契約番号 ９８第０２０号

高度情報化支援ソフトウェアシーズ育成事業

動的適応可能なソフトウェアの系統的構成方式と言語の研究開発

論 文

平成１１年２月

天野 憲樹

要 旨

開放型分散システム環境や移動計算機環境の普及にともない，プログラマは設計の段階でソフトウェアの実行環境およびその動的な状態変化を完全に予測することができなくなった．しかし，多様な実行環境とその動的な状態変化がプログラムの実行に与える影響は大きく，これを無視することはできない．それゆえ，今日のソフトウェアは多様な実行環境で動作し，その動的な状態変化に適応できることが要求される．本研究では，このようなソフトウェアを指して動的適応可能なソフトウェアと呼ぶ．一般に，こうした動的適応可能なソフトウェアを作成することは困難であり，その有効な構成手法が望まれている．

本研究では，動的適応可能なソフトウェアのためのモデル **DAS** を提案し，DAS モデルにもとづくプログラミング言語の基本機構を設計・開発する．DAS モデルは，さまざまな実行環境やその動的な状態変化に適応する機構を持ち，DAS モデルにもとづくプログラミング言語は，動的適応可能なソフトウェアの柔軟な記述を支援する．本研究では，こうした DAS モデルおよびその記述言語の有効性を示すとともに，DAS モデルにもとづく動的適応可能なソフトウェアの有効性を示す．

動的適応可能なソフトウェアを実現する上で，各ソフトウェア自身の振舞いを実行環境の状態に応じて変えることは有効な方法である．しかし，言うまでもなく同一のソフトウェアに対して実行環境ないしその状態ごとに振舞いの異なるバージョンを多数作成することは現実的ではない．その上，実行環境やその状態変化に対するソフトウェアの振舞いは，各ソフトウェアの性質，さらにはその個々の処理の性質に依存する場合が多く，これを外部から汎用的に制御することは困難である．このため，各ソフトウェア自身が実行環境やその動的な状態変化に対処（適応）する能力を持つ必要がある．本研究では，このソフトウェアの能力を指して動的適応可能性と呼ぶ．適応適応可能なソフトウェア，すなわち動的適応可能性を持つソフトウェアは，さまざまな実行環境やその動的な状態変化に適応できるだけでなく，その適応動作により各実行環境固有の特性を最大限に活かすことができるようになると考えられる．

しかし，プログラマが事前に予測可能な実行環境とその状態には限界があるため，プログラマが事前にソフトウェアに与えることができる動的適応可能性にもやはり限界がある．このため，ソフトウェアの動的適応可能性は，拡張が可能な構造で実現され，事前に予測できなかった実行環境やその動的な状態変化に対して，拡張/変更できる必要がある．本研究では，そのような拡張可能な動的適応可能性を備えたソフトウェアのモデル DAS（以下，DAS モデル）を提案する．以下は，DAS モデルの主要な概念である．

- ソフトウェアにおける適応動作の単位を手続きとし，手続き呼び出しにもとづいて適応動作の制御を行なう．
- DAS モデルは，適応可能手続きと呼ばれる実行環境の状態にもとづく総称手続き (総称関数) を持つ．
- 適応可能手続きの制御機構は，自己反映計算を利用して実現される．
- DAS モデルにもとづく動的適応可能なソフトウェアは，メタレベルアーキテクチャを形成する．

これらの概念は，本研究の以下の目標にもとづいている．

- 既存言語によるソフトウェアへの高い親和性．
- 柔軟性の高い動的適応可能性の実現．
- 高い可読性，保守性，拡張性の実現．

既存のソフトウェア資産を開放型分散システム環境や移動計算機環境で有効に利用したいという要求は大きい．それゆえ，既存のプログラミング言語で記述されたさまざまなソフトウェアに動的適応可能性を導入できる方式が望ましい．DAS モデルでは，ソフトウェアにおける適応動作の単位を手続きとし，ソフトウェアの適応動作は適応可能手続きと呼ばれる実行環境の状態にもとづく総称手続き (総称関数) として実現される．ひとつの適応可能手続きは，実行環境の状態に応じた実行コードを持つ複数の適応可能メソッドから構成される．適応可能手続きが呼び出されると，実行環境の状態に応じて適切な適応可能メソッドが選択され，実行される．この手法は，原理的に手続き呼び出しの機構を持つ多くの既存言語により記述されたソフトウェアに適用できる．この結果，既存のソフトウェア資産を有効利用できる可能性が高まる．本研究では，DAS モデルの基本機構を Java 言語のクラスライブラリ (以下，DAS クラスライブラリ) として設計・実装した．そして，実際に DAS クラスライブラリを用いて Java のアプリケーションに動的適応可能性の導入を行なった．これらの実装を通じて，DAS モデルが既存言語およびそのソフトウェア対して高い親和性を持つことを実証した．

本研究の手法において，ソフトウェアの柔軟な動的適応可能性を実現するには，適応可能手続きの制御機構を実行環境の状態に応じて柔軟に変更できることが求められる．DAS モデルでは，この制御機構自体が適応可能手続きを用いて自己反映的に実現される．そのため，この制御機構自体も実行環境の状態変化に適応することができる (実行環境の状態

変化に応じて変化する）. 実行環境の状態変化に応じて，この制御機構を変えることで，ソフトウェアの柔軟な適応動作を実現することが可能となる．たとえば，実行環境の状態に応じたソフトウェアの機能の一時的な制限やその解除といった適応動作を，プログラムを修正することなく実現することができるようになる． さらにこの制御機構が，適応可能手続きを用いて実現されるため，この機構のカスタマイズは，その構造を修正することなく，適応可能メソッドのインクリメンタルな追加で実現できる．これらの利点について，DAS クラスライブラリを用いた例題の実装を通して，実際に確かめることができた．また，ソフトウェアにおける柔軟な適応動作の有効性も実際に例題を用いて示した．

DAS モデルにもとづく動的適応可能なソフトウェアは，メタレベルアーキテクチャを形成する．そのメタレベルは，ソフトウェアにおける適応動作の制御を実現する機構の記述からなり，ベースレベルは，各ソフトウェアにおける問題領域の記述からなる．つまり，適応動作の制御機構は，ベースレベルの記述から完全に分離され，メタレベルに隠蔽される．この記述の分離は，動的適応可能なソフトウェアの可読性，保守性，拡張性を向上させる．この利点についても例題の実装を通して，実際に確かめることができた．

目 次

1 序論 **1**
1.1 背景 . . . 1
1.2 動的適応可能なソフトウェア . . . 3
1.3 動的適応可能なソフトウェアモデル DAS . . . 4
1.4 本論文の構成 . . . 5

2 準備 **7**
2.1 総称関数 . . . 7
2.2 自己反映計算 . . . 8
2.3 まとめ . . . 10

3 事例 :ソフトウェアの適応動作 **11**
3.1 GUI アプリケーション . . . 11
3.2 ネットワークアプリケーション . . . 12
3.3 マルチメディアアプリケーション . . . 14
3.4 移動コードシステム . . . 15
3.5 まとめ . . . 16

4 動的適応可能なソフトウェアモデル DAS **18**
4.1 設計方針 . . . 18
4.1.1 実行環境の表現 . . . 19
4.1.2 状態変化の取得方法 . . . 20
4.1.3 適応動作の単位とその制御 . . . 21
4.1.4 柔軟な動的適応可能性 . . . 24
4.2 DAS モデル . . . 25
4.3 動的適応可能性 . . . 29

4.4 適応の戦略にもとづく動的適応可能性 32
4.5 自己反映計算を用いた適応の機構 35
4.6 まとめ 37

5 DAS クラスライブラリ **39**
5.1 DAS クラスライブラリの構成と機能 39
5.1.1 メタオブジェクトに関するクラス 40
5.1.2 適応可能メソッドに関するクラス 45
5.1.3 ユーティリティのクラス 45
5.2 稼働環境 46
5.3 DAS クラスライブラリを用いたプログラミング 46
5.3.1 記述の概要 46
5.3.2 適応可能メソッドによる適応動作の記述 48
5.3.3 状態イベントを利用した適応動作の記述 51
5.4 動的適応可能な回覧板システム 53
5.4.1 例題の仕様 53
5.4.2 GUI に関する適応動作 55
5.4.3 タイマーイベントの利用 57
5.5 メタオブジェクトのカスタマイズ 58
5.6 まとめ 63

6 関連研究 **64**
6.1 オペレーティングシステム 64
6.2 移動コード/エージェントシステム 65
6.3 言語 ・モデル ・機構 66
6.4 その他 69
6.5 DAS モデルの特徴 72
6.6 まとめ 75

7 結論および今後の課題 **76**
7.1 動的適応可能なソフトウェアモデル DAS 76
7.2 DAS クラスライブラリ 77
7.3 今後の課題 78

謝辞 **81**

本研究に関する発表論文 88

図 目 次

2.1 CLOS における総称関数と通常の Lisp 関数 8
2.2 自己反映的な計算システムの概念図 9

4.1 各モジュールの抽象度に応じた仮想的かつ局所的な実行環境 20
4.2 DAS モデルとアドホックなモデル 27
4.3 メタレベルディスパッチ & ベースレベルディスパッチ 35

5.1 動的適応可能な回覧板システム 53

表 目 次

5.1 メタオブジェクトに関するクラス 40
5.2 実行環境の状態情報 41
5.3 イベントオブジェクトの提供する標準の状態イベント 42
5.4 標準の状態イベントとコールバックメソッド 43
5.5 適応可能メソッドのデータ構造 45
5.6 抽象度の高いディスプレイサイズ 48
5.7 適応の戦略を実装する適応可能メソッド 60
5.8 ユーザ定義イベントのイベント取得メソッドとイベントハンドラ 61
5.9 ユーザ定義イベントとコールバックメソッド 62

第 1 章

序論

開放型分散システム環境や移動計算機環境の普及にともない，さまざまな実行環境とその動的な状態変化に適応できるソフトウェア (以下，動的適応可能なソフトウェア) およびその系統的構成手法が望まれている．本研究の目的は，*1)* 多様な実行環境とその動的な状態変化に柔軟に対処できる機構を備えたソフトウェアのモデル – DAS モデル – を提案し，DAS モデルにもとづくプログラミング言語の基本機構を実装する，*2)* DAS モデルにもとづく動的適応可能なソフトウェアを実際に作成し，その有効性を示す，ことである．本章では，まず本研究の背景について述べ，動的適応可能なソフトウェアの必要性を説く．次に，本研究で目指す動的適応可能なソフトウェアについて，さらにそのソフトウェアモデル DAS について概説する．最後に，本論文の構成を示す．

1.1 背景

近年コンピュータネットワークのインフラストラクチャが急速に整備され，現在ネットワークを通じて大小さまざな種類の計算機が相互に接続されている．こうしたネットワーク上の計算機は，ハードウェアの種類だけでなく，その上で動作しているソフトウェアの種類もまったく異なる場合が少なくない．こうした状況はソフトウェアの開発およびその保守にとって大きな問題となっている．このため，計算機の機種やそのオペレーティングシステム (OS) などに強く依存せず，ネットワーク上のどの計算機でも利用可能な開放型システムの研究開発が盛んに行われるようになった．現在，急速に普及しつつある Java [15] なども，この流れの中に位置づけることができよう．

しかし，開放型システムは別の問題を引き起こす．システム透過性の実現を最大の目標とする開放型システムでは，各計算機固有の特性を活かすことができるソフトウェアの作

成が困難になる．開放型システムは，各計算機の持つ特殊なデバイスなどの利用をある程度制限することで，システム透過性を実現するからである．これはソフトウェアの開発者およびそのユーザにとって大きなジレンマとなっている．つまり，計算機の機種，OS，デバイスなどに依存せずに動作することができ，かつ各計算機が持つ特殊なデバイスなども最大限に利用できるソフトウェアが望まれている．

また，近年の技術革新は計算機の小型化，小電力化を促進し，実用に耐えうる携帯可能な計算機の実現を可能にした．こうした携帯可能な計算機の出現は，それを自由に持ち歩き，移動した先々でネットワークに接続して目的に応じた処理を行うモーバイルコンピューティング [12, 23] の実用化を推進した．現在，高速電話回線網などモーバイルコンピューティングのためのインフラストラクチャが整備されつつあり，移動計算機環境が身近なものとなってきている．

こうした移動計算機環境におけるモーバイルコンピューティングにおいても大きな問題が起きている．移動計算機環境で使用される計算機（以下，移動計算機）は，その携帯性のため，ハードウェア的な制約が大きい．たとえば，移動計算機のディスプレイは，デスクトップ型計算機のそれと比べてかなり小さい．ディスプレイの大小は，グラフィカルユーザインタフェース（GUI）にもとづくアプリケーションを使用する際，その look & feel および操作性に大きな影響を与える．また，移動計算機において携帯性のために制限された機能は，PC カードにより提供されることが多い．たとえば，ネットワークボードなどは通常，移動計算機本体内部に組み込まれず，PC カードの 1 つとして提供されることが多い．移動計算機のユーザは移動した先々の計算機環境やその時々の目的に応じて，PC カードを着脱し処理を行うのが一般的である．このため移動計算機の構成要素は，アプリケーションの実行中に動的に変更される可能性がある．こうした PC カードの活線着脱を支援するために，最近では，Windows95/98 をはじめ，Linux，FreeBSD など，さまざまな OS がプラグ& プレイ機能を提供している．こうしたプラグ & プレイ機能により，これらの OS 上のアプリケーションは，ハードウェア構成の変化を直接意識することなく，処理を継続することができるようになってきている．しかし，こうした OS によるハードウェア構成の隠蔽にもかかわらず，その変化が各アプリケーションに与える影響は無視できない場合が多い．その顕著な事例として計算機の移動にともないネットワークの通信メディアを変更した場合があげられる．計算機の移動前後で，ネットワークの通信メディアを有線から無線に変更した場合，バンド幅の変化，エラー率や一時的な回線切断の増加などがアプリケーションの実行に与える影響は無視できないほど大きいと言える．

これらの問題をすべて内包するもっとも顕著な事例として，ここ数年，活発に研究開発されているモーバイルエージェントシステム [58, 53] があげられる．モーバイルエージェ

ントは，その実行中にネットワークを通じて他の計算機に移動し処理を継続する．この場合，移動前後における計算機の機種やOS，ハードウェア構成，ソフトウェア構成など，その他すべての要素が一度に変化する可能性がある．通常，こうしたモーバイルエージェントシステムでは，特定の計算機やOSなどに依存しないモーバイルエージェントのプラットフォームを提供しており，モーバイルエージェントはそのプラットフォーム上で動作する．しかし，これは前述した問題を払拭するものでないことは明らかである．つまり，モーバイルエージェントは移動先の計算機が持つ特殊なデバイスなどを利用することができなくなる．モーバイルエージェントでは，その移動前後に起こる変化が抜本的かつ急激であるため，問題はより深刻である．それらの変化が，モーバイルエージェントの実行にもたらす影響を完全に予測することは不可能である．

このような状況に対して，アプリケーションは何ら有効な手だてを持たない．実行環境の動的な状態変化を知ることができないからである．また，OSもこのような状況のアプリケーションに対して有効な支援を行うことができない．実行環境の状態変化に対処するアプリケーションの振舞いは，各アプリケーションの性質，さらにはその個々の処理の性質に依存するからである．これをOSなどにより外部から汎用的に支援することは困難である．こうした問題は移動計算機環境が普及しつつある現在，かなり大きな問題として認識されており，今日，実行環境の動的な状態変化に対処する機構を持ったソフトウェアおよびその系統的構成手法が強く求められている．本研究では，このようなソフトウェアを動的適応可能なソフトウェアと呼ぶ．

1.2 動的適応可能なソフトウェア

動的適応可能なソフトウェアを実現する上で，各ソフトウェア自身の振舞いを実行環境の状態に応じて変えることは有効な方法と考えられる．しかし，同一のソフトウェアに対して実行環境ないしその状態ごとに振舞いの異なるバージョンを多数作成することは現実的ではない．その上，実行環境やその状態変化に対処するソフトウェアの振舞いは各ソフトウェアの性質，さらにはその個々の処理の性質に依存する場合が多く，これを外部から汎用的に制御することは困難である．このため，各ソフトウェア自身が実行環境やその動的な状態変化に対処（適応）する能力を持つ必要がある．本研究では，このようなソフトウェアの能力を指して動的適応可能性と呼ぶ．適応適応可能なソフトウェア，すなわち動的適応可能性を持つソフトウェアは，さまざまな実行環境やその動的な状態変化に適応できるだけでなく，その適応動作により各実行環境固有の特性を最大限に活かすことができるようになると考えられる．

しかし，プログラマが事前に予測可能な実行環境とその状態には限界があるため， プログラマが事前にソフトウェアに与えることができる動的適応可能性にもやはり限界がある．このため，ソフトウェアの動的適応可能性は，拡張が可能な構造で実現され，事前に予測できなかった実行環境やその動的な状態変化に対して，拡張/変更できる必要がある．本研究では，そのような拡張可能な動的適応可能性を備えたソフトウェアの実現を目指す．

1.3 動的適応可能なソフトウェアモデル DAS

本研究では，拡張可能な動的適応可能性を備えたソフトウェアのモデル *DAS* (以下，DAS モデル) を提案する．以下は，DAS モデルの主要な概念である．

- ソフトウェアにおける適応動作の単位を手続きとし，手続き呼び出しにもとづいて適応動作の制御を行なう．
- DAS モデルは，適応可能手続きと呼ばれる実行環境の状態にもとづく総称手続き (総称関数) を持つ．
- 適応可能手続きの制御機構は，自己反映計算を利用して実現される．
- DAS モデルにもとづく動的適応可能なソフトウェアは，メタレベルアーキテクチャを形成する．

これらの概念は，本研究の以下の目標にもとづいている．

- 既存言語によるソフトウェアへの高い親和性．
- 柔軟性の高い動的適応可能性の実現．
- 高い可読性，保守性，拡張性の実現．

既存のソフトウェア資産を開放型分散システム環境や移動計算機環境で有効に利用したいという要求は大きい．それゆえ，既存のプログラミング言語で記述されたさまざまなソフトウェアに動的適応可能性を導入できる方式が望ましい．DAS モデルでは，ソフトウェアにおける適応動作の単位を手続きとし，ソフトウェアの適応動作は適応可能手続きと呼ばれる実行環境の状態にもとづく総称手続き (総称関数) として実現される．ひとつの適応可能手続きは，実行環境の状態に応じた実行コードを持つ複数の適応可能メソッドから構成される．適応可能手続きが呼び出されると，実行環境の状態に応じて適切な適応可能

メソッドが選択され，実行される．この手法は，原理的に手続き呼び出しの機構を持つ多くの既存言語により記述されたソフトウェアに適用できる．この結果，既存のソフトウェア資産を有効利用できる可能性が高まる．本研究では，DAS モデルの基本機構を Java 言語のクラスライブラリ (以下，DAS クラスライブラリ) として設計・実装した．そして，実際に DAS クラスライブラリを用いて Java のアプリケーションに動的適応可能性の導入を行なった．これらの実装を通じて，DAS モデルが既存言語およびそのソフトウェア対して高い親和性を持つことを実証した．

本研究の手法において，ソフトウェアの柔軟な動的適応可能性を実現するには，適応可能手続きの制御機構を実行環境の状態に応じて柔軟に変更できることが求められる．DAS モデルでは，この制御機構自体が適応可能手続きを用いて自己反映的に実現される．そのため，この制御機構自体も実行環境の状態変化に適応することができる (実行環境の状態変化に応じて変化する)．実行環境の状態変化に応じて，この制御機構を変えることで，ソフトウェアの柔軟な適応動作を実現することが可能となる．たとえば，実行環境の状態に応じたソフトウェアの機能の一時的な制限やその解除といった適応動作を，プログラムを修正することなく実現することができるようになる． さらにこの制御機構が，適応可能手続きを用いて実現されるため，この機構のカスタマイズは，その構造を修正することなく，適応可能メソッドのインクリメンタルな追加で実現できる．これらの利点について，DAS クラスライブラリを用いた例題の実装を通して，実際に確かめることができた．また，ソフトウェアにおける柔軟な適応動作の有効性も実際に例題を用いて示した．

DAS モデルにもとづく動的適応可能なソフトウェアは，メタレベルアーキテクチャを形成する．そのメタレベルは，ソフトウェアにおける適応動作の制御を実現する機構の記述からなり，ベースレベルは，各ソフトウェアにおける問題領域の記述からなる．つまり，適応動作の制御機構は，ベースレベルの記述から完全に分離され，メタレベルに隠蔽される．この記述の分離は，動的適応可能なソフトウェアの可読性，保守性，拡張性を向上させる．この利点についても例題の実装を通して，実際に確かめることができた．

1.4 本論文の構成

本論文の構成は次のとおりである．2 章では，DAS モデルを実現する上で，基礎となる総称関数と自己反映計算の概念について概説する．3 章では，ソフトウェアの適応動作について事例をあげ，考察する．つまり実行環境の違いやその動的な状態変化がソフトウェアにもたらす影響と，それに対応する有効な処理について考察する．4 章では，DAS モデルについて詳述する．本研究では，DAS モデルの基本機構を Java 言語によるクラスライ

ブラリとして実現した．5 章では，この Java 言語によるクラスライブラリ (DAS クラスライブラリ) について概説し，6 章では，DAS クラスライブラリを用いた動的適応可能なソフトウェアの作成方法について述べる．7 章では，本研究に関連する研究について言及し，本研究との比較を行う．最後に 8 章で，本研究の結論と今後の課題について述べる．

第 2 章

準備

本章では，本研究で提案している動的適応可能なソフトウェアモデル DAS を実現する上で，基礎となる総称関数および自己反映計算の概念について概説する．

2.1 総称関数

総称関数とは，引数の型に関して一般化をおこなった関数であり，多重定義の一種である．言語処理系は，実引数の型から，どの関数を呼び出すかを決定する．このような総称関数を支援する言語として，Ada [14] や CLOS (Common Lisp Object System) [48, 22, 24] などがある．本節では，CLOS における総称関数について概説する．

CLOS における総称関数は，同じ名前を持つ複数のメソッドから構成される (図 2.1)．各メソッドは，それぞれ異なる引数を持ち，独自の操作を実装している．CLOS における総称関数は，これらメソッドへのインタフェースであり，総称関数が呼ばれると，CLOS の処理系は，その引数のクラスを特定し，そのクラスを仮引数として持つメソッドを選択し，呼び出す．このように，どのメソッドを呼び出すかを決定し，それを呼び出す機構を総称的ディスパッチと呼ぶ．

このような総称関数に対して通常の Lisp 関数は，それが行なう操作のインタフェースと実装が一体となっている．通常の Lisp 関数が呼ばれた場合，CLOS の処理系は，その関数を実装している単一のコード本体を直接呼び出す．通常の Lisp 関数のコードが，その呼び出しごとに変化することはないのに対して，総称関数のコードは，その呼び出しごとに (引数のクラスに応じて) 変化する．しかし，呼び出し側にとって総称関数は，通常の Lisp 関数と同じように見える．関数呼び出しに関する構文は同一であり，関数呼び出しにおいて，その関数が通常の Lisp 関数であるか，総称関数であるかを意識する必要はない．また，

通常の Lisp 関数とメソッドの違いは，メソッドは，直接呼び出されることがなく，総称的ディスパッチ機構によってのみ呼び出される点である．

このような総称関数の基本概念は，実行環境の状態変化に対処するソフトウェアの適応動作として応用できる．つまり，ソフトウェアの適応動作を実行環境の状態にもとづく総称関数として実現する．このような総称関数は，各実行環境の状態に応じた処理を実装するメソッドから構成される．そして，その呼び出しにおいて，実行環境の状態に応じた適切なメソッドが選択・実行される．こうした総称関数は，その呼び出しに関して，それを呼び出す側には，実行環境の状態変化を隠蔽することができる．このような総称関数を実現するには，引数の数や型のみによるメソッドの選択ではなく，実行環境の状態を考慮してメソッドの選択を行なう必要がある．つまり，実行環境の状態にもとづく総称的ディスパッチを実現する必要がある．

