「形式手法と仕様記述」の探求に向けて

～形式手法とは？～

国立情報学研究所　石川　冬樹
f-ishikawa@nii.ac.jp

自己紹介

石川 冬樹

from 国立情報学研究所 http://www.nii.ac.jp/

http://research.nii.ac.jp/~f-ishikawa/

- 産業界向け形式手法入門・活用
  - 「トップエスイー」講師
    （SQiPよりも「大学」っぽい？単発セミナーも）
    http://topse.jp/ http://topse.or.jp/
  - 実践ポータルサイト http://formal.mri.co.jp/
  - VDM++入門本
- 本職は研究のお仕事も
  - Webサービス・クラウドも本職，法解釈なんかも

この資料の内容

- 「形式手法」「形式仕様」とは何なのかを知る
  - どういう問題に注目しているのか？
  - どういう考え方で解決を目指すのか？
  - 具体的にはどういう手法・ツールがあるのか？
  - 活用するために何を身につけ，検討するのか？
- 「形式仕様」と「仕様記述」？

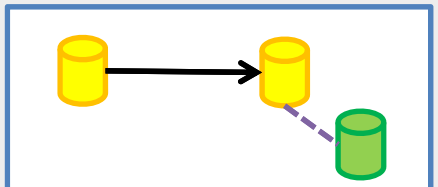
目次

- 「形式手法」「形式仕様」とは何なのかを知る
  - どういう問題に注目しているのか？
  - どういう考え方で解決を目指すのか？
  - 具体的にはどういう手法・ツールがあるのか？
  - 活用するために何を身につけ，検討するのか？
- 「形式仕様」と「仕様記述」？

仕様

- 重要： すべての基になる
  - 計画，設計・実装，テスト定義，チーム間や外注先との共有，派生・修正時の理解・分析，・・・

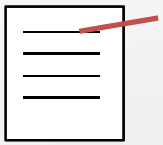
どう誤解の原因（曖昧さ等）をつぶしている？
どう誤り（不整合等）をつぶしている？
（どう前後の成果物とつないでいる？）



1 もしパケットがメッセージを含まなければ（パケット長がパケットヘッダのサイズ以下ならば），パケットは廃棄されなければならない.
2 転送条件
2.1 ・・・

問題（１）：曖昧さ

- 記述においてその意味が一意に定まらない
  - 記法（日本語，図等）自体で一意に決まっていない
  - 一意に定め用いる運用を徹底していない・できない



フラグ1がONになるまではフラグ2はOFF

フラグ1がONになった後に限り，必要なタイミングで，フラグ2をONにして欲しい

フラグ1がONとなると同時にフラグ2をONに

参加者は研究会の開始前に参加するコースを選択する

参加者は研究会の開始前に出席簿に記入する

事務は研究会終了時に担当講師への謝金振込を行う

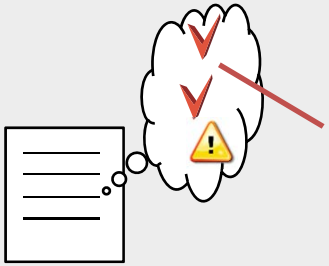
えーと振込タイミングは・・・

問題（2）：誤り（不整合）

- 「正しくない」振る舞いを定めてしまう
  - 「何が正しくて何が正しくないか」という基準を明確に洗い出し，明記，共有していない
  - その基準を満たしているかどうかを，一定程度の信頼ができる方法で分析，検証していない

