

1 アプリケーションのマイグレーション

WebSphere V6.0 に移行する際にアプリケーションのマイグレーションを行う場合の基本的な流れは、下記のフローチャートようになります。

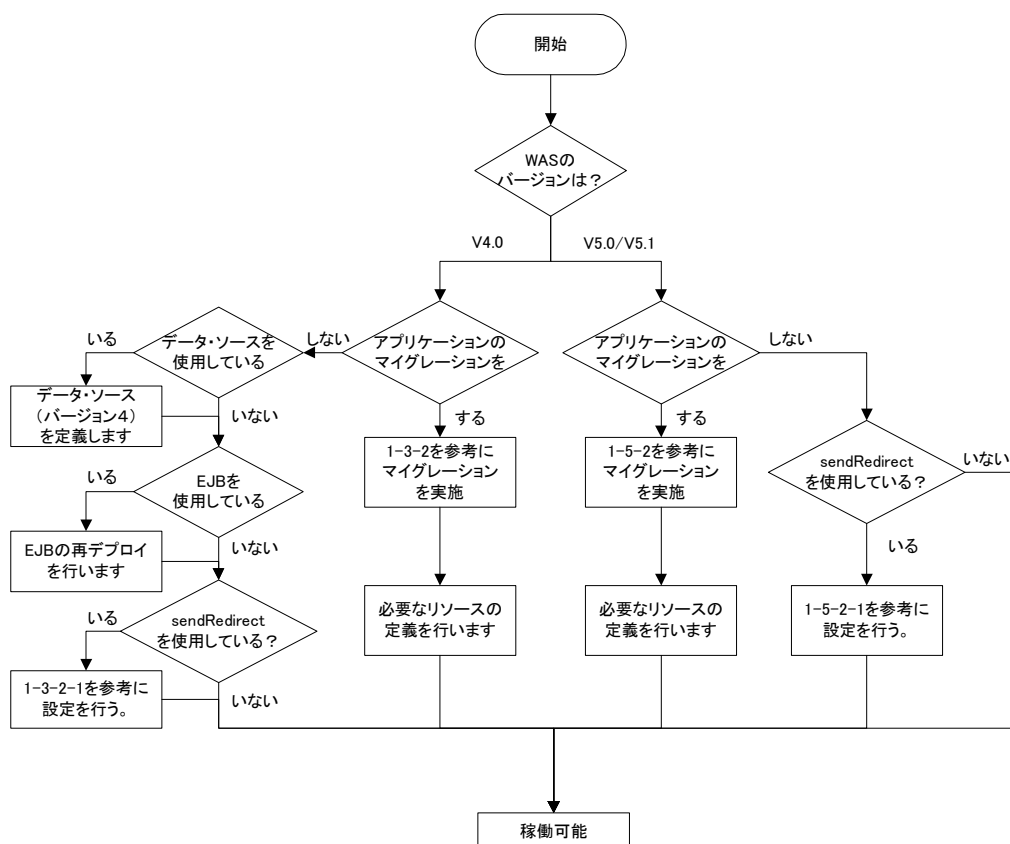


図 1-1 マイグレーションの作業フロー図

1-1 WAS の実行環境について

J2EE 1.4 対応だが、J2EE 1.3/J2EE 1.2(制限付きで) アプリケーションを実行可能です。
J2EE1.2 のアプリケーションでは、WAS V4 互換データ・ソースしか使用できないなど、リソース仕様に関して制限があります。また、IBM 独自機能に関しては、書き換えが必要なケースもあります。

1-2 アプリケーションのマイグレーション(V4 から V6 の場合)

WebSphere V4.0.x では、下記の J2EE バージョン 1.2 のコンポーネントを使用しています。

Java Servlet	2.2
JavaServer Pages (JSP)	1.1
Enterprise JavaBeans (EJB)	1.1
Java DataBase Connectivity (JDBC)	2.0
Java Message Service (JMS)	1.0.2

これが、WebSphere V6.0 になることにより、下記の J2EE バージョン 1.4 に変更する必要があります。この際のアプリケーションへの影響を説明していきます。

Java Servlet	2.4
JavaServer Pages (JSP)	2.0
Java DataBase Connectivity (JDBC)	3.0
Enterprise JavaBeans (EJB)	2.1
Java Message Service (JMS)	1.1

1-3 アプリケーションへの影響

マイグレーションを行う際に、影響が大きい物は CMP EntityBean があります。また、サーブレットや JSP では、文字コードのハンドリングに関わる変更がありますが、大きな変更はサーブレット・フィルタが Java 標準になったことが挙げられます。

とりわけ、EJB 1.1 から 2.0 になった際の CMP は、仕様が大きく変更されていますので、マイグレーションを行うよりも、再作成することをお勧め致します。

また、WAS V5 以降では、API の変更だけでなく、クラス・ローダーなど挙動も異なる物があります。ここでは、コードの変更だけでなく、挙動が変更された物についても紹介していきます。

1-3-1 コードの変更を極力行わない場合

WebSphere V6.0 でも引き続き J2EE 1.2 へのサポートが行われているため、ほとんどのアプリケーションはそのまま稼動することができます。

コードの変更を行わない場合は、下記の点に注意してください。

1. データ・ソースとして V4 互換のデータ・ソースを使用する必要があります。
データ・ソース(バージョン4)に、データ・ソースを定義してください。
2. EJB は、再デプロイが必要となります。

1-3-2 WebSphere V6.0 がサポートするバージョンへコードを変更する場合

WAS に用意されている IBM 拡張を使用していないサーブレット・JSP であれば、ほぼ変更することなく使用することができます。EJB は、SessionBean であればコードレベルでの変更は必要ないと言えますが、CMP EntityBean は新規に作成することになります。

1-3-2-1 Servlet

サーブレットとしての基本機能のマイグレーション項目では、文字コードの判別やコンテンツタイプの指定などが必須になっています。ですが、Servlet+JSP のアプリケーションであれば、ほとんど影響はありません。また、WAS の拡張機能である PageListServlet や MIME フィルターを使用している場合は再作成が必要になります。

●変更が必須となる項目

1. サーブレットを使用してハイパーテキスト・マークアップ言語 (HTML) 出力を生成している。
WAS 4.0 以前ではデフォルトは「text/html」になっていましたが、WAS 5.0 以降では、実行しない場合には Content-type がつかない為に ServletResponse#setContentType()の実行が必須になります。
2. リクエスト文字列の文字コードの判別
WAS 4.0 までは WAS 独自機能による自動判別が有効でしたが、WAS 5.0 以降は、Servlet API 2.2 から新設された規定に従い、デフォルトの文字コードは「8859_1」(en)となりますので、それ以外の文字コードで処理する場合には、アプリケーション中で ServletRequest#setCharacterEncoding()を使用する必要があります。
3. サーブレット構成 XML ファイルの構成情報に依存する PageListServlet を拡張している。
この PageList と互換性を持つコンポーネントは、WAS として提供されていないため、互換機能を持つサーブレットを自作する必要があります。
※WebSphere Developer Domain/Japan で互換サーブレットのサンプルが入手可能です。
http://www-06.ibm.com/jp/software/websphere/developer/wv5/35to50/appendix_1.html
4. MIME フィルターによるサーブレット起動
WAS 3.5 より実装されている IBM 独自機能で、V6 では使用できなくなっています。
代替の機能が提供されないため、Java 標準の Servlet Filter クラスで新規作成する必要があります。
5. response.sendRedirect メソッドを呼び出している。
V4/V5 では、Web アプリケーションのコンテキスト・ルートを使用していましたが、V6 (J2EE 1.4)では、挙動が変更され、Web サーバーのドキュメント・ルートを使用する様になりました。

この場合の対応策としては、下記の 2 つがあります。
 1. コードの変更
response.sendRedirect(Location)の「Location」の位置を、相対パスから絶対パスへ変更します。
 2. com.ibm.websphere.sendredirect.compatibility の設定
WebSphere の設定で、response.sendRedirect の挙動を V4/V5 と同じにすることができます。
 1. コンソールのナビゲーション・ツリーで、「サーバー」>「アプリケーション・サーバー」をクリックします。
 2. 「アプリケーション・サーバー」ページで、構成対象のサーバーの名前をクリックします。
 3. 選択したアプリケーション・サーバーの設定ページの「サーバー・インフラストラクチャー」で、「Java およびプロセス管理」>「プロセス定義」をクリックします。