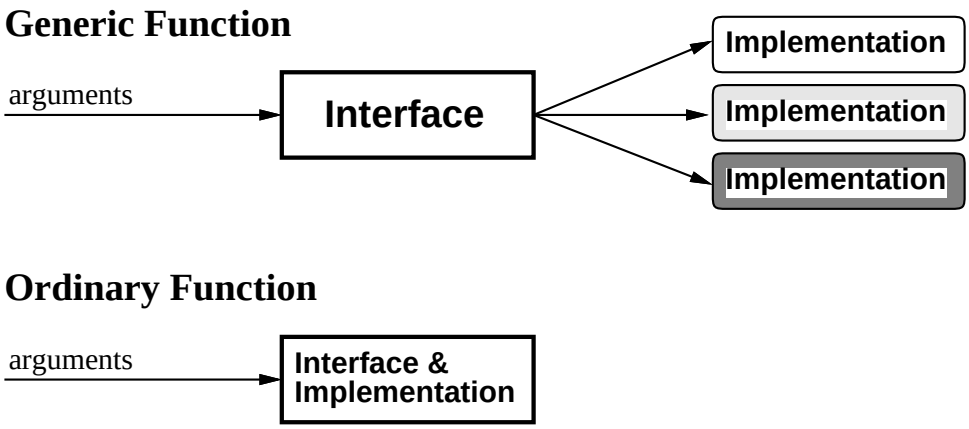

図 2.1: CLOS における総称関数と通常の Lisp 関数

2.2 自己反映計算

自己反映計算 [46, 32, 33, 54] とは，計算システムが自分自身の計算過程に関して行なう計算を指し，言語システムやオペレーティングシステムなどのソフトウェアシステムに対してモジュール化手法を提供する一種の構成原理として有効である．一般に，自分自身の構造や行動を観測する能力を内省 (introspection) と呼ぶ．計算システムにおいて，内省とは，システムが自己の構成/計算過程をモデル化した表現を内部に持ち，それを参照できる (自己参照) 能力であると言える．一方，自己反映的な計算システムは，このような内省の

能力だけでなく，自己の構成や計算過程の表現を変更する能力 (自己改変) も持つ．こうした自己の表現に対する変更は，自己の構成/計算過程に反映される．

一般に，計算システムは，その内部で計算の対象とする問題領域における具体的・概念的実体を，データやオブジェクトといった形の表現として近似的に扱っている．つまり，計算システムを Sとした場合，その問題領域の実体 Dに関する計算とは，実際には，S 内での D の表現 R_Dに対する操作として実現される (図 2.2)．以上から，計算システム S における自己反映計算とは，S がそれ自身の表現 R_S を内部に持ち，それに関する計算を行なうことであると言える．この時，R_S について以下のことが要求される．

1. R_S は，計算システム Sが表現されているアーキテクチャにもとづいて表現され，S 内で，参照/操作が可能でなければならない．
2. 計算システム S の状態は，計算の進行とともに変化するため，R_S もこれに応じて変化する必要がある．さらに，S 内で，R_S に対する操作が行なわれた時，それが S 自身にも反映しなければならない．

特に，上記 2 のような特性は因果的結合 (causal connection) と呼ばれる．つまり，自己反映的な計算システムでは，計算システムとそれが内部に持つ自分自身の表現との間に因果的結合を持つ．

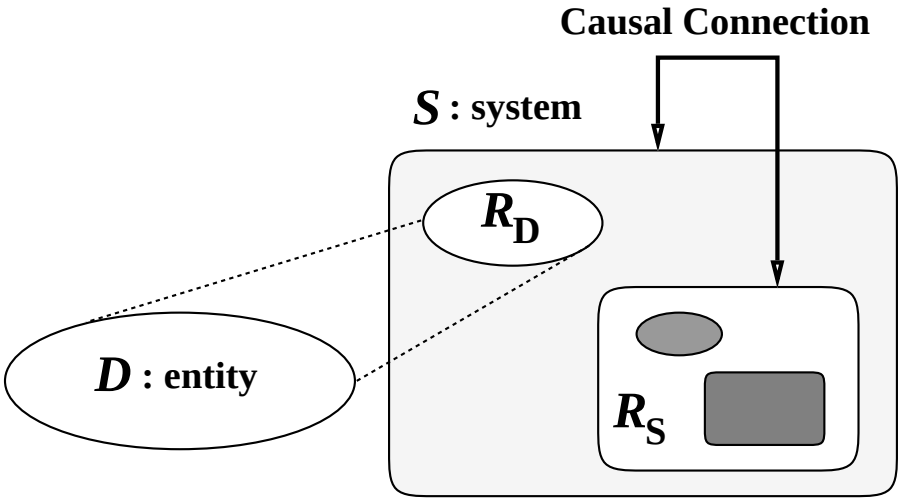

図 2.2: 自己反映的な計算システムの概念図

また，自己反映的な計算システムは，本来目的とする計算を行なうレベル (ベースレベル) と，その計算を制御/カスタマイズするレベル (メタレベル) の 2 階層から構成される

メタレベルアーキテクチャを形成する[1]．メタレベルは，その内部にベースレベルの表現を持ち，両レベルのインタラクションは同一言語にもとづいて行なわれる．ベースレベルからメタレベルの情報を取り出す操作を具体化 (reification)，メタレベルにデータを格納することによる変更の反映を狭義の自己反映 (reflection) と呼ぶ．

このような自己反映計算は，移動計算機環境などに対する動的適応可能なソフトウェアの構築技法として有効であると考えられる．つまり，移動計算機環境のように構造が動的に変化するような実行環境では，自己の構造や計算過程を実行環境に合わせて適切に変更し，実行環境に動的に適応できるようなソフトウェアが有効である．自己反映計算の機構を用いることで，このようなソフトウェアを自然な形で実現することが可能になる．

2.3 まとめ

総称関数と自己反映計算の基本概念について概説した．総称関数に関しては，CLOS における総称関数について概説した．CLOS における総称関数は，異なる操作を実装する複数のメソッドから構成され，その呼び出し時に実引数の型に応じて適切なメソッドが選択され，実行される．

また，自己反映計算とは，計算システムが自己の構成や計算過程に関して行なう計算である．自己反映的な計算システムは，一般にメタレベルとベースレベルから構成され，両レベルによる参照・変更の反映が相互に影響を与えあう因果的結合を持つ．

こうした総称関数や自己反映計算の基本概念は，移動計算機環境のような構造が動的に変化する可能性のある実行環境に対して有効であり，これらの概念は，動的適応可能なソフトウェアの構成技法に応用できる．

[1] 概念的には，メタレベルをベースレベルととらえるメタメタレベルなど，無限の階層構造 (リフレクティブ・タワー) を形成する．

第 3 章

事例 :ソフトウェアの適応動作

本章では，多様な実行環境とその動的な状態変化がソフトウェアにもたらす影響と，それに対処する有効な処理について考察する．なお，以下にあげるソフトウェアの事例は，その実行環境として携帯可能なノートブック型パソコン (以下，ノートパソコン) や携帯情報端末 (以下，PDA) を想定している．ノートパソコンや PDA は，デスクトップ型パソコンやワークステーションなどと比べて，物理的/論理的な構成要素やその状態の変化が抜本的かつ急激に起こる可能性が高く，本研究で扱う対象として適当であると考えられるからである．

3.1 GUI アプリケーション

一般に，ノートパソコンや PDA のディスプレイは，デスクトップ型パソコンのそれと比べて非常に小さい．ディスプレイの大小は，グラフィカルユーザインタフェース (GUI) にもとづくアプリケーション (以下，GUI アプリケーション) を使用する際，その look & feel および操作性に大きな影響を与える．たとえば，GUI アプリケーションを小さなディスプレイのノートパソコンや PDA で使用する場合，その look &feel や操作性は低下すると考えられる．その典型的な問題として，以下のような点があげられる．

- GUI アプリケーションのウインドウやダイアログが大きすぎて，小さなディスプレイに収まらない．
- フォントサイズが大きすぎて，見にくい．

これらの問題は，GUI アプリケーションの単なる look & feel の問題にとどまらない．ディスプレイのサイズと比較して，GUI アプリケーションのウインドウやダイアログが大きす

ぎる場合，その操作にも問題が生じる．たとえば，ディスプレイからはみ出た部分にボタンがあった場合，それを操作することができなくなる．またフォントサイズが大きすぎる場合，ウインドウ内に収まる文字数が必然的に少なくなり，何度もウインドウをスクロールする必要がでてくる，などといった問題が起こりうる．

このため，GUI アプリケーションのウインドウ，ダイアログ，文字，ボタンなどは，そのサイズを各計算機のディスプレイサイズに応じて適宜変えられることが望ましい．また，ボタンなどの部品は，そのサイズの変更だけでなく，その配置もディスプレイサイズに応じて変えられることが望ましい．というのは，単にボタンのサイズを変えただけで，その操作性が向上することはあまり期待できないからである．こうした GUI アプリケーションの適応動作を行なう主な方法として，以下が考えられる．

- ウインドウマネージャがすべての GUI アプリケーションの適応動作をとり行なう．
- 各アプリケーションがみずから適応動作を行なう．

ウインドウマネージャでこうした問題の一切に対処し，各アプリケーションはこれらに関与しないことが理想的ではある．しかし，ウインドウマネージャで汎用的にできる処理には限界がある．たとえば，メインウインドウやフォントのサイズ変更程度であれば，ウインドウマネージャだけで汎用的に対処可能であろう．しかし，アプリケーション中の個々のダイアログなどについてまで，ウインドウマネージャで管理することは困難である．その上，ウインドウやダイアログのサイズを変えただけで，問題が解決する場合は少なく，それらのサイズ変更にともない，ウインドウやダイアログを構成する部品の配置などを適宜変更しない限り，アプリケーションの look & feel および操作性は向上しない．そうした個々のアプリケーションの内部に及ぶ処理を，すべてウインドウマネージャだけで外部から汎用的に制御することは極めて困難である．それゆえ，各 GUI アプリケーション自身がこうした適応動作を自ら行なう必要がある．

3.2 ネットワークアプリケーション

ネットワークインフラストラクチャの整備と携帯可能な計算機の出現により，計算機を持って移動し，移動先からネットワークに接続して電子メールを読む，などといったことが可能になった．このような場合，ネットワークの通信メディアは，移動した先々の計算機環境に応じて変えられる．たとえば，移動した先に電話回線しかない場合は，必然的にモデムを使って電話回線からネットワークに接続することになる．このような計算機の移動

にともなって生じる問題のひとつにIP アドレスの問題がある．移動先でネットワークに接続するには，IP アドレスを取得/変更する必要がある．この問題に対処するために，これまでにVIP [50, 51]，IEIF (Internet Engineering Task Force) MobileIP [42]，DHCP [10]といったネットワークプロトコルが提案されている．これらのプロトコルは，計算機におけるIP アドレスといった低レベルの論理的な状態を隠蔽する．つまり，これらのプロトコルを使用することで，移動するつどにIP アドレスの設定を変更しなければならないといった問題は回避できる．

しかし，これで問題のすべてが解決されるわけではない．ネットワークの通信メディアを変えることで，通信のバンド幅やエラー率などが変化し，それらの変化がアプリケーションの実行に与える影響は決して無視できない．たとえば，ある遠隔サイトのデータベースサーバにアクセスして，必要なデータを取得するネットワークアプリケーションを考える．このクライアントアプリケーションの仕様は，リモートプロシジャコール (RPC) を利用して，データベースサーバに検索文字列を送り，検索結果を受け取るものとする．そして，この検索結果を受け取るまで，データベースサーバとの通信は常に接続されたままの状態である，と仮定する．ここで，このアプリケーションを移動計算機環境で用いた場合について考察する．前述したように，移動計算機環境では，移動先の計算機環境に合わせて，ネットワークの通信メディアが適宜変えられる．たとえば，通信メディアが有線から無線に変更された場合，このアプリケーションはうまく動作しない可能性がある．有線通信と比べて無線通信はバンド幅が極めて狭く，エラー率も非常に高くなる．さらに一時的な回線切断などが増加するために，データベースの検索処理が完了する以前にネットワークの回線接続が遮断されてしまう可能性がある．こうした問題に対処するアプリケーションの適応動作として，以下が考えられる．

- 頻繁にサーバとの接続/切断を行なうようにする．
- 処理の形態をリモートプロシジャコール形式から遠隔評価形式 [47] に変更する．

もっとも単純な解としては，サーバ/クライアント間の接続/切断を頻繁に行なうことである．まず，データベースサーバに検索文字列を送った後，直ちに接続を断つ．その後，再度接続して結果を受け取る．単純な検索だけであれば，この方式で十分かもしれない．しかし，検索結果をもとに，さらに検索を行なうような複雑な検索処理を行なう場合は，やはり問題が残る．ネットワークを介したサーバ/クライアント間の接続/切断を何度も行なう必要が生じる．バンド幅が狭く，接続状態の不安定な無線通信では，このようなサーバ/クライアント間の接続/切断を頻繁に行なう処理は適切ではない．

これに対して，検索処理の形態をリモートプロシジャコール形式から，遠隔評価形式に変更した場合について考える．たとえば，ある程度まとまった単位の検索処理をスクリプト形式でサーバ側に送り，サーバ上でこれを実行するような処理に変更したとする．この場合，検索処理がどんなに複雑になっても，ネットワークを介したサーバ/クライアント間の接続/切断は 2 回 (スクリプトを送る時と，検索結果を受け取る時) で済むため，不安定な無線通信に対して，有効であると考えられる．この方式の短所は，回線接続の回数は少なくて済むが，その分，1 度に送るデータの量が必然的に大きくなる点である．しかし，これはデータを圧縮して送ることで，ある程度回避することができる．このような適応動作は，アプリケーション内部の処理に強く依存するため，アプリケーション側で行なう必要がある．

3.3 マルチメディアアプリケーション

ここ数年，動画や音声などを駆使したマルティメディアアプリケーションが，多数開発されるようになった．特に，WWW (World Wide Web) とそのブラウザの普及により，Web ページに Java で記述された動画アニメーションなどのコンテンツを多数見かけるようになった．ここでは，このようなコンテンツの Web ページをマルティメディアアプリケーションととらえることにし，PDA などを用いて，遠隔地からこれを表示 (動作) させることを考える．近年の技術革新により，ノートパソコンや PDA の性能は著しい進歩を遂げている．特にノートパソコンの中には，デスクトップ型パソコンと比較しても遜色のないものが見られる．しかし，実際にモーバイルコンピューティングなどで実用に供される PDA は，まだまだ性能の低いものが多い．その上，これらの PDA は携帯性という点から，その大きさや計算機資源などに物理的な制約が加わる．たとえば，手のひらサイズの PDA に 17 インチサイズのディスプレイを貼り付けることは物理的に不可能であり，これらは技術革新によっても回避することができない．

PDA を用いて，ネットワークに接続し，先にあげたようなマルチメディアコンテンツのページを表示した場合，現在のところ実用的な速度での表示 (動作) は望めない．一般に PDA では，その計算能力が低く，その計算機資源の制約が大きすぎるからである．この問題に対処する適応動作としては，以下が考えられる．

- 画像の表示間隔を長くとり，その間の画像を間引く．
- 必要でない機能を一時的に制限する．

各 PDA の CPU パワーやメモリ容量に応じて，動画像のフレーム数を適宜変更することは，この問題に対する有効な解となりうる．勿論，この方法は，PDA の計算能力や計算機資源の不足を解消するものではなく，スムーズな動画像の表示 (動作) は，やはり困難ではある．しかし，この方法により，実行時の負荷を減少させることができ，PDA の CPU パワーやメモリ容量に応じた動画像の表示 (動作) が可能となる．また，音声出力装置がついていない PDA の場合には，音声出力処理を行わない，などといったアプリケーションの機能の一時的な制限を行なうことで，実行時の負荷を軽減することも有効な対処方法としてあげられる．こうした適応動作を実現する際に，次のアプローチが考えられる．

- プロキシサーバを用いてすべてのアプリケーションの適応動作を制御する．
- 各アプリケーションが各自で行なう．

プロキシサーバを用いた場合の利点は，言うまでもなく，各アプリケーション側では，何も特別な処理をしなくて済む点である．しかし，その反面これらの処理を柔軟に行なうことが不可能になる．たとえば，各アプリケーション自身がこの処理を行なった場合，どの部分の画像フレームを間引くか，どの機能を制限するか，といった点について，各アプリケーションで決定することができる．しかし，プロキシサーバを用いる場合，各アプリケーションの都合を考慮することは難しい．また，プロキシサーバ方式では，ネットワーク上に多数のプロキシサーバを用意する必要がある．

3.4 移動コードシステム

近年，プログラムを他のサイトに動的に送って実行させる移動コード (あるいは，遠隔コード) にもとづくシステムが数多く提案されている．このような移動コードは，次世代の分散計算システムの枠組として注目を集めている．その代表的な例として，Java によるアプレットがあげられる．Java 以外の移動コードシステムとしては，Scheme48 [25]，CAML [2]，GNU Guile [1] などがある．また，モーバイルエージェントという名称で，コンピュータ間の自律的な移動能力を持った移動コードにもとづくプログラムシステムも数多く提案されている．こうしたモーバイルエージェントシステムとして，Telescript [59]，Aglets [31, 30]，Flage [29]，Plangent [39]，Agent Tcl [16]，April [36] などがある．移動コードにもとづくシステムは，以下のような理由で移動計算機環境に適している．

- 物理的な制約により，計算能力，計算機資源の十分でない移動計算機では，移動コードにより十分な計算能力，計算機資源を持った遠隔ホストにプログラムを送って実行させることで，動的な負荷分散を実現することができる．

- 無線などバンド幅の狭い通信メディアを使用する際，移動コードを利用することで，ネットワークを介したサーバ/クライアント間のインタラクションを削減できる．これは無線通信におけるネットワークの一時的な回線切断などへの対処として有効である．

移動コードやモーバイルエージェントでは，移動前後における計算機の機種やオペレーティングシステム (OS)，ハードウェア/ソフトウェア構成など，その他すべての要素が一度に変化する可能性がある．この変化が抜本的かつ急激であるため，これらのプログラムはうまく動作しない場合がありうる．通常，こうした移動コードシステムやモーバイルエージェントシステムでは，特定の計算機や OS などに依存しないプラットフォームを提供しており，モーバイルエージェントなどは，そのプラットフォーム上で動作する．しかし，これは必ずしも問題解決につながるとは言えない．

たとえば，GUI ベースのモーバイルエージェントを例とすると，3.1 節で述べた問題が発生する．すなわち，モーバイルエージェントは移動先のディスプレイサイズに応じた GUI を実現することができなくなる．モーバイルエージェントのプラットフォームは通常，そうした点について何ら考慮していない．この問題を解決するには，やはりモーバイルエージェント自身が移動先のディスプレイサイズに応じて，自らその GUI を変更する必要がある．3.1 節で述べたように，ウインドウマネージャ (あるいはモーバイルエージェントのプラットフォーム) だけで，各モーバイルエージェント固有の GUI を制御することは困難であるし，移動先の実行環境 (あるいはモーバイルエージェントのプラットフォーム) には，GUI の制御を集中管理するウインドウマネージャが存在しない可能性もあるからである．

3.5 まとめ

本章では，GUI アプリケーション，ネットワークアプリケーション，マルチメディアアプリケーション，そして移動コードシステムを例にあげ，多様な実行環境とその動的な状態変化がこれらのアプリケーションに与える影響を示した．さらに各アプリケーションが実行環境の状態に応じて振舞いを変えることで，その変化に有効に対処できることを示した．こうした実行環境の状態変化に対するアプリケーションの適応動作は，各アプリケーションで行なう必要がある．OS やプロキシサーバなどで，各アプリケーション固有の適応動作を外部から汎用的に支援することには限界があり，各アプリケーションの都合や意図を反映させた適応動作を実現することは困難である．実行環境の状態変化に対する適応動作は，各アプリケーションの処理の性質に依存する場合が多く，各アプリケーションが

実行環境の状態に対して，ある程度の適応の能力，すなわち動的適応可能性を持つ必要がある．

第 4 章

動的適応可能なソフトウェアモデル DAS

本章では，まず，本研究における動的適応可能なソフトウェアモデルの設計方針について述べ，この方針にもとづく DAS モデルを示し，その動的適応可能性について考察する．DAS モデルは，柔軟かつ自己反映的な動的適応可能性を持つ．DAS モデルにおける動的適応可能性の柔軟性，自己反映性についても本章で詳述する．

4.1 設計方針

動的適応可能性を持つソフトウェア，すなわち動的適応可能なソフトウェアは，実行環境の動的な状態変化に自ら振舞いを変えることで，柔軟に対処することができるソフトウェアである．動的適応可能性とは，実行環境の状態変化を認識し，その状態に応じて自らの振舞いを変えるソフトウェアの能力を意味する．さまざまな実行環境やその動的な状態変化に対して，柔軟な適応動作を行なう動的適応可能なソフトウェアを実現するにあたり，以下の点が問題となる．

- 実行環境の表現 :ソフトウェアに実行環境をどのように見せるか．
- 状態変化の取得方法 :実行環境の状態変化をどのように取得するか．
- 適応動作の単位と制御 :実行環境の状態変化に応じた (ソフトウェアの) 適応動作の単位とその制御をどのように設定するか．
- 柔軟な動的適応可能性 :ソフトウェアに柔軟な適応能力 (動的適応可能性) を持たせるにはどうすればよいか．

以下，上記の点について考察し，それに対する本研究の基本方針を述べる．また，本研究における実行環境の状態とは，以下のすべてを含むより広い意味で用いられる．

- ハードウエアの構成要素の状態．
- ソフトウェアの構成要素の状態．
- 計算機内部のプロセスの状態．
- プログラムの入出力データなどの状態．
- 計算機周囲の物理的/論理的な環境の状態 (計算機の位置や IP アドレスなど)．

4.1.1 実行環境の表現

近年の技術革新により，大容量のメモリ装置や 2 次記憶装置が安価に提供されるようになった．これにともない，より複雑な処理をコンピュータに行なわせることが可能になった．これを受けて，今日のソフトウェアは，以前とは比較にならないほど大規模かつ複雑になってきている．ソフトウェアが大規模かつ複雑化するにつれ，そのモジュール化が促進されるようになった．現在，ソフトウェアはさまざまなモジュール群の集合体として構成されることが一般的であり，オブジェクト指向パラダイムやコンポーネントウェアなどもモジュール化技法のひとつととらえることができる．

ソフトウェアのモジュール化については，さまざまな技法が提案されており，そのモジュールの単位は各技法により異なるが，概念的にモジュールとは，ある部分の処理を実現する，他から独立したプログラム片であると考えられる．各モジュールは独立した存在であり，独自の処理内容を持つ．それゆえ，各モジュールの抽象度も異なること場合が一般的である．たとえば，あるアプリケーション中に，システム情報を操作するモジュール A と，グラフィカルユーザインタフェースを実現するモジュール B があった場合，これらのモジュールが扱う情報や処理の抽象度は異なる場合が多い．モジュール A がシステムコールなどを用いて，低レベル (抽象度の低い) のシステム情報を処理するのに対して，モジュール B はライブラリ関数などを用いて，抽象度の高いユーザよりの情報を処理するのが一般的である．

動的適応可能なソフトウェアを実現するには，ソフトウェアから実行環境が見え，ソフトウェアがその状態変化を認識できる必要がある．さまざまな抽象度のモジュールから構成されるソフトウェアに対して，実行環境の見え方が一様であった場合，あるモジュールにとって，その実行環境から得られる情報は有効でない可能性が高い．たとえば，実行環境の抽象度を高いレベルに設定した場合，低レベルなシステムの処理を行なうモジュール

にとって，そこから得られる情報は有効ではない．必要とする低レベルな実行環境の情報が欠落しているからである．逆に，実行環境の抽象度を低いレベルに設定した場合，抽象度の高いユーザよりの処理を行なうモジュールにとって，そこから得られる情報は有効ではない．得られる情報の抽象度が低すぎて扱いにくいからである．それゆえ，各モジュールの抽象度に応じた実行環境を提供することが望ましい．

以上から，ソフトウェアを構成する各モジュールごとに，その抽象度に応じた仮想的かつ局所的な実行環境を提供する必要があると結論できる (図 4.1)．こうした仮想的かつ局所的な実行環境は，それぞれ独自の抽象度を持つが，言うまでもなく実際の実行環境の状態を反映するものでなければならない．

