修 士 論 文

ソースコードの依存解析と視覚化手法に関する研究

指導教官　落水 浩一郎 教授

審査委員主査　落水 浩一郎 教授

審査委員　篠田 陽一 助教授

審査委員　二木 厚吉 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

80122　横山浩一

2001 年 2 月 15 日

要 旨

本稿では、オブジェクト指向言語のソースコードを対象とした、依存解析手法について述べ、コード構造の抽象度をスムーズに変化させることによって、ユーザのメンタルモデルを崩さないブラウズシステムの設計と、プロトタイプ実装を行なう。

目 次

図 目 次

第 1章

はじめに

1.1 背景

過去に作成したソフトウェアの仕様変更や修正、移植などの理由で、コードを読む機会は失われていない。その上、ソフトウェアの巨大化や生産コストの削減のために、既存のコードを再利用したりRADユーティリティを利用することが増加しており、自動生成されたコードを読む必要に迫られる場合がある。そこで、、オブジェクト指向言語においては、ソフトウェアの階層化・モジュール化による情報隠蔽が行なわれ、「関心の分離」がなされているが、それでもなお複雑なクラス構成によって、ソフトウェアの全体的な構造や振舞を把握する事は難しい。

このような場合に、設計・実装時に用いた仕様書等が揃っていれば、そこからソフトウェアの構造を把握することが可能なはずである。しかし、仕様書の情報が更新されておらず古い場合や、そもそも仕様書が残っていないといった状況が多々あり、全体的な構造の把握や正しい認識を難しくしている。

そこで、ソースコード自身からソフトウェア仕様を得ようという試みが行なわれており、クラスビュア・構文エディタ機能を備えたRADユーティリティや、クラスライブラリ理解促進ツールは、図形や色彩といった文字以外の要素を利用して視覚化し提示することにより、情報の認識・取得を補助している。

しかし、これまでの手法では、視覚化する際の抽象度がいくつかの段階に固定されており、ユーザの認識にあわせて大局的な情報と局所的な情報との間で、柔軟に呈示する事が出来ない。例えばクラスビュアでは大局的なクラス構造のみであり、構文エディタでは局所部分の依存関係違反のみを追跡するだけである。

これでは、固定された抽象度の情報を何の関連性も示される事なく切り替えて参照する

事となってしまう。大規模なソフトウェア開発や、他のモジュールの詳細が隠匿されている中で、お互いに参照する事の多いオブジェクト指向プログラミングにおいて、ユーザは構造の把握や認識と詳細部の間等で整合がとれなくなる可能性がある。結果として、複数の情報から改めて 1 つのメンタルモデルを構築する事を強いられる事になる。場合によっては、自身で情報を再構築する事ができず、それぞれの情報の間に何の関連もない別々の情報としてしか利用できない状況も考えられる。そこで、複数の情報を呈示する場合に、切替えによる認識の不整合を防止する手法が必要となる。

1.2 目的

本研究では、オブジェクト指向言語である Java のソースコードにおける、構成要素の可視化と、ユーザの要求にあわせて大局的な情報から局所的な情報までを段階的に変化させて呈示できるシステムとして、SZ_Browser の設計を行う。

これによりユーザは着目点が全体に対して、どの様な関連を持っているのかを見失いにくくなり、ソースコード理解に要する時間の短縮や、正しい理解に有効であると考る。

そこで、粒度を必要に応じて変更できるユーザー・インタフェースを用意し、細粒度から粗粒度まで可変的に視覚化する手法を提案する。これによってテキストベースのソースコードからは困難だった全体像の把握が直感的に行え、クラスビュア等では難しかったコードの詳細と全体像との関連の把握が可能となる。

1.3 構成

本論文の構成について以下に示す。

- 第 1 章では、研究の背景と目的について述べる。
- 第 2 章では、オブジェクト指向言語である Java のプログラムソースを対象とした構成要素間の依存関係の定義と、これを計算機に解析・呈示させる事の有効性について述べる。
- 第 3 章では、基礎である情報視覚化一般について述べた後、ソースコードおよびテキストデータの視覚化研究について紹介する。
- 第 4 章では、ソースコード視覚化技術の現状について、既存のツールの紹介と不満点を述べ、次に述べるソースコード視覚化システムに対する要求を示す。

- 第 5 章では、さきに述べた依存関係と視覚化手法をふまえた上で、Java ソースコードの抽象度を変化させながら呈示するシステムを設計し、プロトタイプの実装について述べる。
- 第 6 章では、本研究のまとめと今後の課題について述べる。

第 2章

依存解析

プログラムの依存解析には、注目点と関連の強い箇所を見つけ出し認知・把握すべき情報を絞る事により、必要な箇所を重点的に作業する事ができるなどの利点があり、開発工程や保守工程においてさまざまな形で用いられている。

本章では、オブジェクト指向言語である Java のプログラムソースを対象とした構成要素間の依存関係の定義と、これを計算機に解析・呈示させる事の有効性について述べる。

2.1 依存関係

ここで言う依存関係とは構成要素間に継承・参照等の意味をもった関係が存在する事であり、依存解析とはソースコードよりその関係を解析・取得する事である。これを用いる事により仕様書などが無い、もしくは情報が十分でない場合などに、それを補う情報として活用する。

2.1.1 単純な関係

クラス継承関係

オブジェクト指向言語では、すでに存在するクラスを継承し利用する事は基本的な事柄である。この際、すでに存在するクラスとそれを継承し利用するクラスの間には、継承する・されるという関係が発生する。Java 言語の場合どのクラスを継承するかは`extends`と`implements`でクラスを定義する際に指定する。したがって、継承するクラス定義から継承されるクラス名を読み取る事は簡単である。

```
class New_class extends Extend_class implements Implement_class {
  ⋮
}
```

図 2.1: クラス定義における継承の指定例

図 2.1 の場合、*New_class* クラスは *Extend_class* クラスと *Implement_class* クラスを継承することがわかる。

しかし、*Extend_class* クラスがどのクラスから継承されているのかを読みとることはそれほど単純ではない。たとえば図 2.1 の定義からは、*Extend_class* クラスは少なくとも *New_class* クラスから継承されていると言う事しか読み取る事はできない。つまり、一つのクラス定義からは定義しているクラスが目的のクラスを継承しているか否かを調べる事しかできない。これは、あるクラスがある一定のクラス群に含まれているどのクラスから継承されているのかを調べる事にはすべてのクラスに対して目的のクラスを継承しているかを調べる必要があると言う事であり、クラス数の多さに比例して増大する手間により、開発などの作業中に人間が取得する事は難しい。

また、クラス間の継承関係についての概観を双方向に呈示する事は重要であり、すべてのクラスに対して目的のクラスが継承されているかを調べるといった、単純で反復回数の多い作業は計算機の得意とする作業であるため、計算機による解析・呈示を行なう事は有効である。

利用関係

メソッド・フィールドの利用・参照はクラス継承関係ほど単純ではない。**クラス名**.**フィールド名**等の一意に特定できる形で利用・参照されていれば実際にメソッド・フィールドを定義している箇所をわずかなコストで特定可能である。

```
class HelloWorld {
 void hello(){
  java.io.PrintStream o = java.lang.System.out ;
  o.println( "Hello World" ) ;
 }
}
```

図 2.2: 明示的なフィールド・メソッドの指定例

図 2.2 の場合、java.lang.System クラスのフィールドである *out* を参照している事を読みとる事ができる。また、java.io.PrintStream クラスのメソッドである *println()* をインスタンスo を介して利用している事を読みとる事ができる。

また、明示的に指定されていなくとも、比較的簡単に参照・利用関係を読みとる事ができる場合もある。

```
class HelloWorld {
 void hello(){
  System.out.println( "Hello World" ) ;
 }
}
```

