演習コースⅡ 形式手法と仕様記述

テスト駆動開発を利用した形式仕様記述

Formal Specification Using Test-Driven Development

伊藤　淳（ＮＥＣソフト株式会社）

研究概要

仕様を厳密に記述する手法である形式仕様記述を利用することで，仕様の検証が可能になり，上流工程における品質を高める効果が期待できる．形式仕様記述の特に事後条件は仕様を表す重要な要素であるが，記述が難しく誤りが起こりやすい．そこで，本研究では，テストケースを起点として仕様を記述することで，テストデータをヒントにして仕様を記述する手法を提案する．本手法は，従来難しかった事後条件における十分性の確認ができるという特長を持つ．

Abstract Formal specification methods provide rigorous ways for specification and allow for its verification leading to quality improvement in upper phases of software development. Although post-condition is a very essential element in formal specifications, its proper definition is a difficult and error-prone task. In response to this issue, this study proposes a specification method that starts with test cases to give insights for specification. As the most notable feature, this method allows for validation of post-condition.

1. はじめに

1.1 要求仕様の自然言語記述に起因する問題点

著者が所属する組織では，業務フロー等の一部の図を除き，要求仕様書（以下，仕様書）を自然言語（日本語）で記述することが一般的である．仕様書を自然言語で記述することで，曖昧な表現や暗黙知が記述されない等の弊害が生じ，上流工程においてバグが埋め込まれることがある．それを下流工程で発見すると，修正のための後戻り工数が大きくなるため，上流工程でバグを埋め込まないことが重要な課題になっている．

1.2 形式手法と形式仕様記述

仕様を厳密に記述する手法の１つとして，形式手法が注目されている．ここで，形式手法とは，数理論理学に基づき，システムの注目する側面を正確に曖昧さのない形で表現する手法であり，上流工程における品質を高める効果が期待できる[1]．この形式手法の１つである VDM（Vienna Development Method）には，オブジェクト指向の概念を持つ形式仕様記述言語として VDM++が存在する．形式仕様記述では，各機能に対して主に以下の３つを記述する．

- 陰仕様：関数・操作の入出力，制約条件のみを書き，関数・操作の本体が未定義の仕様
- 陽仕様：陰仕様で記述した関数・操作の本体に具体的な動作を記述した仕様
- テストケース：ある関数・操作に対し，入力となるテストデータと出力の期待値を定義し，仕様の検証するための記述

形式仕様記述では，テストケースを実行することにより，仕様を検証することが可能である．仕様を検証するとは，陽仕様に対して陰仕様で記述した制約条件に違反していないことを確認することである．制約条件には，事前条件，事後条件，不変条件がある．事前条件は関数・操作を実行する前の状態で成立すべき条件で，事後条件は関数・操作の実行

後の状態で成立すべき条件である．不変条件は，システム全体で必ず成立すべき条件である．この中で特に重要になるのが事後条件である．なぜなら，事後条件は陽仕様に対して，その内容の正しさを一般式の形で宣言的に記述したものであり，手続き的に記述した陽仕様とはその意味合いが異なる．対象となる関数や操作の仕様として表現したい重要な部分は，どのような手続きで結果を返すのかではなく，何をするのかである．この何をするのかに当たる部分が事後条件であり，仕様を記述する上で重要な役割を果たす．

なお不変条件は，事前条件と事後条件が扱えれば同様に扱えるため，本論文では言及しない．

1.3　形式仕様記述に関する課題

形式仕様記述に限らず，仕様を記述する際の誤解や考慮漏れ等で記述に誤りが含まれる可能性がある．もし，仕様の誤りに気が付かない状態で記述を続けると，誤りを残したまま記述を積み重ねてしまう．

形式仕様記述（VDM++）では，テストケースを実行することにより仕様を検証する仕組み持つ．仕様を記述する範囲を小さい機能単位にすれば，仕様の検証により素早くフィードバックが得られ，早い段階で仕様の誤りに気が付くことができるであろう．

ここで，仕様を記述する順序について考える．陰仕様，陽仕様を記述して後からその仕様を検証するためにテストケースを記述する場合と，テストケースを先に記述してから陰仕様，陽仕様を記述する場合があるとする．前者の場合，陰仕様と陽仕様の記述において，受け取る引数として何が必要であるか，また事前にどのような型を定義する必要があるか，そして，受け取った引数からどのような結果を返すかを考えることになるだろう．一方，後者の場合は，先にテストケースを定義するため，必然的にどのような操作が必要で，その操作に引数として具体的に何を与えると，何を返すことが正しいとするかを書くことになるだろう．後者は具体的なデータを例として記述することができるため，前者に比べて記述を支援できると考えられる．

