博 士 論 文

抽象実行に基づく Javaプログラムの発展的プロトタイピングに関する研究

指導教官　片山 卓也 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

尾崎 弘幸

2004 年 3 月

要旨

プロトタイピングは，ソフトウェア開発において，実装工程といった遅い段階からの無駄なバックトラックコストを減らす．しかし，全体を見通した開発と途中段階での全体としての実行を両立することは難しい．したがって，プロトタイプ構築の遅い段階からのバックトラックコストが発生する．

そこで，本論文では，これらを両立できる発展的プロトタイピング技法を提案する．そのアイデアは，抽象解釈に基づいた段階的詳細化によるプロトタイプの構築である．抽象化したオブジェクトから構築を始め，オブジェクトを詳細化することで構築を進める．詳細化によるプロトタイピングが全体を見通す開発を実現する．さらに，オブジェクトを実行時に抽象化することでプロトタイプ全体としての実行を実現する．実行時にオブジェクトを抽象化し，プロトタイプを実行するメカニズムを抽象実行と呼ぶ．

抽象実行の実現には，オブジェクトの詳細化を厳密に決めることが重要である．本論文では，その詳細化の形式的な定義を与える．意味論に踏み込んで詳細化の整合性を保つように定義する．これにより，抽象実行時の振舞いの不整合を防ぐ．

次に，形式的なオブジェクトの詳細化に基づいて，抽象実行支援ツールの実装を与える．ツール支援なく抽象実行を実現することは非常にコストが高い．なぜなら，抽象実行では，データ，メソッド，クラスといった様々な言語要素の変換（抽象化）を必要とするからである．本論文では，仲介オブジェクトを導入し，機械的な抽象実行を実現している．リフレクション技術と XML 技術を用いて仲介オブジェクトを機械的に生成することができるので，効率的にプロトタイプを構築することが期待できる．

最後に，本技法の詳細化の考え方を拡張することで fragile base class problem の起きない安全なクラス継承を実現できることを示す．クラス継承は，オブジェクト指向技術の核となるメカニズムの一つである．しかし，fragile base class problem として知られる致命的な問題がある．本論文では，機能詳細化に機能追加を導入し，クラス継承を実現する．機能詳細化と機能追加によるクラス継承の単調性を証明することで，fragile base class problem が起きないことを示す．

目 次

図 目 次

表 目 次

第 1 章

はじめに

1.1 研究の背景

近年，コンピュータや電子機器が急速に発達し，高品質ソフトウェアをより早く開発しなければならない現実に直面している．インターネットの普及がソフトウェア市場を拡大し，ソフトウェア開発の競争をさらに激しくしている．したがって，ソフトウェア産業で勝ち残っていくためには，高品質ソフトウェアの開発の生産性をますます向上していかなければならない．

かつて，ソフトウェア開発の生産性を向上するためにウォーターフォールモデルが提案された．その提案以前は，開発者の経験に頼ったソフトウェア開発だった．1960 年代には，ソフトウェアの大規模化が進み，従来の経験則のみでは大規模ソフトウェアの開発が難しいという認識が生まれた．これがソフトウェア危機 (software crisis) である．そこから，ソフトウェアを経験則ではなく体系的に開発するソフトウェアプロセスモデルとしてウォーターフォールモデル (waterfall model) が提案された．ウォーターフォールモデルでは，ソフトウェア開発工程は，要求分析 (requirements analysis)，設計 (design)，実装 (coding)，テスト (testing)，保守 (maintenance) からなる．そして，n 番目の工程が終わってから n+1 番目の工程に入るというように逐次的に開発を進める (図 1.1)．例えば，要求分析工程が終わると設計工程に入り，設計工程が終わると実装工程に入る．初期のウォーターフォールモデルでは，ソフトウェアの一生 (software life cycle) は，後戻りすることができないという考え方であったが，現在は，図 1.1 のように一つ前の工程にのみフィー

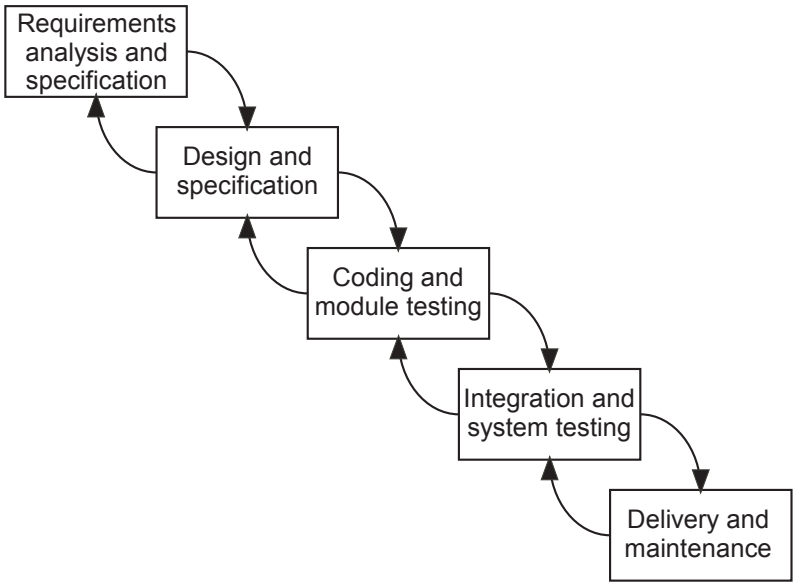

図 1.1: Waterfall model with feedback

ドバックを許すモデルとなっている．しかしながら，後戻りはやむ得ない場合のみとされている．

ウォーターフォールモデルの問題点は，(1) 顧客が自分の要求を明確にすることができない場合や (2) 開発者が今までに開発経験のないソフトウェアを開発する場合に各工程を逐次的に進むことが難しいことである．(1) の場合，顧客は要求分析工程だけで，自分の要求を明確にすることは難しく，開発者は設計のための十分な要求仕様を獲得することが難しい．(2) の場合，実装してみなければ設計の細部を決定することができない場合があり，設計工程までで実装するのに十分な設計仕様を決定することが難しい．したがって，先の工程に進み，そこで得た結果を反映して前の工程に戻ることになり，逐次的に各工程を進むことが難しくなる．その結果，開発コストは高くなり，生産性が落ちてしまう．

この問題を解決する一つの方法としてプロトタイピングがある．プロトタイピングとは，要求分析工程や設計工程において，これから開発しようとしているソフトウェアプロダクトのプロトタイプ (prototype) を構築し，実際に動作を確認することで，顧客と開発者の間の考えを早い段階で統一しようとする試みである．その結果，実装工程といった遅い段階からの無駄なバックトラックを防ぎ，開発コストを下げることができる．実際，[11] によると，プロトタイピングを取り入れ

たウォーターフォールモデル，プロトタイピングモデル，スパイラルモデルなどの多くのソフトウェアプロセスモデルにプロトタイピングは組み込まれている．

プロトタイプには，大きく分けて，使い捨てプロトタイプ (throwaway prototype) と発展プロトタイプ (evolutionary prototype) の 2 種類がある [3, 11]．使い捨てプロトタイプとは，これから開発するソフトウェアプロダクトの実現可能性 (feasibility) の評価や要求の確認のために構築するプロトタイプで，目的を果たすと捨てる．発展プロトタイプとは，逆に，目的を果たしても捨ててしまわず，最終的な生産物の原型となるプロトタイプである．

使い捨てプロトタイプは，安く構築できるという利点と完成するまで顧客が評価できないという欠点がある．使い捨てプロトタイプは，捨てることが前提であるため，ソフトウェア品質を気にする必要はない．したがって，安くプロトタイプを構築することができる．しかし，顧客が評価するのはプロトタイプの完成後である．言い替えると，プロトタイプ構築の途中における顧客の評価は考慮されていない．

逆に，発展プロトタイプは，構築コストが使い捨てプロトタイプよりも高くなるが，途中での評価ができる．発展プロトタイプは，Boehm の定義する evolutionary development model[1] の考え方に基づいている．そのモデルは次のように定義されている．

> A model whose stages consist of expanding increments of an operational software product, with the direction of evolution being determined by operational experience.

したがって，プロトタイプ構築の途中でも顧客が評価することはできる．しかし，大幅な変更に弱く，慎重に構築を進める必要があるので，使い捨てプロトタイプと比べるとコストはかかる．

使い捨てプロトタイプを evolutionary development model の考え方に基づいて構築する手法として吉岡が提案した ISDR 法 [13] がある．ISDR 法は，従来，設計工程に適用する段階的詳細化 (stepwise refinement) をプログラミングに応用したものである．その特徴は，プロトタイプ構築の中間段階で実行可能である点である．従来の段階的詳細化は途中段階での実行を考慮していない．例えば，Refinement Calculus[9] である．Refinement Calculus とは，仕様からコードを記述するための

プログラムの詳細化を形式的に体系づけた公理系である．プログラムの前後で成立する条件を論理式で記述し，その事前条件を弱める，あるいは事後条件を強めることでコード化する手法である．しかし，途中段階では，部分的にコード化されてはいるとはいえ，自然言語で書かれた仕様の部分も含むため，全体としては実行可能ではない．ISDR 法では，途中段階のプログラム実行を実現するために抽象解釈を用いている．具体的には，純粋な関数型言語上で抽象解釈に基づく詳細化を形式的に定義している．

ソフトウェア開発にオブジェクト指向言語を用いることが，近年，主流となりつつある．オブジェクト指向言語では，ソフトウェアをクラスの集まりとして記述する．洗練されたクラスは，再利用性，生産性，保守容易性といった様々なソフトウェア品質を高める．オブジェクト指向言語には，純粋な関数型言語にはない概念，例えば副作用や継承など，があるため，オブジェクト指向言語上で ISDR 法を素朴に適用することは難しい．

そこで，本論文では，ISDR 法の考え方をオブジェクト指向言語に応用することを試みる．多くのオブジェクト指向言語があるが，本論文では，Java を対象としている．その理由は，言語機能がシンプルであることと広く使われていることである．

1.2 本研究のアイデアと目的

本論文では，抽象解釈 (abstract interpretation)[4] に基づく発展的プロトタイピング技法を提案する．これは，直観的には，抽象解釈に基づく Java プログラムの段階的詳細化を用いたプロトタイプ構築法である．オブジェクトの機能を増やす” 発展” を” 詳細化” で実現する．本技法の最大の特徴は，途中段階での実行が可能な点である．この特徴は，抽象解釈に基づくことによって得られる恩恵である．

抽象解釈とは，プログラムの性質を解析するための理論的な枠組である．大規模なプログラムの性質を解析することは多大な労力を必要とする．抽象解釈のアイデアは，解析したい性質に焦点をあてて不必要な情報を削るという抽象化によってその労力を減らすことである．そして，様々な性質を解析するための理論的な体系を提供している．

抽象解釈の手順は，次の通りである．まず解析しようとしている既存のプログラムの扱うデータを抽象ドメイン上に定義する．次に，プログラムの振舞いをその抽象ドメイン上の振舞いとして定義する．そして，その抽象化した振舞いを用いて解析を行なう．既存のプログラムを P，抽象化したプログラムを $P^{\#}$ とすると，抽象解釈では，

$$P \Rightarrow P^{\#}$$

という順序でそれぞれのプログラムを記述している．ここで，抽象化することを関数 α で表現すると，P と $P^{\#}$ の間の関係は，次のように記述することができる．

$$P^{\#} = \alpha(P)$$

本研究のアイデアは，抽象解釈を逆方向に利用し，プログラムを詳細化することによって構成することである．つまり，

$$P^{\#} \Rightarrow P$$

という順序でプログラムを構成することである．もちろん，P と $P^{\#}$ の間の関係は，$P^{\#} = \alpha(P)$ である．言い替えると，P は $P^{\#}$ を詳細化することで得たプログラムである．

このような構成手法は，部分的にではあるが，複雑で規模の大きなプログラム P を構成する前に，さほど複雑でない規模の小さいプログラム $P^{\#}$ で実行/評価することができるという利点がある．早い段階で $P^{\#}$ を評価することで，ソフトウェアプロダクトの実現可能性についてあたりをつけることができる．しばしば，P がほとんど完成しているような非常に遅い段階で実現不可能であることを知ることがある．また，P が完成した後の評価で要求理解に誤りがあることを知ることがある．この時，プログラムを作り直すバックトラックコストは高い．一つのソフトウェアを構成するために，多くのプロトタイプを構築するという現状を考えると，このバックトラックコストは非常に高くなる．本技法は，このコストを減らすことが期待できる．

さらに，オブジェクト単位で考えると，記述と実行/評価のサイクルをより小さくできる．あるオブジェクトの一部を変更すると，別のオブジェクトにその影響を及ぼし，すべての変更が完了するまで実行/評価が困難となる．オブジェクト指

向言語では，モジュール性が高く，他のオブジェクトに影響を及ぼすことがない場合もあるが，本研究の対象としているのは機能の変更なので，多くの場合，影響を及ぼす．ここで，

$$P_1^{\#} \Rightarrow P_2^{\#} \Rightarrow \cdots$$

と開発が進むとする．$P_1^{\#}$ のオブジェクトの一つを変更し，かつ $P_1^{\#}$ よりも先に進み $P_2^{\#}$ まで到達していない場合，その段階で実行/評価が難しいということである．オブジェクト Obj を記述する前に抽象化したオブジェクト $Obj^{\#}$ を記述するということは，必要に応じて，Obj の代わりに $Obj^{\#}$ を使うことができるということである．ここで，オブジェクト Obj_1 と Obj_2 がプログラム P を構成していることを

$$P = \{Obj_1, Obj_2\}$$

と書くことにする．プログラム P と $P^{\#}$ が，

$$\begin{aligned} P^{\#} &= \{Obj_1^{\#}, Obj_2^{\#}\} \\ P &= \{Obj_1, Obj_2\} \end{aligned}$$

であり，かつ

$$\begin{aligned} Obj_1^{\#} &= \alpha(Obj_1) \\ Obj_2^{\#} &= \alpha(Obj_2) \end{aligned}$$

とした時，$P^{\#}$ と P の途中段階 $P' = \{Obj_1^{\#}, Obj_2\}$ において，$Obj_1^{\#}$ とのコラボレーションでは，Obj_2 の代わりに $Obj_2^{\#}$ を利用することで実行をすることができる．ここで，開発の順序は

$$P^{\#} \Rightarrow P' \Rightarrow P$$

である．実行時に，Obj_2 の代わりに $Obj_2^{\#}$ を利用するという代替を行なえば，部分的にでも Obj_2 に閉じている部分では，Obj_2 を用いて計算することになる．静的に Obj_2 から $Obj_2^{\#}$ への変換を行なうと，P' の振舞いは $P^{\#}$ と同じになる．この静的な変換は，実行時の変換と比べて非常に簡単な仕組みで実現できる．なぜなら，計算途中の値を実行が継続できるように管理する必要がないからである．しかし，変更した部分を利用することは全くない．逆に，実行時にその代替を行なうと，$Obj_2^{\#}$ を必要としない間，つまり Obj_2 で実行できる間は Obj_2 を利用することができ，全てではないが Obj_2 を使った実行が可能となる．もちろん，場合によっ

ては全くObj_2を使わないこともある．しかし，変更したオブジェクトを部分的にでも使う場合があるので，静的な変換よりも早い段階で誤りを発見できる可能性が高くなる．P'の段階で誤りを発見できれば，Pが完成するまでの開発コストを減らすことができ，より効率的なプロトタイピングが期待できる．本論文では，このオブジェクトを実行時に抽象化しプロトタイプを全体として実行するメカニズムを抽象実行と呼ぶ．

まず，本研究では，オブジェクトの詳細化を CLASSICJAVA[5] 上で形式的に定義する．CLASSICJAVA とは，Java の形式化の一つである．形式的に言語の意味論を定義した CLASSICJAVA を利用する理由は 2 つある．一つは，オブジェクトの詳細化を形式的に定義するためには，正しく振舞いを抽象化/詳細化しているかどうかの整合性を言語の意味論に踏み込んで議論する必要があるからである．計算機による抽象実行の実現には，まず，様々な言語要素について抽象化/詳細化の形式的な関係づけが必要になる．シンタックス上での関係づけだけでは不十分で，正しく実行できることを保証するために，意味論の上での関係づけが必要である．もう一つの理由は，CLASSICJAVAが非常にコンパクトな形式化であるため，扱いやすいからである．

次に，その形式化に基づく抽象実行を Java で実現するためのプログラミング環境の実装方法を提案する．オブジェクトの詳細化を形式的に定義すると，オブジェクトの抽象化を計算機で実現することができる．原理的には，オブジェクトの抽象化が機械的に可能ならば，抽象実行メカニズムも機械的に可能である．しかし，Java 上での抽象実行メカニズムの実現には以下の 3 つの問題が生じる．

1. 異なるインターフェースを持つオブジェクトへの置換が難しい．
2. オブジェクトの変換にともなう参照の再構築を行なうメカニズムが必要となる．
3. 構築するプロトタイプ毎に抽象実行メカニズムを実現する必要がある．

そこで，本論文では，仲介オブジェクトを導入することでこれらの問題を解決する．仲介オブジェクトは，リフレクション技術の一つである Dynamic Proxy Class API と XML 技術を用いることで機械的に生成することが可能である．これによっ

てプログラマは抽象実行に関するコードを記述する必要がなくなり，効率的なプロトタイピングが期待できる．

最後に，本技法の詳細化の考え方を拡張することで，クラス継承の持つ問題を解決できることを示す．クラス継承は，オブジェクト指向技術の核となるメカニズムの一つである．しかし，fragile base class problem[8] として知られている致命的な問題がある．これは，一見正しく見える機能拡張であるのにもかかわらず，開発者の予期せぬ振舞いをクラス継承が引き起こすという問題である．クラス継承の中心は，機能詳細化と機能追加である．そこで，本論文では，本技法に機能追加の概念を導入し，クラス継承を実現する．そして，実現したクラス継承では，fragile base class problem が生じないことを証明する．

1.3 本論文の構成

本論文の構成は以下のとおりである．

第 1 章 はじめに

本研究の背景，アイデア，期待される成果について述べる．

第 2 章 発展的プロト タイピング技法

ここでは，発展的プロトタイピング技法の概念を長方形オブジェクトの構成例を用いて詳しく説明する．また，その例を用いて抽象実行の概念を説明する．

第 3 章 オブジェクト 発展の形式化

この章では，本技法によるオブジェクトの発展を形式的に定義する．オブジェクトの発展は，データ，状態，機能，クラスの発展からなる．この形式化は Java の形式化の一つである CLASSICJAVA を用いている．この形式化は抽象実行メカニズムを計算機で実現するために必要である．

第 4 章 抽象実行メカニズムの実現

ここでは，抽象実行メカニズムの実現方法を示す．Java で抽象実行メカニズムを実現する時，3 つの問題が発生する．それを解決するために仲介オブジェ

クトを導入する．これは，リフレクション技術と XML 技術を用いることで機械的に生成することができる．それを実現するためのプログラミング環境についても述べる．

第 5 章 ソフトウェア開発事例

本技法を用いて，ブラックジャックシステムと商品在庫システムを構築し，本技法の有用性を示す．

第 6 章 機能発展と機能追加によるクラス継承の実現

機能追加によるクラスの機能拡張を導入し，機能発展と機能追加による機能拡張を提案する．そして，この機能拡張がクラス継承における致命的な問題”fragile base class problem” を起こさないことを示す．そのために，これらを組み合わせたプロトタイピングが機能増加に関して単調であることを証明する．

第 7 章 議論

ここでは，本論文で実現した抽象実行メカニズムの性能評価と発展的プロトタイピング技法の関連研究について述べる．

第 7 章 まとめ

最後にまとめと今後の課題について述べる．

第 2 章

発展的プロトタイピング技法

本章では，発展的プロトタイピング技法の概念，開発プロセス，抽象実行メカニズムを構成例を用いて説明する．本技法の重要なポイントは，振舞いの性質を保存するようにオブジェクトを詳細化することである．この性質の保存が正しい抽象実行を可能にし，プロトタイピングの途中段階での実行を実現している．

2.1 概要

発展的プロトタイピング技法は，データの具体化に基づく漸進的なプロトタイピングのための理論的枠組である．直観的には，抽象解釈に基づく段階的詳細化を用いたプロトタイピング技法である．しかし，一般的な段階的詳細化と違って，本技法は，未完成のプロトタイプを全体として実行できる．その実行を実現するメカニズムが抽象実行である．抽象実行メカニズムの原理は，全体として実行できるようにオブジェクトを抽象化し実行を継続することである．したがって，抽象実行を実現するためには，(1) 抽象化のための数学的な関係づけ，(2) 抽象実行できない状況を排除するための制限が必要である．

本技法では，データとオブジェクトを区別している．Java では多くのデータをオブジェクトとして実現している (例えば，文字列) が，本技法では，内部状態が変化しないオブジェクトをデータと，変化するオブジェクトをオブジェクトと呼んでいる．この区別によって，数学的な関係づけを簡単化することができる．その詳細は，後の章で説明する．

本手法を用いたプロトタイピングの手順は，

1. データの観点から仕様を抽象化
2. 原始プロトタイプを作成
3. データの具体化に沿ったプロトタイプの発展

である（図 2.1）．本技法では，データの具体化に沿ってオブジェクトの機能を漸進的に増やすことで，より複雑なプロトタイプを構成する．途中段階でプロトタイプにオブジェクトを追加することはできない．これは，オブジェクトの追加が抽象実行を妨げるからである．例えば，単にオブジェクト obj を追加したとする．実行の途中で obj の代わりに obj を抽象化したオブジェクト $obj^{\#}$ を必要とした場合，$obj^{\#}$ は開発の途中に存在しないため，抽象化できない．抽象化の情報を与えることなく計算機によって obj から $obj^{\#}$ へ変換することは，一般的に非常に難しい．したがって，プロトタイプを構成するオブジェクト群は，あらかじめ発見されているものとし，開発の途中でオブジェクトを追加することを制限する．

