ICSM'02. IEEE press, pp. 131-140. 2002

Using Application Server To Support Online Evolution

Qianxiang Wang, Feng Chen, Hong Mei, Fuqing Yang
Department of Computer Science and Technology,
Peking University
100871 P.R.C.
{wqx,cf}@cs.pku.edu.cn, meih@pku.edu.cn,

Abstract

The online evolution of application depends highly on its runtime environment. This paper addresses how to support applications that execute on application servers that compose of component containers and common services. From the requirements' viewpoint, evolution of software can be divided into four categories: evolution that does not alter requirements, evolution that alters functional requirements, evolution that alters local constrain requirements, and evolution that alters global constrain requirements. All changes at the requirements level can be mapped to changes at the implementation level. Using our approach, implementation level entities, components and interceptors, are responsible for online evolution. Evolution includes adding, removing, updating, and reconfiguring the entities. One key to our approach is to carefully distinguish states of component and interceptor, e.g., ready, active, executing and evolving. A well-designed architecture and feasible mechanisms for runtime instance loading are also keys to the solution. Based on this approach, an application server prototype, named PKUAS, has been implemented and is introduced in the end.

1. Introduction

Online evolution was firstly mentioned in some safety-critical areas, such as avionics, telecommunication, etc..[1] The first goal of online evolution is to keep systems running even when problems occur. Today, as software is changing from "product-centric" to "service-centric", more and more commercial software also requires online evolution feature. The reasons may include: most services should keep working 24 hours a day, and 7 days a week; both new technologies and consumers' prejudices change very quickly today. As a result, most newly developed software should have the ability of online evolution so as to erase bugs, add new services, enhance system reliability and security, etc.

Traditional online software evolution focuses on reliability of systems, by way of redundancy, degradation, etc [1]. Today, flexibility, extensibility and ease of replacement become the most important factors that motivate online evolution.

By implementing replaceable components, flexible software can be easily customized according to user's preference. Some design patterns give excellent solutions to online replacement [22]. But there is less research on other features such as extensibility. In addition to common problems that all software evolution must face, e.g., architecture change [2], feature interaction [9][21], online evolution suffers from the following two problems:

First, week management of evolving blocks prevents practical online evolution. Although modules and classes are good construction blocks for applications during development, they are poorly maintained at runtime. Processes, which are generated from those blocks, are the units well supported by traditional runtime environment such as the OS, are still hard to modify. There have been some mechanisms for dynamic module loading, e.g. DDLs, but it is still hard for online evolution without other mechanisms such as request buffering. The emergence of some component models, such as EJB [13] and CCM [24], provide the ability to keep the components isolated at runtime, by way of giving each component a standalone management mechanism, such as a component container. Based on that, component frameworks (such as application servers) make it possible to alter components when the application is running. The approach proposed in this paper is based on component containers.

Second, crossing cutting features prevents effective online evolution. Functional requirement has been well supported by structured and object-oriented development methods. But neither method supports constraints well. Implementation of constraints usually involves multiple modules or

classes. This phenomenon is called crossing cutting problem [8]. When evolving constraints, the code to be changed is often scattered throughout the whole application. This situation not only led the evolution to be laborious and time-consuming, but also will result in the uncertainty of the evolution. So, while some research focuses on functional requirement evolution [2], other researchers think the evolution of constraints is much popular and important [6][7]. This paper solves this problem via interceptors.

From the view of implementation, software evolution can occur at the system and application levels. The former concerns evolution of system software, such as operating system, DBMS, and application server. In the paper, we only consider the later, the evolution of application software. Apparently, evolution of application software needs support of system software.

Application Servers are system software that has emerged with the development of computer network and provide the basis for component-based software development (CBSD). CBSD has been widely accepted recently. " By the year 2002, 70 percent of all new applications will be developed using component-based application building blocks (0.8 probability)" [10]. Supported by application servers, components are not only the units for development but also the entities in runtime, which greatly facilitates the runtime evolution of component-based applications.

