修士学位論文

題目

メタモデル記述を用いた成果物間の依存関係追跡手法

指導教員
井上 克郎 教授

報告者
大平 直宏

平成 18 年 2 月 13 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 17 年度 修士学位論文

メタモデル記述を用いた成果物間の依存関係追跡手法

大平 直宏

内容梗概

UML（Unified Modeling Language）の普及にともない，仕様書や設計書をモデルとして形式的に記述するようになってきている．モデルを用いたソフトウェア開発では，初期の要求分析から実装に至るまでの開発工程において，各工程にあったモデルを用いて開発が行われる．この時，成果物を表わすモデルは，対象業務やフレームワーク，開発方法論に依存して様々なものが用いられる．そのため，異なるモデルを厳密な意味論のもとで管理することが重要であり，モデルに修正が加えられた際には，厳密に定められたモデル間の依存関係に基づいて他のモデルも正しく更新されなければならない．しかし，既存の開発方法論では，この種の管理を人手に頼っているため，モデルを用いた開発による利点が生かされていなかった．

そこで本研究では，メタモデルを扱う技術として MOF（Meta Object Facility）に着目し，成果物とその依存関係を統一的に管理可能な枠組みの提案を行う．具体的にはまず，個々の成果物に対して仕様であるメタモデルの定義を行い，メタモデルに基づいてモデルを形式的に検証する手段を提供する．次に，成果物間の依存関係定義および実際の依存情報も同様にメタモデルを用いて形式的に定義することにより，モデル相互の関係を明確にする．これにより，実際に開発を行う際に，成果物間の依存関係を追跡可能な状態として保持することが可能となる．また，提案する枠組みを Web アプリケーションフレームワーク Struts を用いた画面遷移設計に適用し，実際に動作するシステムとして実現可能であることを確認した．

主な用語

モデル駆動型開発（Model Driven Development）
追跡可能性（Traceability）
メタモデル（Metamodel）
MOF（Meta Object Facility）

目 次

1 まえがき

近年，UML（Unified Modeling Language）[29] の普及にともない，ソフトウェア開発は“モデル”を中心に行われるようになった．モデルとは，形式的な意味定義がなされたモデリング言語（ex. UML）を用いて対象の情報を抽象化したものである．UML はオブジェクト指向設計論 [3] にもとづいて，ソフトウェア成果物である仕様書や設計書をモデルとして記述する．UML を用いることによって，開発者間の意味解釈の誤りを減らすことができる．

モデルを中心としたソフトウェア開発の考え方はさらに進み，現在では MDD （Model Driven Development：モデル駆動型開発）と呼ばれる開発手法が提案されている．MDD では，MDA （Model Driven Architecture）[24] の考えにもとづいて，要求分析から実装に至るまでの全ての開発工程をモデル中心に行う．加えて，モデルに対する変換や検証，実行といった高度なプログラム処理を適用する．これにより，設計モデルと実装モデルの分離を可能とし，再利用性や変更容易性に優れたソフトウェアの開発を目指している．

しかし，このようなモデル中心のソフトウェア開発を実現するにあたっては多くの課題が残っている．モデルの変換手法 [11, 12, 18] や検証手法 [8, 21]，モデルを扱うツールの統合 [5] に関する問題など様々な研究がなされており，OMG（Object Management Group）[22] も各種標準 [25]-[29] の策定をすすめている．なかでも，モデルを中心としてソフトウェアを開発する際に重要となるのが，異なるモデルをいかに管理し，それらモデル間の依存関係（トレース情報）をいかに蓄積していくのかという問題である．

モデルの作成には UML が用いられることが多いが，実際の開発においては必ずしも UML モデルによる管理が適切でない場合もある．というのも，ソフトウェア開発における成果物の記述形式は，対象業務や使用するフレームワーク，組織の採用する開発方法論などによって異なるからである．UML は汎用性のため様々なモデルを作成可能であるが，逆に必要以上の内容を記述できてしまうことが成果物を厳密に管理する上で問題となることもある．

また，異なる開発工程において互いに依存するモデルを管理する方法についても標準的な枠組みは定められていない．開発サイクルにおける“要求”の追跡 [10, 13, 14] やソースコードと設計文書間の依存情報の復元 [2] などの研究もあるが，モデルを中心とした開発においてはより汎用的な，モデルレベルでの手法が望まれる．あるいはモデル間のマッピングに関する研究 [1, 6, 9, 15] もあるが，対応関係の記述法に焦点をあてたものが多く，成果物との関わりの中で実際に依存情報を蓄積していくという点に着目したものはない．

そこで，本研究では，メタモデルを扱う技術として OMG 標準の一つである MOF（Meta Object Facility）[23, 26] に着目し，成果物とその依存情報（トレース情報）を統一的に管理可能な枠組みの提案を行う．提案手法は，

1. 個々の成果物（モデル）に対して仕様をメタモデルとして形式的に定義し，各成果物

が仕様を満たす状態で作成されていることを保証する

2. 成果物間の依存関係の定義，および実際に作られたトレース情報も形式的に定義したメタモデルのもとで管理する

という二つの手順によって成果物とトレース情報を管理可能とする．そして，MDD や RUP[19] のようなモデルに繰り返し修正が加えられる開発プロセスにおいても，適切にこれらの情報を更新することで成果物間の依存関係を追跡可能とする．また本研究では，提案する枠組みを Web アプリケーションフレームワーク Struts[4] を用いた画面遷移設計に適用し，実際に動作するシステムとして実現可能であることを確認した．

以降，2 節では研究の背景となるモデル中心のソフトウェア開発手法と Web アプリケーションフレームワーク Struts について説明し，3 節ではメタモデルを定義する枠組みとして着目した MOF について説明する．4 節では提案手法である成果物間の依存関係追跡手法を説明し，5 節では Struts に適用した場合の支援システムについて述べる．最後に，6 節でまとめと今後の課題を述べる．

2　モデルに基づくソフトウェア開発の現状と課題

本節では，モデルを中心としたソフトウェア開発手法について説明した後，このような開発手法の適用が可能なソフトウェアの一例として Web アプリケーションフレームワーク Struts について説明する．その後，モデルを中心としたソフトウェア開発が抱える課題を説明した上で，関連研究について述べる．

2.1　モデルを中心としたソフトウェア開発

近年，モデリング言語である UML（Unified Modeling Language）[29] の普及にともない，“モデル” を中心としてソフトウェアを開発するという考え方が広く認知されつつある．ここでは，まず UML を用いたソフトウェア開発手法について述べ，その後，さらにモデルを重視した開発手法であるモデル駆動型開発について述べる．

2.1.1　UML を用いたソフトウェア開発

UML は OMG（Object Management Group）[22] によってその仕様が標準化されているモデリング言語である．UML はソフトウェア開発の上流工程から下流工程まで利用でき，かつシステムの様々な側面をモデル化できるように，図 1 のような多くのダイアグラムを提供している（UML2.0 現在で 13 種類）．加えて，UML では各ダイアグラムに対する表記法とその意味が厳密に定められているため，UML ダイアグラムを用いて記述したモデルはその構成部品（モデル要素）に至るまで詳細に意味が定義されている．

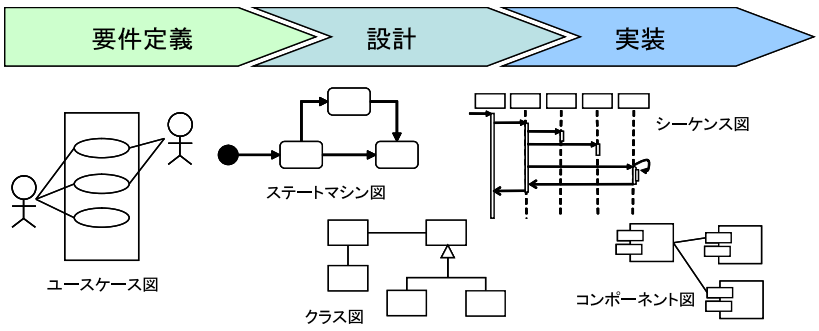

図 1: UML が提供するダイアグラムの一部

従来のソフトウェア開発では，仕様書や設計書のような成果物の多くがテキスト文書として自然言語で記述されていた．そのため，記述すべき内容や使用する語彙に関する規則があらかじめ決められていたとしても，曖昧さを取り除くことはできなかった．その結果，開発

工程間や開発者間で成果物の解釈を誤ってしまうという問題が起こっていた．

しかし，UML の普及によって，各工程の成果物は意味定義が標準化されたモデルとして記述されるようになった．そのため，従来の自然言語に比べるとはるかに曖昧さを減らすことが可能となり，開発者に共通のコミュニケーション基盤をもたらした．結果，モデルの重要性は広く認識されるようになり，自然言語で記述された設計書やプログラミング言語で記述されたソースコードよりも，モデルを中心としてソフトウェア開発を行う手法が認められるようになった．つまり，要求定義から設計・実装工程にいたるまで，各工程の成果物は全て UML を用いてモデルとして記述し，最終的な実装段階でソースコードとしてプログラミングを行うという流れである．UML を用いたソフトウェア開発では，後戻りを許さないウォータフォール型ではなく，互いに関連するモデルを繰り返し修正・洗練させていくことで，段階的にソフトウェアを作り上げていく RUP（Rational Unified Process）[19] のような開発プロセスを用いることが多い．

2.1.2 モデル駆動型開発

モデル駆動型開発（MDD:Model Driven Development）とは，モデル駆動アーキテクチャ（MDA:Model Driven Architecture）[24] という考え方に基づいたソフトウェア開発手法をいう．MDA は，モデルを中心としたソフトウェア開発によって設計と実装を分離し，モデルの自動変換を用いることで移植性や相互運用性，再利用性に優れたソフトウェアの開発を目指す．MDA におけるモデルは，

- CIM（Computation Independent Model）
  計算処理とは独立なモデル．ドメインモデルとも呼ばれ，システムの機能や詳細を知ることなしに作成されるモデル．
- PIM（Platform Independent Model）
  CPU や OS，フレームワーク，実装言語など開発対象となるプラットフォームに依存しないモデル．
- PSM（Platform Specific Model）
  開発対象のプラットフォームに依存したモデル．

に分類され，CIM から PIM，PSM へと徐々に詳細化していくことによってソフトウェアを作り上げていく．このとき，PIM から PSM への詳細化に際しては，MDA 対応のツールがモデルの自動変換を行うことによって，対象の技術・開発言語などに適したモデルの作成を支援する．開発者が個々のモデルを個別に修正していた従来の開発手法とは異なり，MDA ではモデルの実行や検証，変換といったツールによる自動処理を積極的に推進する．

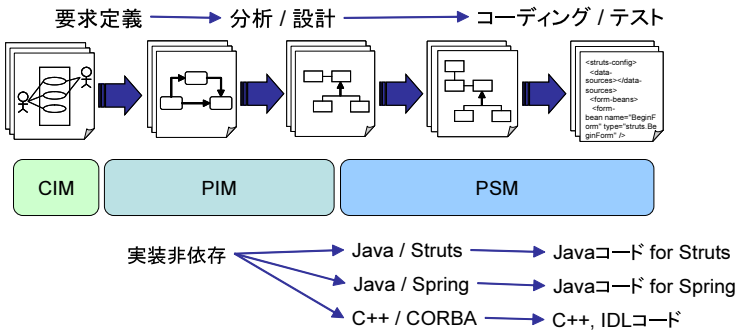

図 2: MDA におけるソフトウェア開発の流れ

例えば，図 2 で表されるように，実装に依存しない設計モデルを作成しておくことで，たとえ開発環境が Java/Spring から Java/Struts に変更したとしても，MDA 対応ツールのモデル変換によって柔軟に対応することができる．また，プログラミング言語もモデルの一種と考えられるため，実装依存モデルから実装コードを生成する際にも MDA 対応ツールの支援を受けることができる．

このように，モデル駆動型のソフトウェア開発では “モデル” をソフトウェア開発の重要な生産物と位置づけ，かつモデルに対する高度なプログラム処理を実現する．その結果，プラットフォームに依存しないモデルの再利用や，異なるプラットフォームへの迅速な対応，開発者の手を介さないモデルの自動処理などの実現を目指している．

2.2 Web アプリケーションフレームワーク Struts

ここではモデルを中心とした開発が適用可能なソフトウェア開発の一例として，Web アプリケーションフレームワーク Struts[4] について述べる．また，本研究では，提案手法の適用対象として Struts を用いているため，Struts の詳細についても議論する．

2.2.1 Web アプリケーションフレームワーク

一般に，Web アプリケーションは，ユーザ画面のフォーム情報を入力として受け付け，リクエストに応じた処理を実行し，結果を次の画面として Web ブラウザ上に出力する（図 3）．ユーザに表示する画面はあらかじめ静的に作成されている場合も実行時に動的に作成される場合もあるが，いずれの場合もその数は開発するアプリケーションの規模に比例して多くなる．そのため，Web アプリケーションを効率的に開発するためのアプリケーションフレームワークが用いられている．Web アプリケーションフレームワークは，画面間の遷移を効率的

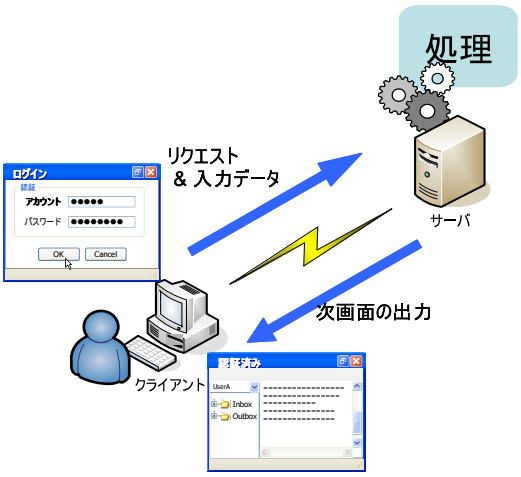

図 3: Web アプリケーションの処理の流れ

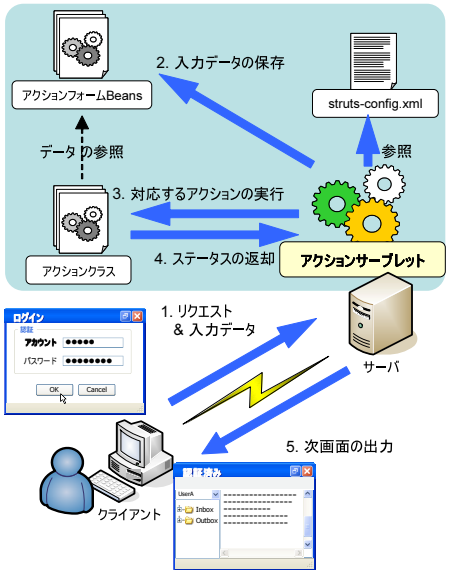

図 4: Struts における処理の流れ

に管理し，Web アプリケーション全般に共通する処理を受け持つことで，開発者が固有部分の開発のみに専念できるようにする．

2.2.2 Struts 概要

Struts[4] は Java 言語を用いた Web アプリケーション開発を支援するためのフレームワークである．Struts を用いた Web アプリケーションでは，アクションサーブレットと呼ばれるコアエンジンがフレームワークに内蔵され，画面遷移を含むアプリケーションの動作全体を制御する．Struts はアプリケーションの機能を次のようなコンポーネントに分離させ，それらをアクションサーブレットが統合する形で 1 つの Web アプリケーションを構成する．

