

2000年度 修士論文

ソフトウェアテストのパターンの  
存在可能性に関する研究

指導教授：深澤 良彰 教授

早稲田大学大学院理工学研究科 情報科学専攻 深澤研究室

学籍番号：g699p012-4

太田健一郎

提出日 2月5日

# 目次

第1章	はじめに	4
1.1	研究の目的	4
1.2	パターンの現状と問題	4
1.3	テストにおけるパターンの現状	5
1.3.1	テストにおけるパターンの少なさ	5
1.3.2	存在するいくつかのパターン	5
1.3.3	行うべきこと	6
第2章	テストのパターンが発見しにくい理由	7
2.1	原因の分類	7
2.2	ソフトウェア業界全体の問題	7
2.2.1	大企業における下請け構造の問題	7
2.2.2	中小企業における品質軽視の問題	8
2.2.3	管理と精神論による問題のすり替え	8
2.2.4	再利用可能なテストに関する問題	8
2.2.5	テスト技術の共有の問題	9
2.3	パターンそのものの問題	9
2.3.1	テストの分野におけるパターンの理解の問題	9
2.3.2	パターンの概念が普及している分野の問題	10
2.3.3	既存の概念に対する問題	10
2.4	テストの定義における問題	10
2.4.1	テストの定義	11
2.5	技術者の態度における問題	12
2.5.1	テストの否定	12
2.5.2	テストへの誤解	12
2.5.3	テストレイト	13
2.5.4	銀の弾丸	13
2.6	問題のまとめ	13
第3章	テストにおけるパターン発見のための解決策	14
3.1	解決策の分類	14
3.2	正しいテストを行っていないプログラマに対して	14
3.2.1	真のテストへの理解	14
3.2.2	テストファースト	15

3.3	テストを行っているプログラマに対して	17
3.3.1	パターンのカタログ	17
3.3.2	パターンの書き方	17
3.3.3	パターンの概念	18
3.3.4	心の余裕	19
3.3.5	おのずから生まれるもの	19
3.4	テスト担当者と品質管理技術者に対して	19
3.4.1	2つの解決策	19
第4章	非公式の単体テストのためのパターンランゲージ	21
4.1	発見のプロセス	21
4.2	パターンランゲージの概要	21
4.3	アンチパターン：クイックハック	24
4.4	パターン名 サンドイッチアプローチ	26
4.5	パターン名 石橋叩き	27
4.6	パターン名 子供のテスト (別名 とりあえず実行)	29
4.7	パターン名 3度目の自動化	30
第5章	パターンの学習と発見	32
5.1	はじめに	32
5.1.1	目的	32
5.1.2	方法	32
5.1.3	流れ	32
5.1.4	評価方法	33
5.1.5	与える知識	33
5.2	学習前の評価	33
5.2.1	目的	33
5.2.2	方法	33
5.2.3	問題	34
5.2.4	アンケート	35
5.2.5	想定する欠陥	36
5.2.6	有効なテスト戦略、用いる知識	36
5.2.7	模範テストケースと結果	37
5.2.8	各テストケースの意味	38
5.2.9	被験者	38
5.2.10	被験者 A に対する評価	39
5.2.11	被験者 B に対する評価	40
5.2.12	被験者 C に対する評価	41
5.2.13	被験者 D に対する評価	43
5.2.14	アンケートの結果と評価	44
5.2.15	総論	48
5.3	学習	48

5.4	学習後の評価	49
5.4.1	目的	49
5.4.2	方法	49
5.4.3	問題	49
5.4.4	アンケート	49
5.4.5	想定する欠陥	51
5.4.6	有効なテスト戦略、用いる知識	51
5.4.7	模範テストケースと結果	51
5.4.8	各テストケースの意味	51
5.4.9	被験者	51
5.4.10	被験者 A に対する評価	51
5.4.11	被験者 B に対する評価	53
5.4.12	被験者 C に対する評価	54
5.4.13	アンケートの結果と評価	55
5.4.14	総論	58
第 6 章 結論		61
付 録 A パターンにおける実例		63
A.1	サンドイッチアプローチパターンの例	63
A.2	石橋叩きパターンの例	64
A.3	子供のテストパターンの例	70

# 第1章 はじめに

## 1.1 研究の目的

デザインパターン [Gamma+95] に始まるソフトウェアのパターンは今や数千を超え、パターンという言葉は広くソフトウェア技術者に知られるようになっている。もはやパターンは十分であるという声も聞かれる。

しかし、ソフトウェアにおいて十分パターンの概念が普及していない領域も依然として存在している。テストもその1つである。テストにおいて常にいわれていることは現場のテスト技術者とテスト理論の乖離である。現場のテスト技術者がテスト理論、テスト手法を学ぶときに発する言葉が、「この理論は確かに有用であるように思えるが、現場でどう実践してよいのかいまいち分からない」というものである。

パターンはソフトウェアのほかの分野において理論と実践の橋渡しをしてきた。本研究ではテストにおいてもパターンはこの役割を果たすことが可能であると考え、テストにおけるパターンがどのようなものであるかを考察し、最終的にテストにかかわるすべての人々はほかのソフトウェアのパターンと同じように潜在的にテストのパターンを持っており、それを書くことが出来るということを示す。そして、同時にテストにおいてパターンを学ぶことによりどのような効果があるのかをテスト手法と比較しながら示す。

本研究が、テストの分野においてパターンの概念が普及し、設計者、プログラマ、テスト担当者、品質管理技術者がパターン・ランゲージを共有する第1歩となることを期待する。

## 1.2 パターンの現状と問題

長年ソフトウェアを開発してきた人なら、ほかの人からの苦労話を聞かされて、それは自分も経験したと思ったことが一度や二度はあるはずである。そして、その人も長い時間をかけて自分と同じ解決策に至ったことを知り、「先に話を聞いていれば、助けになったかもしれないのに」と考えたことも何度かあるはずである。こうしたソフトウェア開発で繰り返し起こる問題と、それに対して皆が繰り返し使っている解法を共有し再利用するのがパターンの目的である。

パターンは特に、今まで熟練者頼みにしてきた知識をうまく再利用できる方法として注目されている。ソフトウェア開発に必要な知識のうち、一部の知識、例えばアルゴリズムやデータ構造などは、すでに広く共有され大学の教育課程にも組み込まれている。しかし他の知識、例えば分析のノウハウや設計のテクニック、プロジェクト管理の知恵などは、各自が経験の中から学んでいるのが実状で、なかなかうまく共有できていない。パターンは、こうした知識を理解しやすい表現で簡潔に書いたものである。

パターンというと、デザインパターン [Gamma+95] を思い浮かべる場合が多い。確かに「デザインパターン」は最もすぐれたパターン・ランゲージの1つだが、ほかにも下記のような多

種多様のパターンがある。

- 2 分析のためのパターン
- 2 設計のためのパターン
- 2 実装のためのパターン
- 2 アンチパターン
- 2 プロジェクト管理のためのパターン

このようにデザインパターンに始まるソフトウェアパターンの流れはPLoP(Pattern Languages of Programs)のようなパターンコミュニティ活動と共に多種多様なパターンを生み出した。パターンの数は今や数千ともいわれ、大抵の問題はその分野のパターンを参照し、利用することによって解決できるようになっているといっても過言ではない。

しかしながら、ソフトウェアの分野によってはパターンの概念がほとんど知られておらず、パターンがほとんど存在しない分野も存在する。本研究の対象とするテストもその1つである。これらの分野に携わる人々は実は最もパターンを求めているのにもかかわらずパターンが存在していない。それには様々な理由がある。本研究ではその理由を探り、それらの領域においてパターンを発見するためにはどのようなことをしたらよいかの提案を行うことを試みる。

## 1.3 テストにおけるパターンの現状

### 1.3.1 テストにおけるパターンの少なさ

「パターンの現状と問題」で述べたように、数多くのソフトウェアパターンが存在するなか、テストのパターンというのはソフトウェアパターン全体に対して余りにも少ない。

これは、`comp.software.testing`のようなネットワーク上のニュースグループを見ていてもよく分かる。あるテスト担当者が極めて具体的な問題に対する解答を求めた場合、例えばテストツールのある関数の使い方についての質問を出した場合、これに対する解答はすぐに得ることが出来る。しかし、ある種の抽象度を持った問題、例えば、クラスのprivateメソッドはいつテストすべきなのか、privateメソッドにもいくつか種類があるがどうしたらよいかという問題に対しては、何々という本を読んで自分で解答を見つけるようにという答えしか返ってこない場合が多い。

これらの議論を見ているとき、彼らはパターン・ランゲージを用いて語り合えばもっと迅速に答えを得ることが出来るのにと感じる。残念ながら、現状では多くの人々はパターンについてほとんど知らないし、テストについて語り合うパターン・ランゲージはほとんど存在していない。

### 1.3.2 存在するいくつかのパターン

とはいえ、テストの分野においてもいくつかのパターンが既に存在している。現在、以下のものを確認している。

- 2 "Testing Object-Oriented Systems Models Patterns, and Tools"[Binder 99] で述べられている 37 種類のテストデザインパターン、17 種類のテスト自動化デザインパターン、16 種類のテストオラクルマイクロパターン
- 2 "Patterns for System Testing"[DeLano+96] で述べられている 20 種類のシステムテストのパターン
- 2 "Testing Patterns"(http://c2.com/cgi-bin/wiki?TestingPatterns) で述べられている 6 種類のパターンと 3 種類のアンチパターン
- 2 "Testing Patterns"[Grand 99] で述べられている 8 種のパターン

各パターンの対象としている領域は大きく異なる。[Binder 99] では主としてテストを設計する際に生じる問題に対し焦点を当てている。[DeLano+96] はシステムテストを行う際に生じる人為的な問題に焦点を当てている。[Grand 99] はテスト手法をパターンの形式で記述している。各パターンを比較してみるとテストにおける問題は様々な領域に分かれており、ある種の問題はそれ以前の問題が解決されていないので、考察する段階にも至っていないものがあるということが分かる。

このなかで、特に [Binder 99] のパターンは単体テストを行う際に大いに参考になる。今までのテスト手法が答えてこなかった問題に対し、[Binder 99] の各パターンは明確な形で答えている。

[Binder 99] のパターンは多岐にわたっており、これを学んだ人のなかにはこれらのパターンで十分ではないかという声もあるかもしれない。しかし、[Binder 99] のパターンすらプログラマやテスト担当者にほとんど知られていないというのが現状である。そのため、これらのパターンを応用して、[Binder 99] が答えてはいないマルチスレッドにおけるテストの問題、分散環境におけるテストの問題に対するパターンというのはいまだ生まれてきていない。

### 1.3.3 行うべきこと

求めているパターンがないからといって、パターンという概念そのものに力がないわけではない。実はわずかに存在する上記のパターンの学習こそが自分自身のパターンを生み出す元となる。パターンとは単に使うものではない。パターンは自分自身もパターンを持っており、それにしたがって行動していることを知る機会である。パターンこそが従来テストで提唱されてきた概念では解決できなかった現場の問題を解決できる手段である。

とはいえ、それにはまずテストにおけるパターンとは何かを人々が知らなくてはならない。パターンを知り、共有することには計り知れない利益があるとはいえ、テストにおいてはいまだそれは実践されていない。

まずは、テストにおいてなぜパターンが普及しにくいのかを考察しなければならない。そのうえで、それらの問題を解決し、パターンを共有し、生み出していく方法を提案する必要がある。本研究ではそれらを順に考察していくことにする。

# 第2章 テストのパターンが発見しにくい理由

## 2.1 原因の分類

ソフトウェアテストにおいて技術者たちがパターン・ランゲージを共有するためには、まずはなぜテストにおいてパターンが普及していないのかを分析しなければならない。ここでは[Alexander+77]に倣い、原因を以下のように分類する。

- 2 ソフトウェア業界全体の問題
- 2 パターンそのものの問題
- 2 テストの定義における問題
- 2 技術者の態度に関する問題

## 2.2 ソフトウェア業界全体の問題

ここで取り上げる問題はパターンの問題というより業界におけるテストの位置付けの問題である。テスト自体がまともに行われてないためパターンを求めようという行動すら起こらない状況にあるということに注目する必要がある。

### 2.2.1 大企業における下請け構造の問題

ソフトウェアというものには明確な製造のプロセスが存在しない。また、それゆえにソフトウェア開発は知識集約型のプロセスであり、ほかの産業のように設計者と施行者を明確に分けることは出来ない。しかし、今日のソフトウェア産業では設計者とプログラマをまるで製造業で行うのごとく分けがちである。特にこれは大企業で顕著である。分析者、設計者、プログラマを分けないことは混沌をもたらす恐れがある(実装溺愛アンチパターン [Brown+98]) が、分担を組織内ではなく、組織そのものに適応することは大きな問題を起す。

プログラマとの信頼関係を築かず、不良の数、文書の数で彼らを管理しようとする。何が誤っているのかや、効率のよいテストとは何かについて考えようとせず、単に欠陥の収束を計測したところでプログラマを苦しめるだけである。管理によって計測された結果を見て考えなければならないのは、いかにしてその不具合を改善できるかであるが、その改善そのものが管理であった場合問題は永遠に収束しない。

本当に考えるべきは、プロセスのランゲージ、プログラムのランゲージ、テストのランゲージである。これらは設計者、プログラマーが完全に分離した縦割り構造からは決して生まれはしない。設計者、プログラマー、テスト担当者が共に悩むことなしにテストのパターン・ランゲージは生まれてこない。残念ながら、今のソフトウェア業界の構造はプログラマーと設計者が共に語り合うという関係ではないというのが事実である。

## 2.2.2 中小企業における品質軽視の問題

これは下請け気会社における問題とほぼ同様であるが、下請けではない独立したソフトウェア会社においてもみられる。

今日のソフトウェア開発においては、多くの場合、品質よりも製品の新規性、開発の迅速性が求められる。まず、市場においてデファクトスタンダードとなるかならないかは企業の命運を決めるものとなっているので、そのためには品質は二の次とならざるうえないという状況にある。まず行わなければならないのは、最新の開発手法、技術の学習であり、品質に関して学習をするひまは残されていない。正式なテスト教育のカリキュラムが組み立てられていることはまれである。[Jones 97a]によると品質と信頼性を第一に完了することを目指したプロジェクトはスケジュールが最もよく守られ、高い生産性及び市場で成功する可能性が高いことが証明されている。証明されているにもかかわらず、組織全体でデスマーチにはまり込んでいるのが現在の状況である。

## 2.2.3 管理と精神論による問題のすり替え

これはほかの業界でもいえることであるが、ある問題が発生したとき、その問題の原因を管理に求めることは多い。あるソフトウェアにおいて、とある欠陥が発生したことにより顧客に重大な被害が及んだため、この欠陥はいかにして防ぐべきだったのかという議論を行っているとする。このとき、多くの場合、問題は技術的なものから管理へさらには精神論へと発展する可能性が高い。管理や精神論に発展したところでその経験が次回いかされることは少ない。

本当に必要なのは、同じような欠陥を抱える恐れがありながらそれを防いだほかの組織について学ぶことである。成功した組織には失敗した組織にはないパターンを持っている。そして成功した組織におけるパターンとは一般に共通している。これを学ぶことなしには精神論の繰り返しである。

## 2.2.4 再利用可能なテストに関する問題

[Jones 97a]によると再利用可能なテスト計画及びテストケースに関して触れている文献が余りにも少ない。再利用に重きを置いているオブジェクト指向コミュニティーにおいてもテスト計画、テストケースの再利用が論じられることはほとんどない。

再利用可能なコード、アーキテクチャ、分析、設計については今やWWWの普及とパターンとオープンソースの動きにより学ぼうという意志さえあれば、それほど苦労なく学ぶことが出来る。しかし、テストに関してはテストケースが企業秘密とかわるためか積極的に公開していかうという動きは薄い。このため、テスト計画が明確に定まっていなような組織の場合、

思考錯誤でテスト計画、テストケースを決めていかなければならない。また、ほかの組織とも比較が出来ないので、どのような問題があるのかも分からない。

とはいえ、WWWの普及によりこの問題は随分解消されてきているようである。例えば、

<http://members.tripod.com/bazman/frame.html>

では、再利用可能なテスト計画の紹介を行っている。また、

<http://edhs1.gsfc.nasa.gov:8001/waisdata/pdr/html/cd4020102.html>

では、あるプロジェクトのテストスイートを完全な形で公開している。上記 Web サイトでの述べられているテスト計画、テストケースと自分たちのものを比較し、そこからパターンを見出すことも可能となる。

