

EJB3.0とメタ・プログラミング

稚内北星学園大学
丸山不二夫

講義の目的

- EJB3.0のプログラミングスタイルを概観する。
 - EJBとそのAlternativeの手法を、メタプログラミングの手法の発展として分析する。
-

メタプログラミングとは何か?

- プログラムを対象とするプログラムを、メタプログラムと呼ぶ。
 - 典型的には、プログラムを生成するプログラム(コードジェネレータ)は、メタプログラムである。
 - また、対象プログラムの振る舞いを監視し、それを外部からコントロールするプログラムは、メタプログラムである。
-

メタプログラミングの背景

- ますます増大するシステムの複雑性。
 - EoD(開発の容易さ)への要求の高まり。
- ↓
- 人間にとってのわかりやすさの追求。
 - GUI
 - Configure by Exception
 - 宣言的なアプローチ。
 - コード生成の自動化。
 - 複雑な状態監視の自動化。
-

メタプログラミングのもう一つの意味

- 人間にとってのわかりやすさ。



- 機械にとってのわかりやすさ。
 - 機械自身がプログラムを理解する。
 - Reflectionの手法の一般化。
 - 様々のバリエーション。
 - 機械にとっての分かり易さは、人間にとっての分かり易さと同じものか?
-

EJB3.0プログラミング

J2EEでの開発の難しさ

- ❑ マルチティアでの複雑なデザインパターン
 - Webアプリケーション化
 - ❑ Entity Beansの利用の難しさ
 - Session Beansだけを使う
 - ❑ CMPコーディングの複雑さ
 - BMPだけを使う
-

J2EE5.0の登場

- ❑ J2EEに対する批判を受け止め、プログラミングスタイルを大幅に改善する。
 - ◆ Ease of Development
 - ◆ Ease of Development
 - ◆ Ease of Development
 - ◆ Ease of Development
 - ◆ Ease of Development
 - ◆
-

J2EE5.0の登場

- ❑ J2EE5.0 = JSF1.0 + EJB3.0
(+ JSTL1.1 + JSP2.1
+ JAXB2.0 + JAX-RPC2.0)
 - ❑ J2EE5.0のEoD =
GUIを利用したJ2EEアプリの開発
+ Annotationを利用したプログラミング
-

EJB3.0の特徴

- ❑ Annotation
 - ❑ Configuration by exception
 - ❑ Dependency injection mechanisms
 - ❑ POJO / POJI
 - ❑ Elimination of Home interface
 - ❑ Simplified CMP
 - ❑
-

EJB3.0プログラミング

- ❑ Homeインターフェースがいらない
 - ❑ Local(Remote)インターフェースがいらない
 - ❑ 特別なcreateやremoveメソッドがいらない
 - ❑ 特別なfindやselectメソッドがいらない
 - ❑ Deployment descriptorがいらない
-

Entity Beans

科目担当テーブル courses

| キー id | 科目 course | 教員 teacher |
|----------|--------------|---------------|
| 1 | Java | 丸山 |
| 2 | XML | 植田 |
| 3 | SQL | 安藤 |

テーブルcoursesに EntityBeanを対応させる

- EntityBeanに対応させる ==> @Entity
- テーブルの名前は courses ==> @Table
- キーは自動生成 ==> @Id
- キー項目の名前は id ==> @Column
- 基本的な項目 course ==> @Basic
- 基本的な項目 teacher ==> @Basic

EJB3.0 CourseBean.java

```
package example;  
@javax.ejb.Entity EntityBeanに対応させる  
@javax.ejb.Table (name="courses") テーブルの名前は  
courses  
public class CourseBean {  
    private int _id;  
    private String _course;  
    private String _teacher;  
    @javax.ejb.Id  
    ( generator=GeneratorType.AUTO キーは自動生成  
    @javax.ejb.Column (name="id") キー項目の名前はid  
    public int getId() { return _id; }  
    public void setId(int id) { _id = id; }  
}
```

EJB3.0 CourseBean.java

```
@javax.ejb.Basic  
public String getCourse() {  
    return _course;  
}  
public void setCourse(String course)  
{  
    _course = course;  
}  
@javax.ejb.Basic  
public String getTeacher() {  
    return _teacher;  
}  
public void setTeacher(String teacher)  
{  
    _teacher = teacher;  
}  
}
```

基本的な項目
course

基本的な項目
teacher

EJB3.0 クライアント側での呼び出し 行の発見 find

```
private EntityManager _manager;  
CourseBean [] course = new CourseBean[2];  
course[0] = (CourseBean)  
_manager.find("CourseBean", new Integer(1));  
course[1] = (CourseBean)  
_manager.find("CourseBean", new Integer(2));  
for (int i = 0; i < course.length; i++) {  
    out.println("科目 : " + course[i].getCourse());  
    out.println("教員 : " + course[i].getTeacher());  
}
```

先のプログラムの出力

```
科目 : Java  
教員 : 丸山  
科目 : XML  
教員 : 植田
```

course[0].getCourse()
course[0].getTeacher()

course[1].getCourse()
course[1].getTeacher()

