修士学位論文

題目

プログラムスライスを用いた
アスペクト指向プログラムのデバッグ支援環境

指導教官
井上克郎 教授

報告者
石尾 隆

平成 15 年 2 月 12 日

大阪大学 大学院基礎工学研究科
情報数理系専攻 ソフトウェア科学分野

平成 14 年度 修士学位論文

プログラムスライスを用いた
アスペクト指向プログラムのデバッグ支援環境

石尾 隆

内容梗概

アスペクト指向プログラミングは，複数のオブジェクトに関連した処理をオブジェクトから分離し，新しいモジュール単位「アスペクト」として記述することで，保守性や再利用性を向上させる．

アスペクトの有用性については数多く報告されているが，同時に，アスペクトは新たな複雑さをプログラムに導入している．アスペクトはオブジェクトとは独立して記述されるため，開発者がオブジェクトのすべての振る舞いを把握することが難しい．また，複数のアスペクトが相互に干渉して動作を阻害するなど，原因の特定が難しい欠陥を作りこむ恐れもある．

このような問題に対して，本研究では，コールグラフによるアスペクト干渉の検出と，プログラムスライシングによるデバッグ支援を提案する．コールグラフは，従来から用いられる手続き単位での呼び出し関係をグラフ化したものである．オブジェクトとアスペクトの制御関係から，不用意なアスペクトの動作，無限ループに陥る可能性などを指摘する．また，プログラムスライシングは，プログラムの依存関係を解析することで開発者が扱う必要があるコードを抽出，提示する手法である．オブジェクト指向プログラムに対して提案されている手法にアスペクトの動作に関する制御情報を加えて拡張し，適用を行う．

本研究では，アスペクト指向言語 AspectJ を対象に，コールグラフ計算とプログラムスライス計算ツールを統合開発環境に組み込む形式で実装し，適用実験を行った．その結果として，アスペクト指向プログラムにおいてアスペクトが与える影響を効果的に開発者に提示できることが示された．

主な用語

アスペクト指向プログラミング
デバッグ
ソフトウェア保守
プログラムスライシング

目 次

1　まえがき

近年，プログラムの新しいモジュール化手法としてアスペクト指向プログラミングが提案され，利用されるようになってきている [1]．従来のオブジェクト指向プログラミングにおいて，ロギングや同期処理のような複数のオブジェクトが関わる処理のことを横断要素と呼ぶ．横断要素は単一のモジュールに記述することはできず，その処理に参加するすべてのオブジェクトにコードが分散するため，保守性を悪化させる要因となっていた．アスペクト指向プログラミングは，一つの横断要素を単一のモジュールに記述するための新しいモジュール単位「アスペクト」を導入している．

アスペクトは，プログラム中の実行時点 (join point) に連動して動作する処理として記述される．実行時点とは，オブジェクト間でやり取りされるメッセージの送受信のタイミングなどを指す．従来のプログラミングが，どのオブジェクトが何をするかという手順を書き下していたのに対して，アスペクトはどのような条件が成立したときに処理を行うか，という単位で処理を記述する．アスペクトの動作条件の記述はオブジェクトの枠にとらわれることがないため，横断要素をオブジェクトから分離し，単独のモジュールとして記述することができる．横断要素をアスペクトとしてモジュール化することで，再利用性および保守性の向上につながる．

アスペクトの応用事例は数多く報告されている．オブジェクト指向プログラミングで用いられているデザインパターン [14] は，複数のオブジェクトがどのように連携するかを説明した設計部品である．これは，オブジェクトの横断要素の一種であると考えられるため，アスペクトとして記述することで，パターン自身が再利用可能なソフトウェア部品となる場合があることが知られている [3]．また，ソフトウェア開発時のデバッグ支援や，分散アプリケーションでのオブジェクトを横断した性質の記述など，様々な場面でその有用性が示されつつある [2, 4]．

しかし，アスペクトの導入が，プログラムに新しい複雑さをもたらすことも指摘されている．アスペクトは，オブジェクトの外部からその振る舞いを変えることができるため，開発者はオブジェクトとして記述されたコードだけではなく，関連するすべてのアスペクトを調べなければ，そのオブジェクトの全ての振る舞いを把握することはできないという問題がある．また，アスペクトの干渉問題と呼ばれる問題も発生している．これは，複数のアスペクトが相互に干渉を与え，単体ではそれぞれ正しいはずのアスペクトが，同時に使うときには正しく動作しなくなるという問題である [5]．また，アスペクトはオブジェクトの振る舞いに連動するという性質のため，プログラマの予期しない時点でアスペクトが動作するといった，発見が困難な欠陥を作り込む可能性もある．

このような問題に対して，アスペクトの予期せぬ動作可能性や，実際に動作したことに

よって発生した障害の原因調査を支援するツールの必要性が指摘されている．

本研究では，アスペクト指向プログラムの開発を支援する方法として，コールグラフによる呼び出し関係の提示と，プログラムスライシング技術 [8] を用いたデバッグ支援を行う．

コールグラフとは，メソッドを頂点とし，呼び出し関係を有向辺とするグラフである．これにアスペクトの頂点と辺を加えて，予期せぬアスペクトの動作や無限ループ発生可能性などの提示を行う．これによって，開発者が制御に関連した欠陥を検出しやすくすると考える．アスペクトの記述を誤ると，アスペクト間の相互依存などから無限ループに陥りやすく，事前の検出を行うことは開発効率を高めるために有効である．

プログラムスライシングとは，プログラム内部の依存関係を解析することで，プログラマが注目すべきコードを提示する技術である．アスペクト干渉など，アスペクトがもたらす複雑さは，アスペクトの相互依存関係によって発生するため，依存関係を利用するプログラムスライシング，特に実行時情報を用いた DC スライス計算 [10, 12] が有効であると期待できる．スライス計算ではコールグラフよりも詳細なプログラム依存グラフを用いるが，オブジェクト指向プログラミングで用いていたプログラム依存グラフに，アスペクトを表現した頂点と辺を加えることによってスライスを実現する．

DC スライスを計算するためにはプログラムの実行時情報が必要となるが，どのようにプログラムの実行を解析するかが問題となる．本研究では，この動的解析自体もアスペクトとして記述し，対象プログラムに追加する方法を選択している．

本研究では，統合開発環境 Eclipse [20] をベースに，コールグラフ，動的解析アスペクトを用いた DC スライスツールの実装を行い，いくつかのアスペクト指向プログラムに対して適用実験を行った．その結果，プログラムスライシングがアスペクト指向プログラミングのデバッグ支援に適していることを示した．

以降，2. ではアスペクト指向プログラミングの特徴と問題点について説明する．3. ではコールグラフを用いた，アスペクトを組み込む際の問題検出について説明する．4. ではプログラムスライシングの概要とアスペクト指向プログラミングへの拡張について説明し，5. では実行時情報を収集するための方法について説明する．6. でアスペクト指向プログラムに対するプログラムスライシングツールの実装と評価について説明する．最後に 7. でまとめと今後の課題を述べる．

2　アスペクト指向プログラミング

2.1　アスペクト指向の特徴

アスペクト指向プログラミングは，オブジェクト指向プログラミングを基に，その弱点を補うプログラミング手法である．オブジェクト指向プログラミングでは，オブジェクトの相互通信としてシステムをモデル化する．オブジェクトはそれぞれ特定の機能を担当し，その機能の実現に必要なデータを内部に保持する．オブジェクトは他のオブジェクトと相互にメッセージ通信を行い，全体としてひとつのシステムを実現する．

しかし，オブジェクトという単位でシステムの機能を分担する都合上，担当するオブジェクトを一つに決められないような特性はうまく扱えない．たとえばシステムの動作を記録していくロギング，エラーが起こったときの例外処理，データベースなどにおけるトランザクションの手続きは，システム内の複数のオブジェクトが連携して実現することが多い．このような処理は横断要素と呼ばれており，横断要素に関わる複数のオブジェクトにコードが分散するという問題がある．コードの分散は，次のような事態を引き起こす．

- 横断要素の仕様が変わると，すべての関連したコードを変更しなければならない．そのためには，横断要素に関連したコードを正しく識別できる必要がある．
- 横断要素を含んだオブジェクトを再利用しようとすると，その横断要素に関連したコードを取り除く，あるいは，関連したオブジェクト群をまとめて再利用するかのどちらかとなる．横断要素の除去には，再利用しようとするオブジェクトに関する十分な知識が必要である．オブジェクト群をまとめて再利用する場合は，不要なオブジェクトや横断要素を引き継ぐことになり，システムを肥大化させる．
- 横断要素だけを再利用することはできない．もし別の場所で同じ横断要素が必要であれば，再度実装しなければならない．

これに対して，アスペクト指向プログラミングは，横断要素を分離・記述するためのモジュール単位「アスペクト」を導入する．

アスペクト指向プログラミングでは，横断要素を捉えるために，オブジェクト指向プログラム内に含まれる実行時点を結合基準 (join points) と呼ぶ．その中から，何らかの処理を行いたい結合基準の部分集合を選択し，処理を関連付ける．結合基準に関連付けられる手続きのことをアドバイス (advice) と呼ぶ．アスペクトは，結合基準とアドバイスを記述することによって構成される．

