日本ソフトウェア科学会第 29 回大会 (2012 年度) 講演論文集

破壊的クラス拡張のスコープを制限するモジュール機構の意味論およびその実装方法

竹下 若菜　赤井 駿平　千葉 滋

本研究では，破壊的クラス拡張のスコープを制限するモジュール機構 method shelters の意味論を簡素化したモジュール機構を提案する．既存のコードを書き換えずに既存のクラスに対して，メソッドの追加や再定義を行うことを破壊的クラス拡張と呼ぶ．破壊的クラス拡張を行うと，変更の衝突が生じる可能性がある．本研究のモジュール機構では，破壊的クラス拡張のスコープの制限をプログラマが行えるようにすることで，この問題を避ける事を可能にする．また，本研究ではこのモジュール機構に意味論を定義し効率の良い実装方法を提案する．本研究のモジュール機構を意味論のままに実装してしまうと，その言語の通常のメソッド探索に加えモジュール機構の動的なメソッド探索も行うため，通常のメソッド探索よりも遅くなってしまう．これを解消するために，通常のメソッド探索のコストに近づけるような実装方法を提案している．

1　はじめに

既存のプログラムを直接書き換えずに，既存のクラスにメソッドを追加したり，既存のメソッドを再定義したりする機能がプログラミング言語にある．この機能は，Ruby [1] の Open class や MultiJava [6] の Open class，GluonJ [5] などいくつかのプログラミング言語にすでに実装されている．この機能のことを本研究では破壊的クラス拡張と呼ぶ．メソッドの再定義とは，既存のメソッド定義を新しい定義で置き換えることを指す．破壊的クラス拡張を用いると，サブクラスを新たに作ったり他のプログラムを書き換えたりせずに，差分のみの記述でクラスに変更を加えることができる．

このような機能は便利である反面，多重再定義の危険を伴う．第三者が作った破壊的クラス拡張を再利用して使いたいことはしばしばある．しかし，第三者が作った破壊的クラス拡張を複数使用すると，同名のメソッド定義が同じスコープ内に複数存在し，多重再定義を引き起こすことがある．多重再定義を避ける方法として，各破壊的クラス拡張のスコープを制限する方法がある．スコープを制限する方法は Classboxes [4][3] や Ruby [1] の Refinements などすでにいくつか存在するが，それぞれに利点があるため一種類のスコープ規則で全ての場合に対応することはできない．

本研究では，exposedly なインポートと hiddenly なインポートという 2 種類のインポートを使い分けることで，様々なスコープの制限を可能にするモジュール機構 method shelters を提案する．2 種類のインポートによって，既存の複数の手法と同等のスコープ規則が実装できる．また，本研究では method shelters のメソッド探索についての形式的な定義を与える．この定義では method shelters のメソッド探索は実行時の文脈に依存したメソッド探索である．これを素朴にプログラミング言語に実装すると，その言語のメソッド探索に加え，method shelters のメソッド探索を行うため，method shelters を導入する前より実行時間のコストがかかる．そこで本研究では，導入する前のメソッド探索のコストと同じコストでメソッド探索できるよ

The semantics and implementation of a module system for scoping destructive class extensions

Wakana Takeshita, Shigeru Chiba, 東京大学情報理工学系研究科創造情報学専攻, Dept. of Creative Informatics, The University of Tokyo.

Shumpei Akai, 東京工業大学大学院情報理工学研究科数理・計算科学専攻, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology.

うな method shelters の実装方法も提案している．

2 破壊的クラス拡張のスコープ

破壊的クラス拡張

プログラムを直接書き換えずに，既存のクラスに変更を加えたい場合がしばしば存在する．例えば，既存のクラスライブラリを自分の使いたいようにカスタマイズしたいときである．既存のクラスに対して，メソッドを追加したい，一部のメソッドの定義を変更したいと思った時に破壊的クラス拡張は有用である．そのクラスに多数のサブクラスが存在している場合，またそのクラスを他のプログラムが使用している場合，それらを直接書き換えずにクラスのメソッドを変更するのは難しい．破壊的クラス拡張を用いれば，差分のみの記述でサブクラスを新しく作ったり，他のプログラムを書き換えたりせずに全体を変更できる．