サーバー・インフラストラクチャー

- ▣ Java およびプロセス管理
 - クラス・ローダー
 - プロセス定義
 - プロセスの実行
 - モニター・ポリシー
- ▣ 管理

図 1-2 response.sendRedirect 設定 1

4. 「プロセス定義」ページで、「Java 仮想マシン」をクリックします。

追加プロパティ

- カスタム・プロパティ
- Java 仮想マシン
- プロセスの実行
- プロセス・ログ
- ロギングおよびトレース

図 1-3 response.sendRedirect 設定 2

5. 「Java Virtual Machine」ページで、「カスタム・プロパティ」をクリックします。

追加プロパティ

- カスタム・プロパティ

図 1-4 response.sendRedirect 設定 3

6. 「カスタム・プロパティ」ページで「新規」をクリックします。

選択	名前	値	説明
なし			
合計 0			

図 1-5 response.sendRedirect 設定 4

7. プロパティの設定ページで、名前に「com.ibm.websphere.sendredirect.compatibility」を、値に「true」を指定して、「OK」をクリックします。

一般プロパティ

* 名前
com.ibm.websphere.sendRedirect

* 値
true

説明

適用 OK リセット 取り消し

図 1-6 response.sendRedirect 設定 5

8. コンソールのタスクバーで、「保管」をクリックします。
9. アプリケーション・サーバーを停止してから、アプリケーション・サーバーを再始動します。

●非推奨となっている項目

- com.ibm.websphere.servlet.filter を使用したサーブレット・フィルタ
Java 標準の Servlet Filter クラスへ移行することが推奨されます。

1-3-2-2 JSP

JSP のマイグレーションでは、明示的なエンコードの指定が必要になることが、大きな変更点となります。また、JSP 内で使用する JavaBean に、パッケージがない場合の扱いが変わっています

●必須となる項目

1. JSP のページタグへの pageEncoding によるエンコードを指定

<% @ page language="java" contentType="text/html; charset=Shift_JIS" pageEncoding="Shift_JIS"%>
この設定がない場合、ブラウザ側での文字判別に失敗による文字化けが発生します。

2. アプリケーションが名前のないパッケージ使用時のページディレクティブの import 追加
JSP 1.2 仕様のセクション 8.2 では以下のように記載されています。

JSP コンテナは JSP ページごとに JSP ページ・インプリメンテーション・クラスを作成します。JSP ページ・インプリメンテーション・クラスの名前は、インプリメンテーションに依存しています。JSP ページ・インプリメンテーション・オブジェクトはインプリメンテーション依存の名前付きパッケージに帰属しています。使用されるパッケージは JSP によって異なる場合があるので、想定事項は最小限にしておく必要があります。明示的なクラスのインポートを行わない場合は、名前のないパッケージを使用しないでください。

例えば、myBeanClass が名前のないパッケージに入っており、これを jsp:useBean タグで参照する場合、以下の例に示すように、ページ指示インポート属性とともに myBeanClass を明示的にインポートしなければなりません。

```
<% @page import="myBeanClass" %>
...
<jsp:useBean id="myBean" class="myBeanClass" scope="session"/>
```

●非推奨となっている項目

- tsx タグ (IBM 拡張タグ) の JSTL への移行
必須の変更はありませんが、[tsx:]で始まる JSP タグが非推奨となりましたので、このタグを使用している場合は、Java 標準の JSTL への置き換えが推奨されます。

下記の表に沿って、タグを変更してください。

tsx tag	JSTL tag
tsx:repeat	c:forEach
tsx:dbconnect	sql:setDataSource
tsx:userid	sql:setDataSourceタグのuser属性
tsx:passwd	sql:setDataSourceタグのpassword属性
tsx:dbquery	sql:query
tsx:dbmodify	sql:update
tsx:getProperty	式言語 (EL)を使用する

表 1-1 tsx タグと JSTL タグの対応表

たとえば、tsx を使用した実装で、下記のようなコードの場合

```
<tsx:repeat start="request.getParameter("index_a") %>" end="<%=request.getParameter("index_b") %>"
index="idx">
<%=idx %>
</tsx:repeat>
```

JSTL で実装した場合下記のようになります。

```
<c:forEach var="index" end="${param.index_b}" begin="${param.index_a}">
<c:out value="${index}" />
</c:forEach>
```

1-3-2-3 JNDI

もし、JNDI の URI 情報などを、ハードコーディングしている場合は、リソース参照を使用した参照方法に変更してください。こうすることで、URI の構成が柔軟になるとともに、データ・ソースや EJB のトランザクション分離レベルなどを設定できるようになります。

ここでは、EJB のホームインターフェイスの呼び出しを例にとりて紹介します。

```
try{
    Properties props = new Properties();
    props.put(InitialContext.PROVIDER_URL,"iiop://localhost:2809");
    InitialContext ctx = new InitialContext(props);
    employeeHome = (EmployeeHome)ctx.lookup("ejb/com/ise/EmployeeHome");
}catch(Exception e){
    e.printStackTrace();
}
```

上記をリソース参照を使用する用に変更すると下記のようになります。

```
try{
    InitialContext ctx = new InitialContext();
    employeeHome = (EmployeeHome)ctx.lookup("java:comp/env/ejb/com/ise/EmployeeHome");
}catch(Exception e){
    e.printStackTrace();
}
```

上記コードでの変更点は、InitialContext に Properties を渡していない点と、JNDI 名がリソース参照を使用した形になっていることです。InitialContext に Properties を渡していないのは、V5 以降 JNDI サーバーは、

分散ネーミング・スペースに対応し各プロセスも持つようになりました、そのため従来のように JNDI サーバーを指定しなくても利用することが可能です。リソース参照を使用するために、`java:comp/env/...`の形式で JNDI 名を指定しています。この形式を利用するときには、配置記述にリソース参照を定義する必要があります。

配置記述やアプリケーションのインストール時に設定する、参照先の JNDI 名を分散ネーミング・スペースに対応した JNDI 名に置き換えます。下記の形式で記述することで、セル全体の EJB を指定することができますようになります。

このときの命名規則は、下記ようになります。

```
/cell/node/[NODE_NAME]/servers/[SERVER_NAME]/[RESOURCE_NAME]
```

```
/cell/clusters/[CLUSTER_NAME]/[RESOURCE_NAME]
```

例1.) セル[host1Cell]上のノード[node01Node]にあるアプリケーション・サーバー[Server1]で、JNDI 名 [com/ise/EmployeeHome]の EJB を指定する場合

```
/cell/nodes/node1Node/servers/serever1/com/ise/EmployeeHome
```

例2.) セル[host1Cell]上のクラスター[SampleCluster]で、JNDI 名 [com/ise/EmployeeHome]の EJB を指定する場合

```
/cell/clusters/SampleCluster/com/ise/EmployeeHome
```

もし、セル外のリソースを利用するなどの理由で、JNDI サーバーの指定を「iiop://」の形式で InitialContextFactory に設定している場合は、「corbaloc:」形式に変更することが推奨されます。

旧) iiop://localhost:2809

新) corbaloc:iiop:localhost:2809

また、この「corbaloc:」形式の場合、複数の JNDI サーバーを記述することで、JNDI へのアクセス障害を回避することができます。この場合の JNDI サーバーへのアクセス順は、記述されている順と同じになります。

例)

```
corbaloc:iiop:host1:2809;host2:2809
```

この場合、host1 で稼働する JNDI サーバーにアクセスできなければ、host2 で稼働する JNDI サーバーにアクセスします。

1-3-2-4 JMS

V6 から、JMS1.1 がサポートされました。

JMS1.0 とのコード的な違いとしては、JMS 接続ファクトリーが共通化された事が変更点となります。

JMS1.0 のコードから、JMS1.1 のコードへ変更を行うには、`javax.jms.QueueConnectionFactory` を `javax.jms.ConnectionFactory` に変更し、`javax.jms.QueueConnection` を `javax.jms.Connection` に変更する事になります。

上記仕様クラスの変更以外のコードは、そのまま使用する事ができます。

ここでは、`ConnectionFactory` などのオブジェクトの作成例を、提示します。

使用するクラスが変更されている以外のコードの変更が無いことが見ていただけると幸いです。

例1) JMS1.0 の場合

```
QueueConnectionFactory qcf = (QueueConnectionFactory)ctx.lookup(jndiNameCF);
```

```
QueueConnection qCon = qcf.createQueueConnection();
```

```
QueueSession qSession= qCon.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
```

```
QueueSender          qSender = qSession.createSender(queue);
```

例2) JMS1.1 の場合

```
ConnectionFactory cf = (ConnectionFactory)ctx.lookup(jndiNameCF);
```

```
Connection          Con = cf.createQueueConnection();
```

```
Session             Session = Con.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
```

```
Sender              Sender = Session.createSender(queue);
```

