アスペクト指向の分散化支援ツール

— SPA Summer 2002 —

西澤 無我　　千葉 滋

東京工業大学 大学院 情報理工学研究科 数理・計算科学専攻

Email: {muga,chiba}@csg.is.titech.ac.jp

要 旨

本論文では、分散ソフトウェアに見られる crosscutting concern について述べる。Crosscutting concern とは、アスペクト指向の用語で、既存のオブジェクト指向技術では、単一のモジュールに分離できず、いくつかのモジュールにまたがってしまう処理を意味する。我々は、現在、この crosscutting concern を独立したモジュールとして記述できるようにした、アスペクト指向の Java 言語用分散化支援ツール Jarcler を開発している。本論文では、Jarcler の設計の概要を示し、また汎用のアスペクト指向言語 AspectJ や既存の分散化支援ツールでは、この crosscutting concern をうまく扱うことがでいないことを述べる。

1　はじめに

今日、分散ソフトウェアの必要性が高まる一方、その開発にかかるコストが問題になっている。分散ソフトウェアの開発は、ネットワーク処理などの分散環境特有の問題に対処しなくてはならず、通常のソフトウェアに比べ、可読性が低く、修正・変更が困難になりがちである。これは、分散にかかわる処理とそれ以外の処理が、プログラム中で密接に入り混じるためである。このため、これら二つの処理をできるだけ分離して、異なるモジュールとして記述することが重要になってくる。

このような記述の分離を実現するために、これまで、いくつかの開発支援ツールが提案されてきた [6, 7]。これらのツールは、Java 言語を対象に、生成したオブジェクトをどのホスト上に配置するか、また、異なるホスト間での遠隔メソッド呼び出しをどのように処理するか、を他の処理から分離して記述することを可能にする。これらの処理は、本来、他の処理と密接に結びついており、Java 言語の機能の範囲内で分離することはできない。そこで、これらのツールは、オブジェクトの配置と遠隔メソッド呼び出しの実装方法の指定だけを他から分離してユーザに記述させる。それを元に分散にかかわる処理を実行するコードをツールが自動合成し、プログラム中の適切な位置に埋め込む。容易に分離可能な部分だけをユーザに分離して記述させ、そうでない部分はユーザには記述させず、自動合成することで、記述の分離を実現している。

しかしながら、このような手法で分離して記述できるのは、分散にかかわる処理の一部である。分離して記述すべきなのは、オブジェクトの配置や遠隔メソッド呼び出しの処理だけではない。アプリケーションによっては、オブジェクトの複製をいくつかのホスト上に配置し、複製間の一貫性 (同一性) を適切に維持しなければならない。このような複製の生成と一貫性の維持も、分散にかかわる処理として分離して記述されるべきである。しかし、多くの場合、必要とされる一貫性は、アプリケーション固有のロジックに特化した緩いものである。厳密な一貫性を維持するためのコードであれば自動合成が可能だが、アプリケーションのロジックに依存した緩い一貫性を維持するコードの自動合成は困難である。よって、従来の手法では分離して記述することができない。

我々は、アスペクト指向技術（AOP）により、分散にかかわる処理のうち、従来より広い範囲の処理を分離して記述できる Java 言語用開発支援ツール Jarcler を作成している。このツールでは、自動合成が困難な部分は、汎用的なアスペクト指向言語である AspectJ [2] のアスペクトに似た形でユーザに記述させ、最終的に自動合成したコードと共にプロ

グラム中に埋め込む。Jarcler のアスペクトは、文法こそ AspectJ のアスペクトに似ているが、記述の内容は Jarcler によって自動合成されるコードと密接な関係がある。このため、Jarcler のアスペクトは、AspectJ のコンパイラ (weaver) で処理することはできず、Jarcler 専用の処理系を新たに作成しなければならない。

Jarcler によって、より広い範囲の分散に関連する処理を別モジュールに分離できるようになる。その結果、ひとつのモジュールの中に分散に関連する処理とそうでない処理が同居し、密接に入り混じる状況を回避できる。このようなモジュール分割をすすめることにより、全体としてソフトウェアの可読性・保守性を改善することができる。

以下、第 2 章では、従来の分散化支援ツールについて概説し、その問題点を例示する。第 3 章では Jarcler の基本設計を示す。第 4 章で関連研究として AspectJ と Jarcler とを比較し、第 5 章で本論文をまとめる。

