UMLのクラス図とJavaプログラムとの関係抽出について

藪谷 悠介[1]　　片山 徹郎[2]

Extraction of Relations between UML Class Diagram and Programs in Java

Yusuke YABUYA, Tetsuro KATAYAMA

Abstract

In software development, UML(Unified Modeling Language) is used more as the development by using object-oriented technology increases. And, the case in which system troubles have occurred owing to insufficient testing has been increasing. This research aims at increasing reliability of programs written in Java object-oriented language by testing them with UML class diagram. As a preparatory research, this paper extracts relations between elements of UML class diagram and source codes in Java. By testing source codes with the extracted relations, it becomes possible to check whether the structure of the program is implemented as it is designed.

Key Words:
software testing, UML(Unified Modeling Language), class diagram, Java, object-oriented

1. はじめに

近年, ソフトウェア開発は, マーケットの競争が激化し, ソフトウェアの規模が大きく複雑になっていく中で, 非常に短期間にリリースしなければならない。また, 頻繁な要求や仕様の変更に柔軟に対応しなければならない状況から, オブジェクト指向技術を前提にした, コンポーネントに基づくインクリメンタルな開発をせざるを得なくなってきている[1]。このようなオブジェクト指向による開発の必要性に伴い, モデリング言語 UML(Unified Modeling Language)[2] が導入されるようになってきており, UML を基にした, オブジェクト指向設計ツールも開発されている[3)4]。

また, システム稼動前のテストを十分に実施できず, そのために障害発生を防げなかった事件が増えてきている。障害の原因となるバグや仕様の不備は, どんなシステムにも必ず潜んでいる。こうしたトラブルの芽を稼動前に摘み取るための最後の砦がテストである。システムのトラブルが頻発するにつれて, プログラムの信頼性向上のための一手法である, テストの重要性が増してきている[5]。

そこで本研究では, UML のクラス図を用いて, オブジェクト指向プログラミング言語 Java[6] で記述されたプログラムのテストを行ない, 信頼性を向上させることを目的としている。本稿では, その予備研究として, UML のクラス図の要素と Java 言語で書かれたソースコードとの対応関係を抽出する。対応関係を抽出することにより, この対応関係に基づいてソースコードをテストすることが可能になる。クラス図をテストに用いる理由は, クラス図はプログラムの構造を表しているので, プログラム (ソースコード) が設計どおりに作られているかどうかを確認することが可能になると考えられるからである。

2. UML のクラス図

2.1 UML とは

UML(Unified Modeling Language)[2] とは, 統一モデリング言語の意味で, ビジネスや各種システムを対象として, その構造とダイナミクス (動的な振る舞いや

1) 情報工学専攻大学院生
2) 情報システム工学科助教授

挙動) を分かりやすく表現するためのビジュアルな言語である。UML を導入することにより, ユーザと開発者, または, 開発者同士のコミュニケーションギャップを解消することができる。また, ユーザからの要求を正確に把握できるようになるため, 仕様の認識違いによる手戻りを削減できる。更に, UML によるオブジェクト指向設計が効果的にモジュール化を促進し, 保守コストを削減することも可能となる[1]。

UML は, ユースケース図, クラス図, オブジェクト図, ステートチャート図, シーケンス図, アクティビティ図, コラボレーション図, コンポーネント図, 配置図の9つのダイアグラムから成り, これらのダイアグラムを状況に応じて使用する。

2.2 クラス図とは

クラス図は, UML によるオブジェクト指向分析設計における中心的なダイアグラムである。クラス図を使うことにより, 対象システムを構成するクラス (概念や物事・事象) とそれらの間に存在する関連 (意味的・物理的なつながり) を表現でき, 問題領域 (ドメイン) やシステムの構造を伝えることができる。また, 各オブジェクトがどのような属性や操作を持っているかも, 合わせて記述できるので, 従来の ER(エンティティ-リレーションシップ) 図の発展したものと考えることもできる。

クラス図には, クラス, クラス間の関係, 多重度, および, インタフェースを記述する。

2.3 クラスの要素

クラスは, 以下の4つの要素から構成される。

- クラス

  クラスは, 基本となるモデル要素で, 類似した構造, 振る舞い, および, 関係を持ったオブジェクトの集合を抽象化したものである。クラスは一般に3つの区画から成り, 1段目の区画はクラスの名前とクラスのプロパティ, 2段目の区画はクラスの属性, 3段目の区画はクラスの操作を記述する。

