Integration of Component-Based Development-Deployment Support for J2EE Middleware

Adirake Pimruang, Kazuhiro Fujieda, and Koichiro Ochimizu

Japan Advanced Institute of Science and Technology, School of Information Science,
1-1 Asahidai, Tatsunokuchi, Ishikawa, Japan
{p-adirak,fujieda,ochimizu}@jaist.ac.jp

Abstract. From the widely use of component middleware, developers can reuse existing components not only developed by in-house development but also provided by other organizations. Some components developed in an organization can be deployed in other organizations via the Internet. Developers need to handle the dependency information between such components in both of development and deployment phases. We propose a system called J2DEP to generate and manage such information in the development phase, and to automate the deployment of components. J2DEP copes with configuration management systems to manage components and the information. It manages the dependency information between in-house components and third vendor components, and provides a consistent set of components in the release and deployment phases.

1 Introduction

The middleware, architecture for the development and deployment of software components, is now widely used in business (e.g. J2EE [1], Microsoft .NET [2] and CORBA [3]). Each component encapsulates part of a software system implementing a specific service or a set of services to support business requirements. To build large business systems, developers need several functionalities from the existing components. They can reuse their own in-house developed components or purchase components from third-party vendors to construct applications in middleware technology. Reuse of existing components can reduce time and cost of software development [4].

Each component can be provided by in-house development or come from other organizations distributed in different locations. These organizations generally publish their components to their release sites as binary units to avoid source code release [5]. Developers can integrate these binary components to develop new components or applications [6]. They, however, could not acquire the dependency information of third vendor components. They need to resolve component dependencies manually when they adopt the third vendor's components. In deployment phase, it is also troublesome and consumes time to deploy the proper

versions of components without their dependency information. Another problem occurs when the version of a component is changed. Version conflicts appear in the deployment phase because the effect of the change cannot be traced [7].

Current software configuration management (SCM) systems do not well provide to enable component-based development and component reuse [8]. Different organizations can provide several versions of components in which their dependency information is not explicitly described [9]. They cannot properly manage the evolution of components developed by third-party organization. Each third party organization develops its components and releases them in its own release policy. For example, some organizations may release sources of their components in their SCM repositories, and others may release binary components in their HTTP server. SCM systems cannot help us to manage the dependency information of components released with different policies.

The software deployment including following activities: obtaining components, these dependents, packaging, and releasing, should be done in automatic ways [10]. To realize automatic deployment, developers generally defines dependency information of components in the release phase. Then, deployers use a deployment tool to obtain components from the release site according to the information. In component-based development, component identification starts from the design phase [11]. Developers implicitly or explicitly use the information in the development phase and redefine it again in the release phase.

What we need is the SCM system that supports component-based development. SCM must support managing component relationships and the change of component versions by different organizations. This means such a system must help developers to adopt components based on different release policies, to generate the dependency information in the development phase and manage both sources and them in SCM repositories. With this system, the deployment process can be performed automatically. Developers can use the dependency information of components to obtain the correct versions of components to be assembled and deployed in developers' middleware in the build and test phase. Also, deployers can get correct versions of components to deploy in user middleware in component the installation phase.

In this paper, we propose J2DEP (J2EE DEvelopment-dePloyment support), which supports configuration management addressing both in development and deployment phases. J2DEP helps developers to create or generate dependency information from imported components and manages sources and dependency information inside a CVS repository to support the development process. Moreover, J2DEP can also publish the binary components to release sites by using FTP/HTTP servers. In the development process, J2DEP helps developers to import remote components, which is developed by different organizations, to its development environment by getting source files from the repository or binary components from the release site. Then, it generates the dependency metadata from imported components and control metadata files in the repository. The dependency metadata mainly represents the component details (e.g. name, version and type) and the method to obtain the component from a repository or

a release site. Finally, in the deployment phase, J2DEP helps developers to get components from release sites, assemble relating components into application components and to install them and the dependency information to the target platform.

The paper is further structured as follows. We introduce the background of this research in Sect. 2 and give an example scenario that motivates this research in Sect. 3 We outline the overview and approach of J2DEP system in Sect. 4. In Sect. 5, we give the implementation details of J2DEP system in development-deployment phase support. We show related works in Sect. 6 and give a conclusion in Sect. 7.

2 Background

J2DEP intends to support both of component development and deployment phases. In this section, we would like to discuss about related works and motivation of this research.

