CEDEC2008

開発方法論

2008年9月10日

鷲崎 弘宜
早稲田大学 / 国立情報学研究所
washizaki@waseda.jp

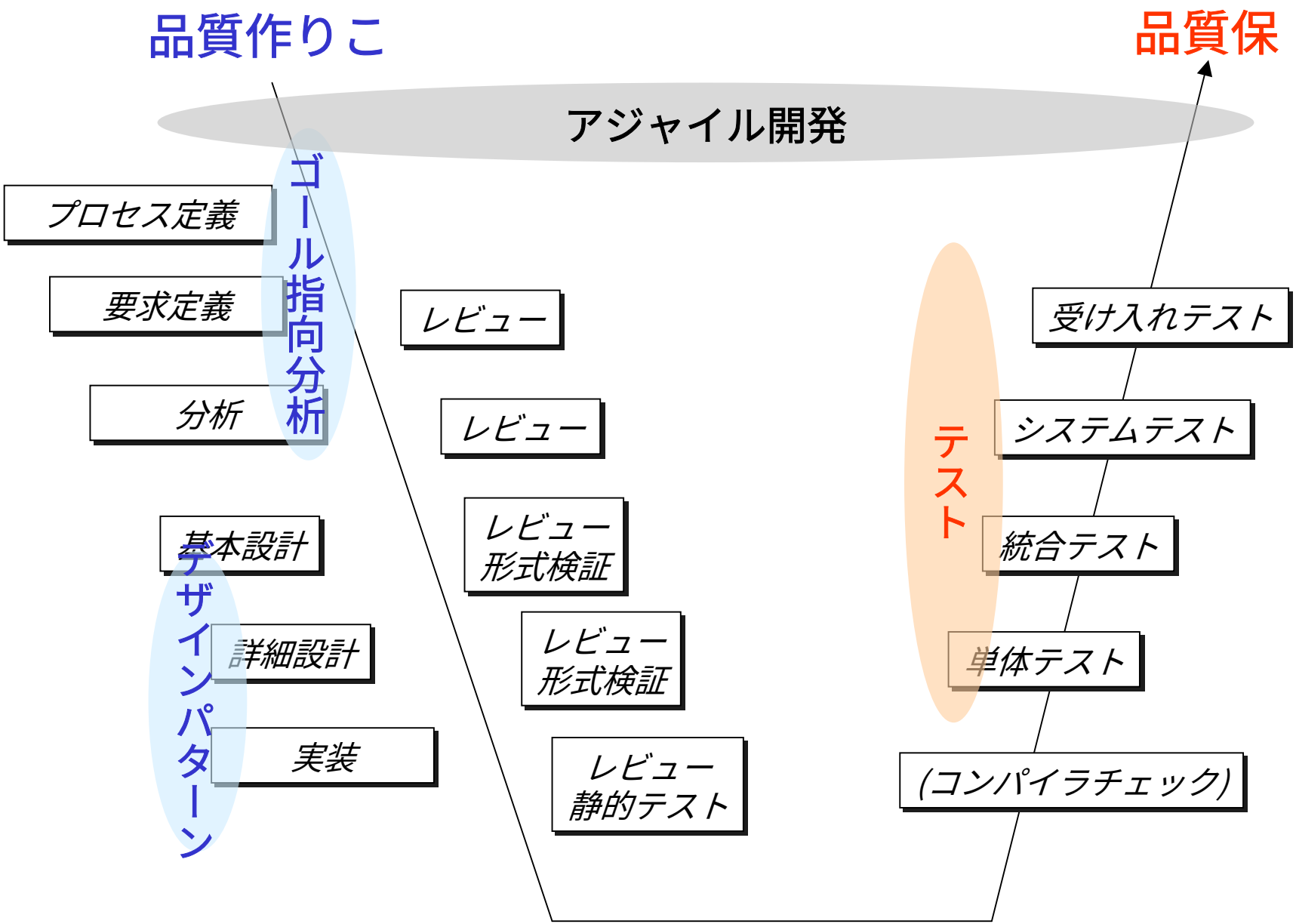
ソフトウェア工学とは何か

- 工学
  - “科学知識を応用して、大規模に物品を生産するための方法を研究する学問” [大辞林]
  - 製品を作るための理論、原理、法則、生産技術のまとまり
- ソフトウェア工学（エンジニアリング）
  - “ソフトウェアの開発、運用、および保守に対する系統的で規律に基づいた定量的アプローチ” [SWEBOK]
  - ソフトウェア製品を作るための理論、原理、法則、生産技術のまとまり
- なぜ必要なのか？
  - 属人性を排除し、一定の品質を保証したい（高品質）
  - 生産性を向上させたい（大規模・多品種）

⇒ 工業化、技術者育成

[SWEBOK] “Software Engineering Body of Knowledge”
IEEE CS/ACM, Software Engineering Coordinating Committee
http://www.swebok.org/

プロセスで見ると


品質作りこ
品質保
アジャイル開発
ゴール指向分析
プロセス定義
要求定義
分析
デザインパターン
基本設計
詳細設計
実装
レビュー
レビュー
レビュー
形式検証
レビュー
形式検証
レビュー
静的テスト
テスト
受け入れテスト
システムテスト
統合テスト
単体テスト
(コンパイラチェック)

ゴール指向分析

要求定義は難しい。なぜか？

- スコープの問題
  - システムの利害関係者は多様である。
  - システムの境界が曖昧である。
- 理解の問題
  - 顧客の要求は時に曖昧である。
  - 顧客の要求と、開発者が考える「顧客の要求」の間には時にギャップがある。
  - 顧客の要求は、顧客が本当に欲している事柄とは時にギャップがある。
- 不安定性の問題
  - 要求、状況、環境は常に変化する。

⇒ 完全な解決策は存在しないが、要求エンジニアリング・工学的技法の適用により制御を個試みる

要求エンジニアリングプロセス

- 1. 開始（inception）
  - ステークホルダの認識
  - 複数の視点を見分ける
  - 自由に質問する（誰が誰のために？利益？ など）
  - [技術] ゴール指向分析など
- 2. 要求獲得（elicitation）
  - 協調的な要求収集: 会議など
  - [技術] ゴール指向分析、品質機能展開、シナリオ、UMLユースケースなど
- 3. 推敲（elaboration）
  - 得られた情報の洗練
  - [技術] 品質機能展開、シナリオ、UMLユースケース/アクティビティ図、アナリシスパターンなど

要求エンジニアリングプロセス（つづき）

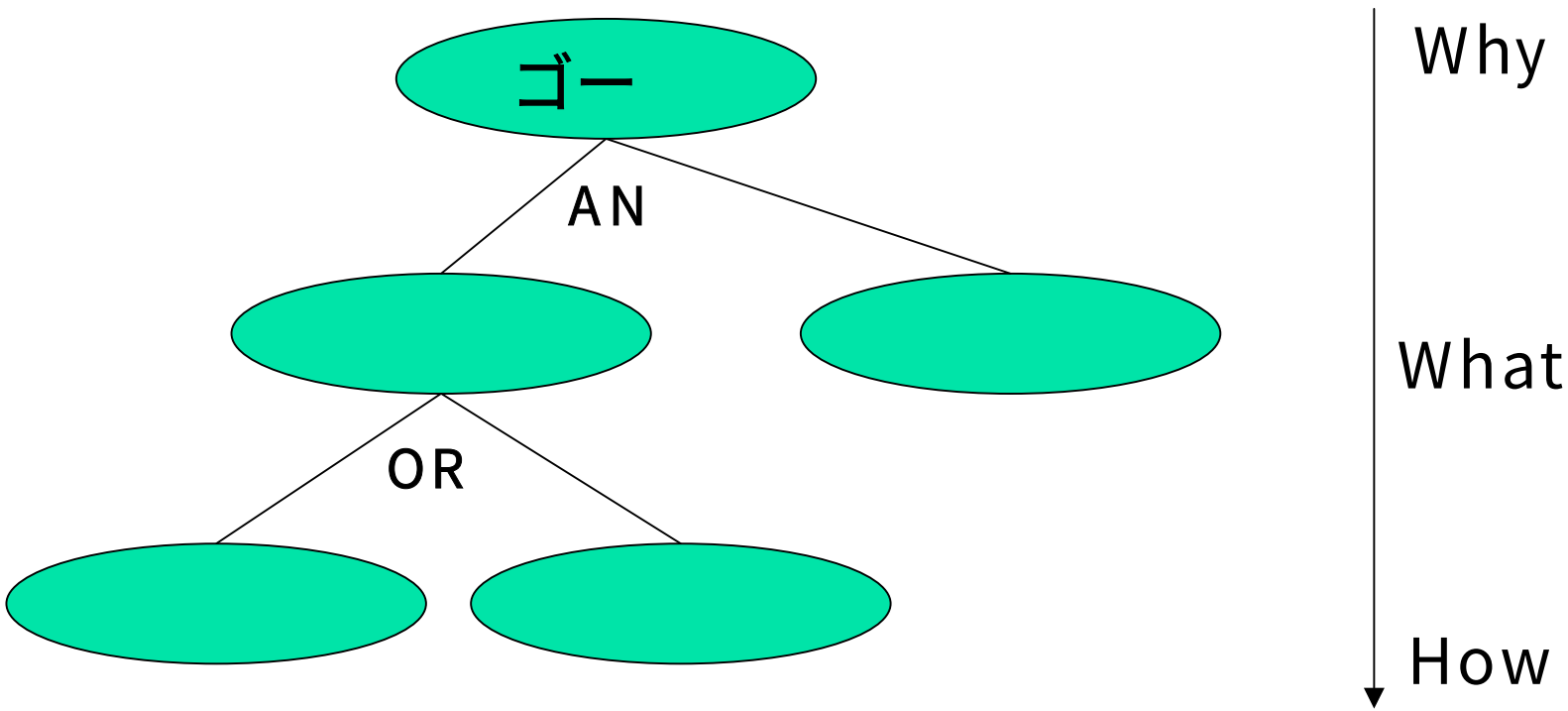
- 4. 交渉（negotiation）
  - 顧客と開発者が win-win となるように双方の利益を定義
  - 優先順位付け、リスク・作業量の蜜所利、要求の削減など
- 5. 仕様化（specification）
  - 特定テンプレートに基づく記述
  - モデルを表す図やプロトタイプによる補足
  - [技術] IEEEstd.830-1998 テンプレート
- 6. 検証（validation）
  - 要求の明確さや抜け、不整合、誤りなどの確認
  - [技術] ゴール指向分析、公式技術レビュー、チェックリスト、 IEEEstd.830-1998 要求仕様書品質（正当性、追跡可能性など）
- 7. 管理（management）
  - 要求から機能や特徴、以降の成果物への追跡性の維持と要求変更時の影響分析
  - [技術] 追跡表（traceability matrix/table）

