差分ベースモジュール：クラス独立なモジュール機構

産業技術総合研究所テクニカルレポート AIST01-J00002-1
一杉 裕志*、田中哲†

2001 年 12 月 6 日

本論文では、差分ベースモジュールと呼ぶモジュール機構と、それを Java 言語に適用したオブジェクト指向言語 MixJuice について述べる。モジュールとは、情報隠蔽、再利用、分割コンパイルの単位である。差分ベースモジュールは、クラスとモジュールの機構を完全に分離し、その代わりにモジュールと差分プログラミングの機構を統一したモジュール機構である。この機構は Java のモジュール機構に比べてシンプルな上、プログラムの拡張性・再利用性・保守性をより向上させる。また、情報隠蔽・再利用の単位を、コラボレーションのような、クラスとは独立した単位で設定することができる。個々のモジュールは mixin のように組み合わせることができるが、このとき、名前の衝突と、実装欠損と呼ぶ現象が生じる可能性がある。これらの問題の解決策についても述べる。我々は、MixJuice を用いてすでにトータルで 2 万行以上のアプリケーションを記述している。

1 イントロダクション

モジュールとは情報隠蔽・再利用の単位であり、クラスとはオブジェクトの雛型である。これら 2 つは本来は異なった概念である。しかし、 C++, Java などのオブジェクト指向言語は、“class” という構文にモジュールとしての役割を持たせている。このようなモジュール機構を**クラスベースモジュール**と呼ぶことにする。一般にはクラスは情報隠蔽・再利用の単位として不適切であり、クラスベースモジュールには問題がある。

クラスが情報隠蔽の単位としては不適切であることは [31] で指摘されている。クラスが情報隠蔽の単位として適切なのは、簡単な抽象データ型に対してだけである。複数のクラスが協調して 1 つの機能を実現する場合には、クラスは情報隠蔽の単位には適さない。この問題を少しでも和らげるために、C++ には friend, protected, namespace などの機構が、Java には package, nested class[12] などの機構が導入された。しかし、これらの機構が導入されたとしても、クラスベースモジュールには根本的な問題がある。クラスが高機能化すると、必然的にクラスのサイズは増大する。しかしこれは同時に、スコープのサイズが肥大化していくことを意味し、システムの保守性を悪くする原因になる。例えば Java の標準ライブラリの中のクラス TreeMap のソースコードはコメントなしで 1000 行（内部クラスを含む）にもなり、ソースコードの変更が、ファイル内の他のどこに影響を及ぼすかを把握することが非常に困難になっている。

また、クラスは再利用の単位としても不適切である。一般にある特定の機能に関連したコードは、複数のクラスにまたがって存在し得る [20]。プログラムの再利用性を向上させるためには、このようなコードも分離して記述できるような機構を、プログラミング言語が提供する必要がある。実際にそのようなプログラミング言語がいくつか提案されている。例えば AspectJ[19], Hyper/J[26], Demeter/Java[22], DJ[25] などである。

クラスではなく**コラボレーション**を再利用の単位とする研究もいくつかある [13][33] [28][17]。ここで、コラボレーションとはある機能に関連し、複数のクラスに属するフィールドおよびメソッドの集合である（図 1）。

コラボレーションを再利用の単位とするプログラミングを行なうためには、プログラミング言語が mixin

*科学技術振興事業団 さきがけ研究 21 ／ 産業技術総合研究所 情報処理研究部門, y-ichisugi@aist.go.jp, http://staff.aist.go.jp/y-ichisugi/
†産業技術総合研究所 情報処理研究部門, akr@m17n.org, http://cvs.m17n.org/~akr/

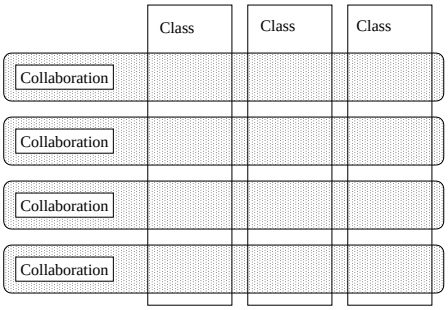

図 1: クラスとコラボレーション

かそれに類する機構を有している必要がある。mixinとは、ある機能を提供するクラスの断片である。既存の mixin を複数組み合わせて、新しいクラスを定義する際に利用することができる。mixin は、 CLOS[29]のように多重継承とクラスの linearize を行なう言語でのプログラミングでは普通に使われている。mixinを使うことにより、プログラムの再利用性を高めることができる。

VanHilst と Notkin の方法 [33] は C++ の templateの機構を使って mixin の機構を実現するもので、これによりプログラムの部品の再利用性を高めている。Mixin layers[28] も同様に template を用いたプログラミング手法であるが、collaboration-based design に基づくプログラミングをより容易にするために、個々の mixin のうち機能的に関連の深いものをひとまとめにして扱うものである。

また、我々はこれらの研究とは独立に、SystemMixin [17] と呼ぶ、 mixin layers と同様の機構を Java 言語上で実装している。そして、この機構を用いて、Java言語のソースコード変換フレームワーク EPP[17, 14]を実装している。EPP は SystemMixin 機構を使って新たなコラボレーションを追加することで、 Java 言語の言語仕様を拡張することができるプリプロセッサである。パーザ、抽象構文木、型システム、名前空間などを扱う重要なクラスがすべて拡張可能な構造をしているため、並列言語 [17]、スレッドマイグレーション [1]、パラメタ付きクラス [14] など幅広い範囲の言語拡張を行なうことができる。

我々は EPP の実装の経験から、クラスではなくコラボレーションを再利用の単位とすることが、EPPのような非常に拡張性の高いプログラムの記述に、特に適していることを確信するに至った。

しかし、 mixin layers も SystemMixin も、情報隠蔽の機構を有していない。しかし実際には、 [13][33]で指摘されているように、コラボレーションは複数のクラスにまたがる invariant を保持する単位であり、再利用の単位だけでなく、情報隠蔽の単位としても適しているはずである。

以上のような背景から、**差分ベースモジュール**と呼ぶ機構を有するオブジェクト指向言語 MixJuice [15] [16] を設計・実装した。差分ベースモジュールは、クラスとモジュールの機構を完全に分離し、その代わりにモジュールと差分プログラミングの機構を統一したモジュール機構である。これにより、従来のクラスベースモジュールの問題を解決し、コラボレーションを情報隠蔽・再利用の単位にすることを可能とする。差分ベースモジュールはシンプルな３つの原理（差分定義の原理、名前空間の継承の原理、名前の衝突回避の原理）に基づいて設計されている。

本論文の以下の構成は次のようになっている。差分ベースモジュールの特徴のうち、2章では差分プログラミング機構としての側面、3章では情報隠蔽機構としての側面から説明する。4章では、複数のモジュールを組み合わせたときに生じる実装欠損の問題と、それを解決する補完モジュールの言語サポートについて述べる。5章ではプログラミング言語 MixJuice のアプリケーション、6章では実装方法に述べ、7章では関連研究、8章では将来の課題について述べる。最後に9章でまとめについて述べる。

2 差分ベースモジュールによる差分プログラミング

2.1 原理と利点

差分ベースモジュールは、下記の原理に従って設計されている。

差分定義の原理：モジュールは、オリジナルのプログラムと拡張後のプログラムとの間の差分である。ここで差分とは、新たな名前の定義の追加と、既存の名前の定義の修正の集合である[1]。

モジュールとは、再利用の単位であり、情報隠蔽の単位であり、分割コンパイルの単位である。複数のモジュールをリンクすることによって、実行可能なアプリケーションが構築される。差分ベースモジュールにおけるリンクとは、「一切の定義を持たない空のプログラム」に対して各モジュールで定義される差分を次々と追加していくことを意味する。

差分ベースモジュールの設計原理は、様々なプログラミング言語に適用可能である。多くのプログラミング言語では、プログラムは、名前とその定義の集合と見なせるが、差分ベースモジュールは、名前とその定義の集合を追加・修正する機構である。例えば手続き型言語のプログラムは手続きやデータ構造の定義の集合であり、Java 言語のプログラムの場合は、クラス、フィールド、メソッドの定義の集合である。

Java 言語に、差分ベースモジュールを適用したものが MixJuice 言語である。つまり MixJuice は、クラス、フィールド、メソッドの定義に対する追加や修正を、モジュールとして記述する言語である。

個々のモジュール同士は継承関係を持ち得る。MixJuice には、モジュールの継承機構と従来のクラス継承の機構の両方が独立に存在する。クラス継承とモジュール継承は、次のように違う。まず、クラス継承は既存のクラスに対する変更部分を記述する機構だが、モジュール継承では、既存のプログラム全体に対する変更部分を記述する機構である。また、クラス継承では、すでに存在するクラスとは違う名前を持った、新しいサブクラスを定義することしかできないが、モジュール継承では、すでに存在するクラスそのものに対し、その名前を変えることなしに、変更を加えることができる。クラス継承は、subtyping を行なう機構であり、それによりメソッドの動的結合を行なうための機構でもある。一方、モジュール継承は、静的な再利用のための機構であり、また 3章で述べるように情報隠蔽のための機構である。

