大規模開発におけるモデル品質向上のためのモデル検証フレームワーク

新美 健一　大澤 浩二

Model Validation Framework for Model Quality Improvement in Large Project

Kenichi Niimi and Kohji Ohsawa

本論文は，UMLモデルの品質確保のためのモデル検証において，プロジェクト固有のモデリング・ルールを容易に定義・検証できるモデル検証フレームワークを提案する．本フレームワークでは，各モデルの要素に合わせて用意された検証ルールの部品の組み合わせにより，検証ルールを柔軟に定義可能なので，プロジェクト固有のモデリング・ルールを順守してモデルが作成されているかのインスペクション（静的チェック）の自動化も容易となり，手戻りコストの予防，標準化徹底のためのコストの削減を図ることが可能となる．

This paper proposes model validation framework which enables us to define the project specific validation rules flexibly and inspect models easily. According to a definition of validation, this framework combines some rules which are prepared for each UML element, and validates models. As a result, it makes possible for us to inspect models automatically, reduce costs of model development and standardization activity.

Words & Phrases：UML，インスペクション，モデル検証，品質マネジメント，オブジェクト制約言語（OCL）
UML, inspection, model validation, Quality Management,
Object Constraint Language（OCL）

1．はじめに

近年，オブジェクト指向開発ではUML（Unified Modeling Language）モデリングによる分析・設計・開発を実施するプロジェクトが多くなっている．また，モデル駆動型開発［1］を採用し，モデルやソースコードの一部自動生成を行う事例も出てきている．オブジェクト指向開発では複数のクラスのメソッドが相互作用を行い一つの機能を実現する．そのため，従来の1機能1メソッドのような開発スタイルと比較し，設計すべき要素が増加している．さらに近年のシステムの大規模化も，モデルの大規模化に拍車をかけている．そのためモデルの品質を効率的に確保する技術が求められており，その解決策の一つとしてモデル検証が挙げられる．

モデル検証とは，インスペクション（静的チェック）［2］を自動化するための技術の一つであり，モデルとしてどうあるべきかという検証ルールを定義し，作成したモデルがその検証ルールに合っているかを自動検証する技術である．ソフトウェア開発のすべてのエラーの50～65%が設計段階で混入するともいわれ［3］，インスペクションを行うことにより設計時のエラーの最大75%の発見に効果があるともされており［4］，モデル検証を実施することでモデル品質の確保と下流工程からの手戻りの防止を効率的に実施できる．さらに，モデル駆動型開発では自動で生成された成果物について細かなインスペクションを行うことはないため，その入力となるモデルの検証は不可避である．

モデル検証ルールには，次の三つのレベルがある．

①**メタモデル・レベル**:いわゆるUMLの仕様に基づくルール（例：クラスはパッケージを保持しない）

②**標準設計レベル**：モデリングする際に順守すべき標準的なルール（例：抽象クラスには具象サブクラスが必要である．循環参照は禁止する．シーケンス図のメッセージに対応する，クラスのオペレーションが存在すること.）

③**プロジェクト固有レベル**：プロジェクト固有の制約に合わせて定義されたルール（例：プロジェクト独自のモデル構造や命名ルール，アーキテクチャーの制約のための検証ルール）

RSA（Rational® Software Architect），RSM（Rational Software Modeler）などのモデリング環境や，従来なされてきたダイアグラム間整合性検証の研究［5］［6］では，①②は検証可能であるが，③の定義・検証はできない．しかし，以下の（1）～（3）のようなアプリケーション開発の状況からプロジェクト固有のモデル検証の実施が急務となっている．

提出日:2008年5月12日　再提出日:2008年9月30日

(1) アーキテクチャーの複雑化

エンタープライズ・レベルのアーキテクチャー［7］，独自フレームワークを利用するための設計など，アプリケーション設計をする上でのルールが多様化している．そのためモデル作成者がすべてのルールを正確に把握することが難しい．

