平成20年度 修士論文

横断的関心事を扱うための
オブジェクト指向言語の拡張

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 07-3704-5

今吉 竜之介

指導教員

千葉 滋 教授

平成21年1月30日

概要

従来のオブジェクト指向プログラミング (OOP) ではロギングやイベント通知といった横断的関心事がソースコード上に散在することが多く、それらを一箇所にまとめることが難しい。そのためプログラムの拡張をする場合、散在している全ての箇所において同じ修正を施す必要があった。例えばデバッグコードを挿入する場合、プログラマはそのコードを呼び出したい全ての箇所に呼び出し命令を記述しなければならない。また、その修正箇所を全て把握していなければ、デバッグコードを除去する場合に漏れや余分な削除といった人為的なバグが混入する恐れがある。この問題を解決するため、本研究では OOP を拡張した新たな言語を提案する。

この言語では、従来の Java をベースに OpenClass、PredicateDispatch、グローバルコンテキストへの参照を導入する。OpenClass とは外部からも仕様が追加可能なクラスのことである。提案言語ではこの機能を実現するため、任意のクラスを拡張する「refine-class」を実装した。これにより横断的関心事の挿入・削除が容易になる。PredicateDispatch とはパラメータの状態によってモジュールの挙動を動的に選択する技術である。これは一つのモジュールに混在する関心事を個々のモジュールとして分離する。本言語では「conditional-method」を用いることで、従来の method dispatch をより細かい条件分岐に対応させる。さらにグローバルコンテキストへの参照により、オブジェクトは caller の情報を取得できる。これらの技術により、我々は散在する横断的関心事を一箇所にまとめる。

横断的関心事を扱う技術としては、既にアスペクト指向プログラミング (AOP) がある。しかし AOP 言語は技術の習得が難しい。例えば代表的な AOP 言語である AspectJ は、ポイントカットやアドバイスといった概念とそれらの文法の理解が必要となる。本研究が提案する言語は、AOP ではなく従来の Java の概念を拡張したものであり、また、この言語は覚えるべき仕様が AspectJ と比べて少ない。そのため習得にかかるコストが AOP 言語よりも少ないという利点がある。

実装は、オープンソースの Java ベースコンパイラである JastAdd を用いておこなった。JastAdd は与えられたソースコードの構文を解析し、得られた個々のノードに対応する Java オブジェクトを生成する。オブジェクトが持つ属性をアドオンのように操作することで我々は構文追加やコン

パイラの挙動変更の実装をおこなった。

謝辞

本研究を進めるにあたり、研究の方向性や構成などの多くの点にわたり指導して頂いた千葉滋教授に心より感謝致します。また、本研究を行う上で様々な指導、協力、激励を頂いた同研究室の皆様に感謝致します。

目 次

第 1 章　はじめに　6

第 2 章　OOP の問題点と既存技術　8
2.1 OOP の問題点 8
2.1.1 関心事の散在 8
2.1.2 異なる関心事の混在 10
2.2 求められる技術 12
2.3 関連研究 12
2.3.1 AOP 12
2.3.2 MultiJava 13
2.3.3 JPred 14

第 3 章　拡張 OOP の提案　15
3.1 拡張言語による問題の解決 15
3.1.1 OpenClass: refine 節の導入 15
3.1.2 PredicateDispatch: when 節の導入 17
3.1.3 外部コンテキストの参照: client 変数の導入 19
3.2 提案言語の利点 20
3.3 仕様 21
3.3.1 refines 節: refine クラス 21
3.3.2 refine クラスによる型階層の変更 28
3.3.3 メソッドの add, replace 31
3.3.4 when 節: 条件付きメソッドの add 36
3.3.5 client 変数; caller の参照 43
3.3.6 メンバ変数の add, replace 47
3.3.7 コンストラクタの add, replace 49
3.3.8 入れ子タイプの add, replace 52
3.3.9 refine クラスによる仕様の変更 54
3.3.10 分割コンパイル 55

第 4 章　提案言語の実装　57
4.1 JastAdd 57

4.1.1 属性文法 57
4.1.2 JastAddJ 58
4.2 言語の実装 58
4.2.1 文法解析 58
4.2.2 クラスへの refine クラス情報の登録 59
4.2.3 refine クラスによる型階層の変更 60
4.2.4 refine クラスからのメンバの add と replace 60
4.2.5 条件付きメソッドの add 処理 63
4.2.6 client に関する処理 65
4.2.7 クラスファイル生成 73

第 5 章 実験・議論とまとめ 74
5.1 実験 74
5.2 AspectJ との比較 74
5.3 まとめ 79

第1章　はじめに

オブジェクト指向プログラミング (OOP) はプログラムをモジュールという単位で分離して構成するための技術である。しかし OOP では、デバッグや例外処理のプログラムが複数のモジュールに散在する場合がある。複数の箇所に記述されるような処理を横断的関心事と呼ぶ。横断的関心事は往々にしてモジュールとして独立させることが困難かあるいは不可能である場合が多い。

例えばデバッグ用のログ出力コードは、それを呼び出したい全ての箇所で呼出し命令が記述されていなければならない。そのような場合、同じコードがプログラムのあらゆる場所に散らばることになり、修正や除去に手間がかかる。プログラマが全ての箇所を把握していなければ、修正漏れによる人為的バグが混入される危険性がある。

既存の研究として、アスペクト指向プログラミング (AOP)[12] が提唱されている。AOP は OOP にアスペクトという概念を取り入れたプログラミングである。AOP は「ある関数 A を呼び出すとき」といったような、ジョインポイントと呼ばれる概念を提案している。AOP 言語ではプログラム中のあらゆる「タイミング」（例えば、変数 B を参照するとき・オブジェクト C を生成するとき、など）をポイントカットと呼ばれる記述をすることで選別し、そのタイミングに対し任意の処理を挿入する。挿入する処理をアドバイスと呼び、横断的関心事を挿入することで AOP は上記の問題を解決する。

そこで我々は、AOP とは異なるアプローチとして、拡張 OOP 言語を提案する。拡張 OOP 言語は AOP 言語と同じように、横断的関心事を扱い易くするが、AOP よりも技術習得のコストが低いという利点がある。なぜなら AOP 言語がジョインポイントやポイントカットといった概念を軸として機構や構文を提案しているのに対し、拡張 OOP 言語はあくまで従来の Java をベースとした概念拡張しか取り入れていないからである。

拡張 OOP 言語は refine クラス、条件付きメソッド、client 参照を従来の Java に取り入れる。これらの機能により、横断的関心事の容易な追加・削除・分離が可能になり、さらにこの言語は散在していた関心事を一箇所にまとめて記述することができる。言語実装には、オープンソースの Java ベースコンパイラ作成ツール JastAdd[4][7][6][9][8] を用いた。

本稿の第 2 章では、従来の OOP の問題点と、提案言語に関する既存技術について述べる。また、第 3 章では、提案する拡張 OOP 言語の概要と仕様を、第 4 章では、同言語の実装について、第 5 章では、代表的な AOP 言語である AspectJ[1] と差異を議論し、さらに本論文をまとめる。

第2章　OOPの問題点と既存技術

この章では、横断的関心事を扱う場合における OOP の問題点について記述する。また、その解決方法として代表的な AOP である AspectJ について述べ、その他の関連研究についても陳述する。

2.1　OOPの問題点

オブジェクト指向とはデータに観点をおいて、プログラムを個々のモジュールに分割する考え方である。それぞれのモジュールは求められている要件に沿うような簡潔なプログラムによって構成されており、カプセル化や再利用という点で他のプログラミング言語より優れている。

しかし一般的なプログラミングにおいて、全てのモジュールを簡潔に記述することは難しい。その理由の一つとして、ある種の処理が複数のオブジェクトに分散してしまうことが挙げられる。代表的なものとしてイベント通知やロギングがある。例えば、以下のようなプログラムを考える。

User クラス

```
1 public class User {
2   private String name = "";
3   public String getName() { return name; }
4   public void setName(String s) { name = s; }
5   public void initName() { setName("default"); }
6 }
```

User クラスは識別子としてメンバフィールドの name を持つ。name 自体は private なメンバであり、外部のクラスからアクセスするには getter あるいは setter を介さなければならない。また、この name を初期化するための initializer を User クラスは定義している。

2.1.1　関心事の散在

このときプログラマが、initName() が User 以外にどのクラスからどの順番で呼ばれているかを知りたいと考えたとする。この場合、ログ出

力をするようにソースを修正するのが最も単純な方法である。例えば、initName の呼び出しに新たな引数として呼び出し元 (caller) の情報を渡し、initName 内でそれを出力するようにメソッド定義を書き換える、というような方法が挙げられる。

initName を修正

```
1 public class User {
2   public void initName(Object caller) {
3     System.out.println(caller.toString() +
4                        "calls User.initName(Object)");
5     setName("default");
6   }
7 }
```

このとき initName 自体の定義が変わっているため、ソースコード上に散在する全ての initName 呼び出しを修正しなければならない。

initName 呼び出しを修正

```
1 public class Sample {
2   public void func(User user) {
3     ...
4     user.initName(this);
5     ...
6     user.initName(this);
7     ...
8   }
9 }
```

このような処理は、initName 呼び出しの数だけ修正が必要となってしまい、プログラマにとってはあまり望ましくない。このログ出力が不要に、あるいは変更が必要になった場合、initName に関する処理を再び修正しなければならない。修正が行き届いていなければ、引数の数が合わないといったことが原因でコンパイルエラーが発生するだろう。

もう一つの修正方法として、initName 自体はそのままで、個々の initName 呼び出しにおいてログを出力するように修正するという手段が考えられる。

initName 呼び出し前にログ出力

```
1 public class Sample {
2   public void func(User user) {
3     ...
4     System.out.println(this.toString() +
```

```
5                         "calls User.initName()");
6       user.initName();
7       ...
8       System.out.println(this.toString() +
9                         "calls User.initName()");
10      user.initName();
11      ...
12    }
13  }
```

しかしこの方法もまた、initName 呼び出しの数だけ修正が必要であることに変わりはない。しかもこの場合、修正が行き届いていない場合でもコンパイルエラーが発生しないため、不正確な情報を生み出してしまう恐れがある。

上記のログ出力に関する処理は、ソースコード上の多くの場所に散在してしまう可能性がある。OOP ではこのような処理を一箇所にまとめモジュール化する技術は存在しない。

2.1.2　異なる関心事の混在

モジュールを簡潔に記述することが困難である理由は他にもある。例えば、上記の User クラスにおいて、name がどのような状態のときに getName が行われるかを調べ、プログラムの終了直前にまとめてファイル出力をさせたい場合を考える。このとき、getName 単体では情報を保存させておくことが不可能なため、新たなメンバフィールドとファイル出力用のプログラムが必要となる。

getName の修正及びメンバの追加

```
1   public class User {
2     private static ArrayList logList = new ArrayList();
3     public String getName() {
4       logList.add(name.trim());
5       return name;  // 本来の処理
6     }
7     public void printLog() { /* ファイル出力 */ }
8     }
9   }
```

修正は User クラス単体及び printLog 呼び出しの箇所のみであるが、getName に関する本来の処理が埋もれてしまうという欠点がある。ログ

出力は主にデバッグとして利用されるので、除去の際にはどのような修正を施したかを覚えていなければならない。例えばこの例では

- メンバフィールドの logList を追加
- getName 内部で logList にデータ保存の処理を追加
- メンバメソッドの printLog を追加
- printLog 呼び出しを追加

という４つの修正箇所を覚えていなければならない。また、デバッグ処理の除去時には、getName の本来の要求部分のプログラムを誤って除去しないように注意しなければならない。

混在の例は他にもある。例えば setName は引数として String 型の参照を渡されるように定義されているが、その値については一切のチェックが行われていない。NullPointerException の発生を懸念したプログラマがそのチェックコードを書き加えたとき、setName は以下のように修正されるだろう。

setName の修正

```
1  public class User {
2    public void setName(String s) {
3      if(s == null)
4        System.out.println("args is null");
5      else
6        name = s; // 本来の処理
7    }
8  }
```

また、さらに「name は 3 文字以上でなければならない」といった拡張が必要となった場合、setName は次のようなプログラムになる。

setName の更なる修正

```
1  public class User {
2    public void setName(String s) {
3      if(s == null)
4        System.out.println("args is null");
5      else if(s.length() < 3)
6        System.out.println("new name is too short");
7      else
8        name = s; // 本来の処理
9    }
10 }
```

setName の本来の役割は name への値の代入であるが、例外処理等の処理により本来の役割に該当する処理がどこにあるかが判別し難くなる。これらの例のように横断的関心事の散在や混在によってプログラムの修正やモジュール化は困難になる。

また、これらの他にも横断的関心事としては

- 同期処理
- メモリ管理
- セキュリティチェック

等の処理が挙げられる。いずれもソースコード上に散在しやすく、モジュールとして分離独立させることは難しい。

2.2　求められる技術

上記のような問題点を解決するためには、本来の処理と横断的関心事の分離が不可欠である。これを満たすことで、横断的関心事の挿入や除去が容易になり、それに伴い誤って本来の処理を書き換えてしまう危険性がなくなる。また、分離した関心事をモジュール毎に一箇所にまとめる技術も求められる。これが満たされているならば、修正の必要があるプログラムが横断的関心事であっても修正箇所が少なくて済む。

2.3　関連研究

横断的関心事を扱う技術として AOP が提唱されているが、本研究では AOP と異なる視点から横断的関心事を扱うことを試みる。本研究は、MultiJava で提案された OpenClass の概念や JPred が提案した Predicate-Dispatch の概念を取り入れた新たな言語を提案する。ここでは、それらの概念の元になった研究について説明する。

2.3.1　AOP

AOP は、横断的関心事をモジュール化する技術である。AOP は、プログラム上にはあらゆる「タイミング」が存在し、そのタイミングを選択し処理を挿入することで横断的関心事をまとめる。タイミングには「関数 A が呼び出されたとき」や「オブジェクト B が生成されたとき」というように、様々な種類のものがある。これらを AOP ではジョインポイント (joinpoint) と呼ぶ。代表的な AOP 言語としては AspectJ が提案されてい