[1]現在のところ、名前の rename 、定義の削除は含まない。

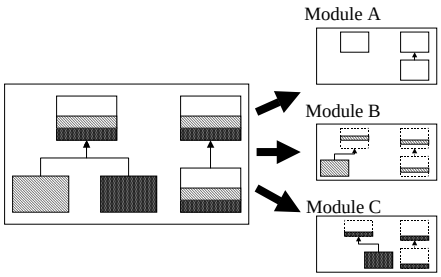

図 2: 複数のクラスをまたがるコードの分離

差分ベースモジュール機構においては、クラスはもはやモジュールとしての機能を持たない。すなわち、クラスは情報隠蔽の単位、再利用の単位、分割コンパイルの単位ではない。

差分ベースモジュールには、従来のクラスベースモジュールと比べて、以下のような利点がある。

1. 「アプリケーションの高い拡張性」
   差分ベースモジュールは、プログラマーが拡張性の高いアプリケーションを記述することを容易にする。

   これは２つの理由による。１つの理由は、すべてのクラス名・メソッド名が、アプリケーションを拡張するプログラマーのための “hook” として働くからである。もう１つの理由は、機能を拡張するモジュールが、 mixin と同様に、モジュールの多重継承を使って複数組み合わせられるという性質を持つからである。（詳しくは 2.3節で述べる。）

2. 「再利用単位のクラス独立性」
   再利用の単位を、クラスという単位とは完全に独立して設定することが可能となる。これにより、コラボレーションなど、複数のクラスにまたがるコードを再利用の単位にできる（図 2）。

3. 「第三者によるアプリケーションの拡張性」
   差分ベースモジュールを使って書かれた任意のアプリケーションに対し、第三者のプログラマーが「拡張モジュール」を提供することが可能になる。これは、従来の “patch file” の作成に相当す

る。ただし patch は文字列レベルで差分の追加を行なうのに対し、差分ベースモジュールでは、言語レベルで差分の記述・追加の処理を行なうため、より信頼性が高い。また、拡張モジュールの実装者はオリジナルのアプリケーションのソースコードを持っている必要がなく、拡張のためのインターフェースを知っているだけでよい。（詳しくは 2.5.1節で述べる。）

4. 「エンドユーザによるモジュールの結合可能性」
   エンドユーザは、様々な機能を実現する複数のモジュールのうち、自分が必要とする機能を実現するモジュールだけを選択し、組み合わせることで、自分独自のアプリケーションを構築することができる。この時、エンドユーザは、複数のモジュールを組み合わせる “glue code” を一切書く必要がない。エンドユーザは使いたいモジュール名の集合を指定するだけで、モジュールをリンクできる。（具体的には 2.5.3節で述べる。）

5. 「モジュールのグループ化の柔軟性」
   差分ベースモジュールでは、複数のモジュールをグループ化し、名前をつけることが可能である。Java 言語は package 機構と “import p.*;” という宣言文により、ある程度クラスライブラリをグループとして扱うことが可能だが、差分ベースモジュールはそれよりもはるかに柔軟なグループ化が可能である。（具体的には 2.3節で述べる。）

以下の節では、 MixJuice の差分ベースモジュールによる差分プログラミングの詳細について述べる。

2.2 モジュール定義の構文

モジュールの定義は図 3のように記述する。（正確な文法定義は付録 Aを参照されたい。）図 3では、モジュール m1 とモジュール m2 を定義している。

モジュール定義の先頭の “extends” 宣言は、そのモジュールが差分を追加する対象となるモジュール名を宣言する。従来のオブジェクト指向言語におけるスーパークラスに似ているため、指定されたモジュールを super-module と呼ぶ。図 3の例では、 m2 は、 m1 を super-module として指定している。またこの時、「m2 は m1 の sub-module である」、あるいは「モジュール m2 は m1 を継承する」と言う。

```
module m1 {
   define class S {
     define S(){}
     define int foo(){ return 1; }
   }
   define class A extends S {
     define A(){}
     int foo(){ return original() + 10; }
   }
   class SS {
     void main(String[] args){
       A a = new A();
       System.out.println(a.foo());
     }
   }
}
module m2 extends m1 {
   class S {
     int foo(){ return original() + 2; }
   }
   class A {
     int foo(){ return original() + 20; }
   }
}
```

図 3: モジュール定義の構文

```
class $S1$ {
  int foo(){ return 1; }
}
class S extends $S1$ {
  int foo(){ return super.foo() + 2; }
}
class $A1$ extends S {
  int foo(){ return super.foo() + 10; }
}
class A extends $A1$ {
  int foo(){ return super.foo() + 20; }
}
class SS {
  void main(String[] args){
    A a = new A();
    System.out.println(a.foo());
  }
}
```

図 4: モジュール m2 に相当する Java プログラム

モジュール m1 のように、 “extends” 宣言を持たないものは、「空のプログラム」に対する差分を定義するモジュールである。

モジュール本体（中かっこの内部）には、オリジナルのプログラムに対する差分を記述する。具体的には、モジュールは、その super-module が表すプログラムに対して以下の変更を加えることができる。

- 新たなクラス[2] の追加
- 既存のクラスに対する新たなフィールドの追加
- 既存のクラスに対する新たなコンストラクタ・メソッドの追加
- 既存のコンストラクタ・メソッドの override

図 3の例では、モジュール m2 は、モジュール m1 で定義されたクラス S のメソッド foo と、クラス A のメソッド foo をそれぞれ override して、その機能を拡張している。

[2]本論文中では、インターフェースに関する明示的な説明はすべて省略する。実際の MixJuice 言語ではクラスと同様にインターフェースも追加・修正が可能である。

モジュール本体の文法は Java 言語とほぼ同じだが、以下の点が違う。

まず、 “define” というキーワードを指定する場合がある点が異なる。“define” というキーワードの付いたクラス・フィールド・コンストラクタ・メソッド定義は、新たなクラス・フィールド・コンストラクタ・メソッドを追加することを示す。“define” がついていないクラス・コンストラクタ・メソッド定義は、既存のクラス・コンストラクタ・メソッドの定義を拡張することを示す。（クラス SS とそのメソッド main はシステムにより最初から定義されているため、“define” は不要である。）

“original()” という式は、 override されたメソッドを、override しているメソッド内から呼び出す場合に使う。これは Java 言語における super へのメソッド呼び出しに相当する。MixJuice における override には２種類ある。１つはクラス継承による override であり、もう１つはモジュール継承による override である。図 3の場合、モジュール m1 内のクラス A の定義内で呼び出している original() が前者であり、モジュール m2 内の２箇所の original() が後者である。

package 機構や、public/protected/private というキーワードは、 MixJuice では用いない。MixJuice において情報隠蔽がいかに表現されるかについては 3章で述べる。

モジュール m2 で定義される MixJuice プログラムは、ほぼ図 4の Java プログラムに相当する。

2.3 モジュールの多重継承

モジュールを定義するとき、 super-module を複数指定することで、モジュールの多重継承を行なうことができる。

図 5は、モジュールの多重継承の例である。m3 は、m2 と同様、 m1 に対する差分を定義するモジュールである。m4 は、m2 と m3 の両方を継承するモジュールである。ここで、 m2 と m3 は共に m1 を super-module に持つので、いわゆるダイアモンド継承の形になる。

各モジュールは、topological sort によって linearize される[3]。これは、 CLOS[29] などの言語で、クラス

[3]現在の実装では、“extends” 宣言における super-module の指定の順序は、linearize の結果に影響を与えない。つまり、m2 と m3 の間の上下関係をプログラマーが指定する方法はない。linearize には”monotonic” [6][3] なアルゴリズムを採用しているが、現在

```
module m3 extends m1 {
   class S {
     int foo(){ return original() + 3; }
   }
   class A {
     int foo(){ return original() + 30; }
   }
}
module m4 extends m2, m3 {
   class S {
     int foo(){ return original() + 4; }
   }
   class A {
     int foo(){ return original() + 40; }
   }
}
```

図 5: モジュールの多重継承

の多重継承の時に各クラスが linearize されるのと同じである。

例えば、モジュール m4 で定義されるプログラムの解釈は、次のように行われる。まず、 m4 自身とその先祖モジュールの集合を求める。それは {m1, m2, m3, m4} である。そして、この集合を super-module/sub-module の上下関係を保存するように topological sort することによって linearize する。linearize した結果を、 linearized list と呼ぶ。この場合、 linearized list は (m1 m2 m3 m4) になる。そして、 linearized list の先頭から順に、空プログラムに対して差分を追加していき、できあがったプログラムを実行する。つまり、プログラム a に対して差分 b を追加した結果を $a \triangleleft b$ と表記するなら、

$$((((\phi \triangleleft m1) \triangleleft m2) \triangleleft m3) \triangleleft m4)$$