(2) プロジェクト・メンバーのスキル

プロジェクトの大規模化による開発メンバーの増大や入れ替えにより，メンバーのスキルのバラつきが生じ，標準化事項がすべてのメンバーに徹底できない．

(3) アーキテクチャーや標準事項の変更

設計・開発を進める中で，アーキテクチャー上の制約や標準事項が変更されることがある．モデルの大規模化のため，変更に対する影響分析が十分にできず，モデルに対する修正漏れが発生する．

プロジェクト固有の検証ルールに従って検証するためには，EMF（Eclipse Modeling Framework）［8］の機能を利用して Eclipse のプラグインとして検証機能を実現したり，オブジェクト制約言語（Object Constraint Language）［9］を用いて検証したりすることも可能であるが，どちらも高い技術を必要とするため個々のプロジェクトで採用するのは困難である．型付きグラフを用いてモデルに対する制約を表現する研究［10］，Abstract State Machine（ASM）という表記でモデルの意味定義をし，ASM コンパイラーにかけてモデルを検証する研究［11］もされているが，個々のプロジェクトで簡単に利用できるほど定義と手順が容易ではない．また,UML モデル要素（クラスやユースケースなど）とダイアグラムや，その描画情報の相関ルール（図上での使用状況や，ある図 A で定義されたクラス間の関連と，ほかの図 B で定義されたクラス間の関連が矛盾していないかなど）を容易に定義できる機構は存在しない．これらの理由から,個々のプロジェクトにおいて，プロジェクト固有の検証ルールに対するモデル検証を実施することは困難であり，設計に対して誤りが混入しているのが現状である．

そこで本論文では，XML を用いて個々のプロジェクトでも容易にプロジェクト固有の検証ルールを定義，検証を実施でき，さらに複雑な検証ロジック実現のために Java を用いて拡張可能なモデル検証フレームワークを提案する．

以下，2 章でモデル検証フレームワークの要件を整理し，3 章で，その要件を実現するフレームワーク Quality Checker for Application Model（QCheckAM）を紹介する．さらに 4 章で，適用事例で実証した適用効果について述べる．

2. モデル検証フレームワークの要件

モデルの検証フレームワークが持つべき要件とその理由を次に示す．

(1) プロジェクト固有の検証ルールが定義可能

主に以下のようなプロジェクト固有のルールを定義し，検証可能とする．

(a) モデルの構造

パッケージの構造やクラス,インターフェースの配置場所，またその命名ルールの定義を行う．

(b) 各モデル要素のプロパティー

各モデル要素のプロパティーの設定の仕方の定義を行う．例えば，オペレーションであれば，可視性や戻り値の型，ステレオタイプの設定などである．

(c) 各モデル要素間の相関関係

モデル要素間の相関関係の定義を行う．例えば，分析モデルにおいて「依存線のある端がバウンダリーの場合には，もう一つの端はエンティティーではないこと」，などである．

(d) 条件付きのルール

特定の場合にのみ（a）～（c）が成り立つようなルールを定義する．例えば，「実装クラスを表現するクラスの場合，必ずクラスを説明する文書を記入していること」などである．

(2) 検証ルール定義の柔軟性

モデル構造やアーキテクチャー上の制約はプロジェクトによって異なるため，プロジェクト固有の検証ルールに合わせて検証ルールを柔軟に定義できる必要がある．また，あらかじめ用意されている検証ロジックだけでは，プロジェクト固有の検証ルールを実現できない可能性もある．そこで，利用者が新たな検証ロジックを作成し，あらかじめ用意されているルールと組み合わせてプロジェクト固有の検証を実現できる必要がある．

(3) 最新の検証ルールを反映できる運用環境

プロジェクト期間中に検証ルールが変更されることは多々ある．検証ルールが変更された場合に，必ず各モデリング担当者に検証ルールを更新してもらうため，日々の開発プロセスに組み込む必要がある．そこで検証ルール定義は構成管理の対象とし，日々必ず行われる構成管理ツールとの同期のタイミングで開発者の手元に取得可能となるようにする．

