平成23年度 学士論文

モジュール機構
method sheltersの
意味論と効率の良い実装方法

東京工業大学 理学部 情報科学科
学籍番号 08-1479-0
竹下 若菜

指導教員
千葉 滋 教授

平成24年2月6日

概要

本研究では、Method Shelters を簡素化した Simplified Method Shelters を提案する。Method Shelters とは、メソッド定義のスコープをコントロールすることでクラス拡張によるメソッド名の衝突を避けることを可能にしたモジュールシステムである。

既存のクラスのクラス定義のコードに手を加えることなく、既存のクラスに新しいメソッドを加えたり、既存のメソッド定義を変更したりすることを破壊的クラス拡張と呼ぶ。既存のクラスに対してクラス定義のコードを修正することなく、自由に変更を行えることは、プログラマに多彩な表現方法を与え、プログラムの書き方の自由度の向上にもつながる。

このようなメリットがある反面、破壊的クラス拡張はいくつかの問題点も持ち合わせている。同じクラスの同名のメソッドを別々のライブラリが定義していたとする。これらのライブラリを同時に使用すると、メソッド名が衝突してしまう。また、既存のメソッドに変更を加えることによって、そのメソッドを使用していたほかの部分に影響を与えてしまう可能性もある。どちらの問題もプログラマの予期せぬ動作を引き起こすことにつながる。これらの問題は破壊的クラス拡張の影響がプログラム全体に及んでしまうために引き起こされることである。

この問題を克服する技術の 1 つとして、Method Shelters が存在する。Method Shelters はチェンバーというアイデアを導入することで、メソッド定義のスコープをコントロールすることを可能にした。スコープをコントロールすることによって、破壊的クラス拡張の影響がプログラム全体に及ぶのを防ぐことが可能となり、メソッド名の衝突を避けることができるようになっている。

しかし、Method Shelters には 2 つの欠点が存在する。第一の問題点として、チェンバーを導入することでセマンティクスの複雑さが増したことがあげられる。Method Shelters のセマンティクスでは、シェルター単位で考える部分とチェンバー単位で考える部分が混在してしまっている。そのため、Method Shelters のメリットを十分に活用するのが難しくなってしまっている。第二の問題点として、言語に導入する際、MethodShelters のメソッド探索のセマンティクスをそのまま導入してしまうと、Method Shelters を導入する前より実効速度が遅くなることが懸念されることがあ

げられる。Method Shelters は、独自のメソッド探索のセマンティクスを持っている。そのため、VM に手を加えずにそのまま導入してしまうと、通常のメソッドディスパッチに加え、Method Shelters のメソッドディスパッチも用いるため、導入前より実行時間が遅くなってしまう。

以上の問題点を解決するために、本研究では Simplified Method Shelters を提案する。Simplified Method Shelters の Method Shelters からの大きな変更点は、チェンバーを廃止したことと 2 種類のインポートを導入したことである。チェンバーをなくすことによって、セマンティクスの簡素化をはかり、より分かりやすいモジュールシステムを提供することを図った。また、2 種類のインポートを導入することにより、チェンバーの無い状態でもスコープのコントロールを行うことを可能にした。これらの変更により、Method Shelters とほぼ同等の機能を保ったまま、セマンティクスを簡素化することに成功した。さらに、セマンティクスからいくつかの性質を導き出すことによって、Simplified Method Shelters の基本的な性質を明確化し、さらなる分かりやすさにつとめた。

また、実行速度の遅さを克服するため、実装方法の提案も行った。Simplified Method Shelters のメソッド探索のセマンティクスでは、動的な情報を必要とするため、実行時探索を行なっている。この動的な情報を減らし、複数ある同名のメソッド定義の中から呼び出すメソッド定義をコンパイル時に 1 つに絞り、そのメソッド定義のメソッド名を変更することにした。こうすることでメソッド名とメソッド定義が 1 対 1 に対応するため、通常のメソッドディスパッチのコストに近づけることができ、速度の向上が図れると考えた。また、提案した実装方法が本当に Simplified Method Shelters のメソッド探索のセマンティクスを満たしているのかどうかといった、正当性の議論も行った。

謝辞

本研究を進めるにあたって、研究の方針や論文の構成など様々な助言および指導をして下さった指導教員の千葉滋教授に心から感謝致します。

また、研究の進め方や論文の書き方、本研究の仕様について赤井駿平氏に多くの助言を頂きました。深く感謝致します。

最後に数々の助言を頂き、様々な知識を与えて下さった千葉研究室の皆様方に心より感謝致します。

目 次

図 目 次

第1章　はじめに

既存のクラスのクラス宣言のコードに直接手を加えることなく、そのクラスを拡張するというのは、とても重要な技術である。クラスを拡張するとは、主に以下の 2 つのことを行うことを指す。

- 既存のクラスに新しいメソッドを追加する
- 既存のメソッドの定義を新しい定義で書き換える

これらの既存のクラスの拡張のことを破壊的クラス拡張と呼ぶ。実際に破壊的クラス拡張をサポートしている言語として Smalltalk[8] や Ruby[1],AspectJ[9],MultiJava[6],GluonJ[5] などがある。既存のクラスに対して、このような破壊的クラス拡張を行えることは、プログラマに多彩な表現方法を与えるとともに、プログラムを書く上での自由度の向上にもつながる。しかし、いくつかの問題もはらんでいる。既存のメソッドに変更を加えることによって、そのメソッドを使っている他の部分に影響が生じる可能性が出てくるのである。また、同じクラスの同じ名前のメソッドを別々のライブラリが定義していた場合、それらのライブラリを同時に使用すると、メソッド名の衝突が起きてしまう。これらは破壊的クラス拡張の影響がプログラム全体に及んでしまうために、起きる問題である。

この問題を克服する既存の研究として、Classboxes[3] や Method Shelters[2] などがあげられる。これらのシステムでは、メソッド定義のスコープをコントロールすることによって、破壊的クラス拡張の影響がプログラム全体に及ぶことを防いでいる。特に Method Shelters はチェンバーと呼ばれるアイデアを採用することで、Classboxes やその他の関連研究の欠点を補うことが可能となっている。

しかし、Method Shelters には欠点もある。第一にチェンバーを導入することでセマンティクスの複雑さが増してしまったこと。第二に言語に Method Shelters を導入する際、Method Shelters のメソッド探索のセマンティクスをそのまま採用してしまうと、実行速度の遅さが懸念されてしまうことである。Method Shelters は独自のメソッド探索の手法を持っている。そのため、VM に手を入れずにそのまま導入してしまうと、通常のメソッドディスパッチに加え、Method Shelters のメソッド探索手法も用いるため、Method Shelters を導入する前より実行時間が遅くなってしまう。

本研究では、Method Shelters を簡素化した Simplified Method Shelters を提案する。Simplified Method Shelters の Method Shelters からの大きな変更点は、セマンティクスの複雑さの原因であったチェンバーをなくしたことである。これによって、セマンティクスの複雑さを減らし、理解のしやすいセマンティクスを提供することが可能となった。さらに、セマンティクスからいくつかの性質を導き出すことによって、Simplified Method Shelters の基本的な性質を明確化し、さらなる分かりやすさにつとめた。

また、実行時間の遅さの改善のため、本研究では実装方法の提案も行う。Simplified Method Shelters のメソッド探索のセマンティクスでは、動的な情報を必要とするため、実行時探索を行なっている。本研究では、この動的な情報を減らすことで、実行時のコストを減らすことができるのではないかと考えた。さらに、複数存在している同名のメソッドのメソッド定義の中から、各メソッド呼び出しが呼び出すメソッド定義をコンパイル時に 1 つに絞ることで、通常のメソッドディスパッチのコストに近づけることを可能にした。また、それらがセマンティクスを本当に満たしているのかといった正当性の議論も行った。

以降、本稿は次のような構成で進んでいく。第 2 章では、背景となっている Method Shelters と関連研究について述べる。第 3 章では、Simplified Method Shelters のセマンティクスとそこから導き出せる性質を明確化する。第 4 章では、実装方法を 2 つ提案し、またその正当性についての議論を行う。第 5 章では本研究のまとめ、および今後の課題について述べる。

第2章　背景と関連研究

この章では、本研究の背景となっている Method Shelters やそれに関連する技術、および関連研究について記述する。

2.1 章では Method Shelters の背景である破壊的クラス拡張について、2.2 章では Method Shelters とその問題点、2.3 章では関連研究について述べる。

2.1　破壊的クラス拡張

破壊的クラス拡張を行うと、既存のクラスのメソッドの定義について、クラス宣言のコードに直接手を加えずに、次の 2 つのようなことができる。

- 既存のクラスに新しいメソッドを追加
- 既存のメソッド定義を新しいメソッド定義で置き換える

既存のメソッドの定義を新しいメソッドの定義で置き換えることを再定義と呼ぶことにする。

破壊的クラス拡張の例として、図 2.1 をあげる。この例は、Ruby[1] を使って書かれている。1 行目から 8 行目では Window クラスを定義しており、その中で setBorder メソッドを定義している。また、その setBorder メソッドを呼び出している setFrame メソッドも定義されている。現在、この定義の setBorder メソッドは WindowsXP のボーダーを定義している。このとき、11 行目から 15 行目のように Window クラスの外で Window クラスの setBorder メソッドを定義する。すると、setFrame メソッド内の setBorder メソッドの呼び出しでは、12 行目から 14 行目で定義された setBorder メソッドの定義が呼び出される。こうすることで、Window クラスの定義を書き換えることなく、Window クラスの中のメソッドの定義を再定義することができる。

このようなメソッドに対する変更というのは、サブクラスを用いても行うことができる。しかし、サブクラスを用いてクラス拡張を行うと、クラス拡張をプログラムに適用するために、オブジェクトの生成を行う必要がある。また、すでにそのメソッドを使用しているプログラムに適用するためには、オブジェクトの生成部分を書き換える必要がある。

```
class Window
  def setBorder()
    # Windows XPのボーダー
  end
  def setFrame()
    setBorder()
  end
end


class Window
  def setBorder()
    # windows7のボーダー
  end
end
```

図 2.1: 破壊的クラス拡張のコード例

破壊的クラス拡張を用いれば、このようなオブジェクトの生成に関する追加や変更を行わなくても、メソッドの定義を変更することができる。

2.1.1　クラス拡張の問題点

クラス拡張の問題点としてメソッド名の衝突があげられる。2 つのライブラリが同じクラスの同じ名前のメソッドを再定義していたとしよう。この 2 つのライブラリを同時に使用すると、このメソッド名は衝突してしまう。

前章であげた Ruby では、メソッド名の衝突が起きている場合、一番最後にされた再定義を適用する。しかし、それでは最後の再定義を行なっているライブラリ以外の場所でそのメソッドを使っていると、プログラマの意図せぬ動作が起きてしまう可能性がある。

さらにもう 1 つの問題点として、メソッド定義の変更がそのメソッドを使用している他の部分にまで、影響を及ぼすことがあげられる。これもまた、プログラマの意図せぬ動作を引き起こすことにつながる。

これらは、クラス拡張による変更がプログラム全体に影響してしまうためにおこる問題である。この問題を解決する方法として Classboxes[3] や Method Shelters[2] などが存在する。Classboxes や Method Shelters はメソッド定義のスコープをコントロールすることで、これらの問題の解決に取り組んだ。本研究ではこの Method Shelters に着目した。

2.2 Method Shelters

Method Shelters[2] はクラス拡張によるメソッド名の衝突を避けることができるモジュールシステムである。Method Shelters は method shelter というモジュールを持ち、以下の 5 つの機能を有する。

1. メソッド定義のスコープのコントロールが可能
2. 他の method shelter をインポートすることが可能
3. インポートされた method shelter の中のメソッド定義を再定義することが可能
4. 再定義からメソッド定義を守ることが可能
5. メソッド探索に曖昧さがない

Method Shelters のアイデアは、クラス拡張の効果を (i) そのメソッドを定義している method shelter 内、と (ii) 定義している method shelter をインポートしている method shelter 内、に限定するというものである。

method shelter は exposed チェンバーと hidden チェンバーという 2 つのチェンバーを持つ。それぞれのチェンバーは、インポート宣言とメソッド定義を含む。インポート宣言では、他のシェルターをインポートすることができる。メソッド定義では、既存のクラスに新しいメソッドを足す、あるいは既存のメソッドを再定義することができる。

図 2.2 が Method Shelters のコード例である。この例では、win7,virusCheck,app の 3 つの method shelter が定義されている。これ以降 “method shelter” のことを “シェルター” とも記述する。20,29,30 行目がインポート宣言である。virusCheck シェルターが win7 シェルターをインポートしており、app シェルターが win7 シェルターと virusCheck シェルターをインポートしている。

文字列 “hide” の後に記述されているのが、hidden チェンバー内のインポート宣言やメソッド定義である。図 2.2 の例では、virusCheck シェルター内の 19 行目から 25 行目までが virusCheck シェルターの hidden チェンバーを表す。hidden チェンバー内のインポート宣言およびメソッド定義はそのシェルターをインポートしているシェルターからは見ることができない。そのため、virusCheck シェルターをインポートしている app シェルターからは、virusCheck シェルターが win7 シェルターをインポートしていることも、setBorder メソッドを再定義していることも分からない。このとき、app シェルター内で makeFrame メソッドと makeVirusCheckFrame メソッドを呼び出すと、それぞれのメソッド内の setBorder メソッドの呼び出しがどの setBorder メソッドの定義を呼び出すか考えてみる。

メソッド呼び出しを行うとき、呼び出しを行なっているシェルターの hidden チェンバー内の定義あるいは hidden チェンバーでインポートしている定義が優先的に呼び出される。そのため、virusCheck シェルターで定義されている makeVirusCheckFrame メソッドの中の setBorder メソッドの呼び出しでは、hidden チェンバー内で定義されている setBorder メソッドのメソッド定義が呼び出される。

win7 シェルターは app シェルターに exposed チェンバーでインポートされ、さらに setBorder メソッドを再定義されている。そのため、app シェルターで makeFrame メソッドを呼び出すと、再定義が適用され、app シェルター内で定義された setBorder メソッドが呼び出される。

```
1  shelter :win7 do
2    class Window
3      def setBorder()
4        # win7のボーダー
5      end
6      def makeFrame()
7        setBorder()
8      end
9    end
10 end
11
12 shelter :virusCheck do
13   class Window
14     def makeVirusCheckFrame()
15       setBorder()
16     end
17   end
18
19   hide
20   import :win7
21   class Window
22     def setBorder()
23       # ウイルスチェック用の特殊なボーダー
24     end
25   end
26 end
27
28 shelter :app do
29   import :win7
30   import :virusCheck
31   class Window
32     def setBorder()
33       # 新しいボーダー
34     end
35   end
36   def eval()
37     makeFrame()  # appシェルターで定義されたsetBorderメソッド
38     makeVirusCheckFrame() # virusCheckシェルターで定義された
39                           # setBorderメソッド
40   end
41 end
```

