東京大学
情報理工学系研究科 創造情報学専攻
修士論文

有効範囲を既知のコールパスに限定することで安全に利用できる Ruby のためのクラス拡張 Method Seals

Method Seals:
Safe class extension in Ruby relying on call-path delimited scope

福室 嶺
Ryo Fukumuro

指導教員　千葉 滋 教授

2016 年 1 月

概要

クラス拡張を安全に利用するためにその有効範囲を既知のコールパス上に限定する言語機構 Method Seals を提案する．クラス拡張は既存のクラスに対してソースコードを書き換えることなく外部からメソッドの追加や再定義を可能にする言語機構であり，Ruby や AspectJ，C# など様々な言語に取り入れられている．クラス拡張はモジュラリティの向上に寄与するが，複数の拡張同士が衝突したり, クラス拡張が想定外の領域で有効になることで誤動作を引き起こしやすい．そこで，Method Seals ではプログラマのコードに対する理解度に応じてクラス拡張の有効範囲を段階的に拡張することができるようにした．プログラマの理解の及ぶ範囲でしかクラス拡張が有効にならないため，クラス拡張が想定外の範囲で有効になり誤動作を引き起こすことは少なくなる．本研究では Ruby 処理系上に Method Seals を実装し，様々な条件下でその実行性能を評価した．

Abstract

We propose method seals, a safe class extension limiting its scope to known call paths. Class extensions allow us to extend an existing class to add or modify a method without rewriting its code, and they are supported by various languages including Ruby, AspectJ, and C#. They are easy to cause unintended behavior due to collision among multiple extensions or being enabled in an unexpected domain while they contribute to improvement modularity. To address such a problem, we propose method seals designed to allow class extensions control their effective range depending on the degree of understanding about a code base of programmers. Method seals regard code in an unknown package as a black box and disable class extensions there. Furthermore, they also temporarily regard callees of such a package as being in a black box. This prevents class extensions from giving an unexpected impact on codes in black boxes and causing an unintended behavior of a whole program. We implement method seals on Ruby interpreter and evaluates its performance under several conditions.

目次

第 1 章

はじめに

現代のソフトウェア開発において，プログラムのモジュラリティを向上させ，コードの再利用性を高めることは半ば常識と化しており，可能な限りソフトウェア内のコードの重複を排し，プログラム中の複数箇所から利用されるコードはモジュールとして切り出すということは当たり前に行われている．また，開発コストの削減あるいはコードの信頼性の向上を目的に，他者によってメンテナンスされているライブラリやフレームワークを用いて開発することも多く，一定以上の規模のソフトウェアを全て一から実装することはまずありえない．通常，ライブラリやフレームワークは，多彩なユースケースに対応できるよう，その挙動をカスタマイズする手段を提供していることが多い．例えば，設定ファイルに応じて振る舞いを変えたり，あるいはインターフェースなどホストとなるプログラミング言語の機能を用いて拡張性を持たせる，などが挙げられる．一般に，よく設計され広く使われているライブラリであれば，非常に高い拡張性を持っており，大抵のユースケースはカバーできるようになっている．しかし，こうした拡張性はあくまで製作者の想像力の働く範囲で提供されており，限界がある．あるいは，あるユースケースについてはメンテナンスコストと需要の兼ね合いでライブラリとしてサポートしないということもあるだろう．複雑な要件を抱えるソフトウェアの開発では，既存のライブラリの想定するユースケースではカバーしきれないことも多い．このような事態に直面した際に取れる選択肢はそれほど多くはなく，例えばライブラリの使用をやめ独自に実装するか，あるいはライブラリを要求を満たすように書き換えることが考えられる．前者は規模によってはコストが掛かり過ぎるし，また後者は，元々のライブラリのアップデートに追従する手間がかかる等，拡張したライブラリをメンテナンスするコストがかかる．

直接ソースコードを書き換えることなく既存のクラスを拡張することが出来るような言語機構が広く利用されている．例えば MultiJava [1, 2] や Ruby [3] で採用されているオープンクラスと呼ばれる仕組みが挙げられる．オープンクラスを採用する言語では，クラスは外部からの変更に対して開いており，メソッドの追加や再定義が容易に行える．Ruby ユーザーの間で特に人気の高い Web フレームワークである Ruby on Rails [4] では Ruby の組み込みクラスさえもオープンクラスを用いて積極的に拡張している．このように既存のプログラムに対して動的にパッチをあてることはゲリラパッチやモンキーパッチなどと呼ばれており，その利便性と危険性は Ruby コミュニティにおいて広く認識されている．

このような言語機構を我々はクラス拡張 (*class extensions*) と呼んでいる．クラス拡張に類する言語機構は Ruby 以外にも，Smalltalk [5] や，Aspect-oriented programming [6]，CLOS [7]，Context-oriented programming [8]，C#[9]，Objective-C [10] 等，様々なプログラミング言語やパラダイムに取り入れられており，広く利用されている．しかしその一方でクラス拡張は意図しない動作を引き起こしやすい．これは拡張されたクラスの振る舞いが，拡張前の振る舞いを前提にしていたコードと互換性があるとは限らないからである．完全な後方互換性を保ってクラス拡張を実装することは常に可能とは限らなく，また，利用範囲の広いクラスへの拡張の場合はクラス拡張がプログラム全体にどのような影響を与えるかを事前に測ることは難しい．

このような問題への対策として，クラス拡張の有効範囲を何らかの方法で制限する仕組みがこれまで数多く提案されてきた．例えば，selector namespaces [11] や，Ruby の refinements，classboxes [12]，method shelters [13]，method shells [14] 等がある．このような手法としてはおおまかに静的にその有効範囲を決定する手法と，動的に決定する手法とが存在する．一般に静的に決定する手法は安全性は高いものの機能性が低いことが多い．例えば，Ruby の refinements ではクラス拡張の有効範囲をレキシカルスコープ下に限定する．これはソースコードの構文構造と，クラス拡張の有効範囲が静的に対応するため，利用者にとってその有効範囲が理解しやすく，誤動作を引き起こす可能性は低い．一方，複数のスコープにまたがってクラス拡張を有効にすることが出来ないため，利用できるシーンが限定される．Context-oriented programming のように動的スコープを用いる手法では，複数のスコープにまたがってクラス拡張を利用することができるため機能性は高い．一方，動的な手法では一定範囲にその有効範囲を絞ることができるものの，その絞った範囲においてはグローバルに拡張が有効になるため，有効範囲を全く制限しないグローバルスコープと同様にクラス拡張の影響範囲を予測することが難しいことがある．また，静的な手法よりもセマンティクスが複雑化しやすい傾向にあり，そもそも利用者にとってその有効範囲の理解が難しいようなものもある．このように，これらの手法は一長一短であり，未だ十分なものになっていないと我々は考えている．

そこで我々は，安全性を損なわない範囲でクラス拡張の有効範囲を制御可能な言語機構 *method seals* を提案する．method seals はプログラマが理解度に応じて段階的にクラス拡張の有効範囲を広げていくことができる．プログラマの理解を超えた範囲で勝手にクラス拡張が有効になることを防げるため，クラス拡張によってプログラムが不意に破壊される危険性を軽減できる．

本論文では，我々の提案する method seals の概要と利用例，そしてその背景となるアイディアについて述べる．我々は method seals のアイディアはクラス拡張に類する言語機構を持つあらゆるプログラミング言語に対して適用可能だと考えているが，今回は特に Ruby へと適用し，実際に Ruby 処理系を拡張することで，method seals を実装した．この拡張処理系の概要と，これを用いて行った実験についても報告する．

以降，2 章ではクラス拡張とその問題点について述べ，何故クラス拡張の有効範囲を制御する必要があるかを論じる．また，同時に関連研究とその問題点を紹介する． 3 章では我々が

提案する method seals の概要を例とともに述べる．また，method seals がどのように安全性を確保するかについて解説する．一方，method seals にはいくつかの制限事項が存在する．これらについても紹介し，また現時点で考えうる対策について述べる． 4 章では我々が実際に Ruby 処理系を拡張して実装した method seals について紹介する．また，実装に関連する Ruby 処理系の内部アーキテクチャについても触れ，更に本実装以前に先駆けて作成したプロトタイプ実装との比較も行う． 5 章では我々が実装した処理系に対して行った性能計測について報告する． 6 章では本論文のまとめと今後の課題について述べる．

第 2 章

クラス拡張とその有効範囲の制御

クラス拡張はプログラムのモジュラリティを向上させる一方で，誤動作を引き起こしやすいという問題がある．クラス拡張は既存のクラスに対してソースコードを編集することなく外部からメソッドの追加や再定義を行えるような仕組みである．本章では，クラス拡張の詳細とその問題点を実例を交えつつ紹介し，さらに関連研究とその問題点についても述べる．

2.1 クラス拡張

直接ソースコードを編集することなく外部から既存のクラスに対してメソッドの追加や再定義を行えるような仕組みを我々はクラス拡張と呼んでいる．クラス拡張に類する仕組みは Ruby [3] や C#[9], アスペクト指向プログラミング (Aspect-oriented programming, AOP) [6], 文脈指向プログラミング (Context-oriented programming, COP) [8] 等，様々なプログラミング言語やプログラミングパラダイムに取り入れられている．クラス拡張を用いることで，複数のクラスにまたがる関心事 (横断的関心事，cross-cutting concern) [6] をモジュールとして切り出すことが可能になり，プログラム全体のモジュラリティの向上が期待できる．あるいは，組み込みクラスやサードパーティライブラリなど直接ソースコードを編集する権限がない場合でも外部からクラスの挙動を拡張することができるため，コードの再利用性が高まる．例えば Ruby ではオープンクラスがクラス拡張に相当する．Ruby のオープンクラスではクラスを拡張するための特別な構文は用意されておらず，クラス定義を行った際に同名のクラスが既に存在していた場合はそのクラスへの拡張となる．

図 2.1 はオープンクラスを用いて整数クラスの除算メソッドを有理数を返すように再定義する例である．Ruby では `class` 文 を用いてクラスを定義 (または再定義) する．また，メソッドの定義には `def` 文を用いる．この例では，`Fixnum` というクラスに対して `/` という名前のメソッドを定義していることになる．Ruby では除算のような基本的な演算子もメソッドとして実現されており，オープンクラスによるメソッドの再定義と組み合わせることで演算子のオーバーライドができる．この例では，これを整数型の除算を行っているプログラムに対して適用することで，そのプログラムそのものを書き換えずに数値計算の精度を変更できる．実際に Ruby で標準で提供される mathn ライブラリ内で同様のことが行われている．図 2.1 の

```
1  puts 1/2          # => 0
2
3  class Fixnum
4    def /(other)
5      quo(other)
6    end
7  end
8
9  puts 1/2          # => 1/2
```

図 2.1. オープンクラスによる組み込みクラスの書換え

ように組み込みクラスさえも容易に拡張できるという拡張性の高さは Ruby の大きな特徴といえる．

クラス拡張の強みは，クラスに対する変更が，そのクラスを元々参照していたクラスからも見えるという点にある (ただし，後述するようにこの利点は危険性の裏返しでもある)．図 2.2 に示すように，継承によるクラスの拡張では新たに別名のサブクラスを作成することになるため，そのクラスを利用していた別のクラスに対し，新たに拡張したクラスを利用させるためには，そのクラスのサブクラスも作成し，新たなクラスを参照するように書き換える必要がある．この例では，HTML の解析を行う HTMLReader クラスのテストを行うために，HTMLReader 内で使われているファイルを読み込むための Reader クラスのスタブを作成しようとしている．ReaderStub は Reader クラスを継承し read メソッドをオーバーライドし，定数文字列を返すようにしている．テスト対象は HTMLReader であるから，このクラスが読み込んでいる Reader クラスを，新たに作成した ReaderStub クラスに差し替え，この定数文字列に対する解析結果が正しいものであるかを調べれば良い．しかし，クラス拡張を使わない場合にはこれを直接実現することは出来ず，代わりに HTMLReader クラスのサブクラス HTMLReaderForTest を作成する必要がある．手間はかかるものの，見かけ上この例は動作する．しかし，read_html メソッドの中身をスーパークラスからコピーしてきて実装しているため，本来 HTMLReader のテストを行いたかったのにも関わらず，実質的に HTMLReaderForTest クラスへのテストになってしまう．開発が進むことで，HTMLReader#read_html の実装が変わっても，HTMLReaderForTest も追従して書き直さない限り新たな実装において正しくテストにパスするかわからず，ほとんどテストとして機能していないといえる．通常，クラス拡張を用いる場合はこのような問題は起きない．また，その他の対策としては，family polymorphism [15] や dependency injection [16] などが知られている．図 2.3 は dependency injection を用いて 図 2.2 の例を改善した例である．HTMLReader はもはや Reader クラスには依存しておらず，依存しているのは read メソッドを持つというインターフェースのみである．[*1] ゆえに，HTMLReader 内の Reader の実装はラインタイムに