仕様の抽象化では，各データに関してそれぞれ一つのデータに抽象化する．そのデータは，本技法において最大の抽象度を持つデータとなる．データには，オブジェクトの機能への入出力データやオブジェクト内に格納されるデータがある．そして，オブジェクト毎に機能全体を表す一つの機能に抽象化する．その機能は，抽象化したデータ上の振舞いとして定義する．一つのデータや機能に抽象化する理由は，非常に簡単なところからプロトタイピングを始め，系統的にプロトタイピングを行なうためである．

次に，抽象化した仕様に基づいて**原始プロトタイプ**を作成する．原始プロトタイプは，系統的に記述可能である．なぜなら，それぞれのオブジェクトは一つの (抽象化された) 機能からなり，その機能は，それぞれ一つの入出力データしか持たないからである．この原始プロトタイプが本技法における最大の抽象度を持つプロトタイプである．

本技法では，**データの具体化**に沿った**オブジェクトの機能発展** (以後，機能発展と省略する) によってプロトタイプの機能を増やし，十分にデータが具体化されるまで繰り返す．データの具体化とは，抽象解釈の枠組に基づいて，より多くの情報をもつデータにすることである．例えば，図 2.2 は，整数全体を表す値 `Int` から

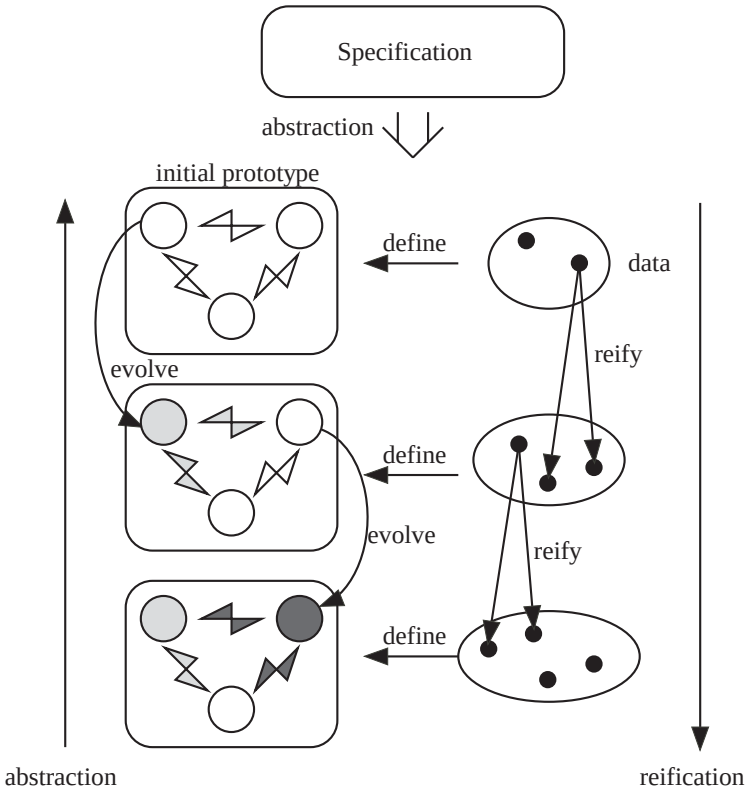

図 2.1: 発展的プロトタイピング技法の全体像

正の整数全体を表す値 `Pos`，ゼロを表す値 `Zero`，負の整数全体を表す値 `Neg` への具体化を表している．重要なことは，値を追加するのではなく，具体化することである．機能発展とは，オブジェクトの機能 $F^{\#}$ と F において，任意の入力と出力について，入力が具体化されているならば，出力も具体化されているように $F^{\#}$ を F に変更することである（図 2.3）．つまり，任意の入力データ $v^{\#}$，v について，

$$v^{\#} \prec^{\mathcal{D}} v \Rightarrow F^{\#}(v^{\#}) \prec^{\mathcal{D}} F(v)$$

が成立するように変更することである．ここで，$v^{\#} \prec^{\mathcal{D}} v$ は v が $v^{\#}$ を具体化したデータであることを表している．例えば，図 2.2 では，`Int` $\prec^{\mathcal{D}}$ `Pos` である．

この入出力データに関する整合性を保つ機能発展は，不均一な抽象度をもつオブジェクト群からなるプロトタイプの実行を可能にする．データの具体化や機能発展の数学的な関係を用いてオブジェクトや機能を抽象化することができる．この抽象化を**縮退**と呼ぶ．縮退よって実行列の抽象度をそろえることで，プロトタ

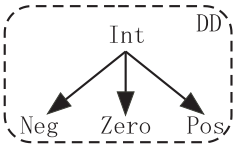

図 2.2: データの具体化例

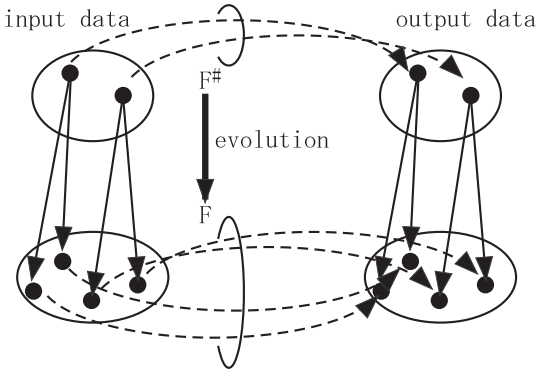

図 2.3: 機能発展の概念

イプは全体として実行可能となる．抽象化したデータを再び具体化したデータに実行時に復元することは難しいため，本技法では，オブジェクトの抽象度を上げる方向でそろえる．

一般的に，オブジェクトは内部状態 (インスタンス変数とその値のペアの集合) を持ち，その機能は内部状態に関して副作用を持つため，機能詳細化をデータの具体化のみで関係付けることは不十分である．機能の副作用を扱うために，本手法では，**ストアの発展**を導入している．ストアとは，プロトタイプを構成するオブジェクトの現在の状態のあつまりである．ストアの発展とは，抽象化したストアをより詳細なストアにすることである．あるオブジェクト obj の機能を利用した場合，もしそのオブジェクトが他のオブジェクトの機能を利用しないなら，obj の機能の出力は，他のオブジェクトの状態に影響されない．しかし，他のオブジェクトの機能を利用する場合，obj の機能の出力は，obj の現在の状態のみならず，他のオブジェクトの状態にも影響される．したがって，本手法では，入力が具体化されていて，かつ機能を利用する前のストアが発展しているならば，出力が具体化されていて，かつ機能を利用した後のストアが発展しているように機能を発展

させる．これによって，状態に関する副作用を扱うことが可能となる．

さまざまな機能発展が考えられるが，我々は，**メソッドの詳細化**，**メソッドの直積分割**，**メソッドの直和分割**の 3 つの最も基本的な機能発展を提供している．直観的に，メソッドの詳細化とは，より具体化された入出力データを扱うメソッドに変更することであり，メソッドの直積分割とは，メソッドをメソッド列に分割することであり，メソッドの直和分割とは，メソッドをメソッドの選択に分割することである．メソッドの直積分割では，入出力は，それぞれ直積になるように具体化し，メソッドの直和分割では，それぞれ互いに素となる集合に具体化する．これらは，構造化プログラミングの考え方に基づいている．

2.2 構成例

長方形オブジェクトの構成例を用いて，本技法を用いたプロトタイプ構築と抽象実行の概念を詳しく説明する．以下は，ここで構築する長方形オブジェクトの仕様である．

長方形オブジェクトの仕様

長方形オブジェクトには，

- サイズを変更する機能
- 移動する機能

がある．ここで扱う座標系は平面座標とする．サイズ変更機能は，変更後の大きさを入力として取る．移動機能は，移動後の座標をを入力とする．さらに，X 軸上の移動と Y 軸上の移動を別々に行なう．X 座標と Y 座標は共に整数とする．

2.2.1 仕様の抽象化

まず，オブジェクトの機能に着目して，データを抽象化し，機能をその上の振舞いとして抽象化する．仕様の抽象化では，各オブジェクトを最も抽象化した一つの機能を持つオブジェクトに抽象化することが目的である．

長方形オブジェクトが扱うデータには，

- 現在位置
- 現在の大きさ
- サイズ変更機能の入力
- 移動後の座標 (X 軸と Y 軸)

がある．現在位置と大きさは，オブジェクトの内部に格納されるデータで，他はそれぞれの機能の入力データである．さらに，内部に格納されるデータは，各機能の入力として与えられる．長方形オブジェクトの機能は，2つあり，どちらとも入力を受け，内部状態を変更し，なにも出力しない機能である．この観点から，機能全体を `op()` メソッドに抽象化する．ここで，`op()` は，位置と大きさを抽象化した値”data” を受け取り，何も返さないメソッドとする．したがって，最も抽象化した長方形オブジェクトのクラスは，図 2.4 となる．

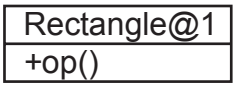

図 2.4: 最も抽象化した長方形クラス

2.2.2 原始プロトタイプの構築

次に，抽象化したクラスをコード化し，最も抽象化したプロトタイプを構築する．この例では，プロトタイプは長方形オブジェクトのみからなる．

```
/* version 1 */
class Rectangle {
    OpDom info = new OpDom ("data");
    void op (OpDom in) {
        info = in;
    }
}
```

インスタンス変数 `info` は，位置と大きさを抽象化した値”data” を格納するための変数であり，現在の位置と大きさを表す．

2.2.3 プロトタイプの発展

長方形オブジェクトを形成するクラスは，図 2.5 に示す流れで発展する．最終的なデータドメインは，図 2.6 である．ここで，クラス名に含まれている @n は n バージョンを表しているが，プログラム上には現れない．これは，異なるバージョンのクラスを区別するために名前を変えているが，プログラム上では，名前を同一視する必要があるからである．最初は，メソッドの直和分割によって，`op()` メソッドを `setSize()` メソッドと `move()` メソッドに分割する．ここで，`setSize()` は，サイズ変更機能を抽象化したメソッドであり，`move()` は，移動機能を抽象化したメソッドである．その結果，`Rectangle@2` クラスを得る．次に，`move()` を `setX()` メソッドと `setY()` メソッドにメソッドの直積分割を用いて分割する．ここで，`setX()` は，X 軸上の移動機能を抽象化したメソッドであり，`setY()` は，Y 軸上の移動機能を抽象化したメソッドである．これによって `Rectangle@3` クラスを作る．3 つ目のステップは，`setX()` と `setY()` の入力を具体化することで，`Rectangle@4` クラスを構築することである．このステップでは，入力である移動距離を正の整数，負の整数，ゼロに具体化する．最後に，`setX()` と `setY()` の入力を整数に具体化し，`Rectangle@5` クラスを構築する．

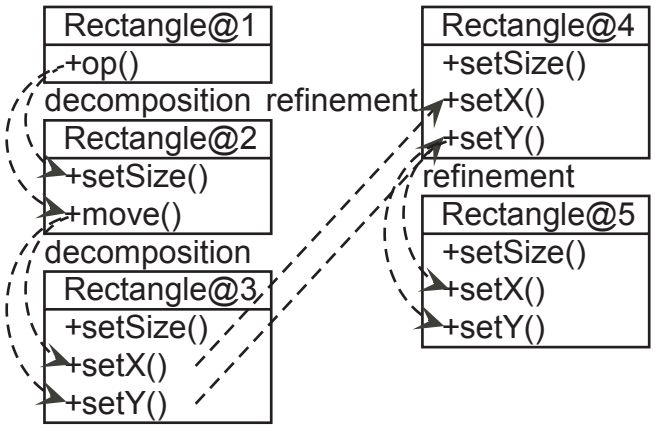

図 2.5: 長方形クラスの発展

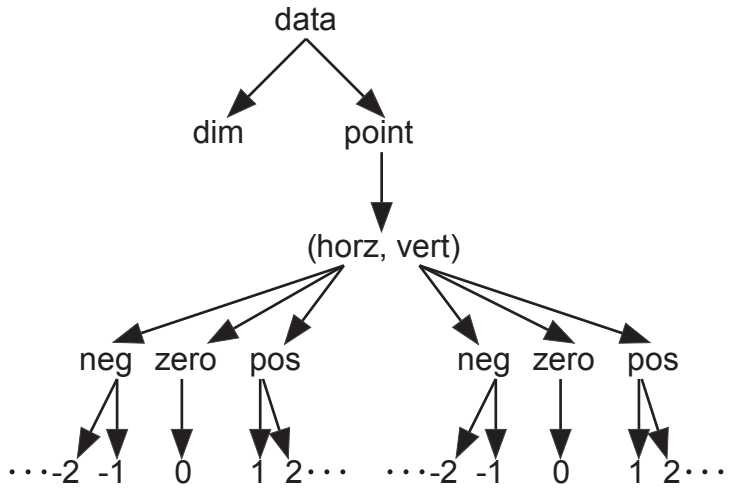

図 2.6: 最終的なデータド メイン

発展 1

以下のプログラムは，Rectangle@1 クラスを発展した Rectangle@2 クラスである．

```
/* version 2 */
class Rectangle {
    DimDom size = new DimDom("Dimension");
    PointDom location = new PointDom("Point");
    public void setSize(DimDom in) {
        size = in;
    }
    public void move(PointDom in) {
        location = in;
    }
}
```

op() を直和分割して，setSize() メソッドと move() メソッドを得る．メソッドの直和分割とは，どれか一つのメソッドが実行されるようにメソッドを分割することである．ここでは，op() を setSize() か move() のどちらかが実行されるように分割している．この直和分割に沿って，入力データ”data” を長方形の大きさを抽象化した値”dim” と移動後の座標を抽象化した値”point” に具体化している．

setSize()は，サイズ変更機能を抽象化したメソッドで，変更後の大きさを抽象化した値”dim” を受け取り，内部状態を変更する．同様に，move()は，移動機能を抽象化したメソッドで，移動後の座標を抽象化した値” point” を受け取り，内部状態を変更する．共に出力データはない．

この直和分割にともないインスタンス変数 info を size と location に詳細化している．これは，op()を直和分割することで，位置と大きさを別々に扱う必要があるからである．

発展 2

次に，move()を setX()と setY()に直積分割する．その結果，以下の Rectangle@3 クラスを得る．

```
/* version 3 */
class Rectangle {
    DimDom size = new DimDom("Dimension");
    AbstHorzDom x = new AbstHorzDom("Horz");
    AbstVertDom y = new AbstVertDom("Vert");
    public void setSize(DimDom in) {
        size = in;
    }
    public void setX(AbstHorzDom in) {
        x = in;
    }
    public void setY(AbstVertDom in) {
        y = in;
    }
}
```

メソッドの直積分割とは，メソッドをメソッドの実行列に分割することである．ここでは，move()を setX();setY()に分割している．setX()と setY()は，それぞれ X 軸上と Y 軸上の移動機能を抽象化したメソッドである．つまり，長方形オブジェクトの移動を X 軸上で移動して Y 軸上で移動するように分割している．

この直積分割にもとない座標情報を抽象化した”point” を X 座標を抽象化した”horz” と Y 座標を抽象化した”vert” の組に具体化している．さらに，これらの値を扱えるように，インスタンス変数 location を x，y に詳細化している．

発展 3

移動メソッド setX() と setY() を詳細化し，以下の Rectangle@4 クラスを作る．

```
/* version 4 */
class Rectangle {
    DimDom size = new DimDom("Dimension");
    HorzDom x = new HorzDom("Zero");;
    VertDom y = new VertDom("Zero");;
    public void setX(HorzDom in) {
        x = in;
    }
    public void setSize(DimDom in) {
        size = in;
    }
    public void setY(VertDom in) {
        y = in;
    }
}
```

メソッドの詳細化とは，より具体化した入出力データを扱うメソッドに詳細化することである．ここでは，setX() と setY() を詳細化している．Rectangle@3 クラスでは，それぞれ”horz” と”vert” しか扱っていない．これらを正の整数を表す値”pos”，負の整数を表す値”neg”，ゼロを表す値” zero” に具体化し，それに沿ってメソッドを詳細化している．具体化した値を扱えるようにインスタンス変数 x と y の型を詳細化している．

発展 4

最後に，setX() と setY() の入力を整数に具体化し，以下の Rectangle@5 クラスを得る．

```
/* version 5 */
class Rectangle {
    DimDom size = new DimDom("Dimension");
    int x = 0;
    int y = 0;
    public void setX(int in) {
```

```
        x = in;
    }
    public void setSize(DimDom in) {
        size = in;
    }
    public void setY(int in) {
        y = in;
    }
}
```

前のメソッドの詳細化と同様に，`setX()`と`setY()`を詳細化している．ここでは，それぞれ整数の座標を受け取り，内部状態を変更するように詳細化している．

2.3 抽象実行

抽象実行とは，プロトタイプを全体として実行するためのメカニズムである．具体的には，オブジェクトの機能発展を逆方向に使う，つまり抽象化し，プロトタイプを構成するオブジェクト群の抽象度をそろえて実行することである．例えば，上の例題で，長方形クラスが`Rectangle@2`で，そのインスタンスを`rect`とする．この時，

```
rect.setX(3)
```

を実行できない．なぜなら，`Rectangle@2`クラスが`setX()`メソッドを持っていないからである．また，長方形クラスが`Rectangle@4`まで進んでいるとき，

```
rect.move(point)
```

を実行できない．これらは，呼び出し側の抽象度と呼ばれる側の抽象度が異なっていることが原因である．抽象実行は，オブジェクトの抽象化を用いて抽象度をそろえ，実行を継続する．

抽象度が異なるオブジェクト間のメソッド呼び出しでは，次の 2 つの状況がある．(1)caller オブジェクトが callee オブジェクトよりも詳細である場合と (2) 逆に callee オブジェクトが caller オブジェクトよりも詳細である場合である．どちらの場合でも，より詳細であるオブジェクトを抽象化することで，原理的には，実行

が可能となる．機械的にオブジェクトを発展することは非常に難しく，抽象化する方向でオブジェクトを変換する．本論文では，抽象実行のためのオブジェクトの抽象化変換を**オブジェクトの縮退**と呼ぶ．

(1)の状況の中で，抽象度が異なるのにもかかわらず，オブジェクトの縮退をすることなく実行を継続することができる場合がある．それは，メソッド呼び出しを抽象化できる場合である．例えば，`Rectangle@4`クラスのオブジェクトに対して，`setX(2)`を実行することを考える．`Rectangle@4`クラスの`setX()`メソッドは，引数の型が`int`ではないので，`setX(2)`を実行できない．しかし，`setX(2)`を`setX(pos)`に変換したとすると，実行可能となる．このメソッド呼び出しの抽象化を，本論文では，**メソッドの縮退**と呼ぶ．

メソッドの縮退では，返り値の型が重要である．この例では，`void`なので，問題は生じない．問題が生じるのは，メソッドの縮退の結果，戻り値の型が変わってしまうケースである．このため，メソッドの縮退を適用できるのは，以下の3つの条件を満たす場合である．

- メソッドを抽象化できる
- 実引数を抽象化できる
- メソッドの縮退を行なっても戻り値の型が変化しない

原理的には，オブジェクトの縮退のみで抽象実行を実現できるが，抽象実行の効率化のためにメソッドの縮退を導入する．オブジェクトの縮退では，内部状態についても抽象化する必要がある．このため，インスタンス変数の数が多くなるにつれてオブジェクトの縮退にかかる時間が増える．一方，メソッドの縮退では，メソッドとデータを抽象化するだけなので，インスタンス変数が増加しても変化しない．もちろん，メソッドの実引数が増えるほど時間がかかるが，一般的には，さほど引数の数は増えない．したがって，メソッドの縮退を導入することで，効率的な抽象実行を期待できる．

第 3章

オブジェクト 発展の形式化

本章では，オブジェクト発展の形式化を与える．この形式化は CLASSICJAVA 上で定義している．オブジェクト発展は，データ，状態，機能，クラスの発展からなる．したがって，まず CLASSICJAVA について述べ，その後，それぞれの発展を形式的に定義する．

3.1 概要

オブジェクトの発展は，非常に多くの言語要素の発展関係からなる．大きく分けると，データ，状態，機能，クラスの発展からなる．これから定義する様々な関係を分類したものが表 3.1 である．

定義の方針は，まず，データに関する発展関係を定義する．次に，データの発展関係を用いて状態の発展関係を定義する．これは，状態がインスタンス変数とその値からなるからである．そして，データの発展関係と状態の発展関係を用いて機能の発展関係を定義する．機能の発展関係では，振舞いの整合性を保存するように定義するため，CLASSICJAVA の意味論が必要となる．本研究では，クラスは機能の集合として扱うので，機能の発展関係を用いてクラスの発展関係を定義する．最後に，クラスの発展関係と状態の発展関係を用いてオブジェクトの発展関係を定義する．

データとオブジェクトを区別することで，このように定義を簡単化することができる．データとオブジェクトを区別しないと，機能の発展関係を定義するため

にオブジェクトの発展関係を必要とする．これは，機能の入出力データがオブジェクトとなるからである．しかし，オブジェクトの発展関係を定義するために機能の発展関係を必要とするので，その定義が非常に複雑になってしまう．データとオブジェクトの区別は，これを防いでいる．

