Rational Software Modeler、Rational Systems Developer、および Rational Software Architect のための モデル構造ガイドライン (「従来の RUP」のオリエンテーション)

ホワイト・ペーパー

Bill Smith、Model Driven Development、IBM Rational Software

V7.0
2006 年 3 月 24 日

目次

1. はじめに

対象読者

このホワイト・ペーパーは、Rational Software Architect (RSA)、Rational Systems Developer (RSD)、および Rational Software Modeler (RSM) 製品 (以下、併せて「**RS*x***」といいます) のユーザー、特に Rational Unified Process (RUP®) のモデリングのガイダンスを RSx に適用したいと考えているユーザーを対象としています。Rational Software Modeler (RSM) のユーザーにも役立ちますが、RSM には存在しない機能 (RSA および RSD だけに用意されている機能) に関する説明が含まれていることに注意してください。

このホワイト・ペーパーでは、RS*x* での新しいモデル・セットの作成を中心に扱っています。RS*x* を使い始めたばかりのユーザーであっても、Rational Rose または Rational XDE を使用していたことがあり、それらの製品からのモデルのインポートを計画しているのであれば、インポートしたモデルを再編成するための参考資料として、このホワイト・ペーパーを利用できます。

このホワイト・ペーパーでは、RUP と UML に関する知識をある程度持っているユーザーを想定しています。また、RS*x* の基本的な概念や「動作理論」をユーザーが理解していることも想定しています。これについては、「The New IBM Rational Design and Construction Products for Rose and XDE Users」で詳しく説明しています。

目的

RUP で記述しているモデル・セット (ユースケース・モデル、分析モデル、設計モデルなど) は、システムの問題/解決領域で入念に定義されたさまざまなパースペクティブを反映しています。このモデル・セットの利便性は、数多くの実世界プロジェクトで実証されています。RUP に準拠しない場合であっても、これらのモデル構造には検討するだけの価値があります。このホワイト・ペーパーでは、RS*x* を使用してこれらを実際に作成する方法を解説します。

その他のモデリング方法についても考察しますが、それらの方法をサポートするためのモデル編成のガイドラインは、このホワイト・ペーパーの対象外です。たとえば、サービス指向アーキテクチャーのためのモデリングに対しては、「ビジネス駆動型開発」という方法を使用できることがあります。これは、ビジネス・プロセス・モデル (UML 2 アクティビティー・モデルとして表される場合もあります) から始まり、ビジネス・プロセス内のタスクを自動化するサービス・セット契約の規定に直接進みます。その方法では、このホワイト・ペーパーで説明している内容とはまったく異なるモデル構造が推奨されることがあります。

重要なことは、このホワイト・ペーパーで取り上げるプロジェクトとモデル構造は、あくまでもガイドラインであって、絶対的な規則ではないということです。RS*x* で特定の RUP 成果物をモデリングするかどうかは、それぞれの開発プロセスで検討しなければならない事柄であり、プロジェクトごとに決定内容が違ってくる場合もあります。また、RUP はプロセスに関する厳密な規則集ではないということも覚えておいてください。むしろ、これはプロセスの定義を規定する「枠組み」です。このようなプロセス定義は、非常に厳密なものから大まかなものまで、さまざまな範囲に及びます。

UML モデリングの使用法についても、厳密に定義されたものもあれば、大まかに定義されたものもあります。正式なアーキテクチャー図面のようにモデルを扱って、実際の作成段階でも厳密にそのモデルに従って作業する場合もあれば、設計の概略を示したスケッチ程度にモデルを扱って、プロジェクトが実装の段階に入ったら破棄するような場合もあるわけです。プロセスやモデリングに関する、こうした両極端のケースに、RS*x* は対応することができます。このホワイト・ペーパーのガイドラインは、自由な発想を妨げるものではありません。それぞれの状況でベストと思えるプロセスに合わせて、RS*x* の各機能を活用する方法を考え出すために、利用してください。

さらに RS*x* によって、モデルを単なる青写真として使用するだけでなく、実装の重要な部分を自動生成するための仕様書として使用できるようになります。そのために活用できるのが、モデルからモデルへの変換機能と、モデルからコードへの変換機能です。変換機能を使って Model-Driven Development (MDD) を行なう場合、モデル構造に関して特別に注意しなければならない点がいくつかあります。モデルと変換機能を使用して MDD を行なう場合は、developerWorks® の Rational 設計エリアおよび作成エリアにある、RS*x* MDD 固有の資料もご覧ください。

範囲

このホワイト・ペーパーでは、RUP のモデル成果物を RSx で表現する方法について解説し、そのような成果物の内部編成構造のためのガイドラインも紹介します。以下の内容は、*取り上げていません*。

- RUP のモデル成果物の概念的な基盤に関する、詳細な説明
- RUP の成果物に関する詳細なセマンティクスや図表を指定するための、プロセスやテクニック

RUP 成果物の定義方法、開発方法、モデリング方法に関する一般的な (ツールに依存しない) 情報については、RUP のマニュアルを参照してください。

RS*x* モデルの開発に関するツール固有のテクニックについては、以下の資料を参照してください。

- 製品の資料 (チュートリアル、サンプル、オンライン・ヘルプ)
- RUP 構成に含まれているツール・メンター (このホワイト・ペーパーも、その一部です)
- developerWorks の RS*x* 関連リソース

表記上の規則

IBM Rational Rose または XDE から移行する RS*x* ユーザーにとって特に役立つ情報は、薄い灰色のテキスト・ボックス内に、補足情報の形式で記載されています。

> ***XDE/Rose***
> 旧バージョンの XDE または Rose のユーザーに役立つ説明。

このホワイト・ペーパーの構成

『基本的な概念と用語』のセクションでは、重要な用語について説明し、RS*x* 製品でモデルを実装する方法の概略を紹介します。

次に、『RUP モデルと RSx モデルの対応関係』のセクションでは、RUP で定義されているモデル・タイプを RS*x* がどのようにサポートしているのかを取り上げます。

その後のセクションでは、各種タイプのモデルを構成するためのガイダンスを提供しています。プロセス、モデリング方法、アーキテクチャー制御などにおいて、基準の厳密さに応じてモデルを使用する方法を説明しているセクションもあります。

最後に、チームでモデルを使用する際の問題点を取り上げます。ファイルの競合やマージを最小限に抑えることを目的にした、スケールの管理およびチーム・メンバー間でのモデル共有化のための戦略についても、ここで取り上げています。

2. 基本的な概念と用語

モデル

RUP では、「特定のパースペクティブから問題/解決領域を記述した、完全な仕様書」をモデルと定義しています。問題領域またはシステムを指定する際に、その領域またはシステムのさまざまなパースペクティブに対応するモデルを複数使用する場合があります。RUP では、以下のような特定のモデル・セットを提案しています。

- ビジネス・ユースケース・モデル
- ビジネス分析モデル
- ユースケース・モデル
- 分析モデル (設計モデルの中に含まれる場合があります)
- 設計モデル
- 実装モデル
- 配置モデル
- データ・モデル

RUP は、ツールとは無関係です。RUP に関する限り、ナプキンやホワイトボードに描いた図であっても、モデリング・ツールで作成したファイルであっても、単に頭の中に思い描いたイメージであっても、すべてがモデルになりえます。RUP の観点からすれば、モデルとは論理的な概念になります。

RS*x* の文脈でモデルを取り上げる場合、論理的な観点から取り上げることもあれば、物理的な観点から取り上げることもあります。たとえば、2 つのチームが 2 つのアプリケーションを扱っているとします。1 つは、3 人の分析者からなるチームで、タイムシート管理アプリケーションで作業を行なっています。もう 1 つは、5 人の分析者からなるチームで、コール・センター・アプリケーションで作業を行なっています。どちらのチームも、さまざまな要件を取り込んでいる段階であり、ユースケース・モデリングのために RS*x* を使用しています。RUP の観点からすると、一方のチームは「タイムシート管理アプリケーションのためのユースケース・モデル」を構築していることになり、もう一方のチームは「コール・センター・アプリケーションのためのユースケース・モデル」を構築していることになります。しかし、チームで RS*x* を使用するのであれば、それぞれのモデルに物理的な側面があることを理解しておくことが重要です。これについては、次のセクションで取り上げます。

モデリング・ファイル

RS*x* モデルは、ファイルとして保持されます。(Eclipse の用語では、ファイルは「リソース」と同義になります[1]。このホワイト・ペーパーや他の資料で「モデリング・リソース」という言葉が出てきたら、それは「モデリング・ファイル」と同じ意味になります。)RS*x* は、広い意味においてのモデリング・ファイルを 2 種類サポートしています。

[1] Eclipse では、リソースはファイルになりますが、Eclipse 環境内の付加的なプロパティーや動作も含みます。ここで説明しているモデリング・ファイルは、Eclipse では「リソース」として取り扱われます。

- **実装前モデリング・ファイル**は、ホスト OS のファイル・システム内に個別のファイルとして保管されます。ファイル名には、「.emx」という拡張子が付きます。このファイルには、次のものが含まれています。
    - UML セマンティクス要素 (クラス、アクティビティー、関係など)
    - UML セマンティクス要素を記述した (場合によっては、Java、C++、または DDL のようなその他のセマンティクス領域内の要素に対する、目に見える参照も記述した) UML 図

- **実装モデリング・ファイル**は、Eclipse ワークスペース内に Eclipse *プロジェクト*として保管されます。このプロジェクトには、次のものが含まれます。
    - 実装成果物 (Java ソース・コード、Web ページ、XML ベースのメタデータ・ファイルなど)
    - 実装成果物を記述して直接反映する、ダイアグラム・ファイル

  モデルのセマンティクスは、実装成果物そのものの中に入っています。たとえば Java 実装のセマンティクス・モデルは、Java ソース・コード・ファイル群としてシリアライズされて、保管されます。それぞれの図は、プロジェクト内にある専用の物理ファイルの中に入っています。ダイアグラム・ファイルには、さまざまな拡張子を使うことができます。最も一般的なものは「.dnx」です。実装モデリング図では、UML の表記法を使用するのが一般的ですが、他の表記法を使用することもあります (データをビジュアル化するための IDEF1X 表記法や Information Engineering 表記法、Web 層を設計するための IBM 独自の表記法など)。

このホワイト・ペーパーで主に取り上げるのは、「実装前」モデルの内部構造を編成する方法についてです。**これ以降に登場する「モデリング・ファイル」という用語は、「実装前」モデリング・ファイル (拡張子が .emx のファイル) のことを指します。**実装プロジェクトの内容を編成するためのガイダンスについては、Rational Software Architect、Rational Application Developer、Rational Web Developer Community Edition のオンライン・ヘルプなどの資料をご覧ください。

モデリング・ファイルには、1 つの (論理) モデルに関する情報がすべて含まれているとは限りません。実際には、モデルのサブセットだけがモデリング・ファイルに含まれることもよくあります。先ほどの例の場合、タイムシート管理アプリケーションのユースケース・モデルで作業をしているチームには、3 人のメンバーがいました。そのようなチームでは、ユースケース・モデルを物理的に 3 つのモデリング・ファイルに分割し、それぞれのメンバーがユースケースの別々のサブセットで作業を行うようにすることで、同じファイルに関する作業の競合を避けることができます。このホワイト・ペーパーの最後のセクションでは、モデルを分割して複数のモデリング・ファイルを管理する場合の問題点を取り上げます。

モデルのタイプ

RUP のモデルには、ユースケース・モデル、分析モデル、データ・モデルなどの、固有のタイプがあります。RS*x* では、モデリング・ファイルは「強く型定義されている」わけではありませんが、「弱く型定義された」複数のモデリング・ファイルを使用する際の規則に従うことができます。この規則に従う場合は、次の 2 つの方法のいずれかで弱い型定義を設定することができます。

- 最初は「ブランクの」モデリング・ファイルにしておき (以下を参照)、その後で名前の付け方やファイルに含める内容 (ファイルに適用する UML プロファイルなど) によって型を設定します。

- 特定のモデル・タイプに相当する、事前定義の「テンプレート・モデル」に基づいて、モデリング・ファイルを作成します。RSx 製品には、このホワイト・ペーパーで説明しているモデル・タイプ用の、テンプレート・モデルのデフォルト・セットが用意されています。また、独自のテンプレート・モデルを作成することもできます (製品の「ヘルプ」、および developerWorks のフォーラムやその他のリソースを参照してください)。

いずれにしても、RSx のモデリング・ファイルの「タイプ」とは、実際にはファイルの命名や内容に関するきまりごとに過ぎません。たとえば、このツールによって、ユースケース・モデルを含むファイルに、ユースケースを実現するクラスも含まれないようにすることはできません (論理的な観点では、RUP は分析モデルや設計モデルの一部と見なされます)。

このホワイト・ペーパーで紹介するガイドラインでは、RSx モデリング・ファイルを型定義して扱うことを、強く推奨します。

ワークスペース、プロジェクト、プロジェクト・タイプ

Eclipse、WebSphere Studio 製品群、Rational Application Developer のいずれかに精通している読者であれば、各ファイルがプロジェクトに含まれていること、プロジェクトにはさまざまなタイプがあること、そして各プロジェクトがワークスペースの中でグループ化されて管理されていることをご存じでしょう。RS*x* のモデリング・ファイルも、他のファイルと同様に、プロジェクトの中に入っています。

本書の目的に沿って、RSx と Rational Application Developer で使用可能な、すべてのプロジェクト・タイプに関する詳しい説明は、ここでは省略します。重要なのは、以下の 2 つのプロジェクト・カテゴリーです。

- **UML プロジェクト**
- **実装プロジェクト** (エンタープライズ・プロジェクト、EJB プロジェクト、Web プロジェクト、C++ プロジェクトなどの特殊なタイプも含む)

すでに述べたとおり、RS*x* は 2 種類のモデリング・ファイルをサポートしています。

- .emx 拡張子を持ち、UML モデルを含むファイル (実装の上にある抽象レベル (要件、分析、設計) でモデリングを行うために使用する)
- 実装セマンティクス (通常はソース・コード・ファイル成果物としてシリアライズされる) を含む Eclipse プロジェクトと、それらのセマンティクスを反映する図

モデルをプロジェクトに割り振るためのルールは、単純です。

a) 「実装前」モデリング・ファイルは、UML プロジェクト内に配置する

b) 基本的に以下の等式が成り立つため、実装モデルには自分自身を処理させる。

[実装モデル] = [実装プロジェクト]

ただし、このルールにはいくつかの例外があります。たとえば、以下の UML モデリング・ファイルは、言語固有の実装プロジェクトに組み込むのが妥当です。

- 設計「スケッチ・モデル」(後述)

- プロジェクト内のコードに対して実行されるテストを記述した、シーケンス図を含むモデル

> ***XDE/Rose***
> Rose と XDE の操作理論には、設計モデルを反復的に洗練しながら、最終的にコードに相当する抽象化レベルにまでもっていった後、コードとモデルの同期テクノロジーを使用して、モデルのセマンティクスをコードそのものと一致させるという方法が含まれています。たとえば XDE では、実装モデルはプロジェクト内のコードおよび図として存在するだけでなく、「コード・モデル」ファイルとしても存在します。そのコード・モデル・ファイルは、実装成果物から独立して保持されるもので、本質的にはそれらの実装成果物のセマンティクスの冗長なコピーを表しています。
>
> RSx の操作理論では、コードより抽象化レベルの高いプラットフォームに中立なモデル (つまり、Enterprise IT 設計モデルなどの設計モデル) を使用することや、変換機能を使用してこれらのモデルからコードを生成することが推奨されています。コード・レベルの抽象化では、RSA、RSD および RAD を使って、UML の表記法で表わされるコード・セマンティクスの図を作成することしかできません。実装レベルの抽象化においては、個別に保持されているセマンティクス・モデルを使った方法は省略されます。
> ただし、RSx でも、コード・レベルの抽象化において UML モデルを定義し、それらのモデルからコードを生成することは不可能ではありません。実際に、そのような使用法も想定されています。しかし、そのようなモデルとコードを自動的に同期させるテクノロジーは、RSx では提供されていません。

概念のまとめ

ここまでの内容を、以下の図にまとめます。この図は、コール・センター・アプリケーションで作業をするチームとタイムシート管理アプリケーションで作業をするチームがあるという、先ほどから取り上げてきたシナリオに基づくものです。

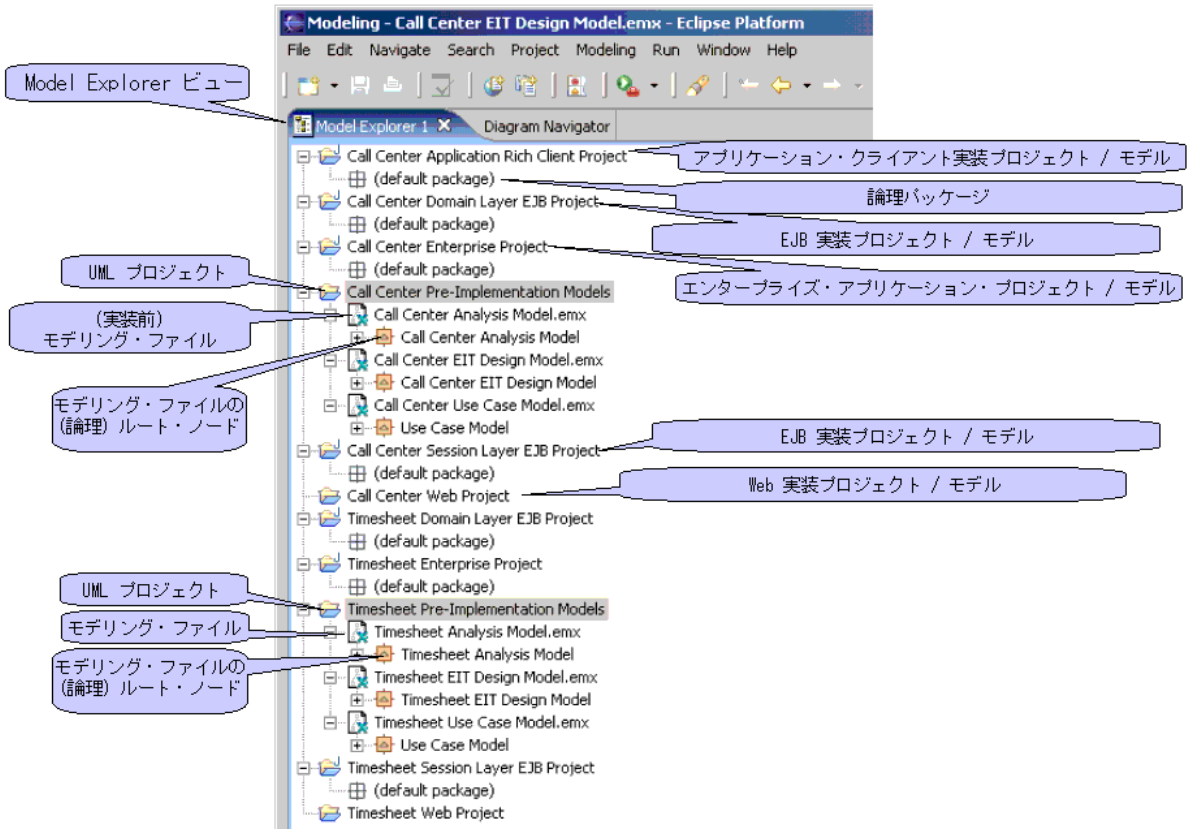


図 2-1

RSX では、モデルの物理ビューと論理ビューを組み合わせて表示するエクスプローラー・ビュー[2]が用意されています。このエクスプローラーでは、ワークスペース内の各プロジェクトが最上位ノードとして表示され、各プロジェクトの中にそれぞれのリソースが表示されます。**図 2-1** に示されているのは、先ほどのシナリオにあった 2 つのアプリケーションに相当するプロジェクト群が、Model Explorer で表示されている図です。UML プロジェクト (実装前モデルのために使用)、および各ソリューションに適したタイプのプロジェクト群 (実装モデルのために使用) も表示されています。

