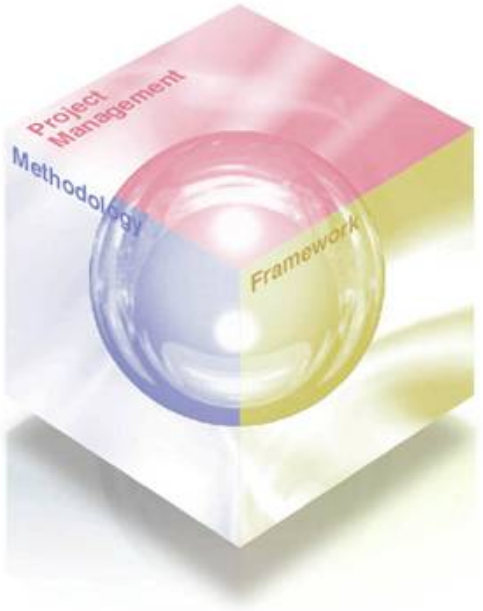
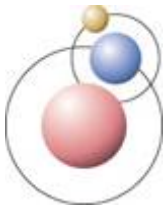
TERASOLUNA® Server Framework for Java（Rich版）
アーキテクチャ説明書
第2.0.0.2版



株式会社NTTデータ

TERASOLUNA®
BEST SOLUTION FOR SYSTEM DEVELOPMENT

NTT DATA



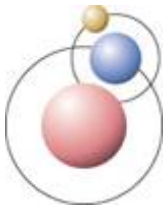
- 本ドキュメントを使用するにあたり、以下の規約に同意していただく必要があります。同意いただけない場合は、本ドキュメント及びその複製物の全てを直ちに消去又は破棄してください。

1. 本ドキュメントの著作権及びその他一切の権利は、NTTデータあるいはNTTデータに権利を許諾する第三者に帰属します。
2. 本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、およびNTTデータの著作権表示を削除することはできません。
3. 本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「参考文献:TERASOLUNA Server Framework for Java (Rich版)アーキテクチャ説明書」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。
4. 前2項によって作成したドキュメント及びその複製物を、無償の場合に限り、第三者へ提供することができます。
5. NTTデータの書面による承諾を得ることなく、本規約に定められる条件を超えて、本ドキュメント及びその複製物を使用したり、本規約上の権利の全部又は一部を第三者に譲渡したりすることはできません。
6. NTTデータは、本ドキュメントの内容の正確性、使用目的への適合性の保証、使用結果についての的確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。
7. NTTデータは、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求(第三者との間の紛争を理由になされる請求を含む。)に関しても、NTTデータは一切の責任を負いません。

- 本ドキュメントで使用されている各社の会社名及びサービス名、商品名に関する登録商標および商標は、以下の通りです。

  - Apache、Tomcatは、Apache Software Foundationの登録商標または商標です。
  - Java, JDK, J2SE, J2EE, JSP, Servletは、米国 Sun Microsystems, Inc.の米国及びその他の国における登録商標または商標です。
  - Oracleは、米国Oracle International Corp.の米国及びその他の国における登録商標または商標です。
  - TERASOLUNAは、株式会社NTTデータの登録商標です。
  - WebLogicは、BEA Systems Inc.の登録商標または商標です。
  - WebSphereは、IBM Corporationの登録商標または商標です。
  - その他の会社名、製品名は、各社の登録商標または商標です。

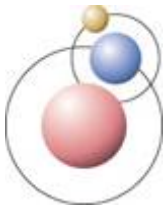
TERASOLUNA



目次

- はじめに
- アーキテクチャ概観
- ①制御情報保持
- ②リクエスト・コントローラマッピング
- ③コントローラ拡張
- ④リクエストデータ解析
- ⑤入力チェック
- ⑥サービス実行
- ⑦トランザクション管理
- ⑧データベースアクセス
- ⑨レスポンスデータ生成
- ⑩メッセージ管理
- ⑪例外処理

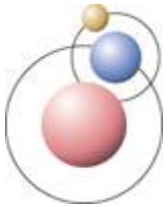
TERASOLUNA



はじめに

概要

- 本資料は、リッチクライアントシステムを構築するための サーバ側フレームワーク「TERASOLUNA Server Framework for Java（Rich版）」を解説した資料である。
- 本フレームワークは、Spring Framework2.0をベースに拡張を行ったフレームワークとなっている。

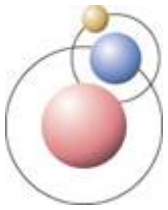


はじめに

動作確認環境

- 対応JDK
  - J2SE5.0
- 対応Webサーバ
  - Tomcat5.5
  - WebLogic Server 9.2J
  - WebSphere6.1
- 対応データベース
  - Oracle10g (10.2.0)
  - Oracle 9i (9.2.0)
  - PostgreSQL8.2

TERASOLUNA



はじめに

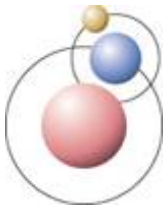
必要モジュール

パターン1

- terasoluna-rich-server
  - Rich版の機能を全て統合した実装したモジュール。
  - これひとつで terasoluna-commons、terasoluna-rich、terasoluna-oxm、terasoluna-dao、terasoluna-ibatis、terasoluna-validator の各モジュールの機能を提供する。

パターン2

- terasoluna-rich
  - Rich版の固有機能を提供するモジュール
- terasoluna-oxm
  - オブジェクト-XML変換機能を提供するモジュール
- terasoluna-commons
  - ユーティリティ機能など共通機能を提供するモジュール



はじめに

任意選択モジュール

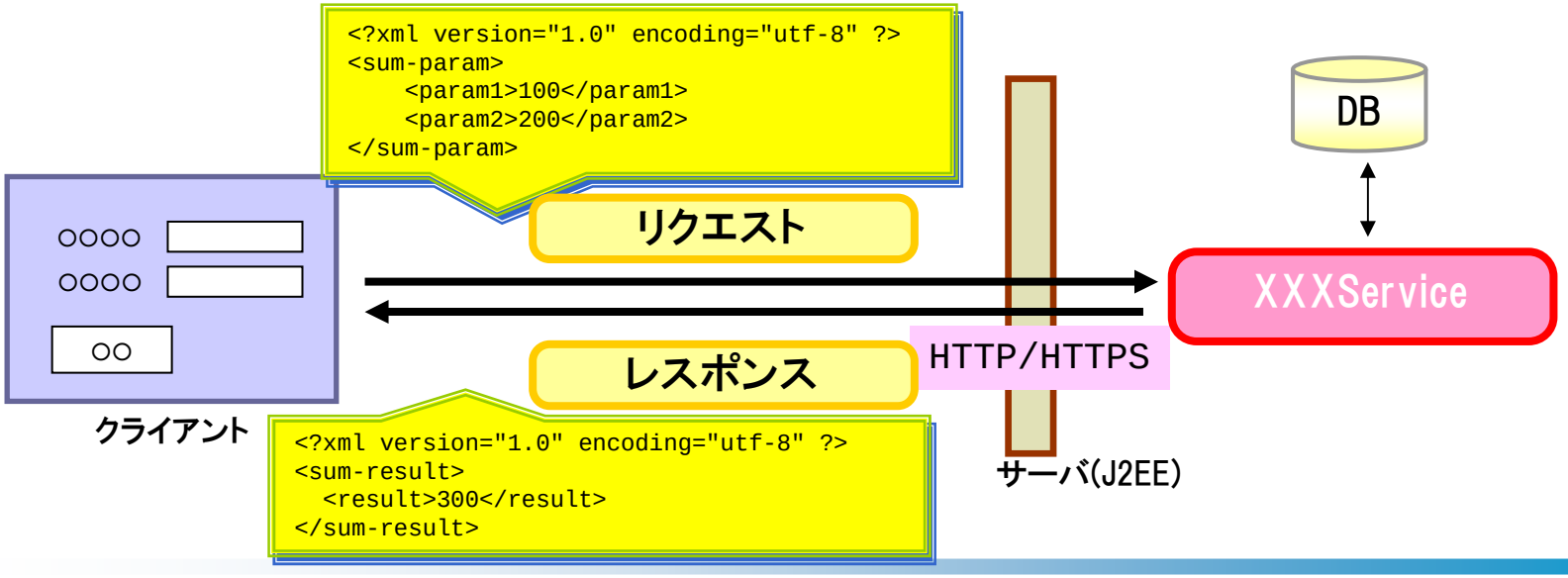
- terasoluna-validator
  - 入力チェック機能を提供するモジュール
- terasoluna-dao
  - DAOインタフェースを提供するモジュール
- terasoluna-ibatis
  - ORマッピングツールiBatisを利用した、データベースアクセス機能を提供するモジュール

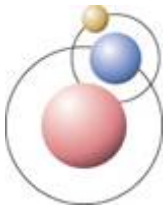
※ terasoluna-rich-serverを使用する場合はこれらのモジュールは不要

はじめに

サーバ・クライアント間の通信方式

- クライアント・サーバ間は、HTTPまたはHTTPSで通信する
  - リクエストはXML、またはクエリ形式で、POSTデータを送信する
  - レスポンスは、XML、またはバイナリ形式とする
- サーバはセッション状態を保持しない（任意のサーバにリクエストを振り分け可能）