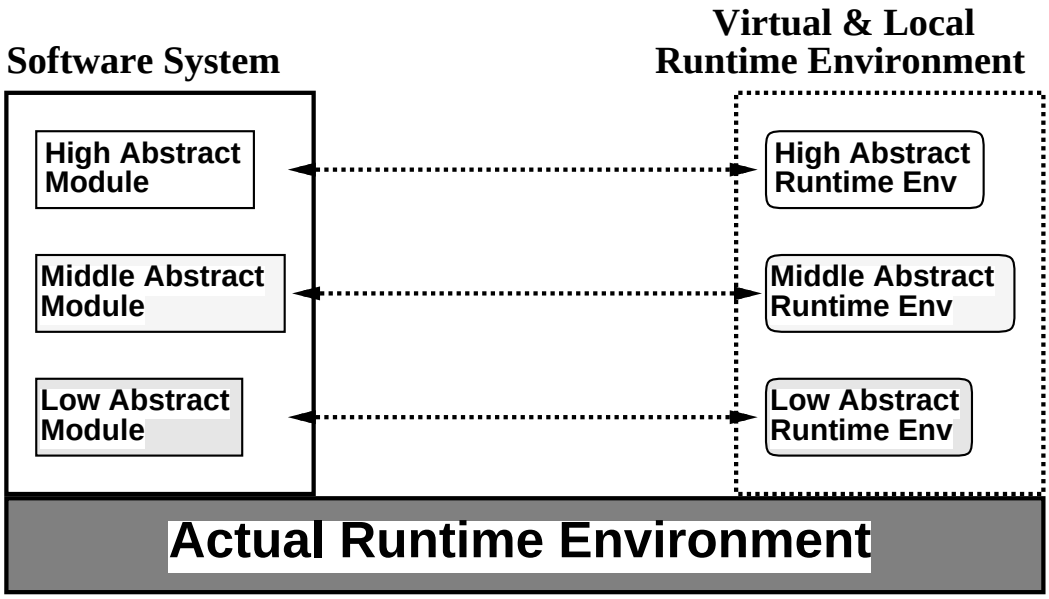

図 4.1: 各モジュールの抽象度に応じた仮想的かつ局所的な実行環境

4.1.2 状態変化の取得方法

動的適応可能なソフトウェアを実現するには，ソフトウェアが実行環境の状態変化を取得できる必要がある．実行環境の状態変化を取得する方法として，以下が考えられる．

- 状態取得関数による取得．
- 非同期イベントによる取得．

まず，システムコールのような実行環境の状態を取得する関数を用意し，各アプリケーションがこれを呼び出すことで，実行環境の状態変化を取得する方法が考えられる．また，実行環境の状態変化を監視し，その変化の発生を非同期なイベントとしてアプリケーションに通知するイベントモジュールを用いる方法も考えられる．後者は，特に実行環境の状態変化を即座に知る方法として有効である．前者があれば，後者を擬似的に実現することも可能であるが，この場合，各アプリケーションがポーリングして実行環境の状態変化を監視しなければならず，実行効率の低下と記述の繁雑さを招く結果となる．非同期イベントモジュールを提供することで，実行環境の状態変化をトリガとする状態変化駆動型の動的適応可能なソフトウェアが，効率良くかつ簡潔に記述できるようになる．つまり，これらの方法は相補的なものであり，アプリケーションの処理の性質に応じて，適宜使い分けられるべきものである．

以上から，動的適応可能なソフトウェアを実現するために，これら両方の状態変化取得方法が必要であると結論できる．

4.1.3 適応動作の単位とその制御

実行環境の状態変化が発生した時，動的適応可能なソフトウェアは何らかの適応動作を起こす．すなわち自らその振舞いを状態変化に応じて変える．この適応動作の単位とその制御を決定することは，動的適応可能なソフトウェアを実現する上で重要なポイントである．この問題は，次にあげる観点から十分に検討する必要がある．

- 柔軟な適応動作を実現するために，大規模から小規模まで柔軟な規模の適応動作が実現できる単位が望ましい．
- 適応動作の記述的な側面からみて適当な単位であることが望ましい．
- 適応動作の独立性・モジュール性からみて適当な単位であることが望ましい．
- 既存言語により記述されたソフトウェアとの親和性からみて適当な単位であることが望ましい．
- 速やかな適応動作を実現するために，適応動作の制御は低コストであることが望ましい．

これらの点から，ソフトウェアにおける適応動作の単位として以下が適切であると考えられる．

- ソフトウェア中の手続き．
- ソフトウェア中のオブジェクト．

適応動作の単位を手続きとした場合，適応動作の制御は手続き呼び出しにもとづいて実現される．ある手続きが呼び出された場合，実行環境の状態に応じて実際に呼び出される手続きを動的に切り替える．この手法では，適応動作の制御に要する時間的なコストは，実行環境の状態を識別する時間と，その状態に適切な手続きを選択する時間の和である．この時間的なコストは，状態に応じた手続きの数にもとづく線形関数で表現できる．これは，手続きの数が増えても，その制御に要する時間的なコストはリニアに抑えられることを意味している．また，適応動作の単位を手続きとする方式は，原理的に手続き呼び出しの機構を持つ既存言語により記述されたソフトウェアへの適用が可能である．しかし，この方式は，手続き呼び出しにもとづくため，アプリケーション側から手続き呼び出しを行なわない限り，適応動作を実現することができない．つまり，実行環境の状態が変化しても，アプリケーションが手続き呼び出しを行なわない限り，状態変化に対処することができない．

適応動作の単位としてオブジェクトも適切な単位であると考えられる．ここで，オブジェクトとは，属性を保持する変数と変数の値を操作するメソッドから構成されるプログラムモジュールである．適応動作の単位をオブジェクトとした場合，その制御方法 (すなわち実行環境の状態に応じてオブジェクト単位で，その切り替えを行なう) には，以下が考えられる．

- メソッド呼び出しにもとづく制御．
- 実行環境の状態変化にもとづく制御．

メソッド呼び出しにもとづく制御は，手続き呼び出しにもとづく制御と類似している．しかし，両者には大きな違いがある．後者では，手続きのみ切り替えられるのに対して，前者では，メソッド呼び出しが行なわれた時，当のメソッドだけでなく，それを含むオブジェクト全体が切り替えられる．また，実行環境の状態変化にもとづく制御は，オブジェクトの切り替えを制御するシステムモジュールによって実現される．このシステムモジュールは，実行環境の状態を監視し，状態変化を検出した際，関連するオブジェクトの切り替えを行なう．つまり，この方式はメソッドを呼び出すことなくオブジェクトの切り替えを実現する．

こうした適応動作の単位と制御の方式には，それぞれ一長一短がある．オブジェクトを単位とした場合，複数のメソッド (手続きとほぼ等価とする) を一度に切り替えられると

いう利点があるが，オブジェクトという明示的なプログラム単位を提供していないプログラミング言語（すなわち非オブジェクト指向言語），あるいは，そのような枠組に沿って構成されていないソフトウェアにこの手法を適用することは困難である．つまり適応動作の単位をオブジェクトとする方式は，オブジェクトという概念を持つプログラミング言語やシステムに特化した方式であり，その適用範囲は制限される．

また適応動作の単位をオブジェクトとし，先にあげたようなシステムモジュールによるオブジェクトの切り替えを行なう方式は，メソッドを呼び出すことなく，オブジェクトを切り替えることができるという利点があるが，オブジェクトの内部状態（ステート）に関する問題を引き起こす．つまり，オブジェクトを切り替える時，オブジェクトのステートを実行環境の状態変化前後で，どのように扱うか，と言う問題が起こる．具体的には，状態変化の前後で必要に応じてオブジェクトのステートを変換しなければならない．しかし，このようなオブジェクトのステートに依存するような処理を適切に記述することは，切り替えの単位によらず困難であり，仮にそのような処理を記述する場合，結局のところ切り替えの単位，すなわちオブジェクトの外部の状態として扱わざるを得なくなる．

以上の理由から，適応動作の単位を手続きとし，その制御は手続き呼び出しにもとづいて実現する方式が適切であると結論する．手続きを単位とした場合，一度の適応動作で一つの手続きの切り替えしか行なうことができないが，これはそれほど短所とは言えない．というのも，実際に呼び出されていない手続きの切り替えを行なう必要はないからである．またこの方式では，手続きが呼び出されない限り，適応動作が実行されず，実行環境の状態変化に即座に適応することが困難であるとも考えられるが，これも以下のようなシステムモジュールを設けることで回避できる．つまり実行環境の状態変化を監視し，状態変化を検出した時，関連する手続きを呼び出すシステムモジュールを提供する．より重要なことは，オブジェクトのようなモジュールの内部状態を考慮する必要がなく，オブジェクトの概念を持たない（非オブジェクト指向）のプログラミング言語やシステムにも適応可能な方式である点である．手続き呼び出し（オブジェクト指向におけるメソッド呼び出し，あるいはメッセージ通信）の機構を持つプログラミング言語であれば，原理的にこの方式を適用することができる．これにより，既存のソフトウェア資産を有効に利用した動的適応可能なソフトウェアの作成が可能になる．これは，既存のソフトウェア資産の有効利用という点だけでなく，ソフトウェアの開発コストを軽減することにも貢献する．

4.1.4 柔軟な動的適応可能性

プログラマが事前に予測できるソフトウェアの実行環境とその状態変化には限界があるため，プログラマが事前にソフトウェアに持たせることができる動的適応可能性にもおのずと限界がある．このため，ソフトウェアの動的適応可能性は，拡張が可能な構造で実現され，事前に予測できなかった実行環境やその動的な状態変化に対して，拡張/変更できる必要がある．そのような柔軟な動的適応可能性を実現するには，以下の機構を導入することが有効である．

- メタレベルアーキテクチャ．
- 実行環境の状態にもとづく総称手続き（適応可能手続き）．
- 自己反映計算を用いた適応可能手続きの制御機構．

ソフトウェアにおける動的適応可能性の実現機構をメタレベルアーキテクチャとして構成することは，その拡張性だけでなく，可読性，保守性の観点からも有効である．この場合，メタレベルは適応動作の制御記述からなり，ベースレベルは対象のソフトウェアが本来目的とする問題領域の記述からなる．適応動作の制御記述は，問題領域の記述とは本来無関係なものであり，両者の混在は，プログラムの可読性を低下させ，その保守と拡張を困難にする．そのため，メタレベルアーキテクチャを用いて，両者を分離することには大きな意味がある．両者をそれぞれ独立して修正・変更することができるようになるからである．

本研究では，適応動作の単位を手続きとし，実行環境の状態に応じて呼ばれる手続きを動的に切り替えることで，ソフトウェアの動的適応可能性を実現する．このような手続きにもとづく適応動作の実現には，実行環境の状態を考慮した総称手続きが有効である．ここで言う総称手続きとは，CLOS における総称関数に実行環境の状態という要素を加味したものと言うことができる．このような総称手続きは，実行環境の状態に応じた実行コードを持つ複数のメソッドから構成され，その呼び出しにおいて実行環境の状態に応じたメソッドが動的に選択・実行される．このような総称手続きの定義および制御機構により，手続きにもとづくソフトウェアの適応動作を独立したモジュールとして記述・制御できるばかりでなく，事前に予測できなかった実行環境の状態にあとで対処する必要がでてきた場合にも，その制御構造を修正することなく，メソッドのインクリメンタルな追加により対処できる．これを条件分岐によるアドホックな実装方式と比較するならば，その利点は明白である．条件分岐により実行環境の状態を場合分けし，呼び出す手続きを切り替えるア

ドホックな実装方式では，時として条件分岐の制御構造を大きく変更する必要にせまられる場合があるからである．

実行環境にもとづく総称手続きを実現するには，CLOS における総称関数よりも多少複雑な機構が必要となる．CLOS の総称関数におけるメソッドディスパッチが，メソッドの引数やその型に応じてなされるのに対して，実行環境の状態を考慮したメソッドディスパッチが必要になるからである．こうした総称手続きは，実行環境の状態に対する条件を内部に持つメソッドと，そのディスパッチにおいて，各メソッドの持つその条件を評価するメソッドディスパッチの機構から実現できる．このような総称手続きを本研究では，適応可能手続き，そのメソッドを適応可能メソッドと呼ぶ．

そして，より柔軟な動的適応可能性を実現するには，その制御方式を実行環境の状態に応じて柔軟に変更できるような機構が有効である．ここで言う制御とは，実行環境の状態に応じた適応可能メソッドのディスパッチ方法 (方針) である．本研究では，この方針を適応の戦略と呼ぶ．事前に予測可能な実行環境とその状態には限界があるため，それに対処する事前に定義可能な適応可能メソッドも限定される．適応の戦略が固定的で柔軟性に欠ける場合，多様な実行環境の状態に対処することは困難である．さまざまな適応の戦略を用意し，状況に応じてそれらを適宜使い分けることができれば，有限の適応可能メソッドをより有効活用することができるようになる．このような柔軟な適応可能手続きの制御機構を実現するには，自己反映計算が有効である．つまり，この制御機構そのものを適応可能手続きを使って，自己反映的に実現するのである．具体的には，適応の戦略を適応可能手続きとし，個々の戦略は適応可能メソッドとして実装する．そして実行環境の状態に応じて，このメソッドディスパッチを行なうことで，戦略を変更する．この自己反映計算を用いた制御機構については，4.5 節にて詳述する．

4.2 DAS モデル

本節では，前節で述べた設計方針にもとづく動的適応可能なソフトウェアモデル DAS について述べる．DAS モデルは，以下の要求を満たすソフトウェアモデルである．

- 各手続きの抽象度に適した仮想的かつ局所的な実行環境を表現する環境オブジェクトを提供する．環境オブジェクトは実際の実行環境を反映し，アプリケーション中で任意の数だけ定義し，生成することができる．個々の環境オブジェクトは，その状態を保持する変数と，それを操作するメソッドから構成される．

- 環境オブジェクトの状態変化を監視するイベントオブジェクトを提供する．イベントオブジェクトは，環境オブジェクトの状態変化を監視するだけでなく，その状態変化を検出すると，この環境オブジェクトを利用しているアプリケーション中に定義された所定の (変化に対処するための) 手続きを呼び出す．本研究では，環境オブジェクトの状態変化を状態イベント，状態イベントに対処する手続きをコールバックメソッドと呼ぶ．

- 手続きにもとづく動的適応可能性を実現する機構を提供する．これは，アプリケーション中の手続きを環境オブジェクトの状態にもとづく総称手続き (適応可能手続き) として定義する機構と実行環境の状態に応じたメソッド (適応可能手続き中のひとつの実装，すなわち適応可能メソッド) を選択・実行する機構からなる．特に後者を適応の機構と呼ぶ．適応の機構において本質的な部分は，適応可能メソッドを選択する方針，すなわち適応の戦略である．適応の戦略は，環境オブジェクト単位に設定される．

- 自己反映計算を用いた適応の機構を提供する．つまり，適応の機構自体の実現に適応可能手続きを用いる．そのため，この適応の機構は自分自身を制御し，それ自身が環境オブジェクトの状態に応じて動的に変更される．適応の機構は，適応の戦略を実装する戦略オブジェクトと，戦略オブジェクトを用いて適応可能手続きを制御するディスパッチオブジェクトから構成される．戦略オブジェクトは，個々の戦略を実装する適応可能メソッドからなる適応可能手続きを持つ．つまり，適応の戦略は環境オブジェクトの状態により動的に変更される．

- DAS モデルにもとづくソフトウェアは，メタレベルアーキテクチャを形成する．メタレベルは適応動作の制御記述からなり，ベースレベルは対象のソフトウェアが本来目的とする問題領域の記述からなる．

このような DAS モデルは，オブジェクト指向パラダイムとの親和性が高いが，ここでは，他から独立したプログラムの単位としてオブジェクトという語を用いており，特にオブジェクト指向パラダイムにもとづく計算実体を指すものではない．つまり，DAS モデルはオブジェクト指向パラダイムに特化したモデルではなく，より広い適用範囲を持つソフトウェアモデルである．

このような DAS モデルにもとづくソフトウェアは，図 4.2 (右側) のようなメタレベルアーキテクチャを形成し，各レベルは以下の要素の記述から構成される．

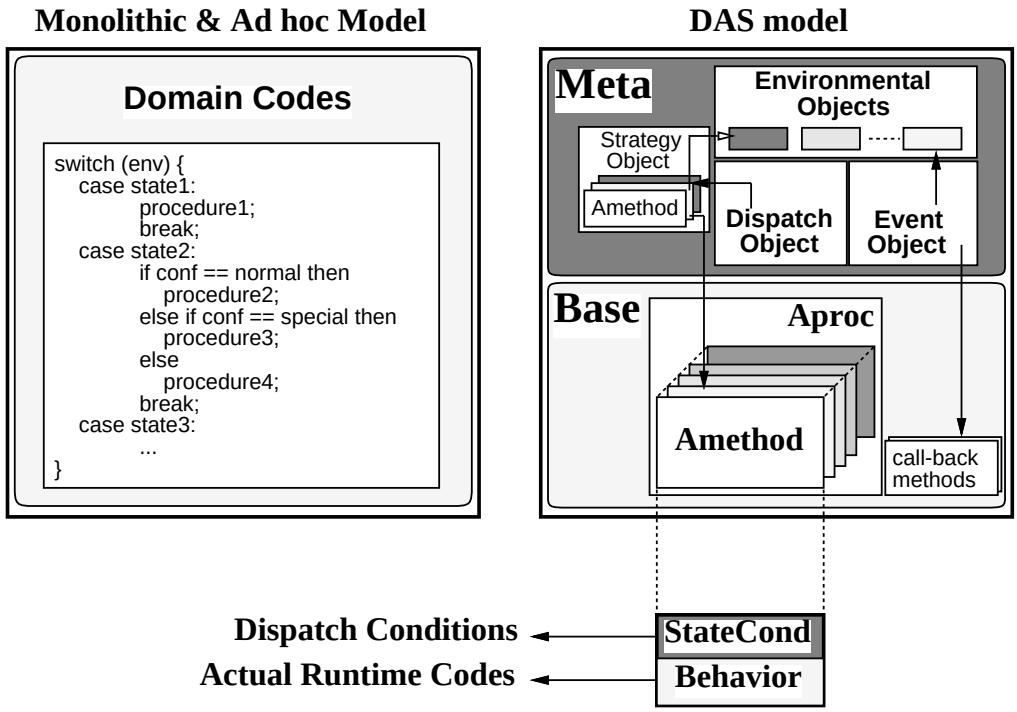

図 4.2: DAS モデルとアドホックなモデル

- メタレベル :環境オブジェクト, イベントオブジェクト, 戦略オブジェクト, ディスパッチオブジェクトの各記述.
- ベースレベル :問題領域, 適応可能手続き群, コールバックメソッド群の各記述.

アプリケーション中で, 実行環境の状態に応じて変化すべき処理は, 適応可能手続きとして定義される. ここで実行環境とは, 環境オブジェクトを指す. 適応可能手続きは, 環境オブジェクトの状態に適した複数の適応可能メソッドから構成される. 各適応可能メソッドは, 環境オブジェクトの状態に対する条件と実際の実行コードから構成される. 環境オブジェクトの状態に対する条件が, 各適応可能メソッドのディスパッチ条件となる. また, コールバックメソッドは, 状態イベントに対処するアプリケーション中に定義された手続きであり, 状態イベント (すなわち, 環境オブジェクトにおける状態変化) の発生時にイベントオブジェクトから呼び出される.

DAS モデルにもとづくソフトウェアの適応動作, すなわち適応可能手続きにもとづく動作は, 以下のように実現される. ここで, 適応可能手続きとその中の適応可能メソッド, この適応可能手続きの抽象度に適した環境オブジェクトが既に定義されているものとする. 簡

単化のため，ここでは，適応の機構における自己反映的動作については述べない．これについては後述 (4.5 節参照) する．

- 適応可能手続きは，環境オブジェクトを引数として呼び出される．適応可能手続きの呼び出しは，メタレベルのディスパッチオブジェクトによりキャッチされる．
- ディスパッチオブジェクトは戦略オブジェクトを生成し，その中に定義された適応の戦略を使って適応可能メソッドのディスパッチを行なう．この際，環境オブジェクトと各適応可能メソッド中のディスパッチ条件を参照し，適切な条件を持つ適応可能メソッドを選択する．
- 戦略オブジェクトによって選択された適応可能メソッドの実行コードがベースレベルで実行される．

また，イベントオブジェクトを利用した状態イベントにもとづく適応動作は，以下のように実現される．ここでも，適応可能手続き，その中の適応可能メソッドおよび環境オブジェクトは，既に定義されているものとする．

- ベースレベルからメタレベルのイベントオブジェクトを起動する．この時，イベントオブジェクトに，環境オブジェクトを渡す．
- イベントオブジェクトは，起動時に受け取った環境オブジェクトの状態変化を監視する．
- 環境オブジェクトの状態変化 (状態イベント) を検出すると，イベントオブジェクトは，その状態イベントに対応するアプリケーション中に定義されたコールバックメソッドを呼び出す．コールバックメソッドには，イベントオブジェクトが監視していた環境オブジェクトが渡される．
- コールバックメソッド中で，渡された最新の状態の環境オブジェクトを用いて，適応可能手続きの呼び出しを行なう．

このような DAS モデルの持つ利点として，以下があげられる．

1. 手続きの抽象度に適した動的適応可能性の実現が可能となる．
2. 手続き呼び出しの機構を持つ既存言語により記述されたソフトウェアへの適用が原理的に可能である．

3. 適応可能メソッドの追加による動的適応可能性の向上が可能となる．

4. 適応の機構の柔軟かつ動的なカスタマイズが，その制御構造を修正することなく，適応可能メソッドのインクリメンタルな追加により可能となる．

5. メタレベルの記述（適応動作の制御記述）の共有・再利用が可能となる．

6. 動的適応可能なソフトウェアの可読性，保守性，拡張性を促進させる．

上記 2 の利点は特に重要である．既存のソフトウェア資産を有効利用できるばかりでなく，動的適応可能なソフトウェアの開発にかかるコストを軽減することが可能となるからである．また，上記 3 と 4 の利点も重要である．事前に予測することができなかった実行環境の状態やその動的な変化に対処するためには，あとでソフトウェアを修正しなければならない．その際，条件分岐によるアドホックな実装のソフトウェア（図 4.2左側）は，その制御構造（条件分岐の構造）を修正する必要が生じる．しかし，DAS モデルにもとづくソフトウェアは，こうした制御構造の修正は一切必要なく，適応可能メソッドという独立した単位のモジュールをインクリメンタルに追加することで，拡張・修正を行なうことができる．特に適応の機構（ここでは，特に適応の戦略を指す）の拡張を行なう場合，アドホックな実装のソフトウェアでは，その制御構造を大幅に修正しなければならないことが多い．これに対して，DAS モデルでは，適応の機構の拡張でさえも適応可能メソッドのインクリメンタルな追加により実現することができる．DAS モデルにおける適応の機構（特に，適応の戦略）が，適応可能手続きとして実現されているからである．

このような DAS モデルは，特定のパラダイムや記述言語から独立した汎用的なモデルであり，3 章であげた GUI アプリケーション，ネットワークアプリケーション，マルチメディアアプリケーション，移動コード/エージェントなどに適用可能である．特に，移動コードシステムへの適用は有効な事例と考えることができる．移動コードにもとづくソフトウェアは，移動にともなう実行環境の状態変化が動的かつ抜本的であるため，移動後の実行環境に対する適応動作が求められるからである．しかし，現在提案されている移動コードシステムの多くは，実行環境への適応をほとんど考慮してはいない．DAS モデルはこうした既存の移動コードシステムに適用できる．