> ***XDE/Rose***
> RSA の Model Explorer とは違って、Rose や XDE の Model Explorer はモデルの論理ビューしか提供しません。RSA の Model Explorer が提供するリソースのビューは、Eclipse Navigator のビューが提供する「純粋に」物理的なビューではないことに注意してください。一部の物理リソースは Model Explorer にも表示されますが、大部分はリソースの「論理的な」ビューを示すアイコンによって表現されます。

[2] 6.x バージョンでは、「Model Explorer」でした。7.0 では、汎用のエクスプローラーにモデルが表示されます。このホワイト・ペーパーの図は 6.0 バージョンを基にしており、「Model Explorer」を使用しています。

図 1-2 では、タイムシート管理のユースケース・モデルをパッケージの形で内部編成して、問題領域を機能面から分割したサブセットに各パッケージを対応させる方法を説明しています。

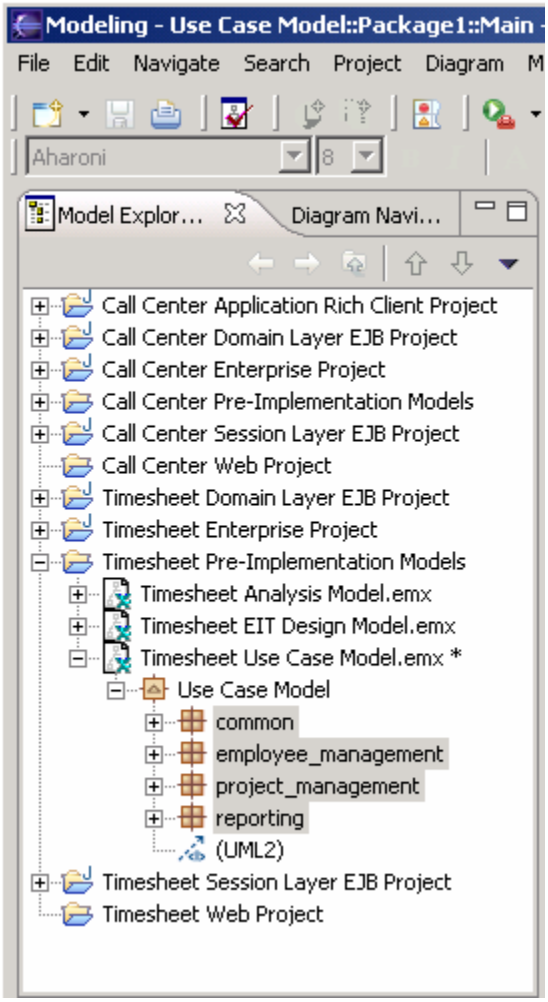


図 2-2

図 2-1 および 1-2 では、実装前モデルがそれぞれ単一のモデリング・ファイル内に入っています。図 1-3 では、タイムシート管理のユースケース・モデルを複数のモデリング・ファイルにリファクタリングして、その問題領域の同じサブセットに対応させる方法を説明しています。そのユースケース・モデル全体を構成するすべてのモデリング・ファイルにおいて、一貫した名前空間を保持できるような名前を、各モデリング・ファイルのルートにどのように付けているかに注目してください。

3. RUP モデルと RS*x* モデルの対応関係

最もよく使われる RUP モデルを、規則に従って RS*x* モデリング・ファイルの「タイプ」に対応させる方法を、以下の表にまとめます。この対応は基本的に単純なものですが、RSx を使って RUP を行なうためのガイドとして、このホワイト・ペーパーを活用するための重要な鍵となります。この表で示している RSx のモデリング・タイプについては、すぐ後の部分で解説します。各種モデル・タイプの内部編成や、各種モデル・タイプを組み込むプロジェクトの種類に関するガイドラインについては、後のほうのセクションで取り上げます。そのガイドラインも、ここに挙げている RSx モデリング・ファイルのタイプに基づいています。

RUP モデル	***RSx* モデリング・ファイルのタイプ**
ユースケース・モデル	「ユースケース・モデル」テンプレートを基にしたモデリング・ファイル (最初はブランクのモデリング・ファイルにしておき、RUP ユースケース・モデルのガイダンスによって内容を制限することもできます)
分析モデル	分析モデル (最初はブランクのモデリング・ファイルにしておき、RUP 分析モデルのガイダンスによって内容を制限することもできます) (設計モデル内では «analysis» パッケージを使うこともできます)
設計モデル	n 層のビジネス・アプリケーションの場合:「エンタープライズ IT 設計モデル」テンプレートを基にしたモデリング・ファイル (最初はブランクのモデリング・ファイルにしておき、RUP 設計モデルのガイダンスによって内容を制限することもできます) 他のタイプのアプリケーションの場合:最初はブランクのモデリング・ファイルにしておき、RUP 設計モデルのガイダンスによって内容を制限できます 設計「スケッチ」の場合:ブランクのモデリング・ファイル オプションの補足要素:実装概要モデルとして使用する、ブランクの追加モデリング・ファイル
実装モデル	実装成果物と図ファイルを含んだ Eclipse プロジェクト
配置モデル	最初はブランクのモデリング・ファイルにしておき、RUP 配置モデルのガイダンスによって内容を制限できます

RS*x* モデリング・ファイルのタイプ

ブランクのモデリング・ファイル

RSx には、「ブランク・モデル」を作成するためのオプションが用意されています (「File」→「New」→「UML Model」→「Blank Model」)。「ブランク・モデル」とは、モデル・テンプレートに基づかないモデリング・ファイルです。特別なプロファイルは適用されていませんし、デフォルトの内容も「Main」という名前の図 (自由形式の) が 1 つあるだけです。**ブランクのモデリング・ファイルは、あらゆるタイプのモデルを作成するための出発点として活用できます。**ブランクのモデリング・ファイルにどんな名前を付けるか、どんな内容を定義するか、どんなプロファイルを適用するかによって、ユースケース・モデル、分析モデル、設計モデル、配置モデルなど、あらゆるタイプの RUP モデルを作成できます。

ユースケース・モデリング・ファイル

RSx には、モデル・テンプレートに基づいて「ユースケース・モデル」ファイルを作成するためのオプションが用意されています。このテンプレートのデフォルトの内容は、***図 3-1*** のとおりです。(「構成材料 (Building Block)」の内容や検索ストリングの使用法に関する説明は、本書の対象外となります。各テンプレートには説明や指示が含まれており、ほとんどの場合はそれで対応できるはずです。)

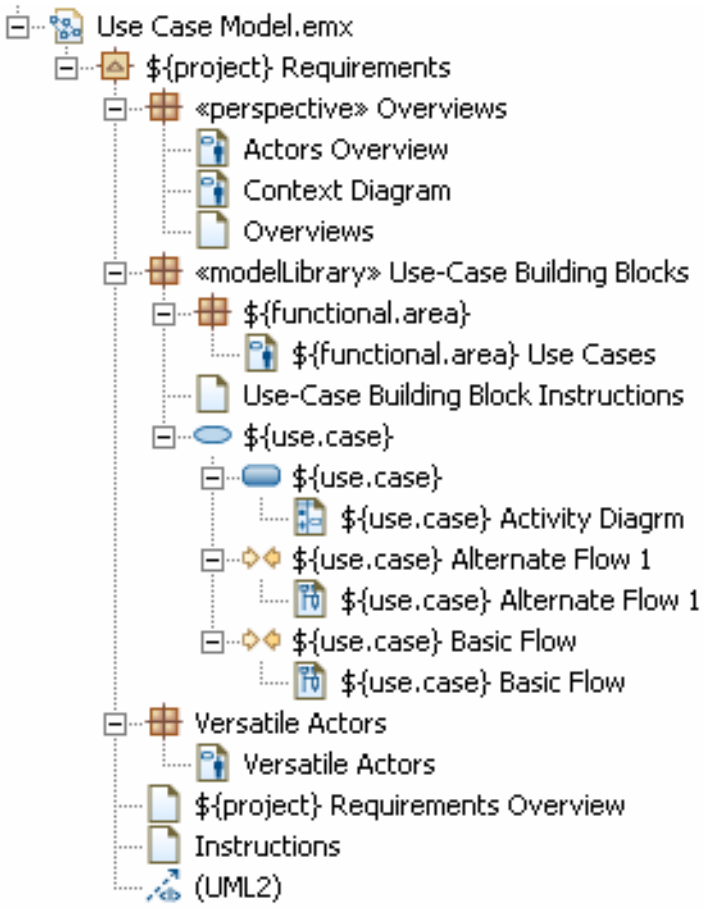


図 3-1

分析モデリング・ファイル

RSx には、モデル・テンプレートに基づいて「分析モデル」ファイルを作成するためのオプションが用意されています。このテンプレートのデフォルトの内容は、***図 3-2*** のとおりです。さらに、このテンプレートから作成されたモデル・ファイルには、「分析」プロファイルが適用されます。

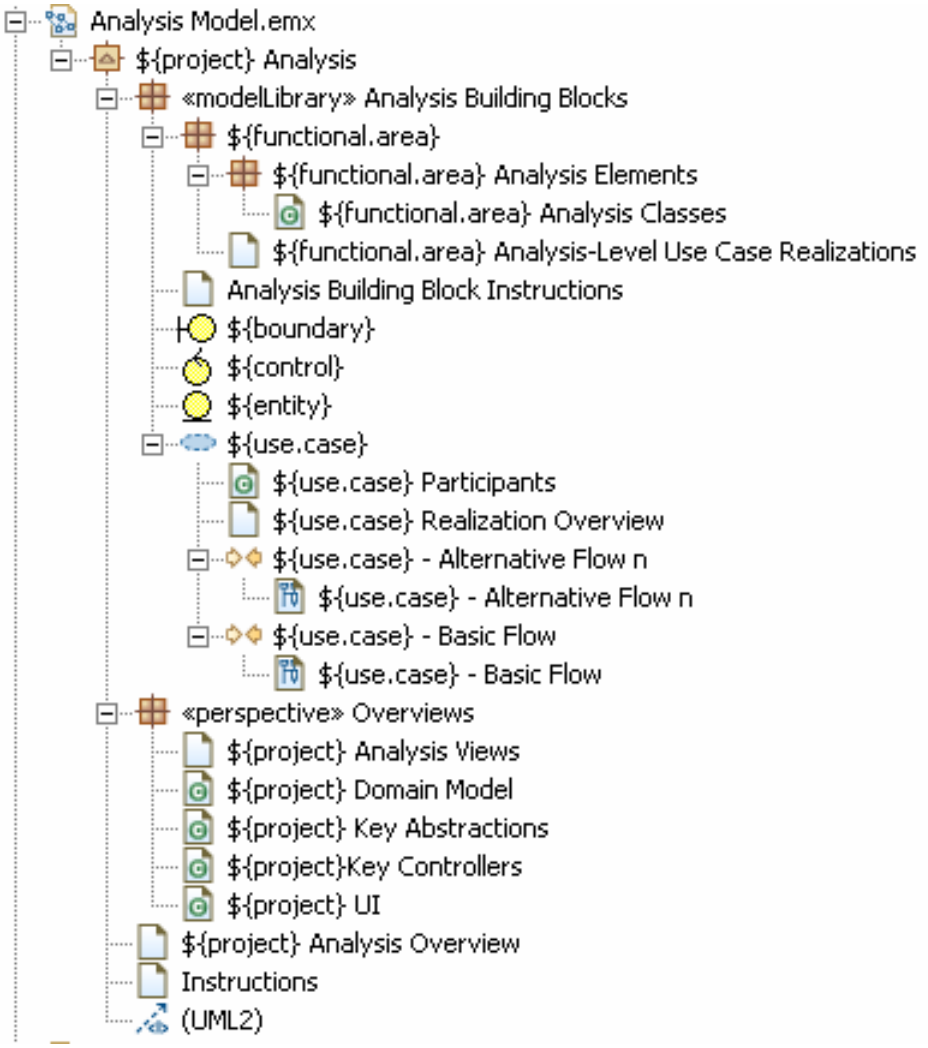


図 3-2

エンタープライズ IT 設計モデリング・ファイル

RSx には、モデル・テンプレートに基づいて「エンタープライズ IT 設計モデル」(EITDM) ファイルを作成するためのオプションが用意されています。このテンプレートのデフォルトの内容は **図 3-3** のとおりです。さらに、このテンプレートから作成されたモデル・ファイルには、「EJB 変換」プロファイル[3]が適用されます。ビジネス・アプリケーションをターゲットにして、そのようなアプリケーションの作成をサポートするために *RSX* のコード生成変換機能を使用する場合、このテンプレートは設計用 (オプションで分析用も可) の使用に適しています。

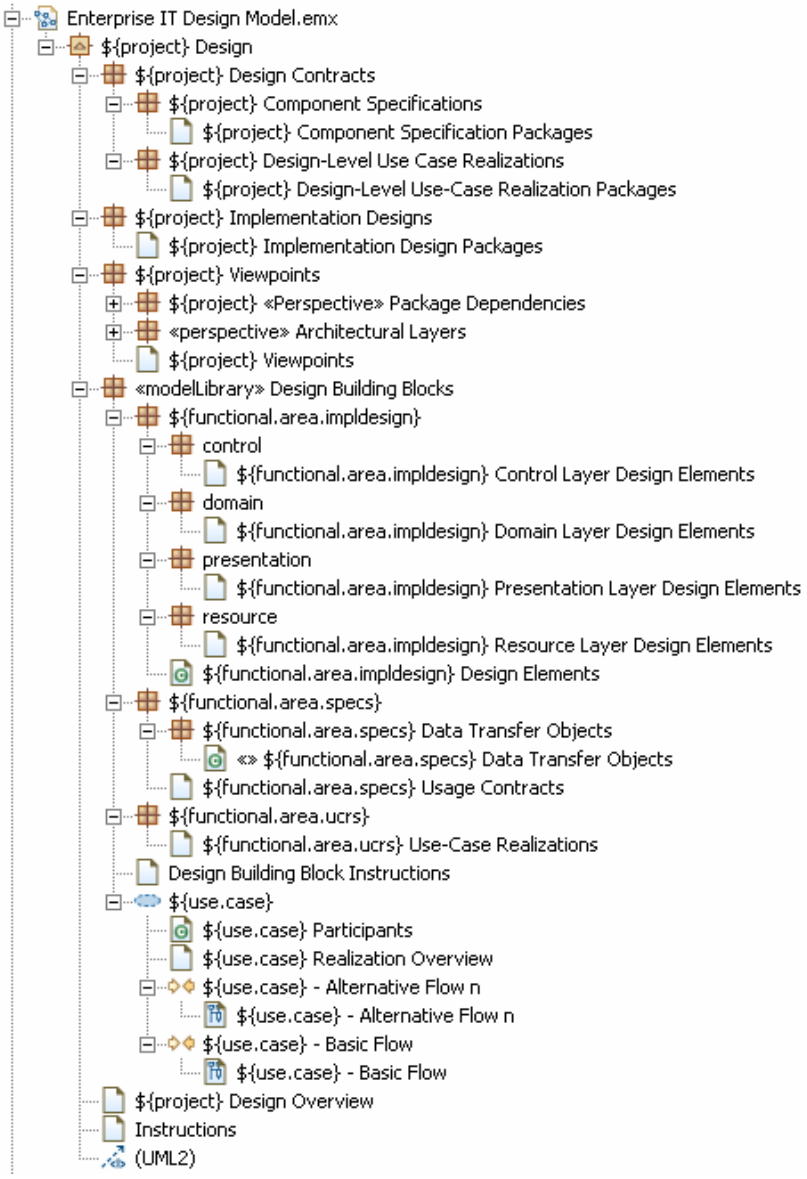


図 3-3

[3] 変換機能を可能にするプロファイル・セットは、EIT 設計モデル・テンプレートの一部として提供されており、製品のアップデート版がリリースされるたびに機能強化される場合が多くあります。

実装概要モデリング・ファイル

設計モデルの一部として「実装概要モデル」を定義すると、実装の編成方法の概要を取り込むために有益となる場合があります。「実装概要モデル」を使用するのは、設計の初期段階です。つまり、コードや関連ファイル (メタデータ、導入記述子など) を入れる実際の RSx または Rational Application Developer プロジェクトやフォルダー/パッケージのための、コードの生成や記述を行う前に使用します。また、このモデルを使用して、そのようなプロジェクトやパッケージの依存関係を示すことができます。これによって、システム構築の要件を容易に識別できるようになる場合があります。さらに、実装概要モデルは、ソリューション・アーキテクチャーのおおまかな概念図を入れておく場所にもなります。

実装モデル

すでに述べたとおり、RS*x* 内の実装モデルは、実装成果物 (およびオプションとして、その成果物を記述した図) を含んだプロジェクトから構成されています[4]。

「スケッチ」モデル

『基本的な概念と用語』のセクションでも述べたように、設計モデルを正式なアーキテクチャー図面として扱い、システムが存在する限り、それを維持して、アーキテクチャー制御のサポートや実施のために使用していくこともできれば、単に説明や検討のための設計案を示したスケッチ程度に扱い、実装に移行する段階で破棄することもできます。RSx は、この両方の使用法をサポートしています。基本的には、この 2 つの方法のいずれかをターゲットにして、RS*x* の各機能が割り振られているわけではありません。設計モデルをどちらの方法で使用するかに応じて、RS*x* のどの機能をどのように活用するかが決まってきます。このホワイト・ペーパー内で提示されるガイドラインにおいて、両者の区別が重要になってくる場合、「スケッチ・モデル」という用語を使用します。これは、そのモデルは「破棄できる」ものとして使用されていることを示しています。

4. モデルの内部構造を編成するための一般的なガイドラインとテクニック

UML モデルの内容を編成するために主に使用するツールは、パッケージです。UML のパッケージには、2 つの大きな目的があります。

- モデル情報の分割、編成、ラベル付け
    - 問題/解決領域内の特定のテーマに対応する要素のグループ化
    - インターフェース、実装、図などのタイプごとにモデル情報を分類
    - 要素同士の依存関係を定義して制御するために各要素をグループ化
    - 同じモデルに関する複数の代替ビューを示した図をグループ化
- 名前空間の確立
    - モデルの各要素のための名前空間
    - モデルの各要素から生成される実装成果物のための名前空間 (モデルと実装言語の名前空間との対応関係を含む場合あり)