2　分散に関する処理の記述の分離

本章では、まず、従来の分散化支援ツールについて概説する。次に、簡単なネットワーク・ゲームの開発を例に、従来のツールの限界を示す。最後に、この限界を回避する方法として、reflection [4] を応用した方法を紹介し、その問題点を論じる。

2.1　既存の分散化支援ツール

Java 言語用の最も素朴な分散化支援ツールは、Java の標準 API に組み込まれている Java RMI [5] である。このツールは、遠隔メソッド呼び出しの処理を実行するコードを、ユーザからのコマンドに従って自動合成する。しかしながら、このツールを使うためには、異なるホスト上のオブジェクトは必ずインタフェース型を使って参照されなければならず、プログラムの記述に一定の制約がかかる。また、オブジェクトの配置はプログラム中に手で直接記述しなければならない。このように、Java RMI では、分散に関する処理を他から分離して記述できるとはいいがたい。

分散に関する処理のうち、オブジェクトの配置と遠隔メソッド呼び出しの実装方法を、分離して記述可能にするのが、Addistant [6, 3] や J-Orchestra [7] である。このようなツールを使うと、ユーザは単一のホスト (Java 仮想機械) 上で動くことを想定して書かれたプログラムと、各オブジェクトをどのホスト上に配置するかを指定する記述だけから、分散処理をおこなうソフトウェアを作成することができる。ここで、前者が分散とは無関係な処理の記述であり、後者が分散に関する処理の記述である。それぞれの処理は、互いに分離して記述できる。

このようなツールでは、ユーザは、分散に関する処理を実行するコードを直接記述しない。実際に動くコードは、ツールがユーザの記述にしたがって自動合成し、プログラム中に埋め込む。また、必要に応じて、ツールが元のプログラムを一部変更する。分散に関する処理を実行するコードは、本来、他のコードと密接に結合しているので、Java 言語の機能の範囲内では分離して記述できない。そこで簡単に分離できる、オブジェクトの配置と遠隔メソッド呼び出しの実装方法だけをユーザに記述させ、実際に動くコードはツールが自動合成する。これにより、結果的に、ユーザの目から見た記述の分離を実現している。

2.2　既存ツールの限界

Addistant のようなツールでは、記述の分離を自動合成によって実現している。このため、分散に関する処理として分離した記述が可能なのは、オブジェクトの配置等、関連するコードを自動合成可能な処理だけである。しかし、分散に関する処理は、それだけではない。

例として、複数のプレーヤーが、ネットワーク経由で互いに対戦できる○×ゲーム (Tic-Tac-Toe) の開発を考える。このゲームは、三つのホストに分散して動作する。二つのホストは各プレーヤーに対する GUI を処理し、残りのホストはゲーム全体の制御を処理する。

ゲームを実現する処理のうち、分散と無関係な処理とは何であろうか？まず無関係なのは、勝ち負けの判定など、ゲームのロジックを制御する処理である。では、二つのホスト上に別々のウィンドウを表示し、プレーヤーからの入出力を受け付ける処理はどうであろうか？ 我々は、そのような処理は分散に関連した処理であり、ゲームのロジックに関連した処理から分離して記述できるべきだと考える。○

×ゲームを単一ホスト上で実現するのに必要な処理だけが、ゲームのロジックに関連した処理であり、それ以外は分散に関連した処理として分離して記述できるべきだろう。

ここでいう、単一ホスト上で動作する○×ゲームとは、ウィンドウを画面に一つだけ表示し、二人のプレーヤーは、そのウィンドウを一緒に見ながら、マウスを交互に使ってゲームを進めるものである。具体的なプログラムは、おおよそつぎのようなものである。以下、説明のために代表的な部分を取り出し、簡略化したものを示す。

まず、○×ゲームのゲーム盤の各ます目を表すクラス Square を示す。

```
class Square {
  TicTacToe ttt;
  char mark;  // 'O', 'X', or ' '
  int pos;    // position

  Square(TicTacToe t, int p) {
    ttt = t; pos = p;
  }
  void setMark(char c) { mark = c; }
  char getMark() { return mark; }
  void pressed() { ttt.clicked(pos); }
}
```

