Web アプリケーション フレームワーク コンポーネント作成ガイド

Sun Java™ Studio Enterprise 7 2004Q4

Sun Microsystems, Inc.
www.sun.com

Part No. 819-1284-10
2004 年 12 月, Revision A

原典: *Web Application Framework Component Author's Guide*
Part No: 819-0724-10
Revision A

目次

はじめに

このガイドでは、Web アプリケーションフレームワークのコンポーネントアーキテクチャと、コンポーネント作成者が新しいコンポーネントをデザイン、作成、配布するときのプロセスを説明しています。今後 Web アプリケーションフレームワークコンポーネントを作成しようとする方を対象読者とし、Web アプリケーションフレームワークアーキテクチャや Sun™ Java ™ Studio Enterprise 7 2004Q4 開発環境 (以降、IDE と呼ぶ) に関する知識がすでにあることを前提にしています。

お読みになる前に

このマニュアルを読み始める前に、サーブレットや JavaServlet ページ™ (JSP™ ページ) などの既存の J2EE Web テクノロジを利用した Web アプリケーションの構築で用いられている概念を理解しておくことを推奨します。また、この章の後半にある Web アプリケーションフレームワークの関連マニュアルを読んで、Web アプリケーションフレームワークアーキテクチャについての知識を習得している必要があります。

詳しい情報は、以下のリソースから得ることができます。

- Java 2 Platform, Enterprise Edition Specification
  `http://java.sun.com/j2ee/download.html#platformspec`
- J2EE Tutorial
  `http://java.sun.com/j2ee/tutorial`
- Java Servlet Specification バージョン 2.3
  `http://java.sun.com/products/servlet/download.html#specs`
- JavaServer Pages Specification バージョン 1.2
  `http://java.sun.com/products/jsp/download.html#specs`

注 – Sun では、本マニュアルに掲載されている第三者の Web サイトのご利用に関しましては責任はなく、保証するものでもありません。また、これらのサイトあるいはリソースに関する、あるいはこれらのサイト、リソースから利用可能であるコンテンツ、広告、製品、あるいは資料に関しても一切の責任を負いません。Sun は、これらのサイトあるいはリソースに関する、あるいはこれらのサイトから利用可能であるコンテンツ、製品、サービスの利用あるいはそれらのものを信頼することによって、あるいはそれに関連して発生するいかなる損害、損失、申し立てに対する一切の責任を負いません。

内容の紹介

第 1 章「概要とコンポーネントアーキテクチャ」では、コンポーネントベースの開発 (CBD) と Web アプリケーションフレームワークコンポーネントライブラリ、コンポーネントクラス、ComponentInfo クラスの概要を説明しています。

第 2 章「コンポーネントの開発」では、Web アプリケーションフレームワークコンポーネントを作成、配布、使用するための基本的な手順を説明しています。

第 3 章「ビューコンポーネントの開発」では、ビューコンポーネントを開発するための基本的な手順を説明しています。

第 4 章「モデルコンポーネントの開発」では、モデルコンポーネントを開発するための基本的な手順を説明しています。

第 5 章「コマンドコンポーネントの開発」では、コマンドコンポーネントを開発するための基本的な手順を説明しています。

第 6 章「ConfigurableBeans (不可視コンポーネント)」では、IDE における ConfigurableBean の使われ方やその役割、Web アプリケーションフレームワークと ConfigurableBean 型の関係を概説しています。

第 7 章「拡張不可モデルとコマンド、コンテナビューコンポーネントの開発と配布」では、拡張不可モデルやコマンド、コンテナビューコンポーネントを開発、配布する手順を概説しています。

第 8 章「デザインアクション」では、デザインアクションを持つ拡張可能コンポーネントの開発について説明するとともに、`ComponentInfo` でデザインアクションを提供する方法を示しています。

付録 A「コンポーネントライブラリの構造」は、コンポーネントライブラリとその構造の概要を説明するとともに、コンポーネントマニフェストを詳述しています。また、コンポーネントタグライブラリ (TLD) ファイルの自動アンパックと「追加ファイル」の自動アンパックについても説明しています。

書体と記号について

書体または記号*	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例。	.login ファイルを編集します。 ls -a を実行します。 % You have mail.
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して表します。	% **su** Password:
AaBbCc123 または**ゴシック**	コマンド行の可変部分。実際の名前や値と置き換えてください。	rm *filename* と入力します。 rm **ファイル名** と入力します。
『 』	参照する書名を示します。	『Solaris ユーザーマニュアル』
「 」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照。 この操作ができるのは「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅をこえる場合に、継続を示します。	% **grep '^#define \ XV_VERSION_STRING'**

* 使用しているブラウザにより、これら設定と異なって表示される場合があります。

関連マニュアル

Java Studio Enterprise のマニュアルとしては、Acrobat Reader (PDF) 形式のマニュアル、チュートリアルと、HTML 形式のリリースノート、オンラインヘルプ、チュートリアルが提供されています。

オンラインで入手可能なマニュアル

ここで紹介しているマニュアルは、docs.sun.com℠ Web サイトおよび Sun Java Studio Enterprise Developers Source ポータルサイト (http://developers.sun.com/jsenterprise) のドキュメントリンクから入手できます。

docs.sun.com Web サイト (http://docs.sun.com) では、インターネットで Sun のマニュアルを参照、印刷、購入することができます。

- 『Sun Java Studio Enterprise 7 2004Q4 リリースノート』- Part No. 819-1302-10

  最新のリリースの変更点や技術的な注意事項を説明しています。

- 『Sun Java Studio Enterprise 7 インストールガイド』(PDF 形式) - Part No. 819-1300-10

  サポートしている各プラットフォームへの Sun Java Studio Enterprise 7 統合開発環境 (IDE) のインストール方法を説明しています。システム要件やアップグレード方法、サーバー情報、コマンド行スイッチ、インストールされるサブディレクトリ、データベースの統合、アップデートセンターの使用方法などの関連情報も記載されています。

- 『J2EE アプリケーションのプログラミング』- Part No. 819-1298-10

  EJB モジュールや Web モジュールを J2EE にアセンブルする方法を説明しています。また、J2EE アプリケーションの配備や実行についても説明しています。

- Web アプリケーションフレームワークのマニュアル (PDF 形式)
  - 『Web アプリケーションフレームワーク コンポーネント作成ガイド』 - Part No. 819-1284-10

    Web アプリケーションフレームワークのコンポーネントアーキテクチャと新しいコンポーネントの設計、作成、配布工程を説明しています。

  - 『Web アプリケーションフレームワーク コンポーネントリファレンスガイド』 - Part No. 819-1286-10

    Web アプリケーションフレームワークライブラリに提供されているコンポーネントを説明しています。

  - 『Web アプリケーションフレームワーク 概要』- Part No. 819-1288-10

    Web アプリケーションフレームワークとその位置づけ、仕組み、他のアプリケーションフレームワークと異なる点を説明しています。

  - 『Web アプリケーションフレームワーク チュートリアル』 - Part No. 819-1290-10

    Web アプリケーションフレームワークを使用して Web アプリケーションを構築する際の仕組みとその手法を紹介しています。

- 『Web アプリケーションフレームワーク 開発ガイド』- Part No. 819-1292-10

  Web アプリケーションフレームワークを使用し、開発するアプリケーションの構成要素として使用可能なアプリケーションコンポーネントの作成および使用の手順と、そのアプリケーションを大部分の J2EE コンテナに配備する方法を説明しています。

- 『Web アプリケーションフレームワーク IDE ガイド』- Part No. 819-1294-10

  Sun Java Studio Enterprise 7 2004Q4 IDE の各部の概要、および Web アプリケーションフレームワークアプリケーションを開発するためのビジュアルツールの使用方法を重点的に説明しています。

- 『Web アプリケーションフレームワーク タグライブラリリファレンス』- Part No. 819-1296-10

  Web アプリケーションフレームワークのタグライブラリを簡単に紹介し、タグライブラリに提供されているタグに対する包括的な参照を示しています。

チュートリアル

Sun Java Studio Enterprise 7 には、IDE の機能を理解する手助けとなるチュートリアルがいくつか用意されています。これらのチュートリアルにある技術、およびコード例は、そのまま、または編集を加えて、実際のアプリケーションの開発に利用することができます。すべてのチュートリアルで、Sun Java System Application Server への配備例が紹介されています。

チュートリアルは、すべて Developers Source ポータルのリンク「Tutorials & Code Camps」から利用可能です。IDE で「ヘルプ」>「コードサンプルとチュートリアル」>「概要」を選択すると、このサイトにアクセスできます。

- 「クイックスタートガイド」は、Sun Java Studio IDE の紹介をしています。チュートリアルは、Sun Java Studio を初めてご使用になる方や、特定の機能について早く知りたい場合は、このガイドから始めてください。これらのチュートリアルは、単純な Web アプリケーションや J2EE アプリケーションの開発方法、Web サービスの生成方法を説明しています。また、UML モデリング、リファクタリングの導入方法についても説明しています。ガイドを終えるための所要時間は数分です。
- 「チュートリアル」は、Sun Java Studio IDE の特定の 1 つの機能に焦点を当てています。ある機能の詳細に関心がある場合は、これらを実行してみてください。例で説明している機能によって、初めからアプリケーションを構築する場合と、提供されたソースファイルを使用して構築する場合があります。チュートリアルは 1 時間以内で完成できます。
- 「概要ビデオ」は、技術の説明がビデオで提供されています。IDE の視覚的な概要や、特定の機能の詳細説明を見ることができます。概要ビデオにかかる時間は数分です。概要ビデオは、任意の個所で開始、終了することもできます。

オンラインヘルプ

Sun Java Studio Enterprise 7 IDE には、オンラインヘルプが用意されています。ヘルプキー (Microsoft Windows 環境では F1 キー、Solaris オペレーティング環境では Help キー) を押すか、「ヘルプ」>「ヘルプ (すべて)」を選択して開くことができます。ヘルプの項目と検索機能が表示されます。

アクセシブルな製品マニュアル

マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセシブルなマニュアルは以下の表に示す場所から参照することができます。

マニュアルの種類	アクセシブルな形式と格納場所
マニュアルとチュートリアル	形式: HTML 場所: `http://docs.sun.com`
チュートリアル	形式: HTML 場所: Developers Source ポータル (`http://developers.sun.com/jsenterprise`) のリンク「Examples & Code Camps」
リリースノート	形式: HTML 場所: `http://docs.sun.com`

第1章

概要とコンポーネントアーキテクチャ

コンポーネントベースの開発 (CBD)

コンポーネントベースの開発 (CBD: Component-Based Development) は、コンポーネントの作成、配布、使用がアプリケーション開発の効率性と信頼性の向上に役立つとして、高い評価を受けているエンジニアリング戦略です。成熟した CBD では、堅牢なコンポーネントモデルとコンポーネントを意識した IDE が統合されています。

コンポーネントの作成者は、特定のコンポーネントモデル (コンポーネントアーキテクチャ) の仕様に従ってコンポーネントを開発する仕事をします。コンポーネントモデルは、コンポーネント配布の手段を規定したコンポーネント構造を明確な形で定義します。円熟したコンポーネントモデルはまた、各コンポーネントが自己記述的であることを可能にし、その機能をコンポーネント使用者に知らせることができます。一般に、コンポーネントは、コンポーネントライブラリというコレクションで配布されます。

コンポーネントは、用途範囲の違いを考慮してさまざまな種類のものを作成することできます。たとえば、ある範囲の特定の開発ニーズに対応して、非常に汎用的に、あるいは水平的にコンポーネントをデザインすることができます。そうしたコンポーネントは、柔軟性とカスタマイズ性を長所とする非常に一般的なコンポーネントになる傾向があります。複数プロジェクトや企業にわたって、さらには Web アプリケーションの世界で多様なアプリケーション開発者層が利用可能であり、一般に特定のルック & フィールに偏りません。また一方で、より狭い、垂直型の一群の開発ニーズを満たすコンポーネントをデザインすることもできます。垂直型のコンポーネントは特定の用途向けに作られており、高レベルの機能と使い易さを提供することを可能にします。この種のコンポーネントの利用可能性は限定されますが、その用途範囲が綿密に定義されているため、パラメータ部分が最低限ですみ、特定のルック & フィールを採用することができます。

一般にコンポーネントの使用者は、アプリケーション開発者です。CBD では、アプリケーション開発の作業は、一群の再利用可能コンポーネントから特定のアプリケーションを組み立てることで構成されます。コンポーネントのカバー範囲が広いほど、アプリケーション専用のコードの量は少なくなります。

コンポーネントを意識した IDE は、コンポーネント使用者にコンポーネントを提示するのに必要です。IDE は自己記述的なコンポーネントを利用し、動的にコンポーネントを提供し、インスタンス化や構成ができるようにします。IDE はパズルの最後の一片ですが、非常に重要です。コンポーネントを意識した IDE がなければ、コンポーネントモデルを実際に使用することはできません。コンポーネントを意識した IDE がなければ、開発者は、他の Java クラスでそうであるように、その公開 API を使ってコンポーネントを利用できるだけです。これに対し、コンポーネントを意識した IDE があれば、開発者はコンポーネントのコレクションをブラウズして、視覚的にコンポーネントを組み立ててアプリケーションエンティティを作成し、さらに、コンポーネント別のプロパティシートに宣言的に情報を埋めていくことによって、コンポーネントを構成することができます。

このマニュアルの以降では、Web アプリケーションフレームワークのコンポーネントモデルと、コンポーネント作成者がそのコンポーネントを利用して、強力なコンポーネントライブラリを作成する方法を説明します。

Web アプリケーションフレームワークコンポーネントとは

Web アプリケーションフレームワークは特定の種類のオブジェクトに対してコンポーネントモデルをサポートしてきました。しかしながら、以前のコンポーネントモデルは、開発者が各コンポーネントの API を学び、自分のアプリケーションでそのコンポーネントを利用するためのコードを書くことを前提にしていました。以前は、このレベルの機能で十分であり、生産性の面で現代の競合製品に比べて大きな強みになっていたのですが、Web アプリケーションフレームワークでは、あらゆる種類の主な Web アプリケーションフレームワークアプリケーションオブジェクト (ビュー、モデル、コマンド) が包含され、コンポーネントモデルが大幅に拡張されています。また、IDE 用の Web アプリケーションフレームワークモジュールは、Web アプリケーションフレームワークアプリケーション用の視覚的な開発環境を構築する、コンポーネントを意識したフル装備の IDE を提供します。

Web アプリケーションフレームワークの用語では、コンポーネントとは、さまざまな種類の、サポートされているコンポーネントクラス (ビュー、モデル、コマンド) の 1 つとメタデータ情報を組み合わせたものです。このメタデータは、`ComponentInfo` クラスと呼ばれる Web アプリケーションフレームワーク固有のクラス内にカプセル化されています。デザイン時には、Sun Java System は ComponentInfo を点検し、使い易い視覚化した形式でコンポーネントを提供します。

ComponentInfo クラスに格納されているメタデータの目的は、Sun Java System などの開発環境でコンポーネントを自動的に利用できるようにすることにあります。開発者は、ComponentInfo クラスを定義しなくても、これまでと同様、自身のアプリケーションでさまざまな種類のコンポーネントを手動で作成、使用することができます。

Web アプリケーションフレームワークのコンポーネントライブラリ

Web アプリケーションフレームワークコンポーネントモデルでは、IDE がコンポーネントを検出できるよう、それらコンポーネントを特定のコンポーネントライブラリ形式にパッケージ化する必要があります。Web アプリケーションフレームワークのコンポーネントライブラリは、コンポーネントクラスからなる標準的な JAR ファイル 1 つと ComponentInfo メタデータクラス、コンポーネントライブラリのマニフェストファイル 1 つで構成されます。コンポーネントライブラリのマニフェストファイルは、このマニュアルの後の方で詳しく説明します。

注 – コンポーネントライブラリ JAR は、コンポーネント以外のものに関係する、任意の数のクラスを含むことができます。コンポーネントライブラリ JAR は、コンポーネントモデルに固有の追加が含まれている標準的な JAR ファイルにすぎません。

アプリケーション開発者は、Web アプリケーションフレームワークのコンポーネントライブラリを自身の Web アプリケーションの WEB-INF/lib ディレクトリに配置することによって、それらライブラリを利用することができます。IDE は、このディレクトリにあるすべてのコンポーネントライブラリを自動的に認識して、マウントします。IDE によってライブラリが検出され、点検が終えると (バックグラウンドのスレッドの待ち時間のために数分時間がかかることがある)、アプリケーション内でライブラリコンポーネントを利用できるようになります。つまり、コンポーネントが IDE に登録された状態になります。

ヒント：ライブラリマニフェストの点検とコンポーネントの登録は、IDE 内で Web アプリケーションフレームワークアプリケーションがマウントされるたびに行われます。コンポーネントモデルは完全に動的であるため、これは当然起きることと考えてください。コンポーネント作成者とアプリケーション開発者はともに、このプロセスがあることを認識しておくべきであり、アプリケーションの WEB-INF/lib ディレクトリからコンポーネントライブラリを削除すると (意図的かどうかには関わらず)、次回アプリケーションをマウントしたときに、そのライブラリのコンポーネントがなくなることに注意してください。

IDE における障害追跡のヒント：Web アプリケーションのマウントが不適切であることは、Sun Java System の初心者によくある間違いの 1 つです。マウントポイントが Web アプリケーション構造のルートに対応している場合、IDE の Web アプリケーションモジュール (Web アプリケーションフレームワークツールセットモジュールはこのモジュールを土台にしている) は マウントされているファイルシステムを 1 つの Web アプリケーションとしてだけ認識します。Web アプリケーションをそのルートディレクトリにマウントしていない場合、IDE は従来のファイルシステムとみなし、期待した Web アプリケーションフレームワークのアプリケーションビューを提供できません。初めてのコンポーネントを構築、テストするときは、これに注意してください。この点に関する混乱を避ける最も簡単な方法は、IDE の「ファイルシステムをマウント」アクションではなく、「Web Application をマウント」を使用する方法です。

標準の Web アプリケーションフレームワークコンポーネントライブラリ

標準の Web アプリケーションフレームワークコンポーネントライブラリには、主要インタフェースと実行時クラス、それに、Web アプリケーションフレームワークアプリケーションの作成に利用できる基本的なコンポーネントが多数含まれています。この Web アプリケーションフレームワークコンポーネントライブラリ は、1 つの JAR ファイルとしてパッケージ化され、アプリケーションの `WEB-INF/lib` ディレクトリに出現します。

IDE を使った Web アプリケーションフレームワークアプリケーションの作成では、標準の Web アプリケーションフレームワークコンポーネントライブラリの最新バージョンが、自動的にアプリケーションの `WEB-INF/lib` ディレクトリに追加されます。以前のバージョンの IDE で作成したアプリケーションを開くと、Web アプリケーションフレームワーク実行時ライブラリを含めてアプリケーションをアップグレードするよう促されることがあります。

コンポーネントクラス

Web アプリケーションフレームワークのコンポーネントクラスは、Web アプリケーションフレームワークの実行時の型 (ビュー、コマンド、モデルのどれか) を定義しているクラスです。

コンポーネントの作成者は、JavaBean 開発者が持つであろう程度にデザイン時の問題について関心を払うだけです。すなわち、コンポーネント作成者は、デザイン時の構成をデザインし、適切な `get` メソッドおよび `set` メソッドを定義する際に表示するプロパティを事前に把握しておく必要があります。しかしながら、JavaBean モデルと異なり、Web アプリケーションフレームワークコンポーネントモデルは、すべての `get` メソッドおよび `set` メソッドをプロパティとして表示するわけではありません。これは、Web アプリケーションフレームワークコアには、コンポーネントの

基になるメソッドとして、デザイン時の構成には向かない get メソッドおよび set メソッドが数多くあることを、Web アプリケーションフレームワークが認識しているためです。このため、Web アプリケーションフレームワークコンポーネントは、関係する ComponentInfo クラスに明示的に指定されているプロパティにだけ、表示するプロパティを制限します。

ComponentInfo クラス

ComponentInfo クラスは、Web アプリケーションフレームワークコンポーネントモデルの心臓部です。論理的には、各 Web アプリケーションフレームワークコンポーネントは、コンポーネントクラス 1 つと ComponentInfo クラス 1 つからなる組み合わせと定義することができます。ComponentInfo クラスは IDE が内観するメタデータを提供して、デザイン時にコンポーネントが存在することを示します。ComponentInfo クラスのデザインでは、デザイン時の問題にだけ集中することができます。ComponentInfo クラスは、Web アプリケーションフレームワークで実行時に何の役割も果たしません。

第2章

コンポーネントの開発

初めてのコンポーネントの開発

この章では、Web アプリケーションフレームワークコンポーネントの作成と配布、使用するために必要な基本的な手順を説明します。

練習問題として取り組み、実際にコンポーネントを構築、テスト実行してください。この章を完了すると、コンポーネント開発の工程全体をよく理解できるようになります。この時点では、細部の多くを理解しようとは思わないでください。このガイドの残りの部分で、各種のコンポーネントに関する細部、メタデータ形式の細部、またコンポーネント作成者が利用可能な特別なオプション機能を詳しく説明します。

この章は、Web アプリケーションフレームワークアプリケーションについて基本的な知識があることを前提にしています。

コンポーネントの種類の決定

ここでは、方法に焦点を当てるために考えられたごく単純な例を紹介します。「MyTextField」という新しい表示フィールドコンポーネントを作成します。目的は、boolean 値を取る「Foo」という新しいプロパティをこのコンポーネントに表示させることにあります。アプリケーション開発者は、視覚的に `MyTextField` を選択して、自分の Web アプリケーションフレームワークページに追加することができます。このコンポーネントは、Web アプリケーションフレームワークの標準の表示フィールドコンポーネントのすべてのプロパティに加えて、新しいプロパティ「Foo」を持つことになります。

コンポーネントクラスの作成

Web アプリケーションフレームワークでは、必ずしも新しいコンポーネントクラスは必要ありません。この点については、このガイドで後ほど説明します。ただし、この例では新しいコンポーネントクラスが必要なため、ここから始めます。

1. **任意の Java エディタで、`mycomponents` パッケージを作成します。**
2. **`mycomponents.MyTextField` クラスを作成します。**
3. **`MyTextField` に `com.iplanet.jato.view.BasicDisplayField` を拡張させます。**
4. **コンポーネントの種類に合わせて適切なコンストラクタを実装します。**

   表示フィールドコンポーネントはすべて、View 型 `parent`、String 型 `name` を取る、2 つの引数のコンストラクタを実装している必要があります。IDE は、すべての表示フィールドコンポーネントがこのコンストラクタを実装していることを前提にしています。
5. **「Foo」という名前の新しい boolean 型のプロパティ用の `get` メソッドおよび `set` メソッドを追加します。**

   これまでの手順を終えると、`mycomponents/MyTextField.Java` は以下のようになります。

```
package mycomponents;

import com.iplanet.jato.view.*;

/**
 *
 * @author component author
 */
public class MyTextField extends BasicDisplayField {

     /** MyTextField の新しいインスタンスを作成 */
     public MyTextField(View parent, String name) {
          super(parent, name);
     }

     public boolean getFoo() {
          return foo;
     }

     public void setFoo(boolean value) {
          foo = value;
     }

     boolean foo;
}
```

これから、このコンポーネントに新しいプロパティを作成しますが、実行時に実際にこのプロパティがコンポーネントとどのように対話するかは、定義されていません。この定義はコンポーネントの作成者が行いますが、ここでは説明しません。

ComponentInfo クラスの作成

ComponentInfo クラスは、IDE がコンポーネントを組み込むのに必要なデザイン時メタデータを定義します。この例では、既存の ComponentInfo を拡張し、本当のオブジェクト指向のスタイルで単にこのクラスを補強します。当然、ComponentInfo インタフェースを最初から実装する方法を選ぶこともできますが、この例では生産的ではありません。

1. mycomponents.MyTextFieldComponentInfo クラスを作成します。
2. MyTextFieldComponentInfo に com.iplanet.jato.view.html2.TextFieldComponentInfo を拡張させます。
3. 引数なしのコンストラクタを実装します。

4. コンポーネントの基本的なデザイン時説明を提供する `getComponentDescriptor()` メソッドを実行します。

5. IDE で表示させるプロパティを識別する `getConfigPropertyDescriptors()` メソッドを実装します。

   継承を利用して、`TextFieldComponentInfo` にすでに定義されているプロパティに新しい「Foo」プロパティを追加します。

   これまでの手順を終えると、`mycomponents/MyTextFieldComponentInfo.Java` は以下のようなコードになります。

注 – 具体例を紹介するための次のコード例には、文字列の値が直接埋め込まれています。表示文字列を各言語対応にする必要があると思われる場合は、リソースバンドルを利用してください。

```
package mycomponents;

import java.util.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.component.*;
import com.iplanet.jato.view.html2.*;


public class MyTextFieldComponentInfo extends TextFieldComponentInfo {

     public MyTextFieldComponentInfo()
     {
          super();
     }

     public ComponentDescriptor getComponentDescriptor() {

          // コンポーネントクラスを示す
          ComponentDescriptor result=new ComponentDescriptor(
               "mycomponents.MyTextField");

          // この名前が、コンポーネントのインスタンスの名前を決めるのに使用される
          result.setName("MyTextField");

          // この表示名が、コンポーネント選択で使用される
          result.setDisplayName("MyTextField Component");

          // この説明がコンポーネントのツールチップのテキストになる
          result.setShortDescription("A simple demonstration of a new component");

          return result;
     }
```

```
    public ConfigPropertyDescriptor[] getConfigPropertyDescriptors() {

        if (configPropertyDescriptors!=null)
            return configPropertyDescriptors;

        // スーパークラスに定義されているあらゆるプロパティを取得
        configPropertyDescriptors=super.getConfigPropertyDescriptors();
        List descriptors=new LinkedList(
            Arrays.asList(configPropertyDescriptors));

        ConfigPropertyDescriptor descriptor = null;

        //「foo」プロパティを追加
        descriptor=new ConfigPropertyDescriptor("foo",Boolean.TYPE);
        descriptor.setDisplayName("Foo Property");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptor.setDefaultValue(new Boolean(false));
        descriptors.add(descriptor);

        // 配列を作成 / 返す
        configPropertyDescriptors = (ConfigPropertyDescriptor[])
            descriptors.toArray(
                new ConfigPropertyDescriptor[descriptors.size()]);
        return configPropertyDescriptors;
    }

    private ConfigPropertyDescriptor[] configPropertyDescriptors;
}
```

コンポーネントライブラリのマニフェストの作成

Web アプリケーションフレームワークコンポーネントはパッケージ化され、通常の JAR ファイルで配布されます。標準の Java 規則に従って、あらゆるクラス (コンポーネント、ComponentInfo、その他補助ファイル) を JAR に入れてください。

さらに、Web アプリケーションフレームワークでは、Web アプリケーションフレームワークの特殊なライブラリマニフェストファイルをコンポーネントライブラリの JAR に含める必要があります。このファイルは、ライブラリ内のコンポーネントのまとまりを記述した単純な XML ドキュメントです ライブラリマニフェストには、任意の個数のコンポーネントを宣言できます。この場合は、作成したばかりのコンポーネント 1 つを宣言するだけです。

Web アプリケーションフレームワークのライブラリマニフェスト名は、`complib.xml` である必要があります。Web アプリケーションフレームワークのライブラリマニフェストは、JAR ファイル内の `/COMP-INF` ディレクトリに入れる必要があります。

1. complib.xml というファイルを作成します。
2. Web アプリケーションフレームワークのライブラリマニフェスト要件を満たす最低限の情報を追加します。
3. MyTextField コンポーネント用のコンポーネント宣言を追加します。

これまでの手順を終えると、COMP-INF/complib.xml は以下のようなコードになります。

注 – ツールを使って XML ファイルを作成した場合は、コードが次のようになっていることを確認してください。XML ツールの中には、ファイルを作成したときに自動的にルート要素を挿入するものがあります。ルート要素が、以下に示されているように <component-library> になっていることを確認してください。XML ファイルが不適切な場合、IDE はコンポーネントライブラリを検出できません。

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
        <tool-info>
                <tool-version>2.1.0</tool-version>
        </tool-info>
        <library-name>mycomponents</library-name>
        <display-name>My First Component Library</display-name>
        <!-- ここにアイコン
        <icon>
                <small-icon>/com/iplanet/jato/resources/complib.gif</small-icon>
        </icon>
        -->
        <interface-version>1.0.0</interface-version>
        <implementation-version>20030221</implementation-version>

        <component>
                <component-class>mycomponents.MyTextField</component-class>
                <component-info-class>mycomponents.MyTextFieldComponentInfo</component-
        info-class>
        </component>

</component-library>
```

コンポーネントライブラリの JAR ファイルの作成

すべてのコンポーネントクラスをを JAR にまとめて、1 つのライブラリとして配布できるようにします。

この JAR ファイルの名前は任意です。

1. **この例では、JAR ファイルの名前を** `mycomponents.jar` **にします。**

   必ずしも Java ソースファイルを JAR に含める必要はありません。

2. **アイコンイメージやリソースバンドルなどの必要な補助リソースを JAR に含めます。**

   この例では、そうしたリソースはありません。

   `mycomponents.jar` の内部構造は、以下に示すコードのようになります。

```
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
COMP-INF/complib.xml
```

コンポーネントのテスト

これで、ライブラリをテストし、配布する準備ができました。プロジェクトの例でテストしてください。この段階では、Web アプリケーションフレームワークモジュールがインストールされ、使用可能になっている IDE を使用する必要があります。IDE で Web アプリケーションフレームワークアプリケーションを構築した経験がない場合は、次に進む前に、Web アプリケーションフレームワークのマニュアルセットに含まれている『Web アプリケーションフレームワーク チュートリアル』を先に終了してください。

注意 – 作成したコンポーネントは、既存のあらゆる Web アプリケーションフレームワークアプリケーションでテストできます。しかし、この練習問題に取り組む過程で構築するすべてのコンポーネント例のためのテスト用アプリケーションとして使用する、新しい Web アプリケーションフレームワークアプリケーションを作成することを推奨します。一般に、この後の手順説明は、Web アプリケーションフレームワークのデフォルト値に従ってテストオブジェクトの名前 (たとえば Page1 など) が生成されることを前提にしています。テスト用アプリケーションのオブジェクト名が手順説明の名前と同じであれば、より楽に説明に従って手順を進めることができます。

---

1. **IDE で Web アプリケーションフレームワークアプリケーションを新しく作成します。**

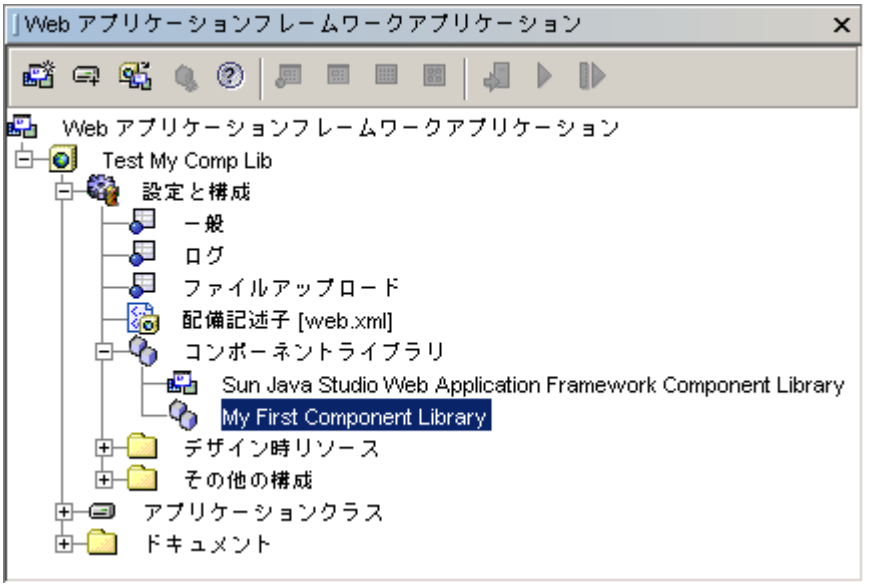
   アプリケーションの名前は自由に決められます。

2. 使用ファイルシステムで、新しい `mycomponents.jar` ファイルを、テスト用アプリケーション内の `WEB-INF/lib` ディレクトリにコピーします。

3. IDE のバックグラウンドスレッドがアプリケーションの `WEB-INF/lib` ディレクトリに新しい JAR が配備されていることを、検出するのを待ちます。

   IDE のバックグラウンドスレッドの再表示間隔によっては、この検出に数秒かかることがあります。

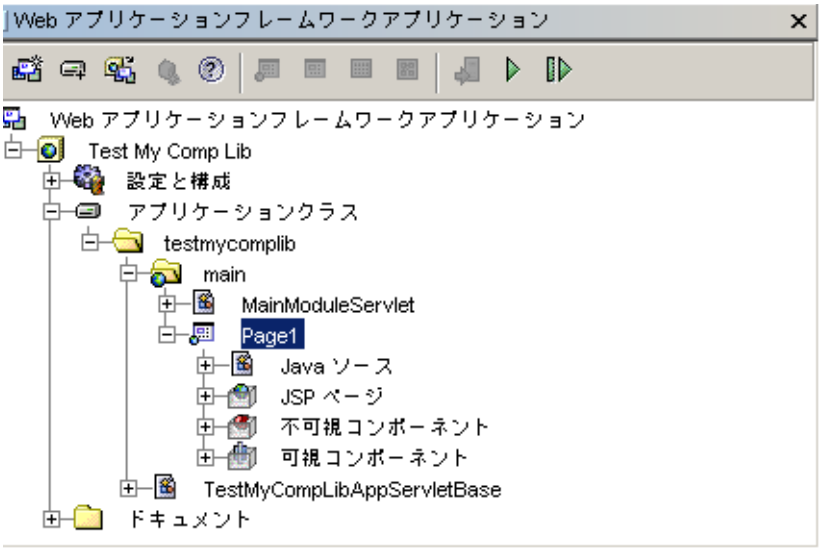
   このライブラリが完全に認識され、機能すると、下図に示すように、その Web アプリケーションフレームワークアプリケーションの「設定と構成」>「コンポーネントライブラリ」ノードの下に新しいライブラリノードが現れます。



4. 新しいページ (ViewBean) オブジェクトを作成します。

   ウィザードのデフォルト値をそのまま使用します。IDE によって「Page1」と命名されます。

5. 新しく作成した Page1 ノードを選択して、展開します。



6. Page1 に「MyTextField Component」のインスタンスを追加します。

   これは、2 つのユーザーインタフェースアクションのどちらを使っても行うことができます。

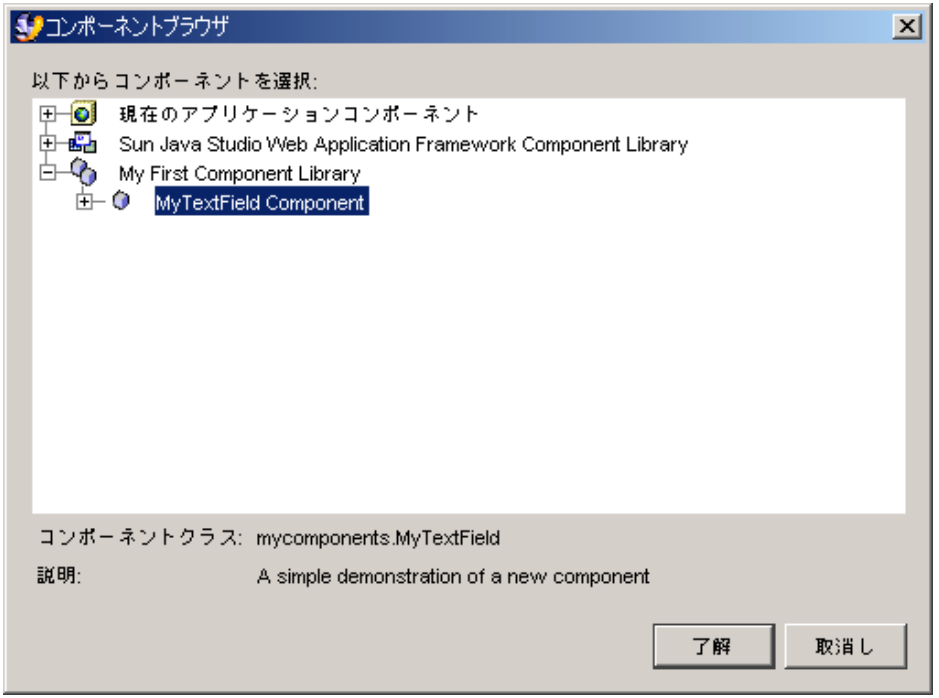
   - コンポーネントパレット (下図) を利用する方法
     - 「可視コンポーネント」セクションを展開します。
     - 「MyTextField Component」項目をクリックします。

     これは、その時点でフォーカスのページノードにインスタンスが追加されます。
   - Page1 の「可視コンポーネント」サブノードを選択して追加する方法

     サブノードを選択したら右クリックし、ポップアップメニューから「可視コンポーネントを追加」アクションを選択します。

     「My First Component Library」と「MyTextField Component」の一般的なアイコンに注目してください。これは、特定のアイコンを指定していないためです。特定のアイコンを指定する機能については、後で学びます。

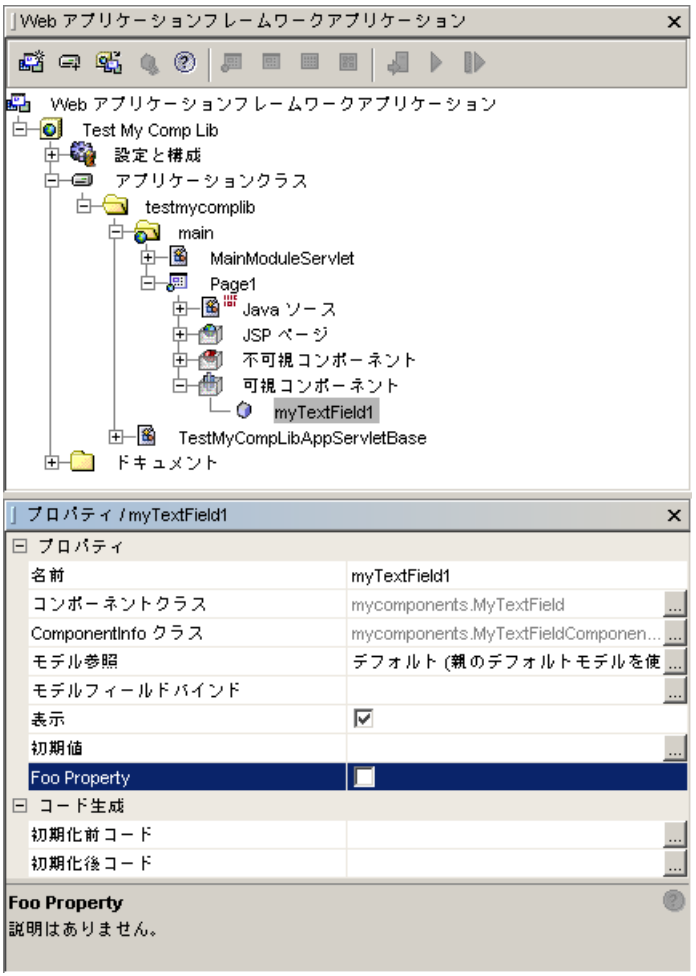
コンポーネントブラウザ (下図) は、コンポーネントパレット (上図) の代用となる機能です。後に説明する可視コンポーネントの追加に関係する手順は、コンポーネントパレットかコンポーネントブラウザのどちらを使っても行うことができます。両者は機能的には同じであり、常にどちらか一方を使っても、両方を使ってもかまいません。



コンポーネントパレットかコンポーネントブラウザから `MyTextField` コンポーネントを選択したら、「myTextField1」という子ビューがページに追加されることを確認します。

7. **子ノードの `myTextField1` を選択します。**

   IDE のプロパティシートによって、継承されたテキストフィールドコンポーネントのプロパティに加えて、新しいカスタムプロパティとして、「Foo Property」プロパティが追加されていることに注意してください。



「Foo Property」プロパティの動作テストをして、自分が考えているように動作することを確認します。

「Foo Property」プロパティには、`True` または `False` のいずれかの値を割り当てることができます。

8. この例では、「Foo Property」プロパティを `True` に設定します。

9. `Page1` Java ファイルにコードが生成されていることを確認します。

   `createChildReserved` メソッド内に、次に示すようなコードが含まれているはずです (実際のインデントは異なる可能性があります)。

```
...
if (name.equals(CHILD_MY_TEXT_FIELD1)) {
     mycomponents.MyTextField child =
          new mycomponents.MyTextField(this, CHILD_MY_TEXT_FIELD1);
     child.setFoo(true);
     return child;
}
...
```

配布

コンポーネントのテストと改良を終えたら、そのコンポーネントライブラリ JAR ファイルを開発者コミュニティーに配布することができます。そのコンポーネントを利用する各アプリケーションにそのコンポーネント JAR ファイルを追加するかどうかは、アプリケーション開発者の自由です。

Web アプリケーションフレームワークコンポーネントの詳細

Web アプリケーションフレームワークコンポーネントのデザイン意図は、アプリケーション開発者が Web アプリケーションフレームワークの実行時の種類 (ビュー、コマンド、モデル) をより迅速に定義できるようにすることにあります。ただし、Web アプリケーションフレームワークコンポーネントがアプリケーション開発者のデザイン時の経験にどのように反映されているかは、Java のオブジェクト指向に接した機会の多様さ (たとえばクラスのサブタイプ化とオブジェクトのインスタンス化など) によって異なります。

Java プログラマとして豊富な経験を持つ Web アプリケーションフレームワークのコンポーネント作成者は、Web アプリケーションフレームワークのアプリケーション開発者が開発プロセスに新しいコンポーネントを組み込む方法を簡単に予測するはずです。そうしたコンポーネント作成者は、アプリケーション開発者がある種類のコンポーネントはサブクラス化し、別の種類のコンポーネントはインスタンス化しようとすることを認識します。ある状況では、完全に有効で完全に構成したブラックボックスとしてコンポーネントを配布できるが、別の状況では、コンポーネントの使用法をそれぞれに設定するようにアプリケーション開発者に要請することを理解しています。

Web アプリケーションフレームワークのコンポーネントモデルと IDE は一体となって、コンポーネント作成者およびアプリケーション開発者が Java オブジェクト指向の多様性のすべてを利用できるようにします。この節では、オブジェクト指向の基本構成要素としての各コンポーネントの役割を明確にするに当たって、Web アプリケーションフレームワークコンポーネントモデルに特有の用語を詳しく説明します。

コンポーネントの説明は、しばしば、多重定義の度合いの大きい用語で溢れています。Web アプリケーションフレームワークコンポーネントをより正確な説明するための土台を提供するに当たり、一部、紛らわしい多重定義用語に頼らないようにするために作成された用語もあります。

配布可能なコンポーネントとアプリケーション固有 (配布不可) のコンポーネント

技術的には、あらゆる Web アプリケーションフレームワークオブジェクト (モデル、ビュー、コマンド) はそれぞれに 1 つのコンポーネントです。しかしながら、必ずしもすべての Web アプリケーションフレームワークコンポーネントがコンポーネントライブラリでの配布を目的としているわけではありません。一部のコンポーネントは、技術的には、あらゆるモデル、ビュー、コマンドが 1 つのコンポーネントである IDE 上でのアプリケーション構築の標準プロセスの一環として単に構築されます。ライブラリに含まれるコンポーネントを配布可能なコンポーネント、アプリケーション内で単に構築されるコンポーネントをアプリケーション固有のコンポーネント (または配布不可のコンポーネント) と呼んで区別します。この区別は純粋に用語の区別であり、厳密で正式な区別ではありません。配布可能なコンポーネントとアプリケーション固有のコンポーネントの間に種類の違いはありません。この区別は便宜上の分類にすぎず、コンポーネント作成者の役割とアプリケーション開発者の役割の区別に役立てることを意図しています。アプリケーション開発者は、アプリケーション固有のコンポーネントを開発し、コンポーネント作成者は配布可能なコンポーネントを開発します。

アプリケーション固有のコンポーネントまたは配布不可コンポーネントは、それが定義されているアプリケーション内でのみ再利用可能なコンポーネントを意味します。これらのコンポーネントがコンポーネントライブラリにパッケージ化されることはありません。一般にアプリケーション固有のコンポーネントには、明示的な `ComponentInfo` が関連付けられていません。1 例として、アプリケーション開発者がアプリケーションでコンテナビューまたはモデルを構築するということは、暗黙に、アプリケーション固有のコンポーネントを構築することを意味します。これは、`Javax.swing` アプリケーション開発者がアプリケーションに固有のパネルやフレームを構築することに似ています。IDE では、こうしたアプリケーション固有のコンポーネントを直接操作することができます。このため、開発者は追加作業なしに、同じアプリケーション内でそれらコンポーネントを利用できます。たとえばアプリケーションに固有の新しいモデルを作成した後、アプリケーション開発者は、その同じアプリケーション内で視覚的にその新しいモデルをビューに接続することができます。アプリケーション固有のコンポーネントの開発は透過的で暗黙であり、それ自体はコンポーネントの作成に関する知識を必要としません。

アプリケーション固有 (配布不可) のコンポーネントは、以下のようなコンポーネントです。

- 一般のアプリケーション開発者が暗黙で開発したコンポーネント
- 同じアプリケーション内での使用だけを目的とするコンポーネント
- 明示的な `ComponentInfo` を伴わないコンポーネント

これに対し配布可能なコンポーネントは、多数のアプリケーションで再利用可能なコンポーネントを意味します。コンポーネント作成者は、配布可能なコンポーネントをコンポーネントライブラリにパッケージ化します。一般に、配布可能なコンポーネントごとに明示的な `ComponentInfo` クラスを 1 つを作成します。通常、再利用というより大きな目的があるため、配布可能なコンポーネントの作成には、デザインに際してより慎重であることが求められます。再び `Javax.swing` に例えれば、配布可能なコンポーネントは、多くの新しいアプリケーションでの使用を目的として配布される、新しいサブタイプの `Javax.swing.JPanel` ということになります。Web アプリケーションフレームワークアプリケーションでは、配布可能なコンポーネントは新しい種類の表示フィールド、あるいは個別化されてはいるが、非常に再利用可能性に優れたコンテナビューです。

配布可能なコンポーネントは、以下のようなコンポーネントです。

- コンポーネントモデルの知識を持つ者 (コンポーネント作成者) によって明示的に開発されたコンポーネント
- 異なるアプリケーションでの利用を目的とするコンポーネント
- 明示的な `ComponentInfo` クラスを伴うコンポーネント
- 配布のためにライブラリにパッケージ化されたコンポーネント

当然、一般的なボトムアップのデザイン慣行に従えば、アプリケーション固有のコンポーネントを明示的に配布可能ステータスに「昇格」させることはあり得ます。こうしたことは、開発チームが別のアプリケーションへの再利用の候補としてアプリケーション固有のコンポーネントを認めたときに起こります。これは通常起こり得ることで、奨励されます。アプリケーション固有のコンポーネントを配布可能ステータスに昇格するには、配布可能なコンポーネントの基準を満たしていればよいわけです。

このため、アプリケーション固有のコンポーネントの作成の単純さや透過性と比較した場合、より複雑なのは配布可能なコンポーネントの作成です。従ってこのガイドではほとんどが配布可能なコンポーネントの作成についての説明となっています。

拡張可能コンポーネントと拡張不可コンポーネント

Web アプリケーションフレームワークのコンポーネントライブラリでは、拡張可能コンポーネントと拡張不可コンポーネントの間に正式な区別があります。コンポーネント作成者は、コンポーネントが拡張可能または拡張不可のどちらであるかを指定する必要があります。この区別によって、コンポーネント作成者は、コンポーネントを

意識する IDE が、アプリケーション開発者の使用法に応じてコンポーネントを表示する方法を制御することができます。IDE は、綿密に定義された、区別した方法で拡張可能コンポーネントと拡張不可コンポーネントの両方を表示します。

コンポーネント作成者にとって拡張可能と拡張不可の区別は重要ですが、現実にコンポーネント使用者がこの区別を意識することはまったくありません。すなわち、これらのどちらであるかを、IDE がアプリケーション開発者に提示することはありません。むしろ、IDE はこれらの微妙な点を自動的に管理し、アプリケーション開発者がアプリケーションの開発にだけ集中できるようにします。一般にアプリケーション開発者が、コンポーネントが拡張可能であるかどうか、またコンポーネントに `ComponentInfo` クラスがあるかどうかを意識する必要はありません。

拡張可能コンポーネント

コンポーネントを拡張可能にするのが適しているのは、アプリケーションの構築で Web アプリケーションフレームワークの新しいサブタイプ (たとえば、新しい種類のモデル、コンテナビュー、コマンドなど) を宣言する必要がある場合です。

IDE は拡張可能コンポーネントを表示して、アプリケーション開発者が直接サブクラス化できるようにします。アプリケーション開発者が使用可能なコンポーネント一覧から拡張可能コンポーネントを選択すると、最終的に IDE は、選択されたコンポーネントのクラスを拡張する新しい Java クラスを作成します。アプリケーションに固有のサブタイプ化によってコンポーネントが正しい方法で使用されると予想される場合、コンポーネント作成者は、そのコンポーネントを拡張可能コンポーネントに指定すべきです。その指定があると、Web アプリケーションフレームワークがその拡張可能コンポーネントの適合先を指示します。あるアプリケーションエンティティがフレームワークエンティティのサブタイプの 1 つである必要がある場合、Web アプリケーションフレームワークは必ず拡張可能コンポーネントが関係する場所でそのことを示します。

以下の例に示すように、拡張可能コンポーネントであることは、コンポーネントライブラリのマニフェスト内の `<extensible-component>` で指定します。

