Vol.2015-SE-187 No.37
2015/3/13

オブジェクト指向プログラム固有なデバッグの困難性を克服する Back-in-Time デバッガの実現に向けて

久米 出[1,a)]　波多野 賢治[2,b)]　中村 匡秀[3,c)]　柴山 悦哉[4,d)]

概要：**逆回し** *(back-in-time)* **デバッガ**はプログラムのトレース (実行履歴) を記録して過去の状態の参照を可能する機能を有しており、デバッグに於ける**診断**のあり方を根本から変える可能性を秘めている。しかしながらその機能を有効に活用するためには、作業者が膨大なトレースの中から適切な実行時点を指定し、その状態の正不正を判定しなければならない。こうした指定や判定は作業者自身のプログラム理解に大きく依存している。
オブジェクト指向プログラミングはプログラムの再利用性や拡張性が向上させる反面、コードの理解を困難にする傾向が指摘されている。我々はこうしたオブジェクト指向プログラムに固有な問題を解決するために、**外挿診断法** *(diagnosis by extrapolation)* という手法を提案し、それを実現する逆回しデバッガを開発中である。本手法は指定された時点の実行文脈を、作業者の既知の情報を用いて抽象化する事によって、問題解決を実現する点に最大の特徴を有している。本論文では実用的なプログラムのデバッグ事例を通じて逆回しデバッガが実装すべき機能と手法の有効性の評価方法を考察する。

1. はじめに

プログラムコード中の誤り、即ち**不具合**箇所の実行によって不正な状態が発生する。この状態の不正或いはその発生を**感染**と呼ぶ。感染は連鎖して外部から観測される不正な状態、則ち**障害**を発生させる。デバッグとは障害からコードの不具合箇所を特定する作業である。不具合が疑われるコード箇所から障害の発生に至るまでの感染の連鎖の過程に関する検証可能な仮説は**診断**と呼ばれる [*1]。診断は行き当たりばったりではない、科学的なデバッグを遂行する上で不可欠な要素である [1]。

診断を下すためには実行時の状態を観察し、その正不正を判断する作業が必要不可欠である。現在実用化されているデバッガを用いれば、プログラム行に**ブレークポイント**を指定して実行を一時停止させ、その時点での呼び出しや変数値を参照する事が可能である。既存のデバッガを用いた診断には三つの問題が挙げられる。まずブレークポイント以前の実行の状態を参照する事が不可能である。プログラムの不具合箇所のかなりの割合は、デバッガから参照出来ない、過去に実行されたコード内に含まれている [2]。ブレークポイント以前の状態を調べるために、その設定と実行が何度も繰り返される事は珍しくない。

ブレークポイントの適切な設定は診断の効率を左右する大きな要因である。これは本質的には「次にどの時点の状態を参照するのか」という問題である。これは診断の作業効率を左右する重要な要因であるにもかかわらず、デバッガによる直接的な支援が実現されていない事が二つ目の問題点である。

三つめは状態の正不正の判定の問題である。参照すべき実行時点が特定されたとして、次にその状態に不正が含まれているか否かを判定しなければならない。しかしながら、こうした判定は必ずしも容易ではない。既存のデバッガはこの判定作業に関しても直接的な支援を与えない。結果として、調査すべき実行時点の特定や、現在参照されている状態の正不正の判定のために、さらなるブレークポイントの設定と再実行が繰り返される事になる。

二つ目と三つめの問題は実質的にはプログラム理解の問題に属するものである。現在多くのシステムがオブジェクト指向言語を用いて実装されているが、オブジェクト指向

[1] 奈良先端科学技術大学院大学
NAIST, Ikoma, Nara 630–0101, Japan
[2] 同志社大学
Doshisha University, Kyotanabe, Kyoto610–0394, Japan
[3] 神戸大学
Kobe Uniersity, Kobe, Hyogo 657–0013, Japan
[4] 東京大学
The University of Tokyo, Bunkyo, Tokyo, 113–8658, Japan
[a)] kume@is.naist.jp
[b)] khatano@mail.doshisha.ac.jp
[c)] masa-n@cs.kobe-u.ac.jp
[d)] etsuya@ecc.u-tokyo.ac.jp
[*1] 不具合、感染、障害、診断は文献 [1] の巻末の定義に従っている。