図 2.3: 非明示的なフィールド・メソッドの指定例

図 2.3 の場合、System クラスが実際にはjava.lang.System クラスである事をクラスパスに従って検索し、そのフィールドである *out* を参照している事が得られる。次に java.lang.System クラスの定義より *out* フィールドはjava.io.PrintStream クラスのインスタンスを返す事が判る。最後に、java.io.PrintStream クラスのメソッドである *println()* ををを参照している事を読みとる事ができる。

しかし、オブジェクト指向言語である Java ではメソッドをサブクラスでオーバーライドする事ができ、ポリモルフィズムが用いられている場合がある。その場合、利用される実装は実行時に決定されるため行なわれるため、静的解析では実際にどのクラスの実装が利用されるのか特定できない。

```
 1: interface Printable {
 2:   void print();
 3: }
 4:
 5: class Base implements Printable {
 6:   protected String message = null;
 7:   public void print(){
 8:     System.out.println(message);
 9:   }
10:   public void message(String s){
11:     message = "Base Class: "+s;
12:     print();
13:   }
14: }
15:
16: class Ride extends Base {
17:   public void message(String s){
18:     message = "Ride Class: "+s;
19:     print();
20:   }
21: }
22:
23: public class Polymorphism {
24:   static void vcall1(Base base){
25:     base.message("Hello World for Base");
26:   }
27:   static void vcall2(Printable obj){
28:     obj.print();
29:   }
30:   public static void main(String[] args){
31:     Ride ride = new Ride();
32:
33:     vcall1(ride);
34:     vcall2(ride);
35:     ride.message("Hello World for Ride");
36:
37:     Base base = new Base();
38:
39:     vcall1(base);
40:   }
41: }
```

図 2.4: ポリモルフィズムが用いられている例

図 2.4 の場合、Polymorphism クラスのメソッドであるvcall1(Base) ではbase は必ずしもBase クラスの実装を伴ったインスタンスであるとは限らない。事実、33 行目によって実行される/verb+vcall+メソッドはBase クラスのインスタンスをパラメータとして受け取っているが実際にはBase クラスを継承するRide クラスの実装でオーバーライドされたインスタンスを受けとるのでRide クラスの実装が利用される (図 2.5 2 行目)。一方、39 行目によって実行されるvcall メソッドはBase クラスのインスタンスを実際に受け取っているのでBase クラスの実装が利用される (図 2.5 5 行目)。

```
1: % java Polymorphism
2: Ride Class: Hello World for Base
3: Ride Class: Hello World for Base
4: Ride Class: Hello World for Ride
5: Base Class: Hello World for Base
6: %
```

図 2.5: Java プログラム Polymorphism の実行結果

つまり、vcall1 メソッドにおいてはmessage は基本となるBase クラスの実装を呼び出す場合の他に、少なくともRide の実装を呼び出す可能性がある事が判る。

次に、34 行目によって実行されるvcall2 メソッドについて見てみるとprintable インタフェースのインスタンスを受けとり、print メソッドを利用しているが、printable はインタフェースであり、実装を持たないため必ずいずれかのクラスによって実装が行なわれたインスタンスを受けとっているはずである。この例の場合はRide クラスによって実装されたprint メソッドを呼び出している (図 2.5 3 行目)。

つまり、vcall2 メソッドにおいてはprint は基本となるPrintable の実装は (実際問題として存在しないので) 決して呼び出さず、それ (Printable) を実装したものが呼び出される。

最後に、35 行目によって実行されるride インスタンスのmessage メソッドは必ずRide クラスの実装が呼び出される (図 2.5 4 行目)。

このように、ポリモルフィズムが用いられている状況でのメソッド呼び出しは 3 種類に分類できる。

1. 基本となるクラスとその系統のクラスの実装のいずれかを呼び出す
2. 基本となるクラスは決して呼び出さずその系統のクラスの実装のいずれかを呼び出す

3. 基本となるクラスの実装を必ず呼び出す

そして、1 と 2 に分類される場合は呼び出される可能性のあるクラス群との間に利用する・されるという関係が成立ち、3 に分類される場合は特定のクラスとの間に利用する・されるという関係が成り立つ。

この分類と関係のあるクラスもしくはクラス群を計算機により解析・呈示する事は有効であろう。

インスタンス生成関係

インスタンス (オブジェクト) はクラス定義より予約語 *new* により生成 (インスタンス化) される。この際、インスタンスの生成を行なっているコードが書かれているクラスと、そこでインスタンス化されるクラスの間には生成する・されるという関係が発生する。

```
Class instance = new Class() ;
```

図 2.6: インスタンスの生成例

クラス生成関係と同様に、あるクラスがどのクラスのインスタンスを生成しているかを読みとる事は容易だが、あるクラスのインスタンスを生成しているクラスをすべて得る事は難しい。クラス生成関係においては全クラスの一定箇所を参照するだけで可能であるが、インスタンス生成関係に関してはクラス定義の任意の箇所でインスタンス化がされる可能性があるので、人間が関係を読みとる事は更に大きな労力が必要となり現実的でない。

2.1.2 設計意図を読み取るために重要な関連

継承の役割の違いを区別した継承関係

オブジェクト指向言語における継承は、実装の継承とインタフェースの継承の二種類の役割を持つ。

- 実装の継承
  親クラスで定義された内部状態とオペレーションの実装をサブクラスへ継承する。
  親クラスの実装を利用して新たにクラスを定義する際に用いられる。

- インタフェースの継承
  親クラスでは内部状態やオペレーションのインタフェースのみを定義し、複数のクラスに対して同じ手順での操作を可能とする仕組みとして用いられる。

なお、Java においては “具象クラス” つまり実装の継承と、全く実装を持たない “インタフェース” の継承の他に、いくつかの実装といくつかのインタフェースが混在したクラスとして “抽象クラス” を継承する場合の 3 つに分類される。

実装を持たないクラスを用いてインタフェースのみ継承する事で、呼び出し・参照の際のクラス間の差異を小さくすることができるため、再利用性を高めるためのデザインパターンにおいて多用されている [1]。

またアプリケーション間で共通した振る舞いのみを実装した、抽象クラスを親クラスとすることで、アプリケーションのフレームワークとなる部分の実装を継承・再利用している。

このように、インタフェースのみ継承する場合と、実装を持つ抽象クラスを継承する場合では、区別されるべきであり、これらを区別する事によって、いくらかのクラス間の関係や設計意図を読み取ることができる。

したがって単なる継承関係ではなく、区別したクラス間の継承関係としてこれらを概観できることは重要である。これらの継承関係はソースコードを静的解析することで抽出できる。

継承した内容の違いによる関係の強さ

実装の継承関係にあるクラス間の依存関係の強さは、ある機能を提供するためのクラス群や、似た機能を提供するためのクラス群を認識するために用いる事が出来る。依存関係の強さは以下の点で判断できる。

- いくつのメソッドをオーバーライドしているか
- 親クラスの public な実装をいくつ利用しているか
- 親クラスの public でない実装をいくつ利用しているか

インスタンスを生成したクラスとそれを利用しているクラスの関係

インタフェースを介した場合 あるクラスが生成したインスタンスが、インタフェースを介して利用される場合、その生成したクラスと、インスタンスを利用しているクラスの対応を読みとる事は難しい。

これは、インタフェースのみを定義した抽象クラスを介して、具象クラスのインスタンスを操作するクラスへは、どこかで生成された具象クラスのインスタンスが何らかの方法で引き渡されているが、引き渡す際に動的に引渡し先などが変化する場合があり、静的解析のみで関連づけを行なう事は出来ない。たとえば Java 言語のケースでは、具象クラスのインスタンスが Object クラスにキャストされて、Hashtable 経由で渡すといったことが可能である。

ただし、関連を持つ可能性のあるすべての具象クラスのインスタンスが、どこで生成されているかを発見することは静的解析で十分可能である。抽象クラスを継承しているすべての具象クラスについて、その具象クラスのインスタンスを生成しているクラスを検出する。再利用性を高めるために、生成のためのデザインパターンが用いられていれば、その結果は特定のクラスに集中するはずである。それを読み取ることができるだけで設計意図はある程度把握できる。

