Post-Development Software Architecture

Gang Huang

Key Laboratory of High Confidence Software Technologies, Ministry of Education

School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China

E-mail: huanggang@cs.pku.edu.cn

Abstract

Software architecture (SA) plays an important role in software development. Since the lifecycle stages post development become more and more important and face with many challenges similar to the development, it is a natural idea to introduce or extend SA into the stages post development. In this paper, we present our practices and experiences on applying software architecture into the deployment and maintenance of J2EE (Java 2 Platform Enterprise Edition) applications, including the tool and principles of SA-based J2EE deployment and SA-based J2EE online maintenance. It demonstrates that 1) SA can help to achieve a holistic, fine-grained and automated deployment of large-scale distributed systems by visualizing the structure of the system to be deployed; 2) SA can provide an understandable, operational and global view for online maintenance by organizing the fragmented and trivial management mechanisms; 3) Extending SA into the stages post development makes it possible that the whole lifecycle of a software system can be governed by SA with many benefits, e.g. consistency, traceability, responsiveness, etc.

Keywords: software architecture, software maintenance, deployment, runtime software architecture

Introduction

Recognized as a critical issue in the engineering of complex software systems, software architecture (SA) becomes an important subfield of software engineering. SA describes the gross structure of a software system with a collection of components, connectors and constraints [27]. Generally, SA acts as a bridge between requirements and implementation and provides a blueprint for system construction and composition [6]. In the past more than one decade, both academia and industry communities have gained plentiful achievements and experiences in SA, most of which focus on software development.

Recently, introducing SA from the development to the post-development phases in software lifecycle, including the deployment, maintenance and evolution, gains more and more attention for the sake of three facts. First, SA is an abstraction of the target system and describes a set of snapshots of the runtime system. Second, the value and applicability of SA in software development are proved. Third, the post-development phases become much more important than ever due to such as the rapid and continuous changes of requirements, the diversity and changes of environments, the return of investment, the time to market, etc.

The efforts on post-development SA try to demonstrate the value and applicability and identify the methodological and technical challenges. Rakic et al. [25] propose the DeSi environment for specifying, manipulating, visualizing, and (re)estimating deployment architectures of large-scale, highly distributed systems. D. Llambiri et al. [15] find that different configurations of the same SA have very different response times of main components. Tu et al. [31] define the build-time architecture view to describe SA when the target system is compiled and linked. Oreizy et al. [23] study how SA can support corrective, perfective and adaptive evolution at runtime and experiment on C2, a layered, event-based architectural style. Garlan et al. [7] investigate the critical issues for using SA at runtime based on their plentiful work on architecture based self-repairing.

Though the above efforts gain exciting achievements and experiences, post-development SA is still far away from maturity and practicability. Particularly, since the software lifecycle can be divided into multiple phases after development, the applications of SA in these phases are different and may interact, sometimes interfere, with each other more or less. It implies that we should investigate the relationship not only between SA at development and SA after development but also between SA in different post-development phases. At the same time, since runtime environments have significant impacts on post-development SA, it had better to experiment on the popular ones, like J2EE and .NET, which would bring plentiful and realistic achievements and experiences.

In this paper, we will present our practices and experiences on post-development SA in J2EE applications, including the architecture based deployment [11][14] and architecture based maintenance [9][10]. We will also discuss what changes post-development SA brings to SA at development and how to cope with these changes in architecture design [33][34][18]. The main purpose of this paper is to re-think and synthesize our previous work on post-development SA from the perspective of SA in the whole software lifecycle for investigating the realistic values and the technical and methodological challenges of introducing SA into deployment and maintenance.

The rest of the paper is organized as follows: Section 2 introduces J2EE and PKUAS as the background; Section 3 and 4 present why, what, how and experimentation results of architecture based deployment and maintenance respectively; Section 5 discusses the relationships between SA at development, deployment and maintenance and corresponding challenges to the architecture centric software engineering; Section 6 introduces some related work and the last section concludes this paper and identifies the future work.

Background

Originally, our efforts to introduce SA into deployment and maintenance are motivated by dealing with some challenging problems in PKUAS [17], which is a J2EE application server. As shown in Figure 1, J2EE is the middleware including J2SE (supports the

execution of Java programs), common services (support functions common to network based systems, such as security, transaction and messaging), a Web Container (supports JSPs and servlets that deal with human-computer interactions and simple business logic) and an Enterprise Java Bean (EJB) Container (supports EJBs that deal with business logic and business data) [30].

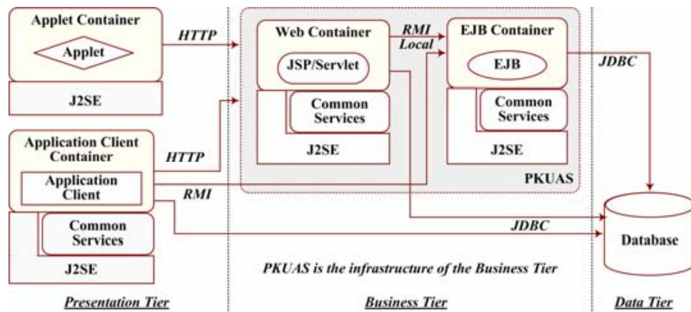

Figure 1. Overview of J2EE

As shown in Figure 2, PKUAS provides all functionalities required by J2EE v1.3 [30] and EJB v2.0 [29] in its componentized structure.

