RUN-TIME INTERFACE MAPPING FOR ON-THE-FLY SOFTWARE COMPONENT INTEGRATION

by
Thanh Van Lam

Submitted in partial fulfillment
of the requirements for the degree of

Doctor of Professional Studies in Computing

at

School of Computer Science and Information Systems

Pace University

September 2002

Thanh Van Lam, 2002

We hereby certify that this dissertation, submitted by Thanh Van Lam, satisfies the dissertation requirements for the degree of Doctor of Professional Studies in Computing and has been approved

____-____________
Lixin Tao Date
Chairperson of Dissertation Committee

____-____________
Sung-Hyuk Cha Date
Dissertation Committee Member

____-____________
Mary Courtney Date
Dissertation Committee Member

School of Computer Science and Information System
Pace University 2002

Abstract

Run-Time Interface Mapping

for On-the-fly Software Component Integration

by
Thanh Van Lam

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies in Computing

September 2002

A logical process in making software components is to divide a huge source code into smaller units that can be designed and implemented independently. Problems rise when those small components are put back together into an entirely functional and reliable application. Existing approaches aim at providing global environments to facilitate component integration in their domains including common services such as communications, interoperating abilities, security, and reliabilities. However, those frameworks come with the price of complex infrastructures and extraneous or unnecessary resources, without which some applications can certainly run. On the other hand, programmers who write programs without frameworks have to reinvent their own common services. Although they might have found their solutions, other programmers might not be aware of that. The run-time interface mapping approach proposed in this thesis enables component reuse with no heavy framework requirements. Software components that adhere to this approach can be loaded on-the-fly at run time. To support our approach, we implement a local environment facility that provides services for searching and loading components at run time. Hence, applications can assemble components dynamically while they are running. Applications can also share local components by using the local environment registry. Each local environment may contain varieties of components based on the same interface but are different in implementations or user preferences. We use a packaging tool to incorporate the run-time interface mapping methodology with local components that take advantage of the local environment facility. Automated packaging procedures ensure delivering of components and applications to a local system. Component makers can implement as many components as they want for an interface based on local environment characteristics. Application integrators publish and solicit for their requirements, then build applications based on component interfaces. Both component makers and application integrators benefit from component specifications. Components can be delivered and run in environments that they are designed and implemented for. This is a beginning step in building adaptive and efficient components and applications.

Acknowledgments

Writing this thesis and keeping up with the DPS program for three years is as tough as life. It is especially life with intensive, serious, and accelerated demands coming from three directions: work, family, and school. A colleague congratulated me with expression of amazement to my juggling acts, “It’s nice to see the ‘good guys’ succeed.” To the good guys and gals of Pace DPS 2002, and years after that, I’ve been there, in the eye of the tiger.

I would like to dedicate this to my parents for their unconditional love. They would never understand how my code has tangled up with endless number of conditions. Thanks to my wife, Song, for sacrifices and understanding. She has been taking care of all the chores around the house and raising our child. Thanks to my daughter, Vanessa, from whom I learn how to learn like a baby, in the early years of the cycle of life. As a family, we felt the deepest pains of losing Song’s parents. It hurt more for both times we could not pay the last visit. This is a tribute to them. We would have wished life were different.

This thesis would not be complete without the guidance from my advisor, Dr. Lixin Tao, who has helped in many aspects of my thinking, writing and publishing of the thesis. Thanks to the Committee Members, Dr. Sung-Hyuk Cha and Dr. Mary Courtney, for comments and suggestions. Thanks to the entire faculty members of the DPS program at Pace University, especially Dr. Fred Grossman, for tireless encouragement and support. Thanks to my classmates for sharing and helping each other. I’ve had the privilege to be part of a group of exceptional professionals. These three years have been the highlights in my life.

I’d like to thank my manager and my colleague at IBM, Ed Oliva, for the opportunity and the challenges. Thanks to IBM for those Learning Assistance Programs that make all this possible.

Table of Contents

Abstract iii

Acknowledgments iv

List of Figures ix

List of Tables x

Chapter 1 Introduction 1

1.1 The Problem of Component Integration *3*

1.2 Software Components Local Environment Approach *5*

1.2.1 Objectives 5

1.2.2 Approach 7

1.2.3 Contributions 11

1.3 Thesis Organization *15*

Chapter 2 Software Component Technologies Status 18

2.1 A Brief History of Software Component *18*

2.2 Component Characteristics *19*

2.2.1 Multiple Integration Stages 19

2.2.2 Multiple layers 23

2.2.3 Diverse packaging requirements 25

2.3 Component Frameworks *26*

2.3.1 DLL, COM+, and .NET 28

2.3.2 CORBA 3 29

2.3.3 Enterprise JavaBeans 30

2.4 Environment as Container *32*

2.4.1 Java Virtual Machine 32

2.4.2 J2EE containers 35

2.5 Component Packaging *36*

2.5.1 Installation Application 36

2.5.2 EJB Deployment Descriptor 37
2.5.3 Package Manager (RPM) 39
2.6 Summary 41

Chapter 3 Overview of Dynamic Application Integration 43

3.1 Overall Architecture and Organizations 45
3.1.1 Run-time Interface Mapping (RIM) 45
3.1.2 Local Environment Registry (LER) 46
3.1.3 Packaging components with application 49
3.2 Application Integration Model 50
3.2.1 Programming skills and roles 50
3.2.2 Supportive roles of LER 51
3.2.3 Writing a simple application: ZipAF 52
3.3 Differences and advantages 53
3.3.1 Dynamic run-time loading versus common objects 55
3.3.2 Component packaging versus distributed modeling 57
3.3.3 Local environment versus remote servers and containers 59
3.4 Summary 61

Chapter 4 Local Environment Registry 62

4.1 LER Requirements and Organizations 63
4.1.1 Persistent storage 64
4.1.2 Component references 65
4.1.3 Searchable local environment 66
4.1.4 LER application interface 67
4.2 LER Internals 68
4.2.1 Registry engine 69
4.2.2 Interface-component reference 71
4.2.3 Local component matcher 75
4.3 LER Application Programming Interface 78
4.3.1 LER installation and maintenance methods 78
4.3.2 Interface registering and loading methods 80
4.3.3 Component managing methods 83
4.3.4 Local environment management methods 85
4.4 LER as a User Component 88

Chapter 5 Run-time Interface Mapping **91**

5.1 RIM Methodology and Requirements *92*
5.1.1 Symbolic referencing 93
5.1.2 Dynamic binding 95
5.1.3 Dynamic code loading with no semantics 96
5.1.4 Trusted components 97
5.1.5 Component specifications and documentations 98
5.1.6 Binary code compatible 98

5.2 RIM and Component Methods *99*
5.2.1 Interface Definition 100
5.2.2 One interface, multiple components 102
5.2.3 Regular arguments 104

5.3 RIM and Component Namespace *105*
5.3.1 Local components and their namespace 105
5.3.2 LER as component dealer 108

5.4 Other Programming Languages Consideration *113*

Chapter 6 Component Packaging with RIM and LER **115**

6.1 Packaging Requirements and Guideline *117*
6.1.1 The LER API 117
6.1.2 System native interfaces 120
6.1.3 Component delivering 122
6.1.4 Components for sharing 122

6.2 Putting together a package *123*

Chapter 7 Examples of Application Integration **126**

7.1 Virtual Pet Application *128*

7.2 Virtual Pet Requirements and Analysis *130*
7.2.1 Write once, integrated on many systems 130
7.2.2 Adapting to individual system 131
7.2.3 Programming interface considerations 131
7.2.4 High priority for owner preferences 134

7.3 Virtual Pet Interfaces *135*
7.3.1 LER interface 136
7.3.2 Existence interface 137

7.3.3 Activity interface 139
7.3.4 Control interface 141
7.4 Virtual Pet Design and Integration 143
7.5 Virtual Pet Delivery 144
7.6 Conclusion 145
Chapter 8 Future Work 147
References 153

List of Figures

Figure 1 Source file decomposition ... 3
Figure 2 FlashLine Java Component Marketplace ... 24
Figure 3 FlashLine .Net/ActiveX/COM Marketplace ... 25
Figure 4 Three-Tier-View: Application, Components, Component Frameworks ... 27
Figure 5 Run-time Interface-Mapping Components ... 46
Figure 6 LER Organization ... 68
Figure 7 LER Interface-Components Reference ... 72
Figure 8 Interface with multiple attributes ... 73
Figure 9 Local component matching ... 76
Figure 10 Attribute enumeration and priority ... 78
Figure 11 Interface Requirement Documentation ... 100
Figure 12 One interface, multiple sub-components ... 103
Figure 13 Nested components at run time ... 104
Figure 14 Alternative LER internal namespace ... 108
Figure 15 Packaging tools ... 124

List of Tables

Table 1 LER User Confiruration Table ... 47
Table 2 LER Interface Reference Table ... 48
Table 3 Summary of LER API... 49
Table 4 LER User Configuration Table for ZipAF ... 51
Table 5 LER Interface Reference Table for ZipAF ... 51

Chapter 1
Introduction

People in software development have been known for their strategy of dividing and conquering. When a problem becomes complex and hard to handle, it is broken down to smaller and more manageable size. The method of decomposition can be applied in all stages of software development cycle, from problem analyses, data structure designs, to program implementation. Especially in big software projects, in which many people are working at the same time, breaking down the pieces has helped in dividing the amount of work among groups or team members.

Thus, for ease of developing and maintaining with the changing pace of technology, software makers continue to divide the computing world into disparate entities, from huge platforms to little components. There is no one standard that defines what a specific component should be. The creator of a component should define what his component is composed with and what purposes the component is supposed to serve. Since we are looking into common usages of software components, we derive two characteristics of a component:

1. It has well defined external interfaces and context dependencies.
2. It supports independent deployments.

A component's external interfaces are what it offers as services for other components in order to use it. Context dependencies, on the other hand, are conditions a component requires or depends upon being provided for it in order to run.

The two characteristics require and complement each other. Application integrators demand for "plug and play" components, which can be reused without the needs of knowing their internals or their development processes. Integrators do not want to depend on component makers in getting the components to work or in debugging component problems. Component makers, on the other hand, want their components to be reused in as many applications as possible.

A third characteristic of components is that they should be in binary format, for the purposes of independent deployment. Source code is not always available with pre-built components. Besides, source code analysis and modification add complexity and difficulties in reusing of components because of the diversity in original designs and needs in reuse. Programmers' skill levels are also diverse factors. Software component provides the ability of using or replacing similar components without bogging down in debugging errors. Documentations and specifications are keys. Both the integrator and the component maker benefit from clearly defined component interfaces and contexts. Our approach gives a more specific and detailed definition of component in Chapter 3.

Software components have been around for many years with the premise of dividing and conquering complex problems in software development. Programmers have had different ways of decomposing a big software structure such as an application design or a logical unit of code into smaller units that are treated in different techniques or terminology called: subroutine, module, function, procedure, library, etc. But all have one common purpose: reusing of available code. Therefore, we have seen the enthusiasm and hope for broader use of components coming back again and again.

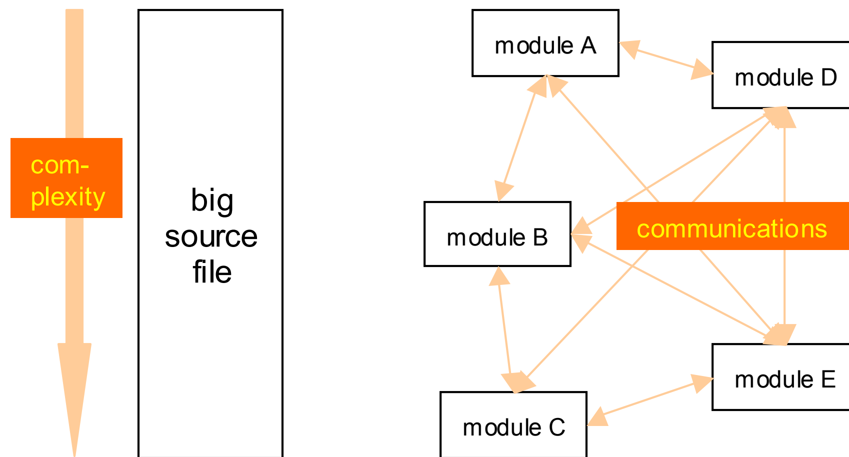

Figure 1 Source file decomposition

When a big source file is broken down into smaller files or modules in some logical structures, the programmer needs to understand only the small modules. From a component maker's perspectives, the module writer does not need to know global application structures or the internal logic of other modules. The complexity in working with big source files is reduced. From an application integrator's perspectives, these modules can be treated as reliable black boxes. Application builder only needs to focus on application level structures. See illustration in Figure 1.

1.1 The Problem of Component Integration

Small components, however, need to be put back together for a complete solution or application. Usually, the process of integrating components into an application occurs in development environment. Programming tools such as assembler or compiler link the object files into executables. All the required components have to be present. Otherwise, errors will occur. With language specific and referencing mechanisms, however, not all modules have to be included within the application executable. Some of the modules

may be provided separately in code library format. Nevertheless, they are linked into the application at one time or another.

Because statically linking of code in development time limits application to fixed code, different methodologies have been introduced to enable dynamic loading of code. A program can reference the name of code that will be available at run time. When the application comes to access that part of code at run time, it has more options than just calling the modules. For example, if the calling of the local code or library code failed, it can be set up to search for the exact piece of code in other locations. It can also try to call different names for the code. All those logic and decisions are made at program development time. As a result, complex code is written for checking of possible conditions.

Other solutions include providing some virtual environments at run time to provide the applications with all types of resources and services. These are known as middleware or component frameworks. They usually cover global environments that include all the things that an application programmer ever wished for. Their required learning curve and complex interfaces often deter attempts in using them. Applications that run inside these global environments depend completely on them to integrate components. However, what these environments provide may be good for certain types of applications but too much for others.

Although development environments may provide plenty of tools and procedures for purposes of building application from components, real problems such as improper interfaces or implementations often occur in run-time environments. Development

environment can help to prepare for the worst of those problems but many diverse run-time environments make solving run-time problems difficult.

1.2 Software Components Local Environment Approach

This thesis proposes solutions for application integration based on discrete local environments using run-time interface mapping methodology. We started with following objectives.

1.2.1 Objectives

1. *This methodology requires minimal programming involved in the development of components being used.* Assuming that components are already available in varieties, the task of building an application is mainly integrating the components. The application programmer takes on the roles of an integrator who starts from a programming level that he can maximize the reuse of components. His tasks then are to decide which components to use and which environments are appropriate to run in. Integrating applications from pre-built components is difficult to start with, because of the disarray characteristics of development and run-time environments. Application integrators and Component makers are often involved in collaborating processes of designing and implementing the components. Much time and effort are invested and accumulated from early development cycles. Our goal is to give application integrator the flexibility in testing and choosing different pre-built components. This approach also benefits component makers who may want to change their implementations for quick responses to changing demands of component markets.

2. *Component integration at run time enables application adaptation*. The goal of building an application in this thesis is to find out what environments are available and which components can be used for the purposes of adapting to those environment conditions. We follow the rules of starting simple and progressing to achieve our goals. The simplest environment is an initial one that has no application running. When an application package is installed, it adds its components and accompanying requirements to the local environment. On the other hand, an application can use components that are already available in the environment. If an application's requirements are already satisfied within the environment, then the application does not have to include logic to check for those requirements. Applications that react this way in development environment will react similarly in run-time environments.
3. *Packaging of components plays important role in getting the right components to the right local environment*. Our approach enables applications to dynamically adapt to local environment at run time without depending on heavy boilerplate containers to provide virtual global environments and fixing the component binding statically at deployment times. We use packaging in combination with a local environment facility that applications can access at run time. Component interfaces can be shared among applications. In order for this to work, each component in use by the application and available in the local environment has to follow a strict interface specification. However, the internal implementation of the component is still depending on the component maker. Actually, this is one of the advantages of using components from different makers. Component maker

identifications, component implementations and local resident locations, and local attributes are all parts of the component namespaces. Application integrator has the options of requesting for unique interface names meeting their specific requirements.

1.2.2 Approach

Our literature surveys show that existing middleware infrastructures and frameworks are complex to start with. They also require investments in time and resource for installations and operations due to the requirements for their common services such as communications, security, and interoperability. The framework has to be in operations before any work on an application can begin. The application is thought of as a part of those larger and predefined computing scenarios, such as:

- Distributed computing: the division of data and application segments have to be well lay out in the designs and implementations of the application.
- Client/Server model: different parts of the application are assigned different roles based on resources and their usage.
- Enterprise computing: these are transaction-processing types of applications, which are usually controlled by component transaction monitors.

Hence, middleware and frameworks are usually introduced to fulfill those predefined requirements. By setting a new application in those environments, it does not have to implement those environments or infrastructures again. However, knowledge and experiences in those environments have to be acquired in time. It is a top-down approach. The programmer has to go through and meet all those requirements before she can start her program.

1.2.2.1 A bottom-up approach

In contrary, we want to start with a simple application integration environment that does not need all of those complex services, especially when the environment is restricted to a local machine. We start from a bottom-up perspective when the integrator putting together an application. There is a small set of local facility that is needed. First, we build the simple local facility as a component for use in an application. Then, we make that a common component for other application integrators to reuse.

Instead of searching for some global environments that can be virtual environments for application integration at run time, this thesis turns to study the local environment that facilitates the run-time integration process. Usually, local environments make up the diversity of application run-time environments. Components can have varieties in different local environments. We take advantage of dynamic loading of local components to resolve application features, functionalities, or customizability.

When integrating components into application, an application integrator uses interface ID to refer to a component that is only loaded dynamically at run time. The integrator has no need to know which component will be loaded. However, she needs to ensure that the component provides the data fields and methods as specified by the interface. Thus, we emphasize the importance of interface and component requirements and specification documents.

Our approach is to establish a simple but fundamental methodology in component reuse. Application integrators play active roles in promoting component reuse, designing the interfaces and requirement documentations. However, the two processes of making and choosing components are as independent as they can be. Based on agreeable

interface requirements, component makers implement as many components as they can in their familiar environments. These components are made available in local environments by means of packaging and delivering. By limiting to local components, we increase the chances of component similarity and compatibility. Thus, components can be shared among applications in the local environment. Applications look within its local environment first for trusted components.

1.2.2.2 Starting from simple program and interfaces

To start writing a program from existing components, a programmer first decides which interfaces the program needs. These interfaces become requirements for successful completion of the program. He searches his own systems or other systems he knows for components that support those interfaces. Obtaining those components and installing them onto the development system is the first step in integrating the application. The programmer's work of integrating these components into his application is one of the many stages of deploying software components.

If some of the required interfaces do not have supporting components, the programmer writes his components for the interfaces. The concept of interface is similar to the Java language Interface or general object oriented methodology in software development. Choosing to implement an interface in different ways is common practices in writing program. However, choosing an intended implementation for an interface at run time requires much programming logic. Although the concept is simple, it is most important that the appropriate component is loaded at run time.

Hence, the application integrator should have ways of specifying his unique interface at run-time. These alternatives are ranging from letting the local environment facility

search and load the component that matches the interface to specifying the particular component to load. By default, the unique interface name is composed by different parts of its namespace. It is most flexible to let the local environment facility come up with the interface identification internally at run time. However, application integrators sometimes need to be able to use the specific components.

The local conditions of the system play important roles in running the application or using the components differently. The application or components need a supporting local facility that keeps track of the components under its control or changing of the environments. This facility takes on the decision process, which includes the searching and loading of a component. Mapping of an interface to a local component is done via unique interface identifications by the local facility. All the programming logic happens in the local facility. The application just makes requests for these identifications. Both the application integrator and the component maker receive the benefits of flexibility in making and reusing components.

The complexity of programming logic to check for different local environments is moved out from application integration logic. Some of the logic is built into the packaging strategy and local environment facility. However, these common components are local environment aware. They are tested in the environments they are supposed to run in. These components, in turns, protect the application from dealing with different local environments. Thus, there are no untested code paths in the application.

1.2.2.3 Deployment cycles

To put together the interface solutions that include both the appropriate components and the local environment facility, we need a packaging method that is also local environment

oriented. Although there are many available packaging methods, our package delivery method depends on and takes advantage of the local facility. Customized package delivers just the required components to each local environment. Application efficiency is ensured on the local system.

The application integrator puts together the package that includes its local facility and components. When this package is installed on a new system, it checks to see if that system has already had the local registry facility, which is the same common component with different contents due to local environments. If the local registry facility exists, the package registers all its interfaces and their corresponding components. If an interface already exists and known by the local facility, it is not registered. The intention is to reuse existing local interfaces and components. Similarly, duplicate components are not added. New installed application reuses and shares with existing applications running on this system.

If a local registry facility does not exist, then this is the first package being installed on the system. The local registry facility is installed and all of the interfaces and components are registered and added respectively. The same procedures of packaging and delivering components or applications recur on any system like cycles of deployment. Each time a new package is installed, interfaces and components are checked against duplications. Each system is kept to minimum components. There are no unused components, from application or system viewpoints.

1.2.3 Contributions

The ability of mapping an interface to a selected component at run time is a major contribution of this thesis. Beyond the general concepts of interface, this approach eases

programmers at starting to write a program. She does not have to make all the decisions upfront, at development time, on choosing component frameworks, components, or run-time environments.

Well-defined requirements and specifications of interfaces are to put contractual rules on how components are built and reused. Although much of the interface can be verified at compiling or linking time, our approach pushes for loading time or run-time verification. Thus, components can be selected for adapting to different run-time environments.

However, this approach does not require the application to be built with all logic and conditional checking in decision-making procedures. Our implementation of the local facility does the work in providing components for the requested interface. This facility registers interfaces and does searching and loading for appropriate components at run time. Because we separate these functions from an application, components and applications become more efficient.

The local facility functionalities can be implemented in different ways. Hence, it is possible to have different interface mapping methodology. That opens up the capabilities of interface referencing to diverse groups of components that may be used in very different purposes such as simulated components or test components.

1.2.3.1 Reuse of components

A prerequisite for the approach of calling interfaces is that, the calling code has to specify the name of the called method and its parameters in order to make the call to the external module. Usually the called interfaces are designed or defined first. Then, the calling code is written following the interfaces. Unless the called module has been in existence

for a long time and has been used for many times, interactivity and re-factoring are often required between the two programmers. Our approach emphasizes the use of available components. It eases the procedures of selecting components at run time when the application does not have to be aware of that. Our implementation of the local environment registry provides services enabling component searching and loading for applications at run time.

In component making and application integration, usually the integrator has no controls over how or what components are made available. If he cannot find what he needs in existing components, he ends up writing his own components. For that reason, our approach lets the integrator make the calls on what component interfaces he needs. He can solicit existing interfaces and choose the ones that fit the requirements of his application. Once the decision is made, component makers have to follow the interfaces strictly. The integrator can also go as far as defining new interfaces. Component makers, however, still have the freedom in implementing their components differently, as long as they adhere to the interface requirements.

1.2.3.2 Dynamic component integration

Dynamically loading components through the local environment registry is different from using libraries. In general, libraries have some drawbacks. If it is statically linked to a program, it might include lots of subroutines that the program never calls. Dynamic linking is an improvement over calling subroutines.

At compile time, a program can just contain reference to the subroutines but not be linked to the actually code in the library, which can be loaded into memory independently from the program. Then at run time, the library subroutines are available whenever a

program makes calls to those subroutines. Added advantage of dynamic linking library is that many programs can share one copy of the library in memory. Because components and libraries are in binary code format, they need to be linked into or referenced from the main application at some stage. Linking of the components is static in development time. Referencing to library modules is resolved at run time.

We design a reference layer based on the concept of interface. The difference is that the calling program uses these interfaces as references or placeholders. On the other side of the interface reference is the local environment facility that supplies the actual component that implements the interface. Thus, calling of interfaces is independent of linked or referenced code but depending on local conditions and environments.

1.2.3.3 Application adapting to local environment

Although each application can be made for one type of solutions, the value of an application is for solving similar problems in many different conditions. Those conditions exist in local environments that the application runs in. Combine all of those conditions and we have globalized environments. But, such a virtual environment that attempts to include or fit all different local environments will be enormous and complex. A local environment facility is necessary for many practical purposes.

With this approach, applications can dynamically adapt to local environments at run time. We define a local environment facility to enable component reuse by actively providing well-defined interfaces as references to those components. The implementation of this local facility is small compared to other virtual global environments. Components that are built with these local environments in mind can eliminate extraneous conditional logic and unnecessary code paths. A program depends

on the local environment facility to make decisions as per what is suitable with this environment.

1.2.3.4 User benefits

To the application integrator, new components and applications that follow this methodology are in better control and maintenance. Each component is accompanied with specifications so that it can be bought or reused following the guidelines. Component external interfaces and context dependencies are well defined for purposes of integration by third party.

Last in the chain of application integration is the user who is the keeper or owner of her own environment. She also benefits tremendously from this methodology because the efficient usage of components makes an application compact. No extraneous code that wastes user resources. Applications are highly customizable and easy for upgrading.

