テスト駆動開発入門
ハンズオン講座（前篇）

goyoki

目的

- TDDの具体的なイメージをつかんでもらう
  - 実際の進め方
  - 運用時に必要性の高い周辺知識
  - TDDのテストの特徴

概要（前篇）

- イントロダクション
- ハンズオン課題1
- TDDの概要
  - 定義/手順/利益など
- ハンズオン課題2
- ソフトウェアテストとしてのTDD

後編について

- 以下は後編で扱う予定です
  - TDDが抱える課題
  - レガシーコード上でのTDD
  - テストコードの改善
  - TDDで確保したテストコードの活用
  - TDDの諸目的

イントロダクション

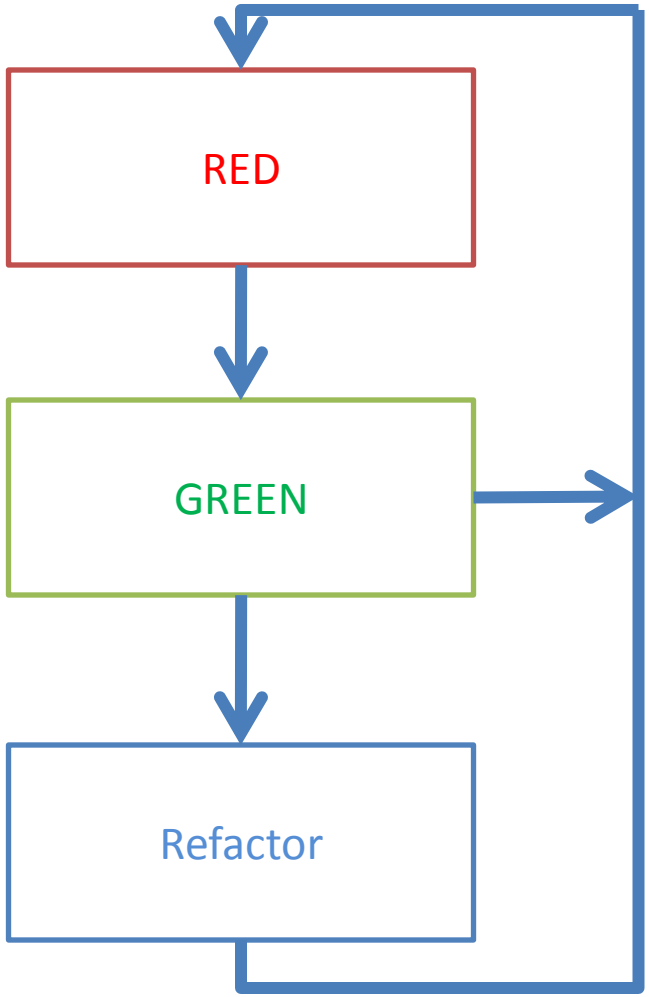
テスト駆動開発（TDD）

- テストファーストプログラミングの１手法
  - アサートファースト
- プログラミング技法
- Kent Beckが具体的な方法論としてまとめる

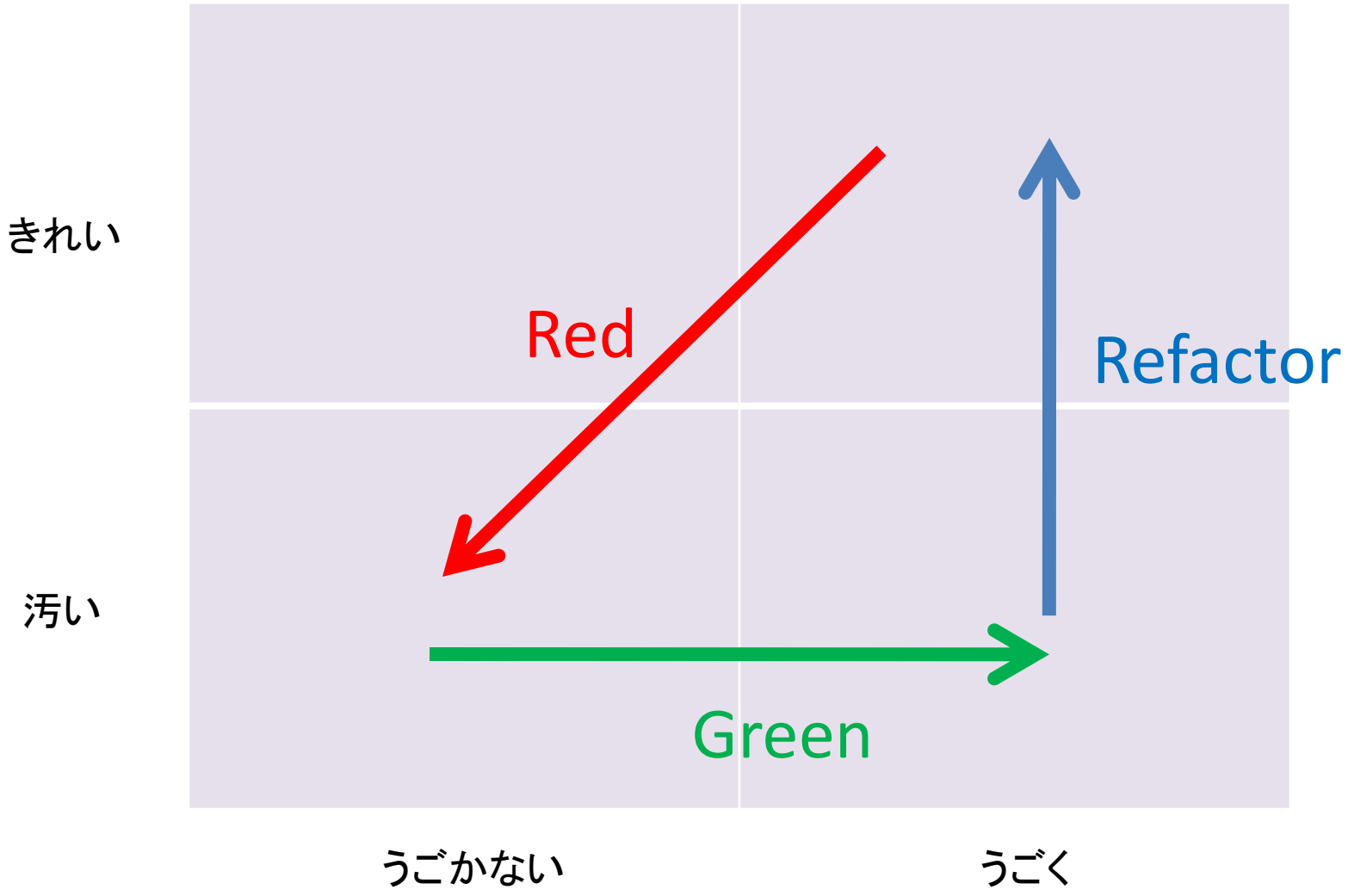
TDDのステップ

1. 最初にテストを書いて実行（RED）
2. テストをパスするまでコードを実装（GREEN）
3. コードをきれいにする（REFACTOR）

これを繰り返しインクリメンタルに実装を進める



コードの４象限とTDDのサイクル



ハンズオン課題1

実装仕様

- うるう年判定関数（グレゴリオ暦）
  - 西暦年が4で割り切れる年は閏年
  - ただし、西暦年が100で割り切れる年は平年
  - ただし、西暦年が400で割り切れる年は閏年

TDDの概要

TDDの定義

- 「テスト駆動開発入門」 Kent Beck
  - バイブル的存在
- 方法論としての厳密な定義を強制しない
  - 「TDDはXPのように絶対的ではない」
    - テスト駆動開発入門
- 有力な原則やアドバイスで方法論を構築

TDDの原則

- Robert C Martinの３原則
  - 失敗するユニットテストを成功させるためにしか、プロダクトコードを書いてはならない。
  - 失敗させるためにしか、ユニットテストを書いてはならない。コンパイルエラーは失敗に数える。
  - ユニットテストを1つだけ成功させる以上に、プロダクトコードを書いてはならない。