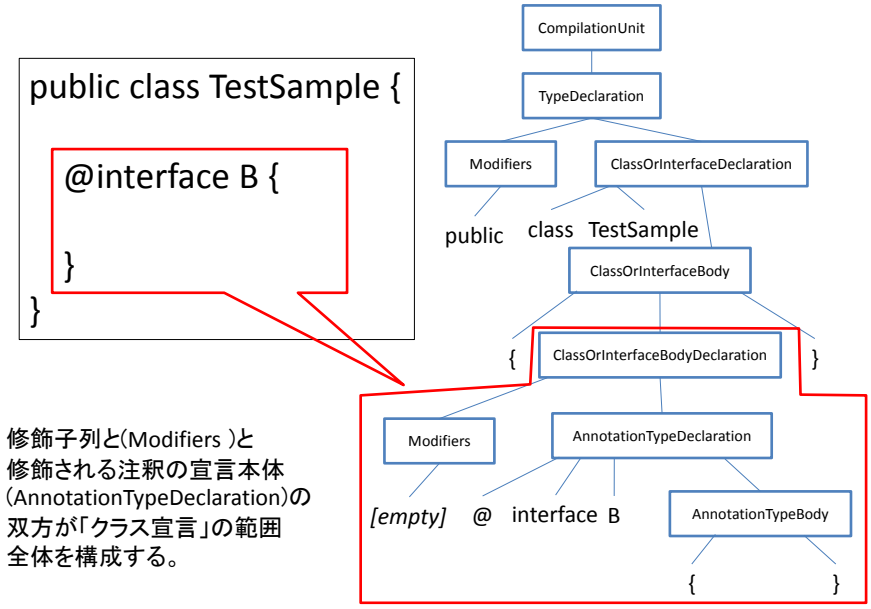

図 1 構文木の例
Fig. 1 Syntax Tree Example

プログラムには再利用性や拡張性を実現する代償としてプログラムが複雑化する傾向が有り、これが理解の妨げとなっている [3]。またプログラムの理解は属人性が高く、同じプログラムのデバッグであっても作業者毎にその知識や実行の認識が異なっている。

現在、過去の状態の参照を可能とする新しいデバッガとして所謂逆回しデバッガ (back-in-time debugger) 或いは全知デバッガ (omniscient debugger) と呼ばれる新しいデバッガ [4], [5], [6] の研究が進められている。また可視化や検索によるプログラム理解の支援 [7], [8], [9], [10] やより柔軟なブレークポイントの設定方式 [11], [12], [13] が研究されている。

しかしながら上記の支援手法はオブジェクト指向プログラム固有の複雑性と理解の属人性の問題の解決には十分な効力を発揮しない。我々はこれらの問題に有効に対処する新しい診断手法、**外挿診断法** *(diagnosis by extrapolation)* を提案し、それを実現する逆回しデバッガを開発中である。デバッグ対象言語としては Java プログラムを対象としている。

外挿診断法はデバッグ時に作業者が観察する内容を作業者自身にとって既知のプログラム要素 *2 や実行時のデータ *3 と関連付ける事によって、従来の手法には無い形の支援を実現する。まず、プログラム実行はこれらの情報に基づいて抽象化される。この抽象化によって、個々の作業者毎に最適化された形でオブジェクト指向プログラム固有な複雑性の問題が解決される。

*2 アーキテクチャ、クラス、メソッド、インスタンス変数、或いは特定のメソッドのパラメータや局所変数
*3 デバッガを通じて参照する特定のオブジェクトやメソッドの呼び出し、或いはある実行時点に於ける特定の命令の実行

さらに各実行時点での変数の値やメソッドの呼び出しも抽象化された実行過程中に位置付けられる。これによって個々の作業者に適した形で状態の正不正判定の支援が実現される。最後に作業者が次にどの時点の実行を調査すべきかをその既知情報からデバッガが推測して候補を呈示する。こうした候補の呈示には過去我々が**兆候解析** *(symptoms analysis)*[14], [15] と呼んだ技術が含まれている。

本論文ではオブジェクト指向固有の複雑性とそのデバッグに於ける問題を、我々自身が遭遇した不具合事例を用いて説明する。次に外挿診断法の基本概念を述べ、開発中のデバッガの機能を説明する。基本概念には (1) 外挿診断法の作業モデル、(2) 作業者の入力に基づく抽象化の原理、そして (3) 作業者とデバッガ間のやりとりが含まれる。これらと関連付ける形でデバッガの機能が説明される。さらに手法の有効性を実証するための評価実験が論じられる。

本稿の以降の部分は以下の通りに構成される。第 2 節では不具合事例が紹介される。提案手法とその評価実験計画がそれぞれ第 3 節と第 4 節で述べられる。第 5 節では関連研究が紹介され、将来課題と結語が第 6 節で述べられる。

2. 不具合事例

2.1 デバッグ対象プログラム

本節では我々自身が体験したオブジェクト指向プログラムの不具合とそのデバッグ作業を紹介する。本例題プログラムは Java のソースコードからクラス、インタフェイス、注釈 (annotation) の宣言部分を特定する構文解析器である。ソースコード上で宣言が占める範囲はその開始と終了の行と列の番号によって表現される。また Java の宣言は入れ子構造を許しているために、構文解析によって宣言の