1-3-2-5 データアクセス

コード的な変更は必要ありませんが、データ・ソース設定「共有属性」のデフォルト値が変更されたことによる挙動の違いがあります。WAS V4 と同じように「共有属性」を使用しない場合は、明示的に「Unshareable」を設定する必要があります。また、従来の `DataAccessBean` が非推奨となりましたので、同様の動作を行う物を自作するか、SDO への移行を行う必要があります。

●確認が必要となる項目

・共有属性

新旧のデータ・ソースでは共有属性のデフォルトが異なります。

「データ・ソース (バージョン 4)」からえられる DB 接続は「Unshareable」でしたが、通常の「データ・ソース」は、指定がなければ DB 接続は「Shareable」となります。この「Shareable」の DB 接続は `close()` を実行してもすぐにはプールに戻らないことに注意してください。サーブレット実行中は保持されて、再度 `getConnection` が実行された場合には同じ接続が渡され、サーブレット実行が終了するとプールに戻る事になります。

これを変更するには、リソース参照で定義してあるデータ・ソース定義の共有属性を、手動で変更する必要があります。

```
<resource-ref>
  <description>
</description>
  <res-ref-name>jdbc/sample</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope> ←この行を編集します。
</resource-ref>
```

共有属性を有効にする場合

```
<res-sharing-scope>Shareable</res-sharing-scope>
```

共有属性を無効にする場合

```
<res-sharing-scope>Unshareable</res-sharing-scope>
```

●非推奨となっている項目

・DataAccessBeans を使用しているアプリケーション

SDO への移行が推奨されていますが、SDO は通常の JDBC(データ・ソース)アクセスとは、アーキテクチャが大きく異なりますので、用途に応じた再作成が必要となります。

補足) SDO について

SDO は、「分離されたデータグラフ」というアーキテクチャをとっています。これは、データ・ソースへの接続とアプリケーションが「Data Mediator Service」により「分離され」、データ・ソースへア

クセした結果が「Data Object」の集合である「Data Graph」に格納されるというアーキテクチャです。

JSR-235 Service Data Object

<http://www.jcp.org/ja/jsr/detail?id=235>

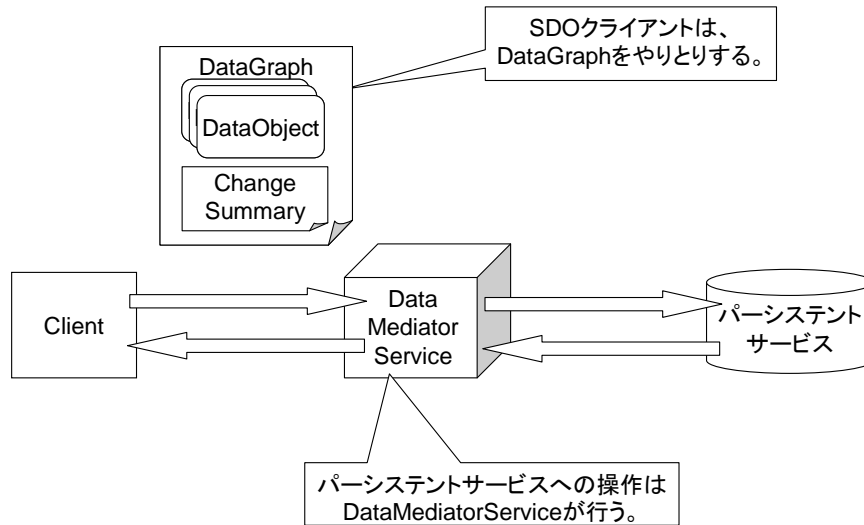


図 1-7 SDO 概要図

参考として、SDO を使用した Select を行うコーディングの例を紹介します。

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Iterator;
import javax.naming.*;
import javax.servlet.*;
import javax.sql.DataSource;
```

```
import com.ibm.websphere.sdo mediator.*;
import commonj.sdo.DataObject;
```

```
public class SDOSample extends HttpServlet implements Servlet {
```

```
    private DataSource ds = null;
```

```
    public SDOSample() {
        super();
    }
```

```
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
```

```
        try{
```

```
            //DataSourceからDB接続を取得
            Connection con = ds.getConnection();
```

```
            //ConnectionWrapperを取得
            ConnectionWrapperFactory cwf = ConnectionWrapperFactory.soleInstance;
            ConnectionWrapper cw = cwf.createConnectionWrapper(con);
```

```

//MetaDataの作成
MetadataFactory mdf = MetadataFactory.eINSTANCE;
Metadata metaData = mdf.createMetadata();

//MetaDataにテーブルの情報を定義する
Table employeeTable = metaData.addTable("EMPLOYEE");

//MetaDataにカラムの情報を定義する
Column empno = employeeTable.addIntegerColumn("EMPNO");
Column firstNameKanji = employeeTable.addStringColumn("FIRSTNAMEKANJI");
Column lastNameKanji = employeeTable.addStringColumn("LASTNAMEKANJI");
Column firstNameKana = employeeTable.addStringColumn("FIRSTNAMEKANA");
Column lastNameKana = employeeTable.addStringColumn("LASTNAMEKANA");
Column groupid = employeeTable.addIntegerColumn("GROUPID");
Column state = employeeTable.addStringColumn("STATE");

//PrimaryKeyの設定
empno.setNullable(false);
employeeTable.setPrimaryKey(empno);

//RootTableとして登録
employeeTable.beRoot();
//metaData.setRootTable(employeeTable);

//JDBC Mediatorの作成
JDBCMediatorFactory jmf = JDBCMediatorFactory.soleInstance;
JDBCMediator jdbcMediator = jmf.createMediator(metaData,cw);

//Filterの作成
Filter filter = mdf.createFilter();
filter.setPredicate("EMPLOYEE.EMPNO = ?");

//Filterに渡す引数の作成
FilterArgument arg = mdf.createFilterArgument();
arg.setName("P_EMPNO");
arg.setType(Column.INTEGER);
filter.getFilterArguments().add(arg);

//TableにFilterをセット
employeeTable.setFilter(filter);

//引数をセットする
DataObject parms = jdbcMediator.getParameterDataObject();
parms.setInt("P_EMPNO",3);

DataObject dataObject = jdbcMediator.getGraph(parms);

//DataObjectの取得
//DataObject dataObject = jdbcMediator.getGraph(); //全検索

//DataObjectから結果の取得
Iterator employeeResults = dataObject.getList("EMPLOYEE").iterator();

while(employeeResults.hasNext()){
    DataObject employeeResult = (DataObject)employeeResults.next();
    System.out.println(

```

```

        employeeResult.getString("EMPNO")+":"+"+
        employeeResult.getString("FIRSTNAMEKANJI")+
        employeeResult.getString("LASTNAMEKANJI")+":"+"+
        employeeResult.getString("FIRSTNAMEKANA")+
        employeeResult.getString("LASTNAMEKANA")
    );
    }
} catch(SQLException e){
    e.printStackTrace();
} catch(InvalidMetadataException e){
    //createMediator()のチェック例外
    e.printStackTrace();
    e.getErrorCode();
} catch(MediatorException e){
    //getGraph()のチェック例外
    e.printStackTrace();
}
}