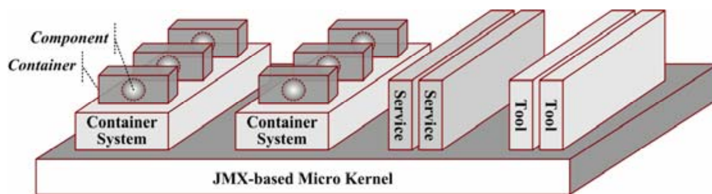

Figure 2. Componentized Structure in PKUAS

- Container system and container: a container provides a runtime space for the components in the deployed applications with lifecycle management and contract enforcement. PKUAS implements standard EJB containers for stateless session bean, stateful session bean, bean-managed entity bean, container-managed entity bean and message-driven bean. One instance of a container holds all instances of one EJB. And a container system consists of the instances of the containers holding all EJBs in an application. Such organization of containers facilitates the management specific to individual applications, such as security realm per application and architectural recovery of a given application.
- Service: it provides the common functions, like naming, communication, security, transaction and log. Particularly, the naming and communication services provide an interoperability framework that enables the components deployed in PKUAS to interact with each other and other components outside PKUAS through multiple interoperability protocols.
- Tool: it provides functions to facilitate the operation of PKUAS, such as deployment and management.
- Micro kernel: it provides a registry and an invocation framework for the above platform components and other management entities, like class loading, relation, timer and monitor.

Based on the componentized structure, PKUAS implements a reflective framework, which supports the monitoring and controlling of the internal states and behaviors of the whole J2EE system. The reflective framework is the most important feature for architecture based deployment and maintenance, which will be discussed later.

Architecture based Deployment

Motivation

Before a component based system can operate with desired functions and qualities, it has to be configured and installed correctly according to runtime environments. This activity is called software deployment, which plays a key role in software lifecycle. Software deployment has gained more and more attention over the past decade as the rapid pervasiveness of the network and distributed systems. Kruchten [13] proposes the "4+1" view model, which defines the deployment view for describing the mapping(s) of the software to the distributed nodes. In OMG's UML (Unified Modeling Language) [21], the deployment diagrams show the configuration of runtime processing elements and the software components, processes, and objects that execute on them.

In J2EE [30], the development process of J2EE applications is divided into three stages: component creation, assembly and deployment. During the deployment stage, the J2EE application is installed on the J2EE application servers with careful configuration and integration with the runtime environments. There are two critical issues in the deployment of J2EE applications.

First, J2EE defines XML based descriptors for standardizing the deployment. Deployers have to write a mass of description elements by hand even though some J2EE deployment tools are used. There are more than 25 items for an EJB, more than 30 items for a Web Application and more than 20 items for a J2EE Applications, including the names of EJBs, the declaration of required resources, security realm and roles, component names for runtime binding, and so on. To deploy a small system, e.g., the J2EE blueprint program of JPS, more than one thousand elements are needed. In fact, almost all of the elements about deployment already exist in and can be transformed from SA descriptions.

Second, when partitioning a system, deployers had better have a high-level guidance. Currently, deployers only have the codes or packages of a J2EE application. What they know is the corresponding components to the system at first sight. Other information of this system such as dependencies of components, interoperations between components, and security properties is sealed, though this information is very helpful for the deployment. It will take deployers many days to understand the whole system by reading related documents or even the source codes, which is obviously challenging and inefficient. So a clear and precise view of the whole system, including the components, the detailed structure and even the desired qualities, should be provided to deployers.

Activities

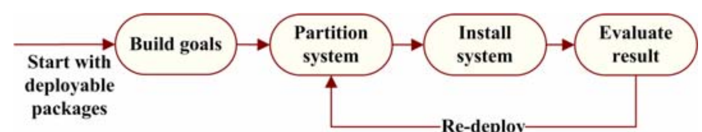

Figure 3. Activities in architecture-based deployment

For addressing the above two issues, it is natural and feasible to introduce SA into J2EE deployment. The architecture based J2EE deployment can be divided into five steps, as shown in Figure 3.

1) Building up the goals of the deployment: before the deployment, deployers need a clear view about the state which the system should achieve after being deployed. Deployers analyze the desired functions and qualities of the system based on the information in SA, consider many other technical and non-technical factors, e.g., operational strategies, and finally build up the goals of the deployment.
2) Partitioning the system based on the software architecture and runtime environment: for utilizing the resources better and improve the performance further, the system should be partitioned into several parts and installed on distributed nodes respectively. On the one hand, as the description of software system's gross structure, software architecture helps deployers to understand the components in the system and the relationships among them quickly, to further analyze the constraints of the components and the system. On the other hand, inspecting the runtime environment can help deployers to understand the environment and think over the factors of the environment totally that maybe impact the system.
3) Installing the system on distributed nodes: before operating, some information should be added to the system for deployment, i.e., deployment descriptors. This information can be transformed from SA descriptions and deployers just need to append a little of information to accomplish the installations.
4) Evaluating the result of the deployment: after the deployment, deployers should evaluate the result of the deployment based on the runtime information, which can be obtained by some management facilities provided by runtime environments. On the one hand, deployers should make a judgment whether the result meets the goals of deployment, and on the other hand, deployers should review other formerly deployed systems to ensure that they will not put any negative impacts on the newly deployed ones, and vice versa.
5) Re-deployment: if the evaluating results cannot meet the goals, the system should be redeployed. This phase consists of the foregoing phases from 2 to 4. This phase will have to be repeated until all of the systems meet their goals of deployment. In some cases, none deployment can meet the goals and then it is necessary to degrade the goals.