2.1 Component Development-Deployment

To realize the problem raised in the development process in middleware, we would like to show the development roles and tasks in J2EE. J2EE development-deployment roles consist of the following main three ones [1].

Application Component Provider An application component provider provides the building blocks of a J2EE application. A provider can develop components and package binary files into an application component. A component from provider may have dependencies on other components. In J2EE, there are two methods to handle such dependencies.

Package dependent components into a new component: The dependencies can be reduced by grouping the related components into a new component. However, this method reduces the degree of component reusability because dependent components become a part of the new component.

Do not package dependent components: This method can maximize the reusability of each component. We have to leave a room for application assemblers to pick and select components to compose J2EE applications.

The problem raised in this role is about the dependencies. In the development phase, generally, developers do not manage the dependency information and its changes of their components in their repositories. Moreover, dependency information in the development phase is often reduced in the release phase because some components may be a part of another component. In this case, the same components may have different dependency metadata in the repository and the release site.

Application Assembler An application assembler groups a set of components developed by application component providers and to assemble them into a J2EE application. An application assembler is responsible for providing assembly instructions describing external dependencies of the application that the deployer must resolve in deployment phase.

To support application assembler automatically, the assemblers need to use dependency information to obtain related component and to generate deployment descriptor about external dependencies.

Deployer A deployer installs components and applications into a J2EE server. He has to resolve all the external dependencies declared by the application component provider and the application assembler to configure them. The application component provider should define dependency information of components properly in the development phase.

2.2 Software Configuration Management (SCM)

Software configuration management concept is to manage charges of software artifacts. The most of SCM systems including RCS [12] and CVS [13] can manage only text file. While component-based development, developer has to deal with both component sources in text format and binary components. So we need a method to manage changes in binary components on local SCM systems (SCM systems of in-house development) when developers want to reuse them.

2.3 Software Deployment

The software deployment life cycle is evolving these activities: package, release, configure, assembler, install, update, remove and adapt [14]. The most of deployment tools can support component development and can manipulate the component dependencies. Some tools can support component development among distributed organizations. SRM [15], Software Dock [14], RPM [16] can manage multiple version of component. But these tools do not connect the deployment process with SCM. TWICS [10] resolves this shortcoming by supporting to get components from SCM repository (third vendor repository or in-house repository) to release and get the component from component publisher to a local SCM repository (source repository of in-house development).

However, deployment tools we mentioned above do not connect the development and deployment phases properly. The component dependencies are generally defined in the release phase. By reuse concept, related components have been defined in the design phase. The developer needs the dependencies defined in the deployment phase especially in the build and test phases, so the dependencies should be defined in the development phase rather than in the release phase.

3 Example Scenario

To clarify the issues of component development-deployment in middleware, we consider the relationship of components developed by different organizations shown in Fig. 1. The rectangle boxes represent components developed by each organization displayed in oval shapes. The arrows show the dependencies among the components. The text below each rectangle box shows its component name, version and release type respectively.

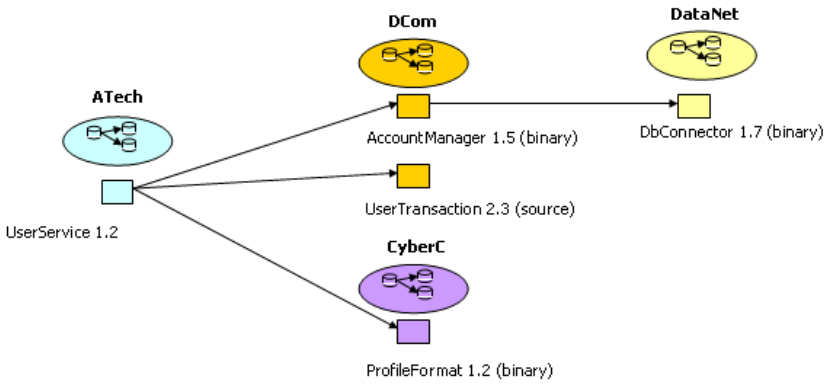

Fig. 1. Example scenario of component-based development

The first issue is that each organization may develop several versions of components and each component may depend on other components. Developers have no support to document the dependency information properly. They need to resolve dependencies manually whenever they get the sources from their repository to put into the development environment. For example, the dependencies of UserService 1.2 on AccountManager1.5, UserTransaction 2.3, DbConnector 1.7 and ProfileFormat 1.2, are not provided in development phase. The dependency information in the release phase cannot be documented properly because it is not defined in the development phase. As the result, we cannot guarantee the consistency throughout the deployment phase.