[4] これらの図を作成するには、「File」→「New」→「UML Model」を使用してモデルを作成するのではなく、「File」→「New」→「Class Diagram」を使用して、コードの「ビュー」を UML（またはその他の）表記で構築できる図を作成してください。個々の図は拡張子 .dnx のファイルとして別々に保持され、コード・ファイルとほぼ同様のバージョン管理が可能です。これらの図には表記が含まれるだけで、セマンティクス情報はまったく含まれません。関連するセマンティクス情報は、コードそのものの中にあります。クラス名や操作のシグニチャーなど、これらの図にある要素を変更すると、実際には背後にあるコードそのものを変更していることになります。同じような変更を (テキスト・エディターを使用して) コードに加えると、変更したコードに対応する図が自動的に更新されます。

- 再利用の単位のための名前空間

従来 RUP では、各種モデル・タイプに応じたパッケージ戦略を提唱してきました。本書のモデル・タイプ固有のセクションは、そのような戦略を反映しています。RSx では、このほかにも編成用のツールをさらに用意しています。これについては、以下で説明します。

«perspective» パッケージを使用して、ビューポイントを表現する

各要素を複数の方法で編成することが望ましい場合は、代替の編成スキームを記述した図を含む、追加のパッケージを作成できます。モデルの内容に関して、モデルのパッケージ・スキームを超えたビューを記述しなければならない場合に、このテクニックはいつでも活用できます。RSx では、このテクニックをサポートするために、UML の「基本プロファイル」の一部として «perspective» パッケージ・ステレオタイプを用意しています。この «perspective» パッケージは、Systems Engineering または IEEE 1471- 2000 の「ビューポイント」にほぼ相当する RUP 機能と言えます。

«perspective» パッケージ内に、セマンティクス要素 (クラス、パッケージ、関連など) を置かないでください。代替の編成条件やアプリケーションのビューポイントに基づいたビューを記述した図だけを、置いてください。«perspective» ステレオタイプをパッケージに適用することによって、可能になることがあります。まず、特定のビューポイントを表現したものとして、そのパッケージをビジュアルで識別できます。また、«perspective» パッケージ内にセマンティクス要素が置かれている場合に警告を出す、モデル検証ルールをサポートできます。さらに、RSx の変換機能がバイパスするべきパッケージであることを示す、マーカーのような役割も果たします。

トピック・ダイアグラムを使用して、特定の条件に関する自己更新型の記述を作成する

記述したい要素を手作業で追加していく「通常の」図とは対照的に、トピック・ダイアグラムは、既存のモデルの内容に対して実行されるクエリーによって、内容を決定します。トピック・ダイアグラムを作成するには、「トピック」のモデル要素を選択してから、そのトピック要素との関係の種類に基づいて、図の中に登場させる他の要素を定義します。モデルのセマンティクスの内容が変われば、それに応じてトピック・ダイアグラムも変わります。

ブラウズ・ダイアグラムによってモデルを検討する

ブラウズ・ダイアグラムは、モデル編成用の *専用* ツールではありません。手作業で図を構成しなくても、モデル内容を発見し、理解できるようになることが目的になります。しかし、モデルの編成においても、ブラウズ・ダイアグラムを活用することで、永続的な図を構成する必要を少なくできる場合があります。これによって、モデルの全体的なサイズが小さくなり、複雑さも軽減されるため、編成作業がいっそう容易になります。

ブラウズ・ダイアグラムは、トピック・ダイアグラムと似ているところがありますが、主な違いは、ブラウズ・ダイアグラムは永続しないという点です。つまり、何かの処理中に常に生成されています。ブラウズ・ダイアグラムを生成するには、図または Model Explorer からモデル要素を選択して、コンテキスト・メニューの「Explore in Browse Diagram」を選択します。選択された要素を記述した図が「中心点」として生成され、その周りに関連要素が放射状に表示されます。もちろん、そのブラウズ・ダイアグラムに表示されている関連要素のいずれかを選択すると、その要素を別のブラウズ・ダイアグラムの中心点とすることができます。つまり、このような操作をいくらでも続けることができるわけです。

図から図へのナビゲーション

RSx には、図から図へのナビゲーションに関するメカニズムが 2 つ用意されています。

- Model Explorer から 1 つの図ノードをドラッグして、別の「ホスト」図にドロップできます。すると、ホスト図の上にアイコンが表示されます。そのアイコンをダブルクリックすれば、参照先の図が開きます。
- モデル内に新しい UML パッケージを作成すると、「Main」という名前の図 (自由形式) が自動的に作成されます。デフォルトでは、この「Main」図がパッケージの「デフォルト」図になります。その図の名前を「Main」以外に変更しても、その図は引き続き「デフォルト」として扱われます。さらに、パッケージ内の別の図を選択して、その図をパッケージの「デフォルト」図にすることもできます。パッケージ自体を別の「ホスト」図の上に置いた場合に、そのパッケージをダブルクリックすることによって、そのデフォルト図を開けるようにすることが、「デフォルト」図の目的です。

これらのメカニズムは、編成に関する以下の**ガイドライン**をサポートしています。このガイドラインは、どんなタイプのモデルにも適用されます。

1. 各モデリング・ファイルの「Main」図 (または他のデフォルト図) を作成して、以下のものを記述します。
    a. モデリング・ファイル内の各最上位パッケージ
    b. モデリング・ファイルのルート・パッケージ内に存在する、他のあらゆる図のアイコン (つまり、デフォルト図自体のアイコンは記述しません)
2. 各最上位パッケージの「Main」図 (または他のデフォルト図) を作成して、以下のものを記述します。
    a. その最上位パッケージに直接含まれているパッケージ
    b. その最上位パッケージに直接含まれている、他のあらゆる図のアイコン
3. 下位のパッケージに関して、この手順を繰り返していきます。

5. ユースケース・モデルの内部編成のためのガイドライン

ユースケース・モデルのおおまかな編成

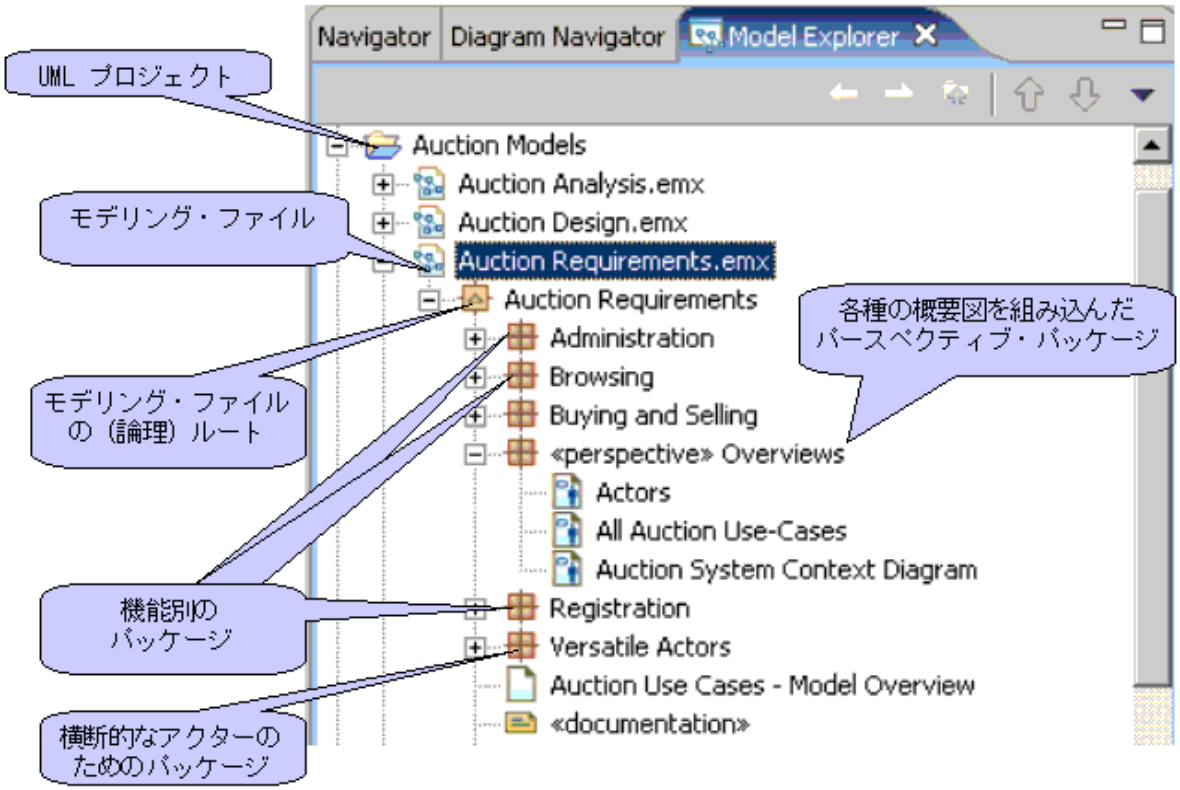


図 5-1

図 5-1 のユースケース・モデルの編成は、以下のガイドラインに基づいています。

1. 最上位のパッケージを使用して、機能別にグループ分けを行います。理由は以下のとおりです。

    - このグループ分けは、ユースケース・モデルの作業をチームで行うときの作業分担に、ほぼ相当します。また、トラブルの原因となるファイルの競合を避けるために、後からユースケース・モデルを複数のモデリング・ファイルに分割する場合にも容易に対応できます (最上位のパッケージごとに、別のモデリング・ファイルを作成するだけですみます)。

    - 他の編成方法に比べて、最終的な実装の編成内容との対応関係が明確になります。特に、変換機能を使用して、下位の抽象レベルを段階的にシードしていく場合は、この点が重要になります。たとえば、ユースケース・モデルに基づいて分析モデルの中にシード内容を生成する場合は、ユースケース・モデルのパッケージ構造が、対象の分析モデルの望ましいパッケージ構造と適切に対応するようにします。次に、分析モデルのパッケージ構造が設計モデルと、設計モデルのパッケージ構造が実装を構成するプロジェクト・セットと、それぞれ適切に対応するようにします。この対応関係がシンプルであれば

あるほど、1 つの抽象レベルから次の抽象レベルへの変換内容を構成する手間を省けるようになります。

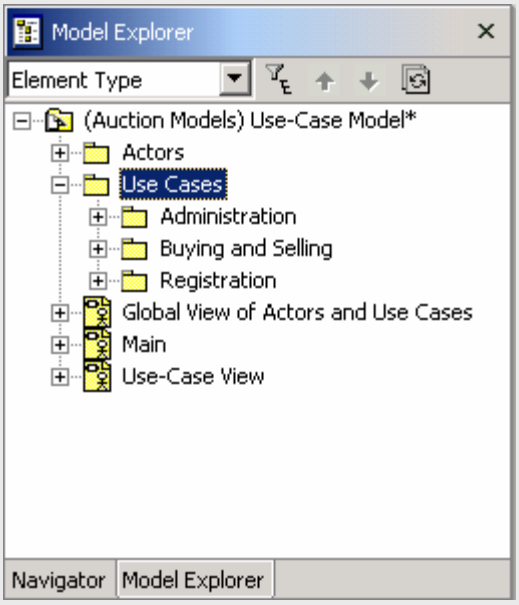
2. 別の最上位パッケージを使用して、用途の広い (汎用性の高い) アクターを取り込みます。

3. «perspective» パッケージの中の図を使用して、ユースケースの概略的な (横断的な) ビューを取り込みます。理由は以下のとおりです。

   - モデルのセマンティクス要素を機能別に編成する一方で、「アーキテクチャーの観点において重要な」ユースケースの概略を示す、横断的なビューを用意できます。

XDE/Rose

RSX のガイダンスでは、アクター用に 1 つのパッケージを作成し、ユースケース用にもう 1 つのパッケージを作成するという、ユースケース・モデルの上位レベルの編成に関する従来のガイダンスが少し改訂されています。そのため、必要なモデルのサイズや複雑さによっては、この XDE ベースの例で示されているような機能指向のグループ分けを実現するために、下位レベルのパッケージを使用する必要があります。



ユースケース・モデルの内容

ユースケースの良い書き方やユースケース・モデリングに関してすべきこと/すべきでないことを詳しく取り上げるのは、本書の意図から外れていますが、アクターとユースケース以外にユースケース・モデルの中に組み込むことができるものについて、ここで簡単に触れておきます。

- **推奨:** モデルのルートに「メイン」の図を作成して、モデルの他のパッケージを記述し、それらのパッケージとそれぞれの「メイン」の図に対するドリルダウンをサポートします。

- **推奨:** 各ユースケース・パッケージに、そのパッケージのユースケース、ユースケース同士の関係、およびユースケースにかかわるアクターを記述した図を組み込みます。(ユースケースの数が多ければ、複数の図を作成したほうがよい場合もあります。)

- **推奨:** 各ユースケースのメインのフローと代替のフローを「Documentation」フィールドに記述します[5] (**図 5-2** を参照)。

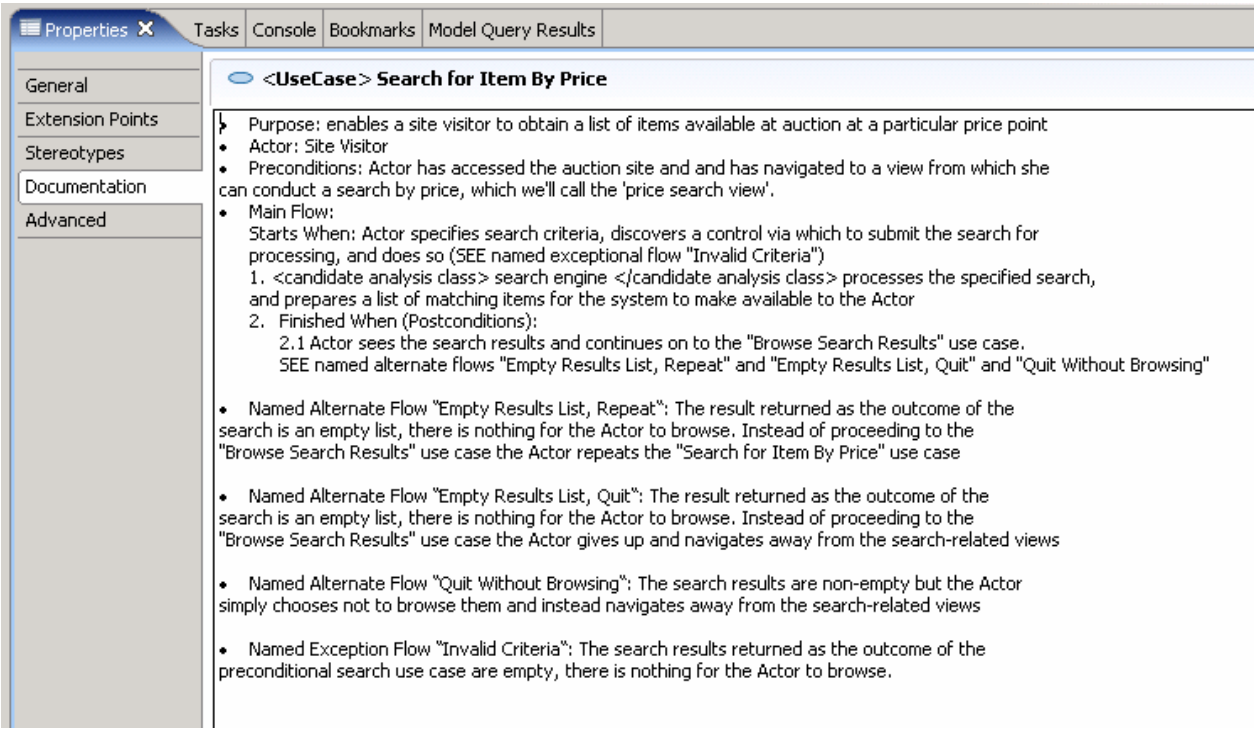


図 5-2

- **オプション:** ユースケースが複雑な場合は、必要に応じてアクティビティー図を追加し、ユースケースのアクティビティー・フロー全体を記述します。(**図 5-3** を参照してください。) **理由は以下のとおりです。**こうすることで、メインのフローや代替/例外のフローのそれぞれに対応する状態を示すことができ、最終的に各種のフローすべてをもう一度収束できるようになります。(RSx でアクティビティー図を追加すると、ユースケースにアクティビティーが自動的に追加され、そのアクティビティーの下に図が組み込まれます。)

- **オプション:**ユースケースに含まれる名前付きのメイン/代替/例外フローのそれぞれに関する「ブラック・ボックス」的な実現内容をモデル化し、ユースケースにコラボレーション事例を追加し、そのコラボレーション事例に、ユースケースのメイン・フローに対応する相互作用インスタンスと、名前付きの代替/例外フローのそれぞれに対応する相互作用インスタンスを追加し、それぞれの相互作用インスタンスのシーケンス図 (またはコミュニケーション図) を作成します。このユースケースのコラボレーション・インスタンスを、分析レベルのユースケース実現内容 (分析モデル内で記述されるもの) や、設計レベルのユースケース実現内容 (設計モデル内で

[5] ユースケース記述の例に示されているフォーマットを実現するには、RTF 対応のエディターでユースケース記述用のテキスト「テンプレート」を作成し、ユースケース記述フィールドにそのテンプレートをコピー・アンド・ペーストします。

記述されるもの) と混同しないでください。これらは、ユースケースの「ホワイト・ボックス」的な実現内容であって、ソリューションの内部要素間の相互作用を記述したものです。ここで示しているユースケース・モデルのコラボレーション事例は、アクターとシステム間のきわめて「ブラック・ボックス」的な相互作用です。(**図 5-3** を参照してください。) **理由は以下のとおりです。**このようにすれば、ユーザーとシステムとの間にどのような相互作用が生じるかについての概略を、非技術系の利害関係者のために示すことができます。また、実装に含めなければならない各種のビュー (画面やページ) を識別するためにも役立ちます。さらに、ユースケースの各種フロー (シナリオ) に名前を付けて、その名前をセマンティック・モデル要素 (つまりコラボレーション事例) に割り当てることによって、名前の体系を正式に確立することができます。