TicTacToe は、勝敗判定など、ゲームの制御をおこなうオブジェクトである。GUI プログラムは、各ます目の getMark() を呼んで各ます目の状態を調べ、画面を表示する。また、ます目がプレーヤーによってクリックされると、GUI プログラムは、そのます目の pressed() を呼ぶ。pressed() は、TicTacToe の clicked() を呼ぶ。引数はクリックされたます目の位置である。clicked() は今、どちらの手番かを調べ、クリックされたます目の状態を setMark() を呼んで適切に変える。

一方、ゲーム盤を初期化するには、

```
TicTacToe t = ...;
for (int i = 0; i < 9; ++i)
  board[i] = new Square(t, i);
```

のような処理が実行される。単一ホスト版のゲームでは、表示されるウィンドウが一つなので、9 個の Square オブジェクトを生成する。

これら単一ホスト版のプログラムの記述とは別に、分散に関する記述をユーザが書き、それら二つをツールが処理して、一つのプログラムに統合し、ネットワーク版の○×ゲームが作成できればよい。分散に関する記述の中には、例えば次のような事柄が示される。

- TicTacToe オブジェクトは、ゲーム全体の制御を処理するホスト上に、Square オブジェクトは、各プレーヤーとの GUI を処理するホスト上に生成しなければならない。また Square オブジェクトは、各プレーヤーごとに 1 個ずつ、別々のホスト上に生成する。したがって全体では 9 組、18 個の Square オブジェクトを生成する。
- Square オブジェクトが、TicTacToe オブジェクトの clicked() を呼ぶときには、どちらのプレーヤーによるクリックかを表す引数を追加して渡さなければならない。もし同じプレーヤーが連続してクリックしている場合には、2 回目以降のクリックを無視する。

一組の Square オブジェクトは、互いに他方の複製になっているといえる。しかし複製間の一貫性は厳密に保たれているわけではなく、二つの複製オブジェクトを完全同一のものとして扱うことはできない。フィールドの値は常に複製間で同一でなければならないが、pressed() の動作は複製間で異なる。

実際には、これ以外にも多くの事柄を記述しなければならないが、先に示したプログラムに関係するのは上記の 2 点である。しかしながら、上記の 2 点ですら、オブジェクトの配置と遠隔メソッド呼び出しの実装方法以上の内容を含んでいる。これらは自動合成できないので、Addistant のようなツールでは、分離した記述が不可能である。

2.3 Reflective ミドルウェアによる対応

上で述べたような記述の分離を実現するために、これまでに reflection [4] 等を応用した拡張可能なミドルウェアが研究されてきた。従来の分散化支援ツールが自動合成したコードは、ネットワーク処理をおこなうため、ミドルウェア中の実行時ライブラリを呼び出す。そこで、この実行時ライブラリの一部を、ユーザが自由に差し替えて機能をカスタマイズできるようにする。実行時ライブラリは、元々アプリケーション・ソフトウェアからは独立しているので、実行時ライブラリのカスタマイズの形で、上で述べた分散に関する処理を記述できれば、記述の分離を実現できたことになる。

Reflection を利用した拡張可能なミドルウェアでは、自動合成されたコード中にフック (hook) を挿入し、実行時ライブラリの呼び出しを実行時に横取りして、ユーザが提供する別なライブラリを呼び出すように変えることができる。横取りしたメソッド呼び出しを受け取るオブジェクトのことを、メタオブジェクトという。ユーザは、このメタオブジェクトを独自に定義し、メソッド呼出しの動作を変えることができる。同様の機構で、オブジェクト生成の動作を変えることができる。一般に、メタオブジェクトのクラスは、ミドルウェアが提供するクラス Metaobject のサブクラスとして定義される。クラス Metaobject は、オブジェクト生成やメソッド呼出しの標準的な動作を実現するメタオブジェクトである。

メタオブジェクトの利用により、○×ゲームの分散に関連する処理を記述・実現する方法を以下に示す。まず Square オブジェクトの生成である。分散化支援ツールは、標準では、new Square() という式を書き換えて、他のホスト player1 と通信し、そのホスト上で Square オブジェクトを生成するようにする。Square オブジェクトは、ホスト player1 だけでなく、別のホスト player2 上でも生成されなければならない。そこで、メタオブジェクトを使って実行時ライブラリの呼び出しを横取りし、次のような処理をおこなう。

```
Object trapCreation(Object[] args) {
  createRemoteObject("player2",
                     "Square", args);
  return super.trapCreation();
}
```