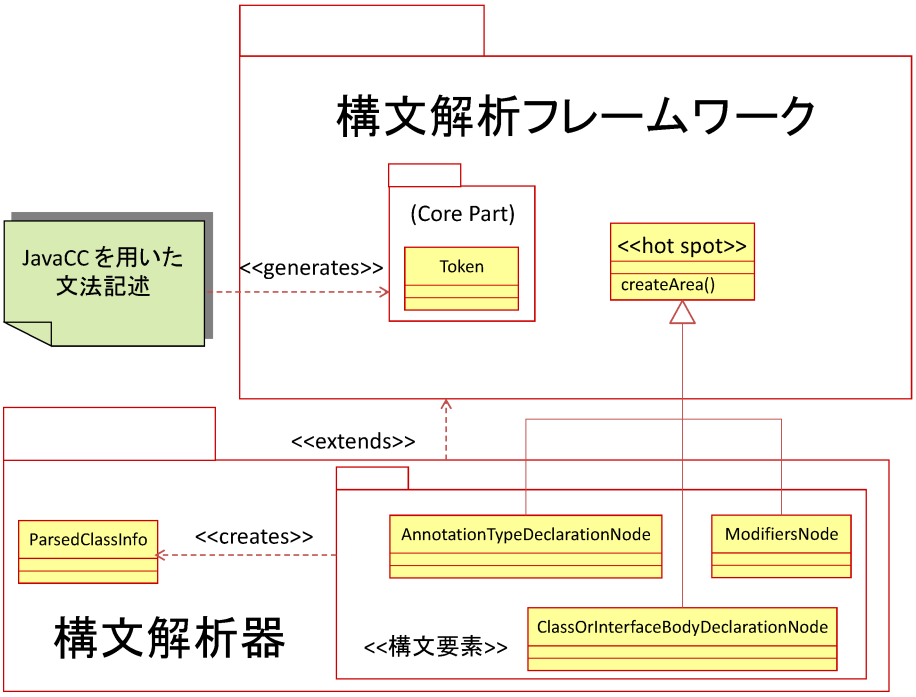

図 2 Java 構文解析器
Fig. 2 Java Parser Program

入れ子構造が特定される。

図 1 に Java のソースコード例とその構文木を示す。このソースコードではクラス `TestSample` が宣言されている。このクラスの宣言内に注釈 `B` の宣言が含まれている。注釈 `B` の宣言部分と対応する構文木が赤枠で囲まれている。一般に入れ子の宣言は修飾子列に続いて宣言本体が続く。注釈 `B` の宣言の場合、空の修飾子列が節点 `Modifiers` に、「`@interface`」で始まり「}」で終わる宣言の本体が `AnnotationTypeDeclaration` に適合する。

一般に注釈の範囲は修飾子列 (注釈 `B` の場合は偶々空であるが) とそれに続く宣言本体の解析が完了した時点で特定される。構文解析木の用語で述べると、`Modifiers` に続いて `AnnotationTypeDeclaration` がソースコード上の記述と適合した時点で注釈の宣言全体の範囲が特定出来る。よって注釈 `B` の宣言範囲の計算は両者を含む `ClassOrInterfaceBodyDeclaration` の解析処理時に実施されるべきである。

構文解析器のアーキテクチャを**図 2** に示す。構文解析器は Java の文法解析フレームワークのアプリケーションとして実装されている。フレームワークは字句解析と構文解析を実行する核心部分と、構文要素を表現するクラスから構成される。後者は核心部分から呼び出される hot spots[16] を表現している。核心部分のコードは Java 5 の文法記述から JavaCC[17] を用いて生成される。核心部分は指定された Java のソースコードを読み込み、字句解析と構文解析を実行する。

ファイルから読み込まれた字句はクラス `Token` によって表現される。各 `Token` インスタンスには読み込まれた字句そのものを表現する文字列と字句がソースコード中に占める範囲 (字句が開始・終了している行と列の番号) が記録されている。各構文要素に対する解析の開始時と終了時に Hot spot に対する呼び出し [*4] が実行され、`Token` インスタンスが渡される。Hot spot クラスには自身がソースコード中に占める範囲を計算するメソッド、`createArea()` が実装されている。この計算には各 hot spot のインスタンスにそれまで渡されていた `Token` インスタンスの位置情報が用いられる。位置情報が計算されると、その内容を記録するオブジェクトが生成され、メソッドの値として返される。

構文解析器のアプリケーション固有部分には上記 hot spot クラスを拡張したクラスが実装されている。これらは構文要素に関するアプリケーション固有な処理を実装している。図 2 中には `ClassOrInterfaceBodyDeclaration`、`Modifiers`、そして `AnnotationTypeDeclaration` に対応するアプリケーション固有部分のクラスが示されている。これらのクラスと構文要素の対応はそのクラス名から明らかである。これら構文要素を表現するクラスに加えて、宣言に関する情報を表現するクラス `ParsedClassInfo` が実装されている。このクラスのインスタンスには宣言されているクラスや (注釈を含む) インタフェイスの名前、入れ子になっている宣言、そして宣言がソースコード中に占める範囲を表現するオブジェクトが保持される。

この構文解析器が図 1 の注釈 `B` の宣言に対して作成する

[*4] このメソッドの表示は図 2 中では省略されている。

ParsedClassInfo インスタンスには占有範囲が設定されない、という不具合が存在する。B の宣言を構成する (空の) 修飾子列と、それに続く宣言本体の解析が完了した時点で初めてこの宣言全体の占有範囲が特定される。よってこれに続く ClassOrInterfaceBodyDeclaration の処理の中で占有範囲オブジェクトが生成され、B に対応する ParsedClassInfo のインスタンス変数に代入されるべきである。

しかしながらこの構文要素に対応する ClassOrInterfaceBodyDeclarationNode のメソッド中にこの処理が含まれていない。よってこのインスタンス変数の値は null 値のままなのである。テストケースの中で、構文解析器が生成した注釈 B の ParsedClassInfo インスタンスを作成し、そのインスタンス変数から参照されている占有範囲オブジェクトのメソッドを呼び出そうとすると例外 NullPointerException が投射される。