表 3.1: 様々な発展関係

データ	データの具体化関係
	データド メイン
	データド メインの発展関係
	データの具体集合
状態	識別子の具体化関係
	フィールド 環境の発展関係
	ストアの発展関係
機能	メソッド の詳細化関係
	メソッド の直積分割関係
	メソッド の直和分割関係
クラス	メソッド 集合の詳細化関係
	メソッド 集合の直積分割関係
	メソッド 集合の直和分割関係

3.2 CLASSICJAVA

CLASSICJAVA とは，Felleisen らが Mix-in と呼ばれる機能の組み合わせの考え方を Java 上で議論するために形式的に定義した Java の形式化の一つである．その最大の特徴は，機能に焦点をあて，本質を残しつつも非常に簡単化した点である．本研究の焦点も機能であり，非常に相性がいい．したがって，その形式化は扱いやすく，本研究のよい土台である．

CLASSICJAVA のシンタックスは，次のとおりである．

$$
\begin{array}{rcl}
P & = & defn^*\ e \\
defn & = & \textbf{class}\ c\ \textbf{extends}\ c\ \textbf{implements}\ i^*\ \{\ field^*\ meth^*\ \} \\
 & & \mid \textbf{interface}\ i\ \textbf{extends}\ i^*\ \{\ meth^*\ \} \\
field & = & t\ fd \\
meth & = & t\ md\ (\ arg^*\ )\ \{\ body\ \} \\
arg & = & t\ var \\
body & = & e \mid \textbf{abstract} \\
e & = & \textbf{new}\ c \mid var \mid \textbf{null} \mid e\underline{:\ c}.fd \mid e\underline{:\ c}.fd = e \\
 & & \mid e.md\ (e^*) \mid \textbf{super}\ \underline{\equiv \textbf{this}\ :\ c}.md\ (e^*) \\
 & & \mid \textbf{view}\ t\ e \mid \textbf{let}\ var = e\ \textbf{in}\ e \\
var & = & \text{a variable name or } \textbf{this} \\
c & = & \text{a class name or } \textbf{Object} \\
i & = & \text{interface name or } \textbf{Empty} \\
fd & = & \text{a field name} \\
md & = & \text{a method name} \\
t & = & c \mid i
\end{array}
$$

このシンタックスからもそのコンパクトさが分かる．アクセス制御や並行性はない．注目すべき点は，式 e である．メソッドの振舞いは関数として定義する．ここで，**new** はオブジェクトの生成関数，**view** はキャストである．Java では，その振舞いを手続きの列として表現するが，CLASSICJAVA では，**let** を用いて実現する．

CLASSICJAVA では，Java のインスタンス変数に関する副作用を扱うために，ストアと呼ばれる概念を導入している．ストアとは，直観的には，オブジェクト群からなるシステム全体の状態である．形式的には，オブジェクトからフィールド環境とそのオブジェクトのクラスの組への関数である．ここで，フィールド環境とは，インスタンス変数から値への関数である．例えば，あるシステムがオブジェクト obj_1 と obj_2 からなるとする．obj_1 はクラス c_1 のインスタンスで，obj_2 はクラス c_2 のインスタンスとする．obj_1 と obj_2 の現在のフィールド環境がそれぞれ $\mathcal{F}_1$ と $\mathcal{F}_2$ である時，ストア $\mathcal{S}$ は，

$$\mathcal{S}(obj_1) = (c_1, \mathcal{F}_1)\ \text{かつ}\ \mathcal{S}(obj_2) = (c_2, \mathcal{F}_2)$$

である．さらに，obj_1 では，インスタンス変数 x に整数 1 が代入されているとすると，フィールド環境 $\mathcal{F}_1$ は，

$$\mathcal{F}_1(x) = 1$$

である．関数として振舞いを記述し，ストアを持ち回ることによって副作用を実現している．次にその意味論を説明する．

CLASSICJAVA の意味論は，文脈書き換えシステムによって定義されている．プログラムの一部をホール [] で置き換えたものを評価文脈と呼び，E と表す．書き換え規則の一般形は，次のとおりである．

$$P \vdash \langle E[e], \mathcal{S} \rangle \hookrightarrow \langle E[e'], \mathcal{S}' \rangle$$

これは，あるプログラム P において，次に評価する式が e でその時の評価文脈とストアがそれぞれ E と $\mathcal{S}$ である時，式 e' とストア $\mathcal{S}'$ に書き換えるという意味である．$\hookrightarrow$ は書換えの操作を表している．

より具体的に，メソッド呼び出しに関する書き換え規則を説明する．定義された規則は 11 個のみであり，その全ては付録とする．メソッド呼び出しの規則は，次のとおりである．

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

これは，あるプログラム P において，オブジェクト obj のメソッド md を実引数 $v_1, \ldots, v_n$ と共に呼び出すと，そのメソッドの本体である式 e に書き換えることを表している．ここで，自己参照のための変数 $this$ や仮引数 $var_1, \ldots, var_n$ は，それぞれ obj や $v_1, \ldots, v_n$ に置き換える．また，この書換えでは，インスタンス変数の変更はないので，ストア $\mathcal{S}$ は変化しない．$\in_P^c$ は，メソッドやインスタンス変数があるクラスで宣言されているかどうかを表す関係である．CLASSICJAVA では，多くの関係が定義されているが，本研究で使う関係を以下に示す．

- $\leq_P^c$ はサブクラス関係を表す．プログラム P において，クラス c がクラス c' のサブクラスである時，$c \leq_P^c c'$ と記述する．

- $\Subset^c_P$ は宣言されたインスタンス変数やメソッドとクラスの関係を表す．プログラム P において，クラス c で t 型のインスタンス変数 x が直接宣言されている時，$\langle t, x\rangle \Subset^c_P c$ と記述する．また，この関係はオーバーロードされており，$\Subset^c_P$ の左側には，インスタンス変数の宣言とメソッドの宣言のどちらとも記述できる．

- $\in^c_P$ はインスタンス変数やメソッドの宣言がクラスに含まれているかどうかを表す関係である．プログラム P において，クラス c かそのスーパークラスで t 型のインスタンス変数 y が宣言されている時，$\langle t, y\rangle \in^c_P c$ と記述する．この関係も上の関係と同様に，オーバーロードされている．また，$\langle t, x\rangle \Subset^c_P c$ ならば，$\langle t, y\rangle \in^c_P c$ は成立するが，その逆は成立しない．

- $\leq_P$ はサブタイプ関係を表す．プログラム P において，型 t が型 t' のサブタイプである時，$t \leq_P t'$ と記述する．

3.3 データの発展

データとその具体化の関係を表現するためにデータドメインを導入する．データドメインとは，以下のように，データの有限集合 Val とその上の具体化関係 $\prec^{\mathcal{D}}$ の組である．

$$(Val, \prec^{\mathcal{D}})$$

ここで，データ v がデータ $v^{\#}$ を具体化したデータである時，

$$v^{\#} \prec^{\mathcal{D}} v$$

と記述する．本技法では，半順序関係を仮定している．図 3.1 は，整数全体を表す抽象値”Int” が抽象値”Pos”，“Zero”，“Neg” に具体化していることを表している．ここで，“Pos” は正の整数全体を，“Zero” はゼロを，“Neg” は負の整数全体を表している．この場合，“Int” と “Pos” の間の関係は Int $\prec^{\mathcal{D}}$ Pos であり，データドメインは次のとおりである．

$$(\{Int, Pos, Zero, Neg\}, \{(Int, Pos), (Int, Zero), (Int, Neg)\})$$

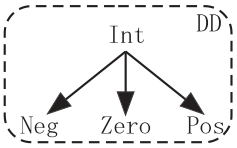

図 3.1: データドメイン

データドメインは，データを具体化することで変化する．図 3.2 はデータドメイン DD から DD' への変化を表している．DD は，"Pos" を $1, 2, 3, \ldots$ に具体化することで DD' に変化している．このようにデータの具体化によってデータドメインを変更することをデータドメインの発展と呼ぶ．我々は，データドメイン $DD^{\#}$ が DD に発展している時,

$$DD^{\#} \sqsubseteq^{\mathcal{D}} DD$$

と記述する．データドメインの発展は，データドメインの有限集合に新しい要素を，データの具体化関係に新しい関係を追加することである．したがって，$\sqsubseteq^{\mathcal{D}}$ で表されるデータドメインの発展関係を次のように定義する．

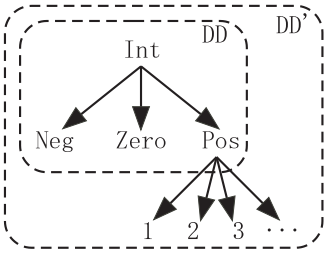

図 3.2: データドメインの発展

定義 1 (データドメインの発展関係)

$DD^{\#}$，DD をデータドメインとする．$DD^{\#} \sqsubseteq^{\mathcal{D}} DD$ と記述するデータドメインの発展関係を次のように定義する．

$$\begin{aligned} DD^{\#} \sqsubseteq^{\mathcal{D}} DD \quad \Longleftrightarrow \quad & data(DD^{\#}) \subseteq data(DD) \\ & \wedge\, rel(DD^{\#}) \subseteq rel(DD) \end{aligned}$$

ここで，データドメイン $DD = (Val, \prec^{\mathcal{D}})$ とすると，$data()$ と $rel()$ は，それぞれデータ集合 Val とその上の具体化関係 $\prec^{\mathcal{D}}$ を表す関数である．

次に，我々は，メソッドの入出力データ集合を表すために，データドメインの具体集合を定義する．データドメインの具体集合とは，データドメインのデータ集合に含まれているデータの中でそれ以上具体化されていないデータの集合である．データドメイン DD の具体集合を $\uparrow DD$ を記述する．図 3.3 はデータドメイン DD とその発展したデータドメイン DD' における具体集合 $\uparrow DD$, $\uparrow DD'$ を表している．

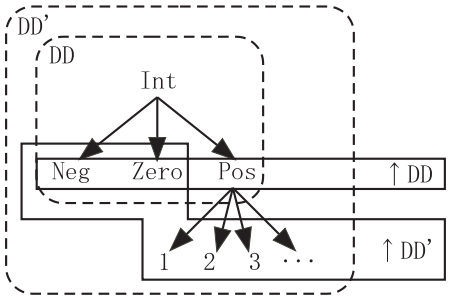

図 3.3: データドメインの具体集合

また，その定義は，次のとおりである．

定義 2 (データドメインの具体集合)

DD をデータドメインとする．$\uparrow DD$ と記述するデータドメインの具体集合を次のように定義する．

$$\uparrow DD \iff \{d \in data(DD) \mid \forall d' \in data(DD).\ d \neq d' \rightarrow (d, d') \notin rel(DD)\}$$

データドメインの持つデータ集合をメソッドの入出力データ集合とすると，メソッドの記述が冗長になる．例えば，上の例で，data(DD) を入力とするメソッド md() と data(DD') を入力とするメソッド md'() を記述したとする．この時，$data(DD) \subseteq data(DD')$ なので，md'() の振舞いは md() の振舞いを含む．したがって，メソッドの記述が冗長になる．この冗長性は，発展したメソッド，例えば md'()，を発展した分だけ記述し，それ以外は抽象化したメソッド，例えば md()，を利用することで解決できる．しかし，この仕組みを実現するためには，実引数を抽象化した時，その値を抽象化したメソッドが扱える必要がある．つまり，実引数とメソッド

をそれぞれ抽象化し実行するとエラーが発生する，あるいは誤った処理が行なわれるという事態は非常に困る．

そこで，本技法では，最も具体的な値，つまり具体集合に含まれる値，を具体化することによるデータドメインの発展を仮定している．言い替えると，すでに具体化した値が存在する値を再び具体化するデータドメインの発展を禁止している．例えば，図 3.2 は，正しいデータドメインの発展である．なぜなら，DD から DD' へのデータドメインの発展は，$\uparrow DD$ に含まれる値 Pos を $1, 2, 3, ...$ に具体化しているからである．逆に，図 3.4 は，誤ったデータドメインの発展である．なぜなら，Pos は $\uparrow DD$ に含まれていないのにも関わらず，DD から DD' へのデータドメインの発展において具体化されているからである．この仮定によって，次の 2 つの性質を保証している．

- $DD \sqsubseteq^{\mathcal{D}} DD'$ が成立するなら，$\uparrow DD'$ に含まれる任意の値 v' について，$v \prec^{\mathcal{D}} v'$ となる値 v が必ず $\uparrow DD$ に含まれている．

- $DD \sqsubseteq^{\mathcal{D}} DD'$ が成立するなら，$\uparrow DD$ に含まれる任意の値 v について，$v \prec^{\mathcal{D}} v'$ となる値 v' が常に $\uparrow DD'$ に含まれている．

したがって，$DD \sqsubseteq^{\mathcal{D}} DD'$ が成立するならば，$\uparrow DD'$ は過不足なく $\uparrow DD$ を発展していることになる．

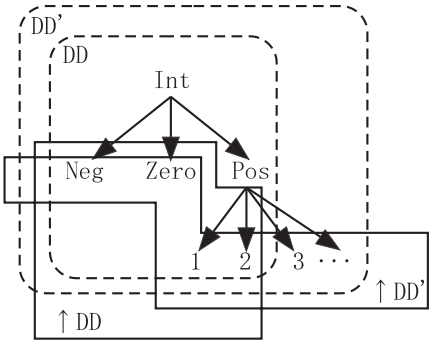

図 3.4: データドメインの間違った発展例

3.4 状態の発展

フィールド環境は，メソッドの詳細化や分割に従って変更する．あるオブジェクトの持つ任意のメソッドを変更しない場合，メソッドの振舞いは変化しないので，そのフィールド環境を変更する必要はない．メソッドを詳細化する場合，より具体化したデータを扱うためにフィールド環境を詳細化する必要があるかもしれない．また，メソッドを分割する場合，分割したメソッドの間でデータの受渡しに使うインスタンス変数を追加する必要があるかもしれない．これらの変更を捉えることのできるフィールド環境の発展を定義する．我々は，フィールド環境 $\mathcal{F}^{\#}$ が $\mathcal{F}$ に発展していることを

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

と記述する．

フィールド環境の発展を形式的に定義するために，識別子の詳細化関係を導入する．フィールド環境は，オブジェクトの持つインスタンス変数とその値からなる．フィールド環境上の発展を議論するためには，値の上の具体化関係だけでなく，識別子上の具体化関係も必要である．我々は，識別子 $x^{\#}$ が x に発展していることを

$$x^{\#} \preceq^{id} x$$

と記述する．

そして，メソッドの詳細化や分割によるフィールド環境の発展を次のように定義する．

定義 3 (フィールド環境の発展関係)

$\mathcal{F}^{\#}$ と $\mathcal{F}$ をフィールド環境とする．$\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$ と記述するフィールド環境の発展関係を次のように定義する．

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

ストアは，オブジェクトの持つフィールド環境からなるので，フィールド環境の発展によって発展する．ストアの発展における重要な条件は，次の 2 つである．

- 発展したストアに含まれる任意のオブジェクトを抽象化したオブジェクトは，必ず抽象化したストアに含まれる．
- 抽象化したストアに含まれる任意のオブジェクトは，その発展したオブジェクトが発展したストアに必ず含まれている．

前者の条件は，発展したストアに (抽象化したストアには含まれていない) オブジェクトを追加することを防ぐ．つまり，抽象実行時にオブジェクトを抽象化できないことを排除するための条件である．この条件を満たさない場合，図 3.5 の (1) のように灰色の部分が発展したストアに含まれることになる．後者の条件は，発展したオブジェクトが発展したストアに含まれていないことを防ぐ．図 3.5 の (2) のように灰色の部分が抽象化したストアに含まれないことを排除するための条件である．抽象実行できるかできないかという点では，(2) のように灰色の部分があっても問題ない．なぜなら，発展したストアに含まれる任意のオブジェクトを抽象化することができるからである．しかし，本技法はトップダウンスタイルのプロトタイプ構成法なので，(2) のようになると全体を見通すことができるという利点が失われる．後者の条件でこれを防いでいる．したがって，これらの条件を満たすストアの発展を次のように定義する．

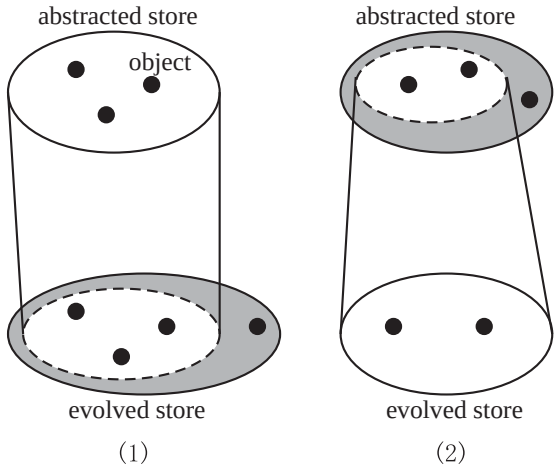

図 3.5: ストアの発展における 2 つの条件

定義 4 (ストアの発展)

$\mathcal{S}^{\#}$ と $\mathcal{S}$ をストアとする．$\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}$ と記述するストアの発展関係を次のように定義する．

$$
\begin{aligned}
& \mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} \\
\Longleftrightarrow \quad & (\forall obj \in dom(\mathcal{S}) \exists obj^{\#} \in dom(\mathcal{S}^{\#}). \\
& \qquad \mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}) \wedge \\
& \qquad (\forall obj^{\#} \in dom(\mathcal{S}^{\#}) \exists obj \in dom(\mathcal{S}). \\
& \qquad \mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}) \\
\text{where} \quad & \mathcal{S}^{\#}(obj^{\#}) = (c^{\#}, \mathcal{F}^{\#}) \; and \\
& \mathcal{S}(obj) = (c, \mathcal{F})
\end{aligned}
$$

ストアの発展は，クラスの発展とフィールド環境の発展を用いて構成するように見えるが，フィールド環境の発展のみで定義している．これは，間違いではない．ストアの発展をクラスの発展と関係付けていない理由は，それぞれの発展が相互に関連し合うことを防ぐためである．クラスの発展は，メソッドの発展から構成され，メソッドの発展は，ストアの発展によって構成される．この時，ストアの発展がクラスの発展によって構成されると，ストアの発展にクラスの発展が必要となり，クラスの発展にストアの発展が必要になる．すると，関係づけが複雑になる．これを防ぐために上の定義ではクラスの発展を用いていない．我々の形式化の手順は，まず，データ上の発展を定義する．次に，オブジェクトのフィールド環境上の発展をデータ上の発展とインスタンス変数上の発展を用いて定義する．フィールド環境上の発展を用いて，メソッドの発展を定義する．メソッドの発展を用いてクラスの発展を定義する．クラスの発展とフィールド環境の発展を用いてオブジェクトの発展を定義する．

3.5 機能の発展

機能発展には，メソッドの詳細化，直積分割，直和分割の 3 つがある．ここでは，それらを定義する．

メソッドの詳細化とは，より具体化したデータを扱えるようにメソッドを変更することである．我々は，メソッドの詳細化によってメソッド $md^{\#}$ を md に変更

することを

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

と記述する．我々は，この $\sqsubseteq^{\mathcal{M}}$ をメソッドの詳細化関係と呼ぶ．この関係は，抽象実行のために使う．$md^{\#}$ の実行が md の実行の抽象解釈となるように，つまり図 3.6 のように，データドメインとストアの発展を用いて定義する．ここで，破線の矢印は抽象化を表現している．その定義は次のとおりである．

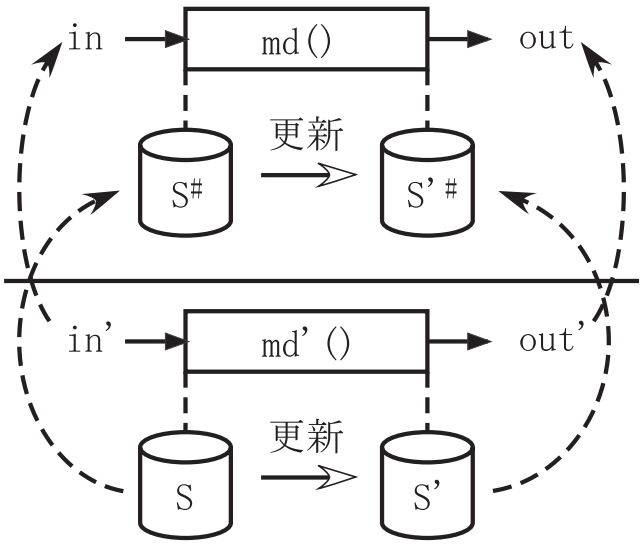

図 3.6: メソッドの詳細化

定義 5 (メソッドの詳細化)

$DD^{\#}_{in}, DD^{\#}_{out}, DD_{in}, DD_{out}$ をデータドメイン，SS をシステムが取りうるストアの集合とする．$md^{\#} \sqsubseteq^{\mathcal{M}} md$ と記述するメソッドの詳細化関係を次のように定義

する．

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

ここで，$md^{\#} : \uparrow DD^{\#}_{in} \rightarrow \uparrow DD^{\#}_{out}$ はオブジェクト $obj^{\#}$ の持つメソッド，$md : \uparrow DD_{in} \rightarrow \uparrow DD_{out}$ はオブジェクト obj の持つメソッドである．また，$\overset{*}{\hookrightarrow}$ は，CLASSICJAVA の書き換え規則を繰り返し適用することを表す．

メソッドの直積分割とは，メソッドを 2 つのメソッドに分割することである．ここで，その分割したメソッドの列が分割前のメソッドの詳細化となる．我々は，メソッドの実行列 $md_1; md_2$ がメソッド $md^{\#}$ の詳細化である時，

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

と記述する．我々は，メソッドの直積分割をメソッドの詳細化と同じように定義する．図 3.7 はメソッドの直積分割を表している．しかし，md_1 を実行した直後に他のメソッドを実行することなく md_2 を実行することを仮定している．その定義は以下のとおりである．

定義 6 (メソッドの直積分割)

$DD^{\#}_{in}, DD^{\#}_{out}, DD_{in}, DD_{out}$ をデータドメイン，SS をシステムが取りうるストアの集合とする．$md^{\#} \sqsubseteq^{\mathcal{M}}_{\times} md_1; md_2$ と記述するメソッドの直積分割関係を次のよう

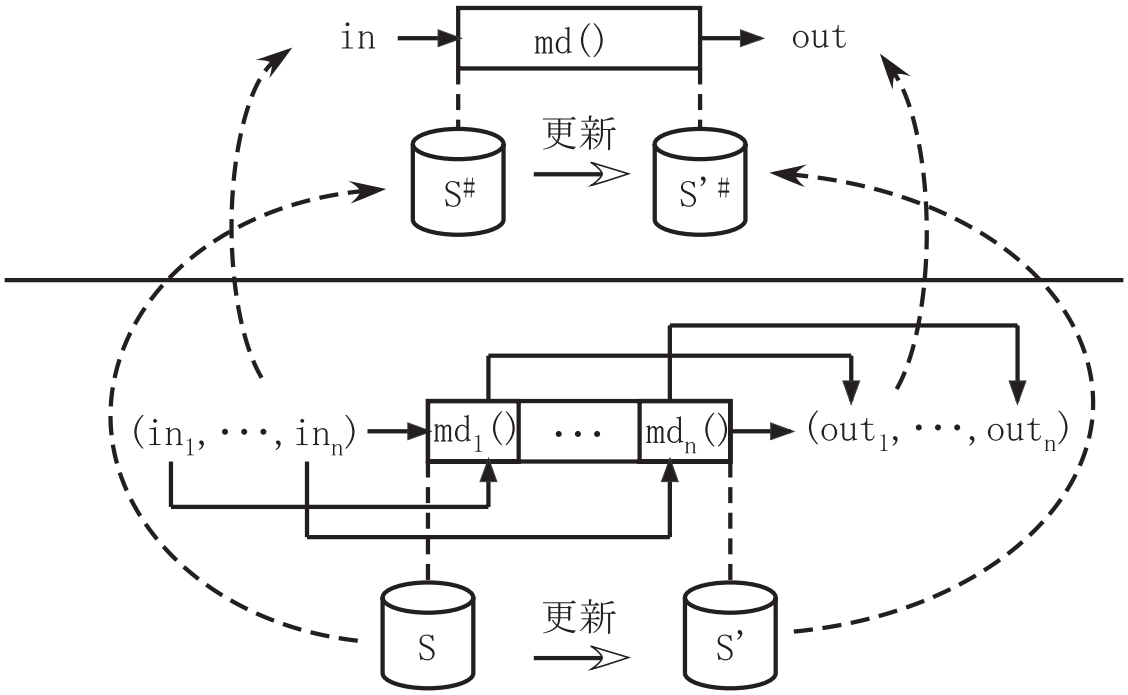

図 3.7: メソッドの直積分割

に定義する.