図 2.2: Method Shelters のコード例

Method Shelters では、このように hidden チェンバーをうまく使うことによって、メソッド定義のスコープをコントロールすることを可能にし、メソッド名の衝突を避けられるようにしている。

Method Shelters のメソッド探索の順序をさらにはっきりさせるために、図 2.3 のような例を用いる。この図では、A,B,C,D の 4 つのシェルターが存在し、それぞれの exposed チェンバーを実線の四角で、hidden チェンバーを点線の四角で表している。さらに、そのインポート関係を矢印を使って表している。インポート関係は (インポートしているシェルター)⟶(インポートされているシェルター) で表す。この図では、シェルター A がシェルター B を exposed チェンバーでインポートし、シェルター B がシェルター C を exposed チェンバーで、シェルター D を hidden チェンバーでインポートしている。

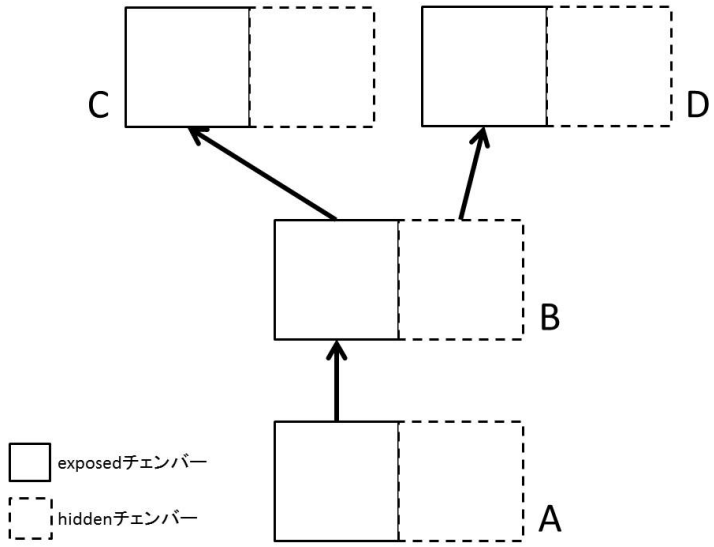

図 2.3: Method Shelters の例

シェルター B でメソッド呼び出しが行われた時にどのチェンバーをどの順序で探索するのか考える。探索順序は図 2.4 のようになる。前述したとおり、hidden で定義しているメソッド定義および hidden でインポートしているメソッド定義が優先的に呼び出される。そのため、図 2.4 の ① → ② のようになる。そこで見つからなければ、exposed チェンバーを探索するのだが、インポートされた先で再定義されている可能性があるため、シェルター B をインポートしているシェルター A の exposed チェンバーも探索する。もし、シェルター A が別のシェルターにインポートされているならば、そのシェルターの exposed チェンバーも探索する必要がある。つまり、exposed チェンバー側の探索を行う際には、再定義されている可能性

のあるチェンバーまで戻って探索する必要がある。このことより、図 2.4 の ③ → ④ → ⑤ のような探索順序となる。

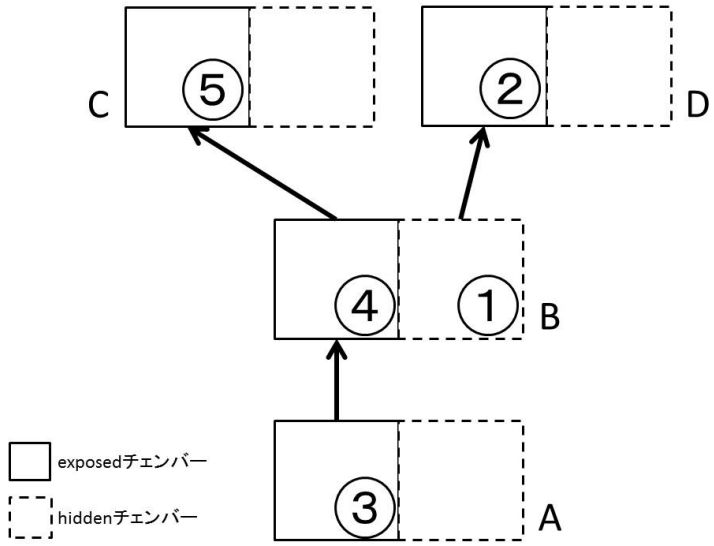

図 2.4: 図 2.3 の B でメソッド呼び出しが行われた場合

これ以降の章では、Method Shelters のセマンティクスから導き出される問題点について述べる。

2.2.1　chamber が存在することによる複雑さ

Method Shelters には hidden チェンバーと exposed チェンバーという 2 つのチェンバーが存在する。Method Shelters ではユーザがこのチェンバーを駆使することで、メソッドのスコープのコントロールや、メソッドを再定義から守ることを可能にしていた。

しかし、シェルターとチェンバーというメソッド定義を格納する入れ物の単位を 2 つ用意することによって、このシステムの複雑さは増してしまったと言える。シェルター単位で考える部分とチェンバー単位で考える部分が混在してしまったのだ。

Method Shelters では、再定義は “チェンバー単位” で行われる。図 2.3 で例を示そう。シェルター B はシェルター D を hidden チェンバーでインポートしている。シェルター D の exposed チェンバーでメソッド m の定義がされており、シェルター B の exposed チェンバーでもメソッド m の定義がされていたと仮定する。このとき、シェルター B の exposed チェンバーでのメソッド m の定義はシェルター D の exposed チェンバーのメソッド m の定義を再定義してはいない。なぜなら、シェルター B 内でメソッ

ド m の呼び出しを行うと、hidden チェンバーでインポートされている定義が優先されることより、シェルター D で定義されたメソッド定義が呼び出されるためである。シェルター B のメソッド定義は、シェルター D のメソッド定義と並列に存在するだけで、定義を置き換えているわけではない。このように、hidden チェンバーでインポートした定義を exposed チェンバーで再定義することはできない。

逆も同じようなことが言える。シェルター B はシェルター C を exposed チェンバーでインポートしている。シェルター C の exposed チェンバーでメソッド n の定義がされており、シェルター B の hidden チェンバーでメソッド n の定義がされていたと仮定する。このとき、シェルター A からメソッド n の呼び出しを行うと、シェルター B の hidden チェンバーはシェルター A からは見えないため、シェルター C の exposed チェンバーのメソッド n の定義が呼び出される。よって、exposed チェンバーでインポートした定義を hidden チェンバーで再定義することはできていない。

このように、再定義というのはチェンバー単位で行われている。

しかし、メソッド探索はどうだろう。メソッド探索でどのチェンバーをどの順番で探索するかは現在いるチェンバーに関係なく、現在いるシェルターによって決まる。例えば、図 2.3 のシェルター B でメソッド呼び出しが行われた場合、その呼び出しが hidden チェンバーで行われたか exposed チェンバーで行われたかに関係なく、図 2.4 のような順序で行われる。メソッド探索をどの順番で行うかは、チェンバー単位でなく、シェルター単位で決定される。

かといって、メソッド探索に関する考え方全てがシェルター単位で決定されるわけではない。前述したとおり、Method Shelters のメソッド探索のセマンティクスでは exposed 側の探索を行う際、再定義をされている可能性があるため、インポートされているシェルターのチェンバーをたどっていく必要がある。どこまでたどっていけばいいのかを考えてみると、再定義されることのないチェンバーにたどり着くまでたどる必要があることが分かる。再定義されることのないチェンバーとは、何からもインポートされていないシェルターの exposed チェンバー、あるいは任意のシェルターの hidden チェンバーである。このように、exposed 側の探索でインポートされているシェルターをどこまでたどるかは、チェンバー単位で考えなければならない。

以上のように、Method Shelters のセマンティクスでは、チェンバー単位で考えなければならない部分とシェルター単位で考えなければならない部分が混在してしまっている。これは、セマンティクスの複雑さを招いている。本研究では、Method Shelters のセマンティクスの複雑さを改善し、より分かりやすいセマンティクスを探求していく。

2.2.2　実装する上での問題点

メソッド探索に関して実装する上での問題点もあげられる。Method Shelters には、前述したような Method Shelters 独自のメソッド探索の手法が存在する。これは同じメソッド名を持つメソッド定義が複数存在する場合があるためである。

例えば、Java に Method Shelters を導入することを考えてみよう。Java の VM には手を加えずにそのまま Method Shelters のメソッド探索のセマンティクスを Java に取り入れたとする。Method Shelters 付きの Java でメソッド探索を行うと、Method Shelters 独自のメソッド探索を行った後、Java の通常のサブクラス関係を利用したメソッドディスパッチを行うこととなる。これは、通常の Java のメソッドディスパッチに比べると、遅くなることが容易に考えられる。

2.3　関連研究

2.3.1　MultiJava

MultiJava[6, 11] は Open class と Multiple dispatch を Java に導入した言語である。本研究と関連が深いのは、Open class に関する部分である。Open class を用いると、既存のクラスのコードを変更せずに新しいメソッドを足すことができる。また、Open class によって、足した新しいメソッドは (i) 定義しているパッケージ、(ii) 定義しているパッケージをインポートしているパッケージ、内でのみ使うことができる。このように、新しいメソッドの定義についてスコープが制限されている。

しかし、MultiJava では、メソッドの再定義を行うことができない。そのため、同名のメソッド定義を複数用意できる Method Shelters のような柔軟な表現方法は持ち合わせていない。

2.3.2　Jiazzi

Jiazzi[10] は program units[7] を導入した Java ベースのコンポーネントシステムである。Jiazzi での unit はコンパイルされた Java コードのコンテナとして存在する。unit は複数のクラスをカプセル化し、それらをインポートしたり、エクスポートしたりすることができ、また他の unit がエクスポートしたクラス集合もインポートすることができる。

Jiazzi では、unit を用いて Open class のような表現をすることができる。インポートしたクラスに対して、新しいメソッドを加えることが可能であるし、異なる unit 中で同じクラスの同名のメソッドを定義することも可能

である。しかし、元のクラスに変更を加えているのではなく、実際には元のクラスのサブクラスとして生成される。さらに、コードを書く側もそれを念頭において実装するように設計されている。そのため、実際にクラス拡張しているというよりは、サブクラスを用いてクラス拡張のような表現をしているに過ぎない。

2.3.3　Classboxes

Classboxes[3] と Classbox/J[4] は classbox というモジュールを言語に導入している。Classbox/J は言語の中でも、Java 言語に classbox を導入したことに関する研究である。classbox を用いると、クラス拡張のスコープを制限することができる。

classbox は、定義とインポート宣言を含むモジュールである。インポート宣言では、他の classbox からクラスや変数をインポートすることができる。インポートしたクラスや変数は、使うことも可能であるし、新しいメソッドを足したりメソッドを再定義したりなど、クラス拡張も行うことができる。classbox 内でされた定義は、(i) 定義している classbox、(ii) それをインポートしている classbox、でしか扱うことはできない。このように、classbox を用いることで、クラス拡張の有効な範囲を制限することができる。

classbox には、クラス拡張のスコープを制限できるというメリットもあるが、また欠点もある。図 2.5 がその例である。図中には win7CB,virusCheckCB,appCB の 3 つの classbox が存在する。win7CB では Window クラスの setBorder メソッドが定義されており、virusCheckCB では Window クラスの setBorder メソッド、Frame クラスの makeVirusCheckFrame メソッドが定義されている。このとき、appCB で win7CB から Window クラスをインポートし、virusCheckCB から Frame クラスをインポートしたとする。classbox では local rebinding が採用されているため、appCB 内の Frame クラスの makeVirusCheckFrame メソッドの中の setBorder メソッドの呼び出しでは、appCB 内のメソッド、つまり win7CB で定義された setBorder メソッドが呼び出されてしまう。そのため、ボーダーは makeVirusCheckFrame メソッドを定義した virusCheckCB 内のボーダーと変わって、win7 用ボーダーになる。appCB 内で win7CB 内で定義したときの動作をする setBorder メソッド、virusCheckCB 内で定義したときの動作をする makeVirusCheckFrame メソッドが使いたいと思ってインポートしたときに、その動きが変わってしまうというのを避けることは classbox ではできない。また、このことは、プログラマの意図せぬ動作を引き起こすことにもつながりかねない。

classbox はモジュールではなく、その中のクラスや変数をインポートする。そのため、モジュールで定めた動きと変わってしまうことがあり、ま

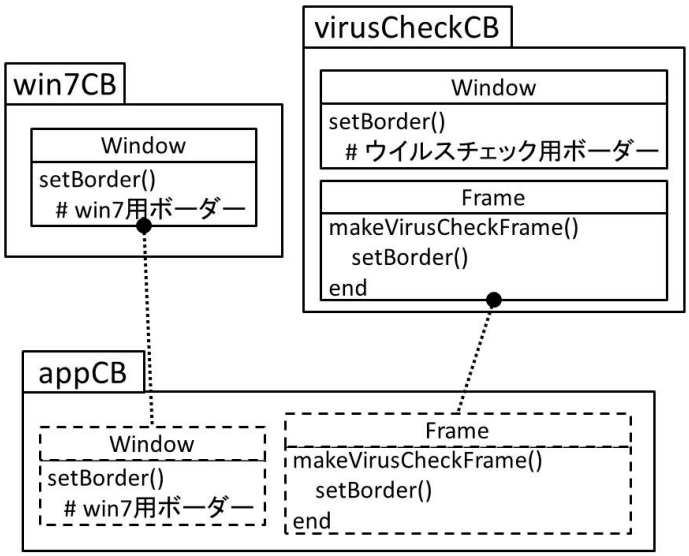

図 2.5: Classbox の問題例

たそれを避けることが難しくなっている。Method Shelters では hidden チェンバーを使うことによって、local rebinding を避けたいときには、プログラマが避けることができるようにしている。また、モジュールごとインポートすることによって、hidden チェンバーを用いた動作がインポートで変わってしまうことをプログラマが防ぐことができるようにしている。Method Shelters では、プログラマが hidden チェンバーと exposed チェンバーを使い分けることによって、local rebinding を適用することも避けることも自在にできるようになっている。

第3章　Method Sheltersのセマンティクスの簡素化と基本的な性質の明確化

従来の Method Shelters のセマンティクスは前章で述べたように、複雑なものであった。本章では、その Method Shelters を簡素化した Simplified Method Shelters を提案すると共に、その定義および性質を明確化していく。