2.2 デバッグ作業

構文解析器の不具合は設計の不正、所謂**不備** *(flaw)* に分類される。この不備を正しく診断するためには以下に述べる二つ事項の確認が必要であった。まず、オブジェクトのインスタンス変数に値が代入されておらず、これが例外の直接の原因である事。次にこのオブジェクトが生成された後にそのイン占有範囲オブジェクトを作成してスタンス変数に代入すべきクラスを特定し、必要な処理が実装されていない事である。

一つ目の事項はプログラムの依存関係を辿る事によって確認される。例外の直接の原因は null 値に対するメソッド呼び出しの試みである。ここでデータの流れを逆に辿る事により、この null 値がある ParsedClassInfo オブジェクトのインスタンス変数の値である事が判明する。このオブジェクトが生成されて以来、該当のインスタンス変数に値が代入されない事はその操作履歴を調べる事によって確認出来る。こうした作業の遂行そのものには、値や操作について、このプログラム固有な意味と関連付けて理解する事は必要とされない。

二つ目の事項は単純に依存関係を調べるだけでは判明しない。本来であれば記載されるべきメソッド createArea() の呼び出し命令がそもそも記述されてない。これは記載されているが実行時の状態の誤りのために呼び出しが実行されない、所謂**実行欠落不正** *(execution omission error)*[18] よりも支援の自動化が困難である。

この問題はプログラム内部の挙動を図 1 に示される構文解析と関連付け、それに基づいて各クラスの役割を特定する事によって解決される。しかしながら図 1 に示される木構造が実行時のメソッド呼び出し構造のような、直接分かり易い形で実現されているわけではない。このような概念と処理の表現上の隔りを埋める鍵を握るのが字句を表現する Token のインスタンスである。

一般に構文解析は本質的にソースコード上の字句列から構文規則を選択・適用する処理である。ソースコードはフレームワークの核心部分で読み込まれ、字句を表現する Token インスタンスを作成する。Token インスタンスの内容に応じて構文規則を適用し、その適用の開始と終了時に hook メソッドの呼び出しを行う。このように、Token と hook メソッドの間には、イベント駆動プログラミングに於けるイベントと呼び戻しメソッドと同様な関係が構築されている。

こうした制御に関する役割に加えて、Token インスタンスにはソースコード上の範囲情報を伝達する役割も果たしている。個々の Token にはその語句がソースコード中に占めている範囲が記録されている。これらの字句はアプリケーション固有部分にメソッドの引数として渡される。アプリケーション固有部分は各構文要素を構成する Token インスタンスを集積し、最初と最後の字句の情報から全体の範囲を特定する。

このように Token はそのインスタンスこそフレームワーク内部で生成されるが、字句という高度な概念を表現するクラスであり、その構造も単純で理解が容易である。何より JavaCC の文書にもこのクラスに関する説明が記載されている。各 Token インスタンスの内容をデバッガで参照する事により、対応する字句が明確に特定出来る。よって Token は作業者にとって既知のプログラム要素であり、その全てのインスタンスも既知の実行時データである。このような作業者にとって既知の情報と実行を結び付ける方針こそが、不備の所在の特定に決定的な役割を果たしのである。

一般に診断を下す作業では「本来あるべきであった実行内容」を理解し、現実の実行と比較する事によって重要な手掛かりが得られる事が指摘されている [1]。このような比較を行う事によって、不正な状態が発生した瞬間を特定出来ると期待されるからである。

この作業で実行されたテストケースでは注釈 B を包含するクラス TestSample は正常に処理されていた。TestSample の占有範囲が決定される過程を Token インスタンスと関連付けて観察した。その結果、修飾子列と宣言本体の親である構文要素が占有範囲の設定を実施している事が判明した。注釈 B の処理でこれに相当する ClassOrInterfaceBodyDeclaration はこうした処理を行っていない事が確認された。この処理を行うコードを追加したところ、テストケースが正常に実行された。以上の作業過程を経て不備の解消が確認された。

2.3 デバッグの障害

上記の作業は現在実用化されている中では標準的な機能を備えた Eclipse のデバッガを用いて遂行された。作業で

は多大な労力が費されたが、その要因を列挙すると以下の通りである：

ブレークポイント設定の制約　上記作業例ではデバッガで参照した特定のインスタンスに対する操作や参照の観察が必要とされた。しかしながら特定のオブジェクトを指定してブレークポイトを設定する機能が存在していないため、代替的な手段を採用せざるを得なかった。

仮説下での作業　実用的なプログラムの全ての実行過程をデバッガを用いたステップ実行によって確認する事は非現実的である。作業者は実行のかなりの部分を実際に追う代わりに、コードを眺めたり呼び出し関係を調査しながら推測せざるを得ない。結果として仮説の上に仮説を重ねる不確かな根拠に基づいた作業が避けられない。

理解の鍵となるクラスの所在　`Token` クラスとそのインスタンスが不備の所在を特定する上で決定的な役割を果たした。しかしながら作業者は当初からこれらの情報を利用しようとする見通しを持っていたわけではなく、プログラムの実行を愚直に追う作業を通じて偶然その重要性に気が付いたに過ぎない。このような重要な手掛かりの発見が作業者の技量のみに依存する現状は、デバッグの体系化が困難な道程にある事を示すものである。