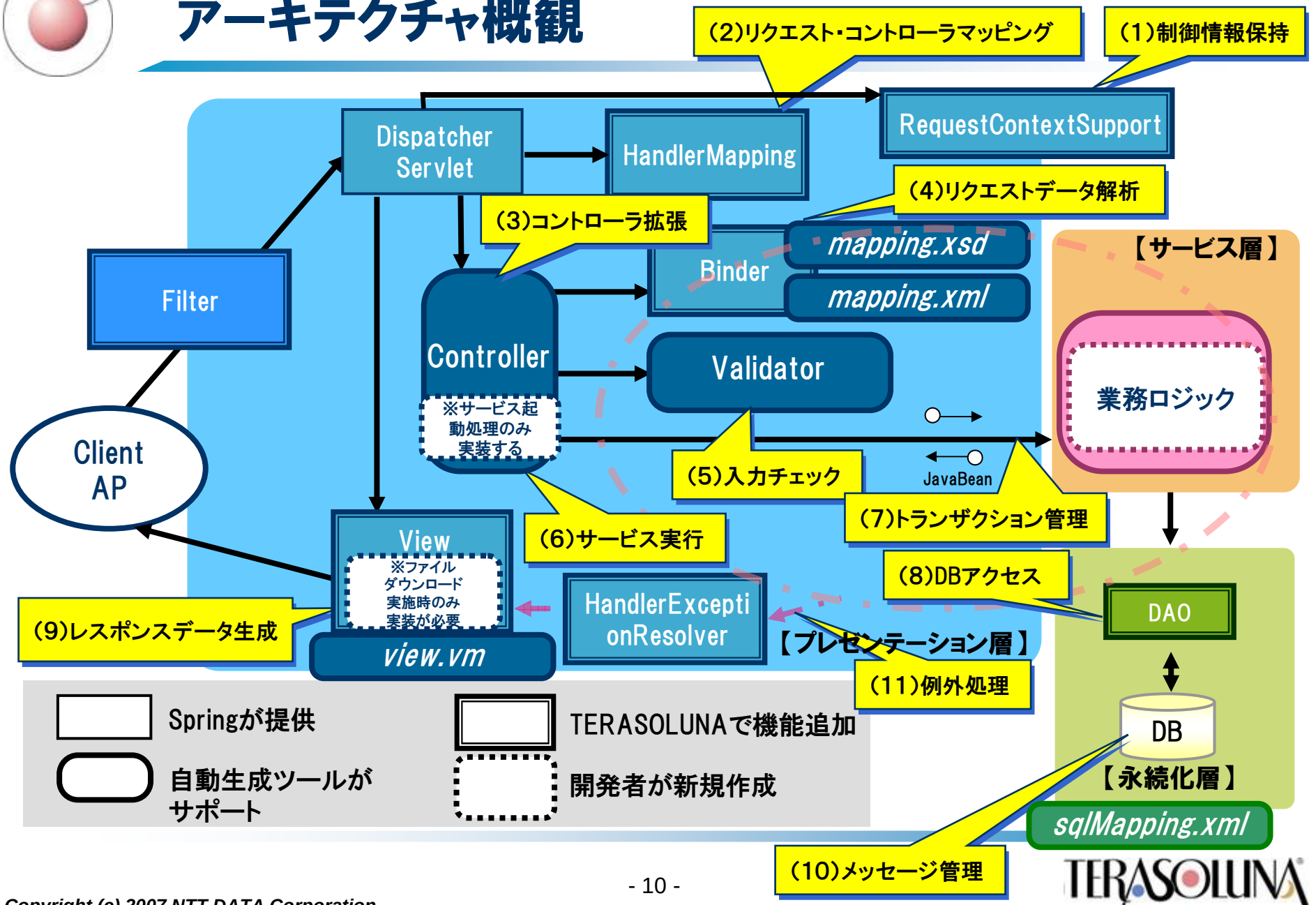
はじめに

クライアント実装について

- クライアント側の実装は、前述の通信方式(XML形式またはクエリ形式、マルチパート形式をPOSTデータとして送信、応答電文はXML、バイナリファイルとする)を満たしていれば、どのような実装でも適用できる
  - これらの方式に該当しない場合(例えば、Webサービスを利用する場合)は、サーバ側のWeb層を別途用意する必要がある
- クライアント側の適用事例としては、次のものがある
  - Microsoft .NET
  - VisualFrame
  - swing
  - Curl
  - Ajax

TERASOLUNA

アーキテクチャ概観

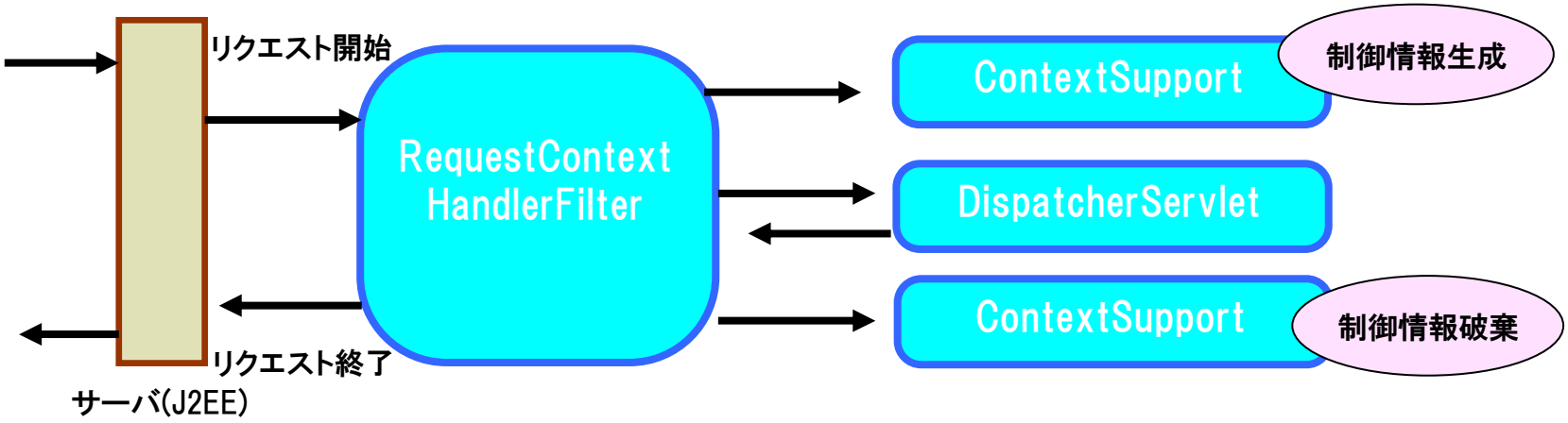


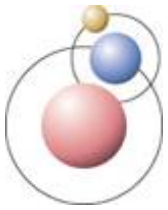
①制御情報保持

制御情報とは

◆リクエスト処理で使用する共通の情報

- コントローラを呼び出すための識別子(リクエスト名)やシステム内で共通に使用する情報のこと





①制御情報保持

RequestContext

◆制御情報を保持するためのクラス

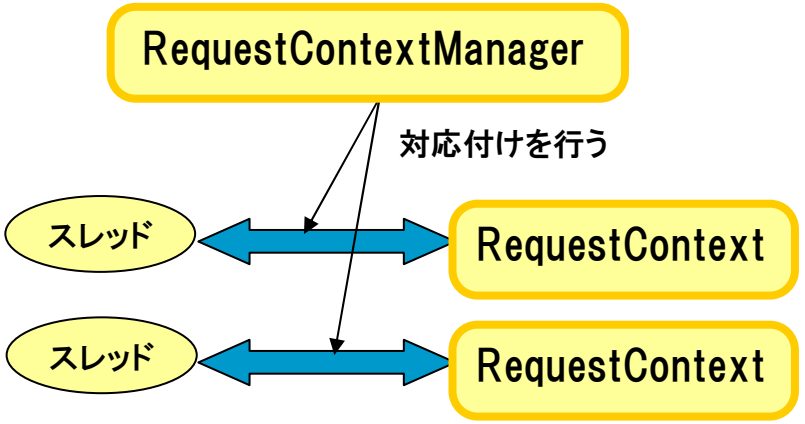
- リクエスト名
  - コントローラを呼び出すための識別子
  - 後述のコントローラと1:1の関係になる
  - 通常、クライアントは、HTTPリクエストをサーバへ送信する際、HTTPヘッダにリクエスト名を設定する
- 業務プロパティ
  - システム内で共通に使用する情報
  - java.util.Properties型で保持されている
  - 使用する情報は、システムの要件によって異なる

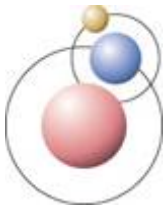
①制御情報保持

RequestContextManager

◆制御情報の管理クラス

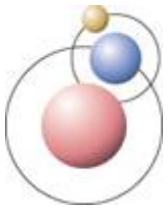
- スレッド単位で制御情報を管理する
  - スレッドごとに、一意の制御情報を扱うことができる
- 一つのスレッドでリクエストを処理することを想定している
  - EJBを利用した場合は、EJBから呼ばれる業務ロジックは別スレッドになるため注意が必要





①制御情報保持

- RequestContextSupport
  - 制御情報を扱うためのインタフェース
- AbstractRequestContextSupport
  - RequestContextSupportインタフェース実装クラス
    - RequestContextManagerを利用した実装を提供
  - リクエストデータをRequestContextに設定する抽象メソッドを定義
- DefaultRequestContextSupportImpl
  - AbstractRequestContextSupportのサブクラス
  - HTTPリクエストのヘッダ”requestName”の値の文字列をリクエスト名としてRequestContextに設定する



①制御情報保持

RequestContextHandlingFilter

- リクエスト開始・終了時にWebアプリケーションから呼び出されるフィルタ
  - リクエスト開始時に、制御情報を生成する
  - リクエスト終了時に、制御情報を破棄する

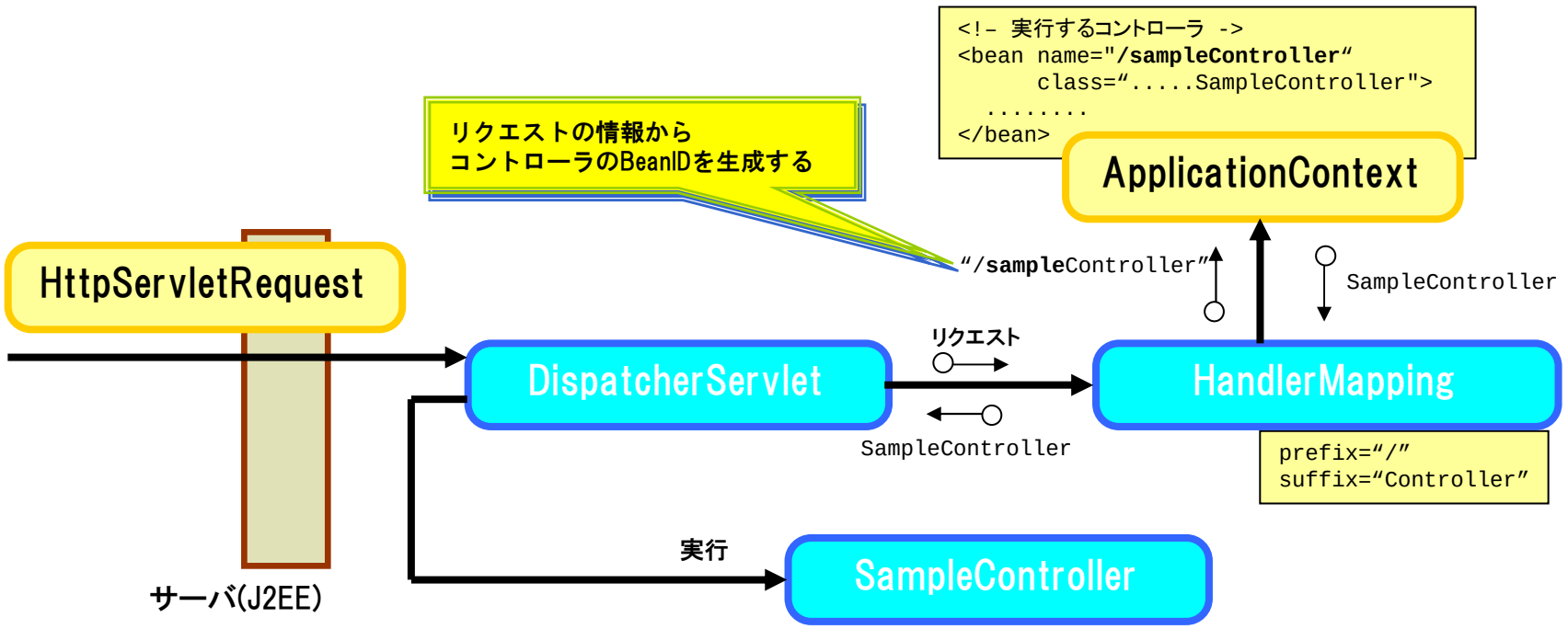
RequestContextHandlingFilterのweb.xml 設定例

```
<web-app>
  (略)
    <filter>
      <filter-name>
        requestContextHandlingFilter
      </filter-name>
      <filter-class>
        jp.terasoluna.framework.web.RequestContextHandlingFilter
      </filter-class>
    </filter>
    <filter-mapping>
      <filter-name>
        requestContextHandlingFilter
      </filter-name>
      <url-pattern>
        /*
      </url-pattern>
    </filter-mapping>
  (略)
```

