形式仕様記述を巡る話題

酒匂 寛

オブジェクト倶楽部 2007 クリスマス

本日の結論

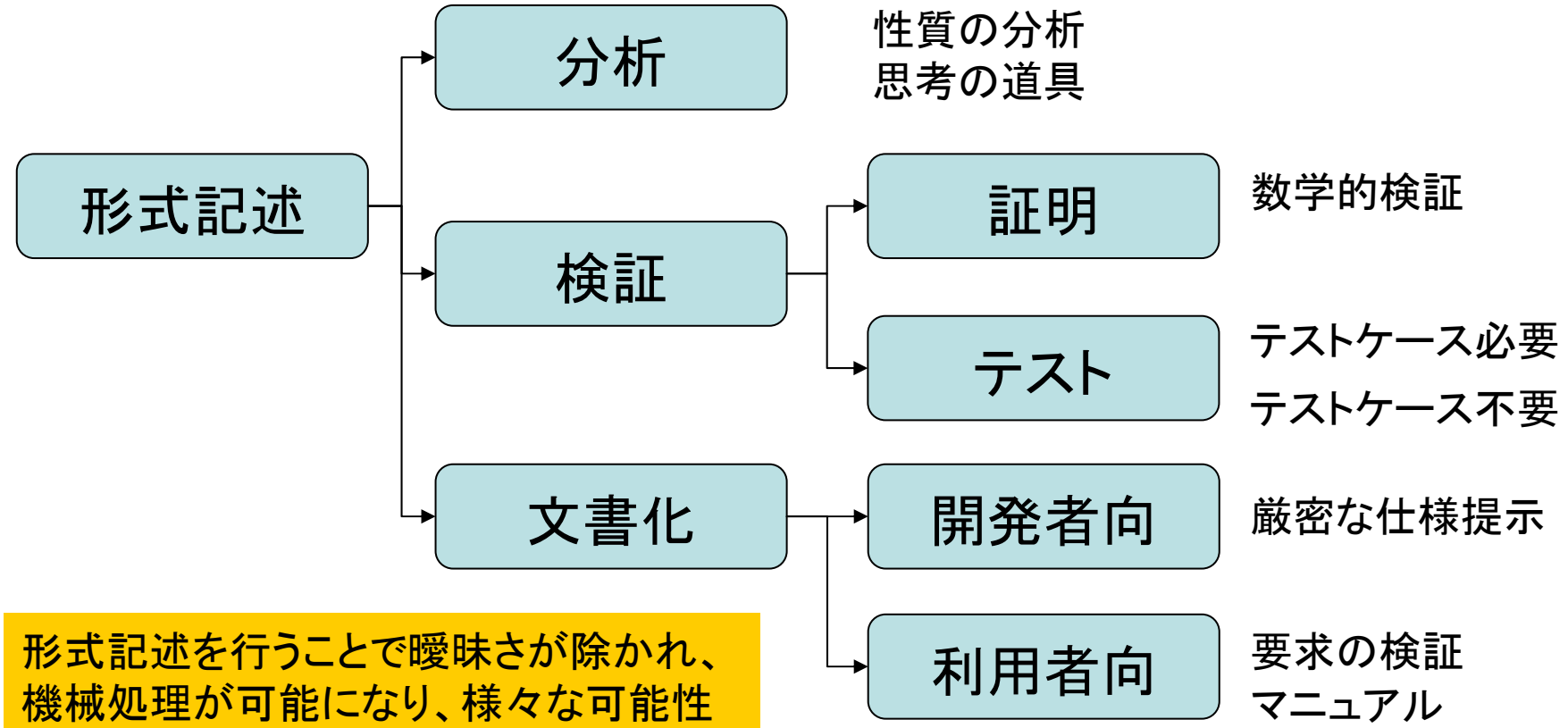
- いますぐ試してみよう
  - 形式手法は明日の技術ではなく、いまそこにある技術。活用すれば仕事を劇的に楽にする
- 形式手法はコミュニケーションを助ける
  - 怠惰な技術者は「形式手法」で楽をしよう、他者との対話自分との対話の基礎を固めて、「本質的に考えるべきこと」に時間を割こう

本日の話題

- 形式仕様とは
- 仕様書の性質
- モデルベースの形式仕様記述
  - 契約による設計
- 現場での事例二つ
- 仕様記述の例二つ → 説明用
- まとめ

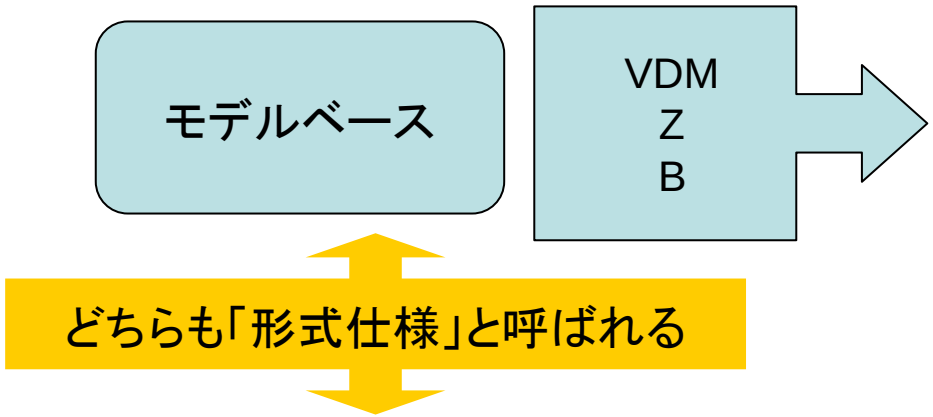
形式手法とは？

ソフトウェア開発において、ある側面の「仕様」を
「厳密」に記述し、開発工程で利用する手段の総称



形式記述を行うことで曖昧さが除かれ、
機械処理が可能になり、様々な可能性
が開ける

代表的な手法・言語



情報の構造や、ある状態から他の状態への変化をモデル化
定理証明、回帰テスト

本日の中心

曖昧さの除去、専用言語の利用によるコンパクトな記述、仕様の「実行」「評価」可能性
汎用の記述に使える

振舞いのモデル化
全状態空間を生成し自動検査

従来の「テスト」では見つからない潜在的な障害を発見できる

段階的モデル構築
自動ツールによるモデル分析

「モデル検査」よりも複雑な構造と条件を記述・検査できる

形式手法が注目される理由

- システムの社会的責任が高まり、より高い信頼性の確保を仕様記述段階でも望まれるようになった
- 利用できる実用的なツールが増えてきた
  - VDMTools (VDM)
  - SPIN (PROMELA)
  - LTSA (LTS)
  - etc, etc…
- 成功事例が増えてきた
  - CSKシステムズ（旧日本フィッツ）
  - フェリカネットワークス
  - 他（海外には既に多数あり）

仕様書の性質

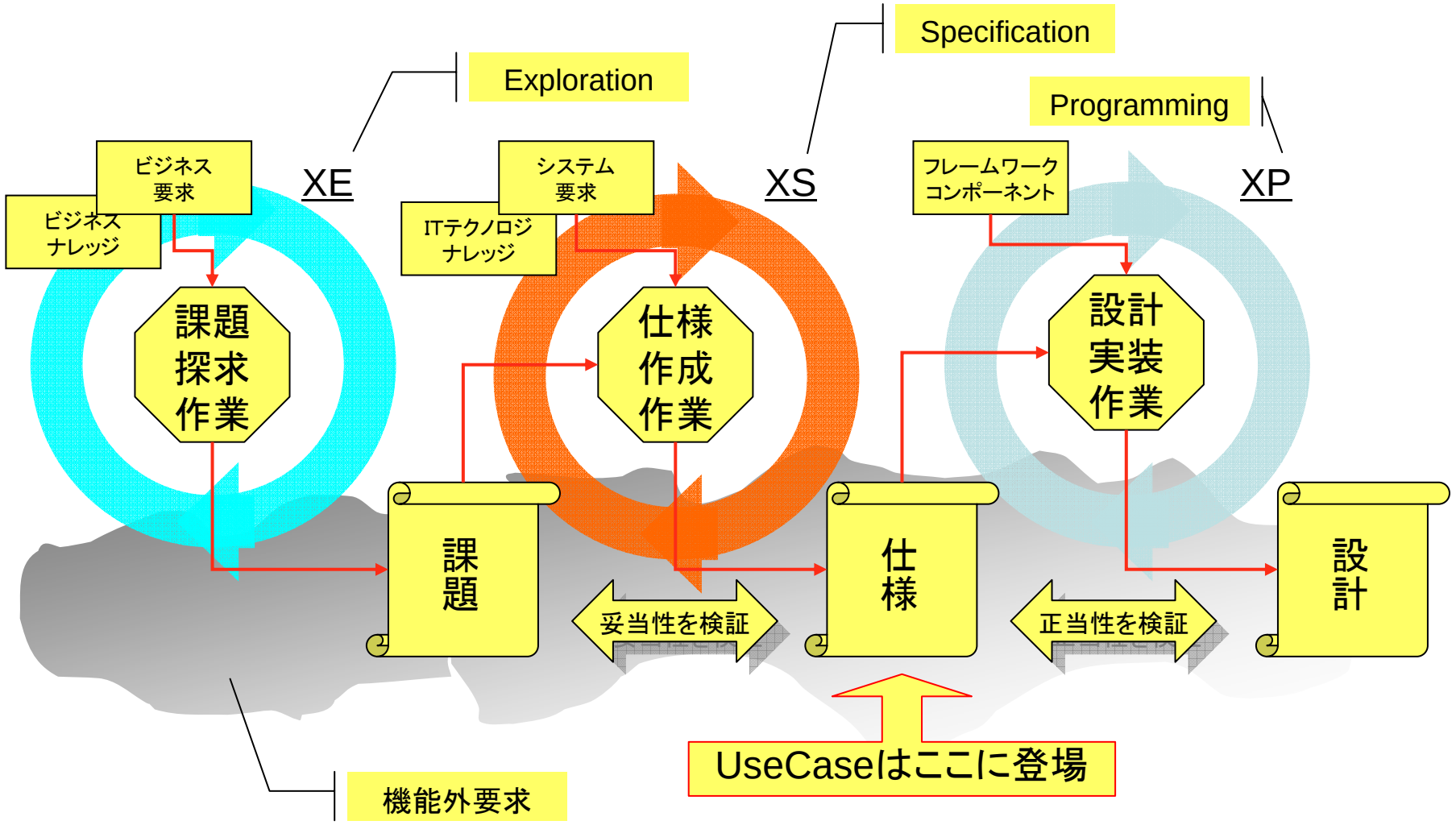
仕様書の位置付け

- 「仕様」は課題と設計をつなぐもの
  - 言い換えれば「問題の解決」が「問題の解法」にどのように結び付けられるかを記述するもの
  - 「問題の解決」とは：
    - どのような条件のもとでどのような効果・状態が得られたり維持されたりすれば、問題が解消されたといえるのかを規定したもの
  - 「問題の解法」とは：
    - 与えられた材料で指定された効果、状態を創出するために具体的に行うべき手順を書いたもの