つまり，テストケースを起点として記述することで具体的なデータをヒントにして仕様を記述することができると考えられる．さらに，小さい機能単位で検証することで，仕様の誤りを摘出しながら最小限の仕様を徐々に作り上げることができるだろう．

1.4　記述の誤りに関する課題

形式仕様記述は，成果物として制約条件も重要である．通常は陽仕様に対して，事後条件が違反検出の役割をするが，事後条件自体の誤りもあり得る．その場合，陽仕様と事後条件の矛盾がテストによって顕在化されれば，誤りを検出することができる．ただし，以下のように，事後条件の誤りの状態によっては，通常のテストでは顕在化されないという問題がある．

事後条件の誤りには，条件が”強い”場合と,”弱い”場合が考えられる．条件が強いとは，本来表現したい条件の一部しか受け入れない状態である．例えば,「辺１と辺２が等しく辺３が異なるとき二等辺三角形であると判定する」という条件があったとき，辺２と辺３が等しく,辺１が異なる組み合わせの二等辺三角形が正しく判定できない場合である．また，条件が弱いとは，本来表現したい条件よりも，大きい範囲で受け入れる状態である．例えば,「二辺が等しいとき二等辺三角形であると判定する」という条件があった場合，三辺が等しくても，二等辺三角形と判定されてしまう場合である．

ここで，テスト対象である陽仕様が正しかった場合に，“正しさの基準である事後条件が間違っていた”という状況を考えてみる．このとき,「正しい陽仕様」を「正しい（true)」と判定することを期待するテストケースを考えると，条件が強い状態の誤りは気が付くことができるが，条件が弱い状態の誤りは気が付くことができない．事後条件は宣言的に一般式を与えるものであり，この誤りは実際に発生しうる．（以下，表 1 参照）

事後条件が強い方向に誤った場合にテストを行うと,「正しい陽仕様」に対してテストが「失敗（false)」する．これは，テスト結果が誤って NG と判定され，原因を追及して考

えると事後条件の誤りにたどり着き，誤りに気が付くことができる．これに対し，事後条件が弱い方向に誤った場合にテストを行うと「正しい陽仕様」を「正しい（true)」と判定する.これは問題がないようだが,事後条件の十分性という観点では検証が不十分である．なぜなら，本来受け入れるべきでない要求を受け入れるような仕様になっているからである．

つまり，「正しい陽仕様を正しいと言える事後条件になっているか？」については，テストの結果，正しいはずがテストで失敗したという状況に気が付くと考えられる．一方，「誤った陽仕様を誤りと言える事後条件になっているか？」については，誤っているはずがテストに通ったという状況になる．しかし，テストが通った場合に，それが間違っていることを疑って分析することは考えにくく，通常はこの状況に気が付かないだろう．

そこで，事後条件の十分性を検証するためには，「正しい陽仕様」を「正しい（true)」と判定すること，「誤っている陽仕様」を「誤り（false)」と判定することの両面をテストする必要がある．前者は一般的な陽仕様のテストにより対応することができ，後者は，陽仕様とは「逆の視点」のテストとして，意識的に追加する必要がある．

表 1 条件の強・弱による誤り検出可否

	誤りの状態	テスト結果	詳細
事後条件	強い	NG	事後条件違反が発生するため，テストが失敗する．陽仕様とテストケースが正しいとすると，テスト結果が NG になるのは事後条件に原因があると考えられる．よって，事後条件に誤りがあると気が付くことができる．
	弱い	OK	テストがパスするため，陽仕様とテストケースが正しいとすると，事後条件に誤りがあることを疑う機会がない．そのため，事後条件に誤りがあっても，誤りがあると気が付くことができない．

1.5 研究の目的

1.3 では，テストデータを例として仕様を記述し，小さい機能単位で検証を行うことの利点について述べた．また，1.4 では，事後条件の十分性を確認するための方法について述べた．

そこで本研究では，以下の２点を達成することを目的とし，実現する手法として「テスト駆動開発を利用した形式仕様記述」を提案する．

- 目的 1：テストケースを仕様記述の起点とし，具体例をヒントにして陽仕様を記述できるようにすること
- 目的２：事後条件の十分性を確認するために，陽仕様のテストとは逆の視点のテストを事後条件に対して実施すること

本論文では，2 章で提案手法について，3 章で例題を用いた提案手法の説明について，4 章で結果と考察について述べる．

2. 提案手法について

2.1 テスト駆動開発とは

提案する手法に利用するテスト駆動開発（TDD：Test-Driven Development）とは，プログラム開発手法であり，テストしやすい設計を導く設計手法であるとも言われている[2][3]．テスト駆動開発のサイクルを以下に示す．