る。AspectJ は Java にアスペクト指向を取り入れた言語である。AspectJ は call や execution といった名前のポイントカット (pointcut) 指定子を記述することで目的のジョインポイントを選択する。さらに before や after といった名前のアドバイス (advice) とポイントカットを組み合わせることで、選択されたジョインポイントに処理を挿入 (weave) する。AspectJ のポイントカット指定子には例として以下のようなものが存在する。

call メソッドを呼び出すときを選択

set フィールドの値に書き込むときを選択

get フィールドの値を読み込むときを選択

initialization クラスのインスタンスを生成するときを選択

execution メソッドを実行するときを選択

handler 例外ハンドラを実行するときを選択

staticinitialization クラスの静的初期化子を実行するときを選択

within 指定したクラスまたはパッケージ内にある joinpoint を選択

withincode 指定したメソッド内にある joinpoint を選択

cflow 指定した制御フローの中にある joinpoint を選択

cflowbelow 指定した制御フローの中にある joinpoint を選択。ただし、指定された演算自体は含まない

if 指定した条件を満たしている状態の joinpoint を選択

本研究は、AOP とは異なるアプローチをおこなう。AOP はジョインポイントやアスペクトという概念を提唱するが、本研究は従来の Java の拡張という視点で横断的関心事を扱う。

2.3.2 MultiJava

MultiJava[3] は Java 言語に OpenClass と multiple dispatch を取り入れた言語である。OpenClass は外部からも機能が追加可能なクラスのことである。multiple dispatch とは、従来の method dispatch を拡張したもので、関数呼び出しの引数の型により dispatch の結果を選択可能にする。これにより煩雑な if-else 文の混入を避けられる。multiple dispatch は Visitor パターン等に役立つ。

本研究は MultiJava と同じく OpenClass の機能を取り入れる。また、method dispatch の更なる拡張とグローバルコンテキストの参照を取り入れることで既存研究との差異化を図る。

2.3.3　JPred

Jpred[13] は Java 言語に PredicateDispatch を取り入れた言語である。PredicateDispatch とは、受け渡されたメッセージの内容によってメソッドの実装を決定するメカニズムである。従来の method dispatch は、メソッドのシグネチャと、(callee の) インスタンスによって結果が決定されるが、PredicateDispatch は関数呼び出しに追加情報を付加させることで細かい dispatch 条件を付けられる技術である。追加情報とは例えば引数の実行時の型であったり、callee 側のメンバ変数の値であったりする。

本研究では JPred と同じく PredicateDispatch の機能を取り入れる。JPred の違いは、本研究では追加情報としてグローバルコンテキストの値を付加させることも可能にする。これにより dispatch の幅は広がり、関心事の分離をより柔軟におこなえるようにする。

第3章　拡張OOPの提案

本章では、前章で示したOOPの問題点を解決するための拡張言語を提案する。それにより問題がどのように解決されるかを具体例を交えながら示し、さらに提案言語の利点について言及する。また、提案言語の仕様についても本章にて述べる。

3.1　拡張言語による問題の解決

前章で示した問題点を解決するために、本研究ではJavaプログラミングに新たな二つの文法を導入する。1つは**refines節**、そしてもう一つは**when節**である。refines節はOpenClassの概念を、when節はPredicate-Dispatchの概念を利用するために用いる。また、本言語ではwhen節において**client**という変数を参照できるようにする。これは外部コンテキストを扱うために用いる。

3.1.1　OpenClass: refine節の導入

refines節は以下のように用いる。

refines節

```
class Ref refines User { /* ... */ }
```

このとき、Refクラスは Userクラスを refine対象として選択する。本研究では『refines』の直前に記述されたクラス（ここではRef）を refineクラスと呼ぶ。refineクラスは通常のクラスとは異なる挙動を示す。以下に前章で記述したUserクラスと、新たな refineクラスの例を示す。

Userクラス

```
public class User {
  private String name = "";
  public String getName() { return name; }
  public void setName(String s) { name = s; }
  public void initName() { setName("default"); }
}
```

Ref1 クラスの例

```
class Ref1 refines User {
  private static ArrayList logList = new ArrayList();
  public void printLog() { /* ファイル出力 */ }
  public String getName() {
    logList.add(name.trim());
    return name;
  }
}
```

refine クラスに書かれているメンバは全て、その refine クラスが持つ refine 対象クラスに定義されているものとして扱われる。上記の例では、logList 及び printLog は共に Ref クラスに記述されているが、これらは User クラスのメンバとして定義される。メンバへのアクセスも同様で、例えばソース上に Ref1.logList というアクセスがあった場合、コンパイラはエラーを出して終了する。正しくは User.logList である。このように refine 対象クラスに新たなメンバを定義するとき、refine クラスはメンバを **add する**と呼ぶ。add されたメンバはプログラム上のどこからであってもアクセスすることが可能となる。

getName() に関しては、Ref1 クラスだけでなく User クラスにも定義されている。このようにメンバの名前が一致している場合、コンパイラは refine クラス側の定義を優先する。その結果、User.getName() は Ref1.getName() に上書きされる。このように refine 対象クラスの既存のメンバを再定義するとき、refine クラスはメンバを **replace する**と呼ぶ。replace されたメンバ（ここでは元々の User.getName()）は原則としてプログラム上のどこからも呼び出すことが不可能となる。全ての User.getName() 呼び出しは Ref1 に定義されている getName() を代わりに呼び出す。ただし add 時と同様に、ソース上に Ref1.getName() を指定するようなアクセスを記述している場合はコンパイルエラーである。元々の User.getName() を呼び出したい場合には、特例として proceed 呼び出しを用いて実現することができる。proceed 呼び出しについては後述する。

refines 節を利用することで前章で挙げた問題点のうち 1 つが解決される。上記の Ref1 クラスは User クラスに logList を持たせ、getName() 呼び出し時にログを記録するよう拡張し、さらにそれを出力させる printLog() を提供している。Ref1 クラスは、User クラスに前章で示したものと同一の機能を持たせることを可能にする。

従来の Java による拡張

```
public class User {
  private static ArrayList logList = new ArrayList();
```

```
  public String getName() {
    logList.add(name.trim());
    return name;
  }
  public void printLog() { /* ファイル出力 */ }
  }
}
```

本言語は従来の Java と異なり、拡張やデバッグ部分が refine クラスとして分離できる。refine クラスは OpenClass を実現するものであり、外部のクラスを直接修正することなく拡張可能にする。プログラマがデバッグ部分を除去する際には、Ref1 クラス自体を除去することでそれを可能にする。また、printLog の呼び出し箇所についても refine クラスを利用すれば直接の修正は不要である。

3.1.2　PredicateDispatch: when 節の導入

when 節は refine クラスにおいてのみ使用することができる。例として以下のようなプログラムを考える。

when 節の使用例

```
class Ref refines User {
  public void setName(String s) when s == null {
    System.out.println("args is null");
  }
}
```

Ref クラスは refine 対象として前節と同じ User クラスを選択している。User クラスには既に setName(String) が存在している。Ref.setName(String) はメソッドのシグネチャ直後に「when s == null」という記述があり、これは「引数が null を指すとき、User クラスの setName(String) の代わりにこのメソッドが実行される」という意味を表す。逆に引数が null を表さないときは、User クラスに元々記述されている setName(String) が実行される。

このような when 節が付随したメソッドを**条件付きメソッド (Conditional Method)** と呼ぶ。条件付きメソッドは refine クラスにおいてのみ定義可能なメソッドである。また、条件付きメソッドのシグネチャは、refine 対象のクラスが持つメソッドの内いずれか 1 つと一致していなければならない。シグネチャの一致とは、メソッド名が同一であり、かつ引数

の数及びそれぞれの型とその順序が一致することである。条件付きメソッドのシグネチャが上記の条件を満たしているとき、refine クラスは refine 対象のクラスにその条件付きメソッドを **add する**と呼ぶ。また、条件付きメソッドと同一のシグネチャを持つメソッドを、その条件付きメソッドの **dispatch 対象**と呼ぶ。

「when」の直後には、任意の boolean 式を記述することができる。add された条件付きメソッドは、自身と同じシグネチャを持つメソッドが呼び出されたタイミングでその boolean 式を評価する。そして式が true を返すならば、本来呼び出される予定のメソッドの代わりにその条件付きメソッドが実行される。このとき、本来呼び出される予定のメソッドは実行されないままである。逆に boolean 式が false を返すときは、本来のメソッドがそのまま実行され、条件付きメソッドは何もしない。

ある一つのメソッドに対して、同じシグネチャを持つ複数の条件付きメソッドを add させることも可能である。システムは、そのメソッドが呼び出された時点でそれらの条件付きメソッドの持つ when 節の boolean 式を順に評価し、最初に true を返す条件付きメソッドを実行する。全ての条件付きメソッドが false となる場合、オリジナルのメソッドがそのまま呼び出される。

条件付きメソッドを用いることで前章の問題点の 1 つを解決できる。以下に例を示す。

Ref2 クラス

```
class Ref2 refines User {
  public void setName(String s) when s == null {
    System.out.println("args is null");
  }
  public void setName(String s) when s.length() < 3 {
    System.out.println("new name is too short");
  }
}
```

User クラスには setName(String) が存在するので、これらの条件付きメソッドは両方とも User クラスに add される。User.setName(String) が呼び出された時点で when 節の評価が開始され、引数が null であるなら最初の条件付きメソッドが実行される。引数が null でないのなら、その String オブジェクトの文字数が 3 未満であるかどうかをチェックし、それが true であるのなら 2 番目の条件付きメソッドが実行される。文字数が 3 以上であるのなら、元から User クラスに記述されていたオリジナルの setName(String) が実行される。

従来の Java では setName(String) の拡張はそのメソッド自身の修正が必要であった。そのため本来の要求とは別に、if-else 文による例外チェック等が挿入されていた。

従来の Java による拡張

```
public class User {
  public void setName(String s) {
    if(s == null)
      System.out.println("args is null");
    else if(s.length() < 3)
      System.out.println("new name is too short");
    else
      name = s;
  }
}
```

例外チェック項目の追加、削除による条件分岐数の増減に対応するのは、従来の Java では困難であったが、本研究の提案言語では、条件付きメソッドとしてメソッド単位で分離することが可能である。また、条件分岐数が増減してもオリジナルのソースを一切修正する必要は無い。この例では、User.setName(String) 自体は修正の必要がなく、「名前を代入する」という本来の要求仕様を満たしていることのチェックも行いやすい。

3.1.3　外部コンテキストの参照: client 変数の導入

提案言語では、外部コンテキストとして、条件付きメソッドにのみ client 変数へのアクセスを許可させる。client が指すものは、呼び出し側 (caller) のインスタンスである。client へのアクセスは、when 節の boolean 式内部あるいはメソッドボディ内部でのみ可能である。

client 変数の使用例

```
class Ref refines Sample {
  public void f() when client instanceof User {
    System.out.println(client.toString());
  }
}
```

上記のプログラムは、f() を dispatch 対象とする条件付きメソッドの例である。when 節には「client instanceof User」という式が記述されている。これは「呼び出し側のインスタンスが User であるかどうか」という条件を意味する。この式が true を返すとき、つまりこの Sample.f() が User

クラスから呼ばれているならば、プログラムはその呼び出し元のインスタンスの toString() を表示する。逆に false を返すならば、この条件付きメソッドは何もしない。

client 変数の使用にはいくつか制限がある。when 節の中においては、client は「client instanceof 〜」という条件文中においてのみ caller を指すという特別な意味を持つ。それ以外の記述では client は通常の変数と同じようにしか扱われない。また、上記の例ではメソッドボディにおいても client へのアクセスをおこなっているが、メソッドボディ内部で client へのアクセスをおこなうには when 節の条件文の中に「client instanceof 〜」を含んでいなければならない。

client 変数を用いることで、前章の問題点を解決することが可能となる。例えば、以下のようなプログラムを考える。

Ref3 クラス

```
class Ref3 refines User {
  public void initName() when !(client instanceof User) {
    System.out.println(client.toString() +
                       " calls User.initName()");
    proceed();
  }
}
```

上記のような条件付きメソッドを User クラスに add することで、全ての initName() 呼び出し箇所の情報を取得することが可能となる。when 節の条件文は「呼び出し元のインスタンスが User 以外である」を表し（NOT が付随していることに注意）、さらにそのときは client.toString() を表示するようなプログラムになっている。また、proceed() は、オリジナルの initName() を呼び出すための特別なメソッドである。

Ref3 クラスにより、initName() の呼び出しに関する一切のソースを変更せずにデバッグが可能となる。従来の Java では、このような修正には initName() の全ての呼び出し箇所を網羅する必要があった。本研究では外部コンテキストの参照機能を用いることが可能なため、initName() の呼び出し元を予め網羅しておく必要は無い。

3.2　提案言語の利点

上記の refine クラスを用いることで前章の問題点が解決されることが分かった。これによりプログラマはモジュール内のデバッグコードや例外処理といった関心事を、本来要求されていた関心事と分離して記述すること

ができる。また、refine クラスは OpenClass の機能を持つため、横断的関心事の挿入、削除をおこない易い。これはオリジナルのソースを修正することなく関心事をクラス単位で分離することが可能なためである。条件付きメソドは method dispatch をより詳細な条件に合わせられるように言語を拡張している。さらに外部コンテキストである caller への参照を取得できるため、callee の状態だけでなく caller の状態を条件分岐の要素として選択することを可能にした。caller の情報は従来の Java では引数として渡すことでしか取得できなかったため、そのソースは caller 及び callee の両方において引数を渡すことを明示しなければならなかった。本言語ではその必要がなくなったため、修正や拡張に対する処理が 1 箇所で済むようになっている。

refine クラスは AspectJ における aspect に近い意味を持つ。しかし本研究の提案言語は AspectJ のような複雑な概念を必要としない。AspectJ はポイントカットやアドバイスといった概念を理解することが必要な言語である。また、文法や単語も本言語と比べて多数存在している。本言語における新文法は refines 節及び when 節のみであり、特別な意味を持つ単語は refines, when, client, proceed の 4 種類のみである。

