Software to Make Building Application System Easier - Seasar2 and S2 Family

The Seasar Project

March 25, 2005

Translated by: H.Ozawa

What Is “Seasar”?

- Features of Seasar
  - It’s an Open Source software（SSL1.1 license）
  - Pure Java
  - DI container + AOP framework
  - Pronounced “see” “sir”
  - Seasar is a mystical dog-like creature in Okinawa (Japan) - Developer of Seasar, Yasuo Higa, is from Okinawa
- Goals of Seasar
  - Reconstructure J2EE to make it more developer friendly
    （Only use the best parts of J2EE and make it lighter)
  - Offer “ease” and “friendliness” to developers

Seasar's History

- Years 2001 to 2　pre-Seasar – Higa was working on an original J2EE server, JTA, connection pooling, etc.
- Years 2002 to 3　initial Seasar version (S0) - Developed online foreign currency exchange system using Tomcat and iBATIS for StarLogic Inc. (WEB+DB PRESS Vol.15,16)
- Year 2003　Seasar V1 – all in one J2EE server
  includes Flowlet, Rulet, Sqlet, Jetty, HSQLDB, GPSS, Eclipse plugin
- Year 2004/4　Seasar2 – evolved to DI container
  restructured former functionalities to be supported by DI

Advantages of Components and DI

- Want to increased application system development productivity
- Want better component reusability support
- But need more complex component than just a simple class
- What to do?

- Use interface
  - separates specification from implementation
  - define specification as an interface
  - define implementation as as an “implements”

Translated by: H.Ozawa

Advantages of Components and DI

- Define Calculator interface
- This interface contains specification of method multiply

```
public interface Calculator {
  public int multiply(int source, int by);
}
```

Advantages of Components and DI

- Calculator interface is still only a specification, so we'll implement it
- Implement as a CalculaMachine class

```
public class CalculaMachine implements Calculator {

  public int multiply(int source, int by) {
    int ret = 0;
    for (int i = 0; i < by; i++) {
      ret = ret + source;
    }
    return ret;
  }
}
```

Translated by: H.Ozawa

Advantages of Components and DI

- Will try using CalculaMachine class

```
public class Sample {

  public static void main(String[] args) {
    Calculator calc = new CalculaMachine();
    System.out.println(calc.multiply(10,100));
  }
}
```

Declare variable "calc" of type "Calculator", and substitute CalculaMachine class entity with "calc"

```
C:\>java Sample
1000
```

Advantages of Components and DI

- CalculaMachine class is not quite satisfactory so create a different class, CalcMachine, that implements a Calculator interface.

```
public class CalcMachine implements Calculator {

  public int multiply(int source, int by) {
    return source * by;
  }
}
```

Advantages of Components and DI

- Will use CalcMachine class

```
public class Sample {
  public static void main(String[] args) {
    Calculator calc = new CalcMachine();
    System.out.println(calc.multiply(10,100));
  }
}
```

Declare variable "calc" of type "Calculator", and substitute CalcMachine class entity with "calc"

Note that only implementation was changed

```
C:\>java Sample
1000
```

Translated by: H.Ozawa

Advantages of Components and DI : Unit Test

- **Trouble, if there is an error in the class implementation!**
  => need assurance the module satisfies the specification
  = unit test

■***Implementation class*** *to test*

```
public class CalcMachine implements Calculator {
  public int multiply(int source, int by) {
    return source * by;
  }
}
```

■*Unit test code*

```
public class CalcMachineTest extends TestCase {
    public void testMultiply() {
            CalcMachine calc = new CalcMachine();
            int ret = calc.multiply(10,100);
            assertEquals(ret,1000);
    }
}
```

Translated by: H.Ozawa

Advantages of Components and DI

- What one further functionality…

Want to exchange without modifying code each time!

```
public class Sample {
  public static void main
    Calculator calc =                                    ;
    System.out.println(calc.multiply(10,100));
  }
}
```

Using a DIContainer

- Move component configuration to an external file

```
public class Sample {
  public static void main(String[] args) {
    S2Container container =
        S2ContainerFactory.create(PATH);
    Calculator calc =
        (Calculator)container.getComponent("calc");
    System.out.println(calc.multiply(10,100));
  }
}
```

Start container (similar to starting JVM)

Register a class in a container

Invoke a method using registered class as an endpoint.