ここで args は、Square のコンストラクタに与えられた引数列である。createRemoteObject() で、ホスト player2 上でもオブジェクトを生成させ、その後、親クラスの trapCreation() を呼び出すことで、標準のオブジェクト生成処理を実行する。この場合は、ホスト player1 でオブジェクトを生成させる。

同様に、TicTacToe の clicked() にどちらのプレイヤーのクリックであるかを伝えるためには、clicked() が呼び出される直前に、TicTacToe オブジェクトが存在するホスト上でメタオブジェクトを使って、これを横取りする。横取りしたメタオブジェクトは、次のような処理を実行し、有効なクリックであるか否かを検査する。

```
Object trapMethodcall(Method m,
                      Object[] args) {
  if (m.getName().equals("clicked"))
    if (getCallerHost() != nextPlayer)
      return null;  // クリックは無視

  return super.trapMethodcall(args);
}
```

ここで m は呼ばれたメソッドを、args は引数列を表す。このように reflection を利用して拡張可能なミドルウェアを提供すれば、分散に関連する処理を、他から分離したモジュールとして記述することができる。

しかしながら、最近多くの研究者が指摘しているように、このようなメタな視点に立った記述は、一般のソフトウェア開発者にとって必ずしも直感的でも簡潔でもない。メタな視点に立っているので、オブジェクトの生成やメソッド呼び出しは、通常の文法ではなく、全てメタオブジェクトのメソッドを呼び出すことで実行しなければならない。このため、開発者はメタオブジェクトが提供する API を熟知していなければならない。

また分散処理の場合はほとんど問題にならないとはいえ、reflection は各種データの reify 処理をおこなうため、実行時オーバヘッドを伴う。上の例では、呼び出されたメソッドを表す Method オブジェクトを生成したり、引数列を Object 配列型に変換することが、reify 処理に相当する。

3 Jarcler の設計

我々は、前章で述べた問題をアスペクト指向技術で解決する分散処理支援ツールを開発している。本章では、この支援ツール Jarcler について述べる。

3.1 Crosscutting concern

前章で示した Square オブジェクトの組は、アプリケーションのロジックに依存して緩く一貫性が維持されるようなオブジェクトの複製であった。この一貫性を処理するコードは、アプリケーションのロジックを記述したモジュールと、支援ツールが自動合成する、分散に関連するコードを集めたモジュール (ミドルウェア) の両方にまたがってしまう (図 1)。すなわち、アスペクト指向でいう crosscutting concern

の一種である。

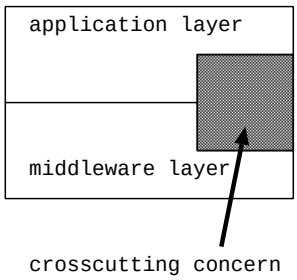

図 1: 複製の一貫性維持

Reflective ミドルウェアによる解決は、メタな視点からの記述を可能にすることで、複製間の一貫性処理をミドルウェアの中だけで記述できるようにするものと考えられる (図 2)。その結果、記述がアプリケーションのロジックを記述したモジュールにまたがることがなくなり、crosscutting concern を解消できる。

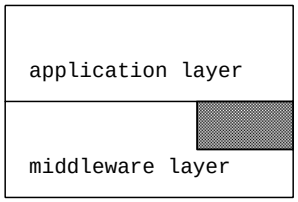

図 2: Reflective ミドルウェアでの方法

Reflective ミドルウェアによる解決には、先に述べたような問題がある。そこで、我々はアスペクト指向技術を導入することで、この crosscutting concern を分離して記述できるようにする。我々のツール Jarcler では、AspectJ に似た構文で、この crosscutting concern をアスペクトとして記述できる (図 3)。Jarcler のアスペクトは、コンパイル時に Jarcler が自動合成したコードと共に、他のプログラム中に埋め込まれる (weave される)。Jarcler は、コードの自動合成および weave 処理を、Javassist [1] を使ってバイトコード変換で実現する。

ここで取上げた crosscutting concern の特徴は、自動合成されたコード、つまりミドルウェア、と密接な結びつきをもつ点である。このため、AspectJ [2] のような、Java 言語用の汎用 weaver を利用して実装することができない。