3.3　仕様

本言語の仕様について記載する。本言語では従来の Java に加えて refine クラスと条件付きメソッドを導入している。ここではそれらの仕様に加え、メンバの add や replace、client 参照や proceed 等についても説明する。

3.3.1　refines 節: refine クラス

refine クラスは、refine 対象のタイプに対し、メンバの add や replace が可能である。refine 対象のタイプはクラスあるいはインターフェイスが該当する。処理系は、refine クラスに定義されたメンバを全て refine 対象のタイプに定義されているものとして扱う。また、refine クラスは、refine 対象のタイプの super type や interface を変更することも可能である。

refines 節の文法

refines はクラス宣言のヘッダにのみ用いることができ、ヘッダに refines を含むクラスは refine クラスとなる。refine クラスは下の文法を満たしていなければならない。

```
refine-class = class-modifier? "class" IDENTIFIER "refines"
               type-access extends-sentence?
               implements-sentence? "{" body-decl* "}"
```

refine クラスのアクセス修飾子は意味を持たない。例えば、以下の refine クラスの定義はいずれも同じである。

```
public class R refines T { ... }
private class R refines T { ... }
class R refines T { ... }
```

アクセス修飾子が意味を持たない理由は、refines クラスに定義したメンバが全て refine 対象のクラスのメンバとして扱われるからである。例えば以下のような refine クラス R とメンバ変数 i があるとき

```
private class R refines T {
  public static int i = 10;
}
```

処理系においてメンバ変数 i は R ではなく T に定義されたものとして振舞う。本システムでは i へのアクセスは R.i ではなく T.i とするのが正しい。そのため R 自身のアクセス修飾子は意味を持たない。refine クラスから add されたメンバは、ソースコード上のどこからでも元々refine 対象のクラスに宣言されていたかのように見える。T に i という名前のメンバ変数が存在しなくとも、R の処理により T.i へのアクセスが可能となる。例えば以下のプログラムはコンパイルに成功する。

```
class R refines T {
  public static int i = 10;
}
class T {
  public static int j = 20;
}
class Sample {
  int f() { return T.i + T.j; } // i は R から T へ add される
}
```

refine 対象の identifier には単純名、限定名を用いることができる。refine 対象となるタイプはクラスあるいはインターフェイスのいずれでも良い。また、refine 対象タイプを指定した直後に extends 節あるいは implements

節を用いることで、refine 対象タイプの super type や interface を変更することができる。以下に例を示す。

```
class R refines T extends S implements I {}
class R2 refines p.T2 extends S2 implements I2 {}
```

R は T の super type を S に変更し、T の interface のリストに I を追加する。同様に R2 は p というパッケージに存在する T2 の super type を S2 に変更し、p.T2 の interface のリストに I2 を追加する。オリジナルの T や p.T2 に super type が既に宣言されている場合、**refine クラスに宣言されている super type が優先**される。interface は super type とは異なり、「変更」ではなく「追加」である。オリジナルの T や p.T2 に既に interface が宣言されている場合、refine クラスはその interface に加えて新たな interface を追加させることができる。例えば以下のソースをコンパイルした場合、T の super type は S1 になり、interface は I1 と I2 の 2 つになる。

```
class R refines T extends S1 {}
class R2 refines T implements I1 {}
class T extends S2 implements I2 {}
```

T は単純名あるいは限定名であり、その名前が示すタイプが一意に定まらなければコンパイルエラーとなる。しかし refine クラスを用いて多くのクラスに一様の処理を追加したいような場合、同様の処理をクラスの数だけ記述する必要があるため煩わしい。本言語ではそれを避けるため、ワイルドカードによる一括処理をおこなうことができる。refine 対象を複数のタイプにしたい場合、refines 直後の identifier 部にワイルドカード*を含めればよい。以下に例を示す。

```
class R0 refines * { ... }
class R1 refines p1.abc* { ... }
class R2 refines *.*xyz { ... }
class R3 refines *.*.*hij* { ... }
class R4 refines p2.** { ... }
```

*は任意の文字列（空でも良い）を意味する。ただしピリオドは*の対象から除く。**は**ピリオドを含めた**任意の文字列（空でも良い）を意味する。

なお、refines 対象の宣言が単純名でも限定名でもない場合（ワイルドカードを含んでいる場合）は、nested type 及び refine クラスは自動的に対象として選択されなくなる。

refine クラスの意味解析

以下は、R を refine クラス、T を R の refine 対象タイプの記述とする。

```
class R refines T { }
```

このとき、R 及び T に関する意味上の制約は以下の通りである。なお、T は単純名あるいは限定名であると仮定する。

- T の対象となり得るタイプは以下の 5 つ
  1. クラス
  2. 抽象クラス
  3. 入れ子クラス
  4. インターフェイス
  5. 入れ子インターフェイス
- refines 解析時に error が発生する条件は以下の通り
  1. R 自身がインターフェイス、抽象クラス、入れ子クラスとして宣言されている場合
  2. T の対象が、上に示した 5 つのタイプに該当しないとき
  3. T の対象が、R 自身あるいは別の refine クラスである場合
  4. T の対象が、refine クラスに宣言されている入れ子タイプのとき場合
  5. T の対象の宣言あるいはソースコードが存在しない場合
  6. T の対象が 2 つ以上存在し、そのいずれを選択しているかが曖昧である場合

上記のエラーが 1 つでも発生した場合、処理系はコンパイルエラーを出して終了する。

T にワイルドカードが含まれている場合は上記と異なり、以下のような制約となる。

- refines 解析時に error が発生する条件は以下の通り
  1. R 自身がインターフェイス、抽象クラス、入れ子クラスとして宣言されている場合
- refines 解析時に warning が発生する条件は以下の通り
  1. T の記述条件に当てはまるタイプが一つも存在しないとき

2. T の記述条件に当てはまるタイプが存在するが、そのいずれにもソースコードがないとき

上記のエラーが 1 つでも発生した場合、処理系はコンパイルエラーを出して終了する。warning が発生した場合、コンパイラはその refine クラスを存在しないものとして処理を続行する。

また、T の記述条件に当てはまるタイプが複数存在している場合で、そのうちの一部のタイプのソースコードをコンパイラが発見できないとき、そのタイプは R の refine 対象から外される。例えば以下のプログラムと、ソースコードの無いクラスファイル T3 がある場合、

```
class R refines T* { }
class T1 { }
class T2 { }
```

R の refine 対象タイプとして選択されるのは T1, T2 の 2 種類である。T3 はソースファイルが存在しないため、refine 対象とはならない。

refine クラスの可視範囲は、既存の Java クラスのそれと異なり、refine 対象のタイプにおける可視範囲と同等の制約を受ける。例えば以下のソースでは 2 つのメンバ変数 x, y のいずれも R に宣言されていないが、コンパイルエラーとはならない。

```
class R refines T {
  int f() { return x + y; } // not compile error
}
class T {
  private int x = 0;
}
class R2 refines T {
  private int y = 1;
}
```

メンバ変数 y は R2 から T に add される。このときプログラム上では T.y というメンバ変数が存在するものとして扱われる。add されたメンバはソース上のどこからでも参照可能であるため（ただしアクセス指定子の影響は別途受ける）、R 上からであっても y を参照できる。R が宣言した f() メソッドは T に add され T のメソッドとして振舞うため、private なメンバである x, y の両方にアクセスを許可されるため、コンパイルエラーにはならない。

refine クラスオブジェクトは不生成

refine クラスは実体を作らない。なぜならば refine クラスに定義されたメンバは全て refine 対象のクラスに定義されたものとして変更されるからである。refine 対象のクラスとして refine クラスを選択することは不可であるため、refine クラスがメンバを保持し続けることは無い。例えば以下のようなプログラムを考える。

```
class T1 { }
class T2 { }
class R refines T* {
  public static int x = 1;
}
```

このプログラムを処理系が扱ったとき、一連の処理が終わった後これは次のようなプログラムと見なされる。

```
class T1 {
  public static int x = 1;
}
class T2 {
  public static int x = 1;
}
class R { }
```

R が持つメンバは全て refine 対象のクラスへと移行し終えているため、R のオブジェクトを生成することに意味は無い。本研究では、refine クラスのオブジェクトを生成するようなコードの記述は禁止している。また、コンパイル後に refine クラスのクラスファイルが生成されることも無い。refine クラスはソースコード上にのみ存在し、コンパイル中のあるタイミングからは元々存在しなかったかのように振舞う。

refine クラスの優先順位

複数の refine クラスが持つ refine 対象が同じタイプであるとき、それらの refine クラスには必ず優先順位が設けられる。なお、2 つの R に対しそれらが同じタイプを refine 対象にしていると判定する条件は以下の通りである。

- それぞれの R が持つ T の記述を t1 と t2 とおく

- それぞれの T が互いに単純名あるいは限定名であり、その対象が一致するとき
- それぞれの T のうち一方だけ (t1) がワイルドカードを含み、そうでない方 (t2) の対象タイプが t1 の示すタイプ群の中に含まれているとき
- それぞれの T が両方ともワイルドカードを含み、両者の対象となるタイプ群の中に、１つ以上同じタイプが重複するとき

例えばパッケージ p1 の中に A というクラスがあるとき、以下の refine クラスは全て同じタイプを refine 対象にしていると判断する。

```
class R0 refines p1.A { ... }
class R1 refines p1.* { ... }
class R2 refines *.A* { ... }
class R3 refines *.* { ... }
class R4 refines ** { ... }
```

また、限定名でなく単純名でも refine 対象のタイプを選択可能なため、次のような refine クラスも上記の refine クラス群と同じタイプ（p1.A）を refine 対象に選択していると判断する。

```
package p1;
class R5 refines A { ... }
```

同じタイプを refine 対象にしている refine クラス間における優先順位は以下のようにして定める。T の限定名のあるいはワイルドカードの有無は優先順位には一切無関係である。

- 任意の T において、T を refines の対象とする R が x(x > 0) 個ある場合、各 R の『パッケージ名. クラス名』を辞書順にした際に、最も値の小さいものから順に R0, R1, ... , Rn (n = x-1) とおき、その優先順位は R0, R1, ... , Rn と定める
- 『R1 が R2 より T に対する優先度が高い』とは、R1 と R2 の両方が同じ T を refine の対象として選択しており、かつ上記の優先順位に従い、R1 のほうが優先されることを指す
- R1 と R2 が同じ refine クラスであるか、または refine 対象となるタイプが１つも一致しない場合、R1 と R2 の優先度は等しいものとする

上記の例では、最も A に対する優先度が高い refine クラスは R5 である。なぜならば R5 の限定名は「p1.R5」であり、これは他の refine クラスよりも辞書順にした際に値が小さくなるからである。R5 の次に A に対する優先度が高いのは R0 であり、以下 R1, R2, R3, R4 と続く。

3.3.2　refine クラスによる型階層の変更

refine クラスで明示された supertype や interface は、refine 対象のタイプの supertype あるいは interface となる。ただし refine 対象のタイプがクラスか、あるいはインターフェイスであるかによって制約は異なる。

refine 対象がクラスの場合

以下のような記述を例に挙げる。ここで T はクラスを表す。

```
class R refines T extends S { }
```

refine クラスのヘッダ部分に extends 節がある場合、refine 対象のクラス T の super class を変更することができる。refine クラスによる super class 変更には以下のような制約がある。

- 次の『条件』を満たせば S をクラス T の super class として変更
  - S の対象が一意に定まる
  - S がソースコードの有無に関わらず定義されている
  - S がクラスあるいは抽象クラスである
  - S が refine クラスでない
  - S が refine クラスで定義された入れ子クラスでない
  - T が S を継承した場合に、型階層に循環が発生しない
- super class 変更に関する error 発生条件は以下の通り
  - 上記の条件のうち 1 つ以上を満たしていない場合
- super class 変更に関する warning 発生条件は以下の通り
  - R より T に対する優先度の高い refine クラスのうちいずれかが T の super class を変更している場合

error が発生した場合、処理系はそこでコンパイルエラーを通知して終了する。warning が発生した場合、R による T の super class の変更はおこなわれず処理が続行される。warning は次のような場合に発生する。

```
class T extends S { }
class R1 refines T extends S1 { }
class R2 refines T extends S2 { } // warning
```

この場合、R1 のほうが R2 よりも T に対する優先度が高いため、T の super class は S から S1 へと変更される。また、処理系は R2 のヘッダ部分を指して warning を発する。R1 の優先により、R2 による T の super class 変更は成されないためである。

次にクラスの interface 追加について説明する。

```
class R refines T implements I { }
```

refine クラスのヘッダ部分に implements 節がある場合、refine 対象のクラス T の interface を追加することができる。refine クラスによる interface 追加の制約は以下の通りである。

- 次の『条件』を満たせば I をクラス T の interface として追加
  - I の対象が一意に定まる
  - I がソースコードの有無に関わらず定義されている
  - I がインターフェイスである
  - I が refine クラスで定義された入れ子インターフェイスでない
  - T が I を実装していない
  - R より T に対する優先度の高い refine クラスのいずれも T の interface に I を追加していない
- interface 追加に関する error 発生条件は以下の通り
  - 上記の条件のうち 1 つ以上を満たしていない場合

error が発生した場合、処理系はそこでコンパイルエラーを通知して終了する。

refine 対象がインターフェイスの場合

以下のような記述を例に挙げる。ここで T はインターフェイスを表す。

```
class R refines T extends I { }
```

refine クラスのヘッダ部分に extends 節がある場合、refine 対象のインターフェイス T の super interface を追加することができる。refine クラスによる super interface 追加には以下のような制約がある。

