修 士 論 文

形式仕様に基づくテストケース生成の有効性に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

中島 亮平

2008 年 3 月

修 士 論 文

形式仕様に基づくテストケース生成の有効性に関する研究

指導教官　二木厚吉 教授

審査委員主査　二木 厚吉 教授
審査委員　Rene Vestergaard 准教授
審査委員　緒方 和博 特任准教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

610063 中島 亮平

提出年月: 2008 年 2 月

概 要

本論文では形式仕様に基づきテストケースを生成し、それも基にテスト駆動開発を行う手法を提案する。形式仕様には CafeOBJ で記述された観測遷移機械モデルを用い、それを基に JUnit で実行可能なテストケースを生成する。提案手法を支援するツールを作成し、それを用いて開発を行った結果仕様に沿ったシステムが開発できることを確認した。

目 次

第1章　はじめに

1.1　背景・目的

テスト駆動開発はプログラムの実装を行う前に、テストケースを記述し、そのテストケースがパスするような実装を行う。このテストケースを開発のベースと考え、現在の実装をパスしないテストケースを記述する、そのテストケースをパスするように実装を行うという作業を繰り返して実装を進めていく。テスト駆動開発の利点はテストケースが仕様そのものであると考えられるため、それをパスすれば仕様を満たしているみなせる。しかしテスト駆動開発ではベースとなるテストケースを開発者が手作業で作成することが多い。そのため、平均してテスト対象コードの 1.5〜2 倍以上にもなるテストコードの記述は、開発者にとって大きな負担となる。またテストケースの作成は開発者の個人の技量に依存するため、テストケースが仕様を満たしていることを保証できない、またテスト漏れがある可能性があるなどの問題がある。これに対して形式仕様に基づきテストケースを生成し、それも基にテスト駆動開発を行う手法を提案する。テスト駆動開発に形式仕様を導入することで、上流工程の段階で仕様の検証が可能となり、仕様の ” 曖昧さ ” を早期発見・排除が可能となる。さらに検証された仕様からテストケースを自動生成することで、仕様を満たしたテストケースの取得やテスト漏れの減少、テスト作成にかかるコストの削減といった効果が期待できる。

第2章　準備

2.1　形式仕様

形式仕様とは、仕様記述の段階で、文法・意味を厳密な数学モデルや論理体系に基づいて定義された形式仕様言語 (Formal Specification Language) を用いて表現された仕様のことである。形式仕様言語で記述された仕様は、仕様自体の機械処理が可能であり、上流工程の時点で機械的な解析・検証も可能なため、より信頼性の高い仕様を作成することができる。形式仕様に求められる性質には以下のものが挙げられる。

厳密性 仕様に曖昧な部分がない。

非冗長性 仕様に冗長な部分がない。

健全性 仕様から検証できることは常に正しい。

形式仕様言語には、多種多様な数学モデル・数理体系に基づく言語が提案されている。代表的なものとして、公理的記述法を用いた Z や VDM、操作的記述法を用いた Estell、そして代数的記述法を用いた OBJ,CafeOBJ,Maude 等がある。ここでは代数的手法に基づき仕様の機械的検証が可能である CafeOBJ について取り上げる。

2.1.1　代数仕様記述言語 CafeOBJ

CafeOBJ は、代数仕様言語 OBJ の流れを汲む実行可能な代数仕様言語である。CafeOBJ には複数の論理の組み合わせを仕様の意味モデルとして持つことができるという特徴があり、その論理として多ソート代数 (Many Sorted Algebra)、順序ソート代数 (Order Sorted Algebra)、隠蔽代数 (Hidden Algebra)、書き換え論理 (Rewriting Logic) を持つ。これらの論理を組み合わせることで、開発対象のシステムに応じた意味モデルの選択が可能となる。また、CafeOBJ は項書き換えシステムをその操作的意味論として持っている。従って、等式論理、書き換え論理の推論規則を用いて定理の証明を行うことができる。CafeOBJ は、以上のような特徴により、他の形式仕様言語と比較して以下のような優位点を持つ。

- モジュールのパラメータ化、モジュールの輸入の際の名前の付け替えが可能で、強力なモジュール化機構を持ち、抽象度、再利用性の高い仕様記述が可能である。

- 項書き換えシステムにより、仕様をそのまま機械的に実行可能であり、仕様記述段階でのプロトタイピングが可能である。
- 順序ソート代数に基づき、ソート間に包含関係が定義でき、例外処理や演算の多重継承を自然に扱える。
- 動的なシステムの変化を記述できる書き換え規則を持つ。
- クラス構文を持ち、オブジェクト指向モデルの記述にも対応している
- Hidden Algebra に基づいた記述手法が可能である。

CafeOBJ の構文

CafeOBJ では代数仕様 (Algebraic Specification) と呼ばれる仕様を記述する。代数仕様はソート (Sort) とソート上での演算 (Operation) で構成される指標 (Signature) により言語を定義し、等式と変数から構成される公理 (axiom) によって、2 項間の等価関係を定義する。CafeOBJ では、指標と公理から構成されるモジュール (module) を 1 つの単位として仕様を記述する。本節では、CafeOBJ の各構文について簡単に説明する。

モジュール宣言

モジュールの宣言には”mod” を用いる。

```
mod module_name{
  module_element*
}
```

module_name にはモジュール名を、*module_element* にはモジュールの各構成要素を記述する。モジュールの構成要素は輸入 (import)、ソート、演算子、変数および等式の各宣言である。モジュールを宣言する際、モジュールの意味論を指定することができる。きつい (tight) 意味論に基づいて記述するときは”mod!”、ゆるい (loose) 意味論の場合は”`mod*`” と記述する。意味論を指定しない場合には”mod” を用いる。

輸入宣言

輸入宣言には、“pr” 又は”ex”,”us” を用いる。

```
pr ( module expression )
```

module_expression には輸入するモジュール名を記述する。モジュール M がモジュール M' を輸入するとは、M' の定義が M で参照可能になることを意味する。それぞれの宣言の違いを以下に示す。

- pr(protecting)：輸入されるモジュール (M') で宣言されたソートに新たな要素を付け加えたり（非冗長)、ソートの既存の要素を同定したり（非混同）はできない。
- ex(extending)：輸入されるモジュール (M') で宣言されたソートの既存の要素を同定してはならない（非混同)。
- us(using)：制限なし。

また、すべてのモジュールにおいて組み込みモジュールBOOL が暗黙のうちに輸入される。BOOL では真理値と論理演算について定義されている。

ソート宣言

CafeOBJ には、可視ソート (visible sort) と隠蔽ソート (hidden sort) の 2 種類のソートがある。可視ソートは始代数に基づく抽象データ型を、隠蔽ソートは隠蔽代数に基づく抽象機械の状態空間を表すのに用いる。可視ソートV 、隠蔽ソートH はそれぞれ以下のように宣言する

```
[ V ]
*[ H ]*
```