②リクエスト・コントローラマッピング

ハンドラマッピング(HandlerMapping)とは

- リクエストとコントローラの対応付けを行うインタフェース



②リクエスト・コントローラマッピング

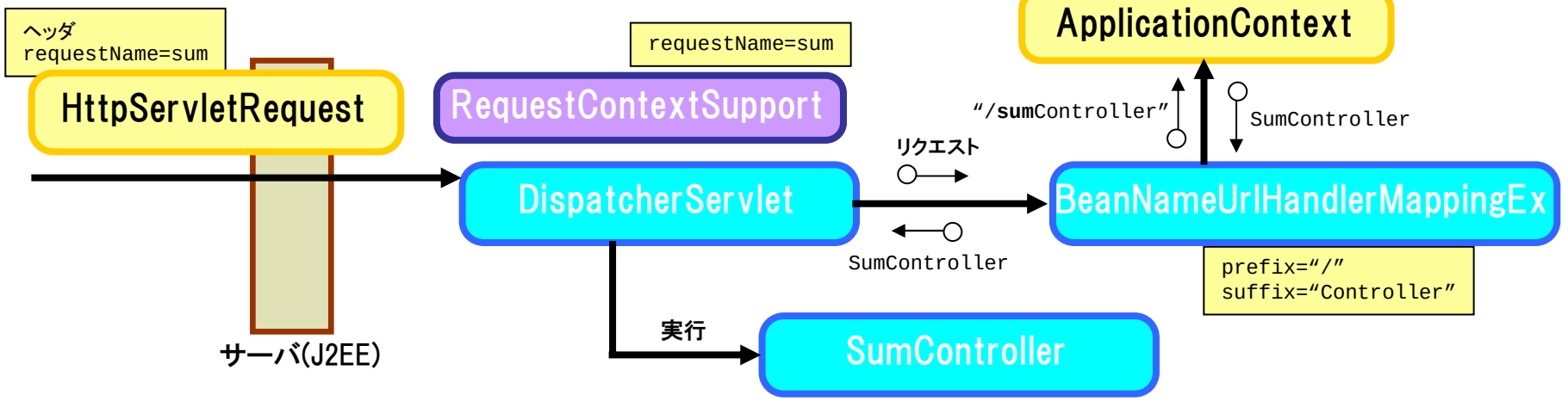
BeanNameUrlHandlerMappingEx

TERASOLUNAで提供するHandlerMappingの実装クラス

- 制御情報であるリクエスト名を使って、実行するコントローラとの対応付けを行うハンドラ

【リクエストとコントローラの対応付けの法則】
(Bean定義の)コントローラID＝プリフィックス＋リクエスト名＋サフィックス
プリフィックスとサフィックスはHandlerMappingのBean定義に記述する。
(下記の例) "/"+"sum"+"Controller"⇒/sumController

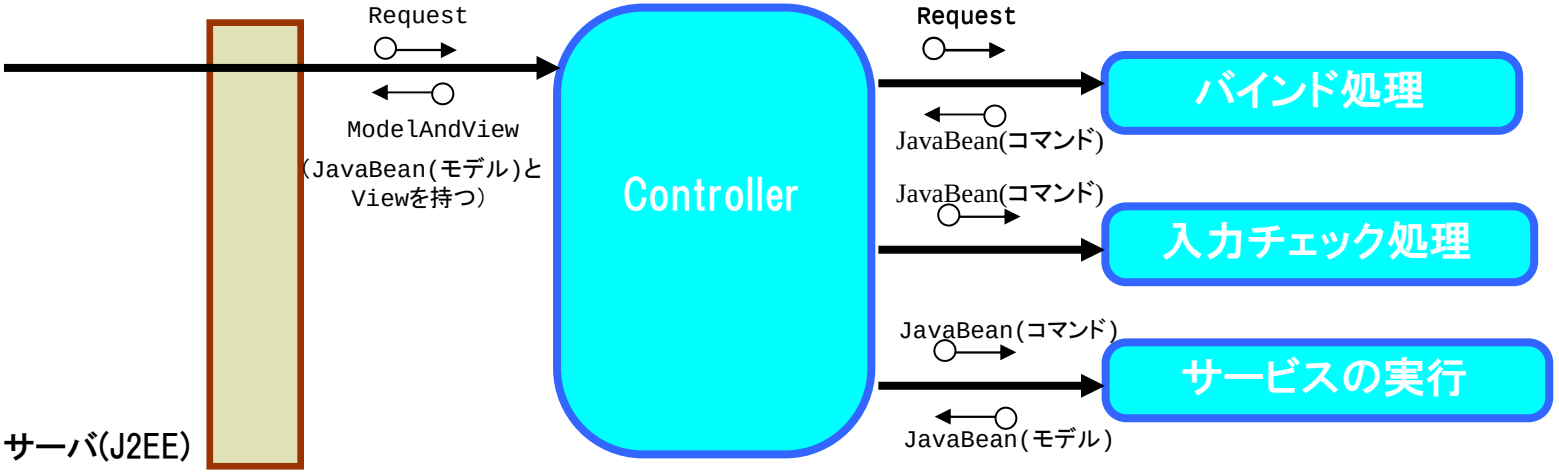
```
<!- 実行するコントローラ ->
<bean name="/sumController"
      class="....SumController">
  ........
</bean>
```

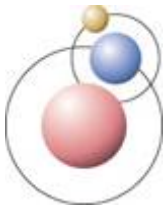


③コントローラ拡張

コントローラ(Controller)とは

- リクエスト処理メソッドを定義するインタフェース
  - リクエスト処理フローを定義する役割を持つ
  - TERASOLUNAでは、Springフレームワークの Controller実装である AbstractCommandControllerクラスを拡張する





③コントローラ拡張

TerasolunaController

◆本クラスでは以下の処理を行う

1. リクエストデータをBinderでJavaBeanにバインドする
   - リクエストデータからバインドされたJavaBeanを「コマンド」と呼ぶ
     - » 詳細は「④リクエストデータ解析」を参照すること
2. JavaBean(コマンド)に対して入力チェックを実行する
   - 詳細は「⑤入力チェック」を参照すること
3. サービス層のインタフェースを呼び出す
   - JavaBean(コマンド)を引数に、サービス層のオブジェクトを実行し、戻り値としてJavaBeanを受け取る
   - サービス層から返されるJavaBeanを「モデル」と呼ぶ
     - » 詳細は「⑥サービスの実行」を参照すること
4. ビュー名を設定する
   - ここで設定されたビュー名を使用してServletはビューを生成する
     - » ビュー名の設定についての詳細は「⑦レスポンスデータ生成」を参照すること

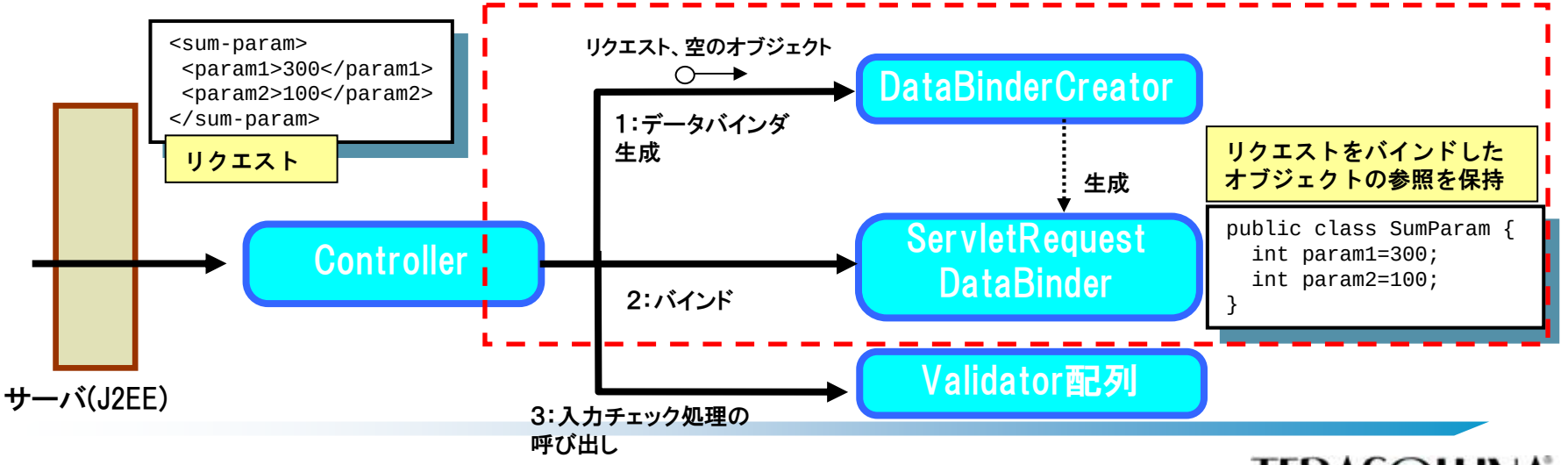
TERASOLUNA

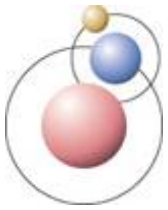
④リクエストデータ解析

リクエストデータ解析とは

◆HTTPリクエストをJavaBeanに変換する機能

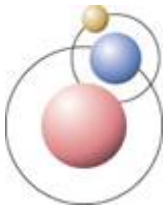
- クエリ形式のリクエストデータ解析機能
  - クエリ形式のリクエストデータをJavaBeanに変換する
- XML形式のリクエストデータ解析機能
  - XML形式のリクエストデータをJavaBeanに変換する
  - XML形式のリクエストに対し、XMLスキーマによるチェックを行う





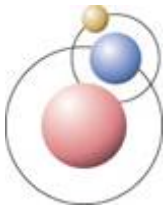
④リクエストデータ解析

- DataBinder
  - リクエストをオブジェクトにバインドするクラスが実装するSpring提供のインタフェース
  - DIコンテナ管理対象外
- ServletRequestDataBinder
  - クエリ形式のリクエストをオブジェクト(コマンド)にバインドするSpring提供のDataBinder実装クラス
  - リクエストデータがバインドされたコマンドオブジェクトの参照を保持している
  - マルチパート形式のリクエストにも対応可能(byte配列またはSpringMVCが提供するクラス型の属性にバインド可能)



④リクエストデータ解析

- XMLServletRequestDataBinder
  - XML形式のリクエストデータをJavaBean(コマンド)に変換するDataBinder実装クラス
  - リクエストデータがバインドされたコマンドオブジェクトの参照を保持している
  - 実際の変換処理は、OXMapper実装クラスに委譲される
    - 詳細は「XML-Object変換機能.pdf」を参照すること
  - XMLスキーマによる型チェックを利用して、詳細なエラーメッセージを取得することも可能



④リクエストデータ解析

ServletRequestDataBinderCreator

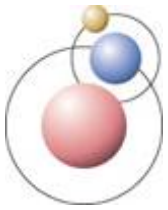
ServletRequestDataBinderを生成するインタフェース

- SpringMVCのデフォルト実装はServletRequestDataBinderクラスをハードコーディングで生成しているためDataBinderの差し替えができない
- ServletRequestDataBinderクラスのサブクラスを使用する場合はDataBinder生成クラスが必要となる。
- コントローラに実装クラスが設定される(DIコンテナ管理)
  - Bean定義ファイルの設定により、容易に実装を入れ替えることが可能

ServletRequestDataBinderCreatorの設定例

```
<!-- クエリ形式データバインダ生成クラス -->
<bean id="queryDataBinderCreator"
  class="jp.terasoluna.fw.web.rich.springmvc.bind.creator.QueryServletRequestDataBinderCreator"/>

<!-- XML形式データバインダ生成クラス -->
<bean id="xmlDataBinderCreator"
  class="jp.terasoluna.fw.web.rich.springmvc.bind.creator.XMLServletRequestDataBinderCreator">
  <property name="binder" ref="xmlDataBinder"/>
</bean>
```