3.1　Simplified Method Shelters

Simplified Method Shelters は simplified method shelter というモジュールを持つ、以下のようなことができるモジュールシステムである。

1. メソッド定義のスコープをコントロールすることができる
2. 他の simplified method shelter をインポートすることができる
3. インポートした simplified method shelter の中のメソッド定義を再定義することができる
4. 既存のクラスに新しいメソッドを追加することができる
5. メソッド定義を再定義から守ることができる

simplified method shelter の中では、revise とインポート宣言の２つを定義することができる。revise では、既存のクラスに新しいメソッドを足すこととインポートしているシェルター内のメソッド定義を新しい定義で置き換えること (再定義) ができる。インポートには exposedly なインポートと hiddenly なインポートの２種類が存在し、インポート宣言をすることで他の simplified method shelters をインポートすることができる。

シェルター内で revise されたメソッド定義はインポートしなければ使うことはできない。こうすることで、シェルターのインポートの連鎖の中と外の間にスコープを設けることができる。さらに、exposedly なインポー

トと hiddenly なインポートを駆使することで、インポート連鎖内にスコープを制限することも可能である。

Simplified Method Shelters のコード例は図 3.1 のようになる。これ以降、“simplified method shelter” のことを “シェルター” とも記述する。図 3.1 では、stdGUI,win7,virusCheck,app の 4 つのシェルターを定義している。インポート宣言は 5,17,29,30 行目に書かれている。win7 シェルターは stdGUI シェルターを exposedly にインポートしている。virusCheck シェルターは stdGUI シェルターを exposedly にインポートしている。app シェルターは exposedly に win7 シェルターを、hiddenly に virusCheck シェルターをインポートしている。main シェルターは app シェルターを exposedly にインポートしている。

さらに、それぞれのシェルターで revise が行われている。win7 シェルターでは、Window クラスの setBorder メソッドと makeFrame メソッドを revise している。これは、setBorder メソッドが Window クラス内にすでに存在する場合にはその定義を再定義し、存在しない場合には Window クラス内に新しく setBorder メソッドを定義することを表す。virusCheck シェルター内では、インポートした setBorder メソッドをさらに再定義し、makeVirusCheckFrame メソッドも定義している。win7 シェルターの makeFrame メソッド、virusCheck シェルター内の makeVirusCheckFrame メソッドは、それぞれ setBorder メソッドを呼び出している。app シェルター内では、さらに Window クラスの setBorder メソッドを再定義し、makeFrame メソッドと makeVirusCheckFrame メソッドを呼び出すような main メソッドを定義している。

さて、このとき、app シェルター内の main メソッドを呼びだしたとする。main メソッドの中では、makeFrame メソッドと makeVirusCheckFrame メソッドが呼び出されている。このとき、makeFrame メソッド内と makeVirusCheckFrame メソッド内の setBorder メソッドの呼び出しでは、win7 シェルター内、virusCheck シェルター内、app シェルター内で定義された 3 つの setBorder メソッドのうち、どの setBorder メソッドの定義を呼び出すだろうか。

まず、makeFrame メソッド内の setBorder メソッド呼び出しについて考える。makeFrame メソッドが定義されている win7 シェルターは app シェルターに exposedly にインポートされている。そのため、インポート先でされた再定義が適用され、makeFrame メソッド内の setBorder メソッドの呼び出しでは app シェルターの setBorder メソッドの定義が呼び出される。

次に makeVirusCheckFrame メソッド内の setBorder メソッド呼び出しについて考える。hiddenly にインポートされたシェルターの中に一度入ると、外のシェルター内の revise の影響を受けなくなる。つまり、makeVir-

usCheckFrame メソッドを呼び出し、hiddenly にインポートしている virusCheck シェルターの中に入ると、app シェルター、win7 シェルターの影響を受けなくなる。そのため、makeCirusCheckFrame メソッド内の setBorder メソッドの呼び出しでは、virusCheck シェルター内の setBorder メソッドの定義が呼び出される。

また、この例では示していないが、hiddenly にインポートしているシェルターの影響は直接インポートしているシェルターにしか及ばない。そのため、virusCheck シェルター内のメソッド定義は app シェルターには影響を与えるが、win7 シェルターからは virusCheck シェルターの存在すら感知することができない。また、app シェルターをインポートしているシェルターがあった場合、そのシェルターからも virusCheck シェルターの存在は感知できない。よって、virusCheck シェルター内のメソッド定義は上書きされることがないため、再定義からメソッド定義を守ることにつながる。

このように、Simplified Method Shelters では、revise 機能を用いることで新しいメソッドの追加やメソッドの再定義といったクラス拡張が可能となり、hiddenly なインポートと exposedly なインポートを使い分けることで他の simplified method shelter をインポートし、そのメソッド定義のスコープのコントロールとメソッド定義を再定義から守ることを可能とした。

グローバルメソッド　シェルターの中で定義されていないメソッドをグローバルメソッドと呼ぶ。Simplified Method Shelters では、グローバルメソッドは名前のないシェルターに入っており、その名前のないシェルターは全てのシェルターに exposedly にインポートされているかのように扱われる。このようにすることで、グローバルメソッドはどのシェルターからも再定義することができる。

***main* メソッド**　*main* メソッドは各シェルターに 1 つ定義することができる。シェルターの中のメソッド定義は外から見ることができない。そのため、シェルターの中のメソッドを外から呼び出すことはできないが、*main* メソッドだけは特別に外から呼ばれることを許している。*main* メソッドを実行することで、そのシェルターの中に入ることができ、上で記述したようなインポート関係や revise を用いたシェルターを使うことができる。図 3.1 の 35 行目が *main* メソッドの例である。

これ以降の節では、Method Shelters からの変更点および revise や import 宣言といった機能の詳細など、Simplified Method Shelters のさらに細かい部分について述べていく。

```
1  shelter stdGUI{
2       :
3  }
4  shelter win7{
5    import stdGUI;
6    revise Window{
7      void setBorder(){
8        // winddows7用のボーダー
9      }
10     void makeFrame(){
11       setBorder();
12     }
13   }
14 }
15
16 shelter virusCheck{
17   import stdGUI;
18   revise Window{
19     void setBorder(){
20       // ウイルスチェック用の特殊なボーダー
21     }
22     void makeVirusCheckFrame(){
23       setBorder();
24     }
25   }
26 }
27
28 shelter app{
29   import win7;
30   hidden import virusCheck;
31   revise Window{
32     void setBorder(){
33       // 新しいボーダー
34     }
35   }
36   void main(){
37     makeFrame();    // appシェルターで定義されたsetBorderメソッド
38     makeVirusCheckFrame();  // virusCheckシェルターで定義された
39                             // setBorderメソッド
40   }
41 }
```

図 3.1: Simplified Method Shelters のコード例

3.1.1　Method Shelters からの変更点

機能の説明の前に Method Shelters から Simplified Method Shelters への変更点について述べる。大きな変更点として、以下の 2 つがある。

チェンバーの廃止　最も大きな変化は、チェンバーを廃止したことである。元々の Method Shelters には、hidden チェンバーと exposed チェンバーが存在しており、それらを駆使することによってメソッド定義のスコープのコントロールを行っていた。しかし、前章で述べたように、チェンバーの存在によって Mfethod Shelters の意味論の複雑さは増してしまった。そこで、Simplified Method Shelters では、チェンバーの考え方そのものをなくすことにした。

2 種のインポートの導入　Simplified Method Shelters では、インポートを hiddenly なインポートと exposedly なインポートの 2 種に分けた。これはチェンバーの代わりとして導入されたアイデアである。Method Shelters の大きな特徴の一つとして、スコープのコントロールが可能なことがあげられるが、Method Shelters ではチェンバーを駆使することによりこれを可能としていた。単純にチェンバーを廃止しただけだと、これらの特徴は失われてしまうが、Simplified Method Shelters は 2 種のインポートを導入することにより、スコープのコントロールを可能とした。

3.1.2　revise 機能

revise 機能を用いることで、以下の 2 つのことが可能となっている。

- 既存のクラスに新しいメソッドを追加できる
- インポートしているシェルター内のメソッド定義を再定義できる

3.1 章で述べたように、グローバルな定義はすべてのシェルターに exposedly にインポートされているかのように扱われる。そのため、グローバルなメソッド定義はどのシェルターからも再定義することができる。

3.1.3　import 宣言

Simplified Method Shelters のインポートには、exposedly なインポートと hiddenly なインポートの 2 種類が存在する。それぞれ他の Simplified Method Shelters をインポートすることが可能である。

インポートに関する定義を定める。α, β, γ をシェルターを表すメタ変数とする。

定義 3.1.1.

- α が β を exposedly にインポートしていることを $\alpha \xrightarrow{e} \beta$ と書く
- α が β を hiddenly にインポートしていることを $\alpha \xrightarrow{h} \beta$ と書く
- α が β を exposedly にも hiddenly にもインポートしていないことを $\alpha \not\longrightarrow \beta$ と書く

定義 3.1.2.

- $\alpha \xrightarrow{e} \beta, \beta \xrightarrow{e} \gamma \Longrightarrow \alpha \xrightarrow{e} \gamma$
- $\alpha \xrightarrow{h}, \beta \xrightarrow{e} \gamma \Longrightarrow \alpha \xrightarrow{h} \gamma$

定義 3.1.2 より、次のような性質を言うことができる。

性質 3.1.1. exposedly なインポートは推移性を持つ

これらの定義および性質より、図 3.1 の例は、以下のように表現することができる。

$$\begin{array}{ll} S2 \xrightarrow{e} S0 & S3 \xrightarrow{e} S2 \\ S2 \xrightarrow{h} S1 & S3 \xrightarrow{e} S0 \end{array} \tag{3.1}$$

3.1.4　method shelter dag

Simplified Method Shelters は DAG を用いることでグラフとして表現することができる。DAG は無閉路有向グラフとも呼び、有向グラフのうち、閉路のないもののことを言う。Simplified Method Shelters は環状インポートを許さない。そのため、インポート関係を有向グラフとして表すと無閉路になる。Simplified Method Shelters を DAG で表したものを method shelter dag と呼ぶことにする。しかし、ただ、無閉路有向グラフで書いただけでは、exposedly なインポートと hiddenly なインポートの区別がつかない。そのため、method shelter dag では、exposedly なインポートを実線の矢印で、hiddenly なインポートを点線の矢印で表す。

例えば図 3.1 の method shelter dag は図 3.2 のようになる。インポート関係は、(インポートされているシェルター)⟶(インポートしているシェルター) のように表す。図 3.2 では、app シェルターが win7 シェルターを exposedly に、virusCheck シェルターを hiddenly にインポートし、win7 シェルターと virusCheck シェルターが exposedly に stdGUI シェルターをインポートしている。

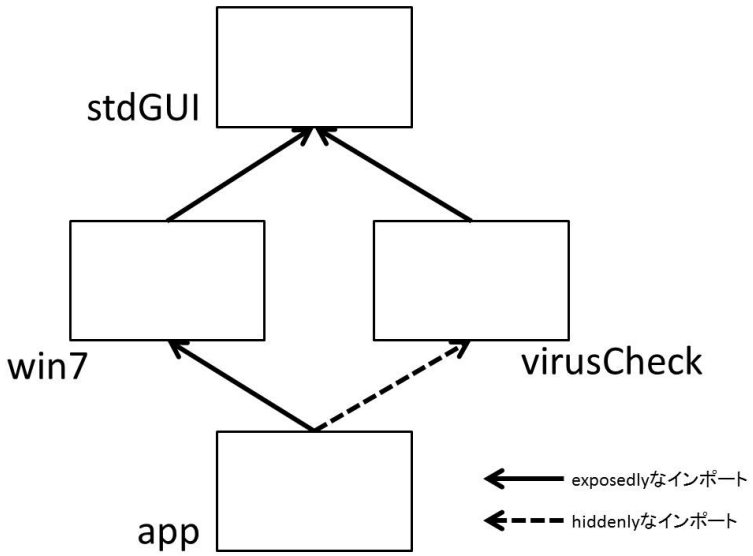

図 3.2: 図 3.1 の method shelter dag

インポートしているシェルターをインポートされているシェルターの子と言う。インポートされているシェルターをインポートしているシェルターの親と言う。図 3.2 の例では、app シェルターは win7 シェルターの子であり、win7 シェルターは app シェルターの親である。

3.2　メソッド探索

この章では、Simplified Method Shelters のメソッド探索のセマンティクスを述べる。3.2.1 章では、メソッド探索のセマンティクスを述べるために文法の一部を定義する。3.2.2 章では、メソッド探索のセマンティクスを示し、3.2.3 章では、3.2.2 章のメソッド探索のセマンティクスを scheme のコードで表すことで、セマンティックの分かりやすさの向上につとめている。

3.2.1　文法

メソッド探索を定義するために、Simplified Method Shelters の文法の一部を定義する。定義は次のようになる。

$$\begin{aligned}
\texttt{SL} &::= \texttt{shelter S}\{\overline{\texttt{IL}}; \overline{\texttt{HIL}}; \overline{\texttt{RL}}\} && (3.2)\\
\texttt{IL} &::= \texttt{import S} && (3.3)\\
\texttt{HIL} &::= \texttt{hidden import S} && (3.4)\\
\texttt{RL} &::= \texttt{revise C}\{\overline{\texttt{M}}\} && (3.5)\\
\texttt{M} &::= \texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}}\{\text{body}\}) && (3.6)
\end{aligned}$$

文法の記述を用意にするために、オーバーラインをひいたアルファベットを数列を表すために用いる。例えば、$\overline{\texttt{M}}$ は $\texttt{M}_1, \texttt{M}_2, ..., \texttt{M}_n$ を表し、$\overline{\texttt{C}}\ \overline{\texttt{x}}$ は $\texttt{C}_1\ \texttt{x}_1, \texttt{C}_2\ \texttt{x}_2, ..., \texttt{C}_n\ \texttt{x}_n$ を表す。S はシェルター名、C はクラス名、m はメソッド名を表すメタ変数とする。

SL はシェルター定義を表し、シェルター名、exposedly なインポート宣言、hiddenly なインポート宣言、revise の定義を含む。IL は exposedly なインポート宣言を、HIL は hiddenly なインポート宣言を表し、どちらもシェルター名を含む。RL は revise の定義を表し、クラス名とメソッド定義を含む。M はメソッド定義を表している。