したがって、抽象クラスを用いてインスタンスを操作しているクラスと、そのインスタンスの具象クラスの間の関連を概観できることは、設計意図の把握には重要である。

デレゲーション デレゲーションはオブジェクトコンポジションを用いて、実装の継承と同じ能力とさらなる柔軟性を実現する手法である。

この際、あるクラスがデレゲーションを使って、別のクラスの実装を利用していることを発見するのは、設計意図を把握する上で重要である。デレゲーションそのものは、デレゲーションを行っているクラスの定義を解析すれば容易に判断できる。

1. インタフェースのみを定義した抽象クラスを継承している (Java では implement している)
2. その抽象クラスを継承している具象クラスを生成している。
3. 一部のオペレーションの実装が、この具象クラスの実装を呼び出すだけになっている。

デレゲーションの判断、デレゲーションに用いているクラスの判別は容易だが、その逆にあるクラスがどのクラスでデレゲーションに用いられているかを把握することは難しい。したがって、デレゲーションの関係にあるクラスの間の関連を概観できることは重要である。

2.1.3 呈示方法

関連には、クラス間、オペレーション間、クラス–オペレーション間など、対象の粒度がさまざまである。また、すべてをクラスの粒度として見せる場合に限定しても、関連する全てのクラスを画面に表示し、クラス間に関連を表す線を一度に表示する事は、複雑過ぎて適切な情報取得が出来ない。

そこで、セマンティックズーミングを用いる事によってこれらの問題を解消する。

関連を扱う対象の粒度が一定でない事に対して、セマンティックズーミングを利用して対象と関連を統合・分割する事によって、さまざまな粒度の関連をユーザの認識にあわせた情報呈示を行なう。

セマンティックズーミングによって表示の対象となる構成要素の抽象度、粒度は変化する。抽象度が下がれば、それまで一つの要素として扱っていた対象が、複数の要素の集合となる。その際、対象が一つであったために一つとして扱っていた関連を分割されたそれぞれの要素との間のより適切な関連として分割され結びつけられる。

逆に抽象度が上がって複数の要素として扱っていたものを、一つの要素として扱う場合、複数の要素が持っていた複数の関連を統合する。

これによって、さまざまな抽象度において透過的な関連呈示を行なう事が出来る。

また、関連の度合は結合・分割によって変化するため、関連する構成要素をどこまで表示するのかについては、着目している要素との関連の度合によって決定する。

したがって、ズーミングのレベルと着目点によって、表示される関連と表示されない関連・要素が決定する。

たとえばクラス名だけの粗粒度での表示では、設計意図を読み取るために重要な関連だけを表示する。メソッドボディのレベルまで拡大し、細粒度での表示を行なう事によって、メソッド同士の利用関係など、詳細な関連を表示する事などが考えられる。

2.2 依存解析の研究

Japid Japid[11] は J-model という Java の構文を、15 個のクラスと 51 個の関連で表現したもので、実体関連モデルに基づいてソースプログラムを解析し、細粒度でソフトウエアを扱うことができる。

C 言語のための CASE ツールプラットフォームである Sapid の技術を、オブジェクト指向言語である Java 向けに応用したもので、Java による、もしくは Java のための CASE ツール作成の支援している。Japid のシステムは、解析器、ソフトウエア

データベース (以下、SDB) およびアクセスライブラリから構成されている。

ソースプログラムは解析器により、解析情報は J-model として SDB へ保存されれ、SDB は一種のオブジェクト指向データベースと見なすことができる。アクセスライブラリは、SDB 中のオブジェクトにアクセスするための Java 言語で記述されたクラスライブラリである。

第 3章

ソースコードの視覚化について

本章ではまず、ソースコードを視覚化し、呈示するための基礎として情報視覚化一般に関する現状について述べる。その上で、コンピュータプログラムのソースコードを含むテキストデータの視覚化研究について触れ、視覚化を行なう事の有用性を述べる。

3.1 情報視覚化

コンピュータグラフィクスによる視覚表示は、これまで科学技術データの視覚化にしか用いられなかった。しかし、高度な表示能力を持つ、低価格で高性能な計算機の普及により、計算機ユーザインタフェース、情報検索、ソフトウェア開発などさまざまな分野でも利用できるようになった。そして、そのような中 Card らにより抽象データとの対話手段・視覚表現として information visualization (情報視覚化) が定義された [7]。

これにより、映像などの視覚情報を表示するための視覚システムから、視覚情報に限定されず、さまざまな情報を抽象化し、内在する構造・傾向などの情報を視覚化し表示するモデルが構築された [4]。

その場合、入力である何らかの情報を、視覚情報として出力するためには、以下の状態を遷移する。

Raw データ 視覚化し理解・認識しようとする情報の基礎となるもの。ソースコードなどは、この状態としてとらえる。

データテーブル Raw データから視覚化するために必要な情報を変換・抜き出したもの。依存関係の情報など、Raw データを特定の条件で抽象化したもの。

視覚化構造 データテーブルを視覚化するために、空間上にマッピングしたもの。依存関係をグラフの形式で 3 次元空間に配置した情報など。

表示情報 視覚化情報を実際にユーザに呈示する状態に変換し、表示されるもの。3 次元空間に配置された依存関係グラフを 2 次元情報へ変換し、ディスプレイ上に表示できる形にしたものなど。

図 3.1: 情報視覚化の参照モデル

これらの状態の間および、視覚化された情報を見たユーザとシステムの間には 3 種類の相互作用が発生する。

Data Transformations 視覚化する情報そのものに対する変更等の要求。RawData から何を抽出・合成するかについて。

Visual Mappings 視覚化手法に関する要求。どのような形式で視覚化し、情報を提示するかについて。

View Transformations 視覚化された情報に対する変更等の要求。表示箇所、視点、視野の変更など。

3.2 利用例とその表示情報、操作

- 計算機ユーザインタフェース

  計算機の持つリソースをアイコンなどの形で抽象化し、それらを直接操作する事によって計算機とのインタラクションを実現している。現在の計算機ユーザインタフェースにほぼ標準として採用されている。

- 情報検索インタフェース

  従来のデータベース検索・表示システムと異なり、情報を空間中に配置し、それらの間には関連としてリンクを張る。ユーザはデータ空間中をリンクをたどって関連情報へアクセスできる。

- プログラムビジュアライゼーション

  ソフトウェア開発を支援するために、テキストベースのソフトウェアの構造や振舞などのプログラムに関する情報を視覚的に示すことである。これに対し、ビジュアルプログラミングとは視覚情報そのものがプログラムであり、区別される。更にプログラムビジュアライゼーションは B. A. Myers により視覚化情報がコードかデータか、描画が静的か動的かによって以下の 4 つに分類されている [8]。

  – 静的コード視覚化システム

    プログラム構造の描画、フローチャートの表示など。

  – 動的コード視覚化システム

    今日の RAD ツールに見られるデバッガ機能が有している、コードの実行部分を動的に表示するもの。

  – 静的データ視覚化システム

    Myers はポインタ接続された構造体データ構造を自動的に図示した。

  – 動的データ視覚化システム

    データの変化などを動的に示すことでアルゴリズムの視覚化を示すなど。

3.3 視覚化の研究

- Pad++

  ズーミング・グラフィカル・インタフェースおよび、アイコンに基づいたインタフェースデザイン。グラフィカルなデータセットへの操作を滑らかなズーミングで行なわれる。

- Jazz

  Pad++のズーミング・グラフィカル・インタフェースを継承し、情報の関連を 2 次元平面上の位置関係とその間のリンクによって表している。ズーミング機構により

表現される奥行き方向への位置関係はそれぞれのオブジェクトに対する意味的ズーミング (semantic zooming) としてとらえている。

- SeeSoft

  プログラムに関する全体情報の取得を目的としている。2 次元平面上にソースコードが 1 行 1 本の線として表現され変更履歴や開発者情報を色を用いて表示する。ユーザはプログラムの概略と変更履歴を取得できる。