[*1] ただし，Ruby には言語機構としてのインターフェースは存在せず，ただオブジェクトが read というメソッドを持っていれば良い．このようなランタイムのオブジェクトの振る舞いのみに依存した緩い型付けはダックタイピングと呼ばれる．

```
1  class HTMLReader
2    def read_html(path)
3      r = Reader.new.read
4      parse_html r.read path
5    end
6  end
7
8  class ReaderStub < Reader
9    def read(path)
10     return "<div>text</div>"
11   end
12 end
13
14 Reader.new.read "dummy"
15 # => "<div>text</div>"
16 HTMLReader.new.read_html "dummy"
17 # => No such file or directory - dummy
18
19 class HTMLReaderForTest < HTMLReader
20   def read_html(path)
21     r = ReaderStub.new.read
22     parse_html r.read path
23   end
24 end
25
26 HTMLReaderForTest.new.read_html "dummy"
27 # => #HTMLElement<DIV "text">
```

図 2.2. 継承によるクラスの拡張

交換可能になり，ここでは ReaderStub を実装として注入している．なお，通常，依存性の注入はそれを担うライブラリ等が一手に引き受け，設定ファイルやアノテーションなどによって定義したシステムの構成設定に従い自動的に各クラスに対して依存性が注入される．

2.2　クラス拡張の危険性

クラス拡張はモジュラリティの向上に寄与するが，一方クラス拡張の利用にはリスクが伴う．図 2.1 の例は数値型の除算の精度をグローバルに書き換えてしまう．これを本来の挙動を前提にしているプログラムに対して適用した場合，プログラム全体にどのような影響を与えるかを詳細に予測することは難しい．多くの場合，誤動作を引き起こす可能性が高い．他にも，複数の拡張が同一クラスを拡張する場合には拡張同士の変更が衝突して意図しない動作を引き起こす可能性がある．

```
1  class HTMLReader
2    def initialize(reader)
3      @reader = reader
4    end
5    def read_html(path)
6      r = @reader.read
7      parse_html r.read path
8    end
9  end
10
11 class ReaderStub < Reader
12   def read(path)
13     return "<div>text</div>"
14   end
15 end
16
17 HTMLReader.new(ReaderStub.new).read_html "dummy"
18 # => #HTMLElement<DIV "text">
```

図 2.3. Dependency Injection を用いて改善したスタブ

2.3 クラス拡張を安全に利用するために

クラス拡張を安全に利用するためには，利用者にとってその影響範囲が明確であるということが必要不可欠である．クラス拡張によって引き起こされる問題の原因の多くは，利用者がクラス拡張を適用する際に，その影響範囲を正しく予測できていないことによる．逆に言えば，クラス拡張の動作とその有効範囲が明らかでさえあれば，クラス拡張という仕組みそのものに起因する問題は起こらない．仮に，クラス拡張が既存のコードに対して悪影響を与える場合でも利用者がその事実を理解さえしていれば多くの場合何らかの対策を講じることが出来る．

Ruby のオープンクラスではクラス拡張は定義した瞬間グローバルに有効になるため，利用者にとってその影響範囲が明確であるとはいえない．セマンティクスそのものは単純で利用者にとっても明確だが，新たにクラス拡張を定義した際にそれがプログラム全体にどのような影響を与えるかということは明確ではない．これを正確に把握するためには利用者がプログラム全てについて理解している必要がある．拡張対象のクラスが自身の作成したクラスであればまだ良いが，組み込みクラスやサードパーティライブラリに対する拡張であった場合，その拡張対象のクラスがプログラム中でどの様に利用されているかをすべて把握するのは容易なことではない．

2.4 関連研究と既存手法

クラス拡張に類する言語機構には様々な物がある．必ずしも本研究のようにクラス拡張の有効範囲制御を主目的とはしていないが，多くの場合，提供される機能の一部として有効範囲の

```
1  module ExtFixnum
2    refine Fixnum do
3      def /(other)
4        quo(other)
5      end
6    end
7  end
8
9  class MyMath
10   using ExtFixnum
11   def half(other)
12     other/2
13   end
14 end
15
16 puts 1/2                          # => 0
17 puts MyMath.new.half(1)           # => 1/2
```

図 2.4. Refinements による組み込みクラスの書換え

制御を行う仕組みが含まれている．本節では関連研究及び既存手法を紹介する．

2.4.1　Ruby の refinements

Ruby の refinements のようにクラス拡張の有効範囲を静的スコープ下に制限する手法はその影響範囲が利用者にとって明確で安全性は高いが，拡張対象のメソッドがスコープをまたいで間接的に呼び出される場合に拡張を有効にすることは出来ない．図 2.4 は refinements を用いて図 2.1 の例を書き直したものである．Refinements はオープンクラスとは異なり，クラス拡張を定義しただけではその拡張が有効にならない．ここでは `ExtFixnum` という名前の拡張を定義し，これを `using` 宣言 (正確にはメソッド) を用いて `MyMath` クラス内で有効にしている．`ExtFixnum` の有効範囲は `MyMath` クラス内に限定されており，その外部には一切影響を与えない．このような手法は安全性という点では有効だが，間接的な呼び出しに対応できないため利用できる場面は限られており，例えば 図 2.5 に示すような例ではうまく動作しない．図 2.5 は，図 2.2 の例を refinements を用いて書き直したものである．しかし実際にはこの例は意図したとおりには動作しない．これは `ReaderStub` の有効化宣言 (16 行目) が，拡張対象である `Reader#read` メソッドの呼び出し (4 行目) から見て参照可能なスコープ内に位置していないことによる．

2.4.2　Selector namespaces

Selector namespaces は Modular Smalltalk [11] で導入された概念で，クラス拡張をモジュール化し，それを特定のスコープにインポートすることでクラス拡張の有効範囲をそのスコープ下に限定することができる．Ruby の refinements によく似ており，複数のスコープに

```
1  class HTMLReader
2    def read_html(path)
3      r = Reader.new
4      parse_html r.read path
5    end
6  end
7
8  module ReaderStub
9    refine Reader do
10     def read(path)
11       return "<div>text</div>"
12     end
13   end
14 end
15
16 using ReaderStub
17 Reader.new.read "dummy"
18 # => "<div>text</div>"
19 HTMLReader.new.read_html "dummy"
20 # => No such file or directory - dummy
```

図 2.5. Refinements によるスタブの実装 (意図通りには動作しない)

跨り間接的に呼び出されるメソッドを拡張することは出来ないという refinements と同様の問題を抱えている．

2.4.3 Context-oriented programming

Context-oriented programming (COP) [8] は文脈に依存したクラスの振る舞いをモジュールとして切り出し，実行時に文脈に応じて振る舞いを切り替えるプログラミングパラダイムであり，様々な実装が存在する [17][18][19][20]．このようなクラス拡張の有効範囲を動的スコープに基づいて決定する手法は，静的スコープに基づく手法に見られる制限は無いものの，グローバルスコープと同様の問題，すなわちクラス拡張の影響範囲が利用者にとって明確ではないという問題が存在する．図 2.6 に図 2.5 の例を動的スコープに基づく手法を用いて実装した例を示す．ただし，Ruby にはこのような機能は存在しないため，この例は擬似コードであることに注意されたい．この例では `with` 文を用いてクラス拡張 `ReaderStub` を有効にすることを宣言しており，またその有効範囲は続くブロック内に限定されている．このブロック内では拡張対象のメソッドが間接的に呼び出されても関係なく，常にクラス拡張は有効になる．したがって，図 2.5 では動かなかった例が図 2.6 では期待どおりに動作するようになる．しかし，動的スコープはグローバルスコープと同様，影響範囲の予測が難しいという問題がある．これまでの例を拡張し，HTML に CSS を読み込む機能を追加したという想定の例を図 2.7 に示す．この例では新たに CSS の解析を行う `CSSReader` クラスを追加し，これを `HTMLReader` 内で呼び出すようにしている．この条件下で同様に `ReaderStub` を `with` 文を用いて有効

```
1  class HTMLReader
2    def read_html(filepath)
3      parse_html Reader.new.read(filepath)
4    end
5  end
6
7  with ReaderStub do
8    # Activate the extension in this block
9    HTMLReader.new.read_html "dummy"
10 end
```

図 2.6. 動的スコープによる手法を用いたスタブの実装

```
1  class CSSReader
2    def read_css(filepath)
3      parse_css Reader.new.read filepath
4    end
5  end
6
7  class HTMLReader
8    def read_html(filepath)
9      parsed = parse_html Reader.new.read(filepath)
10     css_reader = CSSReader.new
11     styles = parsed.get_css_paths.map do |p|
12       css_reader.read_css p
13     end
14     parsed.set_style styles
15   end
16 end
17
18 with ReaderStub do
19   # Activate the extension in this block
20   HTMLReader.new.read_html "dummy"
21 end
```

図 2.7. CSS の読み込みに対応した HTML パーサ

化しようとすると，HTMLReader 内の Reader#read メソッドだけでなく，CSSReader 内の Reader#read にも影響を与えてしまう．仮にクラス拡張の利用者が CSSReader の実装の詳細についての知識がないものとすると，ReaderStub がプログラム全体にどのような影響を与えるかを利用者が予測することはできない．実際，ReaderStub は HTMLReader 向けの定数文字列を返すように再定義するものであった．したがって，CSSReader はこれを解釈できず，結果としてエラーを引き起こす．

2.4.4 Aspect-oriented programming

Aspect-oriented programming(AOP) [6] は，従来のオブジェクト指向プログラミングでは分離の難しい複数のクラスに跨る振る舞いを，アスペクトと呼ばれるモジュールに切り出す手法である．AOP の実装には様々なものがあるが [21][22]，代表的な実装の一つに AspectJ [23] がある．AspectJ は AOP のための Java 向けの拡張である．AspectJ においてアスペクトは 2 つの要素から構成される．1 つは対象クラスに対して与える振る舞いの変更そのものの定義であり，これを *advice* と呼ぶ（クラス拡張に相当）．もう 1 つは，advice をプログラムへどのように適用するかの定義であり，これを *pointcut* と呼ぶ．ユーザーは pointcut を適切に定義することで，クラス拡張の有効範囲を制御することができる．AspectJ で定義できる pointcut の幅は広く，柔軟な有効範囲の制御が可能である．例えば，AspectJ の提供する `cflow` オペレータを用いると，method seals のようにコールスタックの状態に基づいて拡張を切り替えることもできる．ただし，method seals と同等のセマンティクスを得るためには，複雑な条件式を自身で書く必要がある．また，method seals とは異なり，クラス拡張の影響範囲の認識性を保証する仕組みはない．

2.4.5 Classboxes

Classboxes [12] は *classbox* というモジュール内にクラス宣言とクラス拡張を定義することができる．ここで定義したクラスやクラス拡張はその classbox 内だけで有効だが，他の classbox からもインポートすることで自身の classbox 内で利用できるようになる．複数のスコープに跨り間接的に呼び出されるメソッドでも拡張可能 (この機能を *local rebindings* と呼ぶ) だが，一度インポートしたクラス拡張はその classbox 内で常に有効になるため，プログラマがその classbox 内の実装詳細をインポートしたものも含めて全て理解していなければ，意図しないクラス拡張の適用が起こりうる．これはスコープの定義こそ違えど，動的スコープやグローバルスコープと同様の問題である．例えば，図 2.8 は図 2.7 の例を classboxes を用いて書き直したものである．Classbox には Smalltalk 環境の一つである Squeak [24] による実装の他に，Java による Classbox/J [25] ，C#による実装 [26] が存在するが，ここでは Classbox/J を用いた．`ReaderCB` や `HTMLReaderCB` といった classbox にそれぞれ対応するクラスが記述されている．他の classbox のクラスをインポートするためには `import` 宣言を用いる．例えば，この例では `CSSReaderCB` では `ReaderCB` から `Reader` クラスをインポートし，`CSSReader` の実装に用いている．`HTMLReader` も同様である．`StubCB` では `Reader` クラスの拡張を `refine` キーワードを用いて定義している．エントリポイントを含む `AppCB` では，`StubCB` と `HTMLReader` をインポートすることで `Reader` のスタブが有効になった `HTMLReader` を利用する．しかし，`StubCB` のクラス拡張は `CSSReaderCB` へも影響を与える．したがって，動的スコープと同様の問題が発生する．このように，classboxes では，同一 classbox 内に同名のメソッドをそれぞれ違う実装で共存させることは出来ない．

```
1  package ReaderCB;
2  public class Reader {
3      public String read(String filePath) {
4          // ...
5      }
6  }
7  // ---
8  package CSSReaderCB;
9  import ReaderCB.*;
10 public class CSSNode {
11     // ...
12 }
13 public class CSSReader {
14     public CSSNode read(String filePath) {
15         Reader r = new Reader();
16         String s = r.read(filePath);
17         // ...
18     }
19 }
20 // ---
21 package HTMLReaderCB;
22 import ReaderCB.*;
23 import CSSReaderCB.*;
24 public class HTMLElement {
25     // ...
26 }
27 public class HTMLReader {
28     public HTMLElement read(String filePath) {
29         Reader r = new Reader();
30         CSSReader cr = new CSSReader();
31         // ...
32     }
33 }
34 // ---
35 package StubCB;
36 import ReaderCB.*;
37 refine Reader {
38     public String read(String filePath) {
39         return "<div>text</div>";
40     }
41 }
42 // ---
43 package AppCB;
44 import StubCB.*;
45 import HTMLReaderCB.*;
46
47 public class App {
48     public static void main (String[] argv) {
49         HTMLReader hr = new HTMLReader();
50         hr.read("dummy");       // => Error!
51     }
52 }
```