3.2.2　定義

この節では、メソッド探索に関する定義を定めていく。

メソッド探索を行う関数を $\mathit{mbody}(\texttt{m}, \texttt{C}, \texttt{S}, \texttt{S}_\texttt{S})$ とする。関数 *mbody* は、ソースシェルターを $\texttt{S}_\texttt{S}$ として、シェルター S の中でクラス C、メソッド名 m が呼び出されたとき、そのメソッドボディを $\overline{\texttt{x}}$.body in $\texttt{T}(\texttt{T}_\texttt{S})$ という形で返す。現在いるシェルター S のことをカレントシェルターと呼ぶ。また、$\overline{\texttt{x}}$ がパラメータ、body がメソッドボディであり、T はメソッドボディが見つかったシェルター、$\texttt{T}_\texttt{S}$ は次探索で使うソースシェルターを表す。つまり、$\mathit{mbody}(\texttt{m}, \texttt{C}, \texttt{S}, \texttt{S}_\texttt{S})$ を行い、$\overline{\texttt{x}}$.boydy in $\texttt{T}(\texttt{T}_\texttt{S})$ が見つかった後、さらに body の中でメソッド呼び出しがあった場合には、カレントシェルターを T、ソースシェルターを $\texttt{T}_\texttt{S}$ としてメソッド探索が行われるということである。ソースシェルターはメソッド探索で使う引数であり、その前のメソッド探索の結果に依存する。ソースシェルターの初期値はカレントシェルターと同じであるとする。

関数 $\mathit{mbodyimport}(\texttt{m}, \texttt{C}, \texttt{S})$ は関数 *mbody* を定義するために使う関数である。メソッド名 m、クラス C、シェルター S を引数とし、パラメータ $\overline{\texttt{x}}$、メソッドボディ body、シェルター T を $\overline{\texttt{x}}$.body in T という形で返す。

関数 $\mathit{mbodyglobal}(\texttt{m}, \texttt{C}, \texttt{S}, \texttt{S}_\texttt{S})$ はメソッド名 m、クラス C、シェルター S、ソースシェルター $\texttt{S}_\texttt{S}$ を引数とし、パラメータ $\overline{\texttt{x}}$、メソッドボディ body を

$\overline{x}$.body in S(S_S) という形で返す。この関数では、シェルター S とソースシェルター S_S の値は変化せずに、そのまま返される。

関数 *mbody* の定義は図 3.3 のようになる。

図 3.3 より、関数 *mbodyimport* と関数 *mbodyglobal* について次のようなことが言える。

- 関数 *mbodyimport* は、与えられたシェルター内を探索し、なければ与えられたシェルターが exposedly にインポートしているシェルターを関数 *mbodyimport* を使って探索する関数である
- 関数 *mbodyglobal* は、グローバルメソッドを探す関数である

このことと、関数 *mbody* の定義より、Simplified Method Shelters のメソッド探索について次のような定理が成り立つ。

定理 3.2.1. Simplified Method Shelters のメソッド探索は次のような順序で行われる。

1. hiddenly にインポートしているシェルターから exposedly にインポートしているシェルターをたどってメソッドを探す
2. 1 で見つからなければ、ソースシェルターから exposedly にインポートしているシェルターをたどってメソッドを探す
3. 2 で見つからなければ、グローバルな定義からメソッドを探す

定理 3.2.1 の 1 は図 3.3 中の式 (3.7) と関数 *mbodyimport* から言うことができる。2 は図 3.3 中の式 (3.8) と関数 *mbodyimport* から、3 は図 3.3 中の式 (3.9) と関数 *mbodyglobal* から言うことができる。

定理 3.2.1 の 1 の探索を hidden 探索、定理 3.2.1 の 2 の探索を exposed 探索と呼ぶことにする。

例を示す。図 3.2 の app シェルターをソースシェルターとして、app シェルターでメソッド呼び出しが行われたと仮定する。このとき、関数 *mbody* の定義や定理 3.2.1 より、探索順序は図 3.4 のようになる。各シェルターの中の右下にある丸の中の数字が探索順序を示している。① → ② が hidden 探索であり、③ → ④ → ⑤ が exposed 探索である。

hidden 探索では、メソッドがただ 1 つ見つかったならば返すように定義されている。これは、図 3.3 中の式 (3.7) から言うことができる。返す条件に合致しているメソッドが 2 つ以上見つかってしまった場合には、エラーを出すことにする。exposed 探索についても同様に、条件に合致するメソッドが 2 つ以上見つかってしまった場合には、エラーを出す。これらエラーの条件については、次の節の scheme による実装にてさらにはっきりさせる。

$$\frac{\begin{array}{c}\texttt{shelter S}\{\overline{\texttt{IL}};\overline{\texttt{HIL}};\overline{\texttt{RL}}\}\\ \texttt{hidden import }\overline{\texttt{h}}\in\overline{\texttt{HIL}}\\ \exists \texttt{h}_\texttt{i}\quad \mathit{mbodyimport}(\texttt{m},\texttt{C},\texttt{h}_\texttt{i})=\overline{\texttt{x}}.\texttt{e in T}\\ \forall \texttt{h}_\texttt{j}(\texttt{i}\neq\texttt{j})\quad \mathit{mbodyimport}(\texttt{m},\texttt{C},\texttt{h}_\texttt{j})\ \texttt{undefined}\end{array}}{\mathit{mbody}(\texttt{m},\texttt{C},\texttt{S},\texttt{S}_\texttt{s})=\overline{\texttt{x}}.\texttt{e in T}(\texttt{h}_\texttt{i})}\tag{3.7}$$

$$\frac{\begin{array}{c}\texttt{shelter S}\{\overline{\texttt{IL}};\overline{\texttt{HIL}};\overline{\texttt{RL}}\}\\ \texttt{hidden import }\overline{\texttt{h}}\in\overline{\texttt{HIL}}\\ \forall \texttt{h}_\texttt{i}\quad \mathit{mbodyimport}(\texttt{m},\texttt{C},\texttt{h}_\texttt{i})\ \texttt{undefined}\\ \mathit{mbodyiter}(\texttt{m},\texttt{C},\texttt{S}_\texttt{s})=\overline{\texttt{x}}.\texttt{e in T}\end{array}}{\mathit{mbody}(\texttt{m},\texttt{C},\texttt{S},\texttt{S}_\texttt{s})=\overline{\texttt{x}}.\texttt{e in T}(\texttt{S}_\texttt{s})}\tag{3.8}$$

$$\frac{\begin{array}{c}\texttt{shelter S}\{\overline{\texttt{IL}};\overline{\texttt{HIL}};\overline{\texttt{RL}}\}\\ \texttt{hidden import }\overline{\texttt{h}}\in\overline{\texttt{HIL}}\\ \forall \texttt{h}_\texttt{i}\quad \mathit{mbodyimport}(\texttt{m},\texttt{C},\texttt{S}_\texttt{s})\ \texttt{undefined}\\ \mathit{mbodyimport}(\texttt{m},\texttt{C},\texttt{S}_\texttt{s})\ \texttt{undefined}\end{array}}{\mathit{mbody}(\texttt{m},\texttt{C},\texttt{S},\texttt{S}_\texttt{s})=\mathit{mbodyglobal}(\texttt{m},\texttt{C},\texttt{S},\texttt{S}_\texttt{s})}\tag{3.9}$$

$$\frac{\begin{array}{c}\texttt{shelter S}\{\overline{\texttt{IL}};\overline{\texttt{HIL}};\overline{\texttt{RL}}\}\\ \texttt{revise C}\{\overline{\texttt{M}}\}\in\overline{\texttt{RL}}\\ \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e}\}\in\overline{\texttt{M}}\end{array}}{\mathit{mbodyimport}(\texttt{m},\texttt{C},\texttt{S})=\overline{\texttt{x}}.\texttt{e in S}}\tag{3.10}$$

$$\frac{\begin{array}{c}\texttt{shelter S}\{\overline{\texttt{IL}};\overline{\texttt{HIL}};\overline{\texttt{RL}}\}\\ \texttt{import }\overline{\texttt{e}}\in\overline{\texttt{IL}}\\ (\texttt{revise C}\{\overline{\texttt{M}}\}\notin\overline{\texttt{C}})\ \texttt{s.t}\ (\texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x'}})\{\texttt{return e'}\}\in\overline{\texttt{M}})\\ \exists \texttt{e}_\texttt{i}\quad \mathit{mbodyimport}(\texttt{m},\texttt{C},\texttt{e}_\texttt{i})=\overline{\texttt{x}}.\texttt{e in T}\\ \forall \texttt{e}_\texttt{j}(\texttt{i}\neq\texttt{j})\quad \mathit{mbodyimport}(\texttt{m},\texttt{C},\texttt{e}_\texttt{j})\ \texttt{undefined}\end{array}}{\mathit{mbodyimport}(\texttt{m},\texttt{C},\texttt{S})=\overline{\texttt{x}}.\texttt{e in T}}\tag{3.11}$$

$$\frac{\begin{array}{c}\texttt{GlobalCT(C) = class C extends D}\{\overline{\texttt{C}}\ \overline{\texttt{f}};\overline{\texttt{M}}\}\\ \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e}\}\in\overline{\texttt{M}}\end{array}}{\mathit{mbodyglobal}(\texttt{m},\texttt{C},\texttt{S},\texttt{S}_\texttt{s})=\overline{\texttt{x}}.\texttt{e in S}(\texttt{S}_\texttt{s})}\tag{3.12}$$

$$\frac{\begin{array}{c}\texttt{GlobalCT(C) = class C extends D}\{\overline{\texttt{C}}\ \overline{\texttt{f}};\overline{\texttt{M}}\}\\ \texttt{m is not defined in }\overline{\texttt{M}}\end{array}}{\mathit{mbodyglobal}(\texttt{m},\texttt{C},\texttt{S},\texttt{S}_\texttt{s})=\mathit{mbody}(\texttt{m},\texttt{D},\texttt{S},\texttt{S}_\texttt{s})}\tag{3.13}$$

図 3.3: 関数 *mbody* の定義

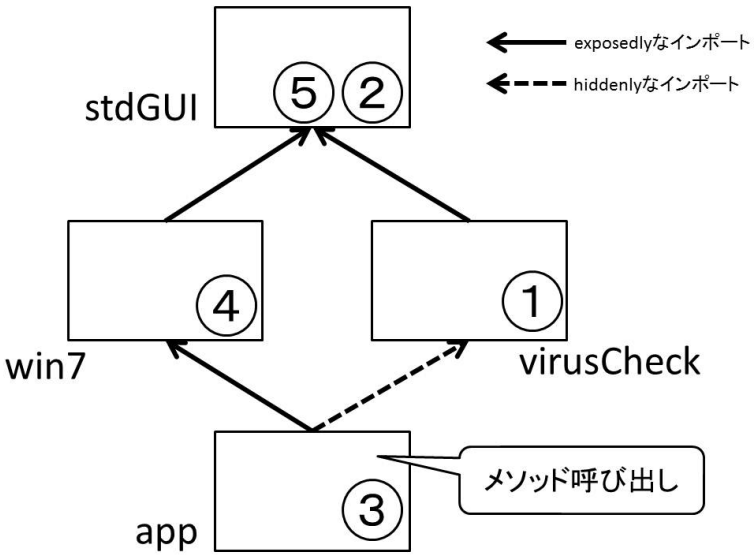

図 3.4: 図 3.2 のメソッド探索順序例

3.2.3　scheme による実装

3.2.2 章では、関数 *mbody* の定義を定めた。この節では、*mbody* を実際に scheme で実装した結果を示すことで、定義の更なるわかりやすさに努める。

関数 *mbody* の scheme での実装は図 3.5 のようになる。

図 3.3 と異なる点として、関数 *mbodyimport* の引数にソースシェルターが追加されている点があげられる。しかし、この変数は *mbodyimport* 内で変更されることはなく、そのままリターンされる。この引数を追加したのは、scheme による実装を読む上でのわかりやすさのためである。残りの部分は図 3.3 の定義ほぼそのままの実装となっている。関数一つ一つについて詳しく見ていく。

関数 *mbody*　図 3.5 の 1 行目から 13 行目までが、関数 *mbody* の実装である。

2 行目から 5 行目までが hidden 探索を行なっている部分である。5 行目の関数 *hiddenly-importings* はシェルターを引数として、そのシェルターが直接 hiddenly にインポートしているシェルターの集合を返す関数である。4、5 行目では関数 *hiddenly-importings* によって得た集合の要素それぞれに対し、関数 *mbody-import* を適用している。さらに、関数 *mbody-import* を適用して得た集合に対し、3 行目で関数 *filter-methods* を適用している。関数 *filter-methods* は集合を調べ、null でない要素がただ 1 つならばそれ

```
1  (define (mbody methodname class shelter source)
2    (let ((hidden-method
3           (filter-methods
4              (map (lambda (s) (mbody-import methodname class s s))
5                   (hiddenly-importings shelter)))))
6      (if hidden-method
7          hidden-method
8          (let
9            ((exposed-method
10              (mbody-import methodname class source source)))
11           (if exposed-method
12               exposed-method
13               (mbody-global methodname class shelter source))))))
14
15 (define (mbody-import methodname class shelter source)
16   (if (method-table-exists? shelter class methodname)
17       (list (method-table-get shelter class methodname) shelter source)
18       (filter-methods
19          (map (lambda (s) (mbody-import methodname class s source))
20               (exposedly-importings shelter)))))
21
22 (define (mbody-global methodname class shelter source)
23   (if (method-global-table-exists? class methodname)
24       (list (method-global-table-get class methodname) shelter source)
25       (mbody methodname (superclass class) shelter source)))
```

図 3.5: scheme による mbody の実装

を返し、0 個であれば null を返す。さらに 2 つ以上あるならば、エラーを出力する。

hidden 探索で見つからなかったならば、9,10 行目で exposed 探索を行う。9,10 行目では、図 3.3 の関数 *mbody* の定義に則った引数を渡し関数 *mbody-import* を呼び出している。

さらに exposed 探索で見つからなかったならば、13 行目で関数 *mbody-global* を呼び出し、グローバルな探索を行う。

関数 *mbody-import*　図 3.5 の 15 行目から 20 行目までが関数 *mbody-import* の実装である。定義と違い、引数としてソースシェルターをとるようになっているが、実装中でソースシェルターの値は変化させていない。関数 *mbody-import* では、与えられたシェルターのメソッドテーブルを見て、メソッド定義が存在するならばそれを返し、存在しないならば与えられたシェルターが直接 exposedly にインポートしているシェルターに対し関数 *mbody-import* を適用するようにしている。20 行目の関数 *exposedly-importings* は関数 *hiddenly-importings* と同じように与えられたシェルターが直接 exposedly にインポートしているシェルターを返す関数である。18 行目では hidden 探索と同様に関数 *filter-methods* が使われており、2 つ以上見つかってしまった場合には、エラーを出力するようになっている。