AspectJ が扱える crosscutting concern は、ユーザが Java 言語で記述したクラス間にまたがるものである (図 4)。AspectJ では、crosscutting concern を処理するコードはアスペクトと呼ばれる。アスペクトは、コンパイル (weave) 時に、通常のクラスのメソッドの中に挿入されるが、具体的にどの部分に挿入されるかは、ユーザが明示的に指定しなければならない。アスペクトを挿入可能な位置の集合を join points といい、個々のアスペクト (正確にはアドバイス) が実際に挿入される位置のことを point cut という。Point cut は join points の部分集合となる。

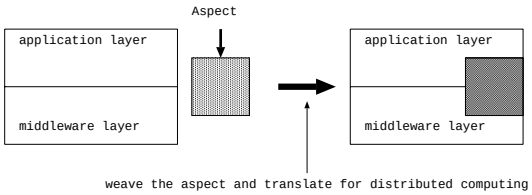

図 3: Jarcler での方法

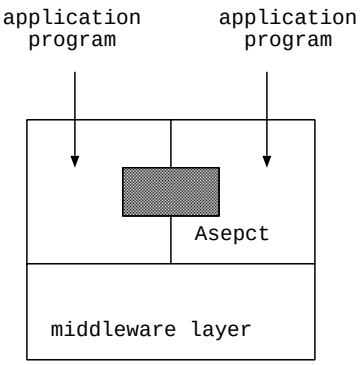

図 4: AspectJ が扱う crosscutting concerns

本論文で取上げた crosscutting concern は、ユーザが記述したクラスと支援ツールが自動合成したクラスの間にまたがるものである。自動合成されるクラスの名前や、そのクラスに定義されるメソッドの一覧等は、支援ツールの実装依存で、ユーザから隠されている。クラス名やメソッド名が不明であるため、ユーザは point cut を指定できず、したがって AspectJ ではその crosscutting concern をうまく扱えない。

支援ツールが自動合成するクラスの仕様をユーザに公開すれば、ユーザは point cut を指定できるようになり、原理的に AspectJ を使って、本論文で取

上げた crosscutting concern を扱えるようになる。しかしながら、自動合成されるクラスは一般に複雑で、ユーザにとってけして直感的でもわかりやすいものでもない。この方法では、Reflective ミドルウェアと同様な問題をかかえてしまう。また、自動合成するクラスの仕様を公開すると、支援ツールの実装者は、その仕様に常にしたがわなければならず、実行効率を改善する工夫などがおこないにくくなる。

3.2 アスペクトによる対戦型 TicTacToe の記述

図 5 に Jarcler を利用して、ネットワーク対戦型○×ゲームを開発するのに必要なアスペクトの記述を示す。構文は、おおよそ AspectJ の拡張になっている。まず、remote から始まる 2 行で、TicTacToe オブジェクトは server ホスト上で、Square オブジェクトは遠隔ホスト上で生成されることを宣言する。次に use から始まる行で、TicTacToe クラス内部で使われている Square クラスは全て SquarePair クラスに置換することを宣言する。SquarePair は、Square オブジェクトの組を表すクラスである。これによって、Square オブジェクトを生成するコードは、次のように書き変えられる。

```
TicTacToe t = ...;
for (int i = 0; i < 9; ++i)
    board[i] = new SquarePair(t, i);
```

SquarePair クラスにより、Square オブジェクトは、各プレーヤーごとに 1 個ずつ別々のホスト上に生成するという処理が実現される。SquarePair クラスの宣言は、図 5 に記述されているとおりである。コンストラクタの中で、Square オブジェクトを 2 個生成していることがわかる。Square オブジェクトは遠隔ホスト上で生成されるので、コンストラクタは Jarcler によって拡張されており、第一引数は、オブジェクトを生成するホスト名である。