を実行する。このような解釈の結果、モジュール m4 に相当する Java プログラムは図 6のようになる。

一般に多重継承の最大の問題点は、名前の衝突であるが、 MixJuice ではこの問題は完全に解決されている。これについては 3.7章で説明する。

モジュールの多重継承は、複数のモジュールのグルーピングにも用いることができる。次のように任意

の実装ではそのためにモジュールの名前の情報を用いている。

```
class $S1$ {
  int foo(){ return 1; }
}
class $S2$ extends $S1$ {
  int foo(){ return super.foo() + 2; }
}
class $S3$ extends $S2$ {
  int foo(){ return super.foo() + 3; }
}
class S extends $S3$ {
  int foo(){ return super.foo() + 4; }
}
class $A1$ extends S {
  int foo(){ return super.foo() + 10; }
}
class $A2$ extends $A1$ {
  int foo(){ return super.foo() + 20; }
}
class $A3$ extends $A2$ {
  int foo(){ return super.foo() + 30; }
}
class A extends $A3$ {
  int foo(){ return super.foo() + 40; }
}
class SS {
  void main(String[] args){
    A a = new A();
    System.out.println(a.foo());
  }
}
```

図 6: モジュール m4 に相当するプログラム

の複数のモジュールを多重継承して新たな名前をつけることで、他のモジュールから同時に継承することを容易にすることができる。

```
module m_x extends m_a, m_b, m_c, m_d {}
```

この方法は、Java 言語の “import p.*;” という構文を用いた、モジュール名に基づくグルーピングよりも強力である。多重継承を用いる方法では、グループのメンバーを任意に選ぶことが可能である。また、複数のグループをさらに多重継承することにより、階層的なグルーピングも行なうことができる。

2.4 差分ベースモジュール独自のプログラミングスタイル

従来のクラスベースモジュールではモジュラリティの高い形に書けなかったプログラムであっても、差分ベースモジュールを用いることによって、モジュラリティの高い形に書ける場合がある。この節では３つ例を挙げて説明する。

差分ベースモジュールにより、木構造に対する traverse の処理を、独立したモジュールとして記述することが可能になる。木構造に対する traverse を素直にオブジェクト指向的に実装すると、各ノードを表すクラスのメソッドを定義するというスタイルになる。しかし従来のオブジェクト指向言語では、このスタイルでは、traverse 処理だけを別のモジュールとして記述することができない。Visitor pattern[10] を用いれば traverse 処理のモジュール化が可能になるが、しかし逆に木構造のノードの種類を追加するためにはソースコードの編集が必要になってしまう。差分ベースモジュールでは、 traverse を処理するメソッドだけを別のモジュールに分離して記述することが可能になる。また、ソースコードを編集しなくても、新たな traverse の処理と新たなノードの両方が追加可能である。

図 7のようなネストした if 文を、図 8のように複数のモジュールに分割して記述することも可能である。このように記述することにより、ソースコードを編集することなしに、新たな条件分岐節を追加することができる。この手法により、再帰下降パーザを分割して拡張可能な構造にすることができる [17]。XML を処理するプログラムもネストした if 文を含む場合が多

```
class F {
  void branch(String s){
    if (s.equals("a")){
      ...
    } else if (s.equals("b")){
      ...
    } else {
      throw new Error();
    }
  }
}
```

図 7: ネストした if 文

```
module framework {
  define class F {
    define void branch(String s){
      throw new Error();
    }
  }
}
module case_a extends framework {
  class F {
    void branch(String s){
      if (s.equals("a")){ ... }
      else { original(s); }
    }
  }
}
module case_b extends framework {
  class F {
    void branch(String s){
      if (s.equals("b")){ ... }
      else { original(s); }
    }
  }
}
```

図 8: ネストした if 文の分割

```
class DataManager {
  java.util.Hashtable table
    = new java.util.Hashtable();
  void initTable(){
    table.put("A", new A());
    table.put("B", new B());
    ...
  }
}
class A {...}
class B {...}
...
```

図 9: テーブルの初期化コード

いが、この手法により拡張性の高い構造にすることが可能である。

テーブルなどの初期化コードも、従来のオブジェクト指向言語では 1 箇所に集中しがちであったが、差分ベースモジュールでは複数のモジュールに分割して記述することが可能である。例えば、図 9の Java プログラムは、図 10 のように複数のモジュールに分割することができる。また、テーブルの初期化メソッドがそのまま拡張モジュールのための hook として働くため、新たなテーブルのエントリの追加が、ソースコードを修正しなくても、モジュールの追加によって可能になる。

2.5 モジュールの実行環境

MixJuice は、コンパイル・リンク・実行の方法に関して、従来のプログラミング言語にない特徴を有する。この節では、これらについて説明する。

2.5.1 モジュールの分割コンパイル

MixJuice では各モジュール毎に分割コンパイルが可能である。モジュールの内部にはクラスの不完全な断片が含まれるが、コンパイル時にはその断片に関して型チェックが行なわれる。モジュールのコンパイルには、そのモジュールの先祖モジュールのみが必要である。なお、先祖モジュールのソースが存在していな

```
module dataManager {
  define class DataManager {
    define java.util.Hashtable table
      = new java.util.Hashtable();
    define void initTable(){}
  }
}
module classA extends dataManager {
  class DataManager {
    void initTable(){
      original();
      table.put("A", new A());
    }
  }
  define class A {...}
}
module classB extends dataManager {
  class DataManager {
    void initTable(){
      original();
      table.put("B", new B());
    }
  }
  define class B {...}
}
...
```

図 10: 初期化コードの分割

くても、コンパイル結果のバイナリ[4]がコンパイラから参照可能であればよい。

分割コンパイルの単位がクラスと直交していることは、現実のアプリケーション開発に適している。プログラムの開発・テストはコラボレーション単位で行なう場合が多い。差分ベースモジュールでは、複数のコラボレーションを、独立したチームで開発・テストすることを可能にする。

2.5.2 モジュールのリンクと実行

MixJuice で書かれたプログラムを実行するためには、必要となるすべてのモジュールをリンクする必要がある。現在我々が配布している MixJuice のパッケージには、リンクと実行を同時に行なう mj コマンドが用意されている。

モジュールを実行するには、実行したいモジュール名を mj コマンドの引数に渡す。mj コマンドは指定されたモジュールをリンクして実行可能な Java プログラムに変換し、 JavaVM 内にロードして実行する。mj コマンドは、クラス SS の main を呼び出す。

これは、 m1, m2, m3, m4 の各モジュールの実行例である。

```
% mj m1
11
% mj m2
33
% mj m3
44
% mj m4
110
```

mj コマンドは、引数で指定されたモジュールのすべての先祖モジュールを自動的にリンクして、実行する[5]。例えば、モジュール m2 を実行する時に、必要なモジュール m1 の名前は、mj コマンドの引数に指定する必要がない。ただし、必要なモジュールが見付からなかった場合はリンク時エラーとなる。

mj コマンドは、先祖モジュールの自動リンク以外に、補完モジュールと呼ぶ、もう１つ別の種類のモジュールを自動的にリンクする。これについては 4章で述べる。

2.5.3 モジュールの選択

アプリケーションのエンドユーザは、異なるプログラマーによって開発された複数のモジュールを選択し、コードを１行も書かずに組み合わせて使うことができる。

例えば m2 と m3 は m1 に対する全く独立した差分だが、エンドユーザが、２つの差分を両方同時に m1 に追加することができる。それには、 mj コマンドの “-s” オプションを使い、追加するモジュールを指定する。

```
% mj -s m2 m3
66
```

mj コマンドは、指定された全ての module を extends する仮想的なモジュールを、”_bottom” という名前で作る。つまり、この場合、次のようなモジュールを作る。

```
module _bottom extends m2, m3 {}
```

そして、あたかも ”mj _bottom” というコマンドが実行されたかのように処理する。すなわち、

$$(((\phi \triangleleft m1) \triangleleft m2) \triangleleft m3)$$

というプログラムを実行することになる。

“-s” オプションは複数指定することができ、任意の個数のモジュールを結合することができる。ただし、１つのモジュールを何度も繰り返し追加することは、現在の MixJuice ではできない。

モジュール結合の機能により、エンドユーザは、様々な機能を実現する複数のモジュールのうち、自分が必要とする機能を実現するモジュールだけを選択し、組み合わせることで、自分独自のアプリケーションを構築することができる。従来は同様のことは、条件コンパイル (#ifdef) や、patch のような外部ツールによって行なっていた。しかし差分ベースモジュール機構では、文字列レベルではなく言語レベルで処理を行なうので、より信頼性が高い。また、モジュール結合にソースコードを必要としないため、あるアプリケーションを拡張するモジュールを、ソースコードを公開せずに配布することができる。

[4]現在の実装では、モジュールのコンパイル結果は、Java の class file の集合という形をしている。

[5]現在の実装では、必要なモジュールは CLASSPATH の中から検索される。

3 差分ベースモジュールにおける情報隠蔽

この章では、MixJuice のモジュール機構が情報隠蔽の観点から見て、Java 言語のモジュール機構よりも強力であることを述べる。

3.1 原理と利点

差分ベースモジュールは、名前の可視性に関しては、下記の原理に従って設計されている。

> **名前空間の継承の原理：**あるモジュール内で定義されるすべての名前は、そのモジュール自身とそのモジュールの子孫モジュールから参照可能である。そして、それらのモジュール以外からは参照することはできない。

ここで言う「名前」とは、具体的にはクラス名・フィールド名・メソッド名を指す。MixJuice のモジュール機構は、可視性に関するこのシンプルなルールだけで、Java 言語よりも柔軟な名前空間制御を可能にする。