- ユーザの入出力画面を表示する JSP ファイルや HTML ファイル
- ユーザがフォームに入力した情報を格納するクラス（アクションフォーム Beans）
- アクションフォーム Beans を参照し，ユーザのリクエストに応じた処理を実行するクラス（アクションクラス）

アクションサーブレットは，これらのコンポーネントの結合方法を定義した設定ファイルとして，struts-config.xml を参照することで動作する．

Struts を用いた Web アプリケーションの大まかな処理の流れは図 4 のようになる．まず，ユーザからのリクエストを受け付けると，アクションサーブレットは，設定ファイル struts-config.xml を参照し，ユーザの入力情報を格納するクラス（アクションフォーム Beans）を決定してデータを保存する．次に，同様に struts-config.xml を参照することで，アクション

サーブレットはリクエストを処理するためのアクションクラスを特定し，これを呼び出す．呼び出されたアクションクラスは，アクションフォーム Beans を参照することでユーザの入力情報を取得し，目的の処理を行って，結果をステータスとして返す．最後に，アクションサーブレットはアクションクラスから返されるステータスをもとに，struts-config.xml を参照して次の遷移先を決定し，ユーザ画面として表示する．

つまり，開発者は，表示画面となる JSP ファイル，ユーザの入力情報を保存するアクションフォーム Beans，リクエストを処理するアクションクラスといった個々のコンポーネントを作成し，最後にアクションサーブレットの動作を決める struts-config.xml を適切に設定することで，Web アプリケーションを開発することができる．

2.2.3 Struts における成果物

Struts を用いた Web アプリケーションでは struts-config.xml と呼ばれる設定ファイルが個々のコンポーネントを結合する．これは図 5 で表されるような XML ファイルとして記述され，以下の設定項目により画面遷移の定義を行う．

- `<form-bean>`タグ
  アプリケーション中で利用されるアクションフォーム Beans の名前 `name` と実際のクラス名 `type` を対応付ける．
- `<action>`タグ
  リクエストを受けるパス `path` とそのリクエストを処理するアクションクラス `type` を対応付ける．入力フォームのデータが渡される場合，そのデータを格納するアクションフォーム Beans 名 `name` とスコープ `scope` を設定する．また，アクションクラスによる処理を介さずそのまま次画面へ遷移させるには `forward` 属性を設定する．
- `<forward>`タグ
  アクションクラスが処理後に返すステータス `name` と遷移先 `path` を対応付ける．

例えば，図 5 は 3 つの画面（スタート画面 `welcome.jsp`，ログイン画面 `logon.jsp`，認証済み画面 `mainMenu.jsp`）の存在を仮定し，これらの画面間の遷移を表現している．`<form-bean>`は `LogonForm` という名前のアクションフォーム Beans を 1 つ用意する．1 つ目の`<action>`は，パス `/Logon` からログイン画面への無条件の遷移を表す．また，2 つ目の`<action>`は，パス `/SubmitLogon` に対する遷移先がユーザの入力情報とアクションクラスの処理結果に依存して決定され，結果が `success` ならば認証済み画面へ，`failure` ならばログイン画面へ振り分けられることを意味する．なお，あるパスへのリクエストがどの画面へ遷移するかは struts-config.xml 中で定義されるが，どの画面がどのパスへリクエストを送るかは画面を表す JSP ファイルへ記述され，struts-config.xml 中には記述されない．

```
<struts-config>
...
<form-beans>  <!--アクションフォーム Beans の設定-->
  <form-bean name="LogonForm"
             type="org.example.LogonForm" />
</form-beans>
...
<action-mappings>  <!--アクションクラスの設定-->
  <action path="/Logon" forward="/logon.jsp" />
  <action path="/SubmitLogon"
          type="org.example.LogonAction"
          name="LogonForm"
          scope="request" />
    <forward name="success" path="/mainMenu.jsp"/>
    <forward name="failure" path="/logon.jsp" />
  </action>
  <action path="/Logoff"
          type="org.example.LogoffAction">
    <forward name="success" path="/welcome.jsp"/>
  </action>
</action-mappings>
...
</struts-config>
```

図 5: struts-config.xml の例

このように，Struts を用いた Web アプリケーションでは画面から画面へ直接遷移させるのではなく，一度アクションを処理するノードを経由してから次画面への遷移が行われる．そのため，画面遷移を設計する際には最終的に struts-config.xml として記述されることを意識した設計が必要となる．例としては，図 6 のような画面遷移図が考えられる．画面遷移図をどのような仕様で作成するかは Struts によって定められているわけではないが，Struts を用いたソフトウェア開発では画面遷移図の設計が struts-config.xml の仕様，すなわちフレームワークの仕様に強く依存する．そのため，開発者が要求される仕様を満たすような画面遷移図を設計していることが要求される．

2.3 モデルを用いたソフトウェア開発の課題

モデル駆動型開発に代表される “ モデル ” 中心のソフトウェア開発手法は，高品質なソフトウェアを短期間で効率的に開発・保守する上で欠かすことのできない技術といえる．その実現に向けて，OMG による各種標準 [25]-[29] の整備が進められている．一方で，モデルの

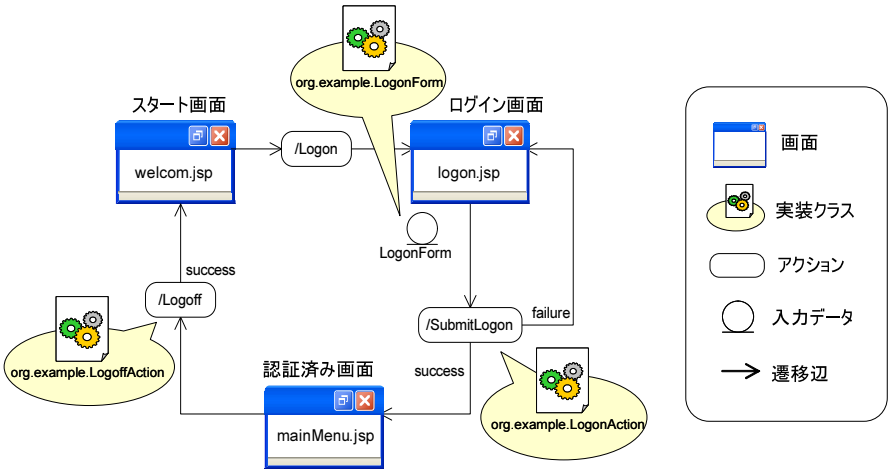

図 6: 図 5 に対応する画面遷移図の一例

自動変換 [11, 12, 18] や検証方法 [8, 21]，マッピング手法 [1, 6, 9, 15] などに関する研究が現在も広く行われており，解決すべき課題となっている．

なかでも，開発工程を通していかに成果物であるモデルを管理し，それら成果物間の依存関係を追跡可能な状態で管理するのかという点が，基本的な課題となっている．開発工程を通してモデルを中心にソフトウェアを開発するとはいえ，各工程で作成されるべきモデルの仕様は異なるのが一般的である．これは開発工程によって異なる場合もあるし，組織や対象業務，フレームワークなどによって異なる場合もある．そのため，各成果物が要求される仕様を満たす状態で作成されていることを検証可能な管理手法でなければならない．また，成果物と同様に，成果物間の依存関係についてもその情報を蓄積しなければならない．多くの成果物が互いに依存する形で作成されるため，一方の成果物の修正は他方の成果物の修正要求につながる．このような成果物を一貫した状態で維持するために，互いに依存関係をもつ成果物間の情報を管理できることが不可欠となっている．

2.4 関連研究

Akehurst ら [1] はモデル間の依存関係の特徴（Relation）と実際の依存情報（Pairs）を分離することによって，依存関係に対する制約を記述可能にし，OCL[28] を用いることで変換規則を記述する手法を提案している．彼らはモデル変換に主眼をおいた手法の提案を行っているが，本研究では成果物を管理するという枠組みの中で依存情報も同様に管理し，実際に動作するシステムとして確認することを目指している．また，Hausmann[15] らは同手法に対して，開発者が理解しやすいマッピング可視化手法について研究している．

Bondé ら [6] はモデル変換エンジン ModTransf[12] を用いることによって，モデル間のマッピングを記述する研究を行っている．彼らの研究は，実装非依存のモデル（PIM）から異なる 2 つの実装依存モデル（PSM）を生成した場合に，それぞれに適用したモデル変換ルール（PIM から PSM へのマッピングとして記述される）を解析することで，PSM 間の相互運用性を確保することが目的となっている．そのため，モデル間の依存関係を蓄積が主眼となっている本研究とは異なる．

また，OMG からは QVT[27] というモデル変換に関する標準が策定されている．自動変換可能なモデル間にはそのままマッピングの保存が可能であると考えられるが，必ずしもあらゆる場合にモデルの自動変換が可能なわけではない．本研究ではモデルの自動変換に焦点をあてるのではなく，成果物とその依存関係を MOF メタモデルのもとで統一的に管理する枠組みの実現を目指している．

3　メタモデルと MOF

本節では，ソフトウェア開発においてメタモデルを定義する必要性について述べ，本研究でメタモデリング言語として着目した MOF について紹介する．

3.1　メタモデルの重要性

ソフトウェア開発で扱われる情報は多様で，個々の情報はあらかじめ定められた記述仕様に従うモデルとして表現される．しかし，仕様の中には自然言語で定義されたものや，必要以上に条件の緩いものが存在し，本来意図した内容よりも曖昧なモデルの作成が可能になる．そのため，作成したモデルが，管理したい情報を正しくかつ厳密に表現できていることを保証できないという問題がある．

そこで，モデルによって表現される意味論を形式的に定義し，意味解釈の曖昧さを取り除くためのメタモデルが必要となる．メタモデルは，モデルとして記述すべき，あるいは記述すべきでない内容を定義し，作成されたモデルが意図した情報を正しく表現していることを保証する．つまり，管理する情報に対する仕様としてメタモデルを定義することで，実際に作成されたモデルの妥当性を検証できる．

3.2　MOF の概要

MOF（Meta Object Facility）[23, 26] とは，メタモデルおよびそのインスタンスであるモデルを定義・操作するための技術として，OMG（Object Management Group）[22] が定める標準的な枠組みである．MOF に関して以下の仕様が定義されている．

- MOF Model
  メタモデルを定義するための言語（メタモデリング言語）としての MOF を構成する各要素の定義.
- Abstract Mapping
  MOF 準拠のメタモデルとそのインスタンスであるモデルとの意味論のマッピングに関する定義.
- MOF to IDL Mapping
  MOF 準拠のメタモデルから CORBA IDL（Common Object Request Broker Interface）を生成するためのマッピングに関する定義．メタモデルのインスタンスであるモデルを管理するためのリポジトリに対する標準手続きを規定.

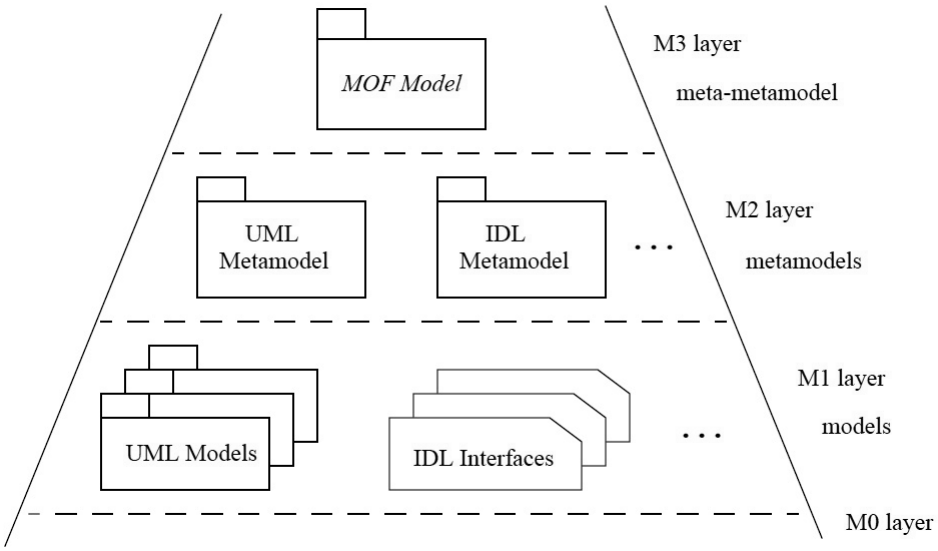

図 7: MOF アーキテクチャレイヤー（出典：MOF Specification v1.4 [23]）

- MOF to XML Mapping
  MOF 準拠のメタモデルから XML（eXtensible Markup Language）の DTD（Document Type Definition）を生成するためのマッピングに関する定義．XML によるモデル情報の相互交換は XMI（XML-based Metadata Interchange）[25] として標準化されている．

メタモデリング言語としての MOF を論じる場合，図 7 のような 4 層階層（M0〜M3）によってデータとメタデータの関係を表現することが多い．

- M0 階層はモデル化したい対象の事物や情報（データ）で構成される．
- M1 階層は M0 階層のデータの構造を規定するメタデータで構成される．メタデータの集合がモデルと呼ばれる．開発者が UML のようなモデリング言語を用いて作成するダイアグラムはモデルに相当する．
- M2 階層は M1 階層のメタデータの構造を規定するメタメタデータで構成される．モデルの構造，意味を規定するため，メタモデルと呼ばれる．UML の意味論やその他のモデリング言語の意味論の定義に相当する．
- M3 階層はメタメタデータの構造を規定する．様々なモデリング言語の意味論を定義するメタモデリング言語の意味定義に相当する．MOF は M3 階層に相当し，自己記述可能な定義となっているため，M4 階層以上は存在しない．

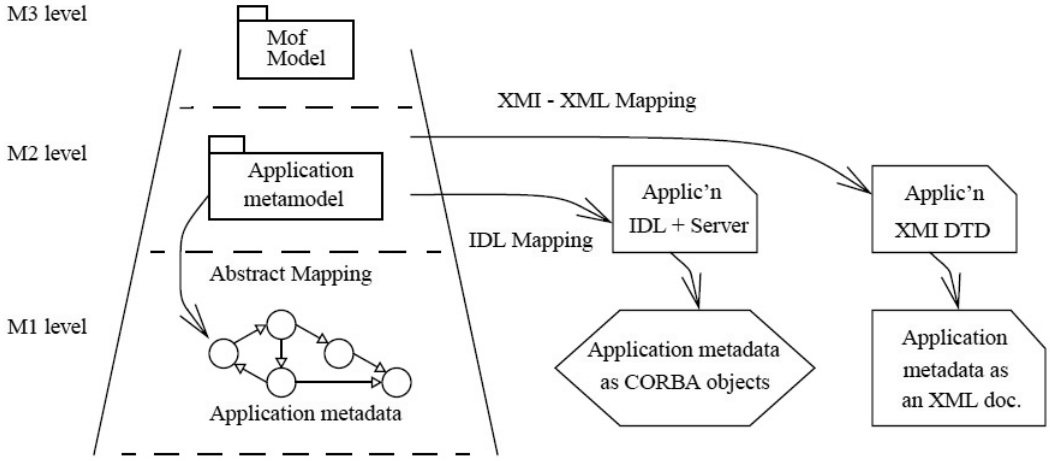

図 8: MOF アーキテクチャレイヤー（出典：MOF Specification v1.4 [23]）

モデリング言語である UML が共通の特徴をもつオブジェクト（M0）を UML クラス（M1）として抽象化するのに対し，メタモデリング言語である MOF は共通の特徴をもつメタオブジェクト（M1）を MOF クラス（M2）として抽象化する．つまり，MOF でいうところのクラスは，UML でいうところのメタクラスに相当しているという点に注意が必要である．