(1)． ある機能を作成するに先立ち，その機能のテストケースを追加する
(2)． テストケースが失敗するように，最初の失敗コードを記述する
(3)． 失敗するテストケースがパスするように必要最低限の実装を行う
(4)． 全てのテストケースを実行し，全てのテストケースがパスすることを確認する
(5)． テストケースがパスする状態を維持しながら，コードの重複を除去しクリーンなコードにする

テスト駆動開発の原則は，「失敗するテストを書くことなしに，新しい機能を追加してはならない」である[3]．例えば，存在しないメソッドをテストケースで呼び出すように書けば，それが必ずエラーになるため，エラーを解消するように実装すればよい．また，テスト駆動開発の原則に従うことで，将来必要になるかもしれないが当面は不要なコードや，不必要な機能の追加を思いとどまらせることができ，無駄に書きすぎることがなくなると言われている．さらに，テストを追加するたびに全テストを再実行することで，誤りに関するフィードバックを得て修正できるため，誤りを残したまま記述を積み重ねてしまうことを抑制できる．

2.2 提案手法の概要と手順

本研究では，テスト駆動開発のサイクルをフレームワークとして利用することで，仕様を繰り返しテストしながら，検証済みの仕様を徐々に作成する「テスト駆動開発を利用した形式仕様記述」の手法を提案する．本手法の概要は以下の通りである．

- 準備：テストリストの作成
- 第１ステップ：陽仕様に対する TDD
- 第２ステップ：事後条件に対する TDD
- 繰り返し

3. 例題を用いた提案手法の説明

3.1 マイヤーズの三角形問題

本手法を説明するに当たり，「ソフトウェア・テストの技法-第２版-」[4]に記載されている例題を題材とした．この例題は，三角形の三辺として，３つの値が与えられたとき，その三角形が正三角形，二等辺三角形，不等辺三角形のうちどれであるかを判定するというものである．この例題の意図としては，テストケースを検討し，考慮漏れがどれだけあるか，つまり様々なパターンを考慮することがいかに難しいかを問うことである．この例題の仕様はシンプルで前提知識を必要としないため，本手法を説明する例題として採用した．

本例題を使用するに当たり，説明を簡略化するための前提条件として最低限のルールを以下のように定めた．

- 三角形クラスは，三角形の種類を返す「種類を判定する」操作を持つものとする
- 「種類を判定する」操作は，引数として三辺を受け取り，種類を返す
- 種類は正三角形，二等辺三角形，不等辺三角形，三角形でない，の４つである
- 初期状態では，テストケースクラス，三角形クラスのみが存在し，操作は存在しないものとする

3.2 準備：テストリストの作成

はじめに，例題の仕様からテストリストを作成する．テストリストとは，作業を開始する前に，作成する必要があると考えられるテストケースを全て書きだしたものである．ここでは，テストリストとして図 1 を作成した．

・三辺の長さが等しいとき正三角形である
・二辺の長さが等しいとき二等辺三角形である
・三辺の長さが異なるとき不等辺三角形である

図 1 最初のテストリスト

以下，3.3，3.4 では，「テストケース」と「陰仕様・陽仕様」を操作単位で抜粋して記述する．表記の凡例を図 2 に示す．

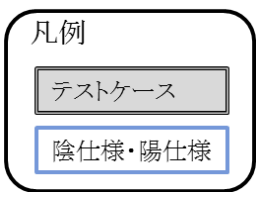

図 2 操作の凡例

3.3 第１ステップ：陽仕様に対する TDD

第１ステップとして，「種類を判定する」の陽仕様を記述するために，陽仕様に対する TDD を実施する．

(1). テストケースの追加

テストリストから１つ選択し，テストケースを作成する．ここでは，リストの先頭から開始することとし，「三辺の長さが等しいとき正三角形である」を選択する．3.1 に記述した例題の前提条件から「種類を判定する」操作は，“引数として三辺を受け取り，種類を返す”ことがわかる．これをテストケースとして記述すると，以下の図 3 のようになる．

```
public 三辺の長さが等しいとき正三角形である : () ==> ()
三辺の長さが等しいとき正三角形である() == (
  assertTrue(i 三角形.種類を判定する(1,1,1) = <正三角形>);
);
```

