社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2003－SE－143　(3)
2003／7／17

副作用の概念を導入したオブジェクト指向分析設計モデルに関する考察

久米 出

概要: オブジェクト指向ソフトウェア開発に於いて副作用が引き起こす問題を説明し、設計手法に基いて問題解決のための基本的な事柄を論ずる。10 万行規模で構造が複雑な CASE ツールを例題として用い、副作用の性質と設計モデル上で表現するうえでの現実的な観点からの課題を考察している。

Introducing Side Effects in Object-Oriented Analysis and Design

Kume Izuru

abstract: We explain the problem caused by side effects in object-oriented software development, and discuss some basic issues toward the solution by our design method. Our example is based on a real CASE tool with complex structure and 100,000 lines of a sorce code, which tells us the nature of side effects and realistic problems to represent side effects in a design model.

1　はじめに

本論文はソフトウェアの構成要素間の副作用を介した依存関係を、**副作用依存**と呼び、副作用依存がソフトウェア開発に引き起こす問題を提起する。さらに依存関係を設計モデル上で明示的に表現する必要性述べ、表現の際に障害となる要因について説明する。副作用を介した依存関係を難しくする要因の一つがオブジェクトの別名 (alisa) である。別名の問題を解決するためのオブジェクト隠蔽手法は 90 年代から研究が進められている。[18, 19, 4, 8, 9, 3, 1, 7] 本研究は従来の手法とは方針からして異なる解決手法を提案するものである。我々は従来の手法は現実のソフトウェア開発に応用するには方針からして無理があると考えており、それを説明するために 10 万行程度の規模の実在する CASE ツールを題材とし、副作用依存がソフトウェアの変更に対してどのように障害として働くかを説明する。以下では対象となるソフトウェアの概略を紹介し、副作用依存がどのような形で発現するのかを示す。

CASE ツールはモデル要素とモデル要素を表現する図形を編集の対象とする。図形は図 (diagram) 上に配置されて表示される。図は設計モデルの一部を視覚的に表現したものであるため、モデルは必ず図の内容を充足しなければならない。言い変えれば図上で表現されているモデル要素やモデル要素間の関係は必ずモデル上に存在していなければならない。一方で但しモデルの内容が全て図上で表現されているとは限らないことに注意して欲しい。モデルが任意の図を充足するためには、図の編集とモデル上の変更の整合性が保たれていなければならない。追加に関してはそれほど問題は無いが、削除に際しては注意が必要である。モデル要素の中には仕様上、自らの存在を他者に依存しているものがある。例えばクラス間を結ぶ関連などがそうした性質を持っている。よってあるモデル要素を削除する場合には、そのモデル要素に依存しているモデル要素も削除する必要がある。

ここで編集作業の取り消し (undo) 機能を実現するためには、まず編集を実行する際に生じた各変更に対して、それを取り消す undo コマンドを用意しておく必要がある。取り消しをする場合にはそれを実行するのであるが、ここでどの undo コマンドをどのような順番で実行するか問題になる。最も分かり易くて安全なのは undo コマンドを一本のスタックに積んでおいて、取り消しの際には上から順番に実行するやり方である。ユーザの各編集行為とそれに伴って積まれた undo コマンドの対応が正しければ、完全に元の状態に復帰することが可能である。

上の単純スタック型の欠点は取り消したくない変更も取り消してしまうことである。ソフトウェアの設計は複数の図によって表現されるが、各図の内容はそれぞれ独立していることが珍しくない。ここである特定の図の変更だけを取り消したいとする。その図の変更の後に別の図を変更している場合には、そちらの変更内容も取り消されてしまう。こうした問題を改善するためには、この図に表現された変更のみを取り消すような実装を施す必要がある。もちろん、この方針に従えばモデルは元通りには復元されないが、それは已むを得ないものとする。モデルの復元は図の内容を充足するに足りる最小限の範囲に留めることになる。こうした取り消し機能を本論文では表現駆動型取り消し機能と呼ぶ。

取り消しによって図形に変更が加えられた場合、モデル内でも対応して変更を行う必要がある。表現駆動型取り消し機能の場合には、単純スタック型のそれと異なり、変更された図形と関係無い変更の取り消しをなるべくしないようにしなければならない。 取り消し内容の選択が正しく無いような場合には例えば図 1 で述べるような現象が生じでしまう。クラス図 1 にクラスを表

