B-021

ソフトウェア開発におけるデザイン進化のモデルの評価
Evaluating a Model for Design Evolution in Software Development

下滝 亜里† 青山 幹雄‡
Asato Shimotaki Mikio Aoyama

1. はじめに

ソフトウェアは，環境やユーザ要求の変化に適応して進化する必要があり，進化容易なソフトウェアの開発のためには，進化の原理の解明が必要である．

本稿では，ソフトウェアの構造的側面から進化を理解するために，構造を決める上位の抽象概念として，設計者が行う意思決定に着目する．意思決定の結果の集合を一般化し「デザイン」と呼ぶ．

以下では，環境やユーザ要求の変化に伴うデザインの変化をデザイン進化として捉え[8]，ソフトウェア進化の統一的なモデルを提案する．デザイン進化の例としてリファクタリング用い，提案モデルの妥当性を評価する．

2. 研究課題

デザインという抽象概念に基づき，ソフトウェア進化をモデル化するにあたっては，次の課題がある．

(1) デザインのモデル：ソフトウェア開発におけるデザイン自体のモデルが必要である．

(2) デザイン進化の表現：デザイン進化のモデルは，要求変化，デザイン変化，要求変化とデザイン変化の間の関係を表現する必要がある．

(3) デザイン進化の過程の表現：変化した要求を満たすには，多くの意思決定が必要である．本稿では，意思決定の過程をデザイン進化の過程と捉える．デザイン進化のモデルは，デザイン進化の過程を表現する必要がある．

3. アプローチ

上記の課題に対し，図 1 に示すように，本稿ではモデル駆動に基づくアプローチ[10]によりデザイン進化をモデル化する．以下はモデル化の手順である．

(1) デザインのメタモデル：特定のデザイン表現に依存しない，デザインのメタモデルを定義する．

(2) デザイン進化のモデル： 要求の変化を，ある要求からある要求への変換として考える．同様に，デザインの変化を，あるデザインからあるデザインへの変換として考える．また，要求を満たすデザインを得る過程を，要求からデザインへの変換として考える．

(3) デザイン進化の過程のモデル：デザイン進化の過程を，デザイン変換の連続的な適用の過程としてモデル化する．

(4) デザイン進化のメタモデル：上記のモデルに基づき，デザイン進化のメタモデルを定義する．

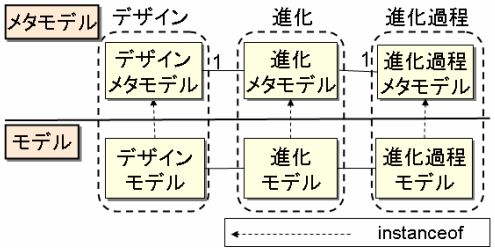

図 1 進化のモデルとメタモデル

4. デザインのメタモデル

従来の設計理論[2][3]に基づき，デザインのメタモデルを定義した (図 2).

(1) デザインプロセス：意思決定のプロセス[2]．ソフトウェアにおけるデザインプロセスとは，要求を満たす問題に対して，構造的側面から解決策を探索し，評価し，決定するプロセスである．プロセスの出力は，デザインである．

(2) デザイン：ソフトウェアの構造的側面に関して設計者が意思決定した結果の集合．

(3) デザインパラメータ：デザインを構成する単位[3]であり，設計者が意思決定する項目．デザインパラメータは，選択肢，もしくは値の集合を表す．たとえば「クラス」「メソッド」「属性」などはデザインパラメータと見なせる[7].

(4) デザイン構造：デザインパラメータとその間の物理的/論理的な依存関係[3]．依存関係には，たとえばメソッドの呼び出しや継承関係などがある．

(5) デザイン表現：特定の目的に合わせて，デザインをマッピングした表現．クラス図や Design Structure Matrix[3]などはそのような表現と見なせる．

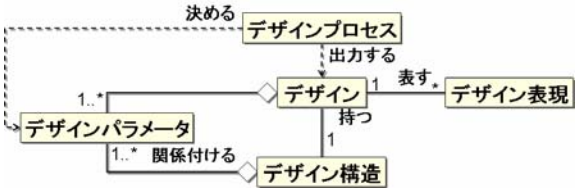

図 2 デザインのメタモデル

5. デザイン進化のモデルとメタモデル

5.1. デザイン進化のモデル

デザイン進化の一般的なモデルを図 3 に示す．本稿では，要求 R_n を集合として表現する．

$$R_n = \{r_n^1, r_n^2, ..., r_n^i, ..., r_n^k\} \quad (1)$$

r_n^i は，個々の要求を表す．

† 南山大学 大学院 数理情報研究科，Graduate School of Mathematical Sciences and Information Engineering, Nanzan University

