ライブラリ移行を目的とした 機能の対応関係パターンとそれに基づく移行手法

Library Transition Techniques based on Function Relation Patterns

藤崎 洋子*　大久保 弘崇†　粕谷 英人‡　山本 晋一郎§

あらまし　ライブラリを利用しているアプリケーションを，同種の機能を提供する別のライブラリを利用するように修正する，ライブラリ移行について考える．本論文は，オブジェクト指向言語とそのクラスライブラリを対象として，ライブラリ移行を行うソースプログラムの書き換え手法を分析する．書き換え手法の選択には，ライブラリ間の機能の対応関係が重要であることを示す．6 種類の対応関係を発見し，4 種類の移行手法から各対応関係に適用する手法の選択指針を定めた．実験の結果，本手法により約 43%のクラス，57%のメソッドが移行できることを確認した．

1　はじめに

多種多様なライブラリが開発されているが，その用途や開発者の違いにより，同種の機能を提供するライブラリが複数存在する．本論文では，2 つのライブラリが同種の機能を提供するとき，それらを「類似ライブラリ」とよぶ．

あるアプリケーションが利用しているライブラリを別の類似ライブラリに移行する場合がある．それは，次のような状況で考えられる．

1. 機能が豊富で性能のよいライブラリに替えるとき
2. 環境の変化によって，現在のライブラリが利用できなくなったとき

ライブラリの移行を行う場合，最小限のプログラム修正によりライブラリを移行することが望ましいが，移行元と移行先のライブラリとの相違点を正確に把握することは簡単なことではない．そのため，ライブラリを移行するための手法の確立と，アプリケーションの最小限の修正で簡単に移行するための支援が必要である．

本論文では，オブジェクト指向言語とそのクラスライブラリを対象とし，最小限のソースプログラムの書き換えによるライブラリ移行手法を，ケーススタディを通して分析する．各機能を実現するクラス数によって機能の対応関係を分類し，対応関係の種類ごとに適した移行手法を提案する．本論文では，Swing から SWT [1] への移行をケーススタディとして用いる．Swing と SWT はどちらも GUI の機能を提供するライブラリである．Swing は Java 標準のライブラリであり，多くのアプリケーションで使用され，SWT は Eclipse [2] の GUI 部分で使用されている．

2　ライブラリの移行

ライブラリはひとまとまりの機能を提供する．クラスライブラリならば，1 つ 1 つの機能は，1 つのクラスまたはクラス群によって実現される．あるプログラムを類似ライブラリに移行させるためには，それぞれの機能の利用箇所を，類似ライブラリの対応する機能を利用するように書き換えることになる．対応する機能が移行先ライブラリで提供されていない場合，その機能を使用しているプログラムを書き換えることはできない．対応する機能がある場合でも，その機能を実現するクラスの使用方法が移行元と移行先で異なっているとき，ソースプログラムの機械的な書き換えは困難となる．

*Yoko Fujisaki, 愛知県立大学大学院 情報科学研究科, fujisaki@yamamoto.ist.aichi-pu.ac.jp

†Hirotaka Ohkubo, 愛知県立大学 情報科学部, ohkubo@ist.aichi-pu.ac.jp

‡Hideto Kasuya, 愛知県立大学 情報科学部, kasuya@ist.aichi-pu.ac.jp

§Shinichiro Yamamoto, 愛知県立大学 情報科学部, yamamoto@ist.aichi-pu.ac.jp

2.1 Swing から SWT への移行

本論文では Swing から SWT への移行をケーススタディとして用いる．Swing は Java 標準でかつ高機能であるため，多くのアプリケーションで GUI を実現するために使われている．SWT は複数プラットフォームのネイティブ GUI ライブラリにアクセスできる共通のライブラリを提供することを目的としており，標準的な機能のみを提供している．そのため，SWT に含まれる機能は Swing よりも少ない．

