

## ソフトウェアデザインにおける進化モデルの枠組み

下滝 亜里† 青山 幹雄‡

†南山大学 大学院 数理情報研究科

‡南山大学 数理情報学部 情報通信学科

環境や要求の変化に対して進化容易なソフトウェアの開発には、要求変化とそれに伴うデザイン構造の変化の理解が重要である。しかし、この関係についての統一的なモデルや枠組みは議論されていない。本稿では、要求変化とデザイン構造の変化の関係をデザイン進化として捉え、デザイン進化のモデルと枠組みを提案する。提案モデルをデザイン進化の従来研究に適用し、モデルの有効性を評価する。

### A Model and Framework of Evolution in Software Design

Asato Shimotaki†, Mikio Aoyama‡

†Graduate School of Mathematical Sciences and Information Engineering,

‡ Dept. of Information and Telecommunication Engineering, Nanzan University

To easily develop evolvable software, It is critical to understand the relationships between requirements change and design structure change. However, there is no unified model and framework for such relationships. In this article, we propose a model and framework of design evolution. We define design evolution as the relationships between requirements change and design structure change. To evaluate the effectiveness of the proposed model, we apply the model to conventional works on the design evolution.

#### 1. はじめに

環境や要求の変化に対応するにつれて、ソフトウェアの構造は複雑になり、理解が困難となることや、拡張性、柔軟性、保守性などの品質特性が低下することが知られている[23][39][40][9]。そのような品質特性の低下は、ユーザ要求に迅速に対応すること困難にし、市場を損なう。そのため、環境や要求の変化に適応して容易に進化できるソフトウェアが求められている。

進化容易なソフトウェアの開発には、デザインが重要な役割を果たす[33]。変更や進化を考慮していないデザインは、ソフトウェアエイジングなどの問題の原因となる[33]。

そのため、進化を中心に置いたデザイン技術が必要である。しかし、保守活動、デザイン活動、ソフトウェアの構造との関係は整理されていない。リファクタリング[11][30]は、予防保守におけるデザイン技術である。デザインパターン[12]は、改良保守を容易にするための技術である。これらの技術は、特定の要求変化を基に、ソフトウェアの構造を変化させるプロセスである。本稿では、要求変化によるソフトウェア構造の変化を、デザイン進化としてモデル化することを提案する。

2 節では、背景と従来研究の問題点を述べる。3 節では、問題点に対するアプローチとしてデザイン進化のモデルを提案する。4 節では、デザイン進化の研究の枠組みを提案する。5 節は今後の課題である。6 節はまとめである。

#### 2. 問題の背景と従来研究

##### 2.1. ソフトウェアのデザイン

ソフトウェアのデザインプロセスとは、ユーザ要求を満たす問題に対する解決策を探索するプロセスである。デザインとは、プロセスの出力であり、ソフトウェアの記述である。デザインパラメータ[5]とは、デザインを構成する要素である。クラス、メソッド、属性などがデザインパラメータと見なせる[26]。図-1 にこれらの関係を示す。

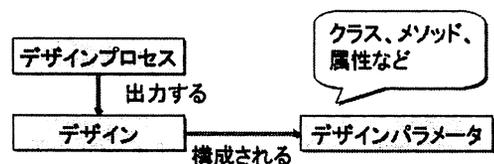


図-1 デザイン、デザインプロセス、デザインパラメータ

デザイン構造とは、デザインパラメータとその間の関係である。依存関係や継承関係などがある。表現可能なデザイン構造は、プログラミング言語[20][4]などのモジュール化技術に依存する。図-2 は Strategy パターンと Observer パターンから成るデザイン構造を示している。

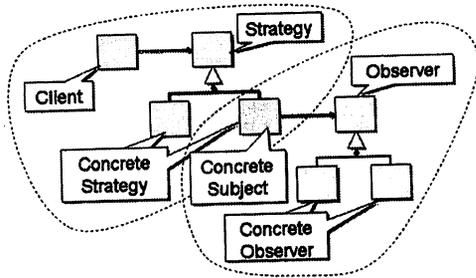


図-2 デザイン構造

## 2.2. ソフトウェア進化と保守

ソフトウェア進化とは、ソフトウェアへの変更の結果である。通常、変更は、ソフトウェア納入後の活動であるソフトウェア保守[37][24]により行われる。保守活動として以下が知られている。

- (1) 修正保守: バグや欠陥を修正する。
- (2) 適応保守: 新しい環境(OS やプラットフォーム)への適応。
- (3) 改良保守: 新しい機能的・非機能的要求の実現。
- (4) 予防保守: 将来起こるかもしれない問題を予防するために、ソフトウェアの構造を改善する。たとえばリファクタリング。