‡ 南山大学 数理情報学部 情報通信学科，Faculty of Mathematical Sciences and Information Engineering, Nanzan University

要求 R_n から R_{n+1} の進化を，単一要求進化と呼ぶ．単一要求進化をモデル化するために要求進化オペレータ C_n を導入する．C_n は，要求 R_n に適用されるオペレータで，適用後の要求を R_{n+1} と定義する．

デザイン S_n から S_{n+1} の進化を，単一デザイン進化と呼ぶ．単一デザイン進化をモデル化するためにプリミティブデザイン進化オペレータ E_n^i を導入する．デザイン S_n から S_{n+1} へのデザイン進化の過程を，任意の数のプリミティブデザイン進化オペレータのシーケンスとして式 (2) で定義する．

$$E_n = E_n^1 + E_n^2 + \cdots + E_n^{m-1} + E_n^m \qquad (2)$$

ここで，E_n をデザイン進化オペレータと呼ぶ．E_n^i は，既存のデザイン S_n^{i-1} に適用されるオペレータであり，オペレータ適用後のデザインを S_n^i とする．E_n をデザイン S_n $(=S_n^0)$ に適用することによりデザイン S_{n+1} $(=S_n^m)$ が得られるとする．ここで m は，要求 R_{n+1} を満たすデザイン S_{n+1} を得るために必要なオペレータ数とする．

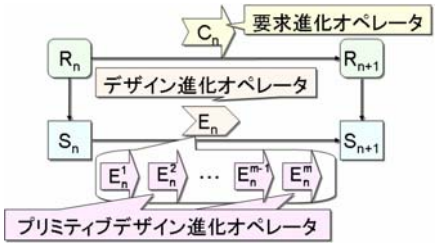

図 3 デザイン進化の一般モデル

5.2.デザイン進化オペレータのメタモデル

図 4 にデザイン進化オペレータのメタモデルを示す．プリミティブオペレータは，デザインを変化させる基本となるオペレータである．プリミティブオペレータを組み合すことで，粗粒度のオペレータであるコンポジットオペレータを表現する．

本稿では，デザインのメタモデルに基づき，以下の三つをプリミティブオペレータとして定義する．

(1) デザインパラメータの追加：新しいデザインパラメータをデザインに追加する．

(2) デザインパラメータの削除：既存のデザインパラメータをデザインから削除する．

(3) デザインパラメータ値の選択：あるデザインパラメータから値を選択する．もしくは，既存の値を変更する．

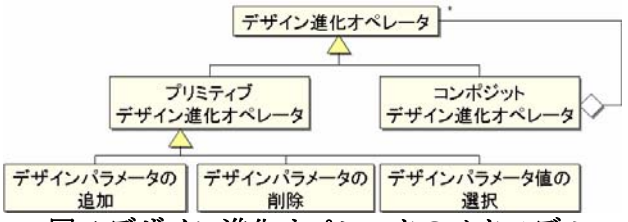

図 4 デザイン進化オペレータのメタモデル

5.3.デザイン進化のメタモデル

デザインのメタモデル，デザイン進化のモデル，デザイン進化オペレータのメタモデルに基づき，デザイン進化のメタモデルを定義した (図 5)．システムは，現在の要求とデザイン，そして一世代のシステム進化からなる．システム進化は，単一進化の集合からなる．システム進化は，特定の進化の表現により表される．単一進化は，単一要求進化と単一デザイン進化からなる．単一要求進化は，進化前と後の要求，その進化の過程を表す要求進化オペレータからなる．単一デザイン進化は，進化前と後のデザイン，その進化の過程を表すデザイン進化オペレータからなる．

要求やデザインは，特定の表現により表される．表現が決まることで，具体的なオペレータが決定される．次節では，木構造のグラフを用いた場合のデザインの表現と，木構造のグラフに対するオペレータを定義する．