Translated by: H.Ozawa

Using a DIContainer

- Configuration file（.dicon file）

```
<?xml version="1.0" encoding="Shift_JIS"?>
<!DOCTYPE components PUBLIC "-//SEASAR//DTD S2Container//EN"
    "http://www.seasar.org/dtd/components.dtd">
<components>
    <component name= "calc" class= "CalcMachine">
    </component>
</components>
```

By just changing here, it is possible to exchange classes without modifying any code

Translated by: H.Ozawa

What does a DI Container Do?

- Dependency Injection
  - Remove dependencies between components during development. More concretely,
    - (1) Remove new (independent of implementation class)
    - (2) Remove instantiation of beans
  - A container injects dependencies during runtime
    - (1) Instead of “new”, the container creates a instance of a variable
    - (2) The container instantiates beans
- Components associates with other components only through their interfaces. In other words, **only interfaces are necessary.**
- DI container associates components according to the configuration file
  - Dependencies are dynamically constructed during runtime

Translated by: H.Ozawa

The Most Important Point of DI

- By separating of interface and implementation, decouple dependencies between implementation classes
- Merits
  - Better maintainability
  - Better quality
  - Lessen development time
  - Improved reusability

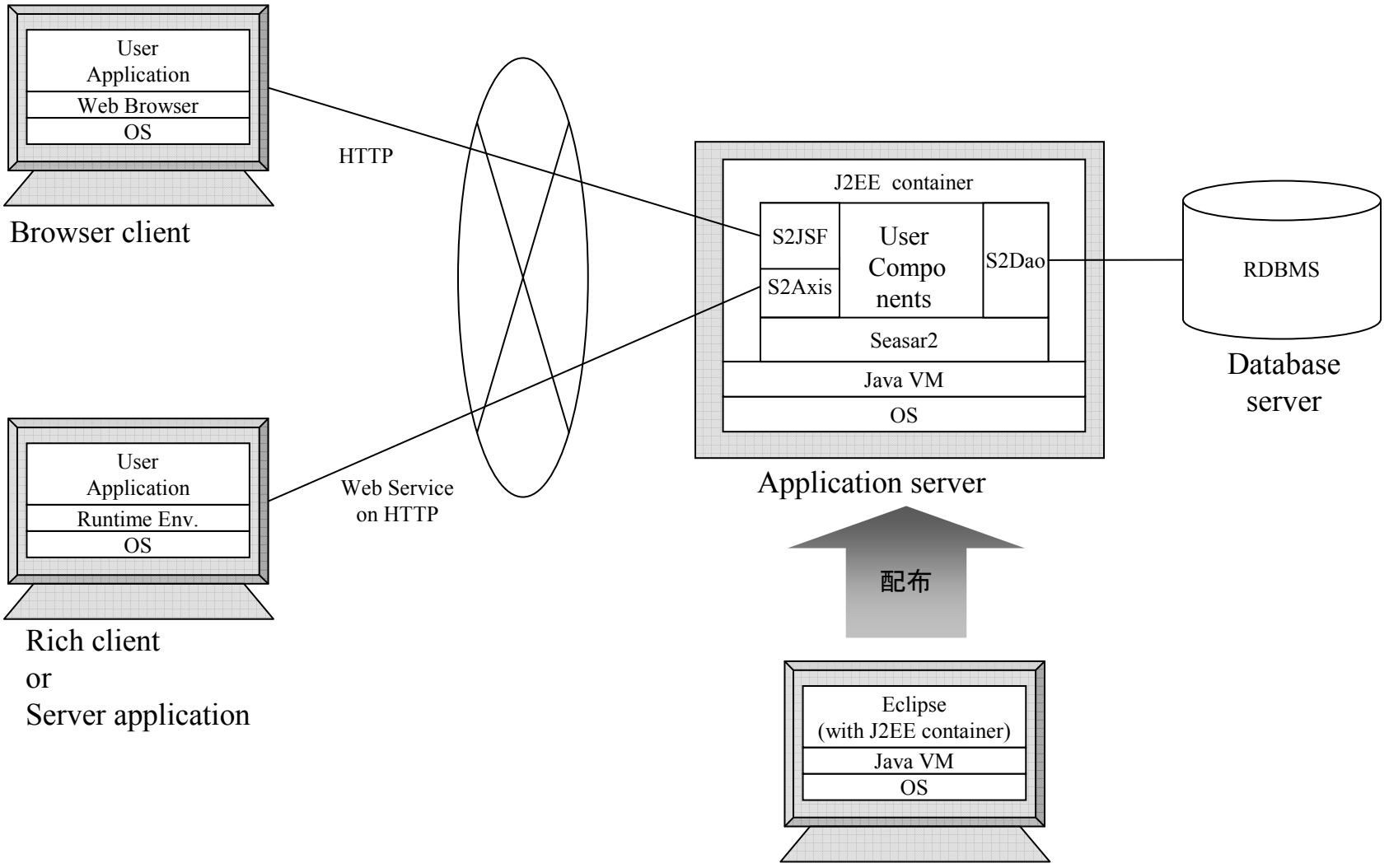
Translated by: H.Ozawa

S2 Family

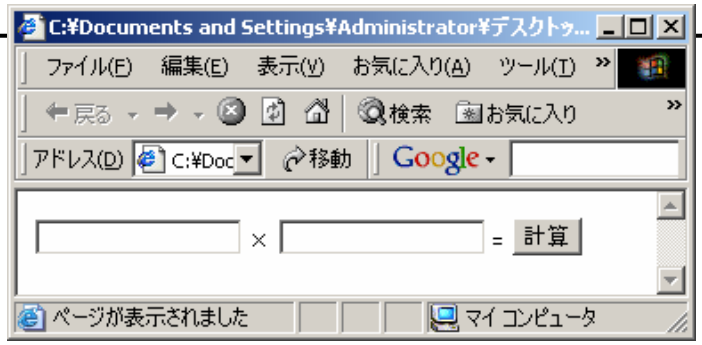
- **Don't want to be concerned about bootstrapping a container!**
- **Can't create a system just by components**

- ***S2JSF Assist develop HTML based systems***
  - Don't have to know Servlet and JSP
  - Automatically set request/response parameters to a POJO
  - Write methods to invoke within HTML