図 2.8. Classboxes を用いた Reader のスタブの作成 (動作しない)

2.4.6 MixJuice

MixJuice [27] は *module* 内にクラスやクラス拡張を定義することができる．また，一度定義された module は別の module から *extends* 宣言を用いて利用あるいは拡張することができる．Classboxes によく似ているが，classboxes の local rebindings に相当する仕組みはなく，間接的に呼び出されるメソッドの拡張には対応できない．一方，同一 module 内におけるメソッド名の衝突は *FQN(Fully-Qualified-Names)* と呼ばれる仕組みで回避可能である．これは，参照したい実装が定義された module 名もメソッド名に含めることでメソッド名の解決についての曖昧性を回避するものである．

2.4.7 Method shelters

Method shelters [13] は本研究の前身研究の 1 つで，*method shelter* というモジュールを導入している．Method shelters は Classboxes によく似ており，method shelter 内にクラスやクラス拡張を定義したり，他の method shelter のものをインポートすることができる．Classboxes と大きく違う点は, method shelter は内部が *exposed chamber* と *hidden chamber* という 2 つの部分に分かれており，どちらにクラスやクラス拡張を定義するかによって，外部からのインポート及び再定義を許可するかどうかを制御できるところにある．これを使い分けることで，classboxes における，同一 classbox 内に同名のメソッドをそれぞれ違う実装で同居させることは出来ないという問題を解決し，再定義され実装の異なる同名のメソッドを同一モジュール内で同時に利用することができる．しかし，拡張同士の衝突の回避という点では有効だが，どちらの chamber に定義するかはインポート先の実装者に委ねられているため，既に定義された shelter の構成によっては外部から思うように拡張できないことがある．例として，図 2.8 の例の一部を Method shelters で書き直したものを 図 2.9 に示す．この例は期待通りに動く．なぜなら，`CSSReaderShelter` では `ReaderShelter` は hidden chamber に配置されている一方，`HTMLReaderShelter` では `ReaderShelter` は exposed chamber に配置されているためである．これにより，`CSSReaderShelter` 内の `Reader` は `AppShelter` による再定義から保護されオリジナルの実装が呼び出される一方，`HTMLReader` のものに対しては再定義された実装が呼び出される．ただし，この例からわかるように，このようにうまくいくかどうかは元々の各 shelter の実装による．今回の例では，偶然 `CSSReaderShelter` では `ReaderShelter` を hidden chamber に配置し，`HTMLReaderShelter` では exposed chamber に配置されていたために期待通り動いたが，この例は非常に恣意的である．現実には，この 2 つの shelter が同一作者によるものであったら `ReaderShelter` は hidden chamber と exposed chamber のいずれかに統一されて配置される可能性が高い．また，元々この例では，クラス拡張の利用者は `CSSReader` の実装の詳細について理解していないという仮定があった．`AppShelter` の実装者はやはり `CSSReader` の実装について知らなければ，`Reader` の再定義がどのような影響を与えるかを予測することは出来ない．つまり，インポートする側がイ

```
1  shelter :CSSReaderShelter do
2    class CSSReader
3      def read_css(file_path)
4        # ...
5      end
6    end
7
8    hide
9    import :ReaderShelter
10 end
11
12 shelter :HTMLReaderShelter do
13   import :ReaderShelter
14   import :CSSReaderShelter
15   class HTMLReader
16     def read_html(file_path)
17       # ...
18     end
19   end
20 end
21
22 shelter :AppShelter do
23   import :HTMLReaderShelter
24   class Reader
25     def read(file_path)
26       "<div>text</div>"
27     end
28   end
29 end
```

図 2.9. Method shelters を用いた Reader のスタブの作成

ンポート先の実装について理解していないと依然として意図しない動作を引き起こしかねなく，グローバルスコープや動的スコープのような問題は依然として存在する．

2.4.8　Method shells

Method shells [14] は本研究のもう 1 つの前身研究である．Method shells は method shelters が複雑化しすぎたことを踏まえ，chamber を廃止し，別途 2 種類のインポート方法を用意することでよりシステムを簡素にした．また，インポート元でどのようにインポートするかを制御できるようになったため，安全性が高まった．しかし，単純なシンタックスを獲得した一方で依然として複雑なセマンティクスを持っており，暗黙のうちに複雑なメソッドディスパッチが行われるためやはりプログラマが全体像を把握することは難しく，意図しない挙動を引き起こしうる．本研究はこれを踏まえ，暗黙的な解決を諦めることでより単純で安全なセマンティクスを得ることを目指したものである．

図 2.10 はこれまでの例を同様に method shells を用いて実装した例である．Method shells

では methodshell というモジュールにクラスやクラス拡張を定義する．別の methodshell にあるクラスやクラス拡張を利用するためには include 宣言を用いる．ここでは例えば，Reader クラスを用いて CSSReader クラスを実装するために css_redear methodshell は reader methodshell を include している．クラス拡張は revise キーワードを用いて定義する．例えばここでは html_reader_stub methodshell 内で Reader クラスのクラス拡張を作成している．include によって連結された methodshell は shell グループを形成する．methoshell m をルートとする shell グループを m shell グループと呼ぶことにすると，app shell グループは，app methodshell，html_reader methodshell , reader methodshell からなる shell グループになる．同一 shell グループ内のクラスとクラス拡張は一度に有効になる．また，link 宣言を用いて別の shell グループを参照することができる．ここでは，html_reader methodshell は css_reader shell グループを参照している．link 宣言によって参照された shell グループはその link 宣言を含む methodshell 内のメソッド・ルックアップ時に優先的に探索される．また，link 先の methodshell 内のメソッドを呼び出している間はそのメソッドの属する shell グループがクラスやクラス拡張の探索に使われる．この 2 種類の include と link をうまく使い分けることでクラス拡張の有効範囲を制御する．図 2.10 は期待通り動作するが，これも method shelters と同様，html_reader methodshell から css_reader methodshell へ link しているということが前提になっている．

```
1  methodshell reader;
2  public class Reader {
3      public String read(String filePath) {
4          // ...
5      }
6  }
7  // ---
8  methodshell css_reader;
9  include reader;
10 public class CSSNode {
11     // ...
12 }
13 public class CSSReader {
14     public CSSNode read(String filePath) {
15         Reader r = new Reader();
16         String s = r.read(filePath);
17         // ...
18     }
19 }
20 // ---
21 methodshell html_reader;
22 include reader;
23 link css_reader;
24 public class HTMLElement {
25     // ...
26 }
27 public class HTMLReader {
28     public HTMLElement read(String filePath) {
29         Reader r = new Reader();
30         CSSReader cr = new CSSReader();
31         // ...
32     }
33 }
34 // ---
35 methodshell html_reader_stub;
36 include html_reader;
37 revise Reader {
38     public String read(String filePath) {
39         return "<div>text</div>";
40     }
41 }
42 // ---
43 methodshell app;
44 include html_reader;
45
46 public class App {
47     public static void main (String[] argv) {
48         HTMLReader hr = new HTMLReader();
49         hr.read("dummy");       // => Error!
50     }
51 }
```

図 2.10. Method shells を用いた Reader のスタブの作成

第 3 章

Method seals

本研究では，2 章で述べたようなクラス拡張が意図しない領域で有効になることによって起こりうる問題を回避するため，有効範囲を既知のコールパスに限定することで安全に利用できる新たなクラス拡張 *method seals* を提案する．Method seals では，利用者がクラス拡張を適用したい範囲を 「意図したところ」 として明示できる．「意図したところ」 と明示しない限りは，そのパッケージとそこから呼び出されるパッケージではクラス拡張が無効になる．すなわち，クラス拡張は必ず利用者の理解の及ぶ範囲でしか拡張が有効にならず，意図しない動作を引き起こす危険性を軽減できる．

本章では method seals の詳細とその利用例について述べる．また，method seals の妥当性や制限事項についても議論する．

3.1 未知のコールパスにおけるクラス拡張の無効化

Method seals では，コードを読んで理解した部分 (パッケージ)，すなわち 「意図したところ」 を利用者が明示できるようになっており，「意図したところ」と明示しない限り，そのパッケージととそこから呼び出されるパッケージ群ではクラス拡張が無効になる．ここでの「意図したところ」 とは，利用者が 「コードを読んで理解した部分（パッケージ)」 を意味している．元来主観的な 「意図したところ」 を言語機構として簡潔に表現するのは困難であることから，ここでは少なくとも利用者の理解の及ばないところにクラス拡張の影響が及ぶことによって混入するバグを防ぐということを目指し，このような定義になっている．この定義の妥当性についての議論は 3.2 節で行う．

Method seals では，パッケージ p とその先で呼び出されるパッケージでクラス拡張 e を無効にすることを 「クラス拡張 e についてパッケージ p を *seal* する」と呼ぶ．また，クラス拡張 e についてパッケージ p に付与された seal 属性を剥がすことを 「クラス拡張 e についてパッケージ p を *unseal* する」と呼ぶ．デフォルトでは全てのパッケージは seal されており，プログラマはクラス拡張の有効範囲を広げるために自身の理解度と必要性に応じて各パッケージを unseal していくことになる．

Method seals で実際にクラス拡張の有効化及び unseal を行うためには unseal 付き using

```
1  using ReaderStub, [HTMLReader]
2  Reader.new.read "dummy" # => "<div>text</div>"
3  HTMLReader.new.read_html "dummy"
```

図 3.1. Method seals によるクラス拡張の有効化

宣言を用いる．拡張の有効無効を切り替えるアルゴリズムは次のようになる．

(a) using 宣言でクラス拡張 e を有効化．宣言を含むパッケージ p で有効になる．
(b) パッケージ p から別のパッケージ q のメソッドを呼び出すと，
　(b1) q が using 宣言の unseal パッケージリストに含まれていれば q 内部でもクラス拡張 e が有効になる．以降 (b) を繰り返す．
　(b2) q が unseal パッケージリストに含まれていない場合は q 内部とその先に呼び出されるパッケージではクラス拡張 e が無効になる．

unseal 付き using 宣言は具体的には例えば図 3.1 のようになる．これはクラス拡張 ReaderStub を有効にし，さらにそれについて HTMLReader を unseal している．using によってまずは自身を含むパッケージでクラス拡張が有効になる (a)．したがって，2 行目の Reader#read 呼び出しはクラス拡張が適用されたものになる．次に HTMLReader#read_html を呼び出すことを考える (b)．今回は HTMLReader は unseal パッケージリストに含まれているため，HTMLReader でもクラス拡張 ReaderStub が有効になる (b1)．したがって，図 2.7 の 9 行目の Reader#read 呼び出しはクラス拡張が適用されたものとなる．一方，12 行目の CSSReader#read_css 呼び出しを考えると，また別パッケージのメソッド呼び出しになるため，(b) の処理が繰り返される．今回は unseal パッケージリストに CSSReader は含まれていないため，クラス拡張は無効になり，CSSReader 内部の Reader#read 呼び出しは拡張前のものが呼び出されることになる (b2)．仮に CSSReader が更に別のパッケージのメソッドを呼び出していたとしても，これ以降 (b) の処理は行われない．