さらに，MOF を用いて記述したメタモデル（M2）からマッピング規則によって CORBA IDL や XML DTD を生成し，対応するモデル（M1）を CORBA オブジェクトや XML 文書にマッピングすることが可能となっている（図 8 参照）．

3.3 MOF の構成要素

ここでは，メタモデリング言語としての MOF に着目し，その基本的な構成要素について説明する．MOF は UML のクラス図に似た意味論をもつが，その表記法自体は定義されていないため，UML クラス図を用いて記述されることが多い．ここでも UML のクラス図を用いることで，MOF の構成要素を簡単に紹介する．

MOF を構成する要素を概観すると，図 9 のようになっている．これらのクラス（M3）が MOF を構成し，そのインスタンス（M2）がメタモデルを構成する．つまり，MOF によるメタモデリングとは図 9 のクラス群のインスタンスによってメタモデルを記述することである．例えば，UML のメタモデル，すなわち言語定義もこれらのクラス群のインスタンスとして定められている．ここでは，これらの要素のうち特に以下の主要概念と関係のある部分について順に説明していく．

- クラス

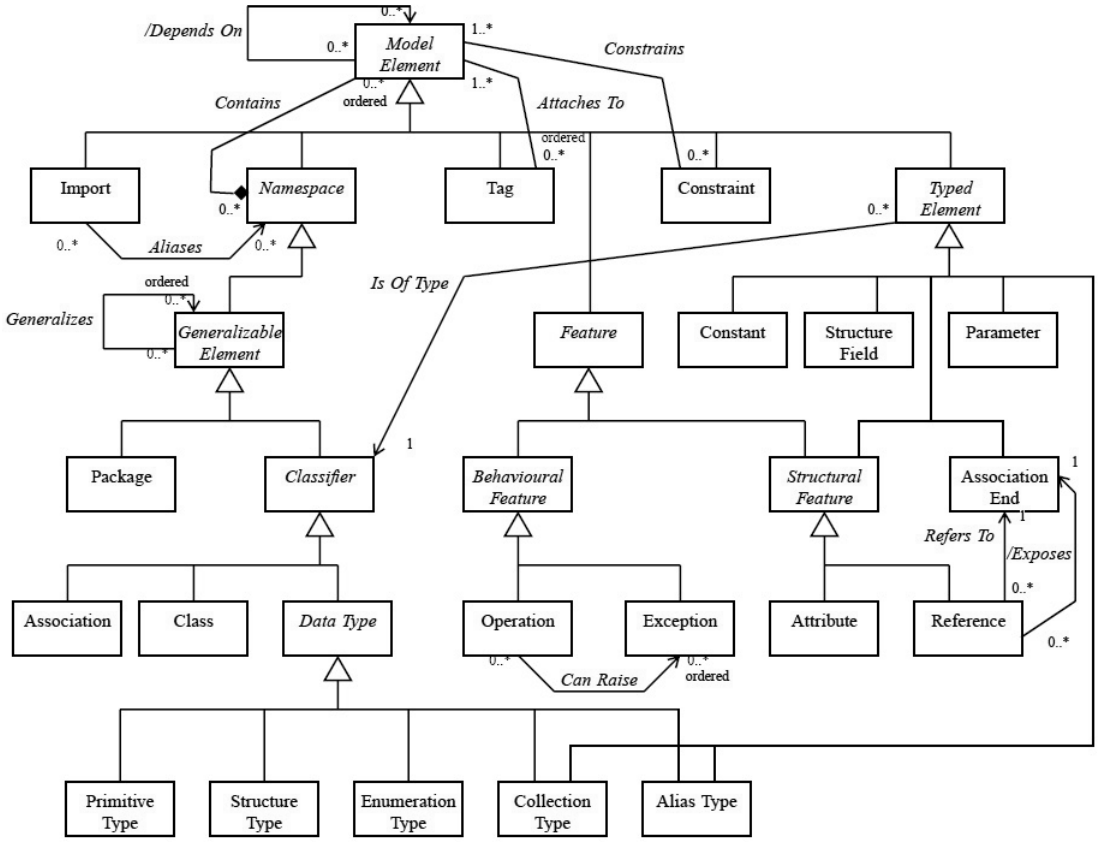

図 9: MOF Model（出典：MOF Specification v1.4 [23]）

- 関連
- データ型

なお，ここでいうクラスや関連はいずれも MOF におけるクラスや関連であって，UML のものとは異なるため，混同しないよう注意が必要である．また，実際の仕様においては，さらに OCL や自然言語を用いて意味論が詳細に規定されているが，ここで全てを説明することはできないため省略する．詳細な定義に関しては，MOF の仕様書 [23, 26] を参照のこと．

クラス

クラスの定義を説明する前に，MOF の中心をなす抽象クラス群について説明する．図 10 で表されるように，いくつかの抽象クラス群が定義されており，それぞれ以下のような特徴をもつ．

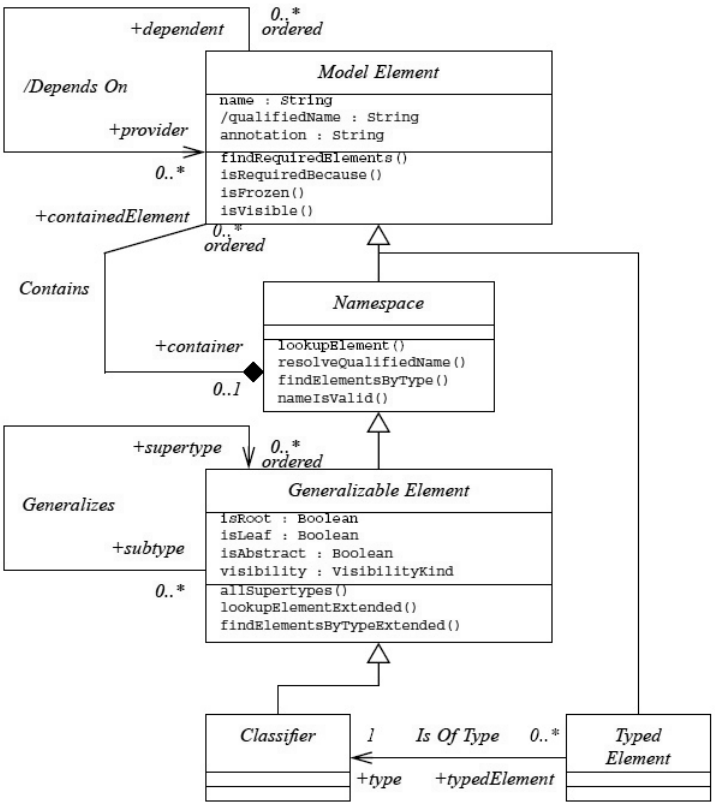

図 10: MOF の基本となる抽象クラス群（出典：MOF Specification v1.4 [23]）

- ModelElement
  MOF Model における最上位要素.
- Namespace
  名前空間を構成し，他の要素を保持可能な要素.
- GeneralizableElement
  汎化・特化が可能な要素．特化された要素は親要素の特徴を引き継ぐ.
- Classifier
  共通の性質をもつインスタンスの分類を提供する要素.
- TypedElement
  型をもつ要素．型の保持は Classifier との関連として定義される.

クラス Class は Classifier のサブクラスであり，かつインスタンス化可能な具象クラスとして定義されている（図 11）．クラスはメタモデルを記述する際に最も基本的な要素となり，

構造的な特徴としての「属性」や，振る舞いの特徴としての「操作」をもつ．これらの特徴は，図 12 で表される Feature のサブクラスとして定義されている．Class は Namespace のサブクラスであるため，他の要素を保持可能となっており，自身が分類しようとしている特徴を Feature として規定する．

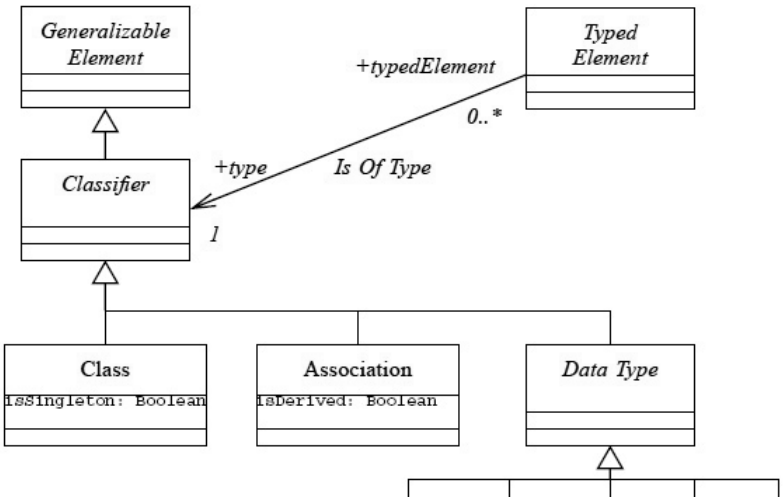

図 11: Class クラス（出典：MOF Specification v1.4 [23]）

Feature は構造的な特徴を表す StructuralFeature と，振る舞いの特徴を表す BehavioralFeature に分類される．構造的な特徴には属性 Attribute や参照 Reference が，振る舞いの特徴には操作 Operation や例外 Exception がある．

関連

関連 Association は，2 つのクラスインスタンス間の関係をモデル化するために用いられる．Association は Classifier のサブクラスとして定義され，関連付ける 2 つのインスタンスは AssociationEnd として定義される（図 13）．そして，Association が AssociationEnd を保持する形で関連をモデル化している．AssociationEnd は TypedElement のサブクラスとなっており，接続されるクラスインスタンスの型 Classifier を保持している．

UML では “関連クラス” のような特別な種類の関連も存在するが，MOF では純粋にオブジェクト間の 1 対 1 の関係をモデル化する．

データ型

データ型はメタモデリングに利用可能な型を規定する．図 14 で表されるように，データ型は抽象クラス DataType としてモデル化されている．データ型は，基本型を表す PrimitiveType

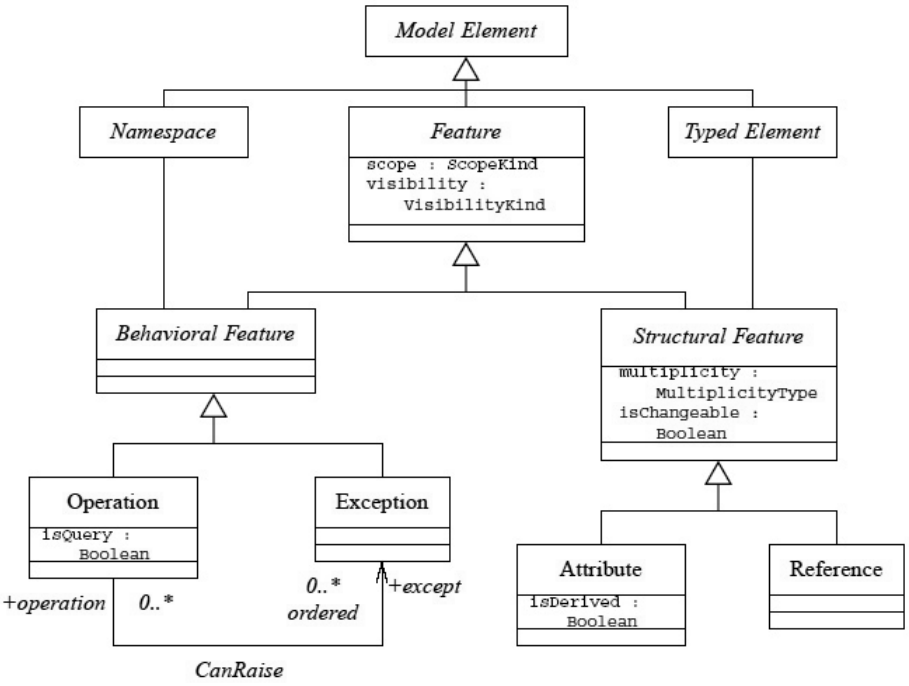

図 12: Feature クラス（出典：MOF Specification v1.4 [23]）

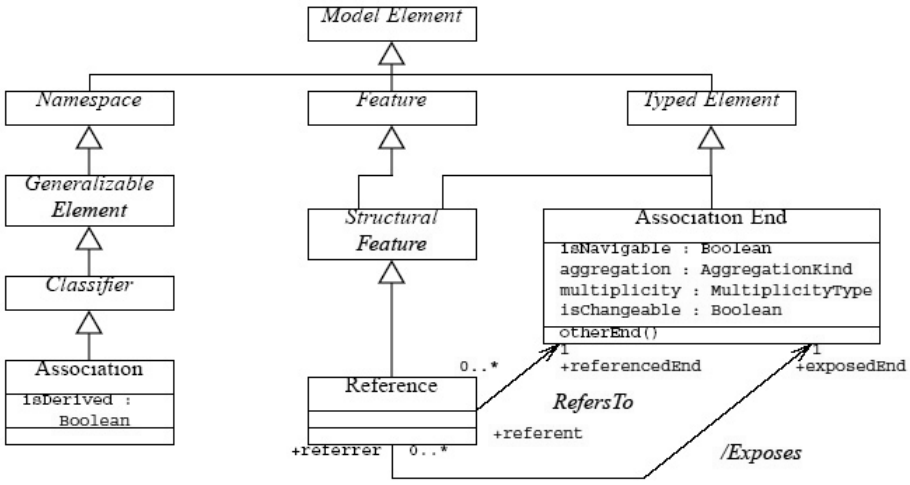

図 13: 関連（出典：MOF Specification v1.4 [23]）

や構造型を表す StructureType，列挙型を表す EnumerationType などによって詳細化される．また，PrimitiveType のインスタンスとして，ブール型 Boolean，文字列型 String，整数型 Integer などがあらかじめ定義されており，MOF によるメタモデルを記述する際には利用可能である．

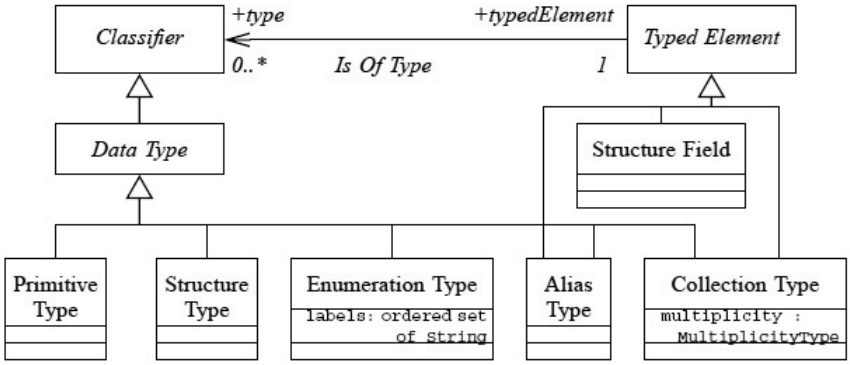

図 14: データ型（出典：MOF Specification v1.4 [23]）

3.4 MOF リポジトリ

MOF によって記述したメタモデル（以降，MOF メタモデル）に対して，そのインスタンスであるモデルを蓄積・管理するためのリポジトリを MOF リポジトリと呼ぶ．MOF リポジトリにアクセスするための API 群は，リポジトリに格納するモデルのデータ構造にのみ依存するため，MOF メタモデルが与えられればその API は一意に決定可能である．