Supporting Tool

All activities in the architecture based J2EE deployment can be performed with the supporting tool, called CADTool [11]. It facilitates deployers to visually pack as well as assemble components. More importantly, based on the software architecture, CADTool extracts most needed information in the deployment from the architecture models in the development. Figure 4 shows the case of deploying JPS onto four nodes with CADTool. JPS is one of the sample applications for J2EE Blueprints, which are for demonstrating how to use the capabilities of the J2EE platform to develop flexible, scalable, cross-platform e-business applications. The "deploy" panel shows the software architecture of JPS, and the "Server's Information" panel shows real-time information of the runtime environment. CADTool can facilitate the deployment with four features.

First, it can visualize SA in the development. If the deployable package contains the architecture description in ABC/ADL[1] [19], CADTool can display the syntax and semantics information produced in the development. If the deployable package contains the layout description of the architecture, CADTool can display the architecture in the same layout as that in the development, which helps to understand the intention of the designers. If the deployable package does not contain the two descriptions, CADTool can automatically construct the architecture from the individual deployable components. However, the last case is not desired because the recovered architecture lacks enough information derived from the development.

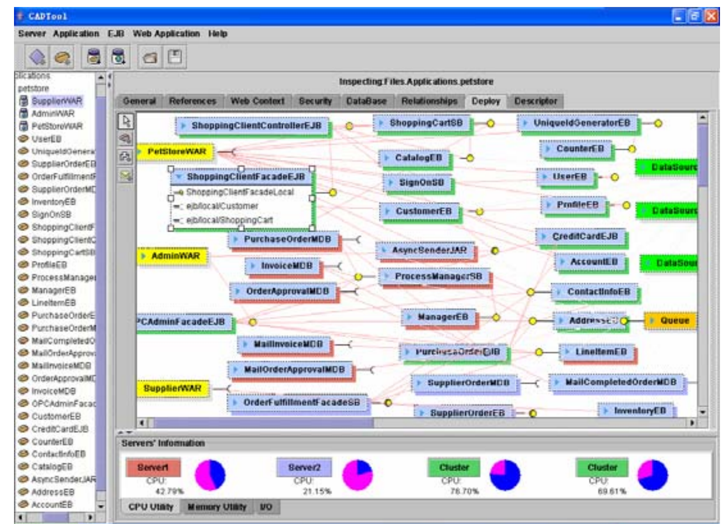

Figure 4. Architecture based Deployment Tool

Second, CADTool can visualize the servers and their capabilities. Using the reflective framework of PKUAS, CADTool can automatically collect and display the servers' information, such as CPU utilization, memory utilization, throughout, etc. The information is useful to determine which components should be deployed into which servers. They also help to investigate whether the deployment works well. For example, the *CatalogEJB* consumes much CPU time. If the component is deployed into the *Server1*, the CPU utilization of the *Server1* may exceed 90% and the *Server1* becomes unstable and easy to crash. Then, it'd better undeploy the *CatalogEJB* in the *Server1* and re-deploy it into the cluster.

Third, CADTool supports the deployment in a drag-and-drop manner. With the above two visual elements, a component can be easily deployed into a server just through dragging the component and dropping it on the target server or vice versa. In traditional deployment tools, deployers have to connect to a given server, load the components to be deployed into the server, and repeat the work again for another server. If a component is deployed into a server, the box representing the component will have a shadow with the same color as the server. In the Figure 4, the red, blue and green colors identify *Server1*, *Server2* and *Cluster (Server3 + Server4)* respectively.

Fourth and final, CADTool can automatically calculate some system-level or scenario-level properties. There are many successful case studies on the quantitative and qualitative evaluations of the given architecture models. However, some properties may be

[1] It is a customizable and extensible architecture description language with emphasis on middleware based systems. Since CADTool only uses the common ADL elements, it can integrate the artifacts in other ADLs without fundamental changes.

wrongly predicted in the design phase and should be re-evaluated in the deployment. Specially, some properties may be only available after the system running for a period, such as the response time and throughput. That means the deployment may not meet the requirements related to these properties. Then the whole or part of system has to be re-deployed with the actual properties. Currently, CADTool can automatically calculate the response time, throughput and reliability of a given scenario.

Evaluation

After deploying JPS and RUBiS (an eBay-like bidding system prototype for evaluating the bottlenecks of such applications) in many different plans and comparing the results, we find a set of principles that demonstrate the values of SA in J2EE deployment besides the automatic generation of deployment descriptors [14].

For example, composite components that are made up of several simple components are natural partitions. In general, interactions among the internal components of a composite component are very frequently. So these components should be deployed on the same node for avoiding heavy network traffic and delay. If a composite component is too large to be deployed on any node, the internal components can be distributed into multiple nodes in the same local area network. Furthermore, different composite components maybe involve the same simple components and then the same simple components might be deployed onto more than one node at the same time.

For another example, interactions between components are critical for evaluating network resources. In a distributed environment, the bottleneck of a system sometimes is the network. The transmit delay, bandwidth and topology structure might affect the system's performance greatly. However, if two components interact with each other rarely and the passing messages are very small, network resources evaluation does not make sense. On the contrary, the evaluation is very important when two components have frequent and heavy interactions.