ゴール指向分析

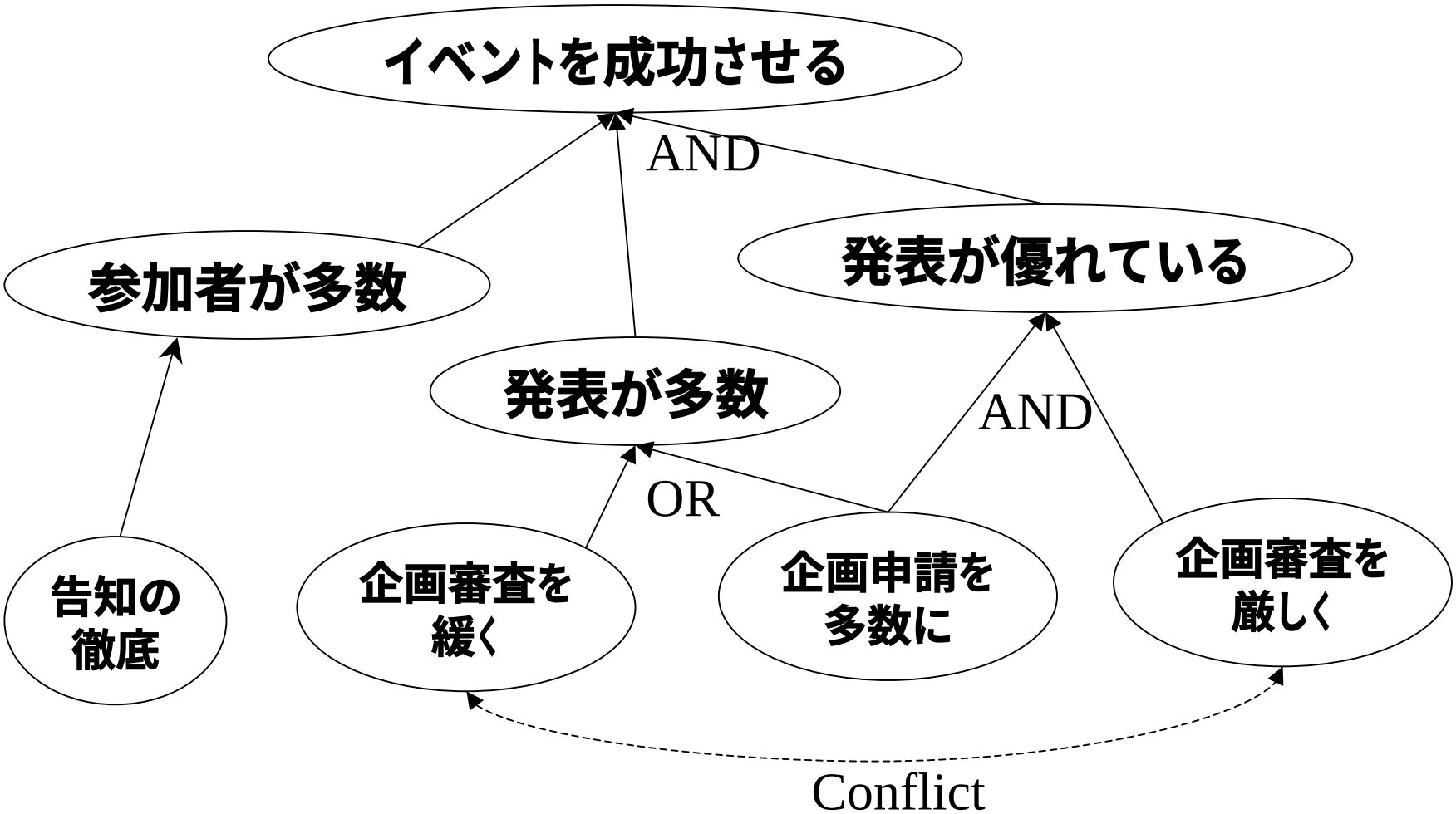
- ゴール（goal）
  - 最終的に達成される目的
  - 要求（what）を実現するための目標、why
- ゴール指向分析（goal-oriented analysis）
  - ゴールの詳細を整理し、要求（さらには仕様: how）の明確化を導く手法
  - ゴールを明確にすることで、要求の獲得や検証を容易に
- ゴール木（ツリー）/ゴールモデル
  - 主に上位から下位へ詳細化する木構造モデル

ゴール木（ツリー） / ゴールモデル

- 主に上位から下位へ詳細化する木構造モデル
  - ＡＮＤ分解: 下位の全てが必須
  - ＯＲ分解: 下位のいずれかが必要
  - 下位は要求（What）・仕様（How）になることがある
- ゴール以外の登場要素
  - ステークホルダ（利害関係者）: 誰のゴール？
  - リソース、モジュール（エージェント）など



ゴールモデルの例



参考・出典: 海谷治彦, 要求工学イブニングチュートリアル第4回ゴール指向要求分析法, http://kaiya.cs.shinshu-u.ac.jp/2004/gora/

ゴール指向分析の種類

- i*（アイスター）
  - アクタ、ゴール、タスク、ソフトゴール（達成を明確には判定困難なゴール）、リソース間の依存関係の記述
  - Strategic Dependency/Rational モデル
  - 利害関係者やアクタが複数の場合に要求獲得に利用
- KAOS（ケイオス or カオス）
  - ゴールから要求さらには操作・割り当てエージェントまでの明確化（Why -> What -> Who）
  - 形式的な要求の記述と検証
  - 支援ツールあり: http://www.objectiver.com/
- NFR（Non-Functional Requirements） Framework
  - 非機能要求、特に品質要求の分析に特化
  - 典型的な品質要求とその達成手段のカタログつき
- 情報
  - 海谷治彦, 要求工学, http://kaiya.cs.shinshu-u.ac.jp/2004/gora/
  - 山本修一郎、要求工学, http://www.bcm.co.jp/site/youkyu/index.html
  - 以降の例:風戸 広史, TopSE 修了制作, http://www.topse.jp/events/symposium07/kazato.pdf

デザインパターン

ソフトウェアパターンとは

- ソフトウェア開発で頻出する特定局面・状況での「問題」に対する「解法」を表現したもの
- ソフトウェアパターンとは、ある文脈について、熟練者による類似した「優れた」ソフトウェアを生み出した成功事例を集めて共通部分を抽象化し、名前を与えた指針

- 初心者によるソフトウェア
  - 機能要求を何とか満たす
  - 非機能要求はおざなり
- 熟練者による「優れた」ソフトウェア
  - 機能要求をきっちり満たす
  - トレードオフの関係にある様々な非機能要求をきれいにみたす

ソフトウェアパターンの記述形式

項目	くだけて言うと	内容
名前	造語	パターンの内容を的確に連起させる造語(あるいは用語)
状況	こういうときに	主に**機能要求**を満たすソフトウェアを開発すべき状況
問題	こうしたかったら	主に**非機能要求**を満たしたいという問題
解決	こうしましょう	採った開発の方針と解決策(思考過程)
結果	こうなります	得られたソフトウェアにおいて**機能/非機能要求**が満たされた状況
フォース	なぜならば	解決策を選択するにあたり考慮する事柄、解決に成功する理由、場に働く「力」

例: パターンのモト

```
class Client {
  Mathematic math;

  void init() {
    math.settings=QUICK;
  }
  void calc() {
    data = math.sort(data);
    ...
  }
```

```
class Mathematic {
  int settings;

  public Data sort(Data data){
    if(settings == QUICK) {
      return quickSort(data);
    } else if(settings == BUBBLE) {
      return bubbleSort(data);
    } else ...
  }
```

```
class LoginView {
  void validateUser() {
    boolean loginResult;
    ...
    if(MODE == TEST) {
      loginResult = true;
    } else {
      // DB接続、認証
    }
    ...
```

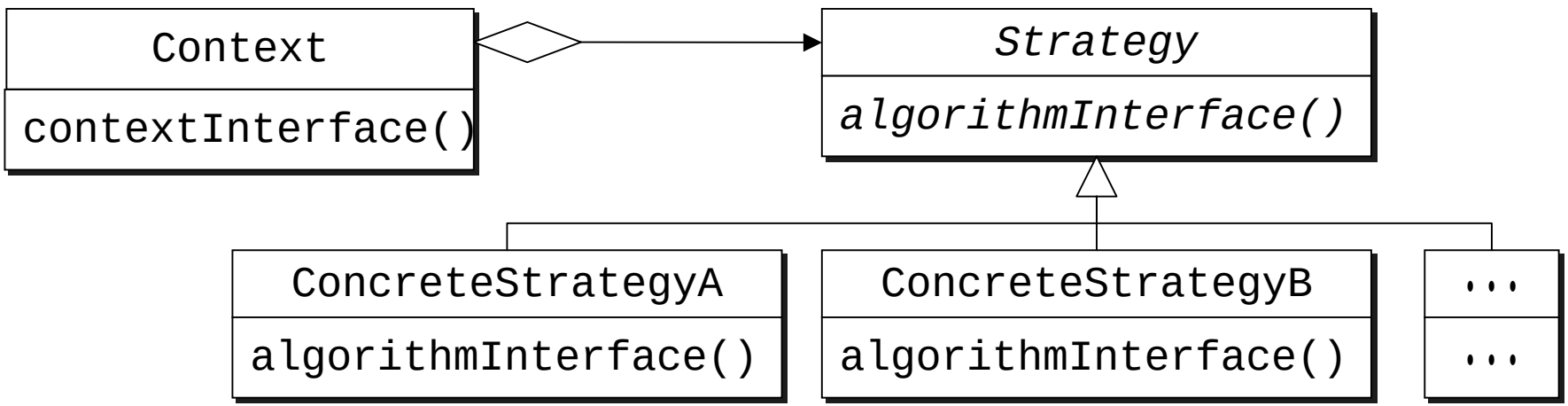
例: Strategy (1/3)[GOF00]

- 状況: 複数のアルゴリズムが存在する処理を実現するオブジェクト指向設計・実装を行っている。
- 問題: 高い保守性を持ちつつアルゴリズムを状況に応じて使い分けたい。
- フォース:
  - 複数のアルゴリズム集合を単一クラスに埋め込み、条件分岐により使い分ける設計が最もシンプルだが、アルゴリズムの追加や修正は困難になる。
  - 異なるアルゴリズムをそれぞれ異なるクラスに埋め込めばアルゴリズムの追加や修正は容易だが、利用方法を共通化しない限り、クライアント側に影響が及ぶ。

[GoF00] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides（著）, 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 改訂版", ソフトバンクパブリッシング, 2000.