public void init() throws ServletException {
    try{
        //DataSourceの取得
        InitialContext ctx = new InitialContext();
        ds = (DataSource)ctx.lookup("java:comp/env/jdbc/sampledb");
    } catch(NamingException e){
        e.printStackTrace();
    }
}
}
}

```

1-3-2-6 SessionBean

コードの変更は必要ありませんが、J2EE マイグレーション・ウィザードでの配置記述のマイグレーションと再デプロイが必要になります。

1-3-2-7 EntityBean

CMPEntityBean を使用している場合、コードのマイグレーションではなく、新規に CMPEntityBean を作成することをおすすめ致します。これは、ファインダーメソッドや CMP の継承の仕様が大きく変更されている事が理由となります。

ファインダーメソッドを例にとりますと、V3.5 および V4.0 の場合[Bean 名]FinderHelper というが用意されており、この InterFace に検索条件を記述するという形なっています。これが EJB V1.1 以降になると、FinderMehod+配置記述での EJBQL の設定となります。

コード的な変更以外として、V5 以降では、分離レベルや共有属性のデフォルト値が変更された事が挙げられます。このため、デフォルト設定のままですと予想外の挙動をすることがあります。

●確認が必要となる項目

・分離レベル

デフォルト値が、REPEATABLE_READ となっていますので、ロックの挙動が変化することに注意してください。

アクセス・intentと DB 側の分離レベルの関係は、下記の表の様になります。

アクセス・intent・プロファイル	分離レベル						更新用
	DB2	Oracle	Sybase	Infomix	Cloudscape	SQL Server	
wsPessimisticUpdate Weakest LockAtLoad (デフォルト・ポリシー)	RR	RC	RR	RR	RR	RR	いいえ (*Oracle、はい)
wsPessimisticUpdate	RR	RC	RR	RR	RR	RR	はい
wsPessimisticRead	RR	RC	RR	RR	RR	RR	いいえ
wsOptimisticUpdate	RC	RC	RC	RC	RC	RC	いいえ
wsOptimisticRead	RC	RC	RC	RC	RC	RC	いいえ
wsPessimisticUpdateNo-Collisions	RC	RC	RC	RC	RC	RC	いいえ
wsPessimisticUpdate- 排他的	S	S	S	S	S	S	はい

表 1-2 アクセス・intent・プロファイルと DB 毎の分離レベル一覧

アクセス・intent を変更するには、EJB の配置記述から設定を変更する必要があります。

1. 設定対象の EJB が存在するプロジェクトの EJB デプロイメント記述子をダブルクリックします。
2. 「アクセス」タブを選択します
3. 「エンティティのデフォルト・アクセス・intent 2.x」にある設定対象の EJB を選択し、「追加」をクリックします。

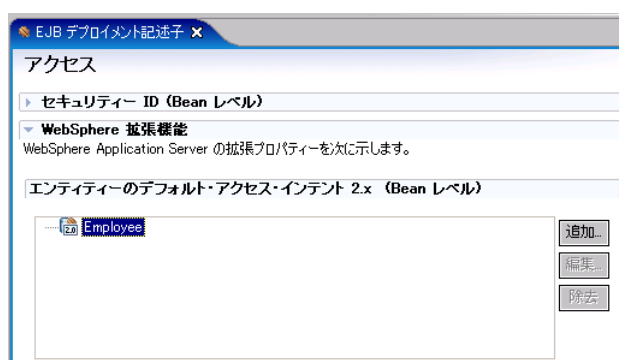


図 1-8 アクセス・intent・プロファイル設定 1

4. 「アクセス・intent 名」を選択し、「終了」をクリックします。

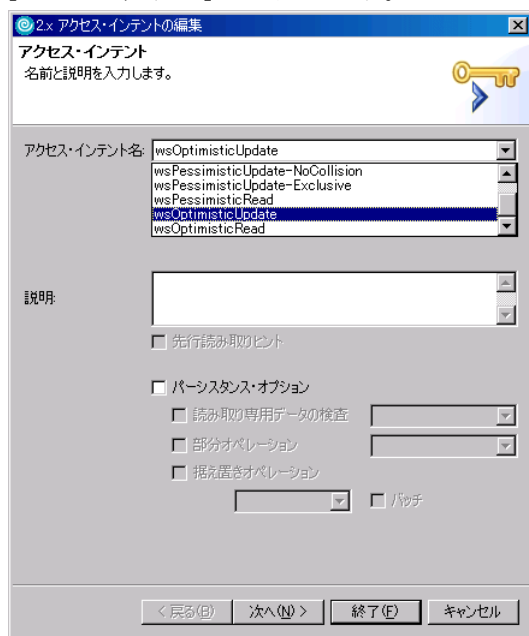


図 1-9 アクセス・intent・プロファイル設定 2

5. エディタを閉じて、設定を保存します。
6. EJB のデプロイを行います。

1-3-2-8 MDB

WAS V6 では MDB の稼働に必要な設定として、リスナー・ポート以外に活動化仕様が追加されました。アプリケーションのコードとしての変更はありませんが、使用する JMS プロバイダーにより、どちらを使用するかが変わります。

WebSphere V6 から搭載されたメッセージング・エンジンを JMS プロバイダーにする場合には、リスナー・ポートか、JMS 活動化仕様を使用し、WebSphere MQ を JMS プロバイダーにする場合は、従来通りリスナー・ポートを使用することになります。

1-3-2-9 クラス・ローダー

WAS V5.0 以降、デフォルトのクラスロードは「WebSphere extension classloader」「Application classloader」「WAR classloader」の順に行われます。さらに、アプリケーションが格納される WEB-INF/classes と、ライブラリ・パスである WEB-INF/lib、MANIFEST.MF に登録できる EAR モジュール内の Jar ファイルのロード順を含めた場合は、下記の順序で読み込まれます。

- ① WebSphere/AppServer/classes
- ② WebSphere/AppServer/lib
- ③ WebSphere/AppServer/lib/ext
- ④ MANIFEST.MF に登録された、EAR 内の other.jar
- ⑤ WEB-INF/classes
- ⑥ WEB-INF/lib

さらに、EAR/WAR モジュールのクラス・ローダー・モードとして、「親が最初」「親が最後」があること考慮すると、表のようになります。

クラス・ローダー・モード		EAR:親が最初 WAR:親が最初	EAR:親が最初 WAR:親が最後	EAR:親が最後 WAR:親が最初	EAR:親が最後 WAR:親が最後
ロード順序	1番目	AS_CLASSES	WEB-INF_CLASSES	MANIFEST.MF	WEB-INF_CLASSES
	2番目	AS_LIB	WEB-INF_LIB	AS_CLASSES	WEB-INF_LIB
	3番目	AS_LIB_EXT	AS_CLASSES	AS_LIB	MANIFEST.MF
	4番目	MANIFEST.MF	AS_LIB	AS_LIB_EXT	AS_CLASSES
	5番目	WEB-INF_CLASSES	AS_LIB_EXT	WEB-INF_CLASSES	AS_LIB
	6番目	WEB-INF_LIB	MANIFEST.MF	WEB-INF_LIB	AS_LIB_EXT

表 1-3 クラスのローディング順序

AS_CLASSES:	[AS_ROOT]/classes
AS_LIB:	[AS_ROOT]/lib
AS_LIB_EXT:	[AS_ROOT]/lib/ext
MANIFEST.MF:	WAR の MANIFEST.MF に登録した、EAR 内の JAR モジュール
WEB-INF_CLASSES	WAR モジュール内の WEB-INF/classes
WEB-INF_LIB	WAR モジュール内の WEB-INF/lib

WAS V4.0 と WAS V5.0 以降でのクラス・ローダーの設定は、下記のようにマップされます。

V5/V6 アプリケーション・クラスローダー・ポリシー	V5/V6 WAR Classloaderポリシー	V4 Module Visibility
単一	アプリケーション	Server
単一	モジュール	Compatibility
複数	アプリケーション	Application
複数	モジュール	Module
複数	モジュール	J2EE

表 1-4 クラス・ローダー設定 対応表

1-3-3 ツールによるマイグレーション

Rational Application Developer や Application Server Toolkit には、J2EE マイグレーション・ウィザードと呼ばれるマイグレーションツールが用意されています。このツールを利用することで、web.xml や ejb-jar.xml などのデプロイメント・ディスクリプターを更新できます。

ただし、更新を行うのはデプロイメント・ディスクリプターだけになりますので、アプリケーションコードの変更は別途行う必要があります。

1. マイグレーション対象のプロジェクトを選択します。
※EAR を選択することで、従属する WAR や EJB-JAR も対象とできます。
2. コンテキスト・メニューより、[マイグレーション][J2EE マイグレーション]を選択します。
3. マイグレーション対象のプロジェクトおよび J2EE バージョンなどを確認し、[次へ]をクリックします。

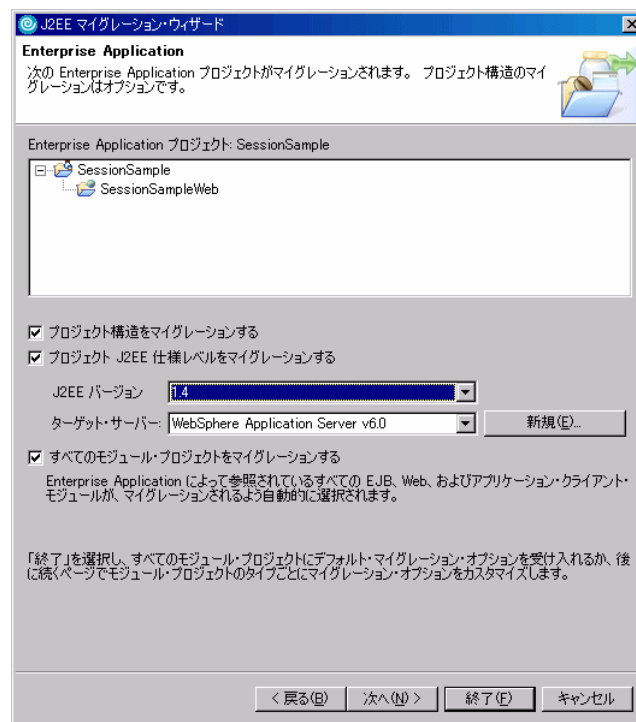


図 1-10 マイグレーション・ウィザード画面 1

4. マイグレーションが完了すると、下記ダイアログが表示されます。
※マイグレーション対象のプロジェクトに対する変更は、[詳細]をクリックすることで確認できます。

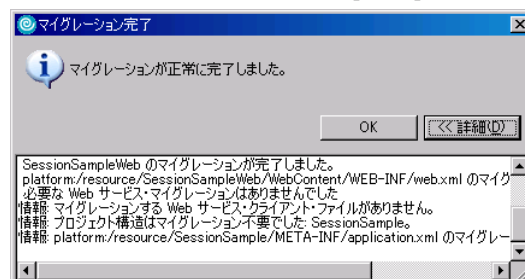


図 1-11 マイグレーション・ウィザード画面 2

1-4 アプリケーションのマイグレーション(V5 から V6 の場合)

WebSphere V5.0/V5.1 では、下記の J2EE バージョン 1.3 のコンポーネントを使用しています。

Java Servlet	2.3
JavaServer Pages (JSP)	1.2
Enterprise JavaBeans (EJB)	2.0
Java DataBase Connectivity (JDBC)	2.0
Java Message Service (JMS)	1.0.2

これが、WebSphere V6.0 になることにより、下記の J2EE バージョン 1.4 に変更する必要があります。この際のアプリケーションへの影響を説明していきます。

Java Servlet	2.4
JavaServer Pages (JSP)	2.0
Java DataBase Connectivity (JDBC)	3.0
Enterprise JavaBeans (EJB)	2.1
Java Message Service (JMS)	1.1

1-5 アプリケーションへの影響

マイグレーションを行う際に、大きく変更される項目が無いため、配置記述をマイグレーションするだけで、ほぼすべてのアプリケーションが稼働すると言えます。

1-5-1 コードの変更を極力行わない場合

WebSphere AS V6.0 でも引き続き J2EE 1.3 へのサポートが行われているため、ほとんどのアプリケーションはそのまま稼働することができます。

1-5-2 WebSphere AS V6.0 がサポートするバージョンへコードを変更する場合

WAS V5.0/V5.1 から V6.0 にマイグレーションする上で、大きく変更しなければならない点は、IBM 拡張機能関連と、sendRedirect の挙動が変更された事への対応が挙げられます。

1-5-2-1 Servlet

サーブレットとしての基本機能のマイグレーション項目では、文字コードの判別やコンテンツタイプの指定などが必須になっています。ですが、Servlet+JSP のアプリケーションであれば、ほとんど影響はありません。また、WAS の拡張機能である PageListServlet や MIME フィルターを使用している場合は再作成が必要になります。

●変更が必須となる項目

1. サーブレット構成 XML ファイルの構成情報に依存する PageListServlet を拡張している。
この PageList と互換性を持つコンポーネントは、WAS として提供されていないため、互換機能を持つサーブレットを自作する必要があります。
※WebSphere Developer Domain/Japan で互換サーブレットのサンプルが入手可能です。
http://www-06.ibm.com/jp/software/websphere/developer/wv5/35to50/appendix_1.html
2. response.sendRedirect メソッドを呼び出している。
V4/V5 では、Web アプリケーションのコンテキスト・ルートを使用していましたが、V6 (J2EE 1.4) では、挙動が変更され、Web サーバーのドキュメント・ルートを使用する様になりました。

この場合の対応策としては、下記の 2 つがあります。

1. コードの変更

response.sendRedirect(Location)の「Location」の位置を変更します。

2. com.ibm.websphere.sendredirect.compatibility の設定

WebSphere の設定で、response.sendRedirect の挙動を V4/V5 と同じにすることができます。

1. コンソールのナビゲーション・ツリーで、「サーバー」>「アプリケーション・サーバー」をクリックします。
2. 「アプリケーション・サーバー」ページで、構成対象のサーバーの名前をクリックします。
3. 選択したアプリケーション・サーバーの設定ページの「サーバー・インフラストラクチャー」で、「Java およびプロセス管理」>「プロセス定義」をクリックします。

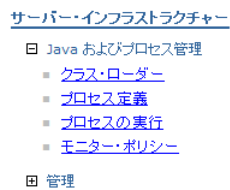


図 1-12 response.sendRedirect 設定 1

4. 「プロセス定義」ページで、「Java 仮想マシン」をクリックします。

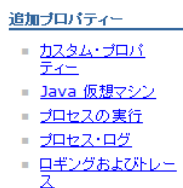


図 1-13 response.sendRedirect 設定 2

5. 「Java Virtual Machine」ページで、「カスタム・プロパティ」をクリックします。

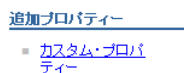


図 1-14 response.sendRedirect 設定 3

- 「カスタム・プロパティ」ページで「新規」をクリックします。



図 1-15 response.sendRedirect 設定 4

- プロパティの設定ページで、名前に「com.ibm.websphere.sendredirect.compatibility」を、値に「true」を指定して、「OK」をクリックします。

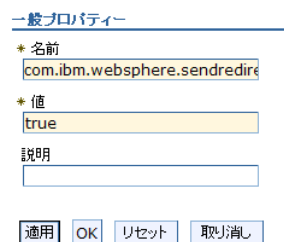


図 1-16 response.sendRedirect 設定 5

- コンソールのタスクバーで、「保管」をクリックします。
- アプリケーション・サーバーを停止してから、アプリケーション・サーバーを再始動します。

1-5-2-2 JSP

必須の変更はありませんが、[tsx:]で始まる JSP タグが非推奨となりましたので、このタグを使用している場合は、Java 標準の JSTL への置き換えが推奨されます。

下記の表に沿って、タグを変更してください。

tsx tag	JSTL tag
tsx:repeat	c:forEach
tsx:dbconnect	sql:setDataSource
tsx:userid	sql:setDataSourceタグのuser属性
tsx:passwd	sql:setDataSourceタグのpassword属性
tsx:dbquery	sql:query
tsx:dbmodify	sql:update
tsx:getProperty	式言語 (EL)を使用する

表 1-5 tsx タグと JSTL タグの対応表

たとえば、tsx を使用した実装で、下記のようなコードの場合