図 3 テストケース「三辺の長さが等しいとき正三角形である」

(2). 最初の失敗コードを記述

三角形クラスには「種類を判定する」操作が定義されていないため，上記テストケースは文法エラーになる．そこで，三角形クラスに「種類を判定する」操作を追加する．引数として受け取る三辺は型を「int」とし，戻り値を「種類」とする．以降はこの自明な失敗に関する記述は省略する．

```
public 種類を判定する : int * int * int ==> 種類
種類を判定する(a 辺 1,a 辺 2,a 辺 3) == (
  is not yet specified
);
```

図 4 陽仕様未定義の操作「種類を判定する」

文法エラーを全て解消すると，テストケースを実行することができる．しかし，「種類を判定する」操作には，陽仕様が定義されていない（is not yet specified）ため，実行すると失敗する．

(3). 失敗するテストケースがパスするように必要最低限の実装を行う

引数（1,1,1）に対してのみ動作するように記述し，テストケースが必ずパスする陽仕様を作成する．

(4). 全てのテストケースを実行し，全てのテストケースがパスすることを確認する

現在はテストケースが１つのみであるが，複数存在する場合はその全てを実行し，全てのテストケースがパスすることを確認する．ここで，失敗になるテストケースがあった場合は，新たに追加したテストケースによって作成した陽仕様に誤りがあることになるだろう．

(5). テストケースがパスする状態を維持しながら，コードの重複を除去しクリーンなコードにする

現在の陽仕様は引数（1,1,1）しか受け付けない仕様になっているため，テストケースに同じ意味のテストデータとして引数（2,2,2）のアサート文を追加する．この状態でテストケースを実行すると失敗する．引数が（1,1,1），（2,2,2）のどちらも満たすように陽仕様を徐々に修正していくと，図 5 のように記述できる．この状態で再度全てのテストケースを実行し，全てのテストケースがパスすることを確認する．

```
public 種類を判定する : int * int * int ==> 種類
種類を判定する(a 辺 1,a 辺 2,a 辺 3)== (
   dcl w 種類 : 種類 := <三角形でない>;
   if (a 辺 1 = a 辺 2 and a 辺 2 = a 辺 3) then (
     w 種類 := <正三角形>;
   )
   return w 種類;
);
```

図 5 陽仕様本実装の操作「種類を判定する」

3.4 第２ステップ：事後条件に対する TDD

次に第２ステップとして，「種類を判定する」に事後条件を追加するために，事後条件に

対する TDD を実施する．

(1)．事後条件のテストケースの追加

「種類を判定する」操作の陽仕様をヒントに，この操作の事後条件を検討する．図 5 で記述した陽仕様が表す意味は，辺 1 と辺 2 が等しくかつ辺 2 と辺 3 が等しいとき，つまり三辺が等しいとき正三角形を返すことを表している．ここで，操作「種類を判定する」に対する事後条件を考えた場合，例えば「三辺が等しいならば種類は正三角形と判定される」と書くことができるだろう．

次に，事後条件の十分性を確認するために，1.3 で述べた陽仕様に対するテストとは逆の視点のテストケースを作成する．ここで逆の視点とは，「誤っている陽仕様」を「誤り（false)」と判定することである．つまり，「三辺が等しくないならば種類は正三角形と判定されない」というテストケースを作成すればよい．このとき，テストデータは三辺が等しくない値「3,4,5」のとき，種類が「正三角形」と判定されない（戻り値が false）になる．これをテストケースとして記述すると，以下の図 6 のようになる．

```
public 三辺が等しくないならば種類は正三角形と判定されない : () ==> ()
三辺が等しくないならば種類は正三角形と判定されない() == (
  assertTrue(i 三角形.三辺が等しいならば種類は正三角形と判定される
                                    (3,4,5,<正三角形>) = false);
);
```

図 6 事後条件のテストケース「三辺が等しくないならば種類は正三角形と判定されない」

(2)．事後条件の最初の失敗コードを記述

まず「種類を判定する」操作の事後条件として，「三辺が等しいならば種類は正三角形と判定される」を追加する．このとき，三角形クラスにはまだこの操作が定義されていないため文法エラーになり，陽仕様を記述した際と同様の手順で，文法エラーの箇所を徐々に修正していくと，以下の図 7 と図 8 のように記述できる．

文法エラーを全て解消して実行できる状態になったところでテストケースを実行し，失敗することを確認する．

```
public 種類を判定する : int * int * int ==> 種類
種類を判定する(a 辺 1,a 辺 2,a 辺 3)== (
･･･省略･･･
)
post 三辺の長さが等しいならば正三角形と判定される(a 辺 1,a 辺 2,a 辺 3,RESULT);
```

図 7 「種類を判定する」操作に事後条件を追加

```
public 三辺が等しいならば種類は正三角形と判定される : int * int * int * 種類 ==> bool
三辺が等しいならば種類は正三角形と判定される(a 辺 1,a 辺 2,a 辺 3,a 種類) == (
  is not yet specified;
);
```