## 2.2.5 テスト技術の共有の問題

[Jones 97a]によると、調査対象の600社の企業のうち99%が単体テストを行っているが、このテストはプログラムを開発したプログラマによって行われるため、どのような技法が使われているのか、欠陥レベル、欠陥除去率はどれほどなのかについては全くデータが存在しない。また、テストの研究者にとって最も身近な話題であるパステストについても単体テストでどの技法がどのように使われているのかという情報はほとんど知られていない。「行っている」という表現もかなりあいまいである。正規のインスペクションと組み合わせるとどのテストケースを省くことが出来るのか、どのようなカバレッジを基本にしているのかという情報を提供しないかぎり、単に単体テストを徹底しているといったところで意味がない。

このような情報を公開し、共有できるかは個々のプログラマと組織の方針にかかわっている。プログラマがよりよいテストの方法を求めらば、組織を超えたコミュニティーでそれを求めることも可能なはずである。組織についても同様である。再利用可能なテストの問題でも取り上げたようにこのような動きは既に始まっている。最近、急速に台頭している方法論XP(eXtreme Programming)のコミュニティーにおいてもその動きは見る事が出来る。テストを重視する彼らの方法論はプログラマに対しテストについて深く考える機会を与えた。XPの広がりと共にこのような動きが広がっていくことが期待される。

## 2.3 パターンそのものの問題

ここではパターンそのものが持つ問題点に注目する。

### 2.3.1 テストの分野におけるパターンの理解の問題

現在の多くのテスト担当者、品質管理技術者はテストパターンといった場合、従来の「テストパターン」を連想する。この「テストパターン」とはテストケースの集合を意味する。この理由からか、本来のソフトウェアパターンの意味でテストのパターンを書いている Linda Lisingは"Testing Patterns"(<http://c2.com/cgi-bin/wiki?TestingPatterns>)において"Testing Patterns"と区別して表現している。

したがって、彼らとソフトウェアパターンの意味でテストのパターンについて話したいなら、彼らにソフトウェアパターンそのものを学んでもらう必要がある。

### 2.3.2 パターンの概念が普及している分野の問題

この問題はソフトウェアのパターンがオブジェクト指向コミュニティを中心に普及したという理由から生じている。事実、ソフトウェアのパターンを見るとオブジェクト指向に関するものが多数を占める。

ここで問題となるのが、テスト担当者及び品質管理技術者の位置である。彼らの多くはオブジェクト指向技術から距離を置いている。なぜなら、数あるオブジェクト指向方法論の多くはテスト及び品質管理についてはオブジェクト指向特有のものがあるとは明確な形では述べてこなかったためである。そのため、開発がオブジェクト指向型であってもテスト及び品質管理は従来通りにやっている組織が多い。

したがって、テスト担当者及び品質管理技術者は余りオブジェクト指向に注目しないため、結果的にオブジェクト指向コミュニティが中心となっているパターンの概念に触れることは極めて少なくなる。

### 2.3.3 既存の概念に対する問題

テストにおいてはわざわざパターンなどというものを共有しなくても、方法論、テスト手法、標準、マニュアル、規則といった既存の概念があれば十分であるという意見がある。この考え方には誤解がある。パターンはこれらを否定するものではないからである。むしろ、補完するものである。テスト手法について考えてみれば、実際にある局面でテストを行う場合、そのままあるテスト手法を適応するという事は少ない。例えば、メソッドのテストを行う場合、制御フローテスト、ドメインテスト、カバレッジの知識を組み合わせる。これらをどのように組み合わせるかは局面によって異なるので一概に規則やマニュアルに出来るものではないし、ましてやテスト手法といえるものではない。この組合せの知識こそが私たちが実際の場面で用いて来たものであり、パターンといえるものである。

したがって、既存の概念に対してパターンが優位であるという議論は意味がなく、双方は補完し合う関係にあるという認識が必要である。

## 2.4 テストの定義における問題

ここではテストそのものの定義に対する誤解があるため、テストは簡単なものである、テストは必要ないなどという誤った意見があるということを述べる。この考え方は通常、テスト担当者、品質管理技術者にはみられないがプログラマや設計者にみられる場合がある。このような考え方が主流を占める限りテストについて真剣な議論はできない。テストのパターン・ランゲージを共有しようとする人々にとっては極めて深刻な事態であることに注意する必要がある。この部分を誤って理解している人々はランゲージを共有しようとする気にすらならない。

## 2.4.1 テストの定義

多くのプログラマがテストの定義を「ソフトウェアが動くことを示すこと」と考えている。この考え方が危険なのは、テストは要求仕様、機能仕様といった仕様が正しく実装されているかどうかを検査することであるという論理に帰結しがちになり、その結果、テストに関してそれ以上考えることを止めてしまうためである。この考え方が誤りであることを示す。

[Myers 79] が述べる定義は以下のとおりである。

テストは、エラーを見つけるつもりでプログラムを実行する過程である。

更に [Myers 79] は以下のようなテストの原則を挙げている。

- 2 テストケースの必須条件は、予想される出力又は結果を定義しておくことである。
- 2 プログラマは自分自身のプログラムをテストしてはならない。
- 2 プログラム開発グループは、自分たちのプログラムをテストしてはいけない。
- 2 それぞれのテストの結果を完全に検査せよ。
- 2 テスト・ケースは、正しい予想ができる入力ばかりではなく、誤った予想しない場合も考えて書かなければならない。
- 2 プログラムを調べるのに、それが意図されたように動くかどうかをみただけでは、半ば成功したに過ぎない。残りの半分は、意図されなかった動きをするかどうかを調べることである。
- 2 プログラムが本当の使い捨てのものでないかぎり、そのテスト・ケースも使い捨てにしてはならない。
- 2 エラーは見つからないだろうという仮定のもとにてテストの計画を立ててはいけない。
- 2 プログラムのある部分でエラーがまだ存在している確率は、既にその部分で見つかったエラーの数に比例する。

[Myers 79] はこれらを原則として述べているが、むしろパターンというべきである。79年当時は原則であったのかもしれないが、現在の開発環境では状況によっては当てはまらない場合もある。[DeLano+96] はこれらの一部をパターンとして記述している（「問題領域」「エンドユーザーの視点」「異常なタイミング」「引っかいてにおいをかぐ」「奇妙な振る舞い」「キラーテスト」「問題の文書化」の各パターン）。

テストの定義として、最後に重要なのは [Binder 99] が述べている、「テストは分析し、設計し、評価するプロセスである」というものである。この定義はテストは決してつまらないものではなく、ソフトウェア開発そのもとと同様、非常に創造的なプロセスであること示している。よいテストケースを作るのはよいソフトウェアを作る以上に難しい [Myers 79]。[Myers 79] と [Beizer 90] の定義からもテストは単に実行するだけのつまらないものではないことがわかる。テストの実行はテスト全体から考えればほんの一部であり実際はほとんどが自動化可能なもの

である。テストにおいて考慮しなければならないのは、テストの分析方法、設計方法、評価方法であり、手動テストをもくもくとやることではない [Beizer 95]。

まとめると次のようになる。

- 2 テストは、エラーを見つけるつもりでプログラムを実行する過程である。
- 2 テストは分析し、設計し、評価するプロセスである。
- 2 テストは非常に創造的であり、知的に挑戦しがいのある仕事である [Myers 79]。

テストのパターン・ランゲージを共有しようとする人々はこれらの原則を心にとめておく必要がある。テストが難しいが創造的であるものという認識があつてこそ、設計においてデザインパターンが有効であったように、テストの設計においても何らかのパターンが有効であることがわかる。テストにおいてパターンを求めるためには、まずテストについて悩まなければならない。人々が同じ問題について悩み、ほかの人々が自分よりもよい解決策を持っていることが分かったときパターンの共有が始まる。

## 2.5 技術者の態度における問題

以下の技術者における問題は Brooks、Demarco、Jones、Weinberg、Yourdon が既に語り尽くしているものである。その中で特にテストに関係するものを挙げた。

### 2.5.1 テストの否定

この問題は技術者の態度における問題の中でも最悪のものである。テストの定義に対する誤った理解、言語万能論、銀の弾丸などの理由によりテストそのものを不要であると考える技術者も存在する。このような状況ではテストに関して真剣な議論を行うことは出来ない。

テストの定義、意義を理解しているが、時間がないので単体テストを省くという考えもあるかもしれない。[Binder 95b] ではその神話に対して、現実を示している。

我々は、自分がテストをしないことによって最終的に生じる損害について考えてみる必要がある。根拠のない楽観主義は何の役にも立たない。

### 2.5.2 テストへの誤解

テストの必要性は感じており、テストを行っていないわけではないが、テストの定義を誤って理解しているために、子供のテスト、突っつきテスト、モンキーテストなどをテストであると思っている技術者も存在する。また、明らかに重複のあるテストケースを作成し、もくもくと手動テストを行わせる技術者も存在する。

彼らも正しいテストの定義を学ばなければならない。そして、正しいテストのパターン・ランゲージに従う必要がある。

### 2.5.3 テストレイト

通常、テストは設計、コーディング、統合などが終了してから行うものとして認識されている。コーディングに限っていえば、確かにテストをすべてコーディングの前段階に行うことはほぼ不可能である。だからといって、テストを先延ばしにして、最終的に未テストの部分を残しながらシステムに組み込むようなことがあってはならない。

プログラマは一部のテストがコーディング以前にも行えるということに注目しなければならない。XPのプラクティスの1つがその実例である。XPには12のプラクティスが存在するがそのうちのテストのプラクティスは従来のテストの考え方とはかなり異なっている。XPの単体テストではコードを書く前にテストケースを作成し、テストコードを書く。このメソッドは何をしなければならないのか、どのテストケースであれば欠陥が出やすいのかを予想し、テストを事前に書く。そして自動化されたテストツールによりプログラマはそれを徹底する。この方法はブラックボックス的であり、コードのカバレッジなどは考慮していないが、そのテストに対する新たな視点はこのようなことが現実にも可能であるという点で注目できる。

単体テストを先延ばしにする傾向が高いプログラマはXPのテスト方法を見習ってみる必要がある。

### 2.5.4 銀の弾丸

[Brooks 75]を始めとした様々な文献が述べているように、すべての問題を一度に解決するような銀の弾丸は存在しない。インスペクションだけでも十分ではないし、正当性証明を行ってもテストは必要である。単体テストを徹底しても統合テスト以降のテストは必要である。

## 2.6 問題のまとめ

パターンの普及を妨げる問題は以上のようなものである。まとめると、プログラマにおいては様々な要因から正しいテストが行われることが少なくなり、その結果、テスト技術の探求を行わないという現象が起きている。また、テスト担当者及び品質管理技術者においては、情報公開の問題から企業を超えたテスト技術の共有が困難になっている。パターン自身の問題としてはテストの分野におけるアプローチが弱いことと技術者たちがパターンを求める段階に至っていないことが挙げられる。これらの結果、テストの分野においてパターンの共有、創出が困難になっているということである。

# 第3章 テストにおけるパターン発見のための 解決策

## 3.1 解決策の分類

これまで、ソフトウェアテストにおいてパターンの普及を阻む問題の分析を行ってきた。最終的にはプログラマ側の問題と、テスト担当者及び品質管理技術者の問題に分かれる。

実はこの二つの問題は大きく形式が異なる。プログラマの問題はそもそも正しいテストが行われていることが少ないのでパターンの土台となるもの自体が少ないということである。この場合、彼らがパターンの概念そのものを知れば、パターンを創出できるという単純な帰結にはならない。正しいテストをわざわざ行ってきたプログラマ、彼らはパターンとなる土台を持っている。このように考えると、テストのパターンを持ちうるプログラマと持ちえないプログラマに分けて解決策を考えなければならない。

また、テスト担当者及び品質管理技術者においてはテストに対する誤解は一般には存在しない。そして重要なことに彼らは既にパターン・ランゲージを胸に秘めている。ただ、パターンの形式を知らず、パターンとして表現する術を知らないだけである。そして、パターンを共有することによる利点を理解していないということもある。彼らに必要なのはパターンそのものの精神を知り、パターン共有に踏み出すことである。このためには彼らにパターンとは何かを理解してもらう必要がある。

したがって、解決策を次のように分類する。

- 2 正しいテストを行っていないプログラマに対して
- 2 テストを行っているプログラマに対して
- 2 テスト担当者と品質管理技術者に対して

## 3.2 正しいテストを行っていないプログラマに対して

ここでは、本論文でいう正しいテストを行っていない、若しくは行ってこなかったプログラマに対して、テストのパターンを創出するにはどのようにしたら良いかを述べる。

### 3.2.1 真のテストへの理解

ソフトウェアに限らずあらゆる分野でテストを行うのは、人間があらゆる意味で誤りを犯しやすいものであり、人間の予想能力に限界があるためである。私たちはあるものを作ろうとし

たときには大抵楽観的な思考のもとに始める。楽観的思考の元では誤りがまぎれ込んでも気付かないことが多い。だが、その作り上げたものに対し、真の意味の質、[Alexander 79]のいう「無名の質」を求めるならば、たとえ破壊的な過程を伴うとしても、その誤りは取り除かなければならない。

他人の作ったソフトウェアに対しては不満をいいながら、自分が同じような過ちを犯してそれを見つける手段を持ちながら行使しないのは、その人間が単なる恥知らずであるか、ソフトウェアの持つ無名の質に近づこうとしていないことを意味する。

結局、プログラマが自分の作り上げるソフトウェアに対し、真の質、すなわち無名の質を求めているかがテストを行うか、行わないかの分かれ目となる。人間の能力の限界から、欠陥を完全に防ぐことは不可能であり、そのためには欠陥を見つけるつもりでテストを行わなければならない [Myers 79]。

### 3.2.2 テストファースト

たとえば、本当の意味でのテストを行うようにプログラマが決意したとしても、[Myers 79]が提示した原則が前に立ちふさがる。すなわち、

- 2 プログラマは自分自身のプログラムをテストしてはならない。
- 2 プログラム開発グループは、自分たちのプログラムをテストしてはいけない。

ということである。しかし、この原則が書かれている前後の文を注意深く観察すると、これはドグマではなく、パターンであることが分かる。すなわち、これら二つは特定の状況と前提が存在する。これらをパターンの形で書き換えてみる。

テストするのは別の人

状況

あるプログラマが自分の担当するプログラムを完成させ、テストをしようと考えている。

問題

だれがそのプログラムをテストすべきか。

フォース

- 2 プログラマがプログラムを設計しコーディングする間は建設的である。
- 2 テストは破壊的である。
- 2 同じ人間が同じ対象に対して建設的であると同時に破壊的であるのは困難である。
- 2 自分自身でテストを書くとプログラムを書いたときと同じ前提条件をテストにも持ち込みやすい。
- 2 自分自身が書いたものに対してテストの結果を見るときは「正しいはずである」と思いがちであり、結果を自分の都合に良いように解釈する。

## 解決策

プログラマは自分自身のプログラムをテストしてはならない。ほかの人にさせなさい。

ここでこのパターンを観察してみると、プログラマはプログラムを一人で作り、テストはプログラムが完成してから作るものであるという前提が状況で述べられていることが分かる。しかし、異なった状況の場合、例えば、二人で一緒にプログラムを作る場合、テストをプログラムが完成させる前に書く場合などでは、解決策は異なるを考えるのが妥当である。このように考えると、プログラムが自身がテストを行っても良い状況というのが存在することも分かる。それは次のようなパターンになる。

## プログラムの前にテスト

### 状況

プログラムを書く前に大まかな設計が定まっている。

### 問題

だれがいつ、そのプログラムをテストすべきか。

### フォース

- 2 既に取り上げたプログラムに対しては、テストが主観的になりやすい。
- 2 事前に出力結果を記述すれば、結果を都合良く解釈するということは低くなる。
- 2 事前にテストを考えられないようなプログラムはそもそも何をやりたいのか分かっていないので、事後のテストは更に困難である。

## 解決策

プログラムの中身を知る前に、客観的にインタフェースを観察できる状態であれば、プログラマがテストケースを考えても良い。プログラムを作成する前に、プログラム設計書、モジュール設計書、ルーチン設計書などプログラムを書くのに必要とする文書からテストスイートを構築しなさい。これらのテストスイートはプログラムの構造に基づいて作ったものではないので、当然ブラックボックス的になる。

ただ、これでも、プログラムを書くときと同様な思い込みを排除できるわけではないことに注意せよ。テストするのは別の人パターンはこのパターンを補完するものである。

ここで、重要なのは [Myers 79] で述べていることは原則というよりパターンであり、選択の余地を含むものであるということである。パターンの場合、状況が異なれば異なった解決策を取っても構わない。