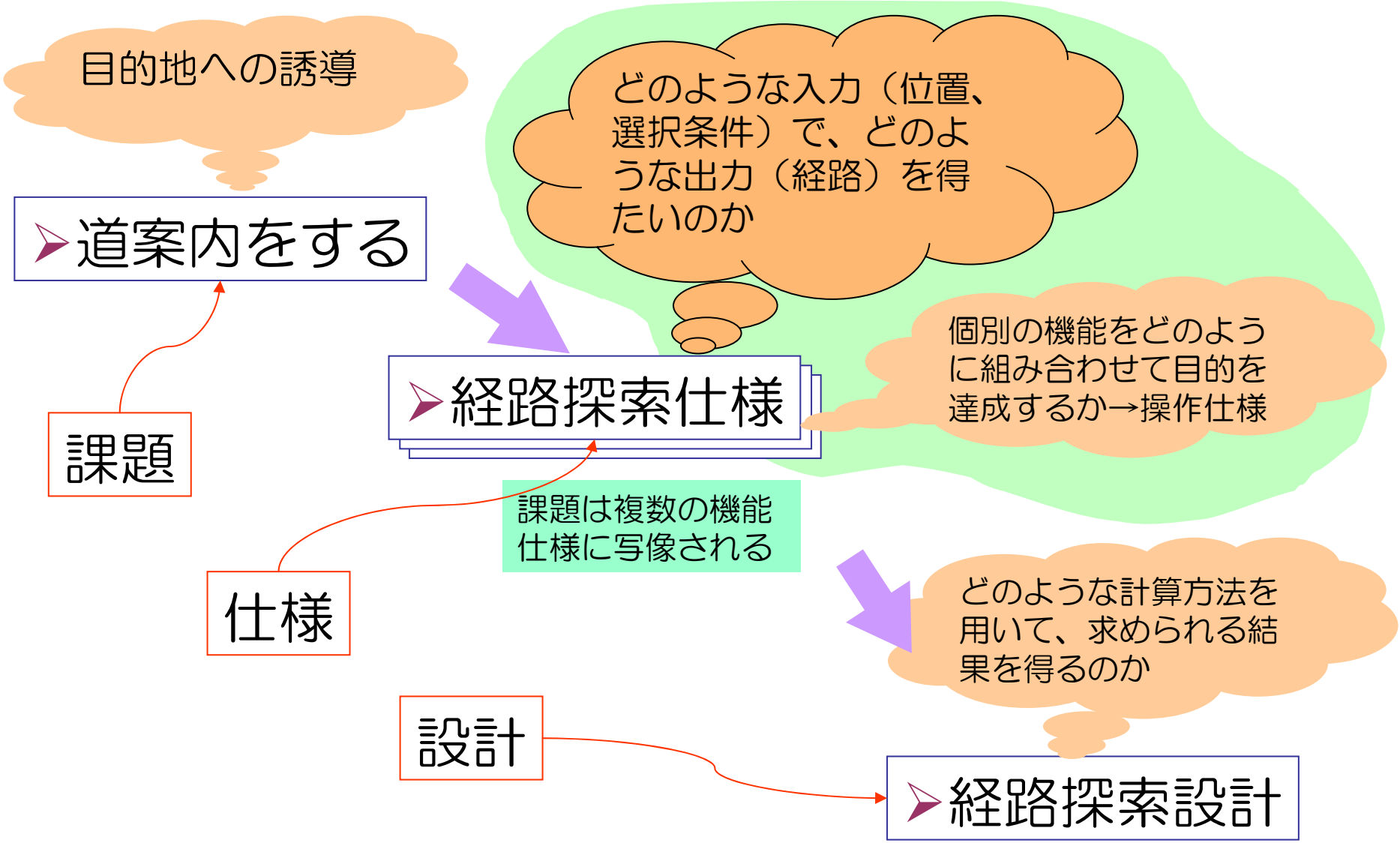
システム開発の3つの視点

- 課題 (要求＋問題領域)
  - 利用者の視点
    - 現実世界にどのような問題を抱えており、そのうちどの部分をシステム化して解決したいと思っているのか
    - どのような状態が「問題の解決」といえるのかを定義
  - 入力：ビジネス要求、利用者の要求、問題領域の知識
- 仕様
  - 利用者と開発者の視点
    - 上で挙げられた課題の解決を、どのようなシステムで支援するのか
    - 「問題の解決」と「問題の解法」の対応関係を定義
  - 入力：課題＋システム要求
- 設計
  - 開発者の視点
    - 要求されるシステムの仕様をどのように設計すべきか
    - どのようにして求められる効果を生み出すか➔「問題の解法」を具体的に定義
  - 入力：仕様＋最新構築技術

課題-仕様-設計の関係



課題・仕様・設計



仕様記述が満たすべき性質

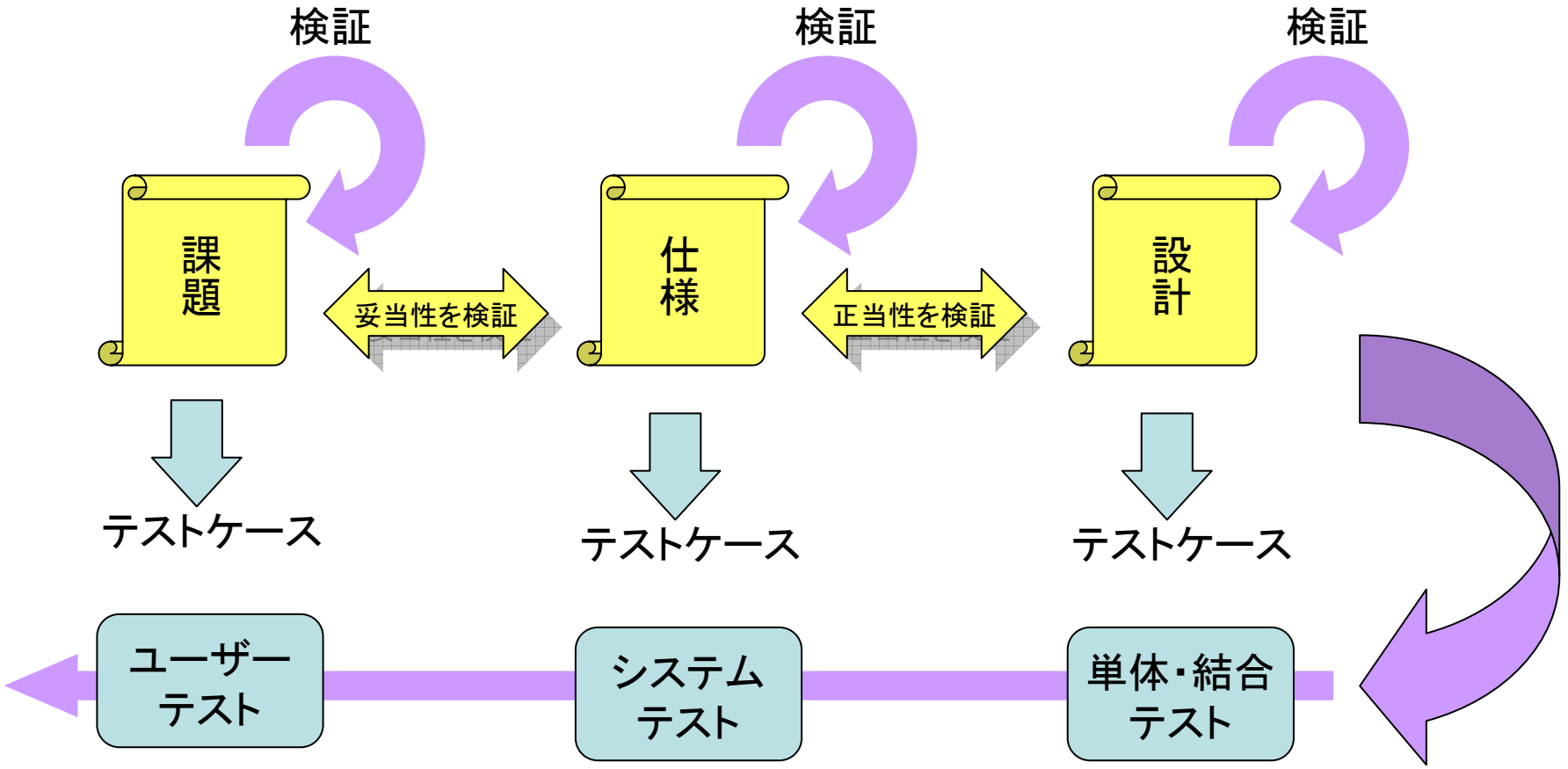
- [課題] 対応する課題の指定
  - どの問題・課題を解決するためにこの仕様は存在しているのか
- [目的] 目的（得られる効果・状態）の明記
  - 前提条件が満たされたという仮定の下で、どのような振舞いが期待されているのか
- [前提] 前提条件（文脈）の明記
  - どのような条件、入力が与えられた情況でこの仕様が求められるのか
- [依存] 依存する仕様の定義
  - どの仕様に依存しているのか

多くの仕様書では、これらがきちんと書かれていない
→（曖昧、未分化、未定義、重複、記述モレ）

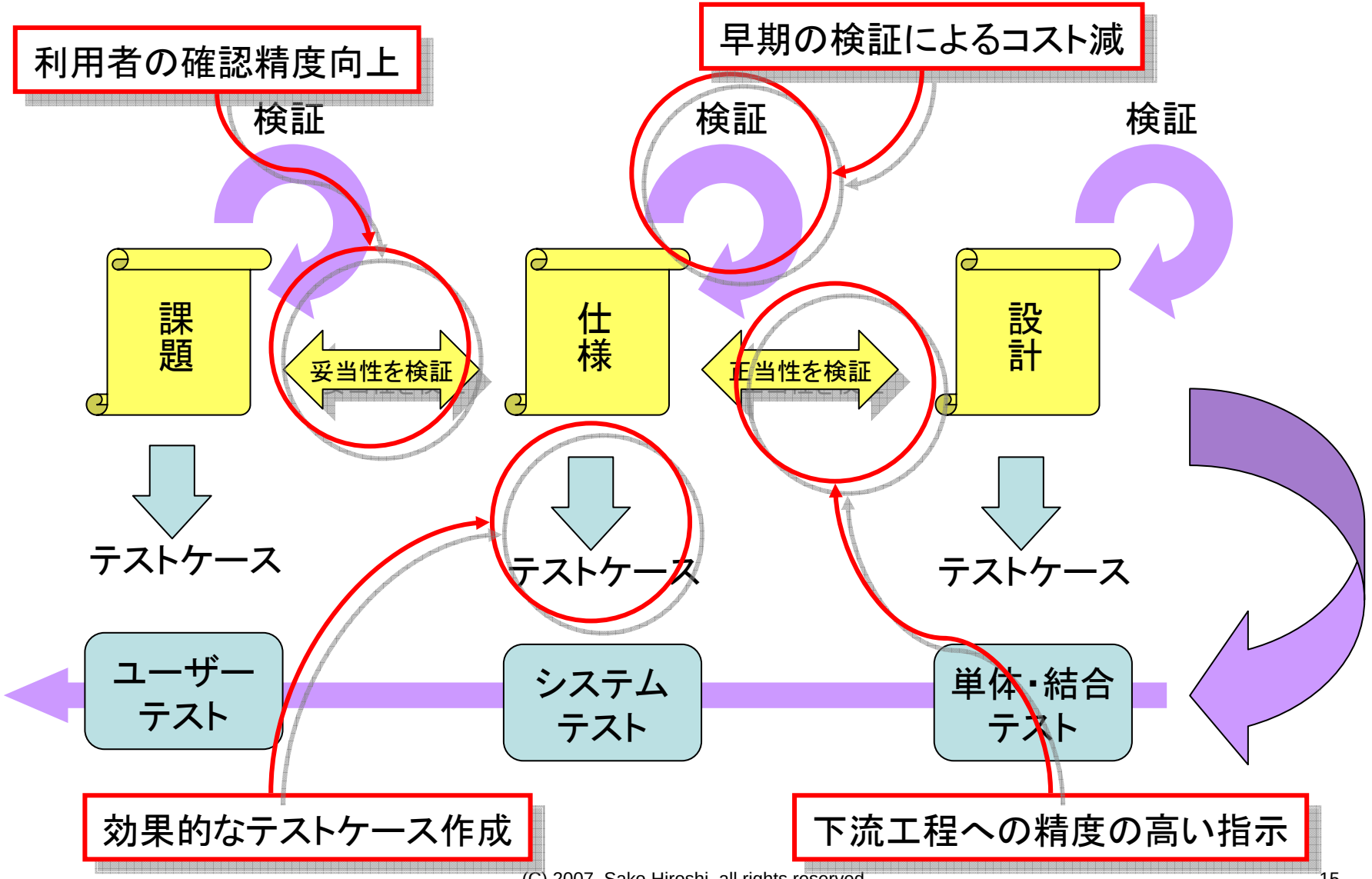
仕様記述の「品質」を評価するには

- 何より「検証可能」でなければならない
  - 「検証可能」な記述を体系的にテストすることによって品質を評価することができる
- 検証可能性に対する3つの視点
  - 抜けモレの排除
  - 記述の一貫性、完全性の確保
  - 記述間（要求⇔仕様、仕様⇔設計、設計⇔実装）の整合性の保証