Architecture based Maintenance

Motivation

Software maintenance is a time-consuming, error-prone and tiring but inevitable and important job. There is a consensus that most of the complexity and cost of software maintenance mainly result from the difficulty of understanding the large-scale and complex software. Since that SA helps to understand large-scale software systems is well recognized, it is a natural idea to improve the understanding of the system to be maintained through its architecture. However, SA at development cannot be directed used in maintenance due to the significant differences between these two phases.

First, SA at development may not be available because the system is developed without an explicit architecting phase or the documents are lost. Moreover, SA at development may have some differences or mismatch with the actual SA built in the target system due to some misunderstandings and mistakes in the detailed design and implementation. This problem can be addressed by software architecture recovery [32]. But SA is usually recovered from the source codes and other documents of the target system based on program comprehension, which cannot capture precise and enough runtime states and behaviors.

Second, since different stakeholders have different requirements, SA at development is for developers and may not fit for maintainers. For example, maintainers require much more details about the system at runtime than developers, e.g., internal structure of a component, details of a connector, and so on. In that sense, SA at maintenance is a refinement of SA at development with plentiful runtime information.

Third, maintainers often adjust the system to meet some new requirements, achieve better performance, correct errors or faults, etc. If maintainers change the system via normal management tools, some mistakes may occur because the normal tools are not designed for architecture based maintenance and then unable to perform the desired changes exactly, i.e., the changes made on the system are less or more than the desired changes.

Activities

Generally, the maintenance can be divided into five steps, i.e., monitoring the states and behaviors of runtime systems, analyzing the runtime information for triggering changes, planning when and what to change when necessary, controlling runtime systems for perform changes and evaluating the effect of changes. For the architecture based maintenance, the process and activities can be refined as shown in Figure 5.

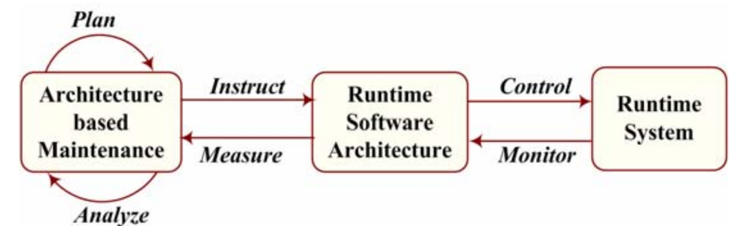

Figure 5. Activities in architecture-based maintenance

1) Monitoring runtime systems from the perspective of SA: as mentioned above, SA at maintenance must maintain a precise and up-to-date mapping to the runtime system. So it is necessary to collect runtime data and transform them into SA elements.
2) Measuring runtime SA for filtering changes: though changes take place time to time inevitably, many of them are unworthy to be handled. At the same time, it is tiring and sometimes impossible for maintainers to analyze any changes in runtime systems. Therefore, changes in runtime SA have to be measured and filtered in terms of maintainers' requirements.
3) Analyzing changes of runtime SA for detecting triggers: if a change indicates the occurrence or even the trend of a loss of functions or a decrease of qualities, it will trigger some corresponding changes. The evaluation of the adaptation is also performed in this step. It should be noted that changes out of the runtime system may also cause adaptations, e.g., allowing new customers to access some services.
4) Planning the adaptations: maintainers have to make some decisions for coping with the serious changes, i.e., when and what to change the states and behaviors of runtime systems.
5) Instructing runtime SA for adaptations: The adaptations have to be transformed into changes of runtime SA.
6) Controlling runtime systems from the perspective of SA: the changes of runtime SA have to be precisely executed in runtime systems.

Supporting Framework

Compared to SA at development, SA at maintenance should be accessible and operable at runtime and provide a more concrete view with plentiful runtime information. For supporting such features, we upgrade PKUAS from a customizable and extensible middleware to reflective middleware, i.e., architecture-based reflective middleware [17], as shown in Figure 6. The core idea is to implement ADL elements as meta entities in the reflective framework, in which the traditional components in middleware platform and applications are base entities. There is a causal connection between base entities and meta entities, that is, changes of meta entities will cause the corresponding changes of base entities immediately, and vice versa. These ADL elements form an accessible and up-to-date SA of the runtime system, which is called runtime software architecture (RSA) [9].

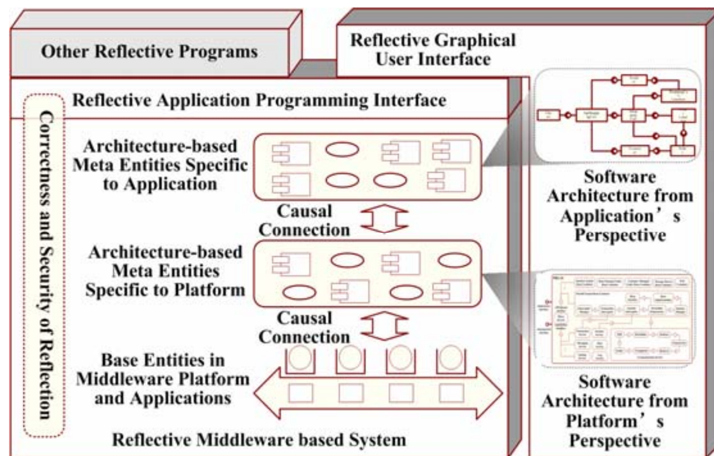

Figure 6. Architecture-based Reflective Framework

The states and behaviors of middleware platform and applications can be observed and adapted from the perspectives of the platform RSA and application RSA respectively. The platform RSA represents the detailed implementation of middleware platform as components and connectors. Middleware applications are invisible, i.e., represented as the attributes of some components. For example, a J2EE application server consists of containers and services while a J2EE application consists of EJBs and Servlets. In the platform RSA, the containers and services are represented as components; their interactions or dependencies are represented as connectors; and the EJBs or Servlets are represented as the attributes of the containers. On the contrary, the application RSA represents the middleware application as components and connectors. The middleware platform is represented as constraints or attributes of components and connectors. For example, J2EE security and transaction services are represented as the security and transaction constraints on the EJBs.