EJB3.0で、テーブル間の関係は
どのように表現されるか？

多対1 Many-to-One の関係

ゼミのテーブル Seminar

| キー seminar_id | ゼミ名 name |
|------------------|-------------|
| 1 | 丸山ゼミ |
| 2 | 植田ゼミ |
| 3 | 安藤ゼミ |
| 4 | 門間ゼミ |
| 5 | 金山ゼミ |

EJB3.0 Seminar.java

```
@Entity
public class Seminar {
    @Id
    @Column ( name= "seminar_id")
    public long getId();
    @Basic
    public String getName();
}
```

学生テーブル Student

| キー student_id | 学生 name | ゼミid seminar_id |
|------------------|------------|--------------------|
| 1 | 藤木文彦 | 1 |
| 2 | 岩本和久 | 1 |
| 3 | 坂本寛 | 2 |
| 4 | 五十嵐理佳 | 4 |
| 5 | 塚本智宏 | 3 |
| 6 | 加藤潔 | 1 |
| 7 | 佐々木政憲 | 2 |
| 8 | 斉藤吉広 | 5 |

丸山ゼミ

多対1

植田ゼミ

EJB3.0 Student.java

```
@Entity
public class Student {
    @Id
    @Column ( name = "student_id" )
    public long getId();
    @Basic
    public String getName();
    @ManyToOne
    @JoinColumn ( name = "seminar_id" )
    public Seminar getSeminar();
}
```

EJB3.0

クライアント側での呼び出し

```
Query allStudent = _entityManager.createQuery(
    "SELECT o FROM Student o");
List students = allStudent.listResults();
for (int i = 0; i < students.size(); i++) {
    Student student = (Student) students.get(i);
    out.println(student.getName() + " は、" +
        student.getSeminar().getName() + "所属です。");
}
```

先のプログラムの出力

```
藤木文彦は、丸山ゼミ所属です。
岩本和久は、丸山ゼミ所属です。
坂本寛は、植田ゼミ所属です。
五十嵐理佳は、門間ゼミ所属です。
塚本智宏は、安藤ゼミ所属です。
加藤潔は、丸山ゼミ所属です。
佐々木政憲は、植田ゼミ所属です。
斉藤吉広は、金山ゼミ所属です。
```

EJB3.0

クライアント側での呼び出し

```
Query semiStudent = _entityManager.createQuery(
    "SELECT s FROM Student s WHERE s.seminar.name =?1");
semiStudents.setParameter(1, "丸山ゼミ");
List students = semiStudent.listResults();
out.println("丸山ゼミ 学生リスト");
for (int i = 0; i < students.size(); i++) {
    Student student = (Student) students.get(i);
    out.println(student.getName());
}
```

プログラムの出力

```
丸山ゼミ 学生リスト
藤木文彦
岩本和久
加藤潔
```

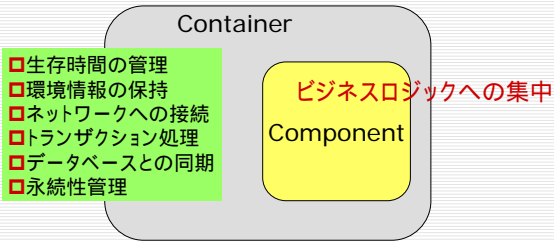
コンテナ・コンポーネント・モデル

メタプログラムとしてのコンテナ

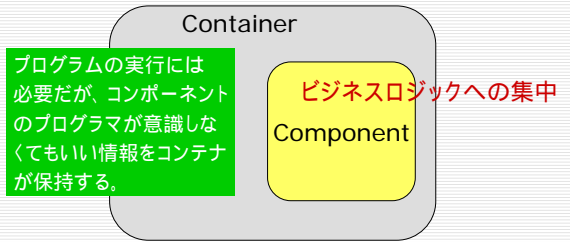
コンテナ・コンポーネント・モデル

- コンポーネントとコンテナの役割分担
- ビジネスロジックへの集中
- コンポーネントの「再利用」
- コンポーネントの連携とデザイン・パターン
- Assemble/Deploy 開発者の役割分担
- 「三つ組」によるプログラミング

コンポーネントとコンテナの機能の分担

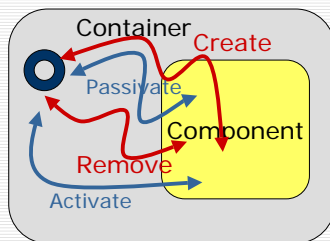


コンポーネントとコンテナの情報の分担



DIの手法、Annotationの手法の基礎

対象プログラムとしてのコンポーネント メタプログラムとしてのコンテナ



対象プログラムの生存時間管理は、
コンテナの基本的な機能の一つ

EJB2.1の三つの定義ファイル

Remoteインターフェース: Bank.java

```
public interface Bank extends EJBObject {  
    public void transferToSaving(double amount)  
        throws RemoteException, InsufficientBalanceException;  
    .....  
}
```

コンポーネントの
インターフェース

Homeインターフェース: BankHome.java

```
public interface BankHome extends EJBHome {  
    public Bank create(String id)  
        throws RemoteException, CreateException;  
}
```