④リクエストデータ解析

- QueryServletRequestDataBinderCreator
  - ServletRequestDataBinderCreator実装クラス
    - クエリ形式のリクエストに対応するServletRequestDataBinderを生成する
- XMLServletRequestDataBinderCreator
  - ServletRequestDataBinderCreator実装クラス
    - XML形式のリクエストに対応するXMLServletRequestDataBinderを生成する

TERASOLUNA

④リクエストデータ解析

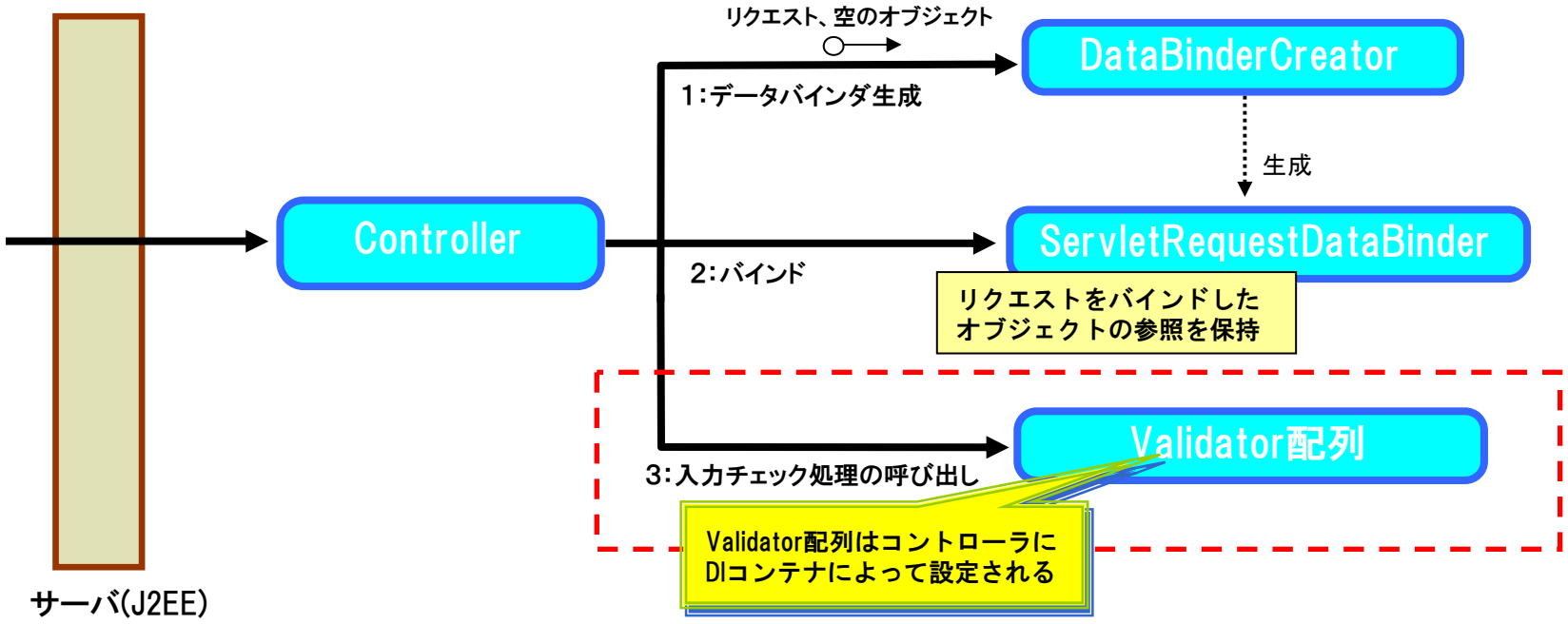
BindException

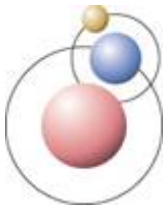
- バインド・入力チェックに失敗した場合、エラー情報を保持するクラス
  - DataBinder実装クラスがBindExceptionを保持する
  - エラーコード、エラーが発生した要素・属性、置換文字列を保持する
  - 例外のハンドリングに関しては「⑪例外処理」を参照

⑤入力チェック

入力チェック

- リクエストデータ解析機能で作成したJavaBean(コマンド)の値を検査する機能。





⑤入力チェック

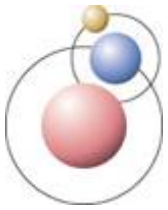
Validator

- 入力チェックを行うためのインタフェース
- コントローラにValidator実装クラスを設定すると、validateメソッドが実行される
  - Validator実装クラスは1つのコントローラに対して複数設定できる
    - Validator実装クラスが1つも設定されていない場合は何もしない
  - Bean定義ファイルにてValidator実装クラスを設定する
  - TERASOLUNAではCommons-Validatorとの連携を行うSpring-modulesを利用した実装を提供している
    - 自動生成ツールによりCommons-Validator設定ファイルの作成を支援する
    - 詳細は「入力チェック機能.ppt」を参照すること
- DBアクセスが必要な入力チェックは、サービス層で行う

⑤入力チェック

BindException

- バインド・入力チェックに失敗した場合、エラー情報を保持するクラス
  - DataBinder実装クラスがBindExceptionを保持する
  - エラーコード、エラーフィールド、置換文字列を保持する
  - 例外のハンドリングに関しては「⑩例外処理」を参照



⑥サービス実行

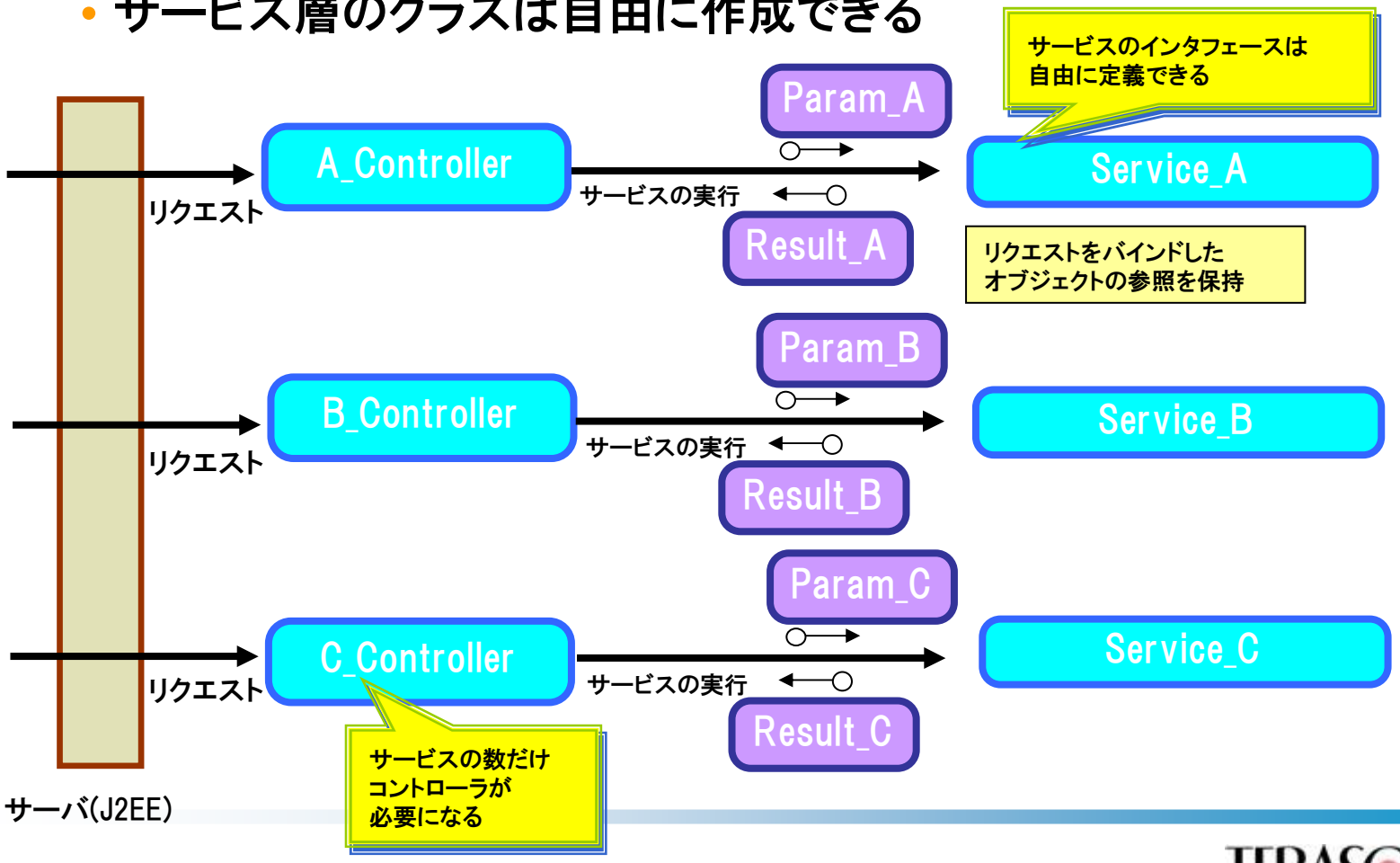
サービスの実行とは

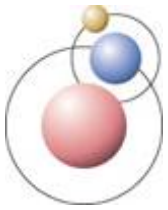
- 業務ロジック処理(サービス)をコントローラが呼び出すこと
- TERASOLUNAでは2つの実現方法を提供している
  1. 業務ロジックをPOJOで定義する
     - 自由なインタフェース設計が可能
     - 業務ロジックの数だけコントローラを作成する必要がある
  2. 業務ロジックにTERASOLUNA提供のサービス用インタフェースを実装させる
     - 業務ロジックがTERASOLUNAに依存するため、再利用性が低下する
     - コントローラを1つにすることができる
  - 方法1は自由な設計のため再利用性が向上するが、作成クラスは多くなる
  - 方法2は再利用性は低下するが、作成するクラスは少なくなる

⑥サービスの実行

1. 業務ロジックをPOJOで定義するパターン

- コントローラはTerasolunaControllerのサブクラスを作成する
- サービス層のクラスは自由に作成できる





⑥サービスの実行

TerasolunaControllerのサブクラス

- コントローラはサービス層のクラスごとにTerasolunaControllerのサブクラスを生成する必要がある
  - サービス層のクラスをコントローラにDIコンテナによって設定する
  - サービス層のクラスを実行するexecuteService()メソッドを実装する必要がある
    - 前処理・後処理を実装するための拡張点も用意している
      - » セッションからコマンドへの反映、モデルからセッションへの反映等が可能
  - サービス層に渡すJavaBean(コマンド)の型、サービス層から受け取るJavaBean(モデル)の型を定義する必要がある
    - JavaSE5.0のGenericsを利用する
- サービス層のクラスは自由に実装できる

⑥サービスの実行

TerasolunaControllerのサブクラス実装例

```
public class SumController extends TerasolunaController<SumParam, SumResult> {
    private SumService sumService;

    public void setSumService(SumService sumService) {
        this.sumService = sumService;
    }

    @override
    protected SumResult executeService(SumParam ivo) {
        return sumService.sum(ivo);
    }
}
```