クラスは、もはやソースコード上での情報隠蔽の単位ではない。クラスのフィールドは、同一のモジュールまたは子孫モジュール内であれば、たとえ別のクラスからであっても、アクセス可能である。差分ベースモジュール機構には、public, protected, private というキーワードも package, nested class の機構も存在しない。

差分ベースモジュール機構は、Java 言語の情報隠蔽機構に比べて以下の利点がある。

1. 「情報隠蔽単位のクラス独立性」
   クラスという単位とは完全に独立して、自由に情報隠蔽の単位を設定することができる。例えばコラボレーションを情報隠蔽の単位（かつ再利用の単位）にできる。（詳しくは 3.6で述べる。）また、プログラマーは、必要最小限の名前空間のみを見えるようにすることで、保守性を高めることができるようになる。このことは、クラスが高機能化し、クラスのサイズが巨大化する時に特に有効である。（具体例は 5.3節で述べる。）

2. 「名前空間構造の柔軟性」
   名前空間のネストや、ネストよりもより一般的な、重なりのある名前空間が表現可能である。これにより、 Java が持っていた nested class の機構を不要にし、言語仕様をシンプルにする。（詳しくは 3.4節および 3.5節で述べる。）

3. 「コード移動の容易性」
   モジュール間のコード移動の自由度が非常に高い。これは、あるコードを super-module や sub-module に移動しても、モジュールのリンク結果のプログラムには全く影響を与えないという、差分ベースモジュールの性質による。これにより、クラスの構造を全く変えなくても、かなりの自由度で、一種のリファクタリング [9] を行なうことができる。例えば複数のクラスが相互依存している場合でも、クラスの構造を変えずに、プログラムを相互依存しない複数のモジュールに整理することができる。（具体的には 3.6節で述べる。）あるアプリケーションのプロトタイプシステムが一枚岩でモジュラリティの悪いプログラムであっても、そのソースコードを書き換えて行くことにより、モジュラリティの高いソースコードへ連続的に移行していくことが容易である。

以下の節では、 MixJuice の差分ベースモジュールによる情報隠蔽の詳細について述べる。

3.2 Black-box reuse

MixJuice では、従来のオブジェクト指向言語と同様に、他のクラスの black-box reuse を行なうことができる。ただし、本論文では、変化しやすい内部実装に依存せず、安定した外部インターフェースのみを利用するプログラミングを black-box reuse であると定義する。

差分ベースモジュールでは、 abstract constructor および abstract method を使って、１つのクラスを、外部インターフェースのみを定義するモジュール（以後、仕様モジュールと呼ぶ）と、実装のみを定義するモジュール（以後、実装モジュールと呼ぶ）に分離することができる。abstract constructor は Java 言語にはない概念だが、コンストラクタのインターフェースだけを分離して定義するためのものである。

図 11は、クラス Point を point という名前の仕様モジュールと point.implementation という名前の実

```
module point {
  define class Point {
    // abstract constructor:
    define abstract Point(int x, int y);
    // abstract methods:
    define abstract void move(int dx, int dy);
    define abstract int getX();
    define abstract int getY();
  }
}
module point.implementation extends point {
  class Point {
    define int x;
    define int y;
    Point(int x, int y){
      this.x = x; this.y = y;
    }
    void move(int dx, int dy){
      x += dx; y += dy;
    }
    int getX(){ return x; }
    int getY(){ return y; }
  }
}
module point.test extends point {
  define class Test {
    define void test(){
      Point p = new Point(1, 2);
      p.move(10, 10);
      ...
    }
  }
}
```

図 11: 仕様モジュールと実装モジュール

```
module colorPoint
  extends point {
  define class ColorPoint extends Point {
    define abstract ColorPoint(Color c,
                               int x, int y);
  }
  define class Color {...}
}
module colorPoint.implementation
  extends colorPoint, point.implementation {
  class ColorPoint {
    define Color c;
    ColorPoint(Color c, int x, int y){
      super(x, y);
      this.c = c;
    }
    ...
  }
}
```

図 12: ColorPoint クラス

装モジュールに分けて定義する例である[6]。モジュール point 内ではクラス Point のコンストラクタとメソッドのインターフェースを、 abstract constructor と abstract method を使って定義している。モジュール point.implementation では、それらに対する具体的な実装を与えている。

他のモジュールは、クラス Point の仕様モジュールのみを継承することで、black-box reuse を行なうことができる。図 11では、モジュール point.test がモジュール point のみを継承することで、black-box reuse を行なっている。

3.3 White-box reuse

MixJuice では前節で述べた black-box reuse に加えて、white-box reuse を行なうことも可能である。ただし、本論文では、あるクラスの外部インターフェースだけでなく、内部実装も利用したプログラミングを white-box reuse であると定義する。

[6]point.implementation という名前はモジュール名に “.” という文字を含んでいるが、これは長いモジュール名を区切るための文字にすぎない。モジュール名の階層構造が言語仕様的に意味を持つことはない。

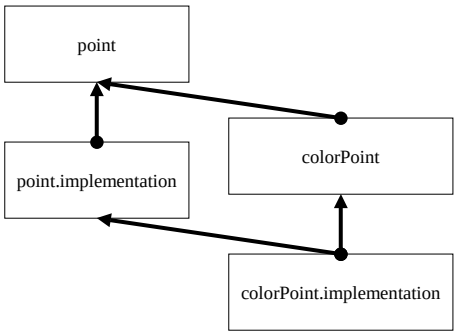

図 13: point と colorPoint のモジュール継承図

Java 言語は、 protected というアクセス修飾子を用いることで、black-box reuse と white-box reuse の両方を可能にしている。内部実装を protected と宣言することにより、subclass に対しては内部実装を公開し、subclass 以外に対しては内部実装を非公開にすることができる。

一方 MixJuice では、実装モジュールを extends することによって white-box reuse を行なう。protected というアクセス修飾子は用いない。図 12は、前節の図 11のクラス Point を継承するクラス ColorPoint の定義例である。ここで、モジュール colorPoint.implementation が、モジュール colorPoint だけでなくモジュール point.implementation も継承している点に注意して欲しい（図 13）。こうすることにより、クラス ColorPoint の実装において、クラス Point のフィールドにもアクセスすることができるようになる。

差分ベースモジュールでは、Java とは異なり、クラスの継承関係とは無関係に、プログラマーが black-box reuse するか white-box reuse するかを自由に選択することができる。例えば、図 11のプログラムでモジュール point.test が point.implementation を継承していれば、それは white-box reuse であり、図 12でモジュール colorPoint.implementation が point.implementation を継承していなければ、それは black-box reuse である。

white-box reuse を行なうと、他のクラスの内部実装が利用できる半面、以下のデメリットがある。まず、依存している他のクラスの内部実装が変更になった時に、自分自身の実装も修正しなければならないというリスクがある。また、他のクラスの内部実装に直接アクセスする場合は、そのクラスの invariant を破らないように、より注意深いプログラミングが必要になる。

現在の MixJuice は、プログラマーが「うっかり」実装モジュールを継承してしまうことを防ぐ機構を持っていない。その代わりに、実装モジュールに対して point.implementation という長いモジュール名を用いるという naming convention を用いることによって、そのような間違いを防いでいる。

```
public class A {
  protected static int x = 0;
  public static class B {
    public int getX(){ return x; }
  }
}
```

図 14: nested class を用いた Java プログラム

```
module A_B {
  define class A { }
  define class B {
    define abstract int getX();
  }
}
module A_B.implementation
   extends A_B {
  class A {
    define static int x = 0;
  }
  class B {
    int getX(){ return A.x; }
  }
}
```

図 15: MixJuice によるネストした名前空間の表現

3.4 ネストした名前空間

モジュールの継承によって、Java が nested class によって実現しているような、ネストした名前空間を

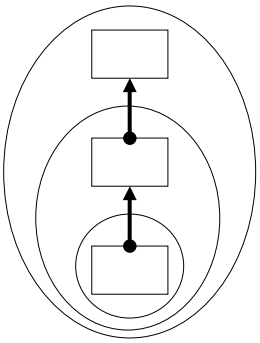

図 16: モジュール継承によってできる名前空間のネスト構造

表現できる。

図 14は、 Java 言語の nested class を用いて、ネストした名前空間を実現する例である。クラス A のフィールド x は public ではないので外部からはアクセスできないが、その内側のクラス B からはアクセスできる。

これと同様のプログラムを MixJuice では図 15のように記述する。モジュール A_B はクラス A, B の public な名前を定義し、モジュール A_B.implementation はクラス A, B の実装と protected な名前を定義している。クラス B は、メソッド getX の中から、クラス A の static field に対し、”A.x” と書いて直接アクセスしている。これは、モジュール内で定義されたすべての名前は、そのモジュール内からアクセスできるという名前空間の原理に従っている。

同様にして、一般に n 重の名前空間のネストは、 n 段のモジュール継承で表現することができる（図 16)。

3.5 重なりのある名前空間

モジュールの多重継承を行なうことにより、重なりがあるような、ネスト構造よりも一般的な名前空間を表現できる。例えば図 3、図 5のようなモジュールのダイアモンド継承は、図 17 のような重なりを持った名前空間を構成する。すなわち、一番下のモジュール m4 は、他のモジュール m1, m2, m3 が作る名前空間の内部に位置することになる。