- 属性

  属性は, オブジェクトが持っているデータの定義である。

- 操作

  操作は, オブジェクトの振る舞いを表す。また, オブジェクトが外部に対して行うサービスと見ることもできる。

- 可視性

  属性と操作の定義において, 可視性を定める必要がある。可視性は, 属性の場合, 他のクラスのオブジェクトから, 直接値を得たり設定したりできるかどうかを指定し, 操作の場合, 他のクラスのオブジェクトから, 直接実行できるかどうかを指定する。

2.4 クラス間の関係

クラス間の関係には, 以下がある。

- 関連

  クラス図において, 関連は, あるクラスと別のクラスが静的な構造関係にある場合に使われる。関連は, クラスの間に実線を引いて示す。クラス間の関連線の中央に, 関連名を記述することができ, 関連の端にはロール名を付けることができる。関連名は, 関連の意味を明らかにするために使用し, ロール名は, 一方のクラスから見たもう一方のクラスの役割や立場を示すために用いられる。また, 関連の終端に矢印を付けることにより「誘導可能性」という意味を関連に持たすことができ, 関連している一方のクラスに対してメッセージを送ったり情報を参照したりできる (その必要がある) ことを表す。

- 集約

  クラス図において, 集約は関連の特殊な形で, 全体と部分の関係を表す。関連の線分の終端に白抜きの菱形を加えて表し, 白抜きの菱形が付いてる方が全体を意味する。

- コンポジション

  クラス図において, コンポジションは集約の一種で, 集約関係がより強い全体部分関係にある場合に使用する。全体がなくなると部分もなくなるような関係である。関連の線分の終端に黒塗りの菱形を加えて表し, 黒塗りの菱形が付いてる方が全体を意味する。

- 汎化

  クラス図において, 汎化は, 一般的なもの (スーパークラス) と, 特殊化されたもの (サブクラス) の関係を表す。関連の線分の終端に白抜きの矢印を加えて表し, 白抜きの矢印が付いてる方が一般的なもの (スーパークラス) である。

- 洗練

クラス図において, 洗練は, 一方が他方を詳細化したり, 情報を追加したりして, 記述レベルが変化している関係を表す。また, 洗練を実現と呼ぶこともある。点線の終端に白抜きの矢印を加えて表し, 洗練したとされる方が矢印の先になる。

- 依存

  クラス図において, 依存は, 2つ以上のモデル要素間の意味的な関係を示す。これは, 対象要素 (依存される側) への変更が, 依存元 (依存する側) の要素への変更を必要とする場合もあることを意味する。点線の終端に矢印を加えて表し, 依存される方が矢印の先になる。

2.5 多重度

クラス図において, 多重度は, クラス間の関係が存在する箇所で導入することができ, あるクラスの1つのインスタンスが, もう一方のクラスのいくつのインスタンスと関係するかを表す数を定義する。

2.6 インターフェース

クラス図でのインターフェースは, メソッドのみを定義したクラス (的なもの) で, オブジェクトの振舞いと実装を分離して定義するために使用される。インターフェースは, クラスに≪ interface ≫を付けて表す。属性は存在しないので, 属性の区画は省略できる。また, インターフェース名を持つ小さな円として簡略した表示をすることもできる。

3. Javaのソースコードとクラス図との関係

本稿では, クラス図の要素とJavaのソースコードとの関係を調べ, それらの対応関係を抽出する。この章では, 今回抽出したJavaのソースコードとクラス図の要素との対応関係について述べる。

3.1 属性および操作と可視性

属性は, 変数に対応している。このため, クラス図で定義された変数は, ソースコードで宣言されなければならない。操作は, メソッドに対応してる。このため, クラス図で定義されたメソッドは, ソースコードで宣言されなければならない。逆に, クラス図で定義されていない変数やメソッドが, ソースコードで宣言されてはならない。ただし例外として, クラス名のメソッド (初期化メソッド) だけは, クラス図に存在しなくてもソースコードで宣言できる。可視性は, Javaでは, +はpublic, #はprotected, -はprivateに対応づけることができる。