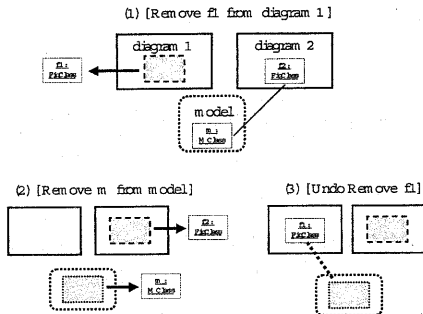

図 1: モデル要素の復活を怠ったことによるバグ

現する図形が表示されているとする。Cut&Paste を用いてクラス図 1 からクラスの図形を消去し、クラス図 2 に貼り付けている。次にクラス図 2 で対応するモデル要素をモデルから消去する。最後にクラス図 1 に対して Cut を取り消す作業を施す。この時二番目の「モデルからの消去」に対する取り消しを忘れてしまうと、モデル中に存在しないはずのクラス画像がクラス図上に表示されてしまうことになる。

こうした結果を引き起こす原因として

- 「モデルからの削除」の効果を打ち消すように期待されたコードの内容が正しくなかった
- 「モデルからの削除」の効果を打ち消すように期待されたコードがそもそも適用されなかった

のどちらかが考えられる。後者のタイプの原因が我々の研究対象である。取り消し機能は過去の編集の結果に基いて実際に適用すべきコードを選択しているはずである。これは取り消し機能は編集機能に対して副作用依存の関係にあることを意味する。正しいコードが適用されていないのであれば、それは副作用依存の関係が正しく実装されていないのであると言える。こうした問題を防ぐためには、まず副作用依存の関係を明らかにする必要がある。後に述べるように、明示化されていない副作用依存をソースコードから発見するのは非常に困難である。副作用依存が原因である場合にも障害が表面化するのは別の箇所である可能性が高い。[10] 上の例で言えば、編集結果をファイルに保存しようとして失敗するまで CASE ツールは表面上問題無く動くかもしれない。

副作用依存は設計の段階から明示化されるべきである、というのが我々の立場である。本論分の目的は副作用依存を設計モデルで表現すべき事柄を明らかにし、表現をする上で障害となる要因を現実的な見地から考察することにある。従来の研究では設計モデル上でオブジェクトの参照を局所化する設計を施すことによって別名問題を解決し、ひいては副作用の問題が解決できると考えられている。我々は参照の局所化は極めて特殊な設計を必要とするため、現実の問題に適用するのは難しいのではないかと思っている。我々の主張や従来の手法による問題点を明らかにするためには、実在するソフトウェアから具体的な例題を作成するのが有効であると考えられる。よって次節では例題に関連する箇所をもう少し詳細に説明し、後の部分の準備とする。

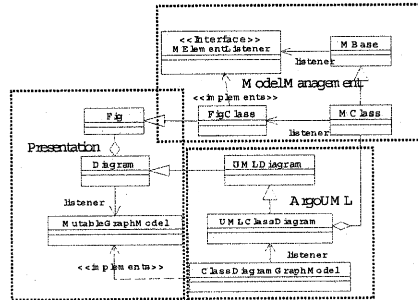

図 2: フレームワーク間の仲立ち

2 ArgoUML

2.1 アーキテクチャ

本節ではフリーな CASE ツール、ArgoUML の設計を説明し、表現駆動型の取り消し機能の設計方針を述べる。[1] 以下の設計は公開されたソースコードを元にしたものである。ArgoUML は UML を用いたソフトウェア開発を支援するツールである。実装言語は Java でコードの総行数は 10 万行[2]に達している。二つの既存のアプリケーションフレームワークを用いて、図形表現とモデル管理を実現し、自らは両者の仲立ちを行っている。(図 2)

表現フレームワークで重要なクラスは図形と図を実装する `Fig` クラスと `Diagram` クラスである。`Fig` オブジェクトに対する変更は、事前に登録された `MutableGraphModel` インタフェースのインスタンスに伝えられる。モデル管理フレームワークは UML のモデル要素を実装している。最も重要なクラスはモデル要素を実装する全てのクラスが継承している `MBase` である。`MBase` は後に説明するように取り消し操作の実装でも重要な役割を果たている。モデル要素上の変更は事前に登録された、`MElementListener` インタフェースを持つオブジェクトに伝達される。`MClass` は UML のメタクラス **Class** を実装しているクラスである。

