修士学位論文

題目

リファクタリング中に生じるコンパイルエラーの自動解消手法

指導教員
井上 克郎 教授

報告者
譜久島 亮

平成 22 年 2 月 8 日

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 ソフトウェア工学講座

平成 21 年度 修士学位論文

リファクタリング中に生じるコンパイルエラーの自動解消手法

譜久島 亮

内容梗概

ソフトウェアの保守を効率的に行うためにソフトウェアの設計品質を高めるリファクタリングが注目されている．リファクタリングとは，ソースコードの外部的振る舞いを保ちながら，内部構造を改善する作業である．頻繁に行われるリファクタリングとして，メンバの移動が挙げられる．開発者がメンバの移動を行うと，メンバ間の参照・被参照の関係が成立しなくなりコンパイルエラーが生じることがある．コンパイルエラー解消のためには，移動するメンバと参照関係にあるメンバの移動や，メンバのアクセス修飾子の変更，メンバのカプセル化を検討しなければならない．多くの場合メンバの移動により生じるコンパイルエラーを解消するための編集手順は複数存在し，各編集手順を適用した結果のソースコードはそれぞれ異なる．既存のリファクタリング支援機能には，リファクタリング中に生じるコンパイルエラーを自動で解消し，編集手順を提示できるものが確認出来ない．よってリファクタリングの経験が少ない開発者の場合，メンバの移動により生じたコンパイルエラーを解消する編集手順を十分に理解しないままソースコードを編集し，不必要な編集を行ったり，欠陥を作り込んだりする可能性が考えられる．そこで本研究では，メンバの移動を行う際に生じるコンパイルエラーを解消する編集手順を，自動的に導出する手法を提案し，Eclipse プラグインとして実現した．具体的には，メンバの移動により，メンバ間の参照・被参照の関係が成立しなくなるコンパイルエラーを検出し，それを除去するための編集手順を探索的に求める．メンバの移動を適用したソースコードを収集し，提案手法が導出する編集手順と比較して妥当性を評価する．

主な用語

オブジェクト指向プログラミング
リファクタリング
Eclipse プラグイン

目 次

1 まえがき **4**

2 リファクタリング **6**
2.1 メンバの移動 . . . 6
2.2 コンパイルエラーが発生するメンバの移動 . . . 9
2.3 リファクタリング支援機能 . . . 10
2.4 リファクタリング支援機能の問題点 . . . 13

3 提案手法 **15**
3.1 編集ステップ導出のための情報取得 . . . 16
3.2 編集ステップの導出 . . . 17
3.3 編集ステップ導出・適用の繰り返し処理 . . . 18
3.4 適用例 . . . 19

4 実装 **23**
4.1 編集適用部 . . . 23
4.2 コンパイルエラー情報取得部 . . . 25
4.3 編集ステップ導出部 . . . 26
4.3.1 メンバの移動の実装 . . . 27
4.3.2 アクセス修飾子の変更の実装 . . . 27
4.3.3 変数のカプセル化の実装 . . . 27

5 適用実験 **28**
5.1 実験内容 . . . 28
5.2 実験結果 . . . 29
5.3 考察 . . . 30
5.3.1 メンバ間の参照方法の検討 . . . 31
5.3.2 導出する編集ステップの工夫 . . . 31
5.3.3 ソースコードの外部的振る舞い . . . 32
5.3.4 導出されたソースコードの選択 . . . 32

6 関連研究 **33**

7 あとがき **35**

謝辞 **36**

参考文献 **37**

1　まえがき

ソフトウェアの保守を効率的に行うためにソフトウェアの設計品質が重要視されている中，ソフトウェアの設計品質を高める手段としてリファクタリングが注目されている．

リファクタリングとは，ソフトウェアの外部的振る舞いを保ちながら，内部の構造を改善する作業である．メソッドの移動やフィールドの移動（以降，メンバの移動）など，繰り返し行われるリファクタリングは，パターンとして書籍やウェブサイトにまとめられている．このようなパターンはリファクタリングパターンと呼ばれ，リファクタリングを適用するべきソースコードの特徴や，その特徴によって決まるソースコードの編集手順がまとめられているものである．例えば，Fowler のウェブサイト [8] には 93 種類のリファクタリングパターンが掲載されている．リファクタリングの経験が少ない開発者は，このようなリファクタリングパターンから熟練者が行うリファクタリングを学習し，ソースコードに適用することができる．

リファクタリングパターンの中で開発者が頻繁に行うものとして，メンバの移動 [10] が挙げられる．このメンバの移動では，移動するメンバが参照しているメンバの移動や，移動後のメンバへの参照方法を検討する必要があり，リファクタリングを行う開発者の意図によって，ソースコードに対する編集手順が複数考えられる．

メンバの移動後，private なメンバの移動や，private なメンバを参照しているメンバを移動した場合，メンバの移動により，メンバ間で参照・被参照の関係が成立しなくなる．このような場合に発生するコンパイルエラーを解消するためには，上記のような移動したメンバと参照関係にあるメンバの移動や，アクセス修飾子の変更，メンバのカプセル化など，開発者の意図によって複数の編集手順が考えられる．

このようにソースコードに対して様々な検討が必要な編集作業は，開発者にとって負担となる．そこで統合開発環境 Eclipse ではリファクタリング支援機能が提供されている．メンバの移動をリファクタリング支援機能で行う際，開発者は移動するメンバや移動先クラス等の情報を入力するだけで指定したクラスにメンバを移動することができる．しかし，メンバの移動によりコンパイルエラーが発生する場合，リファクタリング支援機能は，移動するメンバと参照関係にあるメンバの移動や，メンバのカプセル化等の複数の編集を提示せずにメンバのアクセス修飾子を public または protected に変更する編集を提示してしまう．

リファクタリング経験の豊富な開発者であれば，メンバの移動後のコンパイル可能なソースコードを理解し，それを得るための編集手順を導出することは可能であるが，リファクタリング経験の少ない開発者は，メンバの移動後のコンパイル可能なソースコードを理解できないまま編集作業を行う可能性がある．そのために，どのような編集手順が考えられるかを十分に理解しないままソースコードを編集し，不必要な編集を行ったり，欠陥を作り込んだ

りする可能性が考えられる．

そこで本研究では, Java 言語のおけるメンバの移動に着目し，１つのメンバを他のクラスへ移動する際に生じるコンパイルエラーを解消する複数の編集手順を導出する手法を提案する．

提案手法は，対象とするメンバを他のクラスへ移動した際に，メンバ間の参照・被参照の関係が成立しなくなることが原因で発生するコンパイルエラーを検出し，そのようなコンパイルエラーを除去するための編集手順を探索的に求めることで，適用可能な編集手順の候補や，リファクタリング後のソースコードの候補を導出する．提案手法に基づいて private メンバの移動を支援する Eclipse プラグイン [1] を作成し，また，提案手法で導出されたソースコードの候補の妥当性を検証するために，被験者が手作業で行う，メンバの移動により生じたコンパイルエラーを解消する編集手順を調べ，提案手法で導出する編集の妥当性を述べた．また具体的なメンバの移動例を用いて，提案手法によりメンバの移動により生じたコンパイルエラーをを解消する編集手順を導出した．

以下，2 章では，リファクタリング支援の問題点について述べ，3 章で提案手法と適用例について説明する．4 章では提案手法の実装概要について述べ，5 章で提案手法の妥当性を検証する．6 章で関連研究について述べ，7 章でまとめと今後の課題について述べる．

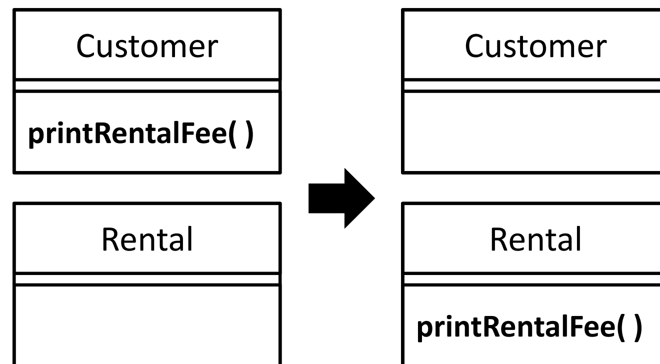

図 1: メンバの移動の例

2　リファクタリング

リファクタリングとは，ソフトウェアの外部的振る舞いを保ちながら内部構造を改善する作業である [2, 13]．文献 [2] では，典型的なリファクタリングがリファクタリングパターン（以降，パターン）としてまとめられている．ここでは，“メンバの移動” パターンに注目して，パターンに基づく編集作業や編集作業を支援するツールの問題点を述べる．

2.1　メンバの移動

メンバの移動は，あるメンバを適切なクラスへ移動することで，クラスの責務を明確にするリファクタリングである．例えば，図 1 では，Customer.printRentalFee() を，Rental へ移動している．

メンバの移動が行われる場合は以下のものが挙げらる．

- あるメンバが，現在所属しているクラスより他のクラスから参照されることが多い
- あるメンバが，将来的に他クラスから参照されることが多くなる
- クラスに所属するメンバ数が増えてきたため，新しいクラスを作成しメンバを移動させ，クラスの機能を分割する（クラスの抽出）
- クラスに所属するメンバ数が少なくクラスの機能が少ないため，既存のクラスへメンバを移動させ，クラスを併合する（クラスのインライン化）

このように様々な目的で行われるメンバの移動リファクタリングは，開発者が頻繁に行うリファクタリングとして知られている [10]．メンバの移動を行う際には，ソースコードに対して以下の検討を行う必要がある．

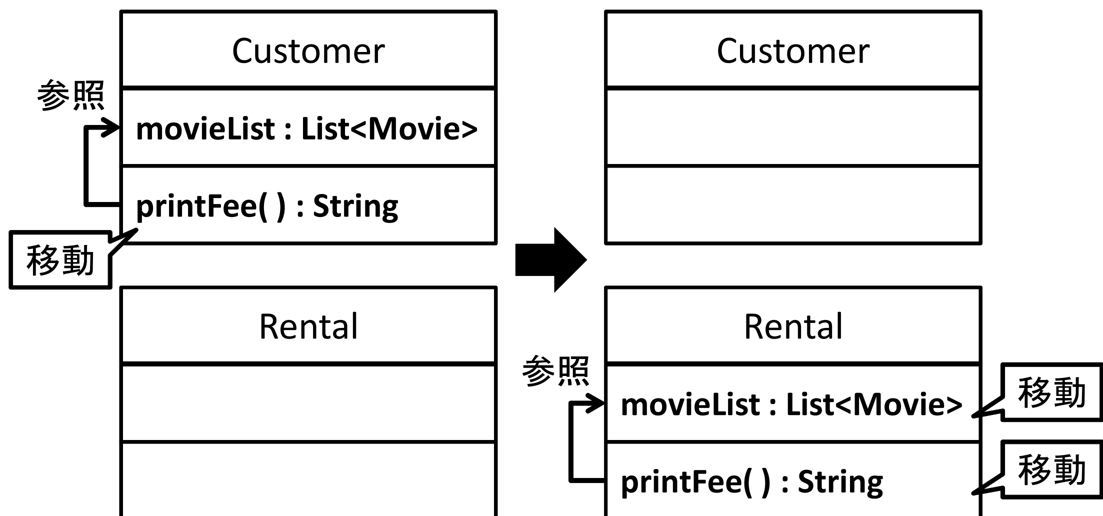

図 2: メンバの移動において，移動するメンバが参照するメンバも同時に移動する例

検討１ 移動するメンバが参照している，または移動するメンバを参照しているクラス内のメンバの移動を検討する．

検討２ メンバを移動先クラスへ移動後，移動元クラスと移動先クラスの間でメンバ間の参照が可能になるような参照方法を検討する．以下のような参照方法が考えられる．

1. 移動元クラス内（移動先クラス内）のフィールドから移動先クラス型（移動元クラス型）のインスタンス変数を取得し，それを介して移動先クラス（移動元クラス）のメンバを参照する．インスタンス変数が取得できない場合は新たに追加する．
2. 移動するメンバがメソッドの場合，移動するメソッドの引数に移動先クラス型の引数を追加し，引数を介して移動先クラスから移動元クラスを参照する