ソート間に順序関係を持たせる場合、次のように宣言する。

```
[V' < V ]
```

このとき、V' は V のサブソートであると言い、V' の持つ全ての項は V の項でもあることを意味する。これを順序ソート (Order Sort) と呼ぶ。順序ソートを導入することで、演算のオーバーロードが明確に定義される。

演算子宣言

演算子は以下のように宣言する。

```
op operation_name : arity -> coarity
```

operation_name が引数にとるソートの組をアリティ(arity) と呼び、演算子が持つソートをコアリティ(coarity) と呼ぶ。アリティに何も持たない場合、その演算子は定数を表している。また演算子中に'_'（下線）がある場合、その位置にアリティを記述することを表している。隠蔽ソートをアリティに持つ場合は、“bop” で宣言する。

変数宣言

宣言された変数は等式で用いられ、指定されたソートの項が代入される。ソート V 上の変数 v は以下のように宣言する。

`var` v : V

同じソートを持つ変数 $v1$,$v2$,... を宣言する場合、"`vars`" を用いる。変数の有効範囲は、宣言されたモジュール内のみである。

等式宣言

等式は、左辺の項 (lhs) と右辺の項 (rhs) の間の等価関係を宣言する。

`eq` lhs `=` rhs .

等式を記述する際には、式の最後に ' . '（ピリオド）が入ることに注意する。条件付き等式の場合、"`ceq`" を用いて宣言する。*condition* は BOOL ソート上の項である。

`ceq` lhs `=` rhs `if` *condition* .

2.1.2 観測遷移システム OTS

観測遷移機械 (OTS:Observational Transition System)[2] は、並行システムや分散システムを状態機械としてモデル化できる。状態機械は、これらのシステムの抽象モデルとして広く用いられているが、OTS はモデル化するシステムをブラックボックスとして捉え、その外部から観測できる情報だけに基づいて、状態機械の性質について議論できる。このことは、内部の実装に依存しない、抽象度の高いモデル化を可能としている。

OTS でモデル化したシステム S では、その任意の状態を包含する状態空間 Υ の存在を仮定する。またシステムの任意の状態はの要素 u として必ず存在している。2 つの状態 $u1, u2 \in \Upsilon$ の等価性は、各観測から得られる値（観測値）の組で識別できるとし、Υ 上の述語 $= S$ で表現する。

$$u_1 =_S \tau u_2 \ \mathit{iff} \ \forall o \in \mathcal{O}.o(u_1) = o(u_2)$$

ただし、各観測の返り値の型 D について、その元 $d_1, d_2 \in D$ の等価性を判定する D 上の述語 $=$ が適切に与えられているものと仮定する。OTS $\mathcal{S}$ は観測の集合 $\mathcal{O}$、初期条件 $\mathcal{I}$、遷移規則の集合 $\mathcal{T}$ の組 $(\mathcal{O}, \mathcal{I}, \mathcal{T})$ で定義される。

- $\mathcal{O}$: 観測
  外部から Υ を観測する関数の集合。各観測 $o \in \mathcal{O}$ は、状態を引数に取り、その状態を構成する任意のデータ型 D を返す関数 $o : \Upsilon \to D$ である。ただし、各 o の値域は、それぞれ異なってもよい。

- $\mathcal{I}$：初期状態
  初期状態の集合。$\mathcal{O}$ は $o \in \mathcal{O}$ の初期値を決め、$\mathcal{I} \subset \Upsilon$ である。

- $\mathcal{T}$：遷移規則
  状態を遷移させる関数の集合。各 $\tau \in \mathcal{T}$ は、ある状態を引数に取り、次の状態を返す関数 $\tau : \Upsilon \to \Upsilon$ である。τ は、効果と効力条件の組で定義する。効果は、τ による各 $o \in \mathcal{O}$ の変化であり、効力条件は、述語 $c_\tau : \Upsilon \to \{true, false\}$ である。いま、2 つの状態 $u_1, u_2 \in \Upsilon$(ただし、$u_1 \neq s\ u_2$) を仮定する。ある遷移規則 τ_1 について、$\tau_1 : u_1 \to u_2$ であるとしたとき、以下の関係を満たす。

  $$(c_{\tau_1}(u_1) = true) \Rightarrow (u_2 = s\tau_1(u_1)) \cap (u_1 \neq s\ \tau_1(u_1))$$
  $$(c_{\tau_1}(u_1) = false) \Rightarrow (u_2 \neq s\tau_1(u_1)) \cap (u_1 = s\ \tau_1(u_1))$$

  これは、c_τ が真であるとき、各 $o \in \mathcal{O}$ の値は τ の効果で定義した値に変化し、c_τ が偽であるなら、o の値は変化しないことを意味する。

OTS $\mathcal{S}$ の実行は、初期状態から始まり、実行の各ステップで遷移規則の 1 つを非決定的に選択し適用する。この実行から、状態の無限列の集合が得られる。$\mathcal{S}$ の実行は、以下の 3 つの条件を満たす状態の無限列 u_0,u_1, ... である。

- 開始性:状態 u_0 において、各 $o \in \mathcal{O}$ は $\mathcal{I}$ を満たす。
- 一貫性:すべての $i \in \{0, 1, 2, \ldots\}$ において、$u_{i+1} = s\tau(u_i)$ となる $\tau \in \mathcal{T}$ が存在する。
- 公平性:各 $\tau \in \mathcal{T}$ に対し、$u_{i+1} = s\tau(u_i)$ を満たす $i \in \{0, 1, 2, \ldots\}$ が無限に存在する。
  また、ある状態 $u \in \Upsilon$ が $\mathcal{S}$ の実行によって現れるとき、u は到達可能であるという。

OTS/CafeOBJ 仕様の構文

前節で定義された OTS のモデルを CafeOBJ で記述する。OTS/CafeOBJ 仕様は以下の構文で表現される。このとき、各データ型 $D, D_k(k = i_1, \ldots, i_m, j_1, \ldots, j_n)$ が可視ソート $V, V_k(k = i_1, \ldots, i_m, j_1, \ldots, j_n)$ 上に適切に定義されていると仮定する。

状態空間 Υ

状態空間 Υ は隠蔽ソートで表現する。ソート名を H としたとき、`*[ H ]*`と記述する。

観測の集合 $\mathcal{O}$

各観測 $o_{i_1 \ldots i_m} \in \mathcal{O}$ は CafeOBJ の観測演算 (Observation) で与えられる。観測 o に対応する観測演算は次のように宣言する。