The platform RSA and application RSA can support different maintenance tasks. Particularly, the value of the application RSA is mainly determined by the semantics it provides. Like CADTool, if SA descriptions in ABC/ADL at development are available, PKUAS can enrich the semantics of the application RSA with plentiful design information. Otherwise, PKUAS can recover SA from execution traces automatically [10] but the recovered SA has the precise syntax and poor semantics. Moreover, in this case, PKUAS has to capture and analyze massive tracing data, which is very expensive.

Architecture based maintenance can be done by people or intelligent computer programs. Then PKUAS provides both API and GUI for accessing the reflective SA. Moreover, since the reflection opens up the internal states and behaviors of the runtime system, it brings new security threats. As a result, PKUAS implements a four-level security framework for controlling the accesses between clients and the reflective framework, between meta entities, between meta entities and base entities, between meta entities and local resources.

Evaluation

It is inevitable that maintaining runtime SA has a significant impact on the performance of the whole system. Though PKUAS makes many tradeoffs between the performance and reflection, such as the optimized invocation framework for base entities and meta entities, the sharing of security information, the prohibition of Java security manager, etc., the performance penalty in real systems has to be evaluated. Based on ECperf, the standard performance benchmark for J2EE application servers, we test three cases, i.e., PKUAS without reflection (no meta entities), with passive reflection (instantiate meta entities but do not collect data per invocation) and with active reflection (instantiate meta entities and collect data per invocation). The first two cases have similar throughput and response time while the last case decreases throughput by 7%-10% and increases response time by 8%-69%. We also find that the performance penalty of runtime SA increases drastically when the whole system has a high workload. So, the maintenance tasks can be done when the system workload is low for reducing the performance penalty.

The experimentation demonstrates that introducing SA into maintenance has a notable but acceptable impact on the performance. Besides the performance impact, SA at maintenance has another critical issue, i.e., how to guarantee the correctness of changes. If the changed SA becomes inconsistent or incomplete, the corresponding changes of the runtime system will cause something wrong. The correctness of architecture based maintenance has two aspects. One is that changed SA is correct and another is that the corresponding changes of the runtime system are performed correctly. The second correctness is guaranteed naturally and directly by the reflective relationships between SA and the runtime system. The first correctness can be achieved by integrating the model checking of SA at development. The model checking is an import issue of SA, which usually requires some formal semantic models to be appended to the architecture description languages [16]. Since SA at maintenance can inherit syntax and semantics from SA at development, it can reuse the model checking tools and information at runtime. However, SA at maintenance contains more information than SA at development, which implies that the model checking could get more data and check more things. For example, if a component supports different numbers of concurrent clients in different hosts, changing its location has to evaluate the concurrency requirement.

Discussion

Impacts of Post-development Software Architecture on Architecture Design

After studying many cases, we find that post-development SA is really valuable and applicable but very difficult to use. Though

runtime SA provides necessary mechanisms for monitoring and controlling runtime systems, it is still too difficult and expensive to maintain the systems in the rapid and continuous changes. The major reason is that runtime SA only solves the "how to do" problem, but cannot answer "why, when and what to do" in the maintenance. In details, middleware has no knowledge of what information should be collected for given quality attributes, e.g., does the response time of the user-interface component is critical to the performance? Furthermore, middleware cannot determine what values should be for satisfying quality requirements, e.g., is the expected response time of the user-interface component 2 seconds or 5 seconds? Finally, middleware cannot make appropriate adaptation decisions by itself, e.g., what should be done when the expected response time of the user-interface component is exceeded. Even though runtime SA can inherit information from SA at development, it is too difficult for middleware to make correct decisions by itself. Such problems become a little easier but still so hard for maintainers because it is not easy to understand SA at development if they do not participate in the development. Deployers and SA at deployment have the same problems.

In our opinion, the difficulty of using post-development SA is mainly resulted from the assumption that SA at development is produced by classical architecture based development methods, which means developers are unaware of post-development SA and then take none or little of deployment and maintenance into consideration. On the other hand, though deploy-time SA and runtime SA refine develop-time SA more or less, the main syntax and semantics of SA are produced at development. In other words, developers have much more knowledge and much better understanding of the system than deployers and maintainers. Since both deployment and maintenance require knowledge from not only post-development but also development, they should be performed by deployers, maintainers and developers together. But in practice, it is impossible to call all of them together for any post-development activity.

Therefore, a possible solution is to make developers aware of post-development SA and then make decisions or suggestions as many as they can. For example, since the analysis and design of quality attributes are one of the major tasks in architecting, architects can analyze and measure the qualities of the whole system and decide how to guarantee the qualities when necessary. They know the priorities of different quality requirements and then can tradeoff among them at the system level. Supposed that the response time of the system exceeds the expected value, knowing that performance is more important than security in the system, architects can reduce the response time via disabling some access control mechanisms for some unimportant components. Such adaptations can be easily performed by runtime SA.