また、これら変更によって起こる法則や現象が知られている。ソフトウェア進化における法則として、複雑性の増大(法則 2)や品質の低下(法則 7)が知られている[23]。現象としては、ソフトウェアエイジング[33]、デザインの劣化[39][40]、コードデコイ[9]などがある。これらの現象の症状として、性能の低下、変更時における新たなエラーの導入、繰返し文や条件文がネストすることによるコードの複雑性の増大が指摘されている。

## 2.3. 問題点

ソフトウェアエイジングなどの問題を防ぐには、デザイン技術が重要な役割を果たす[33]。デザイン技術は問題を防ぐだけでなく、要求変化や進化を中心に置き、容易に進化できるソフトウェアを開発するためのデザイン技術が必要である。

要求変化に対するソフトウェアの進化容易性は、デザイン構造に依存する[28]。そのため、要求変化とデザイン構造の変化の関係の理解が重要となる。しかし、従来研究では、要求変化の種類やそれら個別の要求変化に伴うデザイン構造の変化の関係について議論されることが少ない。また、デザイン構造の変化をデザイン進化として議論する従来研究においても、リファクタリング[38]、デザインパターン[2]、価値追及[5]などの特定の観点から議論されており、統一的なモデルや枠組みがない。要求変化によるデザイン構造の変化を、デザイン進化としてモデル化することが必要である。

## 2.4. アプローチ

デザイン進化に関する定義や概念も統一されていないため、本稿では、デザイン進化の概念整理を行い、デザイン進化のモデルを提案する。

## 3. デザイン進化のモデル

### 3.1. デザイン進化のモデル

デザイン進化は要求の変化により起こると考える。要求変化をモデル化するために、要求変化オペレータ  $C_n$  を導入する。要求変化オペレータ  $C_n$  は、要求  $R_n$  に適用されるオペレータであり、オペレータ適用後の要求を  $R_{n+1}$  と定義する。デザイン進化の一般的なモデルを図-3に示す。

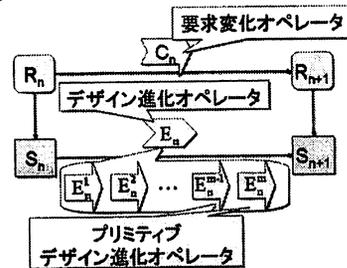


図-3 デザイン進化の一般モデル

デザイン進化をモデル化するために、デザイン構造を変化させるオペレータである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。デザイン進化を、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:

$$E_n = E_n^1 + E_n^2 + \dots + E_n^{m-1} + E_n^m \quad (1)$$

の適用であると定義する。  $E_n$  をデザイン進化オペレータ、あるいは、単にデザインオペレータと呼ぶ。デザイン進化オペレータ  $E_n^i$  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。デザイン進化オペレータ  $E_n$  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  $S_{n+1}$  を得るのに必要な任意のオペレータ数とする。なお、以下では簡単のため、  $S_n$  と  $S_{n+1}$  の代わりに、それぞれ  $S$ 、  $S'$  等を用いる。  $E_n$  や  $R_n$  に関しても同様である。

### 3.2. 要求変化の分類

本稿では、要求を外発的要求と内発的要求に分類する(表 1)。外発的要求 ER (Extrinsic Requirements) とは、ユーザがソフトウェアに望んでいる機能的・非機能的な要求である。内発的要求 IR (Intrinsic Requirements) とは、設計者がソフトウェアに要求する品質や補助的な機能(ロギング機能や統計データの取得など[26])である。

外発的要求と内発的要求を変化させるオペレータを、それぞれ、EC、IC として定義する。要求変化オペレータのメタモデルを図-4に示す。

表 1 外発的・内発的要素

外発的要素	新しい機能の追加, 既存の機能の修正, パフォーマンスの改善など
内発的要素	再利用性・拡張性の向上, 理解容易性の向上, コードの重複の削除, 補助的な機能の導入など

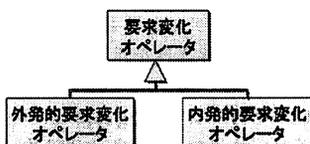


図-4 要求変化オペレータのメタモデル

### 3.3. デザイン進化の分類

デザイン進化を以下の3つの観点から分類する。

- (1) 改善
- (2) 拡張
- (3) 適応

内発的要素の変化により, 改善/拡張のためのデザイン進化が起こると定義し, 外発的な要素の変化により適応のためのデザイン進化が起こるとする。

以下では, 3つのデザイン進化について述べる。

#### 3.3.1. 改善のためのデザイン進化

改善のためのデザイン進化は, 通常リファクタリングにより実現される(図-5)。

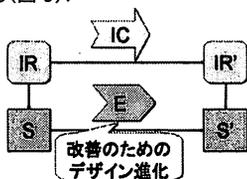


図-5 改善のためのデザイン進化

リファクタリングとは「外部から見たときの振る舞いを保持し, 理解や修正が容易になるように, ソフトウェアの内部構造を変更すること」である[11][30]。リファクタリングは, 「コードの不吉な匂い」を動機に行われる[11]。重複したコード, 長すぎるメソッド, 巨大なクラスなどである[11]。