検討１について例を用いて説明する．図2では，Customer.printFee( ) が Customer.movieList を参照している場合，Customer.printFee( ) を Rental へ移動する際に Customer.movieList の移動も検討し，実際に両者とも移動させた例である． リファクタリング完了後のソースコードは，移動するメンバと関連のあるメンバの移動により，クラスの責務が明確になったと言える．

検討２についても同様に例を用いて説明する．図 3 では，Customer.printFee( ) が Customer.movieList を参照していおり，Customer.print( ) が　Customer.printFee( ) を参照している．Customer.printFee( ) のみを Rental へ移動する場合，図 3 ではメンバ間で以下の参照方法を使用した．

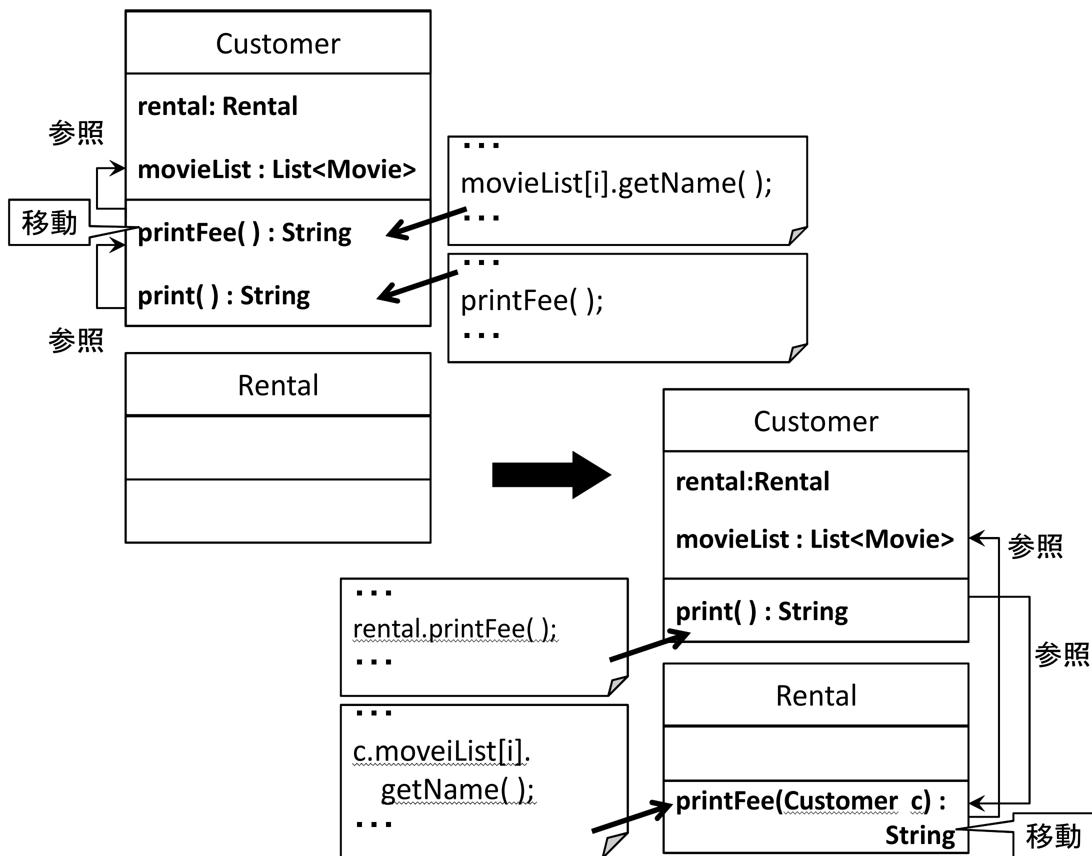

図 3: メンバの移動において，メンバの移動後にメンバ間の参照方法を調整した例

1. 移動元クラスの Customer.print( ) から，移動元クラス Customer 内のフィールドで移動先クラス型のインスタンス変数である Customer.rental を介して，rental.printFee( ) という様に Rental.printFee( ) を参照する．

2. 移動したメソッド Rental.printFee( ) の引数に移動元クラス型の引数 c を追加し，これを介して c.movieList という様に Customer.movieList を参照する．

図 3 の場合，移動元クラス Customer 内に移動先クラス Rental 型のインスタンス変数が存在していたが，メンバの移動を適用するソースコードに，このようなインスタンス変数が存在しない場合も考えられる．その際，開発者は新たに移動先クラス型のフィールドを移動元クラス内に追加することが求められる．

2.2 コンパイルエラーが発生するメンバの移動

上述のように移動するメンバについて必要な検討を行った上でメンバの移動を行っても，メンバの移動により，メンバ間の参照・被参照の関係が成立しなくなる場合がある．これは，private メンバを参照していたメンバの移動や，他のメンバから参照されていた private メンバの移動により起こることが多い．

Java 言語における private メンバは，他のクラスから参照することができない．ここで，$memberA$ が $memberB$ から参照されているとき，$memberA$ を "被参照メンバ"，$memberB$ を "参照メンバ"とする（図 4(a)）．図 4 のように被参照メンバが private メンバである場合，メンバの移動により被参照メンバや参照メンバを移動すると，参照メンバと被参照メンバが異なるクラスに所属し，参照メンバから被参照メンバへ参照ができない（以降，参照切れと呼ぶ．図 4(b),(c) で例を示す）．この場合，ソースコードに適用可能な編集手順は以下のように複数存在し，開発者は以下に挙げる編集のいずれかを選択し，編集をソースコードに適用して参照切れを解決する必要がある．

編集 1 被参照メンバのアクセス修飾子を変更する．

編集 2 被参照メンバが private フィールドの場合，フィールドのカプセル化を行い，参照メンバからカプセル化によって追加した $getter$, $setter$ を介して被参照メンバを参照する．

編集 3 移動した参照メンバ（もしくは被参照メンバ）の所属するクラスへ被参照メンバ（もしくは参照メンバ）を移動する．

図 5(a) において，A.add と A.print は A.p を参照しているため，A.add と A.print は参照メンバ，A.p は被参照メンバである．図 5(a) で，A.add を B へ移動する場合，A.add の被参照メンバである A.p は private メンバであるため， A.add を B へ移動後，図 5(b) で B.add から A.p へ参照できない． 図 5(b) の参照切れに対して，A.p のアクセス修飾子の変更（編集 1）を適用したのが図 5(c)，A.p のカプセル化（編集 2）を適用したのが図 5(d)，被参照メンバである A.p の移動（編集 3）を適用したのが図 5(e) である．図 5(e) では，private な被参照メンバである B.p は A.print から参照できない．よって，A.p の移動を選択し図 5(b) に適用した開発者は，図 5(e) の参照切れに対して，さらに編集 1～3 のいずれかを選択しソースコードに適用しなければならない．

このように，メンバの移動によって生じた参照切れの編集手順は，ソースコードや開発者の意図によって複数存在し，各編集手順から導かれるソースコードはそれぞれ異なるものである．

また，図 5 のようなメンバの移動に伴う編集作業は，ソースコードを頂点，編集手順を構成する各ステップ（以降，編集ステップ）を有効辺とする木構造で表現できる．木の根はリ

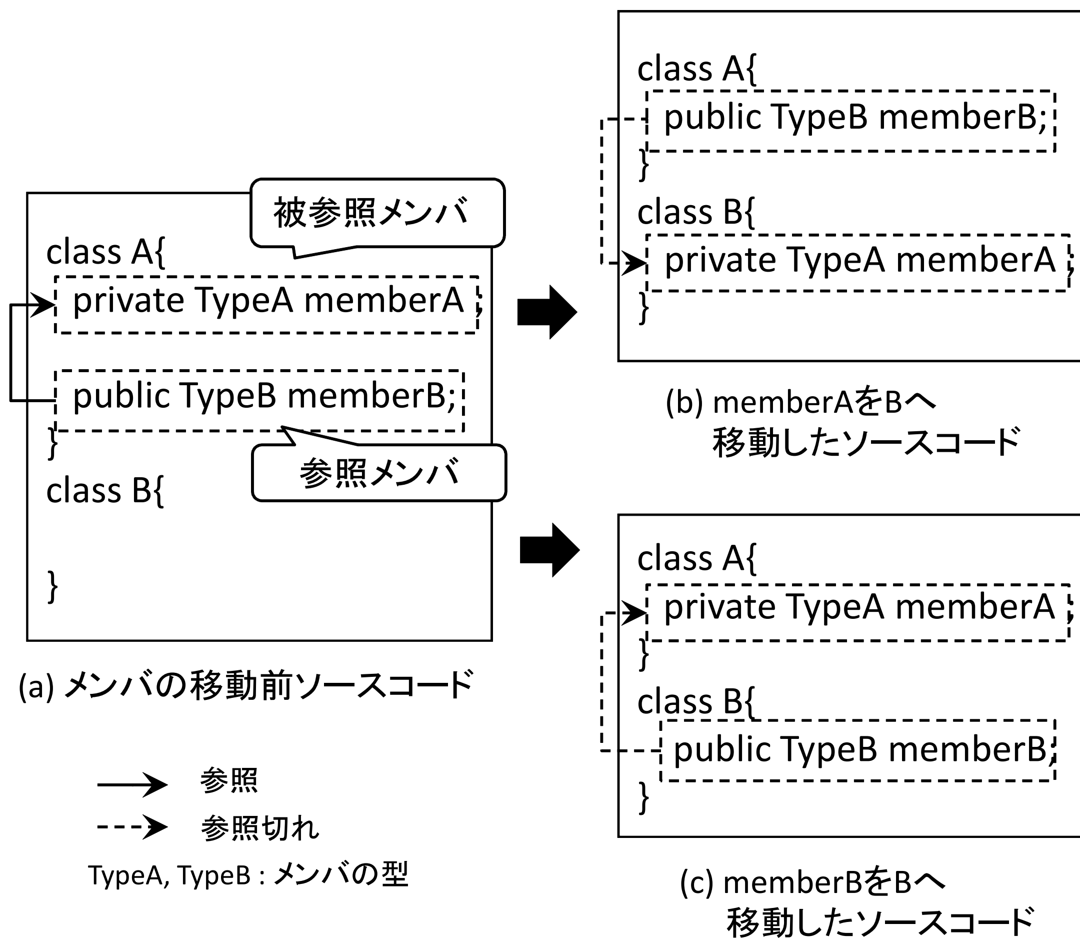

図 4: メンバの移動により参照切れになる例（$memberA$ はメソッドやフィールドが考えられる．$memberB$ はメソッドである．メンバの本体は省略する.）

ファクタリング開始時のソースコード，葉はリファクタリング完了後のソースコードを指し，道は編集手順の 1 つを表す．図 6 は図 5(a) の “メンバの移動”に伴う編集作業を表す木であり，リファクタリングを行う際，開発者は道を選択することで編集手順を決定できる