In [34], we propose a quality attribute scenario based method for finding and analyzing the potential adaptation points in SA during design phase. As shown in Figure 7, the method has four vital steps. First, the application architecture is designed according to the requirements specifications by classical architecture-centric methods. Second, the quality attribute scenarios in the requirements specifications are investigated to find architectural elements that may be changed at runtime and then violate some qualities. Third, for a given scenario, the architect tries to find a solution to change the architecture so as to maintain certain quality attribute specified by the scenario. The application architecture with a set of such solutions can be considered as a self-adaptive architecture. Fourth and final, such SA will be interpreted by runtime SA. We also studies how to deal with dependability with develop-time, deploy-time and runtime SAs [18].

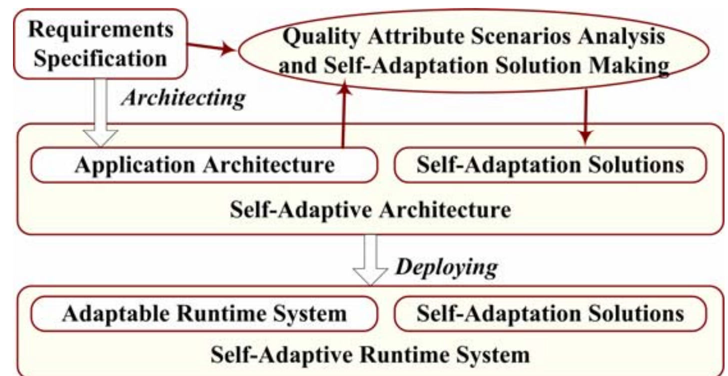

Figure 7. Self-adaptive architecture design

Role of Architecture Description Language

Architecture Description Language (ADL) is proposed to provide formal notations for development and analysis of software architectures [27]. Definitely, ADL is the key to keep the traceability and consistency between SA at development and SA after development. As shown in Tab. 1, the architecture descriptions using ABC/ADL [19] help to generate almost all information needed in the J2EE deployment descriptor [30][29].

Tab. 1 The mappings between ABC/ADL elements and J2EE deployment descriptor elements

ABC/ADL Elements	J2EE Deployment Descriptor Elements
Name of *ComponentDef*	<ejb-name> and <jndi-name> in <module>
Name of the provide player of *ComponentDef*	<home> and <remote> or <local-home> and <local> in <session> or <entity>
Name of the request player of *ComponentDef*	<home> and <remote> in <ejb-ref>; <local-home> and <local> in <ejb-local-ref>
Attributes of *ComponentDef*	<env-entry>, <resource-ref>, <cmp-field> and <primkey-field>
Properties of *ComponentDef and AspectDef*	<ejb-class>, <session-type>, <persistence-type>, <prim-key-class>, <transaction-type>, <reentrant>, <security-role-ref>, <security-role>, <method-permission>

Since SA at maintenance contains more details than SA at development, existing ADLs have to be extended. Figure 8 illustrates the major extension of ABC/ADL for describing SA at maintenance [10]. The *Runtime Software Architecture* consists of all component and connector instances in the runtime system and extends *Configuration* in classical ADLs, which is a snapshot of a runtime system predicted at design. The *Component* contains one or more implementations and the *Connector* contains client-side proxies, connections and server-side proxies, all of which are entities provided by middleware for interoperability.

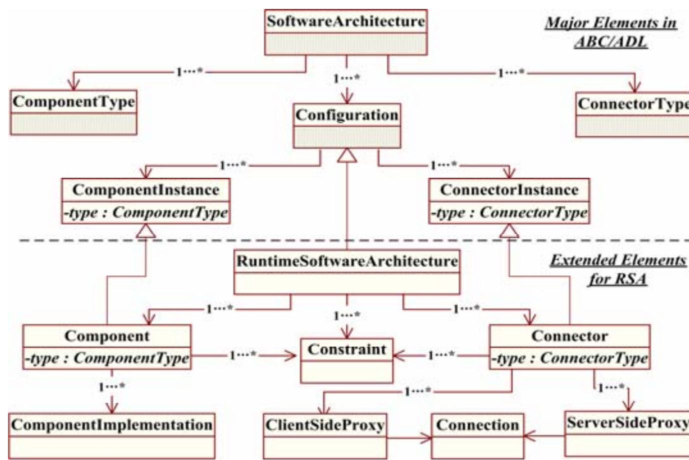

Figure 8. Description of architecture at maintenance

When we try to design SA with consideration of post-development, classical ADLs have to be extended for capturing some important details of runtime systems. Tab. 2 shows how to describe or model runtime changes in ABC/ADL [34]. The communication infrastructure (for interoperability) is represented as a simple or complex connector [33], and the common services provided by middleware are represented as aspects. Their runtime changes can be specified by the change point of a connector and weaving of aspects respectively.

Tab. 2 ADL elements describing runtime changes

Corresponding adaptations in PKUAS		ADL elments	
Common service	adding, removing, replacing even adapting the interceptors	Aspect	*Action* in *Weaving*
Communication infrastructure	adding or removing corresponding interoperability protocols and transport protocols	connector	*ChangePoint* in *predefined* protocol
	adaptability of protocol implementation		ChangePoint in *property*
Combination	combination of above modifications		*ChangePoint* in *userdefined* protocol

It should be noted that the ADL for SA at maintenance is different to that for SA at development that is aware of runtime changes because they have different stakeholders. On the one hand, developers just "be aware of" other than "understand" runtime details. It implies that runtime changes should be represented in terms of the background and comprehension of developers. On the other hand, maintainers need SA capturing runtime details precisely and completely. However, these two different perspectives make it uneasy to keep the traceability and consistency between different phases.

