



CHAPTER 2

CONCEPTS



OBJECTIVES

After completing "Concepts," you will be able to:

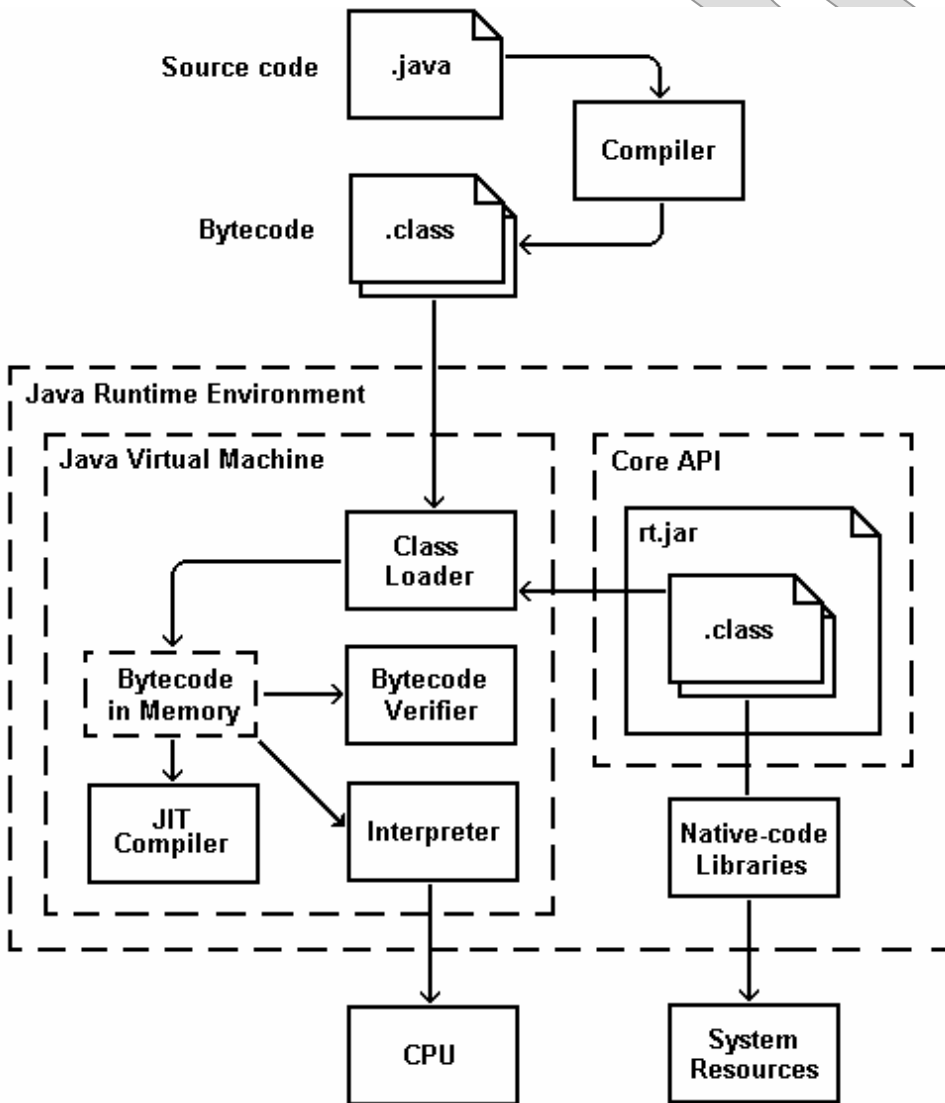
- Describe the importance of intermediate bytecode to the portability of Java software.
- Describe the J2EE architecture of containers and components, and identify the three containers that must be supported by a J2EE application server.
- Explain the lifecycle of a J2EE application and the roles that various people play in development, deployment, and administration.
- Understand the power of declarative development, using XML to declare facts and parameters to an application server so that it can implement many enterprise features "for free."

Bytecode

- Java software, deployed in a single form or version, runs on a wide range of platforms – this is **portability**.
- It is a hybrid of two traditional approaches to high-level languages:
 - **Compiled** languages enjoyed build-time type checking and other error checking, but had to be **cross-compiled** for different target platforms.
 - **Interpreted** languages could be portable, but relied on run-time error checking, a frustrating and inefficient development process.
- Java is **compiled and interpreted**.
 - Source files are compiled to intermediate **bytecode**, which is very compact like the machine languages that traditional compilers write.
 - But then the bytecode is interpreted by the **virtual machine** – which is just what it sounds like!