記述と検証



記述の効果（仕様を中心に考えた場合）



モデルベースの形式仕様記述

- 操作対象のモデルの仕様を「契約」関係で表す
- 「オブジェクト指向入門」に登場する「契約による設計」そのものと言える
- 「オブジェクト指向入門」では実装言語としてあきらめた、強力でわかり易い「記述方式」をあえて取り込んだもの

例：実装上「効率的」ではない記述

```
class「予約」
    …..
    -- この予約が使っている、区間ごとの座席集合
    public 区間利用座席集合 :「路線」
            ==> map (「路線」`「駅」 * 「路線」`「駅」) to  set of「編成列車」`「座席」
    区間利用座席集合(a路線) ==
     return { 区間 |-> 座席集合 | 区間 in set a路線.区間集合(乗車駅, 降車駅)};

class「予約」
```

Design by Contract
契約による設計

“正しい”ソフトウェア構築のために

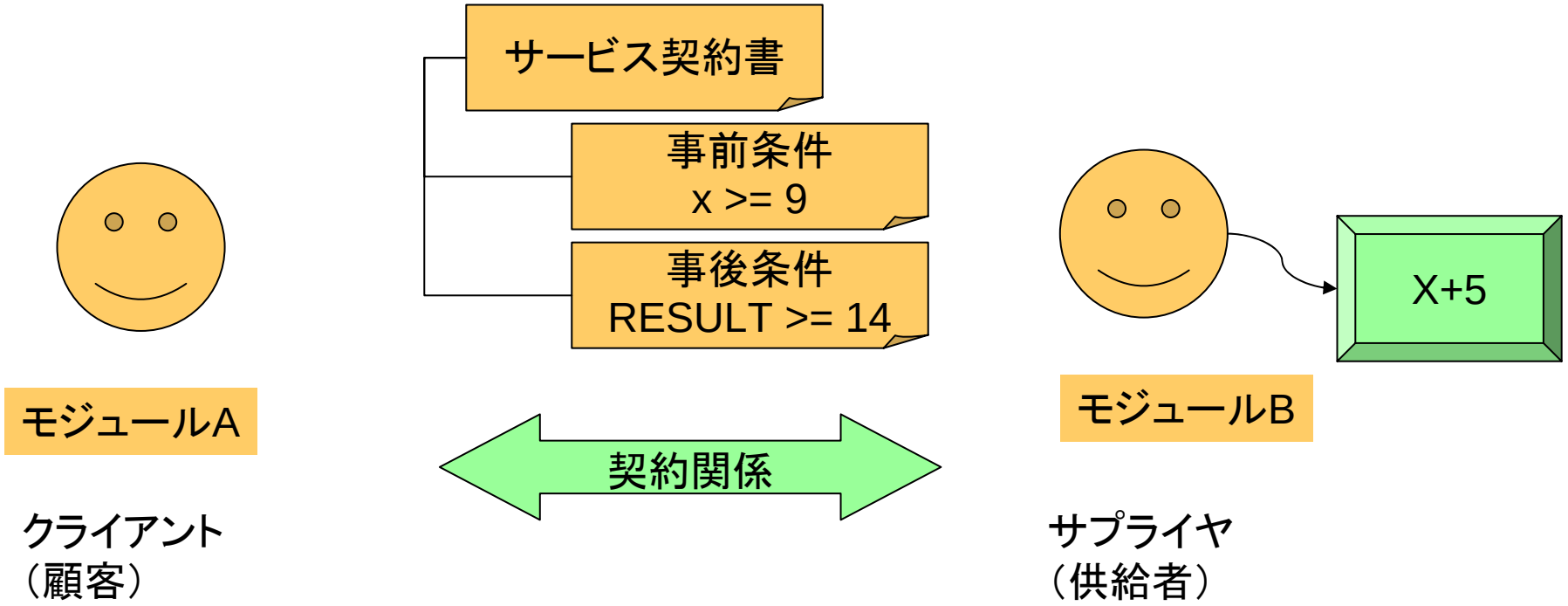
権利と義務の表明

- プログラムの正しさを議論する際にしばしば以下のような表記が用いられる

$$\{P\}\ A\ \{Q\}$$

- この表記の意味は「条件 P が成立しているときに、Aが実行されると条件 Q が成立する」という意味である
- すなわち P は A にとって、呼び出される際に要求できる権利であり、Q は A にとって果たすべき義務ということになる
- P がきちんと成立しているかどうかを気にしなければいけないのは A のクライアント（呼び出す側）であり、A 自身には責任はない
- P は事前条件と呼ばれ、Q は事後条件と呼ばれる。P と Q はモジュール A を利用する場合の双方の権利と義務を記述した「契約書」である