MOF to IDL Mapping では，与えられた MOF メタモデルから CORBA IDL を生成する規則を規定している．開発者が定義した MOF メタモデルから，そのインスタンスを管理するためのデータ構造が作られる簡単な例を図 15 に示す．

図 15 の例は，パッケージ P とクラス C，関連 A からなる単純な MOF メタモデル（図中左）から，そのインスタンスを管理するためのデータ構造（図中中央と右）が作られる様子を表している．図中中央で表されるように，MOF メタモデルは Factory パターンと Proxy パターンを用いて 5 種類のメタオブジェクトにマッピングされ，これらメタオブジェクト間の関連は図中右のようになると定義されている．つまり，

- パッケージ P は，パッケージ P そのものを表すメタオブジェクト P と，P を生成するファクトリ P_Factory にマッピング
- 関連 A は，関連そのものを表すメタオブジェクト A にマッピング

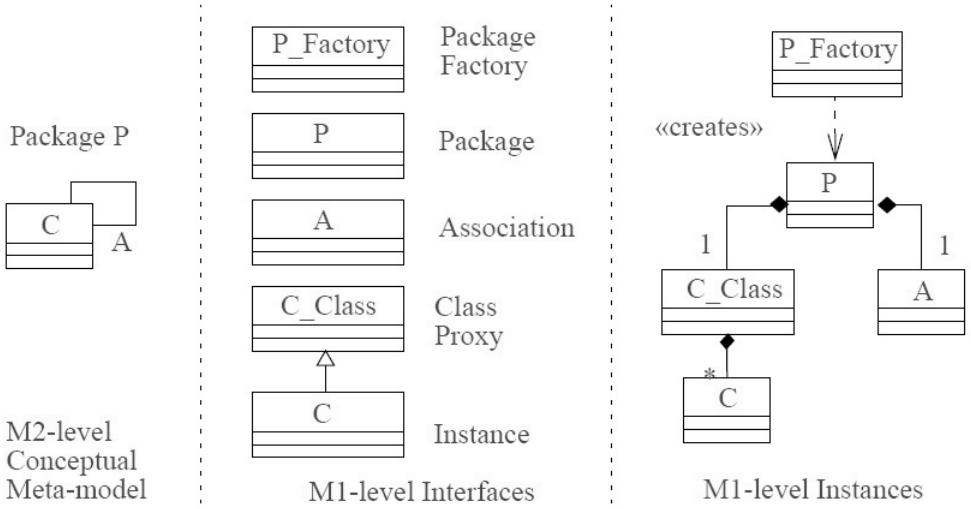

図 15: メタモデルのインスタンスを管理するデータ構造（出典：MOF Specification v1.4 [23]）

- クラス C は，クラス C そのものを表すプロキシ C_Class と，クラス C のインスタンスを表すメタオブジェクト C にマッピング

することによって，MOF メタモデルのインスタンスを管理可能なリポジトリデータ構造を実現する．なお，MOF の仕様ではメタモデルからインタフェースへのマッピングは規定しているが，その実装に関しては定義していない．MOF to IDL Mapping を実装し，MOF リポジトリを生成可能なプログラムとしては Medini[16] がある．また，MOF から Java へのマッピングである JMI（Java Metadata Interface）[17] という標準規則も提唱されており，その実装の一つに MDR[20] がある．

MOF リポジトリがあらかじめ定義されたメタモデルのインスタンスのみを管理可能であることを利用すると，仕様に従った成果物の作成を開発者に強要することができる．図 16 で表されるように，ある成果物の仕様として MOF メタモデルを定義することで，その MOF メタモデルに従ったモデルのみを管理可能なリポジトリをあらかじめ生成することができる．これにより，開発者がメタモデルに従わない誤ったモデルを成果物として作成した場合や，あるいは修正によってメタモデルを満たせなくなってしまった場合にも検出が可能となり，設計ミスを防止できるようになる．

例えば，図 16 中の開発者 A，B が成果物として作成したモデル（要求定義書なり詳細設計書なり）は，あらかじめ定義されたメタモデルの正しいインスタンスとなっているため，リポジトリで管理することができる．しかし，開発者 C が作成したモデルは，MOF オブジェクト A1，C1 の間に関連が記述されており，これはメタモデル中で MOF クラス A，C 間に

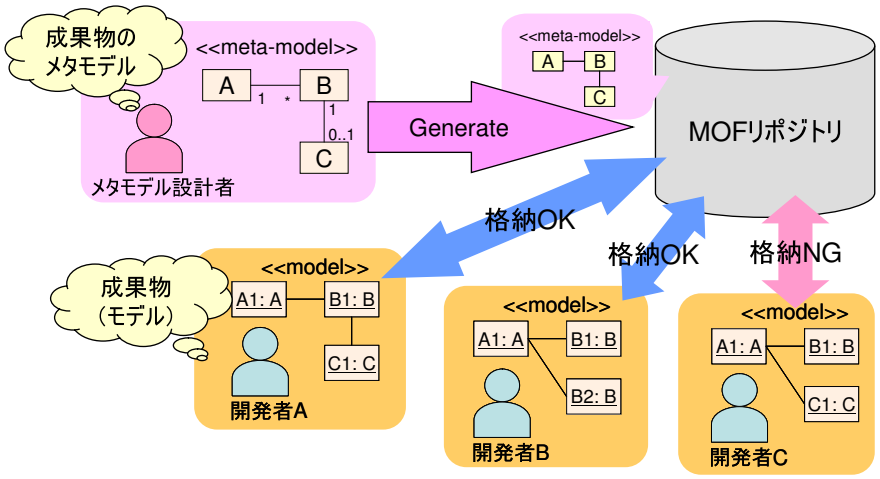

図 16: MOF リポジトリの生成とモデルの管理

は関連が定義されないという仕様に矛盾する．つまり，開発者 C が成果物として作成したモデルは，期待されている仕様を満たさない誤ったモデルであるということになる．

4　提案手法

本節では，メタモデル記述言語として MOF に着目し，メタモデルを用いることによって成果物間の依存関係を追跡可能にする手法を述べる．

4.1　提案手法の概要

提案手法は，以下の手順によって，成果物とその依存関係を MOF リポジトリで管理可能とする．そして，これら MOF リポジトリを操作することによって，成果物の依存関係を追跡可能な状態とする．

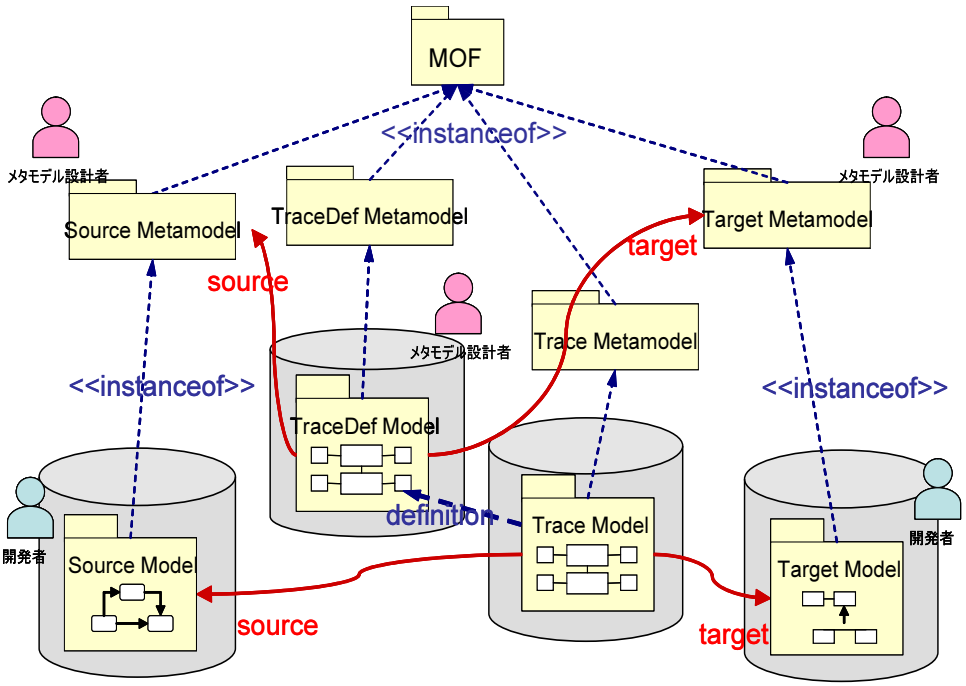

図 17: 概観図

1. 互いに依存関係にある二つの成果物の仕様を MOF メタモデルで定義し，開発者が作成したモデルを MOF リポジトリで管理可能とする
    - 依存元となる成果物のメタモデル（図 17 Source Metamodel）と，開発者が実際に作成するモデル（図 17 Source Model）
    - 依存先となる成果物のメタモデル（図 17 Target Metamodel）と，開発者が実際に作成するモデル（図 17 Target Model）

2. 依存関係の定義，および蓄積を行うための仕様を MOF メタモデルで定義し，依存関係定義・依存情報ともに MOF リポジトリで管理可能とする

    - 依存関係を定義するためのメタモデル（図 17 TraceDef Metamodel）と，依存関係定義（図 17 TraceDef Model）
    - 依存関係を蓄積するためのメタモデル（図 17 Trace Metamodel）と，実際に保存された依存情報（図 17 Trace Model）

これらのモデル群のうち，依存情報を定義するためのメタモデル（TraceDef Metamodel），依存情報を蓄積するためのメタモデル（Trace Metamodel）は提案手法の中で定義する．また，依存元・依存先の成果物のメタモデル（Source Metamodel，Target Metamodel），成果物間の依存関係定義（TraceDef Model）は提案手法の適用対象にあわせてあらかじめ定義される必要がある．成果物のモデル（Source Model，Target Model）と依存情報（Trace Model）は，開発者の実際のモデリング操作に合わせて，提案手法を実現する支援ツールが蓄積する．

4.2 メタモデルを用いた成果物の管理

作成されるべき個々の成果物に対して，望まれる仕様をあらかじめ MOF メタモデルとして定義する．メタモデルの定義は，MOF の意味論に従って図 9 のインスタンスとして記述する．3.4 節で述べたように，定義した MOF メタモデルから，そのインスタンスであるモデルを管理する MOF リポジトリが生成可能である．このように生成した MOF リポジトリを用いることで，開発者が作成するモデルの管理を行うことができる．

図 18 のように，あらかじめ成果物のメタモデルが定められているとする．このとき，開発者がモデリングツールを通して実際に作成したモデルと，その成果物の仕様であるメタモデルを比較する．これにより，成果物のモデルが意図された仕様を満たす状態で作成されているか検証することができる（妥当性検証）．

具体的に成果物に対して検出可能なメタモデル違反には，以下のようなものが挙げられる．

- メタモデルで多重度が 1 以上に設定された属性や関連が，モデル中には存在しない
- 必須な属性に対して空文字や null 値が与えられている
- メタモデルで定義された多重度の上限を超えて属性や関連が作られている
- メタモデルで定義されないオブジェクトや属性，関連が作られている
- メタモデルで定義された属性の型と，モデルでの値が一致していない

このように，開発者が成果物を作成する際に使用可能なモデル要素や制約をメタモデルとして定義することで，実際にメタモデル違反が発生した場合には検出が可能となる．メタモ

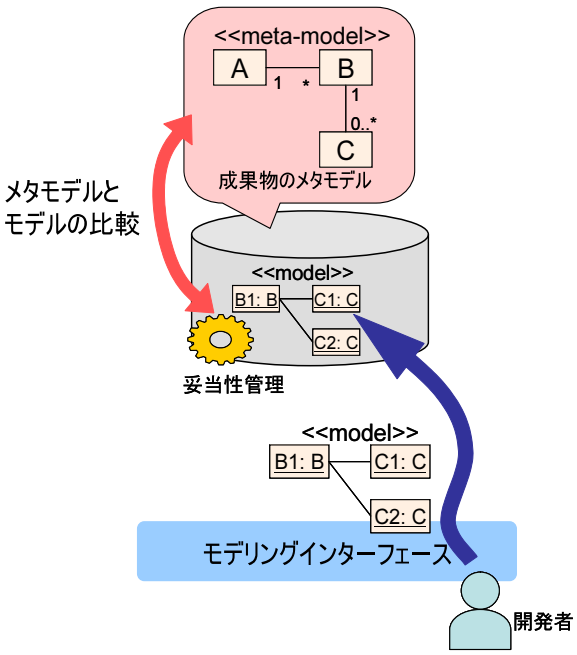

図 18: 成果物の妥当性管理

デルを定義した上で成果物を管理するアプローチは，UML を用いて自由にモデルを作成できる場合よりも，厳密なモデルの作成を開発者に義務づけることができる．なお，4.4 節で述べるが，モデルの妥当性検証は更新が発生した MOF オブジェクトに対して行われる．

本手法は，ある成果物が与えられた場合に，その仕様をメタモデルとして定義することでモデルを厳密に管理する，という枠組み自体を提案する．そのため，提案手法は特定の成果物に依存しない汎用的なものである．成果物のメタモデルは，実際の適用対象に応じて定められるべきであり，Struts の成果物に対して適用した例は 4.5 節に示す．

4.3 メタモデルを用いた依存関係の管理

成果物間の依存関係も成果物と同様の枠組みの中で管理する．依存関係の管理にあたっては，以下の 2 つの MOF リポジトリを用意する．

- 成果物間の依存関係の定義に関する MOF リポジトリ．対象の成果物のメタモデルにあわせてあらかじめ依存関係定義モデルを作成し格納しておく（図 17 TraceDef Metamodel - TraceDef Model）．
- 成果物間の依存情報を蓄積する MOF リポジトリ．依存関係定義にもとづいて実際の依存情報が保存されていく（図 17 Trace Metamodel - Trace Model）．

成果物間の依存関係定義は図 19 のメタモデル（図 17 TraceDef Metamodel 相当）に従うインスタンス（図 17 TraceDef Model 相当）として定義する．ここでは，一方のモデル要素集合の変更が他方のモデル要素集合へ何らかの影響を与えるような，n 対 n の一方向依存関係を定義することを目的としている．