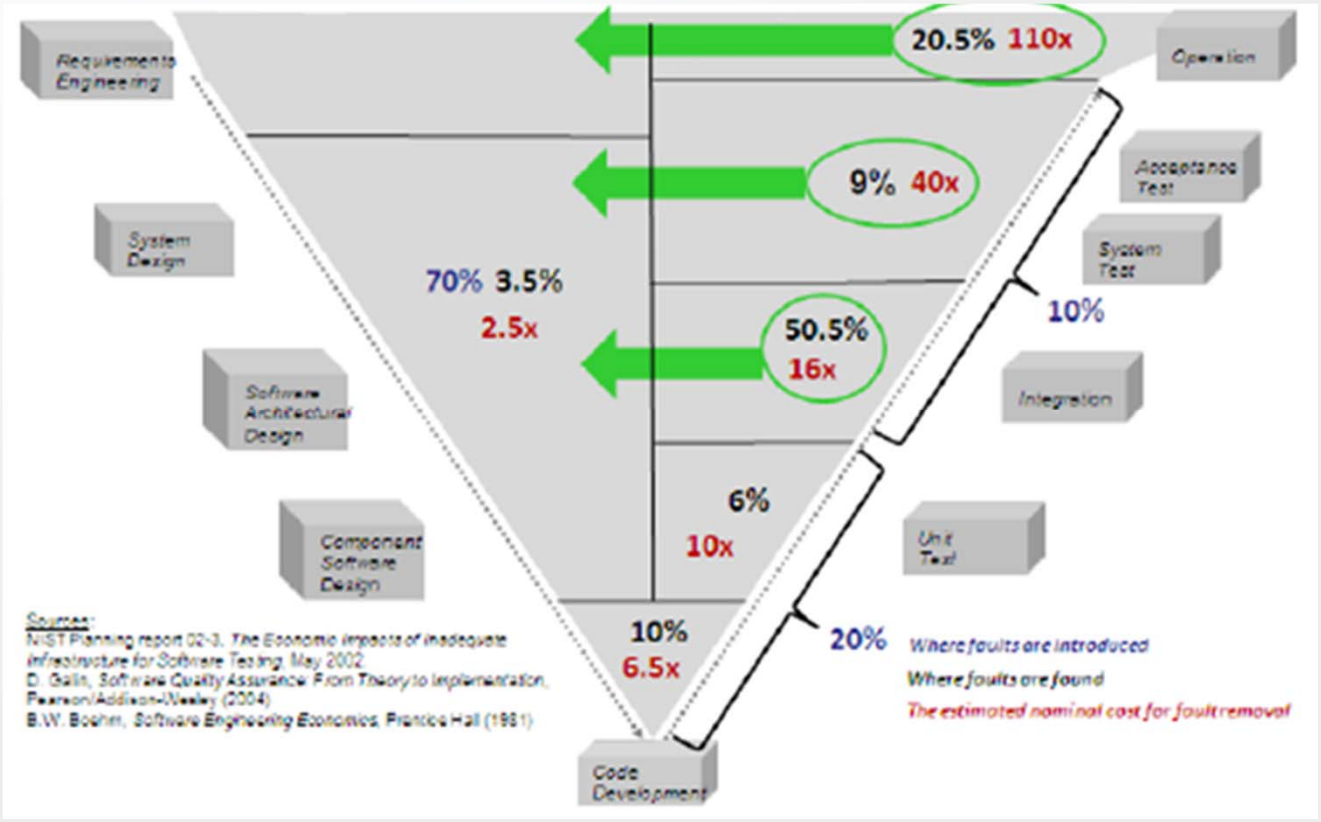
ダブルブッキングが起きることはない

様々な種類の予約操作の一部において，
ダブルブッキング防止のチェックが抜けている



複数プロセスが特定のタイミングでそれぞれ
リクエストを実行しロックをかけると
デッドロックが発生してしまうことがある

補足： 開発の段階と不具合修正コスト



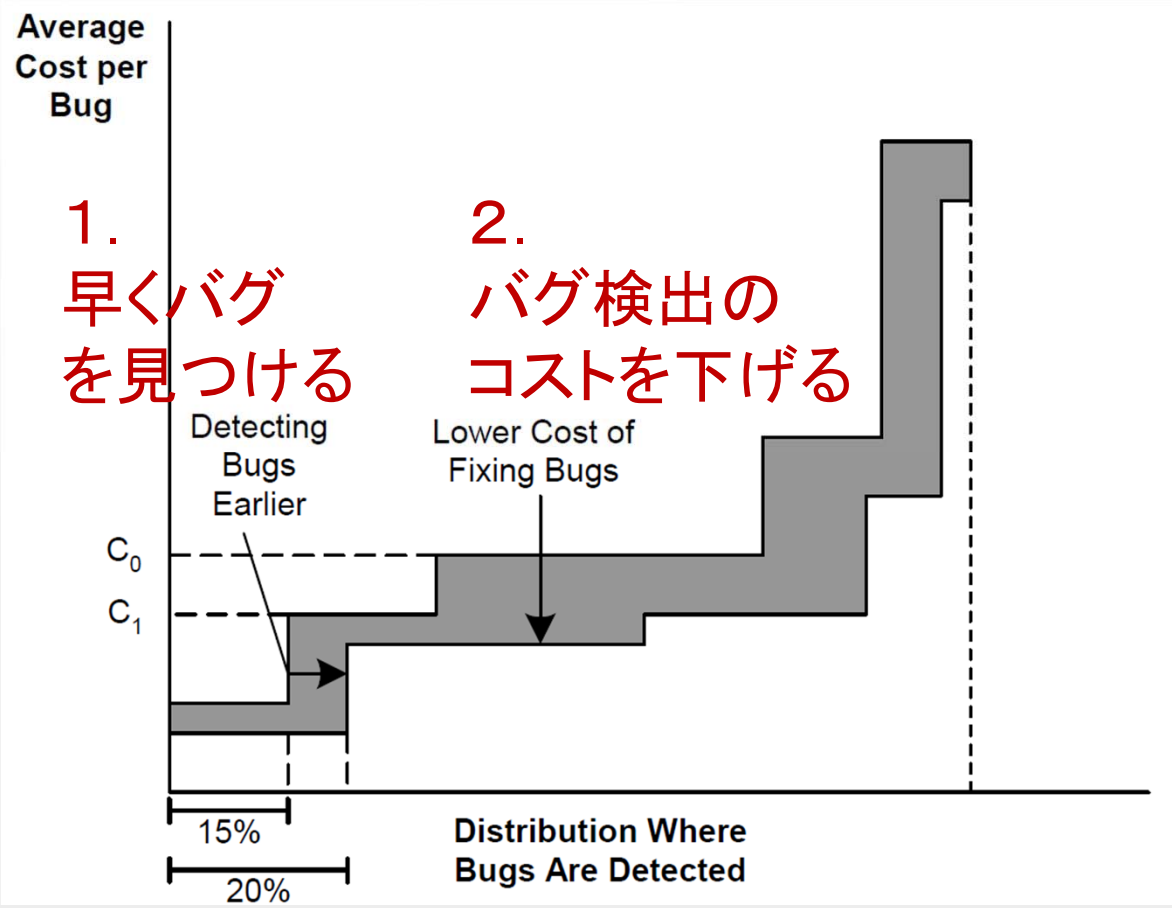
青字：
不具合が埋め込まれる場所

黒字：
不具合が発見される場所

赤字：
不具合の除去におけるコストの増加率

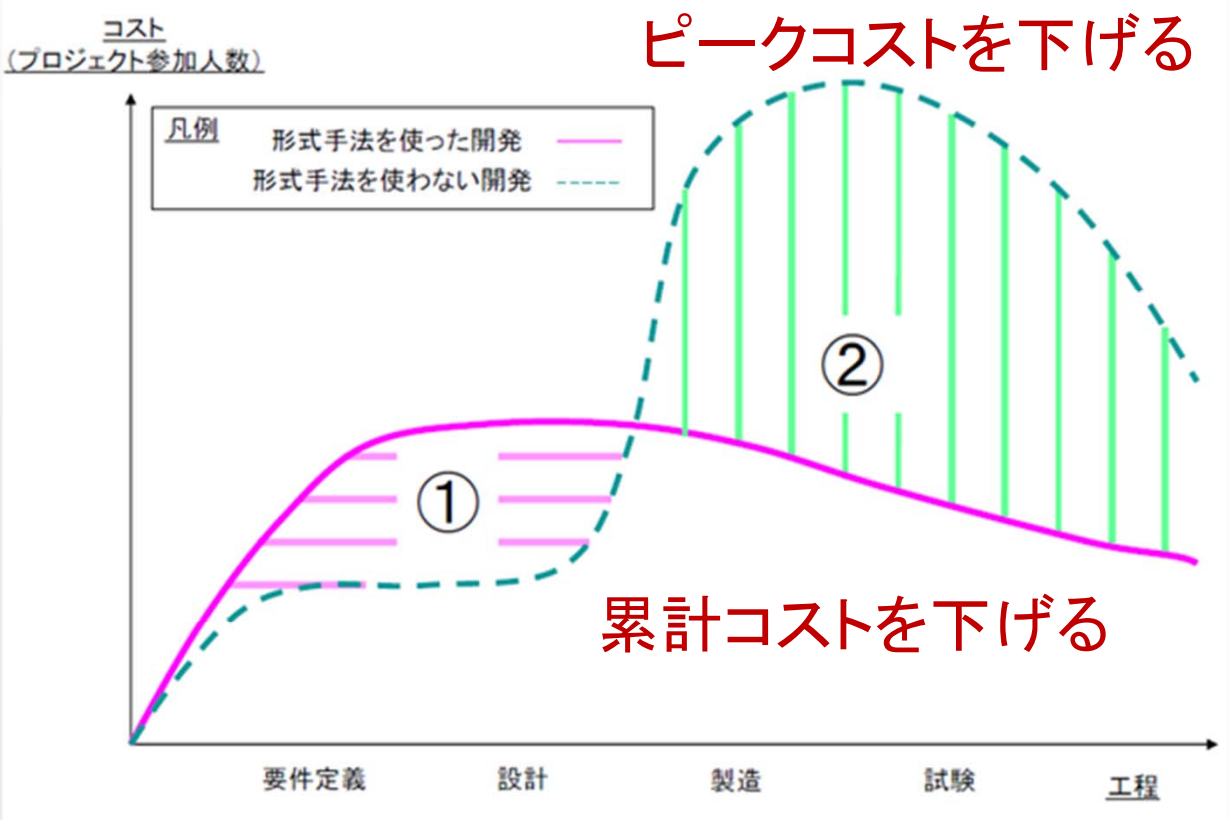
[P. H. Feilerら, System Architecture Virtual Integration: An Industrial Case Study, 2009]

補足： 開発の段階と不具合修正コスト



[The Economic Impacts of Inadequate Infrastructure for Software Testing, NIST, 2002]

補足： 開発の段階と不具合修正コスト



[DSF Webサイト内の紹介資料]

目次

- 「形式手法」「形式仕様」とは何なのかを知る
  - どういう問題に注目しているのか？
  - どういう考え方で解決を目指すのか？
  - 具体的にはどういう手法・ツールがあるのか？
  - 活用するために何を身につけ，検討するのか？
- 「形式仕様」と「仕様記述」？

形式手法とは？（定義）

数理論理学等に基づき品質の高い
ソフトウェアを効率よく開発するための
科学的・系統的アプローチ

システムの注目する側面を正確に，曖昧さのない言語で表現する

曖昧さや思い込みを開発プロセスの早期に排除し，手戻りによるコストを防ぐ

システムの満たす性質について科学的・系統的な分析・検証を行い，品質を高める

形式手法とは？（もっと軽く）

- 仕様・設計を「きちんと」書く過程により
  - 曖昧さを解消することになる
  - 不正確さ，不整合も表面化する
  - システムに対する理解が深まる
- 仕様や設計を「きちんと」書いた結果により
  - 誤解なく情報を共有できる
  - 科学的・系統的な分析・検証ができ，特にツールに支援させることができる