例として，Windows XP の GUI デザインを実装している Window クラスを Windows 7 の GUI のデザインを実装するように変更するときに破壊的クラス拡張を考える (図 1)．図中の “`revise C{ML0 ML1 ....}`” の形のコードがクラス `C` に対する破壊的クラス拡張である．`ML` はメソッド宣言を表す．メソッド宣言は Java の文法と同じように記述する．各 `ML` のメソッド名がすでにクラス `C` 中に存在するならば再定義され，存在しないならばメソッドが追加される．“`revise C{ML0 ML1 ...}`” のような破壊的クラス拡張の宣言をこれ以降 reviser と呼ぶ．図 1 の例では，Windows XP のデザインのフレームの枠を実装している Window クラスの setBorder メソッドを，Windows 7 のデザインのフレームの枠を実装するように Window クラスの外から再定義している．このように再定義することで，setBorder メソッドの定義は新しい定義に置き換えられ，各プログラムの Window クラスの setBorder メソッドの呼び出しでは Windows 7 のフレームの枠を実装している setBorder メソッドが呼び出されるようになる．

このようなメソッドの再定義は Window クラスのサブクラス Win7Window クラスを作っても可能である．しかし，その場合，Window クラスを継承している各サブクラス (図中では Dialog クラスや Frame クラスなど) それぞれについて Win7Window クラスを継承するサブクラスを新たに作る必要がある．また，プログラム中で Window クラスのオブジェクトを生成している全ての部分を Win7Window クラスのオブジェクトを生成するように書き換える必要がある．破壊的クラス拡張を用いれば，reviser の記述のみが必要でサブクラスを新たに作ったり各メソッド呼び出しを書き換えたりする必要はない．そのためコードの再利用性を高めることができる．

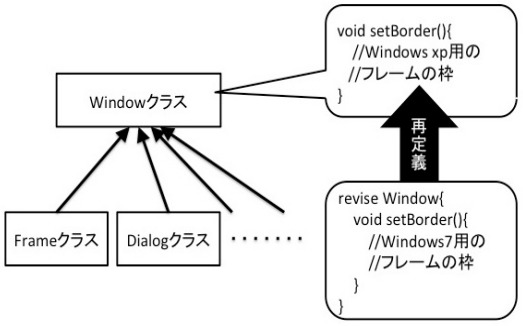

図 1 破壊的クラス拡張の例

多重再定義の危険性

このように一見便利に見える破壊的クラス拡張だが，多重再定義の問題がある．第三者が作った複数の破壊的クラス拡張の定義を再利用して使いたいことはしばしばある．このためクラスライブラリをインポートするように，破壊的クラス拡張もインポートできる．第三者が作った破壊的クラス拡張をプログラム中でインポートして使用すれば，コードの再利用性はさらに増し，プログラマの負担も軽くなる．また，インポートした破壊的クラス拡張のうち一部のメソッド定義を再度再定義することで，プログラマにとってさらに有用なメソッドへと書き換えることができる．

しかし，破壊的クラス拡張を複数インポートすると，同じスコープ内に同じクラスの同じメソッド名が複数存在する多重再定義になる危険がある．例としてある GUI を持ったアプリケーションを作ることを考える．このアプリケーションでは，基本的な GUI のデザインとして Windows 7 の GUI デザインを使う．しかし，フレームの枠のデザインだけはこのアプリケーション専用のデザインに変える．さらに，インス

トール時の管理者権限認証時のダイアログパネルには，ウイルスに偽パネルを出されにくいように通常と異なるデザインのものを使う．このアプリケーションのコード例は図 2 のようになる．図 2 の import 宣言は，パッケージのインポートを宣言している．インポートしたパッケージ内にある reviser はインポートを宣言しているパッケージ内で定義しているかのように扱われる．ここでは，Windows 7 用の GUI と管理者権限認証用の GUI の両方を使うために，第三者が作った win7 パッケージと authentication パッケージを app パッケージでに一緒にインポートする．そして，フレームの枠だけはこのアプリケーション用のデザインにするために，app パッケージの中で Window クラスの setBorder メソッドを reviser で再定義する．