`bop` o `:` $H\ V_{i1} \ldots V_{im}$ `->` V

初期状態の集合 $\mathcal{I}$

初期状態は、初期状態を表す定数 `init` を宣言し、各観測値を等式で宣言する。

```
op init :  -> H
```

観測 o_i の初期値が $f(i)$ で与えられると仮定すると、観測値は以下のように記述できる。

```
eq o(init,X_i1,...,X_im) = f(X_i1,...,X_im) .
```

ここで、$X_k(k = i_1, \ldots, i_m)$ は V_k に対応する CafeOBJ 変数の列であり、$f(X_k)$ は $f(k)$ に対応する項である。

遷移規則の集合 $\mathcal{T}$

$\mathcal{S}$ の遷移規則 τ j_1,...,j_n は作用演算 (Action) で表現される。遷移規則に対応した作用演算は次のように記述される。

```
bop τ :  H V_j1 ... V_jn -> H
```

効力条件が真である状態における、遷移規則 τ の適用後の o の観測値の効果は以下の等式で記述される。

```
ceq o(τ(W,X_j1,...,X_jn),X_i1,...,X_im) = e − τ(W,X_i1,...,X_im)
      if c − τ(W,X_j1,...,X_jn)
```

ここで W はソート H の CafeOBJ 変数であり、$e - \tau(W,X_{i1},\ldots,X_{im})$ は遷移規則が効力条件を満たした場合の観測 o の効果に対応する CafeOBJ の項を表す。$c - \tau(W,X_{j1},\ldots,X_{jn})$ は遷移規則の効力条件である。効力条件は以下のように記述する。

```
bop c − τ :  H V_ji ... V_jn -> Bool
c − τ(W,X_j1,ldots,X_jn) = conditionalterms .
```

効力条件が偽の場合は、どの観測値も変化しないので以下のように記述する。

```
ceq τ(W,X_j1,...,X_jn) = W if not c − τ(W,X_j1,ldots,X_jn) .
```

2.2　テスト駆動開発

テスト駆動開発 (TDD:Test-Driven Development) は Kent Beck が提唱したアジャイル型開発の 1 つである。TDD の大きな特徴は、プログラムの実装に入る前に、先にテストケースを書くという開発スタイルであり、このスタイルはテストファーストとも呼ばれる。テストケースとは、「この処理や操作を実行したらこういう結果になる」という観点から、入力データ・実施する処理の内容・期待する結果をセットにしたものを指す。TDD では次のようなサイクルで開発が進められる。

1. 仕様を元に、テストケースを 1 つだけ作成（追加）する。
2. 作成したテストケースをパスする、最も簡潔な方法でプログラムを実装する。
3. テストを実行し、成功することを確認する。確認後、プログラムの動作・振る舞いを変えることなく、内部構造を整理（リファクタリング）する。
4. 1-3 を繰り返す。
5. 実装を変更した場合、仕様を変更していないことを確認するため、これまで実施されたテストを実行する。
6. 仕様を変更した場合、その変更をテストケースに反映させる。

このように、TDD ではテストとコーディングを繰り返し行い、そのフィードバックによるプログラムの修正、再設計を経て、開発を進めていく。TDD ではテストケースを元に開発を進めることになるが、ここで用いられるテストとは、モジュールの振る舞いが仕様どおりかをチェックするテスト、つまりモジュールに対する機能テストのことである。機能テストはブラックボックステスト、動作テストとも呼ばれ、以下のように定義されている。プログラムやシステムを 1 個のブラックボックスとして扱い、内部構造には注目することなく、入力データに対する出力結果が要求仕様どおりかどうかを検証する。

2.2.1　テスト駆動開発の利点と問題点

TDD では、先にテストケースを記述するという性質上、テストケースを書き終わった段階で、これから書こうとするプログラムの仕様が明確に意識できるようになる。そのため、最適な実装を開発の初期段階から行うことができるという利点がある。また、実装を変更するたびにテストを実施するため、仕様を変更せずに実装を変えたつもりでも不注意で動作がおかしくなった場合でも、発見が容易になる。なお、テストケースは仕様書の代用としても機能する。自然言語で書かれた仕様書では、プログラミング言語ほどの厳密性は望めないうえ、プログラムがその仕様を満たしているという保証もない。しかし、テストケースにはプログラムの期待する動作が正確に記述されているため、それをパスすれ

ば仕様を満たしていることも保証される。このような利点を持つTDDだが、決して万能な開発方法というわけではない。開発のベースとなるテストケースは、開発者個人が仕様書を元に作ることが多い。その場合、平均してテスト対象コードの1.5〜2倍以上にもなると言われるテストコードの記述は負担の増大となる。また作成されるテストケースは、開発者個人の技量に依存するため、テストケースが仕様を満たしているか、テスト漏れはないかといった品質的な問題が付きまとう。またTDDでは、自動テストツールを用いてテストを何度も繰り返すことが前提となっている。そのため、テストツールで自動テストを実施できないシステム、例えばグラフィカルユーザインタフェース(GUI)を扱うものや1つの動作に多大な計算機リソースを消費するものに対しては、TDDは適さない。しかしこれらはTDDの問題というよりもテストツールの問題であり、いずれは解決されるであろう問題である。その他の問題として、テストケースをパスすれば完璧なプログラムが作成できたと考えてしまうことがある。しかしTDDはあくまで開発手法であり、テスト手法などではない。使用されるテストは機能性を確かめるテストであるため、信頼性や効率性などの品質についても従来通りのテストを行わなくてはならない。

2.3 テスティングフレームワーク JUnit

TDDでは頻繁にテストの実行が行われるため、テストを自動的に実行できる環境が必要である。そのような環境を提供するツールをxUnitと呼ぶ。xUnitとは、プログラムをテストするテスティングフレームワークの総称であり、各プログラミング言語毎に用意されている。xUnitの1つであるJUnitは、1997年にKent BeckとEric Gammaによって開発されたものであり、Javaで開発されたプログラムにおいてユニットテスト（単体テスト）の自動化を行うためのフレームワークである。JUnitは以下のような特徴・利点を持つ。一度テストを作成すれば、何度でもテストを実行することができる。また、似たような機能のテストを行うときでも、テストプログラムのコードを再利用することができる。

- テストを自動で実行でき、テスト結果のチェックも自動的に行われる。そのため、即座にフィードバックを得ることができる。
- テストが成功すれば緑色、失敗すれば赤色のプログレスバーが表示されるため、視覚的にチェックすることができる。
- (Kent Beck曰く)即座にテスト結果が分かるため、精神的に安定した状態で作業を進めることができる。
- JUnitとはJavaで開発されたプログラムにおいてユニットテスト(単体テスト)の自動化を行うためのフレームワークである。