We set up the first step for future work in component loading at run time for dynamic programming semantics. Components can also be dynamically searched and located for utilization purposes. The local environment facility can be used as central accessing point for purposes of component reuse.

1.3 Thesis Organization

Chapter 1, "**Introduction**", gives overviews of software components and their usage, which leads to the discussions of problems in application integration especially when putting many disarray of existing components back into a fully functional application. Run-time interface mapping is the foundation of this thesis, which contributes the

software component integration methodology for dynamic application assembly in local environment at run time.

Chapter 2, "**Software Component Technologies Status**", surveys the current component technologies, widely known frameworks such as COM+, .NET, CORBA, and Enterprise JavaBeans, and delves into details of component integration in order to understand the problem at hand. Component linking and compiling have been used in developing applications. However, this thesis focuses in dynamic component loading at run time.

Chapter 3, "**Overview of Dynamic Application Integration**", introduces the overall architecture and methodology for this thesis. A fictitious application called `ZipAF` is used as a sample demonstration for introducing the methodology and its concepts. With this simple example, Run-time Interface Mapping and Local Environment Registry are easier to understand.

Chapter 4, "**Local Environment Registry**", details the API provided by the LER component, which is intended for programs following this methodology. LER is at the center of the application and component interactivities. It resides in the user environment and does local component loading for applications. It also supports application decision making related to local environment conditions.

Chapter 5, "**Run-Time Interface Mapping**", shows the thesis work in deriving the abstraction of the interface to be used by components for integrating into an application that follows this approach. It is described using the Java programming language. More general approach is derived at the end.

Chapter 6, "**Component Packaging With LER and RIM**", describes the common ideas and strategies in packaging and delivering software components. Independent of package format, package delivery can be used in conjunction with maintaining the uniqueness of local environments. The same procedures are carried out through deployment cycles aiming at reusing and sharing components in local system.

Chapter 7, "**Examples of Application Integration**", describes an application put together with this methodology for demonstration purposes. The unique but diverse virtual pets illustrate the working model of mapping interfaces to different components in different local environment at run time.

Chapter 8, "**Future Work**", continues the trends of dynamic component integration at run time: dynamic interfaces. Component search engines can be an extension to LER for more dynamic application semantics. Tentative goals for applications running in local environment include adaptive and efficient software components.

Chapter 2
Software Component Technologies Status

2.1 A Brief History of Software Component

The ideas of reusing software are as old as any programming languages. FORTRAN has subroutines. Modula has modules. Pascal has functions. Programming language designers have long realized that breaking source code into smaller chunks eases code maintenance and program writing efforts. By structuring the program in modular fashion, subroutines can be written in the same source file but in separate units that can be invoked independently from their location. Hence, separation of the calling interfaces and their modules is achieved.

Once successfully tested and used, the subroutines can be physically separated into different source files that are compiled into binary object files. Then, the same binary object files can be linked into different applications. Linkable object code is one form of software component that has been widely adopted in languages such as FORTRAN or C. The separated source files can also be compiled into library file and statically linked to a main program at compile time. This is the starting point of linking a program with external components. To increase reusability, modules are packaged into libraries for general or specific programming needs. Software makers can write and sell software libraries for the purposes of reuse.

With the introduction of object oriented programming languages, objects become the hopeful reusable entities. Some language such as Smalltalk establishes its own environment for objects to execute and interact with each other. Visual Basic, although

was not originally an object oriented programming language, provides Object Linking and Embedding (OLE) techniques for applications to exchange object code at run time. The underline architecture is Component Object Model (COM), which enables object interactions independent of where the objects located within the Windows operating environment. Distributed Component Object Model (DCOM) expanded COM standards to multiple systems in network environments.

2.2 Component Characteristics

2.2.1 Multiple Integration Stages

Both traditional and component software models make use of the modular code composition methods. Small units of code called modules can be combined at different stages of the development cycles to produce a functional application. In general, software components include very broad ranges of code modules or units that can be integrated into application at different time and by different programmers with different levels of programming skill.

2.2.1.1 Compiling and linking time

At compiling time, one or more program source files are translated into binary object files in the form of machine language suitable for a particular operating system or hardware platform. If necessary, multiple object files are combined or linked into one executable or image file. Most things are fixed in this stage. An example of such image file is the Common Object File Format (COFF) [49]. In a COFF file image, all information and data about the executable is organized in static layouts so that different parts of the executable can be loaded into memory at run time by the system process [30].

Although, compilers can apply methods in addressing modes or reference pointers, the end resulted executable file is a fixed entity. To ensure the correctness of application execution, compilers do lots of syntax and type checking. Interfaces used are strictly verified at compilation time. All these are for easing development work. But, to make any changes to compilers, linkers, or binary file formats would be a big effort involving many companies and organizations. Object Oriented Programming and Design Patterns are supposed to help greatly. The Java Virtual Machine and Java Byte Code have come close. Enterprise JavaBeans is a further step in taking advantages of the Java foundation or platform.

The compiler verifies the correct calling of a function from the calling module to the actual function in the called module. There is no magic ways of the calling module knowing the name of the called function or the number and types of its parameters. The programmer has to gather that information before writing the calling module. For example, to call a function named openfile() in a module zipfile.o, the programmer has to make sure that zipfile has this function defined, such as:

```
int openfile(char* filename) {
      …
}
```

In the calling module, at the top of the source file, the function openfile is declared as external, such as:

```
extern int openfile(char *filename);
```

Then, somewhere in the calling module, openfile() can be called as following:

```
returncode = openfile("myfile");
```

Existing components can be linked to newly written program in this stage. So that programmers can have options in choosing the components for their advantages when

building the application. But programmers do not really know what the user actually needs from the application at run time. The concept of selecting components goes hand in hand with run-time environment.

2.2.1.2 Loading time

An executable or object file in storage is loaded into memory as the result of the application invocation. Loading time is the stage in which a static program becomes an executable image in memory. With the ability of choosing the particular modules at this time, the application can change its logic or program flows. Shared libraries are great contributions at this time because they make modules available to any applications running in this local environment. But those libraries have to be loaded at certain time before the applications making requests for the functions or subroutines in those modules. Some libraries can be loaded and unloaded live at the time that other applications are running and using those modules. Examples are the libraries in UNIX and Dynamic Linking Library (DLL) in Windows.

In Java, there are two types of Java binary objects, class or interface. Interestingly, the term “type” is used to present either a class or an interface. The JVM makes types available to the running program through a process of loading, linking, and initialization. Following definitions are from the Java Virtual Machine Specification:

Loading is the process of finding the binary file representing a class or interface type with a particular name. An object of that class or interface is then instantiated in memory. Linking is the process of taking a class or interface and combining it into the runtime state of the Java virtual machine so that it can be executed. Initialization of a

class or interface consists of executing the class or interface initialization method `<clinit>` [45].

Because of the dynamic natures of loading code, it is a risky stage to attempt to make changes to application code. Conflicts and unexpected results can occur. However, this stage can be re-enforced and maintained by a run time environment facility that resolves conflicts provides specific environment information. The risks will be minimized .

2.2.1.3 Running Time

At run time, the executing code is residing in the memory, in its own memory or process space. This is the most dynamic stage of the application. It is also most flexible for changing program logic. However, it is the hardest to transfer from designing and writing the program to figuring out the states and conditions in the application at run time.

Instead of every program doing all the explorations in its run time to acquire the information and conditions, the ideas are to create layers below or around the application to do the work. Those layers such as operating system, device driver, resource management, communication infrastructure, middleware and framework play important roles in this stage of the application. As we can see, the boundaries are endless. The application executing space is not limited to the local memory anymore but it is expanded to include memory elsewhere and the networks too. Many examples of these are: The Linda Programming Environment by David Gelernter, Tspace from IBM Corp., Jini and EJB from Sun Microsystems, and CORBA from The Object Management Group.

Many of those mentioned above are on their ways to become global standards in their domains. Their premise is that these global environments exist where there are needs for

them. But, to explore those needs, we have to go back to the local environment and see what can be done. There are much un-realized potentials in the local environments.

2.2.2 Multiple layers

Software components are commercially available today. Some are called components off-the-shelf. But the processes of building and packaging those components belong to the component makers. There are no particular requirements as per the formats of components. Component makers are free to make up their own components using those application-building methods described earlier in this chapter. Suffice to say that, documents and specifications are of utmost important for them to make the sales.

We use The FlashLine.com [25] Website as an example for off-the-shelf components. The website has a section for component marketplace where it lists many categories of software for sales. The categories are grouped into two major groups of construction formats and availability:

- Java Component Marketplace and
- Net / ActiveX / COM Marketplace.

2.2.2.1 Components with no framework

Listed on the website are many small components sorted in different categories of user interfaces or utilities. See Figure 2. What we learn from these components is that each of them is developed in different ways and their usages are widely different. An integrator has to look into each component to hopefully figure out how it can be integrated into an application. Notice that the items listed under heading Technologies are actually frameworks.

Technologies
Applets
EJB™
JSP™
JavaBeans™
Servlets

Internet/WWW
Infrastructure
eCommerce

Communications
Email
Network
Printing
Wireless

Information Management
Calculators
Content Management
Data Manipulation
Database
Encryption
Internationalization
License/Copy Protection
Reporting

User Interface
Barcodes
Buttons/Tabs
Calendars/Clocks
Charts/Graphs
Command Line
Data Input/Masking
Editors
Forms/Menus
Graphics
Grids/Tables
Instrumentation
Tree Controls

IDE Extensions
Forte Modules

App Server Add-ons
JBoss

Utilities
Compression
File Processing
Image Processing
Libraries
Wrappers

Training
CORBA
Component-Based Development
Java™
Object-Oriented Programming
Test Preparation

Development Tools
Application Servers
Code Generators
Code Testing
Component Managers
Configuration Management
EJB Builders
Editors
Frameworks
Graphics Builders
IDEs
Introspection Tools
Modeling Tools
Obfuscators
Optimizers/Debuggers
Plug-ins
Version Control

Figure 2 FlashLine Java Component Marketplace

2.2.2.2 Components with framework

As pointed out in previous section, framework components include some of those listed under Technologies in Figure 2, such as EJB or Servlet. In Figure 3, more components are listed under .Net, ActiveX, COM, etc. Notice that the names of the components are very familiar with the non-framework components. Thus, integrator has options of choosing either of the components.

User Interface	**Information Management**
Barcodes	Calculators

Buttons/Tabs	Content Management
Calendars/Clocks	Data Manipulation
Charts/Graphs	Database
Data Input/Masking	Encryption
Editors	Internationalization
Forms/Menus	License/Copy Protection
Graphics	Reporting
Grids/Tables	
Instrumentation	**Internet/WWW**
Tree Controls	Infrastructure
	eCommerce

Figure 3 FlashLine .Net/ActiveX/COM Marketplace

2.2.3 Diverse packaging requirements

We've seen above that components can be made with framework or non-framework. They can be integrated into application in different times of development or run time.

From the examples on the FlashLine Website, each listed item is presented in a folder with these tabs:

- Overview
- Reviews
- Docs
- Components
- BUY
- Support

Of those characteristics of a product, an integrator would be interested in "Docs" and "Components". Sampling of the "Components" folder tab of many listed items revealed this message:

> "There are currently no components listed for this product."

That is a very strong statement indicating that these items are for sales as products, which are probably better off reused as a whole product rather than components. It could be true that many of those products are in fact components. There are no requirements that those components have to be packaged in any certain ways.

2.3 Component Frameworks

Facing with diverse components coming from many different component makers, integrators need to go lower than the component presentations and digging into the infrastructure and middleware layers to have better understanding of the components. We study some of the current frameworks in this section. As in the case of FlashLine above, each component does belong to at least one framework. Some of them can be built for different frameworks. These components, as most of the current components, are heavily depending on a specific framework.

Integrators need to work with more than a couple of components for their deployments. Limiting to one framework should not be a problem because there should be plenty of components to work with within that framework. Further more, frameworks are usually built with the goals of unifying different computing elements under it. Therefore, integrators can rely on frameworks for the underline component infrastructures. An analogy of the application using components on top of component frameworks is like the application using those operating system services on top of hardware platforms. Applications are no more required to execute those hardware specific operations. Operating systems provide an abstract layer and general implementation of the hardware devices.

However, in software component, integrators should be able to make use of a particular component. Besides adhering to certain framework implementations, additional higher level of documents and specifications are required from component makers. Therefore, from the application integration point of view, components are at the middle layer between the application and the component framework. At the lower layer

of component frameworks, organizations and specifications are highly well defined. But, that is low-level integration tasks. Application integrators need to put together the middle layer component integration.

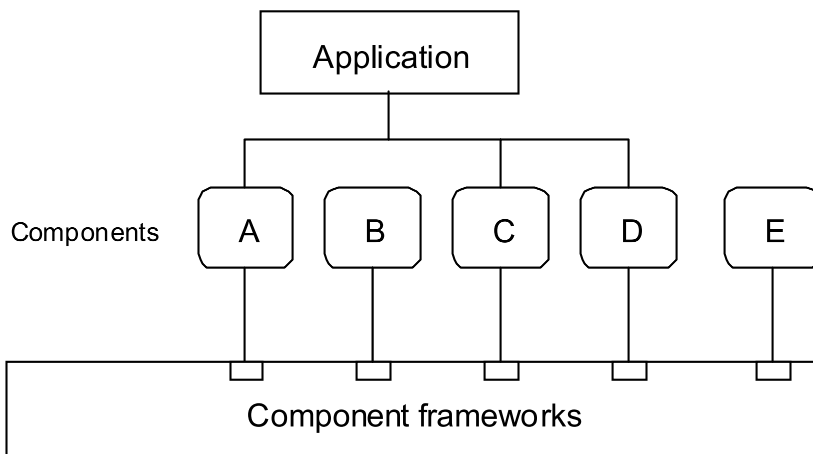

Figure 4 Three-Tier-View: Application, Components, Component Frameworks

The three layers in our views:

- Application layer: contains application logic or workflows that drive the operations of an application. This is also the integration layer where other smaller components are utilized.
- Component layer: contains code units that can be reused. Many components to choose from at this layer. There seems to be two choices, small and disarray of components or components that depend on the Component framework layer.
- Component framework layer: contains virtual environments that aim at globalizing components and applications within certain domains.

Component frameworks make building and reusing components possible in many different ways. However, to be able to understand them and use them correctly present

hurdles to be overcome, not to mention the difficulty in choosing one over another [69]. Nevertheless, following technologies help shape up the perspectives and present the advantages and disadvantages for this thesis model.

2.3.1 DLL, COM+, and .NET

Dynamic Linking Library and Component Object Model Plus are the essential software component model and integration mechanism in Windows Operating System. Using development tools such as Visual Studio, programmers can write applications that can link to object library modules at run time. Hence, components can be reused between these applications [20].

The coming framework, .NET, also promises that these software components can be reused independent of programming languages [51]. A bridge is provided as COM Interoperability, which enables the communications between .NET and COM by providing:

- Common Language Runtime core services
- A set of Application Programming Interfaces
- Software Development Kit

Combined with the Windows Registry, DLL, COM+, and .NET provide most necessary environments for component development and deployment in Windows. They leave all the application design and implementation work to the programmer. There may be more than one ways to integrate these components together.

2.3.2 CORBA 3

Common Object Request Broker Architecture Revision 2.6.1 [53] is the basic model for object oriented and client server programming standards that can be used with any programming languages, operating systems, or networks. The standards provide the following:

- Object Request Broker
- Object Management Architecture: CORBAfacilities and CORBAservices
- Interface Definition Language

CORBA provides the most comprehensive object oriented environments in which objects can be interoperable [60]. With IDL, any programming languages can be mapped into the CORBA object model [56]. In addition to the program itself, the programmer follows strictly the syntax of IDL to specify his interfaces. Following is a short sample of an IDL file for a Java module:

```
module Counter
{     interface Count
      {     attribute long sum;
            long increment();
      };
};
```

Because the mapping of the syntax from a programming language to IDL is very strict, development tools can automatically do the task.

Object Oriented modeling is one way of creating components that can be independently deployed. Furthermore, the ORB is a layer that protects the objects from different elements of operating systems or networks. Notice that, CORBA is based on client server model. The IDL compiler compiles an application into two parts: a stub resides on the client and a skeleton resides on the server. Application code can be

distributed on servers from which client can access at any locations in a network through the ORB.

The CORBA Component Model (CCM) Specifications [55] are extensions of CORBA object model to provide a higher level of abstraction for CORBA services. One example of the extension is the Component Interface Definition Language (CIDL) as to IDL. Other noticeable changes are in the Interface Repository (IR), which provides dynamic access to the ORB. Requests to the IR may include: type checking of request signatures, checking of interface inheritance graphs, etc. Those requests come from any location in any networks.

2.3.3 Enterprise JavaBeans

Enterprise JavaBeans ™ (EJB) [18] is a framework or middleware that bases on the Java programming language. The Java compiler generates class files in Java Byte Code format. Java class files can be loaded and interpreted by the Java Virtual Machine (JVM). Thus, any system or platform that is equipped with JVM can run Java applications. Java is implemented in most of the operating systems and it has the supports of many enterprise technologies such as: JDBC, JNDI, Servlets, JavaServer Pages, Java RMI, CORBA, XML, etc.

However, the basic components that EJB provides are as following:

- Enterprise Bean Interfaces: Home Interface, Remote Interface, Bean Class, and Deployment Descriptor
- Container tools and services

In addition, EJB suggests different roles in providing and deploying components:

- Bean developer

- Application assembler
- Deploying programmer or user
- System administrator
- EJB server provider
- EJB container provider

Those are EJB specific roles. When applying to other component models, the roles may vary to suit the model. Notice that the last three roles devote to managing the server and the container.

Since EJB is a server side component model, the bean container is an important concept [45][47]. The container provides a uniform interface between the bean and the server. A bean is implemented using one of the two interfaces: EntityBean or SessionBean. At run time, a bean executes in a container, which creates a kind of virtual environment to provide supports and services to the bean.

At development time, the bean has to be written based on two interfaces, remote or home, because it is a client server distributed model. A program that writes into the container interfaces always has the same services and supports. Because the container is established in multiple network spaces, a Java Naming and Directory Interface (JNDI) is requires for locating the objects and other resources.

Enterprise JavaBeans container provides mandatory facilities for its components. When a programmer writes his program, he depends on these facilities as running environments. They are examples of what applications require at run time. There are many other types or forms of environments that applications depend upon. Any application, however, starts with a core of requirements without which the application

will fail. How large or how small this core of environments should an application includes is a hard decision for the programmer to make at the time of development. It is a complex relationship between programming complexity and application reliability.

Nevertheless, when the programming work is done and the application is transferred into the user's hand, those environments are reality factors at run time and the user is greatly impacted as the result of application success or failure. There is a gap between how the program is intentionally set up from its development environments to the reality of run-time environments. This model aims at closing that gap.

With a local environment facility in place, the program will do less of conditional checking about environment specifics, similar to the ideas of the bean getting all the services and supports from the container. Further more, by limiting the boundary of application execution to local environments, many complicate aspects of distributing environments can be set aside, to name two important ones: naming and directory or communication infrastructures.

2.4 Environment as Container

2.4.1 Java Virtual Machine

To run an application written in Java, the top level, containing the "main" module, class file is passed to the Java Virtual Machine (JVM). For example, the following command runs the application `ZipAF`:

```
java ZipAF
```

At the request of the command, an instance of the JVM is created to run `ZipAF` in memory until `ZipAF` exits. If `ZipAF` has multiple components, by definition of software component, JVM does not have to load all the components at once initially. Further

more, JVM provides mechanisms for manipulating code at run time. It also provides utilities such as the System class for applications to access lower layer services such as the ones provided by the operating system. Java applications cannot run without JVM or the operating system.

The application runs inside JVM like a container. Typical Java application is written with Java language and run in JVM. Many components are written with Java but not all of them reside in the particular JVM until they are loaded. Components are scattering in local machines. It is a big job and a great advantage for the application integrator to be able to make use of those components. The Java language or its JVM provides the tools but does not address the component problems. Take the capability of loading a class file or a module dynamically for example, Java or JVM do not define where the modules should reside. Application integrator can add much value for the work in these areas.

2.4.1.1 Binary file verification

The JVM is notorious on verifying the byte code before loading it. The Class File Verifier goes through a four-pass process to ensure that loaded class files have a proper structure and that they are consistent [83], granted that there are more security concerns in that process. From the application integrator's points of views, what are the risks involved are as following:

- The called method might not be found for many reasons: slight name changes, typos, linking to incorrect module, etc.
- The called method might have different numbers of parameters
- The number of parameters might be correct but some parameter might have the wrong data type

The list of those run-time errors can go on, while they could have been corrected if the code has been compiled and linked together. The integrator is put into an after-the-fact situation. . If we study Java's loading procedures well, we may be able to do dynamic component loading at run time as we've planned.

2.4.1.2 Java class loaders

The Java class loader sub-system is responsible for locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and resolve symbolic references [45].

JVM makers implement the bootstrap class loader and the system class loader. The bootstrap class loader is part of the JVM and loads trusted classes that include the Java API. The system class loader is a user-defined loader, which is part of the Java application and starts when JVM starts.

Java programmers have the options of writing their own user-defined class loaders. Examples of this class loader are the applet loader in a web browser or the JDBC class loader. Each of the class loaders has different ways of loading and running applets in its own namespace. Each loader loads Java byte code into a separate namespace at run time. Because we are interested in dynamic loading of components, this namespace is important. It is part of the IEC design and implementation. See Section 5.3.1.

Writing a user-defined class loader would be a potential solution for our tentative model of loading locally available components. This class loader can have

- different ways of searching for the components being loaded
- certain rules to verify components at run time

Once the new class loader class is available, it can be started from a Java program using the Java API ClassLoader class' loadClass methods. For example:

```
protected Class loadClass(String name)
      throws ClassNotFoundException;
protected Class loadClass(String name, Boolean resolve)
      throws ClassNotFoundException;
```

The method `loadClass` returns a reference of the class it just loads. With the class reference, the calling program can initialize the class object and call its methods for instance. It is a way of dynamically extending the program at run time.

Nevertheless, when we consider other programming languages, adding a new class loader is like changing the program loader or linker, which are a couple of layers lower than the component layer. Our objective is to have the solutions at the component layer.

2.4.2 J2EE containers

Following similar analogy as for JVM with operating system, J2EE containers are created to run with J2EE server. J2EE runs in JVM. J2EE supports many different types of containers: Enterprise JavaBeans (EJB) container, Web container, Application client container, Applet container [12]. Container has become a means for standardizing run-time environments. All of those containers exist in networked environments. In the views of multi-tier J2EE application development, each of those containers becomes a tier of application implementation such as the Client Tier, the Web Tier, the Enterprise JavaBeans Tier, etc [71].

Taking advantages of those containers, application integrators can specialize in each of the tier or application domain. Containers provide their methods of component management. Hence, it becomes a type of virtual or global environment trying to cover as many different varieties of environments as they can. Such environment is big and

complex to start with [67][68]. Besides being restricted to the server side, some of the restrictions in development and deployment make these components rigid. For example,

- To write a web component, the component maker has to follow the Servlet protocol [15][35]
- To write a component accessing the enterprise servers, the component maker has to write a bean as in JavaBeans requirements [4][51]

2.5 Component Packaging

We've seen many types of components. The non-framework components are made and delivered in many different ways that may or may not resemble each other. General installation tools are available for setting up application installation. In Windows platform, there are InstallShield and InstallAnyWhere. In Linux platform, there is Red Hat Package Manger (RPM) that serve similar purposes. Component frameworks usually have their own deployment methods. We study the EJB deployment descriptor as an example of how components are packaged in development for deployment at run time.

2.5.1 Installation Application

From the development points of view, the installation program is a very important vehicle making sure that the application is set up and run smoothly after. Two good examples in the Windows operating platform are InstallShield and InstallAnyWhere. They install applications, not components. However, Windows applications most often come with DLL components, which are discussed in Section 2.3.1.

Two ways a developer can use InstallShield to setup and install her application: Standard project or Basic MSI (Windows Installer format) project. The Standard project

works with Windows setup.exe, which uses initialization file setup.ini. The Basic MSI does not use setup.exe and setup.ini. It uses a file with the .msi extension.

Following is an exert from [6] to show the contents of the setup.ini file:

```
[Info]
Name=INTL
Version=1.00.000
DiskSpace=8000

[Startup]
CmdLine=
SuppressWrongOS=Y
…
[SupportOS]
Win95=1
Win98=1
WinME=1
WinNT4=1
Win2K=1
…
```

We can see that the file setup.ini is divided into sections such as `Info`, `Startup`, `SupportOS`, etc. It essentially provides all the information required by setup.exe.

In Basic MSI project, the Windows Installer is used. Hence, its MSI file format is required. This file, with the extension .msi, has four primary components [36]:

- Summary Information
- Database of installation instructions
- Application files or references to them
- Transforms

An .msi file is a package file in COM structured storage file format.