このとき，win7 パッケージには Window クラスを Windows 7 の GUI デザインにするための reviser が存在し，authentication パッケージには Window クラスを管理者権限認証用の GUI デザインにするための reviser が存在している．app パッケージ内の setBorder メソッドが再定義したいのは，win7 パッケージの setBorder メソッドであるが，素直に考えると authentication パッケージ内の setBorder メソッドも再定義される．そのため，28 行目の install メソッド内で checkPass メソッドを呼び出し，さらに checkPass メソッド中で setBorder メソッドを呼び出すと，期待する authentication パッケージの setBorder ではなく app パッケージ中の setBorder メソッドの定義が呼び出される．これでは，通常と異なるデザインを使うという目的が果たせない．app パッケージ内の setBorder メソッドの定義を削除したとしても，インポートに優先順序がないと，setBorder メソッドの定義が app パッケージ内に 2 つ存在してしまいメソッド定義が衝突する．インポートに優先順序があれば，どちらかの setBorder メソッドがもう片方の setBorder メソッドを上書きして再定義するが，通常用と管理者権限認証用とで異なるデザインを使うという目的を果たせない．

```
1  package win7;
2  revise Window{
3    void setBorder(){//windows7のフレームの枠}
4    void setBackgroundColor(){...}
5    void makeFrame(){
6        :
7      setBorder();
8        :
9    }
10 }
11
12 package authentication;
13 revise Window{
14   void setBorder(){//管理者権限認証用のフレームの枠}
15   void setBackgroundColor(){...}
16   void checkPass(){
17       :
18     setBorder();
19       :
20   }
21 }
22
23 package app;
24 import win7;
25 import authentication;
26 revise Window{
27   void setBorder(){//このアプリケーション用のフレームの枠}
28   void install(){
29       :
30     checkPass();
31       :
32     makeFrame();
33       :
34   }
35 }
```

図 2　メソッドの衝突例

多重再定義を避けるスコープ規則

多重再定義による問題を避ける方法の一つに，各 reviser のスコープを制限する方法がある．スコープ規則は，Classboxes [4][3] や Ruby [1] の Refinements などすでにいくつか提案されている．Classboxes は破壊的クラス拡張と import 宣言からなる classbox というモジュールを導入している．import 宣言を用いると他の classbox で定義されているクラスを classbox 内にインポートできる．破壊的クラス拡張のスコープは (i) 定義している classbox 内，(ii) それをインポートしている classbox 内である．Refinements は，破壊的クラス拡張をスコープ内で宣言することで，そのスコープ内でのみ破壊的クラス拡張を有効にする．このようにいくつかスコープ規則が存在するが，それぞれに異なる利点があるため，一種類のスコープ規則で全ての場合に対応することはできない．

3 モジュール機構 Method Shelters

Exposedly なインポートと hiddenly なインポートという 2 種類のインポートを使い分けることで，用途に応じて様々なスコープの制限ができるモジュール機構 method shelters を提案する．method shelters を用いると，既存の複数の手法と同等のスコープ規則を実装できる．method shelter は reviser と import 宣言からなるモジュールである．method shelter のことをこれ以降シェルターとも呼ぶ．Reviser を用いると，破壊的クラス拡張を定義できる．シェルターの中で reviser を実装することで，reviser のスコープをシェルターの中に制限できる．import 宣言を用いると他のシェルターをシェルター内にインポートできる．インポートしたシェルター中の reviser はインポートしているシェルターで定義したものであるかのように扱われる．インポートには hiddenly なインポートと exposedly なインポートの 2 種類があり，この 2 つを使い分けることで様々なスコープの制限を可能にする．また，環状インポートはエラーとする．Exposedly なインポートは定義 3.1 のような推移性を持つ．S,T,U はシェルターを表すメタ変数とする．

定義 3.1. S が T を exposedly にインポートしていることを $S \xrightarrow{e} T$，S が T を hiddenly にインポートしていることを $S \xrightarrow{h} T$ と書くとすると，次のような関係が成り立つ．

- $S \xrightarrow{e} T,\ T \xrightarrow{e} U \Rightarrow S \xrightarrow{e} U$
- $S \xrightarrow{h} T,\ T \xrightarrow{e} U \Rightarrow S \xrightarrow{h} U$