厳密な文法・意味論を持つ言語を用い

数理論理学がベース

内容・方法は様々

➡ ***開発早期における信頼性確保***

※分析・検証の仕組みは，プログラムに対して活用することも多い

形式手法とは？（プログラムと対比）

- プログラムを書くと
  - 対象が厳密，明確に，書き出される
  - 型チェック，テストなどの分析・検証を行える

厳密な文法・意味論を持つ言語を用い

数理論理学がベース

内容・方法は様々

➡ ***上流の中間成果物（仕様・設計）でも！***

ただしプログラムのように「一通り動かす」ことが目的ではないため，記述，分析・検証の目的に応じ，

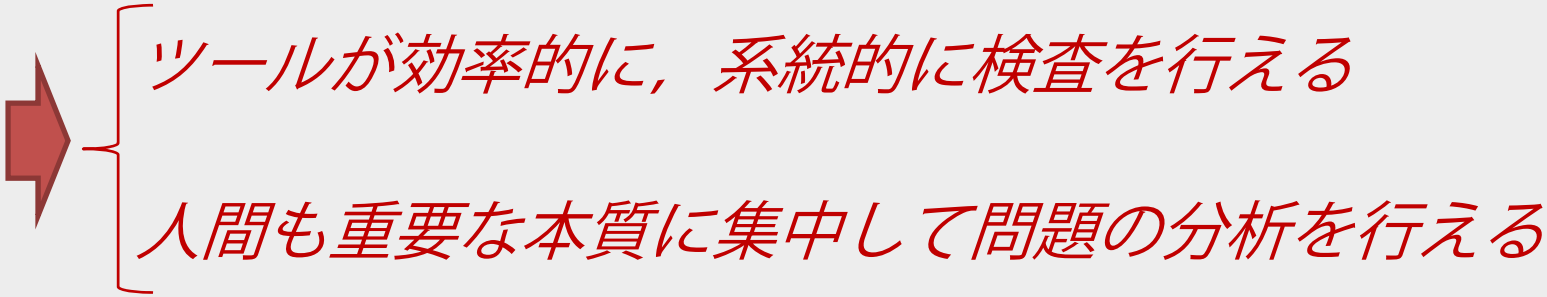
- システム内で扱う対象（範囲・観点）を絞る
- 用いる言語，手法・ツールを選ぶ

ポイント：「モデル」

「モデル」：計算や予測を助けるために
用いられる，システムやプロセスの
単純化された記述（しばしば数学的なもの）

（Oxford英英辞典の石川訳）

- 注目する側面に特化し，抽象化・単純化
  （言い換えると，不要な実現詳細を捨象）



形式手法への（今さらの）注目

- 日本語での書籍が出始め，セミナーも多数
  - SPIN，VDM，Alloy，B，・・・
- 導入促進のための活動が行われている
  - 導入ガイダンス（2011年5月）
    経産省（MRI，NII）
    http://formal.mri.co.jp/
  - 適用手順・イディオム（2011年7月）
    Dependable Software Forum （DSF）
    http://www.nttdata.co.jp/dsf/
  - ・・・

（技術の基礎は数十年前から）

目次

- 「形式手法」「形式仕様」とは何なのかを知る
  - どういう問題に注目しているのか？
  - どういう考え方で解決を目指すのか？
  - 具体的にはどういう手法・ツールがあるのか？
  - 活用するために何を身につけ，検討するのか？
- 「形式仕様」と「仕様記述」？

実際の手法・ツール

先の広い定義に対し，実際には様々な目的に応じて様々なアプローチがある

- ***記述言語***：特定の側面に関する記述に特化していることが多い
  - データ構造？状態遷移？並行動作も書く？
- ***検証方法***：様々で，記述言語とも強く関連する
  - 証明（理屈で正しさを示す）？
  - モデル検査（実行パスを網羅し正しさを示す）？
  - テスト（特定の状況を調べてみる）？
- ***プロセス***：想定する利用工程やその入出力

大ざっぱな分類

- *形式仕様記述*
  - 「仕様」として，システムの状態（データ構造）や，操作やイベントによるその変化を一通り記述する
  - 記述および分析・検証の方針が，手法とツールによって様々である

  代表的な手法：VDM，B，Event-B，Alloy
- *モデル検査*
  - 特に分散・並行システムなど，複雑な状態遷移に絞って記述を行い，網羅的な検証を自動的に行う

    （これが基本的な共通の考え方で，あとはツールの特性が異なるものがいくつか）

  代表的なツール：SPIN，SMV，LTSA，UPPAAL

手法・ツールの例（１）：VDM

（演習コースIIでの取り組み候補の一つ）

- VDM：形式仕様（全体の記述）
  - プログラミングに比較的近い考え方で取り組め，普通のプログラマにとっつきやすい
    - 動かしてみる・テストする
  - よくわからなければ最初の一歩によい
  - 特に分析・検証については，「もっとすごいことができて欲しい」と思うかも
  - それでも「いいものが書ける」「効果が得られる」「活用できる」ためには，悩むこと，議論することがたくさん

手法・ツールの例（２）：SPIN

（演習コースIIでの取り組み候補の一つ）

- SPIN：モデル検査（状態遷移の検証）
  - 並行・分散システムに関するよい勉強になる
    - マルチスレッド，マルチコア，クライアント・サーバ，二重化などを含むシステム
    - デッドロック，安全性，活性，公平性，・・・
  - 排他制御や通信制御などにピンポイントで注目し，実機では再現できないような，タイミングや障害に起因する特定のバグを見つけ出せる
    - 「反例」が出るので，原因追及も行いやすい
  - 逆に，それだけ（強力な検証に特化）

手法・ツールの例（３）：Alloy

（演習コースIIでの取り組み候補の一つ）

- Alloy：形式仕様（全体の記述）
  - モデル検査ツールの良さを採り入れた，強力だが比較的とっつきやすい分析・検証ツールがある
    - 網羅的に調べてみる検証なので，「数学的証明」を行うような検証よりは，とっつきやすい
  - 一方，記述方式に多少クセがあるかも
    - 実装（How）から離れた記述のいい特訓にはなる
  - 制約などの漏れを見つけやすくする，「モデル発見」のツールが面白いかもしれない
    - とにかくいっぱい例を自動生成してみる

VDM： 位置づけ（再）

（演習コースIIでの取り組み候補の一つ）

- VDM： 形式仕様（全体の記述）
  - プログラミングに比較的近い考え方で取り組め，普通のプログラマにとっつきやすい
    - 動かしてみる・テストする
  - よくわからなければ最初の一歩によい
  - 特に分析・検証については，「もっとすごいことができて欲しい」と思うかも
  - それでも「いいものが書ける」「効果が得られる」「活用できる」ためには，悩むこと，議論することがたくさん

VDM： 典型的な例題

- イベント登録管理システム
  - ユーザ，部屋，イベントの情報を管理
    - 「部屋R1の定員は30名である」
    - 「イベントE1は部屋R1で5/11の10-17時に開催」
  - 予約やそのキャンセルの受付
    - 「ユーザ1が，イベントpに対して2名予約登録」