```
<extensible-component>
     <component-class>com.iplanet.jato.view.BasicViewBean</component-class>
     <component-info-class>com.iplanet.jato.view.BasicBeanComponentInfo</component-info-
class>
</extensible-component>
```

拡張可能コンポーネントには、以下のような特徴があります。

- アプリケーション開発者が拡張可能コンポーネントを拡張する新しい種類を作成することを可能にする。
- 抽象化できることがある。
- 新しい種類用のテンプレートとして役立つコンポーネント固有の Java ファイルを指定できる。

拡張可能なコンポーネント例 : 拡張可能な ViewBean、コンテナビュー、モデル、コマンドコンポーネント

拡張不可コンポーネント

コンポーネントを拡張不可にするのが適しているのは、アプリケーションの構築でインスタンスを単純に宣言して、構成すればよい場合です。

アプリケーション開発者が拡張不可コンポーネントを選択すると、最終的にアプリケーション固有のクラスにそのコンポーネントの新しいインスタンスが定義されます。たとえば開発者によってコンテナビューにテキストフィールドかボタンが追加されると、IDE はコンテナビュークラス内で必ずそのデザインをそのテキストフィールドまたはボタンのインスタンスの宣言に変換します。こうして、アプリケーション開発者は、アプリケーション固有のクラスに拡張不可コンポーネントを埋め込んでいきます。これは、古典的な「構築」方式のコンポーネントベースの開発です。

Web アプリケーションフレームワークは、ここでもまた、拡張不可コンポーネントが適切な場所を指示します。たとえば、アプリケーションレベルのページ (ViewBean) やページレット (コンテナビュー) コンポーネントの開発は、アプリケーション開発者はそのコンポーネントに、表示フィールドなどの子ビューオブジェクトを追加できるようにしようと考えます。その結果、IDE は拡張不可コンポーネントの一覧をアプリケーション開発者に提供して、ページまたはページレットに直接追加できるようにします。

以下の例に示すように、拡張不可コンポーネントであることは、コンポーネントライブラリのマニフェスト内の <component> で指定します。

```
<component>
     <component-class>com.iplanet.jato.view.BasicChoiceDisplayField</component-class>
     <component-info-class>com.iplanet.jato.view.html2.ListBoxComponentInfo</component-
info-class>
</component>
```

拡張不可コンポーネントには、以下のような特徴があります。

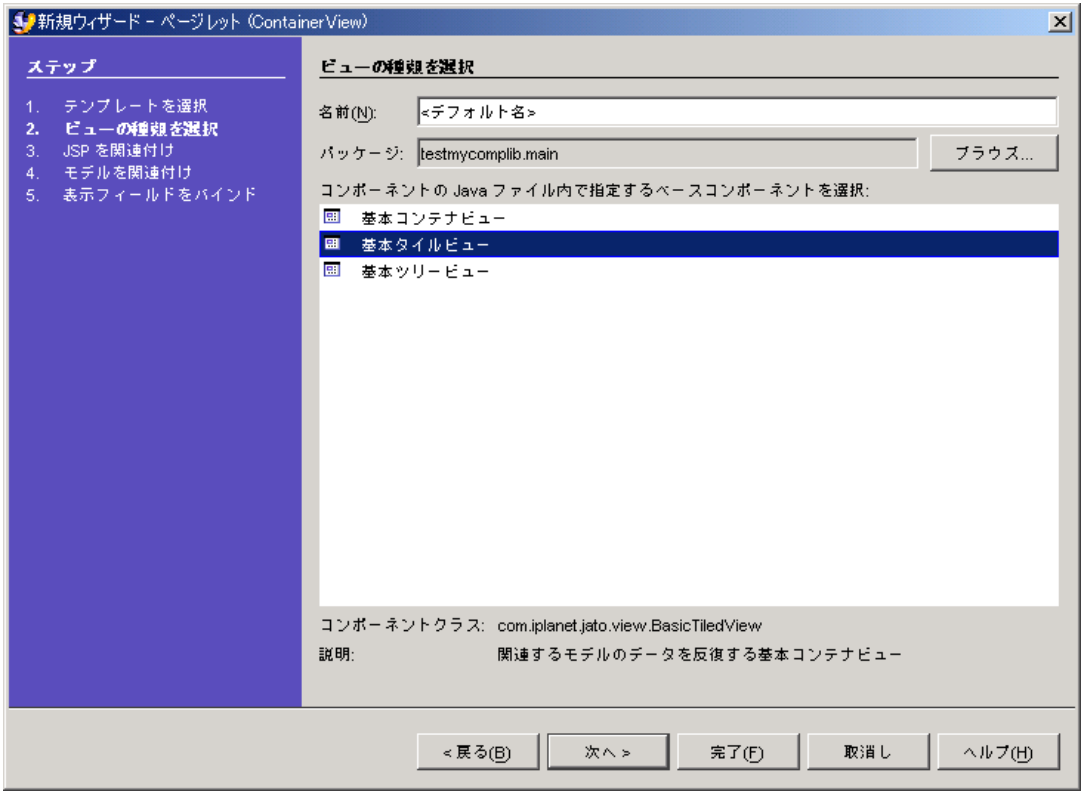
- アプリケーション開発者がコンポーネントの新規インスタンスを簡単に宣言、設定することを可能にする。
- 抽象化できない。
  - きめの細かいコンポーネント例 : 表示フィールドコンポーネント
  - きめの粗いコンポーネント例 : 事前にパッケージ化され完全に構成されている拡張不可のコンテナビュー、モデル、コマンドコンポーネント

IDE における拡張可能コンポーネントと拡張不可コンポーネント

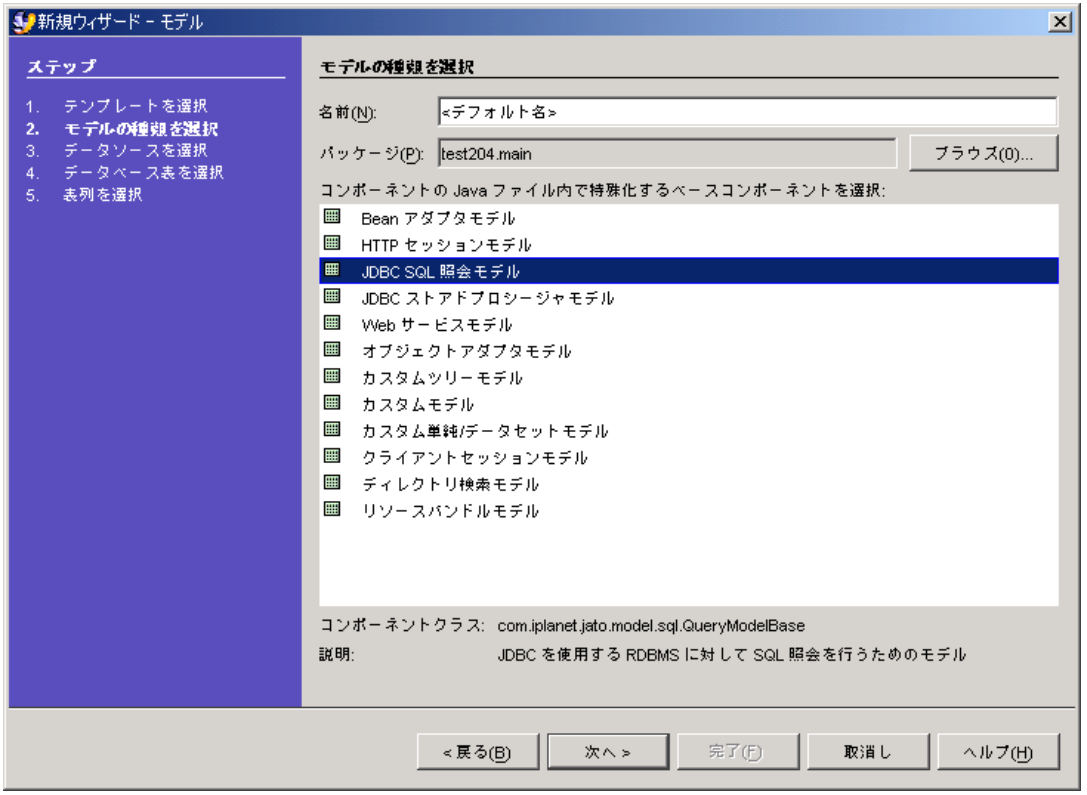
拡張可能コンポーネントと拡張不可コンポーネントの区別が紛らわしいとまだ思われる場合は、この時点で、IDE を参照して、IDE がどのように透過的に拡張可能コンポーネントおよび拡張不可コンポーネントを表示しているかを見てみると良いでしょう。

1. Web アプリケーションフレームワークプロジェクトを開き、「main」モジュールのフォルダを選択します。
2. フォルダ上で右クリックして、「追加」>「モデル」、「追加」>「ページ (ViewBean)」、「追加」>「ページレット (ContainerView)」のいずれかを選択します。

   以下に示すように、これらのアクションは、拡張可能用のコンポーネントブラウザが組み込まれたウィザードを起動します。



3. ウィザードでの操作を完了します。IDE によって、拡張可能コンポーネントのクラスを拡張する新しいクラスが作成されます。



4. **既存のページまたはページレットノードを選択します。**

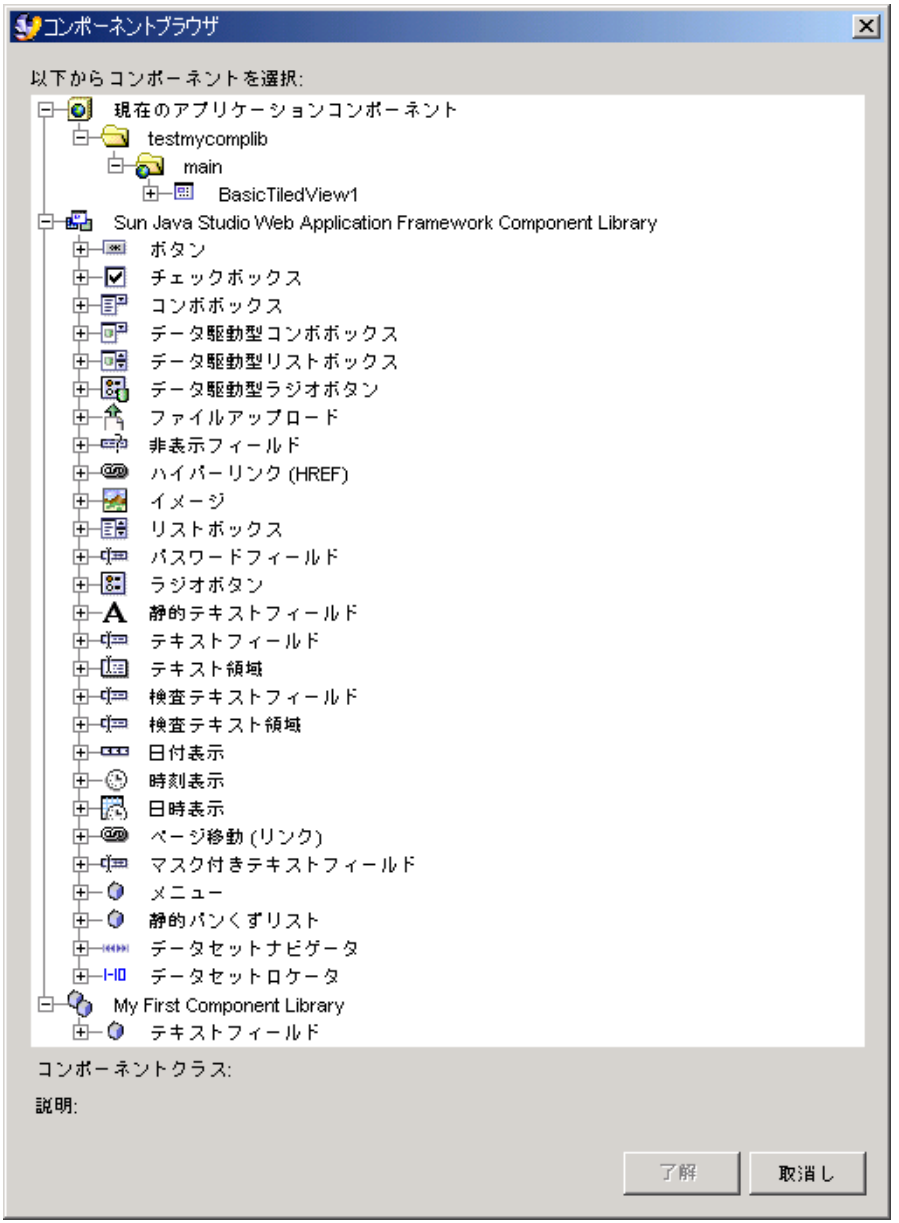
   最上位のノードを展開して、その中の「可視コンポーネント」ノードを見られるようにします。

5. **「可視コンポーネント」サブノードを選択し、右クリックして、「可視コンポーネント 追加」アクションを選択します。**

   これで、拡張不可用のコンポーネントブラウザが起動します (下図を参照)。

6. **子ビューの選択を完了します。**

   この手順によって新しいクラスが作成されるわけではなく、現在の選択されているクラスに子の要素が追加されます。

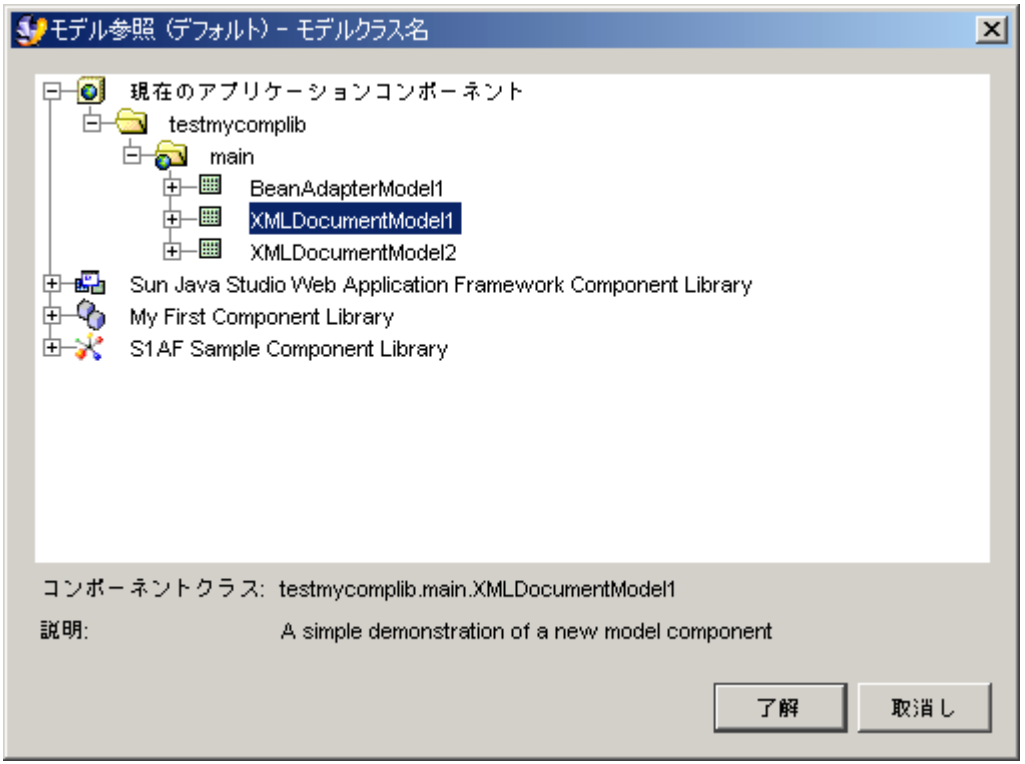


上図は、「可視コンポーネント 追加」アクションの関係で使用されている拡張不可用コンポーネントブラウザの例です。ブラウザが完全に展開されていて、ライブラリ 2 つと現在のアプリケーションの拡張不可コンポーネントが表示されています。

IDE のその他の部分では、拡張不可用コンポーネントブラウザは、特定のプロパティ値に割り当てる際のページ / ページレット、モデル、コマンドの選択に使用されます。たとえば、プロパティで Web アプリケーションフレームワークの使用関係

(ビューはモデルを使用するなど) が表されている場合、プロパティエディタは必ず拡張不可用コンポーネントブラウザを利用して、アプリケーション開発者が有効なターゲットオブジェクトを選択できるようにします。

たとえば「モデルクラス名」型のプロパティの場合は、マウントされているコンポーネントライブラリがある場合にそこに含まれる拡張不可モデルコンポーネントと、現在のアプリケーションに追加されているすべてのモデルを表示する拡張不可用コンポーネントブラウザを使って編集します。同様の動作は、「コマンドクラス名」プロパティの編集にも当てはまります。ただし、その場合は、モデルコンポーネントではなく、コマンドコンポーネントを選択します。



上図は、モデルクラス名のプロパティエディタの関係で使用されている拡張不可コンポーネントブラウザの例です。ブラウザが完全に展開されていて、ライブラリ 2 つと現在のアプリケーションの拡張不可モデルコンポーネントが表示されています。

ComponentInfo の詳細

ComponentInfo クラスは、Web アプリケーションフレームワークコンポーネントモデルの心臓部です。論理的には、各 Web アプリケーションフレームワークコンポーネントは、コンポーネントクラス 1 つと ComponentInfo クラス 1 つからなる組み合わせと定義することができます。ComponentInfo クラスは IDE が内観するメタデータを提供して、コンポーネントがデザイン時に存在することを示します。ComponentInfo クラスのデザインでは、デザイン時の問題にだけ集中できます。ComponentInfo クラスは、Web アプリケーションフレームワークで実行時に何の役割も果たしません。

実際の ComponentInfo クラスは com.iplanet.jato.component.ComponentInfo インタフェースか、そのサブインタフェースの 1 つを実装している必要があります。ComponentInfo クラス名の最後は、「ComponentInfo」接頭辞である必要があります。実際的な場合はいつでも、ComponentInfo クラスとコンポーネントクラスのベース名は同じ名前 (Foo と FooComponentInfo など) にすること推奨します。

次に示すコードの一部は、Web アプリケーションフレームワークコンポーネントライブラリのマニフェストがどのようなものかを示しています。単純なコンポーネント宣言は、コンポーネントクラスと ComponentInfo の組み合わせと見ることができます。<extensible-component> という余分な指定があることに注目してください (Web アプリケーションフレームワークコンポーネントのマニフェストの詳細は、178 ページの「コンポーネントマニフェスト」を参照)。

```
<extensible-component>
     <component-class>com.iplanet.jato.view.BasicViewBean</component-class>
     <component-info-class>com.iplanet.jato.view.BasicViewBeanComponentInfo</component-
info-class>
</extensible-component>
```

ただし、Web アプリケーションフレームワークでは、ComponentInfo クラスとそれに関係するコンポーネントクラスのベース名が異なっていてもかまいません。実際、Web アプリケーションフレームワークでは、複数の ComponentInfo クラスを 1 つのコンポーネントクラスに関連付けることができます。前述したように、論理的には、コンポーネントは、1 つのコンポーネントクラスと 1 つの ComponentInfo からなる組み合わせです。1 つのコンポーネントクラスが複数のそうした組み合わせに関わっている場合があります。

大部分のコンポーネント作成にとって、このことは直感的ではないかもしれませんが、Web アプリケーションフレームワークコンポーネントモデルの非常に効果的で強力な機能の 1 つです。たとえば com.iplanet.jato.view.html2 には、実際には同じコンポーネントクラスに関連付けられている複数の ComponentInfo クラスがあります。たとえば ListBoxComponentInfo、

RadioButtonsComponentInfo、および ComboBoxComponentInfo クラスはどれも、そのコンポーネントクラスとして com.iplanet.jato.view.BasicChoiceDisplayField を指定しています。

以下は、実際の Web アプリケーションフレームワークコンポーネントライブラリのマニフェストの抜粋で、上記のコンポーネントの組み合わせを見ることができます。

```
<component>
     <component-class>com.iplanet.jato.view.BasicChoiceDisplayField</component-class>
     <component-info-class>com.iplanet.jato.view.html2.ListBoxComponentInfo</component-
info-class>
</component>
<component>
     <component-class>com.iplanet.jato.view.BasicChoiceDisplayField</component-class>
     <component-info-
class>com.iplanet.jato.view.html2.RadioButtonsComponentInfo</component-info-class>
</component>
<component>
     <component-class>com.iplanet.jato.view.BasicChoiceDisplayField</component-class>
     <component-info-class>com.iplanet.jato.view.html2.ComboBoxComponentInfo</component-
info-class>
</component>
```

これらのペアは明確に異なる 3 つの組み合わせ、つまり、明確に異なる 3 つの論理コンポーネントを形成します。この自由さが提供する価値は、単に新しい ComponentInfo クラスを定義するだけで、新しいバリエーションのコンポーネントを作成できることにあります。

注 – 同じコンポーネントクラスを使用する他のコンポーネントと区別するには、新しいコンポーネントは、他のコンポーネントが表示しない既存のコンポーネントプロパティを表示するか (たとえば、「複数選択可」プロパティを表示するのは ListBoxComponentInfo だけ)、他の意味のあるコンポーネントメタデータを変更する必要があります。上記の例のコンポーネントの違いは、主としてそれらが宣言する JSP タグの違いにあり、HTML ページに追加されると、それらコンポーネントのルック & フィールが大きく異なるものになります。ただし、コンポーネントの機能そのものは、本質的にそれらのどのコンポーネントも同じです。異なるタグを宣言できること、つまり、1 つのコンポーネントに対して異なる生成方法があるということは、その土台として同じコンポーネントクラスを使用するコンポーネントを定義する最も争いがたい理由です。

宣言メタデータと異なり、ComponentInfo クラスは Java クラスとして指定します。このため、既存の ComponentInfo から新しい ComponentInfo を得ることができ、スーパークラス機能の標準的な継承の恩恵を受けることができます。より固有でより適切なサブタイプがないのであれば、com.iplanet.jato.component.SimpleComponentInfo クラスを任意の新しい ComponentInfo クラスの信頼できる出発点として役立てることができます。

個別化された ComponentInfo インタフェース

Web アプリケーションフレームワークには、コンポーネント作成者がいくつかのコンポーネントに応じて適切な追加のメタデータを指定できるよう、いくつかの個別化されたサブタイプの `ComponentInfo` が用意されています。IDE では、それらの追加のメタデータを利用し、それらと適合する特殊なビジュアル開発のサポートを提供します。

ExtensibleComponentInfo

`com.iplanet.jato.component.ExtensibleComponentInfo` を使用して、開発者は、特に拡張可能コンポーネントに合った追加のメタデータを提供することができます。IDE では、拡張可能コンポーネントは、開発者が新しい Web アプリケーションフレームワークの種類 (モデル、ページやページレット、コマンド) を作成する際のベースクラスとして役立ちます。コンポーネント作成者は、この目的のために `ExtensibleComponentInfo` インタフェースに定義されている特別なメタデータを使用し、新しい種類の構成を制御することができます。具体的には、コンポーネント作成者は、拡張可能コンポーネントから派生するあらゆる新しい種類に対する出発点として役立てる Java クラステンプレートを指定することができます。

その他の種類の個別化された ComponentInfo

個別化された種類の `ComponentInfo` は、その他にもいくつかあります。

- `com.iplanet.jato.component.ExtensibleComponentInfo`
- `com.iplanet.jato.view.ViewComponentInfo`
- `com.iplanet.jato.view.ContainerViewComponentInfo`
- `com.iplanet.jato.command.CommandComponentInfo`
- `com.iplanet.jato.model.ModelComponentInfo`
- `com.iplanet.jato.model.ExecutingModelComponentInfo`

これらインタフェースについては、これらの個別化された `ComponentInfo` インタフェースが関係するさまざまな種類のコンポーネントを作成する手順を説明する章で詳しく説明します。

標準実装の ComponentInfo

Web アプリケーションフレームワークコンポーネントモデルは、綿密に定義されたインタフェースに基づいているため、コンポーネント作成者は、最初から任意の新しいコンポーネント用にそれらインタフェースを実装することができます。ただし、一般に Web アプリケーションフレームワークに実装されている各種の個別化された `ComponentInfo` インタフェースは既製品であり、コンポーネント作成者に対しては、独自のコンポーネントを作成する際は既存の実装の 1 つを拡張することを推奨します。そうすることが労力を省き、作成時間の短縮になります。

第3章

ビューコンポーネントの開発

この章は、7 ページの「初めてのコンポーネントの開発」をすでに読み終えていることが前提になります。

ビューコンポーネント

Web アプリケーションフレームワークのビューの予備的な情報は、『Web アプリケーションフレームワーク 開発ガイド』をご覧ください。

ビューコンポーネントは、可視コンポーネントともいいます。「ビュー」という用語は、Model-View-Controller デザインパターンに由来します。このため、`com.sun.iplanet` ライブラリに含まれている種類の大部分は、ビューという用語を使用しています。ただし、IDE では、企業の開発者にとってわかりやすくするために、ビューよりも可視コンポーネント、ViewBean よりもページの方をよく使用しています。

このマニュアルの目的上、以下のことを了解しておいてください。

- ビューコンポーネントと可視コンポーネントは同義である。
- 子ビューコンポーネントと子可視コンポーネントは同義である。
- ページコンポーネントと ViewBean コンポーネントは同義である。
- ページレットコンポーネントとコンテナビューコンポーネントは同義である。

大まかに言えば、ビューコンポーネントは以下の 2 種類あります。

- 拡張可能ビューコンポーネント
- 拡張不可ビューコンポーネント

拡張可能ビューコンポーネントは、アプリケーション開発者によるさらなる個別化を意図した、Web アプリケーションフレームワークにカスタム実装されたコンテナビューです。たとえば、Web アプリケーションフレームワークコンポーネントライブラリ内の基本コンテナビュー、基本タイルビュー、基本 ViewBean はどれも拡張可能ビューコンポーネントです。

上記の「アプリケーション開発者による個別化」という部分をあまり難しく考えないでください。アプリケーション開発者が行う個別化といっても、子ビューコンポーネントの追加 (IDE で行う) と、それらに関連付けるロジックを追加するだけ、ということがよくあります。

最も見分けがつき、最も簡単に理解しやすい拡張不可ビューコンポーネントは、表示フィールドインタフェースをカスタム実装したものです。Web アプリケーションフレームワークコンポーネントライブラリには、1 ダースを超える表示フィールドコンポーネントが含まれています。これらは古典的なウィジェットあるいはビジュアル制御戦略に確実に分類されるものであり、これらのコンポーネントのことは、コンポーネント開発者もアプリケーション開発者も同様に直感的に理解できます。Web アプリケーションフレームワークはこの最小限のコンポーネントの枠組みを超え、多くのコンポーネント作成者やアプリケーション開発者が以前に目にしたものに比較して、より大きな可能性をコンポーネント分野で提供します。

たとえば、IDE が作成した具体的なコンテナビューの実装はすべて、見分けにくい拡張不可ビューコンポーネントになります。また、アプリケーション開発者が作成するあらゆるコンテナビューは拡張不可コンポーネントです。これは、ほぼあらゆるものがコンポーネントになる Web アプリケーションフレームワーク手法の巧妙な点の 1 つです。こうした様々な種類のコンポーネントの相違点は、配布および再利用のためにそれらをパッケージ化する方法にあります。

ViewComponentInfo

ViewComponentInfo インタフェースを使用し、コンポーネント作成者は、あらゆるビューコンポーネントに適用可能な追加のメタデータを指定することができます。このインタフェースは、拡張可能および拡張不可両方のビューコンポーネントに適用でき、JSP タグをビューコンポーネントに関連付けるなどのメタデータが含まれます。

上記で示したように、1 つのコンポーネントクラスに複数の `ComponentInfo` クラスと組み合わせることによって、さまざまなコンポーネントを作成することができます。たとえば `ListBoxComponentInfo`、`RadioButtonsComponentInfo`、および `ComboBoxComponentInfo` のどれも、そのコンポーネントクラスとして `BasicChoiceDisplayField` を指定しています。これらは、明確に異なる 3 つの組み合わせ、つまり、明確に異なる 3 つの論理コンポーネントを形成します。これら 3 つのコンポーネントを互いに異なるものとしている主な相違点の 1 つは、それぞれが `ViewComponentInfo` の `getJspTagDescriptors()` メソッドを実装していて、異なる `JspTagDescriptor` を返す点です。つまり、これらのコンポーネントは、コンポーネントのインスタンスをアプリケーションに追加する際に IDE ツールが生成する JSP タグが異なることを除けば、ほぼ同じです。このことによって、コンポーネン

ト作成者にはかなりの自由を得るチャンスが生まれます。コンポーネント作成者は、異なるマークアップを生成する JSP タグからなる新しいライブラリを作成し、単に追加の `ComponentInfo` クラスを実装することによってそれらのタグを既存のコンポーネントクラスと組み合わせることができます。

ContainerViewComponentInfo

ContainerViewComponentInfo インタフェースを使用し、コンポーネント作成者は、あらゆるコンテナビューコンポーネントに適用可能な追加のメタデータを指定することができます。このインタフェースは、拡張可能ビューコンポーネントにのみ適用できます。

拡張不可ビューコンポーネントの開発

この節では、入力に対する初歩的な妥当性検査機能をサポートするテキストフィールドコンポーネントを新しく作成する方法について説明します。簡潔にするため、妥当性検査のデザインと実装は最低限のものに留めます。この練習問題は、拡張不可ビューコンポーネントデザインの仕組みを理解するためのものであるため、可能な妥当性検査のサポートについては表面をなぞるだけにします。

注 – Web アプリケーションフレームワークのコンポーネントライブラリ 2.1.1 には、完全に製品化された ValidatingTextField コンポーネントがすでに含まれていますが、このトレーニング用の練習問題では、Web アプリケーションフレームワークのコンポーネントライブラリの機能に近いテキストフィールド検査コンポーネントを作成します。ただし、この練習問題は製品コンポーネントの全機能を実装することを目的にしているわけではないため、作成されるコンポーネントは、Web アプリケーションフレームワークコンポーネントライブラリにあるものと同等ではありません。

この例では、ViewComponentInfo の利用、新しい JSP タグの開発、ConfigurableBean の開発など、Web アプリケーションフレームワークコンポーネントモデルの話題をいくつか取り上げます。

テキスト検査コンポーネントには、次のデザイン時機能を持たせます。

- 「Validator」というプロパティを表示する。このプロパティは、Validator オブジェクトの参照を受け取ります。表示フィールドは、Validator オブジェクトに妥当性検査を委託します。
- 「Validation Failure Message」というプロパティを表示する。このプロパティは、単純な文字列を受け取ります。

テキスト検査コンポーネントには、次の実行時機能を持たせます。

- 入力があった後、その入力値を委託して、Validator オブジェクトの検査を受けるようにする。
- 値が不正で、アプリケーションがページを再表示した場合、検査コンポーネントがその不正値と、アプリケーション開発者提供の検査障害メッセージを表示するようにする。

これらの要件を満たすには、次のクラスをデザイン、実装します。

- コンポーネントクラス - `mycomponents.ValidatingDisplayField`
- `ComponentInfo` クラス - `mycomponents.ValidatingTextFieldComponentInfo`
- JSP の `TagHandler` クラス - `mycomponents.ValidatingTextFieldTag`
- Validator インタフェース - `mycomponents.Validator`
- Validator インタフェースの実装 - `mycomponents.TypeValidator`

新しい JSP タグライブラリ (mycomponents.tld) も定義します。

最後に mycomponents の `complib.xml` を編集して、新しいコンポーネントとタグライブラリ、ConfigurableBean を Web アプリケーションフレームワークのコンポーネントライブラリに追加します。

Validator インタフェースの作成

1. **任意の Java エディタで `mycomponents.Validator` クラスを作成します。**
2. **非常に単純な妥当性検査 API を定義します。**

   デザイン原則に関するヒント：Validator のインタフェースデザインでは、Web アプリケーションフレームワークコンポーネントモデルの ConfigurableBean の能力を利用する準備をします。具体的には、この後で ValidatingTextField プロパティを「Validator」型に定義します。すると、IDE では、アプリケーション開発者は、そのインタフェースを実装する ConfigurableBean 型の動的一覧から選択できるようになります。また、サン以外のコンポーネント作成者は、同じインタフェースの追加の ConfigurableBean 実装を作成、配布することによって、このコンポーネントの枠組みを補強することができます。

   これまでの手順を終えると、`mycomponents/Validator.java` は以下のようになります。

```
package mycomponents;


/**
 *
 * @author component-author
 */
public interface Validator {


        /**
         *
         *
         */
        public abstract boolean validate(Object value);
}
```

Validator インタフェースの実装の作成

1. 任意の Java エディタで mycomponents.TypeValidator クラスを作成します。
2. ValidationRule という String 型プロパティを追加します。
3. Validator インタフェースを実装します。

   これまでの手順を終えると、mycomponents/TypeValidator.java は以下のようになります。

```
package mycomponents;
import com.iplanet.jato.model.*;
import com.iplanet.jato.util.*;


public class TypeValidator implements Validator
{

     public TypeValidator()
     {
          super();
     }

     public String getValidationRule()
     {
          return rule;
```

```
    }

    public void setValidationRule(String value)
    {
        rule=value;
    }

    public boolean validate(Object value)
    {
        if (getValidationRule()==null)
            throw new ValidationException("No validation rule has been set");

        try
        {
            value=TypeConverter.asType(getValidationRule(),value);
        }
        catch (Exception e)
        {
            return false;
        }

        return true;
    }
    ////////////////////////////////////////////////////////////////////////
    // インスタンスの変数
    ////////////////////////////////////////////////////////////////////////

    private String rule;
}
```

デザインのヒント：上記の TypeValidator の初歩的な実装は、単純な String 型プロパティとして ValidationRule を表示します。このままの状態では、IDE は、デフォルトの文字列エディタの編集対象としてこのプロパティを表示します。このため、アプリケーション開発者は、明示的にプロパティの値を「java.lang.String」か「java.lang.Integer」、「java.lang.Float」に設定する必要があります。これは、あまり親切なユーザーインタフェースではありません。この ValidationRule は ConfigurableBean に分類されるため、コンポーネント作成者は完全な JavaBean コンポーネントモデルを利用して、ユーザーインタフェースを改良することができます。理想的にはコンポーネント作成者が、このプロパティ用の独自のプロパティエディタのデザインと配備もします。この場合は、単純なドロップダウン形式のリストのプロパティエディタでも、デフォルトの文字列エディタより大きな改善になります。これでコンポーネント作成者は、選択用の独自のプロパティエディタを指定する TypeValidatorBeanInfo を作成できます。詳細は、第 8 章「デザインアクション」を参照してください。

Web アプリケーションフレームワークコンポーネントクラスの作成

1. 任意の Java エディタで `mycomponents.ValidatingDisplayField` クラスを作成します。
2. ValidatingDisplayField に `com.iplanet.jato.view.BasicDisplayField` を拡張させます。
3. コンポーネントの種類に合わせて適切なコンストラクタを実装します。

   表示フィールドコンポーネントはすべて、View 型 `parent`、String 型 `name` を取る、2 つの引数のコンストラクタを実装している必要があります。IDE は、すべての表示フィールドコンポーネントがこのコンストラクタを実装していることを前提にしています。
4. 「Validator」プロパティ用の取得および設定メソッドを追加します。
5. ValidationFailureMessage 用の取得および設定メソッドを追加します。
6. 求められている要件を満たすために必要な残りのメソッドを実装します。
   - 有効 / 無効状態を示すフラグ
   - 再表示用に無効な値を保持するバッファ
   - Validator を起動する setValue のオーバーライド実装
   - バッファリングされた無効な値を条件付きで返す getValue のオーバーライド実装

   これまでの手順を終えると、`mycomponents/ValidatingDisplayField.java` は以下のようになります。

```
package mycomponents;
import com.iplanet.jato.view.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.util.*;


public class ValidatingDisplayField extends BasicDisplayField {


    public ValidatingDisplayField(View parent, String name) {
        super(parent, name);
    }

    public Validator getValidator()
    {
        return validator;
    }
```

```
    public void setValidator(Validator value)
    {
        validator=value;
    }

    public String getValidationFailureMessage()
    {
        return validationFailureMessage;
    }

    public void setValidationFailureMessage(String value)
    {
        validationFailureMessage=value;
    }

    public boolean isValid()
    {
             return isValid;
    }

    public void setValid(boolean value)
    {
             isValid = value;
    }

    ////////////////////////////////////////////////////////////////////////
    // 値のメソッド
    ////////////////////////////////////////////////////////////////////////

    public Object getValue()
    {
        if (!isValid())
             return getInvalidValue();
        else
             return super.getValue();
    }

    public Object getInvalidValue()
    {
        if (invalidValue !=null)
             return invalidValue;
        else
             return null;
    }

    public void setValue(Object value)
    {
        if (value!=null && getValidator()!=null)
        {
             if (getValidator().validate(value))
             {
```

```
                    try
                    {
                         super.setValue(value);
                         setValid(true);
                    }
                    catch (ValidationException e)
                    {
                         setValid(false);
                         invalidValue=value;
                         setValidationFailureMessage("Exception setting value \""+
                              "on model: "+ e.getMessage());
                    }
               }
               else
               {
                    setValid(false);
                    invalidValue=value;
               }
          }
          else
               super.setValue(value);
     }


     ////////////////////////////////////////////////////////////////////////
     // インスタンスの変数
     ////////////////////////////////////////////////////////////////////////

     private Validator validator;
     private String validationFailureMessage;
     private boolean isValid = true;

     private Object invalidValue;
}
```

カスタム JSP TagHandler クラスの作成

要件では、ValidatingComponent クラスはその検査エラーメッセージを表示する必要があります。このための 1 つの方法、つまり、ここで追求する方法は、新しいコンポーネントとカスタム JSP TagHandler クラスを対にすることです。この対によって、コンポーネントを完全に制御することができます。

1. 任意の Java エディタで mycomponents.ValidatingTextFieldTag クラスを作成します。

2. com.iplanet.jato.taglib.html.TextFieldTag からこのクラスを拡張します。

3. doEndTag メソッドをオーバーライドして、コンポーネントが無効の場合に必ず条件付きで検査エラーメッセージを付加するようにします。

   これまでの手順を終えると、mycomponents/ValidatingTextFieldTag.java は以下のようになります。

```
package mycomponents;
import com.iplanet.jato.util.*;
import javax.servlet.jsp.*;
import com.iplanet.jato.taglib.html.*;
import com.iplanet.jato.util.*;
import com.iplanet.jato.view.*;
public class ValidatingTextFieldTag extends TextFieldTag
{

     public ValidatingTextFieldTag()
     {
          super();
     }


     public int doEndTag()
          throws JspException
     {
          int result=super.doEndTag();

          ContainerView parentContainer=getParentContainerView();
          View child=parentContainer.getChild(getName());
          checkChildType(child,ValidatingDisplayField.class);

          ValidatingDisplayField field=(ValidatingDisplayField)child;
          // フィールドが有効な場合は何もしない
          if (field.isValid())
               return result;

          // 赤で検査エラーメッセージを付加する
          NonSyncStringBuffer buffer=new NonSyncStringBuffer(
               " <font color=\"#FF0000\">");
          buffer.append(field.getValidationFailureMessage());
          buffer.append("</font>");
          writeOutput(buffer);
          return result;
     }
}
```

注 – Web アプリケーションフレームワークのコンポーネントモデルは、コンポーネント作成者が任意の 1 つのコンポーネントに対して複数の JSP TagHandler を指定することを可能にします。詳細は、JspTagDescriptor API を参照してください。

ComponentInfo クラスの作成

`ComponentInfo` クラスは、IDE がコンポーネントを組み込むのに必要なデザイン時メタデータを定義します。この例では、既存の `ComponentInfo` 拡張し、本当のオブジェクト指向のスタイルで単にこのクラスを補強します。当然、ComponentInfo インタフェースを最初から実装する方法を選ぶこともできますが、この例では生産的ではありません。

この例では、最初のコンポーネント例で明らかにした機能をさらに発展させます。つまり、ViewComponentInfo インタフェースがメタデータで提供する重要な機会、すなわち、特定のコンポーネントに対して JSP タグを記述する機能を利用します。

1. `mycomponents.ValidatingTextFieldComponentInfo` クラスを作成します。
2. `ValidatingTextFieldComponentInfo` クラスに `com.iplanet.jato.view.html2.TextFieldComponentInfo` を拡張させます。
3. 引数なしのコンストラクタを実装します。
4. コンポーネントの基本的なデザイン時説明を提供する `getComponentDescriptor()` メソッドを実行します。
5. IDE で表示させるプロパティを識別する `getConfigPropertyDescriptors()` メソッドを実装します。
   - `Validator` プロパティ用の `ConfigPropertyDescriptor` を追加
   - `ValidationFailureMessage` プロパティ用の `ConfigPropertyDescriptor` を追加
6. このコンポーネントのインスタンスが ViewBean またはコンテナビューに追加されるたびに、IDE で、関連付けられている JSP に自動的に追加させる JSP タグを指定する `getJspTagDescriptors()` メソッドを実装します。

   これまでの手順を終えると、`mycomponents/ValidatingTextFieldComponentInfo.java` は以下のようなコードになります。

注 – このコードでは、例として紹介しやすいよう、String 値を直接埋め込んでいます。表示文字列を各言語対応にする必要があると思われる場合は、リソースバンドルを利用してください。

```
package mycomponents;
import java.beans.*;
import java.util.*;
import com.iplanet.jato.component.*;
import com.iplanet.jato.taglib.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.html2.*;


public class ValidatingTextFieldComponentInfo extends TextFieldComponentInfo {

    public ValidatingTextFieldComponentInfo() {
        super();
    }

    public ComponentDescriptor getComponentDescriptor()
    {
        // コンポーネントクラスを示す
        ComponentDescriptor result=new ComponentDescriptor(
            "mycomponents.ValidatingDisplayField");

        // この名前が、コンポーネントのインスタンスの名前を決めるのに使用される
        result.setName("ValidatingTextField");

        // この表示名が、コンポーネント選択で使用される
        result.setDisplayName("ValidatingTextField Component");

        // この説明がコンポーネントのツールチップのテキストになる
        result.setShortDescription("A simple validating text field component");

        return result;
    }

    public ConfigPropertyDescriptor[] getConfigPropertyDescriptors()
    {
        if (configPropertyDescriptors!=null)
            return configPropertyDescriptors;

        // スーパークラスに定義されているあらゆるプロパティを取得
        configPropertyDescriptors=super.getConfigPropertyDescriptors();
        List descriptors=new LinkedList(Arrays.asList(configPropertyDescriptors));

        ConfigPropertyDescriptor descriptor = null;
```

```
            descriptor=new ConfigPropertyDescriptor(
                 "validator",Validator.class);
            descriptor.setDisplayName("Validator");
            descriptor.setHidden(false);
            descriptor.setExpert(false);
            descriptors.add(descriptor);

            descriptor=new ConfigPropertyDescriptor(
                 "validationFailureMessage",String.class);
            descriptor.setDisplayName("Validation Failure Message");
            descriptor.setHidden(false);
            descriptor.setExpert(false);
            descriptors.add(descriptor);

            // 配列を作成 / 返す
            configPropertyDescriptors = (ConfigPropertyDescriptor[])
                 descriptors.toArray(
                      new ConfigPropertyDescriptor[descriptors.size()]);
            return configPropertyDescriptors;
       }

       public JspTagDescriptor[] getJspTagDescriptors()
       {
            JspTagAttributeDescriptor[] attrs=new JspTagAttributeDescriptor[1];
            attrs[0]=new JspTagAttributeDescriptor(
                 TagBase.ATTR_NAME,JspTagDescriptor.ASSUMED_PROPERTY_NAME,null);

