修 士 論 文

抽象解釈に基づく発展的プロトタイピングのための開発環境に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

番 真悟

2003 年 3 月

修 士 論 文

抽象解釈に基づく発展的プロトタイピングのための開発環境に関する研究

指導教官　片山卓也 教授

審査委員主査　片山卓也 教授
審査委員　落水浩一郎 教授
審査委員　権藤克彦 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

110106 番 真悟

提出年月: 2003 年 2 月

概 要

本論文では, 抽象解釈に基づく発展的プロトタイピングを行うための開発環境の設計, 実装を行う. また構築した環境の上で開発実験を行い, 発展的プロトタイピング技法の有効性を検証する.

目 次

図 目 次

第1章　はじめに

1.1　研究の背景

近年, 情報技術の発達により, インターネット上で稼働する大規模なネットワークソフトウェアや, 携帯電話といった複雑な制御を必要とするソフトウェアが要求されている. 大規模で複雑なソフトウェアを構築する際に, 初期段階で複雑な仕様の全貌を正しく理解し設計を行うことは困難である. また, 設計途中でユーザーの要求が変更されることもありえる. この問題に対応するためにシステムを段階的に構築し, その構築の途中段階で実行可能なプロトタイプを作成し, 開発者が仕様の正しさを確認したり, ユーザーにそのシステムが要求を満たしているか評価してもらうことは重要である.

一般的にシステムを段階的に構築する手法として, トップダウンに構築する手法とボトムアップに構築する手法がある. トップダウンにシステムを構築する手法では, システム全体を考慮しながら設計を進めることができるが, 開発の途中段階で実際にシステムを動かすためには, まだ実装されていない下位モジュールのダミーコードとなるスタブを開発者が用意する必要がある [1]. スタブによる実行では, その結果はダミーコードの影響を受けるため, 構築済のプログラムの正しさを確認することが難しいという問題がある. 一方, ボトムアップにシステムを構築する手法では, スタブを用意することなく実行しテストすることが可能であるが, 開発の途中段階でシステム全体の実行を確認できないという問題がある.

尾崎は, これらの問題を解決するために, 発展的プロトタイピング技法 [2] を提案している. この技法では, まず実際に構築するシステムで扱う値を抽象化し, その抽象値を用いてシステム全体を構築する. そして, その抽象値を徐々に具体化し, それにあわせてプログラムを発展させていくことでシステムを段階的に構築する. 各段階のプログラムの発展関係は形式的に定義されているので, システム全体のプログラム間で発展段階が揃っていなくても, その発展関係を用いて発展段階を揃えることで. システム全体の実行を可能としている. このため, 設計/実装/テストのサイクルをより短縮することが期待されている. しかし, この技法を実際の開発に適用するための開発環境や実行環境についてはまだ十分に議論されていない. この手法を用いて実際に開発を行ない, その有効性を検証するためには, それらの環境を構築する必要がある.

関連研究として ISDR 法 [3] がある. これは関数型プログラムの入出力データを段階的に具体化することで, プログラムを段階的に詳細化して開発していく手法である. この研究では副作用のない関数を対象としているが, 本技法では, 副作用のあるオブジェクト指

向言語である Java を対象としている.

1.2 研究の目的

本研究の目的は, 発展的プロトタイピング技法のための開発環境を構築し, その上で実際に開発を行うことで, この技法の有効性を検証することである.

開発環境は以下の 3 つからなる

- 抽象実行処理系
- 発展関係エディタ
- 抽象実行ビジュアライザ

この技法では, 発展段階が異なるオブジェクト間でのメソッド呼出しを, 実行時にオブジェクト間の発展段階を揃えることで, 可能としている. 発展段階を揃える際に, オブジェクトが扱う値を抽象化するため, その実行は抽象値上の実行となる. この技法では, これを**抽象実行**と呼ぶ. この技法では, 抽象実行を実現するための機構の実現については, まだ十分議論されていない. そこで, 本研究では, まず, Java において抽象実行を行うための機構を明らかにし, 抽象実行を行うための処理系の設計と実装を行う.

また, 一般的にプロトタイピングは短期間で行なわれるため, 正確なドキュメントなどは作成しない. しかし, 本技法では, プログラムを発展的に構築するため通常の開発より, 多くのプログラムを作成する. それらの発展関係は十分なドキュメントがなければ, その関係を正しく理解することは困難である.

そこで, 本研究では発展的に構成されたプログラムのドキュメントとなりうる開発ツールとして, 発展関係エディタと抽象実行ビジュアライザの構築を行う. 前者は, プログラムが扱う抽象値の具体化を様子とその抽象値に合わせてプログラムがどのように発展しているかを静的に表現するものである. 後者は, 単純なプロトタイプ実行の様子と, 抽象実行の複雑な振る舞いを視覚化する.

そして最後に, 上記の開発環境の上で実際に開発を行い, 本技法と開発環境を用いてどのようなアプリケーションを構築できるかを示す, また, 本技法の特色である抽象実行の有効性の明らかにする.

1.3 論文の構成

本論文の構成は以下の通りである.

第 2 章 発展的プロトタイピングの具体例として長方形オブジェクトの段階的構成を示し, その技法と形式的定義について述べる.

第 3 章 抽象実行について述べ, 処理系を実現するための要件や問題点について考察を行う. そして, その問題点等の解決策としてプロキシオブジェクトを用いた抽象実行処理系の提案を行い, その具体的な仕組みについて述べる. またプロキシオブジェクトの実現には, Java のリフレクション機能と XML 技術を用いている.

第 4 章 発展的プロトタイピング技法のための開発ツールである, 発展関係エディタ, 抽象実行ビジュアライザの設計と実装について述べる.

第 5 章 発展的プロトタイピング技法と構築した開発環境の上で実際に開発実験を行う. 例題としてトランプゲームのブラックジャックをあげる. この例題では, 抽象領域をうまく定めることで, 抽象実行が有効であることを示す. もう一つの例題として在庫管理システムをあげる. この例題では, 本技法の問題点として, 抽象領域の決定の難しさを示す.

第 6 章 開発実験の結果から, スタブを用いた開発途中段階でのテスト実行と抽象実行の違いを考察し, その有効性について述べる. また, 抽象実行を用いたテストを行うことで, プログラムの実装の誤りを発見できることを示す. さらに, 在庫管理システムの構築で示された問題点について議論を行う.

第 7 章 本研究のまとめを行う. 結論としてうまく抽象領域を定め, 発展的にプログラムを構成することで, 発展的プロトタイピング技法が有用であることを示す. また, 最後に本研究の将来への課題を示す.

第2章　発展的プロトタイピング技法

本章では, まず発展的プロトタイピング技法の概要について述べ, 具体例として長方形オブジェクトの発展的構成を紹介する. 発展的プロトタイピング技法は形式的に定義されたオブジェクト間の発展関係に基づいているので, その定義と例題の形式的な発展関係を示す.

2.1　概要

発展的プロトタイピング技法では, ソフトウェアの開発を段階的に行い, 設計/実装/テスト のサイクルを短くすることで仕様の正しさを確認したり, 仕様の誤りを早期に発見する事を目的としている. しかし, ある 1 つのプログラムの実装が完了しても, 協調して動作するオブジェクト群のすべての段階がそろわなければ実行することはできない. このため, いくらソフトウェアを段階的に構築したとしても, ある 1 つのオブジェクトの実装が遅れると実行/テストを行うことができず, それだけ開発のサイクルが長くなることになる.

この問題に対して, 発展的プロトタイピング技法では, まず最も原始的なオブジェクト群を生成し, そのオブジェクト群で抽象的ながらもプログラム全体を動かせるようにする. そしてそのオブジェクトを形式的な規則に従って互換性を保ったまま発展させる. このため, 発展段階が異なるオブジェクトがあったとしても, 発展の規則に従って各オブジェクトの発展段階を揃えることによって, システム全体として実行/テストすることが可能となる.

発展的プロトタイピング技法では, プロトタイプの開発手順は 2 つの段階に分けられる. 第 1 段階は, 最も抽象的な原始プログラムを構築する抽象化段階である. 第 2 段階は, 原始プログラムの機能を段階的に詳細化, 分解する発展段階である. 以下にその具体的な手順を示す.

抽象化段階　以下の手順で原始プログラムを作成する

1. メソッドの入出力やオブジェクトがもつ変数のデータの集合をそれぞれ抽象値に抽象化する.
2. その抽象値を使って最も原始的なプログラムを作成する.

発展段階　原始プログラムを, 仕様を完全に満すまで, 以下の手順にしたがって繰り返し発展させる.

1. オブジェクトが扱う抽象値を一段階具体化する.
2. 具体化した値に合わせて, インスタンス変数やメソッドを詳細化, 分割させることでプログラムを発展させる.
3. 発展したプログラムを抽象実行し, デバッグを行う.

2.2　構成例

以下は長方形オブジェクトの仕様例である. これ以後, この仕様を用いて本章では発展的プロトタイピング技法, 3 章では抽象実行処理系, 4 章では開発環境について説明する.

仕様例

長方形を表現するオブジェクト

- 長方形のサイズを変更する機能をもつ
- 長方形を表示する座標を移動する機能をもつ
- 長方形の移動は x 座標と y 座標の指定によって行われる

上記の仕様を長方形オブジェクトに対する入力値を図 2.1 のように段階的に具体化するに従って, メソッドの詳細化や分割を行ない, 長方形クラスを図 2.2 のように発展的に構成する.

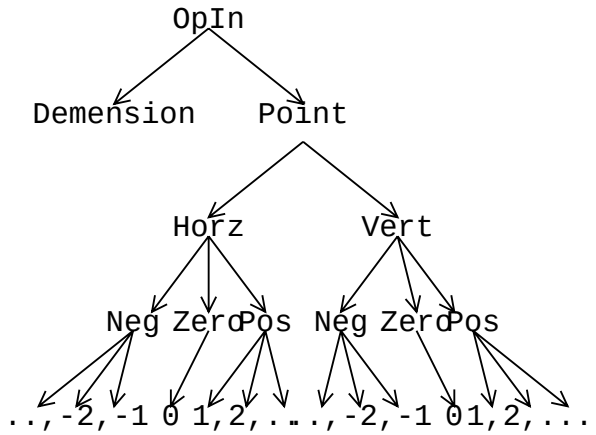

図 2.1: 長方形オブジェクトに対するの入力値の具体化

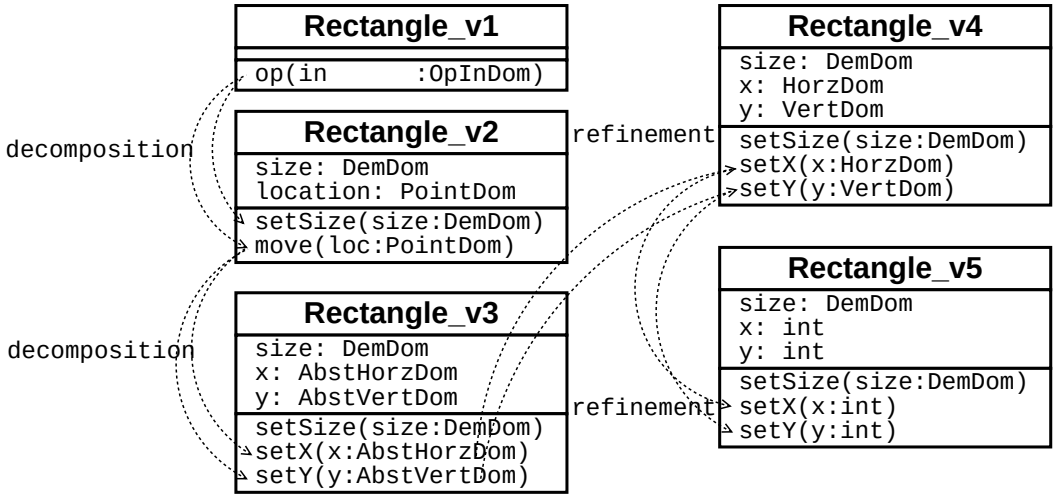

図 2.2: 長方形クラスの発展

抽象化段階

長方形オブジェクトの全ての仕様を最も抽象化した, 抽象的なクラス `Rectangle_v1` を以下に示す.

```
class Rectangle_v1 {
    void op(OpInDom v) {}
}
```

このクラスは, 長方形オブジェクトの機能を抽象化した唯一のメソッド `op()` を持つ. メソッド `op()` は, 長方形のすべての機能の入力値を抽象化した抽象値 $OpIn$ を入力とする.

発展段階 1

`Rectangle_v1` の `op()` をサイズの変更機能と移動機能に分解することで発展させた `Rectangle_v2` を以下に示す.

```
class Rectangle_v2 {
    DemDom size;
    PointDom location;
    void setSize(DemDom size) {
        this.size = size;
    }
    void move(PointDom location) {
        this.location = location;
    }
}
```

Rectangle_v1 のメソッド op() の入力となる抽象値 $OpIn$ を長方形のサイズを表す抽象値 $Demension$ と座標を表す抽象値 $Point$ に具体化する．メソッド op() は入力値としてサイズの変更のための抽象値 $Demension$ か, または座標を変更するための抽象値 $Point$ の**どちらか**を受けとることになるので, メソッド op() をサイズの変更を行うメソッド setSize() と座標の移動を行うメソッド move() に分解する．

メソッドの分解に伴って, Rectangle_v2 には長方形の大きさを表すインスタンス変数として size, 場所を表すインスタンス変数として location を追加している．メソッド setSize() は抽象値 $Demension$ を受取りインスタンス変数 size にセットし, メソッド move() は抽象値 $Point$ を受取りインスタンス変数 location にセットする．

発展段階 2

Rectangle_v2 の move() を縦軸の移動と横軸の移動に分解することで発展させた Rectangle_v3 を以下に示す．

```
class Rectangle_v3 {
    DemDom size;
    AbstHorzDom x;
    AbstVertDom y;
    void setSize(DemDom size) {
        this.size = size;
    }
    void setX(AbstHorzDom x) {
        this.x = x;
    }
    void setY(AbstVertDom y) {
        this.y = y;
    }
}
```

Rectangle_v2 のメソッド move() の入力となる抽象値 $Point$ を X 座標を表す抽象値 $Horz$ と Y 座標を表す抽象値 $Vert$ に具体化すると, メソッド move() は入力値として横方向への移動のための抽象値 $Horz$ と縦方向への移動のための抽象値 $Vert$ を**連続して**受けとることになる．そこでメソッド move() を横方向の移動メソッド getX() と縦方向の移動メソッド getY() の実行列に分割する．

メソッドの分解に伴って, インスタンス変数 point を X 座標と Y 座標の値を格納できるようにインスタンス変数 x とインスタンス変数 y に分割する．メソッド getX() は抽象値 $Horz$ を受取りインスタンス変数 x にセットし, メソッド getY() は抽象値 $Vert$ を受取りインスタンス変数 y にセットする．

発展段階 3

Rectangle_v3 の移動機能を詳細化することで発展させた Rectangle_v4 を以下に示す．

```
class Rectangle_v4 {
    DemDom size;
```

```
    HorzDom x;
    VertDom y;
    void setSize(DemDom size) {
      this.size = size;
    }
    void setX(HorzDom x) {
      this.x = x;
    }
    void setY(VertDom y) {
      this.y = y;
    }
}
```

Rectangle_v3 のメソッド getX() の入力となる抽象値 $Horz$ とメソッド getY() の入力となる抽象値 $Vert$ を, それぞれ, 負の整数を表す 1 つの抽象値 Neg と 0 を表す抽象値 $Zero$ そして正の整数を表す抽象値 Pos に具体化する. Rectangle_v4 では, 値の具体化に伴って, フィールド x, y の型とメソッド setX(),setY() の入力の型を詳細化する.

発展段階 4

最後に Rectangle_v4 の移動座標を整数で指定できるように発展させた Rectangle_v5 を以下に示す.

```
class Rectangle_v5 {
    DemDom size;
    int x;
    int y;
    void setSize(DemDom size) {
        this.size = size;
    }
    void setX(int x) {
        this.x = x;
    }
    void setY(int y) {
        this.y = y;
    }
}
```

Rectangle_v4 のメソッド getX() の入力値と getY() の入力値を Java の組み込みの整数 int の値に具体化し, それに伴って, インスタンス変数 x,y の型とメソッド setX(),setY() の入力の型を詳細化する.

発展的プロトタイピング技法では上記のようにして, オブジェクトのメソッドが扱う値やオブジェクトが状態として持つ値の具体化に伴って, メソッドやインスタンス変数を詳細化, 分割させることでクラスを発展的に構築していく.

本研究で扱う抽象実行は, 発展段階が異なるオブジェクト間でメソッド呼出しを行なう必要があるときに, あるオブジェクトをオブジェクトの発展関係を満す別のオブジェクトに作り替え, 発展

段階を揃えることそのメソッド呼出しを実現している. このオブジェクトの発展関係は, 上記で紹介したようなクラスの発展とオブジェクトの状態の発展関係から定義されている. 例えば図 2.3 では, `Rectangle_v4` のインスタンスと `Rectangle_v5` のインスタンスがあるが, この 2 つのオブジェクトは発展関係にある. それは, クラス `Rectangle_v5` がクラス `Rectangle_v4` を発展させたものであることと, それらのインスタンス変数が持つ値の間に発展関係があることから導かれる.

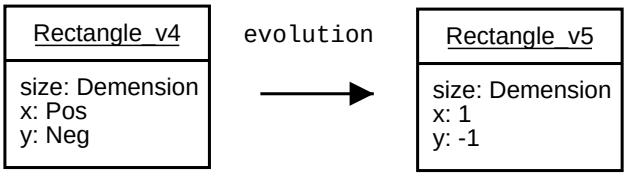

図 2.3: 発展関係にあるオブジェクト

本章の残りでは, オブジェクト間の発展関係を示すために必要となる形式的定義を順番に紹介し, 最終的に形式的な発展関係の定義において, 図 2.3 の 2 つのオブジェクトが発展関係にあることを示す.

2.3 形式的定義

2.3.1 CLASSICJAVA

発展的プロトタイピング技法では, オブジェクトの発展関係を CLASSICJAVA [4] を用いて定義している. CLASSICJAVA は形式的に定義された Java のサブセットであり, 主要な概念のみをコンパクトに形式化している.

オブジェクトの発展関係を形式化する上で重要なことは, いかにしてインスタンス変数の副作用を扱うかということである. CLASSICJAVA では副作用を扱うために**ストア**という概念を用いている. ストアとはあるシステムに存在するオブジェクトそれぞれのフィールド環境の集合であり, フィールド環境とはオブジェクトのインスタンス変数から値への写像である.

CLASSICJAVA の操作的意味論は式とストアの上での文脈書き換えシステムとして定義されており, プログラム P におけるメソッド呼出しは次のように定義されている.

$$
\begin{array}{rl}
P \vdash & \langle E[obj.md(v_1, ..., v_n)], \mathcal{S} \rangle \\
\hookrightarrow & \langle E[e[obj/this, v_1/var_1, ..., v_n/var_n]], \mathcal{S} \rangle \\
\text{where} & \mathcal{S}(obj) = (c, \mathcal{F}) \text{ and} \\
& (md, (t_1 ... t_n \to t), (var_1 ... var_n), e) \in_P^c c
\end{array}
$$

$\mathcal{S}(obj) = (c, \mathcal{F})$ とはオブジェクトからクラスとフィールド環境の組への写像であることを表している. また $\in_P^c$ という関係はあるクラスにあるメソッドが含まれていることを表す. 上記の例ではメソッド $md : t_1 ... t_n \to t$ がクラス c に含まれていることを表し, e は md の本体を表す式, $var_1 ... var_n$ は仮引数を表している.

2.3.2 データの発展

発展的プロトタイピング技法では, 最初にもっとも単純なプログラムを構成する. そのために, メソッドの入出力およびオブジェクトがもつ変数のデータ集合を抽象化する必要がある. 抽象化されたデータやその具体化のようすを表現するために, 以下にデータドメインを定義する.

定義 1 データドメイン
$Data$ をデータ集合, $\prec^{\mathcal{D}}$ を $Data$ 上のデータの具体化関係とする時, データドメイン DD を

$$DD \equiv (Data, \prec^{\mathcal{D}})$$

と定義する. ここで, $Data$ を $data(DD)$,$\prec^{\mathcal{D}}$ を $rel(DD)$ と書く.

例えば, `Rectangle_v1` のメソッド `op()` の入力ドメインを D_1^{in}, `Rectangle_v2` のメソッド `move()` の入力ドメインを D_2^{in}, `Rectangle_v3` のメソッド `setX()` の入力ドメインを D_3^{in}, `Rectangle_v4` のメソッド `setX()` の入力ドメインを D_4^{in}, `Rectangle_v5` のメソッド `setX()` の入力ドメインを D_5^{in} とすると

$$\begin{array}{rcl}
D_1^{in} & = & (\{OpIn\}, \emptyset) \\
D_2^{in} & = & (data(D_1^{in}) \cup \{Point\}, \\
 & & \{(OpIn, Point)\}) \\
D_3^{in} & = & (data(D_2^{in}) \cup \{Horz\}, \\
 & & \{rel(D_2^{in}) \cup (Point, Horz)\}) \\
D_4^{in} & = & (data(D_3^{in}) \cup \{Neg, Zero, Pos\}, \\
 & & \{rel(D_3^{in}) \cup (Horz, Neg), \\
 & & (Horz, Zero), (Horz, Pos)\}) \\
D_5^{in} & = & (data(D_4^{in}) \cup \{..., -1, 0, 1, ...\}, \\
 & & \{rel(D_4^{in}) \cup ..., (Neg, -1), \\
 & & (Zero, 0), (Pos, 1), ...\})
\end{array}$$

となる.

データドメインは, ドメイン中のデータをいくつかのより抽象度の低いデータへ具体化することで, 発展させることができる. ドメインの発展とは既存の抽象値が表している集合を互いに疎な集合に分割し, その分割された集合を表す新たな抽象値によって新たなドメインを構成することである. ドメインの発展を形式的に定義すると以下のようになる.

定義 2 データドメインの発展
DD, $DD^{\#}$ をデータドメインとし, DD が $DD^{\#}$ より発展しているとき $DD^{\#} \sqsubseteq^{\mathcal{D}} DD$ と記述し, 以下のように定義する.

$$\begin{array}{rl}
 & DD^{\#} \sqsubseteq^{\mathcal{D}} DD \\
\iff & data(DD^{\#}) \subseteq data(DD) \\
 & \quad \wedge rel(DD^{\#}) \subseteq rel(DD)
\end{array}$$

例えば, 上記のデータドメインの例で示した $D_1^{in}, D_2^{in}, D_3^{in}, D_4^{in}, D_5^{in}$ では,

$$D_1^{in} \sqsubseteq^{\mathcal{D}} D_2^{in} \sqsubseteq^{\mathcal{D}} D_3^{in} \sqsubseteq^{\mathcal{D}} D_4^{in} \sqsubseteq^{\mathcal{D}} D_5^{in}$$

が成り立つ.