2.5.2 EJB Deployment Descriptor

Before we learn about deployment descriptor, we should understand Java archive (jar) file and its manifest file. The jar file is a way of compressing many different files into

one file. A jar can also be a bundle of application that can be run directly from the jar file without going through the extracting process. A manifest file basically has two parts:

- The headers
- The list of files included in the jar fie

Following is a sample manifest file that is automatically generated.

```
Manifest-Version: 1.0

Name: java/math/BigDecimal.class
SHA1-Digest: TD1GZt8G11dXY2p4olSZPc5Rj64=
MD5-Digest: z6z8xPj2AW/Q9AkRSPF0cg==

Name: java/math/BigInteger.class
SHA1-Digest: oBmrvIkBnSxdNZzPh5iLyF0S+bE=
MD5-Digest: wFymhDKjNreNZ4AzDWWg1Q==
```

As seen in above manifest file, the header contains the "Manifest-Version". The header may also contains package information, such as:

```
Name: java/util/
Specification-Title: "Java Utility Classes"
Specification-Version: "1.2"
Specification-Vendor: "Sun Microsystems, Inc.".
Implementation-Title: "java.util"
Implementation-Version: "build57"
Implementation-Vendor: "Sun Microsystems, Inc."
```

After the header is the list of Java classes. More information is in [77].

Enterprise JavaBeans class objects and other application files are packaged in a jar file. Accompanying with the bean classes, remote interfaces, home interfaces, and primary keys, etc. is a deployment descriptor. The container reads a deployment descriptor to learn about the beans and how they should be managed at run time. Following is an example of an EJB deployment descriptor file in XML format [51].

```
<ejb-jar>
      <enterprise-beans>
            <entity>
                  <ejb-name>CabinEJB</ejb-name>
                  <home>com.titan.cabin.CabinHome</home>
                  <remote>com.titan.cabin.Cabin</remote>
                  <ejb-class>com.titan.cabin.CabinBean</ejb-class>
```

```
                  <prim-key-class>com.titan.cabin.CabinPK</prim-key-
class>
                  <persistence-type>Container</persistence-type>
                  <reentrant>False</reentrant>
            </entity>
      </enterprise-beans>
</ejb-jar>
```

The entity and its attributes in the above example are just a small examples of many other entities and attributes. Since the deployment descriptor is in XML document format, it can be generated by IDE tools and read by container tools in automatic manners so that programmer and deploying users do not have to involve. Also, the tools can provide graphical user interface for developers to generate the descriptor and users to manage those application properties to some extent at run time.

Deployment descriptor is also how web applications being deployed as web services. SOAP is an example of web services. Web applications are Java applications running on the server side. The Java applications owner needs to specify the service information in the deployment descriptor then deploy the application so that the clients know how to access the services [17]. Although deploying application services does not sound like packaging, the idea is to expose information from one application to other applications. The interested information includes the name of the Java service classes and the methods they support, etc. If we are to let applications share components, we'll have to do something similar.

2.5.3 Package Manager (RPM)

Another package management software used in the Linux operating system is the Red Hat Package Manager (RPM). This application is known for building software for various platforms of Linux from a single set of source-code files. It is also used to install, upgrade, verify, and build software archives known as .rpm files. Compared to the above

installation applications and deployment descriptor, RPM does many more tasks for maintaining software packages such as making and compiling the application binary files from source files.

To build a RPM package, a spec file is required as input. Following is a sample spec file extracted from a How to paper on the Red Hat Website [9].

```
Summary: A program that ejects removable media using software control.
Name: eject
Version: 2.0.2
Release: 3
Copyright: GPL
Group: System Environment/Base
Source: http://metalab.unc.edu/pub/Linux/utils/disk-management/eject-
2.0.2.tar.gz
Patch: eject-2.0.2-buildroot.patch
BuildRoot: /var/tmp/%{name}-buildroot

%description
The eject program allows the user to eject removable media
(typically CD-ROMs, floppy disks or Iomega Jaz or Zip disks)
using software control. Eject can also control some multi-
disk CD changers and even some devices' auto-eject features.

Install eject if you'd like to eject removable media using
software control.

%prep
%setup -q
%patch -p1 -b .buildroot

%build
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS"

%install
rm -rf $RPM_BUILD_ROOT
mkdir -p $RPM_BUILD_ROOT/usr/bin
mkdir -p $RPM_BUILD_ROOT/usr/man/man1

install -s -m 755 eject $RPM_BUILD_ROOT/usr/bin/eject
install -m 644 eject.1 $RPM_BUILD_ROOT/usr/man/man1/eject.1

%clean
rm -rf $RPM_BUILD_ROOT

%files
%defattr(-,root,root)
%doc README TODO COPYING ChangeLog

/usr/bin/eject
/usr/man/man1/eject.1

%changelog
* Sun Mar 21 1999 Cristian Gafton <gafton@redhat.com>
- auto rebuild in the new build environment (release 3)

* Wed Feb 24 1999 Preston Brown <pbrown@redhat.com>
```

```
- Injected new description and group.
[ Some changelog entries trimmed for brevity.  -Editor. ]
```

The RPM documents contain more information about the spec file. It essentially includes two parts:

- Application specifications such name, release, version, source location, etc.
- Sequence of commands to be executed to build the package

We can see all the crucial information on building the RPM package is included in this file, which can be edited by a programmer. Or, it can be generated by another application or script. RPM generates a package file that is an archive of all the necessary files for the package.

The advantages of RPM include:

- Ability of making and distributing package to system on different platforms
- User can extract or query information about installed application files or packages
- Giving the package owner total controls of installing and uninstalling packages

2.6 Summary

From the integrator points of view, a component has to be considered individually to see if it fits deployment purposes, based on documentations and specifications. From the technical aspects, many component technologies are available. We study some of them, such as DLL, COM+, CORBA, and EJB. It is important to notice that, all these evaluations and investments of times and resources continue to be done by human, until those business-to-business and business-to-consumer services are widely inter-operable. The application integrator plays an important role. She assembles application at the highest level of software development. To improve the assembling processes,

components are available for deployments. However, components rely on component frameworks for providing the underline infrastructures. Thus, components belong to the middle tier that is below the application and above the component framework. The middle tier usage principles require that components should hide the internals of their component framework and present the component characteristics to the application layer. The current state of components is not at that point yet because it is difficult to reuse components without the requirement documentations and specifications at the component layer. At the lower layer, component frameworks compose of global environments and standards that are overly complex to begin with.

Chapter 3
Overview of Dynamic Application Integration

When a particular application is designed or even just formed in the developer's mind, it is usually not associated with any particular environments. In general, an application is a tool that mimics or reflexes a tool in real life. A clock application is obviously a counter part of the real clock that we use in every day life. We use clocks to tell time. The inventor who invented the original clock did not have any ideas that the world is divided into different time zones. The fact that a clock in New York tells a different time than one in London is not part of the clock's functionality. And, the inventor was right! One could guess if the clock works correctly in one time zone then it will work similarly in another time zone. But one needs to bring the clock to the particular time zone to prove it. The clock is a classic case of an application that separates its functionality from the environments it is in. It is obvious that such small invention has lasted and is going to last beyond time and space.

In this chapter, an example application that does file compression and decompression is used for simple understanding of the model. We also want to demonstrate the simplicity in programming using this model to solve such application problems as:

- Choosing the file format that the user prefers
- Including just enough code that the user needs in the application
- Adding a new file format.

An application can be integrated from software components. Components require mechanism to exchange their interfaces and other requirements in order to "utilize" each

other. This approach bases on the premise that programming flows and operating conditions can be resolved at run time by choosing the appropriate components with satisfied interfaces. Selecting components to integrate into the application at run time is the central idea of this approach.

A program usually can select different components by going through complicated logical conditions. Each program has its own way and unique methods of checking and testing for its requirements. The more different components and environments the program requires the more code and complex logic have to be added. Other side effects include hard coding or overlooking important conditions that can result in application malfunctions or unexpected operations. There is still no guarantee that the whole programming logic of an application works correctly in any predictable condition. Therefore, software vendors have spent much effort and resources in beta testing before product releases. Further more, there are still unpredictable conditions that cost vendors huge expenses in product supports and services.

This approach provides a method for mapping logical components into specific components. The similar components or family of components are required to be written following an agreeable interface. Then, the application or other components will base on these interfaces to tie them together, with the help of a local environment facility.

For demonstration purposes and simplicity in illustrating essential concepts, a fictitious application called ZipAF (Zip Any Format) is used. It is claimed that ZipAF can compress or decompress any compressed file format assuming that the inventor or owner of the file format would provide a component that reads and writes the file format and also does the algorithms of compressing and decompressing a file. ZipAF is

apparently a state-of-the-art component software application. We will see ZipAF's approach in achieving its claims and the details of its design and integration.

3.1 Overall Architecture and Organizations

This approach bases on the run-time interface mapping methodology and packaging of application and components for delivery in local environment. A local environment registry component provides enough common facility for component reuse and sharing.

3.1.1 Run-time Interface Mapping (RIM)

Components used in this model has three characteristics:

- They are in binary format
- They have well defined external interface
- Internally, they use unique interfaces to refer to other components

The first characteristic means that no code modification is allowed on these components. The second characteristic imposes a restriction on what type of components can be used with this model. Only components that have known interface will be used. The third characteristic implies that the use of interfaces is recursively possible and restricted.

A run-time interface-mapping component has these characteristics:

- Each interface has a unique name to ensure that there are no name conflicts when many components are obtained from different component makers. Hence, a request for using the interface can be made by name.
- Each component has to implement at least all the methods in the interface that is written precisely in component specifications. Method names and signatures are well defined in the interface.

- An interface is an abstraction that represents a group of code units that have similar functionality but possibly different implementation. An interface does not necessarily exist in the form of code. It could be a form of text format or a structure of directory tree.

LER takes care of the procedures of recognizing an interface input. Because LER API is well defined, different implements of LER are feasible.

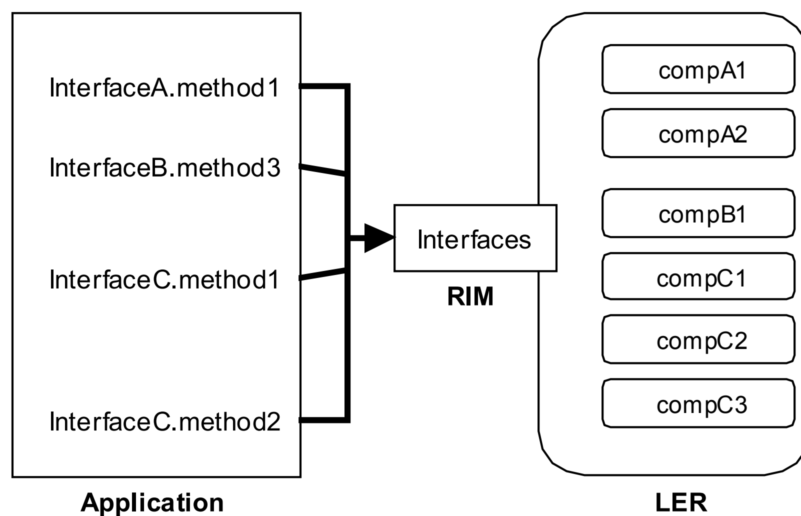

Figure 5 Run-time Interface-Mapping Components

3.1.2 Local Environment Registry (LER)

LER is persistent storage that resides in the local system once the user installs or runs an application. A LER should provide these functionalities:

- Gathering enough information about the current local environments. Information is maintained and tailored to user needs and references. Applications retrieve the required information to help resolve environment specific conditions. The information can also be passed to the component referencing process.

- Maintaining references of interfaces and their family of components that are locally available. An application depends on the LER to return the name or identifier of the appropriate components. This part of the LER acts like a local component search engine that chooses components based on user preference or specific local environment.
- Providing access and search methods for applications and tools to use the facilities in the LER. Components that are built after this model can take advantages of the LER facilities that make them more reusable by applications. Tools can use the LER facilities to update and expand LER, making up a dynamic environment to support applications at run time.

LER is a common access point that can be used with different applications or environments. A simple LER has following organizations:

User configuration table: contains user specific information such as:

Attribute	Value
LER Identification	Name
Package Locations	Search paths
Component Locations	Search paths
User Preference 1	
User Preference 2	
…	

Table 1 LER User Configuration Table

Interface reference table: contains a list of available interfaces and the mapping of each of these interfaces to specific component. The mapping is based on local environment conditions or user preferences.

Interface	Environments			
	attribute 1	attribute 2	attribute 3	
1st Interface	Comp 1	Comp 2		
2nd Interface	Comp 3	Comp 4		
…				

Table 2 LER Interface Reference Table

LER application programming interface: Users of this API are

- Component makers who build their components following this model
- Package integrators who write their programs with this model
- Component tools makers who access and maintain components with this model

Following is a summary table of the API. Details are described in a separate section.

Method	Return	Brief Description
home()	Home dir	Locate the home dir. where LER files locate
home(String Directory)	Succeeded or Failed	Set LER Home to the input directory
root()	Root dir	Locate the root dir. where the paths of component dir. start
root(String Directory)	Succeeded or Failed	Set LER Root to the input directory
registerInterface(Interface)	Succeeded or Failed	Store a new interface info into LER and create an empty component list
registerInterface(Interface, Comp, Attribute, Value)	Succeeded or Failed	Store a new interface with one component associated with the attribute and value
addComponent(Interface, Comp, Attribute, Value)	Succeeded or Failed	Add the component into the interface with associated attribute and value
removeComponent(Interface, Comp, Attribute, Value)	Succeeded or Failed	Remove the component from the interface with associated attribute and value
loadComponent(Interface)	Component reference	Load a component for the interface with default attribute
loadComponent(Interface, UniqueID)	Component reference	Load a component for the interface with unique identification

Method	Return	Brief Description
loadComponent(Interface, Attribute)	Component reference	Load a component for the interface with required attribute and default value
loadComponent(Interface, Attribute, Value)	Component reference	Load a component for the interface with required attribute and value
SetDefaultAttribute(Attribute, Interface)	Succeeded or Failed	Set a default attribute to associate with an interface
setLocalAttribute(Attribute, Value)	Succeeded or Failed	Store the "attribute, value" pair in a table
getLocalAttribute(Attribute)	Value	Find the value for the attribute

Table 3 Summary of LER API

3.1.3 Packaging components with application

A package is a means of distributing application and its components to a local machine. The integrator picks and choose the components to be included in her package. Since there are many ways of representing an interface, here we use a directory tree structure to represent interfaces. As introduced in Chapter 3, we use the example of a file zip and unzip application call `ZipAF`.

The application `zipAF` can zip and unzip three types of compressed file formats: zip, tar, and gz. Following is a sample package for the ZipAF application:

```
c:\CZip Package
c:\CZip Package\LER
c:\CZip Package\LER Installation Tools
c:\CZip Package\ZipAF
c:\CZip Package\ZipAF\ZipFile.zip
c:\CZip Package\ZipAF\ZipFile.tar
c:\Czip Package\ZipAF\ZipFile.gz
```

The package root directory is called `Czip` Package that contains LER and its installation tools. This package has one interface called `ZipFile`, which includes three code components: `ZipFile.zip`, `ZipFile.tar`, and `ZipFile.gz`.

The integrator bought these three code components from three different component makers. Each of them, however, implements a strict interface called `ZipFile` as following:

```
interface ZipAF {
      file_handle open_file(file_name)
      void decompress(file_handle)
      void compress(file_handle)
      void close_file(file_handle)
}
```

The above is just an example of a text file that presents the interface `ZipAF`. Each of the code component: `ZipFile.zip`, `ZipFile.tar`, and `ZipFile.gz` implements all four of those methods: `open_file`, `decompress`, `compress`, and `close_file` in their own way.

3.2 Application Integration Model

3.2.1 Programming skills and roles

This model clearly divides the roles of programmers who are involved with the entire process of making components and building an application into three groups of different skill levels.

Component Maker (CM): writes components according to their interface or specifications. He is trained in programming languages and practical aspects of the component. His goal is to make as many components as possible.

Package Integrator (PI): writes the LER and the program flows that integrate components into an application. She depends on available components and their interfaces or specifications.

Application User (AU): receives an application package from the PI and starts running it on specific system. He wants to use the application as what it is intended for. He also has good understanding of the system or local environments.

3.2.2 Supportive roles of LER

When the user receives the package he first runs the LER Installation Tools, which will search to see if a LER has already existed in this machine. If LER does not exist, then a copy from the package will be installed. The user may have to input the location of LER or let it be default. For the ZipAF application, the two User Configuration Table becomes:

Attribute	Value
LER Identification	BetaLER
Package Locations	c:\Czip Package
Component Locations	c:\Czip Package
Compress File Format	zip

Table 4 LER User Configuration Table for ZipAF

The LER Identification can be arbitrary string. Package Locations and Component Locations were copied from the package. User Preference 1 is set to "zip" as default. The same tools should allow user to change these values anytime later. Assuming there is no other components on this machine, the Interface Reference Table is filled in as following.

Interface	Environments			
	zip	tar	gz	
ZipFile	ZipFile.zip	ZipFile.tar	ZipFile.gz	

Table 5 LER Interface Reference Table for ZipAF

3.2.3 Writing a simple application: ZipAF

In this section, a program or module of code is written as any normal program. Checking for local environments or user preferences can be done, however, most of the time, it should let the environment or user preference dictate the flows or execution of the application.

Although the programmer may not include checking for specific local environment, at run time, the application depends on LER to provide or refer to the required option. In the above Interface Reference Table, the default compress file format is "zip".

When a component from the package is needed, the application calls the interface, which in this case is `ZipFile`. The simple `Czip` program has two simple methods: one for compressing and one for decompressing. They look like following, in pseudo code:

```
Object zcomp = LER.loadComponent("ZipAF")
file_handle fh = zcomp.open(infile)
file_handle out = zcomp.open(outfile)

If (ACTION == compress)
      zcomp.compress(fh, out)
else
      zcomp.decompress(fh, out)

zcomp.close(fh)
zcomp.close(out)
```

Before either one of the compress or decompress methods are called, the `zcomp` object is instantiated from the LER. Because the LER takes the responsibility of initiating the object, the application does not know which component the object is coming from. The integrator only knows how to use the object from the interface or specification.

If the application does not intend to use the default compressing format, then it will have to make the call to LER to set the local attribute, for example:

```
Integer rcode = LER.SetLocalAttribute("ZipFormat", "tar")
```

LER checks to see if "ZipFormat" is an existing attribute in its Interface Reference Table. If ZipFormat is an attribute, LER also checks to see if "tar" associates with an interface. If any of those checks failed, the application receives a "failed" value in `rcode`. Otherwise, `rcode` contains the "succeeded" value and the application can go on to the next step as described earlier.

The attribute name "`ZipFormat`" is a string, which can be passed in by a reference variable. Similar case applied to the attribute value, "tar". The `SetLocalAttribute` method call can become:

```
Integer rcode = LER.SetLocalAttribute(attr_var_name, attr_value)
```

The application now can input any new attribute name and value at run time.

The application is started from the main module as usual. However, the LER plays an important role in choosing the components at run time. Changes in interfaces or components in LER will lead to changes in the execution of the application. As seen in previous section, LER makes all the attribute validations at run time. That leaves only two choices to the application: continue to run if the local environment conditions are appropriate or fail over otherwise. LER can provide extra facility for the application to proceed further if the validations fail. However, this example stops here for simplicity.

3.3 Differences and advantages

In a general file compressing application, usually the programmer has to know what file formats the application can work with, as early as in the design and analysis phase. The simplest solution is to write one application for the purpose of compressing and decompressing one and only one file format. But, that is not a good solution for a user who needs to work with many file formats.

To solve the multiple file formats problem, the programmer will attempt to program the compress and decompress tasks for his application on as many file formats as he can. Then, he provides a menu or command line options for the user to choose the appropriate file format at run time. Nevertheless, the application still provides a fixed number of file formats that may not be what the user wants in the first place.

If a new file format or a new compressing algorithm is discovered after the application is made available to the user, new code has to be added into the application. Even if the new file format is available in the form of library or other component, it may provide new interfaces that are not resemble to the existing interfaces.

Following is a list of programming questions raised at the beginning of this chapter that our model aims at solving for this file compression and decompression application. More concrete examples are described in later chapter.

- LER takes over the tasks of choosing the right file format for the application so that the application does not have to do those tasks. File format is presented as a local attribute that can default to one value according to local environment or user preferences. LER keeps track of local environment attributes and user preferences and validates them for application use.
- LER also provides the appropriate sub-component for application at run time. That is made possible via IEC. Each IEC is composed of optional sub-components, which is chosen by LER at run time. Therefore, the application does not have to include all the code for all the formats it knows.
- Adding a new compress file format that is not known at design and implementation time can be achieved without modification of the code. The new

file format is introduced as a new sub-component that has the same interface as existing sub-components. It is added into the existing interface in LER. The application will just input and set the attribute to a new value.

Existing technologies that were studied in Chapter 2 can be used for designs and implementations of such application. Each technology has its advantages when used in its domains. However, each technology also has its requirements and restrictions that do not fit the needs of all applications. In the following sections we discuss the strong characteristics and the weak points of these technologies for the sake of building this sample ZipAF application.

3.3.1 Dynamic run-time loading versus common objects

If we decided to write the application with COM+, all components of the applications can be built into the Dynamic Linking Library (DLL) format. The foremost advantage is that the ZipAF application can be run on any systems that have the Windows operating system installed. And the components can communicate easily and dynamically with other components at run time utilizing DLL.

All the components of the application can be written using the standard interfaces provided by the Windows operating system. Windows provides most of the lower level services for Input/Output devices, File System Management, or Graphical User Interface. The application integrator can purchase different file compressing or decompressing components from different component makers because they are written in the same standard interfaces.

For the ZipAF application, different component implementations may attribute to differences in file formats, compressing algorithms, performance characteristics, or

specific Input/Output devices. The different characteristics of lower level or hardware platforms are taken care of by Windows.

The foremost drawback of writing components in the Windows COM+ environment is that the components can only run in the Windows operating system because of the binary file format compatibility. If these standard interfaces on Windows are different than any other interfaces on other operating systems, the application integrator is tied to one single operating system. However, consider that Windows has a popular base of users, running in Windows includes large enough domains that the component makers can certainly live with. Also, when we think about a local environment, we take into account the fact that existence of different operating systems is inevitable.

The application integrator runs into some problems though when building the ZipAF application. First of all, the library functions provided by DLL have to be called by static names. These names have to match exactly as the names in the DLL module. That leads to the requirement that the application needs to know what type of file format or compressing algorithms it can work with, assuming those are provided by different components. Then the application integrator usually builds into the application a set of menus to let the user pick the file formats at run time.

The user menu approach provides flexible user interfaces. However, the menu has to present all the possible choices to the user, because it is desired that the ZipAF application can compress as many file formats as possible. Additional programming logic is added for this purpose. But the user often chooses a subset of the file formats, leaving other formats unused. Nevertheless, all the file formats have to be included.

When a new file format is introduced at a later time, it has to be added to the menu. And more logic is added for checking user's choice inputs. As a result, code changes have to be maintained and the application has to be rebuilt every time there are changes in the file formats in order to keep this application common to any compress file formats.

Dynamically loading the components at run time eliminates the needs for specifying fixed component names and the logic of selecting them.

3.3.2 Component packaging versus distributed modeling

CORBA is designed to solve the problems of running an application on multiple platforms, which include operating systems, communication networks, programming languages, etc. It uses distributed computing model to accomplish the daunting tasks. An application can get or call different subroutines or modules from other systems on the networks at run time. The application is a client, which depends on many other code servers and request brokers.

For this `ZipAP` example, the application can be running on a system that has no code that it requires to run. However, the application can make calls that are translated into requests to some remote server having the module. The module is executed on the remote server and then the results are sent back to where the requests originated. This scenario of requesting and executing code happens within the CORBA frameworks. The advantage is that what happen are transparent to the client application. This is just an overly simplified view of how a CORBA application works.

In order to rip the benefits of CORBA, the application has to be designed according to the CORBA frameworks. Basically, the application has to be designed with the client server model in mind. CORBA requires that the programming language used being

mapped into the Object Management Group's Interface Definition Language (IDL). The use of IDL makes CORBA programming language independent so that an application can call modules in many different languages.

However, the IDL perquisite is that the particular programming language has to support the mapping to IDL. Ultimately, the compiler or the integrated development environment may do the IDL mapping automatically. At the current state, the programmer has to be fluent in IDL to work with a CORBA application. That is additional requirements besides the application programming language being used in writing the application. After the application is built, different pieces of the program have to be distributed to the servers and clients before the application can be run.

At run time, the application depends on the servers and the broker databases to be up and running. The tasks of setting up these servers, databases, and networks, etc. may require different set of skills. Those initial requirements are high demands that do not encourage the development of applications starting small and simple. Unless it is assumed that the frameworks have already be in place.