Based on application servers, this paper presents a solution to online evolution of applications. An architecture of application servers that support the dynamic component loading and request buffering is proposed and the states of components are clearly defined according to the requirement of online evolution. Besides, in order to support the evolution of constraints, the Interceptor pattern is introduced. Interceptor is a successful design pattern that is widely used today [11][12]. Interceptors can locate in server side or client side and are used to filter and process messages that are transmitted between client and server. We use interceptors to provide common services such as transaction and security for component invocation and implement constraints of the applications. By altering the interceptors, the constraints can be evolved effectively.

This paper is organized as follows. Section 2 introduces some key concepts about application server, including application, component, container, component implementation, component instance, interceptor, and context. Mapping of requirement changes into implementations is also introduced. Section 3 describes the challenge and solution to component evolution. Section 4 describes the solution to interceptor evolution. Section 5 introduces PKUAS, an application server prototype that supports our approach for online evolution.

2. Application Server

Nearly all of today's application servers support component technology and provide common services to applications. As a result, developers are able to focus on system's business logic, and implementation of constraints is separated from component's implementation codes. Thus we can build up better-structured applications.

One typical structure of application server is illustrated in figure 1. In this structure, application servers involve concepts of application, component, common services, etc.

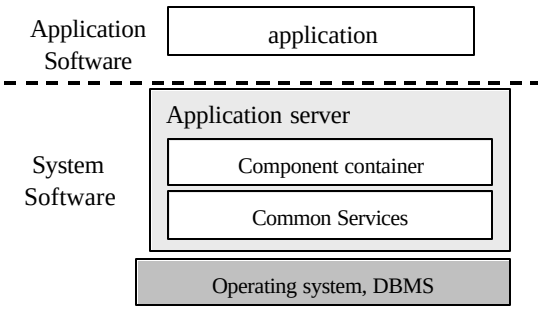

Figure 1 Structure of application server

Application

An application composes of multiple modules, such as procedures, classes, or components, which interact each other. Application software runs over some king of system software such as an OS.

Component

A component is an independent module that undertakes some specific functions in software systems. Although most people focus on component implementation, component specification is a necessary part of a component [14]. Component implementation is the code that implements the functions of component, and component specification describes how to assemble, use, and manage the component. The application that runs on an application server can contain multiple components. The relationship between components and applications is similar to that between employees and employers.

Container

Component containers are the main mechanism in application server to support component technology. Containers give components independent running environment, which makes development and

management of components more feasible and convenience. As middle layer locates between callees and callers, containers are responsible for connection to components, life cycle management of components, and coordination of transaction for components, maintenance of component persistency, identification authentication of clients, etc., while clients are unaware of existence of containers. These functions are implemented as common services, and thus developers can put their attention on the main functions of applications.

Common service

The concept of service becomes mature with the development of the client-server computing model. Belonging to the system level, common services are used to support the similar features of different applications. They are APIs for application developers provided by system software. Common services undertake most of application server's functions. For example, naming service is responsible for locating the server, transaction service is responsible for assuring system's consistence, and security service is responsible for authenticating the client and controlling the access to server, etc.

At application level, applications also provide services that are implemented by components. An application service includes service implementation and service description. Implements of components compose the implementation of service. And description of service includes not only the description of components, but also features that emerged when components are combined with their running environment including security, communication protocol, response time, system's throughput, etc.

3 Solution to online evolution

3.1 Evolution at requirement level

Software evolutions are motivated mainly by changes of requirement, which is still a primary problem in SE research. This paper concerns only those requirements that contribute to software, not those about hardware such as response time, system distribution, etc..

Oreizy distinguished two types of software changes [2]: (1) changes of system requirements, and (2) changes of system implementation that do not alter requirements. And Zave cataloged software requirements into " functions of" and " constraints on" software systems [3], in other words, functional requirements and constraint requirements. We further catalog constraint requirements into local constraint and global constraint.

For local constraints, we mean those that can be mapped into one or one group of function requirements, such as reliability, access control, auditing, etc. For global constraints, we mean those features that can only be incarnated by an integrated system, such as architecture of system, business process, etc. Such requirements can hardly be mapped to independent functions. Global constraints involve too much factors for online evolution and we do not take them into consideration.