図 8 陽仕様未定義の操作（別定義した事後条件）

(3)．失敗するテストケースがパスするように必要最低限の実装を行う

テストケース「三辺が等しくないならば種類は正三角形と判定されない」（図 6）がパスするように，別定義した事後条件「三辺が等しいならば種類は正三角形と判定される」（図 8）では，固定値「false」を返すようにする．

(4)．全てのテストケースを実行し，全てのテストケースがパスすることを確認する

全てのテストケースを実行すると，新たに追加した図 6 のテストケースはパスするが，図 3 の陽仕様のテストケースは「種類を判定する」操作の事後条件が常に false になるた

め，事後条件違反となり失敗する．これは，図 8 で本体を正しく記述していないことが原因である．そこで，図 8 の事後条件の名称と図 5 の陽仕様をヒントにして図 8 の本体を記述する．“三辺が等しい”とは，陽仕様から「a 辺 1 = a 辺 2 and a 辺 2 = a 辺 3」と記述でき，“P ならば Q と判定される”とは，「P => Q」と記述できる．そして，“種類は正三角形”とは，「a 種類 = <正三角形>」と記述できる．よって，図 8 の本体は以下の図 9 のように記述できる．

```
public 三辺の長さが等しいならば正三角形と判定される : int * int * int * 種類 ==> bool
三辺の長さが等しいならば正三角形と判定される(a 辺 1,a 辺 2,a 辺 3,a 種類) == (
  return (a 辺 1 = a 辺 2 and a 辺 2 = a 辺 3) => a 種類= <正三角形>;
);
```

図 9 陽仕様仮実装の操作（別定義した事後条件）

この状態で再度全テストケースを実行すると，今度は図 6 のテストケースが失敗する．これは，図 9 で記述した本体に誤りがあることが原因である．図 6 のテストケースは「三辺が等しくないならば種類は正三角形と判定されない」であるが，三辺が「3,4,5」は「(a 辺 1 = a 辺 2 and a 辺 2 = a 辺 3)」を満たさず，「P ならば Q」の「P」が「false」になる．「P」が「false」のとき，「P ならば Q」の判定結果は，「Q」の値によらず「true」になるため，図 6 のテストケースは失敗していた．この失敗を論理的に正しく修正すると，図 10 のように記述できる．

```
public 三辺の長さが等しいならば正三角形と判定される:int * int * int * 種類 ==> bool
三辺の長さが等しいならば正三角形と判定される(a 辺 1,a 辺 2,a 辺 3,a 種類) == (
  return (a 辺 1 = a 辺 2 and a 辺 2 = a 辺 3) <=> a 種類= <正三角形>;
);
```

図 10 陽仕様本実装の操作（別定義した事後条件）

(5)．テストケースがパスする状態を維持しながら，コードの重複を除去しクリーンなコードにする

事後条件の内容を変更したため，それに合わせて操作名称も変更する．「三辺の長さが等しいならば正三角形と判定される」かつ，「三角形と判定されるならば三辺の長さは等しい」とは，つまり「三辺の長さが等しいときのみ正三角形と判定される」ということである．操作名称変更に伴い，呼び出し元である図 6，図 7 が文法エラーになるため，同様に名称を修正する．

3.5 繰り返し

テストリストからテストケースとして作成済みのものを削除し，適切なものを選択して第 1 ステップ，第 2 ステップを繰り返す．また，新たにテストリストとして追加すべきと気が付いた場合，その都度テストリストに追加する．

例えば，図 5 の陽仕様を記述した段階で，「三辺のうち長さが 0 以下の辺があるとき三角形ではない」ということに気が付いた場合，その時点でテストリストに追加する．

4. 結果と考察

4.1 テスト駆動開発を利用したことによる効果

本手法の「目的 1」については，3 章のシナリオでテストケースごとに小さい機能単位で仕様を記述することで，テストデータをヒントにしながら陽仕様を記述することができた．また，陽仕様を記述した後に，陽仕様をヒントにして事後条件のテストケースを記述することで，一般式の形で宣言的に記述する事後条件を，具体例から徐々に一般化して記述することができるようになり，事後条件の記述を支援する効果が得られた．