Another drawback from CORBA and any other models that rely on remote server code execution is the delays in sending requests and receiving responses. Also, the network connections have to be constantly up through out the time that the application is running. Applications should avoid these situations as possible if they can just run on a stand-alone system. With the increasing hardware power and decreasing product price, the views of client systems have changed from a bunch of dumb terminals to disarray of systems or devices that have their own processing power.

The mobile and dynamic natures of these systems or devices require a light weight and quick packaging and delivering strategy. Different components or new component can be delivered to the local system at anytime. Then the dynamic loading of the components will pick up these components.

Although an user on a system may need to work with different compress file format, he only does that one at a time while running the ZipAF.

3.3.3 Local environment versus remote servers and containers

The EJB framework has similar goals to CORBA – running applications on many different platforms based on the write once run anywhere principals. It is different than CORBA because it is based only on a programming language, Java. In other words, applications that are written in Java within the EJB framework can run anywhere a Java Virtual Machine is available. Java is a popular language and there is no other IDL to learn.

Although some server setups are still required, the requirements are less complicate than CORBA's. Because the server is also part of the Java Enterprise Edition Development Kit, most Java programmers can get the EJB framework up and running with reasonable efforts. In return, the programmer gets the entire solutions for enterprise application development needs such as:

- Enterprise JavaBeans component framework (EJB)
- Java DataBase Connectivity (JDBC)
- Java Naming and Directory Interface (JNDI)
- Java Remote Message Interface (RMI)
- Java Servlets and JavaServer Pages

Java and those frameworks also create many more of other new software technologies. The plentiful of available technologies give programmers better choices to choose from. However, they may cause technology anxiety. Programmers are overwhelmed and weary of the new technologies that are not mature and changing too quickly.

To write a program or a component in EJB, the programmer has to master the object-oriented programming methodology, which is the base for Java programming. Object-oriented technology presents logical approach to component development because the capability of abstraction. From the popular JVM, EJB extends into the concepts of containers, which create common environments for JavaBeans to run in. The EJB server does not have to be running on the local system. The first thing an application does is to request for connection to the server thus the container is opened up for facilitating the JavaBeans and providing other services.

Because of the similar goals of EJB compared to CORBA, there have been work in trying to bridge between these two frameworks or architectures. CORBA has the designs for cross platform infrastructures, while EJB has the new and improved interfaces and architectures.

Having a remote server to provide the container for application to execute could be advantageous for some and not suitable for other applications. Hence, application design plays important roles in deciding if the frameworks will benefit the application operations at run time. For those applications that require and able to utilize the majority of common services provided by EJB, it is worthwhile to design and implement them in EJB. These applications can also take advantages of many components available for EJB integration.

For those applications that are intended to deploy local systems where the user is running the application, there are drawbacks if the EJB server has to be installed locally. The applications in this category also do not require constant communications with other remote servers or resources. They can run autonomously at best on the local system. The ZipAF application is an example.

3.4 Summary

When it comes to software components, there are more than one choices available for component makers and application integrators. Big standardized frameworks have been taking shape after long time and huge resource investments. But the requirements for many different types of software components keep changing. We see the needs for emphasizing the dynamic aspects of application integration at run time. Application integrators should be able to put together an application from pre-built components as quickly and simply as possible without depending on learning or setting up those big frameworks or infrastructures. The methodology also provides flexibility in selecting components at run time via the supports of the local environment facility.

Chapter 4
Local Environment Registry

From an application integration point of view, application abstraction is at the top layer of all environments. Figure 4 illustrates components as a layer under application and component framework is a layer under components. There are more layers under component framework such as virtual machine and operating system. The lowest layer is at the hardware platform. See [61] for "Structure and Flow in a Layered Operating System". In Chapter 2, Section 2.3, we also discuss our view of components as the middle layer between applications and component frameworks. Local Environment Registry (LER) is in the component layer.

Although an application is designed and implemented at the highest level of software development, it runs at the lowest level, the local machine. In certain situations, the application has to check for all the right conditions to arrive at the piece of code that does something to or make use of the available resources. Because run-time environments are dynamic, an application cannot check all the possible conditions at the time of deign or implementation. LER is designed to resolve those conditions for the application.

By starting at a local environment, we lower the number of conditions just limiting to this local environment. By doing the checking in the LER, we reduce similar amount of code from any applications that run in this local environment. With the LER Application Programming Interface, we separate specific environment checking from the application. Applications or components that use the LER API will be more portable from one to

another local environment. LER is stationary. It stays in the same environment and does not change unless the user makes the changes.

LER also plays a key role in choosing the appropriate local components for applications to load at run time. Chapter 5 goes through the design of the Interface Enabled Component and its working model, which depends on LER to search for a sub-component matching the interface requested. Notice from here on that we call our Interface Enabled Component as interface. Where necessary, we use the acronym IEC for clarity.

We've studied how applications are divided into small components and integrated back together with component frameworks, in Chapter 2. The main idea of using lower layers to support upper layers applied in both development and run-time environments. Components and frameworks have had significant impacts on how applications are deployed. In these environments or containers, we study some of the mechanisms or services that they provide. But it is depending on the programmer on knowing how to deploy those services.

One way to reduce complexity is to go back to the small local environment for developing and deploying the application from there. Then, using one of those environments as a starting point for an application development.

4.1 LER Requirements and Organizations

LER is not a replacement of those existing run-time environments. It is a software component that application can use for purposes of adapting to the running environment and obtaining other localized components. Our tentative proposal is to have a LER reside in a user space and a virtual environment such as the Java Virtual Machine. However,

the JVM is not a requirement in the major organizations of LER as described in the following sections.

The fundamental requirement is that the model is designed and implemented in a local environment first. By paying attention to adapting to local environment, it is required that the model does not depend on specific local conditions. However, using default or existing environment attributes is the key. LER is the core component but it requires the application integrator to apply the approach in delivering their applications.

To take advantages of the model, the application integrator has to use the LER component and related services in writing their application workflows. Thus, LER should have a set of well-defined application programming interface that should not be language dependent. However, there is a crucial requirement that LER should be able to load a code unit by name. Chapter 5 has detailed discussions of the Interface Enabled Component, which works in conjunction with LER.

4.1.1 Persistent storage

LER resides on the hard disk of the local machine. Certain storage space is required. In most operating systems, LER exists as one or more files in the file system. The key requirement is that these files can be loaded into memory for faster accessing to the elements in LER. Depending on implementation aspects of this storage, LER can be:

- Text file: containing the LER contents or records in text strings that represent the structures in LER. A process that understands those structures writes the memory contents to the text file or reads the text file then loads the data into memory following appropriate structures. This method may not be most efficient but it is

easy to implement and human users can look or analyze the LER contents if necessary.

- Binary file: similar to text file. However, the contents are packed into binary or byte format. It is more efficient because the memory structures can be dumped directly into the binary file. Then, the binary file can be read directly into memory. That sounds straight forwards with common data structures. However, with object oriented programming, objects in memory may not be written directly out to disk or vice-versa. Serialization is required.
- Relational database: containing the LER contents in forms of data tables and records. The benefits of using databases are that all accessing methods are provided by traditional database management (DBM) program, which may also be a drawback because it is an extra requirement. However, if the database is compatible across different DBM programs and environments, LER is as closer as a common component that can be accessed from applications.

Notice that, LER itself does not contain the sub-components, which are discussed in the next section. LER contains only the references to the sub-components.

4.1.2 Component references

LER has the missions of a local search engine for locating local components that are called sub-components because of the way we map an interface into different code units. See Chapter 5 to learn about Interface Enabled Component. Well-defined mapping schemes are built into LER to decide if a requested interface can be associated with some known sub-components reside locally. It is required that:

- The name of the IEC has to be unique. Each interface has to be registered to LER in order for LER to set up the mapping schemes. The naming conflicts have to be resolved at the time of registration. Otherwise, the interface fails to be registered. LER maker sets the rules for naming the interface and its sub-components. However, the internal mapping scheme should not impact the external methods of accessing and requesting for sub-components.
- LER has a record of where each sub-component located. Each sub-component has to be added into LER for association with an interface. Similar to interface registration, naming conflicts should be resolved at the time a sub-component is added. Sub-components are unique within an interface. One sub-component can be added and referenced from more than one interfaces as long as there are no naming conflicts.
- LER can load the component into memory for applications to use as requested. This requirement may vary from different programming languages or different running environments. However, the basic requirement is that LER does the loading in place of the application or running environment. An application makes the request to LER for an interface. LER services the request by loading a sub-component into the format that the application needs. The process of choosing a local component is upon the LER, not the application.

4.1.3 Searchable local environment

Besides the ability of searching for sub-components, LER also has the ability of containing local environment attributes and searching for them when queried. The major

usage of local attributes is for associating sub-components to their interface. In other words, each interface has as many local attributes as number of sub-components.

- Local attributes can exist in a LER independent of any interfaces. User tools or applications can set or get attribute value using the LER programming interfaces. Notice the requirement for LER is to be own by a user. Thus, local attributes represent user preferences.
- Local attribute values are used as lookup keys associating interface and sub-components. See Chapter 5 for the requirements of IEC design. The keys are predefined by the interface when it is registered. However, it is possible to add a new key if a new component is added into an interface and it requires a new key.
- Because it is possible that the value of an attribute can be simultaneously set by multiple instances of running processes, access-locking scheme needs to be put in place by the LER to assure integrity and avoid deadlock on specific attribute. This may also relate to user authorization or security priority.
- Removing an attribute also involves some corporations. An attribute can only removed when there is no interface referring to it.

4.1.4 LER application interface

LER should have well defined application-programming interfaces so that applications and components can access to the LER services such as:

- Registering an interface
- Adding a sub-component to existing interface
- Selecting a sub-component from an interface
- Getting and setting local attributes and values

Section 4.3 describes the design and implementation of those interfaces in details.

4.2 LER Internals

As discussed in the organizations of LER, the internals of LER include these components:

- The registry engine
- The interface-component reference
- The local attribute matcher

Detailed designs and concepts of those components are described in this section. Where programming concepts are necessary, the Java language is used. Ideas of extensions to other programming languages are discussed later.

From external, LER is known by a set of programming interfaces providing its services.

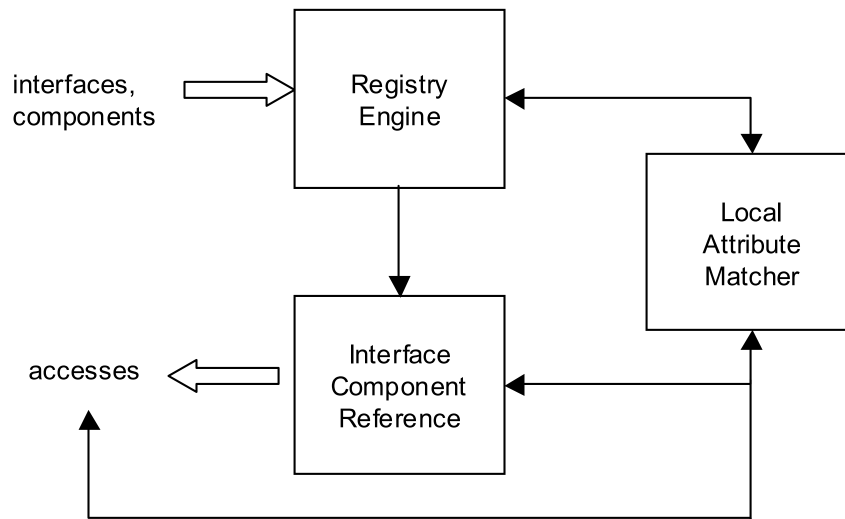

Figure 6 LER Organization

4.2.1 Registry engine

The registry engine is the only built-in component for LER. We want LER to be used in a program without the need for instantiating an object. At the core of LER, the registry has the important tasks of:

- Setting up the initial state for LER such as file location and root path
- Accepting or rejecting the enrollment of an interface from name validating and exchanging local attribute with local attribute matcher

The declaration of LER and its registry engine, `RegEngine`, maybe as following:

```
public class LER extends Object {
      static RegEngine registry = new RegEngine();
      …
      public static class RegEngine {
      …
      }
}
```

`RegEngine` is a member class of LER, which also has a data field for a registry. By declaring the registry static, it can be accessed without LER being instantiated.

The first method provided by the registry is for installing its LER. The installation process defines where files and components located in the local machine. This install method can also be called to set up the LER interface-component reference (ICR) and LER local attribute matcher (LAM). Thus, it is possible for LER to use different ICR and LAM other than the ones come with it. Each install method should be called only once in a local machine.

The second method that the registry needs is for registering an interface. To do that, a default local attribute has to be defined in LER. We will discuss about local attributes in Section 4.2.3. For now, we declare this data field in LER:

```
static LocAttribute attribute = LocAttribute.default();
```

Because the registry is defined as persistent storage, its running state in memory needs to be saved to disk from time to time. Depending on the supporting storage media and the complexity of the registry, this group of tasks can be provided by a separate component. But, for simplicity, we implement a method of the registry to do the job. This method can be extended later.

Following is summary of methods defined by the `RegEngine` class

4.2.1.1 Registry installation methods

```
public Boolean install(String lerHome; String lerRoot);
public I-CRef installIEC(String i-cRef);
public LAMatcher installLAM(String laMatcher);
```

The first install method sets the LER Home directory to lerHome. LER Home is where all the LER files stored. It also sets the LER Root directory to lerRoot. LER Root is the starting directory in the search paths for locating the interfaces and components.

The second install method initializes the interface-components reference. It searches LER Home first for the component name contained in the string i-cRef. If it cannot find the component there, it follows the search paths starting from the LER Root.

The third install method initializes the local attribute matcher. It uses the same search paths as described above. The two classes

```
I-CRef
LAMatcher
```

returned from the second and third install methods are defined in Sections 4.2.2 and 4.2.3.

4.2.1.2 Interface enrollment methods

```
public Integer register(String interface);
public Integer register(String interface, String attributeName);
```

The register method requires at least an interface input. If an attribute name is not presented, the default attribute is used to associate with this new interface. The name of this interface is validated and verified with existing names to ensure correctness and uniqueness.

The registry maintains two hash-tables for referencing the interfaces:

- Interface-to-attributes: matching of an interface to multiple attributes
- Attribute-to-component: matching of an attribute to multiple values each of which is associated with a sub-component

We describe the usage of the hash-tables in Section 4.2.2, Interface-component reference.

4.2.1.3 Registry storage methods

```
public Integer save();
```

This method is for the complete of the design but it is not part of the requirements for this model to work.

4.2.2 Interface-component reference

The interface-component reference is the main component that lets LER provide services to applications that request loading of local components. The reference is a one-to-many relationship between an interface and multiple local attributes. These are the tasks of an interface-component reference:

- Returning the handle of a loaded component based on interface or local attributes
- Adding or removing sub-components for a specific interface

The reference can be implemented with any table-lookup data structures, ranging from array, associated array to relational databases. For demonstration purposes, a HashTable

is used to implement the interface-component reference. We actually need two HashTables as illustrated in following figure.

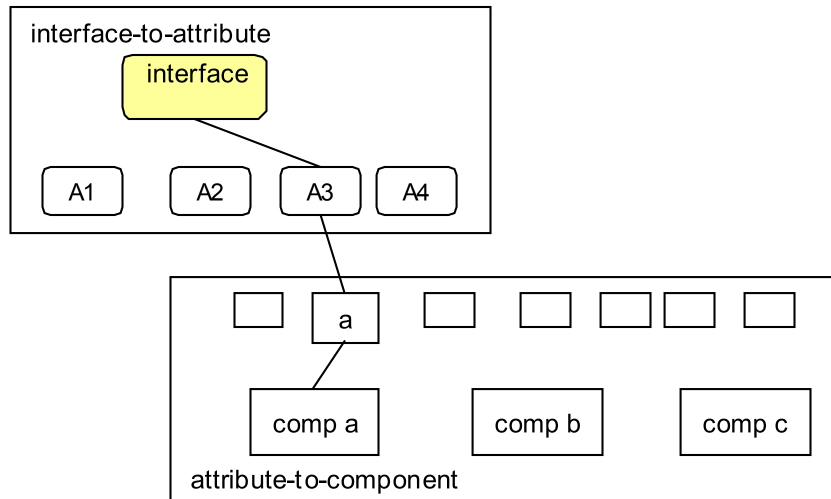

Figure 7 LER Interface-Components Reference

An interface maybe associated with one or more local attributes. Therefore, it is required to start from a given interface name as a key. If the requested attribute is found associated with the interface, a value of the attribute leads to a component.

Interface names and attributes have to be unique. As in the interface-to-attribute hashtable, see Figure 7, there maybe more than just one interface. By definition of a hashtable, the keys, name of interfaces, have to be unique. Because an interface represents a group of components, interface name has to be meaningful. However, attributes and their values do not have to be meaningful, as long as each of them is uniquely represent an attribute, a value of the attribute, a component name, etc.

The process of traversing the hashtables gives rise to the scheme of naming a sub-component of an interface. From Figure 7 for example, the component in box “comp a” may have the name like this:

```
interface-A3-a
```

where:

- interface is the name of the interface
- A3 is the name of an attribute
- a is a specific value of A3

Complex interfaces may require more than one attribute to represent a component. Each attribute also can have multiple values. A hashtable can be implemented so that a key can associate with multiple values. Hashing of a key points to the first value or default value. Subsequent values can be traversed to by a link an array index. See Figure 8.

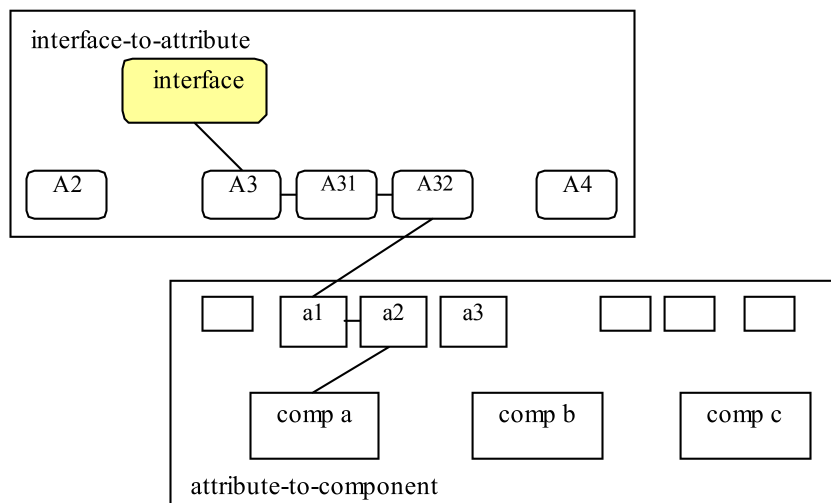

Figure 8 Interface with multiple attributes

The internal hashtables and searching algorithms should be transparent to the external requesting program. There are some public methods with various types of parameters for accessing the interface-components reference.

- Loading a local component based on interface and local attribute
- Setting up location where components can be found
- Adding a component into its associated interface
- Removing a component from its associated interface

4.2.2.1 Loading component methods

```
public Object loadComponent(String interface);
public Object loadComponent(String interface, String attrib);
public Object loadComponent(String interface, String attrib,
      String value);
```

The first method searches for a sub-component in the interface reference that matches the default attribute. The second method searches for sub-component that matches the specified attribute and its default value. The third method searches for a sub-component in the interface reference that matches the attribute and its value as specified.

4.2.2.2 Adding component methods

```
public Boolean addComponent(String interface, String component);
public Boolean addComponent(String interface, String component,
      String Source);
public Boolean addComponent(String interface, String component,
      String Destination)
public Boolean addComponent(String interface, String component,

      String Source, String Destination)
```

The first method assumes that the new component can be found at some default location and it destination location is also default. The second method requires the location of the new component but it is then put in a default location. The third method assumes that the new component is at a default location but it requires input of the destination location. The fourth method does not assume either the new component location or its destination.

4.2.2.3 Removing a component methods

```
public Boolean removeComponent(String component);
public Boolean removeComponent(String component, String

      Location);
```

The first method assumes the component is at a default location, while the second method requires the location specified.

4.2.3 Local component matcher

The local attribute matcher (LAM) is the component that provides all the LER services for utilizing or accessing local attributes. As the thesis is about local run-time environments, local attributes play important roles application initialization and adaptation. The most important tasks are:

- Categorizing local attributes into groups
- Setting and getting attribute with its associated values

We called it the matcher to differentiate it with environmental variable approaches such as the UNIX korn shell, which passes environment variables into the shell script and let the programmer decide what to do with the variables. The local attribute matcher is designed so that the program or the programmer presents a value that is required for an attribute. It is up to the matcher, not the program, to check to see if the value is satisfied. To achieve this goal, the local attribute matcher has to be designed from bottom up. In this thesis, we apply the idea to matching of local attribute values with sub-components of an interface.

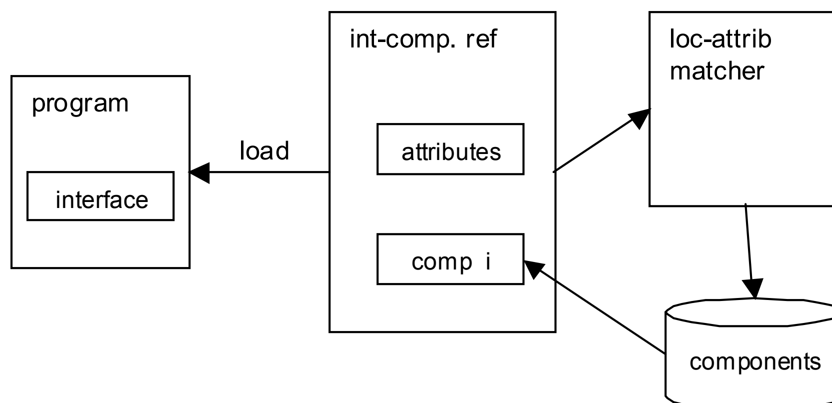

Figure 9 Local component matching

As illustrated in Figure 9 Local component matching, although a program requests for an interface, interface-components reference hashes out the candidate attribute and component. But, local attribute matcher decides which component to be loaded because local attribute matcher is closer to local environment and stationary to local. Notice that, attribute and value are implemented as String for simplicity. In reality, attribute can be a class of objects that implement the Attribute interface. Each different attribute has to implement its own hashcode. It is probably not a bad idea to implement attributes and values as Strings.

First, local attribute matcher has to provide methods for setting and getting the default attribute and default value. A premise of our model is that a local environment has often already had a default setting. Applications do not need to check for attribute values but just use the default.

Second, application can actively set values to local attributes and let the matcher validate that setting. Local attribute matcher provides these methods for application to use so that to separate the logic of checking local attributes from the application.

Methods of ordering and prioritizing the local attributes are also provided. Local attributes are grouped into priority groups. If there is doubt as which attribute should be considered first in the process of choosing an attribute, the one with higher priority is picked by default.

Note that similar attribute-matching scheme can also be applied to an application in place of the interface. While an interface is always reduced to one sub-component in the local machine, an application may be required to run with conditions of AND or OR of

all local attributes associated with it. Well-designed packaging tools can resolve this problem.

4.2.3.1 Setting and getting default attribute methods

```
public Integer setAttributeDefault(String attribute);
public Integer setAttributeDefault(String attribute, String
    value);
public String getAttributeDefault(String attribute);
```

The first method sets an attribute to default attribute to this environment. The second method also sets a default value to the default attribute. The third method returns the default attribute.

4.2.3.2 Setting and getting attribute value methods

```
public Integer setAttributeValue(String attribute, String value);
public String getAttributeValue(String attribute);
```

The `setAttributeValue` method binds the value to the attribute. If that attribute does not exist, it is added. The `getAttributeValue` method returns the default or highest priority value of the attribute.

4.2.3.3 Attribute categorizing methods

```
public Integer upAttribute(String attribute);
public Integer lowerAttribute(String attribute);
public String nextAttribute();
public Boolean hasMoreAttribute();
```

Assuming that the attributes are arranged in the hashtable in certain enumeration order, which also decides the priority of choosing an attribute, the `upAttribute` method move the attribute pass its upper attribute. The `lowerAttribute` method does the opposite.

The third and fourth methods are common in collections of data for the purpose of stepping through the attributes, one at a time.