ArgoUML は図とモデルの変更イベントを処理するためのクラスを実装する。図 2 で示されているのはクラス

[1] ArgoUML の最新バージョンでは取り消し機能は実装されていない。現在開発者用の ML で単純スタック型の取り消し機能の設計に関する議論が行われている段階である。後述するモデル要素に対する取り消し機能は実際に実装されている。

[2] 利用している外部パッケージを含む

図を実現するために必要なクラスである。クラス図とクラスを表現する図形はそれぞれ Fig と Diagram を継承した FigClass と UMLClassDiagram によって実装される。クラス図上の図形の変化は ClassDiagramGraphModel を通じて UMLClassDiagram に伝えられる。クラス図のモデル要素上の変化は ArgoUML が用意した Fig の下位クラスに伝えられる。図 2 からメタクラス Class には FigClass がその働きをしていることが分かる。UMLClassDiagram の親クラス UMLDiagram は一般的な図の機能を実装する。表現フレームワークとモデル管理フレームワークの上記の対応はこのクラスで実装されている。

2.2 イベントとモデル管理

設計の方針を決定するためには、全体のおおまかなイベントの流れと取り消し機能に関してもう少し詳しい調査が必要である。ArgoUML の実装ではユーザの図形に対する編集行為は、その図形の生成以外は全て一旦モデル要素に対する変更に解釈される。モデル要素の変更は図 3 が示すようにイベント (MElementEvent) の形で表現フレームワークに伝えられる。例としてクラス図上で表示されている FigClass に対して「モデルからの削除」を実行した場合を考えてみよう。ユーザからの入力はクラス図 (UMLClassDiagram) の上位クラス Diagram のメソッドによってイベントに変換される。UMLClassDiagram は ClassDiagramGraphModel 経由でこのイベントを受け取る。イベントの内容から FigClass に対する操作であることが判明する。そこで FigClass が表現しているモデル要素、MClass オブジェクトを取得して削除メソッドを実行する。これで MClass オブジェクトはモデル中から削除される。削除の内容は以下の通りになっている。

- 他のモデル要素間の参照関係の削除の実行と削除を通知するイベントの生成
- モデル要素自身の削除を通知するイベントの生成
- モデル要素に存在を依存している他のモデル要素に対する削除メソッドの呼び出し

各モデル要素の状態やモデル要素間の関係を変更するようなメソッドが実行される時には、その変更を取り消すような undo コマンドが作成される。上で説明した削除の例では MClass オブジェクトに対して以下の二つを実行するような複数の undo コマンドが生成される。

- 各参照関係の復活と復活を通知するイベントの作成
- モデル要素の復活を通知するイベントの作成

図 3: undo 機能

イベントの送付と生成された undo コマンドの登録は MFactory の唯一のインスタンスが実行する。undo コマンドは MCheckPointUndoManager にスタックの形で保存される。取り消しを行う場合にはスタックの先頭から順に undo コマンドが実行される。

2.3 表現駆動型取り消し機能の設計

以上の考察を元に表現駆動型取り消し機能の設計を説明する。(図 4) まず MFactory と MElementEvent を一部改良して MElementEvent から MethodUndoAction を参照できるようにする。これによってモデル要素の変更イベントを経由して、undo コマンドを参照することが可能になった。UMLDiagram は UML の様々なモデル図の親クラスであり、図上の図形とモデル要素の対応を知り得る立場にある。図上の変更を取り消す undo コマンドクラス PresentationUndo をこのクラスに管理させることにする。モデルの変更に伴って図形が変更された場合には PresentationUndo オブジェクトを生成してスタックに積む。さらに初めに生成された MethodUndoAction を参照モデルの変更のチェックポイントとして参照する。PresentationUndo によって図上の取り消しを実行する際にはモデルの取り消しもここまで行うのである。モデルの変更を伴わない図上の変更、例えば線を折線にしたり、配置を変えたり、モデルの表現形式を変えたりしてもモデルの変更取り消しは対応しない。よって図上だけの変更を行った後に、図と関係の無いモデルの変更が生じても、変更取り消し作業によって他の図に影響が出るのを防ぐことが可能である。