第 4章

現状

ソースコード視覚化技術の現状について、既存のツールの紹介と不満点を述べ、次に述べるソースコード視覚化システムに対する要求を示す。

4.1 既存ツール考察

ソフトウェアの持つ構文情報から、その構造情報や依存関係は、多くの市販・一般配布されているソフトウェア開発支援ツールが解析・視覚化している。

JBuilder

はじめに、Java 開発環境として作成・公開されている JBuilder の、ソースコード視覚化手法について述べる。

JBuilder には構造ペインと内容ペインが存在する。構造ペインには Java ソースコードのクラス構造が表示されており、構造ペインでの選択によって、内容ペインにおける該当要素の定義がハイライトされる。

つまり、ソースコードを抽象化した構造情報を併記し、両者の間に選択とハイライトという、お互いの対応位置を示す、手がかり情報のリンクを用意することによって、ユーザが両情報を同一の物であるとの認識を補助する仕組みである。

また、ソースコードへの変更が即座に構造ペインへ反映されるため、変更箇所と、その結果表示に変化のあった場所との間に、関連を認識しやすい。

ただし、情報の切替や、import やパッケージの指定による、関係クラスの表示が、先ほど見ていた情報からの切替のみでしか行なわれないので、頻繁に複数のクラスやパッケージなどを、相互に参照する場合には、混乱する可能性がある。

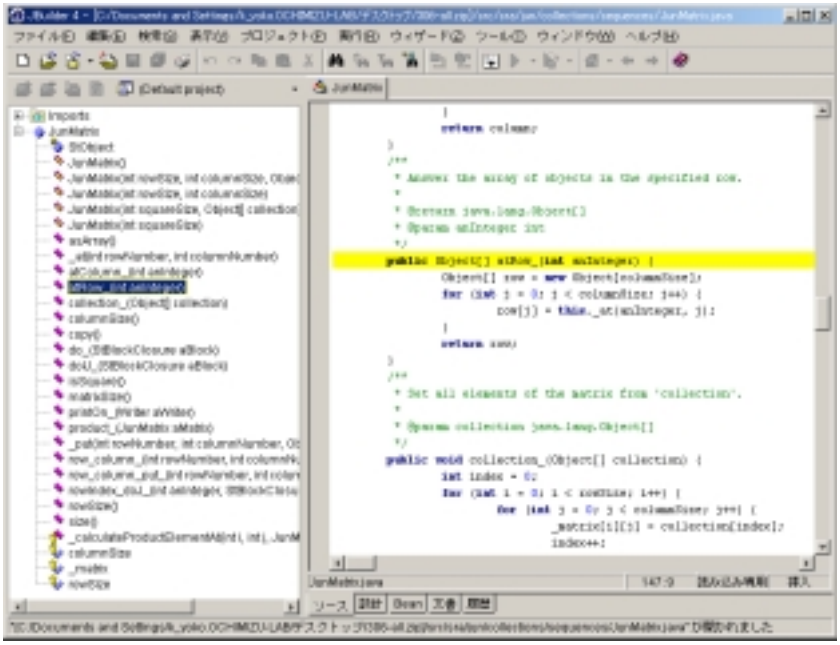

図 4.1: JBuilder 構造ペインと内容ペイン 表示例

Faham

次に、Java クラスライブラリ理解促進ツールとして、一般に公開されている Faham を例にあげる。Faham は Java のクラスライブラリを理解するためのツールとして、Java クラスライブラリの様々な側面を 3 次元空間上に視覚化する。

Faham は表示内容に関して、クラス継承の Cone Tree、クラスサブツリー、クラス概観ビューの 3 種類と、表示形式として 2 次元木構造と、3 次元空間への図形を用いた表示の 2 種類が可能である。

図 4.2 の例では、2 次元のクラス構造表示には、最も詳細に構造情報を呈示した場合から、複数のメソッド、フィールド、コンストラクタなどをそれぞれ種類別に集約し、表示量を減らすことが可能である。この機構により、メソッド、フィールドなどの数が極端に多いクラスの閲覧にかかる負担が軽減され有用である。表示情報の取捨には有効である。

また、JBuilder と同様に、一方の表示中の構成要素をマウスでクリックするなどの、ユーザからのインタラクションで、他方の対応箇所を色を変えるなど、お互いの対応が認知しやすい仕組みが採用されている。

