DOKUZ EYLÜL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

OBJECT ORIENTED APPLICATION FRAMEWORKS COMPARE AND SELECT THE APPROPRIATE DESIGN TECHNIQUE

by

Güler SEZER

January, 2006

İZMİR

OBJECT ORIENTED APPLICATION FRAMEWORKS COMPARE AND SELECT THE APPROPRIATE DESIGN TECHNIQUE

A Thesis Submitted to the

Graduate School of Natural and Applied Sciences of Dokuz Eylül University

In Partial Fulfillment of the Requirements for the Degree of Master of Science

in Computer Engineering, Computer Engineering Program

by

Güler SEZER

January, 2006

İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **“OBJECT ORIENTED APPLICATION FRAMEWORKS COMPARE AND SELECT THE APPROPRIATE DESIGN TECHNIQUE”** completed by **GÜLER SEZER** under supervision of **PROFESSOR DR. ALP R. KUT** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

....

Prof. Dr. Alp R. KUT
Supervisor

....	....
Assoc.Prof. Dr. Yalçın Çebi	Asst. Prof. Dr. Zafer DİCLE
(Jury Member)	(Jury Member)

Prof.Dr. Cahit HELVACI
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Dr. Alp R. KUT for offering me to study in Object Oriented Application Framework and for his advises, support and help to complete my thesis.

I would also like to thank Research Assistant Özlem AKTAŞ, who encouraged me during the writing of the thesis.

I have special thanks to my father Mustafa SAGIT who died last year for his support, patience and making me encouraged all my life.

Güler SEZER

OBJECT ORIENTED APPLICATION FRAMEWORKS COMPARE AND SELECT THE APPROPRIATE DESIGN TECHNIQUE

ABSTRACT

Object oriented frameworks are defined in many ways. The most popular definition: “a framework is a partial design and implementation from an application in a given domain” [Bosch]. In my opinion frameworks are a set of abstract and concrate classes that together comprise a generic solution to similar problems in a specific domain. The core of the framework is made up of abstract classes.

Object-oriented frameworks have been used since the early eighties and now they are becaming increasingly popular. They provide software developers with the means to build an infrastructure for their applications. Also they decrease the time of developing application. A good framework has several properties such as ease of use, extensibility, flexibility, and completeness, which can help to make it more reusable.

The aim of this study is to examine the details of the frameworks and their design techniques. Therefore, I studied basic concepts related with frameworks, design techniques used for frameworks recently and selected an object-oriented technique, which is the most powerful technique in developing framework. Some of the frameworks have been chosen to compare because of the large number of different applications. These frameworks are ACE (Adaptive Communication Enviroment), MET++ (Multimedia Application Framework) and SMA (State Maneger Interface). In addition, more general framework .NET Framework is also selected to be examined. As a result, the most appropriate technique from inside of these techniques is suggested for developing object oriented application frameworks. Also selected frameworks are compared.

Keywords: Object Oriented Frameworks, blackbox framework, whitebox framework, design guidelines, design patterns, software reuse, and domain analysis

NESNE TABANLI UYGULAMA ÇERÇEVELERİNİN KARŞILAŞTIRILMASI VE EN UYGUN TASARLAMA TEKNİĞİNİN SEÇİLMESİ

ÖZ

Nesne tabanlı uygulama çerçeveleri için çeşitli tanımlamalar yapıldı. Bunlardan en populer olanı "Uygulama çerçevesi belirli bir problem alanında kısmi bir tasarım ve kodlamadır" [Bosch]. Benim düşünceme gore uygulama çerçevesi soyut ve somut sınıflardan oluşan bir yapıdır öyle ki bu sınıflar belirli bir problem alanı içersinde karşılaşılan benzer problemler için çözümler oluştururlar. Uygulama çerçevesinin ana bölümü soyut sınıflardan meydana gelir.

Nesne Tabanlı Çerçeveler on sekizinci yüzyılın başlarında kullanılmaya başlandı ve gün geçtikçe daha da populer olmakta. Onlar yazılım geliştiriciler için uygulamalara alt yapı oluşturmaktadır. İyi bir uygulama çerçevesi belirli özelliklere sahip olmalıdır. Bunlar kullanım kolaylığı, genişleyebilirlik, esneklik, tamamlanabilirlik. Bu özellikler yeniden kullanabilirliği arttırır.

Bu çalışmanın amacı nesne tabanlı çerçeveleri ve tasarlama tekniklerinin detaylı bir şekilde incelenmesidir. Bu yüzden uygulama çerçevesi ve bu çerçeveler için kullanılan tasarlama teknikleri ile ilişkili kavramlar üzerinde çalışıldı ve uygulama çerçeveleri geliştirmede kullanılan en güzlü teknik seçildi. Şu anda kullanımda olan çok fazla uygulama çerçevesi olmasından dolayı incelemek için bazı çerçeveler seçildi. Bunlar ACE (Adaptive Communication Enviroment), MET++ (Multimedia Application Framework) ve SMA (State Maneger Interface) 'dır. Bu inceleme ve karşılaştırmadan sonra biraz daha genel olan .NET Framework incelendi. Sonuc olarak kullanılmakta olan dizayn tekniklerinden en uygun olanın seçildi. Buna ilave olarakta daha önceden seçilip incelenen nesne tabanlı uygulama çerçeveleri karşılaştırıldı.

Anahtar sözcükler: Nesne Tabanlı Uygulama Çerçeveleri ,kara kutu uygulama çerçeveleri ,beyaz kutu uygulama çerçeveleri, yazılımda yeniden kullanım, tasarlama teknikleri

CONTENTS

CHAPTER ONE
INTRODUCTION

Object oriented frameworks are a cornerstone of software developing area. A common observation when writing applications is that many parts are similar or the same for many applications. Frameworks try to make these common parts explicit and reusable. In other words the most important idea of frameworks is that a *small number* of components are used over and over. These components become very rebust over the time because they have been used in many different applications. Furthermore, they are very cheap if one compares the development effort with the number of usages. Thus, the development effort can be higher for reusable components without noticeable cost increase for the applications using the components. So, the goal of reuse is to minimise the implement the same code. I think   the next goal of framework is developing application writing any code.

A framework is not only a reuseble code and it also reusable design which can be obtained by abstract classes and interfaces. What we mean about reuseable design? Framework forces their users to follow some desing rules by abstract classes. By definition, a framework is an object-oriented design. It doesn't have to be implemented in an object-oriented language, though it usually is.

Often it is difficult to reuse a software component outside of its original area. Object-oriented frameworks can provide that area in which the component is meant to be reused and thus allow for a significant amount of reuse. The concrete classes provide the reusable components, while the design provides the context in which they are used. A framework is more than a collection of reusable components. It provides a generic solution to a set of similar problems within an application domain. The framework itself is incomplete and provides places called hooks at which users can add their own components specific to a particular application by using different techniques.

Developing a framework differs from developing an application in many ways.

The framework already supplies the architecture of the application, and users fill in the parts left incomplete by the framework. A framework typically includes the main control loop and calls application extensions to perform specific tasks. Unlike the reuse of pure function libraries, framework users give up control of the design. In return, users are able to develop applications much more quickly, and a single framework can form the basis for a whole family of related applications.

Greg Butler gives a definition for frameworks: “A framework is reusable, semi-complete application that can be specialized to produce custom applications. “ Furthermore he composed a principle “Don’t call us, we’ll call you. “. Principle is as named Hollywood principle. The framework calls the custom code, unlike a library, where the custom code calls library code as shown in Figure 1.1.

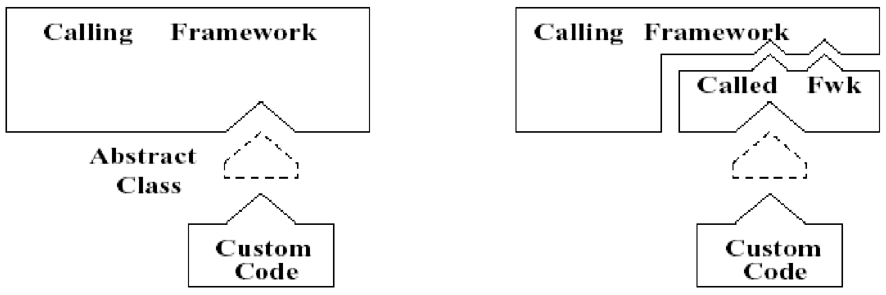


Figure 1.1 Calling Frameworks and Called Frameworks and Custom Codes

A good framework can reduce the cost of developing an application by an order of magnitude because it lets you reuse both design and code.

Unfortuanately, developing a good framework is expensive. A framework must be simple enough to be learned; yet must provide enough features that it can be used quickly and hooks for the features that are likely to change.

CHAPTER TWO

OBJECT ORIENTED FRAMEWORKS CONCEPTS

An application developed from a framework consists of several different parts as shown graphfically in the Figure 2.1. Applications are developed from frameworks by filling in missing pieces and customizing the framework in the appropriate areas.

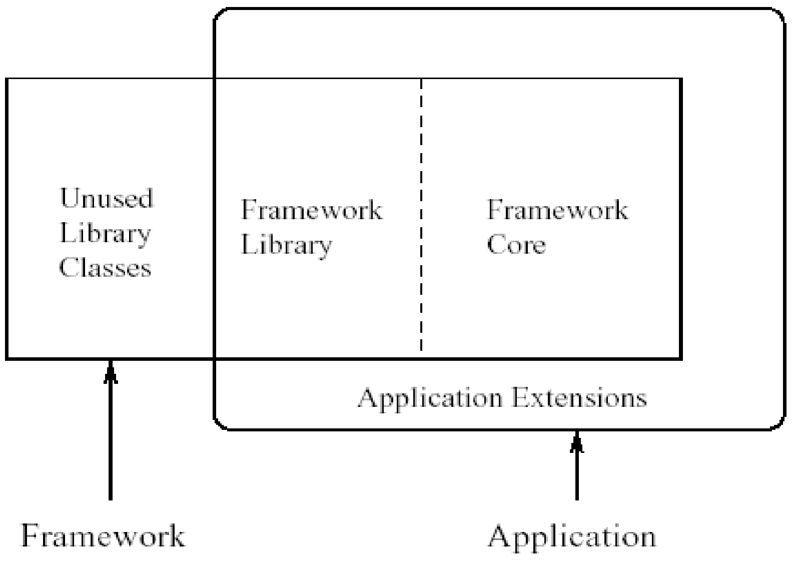


Figure 2.1 Application Developed from a Framework and the parts of Framework

The parts of a framework are:

- Framework Core: The core of the framework, generally consisting of abstract classes, that define the generic structure and behavior of the framework, and forms the basis for the application developed from the framework. However, the framework core can also contain concrete classes that are meant to be used as is in all applications built from the framework.
- Framework Library: Extensions to the framework core consisting of concrete components that can be used with little or no modification by applications

developed from the framework.

- Application Extensions: Application specific extensions made to the framework, also called an ensemble [Cotter and Potel, 1995].
- Application: In terms of the framework, the application consists of the framework core, the used framework library extensions, and any application specific extensions needed.
- Unused Library classes: Typically, not all of the classes within a framework will be needed in an application that can be developed from the framework.

2.1 Main Framework Concepts

In the previous chapter we defined frameworks in many ways. In this chapter we will study on frameworks concepts.

2.1.1 Abstract and Concrate Class

A good framework often includes abstract classes and also interfaces which embody the basic architecture and interactions of the framework for design and implementation reuse. There is no necessary all classes to be abstact in the core of framework but the classes that are hot spots must be abstract. We are going to explain hotspots later. Framework designers derive new classes from abstract classes by filling in the methods deliberately left unimplemented in the abstract classes or by adding functionality. The abstract classes should be flexible and extensible. These classes can define the properties of key, and also they capture the interactions between elements of the framework as well.

A framework will generally have a small number of these core classes, but will also have a number of concrete classes, which form the framework library. These concrete classes inherit from the abstract classes but provide specific and complete functionality that may be reused directly without modication in an application developed from the framework [Gangopadhyay and Mitra, 1995].

2.1.2 Hot Spots and Frozen Spots

A hot spot is point of variability in the framework between applications. Hot spots provide the flexibility and extensibility of the framework and their design is critical to the success of the framework. Two questions to consider about a hot spot are: [Pree, 1995]

- What is the desired degree of flexibility, remembering that flexibility has to be balanced with ease of use?
- Must the behavior be changeable at run-time, in which case composition is preferred over inheritance?

Each hot spot will likely have several hooks associated with it. The hooks describe how specific changes can be made to the framework in order to fulfill some requirement of the application [Froehlich, 1997].

Variations points open to users for implementation in a framework are called *hot spots* while stable parts are called *frozen spots*. Hot spots are implemented as *hook classes* and frozen spots as *template classes*. A template class contains *template methods* that use services of a hook class. The hook class is abstract, so its *hook methods* must be implemented when the framework is extended. Also hook methods often appers in the form of an abstact method inside an abstract class.

2.1.2.1 Template Method Pattern

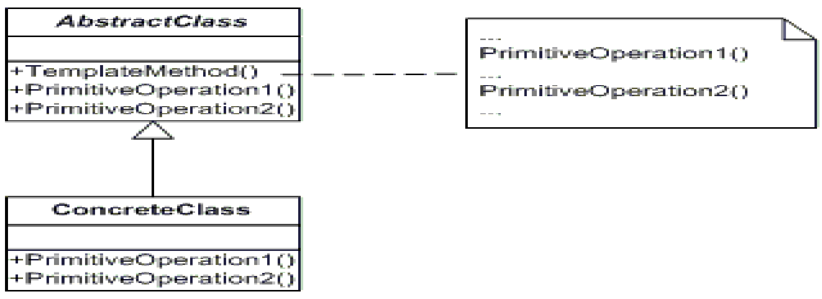


Figure 2.2 Template Method Pattern – Structural Example

```
using System;

namespace DoFactory.GangOfFour.Template.Structural
{
  class MainApp   // MainApp test application
  {
    static void Main()
    {
      AbstractClass c;
      c = new ConcreteClassA();
      c.TemplateMethod();
      c = new ConcreteClassB();
      c.TemplateMethod();
      Console.Read();     // Wait for user
    }
  }

  abstract class AbstractClass   // "AbstractClass"
  {
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

    public void TemplateMethod()    // The "Template method"
    {
      PrimitiveOperation1();
      PrimitiveOperation2();
      Console.WriteLine("");
    }
  }


 class ConcreteClassA : AbstractClass   // "ConcreteClass"
 {
    public override void PrimitiveOperation1()
    {
      Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");
    }
    public override void PrimitiveOperation2()
    {
      Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");
    }
 }

 class ConcreteClassB : AbstractClass
 {
    public override void PrimitiveOperation1()
    {
      Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");
    }
    public override void PrimitiveOperation2()
    {
      Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");
    }
  }
}
```

Schmid (1997) suggests that the variability required from a hot spot can be classified by the following characteristics:

- The common responsibility that generalizes the different alternatives.
- The different alternatives that realize responsibility.
- 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 realizes the binding while in the second case it is the end user responsibility to do it either once or repeatedly.

2.1.2.2 Template and Hook Methods

"Template Method" is one of the design patterns described in the GoF book. Its intent is "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure". The skeleton is called the template method while the deferred steps are called hook methods.

The next figure shows an example of an application derivation from a framework. The left side of the figure shows the structure of the application. The framework defines interfaces with templates and hooks that are used to implement the application specific functionality of the product as shown in the center of the figure.