- 次の『条件』を満たせば I をインターフェイス T の super interface として追加
  - I の対象が一意に定まる
  - I がソースコードの有無に関わらず定義されている
  - I がインターフェイスである
  - I が refine クラスで定義された入れ子インターフェイスでない
  - T が I を実装していない
  - R より T に対する優先度の高い refine クラスのいずれも T の interface に I を追加していない
- interface 追加に関する error 発生条件は以下の通り
  - 上記の条件のうち 1 つ以上を満たしていない場合

error が発生した場合、処理系はそこでコンパイルエラーを通知して終了する。

型階層変更後の整合性チェック

ソースコード上の全ての refine クラスによる型階層変更が終了した後、処理系は階層全体の整合性をチェックする。例えばそれは循環の有無や、抽象メソッドの実体化の有無、継承による可視範囲のチェック等である。以下に例を示す。

```
abstract class Abs {
  int x = 1;
  int f();
}
class T { }
class R refines T extends Abs {
  int f() { return x; }
}
```

上記のプログラムでは、R がクラス T の super class を抽象クラス Abs に変更する。Abs には抽象メソッドである f() が存在し、Abs を継承する

実体クラスは全て f() を実装しなければならない。このとき T もまた、f() を実装する必要が生じる。上記は R によって f() は実装されているためコンパイルエラーにはならない。

また、これらの整合性のチェックは refine に関係の無いクラスにおいてもおこなわれる。

```
class S {
  int f() { return 0; }
}
class U extends T {
  int g() { return f(); } // not compile error
}
class T { }
class R refines T extends S { }
```

上のプログラムでは R によって T の super class は S へと変更される。このとき型階層は、U の親クラスとして T を、T の親クラスとして S をおいた状態となる。U と S の間には継承関係が発生するため、U の可視範囲には S の f() が存在することになる。そのためこのプログラムはコンパイルエラーにはならない。もし R が存在しなかった場合、U の可視範囲に f() は存在しないためこのプログラムはコンパイルエラーとなる。ここで U は refine クラスとは直接の関係は無いが、型階層の整合性のチェック対象として含まれている。

処理系は、ソースコードとして渡された全てのクラスの整合性をチェックし、Java の規則に違反しているような記述が見つかった場合、コンパイルエラーを示して終了する。

3.3.3　メソッドの add, replace

refine クラスに記述された method は、refine 対象のタイプに挿入される。R が持つ method(m とおく) を T に挿入する際に、T が m と同じシグネチャの method(m2 とおく) を持っている場合、R.m は T.m2 を replace する。このとき T.m2 はプログラム上から消去され、T.m2 を呼び出す処理は全て R.m を呼び出す処理へと変換される。T に m と同じシグネチャの method が存在しない場合、R.m は T に add され、T.m という method が生成される。この T.m はプログラム上、どこからでも参照できる。例として以下のようなプログラムが考えられる。

```
class T { }
class T2 {
```

```
  int f() { return 0; }
}
class R refines T {
  int f() { return 1; } // add
}
class R2 refines T2 {
  int f() { return 2; } // replace
}
```

上記の refine クラス R は、refine 対象クラス T にメソッド f() を add する。これにより T はメソッド f() を宣言しているのと同じように振舞う。一方で R2 は、T2 のメソッド f() を自身のメソッド f() で replace する。元々の T2.f() は上書きされ、原則としてプログラム上のどこからも呼び出されることはない。T, T2 のメソッド f() に対する呼び出しは以下のようになる。

```
class Sample {
  void f() {
    new T().f();  // return 1
    new T2().f(); // return 2
  }
}
```

メソッドを replace するかどうかの判定は、同じシグネチャかどうかによって決定される。同じシグネチャとは、return type、修飾子、throw 節、method body の有無に関係なく、名前と引数の数と型で判定される。例えば、以下のプログラムは全て同じシグネチャ「func(int)」を表す。

```
public void func(int i) { ... }
protected int func(int j);
static Object[] func(int k) throws Exception { ... }
```

refine 対象のタイプ T が他の refine クラスのそれと重複するとき、メソッドの add や replace が確実に行われるとは限らない。その判定は「refine クラスの優先順位」に従い、以下のようになる。

- T に対する優先順位に従い、それぞれの refine クラスを R0(最優先), R1, ... , Rn とおく

- 各 Rn が持つ method(M とおく) が以下の条件に当てはまるとき、T に M を add
  - (n = 0 のとき) T に M と同じシグネチャの method が無い場合
  - (n > 0 のとき) M と同じシグネチャの method を、Rn より T に対する優先度の高いどの Rx(x < n) も持たず、かつ T に M と同じシグネチャの method が無い場合
- 各 Rn が持つ method(M とおく) が以下の条件に当てはまるとき、T に M を replace
  - (n = 0 のとき) T に M と同じシグネチャの method がある場合
  - (n > 0 のとき) M と同じシグネチャの method を、Rn より T に対する優先度の高いどの Rx(x < n) も持たず、かつ T に M と同じシグネチャの method がある場合

また、メソッドの add, replace における error や warning の発生条件は以下の通りである。

- error の発生条件
  1. T の対象にインターフェイスが含まれており、かつ R が method body を持つメソッドを定義しているとき
  2. T の対象に実体クラスが含まれており、かつ R が method body を持たないメソッドを定義しているとき
- warning の発生条件
  1. M と同じシグネチャを持つ method が、同じ R 内で M よりも前方に定義されている場合
  2. M が条件を満たせず、add も replace もされなかったとき

warning が発生したとき、M は定義されていないものとして扱われる。例として以下のプログラムを考える。

```
1 class T { }
2 class R1 refines T {
3   int f() { return 1; }
4 }
5 class R2 refines T {
6   int f() { return 2; } // warning
7 }
```

上記のプログラムを処理系が扱うとき、6 行目を指して warning が発生する。T.f() メソッドは既により優先度の高い R1 によって add されているからである。本システムでは add あるいは replace されたメソッドを再び replace することはできない。このとき R2.f() は元々記述されていないものとしてコンパイルが続行される。

refine クラスの method は名前にワイルドカード*を含むことができる。そのような method は add されることはなくなり、replace するかどうかだけを判定する。引数や return type に*を含むことはできない。以下のメソッドは全て同じシグネチャであると判断される。

```
public void func(int i);
private int *(int j) { ... }
static Object[] f*(int x) { ... }
```

proceed 呼び出し

refine クラスのメンバメソッドが refine 対象クラスのメソッドを replace したとき、オリジナルのメソッドはプログラム上のどこからも呼び出すことが出来ない。しかし例外として、**proceed** 関数を用いることで、replace したメソッドからのみ呼び出すことができる。例えば、proceed は以下のように使う。

```
1  class T {
2    void f() { }
3    int g() { return 0; }
4  }
5  class R refines T {
6    void f() { proceed(); }
7    int g() { return proceed(); }
8  }
```

6 行目の f() 内における proceed() は 2 行目の f() を呼び出す。これは R.f() が T.f() を replace しているからである。同様に、7 行目の g() 内における proceed() は、3 行目の g() を呼び出す。同じ proceed() という名前のメソッドであっても replace するメソッド毎に呼び出す対象が変わる。

proceed の return type と引数の数や型はメソッド毎に異なる。以下に例を示す。

```
class T {
  int f(int i) { return i + 1; }
  String g(String s) { return s + "a"; }
}
class R refines T {
  int f(int i) { return proceed(i + 2); }
  String g(String s) { return proceed(s + "b"); }
}
```

f(int) においての proceed は引数が int 型、返り値が int 型のメソッドとして扱われる。一方で、g(String) においての proceed は引数が String 型、返り値が int 型のメソッドとして扱われる。proceed 呼び出しを含むメソッドが replace されるため、T.f() 及び T.g() 呼び出しの例は以下のようになる。

```
class Sample {
  void v() {
    new T().f(3); // return 6
    new T().g("c"); // return "cba"
  }
}
```

proceed の使用においては、以下の点に注意しなければならない。

- proceed の引数の数と型は、それを含むメソッド自身のそれらと必ず一致させなければならない。そうでなければ proceed は特別な意味を持たない
- 抽象メソッドを replace している場合、proceed は特別な意味を持たない
- add された method 内部では、proceed は特別な意味を持たない
- 特別な意味を持たない proceed は、その名前と引数に一致するような適切な method が見つからなければ compile error となる
- replace した method 内部で proceed という名前の関数を呼び出したい場合は、`this.proceed()` のように this アクセスを用いて使用するか、あるいは_proceed のように先頭に_を付けて使用する。

3.3.4　when 節: 条件付きメソッドの add

refine クラスは条件付きメソッド (conditional method) を定義できる。条件付きメソッドは boolean 型の任意の式を含む when 節を持つ。条件付きメソッドの文法は次の通りに定める。

```
conditional-method = modifiers? type IDENTIFIER "("
                     parameter-list ")" array-dims?
                     throw-sentence? "when" boolean-expr
                     "{" method-body* "}"
```

上記の文法を満たすメソッドは全て条件付きメソッドであると解釈する。例えば以下のメソッドは全て条件付きメソッドである。

```
int a(Object o) throws Exception when (o == null) { }
private static char[] b() when this.x == 0 { }
int c(int x, int y) [] when (x == 0) || (y == 0) { }
```

条件付きメソッドは refine 対象クラスのメンバのうち、自身と同じシグネチャを持つメソッドを dispatch 対象として選択する。dispatch 対象となったメソッドがプログラム上で呼び出されたとき、条件付きメソッドは自身の boolean 式を評価し、それが true を返すならば dispatch 対象のメソッドの代わりに自身が実行される。

```
class T {
  int f(int i) { return 1; }
}
class R refines T {
  int f(int i) when i < 0 { return -1; }
}
```

上の例において、クラス T に宣言された f(int) は常に 1 を返すメソッドである。そして refine クラス R は、この f(int) と同じシグネチャの条件付きメソッドを refine 対象クラス T に add する。その結果、処理系は T.f(int) が次のようなプログラムであると解釈する。

```
class T {
  int f(int i) {
    if(i < 0)
      return -1;
    else
```

```
      return 1;
  }
}
```

このように、when 節における boolean 式の評価の結果で f(int) の挙動は変化する。条件付きメソッドによる dispatch 対象メソッドの拡張はプログラム上の全ての呼び出しに対応している。

```
class Sample {
  void v() {
    new T().f(10); // return 1
    new T().f(-10); // return -1
  }
}
```

条件付きメソッドは、refine 対象クラスに dispatch 対象となるメソッドが見つからなければプログラムに何も作用しない。dispatch 対象となるメソッドを見つけるには、まずシグネチャが同じでなければならない。

super type のメソッド

dispatch 対象のメソッドは、refine 対象クラス T か、あるいは T の super type か、それら (T か、T の super type) を refine 対象とする refine クラスのいずれかに定義されていなければならない。以下に例を示す。

```
class S {
  int f(int a) { return 0; }
}
class T extends S {
  int g(int b) { return 1; }
}
class R1 refines S {
  int h(int c) { return 2; }
}
class R2 refines T {
  int i(int d) { return 3; }
}
```

２つの refine クラス R1, R2 により、クラス S, T にはそれぞれメソッドが add される。S にはメソッド f, h の２つが定義されていると見なされ、T にはメソッド g, i の２つが定義されていると見なされる。そして T は S を継承しているため、T の可視範囲には f, g, h, i の４つのメソッドが存在することになる。このとき、次の条件付きメソッドは全て T に add される。

```
class R0 refines T {
  int f(int a) when a < 0 { return 4; }
  int g(int b) when b < 0 { return 5; }
  int h(int c) when c < 0 { return 6; }
  int i(int d) when d < 0 { return 7; }
}
```

４つのメソッドの内、g, i については T あるいは、T を refine 対象として選択している R2 に定義されているため、処理系はこの g, i に関するプログラムを単純に次のように変換したものとして扱う。

```
class T {
  int g(int b) {
    if(b < 0) return 5;
    else return 1;
  }
  int i(int d) {
    if(d < 0) return 7;
    else return 3;
  }
}
```

一方で f, h の２は T ではなく、super type である S と、S を refine 対象として選択している R1 に定義されているメソッドである。refine クラス R0 はあくまで S ではなく T を refine 対象として選択しているため、R0 により S の挙動そのものを拡張させることは避けたい。そのため処理系はこの f, h に関するプログラムを次のように変換したものとして扱う。

```
class T {
  int f(int a) {
    if(a < 0) return 4;
    else return super.f(a); // return 0
```

```
  }
  int h(int c) {
    if(c < 0) return 6;
    else return super.h(c); // return 2
  }
}
```

つまり f, h において条件付きメソッドの実行条件が満たされなかった場合は、オリジナルの f, h が呼び出されなければならないため、super アクセスがおこなわれるのである。

super type のメンバは sub type に継承されるため、クラス T のメソッド f(int) は存在するものとして扱わなければならない。しかし Java では継承を許可しないメソッドが定義可能であるため、本言語においてもそのようなメソッドを拡張するには制約を設けてある。例えば、上記のメソッド f, g, h, i が全て final 宣言されていたと仮定する。final 宣言されたメンバは sub type で上書きすることが禁止される。そのため、メソッド f, h の 2 つについては、コンパイルエラーが発生する。

```
class T {
  final int f(int a) { // compile error
    if(a < 0) return 4;
    else return super.f(a);
  }
  final int h(int c) { // compile error
    if(c < 0) return 6;
    else return super.h(c);
  }
}
```

f, h を定義しているのはあくまで super type 側であるクラス S のため、refine クラス R0 と直接関係のない S を変更することは禁止している。上記のような f, h を拡張したい場合、プログラマは sub type ではなく super type を refine 対象として指定しなければならない。メソッド g, i については T に直接修正を加えることが可能なため、コンパイルエラーにはならない。

```
class T {
  final int g(int b) { // not compile error
    if(b < 0) return 5;
```

```
    else return 1;
  }
  final int i(int d) { // not compile error
    if(d < 0) return 7;
    else return 3;
  }
}
```

継承したメソッドを dispatch 対象に選択する際は、上記のように final 宣言の有無に注意しなければならない。継承における同様の留意点として、native 宣言されたメソッドも error の原因となる。また、super type にて private 宣言されたメソッドは super アクセスを用いてもアクセスを拒否されるため、dispatch 対象とすることはできない。