            JspTagDescriptor htmlTagDescriptor=new JspTagDescriptor(
                 JspTagDescriptor.ENCODING_HTML,"validatingTextField",
                 "/WEB-INF/mycomplib.tld",attrs);

            return new JspTagDescriptor[] {htmlTagDescriptor};
       }

       private ConfigPropertyDescriptor[] configPropertyDescriptors;
  }
```

新しいタグライブラリの TLD ファイルの作成

新しい JSP TagHandler を定義したため、続いて、コンポーネントライブラリに対して JSP ライブラリ TLD ファイルを作成する必要があります。

カスタム JSP ライブラリには、1 つの「弱い」制限があります。IDE の実行中、使用中のすべての JSP ファイルに対する論理的なオブジェクトモデルが 1 つ作成されます。この JSP オブジェクトモデルは、ビューが変化しているときの JSP 内のタグの配置を管理する目的で、ページおよびページレットビューコンポーネント機構によって使用されます。JSP オブジェクトモデルを作成するための JSP ファイルの構文解析中、Web アプリケーションフレームワークコンポーネントのタグライブラリ用のタ

グには、特殊な処理が施されます。カスタム JSP タグライブラリに、Web アプリケーションフレームワークのビューコンポーネントに関係のない追加のタグがある場合、それらのタグは、JSP オブジェクトモデル内で適切に分類されない可能性があります。このため、Web アプリケーションフレームワーク関連のタグはその専用のタグライブラリに分離しておくべきです。JSP オブジェクトモデルの内部では、コンポーネントライブラリのマニフェストに指定されている、タグライブラリ内のタグは、「JATO」タグに分類され、JSP ファイル内の他のすべてのタグは「その他」のタグに分類されます。これが「弱い」制限である理由は、非 Web アプリケーションフレームワークタグがビューコンポーネントタグの管理の妨げになる極端なケースが存在するためです。非 Web アプリケーションフレームワークタグがコンポーネントタグライブラリ内に残っていて、そのタグの「name」属性値が本当の Web アプリケーションフレームワークタグの「name」属性と衝突する場合、JSP オブジェクトモデルは正しく動作しない可能性があります。言い替えれば、「name」属性を持つ非 Web アプリケーションフレームワークタグがある場合は、それらのタグを別のタグライブラリに分離して、極端なケースの問題を回避するようにしてください。

ライブラリの TLD ファイル名は任意です。ライブラリ内のファイルの場所も任意です。この後の手順では、コンポーネントマニフェスト内で新しい TLD ファイルを宣言します。JSP の tld ファイルの完全な説明を行うことは、このガイドでは取り扱いません。この例の場合は、タグ要素を 1 つ (validatingTextField) 含む新しいライブラリを 1 つ (mycomlib) だけ宣言する必要があるということだけ説明しておきます。タグ属性はすべて、文字通り、Web アプリケーションフレームワークコンポーネントライブラリの `jato.tld` ファイルの宣言からコピーすることができます。`jato.tld` ファイルは、IDE によって作成された任意の Web アプリケーションフレームワークアプリケーションの `WEB-INF\tld\com_iplanet_jato` ディレクトリにあります。

1. `mycomponents/mycomplib.tld` ファイルを作成します。
2. 新しいタグライブラリを宣言する基本 tld 情報を追加します。
3. 新しいタグの validatingTextField 用のタグ要素とそれに対応する tag-class `mycomponents.ValidatingTextFieldTag` を追加します。
4. 適切なタグ属性をすべて追加することによって、タグ要素を完成します。Web アプリケーションフレームワークコンポーネントライブラリの TextField タグ用にすでに定義されている、`jato.tld` 内のタグ属性をコピーしてください。

   これまでの手順を終えると、`mycomponents/mycomplib.tld` は以下のようになります。

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

<!DOCTYPE taglib
          PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
          "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<!-- テンプレートのテキスト -->

<taglib>
     <tlib-version>1.0</tlib-version>
     <jsp-version>1.2</jsp-version>
     <short-name>mycomponents.mycomplib</short-name>
     <display-name>mycomponents.mycomplib</display-name>
     <tag>
          <name>validatingTextField</name>
          <tag-class>mycomponents.ValidatingTextFieldTag</tag-class>
          <body-content>empty</body-content>
          <display-name>Validating Text Field</display-name>
          <description></description>
          <attribute>
               <name>name</name>
               <required>true</required>
               <rtexprvalue>false</rtexprvalue>
                    <type>String</type>
          </attribute>
...
          <!-- さらに属性の定義が続く -->
...

     </tag>
</taglib>
```

コンポーネントライブラリのマニフェストの補強

コンポーネントマニフェストは、前述の例ですでに作成しています。ここでは、追加の情報を追加します。

前の例では現れなかった追加の種類の情報を追加することに注意してください。

Web アプリケーションフレームワークのライブラリマニフェスト名は、`complib.xml` である必要があります。Web アプリケーションフレームワークのライブラリマニフェストは、JAR ファイル内の `/COMP-INF` ディレクトリに入れる必要があります。

1. `COMP-INF/complib.xml` ファイルを作成するか、開きます。
2. ValidatingTextField コンポーネントを宣言するコンポーネント要素を追加します。
3. `mycomponents.TypeValidator` を宣言する ConfigurableBean 要素を追加します。
4. `mycomponents.tld` を宣言する taglib 要素を追加します。

これまでの手順を終えると、`COMP-INF/complib.xml` は以下のようになります。

注 – XML ドキュメントとして tld が適切な書式であることを確認してください。先頭の XML タグの前に不適切な先行空白があるなどのちょっとしたことでも、不適切な書式のドキュメントになることがあります。tld ファイルの XML 書式が不適切であると、いくつかのサーブレットコンテナが Web アプリケーション全体を読み込めなくなります。そうしたエラーは追跡するのが難しいことがあります。

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
<tool-info>
<tool-version>2.1.0</tool-version>
</tool-info>
<library-name>mycomponents</library-name>
<display-name>My First Component Library</display-name>
<!-- ここにアイコン
<icon>
<small-icon>/com/iplanet/jato/resources/complib.gif</small-icon>
</icon>
-->
<interface-version>1.0.0</interface-version>
<implementation-version>20030221</implementation-version>

<component>
<component-class>mycomponents.MyTextField</component-class>
<component-info-class>mycomponents.MyTextFieldComponentInfo</component-info-class>
</component>
<component>
<component-class>mycomponents.ValidatingDisplayField</component-class>
<component-info-class>mycomponents.ValidatingTextFieldComponentInfo</component-info-class>
</component>

<configurable-bean>
<bean-class>mycomponents.TypeValidator</bean-class>
</configurable-bean>

<taglib>
<taglib-uri>/WEB-INF/mycomplib.tld</taglib-uri>
<taglib-resource>/mycomponents/mycomplib.tld</taglib-resource>
<taglib-default-prefix>mycomp</taglib-default-prefix>
</taglib>

</component-library>
```

コンポーネントライブラリの JAR ファイルの再作成

最初の例で行ったようにすべてのコンポーネントクラスを JAR ファイルにして、それらのクラスをライブラリとして配布できるようにします。

1. **この JAR ファイルの名前は任意です。**

   この場合は、「mycomponents.jar」という名前にします。

2. **必ずしも Java ソースファイルを JAR に含める必要はありません。**

3. **アイコンイメージやリソースバンドルなどの必要な補助リソースを JAR に含めます。**

   この例では、そうしたリソースはありません。

   この例では、いくつかの新しいクラスと新しい JSP タグライブラリ (1 つ) を含めます。

4. **mycomponents.jar の内部構造は以下のようになります。**

```
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
mycomponents/TypeValidator.class
mycomponents/ValidatingDisplayField.class
mycomponents/ValidatingTextFieldComponentInfo.class
mycomponents/ValidatingTextFieldTag.class
mycomponents/Validator.class
mycomponents/mycomplib.tld
COMP-INF/complib.xml
```

新しいコンポーネントのテスト

これで、ライブラリをテストし、配布する準備ができました。プロジェクトの例でテストしてください。この段階では、Web アプリケーションフレームワークモジュールがインストールされ、使用可能になっている IDE を使用する必要があります。IDE で Web アプリケーションフレームワークアプリケーションを構築した経験がない場合は、次に進む前に、Web アプリケーションフレームワークのマニュアルセットに含まれている『Web アプリケーションフレームワーク チュートリアル』を先に終了してください。

1. 以前に作成したテスト用アプリケーションに新しいバージョンのライブラリを配備します。

   IDE に関する重要な注：IDE で、現在マウントされている JAR ファイルを削除したり、上書きコピーしたりすることはできません。対話形式でコンポーネントライブラリを開発し、そのライブラリをテスト用アプリケーションでテストする際は、このことが難題になります。

   テスト用アプリケーション内で同じライブラリ JAR ファイルの新しいバージョンを繰り返しテストするには、以下の手順を実行します。

   a. テスト用アプリケーションをマウント解除します。

   b. マウント解除が完了したら、オペレーティングシステムのファイルシステムに移動し、マウント解除したテスト用アプリケーションの `WEB-INF/lib` ディレクトリにある古いライブラリ JAR ファイルに新しいライブラリ JAR ファイルを上書きコピーします。

   c. テスト用アプリケーションを再マウントします。

      これで、テスト用アプリケーションが、新しいバージョンのライブラリを選択するようになります。

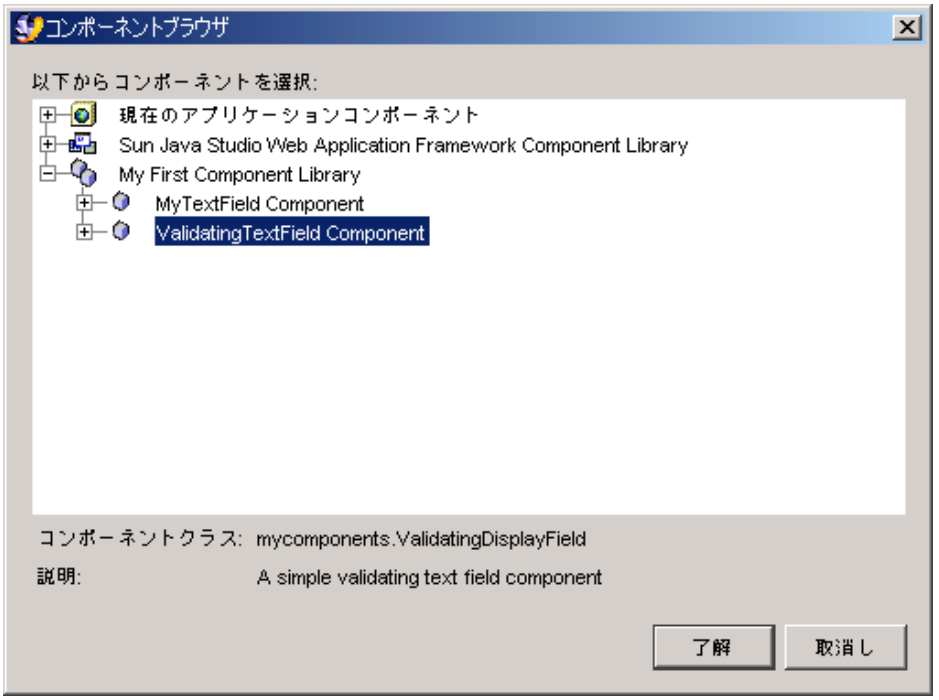
      通常、この手順で問題が起きることはありません。新しい JAR を古い JAR に上書きできないか、テスト用アプリケーションを正しく再マウントできないエラーが発生した場合は、IDE を再起動します。

2. 以前に作成した Page1 オブジェクトを選択します。

3. Page1 に「ValidatingTextFieldComponent 」のインスタンスを追加します。

   コンポーネントパレットからコンポーネントを選択しても、Page1 の「可視コンポーネント」サブノードを選択してから、右クリックし、ポップアップメニューから「可視コンポーネント 追加」アクションを選択してもかまいません。


コンポーネントパレット
可視コンポーネント
時刻表示
日時表示
ページ移動 (リンク)
マスク付きテキストフィールド
メニュー
静的パンくずリスト
データセットナビゲータ
データセットロケータ
My First Component Library
MyTextField Component
ValidatingTextField Component
アプリケーション可視コンポーネント
不可視コンポーネント
選択したオブジェクトに追加するコンポーネントを...



4. 一覧から「ValidatingTextField Component」を選択します。

   「validatingTextField1」という名前の子ビューがページに追加されることを確認します。

5. validatingTextField1 可視コンポーネントノードを選択します。

   IDE のプロパティシートに、継承されたテキストフィールドコンポーネントのプロパティの他に、「Validator」および「Validation Failure Message」という独自のプロパティが表示されていることを確認します。

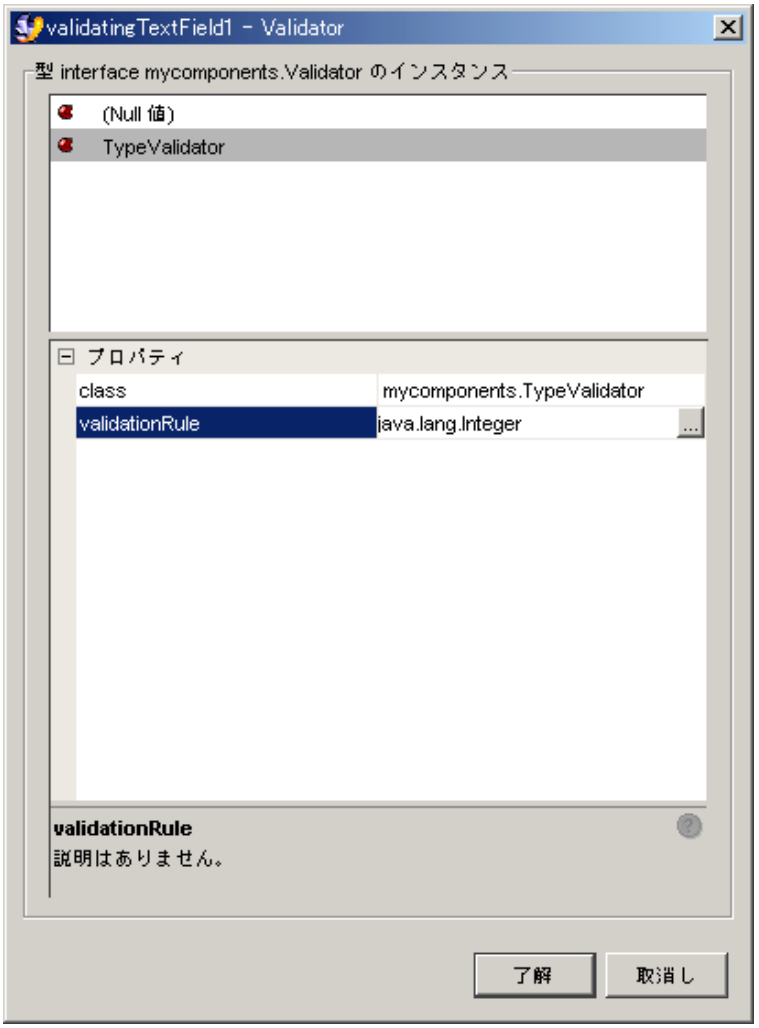
6. 「Validation Failure Message」プロパティを編集します。

   このプロパティに「This is a test failure message」(またはその他の任意のテキスト)を設定します。

7. 「Validator」プロパティを編集します。

   次の専用の ConfigurableBean エディタが表示されます。テスト用に「validationRule」プロパティに「java.lang.Integer」を設定します。

   「validationRule」プロパティには、必ず完全限定のクラス名を指定してください。「Integer」だけだと、実行時に正しく評価されません。「java.lang.Integer」というように完全限定する必要があります。



8. Page の Java ファイルにコードが生成されていることを確認します。

   createChildReserved メソッド内に、次に示すようなコードが含まれているはずです (実際のインデントは異なる可能性があります)。

```
...
else if (name.equals(CHILD_VALIDATING_TEXT_FIELD1)) {
mycomponents.ValidatingDisplayField child =
new mycomponents.ValidatingDisplayField(this, CHILD_VALIDATING_TEXT_FIELD1);
mycomponents.TypeValidator validatorVar =
new mycomponents.TypeValidator();

{ // ローカル変数のスコープの開始
validatorVar.setValidationRule("java.lang.Integer");
{ // ローカル変数のスコープの終了
child.setValidator(validatorVar);
child.setValidationFailureMessage( "This is a test failure message");
return child;
}
...
```

9. 関係する JSP ファイルを開いて、validatingTextField タグが含まれていることを確認します。

   自動的に mycomplib.tld が取り込まれていることにも注意してください (テスト用アプリケーションの全体の内容は、テスト用のページに validatingTextField1 の他に他の子ビューを加えたかどうかによって、下記と異なることがあります)。

```
<%@page contentType="text/html; charset=ISO-8859-1" info="Page1" language="java"%>
<%@taglib uri="/WEB-INF/jato.tld" prefix="jato"%>
<%@taglib uri="/WEB-INF/mycomplib.tld" prefix="mycomp"%>

<jato:useViewBean className="testmycomplib.main.Page1">

<html>
<head>
<title>Page1</title>
</head>
<body>
<jato:form name="Page1" method="post">
<jato:textField name="myTextField1"/>
<mycomp:validatingTextField name="validatingTextField1"/>
</jato:form>
</body>
</html>

</jato:useViewBean>
```

10. Page1 を効果的にテスト実行するには、ボタンと要求処理コードを追加する必要があります。

テストとして、ボタンを追加し、要求処理コード (送付後にページを再表示する) を追加するには、以下に示す手順を実行します。そうすることで、ValidatingTextComponent がデザイン通りに動作しているかどうかを確認することができます。まだ行っていない場合は、次の手順でボタンおよび要求処理コードを追加してください。

注 – 以下の手順は、Web アプリケーションフレームワークアプリケーションの従来型の開発方法を示しています。このガイドでは、この従来型の方法を詳細には説明しません。効果的なテスト用のページを作成するには、この手順または類似の手順が必要になります。

---

a. Web アプリケーションフレームワークライブラリの基本ボタンのインスタンスを Page1 に追加します。

コンポーネントパレットからコンポーネントを選択しても、Page1 の「可視コンポーネント」サブノードを選択してから、右クリックし、ポップアップメニューから「可視コンポーネント 追加」アクションを選択してもかまいません。

テスト用の ViewBean に子「button1」が追加されます。

b. 「button1」ビジュアルコンポーネントノードを選択します。

c. 右クリックし、ポップアップメニューから「イベント」>「handleRequest」アクションを選択します。

これで ViewBean の Java ファイルに、`handleButton1Request` というイベントハンドラメソッドが追加されます。

このテストの場合、`handleButton1Request` の本体を変更する必要はありません。本体は自動的に生成されて、現在のページを再表示するようになっており、これがこのテストの目的です。

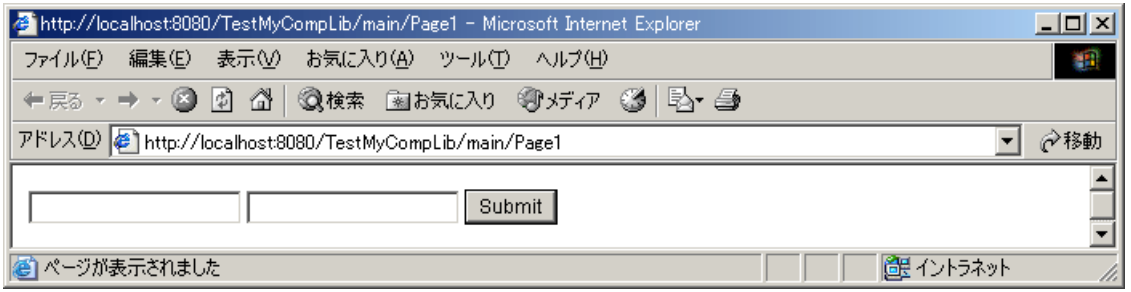
要求ハンドラが次のようになっていることを確認します。

```
public void handleButton1Request(RequestInvocationEvent event) throws Exception {
getParentViewBean().forwardTo(getRequestContext());
}
```

11. Page1 をテスト実行します。

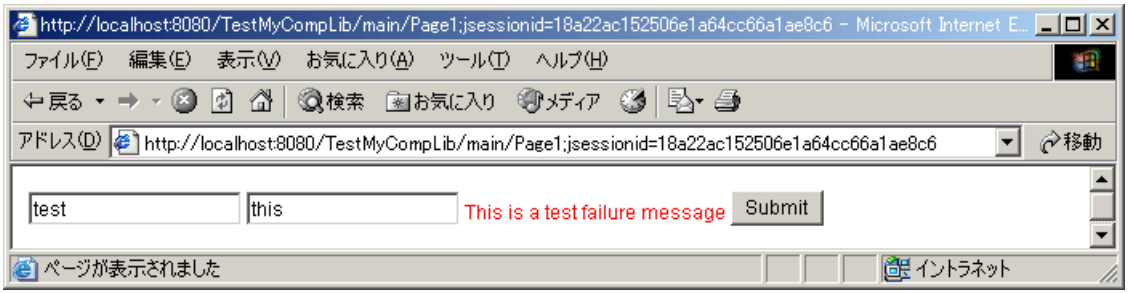
Web アプリケーションフレームワークの ViewBean のテスト実行方法が分からない場合は、『Web アプリケーションフレームワーク チュートリアル』を参照してください。

ブラウザに、以下のような Page1 の出力が表示されます (今回はテキストフィールド 2 つと MyTextField のインスタンス 1 つ、ValidatingTextField のインスタンス 1 つが含まれている)。



12. ValidatingTextField のテキスト入力フィールドに不正な値 (たとえば整数以外の値) を入力し、ページを送付します。

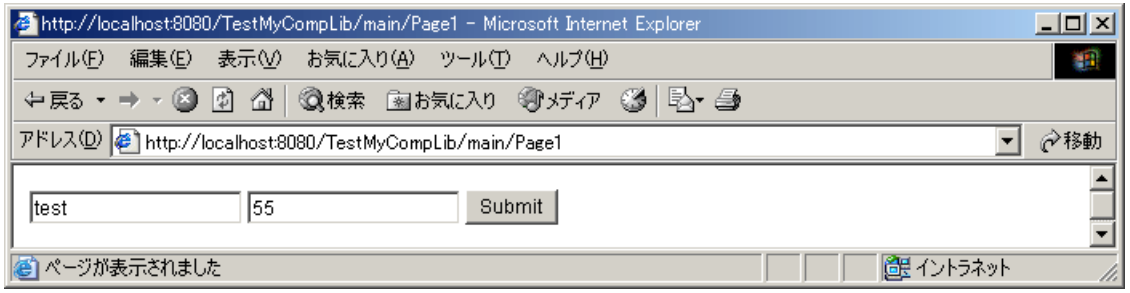
    ValidatingTextField のすぐ後に「Validation Failure Message」プロパティのテキストの入ったページがただちに再表示されます。



13. 有効な値 (たとえば 55 かその他の有効な整数) を入力して、ページを送付します。

    Validation Failure Message テキストの入っていないページが再表示されます。

    依然として Validation Failure Message のテキストが表示される場合は、前に戻って、validatingTextField1 の「Validator」>「ValidationRule」プロパティの値を、「java.lang.Integer」(単に「Integer」ではない) に設定します。



配布

コンポーネントが正しく機能すると、そのコンポーネントを配布できます。前に戻って、機能強化してもかまいません。たとえば Validator の「ValidationRule」プロパティに「java.lang.Integer」を入力するようユーザーに求めることが、多くのエラーが起きる可能性があると判断したと仮定します。その場合は少し時間を使って、カスタムプロパティエディタを開発することを推奨します。このガイドでは詳細は説明しませんが、基本的な JavaBean のリファレンスに記載されています。

拡張可能ビューコンポーネントの開発

この節では、初歩的なページレベルのセキュリティー機能をサポートする新しい ViewBean コンポーネントの作成方法を説明します。簡潔にするため、セキュリティー機能のデザインと実装は最低限のものに留めます。この練習問題は、拡張可能ビューコンポーネントデザインの仕組みを理解することを目的としているため、可能なセキュリティーモデルについては表面をなぞるだけにします。この節を終えると、Web アプリケーションフレームワークにおける拡張可能コンポーネントの役割をきちんと理解できるようになります。この例で実装するいくつかの機能は任意であることを忘れないでください。この例は、拡張可能ビューコンポーネントの作成で求められる必要最低限というわけではありません。

この例では、Web アプリケーションフレームワークコンポーネントモデルに関係する以下のような項目を取り上げます。

- ExtensibleComponentInfo
- コンポーネント提供の Java テンプレート
- IndexedConfigPropertyDescriptor
- EventHandlerDescriptor

アプリケーションおよびページレベルのセキュリティーの分野では開発者の意向が広範囲に異なるため、基本的なデザイン原則として、Web アプリケーションフレームワークでは、セキュリティーの分野では、規定するというより実現しうるという方向を取っています。この例では、Web アプリケーションフレームワークがページレベルの任意のセキュリティーモデルを簡単に実現可能であることを実際に証明します。ただし、この例が最終的または推奨する実装であるわけではありません。

セキュリティー保護された ViewBean コンポーネントには、次のデザイン時機能を持たせます。

- 「RequiredTokens」という索引付きプロパティを表示する。アプリケーション開発者は、「必須」String トークン (たとえば現在のページにアクセスするのに必要なトークン) の任意の一覧が指定されるように、このプロパティを構成します。

- 「GrantTokens」という索引付きプロパティを表示する。アプリケーション開発者は、「付与」String トークン (たとえば現在のページへのアクセスに成功したユーザーに付与されるトークン) の任意の一覧が指定されるように、このプロパティを構成します。
- カスタム実装用に「handleMissingTokens」イベントハンドラを表示する。このことは、IDE 開発者が「イベント」ポップアップメニューから「handleMissingTokens」を選択でき、IDE が、セキュリティー保護された現在の ViewBean の Java ファイルに自動的にそのイベントハンドラを挿入することを意味します。これは、Web アプリケーションフレームワークコンポーネントモデルの高度なオプション機能の 1 つです。

セキュリティー保護された ViewBean コンポーネントには、次の実行時機能を持たせます。

- セキュリティー保護された ViewBean へのアクセスに成功したユーザーにトークンを「付与」する。つまり、アプリケーションのユーザーは、アプリケーションを通る過程でトークンを「蓄積」します。
- 必須トークンとユーザー蓄積トークンの単純比較で自身への実行時アクセスを制限する。アプリケーションのユーザーが必須トークンの一部でも蓄積していない場合は、MissingTokensEvent がトリガーされ、handleMissingTokens という特定のイベントハンドラメソッドが起動されます。また、セキュリティー保護された ViewBean ベースクラスに実装した handleMissingTokens は SecurityCheckException をスローします。Web アプリケーションフレームワークは、キャッチされなかったあらゆる例外に対して行うように、キャッチされなかった SecurityCheckException を自動的に処理します (たとえば標準の Web アプリケーションフレームワークエラーページをユーザーに返す)。セキュリティー保護された ViewBean は handleMissingTokens のこの実装を無効にして、コンテキストに固有の任意の動作を行うことができます。

この実行時モデルは、アプリケーション開発者によって、付与トークンと必須トークンがセキュリティー保護された ViewBean 単位で指定されることを前提にしています。

セキュリティー保護されたこの ViewBean の実装は、実行時に蓄積されたトークンを記録し、上記のセキュリティーモデルを適用します。また、特殊な HttpSession 属性にユーザーごとの蓄積トークンを格納します。

SecurityCheckException をスローするベースクラス版の handleMissingTokens を実装するかどうかは、完全に任意です。よりユーザーに親切なエラーページ、またはコンポーネント作成者の意向で他のものに制御を移すメソッドを実装する方法もあります。厳密に簡潔さと簡単さを考えると、SecurityCheckException をスローするというのが選択肢になります。

これらの要件を満たすには、次のクラスをデザイン、実装します。

- コンポーネントクラス - *mycomponents.SecureViewBean*
- ComponentInfo クラス - `mycomponents.SecureViewBeanComponentInfo`
- 単純なイベントクラス - `mycomponents.MissingTokensEvent`

また、ここでは、SecureViewBean のアプリケーションに固有のサブタイプの土台として IDE が使用するカスタム Java テンプレートも実装します。

最後に mycomponents の `complib.xml` を編集して、その新しいコンポーネントを Web アプリケーションフレームワークのコンポーネントライブラリに追加します。

MissingTokensEvent クラスの作成

1. **任意の Java エディタで `mycomponents.MissingTokensEvent` クラスを作成します。**
2. **イベントハンドラがどのトークンがないかを検出するための非常に単純な API を定義します。**

   これまでの手順を終えると、`mycomponents/MissingTokensEvent.java` は以下のようになります。

```
package mycomponents;
  import java.util.*;

  public class MissingTokensEvent extends Object {

    public MissingTokensEvent(List tokens) {
         missingTokens = new ArrayList(tokens);
    }

    public String toString() {
         Iterator iter = missingTokens.iterator();
         StringBuffer buff = new StringBuffer();
         buff.append("MissingToken count[" + missingTokens.size() + "] ");
         while (iter.hasNext()) {
              buff.append("Token[" + (String)iter.next() + "] ");
         }
         return buff.toString();
    }

    public ArrayList getMissingTokens() {
         return missingTokens;
    }

    private ArrayList missingTokens = null;

}
```

Web アプリケーションフレームワークコンポーネントクラスの作成

1. 任意の Java エディタで `mycomponents.SecureViewBean` クラスを作成します。
2. SecureViewBean に `com.iplanet.jato.view.BasicViewBean` を拡張させます。
3. コンポーネントの種類に合わせて適切なコンストラクタを実装します。

   あらゆる ViewBean コンポーネントは、引数なしのコンストラクタを実装する必要があります。
4. 「RequiredTokens」というプロパティ用の `get` メソッドと `set` メソッドを追加します。
5. 「GrantTokens」というプロパティ用の `get` メソッドと `set` メソッドを追加します。
6. コンポーネントに固有の要件を満たすために必要な残りのメソッドを実装します。
   - securityCheck メソッドのオーバーライド実装 - コンポーネントのページセキュリティーモデルを適用します。
   - コンポーネントの handleMissingTokens メソッドのデフォルト実装

   これまでの手順を終えると、`mycomponents/SecureViewBean.java` は以下のようになります。

```
package mycomponents;
import java.util.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.*;

public class SecureViewBean extends BasicViewBean {

     public SecureViewBean()
     {
          super();
     }

     public String[] getRequiredTokens()
     {
          return requiredTokens;
     }

     public void setRequiredTokens(String[] value)
     {
           requiredTokens = value;
     }
```

```
public String[] getGrantTokens()
{
     return grantTokens;
}

public void setGrantTokens(String[] value)
{
      grantTokens = value;
}

public void securityCheck() throws SecurityCheckException
{
     super.securityCheck();

     // 蓄積されたトークンをセッションから取得
     HashSet accumulated = (HashSet)
          getSession().getAttribute("AccumulatedTokens");
     // 防御的に蓄積コレクションを準備
     if(accumulated == null)
          accumulated = new HashSet();

     // 必要なトークンが存在するかどうかを確認
     if(requiredTokens.length > 0) {
          // 必要なトークンの有無を確認
          List missingTokens = new ArrayList();
          for(int i=0; i<requiredTokens.length; i++)
          {
               if(! accumulated.contains(requiredTokens[i]))
                    missingTokens.add(requiredTokens[i]);
          }

          if(missingTokens.size() > 0)
               handleMissingTokens(new MissingTokensEvent(missingTokens));
     }

     // ここで現在の付与トークンを蓄積トークンに追加
     // 予想されるように、handleMissingTokens によって例外がスローされた場合は
     // この地点に到達しないことに注意
     for(int i=0; i<grantTokens.length; i++)
     {
               accumulated.add(grantTokens[i]);
     }
     getSession().setAttribute("AccumulatedTokens", accumulated);
}

public void handleMissingTokens(MissingTokensEvent e)
     throws SecurityCheckException
{
     // このデフォルト実装は、単に従来の Web アプリケーションフレームワークの
     // SecurityCheckException 処理を起動する
     throw new SecurityCheckException(e.toString());
```

```
    }

    private String[] requiredTokens = new String[0];
    private String[] grantTokens = new String[0];

}
```

拡張可能コンポーネントの Java テンプレートの作成

拡張可能コンポーネントは、アプリケーション定義のエンティティに対するベースクラスの働きをします。このため、Web アプリケーションフレームワークコンポーネントモデルは、拡張可能コンポーネントの作成者がカスタム Java テンプレートを提供する機会を提供します。IDE では、そのコンポーネント提供のテンプレートを利用して、アプリケーション固有のサブタイプを作成します。コンポーネント作成者は、そのカスタムテンプレートを利用して、アプリケーション開発者の作業をさらに充実したものにすることができます。コンポーネント作成者は、`com.iplanet.jato.component.ExtensibleComponentInfo` に定義されている一群のテンプレートトークンを使ってコンポーネント固有の Java テンプレートを用意することができます。トークンの詳細は、ExtensibleComponent API を参照してください。

コンポーネント作成者はまた、Java テンプレート内で任意の Java 構造 (インポート文、メソッド、変数、インタフェース宣言など) を利用することもできます。このカスタムテンプレートは、最低限、拡張可能コンポーネントクラスから新しい Java クラスを拡張する保証をします。

この例のテンプレートは最低限の内容になっています。

1. **新しいディレクトリ** `mycomponents.resources` **を作成します。**
2. **任意のテキストエディタで** `mycomponents.resources.SecureViewBean_java.template` **テンプレートを作成します。**

   テンプレートの内容は、以下のようにします。

注 – a __TOKEN__ pattern の後にトークンが続きます。

```
package __PACKAGE__;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.iplanet.jato.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.event.*;
import com.iplanet.jato.model.*;
import mycomponents.*;


/**
 *
 *
 */
public class __CLASS_NAME__ extends SecureViewBean
{
     /**
      * Default constructor
      *
      */
     public __CLASS_NAME__()
     {
          super();
     }


}
```

ComponentInfo クラスの作成

ComponentInfo クラスは、IDE がコンポーネントを組み込むのに必要なデザイン時メタデータを定義します。この例では、既存の ComponentInfo を拡張し、本当のオブジェクト指向のスタイルで単にこのクラスを補強します。当然、`ComponentInfo` インタフェースを最初から実装する方法を選ぶこともできますが、この例では生産的ではありません。

この例では、前述のコンポーネント例で明らかにした機能をさらに発展させます。つまり、ExtensibleComponentInfo インタフェースがメタデータで提供する 2 つの新しい機会、すなわち、Java テンプレートを指定する機会と、拡張可能コンポーネント用のイベントハンドラメソッドを記述する機会を利用します。

1. mycomponents.SecureViewBeanComponentInfo クラスを作成します。
2. SecureViewBeanComponentInfo に com.iplanet.jato.view.BasicViewBeanComponentInfo を拡張させます。
3. 引数なしのコンストラクタを実装します。
4. コンポーネントの基本的なデザイン時説明を提供する getComponentDescriptor() メソッドを実行します。
5. IDE で表示させるプロパティを識別する getConfigPropertyDescriptors() メソッドを実装します。
   a. RequiredTokens プロパティに対する IndexedConfigPropertyDescriptor を追加します。
   b. GrantTokens プロパティに対する IndexedConfigPropertyDescriptor を追加します。
6. Java テンプレートを実装する getPrimaryTemplateAsStream() メソッドを実装します。IDE は、この拡張可能コンポーネントから派生する新しいクラスに対する出発点として、この Java テンプレートを使用します。
7. あらゆるイベントハンドラメソッドのデザイン時説明を提供する getEventHandlerDescriptors() メソッドを実装します。IDE は、この拡張可能コンポーネントから派生する新しいクラスへの自動挿入時に、この説明を表示します。

   これまでの手順を終えると、mycomponents/SecureViewBeanComponentInfo.java は以下のようになります。

   このコードでは、例として紹介しやすいよう、String 値を直接埋め込んでいます。表示文字列を各言語対応にする必要があると思われる場合は、リソースバンドルを利用してください。

```
package mycomponents;
import java.util.*;
import java.io.*;
import com.iplanet.jato.*;
import com.iplanet.jato.component.*;
import com.iplanet.jato.view.*;


public class SecureViewBeanComponentInfo extends BasicViewBeanComponentInfo
{

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

```
    public ComponentDescriptor getComponentDescriptor()
    {
         final String CLASS_NAME="mycomponents.SecureViewBean";

         ComponentDescriptor descriptor=new ComponentDescriptor(
              CLASS_NAME);
         descriptor.setName("SecurePage");
         descriptor.setDisplayName("Secure ViewBean");
         descriptor.setShortDescription(
              "A Page with a token based security model");
         return descriptor;
    }

    public ConfigPropertyDescriptor[] getConfigPropertyDescriptors()
    {
         if (configPropertyDescriptors!=null)
              return configPropertyDescriptors;

         configPropertyDescriptors=super.getConfigPropertyDescriptors();
         List descriptors=new LinkedList(Arrays.asList(configPropertyDescriptors));

         ConfigPropertyDescriptor descriptor = null;

         descriptor=new IndexedConfigPropertyDescriptor(
              "grantTokens",String.class); // NOI18N
         descriptor.setDisplayName("Grant Tokens"); // NOI18N
         descriptor.setHidden(false);
         descriptor.setExpert(false);
         descriptors.add(descriptor);

         descriptor=new IndexedConfigPropertyDescriptor(
              "requiredTokens",String.class); // NOI18N
         descriptor.setDisplayName("Required Tokens"); // NOI18N
         descriptor.setHidden(false);
         descriptor.setExpert(false);
         descriptors.add(descriptor);

         // 配列を作成 / 返す
         configPropertyDescriptors = (ConfigPropertyDescriptor[])
              descriptors.toArray(
                   new ConfigPropertyDescriptor[descriptors.size()]);
         return configPropertyDescriptors;
    }

 public String getPrimaryTemplateEncoding()
    {
/* 本稼働のバージョンは次のようにリソースバンドル駆動型
return getResourceString(getClass(),
"PROP_SecureViewBean_SOURCE_TEMPLATE_ENCODING", "ascii");
*/
```

```
          return "ascii";
     }

     public InputStream getPrimaryTemplateAsStream()
     {
/* 本稼働のバージョンは次のようにリソースバンドル駆動型

return SecureViewBeanComponentInfo.class.getClassLoader().
getResourceAsStream(
getResourceString(getClass(),
"RES_SecureViewBeanComponentInfo_SOURCE_TEMPLATE",""));
*/

          return SecureViewBeanComponentInfo.class.getResourceAsStream(
               "/mycomponents/resources/SecureViewBean_java.template");
     }


     public EventHandlerDescriptor[] getEventHandlerDescriptors()
     {
          if (eventHandlerDescriptors!=null)
               return eventHandlerDescriptors;

          eventHandlerDescriptors=super.getEventHandlerDescriptors();
          List descriptors=new LinkedList(
               Arrays.asList(eventHandlerDescriptors));

          EventHandlerDescriptor descriptor =new EventHandlerDescriptor(
               "handleMissingTokens",
               "handleMissingTokens",
               "public void handleMissingTokens(MissingTokensEvent e)" +
               "throws SecurityCheckException",
               "throw new SecurityCheckException(e.toString());",
               "");

          descriptors.add(descriptor);

          // 配列を作成 / 返す
          eventHandlerDescriptors = (EventHandlerDescriptor[])
               descriptors.toArray(
                    new EventHandlerDescriptor[descriptors.size()]);
          return eventHandlerDescriptors;
     }

     private ConfigPropertyDescriptor[] configPropertyDescriptors;
     private EventHandlerDescriptor[] eventHandlerDescriptors;
}
```

コンポーネントライブラリのマニフェストの補強

コンポーネントマニフェストは、前述の例ですでに作成しています。ここでは、追加の情報を追加します。

前の例では現れなかった追加の種類の情報を追加することに注意してください。

Web アプリケーションフレームワークのライブラリマニフェスト名は、complib.xml である必要があります。Web アプリケーションフレームワークのライブラリマニフェストは、JAR ファイル内の /COMP-INF ディレクトリに入れる必要があります。

1. COMP-INF/complib.xml ファイルを作成するか、開きます。
2. SecureViewField コンポーネントを宣言する拡張可能コンポーネント要素を追加します。

   これまでの手順を終えると、COMP-INF/complib.xml は以下のようになります。

注 – 見やすくするために、ファイルの冒頭にあるバージョンに対する重要なデルタ部分だけ示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
<tool-info>
<tool-version>2.1.0</tool-version>
</tool-info>
<library-name>mycomponents</library-name>
<display-name>My First Component Library</display-name>

 ...

     <extensible-component>
          <component-class>mycomponents.SecureViewBean</component-class>
          <component-info-class>mycomponents.SecureViewBeanComponentInfo</component-info-
class>
     </extensible-component>

...

</component-library>
```

コンポーネントライブラリの JAR ファイルの再作成

最初の例で行ったようにすべてのコンポーネントクラスを JAR ファイルにして、それらのクラスをライブラリとして配布できるようにします。

1. **この JAR ファイルの名前は任意です。**

   この場合は、「mycomponents.jar」という名前にします。

   必ずしも Java ソースファイルを JAR に含める必要はありません。

2. **アイコンイメージやリソースバンドルなどの必要な補助リソースを JAR に含めます。**

   この例では、いくつかの新しいクラスと、Java テンプレートファイル 1 つを含めます。

3. **`mycomponents.jar` の内部構造は以下のようになります。**

```
mycomponents/resources/SecureViewBean_java.template
mycomponents/MissingTokensEvent.class
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
mycomponents/SecureViewBean.class
mycomponents/SecureViewBeanComponentInfo.class
mycomponents/TypeValidator.class
mycomponents/ValidatingDisplayField.class
mycomponents/ValidatingTextFieldComponentInfo.class
mycomponents/ValidatingTextFieldTag.class
mycomponents/Validator.class
mycomponents/mycomplib.tld
COMP-INF/complib.xml
```

新しいコンポーネントのテスト

1. **以前に作成したテスト用アプリケーションに新しいバージョンのライブラリを配備します。**

   IDE に関する重要な注：IDE で、現在マウントされている JAR ファイルを削除したり、上書きコピーしたりすることはできません。対話形式でコンポーネントライブラリを開発し、そのライブラリをテスト用アプリケーションでテストする際は、このことが難題になります。テスト用アプリケーション内で同じライブラリ JAR ファイルの新しいバージョンを繰り返しテストするには、以下の手順を実行します。

   a. **テスト用アプリケーションをマウント解除します。**

b. マウント解除が完了したら、オペレーティングシステムのファイルシステムに移動し、マウント解除したテスト用アプリケーションの `WEB-INF/lib` ディレクトリにある古いライブラリ JAR ファイルに新しいライブラリ JAR ファイルを上書きコピーします。

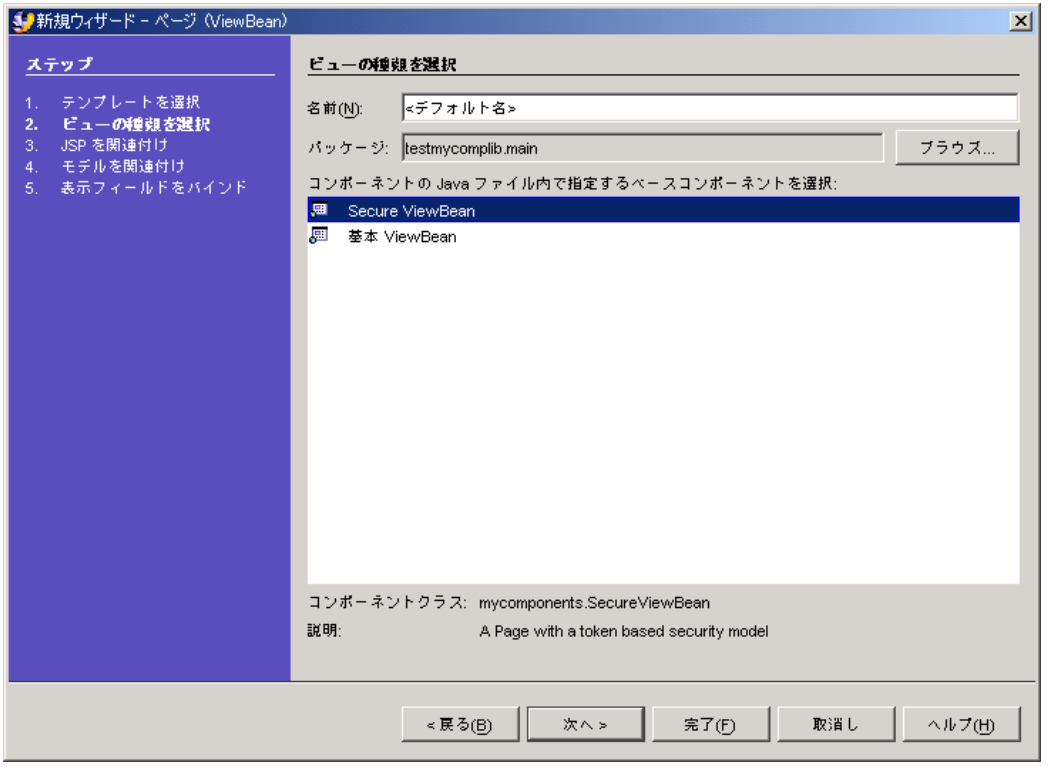
c. テスト用アプリケーションを再マウントします。

これで、テスト用アプリケーションはが、新しいバージョンのライブラリを選択するようになります。

通常、この手順で問題が起きることはありません。新しい JAR を古い JAR に上書きできないか、テスト用アプリケーションを正しく再マウントできないエラーが発生した場合は、IDE を再起動します。

2. 新しい ViewBean オブジェクトを作成します。

以下は、新規 ViewBean ウィザードの画面例です。



3. コンポーネントリストから「Secure ViewBean」を選択して、ウィザードを完了します。

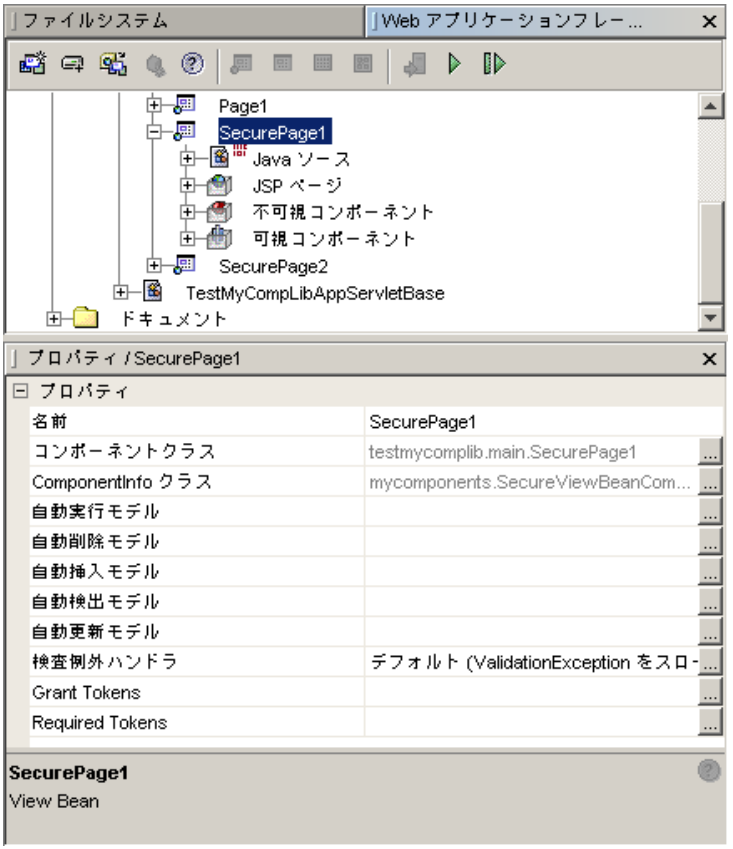
デフォルトの設定をそのまま使用して、ウィザードに SecurePage1 の作成を任せてください。(上記のウィザードの段階で「完了」を選択できます。)

4. ウィザードが終了すると、IDE によって、コンポーネント提供のテンプレートに基づく新しいクラスが作成されます。

5. セキュリティーモデルをテストするため、2 つの目の SecureViewBean を作成します。

   これでアプリケーションには、2 つの SecureViewBean (SecurePage1 および SecurePage2) が含まれることになります。

   新しい SecureViewBean には、「Grant Tokens」および「Required Tokens」プロパティが含まれます。



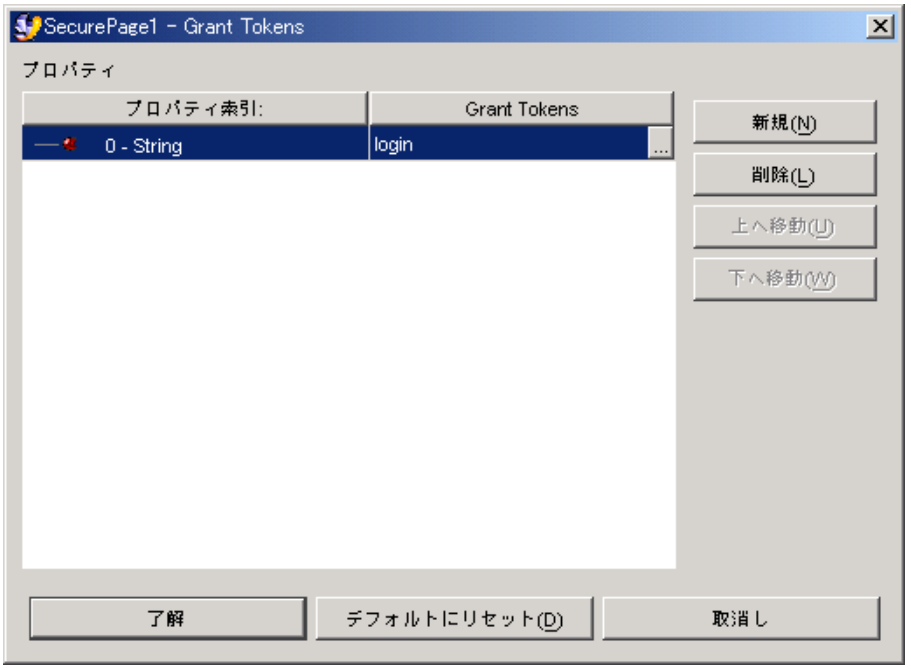
6. トークンのプロパティにいくつかの値を提供してセキュリティーモデルをテストします。

   SecurePage1 の「Grant Tokens」プロパティを選択します。

   プロパティシートで省略符号 (...) ボタンを選択すると、索引付きの String プロパティエディタが起動されます。

7. **プロパティに値「login」を追加します。**

   追加のトークンを追加することができます。



8. **もう一方の Secure ViewBean の SecurePage2 を選択します。**

   その「Required Tokens」プロパティを選択し、値「login」を追加します。

   これで、アプリケーションにページセキュリティー規則を確立したことになります。

   SecurePage2 が「login」トークンを必要とし、SecurePage1 が「login」トークンを付与します。このため、ユーザーが SecurePage2 の前に SecurePage1 を訪れないと、セキュリティー例外がトリガーされることになります。

9. **SecurePage2 はまだ空白ページであるため、このページに関連付けられた JSP (SecurePage2.jsp) に静的コンテンツを追加します。**

   たとえば SecurePage2.jsp に「Welcome to Secure2」テキストを挿入して、ブラウザでそのページであることが分かるようにします。

10. **SecurePage2 をテスト実行します。**

    ブラウザには、SecurePage2.jsp の内容ではなく、次のメッセージが表示されます。

注 – このメッセージが表示されるということは、SecureViewBean セキュリティーモデルが意図したとおりに機能していることを意味します。少なくともアクセス禁止は機能しています。

11. 明確にパスをテストするために、SecurePage1 と SecurePage2 間にリンクを作成します。

    これを行う方法はいくつかあります。

    独自のリンクを実装できます。以下に示すのは、あくまで 1 つの方法です。

    a. Web アプリケーションフレームワークライブラリの基本ボタンのインスタンスを SecurePage1 に追加します。

       コンポーネントパレットからコンポーネントを選択しても、SecurePage1 の「可視コンポーネント」サブノードを選択してから、右クリックし、ポップアップメニューから「可視コンポーネント 追加」アクションを選択してもかまいません。

       テスト用の ViewBean に子「button1」が追加されます。

    b. button1 ビジュアルコンポーネントノードを選択します。

    c. 右クリックし、ポップアップメニューから「イベント」>「handleRequest」アクションを選択します。

       これで SecurePage1 の Java ファイルに、handleButton1Request というイベントハンドラメソッドが追加されます。

    d. handleButton1Request の本体を編集して、以下のようにします。