3.1 Hiddenly なインポートと exposedly なインポート

Hiddenly なインポートと exposedly なインポートがそれぞれどのようなインポートであるか，2 章のアプリケーションの例を method shelters で実装することで示す (図 3)．図 3 のコード中には app,win7,authentication の 3 つのシェルターが存在する．

このアプリケーションは Windows 7 の GUI を基本的な GUI とするが，フレームの枠だけはこのアプリケーション専用のものに変える．そこで，Windows 7 の GUI を実装している win7 シェルターを exposedly にインポートする．app シェルターの import 宣言は 23 行目であり，exposedly 宣言することで，exposedly にインポートできる．さらに app シェルターで Window クラスの setBorder メソッドを reviser で再定義する．exposedly にインポートしたメソッド定義はインポートしたシェルターで再定義できる．そのため，win7 シェルターの setBorder メソッドの定義は，app シェルター内の定義で再定義される．app シェルター内で win7 シェルター内の makeFrame メソッドを呼び出し，makeFrame メソッド中で setBorder メソッドを呼び出すと，app シェルター内の setBorder メソッドの定義が呼び出される．

このアプリケーションではインストール時の管理者権限認証用のダイアログパネルだけは異なる GUI にする．そこで，管理者権限認証用の GUI が実装されている authentication シェルターを hiddenly にインポートする．hiddenly なインポートは “hiddenly” と明記しない．hiddenly にインポートしたシェルター内のメソッドを呼び出すと，そのメソッド中はそのシェルターの中でメソッド呼び出しが行われる．今回の例では，app シェルターで authentication シェルター内の checkPass メソッドを呼び出すと，checkPass メソッド中では authentication シェルター内でメソッド呼び出しが行われ，app シェルターの影響を受けない．そのため，checkPass メソッド中の setBorder メソッドの呼び出しでは authentication シェルター中の 14 行目の setBorder メソッドの定義が呼び出される．

3.2 メソッド探索のセマンティクス

3.1 節では exposedly なインポートと hiddenly なインポートの動きを例で示したが，この節ではさらに厳密な定義を与える．本モジュール機構のメソッド探索は，入力としてクラス名，メソッド名，カレントシェルター，ソースシェルターの 4 つを必要とする．通常の言語のメソッド探索とはカレントシェルターとソースシェルターを必要とする点で異なる．カレントシェルターはメソッド呼び出しが行われたシェルターを指す．カレントシェルターとソースシェルターの初期値は共にルートシェルターである．ルートシェルターとは，一番最初にメソッド呼び出しをしたシェルターの

```
1  shelter win7;
2  revise Window{
3    void setBorder(){//windows7用のフレームの枠}
4    void setBackgroundColor(){…}
5    void makeFrame(){
6           :
7      setBorder();
8           :
9    }
10 }
11
12 shelter authentication;
13 revise Window{
14   void setBorder(){//管理者権限認証用のフレームの枠 }
15   void setBackgroundColor(){…}
16   void checkPass(){
17    :
18     setBorder();
19          :
20   }
21 }
22
23 shelter app imports exposedly win7, authentication;
24 revise Window{
25   void setBorder(){//アプリケーション用の枠}
26   void install(){
27          :
28     checkPass();
29          :
30     makeFrame();
31          :
32   }
33 }
```

図 3　Method Shelters の例

ことを指す．

ソースシェルターは実行時の文脈の情報を表す．複数のシェルターからインポートされているシェルターがあるとき，どのシェルターを通ってきたかという情報が必要となる．図 4 がその例である．図中には app,win7,authentication,stdGUI の 4 つのシェルターが存在している．点線の四角が hiddenly にインポートされているシェルターを表し，実線の四角が exposedly にインポートされているシェルターを表す．この図では app シェルターが win7 シェルターを exposedly に authentication シェルターを hiddenly にインポートし，authentication シェルターと win7 シェルターが stdGUI を exposedly にインポートしている．stdGUI シェルター中では setBorder メソッドを呼び出すような update メソッドが実装されている．setBorder メソッドは win7 シェルターと authentication シェルターで実装されている．このとき，win7 シェルターから update メソッドが呼び出されたならば，update メソッド中の setBorder メソッドの呼び出しは win7 シェルター中の setBorder メソッドの定義を呼び出す．逆に authentication シェルターから update メソッドを呼び出したならば update メソッド中では authentication シェルター中の setBorder メソッドの定義を呼び出す．ソースシェルターとはこのような呼び出しの切り替えを実現するための実行時の文脈を表す．ソースシェルターはそのメソッド探索の前のメソッド探索から与えられる．