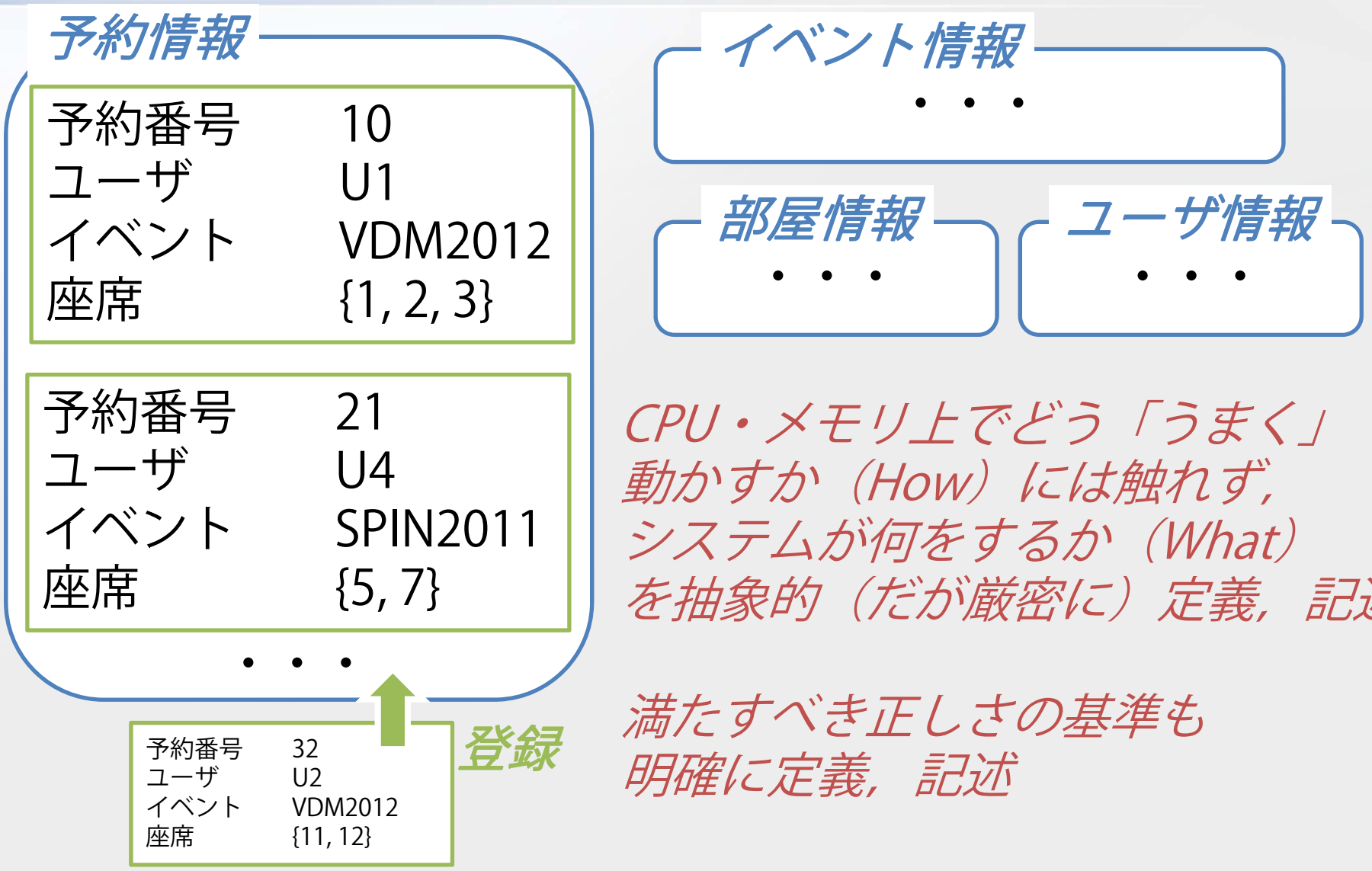
    → 「座席A-35/A-36，予約番号1035Aを割り当て」
    - 「20100035Aの予約をキャンセル」
    - 「イベントqに対し，ユーザ1～9999から5名を抽選で選び予約登録」

VDM： 検討する「正しさ」の基準

（曖昧さの問題に加えて）

- システムの状態が常に満たすべき条件
  - *例： 同じ時間に行われる複数イベントに対し，同じユーザが予約を行っていることはない*
- 操作実行にあたり守るべき前提条件
  - *例： キャンセル操作は，開催日が次の日以降のイベントに対してのみ実行可能である*
- 操作に期待される効果を定める条件
  - *例： 抽選操作の実行結果においては，実行前に対象イベントに対し，予約を行っていなかったユーザが選択されている*

VDM： システム状態のモデル



CPU・メモリ上でどう「うまく」動かすか（How）には触れず，システムが何をするか（What）を抽象的（だが厳密に）定義，記述

満たすべき正しさの基準も明確に定義，記述

VDM： 記述イメージ（1）

```
class EventManager

instance variables

  users : set of User := {};

  rooms : set of Room := {};

  events : set of Event := {};
  inv (forall ev1, ev2 in set events &
          ev1 <> ev2 =>
           ev1.room <> ev2.room or ev1.date <> ev2.date);

  reservations : set of Reservation := {};
  inv …

operations
  …
```

オブジェクト指向型の記述も可能（VDM++言語）

「順序を付けず複数のデータを保持（set）」という抽象データ構造（検索効率等気にしない）

2つのイベントが同じ日時，同じ時間に行われることがないという条件の記述

VDM： 記述イメージ（2）

```
public searchUsersByEvent : Event ==> set of User
searchUsersByEvent(event) ==
  return { res.user | res in set reservations &
                res.event = event }
pre event in set events;
```

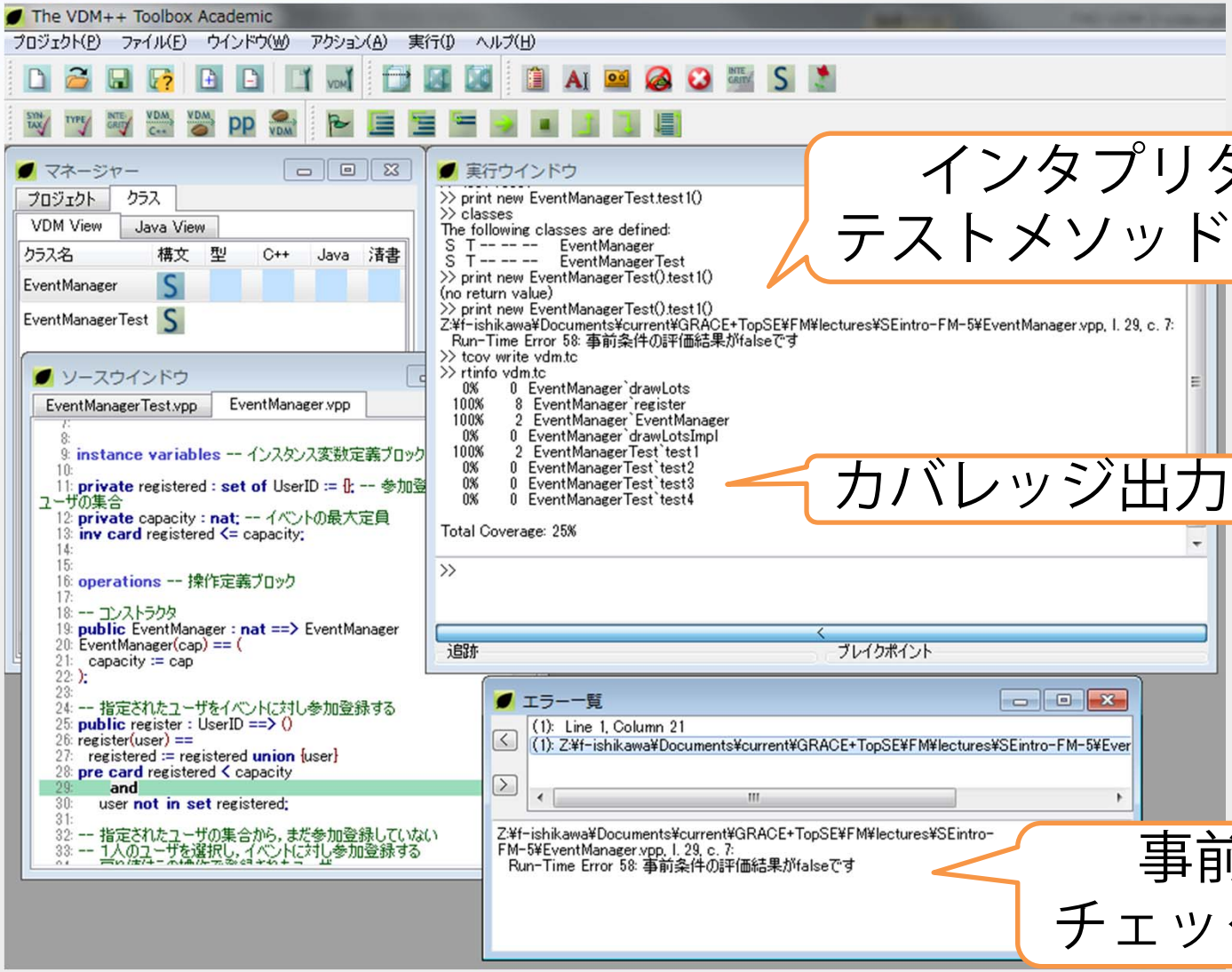
この操作が正しく
実行できる前提条件

条件を満たすユーザを
すべて抜き出すという，
実行可能な記述
（実行効率等気にしない）

```
public sortEventsByUser : set of Event ==> seq of Event
sortEventsByUser(event) == is not yet specified
post forall i, j in set inds RESULT &
        i < j => RESULT(i).date < RESULT(j).date
        and …;
```