```
public void handleButton1Request(RequestInvocationEvent event) throws Exception {
getViewBean(SecurePage2.class).forwardTo(getRequestContext());
}
```

12. SecurePage1 から SecurePage2 までのページフローをテストします。

    a. SecurePage1 をテスト実行します。

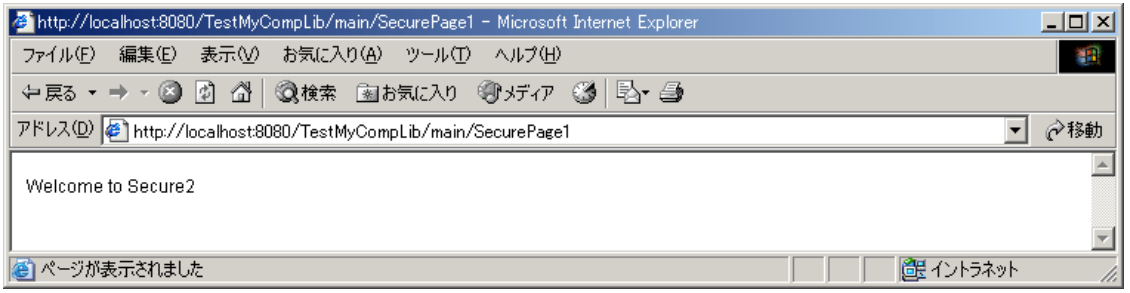
    b. SecurePage1 は、「Submit」というボタン 1 つの空白ページとしてブラウザに表示されます。

       ユーザーには、「login」トークンが付与されます。

    c. 「Submit」ボタンをクリックします。

       これで、handleButton1Request ロジックがトリガーされ、要求が SecurePage2 に転送されます。

       ユーザーが必要なトークンを蓄積しているため、ブラウザに表示される SecurePage2.jsp の内容は下図のようになります。



配布

配布の前に行うことがまだあります。最初に、EventHandlerDescriptor 機能 (handleMissingTokens) をテストします。

SecureViewBeanComponentInfo で、handleMissingTokens というイベントハンドラを記述している EventHandlerDescriptor を宣言していることを思い出してください。この機能をテストする必要があります。

1. 「SecurePage2」ノードを選択します。
2. 右クリックし、ポップアップメニューから「イベント」>「handleMissingTokens」オプションを選択します。

   これで、handleMissingTokens メソッドのスケルトンが SecurePage2.jsp が挿入され、自動的にそのメソッド位置に Java エディタが配置されます。

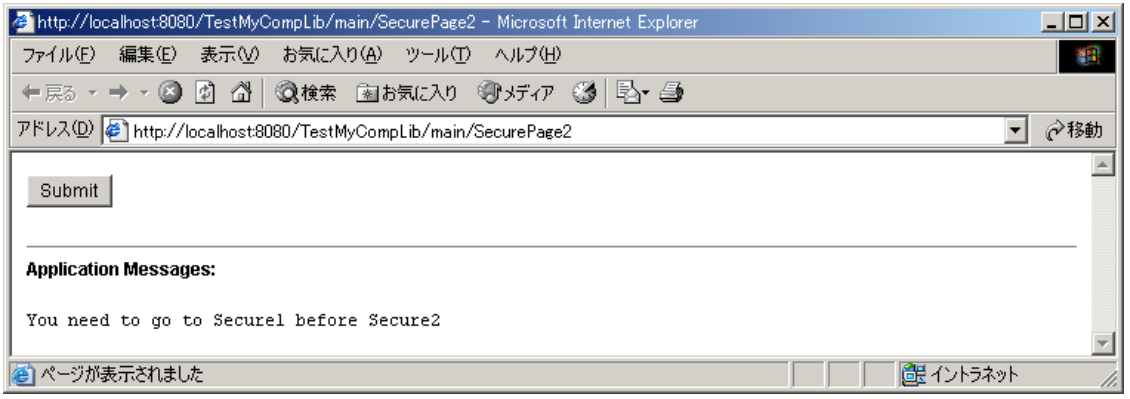
3. メソッドを編集して、このイベントがトリガーされたときに自動的に SecurePage1 にユーザーが戻されるようにします。

   これは、イベントハンドラをテストする 1 つの手段にしかすぎません。アプリケーション開発者は、任意の方法でこのハンドラを実装できます。

```
public void handleMissingTokens(MissingTokensEvent e)throws SecurityCheckException {
        // 不正なアクセスユーザーを SecurePage1 に送る
        appMessage("You need to go to Secure1 before Secure2");
        getViewBean(SecurePage1.class).forwardTo(getRequestContext());
        // 元の要求の処理の打ち切り
        throw new CompleteRequestException();
}
```

4. SecurePage2 を再びテスト実行します。

   イベントハンドラが制御を受け取ったため、今度はブラウザによって SecurePage1 が返されます。



第4章

モデルコンポーネントの開発

この章は、7 ページの「初めてのコンポーネントの開発」をすでに読み終えていることが前提になります。

モデルコンポーネント

理解しやすいモデルコンポーネントとしては、拡張可能モデルコンポーネントがあります。拡張可能モデルコンポーネントは、アプリケーション開発者による個別化を意図したカスタム実装のモデルクラスです。通常、アプリケーション開発者による個別化とは、アプリケーション固有のモデルにスキーマ情報を追加することを意味します。Web アプリケーションフレームワークコンポーネントライブラリには、`QueryModelBase` や `WebServiceModel`、`SessionModel`、`ObjectAdapterModel`、`CustomModel` などの拡張可能モデルコンポーネントが多数含まれています。

ModelComponentInfo

`ModelComponentInfo` インタフェースを使用し、コンポーネント作成者は、あらゆるモデルコンポーネントに適用可能な追加のメタデータを定義することができます。

ExecutingModelComponentInfo

`ExecutingModelComponentInfo` インタフェースを使い、コンポーネント作成者は、そのコンポーネントクラスに `com.iplanet.jato.model.ExecutingModel` インタフェースを実装しているあらゆるモデルコンポーネントに適用可能な追加のメタデータを定義することができます。

拡張不可モデルコンポーネントの作成について

拡張不可モデルコンポーネントの作成は可能です。モデルウィザードを使って新しいモデルを作成するということは、つねに、拡張可能モデルコンポーネントを拡張し、アプリケーション固有のモデルを作成することを意味します。定義では、アプリケーションに固有のこの新しいモデルは拡張不可コンポーネントです。アプリケーション開発者が ModelReference 型のプロパティに値を入力しようとすると、IDE は必ずコンポーネントブラウザを起動して、既存の一群の拡張不可モデルから選択できるようにします。たとえば、表示フィールドの「モデル参照」プロパティを指定すると、IDE にはモデルを選択するブラウザが表示されます。

この、作成した拡張不可モデルを配布できるようにライブラリに追加することもできます。ここでも、拡張不可モデルコンポーネントの作成は可能です。第 7 章の「拡張不可モデルとコマンド、コンテナビューコンポーネントの開発と配布」を参照してください。

拡張不可モデルコンポーネントの開発

拡張不可モデルコンポーネントは、IDE 内で拡張可能モデルコンポーネントから作成された具体モデルです。JAR ファイルで配布され、複数のアプリケーションに組み込めることを除けば、アプリケーション固有のモデルと違いはありません。配布方法は、拡張不可モデル、コンテナビュー、コマンドコンポーネントのどれも共通です。第 7 章の「拡張不可モデルとコマンド、コンテナビューコンポーネントの開発と配布」を参照してください。

拡張可能モデルコンポーネントの開発

この節では、任意の XML ドキュメントに対するアダプタとして働く拡張可能モデルコンポーネントを新規作成する方法を説明します。このアダプタパターンは、Web アプリケーションフレームワークモデルに実装するのによく適しているパターンの 1 つです。この例では、モデルコンポーネントは、Web アプリケーションフレームワークのビューが、Web アプリケーションフレームワークらしい一貫した方法で任意の XML ドキュメントデータにアクセスすることを可能にします。ビューの開発者は、XML の内部のことや XML 固有の API について何も意識する必要はありません。ビュー開発者は、他のあらゆる Web アプリケーションフレームワークモデルのとき同様、XML ドキュメントモデルと対話します。Web アプリケーションフレームワークモデルのデザインの重要な側面の 1 つに焦点を当てます。Web アプリケーションフレームワークモデルの主な目的は、ビューが使用するアプリケーションリ

ソースとしての役割を持たせることにあります。Web アプリケーションフレームワークのビューとモデルの関係の詳細は、『Web アプリケーションフレームワーク開発ガイド』を参照してください。

一般に、新しい拡張可能モデルのデザインは、些細な仕事ではありません。以下の例は、このマニュアル用に簡潔にしているとはいえ、高度です。どのモデルでもそうであるよう、紹介する以外のデザインも可能です。他の例と同様、本稼働に耐える品質のものにするには、さらに改良することを推奨します。この節の目的は、Web アプリケーションフレームワークの拡張可能モデルの実装の仕組みを理解していただくことにあります。

この例では、Web アプリケーションフレームワークコンポーネントモデルに関係する以下のような項目を取り上げます。

- ExtensibleModelComponentInfo
- ModelFieldGroupDescriptor
- ModelFieldDescriptor

XML ドキュメントモデルのデザインにおける主な注目点

このモデルは、ビジネス委託ではありません。モデルには、アダプタとビジネス委託両方のものがあります。たとえば Web アプリケーションフレームワークの標準コンポーネントライブラリの JDBC SQL 照会モデルは、エンタープライズ層と通信する仕事をするため、アダプタ兼ビジネス委託モデルです。

XML ドキュメントモデルは、XML ドキュメントのライフサイクルの面倒を見ません。アプリケーションが XML ドキュメント取得の面倒を見ていると想定します。XML ドキュメントモデルは、アプリケーションによる XML ドキュメントの取得方法について関知しません。アプリケーションが綿密に定義された場所内に XML ドキュメントを配置すると想定します。XML ドキュメントモデルは、必要に応じてその場所から XML ドキュメントにアクセスします。

このデザイン決定には、単にモデルの仕事を簡単にする以上の利点がいくつかあります。1 つの利点として、1 つ以上の XML ドキュメントモデルが同じ XML ドキュメントにアクセスすることを可能にします。アプリケーション開発者にとって、このドキュメントモデルの基本にどのような恩恵があるかは、コンポーネントのテスト中に明らかになります。

もう 1 つの利点は、ドキュメントのライフサイクルの管理用にすでに作成されていた Web アプリケーションフレームワーク以外のコードをアプリケーション開発者がシームレスに利用できるようになることです。

このモデルは、読み取り専用モデルとしてのみ使えるように自身の役割を限定します。このことは、XML ドキュメントデータの読み出しおよび表示はサポートするが、データの変更は促進しないことを意味します。XML ドキュメントの完全な更新サポートの実装については、このガイドでは説明しません。また、サポートしている

こととしていないことを明確に説明している限り、モデルが綿密に定義された機能セットに自身の役割を限定することは完全に正当と認められることです。アプリケーション開発者は、説明されているその使用法に従ってモデルの用途を制限します。

XML ドキュメントモデルコンポーネントには、次のデザイン時機能を持たせます。

- 「Document Scope」というプロパティを表示する。アプリケーション開発者は、3 つある標準のサーブレットコンテナスコープ (要求スコープ、セッションスコープ、アプリケーションスコープ) のうちの 1 つが指定されるように、このプロパティを構成します。このプロパティを設定することによって、アプリケーション開発者は、実行時、指定されたスコープで XML ドキュメントを配置できます。こうしてモデルは、実行時、指定されたスコープからドキュメントをフェッチします。このプロパティのデフォルト値は要求スコープにします。
- 「Document Scope Attribute Name」というプロパティを表示する。これは、「Document Scope」に関係するプロパティです。アプリケーション開発者は、スコープ属性名が指定されるように、このプロパティを構成します。このプロパティを設定することによって、アプリケーション開発者は、指定されたスコープと属性名で XML ドキュメントを配置できます。
- 「Base Dataset Name」というプロパティを表示する。アプリケーション開発者は、XML ドキュメント内へのオフセットが指定されるように、このプロパティを構成します。デフォルト名は、XPath 表現で指定します。どのモデル内でも、モデルフィールド固有のあらゆる XPath 表現は、「Base Dataset Name」が基準になります。このプロパティは空白のままにすることができますが、その場合、モデルフィールド固有の XPath 表現は絶対とみなされます。
- アプリケーション開発者が、デザイン時、モデルに任意の個数のモデルフィールドを追加できるようにする。
  - アプリケーション開発者は、各モデルフィールドが任意のフィールド名を持つようにします。
  - アプリケーション開発者は、各モデルフィールドが XML ドキュメント内の 1 つの値にアクセスするようにします。このアクセスは、「Base Dataset Name」基準にした相対形式か絶対形式 (「Base Dataset Name」がない場合) の XPath 表現で設定する必要があります。
  - ビュー開発者は、Web アプリケーションフレームワークの従来の方法でモデルフィールドにバインドすることができます。

XML ドキュメントモデルコンポーネントには、次の実行時機能を持たせます。

- 指定されたスコープ内の名前付き属性から XML ドキュメントを読み出すことによって防御的に XML ドキュメントにアクセスする。
- フィールド名を XPath 表現にし、XPath 表現をドキュメント内の値に解決する、主要 `com.iplanet.jato.model.Model` メソッドの「getValue(String fieldName)」を実装する。
- `com.iplanet.jato.model.DatasetModel` インタフェースを実装する。あらゆるデータセットモデルは、異なる複数のデータセットに対する一貫したアクセス手段を提供します。この場合、データセットは、XPath 表現が 1 つ以上のノードを返す XML ドキュメントの一部のこともあります。データセットモデルインタ

フェースを実装することによって、アプリケーション開発者はデータセットモデル API を使い、データセット内の複数の値にまたがる繰り返しを行うことができます。これを実現する従来の方法としては、これだけではありませんが、タイルビューをデータセットモデルに関連付ける方法があります。

注 – ここで紹介する実装は、簡潔にするために容易な方法を採用しています。サンプルコードには、実行時の最適化が可能な部分を示すコメントがいくつか含まれていますが、この練習問題よりも複雑なコードが必要になることもあります。

これらの要件を満たすには、次のクラスをデザイン、実装します。

- コンポーネントクラス - `mycomponents.XMLDocumentModel`
- ComponentInfo クラス - `mycomponents.XMLDocumentModelComponentInfo`
- ModelFieldDescriptor クラス - `mycomponents.XMLDocumentModelFieldDescriptor`

また、ここでは、XMLDocumentModel のアプリケーションに固有のサブタイプの土台として IDE が使用するカスタム Java テンプレートも実装します。

最後に mycomponents の `complib.xml` を編集して、その新しいコンポーネントを Web アプリケーションフレームワークのコンポーネントライブラリに追加します。

ModelFieldDescriptor クラスの作成

Web アプリケーションフレームワークのコンポーネントモデルは、`com.iplanet.jato.model.ModelFieldDescriptor` インタフェースの任意の実装を指定する機会を、拡張可能モデルコンポーネントの作成者に提供します。これは最低限のインタフェースです。ModelFieldDescriptor のあらゆる実装はそれぞれに 1 つの JavaBean でもある必要があります。モデルコンポーネントの作成者は、アプリケーション開発者がデザイン時にモデルを定義するようにすることが可能な Bean として ModelFieldDescriptor をデザインします。このためコンポーネント作成者は、JavaBean として表現することが可能な限り、必要とするデザイン時構成機会のすべてを提供することが可能なモデルフィールドをかなり自由にデザインすることができます。

この例では、モデルフィールドのデザイン時構成のニーズは簡単なものです。アプリケーション開発者は、XPath 表現を使って各モデルフィールドを設定できる必要があります。

1. **任意の Java エディタで** `mycomponents.XMLDocumentModelFieldDescriptor` **クラスを作成します。**
2. **基本** `com.iplanet.jato.model.ModelFieldDescriptor` **インタフェースを実装します。**

3. 「XPath」プロパティ用の `get` メソッドと `set` メソッドを追加します。

4. 「FieldClass」プロパティ用の `get` メソッドと `set` メソッドを追加します。

   このプロパティはオプションです。生成した場合、実行時、モデルは XPath 表現を使って読み出された raw 値を強制的に「FieldClass」プロパティで指定された型にします。

   これまでの手順を終えると、`mycomponents/XMLDocumentModelFieldDescriptor.java` は以下のようになります。

```
package mycomponents;

import java.io.*;
import java.util.*;
import com.iplanet.jato.model.*;

/**
 *
 *
 *
 */
public class XMLDocumentModelFieldDescriptor extends Object
     implements ModelFieldDescriptor, Serializable
{

     public XMLDocumentModelFieldDescriptor()
     {
          super();
     }

     public String getName()
     {
          return name;
     }

     public void setName(String name)
     {
          this.name = name;
     }

     public String getXPath()
     {
          return xpath;
     }

     public void setXPath(String xpath)
     {
          this.xpath = xpath;
     }
```

```
    public Class getFieldClass()
    {
        return fieldClass;
    }

    public void setFieldClass(Class fieldClass)
    {
        this.fieldClass = fieldClass;
    }

    private String xpath;
    private String name;
    private Class fieldClass; // この init をデフォルト値に変更してはならい

}
```

Web アプリケーションフレームワークコンポーネントクラスの作成

1. 任意の Java エディタで `mycomponents.XMLDocumentModel` クラスを作成します。
2. XMLDocumentModel に `com.iplanet.jato.view.DatasetModelBase` を拡張させます。
3. XMLDocumentModel に `com.iplanet.jato.view.MultiDatasetModel` を拡張させます。
4. コンポーネントの種類に合わせて適切なコンストラクタを実装します。

   あらゆるモデルコンポーネントは、引数なしのコンストラクタを実装する必要があります。
5. 「DocumentScope」というプロパティ用の `get` メソッドと `set` メソッドを追加します。
6. 「DocumentScopeAttributeName」というプロパティ用の `get` メソッドと `set` メソッドを追加します。
7. 「CurrentDatasetName」というプロパティ用の `get` メソッドと `set` メソッドを追加します。

   このプロパティは、ユーザーにとって分かりやすい表示名である「Base Dataset Path」を取得しますが、その作業は XMLDocumentModelComponentInfo 内で行われます。
8. コンポーネントに固有の要件を満たすために必要な残りのメソッドを実装します。
   - `DatasetModelBase` を抽象化するメソッドを実装します。

- MultiDatasetModel インタフェースが必要とするメソッドを実装します。
- XML ドキュメント適合の実現に必要なすべてのヘルパーメソッドを実装します。

これまでの手順を終えると、mycomponents/XMLDocumentModel.java は以下のようになります。

```
package mycomponents;
import java.util.*;
import com.iplanet.jato.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.model.custom.*;
import com.iplanet.jato.util.*;
import org.w3c.dom.*;
import org.w3c.dom.traversal.*;
import org.apache.xpath.XPathAPI;
import javax.xml.transform.*;
import javax.servlet.jsp.PageContext;


/**
 *
 * @author component-author
 */
public class XMLDocumentModel extends DatasetModelBase implements MultiDatasetModel
{

     public XMLDocumentModel()
     {
          super();
     }


     ////////////////////////////////////////////////////////////////////////////
     // プロパティ
     ////////////////////////////////////////////////////////////////////////////

     public String getCurrentDatasetName()
     {
          // 有効な currentDatasetName を確保するための何らかの防御ロジックを追加
          if(currentDatasetName == null || currentDatasetName.trim().equals(""))
               currentDatasetName = "/";
          return currentDatasetName;
     }

     public void setCurrentDatasetName(String datasetName)
     {
          this.currentDatasetName = datasetName;
     }

     public int getDocumentScope()
```

```
{
     return documentScope;
}


public void setDocumentScope(int documentScope)
{
     this.documentScope = documentScope;
}

public String getDocumentScopeAttributeName()
{
     return documentScopeAttr;
}


public void setDocumentScopeAttributeName(String name)
{
     this.documentScopeAttr = name;
}

public void setDocument(Document value)
{
     doc = value;
}

public Document getDocument()
{
     if(doc == null) {
          // スコープおよび属性名を使ってドキュメントを検出
          // アプリケーションロジックが適切なスコープに doc を
          // 配置していることが前提
          RequestContext rc = RequestManager.getRequestContext();
          String attr = getDocumentScopeAttributeName();
          switch (getDocumentScope())
          {
               case PageContext.REQUEST_SCOPE:
                    doc = (Document)
                    rc.getRequest().getAttribute(attr);
                    break;
               case PageContext.APPLICATION_SCOPE:
                    doc = (Document)
                    rc.getServletContext().getAttribute(attr);
                    break;
               case PageContext.REQUEST_SCOPE:
                    doc = (Document)
                    rc.getRequest().getSession().getAttribute(attr);
                    break;
               default:
                    throw new IllegalArgumentException(
                    "DocumentScope is set to an invalid value " +
```

```
                              getDocumentScope());
          }

          if(DEBUG)
               System.out.println("XMLDocumentModel.getModel doc is " +
               (doc==null?"null":"not null"));

     }
     return doc;
}

////////////////////////////////////////////////////////////////////////
// モデルインタフェースのメソッド
////////////////////////////////////////////////////////////////////////

public Object getValue(String name)
{
     Node node=null;
     try
     {
          node=getNode(name);
     }
     catch (Exception e)
     {
          throw new ModelValueException("Exception getting value for "+
               "field \""+name+"\"",e);
     }

     if (node==null)
          return null;

     Object result=null;
     if (isTextNode(node) || isAttributeNode(node))
     {
          result=node.getNodeValue();

          XMLDocumentModelFieldDescriptor descriptor=(XMLDocumentModelFieldDescriptor)
               getFieldGroup().getFieldDescriptor(name);
          if (descriptor.getFieldClass()!=null)
               result=TypeConverter.asType(descriptor.getFieldClass(),result);
     }
     else
     {
          // ノードをそのまま返し、それを使って行うべきことを呼び出し元に求めさせる。
          // これは、実際に求められていたことと違っているかもしれない。
          result=node;
     }

     return result;
}
```

```
    public Object[] getValues(String name)
    {
         NodeList nodes=null;
         try
         {
              nodes=getNodes(name);
         }
         catch (Exception e)
         {
              throw new ModelValueException("Exception getting values for "+
                   "field \""+name+"\"",e);
         }

         if (nodes==null)
              return new Object[0];

         Object[] result=null;
         try
         {
              List resultList=new LinkedList();
              for (int i=0; i<nodes.getLength(); i++)
              {
                   Node node=nodes.item(i);
                   if (isTextNode(node) || isAttributeNode(node))
                   {
                        Object data=node.getNodeValue();

                        XMLDocumentModelFieldDescriptor descriptor=
(XMLDocumentModelFieldDescriptor)
                             getFieldGroup().getFieldDescriptor(name);
                        if (descriptor.getFieldClass()!=null)
                        {
                             data=TypeConverter.asType(descriptor.getFieldClass(),
                                  data);
                        }

                        resultList.add(data);
                   }
                   else
                   {
                        // ノードをそのまま返し、それを使って行うべきことを呼び出し元に求めさせる。
                        // これは、実際に求められていたことと違っているかもしれない。
                        resultList.add(node);
                   }
              }

              result=resultList.toArray();
         }
         catch (Exception e)
         {
```

```
            throw new ModelValueException("Exception getting values "+
                "for field \""+name+"\"",e);
        }

        return result;
    }



    public void setValue(String name, Object value)
    {
        // 無視
    }

    public void setValues(String name, Object[] value)
    {
        // 無視
    }

    ////////////////////////////////////////////////////////////////////////
    // DatasetModel インタフェースのメソッド
    ////////////////////////////////////////////////////////////////////////

    protected NodeList getCurrentDatasetNodeList()
        throws ModelControlException
    {
        if (nodeList!=null)
            return nodeList;

        if (getDocument()==null)
        {
            throw new ModelControlException(
                "No XML document has been provided");
        }

        try
        {
            // 注 : XPathAPI の代わりに CachedXPathAPI を使って、この呼び出しの効率性を
            // 向上できる。
            // ただし、このためには、この例では役立たない複雑さが多少加わる。
            // また、XML パーサーが DOM レベル 3 をサポートしているのであれば、
            // org.w3c.dom.xpath パッケージを使用することによって、Apache 固有のコードを
            // 使わなくて済む可能性もある。
            nodeList=XPathAPI.selectNodeList(getDocument(),
                getCurrentDatasetName());
        }
        catch (TransformerException e)
        {
            throw new ModelControlException("Exception getting NodeList for "+
                "dataset \""+getCurrentDatasetName()+"\"");
        }
```

```
        return nodeList;
    }


    public int getLocationOffset()
    {
        return 0;
    }


    public int getLocation()
        throws ModelControlException
    {
        Integer index=(Integer)datasetContexts.get(getCurrentDatasetName());
        if (index==null)
        {
            // 単に NodeList の妥当性を検査するための呼び出し
            getCurrentDatasetNodeList();
            return -1;
        }

        return index.intValue();
    }


    public void setLocation(int value)
        throws ModelControlException
    {
        int maxLength=getCurrentDatasetNodeList().getLength();
        if (value>=maxLength || value<-1)
        {
            throw new ModelControlException("Location index out of "+
                "range (max value = "+(maxLength-1)+")");
        }

        datasetContexts.put(getCurrentDatasetName(),new Integer(value));
    }


    public int getSize()
        throws ModelControlException
    {
        return getCurrentDatasetNodeList().getLength();
    }


    public void setSize(int value)
        throws ModelControlException
    {
        throw new ModelControlException("Unsupported operation; "+
```

```
            "model size cannot be set");
    }

    protected boolean ensureValidDataPosition()
        throws ModelControlException
    {
        if (getSize()==0)
            return false; // 読み出すデータなし
        else
        if (getLocation()==-1)
        {
            // 先頭項目の前にいない場合は、先頭項目に移動して
            // データを読み出す必要がある。
            if (!first())
                throw new ModelControlException("Could not move to first item");
        }

        return true;
    }

    ////////////////////////////////////////////////////////
    // XML ノードのメソッド
    ////////////////////////////////////////////////////////////////////////

    public Node getNode(String fieldName)
        throws ModelControlException, TransformerException
    {
        if (!ensureValidDataPosition())
            return null;

        Node contextNode=getCurrentDatasetNodeList().item(getLocation());

        // 注 : XPathAPI の代わりに CachedXPathAPI を使って、この呼び出しの効率性を
        // 向上できる。
        // ただし、このためには、この例では役立たない複雑さが多少加わる。
        // また、XML パーサーが DOM レベル 3 をサポートしているのであれば、
        // org.w3c.dom.xpath パッケージを使用することによって、Apache 固有のコードを
        // 使わなくて済む可能性もある。
        Node n = XPathAPI.selectSingleNode(contextNode,getFieldXPath(fieldName));
        if(DEBUG) {
            if(n == null)
                System.out.println("Warning: getNode found no node at[" +
                    getFieldXPath(fieldName) + "]");
        }
        return n;
    }

    public NodeList getNodes(String fieldName)
        throws ModelControlException, TransformerException
    {
        if (!ensureValidDataPosition())
```

```
            return null;

        Node contextNode=getCurrentDatasetNodeList().item(getLocation());

        // 注 : XPathAPI の代わりに CachedXPathAPI を使って、この呼び出しの効率性を
        // 向上できる。
        // ただし、このためには、この例では役立たない複雑さが多少加わる。
        // また、XML パーサーが DOM レベル 3 をサポートしているのであれば、
        // org.w3c.dom.xpath パッケージを使用することによって、Apache 固有のコードを
        // 使わなくて済む可能性もある。
        NodeList nl = XPathAPI.selectNodeList(contextNode,getFieldXPath(fieldName));
        if(DEBUG) {
            if(nl == null)
                System.out.println("Warning: getNodes found no nodes at[" +
                    getFieldXPath(fieldName) + "]");
        }
        return nl;
    }


    public static boolean isTextNode(Node node)
    {
        if (node==null)
            return false;
        return (node instanceof CharacterData);
    }

    public static boolean isAttributeNode(Node node)
    {
        if (node==null)
            return false;
        return node.getNodeType()==Node.ATTRIBUTE_NODE;
    }

    ////////////////////////////////////////////////////////////////////////
    // ヘルパーのメソッド
    ////////////////////////////////////////////////////////////////////////
    public String getFieldXPath(String fieldName)
    {
        XMLDocumentModelFieldDescriptor descriptor=(XMLDocumentModelFieldDescriptor)
            getFieldGroup().getFieldDescriptor(fieldName);
        return descriptor.getXPath();
    }
    ////////////////////////////////////////////////////////////////////////
    // インスタンスの変数
    ////////////////////////////////////////////////////////////////////////

    private int documentScope = PageContext.REQUEST_SCOPE; // デフォルトでは要求スコープ
    private String documentScopeAttr = "testDoc";
    private String currentDatasetName;
    private Document doc;
```

```
    private NodeList nodeList;
    private Map datasetContexts=new HashMap();
    private String datasetName;

    private static final boolean DEBUG = true;
}
```

拡張可能コンポーネントの Java テンプレートの作成

拡張可能コンポーネントは、アプリケーション定義のエンティティに対するベースクラスの働きをします。このため、Web アプリケーションフレームワークコンポーネントモデルは、拡張可能コンポーネントの作成者がカスタム Java テンプレートを提供する機会を提供します。IDE では、そのコンポーネント提供のテンプレートを利用して、アプリケーション固有のサブタイプを作成します。コンポーネント作成者は、そのカスタムテンプレートを利用して、アプリケーション開発者の作業をさらに充実したものにすることができます。コンポーネント作成者は、`com.iplanet.jato.component.ExtensibleComponentInfo` に定義されている一群のテンプレートトークンを使ってコンポーネント固有の Java テンプレートを用意することができます。トークンの詳細は、ExtensibleComponent API を参照してください。

コンポーネント作成者はまた、Java テンプレート内で任意の Java 構造 (インポート文、メソッド、変数、インタフェース宣言など) を利用することもできます。このカスタムテンプレートは、最低限、拡張可能コンポーネントクラスから新しい Java クラスを拡張する保証をします。コンポーネント作成者はさらに、ソース内でインラインに開発者のドキュメンテーションと通信する手段としてテンプレートを利用し、個別化中に覚えておくべき「推奨手順」や条件、限度情報を提供することもできます。

この例のテンプレートでは最低限の内容を示します。

任意のテキストエディタで `mycomponents.resources.XMLDocumentModel_java.template` テンプレートを作成します。

テンプレートの内容は、以下のようにします。

注 – a __TOKEN__ pattern の後にトークンが続きます。

```
package __PACKAGE__;

import java.io.*;

import java.util.*;
```

```
import javax.servlet.*;
import javax.servlet.http.*;
import com.iplanet.jato.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.util.*;
import mycomponents.*;


/**
 *
 *
 * @author
 */
public class __CLASS_NAME__ extends XMLDocumentModel
{
     /**
      * Default constructor
      *
      */
     public __CLASS_NAME__()
     {
          super();
     }

}
```

ComponentInfo クラスの作成

ComponentInfo クラスは、IDE がコンポーネントを組み込むのに必要なデザイン時メタデータを定義します。この例では、既存の ComponentInfo を拡張し、本当のオブジェクト指向のスタイルで単にこのクラスを補強します。当然、ComponentInfo インタフェースを最初から実装する方法を選ぶこともできますが、この例では生産的ではありません。

この例では、前述のコンポーネント例で明らかにした機能をさらに発展させます。以下では、ExtensibleModelComponentInfo インタフェースがメタデータで提供する 1 つの新しい機会、すなわち、任意のモデルフィールド型を記述する機会を利用します。

1. mycomponents.XMLDocumentModelComponentInfo クラスを作成します。
2. XMLDocumentModelComponentInfo に com.iplanet.jato.model.ExtensibleModelComponentInfo を拡張させます。
3. 引数なしのコンストラクタを実装します。

4. コンポーネントの基本的なデザイン時説明を提供する `getComponentDescriptor()` メソッドを実行します。

5. IDE で表示させるプロパティを識別する `getConfigPropertyDescriptors()` メソッドを実装します。

   ConfigPropertyDescriptor 宣言内のデフォルト値の使われ方に注意してください。

   - 「DocumentScope」プロパティ用の ConfigPropertyDescriptor を追加
   - 「DocumentScopeAttributeName」プロパティ用の ConfigPropertyDescriptor を追加
   - 「CurrentDatasetName」プロパティ用の ConfigPropertyDescriptor を追加

6. Java テンプレートを実装する `getPrimaryTemplateAsStream()` メソッドを実装します。IDE は、この拡張可能コンポーネントから派生する新しいクラスに対する出発点として、この Java テンプレートを使用します。

7. モデルが必要とするモデルフィールド型のデザイン時説明を提供する `getModelFieldGroupDescriptors()` メソッドを実装します。

   ModelFieldDescriptor に加えて ModelFieldGroupDescriptor によってもたらされる余分なレベルの間接性に惑わされないでください。ModelFieldDescriptor は重視すべききわめて重要な機能です。ModelFieldGroupDescriptor は高度なオプション機能です。大部分の Web アプリケーションフレームワークモデルコンポーネントは、単純に標準の `com.iplanet.jato.model.ModelFieldGroup` を利用できます。

   これまでの手順を終えると、`mycomponents/XMLDocumentModelComponentInfo.java` は以下のようになります。

注 – このコードでは、例として紹介しやすいよう、String 値を直接埋め込んでいます。表示文字列を各言語対応にする必要があると思われる場合は、リソースバンドルを利用してください。

```
package mycomponents;

import java.util.*;
import java.io.*;
import com.iplanet.jato.component.*;
import com.iplanet.jato.model.*;

/**
 *
 *
 */
public class XMLDocumentModelComponentInfo extends ExtensibleModelComponentInfo
{

     public XMLDocumentModelComponentInfo()
```

```
    {
        super();
    }

    public ComponentDescriptor getComponentDescriptor()
    {
        // コンポーネントクラスを示す
        ComponentDescriptor result=new ComponentDescriptor(
            "mycomponents.XMLDocumentModel");

        // この名前が、コンポーネントのインスタンスの名前を決めるのに使用される
        result.setName("XMLDocumentModel");

        // この表示名が、コンポーネント選択で使用される
        result.setDisplayName("XML Document Model");

        // この説明がコンポーネントのツールチップのテキストになる
        result.setShortDescription("A simple demonstration of a new model component");

        return result;

    }


    public String getPrimaryTemplateEncoding()
    {
/* 本稼働のバージョンは次のようにリソースバンドル駆動型
return getResourceString(getClass(),
"PROP_XMLDocumentModel_SOURCE_TEMPLATE_ENCODING", "ascii");
*/

        return "ascii";
    }

    public InputStream getPrimaryTemplateAsStream()
    {
/* 本稼働のバージョンは次のようにリソースバンドル駆動型
return XMLDocumentModelComponentInfo.class.getClassLoader().
getResourceAsStream(
getResourceString(getClass(),
"RES_XMLDocumentModelComponentInfo_SOURCE_TEMPLATE",""));
*/

        return XMLDocumentModelComponentInfo.class.getResourceAsStream(
            "/mycomponents/resources/XMLDocumentModel_java.template");
    }

    public ConfigPropertyDescriptor[] getConfigPropertyDescriptors()
    {
        if (configPropertyDescriptors!=null)
            return configPropertyDescriptors;
```

```
        configPropertyDescriptors=super.getConfigPropertyDescriptors();
        List descriptors=new LinkedList(Arrays.asList(configPropertyDescriptors));

        ConfigPropertyDescriptor descriptor = null;

        descriptor=new ConfigPropertyDescriptor(
             "documentScope",Integer.TYPE);
        descriptor.setDisplayName("Document Scope");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptor.setDefaultValue(new Integer(
             javax.servlet.jsp.PageContext.REQUEST_SCOPE));
        descriptors.add(descriptor);

        descriptor=new ConfigPropertyDescriptor(
             "documentScopeAttributeName",String.class);
        descriptor.setDisplayName("Document Scope Attribute Name");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptor.setDefaultValue("");
        descriptors.add(descriptor);

        descriptor=new ConfigPropertyDescriptor(
             "currentDatasetName",String.class);
        descriptor.setDisplayName("Base Dataset Path");
        descriptor.setHidden(false);
        descriptor.setExpert(false);
        descriptor.setDefaultValue("");
        descriptors.add(descriptor);

        // 配列を作成 / 返す
        configPropertyDescriptors = (ConfigPropertyDescriptor[])
             descriptors.toArray(
                  new ConfigPropertyDescriptor[descriptors.size()]);
        return configPropertyDescriptors;
    }

    public ModelFieldGroupDescriptor[] getModelFieldGroupDescriptors()
    {
        if(null != modelFieldGroupDescriptors)
             return modelFieldGroupDescriptors;

        List descriptors=new ArrayList();
        ModelFieldGroupDescriptor descriptor=null;

        descriptor = new ModelFieldGroupDescriptor(
             "Fields",
             ModelFieldGroup.class,
             new ConfigPropertyDescriptor[0],
             XMLDocumentModelFieldDescriptor.class,
```

```
                "addFieldDescriptor",
                "setFieldGroup");

          descriptor.setFieldBaseName("field");
          descriptor.setFieldTypeDisplayName("Field");
          descriptor.setGroupDisplayName("Fields");
          descriptor.setFieldPropertyEditorClass(null);
          descriptors.add(descriptor);

          modelFieldGroupDescriptors = (ModelFieldGroupDescriptor[])
               descriptors.toArray(
                    new ModelFieldGroupDescriptor[descriptors.size()]);
          return modelFieldGroupDescriptors;
     }

     private ModelFieldGroupDescriptor[] modelFieldGroupDescriptors;
     private ConfigPropertyDescriptor[] configPropertyDescriptors;

}
```

コンポーネントライブラリのマニフェストの補強

コンポーネントマニフェストは、前述の例ですでに作成しています。ここでは、追加の情報を追加します。

前の例では現れなかった追加の種類の情報を追加することに注意してください。

Web アプリケーションフレームワークのライブラリマニフェスト名は、`complib.xml` である必要があります。Web アプリケーションフレームワークのライブラリマニフェストは、JAR ファイル内の `/COMP-INF` ディレクトリに入れる必要があります。

1. `COMP-INF/complib.xml` ファイルを作成するか、開きます。
2. XMLDocumentModel コンポーネントを宣言する拡張可能コンポーネント要素を追加します。

   これまでの手順を終えると、`COMP-INF/complib.xml` は以下のようになります。

注 – 見やすくするために、ファイルの冒頭にあるバージョンに対する重要なデルタ部分だけ示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
<tool-info>
<tool-version>2.1.0</tool-version>
</tool-info>
<library-name>mycomponents</library-name>
<display-name>My First Component Library</display-name>

 ...

     <extensible-component>
          <component-class>mycomponents.XMLDocumentModel</component-class>
          <component-info-class>mycomponents.XMLDocumentModelComponentInfo</component-info-
class>
     </extensible-component>

...

</component-library>
```

コンポーネントライブラリの JAR ファイルの再作成

最初の例で行ったようにすべてのコンポーネントクラスを JAR ファイルにして、それらのクラスをライブラリとして配布できるようにします。

1. **この JAR ファイルの名前は任意です。**

   この場合は、「mycomponents.jar」という名前にします。

2. **必ずしも Java ソースファイルを JAR に含める必要はありません。**

3. **アイコンイメージやリソースバンドルなどの必要な補助リソースを JAR に含めます。**

   この例では、そうしたリソースはありません。

   この例では、いくつかの新しいクラスと、Java テンプレートファイル 1 つを含めます。

4. mycomponents.jar の内部構造は以下のようになります。

```
mycomponents/resources/SecureViewBean_java.template
mycomponents/resources/XMLDocumentModel_java.template
mycomponents/MissingTokensEvent.class
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
mycomponents/SecureViewBean.class
mycomponents/SecureViewBeanComponentInfo.class
mycomponents/TypeValidator.class
mycomponents/ValidatingDisplayField.class
mycomponents/ValidatingTextFieldComponentInfo.class
mycomponents/ValidatingTextFieldTag.class
mycomponents/Validator.class
mycomponents/XMLDocumentModel.class
mycomponents/XMLDocumentModelComponentInfo.class
mycomponents/XMLDocumentModelFieldDescriptor.class
mycomponents/mycomplib.tld
COMP-INF/complib.xml
```

新しいコンポーネントのテスト

1. 以前に作成したテスト用アプリケーションに新しいバージョンのライブラリを配備します。

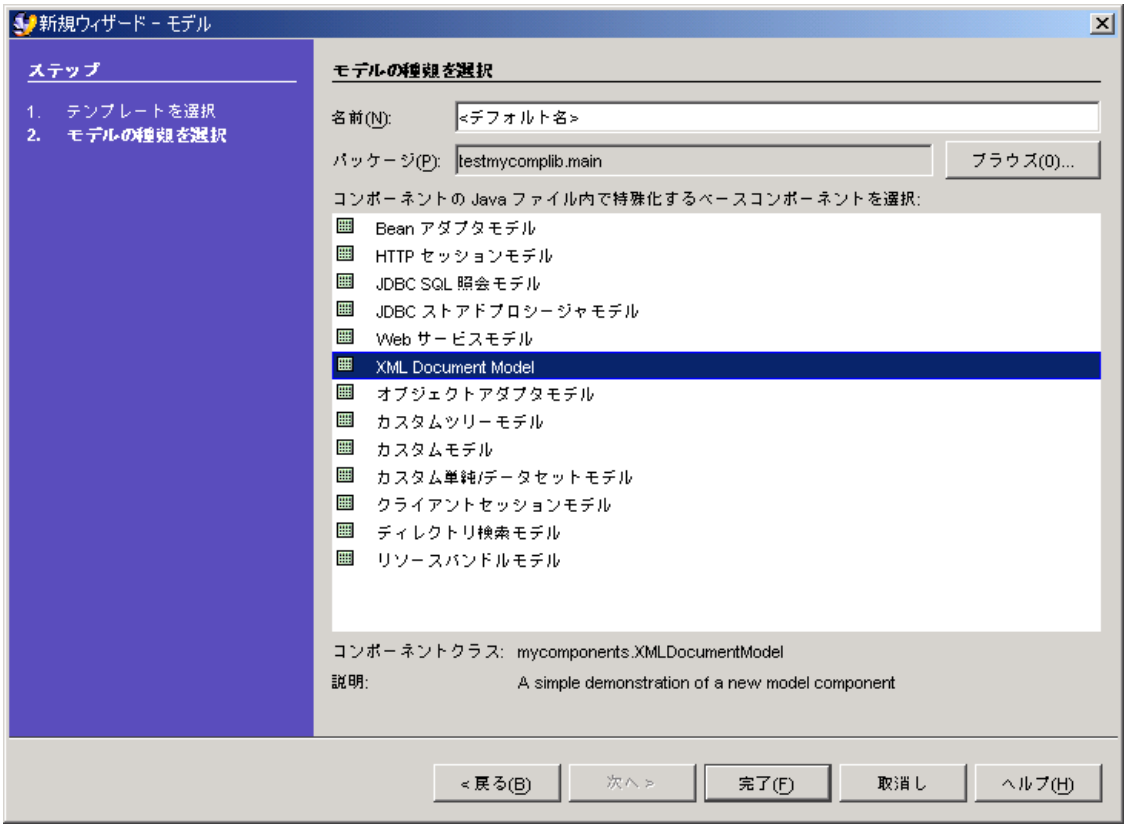
   IDE に関する重要な注：IDE で、現在マウントされている JAR ファイルを削除したり、上書きコピーしたりすることはできません。これらの操作は、IDE と同じ VM を共有し、ファイルロックを共有している ANT タスクを使い IDE で行う必要があります。現在マウントされている JAR ファイルを置き換える必要がある場合は、必ず、IDE を停止してください。IDE ですでに開かれているプロジェクトで新しいバージョンのコンポーネントライブラリをテストするには、まず、IDE を停止します。IDE によって古いバージョンのライブラリ JAR ファイルが解放されると、新しいバージョンの JAR ファイルを古いバージョンに上書きコピーすることができます。新しいバージョンのライブラリを配備すると、再び IDE でアプリケーションを開くことができます。