4.3 動的適応可能性

DAS モデルの動的適応可能性は，実行環境（環境オブジェクト）の状態，適応可能手続き，適応の戦略にもとづく．本節では，これらの定義を行なう．

定義 1　(実行環境の状態) 実行環境を構成する要素 (環境要素) は，有限個であり，ここでは，その数を N とする．そして $Value_i\ (i = 1, \cdots, N)$ を各要素のとり得る値 (状態) の集合とすると，実行環境の状態 $Rstate$ は，以下のように定義される．

$$Rstate = Value_1 \times \cdots \times Value_N$$

ただし，$1 \leq N$ ■

つまり，$Rstate$ は，ある実行環境における遷移可能な状態の集合であり，その元は，ある時点における実行環境の状態を示す．

定義 2　(適応可能メソッド) $StateCond$ を適応可能メソッドが選択される条件 (ディスパッチ条件) の集合，$Behavior$を振舞いの集合とすると，適応可能メソッドの集合 $Amethod$ は，以下のように定義される．

$$Amethod = StateCond \times Behavior$$

ここで，$StateCond$ は，

$$StateCond = Cond_1 \times \cdots \times Cond_N$$

のように定義される．$Cond_i\ (i = 1, \cdots, N)$ の要素は，i 番目の環境要素に関する条件であり，$Value_i$から $true\ (1)$，$false\ (0)$，$undefined\ (0.5)$ への関数として以下のように表現される．

$$Cond_i = Value_i \rightarrow \{true,\ \ false,\ \ undefined\}$$

■

定義 3　(適応可能手続き) $Amethod$ を適応可能メソッドの集合とすると，適応可能手続きの集合 $Aproc$ は，以下のように定義される．

$$Aproc = Amehtod^{+}$$

ここで，$Amehtod^{+}$ は，適応可能メソッド 1 個以上の順序の付いた並びであるとする． ■

定義 4　(適応の戦略) $Rstate$ を実行環境の状態の集合，$Aproc$ を適応可能手続きの集合，$Amethod$ を適応可能メソッドの集合とすると，適応の戦略の集合 $Strategy$は，以下のように定義される．

$$Strategy = Rstate \times Aproc \rightarrow Amethod$$

■

つまり，適応の戦略とは，実行環境の状態と適応可能手続きを引数にとり，その状態に合致する条件を持つ適応可能メソッドを返す関数である．言うまでもなく適応の戦略は重要である．戦略の実現方法によってソフトウェアの動的適応可能性が変化するからである．ここで例として，以下のような適応の戦略を定義する．

例 ： (各種戦略の定義) ある時点における実行環境の状態をσとし，その時点において適応可能手続きρが起動されるとする．ここで，$\sigma \in Rstate$，$v_j \in Value_j$，$\rho \in Aproc$，$m_i \in Amethod$，$\gamma_i \in StateCond$，$\beta_i \in Behavior$，$c^i_j \in Cond_j$のようであるとする (ただし，$1 \leq i \leq k,\ 1 \leq j \leq N$)．

$$\begin{aligned} \sigma &= (v_1, \cdots, v_N) \\ \rho &= (m_1, \cdots, m_k) \\ m_i &= (\gamma_i,\ \beta_i) \\ \gamma_i &= (c^i_1, \cdots, c^i_N) \end{aligned}$$

ここで適応の戦略とは，適応可能手続きρ中から実行環境の状態σに適した適応可能メソッドを 1 つ選択する方針である．ここでは，例として以下の戦略をあげる．

AND 戦略 :適応可能メソッドのディスパッチ条件が，

$$\forall j.c^i_j(v_j) = 1$$

となる (すべて真となる) ものを選択する．ただし，該当するものが複数存在した場合，添字 i の値が最小の適応可能メソッドを選択する． ■

OR 戦略 :適応可能メソッドのディスパッチ条件が，

$$\exists j.c^i_j(v_j) = 1$$

となる (1 つ以上の条件が真となる) ものを選択する．ただし，該当するものが複数存在した場合，添字 i の値が最小の適応可能メソッドを選択する． ■

このような **AND** 戦略や **OR** 戦略を用いた場合，ディスパッチ条件を満たす適応可能メソッドが存在しない場合，適応可能手続きは実行されない．これらの戦略は，実行環境のある状態においてのみ行なう必要のある特別な適応動作の実現に適している．

また，以下の戦略は各環境要素に優先順位にもとづく重み (優先度数) を付け，その総計により適切な適応可能メソッドを選択するものである．

正の優先度数戦略 :環境要素に重み w_j を付け，環境要素に対する条件が真となる時，その環境要素の重みの総計

$$\sum_{j=1}^{n} w_j \cdot \lfloor c_j^i(v_j) \rfloor$$

が最大である適応可能メソッドを選択する．ただし，該当するものが複数存在した場合，添字 i の値が最小の適応可能メソッドを選択する． ■

負の優先度数戦略 :環境要素に重み w_j を付け，環境要素に対する条件が偽となる時，その環境要素の重みの総計

$$\sum_{j=1}^{n} w_j \cdot (1 - \lceil c_j^i(v_j) \rceil)$$

が最小である適応可能メソッドを選択する．ただし，該当するものが複数存在した場合，添字 i の値が最小の適応可能メソッドを選択する． ■

これらの優先度数にもとづく適応の戦略は，環境要素間に優先順位を設定し，できるだけその優先順位の高い環境要素に対する条件が真となる適応可能メソッドを選択する (正の優先度数戦略)，あるいは優先順位の高い環境要素に対する条件が偽となる適応可能メソッドをできるだけ選択しない (負の優先度数戦略) という方針を基本としている．このような優先度数にもとづく適応の戦略は，実行環境のどのような状態に対しても，必ず適応可能メソッドを選択する．つまり，これらの戦略は，実行環境の状態に対して必ず行なう必要のある適応動作の実現に適している．

言うまでもなく，ここにあげた戦略はほんの一例に過ぎない．より重要なことは，さまざまな戦略をこのように定義できること，そして戦略の実現方法により異なる動的適応可能性を柔軟に実現できる点にある．

4.4 適応の戦略にもとづく動的適応可能性

DAS モデルでは，適応の戦略の実現方法によって柔軟な動的適応可能性を実現することができる．本節では，適応の戦略とそれにもとづく動的適応可能性について考察する．例

として，以下のような実行環境 (環境オブジェクト) の状態 r，適応可能手続き p および適応可能メソッド ($m_1,\ m_2,\ m_3$) を仮定する．

$$
\begin{aligned}
r &= \{v_1,\ v_2,\ v_3,\ v_4,\ v_5,\ v_6,\ v_7,\ v_8,\ v_9\} \\
p &= \{m_1,\ m_2,\ m_3\} \\
m_1 &= (s_1,\ b_1) \quad s_1 : (c_1^1(v_1),\ c_2^1(v_2),\ c_3^1(v_3)) = (1,1,1) \\
m_2 &= (s_2,\ b_2) \quad s_2 : (c_4^2(v_4),\ c_5^2(v_5),\ c_6^2(v_6)) = (1,1,1) \\
m_3 &= (s_3,\ b_3) \quad s_3 : (c_7^3(v_7),\ c_8^3(v_8),\ c_9^3(v_9)) = (1,1,1)
\end{aligned}
$$

実行環境のすべての状態を事前に知ることはできないので，予測可能な実行環境の状態は有限であり，それに適した適応可能メソッドの定義も有限となる．ここでは，$s_1,\ s_2,\ s_3$ を事前に予測可能な実行環境の状態，これらの状態に対応する適応可能メソッドをそれぞれ $m_1,\ m_2,\ m_3$ と仮定する．この仮定の上で，以下の適応の戦略について考察する．

1. $\boldsymbol{strategy_1}$: 環境要素の状態に対する条件が，すべて真となる適応可能メソッドを選択する．

2. $\boldsymbol{strategy_2}$: 環境要素の状態に対する条件が，すべて真となる適応可能メソッドを選択するが，該当するものがない場合，デフォルトの適応動作として適応可能メソッド m_1 を選択する．

3. $\boldsymbol{strategy_3}$: 環境要素の状態に対する条件の一部が真となる適応可能メソッドを選択する．

ここで，以下のような実行環境の状態 $state_1,\ state_2,\ state_3$ について考察する．この実行環境の状態は，プログラムの作成時にプログラマが予測した (予測可能な) 状態，すなわち $s_1,\ s_2,\ s_3$ とほんの少しばかり異なる状態を示している．

	θ_1	θ_2	θ_3
$state_1$	(1, 1, 0)	(0, 0, 0)	(0, 0, 0)
$state_2$	(0, 0, 0)	(1, 1, 0)	(0, 0, 0)
$state_3$	(0, 0, 0)	(0, 0, 0)	(1, 1, 0)

さらに，ここで，

$$
\begin{aligned}
\theta_1 &= (c_1^1(v_1),\ c_2^1(v_2),\ c_3^1(v_3)) \\
\theta_2 &= (c_4^2(v_4),\ c_5^2(v_5),\ c_6^2(v_6)) \\
\theta_3 &= (c_7^3(v_7),\ c_8^3(v_8),\ c_9^3(v_9))
\end{aligned}
$$

とする．この場合，上記の戦略を用いた適応可能メソッドのディスパッチは，以下のような結果になる．

$strategy_1$	$:(state_1,\ p) \rightarrow$
	$:(state_2,\ p) \rightarrow$
	$:(state_3,\ p) \rightarrow$
$strategy_2$	$:(state_1,\ p) \rightarrow m_1$
	$:(state_2,\ p) \rightarrow m_1$
	$:(state_3,\ p) \rightarrow m_1$
$strategy_3$	$:(state_1,\ p) \rightarrow m_1$
	$:(state_2,\ p) \rightarrow m_2$
	$:(state_3,\ p) \rightarrow m_3$

まず，$strategy_1$は，上記のいずれの状態においても適応可能メソッドを選択することができない．次に，$strategy_2$は，上記のいずれの状態においても必ずデフォルトの適応可能メソッドm_1を選択してしまう．しかし，$strategy_3$は，$state_1$に対して適応可能メソッドm_1を，$state_2$には適応可能メソッドm_2を，そして$state_3$には適応可能メソッドm_3をそれぞれ選択することができる．この例は，事前にプログラマが予測した (予測可能な) 実行環境の状態とほんの少しばかり異なる実行環境の状態に対して，$strategy_3$は，プログラム作成時にプログラマが意図した適応動作を実現するのに対して，$strategy_{1,2}$は，プログラマが当初意図した適応動作を実現しないことを示している．

ここで，重要な点は，各戦略によって選択される適応可能メソッドが変化する点と，このような戦略を動的に変更することによってソフトウェアの動的適応可能性に柔軟性を持たせることができる点である．たとえば，上記における実行環境の状態$state_1$では，戦略$strategy_3$を用いて，適応可能メソッドm_1を選択・実行することにする．そして状態が$state_1$から$state_3$に変化した際，適応の戦略を$strategy_3$から$strategy_1$に変更することにより，この適応可能手続きを実行不可にすることができる．これは，実行環境の状態に応じたソフトウェアの機能の一時的な制限を示している．また，状態が$state_3$から$state_1$に再度戻った時，適応の戦略を$strategy_1$から$strategy_3$に戻すことにより，この適応可能手続きは再び実行可能となる．これは，先に行なったソフトウェアの機能の一時的な制限に対する解除を示している．このように適応の戦略の変更は，有限の適応可能メソッドを有効利用することにつながる．こうした実行環境の状態にもとづく戦略の動的な変更も，適応の戦略が適応可能手続きとして実現されるため，容易に実現することができる．

4.5 自己反映計算を用いた適応の機構

DAS モデルにおける適応の機構は，自己反映計算を用いて実現される．本節では，この適応の機構における自己反映的な振舞いと自己反映計算にもとづく利点について述べる．

前述したように，適応の機構は適応可能手続きを制御する機構であり，戦略オブジェクトとディスパッチオブジェクトから構成される．戦略オブジェクトは，個々の戦略を適応可能メソッドとする適応可能手続きを持つ．つまり，適応の戦略自体がメソッドディスパッチの対象となる．DAS モデルにおける適応可能メソッドのディスパッチは，以下の 2 段階のディスパッチにより実現される (図 4.3)．

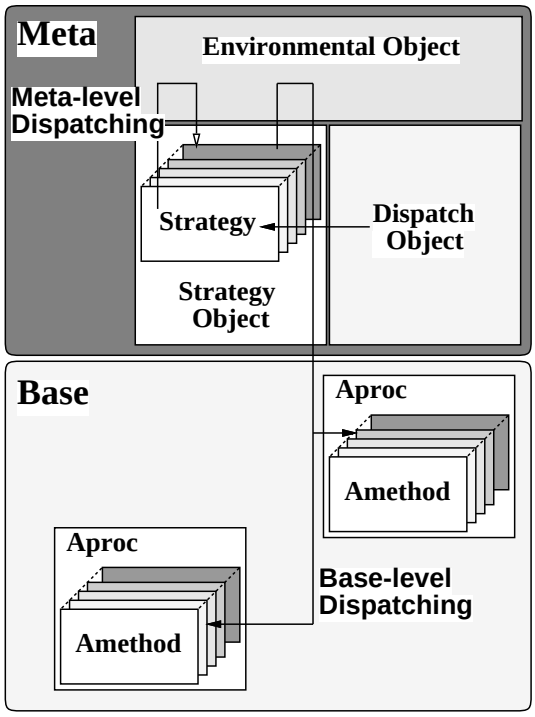

図 4.3: メタレベルディスパッチ & ベースレベルディスパッチ

- メタレベルディスパッチ :戦略オブジェクト中の適応の戦略を決定する．
- ベースレベルディスパッチ :ベースレベル中の適応可能メソッドを決定する．

ベースレベルから適応可能手続きの呼び出しを行なうと，メタレベルのディスパッチオブジェクトに制御が移り，ディスパッチオブジェクトが戦略オブジェクトを生成する．そ

して戦略オブジェクト中にある適応の戦略 (適応可能メソッド) のひとつを使用して，環境オブジェクトの状態に適した戦略のディスパッチを行なう．つまり，自分自身で自分のディスパッチを行なうことになる．これがメタレベルディスパッチである．ここで，ディスパッチオブジェクトがはじめに使用する適応の戦略は，ディスパッチオブジェクトの生成時に指定可能 (戦略名を表す文字列を指定) であり，デフォルトのメタレベルディスパッチは，環境オブジェクト中に定義された変数 (戦略変数) の値 (戦略名を表す文字列) にもとづいて実現されるものとする．しかし，これは固定的なものではなく，柔軟にカスタマイズすることができる．

そして，メタレベルディスパッチで決定された適応の戦略を用いて，ベースレベルから呼び出しのあった適応可能手続き中の適応可能メソッドをディスパッチする．これがベースレベルディスパッチである．

ベースレベルから適応可能手続き呼び出しを行なう時，制御がメタレベルに移り，適応の機構 (言い換えれば，適応の戦略) をデータとしてそれ自身の機構を用いて操作する (ディスパッチする) ことになる．これは 自己反映計算の用語で言う具体化に相当する．また，メタレベルからベースレベルに制御が戻る時，メタレベルディスパッチで選択された戦略が，適応の機構として設定される．これは同じく自己反映計算の用語で言う狭義の自己反映に相当する．メタレベルディスパッチの結果は，ベースレベルディスパッチすなわちベースレベルの適応動作に反映する．ベースレベルは，環境オブジェクト中の戦略変数の値を変える (別の戦略名を指定する) ことで，メタレベルの振舞いを変更することができる．

このように適応の機構を自己反映計算を用いて実現することで，以下のような利点が得られる．

- 環境オブジェクトの状態に応じて適応の機構を柔軟かつ動的にカスタマイズすることができる．
- メタレベルの機構をベースレベルと同一の方式で統一的に拡張・変更することができる．

前節で述べたように，環境オブジェクトの状態に応じて適応の機構を変えることで，有限の適応可能メソッドをより有効利用できるようになる．さらに，環境オブジェクトの状態に応じたソフトウェアの機能の一時的な制限やその解除といった柔軟な適応動作をベースレベルの修正なしに実現することが可能になる．

4.6 まとめ

本章では，本研究における動的適応可能なソフトウェアのモデルについて，その設計方針を述べ，この方針にもとづく DAS モデルについて述べた．DAS モデルの設計方針とは，以下のようなものであった．

- 仮想的かつ局所的な実行環境を提供する．
- 適応動作の単位を手続きとし，その制御は手続き呼び出しにもとづいて行なう．
- 実行環境の状態を取得する関数群および状態変化の発生を通知する非同期イベント機構を提供する．
- 柔軟な動的適応可能性を提供する．

そして，柔軟な動的適応可能性を実現するために以下の機構を導入することとした．

- メタレベルアーキテクチャ．
- 実行環境の状態にもとづく総称手続き（適応可能手続き）．
- 自己反映計算を用いた適応可能手続きの制御機構．

DAS モデルにもとづくソフトウェアはメタレベルアーキテクチャを形成し，メタレベルは以下のオブジェクトから構成される．

- 環境オブジェクト :仮想的かつ局所的な実行環境を表現する．
- イベントオブジェクト :環境オブジェクトの状態変化（状態イベント）を監視し，ソフトウェア中のコールバックメソッドを呼び出す．
- 戦略オブジェクト :適応の戦略を実現する．
- ディスパッチオブジェクト :戦略オブジェクトを用いて適応可能手続きを制御する．

DAS モデルでは，これらのオブジェクトと適応可能手続きにより，ソフトウェアの適応動作を実現する．環境オブジェクトは，適応可能手続きにとっての仮想的かつ局所的な実行環境であり，実際の実行環境を反映する．適応可能手続きは，複数の適応可能メソッドから構成される．適応可能メソッドは，環境オブジェクトの状態に対する条件（ディスパッチ条件）とその状態に対する実行コードから構成される．ディスパッチオブジェクトは，戦

略オブジェクトを用いて，適応可能メソッドのディスパッチを制御する．戦略オブジェクトは，適応の戦略を実装する適応可能メソッドからなる適応可能手続きを持つ．適応の戦略は，環境オブジェクトの状態に応じた適応可能メソッドをディスパッチするための方針である．適応の戦略を使って，戦略オブジェクトは，適応可能メソッドのディスパッチを実現する．適応の戦略は適応可能手続きとして実現されるため，環境オブジェクトの状態に応じて動的に変化する．イベントオブジェクトは環境オブジェクトの状態を監視する．イベントオブジェクトが環境オブジェクトの状態変化（状態イベント）を検出すると，ソフトウェア中に定義された状態イベントに対処するコールバックメソッドを呼び出す．コールバックメソッドは最新の環境オブジェクトをイベントオブジェクトから受けとる．コールバックメソッド中で適応可能手続きを呼び出すことにより，状態イベントにもとづく適応動作を効率的かつ簡潔に記述することが可能になる．

このような DAS モデルは，特定のパラダイムや記述言語から独立した汎用的なソフトウェアモデルであり，その適用範囲は広いが，特に移動コードシステムへの適用が有効である．

第 5 章

DAS クラスライブラリ

本研究では，DAS モデルの基本機構を Java 言語のクラスライブラリ (以下，**DAS クラスライブラリ**) として設計・実装した．DAS クラスライブラリは，DAS モデルのメタレベルを構成するオブジェクト (以下，メタオブジェクト) を提供し，適応可能手続きの実現を支援する．これらのメタオブジェクトは，その記述方法を定めた簡易なプロトコルにより柔軟に拡張することができる．また DAS クラスライブラリは，メタオブジェクトの利用および適応可能手続きの記述に際して若干のコーディング規則を持つ．本章では，DAS クラスライブラリの構成と機能，稼働環境について述べ，DAS クラスライブラリを用いた動的適応可能なソフトウェアの作成方法について実例を用いて示す．

5.1 DAS クラスライブラリの構成と機能

DAS クラスライブラリは，以下のカテゴリのクラス群から構成される．

- メタオブジェクトに関するクラス．
- 適応可能手続きに関するクラス．
- ユーティリティのクラス．

なお，Java はオブジェクト指向言語であり，手続きはオブジェクトに付随するメソッドとして実現されるため，手続きという語の使用はあまり適切ではない．そのため，これ以降，手続きという語のかわりにメソッドという語を用いることとし，適応可能手続きも適応可能メソッドと呼ぶ．これまでは，適応可能手続きの中の個々の実装を指して適応可能メソッドと呼んできたが，適応可能手続きとはもともと適応可能メソッドの集合であるか

ら，これを適応可能メソッドと呼ぶことはそれほど不適切ではない．しかし，両者 (適応可能手続きと適応可能メソッド) の厳密な区別が必要となる場合には，その都度明示することにする．

以下，上記のカテゴリ別に各クラスの機能について概説する．

5.1.1 メタオブジェクトに関するクラス

DAS モデルのメタオブジェクトは，環境オブジェクト，イベントオブジェクト，戦略オブジェクトおよびディスパッチオブジェクトである．これらのメタオブジェクトに関するクラスは以下である (表 5.1)．

表 5.1: メタオブジェクトに関するクラス

メタオブジェクト	クラス
環境オブジェクト	Environment
	StandardEnv
イベントオブジェクト	StandardEventCtrl
	StandardCallBack
	StateEvent
戦略オブジェクト	StandardStrategy
	SpecialStrategy
ディスパッチオブジェクト	BasicCtrl
	StandardCtrl
	SpecialCtrl

以下，各メタオブジェクトごとに関連するクラスについて簡単にその機能を述べる．

- 環境オブジェクトに関するクラス
  環境オブジェクトは，適応可能メソッドの抽象度に応じた仮想的かつ局所的な実行環境を表現するメタオブジェクトである．このような環境オブジェクトは，適応可能メソッドを呼び出す際に，引数として利用される．

  Environment
  最も低レベルな実行環境を表現するクラス．このクラスのインスタンスは最も抽象度の低い実行環境を表現し，その状態情報を保持する変数とその値を操作するメソッド

を持つ．このクラスから取得できる実行環境の状態情報は表 5.2のとおりである．これらの情報のうち⋆印のついたものは，ネイティブメソッドにより取得されるものである．それらのネイティブメソッドは C 言語により記述されており，DLL (ダイナミックリンクライブラリ) として提供されている (名称は，DAS.dll)．また，このクラスはすべての環境オブジェクトのスーパークラスである．

表 5.2: 実行環境の状態情報

状態情報	変数/メソッド	値
Java インタプリタのバージョン	version	文字列
ベンダ固有の識別子	vendor	〃
ベンダの URL	vendorUrl	〃
Java のディレクトリ	homeDir	〃
Java API のバージョン	apiVersion	〃
CLASSPATH の値	classPath	〃
オペレーティングシステム名	osName	〃
ホストの CPU	cpuName	〃
OS のバージョン	osVersion	〃
ファイル ・セパレータ	fileSeparator	〃
パス ・セパレータ	pathSeparator	〃
改行コード	lineSeparator	〃
現在のユーザ名	userName	〃
現在のユーザのホームディレクトリ	userDir	〃
現在のワーキングディレクトリ	userWorkingDir	〃
現在のホスト名	localHostName	〃
現在のホストの IP アドレス	localHostAddr	〃
⋆電源の種別	getPowerSource()	数値
⋆バッテリ残量	getBatteryRemainder()	〃
⋆ディスプレイサイズ (幅)	getDisplayWidth()	〃
⋆ディスプレイサイズ (高さ)	getDisplayHeight()	〃
⋆メモリ容量	getFreeMemory()	〃
⋆ディスク容量	getDiskFreeSpace()	〃

StandardEnv

標準的な実行環境 (環境オブジェクト) を実現するクラスであり．Environment クラスのサブクラスである．このクラスは，適応の戦略名 (文字列) を保持する変数とその値を操作するメソッドを提供し，標準的な環境オブジェクトのテンプレートとして利用される．ユーザは，このクラスのサブクラスとして各適応可能メソッドの抽象度に適した実行環境 (環境オブジェクト) を設定することができる．

- イベントオブジェクトに関するクラス
  イベントオブジェクトは環境オブジェクトの状態変化を監視し，その変化を検出すると，それに対処するためのコールバックメソッドを呼び出すメタオブジェクトである．

StandardEventCtrl

標準的なイベントオブジェクトを定義するクラス．このクラスのインスタンス (つまり，イベントオブジェクト) は，環境オブジェクト (StandardEnv クラスのインスタンス) を受け取り，その状態変化 (状態イベント) の発生を監視する．そして状態イベントの発生を検出すると，そのイベントに対応するコールバックメソッドを起動する．このクラスのイベントオブジェクトが監視する標準の状態イベントは表 5.3のとおりである．なお標準の状態イベントとして 10 個のユーザ定義イベントを自由に設定することができる．

表 5.3: イベントオブジェクトの提供する標準の状態イベント

状態イベント	内容
電源イベント	電源種別の変化を通知する．
バッテリイベント	バッテリ残量の変化を通知する．
メモリイベント	メモリ容量の変化を通知する．
ディスクイベント	ハードディスク容量の変化を通知する．
タイマーイベント	一定時間の経過を通知する．
ユーザ定義イベント 0	ユーザ定義のイベント
...	...
ユーザ定義イベント 9	ユーザ定義のイベント

