アスペクトを用いた表明の記述

石尾隆[†]　神谷年洋[‡]　楠本真二[†]　井上克郎[†]

[†] 大阪大学 大学院情報科学研究科
〒 560-8531 大阪府豊中市待兼山町 1-3

[‡] 独立行政法人 科学技術振興機構 さきがけ
〒 560-8531 大阪府豊中市待兼山町 1-3

契約による設計や Java や C++ などの assert 文といった機構は，一般に表明と呼ばれる．しかし，オブジェクト指向プログラミングにおいては，表明は再利用を阻害してしまう場合があり，また複数のオブジェクト間の関係に対して表明を記述することができないなど，いくつかの制約がある．アスペクト指向プログラミングは，オブジェクトを横断した要素を単一のモジュールとして記述することができるため，アスペクトとして表明を記述することで表明の持つ弱点を補うことができると考えられる．本稿では，Java を題材に，アスペクトとして表明を記述することで得られる利点について議論する．

Assertion with Aspect

Takashi Ishio[†]　Toshihiro Kamiya[‡]
Shinji Kusumoto[†]　Katsuro Inoue[†]

[†] Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama-cho, Toyonaka,
Osaka 560-8531, Japan

[‡] PRESTO, Japan Science and Technology Agency
1-3 Machikaneyama-cho, Toyonaka,
Osaka 560-8531, Japan

Programmers use assertion to enforce design by contract. However, in Object-Oriented Programming, assertion has few drawbacks. One drawback is that assertion may prevent the reuse of a class. Another is that programmers cannot express assertion about inter-object relations. Using Aspect-Oriented Programming technique, which enables to describe a crosscutting concern in one module, programmers can write assertions of inter-object relations. In this paper, we discuss the effect of writing assertions as aspects.

1 はじめに

契約による設計（*Design by Contract*）は，部品ごとの責任の範囲を明確に表現することで，ソフトウェアを機能部品に分解する際の安全性を向上する．契約による設計は，手続きのような処理単位において，その開始時に成立しているべき条件（事前条件）と，終了時に成立しているべき条件（事後条件）とを記述することで実現される．契約による設計や，Java や C++ における assert 文といった機構は，一般に表明（*assertion*）と呼ばれる．

事前条件は手続きへの誤った入力を防ぎ，事後条件は手続きの誤った出力を防ぐ役割を持つ [16]．事前条件が満たせない場合は手続きを利用する側，事後条件が満たせない場合は手続きを実装する側の責任となる．このような責任の分離は，開発者が作業の分担を明確にできるほか，欠陥の原因を切り分ける際の指標となるといった効果がある．

オブジェクト指向プログラミング言語では，しばしばクラスが部品の単位となり，クラスの持つメソッドごとに事前条件，事後条件を記述する．契約による設計を言語要素として取り込んだものとしては Eiffel[14] などがある．Eiffel では，クラスのメソッド単位での事前条件および事後条件を，メソッド本体とは区別して記述することが可能である．また，例外（*Exception*）を，事前条件が満たされているにもかかわらず事後条件が満たせない状況と規定している．Java や C++ で同様の記述を可能にするツールとして，Larch[8]，iContract[9]，jContractor[11]，JML[10] などがある．

しかし，メソッド単位で条件を記述する場合，メソッドの呼び出し側に対する条件や，複数のオブジェクトが連動する一連の処理に対する条件などはうまく記述することができない，この原因は，次のように考えられる．

(1) オブジェクトが他のオブジェクトに依存した条件を持っている，
(2) 制御フロー（呼び出し側オブジェクト）に関する情報を使えない，
(3) 一時的に条件を崩している処理中のオブジェクトが，再帰的に呼び出されることがある．

本稿では，アスペクト指向プログラミングと呼ばれる，複数のオブジェクトを横断した関心事をアスペクトとして分離するプログラミング技術を利用して，複数のオブジェクトを横断した条件やオブジェクト間の相互作用にかかわる条件を分離記述する方法を提案する．

オブジェクトから一部の条件をアスペクト側に分離する利点は，特定の環境に依存した表明などをオブジェクトから分離して記述することが可能になることと，必要に応じて表明の追加および取り外しができるようになることである．

また，提案手法では，制御フローに関する条件式を導入する．これは，AspectJ で導入された cflow pointcut の概念を表明に特化したものである．特定の呼び出し側に限定した条件を記述することで，開発者の意図をより明確に表現できる．