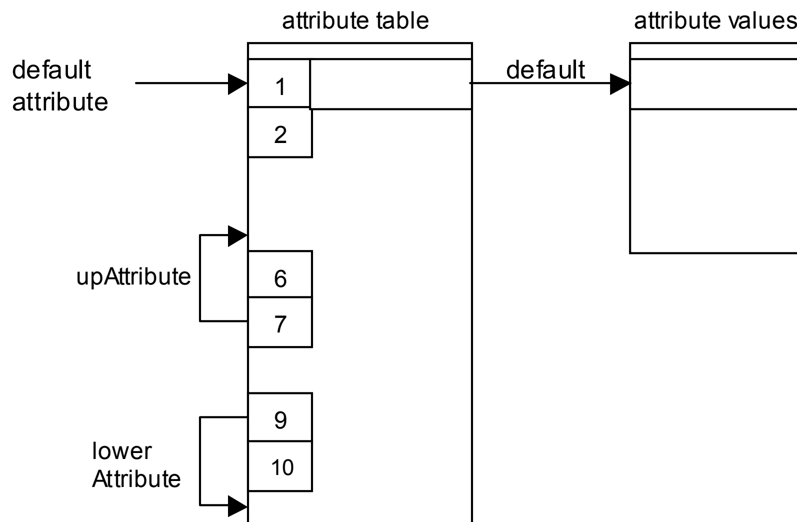

Figure 10 Attribute enumeration and priority

4.3 LER Application Programming Interface

We have shown the internals of LER organization. Although the internal components provide various interfaces, not all of those methods are required to be exposed to external calling code. Because LER is intended as a component to be used in other programs, its public API should be well defined. Listing of those methods is detailed in following sections. All of the methods are declared static because we want them to be class methods, which can be called without instantiating LER every time.

4.3.1 LER installation and maintenance methods

home

```
public static String home()
```

Locates LER HOME directory from its registry. The HOME directory is set when LER is installed. It is where all the LER files reside.

Returns:

A string containing the HOME directory of LER or NULL string if HOME is not set

See Also:
`home(Directory), root(), root(Directory)`

---

home

`public static int home(String Directory)`

Sets LER HOME directory to the directory contained in the input string.

Parameters:
`Directory` – a string containing directory path for LER HOME

Returns:
0 if succeeded or negative number if failed

See Also:
`home(), root(), root(Directory)`

---

root

`public static String root()`

Locates LER ROOT directory from its registry. The ROOT directory is set when LER is installed. It is the starting point for search paths for interfaces and sub-components.

Returns:
A string containing the ROOT directory of LER or NULL string if ROOT is not set

See Also:
`home(), home(Directory), root(Directory)`

---

root

`public static int root(String Directory)`

Sets LER ROOT directory to the directory contained in the input string.

Parameters:
`Directory` – a string containing directory path for LER ROOT

Returns:

0 if succeeded or negative number if failed

See Also:

`home(), home(Directory), root()`

---

save

`public static int **save**(String Directory)`

Saves LER current contents into the directory.

Parameters:

`Directory` – a string containing directory path. If this string is null, LER is saved into the LER HOME directory.

Returns:

0 if succeeded or negative number if failed

See Also:

`home(), home(Directory), root()`

4.3.2 Interface registering and loading methods

registerInterface

`public static int **registerInterface**(String Interface, String Component)`

Adds the new interface into LER registry after validating the name. The component is added with the default attribute and default value.

Parameters:

`Interface` – a string containing the name of the new interface
`Component` – name of the component associated with the interface

Returns:

0 if succeeded or negative error code if failed

Error codes:

-1 Interface already exists
-2 Invalid interface name
-3 No default attribute
-4 Default attribute has no default value

See Also:

`registerInterface(Interface, Component, Attribute, Value)`

---

registerInterface

```
public static int registerInterface(String Interface, String Component,
      String Attribute, String Value)
```

Adds the new interface into LER registry after validating the name. Then a component is added with association to the attribute value. The name of the component is also validated.

If the attribute does not exist, it is added to the local attribute table. If the value of the attribute does not exist, it is added. This attribute and its value is now associated with the component for the interface.

Parameters:

`Interface` – name of the new interface
`Component` – name of the new component
`Attribute` – name of the attribute
`Value` – value of the attribute

Returns:

0 if succeeded or negative error code if failed

Error codes:

-1 Interface already exists
-2 Invalid interface name
-3 Component already exists
-4 Invalid component name

See Also:

`registerInterface(Interface)`

---

loadComponent

```
public static ComponentRef loadComponent(String Interface)
```

Searches for the local sub-component that associates with this interface, using default attribute. If found, the component is loaded into memory.

Parameters:

`Interface` – the interface having desired sub-component associated to

Returns:

Reference of the component loaded if succeeded or NULL if failed

See Also:

`loadComponent(Interface, UniqueID),`
`loadComponent(Interface, Attribute),`
`loadComponent(Interface, Attribute, Value)`

---

loadComponent

```
public static ComponentRef loadComponent(String Interface, String
      UniqueID)
```

Searches for the local sub-component that associates with this interface, using specified unique ID value. If found, the component is loaded into memory.

Parameters:

`Interface` – the interface having the desired sub-component associated to
`UniqueID` – the unique identification associating the interface with its sub-component

Returns:

Reference of the component loaded if succeeded or NULL if failed

See Also:

`loadComponent(Interface)`, `loadComponent(Interface, Attribute)`, `loadComponent(Interface, Attribute, Value)`

---

loadComponent

```
public static ComponentRef loadComponent(String Interface, String
      Attribute)
```

Searches for the local sub-component that associates with this interface, using specified attribute and its default value. If found, the component is loaded into memory.

Parameters:

`Interface` – the interface having the desired sub-component associated to
`Attribute` – the attribute associating the interface with its sub-component

Returns:

Reference of the component loaded if succeeded or NULL if failed

See Also:

`loadComponent(Interface)`, `loadComponent(Interface, UniqueID)`, `loadComponent(Interface, Attribute, Value)`

---

loadComponent

```
public static ComponentRef loadComponent(String Interface, String
      Attribute, String Value)
```

Searches for the local sub-component that associates with this interface, using specified attribute and its specified value. If found, the component is loaded into memory.

Parameters:

`Interface` – the interface having the desired sub-component associated to
`Attribute` – the attribute associating the interface with its sub-component
`Value` – the value of the attribute

Returns:

Reference of the component loaded if succeeded or NULL if failed

See Also:

`loadComponent(Interface), loadComponent(Interface, UniqueID), loadComponent(Interface, Attribute)`

4.3.3 Component managing methods

setSource

```
public static int setSource(String Paths)
```

Sets the paths for LER to find new components. From here on, LER looks into these locations to copy new components into their interface space.

Parameters:

`Paths` – where components can be found

Returns:

0 if succeeded or negative error code if failed
Error codes:
-1 Paths do not exist

See Also:

`addComponent(Interface, Component, Attribute, Value)`

---

addComponent

```
public static int addComponent(String Interface, String Component,
      String Attribute, String Value)
```

Adds the new component into the list of sub-components that reference the interface with association of the attribute and its value.

If the attribute does not exist, it is added to the local attribute table. If the value of the attribute does not exist, it is added. This attribute and its value is now associated with the component for the interface.

Parameters:

`Interface` – name of the interface to be reference of the new component
`Component` – name of the new component
`Attribute` – name of the attribute
`Value` – value of the attribute

Returns:

0 if succeeded or negative error code if failed

Error codes:

-1 Interface does not exist
-3 Component already exists
-4 Invalid component name

See Also:

`removeComponent(Interface, Component, Attribute, Value)`

---

removeComponent

```
public static int removeComponent(String Interface, String Component,
    String Attribute, String Value)
```

Removes the component from the list of sub-components that reference the interface with association of the attribute and its value.

Parameters:

`Interface` – name of the interface to be reference of the new component
`Component` – name of the new component
`Attribute` – name of the attribute
`Value` – value of the attribute

Returns:

0 if succeeded or negative error code if failed

Error codes:

-1 Interface does not exist
-2 Component does not exist
-3 Attribute does not exist
-4 Value does not exist

See Also:

`addComponent(Interface, Component, Attribute, Value)`

4.3.4 Local environment management methods

getAttributes

`public static int getAttributes(String Interface)`

Gets a list of all the attributes associated with the interface.

Parameters:
`Interface` – name of the interface

Returns:
0 if succeed or negative error code if failed
1 if the interface did not exist

See Also:
`getDefaultAttribute(String Interface),`
`setDefaultAttribute(String Attribute, String Interface),`
`getLocalAttribute(String Attribute),`
`setLocalAttribute(String Attribute, String Value)`

---

getDefaultAttribute

`public static int getDefaultAttribute(String Interface)`

Gets the attributes associated at highest priority with the interface.

Parameters:
`Interface` – name of the interface

Returns:
0 if succeed or negative error code if failed
1 if the interface did not exist

See Also:
`setDefaultAttribute(String Attribute, String Interface),`
`getLocalAttribute(String Attribute),`
`setLocalAttribute(String Attribute, String Value)`

---

setDefaultAttribute

```
public static int setDefaultAttribute(String Attribute, String
      Interface)
```

Sets the attribute as a default association with the interface. If the attribute does not exist, it is added.

Parameters:

`Attribute` – name of the attribute
`Interface` – name of the interface

Returns:

0 if succeed or negative error code if failed
1 if the attribute did not exist and has been added

See Also:

`getDefaultAttribute(String Interface),`
`getLocalAttribute(String Attribute),`
`setLocalAttribute(String Attribute, String Value)`

---

setLocalAttribute

```
public static int setLocalAttribute(String Attribute, String Value)
```

Sets the attribute with the value specified. If the attribute does not exist, it is added and assigned the value.

Parameters:

`Attribute` – name of the attribute
`Value` – value to be set to the attribute

Returns:

0 if succeed or negative error code if failed
1 if the attribute did not exist and has been added with the value

See Also:

`getLocalAttribute(Attribute)`

---

getLocalAttribute

```
public static String getLocalAttribute(String Attribute)
```

Gets the value associated with the attribute. If the attribute does not exist, A NULL value is returned.

Parameters:

`Attribute` – name of the attribute

Returns:

Value of the attribute if succeed or NULL if failed

See Also:

`setLocalAttribute(Attribute, Value)`

---

upLocalAttribute

`public static int` **`upLocalAttribute`**`(String Attribute)`

Raises the priority of the attribute to higher than the one above it.

Parameters:

`Attribute` – name of the attribute

Returns:

0 if succeed or negative error code if failed
1 the attribute has already at highest priority
Error codes:
-1 Attribute does not exist

See Also:

`lowerLocalAttribute(Attribute)`

---

lowerLocalAttribute

`public static int lower`**`LocalAttribute`**`(String Attribute)`

Lowers the priority of the attribute to lower than the one under it.

Parameters:

`Attribute` – name of the attribute

Returns:

0 if succeed or negative error code if failed
1 the attribute has already at lowest priority
Error codes:
-1 Attribute does not exist

See Also:

`upLocalAttribute(Attribute)`

4.4 LER as a User Component

Instead of summarizing all the components and API's for LER, we show how a Java program can use the interface in the four API groups:

- LER installation and maintenance methods
- Interface registering and loading methods
- Component management methods
- Local environments management methods

This program is going to register two interfaces into LER. But it happens that one of the interfaces has already existed. We will see how it discovers that. It goes on and adds its own sub-component to the existing interface and makes changes to the local attributes that will select the component to load.

As described earlier, the LER ROOT directory is where all the interfaces are stored. However, the directory can be changed so that different ROOT can be used. Therefore, a program may want to know where the ROOT directory currently is. It calls the root method from LER:

```
String ROOTDIR = LER.root();
if (ROOTDIR != myROOT) {
      System.out("IMPORTANT**, changing ROOT directory");
      LER.root(myROOT);
}
```

Once the ROOT directory is set, it checks the local attribute "fileformat" which currently is not set.

```
if (!LER.getLocalAttribute("fileformat")) {
      LER.setLocalAttribute("fileformat", "zip");
      LER.registerInterface("ZipAF", "mr.comp");
      LER.setDefaultAttribute("fileformat", "ZipAF");
}
```

If LER.getLocalAttribute returns a value, the program has to see what that is and how it will proceed. Here, we assume that attribute is not set. After setting the attribute "`fileformat`", the interface "`ZipAF`" is registered. Component "`mr.comp`" is also added in the same time. Then, the attribute is associated with the interface. The three methods can be replaced with one method call:

```
LER.registerInterface("ZipAF", "mr.comp", "fileformat", "zip");
```

The second interface to be registered is "CommAD". AP stands for any device. The interface is going to be added with the component named "ms.simpleMPI". The associated attribute is "comm." and value is "ethernet". However, the CommAD interface has already existed. In this case, the component is added into the interface.

```
if (!LER.registerInterface("CommAD", "ms.simpleMPI", "comm",
"ethernet") {
      LER.addComponent("CommAD", "ms.simpleMPI", "comm",
      "ethernet");
}
```

With this component and attribute association added, it is necessary to know what attributes have already been associated with this interface. When a component is added, its attribute is the default attribute for referencing to the interface. That is desired most of the time. However, there are situations that an existing attribute is still a better choice. For example, the local machine has a wireless communication device. But, when this ethernet device component is added, it becomes the default and the wireless device become secondary. It may be an advantage to keep the wireless device as default. The lower and upper priority methods are for this purpose.

```
String attr[] = LER.getAttributes("CommAD");
String defaultAttr = LER.getDefaultAttribute("CommAD");
LER.lowerLocalAttribute("comm");
```

Since the interface has already existed, the program may need to load the component and call the interface methods to make sure they work.

```
Object commDev = LER.loadComponent(“CommAD”, “comm.”);
String response = new String();
commDev.send(“Hello”);
response = commDev.receive();
```

Finally, it is a good practice to save the stage of LER at this point.

```
LER.save();
```

This is the end of the example program that demonstrates most of the LER method calls.

Chapter 5
Run-time Interface Mapping

In Chapter 4, we discuss the ability of LER providing the different local components at run time so that the application does not have to develop much of conditional checking for possible different execution paths. We propose that, for a well-defined interface, a different sub-component can be loaded from a different local environment. In this chapter, we show our tentative solution as the Run-time Interface Mapping (RIM) methodology for component integration.

The techniques of loading a file containing binary code into memory vary from operating systems to compilers. What we study in this chapter is the ability of loading the binary code dynamically at run time. We found that the Java language can do that. We will explain how we take advantage of that in designing RIM. We are not certain if this is possible or can be done in other programming languages.

Because the Java language has the capability of loading code by name, we use Java code snippets to show implementation of the sample applications. We are also interested in the JVM as a means to understand how its Application Programming Interfaces (API) are implemented. Fundamentals of API implementation are the foundations for our proposal, Interface Enabled Component.

One advantage of restricting our solutions to a local environment is that the namespaces become smaller as compared to the efforts in covering the namespaces of multiple machines and networks such as in Java Naming and Directory Interface (JNDI).

The JVM also has a special design for its namespaces. To understand the Java namespaces, we look into the class loader mechanisms.

For our model, LER can be thought of as a local directory service for RIM of the components, in conjunction with the Java CLASSPATH. The collaborating between RIM and LER ensures that the application receives appropriate components from a local environment. It is application integration with localized components.

5.1 RIM Methodology and Requirements

Since we are mainly interested in loading component code at run time, the ability of symbolic referencing and dynamic code binding are mandatory. When it is possible, we discuss the ideas in two programming languages: C and Java. C is a basic programming language that has been most popular before Java came along. C's programming interface is much simpler than Java's. On the other hand, Java has gotten most of the technologies that we are looking for in designing and implementing our model and example programs.

While LER provides the API for accesses to the local registry, RIM defines how the registry can be used to the advantages of component integration. Although programmers can use LER in any ways that suit their needs, we suggest RIM to show how the methodology is used, the ways we see it. Notice that LER is the core of the RIM architecture. The discussions here are tightly related to discussions in LER.

Similar to the LER requirements, the most important IEC requirement is dynamic code loading. Without that, we would not have the component reference returned from LER to work with. Actually, LER has the first half of the requirement that the reference of a loaded component can be passed to a calling module. IEC has the second half of the

requirement that it can assume the component is dynamically loaded somewhere else, such as LER.

We are required to work with trusted components in the first step. Some of the reasons we have discussed above. When using Java language for implementation, the class loader does take care of some of the verification processes that might not be available if other languages are considered.

Previously, we learned from the JVM that each class loader loads code into its own namespace. In our model, that namespace or namespaces are actually in LER. The requirement from IEC is that the namespaces here are the same from LER. Again, the design of IEC takes advantages of LER for translating the request from IEC into the LER namespaces.

Last but not least, binary code compatibility is the number one requirement for IEC to work. Working in a local environment gives us this advantage because we know for sure that the code is going to be binary compatible within an operating system. The Java byte code and JVM give us more leverage on some different operating systems. But, binary code compatibility should be considered if this model to be extended to other different programming languages.

5.1.1 Symbolic referencing

In Chapter 2, we discussed that it is necessary to have a module calling methods in another module when the program is divided into many small pieces. Application programming interface makes that possible. Most compilers adhere to the same principles in many aspects of inspecting their input, a program. In C language for example, the calling module has to declare the function or method as “external” so that

the C compiler does not flag the function call as an error because it does not see the called code in the calling module.

In the case of dynamic linking library in UNIX operating system, the library also has to export the signature of the called methods for compiling and linking purposes. A signature of a function includes

- the name of the function
- a list of parameters, each accompanies a data type
- the function return type

A signature serves as a distinctive identifier for its function. Human programmers depend on signatures with meaningful function and parameter names in using the functions. Compilers, though, depend on the correct matching of each item in the signature. Most compilers go through at least two passes of verifying the source code to detect syntax or data type errors that maybe overlooked by programmer.

We've seen sophisticate compilers that automatically generate intermediate source code. They have no way of giving meaningful names to functions or parameters. Thus, functions and parameters are just symbols represented by reference addresses. After all, that is all functions and parameters are in a particular machine language. For example, the calling and called function code above would look something like this:

```
returncode = F0001X("myfile");
int F0001X(char* p001) {
	…
}
```

In the Java language, methods are defined in a class. Before resolving "external" method referencing, the Java compiler has to resolve the external class or object referencing. Because the java source files or class files can reside in different locations, it is very

likely that the names of the files or classes are conflicted, especially when many different programmers are working in the same project.

Java designers use the organizations of packages to reduce the possibility of name conflicts. Although a Java source file can contain relative class or object names, the Java compiler has to translate all of them to absolute name to avoid ambiguity for the JVM at run time. We will study Java's concept of namespaces in the next section.

Methods in Java are more complex to deal with because of the object-oriented properties such as overloading and inheritance. As similar to C compiler, the Java compiler makes sure that all the references are correct. As different to the C language, the Java language is much stricter in data type checking.

5.1.2 Dynamic binding

Dynamic binding is sometimes also called late binding. Java program can decide at run time which types to link. As seen above, a Java program can dynamically load a type by name at run time. Another way of doing that without writing a class loader is the `forName` method from Java API class `Class`. The simple form is:

```
public static Class forName(String className)
      throws ClassNotFoundException;
```

With only a type name required, a Java type can be loaded, linked, and executed from a program. A more advanced form of the method is as follows:

```
public static Class forName(String className, Boolean initialize,
      ClassLoader loader) throws ClassNotFoundException;
```

With this method, a user defined class loader can be used for loading of the Java type. And, with the Boolean parameter, the program has the option of initializing the type after linking. This is called Run Time Type Identification (RTTI) [22]. At this time, the class

data is copied into the method area or method table. The symbolic references are populated into the runtime constant pool and type variables are initialized. Hence, it is a form of late binding. It is what we need for the model.

5.1.3 Dynamic code loading with no semantics

Loading modules that have never gone through the processes of compiling and linking together before is a significant risk that the integrator has to take. Earlier in this chapter, we have paid much attention to understanding how code is loaded and executed at run time.

Because the JVM allocates a separate namespace for each class loader, a small model works better and more efficiently when LER is in the same loader space as the application. However, the setting of LER is the source of dynamic code loading. Programs that use IEC can treat LER as a source of local components. We previously learned Java RTTI as in the form of `forName` method call.

There are other techniques in Java language providing run time type information such as Reflection. Reflection allows the calling program to discover method information from other object at run time. It is the base for JavaBeans graphical programming user interface and Remote Method Invocation (RMI). However, the one original limitation remains – the human programmer has to know which method she wants to call and what computing tasks the method does. Although the syntax of calling the method can be obtained, there is still no semantic information about the method.

With that in mind, our model is going to restrict to a fixed interface and fixed method calling at this time, although loading of the component code is dynamic. We describe that in Section 5.2.

5.1.4 Trusted components

All the troubles and pains taking that the compiler goes through are for keeping programs error free. The goal is to reduce as much ambiguity as possible. Errors at compiling time are easy to correct compared to the ones occur at run time. Reusing software components is to skip these processes of compiling programs, from an application integrator's points of views. Changing the compiler verification processes requires changes or extensions to the compiler.

Since the integrator cannot do much with compiling of the components besides trusting the component maker, other methodologies maybe more effective at integration time, such as:

- modifying code at loading time
- modifying code in memory at run time
- loading alternative code

We are not going to consider the first two methods because our goal is to reuse components without changing them. We are going to derive an alternative way of loading code. But, we trust the component code to the LER, which can do the code preparations or verifications if necessary.

The program that uses an RIM just requests for the reference of the component in need and starts using its methods. It is up to the programmer to decide what to use out of those methods provided. Hence, component specifications and documentations are very important.

5.1.5 Component specifications and documentations

This requirement emphasizes more on the role of the integrator and component maker in order to make this model work. We start by asking the question, "which comes first?" – components from the component maker or the use of those components and their methods from a program that the application integrator writes. Actually, in software components' objectives, the two should be independent.

However, coming from the motivations of software components usages and demands, we suggest that the application integrator take more active roles. Instead of asking what components are available out there for me to use, the application integrator should state his intentions from using components within the contexts of the application to be integrated. For that reason, the requirement documentation comes from the application integrator.

The component specification comes from the component maker who follows the requirement documentation from the application integrator. However, if there is no such component requirement document at the time, the component maker can write her own specification based on the component's design and implementation.

The component requirement documentation and specification should be compatible in formats for the purposes of comparisons. The application integrator is to make decisions whether to use the components in her application.

5.1.6 Binary code compatible

This requirement is a given since the model works in a local machine. This simplifies much of the considerations that distributed models and standards having to deal with. However, there are different levels of compatibility:

- *Binary code format:* Since the components are sold and purchased for use without recompilation, their binary format cannot be altered. LER does not check for the binary format of the component it going to load. An incompatible binary format may cause unpredictable failures.
- *Programming language:* The interface abstraction of the model should be language independent. However, the component reference that is returned by LER may be language dependent. If the model is to work with components generated from different languages, the language that the integrator uses to write the application should be of first priority.
- *Frameworks and containers:* A drawback of frameworks and containers is that even when using the same language and the same binary format, the components cannot be used outside of its particular framework or container.

These are the concerns when implementing, porting, or extending the model:

- The interface should be language independent: The abstraction of the interface is like a contract between the application integrator and the component maker. It is important that they agree upon how the components should be used.
- The components should be binary compatible: Besides adhering to their interface, the components have to be binary compatible in order for LER to load them. LER should take care of specific component requirements before loading them so that the application receiving the component reference sees the same interface.

5.2 RIM and Component Methods

In this thesis, interface is an abstraction of data fields and methods from a type of components. These are defined as sub-components of the interface.

5.2.1 Interface Definition

An interface has a name, zero or more data fields, and zero or more method declarations.

The name of an interface is made up of alphabets or numeric. It could be recognized as meaningful for human programmers. In order for more than one interfaces to be used within the same program, their name have to be unique. However, these interfaces exist outside of programs. LER uses this interface name to search for its sub-components. It is necessary to have extensions to the interface name to make it unique. We discuss that in Section 5.3.

The data fields are variables and their initializations. These are public variables so that the program that is getting the interface reference can access them. The field names are also unique within an interface. Data types and variable evaluations should be as similar as in regular programming language. Notice that these data fields maybe just static or constant in order to discourage their direct modification. A better way to have access to variables in a component is through setting and getting methods.

The method declarations include a list of methods with return type, method name, and parameters and types. Following is an example of the interface requirement documentation. Notice, this is only in text format.

Figure 11 Interface Requirement Documentation

```
# This is a interface-requirement documentation
# Interface Name: ZipFile
# Attributes:
FileFormat = gz | jar | rar | tar | zip

# Data Fields:
maker = "mr. component";
errno = 0;

# Method Declarations:
# Method
Name: Open
In:   String filename
```

```
Return: filehandle
# Method
Name: Close
In:   Filehandle filehandle
# Method
Name: compress
In:   Filehandle inFile
Out:  Filehandle outFile
# Method
Name: decompress
In:   Filehandle inFile
Out:  Filehandle outFile
```

The above requirement document is for human programmers. Application integrators use it to state their need for a component. Component makers use it to develop a specific component. It is a basic form or template. A short form can be written as following:

```
maker = "mr. component";
errno = 0;

# Method Declarations:
Methods

   Filehandle open(String filename)
   int close(Filehandle handle)
   int compress(Filehandle inFile, Filehandle outFile)
   int deCompress(Filehandle inFile, Filehandle outFile)
End
```

In object-oriented language such as Java, abstract Class and Interface give similar effects. The above text content can be translated to a Java Interface as following:

```
public interface ZipFile {
   String maker = new String("mr. component");
   int errno = 0;

   public DataInputStream open(String filename);
   public int close(DataInputStream handle);
   public int compress(DataInputStream inFile, DataOutputStream
      outFile);
   public int deCompress(DataInputStream inFile, DataOutputStream
      outFile);
}
```

It is clear what the integrator's requirements are and what should the component maker commit to provide the satisfactory component. The document does not specify how the methods should be implemented. Therefore, a component maker is free to implement

different components. Actually, that is one of the motivations for the model. Take the `ZipFile` interface for example, the four methods open, close, compress, and decompress can implement any type of file format without violating the interface.