例: Strategy (2/3)

- 解決: 振る舞いのカプセル化＋ポリモーフィズムによる変更
  - 複数のアルゴリズムを共通に利用する１つの方法を定め、
  - 抽象クラス（or インタフェース）に方法のみを抽象メソッドとして定義し
  - 具体的なアルゴリズムの実装は同抽象クラスを継承/実装した個別の具象クラスにおいて、抽象メソッドを継承した具象メソッド内に記述する。



Strategy(3/3):例への適用結果

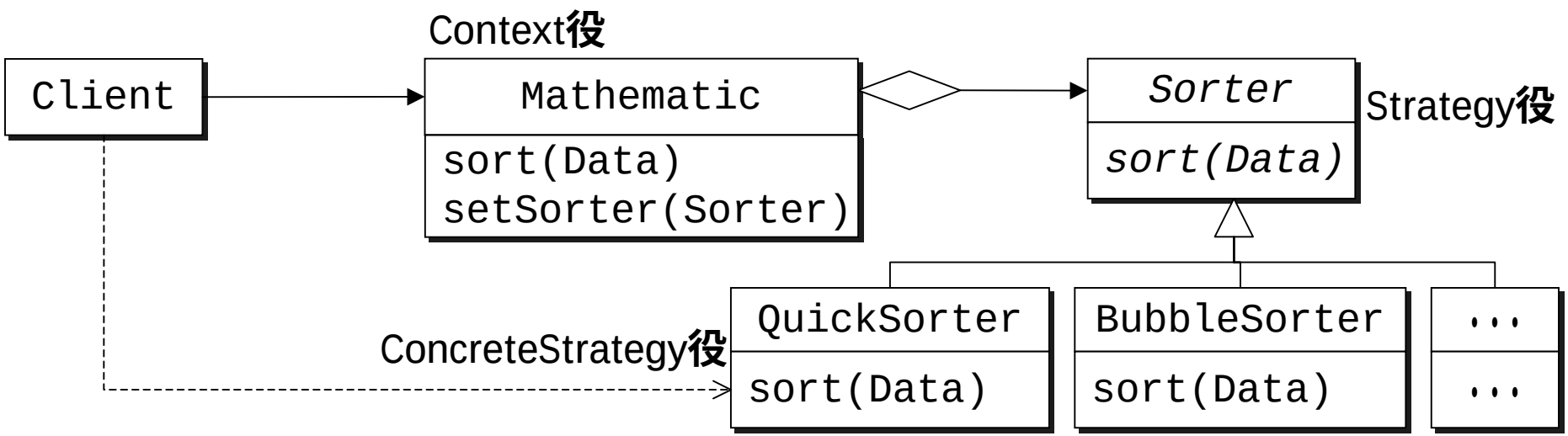
```
class Client {
 Mathematic math;

 void init() {
  math.setSorter(
    new QuickSorter());
 }
 void calc() {
  data = math.sort(data);
}
```

```
class Mathematic {
  Sorter sorter;
  public Data sort(Data data){
    return sorter.sort(data);
  }
```

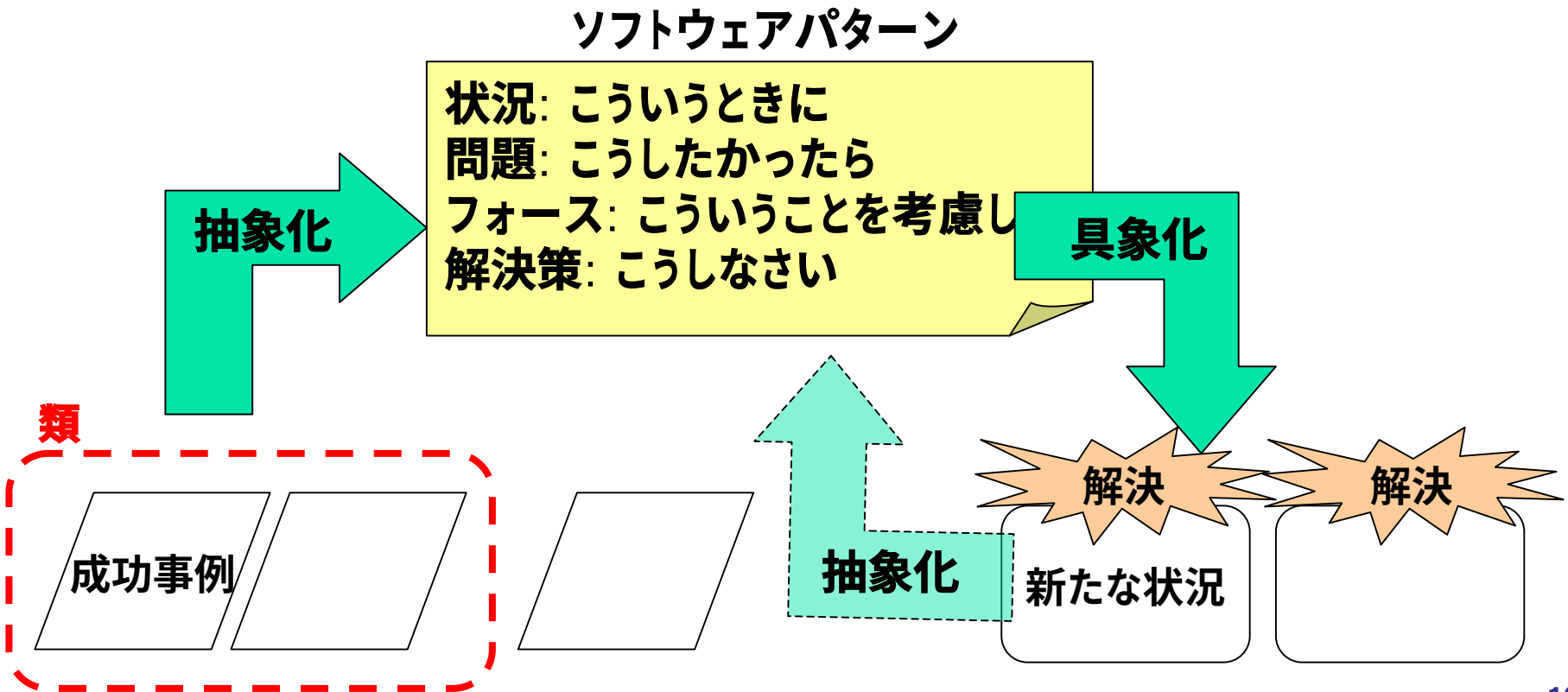
```
abstract class Sorter {
  public abstract Data sort(Data);
```

```
class QuickSorter {
  public Data sort(Data) { ... }
```



ソフトウェアパターンと抽象化/具象化

- 抽出: 開発経験を抽象化してソフトウェアパターンを得る
- 利用:
  - 現実の状況を暗黙に抽象化して、検討中のパターンが扱う状況に適合することを確認する
  - パターンの解決策を具象化して現実の状況に適用する



ソフトウェアパターンの３つの効果

- (1) 再利用: 熟練者の思考過程と成果を一貫して再利用できる
  - 一から考える必要が無い
  - 同じ問題は同じように解決 ⇒ **一貫性のある成果物、高生産**
  - ある状況で特定の問題の解決に成功した解法は、類似の状況で似たような問題の解決に成功する確率が高い

  ※ただし、同じような状況で、同じ過ちを犯す可能性も高い
    - アンチパターン＝共通の失敗事例＋回避策
- (2) 対話: 意思疎通を促進できる
  - パターン名を共通の語彙としてチームで共有しておけば、いちいち内容を伝えるよりも、パターン名を一言伝えた方が効率が良い

  ※類似の概念: データ構造・アルゴリズム、例: 「クイックソート」
- (3) 学習: 優れたソフトウェアの解読を促進できる
  - 優れたソフトウェア（ライブラリなど）には、既知のパターンが適用されている
  - 適用されているパターンが分かれば、元の設計意図を容易に推測し、その意図に沿って活用できる
  - 背景にある技術（例えばオブジェクト指向）への理解を深められる

デザインパターン

- プログラム／クラス設計工程において、典型的な「優れた」設計を導出する過程を、パターンとして表したもの
  - 数個のクラスを対象とするようなソフトウェアの部分的な設計における問題を解決する
  - ソフトウェアの再利用性や柔軟性を高めるために、開発者が設計やコーディングを何回も繰り返した結果得られたもの
  - 標準的なオブジェクト指向を知っていれば実装できる
  - 開発者同士の意思疎通をスムーズにする
- 代表的なデザインパターンカタログ
  - 汎用型
    - オブジェクト指向言語で実装するということ以外には、特に実装方法や処理内容に依存しないもの
    - GoF（Gang of Four: 著者の4人組）デザインパターン [GoF00]
    - [POSA00][J2EE02][PofEAA03] に収録されたデザインパターン
  - 特化型
    - PofEAA（データベースアクセス処理の設計）[PofEAA03]
    - Doug Leeらの並行性パターンを整理した結城のマルチスレッドパターン集（Java言語による並行性処理の設計）[結城02]

GoFパターンカタログの構成

範囲		可変性をもたらす側面		
		生成	構造	振る舞い
	クラス	Factory Method サブクラスでオブジェクト生成させる	Adapter（クラス） Bridge（クラス） クラス構成に継承を利用	Interpreter Template Method 振る舞いの定義に継承を利用
	オブジェクト	Abstract Factory Prototype Singleton オブジェクトでオブジェクト生成させる	Adapter（オブジェクト） Bridge（オブジェクト） Flyweight Proxy Decorator Facade オブジェクト構成に委譲を利用	Chain of Responsibility Command Iterator（オブジェクト） Mediator Memento Observer State Strategy 複数のオブジェクト協調による振る舞いの定義
	混成	Builder	Composite	Iterator（混成）

デザインパターンのまとめ

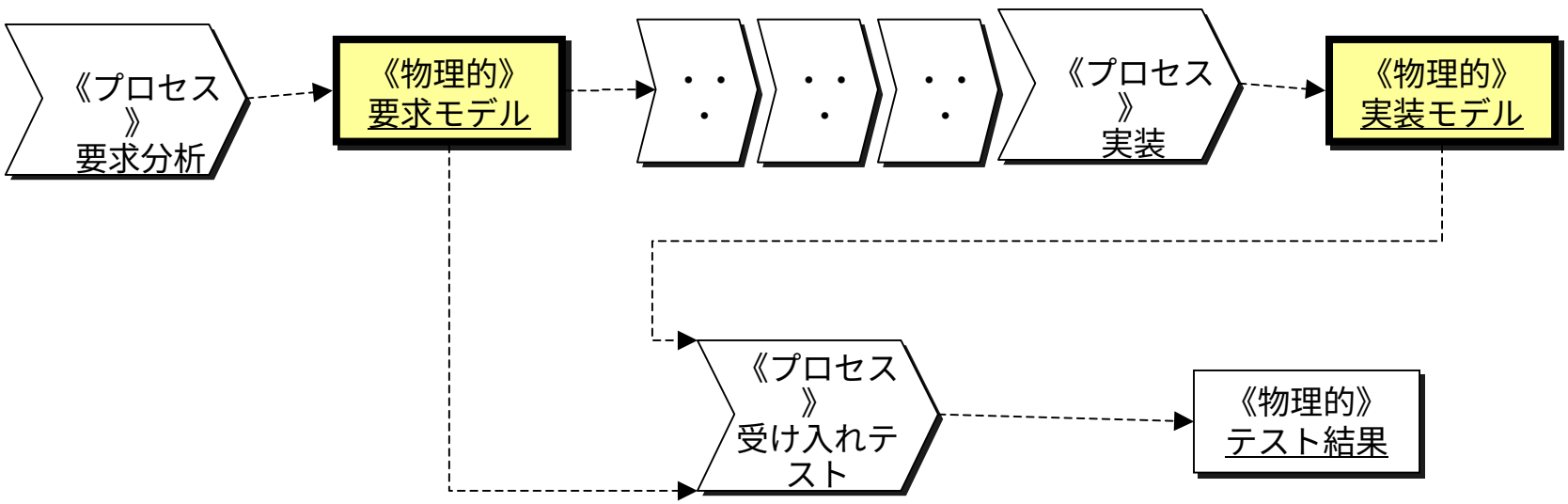
- 最低限、２３のGoFデザインパターンは抑えておこう
  - オブジェクト指向設計における必須のボキャブラリ
    - [オブジェクトハンドブック] 永和システムマネジメント, オブジェクトハンドブック2001, http://www.objectclub.jp/technicaldoc/object-orientation/handbook
  - 拡張性・可変性をもたらす。しかし、メリット・デメリットを理解すること。
  - 代表的な分析パターン、アーキテクチャパターンも抑えておきたい
- 情報源、コミュニティに飛び込もう
  - 鷲崎, 太田, よく分かるソフトウェアパターン, 日経ITPro, http://itpro.nikkeibp.co.jp/article/COLUMN/20061106/252691/?ST=develop&P=1
  - パターンの拡充はオブジェクト指向コミュニティの発展と共に
  - 情報処理学会ソフトウェア工学研究会パターンワーキンググループ