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

ここで，$md^{\#} : \uparrow DD^{\#}_{in} \rightarrow \uparrow DD^{\#}_{out}$ はオブジェクト $obj^{\#}$ が持つメソッドで，$md_1 : D_1 \rightarrow D'_1$ と $md_2 : D_2 \rightarrow D'_2$ はオブジェクト obj が持つメソッドである．さらに，$\uparrow DD_{in} = D_1 \times D_2$, $\uparrow DD_{out} = D'_1 \times D'_2$ である．また，$\overset{*}{\hookrightarrow}$ は，CLASSICJAVA の書き換え規則を繰り返し適用することを表す.

メソッドの直和分割とは，メソッドを 2 つのメソッドに分割することである．ただし，メソッドの直積分割とは違い，分割したメソッドの選択が分割前のメソッドの詳細化となる．我々は，メソッドの選択 $md_1|md_2$ が $md^{\#}$ の詳細化である時,

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

と記述する．我々は，メソッドの直和分割をメソッドの詳細化と同じように定義する．図 3.8 は直観的なメソッドの直和分割を表している．その定義は次のとおりである．

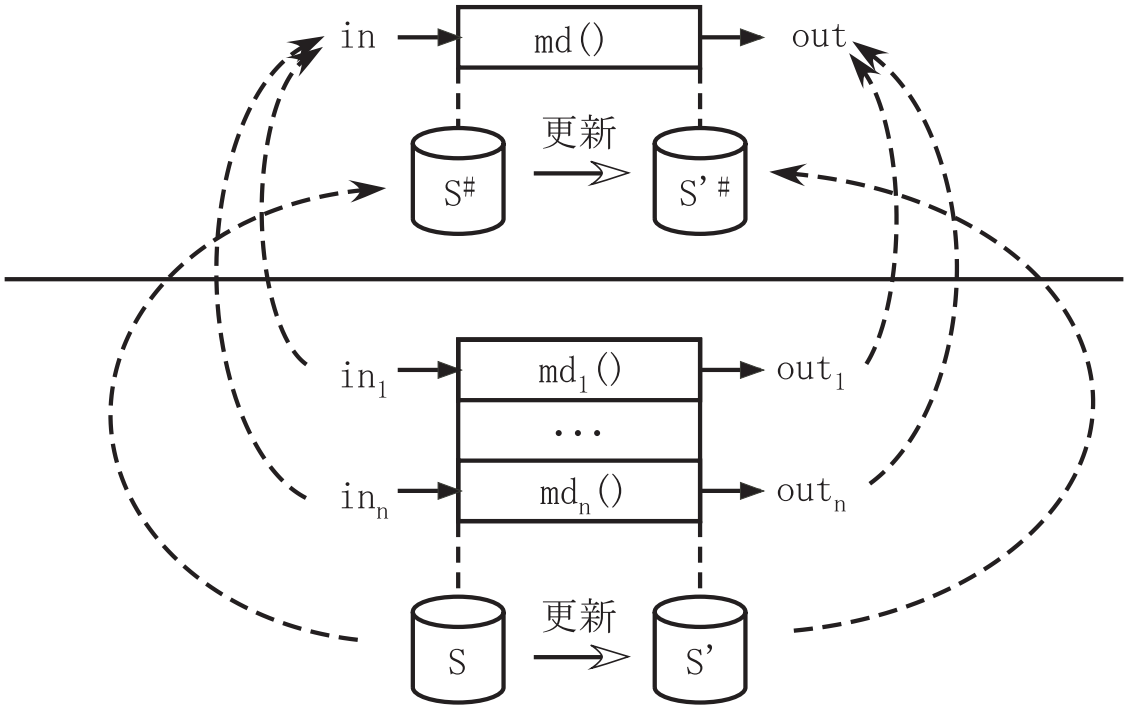

図 3.8: メソッドの直和分割

定義 7 (メソッドの直和分割関係)

$DD_{in}^{\#}, DD_{out}^{\#}, DD_{in}, DD_{out}$ をデータドメイン，SS をシステムが取りうるストアの集合とする．$md^{\#} \sqsubseteq_{+}^{\mathcal{M}} md_1|md_2$ と記述するメソッド直和分割関係を次のように

定義する．

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

ここで，$md^{\#} : \uparrow DD_{in}^{\#} \rightarrow \uparrow DD_{out}^{\#}$ はオブジェクト $obj^{\#}$ が持つメソッドで，$md_1 : D_1 \rightarrow D_1'$ と $md_2 : D_2 \rightarrow D_2'$ はオブジェクト obj が持つメソッドである．さらに，$\uparrow DD_{in} = D_1 \cup D_2, D_1 \cap D_2 = \emptyset$, $\uparrow DD_{out} = D_1' \cup D_2', D_1' \cap D_2' = \emptyset$ である．また，$\overset{*}{\hookrightarrow}$ は，CLASSICJAVA の書き換え規則を繰り返し適用することを表す．

3.6 クラスの発展

クラスの発展とは，クラスの機能を詳細化することである．クラスとは，機能の集まりで，機能はメソッドとインスタンス変数からなる．基本的にクラスはカプセル化されているので，インスタンス変数を他のオブジェクトが直接アクセスすることはできない．他のオブジェクトの視点からクラスの機能の詳細化はメソッドの詳細化と分割である．したがって，我々は，クラスの発展をメソッドの詳細化と分割によって関係付ける．我々は，クラス $c^{\#}$ をクラス c に発展していることを

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

と記述する．クラスをメソッドの集合とみなし，c は，$c^{\#}$ のメソッドを詳細化するか直積分割するか直和分割することによって発展したメソッドの集合とみる．これを一度に定義すると定義が複雑になるため，メソッド集合の詳細化関係，メソッド集合の直積分割関係，メソッド集合の直和分割関係に分けて定義する．これら

は，それぞれ，メソッドの詳細化，直積分割，直和分割によるメソッド集合の発展を表している．それぞれの定義は次のとおりである．

定義 8 (メソッド集合の詳細化関係)

$Meth^{\#}$ と $Meth$ をメソッドの集合とする．$Meth^{\#} \sqsubseteq^{MS} Meth$ と記述するメソッドの詳細化関係を次のように定義する．

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

定義 9 (メソッド集合の直積分割関係)

$Meth^{\#}$ と $Meth$ をメソッドの集合とする．$Meth^{\#} \sqsubseteq_{\times}^{MS} Meth$ と記述するメソッドの直積分割関係を次のように定義する．

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

定義 10 (メソッド集合の直和分割関係)

$Meth^{\#}$ と $Meth$ をメソッドの集合とする．$Meth^{\#} \sqsubseteq_{+}^{MS} Meth$ と記述するメソッド集合の直和分割関係を次のように定義する．

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

これらのメソッド集合上の関係を用いて，次のようにクラスの発展関係を定義する．

定義 11 (クラスの発展関係)

$c^{\#}$ と c をクラス，$Meth^{\#}$ と $Meth$ をそれぞれ $c^{\#}$ と c のメソッド集合とする．$c^{\#} \sqsubseteq^{\mathcal{C}} c$ と記述するクラスの発展関係を次のように定義する．

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

3.7 オブジェクトの発展

オブジェクトの発展とは，オブジェクト $obj^{\#}$ の実行がオブジェクト obj の実行の抽象実行となるように $obj^{\#}$ を obj に変更することである．オブジェクトは副作用を含むため，常に発展関係が成立するとは限らない．つまり，$obj^{\#}$ のクラスを $c^{\#}$，obj のクラスを c とした時，$c^{\#} \sqsubseteq^{\mathcal{C}} c$ であっても，常に $obj^{\#}$ の実行が obj の抽象実行となるとは限らない．なぜなら，オブジェクトの振舞いはそのフィールド環境に依存するからである．$obj^{\#}$ の現在のフィールド環境が $\mathcal{F}^{\#}$ で obj の現在のフィールド環境が $\mathcal{F}$ であり，$\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$ という条件を加えることで，$obj^{\#}$ の実行が obj の抽象実行となる．ここで，オブジェクト obj の現在のフィールド環境が $\mathcal{F}$ であることを

$$obj@\mathcal{F}$$

と書くことにし，$obj^{\#}@\mathcal{F}^{\#}$ が $obj@\mathcal{F}$ に発展していることを

$$obj^{\#}@\mathcal{F}^{\#} \sqsubseteq^{\mathcal{O}} obj@\mathcal{F}$$

と書くことにする．そして，このオブジェクトの発展関係を次のように定義する．

定義 12 (オブジェクトの発展関係)

$obj^{\#}$ と obj をオブジェクト，$\mathcal{F}^{\#}$ と $\mathcal{F}$ をフィールド環境とする．$obj^{\#}@\mathcal{F}^{\#} \sqsubseteq^{\mathcal{O}} obj@\mathcal{F}$ と記述するオブジェクトの発展関係 $\sqsubseteq^{\mathcal{O}}$ を次のように定義する．

$$\begin{aligned} & obj^{\#}@\mathcal{F}^{\#} \sqsubseteq^{\mathcal{O}} obj@\mathcal{F} \\ \Longleftrightarrow \quad & c^{\#} \sqsubseteq^{\mathcal{C}} c \wedge \mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} \end{aligned}$$

ここで，$c^{\#}$ は $obj^{\#}$ のクラス，c は obj のクラスとする．

第 4章

抽象実行メカニズムの実現

本手法は，原理的には，抽象実行メカニズムによってプロトタイピングの途中段階における全体としての実行を可能にするが，Java 上で実現する時、3 つの問題が発生する．本節では，それらの問題を詳しく述べた後，それを解決する仲介オブジェクトとその機械的生成について述べる．

4.1 問題点

一つ目の問題点は，Java において，あるオブジェクトを異なるインターフェースを持つオブジェクトに置換することを認めていないことである．例えば，前の例での `Rectangle@3` のオブジェクトから `Rectangle@2` のオブジェクトへの置換である．本技法において，抽象実行メカニズムは，プロトタイプの実行を継続するためにオブジェクト，メソッド，データなどを実行時に動的に変換するために使う．その時，変換前後のオブジェクトを同一的に扱えることが望ましい．しかしながら，Java では，型上の制約があり素朴に実現することができないため，発展したオブジェクト群を同一的に扱うためのなんらかの仕組みが必要である．

二つ目の問題は，抽象実行メカニズムによってあるオブジェクトを異なるオブジェクトへ変換した時のオブジェクトの参照を管理することが難しいことである．例えば，あるオブジェクト $caller_1$ と $caller_2$ が共にオブジェクト $callee$ を参照しているとする．この状況で，抽象実行メカニズムによって $callee$ が $callee^{\#}$ に変換された時，$caller_1$ と $caller_2$ は共に $callee^{\#}$ を参照するように再構築されるべきであ

る．しかしながら，Java の言語機能として，$callee$ は，どのオブジェクトが自分自身を参照しているのか知ることは難しく，参照の再構築には特別な仕組みを必要とする．この問題は，インターフェースが変化しなくても発生するが，Java において，異なるオブジェクトを同一的に扱うことが難しいという意味で一つ目の問題点に似ている．

三つ目の問題は，これまでの問題とは少し異なり，開発するプロトタイプ毎に抽象実行メカニズムをプログラマが実現しなければならないというコストを本技法が生み出すことである．オブジェクトは発展のコースの逆方向にそって変換される．一般的に，プロトタイプ毎にオブジェクトの発展は異なるので，プログラマは，プロトタイプ毎の発展のコースにそった変換を実現しなければならない．高品質ソフトウェアを開発するためには，多くのプロトタイプを作成する必要があるので，このコストを減らすことは，大変重要なことである．

素朴に，発展したクラスを子クラスとするようにクラス継承を用いることでこれらの問題を解決できるように見える．例えば，前の例の `Rectangle@3` クラスを `Rectangle@2` クラスの子クラスとすることである．残念ながら，この方法では，全ての問題を解決することは難しい．その理由を以下で述べる．

このクラス継承を用いた実装では，継承関係により，発展関係にあるオブジェクト群を同一的に扱うことが可能となる．つまり，より発展したクラスのオブジェクト (例えば，`Rectangle@3` クラスのオブジェクト) を発展前のクラスのオブジェクト (例えば，`Rectangle@2` クラスのオブジェクト) と置き換えることが Java の言語機能上，可能となる．したがって，一つ目の問題点を解決する．さらに，継承関係によって，より発展したクラスは，発展前のインスタンス変数やメソッドを持つため，抽象実行時にあえて発展前のオブジェクトへ変換する必要がなくなる．オブジェクトの変換を必要としないこの方法では，オブジェクトの参照を管理する必要がなくなるため，二つ目の問題も解決する．

しかし，この方法では，オブジェクトの変換を必要としないが，抽象実行メカニズムを用いるためには，インスタンス変数に関する整合性を保たなければならず，三つ目の問題点が解決されない．Java では，メソッド呼び出し時にインスタンス変数に関する副作用が生じるので，その副作用をすべての親クラス (発展前のクラス) で宣言されているインスタンス変数に反映しなければ，抽象実行時に正しくな

い結果を得ることになる．例えば，Rectangle@5 クラスのオブジェクトがまず生成され，setX() と setY() メソッドによって，座標 (-1,-1) から座標 (2,2) に移動した (つまり，インスタンス変数 x と y にそれぞれ 2 が代入される) とする．その時，なんらかの理由で，そのオブジェクトを Rectangle@4 クラスのオブジェクトへ縮退する必要が生じた時，クラス継承を用いた実行では，オブジェクトの変換をすることなく，Rectangle@4 クラスのオブジェクトとして扱うことができる．しかし，現在の座標が変更したことを (Rectangle@4 で宣言されている) インスタンス変数 x と y を反映させなければ (つまり，x と y にそれぞれ正の整数を抽象化した値 pos を代入する)，たとえ Rectangle@4 クラスのオブジェクトとして扱ったとしても正しい結果を得ることはできない．この整合性を保つためのコードはプロトタイプ毎に異なるため，三つ目の問題点は解決されない．

4.2 仲介オブジェクト

これらの問題を解決するために，我々は**仲介オブジェクト**を導入する．仲介オブジェクトとは，発展した全てのオブジェクト (例えば，Rectangle@1 から Rectangle@5 までのオブジェクト) をラップし，caller オブジェクトと callee オブジェクトを間接的につなぐオブジェクトである（図 4.1）．仲介オブジェクトは，以下の 2 つの役割を持つ．

1. callee オブジェクトの縮退の管理
2. メソッド呼び出しの仲介

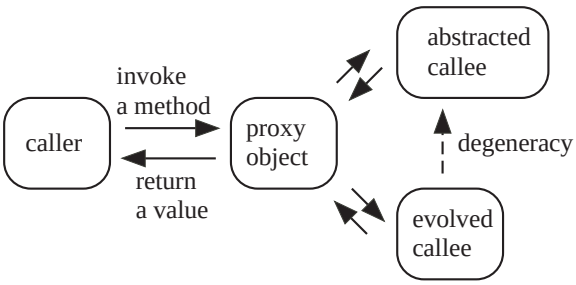

図 4.1: 仲介オブジェクト

仲介オブジェクトに全ての発展したオブジェクトのインターフェースを持たせ，callee オブジェクトを直接参照しないことによって，一つ目の問題点を解決している．caller オブジェクトが参照しているオブジェクトは仲介オブジェクトなので，たとえ callee オブジェクトが縮退によって異なる (発展前の) オブジェクトに変換されたとしても，それに合わせて caller オブジェクトを変化させる必要はない．したがって，どんな発展度を持つ callee オブジェクトであっても，間接的に参照できるので，異なる発展度を持つ全てのオブジェクトを同一的に扱うことができる．さらに，caller オブジェクトの持つ callee オブジェクトへの参照を再構成する必要がなくなるため，二つ目の問題点も解決する．

仲介オブジェクトがメソッド呼び出しの仲介を行なうことが caller オブジェクトや callee オブジェクトから抽象実行に必要なコードの分離を可能にする．これが仲介オブジェクトの機械的な生成を可能とし，三つ目の問題点を解決する．この機械的な生成の詳細は次の節で詳しく述べる．本技法において，縮退はメソッド呼び出し時に発生する．仲介オブジェクトは，発展した全てのオブジェクトへのメソッド呼び出しを全て仲介するので，仲介オブジェクトが縮退を管理することが可能となる．これによって，caller オブジェクトに縮退に関するコードを記述する必要がなくなる．さらに，仲介オブジェクトが callee オブジェクトの縮退を管理することで，callee オブジェクトは自分自身の縮退の管理から解放される．したがって，仲介オブジェクトの構造が本質的なコードと抽象実行に必要なコードとの分離を可能とする．

より具体的に，以下の dispatch() アルゴリズムは，仲介オブジェクトの機能である縮退をともなうメソッド呼び出しの仲介を表している．

```
Object dispatch (Method m, Object[] args) {
    int n = canInvoke (callee, m, args);
    switch (n) {
    case POSSIBLE:
        return m.invoke (callee, args);
    case CALLER_DEGEN:
        throw new NeedDegeneracyException ();
```

```
        case CALLEE_OBJ_DEGEN:
            degeneracy (callee);
            return this.dispatch (m, args);
        case CALLEE_METH_DEGEN:
            Method m' = degeneracyMethod (m);
            Object[] args' = degeneracyData (args);
            return m'.invoke (callee, args');
        }
}
```

このアルゴリズムは，引数としてメソッド m とその引数 $args$ を取り，それぞれ caller オブジェクトが呼び出したメソッドとその引数を表している．このアルゴリズムでは，まず，$callee.m(args)$ が呼び出し可能かどうかを調べる．ここで，その検査を $canInvoke()$ と表している．また，$callee$ は，仲介オブジェクトが現在参照している callee オブジェクトを表している．$canInvoke()$ 関数は，(1) 縮退を必要としないことを表す値 POSSIBLE，(2)caller オブジェクトの縮退が必要であることを表す値 CALLER_DEGEN，(3)callee オブジェクトにおけるオブジェクトの縮退が必要であることを表す値 CALLEE_OBJ_DEGEN，(4)callee オブジェクトにおけるメソッドの縮退が必要であることを表す値 CALLEE_METH_DEGEN のいずれかを返す．

そして，$canInvoke()$ 関数の返す値によって次のように振舞いが変化する．POSSIBLE の場合，縮退する必要がないので，そのままメソッド呼び出しを行なう．CALLER_DEGEN の場合，callee オブジェクトが抽象的すぎてメソッド呼び出しを行ないことを表しているので，caller オブジェクトを管理する仲介オブジェクトに縮退が必要であることを知らせる例外 NeedDegeneracyException を発生させる[1]．CALLEE_OBJ_DEGEN の場合，まず，callee オブジェクトの縮退を行なう．その縮退を $degeneracy()$ として表している．そして，再帰的にこのアルゴリズムを実行する．この再帰的な呼び出しは，caller オブジェクトと callee オブジェクトの発展度が同じになった時，つまり POSSIBLE の場合，のメソッド呼び出しに対す