The second issue is that the some dependencies may be reduced in release phase because developer may combine some components with another component. For example, the dependencies of UserService 1.2 are AccountManager1.5, UserTransaction 2.3, DbConnector 1.7 and ProfileFormat 1.2 in development phase. In release phase, the developer can combine AccountManager 1.5 and DbConnector 1.7 with UserService 1.2. The dependencies of the resulting UserService 1.2 become only UserTransaction 2.3 and ProfileFormat 1.2. The component dependencies in the development phase and the deployment phase can be different.

The third issue is that each organization may develop and publish the components based on different release policies. They may publish the component sources their repository, the binary components in their release server or attach the component sources with the binary components. Developers need not only the dependency information but also the methods to obtain the components. For example, to develop UserService, ATech needs to define the relations to AccountManager, UserTransaction and ProfileFormat. ATech needs to define also the methods to get binary versions of AccountManager and ProfileFormat from corresponding release sites and to get the component sources of UserTransaction from the repository. As the result, the component developer and deployer need the method to handle with different release policies.

To summarize the problem raised in component development-deployment process, current systems come into these shortcomings:

1. Dependency information defined in the development phase cannot be used in the release and deployment phases automatically.
2. Dependencies in the development phase and the deployment phase can be different because to combine components with another component can reduce the dependency information.
3. SCM systems cannot import sources or binary versions of components developed by different organizations into local development environment automatically and cannot manage the dependency information of each component.

We need a system to manage interrelated components in the development phase and to deploy consistent sets of components to middleware automatically.

4 Approach

The J2DEP research project addresses support to component development-deployment process. This system integrates the functionalities of the configuration management, component development and component deployment together.

The key insight of this research is to manage the evolution of third vendor components inside local configuration management. Rather than to bring and control all versions of third vendor components in the local configuration management, J2DEP supports developers to import the external components to development space mentioned below and generate the dependency information as **dependency metadata**. Then, J2DEP keeps and manages the dependency metadata in the local configuration management instead.

4.1 Development Space

Developers can use J2DEP to import third vendor components into the development spaces shown in Fig. 2 and to generate dependency metadata.

A development space is a directory structure to store source files, dependency metadata and dependent components imported by developer corresponding to

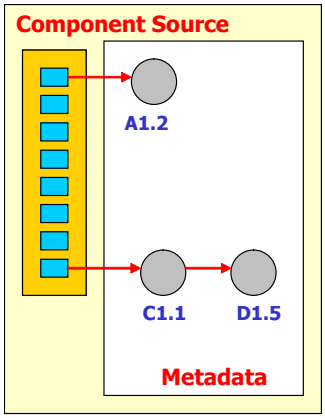

Fig. 2. An example of development space

dependency metadata. In a development phase, a developer can use J2DEP to import third-vendor components into his development space, and then he can control both the sources and the metadata in his local SCM. The details of the development space are shown in Fig. 6 in Sect. 5.

4.2 Dependency Metadata

Dependency Metadata mainly describes details of dependent components (component name, version, vendor, component type, and package type) and the method to obtain components either from source repositories or release sites. A developer can define the relationship among any combination of sources and binary components with dependency metadata.

```
<dependency_component>
  <name>ProfileFormat</name>
  <version>1.2</version>
  <vendor>CyberC</vendor>
  <type>Application Jar</type>
  <packagetype>binary</packagetype>
  <!-- binary package location-->
  <location>www.cyberc.com/release/profileFormat.jar</location>
</dependency_component>
```

Fig. 3. Dependency metadata from a release site

J2DEP supports to generate two kinds of metadata depending on the release policy of each component:

```
<dependency_component>
  <name>UserTransaction</name>
  <version>2.3</version>
  <vendor>DCom</vendor>
  <type>Session Bean</type>
  <packagetype>source</packagetype>
  <!-- source location-->
<location>cvshost.dcom.com</location>
  <cvsroot>/work/cvsroot</cvsroot>
  <authentication>pserver</authentication>
  <tag>UserTransaction-2.3</tag>
</dependency_component>
```

Fig. 4. Dependency metadata from a source repository

- Metadata of a component from a release site, shown in Fig. 3. It consists of component details and URL to download the component
- Metadata of a component from a source repository, shown in Fig.4. It consists of component details, the repository location, the authentication type and the tag name for checking out the component sources.