データドメイン $D_1^{in}, D_2^{in}, D_3^{in}, D_4^{in}, D_5^{in}$ を扱うそれぞれのメソッドでは, 個々のデータドメインに含まれるすべてのデータ集合を扱うわけではない. 例えば D_2^{in} のデータ集合は $\{OpIn, Point\}$ であるが, `Rectangle_2` のメソッド `move()` では抽象値 $Point$ のみを扱う. なぜならば $OpIn$ に関する演算は `Rectangle_1` のメソッド `op()` ですでに定義されているからである. このように, 発展したメソッドでは, 発展関係にあるデータドメイン間の差分だけを扱う. その差分のデータ集合を具体集合とよび, 以下のように定義する.

定義 3 データドメインの具体集合

$$\begin{aligned} \uparrow DD \quad \equiv \quad & \{d \in data(DD) \mid \forall d' \in data(DD). \\ & d \neq d' \rightarrow (d, d') \notin rel(DD)\} \end{aligned}$$

例えば, $D_1^{in}, D_2^{in}, D_3^{in}, D_4^{in}, D_5^{in}$ のデータ集合は

$$\begin{aligned} \uparrow D_1^{in} &= \{OpIn\} \\ \uparrow D_2^{in} &= \{Point\} \\ \uparrow D_3^{in} &= \{Horz\} \\ \uparrow D_4^{in} &= \{Neg, Zero, Pos\} \\ \uparrow D_5^{in} &= \{..., -1, 0, 1, ...\} \end{aligned}$$

となる.

2.3.3 オブジェクトの状態の発展

メソッドの発展関係を定義するためには, 入出力の値の発展関係では不十分である. メソッドの実行には副作用があるので, メソッドの呼出しの前後のオブジェクトの状態間に発展関係がなければならない. そこで本小節では, オブジェクトの状態の発展関係を扱うために, フィールド環境の発展関係とストアの発展関係について示す.

フィールド環境の発展

フィールド環境の発展関係 $\sqsubseteq^{\mathcal{F}}$ はフィールド環境に存在する個々のインスタンス変数の発展関係とそのインスタンス変数がもつ値のデータの発展関係によって定義される. そこで, まずインスタンス変数の識別子の発展関係を定義する.

オブジェクトの状態が発展するとは, インタスタンス変数が扱えるデータドメインが発展することである. そこでインスタンス変数の集合からフィールド環境への写像である識別子 id を導入する.

識別子 id が $id^{\#}$ を発展させたものであるとき以下のように表す.

$$id^{\#} \prec^{id} id$$

例えば,`Rectangle_v4` のフィールド `x` と `Rectangle_v5` のフィールド `x` では

$$\texttt{Rectangle\_v4.x} \quad \prec^{id} \quad \texttt{Rectangle\_v5.x}$$

である.

フィールド環境はあるオブジェクトのインスタンス変数から値への写像である. したがって $\mathcal{F}^{\#}$ が $\mathcal{F}$ に発展しているとは, $\mathcal{F}$ が返す値が $\mathcal{F}^{\#}$ が返す値よりも発展しているということである. よってフィールド環境の発展関係を以下のように定義する.

定義 4 フィールド環境の発展関係
$\mathcal{F}$, $\mathcal{F}^{\#}$ をフィールド環境とし, $\mathcal{F}$ が $\mathcal{F}^{\#}$ より発展しているとき $\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$ と記述し, 以下のように表す.

$$\begin{aligned}\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} \quad \Longleftrightarrow \quad & \forall id^{\#} \in dom(\mathcal{F}^{\#}) \forall id \in dom(\mathcal{F}). \\ & id^{\#} \preceq^{id} id \Rightarrow \mathcal{F}^{\#}(id^{\#}) \preceq^{\mathcal{D}} \mathcal{F}(id)\end{aligned}$$

例えば, あるフィールド環境 $\mathcal{F}^{\#}$ において `Rectangle_v4` のインスタンス変数 `x` の値が Pos であり, フィールド環境 $\mathcal{F}$ において `Rectangle_v5` のインスタンス変数 `x` が 1 を持つとき,

$$\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$$

が成り立つ.

ストアの発展

フィールド環境の発展関係を用いて, ストアの発展関係 $\sqsubseteq^{\mathcal{S}}$ を定義する. ストアの発展関係は, ストア内に存在する個々のオブジェクトのフィールド環境間が発展関係を満さなければならない.

そこで, まずオブジェクトの状態の詳細化関係 $\sqsubseteq_{\mathcal{S}}^{\mathcal{S}^{\#}}$ を以下のように定義する. なお, この定義ではクラスの詳細化関係 $\sqsubseteq^{\mathcal{C}}$ を用いているが, この関係の詳細については後で説明する.

定義 5 オブジェクトの状態の発展関係
$\mathcal{S}^{\#}$, $\mathcal{S}$ をストア, $obj^{\#}$, obj をオブジェク $\mathcal{F}^{\#}$, $\mathcal{F}$ をフィールド環境, $c^{\#}$, c をクラスとする. オブジェクト obj が オブジェクト $obj^{\#}$ より発展しているとき $obj^{\#} \sqsubseteq_{\mathcal{S}}^{\mathcal{S}^{\#}} obj$ と表し, 以下のように定義する.

$$obj^{\#} \sqsubseteq_{\mathcal{S}}^{\mathcal{S}^{\#}} obj \iff \mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} \wedge c^{\#} \sqsubseteq^{\mathcal{C}} c$$

$$where\ \mathcal{S}(obj) = (c^{\#}, \mathcal{F})\ and\ \mathcal{S}^{\#}(obj^{\#}) = (c, \mathcal{F}^{\#})$$

そしてオブジェクトの状態の発展関係を用いることで, ストアの発展関係を次のように定義することができる.

定義 6 ストアの発展関係
$obj^{\#}$, obj をオブジェクトとする. ストア $\mathcal{S}$ がストア $\mathcal{S}^{\#}$ より発展しているとき, $\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}$ と記述し, 以下のように定義する.

$$\begin{aligned}&\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} \\ &\quad \Longleftrightarrow \quad (\forall obj \in dom(\mathcal{S}) \exists obj^{\#} \in dom(\mathcal{S}^{\#}). \\ &\qquad\qquad obj^{\#} \sqsubseteq_{\mathcal{S}}^{\mathcal{S}^{\#}} obj) \wedge \\ &\qquad\quad (\forall obj^{\#} \in dom(\mathcal{S}^{\#}) \exists obj \in dom(\mathcal{S}). \\ &\qquad\qquad obj^{\#} \sqsubseteq_{\mathcal{S}}^{\mathcal{S}^{\#}} obj)\end{aligned}$$

2.3.4 オブジェクトの振舞の発展

メソッドの詳細化

メソッドの詳細化関係 $\sqsubseteq^{\mathcal{M}}$ は, 抽象メソッドとその詳細化メソッドの間の関係である.$md^{\#}$, md をメソッドとするとメソッドの詳細化関係は次のように定義できる.

$$md^{\#} \sqsubseteq^{\mathcal{M}} md$$

md が $md^{\#}$ を詳細化しているとは以下の条件を満しているときである.

- メソッド md の入力値と md 実行前のシステムの状態がそれぞれ, メソッド $md^{\#}$ の入力値と $md^{\#}$ 実行前のシステムの状態より発展しているとき,$md^{\#}$ の出力値と $md^{\#}$ 実行後のシステムの状態もそれぞれ,md の出力値と md 実行後のシステムの状態より発展したものである.
- メソッド $md^{\#}$ の入力値と $md^{\#}$ 実行前のシステムの状態がそれぞれ, メソッド md の入力値と md 実行前のシステムの状態を抽象化したものであるとき,$md^{\#}$ の出力値と $md^{\#}$ 実行後のシステムの状態もそれぞれ,md の出力値と md 実行後のシステムの状態を抽象化したものである.

この上記の条件によって,md が $md^{\#}$ に詳細化するときに, それらの入力値, 出力値, 実行前ストア, 実行後ストアのそれぞれが過不足なく詳細化関係にあることが示される. したがって, メソッドの詳細化関係は以下ように定義できる.

定義 7 メソッドの詳細化関係
$DD^{\#}_{in}$, $DD^{\#}_{out}$, DD_{in}, DD_{out} をデータドメインとし,SS ストア集合とする. オブジェクト obj の持つメソッド $md : \uparrow DD_{in} \to \uparrow DD_{out}$ が, オブジェクト $obj^{\#}$ の持つメソッド $md : \uparrow DD_{in} \to \uparrow DD_{out}$ を詳細化しているとき,$md^{\#} \sqsubseteq^{\mathcal{M}} md$ と記述し, 図 *2.4* のように定義する.

例えば, `Rectangle_5` のメソッド `setX()` は `Rectangle_4` のメソッド `setX()` を詳細化したものであり,

$$\texttt{Rectangle\_v4.setX}() \sqsubseteq^{\mathcal{M}} \texttt{Rectangle\_v5.setX}()$$

が成り立つ.

メソッドの直和分割

メソッド md_1, md_2 のそれぞれが $md^{\#}$ を直和に分割するとき, メソッドの選択を $md_1|md_2$ と表現すると, メソッドの直和関係を以下のように定義できる.

$$md^{\#} \sqsubseteq^{\mathcal{M}}_{+} md_1|md_2$$

この関係は, それぞれの分解メソッドが, 同じメソッドに抽象化されるため, メソッドの詳細化関係に似ている. しかし, 分解メソッドとその抽象メソッドの間にはメソッドの詳細化関係の後者の条件を満たさないので, メソッドの詳細化とは異なる. 分解メソッドは, その全てのメソッドを合わせることによって詳細化メソッドと同じとなる. したがってメソッドの直和分割を次のように定義する.

$$
\begin{aligned}
& md^{\#} \sqsubseteq^{\mathcal{M}} md \\
\iff\quad & DD_{in}^{\#} \sqsubseteq^{\mathcal{D}} DD_{in} \wedge DD_{out}^{\#} \sqsubseteq^{\mathcal{D}} DD_{out} \wedge \\
& (\forall d_{in} \in \uparrow DD_{in} \forall d_{out} \in DD_{out} \\
& \quad \forall \mathcal{S}^{\#}, \mathcal{S}'^{\#}, \mathcal{S}, \mathcal{S}' \in SS \\
& \quad \exists d_{in}^{\#} \in \uparrow DD_{in}^{\#} \exists d_{out}^{\#} \in DD_{out}^{\#}. \\
& \quad \mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} \wedge d_{in}^{\#} \preceq^{\mathcal{D}} d_{in} \\
& \qquad \Rightarrow \mathcal{S}'^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}' \wedge d_{out}^{\#} \preceq^{\mathcal{D}} d_{out}) \wedge \\
& (\forall d_{in}^{\#} \in \uparrow DD_{in}^{\#} \forall d_{out}^{\#} \in DD_{out}^{\#} \\
& \quad \forall \mathcal{S}^{\#}, \mathcal{S}'^{\#}, \mathcal{S}, \mathcal{S}' \in SS \\
& \quad \exists d_{in} \in \uparrow DD_{in} \exists d_{out} \in DD_{out}. \\
& \quad \mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} \wedge d_{in}^{\#} \preceq^{\mathcal{D}} d_{in} \\
& \qquad \Rightarrow \mathcal{S}'^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}' \wedge d_{out}^{\#} \preceq^{\mathcal{D}} d_{out}) \\
\text{where}\quad & P \vdash \langle E[obj^{\#}.md^{\#}(d_{in}^{\#})], \mathcal{S}^{\#} \rangle \overset{*}{\hookrightarrow} \langle E[d_{out}^{\#}], \mathcal{S}'^{\#} \rangle \\
& P \vdash \langle E[obj.md(d_{in})], \mathcal{S} \rangle \overset{*}{\hookrightarrow} \langle E[d_{out}], \mathcal{S}' \rangle
\end{aligned}
$$

図 2.4: メソッドの詳細化定義

定義 8 メソッドの直和分割

$DD_{in}^{\#}$, $DD_{out}^{\#}$, DD_{in}, DD_{out} をデータドメインとし, SS をストア集合とする. それぞれのデータドメインは $\uparrow DD_{in} = D_1 \cup D_2$, $D_1 \cap D_2 = \emptyset$, $\uparrow DD_{out} = D_1' \cup D_2'$, $D_1' \cap D_2' = \emptyset$ である. obj のメソッド $md_1 : D_1 \to D_1'$ と $md_2 : D_2 \to D_2'$ が, オブジェクト $obj^{\#}$ のメソッド $md^{\#} : \uparrow DD_{in}^{\#} \to \uparrow DD_{out}^{\#}$ 直和分割しているとき $md^{\#} \sqsubseteq_{+}^{\mathcal{M}} md_1 | md_2$ と表現し, 図 *2.5* のように定義する.

例えば, `Rectangle_v2` のメソッド `setSize()` とメソッド `move()` は `Rectangle_v1` のメソッド `op()` を直和に分割したものであり,

$$\texttt{op}() \sqsubseteq_{+}^{\mathcal{M}} \texttt{setSize}() | \texttt{move}()$$

が成り立つ.

メソッドの直積分割

メソッドの直積分割関係では, 発展前のメソッドの入出力データ集合を直積になるように発展させる. 分割された個々のメソッドを逐次実行することでより詳細な機能を実現することが可能となる.

メソッドの直積分割関係 $\sqsubseteq_{\times}^{\mathcal{M}}$ は, メソッド md_1, md_2 が $md^{\#}$ を直積分割しているとは, $md_1; md_2$ を md_1, md_2 の逐次実行とすると, 以下のように表現できる.

$$md^{\#} \sqsubseteq_{\times}^{\mathcal{M}} md_1; md_2$$

この時,md_1 と md_2 の実行の間には, いかなるメソッドの呼出しも発生しないと仮定する. よって md_1 実行後のシステムの状態と md_2 実行前のシステムの状態は一致する. 従って, メソッドの直積分割関係は以下のように定義する.

$$
\begin{array}{ll}
md^{\#} \sqsubseteq_{+}^{\mathcal{M}} md_1|md_2 & \\
\iff & DD_{in}^{\#} \sqsubseteq^{\mathcal{D}} DD_{in} \wedge DD_{out}^{\#} \sqsubseteq^{\mathcal{D}} DD_{out} \wedge \\
& (\forall d_1 \in D_1 \forall d_1' \in D_1' \forall d_2 \in D_2 \forall d_2' \in D_2' \\
& \quad \forall \mathcal{S}^{\#}, \mathcal{S}'^{\#}, \mathcal{S}_1, \mathcal{S}'_1, \mathcal{S}_2, \mathcal{S}'_2 \in SS \\
& \quad \exists d_{in}^{\#} \in \uparrow DD_{in}^{\#} \exists d_{out}^{\#} \in \uparrow DD_{out}^{\#}. \\
& \quad d^{\#} \preceq^{\mathcal{D}} d_k \wedge \mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}_k \\
& \qquad \Rightarrow d_{out}^{\#} \preceq^{\mathcal{D}} d_k' \wedge \mathcal{S}'^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}'_k) \wedge \\
& (\forall d_{in}^{\#} \in \uparrow DD_{in}^{\#} \forall d_{out}^{\#} \in \uparrow DD_{out}^{\#} \\
& \quad \forall \mathcal{S}^{\#}, \mathcal{S}'^{\#}, \mathcal{S}_1, \mathcal{S}'_1, \mathcal{S}_2, \mathcal{S}'_2 \in SS \\
& \quad \exists d_1 \in D_1 \exists d_1' \in D_1' \exists d_2 \in D_2 \exists d_2' \in D_2'. \\
& \quad d^{\#} \preceq^{\mathcal{D}} d_k \wedge \mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}_k \\
& \qquad \Rightarrow d_{out}^{\#} \preceq^{\mathcal{D}} d_k' \wedge \mathcal{S}'^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}'_k) \\
\text{where} & 1 \leq k \leq 2 \\
& P \vdash \langle E[obj^{\#}.md^{\#}(d_{in}^{\#})], \mathcal{S}^{\#} \rangle \overset{*}{\hookrightarrow} \langle E[d_{out}^{\#}], \mathcal{S}'^{\#} \rangle \\
& P \vdash \langle E[obj.md_k(d_k)], \mathcal{S}_k \rangle \overset{*}{\hookrightarrow} \langle E[d_k'], \mathcal{S}_k \rangle
\end{array}
$$

図 2.5: メソッドの直和分割定義

定義 9 メソッドの直積分割関係

$DD_{in}^{\#}$, $DD_{out}^{\#}$, DD_{in}, DD_{out} をデータドメイン,SS をストア集合とする. オブジェクト obj が持つメソッド $md_1 : D_1 \to D_1'$ *and* $md_2 : D_2 \to D_2'$ が, $obj^{\#}$ の持つメソッド $md^{\#} : \uparrow DD_{in}^{\#} \to \uparrow DD_{out}^{\#}$ を直積分割しているとき,$md^{\#} \sqsubseteq_{\times}^{\mathcal{M}} md_1; md_2$ と記述し, 図 *2.6* のように定義する.

例えば, `Rectangle_v3` のメソッド `setX()` とメソッド `setY()` は `Rectangle_v2` のメソッド `move()` を直積に分割したものであり,

$$\texttt{move}() \sqsubseteq_{\times}^{\mathcal{M}} \texttt{setX}(); \texttt{setY}()$$

が成り立つ.

クラスの発展

メソッドの詳細化関係と分解関係を用いてクラスの発展関係を定義する. クラス c がクラス $c^{\#}$ に詳細化されているとき, 以下のように表現する.

$$c^{\#} \sqsubseteq^{\mathcal{C}} c$$

一般的にクラスはメソッド定義とフィールド定義から構成されるが, 本技法ではフィールドは他のオブジェクトからアクセスされないと仮定しているので, メソッドの集合によってのみクラスの発展関係を定義する. オブジェクトを抽象データ型と考えた時に, 他のオブジェクトからオブジェクトの構造に直接アクセスできることは好ましくない. したがって本技法もフィールドアクセスは考えない.

$$
\begin{array}{rl}
& md^{\#} \sqsubseteq_{\times}^{\mathcal{M}} md_1; md_2 \\
\Longleftrightarrow & DD_{in}^{\#} \sqsubseteq^{\mathcal{D}} DD_{in} \wedge DD_{out}^{\#} \sqsubseteq^{\mathcal{D}} DD_{out} \wedge \\
& (\forall d_1 \in D_1 \forall d_1' \in D_1' \forall d_2 \in D_2 \forall d_2' \in D_2' \\
& \quad \forall \mathcal{S}^{\#}, \mathcal{S}'^{\#}, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}'_2 \in SS \\
& \quad \exists d_{in}^{\#} \in \uparrow DD_{in}^{\#} \exists d_{out}^{\#} \in \uparrow DD_{out}^{\#}. \\
& \quad d_{in}^{\#} \preceq^{\mathcal{D}} (d_1, d_2) \wedge \mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}_1 \\
& \qquad \Rightarrow d_{out}^{\#} \preceq^{\mathcal{D}} (d_1', d_2') \wedge \mathcal{S}'^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}'_2) \wedge \\
& (\forall d_{in}^{\#} \in \uparrow DD_{in}^{\#} \forall d_{out}^{\#} \in \uparrow DD_{out}^{\#} \\
& \quad \forall \mathcal{S}^{\#}, \mathcal{S}'^{\#}, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}'_2 \in SS \\
& \quad \exists d_1 \in D_1 \exists d_1' \in D_1' \exists d_2 \in D_2 \exists d_2' \in D_2'. \\
& \quad d_{in}^{\#} \preceq^{\mathcal{D}} (d_1, d_2) \wedge \mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}_1 \\
& \qquad \Rightarrow d_{out}^{\#} \preceq^{\mathcal{D}} (d_1', d_2') \wedge \mathcal{S}'^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}'_2) \\
\text{where} & P \vdash \langle E[obj^{\#}.md^{\#}(d_{in}^{\#})], \mathcal{S}^{\#} \rangle \overset{*}{\hookrightarrow} \langle E[d_{out}^{\#}], \mathcal{S}'^{\#} \rangle \\
& P \vdash \langle E[obj.md_1(d_1)], \mathcal{S}_1 \rangle \overset{*}{\hookrightarrow} \langle E[d_1'], \mathcal{S}_2 \rangle \\
& P \vdash \langle E[obj.md_2(d_2)], \mathcal{S}_2 \rangle \overset{*}{\hookrightarrow} \langle E[d_2'], \mathcal{S}'_2 \rangle
\end{array}
$$

図 2.6: メソッドの直積分割定義

メソッド集合の発展関係は, 詳細化メソッド集合関係 $\sqsubseteq^{MS}$, 直積分割メソッド関係 $\sqsubseteq_{\times}^{MS}$, 直和分割メソッド関係 $\sqsubseteq_{+}^{MS}$ の 3 つからなる. そのそれぞれを以下のように定義する.

定義 10 メソッド集合の詳細化

$Meth^{\#}$, $Meth$ をメソッド集合とする. メソッド集合 $Meth$ がメソッド集合 $Meth^{\#}$ を詳細化するとき $Meth^{\#} \sqsubseteq^{MS} Meth$ と表し, 以下のように定義する.

$$
\begin{array}{rl}
& Meth^{\#} \sqsubseteq^{MS} Meth \\
\Longleftrightarrow & (\forall md^{\#} \in Meth^{\#} \exists md \in Meth. \\
& \quad md^{\#} \sqsubseteq^{\mathcal{M}} md) \wedge \\
& (\forall md \in Meth \exists md^{\#} \in Meth^{\#}. \\
& \quad md^{\#} \sqsubseteq^{\mathcal{M}} md)
\end{array}
$$

定義 11 メソッド集合の直積分割

$Meth^{\#}$, $Meth$ をメソッド集合とする. メソッド集合 $Meth$ がメソッド集合 $Meth^{\#}$ を直積分割するとき,$Meth^{\#} \sqsubseteq_{\times}^{MS} Meth$ と表し, 以下のように定義する.

$$
\begin{array}{rl}
& Meth^{\#} \sqsubseteq_{\times}^{MS} Meth \\
\Longleftrightarrow & (\forall md^{\#} \in Meth^{\#} \exists md_1, md_2 \in Meth. \\
& \quad md^{\#} \sqsubseteq_{\times}^{\mathcal{M}} md_1; md_2 \vee md^{\#} \in Meth) \wedge \\
& (Meth = \bigcup_{md^{\#} \in Meth^{\#}} (\{md_k \mid md_k \in Meth \wedge \\
& \quad md^{\#} \sqsubseteq_{\times}^{\mathcal{M}} md_1; md_2\} \cup \{md^{\#} \mid md^{\#} \in Meth\})) \\
\textit{where} & 1 \leq k \leq 2
\end{array}
$$

定義 12 メソッド集合の直和分割

$Meth^{\#}$, $Meth$ をメソッド集合とする. メソッド集合 $Meth$ がメソッド集合 $Meth^{\#}$ を直和分割するとき,$Meth^{\#} \sqsubseteq_{+}^{MS} Meth$ と表し, 以下のように定義する.

$$
\begin{array}{rl}
& Meth^{\#} \sqsubseteq_{+}^{MS} Meth \\
\Longleftrightarrow & (\forall md^{\#} \in Meth^{\#} \exists md_1, md_2 \in Meth. \\
& \quad md^{\#} \sqsubseteq_{+}^{\mathcal{M}} md_1 | md_2 \vee md^{\#} \in Meth) \wedge \\
& (Meth = \bigcup_{md^{\#} \in Meth^{\#}} (\{md_k \mid md_k \in Meth \wedge \\
& md^{\#} \sqsubseteq_{+}^{\#} md_1 | md_2\} \cup \{md^{\#} \mid md^{\#} \in Meth\})) \\
where & 1 \leq k \leq 2
\end{array}
$$

メソッド集合の詳細化, 直積分割, 直和分割関係を用いてクラスの発展関係を以下のように定義する.