StandardCallBack

標準のイベントオブジェクトが提供する状態イベントに対するコールバックメソッドを定義するクラス．このクラスでは，各コールバックメソッドのシグニチャのみが定

義されており，具体的な処理は定義されていない．ユーザはこのクラスを継承し，必要なイベントに対処するコールバックメソッドを上書きすることで，イベントに応じた処理を実現できる．コールバックメソッドの返り値型はvoid であり，引数は状態イベントオブジェクト（StateEvent）と環境オブジェクト（StandardEnv）である．標準の状態イベントとそれに対応するコールバックメソッドは表 5.4 のとおりである．

表 5.4: 標準の状態イベントとコールバックメソッド

状態イベント	コールバックメソッド
電源イベント	powerSourceChanged()
バッテリイベント	batteryRemainderChanged()
メモリイベント	memoryFreeSizeChanged()
ディスクイベント	diskFreeSizeChanged()
タイマーイベント	timerInterrupt()
ユーザ定義イベント 0	userState0Changed()
...	...
ユーザ定義イベント 9	userState9Changed()

StateEvent

イベントオブジェクトからコールバックメソッドに渡される状態イベントオブジェクトを定義するクラス．このクラスのオブジェクトからは，状態イベントの種類や発生時刻などの情報を得ることができる．

- 戦略オブジェクトに関するクラス
  戦略オブジェクトは，実行環境の状態に応じた適応可能メソッドを選択するメタオブジェクトであり，個々の戦略は適応可能メソッドとして定義されている．

  StandardStrategy

  最も基本となる適応の戦略を適応可能メソッドとして持つ戦略オブジェクトのクラス．このクラスでは，4.3 節で定義した AND 戦略，OR 戦略そして Default 戦略（基本的には AND 戦略であるが，適切な適応可能メソッドがない場合は，最も新しく定義された適応可能メソッドを選択する戦略）を定義している．このクラスを利用することで，これらの戦略を利用した適応可能メソッドのディスパッチを実現することができる．

SpecialStrategy

StandardStrategy クラスのサブクラスであり，標準的な戦略オブジェクトを定義するクラスである．スーパークラスが提供する適応の戦略の他に，正の優先度数戦略，負の優先度数戦略 (4.3 節参照) そしてユーザ指向戦略 (適応可能メソッドの選択時にダイアログを開き，各適応可能メソッドのドキュメント (5.1.2 節，5.3.2 節参照) を表示して，ユーザに選択させる戦略) を定義している．このクラスを利用することで，これらの戦略を利用することができる．また，このクラスからスーパークラスに定義されている基本的な戦略を利用することもできる．

MethodList

上記で述べたユーザ指向戦略を実現するための補助クラスである．実行時にダイアログを開いて，適応可能メソッドのドキュメントを表示する．

TypeCheck

適応の戦略が適応可能メソッドのディスパッチを行なう際，適応可能メソッドの型が妥当であるかどうかをチェックするための補助クラスである．

- ディスパッチオブジェクトに関するクラス
  ディスパッチオブジェクトは，戦略オブジェクトを用いて適応可能メソッドを制御するメタオブジェクトである．

  BasicCtrl

  適応可能メソッドのディスパッチを実現する上で，最も基本となる処理を提供するクラス．このクラスのインスタンスは，適応可能メソッドを登録するデータ構造を作成し，戦略オブジェクト (StandardStrategy クラスのインスタンス) を生成して，適応可能メソッドの制御を行なう．またデフォルトのメタレベルディスパッチ (4.5 節参照) で使用される適応の戦略は，このクラスのコンストラクタに戦略名 (文字列) を指定することで変更することができる．

  StandardCtrl

  標準的なディスパッチオブジェクトを定義するクラスである．このクラスでは，メタレベルディスパッチおよびベースレベルディスパッチ (4.5 節参照) を実現するメソッドが定義されている．このクラスのインスタンスは，BasicCtrl クラスのインスタンスを生成して，適応可能メソッドのディスパッチを行なう．このクラスのコンストラクタに戦略オブジェクトのクラス名 (文字列) を指定することで，使用する戦略オブジェクトを変更することができる．

SpecialCtrl

StandardCtrl クラスのサブクラスであり，SpecialStrategy クラスの戦略オブジェクトを使用するディスパッチオブジェクトを定義するクラスである．

5.1.2 適応可能メソッドに関するクラス

適応可能メソッドは，表 5.5の文字列とメソッドから構成される 3 つ組のデータ構造として表現される．

表 5.5: 適応可能メソッドのデータ構造

ドキュメント文字列	適応可能メソッドのドキュメント
条件メソッド	ディスパッチ条件の記述からなるメソッド
コードメソッド	適応動作の記述からなるメソッド

AdaptableMethod

適応可能メソッドを実現するデータ構造を表現するためのクラス．このオブジェクトの配列が，ディスパッチオブジェクト (BasicCtrl クラスのインスタンス) により動的に生成され，適応可能メソッドが登録される．

EvalArgs

さまざまな適応の戦略を用いることができるように適応可能メソッド (の条件メソッド) を定義するためのクラス．さまざまな戦略に用いることができる汎用的な条件メソッドの返り値型として利用する．このクラスを用いることで，柔軟に戦略の変更を可能とする適応可能メソッドを記述することができる．

5.1.3 ユーティリティのクラス

DAS クラスライブラリは記述面での利便性を考慮して，以下のユーティリティのクラスを提供している．

Invocation

オブジェクトの動的な生成を行なうためのユーティリティクラス．

Instantiation

メソッドの動的な呼び出しを行なうためのユーティリティクラス．

5.2 稼働環境

DAS クラスライブラリの稼働環境としては，以下を前提としている．

- WindowsNT/95/98
- Java VM および JDK1.1.5 もしくは 1.1.6

DAS クラスライブラリは，WindowsNT/95/98 からシステム情報を取得するネイティブメソッド (表 5.2) を一部含んでいるが，それを除けば，Java の動作するすべての計算機上で動作する．なお，DAS モデルの基本機構を Java 言語のクラスライブラリとして実装した理由は以下による．

- Java は多くの計算機上に移植され動作しているが，仮想機械形式であるため，各計算機に装備された特殊なデバイスなど，各計算機環境固有の特性を活かしたアプリケーションの作成が難しい．
- Java には Applet など，移動コードを前提としたアプリケーションが多く，それらは移動先の実行環境に適応することが望まれる．

つまり，Java およびそのアプリケーションは，本研究の背景で述べた問題を内包し，本研究の適用領域として適切であると考えられる．また，DAS モデルを Java のクラスライブラリとして実現することで，既存の Java Applet などに動的適応可能性を導入することが可能となる．

DAS クラスライブラリは適応可能メソッドの実現に JDK1.1.5 のリフレクション API (`java.lang.reflection`) を使用している．このような DAS クラスライブラリには，Java および Java リフレクション API にもとづく以下の制約がある．

- 適応可能メソッドは，`public` でなければならない (Java による制約)．
- 適応可能メソッドの戻り値型は，`void` もしくは`Object` 型でなければならない．また引数の型も`Object` 型でなければならない (Java リフレクション API による制約)．

5.3 DAS クラスライブラリを用いたプログラミング

5.3.1 記述の概要

DAS クラスライブラリを用いて動的適応可能なソフトウェアを実装する際，その主な記述は以下である．

- 適応可能メソッドによる適応動作に関する記述．
- 状態イベントを利用した適応動作に関する記述．

これらの記述は，それぞれメタレベル記述とベースレベル記述から構成される．それらの記述の概略を以下に示す．まず，適応可能メソッドによる適応動作は，以下の記述から実現される．

- メタレベル記述 ：
  - 環境オブジェクトの定義 (StandardEnv のサブクラス)．
- ベースレベル記述 ：
  - 使用するディスパッチオブジェクト (メタオブジェクト) の指定．
  - 適応可能メソッドの記述 (条件部の記述に，上記で定義した環境オブジェクトを使用する)．
  - 適応可能メソッドが定義されているクラスのインスタンス生成．
  - 適応可能メソッド呼び出し．

そして，状態イベントを利用した適応動作は，以下の記述から実現される．

- メタレベル記述 ：
  - 環境オブジェクトの定義 (StandardEnv のサブクラス)．
- ベースレベル記述 ：
  - コールバックメソッドのシグニチャを定義するクラス (StandardCallBack) を継承．
  - 取得するイベントに対応するコールバックメソッドの上書き (引数は，イベント状態オブジェクトと環境オブジェクト)．
  - 使用するイベントオブジェクト (メタオブジェクト) の指定．
  - イベントオブジェクトの起動 (カレントオブジェクトと上記で定義した環境オブジェクトを使用する)．
  - イベントオブジェクトの停止．

5.3.2 適応可能メソッドによる適応動作の記述

前述したように適応可能メソッドは，ドキュメント文字列，条件メソッド，コードメソッドの 3 つ組によるデータ構造として表現される．それぞれの要素は Java の文字列とメソッドであるが，通常の文字列やメソッドと区別するために，これらの要素の記述には以下のコーディング規則がある．

ドキュメント文字列	→	`$DOC_`メソッド名`_postfix` 番号
条件メソッド	→	`$COND_`メソッド名`_postfix` 番号
コードメソッド	→	`$CODE_`メソッド名`_postfix` 番号

DAS クラスライブラリにおいて上記の`$DOC_`，`$COND_`，`$CODE_`といったキーワードは予約語である．以下，例として適応可能メソッドによる適応動作の記述を示す．ここでは，ディスプレイサイズに応じて，コンソールに LARGE/MIDDLE/SMALL といった文字列のいづれかを表示する簡単な適応可能メソッドを記述する．

まず，環境オブジェクトの定義を行なう．環境オブジェクトは適応可能メソッドに対する仮想的かつ局所的な実行環境であり，適応可能メソッドの処理に対して適切な抽象度を持つように定義する必要がある．環境オブジェクトの抽象度とは，適応可能メソッドのディスパッチに必要となる情報の抽象度であり，具体的には，環境オブジェクトの情報を保持する変数やその情報を取得するメソッドの抽象度である．

環境オブジェクトを定義するには，標準で提供される`StandardEnv` クラスを継承する．そして，`StandardEnv` 中で定義されているメソッドを組み合わせて，新しい抽象度の高いメソッドを定義する．

表 5.6: 抽象度の高いディスプレイサイズ

ディスプレイサイズ (幅 ×高さ)	戻り値
(600×480) 以下のディスプレイ	SMALL
(800×600) 以下のディスプレイ	MIDDLE
(800×600) 以上のディスプレイ	LARGE

ここでは，`StandardEnv` クラスに定義されているディスプレイの幅のドット数を取得するメソッド`getDisplayWidth()` とディスプレイの高さのドット数を取得するメソッド`getDisplayHeight()` を利用して，表 5.6の仕様のメソッド`display` を定義する [1] ．

[1] 実際には，これと同様な抽象度を持つメソッドが`StandardEnv` クラスには定義されている．

```
// 環境オブジェクトの定義
public class SpecialEnv extends  StandardEnv  {
  // ディスプレイのサイズを取得するメソッドの定義
  public String display () { //     int width, height;
    width = getDisplayWidthSize();
    height = getDisplayHeightSize();
    if ((width <= 600) && (height <= 480)) { return("SMALL"); }
    if ((width <= 800) && (height <= 600)) { return("MIDDLE"); }
    else {
      return("LARGE");
    }
  }
  ...
}
```

すべての場合において，新たな環境オブジェクトの定義が必要となるわけではない．標準で提供されるStandardEnv をそのまま用いても何ら問題がない場合もある．しかし，このような抽象度の高い環境オブジェクトを定義することで，適応可能メソッドの条件部の記述 (条件メソッド) が，分かりやすく簡潔になる．

次に，先に示したコーディング規則にもとづいて適応可能メソッドを記述する．

```
// 適応可能メソッド (LARGE サイズディスプレイ用) の記述
// ドキュメント文字列
public String $DOC_sample_1 = "The method for LARGE display";
// 条件メソッド
public EvalArgs $COND_sample_1(SpecialEnv obj) {
  EvalArgs eArgs = new EvalArgs(obj);
  eArgs.element  = new String[] {"display"};
  eArgs.bools    = new boolean[] {obj.display() == "LARGE" };
  return(eArgs);
}
// コードメソッド
public void $CODE_sample_1(SpecialEnv obj) {
  System.out.println("LARGE");
}

// 適応可能メソッド (MIDDLE サイズディスプレイ用) の記述
// ドキュメント文字列
public String $DOC_sample_2 = "The method for MIDDLE display";
// 条件メソッド
public EvalArgs $COND_sample_2(SpecialEnv obj) {
  EvalArgs eArgs = new EvalArgs(obj);
  eArgs.element  = new String[] {"display"};
  eArgs.bools    = new boolean[] {obj.display() == "MIDDLE" };
  return(eArgs);
}
```

```
// コードメソッド
public void $CODE_sample_2(SpecialEnv obj) {
  System.out.println("MIDDLE");
}

// 適応可能メソッド (SMALL サイズディスプレイ用) の記述
// ドキュメント文字列
public String $DOC_sample_3 = "The method for SMALL display";
// 条件メソッド
public EvalArgs $COND_sample_3(SpecialEnv obj) {
  EvalArgs eArgs = new EvalArgs(obj);
  eArgs.element  = new String[] {"display"};
  eArgs.bools    = new boolean[] {obj.display() == "SMALL" };
  return(eArgs);
}
// コードメソッド
public void $CODE_sample_3(SpecialEnv obj) {
  System.out.println("SMALL");
}
```

上記のコードは，sample という名前の 3 つの適応可能メソッドを表現している．前述したように，各適応可能メソッドは，ドキュメント文字列，条件メソッド，コードメソッドの 3 つ組で表現される．ドキュメント文字列は，ユーザ指向戦略を利用した場合に，ダイアログに表示される文字列である．条件メソッドは，適応可能メソッドのディスパッチ条件を記述するメソッドであり，環境オブジェクトに対する条件を記述するメソッドである．この場合，先に定義した環境オブジェクト (SpecialEnv クラスのインスタンス) に対する条件を記述する．ここでは，ディスプレイのサイズ (LARGE/MIDDLE/SMALL) が適応可能メソッドのディスパッチ条件となる．上記の例では，条件メソッドの型がEvalArgs となっているが，特にこの型である必要はない．しかし，EvalArgs は複数の戦略 (AND 戦略，OR 戦略，正の優先度数戦略，負の優先度数戦略) を使用可能にする補助クラスであり，このクラスを使って上記のように記述すると，あとで適応の戦略を柔軟に変更できるという利点がある．EvalArgs は環境オブジェクトと，ディスパッチに関係する環境オブジェクトの要素名の配列，各要素に対する条件の真偽値の配列を持ち，各戦略にもとづいた適応可能メソッドのディスパッチを支援するメソッドを持つ．しかし，これらの内部動作についてユーザが意識する必要はない．そして，条件メソッドとコードメソッドの第一引数は同じクラスの環境オブジェクトでなければならない．またドキュメント文字列，条件メソッド，コードメソッドはそれぞれコーディング規則にもとづいて，$DOC_，$COND_，$CODE_といったキーワードをその名前の接頭辞として付け，接尾辞としてポストフィックス番号 (数字) を付加する．つまり，これらの名前はユニークである必要がある．

次に，上記で定義した適応可能メソッドの呼び出し手順とその記述を示す．

1. 環境オブジェクトを生成する．

   ```
   SpecialEnv virtualEnv = new SpecialEnv();
   ```

   まず，定義した環境オブジェクトのインスタンスを生成する．環境オブジェクトのコンストラクタに適応の戦略名を指定すると，生成された環境オブジェクトには，指定された戦略が設定される．戦略名の指定を省略した場合は，Default 戦略が設定される．

2. 適応可能メソッドの定義されているクラスを生成する．

   ```
   SampleClass sc = new SampleClass();
   ```

   ここでは，上記の適応可能メソッド定義を含むクラスを`SampleClass` と仮定する．

3. ディスパッチオブジェクトを生成する．

   ```
   StandardCtrl $DISPATCH_StandardCtrl = new StandardCtrl();
   ```

   ここでは，標準のディスパッチオブジェクトを使用して，適応可能メソッドのディスパッチを実現することにする．

4. 適応可能メソッドの呼び出し．

   ```
   Object[] argObjects = new Object[1];
   argObjects[0] = virtualEnv;
   $DISPATCH_StandardCtrl.adaptableCtrl(sc, "sample", argObjects);
   ```

   まず，`Object` 型の配列を作り，その中に環境オブジェクトと適応可能メソッド (のコードメソッド) への引数をパックする．そして上記 3 で生成したディスパッチオブジェクトのメソッド`adaptableCtrl` を用いて，適応可能メソッドの呼び出しを行なう．`adaptableCtrl` の引数は，適応可能メソッドを含むクラスのオブジェクト，適応可能メソッドの名前，適応可能メソッドへの引数である．

以上の手順を通して，先に定義したディスプレイサイズに応じてLARGE/MIDDLE/SMALL のいづれかの文字列をコンソールに表示する適応可能メソッドを実現することができる．

5.3.3 状態イベントを利用した適応動作の記述

次に，状態イベントを利用した適応動作の記述を示す．記述の概要で述べたように状態イベントの利用においてもメタレベルの記述として，まず環境オブジェクトの定義を行な

う．これはアプリケーション中で必要となる抽象度の高いイベントを設定する場合にのみ必要となる．簡単化のため，ここでは，標準のイベントオブジェクトに定義されている状態イベントを用いることとし，その記述の概要を示すことにする．ここでは，電源種別の変化を検出する電源種別イベントの使用例を示す．

```
// 電源種別イベントの利用．
public class SampleClass extends StandardCallBack {
  public static SampleClass currentObj = null;
  public SampleClass() {
    currentObj = this;
  }
  ...
// 環境オブジェクトの生成
  StandardEnv virtualEnv = new StandardEnv();
  ...
// イベントオブジェクトの生成
  StandardEventCtrl $EVENT_StandardEventCtrl = new StandardEventCtrl();
  ...
// イベントオブジェクトの起動
  $EVENT_StandardEventCtrl.enable(currentObj, virtualEnv);
  ...
// イベントオブジェクトの停止
  $EVENT_StandardEventCtrl.disable();
  ...
// 電源種別イベントに対するコールバックメソッド
  public void powerSourceChanged(StateEvent evt, StandardEnv env) {
    ... // 引数として渡される環境オブジェクトを
        // 利用して，適応可能メソッドを呼び出す．
  }
  ...
}
```

上記のコードは，電源種別イベントを用いた適応動作の主要な記述である．まず，コールバックメソッドのシグニチャを定義しているStandardCallBack クラスを継承し，電源種別イベントに対応するコールバックメソッドpowerSourceChanged() (表 5.4) を実装する．コールバックメソッドには，状態イベントオブジェクトと環境オブジェクトが渡される．コールバックメソッド中で，渡された環境オブジェクトを用いて適応可能メソッドを呼び出すことで，状態イベントにもとづく適応動作を実現することができる．上記のコードでは，標準のイベントオブジェクト (StandardEventCtrl クラスのインスタンス) を利用している．イベントオブジェクトの起動には，カレントオブジェクトと環境オブジェクトが必要である．これらのオブジェクトを引数として，イベントオブジェクトの起動を上記のように行なう．イベントオブジェクトは，別スレッドで動作し，渡された環境オブジェ

クトの状態変化を監視する．この場合，電源の種別が変化した時，対応するコールバックメソッド`powerSourceChanged()` を呼び出す．

5.4 動的適応可能な回覧板システム

本研究では，例題として DAS クラスライブラリを用いて動的適応可能な回覧板システム (図 5.1) を作成した．このシステムは，移動コードにもとづくシステムであり，Java ベースの移動エージェントシステム AgentSpace [45] をプラットフォームとして利用している．しかし，DAS クラスライブラリは特定の移動コードシステムに依存しないため，Java ベースの移動コードシステムであれば，どれも利用可能である．本節では，回覧板システムの動作について概略を述べ，その記述を示す．

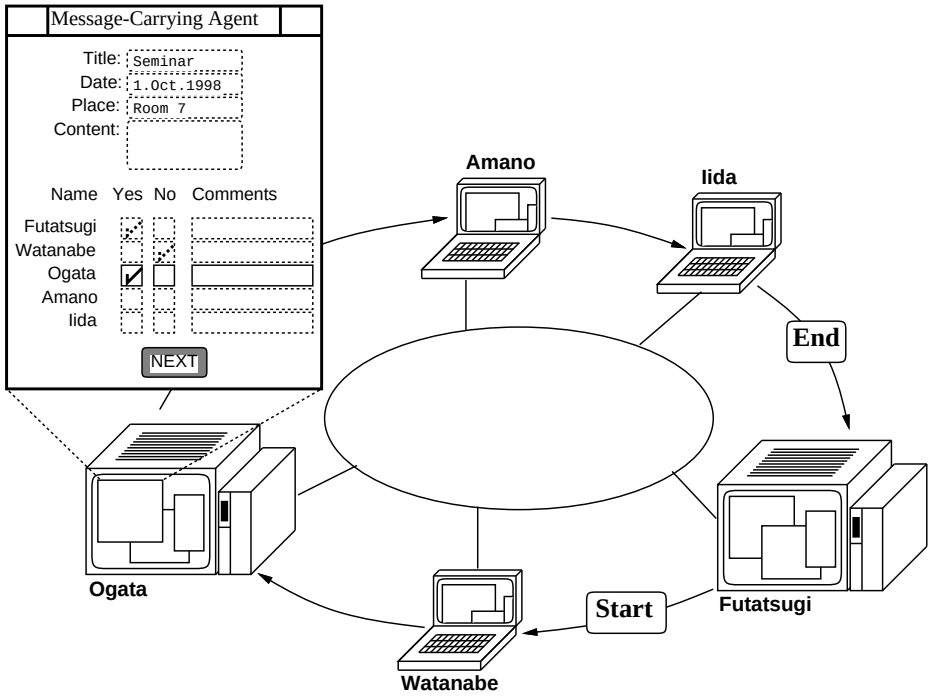

図 5.1: 動的適応可能な回覧板システム

5.4.1 例題の仕様

この回覧板システムは，現実の回覧板を計算機を用いてシミュレートしたものである．つまり，回覧板を表示する移動エージェント (以下，回覧板エージェント) が，現実の回覧

板のように我々の研究室内の計算機を順に移動し，ミーティングやセミナーの日時，場所などの情報を表示して，各ユーザの出欠情報を集める．具体的には，回覧板エージェントは，移動後，回覧内容を表示するウインドウを開き，ユーザによる出欠欄のチェックを待つ．ユーザは出欠欄にチェックした後，ウインドウ上にあるボタンを押すことにより，次のユーザに回覧板エージェントを廻す．回覧板エージェントは，研究室内の計算機のリストを持っており，一定の順序でそれらを巡回する．

このような移動コードにもとづく回覧板システムを例題としたのは，以下の理由による．

- 移動コードを用いることで，現実の回覧板の自然なモデル化が可能である．
- 適応動作により，現実の回覧板の問題点を回避することができる．
- 適応動作により，移動コードシステムにもとづく問題点を回避することができる．
- 移動コードにもとづくシステムは，実行中に他の計算機に移動するため，実行環境の状態変化が急激かつ抜本的であり，本研究の適用領域として適切である．

以下は，このシステムにおける回覧板エージェントの適応動作の仕様である．

- ディスプレイのサイズに応じた GUI の動的な変更．
  - ウインドウの表示位置を変更する．
  - ウインドウや文字の大きさを変更する．
  - ウインドウの大きさに応じてタイトルを変更する．
  - ウィジェット (ウインドウの構成部品) の大きさや配置を変更する．
- ユーザごとに入力欄の enable/disable を行なう．
- ユーザによる入力が 30 分を越えてもない場合，次のホストに移動する．
- 動作中のホストのバッテリ切れにより次のホストに移動する．

まず，計算機のディスプレイサイズに応じて，GUI の変更を行なう．これは，回覧板の look & feel および操作性を高めるための適応動作である．そして，ユーザの入力欄は，ユーザごとに enable/disable される必要がある．他者の入力を勝手に変更することができなくするためである．これは，移動コードシステムにおけるユーザデータのセキュリティに関する適応動作ととらえられる．また，ユーザによる入力が行なわれなかった場合，回覧板