以降，2. では背景の詳細について，3. では提案手法について説明する．4. では例題に対してどのように働くかを評価を行い，5. で関連研究について述べる．最後に 6. でまとめる．

2 背景

2.1 契約による設計

契約による設計（*Design by Contract*）は，部品ごとの責任の範囲を明確に表現することで，ソフトウェアを機能部品に分解する際の安全性を向上する．契約による設計は，手続きのような処理単位において，その開始時に成立しているべき条件（事前条件）と，終了時に成立しているべき条件（事後条件）とを記述することで実現される．

事前条件はコンポーネント側の要求をアプリケーションに伝えることで手続きへの誤った入力を防ぎ，事後条件は，コンポーネントの誤った処理からアプリケーションを保護する役割を持つ [16]．オブジェクト指向プログラミング言語の場合，事前条件と事後条件はオブジェクトのメソッド単位で記述される．事前条件が満たせない場合は手続きを利用する側，事後条件が満たせない場合は手続きを実装する側の責任となる．このような責任の分離は，開発者ごとの作業の分担を明確にし，欠陥の原因を切り分ける際の指標となる．

開発者は，手続きの開始と終了の時点以外にも，その途中の動作が正しいことを確認するために，プログラム文として表明を記述することができる．言語処理系によって多少異なる場合もあるが，表

明文 `assert(expr)` は，次のような意味を持つ．

expr = true　→　何もしない
expr = false　→　例外を送出する

表明には，開発者のためのドキュメントとしての役割もある．開発者は，ソフトウェア部品の個々の事前条件，事後条件，手続き途中に配置された表明文を見ることで，その部品を使うために必要な条件，機能の特性，変更する際の注意点などを知ることができる．ソフトウェア部品の使用者がその部品の実装詳細へ過度に依存することはソフトウェアの変更容易性を損なう原因となるが，それを防ぐために表明は有効である [13]．また，eXtreme Programming[3] をはじめとするアジャイル型の開発プロセスでは，表明を用いてプログラムの動作が正しいことをテストするといった形で表明を活用している．

契約による設計を言語要素として積極的に取り込んだものとしては Eiffel[14] がある．オブジェクト指向プログラミング言語である Eiffel では，クラスのメソッド単位での事前条件および事後条件を，メソッド本体とは区別して記述することが可能である．また，各メソッドに共通な事前条件，事後条件をクラス不変条件として独立させて記述できるほか，例外 (*Exception*) を，「事前条件が満たされているにもかかわらず事後条件が満たせない状況」と規定している．

C++ や Java のように言語要素として契約による設計を取り入れていないプログラミング言語でも，プログラム文として表明を記述するための表明文が用意されている．また，これらのプログラミング言語において，事前条件，事後条件の記述を可能にするツールとして Larch[8]，iContract[9], jContractor[11], JML[10] など，様々なツールが研究，開発されている．その一方で，設計段階で契約による設計を利用するため， UML 上でオブジェクトの制約を記述するための言語 OCL なども提案 されている [18]．

2.2　表明の制約

表明（事前条件，事後条件を含む）の記述はオブジェクトの再利用の安全性を高めるが，それと同時に，次のような問題がある．

(1) 過度の表明が，再利用の柔軟性を低下させることがある．
(2) メソッド呼び出し側に対する仮定が記述できない．
(3) 複数オブジェクトの一連の動作に対する条件が記述できない．

これらの問題をもたらした原因としては，表明によって言及可能なのは，それが帰属するオブジェクトおよびそのオブジェクトからアクセス可能な範囲に限られることが挙げられる．これらの問題は，具体的には以下のような制約として表れる．

(1) 再利用の柔軟性: 表明は，他のオブジェクトに依存した条件を持っている場合がある．
(2) 呼び出し側に対する仮定: 制御フロー（呼び出し側オブジェクト）に関する情報を表明として扱えない．
(3) オブジェクトを横断した表明の条件: 複数のオブジェクトから構成された条件は，オブジェクトに帰属させることが困難な場合がある．

以降，再利用の柔軟性については 2.3 で，呼び出し側に対する仮定は 2.4 で，複数のオブジェクトを横断した条件については 2.5 で述べる．

2.3　再利用の柔軟性

表明はソフトウェア部品の品質や理解容易性に寄与する一方で，プロジェクトごとの要因によって一部の表明が冗長になる場合や，過度に課せられた条件が再利用を阻害する場合がある．ソフトウェア部品についての表明は，次の二つに分類される．