定義 13 クラスの発展関係

$c^{\#}$, c をクラス,$Meth^{\#}$ を $c^{\#}$ のメソッド集合, $Meth$ を c のメソッド集合とする.c が $c^{\#}$ より発展しているとき $c^{\#} \sqsubseteq^{\mathcal{C}} c$ と記述し, 以下のように定義する.

$$
\begin{array}{rll}
c^{\#} \sqsubseteq^{\mathcal{C}} c & \Longleftrightarrow & Meth^{\#} \sqsubseteq^{MS} Meth \vee \\
& & Meth^{\#} \sqsubseteq_{\times}^{MS} Meth \vee \\
& & Meth^{\#} \sqsubseteq_{+}^{MS} Meth
\end{array}
$$

図 2.7 に示したクラス `Rectangle_v4` のメソッド集合 $Meth^{\#}$ とクラス `Rectangle_v5` のメソッド集合 $Meth$ は $Meth^{\#} \sqsubseteq^{MS} Meth$ の関係を満すので,

$$\texttt{Rectangle\_v4} \sqsubseteq^{\mathcal{C}} \texttt{Rectangle\_v5}$$

が成り立つ.

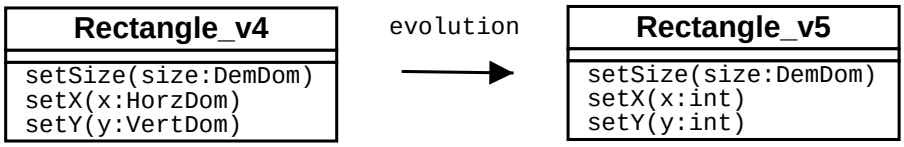

図 2.7: 発展関係にあるクラス

2.3.5 オブジェクトの発展関係

オブジェクト obj とフィールド環境 $\mathcal{F}$ がそれぞれオブジェクト $obj^{\#}$ とフィールド環境 $\mathcal{F}^{\#}$ より発展しているとき, 以下のように記述する.

$$obj^{\#} @ \mathcal{F}^{\#} \approx obj @ \mathcal{F}$$

この関係は実行時の, オブジェクトの抽象解釈に用いられる. オブジェクトの発展関係は, オブジェクトの機能と実行時のシステムの状態によって定義される. オブジェクトの機能は, そのオブジェクトのクラスによって特徴づけられ, システムの状態はそのオブジェクトのフィールド環境に依存する. よってオブジェクトの発展関係を以下のように定義する.

定義 14 オブジェクトの発展関係

$obj^{\#}$, obj をオブジェクト, $\mathcal{F}^{\#}$ と $\mathcal{F}$ をフィールド環境とする. また c を オブジェクト obj のクラス,$c^{\#}$ をオブジェクト $obj^{\#}$ のクラスとする. このときオブジェクトの発展関係は以下のように定義される.

$$obj^{\#}@\mathcal{F}^{\#} \approx obj@\mathcal{F}$$
$$\iff \quad c^{\#} \sqsubseteq^{\mathcal{C}} c \wedge \mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$$

図 2.3 のそれぞれのクラスは, 図 2.7 で示したとおり `Rectangle_v4` $\sqsubseteq^{\mathcal{C}}$ `Rectangle_v5` の関係にあり, それらのインスタンス変数においても以下の関係が成り立っている.

$$\begin{array}{rcll}
\texttt{Rectangle_v4.size} & \preceq^{id} & \texttt{Rectangle_v5.size} & \wedge \\
\text{Demension} & \prec^{\mathcal{D}} & \text{Demension} & \wedge \\
\texttt{Rectangle_v4.x} & \preceq^{id} & \texttt{Rectangle_v5.x} & \wedge \\
\text{Pos} & \prec^{\mathcal{D}} & 1 & \wedge \\
\texttt{Rectangle_v4.y} & \preceq^{id} & \texttt{Rectangle_v5.y} & \wedge \\
\text{Neg} & \prec^{\mathcal{D}} & \text{-1} &
\end{array}$$

従って, `Rectangle_v4` のインスタンスのフィールド環境を $\mathcal{F}^{\#}$, `Rectangle_v5` のインスタンスのフィールド環境を $\mathcal{F}$ とすると, $\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$ が成り立つ. よって, 図 2.3 の 2 つのオブジェクトは発展関係にある.

第3章　抽象実行処理系の設計と実装

本章では, まず具体例を用いて抽象実行について述べたあと, 処理系を実現するための要件や問題点について考察を行う. そして, その問題点等の解決策としてプロキシオブジェクトを用いた抽象実行処理系の提案を行う. プロキシオブジェクトは Java のリフレクション, Dynamic Proxy そして XML を用いて実現している. その具体的な実現技法について述べる.

3.1　抽象実行とは

本研究における抽象実行とは, 具体的な値を扱うオブジェクトがまだ実装されていない時に, 抽象化された値の上で実行を解釈することで, システムの実行を行うことである. 抽象実行が発生するのは, あるメソッド $meth$ の実行を呼び出す側のオブジェクト (caller) と $meth$ を実装するオブジェクト (callee) の発展段階が異なる場合である. 具体的には, 主に, 以下の 2 つのケースが発生する. メソッド $meth$ がメソッド $meth^{\#}$ より発展しているとすると,

1. callee が $meth^{\#}$ を実装しているときに, caller が callee に対して $meth$ を呼び出した場合と
2. callee が $meth$ を実装しているときに, caller が callee に対して $meth^{\#}$ を呼び出した場合

に発生する.

前章の長方形オブジェクトの例を用いて, 抽象実行の例を示す. `Rectangle_v4` の メソッド `setX()` は引数として {Neg, Zero, Pos} のいずれかを受け取るように実装されている.

上記の 1 に当てはまるのは `Rectangle_v4` に対して Java の組込み整数の 1 を入力として `setX()` が呼び出された場合である. この場合, $meth$ を実行する代りに $meth^{\#}$ を実行する. つまり入力値の整数の 1 を抽象値 Pos に抽象化して `setX()` を実行する. 本研究ではこれを**メソッドの縮退による抽象実行**と呼ぶ. 実行前後のオブジェクトの状態は図 3.1 のようになる.

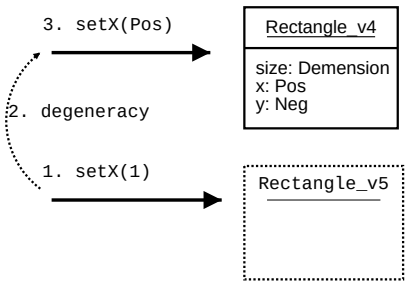

図 3.1: メソッドの縮退例

上記の 2 に当てはまるのは Rectangle_v4 に対して抽象値 $Horz$ を入力として setX() が呼び出された場合である．この場合, Rectangle_v4 のインスタンスを Rectangle_v3 のインスタンスに変化させ抽象値 $Horz$ を入力として setX() を実行する. 本研究ではこれを**オブジェクトの縮退による抽象実行**と呼ぶ. 実行前後のオブジェクトの状態は図 3.2 のようになる

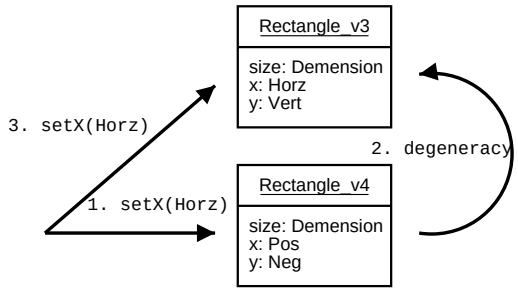

図 3.2: オブジェクトの縮退例

3.2 抽象実行実現の課題

本節では抽象実行の処理系を実現するにあたっての課題について述べる．課題には 2 つの要件と 2 つの問題点がある. 要件は, (1) 発展段階を実行時に変更できること, (2) 抽象実行をあまり意識せずに開発できることである. また問題点は, (1) 発展段階の異なるオブジェクト間の参照の実現, (2) オブジェクトの縮退時の参照の張り替え問題である.

3.2.1 要件

発展段階の実行時の変更

あるメソッドやクラスが実装されているか否は, コンパイル時に知ることができる．このため, 抽象実行をおこなう必要があるメソッド呼出しをあらかじめ判定し, コンパイル時に発展段階を擦り合わせることが可能であると考えられる．しかし, 本研究では発展段階の調節は実行時に行うことにする．それは, すでに完成したプログラムでもあえて抽象実行を行うことでプログラムの理解に役立つと考えているからである.

発展的プロトタイピング技法では最も抽象的なレベルから実装を始め段階的にソフトウェアを構築しているのでその発展の流れを追いかけることによってプログラムの理解が容易になる．例えば, 長方形オブジェクトが Rectangle_v5 まで実装されていたとしても, あえて Rectangle_v1 から徐々に発展させながら, 抽象実行させることによってプログラムの振舞を抽象的なレベルで理解することが可能となる.

抽象実行のためのコードの分離

本研究で発展的に構成するプログラムには, できるだけ抽象実行のためのコードを記述を行わないようにする．抽象実行は, あくまで, テスト実行時に開発段階が異なるオブジェクトが存在した

ときに, 実行を停止することなく最後まで動かすためのものである. このため, 抽象実行に関するコード記述は, 発展的にプロトタイプを構築していく際に常に必要となるわけではない.

したがって, 本研究では抽象実行のためのコードと本来のプログラムコードをなるべく分離できるような設計を試みる.

3.2.2 発展段階が異なるオブジェクトへの参照の実現

本小節では抽象実行を実現するための型システムについて考察する.

Java は強く型付けられた言語であり, コンパイル時に参照先のオブジェクトの型を明示的に指定しなければならない.

前節で抽象実行が発生する例として `Rectangle_v4` のインスタンスに対して, Java の組込み整数である `int` 型の 1 や, 参照型 `AbstHorzDom` で実現される抽象値 $Horz$ を入力として `setX(AbstHorzDom x)` を呼び出していた. しかし, `Rectangle_v4` で定義されている `setX()` の入力の型は `HorzDom` で定義されている. この場合, caller は callee をどのような型として参照すべきか考察する.

以下の, Java の型システムを用いた実現方法について考察する.

- Object 型による参照の実現
- 継承による参照の実現
- Java Interface による参照の実現

Object 型による参照の実現

Java のすべてのクラスは `java.lang.Object` クラスを継承している. このため, すべてのオブジェクトは, `Object` 型として参照することが可能である. `Object` クラスで実装されていないメソッドは Java のリフレクション機能 [5] を用いて実行することが可能である. 例えば, `Rectangle_v4` のインスタンスに対して `int` 型の 1 を入力として `setX()` を呼び出す場合は, `obj` を `Rectangle_v4` のインスタンスであるとすると,

```
Class  clazz = obj.getClass();
Method meth  = clazz.getMethod("setX", new Class[]{int.class});
meth.invoke(obj, new Integer(1));
```

と記述すればよい.

しかしながら, リフレクションによるメソッド呼出しの記述は Java プログラムの一般的な記述法とは異なるため, 開発者は抽象実行を利用するために特殊なコーディングをしなければならない. また, `Object` 型で宣言を行った場合, コンパイル時に型チェックを行うことができない.

継承による参照の実現

発展的プロトタイピング技法では, 発展後のオブジェクトが発展前オブジェクトの機能を包含すると考えられるので, 発展関係を継承によって実現する技法について考察する.

Rectangle_v4 と Rectangle_v3 が継承関係にあるとすると, Rectangle_v4 のインスタンスは, 参照型 AbstHorzDom を入力の型とするメソッド setX() も継承しているため, そのメソッドの縮退による抽象実行が可能となる. しかし, Rectangle_v5 で実装される予定の int を入力の型とするメソッド setX(int x) は実装していない. このため, オブジェクトの縮退による抽象実行は実現できない.

また, 発展的プロトタイピング技法の形式化は継承のないフラットなオブジェクトシステムで考えられているが, あくまで複雑なオブジェクトシステムのための形式化の第 1 歩であり, 将来的に拡張され得るものである. Java では多重継承が認められていないので, あるクラスは 1 つの親クラスしか直接継承できない. そのため, 不用意に抽象実行処理系の実現のためだけに継承を用いることは適切ではない.

Java Interface による参照の実現

Java の参照型にはクラス型以外にインタフェース型が用意されている. これは抽象データ型としてのインタフェースを記述したもので, 抽象メソッドを記述するためのものである. また Java インタフェースはクラスの継承とは異なり, あるクラスが同時複数の Java インタフェースを implements することが許されている.

Rectangle_v4 のインスタンスに対して, メソッドの縮退による抽象実行や, オブジェクトの縮退による抽象実行を実現するための Java インタフェース Rectangle は以下のように定義できる.

```
interface Rectangle {
  void setX(AbstHorzDom x);
  void setX(VertDom x);
  void setX(int x);
}
```

したがって, Rectangle_v4 はインタフェース Rectangle を implements することで抽象実行のための参照を他オブジェクトに提供することが可能となる.

但し, インタフェースを実装するクラスはインタフェースで宣言された抽象的メソッドをすべて実装しなければ実体化することはできない. したがって, Rectangle_v4 で実際に実装すべきメソッド setX(VertDom x) 以外に他クラスで実装済の setX(AbstHorzDom x), setX(int x) を実装する必要がある.

3.2.3 抽象実行時の参照張り替え

オブジェクトの縮退による抽象実行が発生した場合, callee は caller が呼び出したメソッドを実装するクラス, つまり自分よりも抽象的なクラスのインスタンスを生成し, それに対して再度メソッドをディスパッチする必要がある. そして, これ以降の処理は新しく生成したオブジェクトが行わなければならない. なぜならば, オブジェクトの縮退によって情報が失われるため, もとのオブジェクトの状態は不定となるからである.

例えば, Rectangle_v4 のインスタンスに対して 抽象値 Horz を入力として setX() が呼び出された場合, オブジェクトは Rectangle_v3 のインスタンスに縮退され, メソッド呼出しの結果, 抽象値 $Horz$ がセットされる. しかし, そのメソッド呼出しの後の Rectangle_v4 のインスタンスのインスタンス変数 x の値は, {Neg, Zero, Pos} のうちのどの値をセットすればよいかは決定できない.

したがって, 抽象実行以降の処理を継続するために, 縮退前のオブジェクトに対してメソッド呼出しが行われないように, システム全体において縮退前オブジェクトへ参照を持つオブジェクトは縮退後オブジェクトを参照するように変更する必要がある.

smalltalk ではあるオブジェクトに対して `allOwners` というメッセージを送ることで, そのオブジェクトを参照しているすべてオブジェクトを得ることができる. しかし,Java ではそのような機能は実現されていないので, プログラマが実装する必要がある. そのような機能の実現は, 煩雑な処理が必要であり, またシステム中のオブジェクトが増えると, その参照張り替えのコストを無視できなくなる.

3.3 設計

この節では, 前節で述べた課題をふまえ, 抽象実行をおこなうためのプロキシオブジェクトの提案を行う. その後で, 抽象実行時におけるプロキシオブジェクトの振る舞いを示す.

3.3.1 プロキシオブジェクトの提案

プロキシオブジェクトとはあるオブジェクトの代理となるオブジェクトである. 抽象実行におけるプロキシオブジェクトは, 発展的に構成されたメソッド集合 $Meth$ へのアクセスを caller に提供するために, $Meth$ を実装するオブジェクト群の代理を行う. プロキシオブジェクトは, $Meth$ の全てのシグネチャを持ち, その処理を実際の callee に委譲することで, caller は発展段階の異なる callee に対してメソッド呼出しを実行することが可能となる.

再び, `Rectangle` の例を用いてプロキシオブジェクトによって抽象実行がどのように実現されるかを図 3.3 示す.

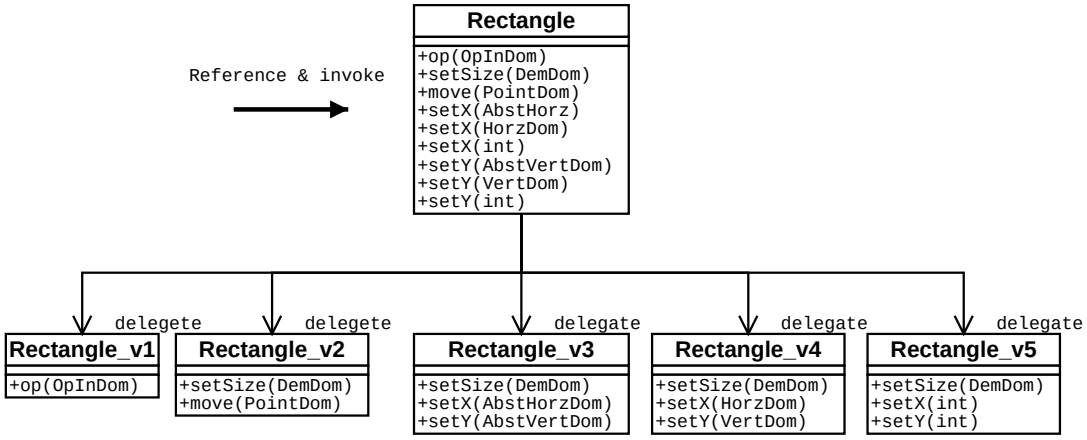

図 3.3: プロキシオブジェクト

プロキシオブジェクトクラス `Rectangle` は `Rectangle_v1` から `Rectangle_v5` までの全てのメソッドを持つ. caller からの呼出しは, プロキシオブジェクトによって, 実際に処理を行うオブジェクトに適切に委譲される. また, caller は callee に対して直接参照を持たないため, 参照の張り替え問題を回避することが可能となる.

また, 開発者の負担を減らすため, プロキシクラスは開発者が記述するのではなく自動生成をうものとする. さらに, 発展レベルをコンパイル時ではなく, 実行前に決定できるような仕組みを持たせるようにするため, 発展に関する情報はプログラムとは別に記述し, 実行時にプロキシオブジェクトによって読み込まれるようにする.

3.3.2 プロキシオブジェクトの設計

この小節では, プロキシオブジェクトの生成から抽象実行までの振る舞いについて述べる.
プロキシオブジェクトのライフサイクルは以下のとおりである.

1. プロキシオブジェクトの生成
2. プロキシオブジェクトに対するメソッド呼出し
3. メソッド呼出しの失敗のトラップ
4. オブジェクトの縮退
5. メソッドの縮退

プロキシオブジェクトの生成

抽象実行時に caller となるオブジェクトは, callee を直接生成せずに代りにプロキシオブジェクトを生成する. このとき, プロキシオブジェクトは代理する対象となるクラス群の中で最も発展しているクラスのインスタンスを生成する. クラスの発展関係は, 半順序関係を満し, 木構造を構成すると仮定している. このため, 単純に最も発展しているものを判定することはできない. 本研究では, 抽象実行の実行者が図 3.4 のようにして, 全クラスに全順序関係を与えてやることによって最も発展しているクラスを決定し, クラスのインスタンス化を行う.

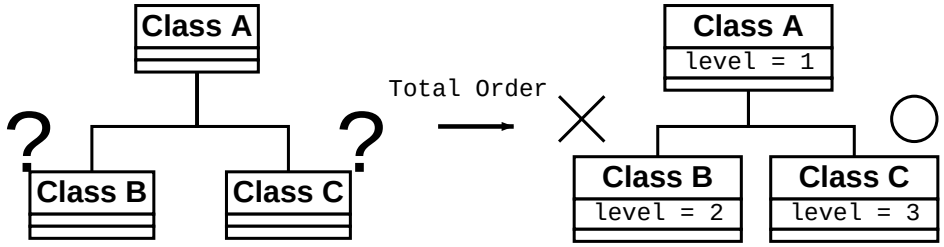

図 3.4: 最も発展したクラス

メソッド呼出し

caller が callee のメソッドを呼び出す場合は, 直接 callee に対して行われるのではなく, プロキシオブジェクトに対して行われる. そしてその呼出しはプロキシオブジェクトによって callee に委譲される.

この時, callee に対してメソッド呼出しが成功すれば, プロキシオブジェクトは caller にメソッド呼出しの戻り値を返し, メソッド呼出しに失敗した場合はエラーを返す.

メソッドの呼出しに成功するとは,

- そのメソッド呼出しにおいて, 抽象実行を行うことなく, 呼出しに成功し戻り値を得る場合か,
- そのメソッド呼出しにおいて, 抽象実行によって, 呼出しに成功し戻り値を得る場合

である.

また, メソッドの呼出しに失敗する場合とは,

- そのメソッド呼出しにおいて, 抽象実行が行えない場合

である.

抽象実行の発生

プロキシオブジェクトは以下の場合に抽象実行を試みる.

- 呼び出し対象のメソッドが callee で実装されていない場合
- callee のメソッドから呼び出しているメソッドの呼出しが失敗した場合

callee で, caller が要求するメソッドが実装されていない状態は以下の 4 つに整理できる.

1. メソッドが callee より発展したクラスで定義されている
2. メソッドが callee より抽象的なクラスで定義されている
3. メソッドが callee とは異なる発展をしたクラスで定義されている
4. メソッドが callee で未実装である

上記のそれぞれの対応方法を以下に示す.

1. プロキシオブジェクトはメソッドの縮退による抽象実行を試みる (図 3.5). (1) まず caller はプロキシオブジェクトに対してメソッド $meth$ の呼出しを行なう. (2) プロキシオブジェクトは callee に対して $meth$ の呼出しを委譲する. (3) プロキシオブジェクトは callee が $meth$ を実装していないことを検出すると, $meth$ を $meth^{\#}$ に抽象化し, callee に対して再度呼出しを行なう. このとき, メソッドの引数 D も, $meth^{\#}$ の引数の型に合わせて $D^{\#}$ に抽象化する. (4) プロキシオブジェクトは caller に対して, $meth^{\#}$ の戻り値を返す.

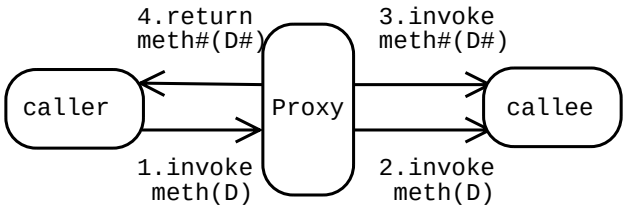

図 3.5: メソッドの縮退

メソッドの縮退では, 戻り値は抽象化は行なわない. 戻り値を抽象化して返してしまうと caller が対応できないからである. また, caller が呼びだしたメソッドが直積分割された一つであっ

た場合もメソッドの縮退をおこなわない. なぜならば, 直積分割されたメソッドの入力値は, ある抽象値を組に分割したものであり, 個別に抽象化することはできないからである.
従って, 上記の 2 つ場合は, メソッドの縮退は失敗となり, 抽象実行を行えずプロキシオブジェクトは caller に対してエラーを返す.