図 3.2 に method seals を用いたシステムの全体像を示す．この図において重要な点は，using を呼び出しているパッケージ (トップレベル) とその using で unseal しているパッケージ (HTMLReader) が拡張の有効なパッケージとなっており，そしてその拡張が有効なパッケージから直接 Reader#read を呼び出している場合に限って，その呼び出し先は ReaderStub によって拡張されたものになっている，というところにある．図が示すように，CSSReader は拡張の有効なパッケージとはなっておらず，CSSReader からは拡張前の Reader#read が呼び出されることになる．ただし，先述したように unseal したパッケージで常に拡張が有効になるというわけではない．ある拡張が有効なパッケージ群は，必ずその拡張に対する using 宣言を含むパッケージを起点として連続した領域になる．例えば，加えて CSSReader についても unseal した場合は 図 3.3 のようになる．拡張の有効なパッケージの範囲が拡張され，CSSReader からも拡張された Reader#read が呼ばれるようになっている．一方，図 3.4 に示すように，CSSReader のみ unseal した場合では，拡張が有効な範囲はトップレベルにまで

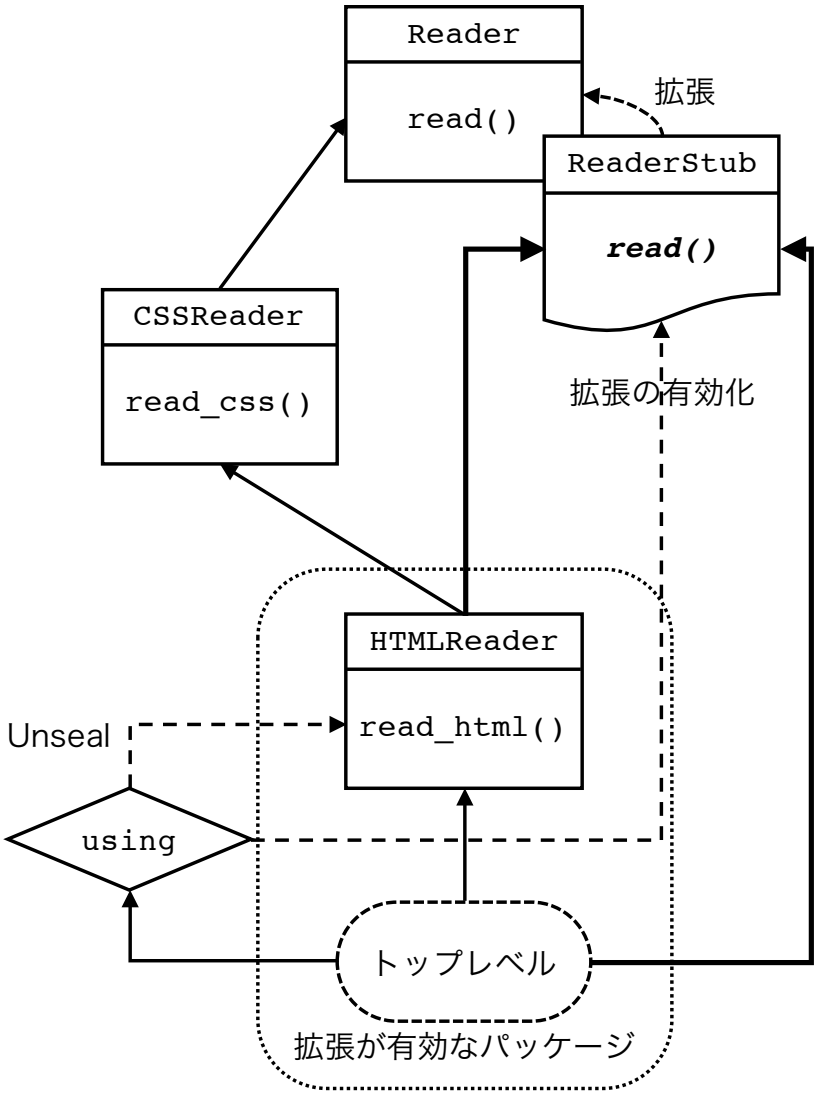

図 3.2. Method seals を用いたシステムの全体像

縮小される．Method seals では seal されたパッケージを飛び越えてパッケージを unseal してもそこで拡張を有効にすることを許さないため，このような結果になる．

3.2 Method seals の有効性

3.1 節 では method seals のセマンティクスと利用例について述べた．本節では，method seals がなぜそのようなデザインになっているのか，また，なぜ本研究で取り組んでいる問題に対して有効であるかについて述べる．

これまで述べてきたように，本研究の目的はクラス拡張の機能性を可能な限り維持しつつ，誤動作を引き起こす危険性を低減したクラス拡張を提案することであり，これに沿って method seals は設計されている． 2 章ではクラス拡張の持つ危険性と，クラス拡張を安全に利用するために何が必要か，すなわち，利用者にとってクラス拡張の影響範囲が明確である必

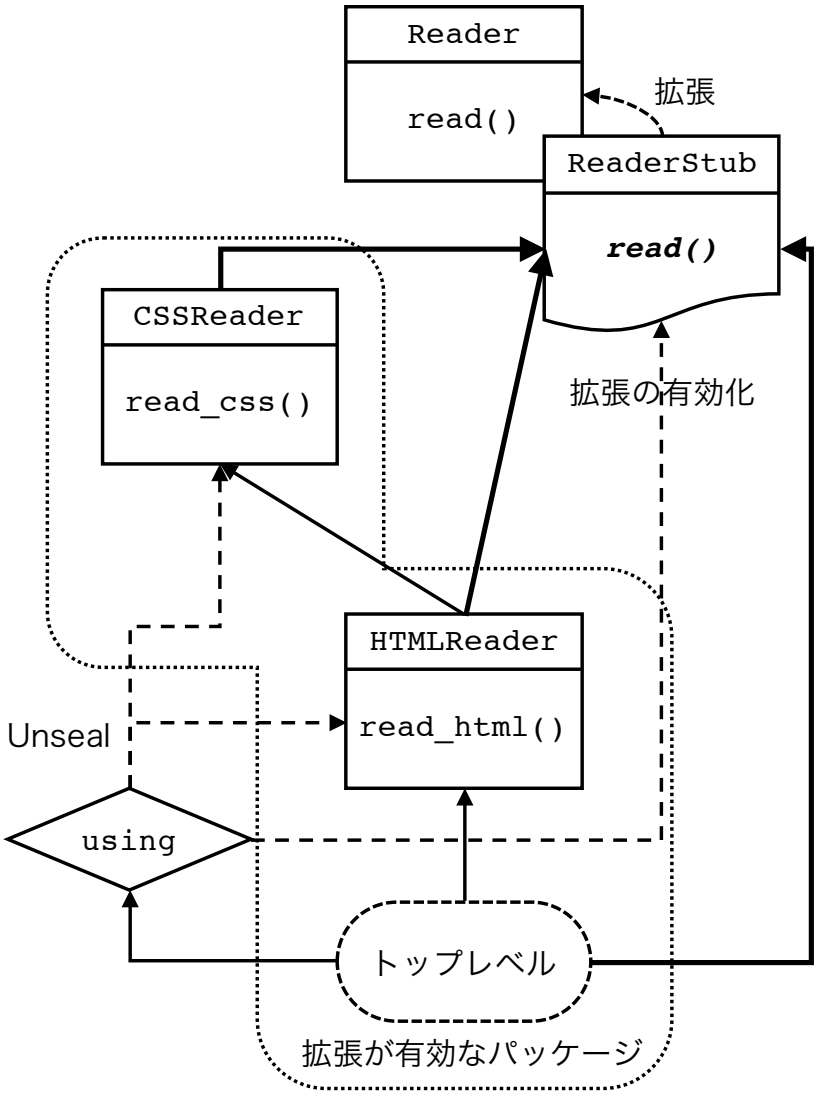

図 3.3. 加えて CSSReader を unseal した場合の全体像

要がある，ということを述べた．理想的には利用者の意図したままにクラス拡張の有効範囲が定まれば，利用者にとってこれ以上の明確さは無いと言える．しかし現実には，元来主観的な利用者の 「意図したところ」 を簡潔な言語機構で表現することは困難である．同一コードベースに対してであっても，求められる理想的な動作は文脈や利用者の意図によって変わり得るため，言語機構側が暗黙的に利用者の意図を読み取ることは実質不可能であり，利用者側が自身の意図を伝えるために何らかのパラメータを渡す必要がある．あらゆる意図を汲み取れるように言語機構を拡張していくと，言語機構側に渡さなければいけないパラメータのパターンは肥大化し，複雑化してしまう恐れがある．そこで我々は，利用者が 「意図したところ」 を柔軟にに表現できるような言語機構を考案することは目指さず，代わりに, method seals の既定する単純なルールの範囲内で 「意図したところ」 を表明出来るようにした．これを実現するためには，「意図」 として表明できる内容に制限をかける必要があり，本機構ではこれを利

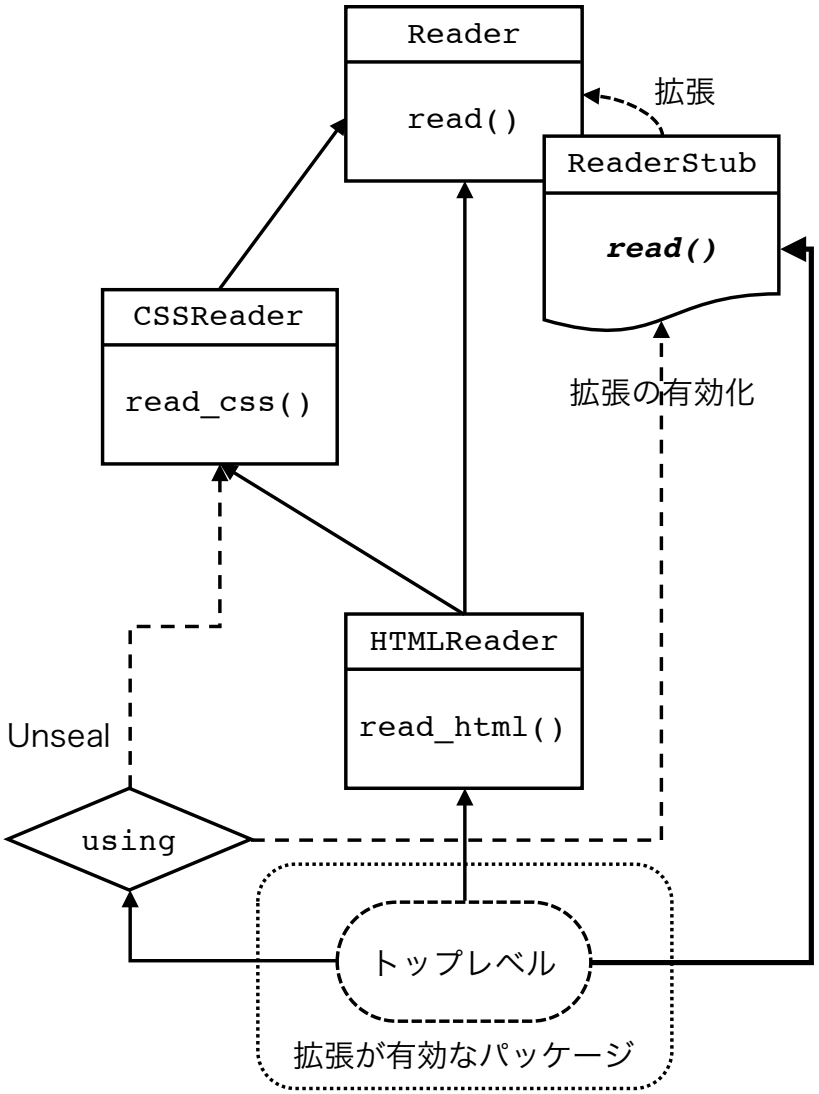

図 3.4. CSSReader のみ unseal した場合の全体像

用者が 「コードを読んで理解した部分」かどうかという点に限定している．これはクラス拡張が利用者の「知らないところ」 で有効になることで混入するバグを防ぐということを最重要視した結果である．また，これには，仮にクラス拡張が一部の既存のコードに対して望ましくない影響を及ぼす場合であっても，その事実さえ利用者が理解していれば，多くの場合は何らかの対策は行えるだろうという期待が根底にある．

更に我々は 「知らないところ」 を「未知のパッケージとそのパッケージから呼ばれる全てのパッケージ」 と定義している．Refinements のように静的な手法を取る場合では，ブラックボックス (「知らないところ」，拡張を有効にすべきでないところ) は図 3.5 の上段のようになる．つまり，クラス拡張の利用者にとって未知のパッケージである `UnknownClass` そのものがブラックボックスである，という考え方である．クラス拡張の有効範囲をレキシカルスコープ下に制限する場合では，そのスコープはソースコード中の構文構造から決まるため，利