2. 新しいモデルオブジェクトを作成します。

   『Web アプリケーションフレームワーク チュートリアル』をまだ完了していていない場合は、終えておいてください。

   以下は、新規モデルウィザードの画面例です。

注 – Web アプリケーションフレームワークのバージョンによっては、以下に示すモデルの一部が表示されないことがあります。重要なことは、「XML Document Model」のエントリが表示されることです。



3. コンポーネントリストから「XML Document Model」を選択して、ウィザードを完了します。

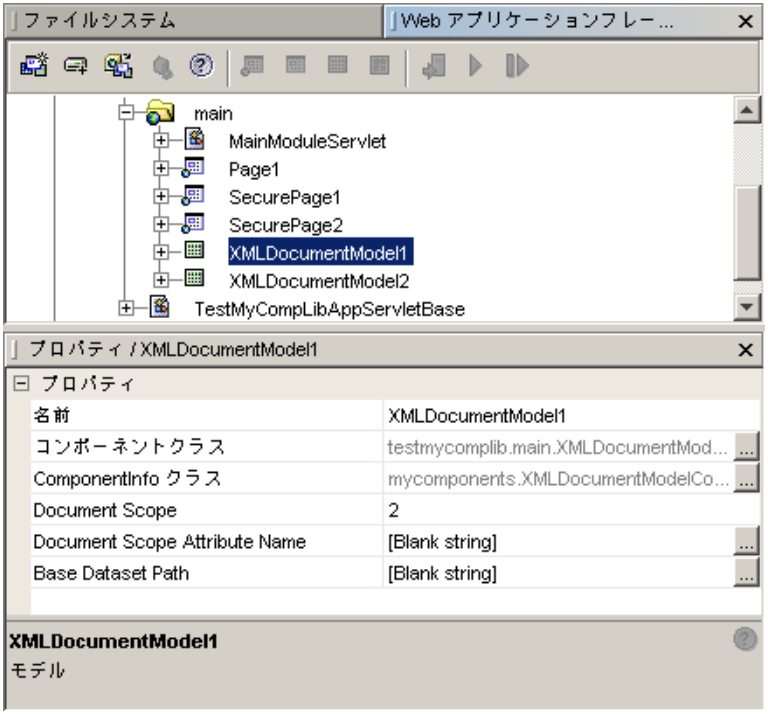
   デフォルトの設定をそのまま使用して、ウィザードに「XMLDocumentModel1」の作成を任せてください。

   ウィザードが終了すると、IDE によって、コンポーネント提供のテンプレートに基づく新しいクラスが作成されます。

4. モードを完全にテストするには、2 つ目の XML ドキュメントモデルを作成します。

   これでアプリケーションに、2 つの XML ドキュメントモデル (XMLDocumentModel1 と XMLDocumentModel2) が含まれることになります。

   「Base Dataset Name」「Document Scope」「Document Scope Attribute Name」プロパティに注目してください。



「Document Scope」プロパティ値が raw 整数の 2 であることに注意してください。これは、XMLDocumentModelComponentInfo によって「DocumentScope」プロパティが宣言されているためです。型は `Integer.Type`、デフォルト値は `javax.servlet.jsp.PageContext.REQUEST_SCOPE` です。このため、IDE では、raw 整数値 (この場合は 2) を表すデフォルトの Integer プロパティエディタを使用することになります。

```
descriptor=new ConfigPropertyDescriptor(
     "documentScope",Integer.TYPE);
descriptor.setDisplayName("Document Scope");
descriptor.setHidden(false);
descriptor.setExpert(false);
descriptor.setDefaultValue(new Integer(
     javax.servlet.jsp.PageContext.REQUEST_SCOPE));
descriptors.add(descriptor);
```

2 が要求スコープに対応していることは、大部分の開発者が知らないと思われるため、これはユーザーインタフェースとして貧弱と思われます。このため、フォローアップとして、デフォルトの Integer プロパティエディタを、もっとユーザーに親切なプロパティエディタに置き換える方法を後で説明します。

新しいモデルコンポーネントをテストするには、適切な XML ドキュメントが必要です。

テスト事例では、任意の XML ファイルをディスクに置きます。実行時、テスト用アプリケーションは、ディスクからそのドキュメントを読み取り、要求スコープに書き込みます。

作成した XML ドキュメントモデルは、XML ドキュメントが存在する場所を関知しません。実際には、多くの場合 XML ドキュメントは、アプリケーションによってエンタープライズ層から動的にフェッチされます。このことは、XML ドキュメントモデルには重要ではありません。

a. 任意のテキストエディタで次の XML を「author.xml」というファイルにコピーします。

b. XMLDocumentModel1 および XMLDocumentModel2 と同じアプリケーションモジュールディレクトリに author.xml を置きます。

   以下で入力するコードは、このファイルがテスト用のモデルと同じディレクトリにあることを前提にしています。これは単純にこの練習問題に都合が良いためです。

```
<?xml version="1.0"?>
<author>
     <name first="Charles" last="Dickens"/>
     <details birth="1812" death="1870"/>
     <works>
          <book title="Great Expectations" publisher="Penguin USA " pages="544"/>
          <book title="Nicholas Nickleby" publisher="Penguin USA " pages="816"/>
          <book title="A Tale of Two Cities" publisher="Signet Classic" pages="371"/>
          <book title="Hard Times" publisher="Bantam Classic" pages="280"/>
          <book title="Oliver Twist" publisher="Tor Books" pages="496"/>
          <book title="David Copperfield " publisher="Penguin USA " pages="912"/>
          <book title="A Christmas Carol" publisher="Bantam Classics" pages="102"/>
          <book title="Our Mutual Friend" publisher="Indypublish.Com" pages="472"/>
          <book title="Bleak House" publisher="Penguin USA " pages="1036"/>
          <book title="The Pickwick Papers " publisher="Penguin USA " pages="848"/>
          <book title="The Haunted House" publisher="Hesperus Press" pages="128"/>
          <book title="Little Dorrit" publisher="Indypublish.Com" pages="460"/>
          <book title="Barnaby Rudge" publisher="Viking Press" pages="766"/>
          <book title="The Mystery of Edwin Drood" publisher="Penguin USA" pages="432"/>
          <book title="Sketches by Boz" publisher="Penguin USA" pages="635"/>
     </works>
</author>
```

c. author.xml の内容を確認します。

作家 1 人に多くの著書エントリがあることに注意してください。ここで大切なことは、2 つのモデル (XMLDocumentModel1 と XMLDocumentModel2) を利用して、同じ XML ドキュメントの異なる部分にアクセスするということです。XMLDocumentModel1 は、スカラー型の作家情報、名前、詳細がアクセスされるように構成します。XMLDocumentModel2 は、非スカラー型の著書目録がアクセスされるように構成します。

これは、XMLDocumentModel をデザインするときに選択した方法です。そうすることによって、モデルコンポーネントの実装が簡単になるばかりでなく、アプリケーションでコンポーネントを使用するのが簡単になりました。

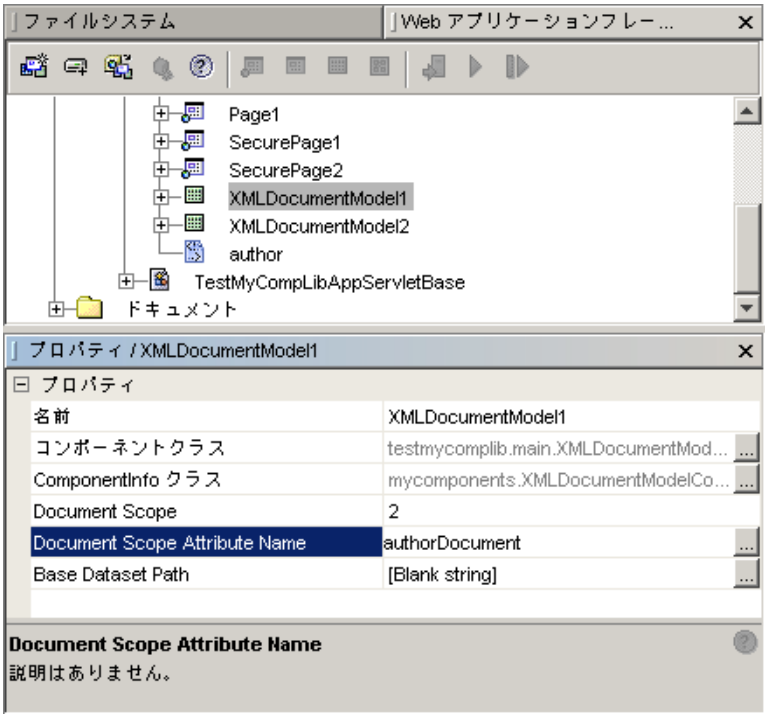
5. XMLDocumentModel1 がスカラー型で作家情報をアクセスするようにします。

a. 「XMLDocumentModel1」ノードを選択します。

b. その「Document Scope Attribute Name」プロパティを編集します。

プロパティ値として「authorDocument」を設定します。

c. 「Document Scope」および「Base Dataset Path」プロパティ値はそのままにします。



d. 「XMLDocumentModel1」ノードを展開して、その「Fields」サブノードを表示します。

e. 「Fields」サブノードを選択します。

f. 右クリックし、ポップアップメニューから「Field 追加」アクションを選択します。

これで、デフォルト名を持つフィールドが自動的に追加されます。

この場合、デフォルト名は「field1」になります。

g. 直前の手順を繰り返して、追加のフィールドの「field2」「field3」「field4」を作成します。

練習問題の目的上、名前は変えずにおきます。

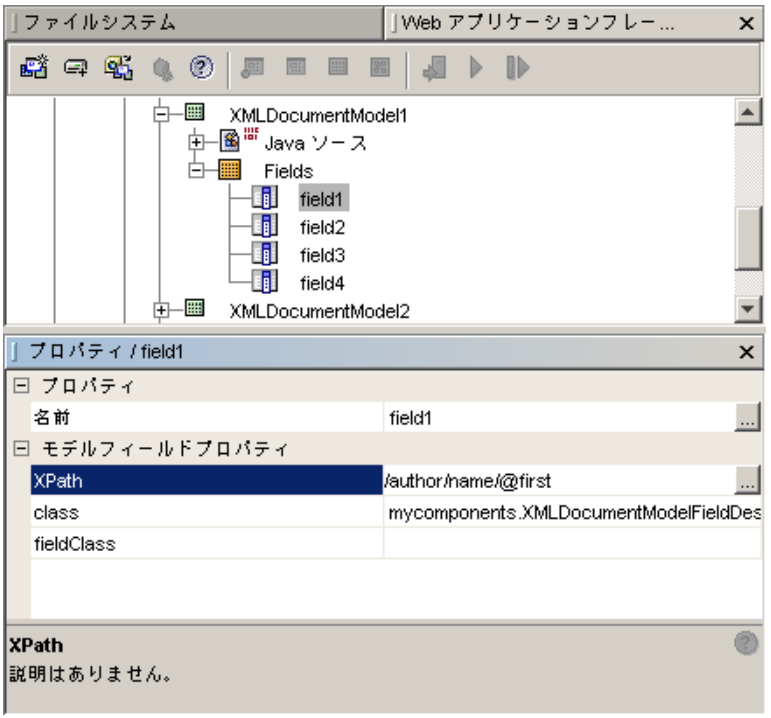
たいていの場合、実際のアプリケーションでは、モデル開発者はフィールド名を変更して、その役割がわかりやすくなるようにします。

h. 「field1」ノードを選択します。

プロパティシートで「モデルフィールドプロパティ」セクションを選択します。

i. field1 の「XPath」プロパティを編集します。

プロパティ値として XPath 表現の「/author/name/@first」を設定します。



j. 直前の手順を繰り返し、残りの 3 つのフィールドの「XPath」プロパティを以下のように設定します。

- field2 の XPath 値を「/author/name/@last」に設定します。
- field3 の XPath 値を「/author/details/@birth」に設定します。
- field4 の XPath 値を「/author/details/@death」に設定します。

6. XMLDocumentModel2 が「books」データセットにアクセスするようにします。

a. 「XMLDocumentModel2」ノードを選択します。

b. その「Document Scope Attribute Name」プロパティを編集します。

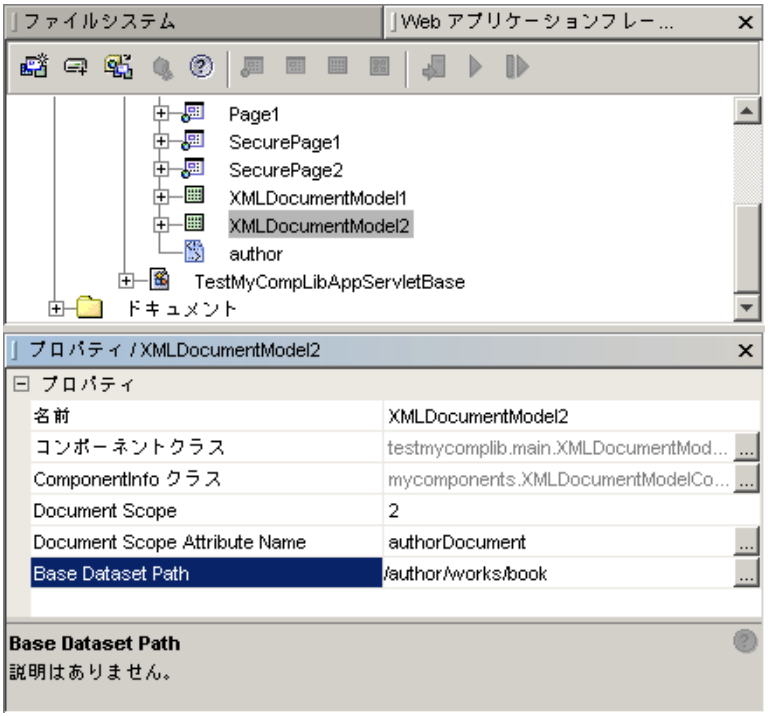
プロパティ値として「authorDocument」を設定します。

c. その「Base Dataset Path」プロパティを編集します。

プロパティ値として XPath 表現の「/author/works/book」を設定します。

これは、著書目録 (Web アプリケーションフレームワークデータセットなど) のアドレスを指定する XPath です。

d. 「Document Scope」プロパティはそのままにします。



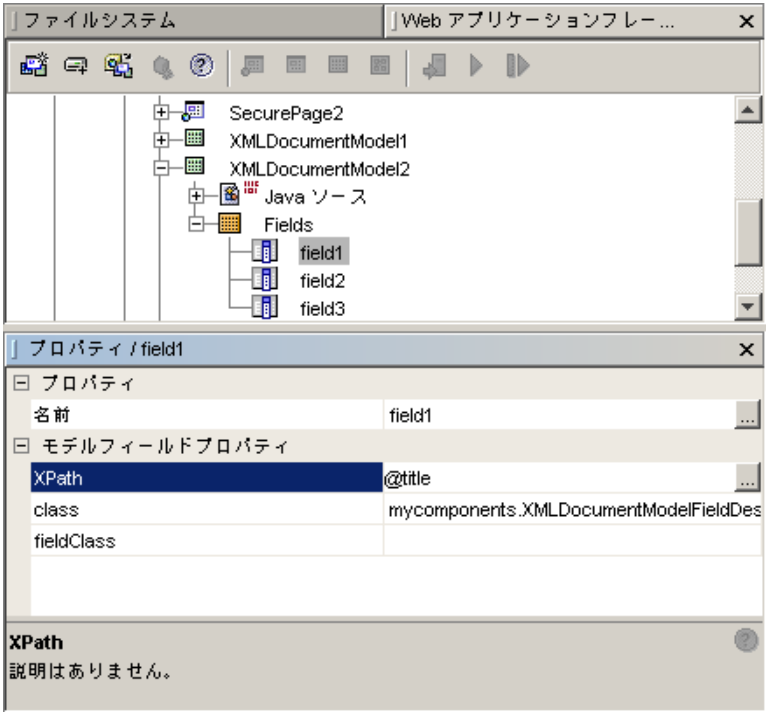
e. 「XMLDocumentModel2」ノードを展開して、その「Fields」サブノードを表示します。

f. 「Fields」サブノードを選択します。

g. 右クリックし、ポップアップメニューから「Field 追加」アクションを選択して、「field1」「field2」「field3」を追加します。

h. 各フィールドの「XPath」プロパティ値として、上記の「Base Dataset Path」プロパティ値を基準にした XPath 表現を設定します。

- field1 の XPath 値を「@title」に設定します。
- field2 の XPath 値を「@publisher」に設定します。
- field3 の XPath 値を「@pages」に設定します。



これでモデルを構成しました。モデルを使用するにはいくつかのビューを作成し、ディスクから `author.xml` ドキュメントを読み取って、要求スコープ属性の「authorDocument」に格納するアプリケーションロジックを用意する必要があります。

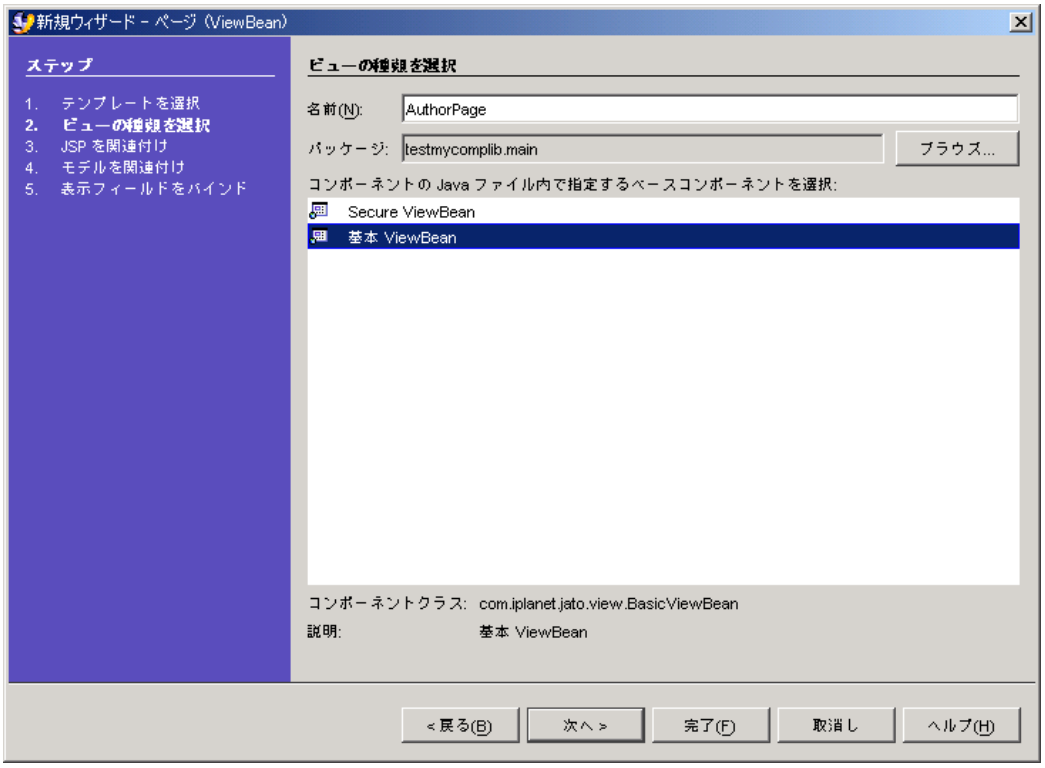
最初に、ビューを作成します。

注意：取り組む前に次の手順全体をお読みください。正しく指示に従い、「新規ビュー」ウィザードの機能全体を利用することによって、時間と手間を省くことができます。下記の詳細手順に従わずに、XMLDocumentModel1 を実行する ViewBean を作成してもまったく問題はありません。任意のスタイルで ViewBean を作成することができます。しかし、Web アプリケーションフレームワーク初心者の場合は、以下の手順が最も簡潔です。

7. AuthorPage を作成します。

   a. 「新規ビュー」ウィザードを起動します。

   - 「ビューの種類を選択」パネルで「基本 ViewBean」コンポーネントを選択します (下図を参照)。
   - 「名前」に「AuthorPage」を設定します。
   - 「次へ」をクリックします。



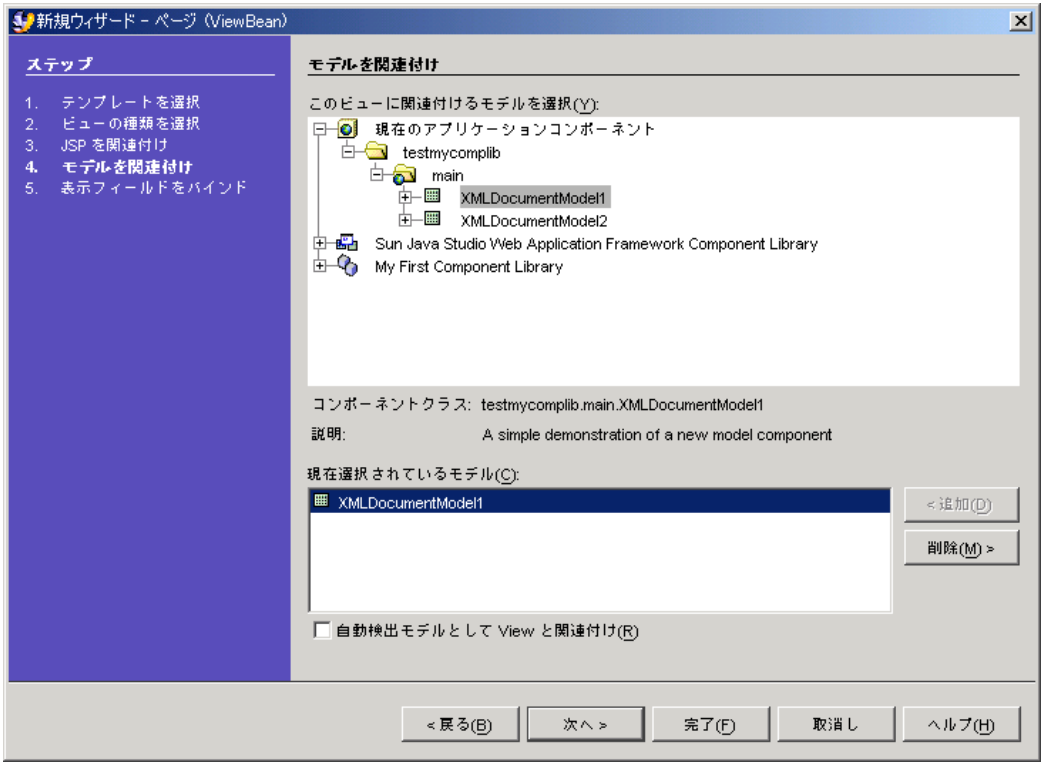
b. 「JSP を関連付け」パネルでは、デフォルト値をそのまま使用し、「次へ」をクリックします。

c. 「モデルを関連付け」パネルで、最終的に「XMLDocumentModel1」が表示されるまで「現在のアプリケーションコンポーネント」ノードを展開します (下図を参照)。

- 「XMLDocumentModel1」を選択し、「追加」ボタンをクリックして、ViewBean とモデルの関連付けを作成します。

  これで、「XMLDocumentModel1」がパネルの「現在選択されているモデル」セクションに現れます。

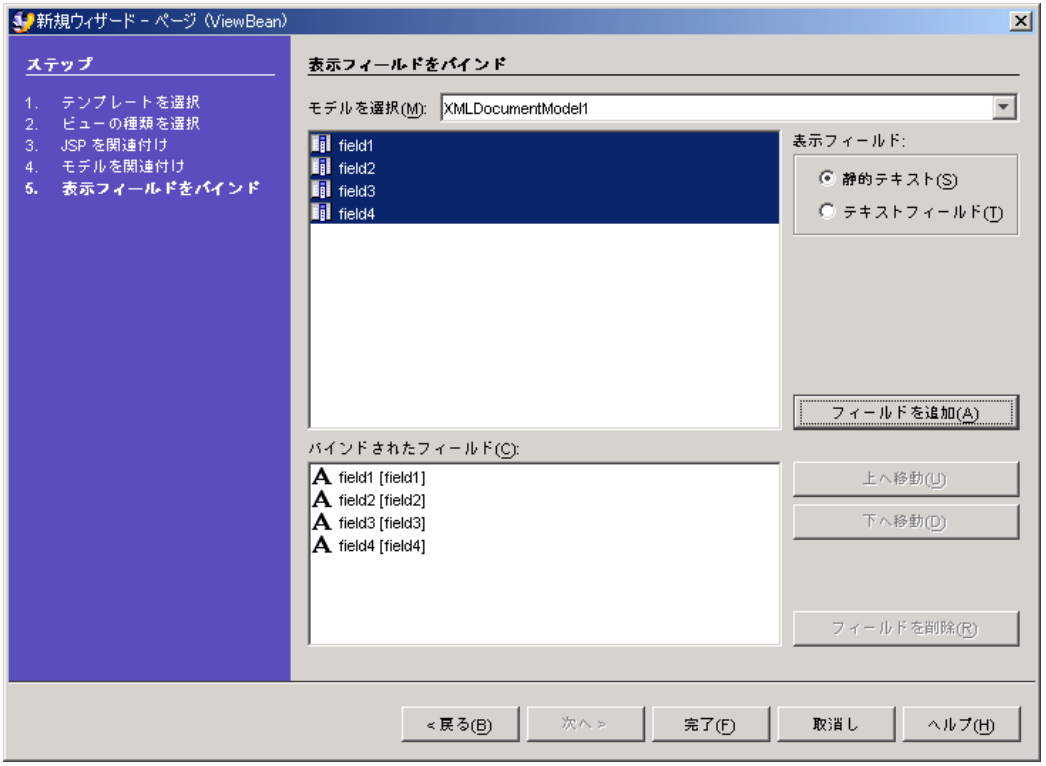
- 「次へ」をクリックします。



d. 「表示フィールドをバインド」パネル (下図を参照) で、4 つあるモデルフィールドすべてを選択し、「フィールドを追加」ボタンをクリックします。

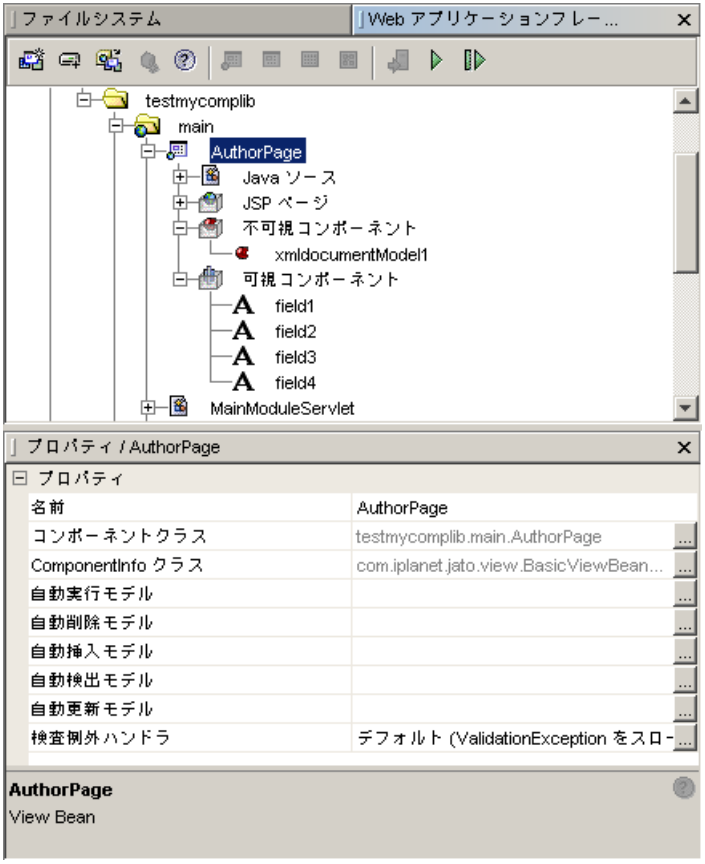
これで、パネルの「バインドされたフィールド」セクションに 4 つのエントリが現れます。

e. 「完了」をクリックします。



上記の方法でウィザードを終了すると、以下の図のような「AuthorPage」ノードが現れます。

子表示フィールドはそれぞれ、対応する XMLDocumentModel1 のフィールドに正しくバインドされています。



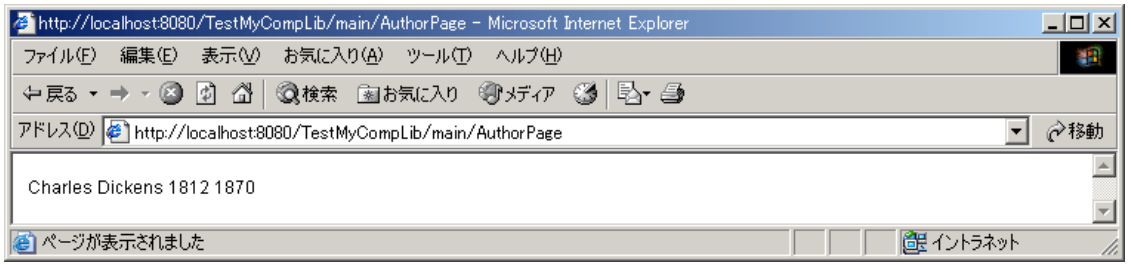
8. AuthorPage.java を開き、コンストラクタに次のコードを追加します。

   このコードはディスクから「author.xml」ドキュメントを読み取り、「authorDocument」という名前の要求スコープ (XMLDocumentModel1 がドキュメントが存在すると想定している場所) 属性に格納します。AuthorPage コンストラクタのこの位置にコードを置くという選択は、単純にテストの目的でそうしているにすぎません。前述したように、XMLDocumentModel は、モデルにアクセスするときに想定している場所に存在する限り、XML ドキュメントが合意された属性にいつ、どのようにして格納されるのか関知しません。先頭に非常に重要な文があることに注意してください。また、getResourceAsStream メソッドのパラメータは、たとえば `getResourceAsStream("/testmycomplib/main/author.xml")` などの、テスト用アプリケーション名を反映したパラメータを受け取る必要があります。

```
import org.w3c.dom.*;
import org.xml.sax.InputSource;
import javax.xml.parsers.DocumentBuilderFactory;

/**
 *
 *
 */
public class AuthorPage extends BasicViewBean
{
     /**
      * Default constructor
      *
      */
     public AuthorPage()
     {
          super();
          try {

               InputSource in = new InputSource(AuthorPage.class.
                    getResourceAsStream("/testmycomplib/main/author.xml"));
               DocumentBuilderFactory dfactory = DocumentBuilderFactory.newInstance();
               dfactory.setNamespaceAware(true);
               Document doc = dfactory.newDocumentBuilder().parse(in);
               doc.normalize(); // 必ずドキュメント内のテキストを正規形式にする
               RequestManager.getRequest().setAttribute("authorDocument", doc);
               System.out.println("Author XML Document has been put into request");
          }
          catch(Exception e) {
               System.out.println("Exception trying to load author.xml" + e);
          }
     }
```

9. アプリケーション内のすべてクラスをコンパイルし、AuthorPage をテスト実行します。

   次のようにブラウザに作家情報が表示されます。



10. タイルビューを作成します。

    XMLDocumentModel2 とそのデータセット機能をテストしてみます。

    このためには、タイルビューを作成する必要があります。本質的には、AuthorPage の作成の手順の繰り返しですが、今度は「基本 ViewBean」ではなく「基本タイルビュー」を選択し、そのビューを XMLDocumentModel1 ではなく、XMLDocumentModel2 に関連付けます。

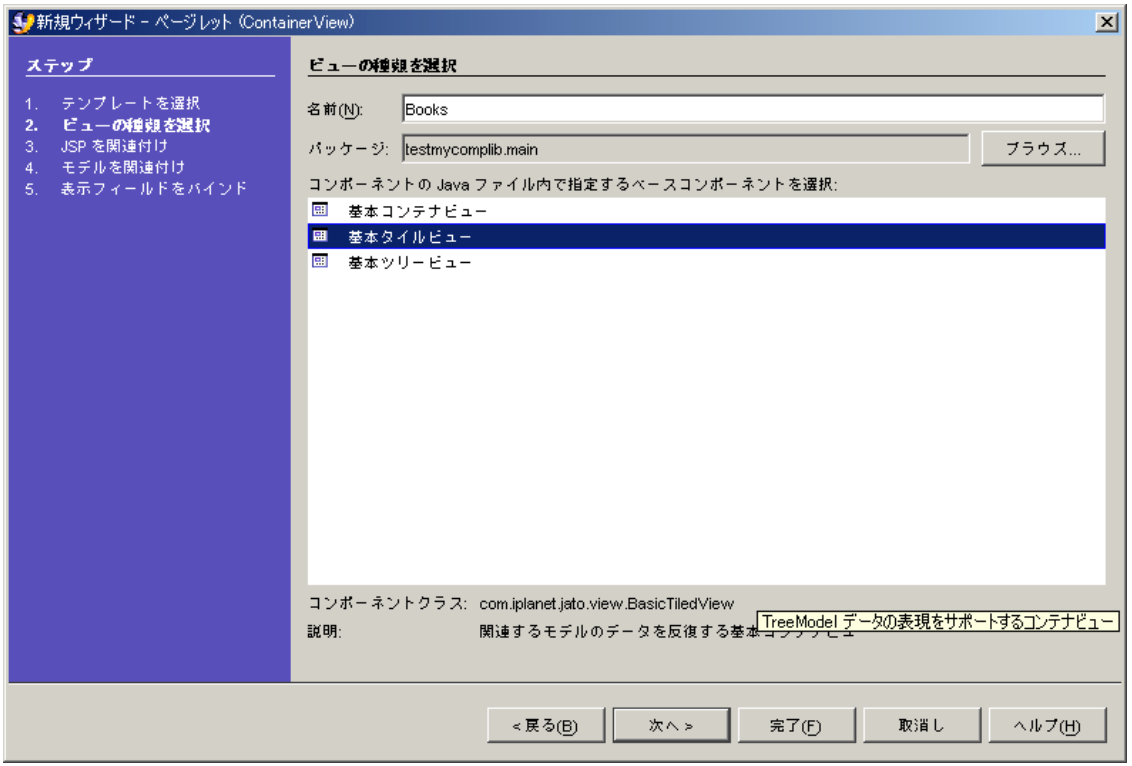
    詳細手順は以下のとおりです。

    a. 「新規ビュー」ウィザードを起動します。

    b. 「ビューの種類を選択」パネルで「基本タイルビュー」コンポーネントを選択します (下図を参照)。

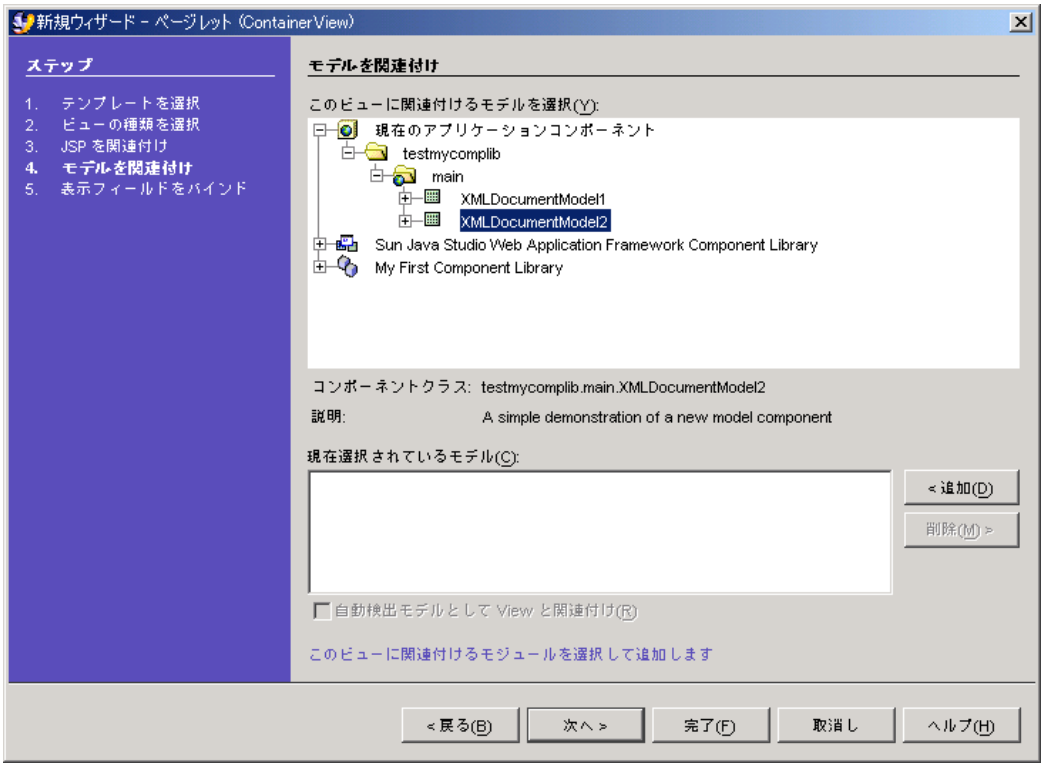
    c. 「名前」に「Books」を設定します。

    d. 「次へ」をクリックします。



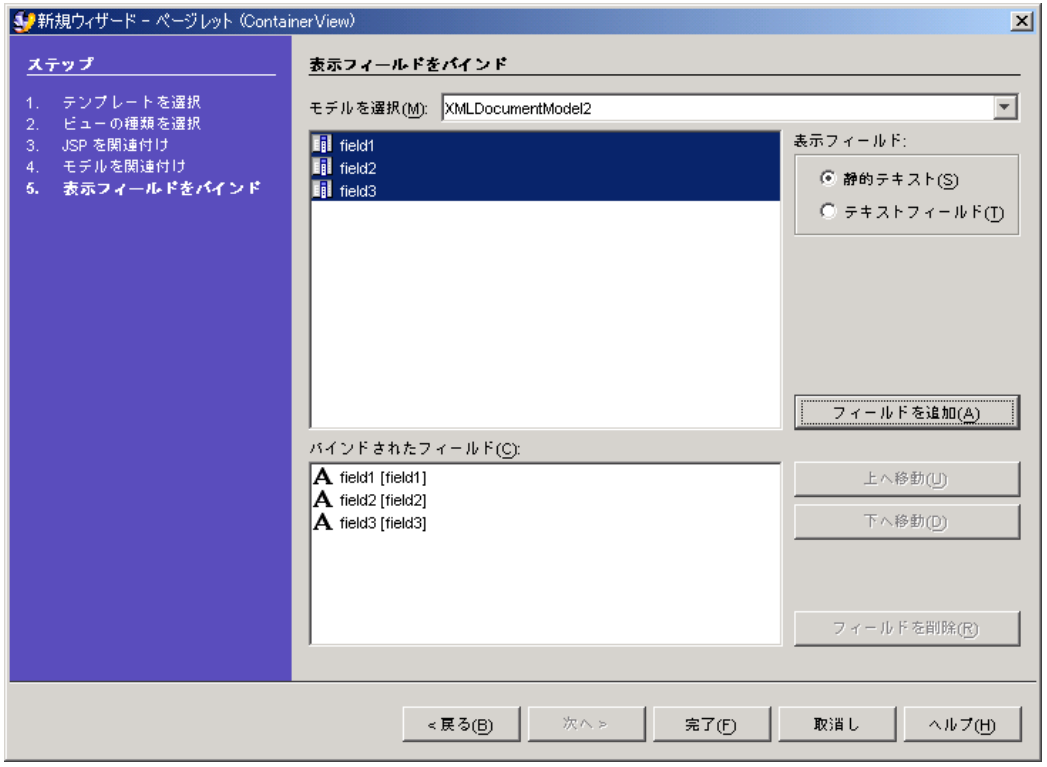
- 「JSP を関連付け」パネルでは、デフォルト値をそのまま使用し、「次へ」をクリックします。
- 「モデル関連付け」パネルで、最終的に「XMLDocumentModel2」が表示されるまで「現在のアプリケーションコンポーネント」ノードを展開します (下図を参照)。
- 「XMLDocumentModel2」を選択し、「追加」ボタンをクリックして、タイルビューとモデルの関連付けを作成します。

  これで、「XMLDocumentModel2」がパネルの「現在選択されているモデル」セクションに現れます。
- 「次へ」をクリックします。

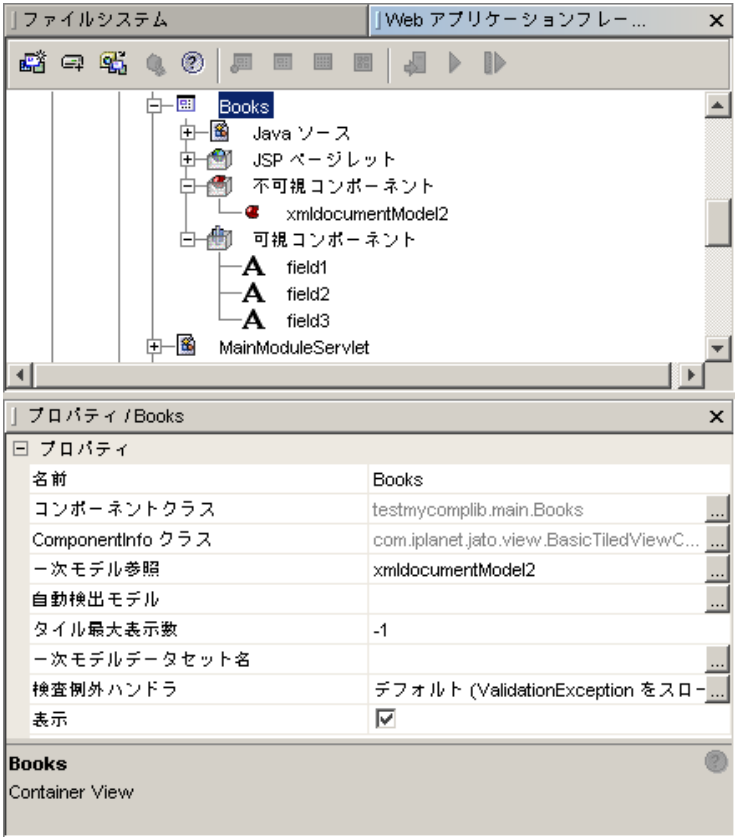


- 「表示フィールドをバインド」パネルで、3 つあるモデルフィールドすべてを選択し、「フィールドを追加」ボタンをクリックします (下図を参照)。

  これで、パネルの「バインドされたフィールド」セクションに 3 つのエントリが現れます。
- 「完了」をクリックします。



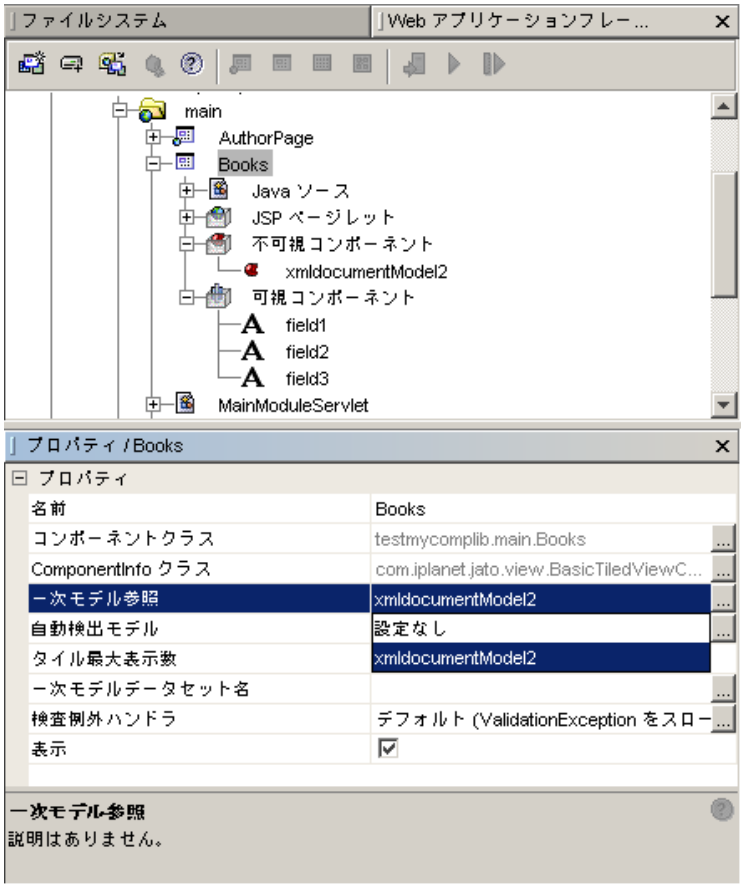
- 上記の方法でウィザードを終了すると、以下の図のような「Books」タイルビューノードが現れます。子表示フィールドはそれぞれ、対応する XMLDocumentModel2 のフィールドに正しくバインドされています。



11. タイルビューの場合は、このマニュアルではこれまでに説明していない追加の構成手順が 1 つ必要になります。

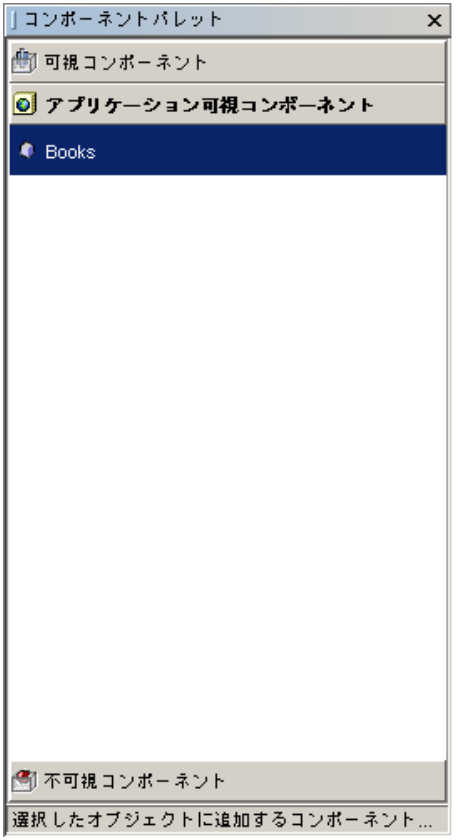
「XMLDocumentModel2」を参照するように、Book TiledView の「一次モデル参照」プロパティを設定する必要があります。このプロパティを設定しないと、このタイルビューをテスト実行しても、データが表示されません。この見落としは、Web アプリケーションフレームワークのアプリケーション開発者によく見られます。

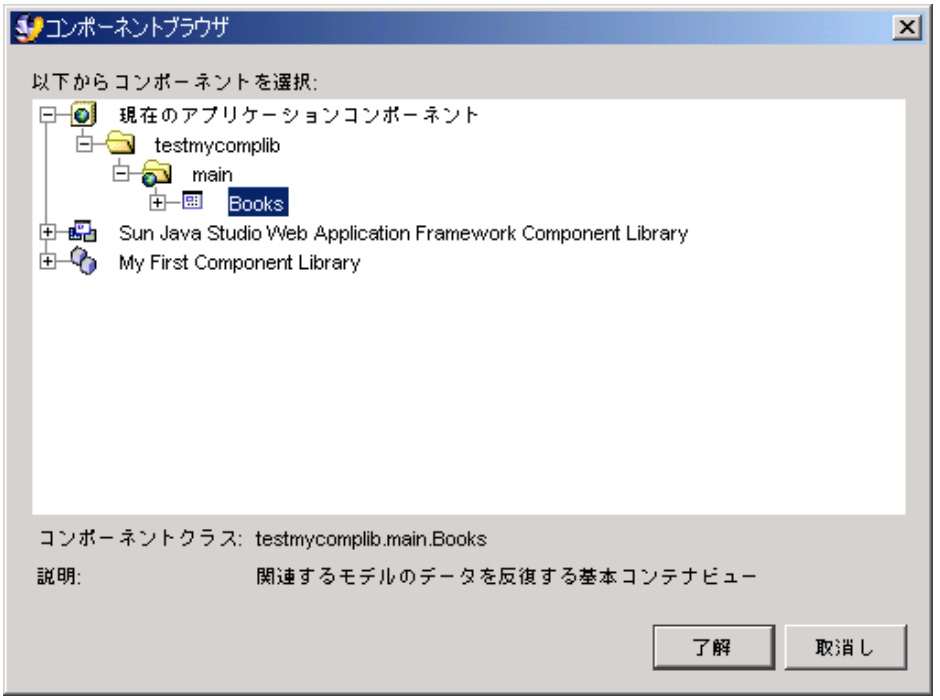
「一次モデル参照」プロパティを編集し、ドロップダウンリストから値「xmldocumentModel2」を選択します (下図を参照)。



12. Books タイルビューをテスト実行するには、ViewBean に追加する必要があります。例をもっと面白くするために、AuthorPage の子として Books タイルビューを追加します。これは、Web アプリケーションフレームワークの階層ビューサポートの好例になります。

    AuthorPage に Books タイルビューのインスタンスを追加します。このインスタンスの追加は、様々なテキストフィールドやボタンコンポーネントに対して前に行ったのと同じで、コンポーネントパレットまたはコンポーネントブラウザのいずれかから Books タイルビューを選択することによって行うことができます。今回を除いて、コンポーネントは、コンポーネントパレットまたはコンポーネントブラウザいずれかの「現在のアプリケーションコンポーネント」セクションに置かれます。


コンポーネントパレット
可視コンポーネント
アプリケーション可視コンポーネント
Books
不可視コンポーネント
選択したオブジェクトに追加するコンポーネント...