> ***XDE/Rose***
> UML 1.x では、「コラボレーション事例」の代わりに「コラボレーション・インスタンス」を使用していました。

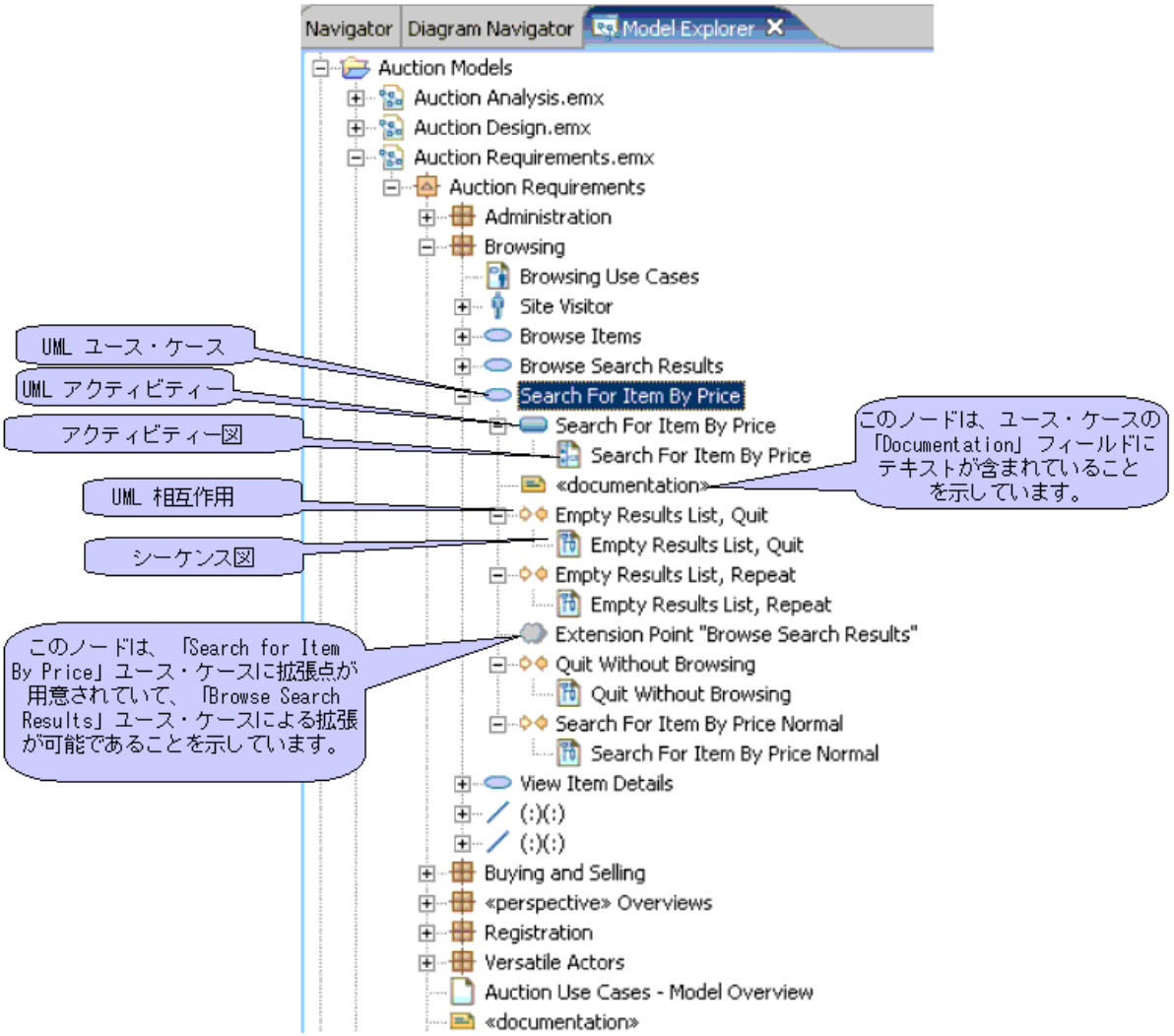


図 5-3

- **オプション:**「アーキテクチャーの観点で重要な」ビューを識別するために RUP のガイダンスを活用している場合、特にソフトウェア・アーキテクチャー説明書を活用している場合は、最上位の «perspective» パッケージを追加して、アーキテクチャーの観点で重要なユースケースを記述したユースケース図を組み込みます。そのパッケージには、「Use-Case View of Architecture」などの名前を付けます。

6. 分析モデルの内部編成のためのガイドライン

ソリューションを映画に例えれば、分析モデルは最初のシーンのようなものです。これは、要件から最終的な設計を生み出すための足がかりになります。主にビジネス領域に関する情報を収集し、ビジネスに近い上位の抽象レ

ベルにおけるソリューション要素の候補を提示します。分析モデルには、分析のための各クラスと、分析レベルのユースケース実現内容を組み込みます。ソリューションに必要なクラスを洗い出す作業は、ユースケース実現内容をモデリングするプロセスから始めます。このプロセスには、主にシーケンス図を使用します。特に、シーケンス図で必要なライフラインを見つけ出し、そのライフラインに相当するクラスを組み込んでいきます。さらに、ユースケース・モデルの内容に基づいて分析モデルの内容を考えるときに適用できる経験則が、いくつかあります。これについては、このセクションの後のほうで触れることにします。

RUP で分析モデルを設計モデルから切り離して保持するべきかどうかは、プロジェクトごとに決定する事柄です。分析モデルを別個に管理すればそれなりの時間がかかるので、その時間に見合うだけの価値があるかどうかを検討しなければなりません。別個の分析モデルを作成しても、その分析モデルを保持しない場合は、分析クラスを設計モデルに移して加工することになります。あるいは、分析モデルを段階的に発展させて、設計モデルを作り出すという方法もあります[6]。製品固有の観点から検討できるオプションを、いくつか紹介します。

1. 分析モデル・テンプレートに基づいて分析モデルを作成し、それを単一のモデリング・ファイルまたはモデリング・ファイルのセットに組み込みます。次に、手作業か自動変換機能によって、エンタープライズ IT 設計モデル・テンプレートに基づき、別のモデル・ファイルまたは別のモデル・ファイルのセット内に、洗練された分析要素を作成してから、分析モデリング・ファイルの扱いを決めます。この場合は、別個の分析モデルを保持するか破棄するかを選択できます。
2. エンタープライズ IT 設計モデル・テンプレートに基づいて、単一のモデリング・ファイルまたはモデリング・ファイルのセット内で、分析レベルのモデリングを行います。この際、分析プロファイルを適用します。この作業では、分析クラスを使用してユースケース実現内容のモデリングを開始し、設計インターフェースにさまざまな動作の役割を引き継がせるために、時間をかけて実現内容を洗練させることができます。
3. 1 番目と 2 番目のオプションを組み合わせて、ある種の分析モデルを設計モデルと同じモデリング・ファイル内に保持するという方法もあります。そのためには、分析の内容を «analysis» というキーワードの付いたパッケージに分離します。こうすることで、より洗練された設計レベルの成果物と同じモデリング・ファイル内に、分析レベルの成果物を保持できます。

RSx の変換機能を使用して実装を生成する場合に覚えておくべき点は、分析レベルの要素を変換機能の入力データとして使用できる場合が多いということです。この場合、分析レベルの要素を設計の要素に洗練するための手作業の一部が不要になります。このような方法で RSx を使用する場合は、上記のオプション 2 または 3 の使用をお勧めします。RSx の一部として組み込まれている標準のコード生成変換機能は、«analysis» キーワードの付いたモデル・パッケージをバイパスするからです。

[6] 実際には、RUP は分析クラスと分析レベルのユースケース実現内容を設計モデル内に作成するオプションを呼び出した後、直接それを設計の形式に発展させます。. この方法で設計モデルが「発見」された場合、「純粋な分析」の観点をいくらか残す方法でパッケージを作成することができます。

分析モデルのおおまかな編成

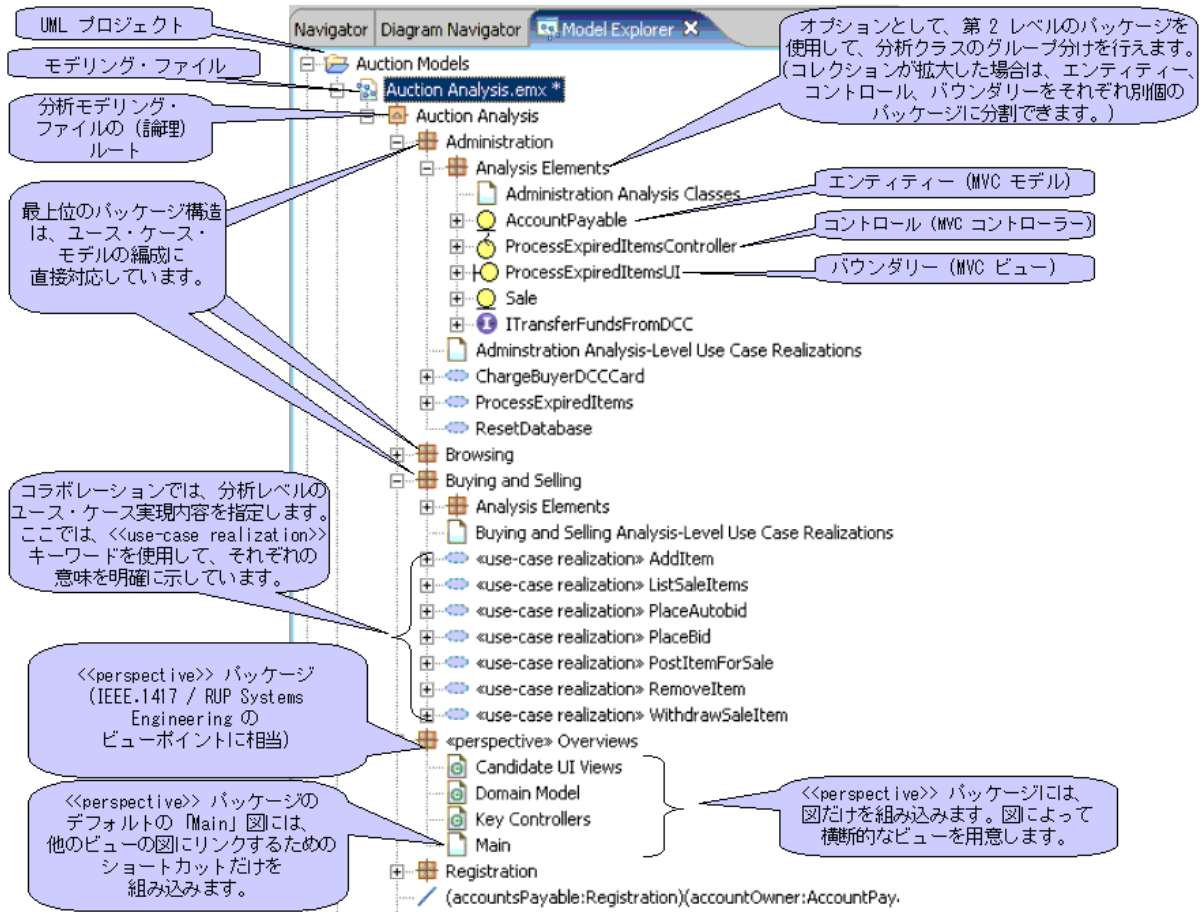


図 6-1

図 6-1 の分析モデルの編成は、以下のガイドラインに基づいています。

1. 最上位のパッケージを使用して、分析クラスを機能別にグループ分けします。理由は以下のとおりです。ユースケース・モデルの場合と同じ理由です。

2. オプションとして、最上位パッケージの中でサブパッケージを使用して、分析クラスを集めて編成することができます。

3. «perspective» パッケージ内の図を使用して、分析の要素の概略的な (横断的な) 代替ビューを取り込みます。理由は以下のとおりです。これによって、モデルのセマンティクス要素を機能別のグループに編成する一方で、さまざまな利害関係者のさまざまな観点を提供することができます。

この方法に少し手を加え、最上位のパッケージを使用して、分析クラスからユースケース実現内容を分離したのが、**図 6-2** です。ここでは、その最上位パッケージの中に、すべての最上位パッケージに対応する機能別のサブパッケージ群を組み込んでいます。このようにユースケース実現内容を分離することによって、分析クラスを含んだパッケージ構造をリファクタリングできます。この際、ユースケース実現内容の編成に*必ずしも* 影響が及ぶわけではありません。(特に、分析モデルを*そのまま* 発展させて設計モデルを作り上げる場合は、クラスのパッケージ編成がユースケースの元の編成とは食い違うような形で発展していく可能性があります。)

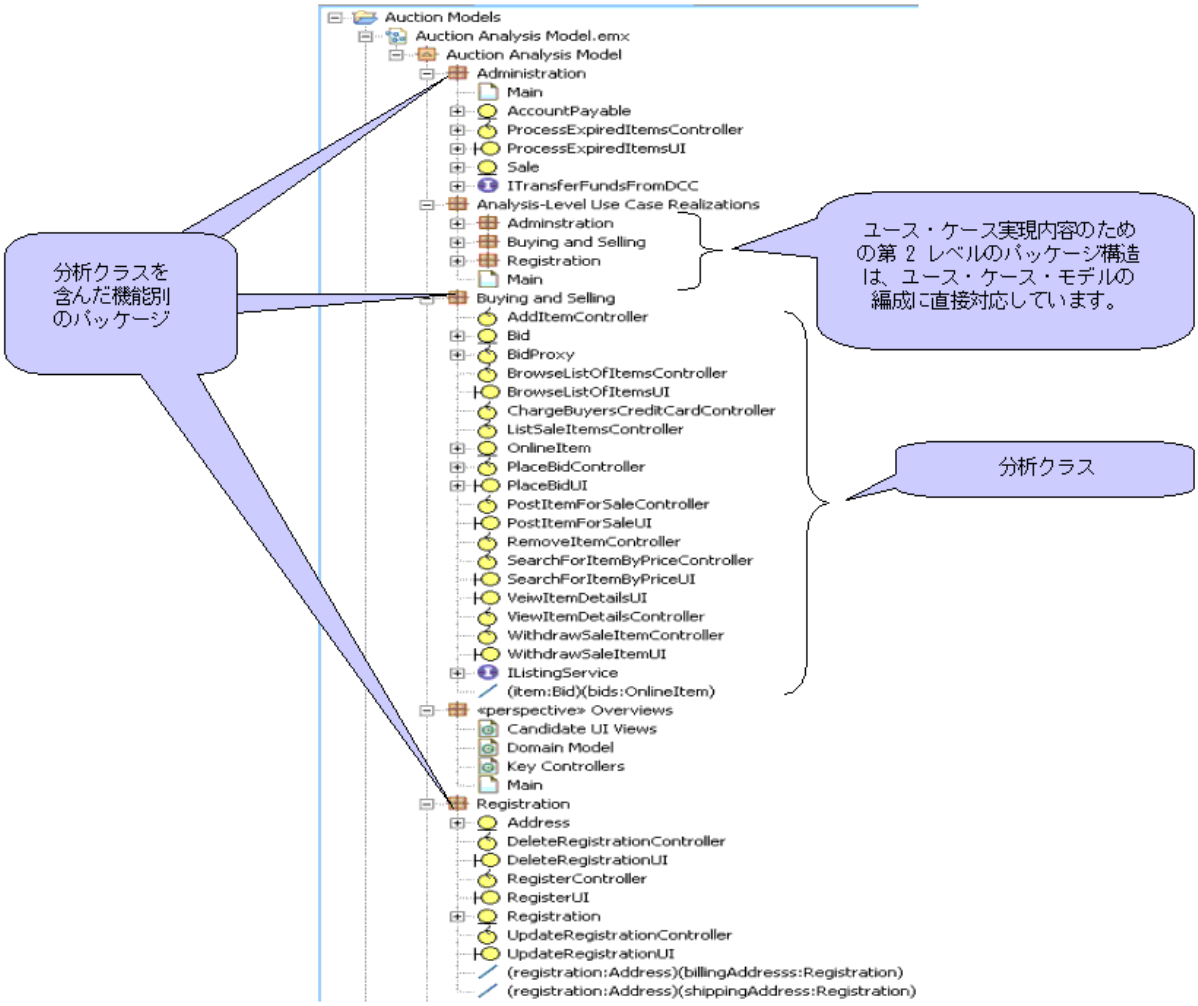


図 6-2

別の企業 (パートナー企業) 内のグループなども含めて、さまざまなグループが作業している状況では、それぞれのグループが作成するモデルの内容をマージしたり再利用したりする場面を最初から想定し、それに合わせた名前付けの体系を考えておくほうがよい場合もあります。その場合は、**図 6-3** のような、インターネット・ドメインの名前空間を反転させた名前付けの体系を使用できます。このこと*自体* が分析モデリングの大きな問題になることはあまりありませんが、分析モデルを*そのまま* 設計モデルに発展させる場合に、設計レベルでの再利用やビジネス統合を想定しているのであれば、あらかじめ計画を立てておくことが望ましいと言え

ます。さらに、分析/設計から生成されるコードの編成との対応関係が適切になるため、コード生成変換機能を構成するときに、その作業を簡略化できるというメリットもあります。

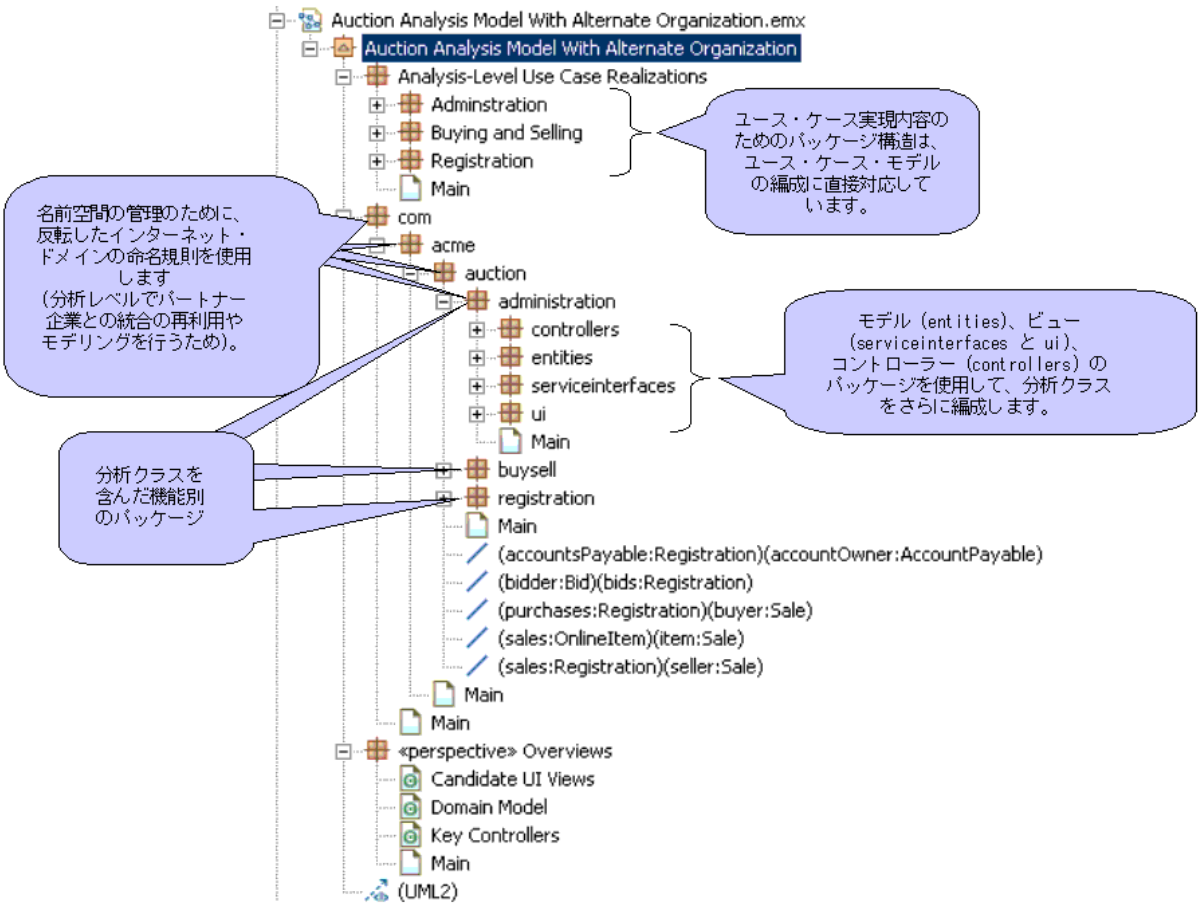


図 6-3

分析モデルの内容

分析クラスを見つけ出す方法は、いくつかあります。1 つは、ユースケース実現内容を示すシーケンス図を描くという方法です。そのシーケンス図から必要なライフラインを見つけ出せば、基本的にはそのライフラインが分析クラスの候補になります。このようにしてクラスを見つけ出した場合は、分析モデルのユースケース実現内容パッケージの中にそれらのクラスを作成できますが、それをそのままにしておくべきではありません。モデルを「リファクタリング」して、分析モデルのおおまかな編成に関するガイドラインのセクションで説明した機能別のパッケージに、分析クラスを移すのが望ましいと言えます (**図 6-1** を参照してください)。