関数 *mbody-global*　図 3.5 の 22 行目から 25 行目までが関数 *mbody-global* の実装である。グローバルなメソッドテーブル内で発見されたならば、それを返し、存在しないならば定義通り、スーパークラスに対して関数 *mbody* を適用する。

3.3　性質

この章では、前章で述べた Simplified Method Shelters のセマンティクスから導き出せる性質について述べていく。

3.3.1　ソースシェルター

この節では、ソースシェルターに関する性質を述べる。

3.2.2 章で記述したとおり、ソースシェルターの初期値はカレントシェルターの初期値と同じであり、探索を繰り返すごとにソースシェルターがどのシェルターになるかは変化していく。

ソースシェルターには次の 3 つのような性質がある。

性質 3.3.1. ソースシェルターは exposed 探索の出発点である

証明. 定理 3.2.1 の 2 から言うことができる。(証明終)

性質 3.3.2. 次探索のソースシェルターは hidden 探索でメソッドが見つかった場合のみ現在のソースシェルターと異なる

証明. 関数 *mbody* の定義である図 3.3 中の式 (3.7)、式 (3.8)、式 (3.9) を見て欲しい。このうち、入力のソースシェルターと出力のソースシェルターが変化しているのは、式 (3.7) のみである。式 (3.7) は hidden 探索でメソッド定義が見つかる場合を表しているため、次探索のソースシェルターは hidden 探索でメソッドが見つかった場合のみ変わることが言える。(証明終)

性質 3.3.3. ソースシェルターとなりうるシェルターは、初期値あるいはいずれかのシェルターに直接的に hiddenly にインポートされているシェルターのみである

証明. 性質 3.3.2 より、ソースシェルターが変わるのは、hidden 探索で見つかったときのみである。そのため、hidden 探索を定義している関数 *mbody* の定義である図 3.3 中の式 (3.7) を見る。これの返り値を見ると、メソッドが見つかった場合には、直接的に hiddenly にインポートしているシェルターを次のソースシェルターとして返している。よって、これは成り立つ。(証明終)

例えば、図 3.2 では、ソースシェルターとなりうるシェルターは S1 と S3 である。

3.3.2　メソッド定義の再定義に関する性質

メソッド探索の定義から導き出せるメソッド定義の上書きに関する性質について述べていく。

まず、性質について述べる前に、いくつかの記法について定義しておく。この節では $\alpha, \beta, \gamma, \omega$ をシェルターを表すメタ変数とし、m をメソッドを表すメタ変数とする。

定義 3.3.1. シェルター α で定義されているメソッド m のメソッド定義を m^{α} と表す

定義 3.3.2. メソッド定義 m^0 がメソッド定義 m^1 を再定義していることを $m^0 \triangleright m^1$ と表す

メソッド定義の再定義に関する性質として性質 3.3.4 のような2つのことがあげられる。

性質 3.3.4.

- exposedly にインポートしているシェルターのメソッド定義は再定義することができる
- hiddenly にインポートしているシェルターのメソッド定義は再定義することができない

証明. 前述したように、再定義とは新しい定義で古い定義を置き換えることである。つまり言い換えると、$m^0 \triangleright m^1$ であるとき、メソッド m を呼び出すと、m^0 の定義が呼ばれるということを意味する。

今、$\alpha \xrightarrow{e} \beta$、$\alpha \xrightarrow{h} \gamma$ であると仮定する。

まず、hiddenly にインポートしているシェルターのメソッド定義が再定義できないことを示す。シェルター γ とシェルター α にメソッド m の定義があると仮定すると、定義 3.3.1 より m^{α} と m^{γ} のように記述することができる。m^{α} が m^{γ} の定義を再定義できていないことを示せばよい。シェルター α 内でメソッド m の呼び出しが行われたとする。メソッド探索は定理 3.2.1 の順で行われる。hidden 探索により、hiddenly にインポートしているシェルターで定義しているメソッドが先に呼び出されるため、たとえ m^{α} があったとしても、m^{γ} が存在するならば、m^{γ} が呼び出される。そのため、$m^{\alpha} \not\triangleright m^{\gamma}$ となる。よって、hiddenly にインポートしているシェルターのメソッド定義は再定義できない。

次に、exposedly にインポートしているシェルターのメソッド定義は再定義できることを示す。シェルター α とシェルター β にメソッド m の定義があると仮定すると、定義 3.3.1 より m^{α} と m^{β} のように記述することが

できる。m^{α} の定義が m^{β} の定義を再定義していることを示せばよい。シェルター α でメソッド m の呼び出しが行われたとする。メソッド探索は定理 3.2.1 の順で行われる。exposed 探索により、シェルター α が exposedly にインポートしているシェルターのメソッド定義ではなく、シェルター α が定義しているメソッド定義が先に発見され呼び出される。そのため、$m^{\alpha} \triangleright m^{\beta}$ となる。よって、exposedly にインポートしているシェルターのメソッド定義は再定義することができる。(証明終)

性質 3.3.4 は次のように言い換えることもできる。

性質 3.3.5. シェルター $\alpha, \beta_i, \gamma_j$(i=0,1,...,p j=0,1,...,q) について、$\alpha \xrightarrow{e} \beta_i (i = 0, 1, ..., p)$ であり、$\alpha \xrightarrow{h} \gamma (j = 0, 1, ..., q)$ であるとする。このとき、カレントシェルターが α であるならば、次のようなことが言える。

$$m^{\alpha} \triangleright m^{\beta_i} \quad (i = 0, 1, ..., p) \tag{3.14}$$

$$m^{\alpha} \not\triangleright m^{\gamma_j} \quad (j = 0, 1, ..., q) \tag{3.15}$$

3.3.3　メソッド呼び出しに関する性質

メソッド呼び出しに関する性質を述べていく。この節では、$\alpha, \beta, \gamma, \omega$ をシェルターを表すメタ変数とし、m をメソッドを表すメタ変数とする。

性質について述べる前に、記法を定義しておきたい。

定義 3.3.3. メソッド m^0, m^1 が存在するときに、メソッド m の呼び出しが行われたとする。このとき、m^0 が呼び出されるならば、m^0 が m^1 よりも優先的に呼び出されるという言い方をする。また、そのことを $m^0 > m^1$ と記述する。また、優先度が同じであることを $m^0 = m^1$ と記述する。

メソッド呼び出しの性質として、次の 2 つのようなことがあげられる。

性質 3.3.6.

- hiddenly にインポートしているメソッド定義が優先的に呼び出される
- そのメソッド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. メソッド探索の順序は、定理 3.2.1 のようになっていた。hidden 探索が一番最初に行われることより、『hiddenly にインポートしているメソッド定義が優先的に呼び出される』ことは明らかであろう。

次に『そのメソッド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』ことを示していく。性質 3.3.4 より、exposedly にインポー

トしている定義のみ再定義できることを示した。よって、メソッド m^{ω} の定義がメソッド m^{β} の定義で再定義されているとき、$\beta \xrightarrow{e} \omega$ であると言うことができる。

ソースシェルターが α_s のときに、シェルター α の中で、メソッド m を呼び出したと仮定する。メソッド探索は hidden 探索 →exposed 探索の順に行われる。そのため、再定義が優先的に呼び出されることを証明するためには、hidden 探索で探索する中に m^{β} と m^{ω} が存在する場合と exposed 探索で探索する中に m^{β} と m^{ω} が存在する場合のどちらも m^{β} が呼び出されることを示せばよい。

1. hidden 探索で探索される中にメソッド m^{β}, m^{ω} の定義がある場合とは、$\alpha \xrightarrow{h} \beta$ となっている場合である。条件より、$\beta \xrightarrow{e} \omega$ であるため、推移的なインポートの定義 3.1.2 から $\alpha \xrightarrow{h} \omega$ も成り立っていることが分かる。シェルター α 内のメソッド呼び出しによる hidden 探索では、直接 hiddenly にインポートしているシェルターから関数 *mbodyimport* を使って、exposedly なインポートをたどって探索される。そして、発見し次第、その定義が返される。$\beta \xrightarrow{e} \omega$ より、関数 *mbodyimport* は先にシェルター β でされた定義を発見するため、m^{β} の定義が返される。よって、hidden 探索で探索される中に m^{β} と m^{ω} が存在する場合には、再定義である m^{β} が優先されていることが示せた。

2. exposed 探索で探索される中にメソッド m^{β}, m^{ω} の定義がある場合とは、$\alpha_s \xrightarrow{e} \beta$ となっている場合である。条件より、$\beta \xrightarrow{e} \omega$ であるため、推移的なインポートの定義 3.1.2 から $\alpha \xrightarrow{e} \omega$ も成り立っていることが分かる。シェルター α 内のメソッド呼び出しによる exposed 探索では、ソースシェルターから関数 *mbodyimport* を使って、exposedly なインポートをたどって検索し、定義を発見し次第、それを返す。よって hidden 探索の方と同様の考え方により、m^{β} が先に発見されるため、exposed 探索で探索される中に m^{β} と m^{ω} が存在する場合には、再定義である m^{β} が優先されている。

以上より、再定義が優先されることも示せた。(証明終)

性質 3.3.6 は次の 2 つの性質として言い換えることもできる。

性質 3.3.7. (hidden 優先) シェルター $\alpha, \beta_i, \gamma_j (i = 0, 1, ..., p \;\; j = 0, 1, ..., q)$ が存在するとする。このとき、シェルター α_s をソースシェルターとして、シェルター α 内でメソッド呼び出しをすると、次のようなことが成り立つ。

$$\begin{array}{l} \alpha_s \xrightarrow{e} \beta_i (i = 0, 1, ..., p) \\ \alpha \xrightarrow{h} \gamma_j (j = 0, 1, ..., q) \end{array} \quad \Longrightarrow \quad \begin{array}{l} m^{\gamma_j} > m^{\beta_i} \\ m^{\gamma_j} > m^{\alpha} \end{array} \tag{3.16}$$

性質 3.3.8. (再定義優先) シェルター $\alpha, \beta_0, \beta_1, \gamma_0, \gamma_1, \omega_0, \omega_1$ が存在し、$\alpha \xrightarrow{e} \beta_i, \alpha \xrightarrow{h} \gamma_j, \omega_k \xrightarrow{e} \alpha (i = 0, 1 \ \ j = 0, 1 \ \ k = 0, 1)$ であるとする。このとき、シェルター α でメソッド呼び出しが行われるならば次のようなことが成り立つ。

$$\beta_0 \xrightarrow{e} \beta_1 \implies m^{\beta_0} > m^{\beta_1} \tag{3.17}$$

$$\gamma_0 \xrightarrow{e} \gamma_1 \implies m^{\gamma_0} > m^{\gamma_1} \tag{3.18}$$

$$\omega_0 \xrightarrow{e} \omega_1 \implies m^{\omega_0} > m^{\omega_1} \tag{3.19}$$

3.2.2 章で述べたエラーに関することを次のような性質として、定義した記号を用いて書き表すことができる。

性質 3.3.9. シェルター $\alpha, \beta_0, \beta_1, \gamma_0, \gamma_1$ とメソッド $m^{\beta_i}, n^{\gamma_j} (i = 0, 1 \ \ j = 0, 1)$ が存在するとする。α をカレントシェルターとすると、次のことが成り立つ。

$$\begin{cases} \alpha \xrightarrow{e} \beta_i (i = 0, 1) \\ \beta_0 \not\longmapsto \beta_1, \beta_1 \not\longmapsto \beta_0 \end{cases} \implies \begin{array}{l} m^{\beta_0} = m^{\beta_1} \\ m^{\alpha} \text{が存在しないならばエラーとなる} \end{array} \tag{3.20}$$

$$\begin{cases} \alpha \xrightarrow{h} \gamma_j (j = 0, 1) \\ \gamma_0 \not\longmapsto \gamma_1, \gamma_1 \not\longmapsto \gamma_0 \end{cases} \implies \begin{array}{l} n^{\gamma_0} = n^{\gamma_1} \\ \text{エラーとなる} \end{array} \tag{3.21}$$

また、定義としては、同じメソッド定義であるが、エラーを出すという場合もある。それは、同じメソッド定義を別のシェルターを通して推移的にインポートしている場合である。このことを次のような性質で表すことができ、証明もすることができる。

性質 3.3.10. シェルター $\alpha, \beta_0, \beta_1, \gamma_0, \gamma_1, \omega$ とメソッド m^{ω} が存在するとする。α をカレントシェルターとすると、次のことが成り立つ。

$$\begin{cases} \alpha \xrightarrow{e} \beta_i (i = 0, 1) \\ \beta_0 \not\longmapsto \beta_1, \beta_1 \not\longmapsto \beta_0 \\ \beta_i \xrightarrow{e} \omega (i = 0, 1) \end{cases} \implies m^{\alpha} \text{が存在しないならばエラーとなる} \tag{3.22}$$

$$\begin{cases} \alpha \xrightarrow{h} \gamma_j (j = 0, 1) \\ \gamma_0 \not\longmapsto \gamma_1, \gamma_1 \not\longmapsto \gamma_0 \\ \gamma_j \xrightarrow{e} \omega (j = 0, 1) \end{cases} \implies \text{エラーとなる} \tag{3.23}$$

証明.　性質 3.3.10 の式 (3.23) の証明

$\alpha \xrightarrow{h} \gamma_j$ より、hidden 探索について考える。シェルター α が直接 hiddenly にインポートしているシェルターをシェルター h_0, h_1 とし、$h_0 \ e \ \gamma_0, h_1 \xrightarrow{e} \gamma_1$ であるとする。このとき、hidden 探索では、

$mbodyimport(m, C, h_i), mbodyimport(m, C, h_j)$ と実行される。前章で述べたように、関数 *mbodyimport* は、与えられたシェルターが exposedly にインポートしているシェルターをたどってメソッド探索を行う関数である。推移的なインポートの性質より、$h_0 \xrightarrow{e} \omega, h_1 \xrightarrow{e} \omega$ であるので、
$mbodyimport(m, C, h_i), mbodyimport(m, C, h_j)$ の両式でそれぞれ m^ω が見つかってしまう。そのため、見つかるのは同じメソッド定義であるが、2つ見つかっているのと同意義となるため、エラーとなる。