2. オブジェクトの縮退による抽象実行を試みる (図 3.6). (1) まず caller はプロキシオブジェクトに対してメソッド $meth$ の呼出しを行なう. (2) プロキシオブジェクトは callee に対して $meth$ の呼出しを委譲する. (3) プロキシオブジェクトは callee が $meth$ を実装していないことを検出すると, callee を callee$^{\#}$ に縮退する, このとき, callee のインスタンス変数 D も, callee$^{\#}$ のインスタンス変数の型に合わせて $D^{\#}$ に抽象化する. (4) プロキシオブジェクトは callee$^{\#}$ に対してメソッド $meth$ の呼出しを委譲する. (5) プロキシオブジェクトは caller に対して, $meth$ の戻り値を返す.

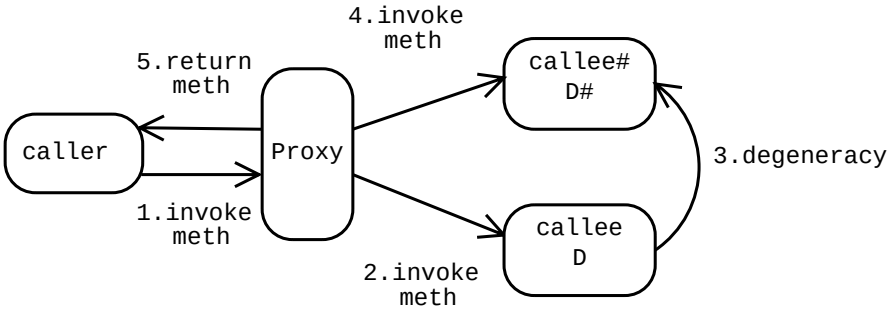

図 3.6: オブジェクトの縮退

3. このような状況は, 呼出し対象となるメソッドのデータドメインに, 複数の異なる発展をしたデータドメインが存在する場合である. 図 3.7 では, 抽象値 $Horz$ が 2 つの異なる発展をした例である.

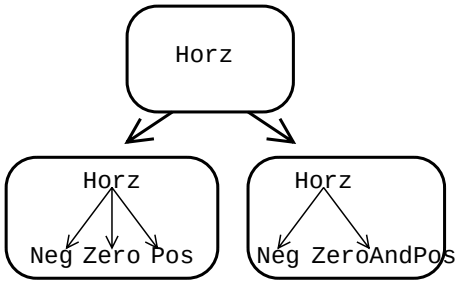

図 3.7: もう一つの発展例

`Rectangle_v4` のメソッド `setX()` は引数として {Neg, Zero, Pos} のいずれかを受け取るように定義されており, 別の発展として, `Rectangle_v4a` ではメソッド `setX()` の引数として {Neg, ZeroAndPos} のいずれかを受け取るように定義されているとする. この場合, `Rectangle_v4`, `Rectangle_v4a` の間に順序関係がないので, オブジェクトの縮退やメソッドの縮退によって抽象実行ができない. よってメソッド呼出しは失敗となり, プロキシオブジェクトは caller にエラーを返す.

4. これは, calleeで実装される予定のメソッドがまだ未実装の状態の時に発生する (図 3.8) . この場合, まずメソッドの縮退を試みる, メソッドの縮退が可能ならば, それにあわせてオブジェクトも縮退を行い, 縮退後のメソッドによってメソッド呼出しが行われる.

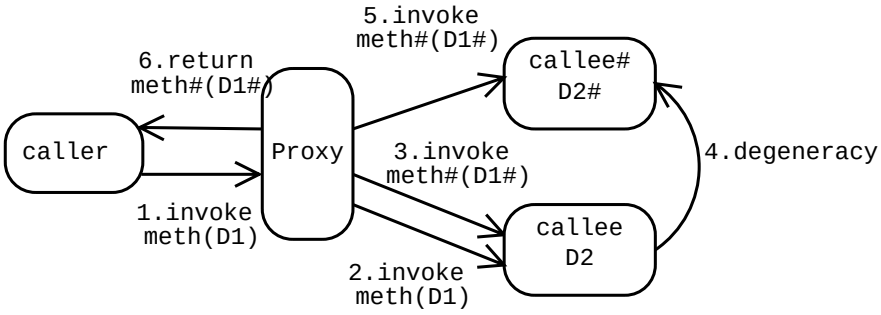

図 3.8: メソッドとオブジェクトの同時縮退

3.4 処理系を実現する諸技術

本節では, 抽象実行を行うためのプロキシオブジェクトを実現するための諸技術について述べる.

3.4.1 リフレクション

抽象実行を行うためには, プログラムの発展状況に応じて, 実行時にそのプログラムの構成を再構成する必要がある. 本研究では, その再構成を Java が提供するリフレクション機能 [5] を用いて実現する. リフレクションとは, 計算システムが, 自分自身の構成や計算過程に関する計算を行うことである [6]. Java の Reflection API では, クラスやメソッド, フィールドなどを Java のオブジェクトとして扱うことができる. これらのメタオブジェクトは, `Class` クラスというメタクラスから得ることができる.

クラスのメタクラスは, そのクラスのインスタンスを生成するメソッド `newInstance()` が用意されており, インスタンスの生成機構をプログラムから利用できる. メソッドのメタクラス `Method` は, そのメソッドを実行するためのメソッド `invoke()` が用意されており, メソッドの実行機構をプログラムから利用できる. また, フィールドのメタクラス `Field` は, そのフィールドの値を参照したり, 変更したりするメソッド `set()`, `get()` が用意されている.

3.4.2 Dynamic Proxy

抽象実行のためのプロキシオブジェクトは Java のダイナミックプロキシ機能 [7] を用いて実現する. ダイナミックプロキシとは, Java のリフレクション機能の一つであり, Java 2 Platform のバージョン 1.3 より導入されている. この機能はある Java インタフェース集合を実装するプロキシクラスやインスタンスを実行時に生成する.

まず, 一般的なプロキシパターン [8] のクラス図とオブジェクト図を図 3.9 に示す.

プロキシパターンは, `Client` が `RealSubject` のメソッド `request()` を呼び出すように設計されていたときに, `RealSubject` のメソッドのシグネチャを定義したインタフェース `Subject` を用

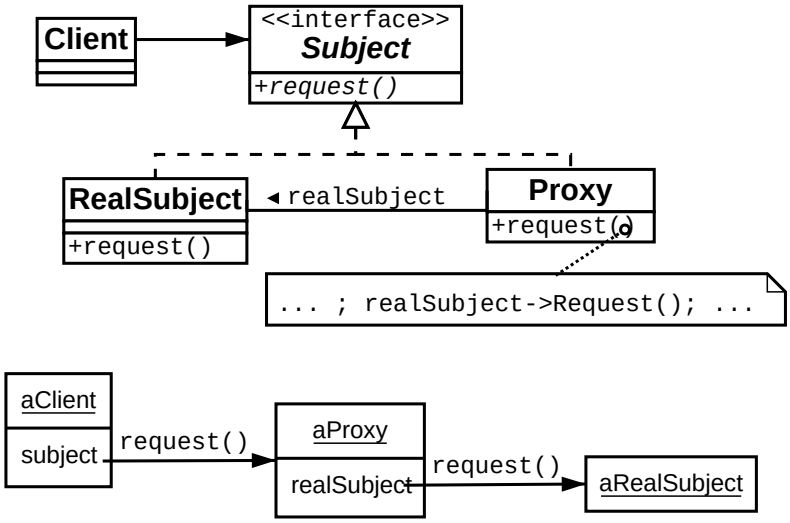

図 3.9: プロキシパターン

意し, それを実装した Proxy のメソッドを経由して Client が RealSubject のメソッドを呼び出すようにする. このような構造にすることによって, request() の本来の機能とは別に, 実行前後に処理を割り込ませることが可能となり, 例えば, メソッド呼び出しのログをとるような機能を実現できる.

しかし, ログをとるような共通の機能を実現するために, 全てのクラスに対してプロキシクラスをあらかじめ作成することは現実的でない. そのような場合にダイナミックプロキシ機能は効果的である. ダイナミックプロキシクラスは, ログの対象となるメソッドのシグネチャを記述したインタフェースと, ロギングの処理を実現する呼出しハンドラから, 実行時にプロキシオブジェクトを生成する (図 3.10).

java.lang.reflect.Proxy のメソッド newProxyInstance() は, ある Java インタフェースと java.lang.reflect.InvocationHander インタフェースを実装した呼出しハンドラオブジェクトから実行時にプロキシオブジェクトを生成する (図 3.10). LoggerProxy のメソッドボディは, Subject インタフェースのメソッドシグネチャと Logger の invoke() メソッドを合成して生成される. したがって, LoggerProxy オブジェクトに対する全てのメソッド呼出しは, Logger の invoke() メソッドによって実現される.

この技術のメリットは, Java インターフェースを入れ替えることによって実行時に, 様々なプロキシを生成することが可能となるため, 複数の専用のプロキシをあらかじめ作成する必要がないことである. 本研究では, これを利用して, 発展関係にあるクラス群ごとにプロキシを用意するのではなく, 共通のハンドラを用意することで, 抽象実行のためのプロキシの実現を試みる. また, 各クラス群ごとの抽象実行のための情報は XML によって与えることで, ハンドラの共通化を実現している.

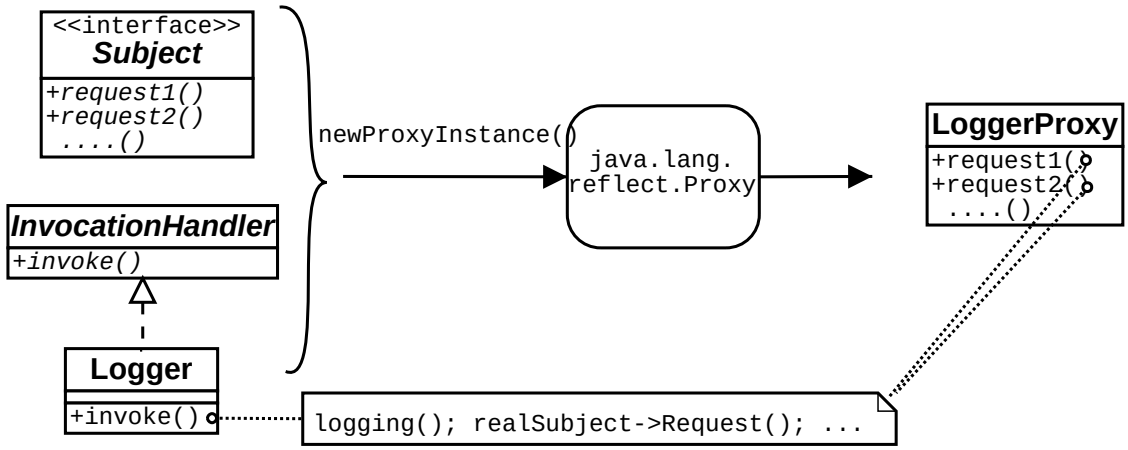

図 3.10: Dynamic Proxy

3.4.3 XML

抽象実行を行うための発展関係に関する情報は XML 文書によって与えられる. XML(eXtensible Markup Language)[9] は汎用的なデータ記述言語であり, データの意味を表すタグ名とそれを補足する属性, そして入れ子による論理構造を用いて, 構造化されたテキストデータを表現する. XML は本質的に木構造のデータの表現に優れている.

本研究で扱う発展関係はすべて木構造になると仮定しているため, 発展関係を記述するデータ構造として XML は有用であると考えている. また, XML は, ID/IDREF 属性を用いることで網構造にすることも可能である. ID 属性とはある XML 文書内である要素を一意に識別するものであり, IDREF は ID 属性を用いて要素から要素への参照を実現する. 本研究では, オブジェクトが扱う値, クラス, フィールド, メソッドの発展関係と個々の要素間の関係を, 入れ子による木構造と ID/IDREF を併用することで表現している.

そして, それらの関係を DTD(Document Type Definition) を用いて定義を行った. DTD は, XML 文書で使用されるタグの種類やその属性, 階層構造, 出現順序などを定義したものである. DTD で定義された規則に従って記述された XML 文書は妥当な XML 文書 (Valid XML Document) と呼ばれる. DTD を定義することで, 妥当な XML 文書か否かを事前にチェックすることが可能となり, 実行時のエラーを減らすことが可能となる.

本研究では XML の処理は DOM(Document Object Model) を使用する. DOM は, XML 文書をアプリケーションから操作するための API の規格である. Java 2 Platform ではバージョン 1.4 より DOM に準拠した API が標準 API として導入されていため, 標準的な Java 開発環境において XML 文書の処理が可能である.

DOM の特徴は, ある XML 文書全体を, XML パーサーを用いて「DOM ツリー」と呼ばれる, 木構造のオブジェクトとしてメモリ中に展開することである. このため, XML 文書の記述順序に関係なく XML 文書全体にアクセスすることが可能となる. また, DOM ツリーのルートとなる `Document` オブジェクトは, メソッド `getElementById()` を持つ. このメソッドによって, DOM ツリーを探索するすることなく, ID 属性をもつ要素を取得することを可能としている.

3.5 発展関係定義文書の設計

本節では, 抽象実行時にプロキシオブジェクトによって読み込まれる, 発展関係を XML 文書を記述するためのデータスキーマの設計について述べる.

DTD で定義した主な要素を表 3.1 に示す. DTD の完全な定義は付録に記す.

表 3.1: 発展関係定義文書の XML 要素

dataDomain	データドメインを表現する, 具体集合のデータ要素からなる
data	データを表現する要素
range	複数のデータ要素を略記するための要素
classDomain	発展関係にあるクラス集合の順序関係を表現する要素
class	クラスを表現する要素, フィールド要素とメソッド要素からなる
field	クラスのインスタンス変数を表現する

データ要素

データ要素 <data>は, 子要素として Java における値としての情報を表す要素を保持する.

データ要素の例として抽象値 $Horz$ の XML 表現を以下に示す

```
1  <data dataID="AbstHorzDom.Horz" evolved="PointDom.Point" name="Horz">
2    <complexValue>
3      <type>
4        <referenceType name="AbstHorzDom"/>
5      </type>
6      <initValue>
7        <simpleValue value="Horz">
8          <type>
9            <referenceType name="java.lang.String"/>
10         </type>
11        </simpleValue>
12     </initValue>
13   </complexValue>
14 </data>
```

1 行目の dataID は ID 型の属性であり, 'データドメイン名. データ名' で表される. evolved は IDREF 型の属性であり $Horz$ を抽象化した $Point$ のデータ要素を参照する. 2 行目から 13 行目までは $Horz$ が実際に Java プログラム上でどのような値として表現されているかを示す. この DTD では Java の値は<simpleValue>要素と<complexValue>要素で表される. 前者は, 組み込みの整数値 int などのプリミティブ型の値と文字列を指す. 後者は文字列を除く参照型の値を指す. <complexValue>要素は子要素として<initValue>要素を持ち, この要素以下に記述された値をコンストラクタに渡すことで値を生成できることを示している.

範囲要素

範囲要素 <range>は, 複数のデータ要素を記述を省力化する要素である. 子要素として具体値の開始値と終了値を表す値をもつ.

以下に抽象値 Pos を Java の組み込み整数の 1 から 100 に具体化した例を示す.

```
1  <range evolved="HorzDom.Pos">
2    <from>
3      <simpleValue value="1">
4        <type>
5          <primitiveType name="int"/>
6        </type>
7      </simpleValue>
8    </from>
9    <to>
10     <simpleValue value="100">
11       <type>
12         <primitiveType name="int"/>
13       </type>
14     </simpleValue>
15   </to>
16 </range>
```

1 行目の evolved は IDREF 型の属性であり抽象値のデータ要素を指す. 2 行目から 8 行目は表すデータの範囲の開始値を示す. 9 行目から 14 行目は表すデータの範囲の終了値を示す. <range>要素を用いることで<data>要素の記述を減らし, XML 文書の記述を削減することが可能となる.

データドメイン要素

データドメイン要素<dataDomain>は, 具体集合となるデータ要素の定義とそのデータドメインから発展したデータドメインを表す.

以下にの具体集合として {Neg, Zero, Pos} をもつ $HorzDom$ の XML 表現を示す.

```
1  <dataDomain dataDomainID="HorzDom">
2    <type>
3      <referenceType name="HorzDom"/>
4    </type>
5    <data dataID="HorzDom.Pos"  ...> ... </>
6    <data dataID="HorzDom.Zero" ...> ... </>
7    <data dataID="HorzDom.Neg"  ...> ... </>
8    <evolvedDataDomain>
9      <reifiedDataDomain>
10       <dataDomain dataDomainID="ConcHorzDom"> ... </></></></>
11 </dataDomain>
```

1 行目の dataDomainID は ID 型の属性である. 2 行目から 4 行目はデータドメインの型を表す. 5 行目から 7 行目はデータドメインの具体集合となるデータ要素である.8 行目から 10

行目は, $HorzDom$ のそれぞれの抽象値を整数値に詳細化することで発展したデータドメイン $ConcHorzDom$ を表している. あるデータドメインから発展したデータドメインは<evolvedDataDomain>要素の子要素になる. そして, そのデータドメインの発展の種類によってタグづけが行われる. 詳細化しただけならば <reifiedDataDomain>, 組に分割されたならば <andDivededDataDomain>, 直和に分割された場合は<orDivededDataDomain> とそれぞれタグづけられる.

クラスドメイン要素

クラスドメイン要素<classDomain>は, ルートとなるクラス要素とそのクラスから発展したクラス群を表す<evolvedClass>要素からなる. この要素がプロキシクラスを生成する単位となる.

以下に, Rectangle_$*$ の発展の様子を表す XML を示す.

```
1  <classDomain classDomainID="Rectangle">
2    <class classID="Rectangle_v1" ...></>
3    <evolvedClass>
4      <class classID="Rectangle_v2" ...></>
5      <evolvedClass>
6        <class classID="Rectangle_v3" ...></>
7        <evolvedClass>
8          <class classID="Rectangle_v4" ...></>
9          <evolvedClass>
10           <class classID="Rectangle_v5"...></>
11         </>
12       </>
13     </>
14   </>
15 </classDomain>
```

1 行目の classDomainID は ID 型の属性である. 2 行目の<class>要素が, このクラスドメインで最も抽象的なオブジェクトを構成するルートクラスとなる. 3 行目から 14 行目までは, ルートクラスから発展したクラスを<evolvedClass>タグで表現している.

クラス要素

クラス要素<class>は, そのクラスで定義されているフィールド要素とメソッド要素で構成される.

以下に, クラス Rectangle_4 を XML で表現したものを示す.

```
1 <class classID="Rectangle_v4" evolved="Rectangle_v3" level="3">
2   <field name="x"         ... />
3   <field name="y"         ... />
4   <field name="size"      ... />
5   <method name="setSize" ... />
6   <method name="setX"    ... />
7   <method name="setY"    ... />
8 </class>
```

1 行目の classID は ID 型の属性である. evolved は IDREF 型の属性であり, このクラスを一つ抽象化したクラス要素を参照する. level 属性はプロキシオブジェクトによって, クラスドメイン内で最も発展したクラスの判定するために, 使用される. この属性値ではルートクラスの発展レベルを 0 として考えているため, この属性をマイナスの値にすることによって, そのクラスのインスタンス化を抑制し, 任意のレベルで抽象実行を行うことが可能となる. 2 行目から 4 行目は, このクラスで定義されているフィールドである. また 5 行目から 7 行目は, このクラスで定義されているメソッドである.

フィールド要素

クラス Rectangle_4 のフィールド x は以下のように XML で表される.

```
1 <field name="x" fieldID="Rectangle_v4.x"
2        domain="HorzDom" evoled="Rectangle_v3.x"/>
```

1 行目の fieldID は ID 型の属性であり, 'クラス名. フィールド名' で表現される. 2 行目の domain 属性 は IDREF 型であり, データドメイン要素かクラスドメイン要素を参照する. したがって, domain 属性がクラスドメイン要素を参照している場合は, このフィールドはオブジェクト間の関連である. domain 属性がデータドメイン要素を参照する場合は, このフィールドはオブジェクトの状態値を保持している.

メソッド要素

クラス Rectangle_4 のメソッド setX は以下のように XML で表される.

```
1 <method name="setX" methodID="Rectangle_v4.setX"
2         evolved="Rectangle_v3.setX" paras="HorzDom" available="no"/>
```

1 行目の methodID は ID 型の属性であり, 'クラス名. メソッド名' で表現される. 2 行目の evolved 属性は IDREF 型であり, 発展元のメソッド要素を参照する. paras 属性は IDREFS 型であり, 引数となる複数のデータドメイン要素への参照となる. available 属性は, メソッドが実装済の時に 'yes' となり, 未実装の時は 'no' となる. この属性によって, 抽象実行をメソッドレベルでコントロールすることが可能となる. また, 戻り値が void 型でない場合は, データドメイン要素を参照する IDREF 型の return 属性を記述する必要がある.

このメソッドの XML 表現で注目すべき事は, メソッドの発展に関する情報は evolved 属性のみであることである. したがって, メソッドの発展が詳細化, 直和分割または直積分割のいずれであるかは直接は明示されない. この区別は, 引数のデータドメイン要素を参照する paras 属性や戻り値データドメインを参照する return 属性によって判定される.

また, Java にはメソッド名を同じで, 仮引数の型が異なるメソッドを定義することを可能とするオーバーロード機能があるが, 本研究では扱っていない. しかしこれは本質的に問題があるわけではない. メソッドのオーバーロードは, methodID 属性に型情報を含ませ, ユニークな ID を生成することで実現が可能である.

3.6 抽象実行を実現するクラス

構成した抽象実行処理系の UML クラス図を図 3.11 に示す.