分析クラスを見つけ出すためのもう 1 つの便利な方法は、以下のような経験則に基づいて、分析モデルにクラスを「シード」することです。

- ユースケース・モデルのユースケースごとに、それぞれ対応する «control» クラスを分析モデルに追加します。«control» クラスは、ユースケースに関連するビジネス・ロジックに相当します。(後の設計段階では、セッション管理などの問題点にも対応することになります。)
- ユースケース・モデルのアクター/ユースケースの関係ごとに、それぞれ対応する «boundary» クラスを分析モデルに追加します。«boundary» クラスは、ソリューションと人間のアクターとのインターフェース、またはソリューションと外部のシステムとのインターフェースに相当します。人間のアクターに対応する «boundary» クラスは、最終的に設計や実装の段階における、複数のユーザー・インターフェースの成果物に対応することになります。外部のシステムに対応する «boundary» クラスは、最終的に設計や実装における、ある種のアダプター層に対応することになります。
- CRC カード分析やユースケース記述のワード分析などのプロセスによって、その他の «control» クラス (動詞) と «entity» クラス (名詞) を識別します。

このシード方法によって分析クラスを識別する場合、分析モデルのおおまかな編成に関するガイドラインのセクションですでに述べた機能別のパッケージに、分析クラスを直接配置することができます (**図 6-1** を参照してください)。

どんな方法で分析クラスを見つけ出すとしても、ほとんどの場合は、元の機能別のパッケージ編成を変更する必要があります。

オプション: 分析クラス・パッケージ内の第 2 レベルのパッケージを使用して、パッケージの内容をさらに編成します (**図 6-4** を参照してください)。

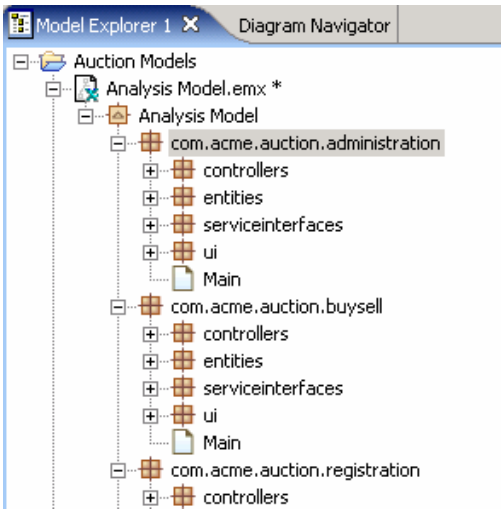


図 6-4

推奨: 分析モデルには、分析クラスの観点からユースケースの実行方法を記述した、分析レベルのユースケース実現内容を組み込むようにしてください。分析レベルのユースケース実現内容 (UML のコラボレーションで表現されるもの) は、それぞれユースケース・モデル内の単一のユースケースを実現したものであり、そのユースケースと同じ名前になります。**図 6-5** を参照してください。分析レベルの実現内容としてモデル化するべき、名前付きのユースケース・フロー[7]ごとに、それぞれ対応するシーケンス図を追加します (これによって、所有側の相互作用が自動的に追加されます)。シーケンス図を作成するときにモデルに追加されるセマンティクス内容のタイプについ

[7] 以前、ユースケース・モデル内に作成したもの。

ては、**図 6-6** を参照してください。(Model Explorer ビューでは、UML 要素のタイプに関するフィルターを設定して表示内容を絞り込み、**図 6-6** にあるような雑然とした表示内容を整理することができます。)

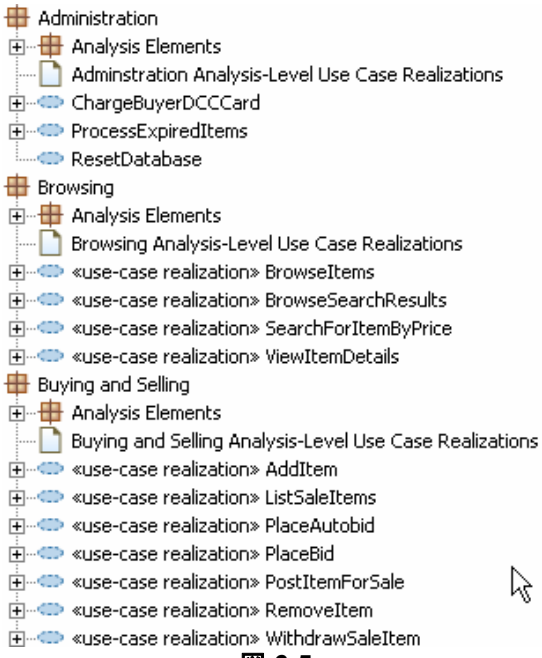


図 6-5

オプション: ユースケース・フローのシーケンス図を作成したら、Model Explorer で所有側の UML 相互作用を選択し、そこにコミュニケーション図を追加できます。その新しいコミュニケーション図には、シーケンス図にかかわっていた分析クラスのインスタンスが自動的に組み込まれます。

推奨: 各ユースケースの実現内容 (UML コラボレーション) から実現内容の依存関係を作成し、それに対応するユースケースをユースケース・モデルから作成します (**図 6-6** を参照してください)。トピック・ダイアグラムや追跡可能性分析などの機能を使用すれば、モデル内の追跡可能性関係を把握できます。したがって、追跡可能性関係を記述するための永久的な図を保持する必要はありません。この関係を作成するには、「使い捨て」の図を使用することをお勧めします。たとえば、以下のような方法があります。

- 自由形式の図を、コラボレーションに追加します。
- その上にコラボレーションをドラッグします。
- その上にユースケースをドラッグします。
- 依存関係を記述します。
- 最後に Model Explorer 内で、その図をコラボレーションから削除します。

推奨: ユースケース実現内容ごとに、対応する「参加者」図を組み込んで、実現内容に加わっている分析クラス (つまり、そのユースケースの実現内容を記述した相互作用図にインスタンスが登場する分析クラス) と、相互作用図に記述されているコラボレーションをサポートするクラス間の関係を示します。**図 6-6** を参照してください。

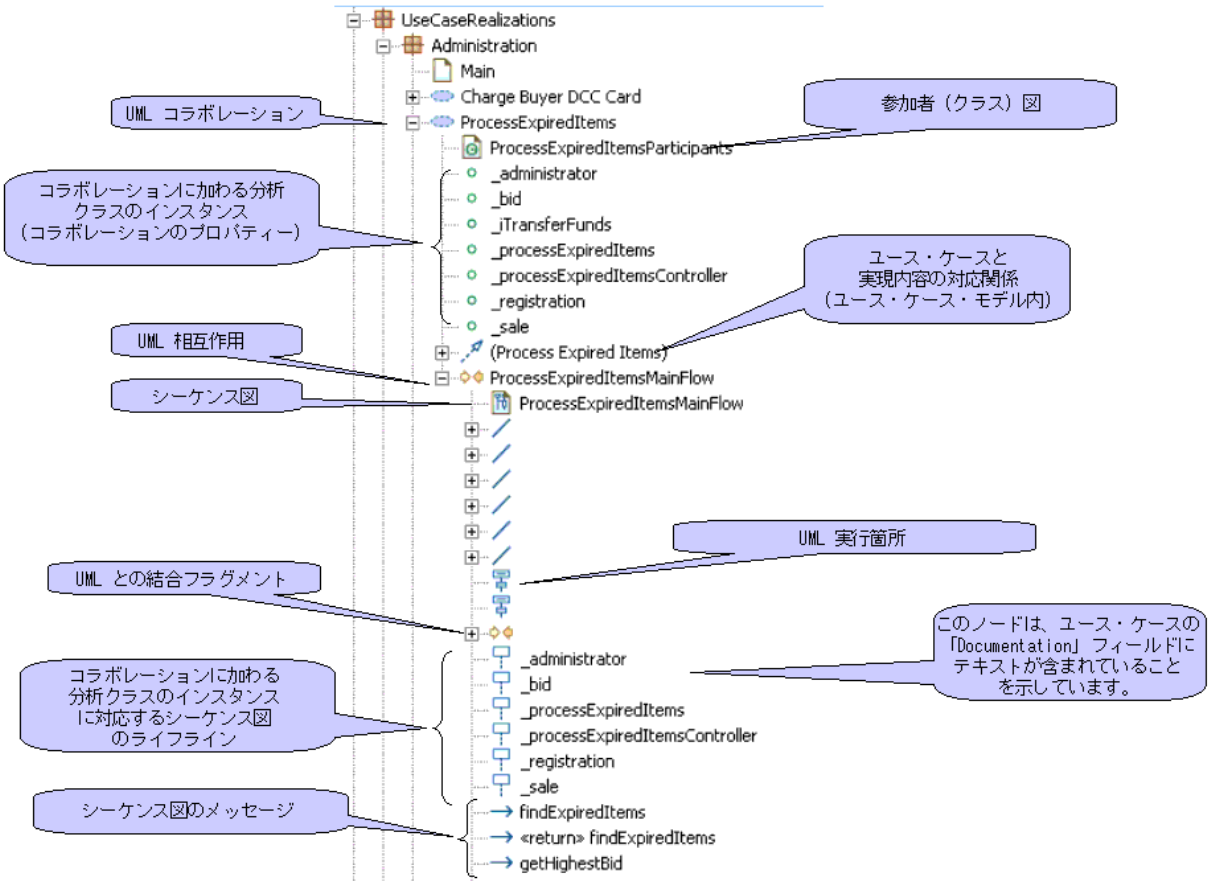
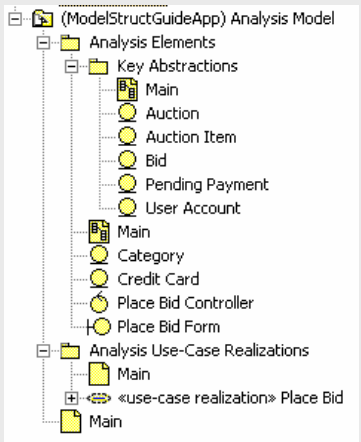


図 6-6

XDE/Rose

これまで推奨されていた**分析モデル**の構造 (下記参照) が RSx では修正され、分析クラスを機能指向のパッケージ編成にすることが強調されるようになりました。また、Key Abstractions パッケージを使用せずに (機能指向のパッケージ方法と矛盾してしまうため)、代わりに «perspective» パッケージ内で Key Abstractions 図 (単一または複数) を使用することになりました。



7. 設計モデルの内部編成のためのガイドライン

設計モデルのおおまかな編成

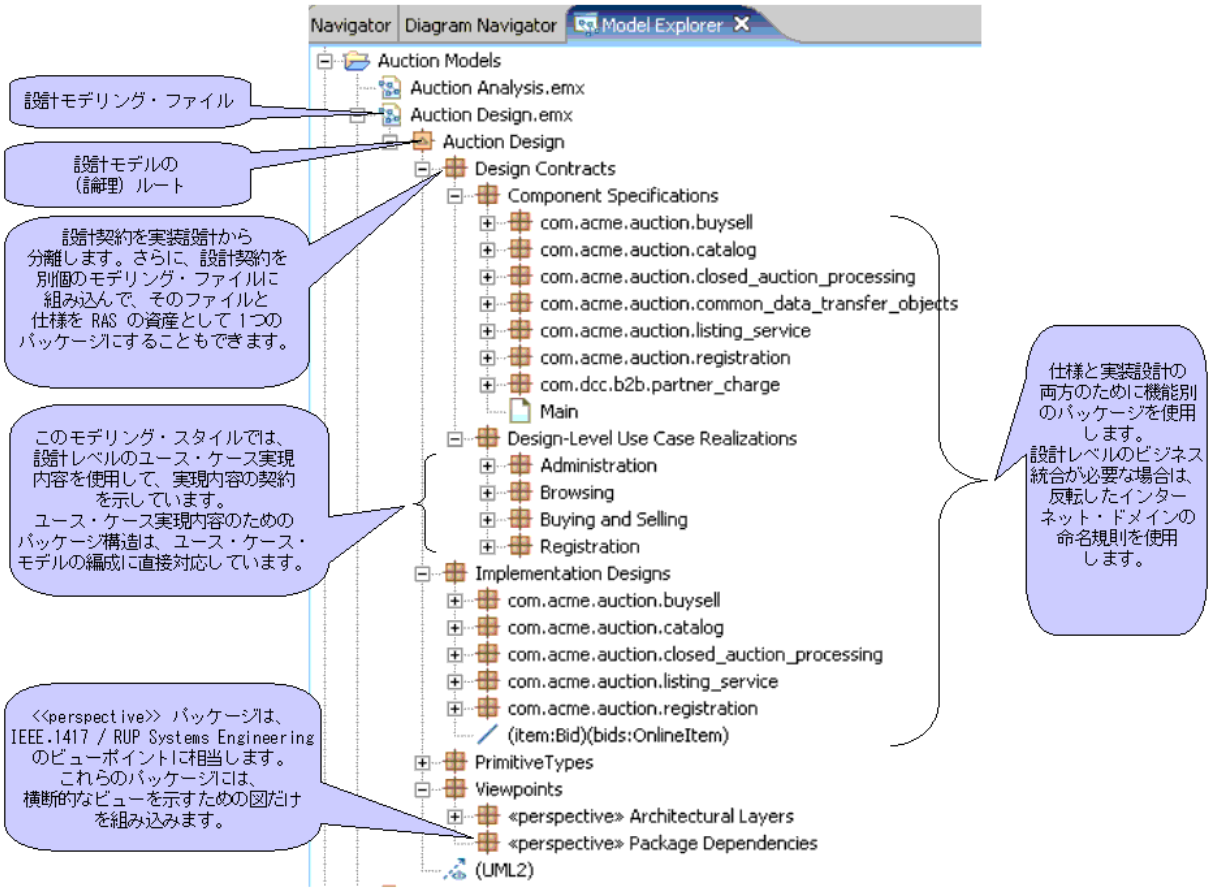


図 7-1

図 7-1 の設計モデルの編成は、以下のガイドラインに基づいています。

1. 仕様と実装設計を分離します。この図では、最上位の「Design Contracts」パッケージと「Implementation Designs」パッケージによって、これを実現しています。

2. 下位レベルのパッケージを使用して、機能別のグループを設定します。たとえば、最初は分析段階で使用した編成から始め、分析クラスを実際の設計クラスやコンポーネント、サービスなどに対応させる方法を決めながら、その編成を発展させていくという方法もあります。(初期の編成スキームは、いずれにしても設計段階で発展していくことが多くなります。これについては、以下の説明を参照してください。)

   ここで、サブシステムについて簡単に説明します。バージョン 2 より前の UML では、サブシステムは特別なタイプのパッケージでした。UML 2 の場合、サブシステムは特別なタイプのコンポーネントであり、そのコンポ

ーネントにパッケージを組み込むことができました。UML 2 では、パッケージの代わりになる有効な編成手段/名前空間として «subsystem» コンポーネントを使用できます。ただし、サブシステムとパッケージを使い分ける方法については、具体的なことは定めてられていません。たとえば、エンタープライズ・レベルのアーキテクチャーでは、アプリケーションの設計サブシステム程度の細分性レベルでパッケージを使用し、アプリケーション全体 (CRM や SCM など) に対応するレベルでサブシステムを使用する、という方法があります。

> ***XDE/Rose***
> この資料の作成時点では、Rose および XDE のモデル・インポート・ツールによって、UML 1.x のサブシステムを UML2 のサブシステムか «subsystem» キーワードが適用されたパッケージにマッピングするオプションを提供する予定となっています。

3. 設計要素の編成は、システムのユースケースの編成 (ユースケース・モデル内での編成や、別個の分析モデルを保持する場合は分析モデル内での編成) とは別個に発展する場合が多いと言えます。パッケージを使用し、設計の契約内容をさらに設計要素仕様 (使用法の契約の場合) や設計レベルのユースケース実現内容 (実現内容の契約の場合) に分割して、ユースケース実現内容のパッケージ下部構造とユースケースそのものの編成との対応関係を維持します。

4. 機能領域の仕様と実装設計を構成する要素の、第 2 レベルの編成スキームの基礎として、アーキテクチャー層を使用することを*検討します* (詳細については、以下の説明を参照してください)。

5. セマンティクス・モデル要素をグループ分けした UML のコンポーネントとパッケージの中に、各グループに固有のビューを示した図を置きます。このガイドラインは、そのグループ分けがビジネス領域の機能別のサブセットに基づいているか、アーキテクチャー層に基づいているか、その他のものに基づいているか、という点にかかわっています。そのパッケージまたはコンポーネント自身と同じ名前の「デフォルト」図を作成し、パッケージのおおまかな内容を示すように構成してください。そのようにしていくつかの図の名前と内容を関連付けておけば、モデルのナビゲートが容易になり、内容を把握するのも楽になります。

6. 反転したインターネット・ドメインの名前空間を、設計モデルで使用することもできます。理由は以下のとおりです。
   - 基本的に、言語固有の実装の場合と同じ理由です。
     a. 複数のモデル駆動型アプリケーションの統合作業がかかわっているシナリオ (特にパートナー企業がかかわっている場合)
     b. 再利用のシナリオ
   - 多くの場合、実装への変換を構成する作業 (場所や名前に関するソースと宛先の対応関係の設定) がシンプルになります。