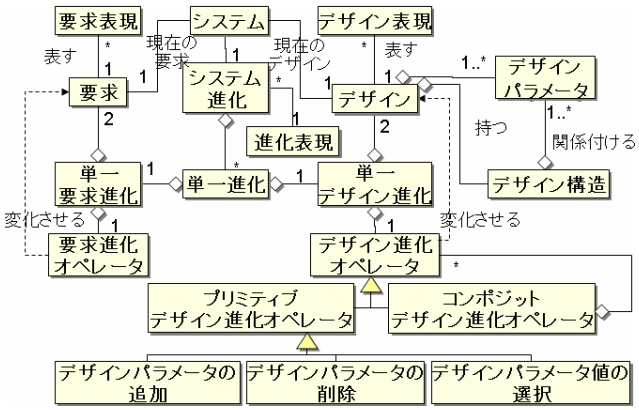

図 5 デザイン進化のメタモデル

6. デザインの表現とデザイン進化オペレータ

6.1.木構造のグラフによるデザイン表現

本稿では，特定の意味表現によらないデザイン表現として木構造のグラフを用いる (図 6)．以下では，[4]の記法を拡張した記法を用いる．L(x)は，ノード x のラベルを表す．ラベルには，たとえばクラス宣言を表す class_declaration などがある．ノードが持つデータを変数名 var とその値 val のペア(var, val)として表す． V(x, var)は，ノード x のデータ var の値を表す．たとえば，x がクラス宣言ノードである場合，V(x, name)は，クラスの名前を表す．ノード x がルートノードでない場合，x の親ノードを P(x)により表す．ノード $v_1,\ldots,v_m$ がノード u の子である時，v_k をノード u の k 番目の子と呼ぶ．

デザインのメタモデルでいえば，ノードとデータがデザインパラメータに対応する．

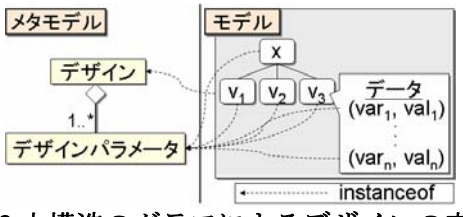

図 6 木構造のグラフによるデザインの表現

6.2.木構造のグラフに対するオペレータ

木構造の基本編集操作として以下が定義されている[4].

(1) ノード追加：ノード y の k 番目の子として，ラベル l とデータ集合 $D=\{(var_1, val_1),\ldots,(var_n, val_n)\}$ を持つリーフノードの追加を，INS((l, D), y, k)で表す．

(2) **ノード削除：**親 P(x) からのノード x の削除を DEL(x) で表す.

(3) **ノード移動：**ノード x がノード y の k 番目の子ノードになり，P(x)から削除されることを MOV(x, y, k)で表す.

(4) **値更新：**val によるデータ var の更新を UPD(x, var, val)で表す.

本稿では，これら編集操作を，それぞれ「ノード追加」「ノード削除」「ノード移動」「値更新」の四つオペレータとして定義する (図 7). ここで「ノード移動」に関しては，デザインのメタモデルで「デザインパラメータの移動」が存在しないため，「ノード追加」と「ノード削除」からなるコンポジットオペレータとして表す.

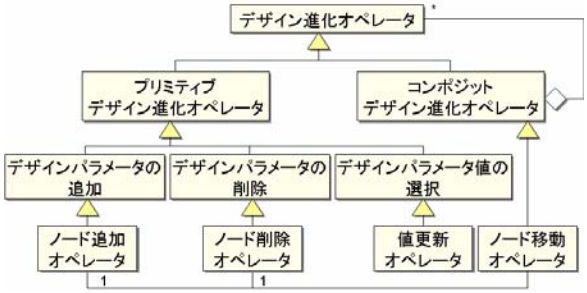

図 7 木構造に対するデザイン進化オペレータ

6.3.木構造による Java プログラム要素の表現

Java におけるプログラム要素をノードとして表し，各ノードが持つデータを定義する．プログラム要素としては，パッケージ宣言，インタフェース宣言，メソッド宣言，フィールド宣言を考えた．以下に，例としてクラス宣言ノードが持つデータを示す.

(1) **クラス名 (name)：**クラス名を表す.

(2) **アクセス修飾子 (access_ modifier)：**アクセス修飾子を表す.

(3) **final 修飾子 (final_ modifier)：**クラスが final かどうかを表す.

(4) **abstract 修飾子 (abstract_modifier)：**抽象クラスかどうかを表す.

(5) **static 修飾子 (static_modifier)：**static なメンバ・クラスかどうかを表す.

(6) **スーパークラス (super_class)：**当該ノードのスーパークラスを表す.

(7) **インタフェース (interfaces)：**当該ノードのインタフェースを表す.