さらに，テストケースを起点としているため，容易に回帰テストを実施することが可能

になり，誤りを残したまま仕様を記述し続けることがなくなった．

4.2　事後条件に対するテストを追加したことによる効果

本手法の「目的２」については，3 章のシナリオで，事後条件をテストするためのテストケースを，陽仕様とは逆の視点で作成する手順を組み込んだ．

3.3 の手順だけでは，陽仕様に対するテストで失敗しないため，仕様に誤りがないと判断してしまい，さらにそれを疑ってテストケースを追加することは通常行わないと考えられる．しかし，実際には事後条件が弱い方向に誤っている場合が考えられ，本手法で追加した 3.4 の手順では，その誤りを検出することができた．

つまり，事後条件の十分性を確認するためには，陽仕様をテストすることと，陽仕様とは逆の視点で事後条件をテストすることの両方が必要であり，本手法ではその確認ができることを示した．

4.3　検出できない仕様の誤りに関する議論

本手法を利用しても検出できない仕様の誤りについて言及する．本手法を利用しても検出できない仕様の誤りは，そもそもテストケースを思いついていない場合である．テストケースを思いついていなければ，テストケースを作成できず，仕様に誤りがあっても検出できない．これは提案手法の限界ではなく，潜在する仕様の誤りに対して，そもそもテスト観点が思いついていない場合，その誤りを検出することは本質的に不可能だと言うことを示している．

また，テストケースのテストデータに誤りがある場合も，仕様の誤りを検出することはできない．テストケースは仕様の正しさを確認するための基準であり，その基準が間違っている場合は，仕様の誤りを検出することはできない．

ただし，テストケースは具体例であり，実感・確信を持って仕様の妥当性の判断する材料として利用しやすい．これに対し，一般式を宣言的に記述する事後条件は，その妥当性を確信することが難しいと考えられる．テストケースをヒントにして仕様を記述する本手法は，記述が難しい事後条件に対して，記述を助ける効果があることを示した．

5.　おわりに

本研究では，テスト駆動開発を利用することでテストケースの具体例をヒントにして仕様が記述できること，そして，事後条件をテストすることで見逃しがちだった誤りを検出しながら仕様が記述できることの２点を実現する手法を提案した．

本手法の徐々に仕様を作り上げていく特長は，形式仕様記述（VDM++）初学者に対して，理解を促す効果が期待できると考えている.今後は,学習教材としての利用価値の検証や，QCD（学習コスト，実装コスト，品質）に関して，本手法の優位性を検証したい．

謝辞

本論文の執筆に当たり,演習コースⅡアドバイザーの荒木啓二郎氏,主査の栗田太郎氏,副主査の石川冬樹氏，研究員の皆様，日本科学技術連盟の事務局の皆様に大変お世話になりました．厚くお礼申し上げます．

参考文献

[1] 形式手法の考え方｜ディペンダブル・システムのための形式手法の実践ポータル, http://formal.mri.co.jp/outline/fm-wot.html, 2014/01/08 アクセス.

[2] Kent Beck,『テスト駆動開発入門』,ピアソン・エデュケーション,2003

[3] Steve Freeman, Nat Pryce,『実践テスト駆動開発-テストに導かれてオブジェクト指向ソフトウェアを育てる-』,翔泳社,2012

[4] Glenford J. Myers, Todd M. Thomas, Tom Badgett, Corey Sandler,『ソフトウェア・テストの技法 第 2 版』, 近代科学社,2006

付録

本手法を検討するに当たり，３章で用いた「マイヤーズの三角形問題」を本手法で記述した．付録として作成したファイルの一覧とその内容を記載する．なお，テストケースを実行するためのファイルや使用したライブラリに関する記載は省略する．

ファイル一覧
- -テストケース.vdmpp
- -三角形.vdmpp

以下，テストケースクラスの内容を図 11，図 12，図 13 の３つに分けて記載する．

```
class テストケース is subclass of TestCase

instance variables
  i三角形 : 三角形;
operations
  public テストケース : seq of char ==> テストケース
  テストケース(aName) == setName(aName);

  public runTest : () ==> ()
  runTest() == (
    setup();
    三辺の長さが等しいとき正三角形である();
    「三辺の長さが等しいときのみ正三角形と判定される」ことは正しい();
    三辺の長さが等しくないならば正三角形と判定されない();
    二辺の長さが等しいとき二等辺三角形である();
    二辺の長さが等しくないならば二等辺三角形と判定されない();
    三辺の長さが異なるとき不等辺三角形である();
    三辺の長さが異ならないならば不等辺三角形ではない();
    ある二辺の長さの和が他の一辺の長さ以下のとき三角形ではない();
    ある二辺の長さの和が他の一辺の長さより大きいとき三角形でないと判定されない();
    三辺のうち長さが0以下の辺があるときエラーを返す();
  );

  public setup : () ==> ()
  setup() == (
    i三角形 := new 三角形();
  );
```