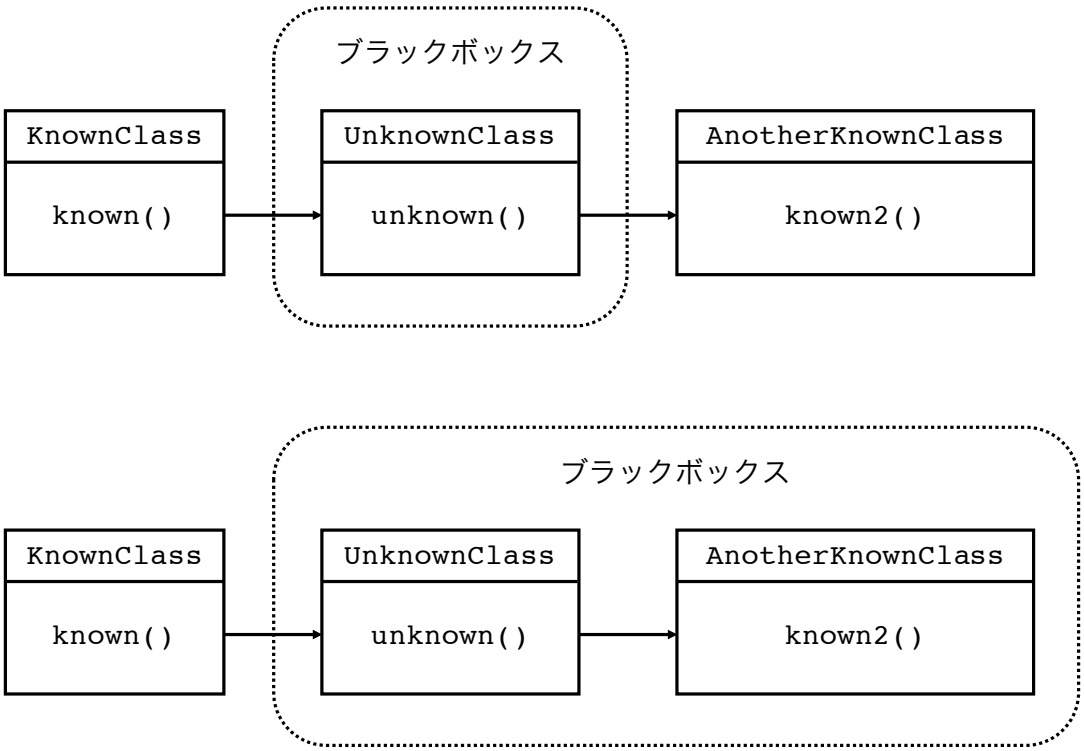

図 3.5. ブラックボックスとして扱う領域の比較

用者の各パッケージのソースコードに対する理解度がそのままランタイムにも適用できる．この事自体は非常に単純明快で，ソースコードレベルの理解とランタイム時の挙動に齟齬が発生しにくいため，変数等のスコープにレキシカルスコープを用いることは良いアイディアに思える．事実多くのプログラミング言語はレキシカルスコープを採用している．

しかしながら，本研究では機能性のために動的な手法をとっており，この静的な考え方をそのまま適用しても，その 「利用者のソースコードレベルの理解とランタイム時の挙動に齟齬が発生しにくい」 というメリットは享受出来ない．したがって，method seals のように動的な手法で安全性を確保するためには考え方を変える必要がある．そこで，我々はブラックボックスの定義を図 3.5 の下段に示すような範囲に拡張した．静的な手法におけるブラックボックスに加え，そこから呼び出されるパッケージ群も，そのパッケージが既知のパッケージであるかどうかに関わらず，ブラックボックスとして扱う．

クラス拡張による変更の影響はその拡張対象クラスの呼び出し元，さらにその呼び出し元へと次々伝搬していく．図 2.7 の例において，クラス拡張の影響がどのように伝搬していくかを 図 3.6 に示す．まず，クラス拡張 `ReaderStub` が有効化されると `Reader` へ影響を及ぼす．`ReaderStub` の影響を受けて振る舞いが変化した `Reader` をここでは便宜上 `Reader'` と表すことにする (以降同様)．`Reader` の元々の振る舞いは与えられたパスのファイルの中身を読み取って返すものであったが，`Reader'` は与えられたパスは常に無視し，定数文字列を返すようになる．さらに，`Reader'` は `Reader` を利用していた全てのパッケージに影響を与える．すなわち，ここでは `HTMLReader` と `CSSReader` である．また，`CSSReader'` は `HTMLReader`

へ影響を与える．HTMLReader は既に Reader’ によって振る舞いが変わっているので，それに加えて振る舞いが変わることになり，これをここでは HTMLReader’’ とする．このように，クラス拡張の影響は次々に伝搬していき，上流のクラスの振る舞いのパターンは爆発的に増えていく．[*1] この例では本来は HTMLReader’ が望ましい振る舞いであったが，最終的に HTMLReader’’ の振る舞いが得られることになる．これは Reader’ が CSSReader に影響を与えること，ひいては更に CSSReader’ が HTMLReader に影響を与えることを利用者が事前に知る手段がないために引き起こされる問題である．クラス拡張の影響は害のあるなしにかかわらず必ず上流へ伝わるため，コールパス上に未知のパッケージがあった場合，そのパッケージにも必ず変更の影響は伝搬しうる．そしてその影響がそのパッケージに害を与えるかどうかはそのブラックボックスたるパッケージの中身を見ない限りは決してわからない．したがって，コールパス上に未知のパッケージが現れたらそれ以降全てのパッケージをブラックボックスとして扱う必要がある．

3.3 制限事項

Method seals にはいくつかの制限事項が存在する．本節ではこれらの制限事項について述べ，また，考えうる対策について議論する．

3.3.1 無害なクラス拡張に対する冗長性

明らかに無害なクラス拡張に対してはクラス拡張を有効にする手順が冗長になりすぎるという問題がある．無害なクラス拡張の例としては，自身が作成したクラスに対する拡張など，影響範囲が明らかである場合や，拡張前の挙動と互換性を持つように丁寧に設計されたクラス拡張などがあげられる．図 3.7 は Ruby の著名なライブラリである Active Supoort [28] の利用例である．Active Support は 同じく著名な Web アプリケーションフレームワークである Ruby on Rails [4] に含まれるライブラリで，Ruby の標準ライブラリに対する拡張のコレクションを提供する．その一つが数値型クラスの文字列表現を得るメソッド to_s を拡張し，電話番号等へのフォーマットに対応するものである．to_s の元々の挙動は基数を受け取り，それに応じた表現を文字列として返すというもので，したがって受け取る引数は数値型である．それ以外の値が与えられると TypeError を引き起こす (2行目)．これに対して Active Support を有効にすると，基数を与えられた場合には既存の挙動と同様に動作しつつ，特定のシンボルが与えられた場合はそれに応じたフォーマッタを利用する (4行目)．拡張前の，数値型以外のパラメータを渡した際にエラーになる，という挙動に依存したコードが存在した場合に問題が起こる可能性があるという点では，完全に問題がないとはいえないが，多くの場合このクラス拡張によって問題が起こる可能性は極めて低いだろう．

議論の余地はあるものの，このようなクラス拡張には method seals を用いる必要がない．

[*1] 図 3.6 には載せていないが，実際にはこれに加えて，HTMLReader の振る舞いパターンには Reader’ からの影響は受けずに CSSReader’ からの影響のみを受けるパターンも有る．

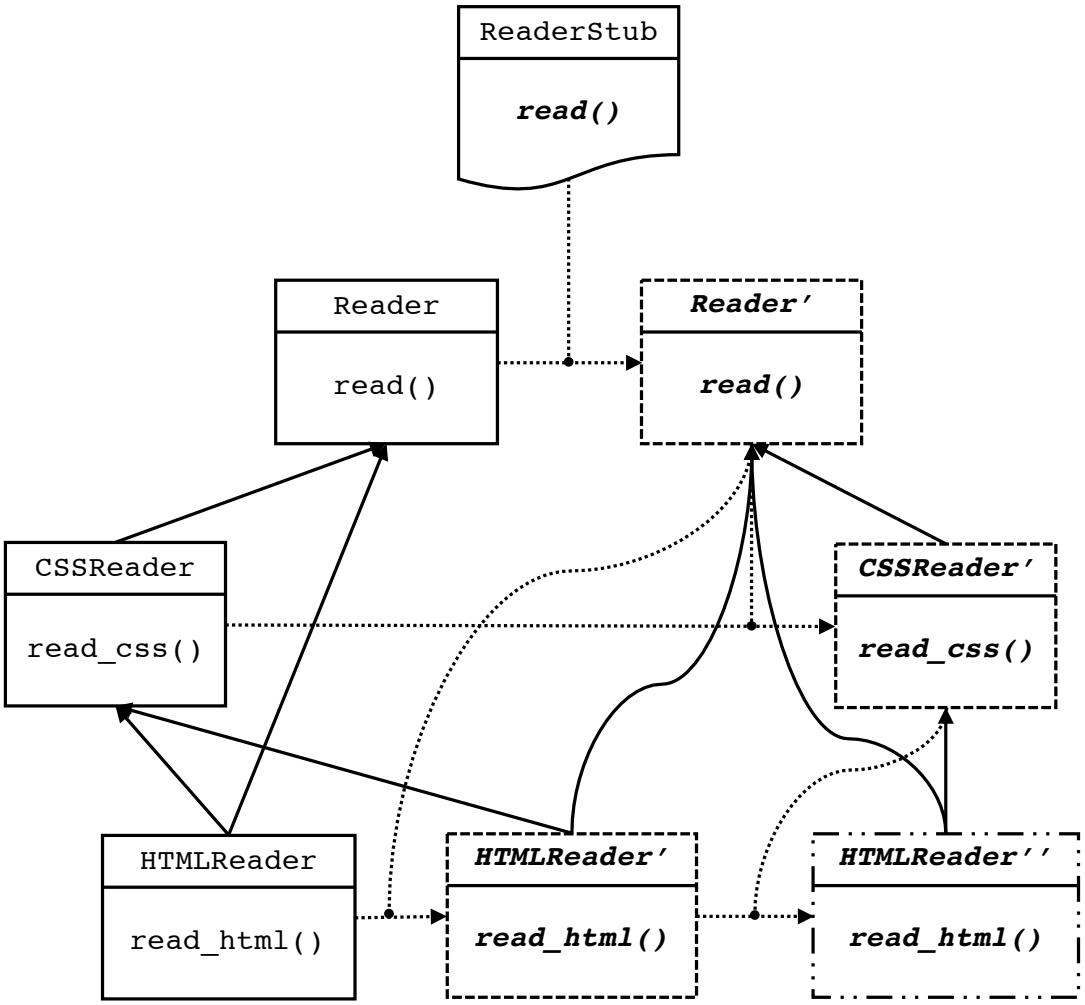

図 3.6. 伝搬してゆくクラス拡張の影響

```
1  5551234.to_s(2)                  # => "10101001011010010000010"
2  5551234.to_s(:phone)             # => TypeError
3  require 'active_support/all'
4  5551234.to_s(:phone)             # => "555-1234"
```

図 3.7. Active Support による互換性を考慮したクラス拡張

利用者にとって本当に安全であるという確信があるクラス拡張であれば，オープンクラスや動的スコープを用いれば良い．あるいは動的スコープを用いてもよい (ただし，Ruby には存在しない)．Active Support のようなライブラリが広く利用されていることからわかるように，安全性よりも利便性を優先したいというニーズは現実には往々にしてあることで，本機構のような安全よりの仕組みとオープンクラスのような利便性よりの仕組みを使い分けていくことが現実解となるのではないかと考えている．実際，Ruby にはオープンクラスと安全寄りにデザインされた refinements という，２つのクラス拡張に類する機構が同居しており，目的によって使い分けられている．

3.3.2 using のネスト

Method seals では using 宣言を用いてクラス拡張を有効にするが，この using が入れ子になる場合が考えられる．この時，method seals の安全性が損なわれる可能性がある．図 3.8 および図 3.9 にこの一例を示す．この例では C3#m3，C2#m2，C1#m1 の順にメソッドが連鎖的に呼ばれる．その際，トップレベルでは C2 に対するクラス拡張 ExtC2 を，C3 内では C1 に対するクラス拡張 ExtC1 をそれぞれ using している．つまり，2 つの using 宣言が同一コールパス上で入れ子構造にある．各 using は適切なパッケージを unseal しているので，C2#m2 と C1#m1 はそれぞれ拡張されたものが呼び出される．このことは 3.1 節 で述べたセマンティクスから明らかである．しかしながら，この動作が本研究で method seals の安全性の根拠として掲げている，「知らないところでは拡張が無効になる」，という原則に従っているかどうかについては疑問が残る．C3 内の using において C2 を既知のパッケージとして宣言しているが，全体としてみると C2 は外側の using 宣言によって拡張されているため，C3 の設計者が想定している C2 の挙動が実際のものと異なる可能性がある．それにも関わらず，C2 を既知のものとして ExtC1 を有効にすることは，「知らないところでは拡張が無効になる」という原則に反するという見方もできる．