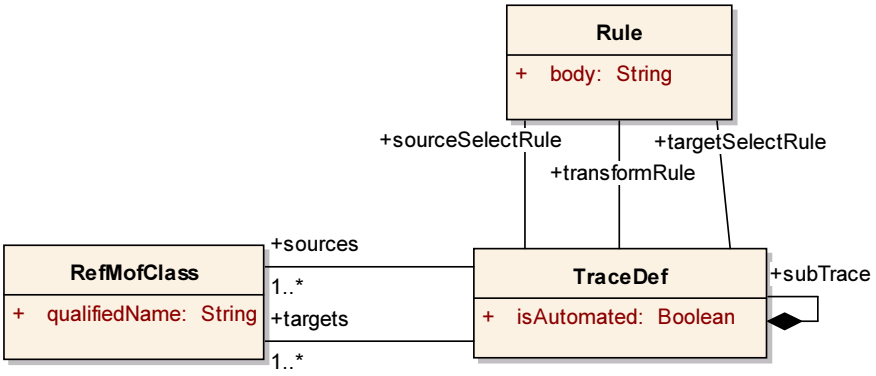

図 19: 依存関係定義のメタモデル（図 17 TraceDef Metamodel）

メタモデル中のクラス RefMofClass は，完全修飾子名を用いて成果物メタモデル中の MOF クラスを間接参照する．依存関係定義クラスは TraceDef は，依存関係の入力となる RefMofClass 群と，出力となる RefMofClass 群との間の関係として定義される．特定の依存関係のコンテキスト中でのみ有効な定義を考慮し，依存関係定義はネストできるようにする．また，依存関係定義の入力条件，すなわち RefMofClass によって間接参照している MOF クラス群が満たすべき条件を TraceDef.sourceSelectRule（クラス Rule）として定義するようにしている．また，同様に出力条件を TraceDef.targetSelectRule，入力 MOF クラス群から出力 MOF クラス群への自動変換が可能な場合の変換規則を TraceDef.transformRule として定義するようにしている．なお，これらの規則がプログラムによってハードコーディングされて実現されるのか，インタプリタによって解釈実行されるのかは実装依存の問題である．本研究は，モデルを自動変換することよりも，その関係を保存することを目的としているため，変換規則などの記述形式についても提案手法の枠組みの中では定めない．

依存関係定義にもとづき，実際に作成された成果物間の依存情報を図 20 のメタモデル（図 17 Trace Metamodel 相当）に従うインスタンス（図 17 Trace Model 相当）として蓄積する．依存関係定義が成果物のメタモデル（MOF クラス）間の n 対 n の対応関係をモデル化したのに対し，依存情報は成果物のモデル（MOF オブジェクト）間の n 対 n の対応関係として記述する．

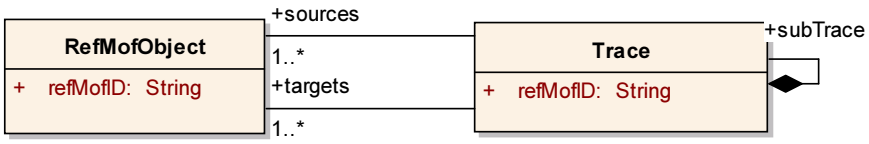

図 20: 依存情報のメタモデル（図 17 Trace Metamodel）

クラス RefMofObject は MofID を用いて MOF オブジェクトを間接参照する．MofID は MOF オブジェクトがもつ固有の識別子 [23] である．また，同様にクラス Trace はその依存関係定義（図 19 中 TraceDef）を MofID によって間接参照する．

開発者が成果物を作成するのにあわせ，依存情報が蓄積される様子を図 21，22 に示す．まずモデルが作成されると，1. その成果物のメタモデルを参照することによって，作成した MOF オブジェクトのクラスが特定される．次に，2. 特定した MOF クラスと関係のある依存関係定義を見つけ出す．この時点で，開発者が作成した MOF オブジェクトに対して，あらかじめ定義されている依存関係定義を取得することができる．さらに，取得したそれぞれの依存関係定義に対して，依存情報の蓄積を行う．ただし，作成したオブジェクト以外にも要求される入力オブジェクトがあるような場合には，必ずしも依存情報の蓄積を行えるわけではない．そのような場合には，入力条件を満たす状態になって初めて依存情報を蓄積可能となる．必要な入力条件を満たしている場合，3. 自動化が可能であるならば出力オブジェクトを変換規則にもとづいて生成し，最後に 4. 入力オブジェクト群と出力オブジェクト群の間に依存情報を蓄積する．自動処理できない場合は，手動で入力オブジェクト群と出力オブジェクト群を指定することで依存情報を蓄積することもできる．

ここでは作成の場合の流れを簡単に説明したが，モデルに変更が行われた場合にも基本的に流れは同じである．変更対象のモデルに対して，メタモデルを参照することでどのような依存関係定義が存在するのか調べることができる．また，実際に蓄積されている依存情報についても依存情報リポジトリから調べることができ，その結果依存関係にある別のモデルを取得することができる，このように依存情報を蓄積することで，あるオブジェクトの変更が他のモデルのどのオブジェクトに影響を与えるかを調べることができるようになる．

4.4 モデルに対する変更と情報の更新

ここでは，開発者がモデルを編集する際に，MOF リポジトリ中のモデルを適切に更新する方法について述べる．MOF メタモデル（MOF クラス）のインスタンスであるモデル（MOF オブジェクトから構成される）に対して開発者が加える操作は，大きく次のように分類される．

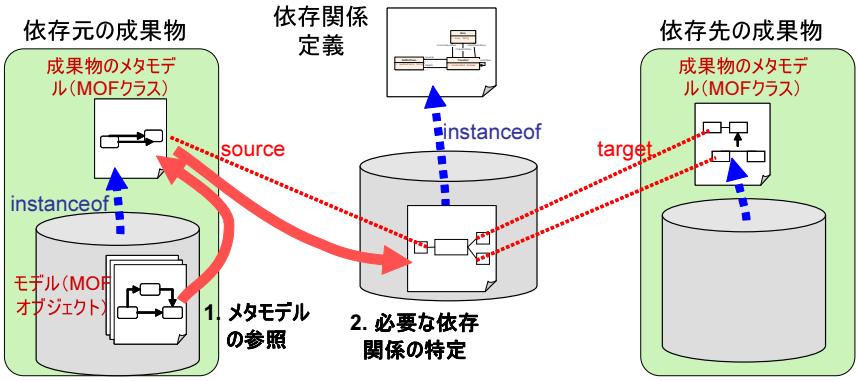

図 21: 依存情報が蓄積される流れ（依存関係定義の特定）

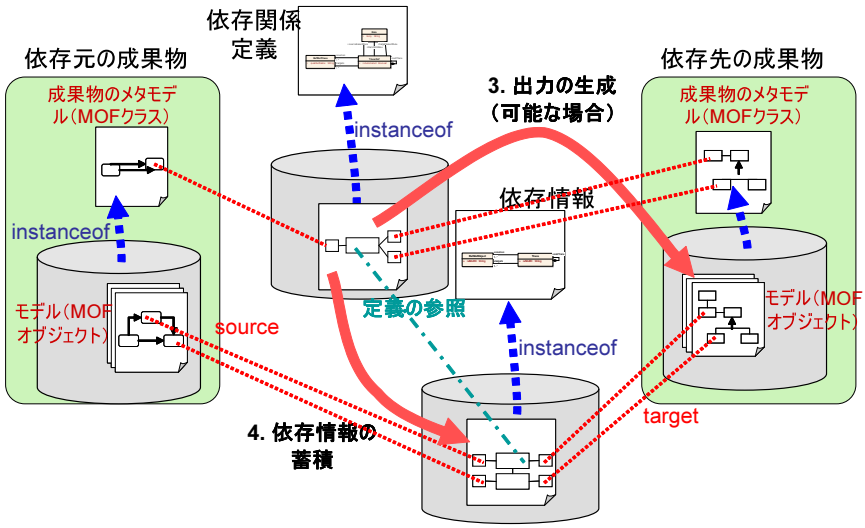

図 22: 依存情報が蓄積される流れ（依存情報の蓄積）

- オブジェクトの追加，修正，削除
- 関連の追加，修正，削除
- 属性の追加，修正，削除

以降では，これらの項目について順に説明する．各操作に対して，トリガとなるイベント，入力，実際の処理，その結果起こりうるメタモデル違反のうちその操作に特徴的なものを C#風の疑似プログラミング言語を用いて説明する．ここで考慮するメタモデル違反は表 1 で挙げるものとする．また，C#風の疑似言語中で用いるオブジェクトのうち他のオブジェクトへの参照をもつものは表 2，リポジトリに対する操作は表 3 で挙げるものとする．

なお，少なくとも操作対象のオブジェクトや属性はメタモデルで定義されていることを前提とし，メタモデルで定義されていない要素を操作しようとしたことによる違反は省略す

表 1: メタモデル違反の種類

メタモデル違反の種類	説明
ATTRIBUTE_REQUIRED	作成を要求される属性が存在する
ATTRIBUTE_REFERS_NON_EXIST_OBJ	属性が存在しないオブジェクトを参照する
ATTRIBUTE_EXCEEDS_UPPERBOUND	属性が上限値を超えて存在する
ASSOCIATION_REQUIRED	作成を要求される関連が存在する
ASSOCIATION_EXCEEDS_UPPERBOUND	関連が上限値を超えて存在する
TRACE_REQUIRED	作成を要求される依存情報が存在する
TRACE_INCONSISTENCY	既存の依存情報に矛盾が発生する

表 2: 疑似言語中で扱うオブジェクト参照

オブジェクトの種類	説明
product.repository	ある成果物 product に対する MOF リポジトリ repository
object.repository	あるオブジェクト object が格納されている MOF リポジトリ repository
object.referedObjects	オブジェクト object を参照しているオブジェクトの集合．具体的には，object と関連をもっているか，自身のもつ属性が object を参照しているようなオブジェクトの集合.
object.traces	オブジェクト object と関連付けらている依存情報の集合

る．依存関係情報に矛盾が生じるという違反（TRACE_INCONSISTENCY）はモデルを操作した場合には常に起こりうるため，これも明示的には記述しない．起こりうる違反を事前に検出して開発者の編集操作を中止させるか，あるいは検出後にフィードバックを返すといった何らかの対処を行うかは実装の問題であるため，ここでは議論しない．

4.4.1 オブジェクト

オブジェクトの追加

MOF オブジェクトの追加は，対象成果物のメタモデル中で定義された MOF クラスのインスタンスを作成し，リポジトリに格納することで行われる．MOF オブジェクトを新たに作成したことにより，そのオブジェクトが必須とする属性や関連の作成が要求される．また，依存して作成されるべき要素や作成されているべき要素との依存情報が要求される．

表 3: 疑似言語中で扱うリポジトリ repository に対する操作

リポジトリに対する操作	説明
addObject(class)	クラス class のインスタンス（オブジェクト）を生成した上でリポジトリに追加する.
removeObject(object)	リポジトリからオブジェクト object を削除する.
addAttribute(object, attribute, value)	リポジトリ中のオブジェクト object に対して, 属性 attribute として値 value を追加する.
updateAttribute(object, attribute, value)	リポジトリ中のオブジェクト object に対して, 属性 attribute の新たな値として value を設定する.
removeAttribute(object, attribute)	リポジトリ中のオブジェクト object に対して, 属性 attribute を削除する.
addAssociation(source, target, assoc)	オブジェクト source からオブジェクト target への関連 assoc を追加する.
removeAssociation(link)	オブジェクト間のリンク link を削除する.

トリガ:
　開発者による明示，またはシステムによる自動処理
入力:
　対象成果物 `product`
　作成するオブジェクトのクラス `class`
処理:

```
createObject(product, class) {
    object = product.repository.addObject(class)
    updateObject(object) // 「オブジェクトの修正」項を参照
}
```

起こりうる違反:
　`ATTRIBUTE_REQUIRED`
　`ASSOCIATION_REQUIRED`
　`TRACE_REQUIRED`

オブジェクトの修正

オブジェクトの修正は，自身が保有する属性や関連の更新によって，状態が変化した可能性があるときに引き起こされる．属性や関連に対する操作，あるいはオブジェクトの追加,

削除といった操作も，最終的には全てオブジェクトの更新を引き起こす.

オブジェクトが更新された場合には，そのオブジェクトの妥当性検証を行い，かつそのオブジェクトと関連付けられている依存情報に矛盾が生じていないかを検証する必要がある．また，オブジェクトが追加されたり削除されたりしたことによって，依存関係定義にもとづいた新たな依存情報の蓄積が必要となる可能性もある．最後に，実際にオブジェクトの状態が変化していた場合には，関連する他のオブジェクトの更新を再帰的に引き起こす.

トリガ:
　　オブジェクトの状態変化
入力:
　　更新されたオブジェクト object
処理:

```
updateObject(object) {
    validatesObject(object) // オブジェクト object の妥当性検証
    foreach (t in object.traces) {
        validatesTrace(t) // トレース情報に矛盾が生じていないか検証
    }
    addTrace(object) // 新たな依存情報の作成が可能かチェック
    if (自身の状態が変化した場合) {
        foreach (o in object.referedObjects) {
            updateObject(o)
        }
        foreach (t in object.traces) {
            updateTrace(t)
        }
    }
}
```

起こりうる違反:
　　全て

オブジェクトの削除

既存のオブジェクトを削除することによって，そのオブジェクトが保有する属性や関連も削除されることになる．削除処理はMOFリポジトリから指定のオブジェクトを削除するだけでなく，そのオブジェクトが保持している依存情報との関連も削除されなければならない．結果として削除されたオブジェクトを参照していたオブジェクトや依存情報に不整合が生じる可能性がある.

トリガ:
　　開発者による明示，またはシステムによる自動処理
入力:

削除するオブジェクト object

処理:

```
removeObject(object) {
    object.repository.removeObject(object)
    foreach (o in object.referedObjects) {
        updateObject(o)
    }
    foreach (t in object.traces) {
        t.removeObject(object) // 対象の依存情報 t から削除
        updateTrace(t)
    }
}
```

起こりうる違反:

ATTRIBUTE_REFERS_NON_EXIST_OBJ
ASSOCIATION_REQUIRED

4.4.2 属性

属性の追加

属性を追加することによって，間接的に追加先のオブジェクトの更新を引き起こす.

トリガ:

開発者による明示，またはシステムによる自動処理

入力:

属性を追加するオブジェクト object
追加する属性 attribute
追加する値 value

処理:

```
addAttribute(object, attribute, value) {
    object.repository.addAttribute(object, attribute, value)
    updateObject(object)
}
```

起こりうる違反:

ATTRIBUTE_EXCEEDS_UPPERBOUND

属性の修正

属性の値を修正することによって，間接的にその属性を保持しているオブジェクトの更新を引き起こす．結果として，依存情報の不整合が起こりうる.