図 11　テストケースファイル（前半）

```
  public 三辺の長さが等しいとき正三角形である : () ==> ()
  三辺の長さが等しいとき正三角形である() == (
    assertTrue(i三角形.種類を判定する(1,1,1) = <正三角形>);
  );
  public 三辺の長さが等しくないならば正三角形と判定されない : () ==> ()
  三辺の長さが等しくないならば正三角形と判定されない() == (
    assertTrue(i三角形.三辺の長さが等しいときのみ正三角形と判定される(1,2,2,<正三角形>) = false);
    assertTrue(i三角形.三辺の長さが等しいときのみ正三角形と判定される(2,3,4,<正三角形>) = false);
    assertTrue(i三角形.三辺の長さが等しいときのみ正三角形と判定される(1,1,3,<正三角形>) = false);
    assertTrue(i三角形.三辺の長さが等しいときのみ正三角形と判定される(1,2,4,<正三角形>) = false);
    assertTrue(i三角形.三辺の長さが等しいときのみ正三角形と判定される(1,2,3,<正三角形>) = false);
  );

  public 二辺の長さが等しいとき二等辺三角形である : () ==> ()
  二辺の長さが等しいとき二等辺三角形である() == (
    assertTrue(i三角形.種類を判定する(1,2,2) = <二等辺三角形>);
    assertTrue(i三角形.種類を判定する(2,1,2) = <二等辺三角形>);
    assertTrue(i三角形.種類を判定する(2,2,1) = <二等辺三角形>);
  );
  public 二辺の長さが等しくないならば二等辺三角形と判定されない : () ==> ()
  二辺の長さが等しくないならば二等辺三角形と判定されない() == (
    assertTrue(i三角形.二辺の長さが等しいときのみ二等辺三角形と判定される(2,3,4,<二等辺三角形>) =
false);
    assertTrue(i三角形.二辺の長さが等しいときのみ二等辺三角形と判定される(1,2,3,<二等辺三角形>) =
false);
    assertTrue(i三角形.二辺の長さが等しいときのみ二等辺三角形と判定される(1,2,4,<二等辺三角形>) =
false);
  );

  public 三辺の長さが異なるとき不等辺三角形である : () ==> ()
  三辺の長さが異なるとき不等辺三角形である() == (
    assertTrue(i三角形.種類を判定する(2,3,4) = <不等辺三角形>);
    assertTrue(i三角形.種類を判定する(2,4,3) = <不等辺三角形>);
    assertTrue(i三角形.種類を判定する(3,2,4) = <不等辺三角形>);
    assertTrue(i三角形.種類を判定する(3,4,2) = <不等辺三角形>);
    assertTrue(i三角形.種類を判定する(4,2,3) = <不等辺三角形>);
    assertTrue(i三角形.種類を判定する(4,3,2) = <不等辺三角形>);
  );
```

図 12 テストケースファイル（中盤）

```
  public 三辺の長さが異ならないならば不等辺三角形ではない : () ==> ()
  三辺の長さが異ならないならば不等辺三角形ではない() == (
    assertTrue(i三角形.三辺の長さが異なるときのみ不等辺三角形と判定される(1,1,1,<不等辺三角形>) =
false);
    assertTrue(i三角形.三辺の長さが異なるときのみ不等辺三角形と判定される(1,2,2,<不等辺三角形>) =
false);
    assertTrue(i三角形.三辺の長さが異なるときのみ不等辺三角形と判定される(1,1,3,<不等辺三角形>) =
false);
  );

  public ある二辺の長さの和が他の一辺の長さ以下のとき三角形ではない : () ==> ()
  ある二辺の長さの和が他の一辺の長さ以下のとき三角形ではない() == (
    assertTrue(i三角形.種類を判定する(1,2,3) = <三角形でない>);
    assertTrue(i三角形.種類を判定する(1,1,3) = <三角形でない>);
    assertTrue(i三角形.種類を判定する(1,2,4) = <三角形でない>);
  );
  public ある二辺の長さの和が他の一辺の長さより大きいとき三角形でないと判定されない : () ==> ()
  ある二辺の長さの和が他の一辺の長さより大きいとき三角形でないと判定されない() == (
    assertTrue(i三角形.ある二辺の長さの和が他の一辺の長さ以下のとき三角形でないと判定される
(1,1,1,<三角形でない>) = false);
    assertTrue(i三角形.ある二辺の長さの和が他の一辺の長さ以下のとき三角形でないと判定される
(1,2,2,<三角形でない>) = false);
    assertTrue(i三角形.ある二辺の長さの和が他の一辺の長さ以下のとき三角形でないと判定される
(2,3,4,<三角形でない>) = false);
  );

  public 三辺のうち長さが0以下の辺があるときエラーを返す : () ==> ()
  三辺のうち長さが0以下の辺があるときエラーを返す() == (
    異常系判定_種類を判定する(0,0,0);
    異常系判定_種類を判定する(0,1,1);
    異常系判定_種類を判定する(1,0,1);
    異常系判定_種類を判定する(1,1,0);
  );

  private 異常系判定_種類を判定する : int * int * int ==> ()
  異常系判定_種類を判定する(a辺1,a辺2,a辺3)== (
    tixe {
      <RuntimeError> |-> IO`print("想定通りエラー発生¥n")
    } in (
      def - = i三角形.種類を判定する(a辺1,a辺2,a辺3) in skip;
      fail("エラーが発生しない場合失敗¥n");
    );
  );

end テストケース
```