クラスベースモジュールでは、名前空間の設定に制限が強いため、必要以上に名前のスコープを大きく

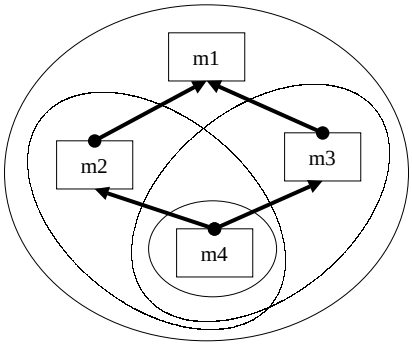

図 17: 多重継承によってできる名前空間

```
class A {  // class A uses class B
  void m1(B b){ ... b.m3(); ...}
  void m2(){...}
}
class B {  // class B uses class A
  void m3(){...}
  void m4(A a){ ... a.m2(); ...}
}
```

図 18: ２つのコラボレーションを含むクラス

せざるを得ない場合が多い。例えば、あるコラボレーションに関連する一部のメソッドにだけアクセスさせたい名前を、public にせざるを得ない場合がある。また、クラスをネストさせると、最も外側のクラスのスコープが大きくなりすぎ、ソースコード間の依存関係が把握しにくくなるという問題がある。

差分ベースモジュールでは、名前のスコープを必要最小限に狭くすることができる。また、ソースコード間の依存関係は、 “extends” 宣言によって明示されるため、プログラマーは常にそれを意識することができる。

3.6 コラボーレーションに基づくモジュール分割

MixJuice ではクラスとモジュールは直交した概念であるため、複数のクラスが関連するコラボレーションを、独立したモジュールに分離することができる。

例えば図 18のプログラムを考える。この例では、クラス A とクラス B がそれぞれ相互に依存している。

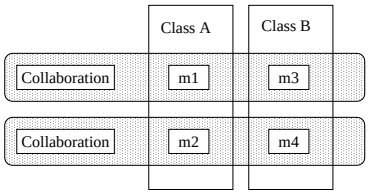

図 19: クラス A と B に含まれる 2 つのコラボレーション

```
module A_B {
  define class A {}
  define class B {}
}
module collaboration_m1_m3
  extends A_B {
  class A {
    define void m1(B b){ ... b.m3(); ...}
  }
  class B {
    define void m3(){...}
  }
}
module collaboration_m2_m4
  extends A_B {
  class A {
    define void m2(){...}
  }
  class B {
    define void m4(A a){ ... a.m2(); ...}
  }
}
```

図 20: コラボレーションのモジュール分割

しかし 2 つのクラスは、実質的には、独立した 2 つのコラボレーションを含んでいる（図 19）。

このプログラムは、図 20のようにモジュール分割することができる。このように記述することで、メソッド m1 と m3 が関係する collaboration_m1_m3 と、メソッド m2 と m4 が関係する collaboration_m2_m4 の 2 つを、全く無関係な 2 つのモジュールに分離することができる。

このようにコード間の依存関係を整理して記述しておくことで、以下のメリットが生じる。

1. それぞれのモジュールが依存するソースコードのサイズが減る。これは一般には、保守性が向上することを意味する。

2. collaboration_m1_m3 と collaboration_m2_m4 は相互に全く依存していないため、他方が存在していなくても、一方をコンパイル・実行することが可能である。このため、別々のプログラマーによる開発・テストが可能になる。

3. それぞれのコラボレーションとは違うバージョンのモジュールを実装することにより、異なるコンフィギュレーションのアプリケーションを構築できるようになる。例えば、 collaboration_m1_m3 の代わりに、 my_collaboration というモジュールを実装して用いることが可能になる。この時、既存のモジュールの再コンパイルは必要がない。

なお、より複雑なプログラムをモジュール分割するより現実的な例は 5.3節で述べる。

3.7 完全限定名

名前空間の多重継承によって起こり得る名前の衝突の問題は、差分ベースモジュールでは完全に解決している。Java 言語では、 import 宣言で生じるクラス名の衝突を解決するためにクラスの完全限定名 (Fully-Qualified-Name, FQN) を使用している。MixJuice はこのアイデアを、フィールド・メソッド名にも適用することで、名前の衝突の問題を解決している。

差分ベースモジュールでは、すべての名前は次に示す原理によって処理される。

名前の衝突回避の原理：すべての名前の定義には、一意に決まる完全限定名が対応づけられる。完全限定名は「その名前を最初に定義したモジュール名」と「単純名」の組で表される。ソースコード中のある地点において、２つ以上の n という単純名が参照可能な時、もしその地点で n という単純名が使われていたら、曖昧であるという理由でコンパイルエラーになる。別々の場所で定義された２つの名前が同一視されたり、片方が他方より優先することは決して起きない。曖昧であるというエラーが起きた場合、プログラマーは単純名の代わりに完全限定名を用いることで、必ずそのエラーを回避することができる。

ただし、モジュール名の一意性は、何らかの規約（ Java 同様、ドメイン名の prefix を用いるなど）によって、保証されているものとする。

MixJuice のソースコード上では、「最初に定義したモジュール名 m 」と「単純名 n」の組で表される完全限定名は、“FQN[m::n]” と表記される[7] 。メソッド完全限定名は、メソッド呼び出しの時だけでなく、メソッドを override する際にも用いることができる。図 21は、モジュール m2 と m3 で別々に定義されたメソッド m を、モジュール m4 で、それぞれ override して、呼び出す例である。

なお、C++ [30] でも c::n という表記で名前衝突の回避を行なうが、セマンティックスは全く違う。C++ では a.c::n() というメソッド呼び出しは virtual funciton call ではないが、MixJuice では a.FQN[m2::m]() という呼び出しは動的結合による普通のメソッド呼び出しである。また、 C++ では、図 21のモジュール m4 が行なっているように、m2::m と m3::m を別々に override することはできない。C++ では、別々に定義されたメソッド名であっても、それらが同一のメソッド名と解釈されるからである。

差分ベースモジュールの情報隠蔽機構の大きな利点は、そのシンプルさである。MixJuice では、名前は super-module から sub-module という１種類の経路のみを通して継承される。一方 Java の場合、仕様はかなり複雑である。例えば単純名でアクセス可能なクラス名には、(1) 同一パッケージ内のクラス、(2)import 文で宣言されたクラス、(3) 自分の outer class のメンバークラス、(4) 自分自身と先祖クラスのメンバークラスという４種類があり得る。これら４種類の間の関係は自明とは言いがたい。

[7] この論文で示した FQN の指定の構文は非常に見苦しいものだが、これは構文解析上の理由による。モジュール名が “.” を含みうると同時に、フィールドやメソッドのアクセスにも “.” が使われる点が問題である。

なお、現実の MixJuice プログラミングにおいては、各名前の有効範囲を従来のオブジェクト指向言語よりも小さくできるため、FQN の指定が必要になることはほとんどない。

4 実装欠損の問題と補完モジュール

4.1 実装欠損

複数のモジュールを組み合わせたとき、実装欠損という現象が起きる場合がある。実装欠損について、以下に詳しく説明する。

図 22のプログラムを考える。m_orig は、abstract なクラス S とそのサブクラス A を定義している。m_sub は、新たなサブクラス B を定義している。一方、 m_abst は、クラス S に abstract method m を定義し、サブクラス A でそれを実装している。m_sub と m_abst は、いずれもリンク時エラーを起こさない、完全なプログラムである。ところが、この両方のモジュールを同時に使おうとすると、クラス B のメソッド m の実装が存在しないために、リンク時エラーになる。このように、正しく動く複数のモジュールを組み合わせたとき、実装されていない abstract method が発生する現象を**実装欠損**と呼んでいる。

4.2 補完モジュール

一般に実装欠損を自動的に補完することは不可能である。誰かが仕様を理解して、実装欠損を補完するモジュールを実装しなければならない。このようなモジュールを**補完モジュール**と呼ぶ。

エンドユーザにとっての使い勝手を良くするために、リンカーは、補完モジュールの自動リンクの機能を持っている。例えば誰かが m_sub と m_abst の間を補完する図 23のような補完モジュールを実装して、リンカーから見える場所に置いたとする。補完モジュールは、 “complements m_sub, m_abst” という “complements” 宣言を持つモジュールとして定義する。ここで、 “complements” 宣言は、コンパイラにとっては、“extends” 宣言と同じ意味である。ただし、

```
module m1 {
  define class A {
    define A(){}
  }
}
module m2 extends m1 {
  class A {
    define int m(){ return 1; }
  }
}
module m3 extends m1 {
  class A {
    define int m(){ return 2; }
  }
}
module m4 extends m2, m3 {
  class A {
    int FQN[m2::m](){
       return original() + 3; }
    int FQN[m3::m](){
       return original() + 4; }
  }
  class SS {
    void main(String[] args){
      A a = new A();
      //System.out.println(a.m()); // error
      System.out.println(
        a.FQN[m2::m]()); // 4
      System.out.println(
        a.FQN[m3::m]()); // 6
    }
  }
}
```