しかし、表示されている情報の種類はクラス情報と、それを図形で示した関連図のみである。情報の見せ方違うのみで、抽象化レベルを変更しつつ、表示を切替えることとは大

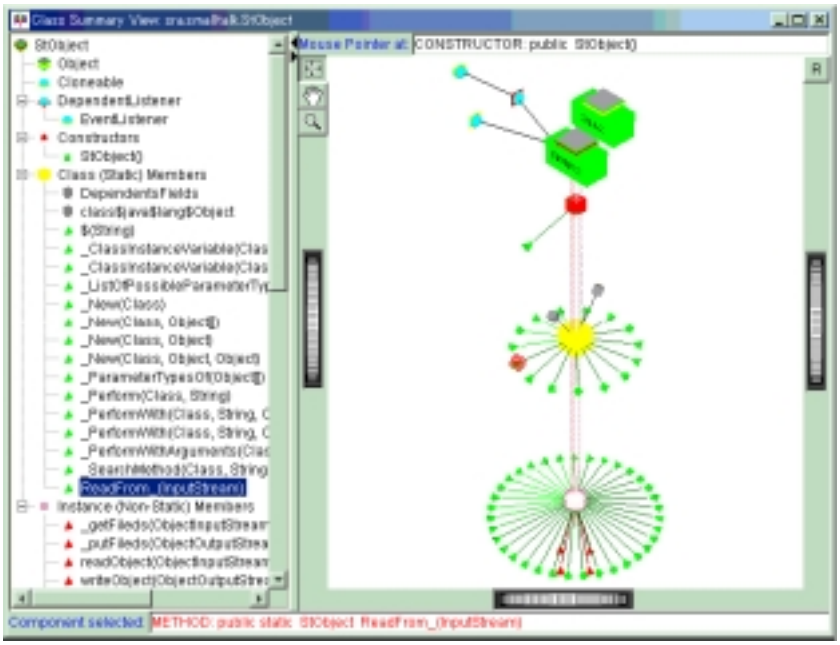

図 4.2: Faham クラス概観ビュー 表示例

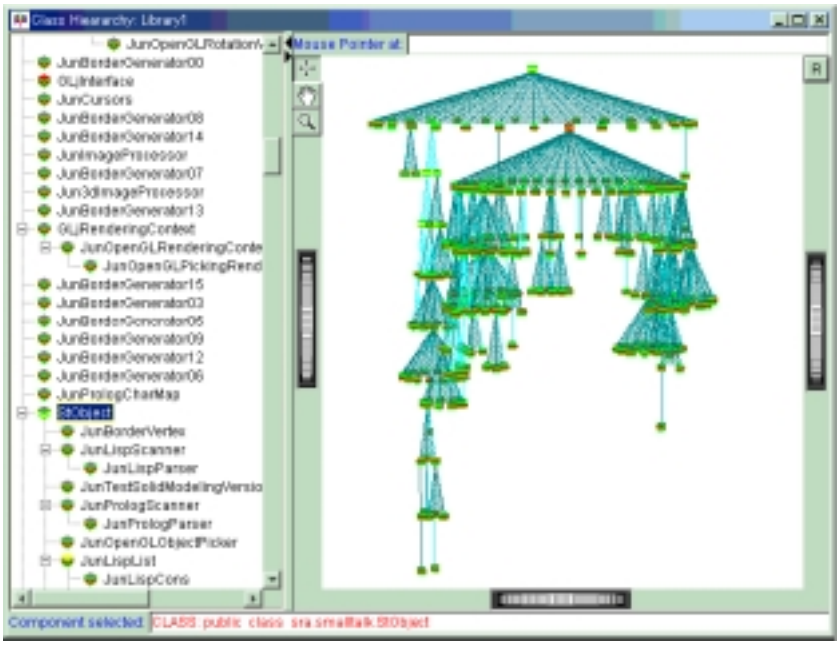

図 4.3: Faham クラス継承 Cone Tree 表示例

きく異なっている。

また、図 4.3 ではクラス継承関係に関する表示がされているが、単純な間は同じ情報しか伝えない 2 つの情報となっており、無駄となることが大きい。

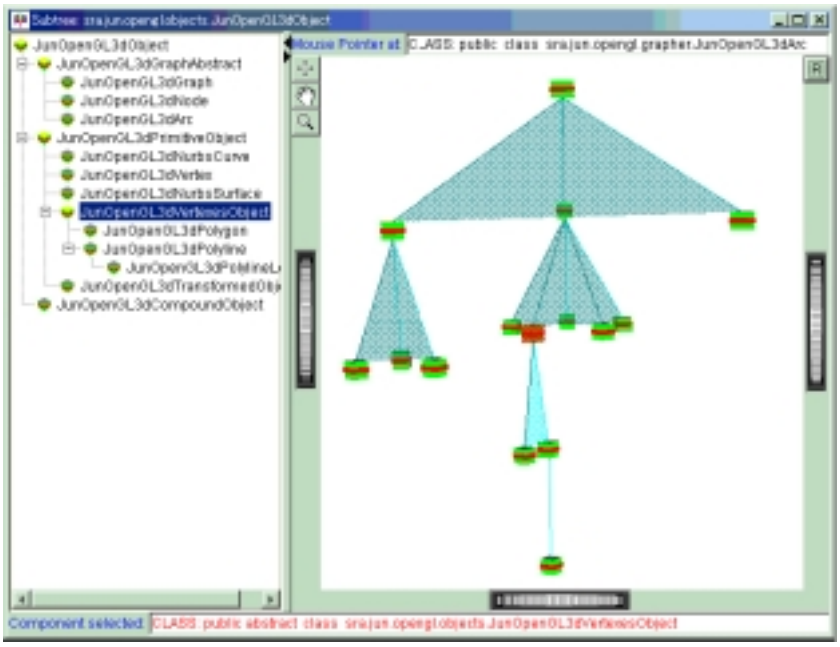

図 4.4: Faham クラスサブツリー 表示例

RATIONAL ROSE

ラショナル ローズはラショナル ソフトウェア社のビジュアルモデリングツールである。

ローズは、オブジェクト指向言語の依存解析は構成度であるが、表示情報量の設定が可能であれば図 4.5 のように関係情報の爆発的な増加を、押えることができ、依存解析に有用であると思われる。

4.2 不満点と解消方法案

多くのツールは関係の検出について適切な情報量での呈示が考慮されておらず、図 4.5 のように関係の Zero hit や Mega hit がに陥ってしまい、意味のある情報の抽出ができなくなってしまう。そのような問題を解決するには、表示情報の取捨選択を行ない、情報量を一定に保つ技術の利用が効果的であると考える。

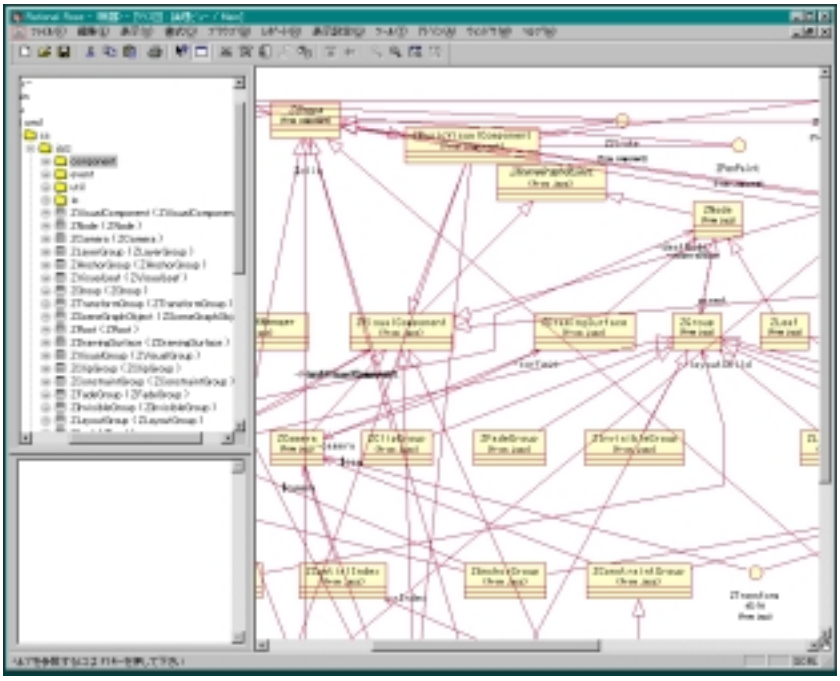

図 4.5: ラショナル ローズ クラス図 表示例

また、情報の抽象化を行ない、構造の簡潔な呈示を行なう場合、抽象度や注目点の変更の際に表示の切替えが起こるが、ユーザが変化についていけず、認識を崩してしまう。

同時に表示して、対応箇所のハイライトなどで対応づける手法が多く用いられているが、Jazz の semantic zooming を用いることで抽象度・注目点を見失いにくい表示 (切替ではなくて) 変更ができないかと考えている。

第 5 章

SZ_Browser の設計と実装

本章では、これまでに述べた依存解析の手法と、情報視覚化のための参照モデルを用いて、Java ソースコードの抽象度を変化させながら呈示するシステムとして SZ_Browser の設計を行ない、プロトタイプシステムとして、その一部を実装することを目指す。

5.1 設計

視覚情報を変更する際に、メンタルモデルを保持するためには、ユーザが許容できる量以上の変化を一度に行なわないことである。ソースコードの構文的構造、意味的構造に対して Zooming User Interface を備えることにより、メンタルモデルを保持しつつ表示の変更を行なう。SZ_Browser は情報視覚化の参照モデル (図 5.1) に従ってデータを視覚化する。

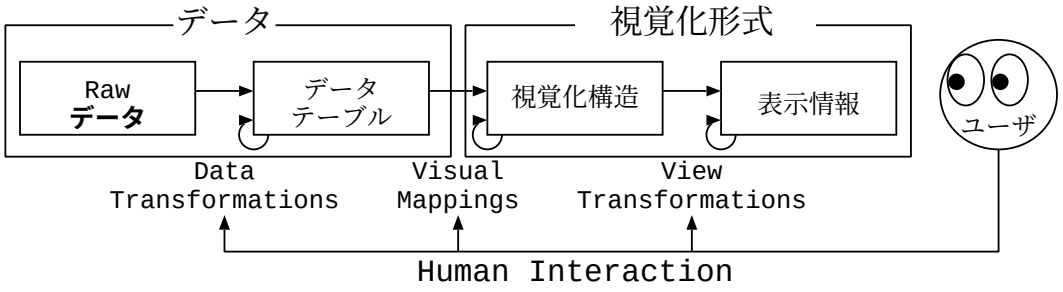

図 5.1: 情報視覚化の参照モデル

5.1.1 状態の定義

参照モデルにおける 4 つの状態を Java ソースコードの視覚化プロセスに対して、どのように適用するのかを示す。

Raw データ Raw データは入力である Java ソースコードそのものである。

データテーブル Raw データであるソースコードから、視覚化に必要な構造情報として、依存関係を抽出したものを保持する。

視覚化構造 視覚表示の対象であるプログラムの、構造情報を 3 次元上にマッピングし、それらの間に依存関係をリンクとして配置したもの。深さ情報はソースコードの粗密度合を示す。