- ***S2Axis Wrap web service***
  - Seamlessly call POJO on a remote server
- ***S2Dao Access RDBMS***
  - Automatic mapping of RDBMS rows and POJOs
  - Most SQL statements are generated
  - Complex SQL statements may be written in an external file

Architecture



Translated by: H.Ozawa



■Create HTML file

```
<html xmlns:m="http://www.seasar.org/maya">
<head>
<meta http-equiv="Content-Type" content="text/html" />
</head>
<body>
<form>
  <span m:inject="h:messages" m:globalOnly="false" m:showDetail="true"/>
  <input type="text" m:value="#{dto.source}"/> *
  <input type="text" m:value="#{dto.by}"/> =
  <span m:value="#{dto.result}"/>
  <input type="submit" value="Calculate" m:action="#{calcAction.execute}"/>
</form>
</body>
</html>
```

S2JSF

```
public class Dto implements Serializable {

  private int source;
  private int by;
  private int result;

  public Dto() {
  }

  public int getSource() {
    return source;
  }
  public void setSource(int source) {
    this.source = source;
  }
  public int getBy() {
    return by;
  }
  public void setBy(int by) {
    this.by = by;
  }
  public int getResult() {
    return result;
  }
  public void setResult(int result) {
    this.result = result;
  }
}
```

```
public class CalcActionImpl implements CalcAction {

  private Dto dto;
  private Calculator calc;
  public CalcActionImpl() {
  }
  public void setDto(Dto dto) {
    this.dto = dto;
  }
  public void setCalculator(Calculator calc) {
    this.calc = calc;
  }

  public String execute() {
    dto.setResult(calc.multiply(dto.getSource(), dto.getBy()) );
    return null;
  }
}
```

```
<components>
  <component name="dto" class="Dto" instance="request"/>
  <component name="calcAction" class="CalcActionImpl" instance="request"/>
  <component class="CalcMachine"/>
</components>
```