これで上手く行くように見えるのだが、実装に関して一つ注意しなければならない点がある。図上の図形に対して「図からの削除」を行うと UMLDiagram はその図形を破棄してしまう。よってそれ以後は別の図上からモデル要素に対して変更を加えらえても察知できなくなってしまう。これによって節 1 で述べたような障害が発生する可能性が生じてしまうのである。次節ではこうした問題を副作用依存という視点から考察する。

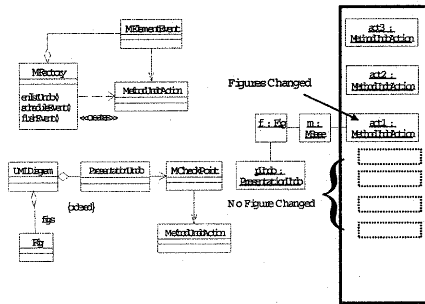

図 4: 取り消し機能の改良

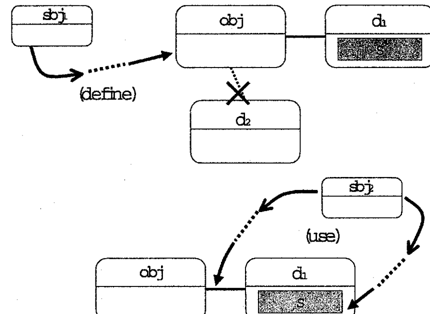

図 5: 副作用依存のモデル

3 問題分析

3.1 副作用依存の性質

ソフトウェアの再利用や保守管理を論じる上でソフトウェアの構成要素間の依存性の分析は大変重要である。通常の登場する依存関係はソースコードや設計モデルから分析できるものが多いが、これはオブジェクト指向モデルを用いた見通しの良さの恩恵と言える。例えばモジュール間の依存関係の強さを表す「強度」という用語[17]を設計モデルから判断するのは必ずしも容易ではなかった。Booch の[2]言うところのオブジェクト指向の基本概念のうち、隠蔽と抽象化を導入することによってRefactoring[5]に見られるように設計の表面に現れている情報のみを用いた判断が可能になったのである。さらに、こうした依存性は広い範囲に波及しないように断ち切ることが難しくない。デザインパターン[6]などはこの性質を利用した再利用設計の成功例である。

一方で副作用は通常、設計モデル上で表現されることは無い。またどの程度広い範囲に依存性が波及しているのかは極端な話、ソースコードを全て読まなければ明らかではない。図5は二つのオブジェクト sbj_1 と sbj_2 の間に副作用依存が形成される様を表している。初めにオブジェクト sbj_1 による副作用として obj と d_2 を結ぶ関係がえ obj と d_1 の関係に変更され、続いて d_1 の状態が S に変更されている。sbj_1 の実行終了後に sbj_2 が obj 経由で d_1 に到達し、d_1 の状態 S の参照を行う。sbj_2 上計算の結果に状態 S のデータが用いられる場合、直前に実行された sbj_1 による副作用が obj と d_1 に残る形で sbj_2 の実行結果に影響を与えたことになる。ここで sbj_2 は obj と d_1 を介して sbj_1 に依存していると言う。sbj_1 を**(副作用の)設定者**、sbj_2 を**(副作用の)利用者**、obj と d_1 を**(副作用の)触媒**と呼ぶ。

副作用依存を完全に解析するためには、

- 実行の結果副作用が残る範囲
- 副作用が残るオブジェクトに対する参照

の二点を明らかにしなければならない。参照関係が一旦設定されたら殆ど変化しないようなソフトウェアならこうした分析も可能かもしれないが、ArgoUML でしばしば用いられているイベントに対する `listener` の設定のように、参照関係が動的に切り替わるような場合にはメッセージの送付先からして判断するのが容易ではない。さらにソフトウェアの見通しを良くするのに貢献した抽象化と隠蔽が、参照関係に基いた依存性を解析する際には障害となるからである。[16]

3.2 設計モデルに於ける表現

ArgoUML では副作用依存性が頻出しているが、ML[3]での議論を見る限りそれが開発の大きな問題にはなっていない。開発者がソースコードの内容を知悉しており、頻繁に情報交換している限りは副作用依存性はあまり問題では無い。しかしながら著者のように、ソフトウェアに対する十分な知識を持たないまま、ソフトウェアの一部のみを変更したい場合には、複雑な副作用依存性は大きな問題となる。開発者達にとっては自分が開発しているコードの副作用が及ぶ範囲 (例えば表示が変更されるGUIや変更されるモデル要素) は明白であり、副作用の結果が正しく残るように注意を払っていればよいのである。よって副作用依存性の出現頻度が高いソフトウェアを保守管理するためには、依存関係の範囲を明確にするような情報がソフトウェアに与えられている必要がある。以下、設計モデルの形で情報を与える場合について考察する。