7. 名前空間の対応関係を設定する際の手間や混乱を避けるために、対象の実装プラットフォームで有効なパッケージ名を使用することを*検討してください*。(簡単に言えば、「下線以外の句読点やスペースを、名前に使わない」ということです。)

8. パッケージ名には小文字を使用して、パッケージ内のクラス名と区別します。

9. インターフェースとそれを実現するコンポーネントやクラスには、別々の名前を使用することを *検討してください*。たとえば、インターフェースと実装の名前に ILoan と Loan を使用するか、または Loan と LoanImpl を使用する、といった具合です。モデルの中でもそのようにする必要があるわけではありませんが、基本的に

コード生成のためには便利な方法です。この方法によっても、変換機能の構成作業の手間をいくらか省くことができます。

10. 以下のようなシナリオでは、コード生成に使用しない分析レベルの内容を «analysis» ステレオタイプのパッケージ内に分離するようにしてください[8]。
    - A) 別個の分析モデルの使用を省略して、設計モデルに分析レベルの内容を取り込み、その内容を抽象化の分析レベルに保持すると同時に、同じモデル内に設計レベルの内容を作成する必要がある場合。
    - B) さらに、モデルからコードへの変換機能を EIT 設計モデルから実行することになる場合。

11. «perspective» パッケージ内の図を使用して、設計要素の概略的な (横断的な) ビューを取り込みます。理由は以下のとおりです。モデルのセマンティクス要素を機能別に編成する一方で、「アーキテクチャーの観点から重要な」内容や様々なタイプの利害関係者を納得させる観点を示す、横断的なビューを用意できます。

設計モデルのパッケージ構造は、時間の経過とともに発展していくものであると認識することが大切です。最終的な編成は、アーキテクチャーを構造化してコンポーネントとサービスにする方法と対応させる必要があります。この方法で設計の「*最終段階*」へと発展させていけば、通常は再利用可能な資産をパッケージ化できる可能性が高くなります。また、設計と、その設計から生成される実装成果物 (コード、メタデータ、文書) を入れる一式のプロジェクトやフォルダーとを、最も直接的に対応させることができます。

ただし、ユースケース・モデルの作成に使用して分析中に修正された編成方法に、「初期段階」の編成をある程度対応させるようにしてください[9]。『分析モデルの内部編成のためのガイドライン』のセクションで説明したように、分析モデルを発展させて所定の設計にしていく方法を、実際に選ぶことができます。言い換えると、設計の初期編成では、凝集したビジネス事項と疎結合のビジネス事項がグループ化され、そこから横断的な要素または再利用可能な要素が分離される傾向があります。この方法による初期編成の作成は、以下の理由から効率的であるとされています。

- 分析モデルまたはユースケース・モデルの内容から、設計モデルの内容を生成する変換機能を使用する場合、ソース・パッケージから目的パッケージへのマッピングが単純で分かりやすくなります。
- 機能の凝集とパッケージの疎結合に基づいて初期編成を作成する方法では、最終的なコンポーネント指向の編成にマッピングできる可能性が明らかに高くなります。つまり、設計プロセスの一部として必要になる、リファクタリングの量が減ることになるわけです。
- パッケージを疎結合にすることで、チームのワークフローが改善される可能性が高くなります。また、その設計を複数のモデリング・ファイルに織り込む場合に、再利用が容易になる可能性も高くなります。

もちろん、他の方法を使用することもできます。「最終段階」の編成として、その方法が望ましい場合もあります。

- J2EE ベースの Web アプリケーション (EJB を含む) が目標である場合は、RSA および Rational Application Developer の J2EE プロジェクトに関する規則に従う必要があります。[10]特に、アーキテクチャー層 (プレゼンテーション層とビジネス層。ビジネス層は、セッションおよびドメインという副層を持つ) に

[8] そのようなパッケージは、変換機能では省略されます。

[9] 分析クラスのパッケージは、発見されるたびに大幅にリファクタリングされるのが普通です。これは、再利用と予期されなかった機能要件のサポートを改善するためです。

[10] 大まかに言うと、システム、アプリケーション、または大規模サブシステムごとに 1 つのエンタープライズ・プロジェクトを置きます。各エンタープライズ・プロジェクトに対して、プレゼンテーション層用の 1 つの Web プロジェクトを置きます。そして、コンポーネントや小さなサブシステムに全般的に対応する EJB プロジェクトや、コンポーネントまたはサブシステムごとのセッション層 (セッション EJB) とドメイン層 (エンティティー EJB) に使用される EJB プロジェクト (通常は個別に設定) などの、複数の EJB プロジェクトを置きます。詳細については、このホワイト・ペーパーのセクション 9 を参照してください。

対応する、最上位レベルの設計パッケージを定義する必要があります。これは明らかにプラットフォームに中立な方法ではないので、設計しているソリューションを J2EE 以外のプラットフォームに実装しないことが分かっている場合にのみ、採用してください。

- より一般的な例として、開発者の専門的な判断により n 層のアプリケーションを構築していて、作業の分担がプレゼンテーション層とビジネス層に対応しているというケースが考えられます。この場合、それらのアーキテクチャー層に対応する最上位レベルのパッケージを使用することができます。しかし、特定の「アーキテクチャー」をサポートするために、特定の「ビジネス機能」をサポートすることを意図したクラスを編成する場合は、注意が必要です。どちらも変更が困難だからです。
- コンポーネント指向、サービス指向、サブシステム指向のいずれでもない編成方法を使用する理由が十分にある場合でも、コード生成の変換機能を構成する際に余分な作業が発生することになりますが、設計の編成を対象のプロジェクトおよびフォルダーのセットにマッピングすることが可能です。特に複雑なマッピングを定義するためには、「マッピング・モデル」と呼ばれる特別なタイプのコンパニオン・モデルを使用できます。

設計モデルの内容

設計モデルに何を含めるべきかに関しては、厳格な規則はありませんが、次の提案が役立つでしょう。

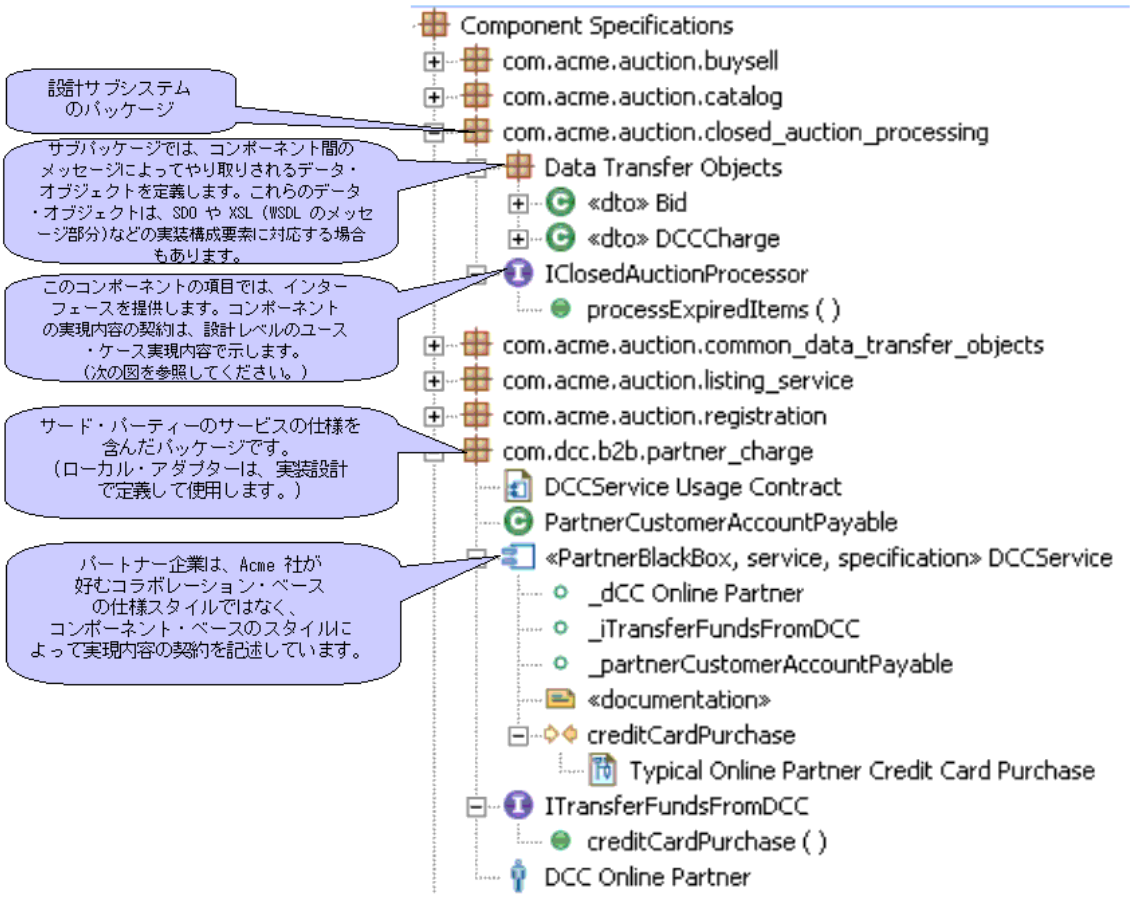

設計サブシステム
のパッケージ
サブパッケージでは、コンポーネント間の
メッセージによってやり取りされるデータ・
オブジェクトを定義します。これらのデータ
・オブジェクトは、SDO や XSL（WSDL のメッセ
ージ部分）などの実装構成要素に対応する場合
もあります。
このコンポーネントの項目では、インター
フェースを提供します。コンポーネント
の実現内容の契約は、設計レベルのユース
・ケース実現内容で示します。
（次の図を参照してください。）
サード・パーティーのサービスの仕様を
含んだパッケージです。
（ローカル・アダプターは、実装設計
で定義して使用します。）
パートナー企業は、Acme 社が
好むコラボレーション・ベース
の仕様スタイルではなく、
コンポーネント・ベースのスタイルに
よって実現内容の契約を記述しています。
Component Specifications
com.acme.auction.buysell
com.acme.auction.catalog
com.acme.auction.closed_auction_processing
Data Transfer Objects
«dto» Bid
«dto» DCCCharge
IClosedAuctionProcessor
processExpiredItems ( )
com.acme.auction.common_data_transfer_objects
com.acme.auction.listing_service
com.acme.auction.registration
com.dcc.b2b.partner_charge
DCCService Usage Contract
PartnerCustomerAccountPayable
«PartnerBlackBox, service, specification» DCCService
_dCC Online Partner
_iTransferFundsFromDCC
_partnerCustomerAccountPayable
«documentation»
creditCardPurchase
Typical Online Partner Credit Card Purchase
ITransferFundsFromDCC
creditCardPurchase ( )
DCC Online Partner

図 7-2

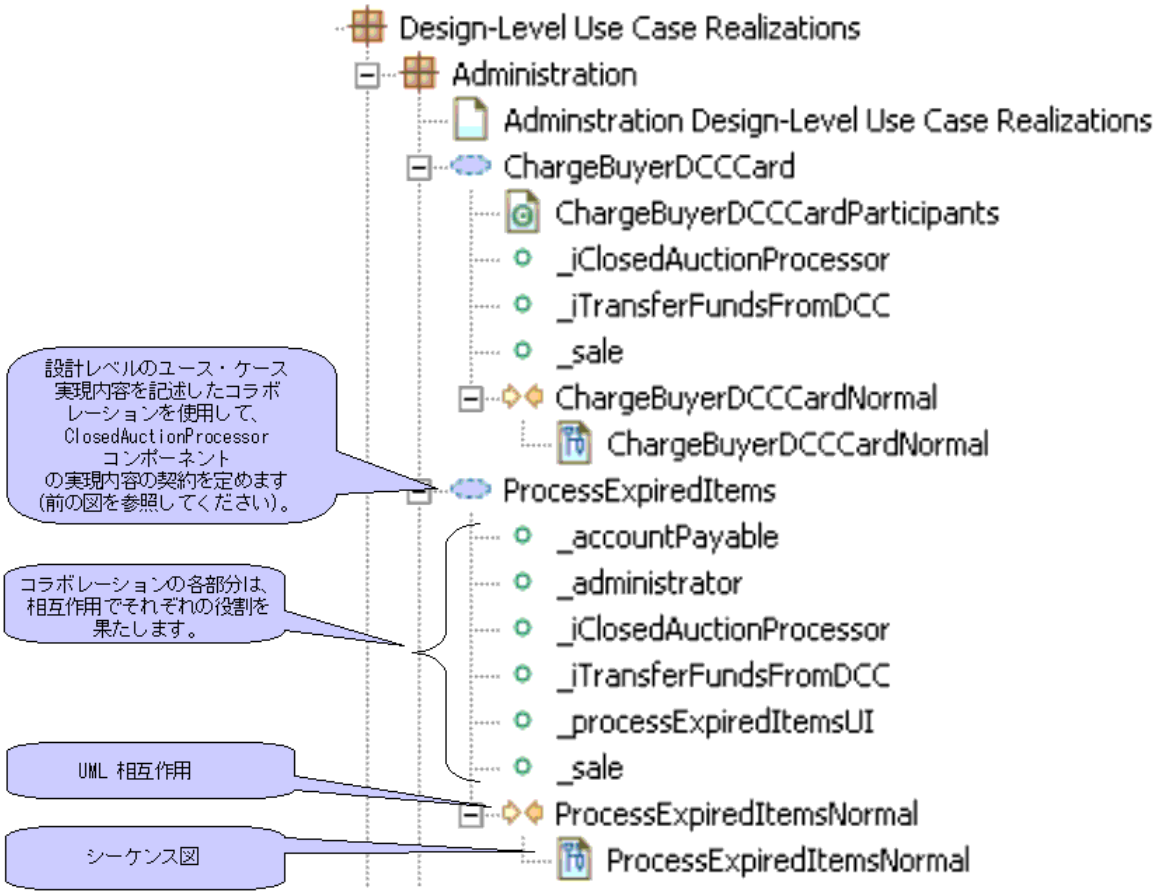


図 7-3

図 7-2 と **図 7-3** は、**図 7-1** に示されている編成の構造に従っており、設計の契約をどのように規定するかを示しています。

- 「ClosedAuctionProcessor」コンポーネントの使用法の契約が、単一のインターフェースとして表現されています[11] (**図 7-2**)。それに対応する実現内容の契約は、コラボレーションとして表現された、単一の設計レベルのユースケース実現内容により規定されています[12] (**図 7-3**)。分析レベルのユースケースの実現内容は分析クラス間のコラボレーションを示しているのに対して、設計レベルの実現内容は抽象化レベルの低い設計要素間のコラボレーションを示していることに注意してください[13]。設計モデルの仕様サブセットを実装設計サブセットとは別にパッケージ化したい場合は、設計レベルのユースケースの実現内

[11] この例ではインターフェースが 1 つですが、コンポーネントに複数のインターフェースを提供することも当然できます。

[12] 他のコンポーネントは複数のシステム・ユースケースに参加している場合があるため、それらのコンポーネントの実現内容の契約が複数のユースケース実現内容の中に含まれることがあります。そのような場合には、コンポーネントのインターフェースと同じパッケージ内に「{コンポーネント名} 使用場所」という図を組み込んで、それらのユースケースの実現内容を構成するさまざまな図へのリンクを、その図に配置することができます。

[13] それに似たもう 1 つの違いとして、設計レベルの実現内容に含まれる「参加者」図の一部は、分析レベルのユースケースの実現内容について提案される参加者クラス図の代わりに (または、これに加えて)、コンポーネントの配線を示すコンポーネント図になる場合があります。

容において、分析仕様要素または設計仕様要素だけを役割として使用し、実装設計要素は使用しないようにすることが大切です。

- サード・パーティーの「DCCService」に対する使用法および実現内容の契約は、共に 1 つのパッケージに入っています[14]。このケースでも使用法の契約が単一のインターフェースで構成されていますが、実現内容の契約は «specification» コンポーネントを使用して表現されています (**図 7-2**)。(それ以外の点では、実現内容の契約はほとんど同じように規定されており、動作 (この場合、「creditCardPurchase」という相互作用) を使用して表現されています。)コラボレーションの代わりにコンポーネントを使用するもう 1 つの例を、**図 7-4** に示します。

- 操作はインターフェース内で定義し、«specification» コンポーネント (使用されている場合) によって実現するか、そのインターフェースを実装している実装設計内の分類子によって実現することができます。

- データ転送オブジェクト (提供されている操作のパラメーター・タイプの役割を果たし、XML スキーマや SDO などの実装構成要素にマッピングできる) の仕様を、使用法の契約に含めることもできます。分散可能として設計していないコンポーネントについては、操作パラメーターとして使用するタイプの仕様として、データ転送オブジェクトを指定しても指定しなくてもかまいません。分散可能なサービス (たとえば、Web サービス) については、サービスの操作がローカル・アドレス・スペース内のオブジェクトを参照してはならないため、DTO を使用する必要があります。

> ***XDE/Rose***
> 以前のバージョンの UML では、ユースケースの実現内容のガイダンスとして、ユースケースごとにコラボレーション・インスタンスを使用することや、実現内容の重要なフローごとに相互作用およびシーケンス図を使用することが、定められていました。
>
> RSx では、相互作用およびシーケンス図を 1 つしか使用できないことが多くなっています。UML2 シーケンス図で、代替実行経路の表記がサポートされるようになったからです。
>
> また、UML2 では「コラボレーション・インスタンス」がなくなりました。代わりに、「コラボレーション・ユース」(タイプとしてコラボレーションが必要) が導入されました。したがって RSx では、ユースケースの実現内容を表現するために、コラボレーションを使用してください。

[14] ここでは、DCC 社が Acme 社に UML 仕様を提供し、Acme 社はその仕様を設計モデルに組み込んだと仮定しています。これは、インターネット・ドメインの名前空間を反転させて使用することが役に立つタイプのシナリオです。