```
<tsx:repeat start="request.getParameter("index_a")" end="<%=request.getParameter("index_b") %>"
index="idx">
<%=idx %>
</tsx:repeat>
```

JSTL で実装した場合下記のようになります。

```
<c:forEach var="index" end="${param.index_b}" begin="${param.index_a}">
<c:out value="${index}" />
</c:forEach>
```

1-5-2-3 JMS

V6 から、JMS1.1 がサポートされました。

JMS1.0 とのコード的な違いとしては、JMS 接続ファクトリーが共通化された事の変更点となります。

JMS1.0 のコードから、JMS1.1 のコードへ変更を行うには、`javax.jms.QueueConnectionFactory` を `javax.jms.ConnectionFactory` に変更し、`javax.jms.QueueConnection` を `javax.jms.Connection` に変更する事になります。

上記仕様クラスの変更以外のコードは、そのまま使用する事ができます。

ここでは、`ConnectionFactory` などのオブジェクトの作成例を、提示します。

使用するクラスが変更されている以外のコードの変更が無いことが見ていただけると幸いです。

JMS1.0

```
QueueConnectionFactory    qcf = (QueueConnectionFactory)ctx.lookup(jndiNameCF);
QueueConnection           qCon = qcf.createQueueConnection();
QueueSession              qSession= qCon.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
QueueSender               qSender = qSession.createSender(queue);
```

JMS1.1