表示情報 3 次元上にマッピングされた情報を、計算機のディスプレイに表示できるように 2 次元平面上にマッピングしなおしたもの。3 番目の情報である深さ情報は、ディスプレイの表面をユーザーの着目点の深度とし、それぞれの構造情報の深さ情報が着目点の深度に対して、ある範囲内であれば表示する。

5.1.2 相互作用の定義

次に、参照モデルにおける 3 つの相互作用を、Java ソースコードの視覚化プロセスに対して、どのように適用するのかを示す。

Data Transformations ソースコードの構文解析および、依存解析を行ない、視覚化の粒度にあわせて適切な構文情報、依存関係を抽出する。今回は 7 段階の粒度レベルを設定し、それぞれに対して構文情報を設定した。

この相互作用が実行されるタイミングは、Raw データが投入された時と、ユーザからのインタラクションによって Raw データから構文情報、依存関係を抽出する手順が変更された場合である。例えば、構文情報よりクラス定義のみ抽出し、依存関係としてクラス継承関係のみ抽出するように変更された場合などが当てはまる。

Visual Mappings データテーブルに格納されている、抽出済みの構文情報および、依存関係に対して、3 次元空間へのマッピングを行なう。この際、粒度情報を深度として扱う。この相互作用が実行されるタイミングは、データテーブルが投入された時と、ユーザからのインタラクションによって構文情報、依存関係を 3 次元空間に再マッピングする場合である。

View Transformations 視覚化構造として3次元空間中にマッピングされている、構文情報および依存関係といった表示オブジェクトをディスプレイに表示する。この際ユーザは3次元空間中の、どのポイントのどの深度へも着目点を移動することができる。

ただし、横方向や斜め方向への視野変更はできない。着目点の深度がマッピングされている構文情報および、依存関係それぞれに設定されている深度との差が、一定範囲以内ならば、その情報もしくは関係は画面上へ表示され、着目点の移動によりその範囲から外れてしまった場合、非表示となる。

この際、範囲との差により表示オブジェクトの表示濃度・大きさを調節することで、スムーズな表示変更の可能なズーミング・ユーザ・インタフェースを実現する。

5.1.3 抽象化レベル

次に、ソースコードの抽象化レベルについて示す。今回最も粗粒度なものから順に、抽象化レベル 1, 2, …, 7 と 7 段階設定した。

レベル 1 今回の実装では、Java プログラムソースコードの最も大きな構造として、クラスを単位とした。よって、抽象化レベルの最も高いレベル 1 は、それがどのクラスかを示す情報である、クラスの名前のみを表示する。

```
Sample
```

図 5.2: 例:抽象化レベル 1

レベル 2 クラス名に加えクラスの型、アクセス制限、継承関係を表示する。

```
public class Sample extends jp.ac.jaist.XX.Extend
 implements Implement, jp.ishikawa.sample.Implement
```

図 5.3: 例:抽象化レベル 2

レベル 3 レベル 2 の情報に加え、クラス定義のうち、外部より参照される可能性のある public 属性を持つフィールド、メソッド、コンストラクタ定義をその名前のみ表示する。

```
public class Sample extends jp.ac.jaist.XX.Extend
  implements Implement, jp.ishikawa.sample.Implement {

  main();
  true_sample2();
}
```

図 5.4: 例:抽象化レベル 3

レベル 4 レベル 3 の情報に加え、クラス定義のうち、外部より参照される可能性のある public 属性を持つフィールド、メソッド、コンストラクタ定義をその名前と型、アクセス制限、例外の指定といった情報を表示する。

```
public class Sample extends jp.ac.jaist.XX.Extend
  implements Implement, jp.ishikawa.sample.Implement {

  public static void main();
  public void true_sample2();
}
```

図 5.5: 例:抽象化レベル 4

レベル 5 レベル 4 の情報に加え、メソッド、コンストラクタの実装以外の情報を表示する。この段階で、このクラスの外部に対してのインタフェースが、すべて示されたことになる。

```
public class Sample extends jp.ac.jaist.XX.Extend
  implements Implement, jp.ishikawa.sample.Implement {

  public static void main(String args[]);
  public void true_sample2();
}
```

図 5.6: 例:抽象化レベル 5

レベル 6 レベル 5 までは classbody の情報のうち、public な属性を持っていないのもは、表示されていなかったが、ここでは全要素について、そのインタフェースとなる情報を表示する。

```
public class Sample extends jp.ac.jaist.XX.Extend
  implements Implement, jp.ishikawa.sample.Implement {

  public static void main(String args[]);
  private static void true_sample1();
  public void true_sample2();
}
```

図 5.7: 例:抽象化レベル 6

レベル 7 入力ソースコードそのものである。最細粒度、詳細な情報として表示する。

```
package jp.ac.jaist.ochimizu.k_yoko;

import jp.ac.jaist.*;
import jp.ac.jaist.ochimizu.*;

public class Sample extends jp.ac.jaist.XX.Extend
  implements Implement, jp.ishikawa.sample.Implement {

  public static void main(String args[]){
    true_sample1();
    true_sample2();
  }
  private static void true_sample1(){
    System.out.println("sample:true_sample");
  }
  public void true_sample2(){
    System.out.println("sample:true_sample");
  }
}
```

図 5.8: 例:抽象化レベル 7

5.2 実装

5.2.1 実装箇所

本システムでは図 5.9 のように、情報視覚化参照モデルにて定義されている、4 つの状態とそれを仲介する 3 つの相互作用をそれぞれ実現することで、本システム全体としての機能を実装している。

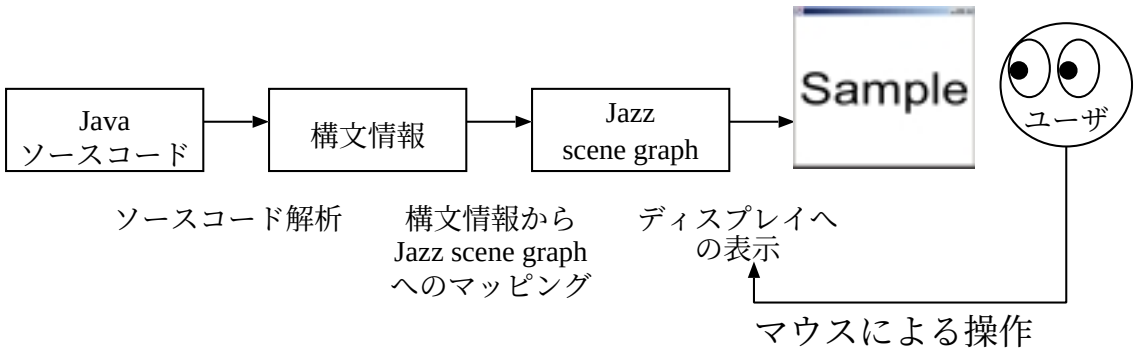

図 5.9: 本プロトタイプの構成

そこで、上の設計をもとに、その一部を実装したプロトタイプシステムを作成した。一部とは、ソースコードの視覚化レベル別の構造情報の解析とその表示である。

依存解析については粒度変化にも対応した依存関係の解析および、表示手法についての設計が不十分であり、実装を断念した。

その結果、参照モデルにて示されている Human Interaction のうち、実現されている箇所は、View Transformations に対する表示情報の変化のみとなっている。

5.2.2 実装に用いた既存の技術

JavaCC

Data Transformations を実装するためには、Java ソースコードから抽象化レベル毎の構文情報を取得するために、構文解析器が必要となる。

そこで、構文解析器の生成には Java Compiler Compiler[9] を用いた。Java Compiler Compiler(JavaCC) とは Sun Microsystems, inc.(Sun) と Metamata, Inc.(Metamata) により開発され、現在 Metamata がフリーで公開している。JavaCC は 100% Pure Java 認定されている Java プログラムであり、多くの Java 実行環境にて動作する構文解析器ジェネレータである。