図 21: FQN の使用例

```
module m_orig {
  // S and a subclass A.
  define abstract class S { }
  define class A extends S { }
}
module m_sub extends m_orig {
  // Add a new subclass of S.
  define class B extends S { }
}
module m_abst extends m_orig {
  // Add a new method of S.
  class S {
    define abstract int m();
  }
  class A {
    int m(){ return 1; }
  }
}
```

図 22: 2つのモジュール間の実装欠損の例

```
module m_compl_sub_abst
  complements m_sub, m_abst {
  class B {
    int m(){ return 2; }
  }
}
```

図 23: 補完モジュール

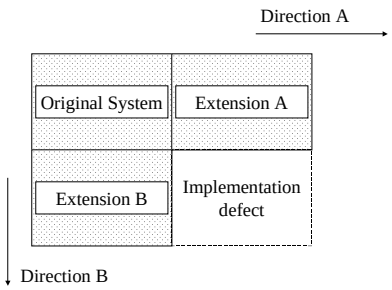

図 24: 2 方向の拡張と実装欠損

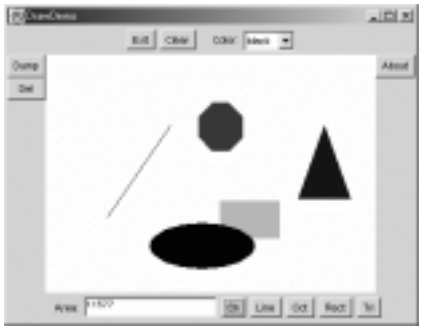

図 25: ドローツール

このモジュールが補完の対象とするモジュール名の情報が、コンパイル結果のバイナリに付加される。

そして、エンドユーザが次のように m_sub と m_abst を組み合わせようとすると、リンカーは補完モジュール m_compl_sub_abst を見つけ、それもリンクの対象とする。

```
% mj -s m_sub m_abst
```

つまりエンドユーザから見れば、補完は自動的に行われるため、実装欠損の問題を意識しなくてもすむ。これにより、エンドユーザが、実装の詳細を知らなくても、モジュールを組み合わせて自分の好みのアプリケーションを構築することを容易にしている。

4.3 他のシステムにおける実装欠損と補完モジュール

実装欠損の問題は、差分ベースモジュールに限らず、拡張性の高いシステムで一般に発生し得る。具体的には、2 方向以上の異なる次元に拡張可能なシステムにおいて発生する（図 24)。

1 つの例は monad transformer を使って書かれたプログラミング言語のインタープリタ [21] に見られる。このインタープリタは、 monad transformer と呼ぶ結合可能な「拡張モジュール」によって拡張可能なインタープリタである。monad は 1 つの抽象データ型だが、 monad transformer という技法を用いることにより、内部のデータ構造と、それに対する operator の両方が拡張可能になる。そのため、2 つの monad transformer を組み合わせるときに、「operator の lift」の定義が必要になる場合がある。これが、我々の呼ぶ補完モジュールに相当する。

もう 1 つのより身近な例は、パーソナルコンピュータ (PC) である。PC のユーザは、利用する OS と周辺機器を選択することができる。しかし、 PC が正しく動作するためには、選択した OS 用の、選択した周辺機器に対応するデバイスドライバを入手してインストールする必要がある。このデバイスドライバが、補完モジュールに相当する。

2 つ目の例は、実装欠損が起き得る拡張可能システムが、必ずしも非実用的ではないことを示すよい例である。拡張モジュールの提供者は、発生し得る全ての実装欠損を、あらかじめ補完しておく必要はない。需要が大きいと思われるモジュール間の補完モジュールだけを実装することで、大半のエンドユーザのカスタマイズの需要を満たすことができる。

5 プログラム例

5.1 ドローツール

典型的なオブジェクト指向アプリケーションの 1 つとして、ドローツールを MixJuice で記述した例について述べる。

図 25は、マウスを使って丸や四角を使った絵を書くツールである[8]。このプログラムは、 abstract class Figure があり、そのサブクラスとして個々の図形のクラスが定義されるというクラス階層を持っている（図 26)。

[8] このサンプルプログラムのソースコードと applet によるデモは、 MixJuice の web ページ [15] において公開されている。

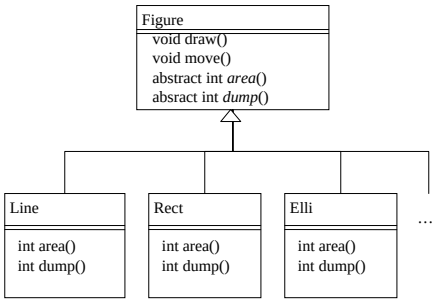

図 26: ドローツールのクラス図

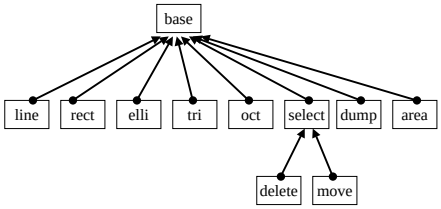

図 27: ドローツールのモジュール継承図

このプログラムは、モジュールを追加することにより、描くことができる図形の種類を追加したり、図形に対する操作を追加することができる。

図形を追加するモジュールは次のように実装できる。まず、クラス Figure の新たなサブクラスを定義する。そして、「ユーザが描画する図形を選択するボタン」を表示するメソッドの定義を拡張して、新たなボタン表示するコードを追加する。ボタンを表示するメソッドは、それ自体が “hook” になっているため、このようにオリジナルのソースコードを修正しなくても新たなボタンが追加できる。

個々の図形に対する操作（位置情報の出力、面積の計算など）も追加可能である。操作を追加するモジュールは、クラス Figure に abstract method を追加し、その全てのサブクラスに対してそれを実装するメソッドを追加する。また、「操作を選択するボタン」の表示メソッドの定義を拡張し、新たなボタンを表示するコードを追加する。

このようなスタイルで、現在下記のモジュールが実装されている。モジュール間の継承関係は図 27のようになっている。

`base` : フレームワーク
`select` : 図形の選択
`delete` : 選択した図形の削除
`move` : 選択した図形の移動
`dump` : 画面情報をテキスト形式で出力
`area` : 全図形の面積の合計を表示
`line` : 直線
`rect` : 長方形
`elli` : 楕円
`tri` : 三角形
`oct` : 八角形

モジュール base は、クラス Figure と、キャンバスの表示、メニューなどを定義する、このドローツールのフレームワークである。図形と操作を選択するためのボタンは１つもなく、これだけではドローツールとしては使えないが、アプリケーションとして動作はする。他の全てのモジュールは、このモジュール base の sub-module として定義される。

図形を追加するモジュールと、操作を追加するモジュールの間には、補完モジュールが必要である。補完モジュールは、 line, rect, elli, tri, oct と dump, area の間の１０個定義されている（図 28）。

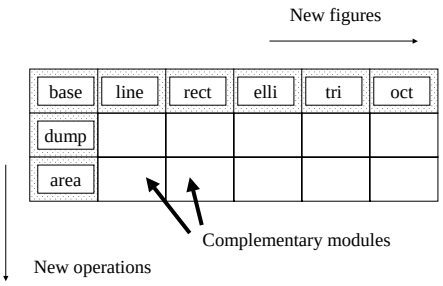

図 28: 図形を追加するモジュールと操作を追加するモジュールの間の補完モジュール

モジュール select, delete, move は図形に対する操作を定義するモジュールだが、図形を追加するモジュールとの間の補完モジュールは必要ない。これらの操作は、すべての図形のスーパークラスであるクラス Figure に対するメソッド追加で実現でき、個々のサブクラスに固有のメソッド定義は必要ないためである。

base と select 以外の 9 個のモジュールは任意に選択可能であるため、2^9 通りの以上の異なるアプリケーションが構築できる。図 25は、現在実装されている全てのモジュールを選択した場合のスクリーンショットである。

5.2 HTTP server

Java で書かれた現実の大規模アプリケーションが MixJuice に移植可能であることを示す例として、http server について述べる。

Java で書かれたオープンソースの 1 万行の HTTP server [9] を MixJuice で書き直した [32]。書き直しは、1 人のプログラマーが約 10 日間で完了した。

サーバーの中心となる部分を、クラスという単位とは独立にモジュール分割することにより、オリジナルのプログラムよりも高い拡張性を持たせることができた。例えば XML で記述された設定ファイルを読み込む部分を 2.4節で述べた方法で拡張可能なスタイルに変更することにより、新たな設定項目を容易に追加できる構造にできた。

[9]オリジナルの HTTP server (Jasper) は下記の URL において公開されている。`http://www.openje.org/jasper/`

また、オリジナルのプログラムが持つ Servlet に関連するコラボレーションを取り除き、より単純な機能のモジュールと交換することで、アプリケーションの最小構成サイズを 280KB から 130KB に減らすことができた。

5.3 HashMap

差分ベースモジュールが、高機能なクラスの内部をモジュール分割することができる例として、JDK1.2 に付属するクラスである java.util.HashMap について述べる。