From Source Code to CPU

- Here is a summary of development, deployment and execution of a Java program:



The Virtual Machine and Runtime

- The virtual machine, or **JVM**, loads compiled Java classes and interprets them in a machine-specific way.
 - Resulting machine code is fed to the actual processor in use at runtime.
- The Java runtime environment, or **JRE**, includes the JVM and the **Core API**.
 - This is the large library of classes that Java applications and components can use to do their work.
 - The Core API is loosely analogous to the standard C and C++ libraries, or the Visual Basic runtime.
 - A key difference is that C programs may use the standard libraries to, say, write a file; Java programs must use the Core API, as the virtual machine prevents direct use of the CPU or OS API.
 - This is both a way of fostering good practice and a pillar of Java's code-security architecture.

- In **Examples\Hello**, there is a dead-simple Java application that simply prints a greeting to the console.
- The source code is as follows, though we'll not worry about any details of Java coding in this course.

```
public class Hello
{
    public static void main (String[] args)
    {
        System.out.println ("Hello, Java!");
    }
}
```

- What we will consider for the moment:
 - How to build and test the application
 - How the application works for a given platform

- There are two simple scripts for this demo:
 - A build script (build.bat or build.sh) to compile the source code in .java files to bytecode in .class files
 - A run script (run.bat or run.sh) to launch the application class in a new JVM
- Open a console and change working directory to **Examples\Hello**.
- View a listing of the files in that directory:

```
dir (or ls)
build.bat
build.sh
Hello.java
run.bat
run.sh
```

- Run the build script. View the directory listing again, and you'll see the new file Hello.class.
- Run the run script and see that the application class runs and prints the greeting back to you:

```
run
Hello, Java!
```

How Does It Work?

- How does this application work on your platform?
- Java software achieves portability on two levels.
- Rather than compiling to machine code for a specific CPU, Java is compiled to portable bytecode.
 - Here's a listing of the bytecode in Hello.class:

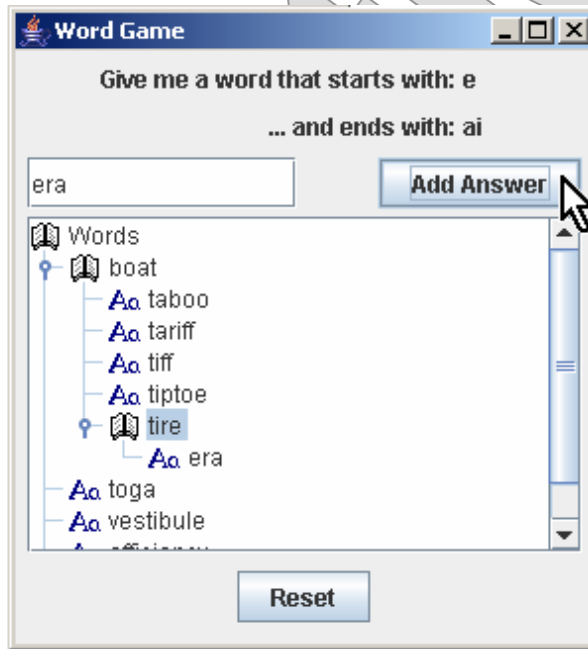
```
0:  getstatic    #2; // gets the printing stream
3:  ldc         #3; // loads the greeting string
5:  invokevirtual #4; // calls the print method
8:  return
```
 - This is interpreted to machine instructions, say for a Pentium chip, at runtime, by the JVM.
 - It would of course be interpreted differently for a RISC processor or a Sun Alpha - etc.
- Where the application needs to interact with the physical system, it makes Core API calls such as `System.out.println` to write to the console output.
 - The Core API implementation for your platform holds the logic for connecting to the console's standard output stream and producing characters to that stream.
 - The operating system is ultimately responsible for implementing the rendering of characters on the screen.