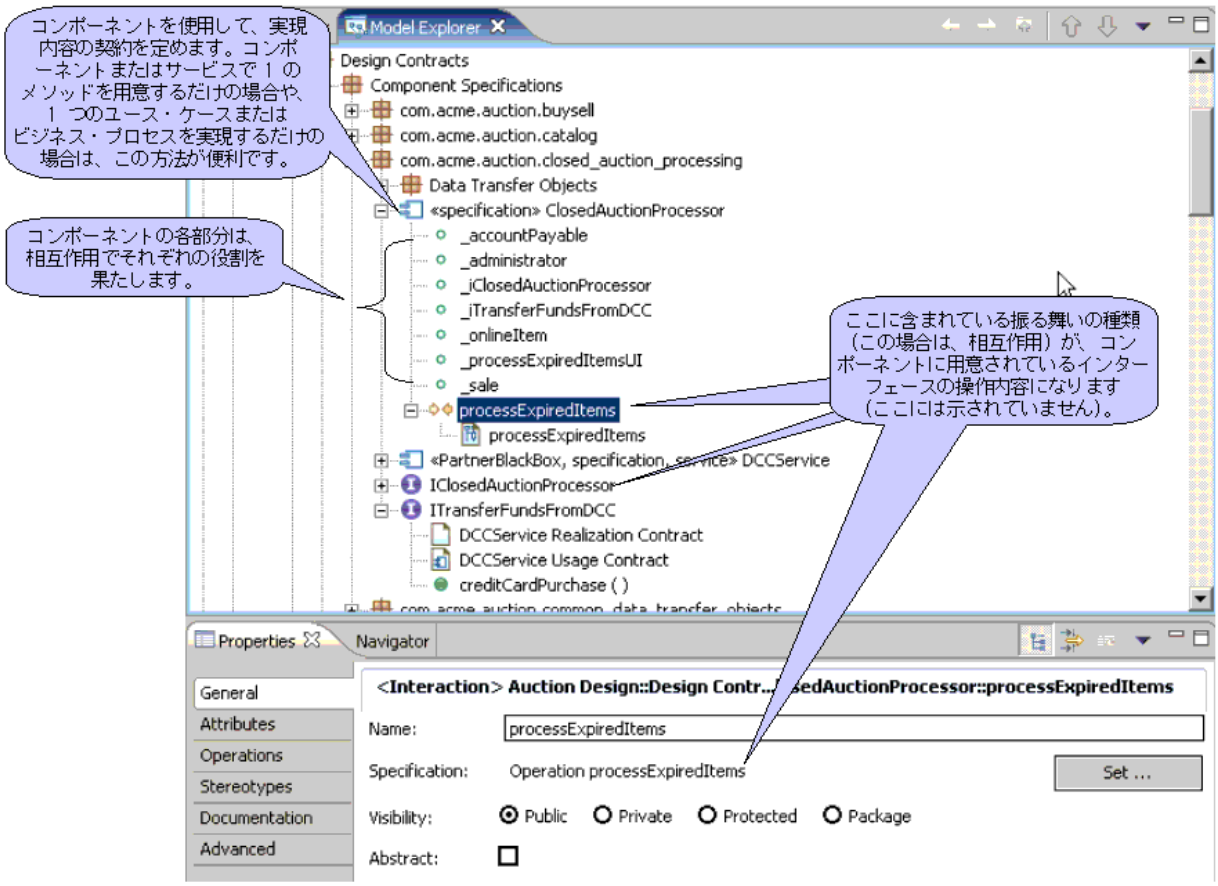


図 7-4

- 実装設計を指定するために利用できる方法の 1 つを、**図 7-5** に示します。実装の構造は、操作を含む単純なクラスを使用して定義します。この方法は、UML 1.x を使用して作成する設計モデルとして、ごく標準的なものです。利用できる 2 つめの方法を、**図 7-6** に示します。これは、UML 2 の目標により沿ったものです。この方法では、クラスの代わりにコンポーネントを使用しています。コンポーネントは、操作を所有しておらず、代わりに動作 (この例では相互作用) を所有しています。

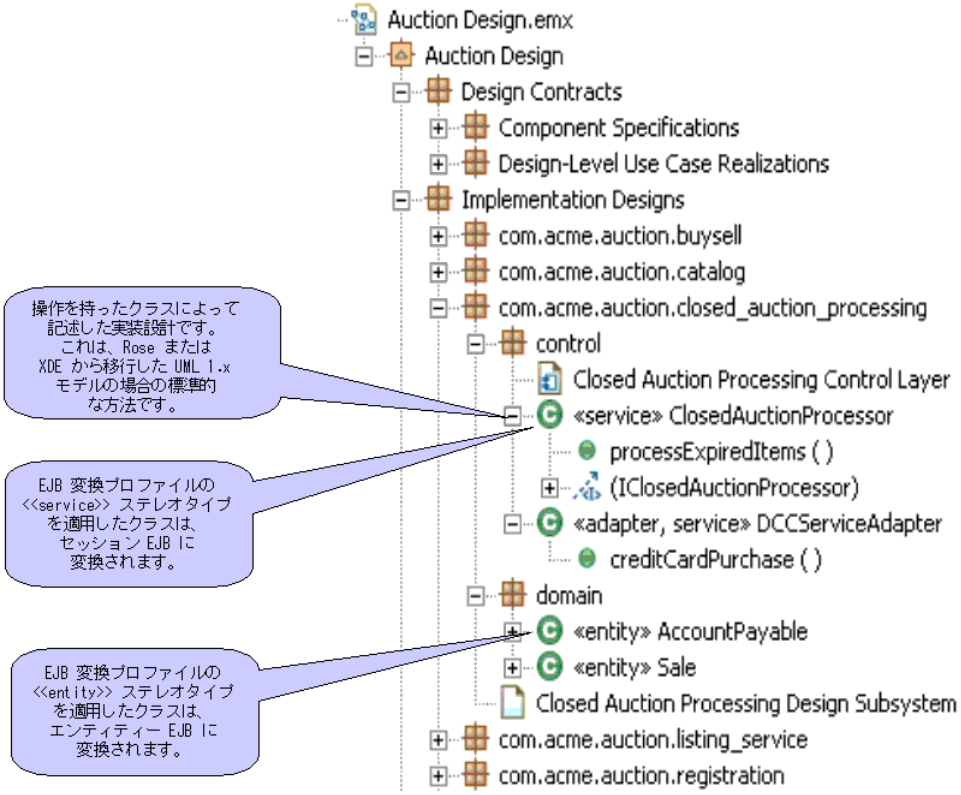

Auction Design.emx
Auction Design
Design Contracts
Component Specifications
Design-Level Use Case Realizations
Implementation Designs
com.acme.auction.buysell
com.acme.auction.catalog
com.acme.auction.closed_auction_processing
control
Closed Auction Processing Control Layer
«service» ClosedAuctionProcessor
processExpiredItems ( )
(IClosedAuctionProcessor)
«adapter, service» DCCServiceAdapter
creditCardPurchase ( )
domain
«entity» AccountPayable
«entity» Sale
Closed Auction Processing Design Subsystem
com.acme.auction.listing_service
com.acme.auction.registration
操作を持ったクラスによって記述した実装設計です。これは、Rose または XDE から移行した UML 1.x モデルの場合の標準的な方法です。
EJB 変換プロファイルの <<service>> ステレオタイプを適用したクラスは、セッション EJB に変換されます。
EJB 変換プロファイルの <<entity>> ステレオタイプを適用したクラスは、エンティティー EJB に変換されます。

図 7-5

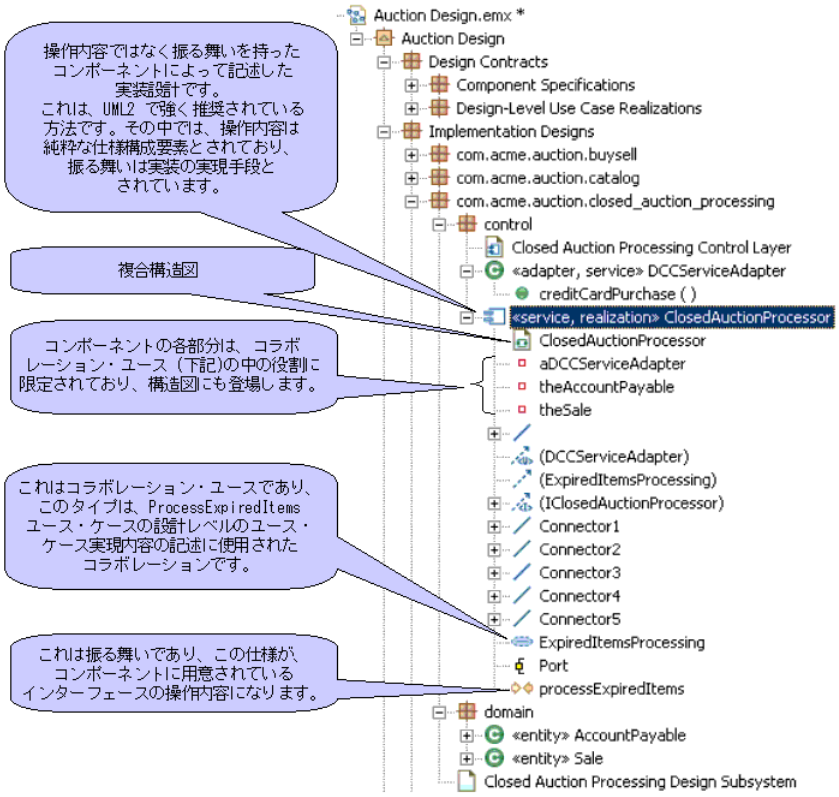


図 7-6

8. 実装概要モデルの内部編成のためのガイドライン

> ***XDE/Rose***
> XDE のモデル構造ガイドラインでは、実装概要モデルは、実装のサブシステム・レベルの概要を提供する手段として推奨されていました。その上で、各サブシステムの詳細は、そのサブシステムを実装するプロジェクトのコード・モデルの中で指定していました。

厳密に言うと、RSx では実装概要モデルを使用する必要はありません。設計モデルの編成ガイドラインに従っていれば、設計モデルの (最終段階の) 編成は自然にコンポーネント (より大きい «subsystem» と、より分散可能な «service» のバリエーションを含む) の周辺に形成されます。その後、変換機能により、設計のパッケージをプロジェクトにマッピングできます。たとえば J2EE 実装の場合、各種の Java、EJB、Web、J2EE アプリケーションや、実装が開発されている他のプロジェクトにマッピングされます。(これらのプロジェクトは、実際にはソリューションの実装モデルを表しています。このホワイト・ペーパーの『基本的な概念と用語』セクションで説明したとおりです。)

ボトムアップの観点で考えるのであれば、実装の編成方法をプロジェクトとフォルダーの観点から考え、それを設計モデルの編成要素として組み込み、変換マッピングが単純明快なものになるようにしたほうがよいと言えます。トップダウンの観点で考えると、設計モデルの編成を設計作業の中で進展させながら、それによって必要な実装

モデル (プロジェクト) のセットを決定するようにしたほうがよいと言えます。どちらにしても、設計モデルの最終段階の編成は、プロジェクトやサブプロジェクトの概要というニーズに自然に対応するはずです。言い換えれば、設計モデルのパッケージを記述する図は、プロジェクトとそのサブフォルダーの概要に相当するものになるはずです。

とはいえ、早い段階からプロジェクト構造のスケッチを作成するのを好む開発者や、視覚的に分かりやすいプロジェクト構造の表現を好む開発者もいることでしょう。「視覚的に分かりやすい」とは、たとえばプロジェクトやフォルダーを表す成果物に «project» や «folder» といった明示的なキーワードを付けたり、さらには «EJB Project» や «Web Project» といったより具体的なキーワードを付けたりする表現のことです。さらに考慮すべきなのは、実装の成果物を (たとえば JAR のように) 具体的に表現するのは、設計モデルとしては不適切であるということです (Rational Software Architect の操作理論では、設計モデルをプラットフォームに対して中立にすることになっています)。しかし、実装概要モデルであれば、そのような成果物を問題なく含めることができます。こうした理由から、実装概要モデルを使用したほうがよい場合があります。以下の**図 8-1** に、実装概要モデルの例を示します。

実装概要モデルは、ソリューションのさまざまな面を表す自由形式の図を取り込むのに適している、ということを最後に挙げます。**図 8-2** は、このホワイト・ペーパーで取り上げている大部分の実例の基になっているオークション・システムの、大まかな概念図です。

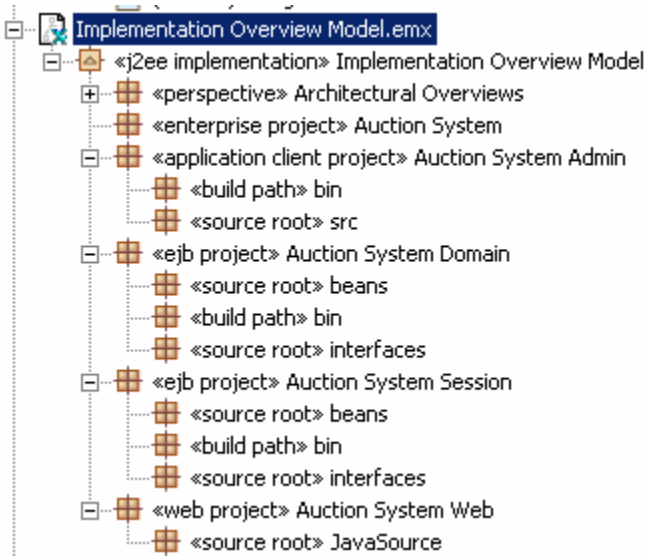


図 8-1

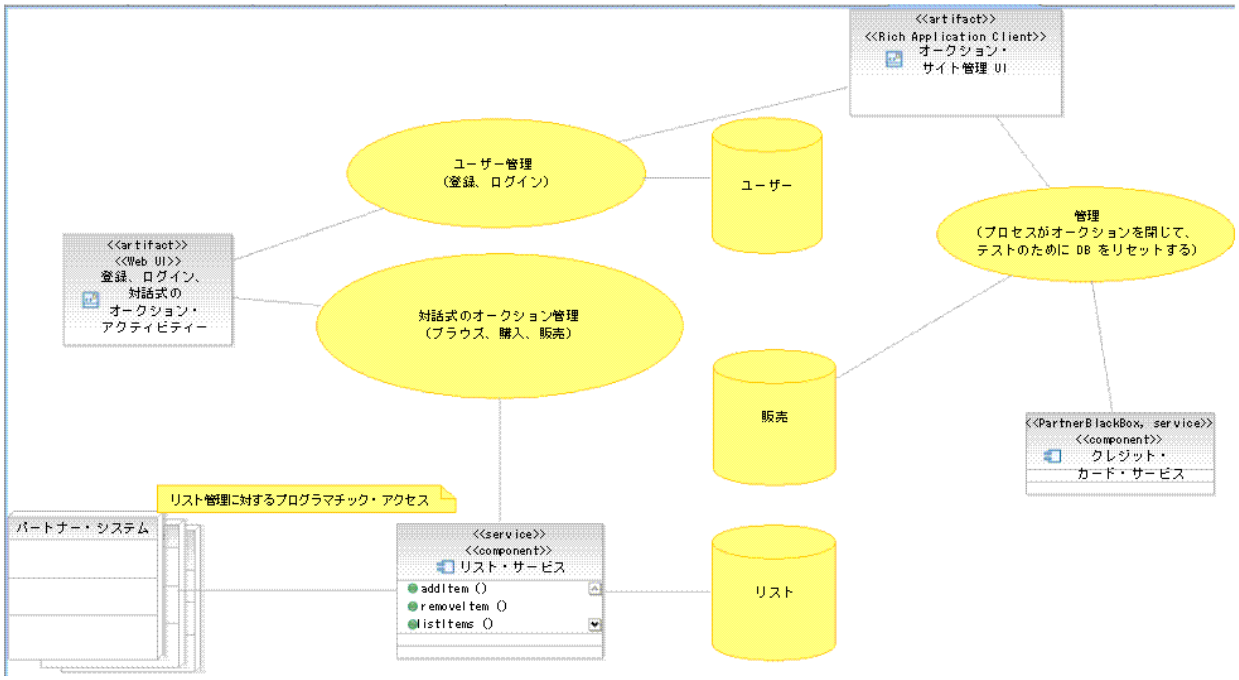

<<artifact>>
<<Rich Application Client>>
オークション・
サイト管理 UI
ユーザー管理
(登録、ログイン)
ユーザー
管理
(プロセスがオークションを閉じて、
テストのために DB をリセットする)
<<artifact>>
<<Web UI>>
登録、ログイン、
対話式の
オークション・
アクティビティー
対話式のオークション管理
(ブラウズ、購入、販売)
販売
<<PartnerBlackBox, service>>
<<component>>
クレジット・
カード・サービス
リスト管理に対するプログラマチック・アクセス
パートナー・システム
<<service>>
<<component>>
リスト・サービス
addItem ()
removeItem ()
listItems ()
リスト

図 8-2

9. 配置モデルの内部編成のためのガイドライン

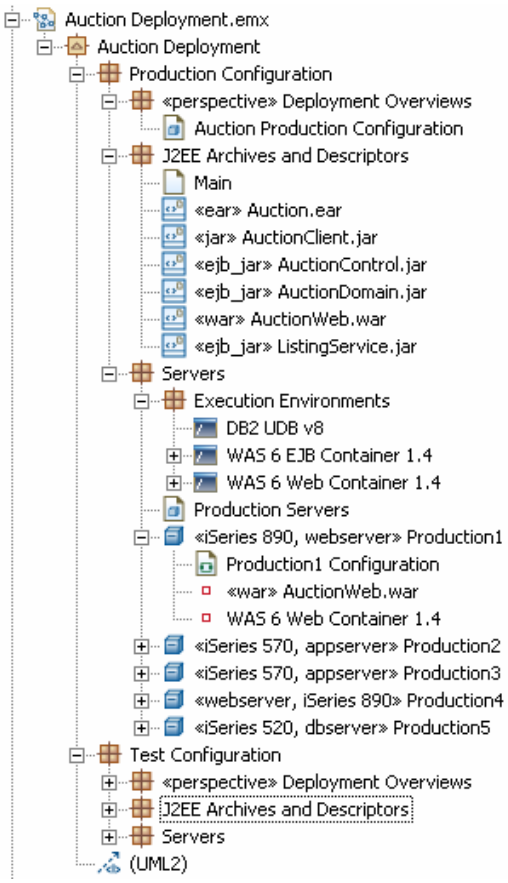


図 9-1

このホワイト・ペーパーでは、他のすべてのモデルに比べて、配置モデルについては説明する必要があまりないと考えています。配置モデルの編成やその内容の選択は下流に対してほとんど意味を持たないことが多いので、意味のあることだけを行えば十分です。ですが、考えのきっかけとなるように、利用可能な戦略と代表的な内容を上の**図 9-1** に示しておきます。この例では、以下の点のみに注意してください。

1. 実稼働構成の仕様を、テスト構成の仕様から分離してあります。

2. 概要 (クラスター、データ・センター、企業の概要など) は、«perspective» パッケージ内に保持されています。

3. ノードと成果物の特化および分類に関しては、パッケージの組み合わせとキーワードの使用という簡便な方法を採用しています。より高度な方法を使えば、それぞれの環境で使用されているリソースのタイプを記述および文書化するのに適した、特別なステレオタイプとプロパティーを定義する、特別な UML プロファイルを開発できます。

10. ソフトウェア・アーキテクチャー説明書を表すためのモデリング・ファイルの使用

モデルを編成するツール (図のリンクや、モデル間参照による複数のモデル・ファイルのサポートなど) が用意されたことによって、RUP ソフトウェア・アーキテクチャー説明書および「アーキテクチャーの 4+1 ビュー」を実質的に表すモデルの作成は、ごく簡単な作業になりました。