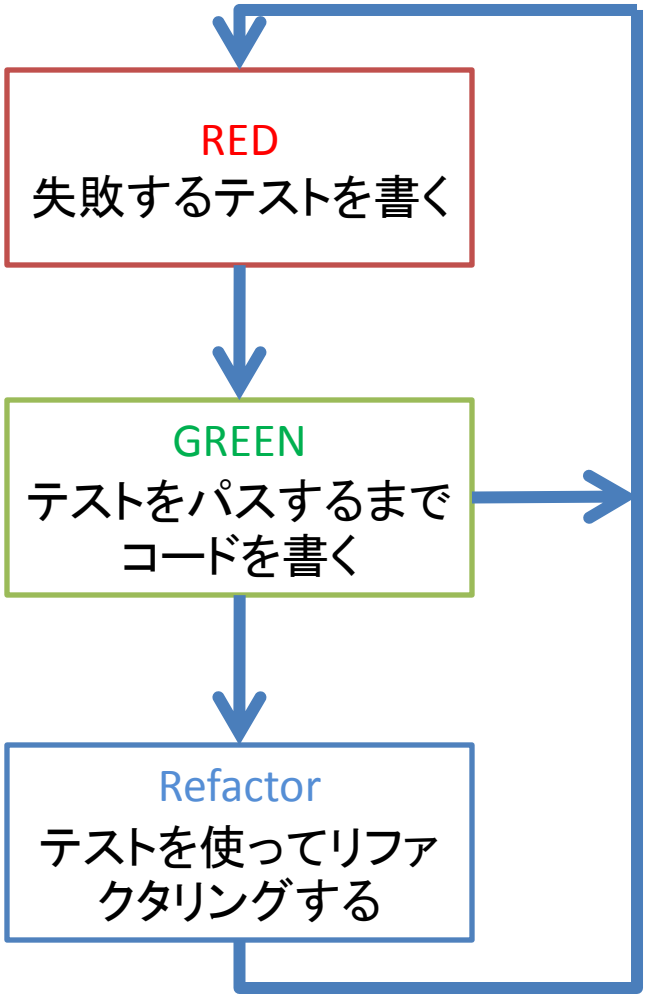
目指すべき

TDDのテストの原則

- 完全な自動テストであること
- 自己完結できるテストであること
  - 初期設定、実行、検証がセットとなっている
- 繰り返し可能なテストであること
  - 何度実行しても結果は同じ
- 独立して実行できるテストであること
  - 一緒/個別に実行しても結果は同じ
  - 順不同でも結果は同じ
- 十分に細粒度であること
  - 作業の最小単位ごとにテストを書く

TDDの手順
(Red、Green、Refactorのサイクル)

TDDの流れ


RED
失敗するテストを書く
GREEN
テストをパスするまで
コードを書く
Refactor
テストを使ってリファ
クタリングする

Redのステップ

- テストを書く
  - 失敗するテストを継ぎ足す
  - 小さなテストを書く
    - ここで書いたテストが実装作業単位になる

Greenのステップ

- テストをパスするコードを書く
  - Redのステップで失敗しているテストを通すまでコードを書く
  - テストが失敗状態のまま他の作業に移らない

Refactorのステップ

- Greenで追加・変更したコードをきれいにする
  - テストがパスした状態を維持する
  - 既存のテストで可能なリファクタリングを行う
    - 新規実装の場合はRedのステップに移る
  - Refactorのために新たにテストを追加してもよい
  - 実行するほどでもないなら飛ばしてもよい

TDDの利益

TDDの利益

1. すばやく継続的なフィードバック
2. 作業の細分化・局所化
3. 単体テスト容易性の向上

すばやく継続的なフィードバック

- 単体テスト環境の整備
- 実装未達の即時検出
- デグレードの即時検出

- TDD上での実装作業は単体テストでリアルタイムに検証される
- TDD上では自動化された回帰テスト環境を容易に構築できる

作業の細分化・局所化

- Defect Localization
- やっていること、やるべきことを絞り込む
- 不完全なコードを1つに絞り込む

- 「テスト=実装仕様」としてやるべきことを具体化できる
- TDDではGreen状態をこまめに確保
  「テスト失敗原因≒直近の作業」を実現

単体テスト容易性の向上

- リファクタリング容易性の向上
- 単体テストの網羅性の向上

- TDDでは製品コードが単体テストに対して最適化される
- リファクタリングやCover&Modify(後編)が容易になり、コードの品質（移植性等）が向上する

TDDでのテクニック
(Red、Green、Refactorのサイクルの中でのテクニック)

「テスト駆動開発入門」における基本パターン

1. Fail It
2. Fake It
3. Triangulate（三角測量）
   1～3を十分に積み重ねた後
4. Obvious Implementation(明白な実装)

1. Fail It

テストコード

```
@Test
Pubic void test引数を倍にして返す()
{
   assertEquals(6,  引数を倍にして返す(3));
}
```