使用できる結合基準は言語処理系によって異なっているが，一般的に使用されるものを次に示す．

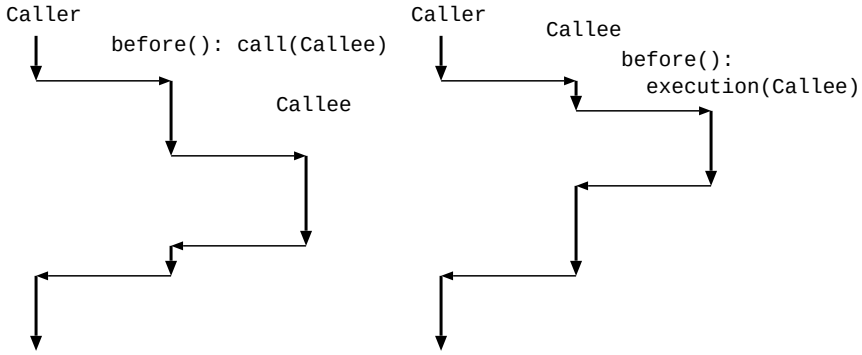

図 1: アドバイスの動作シーケンス例（左: 呼び出し前，右: 実行前）

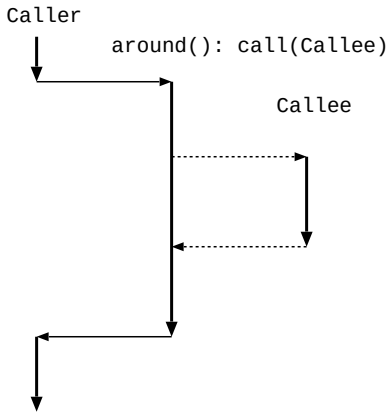

図 2: アドバイスの動作シーケンス例 (メソッド呼び出しを置換する例)

- オブジェクトに対するメソッド呼び出し.
- オブジェクトのメソッドの実行（動的束縛の解決後）.
- オブジェクトの持つフィールドへのアクセス.
- オブジェクトでの例外の発生.

このような結合基準に対して，その直前，直後に処理を挿みこむ，あるいは結合基準となっている処理を置き換える形でアドバイスは動作する．メソッド呼び出しおよびメソッド実行の直前に動作するアドバイスの実行シーケンスを図 1 に示す．また，メソッド呼び出しを置き換えるアドバイスの実行シーケンスを図 2 に示す.

アドバイスは，実行される条件が呼び出される側に書かれた，特殊な手続きとして考えることができる．アスペクトの言語処理系は，結合基準ごとに，そこで動作するアドバイスの

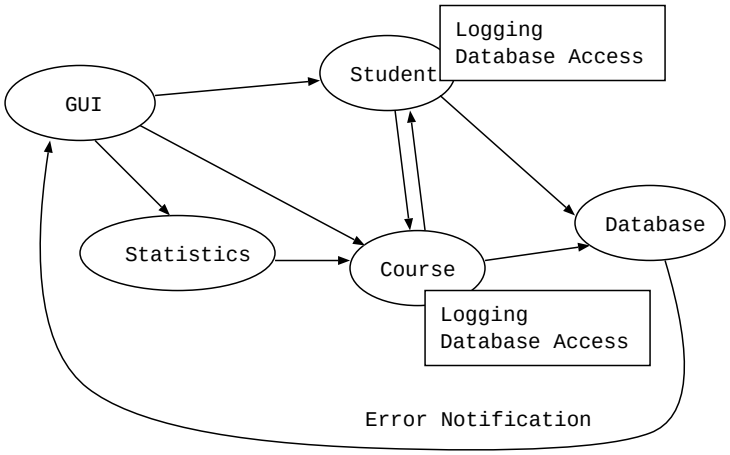

図 3: アスペクト導入前のプログラム

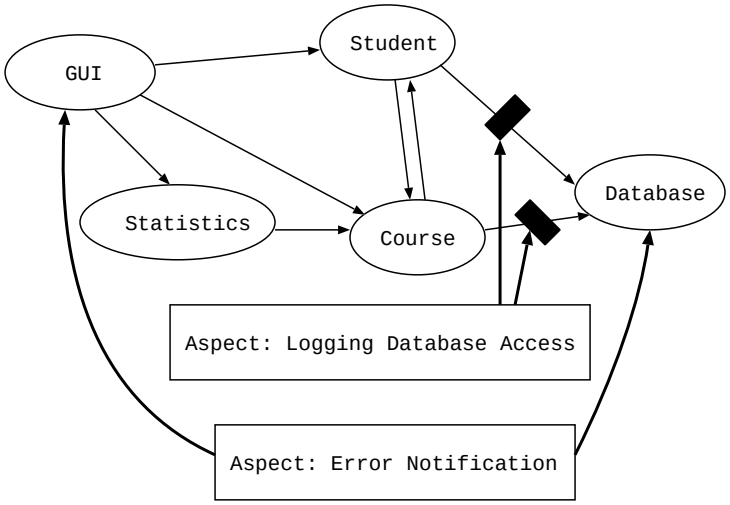

図 4: アスペクト導入後のプログラム

呼び出しを生成する．これによって，オブジェクトが意識せずにアスペクトを動作させることができるようになるため，本来は複数のオブジェクトに分散していたコードを単一のアスペクトにまとめることが容易となっている．

アスペクトが横断要素を分離する例として，学生の成績管理プログラムのオブジェクトモデルを図 3 に示す．この図では，楕円形の頂点がクラスを，有向辺がメッセージを送ることを表している．このプログラムでは，データベースへのアクセスを記録し，また，データベースアクセス中にエラーが起きた場合はユーザの指示を仰ぐために，GUI へとメッセージを送っている．エラー通知処理は Database と GUI を横断した処理であり，アクセス記

録は Student と Course，Database を横断した処理である．これをアスペクト指向で書き直すと，モデルは図 4 のようになる．このアスペクトは，Database クラスへのメッセージ送信に連動してアクセスを記録し，Database アクセス中にエラーが発生した場合は，エラーを GUI 経由でユーザに通知し，ユーザからの指示を Database に伝えることが記述されている．オブジェクト指向の場合と違い，Database から GUI への依存関係が取り除かれている．GUI，Database オブジェクトの独立性が向上したことによって，それぞれ独立した変更，再利用が容易なものとなっている．また，このアスペクトだけを独立して再利用することが可能である．

アスペクトの利点は，コードの分散によって引き起こされる事態が解消されることであり，次のようにまとめられる．

- 横断要素の仕様が変更されても，それに関連した全てのコードは単一のアスペクトに記述されているため，変更漏れがなく，変更の波及も認識しやすい．
- オブジェクトには横断要素のコードが含まれないため，オブジェクトの再利用が容易となる．
- アスペクトは，その結合基準だけを変更することで，他の場所でも再利用することができる．言語処理系によっては，これをオブジェクト指向言語における継承と同等のメカニズムとして実現している場合もある．
- アスペクトが追加されても，オブジェクト側のコードは変更しなくてよい．オブジェクトの動作に対してどのアスペクトが連動するかを決めるのはアスペクトとその言語処理系の仕事である．

その他の特徴として，アスペクトが結合基準に結合されるという特性上，言語処理系によっても異なるが，いくつかの特殊な情報にアクセスできる権限を与えられている場合がある．たとえば，現在実行されようとしている結合基準がどこにあるのか，メッセージに連動する場合はそのメッセージが何であるか，送り主あるいは送り先のオブジェクトがどれであるか，といった実行コンテキストに関する情報である．これらを用いることで，横断要素のうち，実行コンテキストに依存した処理も容易に記述することができるようになる．

2.2 アスペクトの用途

アスペクトの用途については，これまでに多くの研究が行われている．たとえば，従来，オブジェクト指向プログラミングで提案されていたデザインパターンと呼ばれるオブジェクト間の連携の仕組みがあるが，そのうちのいくつかはアスペクトを用いてより簡潔な形で書

```
class SomeClass {

  public void foo(int x) {
    doSomething(x);
  }
  private void doSomething(int x) {
    :
  }
}

aspect LoggingAspect {
  before(): call(void SomeClass.doSomething(..)) {
    Logger.logs(thisJoinPoint);
  }
}

aspect ParameterValidationAspect {
  before(int x):
    args(x) && call(void *.doSomething(..)) {
    if ((x < 0)||(x > Constants.X_MAX_FOR_SOMETHING)) {
      throw new RuntimeException("invalid parameter!");
    }
  }
}
```

図 5: アスペクトのコード例

き換えることが判明している [3, 14]．その他にも，事前・事後条件の強制，分散オブジェクトの記述，プログラム実行の解析など，着実に応用分野が広がりつつある．

アスペクトの用途は，主として次の三つに分類される [6]．

開発途中に，開発作業を支援する目的で導入するアスペクト: 実行速度などの性能計測，プログラム解析のためのメソッド呼び出し関係の記録などに利用され，最終的な製品出荷時には取り除かれる．

プログラムの機能を実現するためのアスペクト: プログラムが正常に動作するために必要なアスペクトである．オブジェクトの事前・事後条件の強制，システム内での統一的な例外処理の実装，トランザクションの実行などである．

性能改善のためのアスペクト: 実行速度の向上やメモリ節約などを目的とする．データの先読みやキャッシュなどがある．

アスペクトのサンプルコードを図 5 に示す．LoggingAspect，ParameterValidationAspect という二つのアスペクトが定義されている．LoggingAspect は SomeClass.doSomething というメソッド呼び出しが行われたことを記録するアスペクトで，ParameterValidationAspect

は，どのクラスであれ doSomething という名前のメソッドに対する呼び出しの引数が範囲内にあるかどうかを検査し，範囲外であれば例外を発生させるアスペクトである．

これらのアスペクトは SomeClass.doSomething への呼び出しでは両方ともが動作するが，相互に独立しているため，言語処理系が任意の順番で実行する．AspectJ においては，順番が重要な場合にはアスペクトの動作優先度を定義できるが，通常の場合は何も定義せず，言語処理系に任せることになる．

2.3 アスペクトのもたらす複雑さ

アスペクトの利便性については広く認められるようになってきたが，次のような新しい複雑さが問題となっている．

1. 同一の条件下で動作する複数のアスペクトが存在しうる．アスペクトの動作順序によって，結果が異なる場合がある．

2. あるアスペクトの動作中に，他のアスペクトの動作条件が成立する場合がある．動作条件によっては，二つのアスペクトの動作が相互に条件を満たしてしまい，無限ループを生じることもある．

3. 動作条件の記述を誤ると，予期せぬアスペクトの動作を招くことがある．ソフトウェアが拡張された際に，それまで使用していた条件の完全性，正確性が損なわれることがある．

先に挙げた三つのうち，1. と 2. はアスペクトの干渉と呼ばれる問題である．アスペクトが単体では機能するが，複数を合わせた場合に動作しなくなることもあるため，干渉を検出する，あるいは予防するための研究が数多く成されている [5].