戻り値のリストにおいては，イベントが
日付が早い順に並んでいることなど明記
（期待する実行結果だけを記述）

VDM： ツールイメージ


The VDM++ Toolbox Academic
プロジェクト(P) ファイル(F) ウインドウ(W) アクション(A) 実行(I) ヘルプ(H)
マネージャー
プロジェクト クラス
VDM View Java View
クラス名 構文 型 C++ Java 清書
EventManager
EventManagerTest
ソースウインドウ
EventManagerTest.vpp EventManager.vpp
9: instance variables -- インスタンス変数定義ブロック
11: private registered : set of UserID := {}; -- 参加登録ユーザの集合
12: private capacity : nat; -- イベントの最大定員
13: inv card registered <= capacity;
16: operations -- 操作定義ブロック
18: -- コンストラクタ
19: public EventManager : nat ==> EventManager
20: EventManager(cap) == (
21: capacity := cap
22:);
24: -- 指定されたユーザをイベントに対し参加登録する
25: public register : UserID ==> ()
26: register(user) ==
27: registered := registered union {user}
28: pre card registered < capacity
29: and
30: user not in set registered;
32: -- 指定されたユーザの集合から，まだ参加登録していない
33: -- 1人のユーザを選択し，イベントに対し参加登録する
実行ウインドウ
>> print new EventManagerTest.test1()
>> classes
The following classes are defined:
S T -- -- -- EventManager
S T -- -- -- EventManagerTest
>> print new EventManagerTest().test1()
(no return value)
>> print new EventManagerTest().test1()
Z:¥f-ishikawa¥Documents¥current¥GRACE+TopSE¥FM¥lectures¥SEintro-FM-5¥EventManager.vpp, l. 29, c. 7:
Run-Time Error 58: 事前条件の評価結果がfalseです
>> tcov write vdm.tc
>> rtinfo vdm.tc
0% 0 EventManager`drawLots
100% 8 EventManager`register
100% 2 EventManager`EventManager
0% 0 EventManager`drawLotsImpl
100% 2 EventManagerTest`test1
0% 0 EventManagerTest`test2
0% 0 EventManagerTest`test3
0% 0 EventManagerTest`test4
Total Coverage: 25%
>>
追跡 ブレイクポイント
エラー一覧
(1): Line 1, Column 21
(1): Z:¥f-ishikawa¥Documents¥current¥GRACE+TopSE¥FM¥lectures¥SEintro-FM-5¥Ever
Z:¥f-ishikawa¥Documents¥current¥GRACE+TopSE¥FM¥lectures¥SEintro-FM-5¥EventManager.vpp, l. 29, c. 7:
Run-Time Error 58: 事前条件の評価結果がfalseです
インタプリタから
テストメソッド呼び出し
カバレッジ出力
事前条件
チェックエラー

SPIN： 位置づけ（再）

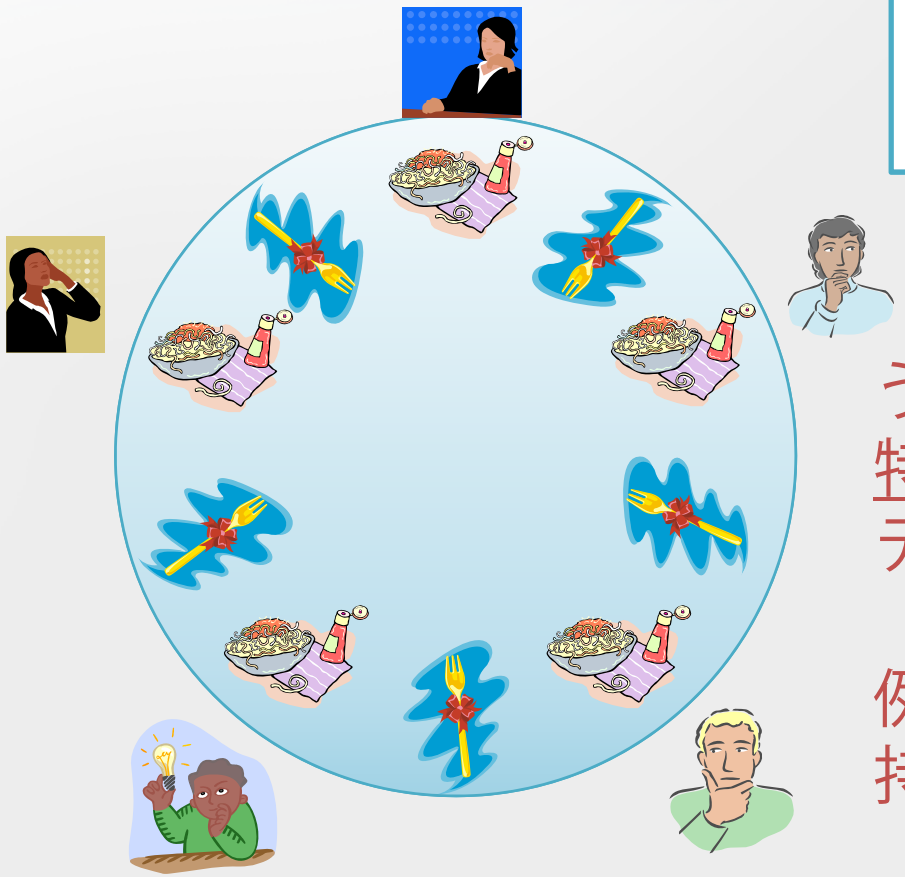
（演習コースIIでの取り組み候補の一つ）

- SPIN： モデル検査（状態遷移の検証）
  - 並行・分散システムに関するよい勉強になる
    - マルチスレッド，マルチコア，クライアント・サーバ，二重化などを含むシステム
    - デッドロック，安全性，活性，公平性，・・・
  - 排他制御や通信制御などにピンポイントで注目し，実機では再現できないような，タイミングや障害に起因する特定のバグを見つけ出せる
    - 「反例」が出るので，原因追及も行いやすい
  - 逆に，それだけ（強力な検証に特化）