2.3 リファクタリング支援機能

リファクタリングパターンに伴うソースコードの編集手順は，複雑な編集ステップで構成されていることが多い．ウェブサイト [8] で紹介されているリファクタリングパターンの一部は，統合開発環境 Eclipse[1] のプラグイン JDT(Java Development tools)[6] で提供されており，リファクタリングパターンに伴なう手作業で行うには困難な編集作業を支援するものである．開発者が，リファクタリングを適用するソースコード中の要素やリファクタリング

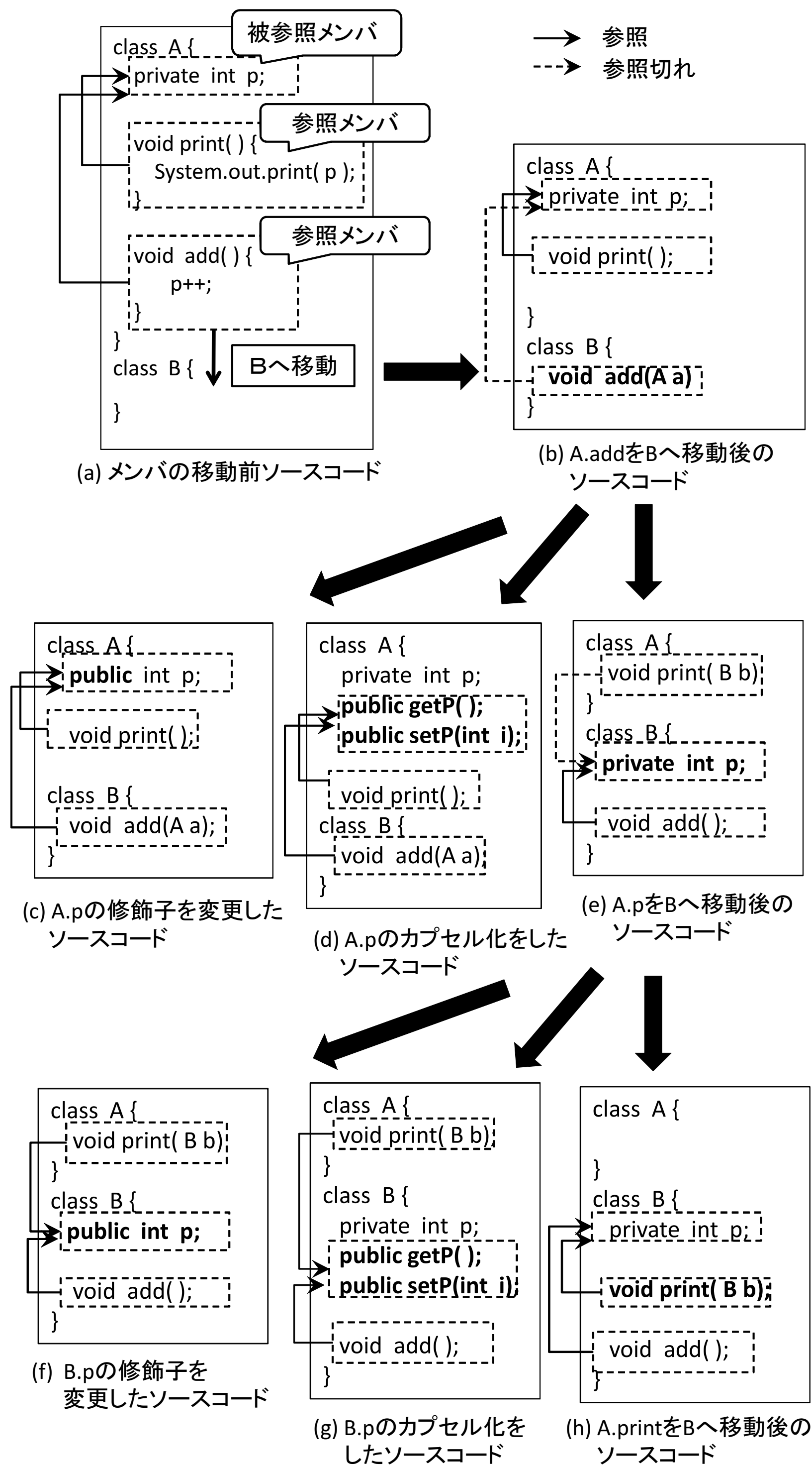

図 5: メンバの移動に伴う参照切れを編集する例（(b)〜(h) のメソッド本体は省略）

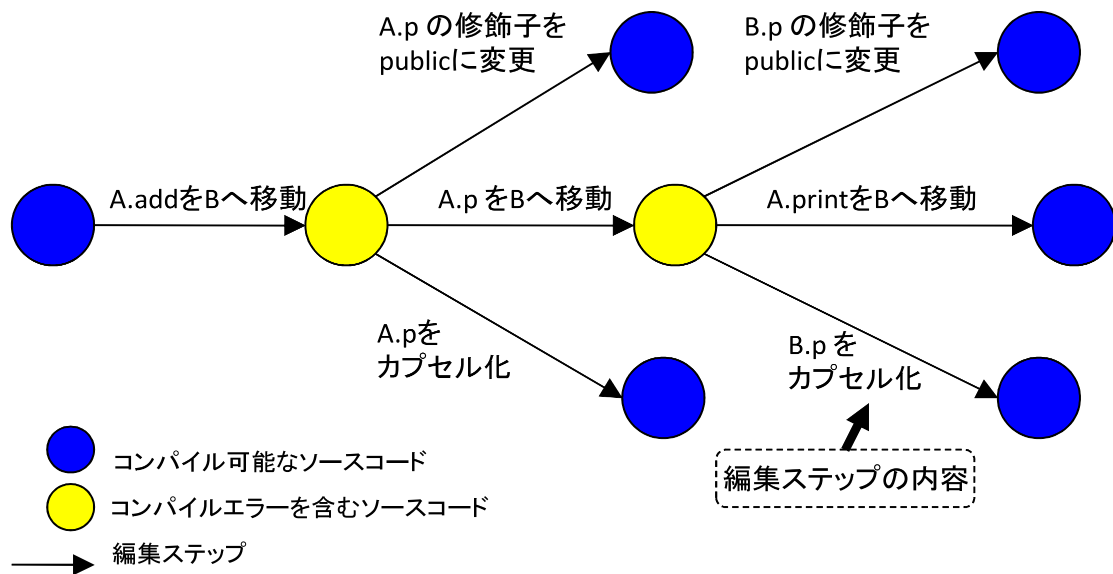

図 6: “メンバの移動”に伴う編集作業の木の例

パターンを選択し，リファクタリングパターン実行に必要な情報を入力することでソースコード編集を行うことができる．

2.1 節で説明したメンバの移動リファクタリングを Eclipse のリファクタリング支援機能を用いて実行する例が図 7，8 である．図 7 で開発者が移動するメンバを選択し，リファクタリングパターン中から “移動” を選択することで，メンバの移動に必要な情報入力を支援するウィザードである図 8 が起動する．図 8 のウィザードでは移動元クラスと移動先クラス間の参照が可能となるように以下の情報入力を必要とする．

- 移動元クラスに存在する，移動先クラス型のインスタンス変数
- 移動するメソッドに移動元クラス型の引数を追加する際の変数名

2.1 節では，メンバの移動の際に必要な検討１，２について述べた．しかし， Eclipse のリファクタリング支援におけるメンバの移動の際は，検討１，２が行われない．具体的には，移動元クラスから移動先クラスを参照するために，移動元クラスに移動先クラス型のインスタンス変数があることが前提となっている．よって移動元クラスに移動先クラス型のインスタンス変数が存在しないソースコードでは，メンバの移動を行うことはできない．さらに，メンバの移動後のソースコードでは，移動先クラスから移動元クラスへの参照方法もメソッドの引数を介する参照に限られている．このように，メンバの移動を適用するソースコードの事前条件，事後条件が限定的であるが，半自動的にメンバの移動を行うことができる．

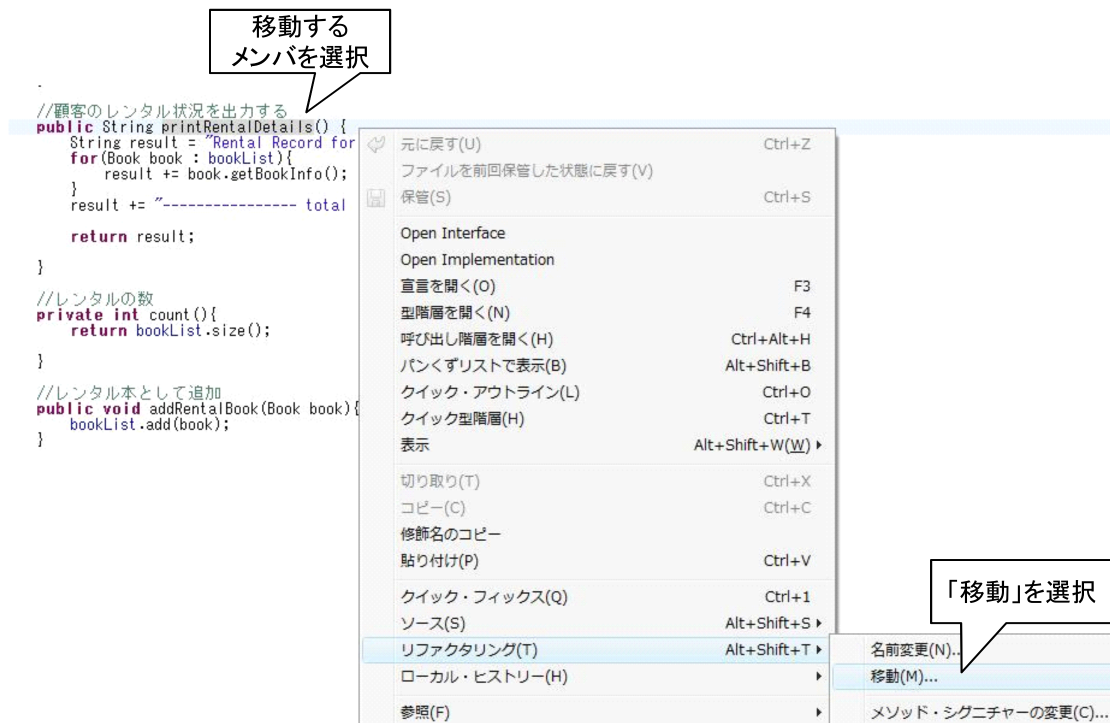

図 7: Eclipse 上でメンバの移動を実行する例

2.4 リファクタリング支援機能の問題点

2.3 節で述べたように，既存のリファクタリング支援機能では，リファクタリングパターンに伴うソースコードの編集作業が半自動的に行える．しかし，既存のリファクタリング支援機能の中で，2.2 節で述べたメンバの移動により参照切れが生じる場合，それを解消する複数の編集手順を表す図 6 のような編集作業の木を提示できるものが確認できない．

その理由として，リファクタリング支援機能でメンバの移動を行った際に参照切れが発生する場合，図 9 のように，リファクタリング支援機能はメンバの可視性を private から protected や public に変更する編集しか提示できないということが挙げられる．これは開発者が意図するメンバのカプセル化に反する編集となる可能性があり，望ましい編集ではない．2.1 節で説明したように，メンバの移動による参照切れを解消するためには，移動元クラスに所属する参照メンバや被参照メンバの移動や，被参照メンバのカプセル化（被参照メンバがフィールドの場合）の編集の適用も考えられる．既存のリファクタリング支援手法では，メンバの移動により生じた参照切れを解消するために，これらの編集を提示することができない．