6.4.Java プログラムに対するオペレータ

各宣言ノードに適用できるオペレータを定義する．以下に，クラス宣言ノードに適用できるオペレータの例を示す (図 8).

(1) **クラス追加オペレータ：**クラス宣言ノードを追加する．INS((class_declaration , {(name, n), (access_ modifier, acc_mod), (final_modifier, final_mod), (abstract_modifier, abst_mod), (static_modifier, stat_mod), (super_class, super), (interfaces, {$inter_1$,…, $inter_n$})}), y, k).

(2) **クラス削除オペレータ：**クラス宣言ノード x を削除する．DEL(x).

(3) **クラス移動オペレータ：**クラス宣言ノード x を移動する．MOV(x, y, k).

(4) **クラスのデータ更新オペレータ：**クラス宣言ノード x のデータ val の値を var で更新する．UPD(x, var, val).

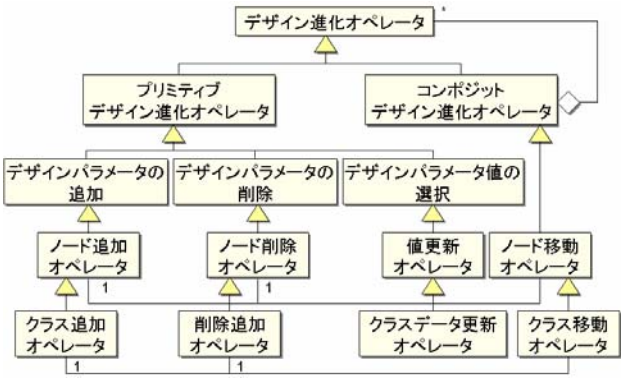

図 8 Java プログラムに対するオペレータの例

7. 例題

リファクタリング[8] の一つである「スーパークラスの抽出」を提案モデルにより表現できることを示す．図 9 に示すように，Salesman と Engineer のクラスがあるとする．Salesman と Engineer には，それぞれ自分の名前を表す name フィールドと，名前を返す getName メソッドが定義されているとする．ここで，name フィールドと getName メソッドの重複をなくすために共通のスーパークラス Employee を作成し，これらのフィールドとメソッドをこのスーパークラスに引き上げる.

このリファクタリングの適用によるデザイン進化は，以下のオペレータの連続的な適用過程として表現できる.

(1) 「クラス追加」オペレータを適用し Employee クラスを作成.

(2) 「スーパークラスの変更」オペレータを適用し，Salesman クラスノードの super_class データを Employee 値で更新.

(3) 「スーパークラスの変更」オペレータを適用し，Engineer クラスノードの super_class データを Employee 値で更新.

(4) 「フィールド移動」オペレータを適用し，Salesman の name フィールドノードを Employee ノードに移動.

(5) 「フィールド削除」オペレータを適用し Engineer ノードから name フィールドノードを削除.

(6) 「メソッド移動」オペレータを適用し，Salesman の getName メソッドノードを Employee ノードに移動.

(7) 「メソッド削除」オペレータを適用し，Engineer ノードの getName メソッドノードを削除.

8. 評価と議論

例題により，以下を確認した.

(1) デザインのメタモデルに基づくデザイン表現を用いて，ソフトウェアの構造をデザインの観点から形式的に表現できること.

(2) デザインの進化過程をデザイン進化オペレータの連続的な適用の過程として表現できること.