フレームワークの実装　`Token` とそのインスタンスをプログラム実行と関連付けるため、フレームワーク側からの hook メソッドを呼び出す時機を調査した。フレームワークの詳細な実行の調査は必要無いにもかかわらず、ブレークポイントの設定箇所を特定するためにフレームワークの実装を調べなければならなかった。本来こうした作業を必要としないようにする事がフレームワークの趣旨であった筈なのに、デバッグではそれが避けられなかった。JavaCC のコードは文法記述から自動生成された事がこの作業をより一層困難なものにした。

オブジェクト指向固有の設計と言語機能　図 1 が示す構文木にクラス継承階層、多相型、動的束縛、Factory 等再利用のための設計 [19]、短かく多数のメソッドから構成される呼び出し関係がプログラムの実行を調査する労力を増大させた。こうした要素がソースコードの読解を困難にする事は周知の事実である。もちろんデバッガを用いてオブジェクトの型を調べながらプログラムをステップ実行すれば、必要な情報は入手出来る。しかしながらこうした煩わしい非効率な操作を伴う調査は作業者に多大な忍耐を強いるものであり、決し望ましいものではない。

上記の要因のうち、初めの二つは本質的にはツールとしてのデバッガの機能の問題と捉えられる。ブレークポイントの設定に関してはソースコード行に基づかない新しい設定手法 [11], [12], [13] が研究されている。またプログラムの依存関係の追跡に関しては、デバッガに対する煩雑な操作から作業者を解放するユーザインタフェイス [8], [20] が提案されている。

残りの三つの問題は作業者によるプログラムの知識や、作業者が現在注目している実行の側面と大きく関わっている。デバッグ作業者は Eclipse の Java 検索機能を活用し、依存性を追いながら様々なクラスを参照する過程を経てプログラムを理解しようとする事が知られている [21]。これらの問題を支援する既存の手法の中でこうした作業者の知識や関心に合わせて適切な支援を与える物は我々の知る限り存在しない。作業者の知識の増大や関心の変遷に合わせて、理解の鍵となる情報を呈示し、これとプログラムの実行を関連付ける事によって作業者の理解を支援する新しい手法が必要とされている。

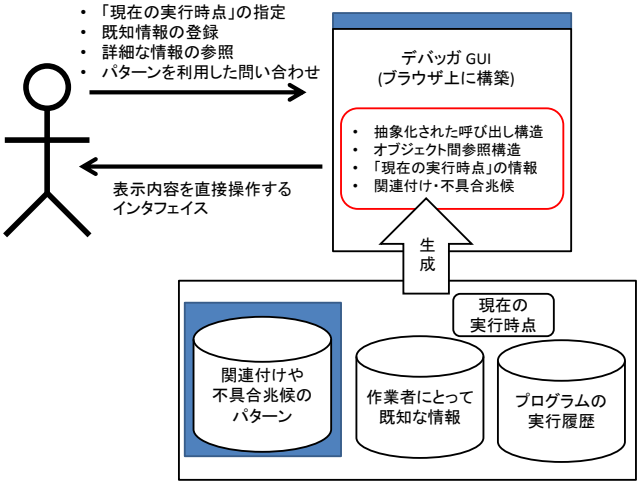

図 3　基本概念
Fig. 3　Basic Concepts

3. 提案手法

3.1 デバッグの基本概念

我々は第 2.3 節の考察を元に逆回しデバッガの存在を前提とした新しい診断方手法である**外挿診断法**を提案する。**外挿**という語は既知情報と関連付けによってプログラムの意味を理解する着想に付されている。外挿診断法ではプログラム実行はメソッド呼び出しの木として形式化される。これは所謂アルゴリズムデバッギング [22], [23] に於ける**計算木**に相当する。診断の目的はメソッド単位で最初の感染の発生箇所と関連の連鎖を特定する事にある。ただし、我々の診断はアルゴリズムデバッギングのそれとは以下の二点で異なっている。

まず調査すべき木の節点 (メソッド呼び出し) は作業者自身が選定する。デバッガは作業者にとって適切な節点を選択するための支援を目的とする。次に、作業者は各節点の実行結果に関する正不正だけでなく、各節点で参照される