The four types of software evolution can be illustrated by the example of an online pet store system [4]. Using this web-based application, users can browse pets, select pets into virtual shopping cart, and create orders for selected pets, etc.. When the application is evolved, we can see:

(1) Changes that do not alter requirements, e.g., to erase some bugs in shopping cart, or to improve algorithm of searching pets.

(2) Changes that alter functional requirements, e.g., to add electronic payment method to pay by credit cards online.

(3) Changes that alter local constraints, e.g., to improve security of order to guarantee that only authenticated users can create orders and only super user can browse all orders have been created. Some special requirements such as logging, auditing, exception handling, are considered as local constraint also. Those requirements cannot work alone, and can attached to functional requirements.

(4) Changes that alter global constraints, e.g., to reduce order process time, to improve system's throughput.

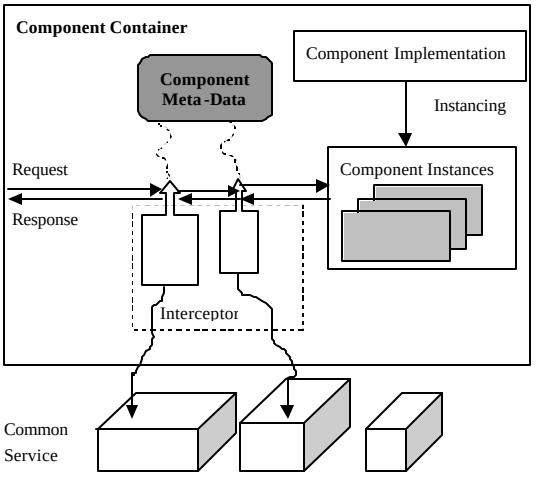

Figure 2 Structure of component container

3.2 Evolution in implementation level

To support online evolution, we proposed a special architecture of the component container that consists of component meta-data, component implementations, component instances, and interceptors, etc.. Figure 2 shows the architecture.

Component meta-data

Component meta-data (CMD) describes detailed knowledge about components in a formal form. Using component meta-data, component can be utilized and managed expediently. The content of component meta-data may includes following information of component: name, basic type, type interface, instance interface, implementation, running state, attribute of transaction, attribute of security, environment entries, etc. Figure 3 illustrates the component meta-data used in our application server, PKUAS. Component meta-data provides abundant information for reflective ability, which makes components much more manageable [15]. It provides the basis for online evolution.

```
public class CMD{
    protected String m_ejbName;
    protected byte m_beanType;
    protected Class m_typeInterface;
    protected Class m_instanceInterface;
    protected Class m_beanClass;
    protected HashMap m_beanMapping;
    protected HashMap m_methodMappin;
    protected HashMap m_methodPermissions;
    protected HashMap m_transactionAttributes;
    protected HashMap m_ejbRefMapping;
    protected HashMap m_environmentMapping;
    protected HashMap m_resourceMapping;
   ……
}
```

Figure 3 Implementation of CMD

Component implementation

Component implementation is the code implementing component's functions. For object-oriented systems, a component consists multiple closely coupled classes. Component implementations are usually stored in files to be delivered and assembled. After components are deployed as parts of applications, they are loaded into component containers, as figure 2 shows.

Component instance

The component instance is a runtime concept like the object in object-oriented system, while component implementation is a concept that exists in both developing time and runtime like the class in object-oriented system. Containers can create multiple component instances from one implementation. Instances may have their own attributes, such as identification, some data and running states.

Interceptor

As a widely adopted design pattern, interceptor is used to intercept and process messages between client and server. Interceptors process messages according to interceptor's configured policy data. In the interceptor pattern, messages are processed and forwarded to their destinations step by step. Interceptor is firstly used by OMG to implement security service such as to authenticate client. But developers soon find that it can be used for many other implementations of constrain requirements.