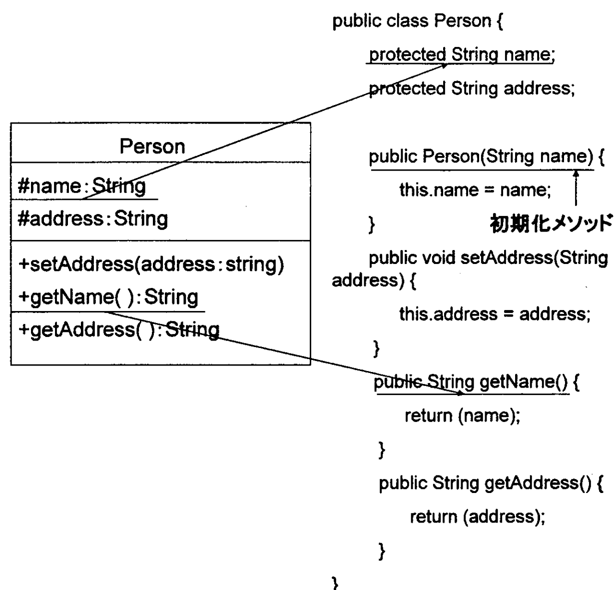


図 1: Personのクラス図とJavaソースコード

図1は, Personのクラス図と, このクラス図を言語Javaで記述したソースコードである。この図を例にし, 属性とソースコード, および, 操作とソースコードの関係を説明する。

Personのクラス図に記述している属性の一つとして, `#name:String`が定義されている。これは, 可視性はprotected, 型はString, 属性名はnameということがクラス図から分かる。よって, Javaソースコード上では, クラスPerson内部で

```
protected String name;
```

と宣言する必要がある。

また, 操作についても同様で, Personのクラス図に記述している操作`+getName():String`は, 可視性はpublic, 型はString, 操作名はgetNameということが分かるので, Javaソースコード上では, クラスPerson内部で

```
public String getName()
```

と宣言する必要がある。なお, ソースコードに, クラス図には存在しないメソッド`public Person(String name)`が存在するが, これは初期化メソッドなので誤りではない。

図2では, 図1で用いた同じクラス図を用いて, ソースコードの誤りの例を示す。この例では, クラス図には存在するがソースコードに存在しない属性`#address:String`がある。このことから, ソースコードに誤りがあることが分かる。さらに, ソースコードに存在するがクラス図に存在しない変数`private int age`と, メソッド`public String getAdress()`があ

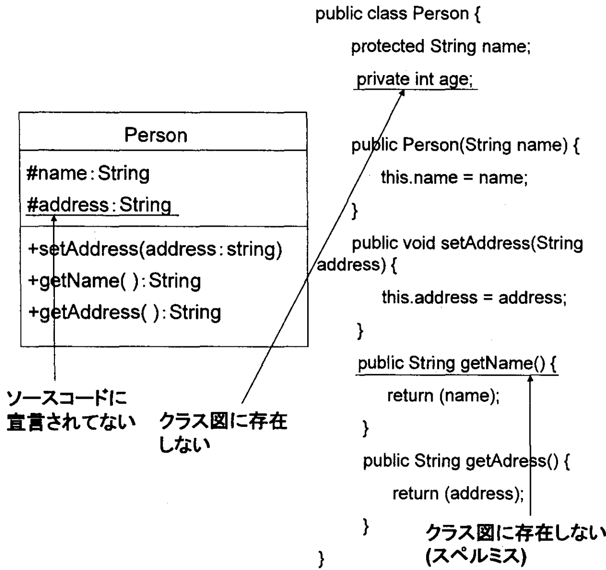


図 2: Person のクラス図と誤りが存在する Java ソースコードの例

る。このことからも，ソースコードに誤りがあることが分かる。なお，後者のメソッドの例は，スペルミス (`address` と書かなければならないのを`adress` と記述) の例だが，この場合も，ソースコード中の誤りとして発見できる。

以上が，クラスの要素である属性および操作と可視性と，Java ソースコードとの対応関係である。この対応関係をまとめたものを，表 1 に示す。

3.2　クラス間の関係

3.2.1　関連

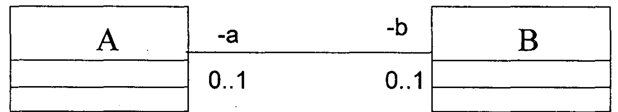
クラス間に関連がある場合，Java では，関連があるクラスのインスタンスを，インスタンス変数に格納して所持する。クラス図に関連が存在しない場合に，他のクラスのインスタンスを，インスタンス変数に格納して所持してはならない。また，関連に向きを持たせると「誘導可能性」という意味になり，矢印が付いている側のクラスのインスタンスを，インスタンス変数に格納して所持する。