Translated by: H.Ozawa

S2Axis

■Register web service as a POJO on S2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE components PUBLIC "-//SEASAR2.1//DTD S2Container//EN"
    "http://www.seasar.org/dtd/components21.dtd">
<components>
    <component name="calc" class="CalcMachine">
        <meta name="s2-axis:service">
            <component class="org.seasar.remoting.axis.ServiceDef">
                <property name="serviceType">
                    @Calculator@class
                </property>
            </component>
        </meta>
    </component>
</components>
```

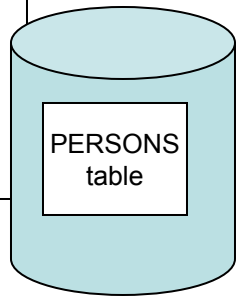
■Invoked from C#


```
private void button1_Click(object sender, EventArgs e)
{
    //  get Web service
    CalculatorService calc = new CalculatorService();
    //  invoke a method
    int ret = calc.multiply(  Convert.ToInt32(textBox1.Text)
                             ,Convert.ToInt32(textBox2.Text)
                           );
}
```

S2Dao

■RDBMS tables are as follows:

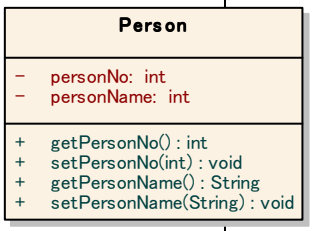
```
CREATE TABLE PERSONS
 ( PersonNo INTEGER, PersonName VARCHAR(100) );
```



■Define a class to hold records

```
public class Person implements Serializable {

  // associate with a table in RDBMS
  public static final String TABLE = "PERSONS";
 // Define fields to hold each column
  private int personNo;
  private String personName;
     :
  // setter/getter and so on…
}
```

to next page



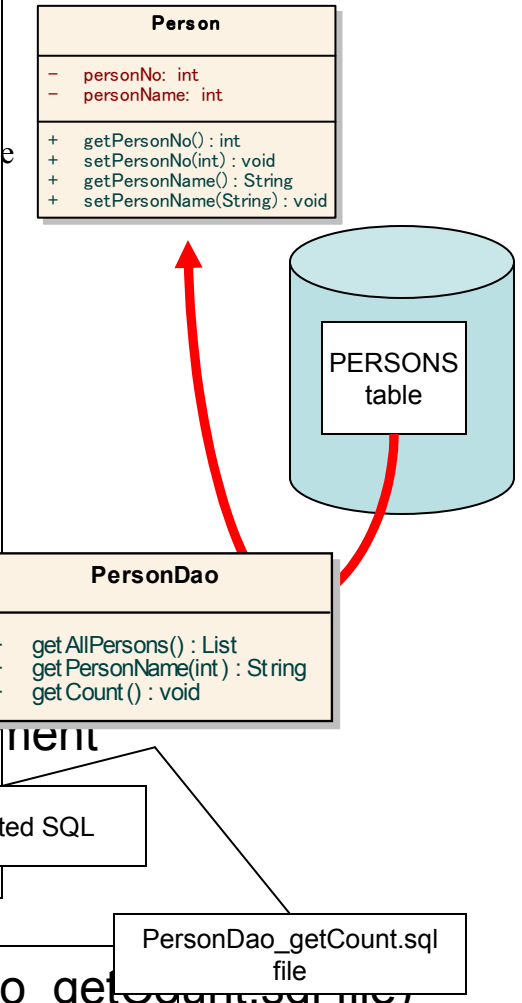
Translated by: H.Ozawa

S2Dao

■Create DAO（Data Access Object）