Interceptors can be thought of implementation level entities that are mapped from constraint requirement. In fact, we divide constraints from the viewpoint of interceptor: if a constraint can be implemented by some interceptors, it can be deemed as a local constraint, or it is a global constraint. It is feasible to organizing requirements based on this rule because requirements are usually induced from users by developers. Moreover, this classification benefits the application evolution. Figure 4 shows the basic interface of interceptors used in PKUAS.

```
public interface Interceptor{
public void init(Container container);
public void beforeInvoke(Object objectId, cspiMsg
mi,Method m);
public void afterInvoke(Object objectId, cspiMsg
mi,Method m);
public void handleException(Object objectId, cspiMsg
mi, Method m,Exception ex);
}
```

Figure 4 Basic interface of interceptor

When deployers deploy application, containers are constructed and different kinds of interceptors are also built according to the application's assemble information. These interceptors are organized into interceptor queues shown in figure 2 to process messages to components according to their own policies.

The context is a concept closely related to the interceptor and is important for dynamic class updating [16]. The called method can be seemed as the extension of the caller method. In order to execute correctly, the called method should get information about caller method, which is provided by the parameters in the traditional programming model such as OO. But in the distributed environment, more information is needed, including caller's identification, caller's requirement for consistency, etc.. This information is stored in the context that is an additional part of the message sent from the client.

In interceptor pattern, context also brings the

required information for interceptors. When processing messages, the interceptor just extracts the interested information from context, and then sends this information to corresponding common service along with other information got from component meta-data. Common services operate and return result to interceptors to reconstruct and pass the messages to their successors. Figure 5 shows an implementation of the invocation of interceptors to illustrate this process. After the target entity processes the client request, the response will be processed through the interceptors queue in a reverse order, which is also shown in the Figure 5.

```
public abstract class Container implements ContainerMBean{
   ……
   invoked(Object objectId, cspiMsg mi)throws Exception{
      Iterator interceptors=m_CMD.getInterceptors();
      while (interceptors.hasNext()){
         interceptorMetaData=(InterceptorMetaData)interceptors.next();
          interceptor=interceptorMetaData.m_interceptor;
          interceptor.beforeInvoke(objectId,mi,targetMethod);
          invokedInterceptors.add(0,interceptorMetaData);
      }
      invoke(objectId,mi,targetMethod,isRemote);
      interceptors=invokedInterceptors.iterator();
      ….
      while (interceptors.hasNext()){
         interceptorMetaData=(InterceptorMetaData)interceptors.next();
          Interceptor=interceptorMetaData.m_interceptor;
          interceptors.remove();
          interceptor.afterInvoke(objectId,mi,targetMethod);
      }
……
}
```

Figure 5 Two process phases of interceptor queue

3.3 From requirement to implementation

From the above discussion, we can see that, for application-server-based applications, functional requirements can be mapped into components and local constraints can be mapped into common services with interceptors. Thus, evolutions at requirement level can be mapped into implementation level as follows:

(1) Evolutions that do not alter requirements. Such evolutions do not alter the system's functions and constraints, so components' interfaces and interceptors' policies need not change. We can erase bugs and improve algorithm by updating component implementations.

(2) Evolutions that alter function requirements. Such evolutions lead to adding and removing components, or altering component's interface to change component's functions. Although we can change the component's interface by deleting old component and then adding new component [2], it is ineffective in practice and we deal with the change in a more direct way.

(3) Evolutions that alter local constraints. Such evolutions do not change components, but common services and interceptors. The changes may include adding and removing interceptor, or altering interceptor's policy.

(4) Evolutions that alter global constraints. Some of global constraints can be changed by alter configuration of applications and common services. But they are usually beyond the scope of online evolution.

As figure 6 illustrates the mapping rules for changes in requirement.

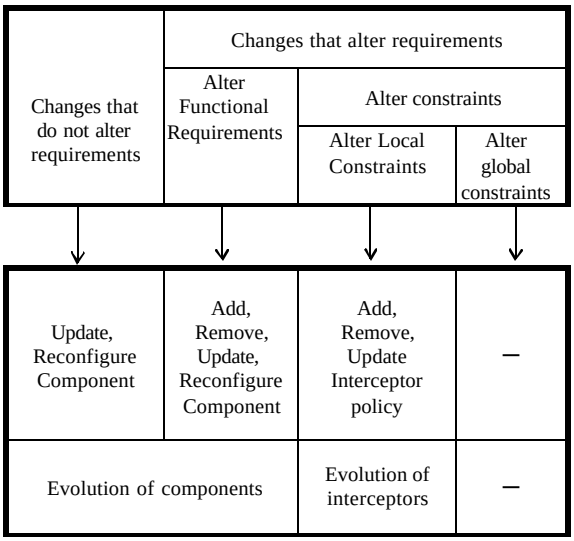

Figure 6 Mapping changes from requirement level to implementation level. Top half of figure lists changes at the requirement level. Bottom half of figure lists changes at the implementation level