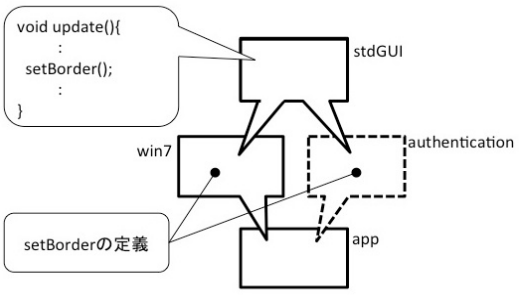

図 4　複数からインポートされているシェルター

本研究ではメソッド探索の意味論の形式化を行った．形式化を示す前にまず本モジュール機構の文法を示す．

$$
\begin{array}{rcl}
\mathtt{FILE} & ::= & \mathtt{SL};(\mathtt{CL} \parallel \mathtt{RL})* \\
\mathtt{SL} & ::= & \mathtt{shelter\ S(IL} \parallel \bullet\mathtt{)} \\
\mathtt{IL} & ::= & \mathtt{imports\ ((exposedly\ S} \parallel \mathtt{S)},)* \\
 & & \mathtt{(exposedly\ S} \parallel \mathtt{S)} \\
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C\{}\overline{\mathtt{C}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\mathtt{\}} \\
\mathtt{RL} & ::= & \mathtt{revise\ C\{}\overline{\mathtt{M}}\mathtt{\}} \\
\mathtt{M} & ::= & \mathtt{C\ m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{body\}}
\end{array}
$$

S, T はシェルターを表すメタ変数である．C はクラス名，f はフィールド名，K はコンストラクタ，m はメソッド名，x は変数名を表すメタ変数である．また，列を表すのにオーバーラインを用いる．例えば，$\overline{\mathtt{M}}$ は $\mathtt{M}_1, \mathtt{M}_2, ..., \mathtt{M}_n$ を表し，$\overline{\mathtt{C}}\ \overline{\mathtt{f}};$ は $\mathtt{C}_1\ \mathtt{f}_1;, \mathtt{C}_2\ \mathtt{f}_2;, ..., \mathtt{C}_n\ \mathtt{f}_n;$ を表す．