4.3 J2DEP Architecture

In Fig. 5, we show the overall J2DEP architecture. The development space is where developers place the component sources, dependencies and perform their development. They can use J2DEP to import dependent components by getting sources from organizations that allow accessing the sources in their repositories or by downloading the binary components from the organizations that publish only binary versions. After they fill out the component information to J2DEP, J2DEP will generate dependency metadata into their development space.

J2DEP connects development space with configuration management API (CM API) to manage versions of component sources and their dependencies and connects with release sites to publish components with dependency metadata. To support consistency of component versions in the deployment phase, J2DEP uses the dependency metadata defined in the development phase.

There are two kinds of middleware deployed components.

- Build and test middleware is for developers to deploy the components obtained from the repository and their dependencies. Developers can build binary components and deploy them with dependencies. They are used in the testing phase.
- User middleware is for end-users who deploy only binary components to operate their business requirements and have no relation to the component development process. Deployer can obtain binary components from release sites to deploy in user middleware.

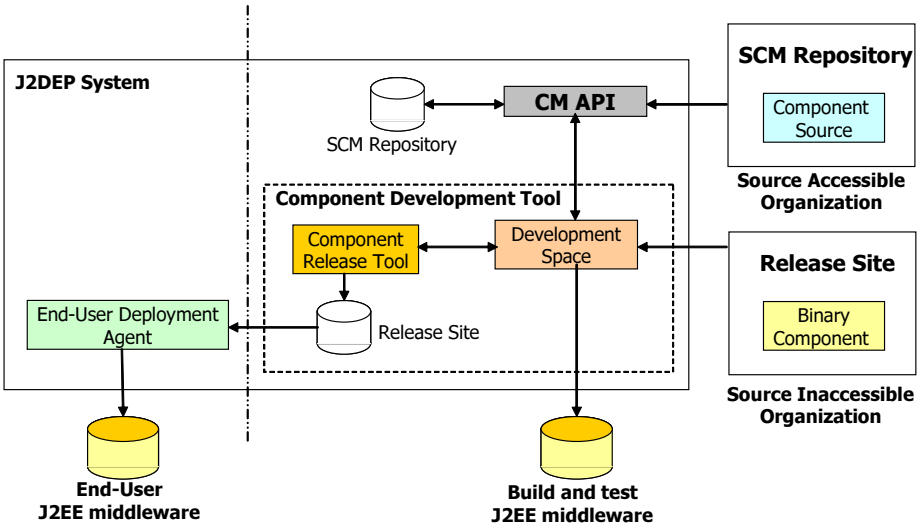

Fig. 5. J2DEP architecture

4.4 Main Functionality

The J2DEP architecture consists of two main parts Component Development Support Tool and End-User Deployment Agent.

Component Development Support Tool This tool supports to generate dependency information for each component and connect a development space with CM API, build and test middleware and release sites.

End-User Deployment Agent This tool supports in the deployment phase to assemble related components to an application component and deploy it in end-users middleware. End-User Deployment Agent will also record the deployed component data to manage dependencies of components on end-user middleware.

5 Implementation

J2DEP supports various kinds of J2EE components based on the architecture described in Sect. 4. J2DEP prototype integrates the development spaces with CVS to manage sources and dependency metadata, HTTP/FTP servers to publish binary components and JBOSS middleware as a platform to deploy J2EE components.

In this section, we show the implementation details of Component Development Support Tool and End-User Deployment Agent. At first, we discuss about Component Development Support. We show how J2DEP supports to create the development spaces, import the related component, generate the dependency metadata, build component, deploy components with dependencies into

the middleware in development sites and release the components to the release site. Then, we show how End-User Deployment Agent can support deployers.

5.1 Component Development Support Tool

Development Space Structure J2DEP supports to build a development space, to create components, to obtain components from the source repository and to import the dependency information of them. In the prototype system, we use the same structure of the development space and the project metadata as Eclipse [17] JDT (Java development tools), so that the developers can continue development with Eclipse easily. The development space structure (as illustrated in Fig. 6) consists of several locations for the following artifacts:

Source directory Component source files, dependency metadata and the encrypted user name and password to connect a source repository or a release site of dependent components.
Binary directory Compiled versions of Java sources.
Dependent component directory Binary versions of dependent components imported by developers to the development space.
Binary version of component Components built from corresponding sources in the source directory.
Project metadata Information about the components in the development space.