性質 3.3.10 の式 (3.22) の証明

式 (3.23) の証明と同じように進めていく。exposed 探索について考える。シェルター α が直接 exposedly にインポートしているシェルターをシェルター e_0, e_1 とし、$e_0 \xrightarrow{e} \beta_0, e_1 \xrightarrow{e} \beta_1$ とする。推移的なインポートより、$e_0 \xrightarrow{e} \omega, e_1 \xrightarrow{e} \omega$ となる。このとき、exposed 探索はソースシェルターを α_s とすると、$mbodyimport(m, C, \alpha_s)$ と実行され、関数 *mbodyimport* を再帰的に呼び出していく。探索していき、$mbodyimport(m, C, \alpha)$ の呼び出しに至ったとき、m^α が存在するならば、それが返されるが、存在しないならば $mbodyimport(m, C, e_0)$, $mbodyimport(m, C, e_1)$ が実行される。関数 *mbodyimport* の性質より、両式でそれぞれ m^ω が見つかる。見つかるのは同じメソッド定義であるが、2つ見つかっているのと同意義となるため、エラーとなる。

3.3.4　hiddenly なインポートに関する性質

以上の性質や定義より、hiddenly なインポートに関して次のような2つの性質を導き出すことができる。

性質 3.3.11. **(i)** hiddenly にインポートしているシェルターの revise の外への影響は、hiddenly にインポートしているシェルターのみ

(ii) hiddenly にインポートされているシェルターの中に入ると、外の revise の影響を受けなくなる

証明. **(i) の証明**　シェルター $\alpha, \beta, \gamma, \gamma, \omega$ が存在するとし、$\omega \xrightarrow{e} \alpha, \alpha \xrightarrow{e} \beta, \alpha \xrightarrow{h} \gamma$ であるとする。(i) を証明するには、γ の中のメソッド定義を β と ω からは感知することができないことを言えればよい。3.1.1 より、次のようなことが言える。

$$\omega \xrightarrow{e} \alpha, \alpha \xrightarrow{h} \gamma \implies \omega \not\longrightarrow \gamma \tag{3.24}$$

$$\alpha \xrightarrow{e} \beta, \alpha \xrightarrow{h} \gamma \implies \beta \not\longrightarrow \gamma \tag{3.25}$$

よって、ω,β と γ との間には、インポート関係がないため、γ の中の revise は直接 hiddenly にインポートしているシェルター以外には、影響を及ぼさない。

直接 hiddenly にインポートしているシェルターには影響を及ぼすことは、性質 3.3.7 から言うことができる。

(ii) の証明 (i) の証明と同様に、シェルター $\alpha,\beta,\gamma,\omega$ とインポート関係を定める。シェルター α を通って、シェルター γ でメソッド呼び出しをしたときに、α,β,ω のメソッド定義を呼び出さないことを示せばよい。シェルター γ の現在のソースシェルターを γ_s とする。すると、シェルター γ に至るまでにシェルター α を通っているため、γ_s は α に直接 hiddenly にインポートされているシェルターとなり、$\alpha\ h\ \gamma_s$ となる。

(i) と同様の考え方で、$\beta \not\longrightarrow \gamma, \omega \not\longrightarrow \gamma$ が導ける。そのため、β,ω 内のメソッド定義は β 内でのメソッド呼び出しでは呼び出されない。後は、α 内のメソッド定義が γ 内のメソッド呼び出しで呼び出されないことを示す。

メソッド探索は、定理 3.2.1 のような順序で行われる。そのため、ソースシェルター β_s の子であるシェルター α 内は探索されない。よって、γ 内でメソッド呼び出しが行われたとき、α 内のメソッド定義は呼び出されない。

以上より、hiddenly にインポートされたシェルターの中に入ると、外の revise の影響を受けなくなる。

第4章　実装方法の提案

Simplified Method Shelters には、第3章で示したようなメソッド探索のセマンティクスが存在する。しかし、これをそのまま実装するのはやや非効率である。

例えば Java に実装することを考える。Simplified Method Shelters のメソッド探索の定義では、入力としてソースシェルターを必要とする。このソースシェルターがどのシェルターであるかはカレントシェルターから一意に決定することができず、実行時に動的な情報として与えられる。そのため、Java に導入する場合、Java の通常のメソッドディスパッチに加えて、ソースシェルターを必要とする Simplified Method Shelters のメソッドディスパッチも行わなければならない。Simplified Method Shelters のセマンティクスそのままの動作では動的にメソッドディスパッチを行うため、通常の Java のメソッドディスパッチよりも遅くなってしまう。

この章では、VM に手を加えることなく、通常のメソッドディスパッチと同じコストでメソッドディスパッチが行えるような Simplified Method Shelters の実装方法を提案していく。本研究では、二種類の実装方法を提案するが、どちらもグローバルな定義については考慮していない。simplified method shelters では、グローバルな探索のあとに継承関係の探索を行うため、継承関係も考慮には入れていない。

4.1　実行時探索を減らす工夫

Simplified Method Shelters では、メソッド探索の際、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つを必要とした。このうち、ソースシェルターはカレントシェルターの情報だけでは取得することができず、前に行ったメソッド探索に依存する。そこで、カレントシェルターからソースシェルターが一意に決められるようになれば、Simplified Method Shelters における動的な情報がほぼなくなるため、通常のメソッドディスパッチとそれほど変わらないコストでメソッド探索が行えるようになるはずである。

ソースシェルターがカレントシェルターの情報だけでは取得できない場合とは、図4.1のように複数の子を持つシェルターが存在する場合である。

図 4.1 のシェルター D はシェルター B と C という 2 つのシェルターを子として持つ。このとき、シェルター D のソースシェルターとして考えられるシェルターはシェルター A とシェルター C の 2 通り存在する。このどちらがソースシェルターとなるかは、その前のメソッド探索、つまりそこに至るまでにどのシェルターを通ってきたかに依存している。どのシェルターを通ってきたかという道筋のことをパスと呼ぶことにする。

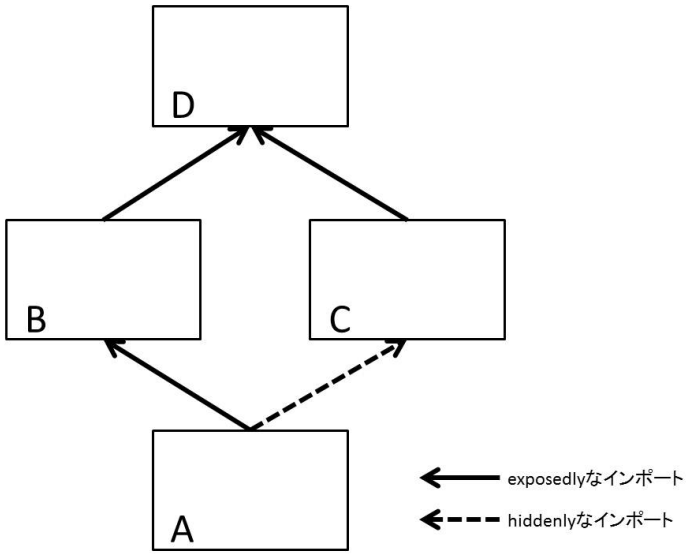

図 4.1: 複数の子を持つシェルター

現在の method shelter dag の表現方法では、図 4.1 のようにソースシェルターを一意に決定できない。そこで method shelter tree という表現方法を次節で紹介する。method shelter tree を導入することによって、カレントシェルターから一意にソースシェルターを決定できるようにする。

4.1.1　method shelter tree の導入

method shelter dag のように、シェルターのインポート条件をそのままグラフにしてしまうと、子を複数持つようなシェルターが存在するグラフとなってしまう。そこでこの節では、子を複数持つようなシェルターが存在したとしても、ソースシェルターが一意に決定できるような木構造のグラフ method shelter tree を提案する。

method shelter tree を定義するために、いくつかの関数を用意する。

まず、関数 *shelters*(dag) と関数 *allPath*(root, dag) を用意する。関数 *shelters*(dag) は dag 内にあるすべてのシェルターの集合を返す。関数 *allPath*(root, dag) は引数としてシェルター root と method shleter dag dag

を与えられると、その root からたどれるすべてのシェルターへのすべてのパスの集合を返す。例えば、図 4.1 の method shelter dag を exDAG と名付けると、

$$\mathit{shelters}(\mathtt{exDAG}) = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\} \tag{4.1}$$

$$\mathit{allPath}(\mathtt{A}, \mathtt{exDAG}) = \{\mathtt{A}, \mathtt{AB}, \mathtt{AC}, \mathtt{ABD}, \mathtt{ACD}\} \tag{4.2}$$

となる。

このような関数を使って、式 (4.3) のような処理を行う。

$$\begin{array}{l}\mathtt{P} = \mathit{allPath}(\mathtt{root}, \mathtt{dag}), \mathtt{S} = \mathit{shelters}(\mathtt{dag}) \\ \forall \mathtt{p} \in \mathtt{P}, \exists \alpha \in \mathtt{S}\ s.t\ \mathtt{p}\alpha \in \mathtt{P}\end{array} \quad \Longrightarrow \quad (\mathtt{p} \longrightarrow \mathtt{p}\alpha) \in \mathtt{E} \tag{4.3}$$

F = (S, E) となるような集合 F を定義する。S と E、root は式 (4.3) で定めた集合および変数とする。すると、F は S をノード集合、E を辺集合とし、root を根とする有向木となる。この F のことを method shelter tree と呼ぶ。$(\mathtt{p} \longrightarrow \mathtt{p}\alpha) \in \mathtt{E}$ より、$\mathtt{p}\alpha$ の親は p のみである。そのため、method shelter tree は各ノードが親を高々1 つ持つような木となる。

例えば、図 4.1 の method shelter dag を式 (4.3) にそって処理を行うと、図のような method shelter tree が得られる。

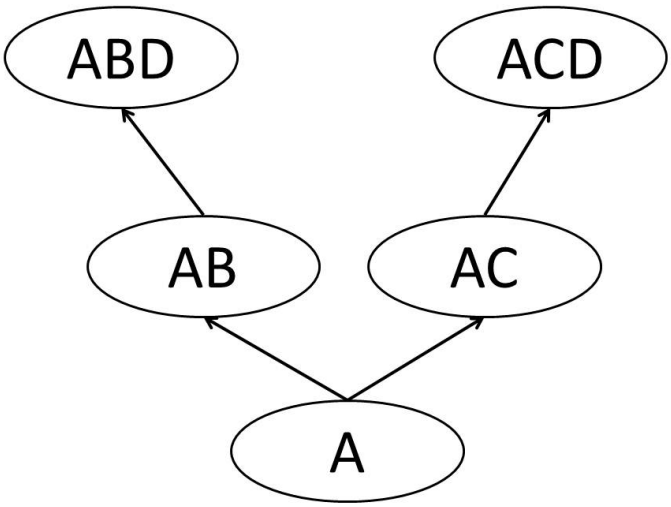

図 4.2: 図 4.1 の method shelter tree

各ノード $\mathtt{p}\alpha$ はパス p を通ったシェルター α を表している。

各ノードがシェルターの情報だけでなく、パスの情報を持っているため、各ノードのソースシェルターは一つに定めることができる。このことは、各ノードの親が 1 つに定まっていることからも言うことができるだろう。

4.2　実装方法 1

method shelter tree の各ノードからソースシェルターは 1 つに定めることができた。この章では、これを利用してよりよい実装方法を探求していく。

この章では、α, β をシェルターを表すメタ変数、$\mathtt{m}, \mathtt{n}$ をメソッド名を表すメタ変数、$\mathtt{p}$ をパスを表すメタ変数、$\mathtt{node}$ をノードを表すメタ変数とする。

4.2.1　定義

各ノードのソースシェルターは一つに定まる。引数としてノードをとり、ソースシェルターを返す関数 *src*(node) を図 4.3 のように定義する。関数 *hidden-imported*?(α, α') を α が α' を直接 hiddenly にインポートしているならば $\mathtt{true}$ を返し、直接 hiddenly にインポートしていないならば $\mathtt{false}$ を返す関数とする。

$$\frac{\mathtt{node} = \alpha}{\mathit{src}(\mathtt{node}) = \alpha} \tag{4.4}$$

$$\frac{\mathtt{node} = \mathtt{p}\alpha'\alpha \qquad \mathit{hidden\text{-}imported}?(\alpha, \alpha') = \mathtt{true}}{\mathit{src}(\mathtt{node}) = \alpha} \tag{4.5}$$

$$\frac{\mathtt{node} = \mathtt{p}\alpha'\alpha \qquad \mathit{hidden\text{-}imported}?(\alpha, \alpha') = \mathtt{false}}{\mathit{src}(\mathtt{node}) = \mathit{src}(\mathtt{p})} \tag{4.6}$$

図 4.3: 関数 *src* の定義

ソースシェルターが一つに決まるということは、exposed 探索でどのシェルターを探索するかが一意に決まるということである。メソッド探索は hidden 探索 →exposed 探索の順に行う。hidden 探索でどのシェルターを探索するかは、hiddenly にインポートしているシェルターが分かれば判明するため、ノードから決定できる。このように、hidden 探索および exposed 探索どちらもどのシェルターを探索するかは、ノードから決定できるため、そのノードから呼び出すメソッドの候補をコンパイル時にひとつに決めることができる。そこでノード $\mathtt{node}$ から呼び出すメソッド $\mathtt{m}$ のメソッ

ド名を $\mathtt{m}_{\mathtt{node}}$ として新たなメソッドを生成する。つまり、式 (4.7) のようなメソッドの生成を行う。

$$\frac{\begin{array}{c}\forall \mathtt{node} \in \mathit{allPath}(\mathtt{root}, \mathtt{dag}),\ \ \mathtt{node} = \mathtt{p}\alpha \\ \alpha_{\mathtt{s}} = \mathit{src}(\mathtt{node}) \\ \mathit{mbody}(\mathtt{m}, \mathtt{C}, \alpha, \alpha_{\mathtt{s}}) = \overline{\mathtt{x}}.\mathtt{body}\ \mathtt{in}\ \beta(\beta_{\mathtt{s}})\end{array}}{\begin{array}{c}\beta \vdash \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\} \\ \Longrightarrow \mathtt{B}\ \mathtt{m}_{\mathtt{node}}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\}\end{array}} \tag{4.7}$$

このように名前を変えたメソッド名を生成する。$\mathit{mbody}(\mathtt{m}, \mathtt{C}, \alpha, \alpha_{\mathtt{s}})$ で呼び出されるメソッド定義はただ 1 つである。(条件に合致するメソッドが 2 つ以上ある場合には、エラーが返されることはすでに述べたとおりである。) そのため、メソッド名 $\mathtt{m}_{\mathtt{node}}$ のメソッド定義もただ 1 つとなる。定義が 1 つであるため、このメソッドを呼び出す際、Simplified Method Shelters で必要とされていた動的な探索は必要となくなる。