エージェントは，入力を待って，同一計算機上に留まることになる．これは現実の回覧板でも起こる問題である．すなわち，ある人がしばらく回覧板を廻さずにいた場合がこれに相当する．このような問題は，現実の回覧板では，回避するすべがない．これに対して，回覧板エージェントでは，タイマーイベントを利用することで，この問題をうまく回避している．さらに，この回覧板エージェントは，バッテリ駆動によるノート型パソコン上での動作中に，バッテリ切れを検出し，次のホストに移動することができる．もし，バッテリが切れ，ホストがダウンした場合，回覧板エージェントは，そこで消滅してしまう．つまりそれまでに集めた情報はすべて無駄になり，再度，回覧板エージェントを廻すことになる．しかし，この回覧板エージェントは，バッテリイベントを利用してこの問題を回避している．これは移動コードシステムにおけるフォールトトレラント処理としての応用が見込める．

そして，より重要なことは，移動コードのプラットフォーム AgentSpace では，タイマーイベントやバッテリイベントなどを提供していない点である．つまり，回覧板エージェント自身がそれらを持ち，みずから適応動作を実現している点である．このため，AgentSpace 以外のプラットフォームでも同様の適応動作を実現することができる．

5.4.2 GUI に関する適応動作

前述したように回覧板エージェントは，計算機のディスプレイサイズに応じて，その GUI を変更する．本節では，この処理の記述 (一部) を以下に示す．ここでは，環境オブジェクトとして 5.3.2 節で定義したものを使用するものとする．

```
// 動的適応可能な回覧板エージェントの定義 (一部)
public class WorkflowAgent extends Agent { // Agent は AgentSpace のクラス

// 適応可能メソッド (SMALL サイズディスプレイ用)
  public String $DOC_createWindow_1 =
                "This method opens the small window for small display.";
  public EvalArgs $COND_createWindow_1(SpecialEnv obj) {
    EvalArgs eArgs = new EvalArgs(obj);
    eArgs.element  = new String[] {"display"};
    eArgs.bools    = new boolean[] {obj.display() == "SMALL"};
    return(eArgs);
  }
  public void $CODE_createWindow_1(SpecialEnv obj) {
    ...
  }
```

```
// 適応可能メソッド (MIDDLE サイズディスプレイ用)
  public String $DOC_createWindow_2 =
                "This method opens the small window for middle display.";
  public EvalArgs $COND_createWindow_2(SpecialEnv obj) {
    EvalArgs eArgs = new EvalArgs(obj);
    eArgs.element  = new String[] {"display"};
    eArgs.bools    = new boolean[] {obj.display() == "MIDDLE"};
    return(eArgs);
  }
  public void $CODE_createWindow_2(SpecialEnv obj) {
    ...
  }

// 適応可能メソッド (LARGE サイズディスプレイ用)
  public String $DOC_createWindow_3 =
                "This method opens the small window for large display.";
  public EvalArgs $COND_createWindow_3(SpecialEnv obj) {
    EvalArgs eArgs = new EvalArgs(obj);
    eArgs.element  = new String[] {"display"};
    eArgs.bools    = new boolean[] {obj.display() == "LARGE"};
    return(eArgs);
  }
  public void $CODE_createWindow_3(SpecialEnv obj) {
    ...
  }
  ...
}
```

上記のコード中で，Agent とは，AgentSpace が提供するエージェントの基本クラスであり，すべての移動エージェントはこのクラスを継承して作成される．また上記で定義された適応可能メソッドは，5.3.2 節で示したものとほぼ同じであるため，ここではその詳細は省略する．以下は，この適応可能メソッドcreateWindow の呼び出しに関する記述である．

```
// ディスパッチオブジェクトの生成．
StandardCtrl $DISPATCH_StandardCtrl = new StandardCtrl();
...
// 環境オブジェクトの生成 (Default 戦略の使用を想定)．
SpecialEnv virtualEnv = new SpecialEnv();
...
// 適応可能メソッドの定義されているクラスの生成．
WorkflowAgent wfa = new WorkflowAgent();
...
// 適応可能メソッドの呼び出し．
Object[] argObjects = new Object[1];
argObjects[0] = virtualEnv;
$DISPATCH_StandardCtrl.adaptableCtrl(wfa, "createWindow", argObjects);
```

ここでは，標準のディスパッチオブジェクトを用いて適応可能メソッドのディスパッチを行なっている．適応可能メソッドの呼び出し手順についての詳細は，5.3.2 節を参照されたい．

5.4.3 タイマーイベントの利用

回覧板エージェントは，タイマーイベントを利用して，ユーザからの入力が 30 分間を超えてもなかった場合，次のホストに移動する．以下のコードは，その記述の一部である．

```
// タイマーイベントの利用．
public class WorkflowAgent extends StandardCallBack {
  public static WorkflowAgent currentObj = null;
  public WorkflowAgent() {
    currentObj = this;
  }
  ...
// イベントオブジェクトの生成
  StandardEventCtrl $EVENT_StandardEventCtrl = new StandardEventCtrl();
  public void arrive() {    // Agent クラスのメソッド．
    StandardEnv virEnv = new StandardEnv(); // 環境オブジェクトの生成．
   // イベントオブジェクトの起動．
    $EVENT_StandardEventCtrl.enable(currentObj, virEnv);
    ...
  }
  public void dispacth() {  // Agent クラスのメソッド．
    ...
   // イベントオブジェクトの停止．
    $EVENT_StandardEventCtrl.disable();
  }
// タイマーイベントに対するコールバックメソッド
  public void timerInterrupt(StateEvent evt, StandardEnv env) {
    ... // 引数として渡される環境オブジェクトを利用して，
        // 隣のホストを決定する適応可能メソッドを呼び出す．
  }
  ...
}
```

状態イベントを利用した適応動作の記述の詳細については 5.3.3 節を参照されたい[2]．ここでは，タイマーイベントに対処するコールバックメソッド`timerInterrupt()` を実装している．具体的な記述については割愛するが，この中で渡された環境オブジェクトから，回覧板エージェントは，現在のホスト名を取得し，現在のホストに応じて，次に移動すべき

[2] ここでは，`StandardCallBack` クラスが`Agent` クラスを継承しているものとする．

ホストを決定する．つまり，適応可能メソッドを使って次に移動すべきホストを決定している．

上記のコード中で，arrive() やdispatch() といったメソッドは，AgentSpace が提供するAgent クラス中に定義されたメソッドである．arrive() は，移動した直後に AgentSpace の実行時システムから呼ばれるメソッドであり，dispatch() は，移動する直前に呼ばれるメソッドである．つまり，ここでは，移動した直後に，イベントオブジェクトを起動して，タイマーイベントを監視し，移動する直前にイベントオブジェクトを停止する．なお，30 分間というタイマーの時間は，変更することができる．

5.5 メタオブジェクトのカスタマイズ

DAS クラスライブラリの提供するメタオブジェクトは，その記述方法を定めた簡易なプロトコル（以下，メタオブジェクトプロトコル）により容易にカスタマイズすることができる．本節では，まずメタオブジェクトプロトコルにより，どのようなカスタマイズが可能になるかを述べ，さらにカスタマイズの方法について簡単に述べる．

DAS クラスライブラリの提供するメタオブジェクトとは，環境オブジェクト，イベントオブジェクト，戦略オブジェクト，ディスパッチオブジェクトといった適応動作の実現を担うオブジェクトである．これらのメタオブジェクトに対して，メタオブジェクトプロトコルを用いて，以下のようなカスタマイズが可能である．

- 環境オブジェクトのカスタマイズ ：
  適応可能メソッドの抽象度に適した仮想的かつ局所的な実行環境とその状態を表現することができる．

- イベントオブジェクトのカスタマイズ ：
  各環境オブジェクトの抽象度に応じた状態イベント（状態変化の発生）およびこれに対処するコールバックメソッドの定義が可能となる．

- 戦略オブジェクトのカスタマイズ ：
  新しい適応の戦略を追加することが可能になる．

- ディスパッチオブジェクトのカスタマイズ ：
  使用する戦略オブジェクトの変更や，メタレベルディスパッチを実現する部分を変更することができる．これにより，適応可能メソッドが呼ばれる度に，毎回メタレベルディスパッチを行なわないようにする，などの処理を実現することができる．

このようなメタオブジェクトのカスタマイズを実現するメタオブジェクトプロトコルは非常に簡単なもので，具体的には，

メタオブジェクトを定義するクラスを継承し，そのメソッドの再定義・追加や適応可能メソッドの追加を行なう，

ことである．以下は，各メタオブジェクトに対するメタオブジェクトプロトコルを用いたカスタマイズ方法である．

- 環境オブジェクト：
  StandardEnv クラスを継承し，メソッドの再定義や追加を行ない，適応可能メソッドの抽象度に応じた実行環境を定義する．
- イベントオブジェクト：
  StandardEventCtrl クラスを継承し，環境オブジェクトの状態変化を監視するメソッドの追加・再定義を行ない，呼ばれるコールバックメソッドの追加・再定義を行なう．
- 戦略オブジェクト：
  StandardStrategy クラスを継承し，新しい戦略を実現する適応可能メソッド (適応可能メソッド名dispatch) を追加する．
- ディスパッチオブジェクト：
  StandardCtrl クラスを継承し，生成する戦略オブジェクトの指定部分やメタレベルディスパッチを実現するメソッドを再定義する．

ここで，これらのメタオブジェクトプロトコルにもとづいたメタオブジェクトの具体的なカスタマイズを示す．ここでは，例として以下のカスタマイズを示す..

- 戦略オブジェクトに新しい戦略 (適応可能メソッド) を追加する．
- ディスパッチオブジェクトで使用する戦略オブジェクトを変更する．
- イベントオブジェクトで監視するイベントの追加・変更を行なう．

戦略オブジェクトのカスタマイズは ,StandardStrategy クラスを継承し ,適応の戦略を実装する適応可能メソッドdispatch() の追加定義を行なう . 以下の記述は , その一部である．

```
// 新しい適応の戦略を戦略オブジェクトに追加する
public class SpecificStrategy extends StandardStrategy {
  ...
// 新しい適応の戦略を実装する適応可能メソッドの定義
// ドキュメント文字列
  public String $DOC_dispatch_7 = "New Strategy";
// 条件メソッド
  public EvalArgs $COND_dispatch_7(StandardEnv obj) {
    EvalArgs eArgs = new EvalArgs(obj);
    eArgs.element  = new String[] {"strategy"};
    eArgs.bools    = new boolean[] {obj.getStrategy() == "new" };
    return(eArgs);
  }
// コードメソッド
  public Method $CODE_dispatch_7(StandardEnv obj,
                                 String methodName, Object[] args) {
  ...
  }
...
}
```

上記のコードでは，戦略名がnew である，新しい戦略を定義しいてる．新しい戦略を定義する上で注意しなければならない点は，ドキュメント文字列，条件メソッド，コードメソッドのそれぞれの名前が既存のものと衝突しないようにすることである (表 5.7)．また，環境オブジェクト中に保持される戦略の名前を示す文字列もユニークでなければならない．また，条件メソッドとコードメソッドのシグニチャも上記のとおりである必要がある．

表 5.7: 適応の戦略を実装する適応可能メソッド

定義クラス	戦略	戦略名	ポストフィックス番号
StandardStrategy	Default 戦略	default	1
〃	AND 戦略	and	2
〃	OR 戦略	or	3
SpecialStrategy	正の優先度数戦略	positive	4
〃	負の優先度数戦略	negative	5
〃	ユーザ指向戦略	user-like	6

なお，getStrategy() メソッドは，環境オブジェクトのクラスStandardEnv 中に定義された戦略変数の値 (戦略名を表現する文字列) を取得するメソッドである．

次に，いま定義した新たな戦略オブジェクトを使用するディスパッチオブジェクトを定

義する．これは，標準のディスパッチオブジェクトのクラスStandardCtrl を継承し，そのコンストラクタ中に親クラス，すなわちStandardCtrl のコンストラクタに戦略オブジェクトのクラス名を表す文字列を渡すことで実現できる．以下のコードは，その例である．

```
// ディスパッチオブジェクトで使用する戦略オブジェクトを変更する
public class SpecificCtrl extends StandardCtrl {
  ...
  public SpecialCtrl() {
    super("SpecificStrategy");
  }
  ...
}
```

最後に，イベントオブジェクトで監視するイベントの追加・変更方法を示す．標準のイベントオブジェクト（StandardEventCtrl クラスのインスタンス）は，デフォルトでユーザ定義イベントを 10 個定義することができる．まず，ユーザ定義イベントの定義方法を以下に示す．ユーザ定義イベントの定義方法は，以下に要約できる．

- StandardEventCtrl クラスを継承する．
- 状態イベント（環境オブジェクトの状態変化）を取得するメソッド（イベント取得メソッド）を再定義する（戻り値型はString，引数はObject 型）．
- イベントに応じたコールバックメソッドを呼び出すメソッド（イベントハンドラ）を再定義する（戻り値はvoid，引数はObject 型）．
- StandardCallBack クラスを継承し，対応するコールバックメソッドを実装する．

StandardEvent クラスには，ユーザ定義イベントに対して以下のイベント取得メソッドとイベントハンドラのシグニチャが定義されている（表 5.8）．

表 5.8: ユーザ定義イベントのイベント取得メソッドとイベントハンドラ

イベント名	イベント取得メソッド	イベントハンドラ
ユーザ定義イベント 0	getUserState0()	userDefinedEvent0()
...	...	...
ユーザ定義イベント 9	getUserState9()	userDefinedEvent9()

また，StandardCallBack クラスには，ユーザ定義イベントに対応する以下のコールバックメソッドのシグニチャが定義されている (表 5.9)．

表 5.9: ユーザ定義イベントとコールバックメソッド

イベント名	コールバックメソッド
ユーザ定義イベント 0	`void userState0Changed(StateEvent s, StandardEnv e)`
...	...
ユーザ定義イベント 9	`void userState9Changed(StateEvent s, StandardEnv e)`

これらのメソッドを上書きすることで，ユーザ定義イベントを定義することができる．

次に，イベントオブジェクトで監視する標準の状態イベントの変更方法を示す．まず，イベントオブジェクトのクラスStandardEventCtrl を継承し，その run() メソッドを上書きする．新しいイベントを追加する場合には，それを監視するメソッドの定義を行なう．そして，それに対応するコールバックメソッドのシグニチャ定義をStandardCallBack を継承したクラス内で行なう．以下は，このカスタマイズを実現する記述である．

```
// イベントオブジェクトのカスタマイズ
public class SpecificEventCtrl extends StandardEventCtrl {
  StateEvent eventObj = new StateEvent();
  public void run() {
    ...
    newEvent(env); // 新しいイベントの監視
    ...
  }
  ...
// 新しいイベントのためのイベントハンドラ
  public void newEvent(Object env) {
    ...
    String State = ((StandardEnv)env).getSomethingState();
    if (State == "changed") {
      eventObj.setOccurTime(System.currentTimeMillis());
      eventObj.setEventKind("New");
      eventObj.setEventState("");
      SpecificCallBack scb = (SpecificCallBack)targetObj;
      // 新しいコールバックメソッドの呼び出し
      scb.newEventOccr(eventObj, (StandardEnv)env);
      ...
    }
  }
}
```

```
// 新しいコールバックメソッドのシグニチャ定義を行なうクラス
public class SpecificCallBack extends StandardCallBack {
  ...
  public void newEventOccur(StateEvent event, StandardEnv env);
  ...
}
```

上記のコードでは,状態イベントnewEventとそのコールバックメソッドnewEventOccur()を定義している.なお,ここでenvは環境オブジェクト,eventObjは状態イベントオブジェクト(StateEventクラスのオブジェクト),targetObjはコールバックメソッドnewEventOccur()を実際に実装しているオブジェクトであるとする.また,環境オブジェクト中に何らかの状態を取得するメソッドgetSomethingState()が定義されているものとし,その状態がchangedであれば,状態イベントオブジェクト(イベントの詳細情報を持つオブジェクト)を作成してコールバックメソッドnewEventOccur()を呼び出す.ここで,targetObjは,新たに定義したSpecificCallBackクラスのコールバックメソッドを実装しているとする.

5.6 まとめ

DASモデルの基本機構を実現するDASクラスライブラリ(Java言語のクラスライブラリ)の構成および機能について述べ,DASクラスライブラリによるプログラミングについて例題を用いて示した.DASクラスライブラリは,DASモデルのメタレベルを構成するオブジェクト(メタオブジェクト)を提供し,適応可能手続きの実現を支援する.メタオブジェクトの利用および適応可能手続きの記述には若干のコーディング規則がある.また,DASクラスライブラリは,メタオブジェクトを拡張する簡易なプロトコル(メタオブジェクトプロトコル)を提供し,ユーザによるメタオブジェクトの柔軟な拡張を可能とする.

本研究では,実際にDASクラスライブラリを用いて移動コードベースの動的適応可能な回覧板システムを実装した.このシステムにおける回覧板エージェントは,GUIの変更を行なう,ユーザによる入力欄のenable/disableを行なう,ユーザの不在や計算機のダウンに対して,次のホストへ移動する,といった適応動作を実現する.この回覧板エージェントの実装では,移動コードのプラットフォームとしてJavaベースの移動コードシステムAgentSpaceを利用したが,DASクラスライブラリ自体は,特定の移動コードシステムに依存するものではない.

また,本章では,メタオブジェクトプロトコルを用いたメタオブジェクトのカスタマイズ方法についても,記述例をあげて示した.メタオブジェクトプロトコルにより,適応動作の実現機構を柔軟かつ系統的にカスタマイズすることが可能になる.

第 6 章

関連研究

これまでにも動的適応可能なソフトウェアに関するさまざまな研究がなされている．本章では，こうした関連研究について概説し，本研究で提案している DAS モデルとの比較を行なう．

6.1 オペレーティングシステム

動的適応可能なオペレーティングシステム (OS) として，Apertos [60]，SPIN [4]，Exokernel [11] などがある．

Apertos は，自己反映オブジェクト指向にもとづく OS であり，アプリケーションの最適な実行を実現するために，その構成を動的に変更することができる．Apertos では，システム全体がオブジェクトの集合で表現される．すべてのオブジェクトは，メタオブジェクトのグループ (メタオブジェクト空間と呼ばれる) によって与えられる実行環境内で実行される．つまり，メタオブジェクトはオブジェクトに対して，その最適なインタフェースや実行の意味付けを与え，そのための機能を提供している．そして，オブジェクトはそのメタオブジェクト空間を必要に応じて別のメタオブジェクト空間に変更することができる．さらに，オブジェクトはその実行環境を提供するメタオブジェクトと明示的に分離され，両者は因果的結合を持つ．

SPIN は拡張可能な OS であり，OS が提供するサービスをカスタマイズするための機構と拡張可能なサービスの基本セットを提供している．それらを用いて，アプリケーションは OS のインタフェースやインプリメンテーションを安全に変更することができる．拡張によりアプリケーションは，実行性能や特別な機能の実現のために OS を特殊化することができる．OS の拡張は型安全な言語によって実装され，拡張された部分はカーネルに動的

にリンクされる．SPIN のアプローチは，カーネル空間で実行されている OS のコードを保護しつつシステムが提供するサービスの拡張を実現する．

Exokernel は，アプリケーションレベルでのリソース管理を実現するための機構を提供する OS である．従来の OS では，アプリケーションの実行性能，柔軟性，機能などが，OS のインプリメンテーション (たとえば，プロセス間通信や仮想メモリなど) の抽象度に応じて制限される．これに対して，Exokernel は物理的なリソースのアプリケーションレベルでの管理を可能とする．Exokernel のアーキテクチャでは，小さなカーネルがライブラリ OS に対してすべてのハードウェアリソースを低レベルのインタフェースを通じてエクスポートする．ライブラリ OS は，このインタフェースを用いてシステムオブジェクトやポリシを実現する．この実現において抽象度が高く，アプリケーションの実行性能や機能の要求にかなう特殊用途のインプリメンテーションを定義することができる．

6.2 移動コード/エージェントシステム

本節では，移動コード/エージェントにもとづく動的適応可能なソフトウェアシステムについて述べる．そのような移動コード/エージェントシステムとして以下のものがある．

Flage [29] は，機能要求の変更に対して自律的に適応できる分散システムを構築するための言語システムである．Flage では，アプリケーションはエージェントによって構成される．エージェントは，メタレベルアーキテクチャを形成する並行モーバイルオブジェクトである．そのメタレベルは，エージェントの挙動の自己制御とその機能要求の変化への適応を実現する．Flage におけるエージェントは，ネットワーク上に点在する場と呼ばれるソフトウェア部品の保管場所を自律的に移動することにより，新しい機能を獲得し，その挙動を変更することができる．また，場を介して複数のエージェントが密接に協調することにより，エージェントの機能変更処理を効率化し，さらに周辺機能との不整合を自動的に検出し解消することができる．

Plangent [39] は，Java ベースの移動エージェントシステムであり，エージェントを記述するスクリプト言語を提供している．Plangent のエージェントは，自らの行動プランを生成するプランニング機能を持つ．Plangent では，プランニングにおいて必須の知識ベースをネットワークのノード毎に持つことができる．また，エージェント自身も固有の知識ベースを持つことができる．プランニングは，これらの知識ベースに含まれるアクションにもとづいて行なわれる．アクションには，ある状況で，あるゴールを解決するための行動プランが定義されており，エージェントは，その時々の状況に応じて適切なアクションを選択し，行動プランを生成することができる．この行動プランが失敗した場合，エージェン

トは再プランニングを行ない，別の方法でゴールを達成することを試みる（再プランニング）．エージェントは移動することで，各ノードに存在する知識ベースを利用し，ノード毎に最適な処理を実現する．

また，Telescript [59] は，エージェントを記述するためのスクリプト言語を備えた移動エージェントシステムである．Telescript は，エージェントが移動先の計算機の計算能力や計算資源を過剰に消費しないための機構としてパーミットという概念を持つ．Telescript では，すべてのエージェントは自分が使用できる計算資源の量や生存時間の最大値を規定したパーミットオブジェクトを持つ．そして，エージェントの実行環境を表現するプレースは，プレース内のエージェントが持つパーミットにさらに制限をかけることができる．Telescript は，エージェントの適応動作を特に意識したシステムではないが，パーミットの概念はプレースという実行環境に対するエージェントの受動的な適応動作ととらえることができる．

その他の移動コードシステムとして，Aglets[30, 31] ，Agent Tcl [16] ，April [36] ，Scheme48 [25] ，CAML [2] ，GNU Guile [1] など，さまざまなシステムが提案されている．しかし，これらのシステムにおける移動エージェントは，その実行環境に対する適応についてほとんど考慮されていない．

6.3 言語・モデル・機構

本節では，動的適応可能なソフトウェアのための言語，モデル，機構などについて述べる．これらの観点からは，以下のような関連研究がある．

適応化コンポジション [19] は，開放型分散システム環境における動的適応可能なソフトウェアを構成するオブジェクト指向言語のための機構である．適応化コンポジションは，計算の進行により，オブジェクトが新しい属性/動作を獲得するオブジェクトの成長を基本概念としている．適応化コンポジションでは，オブジェクトがある環境（オブジェクトの実行環境を表現するオブジェクト）に入ると，その環境にあらかじめ用意されていた属性や動作を取得し，その環境に適したオブジェクトに変換される．このような適応化コンポジションにもとづく言語として Morphe[57, 56] が提案されいる．しかし，適応化コンポジションでは，環境自体が動的に変化することは考慮されていない．

OMEGA [28] は，オブジェクトがマイグレートした先の環境に適応するための方法として動的なメソッド適合化を行なうモデルである．マイグレートされたオブジェクトは，常に以前と同じオブジェクトと通信し続ける必要はなく，マイグレート先の環境内にある適切なオブジェクトと通信すればよい．メソッドの適合化とは，このようなオブジェクトマ

イグレーションの前後で通信するオブジェクトを切り替えた場合に必要となる操作である．すなわち，あるオブジェクト（マイグレートしたメッセージ送信オブジェクト）から送られたメッセージが，それを受けるオブジェクト（マイグレーション先の新しいメッセージ受信オブジェクト）では理解できない場合に対処するための操作である．OMEGA モデルでは，新しいメッセージ受信オブジェクトで解釈できるようにメッセージを変換することで，メソッドの適合化を実現する．しかし，OMEGA モデルでは，メッセージ受信オブジェクトにメッセージを処理する代替可能なメソッドが存在することを前提としており，そのようなメソッドが存在しない場合，メソッドの適合化を行なうことはできない．