[1]dispatch() アルゴリズムでは，この例外に対する例外処理を省略しているが，CALLEE_OBJ_DEGEN の場合と同様の処理を行なうだけである．

る返り値を最終的な値として caller オブジェクトに返すことを実現する．最後に，CALLEE_METH_DEGEN の場合，メソッドの縮退を行ない，再びメソッド呼び出しを行なう．ここで，メソッド m は m' に変換され，その実引数 $args$ は $args'$ に変換される．これらの変換を行なうのが，それぞれ degeneracyMethod() と degeneracyData() である．

4.3 仲介オブジェクトの機械的な生成

どんな仲介オブジェクトも抽象実行による実行の継続は，先に示した dispatch() アルゴリズムによって実現される．このことから，リフレクション技術を用いることで実行時の機械的な仲介オブジェクトの生成が可能となる．我々が用いるリフレクション技術は，Dynamic Proxy Class API である．この API はメソッド起動を仲介するオブジェクトを実行時に生成するために使われ，JDK1.3 から標準クラスライブラリに導入されている．

dispatch() アルゴリズムには $canInvoke()$ 関数や $degeneracy()$ 関数のように発展関係に依存する手続きがあるが，我々は，その発展関係を XML 文書とすることで，発展関係に依存しないアルゴリズムを実現する．発展関係に依存する部分を XML 文書としてコードから分離することでアルゴリズムを共通化する．重要なことは，XML 文書として扱うことではなく，コードから分離することである．

この分離は，dispatch() アルゴリズムを全ての仲介オブジェクトに対して共通化し，機械的な生成を可能にする．より具体的に，このアルゴリズムは `InvocationHandler` インターフェースの `invoke()` メソッドとして実装されている．Dynamic proxy を実行時に生成するためには，`InvocationHandler` インターフェースを実装するクラスが必要である．この実装によって，実行時に仲介オブジェクトを機械的に生成することが可能となり，プログラマは抽象実行に必要なコードを記述することなくプロトタイプを作成することができる．

仲介オブジェクトの機械的生成のアイデアは，Yellin らが提案するソフトウェアアダプタの自動生成と同じである [12]．[12] では，インターフェースの互換性を定義し，interface mapping と呼ばれるメッセージや状態の対応関係を与えることでソフトウェアアダプタの自動生成を可能にしている．本技法では，機能発展がイン

ターフェースの互換性を保ち，XML 文書で記述した発展関係が interface mapping の役割をする．

4.4 プログラミング環境

我々は，仲介オブジェクトを機械的に生成するためのクラスライブラリだけでなく，データの具体化や機能発展を支援する発展エディタや縮退による実行をトレースするためのビジュアライザを含むプログラミング環境を構築した．図 4.2 は，そのプログラミング環境の全体像を表している．

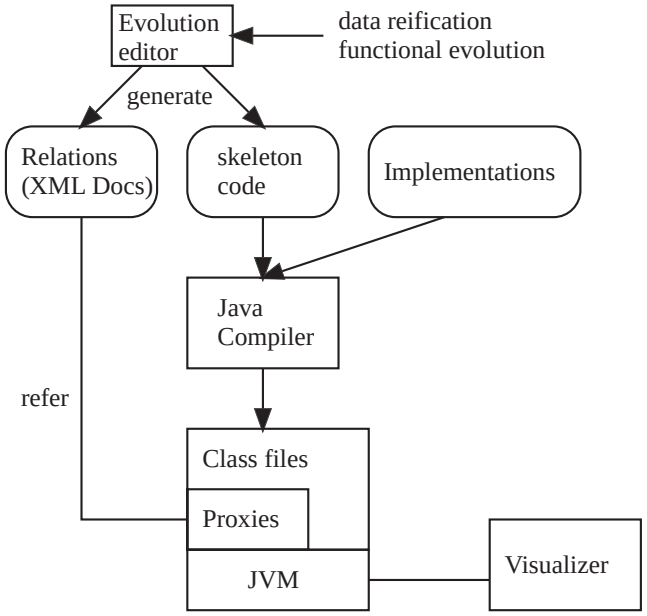

図 4.2: プログラミング環境の全体像

発展エディタ (図 4.3) は，データの具体化や機能発展といった発展関係の記述を支援し，それらの関係をグラフィカルに表示する機能を持つ．さらに，入力された発展関係から，抽象実行時に使われる XML 文書やクラスの雛型といったスケルトンコードを生成する．プログラマは，そのスケルトンコードを元にメソッドの実装を行なう．

我々は抽象実行メカニズムをクラスライブラリとして提供しているため，特別なコンパイラや処理系を必要としない．プログラマが作成したソースコードをコ

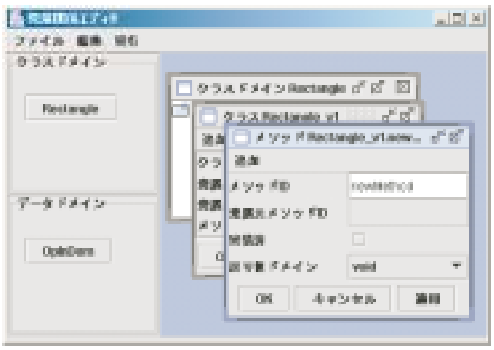

図 4.3: 発展エディタのスナップショット

ンパイルするためには標準的な Java コンパイラを使い，その実行には JVM を使う．処理系を選択する際に問題になるのは，Dynamic Proxy Class API が使えるかどうかだけである．

抽象実行メカニズムを実装したクラスライブラリには，dispatch() アルゴリズムを実装した `invoke()` メソッドだけでなく，仲介オブジェクトを実体化するコンストラクタを含んでいる．このコンストラクタが仲介オブジェクトを機械的に生成する．コンストラクタでは，仲介オブジェクトを実体化し，引数として与えられた文字列に基づいて最も発展している callee オブジェクトを仲介オブジェクトの内部に実体化する．例えば，`Rectangle` オブジェクトの実体化は，`Rectangle()` ではなく，

```
(Rectangle)ProxyFactory.createProxy("Rectangle")
```

と記述する．このコンストラクタは，仲介オブジェクトを実体化した後，その引数を元に XML 文書から最も発展しているクラスの名前を探し，callee オブジェクトを実体化する．

抽象実行をともなう実行を視覚化するビジュアライザは，そのトレースを UML のシーケンス図として表示する (図 4.4)．シーケンス図は，オブジェクト間の相互作用を時間的な流れに沿って表現することに適しているので，どのタイミングで縮退が起きてプログラムの実行がどのように変化したかを見るのに適している．プロトタイプの発展度によって抽象実行が起きるタイミングが変化するため，プログラマにとってデバッグしにくいという欠点を持つ．ビジュアライザは，このデ

バッグを支援する．シーケンス図を表すためには，メソッド起動やその戻り値に関する情報を実行時に獲得する必要がある．我々は，Java Debugger Interface (JDI) を用いることで，これらの情報を獲得している．JDI は，JVM 上で動作しているアプリケーションの実行の制御やオブジェクトの状態の取得などの機能を持つ．

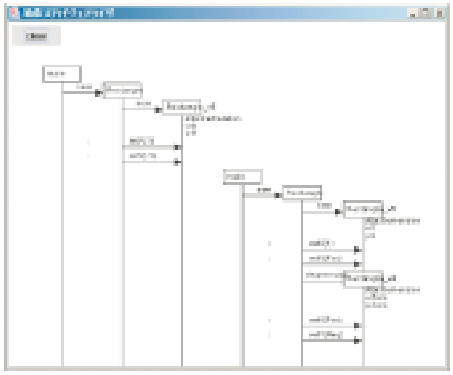

図 4.4: ビジュアライザのスナップショット

第 5 章

ソフトウェア開発事例

本章では，本技法を用いたソフトウェア開発の事例を示す．本論文で示す開発事例は，ブラックジャックシステムと商品在庫システムの 2 つである．

5.1 ブラックジャック

5.1.1 概要

ブラックジャックとは，カードゲームの一つで，プレイヤーとディーラーが手持ちのカードの数字の合計が「21」を超えない範囲で「21」に近づけるゲームである．図 5.1 は，ブラックジャックの全体像を表している．本システムは，プレイヤー，ディーラー，カードが置いてあるカードスタックオブジェクトからなる．プレイヤーが掛け金であるコインを受け取ることでゲームが始まる．次に，プレイヤーはカードスタックからカードを 2 枚引く．そして，ディーラーと勝負し，掛け金や勝敗に応じたコインを受け取る．ブラックジャックは，プレイヤーとディーラーの一対一の勝負であり，プレイヤーの勝ちはディーラーの負けである．本事例で用いるブラックジャックのルールは次のとおりである．

- プレイヤーは，ディーラーに勝てば掛け金の 2 倍を受け取り，引き分けなら掛け金をそのまま受け取り，負けると掛け金が没収される．
- カードの数え方は，2 から 10 まではそのまま，J，Q，K は 10 である．A は特別で，1 または 11 のどちらか都合のよいほうを選択できる．

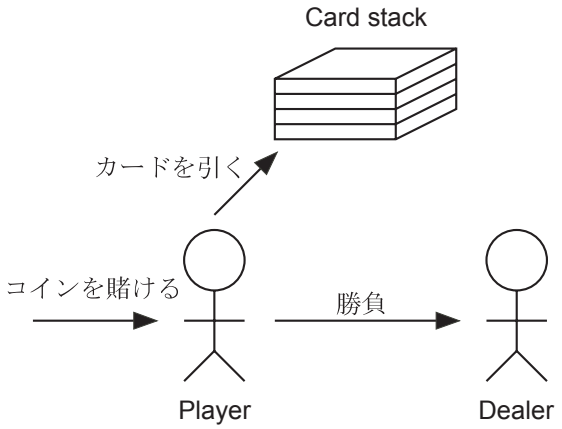

図 5.1: ブラックジャック

- プレイヤーは，カードスタックから 2 枚のカードを引く[1].
- プレイヤーが勝つのは，以下の場合である.
  1. プレイヤーのカードの合計がディーラーのカードの合計を超えた場合
  2. プレイヤーのカードが A と 10，J，Q，K のいずれかの組み合わせ (例えば，A と 10 の組み合わせや A と K の組み合わせ) の場合．この場合を BLACKJACK と呼ぶ[2].
  3. ディーラーのカードの合計が 22 を超えた場合．この場合を BURST と呼ぶ.
- プレイヤーのカードの合計とディーラーのカードの合計が同じ場合，引き分けである.
- ディーラーが勝つのは，プレイヤーが BLACKJACK でなくかつディーラーのカードの合計がプレイヤーのカードの合計を超えた場合である[3].

[1]実際のルールでは，カードの数字の合計が 21 を超えない限り，プレイヤーは何枚でもカードを引くことができるが，簡単化のため，本事例では 2 枚だけカードを引くこととする.

[2]実際は，ディーラーも BLACKJACK の場合，引き分けであるが，本事例では，簡単化のため，プレイヤーが BLACKJACK の場合，無条件でプレイヤーの勝ちとする.

[3]本事例では，プレイヤーは 2 枚しかカードを引かないため，プレイヤーのカードの合計が 21 を超えることはない.

- ディーラーは，手持ちのカードの合計が 17 以上になるまでカードを引き続けなければならない．つまり，ディーラーのカードの合計は，17，18，19，20，21，22 以上のいずれかである．

5.1.2　仕様の抽象化

ここでは，ブラックジャックシステムを各オブジェクトの機能に着目して抽象化する．本システムは，次の 3 つのオブジェクトからなる．

- プレイヤー
- ディーラー
- カードスタック

まず，それぞれのオブジェクトの持つ機能とその入出力を洗い出し，整理する．プレイヤーは，掛け金を入力としてゲームを行なう機能からなる．出力は，ゲームの結果のコインである．ディーラーは，プレイヤーのカードの合計と自分のカードの合計を比較し，勝ち負けを決める機能からなる．その入出力は，それぞれ，プレイヤーのカードの合計と勝敗である．カードスタックは，カード要求に対してカードを渡す機能からなる．入力はなく，出力はカードである．これらを整理すると，本システムで用いるデータは，

- コイン
- カード
- カードの合計
- 勝敗の結果

の 4 つがあり，それぞれのオブジェクトの機能は，

- プレイヤー: ブラックジャックゲームを一回する機能
- ディーラー: それぞれのカードの合計を比較し，勝負を判定する機能

- カード スタック: カードを要求されるとカードを出す機能

である．

次に，これらのデータや機能を抽象化する．各データを次のように抽象化する．

- すべてのコインを抽象化した値”coins”
- すべてのカードを抽象化した値”card”
- すべてのカードの合計を抽象化した値”cards”
- 勝敗を抽象化した値”result”

そして，それぞれの機能を次のように抽象化する．

- プレイヤーの機能を表すメソッド *play_v1()*．このメソッドは，“coins” を受け取り，“coins” を返す．
- ディーラーの機能を表すメソッド *bet_v1()*．このメソッドは，(プレイヤーの)“cards” を受け取り，自分のカードの合計と比較し，“result” を返す．
- カード スタックの機能を表すメソッド *getCard_v1()*．このメソッドは，なにも入力がなく，“card” を返す．

この抽象化の結果，図 5.2 に示すクラスを最も抽象化したクラスとする．

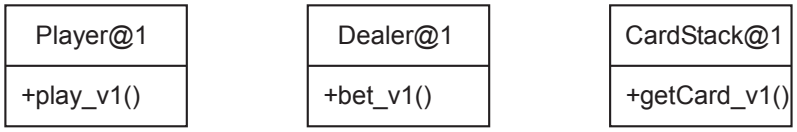

図 5.2: 最も抽象化したブラックジャックシステム

5.1.3 原始プロトタイプの作成

次に，最も抽象化したブラックジャックシステムに基づいて原始プロトタイプを構築する．以下のプログラムは，ブラックジャックシステムの原始プロトタイプを構成するクラス Player@1，Dealer@1，CardStack@1 である．

```
/* version 1 */
class Player {
    CardStack cs = (CardStack)ProxyFactory.createProxy ("CardStack");
    Dealer d = (Dealer)ProxyFactory.createProxy ("Dealer");
    CoinDom_v1 play_v1 (CoinDom_v1 in) {
        CardDom_v1 c1 = cs.getCard_v1 ();
        CardDom_v1 c2 = cs.getCard_v1 ();
        ResultDom_v1 r = d.bet_v1 (new CardsDom_v1 (c1, c2));
        return new CoinDom_v1 ("coins");
    }
}
class Dealer {
    CardsDom_v1 cards = new CardsDom_v1 ("cards");
    ResultDom_v1 bet_v1 (CardsDom_v1 p) {
        return new ResultDom_v1 ("result");
    }
}
class CardStack {
    CardDom_v1 getCard_v1 () {
        return new CardDom_v1 ("card");
    }
}
```

Dealer@1 クラスでは，オブジェクトの生成時にディーラーのカードの合計をインスタンス変数 `cards` に格納する．本来は，カードスタックからカードを引くべきだが，簡単化している．そして，`bet_v1()` メソッドでは，プレイヤーのカードの合計を仮引数 `p` として受け取り，勝敗の結果全体を表す値” result” を返す.

CardStack@1 クラスは，インスタンス変数がなく，`getCard_v1()` メソッドからなる．`getCard_v1()` メソッドは，入力がなく，カード全体を表す値” card” を返す.

Player@1 クラスは，Dealer@1 クラスのオブジェクトと CardStack@1 クラスのオブジェクトをそれぞれ格納するインスタンス変数 `d` と `cs` とメソッド `play_v1()` からなる．`play_v1()` は，掛け金を仮引数 `in` として受け取る．次に，カードスタックからカードを 2 枚引く．そして，それらのカードとともにディーラーの `bet_v1()` メソッドを呼び出す．最後に獲得したコインを返す.

5.1.4 プロトタイプの発展

次に，2回の発展によって原始プロトタイプを発展させる．1回目の発展で，勝敗が分かるようにし，2回目の発展で，すべてを詳細にする．図5.3は，本システムの発展の過程を示している．本システムの発展は図5.4に示すデータの具体化に基づいている．

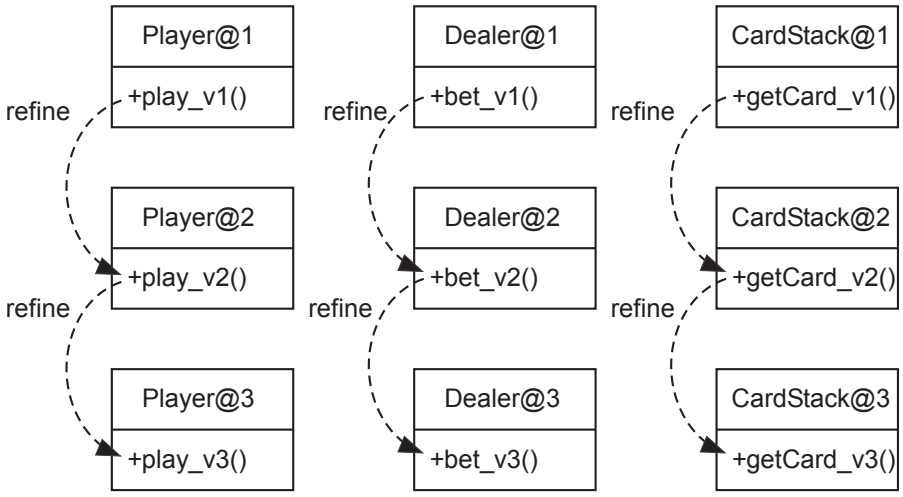

図 5.3: ブラックジャックシステムの発展

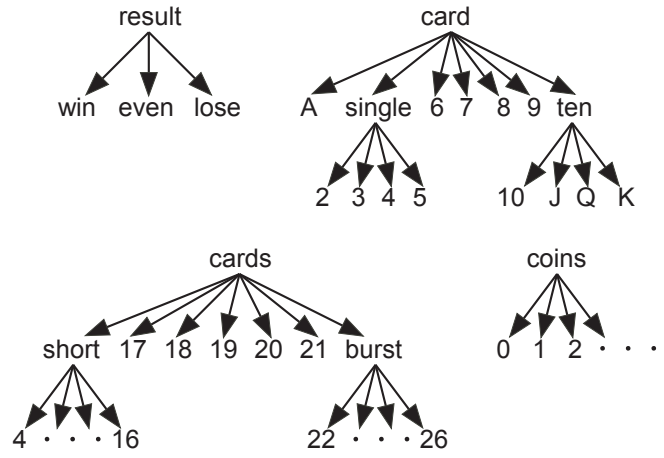

図 5.4: ブラックジャックシステムでのデータの具体化

発展段階 1

まず，勝敗全体を表す値”result” を”win”，“even”，“lose” に具体化する．それぞれ，勝ち，引き分け，負けを表す値である．

次に，カードの合計を比較して勝敗を決めることができるように，カードの合計全体を表す値”cards” を具体化する．“cards” は，4 から 26 の間の整数である．プレイヤーは，2 枚カードを引くので 2 のカードを 2 枚引いた場合の 4 が最小数である．A は，11 を選択でき，できるだけ 21 に近づくように選択すると考えるので，カードの合計が 2 や 3 になることはない．本システムにおけるブラックジャックのルールでは，プレイヤーは 2 枚カードを引くので，カードの合計の最大数は 21 である．つまり，Burst はない．しかし，ディーラーはカードの合計が 17 を超えるまでカードを引き続けなければならず，そのカードの合計の最大数は，カードの合計が 16 の時に 10，J，Q，K のいずれかを引いた場合の 26 である．ディーラーのカードの合計は，17，18，19，20，21，22 以上であり，22 以上の場合，無条件でプレイヤーの勝ちが決定する．プレイヤーのカードの合計は 16 以下の場合がある．これらを整理し，“cards” を” short”，17，18，19，20，21，“burst” に具体化する．“short” は 16 以下のカードの合計を表す値で，“burst” は 22 以上のカードの合計を表す値である．

そして，カードを表す値”card” を具体化する．本システムでは，10，J，Q，K は同じ数を表す．また，2 枚のカードのうち，1 枚でも 2 から 5 の間の数であれば，カードの合計は，“short” である．これらを整理し，“card” を”A”，“single”，6，7，8，9，“ten” に具体化する．ここで，“single” は，2 から 5 までのカードを表す値，“ten” は，10，J，Q，K を表す値である．

最後に，コインを自然数に具体化する．掛け金は 1 以上であるが，0 は，勝負に負けた場合の掛け金の没収を表すとする．

以下のプログラムは，このデータの具体化に基づいて詳細化した Player@2，Dealer@2，CardStack@2 クラスである．

```
/* version 2 */
class Player {
    CardStack cs = (CardStack)ProxyFactory.createProxy ("CardStack");
    Dealer d = (Dealer)ProxyFactory.createProxy ("Dealer");
    int play_v2 (int in) {
```

```
        CardDom_v2 c1 = cs.getCard_v2 ();
        CardDom_v2 c2 = cs.getCard_v2 ();
        ResultDom_v2 r = d.bet_v2 (new CardsDom_v2 (c1, c2));
        if (r.toString().equals ("win")) {
            return in * 2;
        } else if (r.toString().equals ("even") {
            return in;
        } else { // lose
            return 0;
        }
    }
}
class Dealer {
    CardsDom_v2 cards;
    Dealer () {
        int i = new Random ().nextInt (10) + 17;
        String s = null;
        if (i > 21) {
            s = "burst";
        } else {
            s = Integer.toString (i);
        }
        cards = new CardsDom_v2 (s);
    }
    ResultDom_v2 bet_v2 (CardsDom_v2 p) {
        if (cards.equals (new CardsDom_v2 ("burst")) ||
            p.equals (new CardsDom_v2 ("21")) ||
            p.compareTo (cards) > 0) {
            return new ResultDom_v2 ("win");
        } else if (p.compareTo (cards) == 0) {
            return new ResultDom_v2 ("even");
        } else {
            return new ResultDom_v2 ("lose");
        }
    }
}
class CardStack {
    CardDom_v2 getCard_v2 () {
        int i = new Random ().nextInt (13) + 1;
```

```
        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);
    }
}
```