文献[11]でカタログ化されているようなリファクタリングの全てが改善のためのデザイン進化を引き起こすとは限らない。たとえば, 「説明用変数の導入」「アルゴリズムの取替え」などのリファクタリングは, デザイン構造を変えるような変更, つまりモジュール間の関係(たとえば依存関係)を変える変更ではないため, デザイン進化と呼ばない。

#### 3.3.2. 拡張のためのデザイン進化

拡張のためのデザイン進化とは, 将来発生するかもしれない外発的要素変化を考慮したデザイン構造の変化である(図-6)。たとえば, 現在用いているライブラリでは負

荷の増大に伴い性能が問題となる場合に備えて, 異なるライブラリを使用できるようにあらかじめデザインを変更することである。

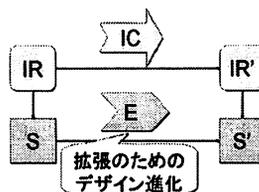


図-6 拡張のためのデザイン進化

#### 3.3.3. 適応のためのデザイン進化

適応のためのデザイン進化とは, 外発的要素の変化を満たすためにデザインを変更することである(図-7)。

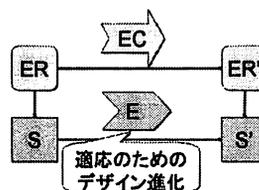


図-7 適応のためのデザイン進化

たとえば, 負荷の増大に伴う性能低下により, デザインの変更が要求される。具体例としては, パターン認識システムでは, 認識結果をより詳細に分析できるように, 新たなログ機能が要求されることや, 計算負荷の軽減のために必要なログ機能を選択できることが要求されることがある。他には, 外発的要素の変化により, State パターンを用いたデザイン構造から, Command パターンを用いたデザイン構造への変更が考えられる[39]。

### 3.4. 適応したデザイン

外発的要素変化  $ER \rightarrow ER'$  に対して変更方法が決まっているデザイン  $S$  を, 適応したデザインと呼ぶ(図-8)。

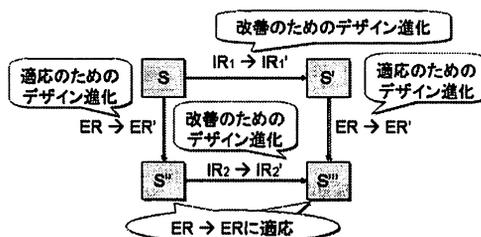


図-8 適応したデザイン

たとえば, Strategy パターンが適用されたデザイン構造  $S'$  に対して, 新しいアルゴリズム (ConcreteStrategy) の追加  $ER \rightarrow ER'$  が起こるとする。通常, Strategy interface を実装するクラスの導入によりこの要求を満たせる。このとき, デ

ザイン  $S'$  は、 $ER \rightarrow ER'$  に対して適応していると呼ぶ。一方、条件文によってアルゴリズムの選択を行うデザイン  $S$  においても、アルゴリズム追加の方法は決まっているため（新しい条件文の追加）、 $ER \rightarrow ER'$  に対して適応していると考ええる。

### 3.5. 適応できないデザイン

図-9 に示すように、外発的的要求変化  $ER \rightarrow ER'$  に適応できないデザインが存在する。

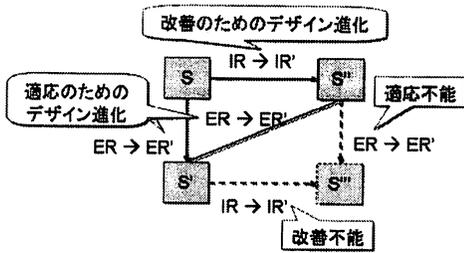


図-9 適応できないデザイン

$S$  を従来の Strategy パターンのデザインであるとする。デザインパターンの実装、つまりデザイン構造は、用いる言語に依存することが知られている。たとえば AspectJ[20] では、局所性、再利用性、合成透過性などのモジュール特性の向上を動機に、従来の Strategy パターンの構造を改善できる[15]。IR  $\rightarrow$  IR' により  $S \rightarrow S''$  となる。ここで、ある外発的的要求が追加され、 $ER \rightarrow ER'$  となる。 $S''$  は、この要求に適応できないため、 $S$  を通って、 $S'$  のデザインに到達する必要がある。また、 $S'$  のデザインは、モジュラリティの点で問題があるが要求  $ER'$  を満たす必要があるため、デザイン  $S'''$  へのデザインの改善はできない。