AL-1/D [40, 41] は，マルチモデルという自己反映計算の枠組（MMRF）にもとづいた並列オブジェクト指向言語である．MMRF は，単一のメタレベルを抽象度やユーザの視点によって複数のモデルに分解するための枠組である．さまざまな抽象度やユーザの視点を分解して表現できるため，メタレベルを複雑にすることなくさまざまな機能をモデル化することができる．さらに，MMRF では，複数のモデルが同時に正しく動作することを保証する機構が含まれている．AL-1/D では，メタレベルオブジェクトはベースレベルオブジェクト（通常の計算を行なうオブジェクト）の振舞いを記述するオブジェクトであり，ベースレベルオブジェクトはモデルにもとづく複数のメタレベルオブジェクトと因果的に結合されている．それゆえ，ベースレベルオブジェクトは，モデルにもとづくさまざまな振舞いを実現することができる．また，AL-1/D ではメタレベルオブジェクトがシステム資源の統計情報を持ち，その情報に対する制約を解くことで，メタレベルオブジェクトのメソッドを自動的に起動することができる．しかし，この統計情報は AL-1/D のシステムによって固定されており，その数も少なく，抽象度も低い．

Gaea [38, 37] は，Prolog 風の論理型言語であり，有機的プログラミングと呼ばれる柔軟な協調動作を行なうプログラムを構築する方法論にもとづいた言語である．この方法論は，動作環境の状況推論，セルと呼ばれるプログラム記述単位の操作，および自己反映計算といった概念を併せ持っている．Gaea により記述されたソフトウェアは，セルの組み合わせとプロセスから構成される．セルの組み合わせは Prolog におけるデータベースに相当し，1 つのプロセスは 1 つの Prolog インタプリタに相当する．Prolog では，質問を処理するためのデータベースはそのインタプリタに対して 1 つしか存在しない．一方，Gaea では処理単位であるプロセス毎に状況に応じたデータベースを動的に生成・変更することができる．このデータベース，すなわちセルの組み合わせがプロセスに対する環境となる．プロセスは実行時にその環境を参照し，その環境（セルの組み合わせの構造）を動的に変更することができる．環境の変更はプロセスの実行に反映し，プロセスはその変更に対して自分自身を適応させることができる．しかし，Gaea はプロセスの環境そのものを明示的に表現す

る機構を持たない．

DART [44] は，動的適応可能なソフトウェアのための自己反映的なミドルウェアである．DART は適応のための機構として，adaptive method と reflective method を提供している．adaptive method は，DAS モデルにおける適応可能手続きとほぼ同じ機構である．すなわち，adaptive method は実行環境の状態に応じた複数のインプリメンテーションを持つ．1 つの adaptive method が呼び出されると，その実行は selector によって制御される．selector は，実行環境の状態に適したインプリメンテーションを選択し実行する．一方，reflective method は，ベースレベルのオブジェクトの実行環境をカスタマイズするためのメソッドである．DART では，メタレベルはメタスペースと呼ばれるメタレベルのオブジェクトの集合から構成される．メタスペースはベースレベルのオブジェクトの実行環境を表現している．1 つの reflective method は 1 つのメタスペースと結び付けられる．ベースレベルのオブジェクトが reflective method を通して呼び出されると，reflector によってそのオブジェクトに関連付けられたメタスペースが適切な操作を実現するために呼び出される．実行環境を変更し，アプリケーションの実行方法を変えるために reflective method は使用される．DRAT システムには，実行環境の変化を取得するイベントモジュールとイベントに応じた適応のポリシを実現する機構があり，selector や reflector はこの機構を通じて，選択すべきメソッドのインプリメンテーションやメタオブジェクトを決定する．しかし，DART では，adaptive method やメタスペースの選択機構が固定的で柔軟にカスタマイズすることはできない．

Eden [52] は，環境適応型計算モデルにもとづく並列オブジェクト指向言語である．環境適応型計算モデルとは，オブジェクトが新しい環境に適応するために，その環境の中で動作するために必要な機能や性質を動的に獲得する計算モデルである．Eden における環境とは，複数のオブジェクト間の協調動作により形成される場である．Eden は，オブジェクト間の協調動作に影響を与える文脈や環境を意味するコンテキストという概念を持つ．Eden では，任意のオブジェクトは複数のコンテキスト (環境) に同時に属することができ，各コンテキストごとに異なるロール (役割) を担う．ロールは，オブジェクトがその環境の中で分担して持つべきデータや手続きを実体化したものである．オブジェクトは，ロールを指定して動的にコンテキストに所属することができる．その際，ロールで定義された属性とメソッドがオブジェクトに付加される．逆にオブジェクトがコンテキストから離脱すると，付加された属性とメソッドがオブジェクトから削除される．コンテキストは，計算の進行とともに新しく生成されたり，削除されたり，あるいは，その内容が変更されたりする．Eden は，並列オブジェクト指向言語 ABCL/R3 [34] をカーネル言語とし，その上に実装された言語である．Eden では，ABCL/R3 の自己反映計算機能を利用して，環境，

ロール，環境適応などの概念を言語の機構として実現している．

また，ソフトウェアにおける適応動作の単位をオブジェクトとし，動的なオブジェクトの切り替えにもとづく動的適応可能なソフトウェアの構成方式として，保木本らの提案する方式がある [17, 18]．彼らの方式では，動的適応可能なソフトウェアはオブジェクトグラフと呼ばれるオブジェクトのコンポジションから構成される．オブジェクトグラフは，オブジェクトを節，リンクと呼ばれるオブジェクトへの参照を辺とする有向グラフである．このようなオブジェクトグラフは，各オブジェクトの実装とは独立して記述される．彼らの手法はオブジェクトグラフの構造を動的に変更することで，実行環境の状態変化に対処する，というものである．しかしオブジェクトグラフの構造は静的に決定されており，動的に変更することができない．そのため，彼らは，適応オブジェクトの概念を導入している．適応オブジェクトは複数のオブジェクトグラフを内部に持つオブジェクトであり，実行環境の状態に応じてオブジェクトグラフを切り替えることで実行環境の状態変化に対処する．しかし，保木本らの手法はオブジェクトの内部状態の変換という困難な問題を内包している (4.1.3 節参照)[1]．特に適応オブジェクトがネストした場合には，その内部状態の扱いは極めて複雑になる．また，オブジェクトグラフを切り替える制御 (言わば，適応の戦略) も固定的である．

6.4 その他

その他の関連研究として，以下のものがある．これらは直接動的適応可能なソフトウェアの構築を意図したものではないが，それを実現する上で有効と考えられる機構・機能を持っている．

Compositional Interface [43] は，主としてソフトウェアの再利用のために提案された機構である．この機構は部分オブジェクトを組み合わせる際に，各部分がどのような環境で使われるかを部分オブジェクトに記述する habitat および新しい部品を生成して組み合わせる use と呼ばれる 2 つの言語機能からなる．つまり，部分オブジェクトとその環境 (を表すオブジェクト) の間のインタフェースを部分オブジェクトに明示することにより部分オブジェクトの再利用性を向上させる．しかし，この機構にもとづいて部分オブジェクトを新たに設計する際，それが使われる環境を完全に把握している必要がある．

動的なオブジェクトの変更を可能とする言語に，Actors [3] や Smalltalk-80 [13] がある．これらの言語は，オブジェクトの become 機能を提供している．この機能を使ってオブジェクトは自己同一性を保持しつつ任意の機能およびスロット (インスタンス変数) を持つ他

[1] 実際，彼らはアドホックな方式でこの変換を行なっている．

のオブジェクトになり変わることができる．しかし，オブジェクトは，become 機能を使うことで，何の制限もなく任意のオブジェクトになり変わることができるため，元のオブジェクトとはまったく関係のないものにもなり得る．このため，become 機能を多用するすると，ソフトウェアの管理・保守が困難になると予想される．

Rome [5]，Clovers [49] も動的なオブジェクトの変更を許容するオブジェクト指向言語である [2]．これらの言語では，ひとつのオブジェクトが同時に複数の表現 (Rome では，representation，Clovers では，leaves と呼ばれる) を持つことができる．これらの言語でもオブジェクトの自己同一性を保持しつつ，新しい表現をオブジェクトに付加，削除することが，できる．ただし，オブジェクトの表現はすでに定義されているクラスに対応しており，オブジェクトのクラス階層に沿ったの表現の変更のみを許している．

ABCL/R [55] は，自己反映計算の機構を持つ並行オブジェクト指向言語である．ABCL/R では，オブジェクトはメタオブジェクトによって表現/実現される．メタオブジェクトは，超循環的にオブジェクトの構造的，計算論的側面のメタレベル表現を具体化している．そのようなメタオブジェクトもまた ABCL/R のオブジェクトとして実現されている．ABCL/R は，自己反映計算のための言語機能を提供しており，その機能を使って並行システムで動作中のオブジェクトの動的な変更を実現することができる．たとえば，他のオブジェクトが持つメソッドを実行時に問い合わせ，自己反映計算によりそのメソッドを動的に獲得する，といったことができる．また ABCL/R の後継言語として，ABCL/R2 [35]，ABCL/R3 [34] などが提案されている．

CLOS [48, 26, 22, 24] は，Common Lisp 上に構築された自己反映オブジェクト指向言語システムである．CLOS では，CLOS を構成する諸言語要素 (クラス/総称関数/メソッド) を CLOS 自身のオブジェクトとして表現するメタオブジェクトが導入されている．これにより，言語システムを実現する諸要素を，プログラム中から参照・変更することができる．CLOS では，メタオブジェクトの外部仕様をプロトコル (メタオブジェクトプロトコル) として提供している．ユーザは，このプロトコルにしたがって，言語システムの拡張・変更を柔軟に行なうことができる．

RbCl [20, 21] は，自己反映アーキテクチャにもとづく並列オブジェクト指向言語である．RbCl の処理系は固定された実行時のカーネルを持たず，ユーザにより，言語処理系のほとんどすべての振舞いをカスタマイズすることが可能となっている．RbCl のプログラムは，メタレベルのオブジェクトから構成されるメタシステムによって実行される．特に重要なメタレベルのオブジェクトは，システムオブジェクトと呼ばれ，システムオブジェクトテーブルに登録されている．ユーザはシステムオブジェクトテーブルに登録されているシステ

[2] 厳密には，Clovers はオブジェクト指向言語のための機構である．

ムオブジェクトをユーザ定義のオブジェクトと交換することでメタシステムの振舞いを動的に変更することができる．RbCl は言語的共生と呼ばれる機構をメタレベルのオブジェクトを使って実現している．RbCl のすべてのメタレベルのオブジェクトは，実際には C++のオブジェクトであるが，言語的共生の機構により，メタレベルの C++によるオブジェクトを RbCl のオブジェクトとして操作可能にしている．

OpenC++ [8, 9] は，自己反映計算にもとづいた拡張可能な C++言語である．OpenC++では，メソッド呼び出しと変数アクセスの実現機構が C++のオブジェクトで実現されており，これは処理系のメタシステムとしてユーザに公開されている．OpenC++のオブジェクトはメタオブジェクトを持つことができ，メタオブジェクトはそのオブジェクトへのメソッド呼び出しと変数アクセスの実現機構を定めている．メタオブジェクトを持つオブジェクトを自己反映オブジェクトと呼び，自己反映オブジェクトにより制御されるメソッドや変数を reflect メソッド，reflect 変数と呼ぶ．reflect メソッドの呼び出しや reflect 変数のアクセスはメタオブジェクトの記述に従って実現される．OpenC++は，メタオブジェクトの記述方法を定めた簡単なプロトコル（メタオブジェクトプロトコル）を提供している．プログラマは，OpenC++のメタオブジェクトプロトコルを用いてメタオブジェクトをカスタマイズし，そのオブジェクトのメソッド呼び出しと変数アクセスの機構を容易にカスタマイズすることができる．

Cecil [6, 7] は，CLOS における総称関数のような多重メソッドを持つオブジェクト指向言語である．Cecil は，その言語要素として述語クラスを提供している．述語クラスは，通常のクラスと同様に親クラス，メソッド，インスタンス変数を持つが，そのインスタンスは述語クラスと関連付けられた述語式が満たされる時，自動的に生成されるという特徴を持つ．このような述語クラスとメソッドを関連付けることにより，メソッドは引数のクラスだけでなく，その動的な値や状態に応じて呼び出されるようになる．このような Cecil のメソッドは，DAS モデルで提供している単純な戦略（AND 戦略や OR 戦略）にもとづく適応可能メソッドと同等の機能を実現していると言える．

AOP [27] は，プログラムのさまざまな側面を別々に記述し，それらを統合することで，プログラムを構成するプログラミング技法である．AOP では，プログラムはコンポーネントとアスペクトに分けられ，別個に記述される．コンポーネントは，汎用的な手続きとしてカプセル化が可能な部分であり，アスペクトはそのようなカプセル化が不可能な部分である．これらはの記述は，Aspect Weaver と呼ばれるツールによってマージされる．AOP 自体は，動的適応可能なソフトウェアを強く意識した技法ではないが，適応動作の記述をコンポーネント，適応動作の制御記述をアスペクトとすることで，動的適応可能なソフトウェアを構築する方法として利用できる．ただし，AOP は，Aspect Weaver を記述する言

語に依存する度合が大きい．

6.5 DAS モデルの特徴

これらの関連研究に対して，本研究で提案している DAS モデルの持つ特徴的な点として以下をあげることができる．

- 特定のパラダイムや記述言語に強く依存しない汎用的なモデルである．
- 適応の機構そのものが実行環境の状態に対して適応可能である．
- 適応の機構（特に，適応の戦略）は，その構造を修正することなく適応可能メソッドのインクリメンタルな追加で拡張・変更することができる．
- 仮想的かつ局所的な実行環境の抽象度に応じた非同期イベント機能を持つ．

先に述べた関連研究は，特定のパラダイムや記述言語に依存するものが少なくない．たとえば，多くの関連研究が，オブジェクト指向パラダイムや言語に特化したアプローチを採用している．それゆえ，これらを非オブジェクト指向パラダイムにもとづくソフトウェアや言語に適用することはできない．これに対して，DAS モデルはオブジェクト指向，非オブジェクト指向を問わず手続き呼び出しの機構を持つ大半の言語に適用することが原理的に可能である[3]これは，DAS モデルを採用することで，既存のソフトウェア資産を有効に活かして動的適応可能なソフトウェアを実装できることを意味している．既存のソフトウェア資産を有効利用することは，ソフトウェアの開発コストを軽減することにもつながる．

また，上記の関連研究では，その適応の機構（特に，適応の戦略）がシステムによって固定されているものが多い．そのため，これらの関連研究では柔軟性のある動的適応可能性を実現するのは困難である．DAS モデルでは，適応の機構自体が適応可能手続きを使って実現されており，それ自身が実行環境の状態変化に適応することができる．適応の機構を実行環境の状態に応じて動的に切り替えること（すなわち，適応の戦略を動的に変更する）は，有限の適応可能メソッドをより有効利用することにつながる．

関連研究のアプローチにおいて，適応動作の単位を手続きとするものは少ない．それゆえ，これらの関連研究のアプローチを用いて DAS モデルの適応可能手続きを実現することは容易ではない．たとえば，CLOS や OpenC++を用いて，適応可能手続きを実現することを考えた場合，それは不可能ではないが容易でもない．その理由は，適応可能手続き

[3]オブジェクト指向言語の場合は，メソッド呼び出しの機構．

を構成する適応可能メソッドは，その引数の型や数，戻り値型がすべて等しくても実行環境の状態によって異なるメソッドとして認識されなければならないからである．Cecil は述語クラスに関連付けられたメソッドにより，DAS モデルの適応可能手続きを実現することができるが，それは単純な戦略 (AND 戦略や OR 戦略) にもとづく適応可能手続きのみである．しかも，Cecil では，適応の戦略の拡張や，戦略の動的な変更といった点については考慮されていない．DART は，DAS モデルにおける適応可能メソッドに似た adaptive method を提供している．しかし，DART では，適応の戦略がやはりシステムによって固定されており，ユーザによる容易なカスタマイズを許容していない．これに対して，DAS モデルでは，適応の戦略を動的に変更することができる上，戦略を実現する適応可能メソッドを追加するだけで，カスタマイズすることができる．DART における自己反映計算は，DAS モデルにおけるそれとはまったく異なる．DAS モデルでは，自己反映計算にもとづいて適応の戦略を動的に変更することで，実行環境の状態に応じたソフトウェアの機能の一時的な制限やその解除といった柔軟な適応動作をベースレベルの記述を修正することなく実現できる．

また，関連研究であげたものの中には，動的適応可能なソフトウェアの実現に有効な自己反映計算の機構を持つものが多数みられた．これらの研究は，動的適応可能なソフトウェアを記述することができる潜在的な能力を持つと言える．しかし，これらの多くは，主として自己のシステムを拡張するために自己反映計算の機構を利用しており，動的適応可能なソフトウェアの構成を意図して実装・考案されたものではない．前述したように，これらの自己反映計算の機構を利用して，動的適応可能なソフトウェアを実現することも不可能ではないが容易でもない．

DAS モデルはソフトウェアに実行時の動的適応可能性を提供するだけでなく，動的適応可能性の拡張機構も提供する．事前に予測できなかった実行環境の状態にあとで対処するためには，適応可能メソッドを追加することで対応する．DAS モデルでは，適応の機構 (特に，適応の戦略) 自体が適応可能手続きとして実現されていることから，この拡張でさえも適応可能メソッド (戦略を実装する) のインクリメンタルな追加でそのプログラムの構造を修正することなく実現できる．条件分岐を用いて実行環境の状態を場合分けするようなアドホックな実装方式と比較した場合，その利点は明らかである．

さらに，DAS モデルは仮想的かつ局所的な実行環境を表現する環境オブジェクトの状態変化を検出し，その変化に対処するコールバックメソッドを呼び出すイベントオブジェクトを提供する．AL-1/D や DART でも，この種のイベントオブジェクトを提供しているが，これらのイベントオブジェクトはその種類と抽象度がシステムによって固定されている．たとえば，DAS モデルのイベントオブジェクトのようにユーザが定義した環境オブジェクト

の状態変化を監視するようにそれらのイベント機構をカスタマイズすることは困難である．

また，本章であげたオペレーティングシステム (OS) は，その上で動作するアプリケーションを最適に実行するために，その構成を動的に変更したり，アプリケーションの抽象度に応じた実行環境を提供するなどを意図して設計されている．つまり，これらの OS は DAS モデルを実現するプラットフォームとして有効である．これらの OS は，各アプリケーションに特化したサービスや抽象化を提供することができるからである．それゆえ，これらの OS と DAS モデルは相補的な関係と位置付けられる．

そして，本章であげた移動コード/エージェントシステムについても同様のことが言える．DAS モデルは，特定の移動コード/エージェントシステムから独立した存在であり，そうしたプラットフォームが提供する機能 (計算機間の移動や通信などを実現する機能) を提供していない．逆に，既存の移動コード/エージェントシステムの多くは，その移動コード/エージェントに対して，適応動作のための機構を提供していない．つまり，これらの移動コード/ エージェントシステムと DAS モデルを組み合わせることで，動的適応可能な移動エージェントを実現することができるようになる．この点について，本研究では，Java ベースの移動エージェントシステム AgentSpace と DAS クラスライブラリを組み合わせて，動的適応可能な回覧板システムを実現し，これを実証している (6 章参照)．

しかし，DAS モデルにも欠点がないわけではない．たとえば，以下は DAS モデルにもとづいて，実際に動的適応可能なソフトウェアを実現する際に問題となる点である．

- 適応可能手続きを記述する際，常に適応の戦略について考慮する必要がある．
- 適応の戦略の変更には十分な注意を要する．
- リアルタイム性が要求されるシステムへの適用は難しい．

DAS モデルでは，適応動作 (適応可能手続き) と適応動作の制御 (適応の戦略) が分離されている．それゆえ，事前に予測できなかった実行環境の状態に対処する適応可能メソッドの追加定義を行なう際，適応の戦略の記述を修正する必要はない．しかし，その適応可能手続きが，どの戦略を使用しているのかを知らなければ，適応可能メソッドのディスパッチ条件を記述することができない．このため，既に定義されている適応可能メソッドのディスパッチ条件を調べたり，新たに定義する適応可能メソッドが使用する環境オブジェクトを調べたりする必要が生じる．つまり，適応可能手続きと適応の戦略は完全に分離されているものの，密接に関係しているため，それら双方を常に考慮しなければならない．

また，DAS モデルでは，適応の戦略を変更することで，有限の適応可能メソッドを最大限に活用することを可能にしているが，戦略の変更には十分な注意を要する．戦略の変更

は環境オブジェクトという局所的な範囲で行なわれ，その影響は該当する環境オブジェクトに限定されるが，その環境オブジェクトがどこで使用されているか，つまりその環境オブジェクトを使用している適応可能手続きを注意深く特定する必要がある．さもないと，戦略の変更により適応可能手続きの予期しない動作を引き起こす結果となる．これは環境オブジェクトと適応可能手続きを 1 対 1 対応とすることで回避できるが，こうした記述は繁雑であり，かつ環境オブジェクトの再利用・共有という観点から好ましい対策とは言えない．というのも，環境オブジェクトを共有する関連した一群の適応可能手続きの振舞いを同時に変更できる，という長所も失われるからである．

DAS モデルにおけるソフトウェアの適応動作は手続きにもとづき，その制御は手続き呼び出しにもとづいて実現される．そのため，手続きよりも小さな粒度の適応動作が必要となる場合，それを支援することは難しい．たとえば，適応可能手続きの呼び出し中に実行環境の状態が変化した場合，その変化に対処することは困難である．適応可能手続きを入れ子にすることはできるが，これは本質的な解決策とは言えない．また，対処しなければならない状態が増えた場合，それに比例して適応可能手続き中の適応可能メソッドも増えることになる．適応可能メソッドの数が多くなれば，そのディスパッチに要する時間は当然大きくなる．これらの点を考慮すると，リアルタイム性が要求されるようなシステムに DAS モデルを適用することは，あまり実用的ではない．

6.6 まとめ

本章では，動的適応可能なソフトウェアに関するさまざな研究を以下のように分類し，概説した．

- オペレーティングシステム：Apertos，SPIN，Exokernel など．
- 移動コード/エージェントシステム：Flage，Plangent，Telescript など．
- 言語・モデル・機構：適応化コンポジション，OMEGA，AL-1/D，Gaea，DART，Eden など．
- その他：Compositional Interface，Actors，Smalltalk-80，Rome，Clovers，ABCL/R，CLOS，RbCl，OpenC++，Cecil，AOP など．

また，これらの関連研究と DAS モデルの比較を行ない，DAS モデルの特徴的な点（長所および短所）について述べた．関連研究のうち，オペレーティングシステムや移動コード/エージェントシステムに関するものは，DAS モデルと相補的な関係にある．

第 7 章

結論および今後の課題

本研究では，動的適応可能なソフトウェアのためのモデル DAS を提案し，DAS モデルの基本機構を Java 言語のクラスライブラリ (DAS クラスライブラリ) として設計・実装した．そして，DAS クラスライブラリを用いて実際に動的適応可能なソフトウェアを作成した．本論文では，これらの設計・実装を通して，開放型分散システム環境や移動計算機環境における実行環境の動的な状態変化に適応できるソフトウェアの実現方法を示し，その有効性について述べた．本節では，本研究により得られた結果を総括し，今後の課題について述べる．

7.1 動的適応可能なソフトウェアモデル DAS

本研究で提案している DAS モデルの主要な概念は，以下である．

- ソフトウェアにおける適応動作の単位を手続きとし，手続き呼び出しにもとづく適応動作の制御を行なう．
- DAS モデルは，適応可能手続きと呼ばれる実行環境の状態にもとづく総称手続き (総称関数) を持つ．
- 適応可能手続きの制御機構は，自己反映計算を利用して実現される．
- DAS モデルにもとづく動的適応可能なソフトウェアは，メタレベルアーキテクチャを形成する．

DAS モデルでは，ソフトウェアにおける適応動作の単位を手続きとし，適応動作は適応可能手続きと呼ばれる実行環境の状態に応じた総称手続きとして実現される．ひとつの適