契約書:事前条件と事後条件



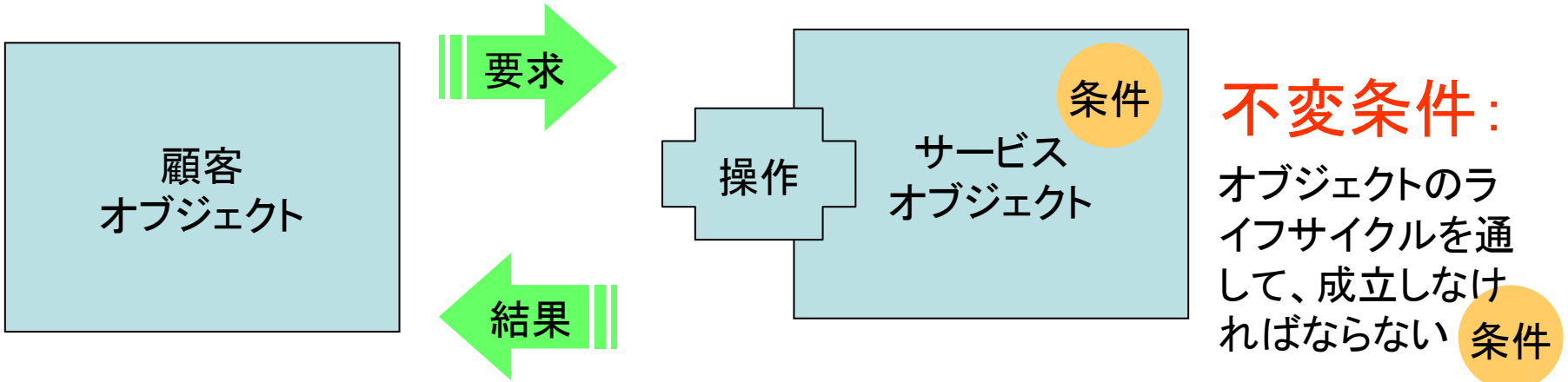
事前/事後条件を扱う

事前事後条件とは、モジュール間の
契約関係を記述する重要な条件

事前事後条件、不変条件の関係

事前条件： 操作に先立ち必要な 条件 が成立している

事前条件が成り立つ状態で操作を使うのは顧客オブジェクトの義務



不変条件：

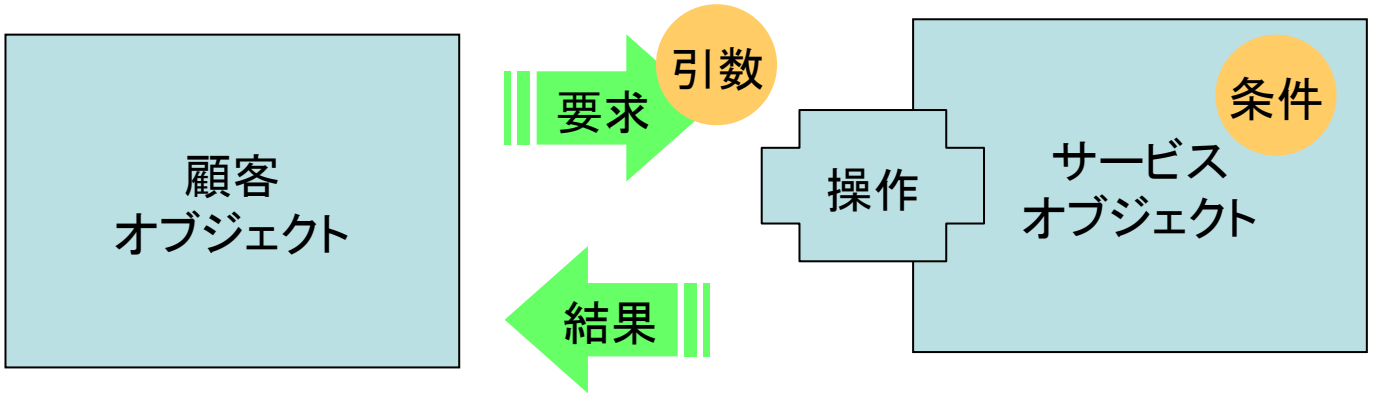
オブジェクトのライフサイクルを通して、成立しなければならない 条件

事後条件： 操作の結果望んだ 条件 が成立している

操作の結果、事後条件を成り立たせるのはサービスオブジェクトの義務

仕様と設計における事前事後条件-1

理想的には事前条件が成り立つ状態で操作を使うのは顧客オブジェクトの義務であるので、本来サービスオブジェクト側は、引数のチェックなどはしなくても良い筈である

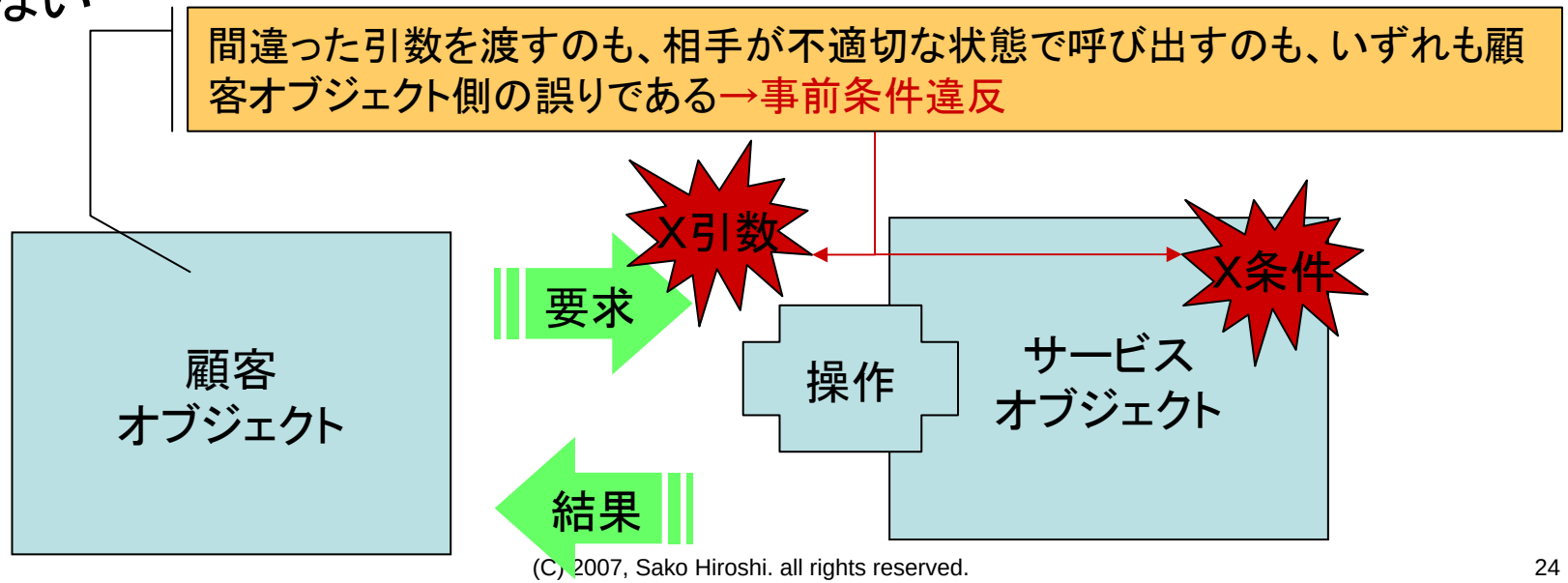


しかし現実には事前条件が成り立たない状態で操作が呼び出されてしまう可能性がある。この原因は主に（1）顧客側の不具合（2）サービス側の内部状態（顧客からは見えない）の不整合にある
では実際のシステムを構築する上で、これをどのように考えるべきだろうか？

仕様と設計における事前事後条件-2

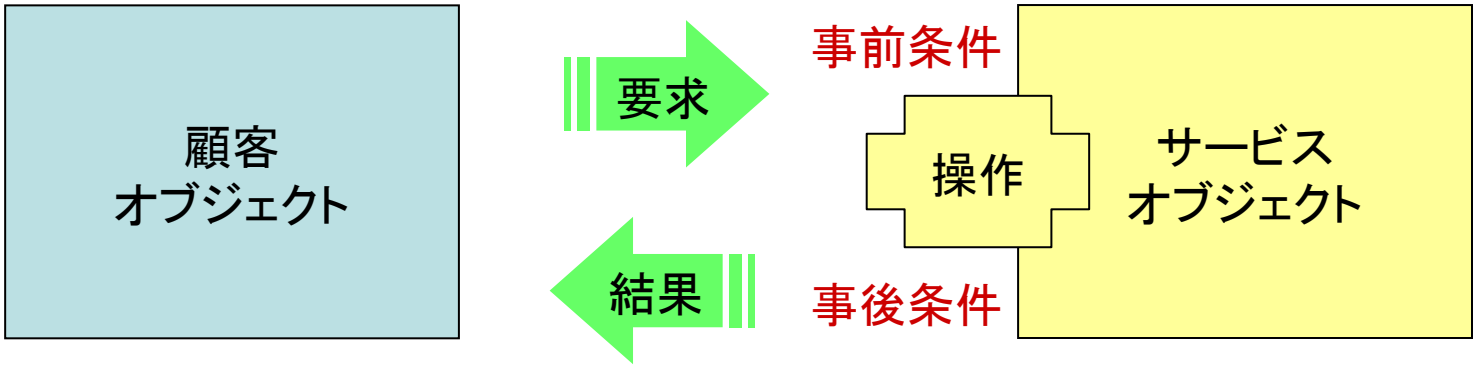
「実行」に目を奪われると、仕様としてモデルを書いている最中でも、サービスオブジェクト側の操作内に「事前条件違反」対策のコード（例外処理）を書きたくなる。しかしそれは、間違いである。

それでは、事前条件を書いている意味がなくなってしまう。あくまでも仕様記述の段階では、事前条件への違反を見つけたらそれを「顧客」オブジェクト側の仕様ミスとして還元・修正しなければならない



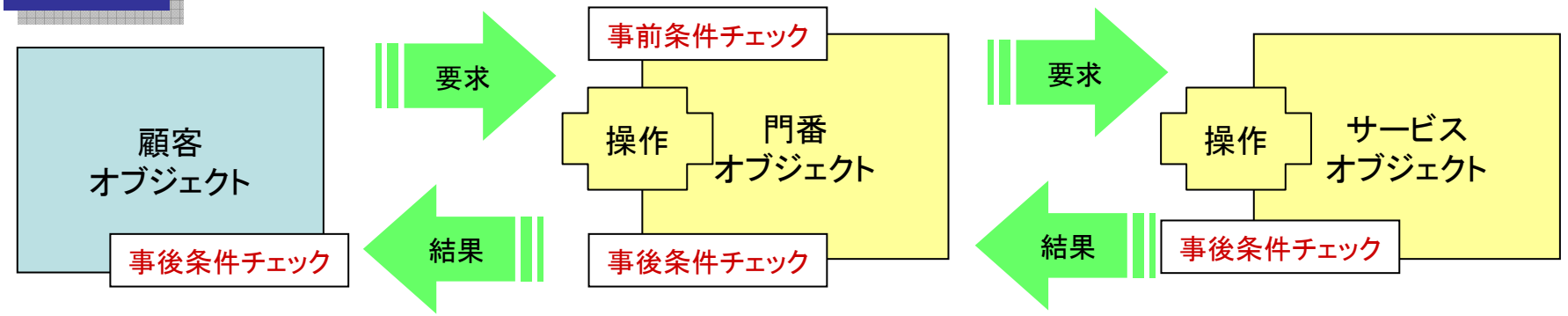
仕様と設計における事前事後条件-3

仕様



顧客オブジェクトの品質がよく分からない場合、事前条件を守って呼ばれるのかどうかが、不明である。このとき設計として「防衛的」に事前条件をチェックするレイヤーを間に挟むこともできる→しかしそれによって顧客オブジェクト側の詰めが甘くなるとすれば本末転倒である

設計

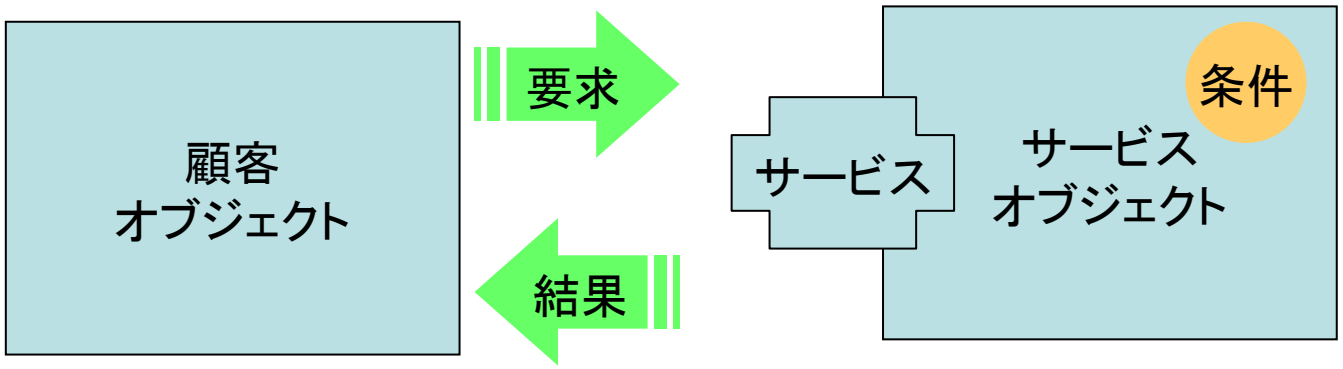


設計上は事前条件チェックをちゃんとして例外ではじけば十分であることが多い

不変条件を扱う

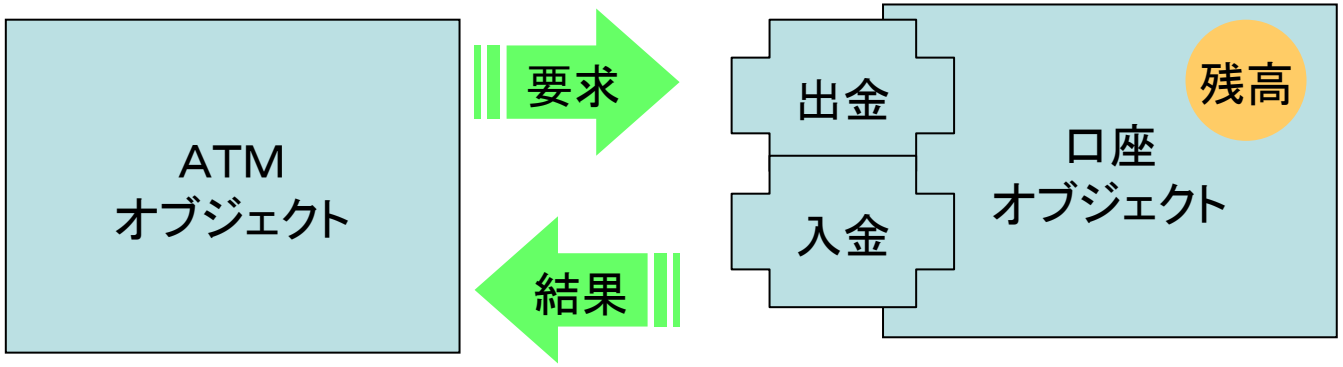
不変条件とは全ライフサイクルを通
してずっと真であり続ける性質

不変条件の意味



不変条件:

オブジェクトのライフサイクルを通して、成立しなければならない 条件



例:

どのようなサービスが使われても常に

残高 ≦ 1000 万
and 残高 ≧ 0

が成り立つ

「年月日」の不変条件（一部）

うるう年の定義：
　４で割れる年はうるう年
　ただし、１００で割れる場合はうるう年にしない、
　ただし、その中で４００で割れるものはやはりうるう年とする

この不変条件により、正しくない 2/29 を生成しようとするとエラーが起きる

形式仕様記述言語による記述

```
class「年月日」

  instance variables
    日 : nat1;
    月 : nat1;
    年 : nat1;
    inv 日 <= 31 and
        月 <= 12 and
        if 月 in set {4, 9, 6, 11}  then
          日 <= 30
        else
         (月 = 2) =>
           (if (年 mod 4 = 0) then
             (if (年 mod 100 = 0) then
               (if (年 mod 400 = 0) then (日 <= 29) else (日 <= 28))
              else
               日 <= 29)
            else
              日 <= 28);

end「年月日」
```