手順1：型パラメータを定義する
SumParamがコマンドクラス（サービス層への引数）
SumResultがモデルクラス（サービス層からの戻り値）

手順2：呼び出したいサービス層のクラスを
DIコンテナを利用して設定する

手順3：業務ロジックを実行する

サービス実行クラス

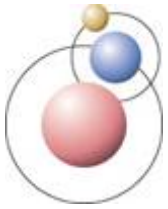
パラメータ

戻り値

サービス層のクラス1つに対して
1つのコントローラが必要

サービス層のクラス実装例

```
public class SumServiceImpl implements SumService {
    // 業務ロジック
    public SumResult sum(SumParam sumParam) throws Exception {
        SumResult result = new SumResult();
        result.setResult((sumParam.getParam1() + sumParam.getParam2()));
        return result;
    }
}
```



⑥サービスの実行

Bean定義ファイルイメージ例

```
<!- 合計算出業務(受信リクエスト:XML形式) ->
<bean name="/sumController"
      class="jp.terasoluna.sample2.web.controller.SumController"
      parent="xmlRequestController">
  <property name="sumService" ref="sumService"/>
</bean>

<!- サービス層のクラス ->
<bean id="sumService"
      class="jp.terasoluna.sample2.service.impl.SumServiceImpl">
  ……
</bean>
```

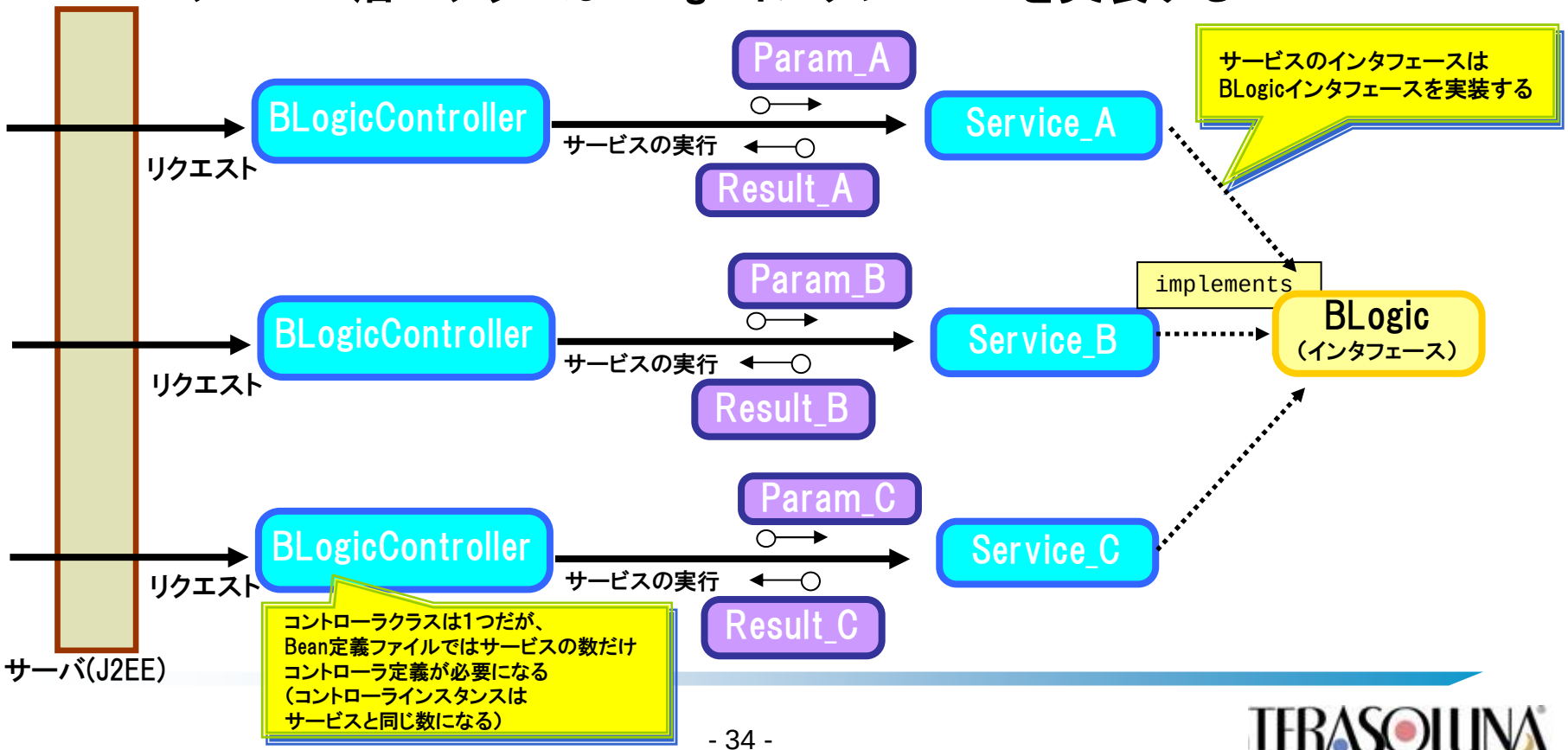
サービス層のクラスを設定

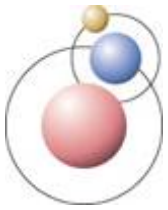
SpringのDIコンテナの機能を利用して、
Sumコントローラ(上記の定義ファイルの"/sumController")に
サービス層のクラス(上記の定義ファイルの"sumService")を
設定している

⑥サービスの実行

2. 業務ロジックにTERASOLUNAが提供するインタフェースを実装させるパターン

- コントローラはBLogicControllerを利用する
- サービス層のクラスはBLogicインタフェースを実装する

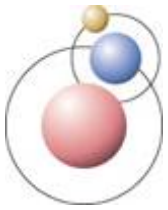




⑥サービスの実行

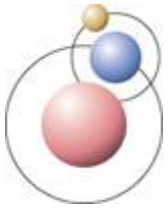
BLogicController

- BLogicインタフェースを実装したサービス層のクラスを呼び出すためのコントローラ
  - TerasolunaControllerのサブクラスとして定義されている
  - 本クラスは実装済みであり、サブクラスの作成は不要になる(コントローラは本クラスのみ)
  - 以下の制約に従う必要がある
    - サービス層のクラスは、BLogicインタフェースを実装しなければならない
    - Bean定義ファイルに、サービスごとのコントローラの設定を行う必要がある(10サービスある場合、10個の設定が必要)
    - Bean定義ファイルに、起動するサービス層のクラス名を設定する必要がある



⑥サービスの実行

- BLogic
  - BLogicControllerを使用するときに、サービス層のクラスが実装すべきインタフェース
    - サービス実装メソッド“executeメソッド”を定義する
  - 引数と戻り値は、実装クラスごとに自由に設定できる
    - 型パラメータ<P,R>に以下を定義し、executeメソッド内の処理をタイプセーフにする
      - P・・・サービスの引数となるクラス(コマンドクラス)
      - R・・・サービスの戻り値となるクラス



⑥サービスの実行

Bean定義ファイルイメージ例

```
<bean name="/maxController" parent="xmlRequestBLogicExecuteController">
  <property name="blogic" ref="maxBLogic" />
</bean>
<bean id="maxBLogic"
      class="jp.terasoluna.sample2.service.blogic.MaxBLogic">
  ……
</bean>
```

サービス層のクラスを設定

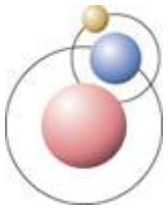
BLogic実装クラスイメージ例

コマンドクラス

```
public class MaxBLogic implements BLogic<MaxParam, MaxResult> {
    public MaxResult execute(MaxParam param) {
        MaxResult result = new MaxResult();
        int[] values = param.getValue();
        int max = values[0];
        for (int i=1; i<values.length; i++ ) {
            if( values[i] > max ) {
                max = values[i];
            }
        }
        result.setMaxValue(max);
        return result;
    }
}
```

サービスの戻り値となるクラス

BLogicのexecuteメソッドが呼ばれる

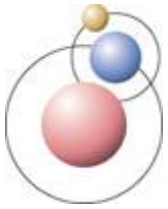


⑦トランザクション管理

トランザクション管理

- コミット・ロールバックはフレームワークが行う
  - AOPを利用した宣言的トランザクションを行うため、開発者からトランザクションを織り込む処理を隠蔽する
  - サービス開始時にトランザクションが開始され、終了時にコミットされる。例外時はロールバックを行う
- サービス層をトランザクション境界とし、1サービス・1トランザクションとする

TERASOLUNA



⑦トランザクション管理

トランザクション管理イメージ例(Bean定義ファイル) 1/2

```
<!-- 単一のJDBCデータソース向けのトランザクションマネージャ -->
<bean id="transactionManager" class="org .…(中略)….DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
<!-- トランザクション定義情報 -->
<bean id="attrSource"
      class="org.… (中略) ….NameMatchTransactionAttributeSource">
  <property name="properties">
     <props>
       <prop key="insert*">PROPAGATION_REQUIRED,-java.lang.Exception</prop>
       <prop key="update*">PROPAGATION_REQUIRED,-java.lang.Exception</prop>
       <prop key="*">PROPAGATION_REQUIRED,readOnly,-java.lang.Exception</prop>
     </props>
  </property>
</bean>
<!-- トランザクション処理(インタセプタの定義) -->
<bean id="transactionInterceptor"
      class="org.…(中略)….TransactionInterceptor">
    <property name="transactionManager“ ref="transactionManager"/>
    <property name="transactionAttributeSource" ref="attrSource"/>
    </property>
</bean>
```

Springで提供されているクラス。
どのメソッドにどのようなトランザクション管理を行うのかを指定する。

Springのデフォルト設定では、検査例外がスローされてもロールバックされない。
例のように、-[例外クラス名]指定するとロールバックされる。

Springで提供されているクラス。
宣言的トランザクションを行うためのインタセプタ。

TERASOLUNA

⑦トランザクション管理

トランザクション管理イメージ例(Bean定義ファイル) 2/2

```
<!-- オートプロキシ設定(Bean定義ファイルのID -->
<bean id="autoProxy"
   class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="interceptorNames">
    <list>
      <idref local="transactionInterceptor"/>
    </list>
  </property>
  <property name="beanNames">
    <list>
      <value>*BLogic</value>
      <value>*Service</value>
    </list>
  </property>
</bean>
```

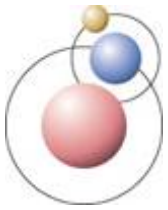
Springで提供されているファクトリクラス。
指定したBeanに対してインタセプタを適用する。

ここに適用したインタセプタが利用される

ここに指定したBean定義ファイルのID
に対してインタセプタが適用される

```
<!-- 業務オブジェクトの定義。 -->
<bean id="sumService"
  class="jp.terasoluna.sample2.service.impl.SumServiceImpl"/>
</bean>
```

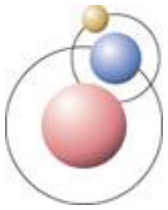
業務ロジック実装者は赤点線と
同様の設定を行う



⑧データベースアクセス機能

データベースアクセス機能

- データアクセスオブジェクトのインタフェースを提供する
  - QueryDAO・・・参照系のデータベースアクセスを行うDAOインタフェース
  - UpdateDAO・・・更新系のデータベースアクセスを行うDAOインタフェース
  - StoredProcedureDAO・・・ストアドプロシージャを実行するDAOインタフェース
- RDBMS製品やO/Rマッピングツールに依存する処理はDAOの実装クラス内に隠蔽する
- DAOのデフォルト実装としてiBatisを用いた実装を提供する
- データベースコネクションのトランザクション管理はサービス層でSpringAOPによる宣言的トランザクション管理を用いるため、開発者が意識する必要はない