また，3. はオブジェクトやアスペクトを変更する際に起きる問題である．他のアスペクトが動作する条件を把握した状態でのプログラミングが必要となっており，統合開発環境などでサポートを試みる動きがあるが，複数のアスペクトが干渉している場合への対処は不十分なのが現状である．

アスペクトの干渉の例を図 6 に示す．メソッド呼び出しをファイルに記録するアスペクトと，ファイル I/O をサーバへのネットワークアクセスに変換するアスペクトがあり，これらは独立では正しく動作する．しかし，これらを同時に動かそうとすると，次のようになる．

1. Client がファイルを利用しようとして，アスペクトが起動される．アスペクトは，その呼び出しをネットワークアクセスに変換する．

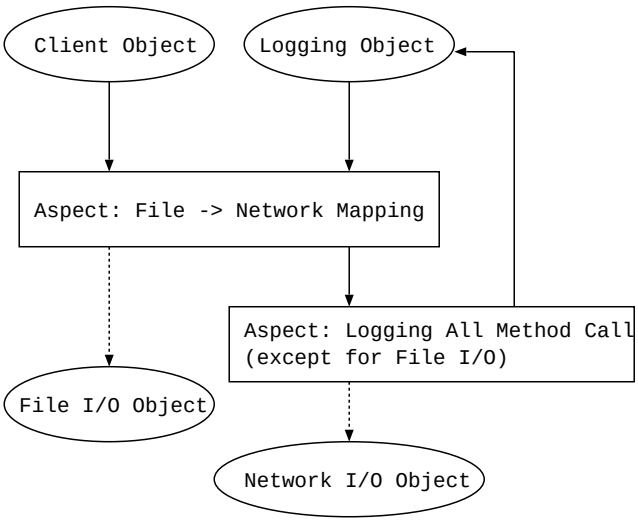

図 6: アスペクト干渉の例

2. ネットワークアクセスは，呼び出し記録アスペクトの監視対象なので，アスペクトへ制御が移る．アスペクトは，Logging オブジェクトを呼び出してデータを出力しようとする．

3. Logging オブジェクトのファイルへの出力も，ネットワークアクセスに変換される．

4. 再び，ネットワークアクセスを記録しようとして，アスペクトが動作する．以降，アスペクトが相互に起動され続ける．

このような，アスペクトの干渉や誤作動によって生じる障害の原因を見つけ出すことは非常に難しい．特に，アスペクト干渉は必要な場合もあるため，一概に禁止するというわけにはいかない．たとえば，ファイル入出力をネットワークアクセスに変換するアスペクトが，もし他のアスペクトへ干渉できないとすると，他のアスペクトが勝手にファイルを書き出す処理を実行する可能性があるため，本来の「横断要素の分離」としての目的を果たせないことになる．

これに対して，アスペクトの干渉を起こさないようにアスペクトの種類を制限する研究もなされている [7] が，本研究では，実際に干渉が起こることで動作不良に陥った場合に，その原因を特定する作業を支援することを考える．

このような問題の多くは，アスペクトが他のアスペクトに対して直接，あるいは間接的に連動し，影響を与えるために生じると考えられるため，アスペクトをプログラムに組み込む時点でのコールグラフによる静的解析，プログラムに組み込んだ後の欠陥除去としてのプログラムスライシングを併用して用いることで，効率よく欠陥を検出することができると考えられる．

3　コールグラフによるアスペクトの干渉検出

アスペクト指向プログラミングでは，不用意な結合基準の指定，アスペクト間の干渉などによって誤動作が生じる可能性がある．この原因の多くは，予期せぬ時点でアドバイスが動作することに起因している．

特に，アスペクトによって無限ループを生じる場合があるため，実行時にデバッガ等で検出するのではなく，プログラムをコンパイルした段階で，その可能性を検出できることが望ましい．

無限ループが発生する可能性を最も簡単に検出するものとして，メソッド間の呼び出し関係を表現したコールグラフ (Call Graph) がある．メソッドを頂点とし，呼び出し関係を有向辺としてグラフを作成し，ある頂点 v から出発して v に到達可能な経路があるとき，無限ループの可能性がある，ということになる．

コールグラフの利点と欠点を次に示す．

利点

- 構築コストが安価である．メソッド呼び出し関係は，各メソッドに含まれているメソッド呼び出し文を収集することによって構築されるが，これは，コンパイラが収集している情報と等価であり，拡張機構を持つコンパイラに動作を付加する形で容易に実装できる．
- 高い完全性を持つ．Java を含め，いくつかのオブジェクト指向言語では，文字列データなどからメソッド呼び出しを行うリフレクション機能を持つため，静的な解析では対応できない．しかし，それ以外の呼び出し関係はすべて抽出可能であるため，得られる結果の完全性は高い．

欠点

- 再帰呼び出しと無限ループの区別が付かないため，正確性が低くなることがある．

コールグラフをアスペクト指向に対して拡張するには，アスペクトの動作をどのようにグラフ上に取り込むかということが問題となるが，ここでは，図 1 および図 2 に示しているようにその動作を手続き呼び出しの一種として考えて，コールグラフを拡張する．

メソッド，アドバイスを頂点とし，メソッド呼び出し，アドバイスの起動を辺としたコールグラフを構築する．ある頂点からアドバイスの頂点へ到達できる，ということはその頂点がそのアドバイスを作動させる可能性がある，つまりアドバイスによる影響を受けている，と考えることができる．コールグラフ上でのアドバイスの頂点間の関係は，次のようにまとめられる．

- あるアドバイス adv_1 の頂点 v_{adv1} から他のアドバイス adv_2 の頂点 v_{adv2} に到達可能であるとき，adv_1 は adv_2 に影響を受けている．

- あるアドバイス adv_1 の頂点 v_{adv1} から，v_{adv1} 自身へ到達可能であるとき，adv_1 は無限ループに陥る可能性がある．

コールグラフは，ソースコードを解析することで容易に計算が可能である．他のアスペクトの干渉検出法として，形式的仕様記述を用いた手法などが考えられるが，それらに比べて実装が容易であり，開発者にとっても直観的な情報が提示できる点が特徴である．

一部の言語処理系では，アスペクトの動作順序や相互依存関係を開発者が細かく指定することができるもの [5] や，そもそも干渉を起こさないためにある種類に限定したアスペクトの利用を行うという試みもある [7]．そのような言語処理系でも，コールグラフを用いて依存関係を表示することは，予期せぬアスペクト間の依存関係が存在しないかどうか，開発者が確認することができるため，有益な情報であると考えられる．

コールグラフは，計算が単純である分，弱点も多い．その一つが，グラフの大きさの問題である．メソッド，アドバイスの数はプログラムの規模に比例して増えていくため，グラフのサイズもそれに比例して大きくなっていき，開発者が目視で確認するには大きくなりすぎる．そこで，ループ発生部分に対する警告メッセージの出力やグラフ上へのマーキング，不要な頂点の除去など，開発者が確認を容易にする機構と併用する必要がある．

今回使用したコールグラフの制限としては，他にも次のようなものがある．

- 無限ループの回避コードを考慮しない．無限ループの問題は，アスペクト指向プログラミングが提案された当初から知られており，AspectJ ではそれを回避するための典型的なコード [16] は既に有名なものとなっている．しかし，そのようなコードを記述は，実行時にのみ有効であり，今回のような単純な静的解析では無限ループの可能性として提示されてしまう．これに関しては，開発者が「予期しているループ」を明示的に宣言し，警告を出力しないように提示しておく必要がある．

- メソッドの再帰呼び出しを考慮しない．再帰呼び出しは，オブジェクト指向プログラミングでも様々な場面で用いられるが，これに連動するアスペクトは，再帰呼び出しのループの中に含まれてしまうため，無限ループ発生の可能性として提示されてしまう．これに関しては，無限ループ回避コードの場合と同様に「予期しているループ」の宣言による対処を行う．

- データに関する干渉は考慮しない．たとえば，あるデータを暗号化するアスペクトと，そのデータをファイルに書き出すアスペクトは，どちらが先に動作するかで結果が変わるため，干渉していると考えるべきである．しかし，コールグラフだけでは，このような問題には対処できない．本研究では，このような場合には，スライス計算によってアスペクトがどのように依存しているかを調べることで，問題に対処する．