例:特急券予約

```
public 新規予約 :「座席予約伝票」 ==> [「予約」]
    新規予約(a伝票) ==
      is not yet specified
    pre
      forall 駅 in set {a伝票.乗車駅, a伝票.降車駅}
                & 駅 in set elems a伝票.列車.路線.路線
    post
      RESULT <> nil =>
        let 新規予約 = RESULT in
          (新規予約.顧客 = a伝票.顧客 and
           新規予約.列車 = a伝票.列車 and
           新規予約.乗車駅 = a伝票.乗車駅 and
           新規予約.降車駅 = a伝票.降車駅 and
           card 新規予約.座席集合 = a伝票.人数)
        and
          card 予約集合 = card 予約集合~ + 1;
```

「予約」または nil が返される

詳細は未定

事前条件

乗車駅、降車駅ともに路線上にある

事後条件

結果が返る場合には、内容は伝票を反映したものになる

そして予約の数が一つ増えている

例:特急券予約 – 予約

```
class「予約」
  instance variables
    public 顧客     :「顧客」;
    public 列車     :「編成列車」;
    public 乗車駅   :「路線」`「駅」;
    public 降車駅   :「路線」`「駅」;
    public 座席集合 : set of「編成列車」`「座席」;
    inv 乗車駅 <> 降車駅 and
          card 座席集合 > 0 and
          座席集合 subset 列車.座席集合;
```

不変条件

例:特急券予約

仕様を「評価」できるように中身を埋めたもの

```
public 新規予約 :「座席予約伝票」 ==> [「予約」]
新規予約(a伝票) == (
   let 空席集合 = a伝票.列車.指定駅間空席集合(予約集合,
                                                a伝票.乗車駅, a伝票.降車駅) in
       if card 空席集合 >= a伝票.人数 then (
         dcl 予約 :「予約」
                  := new「予約」(a伝票,指定数座席取出し(空席集合, a伝票.人数));
         予約集合 := { 予約 } union 予約集合;
          return 予約
       )
       else
         return nil
)
```

形式仕様記述を用いた事例

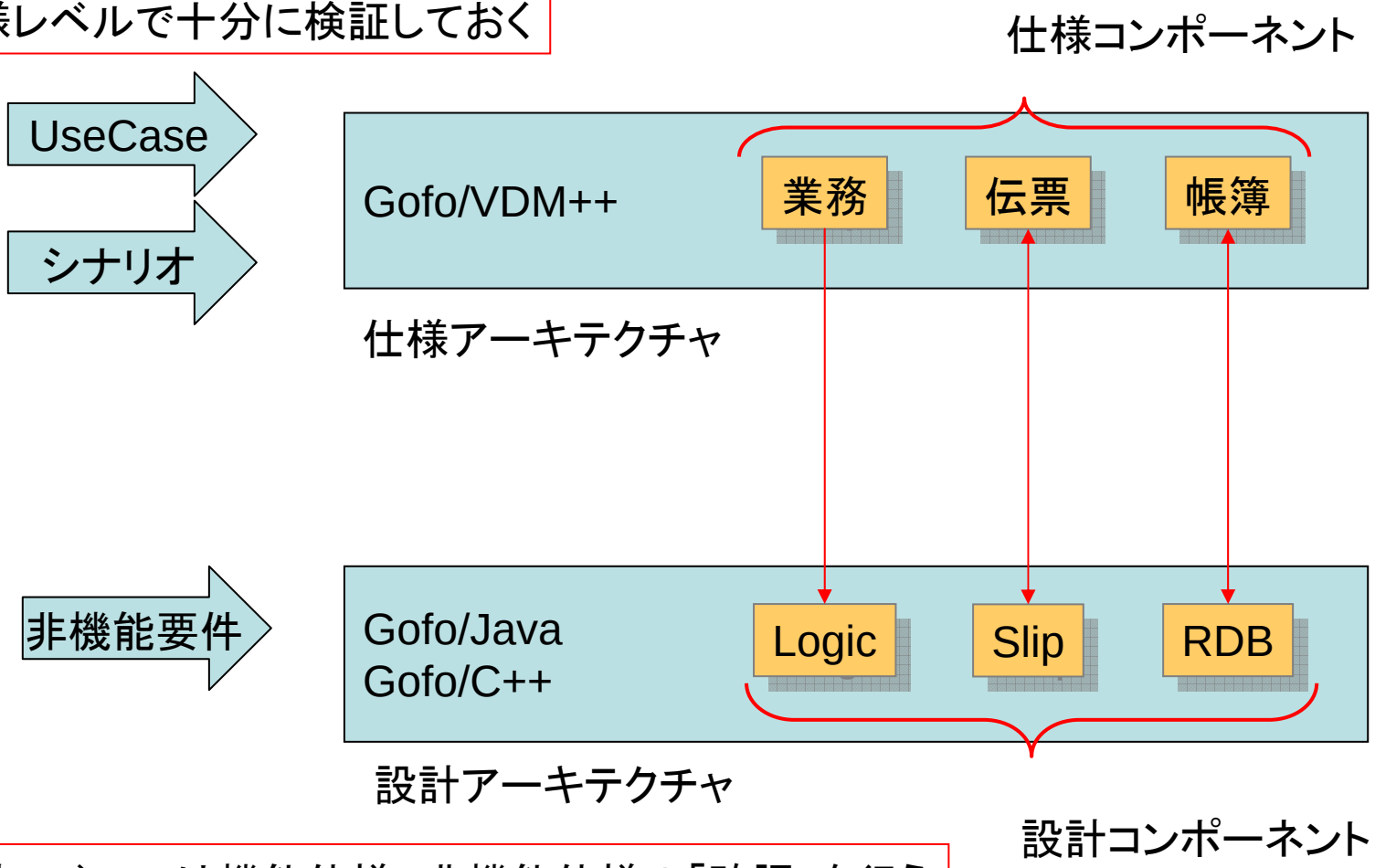
- VDM を利用 -

事例1

- オンライン 証券業務 開発プロジェクト
  - 旧日本フィッツ（現CSKシステムズ）
- 特徴
  - 仕様記述のためのフレームワークを用意
  - 設計アーキテクチャとのマッピングを策定
  - シナリオに基く仕様の検証（仕様の回帰テスト）
- 概要
  - Gofo/VDM++で仕様記述
  - Gofo/C++, Gofo/Java で設計、実装
- 効果
  - リリース後、仕様に起因するエラーはなし

事例1

仕様レベルで十分に検証しておく

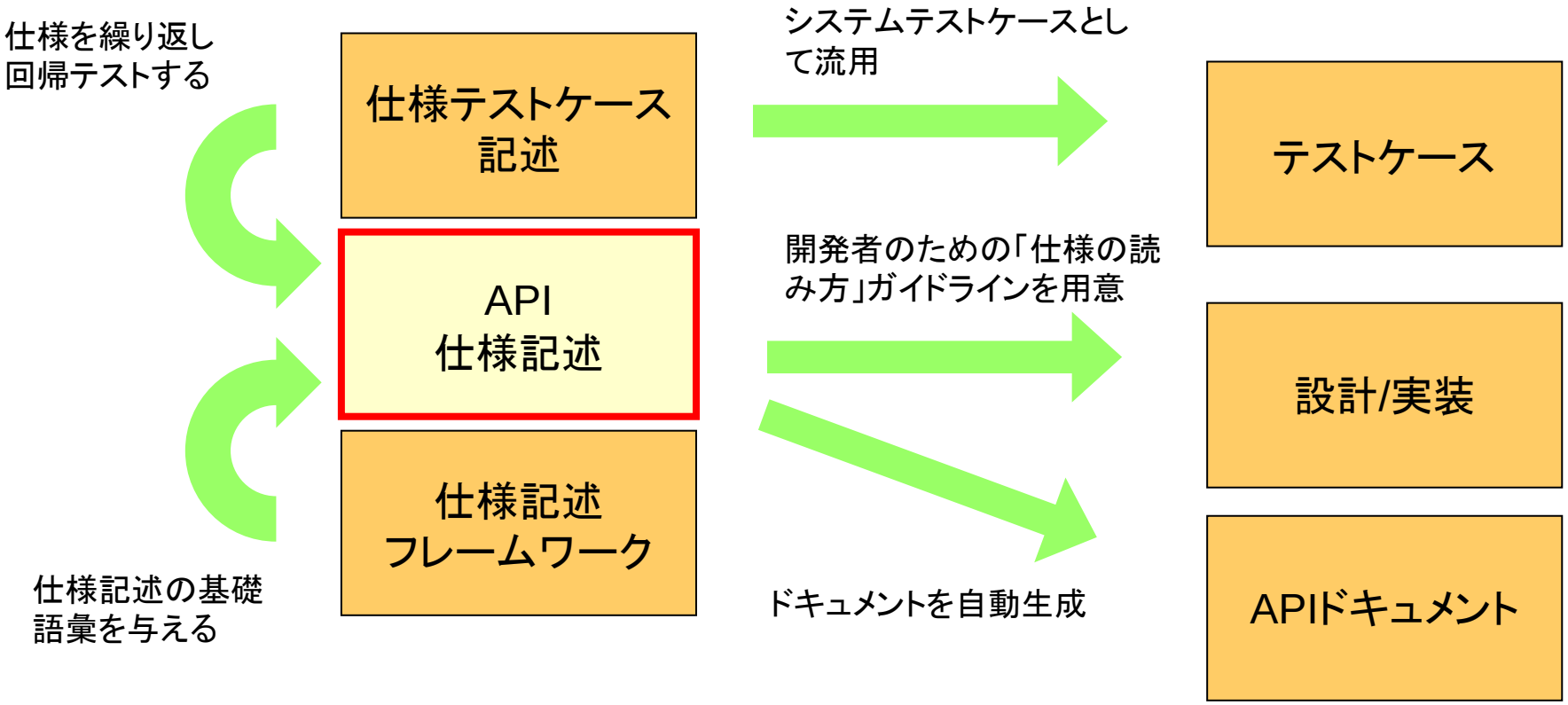


設計レベルでは機能仕様、非機能仕様の「確認」を行う

事例2

- 次世代 FeliCa チップ開発プロジェクト
- 特徴
  - 仕様記述のためのフレームワークを用意
  - ハードウェア/エミュレータが用意される以前から仕様の広範囲な検証が可能（仕様の回帰テスト）
  - 仕様策定者と設計開発者の間のコミュニケーションの質が向上
- 概要
  - API の仕様を VDM++ で記述（フレームワーク化）、すべてのマスターに（→文書化なども行う、いかなる仕様変更も VDM++に反映し検証してから設計に流すルールとしたため仕様の一元管理が実現）
  - VDM++ の記述と C による実装で多くのテストケースを共有（→早い段階から評価環境を検討できた）
  - 構成管理ツールを利用して仕様の検討開発過程の詳細な生産性のデータが得られた（→将来の見積のベースに）
- 効果
  - リリース後、仕様に起因するはエラーなし
  - 開発プロジェクト内でのコミュニケーションの質が劇的に向上

事例2



プロジェクトリポジトリとしての形式仕様記述

DRY (don’t repeat yourself)