2.3.1 JUnit の文法

テストケース生成ツールでは JUnit4 に対応したテストケースを出力する。JUnit4 の文法を解説する、また JUnit4 では以前のバージョンから大きな変更があったため、それについても説明する

アノテーション

@Test そのメソッドがテストメソッドであることを示す。このメソッドにテストを記述する。従来の JUnit でメソッド名が test で始まるメソッドと同じ。

@Before このアノテーションが付加されたメソッドは、@Test アノテーションが付いたメソッドを実行するたびに事前に実行されることを意味する。JUnit4 以前の setup() メソッドと同じ。

@After このアノテーションが付加されたメソッドは、@Test アノテーションが付いたメソッドを実行するたびに、必ず後から実行されることを意味する。JUnit4 以前の tearDown() メソッドと同じ。

パッケージのインポートと TestCase の継承

JUnit 3.8 ではテストケースクラスでは TestCase を必ず継承する必要があったが JUnit4 では継承する必要がない。しかし TestCase クラスを継承しない場合 Assert クラスの static メソッドをメソッド名だけで呼び出すことができないため Assert.assertEquals() のように使わなければならない。これを避けるためには org.junit.Assert を static import すればよい。

用意されているメソッド

JUnit ではテストを実施するために多くのクラス、メソッドが用意されている。その中でテストケース生成ツールに使用したメソッドについて説明する。

assertEquals(Object expected,Object actual) 二つのオブジェクトが等しか判定する

fail(String message) テストを失敗させる、その際メッセージを表示する。

第3章　形式仕様によるテストケース生成

本章では実装を行うためのクラスの枠組み（スケルトンクラス）を生成する方法、OTS/CafeOBJ 仕様からテストケースを生成する規則について述べる。

3.1　OTS/CafeOBJ 仕様と Java クラスの対応

本節では、OTS/CafeOBJ 仕様と Java クラスの対応関係について述べる。OTS ではシステムをブラックボックスとみなし、外部からシステムの情報を得る観測の集合 O、システムの初期状態の集合 I、システムの状態を変化させる遷移の集合 T の組で表現する。この OTS モデルを代数仕様言語 CafeOBJ で記述したものが OTS/CafeOBJ 仕様である。表 3.1 に OTS/CafeOBJ と Java の対応を示す。

3.2　スケルトンクラスへの変換

クラスのメソッドの実装は TDD によって行うが、クラスのメソッド宣言は OTS/CafeOBJ 仕様から自動で生成することができる。ここではクラスとメソッドに変換するルールを説明する。スケルトンクラスはシステム仕様の隠蔽ソートと振舞演算のみから生成する。

OTS/CafeOBJ 仕様		Java
データ仕様	ソート	クラス（データ）
	演算子	クラスメソッド（演算子）
システム仕様	隠蔽ソート	クラス（システム）
	観測演算	メソッド
	遷移演算	メソッド
	初期状態	コンストラクタ（システム）

表 3.1: OTS/CafeOBJ と Java の対応

仕様の隠蔽ソートとシグネチャ

mod* $moduleName${
[H]
bop O_1 : $V_{O_{11}}$ $\cdots$ $V_{O_{1n}}$ -> V_{O_1}
$\vdots$
bop T_1 : $V_{T_{11}}$ $\cdots$ $V_{T_{1m}}$ -> H
$\vdots$
}

スケルトンクラス

public H {
  public H(){}
  public V_{O_1} O_1($V_{O_{11}}$ arg0 , $\cdots$, $V_{O_{1m}}$ arg<n-1>){}
$\vdots$
  public void T_1($V_{T_{11}}$ arg0 , $\cdots$, $V_{T_{1m}}$ arg<n-1>){}
$\vdots$

}

テストケースを分割する

3.3 テストケース生成規則

3.3.1 テストケースの構成

初期化事前状態の設定事前状態の観測値の保存事前状態のチェック遷移関数を実行テストを実行 (assert)

3.3.2 初期化

テストケースの初期化部分ではシステムクラスのインスタンスの生成と、システムクラスのメソッドの引数になるクラス（変数）のインスタンスの生成を行う。

共通の構成

変数定義部分では観測メソッドと遷移メソッドで使用されるすべてのオブジェクトの宣言、初期化を行う。しかしクラスによって初期化方法が違うため、また準備部分でもあるため有効な初期化のコードを自動で生成することは難しい、そのため宣言のみ生成し、初期化はユーザーが行う。

シグネチャが
T : V_{T_1} ... V_{T_n} -> V_T
O : V_{O_1} ... V_{O_m} -> V_O

等式が
eq $O(T(S, X_{T_1}, \ldots, X_{T_n}), X_{O_1}, \ldots, X_{O_m}) =$
eq $O(T(S, \mathbf{X_i}), \mathbf{X_j}) = f(W, \mathbf{X_i}, \mathbf{X_j})$.

```
public void TestCaseName{
  //システムクラスのインスタンス生成
  H instance = new H();

  //初期化
  //遷移関数に使用される変数
  V_{T_1} X_{T_1};
    ⋮
  V_{T_n} X_{T_n};
  //観測関数に使用される変数
  V_{O_1} X_{O_1};
    ⋮
  V_{O_m} X_{O_m};

  //事前状態設定

  //テストに用いる事前状態の観測値を保存する
  V_O before =
  //事前状態のチェック
  instance.T(X_{T_1},...,X_{T_n});
  assertEquals(before,instance.O(X_{O_1},...,X_{O_m}));
}
```

3.3.3 条件部のない等式

基本的な条件部の無い等式からのテストケース生成方法について述べる。初期化部分の生成方法は3.3.2で書いたとおりのためこ言及しない。

```
eq O(T(S, X_T1, ..., X_Tn), X_O1, ..., X_Om) = rhs .
```

```
V_O before = RHS
instance.T(X_T1,...,X_Tn);
assertEquals(before,instance.O(X_O1,...,X_Om));
```

3.3.4 条件付き等式

```
eq O(T(S, X_T1, ..., X_Tn), X_O1, ..., X_Om) = rhs if condition .
```

```
V_O before = RHS
if(condition){
  instance.T(X_T1,...,X_Tn);
  assertEquals(before,instance.O(X_O1,...,X_Om));
}
```

