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

プ ロ グ ラ ム 説 明 書

平成１１年２月

天野 憲樹

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．

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．

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 0.1)．

表 0.1: 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 0.2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 0.2: 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 0.3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 0.3: 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 0.4 のとおりである．

表 0.4: 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．このクラスのオブジェクトは，仕様書中の適応可能メソッド選択機能および適応機構の拡張機能を実現する．

  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，論文 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 (論文 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（論文 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．このクラスのオブジェクトは，仕様書中の適応可能メソッド実行機能および適応機構の拡張機能を実現する．

  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（論文 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（論文 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．

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0.5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 0.5: 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．

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．

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 0.2 ）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 クラスライブラリのインストールと稼働環境の設定は容易である．しかし，以下の点について注意する必要がある．

- 適当なディレクトリに DAS クラスライブラリを展開する．
- `autoexec.bat` に展開したディレクトリへのパスを通す．

これらの操作の詳細については，ウインドウズ関連の書籍もしくは Java 関連の書籍を参照されたい．

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）．

DAS クラスライブラリを用いたプログラミング

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．

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．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0.6の仕様のメソッドdisplay を定義する[1]．

```
// 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．

[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．

表 0.6: 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

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．

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 0.4) 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．

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 0.7)．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 0.7: 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 0.8）．

表 0.8: 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()

表 0.9: 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)

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 0.9）．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．