application developers not only reuse the code but also design expertise embedded in application frameworks when they implement the frameworks. Despite the difficulties of developing and implementing, the application frameworks approach has shown great potential in terms of capturing the domain knowledge, architecture, patterns, components, and programming mechanisms in the context of systems development.

Criteria	Result	Reasons	Penalty factors
Correctness	High	- Application developers write less code. - Class libraries, design patterns and components technology	- High initial cost - Difficult to test
Extendability	Depends (could be low)	Depends on the nature of the framework: - Excessive inheritance coupling may reduce the flexibility and make difficult to combine with other frameworks. - Also, updating component is difficult and problematic for applications developed by implementing frameworks.	- High initial cost, - Increase complexity
Reusability	High	- Architecture, module and code reuse	- High initial cost - Difficult to use
Interoperability	Depends (could be low)	Depends on the nature of the framework: - The nature of reversion of control - With legacy systems	- High initial cost - Increase complexity

Figure 2: Evaluation result

Interoperability

Applications developed using frameworks may have problems to interoperate with other applications since sometimes the frameworks take control of the operation, which potentially increases the difficulty of interoperation with other systems including legacy systems (thread dispatch becomes difficult to manage the construction of applications by combining two or more frameworks because the individual frameworks assumes it has the main control of the application). Thus it is possible to have a low interoperability of an application developed by implementing frameworks. The summarised result of the evaluation is shown in **Figure 2** above.

Conclusions

The experiences accumulated by the research community indicate that application frameworks apply object-oriented concepts, aimed at large-scale reuse likely domain specific and can exist in any development stage. Applications developed by implementing application frameworks may increase quality in terms of correctness and reusability with some penalty factors. The extendability and interoperability may be reduced due to the high inheritance coupling nature of the application developed from application frameworks. The study also shows that the methodological support concerning building and implementing application frameworks is inadequate. Application frameworks technology is still immature and not yet to be another silver bullet but potential is imminent.

References

Booch, G., Rumbaugh, J. & Jacobson, I. (1999). *The unified modelling language user guide.* Addison-Wesley.

Bruel, J.M, & Bellahsene, Z.(eds. 2002) *Advanced object-oriented information Systems*, OOIS 2002 workshops Montpellier, France, September 2, 2002

Brugali, D… et al (2000). *Frameworks and pattern languages: an intriguing relationship,* ACM Computing Surveys, Volume 32, 2000.

Chen, D. J., Koong, C. S., Chen, W. C., Huang, S. K., and Van Diepen. N.W.P (1999). *Integration of reusable software components and frameworks into visual software construction approach.* Journal of Information and Science and Engineering 2000.

Coad, P. & Yourdon, E. (1990). *Object oriented analysis (2nd),* Englewood Cliffs, NJ: Yourdon Pres: Prentice-Hall.

Cockburn A. (1997). *Surviving object-oriented projects: A Manager's Guide.* Addison-Wesley.

Deitel, H.M (2003). *Visual Basic.net For Experienced Programmers Developer series.* Upper Saddle River, NJ: Pearson Education.

Due, R.T. (2002). *Mentoring object technology projects* NJ07458: Prentice Hall PTR.
Eliens, A. (2000). *Object-oriented software development, 2nd ed.* England: Pearson Education.
Fayad, M.E., & Schmit, D.C. (1997). *Object-oriented applications frameworks.*
Communication of the ACM, Oct 1997 v40 n10, 32.
Fayad, M. E., Schmidt, D. & Johnson, R.E. (1999a). *Building application frameworks: object-oriented foundation of framework design.* New York: John Wiley & Sons.
Fayad, M. E., Schmidt, D. & Johnson, R.E. (1999b). *Implementing application frameworks: object-oriented framework at work*. New York: John Wiley & Sons.
Fayad, M. E., & Johnson, R.E. (2000). *Domain specific application frameworks: frameworks experience by industry.* New York: John Wiley & Sons.
Fayad, M. (2000b). *Introduction to the computing surveys' electronic symposium on object oriented application frameworks.* ACM Computing Surveys, Volume 32, No.1, March 2000.
Fayad, M. (2000d). *Enterprise Frameworks: Guidelines for selection.* ACM Computing Surveys, Volume 32, 2000.
Fowler, M. (1997). *UML Distilled, 2nd ed,* Harlow, England: Addison-Wesley.
Graham, I. (2001). *Object-oriented methods principles & practice (3rd).* London: Addison-Wesley.
Johnson, R (1997). *Frameworks for object-oriented software development*. Communication of the ACM, Oct 1997 v40 n10, 39
Lewis T., Rosenstein, L., Pree, W., Weinand, A., Gamma, E., Calder, P., Andert G., Vlissides J.& Schmucker, K. (1995). *Object-oriented application frameworks.* Greenwich: CT Manning.
McConnell, S (1993). *Code complete : a practical handbook of software construction* Redmond, Washington: Microsoft Press.
Meyer, B. (1988). *Object-oriented software construction. Englewood Cliffs* NJ: Prentice Hall.
Paul, C et al (2002). *Evaluating software architectures methods and case studies.* Addison-Wesley.
Pree, W. and Koskimies, K (2000). *Framelets- small and loosely coupled frameworks.* ACM Volume 32, Number 1es, March 2000.
Schmidt, D. C. (1996). *Lessons learned building reusable OO telecommunication software frameworks.* Lucent Labs Multiuse Express magazine, Vol. 4, No. 6, December, 1996.
Schmid, H, A. (1995). *Creating the architecture of a manufacturing framework by design patterns* OOPSLA 95 Austin.
Szyperski C. (1997). *Component software: Beyond object-oriented programming.* Addison-Wesley Longman.
Zamir, S. (1999). *Handbook of object technology (ed.)*. CRC Press LLC.