    http://patterns-wg.fuka.info.waseda.ac.jp/

ソフトウェアテスト

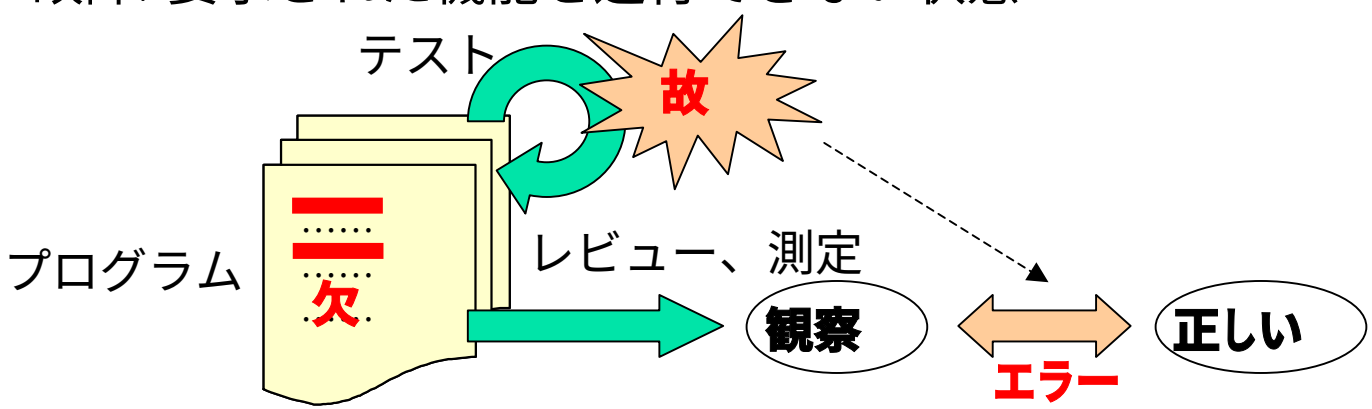
V&V: 検証と妥当性確認

- 検証（Verification）
  - 正しくソフトウェアを作れているのか
  - ある段階・工程の入力と出力のつき合わせ
- 妥当性確認（Validation）
  - 正しいソフトウェアを作れているのか
  - 最初の要求と出力のつき合わせ

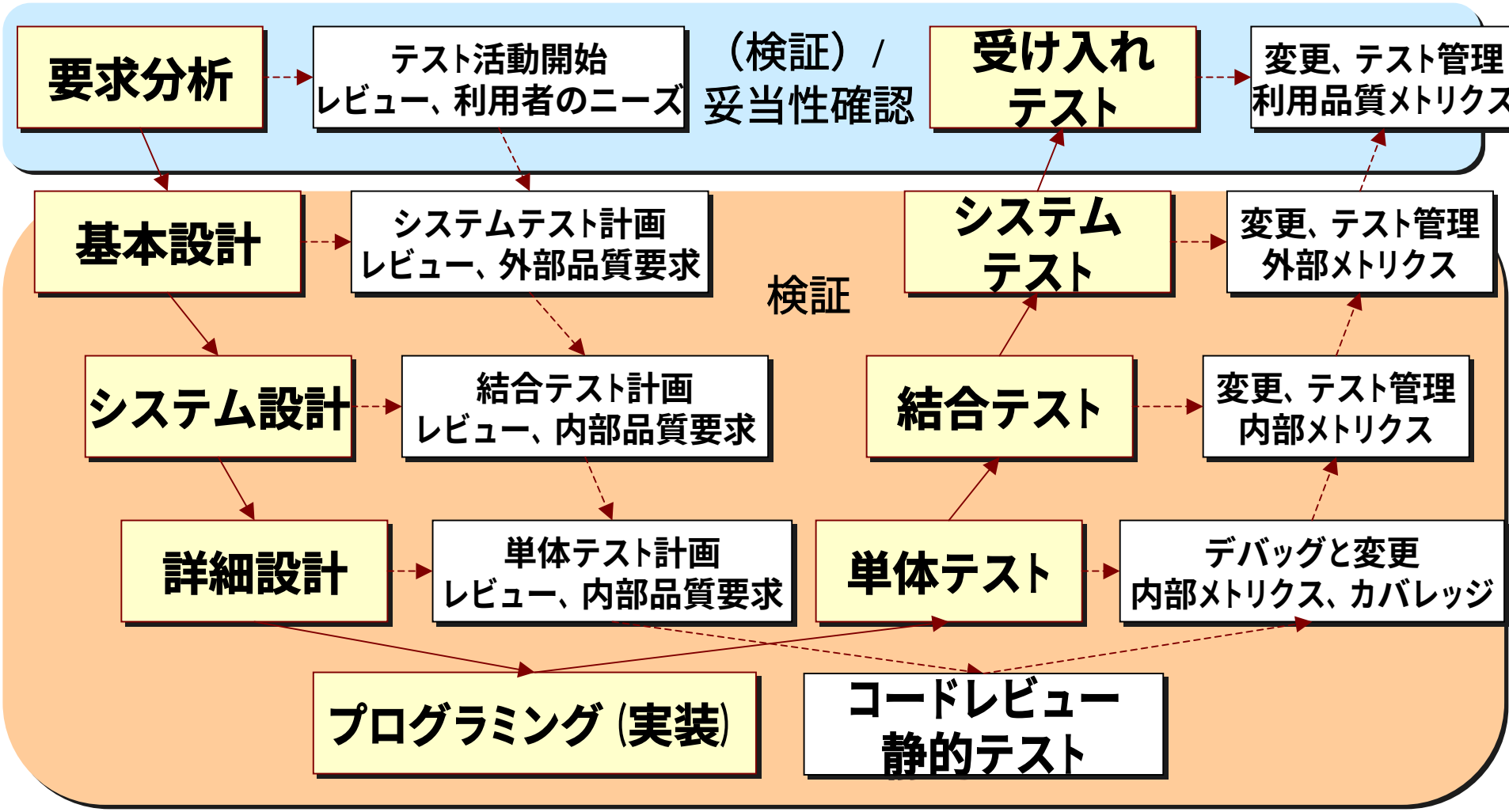


テストとデバッグ

- テスト
  - ソフトウェアの欠陥を見つける目的でソフトウェアを実行し、故障を起こさせる
  - よいテスト = 少ないテストケースで多くの欠陥検出
- デバッグ
  - テストの結果としておきた故障の直接要因となった欠陥を見つけだして、修正
- 用語
  - エラー: 計算、観察、測定された値や条件と、指定もしくは理論的に正しい値や条件との相違
  - 障害、欠陥、バグ: プログラムの間違ったステップ、プロセス、データ定義
  - 故障: 要求された機能を遂行できない状態



W字モデルへ [Spllner00][森06]

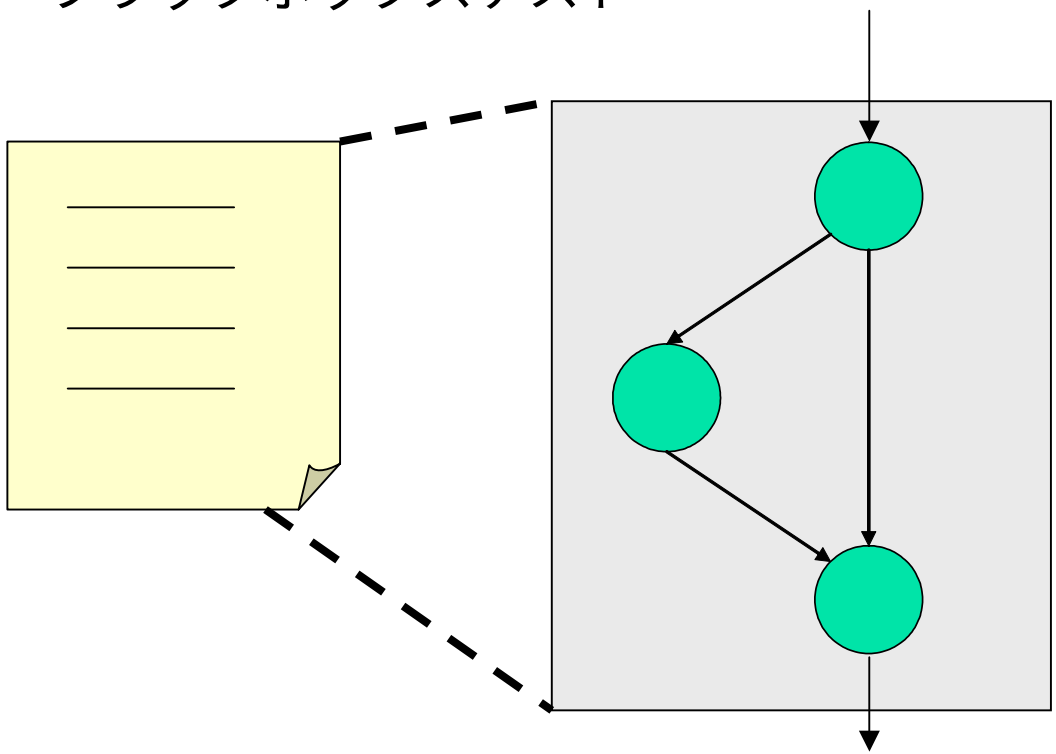


[Spillner00]A. Spillner: From V-Model to W-Model – Establishing the Whole Test Process, 4th Conference on Quality Engineering in Software Technology, 2000.

[森06]森ほか: ソフトウェア品質技術の開発と適用, 東芝レビュー, 61(1) , 2006.