本研究のメソッド探索の形式化は図 5 のようになる．関数 *mbody* がメソッド探索を行う関数である．*mbody* は入力として，クラス名 C，メソッド名

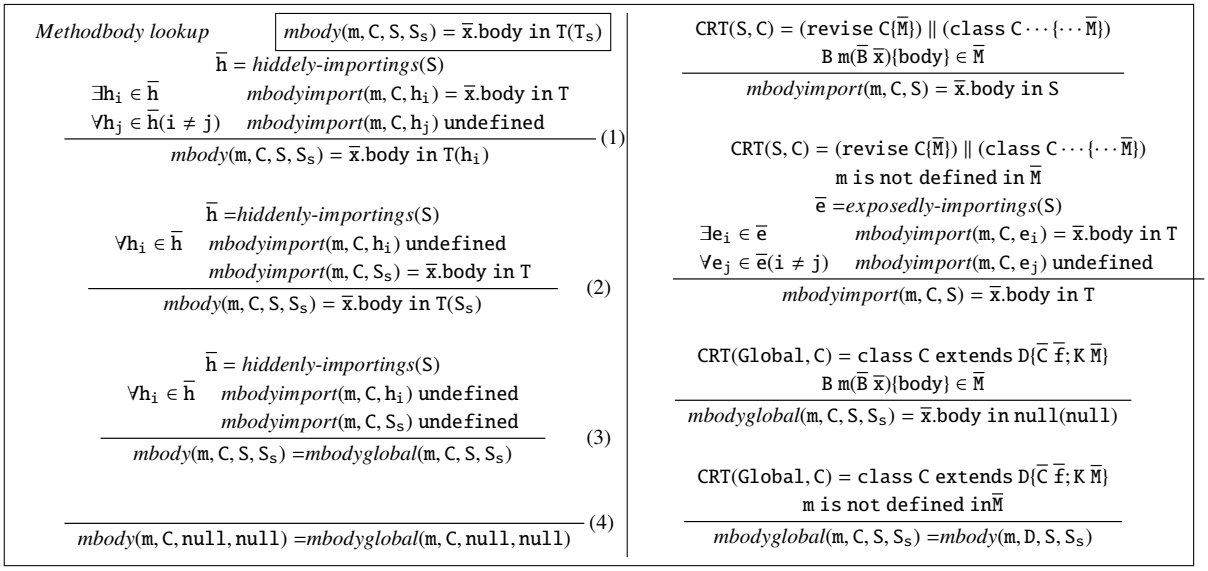

図 5　メソッド探索の形式化

m，カレントシェルター S，ソースシェルター S_s を必要とする．出力はメソッドの引数の集合 $\bar{x}$ とメソッドボディ body，メソッド定義が見つかったシェルター T と次のメソッド探索のソースシェルター T_s の 4 つであり，$\bar{x}$.body in $T(T_s)$ という形で返す．関数 *hiddenly-importings* はシェルター名 S が与えられると，S が hiddenly にインポートしている全てのシェルターを返す．関数 *exposedly-importings* はシェルター名 S が与えられると，S が exposedly にインポートしている全てのシェルターを返す．

メソッド探索の定義から，幾つかの性質が導ける．性質 3.1 は再定義に関する性質である．性質 3.2 は同名のメソッド定義が複数あった時に，メソッド呼び出しでどの定義が呼び出されるのかに関する性質である．

性質 3.1.
- Exposedly にインポートしているシェルターのメソッド定義はインポートしているシェルターで再定義可能
- Hiddenly にインポートしているシェルターのメソッド定義はインポートしているシェルターで再定義不可能

性質 3.2.
- Hiddenly にインポートしている定義が優先的に呼び出される
- その定義に再定義がある場合には，そちらが優先的に呼び出される

証明. Exposedly にインポートしているシェルターのメソッド定義をインポートしているシェルターで再定義できることは，仕様としてすでに述べた (*1)．図 5 の関数 *mbody* の定義である式 (1) から式 (4) までを見ると，次のようなメソッド探索をしている．hiddenly にインポートしているシェルターを関数 *mbodyimport* で探索し，メソッド定義がただ 1 つ存在するならばそれを返す．存在しないならば，ソースシェルターから関数 *mbodyimport* で探索し，メソッド定義がただ 1 つ存在するならばそれを返す．それでも見つからないならば，関数 *mbodyglobal* によりシェルターの外の部分を探索する．そのため，複数定義があるときには hiddenly にインポートしている定義が優先的に呼び出される (*2)．複数定義があっても，hiddenly にインポートしているシェルターのメソッド定義が呼び出されるため，hiddenly にインポートしている定義を再定義するのは不可能である (*3)．また，関数 *mbodyimport* は与えられたシェルターから exposedly にインポートしているシェルターを順にたどって探索をするため，再定義が先に呼び出される (*4)．(*1)(*3) より性質 3.1 が，(*2)(*4) より性質 3.2 が言える．