5.2.2 One interface, multiple components

In object-oriented programming language, an Interface represents the base class that other classes can be implemented. This thesis focuses on interfaces that may vary in wide ranges of implementations or functionalities. However, at run time there may be only one of the implementing components required to exist in a local environment.

We argue that it is not efficient coding practices to include the different varieties of the component when only one is used at run time. The difficulty is in letting the application choose the appropriate implementation at run time. In Chapter 4, we use LER to solve the problem by presenting to the program the same interface that mapped to a particular component at run time. The program has no knowledge of which component implementation is in use.

It may not sound conventional that a program does not need to know which component implementation it needs at run time, because the program usually has to go through many conditions and verifications to arrive at the correct component. We argue that most of the checking is for figuring out some existing local environment or attributes. We use attributes simply to present component characteristics based on which the local environment choose the component for the application.

In our methodology, interface is used like a classification for components based on implementations. The `ZipFile` interface, for example, includes the class of components that implement a specific file compressing and decompressing algorithm and format. A

component starts from bottom up. Our current view is a one-level of classification at a local environment, as illustrated in Figure 12.

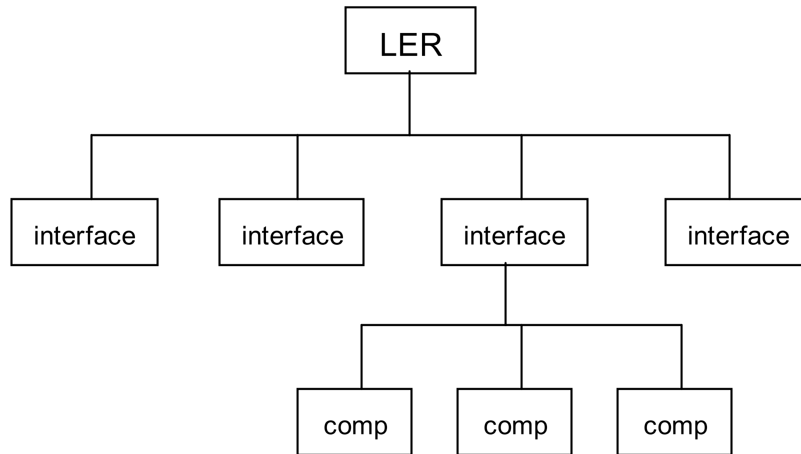

Figure 12 One interface, multiple sub-components

Although a sub-component can contain interfaces as well, it is a HAS-A relationship. There is no inheritance in interfaces. Therefore, the above picture remains the same regarding of the relationships of one component containing other interfaces. The HAS-A relationship can be presented with references or linking pointers. However, an interface can be nested in another component at run time. See Figure 13.

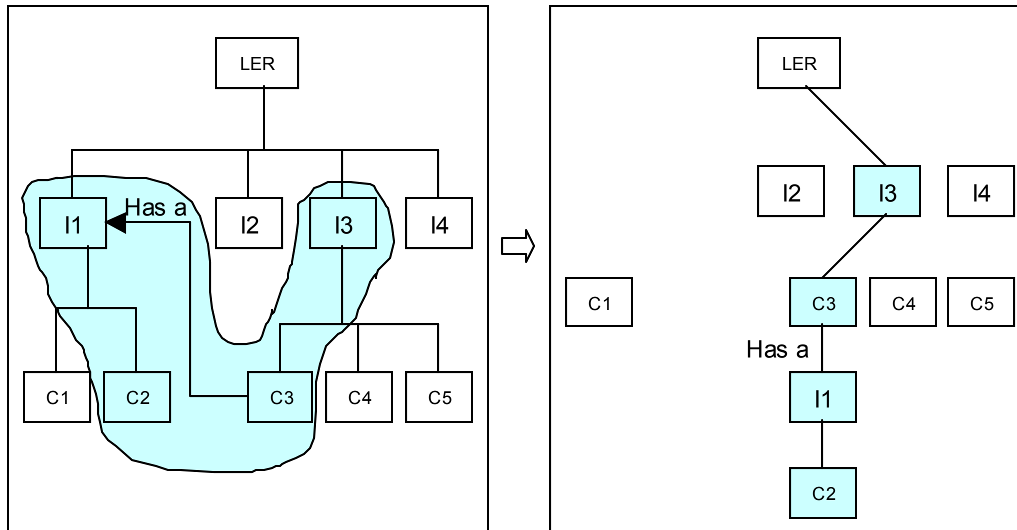

Figure 13 Nested components at run time

At run time, the interface and component boxes, which are paired, are merged together. And, the nested levels of components extend and shrink dynamically. We discuss RIM and LER namespace in Section 5.3.

5.2.3 Regular arguments

As discussed in IEC requirement documentation, the arguments for use in the IEC interface methods are the same as from the sub-component. Following is the same program previously used in Chapter 3. Once the program obtains the interface-component reference, it calls the methods as usual.

```
Object zcomp = LER.GetComponent("ZipFile")
file_handle fh = zcomp.open_file(filename)

zcomp.compress(fh)
zcomp.close_file(fh)
```

Usually, the compiler will detect errors in parameters' names and types, etc. However, at loading and linking time, the failures may be unpredictable. There are many unanswered

questions on how the program can dynamically recognize the errors or discrepancy in method signature. We will talk about this in Chapter 8, Future Work.

5.3 RIM and Component Namespace

As described in LER internals, when an interface is input to LER, the interface is mapped to a sub-component residing in the local machine. From previous sections, it is "one to many" mapping.

5.3.1 Local components and their namespace

In this section, we explain how a particular component can be recognized as representative of an interface in certain conditions. Because the naming rules are depending on the implementation of LER, what we are discussing here may be suggestions only.

In the local machine, LER has the root directory from where interfaces and sub-components are installed. Figure 12 shows a flat structure that each interface has its components under it. Therefore, the name could start with the interface name. However, we have to set aside actual file spaces for storing those components of an interface. We choose to have the name of the interface as a directory under the LER ROOT.

For instance, a LER has its component ROOT at directory /LAIM. A registration of a new interface, `ZipFile`, causes a directory named, `ZipFile`, created under `/LAIM`. Then, all the components that are added to interface `ZipFile` will be store at:

```
/LAIM/ZipFile/
```

The directory represents the interface. It is also a way of organizing components to their interface. The rest of the discussion regarding how a component can be named.

Since we group components based on an interface, components from different makers are going to reside in the same directory. The first thing component makers may want to contain in the component name is to set aside the component from others. It could be a brand name from the component maker. Or, it could be a representation of the implementation. In the requirement documentation, this maybe represented as “maker”. Following are some examples:

```
ms.comp | compguy | dr.lego
```

Different component makers may develop components for the same interface. These components are different in implementation. One way to represent different implementations is by using attributes. For example, the above ZipFile interface has one of its attributes called for different file format. The requirement documentation has a line of attribute specifying:

```
ace | gz | jar | rar | tar | zip
```

Some attributes may not be listed in the requirement documentation. For example, each component may be made at a different version. Or, it may be written using different language, etc. Although it is good practice to have one attribute represents a component, multiple attributes on a component are inevitable sometimes. We offer two ways to combine the attributes. Each way uses a different delimiting character, we called OR and AND.

- One of attributes: for all those attributes, the component requires only one present in order to work. These attributes are separated by the OR delimiters.
- All of attributes: the component requires all attribute present in order to work. These attributes are separated by the AND delimiters.

Following are two examples of a full name of a component of the `ZipFile` interface. We use "+" for the OR delimiter and "*" for the AND delimiter.

```
tar*2**dr.lego
gz+zip++ms.comp
```

The first component is made by `dr.lego` and it only works with the tar file format. The number 2 represents version 2. The second component is made by `ms.comp` and works with either gz or zip file format. Note in both cases, the pair of delimiters signals the end of the attributes. We derive the syntax for naming a component as following:

```
Attribute[dAttribute]*[ddIdentifier]
```

These are the rules extracted from that syntax:

- Only the first attribute is required for the name of the component
- Optional attributes can follow by a delimiter
- d is the delimiter, which has to be the same within a component name
- The maker string is optional at the end. It has to be preceded by two delimiters
- There is a limit on how many attribute values can be part of the component name

Notice that the component maker does not have to make up these names manually at the time the component is installed. For instance, the default attribute can be extracted from LER. The maker identification may come from the package information. In the next section, we discuss how this naming convention is implemented in LER.

There is one draw back in naming the component with values of the attributes when the number of attributes increases. In that case, we suggest eliminating the attribute part in the component name and just use:

```
ddIdentifier
```

When LER sees the component name start with two delimiters, it looks into a text file name with the same Identifier and the extension ".attributes" to find out the attributes related to the component. LER writes the attributes into the file `Identifier.attributes` the same ways as it does to name the component.

Another design alternative is to store every component associated attributes in LER. For the attributes to be referencing to a component outside of LER, a unique identifier is generated and attached to the component name. See Figure 14.

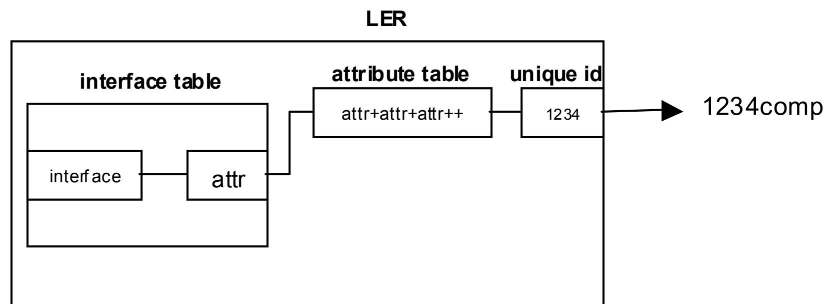

Figure 14 Alternative LER internal namespace

5.3.2 LER as component dealer

As described in Section 4.2, LER Internals, LER takes care of the registration of an interface and adding of components into existing interfaces. LER also loads the appropriate local component, from its registry, upon request made by an application. Armed with the knowledge of LER internals and the component namespace, we are going to discuss the two processes of registering an interface and searching for a component.

5.3.2.1 Registering an interface to LER

A program can use one of the two methods from LER to register an interface:

```
public static int registerInterface(String interface, String
      Component)
```

```
public static int registerInterface(String interface, String
      Component, String Attribute, String Value)
```

Register with default attribute

If only the interface is registered with the first method, the component is associated with the default local attribute and its default value.

At start of `registerInterface`, the input interface is checked against the interface table. If such interface already exists, the method returns immediately with error code indicating that the interface cannot be registered because it already exists. Next, the attribute table is checked for a default attribute. If there is no default attribute set, the first attribute is chosen. The first attribute could be the one that has highest priority. Similarly, a default value or a high priority value has to be found for this attribute in order for the naming of this new component.

For the example of the `ZipFile` interface, a component called `TestCompress` has to be installed. As the name of the component indicates, this component does a test on the file compression component. As a test component, it should be able to do the test on any file format. Therefore, the component should pick up the default file format. The register program calls `registerInterface` as following:

```
rc = LER.registerInterface("ZipFile", "TestCompress")
```

Because this is the first time `ZipFile` is registered with LER, the directory ZipFile is created under the LER component root directory, /LAIM. And, the name of the component is:

```
Default++TestCompress
```

In place of the attribute, Default is used to indicate that this component can be used with any attributes associated with the interface `ZipFile`. Different string or symbolic characters can be used in place of “Default” as long as LER recognizes that as the

meaning of working with any attributes. Another option for this naming convention is to leave out Default such as:

```
++TestCompress
```

We discuss how LER uses this knowledge to find a local component in Section 5.3.2.2.

Register with specific attribute

Instead of registering for the `TestCompress` component, this time we want to register the `ZipFile` interface with a component that can do compress and decompress for the tar file format only. The register program has to use the second method:

```
Rc = LER.registerInterface("ZipFile", "mr.comp", "FileFormat",
      "tar")
```

First, we are going to assume again that the `ZipFile` interface is not registered in LER. As the result of this method call, the directory `ZipFile` is created under /LAIM and the component is named:

tar++mr.comp

Attribute `FileFormat` becomes the default attribute that associates `mr.comp` to `ZipFile` interface. The attribute value tar becomes the default value for the attribute `FileFormat`.

In the case of the interface having already been registered, the register program has to call the `addComponent` method to load this component into the interface namespace. We recall the signature of `addComponent`:

```
Public static int addComponent(String Interface, String
      Component, String Attribute, String Value)
```

To add the above component for the tar file format, the call is:

```
Rc = LER.addComponent("ZipFile", "mr.comp", "FileFormat", "tar")
```

For the addComponent method, LER looks for the interface first to make sure it exists. If the interface does not exist, the call fails. It forms the name of the component as following:

```
tar++mr.comp
```

That component name is compared with existing components in the `ZipFile` namespace. If one component with exact name is there, a backup version is made and the old component is over written with the new component. Future extension may consider automatic versioning the component.

5.3.2.2 Searching for a local component

We start by assuming that LER has the interface `ZipFile`, which has two components locally: `TestComp` and `mr.comp`. In this section, we show how a call of the method `loadComponent` searches for the appropriate component. We recall that there are three forms of `loadComponent`:

```
public static CompRef loadComponent(String interface)
public static CompRef loadComponent(String interface, String
      attribute)
public static CompRef loadComponent(String interface, String
      attribute, String value)
```

As can be seen from the first parameter of all three methods, the interface is the first thing to be searched. If the interface does not exist, the call fails. If the interface exists, further searches will be done for a local component that matches the attribute criteria.

Searching for default component

A default component is the component that is associated with the default attribute of this interface. A program uses this method call when it does not know the specific attributes of the interface and it just needs to load the default component for this interface. The method provided by LER for searching this component is:

```
public static CompRef loadComponent(String interface)
```

The interface name is the only input to the method. LER passes the interface as a key into its interface table to get the default attribute for this interface:

```
attribute = interfaceTable.get(interface)
```

This is the default attribute, which is again hashed into the attribute table for the default value.

```
attrValue = attributeTable.get(attribute)
```

Once this value is found, the component can be located. As we've seen above, there are three alternatives:

- The attribute value can be part of the component name
- The attribute value can be in the attribute file of the component
- The attribute may points to a unique identifier that is part of the name of the component

Loading of the component is discussed in 4.2.

Searching for a component with attribute and default value

In addition to the one interface input, LER provides a second method for searching of an interface associated with a specific attribute:

```
public static CompRef loadComponent(String interface, String
      attribute)
```

Only a call to the attribute table is necessary to find the default value of the specified attribute:

```
attrValue = attributeTable.get(attribute)
```

Once the value is found, the component can located as the alternatives described in previous section.

Searching for a component with attribute and value

The third method for searching of an interface requires specific value of the attribute with a third parameter:

```
public static CompRef loadComponent(String interface, String
      attribute, String value)
```

This method is used in case of an attribute having multiple values and the component is associated with a value that is not the default one.

Searching for a component for an interface with multiple attributes

A complex interface may require more than one attribute to represent a local component, as described in Figure 8. The second `loadComponent` method can be used with the second parameter as a combination of the attributes, such as following example:

```
loadComponent("ZipFile", "rar+tar+zip")
```

5.4 Other Programming Languages Consideration

The goal of this model is dynamic loading of code at run time. That immediately rules out programming languages that cannot do late binding. In other words, using early binding, all code needs to be loaded at the beginning anyway. There are no advantages in using IEC.

C for example, does early binding. C modules are usually built into libraries that lead to our previous discussion about DLL. Although DLL can be loaded and unloaded dynamically, the modules are all in memory unless they are swapped out by system operating services such as paging.

Script languages such as Perl or Python may be candidates for writing the integration application because they provide many mechanisms to glue together modules from other different formats or languages. Python, for example, provides Inter-Language Unification (ILU) for building multilingual, object-oriented class libraries with well-

specified, language-independent interfaces. ILU uses a declarative language called Interface Specification Language (ISL) for purposes of defining object and non-object types [43]. Following is a sample of ISL:

```
INTERFACE ZipFile;
EXCEPTION SomeError;
TYPE ZipFileTAR = OBJECT
   METHODS
      open (fname : CHAR),
      close (fh : INT),
      compress (ifh : INT, ofh : INT),
      decompress (ifh : INT, ofh : INT)
   END
```

Notice that the above specification may not have exact syntax. However, it illustrates the similarity in requirement documentations. And, translating a text file in human understandable format to computer format is a common task.

Although XML is not a programming language, it has strength in data presentations and documentations or specifications. If we are going to extend the interface requirement documentation to be an interface that is understood between components and application, we should consider using XML.

Chapter 6
Component Packaging with RIM and LER

When a user buys an application, it usually comes as a package. Except for some small or onetime deal applications, software vendors usually build the package from their development environments and then ship it out to customers. Most of the customers do not know all the details about components that are in the package. They run the installation program that takes care of all the application prerequisites. Therefore, installation applications carry the missions from the application maker to deliver the application package to the user system.

In Section 2.5.1 of Chapter 2, we learn that the developer has to input or specify much information about the application for the installation application, deployment tool, or package management application. Then, the information is included in a specification or initialization file. This file also lays out the requirements for the application and the steps or instructions for its installation.

Letting the developer enter all the information manually may not be such a good idea because it is easy to make mistake and the procedures are tedious. Many development tools or virtual environment deployment tools make it easier by presenting the information in graphical user interface for the developer to fill in the requirements. Then, those tools generate the necessary specification file.

On the other hand, the developer who produces the package needs to find out about the system that his application is going to be installed. Operating system commands and install scripts serve the purposes. A package management application also helps user find

information about a package and components if it is already installed in the system. LER has these functionalities.

Nevertheless, LER is more than a package management application. It is responsible for loading local components for applications to use at run time. It also lets applications shared components if they are packaged with RIM method. Component packaging tools should take advantages of the LER API, which does most of the component organization on the local machine. Using a set of published API hides the local environment implementation.

Components that are packaged and installed into a local system this way can find each other. Actually, LER does all the searching for components and loads them at requests. The components just have to register their interface, as we describe in Chapter 4. Both the component maker and application integrator need to utilize the packaging tool.

In Chapter 2, we learn about component deployment methodologies in virtual environments such as EJB containers. We also learn about installation and package management applications to assist users in setting up applications in specific operating systems. Although end users might not want to know where their application installed or store data, component makers and application integrator have to work with the locations of the components and methods of maintaining them.

Component makers can store their components anywhere in the local machine. However, there must be ways for the application integrator to find the components and use them. Notice that LER does not include the components. It has the name paths leading to the components. Therefore, a registered component can still be used outside of

LER by any other ways. For example, if an application knows where the components for the interfaces are, it can access them directly without going through LER.

However, accessing local components directly will defeat the idea of this thesis to start with. We do not want an application to bog down in checking logic conditions and specific implementations. We depend on LER and the packaging process take care of the local different characteristics for the application.

6.1 Packaging Requirements and Guideline

6.1.1 The LER API

First thing a packaging tool needs is the LER component to exist in the local machine. We will discuss later in Section 6.2 on how a packaging tool can install its own LER if there is none in the local environment. In this section, we assume that LER exists in the system where the package is going to be delivered. If LER does not exist, any first call in this section will fail and the application will know that LER does not exist.

A packaging tool needs to use the LER API for these purposes:

- Registering a new interface to this LER
- Adding one or more components to existing interface in this LER
- Removing or upgrading components or interfaces in this LER

For simplicity, we are going to assume all the package managing processes are carried out by one user or administrator user who has all the access authorization on this system. Broader packaging scenarios can be applied to multiple users or multiple LER's in one system. However, other issues have to be considered such as:

- User authentication for security and trusted component assurance

- Access permission controls such as file blocking and LER data information integrity
- Naming conventions

6.1.1.1 Registering a new interface

The name of the new interface should be well defined. They are preferably coming from the requirement documentations for those interfaces. The method to call is:

```
registerInterface(String interface)
```

where interface is the name of the new interface. This method returns a Boolean value indicating the register has been successful or failing. If the value is true, the register process has completed successfully. Otherwise, the register failed. Because we are considering the simplest case, failure of register means that the interface already exists. However, if the call fails even before returning, it is possible that LER does not exist in this system. It may be helpful that LER also provide a method to list existing interfaces.

If an interface is registered in this simple format, it has to be followed by adding of the components into this interface. An interface without mapping to any components cannot be loaded.

6.1.1.2 Adding components

As described in the LER API, a component can be added in the mean time with registering of a new interface. The method to call is:

```
registerInterface(String interface, String comp, String
      attribute, String value)
```

Notice that all arguments are in String format for simplicity. It is for convenience because in Java, the `String` class provides the `equal` and `hashcode` methods for using as

a key in a hashtable. We can get the methods to work with `String` arguments. Then, the strings can be converted or replaced with any other data types later.

Similar to the simple format of `registerInterface`, this method fails if the interface has already existed. If the attribute does not exist, it is added into the attribute table with the value. Then, the interface is associated with the component by the attribute's value.

It is important to associate an interface with a specific value of a local attribute. If the packaging program intends to associate this interface and component with the default attribute in this environment, it passes the null string to the arguments attribute and value.

The packaging program can also query the default attribute. Or, it can call a method from LER to associate an attribute value with this interface and the component:

```
setDefaultAttribute(String attribute, String value, String
      interface, String component)
```

This method can be called anytime after the interface is registered.

To add a component to an interface, LER needs to move the physical file of the component to correct location. Then, it forms the namespace or search path for the component. See Section 5.3. A method from LER can be called to set the paths for the locations from where components are copied.

```
setSource(String paths)
```

After the paths are communicated to LER, subsequent `addComponent` calls will look into these locations for components. The method for adding components has following format:

```
addComponent(String interface, String comp, String attribute,
      String value)
```

Notice that the above parameters are identical to the method registerInterface. The difference is, here the interface has to exist for this method to succeed. Similarly to the

other method, this `addComponent` method matches an attribute value to the component for referencing the interface.

6.1.1.3 Upgrading components

To keep the LER API simple, we just add one `removeComponent` method for managing components. If a component has to be replaced with a new component, the old component is removed then the new component is added. The LER method for removing a component is:

```
removeComponent(String interface, String comp, String attribute,
      String value)
```

Notice that, LER has to search the interface for the component that associates with a specific value of an attribute. That is how LER form a unique name for a component. If the component is found, it is removed and a Boolean value of True is returned. Otherwise, the method returns False.

A packaging tool should also consider backing up the component it has just removed for recovery purposes. While the current component may be working fine, replacing a new component implies certain risks. Therefore, saving or archiving the old components gives the user a chance to go back to those components if the new components fail.

Other alternatives in upgrading components include versioning the components. The version numbers may be part of the naming scheme that LER uses.

6.1.2 System native interfaces

LER is very simple and small implementation of a registry providing common services for components. It does not provide all the system services file system operations or user account information.

A packaging tool sometimes needs to know user information for verifying the purchase of the package. The C language provides system calls such as the `getenv` function for query value of environment variables. C also provides many functions for getting user information, such as `getuid`, etc.

The Java language provides couples of ways for accessing the environment from Java applications [76]. A Java program can use the Java Native Interfaces to call C or C++ functions [44]. A C/C++ program or any executable can also be invoked inside a Java program using the Runtime class' exec method, for example:

```
Runtime rt = Runtime.getRuntime();
Process p = rt.exec("genv", env);
```

`genv` is the name of an executable or application. Also in Java language, the System class provides accesses to system services such as standard input, output, and error. It also provides method for setting and getting environment variables in the form of Properties that is also a Java class. For example, a list of local system environment variables can be obtained and printed out on standard output as following:

```
Properties p = System.getProperties();
p.list(System.out);
```

In Section 2.5.2, we study the jar file and its manifest file, which contains a list of properties-value pairs. A packaging program can create similar properties file to store and retrieve properties. Following are two examples of storing and retrieving properties respectively from a file.

```
// Storing properties into the file "package.prop"
FileOutputStream fos = new FileOutputStream("package.prop");
BufferedOutputStream bos = new BufferedOutputStream(fos);
Properties p = new Properties();
p.setProperty("key1", "value1");
p.setProperty("key2", "value2");
p.store(bos, "Packaging");
bos.close();
```

```
// Retrieving properties from the file "package.prop"
FileInputStream fis = new FileInputStream("package.prop");
BufferedInputStream bis = new BufferedInputStream(fis);
Properties p = new Properties();
p.load(bis);
bis.close();
System.out.println("key1=" + p.getProperty("key1"));
System.out.println("key2=" + p.getProperty("key2"));
```

6.1.3 Component delivering

To deliver components is to copy the component files from some storage media into the local system. We see the media as the sources containing components. These sources can be portable such as CD-ROM or a mounted file system such as NFS. Most likely the components are in compress file format such as jar or tar, etc. in order to reduce the requirement for storage spaces. Therefore, packaging tool may require the use of those file-decompressing applications.