製品コード

```
Pubic int引数を倍にして返す(int input)
{

}
```

スケルトン、あるいは未実装

2. Fake It

テストコード

```
@Test
Pubic void test引数を倍にして返す()
{
   assertEquals(6,  引数を倍にして返す(3));
}
```

製品コード

```
Pubic int引数を倍にして返す(int input)
{
   return 6;
}
```

3. Triangulate(三角測量)

テストコード

```
@Test
Pubic void test引数を倍にして返す()
{
   assertEquals(6,  引数を倍にして返す(3));
   assertEquals(10, 引数を倍にして返す(5));
}
```

製品コード

```
Pubic int引数を倍にして返す(int input)
{
   return 6;
}
```

4. Obvious Implementation (明白な実装)

テストコード

```
@Test
Pubic void test引数を倍にして返す()
{
   assertEquals(6,  引数を倍にして返す(3));
   assertEquals(10, 引数を倍にして返す(5));
}
```

製品コード

```
Pubic int引数を倍にして返す(int input)
{
   return input * 2;
}
```

ステップの調整

- ステップ、粒度は不安や慎重度に応じて調整

- 慎重な場合
  - Fail It(Compile Error)→Fait It(Red)→Fake It→ Triangulate→ Obvious Implementation
- 平易な場合
  - Fail It→ Obvious Implementation

注意

- Triangulateの扱いは意見が分かれる
  - 長所 : 作業をより細かいステップに分けられる
    - Fake Itではビルド・実行可能かどうか検証できる
    - インターフェースが妥当か検証するステップになる
  - 短所 : 重複するテストが生まれる
    - Fragile Testの原因となる
- 作業のステップの大小に応じて調整する

TDDでの
テストコードの整理

テストコードの整理

- テストコードを整理する場面
  - 製品コードの変更に追従するために
    - コードの変更で冗長になったテストを最適化する
    - インターフェースの変更に追従する
  - テストコードの品質を向上させるために
    - テスト設計を損なわないようにテスト実装を整理する
    - TDDの中で生まれたテストの冗長性を整理する

テストコードの整理

- TDDの一部として扱われない
  しかし放置するとTDDを非効率なものにする
  - 製品コードのリファクタリング容易性や拡張性を損なう（Fragile Test等）
  - テストコードがミスを見逃すようになる(Buggy Test等)
  - TDDの作業量を無用に増大させる(Assertion Roulette,Slow Test等)
  - テストコードの保守性を低下させる(Obscure Test等)
- TDDの効率を維持するために不可避なステップ

整理のタイミング

- TDDのサイクル上のタイミング
  - Red→Green→Refactor→テストコード整理
    - リファクタリングに合わせて最適化
  - (Red→Green→Refactor)を繰り返す→テストコード整理
    - 蓄積したテストコードをまとめ上げる
- その他タイミング
  - 特定のイベント前
    - コミット前/CIなどへの組み込み直前/派生先への流用前等

整理の目標

- TDDのテストの原則を目指す
  - 完全な自動テストであること/自己完結できるテストであること/
  - 繰り返し可能なテストであること/独立して実行できるテストであること
  - 十分に細粒度であること

- 「単体テスト」としての妥当性を目指す
  - 簡単に実行できるべき
  - テストは品質向上を手助けしてくれるべき
  - テストはテスト対象の理解を手助けしてくれるべき
  - テストはリスクを削減してくれるべき
  - テストは簡単に実行できるべき
  - テストは簡単に変更・保守できるようにするべき
  - システムの機能拡張時でもテストの変更は最小限になるようにすべき

テストコードの整理での注意点・アドバイス

- 慎重に行う
  - テスト設計の等価性を保証する技法は不十分
- 一時的であってもLost Testは避ける
  - アプローチは基本的にParallel Change。テストの削除は代替が十分に揃ってから
- 製品コードを工夫する
  - テストコードの保守性を観点に製品コードを組む
- 言語、ツールの力を借りる
  - IDEのリファクタリング機能といった、低リスクな機能を積極活用

テストコード整理の例

- Custom Assertionによる置換
  - まとまったAssertionのセットを一つのAssertionにまとめる
  - Assertionのセットの重複記述を解消する
- 注意
  - 利点・欠点共にある
  - CUnitといった行番号情報が重要なフレームワークではプリプロセッサでまとめたほうが良い