Books を子ビューとして追加する、「AuthorPage」ノードが以下のようになります。

13. 任意で AuthorPage に関連付けた JSP を書式化することができます。

    デフォルトでは、ViewBean にコンテナビューの子を追加すると、IDE によって、その ViewBean の JSP にその子と子のタグが追加されます。アプリケーション開発者は、JSP ノードで「ビューと同期」アクションを使用して、子のコンテナビューの子のタグをまとめて削除することができます。この機会を利用して、より整った形で生成されるように JSP に基本的な書式化設定も追加してみます。

    a. AuthorPage の「JSP」ノードを選択して、展開します。

    b. 実際の「AuthorPage.jsp」ノードを選択します。

    c. 「AuthorPage.jsp」ノードをダブルクリックして開き、エディタで編集できるようにします。

d. AuthorPage.jsp に基本的な書式化タグ (<p> や
 など) を追加し、著書データ行の間に区切りを入れます。

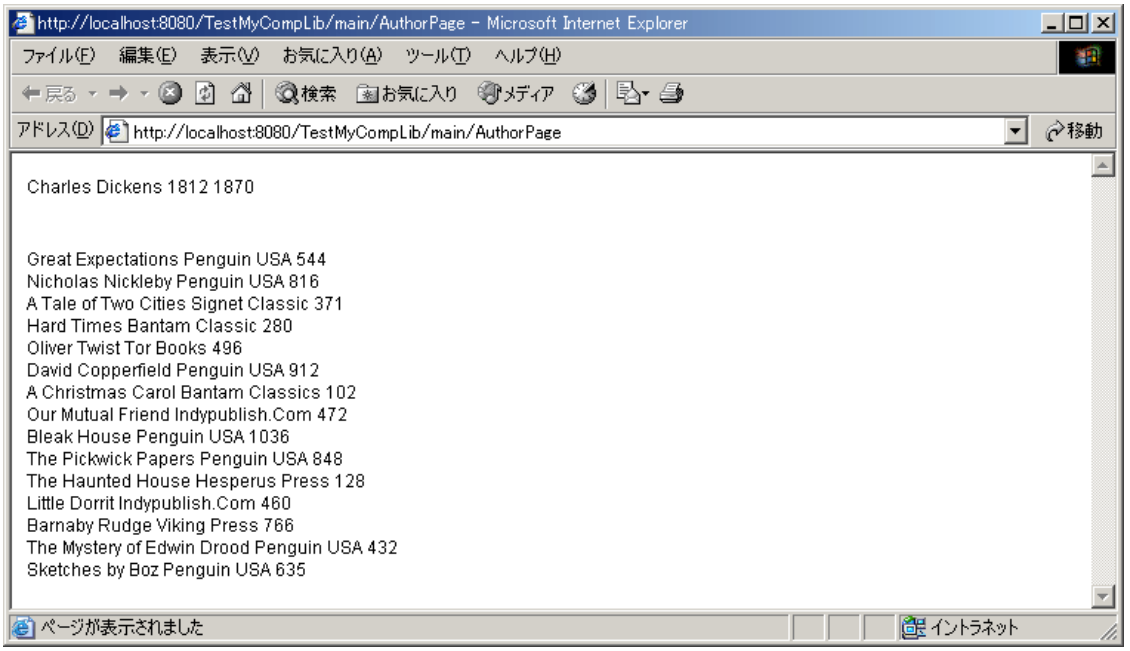
この編集を終えると、AuthorPage.jsp は以下のようになります。

```
<%@page contentType="text/html; charset=ISO-8859-1" info="AuthorPage" language="java"%>
<%@taglib uri="/WEB-INF/jato.tld" prefix="jato"%>

<jato:useViewBean className="testmycomplib.main.AuthorPage">

<html>
<head>
<title></title>
</head>
<body>

<jato:form name="Author" method="post">

<jato:text name="field1"/>
<jato:text name="field2"/>
<jato:text name="field3"/>
<jato:text name="field4"/>
<p>
<jato:tiledView name="books1">
<br>
<jato:text name="field1"/>
<jato:text name="field2"/>
<jato:text name="field3"/>
</jato:tiledView></jato:form>

</body>
</html>

</jato:useViewBean>
```

14. 再び AuthorPage をテスト実行します。

必ず「ページを実行」アクションではなく、「ページを実行 (再配備)」アクションを選択してください。最新の変更結果が表示されません。

ユーザーが必要なトークンを蓄積しているため、ブラウザに表示される AuthorPage.jsp の内容は下図のようになります。



配布

配布の前に行うことがまだあります。「Document Scope」プロパティ用のカスタムエディタを用意します。

raw int 値を表示して直接編集するようになっているため、現在、XMLDocumentModel の「Document Scope」プロパティがユーザーエラーの起きやすい状態になっています。実際に必要なのは、有効な選択肢だけを含むドロップダウンリストを提供するカスタムエディタです。時間を割いて、カスタムプロパティエディタを開発することを推奨します。このガイドでは詳細は説明しませんが、基本的な JavaBean のリファレンスに記載されています。

第5章

コマンドコンポーネントの開発

この章は、7 ページの「初めてのコンポーネントの開発」をすでに読み終えていることが前提になります。

拡張可能コマンドコンポーネントの開発

この章では、35 ページの「拡張不可ビューコンポーネントの開発」で作成した ValidatingDisplayField をさらに価値あるものにする、新しいコマンドコンポーネントの作成方法を説明します。

これは ValidatingCommand というコマンドコンポーネントであり、ValidatingDisplayField のインスタンスを含むページの処理に関係する再利用可能なロジックをカプセル化します。

この練習問題は、拡張可能コマンドコンポーネントデザインの仕組みを理解することを目的としているため拡張可能コマンドの可能性については表面をなぞるだけにします。

拡張可能コマンドの作成方法は非常に簡単です。これまでの章を終えていれば、これまでにその方法はすべて学んでいます。実際のコマンドは構造的に単純です。この章では、コマンドデザインパターンに関する知識を補強して、Web アプリケーションフレームワークにおけるその役割を紹介します。単純ではあるものの、強力なこのパターンを利用する際に想像力を発揮するのに役立ちます。

この例では、Web アプリケーションフレームワークコンポーネントモデルに関係する以下のような項目を取り上げます。

- CommandComponentInfo
- CommandDescriptor

検査コマンドコンポーネントには、次のデザイン時機能を持たせます。

- 開発者が ValidatingCommand をサブクラス化して、ベースクラスにカプセル化された動作に加えてアプリケーションに固有の動作を追加できるようにする。

具体的には、サブクラスの開発者は、従来の Command execute メソッドではなく、コンポーネントに固有の 2 つのメソッド、handleInvalid および handleValid に注力します。アプリケーションコマンド開発者は、コンポーネントのベースクラスに基づいて、コンポーネントが起動されたページが有効か無効かを判定し、適切なハンドラを起動します。このため、アプリケーションコマンド開発者は、有効または無効状態に対する応答に集中することができ、状態の検出について頭を悩ます必要はありません。

- 拡張可能コマンドコンポーネントは、ビューおよびモデルコンポーネントで行ったように、コンポーネント作成者が構成プロパティを指定することを可能にします。ただし、この例では何も定義する必要はありません。

検査コマンドコンポーネントには、次の実行時機能を持たせます。

ValidatingCommand のベースクラスに実装する実行メソッドは、送付された ViewBean に対して深い検索を行い、ValidatingTextField のすべてのインスタンスを検出して、それらのフィールドが無効かどうかを検査します。

- 無効なフィールドが見つかった場合、ValidatingCommand のベースクラスは handleInvalid メソッドを起動する。コマンドに固有の動作を行う場合は、ValidatingCommand が handleInvalid メソッドをオーバーライドすることが前提になります。ベースクラスに実装される handleInvalid は、単に無効なページを再表示します。一部のケースでは、この動作で十分と思われます。
- 無効なフィールドが見つからなかった場合、ValidatingCommand のベースクラスは handleValid メソッドを起動する。コマンドに固有の動作を行う場合は、ValidatingCommand が handleValid メソッドをオーバーライドすることが前提になります。ベースクラスに実装される handleValid は抽象メソッドです。このコマンドは、アプリケーション固有のコマンド開発者が、handleValid メソッドを実装することを前提にしています。

上記のコマンドコンポーネントを実装するという選択は、純粋にスタイルの問題です。前述のように、このコマンドパターンは初歩的なものです。このため、個人的なオブジェクト指向のスタイルは、コンポーネント作成者のデザインで大きな位置を占めます。

これらの要件を満たすには、次のクラスをデザイン、実装します。

- コンポーネントクラス - mycomponents.ValidatingCommand
- ComponentInfo クラス - mycomponents.ValidatingCommandComponentInfo

また、ここでは、ValidatingCommand のアプリケーションに固有のサブタイプの土台として IDE が使用するカスタム Java テンプレートも実装します。

最後に mycomponents の complib.xml を編集して、その新しいコンポーネントを Web アプリケーションフレームワークのコンポーネントライブラリに追加します。

この例では、mycomponents.ValidatingDisplayField の共存が前提になります。まだ、35 ページの「拡張不可ビューコンポーネントの開発」を終えていない場合は、次に進む前に終えておいてください。

Web アプリケーションフレームワークコンポーネントクラスの作成

1. 任意の Java エディタで mycomponents.ValidatingCommand クラスを作成します。
2. ValidatingCommand に com.iplanet.jato.view.BasicCommand を拡張させます。
3. コンポーネントの種類に合わせて適切なコンストラクタを実装します。

   あらゆるコマンドコンポーネントは、引数なしのコンストラクタを実装する必要があります。
4. コンポーネントに固有の要件を満たすために必要な残りのメソッドを実装します。
   - 実行メソッドの実装 - コンポーネントの妥当性検査ロジックを適用します。
   - コンポーネントの handleInvalid メソッドのデフォルト実装
   - コンポーネントの handleValid メソッドの抽象宣言

   これまでの手順を終えると、mycomponents/ValidatingCommand.java は以下のようになります。

```
package mycomponents;
import java.util.*;
import com.iplanet.jato.*;
import com.iplanet.jato.command.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.event.*;


public abstract class ValidatingCommand extends Object implements Command {


    public ValidatingCommand() {
        super();
    }


    public void execute(CommandEvent event) throws CommandException {
        Map map=event.getParameters();

        try {
            boolean isValid = true;
            ViewBean viewBean =  ViewBase.getRootView((View)event.getSource());
            List vFields = getValidatingTextChildren(viewBean);
            Iterator iter = vFields.iterator();
```

```
            while (iter.hasNext()) {
                ValidatingDisplayField vText = (ValidatingDisplayField)iter.next();
                if(! vText.isValid()) {
                    isValid = false;
                    break;
                }
            }

            if( isValid ) {
                handleValid(event);
            }
            else {
                handleInvalid(event, viewBean);
            }

        }
        catch(Exception e) {
            if (e instanceof CommandException)
                throw (CommandException)e;
            else {
                throw new CommandException(
                    "Error executing ValidatingCommand",e);
            }
        }
    }


    public List getValidatingTextChildren(ContainerView container) {
        List result=new LinkedList();

        String[] childNames=container.getChildNames();
        for (int i=0; i<childNames.length; i++) {
            Class childType=container.getChildType(childNames[i]);
            if (ValidatingDisplayField.class.isAssignableFrom(childType)) {
                ValidatingDisplayField child=(ValidatingDisplayField)
                container.getChild(childNames[i]);
                result.add(child);
            }
            else if (ContainerView.class.isAssignableFrom(childType)) {
                ContainerView child=
                    (ContainerView) container.getChild(childNames[i]);
                result.addAll(getValidatingTextChildren(child));
            }
        }
        return result;
    }


    public abstract void handleValid(CommandEvent event) throws CommandException;
```

```
    public void handleInvalid(CommandEvent event, ViewBean invalidVB)
        throws CommandException {
        // デフォルト実装は無効なページを再表示するだけ
        invalidVB.forwardTo(event.getRequestContext());
    }

}
```

拡張可能コンポーネントの Java テンプレートの作成

拡張可能コンポーネントは、アプリケーション定義のエンティティに対するベースクラスの働きをします。このため、Web アプリケーションフレームワークコンポーネントモデルは、拡張可能コンポーネントの作成者がカスタム Java テンプレートを提供する機会を提供します。

IDE では、そのコンポーネント提供のテンプレートを利用して、アプリケーション固有のサブタイプを作成します。コンポーネント作成者は、そのカスタムテンプレートを利用して、アプリケーション開発者の作業をさらに充実したものにすることができます。コンポーネント作成者は、`com.iplanet.jato.component.ExtensibleComponentInfo` に定義されている一群のテンプレートトークンを使ってコンポーネント固有の Java テンプレートを用意することができます。トークンの詳細は、ExtensibleComponent API を参照してください。

コンポーネント作成者はまた、Java テンプレート内で任意の Java 構造 (インポート文、メソッド、変数、インタフェース宣言など) を利用することもできます。このカスタムテンプレートは、最低限、拡張可能コンポーネントクラスから新しい Java クラスを拡張する保証をします。

この例のテンプレートは、ベースクラスで抽象宣言するメソッドの実装に際して開発者の役に立ちます。

任意のテキストエディタで `mycomponents.resources.ValidatingCommand_java.template` テンプレートを作成します。

テンプレートの内容は、以下のようにします。

注 – a __TOKEN__ pattern の後にトークンが続きます。

```
package __PACKAGE__;

import com.iplanet.jato.*;
import com.iplanet.jato.command.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.event.*;
import mycomponents.*;

/**
 *
 *
 */
public class __CLASS_NAME__ extends ValidatingCommand
{
     /**
      * Default constructor
      *
      */
     public __CLASS_NAME__()
     {
          super();
     }

     /**
      *
      *
      */
     public void handleValid(CommandEvent event) throws CommandException
     {
          // TODO - このメソッドは開発者が実装する必要がある
     }

}
```

ComponentInfo クラスの作成

ComponentInfo クラスは、IDE がコンポーネントを組み込むのに必要なデザイン時メタデータを定義します。この例では、既存の ComponentInfo を拡張し、本当のオブジェクト指向のスタイルで単にこのクラスを補強します。当然、ComponentInfo インタフェースを最初から実装する方法を選ぶこともできますが、この例では生産的ではありません。

この例では、前述のコンポーネント例で明らかにした機能をさらに発展させることはありません。

1. mycomponents.ValidatingCommandComponentInfo クラスを作成します。
2. ValidatingCommandComponentInfo に com.iplanet.jato.view.BasicCommandComponentInfo を拡張させます。
3. 引数なしのコンストラクタを実装します。

   新しいプロパティを定義する必要はないため、getComponentDescriptor() メソッドを実装する必要はありません。
4. Java テンプレートを実装する getPrimaryTemplateAsStream() メソッドを実装します。IDE は、この拡張可能コンポーネントから派生する新しいクラスに対する出発点として、この Java テンプレートを使用します。

   これまでの手順を終えると、mycomponents/ValidatingCommandComponentInfo.java は以下のようになります。

注 – このコードでは、例として紹介しやすいよう、String 値を直接埋め込んでいます。表示文字列を各言語対応にする必要があると思われる場合は、リソースバンドルを利用してください。

```
package mycomponents;
import java.util.*;
import java.awt.Image;
import java.io.*;
import com.iplanet.jato.component.*;
import com.iplanet.jato.command.*;


public class ValidatingCommandComponentInfo extends BasicCommandComponentInfo  {


    public ValidatingCommandComponentInfo()
    {
        super();
    }


    public ComponentDescriptor getComponentDescriptor()
    {
        // コンポーネントクラスを示す
        ComponentDescriptor result=new ComponentDescriptor(
            "mycomponents.ValidatingCommand");

        // この名前が、コンポーネントのインスタンスの名前を決めるのに使用される
        result.setName("ValidatingCommand");

        // この表示名が、コンポーネント選択で使用される
```

```
            result.setDisplayName("Validating Command");

            // この説明がコンポーネントのツールチップのテキストになる
            result.setShortDescription("A validating command component");

            return result;
        }


        public String getPrimaryTemplateEncoding()
        {
/* 本稼働のバージョンは次のようにリソースバンドル駆動型
return getResourceString(getClass(),
"PROP_ValidatingCommand_SOURCE_TEMPLATE_ENCODING", "ascii");
*/

            return "ascii"; // NOI18N
        }


        public InputStream getPrimaryTemplateAsStream()
        {
/* 本稼働のバージョンは次のようにリソースバンドル駆動型

return ValidatingCommandComponentInfo.class.getClassLoader().
getResourceAsStream(
getResourceString(getClass(),
"RES_ValidatingCommandComponentInfo_SOURCE_TEMPLATE",""));
*/

            return mycomponents.ValidatingCommandComponentInfo.class.getResourceAsStream(
            "/mycomponents/resources/ValidatingCommand_java.template"); // NOI18N
        }

}
```

コンポーネントライブラリのマニフェストの補強

コンポーネントマニフェストは、前述の例ですでに作成しています。ここでは、追加の情報を追加します。

前の例では現れなかった追加の種類の情報を追加することに注意してください。

Web アプリケーションフレームワークのライブラリマニフェスト名は、`complib.xml` である必要があります。Web アプリケーションフレームワークのライブラリマニフェストは、JAR ファイル内の `/COMP-INF` ディレクトリに入れる必要があります。

1. `COMP-INF/complib.xml` ファイルを作成するか、開きます。

2. ValidatingCommand コンポーネントを宣言する拡張可能コンポーネント要素を追加します。

   これまでの手順を終えると、COMP-INF/complib.xml は以下のようになります。

注 – 見やすくするために、ファイルの冒頭にあるバージョンに対する重要なデルタ部分だけ示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<component-library>
        <tool-info>
                <tool-version>2.1.0</tool-version>
        </tool-info>
        <library-name>mycomponents</library-name>
        <display-name>My First Component Library</display-name>

        ...

    <extensible-component>
        <component-class>mycomponents.ValidatingCommand</component-class>
        <component-info-class>mycomponents.ValidatingCommandComponentInfo</component-info-
class>
    </extensible-component>

        ...

</component-library>
```

コンポーネントライブラリの JAR ファイルの再作成

最初の例で行ったようにコンポーネントクラスを JAR にまとめて、1 つのライブラリとして配布できるようにします。

1. この JAR ファイルの名前は任意です。

   この場合は、「mycomponents.jar」という名前にします。

2. 必ずしも Java ソースファイルを JAR に含める必要はありません。

3. **アイコンイメージやリソースバンドルなどの必要な補助リソースを JAR に含めます。**

   この例では、そうしたリソースはありません。

   この例では、いくつかの新しいクラスと、Java テンプレートファイル 1 つを含めます。

4. **`mycomponents.jar` の内部構造は以下のようになります。**

```
mycomponents/resources/SecureViewBean_java.template
mycomponents/resources/ValidatingCommand_java.template
mycomponents/resources/XMLDocumentModel_java.template
mycomponents/MissingTokensEvent.class
mycomponents/MyTextField.class
mycomponents/MyTextFieldComponentInfo.class
mycomponents/SecureViewBean.class
mycomponents/SecureViewBeanComponentInfo.class
mycomponents/TypeValidator.class
mycomponents/ValidatingCommand.class
mycomponents/ValidatingCommandComponentInfo.class
mycomponents/ValidatingDisplayField.class
mycomponents/ValidatingTextFieldComponentInfo.class
mycomponents/ValidatingTextFieldTag.class
mycomponents/Validator.class
mycomponents/XMLDocumentModel.class
mycomponents/XMLDocumentModelComponentInfo.class
mycomponents/XMLDocumentModelFieldDescriptor.class
mycomponents/mycomplib.tld
COMP-INF/complib.xml
```

新しいコンポーネントのテスト

この項では、Web アプリケーションフレームワークの IDE のその他の機能を紹介します。

- 新規コマンドウィザード
- コマンド記述子プロパティエディタ

1. 以前に作成したテスト用アプリケーションに新しいバージョンのライブラリを配備します。

   IDE に関する重要な注：IDE で、現在マウントされている JAR ファイルを削除したり、上書きコピーしたりすることはできません。

   現在マウントされている JAR ファイルを置き換える必要がある場合は、必ず、IDE を停止してください。IDE ですでに開かれているプロジェクトで新しいバージョンのコンポーネントライブラリをテストするには、まず、IDE を停止します。

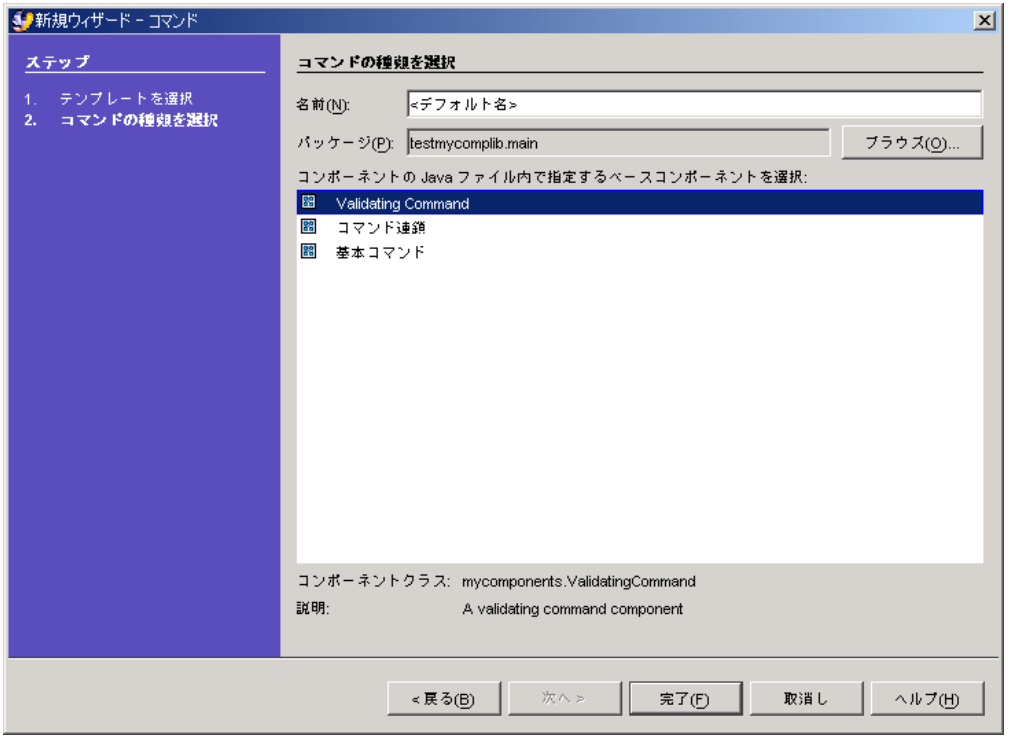
   IDE によって古いバージョンのライブラリ JAR ファイルが解放されると、新しいバージョンの JAR ファイルを古いバージョンに上書きコピーすることができます。

   新しいバージョンのライブラリを配備すると、再び IDE でアプリケーションを開くことができます。

2. 新しいコマンドオブジェクトを作成します。

注 – 『Web アプリケーションフレームワークチュートリアル』をまだ完了していていない場合は、終えておいてください。

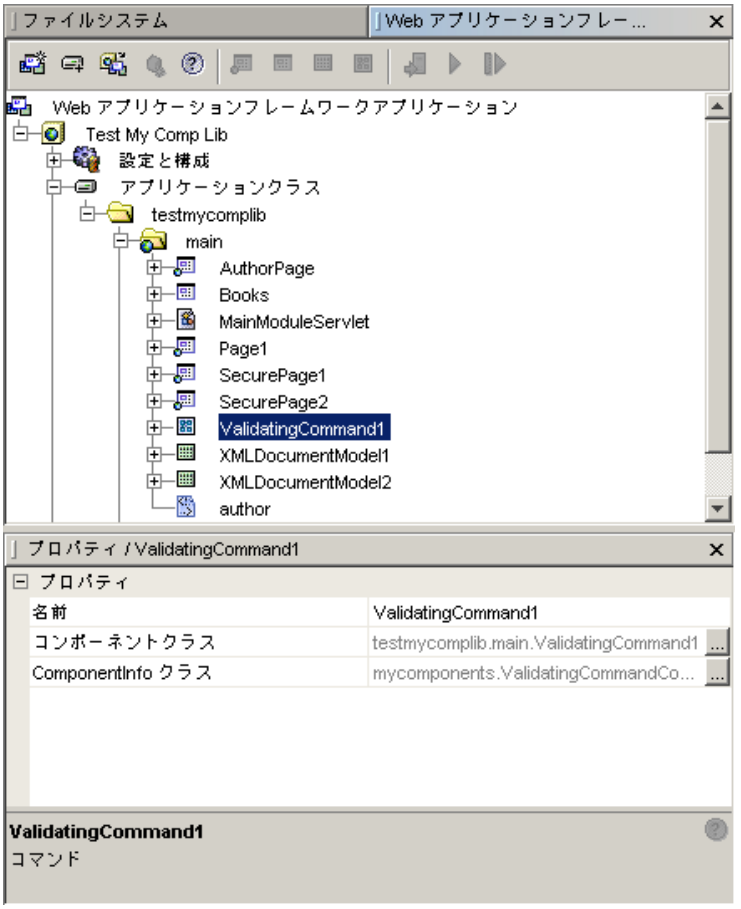
以下は、「新規コマンド」ウィザードの画面例です。



3. コンポーネントリストから「Validating Command」を選択して、ウィザードを完了します。
4. デフォルトの設定をそのまま使用して、ウィザードに ValidatingCommand1 の作成を任せてください。

   ウィザードが終了すると、IDE によって、コンポーネント提供のテンプレートに基づく新しいクラスが作成されます。

   これは、アプリケーションに ValidatingCommand1 オブジェクトが含まれます。



以降の手順では、新しい ValidatingCommand のテストに使用できる 2 つのページがテスト用アプリケーションに含まれていることが前提になります。

必要なこの 2 つのページは以下のとおりです。

- 35 ページの「拡張不可ビューコンポーネントの開発」の Page1

- 35 ページの「拡張不可ビューコンポーネントの開発」の SecurePage1

これらの ViewBean は、上記のテスト用アプリケーションのエクスプローラ画面で見ることができます。

最初に、ValidatingCommand1 のコードを完成する必要があります。

デザインでは、スーパークラスが検査情報の検出を行う一方、アプリケーション固有のクラス (ValidatingCommand1 など) は送付ページが有効なときに何をするかを決定します。

これは、アプリケーションに固有の決定です。コンポーネントを簡単にテストできるよう、単に SecurePage1 を表示する `handleValid` メソッドのコーディングをします。

a. ValidatingCommand1 の Java ソースを開きます。

b. `handleValid` メソッドを実装します。

```
public void handleValid(CommandEvent event) throws CommandException
    {
         ViewBean next = event.getRequestContext().getViewBeanManager().getViewBean(
              SecurePage1.class);
         next.forwardTo(event.getRequestContext());
    }
```

特に ValidatingCommand をテストするには、ValidatingDisplayField をいくつか含む ViewBean が必要です。

そうした ViewBean は、Page1 という名前ですでに作成しています。

5. 「Page1」ノードを選択します。

コマンドコンポーネントをテストするには、コマンドクライアント (ボタンや HREF のような CommandField など) とコマンドオブジェクト間の使用関係を確立する必要があります。

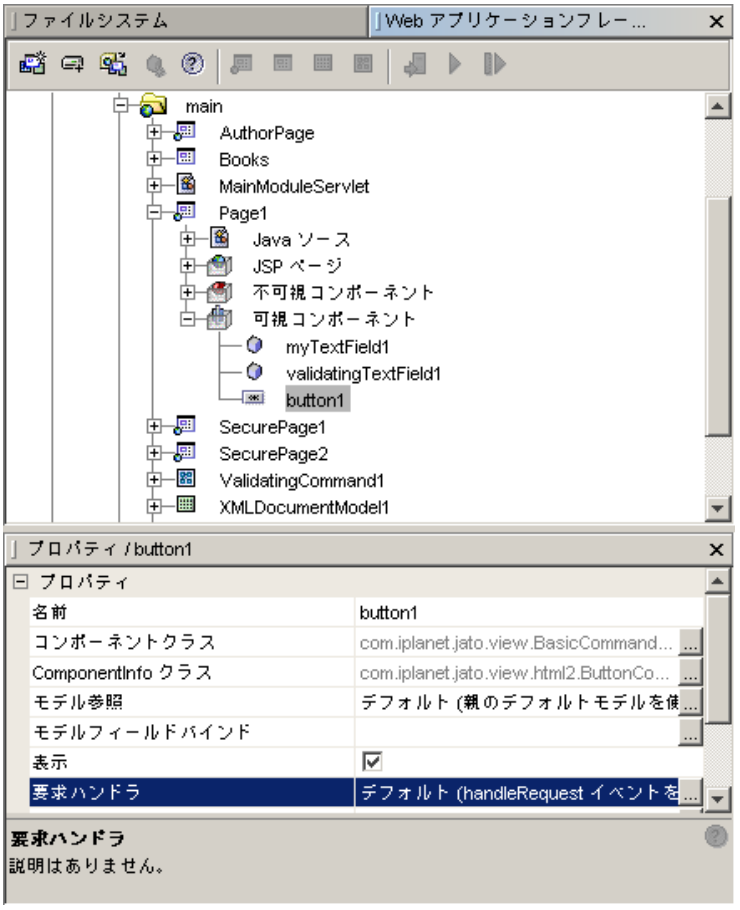
実行時、CommandField はコマンドオブジェクトを使用 (起動) します。この使用関係は、デザイン時に、コマンドインスタンスを宣言的に記述するように CommandDescriptor を構成し、その CommandDescriptor を CommandField に関連付けることによって確立します。

実際には、IDE は、この手順を比較的簡単に行えるようにします。IDE では、開発者はまず CommandField を選択することによって、この複数手順の構成を開始することができます。その後は、CommandField の構成の一環として CommandDescriptor の構成が順を追って自動的に行われます。

学ぶこととは実際にやってみること。以下の手順に従ってみてください。

a. 「可視コンポーネント」サブノードを選択します。

b. 「button1」ノードを選択します (下図を参照)。

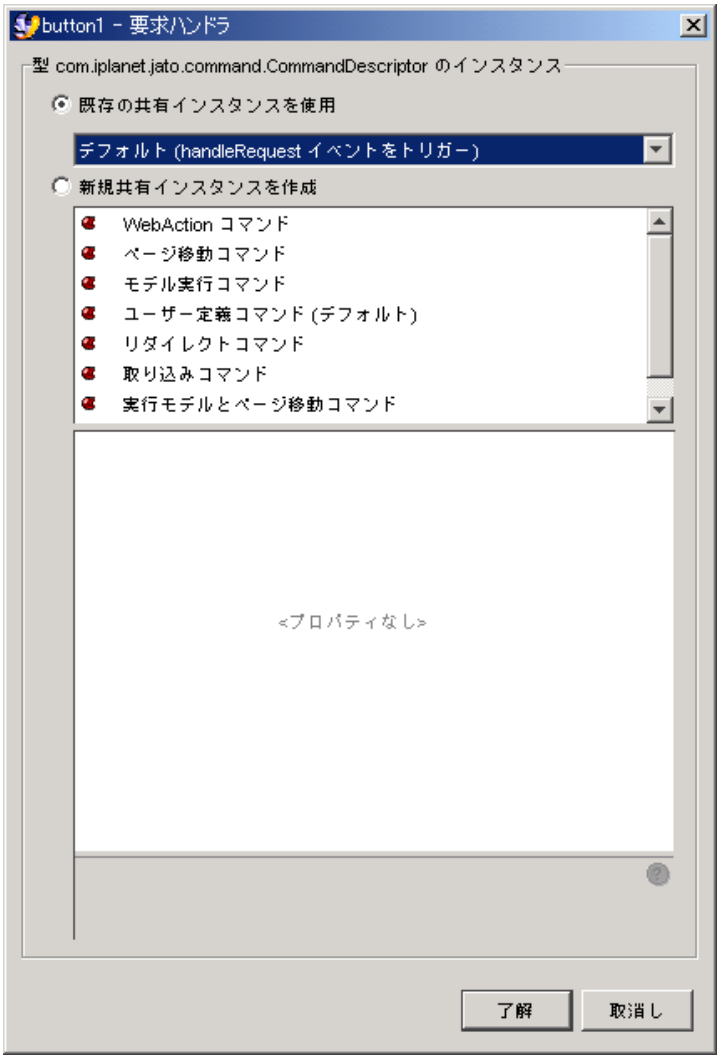


6. 「button1」の「要求ハンドラ」プロパティを編集します。

   a. プロパティの省略符号 (...) ボタンをクリックして、コマンド記述子プロパティエディタを全面表示します。

      これは高機能のエディタで、慣れるには努力が必要ですが、後述する役立つ機会がさらに得られるようになります。

      最初に、エディタを快適に使えるようになる必要があります。コマンド記述子プロパティエディタには、使用可能な CommandDescriptor の型の動的一覧が含まれ、選択された CommandDescriptor の型のプロパティを動的に表示する埋め込みプロパティシート (エディタの下部) も含まれています。このことは、この後の手順で分かります。



7. 「新規共有インスタンスを作成」ラジオボタンを選択します。

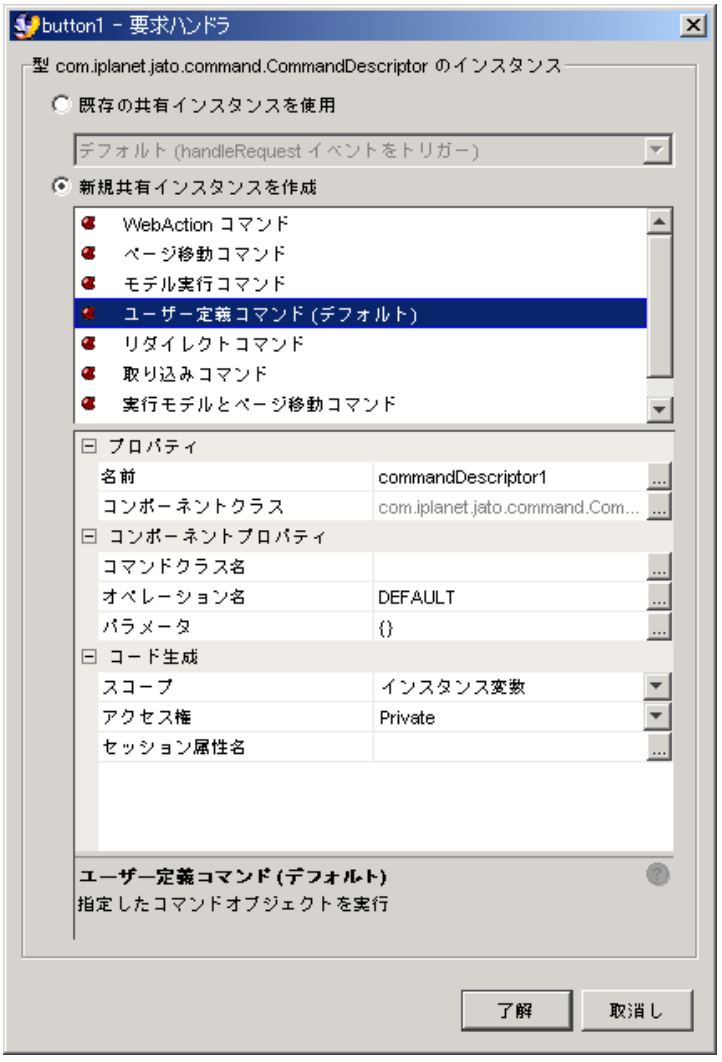
注 – 「共有インスタンス」の意味は、後ほど説明します。

---

8. **使用可能な記述子の型一覧から「ユーザー定義コマンド (デフォルト)」項目を選択します (上記の図を参照)。**

   コマンド記述子の種類を選択すると、エディタ下部に、その記述子の型に固有のプロパティが表示されます。

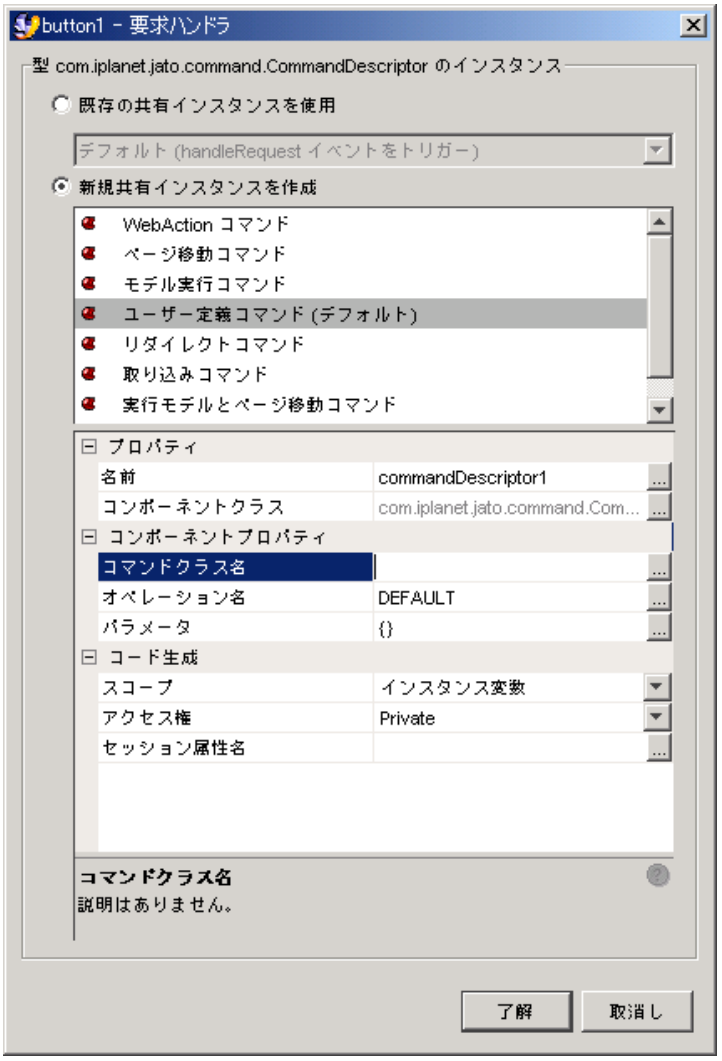
   プロパティエディタは以下のようになります。



以下のように、埋め込みプロパティシートには 3 つのセクションがあります。

- プロパティ
- コンポーネントプロパティ
- コード生成

9. 「コンポーネントプロパティ」セクションを選択します。
10. 「コンポーネントプロパティ」セクションで「コマンドクラス名」プロパティを選択します (下図を参照)。



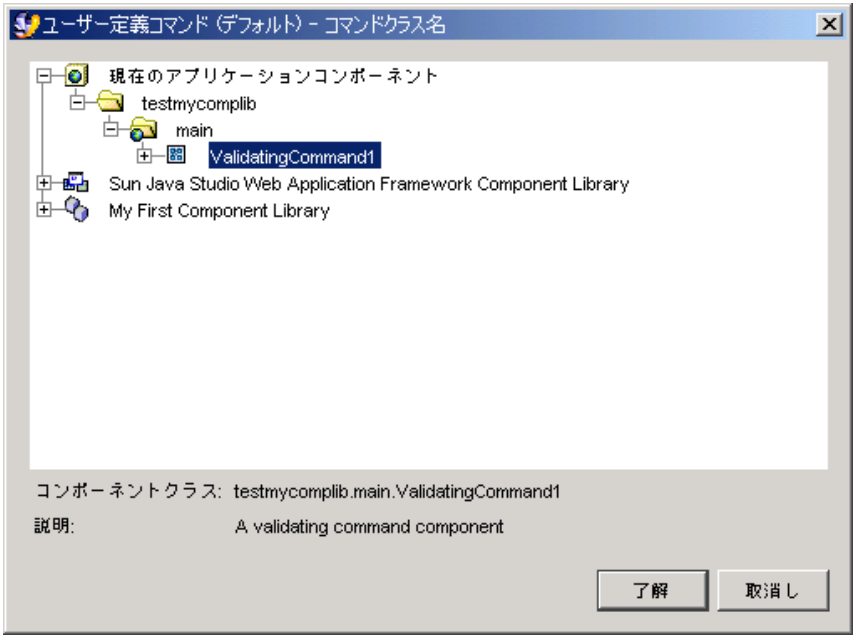
「コマンドクラス名」プロパティは `java.lang.String` 型です。

11. **表示されたプロパティフィールドに直接入力する代わりに、省略符号 (...) ボタンをクリックして、エディタを全面表示します。**

    実際には、「コマンドクラス名」プロパティエディタは他のいくつかのコンテキストで見た拡張不可コンポーネントブラウザと同じであることが分かります。

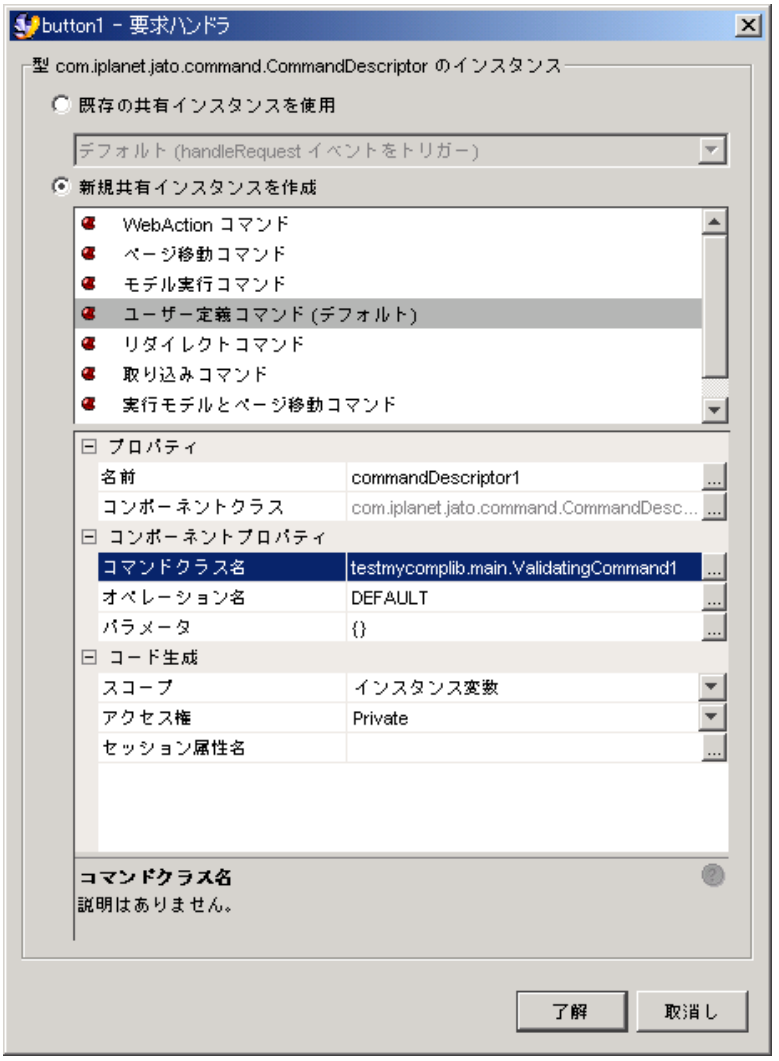
    ただし、このコンテキストでは、コンポーネントの一覧は、現在のコンテキストに適切なコンポーネントだけ表示されるように絞り込まれています。

12. **「現在のアプリケーションコンポーネント」ノードを完全に展開します。「ValidatingCommand1」があることが分かります (下図を参照)。**



13. プロパティエディタから「ValidatingCommand1」ノードを選択し、「了解」をクリックします (上図を参照)。

    CommandDescriptor の埋め込みプロパティシートの「コマンドクラス名」プロパティに、ValidatingCommand1 の完全限定クラス名が含まれていることに注目してください (下図を参照)。



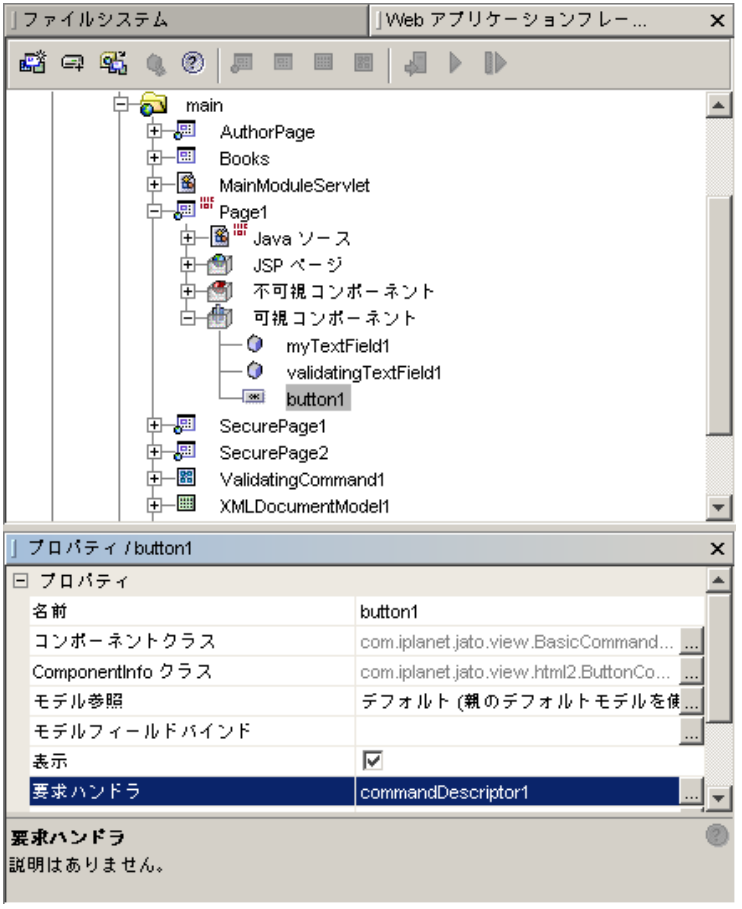
注 – 「コマンドクラス名」プロパティには、<テストするアプリケーション>.main.ValidatingTest1 というように、コマンドクラスの完全限定名を直接入力することもできます。ただし、「コマンドクラス名」プロパティの直接編集は、たとえば、.class ファイルとしてのみ存在していて、上図のようにコマンドコンポーネントブラウザに表示されず、直接選択することができないコマンドクラスを参照する必要がある場合など、特殊な場合にだけ行うことを推奨します。

---

14. 「了解」をクリックして、CommandDescriptor の構成を終了します。

次の構成に進む前に、直前の構成が Page1 に与える影響を完全に理解しておいてください。

button1 の「要求ハンドラ」プロパティの値が「commandDescriptor1」になっていることに注意してください (下図を参照)。



commandDescriptor はどこにあるのかという疑問が湧くかもしれません。

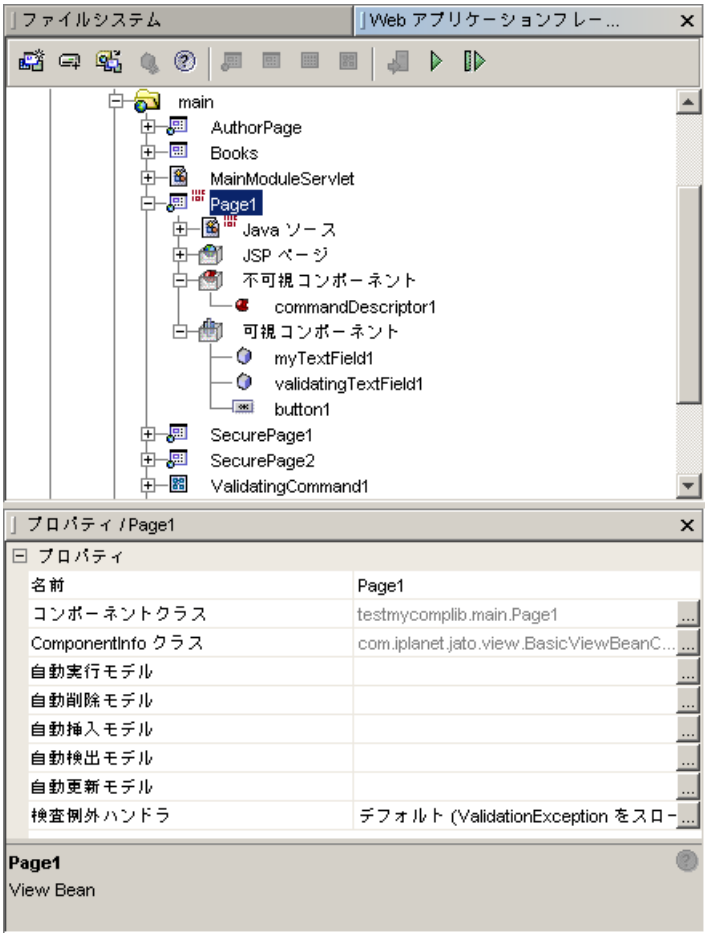
CommandDescriptor プロパティエディタで選択した「新規共有インスタンスを作成」ラジオボタンに対する疑問もあります (上図を参照)。

これらの疑問に対する答えは以下の手順で得られます。

15. Page1 の「不可視コンポーネント」ノードを展開します (下図を参照)。

「commandDescriptor1」という新しい不可視コンポーネントがあることが分かります。これが、少し前に構成した CommandDescriptor オブジェクトです。「共有インスタンス」として構成した CommandDescriptor です。

「共有」ということがどういうものか視覚的に表すために、IDE では、「不可視コンポーネント」ノードを提供し、そこに共有インスタンスのすべてを収容します。



「不可視コンポーネント」ノードは、Web アプリケーションフレームワークのコンポーネントモデル構造です。「不可視コンポーネント」は、現在のコンテナビュー内の別の場所にあるプロパティから参照可能な JavaBean オブジェクトの構成 (構成済み Bean など) にコンテナビューのスコープ空間を提供します。

この例の「commandDescriptor1」は、button1 の「要求ハンドラ」プロパティによって参照される構成済みの CommandDescriptor (JavaBean) です。このコンポーネントモデル機能の特徴は、構成済みの 1 つの不可視コンポーネントが、現在のコンテ

ナビューのスコープ内の複数のプロパティから参照可能であることにあります。たとえば、複数のボタンまたは HREF の CommandDescriptor プロパティが commandDescriptor1 を参照するように設定することも出来ます。

Page1 用に IDE が生成した Java コードを見ると、そのページが Java でどのように表現されているのかがすぐに分かります。コンポーネント作成者とアプリケーション開発者両方にとって、この利点は重要です。さらに説明すると、複数のコマンドフィールド (ボタン、HREF など) がそれぞれにその専用の「コマンド記述子」プロパティ内で参照することによって commandDescriptor1 を共有できるばかりでなく、現在のコンテナビュー内にあって、
`com.iplanet.jato.command.CommandDescriptor` から割り当て可能な型を持つ任意のプロパティが commandDescriptor1 を参照するように設定することもできます。本質的には不可視コンポーネントは、同じクラス内にある任意の個数の他の可視コンポーネントから、型に関して安全な方法で参照することが可能な、IDE で構成可能なクラススコープの JavaBean オブジェクトです。

現時点では、IDE は、不可視コンポーネントが別の不可視コンポーネントを参照する機能をサポートしていません。

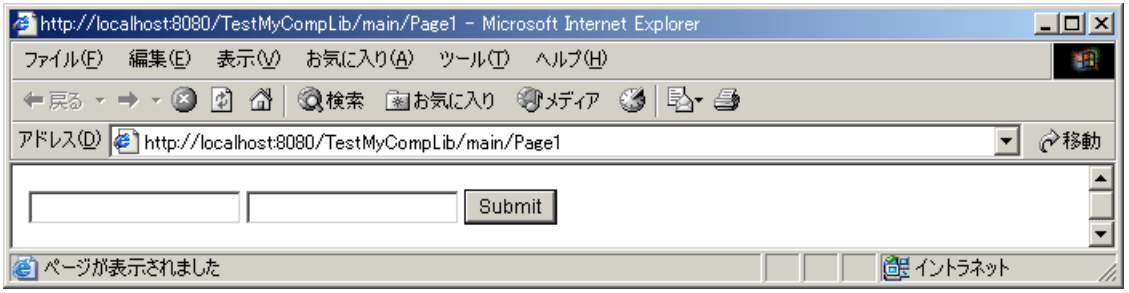
アプリケーション開発者は、不可視コンポーネントの共有的な性質を理解し、重んじる必要があります。既存の不可視コンポーネントの構成を変更すると、実行時にその不可視コンポーネントのインスタンスが参照される、あらゆる環境が間接的な影響を受けます。これは、CommandDescriptor エディタ (もっと広義には、不可視コンポーネントエディタ) が常に、不可視コンポーネントの「新規共有インスタンスの作成」を可能にしている理由です。たいていの場合、1 つのコンテナビュー内の複数のコマンドフィールドが同じ CommandDescriptor インスタンスを共有することはなく、むしろ、CommandDescriptor の別の明確に異なる構成のインスタンスを参照します。

これで、フォーム送付で button1 が指示された場合は必ず ValidatingCommand1 をインスタンス化して、その実行メソッドを起動するように Page1 を構成したことになります。

16. **ValidatingCommand1 で制御するようにした、Page1 から SecurePage1 のページフローをテスト実行します。**

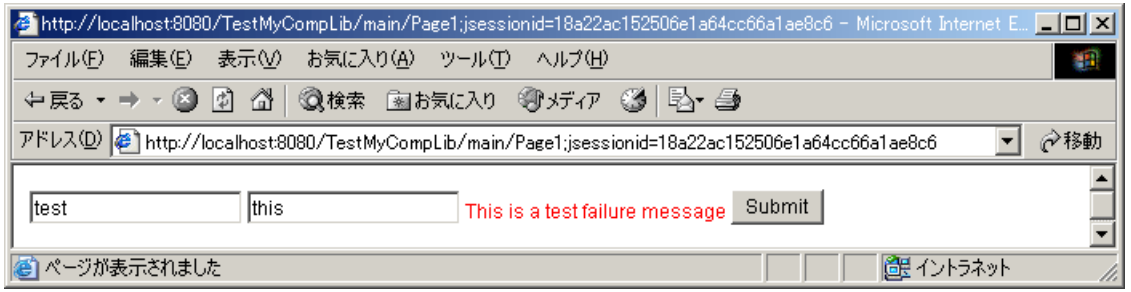
    Page1 のテスト実行

    ブラウザに `Page1.jsp` の内容が表示されます。



17. 「ValidatingTextField」テキストフィールドに不正な値 (任意の非整数) を入力して、ページを送付します。

    ValidatingTextField のすぐ後に「Validation Failure Message」プロパティのテキストの入ったページがただちに再表示されます。



18. 有効な値 (たとえば 55 かその他の有効な整数) を入力して、ページを送付します。

    これで、このマニュアルの前半のときのように Page1 が再表示されるのではなく、ValidatingCommand1 内のロジックによって SecurePage1 が表示されます。

第6章

ConfigurableBean (不可視コンポーネント)

ConfigurableBean は、コンポーネントライブラリのマニフェスト内で ConfigurableBean として明示的に指定されている JavaBean の型です。

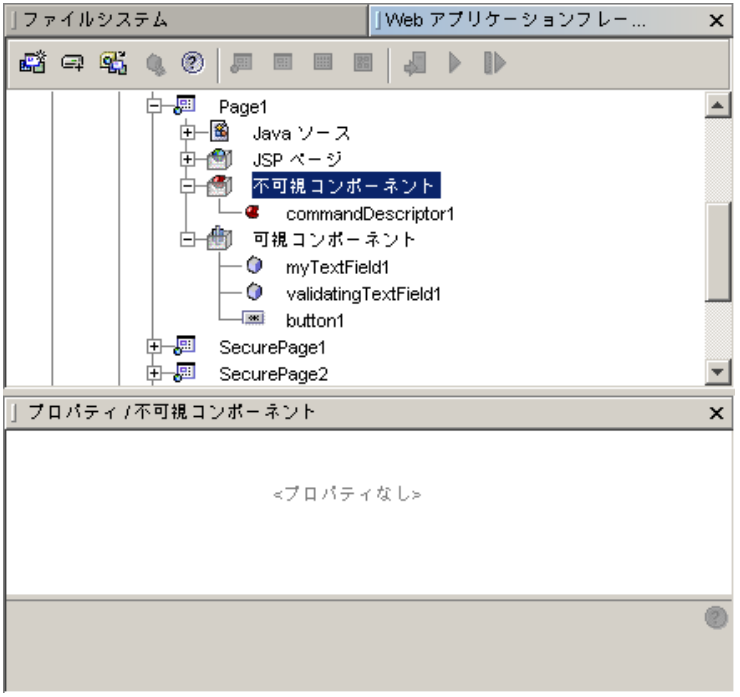
IDE は、自動的にすべてのコンポーネントライブラリマニフェストを点検し、メモリー上に ConfigurableBean 型の動的リストを構築します。検出された ConfigurableBean の型は IDE に登録されていることになります。

以下は、Web アプリケーションフレームワークコンポーネントライブラリのマニフェスト内で、ConfigurableBean をいくつか宣言している例です。

見ると分かるように、この指定は単純です。