このように、モジュール特性の向上は、リファクタリングの動機となるが、要求の変化に対応できないデザインが存在すると考えられる。

### 3.6. デザイン進化の関係

図-10 にデザイン進化の関係の一例を示す。

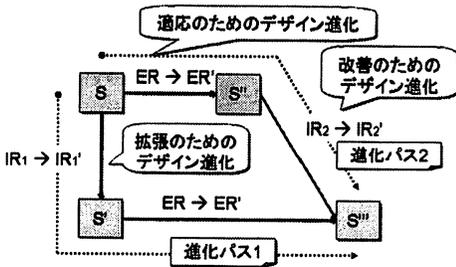


図-10 デザイン進化間の関係

$S \rightarrow S' \rightarrow S'''$  と  $S \rightarrow S'' \rightarrow S'''$  の二つの進化パスがある。パス 1 では、まず、将来の外発的的要求変化 ( $ER \rightarrow$

$ER'$ ) に適応するようにデザインを  $S$  から  $S'$  に変化させている。要求変化  $ER \rightarrow ER'$  が起こり、 $S'$  は  $S'''$  に変化する。

パス 2 では、要求変化  $ER \rightarrow ER'$  を満たすために、デザインが  $S$  から  $S''$  に変化している。適応後のデザインを改善することで  $S'' \rightarrow S'''$  の変化が起こる。

### 3.7. モデルの適用と評価

デザイン進化のモデルを従来のデザイン進化の研究に適用し、モデルの妥当性を評価する。デザイン進化に関する従来研究として、リファクタリング[38]、デザインパターン[2]、モジュラオペレータ[5][26]の三つを取り上げる。これらは、それぞれ異なる適用条件や目的・動機により適用され、デザイン構造を変化させるため、デザインオペレータの一種と見なせる(図-11)。

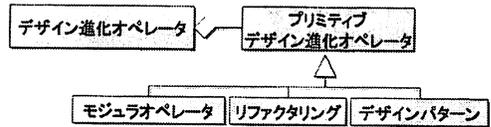


図-11 デザイン進化オペレータのメタモデル

#### 3.7.1. リファクタリング

リファクタリングとは、その定義から、内発的的要求の変化によって起こり、プログラムの振る舞いを保持したままデザイン構造を変更するデザインオペレータである。その結果は、改善のためのデザイン進化であると言える。

リファクタリングが、目的、引数、説明、適用条件、初期状態、目標状態の 6 つの項目により定義されている[38]。例として、以下に inherit[Base, Derived]リファクタリングを示す。ただし、紙面の都合のため 7 つの適用条件中一つだけ載せる。

- (1) 目的: 既存のクラス間に、スーパークラス・サブクラス関係を導入する
- (2) 引数: Base (スーパークラスの名前), Derived (サブクラスの名前)
- (3) 説明: Base を Derived のスーパークラスにする
- (4) 適用条件: Base は、Derived のサブクラスであってはならない。Derived は、Base のスーパークラスであってはならない。
- (5) 初期状態・目標状態: 図-12。

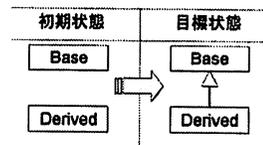


図-12 Inherit[Base, Derived]リファクタリング

リファクタリングは、プリミティブリファクタリングとコンポジットリファクタリングに分類される[30]。プリミティブリファクタリングは、振る舞い保持の基本となる変換である。たとえば

Inherit リファクタリングである。コンポジットリファクタリングは、通常、プリミティブリファクタリングのシーケンスとして定義される。たとえば、メソッド名の変更やメソッドの追加などプリミティブリファクタリングを行い、Visitor パターンを導入し、デザインを改善する例がある[30]。また、データベースのスキーマ変換に基づくリファクタリングと C++ に特化したリファクタリングのシーケンスの適用により、フレームワークを進化させる事例が報告されている[38]。

本稿のモデルで言えば、デザイン進化は要求変化により起こるため、デザインオペレータとしてのリファクタリングは、要求を満たす任意の数のプリミティブリファクタリングあるいはコンポジットリファクタリングのシーケンスと見なせる。これらの関係を図-13 に示す。

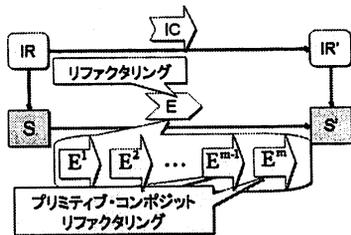


図-13 リファクタリングオペレータ

文献[38]では、デザインが進化する理由として、機能性(機能追加と変更)、再利用性, 拡張性, 保守性の4つが挙げられている。しかし, [38]の焦点は, リファクタリング作業をツールにより自動化しようとするものであり, デザイン進化自体ではない。たとえば, 新しい機能の追加や既存の機能の修正や変更により, デザインがどのように変化・進化するかについては詳しく触れていない。