HashMap のソースコードは、モジュラリティの高い実装にはなっていない。図 29は、ソースファイル HashMap.java に含まれる 3 つのクラス（そのうち 2 つは nested class ）の間の依存関係を示したものである。3 つのクラスは相互に依存している上、お互いのフィールドやメソッドが直接参照できる関係にあり、コードの一部の修正が、ファイル内のどの部分に影響を与えるかが非常にわかりにくくなっている。

HashMap のソースコードは、従来のクラスベースモジュール機構が、高機能なクラスの内部をモジュール化する能力を有していないことを示している。HashMap は、 nested class を巧みに用いることにより、外部には必要最低限の名前しか公開しない実装になっている。しかしその半面、3 つのクラスの field は、コメントを除く約 500 行のソースコードのすべてからアクセス可能になっている。クラスがさらに高機能化すれば、内部実装のスコープはさらに大きくなる。例えば JDK1.2 に付属する別のクラス TreeMap は、コメントを除き約 1000 行もあり、内部の依存関係の理解はいっそう難しくなっている。

MixJuice では、 HashMap の内部を モジュラリティの高い形に書き換えることが可能である。図 30は、HashMap のソースコードを MixJuice で書き直し、7 つのモジュールに分割した場合の、モジュール間の依存関係を示したものである。このように、依存関係にサイクルがなく、しかもサイズが小さく安定したモジュールのみにメソッド実装が依存する形に、モジュール分割することができる。これにより、プログラムの保守性は大きく向上すると考えられる。

HashMap の各メソッドは、 HashIterator に依存するものと依存しないものに分けて 2 つのモジュールで実装されている。このように HashIterator に関連

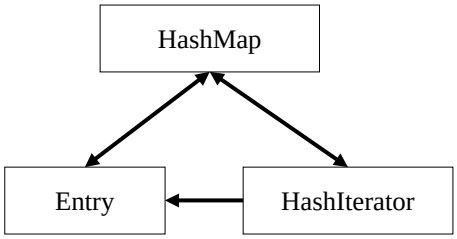

図 29: HashMap.java 内部のクラス間の依存関係

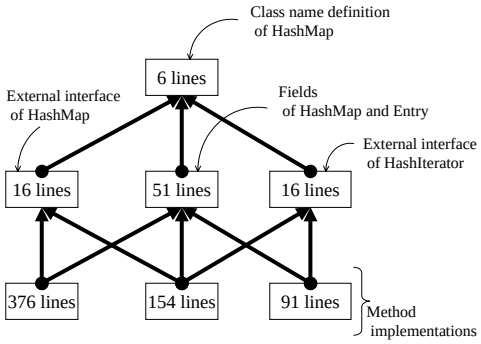

図 30: MixJuice を用いて書き直した HashMap.java のモジュール継承図

するコラボレーションと関連しないものにモジュール分割することにより、HashIterator の仕様や実装を他のモジュールに影響を与えないで変更することが可能になる。

6 実装

MixJuice で書かれたプログラムのソースコードは、プリプロセッサ（ソースコード変換）、 Java コンパイラ、ポストプロセッサ（バイトコード変換）によって処理され、最終的に Java 言語のバイトコードに変換される。

クラスの差分の追加を実現するには、JavaVM が本来持っている継承メカニズムを利用する（図 31）。モジュール本体に含まれるクラスの差分は、プリプロセッサによって、ダミーのスーパークラスを持つクラスに変換される。変換結果のソースコードは、 Java コンパイラによって普通にコンパイルされる。リンク時には、各差分が一列のクラス階層になるように、各差分のスーパークラスがポストプロセッサによって書き換えられ、適宜クラス名も変更されて、「差分を追加した結果」に相当するクラス階層が構築される。

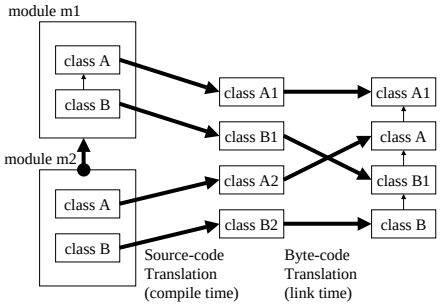

図 31: プリプロセッサとポストプロセッサによる実装

なお、プリプロセッサとポストプロセッサは、Java コンパイラによる型チェックと、JavaVM による byte-code verifier のチェックを通るように、注意深く変換を行っている。

本論文で述べたすべての機能が実装されており[10] 、現在ソースコードとともに公開されている [15]。プリプロセッサの部分に関しては、 MixJuice 自身を使って書き直す作業をすでに始めている。

7 関連研究

Mixin Layers[28] は、特殊な言語を用いず、 C++ の template を駆使して差分を記述する手法である。プログラミングスタイルは、差分ベースモジュールとほぼ同じである。しかしこの手法は、情報隠蔽がない、分割コンパイルができない、デバッグが困難であるという問題がある。MixJuice は、モジュール単位で分割コンパイルでき、デバッグも特別な困難なく可能である。

CLOS[29],Smalltalk[11] などのオブジェクト指向言語では、既存のクラスに対してあとからメソッドを追加することは普通に行なわれる。しかし、 MixJuice

[10]本論文は、 MixJuice Version 1.0 に基づいている。

が行なっているような、既存のクラスの特定のメソッドを override し、しかもオリジナルのメソッドを呼び出すこと（メソッドの差分的拡張）は、普通は行なわれない。またこれらの言語は、型チェック機構を持っていない。

Aspect-oriented programming やその他の手法により、複数のクラスにまたがる “concern” を分離可能にするシステムとして、AspectJ[19], Hyper/J[26], Demeter/Java[22], DJ[25] などがある。AspectJ[19] はメソッドの call graph に着目した concern 、Hyper/J[26] はコンパイルずみのバイトコードに内在する concern 、Demeter/Java[22],DJ[25] は木構造に対する traverse に関する concern をそれぞれ抜きだし、separation of crosscutting concerns を可能にするシステムである。これらのシステムが扱う concern は、差分ベースモジュールが扱うコラボレーションとは直交しているため、これらのシステムと差分ベースモジュールとは相互に補完できる技術である。

BCA[18] は、バイトコード変換により既存のクラスライブラリの再利用性を高めるためのシステムである。delta file と呼ぶ差分を記述することで、ソースコードのない既存のクラスライブラリを変更することができる。しかし、 delta file の型チェック機能は実装されていない。また、専用の JavaVM を必要とする。

Cecil[4], Dubious[24], MultiJava[5] は、いずれもマルチメソッドをサポートするオブジェクト指向言語であるが、クラスとは直交したモジュール機構を有し、モジュール単位で型チェック・分割コンパイルが可能である。既存のクラスのソースコードを編集せずに、クラスにメソッドを追加する機能も有しており、この機能を Chambers らは open class と呼んでいる[24]。しかし、既存のメソッドを差分的に拡張することはできない。MultiJava[5] は、 Java 言語に open class とマルチメソッドの機能を追加した言語である。MultiJava は、バイトコード変換ではなく、Chain of Responsibility pattern を用いて open class を実現しているため、クラスに差分を多く追加するほどメソッド呼び出しは遅くなる。一方 MixJuice の実装方法では、差分をいくら追加しても、メソッド呼び出しが遅くなることはない。

オブジェクト指向言語に ML のような高機能なモジュール機構を入れる試みとして、Moby[8], JavaMod[2] などがある。これらの論文は、形式的定義や型システムなどの理論的側面に主眼が置かれている。これらの研究は、実際に 1 万行以上の大規模ソフトウエアで実用性を検証する段階にはなっていない。

[7] は、 Java に似た言語に mixin を導入する際の型安全などの理論的な側面について述べている。ここでは mixin の最も大きな問題点を名前衝突ととらえ、module を継承するときに rename を行なえるようにしてこの問題を解決している。これに対し、MixJuice では FQN の導入によって衝突の問題を解決している。

8 Future work

8.1 before/after/around

現在はメソッドの拡張の仕方には、既存のメソッドの override しかない。現在までのプログラミングの経験上、CLOS や AspectJ のような、 before/after/around の機能が必要であると考えている。

8.2 パラメタ付きモジュール

現在は、1つの「差分」を複数のクラス階層に適用することができない。モジュールを記述する際、変更するクラスの具体的なクラス名がソースコード中に埋め込まれているからである。したがって、[33],[28] にあるように、linked list をモジュールとして記述して、異なるクラスに追加することはできない。このような機能は C++ の template のような機能、すなわちパラメタ付きモジュールによって実現可能になると考えている。

8.3 Assertion

多重継承の衝突の原因として最もやっかいなものは意味的衝突である。これは、コンパイル時にあるモジュールが想定していたクラス不変条件やメソッドの事前条件・事後条件が、他のモジュールとのリンクにより変化した場合に生じると解釈できる。Eiffel 言語 [23] が持つ assertion の機構を差分ベースモジュールに適用することにより、意味的な衝突の実行時の検出に役立てられる思われる。また、 subclass contracts [27] という機構を発展させることにより、リンク時のエラー検出に利用できると考えている。