Custom Assertionによる置換

```
@Test
Pubic void test3()
{
   ….
   assertEquals(bar.a(),  foo.a());
   assertEquals(bar.b(),  foo.b());
   assertEquals(bar.c(),  foo.c());
}

@Test
Pubic void test2()
{
   ….
   assertEquals(bar.a(),  foo.a());
   assertEquals(bar.b(),  foo.b());
   assertEquals(bar.c(),  foo.c());
}

@Test
Pubic void test1()
{
   ….
   assertEquals(bar.a(),  foo.a());
   assertEquals(bar.b(),  foo.b());
   assertEquals(bar.c(),  foo.c());
}
```

Custom Assertionによる置換

```
@Test
Pubic void test3()
{
   ….
   assertEquals(bar.a(),  foo.a());
   assertEquals(bar.b(),  foo.b());
   assertEquals(bar.c(),  foo.c());
}

@Test
Pubic void test2()
{
   ….
   assertEquals(bar.a(),  foo.a());
   assertEquals(bar.b(),  foo.b());
   assertEquals(bar.c(),  foo.c());
}

@Test
Pubic void test1()
{
   ….
   assertEquals(bar.a(),  foo.a());
   assertEquals(bar.b(),  foo.b());
   assertEquals(bar.c(),  foo.c());
}
```

```
static void assertHoge(fuga exp, fuga
act)
{
   assertEquals(exp.a(),  act.a());
   assertEquals(exp.b(),  act.b());
   assertEquals(exp.c(),  act.c());
}
```

Custom Assertionによる置換

```
@Test
Pubic void test3()
{
   ….
   assertHoge(bar, foo);
}

@Test
Pubic void test2()
{
   ….
   assertHoge(bar, foo);
}

@Test
Pubic void test1()
{
   ….
   assertHoge(bar, foo);
}
```

```
static void assertHoge(fuga exp, fuga
act)
{
   assertEquals(exp.a(),  act.a());
   assertEquals(exp.b(),  act.b());
   assertEquals(exp.c(),  act.c());
}
```

TDDを支える
テスティングフレームワーク

TDDを支える テスティングフレームワーク

- 自動化された単体テストをサポートするフレームワークが多用される
- xUnitフレームワークが一般的
  - TDDで用いられる他のフレームワークでもxUnitと同等の機能を持つことが多い

xUnit

- Kent BeckがSmalltalk用に作成したテスティングフレームワークが元祖
- その設計思想に従った単体テスティングフレームワークの総称
- JUnit、SUnit、NUnit、cppUnitなど

xUnit

- 階層構造、カテゴリによりテストを構造化
  - Test Assert : Test Method : Test Suite
- Four-Phase Testでテストの独立性や保守性を向上
  - Setup→Exercise→Verify→Teardown
- 開発言語の設計能力を活用しテストの保守性を向上
  - JUnit : Test Suiteの定義にClass、カテゴリの定義にアノテーション等を活用

xUnitの構造

http://xunitpatterns.com/XUnitBasics.html:
Sketch Static Test Structure参照

xUnitの構造：基本用語解説

- SUT（System Under Test）
  - テスト対象
- Fixture
  - テスト実行前にSUTに対して行う事前設定
- Test Runer
  - テストを実行Suiteを抜き出してテストを実行するプロセス
- Test Double
  - テスト実行時に、SUTが依存するコードやコンポートンとの代替となるもの