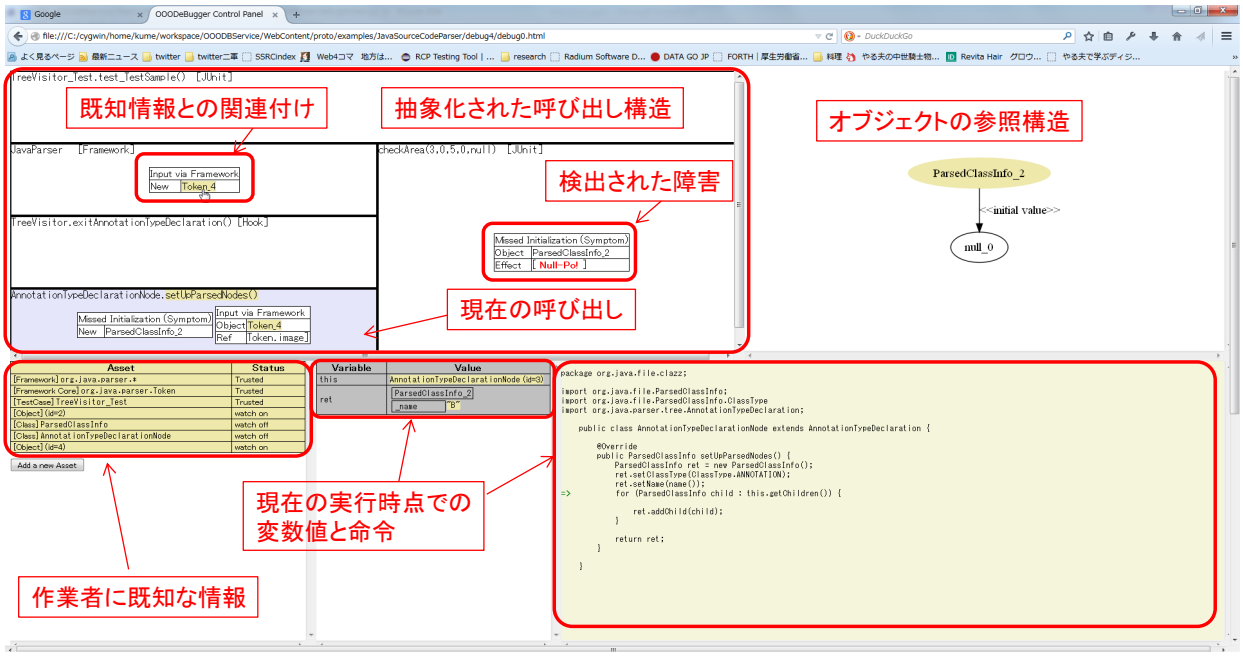

図 4 デバッガの表示画面
Fig. 4 Debugger with Panes

データ構造に関する正不正の指定も出来るようにする。これによってプログラムの不備によって障害が発生する過程を明らかに出来るようにする。

診断を遂行するために作業者は**図 3** に示すデバッガとのやりとりを行いながら不具合箇所とその感染の経路を特定する。デバッガは作業者にとって既知なプログラム要素や作業者が関心を示している実行時のデータを記録している。また作業者が注目している「現在の実行時点」も記録している。

デバッガはプログラムの実行に関して詳細な情報を記録したプログラムの実行履歴を保存している。この実行履歴は文献 [24] に述べられた動的スライスと同等の粒度のデータ構造を有しており、我々が過去に開発した動的解析ツール [14], [15] を用いて取得・保存される。

デバッガにはこれに加えて、**不具合を示唆する挙動 (兆候)** や我々が有用と判断した**関連付けのパターン**をデータとして保持する。不具合を示唆する挙動の幾つかは我々の過去の研究 [14], [15] でも開発されている。デバッガはこれらの情報に基づいて、現在の参照時点の詳細な情報 (現在実行されているソースコード行と変数の値) を表示する。さらにメソッドの呼び出し構造の中でそれが実行されている箇所を抽象化された形で表示する。さらに作業者が関心を有しているオブジェクト同士の参照構造も表示する。

さらに現在表示されている内容から不具合を示唆する兆候や関連付けを辿れる場合には、該当する内容からその詳細を調査するためのユーザインタフェイスがデバッガから提供される。作業者がこのユーザインタフェイスを利用してて、兆候や関連付けを選択すると、それに応じて画面の表示が再構成される。作業者は兆候を選択する事によって感染の連鎖の候補を調査する。また関連付けを選択する事によって、第 2.2 節で述べた Token とそのインスタンスに相当する手掛かりを取得出来る。こうした支援によって作業者が「次に調査すべき箇所」を効率的に決定出来ると期待される。

作業者は既知のプログラム要素や自身が関心を有する実行データと結び付いた兆候や関連付けの検索も行える。例えば第 2.2 節で述べた Token インスタンスはフレームワーク内部で処理された外部入力データをアプリケーション固有部分に伝達する役割を果たしている。これに加えてこれらのインスタンスは hook メソッドの呼び出しの条件も与えている。前者の関連付けパターンによって Token インスタンスを取得し、取得したインスタンス或いはそのクラスの別のインスタンスが関与する関連付けをデバッガに列挙させ、その詳細を表示させる事も可能である。

3.2 ユーザインタフェイス

我々は現在、第 2.3 節で説明したデバッグを遂行するための逆回しデバッガを開発中である。本デバッガの実行画面を**図 4** に示す。図に表示された画面は GUI の評価のために作成されたプロトタイプであり、将来変更される可能性がある。

本デバッガの GUI は JavaScript を用いてウェブブラウザの上に構築されており、Java サーブレットとデータのやりとりを行う。GUI はサーブレットが作成した実行の抽象化表現から表示データを作成し、表示内容や既知情報から兆候や関連付けを参照するための GUI を作業者の操作に応

じて生成する。サーブレットは作業者からの操作を受けて抽象化表現を更新してブラウザに側に返す。JavaScript と Java 側のこうしたデータ交換は Direct Web Remoting[25] 技術を用いて実装されている。

図 4 に示されるように、デバッガの表示画面は五つの面に分割されている。表示画面全体は上下に二つに分割されている。上の画面はさらに左右に二つに、下の画面は横に三つにそれぞれ分割されている。左上の面はプログラム実行全体のメソッド呼び出しを抽象化した内容が表示されている。面を分割する矩形はそれぞれ一つ以上のメソッド呼び出しを表現している。矩形同士の上下関係は上側による下側の呼び出しを表している。右側の矩形は左側のそれよりも後に実行される。色が他と異なる矩形は「現在実行されているメソッド」を意味する。