トリガ:

開発者による明示，またはシステムによる自動処理

入力:

属性を修正するオブジェクト object
修正する属性 attribute
新たな属性値 value
処理:
```
updateAttribute(object, attribute, value) {
    object.repository.updateAttribute(object, attribute, value)
    updateObject(object)
}
```

属性の削除

既存の属性を削除することによって，間接的にその属性を保持していたオブジェクトの更新を引き起こす．結果として，必須属性の欠如や依存情報の不整合が起こりうる．

トリガ:
開発者による明示，またはシステムによる自動処理
入力:
属性を削除するオブジェクト object
削除する属性 attribute
処理:
```
removeAttribute(object, attribute) {
    object.repository.removeAttribute(object, attribute)
    updateObject(object)
}
```
起こりうる違反:
ATTRIBUTE_REQUIRED

4.4.3 関連

関連の追加

オブジェクト間に関連を追加することによって，関連の両端となるオブジェクトの更新を引き起こす．

トリガ:
開発者による明示，またはシステムによる自動処理
入力:
関連の元となるオブジェクト source
関連の先となるオブジェクト target
追加する関連 assoc
処理:
```
addAssociation(source, target, assoc) {
    source.repository.addAssociation(source, target, assoc)
```

```
        updateObject(source)
        updateObject(target)
    }
```
起こりうる違反:
```
    ASSOCIATION_EXCEEDS_UPPERBOUND
```

関連の修正

MOF における関連は，MOF オブジェクト同士を 1 対 1 に接続する単純なリンクに過ぎない．そのため，関連を一度削除した上で追加する処理と考えて問題ない．

関連の削除

既存の関連を削除することによって，その関連を保持していた両端のオブジェクトの更新を引き起こす．

トリガ:
    開発者による明示，またはシステムによる自動処理
入力:
    削除するリンク `link`
処理:
```
    removeAssociation(link) {
        link.source.repository.removeAssociation(link)
        updateObject(link.source)
        updateObject(link.target)
    }
```
起こりうる違反:
```
    ASSOCIATION_REQUIRED
```

4.5 Struts に適用した場合のメタモデル

本節では，提案する枠組みを Web アプリケーションフレームワーク Struts に適用した場合の各 MOF メタモデル例について説明する．具体的には，

- 画面遷移図のメタモデル
- struts-config.xml のメタモデル
- 依存関係のメタモデル

の 3 つのメタモデルを定義する．2.2 節で述べたように，画面遷移図が設計時の成果物，struts-config.xml が実装時の成果物となっている．ここでは，Struts フレームワークにおける画面遷移の設計部分にのみ着目し，図 5，図 6 の例をもとにその概略を説明する．

なお，実際の struts-config.xml の仕様は本論文で述べるよりも広く，それにともなって各メタモデルの定義もより詳細なものにする必要がある．しかし，ここで述べる定義は Struts の画面遷移部分をカバーしているため，詳細化を進める際にはそのまま利用可能である．

4.5.1 画面遷移図の例

画面遷移図の MOF メタモデルは UML のアクティビティ図を拡張して定義する．図 23 上部の 3 つの MOF クラスは，アクティビティ図のメタモデルの中心部分を定義している．具体的には，状態遷移を記述するために，ノード（StNode）と辺（StEdge）を定義し，ノードとノードを辺で接続することによって遷移関係を表現する．また，遷移条件として，辺には遷移可能性を判定するためのガード条件（StValueSpecification）が関連付けられている．実際のアクティビティのメタモデルでは，さらに意味論を厳密に定義するためにノードや辺の種類を詳細化したり，イベントのトリガを定義したりということが行われているが，ここでそれらを全て説明することは不可能であり，本質からも外れるため省略する．

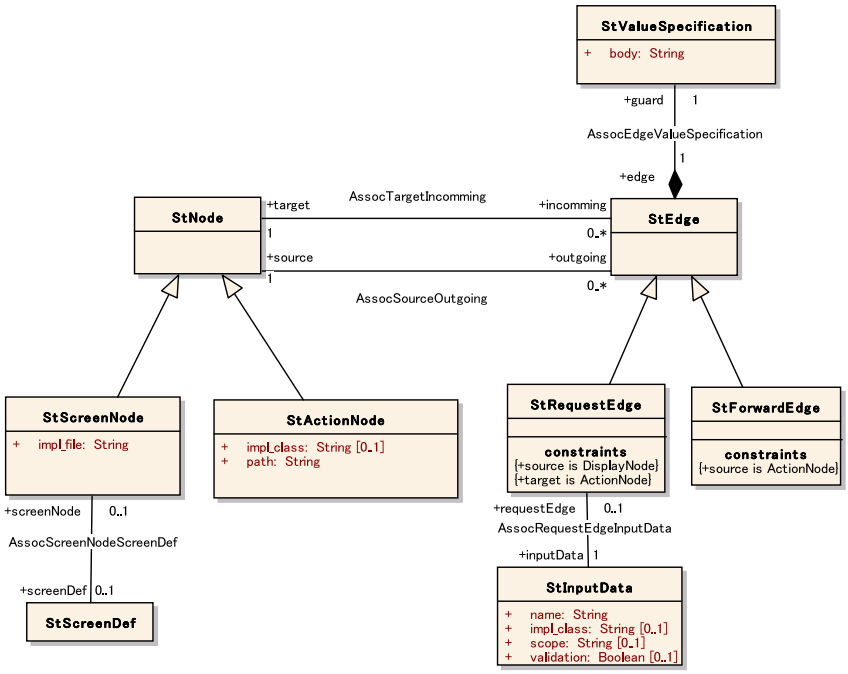

図 23: 画面遷移図の MOF メタモデル

2.2 節で述べたように，Struts では画面と画面を直接遷移可能な状態で結合するのではなく，アクションサーブレットを介して入力データはアクションフォーム Beans へ，処理はアクションクラスへ振り分けられる．そして，その処理結果を受けて次の遷移画面が決定す

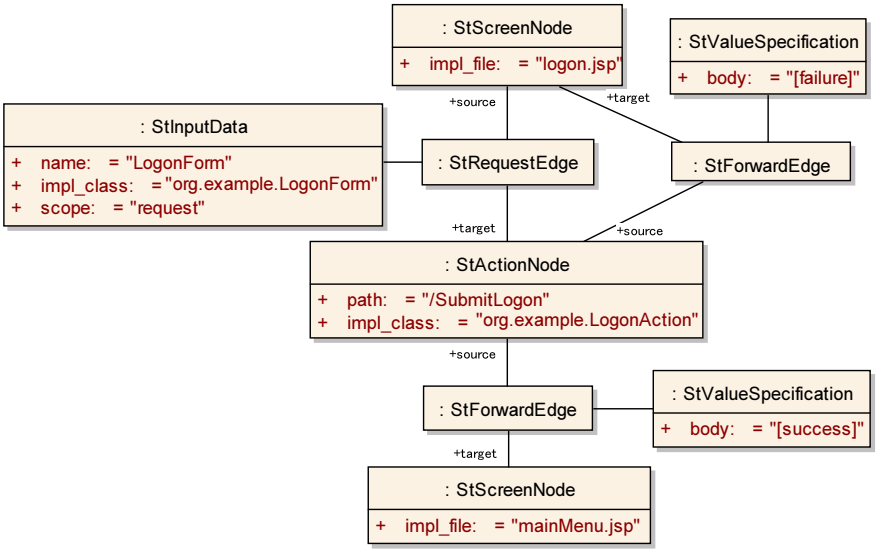

図 24: 画面遷移図のインスタンス例

るという構造になっている．つまり，Struts に適した画面遷移図とは，画面と画面を直接接続するようなモデルではなく，画面から処理ノードへ接続し，処理ノードから次画面へ接続するという，図 6 のようなモデルが良いと考えられる．そこで，提案する MOF メタモデルでは画面と処理ノードをモデル化するために StNode のサブクラスとして StScreenNode と StActionNode を定義している．StScreenNode は画面を表し，それを実現する JSP ファイルの情報を保持する．一方，StActionNode は処理ノードを表し，このノードへのパスやリクエストを処理するアクションクラスの情報を保持する．また，テキストボックスやボタンの表示レイアウトなど画面に関する詳細な設計を StScreenDef として遷移モデルから分離可能にしている．

さらに，これらのノードを接続する 2 種類の辺を定義する．1 つは，画面ノード StScreenNode から処理ノード StActionNode へのリクエストを表す辺 StRequestEdge である．リクエストが送られる際には，画面からユーザの入力情報が渡されうるので，辺 StRequestEdge と関連させる形で StInputData を定義し，実際にデータを保存するためのクラスに関する情報を保持させる．もう 1 つの辺は StActionNode からの遷移を表す辺 StForwardEdge である．アクションクラスの処理結果によって遷移先は複数存在しうるので，その場合は遷移条件を StValueSpecification として記述し，複数の StForwardEdge によって遷移先を分岐させる．

メタモデルのインスタンス例として，図 6 に相当する部分を図 24 に示す．また，図 6 と図 24 で表現形式が異なるが，図 6 は図 24 で表されるモデルの一表現であって，意味として記述しようとしている内容は同一である．ただし，直接図 24 のようなモデルを記述するの

は困難であるため，実用化に際してはモデリングツール上で図 6 のようなグラフィカルな画面遷移図を描き，その背後で厳密な意味定義として図 24 が作成されることを想定している．

4.5.2 struts-config.xml の例

struts-config.xml は XML ファイルであり，そのメタデータ構造は DTD （Document Type Definition:文書定義型）を用いて定義されている．しかし，提案手法では全てのメタモデルを MOF で記述し，成果物を MOF リポジトリのもとで統一的に管理することを目的としているため，struts-config.xml のメタモデルも MOF を用いて定義する．

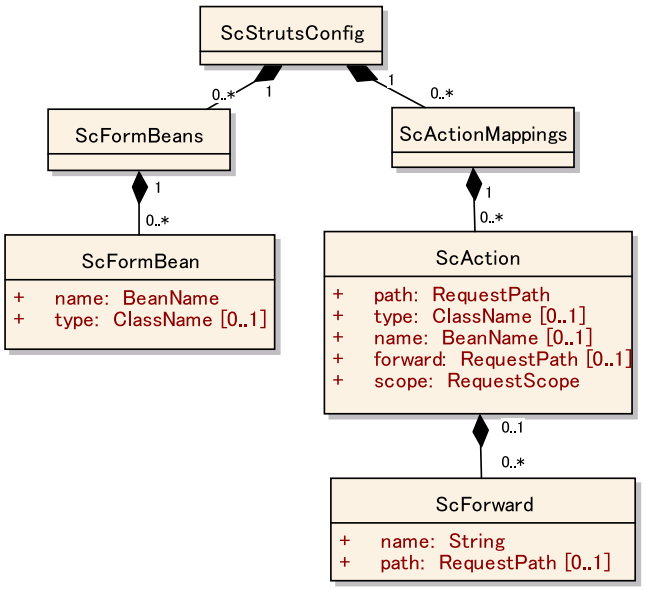

図 25: struts-config.xml の MOF メタモデル

図 25 は struts-config.xml のメタモデルのうち，図 5 で表されるアクションフォーム Beans とアクションクラスの記述に対応する部分である．同メタモデルは，XML 文書における要素 `ELEMENT` を MOF クラスとして定義し，同様に XML 文書における属性 `ATTLIST` を MOF クラスの属性として定義したものである．また，XML 文書中の要素 `ELEMENT` 間の親子関係を，MOF クラス間の集約関係として保持する形で定義している．

メタモデルのインスタンスとして，図 5 相当部分を図 26 に示す．この例では，図 5 中の `<form-bean>` と 2 番目の `<action>` の記述に関する部分（図 6 の `logon.jsp` 画面から `mainMenu.jsp` 画面への遷移部分）のみを抜粋している．図 5 と図 26 では表現形式が異なるが，図 5 は図 26 で表されるモデルのテキスト表現に過ぎず，意味として記述しようとしている内容は同一である点に注意されたい．

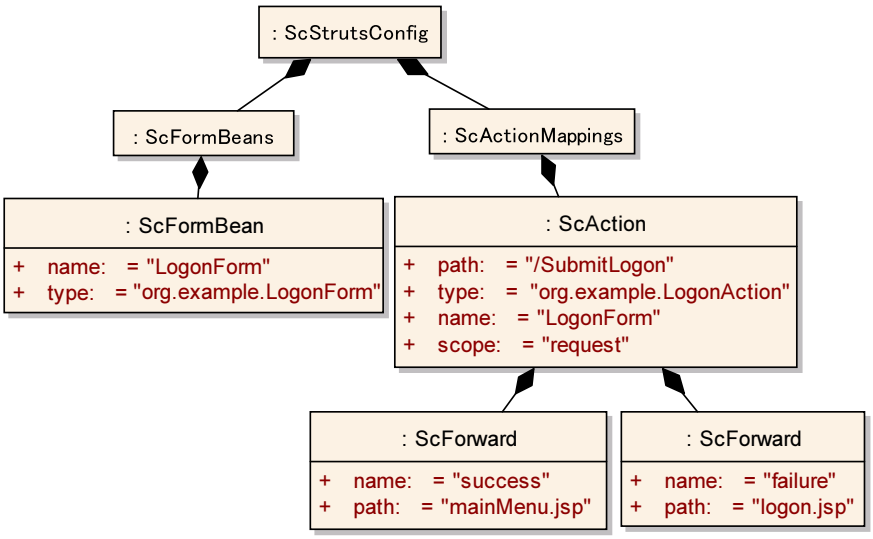

図 26: struts-config.xml のインスタンス例

4.5.3 画面遷移図と struts-config.xml の依存関係の例

画面遷移図と struts-config.xml の依存関係定義に対するモデルは図 27 のように定義する．ただし，クラス Rule のインスタンスをそのまま記述するのは煩雑であったため，ここではノートという形で入力要素群と出力モデル要素群の詳細な対応関係を記述している．また，クラス図に付加されているステレオタイプは，各要素がステレオタイプ中で記述された MOF クラスのインスタンスであるということを意味している．

例えば，図 27 の最上段の依存関係定義 TraceForward は，画面遷移図のメタモデルにおける StForwardEdge や StValueSpecification が，struts-config.xml のメタモデル中の ScForward に影響を与えることを意味する．そして，より詳細な対応関係として，StValueSpecification の body 属性が，ScForward の name 属性に対応付けられることを表す．

あるいは，StInputData を参照する InputDataRef に着目すると，この MOF オブジェクトと関連のある依存関係定義は TraceAction と TraceBean の 2 種類が存在することが分かる．つまり，画面遷移図中で StInputData を作成したり変更したりすることが，struts-config.xml 中の複数の要素へ影響を与えてしまうことを意味している．