応可能手続きは，実行環境の状態に応じた実行コードを持つ複数の適応可能メソッドから構成される．適応可能手続きが呼び出されると，実行環境の状態に応じて適切な適応可能メソッドが選択され，実行される．この手法は，原理的に手続き呼び出しの機構を持つ多くの既存言語により記述されたソフトウェアに適用できる．この結果，既存のソフトウェア資産を有効利用できる可能性が高まる．本研究では，DAS モデルの基本機構を Java 言語のクラスライブラリとして実装し，実際に Java のアプリケーションに動的適応可能性の導入を行なった．これらの実装を通じて，DAS モデルが既存言語およびそのソフトウェア対して高い親和性を持つことを実証した．

そして DAS モデルでは，適応可能手続きの制御機構 (適応の機構) 自身が適応可能手続きを用いて自己反映的に実現される．つまり，適応の機構自身も実行環境の状態に適応することができる (実行環境の状態変化に応じて変化できる)．実行環境の状態変化に応じて，適応の機構を変えることで，実行環境の状態にもとづくソフトウェアの機能の一時的な制限やその解除といった柔軟な適応動作を元のプログラムを修正することなく実現することができるようになる．さらに適応の機構が，適応可能手続きを用いて実現されるため，この機構のカスタマイズはその構造を修正することなく，適応可能メソッドのインクリメンタルな追加で実現できる．これらの利点について，例題の実装を通して実際に確かめることができた．また，ソフトウェアにおける柔軟な適応動作の有効性も実際に例題を用いて示した．

さらに，DAS モデルにもとづく動的適応可能なソフトウェアは，メタレベルアーキテクチャを形成する．メタレベルはソフトウェアにおける適応動作の制御を実現する機構の記述からなり，ベースレベルは各ソフトウェアにおける問題領域の記述からなる．つまり，適応動作の制御記述はベースレベルの記述から完全に分離され，メタレベルに隠蔽される．この記述の分離は，動的適応可能なソフトウェアの可読性，保守性，拡張性を向上させる．この利点について，例題の実装を通して実際に確かめることができた．

また，DAS モデルは仮想的かつ局所的な実行環境を表現する環境オブジェクトを提供し，環境オブジェクトの状態変化をアプリケーションに通知する非同期イベント機能を持つ．本研究では，この非同期イベント機能を用いた動的適応可能なソフトウェアを実際に作成し，その有効性を確かめた．

7.2 DAS クラスライブラリ

DAS クラスライブラリは，DAS モデルの基本機構を実現する Java 言語のクラスライブラリである．DAS クラスライブラリは，DAS モデルのメタレベルを構成するオブジェクト

(メタオブジェクト) を提供し，適応可能手続きの実現を支援する．メタオブジェクトの利用および適応可能手続きの記述には若干のコーディング規則がある．また，DAS クラスライブラリはメタオブジェクトを拡張するための簡易なプロトコル (メタオブジェクト) を含んでいる．このメタオブジェクトプロトコルにもとづいてユーザはメタオブジェクトを柔軟に拡張することができる．

本研究では，実際に DAS クラスライブラリを用いて移動コードベースの動的適応可能な回覧板システムを実装し，移動コードシステムに対する本研究の有意性を示した．このシステムにおける回覧板エージェントは，GUI の変更を行なう，ユーザによる入力欄の enable/disable を行なう，ユーザの不在や，計算機のダウンに対して，次のホストへ移動する，といった適応動作を実現する．この回覧板エージェントの実装では，移動コードのプラットフォームとして Java ベースの移動コードシステム AgentSpace を利用したが，DAS クラスライブラリ自体は，特定の移動コードシステムに依存するものではない．

7.3 今後の課題

本研究に関する今後の課題として，特に以下をものをあげることができる．

- 適応可能手続きの構文支援と構文解析器の設計・実装．
- 動的適応可能なソフトウェアの評価．
- 動的適応可能なソフトウェアの検証．
- DAS モデルに対する並行計算の概念の導入．

適応可能手続きの構文支援と構文解析器の設計・実装：

本研究の目的は，動的適応可能なソフトウェアのモデルを考案し，そのモデルにもとづくプログラミング言語の基本機構を設計・実装することであった．その目的は，DAS モデルの考案と DAS クラスライブラリの設計・実装で十分に達成できた．しかし，実際に動的適応可能なソフトウェアを効果的に記述するには，DAS クラスライブラリだけでは十分ではない．DAS クラスライブラリを用いて適応可能手続きを実現する際，以下の問題点がある．

- 1 つの適応可能手続きにつき，1 つの文字列と 2 つのメソッド (条件メソッドとコードメソッド．5.1.2 節参照) を定義しなければならない．

- 適応可能メソッドと通常のメソッドを区別するために，メソッド名の接頭辞にキーワードをつけなければならない．
- 適応可能メソッド名は一意でなければならず，その接尾辞にポストフィックス番号 (数字) を付加する必要がある．

こうした問題は，適応可能手続きに対する記述面の利便性を考慮すると非常に繁雑である．さらに，これらの問題はすべて組織的に扱うことができる．つまり，適応可能手続きを支援する構文を考案し，その構文から上記の Java コードに変換することが十分に可能であると考えられる．それゆえ，今後の課題としては，適応可能手続き (適応可能メソッド) の構文を考案し，構文解析器を設計・実装して Java の前処理系を作成することがあげられる．これにより，記述の容易さを向上させ，かつ Java との互換性を保つことが可能になる．

動的適応可能なソフトウェアの評価：

本研究では，DAS クラスライブラリを用いて動的適応可能な回覧板システムを作成したが，これ関して具体的な評価を行なってはいない．その理由は，動的適応可能なソフトウェアに関する定量的な評価を行なうことが難しいからである．たとえば，動的適応可能な回覧板システムを例に，その評価について考えることにする．動的適応可能な回覧板エージェントの適応動作は，以下のようなものであった．

- ディスプレイのサイズに応じて GUI を変更する．
- ユーザごとに入力欄の enable/disable を行なう．
- 30 分以上ユーザからの入力がなければ隣のホストに移動する．
- バッテリ切れを検出し，隣のホストに移動する．

GUI の変更は，回覧板の look & feel および操作性の向上を意図した適応動作であり，これを定量的に評価することは困難である．これに関する評価方法としては，複数の被験者によるモニター実験を行なうことが考えられるが，被験者の嗜好や経験などにより評価が左右される可能性があり，必ずしも有効な評価方法であるとは言えない．

GUI 以外の適応動作に関しても同様のことが言える．つまり，それらは有効ではあるが，その有効性を定量的に示しにくいのである．このような動的適応可能なソフトウェアを評価することは容易ではない．現在，適応動作の種類を限定して評価をとることを予定している．回覧板システムの例でも分かるように，適応動作にはさまざまな種類のものがある．それらについて統一的な評価方法を策定することは現実的では

ない．たとえば，実行効率をあげるための適応動作などに限定して，その評価を行なう．具体的には，2 章で述べたようなマルチメディアアプリケーションについて，その動画像を間引くことで，アプリケーションの実行速度がどの程度向上するかを測定する，といった評価を検討している．

動的適応可能なソフトウェアの検証：

本研究で提案している動的適応可能なソフトウェアは，実行環境の状態に応じて，実行する手続きのコードを動的に変更する．すなわち，ソフトウェアの部分的なアルゴリズムを実行時に変える．これはプログラムの操作的な意味を実行時に変えることに他ならない．このため動的適応可能なソフトウェアについて，その実行時の動作に関する正当性を検証することは困難である．しかし，動的適応可能なソフトウェアが実行時に意図した適応動作をするか否かは重要な問題である．この問題に関して，形式仕様記述言語 CafeOBJ を用いて，DAS モデルにもとづく動的適応可能なソフトウェアの操作的意味を定義することを予定している（特に適応可能メソッドのディスパッチ部分の仕様記述）．形式仕様記述言語は，いくつか提案されているが，CafeOBJ は，実行可能な処理系を持つため，単に仕様を与えるだけでなく，その記述を処理系により実行することができる．この処理系による仕様記述の実行を通して，記述の検証や解析を自動的に行なうことができる．

DAS モデルに対する並行・同期の概念の導入：

本研究で提案している動的適応可能なソフトウェアモデル DAS は，適応動作の単位を手続きに設定しているため，その適応動作の影響は，同一アプリケーション内に制限される．しかし，今日ソフトウェアの規模は，ますます大きくなってきており，複数のアプリケーションの組み合わせからひとつのソフトウェアが構成されることが多くなってきている．たとえば，コンポーネントウェアなどは，その好例と言える．こうした複数のアプリケーションから構成される大規模なソフトウェアは，そのアプリケーション間の関係やインタラクションをどう実現するかが鍵となる．

このような複合アプリケーションに動的適応可能性を導入するには，DAS モデルだけでは十分ではない．この問題に対処するために，DAS モデルに並行・同期などの概念の導入を検討している．具体的には，これらの概念を導入して DAS モデルの拡張を行ない，拡張 DAS モデルにもとづく言語の設計・実装を検討している．また，その例題として移動コードベースのマルチエージェントシステムの動的適応可能化などを考えている．

謝辞

本研究を行なうにあたり，終始ご指導して下さった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言語設計学講座助教授 渡部卓雄 博士，同講座教授 二木厚吉 博士に心から感謝致します．

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ソフトウェア基礎講座 教授 片山卓也 博士，同大学院情報科学研究科情報システム学専攻計算機アーキテクチャ講座教授 日比野靖 博士，東京大学大学院理学系研究科情報科学専攻教授 米澤明憲 博士から，有益な助言を頂きました．ここに深く感謝の意を表します．

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計算機アーキテクチャ講座助教授 中島達夫 博士からも，貴重な助言を頂きました．ここに感謝致します．

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言語設計学講座助手 緒方和博 博士には，日頃から議論につきあって頂き，数々の研究上のアドバイスを頂きました．同博士に心から感謝致します．

お茶の水女子大学理学部情報科学科助教授 佐藤一郎 博士には，同博士が実装された移動エージェントシステム AgentSpace の利用を快諾して頂きました．ここに感謝致します．

また，日頃より共に研究活動を行なってきた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言語設計学講座の諸氏に感謝します．

本研究は，情報処理振興事業協会（IPA）「高度情報化支援ソフトウェアシーズ育成事業」の一環として行なわれたものである．

参考文献

[1] GNU Guile. http://www.gnu.org/software/guile/.

[2] The Caml Language. http://pauillac.inria.fr/caml/.

[3] Gul Agha and Carl Hewitt. *Concurrent Programming Using Actors*. The MIT Press, 1987.

[4] Brain N Bershad, Stefan Savage, Przemyslaw Pardyak, and Emin Gun Sirer. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pp. 267–284, 1995.

[5] Bernard Carre, Lenneke Dekker, and Jeanmarc Geib. Multiple and Evolutive Representation in the Rome Language. In *Proceedings of Technology of Object-Oriented Language and Systems (TOOLS'90)*, June 1990.

[6] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92), Lecture Note Computer Science, Vol. 615, Springer-Verlag*, pp. 33–56, June/July 1992.

[7] Craig Chambers. Predicate Classes. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93), Lecture Note Computer Science, Vol. 707, Springer-Verlag*, pp. 268–296, July 1993.

[8] Shigeru Chiba and Takashi Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93), Lecture Note Computer Science, Vol. 707, Springer-Verlag*, pp. 482–501, July 1993.

[9] 千葉滋, 益田隆司. 自己反映言語 open c++ とその分散処理への適用の実際. コンピュータソフトウェア, Vol. 11, No. 3, pp. 33–48, May 1994.

[10] Ralph Droms. Dynamic Host Configuration Protocol (RFC 1541). October 1993.

[11] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, 1995.

[12] George H Forman and John Zahorjan. The Challenges of Mobile Computing. Technical Report TR-93-11-03, Department of Computer Science and Engineering University of Washington, FR-35 Seattle WA 98195, 1994.

[13] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-wesley publishing company, 1983.

[14] Dean W Gonzalez. *Ada – Programmer's Handbook and Language Reference Manual –*. The Benjamin/Cummings Publishing Company, Inc, 1991.

[15] James Gosling, Bill Joy, and Guy L Steele. *The Java Language Specification*. The Java Series. Addison-wesley publishing company, 1996.

[16] Robert S Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of the CIKM'95 Workshop on Intelligent Information Agents*, December 1995.

[17] Akihiro Hokimoto, Kuniaki Kurihara, and Tatsuo Nakajima. An Approach for Constructing Mobile Applications using Service Proxies. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96)*, May 1996.

[18] 保木本晃弘, 中島達夫. オブジェクトグラフに基づくカスタマイズを用いた動的適応ソフトウェア構築のためのアプローチ. ソフトウェア工学の基礎 4, pp. 135–142. 日本ソフトウエア科学会第 4 回ソフトウェア工学の基礎ワーク ショップ (FOSE'97), 近代科学社, December 1997.

[19] 本田康晃, 渡滋, 所真理雄. 適応化コンポジション. コンピュータソフトウェア, Vol. 9, No. 2, pp. 48–62, March 1992.

[20] 一杉裕志, 松岡聡, 米澤明憲. 実行時カーネルのないリフレクティブな分散オブジェクト指向言語の設計と実装. オブジェクト指向コンピューティング I, pp. 31–47. 日本ソフトウエア科学会 第 8 回オブジェクト指向計算ワークショッ プ (WOOC'92), 近代科学社, March 1992.

[21] 一杉裕志, 松岡聡, 米澤明憲. 実行時カーネルのないリフレクティブな並列オブジェクト指向言語の実現方法. コンピュータソフトウェア, Vol. 11, No. 3, pp. 65–77, May 1994.

[22] 井田昌之, 元吉文男, 大久保清貴編. Common Lisp オブジェクトシステム –CLOS とその周辺–. 共立出版, 1989.

[23] Tomasz Imielinski and Henry F. Korth, editors. *MOBILE COMPUTING*. Kluwer Academic Publishers, 1996.

[24] 稲葉浩人, 河込和宏訳. Common Lisp オブジェクト指向 (CLOS). トッパン, 1991.

[25] Richard A Kelsey and Jonathan A Rees. A Tractable Scheme Implementation. *Lisp and Symbolic Computation*, Vol. 7, No. 4, 1995.

[26] Gregor Kiczales, Jim des Rivieres, and Daniel G Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[27] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'97), Lecture Note Computer Science, Vol. 1241, Springer-Verlag*, pp. 220–242, 1997.

[28] 岸本芳典, 小高信人, 本位田真一. OMEGA: マイグレーションのための動的メソッド適合化モデル. オブジェクト指向コンピューティング I, pp. 49–65. 日本ソフトウエア科学会 第 8 回オブジェクト指向計算ワークショップ (WOOC'92), 近代科学社, March 1992.

[29] Fumihiro Kumeno and Shinichi Honiden. Flage: A Programming Language for Adaptive Software. In *Proceedings of the International Symposium on New Models for Software Architecture (IMSA'97)*, pp. 58–69, October 1997.

[30] Danny B Lange and Daniel T Chang. IBM Aglets Workbench, Programming Mobile Agents in Java – A White Paper (draft). August 1996.

[31] Danny B Lange, Mitsuru Oshima, and Kazuya Kosaka. Aglets: Programming Mobile Agents in Java. In *Proceedings of the World-Wide Computing and its Applications (WWCA'97), Lecture Note Computer Science, Vol. 1274, Springer-Verlag*, pp. 253–266, March 1997.

[32] Pattie Maes. Computational Reflection. Ph. D. Thesis 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[33] Pattie Maes and Daniele Nardi, editors. *Meta-level Architecture and Reflection*. ELSEVIER SCIENCE PUBLISHERS B.V., 1988.

[34] Hidehiko Masuhara and Akinori Yonezawa. Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), Lecture Note Computer Science, Vol. 1445, Springer-Verlag*, pp. 418–439, July 1998.

[35] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP'91), Lecture Note Computer Science, Vol. 512, Springer-Verlag*, pp. 231–250, July 1991.

[36] Francis G McCabe and Keith L Clark. *April – Agent PRocess Interaction Language*. No. 890 in Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1995.

[37] Hideyuki Nakashima. Organic Programming for Cooperative Computation. In *Proceedings of the World-Wide Computing and its Applications (WWCA'97), Lecture Note Computer Science, Vol. 1274, Springer-Verlag*, pp. 133–141, March 1997.

[38] Itsuki Noda and Hideyuki Nakashima. Multiagent Soccer on Gaea. In *Proceedings of the International Symposium on New Models for Software Architecture (IMSA'96)*, pp. 9–13, December 1996.

[39] Akihiko Ohsugi, Yasuo Nagai, Yutaka Irie, Masanori Hattori, and Shinich Honiden. PLANGENT: An Approach to Making Mobile Agents Intelligent. *IEEE Internet Computing*, Vol. 1, No. 4, pp. 50–57, August 1997.

[40] 岡村英明, 石川裕, 所真理雄. リフレクション機能を持つ分散プログラミング言語システム AL-1/D. オブジェクト指向コンピューティング I, pp. 1–16. 日本ソフトウエア科学会 第 8 回オブジェクト指向計算ワークショップ (WOOC'92), 近代科学社, March 1992.

[41] 岡村英明, 石川裕, 所真理雄. 自己反映的分散プログラミングシステム AL-1/D におけるオブジェクトの動的再配置. コンピュータソフトウェア, Vol. 11, No. 3, pp. 49–64, May 1994.

[42] Charles Perkins. IP Mobility Support (RFC 2002). October 1996.

[43] Rajendra K Raj and Henry M Levy. A Compositional Model for Software Reuse. In Stephen Cook, editor, *Proceedings of the 3th European Conference on Object-Oriented Programming (ECOOP'89)*, British Computer Society Workshop Series, pp. 3–24. Cambridge University Press, July 1989.

[44] Pierre-Guillaume Raverdy, Hubert Le Van Gong, and Rodger Lea. DART: A Reflective Middleware for Adaptive Applications. In *Technical Report of Center for Computational Physics, University of Tsukuba 98-4*, pp. 37–40, October 1998.

[45] 佐藤一郎. AgentSpace: 高階モーバイルエージェントシステム. 信学技法 COMP97-92, SS97-63, pp. 41–48. 社団法人 電子情報通信学会, March 1998.

[46] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'84)*, pp. 23–35, 1984.

[47] James W Stamos and David K Gifford. Remote Evaluation. *Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 12, No. 4, pp. 537–565, October 1990.

[48] Guy L Steele. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[49] Lynn Andrea Stein and Stanley B Zdonik. Clovers: The Dynamic Behavior of Types and Instances. Technical Report CS-89-42, Department of Computer Science, Brown University, 1989.

[50] 寺岡文男. VIP: ホスト移動透過性を提供するプロトコル. コンピュータソフトウエア, Vol. 10, No. 4, pp. 22–38, July 1993.

[51] Fumio Teraoka, Keisuke Uehara, Hideki Sunahara, and Jun Murai. VIP: A Protocol Providing Host Mobility. *Communications of the ACM*, Vol. 37, No. 8, pp. 67–75, August 1994.

[52] 鵜林尚靖, 玉井哲雄. オブジェクト間協調に基づく環境適応型プログラミング言語 Eden の設計. 情報処理学会論文誌, Vol. 39, No. SIG1(PRO1), pp. 50–60, December 1998.

[53] Jab Vitek and Christian Tschudin, editors. *Mobile Object Systems – Towards the Programmable Internet –*. No. 1222 in Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1997.

[54] 渡部卓雄. リフレクション. コンピュータソフトウェア, Vol. 11, No. 3, pp. 5–14, May 1994.

[55] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88)*, pp. 306–315, 1988.

[56] Shigeru Watari. An Experience in Morphe: Dynamic Binding in Part-Whole Graphs to Enhance Reusability. Technical Report SCSL-TR-93-022, Sony Computer Science Laboratory Inc, 1993.

[57] Shigeru Watari, Yasuaki Honda, and Mario Tokoro. Morphe: A Constraint-Based Object-Oriented Language Supporting Situated Knowledge. Technical Report SCSL-TR-92-005, Sony Computer Science Laboratory Inc, 1992.

[58] Peter Wayner. *AGENTS UNLEASHED – A Public Domain Look at Agent Technology –*. Academic Press, Inc, 1995.

[59] James E White. Telescript Technology – Mobile Agents – White Paper. February 1996.

[60] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, pp. 414–434, 1992.

本研究に関する発表論文

[1] Noriki Amano, Takuo Watanabe: "LEAD++: An Object-Oriented Reflective Language for Dynamically Adaptable Software Model", IEICE TRANSACTIONS on Fundamental of Electronics, Communications and Computer Science (Jun, 1999) (to appear).

[2] 天野憲樹, 渡部卓雄: "LEAD :動的適応可能なソフトウェアを構成する言語の設計と実装", 情報処理学会論文誌, 第 39 巻 第 9 号, pp.2649-2658 (Sep, 1998).

[3] Noriki Amano, Takuo Watanabe: "LEAD: A Language for Dynamically Adaptable Applications", IEICE TRANSACTIONS on Fundamental of Electronics, Communications and Computer Science, Vol.E81-A N0.6, pp.992-1000 (Jun, 1998).

[4] Noriki Amano, Takuo Watanabe: "LEAD++: An Object-Oriented Reflective Language for Dynamically Adaptable Software", OOPSLA'98 International Workshop on Reflective Programming in C++ and Java, Technical Report of Center for Computational Physics, University of Tsukuba 98-4, pp.91-95 (Oct, 1998).

[5] Noriki Amano, Takuo Watanabe: "LEAD++: An Object Oriented Language for Dynamically Adaptable Software Model", IEICE International Technical Conference on Circuit/Systems,Computers and Communications (ITC-CSCC'98), pp.1143-1146 (Jul, 1998).

[6] Noriki Amano, Takuo Watanabe: "A Procedural Model of Dynamic Adaptability and Its Description Language", ICSE'98 International Workshop on Principles of Software Evolution (IWPSE'98), pp.103-107 (Apr, 1998) (revised version to appear in ACM press).

[7] Noriki Amano, Takuo Watanabe: "LEAD: a linguistic approach to dynamic adaptability for practical applications", IFIP International Conference on Systems Imple-

mentation 2000: Languages, Methods and Tools, Chapman & Hall, pp.277-290 (Feb, 1998).

[8] Noriki Amano, Takuo Watanabe: "A Linguistic Approach for Dynamically Adaptable Applications", IPSJ International Symposium on Information Systems and Technologies for Network Society, World Scientific, pp.403-406 (Sep, 1997).

[9] Noriki Amano, Takuo Watanabe: "LEAD: A Language for Adaptable Applications ", IEICE International Technical Conference on Circuit/Systems, Computers and Communications (ITC-CSCC'97), pp.825-828 (Jul, 1997).

[10] 天野憲樹, 渡部卓雄: "動的適応可能なソフトウェア ・モデルのための言語的アプローチ", 日本ソフトウェア科学会 第 1 回プログラミングおよび応用のシステムに関するワークショップ (SPA'98) (Mar, 1998).

[11] 天野憲樹, 渡部卓雄: "自己反映アーキテクチャによる動的適応可能なソフトウェアのための言語", 日本ソフトウェア科学会 第 13 回全国大会, pp.441-444 (Sep, 1997).

[12] 天野憲樹, 渡部卓雄: "モーバイルコンピューティングのための動的適応可能なソフトウェアアーキテクチャ", 日本ソフトウェア科学会 第 13 回オブジェクト指向計算ワークショップ (WOOC'97) (Mar, 1997).

[13] 天野憲樹, 渡部卓雄: "移動計算機環境をサポートする言語 LEAD による非同期イベント処理", 情報処理学会 第 3 回モーバイルコンピューティング研究会 IPSJ mobile computing group 3th workshop, IPSJ SG Notes, pp.71-76 (Nov, 1996).

[14] 天野憲樹, 渡部卓雄: "メタレベルアーキテクチャによる移動計算機環境のための言語", 日本ソフトウェア科学会, pp.281-284 (Sep, 1996).

[15] 天野憲樹, 渡部卓雄: "メタレベルアーキテクチャによる移動計算機環境のための言語", 情報処理学会 第 1 回モーバイルコンピューティング研究会, pp.9-14 (Jul, 1996).

[16] 天野憲樹, 渡部卓雄: "移動計算機環境を支援するインタフェース記述言語", 情報処理学会 第 51 回全国大会, 分冊 5 pp.71-72 (Sep, 1995).