テスト技術の種別

- ブラックボックス
  - テスト対象の要求や仕様に基づいてテスト
- ホワイトボックス
  - テスト対象の内部パス、構造、実装に基づいてテスト
- グレーボックス
  - テスト対象の実装をある程度確認した上で、より高精度なブラックボックステスト

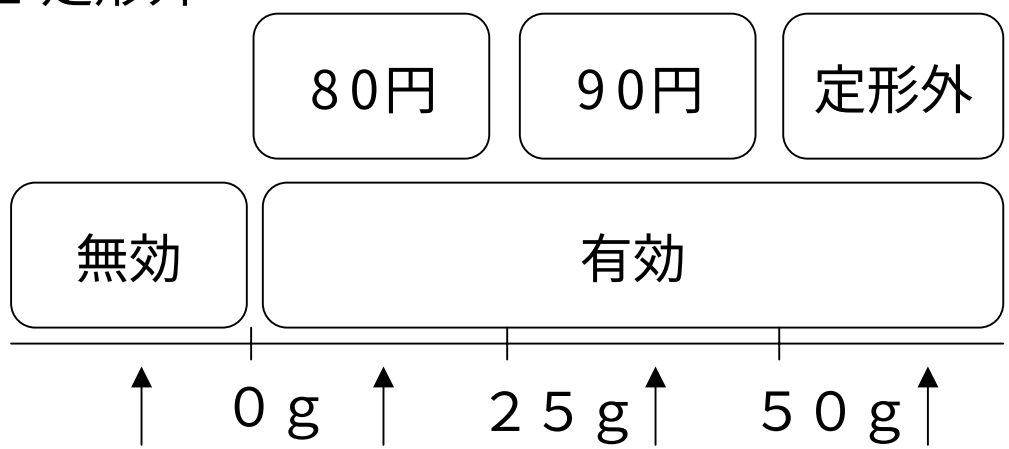


ブラックボックステストの種類

- 入力１つ
  - 同値クラステスト → 境界値テスト
  - エラー推測
- 入力の組み合わせ: 同値・境界値分割の上で
  - 汎用: デシジョンテーブルテスト
  - 直交的: ペア構成テスト（直交表、オールペア法）
  - 依存関係あり: ドメイン分析テスト
- 特別な仕様
  - 状態遷移テスト
  - ユースケーステスト

同値クラステスト（同値分割）

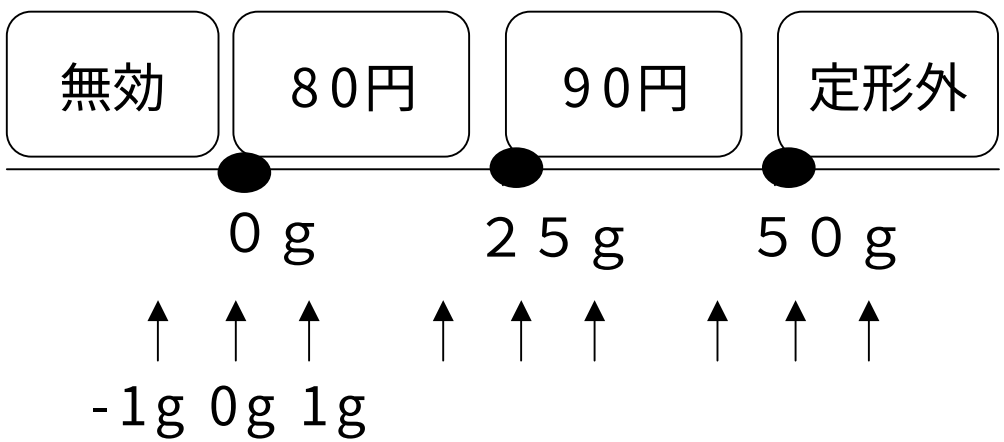
- 仕様から同値クラス群を識別し、各同値クラスごとに代表値（例えば中央値）および期待出力からなるテストケースを作成してテスト
- 同値クラス: 同じ結果をもたらすと期待される入力値の集合。境界値分析のもと。
  - 有効値クラス: 有効な入力値の集合。仕様に応じて細分化
  - 無効値クラス: 無効な入力値の集合。一般に有効値クラスの上下（前後）。
- 例: 郵便料金の計算、ただし整数のみ
  - 25g未満 80円
  - 25g以上50g未満 90円　　-10g, 13g, 37g, 60g
  - 50g以上 定形外



境界値テスト（境界値分割）

- 同値クラスの境界をとりあげるテスト
  - 同値クラスの境界値
  - 境界値の上下の値（たいてい他の同値クラスに属する）
- 欠陥はしばしば境界で作りこまれる
  - 条件分岐、ループ・・・
- 例: 郵便料金

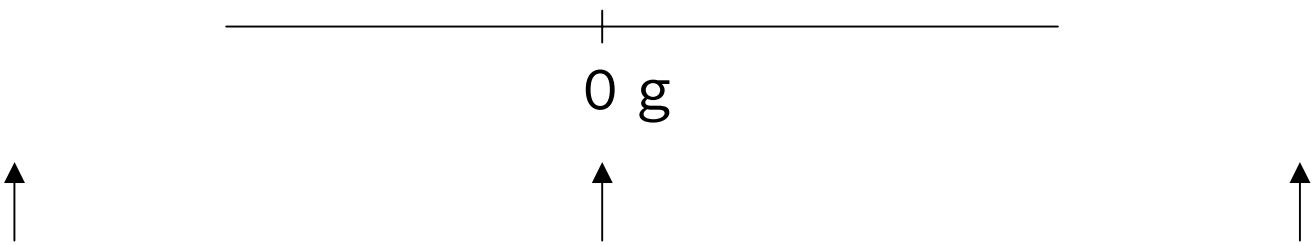
-1g, 0g, 1g, 24g, 25g, 26g, 49g, 50g, 51g



エラー推測

- 考えられる入力値の中で、経験的に欠陥の原因となりそうな特殊な値にもとづいてテスト
- タイプ上の最小値や最大値、デフォルト値、特殊値
  - 整数: 0, -32768, 32767（16bit）, -2147483648, 2147483647（32bit）
  - 文字列: “”（空文字列）
  - ポインタ、オブジェクト: NULL
- 例: 郵便料金

-32768g, 0g, 32767g



組み合わせの必要性と難しさ

- 同値テスト、境界値テスト、エラー推測は、入力の種類が単一であればＯＫ
- しかし、実際には組み合わせが必要
  - 入力値を複数とる機能
  - ソフトウェア部品・サブシステムの組み合わせ
- 組合わせ時のポイント
  - 全ての組み合わせのテストは現実に不可能なので（例: 2^57）
  - 漏れが無いことが重要
  - 無駄が無いことが重要

どのように？

デシジョンテーブル（決定表）

- 入力の条件と実施すべき処理の組み合わせを整理した表
  - 条件: 真 Y、偽 N
  - 処理: 実行する X、しない -
  - 行と列を入れ替えることもある
- テスト対象がメソッドや関数の場合は
  - 条件: 引数の値
  - 処理: 期待する戻り値や、別の何らかの出力

	テストケース番号	1	2	3
条件	条件1	Y	Y	
	条件2	Y	Y	
	条件3	Y	N	
処理	処理1	X	-	
	処理2	-	X	

	条件			処理	
テストケース番号	1	2	3	1	2
1	Y	Y	Y	X	-
2	Y	Y	N	-	X
3					

テストケース	w	h	int size(w,h)
1	0	0	0
2	5	10	50

デシジョンテーブルテスト

- 特定観点に基づいてデシジョンテーブルを作成しテスト
  - 1. 各入力の種類について、同値分割・境界値分割により代表値・境界値の集合を得る
  - 2a. 有効値のテスト: １つの入力の種類について有効値クラスの代表値で固定し、他の入力の種類について無効値も含めてあらゆる代表値・境界値を網羅
  - 2b. 無効値のテスト: 2aをベースとした上で、１つのテストケースにつき、いずれかの入力で無効値クラスが登場するように変更

テストケース	w	h	int size(w,h)
1	-10	10	エラー
2	-1	10	エラー
3	0	10	0
4	1	10	10
			・・・
	10	-10	エラー
	10	-1	エラー
			・・・

a	b	c	処理結果
-10	10	10	エラー
-1	10	10	エラー
0	-10	10	エラー
1	10	-10	エラー
			・・・

組み合わせの網羅性

- 再考: 全ての組み合わせテストは不可能
- Kuhnによる報告 [Kuhn04]
  - 多くの欠陥は少数のパラメータの組み合わせで顕在化
  - 2つのパラメータで70%、3つで90%

⇒ パラメータの全ペアについて全ての値の組み合わせをテストし、必要に応じて集中的に3つの組み合わせを併用すると良い

FTFI: 欠陥に関係しているパラメータ数

FTFI	NASAエキスパートシステム	POSIXモジュール群	組込み	Mozilla	Apache
1	61	82	66	29	42
2	97		97	76	70
3			99	95	89
4			100	97	96
5				99	96
6				100	100

[Kuhn04] Richard Kuhn, “Software Fault Interactions and Implications for Software Testing,” IEEE Transactions on Software Engineering, Vol.30, No.6, 2004

組み合わせテスト

- 入力の種類が多数ある中で、全てのn個の構成について、それらの全ての値の組み合わせをテストする
  - n因子間組み合わせ網羅率１００％
  - 因子: 入力、パラメータの種類
  - 水準: 因子の取りうる値
- 特に n = 2 の場合を All-Pair法、ペアワイズテストと呼ぶ
  - 2因子間組み合わせ網羅率１００％
- All-Pair法の実現法
  - 手作業による表の作成
  - 既存の直交表の利用
  - ツールによる自動生成 http://www.pairwise.org/
    https://sourceforge.jp/projects/pictmaster/
    （AETG, TConfig, AllParis, PICT, PictMaster など）

直交表とは [吉澤07]

- 2因子間組み合わせ網羅率100%かつ2因子の水準の同組み合わせが均等に同回数出現する組み合わせ表
- 直交配列 Orthogonal Array OA(N, m, s, t) : 全 S × S の順序対が λ 回数出現する行数N, 列数mの行列
  - S = { 0, 1, ... , s-1 }, S×S = { 0×0, 0×1, ... , (S-1)×(S-1) }
  - N: 行数 λs^2　　s: 水準数　　λ : λ 回数出現
  - m: 因子数　　t: 強度。t因子間組み合わせ網羅率100%
- 直交表とは、t=2 の直交配列
  - 例: L4直行表は OA(4, 3, 2, 2)

	因子		
テストケース	A	B	C
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

因子		
種別	モード	スイッチ
小	F	Off
小	T	ON
大	F	Off
大	T	ON

[吉澤07]吉澤, 相山, 仙石, “ソフトウェアテスト HAYST法 入門”, 日科技連, 2007.

ブラックボックステストにおける直交表の強み

- 全体が均衡（ムラが無い）
- 3因子間組み合わせ網羅率も高い: 50～90%以上

⇒

- ペアワイズ法
  - 「同回数出現」の制約を取れば、テストケースはもっと減らせる。1/10以下など。
  - しかし、3因子間組み合わせに極端な偏りの生じる可能性
  - ランダム生成後の選択、小さな直交表の横連結＋追加など

直交表によるテスト　　ある因子を固定しての全組み合わせテスト

	因子数	テスト数	2因子網羅率	3因子網羅率	2因子・組合わせ数	2因子・全テスト	3因子・組合わせ数	3因子・全テスト
L4	3	4	100%	50%	${}_3C_2$ = 3	3・4 = 12	${}_3C_3$ = 1	1・8 = 8
L8	7	8	100%	90%	21	84	35	280
L16	15	16	100%	96%	105	420	455	3640