参照切れを解消する編集手順が複数存在するメンバの移動を行う場合， リファクタリン

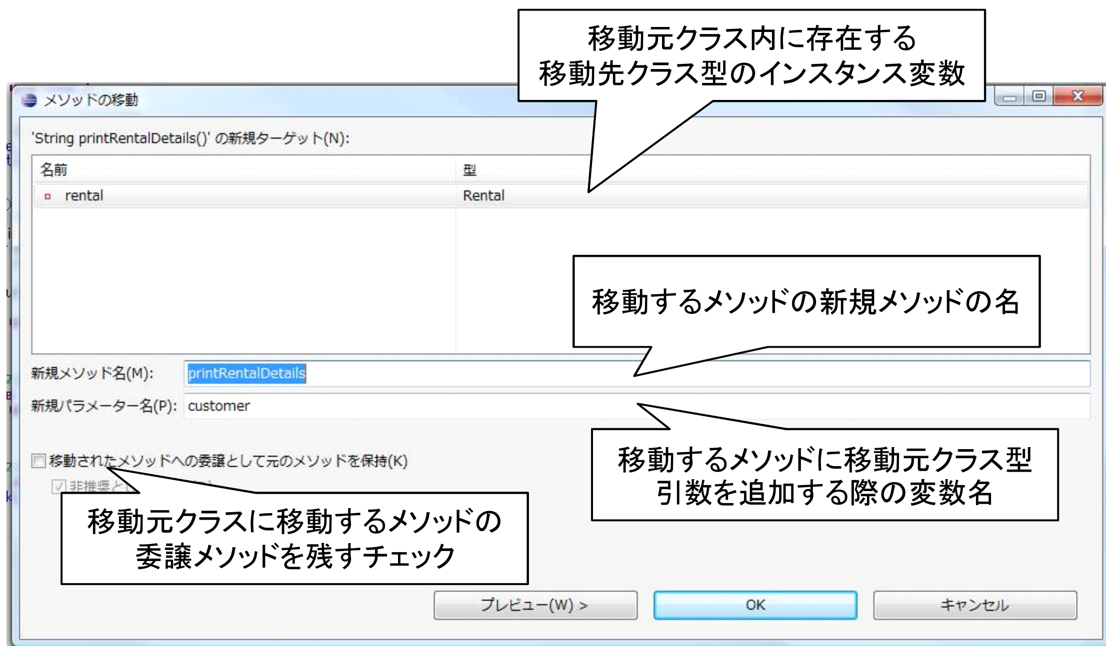

図 8: メンバの移動に必要な情報入力を支援するウィザード

グ経験が豊富な開発者は，既存のリファクタリング支援機能で半自動化されている編集ステップを利用して，目的とするソースコードまでの編集手順を考案し，ソースコードを編集することができるが，リファクタリング経験の乏しい開発者にとって，これは困難である．リファクタリング経験の少ない開発者は，どのような編集手順が考えられるかを十分に理解しないままソースコードを編集し，目的のソースコードを得ることができないという問題が生じる可能性がある．

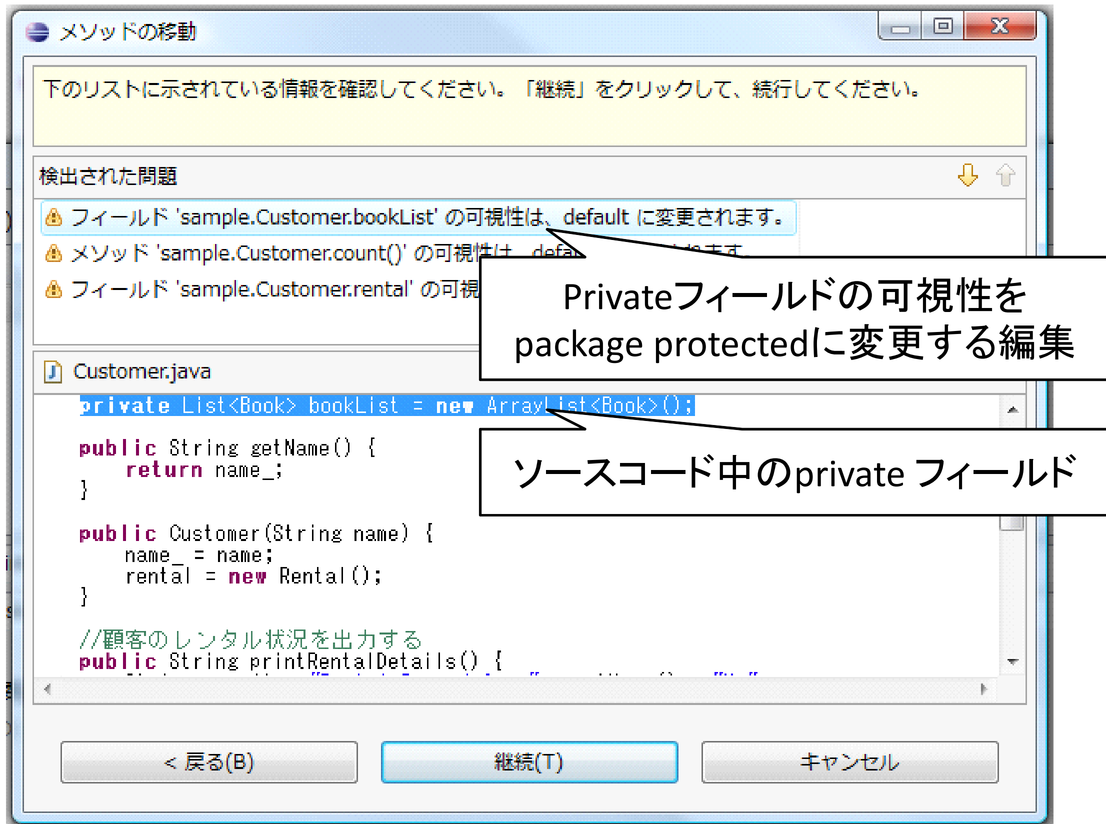

図 9: 参照切れの原因となる被参照メンバの可視性を変更する編集を提示するウィザード

3 提案手法

本研究では，メンバの移動に伴ない生じた参照切れを解消する複数の編集手順を探索し，編集手順を適用した結果のソースコードを自動的に導く手法を提案する．メンバの移動により発生した参照切れを解決する編集ステップを自動的に導出し，ソースコードに適用することで参照切れを解消する編集手順を探索する．図 10 で示すように，以下のプロセスで編集手順を探索する．

1. メンバの移動を適用した参照切れを含むソースコードからコンパイルエラーを取得
2. 参照切れを解決する編集ステップを導出
3. ソースコードに編集ステップを適用し，編集ステップとその適用結果のソースコードを保持
4. 参照切れを含むソースコードに対し，繰り返し編集ステップを導出，適用する

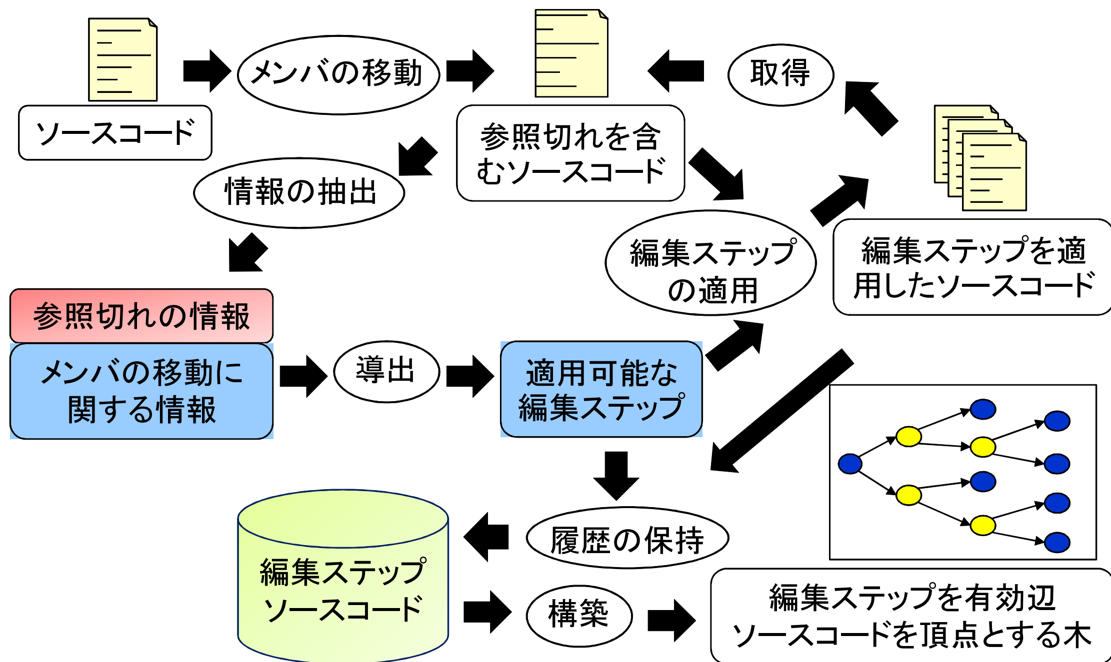

図 10: 提案手法の概要図

5. 全てのソースコードについて参照切れが解消された後，編集手順の探索を終了する

3.1 編集ステップ導出のための情報取得

Eclipse のリファクタリング支援機能を利用したメンバの移動により，ソースコードに参照切れが発生した場合を想定する．メンバを移動後，参照メンバと private な被参照メンバが異なるクラスに属する場合，参照メンバ内で参照切れが起こる．このとき，開発者がリファクタリング支援機能を利用して行ったメンバの移動より，以下の情報を取得する．

- メンバの移動先クラス
- メンバの移動元クラス

また，メンバの移動により生じた参照切れより，以下の情報を取得する．

- アクセス修飾子が private な参照メンバと参照メンバが所属するクラス
- 被参照メンバと被参照メンバが所属するクラス

説明のため，図 4 のように，被参照メンバである $memberA$ と参照メンバである $memberB$ が A に所属し，A から B へメンバを移動後，参照切れが起こったとする（$memberA$ を移

動したソースコードである図 4(b) と，$memberB$ を移動したソースコードである図 4(c) が考えられる）．開発者が適用したメンバの移動と参照切れより図 4(b)，図 4(c) のソースコード例では，表 1 の情報が取得できる．

3.2　編集ステップの導出

表 1 を元にして，参照切れを解決する以下の編集ステップを導出する．

メンバの移動： 表 1 で情報を取得した参照メンバ，被参照メンバのうち，移動元クラスに所属しているメンバを移動先クラスへ移動する．図 4(b) のソースコードの場合，移動元クラスである A に所属している $memberB$ を移動先クラスである B へ移動する（図 11(a)）．図 4(c) のソースコードの場合，移動元クラスである A に所属している $memberA$ を移動先クラスである B へを移動する（図 11(b)）．移動前のメンバへの全ての参照を，移動後のメンバへの参照に変更する．

変数のカプセル： $memberA$ がフィールドの場合，$memberA$ のカプセル化を行う．$memberA$ の所属クラスに $memberA$ の $getter$，$setter$ を追加する.（図 11(c)）$memberA$ への全ての参照を $getter$，$setter$ を介する間接参照に変更する．