```
ConnectionFactory cf = (ConnectionFactory)ctx.lookup(jndiNameCF);
Connection         Con = cf.createQueueConnection();
Session            Session = Con.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
Sender             Sender = Session.createSender(queue);
```

1-5-2-4 データアクセス

従来の `DataAccessBean` が非推奨となりましたので、同様の動作を行う物を自作するか、`SDO` への移行を行う必要があります。

・`DataAccessBeans` を使用しているアプリケーション

`SDO` への移行が推奨されていますが、`SDO` は通常の `JDBC`(データ・ソース)アクセスとは、アーキテクチャが大きく異なりますので、用途に応じた再作成が必要となります。

※`SDO` に関しては、[4-3-2-5 データアクセス]の補足「`SDO`」についてを参照してください。

1-5-2-5 MDB

WAS V6 では `MDB` の稼働に必要な設定として、リスナー・ポート以外に活動化仕様が追加されました。アプリケーションのコードとしての変更はありませんが、使用する `JMS` プロバイダーにより、どちらを使用するかが変わります。

`WebSphere V6` から搭載されたメッセージング・エンジンを `JMS` プロバイダーにする場合には、リスナー・ポートか、`JMS` 活動化仕様を使用し、`WebSphere MQ` を `JMS` プロバイダーにする場合は、従来通りリスナー・ポートを使用することになります。

1-5-3 ツールによるマイグレーション

Rational Application Developer や Application Server Toolkit には、J2EE マイグレーション・ウィザードと呼ばれるマイグレーションツールが用意されています。このツールを利用することで、web.xml や ejb-jar.xml などのデプロイメント・ディスクリプターを更新できます。

ただし、更新を行うのはデプロイメント・ディスクリプターだけになりますので、アプリケーションコードの変更は別途行う必要があります。

1. マイグレーション対象のプロジェクトを選択します。
※EAR を選択することで、従属する WAR や EJB-JAR も対象とできます。
2. コンテキスト・メニューより、[マイグレーション][J2EE マイグレーション]を選択します。
3. マイグレーション対象のプロジェクトおよび J2EE バージョンなどを確認し、[次へ]をクリックします。
4. マイグレーションが完了すると、下記ダイアログが表示されます。
※マイグレーション対象のプロジェクトに対する変更は、[詳細]をクリックすることで確認できます。

1-6 その他の注意点

ここまでは、Servlet や EJB など一般的な J2EE アプリケーションのマイグレーションについて説明しました。ここからは、WAS のバージョンアップによる変更などについて、紹介します。

1. スレッドの扱い

J2EE では、スレッドの扱いについて制限が設けられています。このため、旧バージョンで稼働していた物が、稼働しなくおそれがあります。

- EJB コンテナでは、スレッドの作成・開始・終了は禁止されています。
- トランザクションにかかわるオブジェクト (java.sql.Connection など) は、スレッドをまたいで共有できません。
- HttpServletRequest や HttpServletResponse のインスタンスは、サーブレットの service 実行スレッド内でのみ有効です。

ですので、スレッドを別にする必要があるアプリケーションでは、メッセージングなどの非同期処理を行うなどの変更をお勧めします。

2. 製品情報を取得するための API

com.ibm.websphere.product.product および com.ibm.websphere.product.buildInfo パッケージの利用 com.ibm.websphere.product.product および com.ibm.websphere.product.buildInfo クラスのすべてのメソッドおよびフィールドは使用すべきではありません。

このため、com.ibm.websphere.product.WASProduct のクラス (com.ibm.websphere.product.product および com.ibm.websphere.product.buildInfo のオブジェクトを含む) の以下のメソッドは使用すべきではありません。

```
public product getProductByFilename(String basename)
public product getProductById(String id)
public boolean productPresent(String id)
public boolean addProduct(product aProduct)
public boolean removeProduct(product aProduct)
public Iterator getProducts()
public Iterator getProductNames()
```

```
public String loadVersionInfoAsXMLString(String filename)
public String getProductDirName()
public static String computeProductDirName()
```

com.ibm.websphere.product.WASDirectory の以下のサポートされるメソッドを使用してください。

```
public WASProductInfo getWASProductInfo(String id)
public boolean isThisProductInstalled(String id)
public WASProductInfo[] getWASProductInfoInstances()
public String getWasLocation()
```

また、製品情報（製品名、バージョン、ビルド・レベル、ビルド日）を古い WASProduct API (com.ibm.websphere.product.WASProduct) から入手する代わりに、WASDirectory クラスの以下のメソッドを使用してその情報を入手してください。