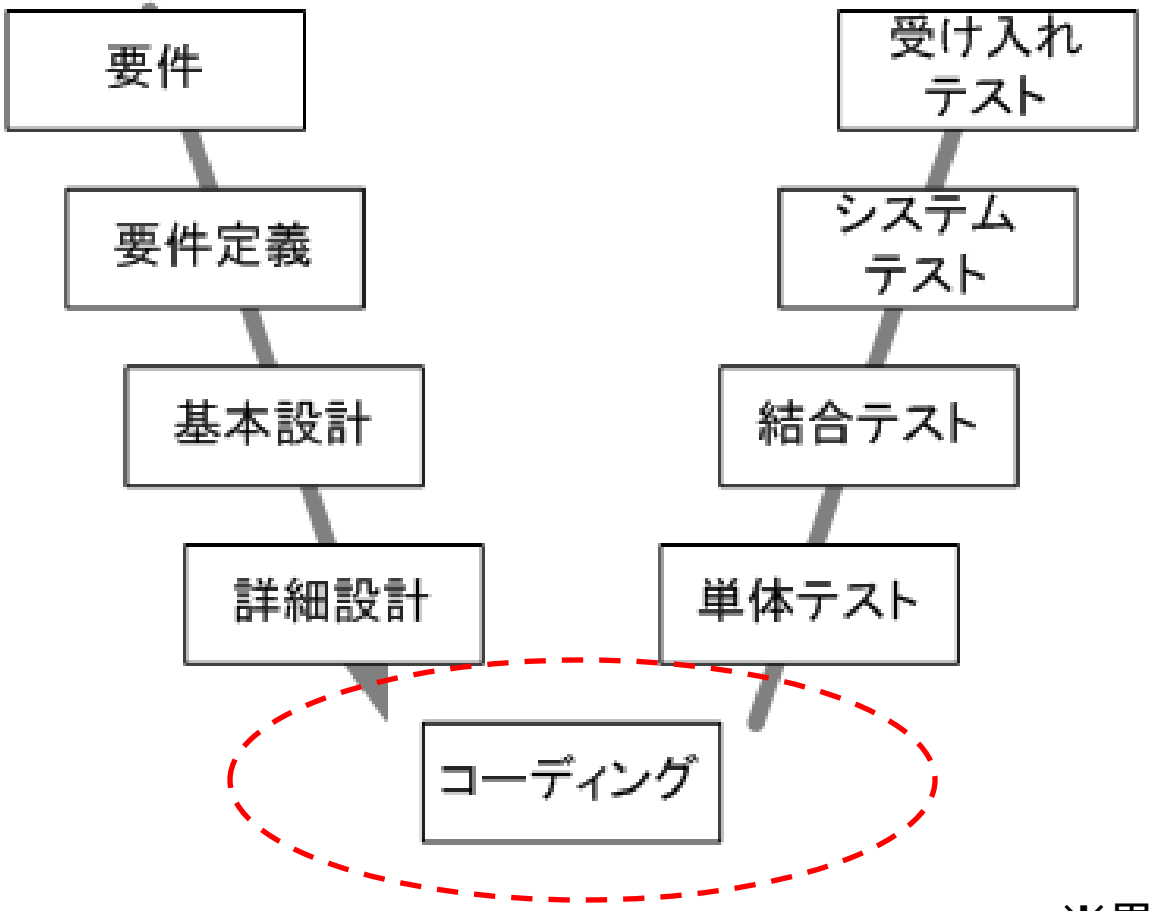
ハンズオン課題2

課題

- 本リストを管理するClass
  - 書名と価格を格納できる
  - 格納順でもっとも古い本を削除できる
  - 格納順の番号で本を検索できる
  - 書名から価格を検索できる

ソフトウェアテスト手法としてのTDD

TDDの主な対象工程



※異論もある

TDDのテストはテストなのか？

- テストの目的・観点が一般的なものと異なる
  - テスト技術者視点としてはテストと見なしても問題ないと思われる
    - 網羅的なバグ出し・QAが主な目的ではない
    - フェーズは主に開発（実装）工程
    - テストレベルは主に単体テスト
    - アプローチは大まかに設計ベース・構造ベース

TDDのテストと
工程検査の単体テスト設計は
何が違うのか？

- 傾向としての違い:
  - テストファーストで作られる
  - 製品コード実装と連携して実装される
  - 実装上の意図を元に設計される
  - 継続的でインクリメンタルに設計される

TDDは
どのようなテスト設計技法なのか？

- TDDでは特定のテスト設計技法を強制しない
  - 一般的な単体テスト設計技法と両立可能
    - Ex)全体的なテスト設計を実施→1つ1つ切り出して Red>Green>Refactorのサイクルを回す
  - ただTDDとの相性として、技法の適・不適はある