開発グループについても同様のパターンを考えることが可能であり、その場合開発グループ自身の事前のテストという解決策もありえる。

ここで、テストについてプログラマが深く考えるためには事前のテストのほうが望ましい。なぜならテストを後に考えることは結局テストレイトの考え方に通じ、テストはいつでもいいこととなりがちだからである。テストレイトでもテストを行うという固い決意があれば、色々なパターンが眠っていることが分かる。しかし、テストファーストのほうがプログラムについて悩む前にテストについて悩むことが可能であり、より多くのパターンを発掘することが可能になる。

### 3.3 テストを行っているプログラマに対して

実のところ、パターンはこれまで何度も述べてきたようにテストを真剣に求める過程で既にその本人自身が作り出している。ただ、それがパターンであることを気付かないだけである。それはそれ自身を作り上げようとして生まれたものではなく、正しいソフトウェア、真の質を持つソフトウェアを作り上げようという強い情熱の元、間接的につくられたにすぎない。これまで私たちが軽視してきたとも簡単な行動、構造にこそ答えがある [Alexander 79]。

パターンとは何かを知り、ほかのパターンを読み、パターンの精神を知り、自分自身を顧みたと、彼は実のところパターンを持っていたこと、捜し求めているものは既に自分の元にあったということを知る。以下で述べる事柄はその門に至るための指標である。

#### 3.3.1 パターンのカタログ

テストを真剣に行う人々が読むべきものはやはりテストのパターンである。

テストプロセスのパターンとしては [DeLano+96] が存在する。読者によっては [DeLano+96] が述べていることは当たり前のことのように見えるしれない。[DeLano+96] が余りにも当たり前に見えるのは解決策のみに注目しているからである。後に述べるがパターンにおいて重要なのは状況、問題、フォース、解決策、これらが密接に係わり合いを持っていることである。平凡に見える解決策ですら、実際は様々な力を考慮しそれらを解消するために生み出されたものである。しかし、当たり前という解決策に至る力の作用を考慮することこそがパターンにおいて重要なのであり、その力を正しく記述できてこそパターンの意味がある。従来の問題と解決策の対のみの処方箋に不足していたのは解決策がその状況においてどのような力を解消しているのかということであった。実際はその状況に働く力の作用が異なれば、異なる解決策となるため、問題、解決策の対はそのままの形では利用できないことが多かったのである。この解決策に作用する力、フォースを見つけ出すプロセスは非常に困難なものである。[Alexander 79] は次のように述べている。

このような不変領域の発見はとてつもなく困難な仕事であり、理論物理学におけるいかなる発見に比べても引けを取らない。

とはいえ、初めはこれほどまで考えてパターンを読む必要はない。システムテスターの経験がある人間ならば [DeLano+96] に多くの解決策を見出せる。

プログラマで単体テスト、結合テストを行っている場合、[Binder 99] が役立つ。特にオブジェクト指向テストにおけるテスト設計に関し強力な解決策を与えている。既に従来 of テスト手法の知識を持ち、それをオブジェクト指向に適応する際に様々な問題があるということを認識しているプログラマであれば、[Binder 99] は新しい解決策であると同時に今まで自分が挑戦してきたことを思い起こしてくれる。

#### 3.3.2 パターンの書き方

[Binder 99]、[DeLano+96] を読み進めていき、[Gamma+95] などほかの分野のパターンを読んでいくにしたがって、自分の周りにもパターンとなりうる解決策が数多く存在していること

が分かってくる。それはテストにおいても同様なのであって、ただ真剣に振り返ってこなかっただけである。

ここで実際にパターンの形で記述してみようという欲求が生じたい場合、取るべき解決策はいくつかある。一番よいのは PLoP (Pattern Languages of Programs) のようなパターンを研究し、新しいパターンを創出しているグループに参加し、パターンの書き方を学ぶことである。

もう1つの選択肢はパターンの書き方に関するパターン、例えば [Meszaros+96] から学ぶことである。[Meszaros+96] ではどのようにしたらよいパターンを書くことができるかについてパターン・ランゲージの形式で述べている。[Meszaros+96] の Mandatory Elements Present Pattern ではパターンを記述するのに基本となるのは、パターンの名前、状況、問題、フォース、解決策であるとしている。状況は解決すべき問題が生じる状況を表す。問題は解決されるべき特定の問題であり、状況とは独立したものであることが望ましいとされる。フォースは解決策を取るに至った理由、考慮した力を表す。

これだけでもとりあえずパターンを書いてみることはできる。しかし、テンプレートに沿ってパターンを書けることは本質ではない。これだけでは自分の書いたパターンがよいパターンかどうか不明である。

### 3.3.3 パターンの概念

そこで、パターンそのものの概念、精神を知る必要が出てくる。パターンの真の意味を知ることなしには表層的にパターンを利用し、記述することしかできない。これは単なるパターンカタログを何度読んだところで得られるものではない。パターンの核心に触れたものは Alexander の著書である。[Alexander 79] はパターンの核心に触れ、[Alexander+77] はその精神を宿した 253 のパターンについて述べ、[Alexander+77] はそれらのパターンを実際に適応した例を述べている。[Alexander 79] が述べているパターンの概念はほぼすべてがソフトウェアのパターンにも適応できるものであり、パターンの真の意味を知るためにも一度は読んでおく必要がある。更にいえば、[Alexander 79] はパターンを使用し、記述する人間の能力が上げれば上がる程、そこで述べられている意味の深さが分かってくる。[Alexander 79] が述べているのは以下のようなものである。

- 2 真剣に取り組む
- 2 独立した「もの」ではない
- 2 個人によって異なる
- 2 進化する
- 2 現実に働く力を認める
- 2 知性ではなく感性からである
- 2 力を解消する
- 2 つくるものではない

<sup>2</sup> 平凡なことである

[Alexander+77]の各パターンを読むと更に重要な変化が起きる。[Alexander+77]は題名のとおり建築に関するパターン・ランゲージであり、これを読むとあらゆる建造物に対し、パターンの視点を持つようになる。

パターンによる観察を続けていくとソフトウェアの構造、プロセスもこのようなパターンにしたがっていることが分かるようになる。実は多くの事象はパターンそのものなのだが、パターンの観点でものを見ることをしていないので、パターンであることが分からないのである。

この状態に至れば、パターンを自分自身で記述することも可能となる。

### 3.3.4 心の余裕

パターンは日々の仕事に追われているときには考えることはできない。心静かに自分自身を振り返るときのみパターンは意識される。ゆっくりと心静かに、自分自身のパターンについて考える時間と心の余裕を持つべきである。

グループでパターンマイニングをする場合も同様である。反省会となじりあいではなく、これまで行ってきたよい解決策について考える機会を持つことが必要である。

### 3.3.5 おのずから生まれるもの

まとめると、パターンが存在しないと思われたのは外部的な要因ではなく、実は自分自身が意識していなかっただけなのである。真剣にテストを行う過程でそれは既に存在しているものであり、パターンの記述方法、精神を学び、常にパターンを意識すればおのずと生まれ、共有できるものなのである。得られる解決策は本人に取っては取るに足らないものであるかもしれないが、ほかの人々には大きな力となるものかもしれない。自分がパターンを記述し、ほかの人々と共有することによってそのパターンは更に進化し、そのほかのよいパターンも生まれてくるという流れになる。

## 3.4 テスト担当者と品質管理技術者に対して

### 3.4.1 2つの解決策

テストの正しい定義を知っているテスト担当者と品質管理技術者に対する解決策は正しいテストを既に行っているプログラマとほぼ同様である。

更に彼ら独自の問題として、パターンの共有が自分の所属する組織に不利益をもたらさないであろうかというものがある。高度なテストのノウハウはその組織を左右するほどのものであるので、容易に公開することは一般にためらわれると考えられているが、実は全く逆である。[DeLano+96]でも分かるとおり、パターンとして公開してこそ、その組織がよりよいものとなる。ある組織では当たり前と思われていることはほかの組織では適応できないかもしれない。それはなぜかというのは広く世に公開しない限り決して分からない。また、パターンは新規の

技術でも何でもなく、むしろほかの組織でもやっていると予想されること記述したものであるから、特許のように使用制限があるわけでもない。

「パターンを共有するメリットは分かったが、我々には社内標準、ISO 標準、マニュアル、規則などがそろっており別にパターンの必要性は感じていない」という意見があるかもしれない。どんなに標準、手順、マニュアルなどがあっても最終的に判断をするのは自分自身である。この自分自身、あるいは組織自身の判断又は判断の対象とすべき解決策こそがパターンなのである。この領域は一般に個人や組織で異なるものとされているが、実はこの領域こそ最も困難なものであり、ほかの人々の解決策が参考になる部分なのである。それゆえにこの領域をパターンとして共有することは個人、ひいては組織全体がよいよいものになるために必要なことであるといえる。

まずは [DeLano+96] のように組織でパターンマイニングを行い、それを公開してみることである。与えることによってのみ得られるということを忘れてはならない。

# 第4章 非公式の単体テストのためのパターンランゲージ

## 4.1 発見のプロセス

これまで述べてきた方法によってテストの分野においてもだれもがパターンを記述できることを示す。ここで示すパターンは提案でも理想でもなく、ただ自分自身の行動を顧みただけに過ぎない。

テストにおいてパターンが生まれ難かったのは、極度の無関心と関心の2つのみが存在してきたからである。極度の無関心はテストにおける質を求めないためパターンを生み出すことが出来ず、極度の関心はすべての問題を「しなければならない」と縛ることによって自然なパターンではなく、規則を作り出していた。

必要なのはテストを無視することでも極度に重視することでもない。ソフトウェアの質というものを考えたときに、自分は何が出来るのかという問いに対し、「テスト」という一手段があるに過ぎない。質のためには何が出来るのかを真剣に考えている中で、おのずからテストはある形を取ってくる。それは目的、状況によって異なるが、共有できるパターンの形を取る。

だれでも独自のやり方というものがある。それはここで示すパターンのような形で書くことが可能である。そのパターンは本人にとっては余りにも当たり前なので改めて明示知として書き表すほどのものではないと思っている。しかし、その当たり前に思えるものにこそ真実が含まれており、ほかの人々のために役立つパターンなのである。今までに示したような方法でパターンを生み出すことはだれにでも可能なことである。それは自分が支配権を握り、自分が創造的刺激を注ぎ、設計を制御するイメージを自分が提供しなければならないようなものではなく [Alexander 79]、ただ静かに自分自身を顧みることによって導かれるものに過ぎない。ここで述べるパターン・ランゲージもそのようにして得られたものであり、決してほかの人々に「～しなければならない」と押しつけるようなものではない。生じる力にただ対処したに過ぎない。

多くの人々がテストのパターンを書き始め、流れとなることによってテストにおけるパターンは進化し、無名の質を備えていくことになる。

## 4.2 パターンランゲージの概要

本パターンランゲージはプログラマが正式な単体テストを行う前に用いる非公式なテストについて述べたものである。プロ、アマを問わず多くのプログラマはたとえ正式なテストの知識を持っていなくても、コーディングを行う際に非公式なテストを行っている。この非公式なテストは人によって方法は様々であるが、各人が用いている方法は繰り返されている。これはよく観察するとパターンの形式を取りうる事が分かる。

本パターンランゲージは筆者が雑誌の投稿プログラムを作成する際に用いているものである。この雑誌の投稿プログラムは毎回クイズが出され、それを解くために用いるものである。このクイズでは例題はコンピュータを使わなくても解くことが出来るが、本題はコンピュータを使わないと時間的にも空間的にもまず解くことが出来ないという性質を持っている。

本パターンランゲージは実際の業務から得られたものではなく、趣味のレベルのものであるが、正しいクイズの解を得るためには正確なテストを行わなければならないという条件があるので、その厳密性は業務ソフトウェアの開発に必ずしも劣るものではない。

## パターンランゲージの構成

本パターンランゲージは以下のような4つのテスト実施のパターンと1つのアンチパターンからなる。

- 2 クイックハック (アンチパターン)
- 2 サンドイッチアプローチ
- 2 石橋叩き
- 2 子供のテスト
- 2 3度目の自動化

各パターンはプログラムの設計とコーディングのプロセスで用いる。各パターンの適用を以下に示す。

本パターンの対象とするプログラムはクイズという形式を取っており問題が常に提示されている。したがって入力と出力の仕様は定まっており、プログラムがどのように振る舞うべきかの仕様はある程度決定している。また、このクイズは数学やパズルの本に掲載されていてアルゴリズムが知られている場合がある。その場合、まず考えるのは既に同様の問題を解決したプログラム若しくはアルゴリズムを探し、それに改良を加え、自分のプログラムとすることであろう。このプロセスはクイズに限らず、業務でも行われることが多い。この際に多くの人々はクイックハックアンチパターンにはまり込むことが多い。クイックハックアンチパターンではこのプロセスで生じる誤りに対して改善策を提案する。

採用するアルゴリズムが決定し、設計の仕様も固まり、コーディングに取りかかるところである。メソッドを完全に作成してから単体テストを行うという方法はプログラマにとって不安を伴うことが多い。何らかに手段でコーディング途中で動作確認以上の非公式なテストを行いたい。この際にサンドイッチアプローチパターンが役立つ。サンドイッチアプローチパターンはメソッドの入力ドメインをチェックするメソッドの前半部と、「まだこのメソッドは出来ていない」という例外を発する後半部をはじめに作るによりコーディング途中のテストを可能とし、なおかつ作成中のメソッドが正式版のソフトウェアに組み込まれることを回避する。

石橋叩きパターンはサンドイッチアプローチパターンとペアで用いることが多い。石橋叩きパターンではコードを自分自身が安心できる範囲で書くたびにそのコードを通過するテストケースを追加していく。それは1行でも数行でもよい。このパターンによって正式なテストを行う前に非公式にテストスイートはステートメントカバレッジを満たす。

下請けのメソッドについては非公式、公式両方のテストを終え、ついにメインのクイズを解くメソッドを実装している。このメソッドの非公式なテストはサンドイッチアプローチパターンと石橋叩きパターンを組み合わせることによって出来るが、公式なテストについては問題が残っている。すなわちクイズは探索問題であり非常に小さな入力に対しては手作業で出力を求めることが出来るが、一般の値若しくは大きな値については手作業で出力を求めることは困難である。そしてクイズが要求しているのは一般に大きな値の入力である。この際に子供のテストパターンが役立つ。子供のテストパターンはテスト対象のメソッドが採用するアルゴリズムがよく知られており数学的に証明されているときに用いられる。

最後に非公式なものであれテストの自動化は必須である。テストが自動化できないとテストを行うのは苦痛となる。3度目の自動化パターンは非公式なテストにおいて、いつ自動化を行うべきかを指摘する。

## 共通の状況

はじめに述べたように、本パターンランゲージは雑誌で出されるクイズをプログラムを作成することによって解くときに用いているものである。このプログラムはかなり小規模なものであり、Java や Smalltalk などのような高級言語を用いれば数 100 行で書いてしまうものがほとんどである。また個人で作成しているものであり、明確な管理基準はほぼ皆無である。したがって、明確な設計プロセス、コーディングプロセス、テストプロセスが存在する大規模な組織とは全く状況は異なる。状況としてはテストプロセスが明確ではない数 10 人でのソフトウェアハウス若しくはフリーのプログラマの開発形態に近い。

このプログラムには正しい仕様が存在し、正しいテストを行わないと正しい答えを得られないという特徴がある。そのためたとえ趣味のレベルであっても同値分割、境界値分析、制御フローなど最低限度の正式なテスト手法の知識は必要である。本パターンランゲージにおける非公式なテストはこれらの正式なテスト手法の一部を利用する。

クイズに限らずアマチュアプログラマ若しくはテストプロセスが明確でないソフトウェアハウスがプログラムを作成するときにはほぼ同様の状況に置かれることになることが予想される。

## 共通のフォース

本パターンランゲージには共通の状況と同時に共通のフォースが存在する。これらはテストそのもののフォースかもしれない。

- 2 アルゴリズムの正当性が検証されただけでは十分ではない。
- 2 実装時にも欠陥は混入する。
- 2 アルゴリズムは正しく実装されなくてはならない。
- 2 作成したプログラムは何らかの方法を使って正当性を確かめなければならない。
- 2 テストはより誤りの少ない方向に進めなくてはならない。
- 2 テスト専用の第3者はおらず、プログラムをテストするのはプログラマ自身である。