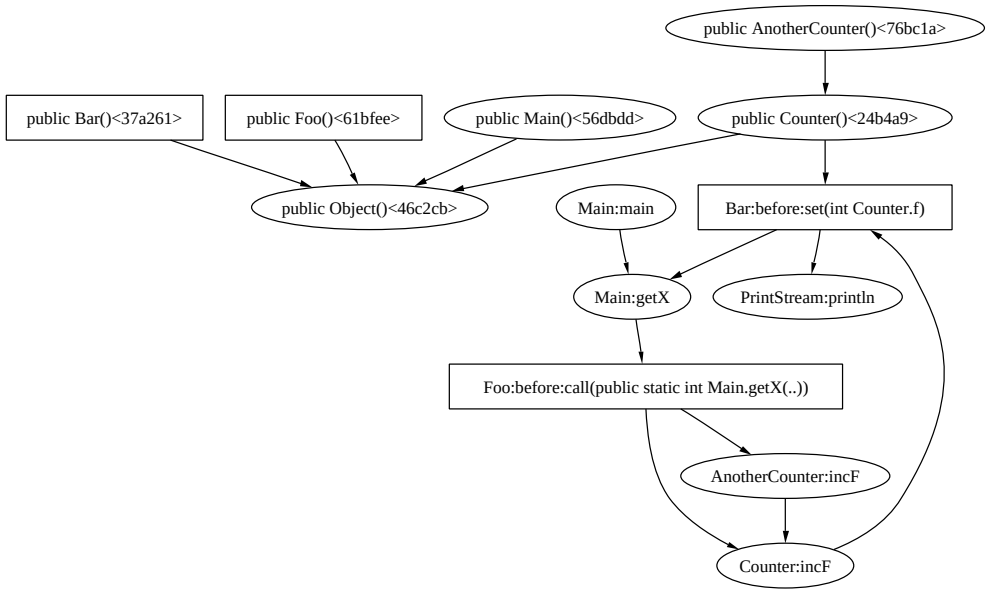

図 7: コールグラフ例

3.1 コールグラフの具体例

コールグラフの具体例を図 7 に示す．矩形の頂点はメソッド，楕円形の頂点はアドバイスを示しており，内部にシグネチャを示している．

メソッドを表現する頂点 Main:getX に連動して動作するアスペクト Foo から，Counter:incF，アスペクト Bar を経由して頂点 Main:getX にたどり着くループが存在していることから，無限ループに陥る可能性を知ることができる．また，Counter クラスのコンストラクタを表す頂点 (”public Constructor”) から Counter クラスのメソッド Counter:incF を表す頂点に到達可能であるため，Counter クラスの初期化中にメソッドが呼ばれ，動作不良を起こす可能性を知ることができる．

このようにグラフを用いて，アスペクト間の干渉が意図したものであるかどうか，またループを自動検出することで無限ループの可能性を検知し，ツールの利用者に提示するものとする．開発者が意図して再帰ループなどを作成する可能性もあるため，そのまま実行することも選択できるものとする．

4　プログラムスライシング

プログラムスライシングは，プログラムに含まれる手続き呼び出し関係や変数の参照・代入関係などの依存関係を解析し，プログラム内の注目すべき部分だけを抽出，提示する技術である [8]．具体的には，ある文のある変数を入力として，その変数の値に影響を与える文の集合を取り出す．入力として与えるコード内の変数の参照位置をスライス基点と呼び，抽出されたプログラム文の集合をプログラムスライス，あるいは単にスライスと呼ぶ．

4.1　スライス計算のアルゴリズム

プログラムスライシングは，次の三つのフェイズからなる．

(1) プログラムからの依存関係の抽出

(2) プログラム依存グラフの構築

(3) グラフ探索によるスライスの抽出

(1) は，データ依存関係，制御依存関係の二つの依存関係を解析するフェイズである．データ依存関係プログラム中の二つの文 s, t について以下の条件が成り立つとき，s から t に対して変数 v に関するデータ依存関係があると言う．

- 文 s で変数 v に値を代入している．
- 文 t で変数 v の値を参照している．
- 文 s から t に到達可能な制御フローがある．
- 文 s から t に到達する制御フローのうち，途中で変数 v への代入文がないような経路が少なくともひとつ存在する．

また，制御依存関係は，プログラム中の文 s, t について，以下の条件が成り立つとき，s から t への制御依存関係が存在すると言う．

- s が条件節である．
- t が実行されるかどうかが，s の判定結果によって決まる．

フェイズ (2) で，これらの依存関係と，手続き・メソッドの呼び出し関係を用いて，プログラム文を頂点，依存関係を有向辺としたグラフを作成する．このグラフをプログラム依存グラフ（Program Dependence Graph，以下 PDG) と呼ぶ．フェイズ (3) では，PDG 上

表 1: スライスの違い

種別	制御依存	データ依存	コスト	スライスサイズ
静的スライス	静的	静的	低	大 (すべての可能性)
DC スライス	静的	動的	中	中 (特定の実行系列)
動的スライス	動的	動的	非常に高い	小 (特定の実行系列)

でスライス基点となる頂点を選び，有向辺を逆向きに探索していくことで，ある注目したい文，変数に影響を与える文を抽出していく．得られた頂点集合をエディタなどに表示されたソースコード上へ反映し，プログラマへ情報を提供する．

プログラムスライシングの性能は，その依存関係の抽出方法によって決まる．情報をソースコードから取得するか，実行時にプログラムの動作を監視して取得するかによって，静的スライスと動的スライスに分けられている [8, 9]．静的スライスはすべての可能性を抽出するため，プログラム理解や検証などに用いられる．一方，動的スライスはある特定の入力に対する実行系列を解析することで，静的スライスよりも対象とするコードを絞り込むため，デバッグ等の支援に用いられる．

動的スライスは，プログラムの実行系列を保存する必要があるため，実行時のコストが非常に高くつくという問題があった．これに対して，制御構造については静的に解析し，データ依存関係とメソッド呼び出し関係を実行時に取り出す Dependence-Cache(DC) スライスが提案されている．DC スライスは，実行経過を監視することでデータ依存関係を抽出するが，実行系列を保存する必要がないため，実用的なコストで計算を行うことができる．DC スライスの計算結果は，静的スライスと動的スライスの中間となる [10, 12]．この三つのスライスの違いをまとめると，表 1 のようになる．

DC スライスの例を図 8 に示す．このプログラムは，IncrementCounter と ShiftCounter という二つのクラスを用意し，引数に応じて一方だけを用いて，結果を出力するプログラムである．ここで，IncrementCounter を用いて動作するような入力 ”inc” を与えたときの実行に対して，スライス基点として，最終出力である矩形 (d) の変数を選択すると，矩形 (a) 〜(f) で囲まれた部分が DC スライスとなる．

このスライスを見ることで，実際に動作した部分がどこかという情報が得られる．もし結果が不正な値となっていた場合，このスライスに含まれた部分を調べればよい．

4.2 アスペクト指向プログラムに対するスライス計算の適用

アスペクト指向プログラムに対してプログラムスライシングを適用し，デバッグを支援することを考える．開発者がバグを発見した際に，その原因となるコードを探し出す作業が

```
class Count {

  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("java Main [sft|inc]");
      return;
    }

    Counter counter;
    boolean isIncrementCounter = false;
    if (args[0].equals("inc")) {
      counter = new IncrementCounter();
      isIncrementCounter = true;
    } else if (args[0].equals("sft")) {
      counter = new ShiftCounter();
    } else return;                                    (a)

    int x = 0;
    for (int i=0; i<1000; ++i) {
      counter.proceed();
      x = counter.value();
      if (x > 1000) break;
      System.out.println(x);
    }                                                 (b)

    String result;
    if (isIncrementCounter) {
      result = "increment counter = ";
      result = result + Integer.toString(x);
    } else {
      result = "shift counter = ";
      result = result + Integer.toString(x);
    }                                                 (c)
    System.out.println(result);                       (d)
  }
}

abstract class Counter {
  private int count = 1;
  public Counter() {}
  public int value() { return count; }
  public void proceed() { count = newValue(count); }  (e)
  abstract protected int newValue(int old);
}

class IncrementCounter extends Counter {
  protected int newValue(int old) {
    return old + 1;                                   (f)
  }
}

class ShiftCounter extends Counter {
  protected int newValue(int old) {
    return old << 1;
  }
}
```

図 8: DC スライス計算例

必要となるが，そこで扱わなければならないコード量をいかに減らすかが重要となる．

本研究では，デバッグ支援に的を絞る．開発者は何らかのテストケースを実行し，その結果が不正である場合に，その原因を調査する作業を支援する．このような状況を想定すると，スライス計算の方法としては，プログラムの実行時情報を部分的に用いる，DC スライスが適切であると考える．発見されたバグを再現させるテストケースを入力としてプログラムを実行し，その実行を観測して得られた情報を基にプログラム依存グラフの構築とスライス計算を行い，開発者にフィードバックを行うものとする．

DC スライスを選択した根拠として，静的な解析だけでは情報が不十分なことが挙げられる．オブジェクト指向プログラミングで導入されたオブジェクトの多態性，動的束縛という機構では，ある一つのメソッド呼び出し文に対応して実際に動作するメソッドが，実行時のコンテキストに依存して決定される．このため，開発者が扱うべきコードの量を減らすことが難しい．また，アスペクト指向プログラミング特有の問題として，実行コンテキストに依存したアスペクトの動作条件の指定が可能である，同一の結合基準に対して複数のアスペクトが連動した場合の動作順序は開発者が特に指定していない限り処理系依存となる，といったことが挙げられる．