4 Evolution demonstration

After mapping requirement changes into implementation, we introduce how to accomplish the desired online evolution at the implementation level in the following.

4.1 Evolution of Components

Evolution of components includes adding, removing, updating, and reconfiguring components. Removing and adding component are not difficult with the support of component container, and

dynamic loader. And reconfiguration of component is well discussed in [22]. The primary challenge to our approach is to update running components, which includes two kinds of updating: one does not alter component's interface, while the other does. The former is the focus in this paper and the latter is treat as adding and deleting components.(**和前面说的矛盾了**)

Challenge

Liang listed three difficulties of dynamically updating a class [17], which are also the difficulties of component updating:

(1) There may be live objects that are instances of a class we want to reload.

(2) How to map values of the static fields into different sets of static fields in the reloaded version of class.

(3) The application may be executing a method that belongs to a class we want to reload.

Solution

Our solution to online component updating is based on fine-grain component management. For a component-based application, system software monitors and controls the states of component when the application is executing. To better support online application evolution, besides the usual component states such as loading and running, we further separate the component's core state into three states: (1) State of ready, which means that the implementation of component has been loaded, but no instance has been created from it; (2) State of active, which means some instances have been created from the component, but no instance is executing. (3) State of executing, which means some instances are executing. Figure 7 illustrates the state diagram in our approach.

As figure 7 shows, we divide client requests into two classes, type requests and instance request, to control the behaviours of component more accurately. Type requests include creating, removing and searching instances and so on, which can be executed by any instance. Instance requests should be accomplished by some specified instances.

When an application is loaded, all its components are on the ready state. When the component receives creating request, the container creates instance for client to response client's other instance requests later. Then this component enters the active state. When the component is on ready state, things become a little more complex, which depend on client's request. If the component receives an instance request, it will enter the executing state until it returns response to client. Then it may return active state again, if no other instance if executing. If the component receives a type request such as just to look up some instances, the component will enter the executing state for a short time, and return active state soon. But if it receives type request of removing, the component may return to the ready state, if all of component's instances are removed.

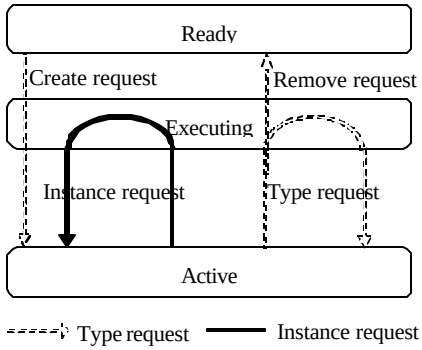

Figure 7 States related with online evolution

Besides these 3 states, we also introduce another special state: evolving state, which means that the component is under online evolution.

For each component to update, we should firstly change its state to " evolving" . If the component is in the evolving state, all incoming requests to it will be buffered in the container.

For a component in the ready state, we can update its implementation directly, after setting it's state to evolving. And after the new implementation has been loaded, the application server sets the component's state back to ready, and process the requests in the buffered queue.

For a component in the active state, we should update not only its implementation, but also its instances. So the whole evolution process is separated into two phases: (1) buffer every incoming request, update implementation of the component, and then process the type requests in buffered request queue by order; (2) buffer every incoming instance request until every instance of component is updated, but process type requests at the same time. However, only when attributes of the component keep unaltered, we can update instance of component directly by copying values of attributes. If not, we just keep these old instances until clients remove them. Another solution is just to delete old instances and inform clients to create a new instance again.