3.3 様々なスコープの制限

Classboxes のようなスコープの制限も Refinements のようなスコープの制限も method shelters で実装できる．

Classboxes のようなスコープの制限

Classboxes, Classbox/J は classbox というモジュールを導入している．classbox は定義と import 宣言からなるモジュールである．図 6 が Classbox/J のコード例である．Classbox/J は Java に classbox を導入している．package 宣言が classbox の宣言を表す．例中には intDivCB と rationalDivCB の 2 つの classbox が存在する．さらに rationalDivCB は intDivCB で実装されている MyInteger クラスと MyCalc クラスをインポートしている．MyInteger クラスの div メソッドは intDivCB 中では割算の結果を int の形で出力するものであったが，rationalDivCB では割り算の結果を分数の形で出力するように refine という宣言で再定義している．インポートしている定義はその classbox 内にあるかのようにふるまうため，rationalDivCB で MyCalc クラスの foo メソッドを呼び出せる．また，MyInteger クラスの div メソッドの再定義は rationalDivCB 内全てに適用される．そのため，19,20 行目の出力はどちらも分数となる．このような classbox によるスコープ制限は method shelters の exposedly なインポートで実装できる．コード例を図 7 で示す．exposedly にインポートしたシェルターの中のクラス定義はインポートしているシェルターで実装しているかのように扱うことができる．また，exposedly にインポートしたシェルターの中のクラス定義はインポートしているシェルターで再定義することができる．そのため，図 7 のコードで図 6 のコードと同等のスコープ規則を実現できる。

```
1  package intDivCB;
2  public class MyInteger{
3    void div(int a, int b){//b/aの結果をintの形で出力}
4  }
5  public class MyCalc{
6    void foo(){
7      new MyInteger().div(2,1);
8    }
9  }
10
11 package rationalDivCB;
12 import intDivCB.MyInteger;
13 import intDivCB.MyCalc;
14 refine MyInteger{
15   void div(int a, int b){//b/aの結果を分数の形で出力}
16 }
17 refine MyCalc{
18   void bar(){
19     new MyInteger().div(2,1); //'1/2'を出力
20     foo();                    //'1/2'を出力
21   }
22 }
```

図 6 Classbox/J のコード例

```
1  shelter intDiv;
2  public class MyInteger{
3    void div(int a, int b){//b/aの結果をintの形で出力}
4  }
5  public class MyCalc{
6    void foo(){
7      new MyInteger().div(2,1);
8    }
9  }
10
11 shelter rationalDiv imports exposedly intDiv;
12 revise MyInteger{
13   void div(int a, int b){//b/aの結果を分数の形で出力}
14 }
15 revise MyCalc{
16   void bar(){
17     new MyInteger().div(2,1); //'1/2'を出力
18     foo();                    //'1/2'を出力
19   }
20 }
```

図 7 図 6 を Method Shelters で記述

Refinements のようなスコープの制限

図 8 が Ruby の Refinements のコード例である．Refinements では，破壊的クラス拡張からなるモジュールを using 宣言したスコープ内でのみ破壊的クラス拡張が有効となる．MyInteger クラスのオリジナルの div メソッドは 2 つの引数の割り算を int の形で返すメソッドである．図 8 のコード例では 2 行目から 6 行目で破壊的クラス拡張が実装され，10 行目で rationalDiv モジュールが using 宣言されている．そのため，rationalDiv モジュールの破壊的クラス拡張のスコープは 10 行目から 14 行目までとなり，18 行目と 20 行目のような出力となる．このような Refinements によるスコープ制限は method shelters の hiddenly なインポートで実装できる．コード例を図 9 で示す．hiddenly なインポートされているシェルターの中のメソッド定義はインポートしているシェルター内から呼

```
module rationalDiv
  refine MyInteger do
    def div(a, b)
      # b/aの結果を分数の形で出力
    end
  end
end

class MyCalc
  using rationalDiv
  def foo()
    m = MyIneger.new
    m.div(2, 1)
  end
end

c = MyCalc.new
c.foo()          # '1/2'を出力
m = MyInteger.new
m.div(2, 1)      # '0'を出力
```

図 8　Refinements のコード例

```
shelter rationalDiv;
revise MyInteger{
  void div(int a, int b){//b/aの結果を分数の形で出力}
}

shelter calc imports rationalDiv;
public class MyCalc{
  void foo(){
    new MyInteger.div(2,1);
  }
}

shelter main imports exposedly calc;
void main(){
  new MyCalc.foo();    //'1/2'を出力
  new MyInteger.div(); //'0'を出力
}
```

図 9　図 8 を Method Shelters で記述

び出すことはできるが，再定義することはできない。そのため、図 9 のようなコードで図 8 のコードと同等のスコープ制限が実装できる。

4　実装方法

本システムをプログラミング言語に導入するときに，導入前のメソッド探索の実行時間のコストと同じコストでメソッド探索ができるような実装方法を提案する．3 章の定義を素朴に実装すると，導入した言語のメソッド探索に加えて，本モジュール機構の文脈依存のメソッド探索を行うため，導入する前よりもメソッド探索の実行時間のコストがかかる．そこで，コンパイル時に文脈に依存しない形へと変換する．