- 2 プログラマがテストを行うとテストは後回しになりやすい。
- 2 欠陥収束管理をするほど複雑なプログラムではない。

## 4.3 アンチパターン：クイックハック

アンチパターンの名前：クイックハック

別名：ハック&コンパイル

頻出スケール：アプリケーション

再構想解の名前：リバースエンジニアリング

再構想解のタイプ：プロセス

根本原因：拙速、無精

対応不全の圧力：機能性の管理

挿話証拠：「ここをこう変えればとりあえず動くだろう。」

### 背景

このアンチパターンは対象とする問題の全体像を理解しない、小手先の理解をソフトウェア開発に用いる場合に生じるものである。

### 一般形式

欠陥修正又は機能追加のために自分、若しくは他人が過去に作ったプログラムの変更を行うという状況を考える。このプログラムには文書が一切存在せず、あるのはソースコードのみである。このような状況のとき、このアンチパターンは出現する。

プログラムに関する文書が存在しない場合、たとえ変更箇所が分かっているとしても、プログラムの全体像を理解出来ず、変更が全体に与える影響が不明確なので、対象箇所を適当に修正し、とりあえず動作を確認するという行動に走りがちである。このような行動は正式なテストを否定し、最終的に溶岩の流れ[Brown+ 98]を生むことになる。

### 病状と結果

- 2 ソフトウェアにはソースコード以外の文書が存在しない。ソースコードにコメントすら存在しないこともある。
- 2 全体的な理解が不可能なので場当たりの修正を行う。

- 2 仕様が理解出来ていないので正しいテストが出来ない。
- 2 テストはコンパイルが通り、とりあえず動くというだけになる。
- 2 修正が元々正しい理解に基づいたものではないので、修正後のコードは更に意味不明なものとなる。

## 典型的な原因

### 2 保守性の欠如

このパターンが発生する根本原因はソースコード以外の文書が存在しないことである。ソースコードだけではプログラムを多面的にとらえることが出来ない。クラス間の協調関係、満たさなければならない仕様、必要とするテストケース、これらをソースコードから理解するのは困難である。自分が将来そのプログラムを保守することを考えれば、文書不在は致命的なことが分かるのだが、目下の仕事に追われそれが出来ない。

### 2 理解の放棄

どうせ全体を理解しようと思っても不可能なのだから、適当に修正して動けばよいという心理は問題が複雑であればあるほど働く。

## 例外的なケース

自分専用のソフトウェアやフリーソフトウェアを自己責任で移植する場合などにはこのパターンは許される。しかし、今後のためにも全体像を理解したほうがよいのは明らかである。

## 再構想による解決

逆説的であるが、全体像を出来る限り把握しようと努力することである。現在は、ソースコードをリバースエンジニアリングすることによって、クラスの協調関係を図示するもの、制御フローグラフを生成するもの、メソッドの仕様書を生成するものなど様々なリバースエンジニアリングツールが存在する。完全とはいえませんが、商用では優れたものが数多く存在する。また、フリーでも十分実用になるものは多い。ソースコードとにらめっこして疲労し、結局適当な修正を行うより、まずこれらのツールを使って全体像を把握するほうが近道である。

## そのほかの視点やスケールへの適応性

このパターンを根本的に防ぐには上流過程で文書の作成を徹底させることである。だが、この方法は文書とソースコードの乖離という新たな問題を引き起こす。一番よいのはすべての文書とソースコードを効率的に連携させることであるが、これを完全に解決するツールは現在のところ存在しない。

## 例

ソースコードだけでは理解が困難な大きさのプログラム、例えば 10 個以上のクラスからなるモジュールを扱う際にはソースコードを読むと同時に UML ツールのリバースエンジニアリング機能を用い、クラス間の協調関係を理解する。これによってモジュールの全体像の理解が速まる。また、これによって協調関係がパターンをなしていることも分かるので、製作者の意図もより理解しやすくなる。これはソースコードのみから理解するのはなかなか困難なことである。

## 4.4 パターン名 サンドイッチアプローチ

### 状況

メソッドの設計も終わり、実装すべき段階に来ている。メソッドの設計仕様からテスト計画も出来上がっている。

### 問題

メソッドが完成してからテストを行うのではなく、徐々に実装を行いながら非公式なテストを行いたい。

### フォース

- 2 テストをするためにはコンパイル出来る形である必要がある。
- 2 入力ドメインを外れた不正な入力だけは実装の初期段階で弾いておきたい。
- 2 未完成のメソッドが本番プログラムにリンクされ、動作してしまうのは問題がある。
- 2 メソッドが巨大な場合、途中のテストを行わずにメソッド完成後のテストをするのは不安である。

### 解決策

メソッドの入り口と出口を最初に作成する。入り口では入力ドメインチェックのみを行うステートメントを記述する。

出口では、「まだこのメソッドは出来ていない」という例外を発生して終了するようにする。これによって、仮にこの未完成のメソッドが他のメソッドから呼び出されても、例外が発生しその時点でプログラムは停止することになり、未完成のメソッドを使ってプログラムがそのまま実行を続けてしまうことを防止出来る。メソッドが完成したら、この例外発生ステートメントをコメントアウトする。

入り口と出口をまず用意することからこのパターンをサンドイッチアプローチと呼ぶ。

このパターンで行うテストは非公式なものであり、メソッドの完成後、公式なテストを行う必要があることに注意すること。

実装の順番は次のようになる。

1. 終了文の直前に、「まだこのメソッドは出来ていない」という例外を発するステートメントを記述する。
2. この時点でコンパイルが可能となるので、メソッドの呼び出しテストを行い、入力の種類に関わらず、「まだこのメソッドは出来ていない」という例外が送出されることを確認する。
3. メソッドの先頭に入力ドメインチェックを行うステートメントを記述する。
4. 正常な入力は受け入れているか、不正な入力は全て弾いているか（無視する、例外を発するなど）をCategory-Partitionパターン [Binder 99] に従ってテストする。
5. メソッドの本体を記述する。石橋叩きパターン [Oota 2000] に従っても良い。石橋叩きパターンに従った場合、ステートメントを記述するたびに非公式のテストを行うことになり、そのテストは必ず「まだこのメソッドは出来ていない」という例外で終了する。
6. 本体を記述し、戻り値も仕様に従って変更し、メソッドが完成したら、公式のテストを行う。

## 実装

Javaにおいて、メソッドに複数の出口がある場合、メソッドの本体をtry節で囲み、finally節に「まだこのメソッドは出来ていない」という例外を記述すると未完成のメソッドで常に「まだこのメソッドは出来ていない」という例外を送出することが可能となる。このイディオムはメソッドの保守の際にも有効である。

## 例

付録 A.1 を参照。

## 関連するパターン

本パターンではメソッドの実装途中で非公式のテストを行うため、石橋叩きパターン [Oota2000] と組み合わせて使用される場合が多い。

## 4.5 パターン名 石橋叩き

### 状況

メソッドの設計も終わり、実装すべき段階に来ているが、今回のメソッドは新しく使うAPIが多い。

## 問題

メソッドが完成してからテストを行うのではなく、徐々に実装を行いながら非公式なテストを行いたい。

## フォーース

- 2 現在の Windows や Java の API は往々にして複雑なものが多く、正しく動作しているのかどうかの確信を得るのが難しい。
- 2 メソッドが巨大な場合、途中のテストを行わずにメソッド完成後のテストをするのは不安である。
- 2 メソッドが完成するまでに正しいと確証できる場所は確証しておきたい。

## 解決策

メソッドを 1 単位 (これは自分の自信の持てる程度によって異なる) 書くたびにそのパスを通るテストケースを作成、実行し、実際に正しい操作が行われ、値が返されているのかを確認する。このパターンを用いると正式な単体テストを行う前に 100 % ステートメントカバレッジが非公式に終了している。

サンドイッチアプローチパターン [Oota 2000] と同様に、このパターンで行うテストは非公式なものであり、メソッドの完成後、公式なテストを行う必要があることに注意すること。

## 実装

このパターンにしたがって非公式にテストを行う場合、何度もテストを行うことになるので、自動化ツールは必須である。

## 論理的根拠

[McConnell 93] ではメソッドに対し、確信を得るまでコンパイルはしてはならないとしており、プログラムが動作すると確信できる前にコンパイルするのは、ハッカー心理の兆候であると述べている。しかし、本パターンでは適当にハックしてから動くかどうかではなく、メソッドの設計はすでに終わっており、コンパイルのたびにテストケースを用意している点が異なっている。テストケースが用意可能であるということは、設計に対する確信が存在しているということである。すなわち、設計が存在しないのではなく、実装時の不安を低減させるために、本パターンを用いるのである。

## 例

付録 A.2 を参照。

## 関連するパターン

本パターンが成立するためにはメソッドの実装途中でコンパイル可能である必要があり、そのためサンドイッチアプローチパターン [Oota 2000] と組み合わせる必要がある。

## 4.6 パターン名 子供のテスト (別名 とりあえず実行)

### 状況

あるプログラム (モジュール、メソッド) に関して使用するアルゴリズムの正当性は検証されている。しかし、そのアルゴリズムを実装したプログラム (モジュール、メソッド) をテストするために使用する入力と出力の組を手作業で求めるのは時間的、精神的に現実的ではない。

### 問題

上記の状況のように入力に対する出力を対象プログラム以外で求めるのが現実的ではない場合どのようにテストを行ったら良いか。

### フォース

- <sup>2</sup> 手作業で入力に対する出力を求めるのはプログラムを作成するよりも難しく、誤りを生じる場合がある。

### 解決策

実際に入力列のみからなるテストスイートを作成、実行して、出力結果を分析してそれが正しいことを証明する。

### 適応可能性

このパターンが有効なのは探索問題などで利用するアルゴリズム自体はそれほど複雑ではなく、出力が決定すれば、その正当性をアルゴリズムにそって検証することが容易な場合である。そして、このパターンを用いて良いのは、手作業で出力を求めるのが現実的に不可能な場合、若しくは求めることはできるが、その求めた出力が誤りを含む可能性が高い場合のみである。

Trusted System Oracle パターン [Binder 99] などオラクルを使って、手作業以外の方法を使って入力に対する出力を求められる場合、若しくは少し苦労すれば手作業で簡単に誤りな

く出力が求められる場合にもこのパターンを用いるのは単なる手抜きである。

## 論理的根拠

このばかりしい程単純なパターンについて Boris Beizer は以下のように述べている [Beizer 90]。

実際にテストを実行して、何が出力されるか見るのが最も簡単である。実にそのとおりだし、別に矛盾したことをいっているわけでもない。要は自己訓練の問題である。出力結果が目前にある場合、特に内部変数の中間結果まで明らかな場合は、出力を予想してそれが正しいことを検証するよりも、実際の出力結果を分析して、それが正しいことを証明するほうが簡単である。

子供のテストをオラクルとして使用する場合、予測の手段として利用するのか、あるいは、純粹に子供のテストなのかを明確に区別できないという問題がある。自制心が強く、自己訓練ができており、また、予想結果を検証するための分析方法がドキュメント化されている場合は、子供のテストを利用しても問題はない。

## 例

付録 A.3 を参照。

## 関連するパターン

Recursive Function Test パターン [Binder 99] は再帰呼出しを含むメソッドに関して、予想される欠陥とそれを検出するのにどのようなテストケースが必要であるかについて述べている。

## 4.7 パターン名 3度目の自動化

### 状況

テストの際に手作業でテストデータを何度も入力している。

### 問題

自動化したほうがよいことは分かっているが、どのタイミングで自動化したらよいのかわからない。

### フォース

<sup>2</sup> 手動でも最初と次ぐらいまでは気を付けてデータを入力する。

- 2 同じデータを何度も手動で入力していると疲れて間違いやすくなる。
- 2 同じ操作を何度もすることに耐えきれぬ回数は人により異なる。
- 2 手動でのテストはそのうち面倒になり行わなくなる。
- 2 手動での入力は使い捨てになりやすい。
- 2 自動化の知識は持っている。

## 解決策

自分が全く同じ操作をして耐えられる回数を超えて実行することが分かれば、自動化できないかどうか考える。テストの規模にもよるが最低でも5回以上の実行で十分自動化の元をとることが出来る。そして、同じ操作に耐えられる回数は更に少なく3回程度である [Kaner+93]。したがって、自動化の目安として3回以上の実行を考えるとよい。

## 実装

テストの自動化は、簡単なバッチファイルから、ベンダーの高価なテストツールまで様々なものがあるが、最も簡単なものでも最大限の利益を挙げることが出来る。

## 論理的根拠

[Beizer 95] では手動のテストは意味のないばかりか、危険であり、出来る限り自動化を計るべきであると述べている。

[Kaner+93] ではテスト担当者は同じ操作を3回以上行ってはいけないと述べている。

[Gamma+98] に代表されるように現在では単体テストの自動化ツールも数多く存在する。

## 関連するパターン

[Binder 99] の Test Harness Design で挙げられるパターンは、すべてテスト自動化に関するものである。

# 第5章 パターンの学習と発見

## 5.1 はじめに

これまで、テストのパターンは実のところ、テストを真剣に行う過程でおのずから生じるものであるということを見てきた。

そして、それを「非公式の単体テストのためのパターン・ランゲージ」で実証した。

しかし、自分自身でパターンを見出せたというだけでは十分ではない。どのような人でも同様のプロセスを踏めば、テストのパターンを自分自身に見出すことが出来るということを示す必要がある。

そのため、ここでは、4人の被験者を用い、実際にテストのパターンを学習することによってどのような効果が得られるのか、自分自身のパターンを生み出すことが出来るのか、本当にパターンはテスト手法と実践を橋渡しするものなのかを分析する。

### 5.1.1 目的

- 2 テストのパターンを学習することによって利用者のテスト能力が上がるのかを探る。
- 2 テストのパターンが本当にテスト手法を補うものであるかを探る。
- 2 テストのパターンを学ぶことによって、自分自身のパターンを見出せるかを探る。

### 5.1.2 方法

4人の被験者を用い、各人にテスト手法とテストのパターンを学習させる。学習の前後に、問題を出し、どのような能力向上があるかを見る。問題にはプログラムのテスト能力を測る古典的な問題である「Myersの三角形」のオブジェクト版を用いる。このプログラムとそれに対するテストケースを作成してもらい、同時にテストとパターンに関するアンケートを取る。

### 5.1.3 流れ

1. こちらから問題を提出
2. 被験者がプログラム及びテストケースを作成
3. 1度目の評価、主に被験者のテストとパターンに対する意識と能力を評価
4. テスト手法とテストのパターンの学習
5. 2度目の評価、テスト能力の向上、自分自身のパターンの発見を評価

#### 5.1.4 評価方法

プログラムと、テストケース、アンケートの3つから1回目には被験者のテストとパターンに関する意識と能力を、2回目はパターンの学習効果を測定する。

##### <sup>2</sup> 模範プログラム、模範テストケース

{ プログラムに残った欠陥

{ 正しいテストケースの数

##### <sup>2</sup> アンケート

{ 1回目：テストとパターンに関する意識と能力に対する質問

{ 2回目：パターンの学習効果とテスト手法との関係に対する質問

#### 5.1.5 与える知識

被験者のうちの3人に1回目の問題の後に以下の知識を与え、学習効果を測定する。各人に異なった知識を与え、下に行くにしたがって実際の場面で役立つことを仮定する。

##### <sup>2</sup> テスト手法 ([Beizer 90] より)

##### <sup>2</sup> パターン ([Binder 99] より)

##### <sup>2</sup> テスト手法 + パターン

### 5.2 学習前の評価

#### 5.2.1 目的

パターンとテスト手法に関して勉強してもらう前に、被験者にどれだけテストに関する知識があるかを知る。同時にどれだけテストを意識しているのか、パターンのようなものを持っているのかも知る。

#### 5.2.2 方法

被験者にあるプログラムとそれに対するテストケースを作成してもらう。同時にアンケートにも答えてもらう。

プログラムに残った欠陥、テストケースの正確性、アンケートの結果から被験者のテストとパターンに関する意識を分析する。

### 5.2.3 問題

コンストラクタで与えられた3つの数に対してそれが正三角形(3辺が等しいもの)、二等辺三角形(2辺のいずれかが等しいもの)、不等辺三角形(3辺が異なるもの)のいずれかを評価する三角形クラスとというものを作成し、これをテストするテストケースを同時に用意せよ。言語はJavaが望ましいが、C++などでも可能である。テストケースはどのような形で用意しても構わない。