3.3.5 右辺にif文を含む場合

CafeOBJのif構文は以下のように記述する。

```
if condition then term_T else term_F fi
```

これは*condition*がtrueの場合$term_T$がfalseの場合$term_F$が選択されることを意味する。

そして右辺にif文を含む等式は以下のような等式である。

```
eq O(T(S, X_T1, ..., X_Tn), X_O1, ..., X_Om)
  = (if condition then term_T else term_F fi) .
```

if文を含まない場合にはひとつの等式に対してひとつのテストケースを生成していたが、等式の右辺にif文を含む場合、conditionがtrue,false二つの場合に対するテストケースを生成する。

if の *condition* が true の場合のテストケースは事前状態チェックでは *condition* が true のときにテストを行うようにし、事前状態の観測値 before には CafeOBJ の等式で *condition* が true の場合に選択される $term_T$ を設定する。false の場合、*condition* が false のときにテストを行うようにし、before には CafeOBJ の等式で *condition* が false の場合に選択される $term_F$ を設定する。

生成される二つのテストケース

if の condition が true の場合に対するテストケース

```
before = term_T;
if(condition){
  instance.T(X_T1,...,X_Tn);
  assertEquals(before,instance.O(X_O1,...,X_Om));
}
```

if の condition が false の場合に対するテストケース

```
before = term_F;
if(!condition){
  instance.T(X_T1,...,X_Tn);
  assertEquals(before,instance.O(X_O1,...,X_Om));
}
```

3.3.6 組み合わせ

これまでに条件部の無い等式、条件付き等式、右辺に if 文を含む等式についてどうテストケースを生成するか説明をしてきたが、これらは単純な例であり実際にはこれらの組み合わせが現れる、ここでは条件付き等式で右辺に if 文を含む場合、右辺に if 文が複数ある場合について説明する。

条件付き等式で右辺に if 文を含む場合

この場合は以下のような等式である。

```
eq O(T(S,X_T1,...,X_Tn),X_O1,...,X_Om)
  = (if condition_1 then term_T else term_F fi)
if condition .
```

右辺に if 文が複数ある場合

この場合は以下のような等式である。

```
eq O(T(S, X_{T_1}, ..., X_{T_n}), X_{O_1}, ..., X_{O_m}) =
  (if condition_1 then
    (if condition_2 then term_{TT} else term_{TF} fi)
  else
    term_F fi)
```

3.3.7 init を含む場合

定数 init はシステムの初期状態を表す定数である。OTS/CafeOBJ 仕様では

```
eq O(init, X_{O_1}, ..., X_{O_m})) = rhs .
```

のように使われ初期状態の観測値を定義するために使う。これをテストケースに変換した場合、システムのクラスを生成した直後に観測を行った場合に相当すると考えられる。そのため以下のようなテストケースを生成する。

```
assertEquals(rhs,instance.O(X_{O_1},...,X_{O_m}));
```

第4章　テストケース生成ツール

第3章ではテストケース生成のアルゴリズムを説明を行ったが、本章では、それを実現するツールの実装について説明する。

4.1　ツールの概要

実装にはJava言語を用いた。すでにOTS/CafeOBJのモジュールをXML形式に変換するソフトウェアBuffetが存在するため、今回作成したツールはBuffetが生成したXML形式のOTS/CafeOBJを読み込み、それを元にテストケース、クラスの枠組みを生成するツールである。XML形式のデータの処理にはDOMを用いた。主な機能を以下に示す。

- テストケース生成
- 関数変換、型変換
- 実装クラスの骨組みを出力する

4.1.1　XMLへの変換

BuffetはOTS/CafeOBJ仕様をXML形式に変換するソフトウェアである。テストケース生成ツールではBuffetの出力したXML形式のOTS/CafeOBJを入力としてテストケースを生成する。テストケース生成ツールを使用するためにはBuffetを使う必要があるため、

4.1.2　ツールの構成

ここではツールを構成しているクラスの説明を行う。表4.1に示す。

4.1.3　処理の流れ

ここではクラスを用いてどのよううな流れ出処理が行われるか説明する。

TestCaseGenerator	メインクラス、他のクラスを生成し利用する。
SkeltonClassGenerator	スケルトンクラスを生成する。
TestCaseXML	テストケースの元になるデータを XML 形式で保持する。
TestCaseString	ひとつのテストケースの元になるデータを String 形式で保持する。それぞれのテストケースを生成する
TestCaseStringList	TestCaseString と全体のテストケースを生成するのに必要なデータを持つ、テストケースクラスを生成する。
TypeConverter	型、関数変換ファイルを読み込みそれを元に型を変換する。

表 4.1: テストケース生成器を構成するクラス

共通

XML 形式の OTS/CafeOBJ 仕様を DOM の parser で読み込む。

スケルトンクラス

1. シグネチャ、隠蔽ソートを読み込む。
2. SkeltonClassGenerator は TypeConverter を利用し型を変換しながらスケルトンクラスを出力する

テストケース

1. テストケース生成に必要なデータを TestCaseXML に格納する。その際 if_then_else_fi でテストケースの分割を行う。
2. TestCaseXML を文字列に変換し TestCaseString に格納する。
3. TestCaseString は TestCaseStringList に追加する
4. TestCaseList はテストケースクラスを生成する、その際メソッドは個々の TestCaseString に任せる

4.1.4 関数変換、型変換

テストケース生成、スケルトンクラス生成については前章で説明したそれぞれの生成ルールにしたがって行うが、Java と CafeOBJ の組み込み型は対応していないため、実装クラスの骨組みとテストケースを生成する際、型と変換する必要がある。この操作は TypeConverter クラスが行う。

4.1.5 実行方法

本節ではCafeOBJのモジュールファイルからテストケースを生成するまでの操作を説明する。
atm.modというファイルがが存在て、それが隠蔽ソート名がAtmであるモジュールATMを含むとすると。

ターミナルで

```
>buffet
```

別のターミナルで

```
>simple atm.mod ATM > atm.xml
>java -classpath "bin_folder" nakajima.TestCaseGenerator atm.xml
```

この結果Atm.javaとAtmTest.javaが生成される。simpleの結果を格納するファイル名はatm.xmlである必要はなく、ユーザーが任意に決めればよい。

4.2 変換の例

ツールを用いテストケース生成を行った例を示す。

CafeOBJ モジュール ATM,USER

```
mod* USER{
    [ User ]
    op _=_ : User User -> Bool { comm }
    eq (U:User = U) = true .
}

mod* ATM{
    pr(INT + USER)
    *[ Atm ]*
    op init : -> Atm
    bop balance : Atm User -> Int
    bop deposit : Atm User Int -> Atm
    bop withdraw : Atm User Int -> Atm
    vars U U2 : User
    var I : Int
    var A : Atm
```

```
    eq balance(init,U) = 0 .
    ceq balance(deposit(A,U2,I),U) =
            (if U = U2 then I + balance(A,U)
            else balance(A,U) fi)
        if     I >= 0 .
    ceq deposit(A,U2,I) = A
        if not (I >= 0) .
    ceq balance(withdraw(A,U2,I),U) =
            (if U = U2 then balance(A,U) - I
            else balance(A,U) fi)
        if balance(A,U) >= I .
    ceq withdraw(A,U2,I) = A
        if not(balance(A,U) >= I) .
}
```