(4) 検証実行の容易性

日々の改善活動が負荷なく実施できるように，検証ルールの定義，検証の実行が容易に行える必要がある．そこで，誰でも慣れ親しんだことのある形式で検証ルールを記述可能とする．また，一回の操作で既存のモデル検証ツールも含め，同時に実行可能とし，エラー個所もすぐに特定できるようにする必要がある．

3. Quality Checker for Application Model (QCheckAM)

この章では，2 章で示した要件を実現するために開発した，モデル検証フレームワークである QCheckAM を紹介する．まず，3.1 節で QCheckAM を利用したモデリングの PDCA サイクルを説明し，要件 (3)，(4) の実現が可能であることを示す．次に要件 (1) ～ (2) を実現するための仕組みとして,3.2 節で,モデル検証を複数のルール部品の組み合わせで実現することを示し，ルール部品の種類と粒度を 3.3 節で，検証ルールの記述方法を 3.4 節で示す．

3.1 モデリングの PDCA サイクル

QCheckAM を利用した日々の開発の流れを図 1 に示す．QCheckAM を利用してモデリングに対する PDCA サイクルを回すことで，常にモデルの品質を確保する．

Plan：まず，アーキテクトや標準化担当がモデルとして守らなければならないルールを XML 形式の検証ルール定義として取りまとめ，構成管理ツールへ登録する．

Do：モデル作成者は，構成管理ツール上のモデルと同期するタイミングで，検証ルール定義の取得／更新を行う．このことにより，アーキテクト／標準化担当が作成した検証ルール定義を確実に手元に持ってくることが可能となる．続いて，モデリング規約に沿ってモデリングを行い，モデルの検証を行う．検証結果とエラーの原因を即座に確認，エラー個所に移動できるため，その場での修正を促すことができる．この検証を行うと，RSA のモデル検証機能も同時に実行可能であるため，モデル作成者は複数の検証を実施しなくても済む．さらに，重要度の高い規約については，モデルに修正を加えたと同時に自動で検証することも可能である．

Check：作成されたモデルをレビューし，アーキテクチャーの制約にあったモデリングとなっているか，属人性が混入していないかなどを確認する．また,残エラー項目を確認し，プロジェクトの進め方の検討材料とする．

Act：レビュー結果をもとに対策を検討し，必要に応じて検証ルールの修正を決定する．このタイミングでの変更は以下のような要因がある．

- アーキテクチャー，規約の変更：プロトタイピングなどを通して必要になったアーキテクチャー，規約の変更である．
- 用語の統一：モデリングをしている際にモデル作成者間で業務的な用語（名詞や動詞）のバラつきが出てくる．このような属人性を排除するために，命名ルールを規約として追加する必要がある．
- 自動化ツールの仕様決定：モデル駆動型開発のためにモデルやソースコードを生成するツールの仕様が決定すると，その仕様に合わせてモデリングを行う必要がある．

このようなことから検証ルール定義の変更をし，構成管理へ登録することで，次の PDCA サイクルでは新たな検証ルールでモデルの検証を実施することができる．

3.2 モデル検証フレームワーク

プロジェクト固有の検証ルールに柔軟に対応するために，QCheckAM はモデル要素の種類に合わせて検証ロジックを実装した「ルール部品」を組み合わせて検証を行う．その流れを制御するのが，モデル検証フレームワークである．図 2 に示すように，モデル検証フレームワークは，まず検証ルール定義に従い，ルール部品を動的に組み合わせる．次に UML モデル要素やダイアグラムの描画情報を読み込み，組み合わされたルール部品を利用して検証を実行し，検証結果の表示を行う．検証ルールに合わせて必要な検証ロジックを組み合わせて検証を実行