TDDと従来の単体テスト設計は一致させられるか？

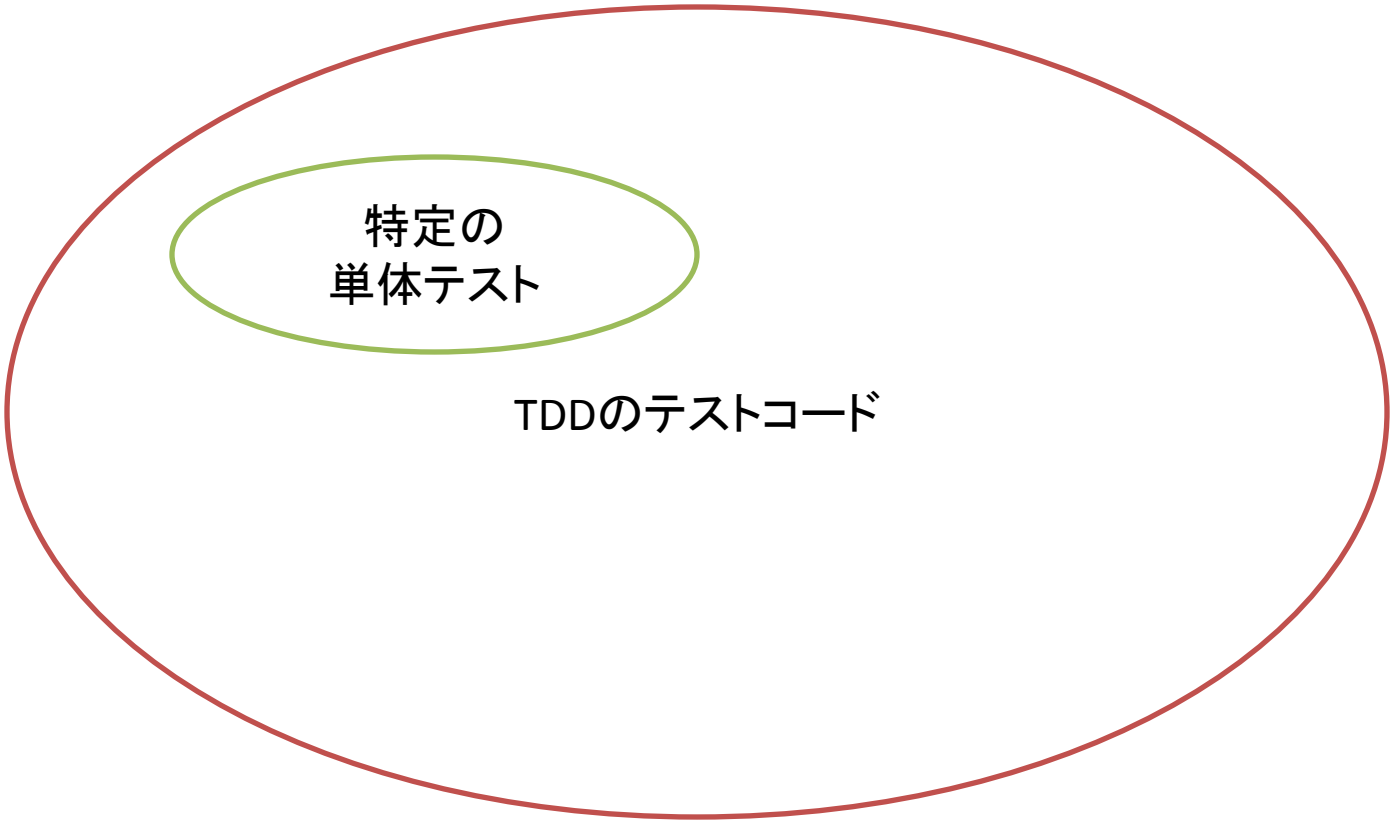
- 一致は可能。しかし一般的に非効率
  - 「実装作業のサポート」という観点が抜け落ちたテスト設計は、TDDの効率を削ぐ
  - 「実装作業のサポート」という観点で設計されたテストは、保証目的のテスト設計として冗長性や抜けを持つことが多い

TDDと従来の単体テスト設計は一致させられるか？

- 一致でなく共存による相乗効果を目指す
  - TDDでは整合性のある単体テストを内包するように実装を進める
  - 工程保証ステップではTDDで確保されたテスト容易性・テストコードを活用する