現在，オブジェクト指向言語 Java に対するプログラムスライシングが既に実現されているが，アスペクト指向プログラムに対する適用は基本的な手段の提案のみであり，詳しい議論はなされていない [19]．

本研究では，対象とするアスペクト指向言語として AspectJ を選択し，Java に対するプログラムスライシングを拡張することで実現する．具体的には，アスペクトが結合された場所からその動作するアスペクトへと手続き呼び出しと同様の関係があると考え，その依存関係を PDG に追加する．

5 プログラム実行時情報の解析

DC スライスでは，プログラムの実行時情報が必要となる．ここで必要なプログラム実行時情報とは，次の通りである．

動的データ依存関係 どこで代入された変数がどこで用いられたか，という依存関係である．一般的には，次のようなアルゴリズムによって計算される．

事前準備
使用される各変数 v に対応したキャッシュ $C(v)$ を用意する．

文 t で v が定義された場合
$C(v)$ の値を t の文番号に更新する．

文 t で v が参照された場合
$C(v)$ に対応する命令と t に対応する命令の間に発生する v に関するデータ依存関係を抽出する．

ただし，実装の際には，オブジェクトのフィールドに関する依存関係に関しては，オブジェクトごとにキャッシュを用意するといった工夫が必要である．

動的束縛の解決 あるメソッド呼び出し文に対して，実際に対応したメソッドがどれか，という情報である．静的解析に比べて，プログラムのコードを絞り込むための有力な材料の一つである．具体的には，次のようなアルゴリズムで計算できる．

メソッド呼び出し直前 メソッド呼び出し文の位置を記録する．また，呼び出そうとしているメソッドを記憶する．

メソッド実行直前 (動的束縛の解決後) 呼び出し文の位置から，呼び出されたメソッドへのメソッド呼び出し情報を記録する．

例外処理 例外の発生による制御の移動が起こったかどうか．静的解析では例外が起こる可能性を考慮していくことは難しいので，動的に解決する．

従来，オブジェクト指向言語 Java を対象としたプログラム実行時情報の解析には，次のような実現方法が利用されていた．

(a) プリプロセッサによる解析命令の埋め込み [12]

(b) Java Virtual Machine Profiler Interface (JVMPI) の利用 [13]

(c) Java Debugger Interface の利用 [17]

(d) Java Virtual Machine (JVM) の改造 [11]

(a) は，Java の構文木上での変換ルールを作成し，解析命令を埋め込む方法である．解析命令と単純に言っても，マルチスレッド動作への対応や，例外処理など，数多くの要素に対処する必要があるため，プリプロセッサの構文的な変換だけでは対応しにくいという問題がある．またプリプロセッサそのものの保守性や再利用性，他のプリプロセッサとの競合などへの対策も必要であり，実現コストが高くなる傾向にある．

(b) は，JVM に用意されているプログラムの性能計測のためのインタフェースである．対象 JVM に監視プログラムを付加して実行することができ，CPU の時間消費やメモリの使用量を調べることができる．メソッドの呼び出しやスレッド，メモリの管理など，主要なプログラムの動作を監視する機能が提供されているが，イベント生成のオーバーヘッドが大きく，実行時のコストが高くなるという問題がある．JVMPI はネイティブインタフェースとして C 言語などで実装する必要がある．また，イベントが非同期で生成されるために同期処理は自分で実装する必要がある，内部でエラーを発生させてしまうと監視対象の JVM ごと異常終了してしまうことから，作成したプロファイラのデバッグが難しいという問題もある．

(c) は，Java を用いてデバッガを作成するための機構とライブラリである．JDI を用いたプログラムは，デバッグ対象とする JVM の持つ Java Virtual Machine Debugger Interface (JVMDI) と通信し，ブレークポイントの設置，フィールドやメソッド呼び出しイベントの取得，各時点でのスタックフレームの取得など，デバッグのための種々の機能が利用できる．しかし，デバッガはソケットを介した通信を行うほか，JVM の状態を取得するためにプログラム本体の実行を頻繁にブロックすることから，オーバーヘッドは大きい．JVMDI を直接扱うこともできるが，その場合でも JVMPI と同様の問題を持つことになる．

(d) は，JVM の公開されたソースコードに手を加えて，プログラムの動作を監視する方法である．この方法は，Java の実行環境におけるすべての情報にアクセスできるという利点がある．しかし，JVM の実装に依存し，JVM のバージョンアップへの対応が必要である．また，(b)(c)(d) 共通して，バイトコードレベルでの処理が必要であり，Just In Time(JIT) コンパイラによる最適化を行うと，得られる結果が変わってしまう可能性がある．そのため，最適化の抑止が必要となり，結果としてパフォーマンス上のオーバーヘッドが生じる．

これらに対し，アスペクトによるプログラム解析の実現は，抽象的な Join Point という形式でプログラムの結合を行うことができるため，(a) の持つ問題点の影響を受けない．また，アスペクトは実行環境ではなくプログラムを変換するため，(b)(c)(d) が持つ JVM への依存性の影響を受けずに済むという利点がある．また，アスペクトは Java ソースコードに変換されるため，小さなプログラムに対してアスペクトを結合し，アスペクトの持つ欠陥を検出することが容易である．

5.1 AspectJ による動的解析の実現

AspectJ の持つ，ソースコード位置情報へアクセスする機能を用いることで，フィールドの参照や代入，メソッド呼び出しに対して，プログラム内の依存関係の解決を行うことができる．

AspectJ を用いると，データ依存解析および動的束縛解決のアルゴリズムは次のように記述することができる．

- **データ依存関係の解決**

  フィールドへの値の代入 代入されたオブジェクトへの参照と，そのフィールドのシグネチャ，代入文の位置を記録する．

  フィールドの値の参照 参照されたオブジェクトと，そのフィールドのシグネチャに一致する代入文の位置を取得し，参照した文の位置へのデータ依存関係を記録する．

- **動的束縛の解決**

  メソッド呼び出し スレッドごとに用意されたスタックへ，メソッド呼び出し位置と呼び出したメソッドの内容を記録する．

  メソッド実行 スレッドごとに用意されたスタックを見て，呼び出し位置から，実際に呼び出されたメソッドへの制御依存関係を記録する．

  メソッド呼び出し終了 スレッドごとに用意されたスタックから，呼び出し情報を取り除く．

  例外の発生 メソッド呼び出し終了と同様の処理を行う．

データ依存関係の解決を行うコードの抜粋を図 9 に，動的束縛の解決を行うコードの抜粋を図 10 に示す．実際には，図に示したコードをマルチスレッドに対応させたものとなっている．

依存関係解析アスペクトは，AspectJ のワイルドカード指定機能を用いて，解析対象となるクラスのすべてのフィールド参照と代入に対して動作するように定義している．

この実装は，利用者がアスペクトのコードを操作せずに利用するためのものである．しかし，ユーザが監視対象から外したいクラスがある場合は，AspectJ の継承機能を用いて，監視対象から除外したいクラスを再定義した新しいアスペクトを作成することができる．

アスペクトが対象プログラムの本来の振る舞いを破壊することはない．アスペクトの結合によって，対象となるプログラムのデータフローと制御フローが変化する．しかし，データフローについては，対象プログラムの値を観測および記録はするが値を書き換えることは

```
public aspect DataDependsAnalysisAspect {

  pointcut target():
    !within(slice.aspect.*);

  pointcut exclude():
    within(somepackage.*);

  pointcut field_set():
    target() && !exclude() &&
    (set(* *) || set(static * *));

  pointcut field_get():
    target() && !exclude() &&
    (get(* *) || get(static * *));

  FieldDef def = new FieldDef();

  before(): field_set() {
    def.put(
      thisJoinPoint.getTarget(),
      thisJoinPoint.getSignature(),
      thisJoinPoint.getSourceLocation());
  }
  before(): field_get() {
    SourceLocation setpos =
      def.get(thisJoinPoint.getTarget(),
              thisJoinPoint.getSignature());
    Logger.logDataDepends(
      thisJoinPoint.getTarget(),
      thisJoinPoint.getSignature(),
      setpos,
      thisJoinPoint.getSourceLocation());
  }
}
```

図 9: AspectJ によるデータ依存解析（抜粋）

```
aspect LoggingAspect {

  pointcut AllMethodCalls():
    !within(LoggingAspect) &&
    call(* *.*(..));

  pointcut MethodExecs():
    !within(LoggingAspect) &&
    execution(* somepackage.*.*(..));

  static Stack callStack = new Stack();
  static JoinPoint lastCall = null;

  Object around(): AllMethodCalls() {
    callStack.push(thisJoinPoint);
    lastCall = thisJoinPoint;
    proceed(); // execute original call
    lastCall = callStack.pop();
  }

  before(): MethodExecs() {
    if (lastCall != null) {
      Logger.logs("executed",
        lastCall.getSignature(),
        lastCall.getSourceLocation(),
        thisJoinPoint.getSignature(),
        thisJoinPoint.getSourceLocation());
    }
  }
}
```