ここで、三角形クラスのインタフェースは以下のような形をしているものとする。3辺  $a, b, c$  が三角形を為す条件は、

$$a > 0, b > 0, c > 0$$

$$s = (a + b + c) / 2 \text{ のとき } s > a \text{ かつ } s > b \text{ かつ } s > c$$

である。

コンストラクタに与える引数がこの条件を満たしていなくてもコンストラクタは例外は発生させないものとする。コンストラクタの引数の入力ドメインは3数ともに任意のint型である。

クラスのインタフェースは以下のようにになっているものとする。privateなメンバやメソッドは任意に作って構わない。

```
package triangle;
```

```
public class Triangle {
```

```
    // コンストラクタに与えられる引数は三角形の条件を満たしていなくてもよい
    public Triangle(int a, int b, int c) {
    }
```

```
    // 正三角形であるとき true
    public boolean isEquilateral() {
    }
```

```
    // 二等辺三角形であるとき true
    public boolean isIsosceles() {
    }
```

```
    // 不等辺三角形であるとき true
    public boolean isScalene() {
    }
}
```

ここで厳密な数学の定義から見ると奇妙であるがMyersの本来のプログラムに従うためにこのプログラム独自の仕様を定義する。本プログラムでは三角形のうち最もその定義に近いもののみがtrueを返す。つまり、

```
Triangle t = new Triangle(3, 3, 3);
```

とした場合、

```
t.isEquilateral() == true
```

となるが

```
t.isIsocel es() == false
```

となる。

数学の定義上は正三角形は二等辺三角形の部分集合であるが元のプログラムが最も適したものを表示するとあったのでこれに従っている。なお、このプログラムの仕様は [Binder 99] に従っている。

## 5.2.4 アンケート

被験者にはテストとパターンに関する意識調査のために表 5.1 のアンケートに答えてもらう。

表 5.1: 学習前のアンケート

No.	質問内容
ソフトウェアテスト一般について	
1	あなたはテストが好き (嫌い) ですか
2	その理由は何ですか
3	テストによって得られるものは何だと思えますか
4	あなたの考えるテストの定義とは何ですか
5	あなたはその定義に従って常にテストを行っていますか
6	あなたはいつテストについて学びましたか
7	あなたは何かからテストを学びましたか
8	あなたはプログラムを作成するとき、テストのしやすさを考慮していますか
9	あなたはいつテスト設計をしますか
10	テストを設計するときに図や表を使いますか
11	あなたはいつテストを実行しますか
12	何を使ってテストをしますか
13	テストケースは再利用できる形にしていますか
14	カバレッジ (網羅) の概念は知っていますか
15	カバレッジを実際に使っていますか
16	同値分割の概念を知っていますか
17	境界値 (限界値) 分析の概念を知っていますか
18	同値分割、境界値分析の概念をテスト設計に用いていますか
このプログラムに対するテストについて	

表 5.1: 学習前のアンケート

1	このプログラムの設計にはどれほど時間がかかりましたか
2	コーディングはどれほど時間がかかりましたか
3	テストの設計はどれほど時間がかかりましたか
4	テストの実行はどれほど時間がかかりましたか
5	デバッグにはどれほど時間がかかりましたか
6	このプログラムをテストするのにどのようなテストの知識、手法を用いましたか
7	その知識、手法を用いるのに自分なりに工夫した点がありますか
8	いつテストケースを作成しましたか
9	何に基づいてテストケースを作成しましたか
10	テスト終了の基準は何ですか
11	テストケースの選択の基準は何ですか
12	テストケースは再利用できる形になっていますか
13	どのような形でテストを行いましたか、手動ですか、何らかのツールを用いましたか
14	ブラックボックス、ホワイトボックスのどちらを重視しましたか
パターンについて	
1	テストを行う際に繰り返し用いている解決策のようなものはありましたか
2	それはテスト手法そのものですか、それとも別のものだと思いますか
3	その解決策はどのような知識を土台にしていると思いますか
4	その解決策はどのような問題を解決するのか振り返ってみたことはありますか
5	もっと良い解決策があると思いますか
6	もっと良い解決策を求めたことがありますか
7	もっと良い解決策はどこから学んでいますか

### 5.2.5 想定する欠陥

表 5.2: 想定する欠陥

欠陥の種類	状況
仕様の誤解	このプログラム独自の三角形の定義を理解していない
オーバーフロー デグレード	仕様に書かれているものをそのまま実装するとオーバーフローが発生する ある欠陥の修正によりほかの正しい機能が破壊される

### 5.2.6 有効なテスト戦略、用いる知識

ホワイトボックスのブランチカバレッジを使ってもオーバーフローの欠陥は検出できない。述語カバレッジについても同様。ブラックボックスの同値分割、境界値分析が必要になる。

仕様の誤解についてもテスト対象のプログラムのカバレッジを満たしただけでは十分ではない。テストドライバ側で、以下のように複数結果のテストを行わなければならない。すなわちコンストラクタで与える入力に対する出力は複数のメソッドの実行結果である。

// 正しい正三角形である場合、それが2等辺三角形と不等辺三角形でないことも検査しなくてはならない

```
public void test_Valid_Equilateral_Triangle() {
    Triangle t = new Triangle(3, 3, 3);

    assert(t.isTriangle());
    assert(t.isEquilateral());
    assert(t.isEquilateral() && !t.isIsocelles() && !t.isScalene());
}
```

表 5.3 に有効なテスト戦略、用いる知識を示す。

表 5.3: 有効なテスト戦略、用いる知識

欠陥	テスト手法	パターン
仕様の誤解	同値分割、境界値分析	Category-Partition, Combinational Function Test
オーバーフロー	同値分割、境界値分析	Category-Partition, Combinational Function Test
デグレード	回帰テスト	Incremental Testing Framework

デグレードに対するパターンは回帰テストのパターンが有効であるが、このプログラムの規模では考えなくても良い。

### 5.2.7 模範テストケースと結果

表 5.4 に模範テストケースと結果を示す。

表 5.4: 模範テストケースと結果

Test Case Description		Test Case Input			Expected Output
		a	b	c	
1	Valid scalene triangle	5	3	4	Scalene
2	Valid isosceles triangle	3	3	4	Isosceles
3	Valid equilateral triangle	3	3	3	Equilateral
4	First permutation of two equal sides	50	50	25	Isosceles
5	Second permutation of two equal sides	25	50	50	Isosceles
6	Third permutation of two equal sides	50	25	50	Isosceles
7	One side zero	1000	1000	0	Invalid
8	One side has a negative length	3	3	-4	Invalid
9	First permutation of two equal sides (invalid)	5	5	10	Invalid
10	Second permutation of two equal sides (Invalid)	10	5	5	Invalid
11	Third permutation of two equal sides (Invalid)	5	10	5	Invalid

表 5.4: 模範テストケースと結果

12	Three sides greater than zero, sum of of two smallest less than the largest	8	2	5	Invalid
13	Permutation 2 of line lengths in test 12	2	5	8	Invalid
14	Permutation 3 of line lengths in test 12	2	8	5	Invalid
15	Permutation 4 of line lengths in test 12	8	5	2	Invalid
16	Permutation 5 of line lengths in test 12	5	8	2	Invalid
17	Permutation 6 of line lengths in test 12	5	2	8	Invalid
18	All sides zero	0	0	0	Invalid
19	Three sides greater than zero, one side equals the sum of the other two	12	5	7	Invalid
20	Permutation 2 of line lengths in test 19	12	7	5	Invalid
21	Permutation 3 of line lengths in test 19	7	5	12	Invalid
22	Permutation 4 of line lengths in test 19	7	12	5	Invalid
23	Permutation 5 of line lengths in test 19	5	12	7	Invalid
24	Permutation 6 of line lengths in test 19	5	7	12	Invalid
25	Three sides at maximum possible value	MV	MV	MV	Equilateral
26	Two sides at maximum possible value	MV	MV	1	Isosceles
27	One side at maximum possible value	1	1	MV	Invalid

ここで順列のあるテストケースはブーリアン式の評価順序に依存した欠陥を検出するためのものである。

MVは例えばJavaのint型の最大値Integer.MAX\_VALUEである。この値は言語、環境によって異なる。

なお、このテストケースで完全に欠陥を検出できるとはいいきれない。例えば、万全を期すならばケース7,8は順列を取る必要があるかもしれない。また、25,26,27についても通常のドメインテストでは「境界値-1」の場合もテストする。

## 5.2.8 各テストケースの意味

表 5.5 に各テストケースの意味と想定する欠陥を示す。

## 5.2.9 被験者

表 5.6 の 4 人を被験者とする。

テストケース	意味、想定する欠陥
1 2 3	正しい入力に対し、正しい出力が得られるか
4 5 6	1つ違いのエラー、評価順序依存などブーリアン式の誤りを想定
7	0の場合で特殊にすりぬける欠陥を想定
8	符号の逆転やオペラント位置の不正を想定
9 10 11	ケース4,5,6の欠陥に加え、三角形そのものの妥当性チェックのスキップを想定、2辺が等しいと特殊な処理を行うかもしれないことを想定
12 13 14 15 16 17	ケース4,5,6の欠陥に加え、三角形そのものの妥当性チェックを行っているかを想定
18	オール0の場合に特殊にすりぬける欠陥を想定
19 20 21 22 23 24	ケース4,5,6の欠陥に加え、三角形そのものの妥当性チェックのスキップを想定
25	整数演算オーバーフローを想定、境界値で欠陥が多く発生するという統計にも基づく
26	ケース25に加え、2辺が等しいと特殊な処理を行うかもしれないことを想定
27	ケース25に加え、境界値では正常値ではない入力に対して、正常であると出力する可能性が高いことを想定

表 5.5: 各テストケースの意味

被験者	名前
A	深澤研究室 M1 渡部
B	深澤研究室 M1 山本
C	深澤研究室 M1 吉本
D	深澤研究室 M0 坂井

表 5.6: 被験者

### 5.2.10 被験者 A に対する評価

#### 用意したテストケース

表 5.7 に被験者 A の用意したテストケースを示す。

正解率は 11/27 である。ケース 6,7,8 で模範テストケース 7 の順列を取っており、この部分は模範テストケースより精度が高いため、+2 している。また、ケース 9,10,11 は順列とはなっていないが、1 辺がほかの 2 辺より大きい場合を a,b,c の各入力に対して行っているため評価できる。

#### 良い点

各テストケースについて、なぜそのテストケースを選んだかを明確に述べている。成立する条件と結果も予想している。

Test Case Description		Test Case Input			Expected Output
		a	b	c	
1	Valid scalene triangle	4	3	2	Scalene
2	Valid isosceles triangle	2	2	3	Isosceles
3	Valid isosceles triangle	3	2	3	Isosceles
4	Valid isosceles triangle	1	3	3	Isosceles
5	Valid equilateral triangle	3	3	3	Equilateral
6	One side zero	0	2	3	Invalid
7	One side zero	1	0	3	Invalid
8	One side zero	3	3	0	Invalid
9	Three sides greater than zero, sum of of two smallest less than the largest	6	2	3	Invalid
10	Three sides greater than zero, sum of of two smallest less than the largest	2	7	1	Invalid
11	Three sides greater than zero, sum of of two smallest less than the largest	3	3	10	Invalid

表 5.7: 被験者 A の用意したテストケース

#### 悪い点

上限の境界値をテストケースに入れていないためプログラムに整数演算オーバーフローに関する欠陥が残っている。また、ソースコード内の条件文分岐の演算を考慮すれば、上限の境界値である0だけでなく、負数もケースに含めるべきである。

#### プログラムに残った欠陥について

整数演算オーバーフローの欠陥が残存している。境界値分析を正しく行い上限の境界値をテストケースに入れていればこの欠陥は検出可能である。

#### 評価

テストケースの選択理由を明確に述べ、結果を予想している点は評価できる。それゆえに、入力ドメインの上限境界値をテストケースとしなかった点はおしい。

### 5.2.11 被験者 B に対する評価

#### 用意したテストケース

表 5.8 に被験者 B の用意したテストケースを示す。

正解率は 9/27 である。ケース 4,5,6 で模範テストケース 7 の順列を取っており、この部分は模範テストケースより精度が高いため、+2 している。

Test Case Description		Test Case Input			Expected Output
		a	b	c	
1	Valid scalene triangle	3	4	5	Scalene
2	Valid isosceles triangle	5	5	4	Isosceles
3	Valid equilateral triangle	2	2	2	Equilateral
4	One side zero	0	2	3	Invalid
5	One side zero	2	0	3	Invalid
6	One side zero	2	3	0	Invalid
7	Three sides greater than zero, sum of two smallest less than the largest	5	2	1	Invalid
8	Permutation 2 of line lengths in test 7	2	5	1	Invalid
9	Permutation 3 of line lengths in test 7	2	1	5	Invalid

表 5.8: 被験者 B の用意したテストケース

### 良い点

構造ベースの分岐網羅を満たすテストケースを作成しており、ソースコード上では分岐網羅を満たしている。分岐網羅が終了基準となっており、終了基準は明確である。

### 悪い点

機能ベースのテストを考慮していない。幸い、欠陥とはならなかったが、上限の境界値をテストケースに入れてテストを行うべきであろう。また、ソースコード内の条件文分岐の演算を考慮すれば、下限の境界値である 0 だけでなく、負数もケースに含めるべきである。また、例外ケースを特殊な数で行っている点が気になる。

### プログラムに残った欠陥について

模範テストケースに関する限り、なし。模範テストケースのテストをすべて通過している。int -> double 変換に不安があるが、テストを行っている限り問題はない。

### 評価

構造ベースの網羅基準を考慮している点と模範テストケースにプログラムがすべて合格した点は評価できる。しかし、もう少し機能レベルのテストケースを考慮しないとプログラムが複雑になった場合、問題が生じる。パターンの学習が期待される。

## 5.2.12 被験者 C に対する評価

### 用意したテストケース

表 5.9 に被験者 C の用意したテストケースを示す。

Test Case Description		Test Case Input			Expected Output
		a	b	c	
1	Valid scalene triangle	3	4	5	Scalene
2	Valid isosceles triangle	2	2	1	Isosceles
3	Valid isosceles triangle	1	2	2	Isosceles
4	Valid isosceles triangle	2	1	2	Isosceles
5	Valid equilateral triangle	1	1	1	Equilateral
6	One side zero	0	1	1	Invalid
7	One side zero	1	0	1	Invalid
8	One side zero	1	1	0	Invalid
9	Three sides greater than zero, one side equals the sum of the other two	1	1	2	Invalid
10	Permutation 2 of line lengths in test 19	2	1	1	Invalid
11	Permutation 3 of line lengths in test 19	1	2	1	Invalid

表 5.9: 被験者 C の用意したテストケース

正解率は 11/27 である。ケース 6,7,8 で模範テストケース 7 の順列を取っており、この部分は模範テストケースより精度が高いため、+2 している。

#### 良い点

構造ベースの分岐網羅を満たすテストケースを作成しており、ソースコード上では分岐網羅を満たしている。分岐網羅が終了基準となっており、終了基準は明確である。

#### 悪い点

機能ベースのテストを考慮していない。特に上限の境界値をテストケースに入れていないためプログラムに整数演算オーバーフローに関する欠陥が残っている。また、ソースコード内の条件文分岐の演算を考慮すれば、下限の境界値である 0 だけでなく、負数もケースに含めるべきである。また、例外ケースを特殊な数で行っている点が気になる。ケース 6,7,8 は 2 辺が等しいと特殊な処理を行うかもしれない欠陥を考慮すると 0 以外の数は等しくない一般値の方が望ましい。テストケース 1 つに対し、1 つの欠陥を想定するのが望ましい。

#### プログラムに残った欠陥について

整数演算オーバーフローの欠陥が残存している。境界値分析を正しく行い上限の境界値をテストケースに入れていればこの欠陥は検出可能である。

## 評価

構造ベースの網羅基準を考慮している点は評価できる。しかし、それでは欠陥を検出するには不十分である。機能ベースのテストも考慮する必要がある。

### 5.2.13 被験者 D に対する評価

#### 用意したテストケース

表 5.10 に被験者 D の用意したテストケースを示す。

Test Case Description		Test Case Input			Expected Output
		a	b	c	
1	Valid scalene triangle	3	4	5	Scalene
2	Valid scalene triangle	1	2	3	Scalene
3	Valid isosceles triangle	4	4	2	Isosceles
4	Valid isosceles triangle	0	1	1	Isosceles
5	Valid isosceles triangle	1	0	1	Isosceles
6	Valid isosceles triangle	1	1	0	Isosceles
7	Valid equilateral triangle	3	3	3	Equilateral
8	One side has a negative length	-1	3	2	Invalid
9	One side has a negative length	3	-1	2	Invalid
10	One side has a negative length	2	3	-1	Invalid
11	Two side has a negative length	-1	-1	2	Invalid
12	Two side has a negative length	-1	2	-1	Invalid
13	Two side has a negative length	2	-1	-1	Invalid
14	Two sides zero	0	0	1	Invalid
15	Two sides zero	0	1	0	Invalid
16	Two sides zero	1	0	0	Invalid
17	All sides zero	0	0	0	Invalid
18	All sides has a negative length	-1	-1	-1	Invalid
19	throws NumberFormatException	1E+09	1E+09	1E+09	Invalid