```
public interface PersonDao {

  // associate with a class to hold records
     public Class BEAN = Person.class;

  //  following method generates SQL statements
     public List getAllPersons();
     public String getPersonName_ARGS = "personNo";
     public String getPersonName(Integer personNo);

  // following method executes an user defined SQL statement
     public int getCount();
}
```

from previous page



■File containing user defined SQL statement (PersonDao_getCount.sql file)

```
SELECT count(*) FROM PERSONS
```

Translated by: H.Ozawa

■Define components in a dicon file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE components PUBLIC "-//SEASAR//DTD
   S2Container//EN"
"http://www.seasar.org/dtd/components.dtd">
<components>
   <include path="dao.dicon"/>
   <component class="PersonDao">
      <aspect>dao.interceptor</aspect>
   </component>
</components>
```

■Call DAO from some other class

```
public class PersonLogicImpl implements PersonLogic {

    private PersonDao personDao;

    public void setPersonDao(PersonDao personDao) {
        this.personDao = personDao;
    }

    public void execute() {
        List persons = personDao.getAllPersons();
        String personName = personDao.getPersonName(123);
        int cnt = personDao.getCount();
    }
}
```

When this class is initiated, DAO will be injected by S2

What is AOP?

- Aspect Oriented Programming
- Feature to add functionalities transparently after application is coded
  - Weaving AOP does not change the core process
  - New functionalities are added using AOP
- Horizontally between components, Vertically between processes
- Crosscutting Concern：
  - Should be attractive to system managers
  - Logging, transaction, authentication, exception
- Function common to application logics in components
- Do not overuse!

Example of Aspect XML Configuration File

```
package tutorial.org.seasar.console;
public class HandlingCar implements Car {
  public void run() {
    System.out.println("Turn right!");
  }
}
```

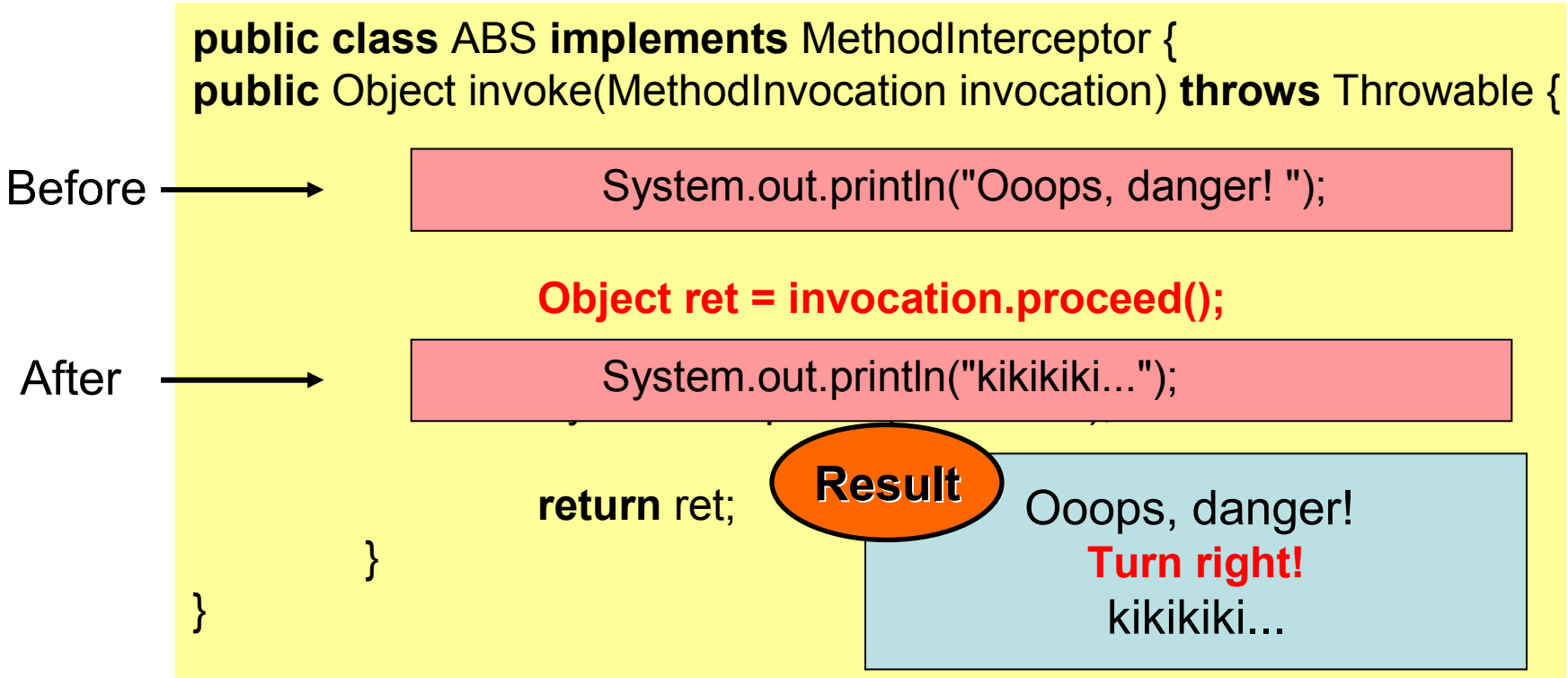
```
<?xml version="1.0" encoding="UTF-8"?>
<components>

  <component name="paintedCar" class="HandlingCar">
    <aspect pointcut="run">
      <component class="ABS"/>
    </aspect>

 </component>
</components>
```