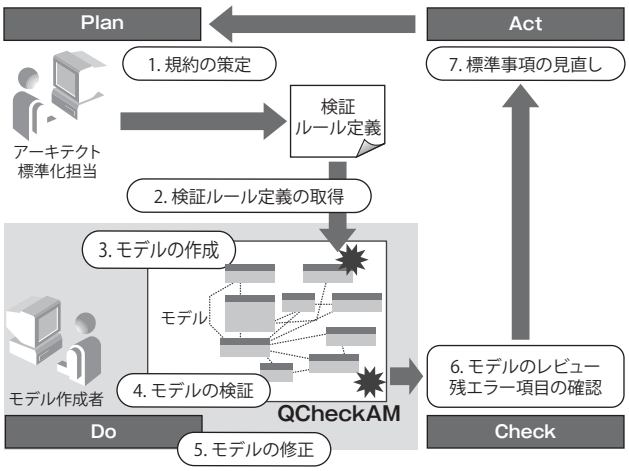

図 1. モデリングの PDCA サイクル

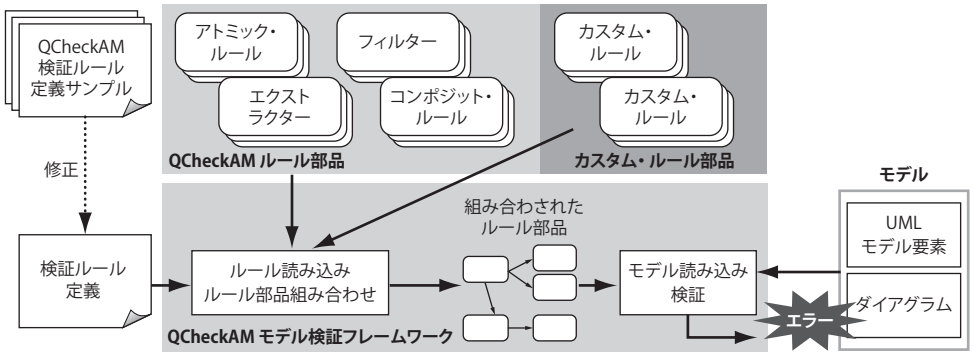

図 2. QCheckAM の検証の流れ

することにより，プロジェクト固有のルールにも柔軟に対応することが可能となっている．さらに，あらかじめ用意されたルール部品では実現できないような複雑な検証ロジックの場合には，カスタム・ルール部品を作成することにより検証可能な内容を増やし，検証することも可能である．

3.3 ルール部品

ルール部品は，各モデル要素の検証内容に合わせて検証ロジックを実装したものである．例えば，「名前付きのモデル要素」を検証するルール部品や，「多重度を持つモデル要素」の設定内容に応じてモデル要素を絞り込むルール部品，「クラスとオペレーションの関連」をたどり，保持するオペレーションを抽出するルール部品などがある．ルール部品は検証の振る舞いの最小単位であり，これらを組み合わせて検証を行うことにより，柔軟にプロジェクト固有の検証ルールを定義可能としている．

(1) ルール部品の種類

ルール部品は以下の 4 種類からなる．それらの関係を図 3 に示す．

(a) アトミック・ルール

UML のプロパティーの設定内容を検証する．プロパティーとは，例えばオペレーションの可視性や，属性の多重度などがある．

(b) フィルター／無視条件

モデル要素を検証対象とするか，無視するかの絞り込みを行う．例えば，「ソースコードに変換するモデルのみ対象にする」，「テンプレートとして利用するためのモデルは対象外とする」などである．

(c) エクストラクター

エクストラクターは，あるモデル要素を検証するときに，別のモデル要素の状態を検証する必要があるときに利用する．例えば，依存線があり，その端のクラスの状態をチェックする必要がある場合には，依存線の端のクラスを抽出（エクストラクト）し，さらにコンポジット・ルールもしくはアトミック・ルールを用いてクラスを検証する．

エクストラクターは UML モデル要素，ダイアグラムやダイアグラム上の描画情報を抽出することが可能である．例えば，「あるクラス（UML モデル要素）を利用しているクラス図（ダイアグラム）を抽出する」，などが可能となる．このことにより，UML モデル要素とダイアグラムや描画情報の間の相関関係を検証することが可能となる．

(d) コンポジット・ルール

アトミック・ルール，フィルター／無視条件，エクストラクターを組み合わせた検証ルールであり，必要に応じて，これらの And, Or, Not の演算も行う．ユーザーから見える検証ルールの単位である．例えば，「ソースコードに変換されるクラス，インターフェース，オペレーション，属性（フィルター条件で表現）には JavaDoc のためのモデル要素「文書」が記述されている（アトミック・ルールで表現）こと」などである．