Dealer@2 クラスでは，オブジェクトの生成時にディーラーのカードの合計をインスタンス変数 cards に格納している．格納される値は，17 から 21 までの整数か”burst” である．ルール上 16 以下の整数はない．以下の文は，17 から 26 までの整数をランダムに生成する．

```
int i = new Random ().nextInt (10) + 17;
```

bet_v2 () メソッドは，プレイヤーのカードの合計を仮引数 p で受け取り，プレイヤーが勝ちなら”win” を，引き分けなら”even” を，負けなら” lose” を返す．Dealer@1 クラスの bet_v1() メソッドは，カードの合計全体を表す値 “cards” を受け取ると勝敗全体を表す値”result” を返していたが，bet_v2() は，それらを具体化した値を入出力としている．

CardStack@2 クラスは，カードを要求するとカードを返す getCard_v2() メソッドのみを持つ．getCard_v2() は入力がなく，A，“single”，6 から 9 までのカード，“ten” のいずれかのカードをランダムに返す．CardStack@1 クラスの getCard_v1() メソッドは，カード全体を表す値”card” を返していたが，get-

Card_v2()は，それを具体化した値を返している．

Player@1 クラスから Player@2 クラスへの詳細化は，play_v1() メソッドから play_v2() メソッドへの詳細化である．play_v2() メソッドは，掛け金を仮引数 in で受け取り，カード スタックから 2 枚カード を引く．次に，ディーラーの bet_v2() を呼び出し，勝負をし，その結果のコインを返す．play_v1() では，“coins” を受け取り，“coins” を返していたが，勝敗が決まるようになったため，play_v2()では，自然数を受け取り，勝敗によって異なる自然数を返すようになっている．

発展段階 2

発展段階 1 で抽象化されている値を具体化し，最終的なブラックジャックシステムを構築する．具体化していない値は，16 以下のカード の合計を表す値” short”, 22から 26 までのカード の合計を表す値”burst”，2から 5 のカード を表す値”single”, 10 から K のカード を表す値” ten” である．発展段階 2 では，これらを次のように具体化する．

- “short” を 4 から 16 の整数に具体化
- “burst” を 22 から 26 の整数に具体化
- “single” を 2 から 5 の整数に具体化
- “ten” を 10，J，Q，K に具体化

以下のプログラムは，このデータの具体化に基づいて詳細化した Player@3，Dealer@3，CardStack@3 クラスである．

```
/* version 3 */
class Player {
    CardStack cs = (CardStack)ProxyFactory.createProxy ("CardStack");
    Dealer d = (Dealer)ProxyFactory.createProxy ("Dealer");
    int play_v3 (int in) {
        CardDom_v3 c1 = cs.getCard_v3 ();
        CardDom_v3 c2 = cs.getCard_v3 ();
        ResultDom_v2 r = d.bet_v3 (new CardsDom_v3 (c1, c2));
```

```
        if (r.toString().equals ("win")) {
            return in * 2;
        } else if (r.toString().equals ("even") {
            return in;
        } else { // lose
            return 0;
        }
    }
}
class Dealer {
    CardsDom_v3 cards;
    Dealer () {
        int i = new Random ().nextInt (10) + 17;
        String s = Integer.toString (i);
        cards = new CardsDom_v3 (s);
    }
    ResultDom_v2 bet_v3 (CardsDom_v3 p) {
        if (cards.compareTo (new CardsDom_v3 ("21")) > 0 ||
            p.equals (new CardsDom_v3 ("21")) ||
            p.compareTo (cards) > 0) {
            return new ResultDom_v3 ("win");
        } else if (p.compareTo (cards) == 0) {
            return new ResultDom_v3 ("even");
        } else {
            return new ResultDom_v3 ("lose");
        }
    }
}
class CardStack {
    CardDom_v3 getCard_v3 () {
        int i = new Random ().nextInt (13) + 1;
        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);
        }
    }
}
```

Dealer@3 クラスは，コンストラクタ，`bet_v2()` を詳細化することで得たクラスである．コンストラクタでは，17 から 26 のカードの合計をランダムに生成しインスタンス変数 `cards` に格納している．`bet_v2()` メソッドでは，“short” と ”burst” の抽象化した値を扱っていたが，`bet_v3()` メソッドでは，それらを具体化した値を扱っている．

CardStack@3 クラスは，`getCard_v2()` メソッドを `getCard_v3()` メソッドに詳細化することで得たクラスである．`getCard_v2()` では，“single” と ”ten” の抽象化した値を扱っていたが，`getCard_v3()` では，それらを具体化した値を扱っている．

5.2 商品在庫システム

商品在庫システムの開発例を示す．非常に簡単化しているが，実際の企業で使われているシステムをモデル化した例題である．したがって，現実的なソフトウェアを記述できるだけの記述力があると考えている．図 5.5 に示すこのシステムの仕様は次のとおりである．

- 顧客は，店に商品を発注する．簡単化のため，商品は一種類しかなく，発注は数量のみとする．
- 店は，受注すると在庫数と比較し，販売可能かどうかを顧客に知らせる．
- 顧客は，販売可能な場合，商品を購入し，そうでない場合は発注を取り消す．

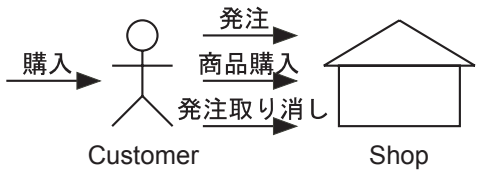

図 5.5: 商品在庫システム

5.2.1 仕様の抽象化

始めに，オブジェクトの扱うデータの観点から仕様の抽象化を行なう．このシステムは，顧客オブジェクトと店オブジェクトからなる．顧客オブジェクトの持つ機能は，購入機能のみであり，店オブジェクトの持つ機能は，比較機能，販売機能，取消機能の 3 つがある．顧客オブジェクトの購入機能の入出力データはない．店オブジェクトの機能の入出力データには，受注機能への入力データである「受注数」，比較機能の出力データである「比較結果」，販売機能の出力データである「商品」がある．さらに，店オブジェクト内に格納されるデータ「在庫数」ある．したがって，顧客オブジェクトを生成する Customer クラスを購入機能を抽象化した `buy()` メソッドのみからなるクラスとする．そして，店オブジェクトを生成する Shop クラスを比較，販売，取消機能を抽象化した `order()` メソッドと在庫数を格納するインスタンス変数 `stk` からなるクラスとする．ここで，`order()` の入出力データとして，それぞれ受注数などの入力全体を表す値”input”，比較結果や商品など出力全体を表す値” result” に抽象化する．さらに，在庫数を全体を表す値”stock” に抽象化する．

5.2.2 原始プロトタイプの作成

次に，原始プロトタイプを作成する．以下のプログラムは，原始プロトタイプを構成するクラス Customer@1 と Shop@1 である．それぞれメソッド `buy()` と `order()` を持つ．Shop@1 クラスは在庫数を格納するインスタンス変数 `stk` も持つ．

```
/* version 1 */
class Customer {
    Shop shop = (Shop)ProxyFactory.createProxy ("Shop");
```

```
    void buy () {
        Res r = shop.order (new In ("input"));
    }
}
class Shop {
    Stock stk = new Stock ("stock");
    Res order (In in) {
        return new Res ("result");
    }
}
```

5.2.3 プロトタイプの発展

そして，原始プロトタイプは，図5.6が示す順序で発展する．この例題には，3つのステップがあり，4回の機能発展によって最終的なクラスCustomer@4とShop@4を構成する．

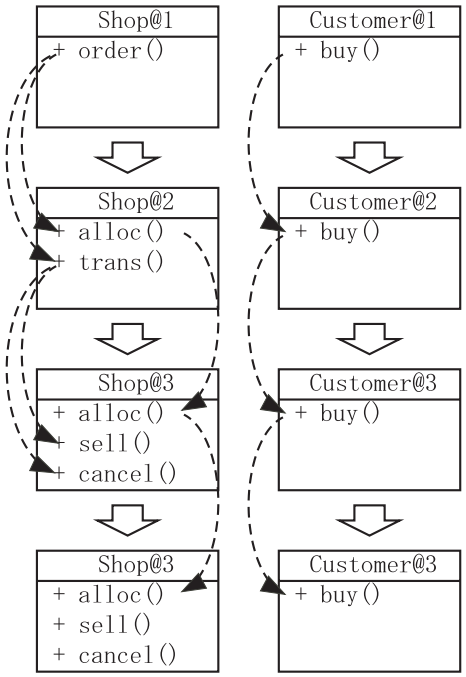

図 5.6: 発展手順

これらの機能発展は，図5.7が示すデータの具体化に基づいている．int，boolean，

void を除く stock や result といった小文字で始まる文字列や整数は値である．Res や Amount といった大文字で始まる文字列と int，boolean は，破線の四角で囲まれている値の集合，つまりメソッドの型を表している．void は何も入出力がないことを表すデータである．

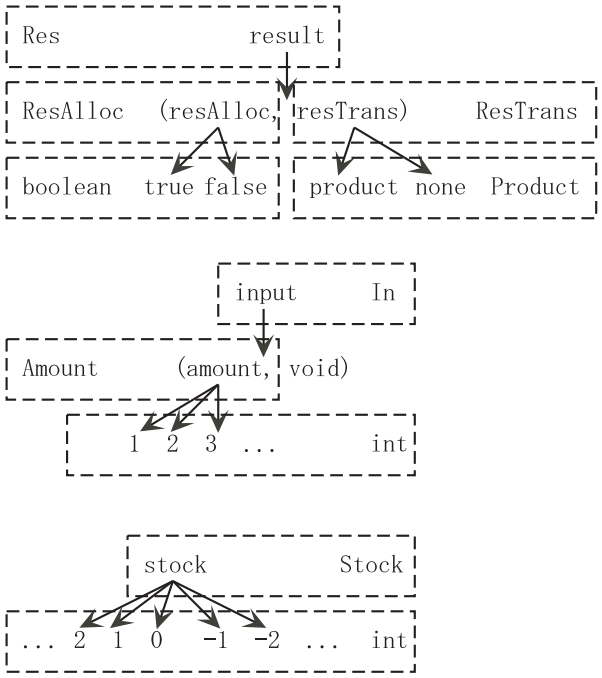

図 5.7: 本例におけるデータの具体化

発展 1

最初のステップは，メソッドの直積分割によって，`order()` を在庫数を比較する機能を抽象化したメソッド `alloc()` と販売，取消機能を抽象化したメソッド `trans()` に分割することである．この分割に沿って，`buy()` を変更する．その結果，以下のクラス Customer@2 と Shop@2 を得る．

```
/* version 2 */
class Customer {
    Shop shop = (Shop)ProxyFactory.createProxy ("Shop");
    void buy () {
        ResAlloc ra = shop.alloc (new Amount ("amount"));
        ResTrans rt = shop.trans ();
```

```
    }
}
class Shop {
    Stock stk = new Stock ("stock");
    Amount amnt;
    ResAlloc alloc (Amount n) {
        amount = n;
        return new ResAlloc ("resAlloc");
    }
    ResTrans trans () {
        return new ResTrans ("resTrans");
    }
}
```

alloc() は，受注数を抽象化した値”amount” を受け取ると，比較結果を抽象化した値”resAlloc” を返す．trans() は何も入力がなく，処理結果を抽象化した値 “resTrans” を返す．order() の入出力データ”input” と”result” はそれぞれ，組”(resAlloc, resTrans)” と”(amount, void)” に具体化している．さらに，alloc() が受け取った入力を格納するためのインスタンス変数 amnt を追加している．これを利用して，alloc() の後に実行する trans() に受注数を受け渡す．

発展 2

次のステップは，2 つの機能発展がある．一つは，メソッドの詳細化による alloc() メソッドの変更である．ここでは，より具体的な入力データが扱えるメソッドに変更している．もう一つは，メソッドの直和分割による trans() の分割である．ここで，trans() をそれぞれ販売機能と取消機能を抽象化したメソッド sell() と cancel() に分割する．ステップ 1 と同様に，buy() も変更する．その結果，以下のクラス Customer@3 と Shop@3 を得る．

```
/* version 3 */
class Customer {
    Shop shop = (Shop)ProxyFactory.createProxy ("Shop");
    void buy () {
        Product p;
        if (shop.alloc (10).cond ()) {
            p = shop.sell ();
```

```
        } else {
            p = shop.cancel ();
        }
    }
}
class Shop {
    int stk = 20;
    int amnt = 0;
    ResAlloc alloc (int n) {
        if (stk >= n){
            stk -= n;
            amnt = n;
            return new ResAlloc ("resAlloc");
        } else {
            return new ResAlloc ("resAlloc");
        }
    }
    Product sell () {
        return new Product ("product");
    }
    Product cancel () {
        return new Product ("none");
    }
}
```

Shop@3 クラスでは，在庫量全体を表す抽象値”stock”が整数に具体化している．これは，`alloc()` メソッドに入力として渡される受注数と在庫数を比較するためである．さらに，受注数を格納するインスタンス変数 `amnt` のデータ型も整数型に具体化されている．直和分割によって得られるメソッド `sell()` と `cancel()` は，それぞれ商品を表す値”product” と空の商品を表す値 “none” を返す．

Customer@3 クラスの buy() メソッド中に使われている `cond()` メソッドは，プログラマが if 文の then 部分を実行するのか else 部分を実行するのかを選択するために使う．`alloc(10)` は，結果的に値”resAlloc” を返すが，その値では抽象的すぎて，`sell()` を実行するのか `cancel()` を実行するのか決定できない．この発展度においては，どちらとも実行される可能性がある．この可能性をプログラマが制御するために `cond()` メソッドを用いる．

発展 3

最後に，メソッドの詳細化によって，alloc()をより具体的な出力データが扱えるメソッドに変更する．その結果，以下のクラス Customer@4 と Shop@4 を得る．

```
/* version 4 */
class Customer {
    Shop shop = (Shop)ProxyFactory.createProxy ("Shop");
    void buy () {
        Product p;
        if (shop.alloc (10)) {
            p = shop.sell ();
        } else {
            p = shop.cancel ();
        }
    }
}
class Shop {
    int stk = 20;
    int amnt = 0;
    boolean alloc (int n) {
        if (stk >= n){
            stk -= n;
            amnt = n;
            return true;
        } else {
            return false;
        }
    }
    Product sell () {...}
    Product cancel () {...}
}
```

alloc()メソッドの出力は，“resAlloc”から販売可能であることを表す値”true”と販売不可能であることを表す値”false”に具体化する．この発展によって，sell()を実行するのかcancel()を実行するのかを決定することができるようになる．したがって，cond()のようなプログラマによる実行制御メソッドが必要なくなる．

第 6 章

機能発展と機能追加によるクラス継承の実現

本章では，機能発展に機能追加の概念を導入することで安全なクラス継承が実現できることを示す．クラス継承は，オブジェクト指向技術の核となる概念である．しかしながら，クラス継承には fragile base class problem と呼ばれる致命的な問題がある．この問題が生じないクラス継承を機能発展と機能追加で実現する．はじめに，この問題を説明する．次に機能追加を導入する．クラス継承の中心は，機能の詳細化と追加であり，機能発展では機能追加を実現できないからである．最後に，機能発展と機能追加による機能拡張が単調であることを証明する．この性質がこの問題を解決する．

6.1　クラス継承における問題点

クラス継承とは，クラスの機能を受け継ぐためのメカニズムである．機能を受け継いだクラスを子クラス，反対に子クラスに機能を譲るクラスを親クラスと呼ぶ．クラス継承をクラスの拡張の道具として利用すると，次のような 3 つの利点がある．

1. クラスの変更を差分的に記述できる．
2. 再利用性が高い．

3. 変更容易性が高い．

子クラスでは，親クラスから機能を受け継ぐため，新たに追加する機能や修正する機能のみを記述するだけでよい．この差分的な記述は，プログラマにとって理解容易性と修正容易性を高める．プログラマは，何が変化したかを子クラスを見ることで理解できる．加えて，このメカニズムによって同じ機能の違う記述 (つまりコピー) を減らすため，修正しやすくなる．

次に，子クラスで機能の変更/拡張を行なえることは，再利用性を高めることにつながる．機能の一部を変更することや追加するためには，コードの変更や追加が必要となる．クラス継承を用いると，親クラスのコードを変更する必要はなく，子クラスで変更/拡張を行なえる．したがって，格段に親クラスの再利用性は高くなる．もちろん，どんな変更/拡張も扱えるわけではないが，親クラスを変更することなく機能拡張できる場合が増える．したがって，再利用性は高まる．

最後に，子クラスのオブジェクトは，親クラスのオブジェクトとして扱うことができるので，親クラスから子クラスへの変更にともなう他のクラスの変更が少なくなる．モジュールの変更が常に互換性を保つわけではない．互換性を得るためにグルーコードなどを記述することがある．クラス継承は，部分型を実現しているため，常に子クラスは親クラスと互換性がある．もちろん，常に親クラスの代わりに子クラスを用いても振舞いが同じとは限らない．しかし，少なくともグルーコードを必要しないため，容易に変更できるという利点がある．

ところが，クラスの変更/修正する時，クラス階層全体を考慮しなければならないという欠点もある．これは，メソッドのオーバーライドと動的束縛による downcall が原因である．downcall とは，親クラスのメソッドの代わりに子クラスでオーバーライドによって定義したメソッドを呼び出すことである．クラスを変更する場合，downcall によってプログラマが意図しないメソッドが呼び出されるかも知れない．例えば，メソッド $m()$ を持つクラス C を考える．$m()$ は，自然数を返すメソッドとする．さらに，C を継承するクラス E を考える．E では，$m()$ を整数を返すメソッドにオーバーライドしているとする．ここで，C に $n()$ を追加することを考える．$n()$ は自然数上のなんらかの計算を $m()$ を使って計算するとする．すると，$n()$ を実行する時，E の $m()$ が動的束縛によって呼び出されることになる．その時，負の整数を返すとすると，$m()$ では，自然数上の計算を行なっているので，エラーが

発生するかもしれない．したがって，C を変更する場合，E を考慮して変更しなければ誤りが発生する．つまり，クラス階層全体を考慮する必要がでてくる．

downcall による問題をまとめたものが fragile base class problem である．この問題には 5 つの側面があり，上の例は，その一つの側面である．この問題は，別の視点から見ると，機能を拡張したのにも関わらず，クラス継承がその機能の単調増加を阻んでいると見ることができる．

この問題は，Mixin[2] を用いたクラス継承のモデルでも解決できない．Mixin とは，機能を提供するクラスの断片である．他の Mixin やクラスと合成しクラスを作る．Mixin は，多重継承における様々な問題を解決し，オブジェクト指向言語に多大な貢献をした．残念なことに，Mixin を用いても fragile base class problem は発生する．なぜなら，Mixin によるクラス継承でもメソッドのオーバーライドは許されているからである．この問題の本質は，再定義されたメソッドが動的束縛によってプログラマの意図とは反して実行されることである．したがって，Mixin を用いたクラス継承のモデルでも fragile base class problem は解決されない．

6.2 機能追加の導入

機能拡張の中心は，機能の詳細化と追加である．これらを現在のオブジェクト指向言語ではクラス継承によって実現している．機能の詳細化はメソッドのオーバーライドを用いたクラス継承，機能の追加はメソッドのオーバーライドを用いていないクラス継承で実現する．例えば，図 6.1 に示すモノクロプリンターからカラープリンターへの機能拡張は詳細化であり，モノクロプリンターからモノクロコピー機への機能拡張は追加である．モノクロプリンターで書類をプリントする機能をメソッド print() が実現している．カラープリンターでは，モノクロプリンターでは扱っていない赤や青などの色を扱う必要があるため，print() をオーバーライドする．メソッド scan() は書類の読み込み機能を表している．コピー機は読み込み機能とプリント機能を組み合わせることで実現できる．

メソッドのオーバーライドを排除したクラス継承では，fragile base class problem が起きない．この問題の原因は，オーバーライドと動的束縛による downcall なので，オーバーライドを排除すると問題が起きなくなる．しかし，それでは機能の

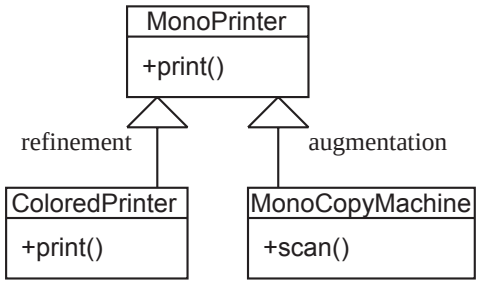

図 6.1: クラス継承による機能の詳細化と追加

詳細化による機能拡張ができなくなり，記述力が低下する．

そこで，メソッドのオーバーライドによる機能の詳細化の代わりに本論文で定義した機能発展を用いる．機能発展では，意味の領域まで踏み込んで，その入出力データや状態に関する整合性を保つように機能を変更する．入出力データや状態の情報が増え，より詳細な振舞いになるという意味での振舞いの増加はあるものの，振舞いの減少/追加を許していない．オーバーライドによる振舞いの減少/追加が downcall での予期せぬ影響につながる．機能発展による機能拡張はこれを防ぐ．

これらを踏まえて，クラス発展では機能追加による機能拡張を扱うことができないので，機能追加による機能拡張を次のように定義する．

定義 13 (機能追加関係)

A と *B* をクラスとする．$A \lhd B$ と記述する機能追加関係 $\lhd$ を次のように定義する．

$$A \lhd B \\ \Leftrightarrow \quad A \prec^c_P B \wedge (md, ...) \Subset^c_P B \Rightarrow (md, ...) \notin^c_P A$$

ここで，*B* は，*A* の直接の子クラスであり，関係 $\prec^c_P$ はそれを表している．

本論文では，以降，この機能追加関係をクラス継承とみなす．これは，オーバーライドを排除している制限されたクラス継承だからである．したがって，$A \lhd B$ である時，A を親クラス B を子クラスと呼ぶ．

次の節で，機能発展と機能追加による機能拡張において問題が発生しないことを証明する．

6.3 単調性の証明

fragile base class problem は，振舞いが変化しないように，あるいは振舞いを詳細化するために機能 (つまりメソッド) の記述を変更したのにもかかわらず，downcall によって実際は振舞いが変わる，あるいは振舞いを詳細化しないことがあるということを示している．したがって，クラス A がクラス $A^{\#}$ を詳細化し，クラス B がクラス $B^{\#}$ を詳細化している時，$A \oplus B$ が $A^{\#} \oplus B^{\#}$ の詳細化であるならば，この問題は起きない．ここで，演算子 $\oplus$ は，機能追加によるクラス合成演算子である．図 6.2 は，その性質を表している．その形式的記述は次のとおりである．

$$A^{\#} \sqsubseteq^{\mathcal{C}} A \wedge B^{\#} \sqsubseteq^{\mathcal{C}} B \Rightarrow (A^{\#} \oplus B^{\#}) \sqsubseteq^{\mathcal{C}} (A \oplus B)$$

この性質を**クラス合成の単調性**と呼ぶ.