SPIN： 典型的な例題

- 食事する哲学者
  - 並行プログラムの比喩

左右どちらかのフォークを
とった後，もう片方をとり，
食事をしてフォークを戻す
（会話はしない）
ようなプログラムを設計



うまく行動を定めないと，
特定の振る舞いの順序で
デッドロックやライブロック

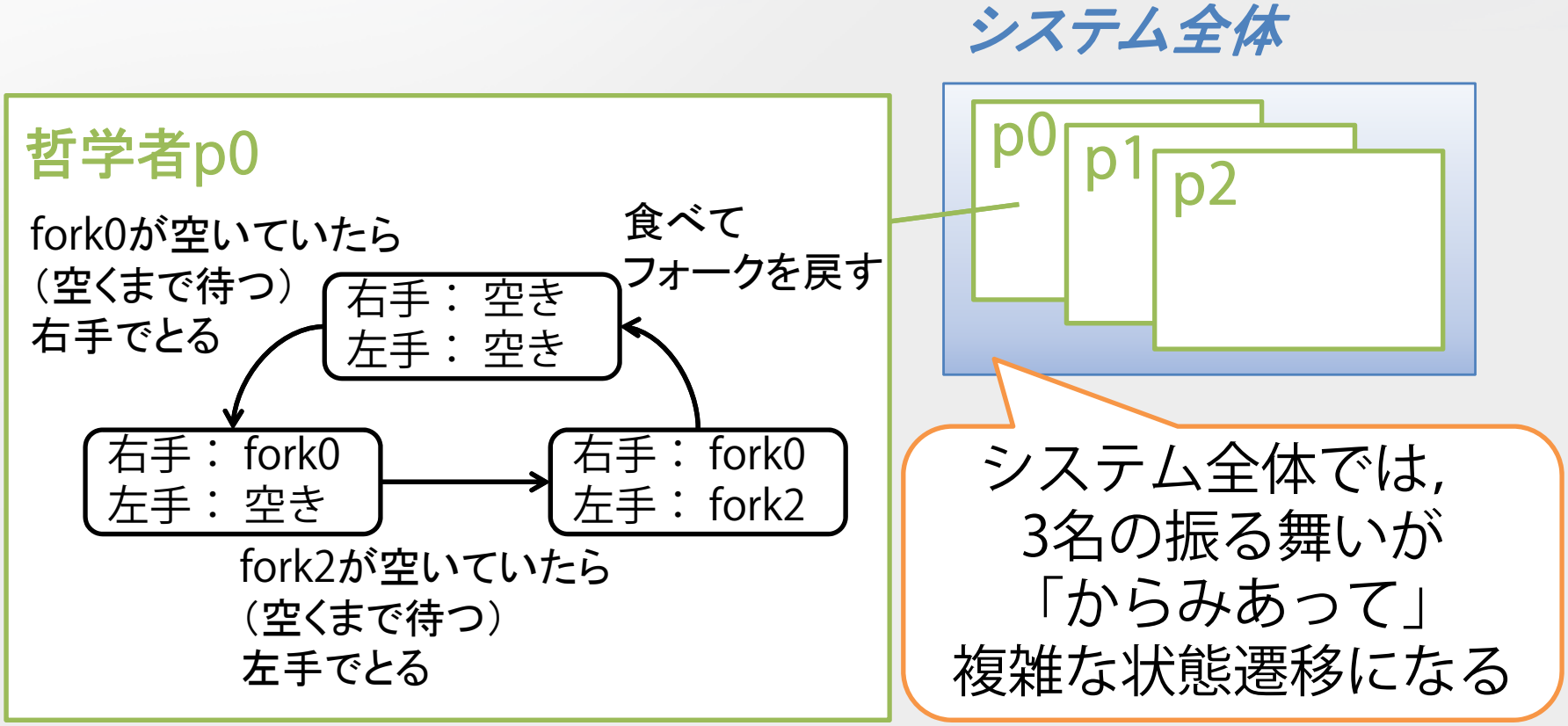
例： 全員が左のフォークを
持ち右が空くのを待ち続ける

SPIN： 検討する「正しさ」の基準

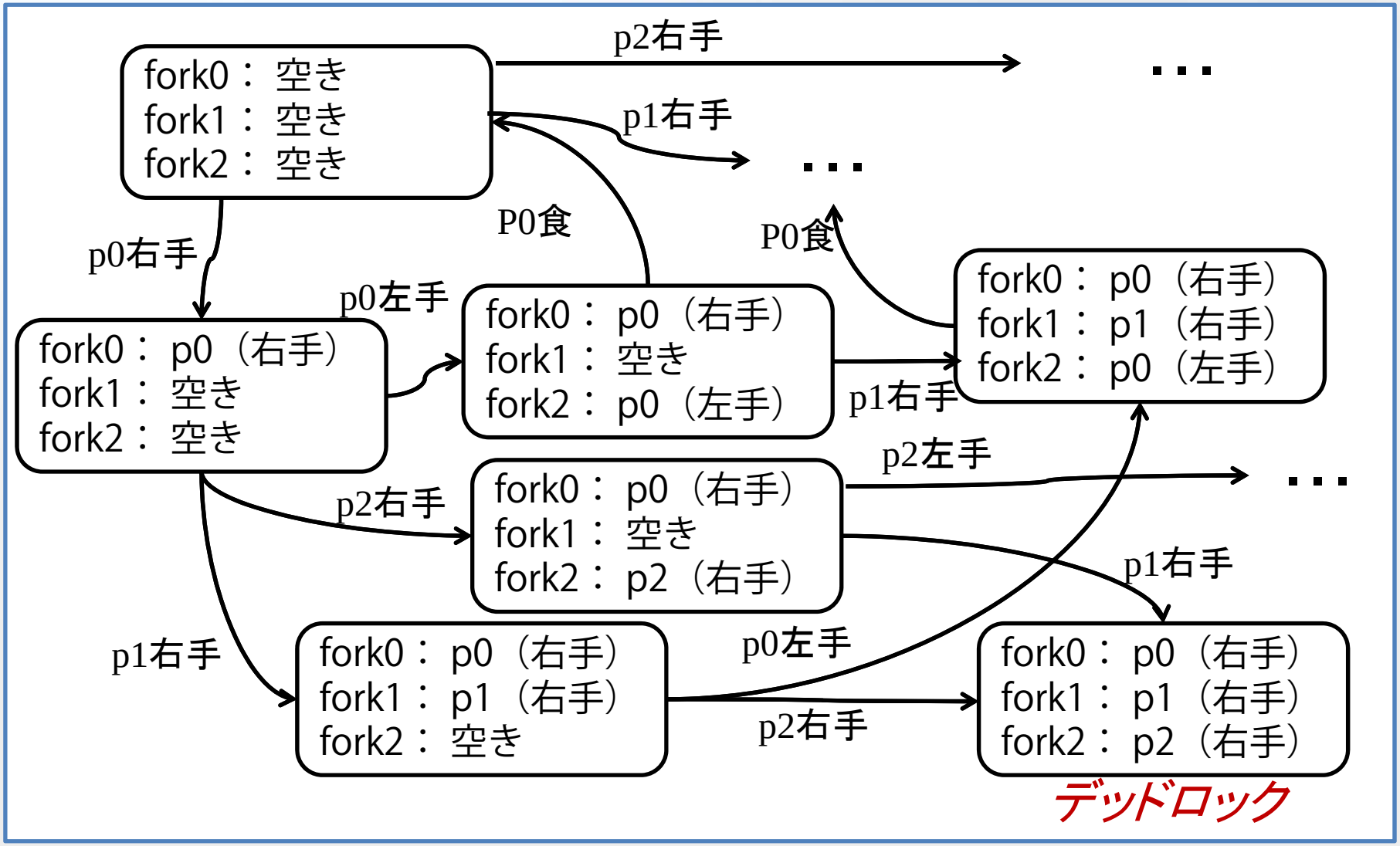
- デッドロックフリー
  - 誰も何もできなくなる状況に到達しない
- 進行性（活性）
  - 食べる以外の行動（様子見・断念してフォークを置く等）を全員が繰り返すだけの状況に陥らない
- 公平性
  - ある哲学者が（ある定義で）「ずっと」食べることができないような状況に陥らない

SPIN： 状態遷移のモデル

- 「右手側からフォークをとる（とったら戻さない）」ルール・3人版



SPIN： 状態遷移のモデル（全体）



SPIN： 記述例

```
mtype = {p0, p1, p2, none};
mtype fork[3] = none;

active proctype P0(){
  do
    :: atomic{fork[0] == none -> fork[0] = p0};
       atomic{fork[2] == none -> fork[2] = p0};
       skip;
       fork[2] = none;
       fork[0] = none;
  od
}
...
```