一度書かれた仕様は、
骨の髄まで活用したい

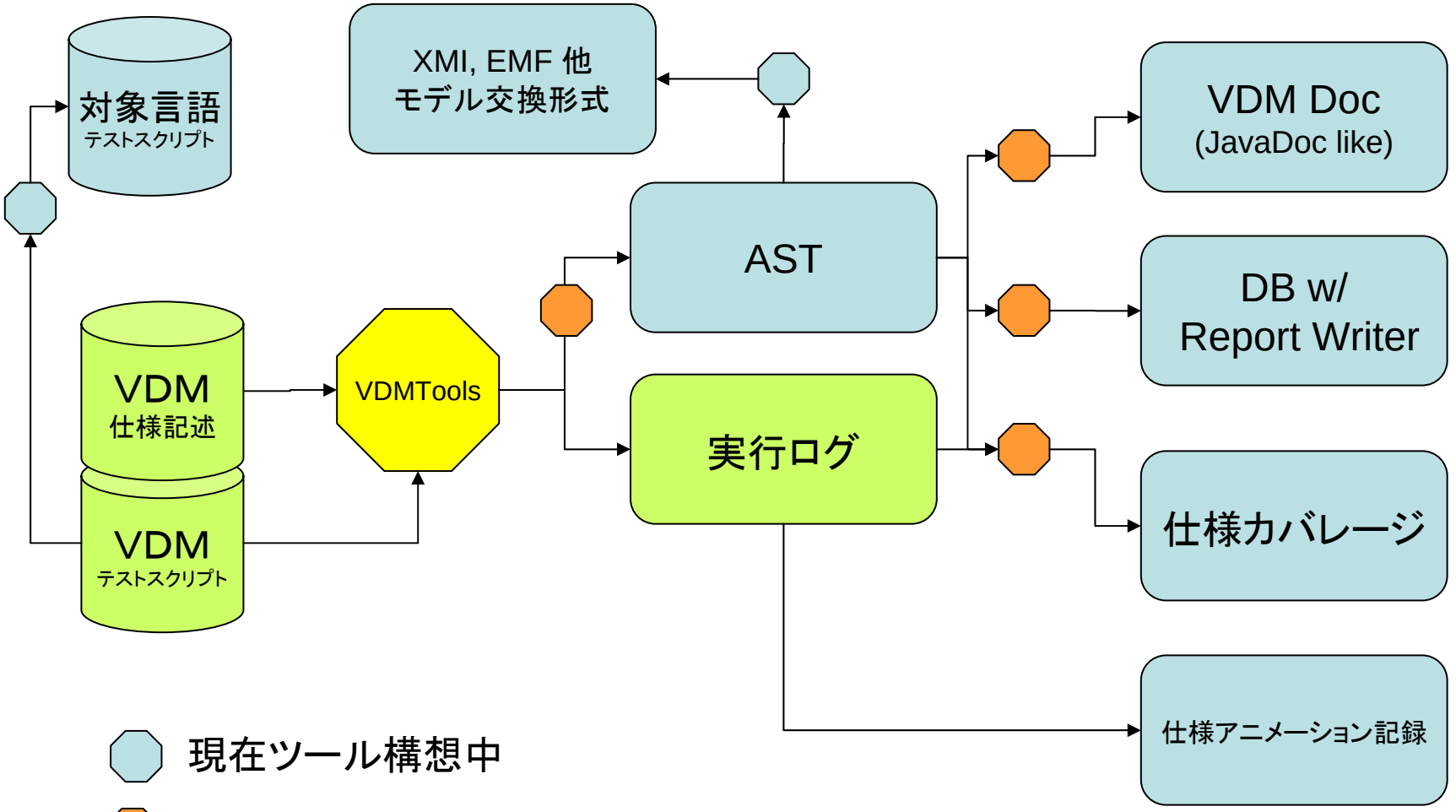
もし仕様書が「形式化」されるなら

- 「課題側」「設計側」が必要とする各種情報（単に段階的詳細化だけではなく）を様々な形で「抽出」利用することが可能
- 例：
  - 型の定義 → DB スキーマ
  - 不変表明 → バリデーションルール
  - 外部操作の仕様 → API 一覧とレファレンスマニュアル
    - JavaDoc のような文書化＋高精度（正確、最新）な情報
  - 状態遷移モデル → 操作マニュアル, テストケース生成
  - 仕様のテストケース → 実装のテストケース
  - その他各種ダイアグラムの生成などを必要に応じて
  - etc…

現場の文脈

- 実は今多くの仕様書が UML ではなく
  - Excel/Wordで書かれている、あるいは書かれようとしている
  - 仕様を何らかの「形式化」（多分無意識）に載せようとしているのだが、「記入用紙」先行でその中の十分な形式化が行われていない
  - 一見フォーマットは整っているものの、保守できない、検証できない仕様の山ができる
- 裏を返せば
  - 内部ではきちんとした仕様記述言語（モデル）を用意するにせよ、外面上はそれなりの「記入用紙」を用意して、穴埋め、検索、仕様アニメーション、仕様の回帰テスト、自在な文書化などの仕掛けを付け加えてやれば採用しようという機運は高まる（ドメインがある程度成熟していることは必要）
- 形式仕様というラベルを貼らなくても以下のような「看板」でも十分魅力的
  - 要件管理のできる仕様書（トレーサビリティ）
  - リファクタリングできる仕様書
  - 構文・型検査のできる仕様書
  - 差分管理のできる仕様書
  - 検証できる仕様書

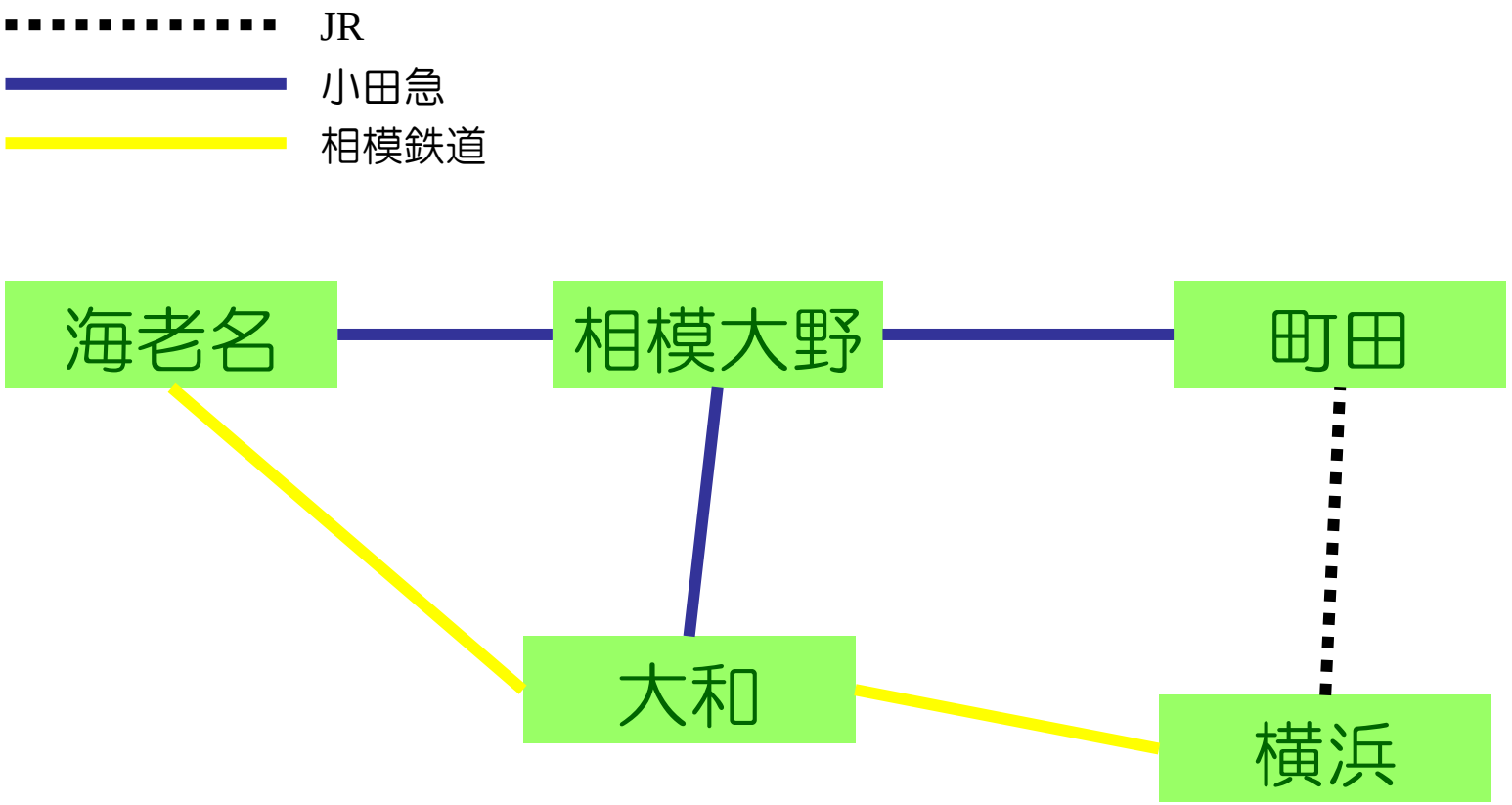
レポジトリ化のスキーム（例）



「仕様記述」を考える例1

最適経路の計算

例：鉄道路線があるとする



要求：ある駅から、別の駅までの「最適経路」を計算したい

「最適経路」を求めよ

日本語の仕様

最適な経路を求めよ

もちろん、これは駄目な例➔何故駄目なのか？

いろいろな定義が曖昧…例えば…

最適：何の観点から見て最適なのか（時間、距離、その他）

経路：経路を定義するものはなにか（始点、終点、経由点）

求めよ：どのような形で結果を得るのか（路線、駅、その他）

では少し改善してみよう。。。

日本語の仕様

最適な経路を求めよ。最適な経路とは指定された条件（所要時間、料金）に沿った、2駅間の最も適する経路のことである。

疑問：

➔何を指定するのか？
➔求める解は1つ？順序あり？
➔そもそも「経路」とは？
➔そもそも「最も適する」とは？
➔…etc…あまり改善されていない

何を指定するのか？

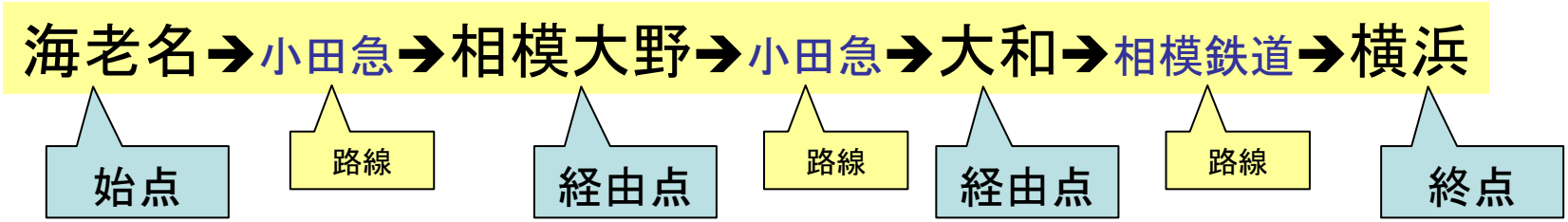
- 始点と終点
  - 必ず始点と終点を指定する
- 経由点
  - 0点以上を指定、始点と終点は含まれない
- 探索条件
  - 時間優先
  - 料金優先
  - 乗換優先