表 5.10: 被験者 D の用意したテストケース

正解率は 10/27 である。

#### 良い点

すべてが模範テストケースに劣っているわけではなく、模範ケースのケース 7,8 ではカバーしていない、0 及び負数に対する順列を行っている点は評価できる。

## 悪い点

テストケース選択の意図が見えない。プログラムの構造と境界条件を分析すれば、ケース 11 ~ 16 は明らかに冗長である。それに対し、模範テストケースで重要なケース 12, 19, 25, 26, 27 がすっかり抜け落ちている。特に上限の境界値をテストケースに入れていないのは致命的であり、そのためプログラムに整数演算オーバーフローに関する欠陥が残っている。

## プログラムに残った欠陥について

整数演算オーバーフローの欠陥が残存している。仕様で与えたとおりに式を実装するとこの欠陥が作りこまれることが分かっていない。分かっていなくても、境界値分析を正しく行い、上限の境界値をテストケースに入れていれば、この欠陥は検出可能である。

また、double 型と int 型の比較も潜在的な欠陥となりうる。

## 評価

ある程度の同値分割は考慮しているが、個々のテストケースは思いつきの域を出ていない。

### 5.2.14 アンケートの結果と評価

アンケートの解答から本プログラムにおけるテストとパターンに関する意識を分析する。表 5.11 に学習前のアンケートの結果を示す。番号は前記の質問の番号である。

表 5.11: 学習前のアンケートの結果

No.	解答	
	被験者 A	被験者 B
	ソフトウェアテスト一般について	
1	好きではない	あまり好きではない
2	面倒だから	やや面倒だから
3	プログラムの信頼性	プログラムの正当性に対する保証
4	予測不能の事態を洗い出す	作成したコードに対する正当性の補完するもの
5	行っていない	完全ではないにしろそう考えて行っている
6	大学の授業	大学 3 年生
7	深澤先生の授業	ソフトウェア工学の講義
8	少しだけ行った	あまり考えていない
9	プログラムが完了してから	コーディングが一通り終わった後
10	使わない。簡単なメモ程度	構造が複雑な場合は使う。今回は使っていない
11	パターンを洗い出し終わってから	コーディングが一通り終わった後

表 5.11: 学習前のアンケートの結果

12	ドライバプログラムを書く	何も使っていない
13	行っていない	していない
14	知っている	アウトライン程度は知っている
15	面倒なのでできる範囲で	使っている
16	忘れていた	アウトライン程度は知っている
17	知っている	アウトライン程度は知っている
18	使っていない	あまり用いていない
このプログラムに対するテストについて		
1	10分程度	30分弱
2	30分	1時間強程度
3	20分	10分程度
4	5分	5分程度
5	10分	30分程度
6	特になし	講義で習った網羅の概念等
7	無駄なテストケースを増やさない	特になし
8	プログラミング終了後	一通りのコーディングが終了後
9	仕様書の三角形の定義	無回答
10	最低限の要求を満たせたかどうか	考え得るすべての状態に関して、期待される結果を得られたとき
11	プログラムの構造を考慮した最低限のテストケース	極端な特異性を持つ場合とごく一般的な値の場合
12	なっていない	あまりない
13	コンストラクタの値を入力、3つのメソッドの返り値を表示	手動
14	極力、テストの労力を減らすため、ホワイトボックス	ブラックボックス重視
パターンについて		
1	プログラムの構造を単純なものにして、ホワイトボックステストを簡単にする	特になし
2	別物だと考える	無回答
3	プログラムの知識	無回答
4	ない	無回答
5	わからない	あると思う
6	ない	求めたことはない
7	学んでいない	無回答
	被験者 C	被験者 D
ソフトウェアテスト一般について		
1	あまり好きではない	まだ経験が無いので何とも言えない

表 5.11: 学習前のアンケートの結果

2	必要だとは考えている	既に述べた
3	暫定的なアプリケーションに関する保証	バグの発見
4	自分のプログラムの論理的な欠陥を埋める作業	プログラム上のミスを発見するため
5	最低正しく動くまではテストを行う	やったことがない
6	ソフトウェア工学の授業	研究室に入って、割と早い時期にテーマがテストになった
7	同上	文献から
8	「自分」が読んだときに何をしているのか、プログラムがどのような流れになっているのかを考慮	全く考える余裕がない
9	テストを行う直前	行わない
10	基本的には使わない。それほど大量なテストケースになる場合は細かな単位で行うため	テスト自体を行わない。今回は図も表も使っていない
11	コンパイルでデバッグ終了後(または、デバッグと並行して)	テストを行わない
12	エディタとコンパイラ	普通に引数を3辺に見立てて行った。よって、特別なツールなどは使っていない
13	いいえ	テキストファイルに記述
14	はい	知っている
15	はい	使ったつもり
16	はい	知っている
17	はい	これは覚えていたつもりだったが、今読み返したら少し違った
18	はい	同値分割は用いたつもり。境界値分析のほうは、やっていない
このプログラムに対するテストについて		
1	設計自体は1時間以内	数分くらい
2	1時間程度	1時間程
3	フローチャート書いたため、検証と併せて1時間強	10~20分くらい
4	30分以内	やったのはほんのいくつか
5	30分程度	エラーが出てそのままにしてあるので、0
6	すべての条件実行とパスを通る、境界近くの値を用いる	同値分割は頭にあった

表 5.11: 学習前のアンケートの結果

7	三角方程式改良	特になし
8	実行直前と設計と同時	プログラムが出来てから
9	上記(フローチャート)	ソースコード
10	今回に限っては、プログラムが簡単だったため。すべてのパスが正しかったため	自分でもう大丈夫だと思ったから
11	良く分からない	ソースコード
12	いいえ	テキストファイル
13	手動。テストプログラムの数値変更	手動
14	フローチャートを書いているのでホワイトボックス	ホワイトボックス
パターンについて		
1	どこまでが正しいパスを通過しているのか、調べる	なし
2	デバッグに近い	なし
3	おおむね既に上で書いたとおり	なし
4	おおむね既に上で書いたとおり	なし
5	はい	分からない
6	はい	なし
7	勉強したい	これから頑張る

#### 被験者 A のアンケートに対する評価

より良い解決策を求めたことがないといいきっているのには問題はあるが、比較的きちんとテストを行っている。一定以上の複雑さが生じた場合、テストのためにプログラムを分割するなど、パターンとなりそうなものも持っている可能性がある。この被験者を見る限りテスト手法の知識がそのまま実際のテストに用いられるわけではなく、何らかの中間の知識が必要なことが分かる。これはパターンであるかもしれないし、全く別のものかもしれない。

#### 被験者 B のアンケートに対する評価

被験者 B のテストケースは少ないが、作成した対象プログラムは模範テストケースをすべて合格している。テストよりも正しくコーディングを行う防衛的プログラミングの能力の方が高いのでテストを行う必要性をそれほど感じていないようである。コードをみてもプログラミング能力は高いようなのでいくつかの基本的なテストのパターンを身に付ければ、更にプログラムの質が上がるであろう。コーディング能力が高いために逆テストに対する関心が低いのが残念である。

## 被験者 C のアンケートに対する評価

テストに関する意識は比較的高い。ただし、テストとデバッグを混同しており、テストに対する考え方は [Beizer 90] のいうフェーズ 0 と 1 の中間である。

テストの終了基準として構造ベースのカバレッジを挙げており、それで十分としているが、整数演算オーバーフローの欠陥が残っている。この欠陥を検出するためには機能ベースのカバレッジが必要となるが、そこまでは頭が回っていないようである。

より良いテストの方法を求めたいとのことなので、Binder のいくつかのパターンを学べばもう少し、良いテストを行えるかもしれない。

## 被験者 D のアンケートに対する評価

全体としてテストに関する意識が低いので現時点でパターンを見出すのは難しい。テストが必要なことは分かっているがほとんど行わないという典型的なアンチパターンのみである。研究用のプログラムしか作っていないので仕方のないことかもしれない。今後の学習に期待する。

### 5.2.15 総論

現在の被験者の状況ではそれほどテストに対する意識が高くないため、テストのパターンといえるようなものは見出せなかった。1 週間の勉強により多少の向上が見られることを期待する。

研究のためのソフトウェアではユーザーと作成者が同一であるために厳密なテストは必要ないため、この環境で良いパターンを見出すというのは難しいことなのかもしれない。

## 5.3 学習

被験者 A, B, C に対し、それぞれ [Beizer 90] と [Binder 99] から表 5.12 のテスト手法及びパターンを学習してもらおう。与えるテスト手法及びパターンは、有効なテスト戦略、用いる知識に基づく。

分類	知識	文献
テスト手法	ドメインテスト	[Beizer 90]
パターン	Category-Partition Combinational Function Test Incremental Testing Framework	[Binder 99]

表 5.12: 学習するテスト手法及びパターン

学習の目的は、

- 2 テストのパターンを学習することによって利用者のテスト能力が上がるのかを探る。
- 2 テストのパターンが本当にテスト手法を補うものであるかを探る。
- 2 テストのパターンを学ぶことによって、自分自身のパターンを見出せるかを探る。

である。

各人にはテスト手法のみ、パターンのみ、両方に分けて学習してもらう。振り分けは表 5.13 のとおりである。

被験者	学習範囲
A	テスト手法
B	パターン
C	テスト手法とパターン

表 5.13: 学習の振り分け

## 5.4 学習後の評価

### 5.4.1 目的

- 2 テストのパターンを学習することによって利用者のテスト能力が上がるのかを探る。
- 2 テストのパターンが本当にテスト手法を補うものであるかを探る。
- 2 テストのパターンを学ぶことによって、自分自身のパターンを見出せるかを探る。

### 5.4.2 方法

テスト手法及びパターンを学習した被験者 A,B,C に学習前に出したのと同様のプログラムとそれに対するテストケースを作成してもらう。同時にアンケートにも答えてもらう。

プログラムに残った欠陥、テストケースの正確性、アンケートの結果から被験者のテスト手法及びパターンの学習度、自分自身のパターンの発見があるかを調査する。

### 5.4.3 問題

5.2 と同様。

### 5.4.4 アンケート

テスト手法とパターンの学習効果を見るため被験者には表 5.14 のアンケートに答えてもらう。

表 5.14: 学習後のアンケート

No.	質問内容
	学習効果について（テスト手法、パターン共通）全員

表 5.14: 学習後のアンケート

1	学習は何らかの役に立ちましたか、例えばよりよいテストケースを選択できたなどの効果はありましたか
2	学習によって有益な（欠陥を検出できるような）テストケースの数を増やすことが出来たか、前回残っていた欠陥を発見できましたか
3	対象とするプログラムに対し、特定の欠陥を予想してテストケースを作れるようになりましたか
4	回テストケースが足りなかったのは単にテスト手法について忘れていただけだったと思いますか、それとも欠陥を予測してテスト戦略を立てなかったからだと思いますか
テスト手法について 被験者 A、被験者 C	
1	テスト手法だけでテストケースを実装する際に困難はありませんでしたか
2	困難であるとしたらテスト手法には何が欠けていると思われましたか
3	もしかしたらその欠けているというのは具体例であって、具体例があればテスト手法でも十分だと思いますか
4	テスト手法の理解が困難であるのは、文章、記述上の問題もあると思いますか
パターンについて 被験者 B、被験者 C	
1	学習したパターンによって有益な（欠陥を検出できるような）テストケースの数を増やすことが出来たか、前回残っていた欠陥を発見できましたか
2	学習したパターンが対象とするプログラムの各面で有益である理由を理解できましたか
3	与えられたパターンの学習は容易でしたか、難しかったですか（英語であるというのは除いてください）
4	各パターンが指摘する欠陥の危険性を理解することにより、テストに対する積極性が高まりましたか（パターンのフォースはあなたに訴えかけるものがありましたか）
5	テスト自動化のパターンによって、テスト自動化の理由と方法を理解できましたか
6	テスト自動化を実践してみましたか
7	学習したパターンを参考にして自分自身のパターンを発見できましたか
テスト手法とパターンの関係について 被験者 C	
1	テスト手法とパターンの一方があれば十分有益なテストケースが作成できると思われましたか
2	両方とも必要だとしたら、どちらがどちらを知識として前提にしていると思いますか
3	テスト手法とパターンではテスト手法のほうが扱う粒度が大きいと思われましたか
4	テスト手法とパターンではどちらが欠陥をより具体的に特定しているように見えましたか
5	テスト手法とパターンではどちらがよりテストに対する積極性を高められましたか

#### 5.4.5 想定する欠陥

5.2 と同様。

#### 5.4.6 有効なテスト戦略、用いる知識

5.2 と同様。

#### 5.4.7 模範テストケースと結果

5.2 と同様。

#### 5.4.8 各テストケースの意味

5.2 と同様。

#### 5.4.9 被験者

テスト手法及びパターンを学習した表 5.15 の 3 人の被験者を対象とする。

被験者	名前	学習範囲
A	深澤研究室 M1 渡部	テスト手法
B	深澤研究室 M1 山本	パターン
C	深澤研究室 M1 吉本	テスト手法とパターン

表 5.15: 学習を行った被験者

#### 5.4.10 被験者 A に対する評価

##### 改良したテストケース

表 5.16 に被験者 A の改良したテストケースを示す。

正解率は 17/27 である。ケース 13,14,15,16,17 はなくてもほかのテストケースで同様の欠陥を検出できるので除いた。ただし、一般値という点では入っていても問題はない。ケース 1,2,3,18,19,20 は順列を取っている点で模範テストケースより優れているために +4 している。

##### 良い点

ドメインテストの知識を活かし、各テストケースについて、なぜそのテストケースを選んだかを更に明確に述べている。成立する条件と結果も予想している。

Test Case Description		Test Case Input			Expected Output
		a	b	c	
1	One side zero (a = 0)	0	2	3	Invalid
2	One side zero (b = 0)	1	0	3	Invalid
3	One side zero (c = 0)	3	3	0	Invalid
4	Three sides greater than zero, one side equals the sum of the other two (s = a)	5	2	3	Invalid
5	Three sides greater than zero, one side equals the sum of the other two (s = b)	3	6	3	Invalid
6	Three sides greater than zero, one side equals the sum of the other two (s = c)	4	6	10	Invalid
7	Valid isosceles triangle (a = 1)	1	2	2	Isosceles
8	Valid isosceles triangle (b = 1)	2	1	2	Isosceles
9	Valid isosceles triangle (c = 1)	2	2	1	Isosceles
10	Valid scalene triangle (s = a + 0.5)	4	2	3	Scalene
11	Valid scalene triangle (s = b + 0.5)	2	4	3	Scalene
12	Valid scalene triangle (s = c + 0.5)	3	2	4	Scalene
13	Valid scalene triangle	2	3	4	Scalene
14	Valid isosceles triangle	2	2	3	Isosceles
15	Valid isosceles triangle	3	2	3	Isosceles
16	Valid isosceles triangle	1	3	3	Isosceles
17	Valid equilateral triangle	1	1	1	Equilateral
18	One side at maximum possible value	MV	1	1	Invalid
19	Permutation 2 of line lengths in test 18	1	MV	1	Invalid
20	Permutation 3 of line lengths in test 18	1	1	MV	Invalid
21	a + b + c = MV + 1	715827884	715827882	715827882	Isosceles
22	a + b + c = MV	715827883	715827882	715827882	Isosceles

表 5.16: 被験者 A の改良したテストケース

### 悪い点

こちらの仕様の問題もあり、被験者のテストケースと模範テストケースでは上限境界値における期待結果が異なっていた。仕様ではケース 21,22 に対し、二等辺三角形を期待していたが、被験者は不等な入力としている。

### プログラムに残った欠陥について

整数演算オーバーフローの欠陥が残存している。これについてはこちらの仕様が明確でなかったことが理由であるかも知れない。

## 評価

ドメインテストの知識を有効にいかし、境界におけるテストケースを設定できている。学習前に比べ、テストケース選択の論拠が明確になった。ただし、前回あった欠陥を回避するためのコードが複雑になっている。