共存プロセス

- 目的の単体テストを内包するようにTDDを進める



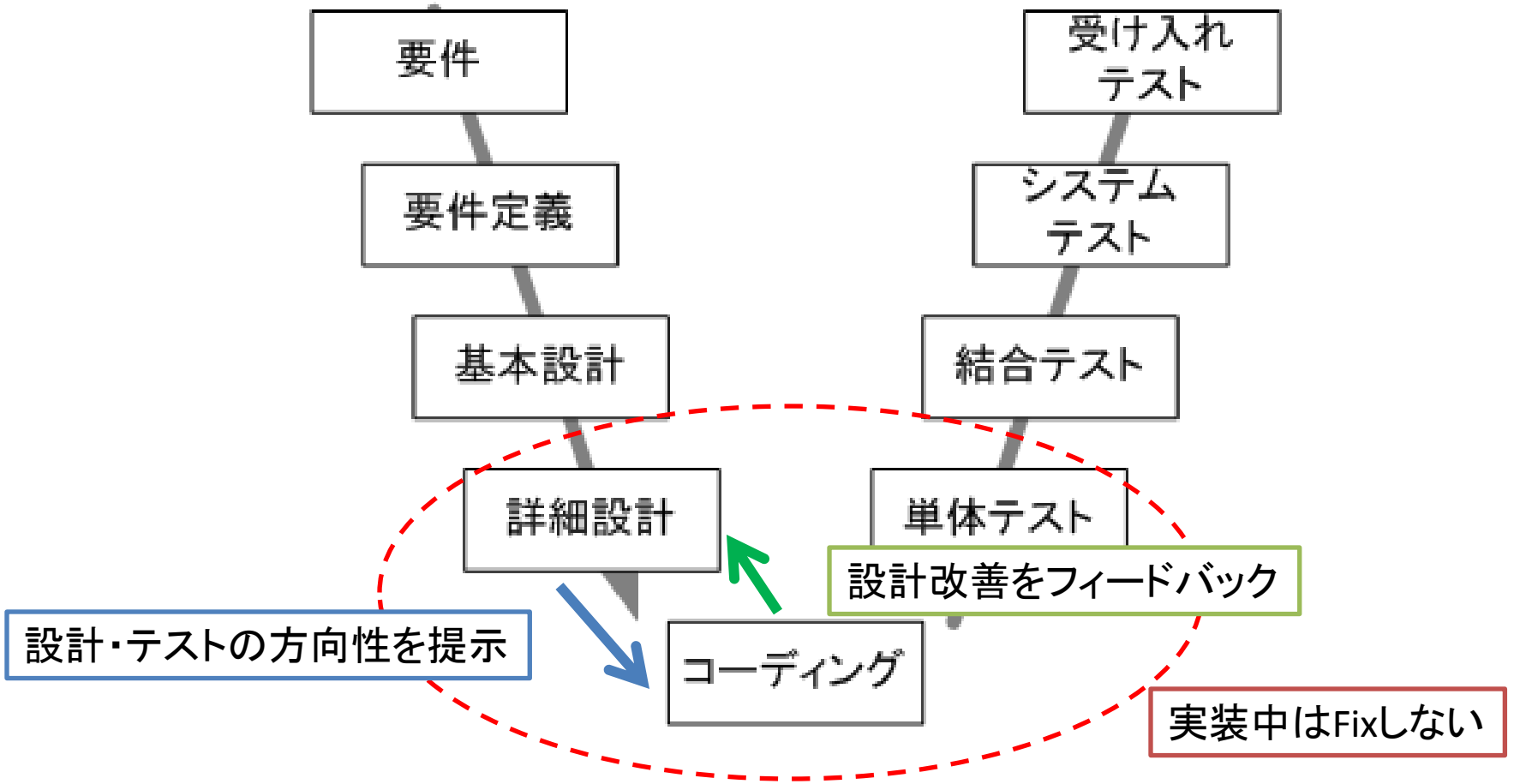
共存プロセス

- Outside-InのTDD
  - 上位設計をブレークダウンしていく

http://xunitpatterns.com/Philosophy%20Of%20Test%20Automation.html:
“Outside-In” development of functionality supported by Test Doubles.
参照のこと

共存プロセス

- **詳細設計とTDDで反復フローを構成**



まとめ

- TDDとは/では
  - Red→Green→Refactorのサイクルでコードとテストを蓄積していく
  - 自動化された単体テストで、継続的かつ即時のフィードバックを実装作業に返す
  - 単体テスト容易性と単体テストコードを確保する
  - ソフトウェアテストとして協調できる