⑧データベースアクセス機能

◆サービス層でのデータアクセスの実装

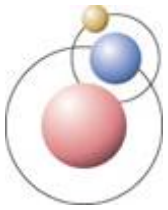
- DAOインスタンスはDIコンテナからビジネスロジックにDIする

```
public class QueryBLogic {
    private QueryDAO dao = null; // インタフェースの型でDAOを宣言
    ・・・daoのgetter/setterはここでは省略
    public Object dbAccess() {
        SelectUserArgBean bean = new SelectUserArgBean(); //プレースホルダ置換用のPOJO
        ・・・省略
        SelectUserResult result =
            dao.queryForObject("select.user", bean, SelectUserResult.class);
        //第一引数は実行する設定のID(DAOの実装に依存する)
        //第二引数はSQLなどにプレースホルダがある場合の置換文字列を格納したPOJO。不要であればnull。
        //第三引数は戻り値のクラス
        ・・・省略
    }
}
```

ビジネスロジック

```
<bean id="addBLogic" scope="prototype"
  class="jp.terasoluna.sandbox.AddBLogic" >
  <property name="dao" ref="queryDAO"/>
</bean>
<bean id="queryDAO" class="jp.terasoluna.xxx.MyQueryDAOImpl"/>
```

Bean定義ファイル



⑧データベースアクセス機能

◆iBatisを用いたDAOのデフォルト実装

- iBatisの機能を利用した以下のクラスを提供する
  - QueryDAOiBatisImpl
  - UpdateDAOiBatisImpl
  - StoredProcedureDAOiBaitsImpl
- 設定
  - Bean定義ファイル・・・データソース、sqlMapClientの設定を行う

Bean定義ファイル

```
<!-- データソースの設定 -->
<bean id="TerasolunaDataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"  destroy-method="close">
  <property name="driverClassName" value="jdbc.driverClassName=oracle.jdbc.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@hostname:1521:sid "/>
  <property name="username" value="oracle"/>
  <property name="password" value="password"/>
</bean>
<!-- iBATIS定義 -->
<bean id="sqlMapClient"
      class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="configLocation" value="WEB-INF/sqlMapConfig.xml"/>
  <property name="dataSource“ ref="TerasolunaDataSource"/>
</bean>
```

sqlMapClient

iBatis設定ファイルへのパス

データソースの設定

⑧データベースアクセス機能

- 設定
  - Bean定義ファイル(続き)・・・DAOの設定を行う

Bean定義ファイル

```
<!-- DAO定義 -->
<bean id="queryDAO"
        class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
    <property name="sqlMapClient"><ref local="sqlMapClient"/></property>
</bean>
```

ここで定義したDAOをビジネスロジックにDIする

DAOにsqlMapClientをDIする

(UpdateDAOiBatisImpl、StoredProcedureDAOiBatisImplの場合も同様)

  - sqlMapConfig.xmlの設定
    - » 特別な定義は不要。sqlMap.xmlへのパスだけを設定する。
    - » Bean定義ファイルのsqlMapClientの設定に記述したパスと一致させる

sqlMapClient.xml

```
<sqlMapConfig>
    <sqlMap resource="sqlMap.xml"/>
</sqlMapConfig>
```

データベースアクセスごとの設定はsqlMap.xmlに記述する

  - sqlMap.xmlの設定
    - » 実行するデータアクセス毎の設定を記述する
    - » iBatisの仕様どおりに記述する

sqlMap.xml

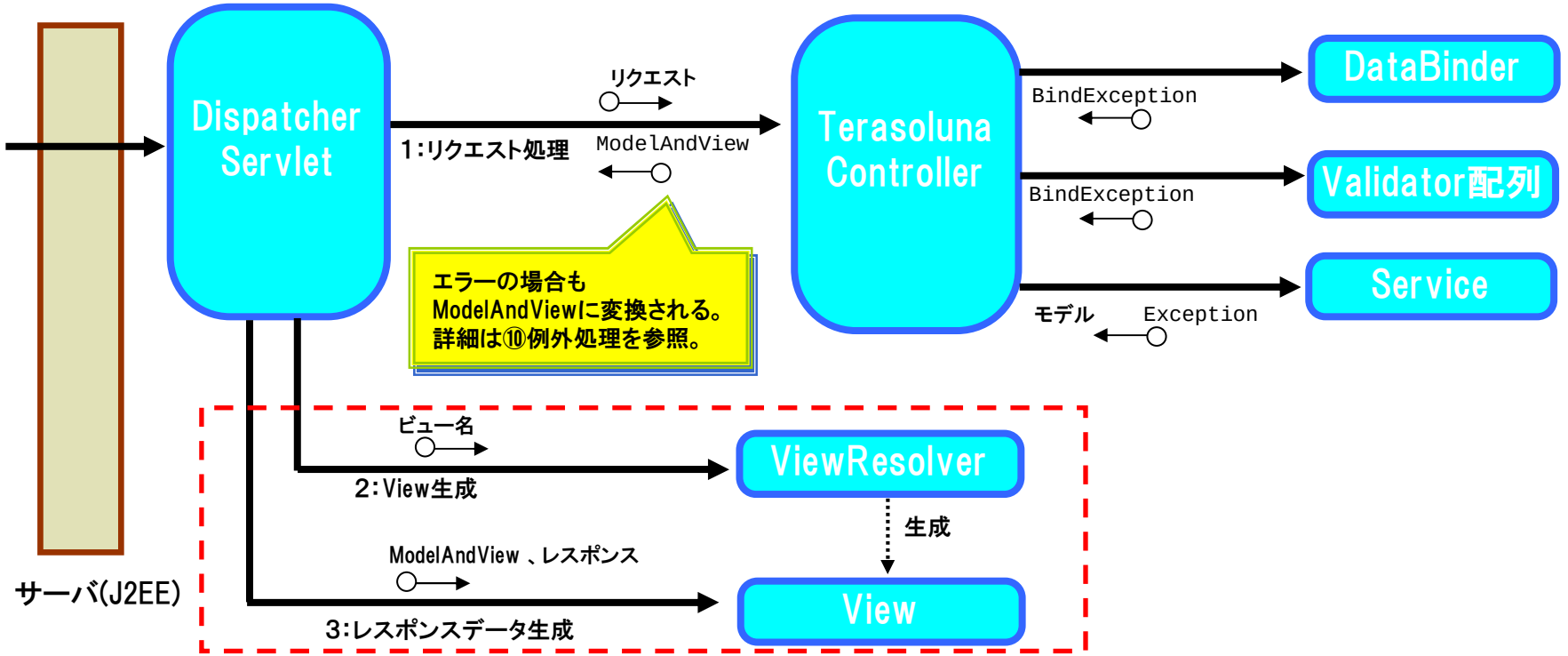
```
<select id="getUser"
        resultClass="jp.terasoluna.xxx.UserBean">
  SELECT ID, NAME, AGE FROM USERLIST WHERE ID = #VALUE#
</select>
```

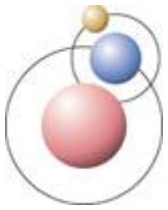
データアクセスの結果を保持するクラス名を設定する

⑨レスポンスデータ生成

レスポンスデータ生成

- サービス実行結果(モデル)をHTTPレスポンスに変換する機能

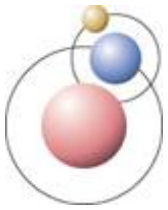




⑨レスポンスデータ生成

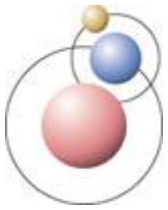
ModelAndView

- モデル(レスポンスに設定する値)とビュー(レスポンス生成機能)を保持するクラス
  - ビュー名、またはビューインスタンスを保持する
    - ViewResolverは、ここで設定したビュー名を使用する
    - ビュー名がNullの場合、ビューインスタンスを使用する
  - レスポンスデータ生成時に使用するデータを保持する
    - Viewはここで設定したデータをもとにレスポンスデータを生成する
    - データはMap形式で保持する
      - » サービスが正常に行われた場合、サービス実行結果がモデルに設定される
      - » 例外が発生した場合、例外オブジェクトがモデルに設定される(エラーコードを設定しておくことも可能)



⑨レスポンスデータ生成

- View
  - HTTPレスポンスデータを生成するクラスが実装すべきインタフェース
  - リクエスト、サービス実行結果（モデル）をもとにHTTPレスポンスデータを作成することができる
  - ビューの選択は、リクエスト実行時にViewResolver実装クラスに委ねられる。



⑨レスポンスデータ生成

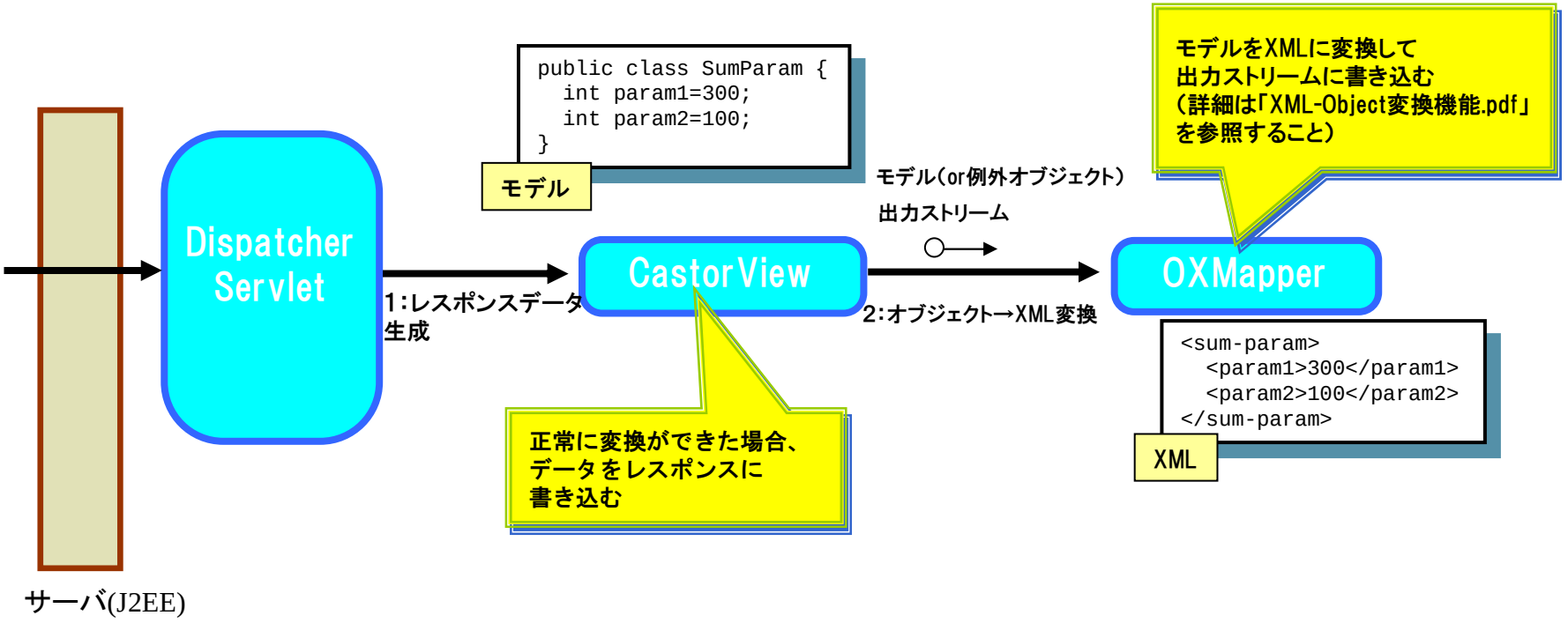
CastorView

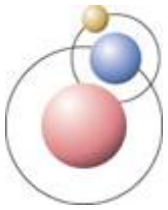
- オブジェクト-XML変換ツール「Castor」を利用して、モデルをXMLデータに変換した結果をレスポンスに設定するクラス
  - 実際の変換処理はOXMapper実装クラスが行う。デフォルトでCastorOXMapperImplクラスが設定されている
  - Castorマッピング定義ファイルによりオブジェクトの属性とXML要素の関連を記述できる(省略可)
    - 変換対象のオブジェクトと同一のパッケージに同一の名前で拡張子".xml"のファイルを使用する
      - 例)sample.SumクラスのCastorマッピングファイル名は、Sum.xmlとなる(ただし、sampleパッケージに配置すること)
    - 1つのマッピング定義ファイルでオブジェクト-XML相互の変換が可能
  - 複雑な電文の変換ができない
  - 詳細は「XML-Object変換機能.pdf」を参照すること