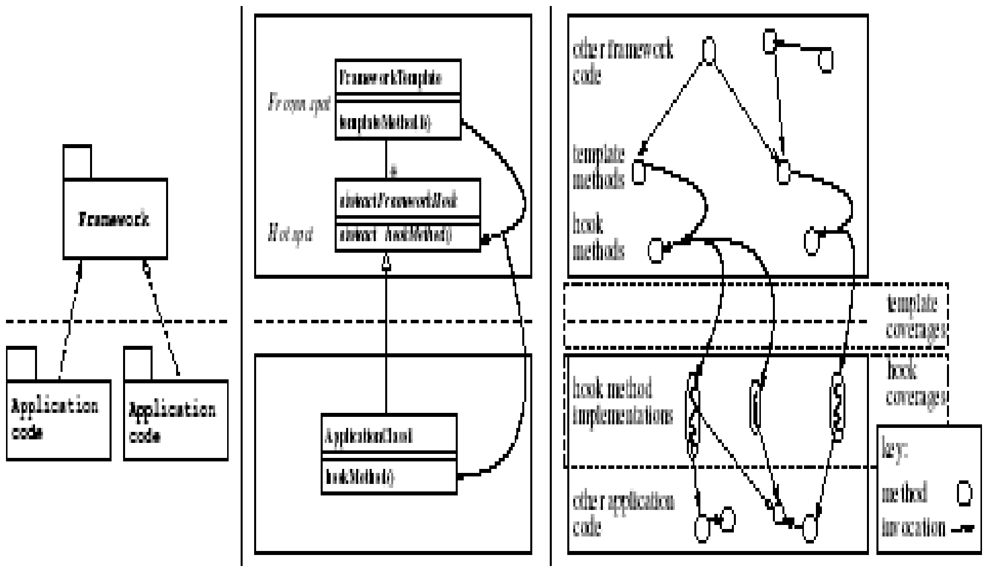


Figure 2.3 An example of the structure of an application derived from a product family based on an object-oriented framework (on the left), the class level interface between the framework and the application specific code element extending it (in the center) and context of the hook and template coverages (on the right).

2.1. 3 Compasition and Inheritence

Inheritance and composition are the two main ways for extending object oriented framework. The inheritance is a simple way to enabling hot spots inside a framework and extending framework. Althought composition is often recommended over inheritance (data driven as opposed to architecture driven) each of these ways has strengths and weaknesses. The type of customization used in each case depends upon the requirements of the framework.

2.1.3.1 Composition

Composition is generally used when interfaces and uses of the framework are fairly well defined, whereas inheritance provides flexibility in cases where the full range of functionality cannot be anticipated.

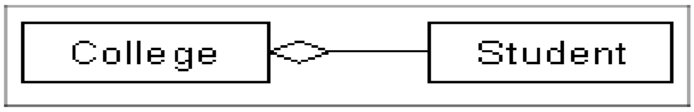


Figure 2.4 Aggregation Relationship Example

When a class is formed as a collection of other classes, it is called an aggregation relationship between these classes. It is also called a "**has a**" relationship.

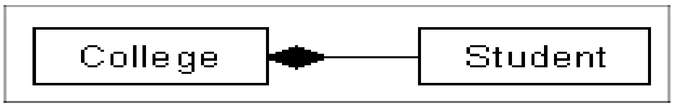


Figure 2.4 Composition Relationship Example

Composition is a variation of the aggregation relationship. Composition connotes that a strong life cycle is associated between the classes.

2.1.3.2 Inheritences

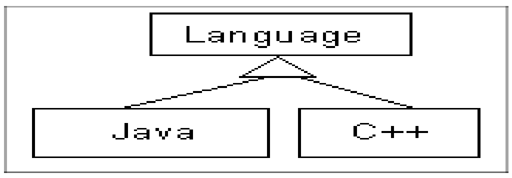


Figure 2.5 Inheritence Relationship Example

Also called an "**is a**" relationship, because the child class **is a** type of the parent class. Generalization is the basic type of relationship used to define reusable elements in the class diagram. Literally, the child classes "inherit" the common functionality defined in the parent class.

The three types of inheritence are as follows:

- Attribute inheritence:
- Implementation inheritence
- Interface inheritance

2.2 A Classification of Frameworks

Several types of frameworks have been identified by the framework group of the german Computer Science Society (GI - Gesellschaft für Informatik). Frameworks can be classified by the domain the framework can be in used and by the way the application-specific behaviour has to be specified. Classified by domain:

- Generic application frameworks (like MVC, ET++, MFC or CommonPoint) provides basic functionality that is common for all programs. They supply, for example, GUI functionality and data management services.
- Domain specific frameworks are more specific in that they provide functionality for a specific problem domain. There are framework s for operating systems, structured editors, databases, and many others.
- Support frameworks provide basic system-level functionality upon which other frameworks or applications can be built. A support framework might provide services for file access or basic drawing primitives.

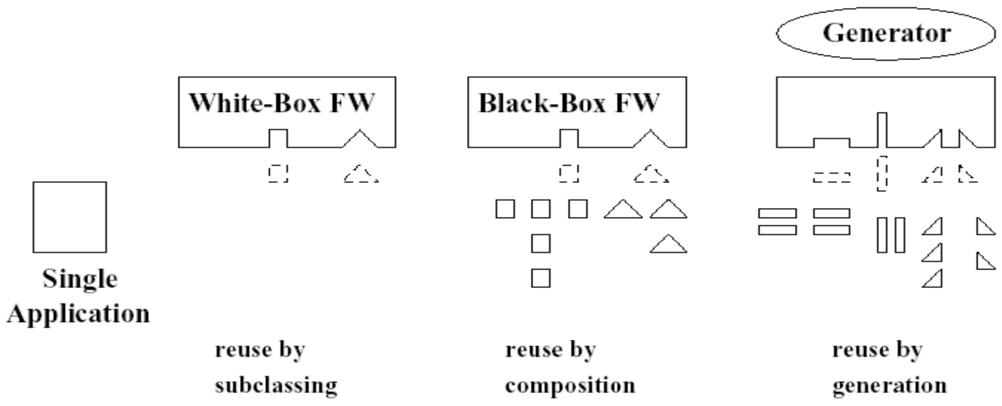


Figure 2.6 Stages of Framework Maturite and White Box, Black Box FW

By the type of specification of the application-specific functionality:

- black-box frameworks
    - interfaces
    - abstract classes

- white-box frameworks
    - components

2.2.1 *White Box Frameworks and Black Box Frameworks*

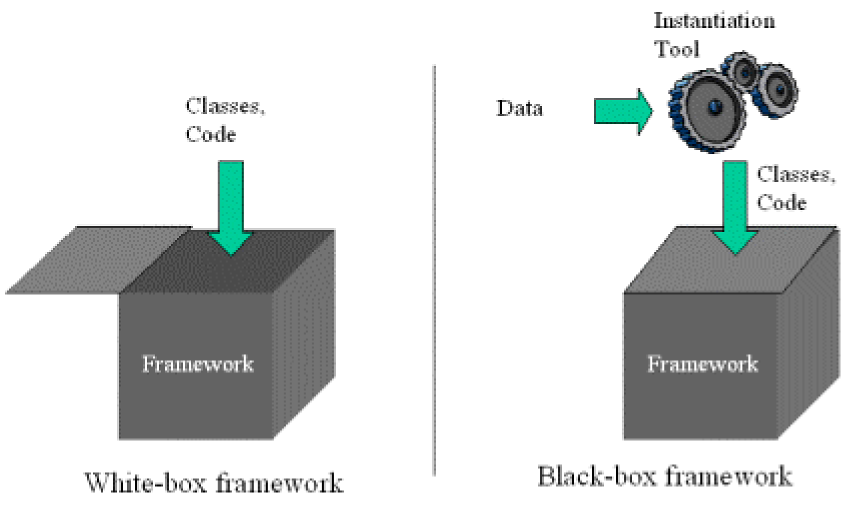


Figure 2.7 White Box Framework – Black Box Framework

2.2.1.1 White Box Frameworks

Use inheritance to build a white box framework by generalizing from the classes in the individual applications [Johnson, Foote, 1988]. Use patterns like Template Method and Factory Method to increase the amount of reusable code in the superclasses from which you are inheriting [Gamma et al., 1995].

White box frameworks, also called architecture driven frameworks rely upon inheritance for extending or customizing the framework [Adair, 1995]. Users are able to add functionality by creating a subclass of a class that already exists within the framework. White box frameworks typically require a more in-depth knowledge to use.

There are several problems with white-box frameworks:

- Every application requires the creation of many new subclasses. While the classes are mostly simple the number makes it difficult to learn the design of an application when it is to be changed.
- A white-box framework is difficult to learn because learning how to use it means learning how it is constructed.

2.2.1.2 Black Box Framework

Use component relationships to build a black box framework. A black-box framework is one where you can reuse components by plugging them together and not worrying about how they accomplish their individual tasks [Johnson, Foote, 1988]. In contrast, white-box frameworks require an understanding of how the classes work so that correct inheritances can be developed.

Black box frameworks, also called data-driven frameworks, use composition and existing components rather than inheritance for customization of the framework [Adair, 1995]. Configuring a framework by selecting components tends to be much simpler than inheriting from existing classes and so black box frameworks tend to be easier to use. Johnson argues that frameworks tend to mature towards black box frameworks.

The users have to know detail about structure of the base class. Hence, it is a white box to the client. In languages with static typing the protocol for the parameter objects is defined in an abstract class. The black box and abstract class are abstractly coupled.

The components are held and organised in a component library. Each of the components has to understand a particular protocol. The user needs to understand only the external interface of the components.

- Black-box frameworks are easier to learn because the user is not required to have knowledge about internal details of the classes he uses.
- Black-box frameworks are less flexible than white-box frameworks. The

number of possible combinations of components is determined by the architecture of the framework. There is no way of defining other behaviour than the one supplied by the framework.

- Using a black-box framework requires less programming because the components are only combined so there is no need for deriving classes and writing methods.
- Black-box frameworks allow the change of behaviour at run-time replacing a component by another component with the same protocol.

Altogether, it can be said that a framework becomes more reusable as the relationships between its parts is defined in composition, instead of using inheritance. Black-box relationships and, therefore, black-box frameworks are an ideal towards which a system should evolve even though the combinations of the objects might be limited. This problem could be compensated by providing a well-defined (white-box) interface for extending the framework

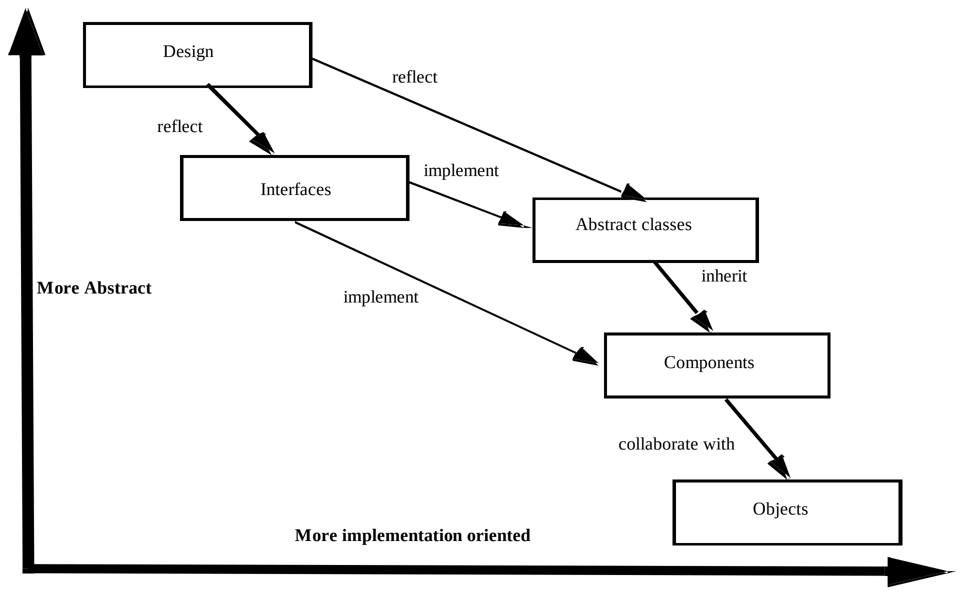


Figure 2.8 Relations between the different elements in a framework

2.3 Characteristics of Frameworks

- The *Model of the collaboration* is incorporated in and determined by the framework. Anyone who uses the framework has to stick with this model and can only change it in the way the designer of the framework designed it into the framework.
- Invariants are usually valid for single classes. In frameworks invariants are often shared by a group of objects.
- Frameworks incorporate *knowledge* of the problem domain. Developing a framework, therefore, requires deep understanding of the problem domain.
- Framework exhibits "inversion of control" at runtime via callbacks. These callbacks invoke the *hook methods* of application-defined components after the occurrence of an event, such as a mouse click or data arriving on a network connection. When an event occurs, the framework calls back to a virtual hook method in a preregistered application component, which then performs application-defined processing in response to the event. The hook methods in the components decouple the application software from the reusable framework software, which allows each to change independently as long as the interface signature and interaction protocols are not modified. Since frameworks exhibit inversion of control, they can simplify application design because the framework rather than the application runs the event loop to detect events, demultiplex events to event handlers, and dispatch hook methods on the handlers that process the events.
- Frameworks reuse design as well as code. Some aspects of a design such as the kind of objects are easily described by code. Other aspects, such as invariants prescribed by the framework are not easily expressed as code.

2.4 Goals of Frameworks

- Make it easy to develop applications.
- Write as little new code as possible.
- Enable novice programmers to write good programs.

- Leverage domain experience of expert programmers.
- Decrease the cost of developing applications.

2.5 Ways to Use A Framework

There are a number of different ways in which to use a framework. Each of them requires a different amount of knowledge about the framework and a different level of skill in using it. Taligent [1995] defines three main ways in which frameworks can be used.

- **As is:** the framework is used without modifying or adding to it in any way. The framework is treated as a black box, or maybe as a collection of optional components that are plugged together to form an application.
- **Complete:** the framework user adds to the framework by filling in parts left open by the framework developers. Completing the framework is necessary if it does not come with a full set of library components.
- **Customize:** the framework user replaces part of the framework with custom code. Modifying the framework in such a way requires a detailed knowledge of how the framework operates.

2.6 Users of Frameworks

In addition to the different ways to use a framework, different people will use a framework with different goals.

- *Regular user:* many users will use a framework in the way that it was meant to be used. They will use it as is, or they will complete the framework as the framework designer intended. A regular user needs to know only enough about the framework to enable them to use it e_ectively and typically do not require an in-depth knowledge of the framework.

- *Advanced user:* some users will want to use the framework in unexpected ways, ways that the framework developers never anticipated or planned for. They will use the framework in the same way as regular users but will also customize the framework or try to add completely new and unanticipated functionality to it. Needless to say, the advanced user needs a deeper understanding of the framework.
- *Framework developer:* a framework can evolve by adding functionality or fixing errors, specialized frameworks can be derived by adding specialized classes, or the framework can be generalized to accommodate a wider domain. The framework developers performing these activities need to know all of the details of the design and implementation of the framework and must keep in mind how changes will affect applications already developed from the framework.
- *Developer of another framework:* some users simply want to learn how the framework achieves its exibility, and need to know about the design and the decisions behind it.

Of the four types of users, the first is probably the most common. A framework is designed for a particular type of application and will be most successful when it is used to build that type of application. As an example consider a framework for building graphical user interfaces. Most users simply want to build a user interface for their application, and will use the framework as intended. A few users will push the interface paradigm to develop custom interface styles. Even fewer will be interested in evolving the framework.