If the local system does not have the one decompressing utility that is required, the packaging tool has to prepare for some other resolutions. Having a `ZipAF` component available would be an answer to this component packaging and delivering problem. `ZipAF` is a simple component that we discuss in Chapter 3.

A packaging tool may also be required to search for the sources containing components so that component names and paths are not hard coded in the program. Both C and Java provide many functions and methods for file system operations.

6.1.4 Components for sharing

Because a component is implemented following the interface requirements, any program can use the component based on the defined interface. This is independent of how the components are implemented or built.

Once a component is loaded by an application, other applications that request loading of the same component use the one that is already loaded in memory. When LER loads a component, it keeps a static reference to the object. When it receives a second request for the same component of the same interface, it returns that object reference without loading a new one. Following is an example of how it is implemented in LER.

```
public static Object loadComponent(String interface, String
component, …) {
      static Object inMemObj = null;
      static int callCount = 0;
      …
      if (callCount > 0) {
            return( inMemObj);
      } else {
            doLoadL(…);
            …
            callCount++;
      }
}
```

When the packaging tool registers an interface and finds out that the interface is already in LER, it assumes that the existing interface is the same as its own.

6.2 Putting together a package

The list of components for a typical package may include the following. Notice, not all of the components will be installed to the local system depending on contents of the system at the time.

- A LER for quick start
- Interface specifications
- Components and other files in compressed format
- Compressing and decompressing applications or components
- Manifest or deployment files

All these components have been described in details earlier. Putting together a package is a good way of reviewing or summarizing what we have to offer with this thesis.

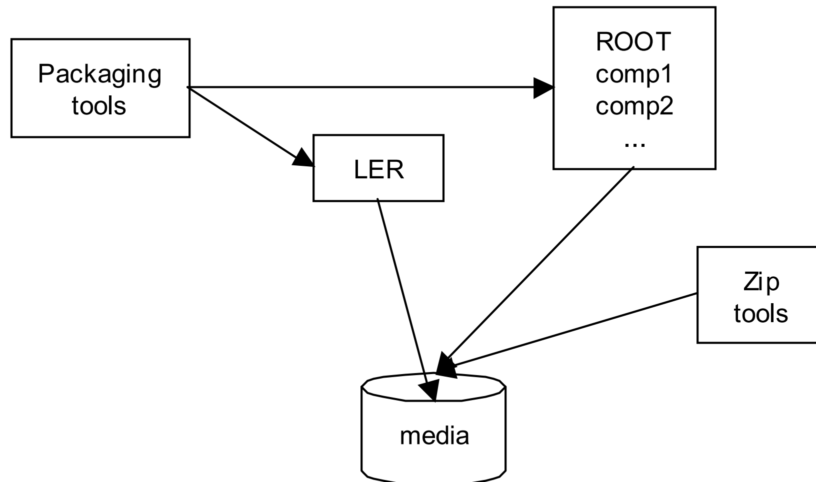

Figure 15 Packaging tools

First, we think of a user who starts this application integration methodology for the first time on a system. There could be other components that are already used by other applications. However, there is no LER on this system. A LER is a core component for a package. It consists of the Java classes or executables for the LER. The package tool has to decide two things:

- The HOME directory for LER
- The ROOT directory for interfaces and components

This LER can contain an empty interface table and an attribute table to start with. Or, the package integrator can load the interfaces and components in this package into LER first. This way, the LER has already had all the necessary registry information. The package can just copy the LER and the components to the local machine and it is done. However, the drawback is, if the local machine has already had a LER then the packaged LER has

no use. The packaging tool has to register the interfaces and add the components all over again to the existing LER.

If it is found out that an interface to be registered by this package has already existed, it is important that the packaged interface can be compared with the existed interface. If the packaging tool cannot do that automatically, the user has to be notified in some way. This is to avoid many problems when running the applications using the interface later.

We did not have rules as per what type of file compressing format should be used in our package. That is left optional for the package integrator to decide. Thus the manifest or deployment files may vary.

Since it is uncertain at the time of packaging if the local system will have the required compressing utility, it is best to include the right one in the package. It is even better if these compressing and decompressing tools are written as components using our RIM method. We've described the `ZipAF` component in Chapter 3. In the next chapter, we will use that component in more elaborate examples.

Chapter 7
Examples of Application Integration

The example application shown in this chapter leverages the methodology in this thesis over other approaches. The application characteristics usually play decisive factors in its design and implementation. This thesis' approach and methodology benefit certain applications in particular domains. Following is a list of our considerations for choosing this application.

- Number of machines or systems the application has to be installed and running on
- Heterogeneous characteristics of the systems
- Availability of existing software frameworks or middleware infrastructures
- Availability of existing software components
- Requirements in installing all the software

Virtual Pet is an application that grows on a computer. It is started at a simplest form based on what it is defined to do. In the growing process, it gathers the materials required for its activities. It requires human user to care for it in certain scenarios. By caring for the pet, the human user directly or indirectly taking care of her system or system resources.

A virtual pet can be thought of as the traditional monitoring application. Intelligent agent is another form of monitoring application. The difference is, intelligent agents are simulation of the human users watching their system. Usually an agent is prepared for the knowledge in watching particular resources or activities and then taking actions accordingly. On the other hand, virtual pets simulate what they monitor. They start in

simplest form and then bind to the resources or activities of interests. From there on, the pet's activities or capabilities closely reflect the activities. The human user just watches her pet grow or play and gets the pictures of activities going on with the resources.

The virtual pet application is chosen because of its diversity in many aspects of designs and executions . Although virtual pets have been popular, there are no clear descriptions on what virtual pets should be. Most of the characteristics of virtual pets depend on their makers and buyers. There are no limits on what types of virtual pets a programmer or a pet owner can think about. In general, there are two types of virtual pet makers and owners.

- Conservative minded people, who like their virtual pets to resemble the real pets that are familiar and loved. They know for sure that these somehow common virtual pets require certain familiar environments. As long as the environment requirements are met, these virtual pets can live a normal and predictable life.
- Adventurous minded people, who like to create or adopt unique and exotic virtual pets. They can put any characteristics on these creatures because the virtual nature of the pets. The majority of these people are computer game players who always seek new excitements. They will watch with curiosity how their virtual pets turn out different than others.

Nevertheless, this example starts with traditional Information Technology (IT) in mind. IT concerns and considerations are put in light of this thesis approach and methodology.

We begin the example at a setting of some far away village, where many server farms are forming. Each farm has ten to 100 servers. These servers are not all kept together in one place. They are scattered in farmers' houses. Tens to hundreds of farmers come

together as joined adventure farms. However, each farmer owns his own servers. There are no regulations or requiring what types of server farmers should have. In addition, farmers are free to install any kinds of software on their servers. They will also buy new leading edge technology servers at anytime when needed.

When the farmers hear of new species of virtual pets, they want to start raising these pets on their servers. In the next section, we describe these virtual pets in details.

7.1 Virtual Pet Application

The first thing that comes to people's mind when talking about virtual pets are creatures that "live in" their computer. The creature may not look anything like the animal pets we know. Examples are the Virtual Pet Creature from RJB Production [63] or Creature Labs [19]. However, the ideas of virtual pets are usually to simulate the pets that we know and love such as dogs, cats, fish, etc. These pets keep the computer user company, as pets are people's best friends. The latest form of this human-pet relationship is Bonzi BUDDY [13]. This pet, in the form of a monkey, does a whole lot of automated tasks for his human friend. He is more like an agent or a personal assistant who has the support of networks of portals on the Web.

The list of Virtual Pet Patents [62] shows that virtual pets researches are usually for the purposes of computer games or companions. Very few patents show how virtual pets can be built. The researches mostly show how the pets can be simulated or presented visually. Also common are the interactivities between the pet and the human user.

In this example, we create our own types of virtual pets using our application integration methodology. The characteristics of these virtual pets are as following:

1. The virtual pet's components or body parts are assembled at run time, although it may not be accurate to call those body parts. A pet may not have a body.
2. There are many variable ways to show the pet visually because there are no ways of telling which components are going to be assembled into the pet.
3. Each instance of a running virtual pet may have different components. Hence, there are no two virtual pets are identical, on the same machine or different machines.
4. A virtual pet is a write-once run-anywhere creature, which is created with the same methodology. However, virtual pets are not homogeneous species.
5. Because virtual pets can be differentiated by natures, they can be used to represent different characteristics of a machine.

Following are some examples of tasks that these virtual pets can do.

- Watching the usage of CPU's in a machine, including the balance of loads on those CPU's
- Maintaining multiple resources on a machine, such as communication ports, storages, file systems, etc.
- Keeping track of software components and their usage in a machine

Those tasks can be general in any machines. However, the detailed procedures on achieving results are diverse enough to have a virtual pet doing completely different procedures on a machine.

In terms of interface and implementations, the general task is like an interface. Although the virtual pets have to adhere to particular interfaces, they can take advantages of different implementations.

7.2 Virtual Pet Requirements and Analysis

7.2.1 Write once, integrated on many systems

Every farmer wants to raise these new species of virtual pets on their server but they have to consider the investments in user skills and server resources. It would be best if the servers already have the required software components. But those are more than just the basic software components.

Since different farmers own servers, it cannot be assumed that all servers should have the software framework such as CORBA or EJB installed. More importantly, existing servers have to run their everyday productions, which may be interrupted and effected by the installation of new software framework. Both CORBA and EJB require the machines to be set up as servers or clients.

A better time for trying this new virtual pet species is when the farmer acquires a new server. The new framework can be installed and the new virtual pets can be tried out. But, the question remains – if the pets are to be propagated onto existing servers, are the farmers ready for that? Because the server environments can be very different, running a pet on one machine does not guarantee it will run on another machine.

This thesis' methodology uses the existing system software plus the components that are required. There are no servers or clients requirements. A package is installed on whatever machine for the virtual pet to run in. This package does the integration for a run-able virtual pet in this particular machine.

The same package can be installed on a different server and a virtual pet is created particularly for this machine. Although it is the same package, the created virtual pet may be different depending on run time environment of each machine.

7.2.2 Adapting to individual system

In the framework or middleware environments, a virtual pet has no problems adapting because the environments are required to be created the same. All the machines would have had the framework installed and configured, in order for the virtual pet to be implemented with this framework. In these server farms, there is some framework known by the name Composition Framework (CF). But it is not known how widely it exists on the machines.

Because the farmers own the system, they cannot be forced to all upgrade to CF. A survey has to be done to see how many systems have already had CF. The survey should include an inventory of the system resources to see if it is appropriate for installing and setting up CF. In order to get these machines up to CF, their owner need supports in system resources and skills for using the framework efficiently. The preparation processes are involved with high number of systems before the design and development of the application, virtual pet, even start. The scope of these preparation processes depends on the number of systems.

Virtual pet is an example that the application can be designed and developed simply on one system and propagated to other systems afterward.

7.2.3 Programming interface considerations

Regardless of the use a framework or not, the virtual pet ideas require thorough considerations on the component interfaces. Because a virtual pet can take on different composition materials and activities at loading time, object oriented methodology such as polymorphism is essential to this design. It is more crucial that a different

implementation of a pet or its component can be loaded at run time. The Java Interface can be used to demonstrate the idea.

Assuming that the virtual pet has an interface for controlling its activities. The Java interface is called `ControlUnit`. There are three implementations available for use with this interface:

- `PetBrain`: A neural net simulation of the pet's brain
- `PetLogic`: An AI simulation of the pet's brain
- `LER`: A component mapping interface

The Java code segment that uses Interface and Polymorphism is as following:

```
ControlUnit petBrain = new PetBrain();
Existence myPet = new Existence();
myPet.init(petBrain);
```

The class Existence can be initialized to a `ControlUnit` interface, which can be determined at load time to one of the above implementations: `PetBrain, PetLogic`, or `LER`. As seen in the above code segment, the init method of the `myPet` object can receive an object, `petBrain`, which implements the `ControlUnit` interface. The `petBrain` object can be easily changed to initiate the `PetLogic` or `LER` component as:

```
ControlUnit petBrain = new PetLogic();
```

or

```
ControlUnit petBrain = LER.loadComponent();
```

The Java Virtual Machine uses the advantage of late binding to match the component `PetBrain` with the `ControlUnit` interface as long as `PetBrain` implements all the methods in `ControlUnit`. The Java Interface is explained in the language references and many other Java books.

Notice that the implemented component has to be explicitly specified and the object, `petBrain`, has to be instantiated somewhere in the program before the it is passed to `myPet.init()`. Therefore, if a new pet control component is added later with a different name such as PetControl, this program has to be modified to use that component:

```
ControlUnit petBrain = new PetControl();
```

This is one of the short falls of Java Interface. There are other disadvantages of Java Interface such as [41]:

- Significant increase in package sizes
- No version migration path
- Name scooping
- No default implementation

Another less popular technique from Java is the ability of loading class files at run time by specifying file name. Without the use of Interface, the above code segment can be rewritten as following:

```
Class controlUnit = Class.forName(args[0]);
Object petBrain = controlUnit.newInstance();
Existence myPet = new Existence();
myPet.init(petBrain);
```

However, because it can be assumed that the class is discovered at run time, the programmer has to go through complex querying procedures to obtain information about the new class. Basic information can be names of the methods and their signatures.

The `LER` interface makes use of this capability and the interface concepts. As seen below, the call to instantiate the component is replaced by a call to the `LER`. This call will never have to change for a new component. In fact, `LER` supports the Run-time Interface Mapping Methodology. Depending on the local attribute setting of the `LER` interface, the reference of a locally loaded component is mapped to `petBrain`.

```
ControlUnit petBrain = LER.loadComponent();
Existence myPet = new Existence();
myPet.init(petBrain);
```

It is also possible to change the implementation of the `LER` interface for new or unique virtual pets.

7.2.4 High priority for owner preferences

Adopting a virtual pet and raising it on a machine is a very personal decision. For real pets at home, owner usually adopt them at very early age. The owner has better chances of training and bonding when the pet is still small. For virtual pets, the user or the pet owner has to have priority over characteristics and activities for their pets.

Besides working on the premises of locally available components, RIM and LER also give highest priority to user preferences. The user can change the attribute setting in LER to control the mapping of interfaces to components. There are alternatives for user preferences.

7.2.4.1 Configuration file

Using configuration file is the most common methodology in setting up user preferences. It is simple and may be most similar to programmer because it has already been done with many applications. The configuration variables in the file have to be defined before the program is implemented. The program has to do all the checking and validation on these variables.

LER can do all that for the program based on local environments. Therefore, logic and code can be reduced from the program.

7.2.4.2 Local database

A local database can also be implemented in different ways:

- One that is only proprietary to the program so that it can implemented efficiently just for the program
- Use of standard database available commercially for storing and retrieving attributes information

LER is similar to the first type of database but it hides the organizations and queries from the program. It does the jobs by well-defined programming interface for the specific purposes of mapping and loading components.

7.2.4.3 Remote server database

This would be appropriate if the virtual pet application is written as client server programming model in which user gets access to her virtual pets via a client application. The real virtual pets reside on the servers or Web servers. The user uses a client user interface or the Web browser to interact with her pets.

The client server version of this program would have a very different design and implementation. The client requires a network or communication connection to get to the virtual pet. If the user has any preferences, the data may have to be stored in the server.

7.3 Virtual Pet Interfaces

From the descriptions of virtual pets in Section 7.1, both the component makers and application integrators have to agree on the minimal requirements for interfaces. Defining the interfaces is the first step. The application integrator has to search for components that provide these interfaces. An application integrator accumulates

components every time new interfaces are required. For this virtual pet application, four interfaces are needed. If some of these components are not available, the application integrator has to write them.

First, a virtual pet requires an interface that loads its components dynamically at run time. A virtual pet also depends on this interface to provide the appropriate components existed in this run-time environment.

Second, a virtual pet requires an interface that presents the pet's existence, such as:

- The materials or components making up its body
- Places that it thrives on

Third, a virtual pet requires and interface that shows its activities, such as: growing, moving, etc.

Finally, a virtual pet requires an interface for controlling its existence and activities.

7.3.1 LER interface

The LER interface and its design and usage are discussed in details in Chapter 4. There is also interface reference in Section 4.3. See Section 4.4 for how to use the LER interface in a program.

In the virtual pet application, LER provides the control interface. Virtual pet program can use LER to store and retrieve user preferences. Also, for an example of a virtual pet that keeps track of software components in a local machine, LER has all information about components it knows.

7.3.2 Existence interface

A virtual pet has different ways of presenting itself into existence. For demonstration purposes, the Existence interface has several methods: `init`, `isAlive`, `registerEvent`, `unRegisterEven`, `waitEvent`, `final`.

7.3.2.1 Init

Although loading of the component marks the existence of the virtual pet, the init method provides a way of initializing a beginning form of the pet. It is suggested that a control unit type of object is passed into init. Then, this control unit is used for interacting with the virtual pet.

The Control Interface is described in Section 7.3.4. Because the Control Interface may have many different implementations, there are as many ways of implementing this init method as well.

7.3.2.2 IsAlive

This method could be a simple returning of True or False. Then, other methods are provided for more detailed status of the virtual pet. Or, different ways of keeping tracks of the virtual pet can be provided.

`IsAlive` can return a reference to a data structure that contains more information about the virtual pet. Even a simple true or false can be implemented in different ways. The options are Boolean value or integer value that represent true or false. An integer value can also be used for a returned code, which may be different from implementations.

7.3.2.3 RegisterEvent

During the time of virtual pet existence, events can be registered so that actions can be carried out when that event occurs. After the event happens, it is registered again for the next occurrence. This method may cause trouble if the event happens rapidly.

7.3.2.4 UnRegisterEvent

A registration or wait for an event is removed. Notice that the handle of an event is required. Therefore, a virtual pet can only un-register the events that it originally registers.

7.3.2.5 WaitEvent

This method is similar to `registerEvent`. However, it only waits for the event to happen once. Because a virtual pet is like a background or daemon process, this method does not block what process execution.

7.3.2.6 Final

This method is one way to end a virtual pet. A virtual pet is like a background process. It is either active or dormant. The final method can be implemented as an event interrupt to cause the virtual pet to wake up or stop what it is doing.

The final method also provides a chance for the pet to wrap up or save its data or states before exiting.

7.3.2.7 Requirements

Following is the text file that contains the requirements for the Existence interface.

```
# This is a interface-requirement documentation
# Interface Name: Existence
```

```
# Data Fields:
constant Event EXIT
constant Event MEALS
errno = 0;

# Method Declarations:
Methods
   int init(Object controlUnit)
   boolean isAlive()
   int registerEvent(Event e)
   int unRegisterEvent(int eventHandle)
   int waitEvent(Event e)
   int final()
End
```

7.3.3 Activity interface

Living in a computer, virtual pets can do unlimited number of tasks, which range from monitoring computing resources to assisting human users. It may not be predictable what a virtual pet can do. Therefore, the interface design has to be very flexible. The basic methods are: `basedOn`, grow, `actionMap, locationMap`.

7.3.3.1 BasedOn

This method takes in the material that makes up a virtual pet. The material does not have to be visible. However, it should be measurable so that the virtual pet can grow or shrink. The material can be any thing new and unpredictable. For simplicity, this method takes only one material.

If there are demands for multiple materials in a virtual pet, the method `locationMap` can be used.

7.3.3.2 Grow

This method bases on the relative mass of the material that makes up a virtual pet. It is the most basic activity of a virtual pet. Some data structures can be designed to present status of the material mass so that it can be monitored.

Other activities of virtual pets may be presented using the `actionMap` method.

7.3.3.3 ActionMap

This method provides ways of referencing an action in an object or component. New activities for virtual pets can be implemented this way without knowing what the activities are when the Activity interface is written. Therefore, the Activity interface is not modified every time a new activity is discovered in some virtual pet.

Since the object that does the activity mapping is a parameter to this method, different object that contains different sets of activities can be referred to. `ActionMap` can be a separate interface known by this Activity interface.

7.3.3.4 LocationMap

In general, a virtual pet should be at a location or position at a given time so that it can be kept tracks of or monitored. However, the locations in a computer either logical or physical are very diverse. Two basic concerns are the position and the unit of measurement.

Similarly as the `actionMap`, `locationMap` can be a separate interface known by this Activity interface.

7.3.3.5 Requirements

Following is the text file that contains the requirements for the Activity interface.

```
# This is a interface-requirement documentation
# Interface Name: Activity

# Data Fields:
errno = 0;

# Method Declarations:
```

```
Methods

   int basedOn(Object handle)
   int grow()
   Object actionMap(String action, Object handle)
   Object locationMap(Object locations)
End
```

7.3.4 Control interface

As described in Section 7.3.1, LER can be used as a control unit for the virtual pet. For more flexibility, however, the Control interface should allow multiple control units. Some virtual pet design may also require replacing, adding, or removing control units at run time. The methods are: `inCharge`, `add`, `remove`, `listControlUnits`, `searchMethod`.

7.3.4.1 InCharge

This method sets the primary control unit for this virtual pet as it receives the object from the parameter. Thereafter, the control unit's methods are made available for controlling the virtual pet.

7.3.4.2 Add

This method adds a new control unit into the list. For simplicity, the priority of control units is in the order when they are added. Priority is necessary for searching for a particular controlling method when multiple control units are present. See the method named `searchMethod`.

7.3.4.3 Remove

The specified control unit will be removed from the list of control units. This method can also be used for replacing a control unit or changing its priority.

To replace a control unit, the new control unit is added first. Then, the current control unit is removed.

In case of multiple control units present, the priority of a control unit can be changed with a two-step procedure: removing it then adding it back at the end of the list.

7.3.4.4 ListControlUnits

This method simply lists the control units for this virtual pet.

7.3.4.5 SearchMethod

As discuss earlier, a virtual pet can have multiple control units. Each control unit may be a component that implements a different control interface. However, there are no restrictions on using components that implement the same or similar interface. Therefore, this method searches the control units in the order of the list for a controlling method. The first method found will be returned.

7.3.4.6 Requirements

Following is the text file that contains the requirements for the Control interface.

```
# This is a interface-requirement documentation
# Interface Name: Control

# Data Fields:
controlUnits = aRef;
errno = 0;

# Method Declarations:
Methods
   Object inCharge(Object handle)
   int add(Object handle)
   int remove(Object handle)
   int listControlUnits()
   Method searchMethod(String method)
End
```

7.4 Virtual Pet Design and Integration

Once the interfaces are all decided and the components are available, designing the virtual pet is a matter of integrating the required interfaces and using the appropriate components.

In discussions of the Pet Interfaces, it occurs again and again that new components or methods may be added later after the virtual pet application is completed. Even with Java Interface and polymorphism, the code that instantiates a component of an interface is required to be modified:

```
Control petBrain = new ControlAble();
```

That statement is going to be replaced by a call to the LER, which loads a component implementing the Control interface:

```
Control petBrain = LER.loadComponent("Control");
```

Although different versions of `loadComponent()` exist for specifying various components, it can be assumed that the local machine has the only one particular component for this `Control` interface. Following is the simple example program for integrating the above interfaces into a virtual pet application.

```
Control petControl = LER.loadComponent("Control");
Existence myPet = LER.loadComponent("Existence");
myPet.init(petControl);
Activity petLive = LER.loadComponent("Activity");
petLive.basedOn(LER.root);
petLive.grow();
waitEvent(myPet.EXIT);
```

With the use of LER, the `Control` interface can be mapped to a local component, which can come with the same package as the virtual pet application. Or, it can be an existing component that has been installed by other application. Later, if a new control unit

component is available, it can replace the current component. The `LER` interface also provides ways for choosing a component at run time.

The control object is passed into the initialization method for marking the existence of this virtual pet. This particular pet is supposed to grow based on the file system at the root of LER. The `grow()` method builds some data structures to keep track of the interfaces under LER ROOT. Notice that the grow() method runs in an infinite loop in the background. It does not block other threads of executions.

Similarly, the `waitEvent` does not block to wait for the EXIT event. Notice that after `myPet` is instantiated, LER keeps a reference of this object. A separately written program can access this object if the two applications are running on the same machine in the same time period. For example, the second application may want to add a second control unit to `myPet` to remotely interact with this virtual pet. It may want to register, un-register, or wait for events that happen to this virtual pet. Following is a sample code segment.

```
RemoteControl remote = LER.loadComponent("RemoteControl");
Control petControl = LER.loadComponent("Control");
Existence myPet = LER.loadComponent("Existence");
petControl.add(remote);
myPet.register(MEALS);
```

7.5 Virtual Pet Delivery

After the application is complete, all the required components are packaged into certain package format for delivery of the virtual pet. Package delivery is a crucial step to ensure component reuse and efficiency.

Unless all of the required interfaces are newly designed and implemented, some of the components are already existed in existing systems. The package planning and delivery work involves with the LER facility.