本プロトタイプでは、視覚化レベル毎に用意したサブ解析プログラムが JavaCC にて生成された構文解析器である。

Jazz

View Transformations を実装する際に、ズーミング・ユーザ・インタフェースの実現に Zoomable User Interface(ZUI) Toolkit である Jazz を用いている。

Jazz は B. B. Benderson らによって開発・公開されている Toolkit であり、Java で実装されている。Jazz は Java2 環境であればおおむね機能する。

本プロトタイプでは画像表示に Jazz API を用いており、オブジェクトの表示、移動、拡大・縮小、フェードイン・フェードアウトを実現している。さらに、視覚化構造として Visual Mapping が出力するデータは Jazz scene graph と呼ばれるツリー構造によって表現している。

5.2.3 実装の詳細

本プロトタイプは先に示した通り、参照モデルにおける相互作用である、3 つの段階に分けて処理を行なっている。

以下にそれぞれの実装について詳細を述べる。

Data Transformations

本プロトタイプでは各抽象化レベル同士の独立性と実装の容易さを考え、それぞれのレベルに対して、構文解析器をそれぞれ用意し、解析システムのサブプログラムとして位置付けている。これによって、各抽象度の定義に独立性の高い指定が可能となり、再帰的に抽象度を荒くしていく手法に比べ、ある抽象度の解析結果のみを条件を変えて再取得することなどができる。

解析部分の実装には、JavaCC に含まれている Java Grammar をもとに、処理内容を実装したものを用いている。以下では、JavaCC における文法定義の基本的な事項について説明した後、各解析サブシステムの一部を実装部を示しつつ、抽象化レベルに沿ったソース構造情報の取得のついて示す。

JavaCC の文法定義の基本は、現実の Java コードを記載し、解析ルーチンの実行や、その結果を受けとる箇所である Java コンパイル部、字句定義を行なう規則部、解析器の生成規則を定義する規則部に分かれる。本プロトタイプの解析サブシステムでは、規則部に Java 言語仕様示される Java 予約語やさまざまな形式のリテラル、演算子などが表されている。規則部には Java 1.2 準拠の構文規則に合致する解析器のスケルトンが記載されて

おり、各サブシステムは構文解析時に目的の構成要素を検出できた際に行なう手続きを実装してある。

例えば、規則部に

```
<CLASS: "class">
<PUBLIC: "public">
<PRIVATE: "private">
<IDENTIFIER: <LETTER>(<LETTER>|<DIGIT>)*>
```

⋮

という記載があり、規則部に

```
void class() :{}
{
  [ <PUBLIC> | <STATIC> ] <CLASS> <IDENTIFIRE>
  {/*Java Statement*/}
}
```

と記載がある場合、この文法定義から作成された構文解析器は入力文字列が “public class Sample” や “private class Example”、単に “class Program” などにマッチし、Java Statement の箇所のコードが実行される。

なお、[] で囲まれた箇所は存在が任意の構文であり、()*で囲まれた場所は存在が 0 回以上であることを示している。<>で囲まれた場所はあらかじめ定義された token で、Java 言語仕様に従ったものが、Java 予約語やさまざまな形式のリテラル、演算子などが構文規則の形で表されている。< IMPLEMENTS: "implements" >や< #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >などである。() のついた箇所はこのような規則定義が他にもあり、関数的に呼び出している。

JavaCC において、token は Token 型のオブジェクトとして定義され、マッチした文字列やその出現場所に関する情報を持っている。その結果、

```
Token t;
t = <IMPLEMENTS>
```

等とすると、文字列”implements”とその文字列の開始行、開始列、終了行、終了列などの情報を取得できる。

次に、各視覚化レベル毎の解析サブプログラムの実装を示す。

level1 においては出力すべき情報が class 名だけであるので、クラス定義を行なっている箇所にて実際の処理は単純である。Java ソースコードを

```
<CLASS> t=<IDENTIFIER>
[ "extends" Name() ]
[ "implements" NameList() ]
ClassBody()
```

としている。これによって、クラス定義を行なっている

```
<CLASS><IDENTIFIER>["extends" Name()]["implements" NameList()] ClassBody()
```

という構文を認識し、<IDENTIFIRE> の箇所にあたる文字列の情報を得ている。

Visual Mappings

本プロトタイプシステムにおいて、データテーブルから視覚化構造へのマッピングとは、Data Transformations を行なうソースコード解析部にて、7 段階の抽象化レベルで抽象化をおこない、構造解析されたデータを取り込み、Jazz scene graph に変換することである。

しかし、実際にはソースコード解析部は 7 つのサブプログラムに分かれており、それらの起動、結果の収集などの作業も行なっている。

これは、本プロトタイプではソースコードが、ダイナミックに変更されることがないため、ソースコード解析は起動時の一度に限られており、サブプログラムを制御するシステムが不要であったため、システムの簡素化のためおこなった。

マッピング部はまず解析部サブプログラムからそれぞれの抽象化レベルで構文解析したデータを取り込み、それを Jazz scene graph にマッピングする。

ここで、本プロトタイプが前提としている Jazz scene graph の構成を示す (図 5.10)。

図 5.10: プロトタイプシステムにおける Jazz scene graph の構成

本プロトタイプでは 7 段階の構文構造を表示するため、scene graph には 7 つのノードグループが存在する。また、それらの間には細粒度の構造へのリンクが設定されていることが示されている。ここで、書くノードの示す情報についてまとめる。

ZRoot scene graph を構成する要素の中核となるノード。必ず必要だが、視覚化表現に関わるパラメータは持っていない。

ZLayerGroup キャンパスにあたる。どのノードグループを保持しているのかを保持する。

ZAnchorGroup 他のノードに対するリンクを張る場合に用いる。リンク対象の ZVisualLeaf のノード番号を ZAnchorGroup のパラメータ`destNode` に指定する。

ZFadeGroup フェードイン・アウトを行なう場合に用いる。Z 軸方向の位置としてノードがフェードイン・アウトするポイントを保持している。

ZVisualLeaf 各ノードグループの表示される要素を管理するノード。

ZText 表示のための文字情報を保持するノード。実際に表示されるソースプログラムの構成要素を保持している。

ZCamera ユーザの視点を示す。

Visual Mapping の実装には通常の Java にて JavaCC からの出力を、Jazz オブジェクトへマッピングした後、Jazz オブジェクト階層図にしたがい、木構造 (一部表示オブジェクトにリンクを設けるために、循環している偽木構造) で出力している。

Visual Mapping では、視覚化レベル別に構文情報を受けとり、3 次元空間中にマッピングしたものを視覚化構造として出力している。

View Transformations

Jazz scene graph に基づいて構成されたデータを表示情報として画面に表示する。

5.3 実行例

以下は本プロトタイプに対して、先に示した図 5.2 のソースコードを入力として、実行した場合の出力画像を示す。

本プロトタイプでは、ソースコードの抽象化レベルとして、最も祖粒度なものから順に抽象化レベル 1, 2, ⋯, 7 としている。

今回の実装ではレベル 1 の抽象度の状態はクラス名のみを設定している。したがって、本プロトタイプに図 5.2 のようなソースコードを入力した場合、最も祖粒度な状態での画像出力は図 5.11 のようになる。

図 5.11: level1 の表示

この状態で、“Sample” の文字をマウスでクリックしてみると、level1 の情報には level2 へのリンクが張ってあるので、level1 の情報が消え、level2 の情報が浮かび上がってくる。(図 5.12, 5.13)