For a component in the executive state, we don't update executing instance immediately, because it is hard to keep the system in a safe situation. We just wait for the execution ends and the component returns to the active state or ready state. Another solution is to interrupt the execution and force the

component into the active or ready state.

4.2 Evolution of interceptors

Evolution of interceptor includes adding, removing, and policy updating of interceptors. Because most interceptors are created when applications are deployed into application server, adding and removing of interceptors are not one difficult problem, also with the support of component container. And online interceptor policy updating is also based on the fine-grain component management as figure 7 illustrated.

For a component in the ready state, when its state has been set to evolving, we can update the policy of its interceptors. After the policy has been updated, we can set component's state to ready again, and process requests in the buffered queue by order.

For a component in the active state, different kinds of interceptors should be treated differently. For some interceptors, such as log, authentication, etc., they concern only one operation, and the policy can be updated immediately. But for interceptors such as transaction that concern multiple operations, the policy can be updated after the whole process such a transaction is completed.

For a component in the executive state, it is dangerous to update policy when the component is executing too. So we have to wait the component returns back to active or ready state.

Recently, many approaches have been developed that are applied to support changes of constraint requirements. Truyen uses wrappers to solve above problems [7]. Aspect that proposed by [8] also can be utilized. Compared with these approaches, there are some benefits using the interceptor pattern.

Functional requirements and their constraints are bind later, until the application is deployed to application server. This improves flexibility of systems.

Constraints are supported by common services. The concentrate implementation of similar constraints for different functions makes constraints evolution more completely and consistently

Constraints are implemented with common services and interceptors together. This facilitates the dynamically updating constraints.

Developers of components do not need to care the existence of interceptors. Application assemblers and deployers can add constraints to components after components have been implemented. This greatly eases development of component.

Of cause, interceptor also has some limits. For example, in order to fulfill user's requirement well, application server should embed enough common services.

5 Implementation

PKUAS is an application server that implements our solution to online evolution, which is developed by the organization that authors work for. The whole structure of PKUAS can be divided into 3 layers, as figure 8 illustrated.

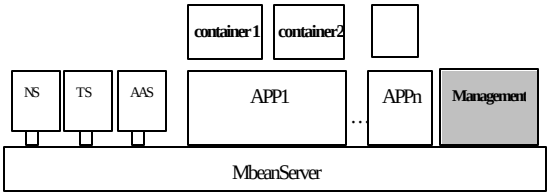

Figure 8 Structure of PKUAS

Containers locate at the top level of PKUAS. By resolving the jar package of application, and retrieving information of component from the deployment description files, PKUAS constructs a container for a component type. Then PKUAS loads the component's implementation, initializes the container, and creates interceptors for it.

Applications, common services and management modules are at the same layer, the middle layer. All of them should register to the bottom layer to make themselves manageable. The management module is responsible for deploying, starting, terminating applications and so on. PKUAS supports most popular common services, such as: Naming Service (NS), Transaction Service (TS), Authentication and Authorization Services (AAS), Presentation Services, etc.

PKUAS makes use of JMX[18], which provides most basic functions for flexible management of applications and common services.

Most component meta-data comes from descriptions file of components. In PKUAS, descriptions related with online evolution are stored in PKUAS.XML, which is generated in the assembling and deploying process. Figure 9 shows a example description of an "Order" component. Besides the ejb-name and jndi-name, it points out that it needs functions of "Account" component to complete its functions. Order's operation of "create" is a transactional method, with the transaction-related attribute of "Required". Clients that access to Order's operation of "getDetails" must be authenticated, and only those clients that have administrator role has right to access this method.

```
<pkuas>
  <ejb-name>TheOrder</ejb-name>
  <jndi-name>estore/order</jndi-name>
  <ejb-ref>
    <ejb-ref-name>ejb/account/Account</ejb-ref-name>
    <jndi-name>estore/account</jndi-name>
  </ejb-ref>
  <container-transaction>
    <method-name>create</method-name>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <method-permission>
        <role-name>administrator</role-name>
        <method>
          <method-name> getDetails </method-name>
        </method>
  </method-permission>
</pkuas >
```

Figure 9 Example of PKUAS.XML

6 Related work

Our approach concerns some other research work, especially Design patterns and Aspect that are discussed respectively below. And a similar work of Lasagne is compared at the last.