すでに Swing を利用して作られたアプリケーションを Eclipse のプラグインとして作り直すとき，Swing から SWT への移行が必要となる．Swing から SWT への移行を行う既存ツールに SwingWT [4] がある．SwingWT は SWT の機能を Swing のインタフェースに合わせるラッパークラスのライブラリである．ラッパークラスによる類似ライブラリへの対応はプログラムの書き換えが不要でコストは最小で済むが，それは対応であって移行とは言えない．プログラムのその後の保守や拡張を考えるとき，移行元と移行先のライブラリの両方をサポートする場合は，ラッパークラスの利用は有用となる．一方，移行元ライブラリを今後サポートしない場合は，アプリケーションの機能が移行元ライブラリに依存したままの状態はリスクとなる．この場合は，ソースプログラムの書き換えによる移行がよく適合する．

本論文ではラッパークラスは最低限使用しない．ソースプログラムの書き換えが困難な場合や，移行後のプログラムの可読性が著しく低下する場合に限り，ラッパークラスを使用する．

3 予備実験

Swing を使用しているいくつかのプログラムに対して，Swing から SWT へ移行する予備実験を行った．対象とするプログラムは Swing の公式チュートリアル [3] にある 129 個から 10 個をランダムに選んだ．これらのプログラムを S1 – S10 とする．S5 は S1 – S10 と異なるプログラム S11 を利用していた．これら 11 個のプログラムに対して，Swing から SWT への移行を手作業で行った．各プログラムの行数，使用している Swing のクラス数，呼び出している Swing のメソッド数を表 1 に示す．

3.1 実験結果

表 1 のプログラムの書き換えを試みた．機能が両ライブラリでどのようなクラスに対応したかを整理する．Swing が提供する機能はコンポーネント，グラフィック，イベント，レイアウトの 4 種類に分類される．各種類ごとに，機能とそれを実現する Swing のクラス，使用するプログラム，および SWT のクラスを表 2 – 5 に示す．

3.2 分析

3.2.1 コンポーネント

表 2 のパネルやフレーム機能は，Swing でも SWT でも 1 つのクラスで実現している．これらの機能は，移行元と移行先で構造やクラスの使用方法が似ている．例えば，パネルやボタンなどの機能を実現する Swing のクラスは，コンポーネント機能を実現する *JComponent* クラスを継承している．同様に，SWT のでは *Control* クラスを継承している．図 1 はクラスの置き換えによる，単純な書き換えの移行例で

表 1 対象プログラムの規模

プログラム	行数	使用している Swing のクラス数	呼び出している Swing のメソッド数
(S1) ListDemo	150	22	47
(S2) PasswordDemo	98	15	24
(S3) RootLayeredDemo	129	20	41
(S4) SliderDemo2	141	14	43
(S5) SplitPaneDemo2	50	9	14
(S6) TableSelectionDemo	190	23	47
(S7) TextFieldDemo	139	19	50
(S8) TextInputDemo	275	27	48
(S9) BoxAlignmentDemo	138	14	28
(S10) FindDemo	71	11	19
(S11) SplitPaneDemo	118	10	26
全体	1499	69	233

表 2　コンポーネント

機能	Swing のクラス	プログラム	SWT のクラス
コンポーネント	*JComponent*	S1 – S11	*Control*
パネル	JPanel	S1 – S4, S6, S8 – S9	Composite
フレーム	JFrame	S1 – S11	Shell
ラベル	JLabel	S2 – S11	Label
プッシュボタン	JButton	S1 – S2, S8 – S10	Button
ラジオボタン	JRadioButton	S6	Button
グループ	JPanel TitledBorder BorderFactory	S1,S3 – S4, S8 – S10	Group
リストデータモデル	DefaultListModel	S1	なし

表 3　グラフィック

機能	Swing のクラス	プログラム	SWT のクラス
サイズ	Dimension	S3,S5 – S6, S8 – S9, S11	Point
位置	Point	S3	Point
イメージアイコン	ImageIcon	S3 – S4, S9, S11	Image
色	Color	S3, S7	Color

表 4　イベント

機能	Swing のクラス	プログラム	SWT のクラス
フォーカスリスナ	FocusListener	S8	FocusListener
フォーカスイベント	FocusEvent	S8	FocusEvent
アクションリスナ	ActionListener	S1 – S4, S6, S8	SelectionListener
アクションイベント	ActionEvent	S1 – S4, S6, S8	SelectionEvent