### 3.7.2. デザインパターン

文献[2]では、デザインパターンは問題空間と解空間のマッピングであると定義され、デザインパターンの観点からデザイン進化が定義されている。また、intensive evolution と extensive evolution にデザイン進化が分類されている。intensive evolution は、問題空間における内部変化により起こる。要求の変化、バグ修正、デザインの改善などがそのような変化に相当する。extensive evolution は、問題空間の拡張により起こる。新しい要求の追加や実行環境を考慮することなどを拡張例として挙げている。

したがって、デザインパターンは、機能追加の時だけでなく、拡張性や構造の改善が必要な時などに適用されるため、デザインオペレータであると見なすことができる。

たとえば、プリミティブリファクタリングのシーケンスの適用により Visitor パターンを導入し、デザインを改善する例が議論されている[38]。したがって、Visitor パターンへのリファクタリングをデザインオペレータと見なせる(図-14)。

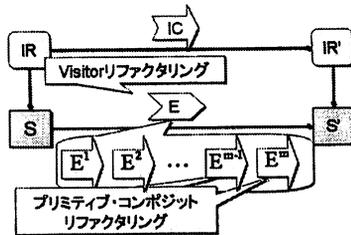


図-14 Visitor リファクタリングオペレータ

Visitor リファクタリングオペレータのメタモデルを図-15 に示す。

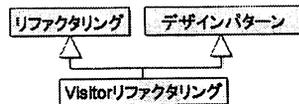


図-15 Visitor リファクタリングオペレータのメタモデル

### 3.7.3. モジュラオペレータ

デザイン進化が価値追及のプロセスとして定義されている[5]。デザイン構造を変化させる6つのモジュラオペレータ (Splitting, Substitution, Augmentation, Excluding, Inverting, Porting) が定義されている。この研究は、ソフトウェアデザインに適用された[35]。しかし、対象はコンピュータ産業であったため、ソフトウェアデザインに適用するには、有効性の検証が必要である。たとえば、アスペクト指向デザインの観点から、7つ目のモジュラオペレータとして Reversion が提案されている[26]。さらに、ケーススタディを用いてデザイン進化がモジュラオペレータの適用の観点から議論されている[26][35]。しかし、焦点はデザイン進化自体ではない。ソフトウェアのデザインを DSM (Design Structure Matrix)を用いて表現すること、NOV (Net Option Value)のモデルを用いて定量的に評価することにある。

モジュラオペレータをソフトウェアデザインに適用する試みはまだ初期段階だと言えるが、モジュラオペレータは、デザインオペレータの一種と見なすことができる。たとえば、アスペクト指向によるモジュール化を Reversion オペレータの適用と見なせる[26]。設計階層ダイアグラム (Design Hierarchy Diagram) [5]を用いて Reversion オペレータの適用例を図-16 に示す。図は、ロギング機能をアスペクトによりモジュール化する例である。

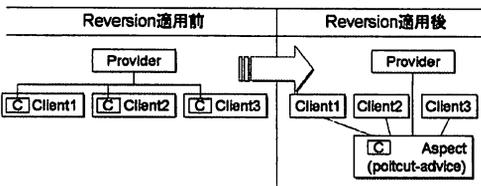


図-16 Reversion オペレータ[26]

設計階層ダイアグラムは、デザインパラメータ間の依存関係を示す。図-16のCはProviderへのアクセスを表すオペレータ適用前では、Clientは、Providerに依存している。Reversion オペレータの適用、つまりアスペクトの適用により、ClientとProvider間の依存関係は解消され、代わりにAspectが導入され、ClientとProviderへの依存関係ができる。

ロギングのような横断的関心事をアスペクトによってリファクタリングする例が報告されている[36][8]。図-16のような単純な例では、Reversion オペレータをアスペクト指向によるリファクタリングだと見なせる(図-17)。また、定義によりReversion オペレータはデザインオペレータである。なお、Reversion オペレータはリファクタリングであるため、プリミティブリファクタリング、あるいはコンポジットリファクタリングのシーケンスで構成されるが、アスペクト指向におけるリファクタリング[31]は研究の初期段階であるため、詳細な分析は今後の課題とする。

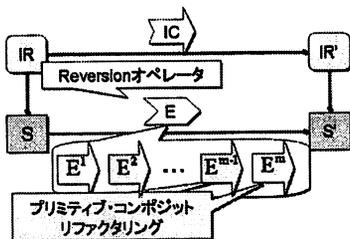


図-17 Reversion オペレータ

図-18 に reversion オペレータをメタモデルの観点から示す。

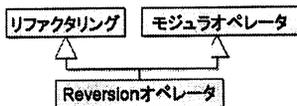


図-18 Reversion オペレータのメタモデル

アスペクト指向におけるリファクタリングカタログ[31]や、デザインパターンをアスペクト指向によりリファクタリングする研究がある[16][31]。Reversion モジュラオペレータ、リファクタリング、デザインパターンの間の関係のデザイン進化のモデルの観点からの詳細な分析は今後の課題とする。