複数の条件付きメソッドの add

条件付きメソッドが refine 対象クラスに dispatch 対象のメソッドを見つけた場合、refine クラスの優先順位に関係無くその条件付きメソッドは add される。しかし add された条件付きメソッドが複数ある場合は、dispatch の結果を一意に定める必要があるため、条件付きメソッド間の優先順位が設けられる。処理系はより優先順位の高い側の条件付きメソッドから順に when 節を評価していき、最初に true を返した条件付きメソッドを実行する。

- 優先順位は次のように決定
  - T に対する優先度がより高い R に定義されている条件付きメソッドほど優先される
  - 同じ R に定義されている条件付きメソッド同士の場合、より前方に定義されているほうが優先される
- 複数の条件付きメソッドを優先度の高い順に並べる (D0(最優先), D1, ... , Dn とする)
- ２つ以上の D の when 節が true を返す場合、その中で最も優先度の高い D が実行され、その他の D は何もしない
- どの D の when 節も true を返さない場合、dispatch 対象となっている method がそのまま実行される

具体的には次のようになる。

```
1  class T {
2    int f(int i) { return 0; }
3  }
4  class R0 refines T {
5    int f(int i) when i > 20 { return 2; }
6  }
7  class R1 refines T {
8    int f(int i) when i > 10 { return 1; }
9  }
10 class R2 refines T {
11   int f(int i) when i > 30 { return 3; }
12 }
```

5行目、8行目、１１行目の f(int) は全て dispatch 対象として T.f(int) を選択する。その優先順位は refine クラスの優先順位に従うため、R0.f(int), R1.f(int), R2.f(int) の順で when 節のチェックが実行される。T.f(int) の呼び出しがあった場合、処理系はまず 5 行目に記述されている boolean 式の `i > 20` をチェックする。これが true を返すならば f(int) は 2 を返す。false を返すならば、次に処理系は 8 行目の式 `i > 10` をチェックする。これが true であるならば f(int) は 1 を返す。false を返すなら最後に処理系は１１行目の式 `i > 30` をチェックする。結果が true ならば f(int) は 3 を返す。false であるなら、全ての条件付きメソッドの when 節が false となっているので、オリジナルである 2 行目の f(int) が実行され 0 を返す。

```
class Sample {
  void v() {
    new T().f(5);  // return 0
    new T().f(15); // return 1
    new T().f(25); // return 2
    new T().f(35); // return 2 (not 3)
  }
}
```

処理系はこの f(int) に関するプログラムを次のように変換したものとして扱う。

```
class T {
  int f(int i) {
```

```
    if(i > 20) return 2;
    else if(i > 10) return 1;
    else if(i > 30) return 3;
    else return 0;
  }
}
```

条件付きメソッドの add における error 及び warning の発生条件は以下のように定める。

- when 節における error の発生条件
  1. 条件付きメソッド (C とおく) の method body が無いとき
  2. C が refine クラス以外の場所で宣言されているとき
  3. dispatch 対象の method(M とおく) の method body が無いとき
  4. M が refine 対象クラスの super type に定義されており、かつ refine 対象クラスから M へのアクセスまたは M の override が不可能であるとき
  5. when 節直後の expression が無い、あるいは expression が boolean 式になっていないとき
  6. C が throws 節を持ち、かつ M が throws 節を持たないとき
  7. C が throws 節を持ち、かつ M の throws 節に列挙した例外型のリストが C の例外型を包括していないとき
  8. C と M の return type が異なり、さらにそれらが M の return type を親側とするような継承関係でないとき
- when 節における warning の発生条件
  1. 条件付きメソッドの dispatch 対象となるようなメソッドが refine 対象クラスに見つからなかったとき

error が発生した場合、コンパイラはそこでエラーを出力して終了する。warning が発生した場合、コンパイラは警告文を出力し、その条件付きメソッドは定義されていないものとして扱われる。

条件付きメソッドは名前にワイルドカード*を含むことができる。引数や return type に*を含むことはできない。例えば、以下のような記述も可能である。

```
class R refines * {
  void set*(int i) when i < 0 {
```

```
    System.out.println("minus");
  }
}
```

proceed 呼び出し

条件付きメソッドは全てボディ内部において proceed 呼び出しが実行できる。これは add されたメソッドと同様で、dispatch 対象のオリジナルを呼び出すために用いられる。以下に例を示す。

```
1 class T {
2   int f(int i) { return i; }
3 }
4 class R refines T {
5   int f(int i) when i < 0 {
6     return proceed(i + 1);
7   }
8 }
```

条件付きメソッドの R.f(int) は、dispatch 対象として T.f(int) を選択している。R.f(int) における proceed 呼び出しの対象は T.f(int) である。proceed 呼び出しには dispatch の補正がかからない。例えば、T.f(int) を呼び出す際に、引数の値が-100 であった場合、これは条件 `i < 0` を満たしているため 5 行目の f(int) が実行される。6 行目では proceed の対象は T.f(int) であり、その引数の値は-99 である。しかしこの proceed 呼び出しには条件付きメソッドによる dispatch は行われないため、必ず 2 行目の f(int) が実行される。

```
class Sample {
  void v() {
    new T().f(100); return 100
    new T().f(-100); return -99
  }
}
```

3.3.5　client 変数; caller の参照

条件付きメソッドの when 節には「client instanceof type-access」という特殊な構文を用いることができる。

```
class R refines T {
  int f() when client instanceof Sample {
    return 1;
  }
}
```

when 節における「client instanceof 〜」は特別な意味を持ち、この構文が含まれている条件付きメソッドは client 情報の参照を持つことができる。client 情報とは、メソッドの呼び出し側のインスタンスであり、上の例では Sample クラスから T.f(int) が呼び出された場合にのみ 1 を返す。Sample クラス以外のインスタンスから f(int) がよばれている場合、この条件付きメソッドは実行されない。

```
class T {
  int f() { return 0; }
}
class Sample {
  void v() {
    new T().f(); // return 1
  }
}
class Sub extends Sample {
  void func() {
    new T().f(); // return 1
  }
}
class Another {
  void other() {
    new T().f(); // return 0
  }
}
```

上記の Sub クラスのように Sample を継承しているクラスも Sample クラスのインスタンスと成り得るため、処理系は条件付きメソッドを実行する。

when 節に特別な意味を持つ client を含む条件付きメソッドは、body 内からも client へのアクセスが可能となる。

```
class R refines T {
  String f() when client instanceof Sample {
    return client.toString();
  }
}
```

このプログラムでは、caller が Sample インスタンスのとき、f() は caller の toString() を返す。

client の自動予測

client 変数は、処理系により java.lang.Object 型として宣言されているため、アクセス時にキャストが必要な場合が多数ある。例えば以下のプログラムでは client へのアクセスにキャストが必要となる。

```
class T {
  String f() { return "a"; }
}
class Sample {
  int x = 0;
  void v() {
    new T().f();
  }
}
class R refines T {
  String f() when client instanceof Sample {
    return ((Sample)client).x; // cast
  }
}
```

しかし、このようなキャスト処理は煩わしいときがある。上記のプログラムでは caller が Sample インスタンスである場合のみを dispatch の条件と明示しているにも関わらず client のキャストにも「Sample」クラスであることを記述しなければならない。

そこで本言語では client 自動予測をおこなう。例えば上記の例では Sample クラスが caller であることは明示されているため、client は Sample クラスに自動キャストされる。

```
class R refines T {
  String f() when client instanceof Sample {
    return client.x + "a"; // not compile error
  }
}
```

client の型が自動予測できないような場合、自動キャストは行われない。

static な caller に対する client

client は、caller サイドにおける this オブジェクトが渡される。本言語では caller が static メソッドであった場合、例外的に client に String オブジェクトが渡されることになる。String の値は呼び出し側の [『パッケージ名』"."『クラス名』":"『メソッドシグネチャ』] である。

```
package p1;

class R refines T {
  String f() when client instanceof Sample {
    return client.toString();
  }
}
class Sample {
  static void v() {
    new T().f(); // return "p1.Sample:v()"
  }
}
```

client に関する error と warning

client への値の代入は決してできず、そのような記述があった場合は compile error となる。

```
class R refines T {
  void f() when client instanceof Sample {
    client = null; // compile error
  }
}
```

クラス T に client という名前のメンバフィールドが存在する場合は、this.client と client を使い分ければ良い。

```
class R refines T {
  int client = 0;
  void f() when client instanceof Sample {
    this.client = 1; // not compile error
  }
  void g() when this.client == 0 {
    // not compile error
  }
}
```

client に関する error 及び warning の処理は次の通りになる。

- error の発生条件
  1. client への値の代入をおこなう命令が記述されているとき
- warning の発生条件
  1. 条件付きメソッドの body に client 情報へのアクセス命令が記述されており、かつその条件付きメソッドが dispatch 対象として選択しているメソッドを、１つ以上の static メソッドが呼び出しているとき

上記の warning が発生するような場合、client 情報へのアクセスのタイミングによっては、実行時例外「`lava.lang.ClassCastException`」が起こる可能性がある。static メソッドが caller であるときは client の参照先は代替の String オブジェクトとなっているからである。String オブジェクトをキャストし損なえば、それは実行時例外である。

3.3.6　メンバ変数の add, replace

メンバ変数の add, replace はメソッドと同様である。メソッドとの違いは、シグネチャではなく名前だけを見ることと、proceed 機能が存在しないことである。

```
class T { }
class T2 {
  int x = 0;
}
```

```
class R refines T {
  int x = 1; // add
}
class R2 refines T2 {
  int x = 2; // replace
}
```

replace されたメンバ変数のオリジナルの値は、プログラム上のどの場所からも参照することができなくなる。

```
class Sample {
  void v() {
    new T().x;  // return 1
    new T2().x; // return 2
  }
}
```

refine 対象のタイプ T が他の refine クラスのそれと重複するとき、メンバ変数の add や replace が確実に行われるとは限らない。その判定はメソッドの add, replace 時と同様に「refine クラスの優先順位」に従う。

- クラス T に対する優先順位に従い、それぞれの refine クラスを R0(最優先), R1, ... , Rn とおく
- 各 Rn が持つ field(F とおく) が以下の条件に当てはまるとき、T に F を add
  - (n = 0 のとき) T に F と同じ名前の field が無い場合
  - (n > 0 のとき) F と同じ名前の field を、Rn より T に対する優先度の高いどの Rx(x < n) も持たず、かつ T に F と同じ名前の field が無い場合
- 各 Rn が持つ field(F とおく) が以下の条件に当てはまるとき、T に F を replace
  - (n = 0 のとき) T に F と同じ名前の field がある場合
  - (n > 0 のとき) F と同じ名前の field を、Rn より T に対する優先度の高いどの Rx(x < n) も持たず、かつ T に F と同じ名前の field がある場合

また、メンバ変数の add, replace における error や warning の発生条件は以下の通りである。

- error の発生条件
  1. refine 対象のタイプにインターフェイスが含まれており、かつ field が初期値を持たないとき
  2. field の初期値に含まれる参照先が、その field よりも後方で定義されている field であったとき
- warning の発生条件
  1. そのメンバ変数と同じ名前を持つ field が、同じ refine クラス内で F よりも前方に定義されている場合
  2. そのメンバ変数が、条件を満たせず add も replace もされなかったとき

warning が発生したとき、そのメンバ変数は定義されていないものとして扱われる。例として以下のプログラムを考える。

```
1 class T { }
2 class R1 refines T {
3   int x = 0;
4 }
5 class R2 refines T {
6   int x = 1; // warning
7 }
```

上記のプログラムを処理系が扱うとき、6 行目を指して warning が発生する。T.x は既により優先度の高い R1 によって add されているからである。本システムでは add あるいは replace されたメンバ変数を再び replace することはできない。このとき R2.x は元々記述されていないものとしてコンパイルが続行される。

3.3.7　コンストラクタの add, replace

refine クラスは、コンストラクタを add, replace できる。

```
class T {
  int x;
}
class T2 {
  int x;
  T2() { x = 0; }
```

```
}
class R refines T {
  T(int i) { x = i; } // add
}
class R2 refines T2 {
  T2() { x = 10; } // replace
}
```

コンストラクタの名前は、refine 対象のクラスと同じにしなければならない。また、replace された際のオリジナルのコンストラクタを呼び出すことはできない。refine クラスが refine 対象のクラスにコンストラクタを追加した場合、refine 対象のクラスには必ずデフォルトコンストラクタが生成される。上の例では refine クラス R はクラス T に引数が int であるコンストラクタを追加している。このため T にはデフォルトコンストラクタ（引数を取らないコンストラクタ）が自動的に定義される。そのためコンパイルのある時点で、上記のプログラムは次のような形に修正される。

```
class T {
  int x;
  T(int i) { x = i; }
  T() { } // add default constructor
}
class T2 {
  int x;
  T2() { x = 10; }
}
```

コンストラクタの add, replace はプログラム全体に反映される。

```
class Sample {
  void v() {
    new T();  // not compile error
    new T(5);  // not compile error
    new T2(); // x = 1
}
```

refine 対象のタイプ T が他の refine クラスのそれと重複するとき、コンストラクタの add や replace が確実に行われるとは限らない。その判定はメソッドの add, replace 時と同様に「refine クラスの優先順位」に従う。

- クラス T に対する優先順位に従い、それぞれの refine クラスを R0(最優先), R1, ... , Rn とおく
- 各 Rn が持つコンストラクタ (C とおく) が以下の条件に当てはまるとき、T に C を add
  - (n = 0 のとき) T に C と同じシグネチャのコンストラクタが無い場合
  - (n > 0 のとき) C と同じシグネチャのコンストラクタを、Rn より T に対する優先度の高いどの Rx(x < n) も持たず、かつ T に C と同じシグネチャのコンストラクタが無い場合
- 各 Rn が持つコンストラクタ (C とおく) が以下の条件に当てはまるとき、T に C を replace
  - (n = 0 のとき) T に C と同じシグネチャのコンストラクタがある場合
  - (n > 0 のとき) C と同じシグネチャのコンストラクタを、Rn より T に対する優先度の高いどの Rx(x < n) も持たず、かつ T に C と同じシグネチャのコンストラクタがある場合