Related Work

Software Architecture in Deployment

The deployment view in the "4+1" views and deployment diagrams in UML describe the prediction or planning of the deployment at development. At the same time, some researchers try to deploy systems with the guide of SA.

Dearle et al. [4] propose a framework for constraint-based deployment and automatic management of distributed applications. In this framework, a purely declarative and descriptive ADL, named Deladas, is used to describe a deployment goal. To satisfy the goal, an automatic deployment and management engine (ADME) tries to generate a configuration, which describes which components are deployed in which hosts. After the initial deployment, the ADME will monitor the deployed application to check whether the deployment satisfies the original goal and re-deploy the application if necessary. This approach has the similar philosophy to our approach on the role of SA in the deployment. However, this approach ignores the plentiful knowledge derived from the development and the runtime states of hosts. Without such knowledge, it is very difficult to generate the proper configuration in a manual or automated way.

Rakic et al. [25] propose DeSi to support flexible and tailorable specification, manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems. They focus on how to take the availability into account in the deployment, including defining a formal foundation and investigating six algorithms to automatically generate the deployment plan. However, in DeSi, the formal specification of the deployed application has to be written by hand and some values in the specification are difficult to retrieve without the support of runtime environments. On the other hand, the formal specification can be automatically generated in CADTool with the plentiful knowledge derived from the development and runtime states of hosts. In our opinion, the work of DeSi can improve the reliability calculation of CADTool, which is under development.

Software Architecture in Maintenance

Oreizy et al. [23] study how SA can support corrective, perfective and adaptive evolution at runtime and experiment on C2, a layered, event-based architectural style. Garlan et al. [8] use the gauges to collect the states and behaviors of the underlying system and adapt the system according to some special requirements at runtime. Rosenblum et al. [26] investigate the architectural concerns in the component interoperability framework and combine the JavaBean model with C2 style. Bril et al. [2] provide a toolset, called URSA, to support program understanding and complexity management in Philips. In these approaches, SA at development is used as a document at hand, that is, such SA cannot accurately and up-to-date describe the target system. And changes made on SA do not cause corresponding changes in runtime systems until maintainers explicitly manipulate runtime systems in other ad hoc ways.

OpenORB [1] adds reflection ability into COM (Component Object Model). It provides four self-representations, including SA of the whole system, the interfaces of components, the interception of components and resources. In fact, the complex views provided by OpenORB and PKUAS are originated by the separation of concerns in reflection proposed by Okamura et al. [22] and the architectural reflection proposed by Cazzola et al. [3]. But OpenORB and PKUAS have quite different implementations. Firstly, OpenORB represents the system in four independent views while PKUAS represents all information in SA. Secondly, OpenORB defines a set of reflective interfaces that the reflective COM objects have to implement. Then, developers of application components have to be aware of reflection. In that sense, OpenORB just implements a reflective component model, just like K-Components [5] and FORMAware [20]. On the contrary, PKUAS implements architectural reflection in a "pure" reflective middleware way, that is, does not define such interfaces so that application developers are unaware of reflection. In our opinion, both

ways have advantages and disadvantages and a reflective EJB component model is under development to improve reflective PKUAS.

Conclusion and Future Work

Making software architecture available in the whole software lifecycle becomes a hot topic recently. This paper just presents our practices and experiences on introducing SA from development into deployment and maintenance. The most important contribution of this paper is that we analyze realistic requirements and benefits of post-development SA and identify technical challenges with demonstration on J2EE which is one of the most popular runtime environments. Another important contribution is that we investigate methodological challenges for extending SA into the whole software lifecycle.

There are many open issues to be addressed. Since our original goal is to facilitate the construction of adaptive component based systems via middleware, we will focus on how to model dynamic SA enabled by middleware and how to utilize the knowledge embedded in SA for making middleware based systems self-adaptive.

Acknowledgements

This effort is sponsored by the National Basic Research Program (973) of China under Grant No. 2005CB321805; the National Natural Science Foundation of China under Grant No. 90612011, 90412011, 60403030.

References

[1] Blair, G.S., Coulson, G., Andersen, A., and etc. 2001. The Design and Implementation of Open ORB 2. IEEE Distributed Systems Online, 2(6).

[2] Bril, R.J., Feijs, L.M.G., Glas, A., Krikhaar, R.L. and Winter, R.M. 2000. Maintaining a legacy: towards support at the architectural level. Journal of Software Maintenance: Research And Practice, 12:143–170.

[3] Cazzola, W., A. Savigni, Sosio, A. and Tisato, F. 1998. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. 6th Reengineering Forum.

[4] Dearle, A., G. Kirby, A. McCarthy. A Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications. International Conference on Autonomic Computing, 2004, pp 300-301.

[5] Dowling, J. and V. Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In Proceedings of Reflection 2001, LNCS 2192, pp.81-88.

[6] Garlan, D., Software Architecture: A Roadmap, The Future of Software Engineering 2000, Proceedings of 22nd International Conference on Software Engineering, ACM Press, 2000, 91-101.