- アプリケーション（利用者）依存の条件．すなわち，ソフトウェア部品が提供している機能のうち，アプリケーションが必要とする機能だけを提供するための条件．
- ソフトウェア部品（提供者）依存の条件．すなわち，ソフトウェア部品が機能を提供するために使用しているアルゴリズムの制約などから生じる条件．

たとえば，任意のクラスのオブジェクトを格納できるような List クラスの利用者が，実際には String クラスのオブジェクトだけを格納したいという要求を持っている場合がある．このとき，任意のオブジェクトを格納できるというのは List の持

つ条件であり，String だけを格納するというのはアプリケーションの持つ条件である．ここで，List を作成した開発者が，設計や実装において使用する型を限定したり，表明によって String だけを格納することを選ぶと，アプリケーションの安全性は向上する．しかし，将来の再利用において，String 以外のクラスに List を適用しようとするときには障害となる．このような，アプリケーション側の制約については，クラス本来の制約とは分離しておくことが望ましい．

クラスの再利用は，しばしば継承という形で行われる．このとき，導出されたクラスのオブジェクトは，基本クラスのオブジェクトの代替として振舞うことが求められる．この条件のことを，Behavioral Subtyping と呼ぶ．Behavioral Subtyping を満たすためには，導出されたクラスは，基本クラスの事前条件が満たされた状態ならば必ず動作し，かつ基本クラスが保証している事後条件は少なくとも満たさなければならない [5]．この関係を，図 1 に示すように事前条件 (require) と事後条件 (ensure) の 2 つの軸で見ると，事前条件を弱めること，事後条件を強めることは許されるが，特定の環境に特化する，つまり事前条件を厳しくするような実装は許されない．たとえば，あるクラスを基に，特定の環境に限定して特化したい場合，リファクタリング手法「インタフェースの導出」と「メソッドの引き上げ」[6] を適用することで，基となったクラスと新しく作成するクラスの共通部分のインタフェースを新しく構築し，Behavioral Subtyping を維持する必要がある．しかし，基本となるクラスが書き換え可能でない場合には対応できず，また，書き換え可能である場合でも，元のインタフェースを使用していた部分を書き換える必要があり，かなりの手間となってしまう [17]．このような場合，既存の表明が必要に応じて解除できることが重要となる．

これらの問題に対処するため，本稿ではアスペクトを用いた表明の記述を提案する．表明は，検査が成功している限り対象プログラムの振る舞いには影響を与えないことから，開発しているソフトウェア部品から容易に切り離せるようにアスペクトして実装することが適当である．表明をアスペクトとして本来のクラスから分離することで，表

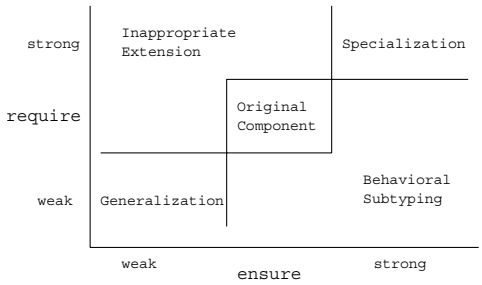

図 1: 表明による拡張の制約

明を目的に応じて分類して記述するといった管理も可能となる．詳細については，3. で述べる．

2.4 呼び出し側に対する仮定の記述

開発者は，作成するメソッドが特定のオブジェクトに利用されることを想定して実装を行っている場合がある．このとき，呼び出しを行うオブジェクトに対する暗黙の情報が，インタフェースに埋め込まれている場合がある．たとえば，平方根を整数に対して求める関数を作成する際，呼び出し側の関数が必ず自然数しか扱っていないと決まっているのであれば，開発者は，入力値を調べるコードを記述するとは限らない．

また，複数のオブジェクトが連携するとき，ある特定のオブジェクトからの呼び出しにのみ処理を許したい場合がある．たとえば，デザインパターンのひとつである facade パターン [7] は，あるサブシステムを構成するオブジェクト群へのアクセスを集中管理する単一の facade オブジェクトを導入する．これによって，オブジェクト群の取り扱いを簡単なインタフェースだけを通じて操作することができる．このような場合，facade 経由でのみアクセスを許可し，それ以外のアクセスを禁止するような表明が考えられる．このようなアクセス制限は，Delegation Layer[15] のようなラッパーオブジェクトを利用する場合にも生じる．