2.7 Advantages and Disadvantages of Using Frameworks

Frameworks provide tremendous leverage for developers of new applications. For example, a framework represents an exible design that can be easily and quickly extended to develop applications. However, frameworks are not appropriate for every application, and here we give some advantages and disadvantages of using a framework.

2.7.1 Advantages

- Reusing expertise: the single biggest advantage of using a framework is that it captures the expertise of developers within that domain. The framework developers are generally experts within the domain and have already analyzed the domain to provide a quality, exible design. That expertise can be transferred to the application developers simply by using the framework.
- Decreased development time: the problem domain does not have to be analyzed again, and the framework often provides a number of components that can be used directly in an application. Users familiar with the framework can develop new applications from a framework in much less time than without the framework. However, there is the disadvantage of learning the framework as discussed below.
- Enhanced quality: the framework should have a well thought out, quality design. Applications developed from the framework will inherit much of the quality design, although poorly developed applications based on high quality frameworks are still possible.
- Reduced maintenance cost: if a family of similar products are developed from a single framework, then maintainers will only have to learn one standard design and will be able to maintain the whole product line more easily.

2.7.2 Disadvantages

- Framework mismatch: committing to a particular framework can be inconvenient if the requirements of the application are incompatible with the design of the framework. It can be disastrous if the incompatibilities are found late in the application development cycle. Unfortunately, knowledge of what the framework can and cannot do primarily comes from experience using the framework, although clear documentation helps to alleviate this problem. Prototype projects can help familiarize users with what a framework can be used for without jeopardizing an important project.

- Learning curve: using a framework requires some amount of learning, just as with any relatively complex tool or technique. A complex framework can require a great deal of time to learn and may not be appropriate if very few applications will be developed from it. The cost of the initial period of learning is lessened if several applications are developed from a single framework.
- Lack of design control: the framework already has a design specified and implemented and any applications developed from it have to conform to that design. Application developers give up most of their control over the design, but this loss is more than offset by the advantages of using a framework.

CHAPTER THREE

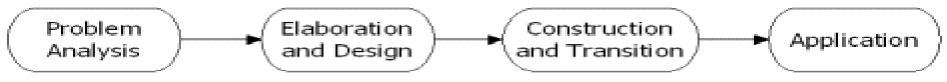
BUILDING OBJECT ORIENTED FRAMEWORKS

Frameworks should be developed from scratch. They, just like most reusable software, have to be designed to be reusable from the very beginning.

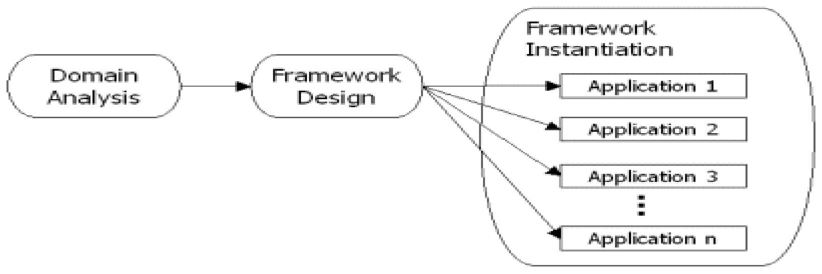
As Booch (1996) suggests object-oriented development in general and framework development in particular requires an iterative or cyclic approach in which the framework is defined, tested and refined a number of times. Additionally, small teams or even individual developers are recommended for framework development so that each member of the development team has a good overall understanding of the framework.

3.1 Design Process

Standard software development methodologies are not suficient for developing object-oriented frameworks (Pree, 1995). For example, we can't design effectively hook methods using traditional methods that tend to focus on the functional design. Hooks are also requirements of a framework, but they are quasi-functional. They do not perform functions within the system, but instead allow the framework to be customized to support a wide range of functionality. Hooks should be considered throughout the process of requirements analysis through to testing (Cline, 1996).



Traditional Object-Oriented Design



Framework Development Process

While there is no agreed upon standard for designing frameworks, some techniques have been proposed (Sparks et al., 1996) (Taligent, 1995) (Johnson, 1993) (Pree, 1995). The proposed approaches are still immature and provide guidelines rather than a fully defined methodology. Several general steps can characterize each of the approaches: analysis, design and implementation, testing, refinement.

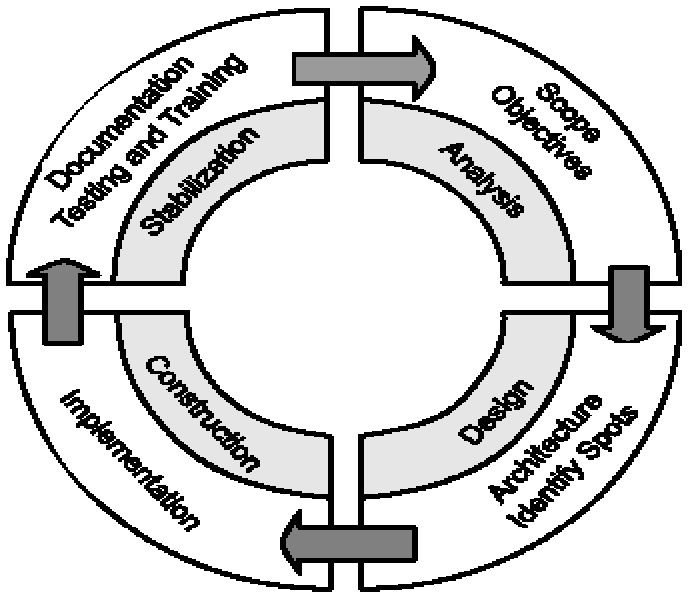


Figure 3.1 Framework Development Process

The steps are the traditional stages of software development, but each is tailored to the design of frame-works. Typically, the framework is not built during a single pass, but through multiple iterations of the steps.

3.1.1 Analysis

All of the software development, the first stage is the analysis of the problem domain. In the case of frameworks, this requires a domain expert. The expert identifies the size of the domain that the framework covers, the abstractions that will be incorporated within the framework, and how variations between applications within the domain will be dealt with.

After the domain of the framework has been determined, analyzing the domain of the framework helps to determine the primary or key abstractions that will form the core of the framework.

3.1.2 Design And Implementation

The design determines the structures for the abstractions, frozen spots and hot spots. The design and implementation of the framework are often intertwined. Abstractions can be dificult to design properly the first time and parts of a framework may have to be redesigned and reimplemented as the abstractions become better understood (Pree, 1995).

In order to develop easy to use and flexible frameworks, Taligent (1995) suggests:

- Reduce the number of classes and methods users have to override
- Simplify the interaction between the framework and the application extensions
- Isolate platform dependent code
- Do as much as possible within the framework
- Factor code so that users can override limiting assumptions
- Provide notification hooks so that users can react to important state changes within the framework

At this stage, the specific hooks for each hot spot must also be designed and specified. Hooks can be described in an informal manner or a semiformal manner using templates (Froehlich et al., 1997).

3.1.3 Testing

There are two types of testing that a framework can undergo. First, a framework should be tested in isolation; that is, without any application extensions. Testing the framework by itself helps to identify defects within the framework, and in so doing isolates framework defects from errors that might be caused by the application extensions, or in the interface between the framework and the application extensions.

Second, the true test of a framework really only occurs when it is used to develop applications. Designers never truly know if a framework can be reused successfully until it actually has been. Using the framework serves as a means of testing the hooks of the framework, the points where interactions between application extensions and the framework occur.

3.1.4 Refinement

After testing, the abstractions of the framework will often need to be extended or refined. Building a framework is a highly iterative process; so many cycles through these steps will be performed before the final framework is produced. That iterative process is necessary for framework life cycle.

3.1 Framework Development Techniques

3.2.1 Classical Buttom Up Iteration

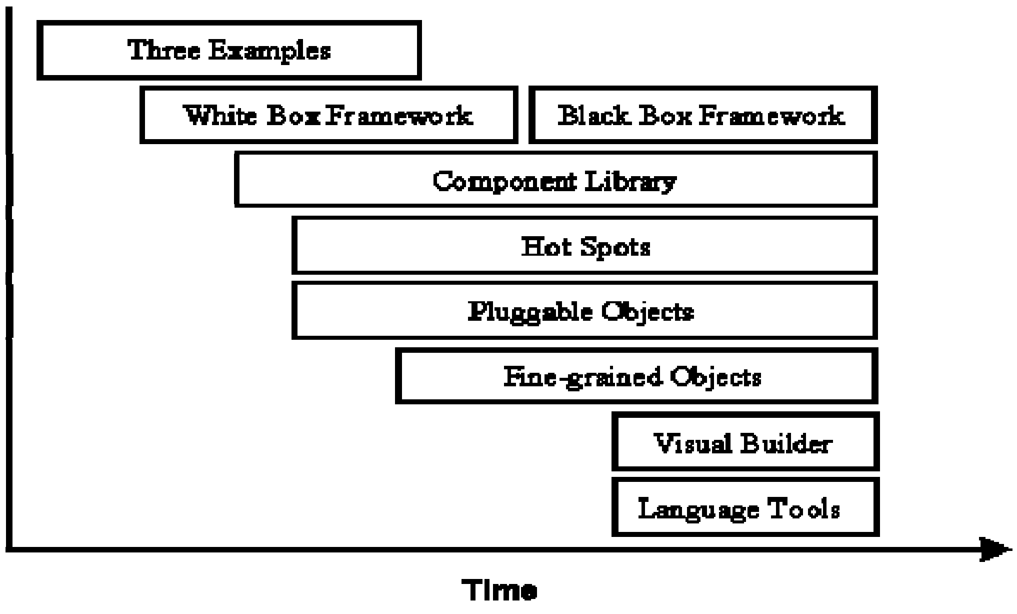


Figure 3.2 The Patterns Relation each other in a sort of time line

3.2.1.1 *Three Examples*

The framework developers use concrete examples to succeed in abstractions. Roberts and Johnson (1996) propose to build three examples in the problem domain in order to identify abstractions to be captured by the framework.

Developing reusable frameworks cannot occur by simply setting down and thinking about the problem domain. No one has the insight to come up with the proper abstractions. Domain experts won't understand how to codify the abstractions that they have in their heads, and programmers won't understand the domain well enough to derive the abstractions. In fact, often there are abstractions that do not become apparent until a framework has been reused. The more examples you look at, the more general your framework will be.

While initial designs may be acceptable for single applications, the ability to generalize for many applications can only come by actually building the applications and determining which abstractions are being reused across the applications. Generalizing from a single application rarely happens. It is much easier to generalize from two applications, but it is still difficult. The general rule is: build an application, build a second application that is slightly different from the first, and finally build a third application that is even more different than the first two. Provided that all of the applications fall within the problem domain, common abstractions will become apparent.

The framework won't be done after three applications. Developers can expect it to continue to evolve. However, it should be useful and they can use it to gather more examples. Just don't acquire too many users initially the framework *will* change!

There are two approaches to developing these applications.

- In the first approach, the applications are developed in sequence by a single team. This allows the team to begin reusing design insight immediately at the possible expense of narrowness.

- In the second approach, the applications are developed in parallel by separate teams. This approach allows for diversity and different points of view at the expense of the time it will take to unify these applications in the future.

Some people have built a series of applications many times in the same problem domain, so they might be able to design a framework without first building an example. They are not counter examples; they've already developed their three applications before they decided to start the framework.

Another way to follow this pattern is to prototype several applications without building industrial strength versions of any of them. Developer will have to refactor it when they use it, but they will be a lot closer than they would be after one application. An advantage of this approach is that developers can tell their customer that they are only buying the rights to use the framework, not complete ownership of it. Even though the application will force the developers to change the framework, they will still retain ownership of it. When they build a series of applications, it is often hard to get the right to use code written for one to build the next.

Framework developers do not need to use any design techniques when they are building these applications. Just use standard techniques, and try to make your their systems flexible and extensible.

The Runtime System Expert framework was initially developed by developing runtime systems for various platforms. The first platform was Tektronix Smalltalk. The second platform was ParcPlace Smalltalk. (Durham, Johnson, 1996) Bill Reynen created a C front-end for the RTL system that required a C runtime system (which was quite trivial)

3.2.1.2 White Box Frameworks

The framework is extended either by inheriting from framework base classes or by overriding pre-defined hooks methods (Fayad and Schmidt, 1997). While developing the subsequent applications, whenever developers realize that they need a class that is nearly the same to a class that they developed in a prior application. At that point developers can create a subclass and override the methods that are different. This is known as programming-by-difference (Johnson, Foote, 1988). After making a couple of subclasses, developers will recognize which parts consistently overriding and which parts are relatively stable. Then, developers will be able to create an abstract class to contain the common portions.

Also, we will encounter that certain methods are almost the same in all of the subclasses. Again, we should factor out the parts that change into a new method. By doing this, the original methods will all become identical and can be moved into the abstract class.

The properties of white box frameworks:

- White Box Frameworks easy to understand.
- Developers need to know the structure of base class.
- It is static and cannot change at runtime.
- 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, because even a simple extension may affect many classes that have to be extended or changed.

The Model-View-Controller framework for graphical user interfaces was originally a white-box framework at the begining. New view and controller classes were created by making subclasses of the View and Controller classes, respectively. For instance, to create a scrolling view, a programmer would have to create a new subclass of ScrollController to handle the scrolling behavior for the view.

3.2.1.3 Component Library

Using a framework, similar objects used in applications can be stored in a library for future reuse. A framework with a good library of concrete components will be easier to use than one with a small library. Various applications develop concrete classes for tailoring a framework to a specific library. The component library of a framework is the result of accumulating such concrete classes that can be reused in future applications. While at the beginning every concrete component can be included in the library in the long run, only those that are often used remain.

Start with a simple library of the obvious objects and add additional objects as you need them. Some time later some of the objects will be problem-specific and never get reused. These will eventually be removed from the library. However, these objects will provide valuable insight into the type of code that users of the framework must write. Others will be common across most or all solutions. From these, you will be able to derive the major abstractions within the problem domain that should be represented as objects in the framework.

In the long run, a class should only be included in the component library if it used by several applications, but in the beginning, you should put all of them in. If a component gets used a lot, it should remain in the library. If it never gets reused, it gets throw out. Many components will get refactored into smaller subcomponents by later patterns and disappear that way.

3.2.1.4 Hot Spots

We have introduced that definition in the previous chapter. In most existing techniques for framework development (Pree, 1995, Pree, 1999,Schmid, 1997, Schmid, 1999, Roberts and Johnson, 1998), hot spots are identified throughout the process. They begin with a particular application model, which is used to define the first framework version, and then it is refined through several iteration cycles, including more and more hot spots. In other approaches, like Bosch's (J. Bosch and Fayad, 1999), a domain analysis model is obtained at the beginning, which makes the framework hot spots more foreseeable.

Many of the Gang of Four design patterns encapsulate various types of changes. The following table shows possible design patterns to use when different portions of the framework change from application to application: (Gamma et al., 1995).

Table 3.1 Design Patterns

What varies	Design Pattern
Algorithms	Strategy, Visitor
Actions	Command
Implementations	Bridge
Response to change	Observer
Interactions between objects	Mediator
Object being created	Factory Method, Abstract Factory, Prototype
Structure being created	Builder
Traversal Algorithm	Iterator
Object interfaces	Adapter
Object behavior	Decorator, State

3.2.1.5 Plaggable Objects

New classes, no matter how trivial, increase the complexity of the system. Complex sets of parameters make parameterized classes more difficult to understand and use. Design adaptable subclasses that can be parameterized with messages to send, indexes to access, blocks to evaluate, or whatever else distinguishes one trivial subclass from another.