A Java GUI Application

EXAMPLE

- In **Examples\WordGame** is an example of a graphical user interface (GUI) application: a word game!
 - Again, build and run using prepared scripts:

```
build
run
```



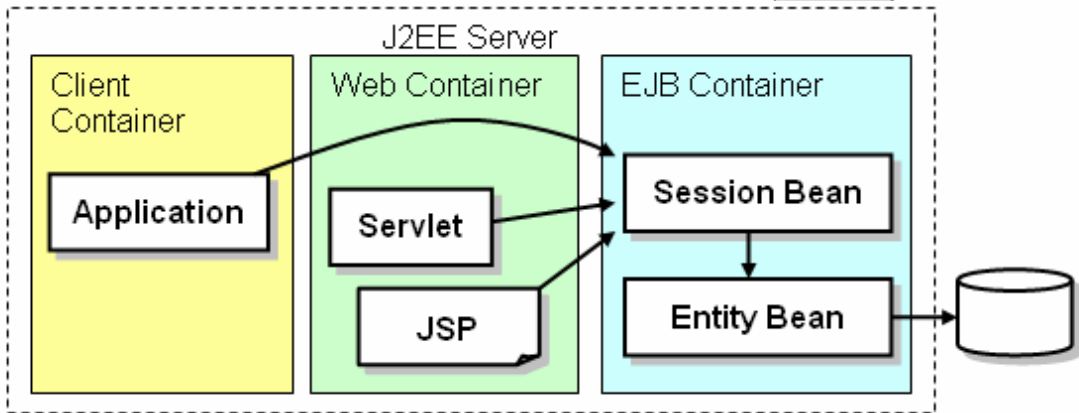
- The Java runtime loads a main class that uses the Core API to interact with the OS' windowing system.
- Specifically, this application uses the **Java Foundation Classes**, or **JFC**.

Containers and Components

- From basic code portability and the capacity to create standalone applications, we now move to key concepts in the J2EE architecture.
- The organizing principle of J2EE is the **container**.
 - A container holds objects or **components**.
 - The container is responsible for the **lifecycle** of the component: i.e. it creates and destroys the component (and others like it) by the container's own policies.
 - The container also mediates: it "sees" every interaction between the component and the outside world.
- By assuming this control, the container is able to provide many useful features:
 - **Remote connectivity**
 - **Scalability**
 - **Availability**
 - **Security**
 - **Transaction support**
- All these features come "for free" to the component developer, as they can be implemented once in the container.
 - We'll see how in a moment.

Three Containers

- In its current version, J2EE defines three types of container:



- The **Web** container supports Servlets and JSPs.
- The **EJB** container is exclusively for EJBs.
- The **client application** container is the least used; it supports otherwise ordinary standalone applications, but gives them visibility to J2EE APIs so that they can, for instance, invoke an EJB.
- A J2EE **application server** is required to support all three containers.

Roles

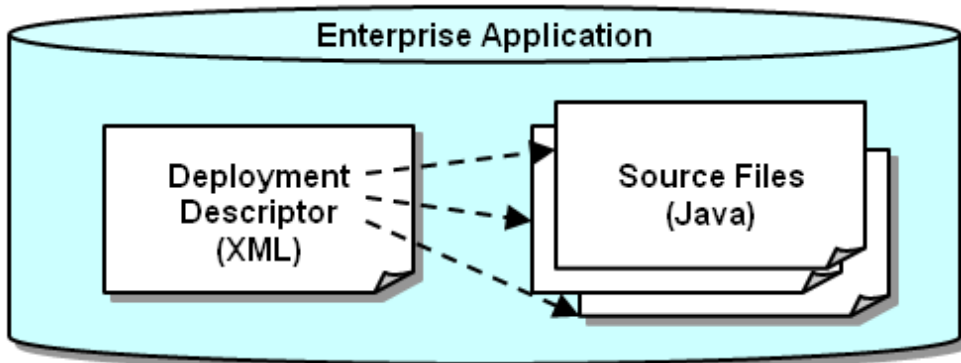
- J2EE also promotes a really inspired idea: recognizing that software development is carried out by people playing various **roles**.
 - On one hand, we knew this!
 - But J2EE formalizes these roles, and gives different responsibilities and privileges to each.
 - This facilitates good workflow and development practices.
 - It also organizes certain logic or key values in the application structure so that they can be managed most appropriately.
- **The four most common roles are:**
 - The **component provider** builds a given component.
 - The **application assembler** creates a comprehensive application from a set of components and possibly "glue" code.
 - The **deployer** installs the application on a given physical system.
 - The **administrator** manages the application in his or her domain.
- **Two more roles are played only by vendors of J2EE application servers:**
 - **Container provider**
 - **Server provider**