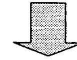
図 3 は，2 つのクラス間に，双方向に関連がある場合の図である。クラス A では，クラス B のインスタンス変数 b を定義するので，Java ソースコード上では，クラス A の内部で

```
private B b;
```

と宣言する必要がある。クラス B では，クラス A のインスタンス変数 a を定義するので，Java ソースコード上では，クラス B の内部で

```
private A a;
```

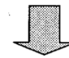




```
Class A{
    Private B b;
}

Class B{
    Private A a;
}
```

図 3: 双方向に関連が存在する場合の例



```
Class A{
    Private B b;
}

Class B{
    //インスタンス変数の定義なし
}
```

図 4: 誘導可能性が存在する場合の例

と宣言する必要がある。

図 4 は，2 つのクラス間に，誘導可能性が存在する場合の図である。クラス A では，図 3 と同様にクラス B のインスタンス変数 b を定義するので，Java ソースコード上では，クラス A の内部で

```
private B b;
```

と宣言する必要がある。クラス B では，インスタンス変数を定義する必要がない。

3.2.2　集約

集約関係がある場合，全体に当たるクラスの変数を，部分に当たるクラスのオブジェクト変数として実装することができる。

図 5 は，Account クラスおよび Person クラスと，Company クラスとが集約関係であることを表している図である。Java ソースコード上では，Company クラスの内部で

```
private Person owner_;
private Person account_;
```

表 1: クラスの要素と Java ソースコードとの対応

クラスの要素	属性	変数として宣言される。クラス図において「可視性 属性名:型=初期値」と書かれ, ソースコードでは, 「可視性 型 属性名=初期値」となる。
	操作	メソッドとして宣言される。クラス図において「可視性 操作名:戻り値の型」と書かれ, ソースコードでは, 「可視性 型 操作名」となる。
	可視性	属性, 操作, ロール名の前に可視性の指示子を使用し, 他のオブジェクトからどのように見えるかを表す。+は public, -は private, #は protected に対応する。

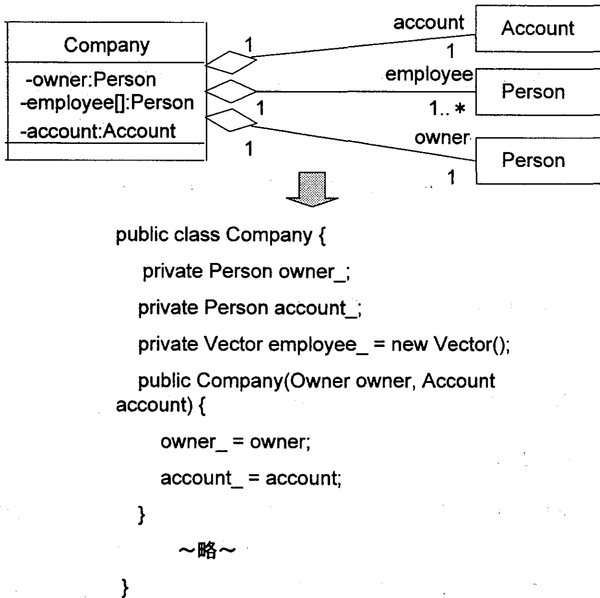


図 5: 集約が存在する場合の例

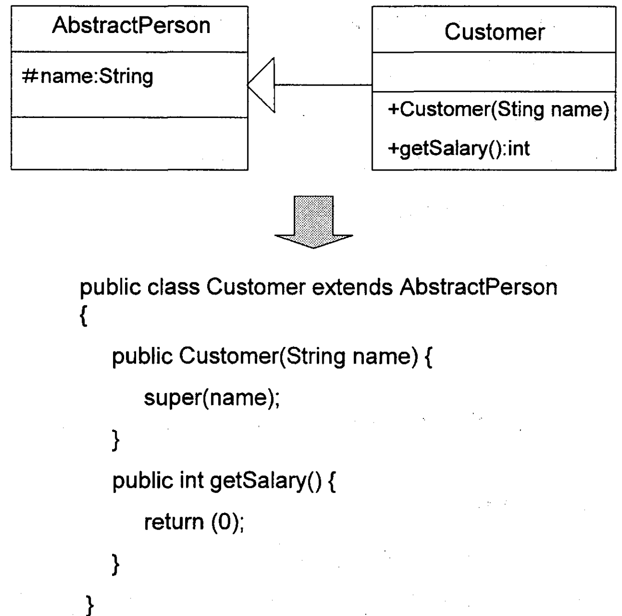


図 6: 汎化が存在する場合の例

```
private Vector employee_ = new Vector();
```

と宣言する必要がある。

3.2.3 コンポジション

Java の言語仕様では, 集約とコンポジションの区別をつけることはできない。このため, コンポジション関係がある場合, Java ソースコード上では, 集約関係と同様の定義を行う。