(0,0) (0,1) (1,0) (1,1) の4通り

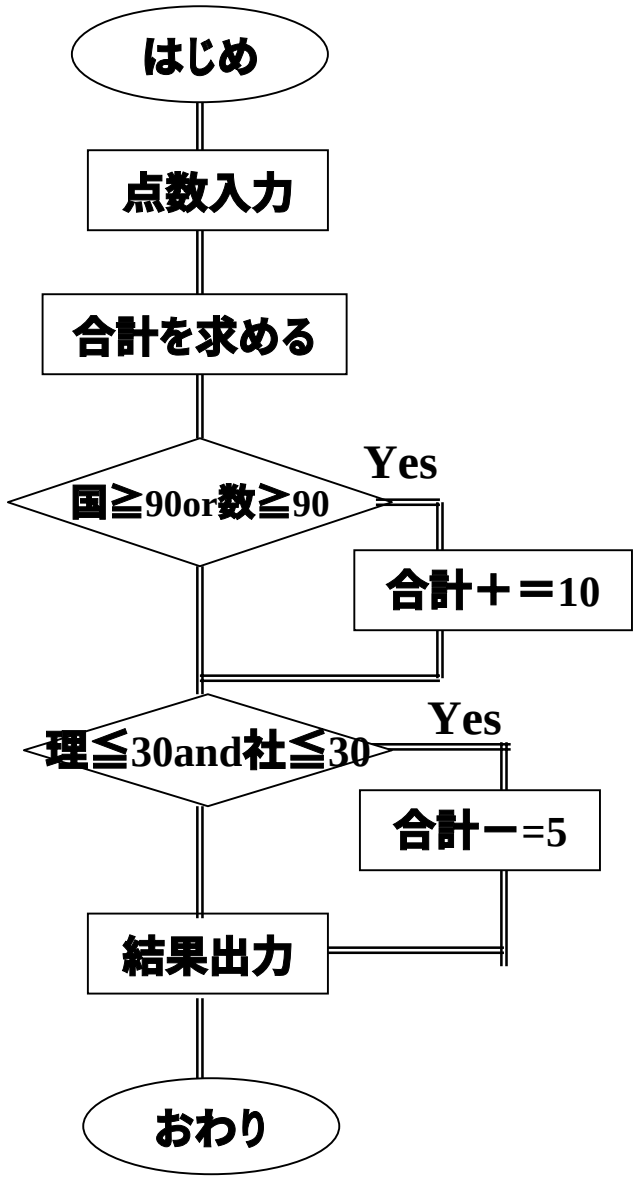
ホワイトボックステスト [Copeland05]

- テスト対象の内部パス、構造、実装に関する知識に基づいたテスト
  - パス: 関数・メソッド内、モジュール内、モジュール間、サブシステム間、システム間
  - 制御フローテスト
  - データフローテスト
- 利点
  - 内部の構造、特にパスに基づく「最終確認」
  - 内部の制御フローやデータフローの誤りを精密に特定
- 欠点
  - フローが仕様に基づいて正しく実装されていることが前提
  - 全パスの網羅は非現実的
  - プログラミングスキルの要求

[Copeland05] Lee Copeland **著**, **宗雅彦 訳**, “**はじめて学ぶソフトウェアのテスト技法**”, 2005

制御フローテストと網羅率（カバレッジ）

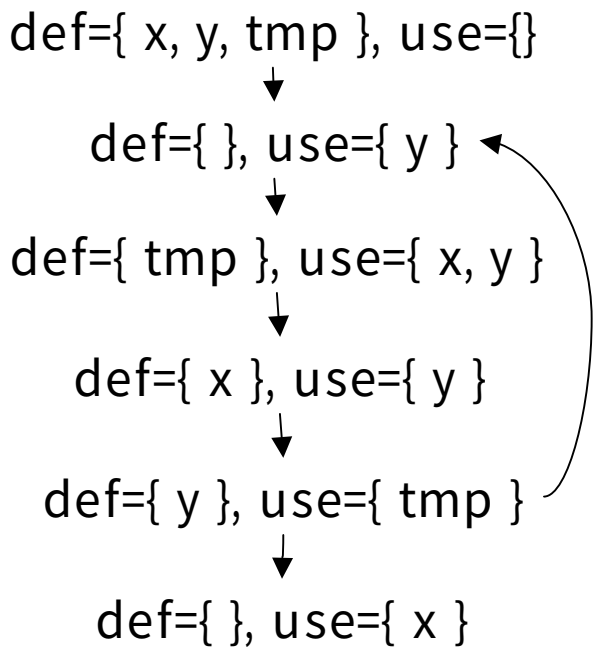
- モジュール内の実行パスを識別して、特定の網羅性（カバレッジ）を達成するようにテスト実行
- 命令網羅
  - すべての命令を少なくとも１回は実行
- 分岐網羅
  - 分岐に対して、真・偽の双方をとるように実行
- 条件網羅
  - 分岐条件中の各条件が真・偽の双方をとるように実行
- 分岐／条件網羅
  - 分岐網羅＋条件網羅
- 複合条件網羅
  - コンパイラ・処理系による評価パスの網羅
- 全パス網羅
  - あらゆるパスの網羅
  - 無限ループを制限するバリエーションあり



データフローテストとDU網羅

- データフローテスト: 変数ライフサイクルの適切さを確認
  - 各変数の定義(def)、使用(use)、消滅(kill)のパターン
  - ○: du, uu, uk, kd
  - △: ud, dd, dk
  - ×: ku, kk
- DU網羅
  - 全変数の定義-使用の関係（DUチェーン）を網羅
  - 分岐網羅を保証しない。ただし、if(...) then { 変数定義なし } のような特殊なケースのみを網羅しない。

```
public int method(int x, int y) {
  int tmp;
  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }
  return x;
}
```



まとめ　「テストに王道なし！」

- 完全なテストはできない
  - 少ない労力でたくさんテスト
- うまいテストの方法はない
  - テストの分類を覚えておき、上手く組み合わせていく
- サポートツールを上手く使ってテスト

アジャイルプロセス:
XPを中心として

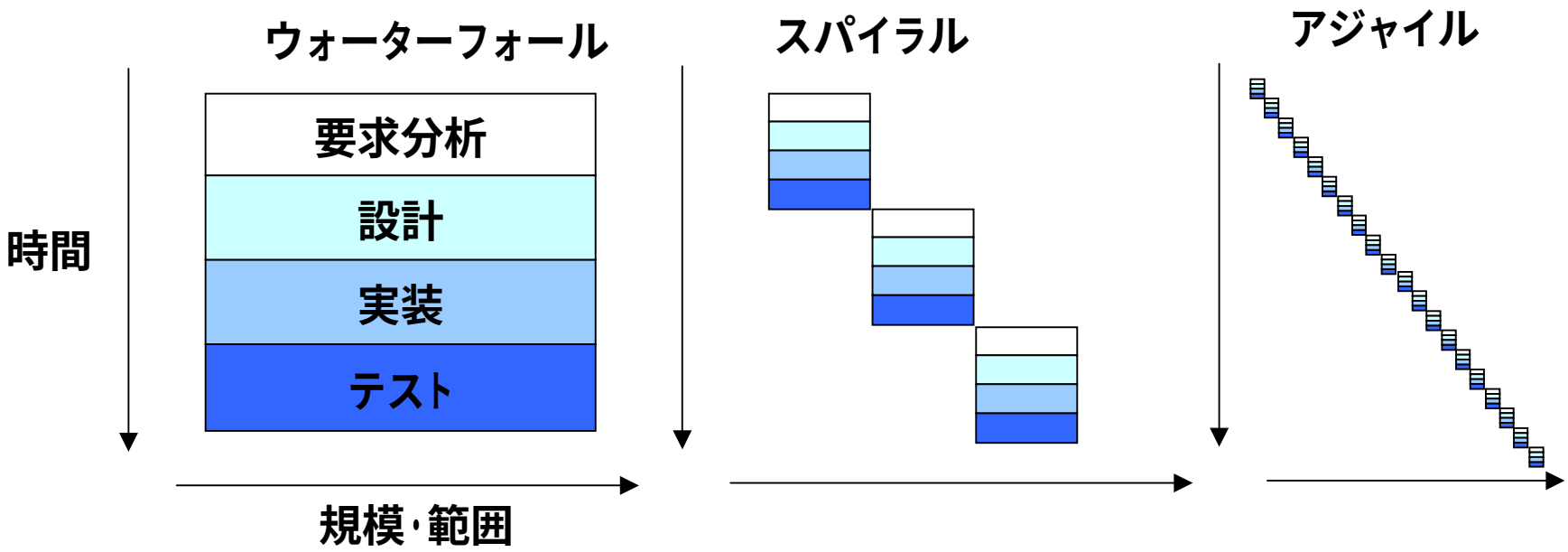
アジャイル宣言

http://www.agilealliance.org/

- *プロセスやツールよりも*、人間と人間関係を重視する
- *ドキュメントよりも*、動くソフトウェアを重視する
- *契約交渉よりも*、顧客との協力を重視する
- *計画に従うよりも*、変化に対応することを重視する

アジャイル開発プロセス

- イテレーション単位での堅実な小規模リリースと、変更への俊敏な対応を重視したスパイラル型プロセス
  - 極小のイテレーションの繰り返し
  - 例: eXtreme Programming (XP), FDD, Scrum



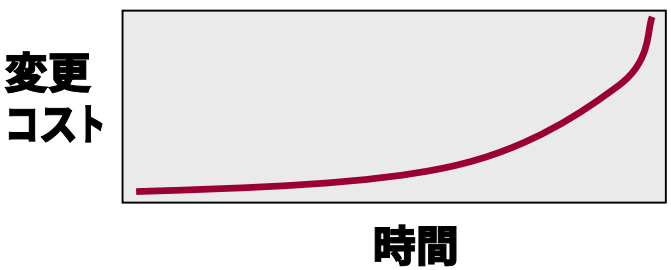
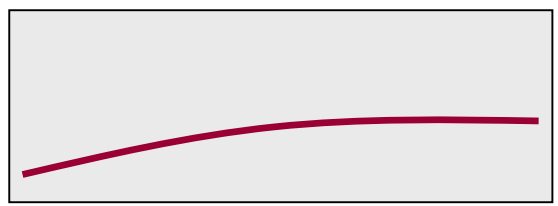
出典 [Nagase02]

- [Nagase02] **長瀬嘉秀**, **開発方法論と開発プロセス**, @IT, 2003, http://www.atmarkit.co.jp/im/carc/serial/column/nagase02/nagase02.html

eXtreme Programming（XP）とは

- 12 (あるいは 25) の実践項目から構成されるアジャイル開発プロセス
  - Kent Beck氏が1999年に発表
  - 破綻していたC3プロジェクト(ダイムラークライスラー社の人事・給与管理システム開発) を成功へ
  - “究極” ではない → “極端”
- 特徴
  - 軽量、俊敏: 取っ掛かり易い
  - 迅速に価値あるものを追求
  - 品質を最重視
  - 機械から人間へ

特徴の比較

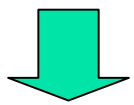
	従来開発手法	XP
開発規模	中・大規模	小・中規模 ※
開発の流れ	順番に	徹底して繰返し
開発期間	長い	短い
要員の構成	分業	密に連携
管理方針	トップダウン （管理ありき）	ボトムアップ （現場ありき）
変更コスト	文書ありき、変更困難 	テスト・実装ありき、変更容易 

部分的には多人数・大規模開発において有効に機能した事例もあり **[Andersson00]**

[Andersson00] Even-Andre Andersson and Lars-Goran Andersson, XP and large distributed software projects, Proc. XP2000, 2000