TERASOLUNA

⑨レスポンスデータ生成

CastorViewの利用イメージ





⑨レスポンスデータ生成

VelocityView

- Springが提供するテンプレートエンジン「Velocity」を利用して、オブジェクトをXMLデータに変換した結果をレスポンスに設定するクラス
  - 高速でかつ柔軟に電文が生成可能
  - 電文の種類ごとにテンプレートが必要
- 例外発生時にはVelocityを使用してレスポンスデータを生成している
- 高速処理が必要な場合、CastorViewを補完する用途で利用可能

⑨レスポンスデータ生成

Velocityテンプレートファイルのイメージ例（exception.vm）

```
<?xml version="1.0" encoding="utf-8" ?>
<DATA>
  <WtxResult>
    <WtxErrNumber>${errorCode}</WtxErrNumber>
    <WtxErrDesc>
      ${exception.class}で${rc.getMessage("errors.${errorCode}")}
    </WtxErrDesc>
  </WtxResult>
</DATA>
```

Velocityテンプレート中に
${　}を指定すると変数が指定できる。
VelocityViewResolverで設定した属性、
ModelAndViewでMapに設定された値を
参照できる。

イメージ例にある変数rcはSpringが提供するRequestContextクラス。
（Bean定義ファイルで設定する）
RequestContextクラスのgetMessageメソッドはBean定義中のリソースバンドルからメッセージを取得できる。

この例では“errors.error01”というキーに
対応するメッセージを探している。
下のイメージ例では
「errors.error01=例外が発生しました」
というメッセージがあった場合の例。

ModelAndView

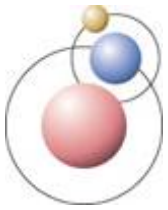
ビュー名：exception

errorCode
（エラーコード）=“error01”

Exception

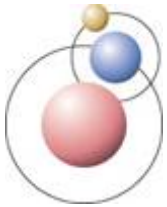
出力される電文のイメージ例

```
<?xml version="1.0" encoding="utf-8" ?>
<DATA>
  <WtxResult>
    <WtxErrNumber>error01</WtxErrNumber>
    <WtxErrDesc>
      Exceptionクラスで例外が発生しました
    </WtxErrDesc>
  </WtxResult>
</DATA>
```



⑨レスポンスデータ生成

- AbstractFileDownloadView
  - バイナリファイル(入力ストリーム)をレスポンスに設定するクラス
    - 抽象クラスであるため、サブクラスを作成する必要がある
    - サブクラスにて設定する項目
      - ダウンロードするファイル情報(入力ストリーム)
      - レスポンスヘッダ
  - ViewResolverはResourceBundleViewResolverを使用する
    - プロパティファイルに「*ビュー名＝クラス名*」で定義する
  - コントローラでビュー名を設定する必要がある
    - プロパティファイルに記述したビュー名を設定する



⑨レスポンスデータ生成

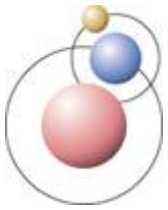
AbstractFileDownloadViewサブクラスのイメージ例

```
public class SampleFileDownloadView extends AbstractFileDownloadView {
    @Override
    protected InputStream getInputStream(
        Map model, HttpServletRequest request) throws IOException {
        ServletContext context = request.getSession().getServletContext();
        return context.getResource("/image/terasoluna.jpg").openStream();
    }

    @Override
    protected void addResponseHeader(
        Map model, HttpServletRequest request, HttpServletResponse response) {
        response.addHeader("Content-Disposition",
            "attachments;filename=" + "terasoluna.jpg");
    }
}
```

ファイルデータを設定するメソッド

レスポンスヘッダを追加するメソッド



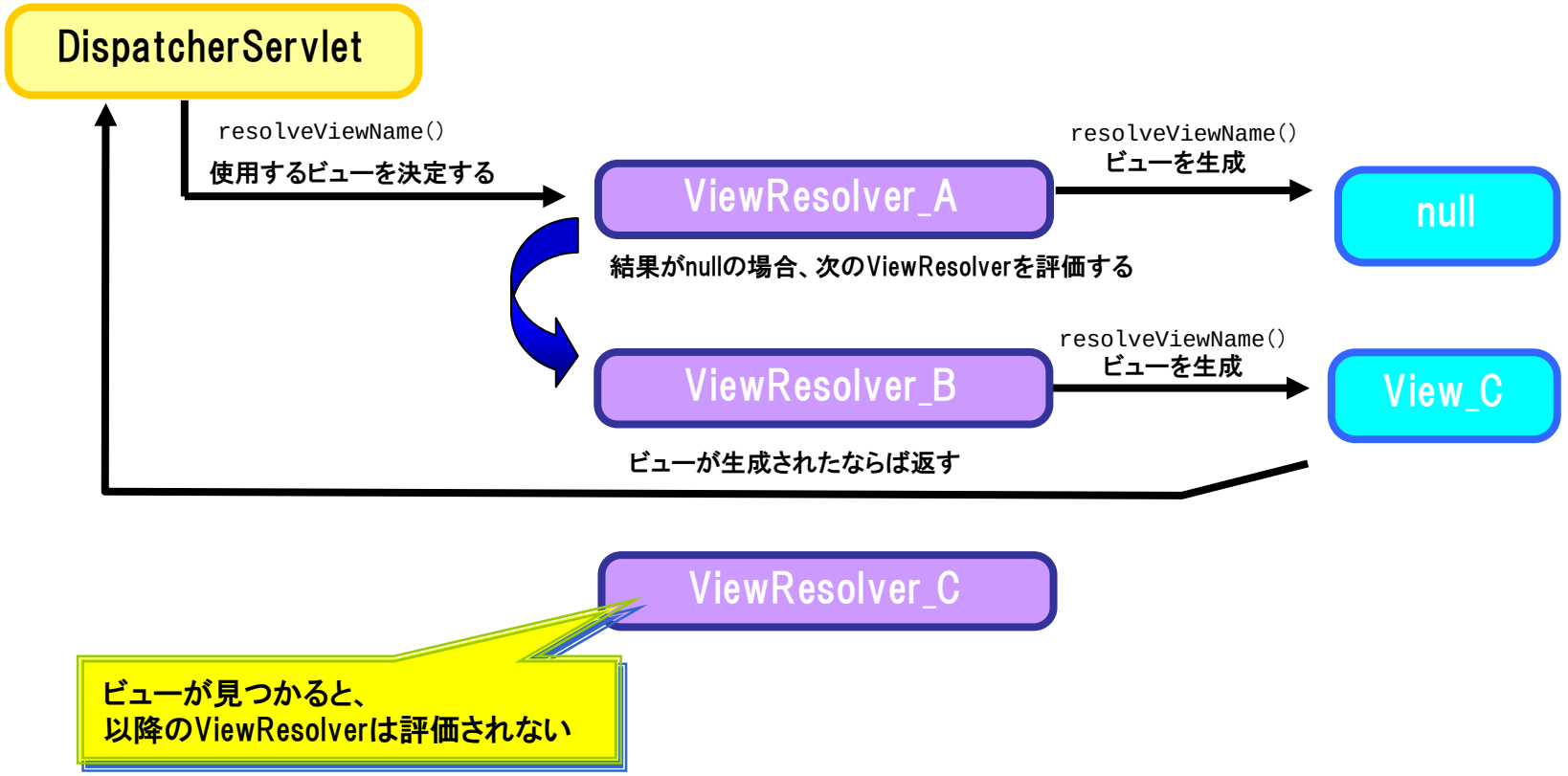
⑨レスポンスデータ生成

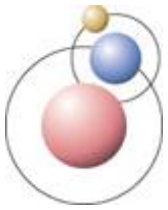
ViewResolverインタフェース

- ビュー名に対応するビューを返すクラスが実装すべきインタフェース
  - ビュー名はコントローラで設定する
  - View resolveViewName(String viewName, Locale locale)メソッドを実装する
  - 複数のViewResolverを使用するため、ビューの生成を行わないときはresolveViewNameメソッドでnullを返す
- 複数のViewResolverを併用することが可能
  - ViewResolverの使用順を指定するには、Orderedインタフェースを実装して、order属性に値を設定する必要がある
  - Orderedインタフェースを実装していなかったり、order属性が指定されていない場合、最後に使用される

⑨レスポンスデータ生成

- ViewResolverの利用イメージ

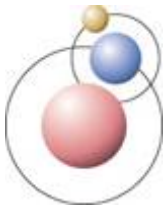




⑨レスポンスデータ生成

CastorViewResolver

- Castorビューを生成するためのViewResolver実装クラス
- oxmapper属性でOXMapperインタフェース実装クラスを指定する必要がある
  - デフォルトではCastorOXMapperImplクラスが設定してある
- ビュー名が空文字、もしくはnullのとき、ビューが生成される
  - デフォルトのビュー名が空文字になっているため、特に設定がない場合、Castorビューが使用される
  - Castorビューを使用しない場合、ビュー名に任意の文字列を入力しておくこと
- Bean定義ファイルにてorder属性を設定することが可能



⑨レスポンスデータ生成

- VelocityViewResolverEx
  - Velocityビューを使用するためのViewResolver実装クラス
  - テンプレートファイルのロケーション設定が必要
    - デフォルトでは「/WEB-INF/velocity/*ビュー名*.vm」がテンプレートファイルとして使用される
  - テンプレートファイル名とビュー名を一致させる
    - テンプレートファイルの名前を「*リクエスト名*.vm」にする
    - コントローラのuseVelocityView属性をtrueにする
      - この設定で「/*リクエスト名*」がビュー名に設定される
  - Orderedインタフェースを実装していないため、常に最後に使用すること

⑨レスポンスデータ生成

ViewResolver　Bean定義ファイル 設定例

```
<!--  Castor用View Resolver  -->
<bean id="castorViewResolver“
      class="jp.terasoluna.fw.web.rich.springmvc.servlet.view.castor.CastorViewResolver">
  <property name="cache“ value=“true”/>
  <property name="requestContextAttribute“ value=“rc”/>
  <property name="contentType“ value=“text/xml;charset=UTF-8”/>
  <property name="order“ value=“2”</property>
  <property name="oxmapper“ ref="oxmapper"/>
</bean>

<!-- オブジェクト・XMLマッピングクラス -->
<bean id="oxmapper" class="jp.terasoluna.fw.oxm.castor.CastorOXMapperImpl"></bean>

<!-- Velocity設定 -->
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name=“resourceLoaderPath” value=“/WEB-INF/velocity/”/>
</bean>

<!--  Velocity用View Resolver  -->
<bean id="velocityViewResolver"
      class="jp.terasoluna.fw.web.rich.springmvc.servlet.view.velocity.VelocityViewResolverEx">
  <property name="cache“ value=“true”/>
  <property name="requestContextAttribute“ value=“rc”/>
  <property name="prefix“ value=“”/>
  <property name="suffix“ value=“.vm”/>
  <property name="exposeSpringMacroHelpers“ value=“true”/>
  <property name="encoding“ value=“UTF-8”/>
  <property name="contentType“ value=“text/xml;charset=UTF-8”/>
</bean>
```