アクセス修飾子の変更： $memberA$ のアクセス修飾子を public に変更する.（図 11(d)）

上記で求められるメンバの移動の編集ステップは，移動元クラスから移動先クラスへの移動に限る．参照切れを解消する編集として，移動先クラスに所属しているメンバを移動元クラスへ移動するメンバの移動も考えられるが，そのようなメンバの移動は以下のいずれかの編集となるので，参照切れを解消する編集ステップとして導出しない．

表 1: 編集ステップ導出のために取得する情報例

項目	図 4(b) のソースコードの場合	図 4(c) のソースコードの場合
メンバの移動先クラス	B	B
メンバの移動元クラス	A	A
参照メンバ	$memberB$	$memberB$
参照メンバの所属クラス	A	B
被参照メンバ	$memberA$	$memberA$
被参照メンバの所属クラス	B	A

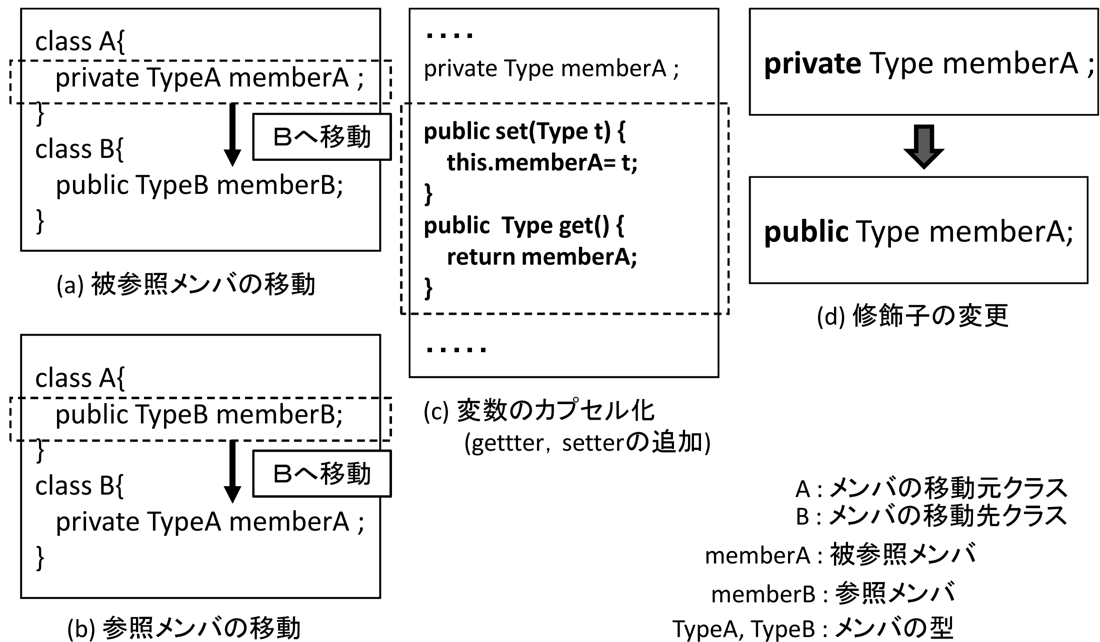

図 11: 編集ステップ

- 開発者がリファクタリング支援機能を利用して行ったメンバの移動を取り消す編集
- 過去に導出された編集ステップによって解消された参照切れを再度生じさせる編集

3.3 編集ステップ導出・適用の繰り返し処理

3.2 節で説明した編集ステップをそれぞれ参照切れが存在するソースコードに適用し，適用結果のソースコードを取得する．メンバの移動を適用した結果，新たな参照切れが生じる場合がある．適用結果のソースコードで参照切れを含むソースコードについて，繰り返し編集ステップの導出と適用を行う．

編集ステップの導出と適用を繰り返す様子を具体例で説明する．図 5(b) は，図 5(a) において A から B へ参照メンバである A.add を移動した，参照切れを含むソースコードである．開発者がリファクタリング支援機能を利用して行ったメンバの移動と参照切れから，表 2 の情報が取得できる．

よって図 5(b) のソースコードに対して導出される編集ステップは以下になる．

メンバの移動: 移動元クラスである A に所属している A.p を移動先クラスである B へ移動

変数のカプセル化: 被参照メンバである A.p のカプセル化

アクセス修飾子の変更: 被参照メンバである A.p のアクセス修飾子を public に変更

導出された編集ステップをそれぞれソースコードに対して適用する．“アクセス修飾子の変更”，“変数のカプセル化” を適用した結果がそれぞれ図 5(c)，(d) であるが，参照切れが解決されたソースコードとなる．しかし，“メンバの移動” を適用した図 5(e) のソースコードは，A.print と B.p の間で参照切れが起こる．よって，図 5(e) のソースコードに対して，繰り返し編集ステップの導出・適用を行う．適用したメンバの移動と図 5(e) の参照切れより，表 3 の情報が取得できる
導出される図 5(e) のソースコードに対する編集ステップは以下になる．

メンバの移動: 移動元クラスである A に所属している A.print を移動先クラスである B へ移動

変数のカプセル化: 被参照メンバである B.p のカプセル化

アクセス修飾子の変更: 被参照メンバである B.p のアクセス修飾子を public に変更

導出された編集ステップを図 5(e) に適用した結果である図 5(f)，(g)，(h) のソースコードは，すべて参照切れが解消されたものとなる．
編集ステップと、編集ステップを適用したソースコードは保持しておき，ソースコードを頂点，編集ステップを有効辺とする編集作業の木を構築する（図 6）．開発者は，編集作業の木の道を選択することで，メンバの移動に伴う編集手順の選択ができる．

3.4　適用例

提案手法を用いて，ある例に対してメンバの移動により生じる参照切れを解消する編集手順と編集手順を適用したソースコードを導出した．対象とするソースコードは図 12 である．Customer.printRentalDetails を Rental へ移動する．図 12 で Customer.printRentalDetails

表 2: 図 5(b) の情報取得例

メンバの移動先クラス	B
メンバの移動元クラス	A
参照メンバ	add
参照メンバの所属クラス	B
被参照メンバ	p
被参照メンバの所属クラス	A

表 3: 図 5(e) の情報取得例

メンバの移動先クラス	B
メンバの移動元クラス	A
参照メンバ	$print$
参照メンバの所属クラス	A
被参照メンバ	p
被参照メンバの所属クラス	B

は 2 つの private フィールド Customer.bookList，Customer.count を参照しており，Customer.printRentalDetails の移動により，図 13 では private フィールドに対する 2 種類の参照切れが生じる．参照切れを解消する編集手順を示すのが図 14 である． Customer.bookList, Customer.count について導出された，メンバの移動，アクセス修飾子の変更，フィールドの場合は変数のカプセル化の編集ステップをソースコードに適用していき，参照切れが解消されたコンパイル可能なソースコードを導出できた．このように導出された編集手順と，編集手順を適用した結果のソースコードは，既存のリファクタリング支援機能では自動的に導出することができないものである．

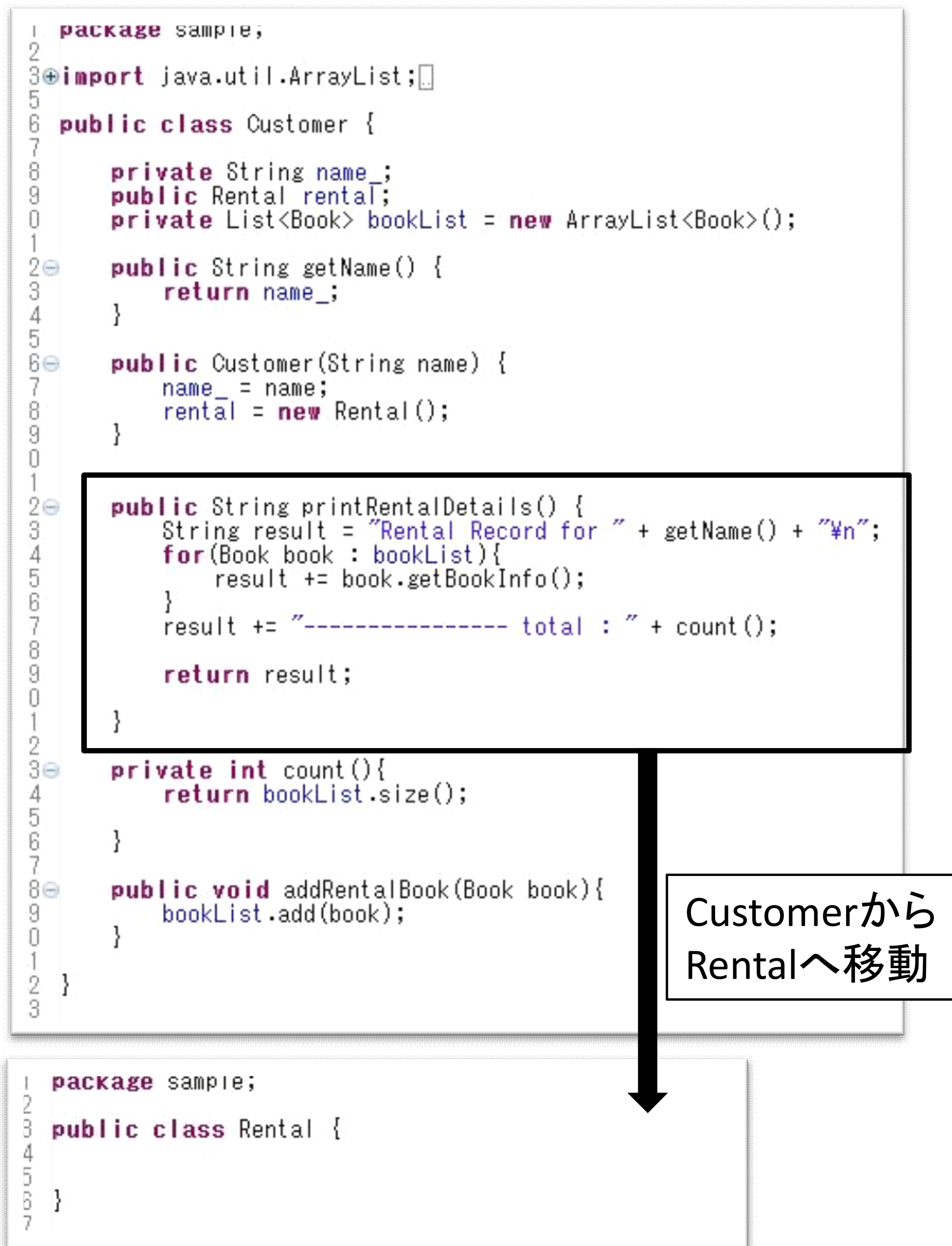

図 12: メンバの移動を適用するソースコード

```java
package sample;

import java.util.ArrayList;

public class Customer {

    private String name_;
    public Rental rental;
    private List<Book> bookList = new ArrayList<Book>();

    public String getName() {
        return name_;
    }

    public Customer(String name) {
        name_ = name;
        rental = new Rental();
    }


    private int count(){
        return bookList.size();

    }

    public void addRentalBook(Book book){
        bookList.add(book);
    }

}
```

移動したメソッドから
参照できない
privateフィールド

移動したメソッドから
参照できない
privateメソッド

```java
package sample;

public class Rental {

    public String printRentalDetails(Customer customer) {
        String result =
            "Rental Record for " + customer.getName() + "¥n";
        for(Book book : customer.bookList){
            result += book.getBookInfo();
        }
        result +=
            "---------------- total : " + customer.count();

        return result;
    }

}
```