(2) ルール部品の粒度

柔軟に検証ルールを定義するために，適切なルール部品の粒度を選択する必要がある．そこで，UML のメタモデルをルール部品の作成単位とした．UML のメタモデルは，UML の属性や関連を規定するためのモデルである．

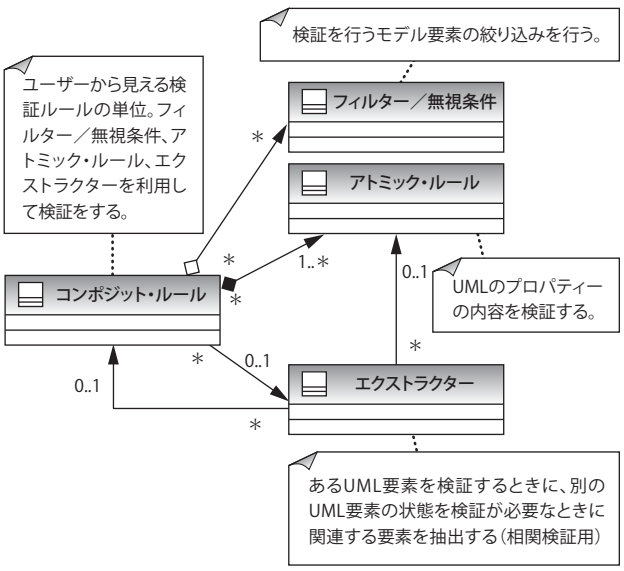

図 3. ルール部品の関係

そのため，UMLの属性や関連の条件を検証するルール部品の単位としてUMLのメタモデルを選択することは，UMLの検証ルール記述に親和性が高く，その組み合わせにより柔軟に検証ルールを定義可能となる．例えば，「名前付きのモデル要素」はUMLのメタモデルの一つであるが，この要素に対するルール部品を用意しておくことで，クラス，インターフェース，属性など名前を持つモデル要素の名前に関する検証を実施することができる．ただし，ユーザビリティーなどの観点から，UMLのメタモデルより大きな単位とする場合もある．また，コンポジット・ルールはそれ以外のルール部品の制御をするため，UMLのメタモデルの単位とは一致しない．

3.4 検証ルール定義

検証ルール定義のフォーマットは，広く使われているXMLを採用することで検証ルールの定義を容易にした．また，あらかじめ検証ルール定義のサンプルを提供し，命名ルールなどのパラメーターを変えるだけでプロジェクト要件に合わせた検証ルール定義を可能とした．QCheckAMでは，モデリングの際の品質チェックを行うための検証ルール定義のサンプルを，ユースケース・モデル，分析モデル，設計モデル用に提供している．

図4に，例として検証ルール定義の一部を抜粋した．「ルール部品」には，アトミック・ルール，フィルター／無視条件，エクストラクター，コンポジット・ルールの実装クラスを表す．例えば図中のUMLTypeNameValidationConditionではUMLの種類（クラス，インターフェースなど）をフィルターするためのルール部品である．そのルール部品に対して，具体的な「条件の値」を指定する．例では，「クラス，インターフェース，属性，操作，パラメータ」を指定している．さらにNamedElementValidationConditionを利用して，完全修飾名に「Implementation_Designs」を含むUMLモデル要素を検証の対象としている．エクストラクターでは，「DepenedencyClientExtractor」で依存線の端のクラスを抽出し，アトミック・ルール「ElementValidationCondition」でそのクラスに対して，「business」キーワードが当たっていることを検証している．つまり依存線が「business」キーワードを持つクラスを結んでいることを検証している．これらを組み合わせてコンポジット・ルールとして検証ルールを定義することができる．また条件の値については正規表現で記述可能とし，命名ルールやモデルの構造などの定義を容易にしている．