また、コンストラクタの add, replace における error や warning の発生条件は以下の通りである。

- error の発生条件
  1. コンストラクタの名前が、refine 対象のタイプ名と一致しないとき
  2. refine 対象のタイプにインターフェイスが含まれているとき
- warning の発生条件
  1. そのコンストラクタと同じシグネチャのコンストラクタが、同じ refine クラス内でより前方に定義されている場合
  2. そのコンストラクタが条件を満たせず、add も replace もされなかったとき

warning が発生したとき、そのコンストラクタは定義されていないものとして扱われる。例として以下のプログラムを考える。

```
1 class T {
2   int x;
3 }
4 class R1 refines T {
```

```
5     T() { x = 1; };
6   }
7   class R2 refines T {
8     T() { x = 2; }; // warning
9   }
```

上記のプログラムを処理系が扱うとき、6 行目を指して warning が発生する。T() は既により優先度の高い R1 によって add されているからである。本システムでは add あるいは replace されたメンバ変数を再び replace することはできない。このとき R2 の持つ T() は元々記述されていないものとしてコンパイルが続行される。

3.3.8　入れ子タイプの add, replace

refine クラスは、入れ子タイプを add, replace できる。

```
class T { }
class T2 {
  class Nest { }
}
class R refines T {
  class Nest { } // add
}
class R2 refines T2 {
  class Nest { } // replace
}
```

refine 対象クラスの元々定義されていた入れ子クラスであれば、その入れ子クラスもまた refine 対象に選択できる。例えば、

```
class T {
  class Nest { }
}
class R refines T {
  class Nest {
    int x = 0;
  }
}
```

このようなプログラムでは、R は refine 対象クラス T の持つ Nest クラスを replace し、新たに x というメンバ変数を定義させている。このプログラムはコンパイルのある時点から、次のようなプログラムとして扱われる。

```
class T {
  class Nest {
    int x = 0;
  }
}
```

このような変換をおこなう refine クラスは次のようにも書くことができる。

```
class T {
  class Nest { }
}
class R refines T.Nest {
  int x = 0;
}
```

入れ子タイプの add, replace に関する制約は、メンバ変数の add, replace のそれとよく似ている。refine 対象のタイプ T が他の refine クラスのそれと重複するとき、入れ子タイプの add や replace が確実に行われるとは限らない。その判定はメソッドの add, replace 時と同様に「refine クラスの優先順位」に従う。

- クラス T に対する優先順位に従い、それぞれの refine クラスを R0(最優先), R1, ... , Rn とおく
- 各 Rn が持つ入れ子タイプ (N とおく) が以下の条件に当てはまるとき、T に N を add
  - (n = 0 のとき) T に N と同じ名前の入れ子タイプが無い場合
  - (n > 0 のとき) N と同じ名前の入れ子タイプを、Rn より T に対する優先度の高いどの Rx(x < n) も持たず、かつ T に N と同じ名前の入れ子タイプが無い場合
- 各 Rn が持つ入れ子タイプ (N とおく) が以下の条件に当てはまるとき、T に N を replace
  - (n = 0 のとき) T に N と同じ名前の入れ子タイプがある場合

– (n > 0 のとき) N と同じ名前の入れ子タイプを、Rn より T に対する優先度の高いどの Rx(x < n) も持たず、かつ T に N と同じ名前の入れ子タイプがある場合

また、入れ子タイプの add, replace における warning の発生条件は以下の通りである。

- warning の発生条件
  1. その入れ子タイプと同じ名前を持つ入れ子タイプが、同じ refine クラス内で N よりも前方に定義されている場合
  2. その入れ子タイプが条件を満たせず、add も replace もされなかったとき

warning が発生したとき、その入れ子タイプは定義されていないものとして扱われる。例として以下のプログラムを考える。

```
1 class T { }
2 class R1 refines T {
3   class Nest { }
4 }
5 class R2 refines T {
6   class Nest { } // warning
7 }
```

上記のプログラムを処理系が扱うとき、6 行目を指して warning が発生する。T.Nest は既により優先度の高い R1 によって add されているからである。本システムでは add あるいは replace された入れ子タイプを再び replace することはできない。このとき R2.Nest は元々記述されていないものとしてコンパイルが続行される。

3.3.9　refine クラスによる仕様の変更

refine クラスはオリジナルのソースコードを上書きすることができる。アクセス指定子や return type の変更等も可能である。

```
class T {
  void f() { }
}
class Sample {
```

```
  void v() {
    int x = new T().f(); // error
  }
}
```

例えば、上記のようなコードはコンパイルエラーが発生する。int 型に void 型を代入することは不可能だからである。そこで refine クラスが、T.f() を以下のように replace したとする。

```
class R refines T {
  int f() { return 0; } // replace
}
```

このとき、前述のコンパイルエラーは解消される。このように、refine クラスによる仕様の変更はプログラム全体に反映される。

本言語がこのような仕様になっているのは、Mock 等を利用したい場合に refine クラスが役立つからである。まだ存在していないメソッドや変数が一時的に必要になった場合に、オリジナルのソースではなく refine クラスを用いることでそれらが実現できる。

3.3.10　分割コンパイル

refine クラスは、refine 対象となるクラスを修正する機能を持つ。そのため、refine クラスのコンパイルには、必ず refine 対象クラスのコンパイルも含まれなければならない。

```
class R refines T {
  int x = 0;
}
```

refine クラス R をコンパイルするとき、コンパイラには refine 対象クラス T のソースファイルが渡されなければならない。そうでなければコンパイラはエラーを出力して終了する。

R 及び T は refine 節の前後で確実に記述されているため、それらがコンパイラにとって必要なファイルであることは予測し易い。しかし client 参照の必要が生じた場合、コンパイラは分割コンパイルをすることができない。

```
class R refines T {
  int f() when client instanceof Sample {
    return 1;
  }
}
```

例えば、上のプログラムの場合、R 及び T のソースファイルだけでは T.f() がプログラム上どこから呼ばれるかが判断できない。全ての呼び出し箇所において client 情報は登録されなければならないため、これは分割コンパイルが不可能である。

そのため本言語では、T.f() を呼び出すような箇所は全てコンパイラへのソースファイルとして明示しておかなければならない。

第4章　提案言語の実装

この章では、3章で提案した言語をどのように実装したかを説明する。また、実装のために使用したツール「JastAdd」についても記述する。

4.1　JastAdd

JastAdd はオープンソースのコンパイラ作成ツールである。これはプログラムソースを解析した際に生成される AST を、Java 言語によって操作することでコンパイラを作成する。ソースの字句解析及び構文解析には任意のツールを用いることができる。JastAdd が提供する機能は、AST の各ノードに対応するクラスの定義を決定することである。AST の全てのノードは必ず Java のクラスオブジェクトとして生成される。JastAdd はそのクラスにどのような処理をさせるかを定義する言語を持つ。また、それらの定義言語はアドオンのように随時追加させることが可能になっており、これはアスペクト指向のプログラミングスタイルに似ている。

4.1.1　属性文法

JastAdd は属性文法 (Attribute Grammar) の概念を取り入れている。属性文法は、AST 上の各ノードに属性を付加し、その値によりコンパイラの処理内容を決定させる手法である。属性には、型や名前のほかに、宣言場所といったものの情報を保持させる。属性は各ノードで個別に決定されるもの以外にも、AST 上の親あるいは子ノードから伝達され自動的に計算されるものがある。そのため属性文法は、関係の深いノード同士に共通の処理をおこなう場合に向いている。

JastAdd は、各ノードが Java クラスとして操作できるため、属性をクラスメンバとして保持する。そのため各々の属性の情報に Java ライクにアクセスできるという利点がある。

4.1.2　JastAddJ

JastAddJ[5] は、JastAdd で作成された Java1.4 & 1.5 コンパイラである。JastAddJ は Java のソースコードから、クラスファイルを生成する。JastAdd の開発チームは、Java1.4 用のコンパイラを作成し、それに Java1.5 用のコンパイラを追加実装することで JastAddJ を作成した。これは、JastAdd がアドオン形式でコンパイラの機能を追加できるためである。本研究ではこのスタイルに習い、JastAddJ に refines 節等の機能を追加することで提案言語を実装する。

JastAddJ は Java1.4 の Frontend と Backend、そして Java1.5 の Frontend と Backend、の４種から成る。我々が機能を追加した部分は、Java1.4Frontend 部分に対してである。機能追加の方法は、オリジナル JastAddJ の修正ではなく、新たな定義ファイルを外部から取り込ませることで実現可能である。

4.2　言語の実装

JastAdd は AST の各ノードに対応する Java クラスを定義する。我々は、JastAddJ によるコンパイルのあるタイミングで refine クラス及びそこに定義されているメンバの専用処理を実装する。

4.2.1　文法解析

本言語は、従来の Java の文法に、refine クラスと条件付きメソッドの構文を追加している。JastAdd は文法解析の際に任意の解析器を用いることができる。JastAdd の開発チームは、代表的な解析器として JFlex[11] と beaver[2] のセットと、JavaCC(JJTree)[10] を提案している。本研究では JastAddJ に習い、字句解析器として JFlex を、構文解析器として beaver を用いた。

コンパイラが refine クラスの構文解析に成功したとき、AST ノードとして RefiningClassDecl クラスのオブジェクトが生成される。RefiningClassDecl は refine 対象のクラス群の情報を持つ。refine 対象のクラスは最低でも１つ存在し、refine 対象クラスの指定にワイルドカードを用いている場合には２つ以上のクラスを対象として持つ場合がある。

また、コンパイラが条件付きメソッドの構文解析に成功したとき、AST ノードとして ConditionalMethodDecl クラスのオブジェクトが生成される。ConditionalMethodDecl は dispatch 対象のメソッド群の情報と、dispatch 条件文 (when 節直後の boolean 式) を持つ。

我々は、JastAddJ に機能を追加し、RefiningClassDecl 及び ConditionalMethodDecl クラスを定義した。前者は Java クラスを表す AST ノードに対応する ClassDecl を継承している。また、後者はメンバメソッドを表す AST ノードに対応する MethodDecl を継承している。RefiningClassDecl が持つ情報は「ClassDecl に定義された情報 + refine 対象クラスの参照」であり、ConditionalMethodDecl が持つ情報は「MethodDecl に定義された情報 + dispatch 対象メソッドの参照 + dispatch 条件式」である。また、それらに伴い ClassDecl には「自身を Refine 対象として選択している refine クラス群への参照」を情報として新たに持つよう定義し、同様に MethodDecl には「自身を dispatch 対象として選択している条件付きメソッド群への参照」を新たに持つよう定義している。

4.2.2　クラスへの refine クラス情報の登録

コンパイラは構文解析終了後、全ての refine クラスに対し、refine 対象クラスのノードを正確に参照可能かどうかをチェックする。ノードが参照不可能であるときは、それは refine 対象クラスが存在しないか、あるいはソースコードが入力されたソースファイル群の中に見当たらない場合である。そのような場合、コンパイラはエラーを出力して終了する。refine 対象クラスが全て見つかったとき、次にコンパイラは対象クラス側のノードオブジェクトに、それ自身を refine 対象クラスとして選択している refine クラスを登録する。

```
class T { }
class R1 refines T { }
class R2 refines T { }
```

上のクラス T に対応する AST ノードは T という名前の ClassDecl オブジェクトである。コンパイラはその T オブジェクトに refining_list として R1, R2 ノードへの参照を持たせる。この登録により、T ノードからは自身を refine しようとする refine クラスが存在し、それが R1, R2 ノードであることが分かる。

この refine クラスのリストはコンパイラによって優先順位の高いほうから（辞書順で）登録される。上の例ではリストの 1 番目は R1 で、2 番目は R2 である。これはコンパイラがソースファイルを辞書順に従って解析しているからである。

4.2.3　refine クラスによる型階層の変更

３章の仕様で述べた通り、refine クラスは型階層を変更することができる。コンパイラは各クラスが持つ refining_list を参照し、いずれかの refine クラスによって super type あるいは interface が変更されている場合に型階層を変更する。入力ファイルとして与えられた全てのクラスで同様の処理を順次おこない、型階層に循環等の不具合が発生した瞬間にエラーを出力して終了する。この処理の時点ではあくまで継承関係の不具合をチェックするだけである。抽象メソッドのオーバーライドが成されないまま継承が成立した場合でもこの時点ではエラーとして出力しない。なぜならばそれは次の段階におけるメンバの add で解消される可能性があるからである。

4.2.4　refine クラスからのメンバの add と replace

refine クラスに定義したメンバは refine 対象のクラスに add または replace される。add とは、オリジナルのソースに定義されていないメンバを refine クラスが定義することである。また、replace とはオリジナルのソースに定義されているメンバを refine クラスが再定義（上書き）することである。ここでは refine クラスに定義されたメンバ変数、メソッド、入れ子クラス、コンストラクタを refine 対象のクラスにコピーする。条件付きメソッドのみ、add されずに次の段階へ持ち越される。

add 処理

まずコンパイラは、add 条件を満たすメンバだけを全てのクラスに add する。replace は一切おこなわれない。コンパイラは全てのクラスノードの refining_list を参照し、リストにある全ての refine クラスのメンバの中から『そのクラス自身が定義していないメンバ』のみコピーを取得し、それ自身に定義する。

入力ソース

```
class T {
  int x = 0;
}
class R1 refines T {
  int x = 1; // replace
}
class R2 refines T {
```

```
  int y = 2; // add
}
```

例えばこのソースから生成される AST のクラス T のノードは、refining_list として R1, R2 を登録されている。コンパイラは T に R2.y のみを add する。この段階では replace はおこなわれない。add と replace の間では、コンパイラによる意味解析がおこなわれる。意味解析の対象には、refine クラスは含まれない。コンパイラは、入力されたソースが以下のように変換された時点で最初の意味解析をおこなう。

最初の意味解析時の入力ソース

```
class T {
  int x = 0;
  int y = 2;
}
class R1 refines T {
  int x = 1;
}
class R2 refines T { }
```