表 5　レイアウト

機能	Swing のクラス	プログラム	SWT のクラス
ボックスレイアウト	BoxLayout	S1, S6, S8 – S9	FillLayout
グリッドレイアウト	GridLayout	S2 – S3	GridLayout
スプリングレイアウト	SpringLayout	S8	FormLayout FormData FormAttachment

ある．これはパネルの子としてラベルを配置している．Swing の *JComponent* クラスでは `add` メソッドによって，そのコンポーネントに引数として渡されたコンポーネントを配置する (図 1(a) 行 6)．一方，SWT の *Control* クラスではコンストラクタの引数に親となるコンポーネントを指定することによって，親コンポーネントに自身を配置する (図 1(b) 行 1．dummy は本来 panel を指定する)．親コンポーネントを指定するタイミングの違いにより，クラス名の書き換えだけでは移行できない．この場合は，*Control* クラスの `setParent` メソッドを用いて，オブジェクト生成時にはダミー値を与え (図 1(a) 行 1 を図 1(b) 行 1 に書き換え)，`add` メソッドが呼ばれた行を，親コンポーネントを再指定するように書き換える (図 1(a) 行 6 を図 1(b) 行 6 に書き換える) ことで，1 行ごとの変換で移行できる．

```
1  JLabel label = new JLabel();
2  label.setText("Hello!");
3
4  JPanel panel = new JPanel();
5
6  panel.add(label);
```

(a) Swing

⇒

```
1  Label label = new Label(dummy, SWT.CENTER);
2  label.setText("Hello!");
3
4  Composite panel
5      = new Composite(dummy, SWT.EMBEDDED);
6  label.setParent(panel);
```

(b) SWT

図 1　コンポーネント 変換例

グループ機能は，Swing では 3 つのクラス，SWT では 1 つのクラスで実現している．Swing では JPanel, TitledBorder, BorderFactory が，SWT では Group がグループ機能を実現する．JPanel, TitledBorder, BorderFactory の 3 クラスで行っている責務を，Group が行うように書き換えることで，Swing から SWT へ移行した．この書き換えは，静的解析を用いて自動化することが可能な複雑さであった．

3.2.2 グラフィック

表 3 のサイズや位置を表す機能は，実現する Swing と SWT のクラスが一対一対応している．Swing では，これらの機能を実現するクラスのオブジェクトは，引数なしでも生成できる．SWT では引数にサイズや位置を指定して，オブジェクトを生成する．オブジェクトの生成方法以外，これらの機能に対して Swing と SWT に大きな違いはなかった．Swing のクラスでは，引数なしでオブジェクト生成した場合，サイズや位置は 0 が初期設定される．従って，これらの機能は，クラス名，メソッド名の差し替えで機械的に書き換えられる．

色を表す機能は Swing でも SWT でも Color クラスが実現している．Swing の Color クラスには，色を名前で扱うためのクラス定数があるが，SWT にはそれがない．このような場合，移行後も内部的に Swing の Color クラスを利用する書き換え方法も考えられる．図 2(b) に示すように，Swing の Color クラスにある orange を使用し，その RGB 値を使って SWT の Color オブジェクトを改めて生成する．

```
1 import java.awt.Color;
2 ...
3 Color orange = Color.orange;
4
5
6
```

(a) Swing

⇒

```
1 import org.eclipse.swt.graphics.Color;
2 ...
3 java.awt.Color org = java.awt.Color.orange;
4 Color orange
5     = new Color(dummy, org.getRed(),
6                 org.getGreen(), org,getBlue());
```

(b) SWT

図 2　グラフィック 変換例

3.2.3 イベント

イベントのカテゴリでは，イベントクラスやイベントリスナそのものについては，機能はよく対応していた．特にイベントリスナについては，単純なクラス名の置き換えで移行できた．イベントリスナをコンポーネントに追加する際，その対象となるコンポーネントが異なる場合がある．また，イベントが実際に発生するタイミングが異なる場合もある．この 2 点の問題を解決するために，移行のための書き換えはプログラムの意味を考えて行う必要があった．またこの際，Swing の一行が SWT では数行の操作になり冗長になる箇所があったため，これをまとめるラッパークラスを作成した．移行後のソースプログラムは移行前と同等の可読性をもつようになった．