矩形には呼び出しの頂点にあるメソッドのクラス・メソッド名・型といった情報の他に、メソッドが関与している兆候や関連付けに関する情報が表の表の形で表示されている。これによって呼び出し間の依存関係や感染の連鎖への関与を作業者の既知の情報に合わせた表示 *5 が実現されている。

右上には作業者が注目しているオブジェクトの参照関係が表示されている。ここで表示されている関係図は GraphViz [26] によって生成された HTML の Map 要素である。関係図のオブジェクト (節点) や参照関係 (辺) を直接操作してその詳細な情報 (クラスや変数名のような基本情報や操作が実行された時点) を参照出来る。

左下の面には作業者に既知のプログラム要素と作業者が関心を有している実行時のデータを登録する表が表示されている。前者はプログラムのクラスやメソッド、インスタンス変数、命令文や局所変数に及ぶ。後者はデバッガで参照されたオブジェクトやメソッド呼び出し、特定のメソッド呼び出しの中の値の参照や分岐命令を含む。作業者はアプリケーションフレームワークやテストケースを構成するパッケージを登録する事が可能である。フレームワークやテストケースのコードに関しては、その正しさを無条件で信頼するように指定出来る。

下側中央と右下の面にはそれぞれ現在の実行時点に於ける局所変数の値とこれから実行されるコード内の命令文が表示されている。ソースコード上の行を指定して実行を進めたり戻す事が可能である。この操作によってサーブレット側に「現在の実行時点」の変更が通知される。図中の局所変数値の一つが黄色く表示されているのはこの値の生成参照される過程が関与する兆候や関連付けが存在する事を示している。このように、局所的に参照される値からも調査対象候補の特定が可能である。

*5 表よりも直感的に理解し易い表示形式を検討中である。

4. 評価実験計画

デバッガの開発と同時に、我々はその評価実験の設計も進めている。評価実験の目的は、作業者の知識や関心に合わせた本デバッガの支援機能の有効性を評価する事にある。現在我々は二通りの評価方式を検討している。いずれの方式を採用する場合でも、評価のためのデータを得るために被験者によるデバッグ作業を実施する。

一つ目の評価方式では、同じデバッグ作業を支援機能を有するデバッガと有しないものでそれぞれ実施してその成績を比較する。支援機能を有しない版のデバッガとしては既存の逆回しデバッガ (例えば [20], [27]) を利用するか、本デバッガの表示を制限した版を利用する。逆回し機能の面で既存のデバッガと本デバッガの能力が異なる場合には、実験の公平さを実現するために本デバッガの機能制限版を利用すべきと考える。

二つ目の評価方式では作業者の使い勝手を評価する。作業者が必要とする情報が必要な時機に呈示されたか否か、作業者が望む情報を少ない操作量で取得出来たか否かを評価する。評価方式の場合、作業者の内面の表す情報を取得する必要がある。内面は作業者の行為に反映されるという立場では、デバッガに対する作業者の操作やコードの表示画面の注視等、作業履歴を取得して解析すべきと考える。

作業者の内面を表現するより直接的な方式として “think aloud”[28] と呼ばれるデータ採取法も検討中である。これは作業者にその思考内容を絶え間無く発話させ、発話内容を分類・解析する手法である。これに加えた作業の画面や操作を録画し、作業者自身にその意図を問い質すデータ採取法も検討中である。

5. 関連研究

逆回しデバッガ [4], [5], [6] の歴史は浅く、実用的なプログラムをデバッグする際の実行効率的に関してその実用性が疑問視する意見 [12] も見掛けられる。我々の過去の研究 [14], [15] で得た知見に依れば、通常の PC 上で実用的なプログラムを実行し、文献 [24] で扱われているものとと同等な、極めて詳細な内容を有する実行履歴の処理には数分を要する。我々の現在の実装は実行効率に関して何の工夫もなされておらず、今後の改善も期待出来る。通常のデバッガのようにリアルタイムでプログラムの実行停止を制御しないのであれば受け入れられる結果であると考えている。

近年では洗練されたユーザインタフェイスを備えた逆回しデバッガが実用化されている [8], [20], [27] 。Whyline [8], [20] は指定した命令文が「何故実行されたのか (或いはされなかったのか)」を作業者が問い合わせるための洗練されたユーザインタフェイスを実装している。プログラム

の実行時の依存関係を辿る際にこの機能は大いに効力を発揮している。JIVE はプログラム実行から UML の順序図を作成する、逆工程 (reverse engineering) 機能を有しており、Java のプラグインとして実用化されている。

6. おわりに

本稿では作業者の知識や関心に応じてオブジェクト指向プログラム固有のデバッグの問題解決を支援する新しいデバッグ方式を提案した。このデバッグ方式を実現する逆回しデバッガの機能とそれを用いた検証方式に関して議論した。

謝辞 この研究は栢森情報科学振興財団、科研費基盤 (A)24500352、基盤 (B)23300009 及び 26280115、基盤 (C)24500079、挑戦的萌芽 25540150 の助成を受けて遂行された。