For example, when the application integrator puts together a virtual pet package, it may include these components:

```
LER                  ; default LER component
VirtualPet           ; the integrated application
SimControl           ; control unit component
RemoteControl        ; remote control unit component
LiveForever          ; existence component
FunActivity          ; activity component
```

The packaging tool considers the facts that not all of those components will have to be installed on the destination system. A LER may have already existed. Some of the components may also be used in use on the system. It is likely that the package will not install its own components but let the virtual pet share or reuse the existing components. However, other considerations arise for deciding the compatibility of the components. Our approach is to take those considerations out of the application and let the packaging tools and the implementation of LER provide the solutions, from a local environment point of views. It is the use of local components that decide the characteristics of the virtual pet.

7.6 Conclusion

In this virtual pet example application, we show the important aspects of mapping an interface to a different component at run time. Even though the basic interfaces of a virtual pet are clearly defined at design and implement time, each virtual pet is not tied up to a particular implementation of the components. In theory, each machine has different

attributes and existing components so that imposing unify and global environments for these virtual pets would not be practical.

The LER plays the role of a practical petting facility, which produces unique virtual pets based on the local system conditions. Uniqueness and diversity are some of the important characteristics of these virtual pets. We also show in the example that different applications can share access to the same object at run time. Our strategy of component reuse is based on local availability. Component makers and application integrators can make many varieties of components and applications that run on varieties of systems.

Package delivery is the key strategy. An integrated virtual pet application looks simple and has no particular checking for local conditions at the application integration level. That is due to the LER interface hiding the technical details of dynamically loading components at run time. LER interface abstraction gives us the advantages of mapping interfaces to particular components.

It is the packaging and delivering of the virtual pet to a particular system that creates a particular pet at the time it is run. If there are no two machines that are alike in these server farms, there are no two virtual pets that are alike.

Chapter 8
Future Work

Run-time interface mapping methodology has the premise for application integrator. No more ambiguity in choosing components for the applications. Components can be dynamically selected at run time based on the interface and local environment conditions. The application does not have to figure out which component to load. The local environment facility does the loading because it has the information about local components and environment attributes. Therefore, component programmers or application integrators are free of embedding much of the decision logic in their programs.

Component programmers who write components can use the interface requirements as their contract, which is demanded by application integrators or other component users. As long as the requirements are followed in the naming of interface, components, methods, and attributes, they can implement the internals of their components anyway they want to. This is the first principle in software components – they are developed for independent uses. Third party developers who want to deploy these components only need to know the component published interfaces.

The local environment facility is implemented as a simple common component, not a global framework or a heavy middleware. Either component makers or application integrators can start to work with components quickly with this approach. They can even implement their own local facility based on the design concepts. We keep this local

facility simple. But, it is flexible and it can be extended as we are going to discuss next in following paragraphs.

Mapping of interface methods:

in the current interface mapping method, component methods are called based on requirement specifications. If the interface is mapped to a component that does not have the corresponding method, the call fails with unpredictable consequence. That is a calculated risk, which can be minimal by thorough packaging and delivering of components. Verifications of components at adding time can also help. We discuss about that later.

However, the risks can be reduced via mapping the interface methods into component methods. The idea is similar to mapping an interface to multiple components. However, the reference entered in the program is a method name. This way, it is possible that a method can be mapped into multiple interfaces. The same method name can exist in different interfaces. Therefore, if a method call fails in one interface, the local environment facility can extend the search of similar method in other interfaces.

Interface verifications:

As we discuss above, when a component is added into an existing interface in the local facility, there is currently no checking to ensure the component has fulfilled all the required methods in the interface. Because it is not required that the component has to be recompiled and linked into the application, problems of unknown methods and parameters type mismatch only appear at run time. These are typical problems in using pre-built components.

Because adding a component into the local environment facility is a separate step that is carried out before the component is used, different procedures can be added to verify the component with the interface. This can be done with Java language. A list of all methods from the component can be obtained and then compared to the methods in the referenced interface. Or, it could be as simple as trying to load the component and call each of the methods in the interface. Errors can be discovered at adding time. And the component can be rejected.

Component and interface versioning:

Software versioning is not new in development environments. And components are no exception. It is inevitable that component makers will have to make changes or enhancements to their components over times. We discuss the possibility of this in our component naming and packaging methods. Actually the version number of a component plays a part in the component name. Once a component has more than one version, version would become an attribute for mapping the current version of a component to the interface.

We should also consider versioning an interface. Although the application integrator or whoever creates or designs a interface have surely put lots of considerations on all thinkable requirements, needs for a certain interface do change from time to time. Creating a new interface just to add a method to an existing interface is not something component makers want to do often. The number of interfaces should be kept to minimal. More research is required on how this should be resolved.

Component testing and analysis tools:

The scope of this research does not allow elaborate and thorough testing of the implementations. However, because it is simple to design a new interface and adding components into it, this approach encourages better testing of the components. We see testing as a very important requirement in adapting component reuse. Testing in development environments is not sufficient anymore because of the diversity of run time environments.

Second to testing tools are run-time analysis tools. The application integrator or programmer who deploys components does not have the same level of understanding of the component internals as the component maker. Besides, run-time errors are always the hardest ones to resolve. With this approach, we have the luxury of replacing a problematic component with a similar one. We also have the ability of isolating the problem into small component. More work in this area would be very worthwhile.

Software packaging and local environment management tools:

The method of run-time interface mapping will prove its values when more helpful tools are built for the purposes of applying the concepts. Software packaging tools can automate most of the procedures we describe in this thesis. Ultimately, applications can integrate themselves using available local components and the supports of the local facility. Local environment management tools can present the local facility to the user in easy to understand and use graphical interfaces. Component makers and application integrators are only intermediate users. The ultimate users are the end users who actually use the applications.

It is Only a Beginning…

This thesis not only has practical values in programming and application integration, it can also be applied to emerging computing technology. Three possible areas are GRID Computing, Pervasive Computing, and Computing Utilities.

GRID Computing:

We approach GRID Computing from the bottom up strategy. Although GRID Computing stems from and thrives on distributed computing, we see its core problems in local environments. On hundreds or thousands of computers that make up the network grid, it is difficult to blanket out a cross-platform strategy for utilizing each of the computer. It is plainly impractical. It is even harder to keep track of all local information at a central controlling point that becomes a bottleneck or a single failure point. Information about local environment of each machine is crucial in many aspects of GRID Computing, for example job scheduling and dispatching. Our approach is for delivering applications and components in such environments.

Pervasive Computing:

Computers will be everywhere in the age of Pervasive Computing. Furthermore, computers will transform into different shapes and forms. The typical form of a desktop computer with a square box, a monitor, a keyboard, and a mouse is only one of the many forms of the future computers. Even at the time of this writing, we are seeing more and more of non-conventional computers, such as wearable, jewelry, head-mounted computers, etc. Each of these devices has different characteristics in terms of computing environments. It will not be practical to say that an application can be written on one of the devices and then run on all of other devices. Reuse of software has to happen at finer grains, such as components. Because most of these devices have to take special forms or

hide in odd places, they may not have the most powerful processing power or abundant of storage space. Applications running on these devices have to be very efficient to the points that no extraneous and unused resources are allowed. Our methodology will adapt and use just the minimal components to run the tasks.

Computing Utilities:

Will computing resources be ever delivered like utilities such as electricity or water? And what is the computing “stuff” that will be delivered over the wires? That is just the pure computing resources such as processing power, storage space, or network bandwidth. The home users still need different types of electric or electronic devices to make use of electricity. Similarly to water, home users need to add things into water to make lemonade, tea, coffee, or bubble bath. All those devices and ingredients add value to the commodity electricity or water. An end user cannot make those by themselves. That is where we see the roles of software components. We also do not overlook the facts that personal computing power is increasing in amazing pace. Each individual home user can have her own power generator or water well. A local machine can easily become a source of computing utilities. This matches with the GRID Computing perspectives.

In our views, a local machine is an autonomous computing entity. It only looks else where for other resources when it needs them.

References

[1] P. Allen; *Realizing e-Business with Components*; Component Software Series; Addison-Wesley, 2001.

[2] P. Allen; "The State of the Practice", *Component Development Strategies Monthly Newsletter;* vol. XI, no. 3, March 2001.

[3] P. Allen; "Component Specification and Assessment", *Component Development Strategies Monthly Newsletter;* vol. XI, no. 10, October 2001.

[4] S. Asbury and S. R. Weiner; *Developing Java™ Enterprise Applications 2nd Edition;* John Wiley & Sons Inc., 2001.

[5] E. C. Bailey; *Maximum RPM, Taking the Red Hat Package Manager to the Limit*; Website, http://www.rpm.org/max-rpm/index.html; Red Hat Inc., 2000.

[6] B. Baker; *Getting Started with InstallShield Developer and Windows Installer Setups*; InstallShield Press, 2002.

[7] B. Baker; *The Official InstallShield for Windows Installer Developer's Guide;* Hungry M&T Books, 2000

[8] B. Ball et al.; *Red Hat Linux 7.2 Unleashed*; Sams, 2002.

[9] D. Barnes; "RMP How to, RPM at Idle"; Website, http://www.rpm.org/RPM-HOWTO/; Red Hat, Inc, 1999.

[10] R. Ben-Natan and O. Sasson; *IBM SanFrancisco Developer's Guide;* Application Development Series; McGraw-Hill, 2000.

[11] H. Beyer and K. Holtzblatt; *Contextual Design: A Customer-Centered Approach to System Designs;* Morgan Kaufmann, 1997.

[12] S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, and B. Stearns; *The J2EE™ Tutorial*, The Java Series; Addison-Wesley, 2002.

[13] Bonzi BUDDY; "Bonzi.com"; Website, http://www.bonzi.com/bonziportal/index.asp; Bonzi.com Software, Inc., 2002.

[14] J. Bosch; *Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach;* Addison-Wesley, 2000.

[15] D. R. Callaway; *Inside Servlets 2nd Edition, Server-Side Programming for the Java™ Platform;* Addison-Wesley, 2001.

[16] X. Castellani and S. Y. Liao; "Development Process for the Creation and Reuse of Object-Oriented Generic Applications and Components", *Journal of Object-Oriented Programming (JOOP)*; vol. 11, no. 3, pp. 21-34, June 1998.

[17] E. Cerami; *Web Services Essentials*; O'Reilly & Associates, Inc., 2002.

[18] J. W. Cooper; *Java™ Design Patterns, A Tutorial*; Addison-Wesley, 2000.

[19] Creature Labs; "Virtual Creatures"; Website, http://www.creaturelabs.com/tech_index.htm; Creature Labs, 2002.

[20] L. G. DeMichiel, L. U. Yalcinalp, and S. Krishnan; *Enterprise JavaBeans™ Specification, Version 2*; Java Website, http://java.sun.com/ejb; Sun Microsystems, Palo Alto, 2001.

[21] J. Durham; "History-Making Components, Tracing the roots of components from OOP through WebServices", *IBM DeveloperWorks Website,* http://www-106.ibm.com/developerworks/components/library/co-tmline/index.html; 2001.
[22] B. Eckel; *Thinking in Java™*; Prentice Hall Inc., New Jersey, 1998.
[23] J. Engel; *Programming for the Java™ Virtual Machine;* Addison Wesley Longman, Inc., Reading, Massachussetts, 1999.
[24] M. E. Fayad, D. C. Schmidt, and R. E. Johnson; *Building Application Frameworks, Object-Oriented Foundations of Framework Design*; John Wiley & Sons, 1999.
[25] FlashLine Website; "Component Marketplaces"; http://www.flashline.com/; FlashLine Inc., 1999-2002.
[26] K. A. Gabrick and D. B. Weiss; *J2EE and XML Development;* Manning, 2002.
[27] M. Galkovsky; "DLLs the Dynamic Way", *MSDN Library Website*, http://msdn.microsoft.com/library/en-us/dndllpro/html/dlldynamic.asp?frame=true**;** Pervasive Software, Nov. 1999.
[28] D. Garlan, R. Allen, and J. Ockerbloom; "Architectural Mismatch: Why Reuse Is So Hard?", *IEEE Software*; vol. 12, no. 6, pp. 17-26, Nov. 1995.
[29] K. Geihs; "Middleware Challenges Ahead," *IEEE Computer*, vol. 34, no. 6, pp. 24-31, June 2001.
[30] G. R. Gircys; *Understanding and Using COFF*; O'Reilly, November 1988.
[31] A. Gomez-Perez and A. Lozano; "Impact of Software Components Characteristics Above Decision-Making Factors"; Website: www.sei.cmu.edu/cbs/cbse2000/papers/05/05.pdf; Carnegie Mellon University.
[32] J. Gosling, B. Joy, G. Steele, and G. Bracha; *The Java Language Specification, 2nd Edition*; Sun Microsystems Website, http://java.sun.com/docs/books/jls/; Sun Microsystems, 2000.
[33] J. Han; "Characterization of Components; Peninsula School of Computing and Information Technology", Monash University, McMahons Road, Frankston, Vic. 3199, Australia; Website, http://www.sei.cmu.edu/cbs/icse98/papers/p4.html; Carnegie Mellon University, 2001.
[34] G. T. Heineman and W. T. Councill, Editors; *Component-Based Software Engineering, Putting the Pieces Together*; Addison-Wesley, 2001.
[35] J. Hunter; *Java™ Servlet Programming 2nd Edition*; O'Reilly & Associates Inc., 1998, 2001.
[36] InstallShield; "An Introduction to Windows Installer and InstallShield Developer White Paper"; Website, http://www.installshield.com/downloads/isd/whitepapers/Developer_Overview9-01.doc; InstallShield, Sept. 2001.
[37] R. Johnson; "Dynamic Object Model", *Object View*; no. 5, pp. 30-34, 2000.
[38] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and M. Sakkinen; "Bottom-up Design of Active Object-Oriented Databases," *Communications of the ACM*; vol. 44, no. 4, April 2001.
[39] R. Keller, and U. Holzle; "Binary Component Adaptation", *University of California Website*, http://www.cs.ucsb.edu/labs/oocsb/papers/TRCS97-20.html; ECOOP, 1998.

[40] M. Koutlis, P. Kourouniotis, K. Kyrimis, and N. Renieri; Inter-component communication as a vehicle towards end-user modeling; Computer Technology Institute, Patras, Greece; Website, http://www.sei.cmu.edu/activities/cbs/icse98/papers/p7.html; Carnegie Mellon University, 1998.
[41] P. Kriens; "A perspective on interfaces, How to solve problems with Java's interfaces"; Website, http://www.javaworld.com/javaworld/jw-02-1998/jw-02-perspectives_p.htm; Java World, 1998.
[42] R. Lee and S. Seligman; *JNDI API Tutorial and Reference Building Directory-Enabled Java Applications;* The Java Series; Sun Microsystems, 2000.
[43] A. Lessa; *Python Developer's Handbook;* Sams Publishing, 2001.
[44] S. Liang; *The Java™ Native Interface Programmer's Guide and Specification;* Website, http://java.sun.com/docs/books/jni/html/titlepage.html; Sun Microsystems, 1999, 2002.
[45] T. Lindholm and F. Yellin; *The Java™ Virtual Machine Specification, 2nd Edition;* Addison-Wesley, 1999.
[46] F. Marinescu; "The State of The J2EE Application Server Market, History, important trends and predictions", The Server Side Website, http://www.theserverside.com/resources/middleware/StateOfTheServerSide.html; The Middleware Company, 2001.
[47] V. Matena and B. Stearns; *Applying Enterprise JavaBeans, Component-Based Development for the J2EE Platform*; The Java Series Enterprise Edition; Boston, Addison-Wesley, 2001.
[48] D. Mennie and B. Pagurek; "An Architecture to Support Dynamic Composition of Service Components"; Systems and Computer Engineering, Carleton University, 1125 Colonel By Driver, Ottawa, ON, Canada, K1S 5B6; Website, http://citeseer.nj.nec.com/mennie00architecture.html; NEC Research Institute, 1997-2002.
[49] Microsoft Corporation; "Microsoft Portable Executable and Common Object File Format Specification", Revision 6.0; Website, http://www.microsoft.com/hwdev/hardware/PECOFF.asp; Microsoft Corporation, February 1999.
[50] T. Miyoshi and M. Azuma; "An empirical study of evaluating software development environment quality," *IEEE Transactions on Software Engineer*, vol. 19, no. 5, pp. 425-435, May 1993.
[51] R. Monson-Haefel; *Enterprise JavaBeans, 2nd Edition*; O'Reilly & Associates Inc., 1999, 2000.
[52] G. C. Murphy, D. Notkin, and K. J. Sullivan; "Software reflexion models: bridging the gap between design and implementation", *IEEE Transactions on Software Engineer*; vol. 27, no. 4, pp. 364 – 380, April 2001.
[53] A. Nathan; *.NET and COM, The Complete Interoperability Guide*; Indiana, Sams Publishing, 2002.
[54] T. Neward; *Server-Based Java Programming*; Manning Publications, 2000.
[55] Object Management Group; *Common Object Request Broker: Architecture and Specification,* Revision 2.6.1; OMG Website,

http://cgi.omg.org/docs/formal/02-05-08.pdf; Object Management Group, Inc., 1991-2001.

[56] Object Management Group; Catalog of OMG CORBA™ / IIOP™ Specifications; OMG Website, http://www.omg.org/technology/documents/corba_spec_catalog.htm#ccm**;** Object Management Group, Inc, 1997-2002.

[57] Object Management Group; Catalog of OMG IDL / Language Mappings Specifications, *OMG Website*, http://www.omg.org/technology/documents/idl2x_spec_catalog.htm**;** Object Management Group, Inc, 1997-2002.

[58] Open Component Foundation; "Enhancing Local Software Industry's Competitiveness"; Website, http://www.open-components.com/; Centre for Innovation and Technology, Faculty of Engineering, The Chinese University of Hong Kong, 2002.

[59] R. Orfali and D. Harkey; *Client/Server Programming With Java and CORBA, 2nd Edition*; John Wiley & Son, Inc., 1998.

[60] P. J. Perrone and V. S. R. "Krishna" R. Chaganti; *Building Java Enterprise Systems with J2EE, The Authoritative Solution*; SAM Publishing, 2000.

[61] J. R. Pinkert and L. L. Wear; *Operating Systems Concepts, Policies, and Mechanisms*; Prentice Hall, Inc., New Jersey, 1989.

[62] Polson Enterprises Research Services; Virtual Pet Patents; Web site: http://www.virtualpet.com/vp/media/mpets/vppat.htm; Polson Enterprises Research Services, 2002.

[63] RJB Productions, Virtual Pet Creature, Website, http://www.rjbproductions.co.uk/vpc/about.htm; RJB Productions, 2002.

[64] S. Robinson and A. Krassel; "COMponents"; MSDN Website, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_components.asp; Panther Software, http://www.panthersoft.com/, 1997.

[65] R. C. Seacord, Hissam, A. Scott, K. C. Wallnau; Agora: "A Search Engine for Software Components", *Technical Report*; Carnegie Mellon University, 1998.

[66] L. Sha; "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20-28, July/Aug. 2001.

[67] H. Sheil; "Java Project Dangers! Avoid these 10 J2EE dangers to ensure your enterprise Java project's success", *JavaWorld Website*, http://www.javaworld.com/javaworld/jw-03-2001/jw-0330-ten_p.html; JavaWorld.com, March 2001.

[68] H. Sheil; "To EJB, or not to EJB? Addressing the issues and decisions that go into adopting an EJB-based solution", *JavaWorld Website*, http://www.javaworld.com/javaworld/jw-12-2001/jw-1207-yesnoejb_p.html; JavaWorld.com, Dec. 2001.

[69] H. Sheil and M. Monteiro; "Rumble in the Jungle: J2EE vs. .NET, Part 1, How do J2EE and Microsoft's .Net compare in enterprise environments?", *JavaWorld Website,* http://www.javaworld.com/javaworld/jw-06-2002/jw-0628-j2eevsnet_p.html#resources; JavaWorld.com, June 2002.

[70] J. Siegel, PhD and others, *CORBA 3 Fundamentals and Programming*, 2nd Edition, New York, Wiley Computer Publishing, 2000.
[71] I. Singh, B. Stearns, M. Johnson, and the Enterprise Team; *Enterprise Applications with the J2EE Platform, 2nd Edition*, The Java Series; Addison-Wesley, 2002.
[72] Software Engineering Institute; "Annotated Bibliography of COTS Software Evaluation"; Website, http://www.sei.cmu.edu/cbs/papers/eval_bib.html; Carnegie Mellon University, 1998-1999.
[73] M. Sparling; "Lessons learned: through six years of component-based development", *Communications of the ACM*, vol. 43, no. 10, pp. 47-53, 2000.
[74] M. Sparling; "Is there a Market for Components?" Website, http://www.cbd-hq.com/articles/2000/000606ms_cmarket.asp; Castek Software Factory, 2000.
[75] F. Steimann; "Role = Interface: A Merger of Concepts", *Journal of Object-Oriented Programming, JOOP Website*, http://www.joopmag.com; pp. 23-32, Oct/Nov 2001.
[76] Sun Microsystems; Accessing the Environment From Java Applications; Website, http://developer.java.sun.com/developer/JDCTechTips/2001/tt1204.html#tip1; Sun Microsystems, 2001.
[77] Sun Microsystems; Guide to Features – Java Platform; Website: http://java.sun.com/products/jdk/1.2/docs/; Sun Microsystems, 1995-2002.
[78] Sun Microsystems; *Java™ 2 Platform, Enterprise Edition (J2EE™) Specification,* Version 1.3; Sun Microsystems, 2001.
[79] C. Szyperski; *Component Software, Beyond Object-Oriented Programming;* Addison-Wesley, 1997.
[80] C. Szyperski; "Component Software and the Way Ahead", *Foundations of Component-Based Systems*; Edited by Leavens, G. T. and Sitaraman, M; Cambridge, Cambridge University Press, 2000, pp. 1-20.
[81] J. Tian and M. V. Zelkowitz; "Complexity measure evaluation and selection"; *IEEE Transactions on Software Engineer*, vol. 21, issue 8, pp. 641-650, Aug. 1995.
[82] B. Venners; "Designing with interfaces, One programmer's struggle to understand the interface"; Website, http://www.javaworld.com/javaworld/jw-12-1998/jw-12-techniques_p.htm; Java World, 1998.
[83] B. Venners; *Inside the Java 2 Virtual Machine, 2nd Edition*; McGraw-Hill, 1999.
[84] A. Vogel and M. Rangarao; *Programming with Enterprise JavaBeans, JTS, and OTS; Building Distributed Transactions with Java and C++;* Wiley & Sons, 1999.
[85] K. C. Wallnau, S. A. Hissam, and R. C. Seacord; *Building Systems from Commercial Components;* SEI Series in Software Engineering; Addison-Wesley, 2002.
[86] G. Wang and H. A. MacLean; "Architectural Components and Object-Oriented Implementations"; Applied Research and Technology, The Boeing Company, Seattle, WA 98124; Website, http://www.sei.cmu.edu/cbs/icse98/papers/p9.html; Carnegie Mellon University, 2001.

[87] G. Wang and H. A. MacLean; "Software Components in Contexts and Service Negotiations"; Applied Research and Technology, The Boeing Company, Seattle, WA 98124; Website, http://www.sei.cmu.edu/cbs/icse99/papers/23/23.htm; Carnegie Mellon University, 2001.
[88] J. A. Whittaker; "Software's Invisible User," *IEEE Software*, vol. 18, pp. 84-88, 2001.
[89] J. D. Williams; "Can component solve integration conundrum?", *Application Development Trends*, vol. 8, pp. 27-34, 2001.
[90] S. Williams and C. Kindel; "The Component Object Model: A Technical Overview"; Developers Relation Group, Microsoft Corporation, MSND Website: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_compp r.asp; Microsoft Corporation, 1994.
[91] P. Wisse; *Metapattern, Context and Time in Information Models*; Addison-Wesley, 2001.
[92] C. Wohlin and P. Runeson; "Certification of software components", *IEEE Transaction on Software Engineering*, vol. 20, pp. 494-499, 1994.
[93] S. Yacoub, H. Ammar, and A. Mili; "A Model for Classifying Component Interfaces"; CSEE Department, West Virginia University, Morgantown, WV 26506; Website, http://www.sei.cmu.edu/cbs/icse99/papers/31/31.pdf; Carnegie Mellon University, 2001.
[94] S. Yacoub, H. Ammar, and A. Mili; "Characterizing a Software Component; CSEE Department", West Virginia University, Morgantown, WV 26506; Website: http://www.sei.cmu.edu/cbs/icse99/papers/34/34.htm; Carnegie Mellon University, 2001.
[95] J. W. Yoder; "MetaData and Adaptive Object-Model Pages," presented at ECOOP, 1998.
[96] S. H. Zweben, B. W. Weide, and J. E. Hollingsworth; "The effects of layering and encapsulation on software development cost and quality", *Software Engineering, IEEE Transactions on*, vol. 21, pp. 200 - 208, 1995.