Declarative Development

- J2EE developers quickly notice a pattern in application structure: their Java source files exist alongside XML files that state certain facts about the Java classes.
- Thus the architecture recognizes that not all facts should be coded in a programming language.
- Some information is better defined in XML – a format that
 - Is easy to read and write without specialized tools
 - Is highly expressive, able to capture information sets of many shapes and sizes
 - Requires no compilation or other transformation to be shared between all J2EE roles
- Thus only the component provider (and maybe the application assembler) write Java code – but everyone works with XML.

Deployment Descriptors

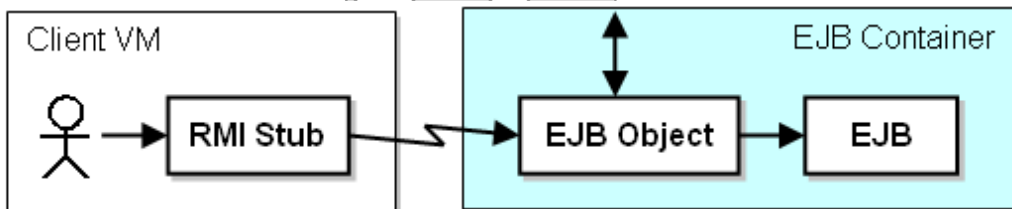
- The XML files are called **deployment descriptors**, and they capture many kinds of information:



- What Java classes are to be treated as what kinds of EJB components
- Where components should be published: a URL for a servlet, or a unique name for an EJB
- Who should be able to use what components, i.e. security information
- Various other values that are **volatile** or deployment-specific: local threshold values or network names, for instance

Remote Connectivity

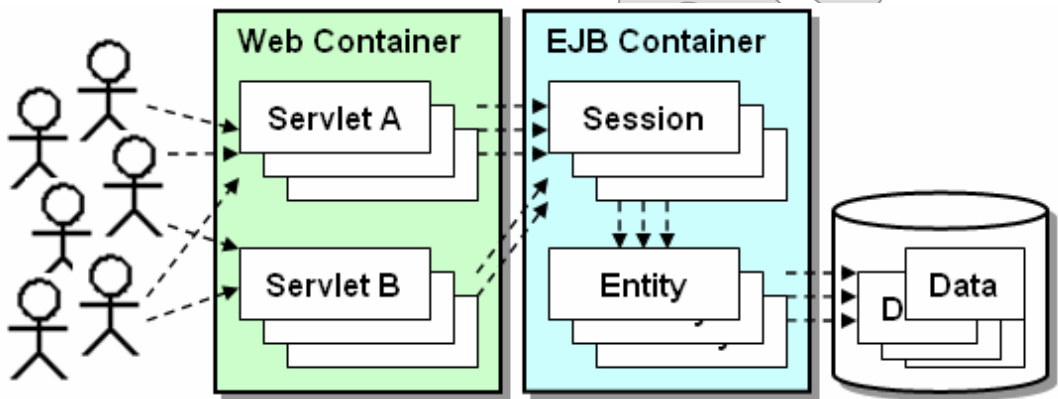
- J2EE components enjoy remote connectivity, implemented for them by the container.
- Servlets and JSPs are available through HTTP requests, thanks to interactions between the Web server and a J2EE Web container.
- For EJBs, J2SE defines a **remote method invocation (RMI)** standard.
 - A subset of RMI that supports CORBA, **RMI over IIOP** or **RMI/IIOP**, is the default transport for EJBs.



- All J2EE components thus enjoy remote connectivity and **location transparency**, which means that components and their clients are blissfully ignorant of their true proximity:
 - They could be in the same process space.
 - They could be in different processes on one machine.
 - They could be flung halfway around the world and talking via the Internet.
- No code in the component or client has to change to adapt to these different proximities.