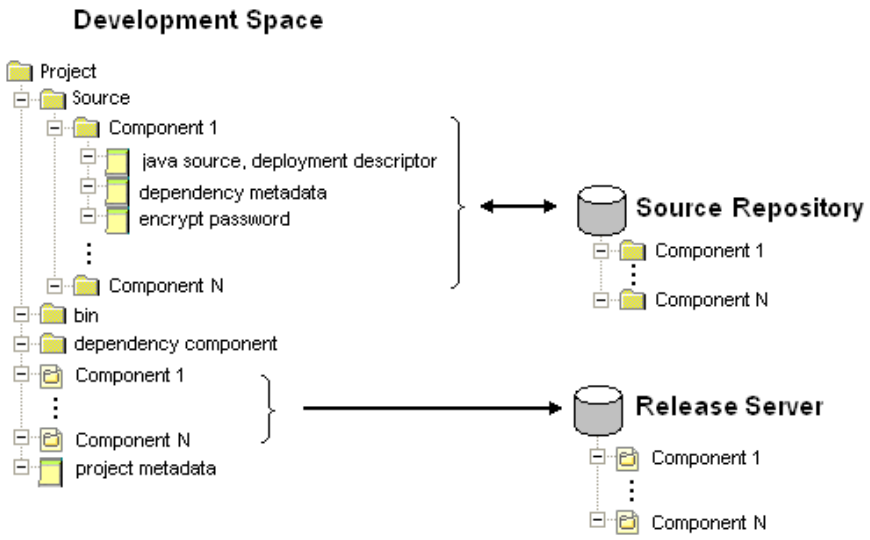

Fig. 6. Development space, source repository and release site

Development Space Initialization Developers can start developing components by initializing a development space. J2DEP helps to create a new development space shown in Fig. 6. They can also open an existing development space from a project metadata file. Then, all components of the development space defined in the project metadata will be opened by the J2DEP tool.

J2DEP connects the development space with a CVS repository shown in Fig. 4. Developers can create new components or pull existing components from source repository to the development space. To create a new component, J2DEP supports to prepare source files and a deployment descriptor for a J2EE component. To pull an existing component from a repository, developers have to inform J2DEP about the component name and version to check out the component by CVS tagging. J2DEP supports to check out sources, dependency metadata and encrypt user name and password file from source repository. Then, J2DEP will get dependent components corresponding to dependency metadata and put into Dependent component directory to prepare for the build and test process in the development phase.

Once, the developers create or check out components into the development space, they can continue development by using normal configuration management procedures (e.g. using CVS).

Dependent Component Import Tool J2DEP helps developers to import the dependent components from either in-house or third vendor developments to the development spaces. The dependent components can be component sources from the SCM repositories or binary components from the release sites.

Developer has to select the methods to import the component from the repositories or the release sites. To import the dependent components, J2DEP supports the developers to import by three methods (1) downloading from release sites (2) checking out from repositories (3) copying from the local computer. J2DEP will import the components into Dependent Component Directory. Developers need to inform J2DEP about the component details and the connection details shown in Fig. 7 and Fig. 8.

After developers fill out the form, J2DEP generates the dependency metadata, like in Fig. 3 and Fig. 4, for the imported components. For the user name and the password to connect a repository or a release server, J2DEP will encrypt them and generate a new file separated from dependency metadata.

Notice that, the developers need to add the next level dependencies manually, when the component does not include any metadata (e.g. the component is not built by J2DEP).

Component Packaging and Developer Site Deployment Support When a developers finish developing by using the normal configuration management procedures, J2DEP supports to build the Java source files and package the compiled sources into a binary component automatically.

J2DEP supports to generate the default J2EE deployment descriptor and assemble the related components in Component Dependency Directory of the