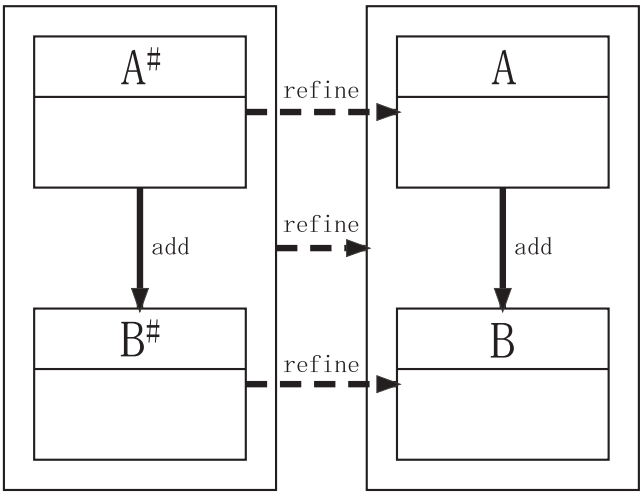

図 6.2: 証明する性質

機能追加関係は交換律が成立しない．したがって，このクラス合成演算子でも交換律は成立しない．しかし，クラスをメソッド集合とみると，機能追加関係ではオーバーライドを排除しているため，クラスの合成を和集合で表現できる．したがって，クラス合成演算子 $\oplus$ を次のように定義する．

定義 14 (クラス合成演算子)

A と B をクラスとする．$Meth_A$ と $Meth_B$ をそれぞれ A と B のメソッド集合とする．$A \oplus B$ と記述するクラス合成演算子 $\oplus$ を次のように定義する．

$$A \oplus B \iff Meth_A \cup Meth_B$$

クラス合成の単調性を証明するためには，**機能追加における振舞いの不変性**，**ストア合成の単調性**，**フィールド環境合成の単調性**の 3 つの性質が必要となる．機能追加における振舞いの不変性とは，たとえクラスを合成したとしても任意のメソッドの振舞いは変化しないという性質である．これは，クラス合成にはオーバーライドがないので成立する性質である．振舞いが変化しなければ，発展関係が崩れないように見えるが，余分なインスタンス変数，つまり子クラスで宣言する変数，が追加されても発展関係が常に成立するとは限らない．子クラスで宣言したインスタンス変数に関しても発展関係が常に成立することを示す必要があるため，この性質は自明ではない．それを示すためにストア合成の単調性を証明する．ストア合成の単調性は，それぞれ発展関係にあるストアを合成しても発展関係が成立するという性質である．また，フィールド環境合成の単調性は，同様に，それぞれ発展関係にあるフィールド環境を合成しても発展関係が成立するという性質である．ストア合成の単調性は，フィールド環境合成の単調性を利用して証明している．

ここで，クラス合成の単調性を証明するのに必要な 2 つの仮定を置く．一つ目は，親クラスと子クラスの間に発展関係が全く存在しないという仮定である．つまり，発展関係が全く存在しないクラスしか合成できないということである．ここでの発展関係とは，インスタンス変数上の発展関係とメソッド上の発展関係である．機能の追加は新しい概念，振舞いなどを付け加えるものなので，この仮定は，それほど強い制限ではないと考えている．この仮定によって，機能発展と機能追加は独立となる．本論文では，この仮定を**独立性の仮定** と呼ぶ．二つ目は，子クラスでは，親クラスのインスタンス変数を操作しないという仮定である．子クラスで親クラスのインスタンス変数を操作できることがクラス継承の魅力的な点の一つである．しかし，直接親クラスのインスタンス変数を操作することは危険で，うまく使いこなすことができないのが現状である．したがって，この仮定も強い制限ではないと考えている．この仮定を**機能分離の仮定**と呼ぶ．

6.3.1 フィールド 環境合成の単調性

クラス A のオブジェクトが取り得るフィールド 環境の集合を $FD(A)$ と記述する．クラス B がクラス A の子クラスである場合，クラス B のオブジェクト obj_B が持つインスタンス変数集合 Id_B とクラス A のオブジェクト obj_A が持つインスタンス変数集合 Id_A の間には $Id_A \subseteq Id_B$ という関係が存在する．さらに，任意のインスタンス変数 $x \in Id_A$ の値が obj_A と obj_B において等しい場合，obj_A のフィールド環境 $\mathcal{F}_A \in FD(A)$ と obj_B のフィールド 環境 $\mathcal{F}_B \in FD(B)$ の間に $\mathcal{F}_A \subseteq \mathcal{F}_B$ という関係が存在する．$\mathcal{F}_B$ を $\mathcal{F}_A$ とその差分によって表現するために，+ 演算を導入し，$\mathcal{F}_B$ を $\mathcal{F}_A + \Delta\mathcal{F}$ と記述する．ここで，$\Delta\mathcal{F}$ はクラス B で宣言されたインスタンス変数のみのフィールド 環境を表し，$\mathcal{F}_B - \mathcal{F}_A$ である．一般的なクラス継承では，親クラスで宣言されたインスタンス変数と同じ識別子のインスタンス変数を子クラスで宣言することが許されていないので，ここでは，$dom(\mathcal{F}_A) \cap dom(\Delta\mathcal{F}) = \emptyset$ を仮定する．すると，以下のフィールド 環境合成の単調性が成立する．

Lemma 1

$$\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} \wedge \Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \Delta\mathcal{F} \iff \mathcal{F}^{\#} + \Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} + \Delta\mathcal{F}$$

Proof まず，$\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} \wedge \Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \Delta\mathcal{F} \Rightarrow \mathcal{F}^{\#} + \Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} + \Delta\mathcal{F}$ が成立することを示す．+ 演算の仮定より，$x \in dom(\mathcal{F}^{\#})$ ならば，$(\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x) = \mathcal{F}^{\#}(x)$ が成立し，$x \in dom(\Delta\mathcal{F}^{\#})$ ならば，$(\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x) = \Delta\mathcal{F}^{\#}(x)$ が成立する．$\mathcal{F} + \Delta\mathcal{F}$ についても同様である．任意のインスタンス変数 $x^{\#} \in dom(\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})$, $x \in dom(\mathcal{F} + \Delta\mathcal{F})$ について，

- $x^{\#} \in dom(\mathcal{F}^{\#})$ かつ $x \in dom(\mathcal{F})$ の場合，$\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$ より $x^{\#} \preceq^{id} x \Rightarrow (\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x^{\#}) \preceq^{\mathcal{D}} (\mathcal{F} + \Delta\mathcal{F})(x)$ が成立する．
- $x^{\#} \in dom(\Delta\mathcal{F}^{\#})$ かつ $x \in dom(\Delta\mathcal{F})$ の場合，$\Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \Delta\mathcal{F}$ より $x^{\#} \preceq^{id} x \Rightarrow (\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x^{\#}) \preceq^{\mathcal{D}} (\mathcal{F} + \Delta\mathcal{F})(x)$ が成立する．
- $x^{\#} \in dom(\mathcal{F}^{\#})$ かつ $x \in dom(\Delta\mathcal{F})$ の場合，独立性の仮定より，$x^{\#} \preceq^{id} x \Rightarrow (\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x^{\#}) \preceq^{\mathcal{D}} (\mathcal{F} + \Delta\mathcal{F})(x)$ が成立する．

- $x^{\#} \in dom(\Delta\mathcal{F}^{\#})$ かつ $x \in dom(\mathcal{F})$ の場合，独立性の仮定より，$x^{\#} \preceq^{id} x \Rightarrow (\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x^{\#}) \preceq^{\mathcal{D}} (\mathcal{F} + \Delta\mathcal{F})(x)$ が成立する．

次に，$\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} + \Delta\mathcal{F} \Rightarrow \mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} \wedge \Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \Delta\mathcal{F}$ が成立することを示す．独立性の仮定から，任意のインスタンス変数 $x^{\#} \in dom(\mathcal{F}^{\#})$，$x \in dom(\Delta\mathcal{F})$ について，$x^{\#} \not\preceq^{id} x$ が成立する．$\mathcal{F}$ と $\Delta\mathcal{F}^{\#}$ についても同様である．さらに，+ 演算の仮定より，$x \in dom(\mathcal{F}^{\#})$ ならば，$(\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x) = \mathcal{F}^{\#}(x)$ が成立し，$x \in dom(\Delta\mathcal{F}^{\#})$ ならば，$(\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x) = \Delta\mathcal{F}^{\#}(x)$ が成立する．$\mathcal{F} + \Delta\mathcal{F}$ についても同様である．したがって，任意のインスタンス変数 $y^{\#} \in dom(\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})$，$y \in dom(\mathcal{F} + \Delta\mathcal{F})$ について，

- $y^{\#} \in dom(\mathcal{F}^{\#})$ かつ $y \in dom(\mathcal{F})$ ならば，$x^{\#} \preceq^{id} x \Rightarrow (\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x^{\#}) \preceq^{\mathcal{D}} (\mathcal{F} + \Delta\mathcal{F})(x)$ の仮定から，$\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$ が成立する．
- $y^{\#} \in dom(\Delta\mathcal{F}^{\#})$ かつ $y \in dom(\Delta\mathcal{F})$ ならば，$x^{\#} \preceq^{id} x \Rightarrow (\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})(x^{\#}) \preceq^{\mathcal{D}} (\mathcal{F} + \Delta\mathcal{F})(x)$ の仮定から，$\Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \Delta\mathcal{F}$ が成立する．■

6.3.2 ストア合成の単調性

フィールド環境上の + 演算を導入した結果，オブジェクトのフィールド環境を差分的に記述できるようになった．概念的にストアはオブジェクトのフィールド環境のコレクションなので，この演算を用いて，ストアを差分的に記述するストア上の + 演算を定義する．任意のストア $\mathcal{S}$ について，任意のオブジェクト obj のフィールド環境が $\mathcal{F} + \Delta\mathcal{F}$ である時，obj から $\mathcal{F}$ への写像 $\mathcal{S}'$ と obj から $\Delta\mathcal{F}$ への写像 $\Delta\mathcal{S}$ を用いて $\mathcal{S}$ を $\mathcal{S}' + \Delta\mathcal{S}$ と記述する．以下に形式的な定義を示す．

定義 15 ストアの差分的記述

$$\mathcal{S} + \Delta\mathcal{S} = \{obj \mapsto (c', \mathcal{F} + \Delta\mathcal{F}) \mid \mathcal{S}(obj) = (c, \mathcal{F}) \wedge \Delta\mathcal{S}(obj) = (c', \Delta\mathcal{F})\}$$

ストアを + 演算を用いて差分的に記述ことを目的としているので，$\mathcal{S} + \Delta\mathcal{S}$ と記述されたストアについて，$dom(\mathcal{S}) = dom(\Delta\mathcal{S})$ であることを仮定する．さらに，フィールド環境上の+ 演算は継承関係によって合成されるフィールド環境を差分的に記述するためのオペレータなので，$c \trianglelefteq c'$ を仮定する．$c \triangleleft c'$ の場合は直観的

だが，$c = c'$ の場合は，サブクラスを持たないクラスのオブジェクトのフィールド環境を差分的に記述するケースで，$\Delta\mathcal{F} = \emptyset$ とする．すると，次のストア合成の単調性が成立する．

Lemma 2

$$\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} \wedge \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \Delta\mathcal{S} \iff \mathcal{S}^{\#} + \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} + \Delta\mathcal{S}$$

Proof　まず，$\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} \wedge \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \Delta\mathcal{S} \Rightarrow \mathcal{S}^{\#} + \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} + \Delta\mathcal{S}$ が成立することをを示す．ストアの差分的記述の仮定より，$dom(\mathcal{S}^{\#}) = dom(\mathcal{S}^{\#} + \Delta\mathcal{S}^{\#})$，$dom(\mathcal{S}) = dom(\mathcal{S} + \Delta\mathcal{S})$ なので，$\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}$ によって $obj^{\#} \sqsubseteq_{\mathcal{S}}^{\mathcal{S}^{\#}} obj$ となる任意のオブジェクト $obj^{\#} \in dom(\mathcal{S}^{\#})$，$obj \in dom(\mathcal{S})$ について，$obj^{\#} \sqsubseteq_{\mathcal{S}+\Delta\mathcal{S}}^{\mathcal{S}^{\#}+\Delta\mathcal{S}^{\#}} obj$ が成立することを示せばよい．$(\mathcal{S}^{\#} + \Delta\mathcal{S}^{\#})(obj^{\#}) = (c^{\#}, \mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})$，$(\mathcal{S} + \Delta\mathcal{S})(obj) = (c, \mathcal{F} + \Delta\mathcal{F})$ とし，$obj^{\#} \sqsubseteq_{\mathcal{S}}^{\mathcal{S}^{\#}} obj$ であることを仮定すると，$\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$ である．さらに，$\Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \Delta\mathcal{S}$ より，$\Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \Delta\mathcal{F}$ が成立する．Lemma 1 より $\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} + \Delta\mathcal{F}$ が成立するので，$obj^{\#} \sqsubseteq_{\mathcal{S}+\Delta\mathcal{S}}^{\mathcal{S}^{\#}+\Delta\mathcal{S}^{\#}} obj$ が成立する．

次に，$\mathcal{S}^{\#} + \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} + \Delta\mathcal{S} \Rightarrow \mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} \wedge \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \Delta\mathcal{S}$ が成立することをを示す．これも同様に任意のオブジェクト $obj^{\#} \in dom(\mathcal{S}^{\#} + \Delta\mathcal{S}^{\#})$，$obj \in dom(\mathcal{S} + \Delta\mathcal{S})$ について，$obj^{\#} \sqsubseteq_{\mathcal{S}+\Delta\mathcal{S}}^{\mathcal{S}^{\#}+\Delta\mathcal{S}^{\#}} obj$ が成立する時，$obj^{\#} \sqsubseteq_{\mathcal{S}}^{\mathcal{S}^{\#}} obj$ かつ $obj^{\#} \sqsubseteq_{\Delta\mathcal{S}}^{\Delta\mathcal{S}^{\#}} obj$ が成立することを示せばよい．$(\mathcal{S}^{\#} + \Delta\mathcal{S}^{\#})(obj^{\#}) = (c^{\#}, \mathcal{F}^{\#} + \Delta\mathcal{F}^{\#})$，$(\mathcal{S} + \Delta\mathcal{S})(obj) = (c, \mathcal{F} + \Delta\mathcal{F})$ とし，$obj^{\#} \sqsubseteq_{\mathcal{S}+\Delta\mathcal{S}}^{\mathcal{S}^{\#}+\Delta\mathcal{S}^{\#}} obj$ であることを仮定すると，$\mathcal{F}^{\#} + \Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F} + \Delta\mathcal{F}$ が成立する．Lemma 1 より，$\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \mathcal{F}$ かつ $\Delta\mathcal{F}^{\#} \sqsubseteq^{\mathcal{F}} \Delta\mathcal{F}$ が成立するので，$obj^{\#} \sqsubseteq_{\mathcal{S}}^{\mathcal{S}^{\#}} obj$ かつ $obj^{\#} \sqsubseteq_{\Delta\mathcal{S}}^{\Delta\mathcal{S}^{\#}} obj$ が成立する．■

6.3.3　クラス合成のよる振舞いの不変性

クラス合成の単調性を証明するためには，クラス合成によってメソッドの振舞いが不変であることを示す必要がある．振舞いが変化してしまうと，メソッドの発展関係が失われてしまうからである．したがって，次のクラス合成による振舞いの不変性を証明する．ここで，$P + c$ は，プログラム P にクラス c を追加することを表現している．

Lemma 3

$c \lhd c' \wedge (md, ..., e) \notin^c_P c' \wedge P + c' \vdash \langle E[obj'.md(v)], \mathcal{S} + \Delta\mathcal{S} \rangle \overset{*}{\hookrightarrow} \langle E[v'], \mathcal{S}' + \Delta\mathcal{S} \rangle$
$\Rightarrow P \vdash \langle E[obj.md(v)], \mathcal{S} \rangle \overset{*}{\hookrightarrow} \langle E[v'], \mathcal{S}' \rangle$

Proof $c \lhd c'$ かつ $(md, ..., e) \notin^c_P c'$ より，オブジェクト obj が起動するメソッド md の body 式とオブジェクト obj' が起動するメソッド md の body 式は等しい．さらに，md は $\Delta\mathcal{S}$ を更新せず，$\mathcal{S}$ を $\mathcal{S}'$ に更新する．したがって，$P \vdash \langle E[obj.md(v)], \mathcal{S} \rangle \overset{*}{\hookrightarrow} \langle E[v'], \mathcal{S}' \rangle$ が成立する．■

6.3.4 クラス合成の単調性

これらの補題を用いて，以下のクラス合成の単調性を証明する．

Theorem 1

$$A^{\#} \sqsubseteq^{\mathcal{C}} A \wedge B^{\#} \sqsubseteq^{\mathcal{C}} B \Rightarrow (A^{\#} \oplus B^{\#}) \sqsubseteq^{\mathcal{C}} (A \oplus B)$$

Proof この定理を証明するには，クラス $A^{\#}$ とクラス A の間に発展関係が存在する時，それぞれにクラス $B^{\#}$ とクラス B を追加した時，発展関係が存在することを示す必要がある．機能分離の仮定があるので，$B^{\#}$ や B でそれぞれの親クラスのフィールド環境が変化することはない．したがって，メソッド間に発展関係が存在する時，それにクラス $B^{\#}$ と B で宣言されているインスタンス変数に関するストアの差分を追加しても発展関係が成立することを示すことである．これを示すと，その逆，つまり $B^{\#}$ と B の間に発展関係が存在する時にそれぞれ $A^{\#}$ と A で宣言したインスタンス変数に関するストアの差分を追加しても発展関係が成立することも示すことができる．それをより具体的に示すと，任意のストア $\mathcal{S}^{\#}$，$\mathcal{S}'^{\#}$，$\mathcal{S}$，$\mathcal{S}'$ と任意の差分 $\Delta\mathcal{S}^{\#}$，$\Delta\mathcal{S}$ について，

$c \lhd c' \wedge c^{\#} \lhd c'^{\#} \wedge (md^{\#}, ...) \notin^c_P c^{\#} \wedge (md, ...) \notin^c_P c' \wedge$
$(\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} \wedge v^{\#} \sqsubseteq^{\mathcal{D}} v \wedge P \vdash \langle E[obj^{\#}.md^{\#}(v^{\#})], \mathcal{S}^{\#} \rangle \overset{*}{\hookrightarrow} \langle E[v'^{\#}], \mathcal{S}'^{\#} \rangle \wedge$
$P \vdash \langle E[obj.md(v)], \mathcal{S} \rangle \overset{*}{\hookrightarrow} \langle E[v'], \mathcal{S}' \rangle \Rightarrow \mathcal{S}'^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}' \wedge v'^{\#} \sqsubseteq^{\mathcal{D}} v')$
$\Rightarrow (\mathcal{S}^{\#} + \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} + \Delta\mathcal{S} \wedge v^{\#} \sqsubseteq^{\mathcal{D}} v \wedge$
$P \vdash \langle E[obj^{\#}.md^{\#}(v^{\#})], \mathcal{S}^{\#} + \Delta\mathcal{S}^{\#} \rangle \overset{*}{\hookrightarrow} \langle E[v'^{\#}], \mathcal{S}'^{\#} + \Delta\mathcal{S}^{\#} \rangle \wedge$
$P \vdash \langle E[obj.md(v)], \mathcal{S} + \Delta\mathcal{S} \rangle \overset{*}{\hookrightarrow} \langle E[v'], \mathcal{S}' + \Delta\mathcal{S} \rangle \Rightarrow \mathcal{S}'^{\#} + \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}' + \Delta\mathcal{S} \wedge v'^{\#} \sqsubseteq^{\mathcal{D}} v')$