```
com.ibm.websphere.product.WASDirectory.getName(String)
com.ibm.websphere.product.WASDirectory.getVersion(String)
com.ibm.websphere.product.WASDirectory.getBuildLevel(String)
com.ibm.websphere.product.WASDirectory.getBuildDate(String)
```

3. V3.5 からのマイグレーション

V3.5 からの移行では、使用される API の違いのみならず、アプリケーションのデプロイ方法などが大きく異なります。V3.5 から V6 の間で大きくアーキテクチャが変更されるのは、V3.5 から V5.0 といえますので、まずは V5.0 へのマイグレーション方法を参照してください。

参考)

WebSphere Developer Domain 「WAS V3.5 から V5.0 への移行ガイド」

<http://www.ibm.com/jp/software/websphere/developer/wv5/35to50/>

1-7 開発ツールのマイグレーション

ここでは、既存の開発環境である WebSphere Studio Application Developer(Studio)や VisualAge for Java(VAJ)から Rational Application Developer(RAD)へ移行する方法を紹介します。

1-7-1 VisualAge for Java + WebSphere Studio V4.0 からのマイグレーション

VisualAge for Java(VAJ)+WebSphere Studio(Classic Studio)から直接マイグレーションを行うことはできません。WAR や EJB JAR としてエクスポートした物を RAD にインポートし直すことになります。

VisualAge for Java エンタープライズ版にある、リポジトリ・サーバーで構成管理を行っている場合も同様です。RAD では、構成管理ツールとして ClearCase LT (ClearCase も可能)もしくは CVS を使用することになります。

1-7-1-1 サーブレット+JSP 開発環境のマイグレーション

Servlet+JSP の開発環境を VAJ+Classic Studio から RAD に移行する場合、Classic Studio から WAR ファイルをエクスポートし、その WAR ファイルを RAD にインポートした後、マイグレーションを行います。

1. Classic Studio からのエクスポート

1. Classic Studio の「servlet」フォルダ以下にある java のクラス・ファイルおよびソース・ファイルを選択します。

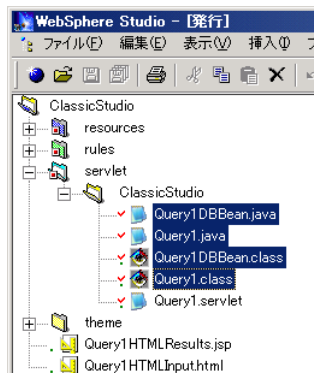


図 1-17 Studio と VAJ の同期 1

2. 「プロジェクト」>「VisualAge for Java」>「VisualAge から更新」を選択して、VAJ から更新されたソース・ファイルとクラス・ファイルを取得します。

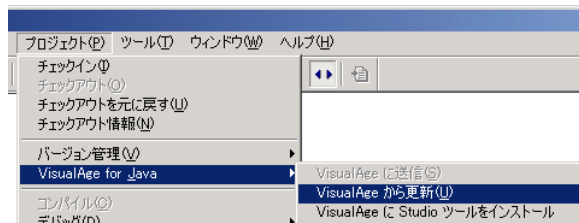


図 1-18 Studio と VAJ の同期 2

3. 「プロジェクト」>「Web アーカイブ・ファイルの作成」を選択します。

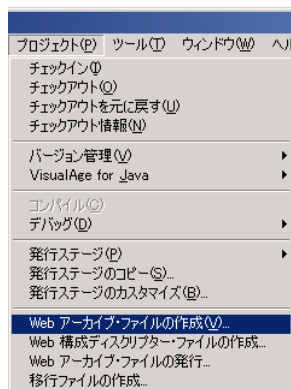


図 1-19 Studio による、Web アーカイブ・ファイルの作成 1

4. 「Web アーカイブ・ファイルの作成」ダイアログが表示されます。もし、「Web 構成ディスクリプター・ファイル名」がない場合は、「作成」をクリックし作成してください。

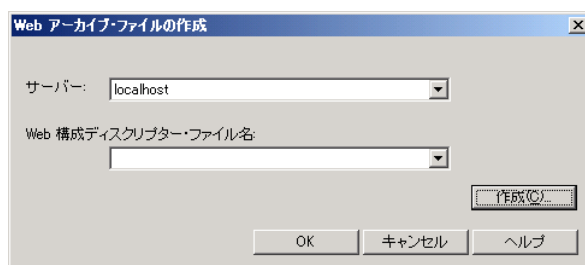


図 1-20 Studio による、Web アーカイブ・ファイルの作成 2

5. 「Web 構成ディスクリプター・ファイルの作成」を行う画面では、「ディスクリプター・ファイル名」を「WEB-INF/web.xml」に、「サーブレット」ではサーブレットとして登録するクラスを選択します。設定が完了したら、「作成」をクリックします。

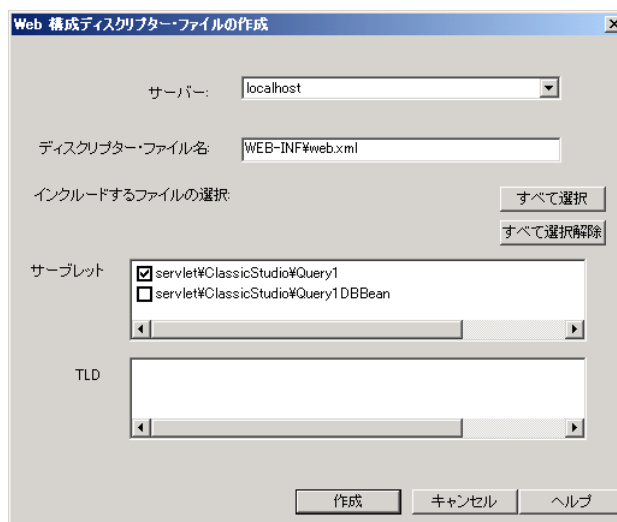


図 1-21 Studio による、Web アーカイブ・ファイルの作成 3

6. 「Web アーカイブ・ファイルの作成」ダイアログに戻りますので、「OK」をクリックします。
7. ファイル作成のダイアログが表示されますので、任意の場所に作成してください。

2. VAJ からのソース・ファイルのエクスポート

1. エクスポート対象となるプロジェクトを選択し、コンテキスト・メニューから「エクスポート」を選択します。

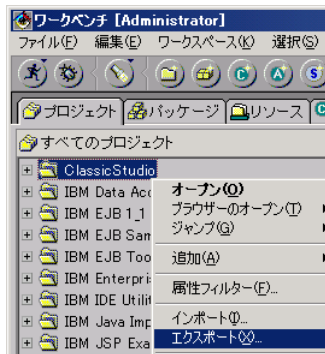


図 1-22 VAJ からのエクスポート 1

2. 「エクスポート」のダイアログが表示されますので、「ディレクトリ」を選択します。

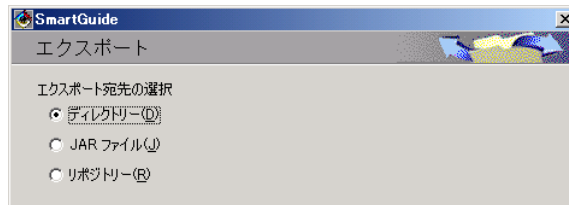


図 1-23 VAJ からのエクスポート 2

3. 必要に応じてフィールドに入力し（「java」チェック・ボックスが必ず選択されるようにしてください）、データベースを選択して、「終了」をクリックします。

3. RAD へのインポート

1. 「ファイル」>「インポート」を選択します。



図 1-24 RAD でのインポート 1

2. 「インポート」のダイアログが表示されますので、「WAR ファイル」を選択して、「次へ」をクリックします。

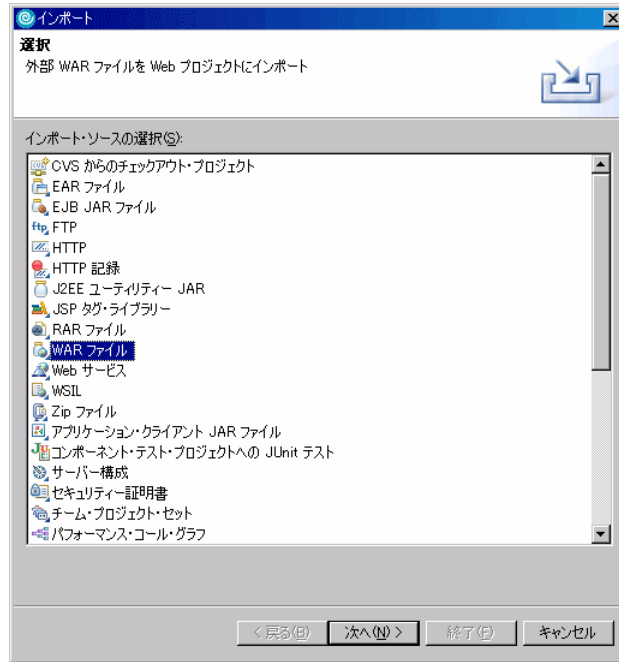


図 1-25 RAD でのインポート 2