3.2.1.6 Fine Grained Objects

When our objects number increase, the system will be difficult to understand. Because of that we are refactoring component library to make it more understandable and reusable. The component library must be used effectively by domain experts and non-programmer.

Anywhere in your component library that you find classes that encapsulate multiple behaviors that could possibly vary independently, create multiple classes to encapsulate each behavior. Wherever the original class was used, replace it with a composition that recreates the desired behavior. This will reduce code duplication, as well as the need to create new subclasses for each new application.

3.2.1.7 Black Box Framework

Use inheritance to organize your component library and composition to combine the components into applications. Essentially, inheritance will provide taxonomy of parts to ease browsing and composition will allow for maximum flexibility in application development. When it isn't clear which is the better technique for a given component, favor composition?

A black-box framework is one where you can reuse components by plugging them together and not worrying about how they accomplish their individual tasks (Johnson, Foote, 1988). In contrast, white-box frameworks require an understanding of how the classes work so that correct subclasses can be developed.

People like to organize things into hierarchies. These hierarchies allow us to classify things and quickly see how the various classifications are related. By using inheritance, which represents is-a relationship, to organize our component library, we can rapidly see how the myriad of components in the library is related to each other. By using composition to create applications, we both avoid programming and allow the compositions to vary at runtime.

To convert the white box to back box we have to convert inheritance relationships to component relationships. Pull out common code in unrelated (by inheritance) classes and encapsulate it in new components. Many of the previous patterns will provide the techniques for locating and creating new component classes.

3.2.1.8 Visual Builder

Now we are using Black-Box Framework and we can make an application by connecting objects. An application comprises two parts: The script that connects the objects of the framework and turns them on and the behavior of the objects. The connection script is usually similar for each application, but the specific objects are different.

3.2.1.9 Language Tools

Language tools such as compilers, interpreters, and code generators are a critical part of the framewok. Any application using that framework will include several procured tools and very likely several in-house tools. Experience shows that the only guarantee with such tools is change: the underlying language may change due to improvements or extensions. The specific changes that will be made are rarely known at the outset, but change is always necessary.

The framework we created is became a programming language. It will require language tools to help debug and understand it as we explain in the previous paragraph. We have a languge tool in our framework but it is generally inadequate for dealing with the specialized composition relationship between objects. Because our framework will be filled with little object that all look alike and some of them completely unnecessary for building an application.

So we need to develop new and robust language tools. But building good tools is an expensive task that can beetwen e viewed as overhead, although language tools are indispensable for frameworks.

3.2.2 Top Down Development

- Application Family Engineering (AFE): Design framework as layered architecture of components.

- Component System Engineering (CSE): Design flexibilty into each component.

- Application System Engineering (ASE): Development with reuse of framework. (Application Engineering)

The reason for developing system families is that it pays to develop all common aspects of highly related systems only once. These “core assets” consist of a domain model, a reference architecture and implementation components. (Jacobson 1997) describes the engineering processes needed for developing system families, namely Application Family Engineering (AFE), Component System Engineering (CSE), and Application System Engineering (ASE). While the first two focuses on developing the core assets of the domain, (also known as *domain engineering*) ASE focuses on developing the actual applications by reusing as much core assets as possible (and therefore also called *application engineering*). To support these engineering processes all artefacts that are produced, such as domain models, architectures, and components, must be stored and managed.

3.2.2.1 Domain Analysis

Domain analysis is first introduced in the 1980s. There are different descriptions for it:

- It is an activity within domain engineering and is the process by which information used in developing systems in a domain is identified, captured, and organized with the purpose of making it reusable when creating new systems (Prieto Diaz 1990).

- Another description is given by Software Engineering Institute of Carnegie Mellon University. “ Domain analysis is the process of identifying, collecting, organizing, and representing the relevant information in a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain. “

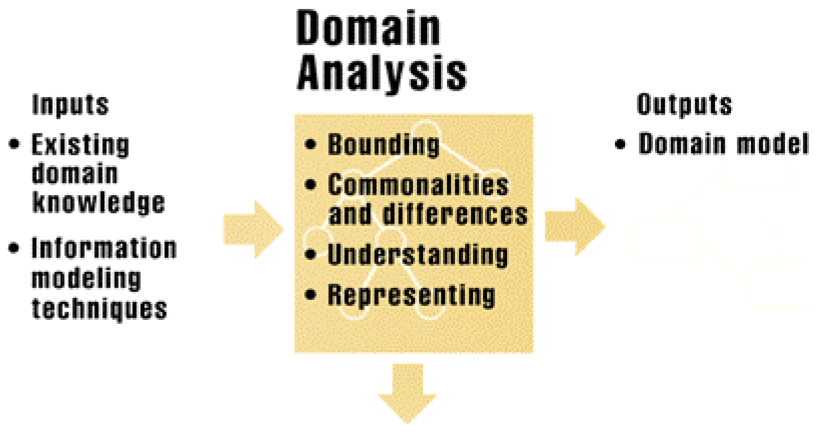

Domain Analysis
Inputs
• Existing domain knowledge
• Information modeling techniques
• Bounding
• Commonalities and differences
• Understanding
• Representing
Outputs
• Domain model

Figure 3.3 Domain analysis and the part of the process

Domain Analysis should carefully bound the domain being considered, consider commonalities and differences of the systems in the domain, organize an understanding of the relationships between the various elements in the domain, and represent this understanding in a useful way (*CARDS 1994).*

The purpose of Domain Analysis is:

- Select and define the domain,
- Build the domain model.

There are numerous Domain Analysis techniques. Each technique focuses on increasing the understanding of the domain by capturing the information in formal models. Discusses six different domain analysis approaches:

- FODA :Feature-Oriented Domain Analysis (developed at Software Engineering Institute)
- ODM :Organization Domain Modeling (M. Simos)
- Draco (J. Neighbors)
- DARE :Domain Analysis and Reuse Environment (W. Frakes & R. Prieto-Diaz)
- DSSA :Domain-Specific Software Architecture (ARPA)
- FAST :Family-Oriented Abstraction, Specification, and Translation(D. Weiss)
- ODE : Ontology-based Domain Engineering (Falbo et al.)

3.2.2.1.1 FODA – Feature Oriented Domain Analysis (Developed at Software Engineering Instite)

Feature-oriented domain analysis (FODA) is a domain analysis method based upon identifying the prominent or distinctive features of a class of systems. FODA resulted from an in-depth study of other domain analysis approaches (Kang 1990).

FODA uses to affect the maintainability, understandability, and reusability characteristics of a system or family of systems. Also it lacks a concrete description of the transition from a feature model to architecture.

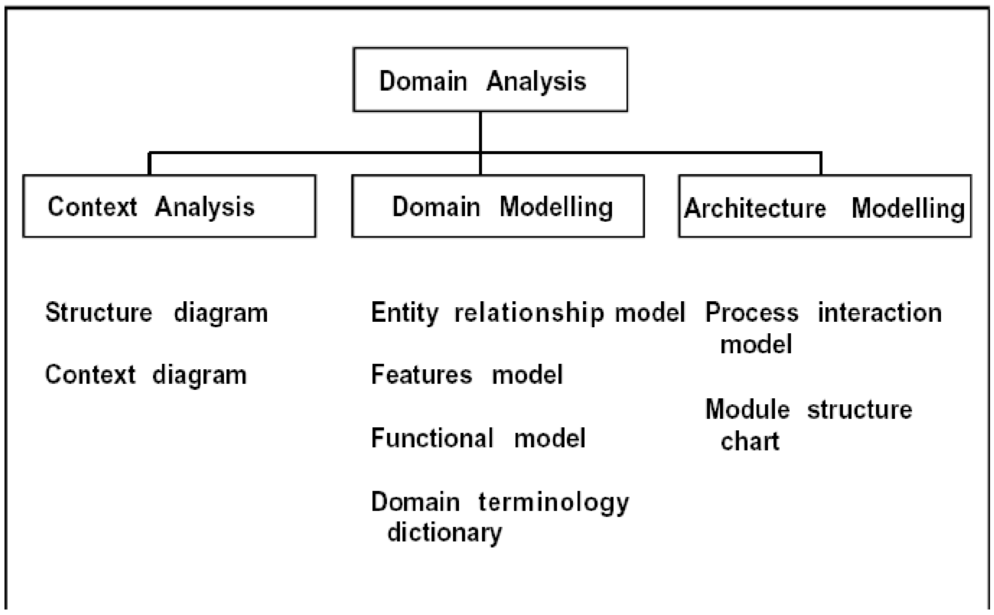


Figure 3.4 Phases and Products of Domain Analysis

The FODA process is divided into three phases:

- Context analysis: The results of this phase provide the context of the domain. This requires representing the primary inputs and outputs of software in the domain as well as identifying other software interfaces.
- Domain modelling: The products of this phase describe the problems addressed by software in the domain. They provide:
    - features of software in the domain
    - standard vocabulary of domain experts
    - documentation of the entities embodied in software
    - generic software requirements via control flow, data flow, and other specification techniques
- Architecture modelling: This phase establishes the structure of implementations of software in the domain. The representations generated provide developers with a set of architectural models for constructing applications and mappings from the domain model to the architectures. These architectures can also guide the development of libraries of reusable components.

As a result, the FODA feasibility study established methods for performing a domain analysis, described the products of the domain analysis process, and established the means to use these products for application development.

3.2.2.1.2 DSSA- Domain-Specific Software Architecture (ARPA)

Hayes-Roth defines in 1995 domain-specific software architecture (DSSA) as comprising:

- a reference architecture, which describes a general computational framework for a significant domain of applications,
- a component library, which contains reusable chunks of domain expertise, and
- An application configuration method for selecting and configuring components within the architecture to meet particular application requirements.

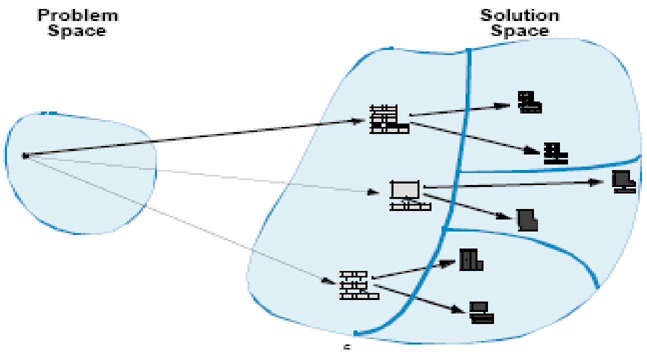


Figure 3.5 Architecture based software development

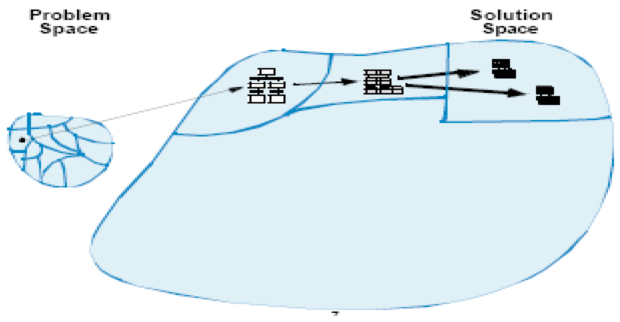


Figure 3.6 DSSA based software development

As we see in Figure 3.5 architecture based software development separate parts the solution space. But in domain specific software architecture separate parts the problem space and the solution spaces.

3.2.2.2 Design Patterns

Patterns are devices that allow programs to share knowledge about their design. In our programming, we encounter many problems that have occured, and will occur again. We can use design patterns for these types of problems.

As we told patterns are attempts to describe successful solutions to common software problems. The long-term goal is to develop handbooks for software engineers. Eventhough we have a long way for that goal, most of the patterns make general problems simple for software developers. Also we can say that they are successful for a short term. Not only do patterns teach useful techniques, they help people communicate better, and they help people reason about what they do and why. In addition, patterns are a step toward handbooks for software engineers.

A pattern is a recurring solution for a standard problem developers encountered. When related patterns are woven together they form a ``language" that provides a process for the orderly resolution of software development problems. Pattern languages are not formal languages, but rather a collection of interrelated patterns, though they do provide a vocabulary for talking about a particular problem. Both patterns and pattern languages help developers communicate architectural knowledge, help people learn a new design paradigm or architectural style, and help new developers ignore traps and pitfalls that have traditionally been learned only by costly experience.

Schmidt (1996) gives the following values:

- Success is more important than novelty. The longer a pattern has been used successfully, the more valuable it tends to be. In fact, novelty can be a liability, because new techniques are often untested. Finding a pattern is a matter of discovery and experience, not invention. A new technique can be documented as a pattern, but its value is known only after it has been tried. This is why most patterns describe several uses.

- Emphasis on writing and clarity of communication. Most pattern descriptions document recurring solutions using a standard format. We look forward to the day when we will have handbooks for software engineers. Therefore, we write our patterns in a form that is like a catalog entry. In this sense, pattern descriptions are both a literary style and technical documentation.
- The emphasis on clear writing stems from our collective experience developing complex software systems. In many cases, projects failed because developers were unable to communicate good software designs, architectures, and programming practices to each other. Well written pattern descriptions improve communication by naming and concisely articulating the structure and behavior of solutions to common software problems.
- Qualitative validation of knowledge. Another part of our ethic is to qualitatively describe concrete solutions to software problems, instead of quantifying or theorizing about them. There is a place for theoretical and quantitative work, but we feel such activities are more appropriate in a context separate from discovering and documenting patterns. Our goal is to appreciate and reward the creative process that expert developers use to build high quality software systems.
- Good patterns arise from practical experience. Every experienced developer has valuable patterns that we would like him or her to share. We value the experience of all software developers, and do not think that a few people have the patterns, and everybody else just sits back and learns them. That is why our use of writer's workshops has been so successful at pattern conferences. In a writer's workshop, participants discuss the strengths and weaknesses of each pattern, accentuate positive aspects of the patterns, share their own experience, and suggest improvements in content and style. Writer's workshops assume that we all can learn from each other.
- Recognize the importance of human dimensions in software development. The purpose of patterns is not to replace developer creativity with rote

application of rigid design rules. Neither is we trying to replace programmers with automated CASE tools. Instead, our intent is to recognize the importance of human factors in developing software. This recognition appears in design patterns when we discuss their effect on the complexity and understandability of software systems. In addition, this recognition shows itself in patterns on effective software process and organization.

3.2.2.2.1 Composite Design Pattern

When we are developing systems, we always need to implement some component, which may be either an individual object or a collection of objects. The composition pattern is a solution to represent tree structures. For example you can use the composition pattern to built job representation. A manager has workers so it is node with additional branch in three structures. Every worker is a leaves. We can give a lot of example to show using that pattern.