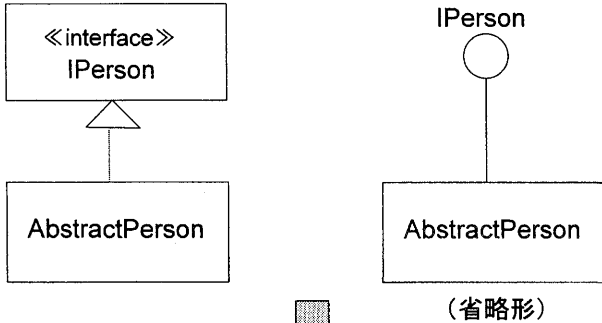
3.2.4 汎化

汎化関係がある場合, 矢印が付いてる側のクラスを継承する。Java では, 予約語 `extends` を用いて対応する。継承とは, すでに用意してあるクラスの属性と操作を引き継いで, 新しいクラスを作成することである。そのため, 汎化の関係がある場合, スーパークラスに宣言されている変数とメソッドを, 可視性が`private` でない限り, サブクラスで使用することが可能である。可視性が`private` の場合は, 変数が存在するクラス自身でのみ利用できることになるので, 汎化関係でも, 利用することはできない。

図 6 は, Customer クラスと AbstractPerson クラスが汎化関係であることを表している。Customer クラスが AbstractPerson クラスを継承しているので, Java ソースコード上では, Customer クラスを

```
public class Costomer extends
        AbstractPerson
```

と宣言する必要がある。また, ソースコードに変数`name` があるが, これは, クラス図の Customer クラスの属性には存在しない。しかし, 変数 `name` は, AbstractPerson クラスの変数であり, 可視性は`protected` であり, Customer クラスが AbstractPerson クラスのサブクラスであるので, 誤りではない。この場合, クラス図の Customer クラスの属性に存在してないが, 使用可能である。




```
publict class AbstractPerson implements IPerson {
    protected String name;
    protected String address;
    public AbstractPerson(String name) {
        this.name = name;
    }
    ~略~
}
```

図 7: インターフェースが存在する場合の例

3.2.5 洗練

洗練の関係は, ほとんどの場合, インターフェースを用いることによって, その関係を表す。Java では, インターフェースの実装を意味する予約語 `implements` を用いて対応する。インターフェースを用いない場合については, 事例が少ないため, 今回は対応関係を抽出できなかった。

図7は, AbstractPerson クラスがインターフェースである IPerson を洗練したものとなっていることを表す。 AbstractPerson クラスがインターフェースである IPerson を洗練しているので, Java ソースコード上では, AbstractPerson クラスを

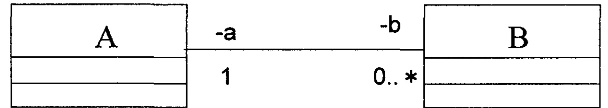
```
public class AbstractPerson implements
                IPerson
```

と宣言する必要がある。

3.2.6 依存

依存の関係がある場合, 矢印の元のクラスが先のクラスに依存することを表す。Java での依存の形態は様々であり, メソッドの引数や戻り値となる場合, メソッド中で`new` 演算子を使用する場合, インターフェースを利用する場合などがある。よって, 依存については, 一般的に言える対応関係は見出せなかった。

以上が, クラス間の関係と Java ソースコードとの対応関係である。この関係をまとめたものを表2に示す。



クラス型の配列を使用する場合

```
Class A{
        private B[] b = new B[];
                }
```

多重度が固定だった場合、この[]の中に上限値を入れる。

Vector型の配列を使用する場合

```
Class A{
        private Vector b = new Vector();
                }
```