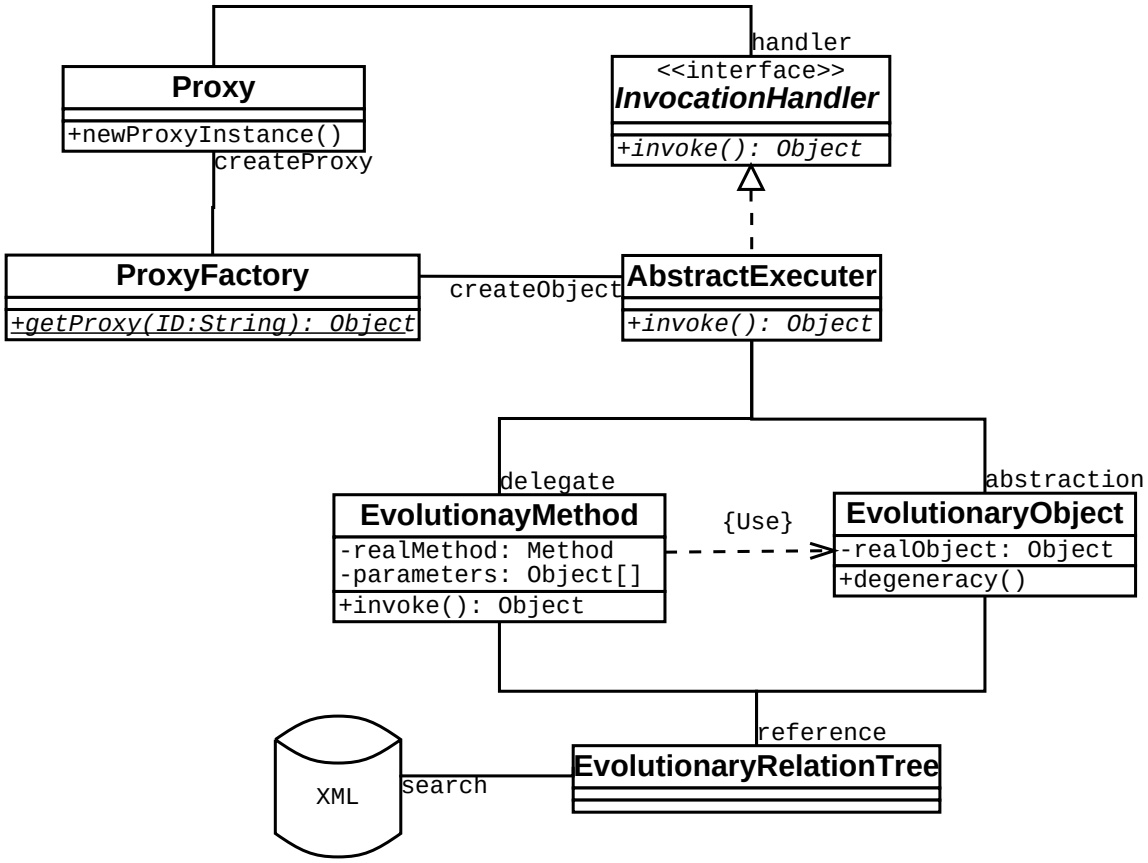

図 3.11: 抽象実行処理系のクラス図

ProxyFactory クラス

プロキシオブジェクトを生成するクラスである. プログラムの開発者は他のオブジェクトを生成するコードを記述する場合に, このクラスに対して, オブジェクトの生成を要求しなければならない. このクラスによって開発者は抽象実行のためのプロキシオブジェクトのの具体的な生成方法を知ることなく抽象実行の仕組みが使用可能となる.

AbstractExecuter クラス

プロキシオブジェクトを生成する際の呼出しハンドラのクラスである.
java.lang.reflect.InvocationHander インタフェースを実装し, 唯一の public メソッドとして invoke() を実装する. プロキシオブジェクトへのメソッド呼出しはすべてこの invoke() が処理することになる.

EvolutionaryObjecrt クラス

プロキシされる発展的に構成されたオブジェクトをラップするクラスである. 抽象実行時にオブジェクトの縮退を行うメソッド degeneracy() を持つ.

EvolutionaryMethod クラス

発展的に構成されたオブジェクトのメソッドをリフレクトする Method オブジェクトとメソッドの入力となる引数オブジェクト配列をラップするクラスである. 抽象実行時に Method オブジェクトと引数を抽象化させるメソッド degeneracy() を持つ.

EvolutionaryRelationTree クラス

オブジェクトやメソッドの縮退時に, 発展関係が格納された XML DOM ツリーを走査し, 縮退に必要な情報を提供するヘルパークラスである.

3.7 抽象実行処理

3.7.1 プロキシの生成

プロキシは以下の手順によって生成される.

プロキシ生成の手順

1. プロキシインタフェースのロード
2. 呼出しハンドラの生成
3. プロキシインスタンスの生成

プロキシインタフェースのロード

プロキシを要求するするオブジェクトは,ProxyFactory クラスのクラスメソッド getProxy() を呼び出す. このとき生成したいオブジェクトが所属するクラスドメイン名を指定する. クラスドメイン名は, あるクラスドメイン内のクラス集合がもつすべてのメソッド集合のシグネチャが定義された, Java インタフェースに 1 対 1 に対応する. その Java インタフェースを Class クラスのクラスメソッドである forName() によってロードする.

呼出しハンドラの生成

呼出しハンドラオブジェクトとなる AbstractExecuter のコンストラクタは, クラスドメイン名を受取り, そのドメイン内で最も発展したクラスのインスタンス生成する. そして生成したオブジェクトとその発展関係情報をラップする EvolutionaryObject のインスタンスを生成し, それに対する参照を保持する. EvolutionaryRelationTree は最も発展しているクラスを, XML の<class> 要素がもつ level 属性によって判断している.

プロキシインスタンスの生成

最後に, 1. でロードした Java インタフェースと, 2. で生成した呼出しハンドラを入力として, java.lang.reflect.Proxy クラスのクラスメソッド newProxyInstance() を呼び出すことで, プロキシオブジェクトを生成する.

3.7.2 メソッドの呼出し

プロキシインスタンスに対するメソッド呼出しは, すべて AbstractExecuter オブジェクトの invoke() メソッドにディスパッチされる.

invoke() の処理は以下の流れで進められる.

メソッド呼出の手順

1. メソッドの識別
2. EvolutionaryMethod オブジェクトの生成
3. メソッド呼び出し
4. 抽象実行処理

メソッドの識別

プロキシオブジェクトから実オブジェクトへの呼出しは EvolutionaryMethod からリフレクションによって行われるため, 実オブジェクトの Method オブジェクトを取得する必要がある. invoke メソッドは, 実引数としてプロキシのメソッドオブジェクトをとる. このメソッドオブジェクトからメソッドの名前, 戻り値および仮引数の型を取得し, これらの情報をもとに EvolutionaryRelationTree オブジェクトが XML DOM ツリーを探索することで, 呼び出すべきメソッドを判断する. 探索の入り口は, AbstractExecuter が参照してるオブジェクトが所属するクラスドメインで最も発展し

ているクラス要素であり, 発展木の葉となるクラスである. 探索は葉から根に向かって行われる.

EvolutionaryMethod オブジェクトの生成

呼び出すべきメソッドが識別できた場合, そのメソッドが実装されているクラスオブジェクトから Method オブジェクトを取得する. そして Method オブジェクトとメソッドの引数であるオブジェクト配列そしてメソッドの発展情報をラップする EvolutionaryMethod インスタンスを生成する. EvolutionaryMethod によって Method オブジェクトは縮退可能なリフレクションオブジェクトとして扱うことが可能となる.

メソッド呼び出し

メソッド呼出しはすべてリフレクションによって行われる. EvolutionaryMethod の実行は, EvolutionaryObject を入力として invoke() メソッドを呼び出すことで実現される.

エラー処理

上記の処理中に発生したエラーとその処理は以下の通りである.

- **メソッド識別エラー**

  EvolutionaryRelationTree が XML DOM ツリーの探索において, 一致する Method オブジェクトを見つけられなかった場合である. この場合は, プロキシオブジェクトは caller に対してエラーを返す.

- **メソッドが抽象的**

  XML 文書の探索において, Method オブジェクトが, 現在の EvolutionaryObject が保持しているオブジェクトのクラスより根に近いクラスで見つかった場合である. この場合, Method オブジェクトを実装するクラスのオブジェクトまで縮退することで, 実行が可能となる. したがって, この場合は EvolutionaryObject の縮退メソッド degeneracy() を実行し, プロキシオブジェクトに対して再帰的にメソッドを呼び出す.

- **メソッドが具体的**

  XML 文書の探索において, Method オブジェクトが, 現在の EvolutionaryObject が保持しているオブジェクトのクラスより葉に近いクラスで見つかった場合である. この場合, Method オブジェクトを現在オブジェクトが実装している抽象的な Method オブジェクトにまで縮退することで実行が可能となる. したがって, この場合は EvolutionaryMethod の縮退メソッド degeneracy() を実行し, プロキシオブジェクトに対して縮退後のメソッドを呼び出す.

  但しこれが可能な条件は, メソッドの戻り値のデータドメインが発展していなくて, そのメソッドが直積分割されたメソッドの一部でない場合である. これらの条件を満さない場合は caller に対してエラーを返す.

- **メソッド呼出しに成功したが,callee においてエラーが発生した**
  この場合は,「メソッドが具体的」の場合と対応は同様である.

3.7.3 オブジェクトの縮退

オブジェクトの縮退は EvolutionaryObject のメソッド degeneracy() で行われる. 縮退の具体的な手順を以下に示す.

オブジェクトの縮退手順

1. 縮退後クラスのインスタンス化
2. 縮退前オブジェクトのフィールド値の識別
3. 値の抽象化
4. 縮退後オブジェクトのフィールドへ値をセット
5. 他オブジェクトへのリンクのセット

縮退後クラスのインスタンス化

まず, 縮退前のオブジェクトに対してメソッド getClass() を呼び出すことで縮退前の Class オブジェクトを取得し, さらにそのオブジェクトに対し, getName() メソッドを呼び出すことでクラス名を取得することができる. クラス名は, XML 文書において, クラス要素を識別できる一意な ID であり, 発展関係を格納した DOM ツリーに対してメソッド getElementByID() を呼び出すことでクラス要素を取得することが可能となる. そして, このクラス要素の evolved 属性から次に縮退すべきクラス名を得ることができる. java.lang.Class のクラスメソッド forName() によって縮退後クラスをロードし, newInstance() メソッドによって縮退後オブジェクトをインスタンス化できる.

縮退前オブジェクトのフィールド値の識別

縮退前オブジェクトの Class クラスから getDeclaredField() メソッドによって縮退前オブジェクトが持つすべて Field オブジェクトを取得する. フィールドが保持する値はメソッド get() によって取得することが可能である.

値の抽象化

縮退後クラスのフィールドのデータドメインに合わせて, ステップ 2 で取得したデータを抽象化する. この処理は<data>要素に格納された情報をもとにクラスのロード, インスタンス化を行う.

縮退後オブジェクトのフィールドへ値をセット

縮退後オブジェクトの Class クラスから getField() メソッドによって Field オブジェクトを取得し, メソッド set() によって抽象化した値を設定する.

他オブジェクトへのリンクのセット

縮退前オブジェクトがもつ他オブジェクトへのリンクはそのまま縮退後オブジェクトのフィールドにセットする. 他オブジェクトへのリンクはすべてプロキシオブジェクト経由で行われているので実行時に型エラーが発生することはない.

第4章　開発環境の設計と実装

4.1　開発環境の概要

発展的プロトタイピング技法のための開発環境の全体図を図 4.1 に示す.
構築した開発環境は以下の 3 つで構成される.

- 発展関係エディタ (左上)
- 抽象実行処理系 (左下)
- 抽象実行ビジュアライザ (右下)

本研究では, この開発環境を用いて以下の流れでプロトタイプの構築を行う.

1. 発展関係エディタを用いて, 発展的に構成するクラスやそのクラスで使用する値の関係を XML 文書に記述する.
2. 1. で記述された情報からプログラムの雛形と抽象実行プロキシのための Java インタフェースを生成する.
3. テキストエディタを用いて, 2. で生成された雛形の内部ロジックを記述する.
4. 3. で作成した完全なソースコードを, 2. で生成した Java インタフェースとともにコンパイルし, Java プログラムのバイトコードを生成する.
5. 前章で説明した抽象実行プロキシオブジェクトとともに, 4. で生成したプログラムを抽象実行する.
6. 必要があれば, 抽象実行ビジュアライザを用いて抽象実行の振る舞いを確認する.

本章の残りでは, 発展関係エディタと抽象実行ビジュアライザの詳細について述べる.

4.2　発展関係エディタ

発展的プロトタイピング技法では段階的にプログラムを構築していくため, 従来の開発手法より多くのプログラムが必要となる. また前章で提案した抽象実行処理系では発展関係を記述した XML 文書を開発者が記述する必要がある. そこで本節では, XML 文書を記述するためのエディタと, XML 文書から発展的に構成されたプログラムの雛形を生成するシステムを提案する.

4.2.1　エディタ構築の目的

本研究では, 発展関係を記述に特化した XML エディタの構築を行う.

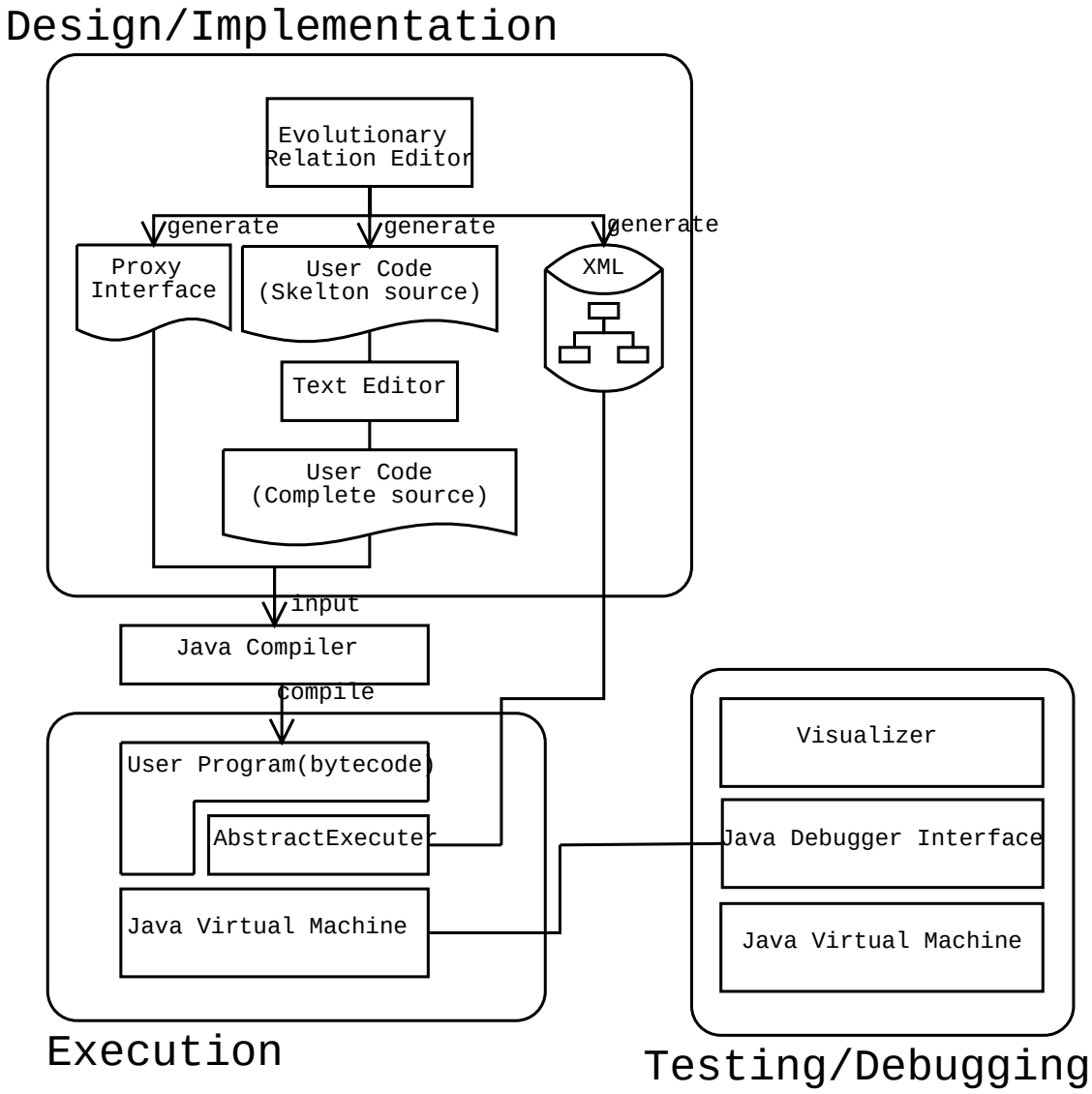
Design/Implementation
Evolutionary
Relation Editor
generate
generate
generate
Proxy
Interface
User Code
(Skelton source)
XML
Text Editor
User Code
(Complete source)
input
Java Compiler
compile
User Program(bytecode)
AbstractExecuter
Java Virtual Machine
Visualizer
Java Debugger Interface
Java Virtual Machine
Execution
Testing/Debugging

図 4.1: 開発環境概要図

XML 文書はがタグつき文書であるため人間にとって理解しやすいが, 一方でそれを人の手で記述する場合には多くの労力を必要とする. また XML 文書が人間に理解しやすいといっても文字情報だけであるため, 情報量が多くなりすぎると人間の理解の限界を超えてしまう.

この問題を解決するために XML エディタが存在する. 多くの XML エディタは単純にデータをタグでマークアップするだけでなく, DTD の妥当性チェックも行う. また GUI を用いて XML 文書を表示し, 必要な情報のみを表示することで要素数が増加しても, その構造の理解を容易にする.

しかしながら, 我々は本研究において一般的な XML エディタの機能では不十分であると考える. それは,

1. DTD の妥当性チェックだけでは発展関係の正しさをチェックできない
2. XML ツリーの視覚化だけでは発展的に構成されたプログラムの表現に限界がある.

からである.

1. の例としては, DTD ではあるメソッドの戻り値データドメインが, その文書の中で必ず定義されていることはチェックできるが, あるメソッドとそれを詳細化したメソッドがあったときに, それらの戻り値データドメインが正しく詳細化関係になっているかということはチェックできない.
2. の例としは, XML 文書をそのままのツリー表現することで, あるクラスの発展関係を木構造で表現することが可能であるが, メソッドの発展関係だけを木構造で表現することはできない.

そこで本研究で提案するエディタは,

1. 発展関係の意味にまで踏み込んだチェックを行ない,
2. 開発者が望む粒度での発展関係の表示を行なう

ことを目的とする.

4.2.2 発展関係の視覚化

構築した発展関係エディタを図 4.2 に示す.

図 4.2: 発展関係エディタ

ウィンドウの左側にクラスドメインとデータドメインの一覧が表示されている. また, ウィンドウの中央部にデータ, データドメイン, クラス, フィールドメソッドのそれぞれの発展関係がツリー状に表示されているのがわかる. またツリーの各ノードから, それらのノードの編集を行うことができる.

4.2.3 エディタによる発展関係の記述

発展関係の記述は, Java で記述した GUI アプリケーションの上で行う. その記述を行う際に, 第 2 章の形式的定義に合わない関係を排除する必要がある. 特に発展関係を記述する際に注意すべきことは, 発展関係にあるメソッドやインスタンス変数とそのデータドメインの関係である. 例えば, あるインスタンス変数 id があるインスタンス変数 $id^{\#}$ より発展していることを記述する場合, id のデータドメインは $id^{\#}$ のデータドメインより発展していなければならない. 本エディタでは, 発展関係の不整合をなくすため, id のデータドメインの指定は $id^{\#}$ のデータドメインから発展したデータドメインのリストから選択させる. これは, メソッドの戻り値や引数の指定でも同様である. したがって, 発展後のフィールドやメソッドを記述する場合, 先に発展元のインスタンス変数やメソッドのデータドメインを発展させたデータドメインを定義する必要がある.

また, クラスの発展関係の記述は, インスタンス変数やメソッドの発展関係の記述によって行われる. あるクラスを発展させたクラスを記述する場合は, 上記で示した順序で, もとのクラスのインスタンス変数と発展関係にあるインスタンス変数と, もとのクラスのメソッド集合と発展関係にあるメソッド集合を定義する必要がある.

発展関係の編集例を図 4.3 に示す.

図 4.3: インスタンス変数の編集例

図 4.3 では,`Rectangle_v5` のインスタンス変数 `x` の編集を行っている. `x` のデータドメインの選択肢として, `Rectangle_v4` のインスタンス変数 `x` のデータドメイン `HorzDom` より発展したデータドメインだけがコンボボックスに表示されているのがわかる.

4.2.4 自動生成されるコード

発展的プロトタイピング技法では, 各発展段階ごとに実際に動くコードを生成するため通常のシステム開発と比べてかなり多くのソースコードを記述しなければならない. しかしながら抽象実行に必要となる XML 文書には多くのプログラムに関する情報が埋め込まれている. そこで, XML 文書で記述された情報をもとにコードの自動生成することで, 開発の省力化を試みる.

自動生成されるものは

- Proxy オブジェクト生成のための Java インタフェース
- 発展的オブジェクトのクラスの雛形

- データドメインクラス

である.

Proxy オブジェクト生成のための Java インタフェース

Proxy オブジェクト生成のための Java インタフェースは, 各クラスドメインに対して一つ生成される. このインタフェースでは, そのクラスドメインのクラス群で定義された全てのメソッドシグネチャを定義する.

例えば, 長方形オブジェクトの例では `Rectangle.java` をファイル名として, 以下のようなソースコードが生成される.

```
interface Rectangle{
    op(OpInDom v0);
    setSize(DemDom v0);
    move(PointDom v0);
    setX(AbstHorzDom v0);
    setY(AbstVertDom v0);
    setX(HorzDom v0);
    setY(VertDom v0);
    setX(int v0);
    setY(int v0);
}
```

発展的オブジェクトのクラスの雛形

発展的オブジェクトのクラスの雛形として, フィールド定義とメソッドのシグネチャが生成される. この雛形は開発者メソッドボディを記述することで, 完全なクラス定義となる.

例えば, クラス `Rectangle_v5` の雛形は `Rectangle_v5.java` をファイル名として以下のようなソースコードが生成される.

```
class Rectangle_v5 {
    int x;
    int y;
    DemDom size;
    setSize(DemDom v0) {}
    void setX(int v0) {}
    void setY(int v0) {}
}
```

データドメインクラス

一つのデータドメインに対して一つのクラスが生成される. このデータドメインクラスはそのデータドメインの具体集合の値として, インスタンス化される. 生成されるデータドメインクラス

のコンストラクタは, ある値を文字列表現したものを受け取るように定義される. データドメイン上の演算は, 必要に応じて開発者が定義しなければならない.

例えば, データドメイン OpInDom では, OpInDom.java をファイル名として, 以下のようなソースコードが生成される.

```
class OpInDom {
    String value;
    OpInDom (String v) {
        value = v;
    }
    boolean equals(Object obj) {
        boolean rtv = false;
        if (obj instanceof OpInDom &&
            value.equals(((OpInDom)obj).value)) {
            rtv = true;
        }
        return rtv;
    }
}
```

上記で定義されたメソッド equals() は, 抽象実行時に値を抽象化するために用いられるものである.

4.3 抽象実行ビジュアライザ

4.3.1 ビジュアライザ構築の目的

本研究で提案するビジュアライザは以下の 2 点を目的とする.

1. 抽象実行の振る舞いを視覚的に表現する
2. プログラムのドキュメントの代りとなる

抽象実行はあるプログラムの開発者が, そのプログラムから使用する別のプログラムの発展状況を意識することなく, プログラムの開発や実行を可能とする. しかしながら, 実際にテストをする際に, どこで抽象実行が発生したのか, どのプログラムに抽象化されて実行されたかを知ることは困難である. そのため, その抽象実行の振る舞いを視覚化することで, 開発者のプログラムのテスト実行の補助を行う.

また, 発展的プロトタイピング技法では, 仕様が明らかになっていないプログラムを, 段階的に仕様を決定しながら構築していく. よってプログラムを構築する前に明文化されたドキュメントが存在しない. このため, 開発者間や開発者と顧客との間でで仕様について議論する場合にはソースコードをベースにして議論しなければならない. ソースコードは, 同じ仕様を満たすものであっても個人によって実現方法が異なるため, コミュニケーションの道具としては適切ではない. そこで, プログラムの実行情報から, 開発者間や開発者と顧客のコミュニケーションの道具となりうるドキュメントの生成を試みる.