このメソッドを呼び出すようにメソッド呼び出し部分を書き換えればよい。メソッド呼び出しを書き換えるために、いくつか関数を定義する。

まず関数 $\mathit{mname}(\mathtt{m}, \mathtt{node})$ を用意する。この関数はメソッド名 $\mathtt{m}$ とノード $\mathtt{node}$ が与えられると、式 4.7 で定めたメソッド名 $\mathtt{m}_{\mathtt{node}}$ を返す関数である。さらに関数 $\mathit{getnode}(\alpha, \alpha_{\mathtt{s}})$ を用意する。この関数は式 (4.8) のように定義する。

$$\frac{\mathtt{node} = \mathtt{p}\alpha,\ \ \mathit{src}(\mathtt{node}) = \alpha_{\mathtt{s}}}{\mathit{getnode}(\alpha, \alpha_{\mathtt{s}}) = \mathtt{node}} \tag{4.8}$$

これらの定義した関数を用いて、メソッド呼び出しを図 4.4 のような手順で変換する。

$$\frac{\begin{array}{c}\mathtt{node} = \mathit{getnode}(\alpha, \alpha_{\mathtt{s}}),\ \ \mathtt{m}' = \mathit{mname}(\mathtt{m}, \mathtt{node}) \\ \mathtt{node} \vdash \mathtt{B}\ \mathtt{n}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\}\end{array}}{\mathtt{body}[\mathtt{m} \longmapsto \mathtt{m}']} \tag{4.9}$$

図 4.4: 実装方法 1 によるメソッド呼び出しの書き換え

前述したとおり、各ノードから呼び出すメソッドは一つに決まるので、関数 *mname* から導ける $\mathtt{m}'$ の定義も一つに決まる。そのため、$\mathit{mbody}(\mathtt{m}, \mathtt{C}, \alpha, \alpha_{\mathtt{s}})$ のうち、ソースシェルターやカレントシェルターの情報は必要なくなり、メソッド名とクラスだけでメソッド探索が行えるようになる。

4.2.2　正当性の議論

この節では、実装方法 1 が本当に Simplified Method Shelters のメソッド探索の定義と合致しているのかを検証する。

図 4.5 のような例を考える。この例では、*main* メソッドを持つシェルター α とメソッド $\mathtt{m}$ を持つシェルター β が存在する。メソッド *main* の中では、メソッド $\mathtt{m}$ の呼び出しがされている。シェルター名のあとのかっこの中身は、現在のソースシェルターであるとする。つまり、この図はソースシェルターが $\alpha_\mathtt{s}$ であるシェルター α の中の *main* メソッドでメソッド $\mathtt{m}$ を呼び出すと、ソースシェルターを $\beta_\mathtt{s}$ とするシェルター β の中のメソッド $\mathtt{m}$ の定義が呼び出されることを意味する。このことは式 (4.10) のように書き表せる。

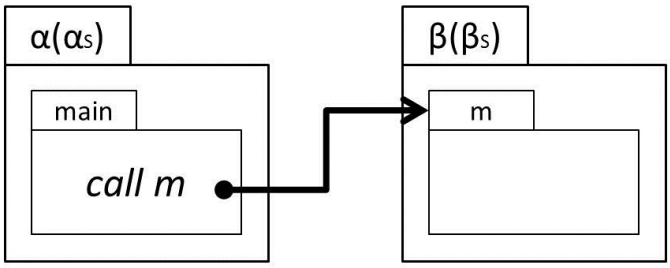

図 4.5: 正当性の議論のためのメソッド呼び出しの例

$$mbody(\mathtt{m}, \mathtt{C}, \alpha, \alpha_\mathtt{s}) = \overline{\mathtt{x}_\mathtt{m}}.\mathtt{body}_\mathtt{m}\ \mathtt{in}\ \beta(\beta_\mathtt{s}) \tag{4.10}$$

実装方法 1 の方法で、メソッド呼び出しを書き換えたときにも、同じ定義が呼び出されることを示す。

正当性　初期値はカレントシェルターが α、ソースシェルターが $\alpha_\mathtt{s}$ である。

すでに method shelter tree は method shelter dag から作られているものと仮定する。今回、使用する method shelter tree を `tree` とする。

実装方法 1 では、シェルターではなくパスの情報を持った method shelter tree のノードを中心に扱う。カレントシェルターとソースシェルターから現在ノードを求める式は式 (4.8) である。現在のノードを $\mathtt{node}_\alpha$ とすると、式 (4.11) のようになる。

$$\mathtt{node}_\alpha = getnode(\alpha, \alpha_\mathtt{s}) \tag{4.11}$$

$\mathtt{node}_\alpha$ の中のメソッド呼び出しは全て図 (4.4) のような規則で書き換えられる。どのメソッド名を呼び出すように書き換えられるかは、関数 *mname* によって決定されている。$\mathit{mname}(\mathtt{m}, \mathtt{node}_\alpha) = \mathtt{m}_{\mathtt{node}_\alpha}$ となるため、$\mathtt{node}_\alpha$ の中のメソッド m のメソッド呼び出しは、図 4.6 のようにメソッド $\mathtt{m}_{\mathtt{node}_\alpha}$ の呼び出しとなる。つまり、式 (4.10) より、メソッド名 $\mathtt{m}_{\mathtt{node}_\alpha}$ のメソッドの定義がシェルター β の中のメソッド m の定義と同じであることを示せばよい。

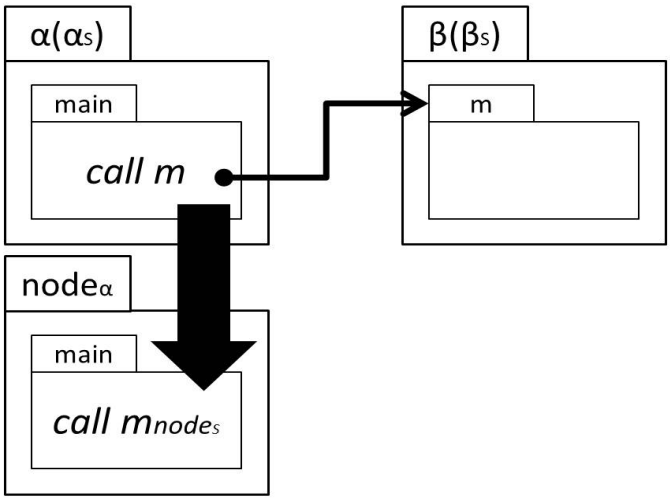

図 4.6: 図 4.5 のメソッド呼び出し部分の変換

メソッド名 $\mathtt{m}_{\mathtt{node}_\alpha}$ のメソッドを作りだす規則は式 (4.7) である。この式からメソッド $\mathtt{m}_{\mathtt{node}_\alpha}$ の定義とシェルター β 内のメソッド m の定義が同じであることを示していく。

(i) シェルター α は tree 内のノードであるため、

$$\mathtt{node}_\alpha \in \mathit{allPath}(\mathtt{tree}) \tag{4.12}$$

である。

(ii) 式 (4.8) と式 (4.11) より、次のことが言える。

$$\mathtt{node}_\alpha = \mathtt{p}\alpha(\mathtt{p} \in \mathit{allPath}(\mathtt{tree}) \cap \phi) \tag{4.13}$$

$$\mathit{src}(\mathit{node}_\alpha) = \alpha_\mathtt{s} \tag{4.14}$$

(iii) 式 (4.10) より、

$$\mathit{mbody}(\mathtt{m}, \mathtt{C}, \alpha, \alpha_\mathtt{s}) = \overline{\mathtt{x}_\mathtt{m}}.\mathtt{body}_\mathtt{m}\ \mathtt{in}\ \beta(\beta_\mathtt{s}) \tag{4.15}$$

以上の (i) から (iii) を $\mathtt{n} = \mathtt{node}_\alpha$ として、図 4.4 のメソッドの生成の式に代入していく。(i)、(ii) より $\mathtt{node}_\alpha = \mathtt{p}\alpha(\mathtt{p} \in \mathit{allPath}(\mathit{tree}) \cap \phi), \alpha_s = \mathit{src}(\mathtt{node}_\alpha)$ となる。そのため、(iii) より $\overline{\mathtt{x}} = \overline{\mathtt{x_m}}, \mathtt{body} = \mathtt{body_m}, \beta = \beta, \beta_\mathtt{S} = \beta_\mathtt{s}$ となる。以上より、図 4.4 の式は式 (4.16) のようになる。

$$\frac{\begin{array}{c}\mathtt{node}_\alpha \in \mathit{allPath}(\mathtt{tree})\ \ \mathtt{node}_\alpha = \mathtt{p}\alpha(\mathtt{p} \in \mathit{allPath}(\mathtt{tree}) \cap \phi) \\ \alpha_\mathtt{s} = \mathit{src}(\mathtt{node}_\alpha) \\ \mathit{mbody}(\mathtt{m}, \mathtt{C}, \alpha, \alpha_\mathtt{s}) = \overline{\mathtt{x_m}}.\mathtt{body_m}\ \mathtt{in}\ \beta(\beta_\mathtt{s})\end{array}}{\begin{array}{c}\beta \vdash \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x_m}})\{\mathtt{body_m}\} \\ \Longrightarrow \mathtt{B}\ \mathtt{m}_{\mathtt{node}_\alpha}(\overline{\mathtt{B}}\ \overline{\mathtt{x_m}})\{\mathtt{body_m}\}\end{array}} \tag{4.16}$$

よって、$\mathtt{m}_{\mathtt{node}_\alpha}$ の定義が、ソースシェルターを $\beta_\mathtt{s}$ とするシェルター β の中のメソッド $\mathtt{m}$ の定義と同じであることを示すことができた。

4.3　実装方法 2

4.2 章の実装方法だと、メソッド名の違う同じ定義が 2 つ以上生成されてしまう場合がある。例えば、図 4.7 を見て欲しい。シェルター $\mathtt{A} \sim \mathtt{E}$ があり、シェルター $\mathtt{C}$ でメソッド $\mathtt{m}$ が定義されている。このとき、実装方法 1 の方法では、シェルター $\mathtt{C}$ から呼び出すメソッド $\mathtt{m_C}$、シェルター $\mathtt{E}$ から呼び出すメソッド $\mathtt{m_E}$ が生成される。しかし、$\mathtt{m_C}$ と $\mathtt{m_E}$ はどちらも $\mathtt{C}$ で定義されたメソッド $\mathtt{m}$ のメソッド名を変えただけのものである。この章では、このような同じメソッド定義が名前を変えて複数生成されてしまうような重複をなくす実装方法 2 を提案する。

実装方法 1 と同様に、最終的な目標として動的な情報に左右されないような形にメソッド呼び出しを書き換えることを目標とする。

4.3.1　定義

関数 $\mathit{allPath}(\mathtt{root}, \mathtt{dag})$、$\mathit{src}(\mathtt{node})$、$\mathit{getnode}(\alpha, \alpha_\mathtt{s})$ は実装方法 1 と同じ定義であるとする。新たに関数 $\mathit{allsrc}(\mathtt{tree})$ を用意する。この関数は method shelter tree が与えられると、ソースシェルターになりうるシェルターの集合を返す。性質 3.3.3 より、ソースシェルターになりうるシェルターとは、直接 hiddenly にインポートされているシェルター、または初期値のみである。例えば、図 4.7 の場合では、$\mathit{allsrc}(\mathtt{tree}) = \{\mathtt{A}, \mathtt{C}\}$ となる。

hidden 探索というのは、直接 hiddenly にインポートしているシェルターから関数 *mbodyimport* で探索することである。exposed 探索とは、ソースシェルターから関数 *mbodyimport* で探索することである。性質 3.3.3 より、

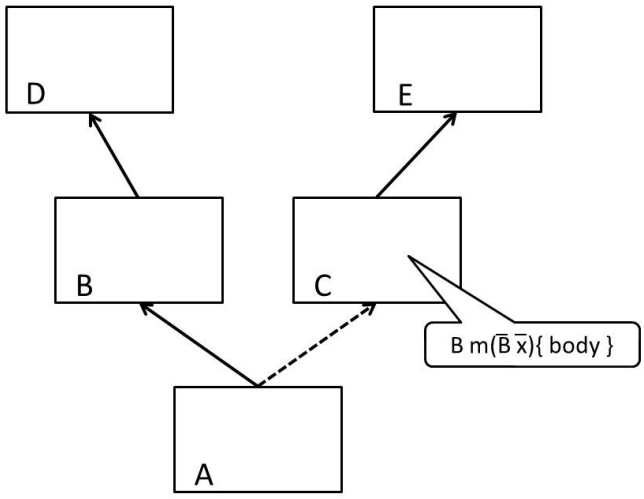

図 4.7: コードが重複してしまう例

直接 hiddenly にインポートしているシェルターはソースシェルターになりうるシェルターである。つまり、hidden 探索、exposed 探索、どちらもソースシェルターになりうるシェルターから関数 *mbodyimport* で探索を行なっているにすぎない。

そこで、メソッド名の変換を式 (4.17) のような形で行うことにする。

$$\frac{\begin{array}{c}\forall \alpha \in \mathit{allsrc}(\mathtt{tree}) \\ \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \alpha) = \overline{\mathtt{x}}.\mathtt{body}\ \mathtt{in}\ \beta\end{array}}{\begin{array}{c}\beta \vdash \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\} \\ \Longrightarrow \mathtt{B}\ \mathtt{m}_{\alpha}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\}\end{array}} \tag{4.17}$$

mbodyimport(m, C, α) でメソッド定義が返される場合、2 つ以上合致されるメソッド定義があった場合には、エラーが出力されることはすでに述べた。関数 *mbodyimport*(m, C, α) でメソッド定義が返される場合、そのメソッド定義はただ 1 つである。つまり、メソッド名 $\mathtt{m}_{\alpha}$ のメソッド定義が 1 つのみであると言うことができる。実装方法 1 と同様に、定義が 1 つであるため、このメソッドを呼び出す際、Simplified Method Shelters で必要とされていた動的な探索は必要となくなる。そのため、メソッド呼び出し部分をこのメソッドを呼び出すように書き換えればよい。

メソッド呼び出しを書き換えるために、関数 *mname*(m, node) に変更を加える。hidden 側に定義があった場合にはその定義のメソッド名を、なかった場合には exposed 側で定義されている名前を返すように変更する。