図 5.12: level1 から 2 へ変化する様子

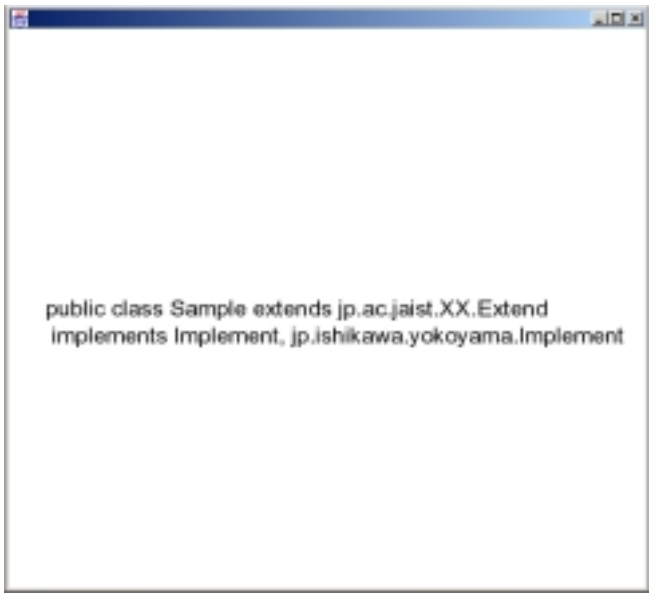

図 5.13: level2 の表示

level2 では抽象化の条件をクラスの型宣言と継承しているクラスの情報のみを表示することとしている。

以後同様に、level3, …, 6, 7 と同様の手順で抽象度が低くなり、より詳細な情報を呈示する。(図 5.14, 5.15)

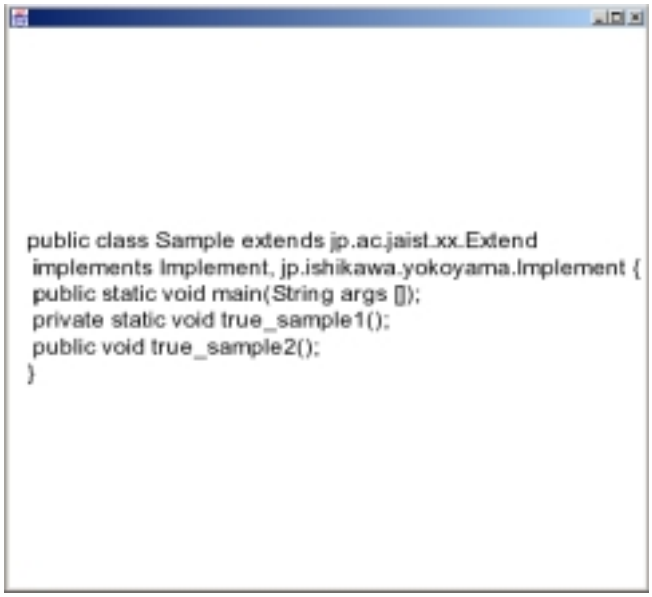

図 5.14: level6 の表示

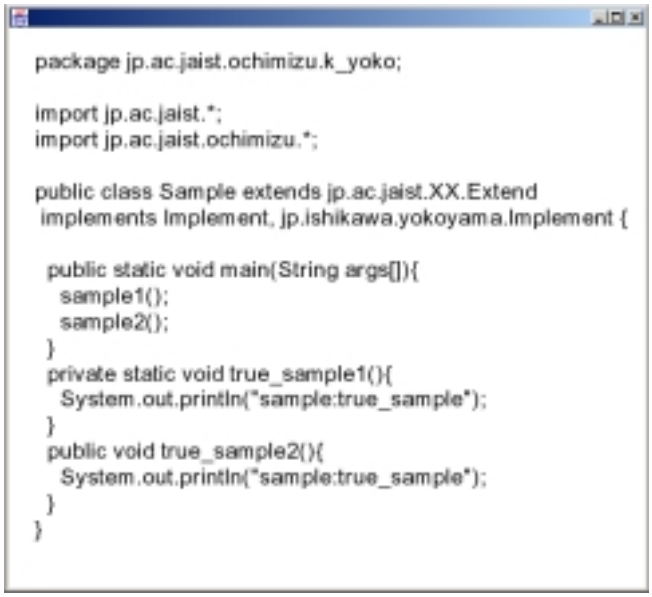

図 5.15: level7 の表示

第 6 章

おわりに

6.1 まとめ

本研究では、オブジェクト指向言語である Java を対象とした、ソースコード理解のために解析・取得が有用であると思われる依存関係の提案をし、その取得方法について検討を行なった。また、その過程でそれらの取得を人間が、すべて手作業で行なうことの非効率性について示し、計算機による依存解析の重要性について述べた。

また、解析によって得られた情報を、情報視覚化の原理に基づき呈示するための、枠組について示し、ソースコードの解析結果の視覚化に対して有効であることを確認するために、参照モデルが持つ、ソースプログラムの解析・表示にとっての 4 つの状態と 3 つの相互作用を定義し、ソース構造・依存関係の視覚化システムについての設計と、その一部をプロトタイプシステムとして実装を行なった。

6.2 今後の課題

依存解析に関して、不完全なソースコード依存関係についての整理と、解析手順の作成を行なう必要がある。

ソースコード視覚化システムの振舞を依存解析、構造解析、ユーザからのインタラクションの 3 つに分類し、その上で、情報視覚化参照モデルとの関連を整理する必要がある。

謝辞

本研究を行なうに当たり、終始御指導を賜わった落水 浩一郎教授に深謝致します。

また、日頃から有益な御助言をいただき、多面に渡って励ましていただいた落水・篠田研究室 助手 村越 広享 博士・服部 哲 博士・藤枝 和宏 博士に深く感謝致します。

最後に、本論文をまとめるに当たって御協力いただいた落水・篠田研究室の諸兄に深く感謝致します。特に猪俣 敦夫、藤田 充典両氏にはお世話になりました。厚く御礼申し上げます。

参考文献

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlisside : *Design Patterns*, Addison–Wesley, 1994.
(邦訳 : 本位田 真一, 吉田 和樹 : デザインパターン, SOFTBANK, 1995)

[2] 小池 英樹, 石井 威望 : 3 次元ソフトウェア視覚化の枠組と実例による有効性の評価, 情報処理学会論文誌, Vol.33, No.6, pp.778–798, 1992

[3] Bederson, B. B. : Pad++ : Advances in Multiscale Interfaces., *Proceedings of CHI'94,ACM Conference on Human Factors in Computing System*, New York, pp.315–316

[4] Stuart K. Card and Jock D. Mackinlay and Ben Shneiderman : Information Visualization, *Readings in Information Visualization Using Vision to Think*, Morgan Kaufmann Publishers, Inc., 1999, pp.1–34

[5] H. Koike, Fractal views : A fractal-based method for controlling information display, *Proceedings of 1993 IEEE / CS Symposium on Visual Languages (VL '93)*, IEEE CS Press, 1993, pp.55–60

[6] 小池 英樹 : ビジュアリゼーション, ビジュアルインタフェース – ポスト *GUI* を目指して, bit 別冊, pp. 24–44, 1996.

[7] G. G. Robertson, S. K. Card and J. D. Mackinlay : Information visualization using 3dinteractive animation. *Communication of the ACM*, Vol. 36, No. 4, pp.57–71, 1993.

[8] B. A. Myers : Visual programing, programing by example and program visualization : A taxonomy, *Proceeding of the ACM Conference on Human Factors in Computing Systems (CHI'86)*, pp.59–66, ACM Press, 1986.

[9] Metamata, Sun Microsystems, inc. : Java Compiler Compiler Documentation, http://www.metamata.com/javacc/, 1997

[10] Benjamin B. Benderson, Jon Meyer, Lance Good : Jazz : An Extensible Zoomable User Interface Graphics Toolkit in Java, *In Proceedings of User Interface and Software Technology (UIST 2000)*, ACM Press, 2000.

[11] Yoshinari Hachisu : CASE Tool Platform for Object-Oriented Program - Japid: Its Design and Implementation, School of Engineering, Nagoya University, Ph.D Dissertation, 1999.