3.2.4 レイアウト

ボックスレイアウト機能やグリッドレイアウト機能は，Swing と SWT のクラスが一対一対応している．これらの機能を実現するクラスは，Swing と SWT でオブジェクト生成方法に大きな差はなく，また，可読性に影響するほどのメソッドの違いもない．そのため，クラス名，メソッド名を書き換えるだけで移行できる．

スプリングレイアウト機能は，Swing では 1 つのクラスが，SWT では 3 つのクラスが実現している．この機能は一連の制約に基づいてコンポーネントを配置する．Swing では，SpringLayout を使って実現する．SWT では，配置位置を設定する FormData，原点との距離を設定する FormAttachment，レイアウト機能を実現する FormLayout の 3 つのクラスを組み合わせて実現する．SpringLayout は，コンストラクタに引数を与えることによって，配置する位置や距離を指定する．しかし，プログラム S8 で使われた SpringLayout は，引数なしでオブジェクト生成し，それ以降，このオブジェクトは使用されていなかったため，SpringLayout を FormLayout に書き換えて移行を行うのみだった．

また，SWT では，コンポーネントを配置するために，レイアウトの設定が必要である．SWT で提供されていないレイアウト機能を移行する場合，別のレイアウト機能を実現するクラスに書き換えなければいけないが，この場合，コンポーネントの配置位置が変わり，移行後のプログラムの見栄えも変わる．しかし，見栄えの感覚は個人差があるため，適度によい見栄えであれば，コンポーネントの配置位置が異なっていても問題はない．Swing や SWT のような GUI 機能を提供するライブラリ

のレイアウトは，適度に良い機能を提供するはずである．従って，レイアウト機能に関しては，移行先のライブラリに合わせることも考えられる．移行先のライブラリに合わせる作業は，ライブラリ移行作業の後で行うことなので，実験では一時的な処置として，SWT に対応していない Swing のクラスはすべて，SWT の FillLayout に変換した．

4　対応関係と移行手法

3.1 章の分析を一般化するため，ライブラリの各機能を実現するクラス数による対応関係の分類と，各対応関係に適した移行手法の選択について議論する．

4.1　機能の対応関係

ライブラリを移行する際に，はじめに両ライブラリの機能とその対応関係を定めることが必要である．各対応関係に適した手法でソースプログラムを書き換えることにより，ライブラリを移行する．本節では，ライブラリ移行において出現する，機能とクラスの対応関係のパターンを挙げる．図 3 では長方形が機能を，その中の円が機能を実現するクラスを表している．

1:1 パターン (1 to 1 パターン)　(図 3(a))
移行元の 1 つのクラスで実現される機能を，移行先の 1 つのクラスで実現できる対応関係．この対応関係には似通った N 個の機能を，移行元では N 個のクラスが，移行先では 1 つのクラスが実現する場合の対応関係も含まれる．例えば，Swing の JButton や JRadioButton は，プッシュボタンやラジオボタンなどのボタン機能を実現するクラスである．SWT では Button クラスによってこれらのボタンを実現する．Swing から SWT への移行を考えた場合，Swing の 1 つのクラスが SWT の 1 つのクラスに対応しているので，1:1 パターンと考える．

多クラス結合パターン　(図 3(b))
移行元の複数のクラスの組み合わせで実現される機能を，移行先の 1 つのクラスで実現できる対応関係．3.2.1 節のグループ機能がこのパターンの例である．

機能分散パターン　(図 3(c))
移行元の 1 つのクラスで実現される機能を，移行先の複数のクラスのいずれか 1 つのクラスで実現できる対応関係．この対応関係では移行元の 1 つのクラスが複数の機能を実現しており，利用する機能の種類はこのクラスのメソッドなどで指定する．移行先ライブラリでは，1 つのクラスがその機能の 1 つの種類を実現し，他のクラスは別の種類を実現している．