関数 *mname* は図 4.18 のようになる。

$$\frac{\begin{array}{c}\mathtt{node} = \mathtt{p}\alpha \\ \overline{\mathtt{h}} \in \mathit{hiddenly\text{-}importings}(\alpha) \\ \exists \mathtt{h_i} \forall \mathtt{h_j} (\mathtt{i} \neq \mathtt{j}) \quad \mathtt{m_{h_i}}\ \mathit{is\ defined}\ \&\ \mathtt{m_{h_j}}\ \mathit{is\ undefined}\end{array}}{\mathit{mname}(\mathtt{m}, \mathtt{node}) = \mathtt{m_{h_i}}} \tag{4.18}$$

$$\frac{\begin{array}{c}\mathtt{node} = \mathtt{p}\alpha \\ \overline{\mathtt{h}} \in \mathit{hiddenly\text{-}importings}(\alpha) \\ \mathtt{snode} = \mathit{src}(\mathtt{node}) \\ \forall \mathtt{h_i} \in \overline{\mathtt{h}} \quad \mathtt{m_{h_i}}\ \mathit{is\ undefined}\end{array}}{\mathit{mname}(\mathtt{m}, \mathtt{node}) = \mathtt{m_{snode}}} \tag{4.19}$$

図 4.8: 実装方法 2 の関数 *mname* の定義

これらの関数を使って、図 4.9 のような手順でメソッド呼び出しを変換する。

$$\frac{\begin{array}{c}\mathtt{node} = \mathit{getnode}(\alpha, \alpha_\mathtt{s}), \quad \mathtt{m}' = \mathit{mname}(\mathtt{m}, \mathtt{node}) \\ \alpha \vdash \mathtt{B}\ \mathtt{n}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\}\end{array}}{\mathtt{body}[\mathtt{m} \longmapsto \mathtt{m}']} \tag{4.20}$$

図 4.9: 実装方法 2 によるメソッド呼び出しの書き換え

メソッド $\mathtt{m}'$ の定義は前述したとおり、ただ一つである。そのため、メソッド探索の際にソースシェルターやカレントシェルターの情報が必要なくなり、メソッド名とクラスだけでメソッド探索が行えるようになる。

4.3.2　正当性の議論

この節では、実装方法 2 が Simplified Method Shelters の定義を満たしているかを示す。

今回の議論では、hidden 探索でメソッドが見つかる場合と exposed 探索でメソッドが見つかる場合に分けて考える。まず、図 3.3 の式 (3.7) と式 (3.8) の定義を関数 *hiddenly-importings* を使った形に書き換える。書き換えると、図 4.10 のような式になる。

式 (4.21) は hidden 探索でメソッド定義が見つかった場合の関数 *mbody* を表し、式 (4.22) は exposed 探索で見つかった場合の関数 *mbody* を表して

$$\frac{\begin{array}{cc} & \overline{\mathtt{h}} = \mathit{hiddenly\text{-}importings}(\alpha) \\ \exists \mathtt{h_i} & \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \mathtt{h_i}) = \overline{\mathtt{x}}.\mathtt{body\ in}\ \beta \\ \forall \mathtt{h_j}(\mathtt{i} \neq \mathtt{j}) & \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \mathtt{h_j})\ \ \mathit{undefined} \end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \alpha, \alpha_\mathtt{s}) = \overline{\mathtt{x}}.\mathtt{body\ in}\ \beta(\mathtt{h_i})} \tag{4.21}$$

$$\frac{\begin{array}{c} \overline{\mathtt{h}} = \mathit{hiddenly\text{-}importings}(\alpha) \\ \forall \mathtt{h_i} \quad \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \mathtt{h_i})\ \ \mathit{undefined} \\ \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \alpha_\mathtt{s}) = \overline{\mathtt{x}}.\mathtt{body\ in}\ \beta(\alpha_\mathtt{s}) \end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \alpha, \alpha_\mathtt{s}) = \overline{\mathtt{x}}.\mathtt{body\ in}\ \beta(\alpha_\mathtt{s})} \tag{4.22}$$

図 4.10: 関数 mbody の別の書き方

いる。

この書き換えた関数 *mbody* を用いて、議論を進めていく。今回使用する method shelter tree を `tree` とする。ソースシェルターが $\alpha_\mathtt{s}$ であるとき、あるシェルター α からメソッド $\mathtt{m}$ を探索するとして、議論を始める。

hidden 探索　シェルター α が直接 hiddenly にインポートしているシェルターを $\overline{\mathtt{h}}$ とする。hidden 探索でメソッド定義が発見される条件というのは、式 (4.23) である。

$$\begin{array}{ll} \exists \mathtt{h_i} \in \overline{\mathtt{h}} & \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \mathtt{h_i}) = \overline{\mathtt{x}}.\mathtt{body\ in}\ \beta \\ \forall \mathtt{h_j} \in \overline{\mathtt{h}}(\mathtt{i} \neq \mathtt{j}) & \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \mathtt{h_j})\ \ \mathit{undefined} \end{array} \tag{4.23}$$

また、$\overline{\mathtt{h}}$ の要素はシェルター α に直接 hiddenly にインポートされているため、$\overline{\mathtt{h}} \in \mathit{allsrc}(\mathtt{tree})$ である。そのため、式 (4.17) より実装方法 2 の変換で $\forall \mathtt{h_k} \in \overline{\mathtt{h}}$ に対し、式 (4.24) のようなことが成り立っていると言えるだろう。

$$\frac{\begin{array}{c} \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \mathtt{h_k}) = \overline{\mathtt{x}}.\mathtt{body\ in}\ \beta \\ \beta \vdash \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\} \end{array}}{\exists \mathtt{m_{h_k}}\ s.t\ \mathtt{B}\ \mathtt{m_{h_k}}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\}} \tag{4.24}$$

この式より、hidden 探索で発見されるメソッド定義は式 (4.17) で名前を変えて用意されていることが分かる。あとは、図 4.9 のメソッド呼び出しの書き換えで、ソースシェルターが $\alpha_\mathtt{s}$ でシェルターが α のときのノードの中のメソッド $\mathtt{m}$ の呼び出しが、hidden 探索でメソッドが見つかる場合にはメソッド $\mathtt{m_{h_i}}$ の呼び出しに変更されていることを示せばよい。

式 (4.9) では、ソースシェルターが α_s、シェルターが α のとき、α の中のメソッド $\mathtt{m}$ のメソッド呼び出しを関数 *mname* で取得できるメソッド名に書き換えている。そこで、関数 *mname* の定義を見る、関数 *mname* の定義は図 4.8 中の式 (4.18) と式 4.19 である。式 (4.18) では、式 (4.23) の条件のとき、メソッド名 $\mathtt{m}_{\mathtt{h_i}}$ を返すように定義されている。

以上より、hidden 探索で発見される場合、実装方法 2 は定義を満たすように実装されていることが分かる。

exposed 探索　シェルター α が hiddenly にインポートしているシェルターを $\overline{\mathtt{h}}$ とする。exposed 探索でメソッド定義が発見される条件というのは、式 (4.25) のようになる。

$$\begin{aligned} \forall \mathtt{h_i} \in \overline{\mathtt{h}} \quad & \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \mathtt{h_i})\ \ \mathit{undefined} \\ & \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \alpha_S) = \overline{\mathtt{x}}.\mathtt{body}\ \mathtt{in}\ \beta \end{aligned} \tag{4.25}$$

α_s はシェルター α のソースシェルターであるため、$\alpha_s \in \mathit{allsrc}(\mathtt{tree})$ である。そのため、式 (4.17) の変換より、式 (4.26) のようなことが言えるだろう。

$$\frac{\begin{array}{c} \mathit{mbodyimport}(\mathtt{m}, \mathtt{C}, \alpha_s) = \overline{\mathtt{x}}.\mathtt{body}\ \mathtt{in}\ \beta \\ \beta \vdash \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\} \end{array}}{\exists \mathtt{m}_{\alpha_s}\ s.t\ \mathtt{B}\ \mathtt{m}_{\alpha_s}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{body}\}} \tag{4.26}$$

この式より、exposed 探索で発見されるメソッド定義は式 (4.17) で名前を変えて用意されていることが分かる。あとは、hidden 探索のときと同様にメソッド呼び出しの書き換えで、ソースシェルターが α_s、シェルターが α のときのノードの中のメソッド $\mathtt{m}$ の呼び出しが、exposed 探索で発見されるときにはメソッド $\mathtt{m}_{\alpha_s}$ の呼び出しに変更されていることを示せばよい。

hidden 探索のときと同様に、関数 *mname* で取得できるメソッド名を調べる。関数 *mname* の定義は図 4.8 中の式 (4.18) と式 (4.19) である。式 (4.25) の条件のとき、式 (4.19) より関数 *mname* はメソッド名 $\mathtt{m}_{\alpha_s}$ を返す。よって、図 4.9 の規則で変換すると、メソッド $\mathtt{m}$ の呼び出しはメソッド $\mathtt{m}_{\alpha_s}$ の呼び出しに書き換えられていることが分かる。

以上より、exposed 探索で発見される場合も、実装方法 2 は定義を満たす。

第5章　まとめと今後の課題

5.1　まとめ

既存のクラスの拡張を行う際に、プログラム全体に影響が及ぶことを避ける方法として、スコープのコントロールがあげられる。Method Shelters はシェルター内のチェンバーにメソッド定義を格納することで、メソッド定義のスコープをコントロールすることを可能にした。しかし、チェンバーが存在することによって、Method Shelters のセマンティクスは複雑さを増し、難解なものになっていた。

本研究では、そのセマンティクスの複雑さを改善するために、チェンバーを排除した Simplified Method Shelters を提案した。チェンバーをなくすことでより分かりやすいセマンティクスを与え、基本的な性質を明確化することに成功した。また、チェンバーの代替案として、hidden インポートと exposed インポートという 2 種類のインポートを導入した。この 2 種類のインポートを駆使することによって、メソッド定義のスコープをコントロールすることや、メソッド定義を再定義から守ることを可能にした。

さらに、本研究では、Simplified Method Shelters を言語に実装する 2 つの方法を提案した。Simplified Method Shelters を言語に導入する際、セマンティクスそのままの実装では実行時探索を必要としていた。これは、メソッド探索に関して、動的な情報を必要としていたためであった。本研究の提案した実装手法では、method shelter tree の形にすることにより、動的な情報を減らした。さらに、メソッド名を変更したメソッドを呼び出すようにすることによって、メソッド呼び出し時に呼び出すメソッド定義の候補を 1 つに絞ることを可能にした。これによって、実行時間の短縮をはかった。

5.2　今後の課題

5.2.1　継承関係の考慮

本研究の実装方法の提案では、グローバルメソッドおよびクラスの継承関係は考慮していない。実際に実装を行う上では、クラスの継承関係の考慮は重要な議題になってくるだろう。

第 4 章では実装方法を 2 つ提示したが、そのどちらもメソッド名を変更するというものだった。ただし、このままの実装では図 5.1 のような場合に、正しいメソッドを呼ぶことができない。図 5.1 では、SuperClass と SubClass という 2 つのクラスが存在し、SubClass は SuperClass を継承している。また、シェルター S0 の中で、SubClass の setPhrase メソッドを revise している。このとき、第 4 章の実装方法では、どちらの実装方法でもシェルター S0 が revise しているメソッド定義は名前を変えて用意されてしまう。このとき、図 5.1 の 13 行目から 16 行目のように、SuperClass として宣言した変数に SubClass のインスタンスを代入し、setPhrase メソッドを呼び出すことを考える。本来であれば、シェルター S0 で定義したメソッド定義が呼ばれるべきであるが、第 4 章の実装方法のままでは、この部分のメソッド呼び出しは書き換えられないため、“Hello” を出力してしまうであろう。

```
1  class SuperClass{
2    void setPhrase(){
3      System.out.println(‘‘Hello’’);
4    }
5  }
6  class SubClass extends SuperClass{}
7
8  shelter S0{
9    revise SubClass{
10     System.out.println(‘‘Good Afternoon’’);
11   }
12   public static void main(){
13     SuperClass C1;
14     SubClass C2 = new SubClass();
15     C1 = C2;
16     C1.setPhrase();
17   }
18 }
```

図 5.1: 継承関係の例

このような複雑な継承関係にも対応できるようなより細かな実装方法を今後考える必要がある。

5.2.2　super 呼び出し

super 呼び出しでどの定義を呼び出すかは、下のような候補があげられる。

(i) 1 つ前にされたメソッド定義

(ii) simplified method shelter の外で定義されたオリジナルのメソッド定義

(iii) 通常の super 呼び出しと同じ親クラスの定義

(i) を導入するのであれば、hiddenly にインポートしているメソッド定義と exposedly にインポートしているメソッド定義、グローバルな定義など様々な定義の中からどれを呼び出すかについて議論を行う必要がある。(ii) については、simplified method shelter の中で追加されたメソッド定義はどうするのか、といったことをきちんと定める必要がある。

(i) から (iii) のうち、どれを導入すれば勝手がいいのか、実装面も考慮した上で今後考えていかねばならないだろう。

5.2.3　実際の言語に導入

さらなる今後の目標として、実際の言語に実装してみる、ということがあげられる。

2.2.2 章では、実装時の欠点の例として、Java の VM を変更せずに実装する場合をあげた。そして、第 4 章では、VM を変更せずに通常のメソッドディスパッチと同じコストになるような実装を目指した実装手法を提示した。この実装方法で本当に実装できるのか、継承関係・グローバルメソッド・super 呼び出しを考慮した後、Java に Simplified Method Shleters を導入することで試していきたい。その際、実装可能かという観点はもちろん、コストの面についても測定を行い、データ化する必要があると考えている。

参考文献

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

[2] Akai, S. and Chiba, S.: Method Shelters:Avoiding Conflicts among Class Extensions Caused by Local Rebinding, AOSD'12 (2012).

[3] Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Classboxes: Controlling visibility of class extensions, *Computer Languages, Systems and Structures* (2005).

[4] Bergel, A., Ducasse, S. and Nierstrasz, O.: Classbox/J: controlling the scope of change in Java, *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, New York, NY, USA, ACM, pp. 177–189 (2005).

[5] Chiba, S., Igarashi, A. and Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch, *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, New York, NY, USA, ACM, pp. 539–554 (2010).

[6] Clifton, C., Leavens, G. T., Chambers, C. and Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for Java, *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, New York, NY, USA, ACM, pp. 130–145 (2000).

[7] Flatt, M. and Felleisen, M.: Units: cool modules for HOT languages, *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, New York, NY, USA, ACM, pp. 236–248 (1998).

[8] Goldberg, A. and Robson, D.: *Smalltalk-80: the language and its implementation*, Addison-Wesley Longman Publishing Co., Inc. (1983).

[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.: An overview of AspectJ, ECOOP 2001—Object-Oriented Programming, pp. 327–354 (2001).

[10] McDirmid, S., Flatt, M. and Hsieh, W. C.: Jiazzi: new-age components for old-fasioned Java, *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, New York, NY, USA, ACM, pp. 211–222 (2001).

[11] Millstein, T., Reay, M. and Chambers, C.: Relaxed MultiJava: balancing extensibility and modular typechecking, *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, New York, NY, USA, ACM, pp. 224–240 (2003).