#### 4. デザイン進化研究の枠組み

デザイン進化研究の枠組み(図-19)を基に、提案したデザイン進化のモデルと従来研究の関係を議論する。

デザインの構造的側面から進化の観察を行うことや、進化をモデル化する研究がある。また、モジュール化の技術の観点から、デザイン構造を改善し、要求変化に対する進化容易性を向上させる研究がある。繰り返し起こるデザイン構造をパターンとして明示的に捉える研究、あるい

は、再構築により構造を改善する技術がある。一方、ユーザに対してソフトウェアが提供するサービスの進化を調査する研究がある。以下では、これらの研究について、デザイン進化のモデルの観点から議論する。

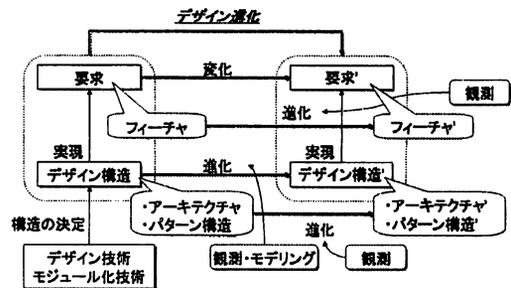


図-19 デザイン進化の枠組み

#### 4.1. ソフトウェア進化

ソフトウェアのリリース毎のモジュール数の変化の調査がある[23]。モジュール数の変化履歴をデザイン変化・進化の分析に用いるのは困難であると考えられる。

ソフトウェアが変更されるにつれて発生する現象として、ソフトウェアエイジング[33]、デザインの劣化[39][40]、コードデコイ[9]などが報告されている。これらの現象における症状、原因、予防策などが議論されている。症状は、性能の低下や変更時における新たなエラーの導入などである。原因としては、デザイン原則の違反、時間制約、要求の変更などが挙げられている。予防策として、変更を考慮してデザインすることやデザインレビューが提案されている。

ソフトウェア進化研究において、今後取り組むべき18種の研究課題が提案されている[29]。その内のいくつかは、進化とデザインに関わる課題であるが、本稿で提案したようなデザイン進化については言及されていない。

#### 4.2. 進化のモデリング

グラフ理論の観点からソフトウェア進化のモデルが提案されている[34][6]。文献[6]では、パラメータと戻り値、メソッドと属性、クラスの三つのレベルの進化が議論されている。本稿では、要求変化を外発的・内発的的要求に分類すると共に、デザイン進化に関しても分類したが、これらの研究ではそのような観点で議論されていない。

#### 4.3. モジュール化技術と進化容易性

図形エディタの例を用いて、実装の局所性、実装が明示的(Explicit)か暗黙的(Implicit)かの度合い、進化容易性の観点から7つの異なる実装の分析が行われている[21]。10個の個々の変更タスクを各実装に行うことにより、変更の容易性を評価している。変更タスクは、ダブルバッファリング、ロギング機能の追加、Shape サブクラスとしてCircleの追加、メソッドの名の変更、関係のないクラスやメソッドの追加などである。

文献[32]でも同様の分析が行われている。5つの変更

シナリオを基に、オブジェクト指向とアスペクト指向の実装の比較が行われている。変更シナリオは、オブジェクトのグラフの変更、FigureElement のサブクラスとして Circle の追加、制御フローの変更、FigureElement への color フィールドの追加などである。

XPI (Crosscut Programming Interface)と呼ばれるアスペクト指向デザインの方法が提案されている[14]。XPI はアドバイスが適用されるコードであるベースコードとアスペクトコードの間のインタフェースとなるデザインルール[5]であるため、従来のベースコードがアスペクトコードを意識しない obliviousness アプローチ[10]と比べて、変更を容易にする。XPI と従来のアプローチの比較のため、図形エディタの例と二つの変更シナリオが用いられている。一つ目は、Point クラスのフィールドの public から private への変更である。もう一つは、color 属性の追加である。

これらの研究では、多くの場合、デザイン構造に対する要求変化は一度だけであり、段階的な要求変化とそれに伴うデザイン構造の変化の関係について議論されることは少ない。

#### 4.4. ソフトウェアパターンと進化

リファクタリングによりデザインパターンが形成されていく過程がカタログ化されている[18]。しかし、パターン適用後のデザイン構造の変化には触れられていない。たとえば、要求変化のため、State パターンから Command パターンへの変化が考えられるが[39]、このようなデザインの変化については述べられていない。

デザインパターン間の進化のパターンが議論されている[2]。しかし、そのようなパターン進化のパターンについての評価は行われていない。たとえば、AbstractFactory パターンから Builder パターンへのマイクロ進化がどのような要求変化により起こるのかは具体的に述べられていない。