4.3.2 抽象実行ビジュアライザの実現のための諸技術

抽象実行ビジュアライザは, プログラムが実行されている Java バーチャルマシンに対して JDI(Java Debugger Interface) を用いて接続し, メソッドの呼び出しの情報などを取得する. そして UML のシーケンス図をもとにした図によって抽象実行の振る舞いを表現する. 以下にこれらの諸技術について述べる.

シーケンス図

シーケンス図は UML(Unifed Modeling Language) [10] で定義されているダイアグラムの一つである. UML はオブジェクト指向分析・設計で用いられる仕様記述言語であり, 複数のオブジェクト指向開発プロセスで用いられていたダイアグラムを統一する目的て開発された. UML は, 現在, オブジェクト指向開発における仕様記述言語のデファクトスタンダードとなっている.

UML ではオブジェクト間の相互作用を記述するダイアグラムとして, シーケンス図とコラボレーション図の 2 種類が用意されている. どちらもオブジェクト間のメッセージのやりとりを表現するものであるが, 前者はメッセージの時間的順序に注目しており, 後者はオブジェクトの構造的な配置の表現に注目している.

本研究ではシーケンス図が,

1. 多くの開発者や顧客にとって馴染みぶかい図であり,
2. 抽象度の時間的な変化を表現しやすい

ことから, 我々は抽象実行の振る舞いの表現に適している考えている. 抽象度の変化を時間的に捉えることで, システムの実行がどこまで期待通りに進行し, どこでシステムの抽象化が発生しているかを把握しやすくなると考えている.

Java Debugger Interface

Java Debuger Interface(JDI) は Java Platform Debugger Architecture (JPDA) を実現するインタフェースの一つである. [11] JPDA は Java プログラミング言語のデバッグツールを, ハードウェア, オペレーティングシステム, 仮想マシンの実装などプラットフォーム固有の詳細事項に注意を払わずに簡単に作成できるように, 標準的なインタフェースを提供している.

本研究では JDI を用いることで, メソッド呼出しをログに記録するといったような特殊な機能をユーザーのコード内に埋め込むことなく, 実行時の振る舞いを視覚化する.

JDI では, プログラム実行のユーザコードレベルの情報を, イベントオブジェクトから取得することが可能である. 例えば, メソッドの呼出時には `com.sun.jdi.event.MethodEntryEvent` が生成される. このイベントオブジェクトからメソッドの情報をもつオブジェクト `com.sun.jdi.Method` や呼出スタックの状態を表すオブジェクト `com.sun.jdi.StackFrame` を取得することができる. 抽象実行ビジュアライザでは, 抽象実行時に発生する様々なイベントを JDI を用いて捕捉し, そのイベントオブジェクトから必要な情報を取り出すことで, その実行時の情報の視覚化を実現している.

4.3.3 抽象実行時の振舞の視覚化

抽象実行ビジュアライザが生成したシーケンス図を図 4.4 に示す.

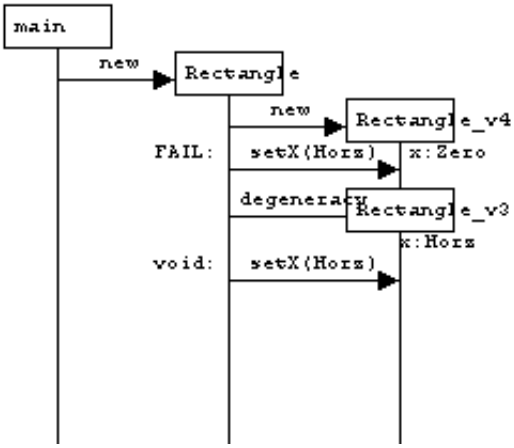

図 4.4: 抽象実行ビジュアライザ

図 4.4 は, `Rectangle_v4` のインスタンスに対して抽象値 $Horz$ を入力として `setX()` を呼び出したことで, 抽象実行が発生した例である.

まず `main` メソッドからプロキシオブジェクト `Retangle` が生成され, そのプロキシオブジェクトから `Rectangle_v4` のインスタンスが生成されている. `Rectangle_v4` に対するメソッド呼出しは `Retangle` を経由して行われている. `Retangle` は, `Rectangle_v4` で呼出しが失敗 (FAIL) したことを検出すると, `Rectangle_v4` を `Rectangle_v3` に縮退 (degeneracy) する. この縮退によって, フィールド `x` の値が抽象値 $Zero$ から抽象値 $Horz$ に抽象化されているのがわかる. そして, `Retangle` は, 再度 `setX()` を呼び出しを行っている.

第5章　発展的プロトタイピングと開発環境を用いた構成例

本章では, 構築した開発環境の上で, 実際に発展的プロトタイピングを用いた開発を行う. 例題としてトランプゲームのブラックジャックと在庫管理システムをあげた.

前者の例では, メソッドの詳細化のみに注目し, 開発の途中段階でビジュアライザによって抽象実行を行うことの有効性を示す.

後者の例ではメソッドの詳細化, 直積分割そして直和分割を用いた構成例を示す. また, 抽象領域上の演算を定義する際に問題が発生することを示す.

5.1　ブラックジャック

ブラックジャックのルールを以下に簡単にルールを説明する. システムを単純化するために実際とは多少異なるルールを採用している.

ルール

- 手持ちのカードの数字の合計が「21」を超えない範囲で「21」に近くなることを競うゲームである
- プレイヤーはゲーム開始時にコインを賭ける. 勝てば 2 倍, 引き分けならばそのまま, 負けならば没収となる.
- カードの合計の計算方法は 2 から 9 のカードはそのまま,10,J,Q,K は Ten と呼び 10 と数える,A は 1 または 11 と数える
- 実際のゲームではプレイヤーは「21」を超えない限りカードを何枚でも引くことが可能であるが, 本システムでは常に 2 枚だけをカードスタックから引くことにする
- ディーラーは手札の合計が 17 以上になるまでカードを引き続けなければならない. したがって, ディーラーの手札は 17,18,19,20,21 または 22 以上のいずれかとなる.
- プレイヤーの手札が A と 10,J,Q,K のいずれかとの組である場合は BLACKJACK といい, ディーラの手札に関係なくプレイヤーの勝ちとする.
- プレイヤーの手札が 22 を超えることはないので, ディーラーが 22 以上だった場合は Burst といい, プレイヤーの勝ちとなる

抽象化段階

本システムを構成するオブジェクトはプレイヤー, カードスタック, ディーラーである. 概要図を図 5.1 に示す.

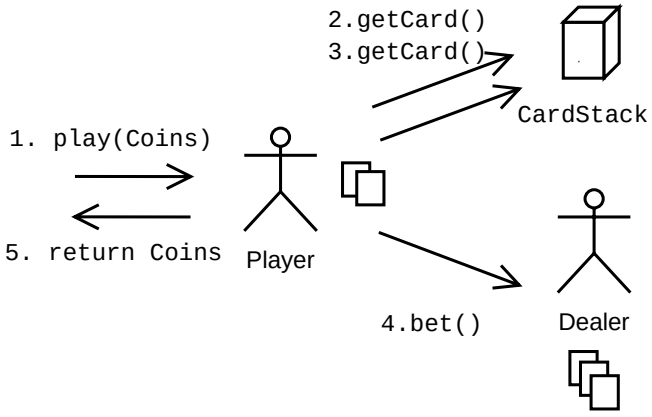

図 5.1: ブラックジャック概要図

ゲームを開始するとプレイヤーオブジェクトは以下のように振る舞う.

(1) 掛け金であるコインを入力としてゲームを開始する

(2) (3) カードスタックに対してカードを要求し, カードを 2 枚受け取る

(4) カードの値の合計を入力として, ディーラと勝負する. ディーラーは自分の手札と比較し, 勝敗または引き分けの結果を返す

(5) 掛け金と結果に応じて, コインを出力する

上記の振る舞いに現れるメソッドの入出力の値とオブジェクトが状態として持つ値を整理すると以下のようになる.

- コイン
- カード
- カードの値の合計, ディーラの手札の合計
- 賭けの結果

それぞれの最も抽象的なオブジェクトを定義するために上記の各値を一つ抽象値としてまとめ, データドメインを以下のように定義する.

$$
\begin{aligned}
D_1^{Coins} &= (\{Coins\}, \emptyset) \\
D_1^{Card} &= (\{Card\}, \emptyset) \\
D_1^{Cards} &= (\{Cards\}, \emptyset) \\
D_1^{Result} &= (\{Result\}, \emptyset)
\end{aligned}
$$

以上から, 抽象的なクラス `Player_v1`, `CardStack_v1`, `Dealer_v1` を以下のように定義する.

```
class Player_v1 {
    CardStack stack_v1;
    Dealer dealer_v1;
    Player_v1 () {
        dealer_v1 = (Dealer)ProxyFactory.createProxy("Dealer");
        stack_v1  = (CardStack)ProxyFactory.createProxy("CardStack");
    }
    CoinDom_v1 play_v1(CoinDom_v1 v0) {
        CardDom_v1 c1   = stack_v1.getCard_v1();
        CardDom_v1 c2   = stack_v1.getCard_v1();
        ResultDom_v1 rv = dealer_v1.bet_v1(new CardsDom_v1(c1, c2));
        return new CoinDom_v1("Coins");
    }
}

class CardStack_v1 {
    CardDom_v1 getCard_v1() {
        return new CardDom_v1("Card");
    }
}

class Dealer_v1 {
    CardsDom_v1 cards_v1;
    Dealer_v1 () {
        cards_v1 = new CardsDom_v1("Cards");
    }
    ResultDom_v1 bet_v1(CardsDom_v1 p) {
        return new ResultDom_v1("Result");
    }
}
```

Player_v1 のコンストラクタでは, Dealer_v1 や CardStack_v1 を直接生成せずに, ProxyFactory のクラスメソッド createProxy() にクラスドメイン名を入力として呼び出すことでプロキシオブジェクトを動的に生成し, 間接的に参照を得ている. 従って他オブジェクトに対する全てのメソッド呼出しはプロキシ経由で行われることになる.

Player_v1 のメソッド play() に抽象値 $Coins$ を入力として呼び出したときに, ビジュアライザが生成した図を 5.2 に示す.

まず, play_v1() は, getCard_v1 を 2 回呼び出し, 抽象値 $Card$ を 2 回得る. 次に 2 つの $Card$ から抽象値 $Cards$ を生成し, それを入力として bet_v1 を呼出し, 抽象値 $Result$ を得る. 最後に, play_v1() の結果として抽象値 $Coins$ を返す.

発展段階 1

次に各データドメインを一つ具体化したデータドメインを定義し, メソッドおよびオブジェクトの状態を発展させる. この段階では, 賭けの結果がはっきりするようにデータドメインを発展させることにする.

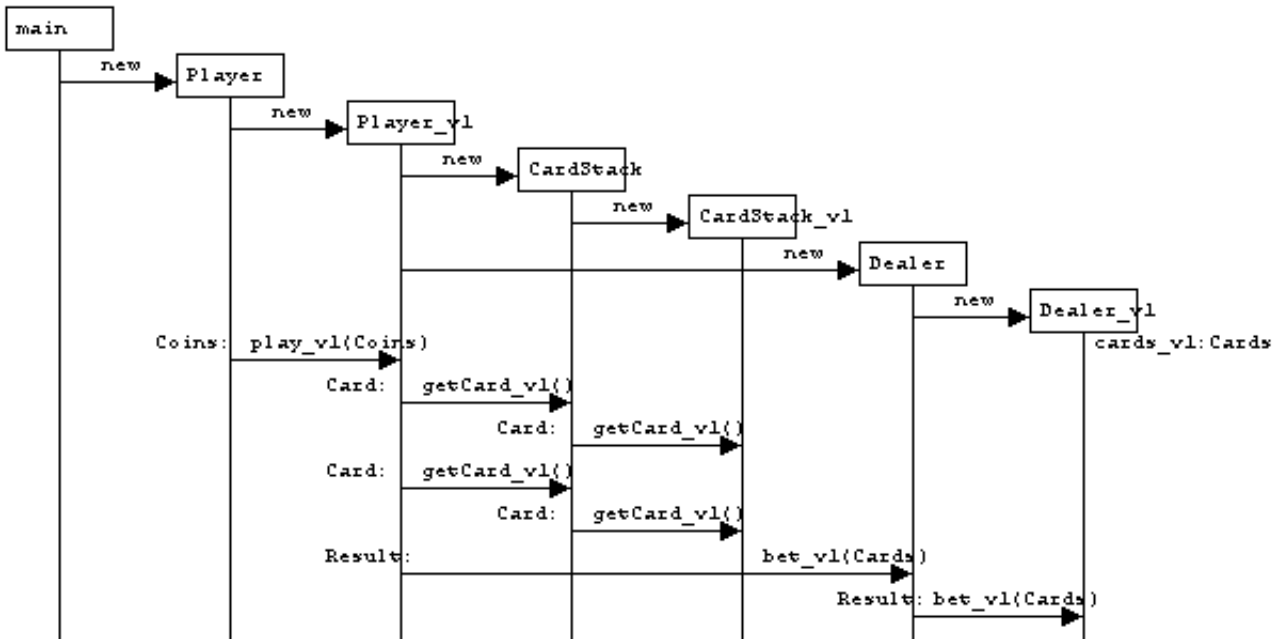

図 5.2: 抽象化段階のブラックジャックプログラムのシーケンス図

ルールによるとプレイヤーが A と Ten を引いた場合は BLACKJACK となり勝負がつくので抽象値 $Card$ をこの 2 つとそれ以外に一時的に具体化する. 次にカードの値の合計値 $Cards$ について考える. ディーラの持ち札は 17 から 21 か 22 以上である. そこで, カードの合計値を 17 未満の $Short$ と 22 以上の $Burst$ とそれ以外に一時的に具体化する. ディーラーが $Burst$ でプレイヤーのカードの合計が $Short$ の場合はユーザーの勝ちが決まるので, $Card$ 2 枚の合計が安全に $Short$ になるカードを考える. 2 から 5 のカードはもう一枚のカードがたとえ A であっても合計が $Short$ になるので 2 から 5 のカードを抽象値 $Single$ に具体化する. 最後にここまでで抽象化されなかった値は実際のカードや値に具体化する.

発展後のデータドメインは $D_2^{CardDom}$ と $D_2^{CardsDom}$ は以下のようになる.

$$
\begin{aligned}
D_2^{CardDom} &= (data(D_1^{CardDom}) \cup \{An, Single, 6, 7, 8, 9, Ten\}, rel(D_1^{CardDom}) \cup \\
&\quad \{(Card, A), (Card, Single), (Card, 6), \\
&\quad (Card, 7), (Card, 8), (Card, 9), (Card, Ten)\} \\
D_2^{CardsDom} &= (data(D_1^{CardsDom}) \cup \{Short, 17, 18, 19, 20, 21, Burst\}, \\
&\quad rel(D_1^{CardsDom}) \cup \{(Cards, Short), (Cards, 17), (Cards, 18), (Cards, 19), \\
&\quad (Cards, 20), (Cards, 21), (Cards, Burst)\})
\end{aligned}
$$

上記のように具体化することによって, 勝ち負けまたは引き分けを決定することが可能となり, 賞金も計算することができる. したがって $Coins$ を Java の 1 以上の組み込み整数に, $Result$ を勝ちを表す Win, 引き分けを表す $Even$ そして敗けを表す $Lose$ にそれぞれ具体化する.

発展後のデータドメイン $D_2^{CoinsDom}$ と $D_2^{ResultDom}$ は以下のようになる.

$$
\begin{aligned}
D_2^{CoinsDom} &= (data(D_1^{CoinsDom}) \cup \{1, 2, 3, ...\}, n \\
&\quad rel(D_1^{CoinsDom})\{(Coins, 1), (Coins, 2), ...\}) \\
D_2^{ResultDom} &= (data(D_1^{ResultDom}) \cup \{Win, Even, Lose\}, \\
&\quad rel(D_1^{ResultDom})\{(Result, Win), (Result, Even), (Result, Lose)\})
\end{aligned}
$$

以上から, Player_v1, CardStack_v1, Dealer_v1 をそれぞれ発展させたクラス Player_v2, CardStack_v2, Dealer_v2 を以下に示す.

```
class Player_v2 {
    CardStack stack_v2;
    Dealer dealer_v2;
    int play_v2(int v0) {
        CardDom_v2 c1   = stack_v2.getCard_v2();
        CardDom_v2 c2   = stack_v2.getCard_v2();
        ResultDom_v2 rv = dealer_v2.bet_v2(new CardsDom_v2(c1, c2));
        if (rv.toString().equals("Win")) {
            return v0 * 2;
        } else if (rv.toString().equals("Even")) {
            return v0;
        } else { // Lose
            return 0;
        }
    }
}

class CardStack_v2 {
    CardDom_v2 getCard_v2() {
        int i = 1 から 13 までの数を生成
        String s = null;
        switch (i) {
        case 1:
            s = "A";
    break;
        case 2:
        case 3:
        case 4:
        case 5:
            s = "Single";
    break;
        case 10:
        case 11:
        case 12:
        case 13:
    s = "Ten";
            break;
        default:
```

```
            s = Integer.toString(i);
        }
        return new CardDom_v2(s);
    }
}

class Dealer_v2 {
    CardsDom_v2 cards_v2;
    Dealer_v2 () {
        int i = 17 から 26 までの数を生成
        String s = null;
        if (i>21) {
            s = "Burst";
        } else {
            s = Integer.toString(i);
        }
        cards_v2 = new CardsDom_v2(s);
    }
    ResultDom_v2 bet_v2(CardsDom_v2 p) {
        if ( cards_v2.equals(new CardDom_v2("Burst")) ||
            p.equals(new CardDom_v2("21"))           ||
            p.compareTo(d) > 0) {
            return new ResultDom_v2("Win");
        } else if (p.compareTo(d) == 0) {
            return new ResultDom_v2("Even");
        } else {
            return new ResultDom_v2("Lose");
        }
    }
}
```

Player_v2 のメソッド play_v2() に整数 10 を入力として呼び出したときに, ビジュアライザが生成した図を 5.3 に示す.

まず, play_v2() は, 1 回目の getCard_v2() の呼出しで A を得る. 次に 2 回目の getCard_v2() の呼出しで 2 から 5 のカードを抽象化する $Single$ を得る. A と $Single$ の合計値は $Short$ となるので, それを入力として bet_v2() を呼び出す. bet_v2() は, ディーラーの持ち札の値 $Burst$ とプレイヤーの値 $Short$ を比較する. 結果は, プレイヤーの勝ちなので Win を返し, 最終的に play_v2() は, 入力値 10 を 2 倍した値 20 を出力する.

発展段階 2

最後にデータドメイン D_2^{Card} の値を実際のトランプのカードに具体化する.

したがって発展したデータドメイン D_3^{Card} と D_3^{Cards} は以下のようになる.

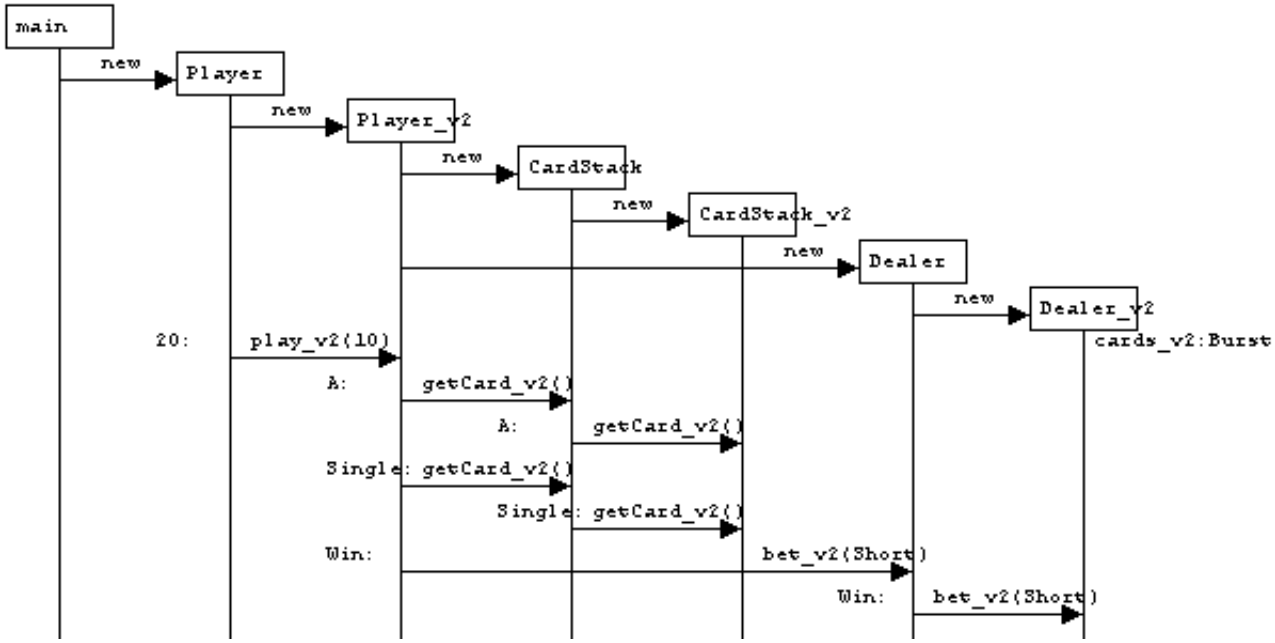

図 5.3: 発展段階 1 のブラックジャックプログラムのシーケンス図

$$
\begin{aligned}
D_3^{Card} &= (data(D_2^{Card}) \cup \{A, 2, 3, 4, 5, 6, 10, J, Q, K\}, rel(D_2^{Card}) \cup \{(Single, 2), (Single, 3), \\
&\quad (Single, 4), (Single, 5), (Ten, 10), (Ten, J), (Ten, Q), (Ten, K)\} \\
D_3^{Cards} &= (data(D_2^{Cards}) \cup \{1, ..., 16, 22, 23, 24, 25, 26\}, \\
&\quad rel(D_2^{Cards}) \cup \{(Short, 1), ..., (Short, 16), (Burst, 22), (Burst, 23), (Burst, 24), \\
&\quad (Burst, 25), (Burst, 26)\})
\end{aligned}
$$

最終的に構築したデータドメインを図を 5.4 に示す.