我々は過去にユースケース[13]によって表現されたシナリオ分析を元にソフトウェアの機能のデータ共有関係を決定し、概念レベルのモデル上の副作用依存関係を表現する研究を進めてきた。[11, 12] ユースケースで記述された機能を実現しているクラスを特定し、そのクラスを呼び出しの頂点とする実行系列下におけるオブジェクトの保護機構を実現することが研究の目的であった。副作用の設定者と利用者は共にユースケースが特定さ

[3] ArgoUML の開発上の打ち合わせは原則として全てオープンな ML 上で行われている。ML の議論は大変活発で一日に何十通ものやりとりが交わされるのも珍しくない

れているクラスである。ここでは媒体は保護下にあるオブジェクトであり設定者と利用者との構造的な関係に基いて特定される。この関係を設計モデルで記述するのである。ユースケースはインクルード関係に基いて階層化されていると考え、その階層を反映する形で保護領域が形成される。全ての実行は必ずあるユースケースの機能を実現するものであり、オブジェクトに対する参照は実行系列のユースケースとオブジェクトを保護しているユースケースの関係によって可否が動的に決定される。ここでユースケースの関係を決定しているのがシナリオレベルでのデータ共有関係である。副作用依存はアプリケーション・ロジックから明白なもの以外は全て発生時にエラーとして検出されるので、副作用依存によって生じるバグの所在が分かりやすくなっている。

しかしながら、この方法には現実のソフトウェア開発に応用する上で重大な問題点が存在する。ソフトウェアのモデルとして分析レベルの抽象度のものを仮定しているため、設計上の記述自身の整合性や分析レベルでは現れない副作用依存性の対処に対する考察が不十分であった。例えば第 2 節の図 4 は `UMLDiagram` と `Fig` の関係が Cut 操作のために変更されてしまっているのがバグの原因となっている。これは Cut 操作の実行の過程で数多く生成される変更の一つであるが、現実問題としてこうした細かい変更まで注意を払って設計することが可能かどうかは分からない。さらに、実はこのクラス図そのものがかなり抽象度の高い記述であり、実際の設計モデルでは同一の媒体が複数の図に寸断されている可能性が高い。副作用依存関係が膨大な数の設計図の群に散らばってしまう恐れが強い。

副作用が多数の設計に散らばって存在する問題は我々にプログラミングにおける “Separation of Concerns” の問題とアスペクト指向プログラミングを [14] の導入を連想させる。アスペクト指向プログラミングの本来の動機は、機能分割とは異なる視点におけるソフトウェアの分割と統合であった。[15] 一般にオブジェクト指向設計では責務 (repsonsibility) と協調 (collaboratoin) を定義することによってシステムを分割していく場合が多い。[20] 副作用依存の場合には設定者、利用者、媒体の役割が割り振られたクラスが分割の対象となる。それぞれのクラスの分割によって副作用の内容が分割されるはずであり、さらには分割と統合に対する整合性が定められるはずである。

4 関連研究

オブジェクトの状態の永続性が副作用として残る際の問題は 1990 年代の初頭に Jhon Hogg によって提起された。[9, 8] Jhon Hogg はその先駆的な論文で island という保護領域を定義して、オブジェクトに対する直接的な参照を規制する手法を提案した。保護領域を実装するために彼は参照の単一性を表現する特殊な型システムを提案し、型付け可能なオブジェクトが参照を規制されていることを保証した。[8] 以後多くの研究がなされてきたが、基本的にはオブジェクトの別名が抑制されるような保護領域の実現を目的としている点で共通している。門番を設定して、外部との直接通信を遮断しているやりかたが多い。

通常は型情報等を用いて保護領域の実現を静的に検証する手法が主流である。[19, 4, 1, 7] James Noble らの研究 [19, 4] はクラスに対して、保護オブジェクトの参照に関する役割を明示的なキーワードを用いて保護領域を設定しようとするものである。 これの手法は保護の対象となるオブジェクトが保護領域間をあまり移動しないような、静的な関係を保っている場合を想定しているように思われる。パラメータ型の導入によって記述の柔軟性に対する工夫が注目される。