[7] Garlan, D., B. Schmerl, Using Architectural Models at Runtime: Research Challenges, European Workshop on Software Architectures, 2004, pp. 200-205.

[8] Garlan, D., Schmerl, B. and Chang, J.C. Using Gauges for Architecture-Based Monitoring and Adaptation. The Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, 12-14 December, 2001.

[9] Huang, G., H. Mei, Q.X. Wang. Towards Software Architecture at Runtime. ACM SIGSOFT Software Engineering Notes, Vol. 28, No. 2, March 2003.

[10] Huang, G., Mei Hong, Yang Fuqing. Runtime Recovery and Manipulation of Software Architecture of Component-based Systems. Journal of Automated Software Engineering, Springer, Springer, Vol. 13 No. 2, 251-278, Feb. 2006.

[11] Huang, G., Meng Wang, Liya Ma, ling Lan, Tiancheng Liu, Hong Mei. Towards Architecture Model based Deployment for Dynamic Grid Services. In Proceedings of IEEE International Conference on E-Commerce Technology for Dynamic E-Business, 2004, pp. 14-21.

[12] Huang, G., Tiancheng Liu, Hong Mei, Zizhan Zheng, Zhao Liu, Gang Fan. Towards Autonomic Computing Middleware via Reflection. In Proceedings of Annual International Computer Software and Application Conference (COMPSAC), 2004, pp. 122-127.

[13] Kruchten, P. The 4+1 view model of architecture. IEEE Software, 1995, Vol. 12, No. 6, pp. 42–50.

[14] Lan, L., Gang Huang, Liya Ma, Meng Wang, Hong Mei, Long Zhang, Ying Chen. Architecture based Deployment of Large-Scale Component based Systems: the Tool and Principles. Proceedings of 8th International SIGSOFT Symposium on Component-based Software Engineering (CBSE), LNCS 3489, 2005, pp. 123-138.

[15] Llambiri, D., Alexander Totok, Vijay Karamcheti. Efficiently Distributing Component-Based Applications Across Wide-Area Environments. 23rd International Conference on Distributed Computing Systems (ICDCS 2003), pp. 412-421.

[16] Medvidovic, N., Taylor R. A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, 2000, Vol.26, No.1: 70-93.

[17] Mei, H. and G. Huang. PKUAS: An Architecture-based Reflective Component Operating Platform, invited paper, 10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004, pp. 163-169.

[18] MEI, H., Gang HUANG, W.T. TSAI, Towards Self-Healing Systems via Dependable Architecture and Reflective Middleware. 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005), Feb 2-4, 2005, Arizona.

[19] Mei, H., F. Chen, Q. Wang and Y. Feng. ABC/ADL: An ADL Supporting Component Composition. 4th International Conference on Formal Engineering Methods (ICFEM 2002), pp. 38-47.

[20] Moreira, R.S., G. S. Blair, E. Carrapatoso. A Reflective Component-Based & Architecture Aware Framework to Manage Architecture Composition. 3rd International Symposium on Distributed Objects and Applications (DOA 2001). pp. 187-196.

[21] Object Management Group. Unified Modeling Language Specification, Version 1.5, formal, 2001.

[22] Okamura, H., Y. Ishikawa, and M. Tokoro. AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework, Proc. Int'l Workshop on Reflection and Meta-level Architectures, Japan, 1992, pp. 36-47.

[23] Oreizy, P., N. Medvidovic, R. N. Taylor. Architecture-based runtime software evolution. 20th International Conference on Software Engineering, 1998, pp 177-186.

[24] Perry D. and A. Wolf, Foundations for the Study of Software Architecture, ACM SIGSOFT Software Engineering Notes, 1992, 17(4): 40-52.

[25] Rakic, M.M., S. Malek, N. Beckman and N. Medvidovic, A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings, 2nd International Working Conference on Component Deployment, Edinburgh, UK, 2004.

[26] Rosenblum, D.S. and Natarajan, R. 2000. Supporting Architectural Concerns in Component Interoperability Standards, IEE Proceedings – Software, 147(6):215-223.

[27] Shaw, M., D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[28] Soley, R. and the OMG Staff Strategy Group, Model Driven Architecture: OMG White Paper, Draft 3.2, http://www.omg.org/mda, Nov 27th, 2000.

[29] SUN Microsystems, Enterprise JavaBeans Specification, Version 2.0, Final Release, 2001.

[30] SUN Microsystems, Java 2 Platform Enterprise Edition Specification, Version 1.3, 2001.

[31] Tu, Q. and M.W. Godfrey, The Build-Time Software Architecture View, IEEE International Conference on Software Maintenance (ICSM 2001), pp. 398-407.

[32] Van Deursen, A. Software Architecture Recovery and Modeling: [WCRE 2001 discussion forum report]. ACM SIGAPP Applied Computing Review, 2002, 10(1): 4-7.

[33] Zhu, Y., Gang Huang, Hong Mei, Modeling Diverse and Complex Interactions Enabled by Middleware as Connectors in Software Architectures. Accepted by the 10th IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS), Shanghai, China, 16-20 June 2005.

[34] Zhu, Y., Gang Huang, Hong Mei, Quality Attribute Scenario Based Architectural Modeling for Self-Adaptation Supported by Architecture-based Reflective Middleware, In Proceedings of 11th Asia Pacific Software Engineering Conference (APSEC), 2004, pp. 2-9.