図 10: AspectJ による動的束縛の解析（抜粋）

なく，またオブジェクトを弱参照で取り扱うため，対象プログラムの振る舞いを変更することはない．弱参照とは，オブジェクトへの他のすべての参照がすべて破棄されると自動的に破棄されるような参照の機構であり，Java では標準で提供されている．制御フローに関しては，単純な実装では無限ループが発生する可能性があるという問題があるため，それを回避するための実装を行っている．この問題については，実装上の制限として後で詳しく説明する．

5.2 AspectJ における実装上の制限

5.2.1 Join Point の制限

アスペクト指向プログラミングでは，利用可能な Join Point と，それに対して適用可能な演算によってアスペクトの記述可能な範囲が制限される．AspectJ では，ローカル変数の読み書きや制御構造は Join Point として含まれない．これは，ローカル変数や制御構造に対する横断処理が必要とされるケースが少ないこと，またパフォーマンス上著しいオーバーヘッドを引き起こすことに起因する．

動的データ依存解析では，本来ならばすべての変数における値の授受を監視しなければならないため，AspectJ では厳密な実装は不可能である．しかし，ローカル変数に関するデータ依存関係は単一の手続き内で完結しているため，オブジェクト指向における実行時決定要素の影響を受けにくく，静的に解析しても十分な精度を得られると予測される．この件に関しては，後述する適用実験の考察で議論する．

5.2.2 ソースコードの制限

AspectJ はソースコードに対してアスペクトの結合を行うため，ライブラリに対してはアスペクトの結合を行うことはできない．ここでライブラリとは，Java ソースコードが存在しないバイナリ形式の再利用可能なコンポーネントを指す．

これに対して，本研究では，以下の理由からライブラリは解析対象から除外する方針を採った．

ライブラリの信頼性は高い． ライブラリは再利用の単位であり，その内部は十分に信用できるコードであると考えられる．そのため，必要以上に詳細な解析は必要ない．

ライブラリのコード量は非常に多い． ライブラリの量は，利用するプログラムと比較して非常に多く，動的に解析するコストが高くなる．

プログラムがライブラリ側からのコールバックを利用する場合，プログラムのある地点からライブラリ内部を経由してプログラムの別の地点へと，隠れた依存関係を生じることがあ

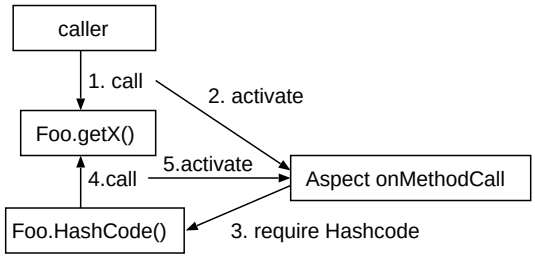

図 11: ループ発生例

る．これは Java バイトコードでの依存解析を行うことで知ることができる [11]．しかし，ファイル入出力やデータ構造のような基本的で重要なオブジェクトに対しては，後述する Java 言語上の制限から，バイトコードを用いても依存解析を行うことができない．そのため，バイトコードレベルでのアスペクトの結合を行えたとしても，実際に影響を与えられる範囲は広いとは言えず，ソースコードが存在する範囲での結合と解析で十分である．

実行時の情報を使用しない，ライブラリに対するメソッド呼び出し部分については，静的解析によって補うことになる．具体的には，メソッドの引数，参照しているオブジェクトから戻り値へのデータへの依存関係があるとみなす．また，呼び出した結果，そのクラスを経由して他の観測対象となるメソッドが一つ以上呼び出される可能性もある．これについては，ライブラリへの呼び出しから，実際に呼ばれたメソッド群に対しての呼び出し関係を記録し，依存関係を設定する．

5.2.3 Java 言語上の制限

AspectJ ではアスペクトを Java で平易に記述できるという利点があるが，アスペクトにも，データの収集に利用するクラスに対して依存関係が生じてしまう．そのため，モジュールが利用しているクラスに対してアスペクトを結合して解析しようとすると，ループが生じることがある．

ループの発生例を 11 に示す．この図では，メソッド Foo.getX を呼び出すが，そのメソッド呼び出しに対応してアスペクトが作動する．アスペクトは Foo に対してハッシュコードを要求するが，Foo.hashCode が getX メソッドを用いて計算されている場合，getX 呼び出しが再びアスペクトが作動してループに陥ってしまう．

このループの発生の問題は，バイトコードを加工するアプローチであっても同様で，JVM 改造アプローチのような言語の枠を越えた手段を用いない限り本質的に解決することはできない．

しかし，Java 標準ライブラリのクラスに対する解析を行わない限り，ループの原因とな

るメソッドは限られる．具体的には，アスペクトから標準ライブラリを通じて間接的に呼ばれるオブジェクトの文字列表現への変換 (Object.toString)，ハッシュコードの計算 (Object.hashCode) の二つである．アスペクトから toString，hashCode への呼び出しを避けること，また toString，hashCode へのアスペクトの結合を避けることでこの問題を回避することができる．この対処は toString，hashCode について収集する情報を制限してしまうが，これらのメソッドの役割は，通常そのメソッドだけで完結しているので，このような原因による情報の完全性の低下は，実用上の影響を与えないと考えられる．

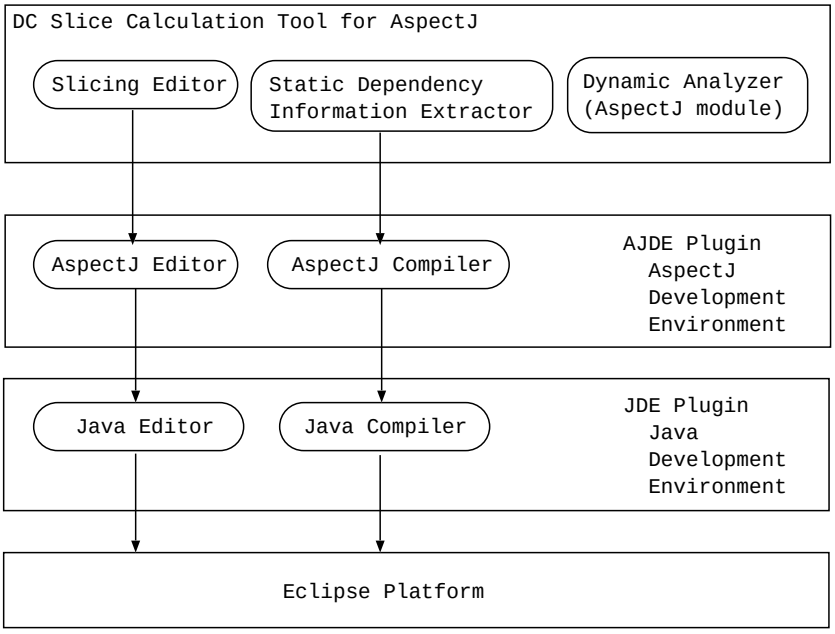

図 12: ツールの構成

6 実装と評価

6.1 実装の概要

デバッグ作業は，コードを書き換え，テストケースを再実行するという反復作業として考えられる．このような反復テストをサポートする統合開発環境も多く，プログラムスライシングのような支援ツールも，それらと統合して利用可能にすることが望ましい．本研究では，統合開発環境として Eclipse [20] を選択し，これに統合する形式でツールの実装を行った．

Eclipse はオープンソースの統合開発環境で，Java で記述したプラグインを追加することでエディタやコンパイラの機能を拡張することができる．Java および AspectJ を対象としたソースコード入力等の開発支援プラグインが既に開発されているため，それにスライス計算機能を上乗せする形式で実装を行った．図 12 に，プラットフォームの構成を示す．

Eclipse のプラグインでは，ファイルの保存やコンパイルの終了など，重要なイベントに応じて動作するアクションを定義することができる．そこで，次のようなツールとして実装した．

コンパイル時の静的情報抽出 AspectJ のプログラムがコンパイルされたとき，ローカル変数および制御構造に関する静的依存関係の抽出を行う．また，メソッドおよびアドバ

イスの呼び出し関係を抽出し，コールグラフの構築，出力を行う．

静的スライス計算 テキストエディタ上でソースコードを選択し，ツールバーからスライス計算を指示することで，静的情報のみによるスライス計算を行う．静的情報のみの場合，メソッドの動的束縛やフィールドのデータ依存関係のすべての組み合わせを考えるため，結果としてはそれほどソースコード量は減らせないが，この時点でも，出力データがどのような経路から届く可能性があるか，調べることができる．計算結果は，エディタ上に直接反映される．

動的解析 動的解析アスペクトを付加してプログラムを実行する．動的解析アスペクトを加えてプログラムをコンパイルし実行する．動的解析アスペクトは，解析結果をファイルに出力する．Eclipse の Java プラグインが，コンパイルすべきファイルの構成や種々の設定を取り扱うための機構を提供しているため，それを利用することで，本来の構成と区別して扱うことができる．

DC スライス計算 動的解析の結果が存在する場合，ユーザはそのデータを読み込んでスライス計算に反映させることができる．計算結果がエディタ上に表示されるのは，静的スライスの場合と同様である．

DC スライス計算には実行時のプログラムを監視する処理が必要である．この処理自体も横断要素の一つとして考えると，アスペクトを用いて実現することが自然である．AspectJ ではローカル変数の監視はできないが，実行時のオブジェクトの個々のインスタンスの識別と，フィールドのデータ依存関係を取り出すだけで十分有効であることは Java を対象としたプログラムスライシングの実現で既に示されている [2]．　システムの動作概要を図に示す．利用者はまず静的な呼び出し関係をツールによって調べ，ループが存在する場合はそれを実行しても問題ないか確認する．次に実行を監視するアスペクトを追加し，そのアスペクトの動作が他のアスペクトの振る舞いの影響を受けないこと，あるいは影響を受けても問題ないことを確認する．そしてテストケースを与えてプログラムを実行し，DC スライスに必要な実行時情報を抽出する．得られた情報を静的な解析情報と合わせて PDG を構築し，ユーザがエディタから指定したスライス基準から，グラフ探索を行ってスライスを計算，結果をエディタに反映する．