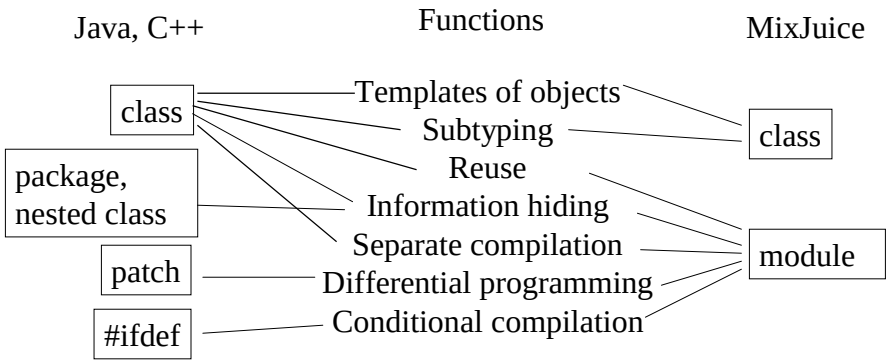

図 32: クラスとモジュールの役割

8.4 Dynamic loading

現在の実装は、普通の Java 言語同様、クラス単位の dynamic loading はできるが、モジュールの dynamic loading はできない。モジュールの dynamic loading が可能になれば、動的に挙動を変化させるアプリケーションが構築できると思われる。しかし、型的な安全性を保つためには動的にロードできるモジュールにかなり制約を入れる必要があると思われる。

9 まとめ

差分ベースモジュールと呼ぶ機構を提案し、それを Java 言語に適用したオブジェクト指向言語 MixJuice を設計・実装した。差分ベースモジュールは、クラスとモジュールの機構を完全に分離し、その代わりにモジュールと差分プログラミングの機構を統一したモジュール機構である。差分ベースモジュールは、アプリケーションの拡張性・再利用性・保守性の向上に対して多くの面で貢献する。また、個々のコラボレーションを別々のモジュールに分離することで、コラボレーション単位での開発・テストを可能にする。

差分ベースモジュールでは、クラスベースモジュールと比べ、クラスとモジュールの役割分担が図 32のように、整理されている。また、従来は言語コアの外部で行なわれていた条件コンパイルや差分の追加の機能は、モジュール機構がより安全な方法でサポートしている。

我々はすでに、トータルで２万行以上におよぶアプリケーション記述を行なっているが、現在のところ、差分ベースモジュールの本質的なデメリットは見付けていない。唯一のデメリットは、ソースコードが複数のモジュールに分散しているため、読みにくいと感じる場合がある点である。しかし、これはオブジェクト指向言語に対する批判、すなわちソースコードがスーパークラスとサブクラスに分散して読みにくくなる、と全く同じ問題である。ソースコードのモジュラリティと読みやすさは、場合によってはトレードオフの関係がある。この問題は、現在のオブジェクト指向プログラミングで行なわれているように、ソースコードブラウザのような外部ツールや、UML などの外部ドキュメントによって補うべき問題である。実際、各クラスやメソッドの役割がはっきりしていれば、クラス定義が複数のモジュールに分散していてもプログラムが読みにくいことはない。むしろ関連したメソッドが近い場所にあるため、かえって読みやすくなる。

差分ベースモジュールの原理は非常にシンプルであり、Java 言語以外のプログラミング言語にも容易に適用できる。また、言語仕様がシンプルであるため、パラメタ付きモジュールなど、言語機能を大きく発展させる可能性を有している。

参考文献

[1] Hirotake Abe, Yuuji Ichisugi, and Kazuhiko Kato. An implementation scheme of mobile threads with a source code translation technique in Java. In *IPSJ:PRO*, volume 41, pages 29–40. IPSJ, March 2000. in Japanese.

[2] Davide Ancona and Elena Zucca. True modules for Java classes. In *Proc. of the ECOOP2001*, 2001.

[3] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Proc. of the OOPSLA'96*, pages 69–82, 1996. Published as ACM SIGPLAN Notices, volume 31, number 10.

[4] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[5] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proc. of the OOPSLA2000*, pages 130–145, October 2000. Published as ACM SIGPLAN Notices, volume 35, number 10.

[6] R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proc. of the OOPSLA'94*, pages 164–175, October 1994. Published as ACM SIGPLAN Notices, volume 29, number 10.

[7] Dominic Duggan and Ching Ching Techaubol. Modular mixin-based inheritance for application frameworks. In *Proc. of the OOPSLA2001*, 2001.

[8] Kathleen Fisher and John Reppy. The design of a class mechanism for Moby. *ACM SIGPLAN Notices*, 34(5):37–49, May 1999.

[9] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Welsley, 1995.

[11] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation*. Addison-Wesley, 1983.

[12] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. Java series. Addison-Wesley, second edition, 2000.

[13] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object oriented systems. In *Proc. of the ECOOP/OOPSLA'90, Ottawa*, pages 169–180, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[14] Yuuji Ichisugi. EPP home page. `http://staff.aist.go.jp/y-ichisugi/epp/`.

[15] Yuuji Ichisugi. The programming language MixJuice. `http://staff.aist.go.jp/y-ichisugi/mj/`.

[16] Yuuji Ichisugi. MixJuice : An object-oriented language with simple and powerful module mechanism. Extended abstract of OOPSLA2000 poster session, October 2000.

[17] Yuuji Ichisugi and Yves Roudier. The extensible Java preprocessor kit and a tiny data-parallel Java. In *ISCOPE'97, California*, LNCS 1343, pages 153–160, December 1997.

[18] R. Keller and U. Höelzle. Binary component adaptation. In *Proc. of the ECOOP'98*, LNCS 1445, pages 307–329, 1998.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. of the ECOOP2001*, 2001.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the ECOOP'97*, LNCS 1241, pages 220–242, 1997. Invited Talk.

[21] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proc. of the POPL'95*, pages 333–343, January 1995.

[22] Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java (research demonstration). In *International Conference on Software Engineering*, pages 604–605, 1997.

[23] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Ed.* Prentice-Hall, Inc., 1997.

[24] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proc. of the ECOOP'99*, LNCS 1628, pages 279–303, 1999.

[25] Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , LNCS 2192, pages 73–80, Kyoto, Japan, September 2001.

[26] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proc. of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer*, 2000.

[27] Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Proc. of the OOPSLA 2000, Minneapolis, Minnesota*, volume 35(10), pages 208–228, October 2000. Published as ACM SIGPLAN Notices, volume 35, number 10.

[28] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *Proc. of the ECOOP'98*, LNCS 1445, pages 550–570, 1998.

[29] G.L. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.

[30] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, third edition, 1997.

[31] C.A. Szyperski. Import is not inheritance – why we need both: Modules and classes. In *Proc. of the ECOOP'92*, LNCS 615, 1992.

[32] Akira Tanaka and Yuuji Ichisugi. Modularization of HTTP server with programming language MixJuice. In *Proc. of JSSST 18th Annual Conference*, September 2001. in Japanese.

[33] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. In *Proc. of the OOPSLA'96*, October 1996. Published as ACM SIGPLAN Notices, volume 31, number 10.

A 文法

ここでは、 Java 言語の文法に対する差分のみを載せる。

- “*” は 0 回以上の繰り返しを表す。
- “<e>” は空文字列を表す。
- “<original alternatives>” は、Java 言語で定義されている選択肢が他にあることを表す。
- 断りのない限り、大文字で始まるワードは非終端記号、小文字で始まるワードと記号は終端記号を表す。
- 下記の非終端記号については、 Java 言語と同様である。

  Extends, Implements, Throws, FormalParameterList, VariableDecorators, ArgumentList, Block, Identifier, Name, Expression

```
MJCompilationUnit:
     MJModule*

MJModule:
     module MJModuleName MJModuleHeaderDeclaration* {
         MJTypeDeclaration*}

MJModuleHeaderDeclaration:
     extends MJModuleNameList
     complements MJModuleNameList

MJModuleNameList:
     MJModuleName
     MJModuleName , MJModuleNameList

MJTypeDeclaration:
     MJDefine MJModifier MJClassKeyword MJName
         Extends Implements { MJClassBodyDeclaration* }

MJClassKeyword:
     class
     interface

MJClassBodyDeclaration:
     MJFieldDeclaration
     MJConstructorDeclaration
     MJMethodDeclaration

MJFieldDeclaration:
     MJDefine MJModifier Type VariableDecorators

MJConstructorDeclaration:
     MJDefine MJModifier MJName ( FormalParameterList )
         Throws MJMethodBody

MJMethodDeclaration:
     MJDefine MJModifier Type MJName ( FormalParameterList )
         Throws MJMethodBody

MJDefine:
     <e>
     define

MJModifier:
     <e>
     abstract
     static

MJMethodBody:
     ;
     Block

MJName:
     Identifier
     MJFQN

MJFQN:
     FQN [ MJModuleName :: Identifier ]
     // This "FQN" is not a non-terimnal symbol, but a token.

MJModuleName:
     Name

Primary:
     <original alternatives>
     original ArgumentList
     MJFQN                              // field access
     MJFQN ArgumentList                 // method invocation
     Expression . MJFQN                 // field access
     Expression . MJFQN ArgumentList    // method invocation
     Name . MJFQN                       // static field access
     Name . MJFQN ArgumentList          // static method invocation
     MJFQN . MJFQN                      // static field access
     MJFQN . MJFQN ArgumentList         // static method invocation
     new MJFQN ...                      // instance creation

Type:
     <original alternatives>
     MJFQN
```