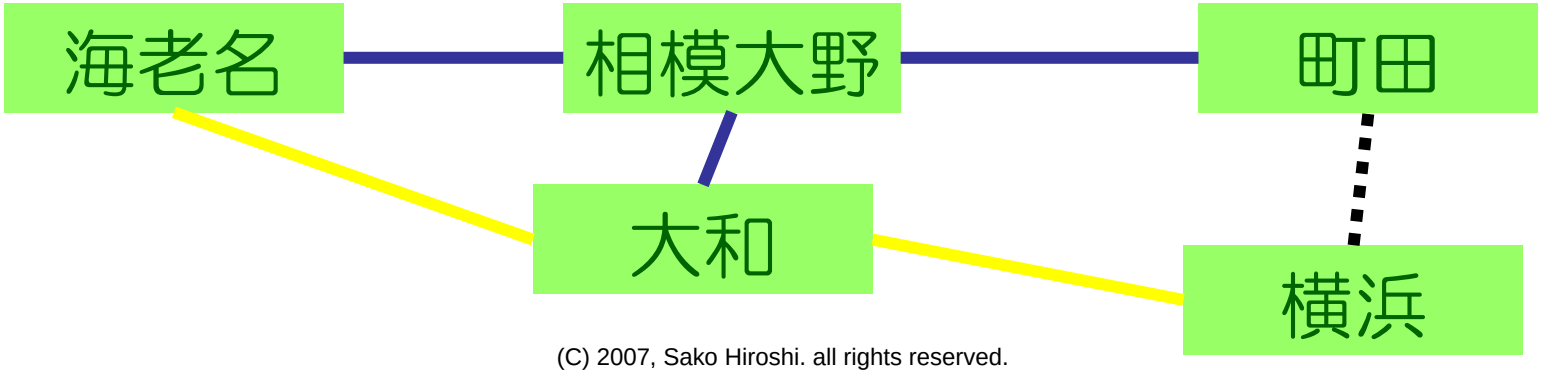
求める解は1つ？順序あり？

- 与えられた条件の下で得られた経路を、指定された探索条件による評価順に並べて返す
- 例えば
  - より安い順
  - より早い順
  - より乗換回数が少ない順

そもそも「経路」とは？

ここで言う「経路」とは始点、終点、経由点の集合が指定され、途中利用する路線が曖昧なく判断できるような情報である





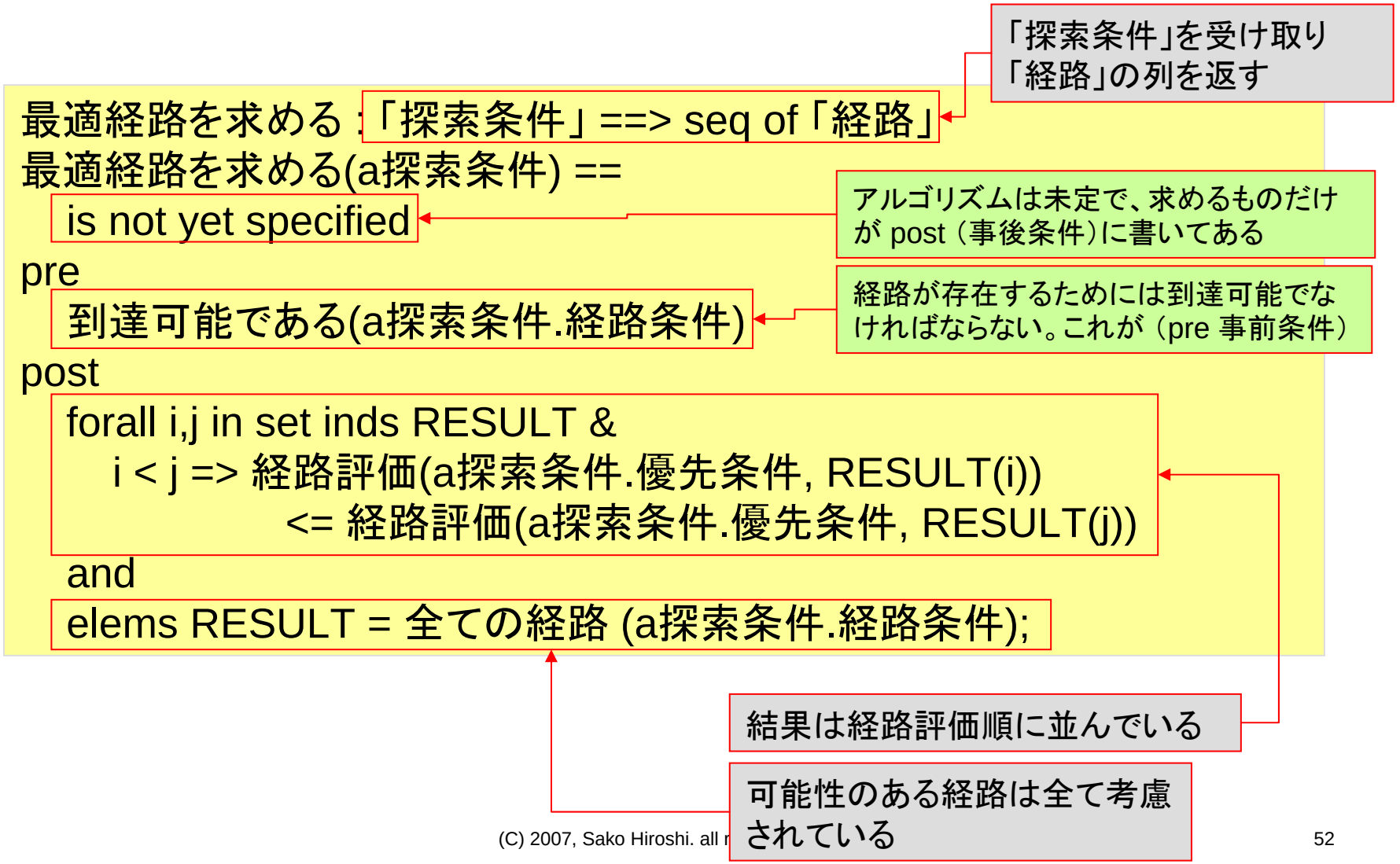
そもそも「最も適する」とは？

- 指定された探索条件によって経路を評価した際に、最も「有利」になること。
- すなわち以下のようなもの
  - 最も料金が安い
  - 最も時間が短い
  - 最も乗換え回数が少ない

再び「仕様」に戻って

- ではここまでに考えた細かい条件や定義を、曖昧さのない形で記述し、他の人に伝えるにはどうしたら良いのだろうか？
- 日本語でも注意深く記述して、注意深く読めば誤解は少なくなる。しかし、ちょっとしたミス、個人間の記述の違い、用語の齟齬は排除しきれない
- また機械の支援を受けた検証も行い難い

形式仕様の例（主要部分）



形式仕様の例（補助部分）

```
全ての経路 :「経路条件」==> set of「経路」
全ての経路(a経路条件) ==
      is not yet specified;
```

指定した経路条件に合致する全ての経路の「集合」を求める

```
到達可能である :「経路条件」==> bool
到達可能である(a経路条件) ==
      return card 全ての経路(a経路条件) > 0;
```

経路が一つでも存在すれば到達可能であるとみなす

```
経路評価 :「優先条件」*「経路」==> int
経路評価(a優先条件, a経路) ==
      return
        cases a優先条件:
          <料金> -> 料金計算(a経路),
          <時間> -> 時間計算(a経路),
          <乗換> -> 乗換計算(a経路)
      end;
```

優先条件と経路を与えて、経路としての「評価値」を求める

形式仕様定義に使った型

```
--「駅」は区別できれば良いだけ
「駅」= token;

--「優先条件」と「経路条件」を合わせて「探索条件」に
「優先条件」= <時間> | <料金> | <乗換>;
「経路条件」:: 発駅 :「駅」
               着駅 :「駅」
               経由駅集合 : set of「駅」;
「探索条件」:: 優先条件 :「優先条件」
               経路条件 :「経路条件」;

--「路線」も区別できればよいだけ
「路線」= token;

--「経路」は経由する駅の列と、各駅間の路線で構成
「経路」:: 駅列 : seq of「駅」
           路線列 : seq of「路線」
           inv p == len p.駅列 = len p.路線列 + 1;
```

優先条件は、<時間>、<料金>、<乗換> の3種類ということがわかる

経路条件は発駅、着駅、経由駅の集合を指定したもの（→何か条件が？）

探索条件は、優先条件と経路条件から構成されている

全体定義（陰仕様初版）

```
class「経路探索」

 types

  「駅」= token;

  「優先条件」= <時間> | <料金> | <乗換>;

  「経路条件」:: 発駅 :「駅」
                着駅 :「駅」
                経由駅集合 : set of「駅」;

  「探索条件」:: 優先条件 :「優先条件」
                経路条件 :「経路条件」;

  「路線」= token;

  「経路」:: 駅列 : seq of「駅」
           路線列 : seq of「路線」
   inv p == len p.駅列 = len p.路線列 + 1;

 operations

  「経路探索」: () ==>「経路探索」
  「経路探索」() ==
    is not yet specified;

  最適経路を求める :「探索条件」==> seq of「経路」
  最適経路を求める(a探索条件) ==
    is not yet specified
  pre
    到達可能である(a探索条件.経路条件)
  post
    forall i,j in set inds RESULT &
      i < j => 経路評価(a探索条件.優先条件, RESULT(i))
                 <= 経路評価(a探索条件.優先条件, RESULT(j))
    and
      elems RESULT = 全ての経路 (a探索条件.経路条件);
```

```
  経路評価 :「優先条件」*「経路」==> int
    経路評価(a優先条件, a経路) ==
      return
        cases a優先条件:
          <料金> -> 料金計算(a経路),
          <時間> -> 時間計算(a経路),
          <乗換> -> 乗換計算(a経路)
      end;

    料金計算 :「経路」==> int
    料金計算(a経路) ==
      is not yet specified;

    時間計算 :「経路」==> int
    時間計算(a経路) ==
      is not yet specified;

    乗換計算 :「経路」==> int
    乗換計算(a経路) ==
      is not yet specified;

    全ての経路 :「経路条件」==> set of「経路」
    全ての経路(a経路条件) ==
      is not yet specified;

    到達可能である :「経路条件」==> bool
    到達可能である(a経路条件) ==
      return card 全ての経路(a経路条件) > 0;

end「経路探索」
```

機械処理を行い、構文エラー、型エラーは無い状態になっている
「実行して検証」するためには陽仕様化が必要

「仕様記述」を考える例2

「休日」の考察

法律の文章と「仕様」

- この考察の過程では、休日の定義に大きく関わる法律「国民の祝日に関する法律」を参照する。
- 法律の文章を「形式仕様記述言語」で書き直してみると扱いやすくなるのかが大きなテーマである。
- 例題の記述には VDM++ を利用する。もしこうした試みがうまくいくようなら、自然言語ではわかりにくい「規格書」や「標準書」なども記述してみる価値がでてくることになる。

休業日の定義

- 週休二日の組織では、例えば。。。
  - （1）土曜日
  - （2）日曜日
  - （3）年末年始
  - （4）「国民の祝日に関する法律」に定める休日
  - （5）特別な日（創立記念日など）
- といった日を休業日と定めていることが多い

以下では主に（4）の「国民の祝日に関する法律」に定める休日を考察していく

国民の祝日に関する法律
（昭和２３年法律第１７８号）

第一条　自由と平和を求めてやまない日本国民は、美しい風習を育てつつ、よりよき社会、より豊かな生活を築きあげるために、ここに国民こぞつて祝い、感謝し、又は記念する日を定め、これを「国民の祝日」と名づける。