マイクロパターンの進化とバグ発生の関係性が調査されている[22]。マイクロパターンは、デザインパターンと異なり、抽象度が低く、ソースコードやバイトコードから自動検出可能な単一のクラスレベルのパターンである[13]。また、マイクロパターンは単一のクラスを特徴付けるパターンであるため、複数のクラスが役割を持つデザインパターンと異なり(マイクロ)アーキテクチャを構成しない。

デザイン進化の観点で言えば、マイクロパターンの進化や進化パターンの調査は、デザインパターンのデザイン構造の進化の調査よりも有益でない。従来のデザインパターンと異なりマイクロパターンは、デザインの課題を表現しないため、特定のマイクロパターンから、どのようなデザインの課題があったのかを読み取るのは難しい[13]。そのため、マイクロパターンは、デザインオペレータと見なせないと言える。

#### 4.5. アーキテクチャと進化

アーキテクチャ進化を中心に置いた開発プロセスが提案されている[7]。また、アーキテクチャ進化をケーススタ

ディとして、デザイン劣化の現象が報告されている[40]。

アーキテクチャレベルにおけるリファクタリングカタログが提案されている[25]。リファクタリングを行うきっかけとしてコードの不吉な匂い(Code Smell)が用いられているように、アーキテクチャレベルでも不吉な匂いがあるとしている。それら不吉な匂いに対応する形で、大きなリファクタリング(Large Refactoring)がカタログ化されている。

本稿では、デザインオペレータの適用により、デザイン構造の変化が起こるとした。具体的なデザインオペレータとして、リファクタリング、デザインパターン、モジュラオペレータを議論した。しかし、アーキテクチャレベルの進化を必要とするような要求変化の場合には、これらのデザインオペレータの適用では不十分かもしれない。

#### 4.6. フィーチャと進化

サービス、あるいは、ユーザに見えるフィーチャの進化が研究されている[1][17]。たとえば、Microsoft Word の 3 つのバージョン(2.0, 95, 97)におけるフィーチャ進化が報告されている[17]。これらの研究では、フィーチャ進化がデザイン構造に与える影響について詳しく議論されていないが、プロダクトラインの観点からフィーチャ進化が述べられている。プロダクトラインでは、システムは、フィーチャの組合せから成る。そのため、サービス進化の特徴を捉えておくこと役立つ[1]。一方、実装技術の観点からは、フィーチャをモジュール化の単位として扱うプログラミング言語が提案されている[27][3]。したがって、フィーチャ進化とそれらの言語によるデザイン構造の進化の関係の調査が考えられる。

### 5. 今後の課題

今後の課題としては、モデルの改良がある。まず、デザインオペレータとしてのデザインパターン、リファクタリング、モジュラオペレータ間の関係を明確にする必要がある。次に、具体的な要求変化を満たすデザインオペレータの特定が必要である。特に、アーキテクチャレベルの要求変化に適したデザインオペレータが必要である。特定後、オペレータの自動適用による開発支援が期待できる。

### 6. まとめ

本稿では、要求変化に伴うデザイン構造の変化をデザイン進化として捉え、デザイン進化のモデルと枠組みを提案した。要求変化を外発的・内発的要求に分類し、それら個別の要求変化の観点から、デザイン進化の分類を行った。デザイン進化の従来研究にモデルを適用し、有効性を確認した。

### 参考文献

- [1] A. I. Antón and C. Potts, Functional Paleontology: The Evolution of User-Visible System Services, IEEE Trans. on Software