この問題について十分な検討が行われたとは言い難いが，現時点ではこの問題への対応として 3 通りの方向性があるのではないかと考えている．

まず 1 つは，内側の using でクラス拡張を有効にするためには，C2 だけでなく C2 を拡張するクラス拡張，すなわち ExtC2 についても unseal しない限り，拡張された C2 を既知のパッケージとして扱わないという方針である．これは「知らないところでは拡張が無効になる」という原則から考えると理に適っているように思われる．しかし，現実には扱いにくい側面もある．この方針では，C3 を設計した時点，自身がどのようなクラス拡張と組み合わせて使われるかについての知識が必要である．こうなると，C3 は外部と密結合になり，組み合わせて利用できるクラス拡張の種類が設計時点で決まってしまうことになり拡張性が損なわれる．

一方，現状のセマンティクスを維持するという選択肢もある．外側の using 宣言では，C3 を既知のパッケージとして宣言している．C3 が既知であるということは，C3 内の using 宣言の存在についても外側の using の設計者は了解しているはずであり，ExtC2 と ExtC1 の組み合わせで何が起きるかも理解しているはずであるから，ExtC1 を有効にしても問題ない，というのがこの選択に関しての考え方である．しかし，本来のセマンティクスに立ち返って考えて見るとこの方針にも違和感が残る．C3 が既知のパッケージだからといって，その中で参照されているパッケージの中身まで既知とは限らない．ExtC1 も例外ではなく，ExtC1 が C3 内で参照されているからといって，本来その中身まで理解している必要はない．

そこで，3 つ目の方針としては，外側の unseal で内側の using で指定されているクラス拡張を指定するようにするということが考えられる．ここで指定しなかった場合，外側の using で有効にしようとしていたクラス拡張は無効になる．図 3.8 の例では，外側の using(38 行目) で unseal リストに ExtC1 を追加することになる．C3 において ExtC1 を有効にすると，その

```
1  class C1
2    def m1
3      puts "m1"
4    end
5  end
6
7  class C2
8    def m2
9      puts "m2"
10     C1.new.m1
11   end
12 end
13
14 class C3
15   using ExtC1, [C2]
16   def m3
17     puts "m3"
18     C2.new.m1
19   end
20 end
21
22 module ExtC1
23   refine C1 do
24     def m1
25       puts "m1'"
26     end
27   end
28 end
29
30 module ExtC2
31   refine C2 do
32     def m2
33       puts "m2'"
34       C1.new.m1
35     end
36   end
37 end
38
39 using ExtC2, [C3]
40 C3.new.m3
```

図 3.8. ネストした using 宣言

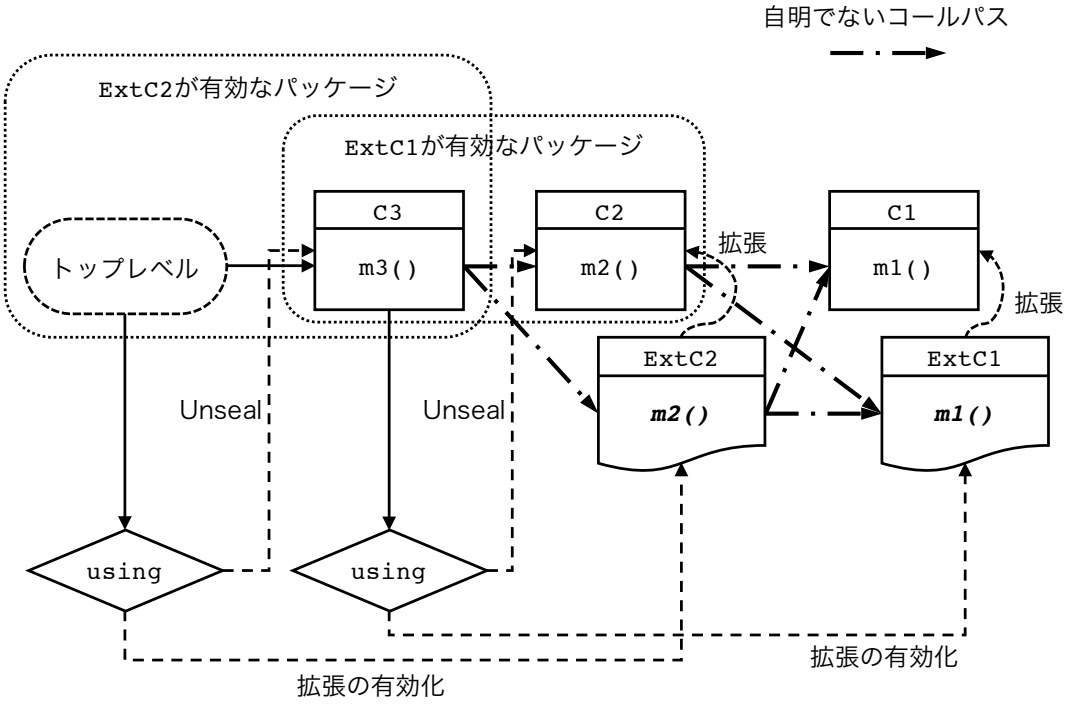

図 3.9. ネストした using 呼び出しにおける自明でないコールパス

影響範囲は自身 (C3) から拡張対象そのものである C1 までのコールパス上の全てのパッケージであり，すなわちここでは C3, C2, C1 になる．つまり，これらのパッケージは，ExtC1 によって変容しており，ExtC1 の実装について理解していないかぎり，拡張前のパッケージについて理解しているだけではこれらを既知のパッケージとしてみなすことはできない．したがって，外側の using において，ExtC1 も unseal しない限り，C2 以降のコールパス上で ExtC2 が有効にならないという動作は「知らないところでは拡張が無効になる」という原則にも従っており理屈に適っているといえる．したがって，現段階ではこの 3 つ目の方針が最も妥当であると考えられる．

また，以上の方針と平行して，ネストした using が干渉しあう場合は警告を出す事も考えられる．

3.3.3 unseal されたパッケージに対するコードの変更

Method seals ではパッケージを unseal を用いて既知のものと宣言することでクラス拡張の有効無効を切り替えるが，一度 unseal したパッケージのコードベースが開発が進むなどして変更された場合，そのパッケージはもはや既知のパッケージとは言えないという懸念がある．これまで述べてきたように，method seals では「知らないところでは拡張が無効になる」ことを根拠に安全性を主張しており，その前提には unseal 操作によって，各パッケージの既知フラグが正しく設定されていることがある．したがって，この前提が崩れると，method seals 全体の安全性の主張が難しくなってしまう．ただし，この問題は本研究特有の問題では

なく，Aspected-oriented programming や Context-oriented programming など，クラス拡張に類する言語機構では，外部から既存のクラスに対して変更を加えるという性質上，一般に拡張対象のコード変更に対して弱いといえる．例えば，AOP の分野では，プログラムの編集前に指定していたポイントカット（変更を差し込む場所の定義）が編集後には正しく動かくなってしまう問題が広く認識されている (*fragile pointcut* [29])．しかしながら，method seals ではコールパス上に現れるパッケージの構成にまで依存しているため，特にコードの変更の影響を受け易いといえる．この問題について，現時点で明確な対策はないが，既存のコードベースに対してハッシュ値を取ってそれを元に変更を検知するなどは考えられる．ただし，言語内に完結した機能として実現することは困難であることが予測されるため，外部ツールによる支援に頼らざるを得ない可能性が高い．Aspected-oriented programming の分野では，クラス拡張 (アスペクト) の有効範囲を可視化するなどして AOP によるプログラミングを支援する開発環境についての研究・開発が盛んに行われている [30][31][32]．同様のアプローチを用いてクラス拡張の有効範囲の視覚化するなどして，こうした問題を防止するということも考えられる．

第 4 章

実装

本研究では，3 章で提案した method seals を Ruby 上に実装した．最も一般的に使われている Ruby の公式処理系である MRI(Matz' Ruby Implementation) を拡張することでこれを実現している．ベースとなっているバージョンは Ruby 2.1.4 (r48285) である．本章で言及する Ruby の内部アーキテクチャについても，当該バージョンに基づくことに注意されたい．本章では，MRI の概要と我々が MRI に対して行った拡張の詳細について述べ，また，本実装に先駆けて Ruby のライブラリとして実装したプロトタイプ版との比較についても言及する．

4.1 MRI(YARV)

Ruby には JRuby [33] や Rubinus [34] 等様々な処理系が存在するが，本研究では公式処理系である MRI に対して実装を行った．MRI は最も広く使われている処理系で，C 言語によって記述されている．MRI の内部実装については [35] に詳しい．MRI における Ruby プログラムの実行の流れを図 4.1 に示す．Ruby プログラムはまずパーサージェネレータ (Bison [36]) により生成されたパーサーによって抽象構文木 (AST) へと変換される．さらにこれを YARV(Yet Another Ruby VM) と呼ばれる仮想機械向けのバイトコード命令列へコンパイルし，これを YARV が解釈実行することでプログラムが実行される．MRI ではバージョン 1.9 より YARV を導入した．バージョン 1.8 以前では，抽象構文木を直接解釈実行しており，YARV の導入によりパフォーマンスは飛躍的に向上した．

YARV はバイトコード命令列を処理するために用いられる内部スタックとは別に，Ruby レベルのコールスタックも保持している．これは通常，Ruby のメソッドが呼ばれる際に新たにスタックフレームをプッシュし，離脱する際にポップする．スタックフレームには現在のプログラムカウンタやスタックポインタ，呼び出し中のメソッドのレシーバなどの情報が含まれている．我々の実装ではこのスタックフレームを拡張することで，各メソッド呼び出し間でコンテキストを共有している．ここで共有しているコンテキストには，現在のコールパス上で using されているクラス拡張とそれらに関して unseal されているパッケージに関する情報が含まれている．

また，MRI のメソッドディスパッチはメソッドの種類によって大きく異る．メソッドの種

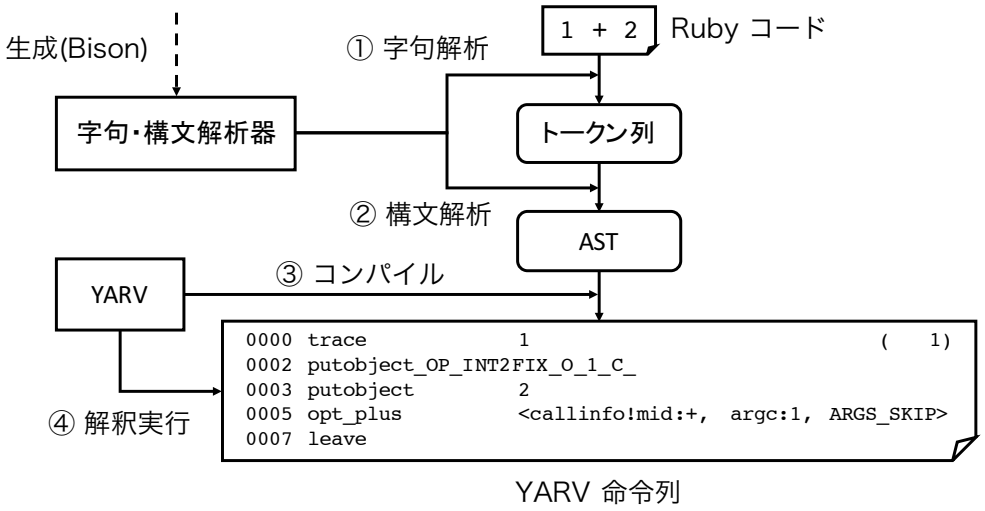

図 4.1. Ruby 処理系 (MRI) の全体像

類には 11 種類あり，それぞれ異なる処理を行う．Refinements によって追加あるいは再定義された (可能性のある) メソッドは内部的には VM_METHOD_TYPE_REFINED というメソッドタイプになっており，メソッドディスパッチ時に現在のレキシカルスコープ上に保存されている refinements の情報を探し，見つかればそれに応じて拡張されたメソッドを呼び出す．我々の実装は refinements に対する拡張となっており，この VM_METHOD_TYPE_REFINED 向けのメソッドディスパッチを拡張し，スタックフレームを介して共有してきたコンテキストを用いて実際に実行するメソッドの内容を決めるようにしている．Refinements が行うこのメソッドディスパッチは一般に非常にコストがかかるが，MRI ではインラインメソッドキャッシュによって，前回の処理の結果が保存されているので，2 回目以降の呼び出しでは高速になる．ただし，現在の我々の実装では VM_METHOD_TYPE_REFINED タイプのメソッドについてはインラインメソッドキャッシュが働かないようにしている．これは，MRI に実装されている，メソッドが再定義されている可能性があるかかどうかを判断するためのアルゴリズムが，method seals のセマンティクスには適用できず古いキャッシュが使われてしまうため，これを回避する必要があったためである．詳細についてはは 5.2 節 で実験の結果と合わせて述べることにする．