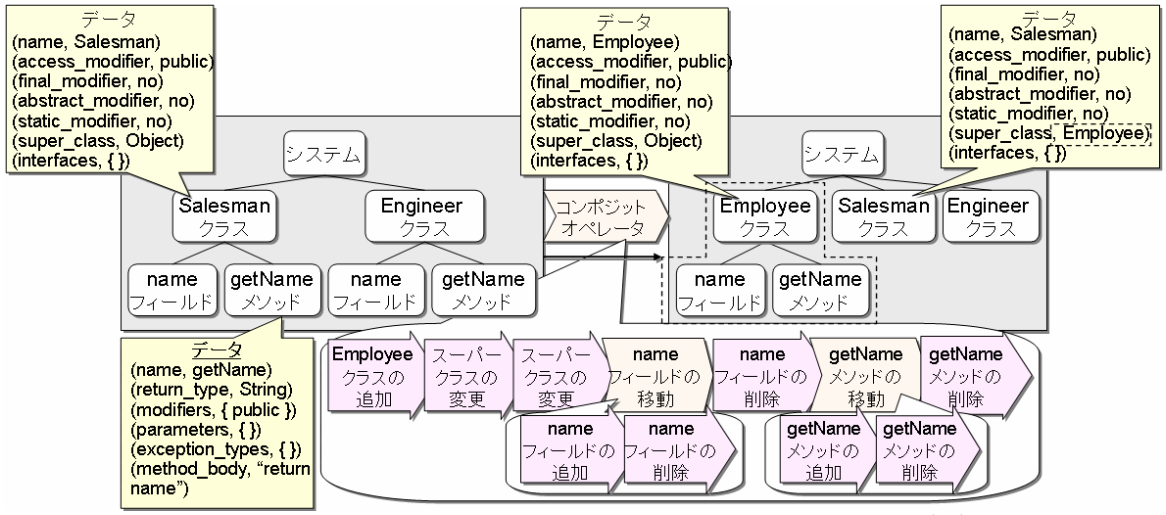

図 9 「スーパークラスの抽出」のリファクタリングの適用によるデザイン進化

(3) リファクタリングのような経験に基づくデザイン技術の適用を，デザインの進化過程として形式的に表現できること．

以上のことから，提案モデルがソフトウェア開発におけるデザイン進化を統一的かつ形式的に表現できると考えられる．

提案モデルを用いた具体的な適用としては，以下が考えられる．

(1) 提案モデルはモデル駆動に基づいているため，デザイン進化の自動化により，進化に必要な時間やコストの削減が期待できる．

(2) オペレータの適用数などの観点から，デザイン進化のコストや複雑性の定量的な測定が行える．

9. 関連研究

従来のデザイン進化の研究は，リファクタリング[9]やデザインパターン[1]など異なる観点から行われている．しかし，本稿で提案したような統一的な進化モデルは提案されていない．

リファクタリングの観点からは，文献[9]でデザイン進化が議論されている．プリミティブなリファクタリングの集合を提案し，その連鎖でソフトウェア進化を実現する．デザインの表現には，クラス図を用いている．

デザインパターンの観点からは，文献[1]でデザイン進化を，デザインパターン間の進化として捉えている．進化の過程を表現するために，「置き換え」と「接続」の二つのパターン進化オペレーションが定義されている．

文献[7]では，DSM[3]をデザイン表現に用いている．デザイン進化をモジュラオペレータ[3]の適用の観点から議論している．モジュラオペレータとは，モジュール化された構造を作る，もしくはそのような構造に対して適用される行為である[3]．モジュールの追加や削除などのオペレータが定義されている．モジュラオペレータのコンセプトは一般的であるが，ソフトウェアのデザインへの適用は十分に議論されていない．

10. まとめと今後の課題

本稿では，デザインと進化のメタモデルに基づき，ソフトウェアの構造的進化を理解するための統一的な進化モデルを提案した．リファクタリングを進化の例として用い，提案モデルの妥当性を示した．

今後の課題として，他の例題によるモデルの妥当性のさらなる検証とオペレータ適用順序の分析を検討する．

参考文献

[1] M. Aoyama, Evolutionary Patterns of Design and Design Patterns, Proc. of ISPSE ‘00, Nov. 2000, pp. 110-116.

[2] C. Y. Baldwin, Steps Toward a Science of Design, NSF PI Conference on the Science of Design, Feb. 2007.

[3] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*, MIT Press, 2000.

[4] S. Chawathe, et al., Change Detection in Hierarchically Structured Information, Proc. of the ACM SIGMOD COMAD, Jun. 1996, pp. 493-504.

[5] M. Fowler, et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[6] E. Gamma, et al., *Design Patterns*, Addison-Wesley, 1995.

[7] C. V. Lopes and S. Bajracharya, Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value, *Trans. on AOSD I*, LNCS 3880. Springer, 2006, pp. 1-35.

[8] 下滝 亜里，青山 幹雄，ソフトウェアデザインにおける進化モデルの枠組み，情報処理学会第 154 回ソフトウェア工学研究会，Vol. 2006-SE-154, No. 12, Nov. 2006, pp. 73-80.

[9] L. Tokuda and D. Batory, Evolving Object-Oriented Designs with Refactorings, *J. of Automated Software Eng.*, Vol. 8, No. 1, Jan. 2001, pp. 89-120.

[10] A. van Deursen, et al., Model-Driven Software Evolution: A Research Agenda, *Proc. of MoDSE 2007*, Mar. 2007, pp. 41-49.