その他にも，呼び出し側の状態に依存した条件といった記述は，現状の表明記述言語では記述することができない．これは，従来，表明がコンポーネント（呼び出される側）のみに依存し，呼び出し側に対する条件を書けないためと考えられる．

本稿では，この問題を解決するために，制御フローに関する表明の記述を導入する．詳細については，3. で述べる．

2.5 オブジェクトを横断した表明の条件

複数のオブジェクトから構成された条件は，単一のオブジェクトに帰属させるのは困難な場合がある．

例として，Model と View のオブジェクト間の相互参照が１対１に対応している，という条件を考える．このとき，新しい View に変更するといった更新作業を行うと，その作業途中で一時的に表明が崩れる場合がある．このような処理の途中では，現在が更新作業の途中であるから相互参照に関する表明の検査はしない，という情報をどこかに記憶しておく必要がある．しかし，どのインスタンスが更新中であるかという情報を Model 側と View 側に分散させると，一貫性のある状態の保持が難しくなる．

このようなオブジェクト間の関連に対する表明や，その実現に必要なフラグなどをアスペクトに分離しておくことで，関連しあったオブジェクト単体での再利用性や，変更容易性を向上できると考えられる．

3 横断的な表明の記述

本章では，表明をどのようにアスペクトに分離するか，また制御フローをどのように表明として記述するか，という方法について議論する．

表明文は，クラスの開発者が，プログラム文としてメソッド中の任意の位置に記述する．Java であれば `assert` キーワードが，他の言語でも類似の言語要素あるいは関数などによって実現されている．

事前条件および事後条件は一般に，メソッド実行の前後に検査される表明文をアスペクトを用いて記述することで実現可能である．従って，以降，表明はすべて表明文によって記述するものとし，表明文の記述法のみについて議論する．参考までに，事前条件を AspectJ で記述した例を図 3 に示す．

表明文は，アルゴリズムの正しさなどを特定の実行時点で検査するために記述するが，AspectJ で採用されているようなメソッド呼び出しなどを単位とした Join Point 記述ではループの先頭や条件文の終了時といった実行時点に適用しにくい．そこで，表明文そのものは，従来と同様の形式で記述を行うが，表明文に用いる真偽値を返すようなメソッドを作成し，そのメソッドにアスペクトを関連付けることで，このような実行時点に適用可能な表明文を実現する．

本稿では，プログラミング言語要素としての型チェックなどの議論については省略し，どのような記述を目標としているかの基本的なアイディアについて説明する．

まず，クラス側の記述は，従来と同様である．表明専用の関数を呼び出すような記述を，表明文として記述する．

```
// f is a valid file stream.
assert(FileIsOpened(f));
```

上の文で FileIsOpened(f) として呼んでいる真偽判定の実装を，アスペクト側で行う．

```
aspect FileAssertion {
  ASSERT FileIsOpened(FileInputStream f)
    { return f.ready(); }
}
```

便宜上，`ASSERT` という修飾子を記述しているが，本体は，真偽値を返すメソッドとして実装される．`ASSERT` と通常のメソッドとの違いは，戻り値が真偽値に限定されていることと，本体の定義が複数あるかもしれないこと，である．

実行時には，クラス開発者が準備した assert という実行時点で連動するアスペクトが条件の評価を行い，すべての条件が通過したかどうかをテストする．

この方法の有利な点は，アスペクト実装者など，他のクラス開発者が，後から表明を追加できることである．

3.1 制御フローに関する表明

メソッドの再帰呼び出し時は，一部の不変条件が崩れている可能性がある．たとえば，不変条件が崩れた状態から復帰する作業に必要なメソッド呼び出しなどが該当する．

制御フローに対応した表明記述を可能にすることで，そのような条件に対処する．そのために，AspectJ の pointcut 指定子である cflow を論理式として使用する．

```
// Precondition: f is a valid input stream.
before(FileInputStream f) : this(f) && execution(* MyFileReader.read(..)) {
  assert(f.ready());
}
```