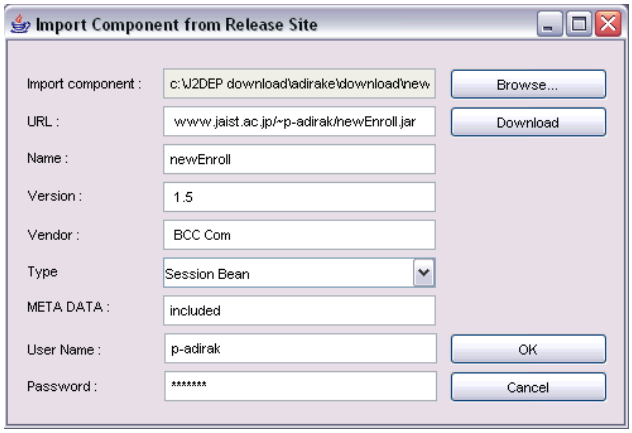

Fig. 7. The import tool for binary components from release sites

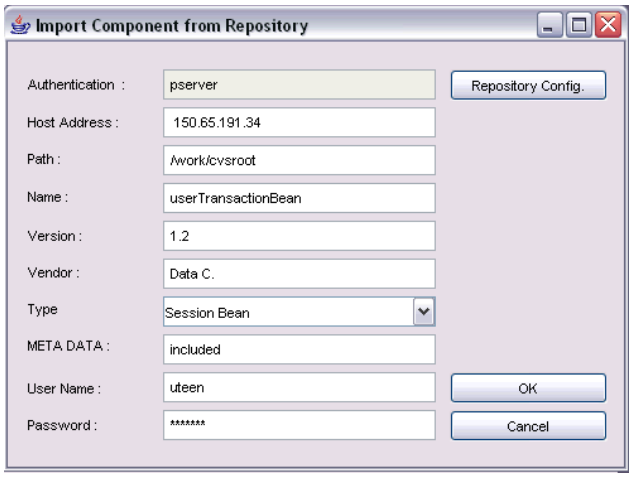

Fig. 8. The import tool for component sources from repositories

development space into a single J2EE module. They can deploy or redeploy a J2EE module to their middleware server to perform the testing phase. They can use this tool to undeploy components and their dependencies from middleware.

Component Release Tool After developers finish testing the components in their development space, they can release the binary components to release sites. J2DEP supports developers to connect to FTP/HTTP servers and to upload the binary versions of components. Developers or end-users can download the components from the release sites by Developer Deployment Support or End-User Deployment Agent.

In the release phase, the developers need to select components and combine them into a new component for releasing. If they select a component to be included in other component package, the dependency on the component is removed. For example in Fig. 9, the dependency metadata between UserWebComponent and transactionBean is removed because transactionBean becomes a part of UserWebComponent.

The developers also need to specify the release policy for each component that contains sources. If they do not want to release component sources, they have to inform J2DEP about the location to upload the binary component.

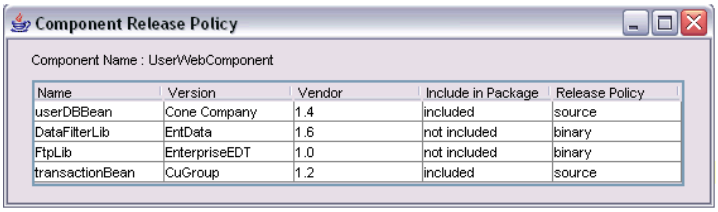

Fig. 9. Component release tool

5.2 End-User Deployment Agent

End-User Deployment Agent supports to deploy and undeploy the binary version of components to the end-user middleware.

Deploy support This function downloads components and their dependent components to middleware. To deploy components, the deployers have to inform the component name, version and location to download each component to End-User Deployment Agent. After finishing downloading components, End-User Deployment Agent generates the J2EE default deployment descriptor and assemble every component into an application. Then, it records the components that they have already deployed into a log file. When deployers want to deploy again, the existing components in middleware are not downloaded again.

Undeploy support This function removes components and their external dependencies. To maintain the consistency of the components in middleware, End-User Deployment Agent can remove only the components that are not shared by other components

6 Related Works

J2DEP is built to connect component development and deployment processes. There are several tools to support these processes. We discuss some integrated development environments (IDEs) such as Eclipse [17], NetBeansIDE [18] and JBuider [19]. We also discuss RPM [16] and TWICS [10] for component deployment support.

We compare the functionalities of each tool in development-deployment process in Table 1. These IDEs support the development process of Java including J2EE applications. They can support J2EE development by several kinds of plugins. Developers can use these tools to create various kinds of J2EE components, manage the component sources in their repositories, and test them. Although these tools provide the interface to connect the development environment with an SCM repository, the dependencies of components are not well resolved and managed in the repository. Developers have to manage the dependency information in the local configuration management manually.