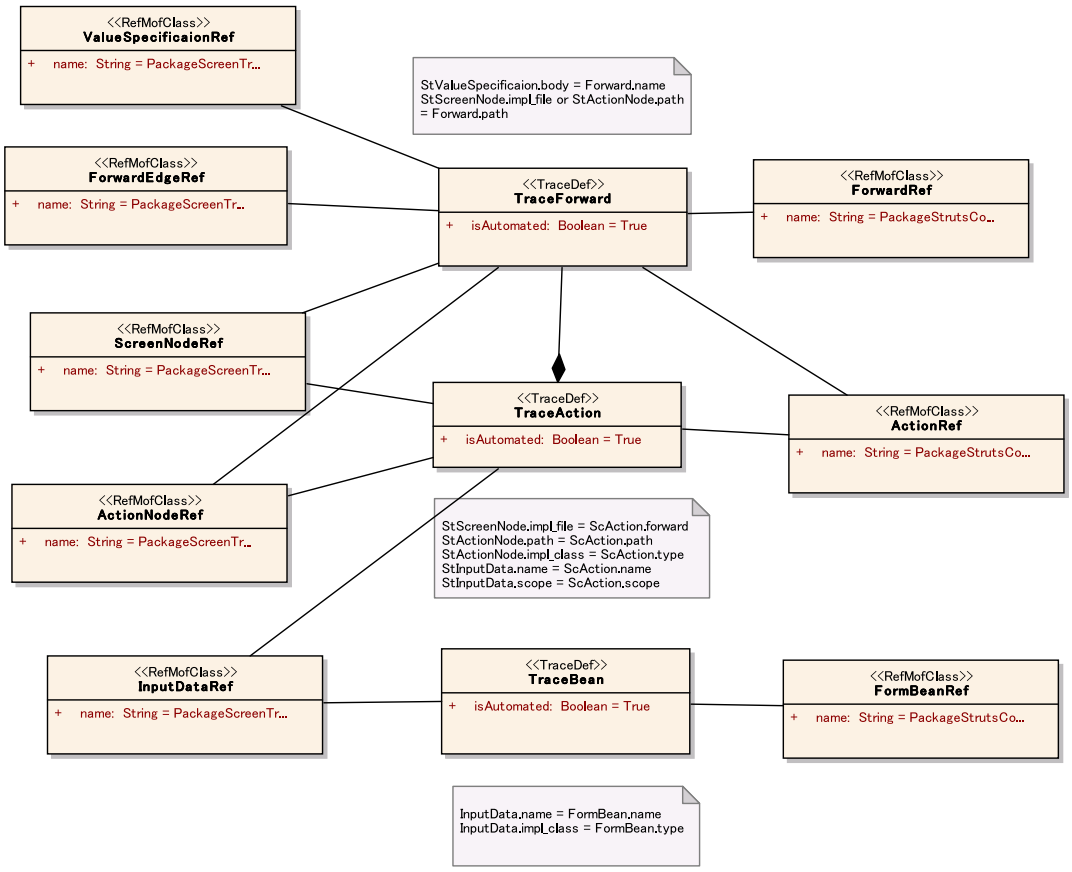

図 27: 画面遷移図と struts-config.xml の依存関係定義

5　システムの実装

本節では，実装したシステムについて説明する．本システムは，提案手法を Struts フレームワークの画面遷移設計に適用した場合の支援システムとなっている．後述する UML モデリングツール Enterprise Architect[30] のアドインとして実装されており，開発環境は以下の通りである．

- CPU: Intel Pentium M 1.2GHz
- RAM: 760MB RAM
- OS: Microsoft Windows XP
- 言語: C#
- LOC: 約 7,000 行（リポジトリおよびリポジトリとの結合部は除く）

5.1　利用ソフトウェア

本システムを実装するにあたって，Enterprise Architect[30]，Medini[16]，Janeva[7] の 3 つのソフトウェアを利用している．ここでは，これらのソフトウェアの機能・特徴について簡単に説明した上で，各ソフトウェアを利用している目的を述べる．

Enterprise Architect

Enterprise Architect[30] は Windows 上で動作する UML モデリングツールである．Enterprise Architect は UML2.0 に対応した 13 のダイアグラムを視覚的に編集可能な環境を開発者に提供し（図 28），ソフトウェア開発時の分析・設計に利用できる．加えて，VisualBasic や C#によって記述された独自アドインを追加する仕組みを備えているため，ユーザが独自の機能を追加することが可能となっている．

本研究では，開発者が行うモデル編集を扱うシステムを実装する必要があったため，その基盤プログラムとして Enterprise Architect に着目した．つまり，アドインとしてシステムを実装することで，Enterprise Architect があらかじめ備えるモデリングツールとしての機能をできる限り再利用する．

Medini

Medini[16] は MOF to IDL Mapping を実装した MOF メタモデリングツールである．Medini は Enterprise Architect のアドインとして動作し，以下の機能を実現する．

- MOF メタモデル作成環境の提供

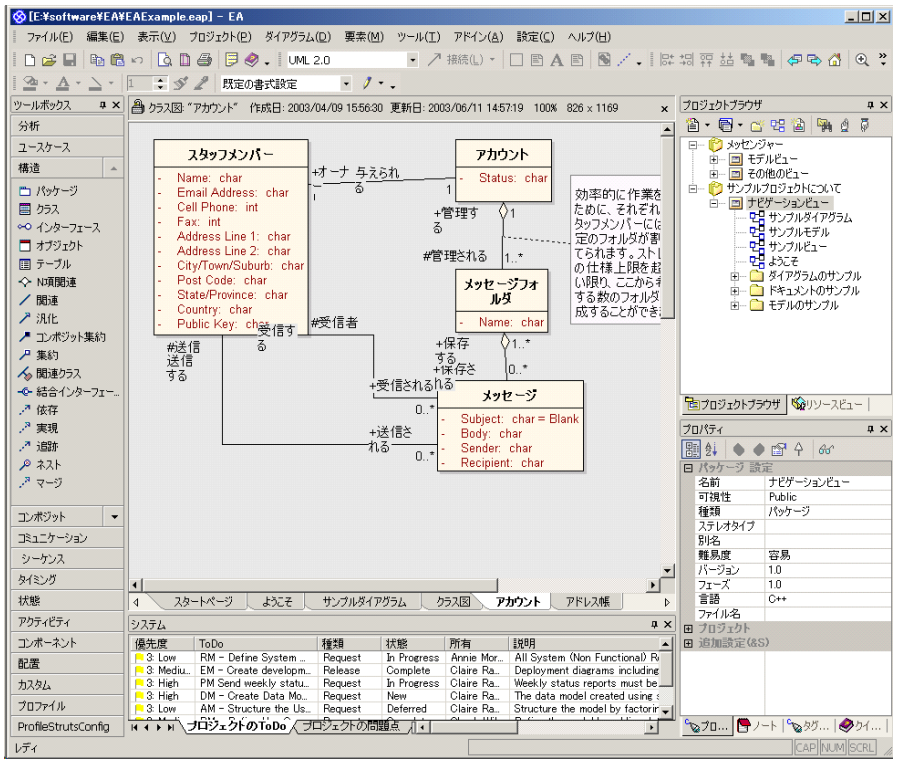

図 28: Enterprise Architect スクリーンショット

- MOF メタモデルから MOF リポジトリプログラムの生成

本研究では，成果物とその依存関係に対する MOF メタモデルを定義するために Medini を利用し，さらに同ソフトウェアを用いて MOF リポジトリプログラムを生成している．

Janeva

Janeva[7] は Microsoft .NET Framework アプリケーションと CORBA ランタイム環境との相互運用を可能とするミドルウェアである．Janeva を用いることで，Microsoft .NET Framework を用いて構築したクライアントアプリケーションから，CORBA で構築されたサーバーサイドコンポーネントに接続することが可能となる．

本研究では，Medini が生成した MOF リポジトリプログラムと Enterprise Architect アドインとして実装した C#プログラムを統合し，C#プログラムから MOF リポジトリを操作できるようにするために Janeva を利用している．

5.2 システムの概要

本システムの概要について述べる．システムは図 29 で表されるような構成となっており，大きく成果物管理部，依存関係管理部，制御部から構成される．成果物管理部は，MOF リポジトリを用いて開発者が作成したモデルを蓄積し，メタモデルをもとに妥当性の検証を行う．依存関係管理部は，実際に作成されたモデル間の依存関係を蓄積する．制御部は，Enterprise Architect を通して開発者が行うモデルの編集操作を監視し，成果物管理部と依存関係管理部を統合することによって開発者のモデリングを支援する．

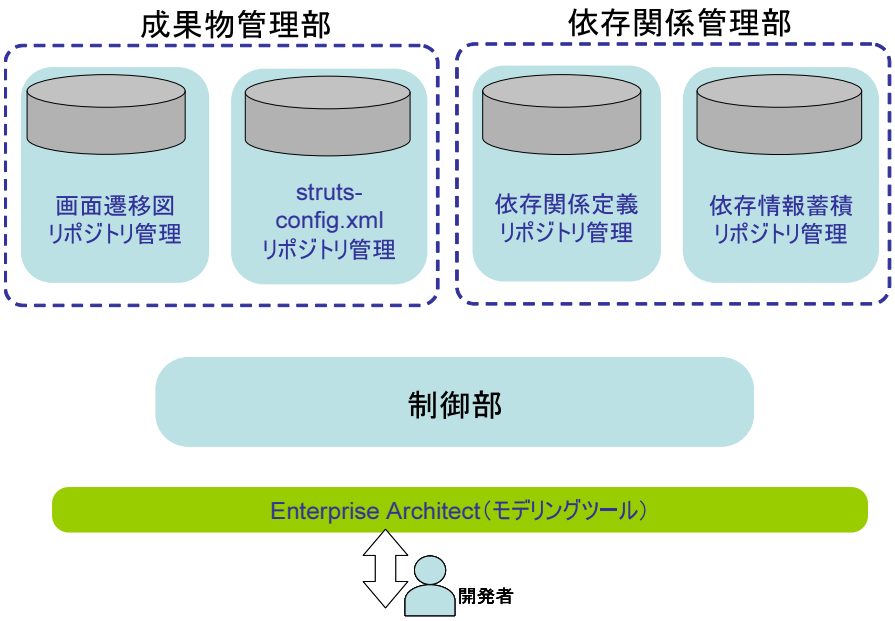

図 29: システム構成図

Enterprise Architect は UML のモデリングツールであるため，MOF オブジェクトやその属性，関連の編集といった操作を直接扱うことはできない．しかし，本システムは Enterprise Architect に対する UML クラスの作成を内部で MOF オブジェクトの作成と解釈することによってこの問題を解決している．つまり，開発者が Enterprise Architect 上で UML クラスの作成，属性の編集，関連の作成といった操作を実行すると，システム内部では MOF オブジェクトの作成，属性の編集，関連の作成が行われたものと解釈し動作する．

5.3 システムの詳細

本システムが備える詳細な機能について，成果物管理部，依存関係管理部，制御部の各コンポーネント毎に説明を加える．

5.3.1 成果物管理部

成果物管理部は，あらかじめ定義された成果物の MOF メタモデルをもとに，開発者が作成したモデルの蓄積や妥当性検証を行う．本システムでは，Struts フレームワークの画面遷移設計に対する適用例を実装しているため，成果物管理部は以下の二つのサブコンポーネントに分割される（図 29）．

- 成果物 “画面遷移図” を管理する部分
- 成果物 “struts-config.xml” を管理する部分

成果物を管理する各コンポーネントは，自身が取り扱う成果物に応じた MOF メタモデル・MOF リポジトリを保持している．これらのコンポーネントは制御部の要求に応じて動作し，それぞれが次のような機能を実現する．

- MOF リポジトリ中のモデルデータを更新（追加・修正・削除）する
- 管理するモデルが MOF メタモデルを満たしているかを検証する

成果物管理部が検出したメタモデル違反は制御部に通知され，図 30 のようなメッセージウィンドウを通して開発者に警告がフィードバックされる．メタモデル違反は 4.2 節 で述べたものが自動検出され，対象の MOF オブジェクトやエラーメッセージとともに表示される．また，リストから項目を選択することで，問題を含む MOF オブジェクトの編集画面に切り換えることができるようになっている．

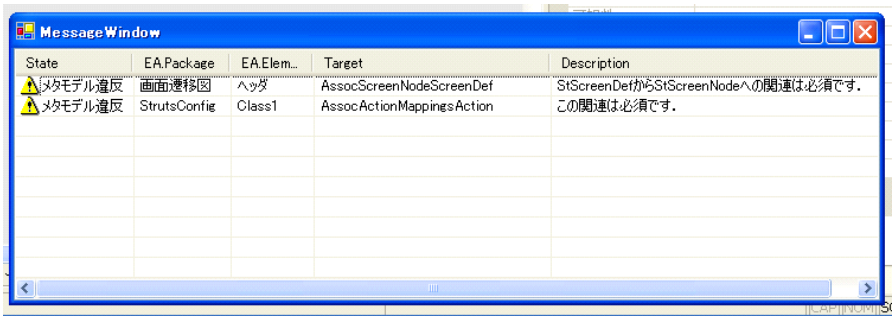

図 30: メタモデル違反の通知ウィンドウ

5.3.2 依存関係管理部

依存関係管理部は，成果物間に作成されるべき依存関係の定義を保持し，実際に作成された依存関係の蓄積を行う．本システムでは，Struts フレームワークへの適用例を実装しているため，画面遷移図と struts-config.xml 間に存在すべき依存関係を管理することになる．

成果物管理部と同様に，依存関係管理部も MOF リポジトリを保持している．開発者が成果物のモデルを作成すると，それらの依存関係がこのリポジトリに蓄積されていく．依存関係管理部も制御部の要求に応じて動作し，以下のような機能を実現する．

- 与えられた MOF オブジェクト（モデル）に対し，依存して作成されるべき MOF オブジェクトが存在するかを調べる
- 与えられた MOF オブジェクトに対し，既に依存して作成された MOF オブジェクトが存在するかを調べる
- モデル間に作成された依存関係を MOF リポジトリに蓄積する

設計成果物だけが作成されて実装が作成されていない場合のように，あらかじめ定義された依存関係が実際には作成されていない状況が起こりうる．このような場合には制御部に通知され，図 31 のようなメッセージウィンドウを通して開発者に警告がフィードバックされる．

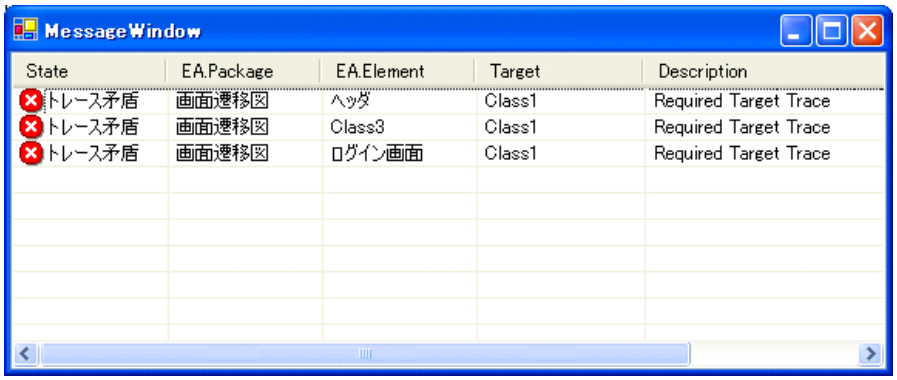

図 31: 依存関係矛盾の通知ウィンドウ

5.3.3 制御部

制御部は，Enterprise Architect のイベントハンドラとして開発者とのインタラクションを実現する．制御部は，成果物管理部と依存関係管理部を統合し，開発者があらかじめ定義されたメタモデルに従ったモデルを作成できるよう支援する．具体的には，以下のような機能を実現している．

- メタモデルを満たす MOF オブジェクトの作成のみを許可する
- メタモデル違反を検出し，警告・修正要求をフィードバックする

- ある MOF オブジェクトに依存する別の MOF オブジェクトを検出し，警告・修正要求をフィードバックする
- 自動的に作成可能な依存オブジェクトは自動生成する
- 開発者のモデリングにあわせて，成果物および依存関係の情報を適宜更新する

5.4 利用例

支援システムのメイン画面は図 32 のようになっている．画面左のツールボックスには，成果物のメタモデルによって作成可能であると定められた MOF クラスが表示される．開発者は，マウスでアイコンをクリックし，モデルエディタ上に配置することによってそのインスタンスである MOF オブジェクトを作成する．これにより，メタモデルで定義されていない誤ったクラスインスタンスを作成されることを防止する．作成中のモデル一覧は，画面右にパッケージ階層として表示される．