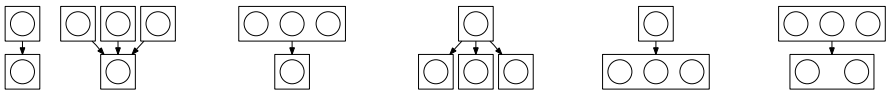
多クラス分散パターン　(図 3(d))
移行元の 1 つのクラスで実現される機能を，移行先の複数のクラスの組み合わせで実現できる対応関係．

N:M パターン (N to M パターン)　(図 3(e))
移行元の N 個のクラスの組み合わせによって実現される機能を，移行先の M 個のクラスの組み合わせで実現できる対応関係．

対応なし
移行元のクラスが実現している機能を，移行先ライブラリが提供しない場合．

4.2　移行手法

本論文では機能の対応関係に基づき，実現しているクラスの差し替えによって，ソースプログラムを書き換えてライブラリを移行させる．その複雑さにより書き換



(a) 1:1　(b) 多クラス結合　(c) 機能分散　(d) 多クラス分散　(e) N:M

図 3　機能の対応関係

え手法を分類する．書き換え手法はより単純な手法が望ましいが，対応関係のパターンにより適用可能な手法が限定される．本節ではソースプログラムの書き換え手法を分類し，適用できるパターンを明らかにする．

4.2.1 単純な置換

プログラムが使用しているクラス名やメソッド名を，対応するクラスやメソッドの名前に書き換えることによって変換を行う．1:1 パターンでは，メソッドが移行元と移行先で一対一対応する場合には，クラス名やメソッド名を書き換えるだけの単純な置換が適用できる．移行先のメソッドの引数として，生成されていないオブジェクトを必要とし，かつ，そのオブジェクトを後で再指定できる場合は，図 1 のようにダミー変数を使って変換する．単純な置換は，1:1 パターンや多クラス分散パターンで適用できる．また，クラス名やメソッド名を書き換えるだけなので，簡単に自動化が行える．

4.2.2 静的解析を用いた置換

単純な置換の延長として，プログラムを静的解析した情報を活用して書き換えができる場合がある．例えば，3.2.1 節のコンポーネントの例のように，移行先のメソッドの引数にその時点で生成されていないオブジェクトが必要な場合には，引数なるオブジェクトを先に生成できるようにデータフロー解析を行う．そして，引数となるオブジェクトを生成する行を入れ替える．これによって，図 1 のようなダミー変数を用いずに変換できる可能性がある．

オブジェクト生成時にはどの機能を実現するか分からない機能分散パターンでは，静的解析により実現する機能が決定できれば，単純な置換と同様に書き換えられる．

多クラス結合パターンでは，移行元の各クラスを，それに対応するクラスに変換することによって単純な置換で移行できる可能性がある．しかし，グループ機能のように，移行元の 1 つのクラスに対応するクラスが移行先にない場合は，単純な置換では移行できない．この場合には，静的解析を用いた置換が有用である．静的解析によって，移行元のクラスが実現している機能を決定することができる．

4.2.3 ラッパーを用いた置換

移行元ライブラリと同じインタフェースをもち，内部では移行先ライブラリを呼び出すラッパークラスを定義することで移行元ライブラリの機能がそのまま利用できる．イベントクラスのように，移行元の 1 つのメソッドが移行先の複数のメソッドに対応するために，移行後のプログラムが冗長で可読性が低くなる場合，ラッパークラスを用いることで可読性の低下を防ぐことができる．ラッパークラスができれば，ソースプログラムの変換は，クラスの利用箇所をラッパークラスに書き換えるだけでよい．この移行手法は，機能分散パターンや N:M パターンで有効である．また対応なしの機能にも有効である．2.1 節で述べたように，移行元ライブラリをサポートしない環境では，ラッパーの使用は望ましくない．ラッパーを用いた置換は，単純な置換や静的解析を用いた置換の適用が困難な場合にのみ適用すべきである．