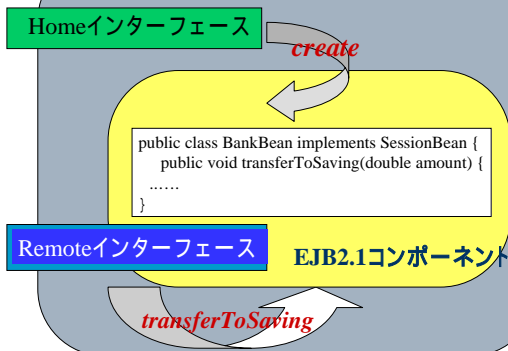
コンテナの
インターフェース

EJBクラス: BankBean.java

```
public class BankBean implements SessionBean {  
    public void transferToSaving(double amount) {  
        .....  
    }  
}
```

コンポーネントの
実装

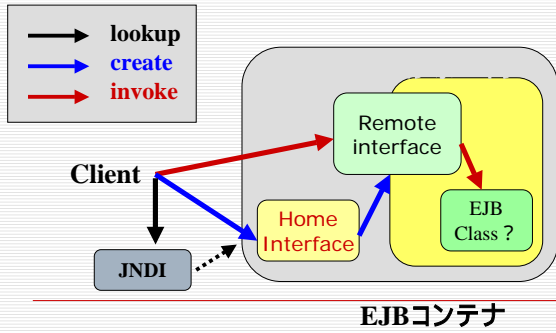
EJB2.1コンテナ



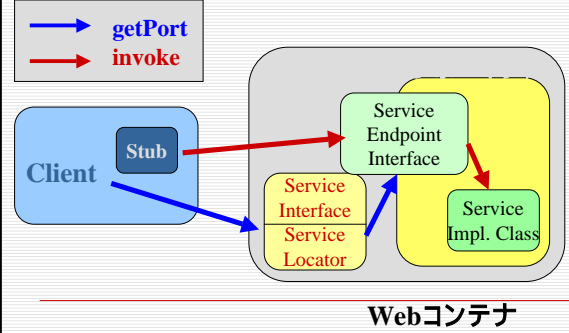
コンテナ・コンポーネント・モデルの一般性

- J2EE/EJB
- JAX-RPC
- J2EE1.4/WS
- Grid/OGSI
- Grid/WSRF

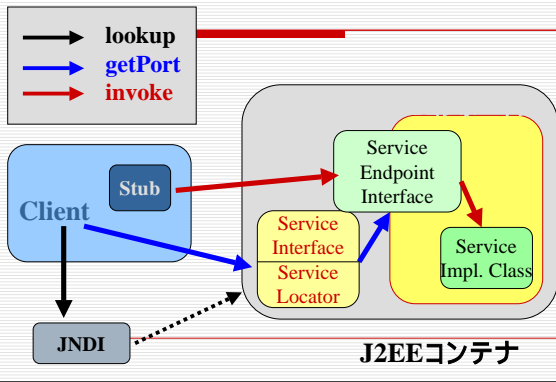
Clientから見たJ2EEのコンテナとEJB2.1の三つの定義ファイル



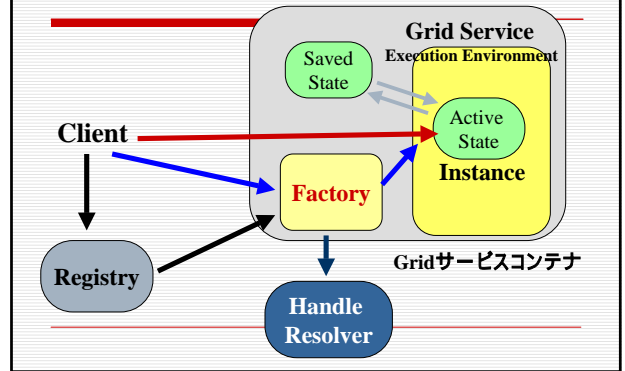
JAX-RPCがWSDLから生成するクラスとコンテナ



J2EE1.4のWebサービス対応



Gridサービスコンテナ



J2EE, Web Service, Grid/OGSI のコンテナ / コンポーネントの比較

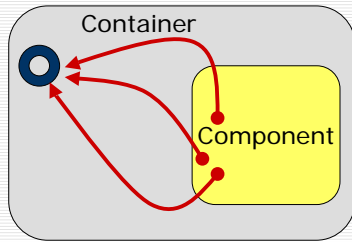
| | J2EE | Web Service | Grid Service |
|--------------------|--------------|-----------------|--------------------|
| Containerの発見 | JNDI | UDDI / WSIL | Registry |
| Containerのインターフェース | Home Intf. | Service Intf. | Factory |
| Componentの生成 | create | get<Port> | createService |
| Componentのインターフェース | Remote Intf. | Remote Endpoint | Handle / Reference |