```
<configurable-bean>
       <bean-class>com.iplanet.jato.model.SimpleModelReference</bean-class>
</configurable-bean>
<configurable-bean>
       <bean-class>com.iplanet.jato.command.CommandDescriptor</bean-class>
</configurable-bean>
<configurable-bean>
       <bean-class>com.iplanet.jato.view.command.WebActionCommandDescriptor</bean-class>
</configurable-bean>
```

ConfigurableBean がこうしたコンポーネントの専門的な名前であることに注意してください。これは、コンポーネントライブラリマニフェスト内のそれらエンティティの宣言で使用されている名前です。ただし、IDE 上では、アプリケーション開発者にとって使い馴れた用語、不可視コンポーネントとして表示されます。



ConfigurableBean と不可視コンポーネントが本質的には同じものであることを理解する必要があるのは、コンポーネント作成者だけです。

専門的には、IDE は ConfigurableBean をノードとして表示しますが、その特殊なケースとして、不可視コンポーネントをコンテナビューのサブノードとして表示するにすぎません。

正式には、あらゆる不可視コンポーネントは ConfigurableBean ですが、必ずしもすべての ConfigurableBean が不可視コンポーネントではありません。実際には、IDE 内で使用されている ConfigurableBean が明示的なノードとして現れないその他のケースもあります。「ConfigPropertyDescriptor API - Value Policy」を参照してください。

IDE での ConfigurableBean の利用のされ方と役割

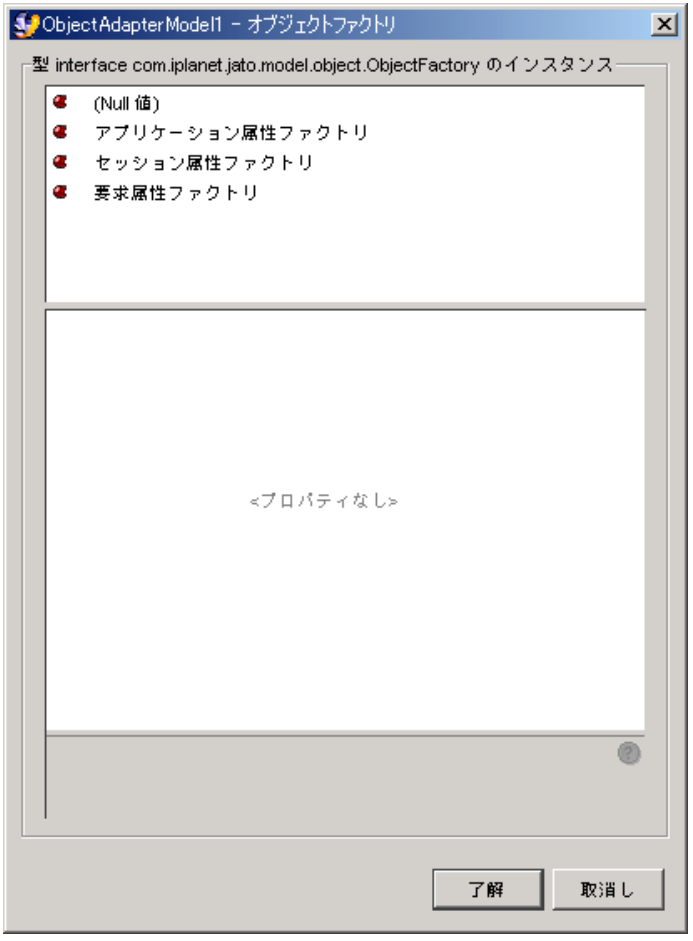
ConfigurableBean は、Web アプリケーションフレームワーク IDE では、綿密に定義されているものの、微妙な役割を果たす通常の JavaBean の型にすぎません。Web アプリケーションフレームワークコンポーネントモデルは、ConfigurableBean の力を借りて、標準の Web アプリケーションフレームワークコンポーネント (モデル、ビュー、コマンド) を補完します。

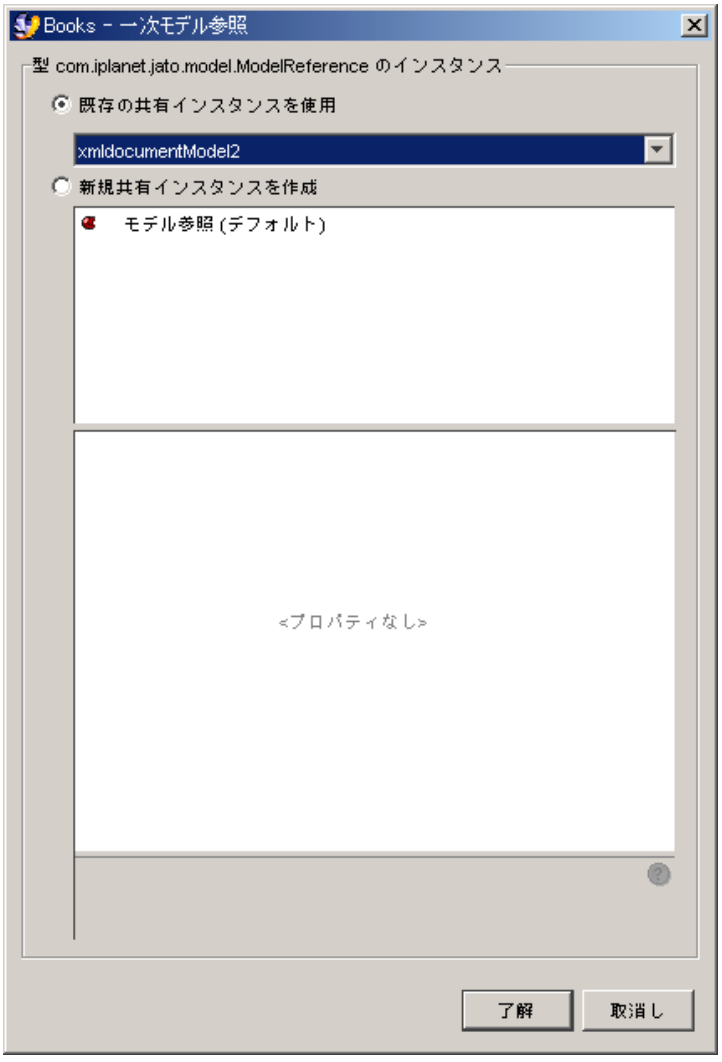
具体的には、ConfigurableBean は、Web アプリケーションフレームワークの ConfigPropertyDescriptor から始まる枠組みを完成します。コンポーネント作成者は、デザイン時の構成に表示する必要がある構成プロパティを指定する必要があると

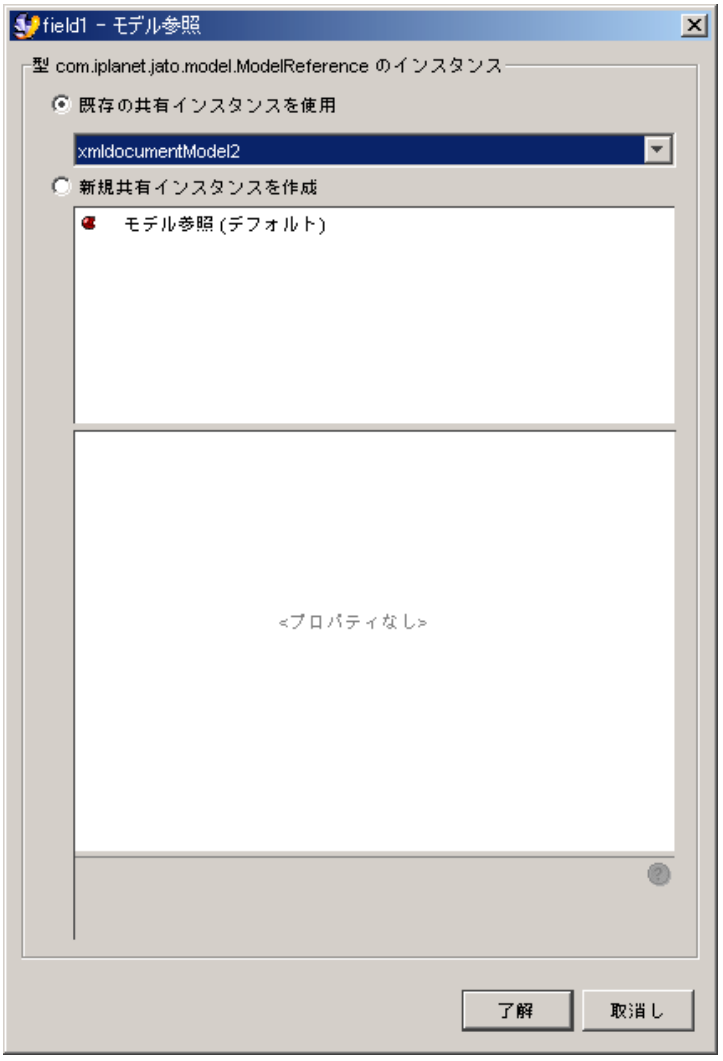
必ず ComponentInfo に ConfigPropertyDescriptor を追加します。各 ConfigPropertyDescriptor はプロパティの「型」の指定です。アプリケーション開発者は、IDE でそれらのプロパティを編集および構成する必要があります。プロパティが型指定されるため、IDE はその情報を利用して、型に固有のエディタを提供することが出来ます。たとえば構成プロパティの型が Boolean.TYPE の場合、IDE は標準の Boolean エディタを起動します。この動作は、JavaBean を意識したあらゆる IDE の特徴です。

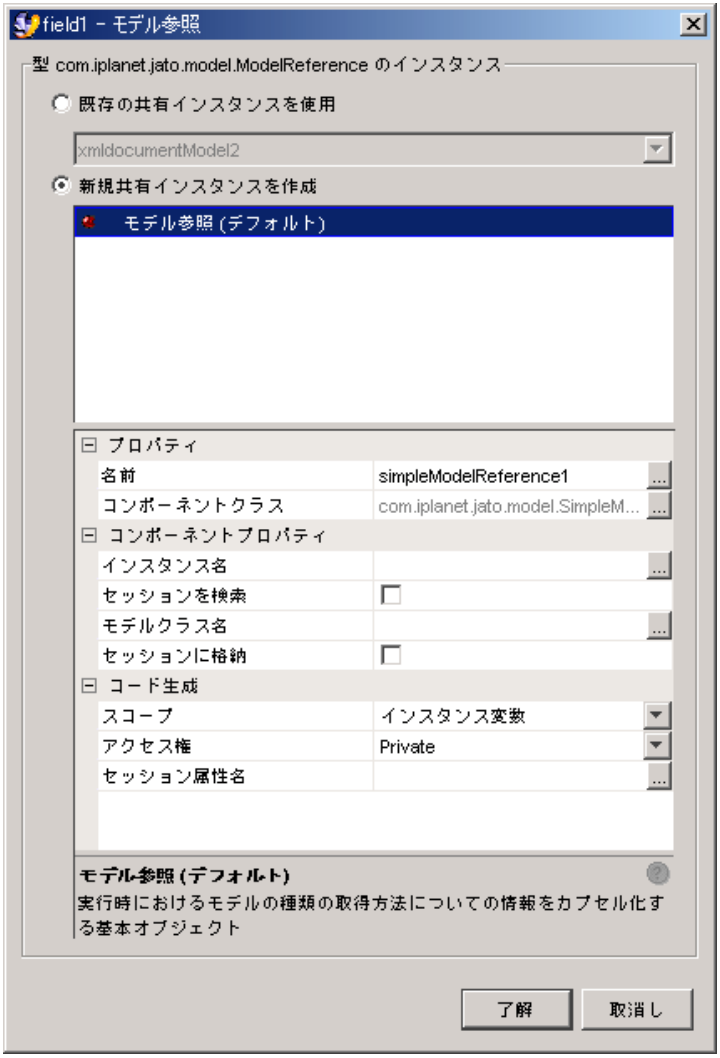
ただし、Web アプリケーションフレームワーク IDE は、標準の JavaBean エディタの持つ機能を超える機能も提供します。この追加機能は、ConfigurableBean 指定の型に対応するプロパティ型の、Web アプリケーションフレームワーク構成プロパティに IDE が提供する特殊な処理に関係しています。

IDE は、ルックアップアルゴリズムを使用して、構成プロパティの型が登録されている ConfigurableBean の型に対応しているかどうかを判定し、対応している場合は、2 つある ConfigurableBean エディタのうちの 1 つを自動的に起動します。以下は、この特別な Web アプリケーションフレームワーク ConfigurableBean エディタを示しています。


ObjectAdapterModel1 - オブジェクトファクトリ
型 interface com.iplanet.jato.model.object.ObjectFactory のインスタンス
(Null 値)
アプリケーション属性ファクトリ
セッション属性ファクトリ
要求属性ファクトリ
<プロパティなし>
了解
取消し


Books - 一次モデル参照
型 com.iplanet.jato.model.ModelReference のインスタンス
既存の共有インスタンスを使用
xmldocumentModel2
新規共有インスタンスを作成
モデル参照 (デフォルト)
<プロパティなし>
了解
取消し


field1 - モデル参照
型 com.iplanet.jato.model.ModelReference のインスタンス
既存の共有インスタンスを使用
xmldocumentModel2
新規共有インスタンスを作成
モデル参照 (デフォルト)
<プロパティなし>
了解
取消し



IDE は、値のポリシーという、ConfigPropertyDescriptor 内のさらに微妙な情報に基づいて上図 2 つのエディタの一方を起動します。値のポリシーの詳細は、ここでは説明しません。詳細は、「ConfigPropertyDescriptor API - Value Policy」を参照してください。

この節では、レイアウトが根本的に異なっている間は、ConfigurableBean エディタのどちらも共通の中核機能を提供することを見ておくことで十分です。これらのエディタはともに、構成プロパティの型からアクセス可能な ConfigurableBean の型の動的一覧をアプリケーション開発者に提供します。これは、ConfigurableBean コン

ポーネントに対する重要な付加価値機能です。それ以外の方法では通常大きな制約を受けるプロパティに、IDE が新しい選択肢をシームレスにまた動的に組み込むことを可能にする仕組みです。

たとえば IDE は、CommandDescriptor サブタイプの動的一覧を表示する ConfigurableBean エディタを使って、CommandDescriptor 型の構成プロパティを編集することを可能にします。アプリケーション開発者はまず、一覧から CommandDescriptor の型を選択し、その型のインスタンスを構成します。当然、選択されたサブタイプのプロパティは、従来の JavaBean ロジックで動的に表示されます。

標準の JavaBean 処理に任せた場合に起動される非常にシンプルな CommandDescriptor エディタに制限されることなく、Web アプリケーションフレームワーク IDE は、独自のプロパティセット、潜在的には、カスタムプロパティエディタを持つカスタム ConfigurableBean 型をコンポーネント作成者が導入する無限の機会を提供します。IDE は、そのようにして導入された型、プロパティシート、およびエディタの組み合わせを、単に定義されたプロパティに対する提供情報として、自身の中で透過的に利用します。実際、ConfigurableBean エディタは、強力なあまり前例のない特別なレベルの間接性を導入します。前例がないため、コンポーネント作成者が、エディタの提供する機会を実際に完全に認識するまでに多少の時間がかかるかもしれません。

Web アプリケーションフレームワークと ConfigurableBean の型の関係

ConfigurableBean は、Web アプリケーションフレームワークコンポーネントモデルの付加価値機能です。

コンポーネント作成者は、必ずしも ConfigurableBean 機能を使用する必要はありません。Web アプリケーションフレームワーク実行時環境やフレームワーク API における ConfigurableBean の正式な概念というものはありません。むしろ、ConfigurableBean は、コンポーネント作成者に力を付与し、開発者には IDE 経験を豊かなものにするために、コンポーネントモデルが提供する機能です。コンポーネント作成者は、既存のコンポーネントの補強、あるいは完全に新しいコンポーネントの機能強化の目的に 新しい ConfigurableBean 型を提供することが奨励されます。

Web アプリケーションフレームワークコンポーネントライブラリには、比較的多くの ConfigurableBean 型が定義されています。コンポーネント作成者は、最高の実例を提供する上でも、それらの ConfigurableBean 型の使用法に慣れてください。ConfigurableBean 型の満たす構成プロパティが、Web アプリケーションフレームワークの標準コンポーネントで宣言されている方法を理解してください。新しいコンポーネントを作成するばかりではなく、すでに定義されている以下の Web アプリケーションフレームワークの構成プロパティに適切な追加の ConfigurableBean 型を提供することによって、既存の Web アプリケーションフレームワークコンポーネントをすぐに補強できることを理解するはずです。

以下の表を、構成プロパティの検討に役立ててください。

ComponentInfo	ConfigPropertyDescriptor	プロパティ型から割り当て可能な ConfigurableBean
BasicDisplayFieldComponentInfo	ConfigPropertyDescriptor ("modelReference",com.iplanet.jato.model.ModelReference.class)	com.iplanet.jato.model.SimpleModelReference
BasicCommandFieldComponentInfo	ConfigPropertyDescriptor ("commandDescriptor", com.iplanet.jato.command.CommandDescriptor.class)	com.iplanet.jato.command.CommandDescriptor com.iplanet.jato.view.command.WebActionCommandDescriptor com.iplanet.jato.view.command.ExecuteModelCommandDescriptor com.iplanet.jato.view.command.GotoViewBeanCommandDescriptor
ObjectAdapterModelComponentInfo	ConfigPropertyDescriptor ("objectFactory", com.iplanet.jato.model.object.ObjectFactory.class)	com.iplanet.jato.model.object.factory.SessionAttributeFactory com.iplanet.jato.model.object.factory.ApplicationAttributeFactory com.iplanet.jato.model.object.factory.RequestAttributeFactory
BasicChoiceDisplayFieldComponentInfo	IndexedConfigPropertyDescriptior ("choices",com.iplanet.view.Choice.class)	com.iplanet.jato.view.SimpleChoice

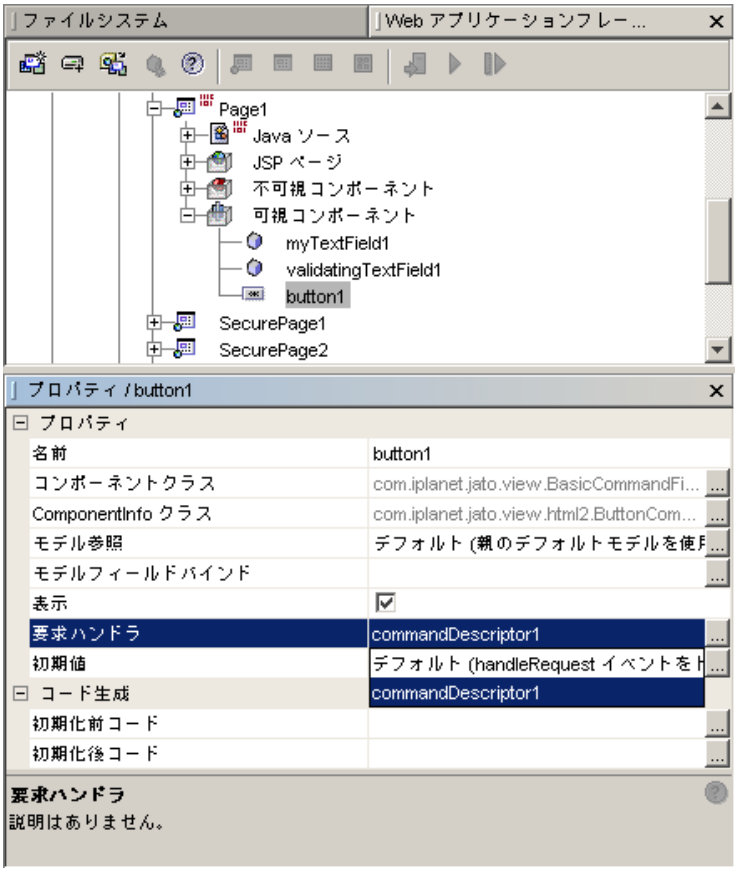
ConfigurableBean 例：CommandDescriptor

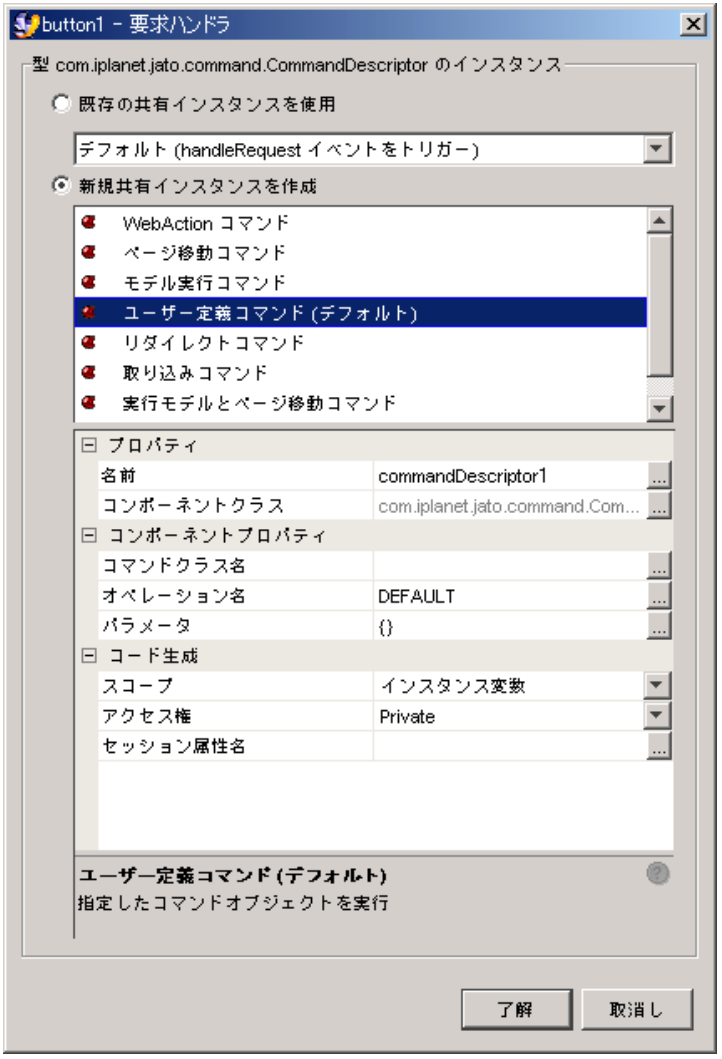
拡張可能コマンドコンポーネントは、明白にそれと分かるコマンドコンポーネントです。拡張可能モデルコンポーネントは、アプリケーション開発者による個別化を意図したカスタム実装の `com.iplanet.jato.command.Command` インタフェースです。通常、アプリケーション開発者による個別化とは、アプリケーション固有のモデルにスキーマ情報を追加することを意味します。コマンドオブジェクトの正式な構造は最低限のものであり、非常に単純なインタフェース `com.iplanet.jato.command.Command` の任意の実装です。このため、IDE 内で、プロパティの仕様を超えて新しいコマンド型を正式に作成する機会はそれほど多くありません。

Web アプリケーションフレームワークは、コマンドコンポーネントについてその他の機会を提供します。この機会を理解するには、コンポーネント作成者は CommandDescriptor の正式な役割を完全に理解する必要があります。実際には、CommandDescriptor は、コマンドオブジェクトインスタンスのデザイン時構成を可能にする構成可能 Bean です。

Web アプリケーションフレームワークおよび IDE は CommandDescriptor を利用して、アプリケーション開発者がコマンドオブジェクトの使用法を設定することを可能にします。すなわち、CommandField とコマンドオブジェクトの正式な使用関係があり、この使用関係は CommandDescriptor によって調停されます。CommandField は、起動されたときにコマンドオブジェクトを呼び出すビュー (ボタンや HREF など) です。アプリケーション開発者は、フィールドの「コマンド記述子」プロパティを介して CommandField が起動されたときに呼び出すコマンドオブジェクトを指定します。

CommandDescriptor は、実行時にコマンドクラスの特定のインスタンスを作成し、そのインスタンスを起動するために必要な情報をエンコーディングした Web アプリケーションフレームワークオブジェクトです。CommandDescriptor では最低限、実行時にインスタンス化するコマンドクラスを指定します。CommandDescriptor はまた、開発者がコマンド固有のパラメータ値を指定することを可能にします。たとえば、1 つのコマンドオブジェクトを複数の CommandField に関連付けることは非常によくあることです。各 CommandField は、明確に構成された 1 つの CommandDescriptor を使用して、そのコマンドの実行を指示、制御します。これに関する内容の詳細は、『Web アプリケーションフレームワーク 開発ガイド』をご覧ください。


ファイルシステム
Web アプリケーションフレー...
Page1
Java ソース
JSP ページ
不可視コンポーネント
可視コンポーネント
myTextField1
validatingTextField1
button1
SecurePage1
SecurePage2
プロパティ / button1
プロパティ
名前
button1
コンポーネントクラス
com.iplanet.jato.view.BasicCommandFi...
ComponentInfo クラス
com.iplanet.jato.view.html2.ButtonCom...
モデル参照
デフォルト (親のデフォルトモデルを使用
モデルフィールドバインド
表示
要求ハンドラ
commandDescriptor1
初期値
デフォルト (handleRequest イベントをト
コード生成
commandDescriptor1
初期化前コード
初期化後コード
要求ハンドラ
説明はありません。



CommandDescriptor の役割を前提にすると、コンポーネント作成者には、拡張不可コマンドコンポーネントとコンポーネントに固有の CommandDescriptor クラスを組み合わせることによって非常に豊かなコマンドコンポーネントの構想を実現する機会が存在することになります。

たとえばコンポーネント作成者は、拡張不可コマンドコンポーネントに加えて、開発者がその拡張不可コマンドコンポーネントの起動を視覚的に設定することを可能にするカスタム CommandDesciptor クラスを作成、配布することができます。カスタム CommandDescriptor そのものは、ConfigurableBean コンポーネントとして配布できます。視覚的には、ConfigurableBean は、IDE によって不可視コンポーネントとして表示されます。

第7章

拡張不可モデルとコマンド、コンテナビューコンポーネントの開発と配布

前述したように拡張可能コンポーネントと拡張不可コンポーネントの間には 1 つの基本的な違いがあります (21 ページの「拡張可能コンポーネントと拡張不可コンポーネント」を参照)。

これまでの練習問題では、5 つのコンポーネントの開発と配布、テストサイクルの例を紹介しました。それらコンポーネントの 2 つ以外はすべて拡張可能コンポーネントです。

- MyTextField (拡張不可表示フィールドコンポーネント)
- ValidatingDisplayField (拡張不可表示フィールドコンポーネント)
- SecureViewBean (拡張可能 ViewBean コンポーネント)
- XMLDocumentModel (拡張可能モデルコンポーネント)
- ValidatingCommand (拡張可能コマンドコンポーネント)

拡張不可モデルやコンテナビュー、コマンドコンポーネントの開発、配布について

拡張不可モデル、コンテナビュー、コマンドコンポーネントは、簡単に開発、配布できます。それらは、それぞれ拡張可能なモデル、コンテナビュー、コマンドコンポーネントから IDE 内で作成された具体的なモデル、コンテナビュー、コマンドです。他のすべての配布コンポーネント同様、複数のアプリケーションに組み込まれることを明白に意図して、コンポーネントライブラリ JAR ファイルで配布されることを除けば、アプリケーション固有のモデルやコンテナビュー、コマンドと何ら違いはありません。配布方法は、拡張不可モデル、コンテナビュー、コマンドコンポーネントのどれも共通です。

拡張不可モデルやコンテナビュー、コマンドコンポーネントを開発、配布する理由

拡張不可モデルやコンテナビュー、コマンドコンポーネントは、コンポーネント作成者がよく利用されるコンポーネントをそのコンポーネント使用者に供給するいくつかの機会を提供します。この後の説明は、この話題の表面をなぞっているに過ぎません。

拡張不可モデルやコンテナビュー、コマンドコンポーネントが提供する機会で最もはっきりしているのは、コンポーネント作成者が、大規模な再利用可能アプリケーションや組織規模のコンポーネントに小さな基本構成要素を提供する以上のことを行えるようになることです。一般に拡張不可モデルやコンテナビュー、コマンドコンポーネントは、コンポーネント食物連鎖の頂上を形成します。コンポーネント作成者が、複雑で非常にきめの粗い任意のコンポーネントを提供することを可能にします。表示フィールドコンポーネントを最もきめの細かいコンポーネントとみなすなら、拡張不可モデルやコンテナビュー、コマンドコンポーネントは、コンポーネントスペクトルでその退去にするコンポーネントです。企業などの組織は、拡張不可モデルやコンテナビュー、コマンドコンポーネントからなる水平または垂直型の非常に洗練されたライブラリを作成することができ、そのライブラリの非常に大規模で非常に再利用可能、非常に強力な基本構成要素から、アプリケーション開発者はアプリケーションを組むことができます。

たとえば拡張不可モデルコンポーネントは、事前にパッケージ化されたすぐに使える形で特定の組織データを利用できるようにすることを可能にします。アプリケーション開発者は、新しいビューを単に定義し、それらのアプリケーションに固有のビューを事前にパッケージ化されたモデルに視覚的にバインドすればよいだけです。

拡張不可コンテナビューは、小さなビューの任意の複雑な集合からなる、事前に構成された可視基本構成要素を提供することができます。

さらに、拡張不可コマンドコンポーネントは、プラグ & プレイの動作を提供することができます。

拡張不可コンポーネントは、ライブラリ内で他の拡張不可コンポーネントを使用するよう事前に構成することができます。たとえば ShoppingCart コンテナビューコンポーネントを、関係する拡張不可モデルコンポーネントを使用するように事前に構成することができます。

事前に構成されたこうしたコンテナビューおよび モデルコンポーネントは、一体として、事前に統合され、すぐに使える形でビジュアルプレゼンテーションおよびデータアクセス手段を提供します。このことに加えて、コンポーネント作成者は、1 つ以上の拡張不可コマンドコンポーネントを使用しするように事前にそのコンテナビューコンポーネントを構成し、そうすることで、複合体に事前に統合されたコマンド動作を追加することができます。

組織は、統合された拡張不可モデルやコンテナビュー、コマンドコンポーネントをまとめたものからなるツールボックスを作成することができます。それらツールボックスは、アプリケーションの短期開発のために組織内で使用したり、広範囲の BtoB (business to business) アーキテクチャの一環としてパートナーに提供したりすることができます。

機会に限りはありません。

拡張不可モデル、コンテナビュー、コマンドコンポーネントの開発

1. **通常の Web アプリケーションフレームワークアプリケーションオブジェクトと同様にして、Web アプリケーションフレームワーク IDE でコンポーネントを開発します。**
   - 適切なモデルかコンテナビュー、コマンドウィザードを使用して、コンポーネントを作成します。
   - コンポーネントのプロパティを構成します。
   - コンポーネントの Java クラスに任意の動作を追加します。
   - オプションで、コンテナビューにビューコンポーネントを追加します (コンテナビューコンポーネントのみ)。
   - オプションで、JSP ページレットを関連付けて、コンテナビューコンポーネントの生成仕様を提供します (コンテナビューコンポーネントのみ)。
   - オプションで、モデルにモデルフィールドかモデルオペレーションを追加します (モデルコンポーネントのみ)。
2. **この新しいコンポーネント用の ComponentInfo クラスを作成します。**

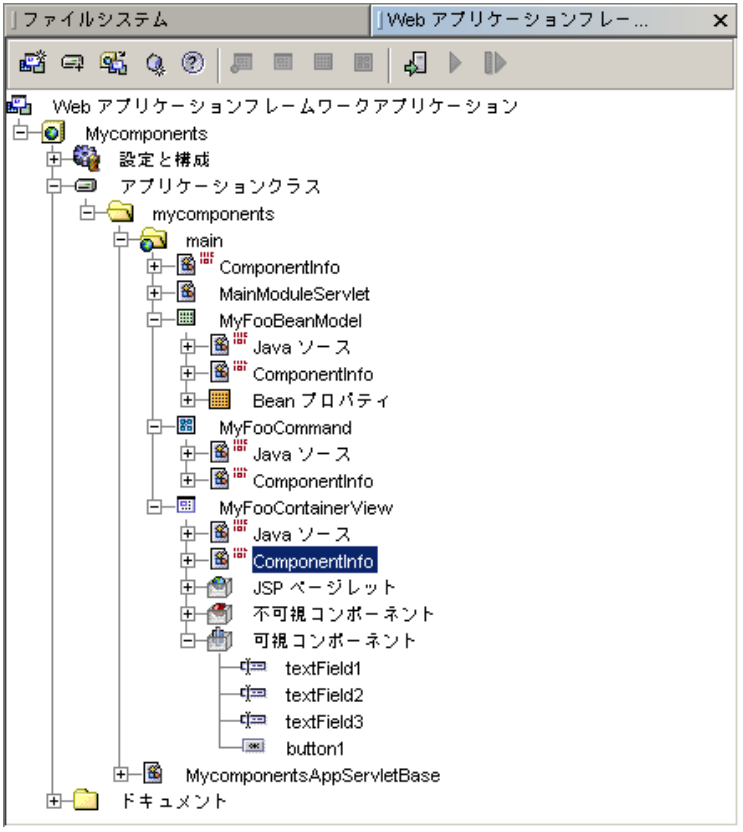
   厳密には必須ではありませんが、作成することを強く推奨します。

   コンポーネント固有の ComponentInfo を作成して、最低限、コンポーネント固有の ComponentDescriptor を提供してください。オプションで、ComponentInfo を使って、拡張不可コンポーネントに合った高度なコンポーネントモデル機能のすべてを指定することもできます。

注 – IDE 内で視覚的に拡張不可モデルやコンテナビュー、コマンドコンポーネントを作成するには、Web アプリケーションフレームワークアプリケーションのコンテキスト内でそれらコンポーネントを作成する必要があります。すなわち、IDE を使って拡張不可モデルやコンテナビュー、コマンドコンポーネントを作成するには、最初に Web アプリケーションフレームワークアプリケーションを作成する必要があります。これは、現在 IDE が「ライブラリのみのデザイン」モードをサポートしていないためです。将来のバージョンの Web アプリケーションフレームワークの IDE では、開発者が「新規ライブラリ」と「新規アプリケーション」のいずれかを選択できるようになるかもしれません。現在でも、Web アプリケーションフレームワークアプリケーション内で Web アプリケーションフレームワークの新規オブジェクトウィザードだけ利用すればよいだけです。コンポーネント作成者が「アプリケーション」内で新しいコンポーネントを作成したとしても、その事実は、コンポーネントのその後の独立とは何の関係もありません。アプリケーションは、コンポーネント作成者が作成中に完全な可視 IDE 機能セットを利用することを可能にする、IDE が認識するコンテキストを提供するだけです。コンポーネント作成の最終的な成果はコンポーネントの Java リソースであり、これは、視覚的なデザインが行われた基のアプリケーションから完全に独立しています。

新しい拡張不可モデルやコンテナビュー、コマンドコンポーネントのデザインの目的にだけ新しい Web アプリケーションフレームワークを作成する場合、アプリケーションそのものの名前は重要ではありません。推奨する方法の 1 つは、アプリケーションを、その新しい拡張不可モデルやコンテナビュー、コマンドコンポーネントの「テスト用アプリケーション」と考えることです。テスト用アプリケーションでのコンポーネントのパフォーマンスについて満足な結果が得られたら、コンポーネントライブラリ JAR ファイルにその新しいコンポーネントを追加してください。

IDE は、Web アプリケーションフレームワークコンポーネントの ComponentInfo Java ソースをコンポーネントの「一部」とみなします。このことは、コンポーネントの Java ソースがその一次ノードの子として現れるのとちょうど同じように、ComponentInfo Java ソースノードがコンポーネントの一次ノードとして現れることを意味します。


ファイルシステム
Web アプリケーションフレー...
Web アプリケーションフレームワークアプリケーション
Mycomponents
設定と構成
アプリケーションクラス
mycomponents
main
ComponentInfo
MainModuleServlet
MyFooBeanModel
Java ソース
ComponentInfo
Bean プロパティ
MyFooCommand
Java ソース
ComponentInfo
MyFooContainerView
Java ソース
ComponentInfo
JSP ページレット
不可視コンポーネント
可視コンポーネント
textField1
textField2
textField3
button1
MycomponentsAppServletBase
ドキュメント

拡張不可モデル、コンテナビュー、コマンドコンポーネントの配布

1. **これまでに説明していない追加のサブ要素 1 つとともに、コンポーネントライブラリの `complib.xml` にコンポーネント要素を追加します。**
   - コンポーネント要素には、`design-reference-resource` サブ要素を含める必要があります。
   - `design-reference-resource` サブ要素には、コンポーネントのオブジェクト定義リソースの場所を指定する必要があります。
2. **Web アプリケーションフレームワークコンポーネントライブラリ JAR に拡張不可コンポーネントを追加します。**
   - Web アプリケーションフレームワークコンポーネントのときと同様、すべてのクラスと他のコンポーネントに固有のリソースを含めます。
   - コンポーネントのオブジェクト定義ファイルを含めます。これは重要な違いです。このマニュアルでこれまでに説明した他のコンポーネントでは、このファイルは必要ありません。
   - 特殊な「追加ファイル」リソースを含めます。185 ページの「「追加ファイル」の自動アンパック」を参照してください。

     共通追加ファイルリソースとしては、コンテナビューコンポーネントの関連付けされた JSP ページレットファイルなどが挙げられます。

   以下は、`complib.xml` 内の拡張不可コンポーネントエントリの例です。