参照切れが
発生したメソッド

図 13: メンバの移動により参照切れが発生したソースコード

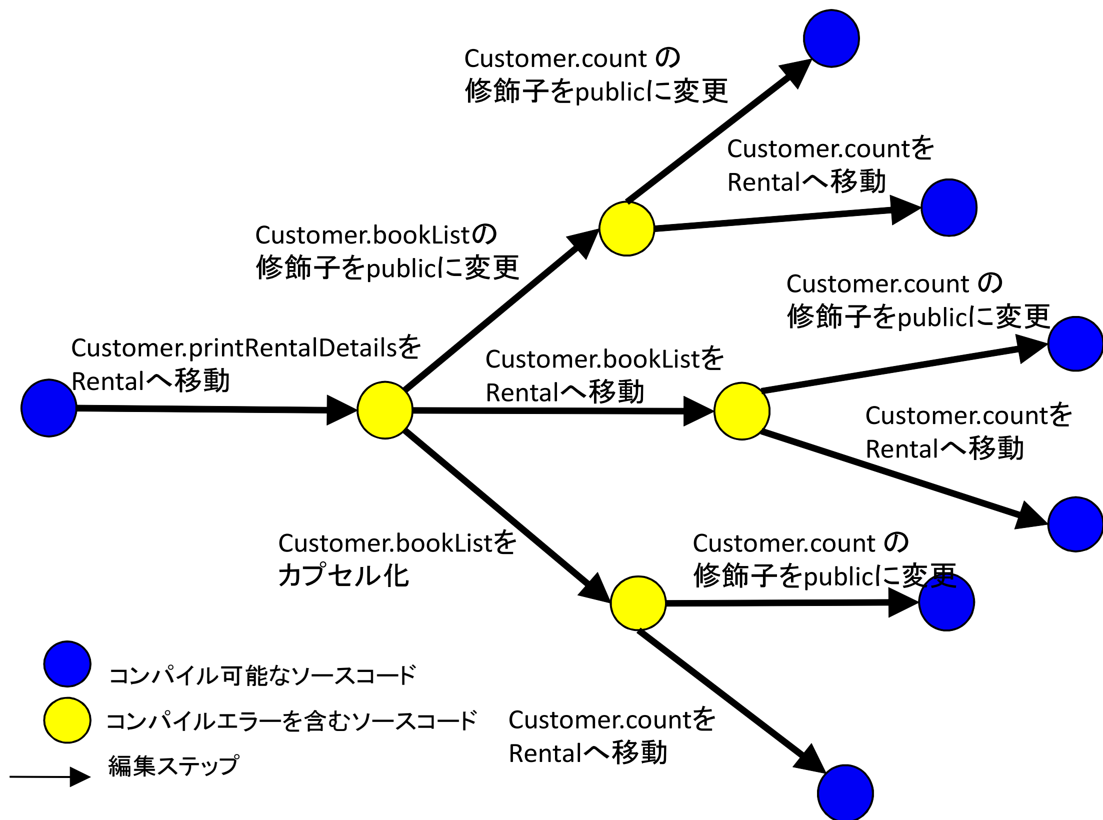

図 14: 図 13 の参照切れを解消する編集手順

4　実装

本章では，3 節で述べた提案手法を実装した Eclipse プラグインについて説明する．

提案手法を Eclipse プラグインとして実装する．図 15 で示すように，提案手法を実装したプラグインは，ソースコードに対して編集を適用する編集適用部，編集適用後のソースコードから編集ステップ導出のために必要な情報を取得するコンパイルエラー情報取得部，参照切れを解消する編集ステップを導出する編集ステップ導出部に分かれている．また，ソースコードに対して適用した編集手順と，編集を適用後のソースコードは編集履歴として保存する．

4.1　編集適用部

編集適用部では，開発者から与えられた編集や，編集ステップ導出部より導出された編集ステップをソースコードに適用する処理を行う．Eclipse ではソースコードの抽象構文木に対して編集を行うための API が用意されており，それらを利用して抽象構文木の編集の内容

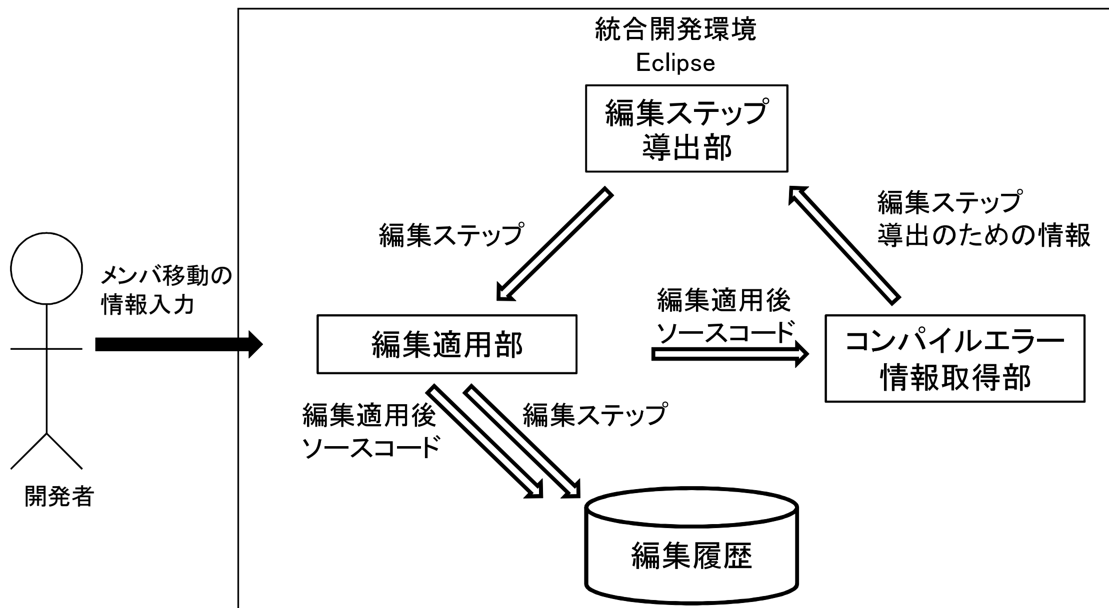

図 15: コンパイルエラー自動解消プラグイン

を指定することによりソースコードの編集が可能である．

Eclipse のリファクタリング機能の開発のために，LTK(Language Tool Kit) と呼ばれる API 群が提供されており，提案手法の実装に利用する API は，LTK のパッケージである org.eclipse.ltk.core.refactoring で提供されている以下のものである．提案手法で定めた編集ステップの適用は，以下で説明する Refactoring クラスや，Refactoring クラスのインスタンスを操作するクラスを利用することで実現できる．

Refactoring: リファクタリング支援機能で提供されている編集作業のプロセスがまとめられているクラス．ソースコードに対して編集作業を行う際の条件チェックや，適用する編集内容が記述された変更オブジェクトの生成を行う．提案手法で定めた編集ステップの生成は，このクラスを継承したクラスを実装し，変更オブジェクトの生成を行うことで実現できる．

Change: ソースコードの抽象構文木に対して適用する編集内容が記述されたクラス．Refactoring クラスのインスタンスから生成される．編集ステップの内容を保持する．

CheckConditionsOperation: Refactoring クラスのインスタンスを操作して，条件チェックの実行を行うクラス．Refactoring クラスのインスタンスで実装されている事前条件チェック，事後条件チェックを実行する．

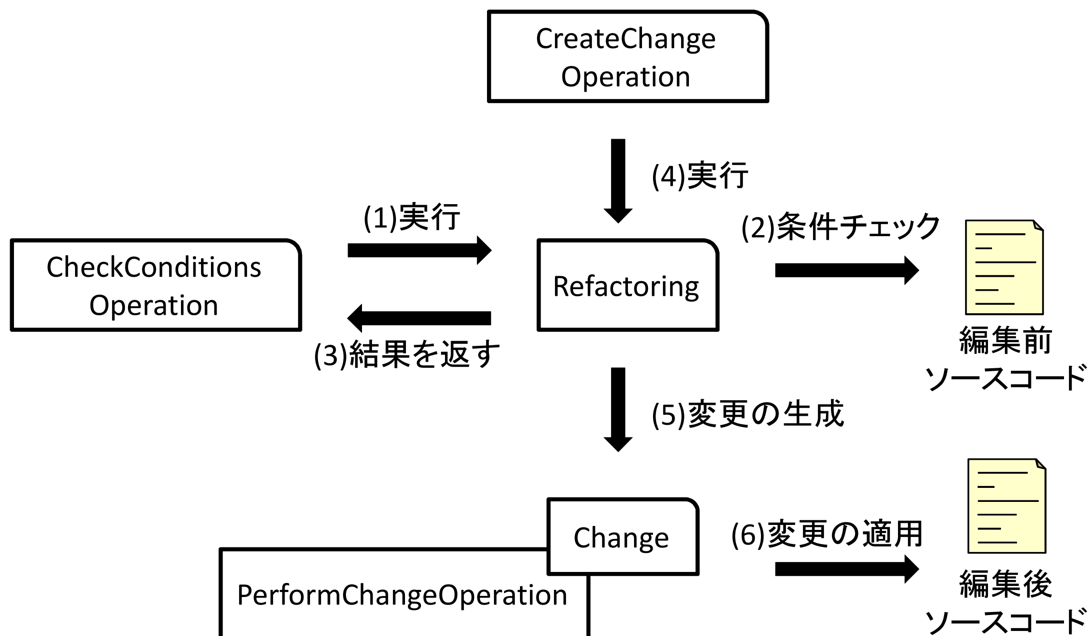

図 16: Eclipse のソースコードに対する編集適用の流れ

CreateChangeOperation: Refactoring クラスのインスタンスを操作して，変更オブジェクトの生成を実行するクラス．Change クラスのインスタンスの生成を実行する．

PerformChangeOperation: CreateChangeOperation クラスのインスタンスにより生成された Change クラスのインスタンスをソースコードに適用する処理を実行するクラス．Chnage クラスのインスタンスに実装されている抽象構文木の書き換え内容に従ってソースコードの抽象構文木の書き換えを実行する．

図 16 で処理の流れを示す．Refactring クラスのインスタンスを用いて，CheckConditionsOperation クラスのインスタンスによって実行される条件チェックや，CreateChangeOperation クラスのインスタンスによって実行される変更の生成を行う．変更の生成によって Change インスタンスが生成された後，PerformChangeOperation クラスのインスタンスによってソースコードに対する変更の適用が実行される．編集ステップ導出部で導出する Refactoring インスタンスである編集ステップを元に図 16 の処理を行う．

4.2 コンパイルエラー情報取得部

コンパイルエラー情報取得部では，開発者が行ったメンバの移動や，導出された編集ステップを適用したことにより，参照切れが発生したソースコードについて 3.1 節で説明した

各種情報を取得する．Eclipse ではソースコードに編集を適用すると自動的にソースコードがコンパイルされる．ソースコード中に含まれる参照切れを取得するために，ソースコードの構造解析や意味解析の結果を保持した抽象構文木の要素である CompilationUnit を利用する．Eclipse では，ソースコードの構造解析や意味解析の結果を保持した抽象構文木を利用するための API が Eclipse プラグインである org.eclipse.jdt.core の org.eclipse.jdt.core.dom パッケージで提供されている．