コンテナ・コンポーネントとコンテキスト

Dependency Injection

EJB2.1

コンポーネントがコンテナから情報を取得する



EJB2.1での環境情報の取得

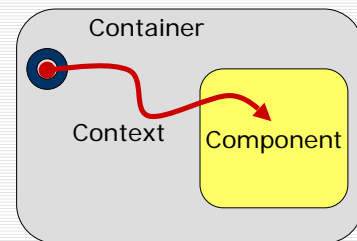
```
public abstract class CustomerEJB implements
    javax.ejb.EntityBean {
    .....
    public abstract void setAccount(AccountLocal account);
    .....
    public void ejbPostCreate(String userId) throws
        CreateException {
    try {
        InitialContext ic = new InitialContext();
        AccountLocalHome alh = (AccountLocalHome)
            ic.lookup("java:comp/env/ejb/Account");
        AccountLocal account =
            alh.create(AccountLocalHome.Active);
        setAccount(account);
        .....
    }
```

制御の逆転 (Inverse of Control) と 従属性注入 (Dependency Injection)

- コンテキストを通じたコンポーネントによるコンテナの環境情報の取得は、コンポーネントの主体的な活動のように見える。
- ただ、獲得された情報の本来の担い手は、コンテナの側。本来的に情報を持つべきものと、それから一時的に情報を取得するものとの区別をする。
- こうした区別を明確にするため、主客を逆転して、コンポーネントがコンテナの情報を獲得するのではなく、コンテナがコンポーネントに情報を注入すべき。

EJB3.0

コンテナがコンポーネントに情報を注入する



制御の逆転 (Inverse of Control) と 従属性注入 (Dependency Injection)

DIとIoC

- Dependency Injection = コンテナの情報をコンポーネントに注入する
- Dependency Injection = メタ・プログラムの情報をプログラムに与える
- IoC = プログラムがメタ・プログラムをコントロールするのではなく、メタ・プログラムがプログラムをコントロールする。

EJB3.0

DI Annotation on Setter Methods

@Inject

```
public void setGreeting(String greeting){
    _greeting = greeting;
}
```

```
<ejb-server jndi-name="java:comp/env/ejb">
  <bean type="qa.TestBean">
    <init greeting="Hello, World from web.xml"/>
  </bean>
</ejb-server>
```

Resin configuration-file

EJB3.0

DI Annotaion on Fields

```
.....  
@Inject(jndiName =  
    "org.jboss.tutorial.injection.bean.Calculator")  
private Calculator calculator;  
.....
```

EJB3.0

DI Annotaion on Setter Methods

```
@EJB (name="calculator", jndiName =  
    "org.jboss.tutorial.injection.bean.Calculator")  
public void setCalculator(Calculator c){  
    set = c;  
}
```

「三つ組み」プログラミングの役割

別プログラムを生成する
Syntax Sugar

EJB2.1の三つの定義ファイル

□ Remoteインターフェース

```
public interface Hello extends  
    javax.ejb.EJBObject {  
    public String sayHello() throws  
  
        java.rmi.RemoteException;  
}
```

EJB2.1の三つの定義ファイル

□ Homeインターフェース (これもRemoteである)

```
public interface HelloHome  
    extends javax.ejb.EJBHome {  
    public Hello create() throws  
        java.rmi.RemoteException,  
        javax.ejb.CreateException;  
}
```

Homeインターフェースのcreateメソッドは、
Remote インターフェースを返す。

EJB2.1の三つの定義ファイル

□ EJBクラス

```
public class HelloEJB implements  
    javax.ejb.SessionBean {  
    public String sayHello() {  
        return "Hello World!";  
    }  
    public void ejbCreate() {}  
    public void setSessionContext(SessionContext  
        sc) {}  
    public void ejbRemove() {}  
    .....  
}
```

EJB2.1の三つの定義ファイルを分析する

- 二つのRemoteインターフェース
- しかし、この二つのRemoteインターフェースを実装したクラスは定義されていない。
- RemoteインターフェースのsayHelloメソッドを実装したように見えるEJBクラスは、形式的には相互に関係がない。
- EJBクラスは、Remoteインターフェースさえ持っていない。

```
public interface HelloHome extends javax.ejb.EJBHome {
    public Hello create() throws
        java.rmi.RemoteException, .....;
}
```

RMI/IIOP

```
public class HelloHomeImpl extends
    PortableRemoteObject implements HelloHome {
    public Hello create() throws RemoteException,..... {
        return (Hello).....;
    }
    .....
}
```

```
public interface Hello extends javax.ejb.EJBObject {
    public String sayHello() throws
        java.rmi.RemoteException;
}
```

RMI/IIOP

```
public class HelloEJB_EJBObjectImpl extends
    PortableRemoteObject implements Hello {
    public String sayHello() throws RemoteException {
        return "Hello World!";
    }
    .....
}
```

```
public String sayHello() throws RemoteException {
    .....
    // 元のEJBクラスのインスタンス
    HelloEJB helloejb = (HelloEJB) .....;
    .....
    // メソッド呼び出しの前処理
    container.preInvoke(...);
    // 元のEJBクラスのメソッド呼び出し
    String s = helloejb.sayHello();
    // メソッド呼び出しの後処理
    container.postInvoke(...);
    .....
    return s;
}
```