Table 1. Functionalities of J2DEP and related tools

	IDEs	RPM	TWICS	J2DEP
Connect the tool with the development space	◯	×	$\triangle_2$	$\triangle_4$
Generate dependency information	×	$\triangle_1$	$\triangle_3$	◯
Connect the tool with SCM	◯	×	◯	◯
Manage dependency information in SCM	×	×	◯	◯
Component release support	×	◯	◯	◯
Deploy components to the development space	◯	◯	◯	◯
Deploy components to end-users' sites	×	◯	◯	◯
Deploy components with dependencies	×	×	$\triangle_2$	◯

◯= supported, ×= not supported, $\triangle_=$ partially supported

$\triangle_1$ Generate the dependency but no detail about the component location.

$\triangle_2$ Support only components built by TWICS.

$\triangle_3$ Generate the dependency information only in the release phase.

$\triangle_4$ The target space must be the same as Eclipse development space in the development phase.

7 Conclusion

In this research, we proposed J2DEP, the integration of component-based development-deployment and software configuration management in middleware. This system is designed to support managing the interrelation of in-house components and third vendor components. J2DEP links between component development and component deployment. The dependency metadata defined in the development phase and component sources can be managed in local configuration management. J2DEP uses dependency metadata to support the build and release processes for developers. J2DEP also helps a developer or a deployer to deploy consistent sets of components in either developer site or user site.

Currently, J2DEP lacks some features for the component development-deployment support. For example, developers have to merge different versions of dependency metadata manually. We need to implement these features. In the current prototype, J2DEP is a tool separated from the other development tools. In the future, it would be advantage to implement J2DEP as the other development tool plug-ins, for example, Eclipse or NetBeanIDE. This would allow developers to gain the benefit from all infra-structure provided by the other tools.

References

1. Inderjeet, S., Stearns, B., Johnson, M.: Designing Enterprise Applications With the J2Ee Platform. Addison-Wesley (2002)
2. Corporation, M.: Microsoft .NET technical resources. `http://www.microsoft.com/net/technical/` (2004)
3. Object Management Group, Inc.: Common Object Request Broker Architecture: Core Specification. formal/04-03-12 edn. (2004)
4. Whitehead, K.: Component Based Development: Principles and Planning for Business Systems. Addison-Wesley (2002)
5. Cervantes, H., Hall, R.S.: Autonomous adaptation to dynamic availability using a service-oriented component model. In: Proceedings of 26th International Conference on Software Engineering (ICSE'04). (2004) 614–623
6. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. 2nd edn. Addison-Wesley (2002)
7. Schmidt, D.C., Vinoski, S.: The corba component model: Part 1, evolving towards component middleware. C/C++ Users Journal (2004)
8. Weber, D.W.: Requirements for an scm architecture to enable component-based development. In: Proceedings of the 10th International Workshop on SCM. (2001)
9. Edwards, S.H., Gibson, D.S., Weide, B.W., , Zhupanov, S.: Software component relationships. In: the 8th Annual Workshop on Institutionalizing Software Reuse. (1997)
10. Sowrirajan, S., van der Hoek, A.: Managing the evolution of distributed and interrelated components. In: Proceedings of the 11th International Workshop on SCM. LNCS 2649, Springer-Verlag (2003) 217–230
11. Larsson, M., Crnkovic, I.: Configuration management for component-based systems. In: Proceedings of the 10th International Workshop on SCM. (2001)

12. Free Software Foundation, Inc.: RCS. http://www.gnu.org/software/rcs/rcs.html (2003)
13. Cederqvist, P., et al.: Version management with CVS for cvs 1.11.17. http://www.cvshome.org/docs/manual/ (2004)
14. Hall, R.S., Heimbigner, D., Wolf, A.L.: A cooperative approach to support software deployment using the software dock. In: Proceedings of the 21st International Conference on Software Engineering (ICSE'99). (1999) 174–183
15. van der Hoek, A., Hall, R.S., Heimbigner, D., Wolf, A.L.: Software release management. In: Proceedings of the 6th European Software Engineering Conference (held jointly with the 5th ACM SIGSOFT international symposium on Foundations of Software Engineering). (1997) 159–175
16. The RPM community: www.rpm.org homepage. http://www.rpm.org/ (2002)
17. Eclipse Foundation: eclipse.org. http://www.eclipse.org/ (2004)
18. netBeans.org: NetBeans IDE. http://www.netbeans.org/products/ide/ (2004)
19. Borland Software Corporation: Borland JBuilder. http://www.borland.com/jbuilder/ (2004)