図 8: 多重度が複数存在する場合の例

3.3 多重度

多重度は, クラスがインスタンスとの間にいくつ関連があるかを表すことから, 多重度が, 0 個または 1 個の場合は, Java では, 関連の場合と同様である。多重度が複数 (0〜N 個または 1〜N 個) である場合は, Java では, 多重度が複数の方のクラス型の配列や `Vector` 型の配列を使用して実現する必要がある。多重度が固定である場合は, 多重度の上限値の配列を実現する必要がある。また, 多重度の下限値については, Java では配列のインデックスなどで下限値を宣言することができないため, 今回は対応関係を抽出できなかった。Java では, 配列値を作成するには, `new` 演算子を使用する必要がある。

図8は, 多重度が複数存在する場合の例である。クラス A がクラス B のインスタンスを 0〜N 個保持できることを表している。また, 配列値の作成には, `new` 演算子を使用する必要がある。よって, Java ソースコード上では, クラス A の内部で

```
private B[] b = new B[];
```

と宣言する必要がある。もしくは, `Vector` 型の配列を使用する場合では, クラス A の内部で

```
private Vector b = new Vector();
```

と宣言する必要がある。もし, 多重度が固定の場合は, [] 内に, 多重度の上限値を入れることになる。

以上が, 多重度と Java ソースコードとの対応関係である。この対応関係をまとめたものを表3に示す。

表 2: クラス間の関係と Java ソースコードとの対応

クラス間の関係	関連	関連があるクラスのインスタンスを, インスタンス変数に格納して所持する。
	誘導可能性	関連に向きを持たした場合で, 関連と同様にインスタンスをインスタンス変数に格納するが, 矢印が付いた方のクラスは, インスタンス変数の宣言はされない。
	集約	変数を, 部分に当たるクラスのオブジェクト変数として実装する。
	コンポジション	Java の言語仕様では, 集約とコンポジションの区別をつけることはできず, 集約と同様である。
	汎化	矢印が付いてる側のクラスを継承する。クラス A がクラス B を継承する場合, クラス A は, `public class A extents B` と宣言される。サブクラスは, スーパークラスの変数と操作の可視性が`private` でない限り変数と操作を使用することが可能である。
	洗練	洗練は実現とも言われ, インターフェースの実装などに使用される。この場合の対応関係は, クラス A がインターフェースである B を実装する場合, `public class A implements B` と宣言される。他の場合については, 事例が少ないため, 今回は対応関係を抽出できなかった。
	依存	さまざまな場合に用いられるため, 現段階では, 一般的な対応関係が見出せない。

表 3: 多重度と Java ソースコードとの対応

多重度	多重度が 0 個または 1 個	ソースコードは, 関連の場合と同様に書く必要がある。
	多重度が複数	多重度が複数の方のクラス型の配列や`Vector` 型の配列を使用する。
	多重度が固定	多重度の上限値の配列を実現する必要がある。多重度の下限値については, 今回は対応関係を抽出できなかった。Java では, 配列を作成するには, 演算子`new` を使用する必要がある。

4. 考察

本研究は, UML のクラス図を用いて, オブジェクト指向プログラミング言語 Java で書かれたプログラムのテストを行ない, 信頼性を向上させることを目的としている。本稿では, その予備研究として, クラス図の要素と Java のソースコードとの対応関係を抽出した。

クラスの要素, 多重度, クラス間の関係として, 関連, 誘導可能性, 集約, 汎化, 洗練と Java のソースコードとの関係を抽出対象とした。しかし, いくつかの対応関係を抽出することができなかった。以下に抽出できなかった要素とその理由を挙げる。

- 多重度の下限値

  Java では, クラス図で下限値が 0 以外の数値を指定された場合でも, 下限値が 0 の場合と同様に, 多重度の上限値の配列を宣言して実現する以外に方法が無い。このため, 多重度の下限値と Java ソースコードとの対応関係を抽出できなかった。

- 洗練 (インターフェースを用いない場合)

  インターフェースを用いない場合の洗練関係については, 事例が少ないため発見できず, 対応関係を抽出できなかった。

- 依存

  依存関係は, Java では, メソッドの引数や戻り値となる場合, メソッド中で`new` 演算子を使用する場合, インターフェースを利用するなど, さまざまな場合に用いられるため, 現段階では, 一般的な対応関係が見出せなかった。

UML の図を用いてテストする手法として, eUML (embedded Unified Modeling Language) がある [7]。eUML では, 状態遷移表を導入したテストケース設計を用いている。UML のステートチャート図から作ら