メソッドのはさみ込み

EJB2.1の「三つ組み」は、挟み込みで
メソッドを書き換えるための枠組み
に他ならない

- コードの書き換えメカニズムとしてAOPの先駆としての「三つ組み」
- コード・ジェネレータのSyntax Sugarとしての「三つ組み」

EJB2.1のCMP

- Abstractクラスで「実装」?
- プログラマが書くコードと、生成されるコードとの関係が見えにくい。
- HomeインターフェースのFinder
- EJBコード以外に、Deployment Descriptor内にSQLを書くというスタイル。

Syntax Sugar の複雑化

JBOSS AOP

```
public Object localInvoke(Method method, Object[] args)
    throws Throwable {
    ClassLoader oldLoader =
        Thread.currentThread().getContextClassLoader();
    ThreadLocalENCFactory.push(enc);
    try {
        Thread.currentThread().setContextClassLoader(classloader);
        long hash = MethodHashing.calculateHash(method);
        Method beanMethod = (Method) methodMap.get(new long(hash));
        if (beanMethod == null) {
            throw new RuntimeException(".....");
        }
        EJBContainerInvocation newSi = new
            EJBContainerInvocation(beanMethod, args, serverInterceptors);
        newSi.setContainer(this);
        return newSi.invokeNext();
    } finally {
        Thread.currentThread().setContextClassLoader(oldLoader);
        ThreadLocalENCFactory.pop();
    }
}
```

```
<aop>
<interceptor
class="org.jboss.aspects.remoting.InvokeRemoteInterceptor"
scope="PER_VM"/>
<interceptor
class="org.jboss.aspects.security.SecurityClientInterceptor"
scope="PER_VM"/>
<interceptor
class="org.jboss.aspects.tx.ClientTxPropagationInterceptor"
scope="PER_VM"/>
.....
</aop>
```

ejb3-interceptors-aop.xml

JBoss AOPでのIntercept

Annotationの手法

Semantics Sugar

EJB3.0 CourseBean.java

```
package example;
@javax.ejb.Entity EntityBeanに対応させる
@javax.ejb.Table (name="courses") テーブルの名前は courses
public class CourseBean {
    private int _id;
    private String _course;
    private String _teacher;
    @javax.ejb.Id
    ( generator=GeneratorType.AUTO キーは自動生成
    @javax.ejb.Column (name="id") キー項目の名前はid
    public int getId() { return _id; }
    public void setId(int id) { _id = id; }
```

EJB3.0

@Tableの定義

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Table
{
    String name() default "";
    String catalog() default "";
    String schema() default "";
    UniqueConstraint[] uniqueConstraints() default {};
    boolean specified() default true; // For internal use only
}
```

メタプログラミングとしてのAnnotationプログラミング

EJB3.0 @Table(name=...)の実装例

```
private String getTable(Class clazz) {
    Table table = (Table)
        clazz.getAnnotation(Table.class);
    if (table != null && table.name() != null
        && !"".equals(table.name().trim()))
    {
        return table.name();
    }
    return cfg.createMappings().getNamingStrategy().
        classToTableName(clazz.getName());
}
```

メタプログラミングとしてのAnnotationプログラミング

EJB3.0 @Table(schema=...)の実装例

```
private String getSchema(Class clazz) {
    Table table = (Table)
        clazz.getAnnotation(Table.class);
    if (table != null &&
        table.schema() != null &&
        !"".equals(table.schema().trim()))
    {
        return table.schema();
    }
    return null;
}
```

一般プログラマと Annotationプログラマ

- 一般のプログラマは、Annotationを利用してプログラムを作るが、Annotationプログラムを書く必要はない。
- Annotationプログラマは、具体的な処理を行うプログラムを対象として、その振る舞いを規定するメタ・プログラムとしてAnnotationプログラムを書く。

EJB3.0以外での Annotationの手法

Semantics Sugarとしての Annotation

JSR181 -- Web Services Metadata for Java Platform

```
@WebService(
    name="ExampleWebService",
    targetNamespace="http://openuri.org/11/2003/
    ExampleWebService")
@SOAPBinding(
    style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.LITERAL)
public class ExampleWebServiceImpl {
    @WebMethod( action="urn:login" )
    @WebResult( name="Token" )
    public LoginToken login(
        @WebParam( name="UserName" ) String username,
        @WebParam( name="Password" ) String password ) {
        // ...
    }
    // ...
}
```

RPC/literal

BottomUpで、 先のJavaコードから生成されるWSDL

```
.....
<message name="login">
  <part name="UserName" type="s:string"/>
  <part name="Password" type="s:string"/>
</message>
} @WebParam

<message name="loginResponse">
  <part name="Token" type="LoginToken"/>
</message>
} @WebResult

<portType name="ExampleWebService">
  <operation name="login">
    <input message="tns:login"/>
    <output message="tns:loginResponse"/>
  </operation>
} @WebMethod
.....
```

Annotationの手法 (JSR181)

- Javaプログラムの持つ情報 < WSDLの持つ情報
- Javaプログラムの持つ情報 + annotationの情報 = WSDLの持つ情報

JSR220 -- Enterprise JavaBeans 3.0 Specification

```
@javax.ejb.Entity
public class Course {
    @javax.ejb.Id
    @javax.ejb.Column ( name="student_id" )
    public long getId()