ASTParser: ソースコードの構造解析，意味解析の結果を保持する抽象構文木を生成するための機能を提供するクラス．与えられらたソースコードの抽象構文木を生成することができる．生成される抽象構文木は org.eclipse.jdt.core.dom.ASTNode クラスのインスタンスで構成される．

CompilationUnit: org.eclipse.jdt.core.dom.CompilationUnit で定義されている．CompilationUnit は ASTNode の一種で，ソースコードの抽象構文木における根に相当する．抽象構文木を表す ASTNode クラスの子クラスであれば，CompilationUnit クラスのインスタンスにキャストできる． CompilationUnit よりソースコードのコンパイルエラーが取得できる．具体的には， CompilationUnit.getProblems() を呼び出すことにより，抽象構文木内に存在するコンパイルエラーの集合を取得することができる．

IProblem: CompilationUnit クラスのインスタンスから取得するコンパイルエラーを表現するためのインターフェースである．本研究で着目している参照切れは IProblem インターフェースを実装した org.eclipse.jdt.internal.compiler.problem.DefaultProblem クラスで表現される．IProblem.getArguments() を呼び出すことで，参照ができない private な被参照メンバと参照切れが存在する参照メンバが特定できる．特定された被参照メンバと参照メンバより 3.1 節で説明した情報を取得する．

4.3 編集ステップ導出部

3.2 節で説明したように，コンパイルエラー情報取得部で取得された情報をもとに，編集ステップを導出する処理を行う．

メンバの移動，アクセス修飾子の変更，変数のカプセル化の編集ステップはそれぞれ 4.1 節で述べた Refactoring クラスを実装することにより実現する．さらに，編集ステップをソースコードに適用する際，編集により影響を受けるソースコード（例えばメンバの移動を適用後，メンバへの参照を書き換えることが必要なクラス）をプロジェクト内から検索しなければならない．このような検索が実装された org.eclipse.core.resources.SearchEngine では Refactoring インスタンスによって抽象構文木の書き換えを行う対象ク

ラスの検索設定を行うことができる．各編集ステップについて，Refactoring クラスの実装と org.eclipse.core.resources.SearchEngine の設定を行う．また，抽象構文木の書き換えを実装する場合，抽象構文木の書き換えを行うための API である org.eclipse.jdt.core.dom.rewrite. ASTRewrite を利用する．

4.3.1 メンバの移動の実装

メンバの移動は，移動するメンバがフィールドの場合はインスタンス変数の移動を行い，メソッドの場合はメソッドの移動を行う．移動に関するリファクタリングは複雑な条件チェックや変更の生成を必要とするために，Refactoring クラスとその処理を委譲された Processor クラスで実装されている．Eclipse では Refactoring クラスで実装する処理を org.eclipse. ltk.core.refactoring.participants.RefactoringProcessor クラスに委譲している．

インスタンス変数の移動の実装は，Eclipse のリファクタリング支援機能で提供されている static メンバの移動を改良して行った．static の移動に対応する Refactoring クラスは org.eclipse.ltk.core.refactoring.participants.MoveRefactoring であり，Processor クラスは org.eclipse.jdt.internal.corext.refactoring.structure. MoveStaticMembersProcessor である．これら２つのクラスで，メンバのアクセス修飾子が static であるかチェックする部分や，移動による参照の変更方法をインスタンス変数用に改良した．メソッドの移動の実装はリファクタリング支援機能で提供されているメソッドの移動を利用した．メソッドの移動に対応する Refactoring クラスは org.eclipse.ltk.core.refactoring.participants.MoveRefactoring であり，Processor クラスは org.eclipse.jdt.internal.corext.refactoring.structure. MoveInstanceMethodProcessor である．

4.3.2 アクセス修飾子の変更の実装

アクセス修飾子の変更は，リファクタリング支援機能で Refactoring クラスで実装された機能として提供されていない．よって，org.eclipse.ltk.core.refactoring クラスの子クラスとして ChangeModifierRefactoirng クラスを実装した．アクセス修飾子の変更は，適用するソースコードには構造的な制限はないので事前条件チェックや事後条件チェックは実装していない．

4.3.3 変数のカプセル化の実装

変数のカプセル化はリファクタリング支援機能で提供されている変数のカプセル化を利用した．変数のカプセル化に対応する Refactoring クラスの子クラスは org.eclipse.jdt.internal. corext.refactoring.sef.SelfEncapsulateField である．

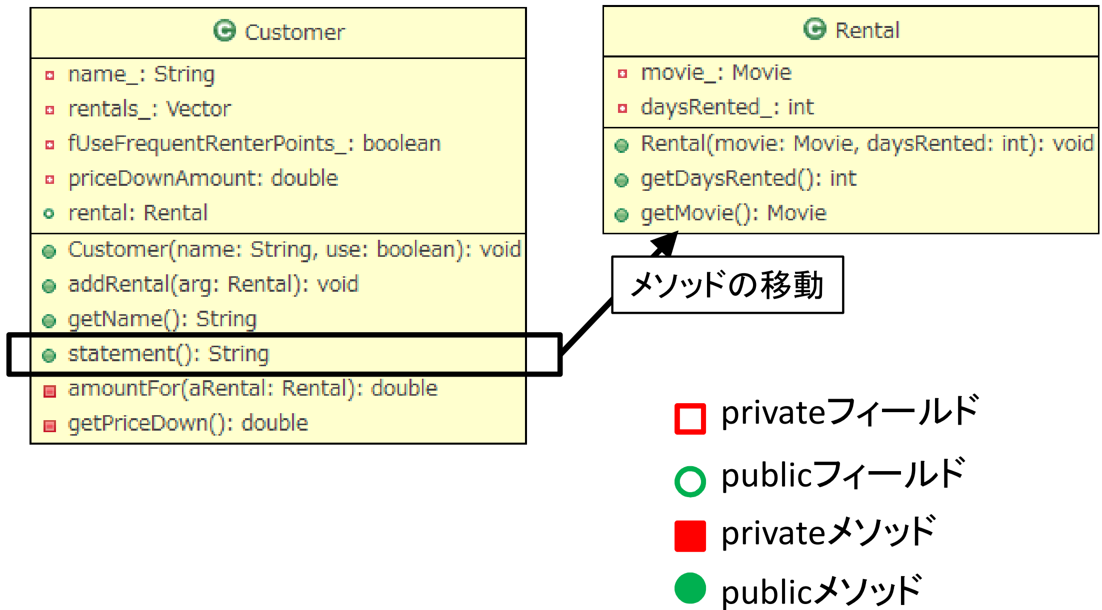

図 17: 実験対象ソースコードのクラス図

5　適用実験

提案手法が出力する編集手順の妥当性を確認するために，被験者 6 人に対して参照切れが生じるメンバの移動リファクタリングを実施した．被験者 6 人が参照切れを解消する編集を行ったソースコードを，メンバの移動により参照切れが解消されたソースコードの正解集合とし，提案手法で導出したソースコードと比較した．比較の結果，被験者が行った参照切れを解消するための編集手順が，提案手法で導出される複数の編集手順に多く含まれることが確認できた．

5.1　実験内容

被験者が適用した編集手順と提案手法が導出した編集手順の比較行った．具体的には以下のような条件で作業を実施した．

対象 書籍 [2] で使用されているビデオレンタルの料金計算プログラム（図 17）．

内容 図 17 の Customer.statement() を Retanl クラスに移動することにより生じる参照切れを解消する編集を行う．参照切れが解消されたソースコードを可能な限り求める．

環境 統合開発環境 Eclipse 上のエディタ

制約 Eclipse のリファクタリング支援機能による編集作業は適用しない

また，参照切れが解消されたソースコードを導出後，被験者に対してインタビューを行い，編集の手順を記録した．

5.2 実験結果

メンバの移動により生じる参照切れを解消したソースコードが 10 通り得られた．提案手法で，同じソースコードを用い参照切れを解消したソースコードを導出すると 30 通りのソースコードが得られた．それぞれのソースコードについて，被験者がソースコードに適用した編集内容を，メンバの移動，アクセス修飾子の変更，カプセル化，その他に分類して適用回数をまとめたのが表 4 である．

表 4 のソースコード A，D，E，F，G，H はその他の編集を適用していないソースコードであり，提案手法で導出する編集ステップの組み合わせで参照切れを解消したソースコードであった．これらのソースコードは提案手法で導出されるソースコードの集合に含まれるものであった．また，被験者が参照切れ解消の目的以外で，その他の編集を適用したソースコードが見られた（ソースコード B，C，I，J）．各ソースコードに適用されたその他の編集は以下の内容であった．

ソースコード B 移動先クラスに所属するメンバから，移動元クラスに所属するメンバへの参照のために，移動先クラスに移動元クラス型のインスタンス変数を追加した．

表 4: 編集の種類別適用回数

	メンバの移動	アクセス修飾子の変更	変数のカプセル化	その他
ソースコード A	1	2	1	0
ソースコード B	2	1	1	1
ソースコード C	1	1	1	1
ソースコード D	1	0	2	0
ソースコード E	2	1	1	0
ソースコード F	1	1	1	0
ソースコード G	1	2	0	0
ソースコード H	2	1	1	0
ソースコード I	2	1	1	1
ソースコード J	2	1	0	1

ソースコード C インスタンス変数を使っていないメソッドを static アクセス修飾子を追加しクラスメソッドにした.

ソースコード I List 型のインスタンス変数にジェネリクスを追加した.

ソースコード J メンバの移動の結果，他のクラスから利用されていない getter メソッドを削除し，変数への参照を直接参照にした.

ソースコード B は，移動元クラスと移動先クラス間のメンバの参照のために，Eclipse のリファクタリング支援機能がソースコードに適用する編集とは異なる編集が適用された．2.3 節のように，Eclipse がメンバの移動を適用する際のソースコード変換は限定的である．提案手法は Eclipse のメンバの移動を利用して編集ステップの適用をするために Eclipse のソースコード変換の結果と異なるースコードであるソースコード B は提案手法では導出できない.

ソースコード C は，インスタンス変数を利用していないメソッドは, クラスメソッドとして宣言できるため，被験者の判断でメソッドに static を追加する編集が行われた．この編集はコンパイルエラーを解消するための編集とは異なる.

ソースコード I は，被験者に用意したソースコードの不備で List 型変数にジェネリクスを追加する編集が行われた．この編集はコンパイルエラーを解消するための編集とは異なる.

ソースコード J は，他のクラスから利用されていない getter メソッドで参照できる変数は外部に公開する必要がないため，getter メソッドを削除し，クラス内で変数に直接参照する編集が行われた．この編集はコンパイルエラーを解消するための編集とは異なる.

その他の編集をソースコードに適用した被験者は，参照切れを解消するために提案手法で導出する 3 種類の編集ステップを利用していた．また，参照切れ解消のために提案手法で導出する編集ステップ以外の編集を適用しているソースコードは見られなかった．コンパイルエラーを解消する目的以外の目的で行われた編集を無視すると，ソースコード B 以外のソースコードに適用された編集手順は提案手法で導出可能である.

以上より，被験者が編集した 10 通りのソースコードのうちソースコード B を除く 9 通りに関しては，提案手法で導出される編集手順が適用されたソースコードである．よって，提案手法で導出された編集手順は妥当であると言える.

5.3 考察

実験の結果の考察を以下に述べる.