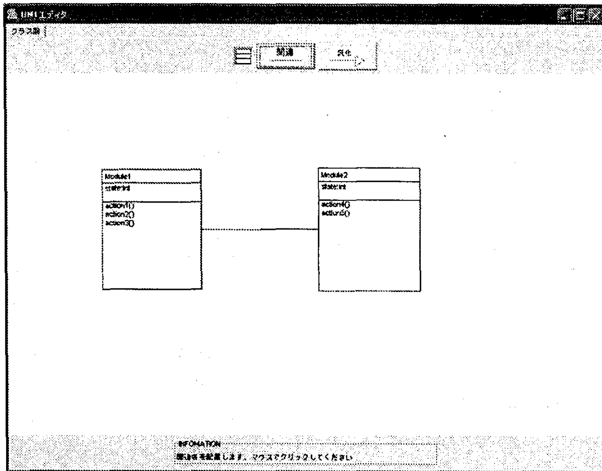


図 9: 作成中のツールの入力画面

れた状態遷移表を用いることにより, 要求仕様の「モレ」や「ヌケ」を発見することに対して効果的である。他にも, UML のステートチャート図を利用した統計的テスト手法が提案されている [8]。本研究では, Java で記述されたプログラムをテストする際に, クラス図を利用する。このため, プログラム (ソースコード) の構造が設計どおりに作られているかどうかを確認することに対して効果的だと考えられる。

UML を基にしたオブジェクト指向設計ツールの中には, クラス図から Java ソースコードのスケルトンを自動的に生成する機能を持つツールもいくつか存在する [3][4]。本稿では, クラス図の要素と Java ソースコードとの対応関係を抽出した。抽出した対応関係は, プログラミングが完了した Java プログラムに対して, テストの際に利用することが可能である。

現在, 本稿で抽出した対応関係を基に, クラス図を入力とし, Java ソースコードのテストすべき項目を出力するツールを作成中である。図 9 は, 作成中のツールの入力画面であり, 双方向に関連があるクラス図を描いたものである。

5. おわりに

本研究は, クラス図を利用したテストによる Java プログラムの信頼性向上を目的としている。本稿では, クラス図の要素と Java ソースコードとの対応関係を抽出した。抽出した対応関係を用いてソースコードをテストすることにより, プログラムの構造が設計どおりに作られているかどうかを確認することが可能になると考えられる。今後の課題としては, 以下が挙げられる。

- テスト支援ツールの作成

  今回は, UML のクラス図の要素と Java 言語で書かれたソースコードとの対応関係の抽出を行った。今後は, 抽出した対応関係を基に, UML のクラス図を入力とし, Java ソースコードのテストすべき項目を出力するツールを作成する必要がある。

- 抽出した対応関係の充分性の検証

  今回抽出した対応関係に対して充分性を検証する必要がある。多重度の下限値, 洗練 (インターフェースを用いない場合), 依存に関しては, 現段階では, 対応関係が見出せなかった。今後は, より多くの事例を参考にする事により Java ソースコードとの更なる対応関係の抽出に努める。

- UML の他のダイアグラムの利用

  クラス図は, 静的テストには適しているが, 動的テストには適していない。このため UML の他のダイアグラムも用いた, 統合的なテスト手法を考える必要がある。

参考文献

1) 株式会社オージス総研: “かんたん UML”, 翔泳社 (1999).

2) UML Home Page: http://www.uml.org/

3) Rational Rose: http://www-6.ibm.com/jp/software/rational/

4) VEST-SAVER for Java Ver1.3: http://www.vest.co.jp/

5) 日経コンピュータ: “テスト徹底に挑む企業 トラブルの目を見逃すな!”, 日経 BP 社, No.553, pp.40-41 (2002).

6) David Flanagan(イデア コラボレーションズ株式会社訳): “Java クイックリファレンスマニュアル 第 3 版”, オライリー・ジャパン (2000).

7) 渡辺博之, 渡辺政彦, 堀松和人, 渡守武和記: “組み込み UML—eUML によるオブジェクト指向組み込みシステム開発”, 翔泳社 (2002).

8) 高木智彦, 古川善吾: “プログラムの実行履歴を用いた利用モデルの作成方法について”, ソフトウェアシンポジウム 2003 予稿集, pp.41-48 (2003).