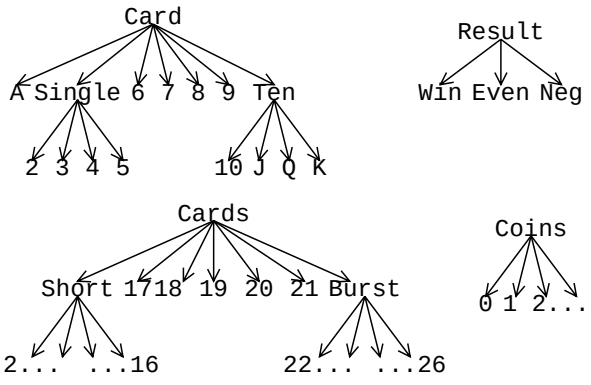

図 5.4: ブラックジャックの最終データドメイン

以上から, `Player_v2`, `CardStack_v2`, `Dealer_v2` をそれぞれ発展させたクラス

Player_v3, CardStack_v3, Dealer_v3 を以下に示す.

```
class Dealer_v3 {
    CardsDom_v3 cards_v3;
    Dealer_v3 () {
        int i = 17 から 26 までの数を生成
        String s = Integer.toString(i);
        cards_v3 = new CardsDom_v3(s);
    }
    ResultDom_v2 bet_v3(CardsDom_v3 p) {
        if (cards_v3.compareTo(new CardsDom_v3("21")) > 0 ||
            p.equals(new CardDom_v2("21"))                ||
            p.compareTo(d) > 0) {
            return new ResultDom_v2("Win");
        } else if (p.compareTo(new CardsDom_v2("21")) == 0) {
            return new ResultDom_v2("Even");
        } else {
            return new ResultDom_v2("Lose");
        }
    }
}

class CardStack_v3 {
    CardDom_v3 getCard_v3() {
        int i = 1 から 13 の値を生成
        String s = null;
        switch (i) {
        case 1:
            return new CardDom_v3("A");
        case 11:
            return new CardDom_v3("J");
        case 12:
            return new CardDom_v3("Q");
        case 13:
            return new CardDom_v3("K");
        default:
            return new CardDom_v3(i);
        }
    }
}
```

Player_v3 のメソッド play_v3() に整数 10 を入力として呼び出したときに, ビジュアライザが生成した図を 5.5 に示す.

まず, play_v3() は, 1 回目の getCard_3() の呼出しで A を得る. 次に 2 回目の getCard_v3() の呼出しで 5 を得る. A と 5 の合計値は 16 となるので, それを入力として bet_v3() を呼び出す. bet_v3() は, ディーラーの持ち札の値 22 とプレイヤーの値 16 を比較する. 結果は, プレイヤーの勝ちなので Win を返し, 最終的に play_v3() は, 入力値 10 を 2 倍した値 20 を出力する.

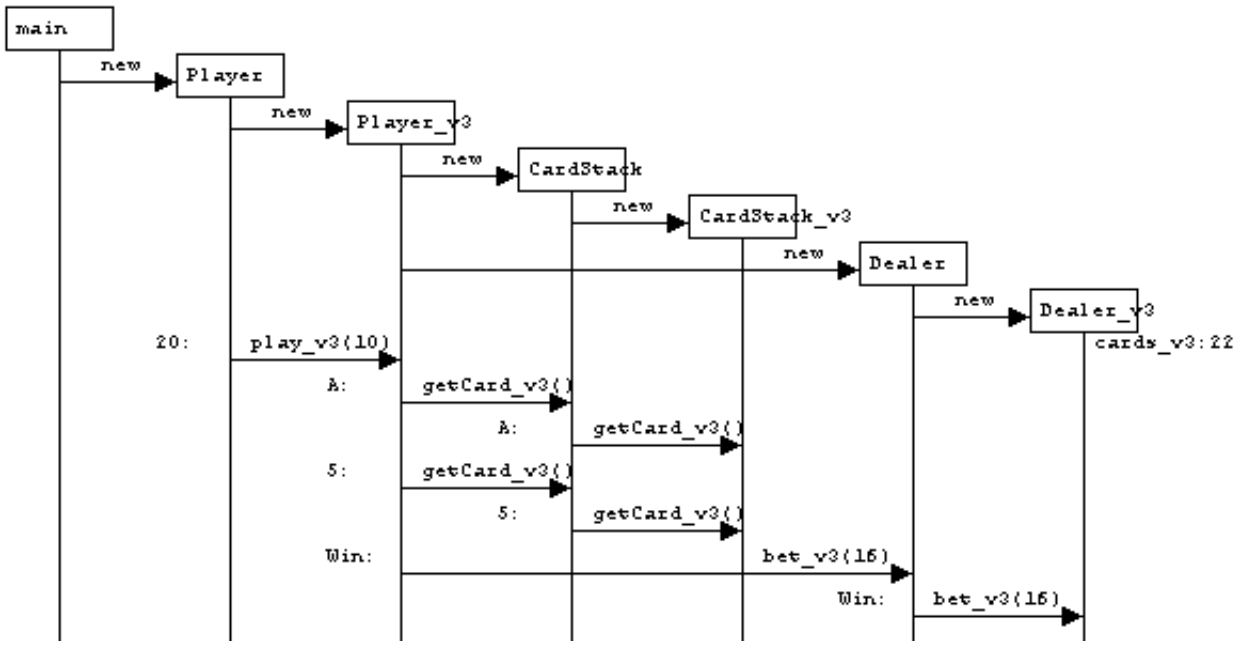

図 5.5: 発展段階 2 のブラックジャックプログラムのシーケンス図

抽象実行例

発展的プロトタイピング技法においては, 個々のオブジェクトの発展段階が異なっていても抽象解釈を行うことによって, システムを中断させることなく最後まで実行することが可能である.

このブラックジャックのプログラムにおける抽象実行例について説明する. 開発状況として, Player_v3, CardStack_v3 の開発は完了しているが, Dealer_v3 のメソッド bet_v3() はまだ実装されてないと仮定する. この場合 bet_v3() の代りに bet_v2() を実行することで抽象実行が可能となる. ビジュアライザが生成した実行の様子を図 5.6 に示す.

まず, play_v3() は, 1 回目の getCard_3() の呼出しで A を得る. 次に 2 回目の getCard_v2() の呼出しで 5 を得る. A と 5 の合計値は 16 となるので, それを入力として bet_v3() を呼び出す. ここまでは, 発展段階 2 で示した例と同じである. bet_v3() は, まだ実装されていないので, 呼出しが失敗 (FAIL) する. この失敗を検出した Dealer プロキシは, 以下の手順にて抽象実行を行う.

1. bet_v3() の抽象的なメソッド bet_v2() を取得する
2. bet_v3() の引数 16 を $Short$ に抽象化する
3. Dealer_v3 に対して, bet_v2() を呼び出し, Dealer_v3 に bet_v2() が実装されていないことを検出する
4. Dealer_v3 の抽象的なオブジェクト Dealer_v2 を生成する.
5. Dealer_v3 のフィールド値 22 を抽象化した $Burst$ を Dealer_v2 にセットする.
6. Dealer_v2 に対して bet_v2() を呼出し, 戻り値 Win を得る

この結果, play_v3() は, 入力値 10 を 2 倍した値 20 を出力する.

このように, Player_v3 は Dealer の発展段階を気にすることなく play_v3() のテストを行うことが可能となる.

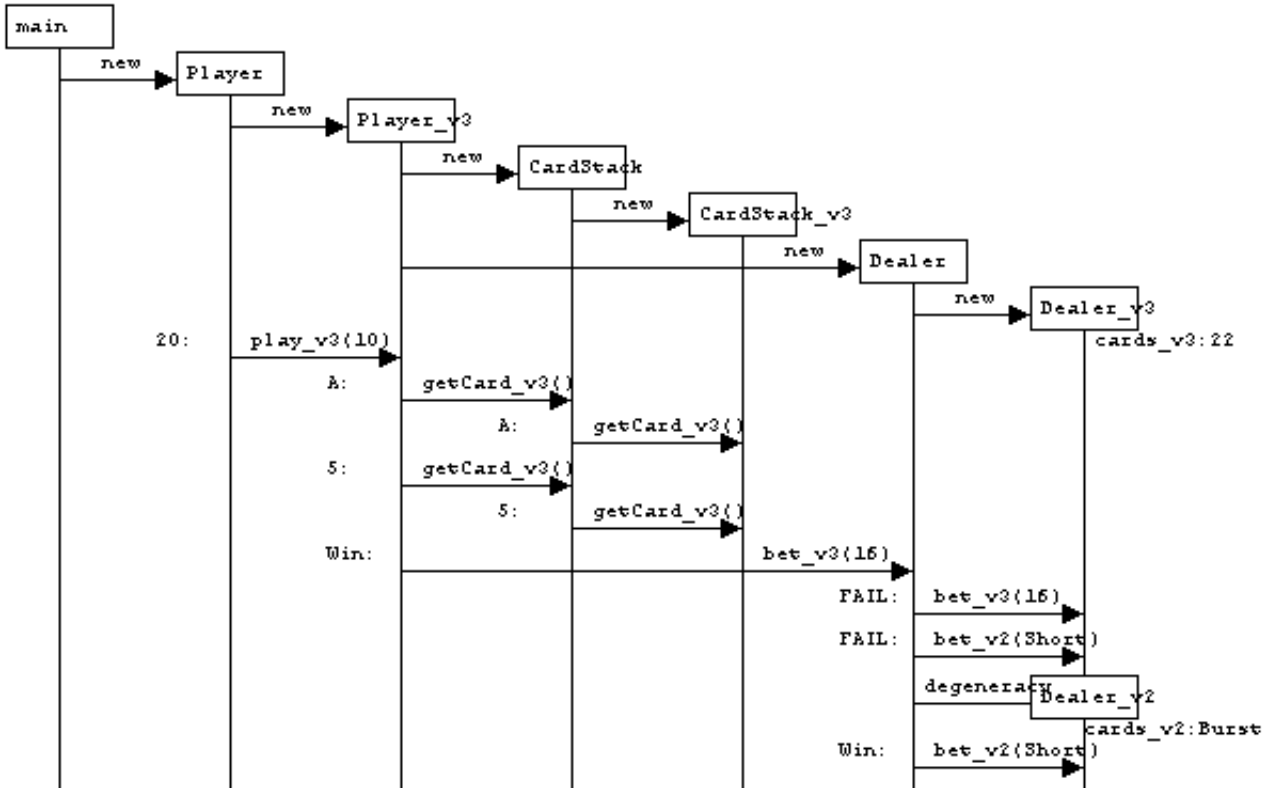

図 5.6: ブラックジャックプログラムの抽象実行のシーケンス図

5.2 在庫管理システム

もう一つの例題として, ある商店の在庫管理システムを取り上げる. 前節の例題ではメソッドの詳細化しか扱わなかったが, この節ではメソッドの直積分割, 直和分割も用いる. この例を通して抽象値の具体化の困難さを示す.

仕様例 在庫管理システム

- 顧客は店に対して商品の発注処理を行う
- 店は受注数量と在庫数量を確認し引き当てを行う.
- 店は完全に引き当てが可能であったかを顧客に通知する.
- 顧客は要求した分の在庫があれば, 購入処理を行い, 商品を得る.
- 部分的な引き当ての場合は引き当て分をキャンセルする.

本システムを構成するオブジェクトは顧客と店である. 概要図を図 5.7 に示す.

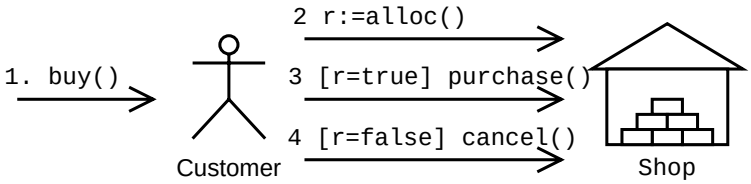

図 5.7: 在庫管理システム概要図

抽象化段階

上記の仕様に現れる, メソッドの入出力になる値とオブジェクトが状態として持つべき値を整理すると以下のようになる.

- 受注数
- 在庫数
- 引き当て結果
- 商品

それぞれの最も抽象的なオブジェクトを定義するために上記の各値を一つ抽象値としてまとめる.

引き当て結果と商品は一連の処理を行う際に出力として得られるものである. まず最も抽象的なメソッドをつ定義するために, 引き当て結果と商品は $Result$ という 1 つの抽象値にまとめる. また, 受注数を抽象値 $Amount$ に, 在庫数量を抽象値 $Invent$ にそれぞれ抽象化する.

そこでデータドメインを以下のように定義する.

$$
\begin{aligned}
D_1^{Amount} &= (\{Amount\}, \emptyset) \\
D_1^{Invent} &= (\{Invent\}, \emptyset) \\
D^{Result} &= (\{Result\}, \emptyset)
\end{aligned}
$$

上記のデータドメインを用いて, 抽象的なクラス Shop_v1 と Customer_v1 を以下のように定義する.

```
class Shop_v1 {
    InventDom_v1 invent = new InventDom_v1("Invent");
    ResultDom order(AmountDom v0) {
        return new ResultDom("Result");
    }
}

class Customer_v1 {
    Shop link_v1;
    Customer_v1 () {
        link_v1 = (Shop)ProxyFactory.createProxy("Shop");
    }
    void buy() {
        link_v1.order(new AmountDom("Amount"));
    }
}
```

Custmer_v1 のメソッド buy() を呼び出したときに, ビジュアライザが生成した図を 5.8 に示す.

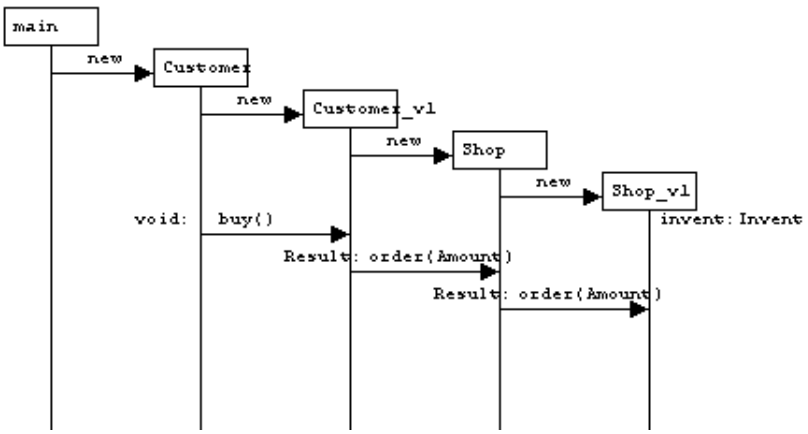

図 5.8: 抽象化段階の在庫管理システムのシーケンス図

buy() は, 抽象値 $Amount$ を入力として order() を呼び出し, 抽象値 $Result$ を得る.

発展段階 1

次に, 発注処理の出力であるデータドメイン D^{Result} を, 引き当て処理の出力となるデータドメイン D^{Alloc} とそれ以外の処理の出力のデータドメイン D^{Tran} の組に分割する.

その結果, データドメインは,

$$\begin{aligned} D^{Alloc} &= (data(D^{Result}) \cup Result, (Result, Alloc)) \\ D^{Tran} &= (data(D^{Result}) \cup Result, (Result, Tran)) \end{aligned}$$

となり, それに従って Shop_v1 のメソッド order() を alloc() と tran() に直積分割する. 従って,Shop_v1 の発展クラス Shop_v2 と Customer_v1 の発展クラス Customer_v2 を以下のように定義する.

```
class Shop_v2 {
    InventDom_v1 invent = new InventDom_v1("Invent");
    AllocDom alloc(AmountDom v0) {
        return new AllocDom("Alloc");
    }
    TranDom tran(AmountDom v0) {
        return new TranDom("Tran");
    }
}

class Customer_v2 {
    Shop link_v2;
    Customer_v2 () {
        link_v2 = (Shop)ProxyFactory.createProxy("Shop");
    }
    void buy() {
        link_v2.alloc(new AmountDom("Amount"));
        link_v2.tran(new AmountDom("Amount"));
    }
}
```

Customer_v1 の buy() では, order() 呼出しのみであったが, Customer_v2 の buy() では, alloc() と tran() 呼出しの列に発展している.

Customer_v2 のメソッド buy() を呼び出したときに, ビジュアライザが生成した図を 5.9 に示す.

buy() は, 抽象値 $Amount$ を入力として alloc() を呼び出し, 在庫引き当ての結果の抽象値 $Alloc$ を得る. その後に抽象値 $Amount$ を入力として tran() を呼び出し, 抽象値 $Tran$ を得る.

発展段階 2

次に, 取引処理 tran() を購入処理 purchase() と取消処理 cancel() に直和分割する. 抽象値 $Tran$ を, 購入処理の出力として商品を表す $Products$ と取消処理の出力としてなにも得られないことを表す $None$ に具体化する. それぞれの値は直和のドメイン $D^{Product}$, D^{Empty} で以下のように定義される.

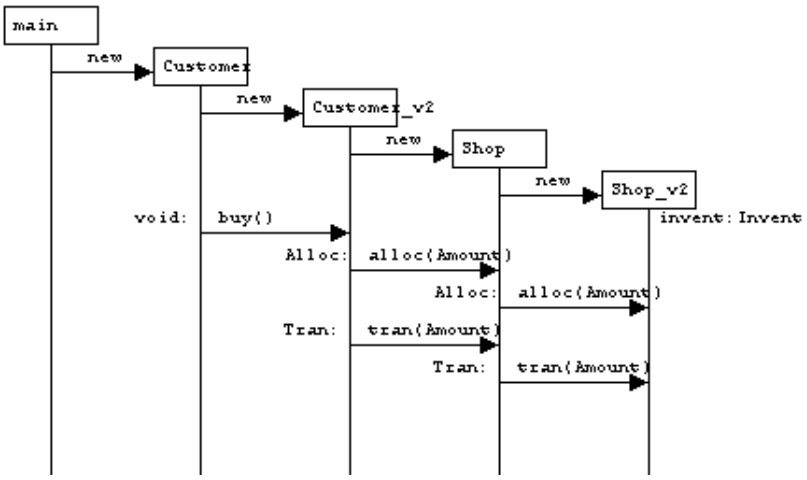

図 5.9: 発展段階 1 の在庫管理システムのシーケンス図

$$
\begin{aligned}
D^{Product} &= (data(D^{Tran}) \cup \{Products\}, \{(Tran, Products)\}) \\
D^{Empty} &= (data(D^{Tran}) \cup \{None\}, \{(Tran, None)\})
\end{aligned}
$$

また, 顧客が購入処理と取消処理のどちらを行うかは, 引き当て後の在庫がマイナスにならないか計算することによって決定される. そこで在庫を表す抽象値 $Invent$ を, 在庫がある状態を抽象化する $InStock$, 在庫が空の状態を抽象化する $OutStock$, 在庫がマイナスの状態を抽象化する $Minus$ に具体化する. これによってデータドメイン $D_2^{InventDom}$ は以下のように定義される

$$
\begin{aligned}
D_2^{Invent} = (&data(D_2^{Invent}) \cup \{InStock, OutStock, Normal\}, \{(Invent, InStock), \\
&(Invent, OutStock), (Invent, Minus)\})
\end{aligned}
$$

以上から,Shop_v2 の発展クラス Shop_v3 と Customer_v2 の発展クラス Customer_v3 を以下のように定義する.

```
class Shop_v3 {
    InventDom_v2 invent = new InventDom_v2("InStock");
    boolean alloc(AmountDom v0) {
        return invent_v2.sub(v0).ge(new InventDom_v2("Empty"));
    }
    ProductDom purches(TranAmountDom v0) {
        return new ProductDom("Products");
    }
    EmptyDom cancel(TranAmountDom v0) {
        return new EmptyDom("None");
    }
}

class Customer_v3 {
```

```
    Shop link_v3;
    Customer_v3 () {
        link_v3 = (Shop)ProxyFactory.createProxy("Shop");
    }
    void buy() {
        if(link_v3.alloc(new AmountDom("Amount")))
            link_v3.purchase(new AmountDom("Amount"));
        } else {
            link_v3.cancel(new AmountDom("Amount"));
        }
    }
}
```

buy() は, 抽象値 $Amount$ を入力として alloc() を呼び出し, 在庫引き当ての結果を Java の組み込みの boolean 値で得る. そしてその結果に応じて購入処理またはキャンセル処理のどちらかを行う.

しかし, この定義は実際に実行することはできない. なぜならば, alloc() メソッドでは InventDom_v2 上での演算を必要としているが, InventDom_v2 は抽象領域であるためその演算を決定的に定義できないからである.

問題となるのは以下の 2 行である.

```
        invent_v2 = invent_v2.sub(v0);
        return invent_v2.ge(new InventDom_v2("Empty"))
```

v0 には抽象値 $Amount$ が入る. 仮にフィールド invent_v2 が抽象値 $InStock$ を保持していたとする. $InStock$ から $Amount$ を引いた値が $InStock$ になるか $OutStock$ それとも $Minus$ になるかは単純に決定できない. これを決定するためには $InStock$ と $Amount$ の大小関係を定義する必要があるが, 抽象値のドメインをうまく決めなければ定義することは不可能である.

この問題に関する解決法は次章で議論することにし, ここではこの例題の段階的構築を先に進める.

発展段階 3

前段階で問題となった在庫数量と受注数量を Java の組み込み整数の int 型に具体化する.

$$
\begin{aligned}
D_2^{Amount} &= (data(D_1^{Amount}) \cup \{1,2,3,...\}, \{(Amount,1),(Amount,2),...\}) \\
D_3^{Invent} &= (data(D_2^{Invent}) \cup \{...,-1,0,1,...\}, \\
&\quad rel(D_2^{Invent}) \cup \{....,(Minus,-1),(OutStock,0),(InStock,1),...\})
\end{aligned}
$$

最終的に構築したデータドメインを図を 5.10 に示す.

以上から,Shop_v3 の発展クラス Shop_v4 と Customer_v3 の発展クラス Customer_v4 を以下のように定義する.

```
class Shop_v4 {
    int invent = 10;
    boolean alloc(int v0) {
        invent -= v0;
```

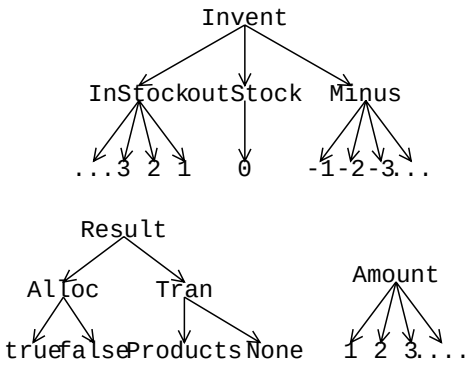

図 5.10: 在庫管理システムの最終データドメイン

```
        return invent >= 0;
    }
    ProductDom purchase(TranAmountDom v0) {
        return new ProductDom("Products");
    }
    EmptyDom cancel(TranAmountDom v0) {
        return new EmptyDom("None");
    }
}

class Customer_v4 {
    Shop link;
    Customer_v4 () {
        link = (Shop)ProxyFactory.createProxy("Shop");
    }
    void buy() {
        if (link_v4.allocation_v2(10))
            link.purchase(10);
        else
            link.cancel(10);
    }
}
```

Customer_v4 のメソッド buy() を呼び出したときに, ビジュアライザが生成した図を 5.11 に示す.
Shop_v4 の在庫数量は 10 なので引き当て可能となり true を返す. 引き当てが可能だったのでメソッド purchase() を呼出し, 製品 $Products$ を得る.