You can see a code part written in C# show that pattern.

```
using System;

using System.Collections;

namespace DoFactory.GangOfFour.Composite.Structural
{

  // MainApp test application

  class MainApp
  {
    static void Main()
    {
      // Create a tree structure
      Composite root = new Composite("root");
      root.Add(new Leaf("Leaf A"));
      root.Add(new Leaf("Leaf B"));

      Composite comp = new Composite("Composite X");
      comp.Add(new Leaf("Leaf XA"));
      comp.Add(new Leaf("Leaf XB"));

      root.Add(comp);
      root.Add(new Leaf("Leaf C"));

      // Add and remove a leaf
    Leaf leaf = new Leaf("Leaf D");
    root.Add(leaf);
    root.Remove(leaf);

    // Recursively display tree
    root.Display(1);

    // Wait for user
    Console.Read();
  }
}

// "Component"

abstract class Component
{
  protected string name;

  // Constructor
  public Component(string name)
  {
    this.name = name;
  }
  public abstract void Add(Component c);
  public abstract void Remove(Component c);
  public abstract void Display(int depth);
}

// "Composite"

class Composite : Component
{
  private ArrayList children = new ArrayList();

  // Constructor
  public Composite(string name) : base(name)
  {
  }

  public override void Add(Component component)
  {
    children.Add(component);
  }

  public override void Remove(Component component)
  {
    children.Remove(component);
  }

  public override void Display(int depth)
  {
    Console.WriteLine(new String('-', depth) + name);

    // Recursively display child nodes
    foreach (Component component in children)
    {
      component.Display(depth + 2);
    }
  }
 }

 // "Leaf"

 class Leaf : Component
 {
   // Constructor
   public Leaf(string name) : base(name)
   {
   }

   public override void Add(Component c)
   {
     Console.WriteLine("Cannot add to a leaf");
   }

   public override void Remove(Component c)
   {
     Console.WriteLine("Cannot remove from a leaf");
   }
   public override void Display(int depth)
   {
     Console.WriteLine(new String('-', depth) + name);
   }
}
}
```

3.2.2.2.2 *Observation Patterns*

Observer pattern has two main actors, the observer and the subject. The observer is responsible for displaying the changes to the user. The subject, on the other hand, is a business object from the problem domain. As depicted in Figure 3.7, a logical association exists between the observer and subject.

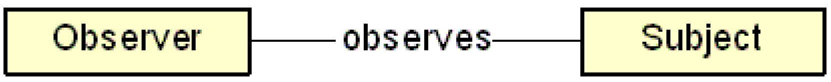


Figure 3.7 Observation pattern and the object of pattern

Most frameworks implement registration and notification via callbacks. The steps of observation pattern:

- The observer registers with the subject.
- When a change occurs, the subject notifies the observer of the change.
- The observer unregisters from the subject.

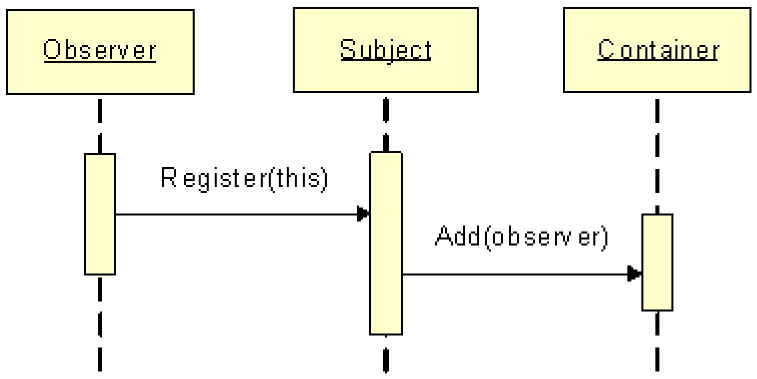


Figure 3.8 Observer Registration

Doug Purdy from Microsoft Corporation gives a useful example for observation pattern. Suppose we have a simple application that tracks stock prices throughout the day. Within this application we have a **Stock** class that models various stocks that are traded on the NASDAQ. This class contains an instance variable which represents the current ask price, which fluctuates throughout the day. In order to display this information to the user, the application uses a **StockDisplay** class that writes to stdout (standard output). Within this application, an instance of the **Stock** class acts as the subject and an instance of the **StockDisplay** class as the observer. As the ask price changes over the course of the trading day, the current ask price of the **Stock** instance changes as well (how it changes is not germane). Since the **StockDisplay** instance is *observing* the **Stock** instance, these state changes (modification of the ask price) are displayed to the user as they occur.

The use of this observation process ensures that a boundary exists between the **Stock** and **StockDisplay** classes. Suppose that the requirements for the application change tomorrow, requiring the use of a form-based user interface. Enabling this new functionality is a simple matter of constructing a new class, **StockForm**, to act as an observer. The **Stock** class would not require any modification whatsoever. In fact, it would not even be aware that such a change was made. Likewise, if a change in requirements dictated that the **Stock** class retrieved asks price information from another source (perhaps a Web service rather than from a database), the

StockDisplay class would not require modification. It simply continues to observe the **Stock**, oblivious to any changes.

```
public class MainClass  {

public static void Main(){

    //create new display and stock instances

    StockDisplay stockDisplay=new StockDisplay();

    Stock stock=new Stock();

    //create a new delegate instance and bind it
    //to the observer's askpricechanged method

    Stock.AskPriceDelegate

    aDelegate=new Stock.AskPriceDelegate(stockDisplay.AskPriceChanged);

    Stock.AskPriceDelegate

    aDelegate1=new Stock.AskPriceDelegate(stockDisplay.TellMe);


    //add the delegate to the event
    stock.AskPriceChanged+=aDelegate;

    stock.AskPriceChanged+=aDelegate1;

    //loop 100 times and modify the ask price
    for(int looper=0;looper < 100;looper++)
    {
        stock.AskPrice=looper;
    }


    //remove the delegate from the event
    stock.AskPriceChanged-=aDelegate;

    stock.AskPriceChanged-=aDelegate1;

}//Main

public class Stock
{
    public Stock()
    {

    }
        //declare a delegate for the event

        public delegate void AskPriceDelegate(object aPrice);
```

```
          //declare the event using the delegate
          public event AskPriceDelegate AskPriceChanged;

          //instance variable for ask price
          object _askPrice;

          //property for ask price
          public object AskPrice
          {

            set
            {

                //set the instance variable
                _askPrice=value;

                //fire the event
                AskPriceChanged(_askPrice);

            }

        }//AskPrice property

}//Stock class


//represents the user interface in the application
public class StockDisplay
{

    public void AskPriceChanged(object aPrice)
    {

        Console.Write("The new ask price is:" + aPrice + "\r\n"); }

        public void TellMe(object aPrice)
        {

            Console.Write("The second function test: Guler Sezer");

        }

}//StockDispslay class
```

3.3 Hot Spot Generalization

That development techniques is based on the flexible points called hot spots. The quality of a framework is directly related to the flexibility required in a domain, explicit identification of domain-specific hot spots can indeed help.

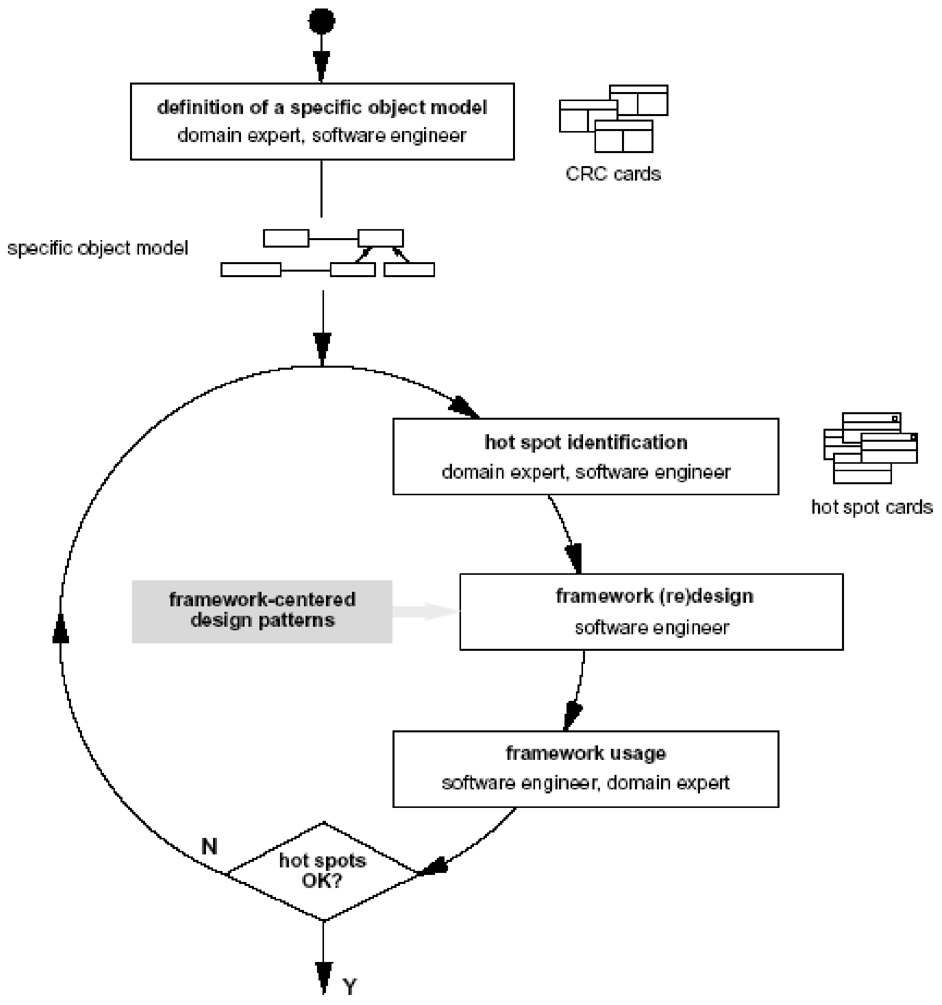

definition of a specific object model
domain expert, software engineer
CRC cards
specific object model
hot spot identification
domain expert, software engineer
hot spot cards
framework-centered design patterns
framework (re)design
software engineer
framework usage
software engineer, domain expert
N
hot spots OK?
Y

Figure 3.9 Hot-spot-driven development process (adapted from Pree, 1995).

Hot Spot Generation obtain code reuse and reduces development time, risk for errors. But besides those advantages it is very complex, abstrack and there are no adequate documentation techniques. As shown in Figure 3.9 that development technique has three steps:

- Identify hotspots: Domain-specific knowledge is required to find hot spots. Only domain analysis can help to acquire this knowledge and also domain experts. After identify hotspots we created metapatterns(hot spots cards) for documented each of hotspots in the framework. These metapatterns include information about hotspots. Then show relations between that points.

- Framework design: After domain experts have initially identified and documented the hot spots, software engineers have to modify the object model in order to gain the desired hot spot flexibility. They also use some patterns to satisfactory frameworks.
- Framework usage: A framework needs to be used several times in different applications in order to detect its weaknesses, that is, inappropriate or missing hot spots.

3.4 Use Case Assortment

The framework is based on the set theory and the notion of *pre* and *post* conditions. The method is expected to be usable within an incremental and iterative development process driven by use cases (J. Runbaugh, 1999).

Use Case Assortment is one of the first and primary means of gathering requirements in the behavioral methodology. Use cases are a standard technique for gathering requirements in many modern software development methodologies.

Use cases provide a mechanism for breaking down a given problem into smaller scenarios that reflect how the software will be used by external agents called actors (Miller). These actors might be customers or other parts of the system with which the software must interact. The use-case model describes the functional requirement.

That has mainly three steps:

- Capture functionality as use cases: A good source for identifying use cases is external events. Think about all the events from the outside world which the developer wants to react. A given event may cause a system reaction that does not involve users, or it may cause a reaction primarily from the users. Identifying the events that the developers need to react to will help them identify the use cases.

- Organize set of use cases to reflect commonality and variablty

    - Intruduce abstract use case for commonality
    - “Extends” to show variablity
    - use heuristics
- Now design and implement

CHAPTER FOUR

OBJECT ORIENTED FRAMEWORK EXAMPLES

We are studying on some of application frameworks and a foundation framework (.Net Framework). These application frameworks are ACE (Adaptative Communination Enviroment), MET++ (Multimedia Application Framework), and SMI++ (State Manager Interface).

4.1 ACE (Adaptative Communination Enviroment)

4.1.1 An Overview of ACE

ACE is a highly portable, widely used, open-source host infrastructure middleware toolkit. It is open source; freely avaiable software and we are free to use it. The core ACE library contains roughly a quarter million line of C++ code that comprises approximately 500 classes. Many of these classes cooperate to form ACE's major frameworks. The ACE toolkit also includes higher-level components, as well as a large set of examples and an extensive automated regression test suite.

To separate concerns, reduce complexity, and permit functional subsetting, ACE is designed using a layered architecture, shown in the next figure. The capabilities provided by ACE span the session, presentation, and application layers in the OSI reference model. The foundation of the ACE toolkit is its combination of an OS adaptation layer and C++ wrapper facades, which together encapsulate core OS network programming mechanisms to run portably on all the OS platforms. The higher layers of ACE build on this foundation to provide reusable frameworks, networked service components, and standards-based middleware.

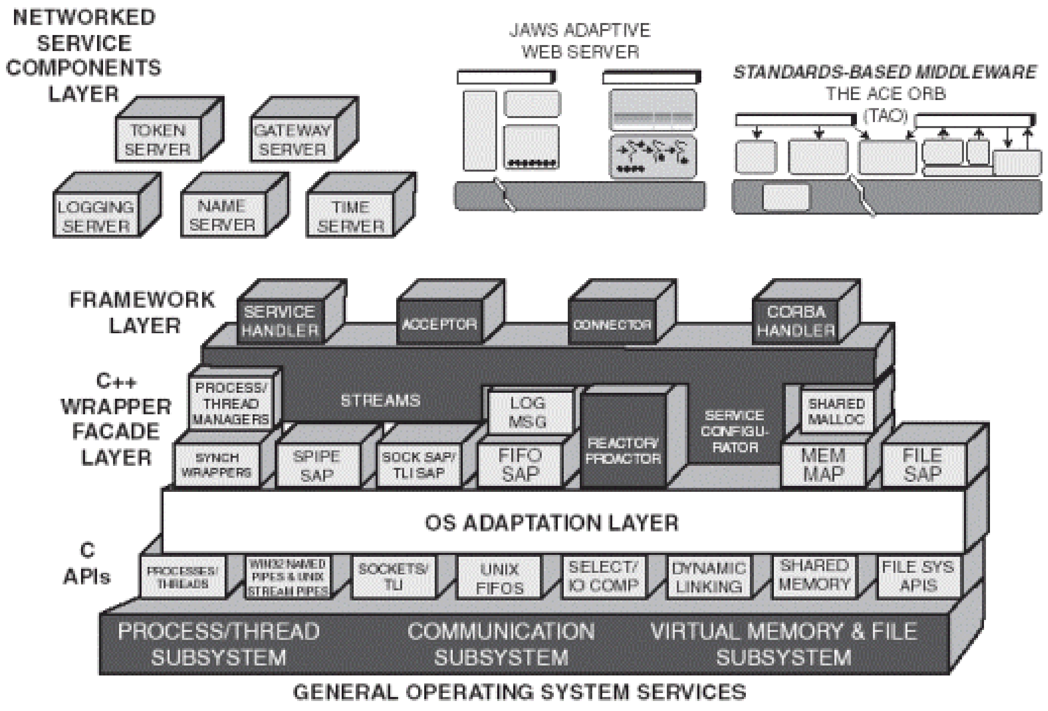


Figure 4.1 ACE Architecture, Component Layers and Framework Layer

4.2 MET++ (Multimedia Application Framework)

MET++ is an object-oriented application framework that supports the development of multimedia applications by providing reusable objects for 2D graphics, user interface components, 3D graphics, video, audio, and music (Ackerman 1996). The standard behaviour of a multimedia application such as time synchronisation and user interaction (file dialog, cut-copy-paste, multi-level undoable commands, etc.) manages by MET++. A developper will customize the MET++ application framework by composing reusable objects, by building subclasses through inheritance, and by overwriting hook methods to add his specific functionalitly (Ackerman 1996).