- Eng. Vol. 29, No. 2, Feb. 2003, pp. 151-166.
- [2] M. Aoyama, Evolutionary Patterns of Design and Design Patterns, Proc. of ISPSE '00, 2000, pp. 110-116.
- [3] S. Apel, et al., Aspectual Mixin Layers: Aspects and Features in Concert, Proc. of ICSE '05, 2005, pp. 122-131.
- [4] I. Aracic, et al., An Overview of CaesarJ, Trans. on AOSD I. LNCS 3880. Springer, 2006, pp. 135-173.
- [5] C. Y. Baldwin and K. B. Clark, Design Rules: The Power of Modularity, MIT Press, 2000.
- [6] S. Ciraci and P. van den Broek, Modelling Software Evolution using Algebraic Graph Rewriting, Proc. of ACE '06, 2006, <http://www.cs.rug.nl/~paris/ACE2006/>.
- [7] M. Christensen, et al., Design and Evolution of Software Architecture in Practice, Proc. of TOOLS Pacific '99, 1999, pp. 2-15.
- [8] A. Colyer and A. Clement, Large-scale AOSD for Middleware, Proc. of AOSD '04, 2004, pp. 56-65.
- [9] S. G. Eick, et al., Does Code Decay? Assessing the Evidence from Change Management Data, IEEE Trans. on Software Eng. Vol. 27, No. 1, Jan. 2001, pp. 1-12.
- [10] R. E. Filman and D. P. Friedman, Aspect-Oriented Programming is Quantification and Obliviousness, Aspect-Oriented Software Development, Addison-Wesley, 2005, pp. 21-35.
- [11] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
- [12] E. Gamma, et al., Design Patterns, Addison-Wesley, 1995.
- [13] J. Y. Gil and I. Maman, Micro Patterns in Java Code, Proc. of OOPSLA '05, 2005, pp. 97-116.
- [14] W. G. Griswold, et al., Modular Software Design with Crosscutting Interfaces, IEEE Software, Vol. 23, No. 1, Jan./Feb. 2006, pp. 51-60.
- [15] J. Hannemann and G. Kiczales, Design Pattern Implementation in Java and AspectJ, Proc. of OOPSLA '02, 2002, pp. 161-173.
- [16] J. Hannemann, et al., Role-Based Refactoring of Crosscutting Concerns, Proc. of AOSD '05, 2005, pp. 135-146.
- [17] I. Hsi and C. Potts, Studying the Evolution and Enhancement of Software Features, Proc. of ICSM '00, pp. 143-151.
- [18] J. Kerievsky, Refactoring to Patterns, Addison-Wesley, 2004.
- [19] G. Kiczales, et al., Aspect-Oriented Programming, Proc. of ECOOP '97, 1997, pp. 220-242.
- [20] G. Kiczales, et al., An Overview of AspectJ, Proc. of ECOOP '01, 2001, pp. 327-353.
- [21] G. Kiczales and M. Mezini, Separation of Concerns with Procedures, Annotations, Advice and Pointcuts, Proc. of ECOOP '05, 2005, pp. 195-213.
- [22] S. Kim, et al., Micro Pattern Evolution, Proc. of MSR '06, 2006, pp. 40-46.
- [23] M. M. Lehman and J. F. Ramil, Metrics and Laws of Software Evolution - The Nineties View, Proc. of Metrics '97, 1997, pp. 20-32.
- [24] B. Lientz and E. Swanson, Software Maintenance Management, Addison-Wesley, 1980.
- [25] M. Lippert and S. Rook, Refactoring in Large Software Projects, John Wiley & Sons, 2006.
- [26] C. V. Lopes and S. Bajracharya, An Analysis of Modularity in Aspect-Oriented Design, Trans. on AOSD I, LNCS 3880. Springer, 2006, pp. 1-35.
- [27] R. Lopez-Herrejon, et al., Evaluating Support for Features in Advanced Modularization Technologies, Proc. of ECOOP '05, 2005, pp. 169-194.
- [28] T. Mens and A. H. Eden, On the Evolution Complexity of Design Patterns, Electronic Notes in Theoretical Computer Science, Vol. 127, No. 3, Elsevier, 2004, pp. 147-163.
- [29] T. Mens, et al., Challenges in Software Evolution, Proc. of IWPSE '05, 2005, pp. 13-22.
- [30] T. Mens and T. Tourwé, A Survey of Software Refactoring, IEEE Trans. on Software Eng. Vol. 30, No. 2, Feb. 2004, pp. 126-139.
- [31] M. P. Monteiro, Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts, PhD Thesis, Univ. do Minho, 2005.
- [32] K. Ostermann, et al., Expressive Pointcuts for Increased Modularity, Proc. of ECOOP '05, 2005, pp. 214-240.
- [33] D. L. Parnas, Software Aging, Proc. of ICSE '94, 1994, pp. 279-287.
- [34] V. Rajlich, Modeling Software Evolution by Evolving Interoperation Graphs, Annals of Software Eng., Vol. 9, 2000, pp. 235-248.
- [35] K. Sullivan, et al., The Structure and Value of Modularity in Software Design, Proc. of 9<sup>th</sup> ESEC/FSE, 2001, pp. 99-108.
- [36] K. Sullivan, et al., Information Hiding Interfaces for Aspect-Oriented Design, Proc. of 13<sup>th</sup> ESEC/FSE, 2005, pp. 166-175.
- [37] E. Swanson, The Dimensions of Maintenance, Proc. of ICSE '76, 1976, pp. 492-497.
- [38] L. A. Tokuda, Evolving Object-Oriented Designs with Refactorings, Ph.D. Thesis, Univ. of Texas at Austin, 1999.
- [39] J. van Gurp and J. Bosch, Design Erosion: Problems and Causes, J. of Systems & Software, Vol. 61, No. 2, Mar. 2002, pp. 105-119.
- [40] J. van Gurp, et al., Design Preservation over Subsequent Releases of a Software Product - A Case Study of Baan ERP, J. of Software Maintenance and Evolution, Vol. 17, No. 4, Jul. 2005, pp. 277-306.