    @javax.ejb.Basic
    public String getName()

    @javax.ejb.ManyToMany( targetEntity="Course" )
    @javax.ejb.AssociationTable (
        table=@javax.ejb.Table (name="student_course_map"),
        joinColumns=@javax.ejb.JoinColumn (name="student_id"),
        inverseJoinColumns=@javax.ejb.JoinColumn
        (name="course_id")" )
    public Collection getCourses()
}
```

Annotationは「意味」を持つ

- @javax.ejb.ManyToMany ...
テーブルStudentはテーブルCourseと「多対多」の関係にある。
- @javax.ejb.AssociationTable ...
この二つのテーブル間の「多対多」の関係は、テーブル student_course_mapで表現され、そのテーブルは、この Studentテーブルのstudent_id項目と、Courseテーブルのcourse_id項目から構成され、次のメソッドgetCoursesは、この「多対多」の関係の下で、キーstudent_idに対応する、複数のcourse_id連の集合を返す。

Annotationの手法 (JSR220)

- Javaプログラムの持つ情報 < テーブル間の関係の持つ情報
- Javaプログラムの持つ情報 + annotationの情報 = テーブル間の関係の持つ情報

JSR222 -- The Java™ Architecture for XML Binding (JAXB) 2.0

```
public class Trade {
    @XmlAttribute
    String getSymbol();
    void setSymbol(String);
    @XmlElement
    int getQuantity();
    void setQuantity(int);
}
```

```
<xsd:complexType
  name="trade">
  <xsd:sequence>
    <xsd:attribute
      name="symbol"
      type="xsd:string"/>
    <xsd:element
      name="quantity"
      type="xsd:int"/>
  </sequence>
</xsd:complexType>
```



Annotationの手法 (JSR222)

- Javaプログラムの持つ情報 < XML Documentの持つ情報
- Javaプログラムの持つ情報 + annotationの情報 = XML Documentの持つ情報

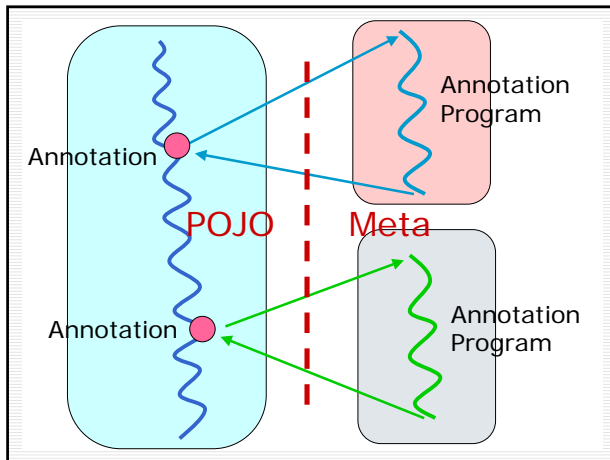
Annotationの手法

- Javaプログラムの持つ情報 < があるEntityの持つ情報
- Javaプログラムの持つ情報 + annotationの情報 = があるEntityの持つ情報

Annotation = Semantic Sugar

AnnotationとAOP

あるいは、
AnnotationとContainer Model



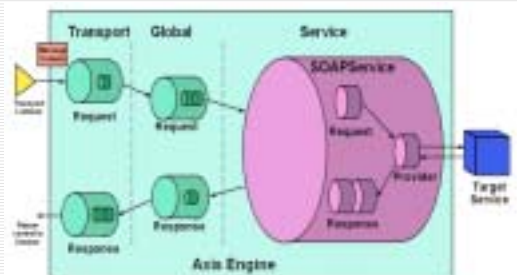
メソッドの書き換えと メッセージの書き換えの同等性

Webサービスとメタ・プログラミング

AXIS/GridでのHandlerの役割

- Webサービスでも、Request, Responseの通り道に、Handlerを設定することで、もとのオペレーションを挟み込む形でのオペレーションの変形が自然に導入されている。
- Logging, Security(Authentication), Metering, Tracing 等、WS-Handlerが良く使われる分野が、AOPの書き換え技法でも良く使われるのは偶然ではない。それは、同じ構造を持つ。

Handler でのメッセージの書き換え



AXIS

Deploy.wsdd

```
<deployment name="test".../>
.....
<service name="urn:xmletoday-delayed-quotes" provider="java:RPC">
  <parameter name="className" value="samples.stock.StockQuoteService"/>
  <parameter name="allowedMethods" value="getQuote test"/>
  <parameter name="allowedRoles" value="user1,user2"/>

  <requestFlow name="checks">
    <handler type="java:org.apache.axis.handlers.SimpleAuthenticationHandler"/>
    <handler type="java:org.apache.axis.handlers.SimpleAuthorizationHandler"/>
  </requestFlow>
</service>
.....
</deployment>
```

AXIS

Handler プログラミング