6.1 Design patterns

Some design patterns also provide flexibility of software system's behaviors, e.g. Decorator and Observer [22]. By applying proper patterns, developers can alter applications without modifying existing code. But design patterns primarily work in the design and development phases. After compilation, the flexibility provided by patterns is quite limited. For instance, developers can use Observer to change the handlers for some events at runtime, but the application's non-event-triggered functions cannot be altered.

Besides, the usage of design patterns is comparatively complex, especially when combining more than one pattern. How to appropriately choose and apply patterns in developing applications needs lots of knowledge about patterns and is a difficult problem even for an experienced developer. Moreover, when application is evolved, developers must be aware of used patterns, or the modification may lead to unexpected results. Our approach also adopts design patterns: Interceptor is a key to our approach. By embedding patterns in the application server, we achieve the needed flexibility of application while the developers can be totally unaware of the underlying mechanism.

6.2 Aspect

Study on Aspect is another important area we get inspiration. Aspect is a way to encapsulate and modularize crosscutting concerns that used to be scattered over the whole system, such as security, logging, etc.. Aspect-oriented programming (AOP) [8] and Aspect-oriented framework (AOF)[23] allow changes of application's behaviors by altering aspects of the application core in a non-invasive manner. This concept is very similar to our common service; in fact, we may call evolution of interceptors as evolution of aspects. But those approaches mainly operate at the class-level while our approach at instance-level.

Aspect-related approaches focus on the better development of software by modularizing the implementation of crosscutting features and weaving aspects and components at compilation or runtime. Changes of requirement will occurs interrupting and reweaving of the whole application. Our AS-based method is able to modify the weaving policy as well as the implementation of the aspect (common service) itself based on the Interceptor mechanism. Moreover, our approach can alter the components, which is beyond the capability of Aspect approaches.

6.3 Lasagne

Truyen presents a dynamic customization model Lasagne in [7]. Lasagne defines an architecture for dynamic and selective extensions of component-based applications using wrappers. By passing wrapper specifications through the collaboration process and properly programming wrappers, it supports a context-specific, non-invasive integration of system-wide extension on a per collaboration basis. It is not difficult to achieve online evolution of local constraints, or even some global constraints, based on Lasagne. Obviously, Lasagne does not support the updating of components, which is an important task in online evolution.

As a wrapper-based approach, Lasagne provides client-specific composition but cannot achieve call-site composition non-invasively. This limitation results in the difficulty of managing composition policies. Instead, our approach manages the integration in a centric manner based on the component's and service's specifications, which is much easier for managers to use.

In order to achieve the flexibility of composition, Lasagne requires developers to program the wrapper and wrapper specification carefully, which may be a

heavy burden when the system is very complex. Based on AS, our approach releases developers from the detailed work about common services and makes them put most attention on application's business logic.

7 Discussion

This paper mainly considers online evolution of applications that executed in the application server, with the support of component technology and interceptor technology. Component-based software development has been widely accepted: " By the year 2002, 70 percent of all new applications will be deployed using component-based application building blocks (0.8 probability)" [10]. The interceptor is a successful design pattern that being widely used today [11][12]. Interceptors can locate in server side or client side. In both situations, interceptors are used to filter and process messages that are transmitted between client and server.

The application server is considered as the third generation system software in network environment, succeeding operating system and database management system. Many application servers have been developed today, such as WebLogic, iPortal ApplicationServer, Websphere, and iPlanet, etc. Because most application servers support EJB specification [13], we use EJB as the main referenced component model in our approach and write all demonstrating codes in Java.

Evolution of software is becoming an important feature of software. But by now, most application servers only provide the ability of hot deployment, while nearly no online evolution is considered.

Using components and interceptors that supported by application servers, we can trace changes from requirement to implementation, and provide more support for application evolution. We hope this paper can also give some suggestion for the development of application servers.