参考文献

[1] Zeller, A.: *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, Morgan Kaufmann (2009).
[2] Liblit, B., Naik, M., Zheng, A. X., Aiken, A. and Jordan, M. I.: Scalable Statistical Bug Isolation, *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, New York, NY, USA, ACM, pp. 15–26 (online), DOI: 10.1145/1065010.1065014 (2005).
[3] Wilde, N. and Huitt, R.: Maintenance Support for Object-Oriented Programs, *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, pp. 1038–1044 (1992).
[4] Hofer, C., Denker, M. and Stéphane Ducasse: Design and Implementation of a Backward-In-Time Debugger, *Proceeding of NODe 2006*, Lecture Notes in Informatics, Vol. P-88, pp. 17–32 (2006).
[5] Lewis, B.: Debugging Backwards in Time, *International Workshop on Automated Debugging (AADEBUG)* (2003).
[6] Pothier, G., Tanter, Éric. and Piquer, J.: Scalable Omniscient Debugging, *OOPSLA*, ACM, pp. 535–552 (2007).
[7] Czyz, J. K. and Jayaraman, B.: Declarative and visual debugging in Eclipse, *OOPSLA Workshop on Eclipse Technology eXchange*, ACM, pp. 31–35 (2007).
[8] Ko, A. and Myer, B.: Finding Causes of Program Output with the Java Whyline, *SIGCHI: Human Factors in Computing Systems*, ACM, pp. 1569–1578 (2009).
[9] Lienhard, A.: *Dynamic Object Flow Analysis*, Lulu.com (2008).
[10] Lencevicius, R., Hölzle, U. and Singh, A. K.: Dynamic Query-Based Debugging of Object-Oriented Programs, *Automated Software Engineering*, Vol. 10, No. 1, pp. 39–74 (2003).
[11] Zhang, C., Yan, D., Zhao, J., Chen, Y. and Yang, S.: BPGen: An Automated Breakpoint Generator for Debugging, *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, New York, NY, USA, ACM, pp. 271–274 (online), DOI: 10.1145/1810295.1810351 (2010).
[12] Ressia, J., Bergel, A. and Nierstrasz, O.: Object-Centric Debugging, *International Conference on Software Engineering*, IEEE, pp. 485–495 (2012).
[13] Chern, R. and Volder, K. D.: Debugging with control-flow breakpoints, *International Conference on Aspect-Oriented Software Development*, ACM, pp. 96–106 (2007).
[14] Kume, I., Nitta, N., Nakamura, M. and Shibayama, E.: A Dynamic Analysis Technique to Extract Symptoms That Suggest Side Effects in Framework Applications, *Symposium On Applied Computing*, ACM, pp. 1176–1178 (2014).
[15] Kume, I., Nakamura, M., Nitta, N. and Shibayama, E.: Toward a dynamic analysis technique to locate framework misuses that cause unexpected side effects, *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on*, pp. 1–6 (online), DOI: 10.1109/SNPD.2014.6888730 (2014).
[16] Pree, W.: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley (1994).
[17] Java Compiler Compiler: https://javacc.java.net/.
[18] Zhang, X., Tallam, S., Gupta, N. and Gupta, R.: Towards Locating Execution Omission Errors, *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, New York, NY, USA, ACM, pp. 415–424 (online), DOI: 10.1145/1250734.1250782 (2007).
[19] Gamma, E., Helm, R., Johnson, R. and Vissides, J.: *Design Patterns*, ADDISON-WESLEY (1994).
[20] Ko, A. J. and Myers, B. A.: Extracting and Answering Why and Why Not Questions About Java Program Output, *ACM Trans. Softw. Eng. Methodol.*, Vol. 20, No. 2, pp. 4:1–4:36 (online), DOI: 10.1145/1824760.1824761 (2010).
[21] Ko, A., Myers, B., Coblenz, M. and Aung, H.: An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks, *Software Engineering, IEEE Transactions on*, Vol. 32, No. 12, pp. 971–987 (online), DOI: 10.1109/TSE.2006.116 (2006).
[22] Shapiro, E. Y.: *Algorithmic Program DeBugging*, MIT Press, Cambridge, MA, USA (1983).
[23] Caballero, R., Hermanns, C. and Kuchen, H.: Algorithmic Debugging of Java Programs, *Electronic Notes in Theoretical Computer Science*, Vol. 177, No. 0, pp. 75 – 89 (online), DOI: http://dx.doi.org/10.1016/j.entcs.2007.01.005 (2007).
[24] Wang, T. and Roychoudhury, A.: Using Compressed Bytecode Traces for Slicing Java Programs, *International Conference on Software Engineering*, IEEE, pp. 512–521 (2004).
[25] Direct Web Remoting: http://directwebremoting.org/dwr/index.html.
[26] GraphViz: http://www.graphviz.org/.
[27] Czyz, J. K. and Jayaraman, B.: Declarative and Visual Debugging in Eclipse, *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, New York, NY, USA, ACM, pp. 31–35 (online), DOI: 10.1145/1328279.1328286 (2007).
[28] Karahasanović, A., Levine, A. K. and Thomas, R.: Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study, *Journal of Systems and Software*, Vol. 80, No. 9, pp. 1541–1559 (2007).