図 2: assert 文と AspectJ を用いた事前条件の記述例

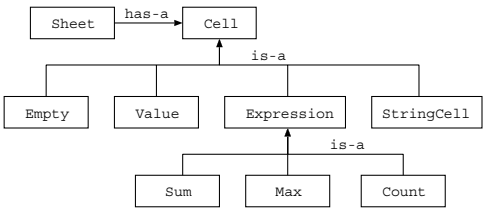

図 3: 表計算プログラムのクラス構成

```
assert ( cflow (aClass.aMethod) );
assert ( cflow (obj.aMethod) );
```

この表明文は，引数として指定されたシグネチャ aClass.aMethod が呼び出しスタック上に積まれているかどうかを真偽値として返す．obj.aMethod と記述した表明は，クラスの部分を特定のオブジェクトのみに限定した場合である．これによって，特定のオブジェクト，メソッドから呼ばれているか否かを判定することができる．基本的には，事前条件として記述することで，特定の制御フローの下でのみ条件を検査する，といった用法を想定している．

4 例題と評価

例題として，簡単な表計算アプリケーションを用いる．主要な構成要素は Sheet，Cell という 2 種類のクラスである．Sheet は表であり，Cell がその構成要素である．Cell 自体は抽象クラスであり，Empty（空），Value（整数値），StringCell（文字列），Expression（式）のいずれかを格納する．Expression は，ある一定範囲の有効な（Empty でない）Cell の個数を数える Count と，ある一定範囲の Value の値の和を計算する Sum，最大値を求める Max がある．クラスの構成を図 3 に示す．

カンマ区切りで記述されたテキスト形式のファイルから表の各セルの状態を読み込み，式の値を計算し，結果を出力する処理を考える．このとき，Sum オブジェクトは，他のセルの値を調べにいくことになる．調べるセルが他の Sum オブジェクトや Max オブジェクトであれば，先にそのセルの値を評価する必要があるため，再帰的に処理が進行する．セル間に循環参照があった場合，値決定の処理が無限ループに陥る．循環参照に陥っていないかどうか，表明を用いて検査を行う．

図 4 に，循環参照となるような呼び出しはエラーとするような表明を，値評価メソッド value() の中に直接記述したものを示す．

この表明文を，AspectJ を用いて記述しようとすると，図 5 のように新たな例外クラスや，フラグを制御するコードを記述しなければならないことがわかる．従来の Java のみで実現しようとすると，AspectJ で作成した記述が，実際に関連した各クラスに分散することになる．

5 関連研究

5.1 アスペクト指向プログラミング

アスペクト指向プログラミングは，横断的関心事の分離を基本的な考え方としている．複数のオブジェクトを横断した表明をアスペクトとして記述することが自然であると考えられる．AspectJ を用いると，容易に事前条件と事後条件を記述することができる．しかし，プログラム文の特定の位置で表明を検査するという場合には，一般的な Join Point として記述することは難しい．本稿で提案した手法を用いれば，表明を検査するメソッドの本体を，アスペクト側から改変することで，クラス側に記述した表明は簡潔に済ませながら，アスペクト側にコンテキストに応じた表明を分離して記述することができる．

5.2 Java Modeling Language (JML)

Java Modeling Language (JML) [10] は，Java クラスの振る舞い仕様を記述するための言語である．JML の記述は Java ソースコードのコメントに埋め込まれる．JML では，オブジェクトの状態を表現するモデル変数を使って事前条件，事後条件を記述することができる．その他にも，事後条件を満たせない例外発生時の状態についても記述

```
class Cell {
  public abstract int value();
  :
}
class Expression extends Cell {
  :
}
class Sum extends Expression {

  // return Iterator for SUM target cells.
  Iterator getRangeIterator() {
    :
  }

  // return the sum of cells' value.
  public int value() {
    assert(!cflow(int this.value())); // prohibit recursive call
    int sum = 0;
    for (Iterator it=getRangeIterator(); it.hasNext(); ) {
       sum += ((Cell)it.next()).value();
    }
    return sum;
  }
  :
}
```

図 4: 循環参照を防ぐ表明を提案手法で記述した例

```
class AssertionViolation extends RuntimeException {}
pointcut execValue(Expression e): this(e) &&
                                  execution(int Expression.value());
boolean Expression.now_executing = false;
Object around(Expression e): execValue(e) {
  if (e.now_executing) throw
    new AssertionViolation("Cyclic Reference is Detected !");
  e.now_executing = true;
  Object o = proceed(c);
  e.now_executing = false;
  return o;
}
```