### 5.4.11 被験者 B に対する評価

#### 追加したテストケース

表 5.17 に被験者 B の追加したテストケースを示す。

Test Case Description		Test Case Input			Expected Output
		a	b	c	
10	Permutation 2 of line lengths in test 2	4	5	5	Isosceles
11	Permutation 3 of line lengths in test 2	5	4	5	Isosceles

表 5.17: 被験者 B の追加したテストケース

正解率は 11/27 に上昇した。

#### 良い点

2 辺が等しい場合の評価順序依存を考慮したテストケースを追加している。Combinational Function Test パターンの学習効果が反映されたものと思われる。

#### 悪い点

依然として、上限の境界値及び負数がテストケースに入っていない。Category-Partition パターンの学習が反映されていない。

#### プログラムに残った欠陥について

学習前の評価時点で模範テストケースに関する限り、なし。模範テストケースのテストをすべて通過している。int -> double 変換に不安があるが、テストを行っている限り問題はない。

## 評価

パターンの学習によって、ある程度欠陥を予想してテストを行うことが可能になっている。しかし、以前として境界値付近のテストケースが弱い。

## 5.4.12 被験者 C に対する評価

### 改良したテストケース

表 5.18 に被験者 C の改良したテストケースを示す。

表 5.18: 被験者 C の改良したテストケース

Test Case Description		Test Case Input			Expected Output
		a	b	c	
1	Valid equilateral triangle (Open o <sup>®</sup> inside)	1	1	1	Equilateral
2	First permutation of one side zero (On Point)	0	1	1	Invalid
3	permutation 2 of line lengths in test 2	1	0	1	Invalid
4	permutation 3 of line lengths in test 2	1	1	0	Invalid
5	Three sides at maximum possible value(Open o <sup>®</sup> inside)	MV	MV	MV	Equilateral
6	maximum on point	MV+1	MV	MV	Invalid
7	permutation 2 of line lengths in test 6	MV	MV+1	MV	Invalid
8	permutation 3 of line lengths in test 6	MV	MV	MV	Invalid
9	Two equal sides (On Point)	2	1	1	Invalid
10	permutation 2 of line lengths in test 9	1	2	1	Invalid
11	permutation 3 of line lengths in test 9	1	1	2	Invalid
12	Two sides at maximum possible value	1	MV	MV	Isosceles
13	permutation 2 of line lengths in test 12	MV	1	MV	Isosceles
14	permutation 3 of line lengths in test 12	MV	MV	1	Isosceles
15	One side at maximum possible value	MV	MD+1	MD+1	Isosceles
16	permutation 2 of line lengths in test 15	MD+1	MV	MD+1	Isosceles
17	permutation 3 of line lengths in test 15	MD+1	MD+1	MV	Isosceles
18	Upper O <sup>®</sup> Point for Valid equilateral triangle	MD1+1	MD1-1	MD1	Scalene
19	permutation 2 of line lengths in test 18	MD1	MD1+1	MD1-1	Scalene
20	permutation 3 of line lengths in test 18	MD1+1	MD1	MD1-1	Scalene
21	Lower O <sup>®</sup> Point for Valid equilateral triangle	MD1-1	MD1+1	MD1	Scalene
22	permutation 2 of line lengths in test 21	MD1	MD1-1	MD1 +1	Scalene
23	permutation 3 of line lengths in test 21	MD1-1	MD1	MD1+1	Scalene
24	Upper O <sup>®</sup> Point for Valid equilateral triangle On Point for Valid Isosceles triangle	MD1+1	MD1+1	MD1-2	Isosceles
25	permutation 2 of line lengths in test 24	MD1-2	MD1+1	MD1+1	Isosceles
26	permutation 3 of line lengths in test 24	MD1+1	MD1-2	MD1+1	Isosceles
27	Lower O <sup>®</sup> Point for Valid equilateral triangle On Point for Valid Isosceles triangle	MD1-1	MD1-1	MD1+2	Isosceles
28	permutation 2 of line lengths in test 27	MD1+2	MD1 -1	MD1-1	Isosceles
29	permutation 3 of line lengths in test 27	MD1-1	MD1+2	MD1-1	Isosceles

ここで、 $MD=MV/2$ ,  $MD1=(MV+1)/2$  とする。

正解率は模範テストケースを完全に上回っている。正三角形、二等辺三角形、不等辺三角形の境界を求めており、模範テストケースより精度が高い。

#### 良い点

ほぼ万全といえる。ドメイン分析の学習が生きている。

#### 悪い点

なし。強いていえば、下手をするとプログラムを書く以上にドメイン分析に時間がかかる。

#### プログラムに残った欠陥について

模範テストケースに関する限り、なし。

#### 評価

ドメインテストの知識がいかされテストケースの数が大幅に向上している。ほとんど何もいうことがないほどテストの精度は高く、模範テストケースを上回っている。

ただ、テスト自動化を行っていないので、これだけのテストケースを手作業で求め、入力する際に誤りを含む確率が高くなることには注意しなければならない。また、仕様変更や機能追加で回帰テストを行う際にも手作業での入力は困難である。

### 5.4.13 アンケートの結果と評価

#### アンケートから

- 2 テストのパターンを学習することによって利用者のテスト能力が上がったか
- 2 テストのパターンが本当にテスト手法を補うものであるか
- 2 テストのパターンを学ぶことによって、自分自身のパターンを見出せるか

を分析する。

表 5.19 と表 5.20 にアンケートの結果を示す。

表 5.19: 学習後のアンケート (共通)

No.	解答
	学習効果について (テスト手法、テストパターン共通) 全員

表 5.19: 学習後のアンケート (共通)

	被験者 A	被験者 B	被験者 C
1	少しは役に立った。普段面倒でやっていない部分を見直すきっかけにはなった	テストケースの不足を発見できたという点で、効果があった	はい。とりあえず、学習にのっとなってテストケースを作成したため
2	はい	不足していたテストケース 2 つは発見できた	テストケースの数は増えたとし、最初のプログラムの欠陥は駆除できた
3	多少は作れるようになった	今回の結果だけでは何とも言えないが、今回の学習がその足しにはなったような気にはなっている	パステスト及び境界値チェックにおいては作れるようになったが、ドメイン分析では出来なかった
4	コンピュータの値の上限を考慮に入れるのを忘れていたため	後者	前回はミスだったと考える。完全に頭からは抜けていた

表 5.20: 学習後のアンケート (テスト手法、パターン)

No.	解答	
テスト手法について 被験者 A、被験者 C		
	被験者 A	被験者 C
1	特になかった	有。せめて 3 次元以上でのサンプルを提示しなければ何をやっているのかが分からない。2 次元では話を単純化しすぎである
2	わからない	できれば、手法をいかせるツールが欲しい
3	そうかもしれない	確実に少しは楽になった。ただ、それでも難しいような気がする
4	具体例があれば問題ない	数学的にはしっかり筋が通っている。ただ、そのイメージをふくらますための具体的なサンプルや境界を判別する助けとなるツールがないとつらい
パターンについて 被験者 B、被験者 C		
	被験者 B	被験者 C
1	学習効果について 2 の答えと同様	代表的な点のみを考慮する Combinational Function Test では欠陥は出てこなかったことは前記
2	完全ではないが、ある程度は理解できた	実践投入しやすい点。特殊な場合をのぞき一般的な欠陥は取り除けること

表 5.20: 学習後のアンケート (テスト手法、パターン)

3	英語という部分を除いてもやや難しかった	容易
4	積極性というか、テストの必要性みたいなものは感じた	いいえ。恐らくある程度の「代表点によるチェック」以外のことは行わない。個人的に組むプログラムでは面倒なため。お金が絡むプログラム(実務)でも恐らくしない
5	やはり完全ではないが、ある程度は理解できた	なんとなくはわかったが、めんどくさそうと言うのが本音である
6	そこまではしていない	いいえ。標準入力とファイル出力ぐらいは導入を試みたが、今回はMVのような定数が多数でてくるので、かえって面倒。原始的方法の方がコストは低かった
7	そこまでには至っていない	今回はドメイン分析の方を主体にしたので、いいえ。普段も大きく変わったことをすることはない。素直に自分のやることを振り返ればパターンになっていると言われそうである。でも本当に Combinational Function Test くらいしかない
テスト手法とパターンの関係について 被験者 C		
被験者 C		
1	環境が整っても正直ドメイン分析はやりたくない。ただ、性能はある程度高いと思われる。実践投入の困難さで有益といえるかは不明。パターン単体ではドメイン分析ほどの性能はない。簡単な点と境界付近を調べる知識とを組み合わせれば、使える点が評価できる	
2	パステストの時点で分岐した case はドメインであるので、パステストがドメイン分析の考え方を使っていると考えるほうが自然。逆はドメインが必ずしもパスに対応しているわけではないので少し不自然。本質的にはおなじことのような気もする	
3	はい。ただ、今回の例からはそれを読みとるのは困難かもしれない。今回の例ではパスが単純なこととそのわりにドメイン分析が困難であることから、ドメイン分析のほうが細々と考える必要があった。テストケースの量からも分かると思う。もっと大規模なプログラムを考えると Combinational Function Test が使えないので、その場合にドメイン分析をするということになるであろうが、これ以上複雑な場合にドメイン分析は絶対やりたくない	
4	目に見えて欠陥の原因が分かるのはパステストのようなパターン。テスト手法の場合はドメインの欠陥はわかるが、候補はいくつか考えられる	
5	ドメイン分析は嫌いである	

#### 被験者 A のアンケートに対する評価

被験者 A はテスト手法のみでも具体例があれば問題ないという結論である。これは予想外であったが、被験者 A のテストケースの作成方法を見ると、テスト手法を実際の場面に落とし込む方法を既に身に付けているようである。そのため、テスト手法のみでも問題がなかったと考えられる。すなわち、既にほかの被験者が学習した Category-Partition と Combinational Function Test の両パターンに近い形のものを持っていたと考えられる。

#### 被験者 B のアンケートに対する評価

元々プログラムに欠陥がなかったため、パターンの学習によるテストケースの追加のメリットは薄かったと被験者は感じている。ただ、パターンの想定する欠陥によってテストの積極性が高まったのは確かである。

最大の関心であるパターンの発見については、この短期間では困難であるということが分かる。各人のパターンは意識せず何度も繰り返し用いているものであるが、より良いテストを求める過程で生じるものなので、そのより良いテストを求めるというのを常に行っていない状況では、例えほかのパターンを見ても自分自身のものを見出すというのは難しいことなのかも知れない。

#### 被験者 C のアンケートに対する評価

被験者は正確などメイン分析を行い高精度のテストケースを作成してくれたが、これ以上複雑なプログラムではドメイン分析を行いたくないという意見である。ここにテスト手法の限界を見ることが出来る。欠陥の検出力も高く、テスト手法に具体例を載せれば分かりやすくはなるのかも知れないが、以前として何らかのツールの補助がなければ、プログラマが片手間に行うほど容易なものとはならないということが分かる。

対して、ドメインテストに対応するパターンである Category-Partition + Combinational Function Test では正確なドメイン分析に比べて欠陥除去の正精度は低いが行いやすいという利点があり、これがプログラマの単体テストで何度も行われてきた理由であると考えることが出来る。

現場のプログラマたちがいうように、確かに各種のテスト手法は欠陥の検出率も高く有用であると思えるが、実行に移すのが困難であれば意味がない。そのためある程度の欠陥を妥協し、現場で無理なくプログラマが行えるようにしたものがパターンであることが分かる。手法とパターンのどちらが正しいかという話は不毛であり、研究者がより有効なテスト手法を考え、それをプログラマ、テスト担当者が実践に移し、個々の場合に対応させたものがパターンであるといえる。

### 5.4.14 総論

#### 利用者のテスト能力の向上について

いずれの被験者も何も情報を与えていない最初の状態に比べて、テストの能力は大幅に向上している。特にテスト手法とパターンの両方を学習した被験者 C は顕著であり、模範テストケー

スすら上回っている。

テスト手法とパターンの両方ともそれを「知る」ことによって能力向上を図ることが出来るというのは予想通りであった。ただ、実際に継続して行えるかという別問題である。行うことは可能であるが、余りにも面倒で非現実的であるというのがドメインテストの特徴であった。逆に Category-Partition と Combinational Function Test の2つのパターンは厳密性においてドメインテストに劣るものの実用という観点では極めて有効でありプログラムの役に立つものであり、継続して行うには適するものであるといえる。ここが理論を土台としているテスト手法と、現場を土台としているパターンの差であるといえる。

あくまでテスト手法はこの場面でパターンを適応する際の土台として学習するというのが現実的な解であるといえる。

## テスト手法とパターンの関係について

前記で述べたように、テスト手法は強力ではあるが、すべての場面で手法通りにテストを行うのにはプログラマにおけるテストの位置を考慮すると時間的にも空間的にも困難である。そのため、テスト手法を個々の場面で適応する実用的な知恵が必要となる。その知恵がパターンであり、手法と現場の状況を間を取り結ぶものである。

また、良いテスト手法でも何らかの補助ツールがない限りはたとえテスト設計のパターンを用いても困難であることが被験者のアンケートからは分かる。そのためにはテスト自動化のパターンとそれを実装したツールが欠かせないことが分かる [Gamma 98]。模範テストケースの作成にはこの自動化ツールを用いたのでテストの苦痛はかなり減少した。「非公式の単体テストのためのパターン・ランゲージ」で考察したことを踏まえて、効率的かつプログラマに精神的苦痛をもたらさないテストには知識面でのテスト手法とパターン、物理面でのツールの連携が欠かせないということが分かった。

## 自分自身のパターンの発見について

各被験者ともこの実験だけでは自分自身のパターンの発見、何がパターンで何がパターンではないのかが分かるのは困難であると述べている。

自分自身のパターンはテストを何度も行う過程でおのずから完成されるものであり、一度ほかのパターンを用いてみたから発見出来るものではないということが分かる。自分自身のパターンとは様々なテスト手法、コンストラクションの方法、デバッグの方法などを取り込んだうえで作られるものであり、これまで余りテストを行ってこなかったプログラマがテストを行いはじめたすぐに見つけ出すというというのは困難であることが分かる。

しかし注意深く観察すれば、パターンらしいものもある。例えば、テスト設計を事前に行うことも1つのパターンであり、プログラム作成後に行うことも1つのパターンである。また、「非公式の単体テストのためのパターン・ランゲージ」のようにプログラムを作成しながらテストケースを追加し、テストケースが必要以上に複雑になったら、メソッドの分割、リファクタリングを行うというのもテストケースの数や複雑度が定量的に求められるものであれば、パターンといえるものかも知れない。被験者 A についてはテストケースの作成に際し、仕様から別にテストの仕様を考えるというパターンを身に付けていた。

このように注意深く自分自身の行動を振り返ってみればパターンらしいものを見つけることが可能である。それを意識しながら自分自身の方法を洗練させていけば、それは独自のパターンとなっているかもしれないし、既存の優れたパターンをなぞったものになっているかもしれない。いずれにせよ、それはこれまで述べてきたようにテストを真剣に行う過程で作られるものにほかならない。それを見つけるためにはプログラマ自身の努力だけでなく、ほかの優れたパターンとそのパターンをサポートするようなツールが必要であるというのが結論である。

## 第6章 結論

この研究をはじめた当時、なぜテストにおいてパターンは少ないのか、どうやったら発見することが出来るのかということに注目していた。無理やり作り出そうとしたこともあったが失敗した。原因はソフトウェア業界にあるのか、テストそのものにあるのか、パターンという概念そのものにあるのかということも分析した。

しかし、結局のところ、[Alexander 79]における建築のパターンと同様に、テストのパターンの発見が難しかったのは個人的な問題であり、テストのパターンはテストを真剣に行う過程おのずから存在するものであった。無理に見つけ出したり、提唱したりするものではない。プログラマ、テスト担当者、品質管理技術者のいずれもテストを常に行う過程でその行為はおのずからパターンとなっている。

テストにおいてパターンが生まれ難かったのは、極度の無関心と関心の2つのみが存在してきたからである。極度の無関心はテストにおける質を求めないためパターンを生み出すことが出来ず、極度の関心はすべての問題を「しなければならぬ」と縛ることによって自然なパターンではなく、規則を作り出していた。必要なのはテストを無視することでも極度に重視することでもない。ソフトウェアの質というものを考えたときに、自分は何が出来るのかという問いに対し、「テスト」という一手段があるに過ぎない。質のためには何が出来るのかを真剣に考えている中で、おのずからテストはある形を取ってくる。それは目的、状況によって異なるが、共有できるパターンの形を取る。