4.2　処理の流れ

我々の実装では 3.1 節 で示したアルゴリズムをほぼそのまま素朴に実装している．クラス拡張を有効にしてから拡張対象のメソッドが呼ばれるまでの処理の流れは大まかに次の通りである．

(a) using 宣言でクラス c に対するクラス拡張 e を有効化．この時，e と unseal パッケージのリストに関する情報を現在のレキシカルスコープ (cref) に保存する．

(b) メソッド呼び出しが行われると新たにスタックフレームを作成し，これをコールスタックに積む．このとき，
  (b1) 現在のレキシカルスコープまたはスタックフレームに e 関する情報が存在するか調べ，存在する場合はさらに呼び出すメソッドが属するパッケージが unseal パッケージリストに含まれているかについて調べる．含まれている場合は新たにコールスタックに積むスタックフレームに e に関する情報を埋め込む．以降，拡張対象のクラス c に到達するまで (b) を繰り返す．
  (b2) e に関する情報が見つからないか，呼び出すメソッドが属するパッケージが unseal パッケージリストに含まれていない場合はスタックフレームには e に関する情報は埋め込まれない．
(c) c のメソッドが呼び出される． このとき行われるメソッドルックアップにおいて，現在のスタックフレームに e に関する情報があった場合は e を有効にする．

4.3 プロトタイプ版との比較

本研究では，Ruby 処理系を拡張することで method seals の実装を行ったが，これに先駆けて Ruby のライブラリという形で実装したプロトタイプ版が存在する．プロトタイプ版では Ruby で実装されており，通常のライブラリとして利用する．ライブラリとして実装している関係上，文法上の制限やパフォーマンス上の問題を抱えていた．本節ではプロトタイプ版と最新の実装の違いについて述べる．

4.3.1 シンタックス上の相違点

3.1 節 でも示したように，method seals のシンタックスは Ruby の refinements の自然な拡張になっている．図 4.2 は method seals を用いてクラス拡張を定義する例である．これは refinements と全く同一である．クラス拡張を定義するためには `module` を定義し，その中で `refine` メソッドを呼び出す．`refine` メソッドは拡張対象となるクラスと，拡張の内容を定義したブロックを受け取る．メソッドを再定義する場合は，メソッド内で `super` を用いて再定義前の実装を呼び出すことができる．一方，プロトタイプ実装におけるクラス拡張の定義を図 4.4 に，クラス拡張の有効化を図 4.5 に示す．クラス拡張を定義するためには `seal_define` メソッドを用いる．`seal_define` の第 1 引数には拡張対象のクラス，第 2 引数にはクラス拡張の名前をシンボルで渡す．定義したクラス拡張を有効化するためには `seal_using` メソッドを用いる．`seal_using` の第 1 引数にはクラス拡張の名前，第 2 引数には unseal 対象のクラスをリストで渡す．更にクラス拡張を利用するコードをブロックとして渡す．Method seals は refinements の自然な拡張として表現できる (何も unseal しなければ refinements と同等) が，ライブラリとして実装した都合上，独自のシンタックスを持つに至った．

```
1  class A
2    def a
3      "A"
4    end
5  end
6
7  module ExtA
8    refine A do
9      def a
10       "Extended␣" + super
11     end
12   end
13 end
```

図 4.2. Method seals におけるクラス拡張の定義

```
1  def B
2    using ExtA
3    def b
4      A.new.a
5    end
6  end
7
8  def C
9    def c
10     A.new.a
11   end
12 end
13
14 using ExtA
15 A.new.a
16
17 using ExtA, [C]
18 C.new.c
```

図 4.3. Method seals におけるクラス拡張の有効化

```
1  seal_define Reader, :ReaderStub do
2    def_method(:read) do
3      "<div>text</div>"
4    end
5  end
```

図 4.4. プロトタイプ実装におけるクラス拡張の定義

```
1  seal_using(:ReaderStub, [HTMLReader]) do
2    HTMLReader.new.read_html "dummy"
3  end
```

図 4.5. プロトタイプ実装におけるクラス拡張の有効化

```
1 t = TracePoint.trace(:call) {|tp|
2   # Obtain current class
3   tp.self.class
4 }
5
6 # Disable hook
7 t.disable
```

図 4.6. TracePoint を用いたメソッドコールへのフック

4.3.2 実装方法上の相違点

Method seals を実現するためには，各メソッド呼び出しに対して method seals に関する処理を挟むか，拡張対象となるメソッドが呼ばれる際にコールスタックの情報を取得できる必要がある．Ruby ではコールスタックに容易にアクセスする方法はないが，主にデバッガやプロファイラでの利用を想定した API である `TracePoint` を用いることで各メソッド呼び出しに対してフック処理を差し込むことは比較的容易にできる．図 4.6 は `TracePoint` を用いた例である．`TracePoint#trace` メソッドにイベントの種類とフック処理の内容をブロックで渡すことで毎メソッドコールごとに独自の処理をはさみこめる．これを用いて，フック処理内で現在のクラスを取得し，この情報と unseal パッケージを比較することで拡張の有効無効を切り替えることができた．また，プロトタイプ版ではメソッドの定義には `def` を用いず，`def_method` という独自の API を使うようにしていたが，これは拡張に含まれるメソッドの取得を容易にするためである．ここで取得したメソッドは各拡張を表す文脈オブジェクトに格納される．拡張が有効の有効化宣言が行われるとこれらのメソッドはモジュールに変換され，拡張対象のクラスに対し `prepend` する．`prepend` とは，モジュールを対象のクラスの継承関係の最前面に追加する Ruby のメソッドで，最前面に追加されたモジュール内のメソッドはメソッド・ルックアップ時に最初に見つかるものとなる．したがって，ここでは拡張対象のメソッドを追加あるいは再定義する手段として用いている．また，`def_method` で定義したメソッドの内容は実際には method seals のためのメソッドディスパッチを行う処理が差し込まれる．これによって，そのメソッドが呼ばれるまでのコールパスの情報を利用し，拡張前のメソッドと拡張後のメソッドのどちらを呼べばよいか判断し，適切な処理を呼び出す．

第 5 章

実験

4 章で述べたように，method seals の実装においてはスタックフレームやメソッド・ルックアップ等，言語の根幹部分への拡張を行っている．したがって，method seals によるパフォーマンスへの影響が広範に及ぶと予測される．本章では，method seals のオーバーヘッドについて調査するために行った実験について述べる．実験環境は CentOS release 6.2，Intel(R) Xeon(R) CPU E5-2687W 3.10GHz，メモリは 64GB である．

5.1 通常のメソッド呼び出しにおけるオーバーヘッド

まず，method seals を用いない場合のメソッド呼び出しのオーバヘッドを計測した．空のメソッドを 10 秒間繰り返し呼び出し，1 秒あたりのイテレーション回数の平均を計算した．この結果を 表 5.1 に示す．この結果から，我々が拡張した処理系は拡張前のものに対し，3% 程度のオーバーヘッドが存在することがわかった．Method seals では新たにスタックフレームを積む際に毎回，現在クラス拡張が `using` されていないかを確認し，必要に応じて前のスタックからクラス拡張に関する情報を引き継ぐ処理を行う．これにより，method seals を使わない場合でも全てのメソッドコールに対して多少のオーバーヘッドが発生するものとみられる．

5.2 クラス拡張を用いた場合のメソッド呼び出しにおけるオーバーヘッド

次に，クラス拡張を利用した場合のメソッド呼び出しのオーバーヘッドを計測した．本実験で用いたプログラムの一部を図 5.1 に示す．空のメソッドに対し，クラス拡張を用いて同様

表 5.1. クラス拡張を用いないメソッド呼び出しのパフォーマンス

	1 秒あたりの平均イテレーション数	標準偏差 (%)
標準 VM	9.280×10^6	2.7%
拡張 VM	9.038×10^6	1.2%

に空のメソッドで再定義し，この再定義されたメソッドを繰り返し呼び出すことでパフォーマンスを計測した．また，これをオープンクラスを用いた場合，refinements を用いた場合，method seals を用いた場合のそれぞれについて計測した．Method seals を用いる場合については最終的に拡張されたメソッドが呼ばれる場合 (unsealed) と，拡張前のメソッドが呼ばれる場合 (sealed) の 2 通りについて計測した．なお，オープンクラスは構文が異なるため，別途図 5.2 に示すベンチマークプログラムを作成した．条件は 5.1 節に示したものと同様である．この計測結果を 表 5.2 に示す．この結果が示すように，オープンクラスと refinements を用いた場合では 表 5.1 に示したクラス拡張を用いない場合とくらべても殆ど実行速度に変化が見られない．一方，method seals を用いた場合はそうでない場合に比べて大きく性能が劣化していることが分かる．特に，最終的に拡張されたメソッドが呼び出される場合においては，2 倍以上のオーバーヘッドが存在する．これはコールパス上の全パッケージと unseal されたパッケージとの比較処理のオーバーヘッドが大きいためと予測される．ただし，我々の実装では，`using` 呼び出しの時点で unseal されたクラスには印を付けておき，明らかに seal されているパッケージがコールパス上に現れたらそれ以降 method seals に関する処理の殆どを行わないようにしている．拡張前のメソッドが呼び出される場合ではいくらかパフォーマンスの劣化が穏やかなのは，このことが有効に働いていると思われる．

また，method seals を用いた場合で共通して現れるオーバーヘッドは，インラインメソッドキャッシュの有無によるものだと推測される．Ruby は本来前回のメソッドルックアップをキャッシュすることによってメソッドディスパッチを高速化するが，現在の我々の実装では，クラス拡張によって拡張されたメソッドについてはインラインメソッドキャッシュを無効にしている．Ruby のインラインメソッドキャッシュでは，クラスが再定義された場合にステートカウンタをインクリメントし，このステートカウンタがキャッシュが作成された時のものと同じである限りはそのキャッシュが利用され続ける．しかしながら，method seals ではクラスが再定義されない場合でも辿ったコールパス次第でメソッドの定義が変化しうるため，既存のアルゴリズムでは正しいキャッシュが読み込まれず，正しくない挙動を引き起こす可能性がある．そこで我々の現状の実装では拡張対象のメソッドに関してのみインラインメソッドキャッシュを無効にしており，結果としてパフォーマンスへの影響も大きくなったものと見られる．なお，refinements が，破壊的にクラスを書き換えるオープンクラスやそもそもクラス拡張を利用していない場合に匹敵するパフォーマンスを見せているのは，このインラインメソッドキャッシュのおかげで毎回 refinements に関する処理を行う必要が無いためと予測される．

実際にインラインメソッドキャッシュの影響を調査するため，全体としてインラインメソッドキャッシュを無効にした上で同様の計測を行った結果を 表 5.3 に示す． 表 5.2 の結果と比較すると，オープンクラスと refinements に関しては大きくパフォーマンスが劣化した一方，method seals を用いた場合では比較的小さな変化にとどまった．今回のベンチマークプログラムでは，再定義されたメソッドの呼び出しのみを計測対象としているため，method seals を用いた例ではインラインメソッドキャッシュの恩恵を全く受けていなかった．したがって，全体としてインラインメソッドキャッシュを無効にしても結果に変化がないのは当然と言える．また，refinements はオープンクラスに比べるとインラインメソッドキャッシュを無効に

```
1  # Define a class which has an empty method to measure the
2  # overhead of method calls.
3  class TargetClass
4    def empty_method
5    end
6  end
7
8  # Define a class extension to redefine the empty method.
9  module TargetClassExtension
10   refine TargetClass do
11     def empty_method
12     end
13   end
14 end
15
16 # Activate the extension
17 using TargetClassExtension
18
19 c = TargetClass.new
20
21 # Run the benchmark against the redefined method calls.
22 run_bench do
23   c.empty_method
24 end
```

図 5.1. クラス拡張を用いたメソッド呼び出しのベンチマークプログラム

```
1  # Define a class which has an empty method to measure the
2  # overhead of method calls.
3  class TargetClass
4    def empty_method
5    end
6  end
7
8  # Redefine the empty method with open class.
9  class TargetClass
10   def empty_method
11   end
12 end
13
14 c = TargetClass.new
15
16 # Run the benchmark against the redefined method calls.
17 run_bench do
18   c.empty_method
19 end
```

図 5.2. オープンクラスを用いたメソッド呼び出しのベンチマークプログラム

表 5.2. クラス拡張を用いたメソッド呼び出しのパフォーマンス

	1 秒あたりの平均イテレーション数	標準偏差 (%)
Open class	9.162×10^6	0.9%
Refinements	9.220×10^6	1.9%
Method seals(unsealed)	4.628×10^6	0.3%
Method seals(sealed)	7.099×10^6	0.4%