生成したJUnitテストファイル AtmTest.java

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import org.junit.Test;

public class AtmTest{

@Test
public void  test_init_balance(){
    Atm atm = new Atm();
    User U;
    assertEquals(0,atm.balance(U));
}

@Test
public void  test_deposit_balance_0(){
    Atm atm = new Atm();
    User U2;
    int I;
    User U;
    int before1=atm.balance(U);
    if(U.equals(U2)){
        if(((I>=0))){
```

```
            atm.deposit(U2,I);
            assertEquals((I+before1),atm.balance(U));
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_deposit_balance_1(){
    Atm atm = new Atm();
    User U2;
    int I;
    User U;
    int before1=atm.balance(U);
    if(!(U.equals(U2))){
        if(((I>=0))){
            atm.deposit(U2,I);
            assertEquals(before1,atm.balance(U));
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_deposit_balance_2(){
    Atm atm = new Atm();
    User U2;
    int I;
```

```
    User U;
    int before1=atm.balance(U);
    if((!((I>=0)))){
        atm.deposit(U2,I);
        assertEquals(before1,atm.balance(U));
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_withdraw_balance_0(){
    Atm atm = new Atm();
    User U2;
    int I;
    User U;
    int before1=atm.balance(U);
    if(U.equals(U2)){
        if(((atm.balance(U)>=I))){
            atm.withdraw(U2,I);
            assertEquals((before1-I),atm.balance(U));
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_withdraw_balance_1(){
    Atm atm = new Atm();
    User U2;
    int I;
    User U;
```

```
    int before1=atm.balance(U);
    if(!(U.equals(U2))){
        if(((atm.balance(U)>=I))){
            atm.withdraw(U2,I);
            assertEquals(before1,atm.balance(U));
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_withdraw_balance_2(){
    Atm atm = new Atm();
    User U2;
    int I;
    User U;
    int before1=atm.balance(U);
    if((!((atm.balance(U)>=I)))){
        atm.withdraw(U2,I);
        assertEquals(before1,atm.balance(U));
    }
    else{
        fail("preparation error");
    }
}
}
```

第5章　実験

作成したツールを用いて提案手法を用いて実際に開発を行った。ショッピングカートの仕様を用いた。仕様の概要を以下に示す。

- 観測関数

  total カートに入っているすべてのアイテムの合計金額

  subtotal カートの中の指定されたアイテムの合計金額

  itemPrice 指定されたアイテムの価格

- 遷移関数

  addItem アイテムを登録する

  deleteItem アイテムの登録を削除する

  inCart アイテムをカートに入れる

  outCart アイテムををカートから除く

addItem,deleteItem でアイテムの登録場所にアイテムを登録、削除でき、itemPrice によってそれを確認できる。

inCart,outCart で addItem で登録され、deleteItem で削除されていないアイテムを指定された数、カートに入れる、またはカートから出すことができる。

total,subtotal はカートに入っている商品を計算する。

5.1　開発の手順

開発の大まかな流れを説明する。

- 仕様を記述する。
- 型変換、関数変換ファイルを準備する。
- テストケース生成ツールで実装クラスの骨組みとテストケースを生成する。
- コンパイルを通過できるようにコードを記述する

  - 必要なクラスを用意する。
  - 実装クラスのメソッドの戻り値のコードを記述する。
  - テストケースクラスの変数の初期化コードを記述する。
- Preparation のコードを記述する。
- テスト駆動開発を行う。

5.1.1 型変換、関数変換のファイルを準備する

CafeOBJ と Java の型では組み込みデータ型の名前が違うため変換する必要である、今回作成したツールではデータ型の対応を記述したファイルを用意することで、テストケースの雛形を作成する際にデータ型と関数の名前を変換することができる。

型変換ファイルは以下のように記述する。
変換対象　変換先

関数変換ファイルは以下のように記述する
変換対象　変換先
関数の位置

5.1.2 データ仕様のクラスを実装する

例題ではモジュール INT と ITEM をインポートし、システム仕様で利用している。INT は Java の int と共通点が多いため関数名を変換することで利用できる。ITEM は Java に対応するクラスが存在しない。そのためユーザーが対応するクラスを作成する必要がある。Item クラスを作成し, 二つのインスタンスが等しいか判定する equals メソッドをを実装した。

5.1.3 実装クラス

出力されたスケルトンクラスはそのままではコンパイルを通過しないそのためコンパイルを通過可能なように最低限実装を行う。itemPrice では太字の部分を記述した。他のメソッドも同様に実装する。

```
public int itemPrice(Item arg0){
  return 0;
}
```

5.1.4 テストケースクラスの初期化部分を記述する。

テストケース生成ツールは必要な変数の宣言のコードは出力するが、初期化のコードは出力しないため、ユーザーが入力する必要がある。まずはじめはコンパイルが通るように初期化コードを記述する。I にはとりあえず 0 を代入する。Item はユーザーが仕様を元に記述したクラスなので、そこで定義した方法に従ってインスタンスを生成し代入する。

生成されたテストケース

```
@Test
public void  test_addItem_itemPrice_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    if(It1.equals(It2)){
        if(((I>0))){
            shoppingcart.addItem(It1,I);
            assertEquals(I,shoppingcart.itemPrice(It2));
        }
        else{
    fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}
```

コード追加後

```
@Test
public void  test_addItem_itemPrice_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1=new Item();
    int I=0;
```

```
    Item It2=new Item();
    if(It1.equals(It2)){

            省略

    }
}‘
```

5.1.5 準備コードを記述する

とりあえずコンパイルが通過する状態になったが現時点ではテストケースとして機能していない。準備条件を見たすコードを記述することで有効なテストケースにする。ここで準備条件は `It1.equals(It2)` と `I>0` である。`It1.equals(It2)` を満たすために `It2=It1;` を追加し、`I>0` を満たすために `int I=0;` を `int I=10;` に変更する。

```
@Test
 public void  test_addItem_itemPrice_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1=new Item();
    int I=10;
    Item It2=new Item();
    It2=It1;
    if(It1.equals(It2)){
        if(((I>0))){
            shoppingcart.addItem(It1,I);
            assertEquals(I,shoppingcart.itemPrice(It2));
        }
        else{
    fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }

 }‘
```

5.1.6 実装

このシステムの仕様から、inCart,outCart,subTotal,toatl はaddItem,deleteItem,itemPrice に依存していることがわかるまずそのため addItem,deleteItem,itemPrice の実装を進めて行く。

この時点で addItem,deleteItem,itemPrice に関するテストケースでは、準備条件を満たす準備コードを記述することで preparation error が出ていない状態になっていると仮定する。その状態でテストを実行すると test_addItem_itemPrice_0 で