When object technology becomes mature, researchers start to chase the technology beyond objects. Booch thinks, "There's something deeper, something that's truly beyond objects". " First, there's the patterns movement ".“ Second, there's the growing understanding of the importance of multiple views in the science and practice of software architecture". Third, " there's the emerging area of aspect-oriented programming (AOP)"[19]. SOC[9]、AOP[8]、DFC[20] all have proposed heuristic approaches. Maybe the emergency of technology beyond objects is just in the near future.

The work introduced in this paper is supported by the Natural Science Foundation of China under Grant Nos. 60103001. We would also like to give special thanks to Sean DcDirmid for his kind work of correcting this paper.

Reference

[1] Lui Sha, Ragunathan Rajkumar and Michael Gagliardi, " Evolving Dependable Real-Time Systems", Software Engineering Institute Carnegie Mellon University, 1996

[2] Peyman Oreizy Nenad Medvidovic Richard N. Taylor "Architecture-Based Runtime Software Evolution", ICSE'98, 1998.

[3] P. Zave, " Classification of Research Efforts in Requirements Engineering" , In proceedings of the 2nd IEEE International Symposium on Requirements engineering (RE95), pages 214-216 York, England, 1995

[4] SUN, JPS, http://java.sun.com/blueprints/index.html.2001.

[5] Pamela Zave, " Requirements for Evolving Systems: A Telecommunications Perspective", IEEE Symposium on Requirements Engineering, 2001.

[6]Colin Atkinson ,Thomas Kühne and Christian Bunse, " Dimensions of Component-based Development", Proceedings of the 4th Internatinal Workshop on Component-Criented Programming, 1999.

[7] Truyen, E.; Vanhaute, B.; Joosen, W.; Verbaeten, P.; Jorgensen, B.N.; Leuven, " Dynamic and selective combination of extensions in component-based applications", ICSE, 2001.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, "Aspect-Oriented Programming" , ECOOP' 97 1997.

[9] Johan Brichau, Maurice Glandrup, Siobhan Clarke, and Lodewijk Bergmans, "Advanced Separation of Concerns", ECOOP, 2001.

[10]Phipps,D. " CBD: A Path to Uniform Component Design." Gartner Group Strategic Analysis Report. Stamford, CT: Gartner Group, Jan.2000.

[11]OMG, " corba23_book" , http://www.omg.org, 1999.10

[12]JBOSS, " Jboss" , http://www.jboss.org, 2001.

[13]SUN " EJB" , http://java.sun.com/products/ejb/, 2001.

[14]Yang Fuqing, Wang Qianxiang, Mei Hong, Chen Zhaoliang, "Reuse-Based Software Production Technology", Science in China, 2001.No.1.

[15]Gregory T. Sullivan & Jonathan R. Bachrach, " Advanced Programming Language Technology for Reflective, Dynamic, Adaptive Software"

http://www.ai.mit.edu/research/abstracts/abstracts2000/pdf/z-sullivan-bachrach.pdf.

[16] Linda M. Seiter, Member, IEEE Computer Society, Jens Palsberg, and Karl J. Lieberherr, ," Evolution of Object Behavior Using Context Relations" *IEEE Transactions on Software Engineering*, VOL. 24, NO. 1, JANUARY 1998

[17] Sheng Liang, Gilad Bracha, " Dynamic Class Loading in the Java Virtual Machine", OOPSLA'98, 1998

[18] SUN " JMX", http://java.sun.com/products-JavaManagement, 2001.

[19] Grady Booch, "Through the Looking Glass", Software Development, July 2001.

[20] Michael Jackson and Pamela Zave, "Distributed feature composition: A virtual architecture for telecommunications services" *IEEE Transactions on Software Engineering* XXIV(10):831-847, October 1998).

[21] E.Jane CAMERON, Nancy D.GRIFFETH, Yow-Jian LIN, " A Feature Interaction Benchmark for IN and Beyond", March, IEEE Communications Magazine 1993.

[22] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, " Design Patterns Elements of Reusable Object-Oriented Software", Addison Wesley, 1995.

[23] Johan Brichau, Maurice Glandrup, etc, " Advanced Separation of Concerns" , ECOOP 2001.

[24] OMG, "CCM", http://www.omg.org, 1998.