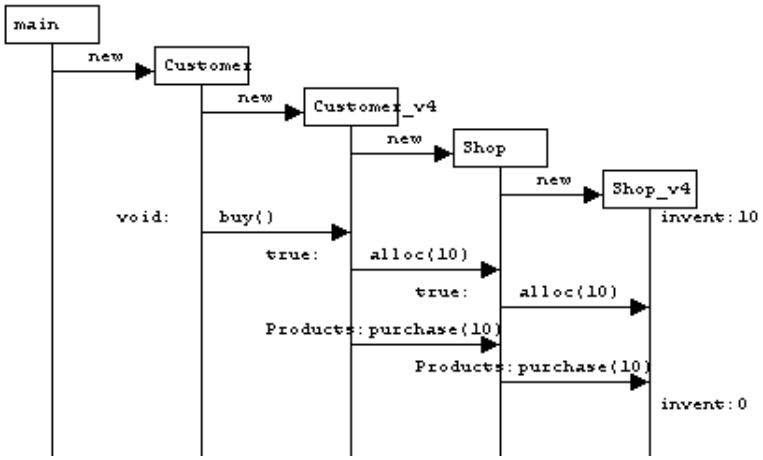

図 5.11: 発展段階 3 の在庫管理システムのシーケンス図

第6章　議論

6.1　処理系および開発環境に関する議論

6.1.1　スタブと比較した抽象実行のメリット

開発途中段階でプログラムのテストを行うためにしてスタブを用いる手法がある．ここでは, 抽象実行とスタブを作成してテスト実行する場合とどのような点で異なるかについて議論する．

スタブに対する抽象実行のメリットは以下の 2 点である．

- プログラマがテスト実行のためのコードを書く必要がない
- 抽象実行によって得られる値は, 必ず実際のコードと対応づけられている

スタブを用いた実行では, プログラマはそのテストのためだけにスタブを用意する. スタブは, まだ実装されていないメソッドの振る舞いをシミュレートするオブジェクトである．スタブは実際に作成される予定のメソッドと同じシグネチャを持ち, それが返す値は, 常に同じ値を返す場合もあれば, 違う値を返すこともある．また, 呼び出す側のエラー処理をチェックするために不正な値を含むこともある. スタブは, 実際に作成されるメソッドの出力値や, メソッド呼出しの副作用の結果として得られるオブジェクトのインスタンス変数の値の一部しかシミュレートしない. なぜならば, スタブがどの場合にも現実にあうものであれば, もはやそればスタブではなく実際のプログラムだからである．

抽象実行は異なる発展段階にあるオブジェクト間でのメソッド呼出しを可能にする. この機能によってプログラマはあるオブジェクトをテストする際に, そのオブジェクトが使用している他オブジェクトのメソッドが実装されるのを待たなくてもよい. 抽象実行ではプログラマはスタブを用意する必要はなく, 未実装のメソッドの代りにそのメソッドの抽象的なメソッドを使用することで実行を可能とする．例えば, 前章の BLACKJACK の抽象実行の例では, `Player_v3` のメソッド `play_v3()` は `Dealer_v3` のメソッド `bet_v3()` が実装されていない場合でもテストすることが可能であった. この場合, メソッド `bet_v3()` は メソッド `bet_v2()` に縮退され, 抽象実行される．抽象実行によって返される値やメソッド呼出しの副作用の結果としてのオブジェクトの状態値は, 実際のメソッドの呼出し結果として得られるものと必ず対応関係がつけられている. スタブをこのような対応関係を満すようにつくるにはコストがかかる．

特にメソッド `bet_v2()` の出力値はメソッド `bet_v3()` の出力値と全く同じものである. このようなことが可能となるのはメソッド `bet_v2()` を定義する際に, 出力値を決定するためにメソッドの入力値やオブジェクトの状態値をうまく抽象化しているからに他ならない．

メソッド `bet_v2()` では, メソッドの結果となる出力値の集合 {Win, Even, Lose} から出力値が一意に決まるように, メソッドの入力値集合やオブジェクトの状態値集合を抽象化している．

このように, 発展的プロトタイピングでは, まずメソッドの出力であるデータドメインを発展させることを考え, その出力値を決定するために抽象値を用いて入力値のデータドメインや状態値のデータドメインを発展させることで, 効果的な抽象実行が可能となる．

6.1.2 抽象実行によるテスト技法

本小節では, 抽象実行を用いて, あるメソッドを発展させたメソッドが正しく実装されているかテストする手法を提案する.

発展的プロトタイピング技法では, 発展関係にある 2 つのメソッドのそれぞれの呼出しにおいて, メソッドの入力値とメソッド実行前のシステムの状態が対応関係にある場合は, メソッドの出力値とメソッド実行後のシステムの状態も必ず対応関係がついている. この関係が形式的に定義づけられているため抽象実行が可能となる. ここでは, 抽象実行を用いて, 発展前後のメソッドの実行において対応関係を確認することで, 発展後のメソッドが発展前メソッドで与えられた抽象値上での仕様を満しているかテストすることが可能であることを示す.

テスト方法

あるメソッド $md^{\#}$ とそれを発展させた md があったときに, 以下の手順によって md が正しく実装されているかをテストすることが可能である.

1. md を実行する
2. md の入力値, 出力値, 実行前後のストアのスナップショットをとる
3. md の代りに $md^{\#}$ を抽象実行する
4. $md^{\#}$ の入力値, 出力値, 実行前後のストアのスナップショットをとる
5. md と $md^{\#}$ のスナップショットを比較する

スナップショットの比較において, 入力値, 実行前のストアに発展関係があるにも関わらず, 出力値や実行後のストアに発展関係が無い場合は, どちらかのメソッドに誤りがあることを発見できる. よって, 常に発展前の抽象的メソッドのテストを完全に行えば, 発展後のメソッドの誤りを発見することが可能となる.

但し, このテストによって発見できるエラーはあくまで抽象解釈の上での間違いであって, md が完全に仕様を満しているかをチェックするものではない.

実装

抽象実行によるテストは以下のクラスによって行われる

- `Store`
  システムに存在する発展的オブジェクトを保持するコレクションクラスである. 実行時に生成された発展的オブジェクトは `AbstractExecuter` によってこのオブジェクトに追加される. メソッドの呼出し前後のストアの状態をとるためにメソッド snapshot() を持つ. このメソッドは, Store に登録されている全ての発展的オブジェクトのメソッド clone() を呼出して, ストア全体のコピーを生成する.

- `Verifyer`
  `Verifyer` は以下の手順でストア集合の対応関係をチェックする

  - テスト対象となるメソッド実行列において, それぞれのメソッドの実行前後の Store のスナップショットを Stack に積み上げる.
  - 抽象実行のメソッド実行列において, それぞれのメソッドの実行前後の Store のスナップショットを Stack に積み上げる.

– それぞれスタックからストアのスナップショットを取り出して発展関係をチェックする

テスト例

再び BLACKJACK の構成例を用いて説明する. 以下はクラス Dealer_v3 のメソッド bet_v3() の誤った実装例である.

```
class Dealer_v3 {
    ResultDom_v2 bet_v3(CardsDom_v3 p) {
        if (p.equals(new CardDom_v2("21")) ||
            p.compareTo(d) > 0) {
            return new ResultDom_v2("Win");
        } else { ... }
    }
}
```

この例での仕様では

- プレイヤーの手札が 22 を超えることはないので, ディーラーが 22 以上だった場合は Burst といいプレイヤーの勝ちとなる.

となっているが, その条件が記述されていない.
この誤りを発見するためにテスト実行した結果のスタックの状態を図 6.1 に示す.

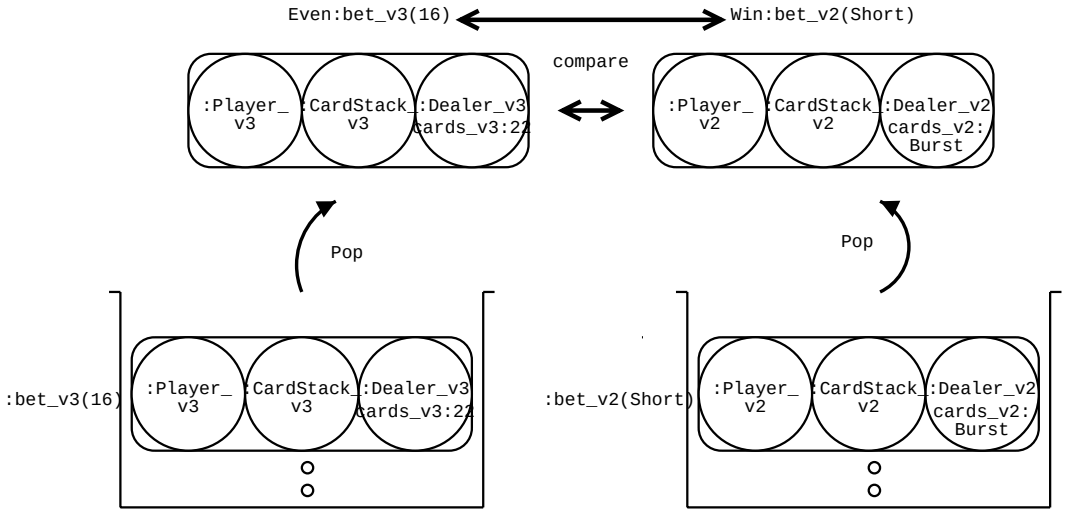

図 6.1: テスト中のスタック

前章の例と同じ状況を考える. bet_v3() は, ディーラーの持ち札の値 22 とプレイヤーの値 16 を比較する. 本来ならばディーラーの持ち札が 22 以上の場合プレイヤーの勝ちなので Win を返すはずが $Even$ を返している. 一方, 抽象実行した結果のスナップショットでは bet_v2 は, ディーラーの持ち札の値 $Burst$ とプレイヤーの値 $Short$ を比較し正しく Win を返している. この 2 つのメソッド実行後のスナップショットについてストアと戻り値がそれぞれ発展関係があるか調べる. この結果, 発展関係を満していないということが分かり, bet_v3() に誤りがあることが検出できる.

6.2 発展的プロトタイピング技法に関する議論

6.2.1 抽象領域上の演算について

BLACKJACK の例で, 先にメソッドの出力であるデータドメインを発展をを考え, その出力値を一意に決定するために抽象値を用いて入力値のデータドメインや状態値のデータドメインを発展させる手法を示したが, アプリケーションの性質によっては, このようなドメインを見つけることが難しいことがある.

前章の在庫管理システムがその例である. 先の例でクラス `Shop_v3` のメソッド `alloc()` が実行できない理由は, 抽象値 $Amount$ と抽象値 $InStock$ の大小関係を決められないため, クラス `InventDom_v2` の演算 `sub()` や述語 `ge()` が定義できないことにあった. 先の例では, 最終的に $Amount$ や $InStock$ を Java の組み込み整数である `int` に発展させることで問題を回避した. しかし, 扱う製品が単純に個数や重さだけでは容量を扱えなかったり, 複雑な引き当て処理が必要だった場合にはそのデータドメイン上の演算や述語を単純に定義できない.

発展的プロトタイピング技法では, なるべく抽象値を用いて仕様を定義し段階的につくることによって, 具体的なデータ構造の決定を遅らせたり, 抽象値に基づいて実行を行うことにある. そこで非決定的な抽象値上の演算や述語は, 必要に応じて実行時に決定できるようにシステムを拡張する. 具体的には実行時に, プログラムの実行者に対してプロンプトを表示し, その入力によって演算結果などを決定するものとする.

実装

- クラス `DataGenerator` このクラスは, 実行時にユーザーが選択した, あるデータドメインに属する値を生成するクラスメソッド `generate()` と真偽値を生成するクラスメソッド `assign()` を持つ.

使用例

`DataGenerator` を用いて, クラス `InventDom_v2` の演算 `sub()`, 述語 `ge()` は以下のように定義できる.

```
class InventDom_v2 {
    String value;
    boolean ge(InventDom_v2 v) {
        boolean rtv = false;
        if (value.equals("InStock")  && v.value.equals("OutStock")) ||
            value.equals("InStock")  && v.value.equals("Minus"))    ||
            value.equals("OutStock") && v.value.equals("OutStock")) ||
            value.equals("OutStock") && v.value.equals("Minus"))) {
            rtv = true;
        } else {
            return DataGenerator.assign();
        }
        return rtv;
    }
```

```
    InventDom_v2 sub(AmountDom v0) {
        return (InventDom_v2) DataGenerator.generate("InventDom_v2");
    }
}
```

このように定義することで, 前章の例題在庫管理システムの発展段階 2 を実行することが可能となる.

第7章　おわりに

7.1　まとめ

本研究では, 大規模化, 複雑化するソフトウェアを構築する際に, 発展的プロトタイピング技法が有効であると考え, その技法を実際の開発に適用するために開発環境の設計, 実装を行った . またその開発環境の上で実際に開発実験を行った.

構築した開発環境は,

- 抽象実行処理系
- 発展関係エディタ
- 抽象実行ビジュアライザ

からなる.

抽象実行処理系は, 抽象実行プロキシオブジェクトによる実現を提案した. プロキシオブジェクトは, Java のリフレクション機能と XML を用いて実装した. また, 発展関係を定義するための XML 文書を記述する発展関係エディタを構築した. このエディタは, XML 文書からプロキシオブジェクトの生成に必要となる Java インタフェースやソースコードの雛形の自動生成も行う. さらに, JDI を用いて抽象実行時の情報を取得し, シーケンス図として視覚化する抽象実行ビジュアライザの構築を行った.

開発実験では,

- ブラックジャック
- 在庫管理システム

の構築を行った.

前者の実験によって, 抽象値を用いて段階的にメソッドを詳細化し, プログラムを構築することが可能であることを示した. また, 抽象実行を行うことで, 開発段階が異なるオブジェクト間でメソッド呼出しが実現できることを示した. 後者の例では, メソッドの直和分割や直積分割も用いて, プログラムを構築することが可能であることを示した. また, 抽象領域を定義することの難しさを示した.

7.2 結論

本研究では, 発展的プロトタイピング技法のための開発環境の構築とその上での開発実験の結果から, 以下の 2 点を結論とする.

- 抽象ドメインをうまく定めて, 段階的にプログラムを構築することで抽象実行を効果的に行うことが可能となる.
- 抽象ドメインをうまく定めることができないアプリケーションにおいても, 実行時に演算を定義することで抽象実行が可能となる.

ブラックジャックの構成例の抽象実行から, 抽象実行による開発途中段階でのプログラム実行は, 一般的な段階的構築時にスタブを用いて抽象実行を行う場合と比較して, 実行のコストや実行結果の精度において有効であることを示した. スタブを用いたテスト実行は, まず前提としてテスト実行者がスタブとなるダミーコードを生成する必要があるが, 抽象実行では他のオブジェクトの開発状況を意識する必要がなく, テスト実行を行うためのコードを生成する必要はない. しかも, 入力となるデータドメインやオブジェクトの状態となるデータドメインをうまく定めることで抽象実行によって得られる結果が抽象実行を行わない場合に得られる結果と全く同じになることを示した. このような結果はスタブからでは決して得られないものである. また完成したプログラムの実行結果と抽象実行による実行結果を比較することで, 完成したプログラムが, 抽象値レベルで与えられた仕様を満しているか確認できることを示した.

一方, 在庫管理システムの構築で, 抽象領域の定義の難しさを示した. 抽象領域によっては抽象領域上の演算をうまく定義できない. そのため, その演算を用いるプログラムは抽象実行による実行が不可能となる. しかし, 本研究ではその解決策として, 実行時にユーザーがその演算結果を明示することによって, 抽象実行が可能となることを示した.

7.3 今後の展望

大規模な開発への適用

本研究の開発実験の例題は規模の小さいものである. 今後の課題として発展的プロトタイピング技法や本研究で構築した開発環境を用いて, 大規模なプログラムを実際に構築することが可能であるか示す必要がある.

大規模な開発を行う際の問題として, ソースコードの記述量の増大があげられる. 本研究では, XML 文書よりソースコードの雛形を生成することで, プログラムの記述量を減らす方法を提案したが, 大規模な開発においてはこれだけでは十分ではない. さらにコード記述を省力化するために, メソッドボディ内のロジックに関しても自動生成する手法を考案する必要がある.

本研究で提案した XML 文書の DTD では, 型レベルでの静的な発展関係の情報しか含んでいないが, この記述に動的な情報をさらにつけ加えることで, メソッド内部のロジックを自動生成することが可能になると考えられる. 例えば, 他オブジェクトの状態に依存することなく, そのオブジェクトの状態とメソッドの入力値によって, 出力値やメソッド実行後のオブジェクトの状態値が分かる場合は, それらの情報からメソッドの内部ロジックを自動生成することが可能になると考えられる.

抽象実行を用いたテスト

本研究では抽象実行を用いて, 発展的に構成されたプログラムが前段階の与えられた仕様を満すかどうかテストする手法を提案した. しかし本研究で示した例では, 実際のテスト手法としては不十分である. だが, 抽象実行を利用することでもっと効果的なプログラムテストを行なえる可能性がある. 例えば, 抽象値によるテストを網羅的に行うことでテストカバレッジの向上を図ったり, 抽象実行を用いて複数の発展段階でのテストを自動化することが可能になると考えられる. これによって, 仕様の誤りの早期発見や, プログラムテストコストの削減が期待される.

他のオブジェクト指向言語への適用

本研究では, リフレクションと XML 技術を用いることで抽象実行のためのプロキシオブジェクトを実現できることを示した. これらの機能は, Java 特有のものではないため, CLOS, Smalltalk そして Python といったリフレクション機能を備えた他のオブジェクト指向言語に対しても適用することが可能であると考えられる.

プログラム保守への適用

本研究では, 新規開発時の段階的なプログラムの構築に焦点をあてていたが, この技法は, プログラム保守時にも有効であると考えられる. 例えば, ある直和に分割したメソッド $meth_1$, $meth_2$ があったとする. このプログラムの保守時に仕様の変更が発生し, $meth_2$ の入力となるデータドメインを再編成する必要があった場合, $meth_2$ に関しては再定義する必要があるが, $meth_1$ に関しては再利用できる可能性がある. このようにオブジェクトが扱うデータドメインの再分割や統合を行なった時に, 既存のプログラムのどの部分が再利用可能であるかといった, プログラムの再構成に関する議論を行なうことで, 発展関係エディタやソースコードの自動生成システムの拡張が可能となる. その結果, 発展的プロトタイピング技法をプログラム保守の技術に拡張できる可能性がある.

謝辞

本研究を行なうにあたり, 終始御指導賜りました片山卓也教授に深く感謝致します. 本研究を進める上で, 有益な助言を頂きました権藤克彦助教授に心よりお礼申し上げます. 尾崎弘幸氏には本研究のテーマと多くのアドバイスを与えて頂き深く感謝致します. 日頃よりお世話になった青木利晃助手をはじめとするソフトウェア基礎講座の皆様に厚くお礼を申し上げます.

最後に, 私を快く大学院に留学させて頂き, 貴重な機会を与えて頂いたうえ, 生活面でも全面的に支援くださいましたクオリカ株式会社 (旧社名 コマツソフト株式会社) に心より感謝致します.

参考文献

[1] Boris Beizer 著, 小野間彰, 山浦恒夫 訳, ソフトウェアテスト技法, 日経 BP 出版センター, 1994.

[2] Hiroyuki Ozaki, Katsuhiko Gondow and Takuya Katayama: *Evolutionary Prototyping Technique Using Abstract Interpretation in Java*, Proc. of International Symposium on Future Software Technology (ISFST '02), 2002.

[3] Nobukazu Yoshioka, Masato Suzuki and Takuya Katayama: *Incremental Software Development Method Based on Abstract Interpretation*, Proc. of IWSSD-10, IEEE, 1998.

[4] Matthew Flatt, Shriram Krishmamurthi and Matthrias Fellenisen: *Classes and Mixins.* In Proc. 25th ACM Symposium on Principles of Programming Languages, January 1997.

[5] Java Software: Reflection. In *Java 2 SDK, Standard Edition Documentation.* Sun Microsystems, Inc., 1999. available at http://java.sun.com/jdk/.

[6] 渡部卓雄. チュートリアル リフレクション. コンピュータソフトウェア, Vol.11, No.3,pp.5-14, May 1994.

[7] Java Software: Dynamic Proxy Classes. In *Java 2 SDK, Standard Edition Documentation.* Sun Microsystems, Inc., 1999. available at http://java.sun.com/jdk/.

[8] Erich Gamma, Richard Helm, Ralph Johnson, John Vissides, *Design Patterns Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[9] World Wide Web Consortium, *Extensible Markup Language(XML) 1.0 (Second Edition).* available at http://www.w3.org/TR/REC-xml.

[10] Object Management Group, *Unified Modeling Language(UML), Version 1.4.* available at http://www.uml.org/.

[11] Java Software: Java Platform Debugger Architecture. In *Java 2 SDK, Standard Edition Documentation.* Sun Microsystems, Inc., 1999. available at http://java.sun.com/jdk/.

付 録A　付録

A.1　発展関係定義DTD

```
<!ELEMENT evolutionary-java-program (classDomain|dataDomain)*>
<!ELEMENT classDomain (class, evolvedClass*)>
<!ATTLIST classDomain  classDomainID ID #REQUIRED>
<!ELEMENT class (field|method)*>
<!ATTLIST class classID   ID    #REQUIRED
                evolved   IDREF #IMPLIED
                level     CDATA #REQUIRED>
<!ELEMENT field EMPTY>
<!ATTLIST field  fieldID    ID     #REQUIRED
                 name       CDATA  #REQUIRED
                 domain     IDREF  #REQUIRED
                 evolved    IDREF  #IMPLIED >
<!ELEMENT method EMPTY>
<!ATTLIST method methodID   ID       #REQUIRED
                 name       CDATA    #REQUIRED
                 paras      IDREFS   #IMPLIED
                 return     IDREF    #IMPLIED
                 evolved    IDREF    #IMPLIED
                 available  (yes|no) #REQUIRED>
<!ELEMENT evolvedClass (class, evolvedClass*)>
<!ELEMENT dataDomain (type, (data|range)+, evolvedDataDomain*)>
<!ATTLIST dataDomain dataDomainID   ID    #REQUIRED>
<!ELEMENT evolvedDataDomain  (reifiedDataDomain   |
                              andDividedDataDomain |
                              orDividedDataDomain) >
<!ELEMENT reifiedDataDomain    (dataDomain)  >
<!ELEMENT andDividedDataDomain (dataDomain)* >
<!ELEMENT orDividedDataDomain  (dataDomain)* >
<!ELEMENT data  (simpleValue|complexValue)>
<!ATTLIST data   dataID   ID    #REQUIRED
                 name     CDATA #REQUIRED
                 evolved  IDREF #IMPLIED>
<!ELEMENT simpleValue (type)>
<!ATTLIST simpleValue value CDATA #REQUIRED>
<!ELEMENT complexValue (type, initValue*)>
```

```
<!ELEMENT initValue    (simpleValue|complexValue)>
<!ELEMENT range (from, to)>
<!ATTLIST range  evolved IDREF #IMPLIED>
<!ELEMENT from  (simpleValue) >
<!ELEMENT to    (simpleValue) >
<!ELEMENT type ( primitiveType | referenceType) >
<!ELEMENT primitiveType EMPTY>
<!ATTLIST primitiveType name (byte|short|int|long|char|float|double|boolean) #REQUIRED>
<!ELEMENT referenceType EMPTY>
<!ATTLIST referenceType name CDATA #REQUIRED>
```