```
java.lang.AssertionError: expected:<10> but was:<0>
```

とエラーが発生する。現在すべてのコンパイルが通るようにとりあえず 0 を返すように実装してあるためである。そのためこのテストケースが通過するように itemPrice が常に 10 を返すように実装する。そうすると test_addItem_itemPrice_0 は通過するが、今度は test_init_itemPrice で

```
java.lang.AssertionError: expected:<0> but was:<10>
```

とエラーが出る。itemPrice は初期化したばかりの状態では 0 を返さなければいけないのに常に 10 を返すためである。これでこのやり方ではうまくいかないことがわかる。そのため実装クラスに itemprice,item というオブジェクトを用意して、addItem の値を保存することにする。

```
class ShoppingCart{
    int itemprice;
    Item item;
    public ShoppingCart(){
        itemprice=0;
        item=null;
    }
    public int subtotal(Item arg0){
        return 0;
    }
    public int itemPrice(Item arg0){
        if(arg0.equals(this.item))
            return this.itemprice;
        else!
            return 0;
    }
```

```
    public void addItem(Item arg0,int arg1){
        this.item=arg0;
        this.itemprice=arg1;
    }
    public int total(){
        return 0;
    }
    public void outcart(Item arg0,int arg1){
    }
    public void deleteItem(Item arg0){
    }
    public void incart(Item arg0,int arg1){
    }
}
```

5.1.7 反復作業

これ以降も実装、エラーの発生するようなテストケース作成を繰り返す。その結果、addItem,deleteItem,ItemPrice, の実装完了した時点で以下のようなコードが得られる。

```
import java.util.HashMap;

class ShoppingCart{
    HashMap<Item, Integer> itemprice;
    Item item;
    public ShoppingCart(){
        itemprice=new HashMap<Item, Integer>();
    }
    public int itemPrice(Item arg0){
        if(itemprice.containsKey(arg0))
            return itemprice.get(arg0);
        else
            return 0;
    }
    public void addItem(Item arg0,int arg1){
        if(arg1>0)
```

```
            this.itemprice.put(arg0, arg1);
    }
    public void deleteItem(Item arg0){
        if(itemprice.containsKey(arg0))
            itemprice.remove(arg0);
    }
    public int total(){
        return 0;
    }
    public void outcart(Item arg0,int arg1){
    }
    public void incart(Item arg0,int arg1){
    }
    public int subtotal(Item arg0){
        return 0;
    }
}
```

5.2 実験結果

ツールを用いることで仕様から 24 個のテストケースの雛形が得られた。そしてそれを元に TDD を行うことで、雛形から生成したテストケースをすべて満たす実装を得ることができた。表 5.1 に実験に用いた仕様、ツールから生成したテストケースの雛形、雛形を埋めて作成したテストケース、テストケースを基に実装した Java のコードのそれぞれのソースコードの量を示す。

仕様	100 行
テストケースの雛形	400 行
テストケース	500 行
Java による実装	70 行

表 5.1: コード量

5.3 考察

- 埋められないテストケースがある。
  テストケースの雛形に複数の準備条件がある場合、それが満たせないよ

うな場合があった。

- データ型をユーザーが実装する必要がある。
  振る舞い型（仕様）に関してはテストケースを生成するが、データ型に関してはサポートが無い
- Item の総数が二つでもテストを通ってしまう。
  仕様では Item はいくつでも保持でき、そのような実装になるべきであるが、今回生成したテストケースでは二つの Item しか保持できない実装でも、生成したテストケースを通過してしまう。

第6章　おわりに

6.1　まとめ

本研究では形式仕様よりテストケース生成し、それを元にテスト駆動開発の提案、それを支援するツールの提案し、それを用いた開発の実践を行った。その結果、ツールが生成した事前条件チェックにしたがってテストケースを記述することで、そのテストケースすべてを満たすJavaの実装を得ることができた。

6.2　今後の課題

- もっと多くの実験が必要
  実験的な開発は今回を行っただけでは十分ではなく、もっと多くの事例研究が必要である。

- テストケース生成以降のサポート
  現在のツールはテストケースの雛形生成までしかサポートしていない、それ以降の作業をサポートする機能が必要である。

- 埋められないテストケースの問題への対処
  今回の実験では準備の条件に従った準備を記述する際に、埋められないことを確認したら、そのテストケースの雛形を削除して開発を続けた。これは非常に手間がかかるためツールに埋められないテストケースを除く機能が必要である。

- Itemの総数が二つの実装でもテストを通ってしまう問題
  今回用いた仕様だけでは対処できないため、仕様以外の補助的な情報を用いて対処する方法が考えられる。

6.3　謝辞

本研究を進めるにあたり、御指導頂いた二木厚吉教授に深く感謝致します。また、有益な御助言をして頂いた中村正樹助手に御礼を申し上げます。最後に、公私に渡り付き合って頂いた言語設計学講座の皆様に感謝致します。

参考文献

[1] CafeOBJ official homepage, http://www.ldl.jaist.ac.jp/cafeobj/.

[2] Kazuhiro Ogata,Kokichi Futatsugi, Modeling and Verification of Distributed Real-Time Systems Based on CafeOBJ Proceedings of the 16th IEEE international conference on Automated software engineering IEEE Computer Society, 2001.

[3] CafeOBJ 入門, http://www.jaist.ac.jp/ t-seino/lectures/cafeobj-intro-j/.

[4] Java.com http://java.com/ .

[5] JUnit.org, http://junit.org/ .

[6] Kent Beck, 長瀬嘉秀, (株) テクノロジックアート, テスト駆動開発入門, ピアソンエデュケーション, 2003.

[7] 大歳雅之, JUnit で実践するテストファーストによる設計, JavaWorld, Vol.102, 2005, pp.148.165.

[8] 川端光義, 倉貫義人, 児玉督司, 長瀬嘉秀, バグがないプログラムのつくり方, 翔泳社, 2004.

付 録A　ShoppingCart.java