さらにコンポジット・ルールごとに重要度や動作モード，メッセージなどを設定して検証ルールとして定義する．動作モードはバッチ・モードとライブ・モードを指定することが可能で，バッチ・モードはメニューから検証を実行したときのみ実行され，ライブ・モードは変更を加えたモデルに対して毎回検証が実行される．リアルタイムで検証を行いたい場合には，ライブ・モードを指定する．

4. モデル検証による効果

モデル検証の効果を確認するために，簡易的にモデル検証ツールを作成，実プロジェクトへ適用し，その効果を確認した．次に，モデル検証をQCheckAMを利用して実施した場合のさらなるメリットの考察を行う．

4.1 モデル検証ツール適用事例

RSA上で動作するモデル検証ツールをJavaで実装し，実プロジェクトに適用した．最新のモデルの規模は，検証対象のモデル要素数にして約5万ほどであり，モデリング実施者の最大の人数は20名ほどである．主要な検証内容を以下に示す．

- パラメータの多重度の設定内容
- 属性，パラメーターの型の設定有無
- パッケージ名とモデリング規約との整合
- クラス名とモデリング規約との整合

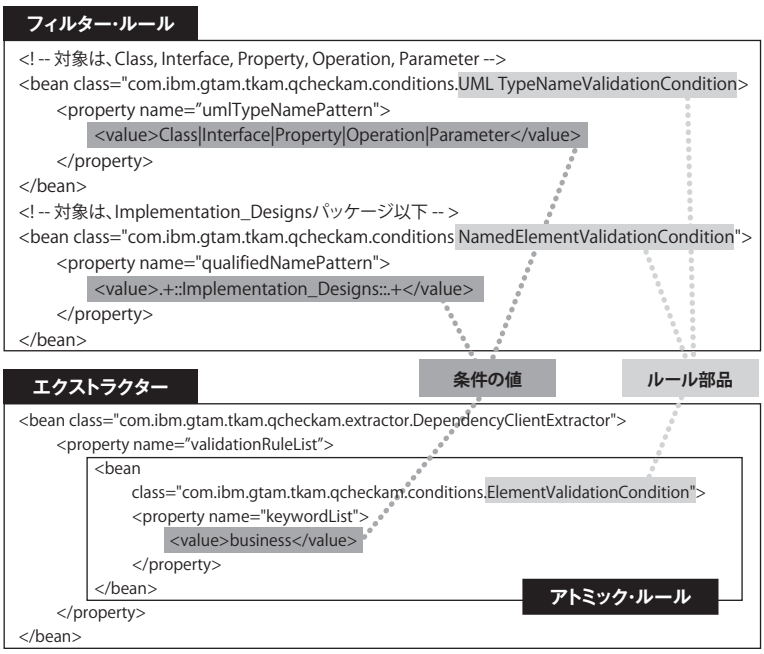

図4. 検証ルール定義

- インターフェース名とモデリング規約との整合
- Java の命名ルールとの整合

2008 年 3 月から検証ツール適用を開始し，違反項目の推移を観察した．2008 年 1 月の各チェック内容に対するエラー要素数を 100 としたときのエラー要素数の推移を図 5 に示す．適用した結果から得られた考察を以下に述べる．

(1) エラー要素の早期発見と修正

図 5 ではインターフェースや型の設定など，早期に固めておくべき項目を中心に，エラー要素を発見し，修正につなげられたことが分かる．また，名前などは設計を進める中で修正をしていくためエラー要素数の減少は緩やかに進んだが,モデリング実施者 20 名に対しネーミング･ルールの周知徹底が 3 カ月の間で確実に行われ，習熟効果が出ているといえる．

(2) レビュー時間の短縮

モデル検証ツール導入前も人手でレビューを行っていたため，時間がかかり，レビュー漏れも多かった．モデル検証ツールを導入することにより，数秒で検証が可能なため，レビュー時間の大幅な削減が可能となった．

(3) モデル品質の見える化と手戻りの減少

エラー数，エラー個所が分かるため，次工程へ進むことが可能であるか判断が可能となり，次工程からの手戻りも減少した．

(4) レビュー精度の向上

細部まで網羅的に検証が可能となった．例えば，図 6 の＜誤＞で示されているモデルは，モデリング中の操作ミスで OrderDeailDTO クラスをドラッグ&ドロップして OrderDTO クラスのインナー・クラスにしてしまった例である．このモデル要素は自動生成される要素であり,レビュー対象からも外れていたため，検証ツールを利用することで初めて明らかになった．