[Alexander 79]の不滅の特性、そして老子の「道」が説くとおり、むやみやたらに新しい方法に飛びつくのではなく、自分の中にある良い解決策に注目し、それを公開していくことこそがテストのパターンの始まりである。そのパターンの発見の過程を本論文では示し、その実践を「非公式の単体テストのためのパターン・ランゲージ」で示した。そして、テストのパターンがテスト手法の補助となり、新たなパターンを生み出すきっかけとなることを「パターンの学習と発見」で示した。被験者たち自身はパターンを見つけ出すことは出来なかったが、パターンとなりそうなものを見出すことは出来た。

テストを求める過程でパターンがおのずから生まれるというのは

"TestingPatterns"(<http://c2.com/cgi-bin/wiki?TestingPatterns>)

における最近の活動においても見る事が出来る。テストファーストを重視する方法論であるXPの実践と共にテストのためのパターンが数多く提唱され始めたのである。本論文が主張してきたことは既に世の中では行われていたようである。その意味では落胆もあるが、同時にプログラマがテストを真剣に行えばパターンはおのずから生まれるという本論文の正しさが証明されたことでもある。

パターンは一部の人間のためのものではなく、求めればすべての人間が共有し、発見できるものであり、テストにおいてもその活動が始まってきたのは喜ばしいことである。

# 謝辞

まず、研究室の深澤先生と研究に協力してくれた M1 の吉本君、山本君、渡部君、M0 の坂井君に感謝致します。

そして、本研究で最も重要な「非公式の単体テストのためのパターン・ランゲージ」の完成に力を貸して頂いた友野さん、長谷川さん、藤野さんをはじめとする JLoP のメンバーに感謝致します。

また、テスト技術に関して積極的な議論の場を提供して頂いたテスト技術者交流会の西さん及びメンバーの皆様に感謝致します。

# 付録A パターンにおける実例

## A.1 サンドイッチアプローチパターンの例

0以上の整数を引数とし、各桁の数を足して返すようなメソッド `int digit_plus(int a_n)` について考える。`int digit_plus(int a_n)` は0以上の整数 `a_n` に対し `digit(0) = 0`、`digit_plus(123) = 6`、`digit_plus(777) = 21` のように振る舞うものとする。

本パターンに従ってメソッドの実装を行いながらテストを行う手順は以下のようになる。

1. 最初に「まだこのメソッドは出来ていない」という例外を発するステートメントを記述する。

```
private static int digit_plus(int a_n) {
    throw new RuntimeException("Method digit_plus is not implemented yet.");
}
```

2. いかなる入力値に対しても、「まだこのメソッドは出来ていない」という例外を送出することを確認するテストを行う。
3. 入り口に入力ドメインを検査するコードを加える。

```
private static int digit_plus(int a_n) throws IllegalArgumentException {

    // 引数の領域を検査
    if(a_n < 0) {
        throw new IllegalArgumentException("Your input is a_n = " + a_n + ". " +
            "Method digit_plus can accept a_n >= 0.");
    }

    throw new RuntimeException("Method digit_plus is not implemented yet.");
}
```

4. 入力ドメインに関して、正常値、不正値のテストを行う。Onポイントの `a_n = 0`、O<sup>o</sup>ポイントの `a_n = -1` などである。`a_n < 0` で `IllegalArgumentException` が送み出されることを確認する。
5. メソッドの本体を実装する。本体が非常に複雑な場合は石橋叩きパターン [Oota 2000] を使用しても良い。ここではそれほど複雑ではないので、そのまま実装する。メソッド完成後、「まだこのメソッドは出来ていない」という例外はコメントアウトする。

```

private static int digit_plus(int a_n) throws IllegalArgumentException {

    // 引数の領域を検査
    if(a_n < 0) {
        throw new IllegalArgumentException("Your input is a_n = " + a_n + ". " +
            "Method digit_plus can accept a_n >= 0.");
    }

    // 各桁の数を加算
    String a_n_string = Integer.toString(a_n);

    int digit_plus = 0;
    for(int a_n_string_index = 0; a_n_string_index < a_n_string.length();
        a_n_string_index++) {
        digit_plus += Character.getNumericValue(a_n_string.charAt(a_n_string_index));
    }
    return digit_plus;

    /*
    throw new RuntimeException("Method digit_plus is not implemented yet.");
    */
}

```

6. 正式なテストを行う。

## A.2 石橋叩きパターンの例

0以上の整数を引数とし、各桁の数を降順に並び替えた数を返すようなメソッド `int down_order(int a_n)` について考える。`int down_order(int a_n)` は0以上の整数 `a_n` に対し `down_order(0) = 0`、`down_order(123) = 321`、`down_order(777) = 777` のように振る舞うものとする。

1. 最初にサンドイッチアプローチパターン [Oota 2000] などにしたがってメソッドの雛型を構築する。

```

static int down_order(int a_n) throws IllegalArgumentException {

    // 引数の領域を検査
    if(a_n < 0) {
        throw new IllegalArgumentException("down_oder Method accepts a_n >= 0.");
    }
    throw new RuntimeException("Method down_order is not implemented yet.");
}

```

```
}
```

2. サンドイッチアプローチパターンにしたがったテストを行う。例えば、次のようなテストスイートを実行する。

```
package circulate_sequence;

import java.lang.*;
import junit.framework.*;

public class Down_Order_Test extends TestCase {
    public Down_Order_Test(String name) {
        super(name);
    }
    public static void main(String[] args) {
        junit.textui.TestRunner.run (suite());
    }
    public static Test suite() {
        return new TestSuite(Down_Order_Test.class);
    }
    public void test0() {
        try {
            Circulate_Sequence.down_order(0);
        }
        catch(RuntimeException e) {
            e.printStackTrace();
        }
    }
    public void test1() {
        try {
            Circulate_Sequence.down_order(1);
        }
        catch(RuntimeException e) {
            e.printStackTrace();
        }
    }
    public void testm1() {
        try {
            Circulate_Sequence.down_order(-1);
        }
        catch(IIllegalArgumentExcepti on e) {
```

```

        e.printStackTrace();
    }
    catch(RuntimeException e) {
        e.printStackTrace();
    }
}
public void test123() {
    try {
        Circulate_Sequence.down_order(123);
    }
    catch(RuntimeException e) {
        e.printStackTrace();
    }
}
}

```

3. メソッドを自分の確信できる単位で書くたびにテストケースを作成し、テストを行う。

```

static int down_order(int a_n) throws IllegalArgumentException {

    // 引数の領域を検査
    if(a_n < 0) {
        throw new IllegalArgumentException("down_oder Method accepts a_n >= 0.");
    }

    // 各数を初期化
    int number[] = new int[10];

    for(int number_index = 0; number_index < number.length; number_index++) {
        number[number_index] = 0;
    }

    // a_n の各桁の数を保存
    String a_n_string = Integer.toString(a_n);

    // テスト用
    System.out.println("For a_n = " + a_n + ", a_n_string = " + a_n_string);

    throw new RuntimeException("Method down_order is not implemented yet.");
}

```

4. 先程と同様のテストケースが使用できる。
5. テストによって書き足したコードに対して確信が得られれば、次のコードを書き、同様に非公式にテストを行っていく。

```
static int down_order(int a_n) throws IllegalArgumentException {

    // 引数の領域を検査
    if(a_n < 0) {
        throw new IllegalArgumentException("down_oder Method accepts a_n >= 0.");
    }

    // 各数を初期化
    int number[] = new int[10];

    for(int number_index = 0; number_index < number.length; number_index++) {
        number[number_index] = 0;
    }

    // a_nの各桁の数を保存
    String a_n_string = Integer.toString(a_n);

    for(int a_n_string_index = 0; a_n_string_index < a_n_string.length();
        a_n_string_index++) {
        number[Character.getNumericValue(a_n_string.charAt(a_n_string_index))]+=;
    }

    // テスト用
    System.out.println("For a_n = " + a_n + ", ");
    for(int number_index = 0; number_index < number.length; number_index++) {
        System.out.println("number[" + number_index + "] = " + number[number_index]);
    }

    throw new RuntimeException("Method down_order is not implemented yet.");

}
```

6. メソッドの完成後は正式なテストを行う。

## 完成したメソッド

```
static int down_order(int a_n) throws IllegalArgumentException {

    // 引数の領域を検査
    if(a_n < 0) {
        throw new IllegalArgumentException("down_oder Method accepts a_n >= 0.");
    }

    // 各数を初期化
    int number[] = new int[10];

    for(int number_index = 0; number_index < number.length; number_index++) {
        number[number_index] = 0;
    }

    // a_nの各桁の数を保存
    String a_n_string = Integer.toString(a_n);

    for(int a_n_string_index = 0; a_n_string_index < a_n_string.length();
        a_n_string_index++) {
        number[Character.getNumericValue(a_n_string.charAt(a_n_string_index))];
    }

    // 降順に変形
    StringBuffer down_order_a_n_string = new StringBuffer();

    for(int number_index = number.length - 1; number_index >= 0; number_index--) {
        while(number[number_index] != 0) {
            down_order_a_n_string.append(number_index);
            number[number_index]--;
        }
    }
    return Integer.parseInt(down_order_a_n_string.toString());
    /*
    throw new RuntimeException("Method down_order is not implemented yet.");
    */
}
```

## 正式なテストケース

```
package circulate_sequence;
```

```

import java.lang.*;
import junit.framework.*;

public class Down_Order_Test extends TestCase {
    public Down_Order_Test(String name) {
        super(name);
    }
    public static void main(String[] args) {
        junit.textui.TestRunner.run (suite());
    }
    public static Test suite() {
        return new TestSuite(Down_Order_Test.class);
    }
    // On ポイント
    public void test0() {
        assert(Circulate_Sequence.down_order(0) == 0);
    }
    // Off ポイント
    public void testm1() {
        try {
            Circulate_Sequence.down_order(-1);
            fail("Should raise an IllegalArgumentException.");
        }
        catch(IllegalArgumentException e) {
            assert(true);
        }
    }
    // 一般値
    public void test1() {
        assert(Circulate_Sequence.down_order(1) == 1);
    }
    // 一般値
    public void test123() {
        assert(Circulate_Sequence.down_order(123) == 321);
    }
    // 一般値
    public void test213() {
        assert(Circulate_Sequence.down_order(213) == 321);
    }
    // 一般値

```

```

public void test321() {
    assert(Circulate_Sequence.down_order(321) == 321);
}
}

```

### A.3 子供のテストパターンの例

ハノイの塔

ハノイの塔のアルゴリズムは良く知られており、その正当性は既に証明されている。しかし、そのアルゴリズムを実装したプログラムが正しいかどうかはテストを行わなければ判明しない。以下の Java プログラムが実装例である。

```

package hanoi;

import java.lang.*;

public class Hanoi {

    public static void main(String[] args) {

        if(args.length != 1) {
            System.out.println("Usage Hanoi disk_number");
            System.exit(1);
        }

        hanoi(Integer.parseInt(args[0]), "a", "b", "c");

    }

    public static void hanoi(int n, String a, String b, String c) {
        if(n < 1) {
            throw new IllegalArgumentException("Method hanoi needs over 1
                for the argument \"int n\".");
        }

        if(n == 1) {
            System.out.println(n + "の円盤を " + a + " から " + b + " に移す");
        }
        else {
            hanoi(n - 1, a, c, b);
            System.out.println(n + "の円盤を " + a + " から " + b + " に移す");
        }
    }
}

```

```

        hanoi (n - 1, c, b, a);
    }
}
}

```

メソッド「hanoi」に対してテストを行う際に可変なのは引数 int n である。テストケースとして用意されるのは、Recursive Function Test パターン [Binder 99] に従えば、On ポイントの  $n = 1$ 、Off ポイントの  $n = 0$ 、一般値の  $n \geq 2$  (これは任意で良い) などである。ここで例えば  $n = 2$  の出力は予想可能であるが、 $n = 6$  となると手作業で正しい結果を予想するのは難しくなる。このような場合、本パターンを適応する。ただし、上記のような有名なアルゴリズムの場合、Trusted System Oracle パターン [Binder 99] を適応できるかもしれない。

実際のテストケースは以下のようなになる。

```

package hanoi;

import java.lang.*;
import junit.framework.*;

public class HanoiTest extends TestCase {
    public HanoiTest(String name) {
        super(name);
    }
    public static void main(String[] args) {
        junit.textui.TestRunner.run (suite());
    }
    public static Test suite() {
        return new TestSuite(HanoiTest.class);
    }
    public void setUp() {
        System.out.println();
    }
    public void tearDown() {
        System.out.println();
    }
    // On ポイント
    public void test1() {
        Hanoi.hanoi (1, "a", "b", "c");
    }
    // Off ポイント
    public void test0() {
        try {
            Hanoi.hanoi (0, "a", "b", "c");
        }
    }
}

```

```
        fail("Should raise an IllegalArgumentException.");
    }
    catch(IllegalArgumentException e) {
        e.printStackTrace();
    }
}
// 一般値
public void test2() {
    Hanoi.hanoi(2, "a", "b", "c");
}
// 一般値「子供のテスト」パターンを適応
public void test6() {
    Hanoi.hanoi(6, "a", "b", "c");
}
}
```

## 参考文献

[Alexander 75] Christopher Alexander. The Oregon Experiment. New York: Oxford University Press, 1975. (宮本雅明訳、オレゴン大学の実験、鹿島出版会)

[Alexander+77] Christopher Alexander, Sara Ishikawa, Murray Siverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language. Oxford University Press, New York, 1977. (平田翰那訳、パターンランゲージ、鹿島出版会)

[Alexander 79] Christopher Alexander. The timeless way of building. New York: Oxford University Press, 1979. (平田翰那訳、時を越えた建設の道、鹿島出版会)

[Beizer 90] Boris Beizer. Software testing techniques, 2nd ed. New York: International Thompson Computer Press, 1990. (小野間彰、山浦恒央訳、ソフトウェアテスト技法、日経BP出版センター)

[Beizer 95] Boris Beizer. Black box testing. New York: John Wiley & Sons, 1995. (実践的プログラムテスト入門、日経BP出版センター)

[Binder 95b] Robert V. Binder. Testing objects: myth and reality. Object Magazine 5(2):73-75, May 1995.

[Binder 99] Robert V. Binder. Testing Object-Oriented Systems Models Patterns, and Tools. Reading, Mass: Addison-Wesley, 1999.

[Brooks 75] Frederick P. Brooks, Jr. The mythical man-month: essays on software engineering. Reading, Mass.: Addison-Wesley, 1975. (滝沢徹、牧野祐子、富沢登訳、人月の神話、アジソンウェスレイ)

[Brown+98] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick III, Thomas J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. New York: John Wiley & Sons, 1998. (岩谷宏訳、アンチパターン ソフトウェア危篤患者の救出、ソフトバンク)

[DeLano+96] David E. DeLano and Linda Rising. "Patterns for System Testing" In Robert Martin, Dirk Riehle and Frank Buschmann(eds). Pattern Languages of Program Design 3. Reading, MA: Addison-Wesley, 1996, pp. 503-525.

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of reusable object-oriented software. Reading, Mass.: Addison-Wesley,

1995. (本位田真一、吉田和樹監訳、オブジェクト指向における再利用のためのデザインパターン、ソフトバンク)

[Gamma+98] Erich Gamma and Kent Beck. Test infected. programmers love writing test. The Java Report 3(7):37-50, July 1998.

[Humphrey 90] Watt S. Humphrey. Managing the Software Process. Reading, MA: Addison-Wesley, 1990. (藤野喜一監訳、ソフトウェアプロセス成熟度の改善、日科技連)

[Jones 97a] Capers Jones. Software quality: analysis and guidelines for success. London: International Thompson Computer Press, 1997. (富野壽監訳、ソフトウェア品質のガイドライン、共立出版)

[Kaner+93] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. Testing computer software. 2nd ed. New York: Van Nostrand Reinhold, 1993.

[McConnell 93] Steve McConnell. CODE COMPLETE. Microsoft Press, a division of Microsoft Coporation, Redmond, Washington, U.S.A. 1993. (石川勝訳、コードコンプリート、アスキー出版局)

[Meszaros+96] Gerard Meszaros and Jim Doble. "Pattern Language for Pattern Writing" In Robert Martin, Dirk Riehle and Frank Buschmann(eds). Pattern Languages of Program Design 3. Reading, MA: Addison-Wesley, 1996, pp. 529-574

[Myers 79] Glennford J. Myers. The art of software testing. New York: John Wiley & Sons, 1979. (松尾正信訳、ソフトウェア・テストの技法、近代科学社)