```
class ShoppingCart{
public ShoppingCart(){

}
public int total(){

}
public void outcart(Item arg0,int arg1){

}
public void deleteItem(Item arg0){

}
public void incart(Item arg0,int arg1){

}
public int subtotal(Item arg0){

}
public int itemPrice(Item arg0){

}
public void addItem(Item arg0,int arg1){

}
}
```

付 録B　ShoppingCartTest.java

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import org.junit.Test;

public class ShoppingCartTest{

@Test
public void  test_init_itemPrice(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    assertEquals(0,shoppingcart.itemPrice(It1));
}

@Test
public void  test_init_subtotal(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    assertEquals(0,shoppingcart.subtotal(It1));
}

@Test
public void  test_init_total(){
    ShoppingCart shoppingcart = new ShoppingCart();
    assertEquals(0,shoppingcart.total());
}

@Test
public void  test_incart_subtotal_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
```

```
    Item It2;
    int before1=shoppingcart.subtotal(It2);
    int before2=shoppingcart.itemPrice(It2);
    if(It1.equals(It2)){
        if((((shoppingcart.itemPrice(It1)>0)&&(I>0)))){
            shoppingcart.incart(It1,I);
            assertEquals((before1+(before2*I)),shoppingcart.subtotal(It2));
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_incart_subtotal_1(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.subtotal(It2);
    if(!(It1.equals(It2))){
        if((((shoppingcart.itemPrice(It1)>0)&&(I>0)))){
            shoppingcart.incart(It1,I);
            assertEquals(before1,shoppingcart.subtotal(It2));
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}
```

```
@Test
public void  test_incart_subtotal_2(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.subtotal(It2);
    if((!(((shoppingcart.itemPrice(It1)>0)&&(I>0))))){
        shoppingcart.incart(It1,I);
        assertEquals(before1,shoppingcart.subtotal(It2));
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_incart_total_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    int before1=shoppingcart.total();
    if((!(((shoppingcart.itemPrice(It1)>0)&&(I>0))))){
        shoppingcart.incart(It1,I);
        assertEquals(before1,shoppingcart.total());
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_incart_total_1(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    int before1=shoppingcart.total();
    int before2=shoppingcart.itemPrice(It1);
```

```
    if((((shoppingcart.itemPrice(It1)>0)&&(I>0)))){
        shoppingcart.incart(It1,I);
        assertEquals((before1+(before2*I)),shoppingcart.total());
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_incart_itemPrice_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.itemPrice(It2);
    shoppingcart.incart(It1,I);
    assertEquals(before1,shoppingcart.itemPrice(It2));
}

@Test
public void  test_outcart_itemPrice_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.itemPrice(It2);
    shoppingcart.outcart(It1,I);
    assertEquals(before1,shoppingcart.itemPrice(It2));
}

@Test
public void  test_outcart_subtotal_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.subtotal(It2);
```

```
    int before2=shoppingcart.itemPrice(It1);
    if((shoppingcart.subtotal(It2)>(shoppingcart.itemPrice(It1)*I))){
        if(It1.equals(It2)){
            if((((shoppingcart.itemPrice(It1)>0)&&(I>0)))){
                shoppingcart.outcart(It1,I);
                assertEquals((before1-(before2*I)),shoppingcart.subtotal(It2));
            }
            else{
                fail("preparation error");
            }
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_outcart_subtotal_1(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.subtotal(It2);
    if(!((shoppingcart.subtotal(It2)>(shoppingcart.itemPrice(It1)*I)))){
        if(It1.equals(It2)){
            if((((shoppingcart.itemPrice(It1)>0)&&(I>0)))){
                shoppingcart.outcart(It1,I);
                assertEquals(before1,shoppingcart.subtotal(It2));
            }
            else{
                fail("preparation error");
            }
        }
        else{
```

```
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_outcart_subtotal_2(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.subtotal(It2);
    if(!(It1.equals(It2))){
        if((((shoppingcart.itemPrice(It1)>0)&&(I>0)))){
            shoppingcart.outcart(It1,I);
            assertEquals(before1,shoppingcart.subtotal(It2));
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_outcart_total_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    int before1=shoppingcart.total();
    int before2=shoppingcart.itemPrice(It1);
    if((shoppingcart.subtotal(It1)>(shoppingcart.itemPrice(It1)*I))){
        if((((shoppingcart.itemPrice(It1)>0)&&(I>0)))){
```

```
            shoppingcart.outcart(It1,I);
            assertEquals((before1-(before2*I)),shoppingcart.total());
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_outcart_total_1(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    int before1=shoppingcart.total();
    if(!((shoppingcart.subtotal(It1)>(shoppingcart.itemPrice(It1)*I)))){
        if((((shoppingcart.itemPrice(It1)>0)&&(I>0)))){
            shoppingcart.outcart(It1,I);
            assertEquals(before1,shoppingcart.total());
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_outcart_total_2(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    int before1=shoppingcart.total();
```

```
    if((!(((shoppingcart.itemPrice(It1)>0)&&(I>0))))){
        shoppingcart.outcart(It1,I);
        assertEquals(before1,shoppingcart.total());
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_addItem_itemPrice_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    if(It1.equals(It2)){
        if(((I>0))){
            shoppingcart.addItem(It1,I);
            assertEquals(I,shoppingcart.itemPrice(It2));
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_addItem_itemPrice_1(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.itemPrice(It2);
    if(!(It1.equals(It2))){
        if(((I>0))){
```

```
            shoppingcart.addItem(It1,I);
            assertEquals(before1,shoppingcart.itemPrice(It2));
        }
        else{
            fail("preparation error");
        }
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_addItem_itemPrice_2(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.itemPrice(It2);
    if((!((I>0)))){
        shoppingcart.addItem(It1,I);
        assertEquals(before1,shoppingcart.itemPrice(It2));
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_addItem_subtotal_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    Item It2;
    int before1=shoppingcart.subtotal(It2);
    shoppingcart.addItem(It1,I);
    assertEquals(before1,shoppingcart.subtotal(It2));
}
```

```
@Test
public void  test_addItem_total_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int I;
    int before1=shoppingcart.total();
    shoppingcart.addItem(It1,I);
    assertEquals(before1,shoppingcart.total());
}

@Test
public void  test_deleteItem_itemPrice_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    Item It2;
    if(It1.equals(It2)){
        shoppingcart.deleteItem(It1);
        assertEquals(0,shoppingcart.itemPrice(It2));
    }
    else{
        fail("preparation error");
    }
}

@Test
public void  test_deleteItem_itemPrice_1(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    Item It2;
    int before1=shoppingcart.itemPrice(It2);
    if(!(It1.equals(It2))){
        shoppingcart.deleteItem(It1);
        assertEquals(before1,shoppingcart.itemPrice(It2));
    }
    else{
        fail("preparation error");
    }
```

```
}

@Test
public void  test_deleteItem_subtotal_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    Item It2;
    int before1=shoppingcart.subtotal(It2);
    shoppingcart.deleteItem(It1);
    assertEquals(before1,shoppingcart.subtotal(It2));
}

@Test
public void  test_deleteItem_total_0(){
    ShoppingCart shoppingcart = new ShoppingCart();
    Item It1;
    int before1=shoppingcart.total();
    shoppingcart.deleteItem(It1);
    assertEquals(before1,shoppingcart.total());
}

}
```