国民の祝日に関する法律
（昭和23年法律第178号）

第二条　「国民の祝日」を次のように定める。
元日　一月一日　年のはじめを祝う。
成人の日　一月の第二月曜日　おとなになつたことを自覚し、みずから生き抜こうとする青年を祝いはげます。
建国記念の日　政令で定める日　建国をしのび、国を愛する心を養う。
春分の日　春分日　自然をたたえ、生物をいつくしむ。
昭和の日　四月二十九日　激動の日々を経て、復興を遂げた昭和の時代を顧み、国の将来に思いをいたす。
憲法記念日　五月三日　日本国憲法 の施行を記念し、国の成長を期する。
みどりの日　五月四日　自然に親しむとともにその恩恵に感謝し、豊かな心をはぐくむ。
こどもの日　五月五日　こどもの人格を重んじ、こどもの幸福をはかるとともに、母に感謝する。
海の日　七月の第三月曜日　海の恩恵に感謝するとともに、海洋国日本の繁栄を願う。
敬老の日　九月の第三月曜日　多年にわたり社会につくしてきた老人を敬愛し、長寿を祝う。
秋分の日　秋分日　祖先をうやまい、なくなつた人々をしのぶ。
体育の日　十月の第二月曜日　スポーツにしたしみ、健康な心身をつちかう。
文化の日　十一月三日　自由と平和を愛し、文化をすすめる。
勤労感謝の日　十一月二十三日　勤労をたつとび、生産を祝い、国民たがいに感謝しあう。
天皇誕生日　十二月二十三日　天皇の誕生日を祝う。

国民の祝日に関する法律
（昭和２３年法律第１７８号）

第三条　「国民の祝日」は、休日とする。
２　「国民の祝日」が日曜日に当たるときは、その日後においてその日に最も近い「国民の祝日」でない日を休日とする。
３　その前日及び翌日が「国民の祝日」である日（「国民の祝日」でない日に限る。）は、休日とする。

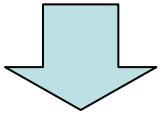
「国民の祝日に関する法律」の目的は？

- 定められた「祝日」を基に、この法律の定める「休日」を定義するのが「目的」
  - 祝日集合＝休日集合ではなく
  - 祝日集合⊆休日集合である
- いわゆる「振り替え休日」が存在するためこうしたズレが生じることになった

「休日」の仕様化

第三条　「国民の祝日」は、休日とする。
２　「国民の祝日」が日曜日に当たるときは、その日後においてその日に最も近い「国民の祝日」でない日を休日とする。
３　その前日及び翌日が「国民の祝日」である日（「国民の祝日」でない日に限る。）は、休日とする。

簡単にアウトラインを書くと、この法律によって定められる休日は以下のようなものになる



休日 =「国民の祝日」集合
　　∪ いわゆる「振替休日」集合
　　∪「前日翌日が祝日である日」集合

∪←この記号は和集合を表す。AUBはA「または」Bに含まれる要素の集合

祝日と振替休日だけを考えた最初の版

休日の定義

休日 =「国民の祝日」集合
　　∪ いわゆる「振替休日」集合

VDM++による記述（部分）

```
休日？ :「日付」-> bool
休日？(date) ==
  date in set dunion {国民の祝日集合(), 振替休日集合()};
```

ここでは 休日？(date) という関数を定義している。date を渡すと、休日か否かを判定して true/false を返す関数である。

国民の祝日集合(). 振替休日集合() はそれぞれ相当する日付の集合を返す関数であるとする。

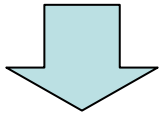
なお

a in set S という書き方は、普通の数学表記では a ∈ S

dunion {A, B, C} という書き方は、A∪B∪Cに相当している

振替休日の仕様

２　「国民の祝日」が日曜日に当たるときは、その日後においてその日に最も近い「国民の祝日」でない日を休日とする。



```
振替休日集合 : () -> set of「日付」
振替休日集合() ==
        {最初の平日(d) | d in set (日曜日集合()
                                    inter 国民の祝日集合())};
```

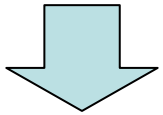
なお 最初の平日(d) という関数は、d を渡すと d 以降最初の平日（国民の祝日でもなく日曜日でもない）を返す。日曜日と国民の祝日が重なった日の集合から、最初の平日を求めて集合にしている

また

A inter B という書き方は、普通の数学表記では A∩Bに相当

祝日に挟まれた「日」の仕様

３　その前日及び翌日が「国民の祝日」である日（「国民の祝日」でない日に限る。）は、休日とする。



まずこの項に相当する「日」の集合を考える

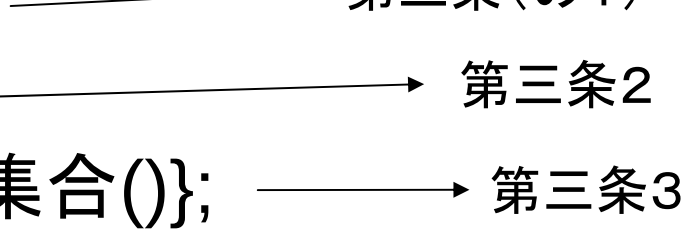
```
祝日に挟まれた平日集合 : () -> set of 「日付」
祝日に挟まれた平日集合() ==
        { d | d in set 全日付集合() &
            d not in set 国民の祝日集合() and
            前日(d) in set 国民の祝日集合() and
            翌日(d) in set 国民の祝日集合()};
```

前日(d), 翌日(d) はそれぞれ d で指定される日の

前日、翌日を返す関数

条文を全て反映した「休日」関数

休日？ :「日付」-> bool
休日？(date) ==
　date in set dunion
　{国民の祝日集合(),
　 振替休日集合(),
　 祝日に挟まれた平日集合()};



「休日」クラス全体像

```
class「休日」

    types

      public「日付」= token;

    functions

      休日？:「日付」-> bool
      休日？(date) ==
        date in set dunion {日曜日集合(), 国民の祝日集合(),
                            振替休日集合(), 祝日に挟まれた平日集合()};

      振替休日集合 : () -> set of「日付」
      振替休日集合() ==
        {最初の平日(d) | d in set (日曜日集合() inter 国民の祝日
集合())};

      日曜日集合 : () -> set of「日付」
      日曜日集合() ==
        is not yet specified;

      国民の祝日集合 : () -> set of「日付」
      国民の祝日集合() ==
        is not yet specified;

      祝日に挟まれた平日集合 : () -> set of「日付」
      祝日に挟まれた平日集合() ==
        { d | d in set 全日付集合() &
                        d not in set 国民の祝日集合() and
                        前日(d) in set 国民の祝日集合() and
                        翌日(d) in set 国民の祝日集合()};

      最初の平日 :「日付」->「日付」
      最初の平日(date) ==
        is not yet specified;

      前日 :「日付」->「日付」
      前日(date) ==
        is not yet specified;

      翌日 :「日付」->「日付」
      翌日(date) ==
        is not yet specified;

      全日付集合 : () -> set of「日付」
      全日付集合() ==
        is not yet specified;

end「休日」
```

陰仕様で表したもの。あと中身を少し埋めると実行可能となる。2009年９月22日は休日だろうか？

とりあえずのまとめ

形式仕様記述の効果（1）

- 開発対象の要件記述の精度向上
  - 要件記述のうち各種「定義」にかかわる部分を形式仕様記述言語で記述することにより、記述精度を向上させ、曖昧さや誤りによる手戻りを防ぐことができる
- 例えば
  - 要件記述の型と構文を検査することにより、用語や条件式や計算式などの基本的な誤りを除去
  - 問題領域に存在する制約条件を記述しておくことにより要求の矛盾を検出

形式仕様記述の効果（2）

- 仕様の様々な可視化
  - しっかりとしたモデル（記述）を中心におくことで、仕様を様々な角度から可視化（文書化）できる
- 誤りのない仕様の作成
  - きちんとした仕様を元に検証を行い、開発の早い段階で仕様の誤りを取り除きやすくなる
- 曖昧さのない仕様の作成
  - 前提条件や果たすべき機能、データの型や各種条件などを厳密に定義して設計者（開発者）に伝えることができる
- 実装検証の支援
  - 仕様検証にもちいたテストケースを実装の検証にも流用することができる

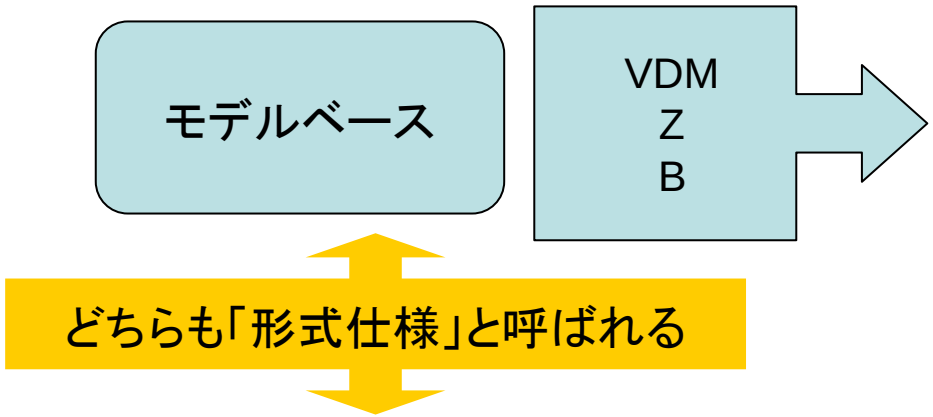
形式仕様記述により
更に期待される副次的効果

- 仕様の資産化
  - 検証済みの仕様はそれ自身貴重な資産となり、再利用性も向上する
- 仕様変更後の検証負荷を軽減
  - 仕様の回帰テストを行うことにより、仕様変更がシステム全体に及ぼす影響をより少ない負荷で検証できる
- 仕様の差分管理が可能に
  - テキスト記述ならば、通常のファイルと同様なやりかたで差分管理が可能である、構成管理システムと組み合わせれば仕様変更量を追跡することも可能
- プロジェクト管理の定量化
  - 仕様管理が精密化することにより、開発プロジェクトの各種管理指標の定量化が容易になる

本日の結論（再）

- いますぐ試してみよう
  - 形式手法は明日の技術ではなく、いまそこにある技術。活用すれば仕事を劇的に楽にする
- コミュニケーション
  - 怠惰な技術者は「形式手法」で楽をしよう、他者との対話自分との対話の基礎を固めて、「考えること」に時間を割こう

代表的な手法・言語（再）



VDM
Z
B

情報の構造や、ある状態から他の状態への変化をモデル化
定理証明、回帰テスト

本日の中心

曖昧さの除去、専用言語の利用によるコンパクトな記述、仕様の「実行」「評価」可能性
汎用の記述に使える

PROMELA
SMV
LTSA

振舞いのモデル化
全状態空間を生成し自動検査

従来の「テスト」では見つからない潜在的な障害を発見できる

New Face

Alloy
(Larch)

段階的モデル構築
自動ツールによるモデル分析

「モデル検査」よりも複雑な構造と条件を記述・検査できる

もうひとこと

- で、結局「形式仕様記述って何？」

書いたこと、考えたことを「無駄にしない」技術