最も単純な方法は、**図 10-1** にならって作業することです。モデリング・ファイルを作成し、4+1 ビューに対応する単純なパッケージ・セットをそのファイルに追加します。(この例では、プロセス・ビューのパッケージを省略しています。この例のシステムでは、並行性の方法があまり示されていないからです。)

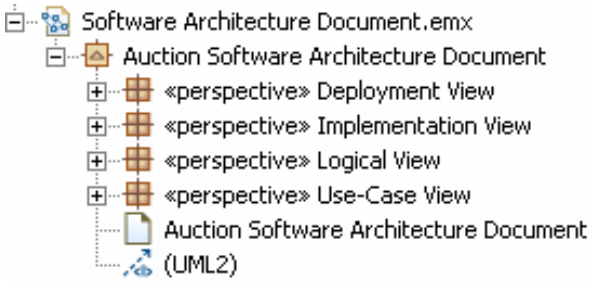


図 10-1

図 10-2 の例にならえば、デフォルトの図を作成できるはずです。この図に、注記やテキストを追加することもできます。

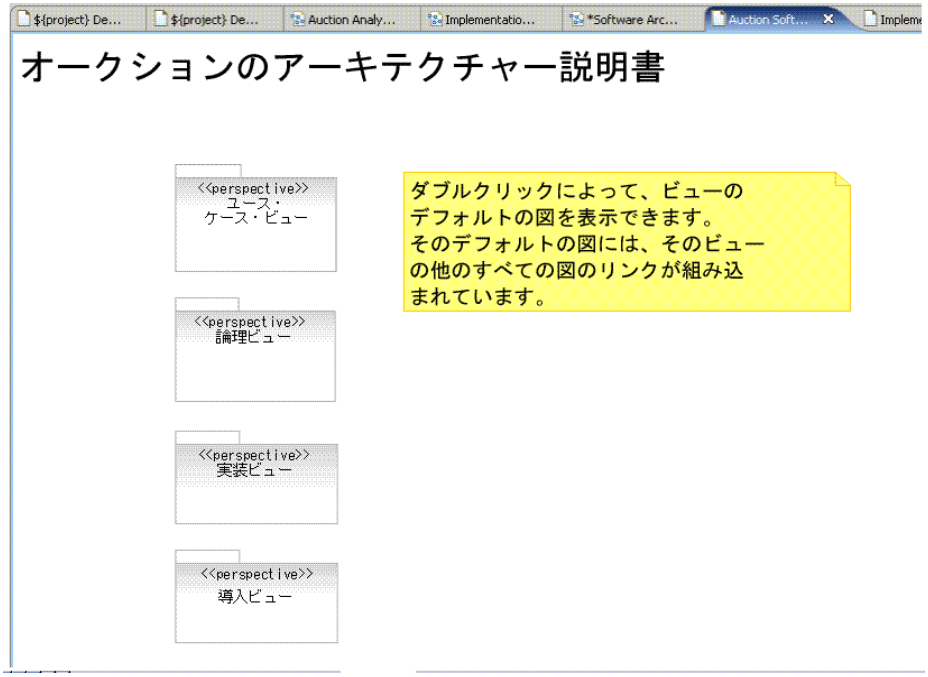


図 10-2

続いて、ソフトウェア・アーキテクチャー説明書のモデリング・ファイル内に、図を作成します。その方法は、以下のとおりです。

- 他のモデリング・ファイルにある UML セマンティクス要素を使用して、それらのモデリング・ファイルには含まれていないが、アーキテクチャー説明書の一部として必要な、新しいビューとなる図を作成します。

- ソフトウェア・アーキテクチャー説明書のモデリング・ファイルに含まれる、幾何学形状や「その場限り」の UML 要素で構成される図を作成します。(そのような UML 要素は、文書化や明確化だけの目的で使用してください。記述しているソリューションの実際の実装に対する、セマンティクス上の重要性を持たせてはいけません。)

- 他のモデリング・ファイル内にある、既存の図に対するリンクを含むだけの図を作成します。(この方法が有効なのは、アーキテクチャー説明書のモデリング・ファイルを、他のモデリング・ファイルと一緒に読者に配布する場合です。アーキテクチャー説明書を Web で公開する場合には、上記のいずれかの方法を使用してください。)

11. チーム開発とモデル管理の考慮事項

このセクションでは、モデルを複数のモデリング・ファイルとして分割するべきなのはどんな場合か、そうするべき理由は何であるのか、ということに関する考慮事項を紹介します。これらの問題の総合的な説明については、RSx のオンライン・ヘルプを参照してください。ここでは、並行開発の概念と、成果物の複数コピーに伴って作成される変更内容のマージという概念について、ある程度精通している読者を想定しています。

ユースケース・モデル、分析モデル、設計モデルなど、RUP が認識する各種のモデルについては、『基本的な概念と用語』セクションで簡単に説明しました。RSx における次のような状況が、図で例示されています。

- 複数のアプリケーションをビルドする場合は、各タイプに複数のモデルがある場合がある (たとえば、複数のユースケース・モデルや複数の分析モデルなど)
- (論理的な意味での) モデルは、1 つ以上のモデリング・ファイルとして保持することができる。たとえば、「アプリケーション 'X' の設計モデル」は、1 つのモデリング・ファイルとして保持することも、複数のモデリング・ファイルの集合として保持することもできる。

モデルの分割

モデルを複数のモデリング・ファイルに分割するテクニックについては、RSx のオンライン・ヘルプに記載されているので、ここでは説明しません。ここでは、分割を行う時期と理由について説明します。特定のモデルを複数のモデリング・ファイルとして維持する状況としては、2 つの状況があります。

1. モデルのサイズが管理不可能なほど大きくなっているか、そのパッケージ構造が管理不可能なほど深くなっている場合[15]
2. モデリング・ファイルへの変更の並行デリバリーが多すぎるという状況が発生しはじめたため、3 人以上の変更投稿者[16]とのマージを実行することが必要になった場合(この説明が分かりにくい場合は、次を参照してください。)

チームでのモデリング

構成管理ポリシーによってモデルの並行開発[17]が認められている場合、ファイルに対して (事実上、論理モデルや、ファイルが表すモデル・サブセットに対しても) 非整合の変更が加えられます。ある時点で、変更内容をマージする必要があります。変更の一部が別の変更と矛盾することが判明した場合、マージは「**単純ではない**」と見なさ

[15] 分割が主に必要となるのは、ユーザーのコミュニティーで使用しているマシンが扱えないほど、ファイルが大きくなりすぎたときです。たとえば、ディスク上で 30MB のサイズになったモデルを、RAM が 1 GB の マシンで日常的に操作するのは、非常に難しくなります。そのようなマシンでは、RAM 内でのサイズを 5 ～ 10MB 程度に保つことを目標にして、モデルを分割したほうがよいでしょう。もう 1 つは RAM を増設することです (比較的費用のかからない解決策ですが、非常に大きい効果が得られます。2 GB の RAM を搭載したマシンでは、スワップ･ファイルが発生せず、Eclipse のほぼすべての操作が高速で実行されるため、非常に大きいモデルの作成もうまくこなすことができます。)

[16] RSx モデルのマージ・ツールは、最大で 3 人の投稿者のマージをサポートしています。2 人は「変更」の投稿者で、1 人は「ベース」の投稿者 (別名、共通の祖先) です。処理を行う「変更」投稿者が 3 人以上になると、マージのカスケードが始まります。これ以降、2 人の「変更」投稿者のうちの 1 人が、セッション内の事前に実行済みのマージの結果セットとなります。

[17] 「並行」開発ポリシーの例:

- モデリング・ファイルは、非排他的アクセスをチェックアウトことができます。
- モデリング・ファイルは、複数の実行者による開発の流れの中で、並行して処理されます。

れます。矛盾している変更のうちのどれを「優先する」かを、極める必要があるためです。(「単純な」マージとは、非整合の変更内容に矛盾がなく、モデルのマージ・エンジンがユーザーの介入なしで実行できるマージを指します。)
単純ではないマージを行なうのは、手間が掛かる場合があります。どのようにすれば、これを最小限に抑えることができるのでしょうか。

矛盾の発生と、その結果として起こる単純ではないマージを回避するには、2 つの基本的な対抗手段があります。1 つは「強力なアーキテクチャー」で、もう 1 つは「強力な所有権」です。この両者は、密接な関係を持っています。強力なアーキテクチャーによって、強力な所有権が *可能になるのです*。

対抗手段 その 1:強力なアーキテクチャー

ここでの「強力なアーキテクチャー」は、主に分解のことを指します。ここで適用されるアーキテクチャー分解の *原理* は、オブジェクト指向開発、コンポーネント・ベース設計、およびサービス指向アーキテクチャーを動かす原理と同じものです。

- ビジネス機能を最大限分離するよう努める。

- 密結合のままにしておく必要がある事柄を 1 つにまとめ、そのグループをほかのグループから分離させる。

- 分解した結果に多数の「グレーン」が含まれている場合は、配属モデルに応じて (強力なアーキテクチャーと強力な所有権には密接な関係があるため)、それらのグレーンを親和性の高い集合体 (モデリングの観点では UML パッケージを意味する) にグループ化することができる。

- 多数の (場合によってはすべての) 分解単位に関係しなければならないものが、*必ず* 存在する。それらのものを「共通」パッケージの中で 1 つにまとめる。そして、「共通」事項の固定に焦点を当てた反復の開始点に、小さい「ウォーターフォール」を組み込む形で、各開発の反復を計画する。

- 時間的な要因も存在する。ソリューションに対する理解が、抽象的なものからより具体的なものに移るにしたがって、アーキテクチャー (およびモデル) を最良の編成にするためのセンスが進化していく。それぞれの段階から次の段階 (ビジネス分析、要件、アプリケーション分析、おおまかな設計など) に移行する時点で、モデルのリファクタリング (再編成) を計画することが望ましい。

ソリューションを調べたときに、すべての要素が高い相互依存を示し、密結合であるように見えたのであれば、アーキテクチャーに対する作業が必要であるか、問題領域の本質に関して、真に問題箇所を分解することができないことを意味するものがあるか、いずれかです。いずれにせよ、選択肢は 2 つです。
    - プロジェクトを非常に小さいチームに割り当てる。そのチームで物理的スペースを共有し、あるチームが行う変更が他のチームの成果物に影響を与える可能性について、別のチームと積極的に話し合うことで、解決を図る。
    - 単純ではないマージを、大量に実行する準備を整える。

XDE/Rose

Rose ユーザーまたは XDE ユーザーであるなら、重要なマージを行った経験があるでしょう。RSx ではこの重要なマージの面倒がはるかに軽減されます。主な違いの 1 つは、RSx はファイルまたはサブユニットの*階層* ではなく、関連するモデリング・ファイルの*網目* の上で作動するということです。この重大な違いは、大まかな論理的分解を、RSx では物理レベルでよりサポートできることを意味します。

また、より優れたツールとの比較マージもサポートします (つまり、**非常に**高レベルのマージを行うことができます)。

- RSx のバックエンド・マージ・エンジンは、XDE のバックエンドの 10,000 (1 万) 倍高速。
- また、「複合デルタ」(ダイアグラムによって変更をソートしたり、大きなダイアグラム作成ジェスチャーをグループ化してグループおよび/またはアトミック・オペレーションを使用可能にする、グループ化メカニズム) 、「モデルの整合性の保護」、「アトミシティ」、および「カスケード・デルタ処理」など多数のテクノロジーを実装している。これらは単なるファイルの編集とほぼ同じレベルにマージすることで、ファイル破損の可能性を削減。
- レイアウト/外観の不一致を解決するための、真のビジュアル・ダイアグラム・マージ GUI がある。

対抗手段 その 2:強力な所有権

いったん強力なアーキテクチャー分解を確立してしまえば、アーキテクチャーのコンポーネントの「強力」な所有権を個々の実行者や小さいチームにマッピングすることは、とても簡単である (専門的なスキルは別にして) ことが分かるはずです。モデル内のそれぞれの論理的パッケージ (またはブランチ) を 1 人の実行者が独占的に処理できる場合、そのモデルで実行するマージはたいていは単純です (モデルが 1 つのモデリング・ファイルとして保管されているか、複数のモデリング・ファイルとして保管されているかとは無関係です)。作業内容についてメンバー同士が気軽に話し合うことが多い小規模なチームが、それぞれのブランチを独占的に処理できる場合も、同じことが言えるでしょう。

モデルを複数のモデリング・ファイルに分割することによって、単純でないマージを回避することができるのでしょうか。一言で言えば、「できません」。**アーキテクチャーの相互依存性は論理的な現象であって、物理的な現象ではありません。**モデルを複数のモデリング・ファイルに分割しても、ファイル内の参照として表現されていた要素の相互依存性が、代わりにファイル間の参照として表現されるようになるだけです。矛盾点を解決することが、簡単になるわけではありません (むしろ難しくなります)。さらに、ファイル間の参照を導入すると、破損する可能性のあるポイントまで導入してしまうことになります (補足説明を参照)。

補足:モデリング・ファイル間の参照

2 つのモデリング要素が異なるモデリング・ファイル内に存在し、それらの関連を作成するときには、「モデリング・ファイル間の参照」が作成されます。モデリング・ファイル (.emx ファイル) はホスト OS ファイル・システムで公開され、Eclipse 環境外で移動、リネーム、あるいは別の方法で変更される場合があるため、これらの参照は破損する可能性のあるポイントを表します。ただし、モデリング・ファイルが常に Eclipse 環境で変更および変更管理され、ユーザーが以下のガイドラインを順守している限りは、破損することはありません。

モデリング・ファイルの「クロージャー」 (つまり、相互参照するモデリング・ファイルの集合) を処理 (編集) する際は必ず、そのクロージャーのすべてのモデリング・ファイルがワークスペースに存在しなければなりません。これは、クロージャーに含まれるすべてのモデリング・ファイルが同一のプロジェクトに存在することを厳密に意味するものではありませんが、標準の CM ワークフローではすべてのモデル・ファイルが同一のプロジェクトで処理されるため、1 つのプロジェクトの使用は通常、すべてのモデルの存在を保証します。

ここまでをまとめると、以下のようになります。

- 強力なアーキテクチャーがない場合、または強力なアーキテクチャーがあっても強力な所有権がない場合は、モデルの分割による軽減がない、単純ではないマージが頻繁に行われます。

- 強力なアーキテクチャーと強力な所有権がある場合、単純ではないマージの頻度が大幅に削減されます (ただし、発生しなくなるわけではありません)。コンポーネントの相互依存性は常に存在するため、このようなマージがなくなることはありません。ほぼ唯一の例が、前述の「共通」要素です。

- モデルを論理的に構成して、複数の実行者が矛盾した変更を取り込まずに、並行してモデリング・ファイルを処理できるようにすることに比べれば、モデルを複数のファイルに分割することは、あまり重要ではありません。

- ただし、他の使用可能なモデリング・ツールよりも、RS*x* はモデルのマージがはるかに高速で、より効果的に処理される点は、救いといえます。

モデルを分割する時期は、いつが最適でしょうか。モデル全体のサイズがハードウェアの限界に差し掛かる状態になるまで、この作業を行なわないようにしてください。作成されるモデリング・ファイルは、通常は「独占的に」 (どの時点であっても、1 つのファイルをチェックアウトしているチーム内のメンバーは 1 人だけ、という状態で)、**かつ**「分離された状態で」 (関連するモデル要素を含む他のファイルにアクセスしなくても、目的のファイルにほとんどの変更を加えることができる状態) 処理することができます。

最後に: RSx バージョン 6.x から 7.x への変更点

RSx が最初にリリースされた時点 (バージョン 6.x) では、Rose や XDE でサポートされているような「サブユニット」という概念はサポートされていませんでした (サブユニットは、モデルのサブセットを含むモデリング・ファイルであり、Model Explorer ビューに表示されるモデルの論理ビューには表示されないという意味で「透過的」です)。代わりに、モデルの分割は、複数の最上位モデル (「最上位」とは、それぞれのモデリング・ファイルが、個別の最上位エントリーとして、Model Explorer ビュー内に表示されるという意味) を定義することに限定されていました。

RSx には、既存モデルの UML パッケージを選択し、そのパッケージを基にして「モデルを作る」ことができる機能もあります。パッケージを「切り取って」、新規のモデリング・ファイルを作成する、と考えてください。その結果、Model Explorer ビュー内では、パッケージが新しい最上位の論理モデルになったように見えます。元の包含関係の階層構造の可視性は失われますが、このようにしてパッケージから「切り取られた」モデリング・ファイルを開くときは、「切り取られた」すべてのモデルも常に開きます。これによって、不必要なチェックアウトが行われる可能性があります。また、Model Explorer ビューが過密状態になり、エディター区画に多くのタブが表示されるため、ナビゲーションしにくくなる可能性があります。

バージョン 7.0 の RSx では、サブユニットのサポートが追加されました。これによって、階層構造の可視性を失うことなく、モデルを複数のファイルに分割できるようになりました。さらに、エディター区画の「モデル・エディター」タブが除去されたので、ナビゲーションの問題も解消されています。

しかし、以前の 6.x の経験から学んだ、モデルの構成に関する教訓は、現在でもまだ生きています。上で述べたように、パッケージから切り取られたモデリング・ファイルを開くと、切り取られたモデルがすべて開くことになります。これは、「モデルを作る」機能を使用する代わりに、分割戦略を前もって計画しておくことによって回避できます。

この後に「モデルを作る」機能を使います。通常は単一のモデリング・ファイルから始めて、次にソリューション・アーキテクチャーの編成を表すパッケージを作成します。たとえば、ソリューションの主要なコンポーネントまたはサブシステムごとにパッケージになる場合があります (ここでの「コンポーネント」および「サブシステム」という用語は、正式な UML セマンティクス定義ではなく、初期のコンピューティングに遡った、非公式な使い方をしています)。「共通」、「ユーティリティー」、または「フレームワーク」のコンポーネントのためのパッケージになる場合もあります。こうしたモデルのサイズが大きくなると、アプリケーション固有のコンポーネントに対応するパッケージが別個のモデリング・ファイルとして「切り取られる」ことになります。これによって、そのコンポーネントを構築するチームが、(理論上は) モデル・ファイルを分離された状態で処理できるようになります。しかし、実際には、コンポーネント固有のモデル・ファイルのいずれかを開くと、元の「マスター」モデル (「共通」部分が存在するモデル) が開き、それによってその他のアプリケーション・コンポーネント固有のモデルすべてが開くことになります。結局、前述のとおり、不必要なチェックアウトが発生してしまうので、ナビゲーションが難しくなります。

改善策は、「共通」のコンポーネントのモデル (共通するコンポーネントや再利用可能なコンポーネントの連続する層を定義する、つまり実装アーキテクチャーの抽象化層を反映する、複数のモデルの場合もある) とともに、アプリケーション固有のコンポーネントごとに別個のモデルを定義するように、あらかじめ計画しておくことです。これによって、アプリケーション固有のコンポーネント・モデルを、それを所有するチームが開くことができ、アプリケーション固有のモデルが依存する「共通」のモデルだけを、開けばよいことになります。