5.3.1　メンバ間の参照方法の検討

5.2 節では，被験者にメンバの移動により生じる参照切れを解消する編集作業を実施し，提案手法では導出できないソースコードについて説明した．提案手法の改善策として 5.2 節のソースコード B を提案手法で導出できるようにするには，移動元クラスと移動先クラス間の複数の参照方法を検討する必要がある．複数の参照方法については 2.1 節で説明したが，複数の参照方法を検討するために提案手法に次の検討を追加する必要がある．移動元クラスから移動先クラスへの参照方法を決定するために，以下の検討を行う．

- 移動元クラスに存在しているフィールドから移動先クラス型のインスタンスが取得できないか検討する．取得できる場合，取得するインスタンス変数を介して移動先クラスのメンバを参照することができる．

- 上記の検討で移動先クラス型のインスタンス変数が取得できない場合は，移動元クラスに移動先クラス型のインスタンス変数を宣言する．宣言したインスタンス変数を介して移動先クラスのメンバを参照することができる．

また，移動先クラスから移動元クラスへの参照方法を決定するために，以下の検討を行う．

- 移動先クラスに存在しているフィールドから移動元クラス型のインスタンスが取得できないか検討する取得できる場合，取得するインスタンス変数を介して移動先クラスのメンバを参照することができる．

- 上記の検討で移動先クラス型のインスタンス変数が取得できない場合は，移動元クラスに移動先クラス型のインスタンス変数を宣言する．または，メソッドを移動してきた場合，メソッドの引数に移動元クラスのインスタンスを渡す．引数を介して移動元クラスのメンバを参照することができる．

提案手法で適用するメンバの移動の際，上記の検討により可能な編集全てを編集ステップとして導出し，適用することで提案手法で導出できるソースコードを増加させることができる．

5.3.2　導出する編集ステップの工夫

書籍 [2] で挙げられるリファクタリングパターンには，メソッドの引き上げとメソッドの引き下げのように，それぞれが逆の意味合いを持つリファクタリングパターンの組が存在する．これは，被験者が行う編集ステップにも当てはめることができる．参照切れを解消するためにメンバの移動を繰り返し行った結果，メンバの移動前は他のクラスから利用されていた getter メソッドがクラス内部からのみ利用されるようになることがある．このような場合，5.2 節で述べたその他の編集で，他のクラスから利用されない getter メソッドを削除し，

クラス内で変数への参照を直接参照に変更するという編集が見られた．これは変数のカプセル化とは逆の意味合いを持つ編集である．

提案手法の改善策として，現状で導出可能な編集ステップの逆の意味合いを持つ編集ステップの導出と適用を行うことが考えられる．例えば，変数のカプセル化を解消するためにgetter，setterメソッドを削除するような， 変数のカプセル化とは逆の編集ステップを導出し，ソースコードに適用することができれば，5.2節で説明したその他の編集が行われたソースコードを導出することができる．同様に，アクセス修飾子をpublicからprivateに変更するような，提案手法で導出するアクセス修飾子の変更とは逆の編集ステップの導出も考えることができる．publicからprivateにアクセス修飾子を変更する編集ステップは，参照切れを解消するためにメンバの移動を繰り返し行った結果，他のクラスから参照されなくなるpublicメンバが存在するソースコードに対して導出を検討するのが望ましい．

5.3.3　ソースコードの外部的振る舞い

5.2節では，提案手法で導出する編集ステップが，メンバの移動の際に生じた参照切れを手作業で解消する編集内容と，類似していることを示した．しかし，5.2節の実験では，リファクタリングによって振る舞いが変化する可能性について考慮していないため，被験者が適用した編集手順の中には， ソースコードの外部的振る舞いが変化するものが存在すると考えられる．提案手法でも同様にソースコードの外部的振る舞いについて考慮しないまま編集ステップの導出を行っているため，編集手順の適用によるソースコードの外部的振る舞いの保存が保証されない．

提案手法の改善策として，メンバの移動により生じる参照切れを解消し，かつソースコードの外部的振る舞いが保たれたソースコードを導出することが考えられる。導出されるソースコードの外部的振る舞いを保つためには，ソースコードのテストケースが必要である．テストケース内で呼び出されているメソッドがメンバの移動の対象となる場合には，移動するメソッドに委譲メソッドを残す必要がある．

5.3.4　導出されたソースコードの選択

提案手法では，メンバの移動により生じるコンパイルエラーを解消したソースコードを導出するが，現状では，開発者は複数あるコンパイル可能なソースコードから１つを選択することが困難である．複数あるコンパイル可能なソースコードをソースコードの品質などを用いて評価し，開発者へ選択の支援を行うことが重要である．ソースコードの品質については，メンバの移動による凝集度や結合度の変化，またアクセス修飾子の変更により外部に公開されたメンバの数などで評価することが考えられる．

6　関連研究

ソースコードの品質改善に有用と考えられるリファクタリングの組み合わせを探索的に求め，開発者に提示する手法の研究が行われている [3, 12]．また，Hayashi らは，ソフトウェアの開発履歴中から，適用されたリファクタリングの組み合わせを探索する手法を提案を行っている [4]．これら手法は，適用の可能性があるリファクタリングの組み合わせの中から，目的に適う組み合わせを特定する探索問題を解く手法と言える．単一のソースコードが対象であっても，適用の可能性がリファクタリングの組み合わせは膨大な数になることがあるため，種々のヒューリスティックもしくはメタヒューリスティックを利用して，探索にかかる時間の削減を試みている．本研究では，あるリファクタリングを行う際に，続けて適用するとコンパイル可能なソースコードを得られるリファクタリングの列を導出している．本研究で提案している手法では，コンパイル可能なソースコードが得られることを目的としており，またメンバの移動のみを対象としているため，導出されるリファクタリングの列の数が膨大になることはなかった．しかし，品質の高い（例えば，複雑度が低い）ソースコードが得られることを目的とした場合や，もしくはメンバの移動より複雑なリファクタリング（例えば，スーパークラスの抽出）を支援することを目的とした場合，導出されるリファクタリングの列の数が膨大になる可能性が考えられる．提案手法を拡張した結果，導出されるリファクタリングの列の数が膨大になった場合，O ’Keeffe らの実験で用いているメタヒューリスティックを利用して，導出にかかる時間を削減する方法が考えられる．

本研究では，リファクタリング中に発生するコンパイルエラーの解消を行うための編集手順を導出を行っている．しかし，リファクタリングの最終目的はソースコードの品質改善であることを考えると，提案手法が導出した編集手順の中から，品質改善に有用なもののみを抽出する手法が必要であると考えられる．リファクタリングよる品質改善の尺度として，種々のソフトウェアメトリクスが利用されている [7, 14, 5]. 提案手法が導出するソースコードをこれらソフトウェアメトリクスを用いて順位付けすることで，開発者は品質の高いソースコードから順番に検討することが可能となると考えられる．リファクタリングによる品質改善の尺度としてメトリクスが用いられる一方で，リファクタリングによる品質改善とソフトウェアメトリクス値が相関を持つとは限らないという議論 [15] もあるため，提案手法を拡張した際に用いるソフトウェアメトリクスの選択は，慎重に行う必要がある．

Mens らは，モデルリファクタリング [9] という概念を提案している．モデルリファクタリングとは，UML 等で記述されたソフトウェアモデルに対するリファクタリングのことを指す．Mens らは，ソフトウェアモデルに対してリファクタリングを行ったなら，それに伴って対応するソースコードを自動生成する手法を実現すべきと述べている．本研究は，メンバの移動というモデル上のリファクタリングを行うと，それに伴い対応するソースコードを自

動的に書き換える手法の提案を行っていると考えることができる．

開発者が行ったリファクタリングについて調査を行う研究が行われている [16, 10, 11]．今後，提案する手法を他のリファクタリングパターンに対応する，もしくは提案手法を実装したツールのユーザインタフェースを改善するにあたって，これらの調査を参考にしたいと考えている．Murphy らの調査によると，名前の変更のリファクタリングに次いで，移動を行うリファクタリングが行われていた．また，Muphy-Hill らの調査によると，メンバの移動を行うリファクタリングはツールを使わず行われていた．このことから，メンバの移動を行うリファクタリングの効率を高めるために，多くの開発者にとって使いやすい，自動的にメンバの移動を行うツールが必要であると考えられる．

7　あとがき

本研究では，リファクタリングを行う際に生じるコンパイルエラーについて，既存のリファクタリング支援機能ではコンパイルエラーを解消する複数の編集手順を提示できない問題点に注目した．また，問題点を解決するために，特にメンバの移動の際に生じる参照切れに注目し，メンバの移動を行う際に生じる参照切れに関して，それを自動で解消するリファクタリング支援手法を提案した．

メンバの移動の際に生じる参照切れを解消する編集ステップとして，メンバの移動，アクセス修飾子の変更，変数のカプセル化を提案したが，手作業で参照切れを解消する際にも，これらの編集ステップが利用されていることを実験で示した．また提案手法の適用例を元に，メンバの移動の際に生じる参照切れを解消する編集手順の例を示した．

今後の課題として，１つめにソースコードの外部的振る舞いを考慮した編集ステップの導出が挙げられる．開発者が保持しているテストケースを元に，保存する振る舞いを判別し，振る舞いを保存するメンバの移動を編集ステップとして導出するべきである．２つめに，導出されたソースコードの選択支援が挙げられる．導出された複数のソースコードについて，凝集度や結合度等のメトリクスを用いたランク付けやアクセス修飾子の変更により外部に公開されたメンバの数で安全性を示す，といった選択支援が考えられる．

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，御助言を頂きました 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 准教授に心から感謝致します.

本論文を作成するにあたり，適切な御助言を頂きました 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 石尾 隆 助教に心から感謝致します.

本研究を通して，様々な御協力を頂きました 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 吉田 則裕 氏に深く感謝致します.

本研究を通して，様々な御助言を頂きました 立命館大学情報理工学部情報システム学科 丸山 勝久 教授，東京工業大学大学院情報理工学研究科 計算工学専攻 林 晋平助教に深く感謝致します.

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致します.

参考文献

[1] Eclipse. `http://eclipse.org`.

[2] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[3] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1106–1113, New York, NY, USA, 2007. ACM.

[4] S. Hayashi, Y. Tsuda, and M. Saeki. Detecting occurrences of refactoring with heuristic search. In *Asia-Pacific Software Engineering Conference*, pp. 453–460, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[5] Y. Higo, Y. Matsumoto, S. Kusumoto, and K. Inoue. Refactoring effect estimation based on complexity metrics. In *ASWEC '08: Proceedings of the 19th Australian Conference on Software Engineering*, pp. 219–228, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Java Development tools. `http://www.eclipse.org/jdt/`.

[7] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. *Software Maintenance, IEEE International Conference on*, 0:0576, 2002.

[8] M. Fowler. `http://refactoring.com`.

[9] T. Mens, G. Taentzer, and D. Müller. Challenges in model refactoring. In *Proc. 1st Workshop on Refactoring Tools*. University of Berlin, 2007.

[10] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23(4):76–83, 2006.

[11] E. Murphy-Hill and A. P. Black. High velocity refactorings in eclipse. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pp. 1–5, New York, NY, USA, 2007. ACM.

[12] M. O'Keeffe and M. O. Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.*, 20(5):345–364, 2008.

[13] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[14] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, p. 30, Washington, DC, USA, 2001. IEEE Computer Society.

[15] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *WoSQ '07: Proceedings of the 5th International Workshop on Software Quality*, p. 10, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pp. 458–468, Washington, DC, USA, 2006. IEEE Computer Society.