解析対象はクラス T に該当するノードとその子ノードである。refine クラスに該当する側 (R1, R2) は解析対象にはならない。refine クラスを解析しない理由は 2 つある。1 つは、refine クラスのメンバ定義は全て refine 対象クラスに定義されたものとして扱うからであり、もう 1 つは refine クラスはクラスファイルも生成されず、refine クラスオブジェクトを生成するようなコードを禁止しているためである。refine クラスはあくまで構文解析の対象であり、意味解析をおこなうことに意味は無い。refine クラスはメンバの add, replace が終了した時点で元々存在しなかったものとして扱われるのである。

入力ソースの最終的な形式

```
class T {
  int x = 1;
  int y = 2;
}
```

コンパイラが add と replace のタイミングを分離し、その間で意味解析をすることには理由がある。それは、replace によりオリジナルのソースが失われるためである。もしもコンパイラが replace 後に意味解析をおこなっていた場合、replace される前のオリジナルのソースが発見されない

ため、そこに含んでいたエラーが発見されない恐れがあるからである。例えば、以下のようなプログラムを考える。

エラーコードを replace する例

```
class T {
  int x = p; // compile error?
}
class R refines T {
  int x = 1;
}
```

オリジナルの T.x は p という値を参照しているが、p という変数の参照先は見つからないため、これはエラーとなるべきである。仮に replace が意味解析より先におこなわれた場合、T.x の値は 1 となってしまい、p への参照コードは上書きされ消えてしまう。本研究では refine クラスを mock 等の目的以外においてエラーコードを封じ込めるような形で使用するのはあまり望ましくないと考えているため、replace よりも前に意味解析をおこなうという仕様にしている。

また、意味解析が add よりも後であるのは、こちらは mock として refine クラスを利用できると考えたからである。さらにもう 1 つの理由として、型階層の変換に伴うメソッドの実体化に影響を及ぼすからである。以下に例を挙げる。

```
abstract class Abs {
  int f();
}
class T { }
class R refines T extends Abs {
  int f() { return 0; }
}
```

refine クラス R はクラス T の super class を抽象クラス Abs へと変更している。この型階層の変更は、以前に述べた通り、メンバの add や replace よりも先におこなわれる。もしも意味解析が add よりも前の段階でおこなわれた場合、上記の例ではクラス T がメソッド f() を実装していないというエラーが発生する。そのため、意味解析は add の後におこなう仕様になっている。また、仮に型階層の変更よりも以前に意味解析をおこなう場合、refine クラスを mock として利用することができないという欠点が残る。

replace 処理

メンバを replace する際に、注意しなければならないのは特別な意味を持つ proceed の存在である。proceed は上書きされる前のオリジナルのメソッドを呼び出すための関数である。

proceed を含むソース

```
class T {
  int f(int i) { return i * 2; }
}
class R refines T {
  int f(int i) { return proceed(i + 1); }
}
```

replace の直前に、メンバ関数に対応するノードオブジェクトは、自身を replace しようとするメンバ関数を全て調査する。ここでは T.f(int) に対応する MethodDecl オブジェクトが、R.f(int) の内部を検索する。もしもそこに proceed 関数が無ければ、T.f(int) は上書きされてソース上から消滅したように扱われる。ここでは proceed 関数が発見されるため、T.f(int) は proceed の対象関数としてソース上に残される。そのため上のソースは以下のように変換される。

proceed を含むソースの変換

```
class T {
  int f(int i) { return f$proceed$1(i + 1); }
  int f$proceed$1(int i) { return i * 2; }
}
```

オリジナルの f(int) は便宜上、名前を変換してクラス T に残る。名前の最後の数字「1」は、そのメンバがクラスメンバの中で上から何番目に定義されているかを表している。最初の定義は 0 番目であるため、ここでは 1 が付加される。全ての f(int) 呼び出しは自動的に、replace した側の f(int) を呼ぶ処理へと変換されたことになる。proceed が無い場合、オリジナルのソースはどこからも呼び出されることがないため、ソースには残らない。proceed 関数の引数がそのメソッドと異なる場合、proceed は特別な意味を持たない。この場合、コンパイラはこれを通常の関数呼び出しと同じものとして処理を続行する。

4.2.5　条件付きメソッドの add 処理

refine クラスが保持するメンバの中でも条件付きメソッドは処理が異なり、その他のメンバの add, replace が全て終了した後に処理がおこなわ

れる。

条件付きメソッドを含む入力ファイル

```
class T {
  int f(int i) { return 0; }
}
class R refines T {
  int f(int i) when i < 0 {
    return 1;
  }
  int f(int i) when i < -10 {
    return 2;
  }
}
```

条件付きメソッドは同じメソッドを dispatch 対象として選択することができる。上記のソースでは T.f(int) を 2 種類の条件付きメソッドが拡張している。T.f(int) が呼び出された時点で when 節の boolean 式を評価させるために、コンパイラは T.f(int) の最初に if 文による条件分岐を生成する。

条件付きメソッドを変換

```
class T {
  int f(int i) {
    if(i < 0) {
      return f$1(i);
    }
    if(i < -10) {
      return f$2(i);
    }
    return 0;
  }
  int f$1(int i) { return 1; }
  int f$2(int i) { return 2; }
}
```

条件付きメソッドは便宜上名前を変換して add される。ここでは 2 種類の f(int) は優先順位に従い f1(int) と f2(int) という名前に変換される。オリジナルの f(int) においては if 文が条件付きメソッドの数だけ生成される。各 if 文の条件式は、それぞれの条件付きメソッドにて記述されていた when 節の boolean 式と同じである。

また、特別な意味を持つ proceed は条件付きメソッドにおいても使用可能なため、dispatch 対象のメソッドは自身のオリジナルソースを残すかどうかを判断しなければならない。

条件付きメソッドが proceed を含むソース

```
class T {
  int f(int i) { return i * 2; }
}
class R refines T {
  int f(int i) when i < 0 {
    return proceed(-i);
  }
}
```

proceed 呼び出しには dispatch による条件分岐は実行されない。この制約を満たすため、コンパイラはオリジナルの f(int) と同等の処理をするメソッドをコピーして同じクラスに名前を変更して定義する。

変換後のソース

```
class T {
  int f(int i) {
    if(i < 0) {
      return f$1(i);
    }
    return i * 2;
  }
  int f$1(int i) { return f$proceed$2(-i); }
  int f$proceed$2(int i) { return i * 2; }
}
```

この変換により、proceed 呼び出しの対象となる関数は dispatch の影響を受けない。

条件付きメソッドの add が全て終了した後、コンパイラは 2 度目の意味解析をおこなう。この意味解析の対象となるのは、1 度目の意味解析をおこなった対象メンバ以外である。それは replace されたメンバと、add された条件付きメソッドである。

4.2.6　client に関する処理

client 情報が必要なとき、コンパイラはその情報を保持するためのメンバ変数を自動的に定義する。前章で述べた通り、caller を取得するための

client 変数は必ず条件付きメソッドの when 節の中に記述していなければならない。

caller 情報を必要とする条件付きメソッド

```
class T {
  int f(int i) { return i * 2; }
}
class R refines T {
  int f(int i) when client instanceof Sample {
    return -i;
  }
}
```

上記のように、when 節に「client instanceof 〜」というスタイルの条件式が含まれているとき、変数 client は T.f(int) の caller オブジェクトを指すように変換される。このようなソースを入力したとき、コンパイラは T.f(int) の method dispatch には client 情報が必要であると解釈し、refine 対象クラスである T に client 情報を保持するためのメンバ変数を定義する。

メンバ変数 client の追加

```
class T {
  int f(int i) {
    if(client$0 instanceof Sample) {
      return f$1(i);
    }
    return i * 2;
  }
  int f$1(int i) { return -i; }
  public static java.lang.Object client$0;
}
```

client という名前のメンバ変数が予め定義されている可能性を考慮し、caller 情報を取得するための client は便宜上名前を変換され定義される。また、コンパイラは全ての T.f(int) 呼び出し箇所において client$0 にその呼び出し箇所のインスタンスを保持させるように処理を追加する。これらの処理により、本言語では caller オブジェクトを参照可能にする。client$0 にアクセスできないようなクラスの場合はそもそも T.f(int) にアクセスすることができないため問題は無い。

client$0 というメンバ変数はマルチスレッドに対応していないため、本言語ではこれを更に変更してスレッド毎に個別の値を保持できるようソース

を変換する。具体的には java.lang.ThreadLocal クラスを用いる。ThreadLocal は実行されているスレッド毎に違う値を保存しておくことのできるクラスである。値の登録は set メソッド、参照は get メソッドでおこなう。上記のソースは次のように変換される。

メンバ変数 client の変換

```
class T {
  int f(int i) {
    if(client$0.get() instanceof Sample) {
      return f$1(i);
    }
    return i * 2;
  }
  int f$1(int i) { return -i; }
  public static java.lang.ThreadLocal client$0 =
    new java.lang.ThreadLocal();
}
```

これで client 情報はマルチスレッドに対応可能となった。この処理に加え、全ての T.f(int) 呼び出し箇所では client 情報の登録がおこなわれる。例えば、以下のような Sample クラスの入力ファイルを考える。

T.f(int) を呼ぶ処理

```
class Sample {
  void v() {
    int x = 1;
    new T().f(x);
    x = 2;
  }
}
```

これは v() の内部で T.f(int) を呼び出しているため、client 情報の登録がおこなわれる必要が生じる。また、フローが v() を抜ける場合には client 情報を登録直前の状態に戻す必要があるため v() メソッドは以下のように変換される。

T.f(int) を呼ぶ処理の変換

```
1 class Sample {
2   void v() {
3     java.lang.Object o$0 = T.client$0.get();
4     T.client$0.set(this);
5     int x = 1;
```

```
6       new T().f(x);
7       x = 2;
8       T.client$0.set(o$0);
9     }
10  }
```

3 行目の式は、client に自身のオブジェクトを登録する前のデータをローカル変数として保存する。この値は 6 行目で client 情報を元の状態に戻すために用いられる。4 行目は client に自身を登録する処理をおこなっている。これらの変換により、client には Sample オブジェクトが登録され、caller 情報による method dispatch がおこなわれる。client 情報の登録は、メソッド v() の最初と最後でおこなう。T.f(int) 呼び出しの直前直後で client 情報を扱わない理由は、for 文や while 文の条件式等、前後で client 情報の登録をおこなうのが難しい場合が存在するからである。

本言語では T.f(int) を呼び出す箇所は、入力されたソースファイルの中でしか検索することができない。そのため client 情報に関しては分割コンパイルができない。

例外処理への対応

フローの途中で例外が throw されたとき、client 情報が修正されないまま制御が移行してしまう恐れがある。例えば上記の例で T.f(int) が例外を起こしたならば、8 行目でおこなわれるはずだった client 情報の再登録が実行されず、client 情報がずれたまま処理が続行されてしまう恐れがある。

T.f(int) が例外を起こす場合

```
1   class Sample {
2     void v() {
3       java.lang.Object o$0 = T.client$0.get();
4       T.client$0.set(this);
5       int x = 1;
6       new T().f(x); // throw Exception
7       x = 2;
8       T.client$0.set(o$0); // no flow
9     }
10  }
```

しかし Java の例外処理には制約がある。例外発生後、それを catch しない限りフローは Java のソースには戻らない。つまり、Java のフローが再開される場所には必ず catch 節が存在するはずである。

func() が例外を起こす場合

```
class Sample {
  void v() {
    int x = 1;
    new T().f(x);
    try {
      func();
    } catch(Exception e) { }
    new T().f(x);
    x = 2;
  }
}
```

catch 節は例外を catch し、フローを再開させる。本言語ではフロー再開時には caller 情報を再登録するようソースを変換する。

catch 節の変換

```
class Sample {
  void v() {
    java.lang.Object o$0 = T.client$0.get();
    T.client$0.set(this);
    int x = 1;
    new T().f(x);
    try {
      func();
    } catch(Exception e) {
      T.client$0.set(this); // add new code
    }
    new T().f(x);
    x = 2;
    T.client$0.set(o$0);
  }
}
```

v() の範囲からでは func の内部実装までは分からない。もしも func 内部に T.client$0 への登録処理が存在し、そして例外処理によって client$0 の値が修正されないまま catch 節にフローが移行した場合、１２行目の T.f(int) 呼び出しは client 情報がずれたままになる可能性がある。そのためコンパイラは catch 節内部に client 情報の登録処理を再び追加する。例では１０行目がそれである。

この実装の利点は、各メソッドがローカル変数として client 情報を随時変更することで、フローの推移に対応できることである。例えば複数のメソッド f(), g(), h(), i() が全て同じ client に情報登録をおこなう場合を考える。f は g を呼び、g は h を呼び、そして h は i を呼ぶとする。この場合、フローは「f() → g() → h() → i()」のようになる。i が client 情報を登録するとき、ローカル変数として i は h を定義するクラスオブジェクトを保存する。同様に h は g のそれを保存し、g は f のそれを保存する。例外処理によりフローが飛んだ場合であっても各メソッドは予め保存しておいたローカル変数により client 情報の正確な修正が可能となる。仮に client 情報をローカル変数ではなくスタック形式で保存していた場合、例外処理が発生する度に全ての client 情報用メンバ変数をチェックし、フローの移行に沿うように正確に要素を pop しなければならないだろう。

client の正確な登録

プログラムの途中に return 文や throw 文があるとき、client の再登録が必要となるため、コンパイラは処理を自動で追加する。

return, throw を含むソース

```
class Sample {
  void v() throws Exception {
    int x = 0;
    if(new T().f(x) == 1)
      return;
    else if(x == 2)
      throw new Exception();
    x = 3;
  }
}
```

上記のプログラムは return や throw 文により途中でメソッドを抜ける場合があるため、コンパイラは漏れのないように client 情報を操作する。

return, throw を含むソースの変換

```
class Sample {
  void v() throws Exception {
    java.lang.Object o$0 = T.client$0.get();
    T.client$0.set(this);
    int x = 0;
    if(new T().f(x) == 1) {
      T.client$0.set(o$0); // add new code
```

```
      return;
    }
    else if(x == 2) {
      T.client$0.set(o$0); // add new code
      throw new Exception();
    }
    x = 3;
    T.client$0.set(o$0);
  }
}
```