Scalability

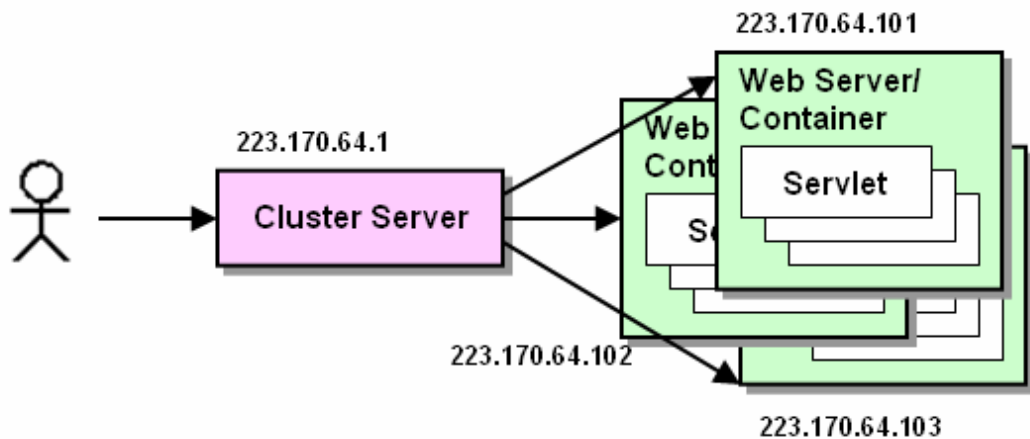
- Containers can provide excellent scalability on behalf of their hosted components.
- The key technique in achieving scalability is **pooling**.



- The container will manage a pool of instances of a given component type.
 - The size of the pool is controlled by the administrator.
 - To the outside world, it looks as though there is one component, or N components; but in fact there may be M components in memory.
 - Components must be able to **switch contexts** to live in pools; this places a few constraints on their implementation.
 - Especially, they cannot make assumptions about their own **identity** that would be safe in the J2SE context.
- **By this arrangement, the container is able to serve hundreds or thousands more clients than it keeps service objects in memory – or to manage much more persistent data than it holds in memory at one time.**

Availability

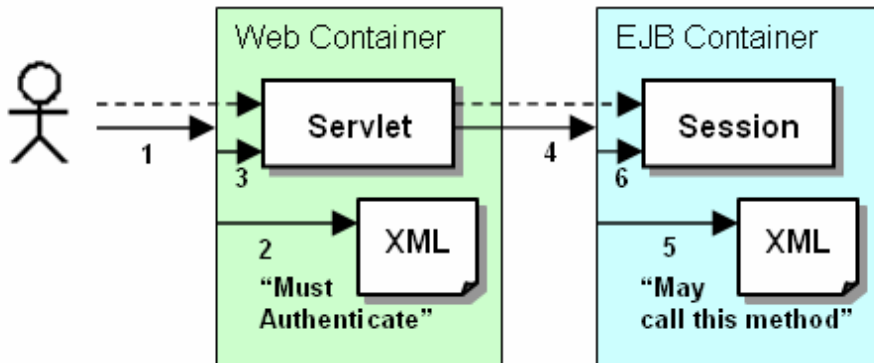
- Closely related to scalability is availability – the property of a component that it is always available for client use.
- But servers crash, and network connections fail.
- J2EE application servers can support **clustering**, a technique by which multiple processes on multiple networked hosts run redundantly.



- A cluster is far more fault-tolerant than a single machine.
- The trick is making the cluster of server processes, and the pools of objects within them, look like one component at one address to the client.
- A **cluster server** accomplishes this.
- Clustering is one of the least-specified J2EE features; it is supported as a standard, but clustering strategies may be proprietary.

Security

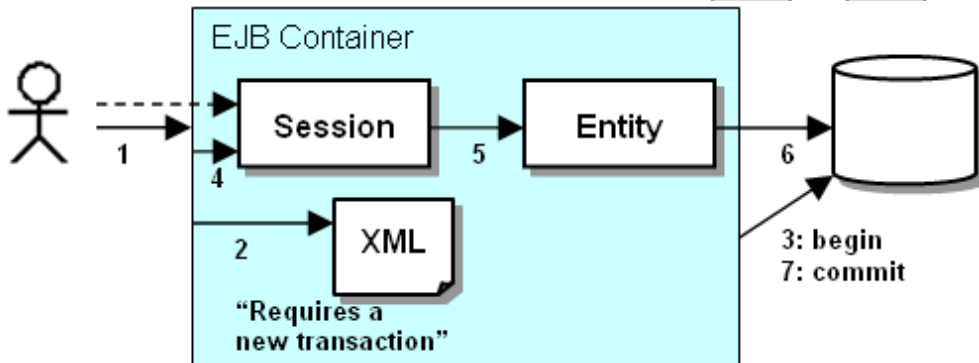
- J2EE defines security mechanisms that assure that components will not be used by unauthorized clients.