表 5.3. クラス拡張を用いたメソッド呼び出しのパフォーマンス (インラインキャッシュ無効時)

	1 秒あたりの平均イテレーション数	標準偏差 (%)
Open class	7.954×10^6	1.5%
Refinements	6.392×10^6	1.1%
Method seals(unsealed)	4.416×10^6	0.6%
Method seals(sealed)	6.720×10^6	0.3%

した影響が大きく出ており，method seals において最終的に拡張前のメソッドが呼び出されるケースよりも性能が落ちている．これは元来 refinements によって拡張されたメソッドの呼び出しは一般のメソッド呼び出しよりもコストが大きく，そのためインラインメソッドキャッシュの恩恵がより大きかったものと思われる．

5.3 同一コールパス上のクラス拡張数によるパフォーマンスの変化

using によって有効化されたクラス拡張が同一コールパス上に複数現れる場合，クラス拡張ごとに処理を行う必要が有るため，クラス拡張が増えれば増えるほどパフォーマンスが劣化することが予測される．これを確認するため，同一コールパス上のクラス拡張の数を変化させたときに観測されるパフォーマンスの変化を計測した．

この実験では，指定した数のクラス拡張を用いるベンチマークプログラムを自動生成し，生成したプログラムごとにパフォーマンスを測定した．各プログラムごとの測定についてはこれまでの条件と同様である．図 5.3 にクラス拡張数 2 で生成したベンチマークプログラムの一部を示す．指定した数のクラス拡張を生成し，それらを 1 つのクラスに対して適用するようになっている．

この実験によって得られた結果を 図 5.4 に示す．予測通り，クラス拡張数が増えるにつれ，パフォーマンスが低下する傾向にあることがわかる．ただし，一般に同一コールパス上のクラス拡張の数はそこまで大きくならず，我々の想定するユースケースでは多い場合でも 100 にも満たないと思われる．クラス拡張が 1000 ある場合でも 10 %にも満たないオーバーヘッドで

```
1  class C
2    def m
3    end
4  end
5
6  module Ext0
7    refine C do
8      def m
9      end
10   end
11 end
12
13 module Ext1
14   refine C do
15     def m
16     end
17   end
18 end
19
20 using Ext0
21 using Ext1
22
23 c = C.new
24 run_bench do
25   c.m
26 end
```

図 5.3. コールパス長 2 で生成したベンチマークプログラム

あり，クラス拡張数の増加による速度低下が実用上問題になる可能性は低い．

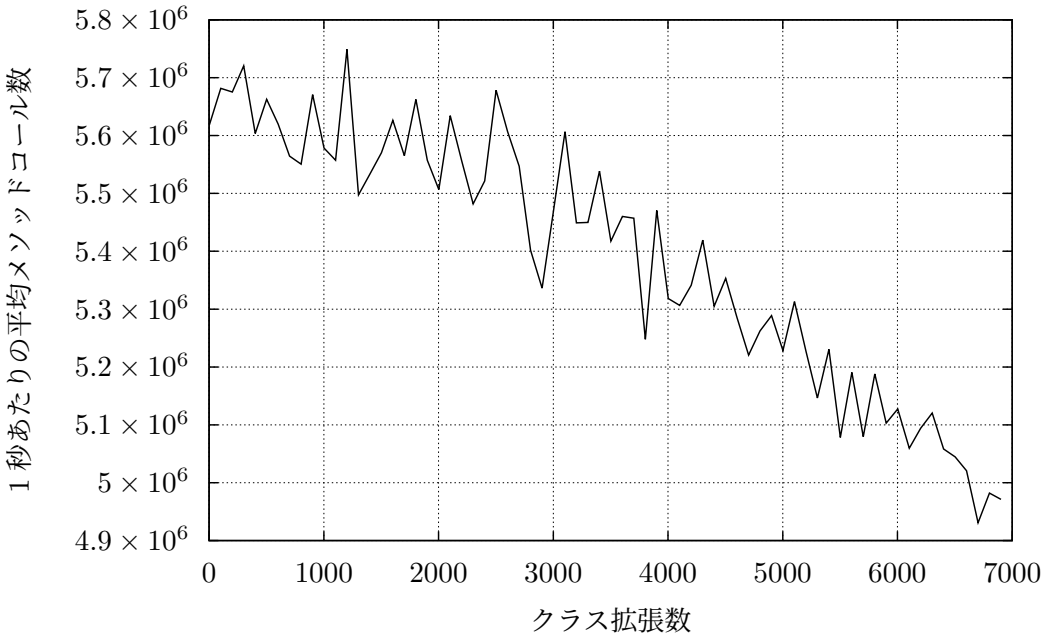

図 5.4. クラス拡張数によるメソッド呼び出しパフォーマンスの変化

第 6 章

まとめと今後の課題

6.1 まとめ

本研究では，クラス拡張を安全に利用するために，プログラマのコードに対する理解度に応じて段階的に有効範囲を拡張できる言語機構 method seals を提案した．クラス拡張は，直接ソースコードを書き換えることなく，既存のクラスに対してメソッドの追加や再定義を行える．一般に，クラス拡張は既存のクラスを拡張するという性質上，拡張対象のクラスがプログラム上のどこで使われているかを完全に把握した上で利用しなければ不意にプログラムを破壊しかねない．クラス拡張を安全に利用するためにはその有効範囲を適切に制御する仕組みが必要不可欠である．Ruby の refinements 等，静的な手法では高い安全性を得ることができているものの，複数のレキシカルスコープをまたぎ間接的に呼びだされたメソッドでクラス拡張を有効にすることはできず，利便性の点では問題があった．また，動的なスコープ用いた既存手法では，モジュラリティの向上など機能性に重点を置くものが多く，安全性はそれほど重視されてこなかった．ある程度有効範囲を限定することで，利便性を確保しつつもある程度のリスクを軽減することはできていたが，その限定された有効範囲内においては常にクラス拡張が有効になるため，その影響を予測することは依然として難しい．

そこで，我々はクラス拡張の有効範囲を利用者が理解している領域に限定することで，クラス拡張が不用意に有効になることで引き起こされる問題を回避する言語機構 method seals を提案した．Method seals では，利用者に対しソースコード上の既知の領域とそうでないところを明示することを強制し，これらの領域が正しく設定されている限り，クラス拡張は必ず利用者の理解の及ぶ範囲内でのみ有効になる．また，利用者が理解しているソースコードの範囲をそのまま静的に拡張の有効範囲として適用するのではなく，コールパス上にブラックボックス (利用者が理解していないソースコードを含むパッケージ) が現れた時点で，それ以降のコールパス上全てでクラス拡張を無効とした．これはクラス拡張によって振る舞いが変化したパッケージの影響は，そのパッケージを利用する別のパッケージへと伝搬していき，それはコールパス上のブラックボックスにも到達しうるためである．ブラックボックスへの影響の大きさを調べるためにはブラックボックスの詳細を調べるほかない．

更に我々は method seals を Ruby 処理系を拡張することで実装し，また，その実効性能に

ついて確認するための実験を行った．この実験結果によって，我々の実装にはまだ性能上の課題があることがわかった．また同時に，その要因としては，そもそものメソッド・ルックアップのコストの大きさに加え，インラインメソッドキャッシュが効かないことが大きいと確認するに至った．

6.2 今後の課題

今後の課題としては，提案手法をより実用的なプログラムに対して適用することで基本的なアイディアの有用性を検証する，実用化に向けてパフォーマンスの改善を図る等が考えられる．また，3.3 節 で述べたような method seals における制限事項について何らかの対策を講じる必要がある．

発表文献と研究活動

(1) 福室嶺，千葉滋． コールスタックに基づいてクラス拡張の有効範囲を制御するための言語機構の提案． 日本ソフトウェア科学会第 32 回大会， 2015.9.8-11.

参考文献

[1] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. *SIGPLAN Not.*, Vol. 35, No. 10, pp. 130–145, October 2000.

[2] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed multijava: Balancing extensibility and modular typechecking. In *ACM SIGPLAN Notices*, Vol. 38, pp. 224–240. ACM, 2003.

[3] David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly Media, Inc., 2008.

[4] David Heinemeier Hansson. Ruby on rails. `http://rubyonrails.org/`.

[5] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[7] Andreas Paepcke. *Object-oriented programming: the CLOS perspective*. MIT press, 1993.

[8] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, Vol. 7, No. 3, pp. 125–151, 2008.

[9] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] Lewis Pinson. *Object-Oriented Programming with Objective-C*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[11] Allen Wirfs-Brock and Brian Wilkerson. A overview of modular smalltalk. In *ACM SIGPLAN Notices*, Vol. 23, pp. 123–134. ACM, 1988.

[12] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems & Structures*, Vol. 31, No. 3, pp. 107–126, 2005.

[13] 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*, pp. 131–142. ACM, 2012.

[14] Wakana Takeshita and Shigeru Chiba. Method shells: avoiding conflicts on destructive class extensions by implicit context switches. In *Software Composition*, pp. 49–64. Springer, 2013.

[15] Erik Ernst. Family polymorphism. In *ECOOP 2001—Object-Oriented Programming*, pp. 303–326. Springer, 2001.

[16] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://www.martinfowler.com/articles/injection.html`, January 2004.

[17] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of contextl. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS '05, pp. 1–10, New York, NY, USA, 2005. ACM.

[18] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented Programming with Java. *Information and Media Technologies*, Vol. 6, pp. 399–419, 2011.

[19] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with contexts. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 5235 LNCS, pp. 396–407, 2008.

[20] Robert Hirschfeld, Hidehiko Masuhara, and Atsushi Igarashi. L: Context-oriented programming with only layers. In *Proceedings of the 5th International Workshop on Context-Oriented Programming*, p. 4. ACM, 2013.

[21] Sean McDirmid, Matthew Flatt, and Wilson C Hsieh. Jiazzi: new-age components for old-fasioned java. In *ACM SIGPLAN Notices*, Vol. 36, pp. 211–222. ACM, 2001.

[22] Shigeru Chiba and Rei Ishikawa. Aspect-oriented programming beyond dependency injection. In *ECOOP 2005-Object-Oriented Programming*, pp. 121–143. Springer, 2005.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *ECOOP 2001—Object-Oriented Programming*, pp. 327–354. Springer, 2001.

[24] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. In *ACM SIGPLAN Notices*, Vol. 32, pp. 318–326. ACM, 1997.

[25] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/j: Controlling the scope of change in java. *SIGPLAN Not.*, Vol. 40, No. 10, pp. 177–189, October 2005.

[26] Markus Lumpe and Jean-Guy Schneider. On the integration of classboxes into c#. In *Software Composition*, pp. 307–322. Springer, 2006.

[27] Yuuji Ichisugi and Akira Tanaka. Difference-based modules: A class-independent module mechanism. In *ECOOP 2002—Object-Oriented Programming*, pp. 62–88.

Springer, 2002.

[28] David Heinemeier Hansson. Active support. `https://github.com/rails/rails/tree/master/activesupport`.

[29] Maximilian Stoerzer and Juergen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 653–656. IEEE, 2005.

[30] Andy Clement, Adrian Colyer, and Mik Kersten. Aspect-oriented programming with ajdt. In *ECOOP Workshop on Analysis of Aspect-Oriented Software*, 2003.

[31] Terry Hon and Gregor Kiczales. Fluid aop join point models. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pp. 712–713, New York, NY, USA, 2006. ACM.

[32] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pp. 2503–2512, New York, NY, USA, 2010. ACM.

[33] JRuby. `http://jruby.org/`.

[34] Evan Phoenix, et al. Rubinius: The ruby virtual machine. `http://rubinius.com/`.

[35] Pat Shaughnessy. *Ruby Under a Microscope: Learning Ruby Internals Through Experiment*. No Starch Press, 2013.

[36] John Levine. *Flex & Bison: Text Processing Tools*. O'Reilly Media, Inc., 2009.

謝辞

本研究を進めるにあたり，研究方針や論文構成，発表手法，研究アイディアなど多岐にわたって手厚くご指導賜りました指導教員の千葉滋教授に心より感謝致します．

また，日々本研究の議論にお付き合い頂き，様々な示唆を与えてくださった市川和央氏に感謝の意を表します．市川氏には本研究に直接関わることだけでなく，日々の研究生活に至るまで幅広くご支援頂きました．Maximilian Scherr 氏には本研究への有用な助言や英文校正などのご支援を頂き，大変感謝しております．

最後に，日頃から研究室生活やミーティングを通じてご支援くださった千葉研究室の皆様と，当初は筆者の大学院への進学に反対していたにも関わらず最終的に応援し支えて下さいました両親に心より感謝致します．