3. 「WAR ファイル」に、Classic Studio からエクスポートした WAR ファイルを指定し、必要に応じて「Web プロジェクト」および「EAR プロジェクト」の名称を変更します。設定が完了したら、「終了」をクリックします。



図 1-26 RAD でのインポート 3

4. 作成された Web プロジェクトを展開します。
5. VAJ からエクスポートしたクラスをパッケージレベルで、作成された Web プロジェクトの「JavaSouces」にドラッグアンドドロップします。

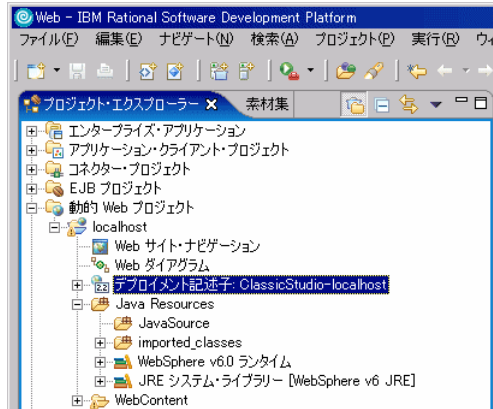


図 1-27 RAD でのインポート 4

4. 構成ファイルのマイグレーション

1. 作成された、EAR プロジェクトを選択し、コンテキスト・メニューより、[マイグレーション][J2EE マイグレーション]を選択します。
2. マイグレーション対象のプロジェクトおよび J2EE バージョンなどを確認し、[次へ]をクリックします。
3. マイグレーションが完了すると、完了ダイアログが表示されます。

1-7-1-2 EJB 開発環境のマイグレーション

EJB の開発環境を VAJ から RAD に移行する場合、VAJ から ejb.jar の形で EJB をエクスポートし、その ejb-jar を RAD にインポートした後、マイグレーションを行います。

1. VAJ からのエクスポート

1. 「F2」キーを押し、「クイック・スタート」を表示させます。
2. 左側のペインにある「フィーチャー」を選択後、右側のペインに表示される「フィーチャーの追加」を選択します。

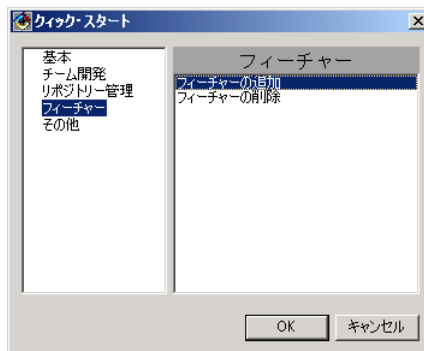


図 1-28 VAJ フィーチャーの追加 1

3. フィーチャーの一覧が表示されますので、「Export Tool for Enterprise Java Beans 1.1」を選択し「OK」をクリックします。

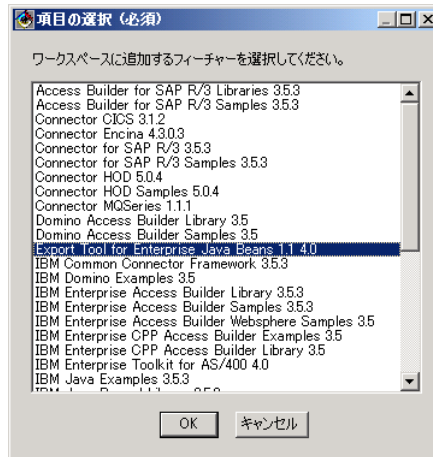


図 1-29 VAJ フィーチャーの追加 2

4. 「ワークベンチ」の EJB ページで、エクスポートしたい EJB グループを右マウス・ボタン・クリックし、「エクスポート」->「EJB 1.1 JAR」とクリックします。

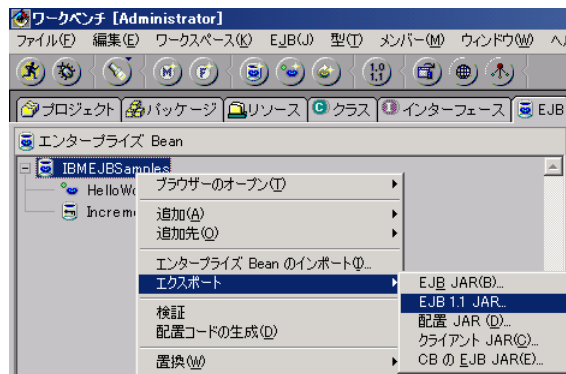


図 1-30 VAJ から EJB のエクスポート 1

5. 必要に応じてフィールドに入力し（「.java」チェック・ボックスが必ず選択されるようにしてください）、データベースを選択して、「終了」をクリックします。

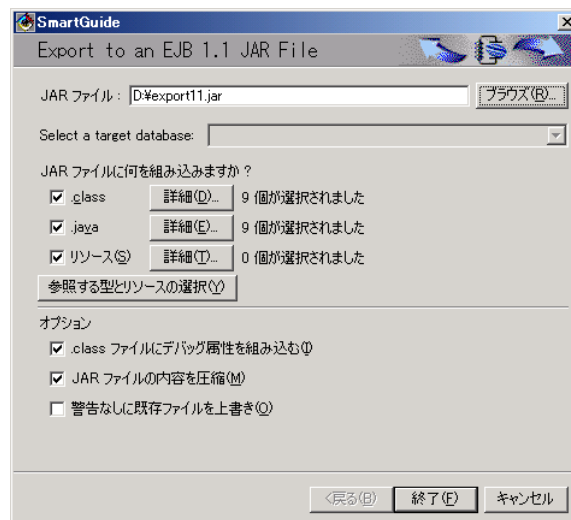


図 1-31 VAJ から EJB のエクスポート 2

2. RAD へのインポート

1. 「ファイル」>「インポート」を選択します。
2. 「インポート」のダイアログが表示されますので、「EJB JAR ファイル」を選択して、「次へ」をクリックします。

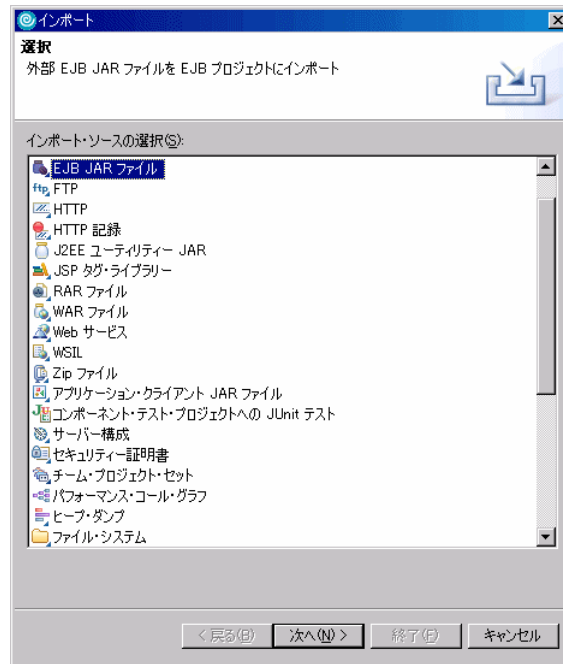


図 1-32 RAD へのインポート 1

3. 「EJB JAR ファイル」に、VAJ からエクスポートした EJB JAR ファイルを指定し、必要に応じて「EJB JAR プロジェクト」および「EAR プロジェクト」の名称を変更します。設定が完了したら、「終了」をクリックします。

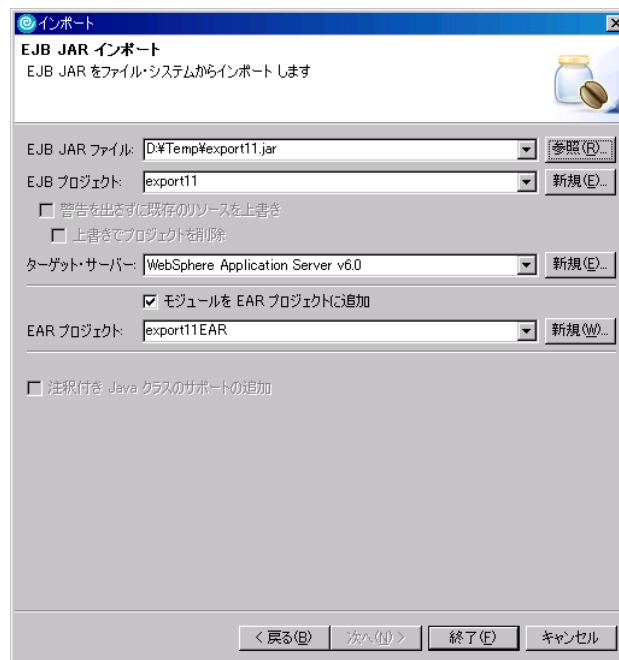


図 1-33 RAD へのインポート 2

3. 構成ファイルのマイグレーション

1. 作成された、EAR プロジェクトを選択し、コンテキスト・メニューより、[マイグレーション][J2EE マイグレーション]を選択します。
2. マイグレーション対象のプロジェクトおよび J2EE バージョンなどを確認し、[次へ]をクリックします。
3. マイグレーションが完了すると、完了ダイアログが表示されます。

1-7-2 WebSphere Studio Application Developer から のマイグレーション

WASD から RAD へ移行する場合、ワークスペースをそのままマイグレーションすることがマイグレーションの近道です。このような場合、ワークスペースのコピーを取得していただいた上で、RAD から既存のワークスペースを開きます。

1. 既存ワークスペースを、ファイルレベルでバックアップします。
2. RAD を起動し、既存ワークスペースを選択します。

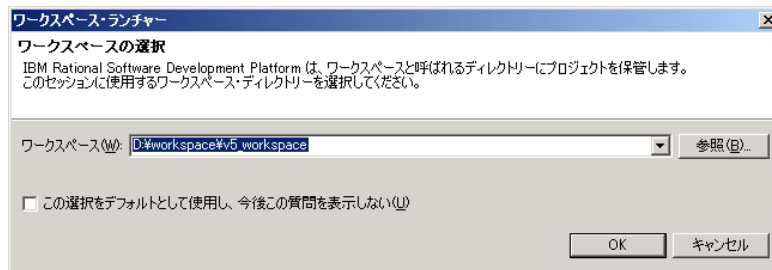


図 1-34 RAD ワークスペース選択画面

3. RAD 起動後、「ワークベンチのレイアウトを復元できません」というメッセージが表示されますので、そのまま「OK」をクリックします。

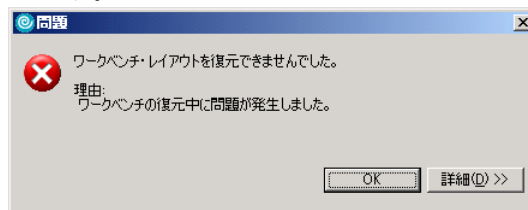


図 1-35 RAD エラー画面

4. ようこそ画面を閉じます。
5. [ウィンドウ]より[パースペクティブのリセット]を選択します。

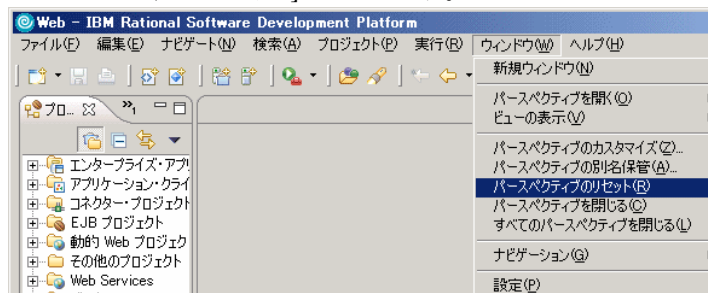


図 1-36 RAD のパースペクティブのリセット 1

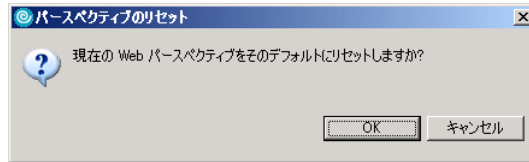


図 1-37 RAD のパースペクティブのリセット 2

ワークスペースをマイグレーションした際には、下記のように旧バージョンのライブラリをマイグレーションするかどうかを確認するダイアログが表示されることがあります。RAD の持つライブラリにマイグレーションするのであれば、[適用する]をクリックし、ライブラリのマイグレーションを行ってください。

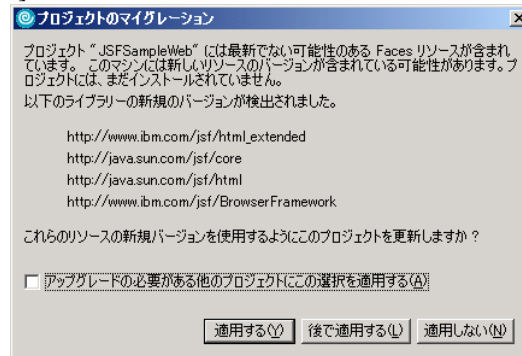


図 1-38 RAD ライブラリのアップグレード確認画面