Confinement Type[1] は Java の言語機能を利用して保護領域を実現している。保護下にあるオブジェクトの内容を保護領域の外に漏らさないことを目的としているが、保護領域に特定の門番を設定せずにログラミングのルールを定めることによって保護を実現している点に特徴がある。メソッドに対しては自らの内容を外に漏らさないような匿名性 (anonymity) のルールを定め、保護されるべきクラスに対しては封じ込め (confinement) ルールを定めている。Grothoff[7] は既存のクラスを解析して封じ込め属性を付与するためのツールを開発し、実際に実験を行っている。

保護領域を定める手法の最大の問題点は、間接的参照であれば保護下にあるオブジェクトの状態を変えてしまえる可能性が排除し切れない点にある。門番がある場合には門番の仕様によって変更を阻止しやすくなるが、前節で述べたように、設定者、利用者、媒体の組の数だけ対処を行う必要が出てきてしまう。こうした設計が優れているとは思えない。規模の大きいプログラムに対して、どれくらいの規模の保護領域が必要となるかも現実的な見地からも検討すべきである。保護領域の規模があまりにも小さいと、保護そのものの機能を果たすのが難しくなる。一方で保護の規模が大き過ぎる場合には、保護領域の中でさらに保護を考える必要が生じるため、実際には副作用の解決としてはあまり役に立たない可能性も出てくると考えられる。

5 おわりに

本論文はオブジェクト指向ソフトウェア開発における副作用問題を論じた。特に実在する大規模なソフトウェ

アを調査することにより、現実的な見地から問題を解決することを目指した考察を行った。副作用に対する問題意識を共有する研究として、オブジェクトの別名問題の研究が挙げられる。別名問題は理論的な見地から研究が進められている面が強く、現実のソフトウェア開発に対してどのような応用が可能であるのかが明らかにされていない傾向が見受けられる。我々は別名問題ではなく、設計モデル上での副作用の問題として研究を進めた。副作用の問題はある程度の規模のソフトウェアの内部で生じやすいと考えられるため、設計を用いた我々の手法はより現実的な対処が可能であると考えている。

本論文では設計表現と実装の整合性について触れられていない。我々は過去の研究で [12] ユースケースの構造によって挙動が決定するような動的な検証アルゴリズムをインファレンス・ルールの形で説明した。今後の課題としては、設計モデル上でのクラスの分割に対する整合性の保証と検証アルゴリズムの対応を考えている。

参考文献

[1] Boris Bokowski and Jan Vitek. Confined types. In *ACM OOPSLA*, pages 82–95, 1999.

[2] Grady Booch. *Object Oriented Analysis and Design with Applications.* The Benjamin Cummings Publishing Co. Inc., second edition, 1994.

[3] Ciarán Bryce and Chrislain Razafimahefa. An approach to safe object sharing. In *ACM OOPSLA*, pages 367–381, 2000.

[4] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *ACM OOPSLA*, pages 48–64, 1998.

[5] Martin Fowler. *Refactoring.* Addison-Wesley, 1999.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides. *Design Patterns.* ADDISON-WESLEY, 1994.

[7] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *ACM OOPSLA*, pages 241–253, 2001.

[8] John Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM OOPSLA*, pages 271–285, 1991.

[9] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The geneva conversion on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.

[10] Kume Izuru. Semantically consistent system integration on object-oriented databases. In *International Conference on Management of Data*, 1998.

[11] Kume Izuru. A uml-based approach to resolve architectural mismatch between components with respect to shared objects. In *Data Engineering Workshop.* The Institute of Electronics, Information and Communication Engineers, 2000.

[12] Kume Izuru. Object-oriented analysis and design approach for safe object sharing. In *International Conference on Engineering of Complex Computer Systems.* IEEE, 2001.

[13] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* The ACM Press, 1992.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, Lecture Notes in Computer Science, 1997.

[16] Li Li and A. Jefferson Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *International Conference on Software Maintenance*, pages 171–184. IEEE, 1996.

[17] Glenford J. Myers. *Composite/Structured Design.* Van Nostrand Reinhold, 1978.

[18] James Noble and John Potter. Change detection for aggregate objects with aliasing. In *Australian Software Engineering Conference.* IEEE Press, 1997.

[19] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, 1998.

[20] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.