図 13 テストケースファイル（後半）

以下，三角形クラスの内容を図 14，図 15 の２つに分けて記載する．

```
class 三角形

  types
    public 種類 = <正三角形> | <二等辺三角形> | <不等辺三角形> | <三角形でない>;

  operations
    public 種類を判定する : int * int * int ==> 種類
      種類を判定する(a辺1,a辺2,a辺3)== (
        dcl 結果 : 種類 := <三角形でない>;
        if (a辺1 + a辺2 <= a辺3 or a辺1 + a辺3 <= a辺2 or a辺2 + a辺3 <= a辺1) then (
          結果 := <三角形でない>;
        ) else if (a辺1 = a辺2 and a辺2 = a辺3) then (
          結果 := <正三角形>;
        ) else if (a辺1 = a辺2 or a辺2 = a辺3 or a辺1 = a辺3) then (
          結果 := <二等辺三角形>;
        ) else (
          結果 := <不等辺三角形>;
        );
        return 結果;
      )
      pre 三辺の長さはすべて0より大きいこと(a辺1,a辺2,a辺3)
      post ある二辺の長さの和が他の一辺の長さ以下のとき三角形でないと判定される(a辺1,a辺2,a辺
3,RESULT) and
          三辺の長さが等しいときのみ正三角形と判定される(a辺1,a辺2,a辺3,RESULT) and
          二辺の長さが等しいときのみ二等辺三角形と判定される(a辺1,a辺2,a辺3,RESULT) and
          三辺の長さが異なるときのみ不等辺三角形と判定される(a辺1,a辺2,a辺3,RESULT);
```

図 14 三角形クラスファイル（前半）

```
    public 三辺の長さはすべて0より大きいこと : int * int * int ==> bool
      三辺の長さはすべて0より大きいこと(a辺1,a辺2,a辺3) == (
        return a辺1 > 0 and a辺2 > 0 and a辺3 > 0;
      );

    public 三辺の長さが等しいときのみ正三角形と判定される : int * int * int * 種類 ==> bool
      三辺の長さが等しいときのみ正三角形と判定される(a辺1,a辺2,a辺3,a種類) == (
        return a種類 <> <三角形でない> and (a辺1 = a辺2 and a辺2 = a辺3) <=> a種類 = <正三角形>;
      );

    public 二辺の長さが等しいときのみ二等辺三角形と判定される : int * int * int * 種類 ==> bool
      二辺の長さが等しいときのみ二等辺三角形と判定される(a辺1,a辺2,a辺3,a種類) == (
        return a種類 <> <三角形でない> and a種類 <> <正三角形> and (a辺1 = a辺2 or a辺2 = a辺3 or
a辺1 = a辺3) <=> a種類 = <二等辺三角形>;
      );

    public 三辺の長さが異なるときのみ不等辺三角形と判定される : int * int * int * 種類 ==> bool
      三辺の長さが異なるときのみ不等辺三角形と判定される(a辺1,a辺2,a辺3,a種類) == (
        return a種類 <> <三角形でない> and (a辺1 <> a辺2 and a辺2 <> a辺3 and a辺1 <> a辺3) <=>
a種類 = <不等辺三角形>;
      );

    public ある二辺の長さの和が他の一辺の長さ以下のとき三角形でないと判定される : int * int * int *
種類 ==> bool
      ある二辺の長さの和が他の一辺の長さ以下のとき三角形でないと判定される(a辺1,a辺2,a辺3,a種類) ==
(
        return (a辺1 + a辺2 <= a辺3 or a辺1 + a辺3 <= a辺2 or a辺2 + a辺3 <= a辺1) <=> a種類 = <
三角形でない>;
      );

end 三角形
```

図 15 三角形クラスファイル（後半）

以上.