次の

```
static char Square.myMark;
```

は、Square クラスに static フィールド myMark を追加することを指示する (introduction)。このフィールドは、各ホストを利用しているプレーヤーが、○と×のどちらかであるかを示す。その次の on 以下は、ホスト player1 と player2 上で、初期化時に

```
public aspect RemoteGUI {
  remote class TicTacToe at "server";
  remote class Square;

  use SquarePair
    insteadof Square in TicTacToe;

  class SquarePair {
    Square s1;
    Square s2;

    SquarePair(TicTacToe ttt, int p) {
      s1 = new Square("player1", ttt, p);
      s2 = new Square("palyer2", ttt, p);
    }

    void setMark(char c) {
      s1.setMark(c);
      s2.setMark(c);
    }
        :
  }

  static char Square.myMark;

  on "player1" {
      Square.myMark = 'O';
  }

  on "player2" {
      Square.myMark = 'X';
  }

  void around(TicTacToe t, int pos):
      target(t) && args(pos)
      && within(Square) && callerside
      && call(void clicked(int)) {
    t.clicked(pos, Square.myMark);
  }

  void TicTacToe.clicked(int pos,
                         char player) {
    if (player != nextPlayer)
      return null;

    clicked(pos);
  }
      :
}
```

図 5: Jarcler による記述例

実行する処理を表す。Square クラスは、各ホスト上の Java 仮想機械 (JVM) に別々にロードされるので、static フィールドの値は、各ホストによって変えることができることに注意されたい。

次の around 以下と clicked() の定義は、自分の手番でないプレーヤーがクリックしたときに、それを無視するための処理の記述である。まず around アドバイスで、Square クラスの元の pressed() メソッドを変更する。元のメソッドは引数 pos で clicked() を呼び出していたが、引数に myMark も渡すように変更する。around に続いて callerside とあるのは、このアドバイスを TicTacToe オブジェクトが存在するホストではなく、呼び出し元のホストで実行することを指示している。callerside は AspectJ には存在しない機能である。

around アドバイスの下の記述は、TicTacToe クラスに int と char の二つの引数をとる clicked() メソッドを追加するためのものである。このメソッドは、現在、クリックしたプレーヤーの手番であるかどうかを調べ、手番であるときだけ元からある clicked() を呼んで処理をおこなう。

4 まとめ

本論文では、我々が現在開発中のアスペクト指向の Java 言語用分散化支援ツール Jarcler について述べた。このツールを用いると、オブジェクトの分散配置や遠隔メソッド呼び出しだけでなく、緩い一貫性を維持するオブジェクトの複製を実装するためのプログラムも、ユーザによる簡単な指示で自動合成させることができる。多くのアプリケーション・ソフトウェアでは、厳密な一貫性を保った複製ではなく、アプリケーション・ロジックに強く依存した緩い一貫性を保つ複製が必要である。このような複製の実装は従来の技術では困難であったが、我々はアスペクト指向技術を導入することにより、これを可能にした。

Java 言語用の汎用アスペクト指向言語として AspectJ が有名だが、AspectJ と従来の分散化支援ツールを組み合わせて使うだけでは、Jarcler と同等の機能は得られない。Jarcler で扱う crosscutting concern は、ユーザが書いたクラスと、支援ツールが自動合成するクラスにまたがるものである。AspectJ は、ユーザが書いた異なるクラスにまたがる corss-cutting concern を扱う言語であるため、Jarcler では専用の言語処理系を用意しなければならなかった。

参考文献

[1] S. Chiba, Load-time Structural Reflection in Java, In *Proccedings of ECOOP 2000, LNCS 1850*, Springer Verlag, pp.313–336, 2000.

[2] G. Kiczales, J. Lamping,C. Maeda, Aspect-Oriented Programming, In *Proceedings of ECOOP 1997, LNCS 1241*, June, 1997.

[3] M.Nishizawa, M.Tatsubori, S.Chiba, プログラム分散化のためのアスペクト指向言語, In *Proceedings of SPA 2002*, March, 2002.

[4] B.C.Smith, Reflection and Semantics in Lisp, In *Proceedings of ACM Symp. on Principles of Programming Languages*, pp.23–35, 1984.

[5] The Java Remote Method Invocation Specification, Sun Microsystems, Inc., http://java.sun.com/products/jdk/rmi/, 1997.

[6] M. Tatsubori,T. Sasaki,S. Chiba, and K. Itano, A Bytecode Translator for Distributed Execution of "Legacy" Java Software, In *Proceedings of ECOOP 2001, LNCS 2072*, Springer, pp.313–336, 2001.

[7] E. Tilevich, Y. Smaragdakis, J-Orchestra: Automatic Java Application Partitioning, In *Proceedings of ECOOP 2002 — Object-Oriented Programming, LNCS 2374*, pp.178–204, 2002.