```
<component>
          <component-class>mycomponents.MyFooCommand</component-class>
          <component-info-
class>com.iplanet.jato.command.BasicCommandComponentInfo</component-info-class>
          <design-reference-resource>/mycomponents/MyFooCommand.command</design-reference-
resource>
</component>
```

IDE での拡張不可モデルやコンテナビューコマンドコンポーネントの作成を取り上げたとのは意図的であり、大切であるためです。このマニュアルの以前のコンポーネント例はどれも、IDE 内部でのコンポーネントの種類そのものの開発を想定していないか、その必要もありません。IDE 内でコンポーネントを視覚的に使用するには、Web アプリケーションフレームワーク対応の IDE を使用する必要がありましたが、コマンドコンポーネントや ComponentInfo、`complib.xml` の作成、コンポーネントライブラリ JAR ファイルの作成に、IDE を使用する必要はありませんでした。

しかしながら、拡張不可モデルやコンテナビュー、コマンドコンポーネントでは、このことは当てはまりません。オブジェクト定義ファイルと呼ばれる拡張不可コンポーネントのメタデータを生成できるのは IDE だけであるため、それらコンポーネントは IDE 内で開発する必要があります。要約すれば、拡張不可モデルやコンテナビュー、コマンドコンポーネントの配布の鍵は、そのクラスおよび ComponentInfo とともにオブジェクト定義ファイルを配布することにあります。

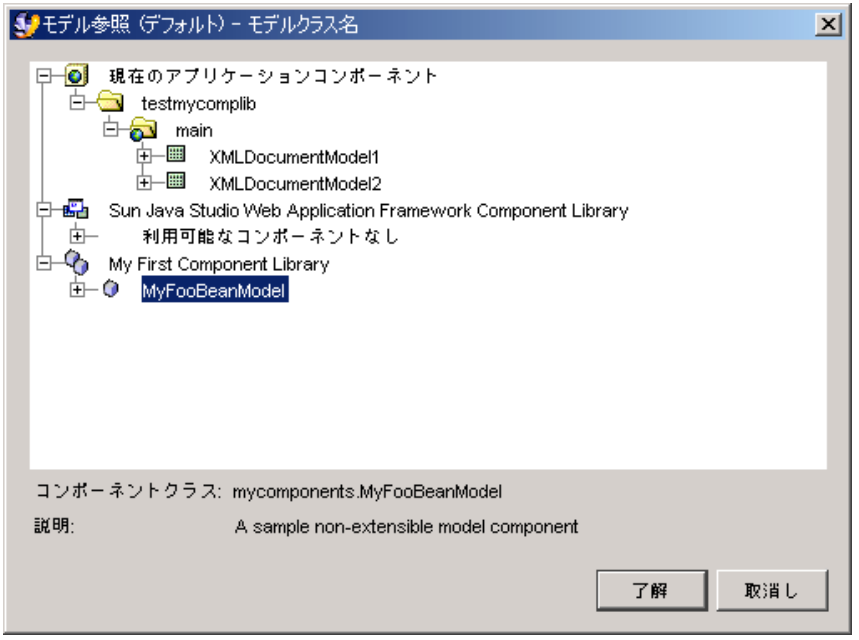
同じライブラリ内に、拡張不可と拡張可能両方のコンポーネント (モデル、コンテナビュー、コマンド) を混在させることはできるのでしょうか。

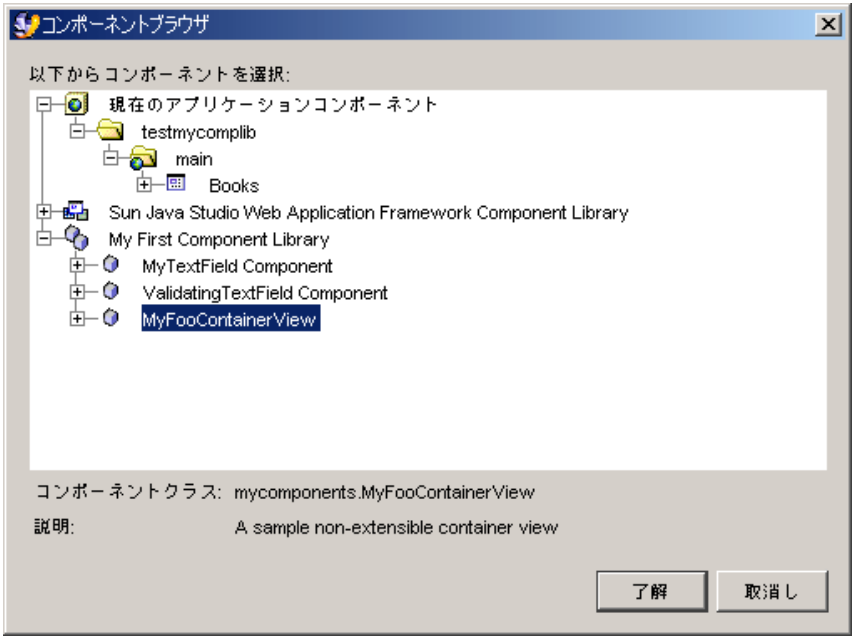
はい、可能です。むしろそれを想定してデザインされており奨励しています。

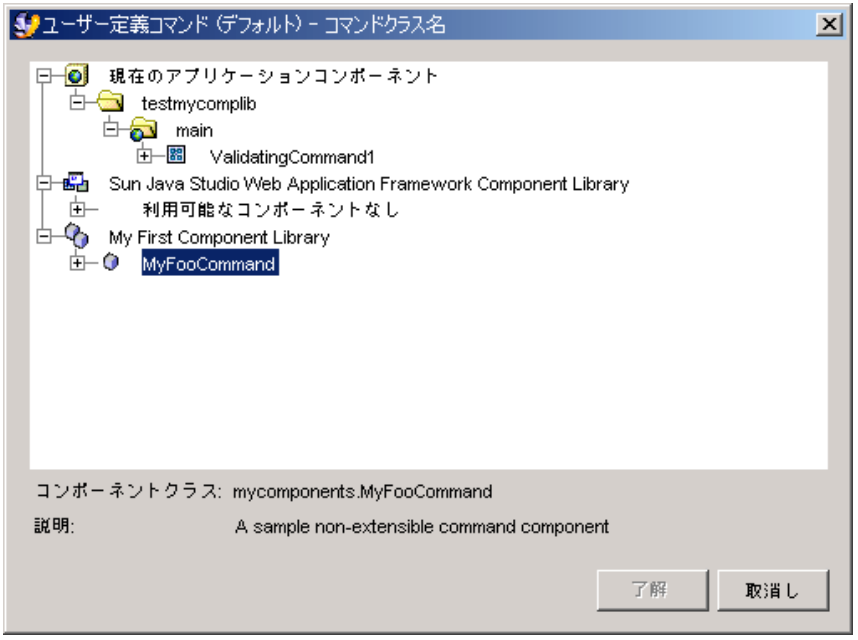
Web アプリケーションフレームワークコンポーネントライブラリで拡張不可モデルやコマンド、コンテナビューコンポーネントを配布した後、アプリケーションはそれらコンポーネントをどのように利用するのでしょうか。

Web アプリケーションフレームワークコンポーネントライブラリ内の拡張不可モデルやコマンド、コンテナビューコンポーネントは、IDE 内では、現在のアプリケーションの拡張不可コンポーネントが現れるのと完全に同じ IDE コンテキスト内に現れます。同語重複に聞こえますが、意図的にそうしています。このことは、Web アプリケーションフレームワークの IDE からは、モデル、ビュー、コマンドオブジェクトはどれはコンポーネントであることを指摘しています。IDE に関する限り、コンポーネントがコンポーネントライブラリの JAR または現在のアプリケーション空間内のどちらで検出されたかは重要ではありません。

下図は、このことを表す例です。IDE 内のさまざまなコンポーネント選択で、アプリケーション開発者はライブラリ提供の拡張不可コンポーネントとアプリケーション定義の拡張不可コンポーネントを自由に、また透過的に選択することができます。


モデル参照 (デフォルト) - モデルクラス名
現在のアプリケーションコンポーネント
testmycomplib
main
XMLDocumentModel1
XMLDocumentModel2
Sun Java Studio Web Application Framework Component Library
利用可能なコンポーネントなし
My First Component Library
MyFooBeanModel
コンポーネントクラス: mycomponents.MyFooBeanModel
説明: A sample non-extensible model component
了解
取消し


コンポーネントブラウザ
以下からコンポーネントを選択:
現在のアプリケーションコンポーネント
testmycomplib
main
Books
Sun Java Studio Web Application Framework Component Library
My First Component Library
MyTextField Component
ValidatingTextField Component
MyFooContainerView
コンポーネントクラス: mycomponents.MyFooContainerView
説明: A sample non-extensible container view
了解
取消し



オブジェクト定義ファイル (拡張不可コンポーネントのメタデータ)

コンポーネント作成者およびアプリケーション開発者のどちらも、オブジェクト定義ファイルについては、ファイルが IDE によって作成され、最初のクラスのアプリケーションリソースとして扱う必要があることを理解しておけば十分でしょう。

- オブジェクト定義ファイルは、従来のアプリケーションリソースとともにソースコード管理システムに保存します。
- オブジェクト定義ファイルを手動で編集してはいけません。
- オブジェクト定義ファイルに実行時の値はなく、サーブレットコンテナに配備する必要はありません。
- オブジェクト定義ファイルは、拡張不可モデルやコンテナビュー、コマンドコンポーネントとともに配布する必要があります。

オブジェクト定義ファイルに関する次の詳細情報は単なる参考です。コンポーネント作成者とアプリケーション開発者のどちらの知っている必要のない、Web アプリケーションフレームワークコンポーネントモデルの実装の詳細です。

Web アプリケーションフレームワークの IDE は、そのオブジェクト定義ファイル内に XML 形式でデザイン時の状態を格納します。オブジェクト定義ファイルという用語は、そうしたファイルに対する総称です。オブジェクト定義ファイルには、以下の特徴があります。

- Web アプリケーションフレームワークのモデル、コンテナビュー、コマンド DTD に準拠した XML ファイルです。
- Web アプリケーションフレームワークのデザイン時状態の信頼できる表現です。
  - IDE は、アプリケーション開発者のデザイン時の決定 (オブジェクトの階層宣言プロパティの構成など) を収集する目的で作成します。
  - IDE が、IDE セッションにまたがってデザイン時状態を復元する目的で読み取ります。
  - IDE は、オブジェクト定義ファイル内に格納されているデザイン時状態に相当する Java コードをコンポーネントの Java ソースファイル内に生成します。このコードは、IDE の Java ソースコードエディタに対して保護されたブロック (編集不可ブロックなど) として生成されます。
- Web アプリケーションフレームワークの IDE が認識するファイル接頭辞の `.model`、`.viewbean`、`.cview`、`.command` のどれかを持ちます。
- 実行時の値はありません。

  IDE は、オブジェクト定義ファイル内に格納されているデザイン時状態に相当する Java コードをコンポーネントのアプリケーションクラス内に生成します。このため、オブジェクト定義ファイルは、まったく実行時の役割はなく、存在することもありません。

第8章

デザインアクション

この章は、7 ページの「初めてのコンポーネントの開発」および 77 ページの「モデルコンポーネントの開発」をすでに読み終えていることが前提になります。

コンポーネントデザインアクションを持つ拡張可能コンポーネントの開発

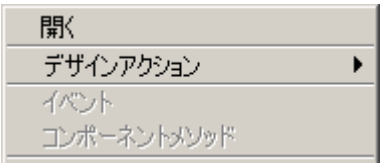
この章では、拡張可能コンポーネントにデザインアクションを追加するのに必要な基本的な手順を説明します。

コンポーネントデザインアクションは、コンポーネント用の任意のデザイン時動作をカプセル化します。デザインアクションでは、「About」や「Credits」ダイアログの送信や妥当性検査の実行、コンポーネントの自動構成、サポートファイルの同期、メッセージや警告の表示、複雑なエディタやウィザードの起動、さらには、コンポーネントオブジェクトモデルの細部の編集などを行うことができます。

コンポーネントデザインアクションをサポートするには、Web アプリケーションフレームワークコンポーネントアーキテクチャに対する、DesignableComponentInfo という新しいインタフェースを導入します。DesignableComponentInfo は、コンポーネント作成者が拡張可能コンポーネントに対して特殊なデザイン時動作を定義することを可能にする、オプションの特殊な ExtensibleComponentInfo です。DesignableComponentInfo の実用的な用途は、IDE で開発者にデザインアクションを提供することです。拡張可能コマンドやモデル、ビューの ComponentInfo クラスはどれも、オプションで DesignableComponentInfo を実装して、ComponentDesignActionDescriptor を表示することができます。

IDE モジュールは拡張可能コンポーネントのデザインアクションのみ提供するため、DesignableComponentInfo はこの規則を課すために ExtensibleComponentInfo を個別化したものです。

ComponentDesignActionDescriptor がある場合、IDE はそれを使用して、コンポーネントのノードのコンテキストメニュー「デザインアクション」に定義済みデザインアクションのサブメニューリストを表示します。



DesignableComponentInfo は、Web アプリケーションフレームワークコンポーネントアーキテクチャに最近追加されたものであるため、現在のところ、Bean アダプタモデルだけが DesignableComponentInfo を使用して、「フィールドを更新」デザインアクションを表示します。

コンポーネントデザインアクションとは

コンポーネントデザインアクションは事実上どのようなアクションの実行にも使用することができますが、いくつか考慮すべき指針があります。

デザイン時に拡張可能コンポーネントを構成する際に開発者が期待する標準の機能は、一次コンポーネントノードのプロパティシートかサブノードの内包プロパティシート、あるいはその両方です。このとき、コンポーネント作成者は最初に構成プロパティの自動サポートに、さらにオプションで、複雑な構成プロパティに使用できるカスタムエディタに目を向けるべきです。構成プロパティやカスタムエディタの代用にデザインアクションを使用することは、進むべき方向として間違っています。

いくつかのコンポーネントのデザイン面の管理では、個別のコンポーネント構成プロパティエディタだけでは十分でないことがしばしばあります。構成プロパティ用のエディタには、同じコンポーネントオブジェクトモデルの他の構成プロパティや他の部分に対するスコープがありません。このため、関連する構成プロパティの一部を全体として編集することができません。モデルやコンテナビューの場合、モデルフィールドや子ビューのまとまりを一緒に編集する手段はありません。また、現在のコンポーネントアーキテクチャは、新しいコンポーネントの作成で開発者が使用するウィザードまたは初期化機構を指定できるようになっていません。

こうした問題に対する 1 つの解決策がコンポーネントデザインアクションです。この章の後ろで説明するように、コンポーネントデザインアクションの実行中、コンポーネント作成者には、非常に強力なデザイン変更を実装できる (連続的で再入可能なデザイン変更の実装も可能) コンテキストが提供されます。

コンポーネントデザインアクションの好例は、Web アプリケーションフレームワークコンポーネントライブラリの Bean アダプタモデルにあります。Bean アダプタモデルには、適用された JavaBean の型やクラスを開発者が指定することを可能にする「Bean 型」構成プロパティがありますが、開発者が Bean プロパティ用のモデルフィールドを自動的に生成する簡単な方法はありません。Bean アダプタモデルの「フィールドを更新」デザインアクションは、Bean 型構成プロパティの妥当性を検

査し、少なくと一群の Bean プロパティのすべてが代表的なモデルフィールを持つようにします。このデザインアクションは、適用された Bean 変更として使用できるため、連続的なデザインをサポートします。

コンポーネントデザインアクションには、コンポーネントオブジェクトモデルを使用しない機構によって構成することが可能な「ブラックボックス」コンポーネントをコンポーネント作成者が発行する機会が提供されます。たとえばカスタムの実行時 XML 記述子またはプロパティファイルを使用するビューまたはモデルコンポーネント、あるいは新しいコンポーネントの土台となるものがすでに存在していると仮定します。この場合、コンポーネント作成者には、このカスタム構成用のエディタを提供する手段が必要です。ComponentDesignContext の高度な API は、こうした状況に対応します。

この章の以降の部分は、`com.iplanet.jato.component.design` および `com.iplanet.jato.component.design.objmodel` パッケージの JavaDoc をすでに読み終えていることが前提になります。

DesignableComponentInfo および ComponentDesignActionDescriptor API を使用して拡張可能コンポーネントに ComponentDesignAction を指定する単純な例を紹介します。

ComponentInfo でのデザインアクションの提供

拡張可能コンポーネント用のコンポーネントデザインアクションを提供する第 1 のステップは、DesignableComponentInfo を実装することです。

下記の例では、XMLDocumentModel の例に追加をします。名前や論理名、情報ダイアログを開発者に提供し、構成プロパティなどのコンポーネントの詳細をダンプする、「About」という単純なデザインアクションを作成します。

1. ComponentInfo に `DesignableComponentInfo` を実装します。
2. `getComponentDesignActionDescriptors()` メソッドを実装します (下記のコード例を参照)。

3. ComponentDesignActionDescriptor Bean は ComponentDesignAction クラスの割り当てを必要とするため、アクションクラスを作成する必要もあります。

   最低限必要なのは、performAction(ComponentDesignContext) メソッドなどの ComponentDesignAction を実装しているクラスです。

   このための簡単な方法は、下記のコード例で見られるように ComponentInfo の内側のクラスでアクションクラスを定義する方法です。

   コンポーネントデザインアクション機構は、デザインアクションクラスのデフォルトのコンストラクタ (引数なし) を呼び出すだけです。このため、何の意味もないため、代替コンストラクタの使用は避けます。

   速やかに復帰する以外のことを行うのに、performAction() メソッドは必要ありません。下記の最小コード例では、モード付きダイアログを送信し、ヘルパーメソッドの aboutDisplayMessage() を呼び出します。

   このコードでは、例として紹介しやすいよう、String 値を直接埋め込んでいます。表示文字列を地域対応する必要があると思われる場合は、リソースバンドルを利用してください。

   下記の例は、XMLDocumentModelComponentInfo.java に追加する必要があるコードを表しています。関係のないコードは省略し、その部分は省略符号 (...) で表しています。

```
. . .
import com.iplanet.jato.component.design.*;
import com.iplanet.jato.component.design.objmodel.*;

public class XMLDocumentModelComponentInfo extends ExtensibleModelComponentInfo
        implements DesignableComponentInfo
{
        . . .

        public ComponentDesignActionDescriptor[] getComponentDesignActionDescriptors()
       {
                if(null != designActionDescriptors)
                        return designActionDescriptors;

                List descriptors=new ArrayList();

                ComponentDesignActionDescriptor descriptor = new
ComponentDesignActionDescriptor(AboutDialog.class);
                descriptor.setName("About");

                descriptor.setDisplayName("About");
                descriptor.setShortDescription(
                        "Displays a small list of component details");
                descriptors.add(descriptor);

                designActionDescriptors = (ComponentDesignActionDescriptor[])
                        descriptors.toArray(
                                new ComponentDesignActionDescriptor[descriptors.size()]);
```

```
                        return designActionDescriptors;
                }

                public static class AboutDialog implements ComponentDesignAction
{
                        public void performAction(ComponentDesignContext context)
                                throws DesignActionException
                        {
                                javax.swing.JOptionPane.showMessageDialog(
                                        context.getMainWindow(),
                                        aboutDisplayMessage(context),
                                        "XMLDocumentModel About Design Action",
                                        javax.swing.JOptionPane.INFORMATION_MESSAGE);
                        }

                        private String aboutDisplayMessage(ComponentDesignContext context)
                        {
                                StringBuffer msg = new StringBuffer(
                                        "Component Name: " +
                                        context.getComponentInfo().getComponentDescriptor(
).getName() +
                                        "\nComponent Logical Name: " +
                                        context.getComponentLogicalName() + "\n");
                                ConfigPropertyNode[] props =
((ConfigPropertyNodeContainer)
                                        context.getPrimaryObjectModel()).getConfigProperty
Node();
                                for(int i=0;i<props.length;i++)
                                {
                                        props[i].dump(msg, "\t");
                                        msg.append("\n");
                                }
                                return msg.toString();
                        }
        }

        . . .

        private ComponentDesignActionDescriptor[] designActionDescriptors;
}
```

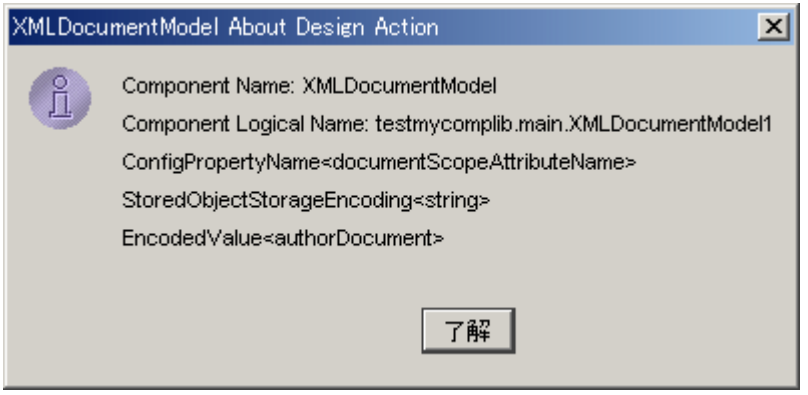
AboutDialog アクションは、モード付きの Swing 情報ダイアログボックスが開発者に表示されます。ComponentDesignContext の「MainWindow」プロパティは、Swing 可視コンポーネントの配置に使用しています。表示するメッセージは、ヘルパーメソッドの `aboutDisplayMessage()` から取られます。

ここでもまた、ComponentDesignContext のさまざまなプロパティを使って、名前や論理名などのコンポーネントに関する情報が取得され、構成プロパティを順にループして、その値をダンプします。

構成プロパティにアクセスするため、オブジェクトモデルインタフェースの ConfigPropertyNodeContainer および ConfigPropertyNode が利用されています。この AboutDialog コード例は、コマンド、ContainerView、ViewBean、モデル拡張可能コンポーネントで利用することができます。モデル専用ではありません。

アプリケーションでこのコンポーネントをコンパイルし、パッケージ化して、使用すると、XMLDocumentModel のインスタンスによって、「About」デザインアクションが提供されます。

アクションの実行結果は以下のようになります。



付録 A

コンポーネントライブラリの構造

コンポーネントライブラリの概要

Web アプリケーションフレームワークコンポーネントは、通常の JAR ファイルにパッケージ化されて配布されます。JAR ファイルには、以下が含まれる必要があります。

- コンポーネントライブラリマニフェスト (`/COMP-INF/complib.xml`)
- コンポーネントライブラリ固有の Java リソース (コンポーネントクラス、ComponentInfo クラス、リソースバンドル、コンポーネントのアイコンイメージ、その他補助ファイル)。標準の Java 規則に従って、あらゆるクラス (コンポーネント、ComponentInfo、その他補助ファイル) を JAR に入れてください。

オプションで、Web アプリケーションフレームワークのコンポーネントライブラリ JAR は、以下を含むこともできます。

- `/webapp` という名前の特殊なディレクトリ

  この `/webapp` ディレクトリの内容は「追加ファイル」と呼ばれます。これは、開発者がコンポーネントライブラリ JAR で任意の追加ファイルを配布することを可能にする、Web アプリケーションフレームワーク IDE の付加価値機能です。Web アプリケーションフレームワーク IDE は、Web アプリケーション開発環境に追加ファイルを「アンパック」します。詳細は、185 ページの「「追加ファイル」の自動アンパック」を参照してください。

コンポーネントライブラリの構造

Web アプリケーションフレームワークのコンポーネントライブラリ JAR の内容は、以下のように構成されている必要があります。

```
/COMP-INF/complib.xml
/[Java クラスとリソース]
/webapp/[IDE によるデザイン時自動抽出用の追加ファイル]
```

注：webapp ディレクトリは任意

コンポーネントマニフェスト

Web アプリケーションフレームワークでは、Web アプリケーションフレームワークの特殊なコンポーネントライブラリマニフェストファイルをコンポーネントライブラリの JAR に含める必要があります。このファイルは、ライブラリ内のコンポーネントのまとまりを記述した単純な XML ドキュメントです。コンポーネントライブラリマニフェストでは、任意の個数のコンポーネントと関連付けられた Web アプリケーションフレームワークコンポーネントモデルリソースを宣言することができます。

IDE は、Web アプリケーションフレームワークの `WEB-INF/lib` ディレクトリにマウントされているすべての JAR ファイルを自動的に内観します。具体的には、JAR の中にコンポーネントライブラリマニフェストファイルがないかどうかを調べます。JAR ファイル内の所定の場所で有効なコンポーネントライブラリマニフェストファイルを検出した場合は、正しく宣言されているすべてのコンポーネントを表示し、IDE でデザイン時利用できるようにします。JAR ファイル内の所定の場所で有効なコンポーネントライブラリマニフェストファイルが見つからないか、コンポーネントライブラリマニフェストが無効な場合は、その JAR をコンポーネントライブラリとみなしません。

コンポーネントライブラリマニフェストは、次の厳密な規則に従っている必要があります。

- コンポーネントライブラリマニフェストファイルの名前が `complib.xml` である。
- `complib.xml` ファイルが正しく書式化された XML ファイルである。
- `complib.xml` の内容が jato-component-library_1_0.dtd に従っている (下記を参照)。
- `complib.xml` ファイルが、コンポーネントライブラリ JAR の `/COMP-INF` ディレクトリ内にある。

jato-component-library_1_0.dtd

```
<!--
コンポーネントマニフェストのルート要素は component-library 要素
-->
<!ELEMENT component-library (tool-info, library-name, display-name,
    description?, legal-notice?, icon?, interface-version, implementation-
version,
    author-info?, taglib*, component*, extensible-component*,
    configurable-bean*)>

<!--
tool-info 要素には、このライブラリの対象になっているツール環境に関する情報を含む
-->
<!ELEMENT tool-info (tool-version)>

<!--
tool-version 要素には、このライブラリが対象にしている Web アプリケーションフレームワーク
/JATO のインタフェースのバージョン情報を含める。「2.1.0」などのようにドット区切りの
バージョン番号を指定する。
-->
<!ELEMENT tool-version (#PCDATA)>

<!--
library-name 要素には、コンポーネントライブラリの内部名を含める。
この名前はグローバルに一意であるとみなされ、標準の Java パッケージ命名規則に従っている必要が
ある。たとえば Web アプリケーションフレーム/JATO コンポーネントライブラリのライブラリ名は、
そのパッケージ構造のルートの「com.iplanet.jato」である。
-->
<!ELEMENT library-name (#PCDATA)>

<!--
display-name 要素には、GUI ツールで提供するライブラリの短い表示名を含める。
-->
<!ELEMENT display-name (#PCDATA)>

<!--
description 要素は、その親要素の説明テキストを含むのに使用される。
-->
<!ELEMENT description (#PCDATA)>

<!--
legal-notice 要素には、このライブラリに付属する法律または著作権テキストを含める。この要素
は、つねにこのライブラリの付近にこの情報を置く別の機会を提供することを意図しているが、ライセ
ンス条件やその他法的な拘束条件をライブラリユーザーに伝える十分な手段とみなされるべきではな
い。
-->
<!ELEMENT legal-notice (#PCDATA)>
```

```
<!--
icon 要素には、GUI ツールで Web アプリケーションを表すのに使用する小さなイメージと大きなイメージの、そのアプリケーション内の位置を示す small-icon と large-icon 要素を含める。最低限、ツールは GIF 形式のイメージを受け付ける必要がある。
-->
<!ELEMENT icon (large-icon?, small-icon?)>

<!--
large-icon 要素には、大きさアイコンイメージ (32x32 ピクセル) を含むファイルの、ライブラリ内のリソース名を含める。リソース名は、個々のパス要素をスラッシュ (/) で区切った標準の Java リソース名構文に従っている必要がある。
-->
<!ELEMENT large-icon (#PCDATA)>

<!--
small-icon 要素には、大きさアイコンイメージ (16x16 ピクセル) を含むファイルの、ライブラリ内のリソース名を含める。リソース名は、個々のパス要素をスラッシュ (/) で区切った標準の Java リソース名構文に従っている必要がある。
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
author-info 要素には、このライブラリの作成者に関する情報を含める。
-->
<!ELEMENT author-info (author*, info-resource*) >

<!--
author 要素には、このライブラリの個人作成者に関する情報を含める。
-->
<!ELEMENT author (author-name, description?, author-contact?) >

<!--
author-name 要素には、作成者の氏名を含める。
-->
<!ELEMENT author-name (#PCDATA)>

<!--
author-contact 要素には、作成者の連絡先 (通常は電子メールアドレス) を含める。
-->
<!ELEMENT author-contact (#PCDATA)>

<!--
info-resource には、発行者のホームページへのリンク、API の資料への公開リンク、サポート用電子メールアドレスなどの、このライブラリに関係する外部情報リソースに関する情報を含める。
-->
<!ELEMENT info-resource (info-resource-name, description?, info-resource-
contact?) >
```

```
<!--
info-resource-name 要素には、このライブラリのユーザーに提供可能なリソースの任意の叙述名を
含める。
-->
<!ELEMENT info-resource-name (#PCDATA)>

<!--
info-resource-contact 要素には、電子メールアドレスや HTTP リンクなどの、リソースの実際の
連絡先情報を含める。
-->
<!ELEMENT info-resource-contact (#PCDATA)>

<!--
interface-version 要素には、含まれているコードのインタフェース互換性の判定に使用するこのラ
イブラリのインタフェースのバージョン情報を含める。このバージョン情報には、「1.0.0」というよ
うにドット区切りのバージョン番号を指定する。バージョン番号は、必要に応じてドットで区切った要
素を含むことができる。
-->
<!ELEMENT interface-version (#PCDATA)>

<!--
implementation-version 要素には、含まれているコードの実装バージョンの判定に使用するこのラ
イブラリのインタフェースのバージョン情報を含める。通常、このバージョン情報はタイムスタンプま
たは構築番号の形式にする。このバージョン情報には、「2003.1.31」というようにドット区切りの
バージョン番号を指定する。バージョン番号は、必要に応じてドットで区切った要素を含むことができ
る。
-->
<!ELEMENT implementation-version (#PCDATA)>

<!--
taglib 要素には、このライブラリに含まれるすべての JSP タグライブラリを宣言する。
宣言されたタグライブラリは、自動的にライブラリかアンパックされ、アプリケーションの web.xml
内のこの URI 下に登録される。
-->
<!ELEMENT taglib (taglib-uri, taglib-resource, taglib-default-prefix)>

<!--
taglib-uri 要素には、アプリケーション内のタグライブラリの識別に使用する論理 URI を含める。
この URI は、宣言されているタグライブラリ記述子に応じて、アプリケーションの web.xml に登録さ
れる。この URI は純粋に論理的なものであり、タグライブラリ記述子ファイル（GUI ツール単独で決
定された物理的な場所にアンパックされる）の物理的な場所とは何の関係もないことに注意。この URI
は、コンポーネントの JspTagDescriptor 内の TaglibURI プロパティ値に一致している必要がある。
-->
<!ELEMENT taglib-uri (#PCDATA)>

<!--
```

```
taglib-resource 要素には、タグライブラリの taglib 記述子ファイル (.tld) のリソース名を含
める。リソース名は、個々のパス要素をスラッシュ (/) で区切った標準の Java リソース名構文に
従っている必要がある。このファイルは自動的に抽出されて、アプリケーションに登録される。
-->
<!ELEMENT taglib-resource (#PCDATA)>

<!--
taglib-default-prefix 要素には、タグライブラリを使用する JSP ページで、このタグライブラリ
に使用するタグ接頭辞を指定する。たとえば Web アプリケーションフレーム/JATO タグライブラリの
デフォルトの接頭辞は「jato」。
この接頭辞は、任意のページで JSP 作成者が変更できる。この要素は、タグライブラリ宣言がページに
自動的に追加されるときの接頭辞のデフォルト名を提供するにすぎない。
-->
<!ELEMENT taglib-default-prefix (#PCDATA)>

<!--
component 要素には、このライブラリ内の拡張不可コンポーネントを宣言する。
コンポーネントマニフェスト内ですべてのコンポーネントを宣言して、デザイン時に認識されるように
する必要がある。
-->
<!ELEMENT component (component-class, component-info-class, design-reference-
resource?)>


<!--
extensible-component 要素には、このライブラリ内の拡張可能コンポーネントを宣言する。コン
ポーネントマニフェスト内ですべての拡張可能コンポーネントを宣言して、デザイン時に認識されるよ
うにする必要がある。
-->
<!ELEMENT extensible-component (component-class, component-info-class)>

<!--
component-class 要素には、コンポーネントの完全限定クラス名を指定する。
-->
<!ELEMENT component-class (#PCDATA)>

<!--
component-info-class 要素には、コンポーネントの ComponentInfo クラスの完全限定名を指定
する。

-->
<!ELEMENT component-info-class (#PCDATA)>

<!--
design-reference-resource 要素には、デザイン時のコンポーネントの点検に使用するメタデータ
ファイルリソースを指定する。一般にこの宣言のないコンポーネントは、ComponentInfo を使わない
限りデザイン時に点検できない。リソース名は、個々のパス要素をスラッシュ (/) で区切った標準の
Java リソース名構文に従っている必要がある。
-->
```

```
<!ELEMENT design-reference-resource (#PCDATA)>

<!--
configurable-bean 要素には、このライブラリに含まれている不可視 Bean コンポーネントを宣言する。
-->
<!ELEMENT configurable-bean (bean-class)>

<!--
configurable-bean 要素には、不可視コンポーネント Bean の完全限定クラス名を指定する。
-->
<!ELEMENT bean-class (#PCDATA)>
```

コンポーネントのタグライブラリ (TLD) ファイルの自動アンパック

ライブラリ開発者は Web アプリケーションフレームワークコンポーネントライブラリの一部として、ライブラリのビューコンポーネントの生成をサポートする 1 つ以上のタグライブラリを提供できます。タグライブラリはコンポーネントライブラリのコンポーネントマニフェストファイルで宣言され、IDE がコンポーネントライブラリを認識すると、そのタグライブラリ記述子 (.tld ファイル) がアプリケーションで使用できるようにライブラリ JAR ファイルから自動的にアンパックされます。さらに IDE は、タグライブラリエントリを `web.xml` ファイルに自動的に追加します。

別のライブラリの同名のファイルが衝突しないように、タグライブラリ記述子ファイルはライブラリの名前に基づいてアプリケーションの `WEB-INF/tld` ディレクトリの下の特別な場所にアンパックされます。このスキーマでは、ドット (.) が下線 (_) に置き換えられて、ライブラリ名がディレクトリ名に変換されます。たとえば、Web アプリケーションフレームワークコンポーネントライブラリの内部ライブラリ名が「com.iplanet.jato」の場合は、タグライブラリ記述子のアンパック時に「com_iplanet_jato」に変換されます。Web アプリケーションフレームワークコンポーネントライブラリのタグ記述子ファイルは、最終的にはアプリケーションの `WEB-INF/tld/com_iplanet_jato` ディレクトリの下に配置されます。

派生した、タグ記述子の物理ディレクトリ名は、アプリケーションが使用できるように `web.xml` ファイルの論理リソース名に自動的に登録されます。この論理名は、コンポーネントライブラリの作成者が選択し、コンポーネントライブラリマニフェストに指定します。Web アプリケーションフレームワークコンポーネントライブラリの場合、記述子はリソース `/WEB-INF/jato.tld` として登録されます。

タグ記述子のアンパック機構は、タイムスタンプを使用して、新しいバージョンのライブラリがアプリケーションに追加されたときに既存ファイルを上書きするかどうかを判断します。この機能により、開発者は 1 つのステップだけでアプリケーションのコンポーネントライブラリをアップグレードすることができます。

このマニュアルで説明している「mycomponents」ライブラリの例を参照してください。ライブラリ作成者は mycomplib.tld という名前のタグライブラリ tld ファイルを作成し、mycomponents パッケージ内に任意の場所に配置しています。このため、mycomponents.jar の中を見ると、tld は以下のような物理位置に現れます。

```
/COMP-INF/complib.xml
/mycomponents/*.class
/mycomponents/mycomplib.tld
/mycomponents/...
```

complib.xml 内では、コンポーネント作成者は taglib 要素を以下のように宣言しています。:

```
<taglib>
      <taglib-uri>/WEB-INF/mycomplib.tld</taglib-uri>
      <taglib-resource>/mycomponents/mycomplib.tld</taglib-resource>
      <taglib-default-prefix>mycomp</taglib-default-prefix>
</taglib>
```

mycomponens.jar が Web アプリケーションフレームワーク Web アプリケーションの WEB-INF/lib ディレクトリに配備されている場合、IDE は、taglib 要素の表す構成に基づいて、いつでも自動的に以下の作業をします。この作業によって、実行時 JSP エンジンは正しくタグライブラリのある場所を見つけることができます。このため、アプリケーション開発者が構成を行う必要はありません。

- Web アプリケーションの web.xml ファイルに以下のエントリを追加する。この web.xml によって、論理リソースととその物理的な場所との従来のサーブレットコンテナの実行時マッピングが作成されます。

```
<taglib>
      <taglib-uri>/WEB-INF/mycomplib.tld</taglib-uri>
      <taglib-location>/WEB-INF/tld/mycomponents/mycomplib.tld</taglib-location>
</taglib>
```

- mycomponents.jar から mycomplib.tld を抽出して、次の場所に置く。:

```
[現在のアプリケーション]/WEB-INF/tld/mycomponents/mycomplib.tld
```

また、デザイン時、開発者がアプリケーションビューを構築するときに、IDE は以下のことを行います。

- 対応する JSP ファイル内に適切なタグライブラリ宣言が存在するようにする。この宣言には、complib.xml の taglib 要素内にコンポーネント作成者が指定した「接頭辞」が含まれます。

```
<%@taglib uri="/WEB-INF/mycomplib.tld" prefix="mycomp"%>
```

- IDE がコンポーネントライブラリ固有の追加タグを JSP に追加するときに、IDE が taglib 命令に宣言されている接頭辞を利用するようにする。以下はその例です。

```
<mycomp:validatingTextField name="validatingTextField1"/>
```

Web アプリケーションフレームワークは、taglib 接頭辞が JSP ページ固有の命令であると認識します。J2EE は、1 つのページで taglib 命令を使用して、含まれているタグライブラリに対する任意の接頭辞を宣言することを可能にしています。IDE は、JSP を構文解析してタグを見つけるとき、あるいは、開発者のビューデザインの決定事項とともに自動的に追加のタグを JSP に挿入する場合、つねに現在の taglib 命令が宣言している接頭辞を利用します。このとき IDE は単に、complib.xml に指定された taglib「接頭辞」を使用して、アプリケーションの JSP ファイルに初期 taglib 命令を挿入するだけです。その後で、アプリケーション開発者は、ページ固有の taglib 命令に宣言されている接頭辞を手動で変更することができます。以降、IDE は追加のタグのすべてに新しく宣言された接頭辞を使用しますが、自動的には、すでに宣言されているタグを、調整された接頭辞と符号するように変更することはありません。これは、アプリケーション開発者の問題です。このことをここで言及しているのは、コンポーネント作成者が、complib.xml の taglib 要素のデザイン時の用途を完全に理解してもらうためです。

「追加ファイル」の自動アンパック

Web アプリケーションフレームワークのコンポーネントライブラリ JAR には、オプションで、/webapp ディレクトリの下に階層的に編成された任意の「追加ファイル」が含まれていることがあります。

Web アプリケーションフレームワークのコンポーネントライブラリ JAR の内部「/webapp」ディレクトリと、「webapp」という共通サーブレットコンテナディレクトリを混同しないでください。両者には何の関係もありません。

Web アプリケーションフレームワークのコンポーネントライブラリ JAR の /webapp ルート内のファイルを階層的に編成するかどうかは、完全に、コンポーネントライブラリ作成者の自由です。付加価値機能として、Web アプリケーションフレームワーク IDE は、/webapp ルートを基準にした追加ファイルの場所に直接対応して、これらの追加ファイルを Web アプリケーション開発環境に「アンパック」します。

注 – これは、Web アプリケーションフレームワーク IDE がコンポーネント作成者に提供する、純粋に付加価値、完全にオプションの「リソース配布」の機会です。前提になることは、抽出されたファイルによって、コンポーネント作成者が決定した任意のデザイン時または実行時値を提供するということです。また、この任意値を提供するには、ファイルを Web アプリケーションのファイルシステムに抽出する必要があることも前提になります。それ以外の場合、JAR の「追加ファイル」(たとえば /webapp) の部分に値を入れる必要はなく、むしろ、JAR 内の従来の場所に置いて、Java 実行時に選択されるようにします。

たとえば、次の /webapp 構造を含む mycomponents.jar を考えてみます。:

```
/mycomponents/...
/mycomponents/mycomplib.tld
/webapp/mycomponents/foo.jsp
/webapp/mycomponents/bar.jsp
/webapp/mycomponents/images/banner.gif
/webapp/WEB-INF/jato/templates/jsp/MyViewBeanJSP.jsp
/webapp/WEB-INF/jato/templates/jsp/FooContainerViewJSP.jsp
/webapp/WEB-INF/lib/helper.jar
/webapp/WEB-INF/mycomponents/config-files/configA.xml
/webapp/WEB-INF/mycomponents/config-files/configB.xml
```

上記の内容を持つ mycomponents.jar が Web アプリケーションフレームワークアプリケーションに配備されている場合、IDE は /webapp の内容をその Web アプリケーションフレームワークの構造に抽出します。

たとえば、Web アプリケーションフレームワーク IDE によって作成された次の初期構造を持つ、「AppOne」という Web アプリケーションフレームワークアプリケーションを考えてみます。

```
AppOne/index.html
AppOne/WEB-INF/classes/...
AppOne/WEB-INF/jato/templates/jsp/DefaultViewBeanJSP.jsp
AppOne/WEB-INF/jato/templates/jsp/DefaultContainerViewJSP.jsp
AppOne/WEB-INF/lib/jato-2_1_0.jar
AppOne/WEB-INF/tld/com_iplanet_jato/jato.tld
```

mycomponents.jar が配備 (たとえば AppOne/Web-INF/lib にドロップ) されると、IDE は、それが Web アプリケーションフレームワークのコンポーネントライブラリであることを認識して「追加ファイル」を抽出し、その結果として、次の統合構造が作成されます。

```
AppOne/index.html
AppOne/mycomponents/foo.jsp
AppOne/mycomponents/bar.jsp
AppOne/mycomponents/images/banner.gif
AppOne/WEB-INF/classes/...
AppOne/WEB-INF/jato/templates/jsp/DefaultViewBeanJSP.jsp
AppOne/WEB-INF/jato/templates/jsp/DefaultContainerViewJSP.jsp
AppOne/WEB-INF/jato/templates/jsp/MyViewBeanJSP.jsp
AppOne/WEB-INF/jato/templates/jsp/FooContainerViewJSP.jsp
AppOne/WEB-INF/lib/jato-2_1_0.jar
AppOne/WEB-INF/lib/mycomponents.jar
AppOne/WEB-INF/lib/helper.jar
AppOne/WEB-INF/mycomponents/config-files/configA.xml
AppOne/WEB-INF/mycomponents/config-files/configB.xml
AppOne/WEB-INF/tld/com_iplanet_jato/jato.tld
AppOne/WEB-INF/tld/mycomponents/mycomplib.tld
```

コンポーネントライブラリの作成者は、追加ファイル機能を利用して、抽出に向いていると思われる任意のリソースを提供することができます。これだけに限りませんが、よくある追加ファイル例としては、以下があります。

- コンポーネント固有の JSP ファイル (コンポーネントページレットなど)
- 任意の Web アプリケーションドキュメントリソース (イメージ、静的 HTML ページ、スタイルシート、JavaScript など)
- 任意の追加 JAR ファイル

  たとえばコンポーネントライブラリがカスタム XML 構文解析ライブラリに依存していると仮定します。コンポーネント作成者は、コンポーネントライブラリ JAR にその JAR を「バンドル」できます。可能性としては、これは、アプリケーション開発者が余分なライブラリを手動で配備するよりも便利な配布モデルです。
- 任意の Web アプリケーション WEB-INF リソース

  たとえば、コンポーネント作成者が、任意の構成ファイルを使用して特別な構成をサポートする一群のコンポーネントを作成したと仮定します。その場合、それら構成ファイルをコンポーネントライブラリと「バンドル」し、追加ファイル機能を使用して適切な場所に抽出することができます。

注 – コンポーネントライブラリの tld ファイルの自動抽出は、別の機構で処理されます。コンポーネントライブラリの tld ファイルは、`/webapp` ルート下に置かないでください。むしろ、コンポーネントライブラリ JAR 内の、通常の「リソース」に適切な場所 (たとえば上記の `mycomplib.tld`) に置いてください。詳細は、183 ページの「コンポーネントのタグライブラリ (TLD) ファイルの自動アンパック」を参照してください。

索引

X

あ

お

か

こ

し