列挙型

原子的に，フォークが空いているなら確保

「食べる」ことは今回のモデル化で本質ではないので捨象

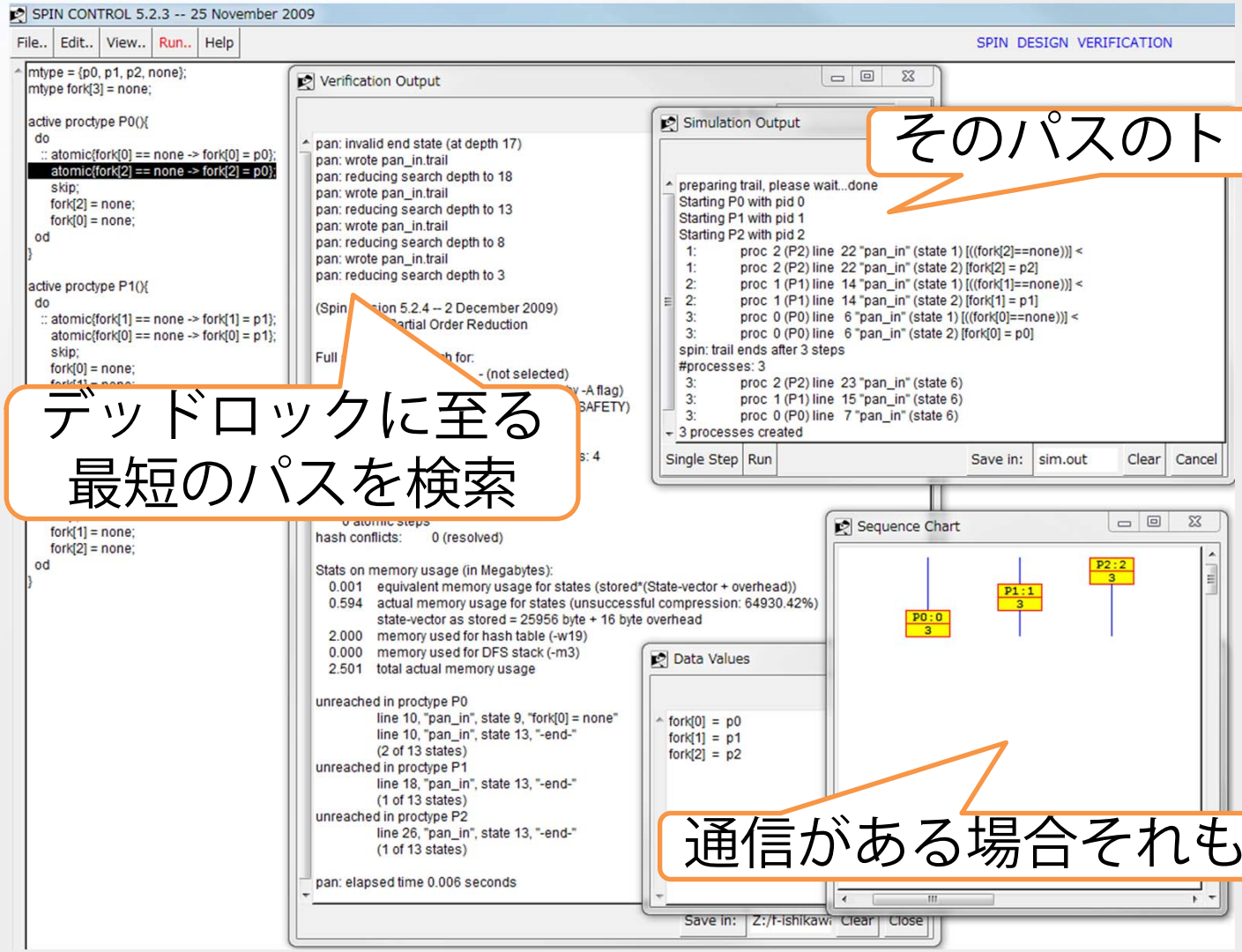
合成はツール側で

```
[ ] (fork[2]==p2 -> <> fork[1]==p2)
```

正しさの基準も

P2が右手のフォークをとったときはいつでも，その後いつか左手のフォークをとる

SPIN： ツールイメージ



SPIN： 別の記述例（通信含む）

```
#define SIZE 4

byte msg[SIZE];

chan s2r = [2] of {byte};

proctype Sender() {
  byte i;
  do
    :: i == SIZE -> break;
    :: else -> s2r ! msg[i];
               i++;
  od
}
```

バッファ長さ2，byte型の通信チャンネル

配列内のメッセージを順次送信

```
proctype Receiver() {
  byte j;
  byte rmsg;
  do
    :: j == SIZE -> break;
    :: else -> s2r ? rmsg;
       assert (rmsg == msg[j]);
       j++;
  od
}

proctype Lost() {
  byte drop;
  do
    :: s2r ? drop;
  od
}
```

メッセージを受信，順番通りか検証

通信の失敗を表現（メッセージを受信してしまう）

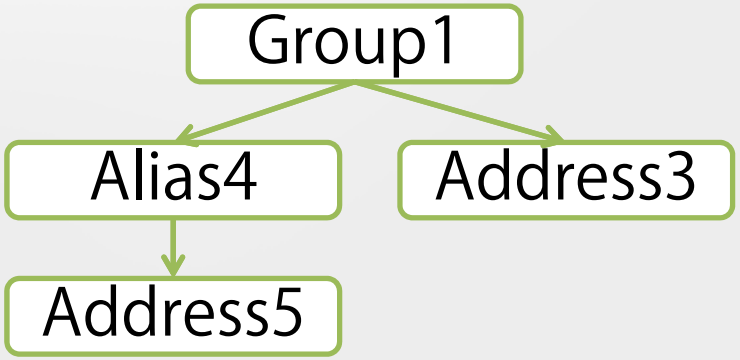
Alloy： 位置づけ（再）

（演習コースIIでの取り組み候補の一つ）

- Alloy： 形式仕様（全体の記述）
  - モデル検査ツールの良さを採り入れた，強力だが比較的とっつきやすい分析・検証ツールがある
    - 網羅的に調べてみる検証なので，「数学的証明」を行うような検証よりは，とっつきやすい
  - 一方，記述方式に多少クセがあるかも
    - 実装（How）から離れた記述のいい特訓にはなる
  - 制約などの漏れを見つけやすくする，「モデル発見」のツールが面白いかもしれない
    - とにかくいっぱい例を自動生成してみる

Alloy：一つの例題

- メールアドレス帳の仕様定義
  - グループには1個以上の要素が含まれる
  - エイリアスは1個だけの要素を含む



[抽象によるソフトウェア設計－Alloyではじめる形式手法－] より

Alloy：記述イメージ（1）

```
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}
sig Alias, Group extends Name {}
sig Book {
  names: set Name,
  addr: names -> some Target }
  {
    all a: Alias | lone a.addr
  }
```

型やクラスのようなもの（signature）を定義

AliasかGroupからは少なくとも1つのTargetを指す
{ (Group1, Alias4), (Alias4, Address5), …}

Aliasは高々1つのTargetを指す

[抽象によるソフトウェア設計－Alloyではじめる形式手法－] より

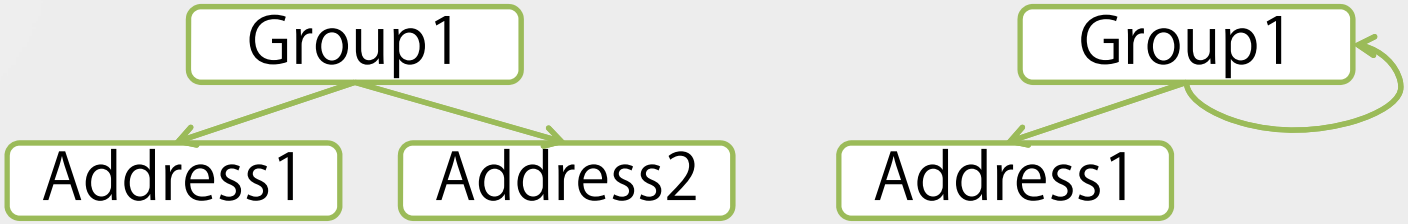
Alloy：ツールイメージ（１）

- この時点でツールを使ってみる
  （Alloy Analyzer）

```
pred show (b: Book) {
  #b.addr > 1
}
run show for 3 but 1 Book
```