```
public class TestMimeHeaderHandler extends BasicHandler {

    public void invoke(MessageContext msgContext) throws
    AxisFault {
        Message requestMessage =
            msgContext.getRequestMessage();
        Message responseMessage = new
            Message(requestMessage.getSOAPEnvelope());
        String[] fooHeader =
            requestMessage.getMimeHeaders().getHeader("foo");
        if (fooHeader != null) {
            responseMessage.getMimeHeaders().addHeader("foo",
                fooHeader[0]);
        }
        msgContext.setResponseMessage(responseMessage);
    }
}
```

AXIS

Client-deploy.wsdd

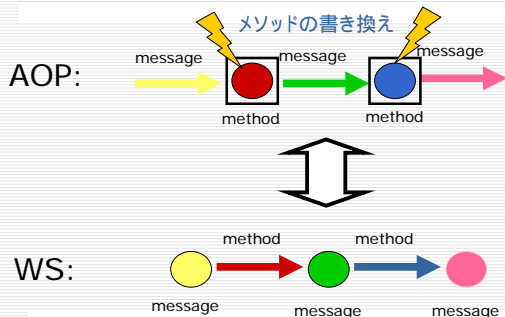
```
<service name="urn:xmletoday-delayed-quotes">
  <requestFlow>
    <handler type="username-setter"/>
    <handler type="wssecurity-sender"/>
  </requestFlow>
  <responseFlow>
    <handler type="wssecurity-receiver"/>
  </responseFlow>
</service>
```

AXIS

Server-deploy.wsdd

```
<service name="DSigEncStockQuoteService"
  provider="java:RPC">
  <requestFlow>
    <handler type="wssecurity-receiver"/>
    <handler type="simple-authenticator"/>
  </requestFlow>
  <responseFlow>
    <handler type="wssecurity-sender"/>
  </responseFlow>
  <parameter name="className"
    value="com.ibm.services.applications.StockQuoteService"/>
  <parameter name="allowedMethods" value="getQuote"/>
</service>
```

メソッドとメッセージの双対性

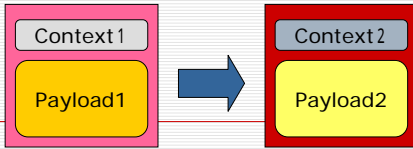


AOP的「書き換え」 vs. WS的「書き換え」

- AOP的「書き換え」が対象としているのは、空間的・論理的な構造として把握可能な、密に結合した構造
- Webサービスの「書き換え」が対象としているのは、時間の流れの中で継起する、相互には疎な結合しかしていないメッセージ

メッセージの書き換え

- 一般プログラマには、コンテキスト部分は Transparent で見えないが、環境としては存在している。
- メタ・プログラマには、コンテキストへのアクセスが許されなければならない。



WSRF

server-config.wsdd (1)

```
<requestFlow>
<handler type="java:org.globus.wsrfl.handlers.AddressingHandler"/>
<handler type="java:org.globus.wsrfl.handlers.URLMapper"/>
<handler type="java:org.globus.axis.handlers.ServiceDescHandler"/>
<handler type="AuthenticationServiceHandler"/>
<handler type="java:org.globus.wsrfl.handlers.MessageLoggingHandler"/>
<handler type="java:org.globus.wsrfl.handlers.JNDIHandler"/>
<parameter name="className"
value="org.globus.wsrfl.impl.security.authentication.wssec.WSSecurityHandler"/>
</handler>
<handler
type="java:org.globus.wsrfl.impl.security.authentication.SecurityPolicyHandler"/>
<handler
type="java:org.globus.wsrfl.impl.security.authorization.AuthorizationHandler"/>
<handler type="java:org.globus.wsrfl.handlers.FaultHandler"/>
</requestFlow>
```

WSRF

server-config.wsdd (2)

```
<responseFlow>
<handler type="java:org.globus.wsrfl.handlers.WSDLHandler"/>
<handler type="java:org.globus.wsrfl.handlers.AddressingHandler"/>
<handler type="java:org.globus.wsrfl.handlers.JAXRPCHandler">
<parameter name="className"
value="org.globus.wsrfl.impl.security.authentication.securemsg.X509EncryptHandler"/>
</handler>
<handler type="java:org.globus.wsrfl.handlers.JAXRPCHandler">
<parameter name="className"
value="org.globus.wsrfl.impl.security.authentication.securemsg.X509SignHandler"/>
</handler>
<handler type="java:org.globus.wsrfl.handlers.JAXRPCHandler">
<parameter name="className"
value="org.globus.wsrfl.impl.security.authentication.secureconv.GSSHandler"/>
</handler>
<handler
type="java:org.globus.wsrfl.handlers.MessageLoggingHandler"/>
</responseFlow>
```

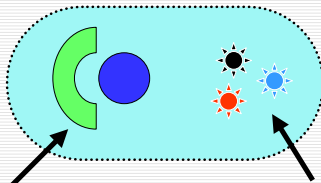
WSRFのコンテナ

点状のEndpointから、
広がりを持つEndpointReferenceへ

WS-Resourceとは何か?

「状態を持たないWebサービス」と
「状態を持つリソース」を
分離したうえで、組み合わせたもの

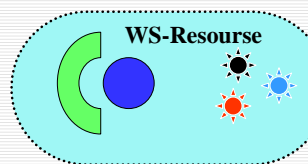
WS-Resource



状態を持たないWebサービス

状態を持つリソース

WS-Resourceの「状態」としての WS-Resource Properties Document

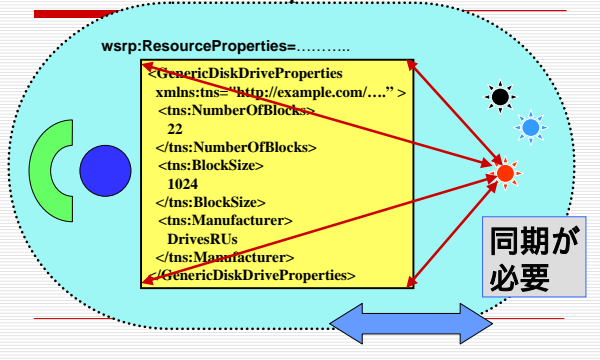


WSDLのportTypeでResource
Properties Documentを定義