As Bernard Wagner explained it is a portable object-oriented C++ multimedia application framework developed at the University of Zürich. It is based on the object-oriented application framework ET++. ET++ consists of several frameworks, which support the development of desktop applications with graphical user interfaces. ET++ has a layered architecture addressing the following goals: portability among operating systems and windowing systems, generic data structures, support for graphic user interfaces, and desktop applications. The abstractions in ET++ are highly integrated and anticipate all generic interaction between application components. Thus a developer using the framework need only fill predefined slots with the application-specific content.

MET++ is built on top of ET++. It doesn’t change the architecture and style defined by ET++ but add new features. The multimedia extensions provided by MET++ are:

- 3D graphics
- Audio and music
- Video
- Time synchronization
- Visual programming.

MET++ has boon using in numerous multimedia projects, commercial applications and it is very efficient in these applications.

It has building blocks that are so-called DataUnits and DataPorts. The DataPorts provide the input/output to the DataUnits. DataUnits have several categories:

- Mathematical functions,
- GUI components,
- Wrappers,
- Data containers
- Data mappers.

MET++ uses the Adapter design Pattern for the visual programming environment to wrap existing media abstractions. Using this environment, a user can explore the behaviour and protocol of a media abstraction available in MET++ before programming against its API using C++.

The visual programming environment has been successfully employed in the areas:

- Interactive data visualization

- Animation
- sonification of animation
- Visualization of sound.

MET++ is not just a library or collection of isolated classes but a framework that pre-integrates the components and predefines their style of interaction. For example all time-dependent media can be edited regardless of their specific type in a special grouping editor provided by MET++.

4.3 SMI++ (State Manager Interface)

SMI++ is an object oriented application framework based on C++. It is developed by DELPHI and used since 1989. SMI, the State Management Interface the experiment is viewed as a collection of objects behaving as finite state machines. SMI objects can represent concrete entities, for example a hardware device or abstract entities like a logical sub-system. The objects representing concrete entities interact with the hardware they model and control through driver processes or proxies. These objects are tipically organized in hierarchical structures called domains allowing up to the full automation of the experiment by a top-level object.

The object model of the experiment is described using a dedicated language - SML - State Manager Language. This language allows detailed specification of the objects such as their states, actions and associated conditions.

4.3.1 The SMI Model

SMI is a tool for developing control systems; it is based on the concept of Finite State Machines (FSM). Finite state machines are a simple way to describe control systems, complex systems can be broken down into small and simple FSMs that are hierarchically controlled by other FSMs. Using SMI the experiment can be decomposed and described in terms of objects behaving as finite state machines.

SMI objects can represent concrete entities, for example a hardware device or abstract entities like a logical sub-system. The objects representing concrete entities interact with the hardware they model and control through driver processes or proxies. The objects are typically organised in hierarchical structures called domains.

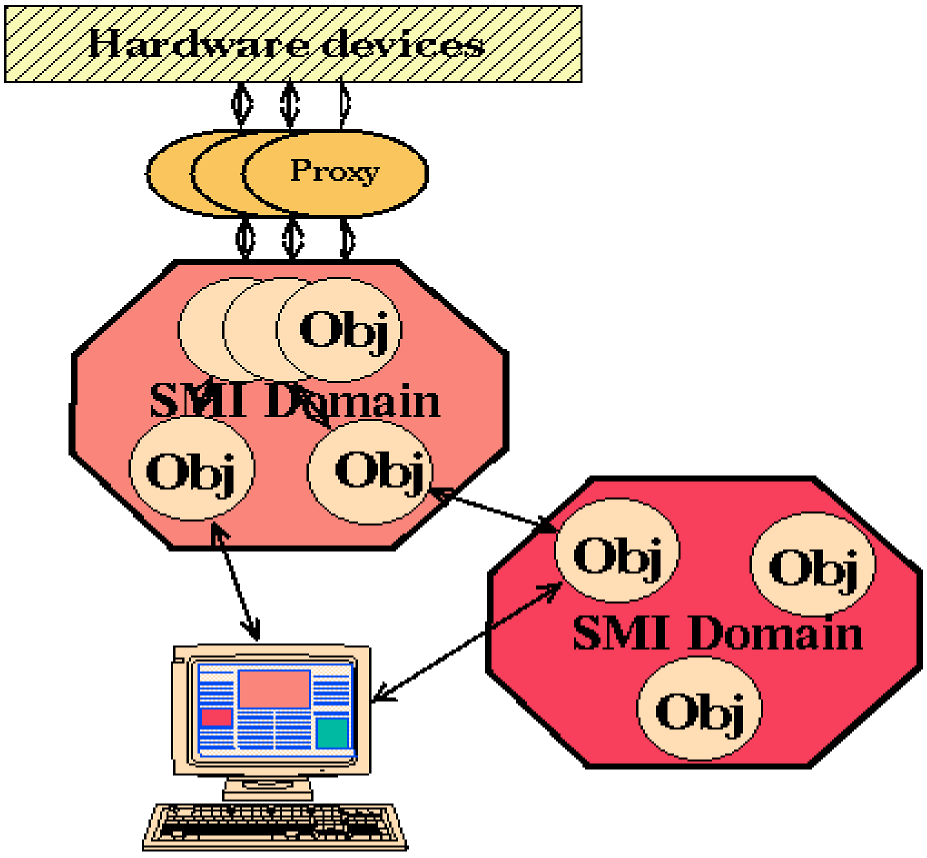


Figure 4.2 SMI Mode and SMI Domain Relationships

The SMI mechanism allows an easy reconfiguration of the system: modifying or replacing proxies and logical modifications can easily integrate changes in the hardware by changing the SMI code. The decoupling between the actual actions on the hardware (done by the Proxies) and the control logic (residing in the SMI objects) makes the evolution of a system from its first test phase up to final complexity a very smooth process

4.3.2 SML The Language

The object model of the experiment is described using State Manager Language (SML). This language allows detailed specification of the objects such as their states, actions and associated conditions. The main characteristics of this language are:

- Finite State Logic
  Objects are described as finite state machines. The only attribute of an object is its state. Commands sent to an object trigger actions that can bring about a change in its state.
- Sequencing
  an action on an abstract object is specified by a sequence of instructions, mainly consisting on commands sent to other objects and logical tests on states of other objects. Actions on concrete objects are sent off as messages to the Driver Control Processes.
- Asynchrounous
  several actions may proceed in parallel: a command sent by object-A to object-B does not suspend the instruction sequence of object-A. Only a test by object-A on the state of object-B suspends the instruction sequence of object-A if object-B is still in transition.
- AI-like rules
  each object can specify logical conditions based on states of other objects. These when satisfied will trigger an action on the local object. This provides the mechanism for an object to respond to unsolicited state changes of its environment.

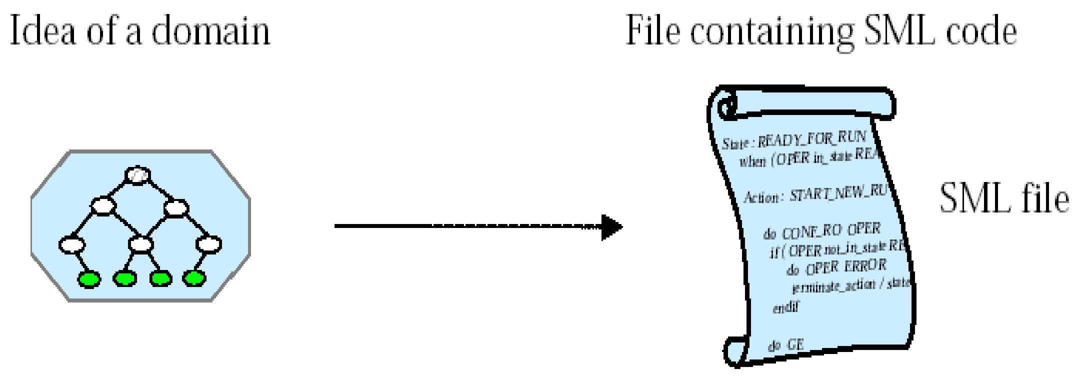


Figure 4.3 SML File and the relationship between idea of a domain and SML File

Example of SML code

```
object : RUN_CONTROL
  state : READY
    action : START_RUN
      do MOUNT TAPE
      if TAPE not in_state MOUNTED then
        do MOUNT_ERROR ERROR_OBJ
        terminate_action/state=ERROR
      endif
      do START READOUT_CONTROLER
      if READOUT_CONTROLER in_state RUNNING
        terminate_action/state=RUN_IN_PROGRESS
      ...
  state : RUN_IN_PROGRESS
    when TAPE in_state FILE_FULL
      do PAUSE_RUN
    when READOUT_CONTROLER in_state ERROR
      do ABORT_RUN
    action : ABORT_RUN
      ...

object : READOUT_CONTROLER/driver
  state : READY
    action : START
    ...
  state : RUNNING
    action : PAUSE
    action : ABORT
    ...
```

4.3.3 State Manager

Logic Engine that reads the SML file and ‘drives’ the described model:

- Responds to external commands
- Responds to asynchronous changes in the environment
- Sends out ‘properly’ sequenced commands to other domains and proxy processes

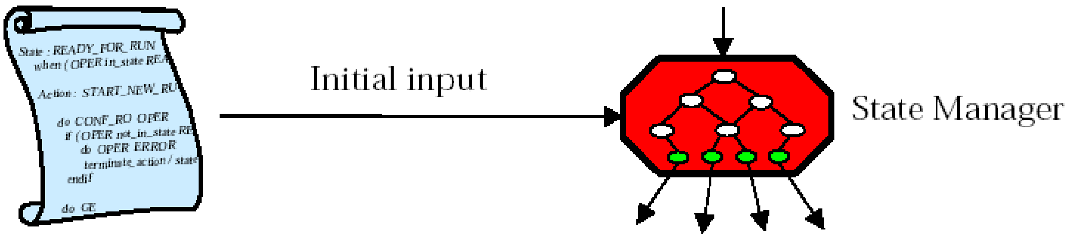


Figure 4.4 State Manager and the relationship between SML Code and State Manager

4.4 .NET Framework

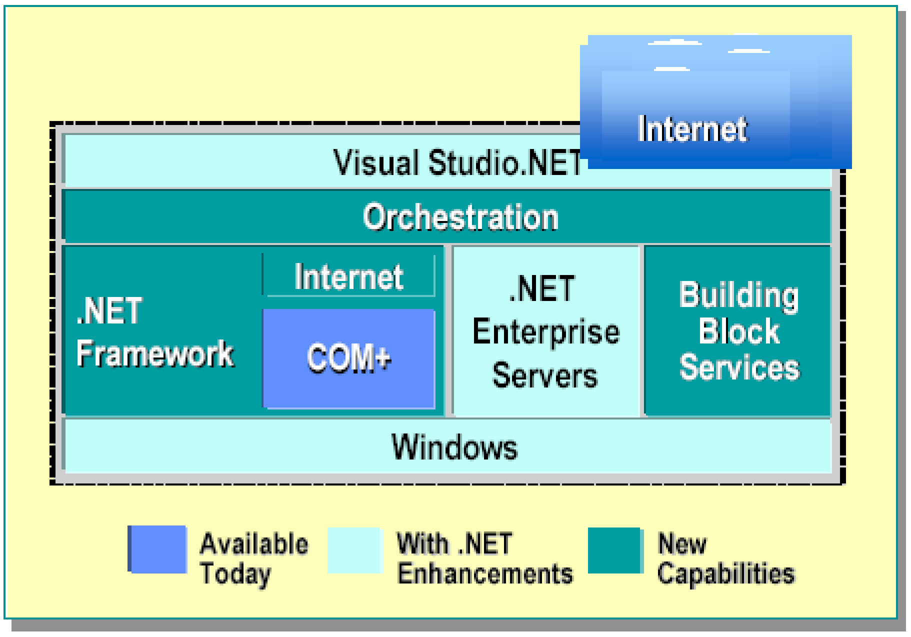


Figure 4.5 The .NET platform is made up of several core technologies

Microsoft .NET is a set of Microsoft software technologies for rapidly building and integrating XML Web services, Microsoft Windows-based applications, and Web solutions. The .NET Framework is a language-neutral platform for writing programs that can easily and securely interoperate, using a system similar to Java/Java Virtual Machine (JVM). And also it is a foundation framework which is my master subject's parts. Because of that we are going to learn deeply information about that framework and then we can compare these frameworks.

It standardizes common data types and communications protocols so that components created in different languages can easily interoperate. For example you can create a C# component and use it in Visual Basic. Also a component which is created in J# can convert in any language in .Net platform.

".NET" is also the collective name given to various software components built upon the .NET platform. These will be both products (Visual Studio.NET and Windows.NET Server, for instance) and services (like Passport, .NET My Services (a.k.a. HailStorm), and so on).

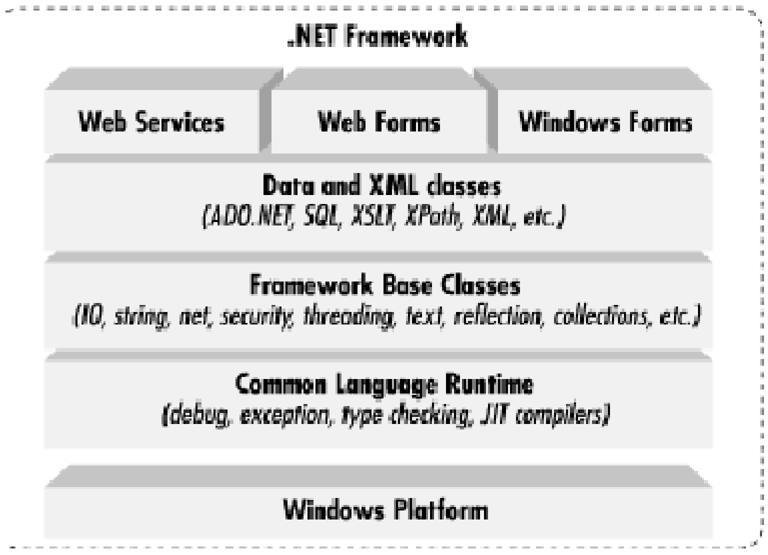

.NET Framework
Web Services
Web Forms
Windows Forms
Data and XML classes
(ADO.NET, SQL, XSLT, XPath, XML, etc.)
Framework Base Classes
(IO, string, net, security, threading, text, reflection, collections, etc.)
Common Language Runtime
(debug, exception, type checking, JIT compilers)
Windows Platform

Figure 4.6 Net Framework Achitecture

The .NET Framework sits on top of the operating system, which can be any flavor of Windows, and consists of a number of components. Currently, the .NET Framework consists of:

- Four official languages: C#, VB .NET, Managed C++, and JScript .NET
- The Common Language Runtime (CLR),
- Framework Class Library (FCL).