(5) モデリング規約の徹底

検証ツール導入まではエクセル，ワードなどを利用してモデリングの規約を行っていたが，人による理解度もまちまちであったり，規約が変更された際にモデル作成者に伝わらなかったりし，モデリング規約が徹底されていなかった．検証ツールを利用することにより，モデリング規約の徹底が可能となり，それまで週に数件発生していたルール不順守に由来する手戻りは収束した．また，検証ツールを構成管理ツール経由で配布したため，標準化展開のコスト削減も可能となった．

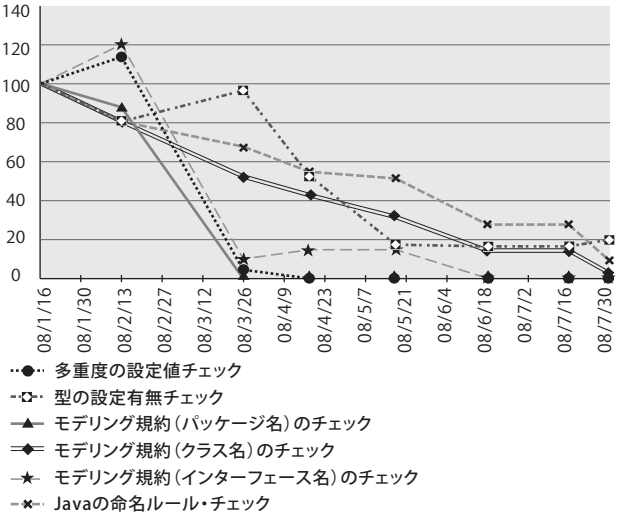

図 5. エラー要素数の推移

(6) 属人性の排除

PDCA サイクルを通して，モデリング中に見えてきた名前の付け方などの揺れは，命名ルールを新たに規約として盛り込むことで是正した．命名ルールのようなノウハウはメンバーの参画した時期などによって習得度合いも異なることが問題になっていたが，検証ツールを利用することによりノウハウの共有が促進できた．

(7) モデリング規約変更による影響度の見積もり

検証ツール導入までは，モデリング規約を変更する際にどれだけの影響があるか把握が難しかった．規約を変更する際に，この検証ツールを用いて規約の変更前，変更後を比較することにより，規約変更による影響度を見積もることが可能となった．

4.2 QCheckAM 利用による効果

検証ツールを独自で開発するのと比較し，QCheckAM を利用することでさらに以下のような効果が得られる．

(1) 導入コスト

4.1 で適用した検証ツールと同様の検証内容を QCheckAM の検証ルール定義として表現し，検証が可能であることを確認した．プロジェクトに適用した検証ツールのうち，今回例示した検証部分の設計・実装は 8 人日ほどかかったが，QCheckAM では検証ルール定義を 1 時間ほどで作成し，検証することができた．

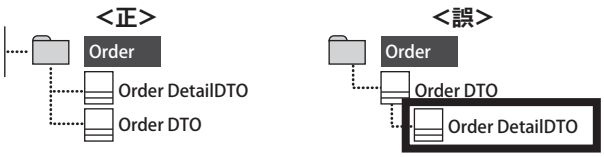

図 6. エラー抽出の例

(2) プロジェクト固有のモデル検証の容易さ

検証ツール作成にはUMLモデルの操作やRSAの高度な知識が必要であり個々のプロジェクトで作成するのは困難であるが，QCheckAMではUMLの知識と検証ルール定義の作成方法さえ理解すればよいため，容易に利用できる.