リンクを2個以上含むような
構造例を表示してみる

諸々を最大3個生成して
（アドレス帳だけは1個）
構造例を探す



適当に名前が付いて
いろいろ生成される

Alloy：記述イメージ（１・修正）

```
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}
sig Alias, Group extends Name {}
sig Book {
  names: set Name,
  addr: names -> some Target }
  {
    all a: Alias | lone a.addr
    no n: Name | n in n.^(addr)
  }
```

どの要素からも，リンクを
任意の回数たどって，
自分に戻ってくることはない

[抽象によるソフトウェア設計－Alloyではじめる形式手法－] より

Alloy：記述イメージ（2）

```
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}
sig Alias, Group extends Name {}
sig Book {
  names: set Name,
  addr: names -> some Target }
  {
    all a: Alias | lone a.addr
    no n: Name | n in n.^(addr)
  }
pred add (b, b’: Book, n: Name, t: Target)
  {b’addr = b.addr + n -> t}
pred del (b, b': Book, n: Name, a: Addr)
  {b'.addr = b.addr - n-> a}
```

リンクの追加と削除に関する操作定義を追加

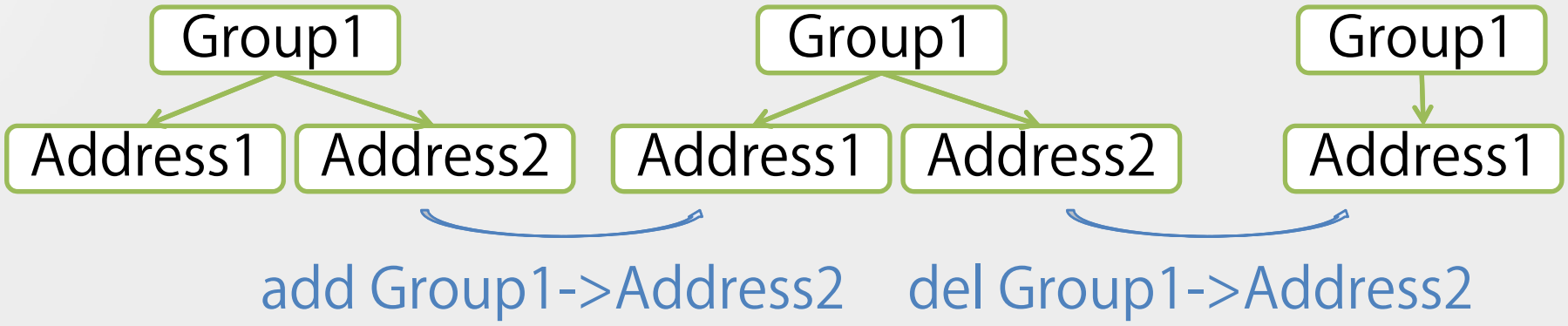
[抽象によるソフトウェア設計－Alloyではじめる形式手法－] より

Alloy：ツールイメージ（2）

```
assert delUndoesAdd{
  all b, b', b'': Book, n: Name, a: Addr |
    add [b, b', n, a] and del [b', b'', n, a]
implies b.addr = b''.addr
}
check delUndoesAdd for 3
```

どんなアドレス帳，名前，
アドレスでも，
同じリンクを追加し，
そのリンクを削除すると，
最初に戻る？

諸々を最大3個生成して網羅的に調べてみる



補足：プログラムでの活用

- プログラムにて様々な条件を記述
  - 例：JML（Java Modeling Language）
    （ユニットテストへのアサーション埋め込みツールや，潜在エラーの警告ツール）
- プログラムに対するモデル検査
  （ソフトウェアモデル検査）
  - 例：Java PathFinder
- 抽象モデルよりも，検証内容が詳細になる
  - ポインタ，非null条件，配列インデックス，・・・
  - コードやその実行環境が複雑なので，検証能力に限界が生じたり，工夫が必要になったりする

目次

- 「形式手法」「形式仕様」とは何なのかを知る
  - どういう問題に注目しているのか？
  - どういう考え方で解決を目指すのか？
  - 具体的にはどういう手法・ツールがあるのか？
  - 活用するために何を身につけ，検討するのか？
- 「形式仕様」と「仕様記述」？

形式手法とは？（再）

- 仕様・設計を「きちんと」書く過程により
  - 曖昧さを解消することになる
  - 不正確さ，不整合も表面化する
  - システムに対する理解が深まる
- 仕様や設計を「きちんと」書いた結果により
  - 誤解なく情報を共有できる
  - 科学的・系統的な分析・検証ができ，特にツールに支援させることができる

厳密な文法・意味論を持つ言語を用い

数理論理学がベース

内容・方法は様々

➡ ***開発早期における信頼性確保***

※分析・検証の仕組みは，プログラムに対して活用することも多い

学ぶこと： 準備

- 基礎知識： 数理論理学の基礎の基礎
  - 集合論：

    メモリなどを想定した操作の仕組みや効率に触れずに，状態やその変化を表現する道具
  - 命題論理・述語論理：

    Yes/No と明確に判断できる言い回しをする道具
    - プログラムのif文などの条件式より少し強力に

（＋ 個々の手法・ツールで必要なもの）

- あとはSPINの場合の「時相論理」くらい

学ぶこと

- 形式手法の考え方
  - ~~様々な手法・ツールの知識~~
    （短期で学ぼうとしてもたぶん何も身につかない）
- ⇨1個の手法・ツールの技術を学ぶことを通し，
  - 形式手法の考え方と十分向き合い，体験し，それに従って，定義や記述，分析・検証の良し悪しの議論や，進め方の検討を行えるようになる
    - 特定手法・ツールを使えることは二の次？
  - 自分たち自身が向き合わなければならない難しさを実感するところまで踏み込む
  - 形式手法の考え方やその手法・ツールの利点と欠点・限界を，実感をもって議論できるようになる

検討・議論すること

- 自分たちが抱える問題点との関連性・マッチングを考え抜く
  - 考え方の必要性，活用方法
  - （マッチするなら）特定の手法・ツールの必要性，活用方法
- 自分たちが抱える問題点と，考え方や手法・ツールとのギャップを考え抜く
  - 何がまだ足りないのか？
  - どれだけ効果的なのか？
  - どうすれば「今の現場」にはまるのか？
  - ・・・（山ほど）

あくまで
解法の基（種）
となる一般論

「演習」だとしても，これらがないと，ほぼ意味がない！

取り組みの例（１）

- 日本語による仕様記述の考え方（清水さんのものなど）とは，どう得意なこと・効果が違い，どう補完するのか？
- テストファーストといった考え方で仕様を向上することとは，どう得意なこと・効果が違い，どう補完するのか？
- WebアプリのようにHow（実装フレームワーク）が早い段階から決まっている場合に，どこまでどうHowを意識した記述を上流で行うべきか？

取り組みの例（２）

- 去年すごくはまった，うちのあの案件で，もしも形式仕様を使っていたらどうなっていただろうか？
- 現状用いているExcelベースのWebアプリ設計書に対し，必要なところだけ厳密な言語を決めて，テスト自動生成・実行を可能にできるか？
- この業界でいつも使っているフォーマットからの言語変換により，皆にできる限り意識させずに，形式手法のツールを使うようにできるか？
- Howのない形式仕様から，どうHowを入れ込んでプログラムを効率よく導けばよいか？

目次

- 「形式手法」「形式仕様」とは何なのかを知る
  - どういう問題に注目しているのか？
  - どういう考え方で解決を目指すのか？
  - 具体的にはどういう手法・ツールがあるのか？
  - 活用するために何を身につけ，検討するのか？
- 「形式仕様」と「仕様記述」？

個人的な思想

- 「形式手法」そのものは，最終的に現場で使わなくてもよいです
  - 「手段」を「目的」に変えないでください
  - 特に「強力な検証」の方は，必要なとき，必要なポイントで，必要な人が使えればよいです
  - もちろん，本当にニーズに合致するならぜひ使ってください
- それより，今日聞いたことを，明日（今日）活きる意識，原則（注意点）にしましょう！
  - 要求仕様や設計の内容の決め方，日本語としての表現の仕方，レビューの仕方，・・・