4.2.4 移行元ライブラリを用いた置換

3.2.2 節の Color のような場面で，移行後も移行元ライブラリのクラスを利用する書き換えがある．移行元のクラスが実現する機能を，移行先では別のクラスのメソッドを呼び出すことによって実現する場合，このような書き換えを行う．移行元のクラスの情報を保持し，移行先のクラスのメソッドに保持した情報を引数として与えるように書き換えることで移行できる．移行元ライブラリを用いた置換は，移行先のメソッドの引数として必要な値を，移行元のクラスのメソッドで取得でき，取得した値が移行先のメソッドの引数の型と一致していなければ適用できない．また，この手法には，移行後のプログラムは移行元のライブラリがサポートされている環境でしか動作しないという制限もある．

5 評価

3 章で用いたプログラムの書き換えにおいて，本手法がどのくらい適用できたかを述べる．移行後のプログラムで，コンパイルエラーの原因とならなかったメソッ

ドを「変換できたメソッド」とし，以下の式によって，変換成功率を計算する．

$$\text{変換成功率 [\%]} = \frac{\text{変換できたメソッド数}}{\text{プログラムで呼び出しているメソッド数}} \times 100 \quad (1)$$

対応関係ごとの使用メソッド数，変換できたメソッド数，変換成功率および適用した移行手法を表 6 に示す．表 6 の 「単」,「静」,「ラ」,「移」はそれぞれ「単純な置換」,「静的解析を用いた置換」,「ラッパーを用いた置換」,「移行元ライブラリを用いた置換」を表す．

表 6　変換結果

対応関係	使用メソッド数	変換できたメソッド数	変換成功率 (%)	移行手法
1:1 パターン	178	130	73.0	単, ラ, 移
多クラス結合パターン	5	1	20.0	静
機能分散パターン	0	0	–	–
多クラス分散パターン	1	0	0.0	–
N:M パターン	0	0	–	–
対応なし	49	1	2.1	移
計	233	132	56.7	–

表 6 より 57%のメソッドが変換できた．すべてのメソッドを変換できたクラスは 69 個中 30 個で，43%のクラスが移行できた．特に 1:1 パターンに分類されたクラスでは，7 割のメソッドが変換できた．この対応関係は単純な置換が適用しやすく，移行を行うのが簡単であった．一方で，多クラス結合パターンや対応なしに分類されたクラスはあまり変換できず，コンパイルエラーの原因となるメソッドが多かった．

コンパイルエラーの原因となるメソッドでも，それによってプログラムの振る舞いを変えるものから，全く振る舞いを変えないものまである．そこで，コンパイルエラーが出ている部分をコメントアウトし，コンパイルエラーを回避したプログラムがどのくらい変換前のプログラムと同様に振る舞うかを調べた．ここでは，“レイアウト” と “振る舞い” に注目し，変換後のプログラムを 3 段階で評価した．

レイアウト

- ◯ ： 変換前と同じ部品が，同じ位置に配置されている．
- △ ： 変換前と部品の位置が異なる．または，一部の部品が表示されていない．
- × ： 多くの部品が表示されていない．

振る舞い

- ◯ ： 全ての部品が，変換前と同じ振る舞いをする．
- △ ： 一部の部品で，変換前と異なる振る舞いをする．またはエラーがでる．
- × ： 多くの部品で，変換前と異なった振る舞いをする．
- – ： レイアウトがうまく表示されていないため，振る舞いを確かめることができない．

移行後のプログラム実行時の評価結果を表 7 示す．また移行前，移行後のプログラムの行数も示す． S5 と S11 はレイアウトも振る舞いも移行前のプログラムと変わらなかった．これらのプログラムでコンパイルエラーとなっているメソッドは，プログラムの振る舞いを変えるほどの影響を与えないため，なくても問題はない．従って，S5 と S11 は移行に成功したとみなせる．S9 ではレイアウトが移行前と異

表 7　移行後のプログラム実行時の振る舞いとレイアウトの評価

プログラム	レイアウト	振る舞い	移行前の行数	移行後の行数
S1	◯	△	150	193
S2	◯	△	98	126
S3	△	×	129	169
S4	△	△	141	180
S5	◯	◯	50	70
S6	△	×	190	264
S7	×	–	139	165
S8	×	×	275	304
S9	△	◯	138	163
S10	×	–	71	80
S11	◯	◯	118	143
計	–	–	1499	1857