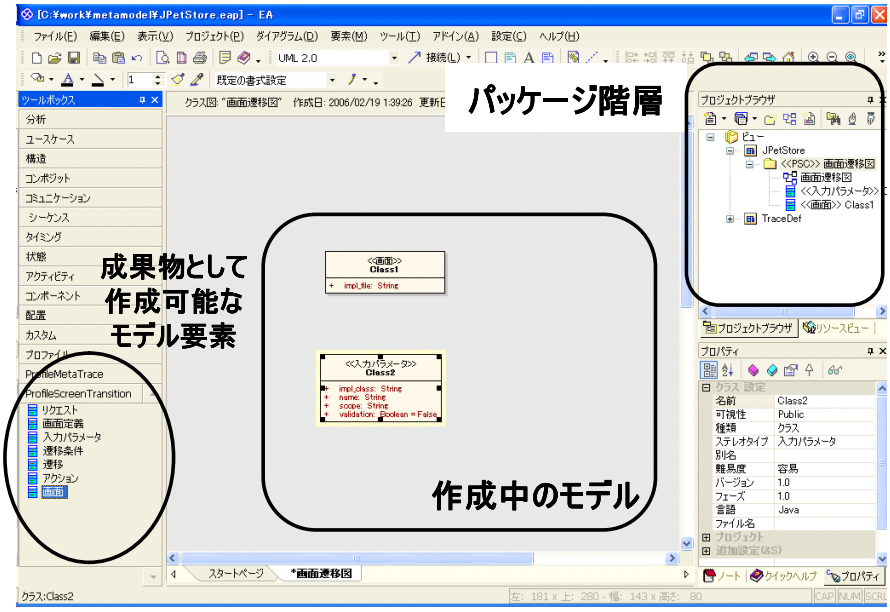

図 32: システムのスクリーンショット

開発者が MOF オブジェクトを配置すると，システムはメタモデルを参照し，必要な属性を自動で追加する．ただし，この状態では値として初期値が与えられるため，適切な値に修正する必要がある．開発者は，図 33 のような編集ウィンドウで MOF オブジェクトに適切な値を設定することを繰り返し，モデルを詳細化していく．

Struts の画面遷移設計においては，画面遷移図が設計成果物，struts-config.xml が実装成果物となっている．struts-config.xml は画面遷移図の情報を参照する形で作成される．そのため，

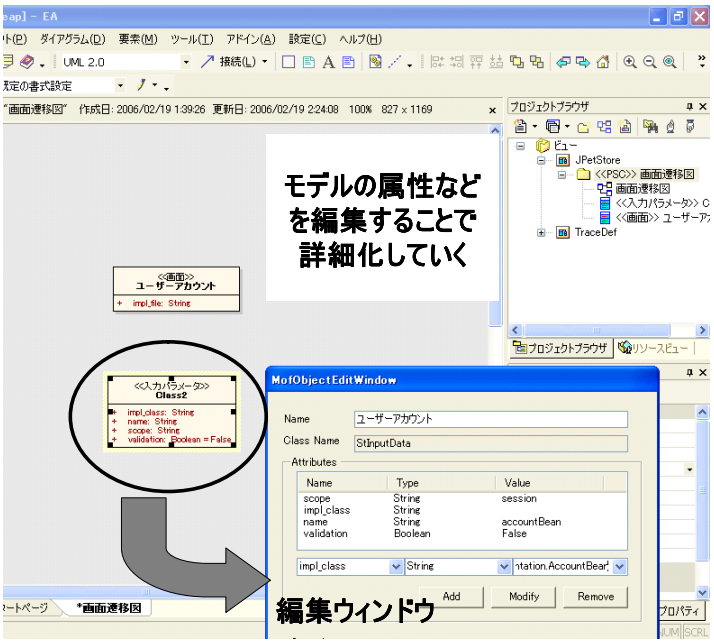

図 33: MOF オブジェクトの編集

支援システムは開発者が作成した画面遷移図から自動的に生成可能部分の struts-config.xml モデルを作成する．支援システムは 4 節で述べた枠組みによって，成果物であるモデル間の依存関係を管理する．ただし，自動変換に関する記述規則は提案手法の枠組みの中では定めていないため，この支援システムではハードコーディングによって実現している．

画面遷移図のモデルと，それに対応して作成されている struts-config.xml のモデルを図 34 に示す．画面遷移図中の「ユーザアカウント」の情報は，struts-config.xml における「FormBeans」によって参照される．なお，Enterprise Architect では 2 つのモデルを左右に並べて同時に編集する機能がないため，Enterprise Architect のアドインである本支援システムも現状ではそのようなモデリングを行うことはできない．実際にモデリングするときには，成果物毎に画面を切り換えながら作業することになる．

また，開発者が作成したモデルに対して，支援システムはそのメタモデルとの間で妥当性の検証を行う．開発者が何らかの問題のあるモデルを成果物として作成した場合にはメタモデル違反が警告される．また，作成したモデルに依存して作られなければならない他のモデルが残っている場合や，依存関係にあるモデルのうち一方のみが修正された上に，自動で解決できないような場合にも，同様に修正要求をフィードバックする．これにより，開発者が正しいモデルを作成できるよう支援できる．

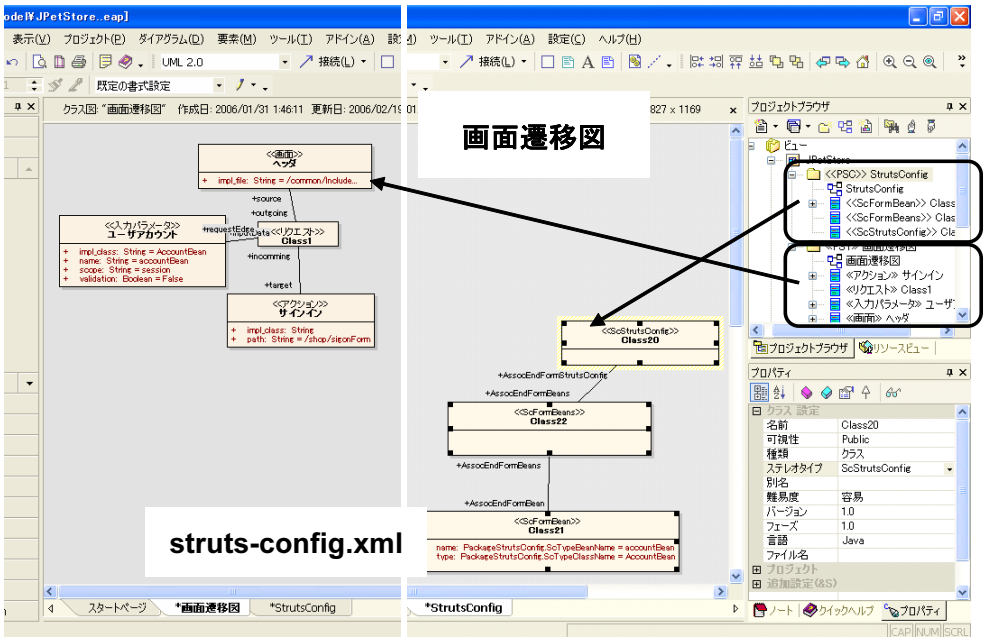

図 34: 画面遷移図と struts-config.xml の編集

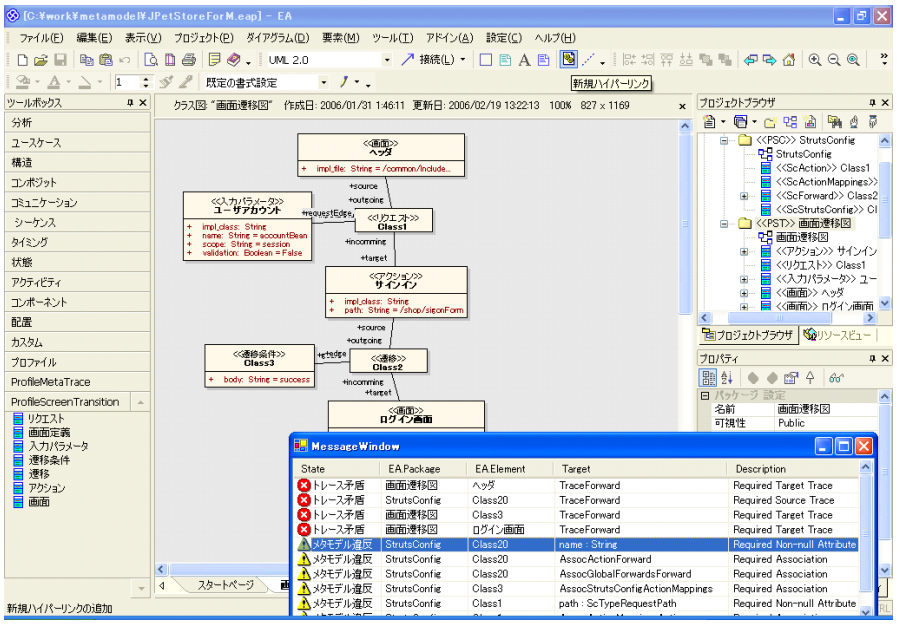

図 35: 開発者へのフィードバック

6 むすび

本研究では，メタモデルを扱う技術として MOF（Meta Object Facility）に着目し，成果物とその依存情報を統一的に管理可能な枠組みの提案を行った．提案する枠組みは特定の成果物に依存するものではないため，モデルを中心としたソフトウェア開発における基盤技術となることが望まれる．

また，提案手法を Web アプリケーションフレームワーク Struts を用いた画面遷移設計に適用し，実際に動作するシステムとして実現可能であることを確認した．

今後の課題としては，以下の事項が挙げられる．

- 変換規則との統合
  現在はモデルの自動変換よりもモデル間の関係性を保存することに重点をおいているが，モデル変換技術と統合することにより，より実用的なモデル管理技術の実現を図る．

- 成果物毎のモデル表記法の定義
  あらかじめ成果物に対して定義された MOF メタモデルのもとでモデリングを行う際に，開発者が直接インスタンスである MOF オブジェクトを操作するのは冗長かつ煩雑である．そのため，実際に何らかの成果物に対するモデリングツールを実装する際には，対象の成果物に応じて，UML と同様の適切な表記法を定めることが望ましい．

- 追加の評価実験
  本研究では Struts フレームワークに対する規模の小さな適用にとどまったが，より大規模なテストケースへの適用や，あるいは Struts とは異なる成果物への適用を行う．

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝致します.

本研究の過程を通して，常に適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 楠本 真二 教授に心より感謝致します.

本研究の過程を通して，適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 助教授に心より感謝致します.

本研究の過程を通して，逐次適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 岡野 浩三 助教授に心より感謝致します.

本研究の過程を通して，逐次適切な御指導，御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 氏に深く感謝致します.

本研究に対して，開発現場の視点から貴重なコメントを頂いた，株式会社 NTT データ 塚本 英昭 氏，我妻 智之 氏，山下 裕介 氏に深く感謝致します.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上研究室の皆様に深く感謝いたします.

参考文献

[1] Akehurst, D. H. and Kent, S.: A Relational Approach to Defining Transformations in a Metamodel, *Proceedings of the International Conference on The Unified Modeling Language*, Springer-Verlag, pp. 243–258 (2002).

[2] Antoniol, G., Canfora, G., Casazza, G. and Lucia, A. D.: Information Retrieval Models for Recovering Traceability Links between Code and Documentation, *Proceedings of the International Conference on Software Maintenance*, San Jose, USA, pp. 40–49 (2000).

[3] 青木　淳：オブジェクト指向分析設計入門，ソフト・リサーチ・センター (1993).

[4] Apache Struts Project:
`http://struts.apache.org/`.

[5] Blanc, X., Gervais, M.-P. and Sriplakich, P.: Model Bus: Towards the Interoperability of Modelling Tools, *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004*, Springer-Verlag, pp. 17–32 (2005).

[6] Bondé, L., Boulet, P. and Dekeyser, J.-L.: Traceability and Interoperability at Different Levels of Abstraction in Model Transformations, *Forum on Specification and Design Languages(FDL'05)*, Lausanne, Switzerland (2005).

[7] Borland Software Corporation: *Janeva*.
`http://www.borland.co.jp/janeva/`.

[8] Briand, L. C., Labiche, Y. and O'Sullivan: Impact Analysis and Change Management of UML Models, *Proceedings of the International Conference on Software Maintenance*, Amsterdam, Netherlands, pp. 256–265 (2003).

[9] Caplat, G. and Sourrouille, J. L.: Model Mapping in MDA, *Workshop in Software Model Engineering*, Dresden, Germany (2002).

[10] Champeau, J. and Rochefort, E.: Model Engineering and Traceability, *Workshop Model Driven Architecture in the Specification, Implementation and Validation of Object-oriented Embedded Systems*, San Francisco, USA (2003).

[11] Czarnecki, K. and Helsen, S.: Classification of Model Transformation Approaches, *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture* (2003).

[12] Dumoulin, C.: *ModTransf: A model to model transformation engine*. http://www.lifl.fr/ dumoulin/modTransf/.

[13] Egyed, A.: A Scenario-Driven Approach to Traceability, *Proceedings of the International Conference on Software Engineering*, Toronto, Canada, pp. 123–132 (2001).

[14] Gotel, O. C. Z. and Finkelstein, A. C. W.: An Analysis of the Requirements Traceability Problem, *Proceedings of the International Conference on Requirements Engineering*, Colorado Springs, USA, pp. 94–101 (1994).

[15] Hausmann, J. H. and Kent, S.: Visualizing Model Mappings in UML, *Proceedings of the 2003 ACM symposium on Software visualization*, San Diego, USA, pp. 169–178 (2003).

[16] IKV^{++} Technologies AG: *Medini*. http://www.ikv.de/.

[17] Java Metadata Interface (JMI): http://java.sun.com/products/jmi/.

[18] Judson, S. R., France, R. B. and Carver, D. L.: Specifying Model Transformations at the Metamodel Level, *Workshop in Software Model Engineering*, San Francisco, USA (2003).

[19] Kruchten, P.: *The Rational Unified Process, An Introduction*, Addison-Wesley (2000).

[20] Metadata Repository (MDR) Project: http://mdr.netbeans.org/.

[21] Nentwich, C., Emmerich, W. and Finkelstein, A.: Consistency Management with Repair Actions, *Proceedings of the International Conference on Software Engineering*, Portland, USA, pp. 455–464 (2003).

[22] Object Management Group: http://www.omg.org/.

[23] Object Management Group: *Meta Object Facility (MOF) Specification, Version 1.4* (2002).

[24] Object Management Group: *MDA Guide Version 1.0.1* (2003).

[25] Object Management Group: *Meta Object Facility (MOF) 2.0 XMI Mapping Specification, Version 2.1* (2003).

[26] Object Management Group: *Meta Object Facility (MOF) 2.0 Core Specification* (2004).

[27] Object Management Group: *MOF 2.0 Query/View/Transformation Final Adopted Specification* (2005).

[28] Object Management Group: *OCL 2.0 Specification, Version 2.0* (2005).

[29] Object Management Group: *UML Superstructure Specification, Version 2.0* (2005).

[30] Sparx Systems: *Enterprise Architect*. `http://www.sparxsystems.com.au/`.