ソースコードの規模としては，ユーザインタフェースおよびスライス計算部が 4300 行，動的解析部が約 1000 行となった．これは，AspectJ などの既存プラグインの機能を利用したことで，必要最小限の実装で済んだためである．

ここで，ツールのスクリーンショットを図 13 に示す．エディタ部に引かれている波線が，計算されたスライスの結果である．

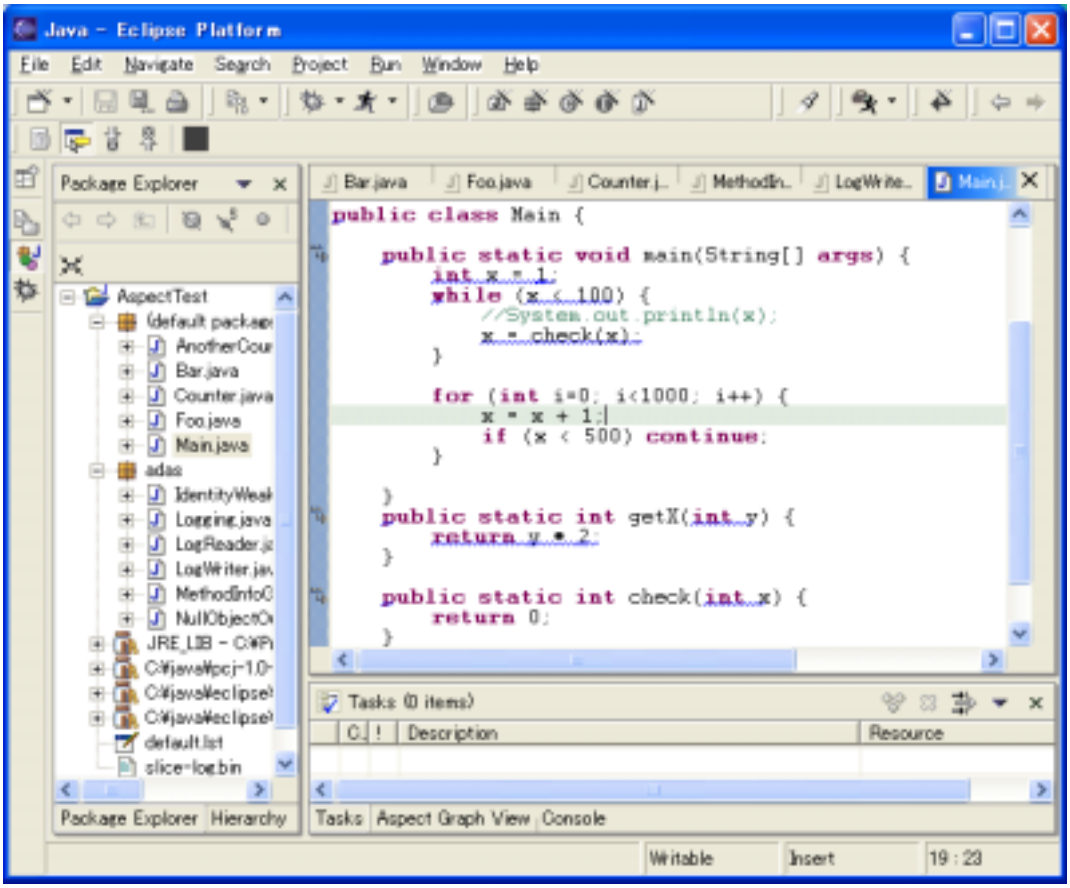

図 13: スクリーンショット

表 2: 適用したコード

名称	サイズ	アスペクト
ChainOfResponsibility	517	ChainOfResponsibility, MyChain
Observer	667	ObserverProtocol, ScreenObserver, ColorObserver
Singleton	375	SingletonProtocol, SingletonInstance
Mediator	401	MediatorProtocol, MediatorImplementation
Strategy	465	StrategyProtocol, SortingStrategy

表 3: 静的スライスと DC スライスでのサイズ比較

対象	静的スライス	DC スライス
Singleton	200	173
Strategy	113	60

6.2 適用実験

作成したツールを，Java および AspectJ を用いたデザインパターンの実装例 [3] に対して適用した．デザインパターンは全部で 23 種類あるが，そのうち，アスペクトとして再利用可能なモジュールとなることが判明している ChainOfResponsibility，Observer，Mediator，Singleton，Strategy パターンを実装したサンプルコードを使用した．これは，文献 [3] の作者らがインターネット上で公開しているものである [15]．

表 2 に示す．いずれも，パターンの骨組みを抽象アスペクトとして定義している．使用する際には，アスペクトを継承して，たとえば Singleton パターンならどのクラスが Singleton の役割を果たすのか，といった実装の詳細を定義したアスペクトを定義するという方式になっている．

それぞれの実装例をコンパイル，実行し，スライス計算を行った．以降，その結果について述べる．

6.3 スライス結果

スライス基点としてアスペクトの中の頂点を選んで，スライス計算を行った．計算した結果のうち，Singleton パターンと Strategy パターンの実装例に対する静的スライスと DC スライスのサイズの例を示す．これらは，どちらも静的スライスの「すべての可能性を考える」性質によって差が生じていた．他のパターンについては，Java のコレクションフレームワーク [18] のような動的なデータ構造を経由した依存関係を単純な静的スライスでは追

いかけることができなかったため，静的スライスが正しい結果を出力することはできなかった．DC スライスでは，実行時情報からこれらの依存関係を取り出すことに成功している．

得られたスライス結果と同様の結果を手作業で得るためには，次のような作業が必要であった．

1. スライス基点としてアドバイスを選んでいるので，そのアドバイスがいつ実行されるかを調べるために，親アスペクト（継承元）のアスペクトの結合基準の定義を調べる．
2. 親のアスペクトでは結合基準が抽象クラスを対象に定義されているため，その抽象クラスを継承したクラスを列挙する．
3. クラスごとに，結合基準となる点を探す．
4. 見つけた結合基準から，基点となったアドバイスへ送られているデータの依存関係を調べる．

この作業での最も負荷の高い点は，複数のファイルに分散して定義された，クラスやアスペクトの定義を追跡する作業であった．プログラムスライシングを用いて，関連した文を機械的に列挙することで，作業効率の向上が期待できる．

また，プログラムスライシングを用いることで，開発者の誤解しやすい点を指摘することができる．図 14 は，あるメソッド呼び出しを別のメソッド呼び出しに置き換えるアスペクトを含んだプログラムである．一見した限りでは，依存関係は図に矢印で示したように見えるが，実際の依存関係は図 15 のようになる．

このようなメソッド置き換えアスペクトは，単体テスト時の便宜的な実装の交換や，未実装部分のサポートなどに使用される．通常，アスペクトがクラスとは別のファイルに記述されているため，このようなアドバイスの定義による依存関係の違いを認識することは難しい．また，図 14 と図 15 の例では一行分の違いしかないが，実際にはこのクラスを継承したクラスや，そこで使用されている他のクラス，アスペクトへと依存関係は波及していく恐れがある．