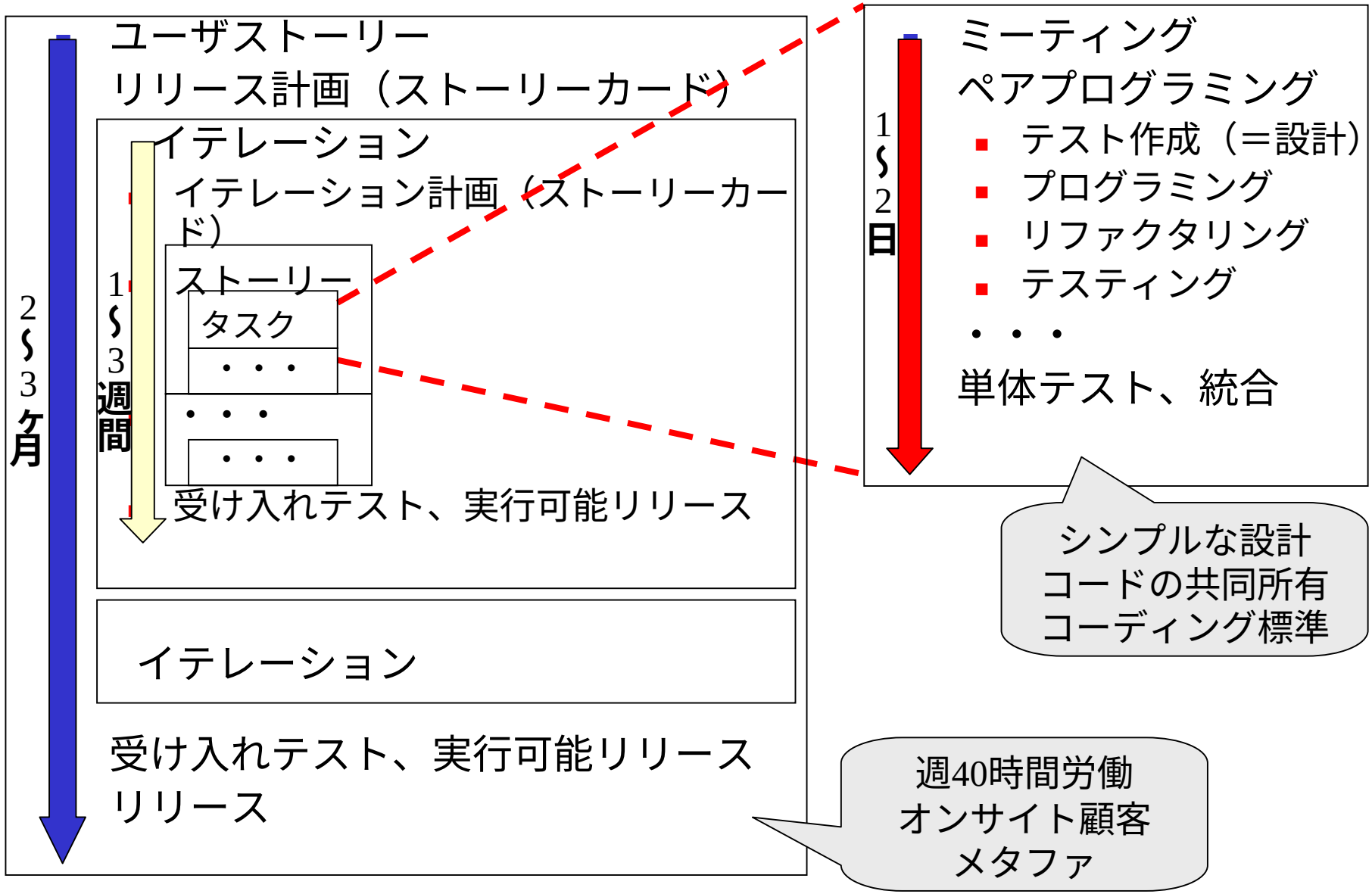
４つの価値: もっとも大切にすべきこと

- コミュニケーション
  - 会話が基本
- シンプルさ
  - 実現するためのもっともシンプルな方法
- フィードバック
  - 車の運転、細かな軌道修正の繰返し



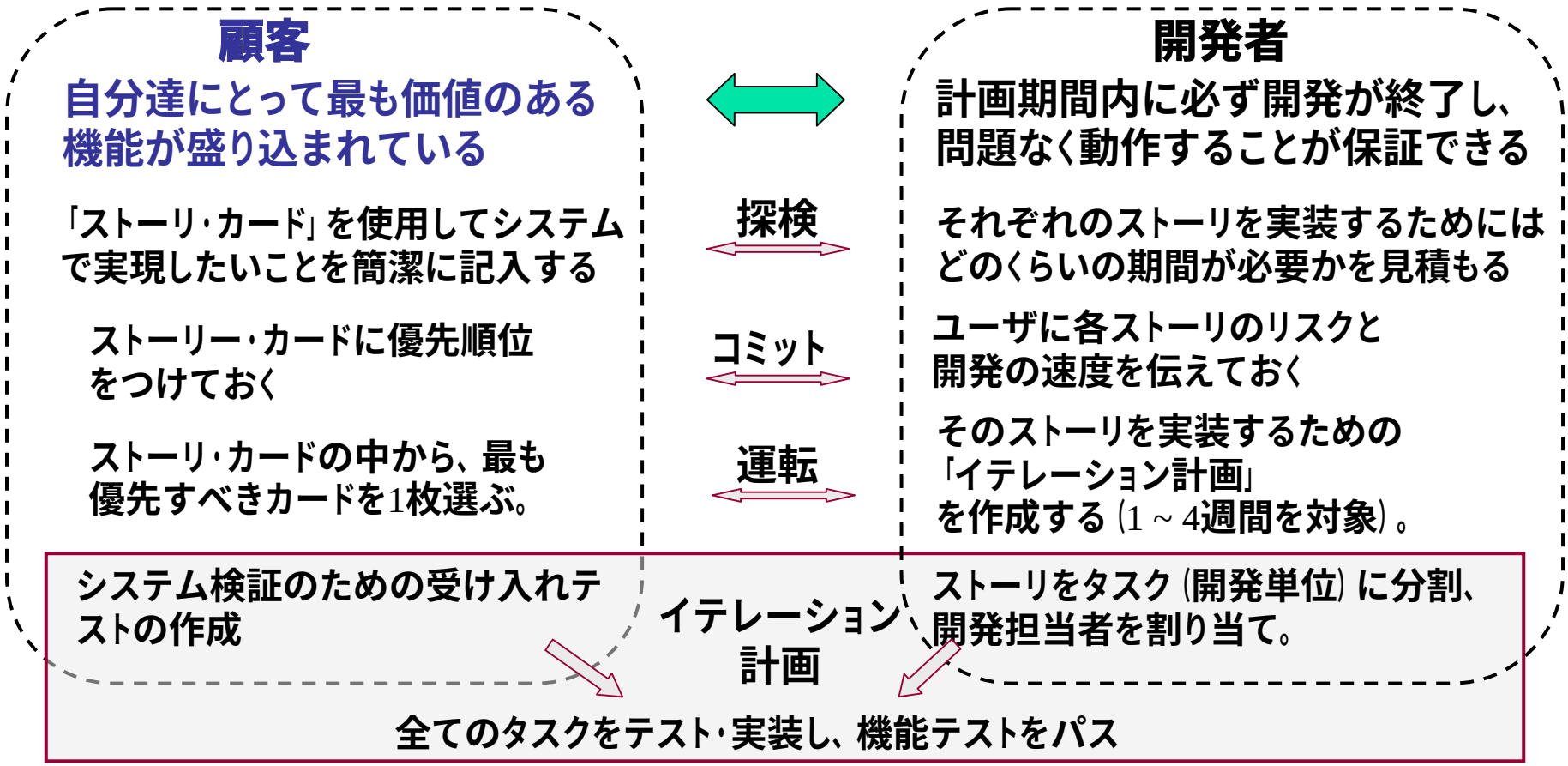
- 勇気
  - コミュニケーション + シンプルさ + フィードバック

XPによる典型的な開発



計画ゲーム (The Planning Game)

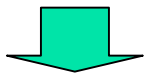
- ビジネス優先度と技術的見積により次回リリースの範囲を早急に決める



イテレーションは完了、システムのリリース

ペアプログラミング (Pair Programming)

- テストやコーディング、リファクタリングを
  一台のマシンを用いて二人で作業にあたる
  - ピアレビュー技法の一種
  - 自分以外のもう一対の眼
    - 間違いに気づきやすい
  - 開発者間の対話
    - 議論が必要な時にすぐ議論できる

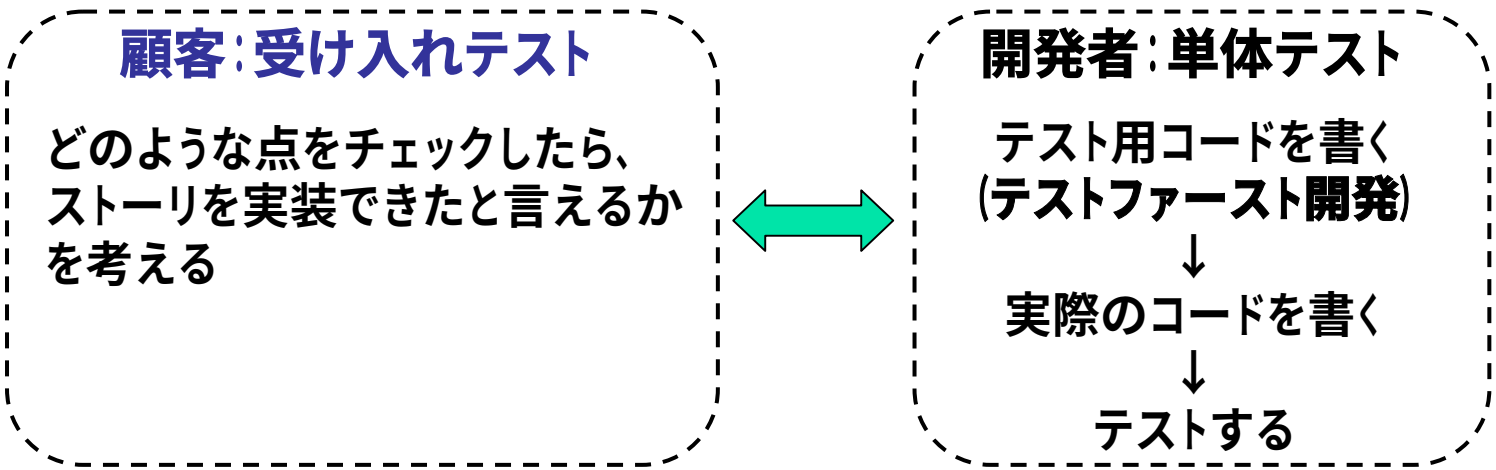


- **ソフトウェアの品質向上**
- **開発者同士のコミュニケーションの活性化**
- **教育効果が高い**
- **シンプルな設計がしやすくなる**
- **例:ペアが別個に作業する場合よりも工数**15%**増大したが欠陥率を**85%**に下げた** [Cockburn00]

[Cockburn00] Alistair Cockburn and Laurie Williams, The Costs and Benefits of Pair Programming, Proc. XP2000, 2000

テスティング (Testing)

- 開発者は継続的に単体テストを書く
  顧客は受け入れテスト（のシナリオ）を書く
  - テスト駆動による単体テスト
    - テスティングフレームワークの利用: JUnit、CppUnit、NUnit
    - テストファースト/駆動開発: まずテスト⇒ 設計技法の一種
    - 効果: 生産性変えず40～50%まで欠陥率低下 [Janzen05]
  - 受け入れテスト
    - 実現を期待するシナリオを記述。テストコードは開発者が記述。
    - ツール: FitNesse、xUnitなどの活用