なっていた．しかし，レイアウト機能は，3.2.4 節で述べたように，移行先ライブラリのクラスを使うことによって，振舞いに影響しない程度の表示ができれば問題ない．よって S9 のように，振舞いが変わらなければ，移行に成功したとみなせる．S5, S9, S11 では変換成功率が示す数字よりも，移行に成功したと判断できる．

S7, S10 はフレーム以外の部品が表示されず，各部品の振舞いが確認できなかった．これらのプログラムは，GroupLayout を使って全ての部品を配置している．GroupLayout はコンポーネントをグループごとに配置するクラスであるが，SWT にはこのクラスに対応するクラスは存在しない．そのため，GroupLayout を変換できなかった．そこで，SWT の別のレイアウトクラスを使って各部品を表示し，プログラムの振舞いを確認した．S10 では各部品が移行前と同じ振舞いをし，また，レイアウトも移行前に近いレイアウトになった．従って S10 は移行に成功したとみなせる．S7 は一部の部品が移行前と異なる振舞いをした．S7 は検索文字をハイライト表示するプログラムだが，ハイライト表示機能を実現するクラスが SWT にないため，変換できなかった．ハイライト表示機能以外は移行前と同じ振舞いをした．

S1, S2, S4 は一部の部品が移行前と異なる振舞いをした．移行前と異なる振舞いをする部品の多くは，S10 のように移行後のプログラムを少し修正することによって，移行前と同様の振舞いをしたそのため，これらのプログラムは，本手法を適用後，少ないコストで移行が行えると判断できる．S3, S6, S7, S8 では多くの部品が移行前と異なる振舞いをした．これらの部品は同じような振舞いをするクラスがないため，移行後のプログラムを修正できない．また，どの移行手法も適用できなかった．

以上より，本手法によって 57%以上のメソッドが移行可能であり，これらを自動化できれば移行の際のコストが削減される．また，変換できたメソッドの多くは 1:1 パターンであるため自動化がしやすい．これにより，プログラムの主要な機能以外の移植作業におけるコストが削減され，開発者はより本質的な移行作業に専念できる．

6 おわりに

本論文ではライブラリ移行支援を目的とし，機能の対応関係に基づいたソースプログラム書き換えによる移行手法を提案した．ソースプログラム書き換え手法は複数あり，機能の対応関係から手法を選択する基準を与えた．具体的に 6 種類の対応関係を発見し，4 種類の移行手法から各対応関係に適した手法を選択した．いくつかのプログラムに対して本手法を適用した結果，約 57%のメソッドが変換でき，43%のクラスの移行に成功した．また，変換できなかったメソッドでも，移行後のプログラムを少し修正することでより多くのメソッドが変換できた．従って 43%以上のクラス，57%以上のメソッドが移行でき，ライブラリ移行のコストが削減できた．

今後の課題は，ライブラリの移行をできる限り自動化することである．最も簡単な手法である単純な置換による移行でも人手で行うにはコストがかかるため，自動化することによってコストを軽減することを目指す．また，現在はクラスの対応関係の分類のみだが，メソッドの対応関係も分類することによって，より確実な移行ができ，コスト削減につながる．また，表 6 に示した各パターン出現頻度は，対象が GUI であることに依存している可能性があるため，別の分野のライブラリを対象とした評価実験が今後の課題として必要である．

参考文献

[1] eclipse.org. “SWT: The Standard Widget Toolkit”, http://www.eclipse.org/swt/
[2] eclipse.org. “Eclipse”, http://www.eclipse.org/
[3] Sun MicroSystems. “The Java Tutorials”, http://java.sun.com/docs/books/tutorial/uiswing/
[4] R. Rawson-Tetley. “SwingWT”, http://swingwt.sourceforge.net/
[5] Sun MicroSystems. “Solaris 共通デスクトップ環境 Motif への移行”, http://docs.sun.com/app/docs/doc/805-5830/
[6] 深谷 和弘, 白銀 純子, 深澤 良彰. “GUI ウィジェット変更に伴うソースコード変更支援”, 電子情報通信学会ソフトウェアサイエンス研究会, Vol.104, No.571, pp.11–16.