(3) 検証ルール定義の柔軟性

オブジェクト指向分析設計を実施する際の標準化，品質向上，工数削減をサポートするためのガイドと支援ツールであるGTAM（Guide and Toolkit for Application Modeling）で提供されるモデル品質チェック・リストをもとに，機械的に検証可能と思われる30のチェック内容をQCheckAMで実現した．そのうち，29個は3.3節で示した粒度のルール部品で実現することができ，UMLのメタモデルを単位としたルール部品を組み合わせることにより，柔軟に検証ルールの定義が可能であることが確認できた.

5. おわりに

大規模のオブジェクト指向開発において，モデル品質を確保するためのモデル検証フレームワークを提案し，実プロジェクトのモデル検証の事例を通してその有用性を確認した．本フレームワークを利用することにより以下のことが可能になる.

（1）プロジェクト固有の検証ルールの定義
（2）PCDAサイクルを通じたモデル品質向上
（3）容易な検証ルール定義

今後は，アーキテクチャー・テンプレートをモデルとして定義し，そのモデルから検証ルール定義を自動生成することで，アーキテクチャーの策定からアーキテクチャーのガバナンスを一連の流れで実施する技術や，日々モデルの検証を自動で行う環境を整えて品質の変化をモニタリングする技術に応用が可能と考えている.

謝辞

QCheckAMの開発にあたって，お忙しい中有益なご意見をくださったGTAM開発チームの皆様に感謝を申し上げます.

参考文献

［1］ David S.Frankel,日本アイ･ビー･エム TEC-J MDA分科会：MDAモデル駆動アーキテクチャ，SIBアクセス，ISBN4-4340-3813-3（2003）.
［2］ M.E. Fagan:“Design and code inspections to reduce errors in program development,” IBM Systems Journal,Vol.15,No.3,pp.182–211（1976）.
［3］ ロジャーS.プレスマン :実践ソフトウェアエンジニアリング-ソフトウェアプロフェッショナルのための基本知識- ,日科技連出版社，ISBN-13: 978-4817161482（2005）.
［4］ C. ジョーンズ:システム開発の生産性,マグロウヒルブック，ISBN-13: 978-4895010979（1986）.
［5］ Van Der Straeten, Mens Tom., Simmonds Jocelyn, and Jonckers Viviane:“Using Description Logic to Maintain Consistency between UML Models,” Proceedings of 6th International Conference on the Unified Modeling Language ,Vol.2863, pp.326-340（2003）.
［6］ 鷲見 毅,渡辺 晴美,大西 淳:“ステレオタイプによるUMLモデル間の整合性検証支援手法,” 情報処理学会論文誌Vol.43,No.6, pp.1554-1562（2002）.
［7］ F.ブッシュマン:ソフトウェア アーキテクチャ，近代科学社，ISBN4-7649-0283-4,（2000）.
［8］ Eclipse Modeling Framework Project（EMF）, http://www.eclipse.org/modeling/emf/
［9］ Object Constraint Language Specification, version 2.0, http://www.omg.org/technology/documents/formal/ocl.htmet
［10］ Aliki Tsiolakis and Hartmut Ehrig: "Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars," Proceedings of Workshop on Graph Transformation Systems, pp.77-86（2000）.
［11］ Wuwei Shen, Kevin Compton, and James Huggins: "A toolset for supporting UML static and dynamic model checking," Proceedings of 16th IEEE International Conference on Automated Software Engineering, pp.147-152（2001）.

日本アイ･ビー･エム株式会社
GBS プロセス･メソッド&ツールズ
ITスペシャリスト
新美 健一 Kenichi Niimi
［プロフィール］
2004年に日本IBM入社.ITスペシャリスト．オブジェクト指向分析設計のためのアセット「GTAM」の開発を担当.現在は,GTAMやモデル駆動型開発手法のプロジェクトへの展開に従事.
niiken@jp.ibm.com

日本アイ･ビー･エム株式会社
GBS プロセス･メソッド&ツールズ
主任ITスペシャリスト
大澤 浩二 Kohji Ohsawa
［プロフィール］
2000年日本IBM入社.主任ITスペシャリスト.オブジェクト指向分析設計のためのアセット「GTAM」の開発を担当.現在は,オブジェクト指向分析設計,モデル駆動型開発によるアプリケーション構築に従事.
oosawak@jp.ibm.com