4.4.1 CLR (Common Language Runtime)

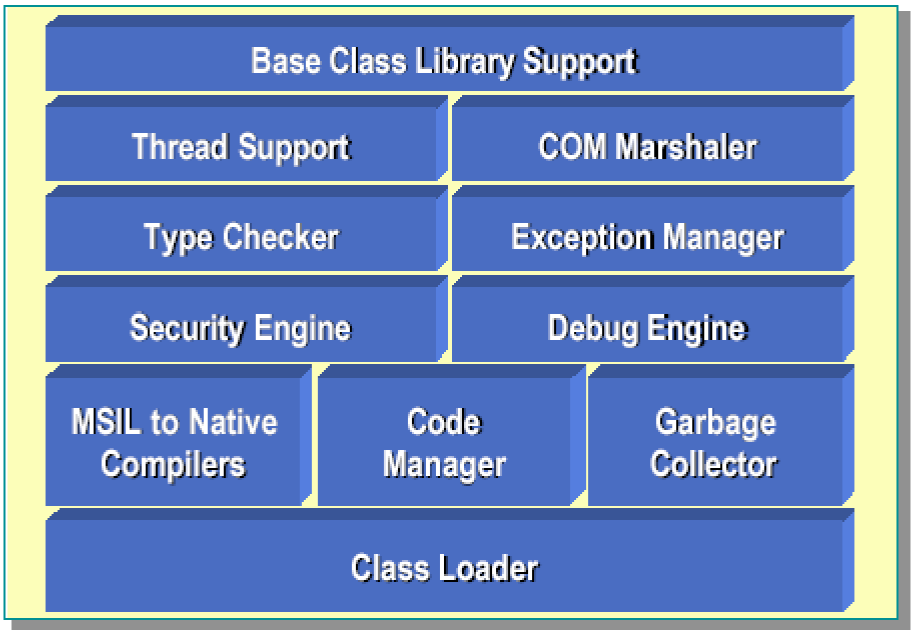


Figure 4.7 Comman Language Runtime and the architecture of CLR

The most important component of the .NET Framework is the CLR, which provides the environment in which programs are executed. The CLR includes a virtual machine, analogous in many ways to the Java virtual machine. At a high level, the CLR activates objects, performs security checks on them, lays them out in memory, executes them, and garbage-collects them. (The Common Type System is also part of the CLR.) (Liberty, J. 2001)

The CLR is described as a managed execution environment that handles memory allocation, error trapping and interacting with the operating system services. The most important features of CLR are:

- Conversion from a low-level assembler-style language, called Intermediate Language (IL), into code native to the platform being executed on.
- Memory management, notably including garbage collection.Checking and enforcing security restrictions on the running code.
- Loading and executing programs, with version control and other such features.

When the developer compiles her code on the .NET platform, the compiler doesn’t produce a traditional executable file, but rather compiles the code into Microsoft Intermediate Language (MSIL). MSIL is CPU independent and it is much higher level than most machine languages. One written and built, a managed .NET application can execute on any platform that supports the .NET CLR.

.NET programs are constructed from “Assemblies”. An Assembly is a compiled and versioned collection of code and metadata. This metadata describes the interface of the component - for instance, what methods it provides, what parameters they take, and what they return. The presence of metadata in the file along with the MSIL enables your code to describe itself, which means that there is no need for separate type libraries or Interface Definition Language (IDL). The runtime locates and extracts the metadata from the file as needed during execution.

All Assemblies contain a Manifest, which contains the Assembly name, version, and locale, has a list of files that form the Assembly, what dependencies the Assembly has, and what features are exported by the Assembly. When you want to execute the code MSIL converted the execute code by JIT compiler. CLR include one or more JIT compiler.

The following features of the .NET framework are given by Liberty in the book of Programming C#:

- Managed Code - is code that targets .NET, and which contains certain extra information - “metadata” - to describe itself. Whilst both managed and unmanaged code can run in the runtime, only managed code contains the information that allows the CLR to guarantee, for instance, safe execution and interoperability.
- Managed Data - With Managed Code comes from Managed Data. CLR provides memory allocation and deallocation facilities, and garbage collection. Some .NET languages use Managed Data by default, such as C#, Visual Basic.NET and JScript.NET, whereas others, namely C++, do not. Targeting CLR can, depending on the language you’re using, impose certain constraints on the features available; for instance, C++ loses multiple inheritances. As with managed and unmanaged code, one can have both managed and unmanaged data in .NET applications - data that doesn’t get garbage collected but instead is looked after by unmanaged code.
- Common Type System - The CLR uses something called the Common Type System (CTS) to strictly enforce type-safety. This ensures that all classes are compatible with each other, by describing types in a common way. CTS defines how types work within the runtime (their declaration and usage), which enables types in one language to interoperate with types in another language, including cross-language exception handling. As well as ensuring that types are only used in appropriate ways, the runtime also ensures that code doesn’t attempt to access memory that hasn’t been ` allocated to it (that is to say, the code is type-safe).
- Common Language Specification - The CLR provides built-in support for language interoperability. However, this support does not guarantee that the code you write can be used by developers using another programming language. To ensure that you can develop managed code that can be fully used by developers using any programming language, a set of language features and rules for using them called the Common Language Specification

(CLS) has been defined. Components that follow these rules and expose only CLS features are considered CLS-compliant.

4.4.2 The Class Library

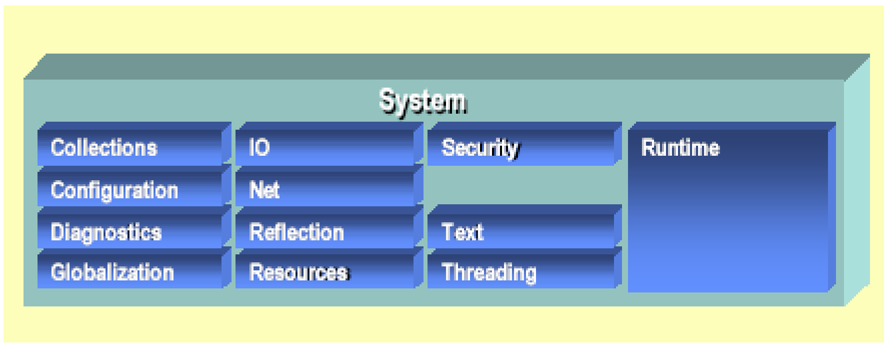


Figure 4.8 Base Class Libraries

As explained in MSDN from Microsoft documentation .NET class configuration is a single-rooted hierarchy. It's containing over 7000 types. The root of the namespace is called System; this contains basic types like Byte, Double, Boolean, and String, as well as Object. All objects derive from System.Object. As well as objects, there are value types. Value types can be allocated on the stack (which is generally quicker to some degree than allocation on the heap), which can provide useful flexibility. There are also efficient means of converting value types to object types if and when necessary.

To access any of the platform's features, we need to know which namespace contains the types that expose the facilities we're after. When we want to customize any type's behavior, we can simply derive our own type from the desired FCL type. The object-oriented nature of the platform is how the .NET Framework presents a consistent programming paradigm to software developers. Also, developers can easily create their own namespaces containing their own types. These namespaces and types merge seamlessly into the programming paradigm.

CHAPTER FIVE

CONCLUSION

In the research described in this thesis object-oriented application frameworks are investigated to get architectural description, and also the design techniques that obtain reusable implementation and design are examined.

ACE (Adaptative Communination Enviroment), MET++ (Multimedia Application Framework), and SMI++ (State Manager Interface) are domain specific frameworks. On the other hand .Net Framework is an object oriented foundation frameworks.

An examination of domain specific frameworks shows that all of them are succefull in most of the applications which they used. Although ACE doesn’t have robust library, it is open source. Because of that developer can understand framework very quickly, use it and extended it. And also developer can find many examples for using that framework. MET ++ is complex but effective framework for multimedia application. It is not open source but it is succeed in design and implementation reusage. The last framework which was investigated has own language but components are unchangeable.

In the previous chapter object oriented application development approaches are classified into four categories:

- Classical Bottom Up Iteration
- Top Down Development
- Hot Spot Generalization
- Use Case Assortment

All these approach is using for developing application frameworks. In my opinion the most efficient approach for framework development and evolution will be a hybrid approach. It combines the modeling aspects of the top-down domain engineering approaches like domain anlysis , and the iterative, refactoring approaches of the bottom-up iteration object-oriented enviroment. First domain analysis must be done and then classical bottom up iteration can be used with design patterns.

In conclusion, our main purpose is usage of design and implementation. We are developing frameworks for that aim. I think the hybrid methodology can be succesful for obtaining easy and effective object oriented frameworks.

REFERENCES

Adair, D. (1995). *Building Object-Oriented Frameworks. AIXpert. Feb.*

G. Arango, G., & R. Prieto-Diaz, R. (1991). *Domain Analysis Concepts and Research Directions, IEEE Computer Society.*

Bosch, J., Molin, P., Mattson, M., & Bengtsson, P. (1999). *Object oriented frameworks problems and experiences. Building Application. Frameworks, Fayad ME, Schmidt DC, Johnson RE (eds.). Wiley & Sons.*

Fayad, M., & Schmidt, D. (1999). *Building Application Frameworks: Object Oriented Foundations of Design. First Edition, John Wiley & Sons.*

Booch, G. (1994). *Designing an Application Framework. Dr. Dobb's Journal. 19(2):24-32.*

Butler, Greg. Object Oriented Framework. Department of Computer Science Concordia University, Montreal
gregb@cs.concordia.ca http://www.cs.concordia.ca/~faculty/gregb

Bosch, J. (1999). *Design of an object-oriented framework for measurement systems. Object Oriented Application Frameworks, Fayad ME, Schmidt DC, Johnson RE (eds.). Wiley & Sons.*

Mattsson, M., & Bosch, J. (1997) *Framework composition problems, causes and solutions. Proceedings Technology of Object-Oriented Languages and Systems, U.S.A., August 1997.*

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995) Design Patterns Elements of Reusable Object Oriented Software. Addison-Wesley.

Sparks, S., Benner, K., & Faris, C. (*1996*). *Managing object oriented framework reuse. Computer; 29(9):53–61.*

Bengtsson, P., & Bosch, J. (*1999*). *Haemo dialysis software architecture design experiences. Proceedings of the 21st International Conference on Software Engineering, May.*

Roberts, D., & Johnson, R. (1998). *Patterns for evolving frameworks. Pattern Languages of Program Design 1998; 3:471–486.*

Riehle, D., & Gross, T. (1998). *Role model based framework design and integration. Proceedings of OOPSLA '98. ACM Press; 117–133.*

Pree, W. (1994). *Design Patterns for Object-Oriented Software Development. Addison-Wesley: Reading, MA.*

Daly, J., Brooks, A., Miller, J., Roper, M., & Wood, M. *The effect of inheritance on the maintainability of object oriented software:An empirical study. Proceedings of International Conference on Software Maintenance. IEEE Computer Society Press:Los Alamitos, CA, U.S.A.; 20–29.*

Schmidt, DC., & Fayad ,ME. (1997). *Lessons learned uilding reusable OO frameworks for distributed software. Communications of the ACM; 40(10):85–87.*

Johnson, RE., & Foote, B. (1988). *Designing reusable classes. Journal of Object Oriented Programming; 1(2):22–35.*

D'Souza, D., & Wills, AC. (1999). *Composing modelling frameworks in catalysis. Building Application Frameworks Object Oriented foundations of Framework Design, ch. 19, Fayad ME, Schmidt DC, Johnson RE (eds.). John Wiley & Sons.*

Bosch, J. (1998). *Specifying frameworks and design patterns as architectural fragments. Proceedings Technology of Object-Oriented Languages and Systems; 268–277.*

Bosch, J. (2001). *Design and Use of Software Architectures—Adopting and Evolving a Product Line Approach. Addison-Wesley, 2000.Copyright Ó 2001 John Wiley & Sons, Ltd. Softw. Pract. Exper.; 31:277–300*

Weiss, *G. (2000). Multiagent systems: a modern approach to distributed artificial Intelligence. The MIT Press, Second printing.*

Ferber, J. (1999). Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Pub Co.

Garcia, A., *Silva, V., Lucena, C., & Milidiú, R. An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems.*

Garcia, A., Lucena, C. J., & Cowan, D.D. *Engineering Multi-Agent Object-Oriented Software with Aspect-Oriented Programming.*

Garcia, A., & Lucena, C. J. (2001). *An Aspect-Based Object-Oriented Model for Multi-Agent Systems. 2nd Advanced Separation of Concerns Workshop at ICSE'2001.*

Kendall, E., Krishna, P., Pathak, C., & Suresh, C. (1999) *A Framework for Agent Systems. In: Implementing Application Frameworks – Object-Oriented Frameworks at Work, M. Fayad et al. (editors), John Wiley & Sons.*

Silva, O., Garcia, A., & Lucena, C.J. (2001). *A Reflective Tuple SpaceEnvironment for Dependable Mobile Agent Systems. III WCSF at IEEEMWCN 2001, Recife, Brasil.*

Milidiu, R.L., Lucena, C.J., & Sardinha, J.A.R.P. (2001). *An object-oriented framework for creating offerings. 2001 International Conference on Internet Computing (IC'2001).*

Fontoura, M.F., Haeusler, E.H., & Lucena, C.J.P. (1998). *The Hot-Spot Relationship in OO Framework Design. MCC33/98, Computer Science Department, PUC-Rio.*

Rosana, T., Braga, V., & Paulo, Cesar Masiero *Journal of Object Technology. Finding framework hotspots in pattern languages. Instituto de Ciˆencias Matem´aticas e de Computa¸c˜ao Universidade de S˜ao Paulo Brazil*

Kyo, C. Kang., Sholom, G. Cohen., James, A. Hess., William, E. Novak., & A. Spencer, Peterson (1990). *Feature-Oriented Domain Analysis (FODA)Feasibility Study*

Franek, B. (1998) *Object Oriented Framework for Designing and Implementing Distributed Control Systems Rutherford Appleton Laboratory Great Britain*

Woolridge, Richard (1999) *An Intraduction of Use Case Analysis* *http://www.cbd-hq.com/articles/1999/991115rw_caseanalysis.asp*

Wolfgang, Pree *Hot-Spot-Driven Framework Development University of Constance D-78457 Constance, Germany*

Hansen, Todd., & Granville, Miller *Definition and Verification of Requirements Through Use Case Analysis and Early Prototyping*

APPENDICES

A. ACE (Adaptative Communination Enviroment) Class Diagram

[Containers]

- Array.cpp
- Array.h **[doxygen]**
- Array.inl
- Containers.cpp
- Containers.inl
- Containers.h
- Hash_Map_Manager.cpp
- Hash_Map_Manager.h **[doxygen]**
- Filecache.cpp
- Filecache.h **[doxygen]**
- Free_List.cpp
- Free_List.inl
- Free_List.h **[doxygen]**
- Managed_Object.cpp
- Managed_Object.h **[doxygen]**
- Managed_Object.inl
- Map_Manager.cpp
- Map_Manager.h **[doxygen]**
- Map_Manager.inl
- Object_Manager.cpp
- Object_Manager.inl
- Object_Manager.h **[doxygen]**
- SString.cpp
- SString.h **[doxygen]**
- SString.inl

[Concurrency]

- Activation_Queue.h **[doxygen]**
- Activation_Queue.cpp
- Atomic_Op.inl
- Future.h **[doxygen]**
- Future.cpp
- Method_Request.h **[doxygen]**
- Method_Request.cpp
- Process.cpp
- Process.h **[doxygen]**
- Process.inl
- Process_Manager.cpp
- Process_Manager.h **[doxygen]**
- Process_Manager.inl
- Sched_Params.cpp
- Sched_Params.h **[doxygen]**
- Sched_Params.inl

- Synch.cpp
- Synch.h
- Synch.inl
- Synch_Options.cpp
- Synch_Options.h **[doxygen]**
- Synch_Options.inl
- Synch_T.cpp
- Synch_T.h
- Synch_T.inl
- Thread.cpp
- Thread.h **[doxygen]**
- Thread.inl
- Thread_Manager.cpp
- Thread_Manager.h **[doxygen]**
- Thread_Manager.inl
- Token.cpp
- Token.h **[doxygen]**
- Token.inl