このような，アスペクトによって変化した振る舞い，依存関係を指摘することで開発者を支援できることが，プログラムスライシングの大きな利点である．

6.4 実行コスト

スライス計算の実行プロセスにおいて，実行コストは以下の場面で影響を与える．

コンパイル時 静的情報収集のために計算コストとメモリを必要とする．

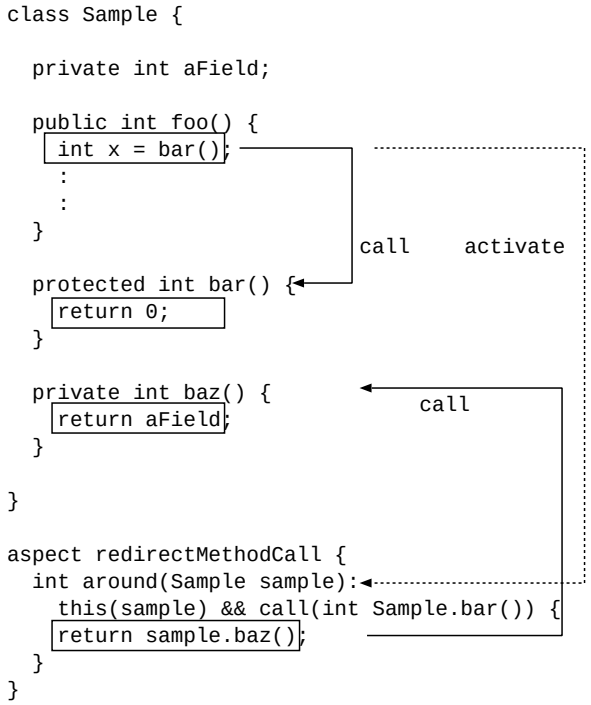

図 14: メソッド置換アスペクトの例

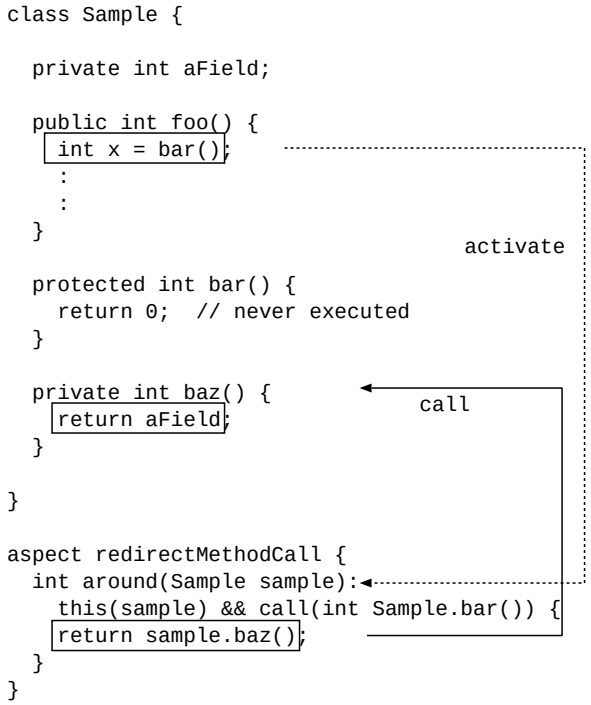

図 15: メソッド置換アスペクトのスライス結果

表 4: 動的解析処理の実行時間

対象	実行時間	動的解析を付加した実行時間
ChainOfResponsibility	3.76	3.93
Observer	0.32	0.37
Mediator	3.21	5.69
Singleton	0.14	0.32
Strategy	0.18	0.22

動的情報解析時 動的情報収集モジュールを付加するため，通常の実行に比べて計算コストとメモリを必要とする．

スライス計算実行時 スライス計算は，構築されたグラフの探索問題であり，頂点数に比例した時間で終了する．

コンパイル時の処理は，AspectJ コンパイラが意味解析なども含めて構築したモデル情報にアクセスできるため，構文木を 1 回トラバースするだけで終了する．アルゴリズムの計算量の厳密な評価ではないが，トラバース処理は構文木のサイズに比例し，コンパイラの計算量に比べれば相対的に小さい．

動的情報解析に必要な時間コストを表 4 に示す．

また，メモリ消費量に関しては，解析結果のプログラム依存グラフのサイズによる違いが影響を与えるが，アスペクトの種別によって，大きな差が出ることが判明した．

上記に示したようなデザインパターンを記述したアスペクトでは，デザインパターンすべてコンパイルし解析した結果，コード行数は合計で 10000 行ほどで，メモリは約 20MB，統合開発環境の基本消費量を合わせても約 60MB のメモリしか消費しなかった．これに対して，動的解析アスペクトのような，すべてのメソッド呼び出しとフィールドアクセスを記録するアスペクトを付加したところ，ソースコードのサイズは 1 割，1000 行ほどしか増加しないが，依存関係の量が膨れ上がることで，作業に 500MB，25 倍以上のメモリを消費する結果となった．

この結果は，アスペクト指向プログラムの依存関係の増加量が，単純なソースコードの行数では測れず，アスペクトの定義を調べなければならないことを示している．アスペクト指向言語のコンパイラが，依存関係が多すぎるために動作を失敗する可能性 [16] が従来より指摘されていたが，それと同様の問題を，プログラムスライシングも抱えている．コンパイラが用いている分割コンパイルのような技術を，プログラムスライシングでも活用する必要がある．

7　むすび

本研究では，アスペクト指向プログラムに対してプログラムスライシングを適用し，デバッグ支援を行うための開発環境の実装を行った．

アスペクト指向プログラミングの特徴は，横断要素と呼ばれる複数のオブジェクトが関わる処理を，アスペクトという単位でモジュール化することにある．横断要素のコードが複数のオブジェクトに分散することがなくなり，保守性，再利用性を改善することができる．

アスペクトは，行うべき処理と，その処理を行う動作条件とをペアにしたアドバイスという単位の集合として記述される．アドバイスは，オブジェクトにおけるメソッドに類似した単位である．動作条件を書いておけば，言語処理系が実際の呼び出し処理を生成してくれるので，オブジェクト側のコードを書き換えることなく，オブジェクトの振る舞いを変えることができる．

アスペクトの利点は多いが，オブジェクトから明示的な呼び出し処理なしに実行されるため，予想外の場所でアスペクトが動作してしまう，あるいは複数のアスペクトが動作するために結果が予想できない，といった問題が発生することがある．このような問題への対処のために，コールグラフとプログラムスライシングを用いる．

コールグラフは，メソッド呼び出しとアスペクトの連動関係を抽出した有向グラフである．このグラフ上で無限ループの検出や，実行を不可能にするような呼び出し関係を調べることができる．コールグラフによって制御の誤りのうち重大なものが取り除かれた段階で，プログラムの出力が得られるようになる．ここで，出力結果が正しくないという場合に，プログラムスライスを用いた原因の調査を行う．

アスペクト指向プログラムに対してプログラムスライシングを適用することは，アスペクトの起動をメソッドの呼び出しと同等に考えることで実現される．ただし，同一の結合基準に複数のアスペクトが連動する場合，その動作順序はコンパイラや実行環境の実装に依存するため，単純な静的スライスではデバッグ支援には不足だと考えられる．そこで，スライス計算アルゴリズムとして，データ依存関係として実行時情報を用いる DC スライスを採用した．

以上を踏まえて，コールグラフ構築および DC スライス計算ツールを統合開発環境 Eclipse へのプラグインの形式で実装した．そして，アスペクトを用いたサンプルプログラムに対してスライス計算の実験を行い，原因の検出に有効であることを示した．

アスペクト指向プログラムに対するプログラムスライシングでは，アスペクトの結合基準の定義によって生じる依存関係の量によって，メモリ消費量が大きく左右されることが判明した．大規模なプログラム，多数のアスペクトを扱うために，必要最小限の範囲での依存関係の解決を行うなど，スケーラビリティの改善を今後の課題とする．

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝致します.

本論文を作成するにあたり，常に適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 楠本 真二 助教授に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を賜りました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 助手に心から感謝致します.

最後に，その他様々な御指導，御助言等を頂いた 大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上研究室の皆様に深く感謝いたします.

参考文献

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin: ”Aspect Oriented Programming”, Proceedings of ECOOP, vol.1241 of LNCS, pp.220-242(1997).

[2] 石尾隆, 楠本真二, 井上克郎: ”アスペクト指向プログラミングの動的スライス計算への応用”, 2002 年電子情報通信学会総合大会講演論文集，D-3-4，p.30 (2002).

[3] J. Hannemann, G. Kiczales: ”Design Pattern Implementation in Java and AspectJ”, Proceedings of OOPSLA 2002, pp.161-173, Nov. (2002).

[4] S. Soares, E. Laureano, P.Borba: ”Implementing Distribution and Persistence Aspects with AspectJ”, Proceedings of OOPSLA 2002, pp.174-190, Nov. (2002).

[5] R. Pawlak, L. Seinturier, L. Duchien, G. Florin: ”JAC: A Flexible Solution for Aspect-Oriented Programming in Java”, Proceedings of REFLECTION 2001, pp.1-24 (2001).

[6] I. Kiselev: ”Aspect-Oriented Programming with AspectJ”, Sams Publishing, Indiana (2002).

[7] 一杉裕志, 田中哲, 渡部卓雄: ”安全に結合可能なアスペクトを提供するためのルール”, ソフトウェア科学会第 19 回大会, Sep.(2002).

[8] M. Weiser: ”Program slicing”, IEEE Transactions on Software Engineering, SE-10(4):352-357(1984).

[9] H. Agrawal and J. Horgan: “Dynamic Program Slicing”, SIGPLAN Notices, Vol.25, No.6, pp.246-256 (1990).

[10] T. Takada, F. Ohata, K. Inoue: ”Dependence-Cache Slicing:A Program Slicing Method Using Lightweight Dynamic Information”, Proceedings of the 10th International Workshop on Program Comprehension (IWPC2002), pp.169-177, Paris, France, June (2002).

[11] 誉田 謙二，大畑 文明，井上 克郎，”Java バイトコードにおけるデータ依存解析手法の提案と実現”，コンピュータソフトウェア，Vol.18, No.3, pp.40-44(2001).

[12] F. Ohata, K. Hirose, M. Fujii, and K. Inoue: ”A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information”, Proceedings of APSEC2001, pp.273-280(2001).

[13] S. Kusumoto, M. Imagawa, K. Inoue, S. Morimoto, K. Matsusita and M. Tsuda: "Function point measurement from Java programs", Proc. of the 24th International Conference on Software Engineering, pp. 576-582 (2002).

[14] E. Gamma, R. Helm, R. Johnson, J. Vlissides: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley (1995).

[15] Jan Hannemann: "Aspect-Oriented Design Pattern Implementations", http://www.cs.ubc.ca/ jan/AODPs/

[16] AspectJ Team, "The AspectJ Programming Guide", http://aspectj.org/doc/dist/progguide/

[17] "Java Platform Debugger Architecture", http://java.sun.com/j2se/1.4/docs/guide/jpda/architecture.html

[18] "The Collections Framework", http://java.sun.com/j2se/1.4/docs/guide/collections/

[19] J.Zhao, "Slicing Aspect-Oriented Software", Proceedings of IWPC2002, pp.251-260 (2002).

[20] Eclipse Project, http://www.eclipse.org/