4.1　実装方法の提案

ソースシェルターをカレントシェルターから一意に決定できれば，メソッド探索時の文脈に依存する入力が減らせるため，導入する前のメソッド探索の実行時間のコストに近づく．ソースシェルターをカレントシェルターから一意に決定するために新たな構造の tree を作成する．tree は式 (5) のようにして作成する．関数 *shelters*(root) はシェルター root からたどることのできる全てのシェルターを返す．関数 *allPath*(root) はシェルター root からたどることのできるすべてのシェルターへのすべてのパスを返す．S はシェルター名を表す．p はルートシェルターを先頭とするシェルター名の列であり，パスを表す (例えば $p = S_0S_1S_2$ のようになる)．E は tree の辺の集合を表し，*shelters*(root) が tree のノード集合である．作成された tree ではノードさえ分かればソースシェルターを一意に決定でき，呼び出すメソッド定義も一意に決定できる．

$$\begin{cases} P = \mathit{allPath}(\mathtt{root}) \\ \forall p \in P, \exists S \in \mathit{shelters}(\mathtt{root})\ \mathit{s.t}\ pS \in P \end{cases} \Longrightarrow (p \to pS) \in E$$

$$\mathtt{tree} = (\mathit{shelters}(\mathtt{root}), E) \tag{5}$$

さらに，コンパイル時に一意に決定したメソッド定義のメソッド名を変える．メソッド名を変更すると，メソッド名とメソッド定義が一対一に対応する．そのメソッド定義を呼びだしているメソッド呼び出しも変更したメソッド名を呼び出すように書き換える．このような処理をコンパイル時に行うことで，クラス名とメソッド名から呼び出すメソッド定義が一意に決定できるようになり，通常のメソッド探索と同じ実行時間のコストで探索できる．

5　関連研究

5.1　前身研究：method shelters

本研究には前身研究[2] が存在する．前身研究時の method shelters では，シェルターの中を exposed chamber と hidden chamber という 2 つの chamber に分割していた．しかし，chamber が存在するために定義が複雑になっていた．そこで，本研究では chamber をな

くし，代わりに hiddenly なインポートと exposedly なインポートを導入した．また，前身研究では形式化がされていなかったが，本研究ではメソッド探索に対して形式化を与えている．

6 まとめ

破壊的クラス拡張は便利である反面，メソッドの多重再定義の危険がある．多重再定義はスコープを制限すると避ける事ができる．スコープの制限方法には様々なものがあるが，それぞれに利点があるため，一種類のスコープ規則で全ての場合に対応することはできない．

本研究では，exposedly なインポートと hiddenly なインポートを使い分けることで，様々なスコープの制限を可能にするモジュール機構 method shelters を提案した．2 種類のインポートを導入することで，多重再定義の危険を避けることが可能となった．また，Classboxes や Refinements といった他のシステムのようなスコープの制限方法も method shelters では実現できる．さらに，本研究では，このモジュール機構のメソッド探索の形式化も行った．

また，実装方法の提案も行った．形式化した method shelters のメソッド探索は，実行時の文脈に依存する．そのため素朴にプログラミング言語に導入すると，その言語のメソッド探索に加え，method shelters のメソッド探索も行うため余分に実行時間のコストがかかる．本研究では，コンパイル時に実行時の文脈に依存しない形に変換することで，method shelters を導入する前のメソッド探索の実行時間のコストと同じコストでメソッド探索ができるような実装方法を提案した．

参考文献

[1] Ruby programming language. http://www.ruby-lang.org/.

[2] Shumpei Akai and Shigeru Chiba. Method shelters: avoiding conflicts among class extensions caused by local rebinding. In Proceedings of the 11th annual international conference on Aspect-oriented Software Development, AOSD '12, pages 131–142, New York, NY, USA, 2012. ACM.

[3] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/j: controlling the scope of change in java. In Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05, pages 177–189, New York, NY, USA, 2005. ACM.

[4] Alexandre Bergel, Stphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. In Computer Languages, Systems and Structures, 2005.

[5] Shigeru Chiba, Atsushi Igarashi, and Salikh Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications, OOPSLA '10, pages 539–554, New York, NY, USA, 2010. ACM.

[6] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '00, pages 130–145, New York, NY, USA, 2000. ACM.