また、dispatch 対象のメソッドが return 文に含まれているとき、client 情報を前の状態に戻す処理がおこわれないまま return が実行されてしまうため、return 文の式をローカル変数として保存し、client 情報を操作してから保存値を return するよう変換する。

return 文に T.f(int) 呼び出しがあるソース

```
class Sample {
  int v() {
    int x = 0;
    return new T().f(x) + 1;
  }
}
```

上のソースはローカル変数 temp$0 を用いて以下のように変換される。

return 文に T.f(int) 呼び出しがあるソースの変換

```
class Sample {
  int v() {
    java.lang.Object o$0 = T.client$0.get();
    T.client$0.set(this);
    int x = 0;
    int temp$0 = new T().f(x) + 1; // add new code
    T.client$0.set(o$0);
    return temp$0; // change return value
  }
}
```

変数 temp の型はそのメソッド v() の return type とし、一時的に return の値を保存する。

static メソッドからの client 登録

static メソッドは this の参照ができない。そのため client 登録時には this 以外の値を必要とする。本言語は static メソッドは client に java.lang.String オブジェクトを渡す仕様になっている。例えば、変換後のソースは以下のようになる。

static メソッドからの呼び出し

```
class Sample {
  static int v(int x) {
    java.lang.Object o$0 = T.client$0.get();
    T.client$0.set("Sample:v(int)"); // set String object
    new T().f(x);
    T.client$0.set(o$0);

  }
}
```

String の値は、クラスのフルパス名 (Sample) にコロン (:) をつけ、さらにメソッドシグネチャ(v(int)) を付加したものになる。これに伴い、dispatch 対象となったメソッド側における if 文の条件分岐も変更の必要が生じる。

String に対応する変換

```
class T {
  int f(int i) {
    if((client$0.get() instanceof Sample) ||
       ((client$0.get() instanceof java.lang.String) &&
        (((java.lang.String)client$0.get()).
          startsWith("Sample:")
      ))) {
      return f$1(i);
    }
    return i * 2;
  }
}
```

client のインスタンスが Sample だけでなく、String かつ値が「Sample:〜」である場合にも条件式は true を返さなくてはならない。コンパイラはこのような複雑な条件式を自動で生成する。

4.2.7　クラスファイル生成

コンパイラはこれまでの変換を全て終えた後に、クラスファイルを生成する。生成されるのは refine クラス以外のクラスまたはインターフェイスに該当するファイルである。その中でも refine 対象として選択されたクラス及び client 情報への登録が必要なクラスは、全てソースコード上の変換がおこなわれた上でのクラスファイル化となっている。ソースコードの変換に不備がある場合、コンパイラはエラーを出力して終了するため、変換されたソースコードがクラスファイル化に成功するということは、従来の Java コンパイラにも処理を成功させるような変換がおこなわれているということである。なぜなら本研究では Java の Frontend のみを拡張し、Backend には一切手を加えていないからである。我々がおこなったのは、refine クラスによる拡張を従来の Java スタイルに自動変換するような処理を追加定義したことであり、そのため本言語は Java が本来持つ型の安全性といったものを破壊することはない。

第5章　実験・議論とまとめ

本章ではこれまで述べた提案言語と代表的な AOP 言語である AspectJ との差分について述べ、最後に本研究のまとめをおこなう。また、提案言語と従来の Java の差異についても記述する。

5.1　実験

本研究では、提案言語における 2 種類の dispatch についての実行速度を観測した。2 種類の dispatch とは、predicate dispatch と、caller 情報による dispatch である。

Java は元来、single dispatch の機能を持つため、比較の基準はこの single dispatch である。single dispatch に加えて引数の実行時の型情報による dispatch をおこなったのが predicate dispatch である。さらに caller 情報の実行時の型による dispatch をおこなったのが client dispatch である。この実験では、caller 情報の参照だけでなく、登録に関する処理も時間に含めておこなった。

dispatch の種類	1 回あたりの実行速度
single dispatch	0.449 (nano 秒)
predicate dispatch	0.682
client dispatch	3.329

実験結果として、predicate dispatch は single dispatch の 1.52 倍、client dispatch は 7.41 倍となった。caller 情報の操作が重いのは、情報を保持するための ThreadLocal オブジェクトへのアクセスが遅いためである。ThreadLocal の getter や setter は常に自身のスレッド番号のチェックをおこなうため、このような差が生じる。

5.2　AspectJ との比較

第 2 章で述べた問題は AspectJ でも解決可能である。2 章では、以下のような User クラスにデバッグ処理や例外処理を追加したい際の問題点を述べた。

```
class User {
  private String name;
  public String getName() { return name; }
  public void setName(String s) { name = s; }
  public void initName() { setName("default"); }
}
```

getName に関するログ取得、setName の例外処理、initName の呼び出し箇所のファイル出力といった処理は、AspectJ では以下のように記述することができる。

```
import java.util.*;
aspect Aspect {
  private ArrayList User.logList = new ArrayList();
  public String User.getNameCopy() { return getName(); }
  before(User user) : target(user) && execution(public
                      String User.getName()) &&
                      !cflow(call(public String
                      User.getNameCopy())) {
    user.logList.add(user.getNameCopy().trim());
  }
  public void User.printLog() {
    for(int i = 0; i < logList.size(); i++) {
      System.out.println("log:" + logList.get(i));
    }
  }
  void around(String s) : args(s) && execution(public void
                          User.setName(String)) {
    if(s == null) {
      System.out.println("args is null");
    }
    else if(s.length() < 3) {
      System.out.println("new name is too short");
    }
    else
      proceed(s);
  }
  before(Object client) : this(client) && call(public
```

```
                          void User.initName())
                          && !within(User) {
    System.out.println(client.getClass().getName() +
                       " calls User.initName()");
  }
}
```

これと同等の処理を提案言語では refine クラスを用いることで次のように書ける。

```
import java.util.*;
class Ref refines User {
  private static ArrayList logList = new ArrayList();
  public String getName() {
    logList.add(name.trim());
    return name;
  }
  public void printLog() {
    for(int i = 0; i < logList.size(); i++) {
      System.out.println(logList.get(i));
    }
  }
  public void setName(String s) when s == null {
    System.out.println("args is null");
  }
  public void setName(String s) when s.length() < 3 {
    System.out.println("new name is too short");
  }
  public void initName() when !(client instanceof User) {
    System.out.println(client.getClass().getName() +
                       " calls User.initName()");
    proceed();
  }
}
```

AspectJ では、aspect、インタータイプ宣言、アドバイス、ポイントカットといった構文や概念が必須である。一方で提案言語では、習得すべき構文は少なく、また、概念は従来の Java を拡張したものとして考えれば理解し易い。

AspectJ における aspect は、refine クラスに似ている。また、AspectJ のインタータイプ宣言は、メンバの add, replace と同等のことができる。AspectJ はメンバ毎に追加先のクラスを宣言するが、一方で refine クラスは追加先のクラスを予め refines 節で定義しておくという違いがある。

aspect と refine クラス

```
aspect Aspect { int Sample.x = 0; }
class Ref refines Sample { int x = 0; }
```

AspectJ と提案言語で特に差が見られるのが、アドバイスとポイントカットの定義である。提案言語ではアドバイス、ポイントカットといった概念に相当するものはない。なぜなら AOP はジョインポイントというプログラム中の様々な「タイミング」に処理を追加するという思想の元に設計されており、一方で提案言語は従来の Java の拡張という思想の元に設計されているからである。

例えば AspectJ では call と execution という 2 種類のポイントカットがある。これらは両者とも任意のメソッド定義を引数として指定することができる。2 つの違いはメソッドを「call した時 (caller 側)」か「execution した時 (callee 側)」かということである。本言語ではメソッド呼び出しにこのようなタイミングの区分けをおこなってはいない。caller 側の情報が必要であるなら client 参照を、callee 側の情報が必要であるなら従来の Java のように this アクセスをおこなえばよい。また、AspectJ では caller や callee のインスタンスを取得する際には専用のポイントカット指定子が要求され、さらにアドバイス指定子の引数としても宣言していなければならない。this ポイントカットや target ポイントカットについても理解しなければ AspectJ は使いこなせない。

AspectJ には before や after アドバイスがあるが、本言語ではこれに該当する機能として「メンバ replace + proceed 呼び出し」がある。proceed 呼び出しは AspectJ における around アドバイスにも同等の概念が存在する。我々が提案するのは従来の Java と同様のメンバ宣言に proceed の概念だけを組み合わせたものである。

提案言語と AspectJ の大きな違いは、提案言語には「ソースの記述そのものが拡張したい対象クラスの中にコピーされる」という概念があり、AspectJ には「アスペクトが拡張したいクラスを拡張する」という概念がある。例えば以下の refine クラスと aspect は同じ処理とはならない。

```
class Ref refines User {
  private static ArrayList logList = new ArrayList();
  public String getName() {
    logList.add(name.trim());
```

```
    return name;
  }
}
aspect Aspect {
  private ArrayList User.logList = new ArrayList();
  before(User user) : target(user) && execution(public
                      String User.getName()) {
    user.logList.add(user.name.trim());
  }
}
```

上記の場合、refine クラスは問題なくコンパイルに成功する。しかし aspect はそうはならない。なぜなら user.name というアクセスは name が private であり許可されないためである。これは、aspect 内のアドバイスの記述はあくまでその aspect が持つ記述であり、外部クラスとは関係がないとされるためである。User.getName() を execution するタイミングであることが明示されているにも関わらず、aspect による拡張にはこのような制約が課せられる。さらに user.logList のほうは logList というメンバ変数が Aspect から追加されているとコンパイラが判断するため、private であるにも関わらずコンパイルエラーにはならない。logList は name と一見同じに見えるが、アクセスにはこのような差異が生じる。これは決して直感的なコードとはいえない。(コンパイルエラーを抑えるためには、name ではなく getName() へアクセスさせれば良いが、そのときは getName() 実行のタイミングが before アドバイスにも含まれてしまうため、実行時に無限ループとなり StackOverFlow が発生してしまう。また、もう一つの方法として特権アスペクトを用いる手があるが、ソース自体が直感的でないという問題が残る) 一方で提案言語では、logList と name に差異は無い。Ref.getName() はコンパイラにより User.getName() に定義されているものとして振舞うため、name へのアクセスが拒否されることはない。

以上のように提案言語と AspectJ は、従来の Java よりも横断的関心事を扱い易いという共通点があるが、構文や制約の理解に大きな差が生じる。提案言語では refines 節、when 節、client、proceed の 4 種の構文と概念を新たに習得すればよい。AspectJ では追加構文の量や、多くのポイントカット指定子（call, execution, this, target, args, within, cflow, initialization 等)、そして特別な意味を持つ変数（proceed, thisJoinPoint, thisJoinPointStaticPart 等）のように、覚えるべき量が提案言語よりも多く概念も Java とは大きく離れているという欠点が挙げられる。

5.3　まとめ

本研究では、横断的関心事を扱うための拡張 OOP 言語を提案した。この言語は従来の Java をベースに、refine クラスと条件付きメソッド、client 変数を導入した。refine クラスは任意のクラスを選択し、自身の持つメンバを対象のクラスに定義させることができる。これにより提案言語は OpenClass の機能を持つ。また、条件付きメソッドを対象のクラスに refine クラスから定義させることで、method dispatch をより細かい条件に対応させた。さらに条件文には method の caller インスタンスへの参照を持たせることで、グローバルなコンテキストを dispatch の条件として選択させることができる。これらの技術により、提案言語では関心事の追加・削除をクラスに直接修正を加えることなくできる。また、モジュールをさらに細かい条件に沿って分離させられる。caller サイドに散在する処理を callee 側でまとめることも可能になったため、横断的関心事を 1 箇所に記述できる。

実装には Java ベースのオープンソースコンパイラ作成ツールである JastAdd を用いた。JastAdd は構文解析器から得られた AST の各ノードに対応する Java クラスを定義し、ユーザはその挙動を拡張することでコンパイラを自由に作成できる。各ノードは属性を持ち、Java クラスではそれらの属性はクラスメンバとして定義される。そのため、ユーザは属性を Java プログラミングの感覚で扱うことができる。

提案言語は AOP と異なるアプローチをおこなっている。両者とも横断的関心事を扱うための技術であるが、AOP がジョインポイントを提唱しているのに対し、提案言語はあくまで Java の拡張として捉えている。提案言語は AspectJ と比べて複雑な概念を必要とせず、追加構文の総数も少ない。そのため提案言語は AOP 言語と比べて習得のコストが低いという利点がある。

参考文献

[1] AspectJ: http://www.eclipse.org/aspectj/.

[2] beaver: http://beaver.sourceforge.net/.

[3] Clifton, C. and Leavens, G. T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java, *In OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 130–145 (2000).

[4] Ekman, T. and Hedin, G.: Modular name analysis for Java using JastAdd, *Generative and Transformational Techniques in Software Engineering*, LNCS, Vol. 4143, Springer (2006).

[5] Ekman, T. and Hedin, G.: The JastAdd Extensible Java Compiler, *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pp. 1–18 (2007).

[6] Ekman, T. and Hedin, G.: The JastAdd system — modular extensible compiler construction, *Sci. Comput. Program.*, Vol. 69, No. 1-3, pp. 14–26 (2007).

[7] Hedin, G. and Magnusson, E.: JastAdd–an aspect-oriented compiler construction system, *Science of Computer Programming*, pp. 37–58 (2003).

[8] JastAdd: http://jastadd.cs.lth.se/web/projects/index.shtml.

[9] JastAdd: http://jastadd.org/.

[10] javacc: https://javacc.dev.java.net/.

[11] jflex: http://www.jflex.de/.

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-oriented programming, pp. 220–242 (1997).

[13] Millstein, T.: Practical predicate dispatch, *SIGPLAN Not.*, Vol. 39, No. 10, pp. 345–364 (2004).