```
<GenericDiskDriveProperties
xmlns:tns="http://example.com/..." >
<tns:NumberOfBlocks>
22
</tns:NumberOfBlocks>
<tns:BlockSize>
1024
</tns:BlockSize>
<tns:Manufacturer>
DrivesRUs
</tns:Manufacturer>
</GenericDiskDriveProperties>
```

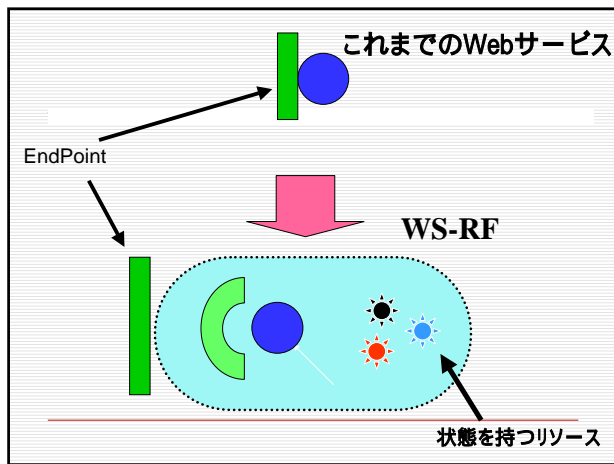
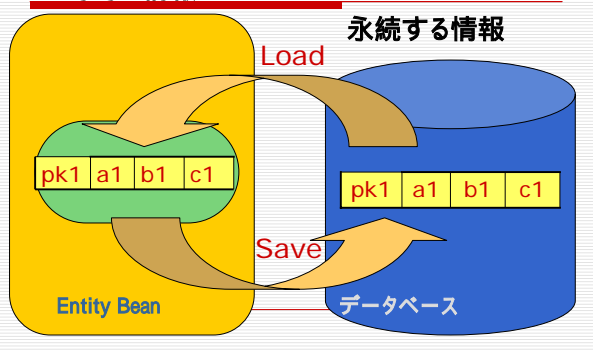
```
<!-- Association of resource properties document to a portType -->
<wsdl:portType name="GenericDiskDrive"
wsrf:ResourceProperties="tns:GenericDiskDriveProperties" >
```

Statefull-Resourceの「Projection」としての WS-Resource Properties Document



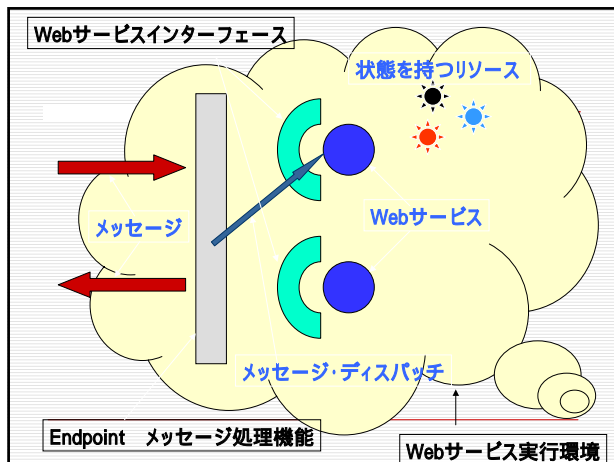
J2EE Entity Beanでの同期

一時的な情報



Webサービスの変化

- point-wiseなEndpointから、内部に広がりをもつEndpointReferenceへ
- Webサービスのコンテナの再構築としてのGrid/WSRFでのHomeインターフェースの利用



メタプログラミングの広がり

- BPEL Webサービスを対象としたメタ・プログラミング
- JMX プログラムコンポーネントの監視と制御
- DSL MS Domain Specific Language

メタプログラミングの背景

- ますます増大するシステムの複雑性。
- EoD (開発の容易さ) への要求の高まり。



- 人間にとってのわかりやすさの追求。
 - GUI
 - Configure by Exception
 - 宣言的なアプローチ。
- コード生成の自動化。
- 複雑な状態監視の自動化。

メタプログラミングのもう一つの意味

- 人間にとってのわかりやすさ。



- 機械にとってのわかりやすさ。
 - 機械自身がプログラムを理解する。
 - Reflectionの手法の一般化。
 - 様々のバリエーション。
- 機械にとっての分かり易さは、人間にとっての分かり易さと同じものか？