- **Authentication** mechanisms are standardized primarily at the HTTP front door to an application.
 - A Web application can declare a requirement for HTTP basic or even certificate-based authentication.
- **Authorization** policies exist at many levels.
 - An authenticated client may be granted or denied privileges to see a certain Web page.
 - An EJB may authorize callers at component or even individual method level.
- J2EE deployment descriptors can define security **roles** to which users and groups can be assigned.
 - These roles would be defined by an application assembler.
 - But since the actual users and groups will vary by locale, they are assigned to these roles by an administrator.

Transactionality

- For EJBs only, there is also declarative support for ACID transactions.



- The EJB container reads declarations in the deployment descriptor that inform it as to how to start and end transactions as calls come in to an EJB at runtime.
 - A given method may require a transaction to be in force; require its own, new transaction; allow enlistment in any existing transaction, refuse to participate in a transaction, etc.
 - The container, as it watches calls come in and responses go out, interacts with one or more external **resource managers** to assure that transactions are started, ended, and rolled back appropriately.
- This is a massive savings of development time and trouble.

- Consider the EJB deployment descriptor in **Examples\ElectronicDJ**.
 - We're just scratching the surface of this example right now; we'll study it in increasing detail through the rest of the course.
 - See the file **ejb-jar.xml**.
- The deployment descriptor is packed into the application along with the compiled Java classes.
- It tells the application server everything it needs to know in order to manage the EJBs in this application correctly.
- We'll consider just a few fragments of this large file.

- The descriptor declares to the container that the application contains several EJBs.
 - Here's the declaration of an **entity bean** that manages persistent data about music in the database:

```
<ejb-jar version="2.1" ... >
  <enterprise-beans>
    <entity>
      <description>...</description>
      <display-name>AlbumEJB</display-name>
      ...
    
```

- Here's a `session bean` that interacts with clients that want to add certain songs to their collections:

```
<session>
  <description>...</description>
  <display-name>DJ</display-name>
  <ejb-name>DJ</ejb-name>
  <home>cc.music.DJHome</home>
  <remote>cc.music.DJ</remote>
  <ejb-class>cc.music.DJEJB</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container
  </transaction-type>
  ...

```

- This last piece tells the container which Java class files make up the **DJ** bean: an implementation class, a business ("remote") interface, and a factory ("home") interface.

- Much later in the file, we see definitions of security information:

- Security roles ...

```
<assembly-descriptor>
  <security-role>
    <description>Administrator...</description>
    <role-name>WebDJAdmin</role-name>
  </security-role>

  <security-role>
    <description>A person...</description>
    <role-name>WebDJAdult</role-name>
  </security-role>
```

- ... and authorization policies ...

```
<method-permission>
  <role-name>WebDJAdmin</role-name>
  <method>
    <ejb-name>AlbumTestEJB</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>addAlbum</method-name>
    <method-params>
      <method-param>int</method-param>
      <method-param>...String</method-param>
      <method-param>...String</method-param>
    </method-params>
  </method>
  ...
```

- Finally, note the transactionality declared for the **DJ** bean:

```
<container-transaction>
  <method>
    <ejb-name>DJ</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>RequiresNew
  </trans-attribute>
</container-transaction>
```

- This tells the container to start a new transaction that isolates each incoming call to any method on the bean.
- This transaction will enlist any database actions taken from when the method is called to when it returns.

SUMMARY

- The basic value proposition of Java – portable, object-oriented software – provides the foundation for J2EE.
- Then, J2EE brings several brilliant inventions (or discoveries!) of its own:
 - The **container** and contained component
 - **Context and lifecycle** relationships
 - **Declarative development** using XML
 - **Roles** as a way to define distinct responsibilities and to allow multiple parties to collaborate smoothly over the complete lifecycle of an enterprise component or application