図 5: 循環参照を防ぐ表明を AspectJ で記述した例

することができ，柔軟性の高い記述が可能である [4]．しかし，複数のオブジェクトが連動した場合に,「誰の」不変条件であるかが明確でない場合にどのように記述するか，ガイドラインが存在しない．

JML を AspectJ 向けに拡張することも提案されている [19] が，AspectJ によって導入された cflow などのいくつかの要素をサポートしていない．そのため，今回提案したようなオブジェクト間の関連についての表明を記述することには不向きである．

6 まとめと今後の課題

オブジェクトを横断した表明に対して，現在利用可能な記述法では，直接的な記述をすることはできない．解決策として，アスペクトを用いた表明の付加と，制御フローを用いた表明の記述方法を提案した．提案した言語要素については，AspectJ に変換するプリプロセッサとして実現する予定である．ビルドツールである Ant [1] と連携することでプリプロセッサを簡単に開発プロセスに組み込めるようにすることを考えている．

課題としては，表明を定義している真偽値を返すメソッドは副作用を持つべきではないが，それを強制する手段がないという問題がある．C++ で用いる const キーワードのような，副作用がないことを開発者が言明する形式を用いることで，表明として使用されているメソッドが副作用を持たないと宣言されていない場合に警告を出すといった方法によって，問題を緩和することが可能であると考えられる．

参考文献

[1] The Apache Ant Project. http://ant.apache.org/

[2] The AspectJ Team: *The AspectJ Programming Guide*.

http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/

[3] Beck, K.: *Extreme Programming Explained*. Addison-Wesley Pub Co., Boston, MA, 1999.

[4] Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G. T., Rustan, K., Leino, M. and Poll, E.: An overview of JML tools and applications. *In Proc. of 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, pp. 73-89, Vol. 80 of Electronic Notes in Theoretical Computer Science, 2003.

[5] Findler, R. B., Latendresse, M. and Felleisen, M.: Behavioral Contracts and Behavioral Subtyping. *In Proc. of the 9th Foundations of Software Engineering (FSE2001)*, 2001.

[6] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Pub Co., Boston, MA, 1999.

[7] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns*. Addison-Wesley Pub Co., Boston, MA, 1995.

[8] Guttag, J. V., Horning, J. J. and Wing, J. M.: The Larch Family of Specification Languages. *IEEE Software*, 2(5), p.24-36, Sept.1985.

[9] iContract HomePage. http://www.reliable-systems.com/tools/iContract/iContract.htm

[10] JML Reference, http://www.dc.fi.udc.es/ai/tp/jml/JML/docs/jmlrefman/jmlrefman/

[11] Karaorman, M., Holze, U. and Bruno, J.: jContractor: A Reflective Java Library to Support Design By Contract, Technical Report, Univ. California, Santa Barbara, Department of Computer Science (http://www.cs.ucsb.edu/labs/oocsb/papers.html).

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming. *In Proc. of the 11th annual European Conference for Object-Oriented Programming (ECOOP97)*, vol.1241 of LNCS, pp.220-242, 1997.

[13] McCamant, S. and Ernst, M. D.: Predicting Problems Caused by Component Upgrades, *In Proc. of the 11th Foundations of Software Engineering (FSE 2003)*, pp.287-296, 2003.

[14] Meyer, B.: *Object Oriented Software Construction*. Prentice Hall, New York, NY, 1988.

[15] Ostermann, K.: Dynamically Composable Collaborations with Delegation Layers. *In Proc. of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, pp.89-110, 2002.

[16] Romanovsky, A.: Exception Handling in Component-based System Development. *In Proc. of the 25th Annual International Computer Software and Application Conference (COMPSAC 2001)*, pp. 580-586, 2001

[17] Tip, F., Kiezun, A. and Baumer, D.: Refactoring for Generalization using Type Constraints. *In Proc. of 18th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, pp.13-26, 2003.

[18] Warmer, J. and Kleppe, A.: *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Pub Co., Boston, MA, 1998.

[19] Zhao, J. and Rinard, M.: Pipa: A Behavioral Interface Specification Language for AspectJ. *In Proc. of Fundamental Approaches to Software Engineering (FASE) 2003*, pp.150-165, 2003.