pointcut attribute specifies method to Aspect

Translated by: H.Ozawa

MethodInterceptor

- MethodInterceptor divides into 2 parts during execution
  - Before and after MethodInvocation#proceed() invocation
  - Before invocation is “Before”
  - After invocation is “After”



Transaction Processing (without AOP)

- Application Logic
  - Start a transaction if not already started
  - **Application program codes**
  - Commit transaction if there is no error
  - Rollback transaction if there is an error

Transaction Processing (with AOP)

- Application Logic
  - **What the application is suppose to do**
- Crosscutting concern
  - Start a transaction if it is not already started
  - **Call on the application logic**
  - Commit transaction if there is no error
  - Rollback transaction if there is an error
- Weave crosscutting concern into application logic by associating them in a configuration file

```
<component class="application logic class">
    <aspect>
        <component class="crosscutting concern class"/>
    <aspect>
</component>
```

S2Tx

- J2EE Transaction
  - Implements JTA
  - Seasar2 Extension package
- Transparent Service
  - Only configuration is necessary. There is no need to write code associate Tx

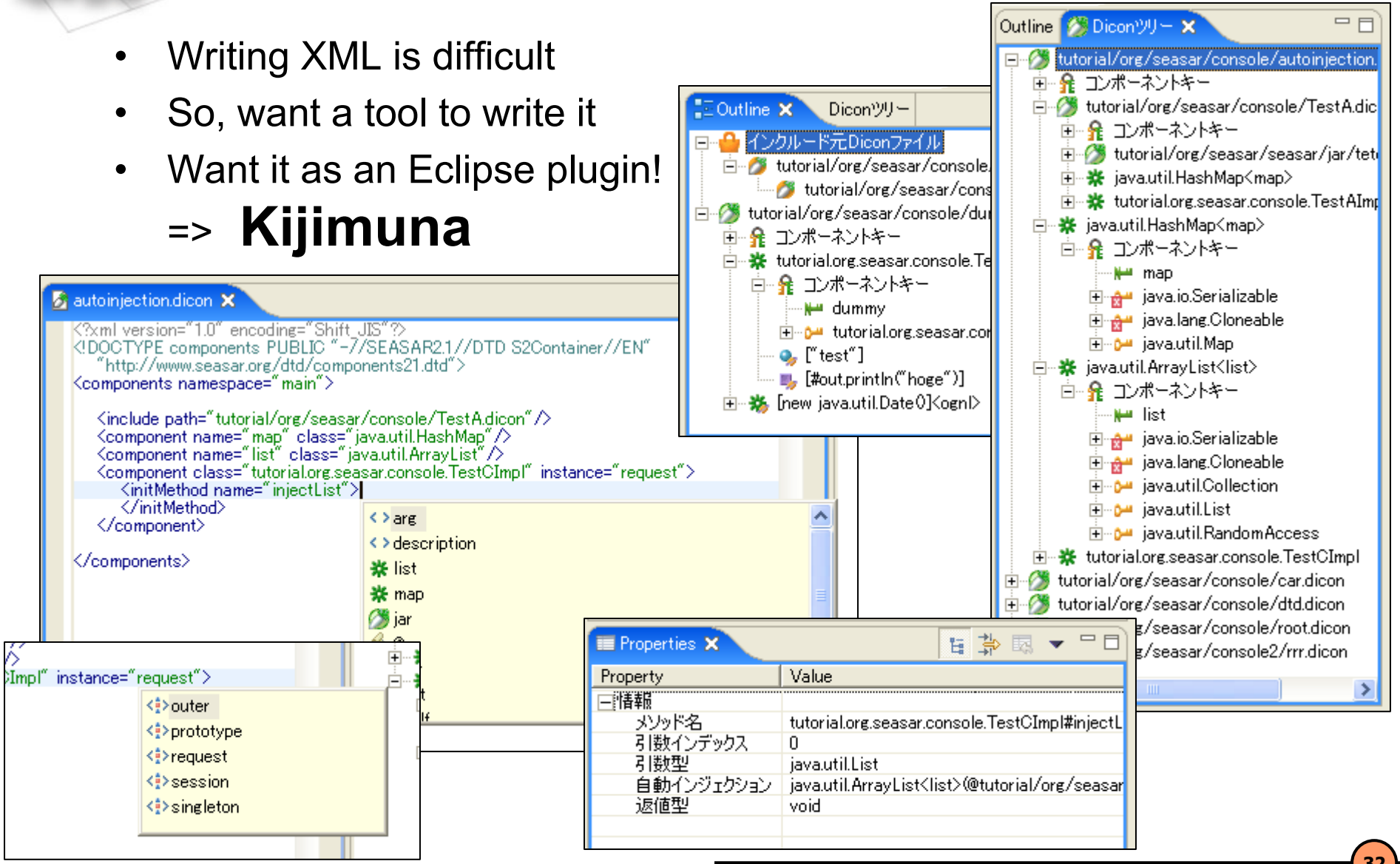
```
<?xml version="1.0" encoding="UTF-8"?>
<components>
  <include path="j2ee.dicon">

  <component class="PersonLogicImpl">
    <aspect>j2ee.requiredTx</aspect>
  </component>
</components>
```

Only have to specify Tx Interceptor in aspect tag

Create Configuration Files Easily!

- Writing XML is difficult
- So, want a tool to write it
- Want it as an Eclipse plugin!

=> **Kijimuna**



Translated by: H.Ozawa

Current S2 Family

- Core Products
  - S2Container（DI container）
  - S2AOP（conform with AOP alliance）
  - S2Tx（automatic transaction control ）
  - S2DBCP（connection pooling ）
  - S2JDBC（similar to Jakarta DbUtils）
  - S2Unit（similar to test first tool）
- Peripheral Products
  - S2JSF
  - S2Dao
  - S2Axis
  - S2Remoting
  - S2GroovyBuilder（write configuration file in Groovy）
  - S2OpenAMF（Flash Remoting）
  - S2Hibernate
  - S2Struts
  - S2Tapestry
  - Maya
  - etc…(sandbox projects)
- Eclipse Plugins
  - Kijimuna
  - S2JSF plugin

Translated by: H.Ozawa

Future of Seasar

- What would be the relationship with the next generation J2EE – especially EJB3?

Seasar Project will support EJB3.0

Please Look Forward!

Translated by: H.Ozawa

Thank you

Translated by: H.Ozawa