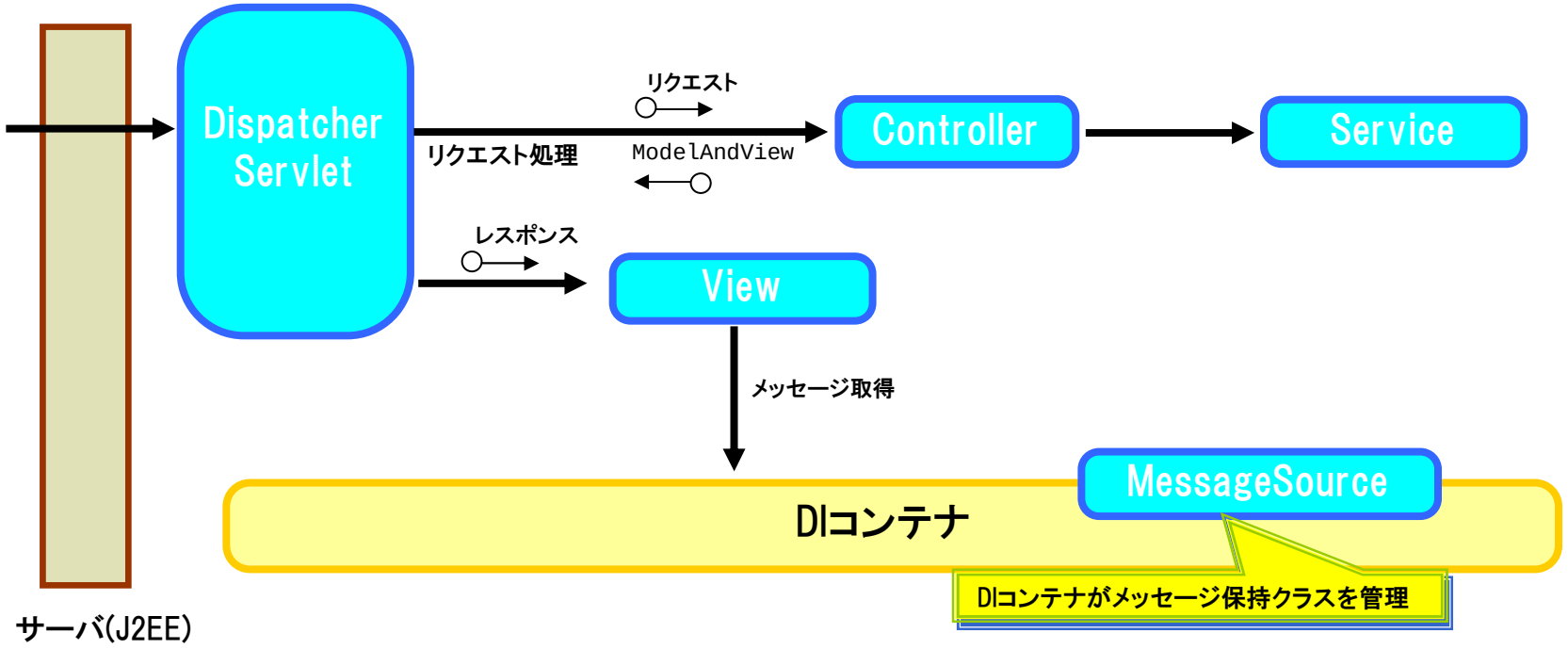
オブジェクト-XML変換に使用する

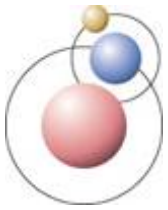
Velocityテンプレートファイルの場所

⑩メッセージ管理

メッセージ管理とは

- フレームワーク内・サービス実行時に使用されるメッセージ（エラーメッセージ等）を保持・取得する機能





⑩メッセージ管理

MessageSource

- メッセージを保持するクラスが実装すべきインタフェース
- Springでは「messageSource」というIDのBean定義がメッセージクラスとして認識される

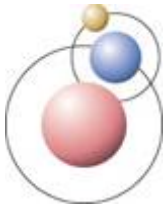
ResourceBundleMessageSource

- リソースバンドルを利用したMessageSource実装クラス
  - プロパティファイルからメッセージを取得する(複数ファイル可)
  - 国際化対応可能
  - Webアプリケーション起動中のメッセージリロードはできない

ResourceBundleMessageSource　Bean定義ファイル 設定例

```
<!-- メッセージの設定 -->
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames" value="applicationResources,errors"/>
</bean>
```

プロパティファイル名を記述する
プロパティファイルはクラスパス上になければならない
設定例のようにカンマ区切りで複数のファイルをロードできる



⑩メッセージ管理

DataSourceMessageSource

◆データベースを利用したMessageSource実装クラス

- データベースからメッセージを取得する
  - メッセージ取得クラス(DBMessageResourceDAOインタフェース実装クラス)を設定(DI)する必要がある。
    - » デフォルト実装としてDBMessageResourceDAOImplを提供している
- 国際化対応可能
- Webアプリケーション起動中のメッセージリロードが可能

DataSourceMessageSource　Bean定義ファイル 設定例

```
<!-- メッセージの設定 ->
<bean id="messageSource"
      class="jp.terasoluna.fw.message.DataSourceMessageSource">
  <property name = "DBMessageResourceDAO“ ref="DBMessageResourceDAO"/>
</bean>

<bean id="DBMessageResourceDAO"
      class="jp.terasoluna.fw.message.DBMessageResourceDAOImpl">
  <property name = "dataSource“ ref = "dataSource"/>
</bean>
```

DBからメッセージを
読み込むためのDAO

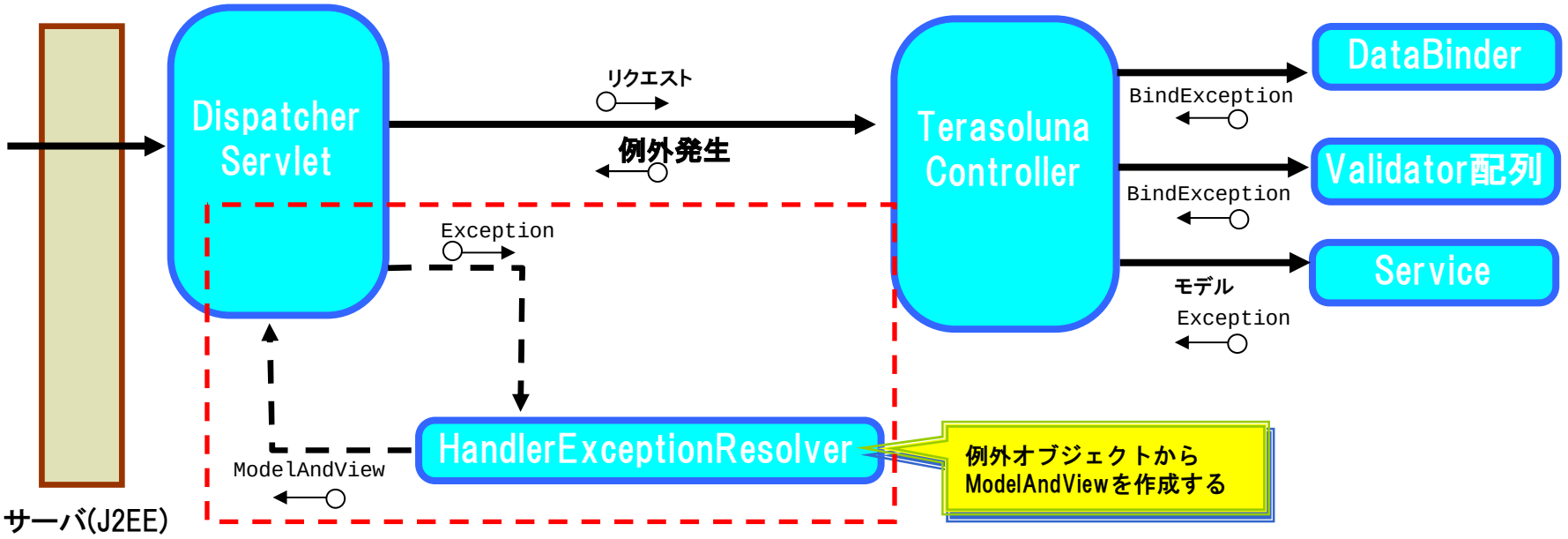
デフォルトのSQLは
SELECT CODE,MESSAGE FROM MESSAGES
テーブル名 = MESSAGES
メッセージコードを格納するカラム名 = CODE
メッセージ本文を格納するカラム名 = MESSAGE

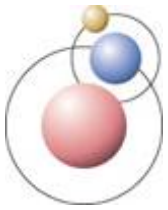
⑪例外処理

例外処理の一元化

Springフレームワークの例外ハンドリング機能を利用する

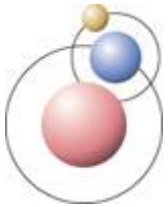
- HandlerExceptionResolverインタフェース
  - コントローラ内で発生した例外をハンドリングする
  - 例外オブジェクトからビューを作成する役割を持つ





⑪例外処理

- SimpleMappingExceptionResolverEx
  - 例外クラス名とビュー名の対応付けを行うHandlerExceptionResolver実装クラス
    - Bean定義ファイルに例外クラス名とビュー名の対応付けを記述する
    - 使用するビューはViewResolverによって決定される
      - デフォルトはVelocityビューが使用される
  - 例外クラス名と対応付けられたビュー名と、スローされた例外を保持したクラス(ModelAndViewクラス)を返す
  - 例外の型ごとにエラーコードの設定が可能
  - 例外の設定に順序性をもたせることが可能
    - Bean定義ファイルの上から順番に、発生した例外の型が一致するか評価を行う



⑪例外処理

SimpleMappingExceptionResolverEx　Bean定義ファイル 設定例

```
<bean id="handlerExceptionResolver"
      class="jp.terasoluna.(省略).SimpleMappingExceptionResolverEx">
  <property name="linkedExceptionMappings">
    <map>
      <entry key="jp.terasoluna.(省略).UnknownCommandException">
        <value>exception,8004C003</value>
      </entry>
      <entry key="org.springframework.validation.BindException">
        <value>bindException</value>
      </entry>
      <entry key="jp.terasoluna.(省略).SystemException">
        <value>systemException</value>
      </entry>
      <entry key="jp.terasoluna.(省略).ServiceException">
        <value>serviceException</value>
      </entry>
      <entry key="java.lang.Exception">
        <value>exception,8004C999</value>
      </entry>
    </map>
  </property>
</bean>
```

上から順に評価される
処理中の例外が合致する
まで評価を続ける

```
<entry key="発生する例外クラス名">
  <value>ビュー名[,エラーコード]</value>
</entry>
```

ここに発生する例外とViewの対応を羅列していく。
自クラスまたは親クラスの例外型が対応した
設定を実行する。
たとえば、全ての例外は、java.lang.Exceptionを
継承しているので、明示的に定義されていない
例外が発生した場合でも、
必ず最後のjava.lang.Exceptionの設定で
例外ハンドリングが行われる。