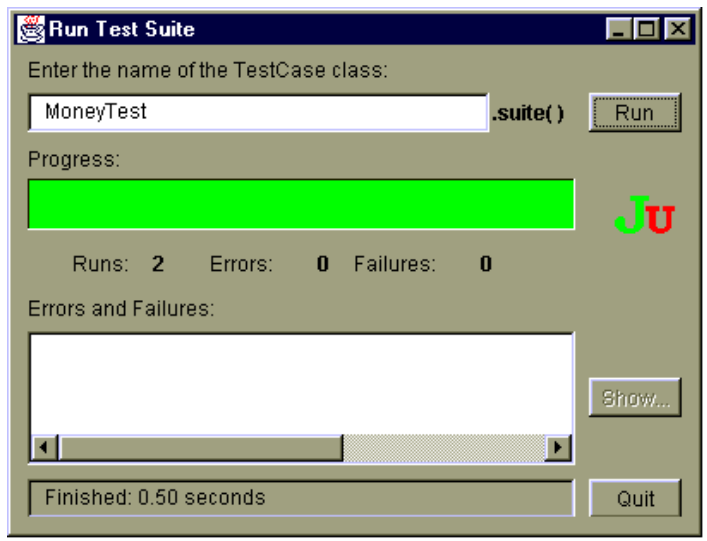


[Janzen05] David Janzen and Hossein Saiedian, Test-Driven Development: Concepts, Taxonomy, and Future Direction, IEEE Computer, Vol.38, No.9, 2005

例1: タスク・設計

- タスク「入力された10進数を2進数で表示する」
  - 入出力は標準的なコンソール
  - 使用言語はJava
  - テスティングフレームワーク: JUnit

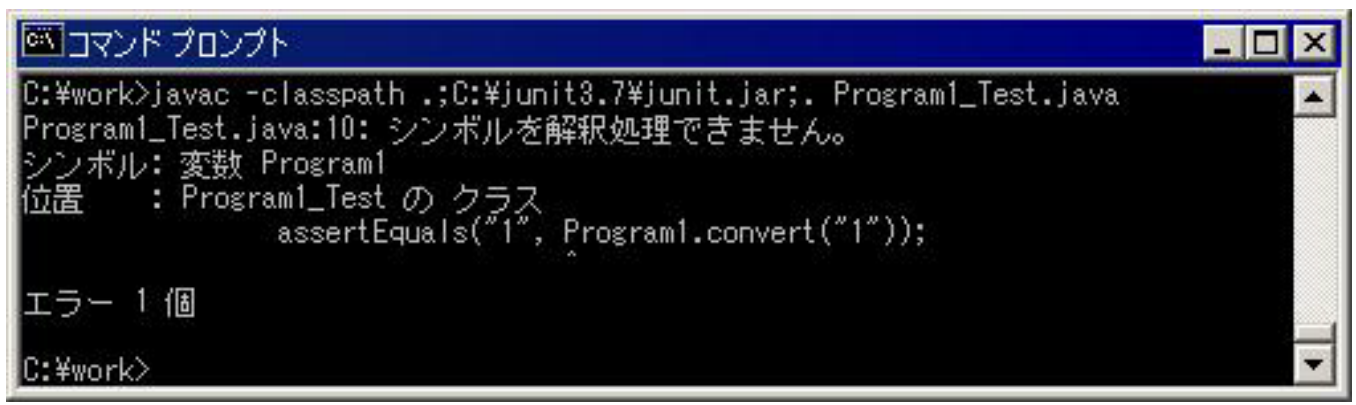
  （ http://www.junit.org/ から無料で入手可能 ）
- 与えられたタスクを実現する最もシンプルな設計をペアで話し合う
  - 対象クラスを Program1 とし，変換メソッドを， String を受け取って String で返す convert という名前のクラスメソッドとする
  - 標準入力を用いた入力機能のみを考慮



例2: まずはテスト作成

- Program1_Test.java を作成，コンパイルしたところ，クラス Program1 が存在しないためコンパイルエラー

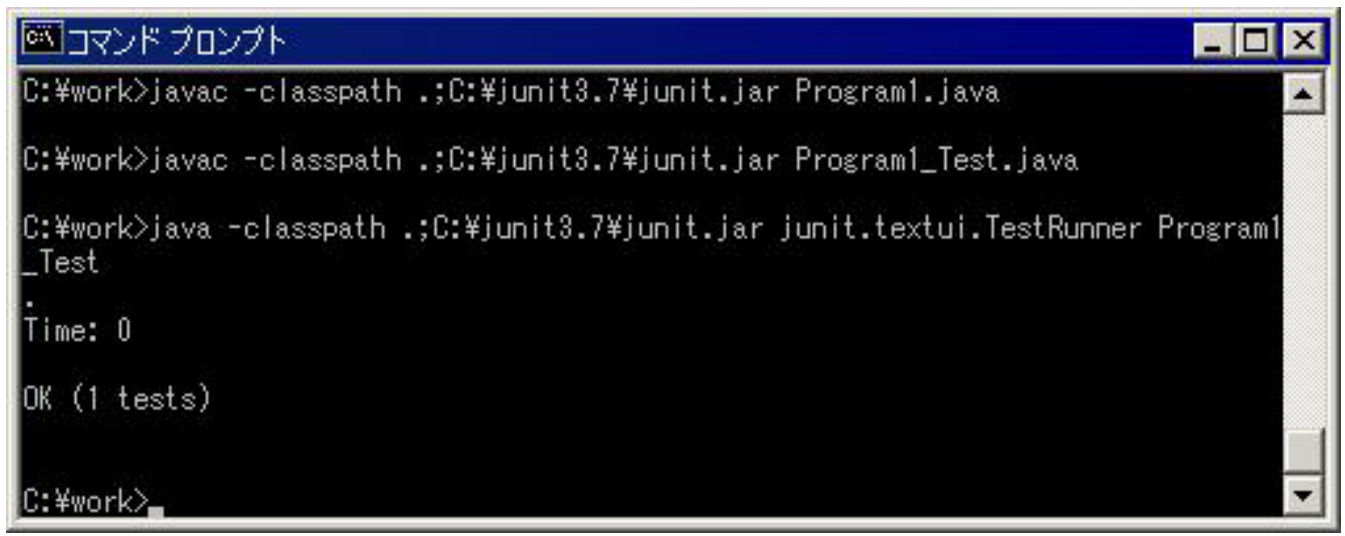
```
// Program1_Test.java
import junit.framework.TestCase;
public class Program1_Test extends TestCase {
  public Program1_Test(String name) {
    super(name);
  }
  public void testFirst() {
    assertEquals("1", Program1.convert("1"));
  }
}
```



例3: 実装

- テストを通る、もっともシンプルな実装を行う
- テストが通ることを“必ず”確認する

```
// Program1.java
public class Program1 {
  public static String convert(String value) {
    return "1";
  }
}
```



例4: ちょっとテスト、ちょっとコード

- 求められるタスクの、予想される入力と出力をテストとして追加
- テストを追加して、テストに失敗し、コードを作成: この繰返し

テストコード

```
// Program1_Test.java (抜粋)
public void testFirst() {
 assertEquals("1",Program1.convert("1"));
 assertEquals("10",Program1.convert("2"));
 assertEquals("1010",
  Program1.convert("10"));
}
```

①

実装コード

```
// Program1.java (抜粋)
public static String convert(
                    String value) {
 int temp = Integer.parseInt(value);
 String result = new String();
 while(temp > 0)
  int amari = temp % 2;
  result =
   String.valueOf(amari) + result;
  temp /= 2;
 }
 return result;
}
```

②

```
// Program1_Test.java (抜粋)
public void testFirst() {
 assertEquals("0",Program1.convert("0"));
  ……
```

③

```
// Program1.java (抜粋)
public static String convert(
                      String value) {
 int temp =Integer.parseInt(value);
 if(temp == 0) { return "0"; }
```

例5: 勇気をもってリファクタリング

- テストがあるからこそ、動いているコードを変更できる
- 重複コードの排除。コメントを入れたいときが良い機会

```
// Program1.java (抜粋)
private static final char[] DIGITS = { '0', '1' };
private static final char[] RESULTS = new char[8];

public static String convert(String value) {
  int temp = 0;
  try { temp = Integer.parseInt(value); }
  catch(NumberFormatException nfe) { processException();  }
  if( temp < -128 || temp > 127 ) { processException(); }
  StringBuffer result = new StringBuffer();
  for(int i = 0; i < 8; i++) {
    RESULTS[i] = DIGITS[(temp >>> (7 - i)) & 1];
  }
  return String.valueOf(RESULTS);
}

private static void processException() {
  System.err.println("入力は -128 ～ 127 の範囲です");
  throw new IllegalArgumentException("入力は -128 ～ 127 の範囲です");
}
```

```
RESULTS[i] =
 getCharByIndex(temp,i);
```

```
private static char getCharByIndex(int value, int index) {
 return DIGITS[(value >>> (7 - index)) & 1];
}
```

XPのまとめ

- XP2.0
  - Kent Beck, “XPエクストリーム・プログラミング入門: 変化を受け入れる”（第2版）, ピアソンエデュケーション, 2005
  - 価値の追加: 尊重。また、組織に応じて安全性など
  - 14の基礎プラクティス: 全員同席、チーム全体、情報豊富な作業空間、活気、ストーリー、1週間・四半期サイクル、ゆとり、ペアプロ・個人空間、10分ビルド、常時結合、テスト駆動、インクリメンタル設計
  - 11の応用プラクティス: 実顧客の参加、チームの継続・縮小、根本原因、スコープ契約、利用分払い、コード共有、コードとテストのみの保持、単一コードベース、インクリメンタル・日次配備
- 数多くの書籍・雑誌記事
  - XP Series（邦訳版あり）: 入門、導入、実行計画、実践記編、検証編、プログラミング・アドベンチャー、適用編、懐疑編など
  - 「eXtreme Programming テスト技法」翔泳社
  - 「eXtreme Programming 実践レポート」翔泳社
- 日本XPユーザグループ　http://www.xpjug.org/
  - セミナーや勉強会の開催、XP祭り
- eXtreme Programming FAQ
  http://www.objectclub.jp/community/XP-jp/
  - XP関連リンク集など
- アジャイルプロセス推進協議会　http://agileprocess.jp/
  - 企業が主体の活動

全体のまとめ: SWEBOKにみる位置

要求	設計	構築	テスティング	保守
要求の基礎的概念	設計の基礎的概念		テスティングの基礎的概念	保守の基礎的概念
要求エンジニアリングプロセス	設計における主要な問題	構築の基礎的概念	テストレベル	
要求の抽出	構造とアーキテクチャ		テスト技法	保守プロセス
要求分析	設計品質の分析評価	構築の管理	テストに関した計量尺度	保守における主要な課題
要求仕様	設計のための表記			
要求の妥当性確認				
実践上の考慮事項	設計戦略および手法	実践上の考慮事項	テストプロセス	保守のための技法

構成管理	マネジメント	プロセス	ツールおよび手法	品質
SCMプロセスのマネジメント	開始と範囲定義	プロセス実現および変更		品質の基礎的概念
構成の識別	プロジェクト計画	プロセス定義	ツール	
構成制御	プロジェクト実施	プロセスアセスメント		マネジメントプロセス
構成状態記録および報告	レビューおよび評価			
構成監査	終結		開発手法	
リリース管理および配布	計量	プロセス計量		実践上の考慮事項

ゴール指向分析

デザインパターン

テスト

アジャイル開発