が成立することを示すことである．Lemma 2 より，$\mathcal{S}^{\#} + \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S} + \Delta\mathcal{S}$ から $\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}$ と $\Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \Delta\mathcal{S}$ が導かれる．Lemma 3 より，$c \lhd c'$ かつ $(md, ...) \notin^{c}_{P} c'$ かつ $P \vdash \langle E[obj.md(v)], \mathcal{S} + \Delta\mathcal{S}\rangle \overset{*}{\hookrightarrow} \langle E[v'], \mathcal{S}' + \Delta\mathcal{S}\rangle$ から $P \vdash \langle E[obj.md(v)], \mathcal{S}\rangle \overset{*}{\hookrightarrow} \langle E[v'], \mathcal{S}'\rangle$ が導かれる．同様に，$P \vdash \langle E[obj^{\#}.md^{\#}(v^{\#})], \mathcal{S}^{\#}\rangle \overset{*}{\hookrightarrow} \langle E[v'^{\#}], \mathcal{S}'^{\#}\rangle$ も導かれる．仮定から，$\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}$ と $v^{\#} \sqsubseteq^{\mathcal{D}} v$ と $P \vdash \langle E[obj.md(v)], \mathcal{S}\rangle \overset{*}{\hookrightarrow} \langle E[v'], \mathcal{S}'\rangle$ と $P \vdash \langle E[obj^{\#}.md^{\#}(v^{\#})], \mathcal{S}^{\#}\rangle \overset{*}{\hookrightarrow} \langle E[v'^{\#}], \mathcal{S}'^{\#}\rangle$ から $\mathcal{S}'^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}'$ と $v'^{\#} \sqsubseteq^{\mathcal{D}} v'$ が導かれる．Lemma 2 より，$\mathcal{S}'^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}'$ と $\Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \Delta\mathcal{S}$ から $\mathcal{S}'^{\#} + \Delta\mathcal{S}^{\#} \sqsubseteq^{\mathcal{S}} \mathcal{S}' + \Delta\mathcal{S}$ が導かれる．これは，メソッドの詳細化関係についてのみ示しているが，同様にメソッドの直積分割関係，直和分割関係についても示すことができる．■

第 7 章

議論

7.1 性能評価

我々は，仲介オブジェクトによる抽象実行メカニズムの実現方法が実用上問題がないことを性能評価実験によって示す．本技法を用いて開発したプログラムと一般的なプログラムの大きな違いは，仲介オブジェクトを利用しているかいないかである．オブジェクトのインスタンス化に多少の違いはある[1]が，記述面でそれほど大きな違いはない．しかし，実行面では，メソッド呼び出しの仲介と仲介オブジェクトの生成の分だけ実行速度が遅くなる．実験 1 では，メソッド呼び出しの仲介における実行速度に問題がないこと，実験 2 では，仲介オブジェクトの生成における実行速度に問題がないことを示す．実験 3 では，メソッドの縮退による抽象実行がオブジェクトの縮退による抽象実行よりも効率的であることを示す．メソッドの縮退による抽象実行は抽象実行メカニズムの効率かのために導入されている．動作環境は，Windows2000，512MB RAM，PentiumIII(800MHz)，JDK 1.4.1 である．

[1]とはいえ，コンストラクタを呼び出す代わりに `ProxyFactory.createProxy()` メソッドを呼び出すだけである．

表 7.1: メソッド呼び出しの仲介にかかる実行時間

	1000 回	10000 回	30000 回
事例 1	1231 ms	5578 ms	14771 ms
事例 2	4727 ms	69901 ms	395108 ms
事例 3	4036 ms	61268 ms	311337 ms

7.1.1 メソッド呼び出しの仲介

実験 1 では，以下の 3 つの事例をそれぞれ 1000 回，10000 回，30000 回繰り返し，メソッド呼び出しの仲介にかかる時間を測定した．表 7.1 がその結果である．

事例 1: Shop クラスが `Shop@3` まで発展している状況で `sell()` メソッドを呼び出す．この時，仲介オブジェクトは縮退を行なうことなくメソッド呼び出しを仲介するだけである．

事例 2: Shop クラスが `Shop@3` まで発展している状況で `trans()` メソッドを呼び出す．この時，仲介オブジェクトは，`Shop@3` クラスのオブジェクトを `Shop@2` クラスのオブジェクトに縮退させ，再び `trans()` メソッドを呼び出し，得られた値を返す．

事例 3: Shop クラスが `Shop@2` まで発展している状況で `alloc(10)` を呼び出す．この時，仲介オブジェクトは，メソッドの縮退によって `alloc(10)` を `alloc(amount)` に変換し，その呼び出しによって得られた値を返す．

この実験は，できるだけ仲介オブジェクトの処理時間のみを比較できるようにしている．呼び出されるメソッド `sell()`，`trans()`，`alloc()` は，単に (データとしての) オブジェクトを返すだけであり，処理時間に大きな差はない[2]．あらかじめ仲介オブジェクトと Shop オブジェクトを生成しているので，インスタンス化にかかる時間は測定に含まれていない．さらに，これらのオブジェクトは，繰り返しの回数分用意しているので，繰り返しの 1 回目しか縮退しないことや繰り返しの中で毎回インスタンス化することがないように配慮している．

表 7.1 の実験結果は，もっとも時間のかかる場合でも 3 万回で約 400 秒しかかかっていないことを示している．プロトタイプが m 個のオブジェクトからなり，一つの

[2] `alloc()` は代入文の分だけ処理が多いが，仲介オブジェクトの処理と比べると極めて小さい．

オブジェクトはもっとも抽象的なオブジェクトで，それ以外のオブジェクトはそれぞれ n バージョンまで発展していたとする．この時，最悪すべてのオブジェクトの発展度を最も抽象的なレベルまで縮退する必要があり，その回数は $(m-1)(n-1)$ 回である．このことから，3 万回の縮退が生じるケースは，あるプロトタイプが 301 個のオブジェクトからなり，それぞれが 101 バージョンまで存在する場合や 3001 個のオブジェクトが 11 バージョンまで存在する場合である．これは最悪の場合なので，実際にはこれらより大きな規模のプロトタイプにおいて発生する回数である．

7.1.2 仲介オブジェクトの生成

実験 2 では，実験 1 で用いた事例 2 と多少の変更を加えた事例 2-1，2-2 について，仲介オブジェクトの生成をそれぞれ 1000 回，10000 回，30000 回繰り返し，その実行時間を測定した．その結果が表 7.2 である．事例 2-1 と 2-2 は，事例 2 における Shop オブジェクトに余分なインスタンス変数をそれぞれ 2 つと 4 つ追加した事例である．これは，仲介オブジェクトの生成には，callee オブジェクトの生成も含まれているので，インスタンス変数の個数が異なる場合での実行速度を比較するためである．ここで，追加したインスタンス変数は，`amnt` と同じ型を持ち，変数名のみが異なっている．もちろん，`amnt` と同様に初期値の代入を行なっている．

表 7.2 は，以下の 2 つのことを示している．

1. インスタンス変数の個数が増えてもさほど実行時間に差がない．

2. 3 万個の仲介オブジェクトを生成したとしても，たった 8.6 秒しかかからない．

したがって，この実験から，仲介オブジェクトの生成における実行速度に実用上問題ないことがいえる．

7.1.3 メソッドとオブジェクトの縮退における実行速度の比較

メソッドの縮退による抽象実行は，オブジェクトの縮退による抽象実行によって代替可能であるが，効率化のために用意されている．しかし，表 7.1 の結果から

表 7.2: 仲介オブジェクトの生成にかかる実行時間

	1000 回	10000 回	30000 回
事例 2	1232 ms	3415 ms	8222 ms
事例 2-1	1282 ms	3445 ms	8493 ms
事例 2-2	1272 ms	3605 ms	8623 ms

表 7.3: 縮退の違いによる実行時間の比較

	1000 回	10000 回	30000 回
事例 2-1	7000 ms	92903 ms	464297 ms
事例 2-2	9263 ms	115596 ms	532706 ms
事例 3-1	4507 ms	65154 ms	323335 ms
事例 3-2	4466 ms	64713 ms	321432 ms

それほど効率化されているようには見えない．事例 2 がオブジェクトの縮退による抽象実行で，事例 3 がメソッドの縮退による抽象実行である．これは，実験 1 では，Shop オブジェクトの持つインスタンス変数の数が少ないため，オブジェクトの縮退にかかる時間とメソッドやデータの縮退にかかる時間に大幅な差がないからである．したがって，我々は，事例 2 と事例 3 において，Shop オブジェクトの余分なインスタンス変数を追加した場合における実行速度の変化を調べる実験を行なった．

実験 3 では，実験 2 で用いた事例 2-1，2-2 と実験 1 で用いた事例 3 に多少の変更を加えた事例 3-1，3-2 について，実験 1 と同様にそれぞれ 1000 回，10000 回，30000 回のメソッド呼び出しにかかる時間を測定した．表 7.3 がその結果である．ここで，それぞれ，事例 2-1，2-2 はオブジェクトの縮退，事例 3-1，3-2 はメソッドの縮退による抽象実行である．

事例 3-1 と 3-2 は，事例 2-1，2-2 と同様に，事例 3 の実験を Shop オブジェクトに余分なインスタンス変数をそれぞれ 2 つと 4 つ追加した事例である．Shop@2 においては，唯一の値 amount が格納され，Shop@3 においては，整数 (厳密には 1 以上の整数) が格納される．したがって，事例 2-1，2-2 において，オブジェクトの

縮退時には，整数から amount への変換を必要とする．

表 7.3 から，オブジェクトの縮退による抽象実行は，オブジェクトの持つインスタンス変数が増えれば増えるほど実行速度が遅くなっていることが分かる．しかし，メソッドの縮退による抽象実行では，たとえインスタンス変数が増えても実行速度がほとんど変化していないことが分かる．したがって，プロトタイプに含まれるオブジェクトの規模が大きくなるほどメソッドの縮退による抽象実行が効果的であることがいえる．

7.2 機能拡張とソフトウェア品質

本技法では，メソッドのオーバーライドを制限している．従来のオーバーライドは，型上の制約を満たす限り任意の変更を許している．これは，プログラミングの自由度を高めているが，誤りの混入を増やす原因でもある．本技法の許しているオーバーライドは，型上の制約だけではなく，入出力データ上の整合性まで満たしているものである．逆に，制限されているのは，super を用いた親クラスの持つメソッドの再利用である．super を用いて，子クラスの持つ共通の機能を親クラスに持たせ，再利用することがある．

基本的に，本技法において，親クラスに相当するクラスは，抽象化したクラスなので，一度，抽象化したクラスのメソッドを実行し，再び発展したクラスのメソッドの実行を続けることは難しい．オブジェクトを抽象化する必要がある場合は，実行を継続できない．

我々は，機能拡張とソフトウェア品質の向上を別々に行なうべきだと考えている．プロトタイピングの目的を考えると，ソフトウェア品質を高める必要性はあまり高くない．早い段階でソフトウェア品質を考慮すると，問題が複雑になり，プロトタイピングコストが高くなる．したがって，本技法において再利用性を高めるためのオーバーライドが制限されていても問題ないと考えている．

再利用性を向上する技術として，リファクタリング [10] がある．リファクタリングとは，システムの振舞いを変えることなくコード理解や修正が容易になるようにコードを徐々に変更することである．本技法では，リファクタリングを扱えない．なぜなら，リファクタリングによってさまざまな発展関係を崩してしまう

からである．しかし，機能拡張と再利用性の向上は方向性が異なっているので，関心事の分離の原理から，別々に行なうべきであると考えている．

しかしながら，全くリファクタリングを本技法に取り込めないわけではない．全てのオブジェクトの発展度がそろった状態で一度本技法によるプロトタイプの発展をやめ，リファクタリングを行ない，その後のプロトタイプを原始プロトタイプとして扱い，再び本技法によるプロトタイプの発展を続けるというプロセスを経ることでリファクタリングを本技法に取り入れることが可能であると考えている．

7.3 関連研究

抽象実行をソフトウェア工学に適用するアイデアは新しいわけではない．吉岡らによって提唱された ISDR(Incremental Software development method based on Data Reification) は関数型言語 ML における機能発展を扱っている [13]．ISDR でも本技法と同様に機能の入出力データにおける整合性を保つ機能発展が定義されている．しかしながら，純粋な関数型言語に基づいているため，機能の副作用を扱うことができない．さらに，ISDR では，関数の発展に焦点をあてているので，ストラクチャといったモジュールを構成することができない．本技法では，ストアの発展を導入することで，副作用を含む機能の発展を扱うことができ，メソッドの詳細化や分割によるクラスの発展を導入することでモジュールの発展も扱うことができる．

我々の研究は，具体的な値を使うことなくプログラムを実行するという意味で symbolic execution[6] に似ている．symbolic execution では，プログラムは変数名を値として扱うことによって実行される．逆に，本技法では，プログラマによって与えられた抽象化された値やその上の操作によって実行される．さらに，本技法は，部分的に実装されたプログラムの実行も許している．

実行時にモジュール (コンポーネント) のバージョンを制御する CCM(Component Configuration Management)[7] がある．これは，従来ソフトウェアの開発フェーズで適用されてきたコンフィグレーションマネージメントを実行フェーズに適用したものである．本技法における仲介オブジェクトは，実行時にオブジェクトのバージョンを変更するという意味で CCM の一種である．[7] では，CCM に必要な入出

力，インターフェース，振舞いの 3 つの互換性が定義されている．一般的に，これらの互換性を満たすようにモジュールを更新することは難しい．本技法では，機能の更新を機能発展によって制限することで入出力と振舞いの互換性を実現している．さらに，メソッドの詳細化や分割といった機能発展を厳密に定義し，それらによってのみ機能を更新するように制限することでインターフェースの互換性を実現している．

第 8 章

まとめと今後の課題

本論文では，抽象解釈に基づく発展的プロトタイピング技法を提案した．オブジェクトの発展関係を抽象解釈に基づいて形式的に定義することで，概念的には，途中段階での実行が可能となることが明らかになった．オブジェクトの発展関係は，機能の入出力データ上の発展関係と実行前後のシステム全体の状態上の発展関係によって定義している．これにより，インスタンス変数に関する副作用を扱うことができる．

次に，本技法によるプロトタイピングを支援するプログラミング環境を提供した．これにより，抽象実行メカニズムが実現可能であることとリフレクション技術や XML 技術によって効率的なプロトタイピングが可能となることが明らかになった．プログラミング環境は，プロトタイプの発展記述を支援する発展エディタ，抽象実行のためのクラスライブラリ，抽象実行をともなうプロトタイプの実行を視覚化するビジュアライザからなる．そのクラスライブラリは，リフレクション技術と XML 技術によって，抽象実行を制御するための仲介オブジェクトを機械的に生成する．仲介オブジェクトの性能評価実験によって，大きな規模でのプロトタイピングに適用可能であることを示した．

最後に，機能追加による機能拡張を導入した．機能発展と機能追加による機能拡張では，クラス継承での問題点である fragile base class problem が起きないことをクラス合成の単調性を証明することによって示した．クラス合成の単調性は，クラス合成における振舞いの不変性，ストア合成の単調性，フィールド合成の単調性の 3 つの補題を用いて証明している．これらの性質が成立することで，開発者は

クラス毎の機能に着目したプロトタイピングに集中できることが明らかになった．

今後の課題は 2 つある．一つは，発展関係のテスト/検証環境を構築することである．現段階では，発展の正しさをチェックする環境がない．しかし，様々な発展関係から，概念的には，網羅的にテストケースを生成することが可能である．なぜなら，メソッドの全ての入出力データや状態について関係づけがあるからである．難しい点は，テストケースが膨大な量になってしまうことである．もう一つは，プロトタイプ発展の枝分けやマージといったプロセスを本技法に取り入れることである．現段階での本技法では，プロトタイピングのバックトラックを考慮していない．これは，開発を一歩進めるという点に焦点をあてているためである．しかし，開発中にバックトラックすることなくプロトタイピングを終えることはほとんどない．したがって，これらを考慮することはとても重要なことである．難しい点は，たとえ枝分けやマージをしたとしても抽象実行が可能であることを保証することである．

謝辞

本研究を行なうに当たり, 終始御指導を賜わった片山 卓也教授に深謝致します.

また, 日頃から有益な御助言をいただき, 多面に渡って励ましていただいた東京工業大学 情報理工学研究科 計算工学専攻の権藤 克彦助教授に感謝致します.

最後に, 本論文をまとめるに当たって御協力いただいた片山研究室の諸兄に厚く御礼申し上げます.

参考文献

[1] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. Joint ACM Conf. on OOPSLA and ECOOP*, pages 303–311. ACM Press, 1990.

[3] Mehdi Jazayeri Carlo Ghezzi and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[4] P. Cousot and R.Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of POPL 77: The 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[5] Matthew Flatt, Shriram Krishnamurthi, and Matthrias Fellenisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, 1998.

[6] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[7] Magnus Larsson and Ivica Crnkovic. New challenges for configuration management. In *Proceedings of 9th Symposium on System Configuration Management*, pages 232–243. Springer Verlag, 1999.

[8] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. *Lecture Notes in Computer Science*, 1445:355–382, 1998.

[9] Carroll Morgan. *Programming from Specifications Second Edition*. Prentice Hall, 1994.

[10] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[11] Shari Lawrence Pfleeger. *Software Engineering: theory and practice*. Prentice Hall, 2001.

[12] Daniel M. Yellin and Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. In *Proc. of OOPSLA*, pages 176–190. ACM Press, 1994.

[13] Nobukazu Yoshioka, Masato Suzuki, and Takuya Katayama. Incremental software development method based on abstract interpretation. In *Proc. of IWSSD-9*, pages 126–134, 1998.

本研究に関する発表論文

[1] Hiroyuki Ozaki, Katsuhiko Gondow and Takuya Katayama: “Evolutionary Prototyping Technique Using Abstract Interpretation in Java”, In Proc. of The 7th International Symposium on Future Software Technology, 2002.

[2] Hiroyuki Ozaki, Katsuhiko Gondow and Takuya Katayama: “Class Refinement for Software Evolution”, In Proc. of Sixth International Workshop on Principles of Software Evolution, IEEE Computer Society Press, pages 51-56, 2003.

[3] Hiroyuki Ozaki, Shingo Ban, Katsuhiko Gondow and Takuya Katayama: “An Environment for Evolutionary Prototyping Java Programs based on Abstract Interpretation”, In Proc. of 10th Asia-Pacific Software Engineering Conference, IEEE Computer Society Press, 2003, to be presented.

[4] 尾崎 弘幸，番 真悟，権藤 克彦，片山 卓也: “抽象解釈に基づく発展的プロトタイピング技法のための支援環境の設計と実装”，電子情報通信学会論文誌 (D-I) 投稿準備中.

第 A章

CLASSICJAVAの操作的意味論

以下に，CLASSICJAVA の操作的意味論を示す.

$$P \vdash \langle E[\mathbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle E[object], \mathcal{S}[object \mapsto \langle c, \mathcal{F}\rangle]\rangle \qquad [new]$$

where $object \notin \mathrm{dom}(\mathcal{S})$ and $\mathcal{F} = \{c'.fd \mapsto \mathbf{null} \mid c \leq^c_P c' \text{ and } \exists t \text{ s.t. } \langle c'.fd, t\rangle \notin^c_P c'\}$

$$P \vdash \langle E[object\underline{:\ c'}.fd], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}\rangle \qquad [get]$$

where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$ and $\mathcal{F}(c'.fd) = v$

$$P \vdash \langle E[object\underline{:\ c'}.fd = v], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}[object \mapsto \langle c, \mathcal{F}[c'.fd \mapsto v]\rangle]\rangle \qquad [set]$$

where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$

$$P \vdash \langle E[object.md(v_1, \ldots, v_n)], \mathcal{S}\rangle \hookrightarrow \langle E[e[object/\mathbf{this}, v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}\rangle \qquad [call]$$

where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$ and $\langle md, (t_1, \ldots, t_n \to t), (var_1, \ldots, var_n), e\rangle \in^c_P c$

$$P \vdash \langle E[\mathbf{super}\underline{\equiv object : c'}.md(v_1, \ldots, v_n)], \mathcal{S}\rangle$$
$$\hookrightarrow \langle E[e[object/\mathbf{this}, v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}\rangle \qquad [super]$$

where $\langle md, (t_1 \ldots t_n \to t), (var_1 \ldots var_n), e\rangle \in^c_P c'$

$$P \vdash \langle E[\mathbf{view}\ t'\ object], \mathcal{S}\rangle \hookrightarrow \langle E[object], \mathcal{S}\rangle \qquad [cast]$$

where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$ and $c \leq_P t'$

$$P \vdash \langle E[\mathbf{let}\ var = v\ \mathbf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle E[e[v/var]], \mathcal{S}\rangle \qquad [let]$$

$$P \vdash \langle E[\mathbf{view}\ t'\ object], \mathcal{S}\rangle \hookrightarrow \langle \text{error: bad cast}, \mathcal{S}\rangle \qquad [xcast]$$

where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$ and $c \not\leq_P t'$

$$P \vdash \langle E[\mathbf{null}\underline{:\ c}.fd], \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S}\rangle \qquad [nget]$$

$$P \vdash \langle E[\mathbf{null}\underline{:\ c}.fd = v], \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S}\rangle \qquad [nset]$$

$$P \vdash \langle E[\mathbf{null}.md(v_1, \ldots, v_n)], \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S}\rangle \qquad [ncall]$$