[Config]

- config.h
- Basic_Types.cpp
- Basic_Types.h
- Basic_Types.inl
- Version.h

[Connection]

- Acceptor.cpp
- Acceptor.h **[doxygen]**
- Acceptor.inl
- Asynch_Acceptor.cpp
- Asynch_Acceptor.h **[doxygen]**
- Asynch_Acceptor.inl
- Asynch_IO.cpp
- Asynch_IO.h
- Asynch_IO.inl
- Connector.cpp
- Connector.h **[doxygen]**
- Connector.inl
- Dynamic_Service.cpp
- Dynamic_Service.h **[doxygen]**
- Dynamic_Service.inl
- Strategies.cpp
- Strategies.h
- Strategies.inl
- Strategies_T.cpp
- Strategies_T.h
- Strategies_T.inl

- Svc_Handler.cpp
- Svc_Handler.h **[doxygen]**
- Svc_Handler.inl

[IPC]

[IO_SAP]

- IO_SAP.cpp
- IO_SAP.h **[doxygen]**
- IO_SAP.inl

[DEV_SAP]

- DEV.cpp
- DEV.h **[doxygen]**
- DEV.inl
- DEV_Connector.cpp
- DEV_Connector.h **[doxygen]**
- DEV_Connector.inl
- DEV_IO.cpp
- DEV_IO.h **[doxygen]**
- DEV_IO.inl
- TTY_IO.cpp
- TTY_IO.h **[doxygen]**

[FILE_SAP]

- FILE.cpp
- FILE.h **[doxygen]**
- FILE.inl
- FILE_Connector.cpp
- FILE_Connector.h **[doxygen]**
- FILE_Connector.inl
- FILE_IO.cpp
- FILE_IO.h **[doxygen]**
- FILE_IO.inl

[IPC_SAP]

- IPC_SAP.cpp
- IPC_SAP.h **[doxygen]**
- IPC_SAP.inl

[Addr]

- Addr.cpp
- Addr.h **[doxygen]**
- Addr.inl
- DEV_Addr.cpp
- DEV_Addr.h **[doxygen]**
- DEV_Addr.inl
- FILE_Addr.cpp
- FILE_Addr.h **[doxygen]**
- FILE_Addr.inl

- INET_Addr.cpp
- INET_Addr.h **[doxygen]**
- INET_Addr.inl
- SPIPE_Addr.cpp
- SPIPE_Addr.h **[doxygen]**
- SPIPE_Addr.inl
- UNIX_Addr.cpp
- UNIX_Addr.h **[doxygen]**
- UNIX_Addr.inl
- UPIPE_Addr.h **[doxygen]**

[FIFO_SAP]

- FIFO.cpp
- FIFO.h **[doxygen]**
- FIFO.inl
- FIFO_Recv.cpp
- FIFO_Recv.h **[doxygen]**
- FIFO_Recv.inl
- FIFO_Recv_Msg.cpp
- FIFO_Recv_Msg.h **[doxygen]**
- FIFO_Recv_Msg.inl
- FIFO_Send.cpp
- FIFO_Send.h **[doxygen]**
- FIFO_Send.inl
- FIFO_Send_Msg.cpp
- FIFO_Send_Msg.h **[doxygen]**
- FIFO_Send_Msg.inl

[SOCK_SAP]

- LOCK_SOCK_Acceptor.cpp
- LOCK_SOCK_Acceptor.h **[doxygen]**
- LSOCK.cpp
- LSOCK.h **[doxygen]**
- LSOCK.inl
- LSOCK_Acceptor.cpp
- LSOCK_Acceptor.h **[doxygen]**
- LSOCK_Acceptor.inl
- LSOCK_CODgram.cpp
- LSOCK_CODgram.h **[doxygen]**
- LSOCK_CODgram.inl
- LSOCK_Connector.cpp
- LSOCK_Connector.h **[doxygen]**
- LSOCK_Connector.inl
- LSOCK_Dgram.cpp
- LSOCK_Dgram.h **[doxygen]**
- LSOCK_Dgram.inl
- LSOCK_Stream.cpp
- LSOCK_Stream.h **[doxygen]**

- LSOCK_Stream.inl
- SOCK.cpp
- SOCK.h **[doxygen]**
- SOCK.inl
- SOCK_Acceptor.cpp
- SOCK_Acceptor.h **[doxygen]**
- SOCK_Acceptor.inl
- SOCK_CODgram.cpp
- SOCK_CODgram.h **[doxygen]**
- SOCK_CODgram.inl
- SOCK_Connector.cpp
- SOCK_Connector.h **[doxygen]**
- SOCK_Connector.inl
- SOCK_Dgram.cpp
- SOCK_Dgram.h **[doxygen]**
- SOCK_Dgram.inl
- SOCK_Dgram_Bcast.cpp
- SOCK_Dgram_Bcast.h **[doxygen]**
- SOCK_Dgram_Bcast.inl
- SOCK_Dgram_Mcast.cpp
- SOCK_Dgram_Mcast.h **[doxygen]**
- SOCK_Dgram_Mcast.inl
- SOCK_IO.cpp
- SOCK_IO.h **[doxygen]**
- SOCK_IO.inl
- SOCK_Stream.cpp
- SOCK_Stream.h **[doxygen]**
- SOCK_Stream.inl

[SPIPE_SAP]

- SPIPE.cpp
- SPIPE.h **[doxygen]**
- SPIPE.inl
- SPIPE_Acceptor.cpp
- SPIPE_Acceptor.h **[doxygen]**
- SPIPE_Acceptor.inl
- SPIPE_Connector.cpp
- SPIPE_Connector.h **[doxygen]**
- SPIPE_Connector.inl
- SPIPE_Stream.cpp
- SPIPE_Stream.h **[doxygen]**
- SPIPE_Stream.inl

[TLI_SAP]

- TLI.cpp
- TLI.h **[doxygen]**
- TLI.inl
- TLI_Acceptor.cpp

        - TLI_Acceptor.h **[doxygen]**
        - TLI_Acceptor.inl
        - TLI_Connector.cpp
        - TLI_Connector.h **[doxygen]**
        - TLI_Connector.inl
        - TLI_Stream.cpp
        - TLI_Stream.h **[doxygen]**
        - TLI_Stream.inl

    [UPIPE_SAP]

    - UPIPE_Acceptor.cpp
    - UPIPE_Acceptor.h **[doxygen]**
    - UPIPE_Acceptor.inl
    - UPIPE_Connector.cpp
    - UPIPE_Connector.h **[doxygen]**
    - UPIPE_Connector.inl
    - UPIPE_Stream.cpp
    - UPIPE_Stream.h **[doxygen]**
    - UPIPE_Stream.inl

    [Misc]

    - IOStream.cpp
    - IOStream.h **[doxygen] [doxygen]**
    - IOStream_T.inl
    - Pipe.cpp
    - Pipe.h **[doxygen]**
    - Pipe.inl
    - Signal.cpp
    - Signal.h
    - Signal.inl

[Logging and Tracing]

- Dump.cpp
- Dump.h
- Dump_T.cpp
- Dump_T.h
- Log_Msg.cpp
- Log_Msg.h **[doxygen]**
- Log_Msg.inl
- Log_Priority.h
- Log_Record.cpp
- Log_Record.h **[doxygen]**
- Log_Record.inl
- Trace.cpp
- Trace.h **[doxygen]**
- Trace.inl

[Memory]

[Mem_Map]

- Mem_Map.cpp
- Mem_Map.h **[doxygen]**
- Mem_Map.inl

[Shared_Malloc]

- Malloc.cpp
- Malloc.h **[doxygen]**
- Malloc.inl
- Malloc_T.cpp
- Malloc_T.h
- Malloc_T.inl
- Memory_Pool.cpp
- Memory_Pool.h
- Memory_Pool.inl

[Shared_Memory]

- Shared_Memory.h **[doxygen]**
- Shared_Memory_MM.cpp
- Shared_Memory_MM.h **[doxygen]**
- Shared_Memory_MM.inl
- Shared_Memory_SV.cpp
- Shared_Memory_SV.h **[doxygen]**
- Shared_Memory_SV.inl

[Utils]

- Obstack.cpp
- Obstack.h **[doxygen]**
- Read_Buffer.cpp
- Read_Buffer.h **[doxygen]**
- Read_Buffer.inl

[Misc]

- ARGV.cpp
- ARGV.h **[doxygen]**
- ARGV.inl
- Auto_Ptr.cpp
- Auto_Ptr.h
- Auto_Ptr.inl
- Date_Time.cpp
- Date_Time.h **[doxygen]**
- Date_Time.inl
- Dynamic.cpp
- Dynamic.h **[doxygen]**

- Dynamic.inl
- Get_Opt.cpp
- Get_Opt.h **[doxygen]**
- Get_Opt.inl
- Registry.cpp
- Registry.h **[doxygen]**
- Singleton.cpp
- Singleton.h **[doxygen]**
- Singleton.inl
- System_Time.cpp
- System_Time.h **[doxygen]**

[Name_Service]

- Local_Name_Space.cpp
- Local_Name_Space.h **[doxygen]**
- Local_Name_Space_T.cpp
- Local_Name_Space_T.h **[doxygen]**
- Name_Proxy.cpp
- Name_Proxy.h **[doxygen]**
- Name_Request_Reply.cpp
- Name_Request_Reply.h
- Name_Space.cpp
- Name_Space.h **[doxygen]**
- Naming_Context.cpp
- Naming_Context.h **[doxygen]**
- Registry_Name_Space.cpp
- Registry_Name_Space.h **[doxygen]**
- Remote_Name_Space.cpp
- Remote_Name_Space.h **[doxygen]**

[OS Adapters]

- ACE.cpp
- ACE.h **[doxygen]**
- ACE.inl
- OS.cpp
- OS.h **[doxygen]**
- OS.inl

[Reactor]

- Event_Handler.cpp
- Event_Handler.h **[doxygen]**
- Event_Handler.inl
- Event_Handler_T.cpp
- Event_Handler_T.h **[doxygen]**
- Event_Handler_T.inl
- Handle_Set.cpp
- Handle_Set.h **[doxygen]**

- Handle_Set.inl
- Priority_Reactor.cpp
- Priority_Reactor.inl
- Priority_Reactor.h **[doxygen]**
- Proactor.h **[doxygen]**
- Proactor.inl
- Proactor.cpp
- Reactor.cpp
- Reactor.h **[doxygen]**
- Reactor.inl
- Reactor_Impl.h **[doxygen]**
- Select_Reactor.cpp
- Select_Reactor.h
- Select_Reactor.inl
- WFMO_Reactor.cpp
- WFMO_Reactor.h **[doxygen]**
- WFMO_Reactor.inl
- XtReactor.cpp
- XtReactor.h **[doxygen]**

[Service_Configurator]

- DLL.cpp
- DLL.h **[doxygen]**
- Parse_Node.cpp
- Parse_Node.h **[doxygen]**
- Parse_Node.inl
- Service_Config.cpp
- Service_Config.h **[doxygen]**
- Service_Config.inl
- Service_Manager.cpp
- Service_Manager.h **[doxygen]**
- Service_Manager.inl
- Service_Object.cpp
- Service_Object.h **[doxygen]**
- Service_Object.inl
- Service_Repository.cpp
- Service_Repository.h **[doxygen]**
- Service_Repository.inl
- Service_Types.cpp
- Service_Types.inl
- Service_Types.h
- Shared_Object.cpp
- Shared_Object.h **[doxygen]**
- Shared_Object.inl
- Svc_Conf.h
- Svc_Conf_l.cpp
- Svc_Conf_y.cpp
- Svc_Conf_Tokens.h

[Streams]

- IO_Cntl_Msg.cpp
- IO_Cntl_Msg.h **[doxygen]**
- IO_Cntl_Msg.inl
- Message_Block.cpp
- Message_Block.h **[doxygen]**
- Message_Block.inl
- Message_Queue.cpp
- Message_Queue.h **[doxygen]**
- Message_Queue.inl
- Message_Queue_T.cpp
- Message_Queue_T.h
- Message_Queue_T.inl
- Module.cpp
- Module.h **[doxygen]**
- Module.inl
- Stream.cpp
- Stream.h **[doxygen]**
- Stream.inl
- Stream_Modules.cpp
- Stream_Modules.h
- Stream_Modules.inl
- Task.cpp
- Task.h **[doxygen]**
- Task.inl
- Task_T.cpp
- Task_T.h
- Task_T.inl

[System_V_IPC]

[System_V_Message_Queues]

- SV_Message.cpp
- SV_Message.h **[doxygen]**
- SV_Message.inl
- SV_Message_Queue.cpp
- SV_Message_Queue.h **[doxygen]**
- SV_Message_Queue.inl
- Typed_SV_Message.cpp
- Typed_SV_Message.h **[doxygen]**
- Typed_SV_Message.inl
- Typed_SV_Message_Queue.cpp
- Typed_SV_Message_Queue.h **[doxygen]**
- Typed_SV_Message_Queue.inl

[System_V_Semaphores]

- SV_Semaphore_Complex.cpp
- SV_Semaphore_Complex.h **[doxygen]**

    - SV_Semaphore_Complex.inl
    - SV_Semaphore_Simple.cpp
    - SV_Semaphore_Simple.h **[doxygen]**
    - SV_Semaphore_Simple.inl

  [System_V_Shared_Memory]

    - SV_Shared_Memory.cpp
    - SV_Shared_Memory.h **[doxygen]**
    - SV_Shared_Memory.inl

[Timers]

- High_Res_Timer.cpp
- High_Res_Timer.h **[doxygen]**
- High_Res_Timer.inl
- Profile_Timer.cpp
- Profile_Timer.h **[doxygen]**
- Profile_Timer.inl
- Time_Request_Reply.cpp
- Time_Request_Reply.h
- Time_Value.h **[doxygen]**
- Timer_Hash.cpp
- Timer_Hash.h
- Timer_Hash_T.cpp
- Timer_Hash_T.h **[doxygen]**
- Timer_Heap.cpp
- Timer_Heap.h
- Timer_Heap_T.cpp
- Timer_Heap_T.h **[doxygen]**
- Timer_List.cpp
- Timer_List.h
- Timer_List_T.cpp
- Timer_List_T.h **[doxygen]**
- Timer_Queue.cpp
- Timer_Queue.h
- Timer_Queue.inl
- Timer_Queue_Adapters.cpp
- Timer_Queue_Adapters.h
- Timer_Queue_Adapters.inl
- Timer_Queue_T.cpp
- Timer_Queue_T.h **[doxygen]**
- Timer_Queue_T.inl
- Timer_Wheel.cpp
- Timer_Wheel.h
- Timer_Wheel_T.cpp
- Timer_Wheel_T.h **[doxygen]**

[Token_Service]

- Local_Tokens.cpp
- Local_Tokens.h
- Local_Tokens.inl
- Remote_Tokens.cpp
- Remote_Tokens.h
- Remote_Tokens.inl
- Token_Collection.cpp
- Token_Collection.h **[doxygen]**
- Token_Collection.inl
- Token_Manager.cpp
- Token_Manager.h **[doxygen]**
- Token_Manager.inl
- Token_Request_Reply.cpp
- Token_Request_Reply.h
- Token_Request_Reply.inl
- Token_Invariants.h
- Token_Invariants.inl
- Token_Invariants.cpp