第1回組込みソフトウェア技術者・管理者向けセミナー

技術者向けセミナー

2002年10月15日

会場：日本規格協会本部ビル６Ｆ

- 本ドキュメントは1st Open SESSAME セミナーで使用したテキストです。
- 本ドキュメントの利用について、以下の点にご留意下さい。

1. このドキュメントは，組込みソフトウェア管理者･技術者育成研究会（SESSAME）が著作権を所有しています。
2. このドキュメントは、企業内の研修･教育に無料でご利用になれます。
   ただし、本ドキュメントの全体または一部を利用した公開セミナー等、営利を目的とした複製，利用をされる場合、および報道を目的とした公開の際にはあらかじめ組込みソフトウェア技術者･管理者育成研究会（SESSAME）の事務局から承諾を受ける必要があります。
3. 本ドキュメントの全体あるいは一部を引用される場合は、営利目的･企業内教育目的を問わず、必ず 組込みソフトウェア技術者･管理者育成研究会（SESSAME）が作成したドキュメントであること、および著作権の所在（Copyright (C) 組込みソフトウェア技術者･管理者育成研究会）、およびこのページの末尾にあるクリップアート･写真に関する著作権を明記して下さい。
4. 本ドキュメントを利用したことによって生ずるいかなる損害に関しても、組込みソフトウェア技術者･管理者育成研究会（SESSAME）は一切責任を負わないものとします。
5. 本ドキュメントに関するご意見･ご提言･ご感想･ご質問等がありましたら、組込みソフトウェア技術者･管理者育成研究会（SESSAME）事務局までE-Mailにてご連絡ください。
6. 本ドキュメントは、内容の改善や適正化の目的で予告無く改訂することがあります。

- 組み込みソフトウェア技術者･管理者育成研究会事務局：
  〒113-8656 東京都文京区本郷7-3-1 東京大学大学院 工学系研究科 化学システム工学専攻
  飯塚研究室
  E-mail: sessame@blues.tqm.t.u-tokyo.ac.jp

- 本ドキュメントでは、Microsoft社のClip Art Galleryコンテンツ、およびAdobe社のストックフォト画像のコンテンツを使用しています。それぞれ、Microsoft社、Adobe社が著作権を所有しています。 営利を目的として本ドキュメントをご使用になる場合、各社の著作権条項に抵触する場合がありますのでご注意ください。

SESSAMEの紹介およびコースの概要

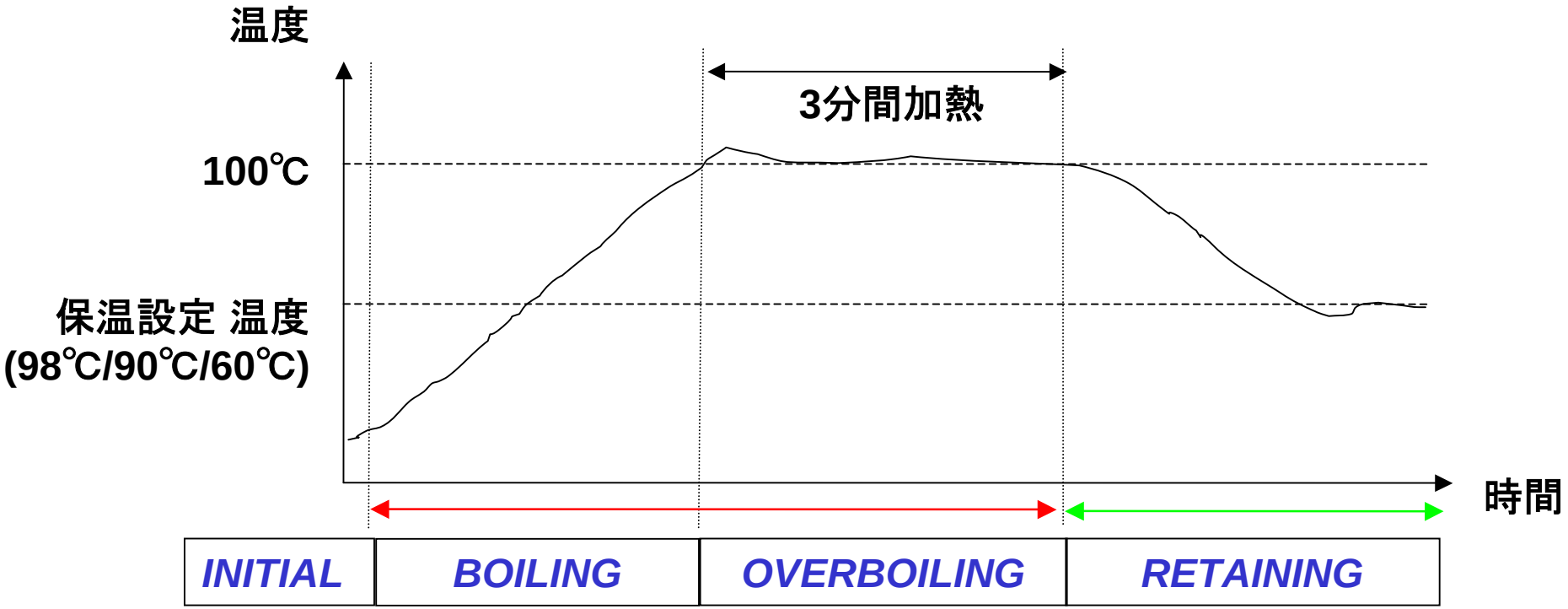
担当：東陽テクニカ　二上貴夫

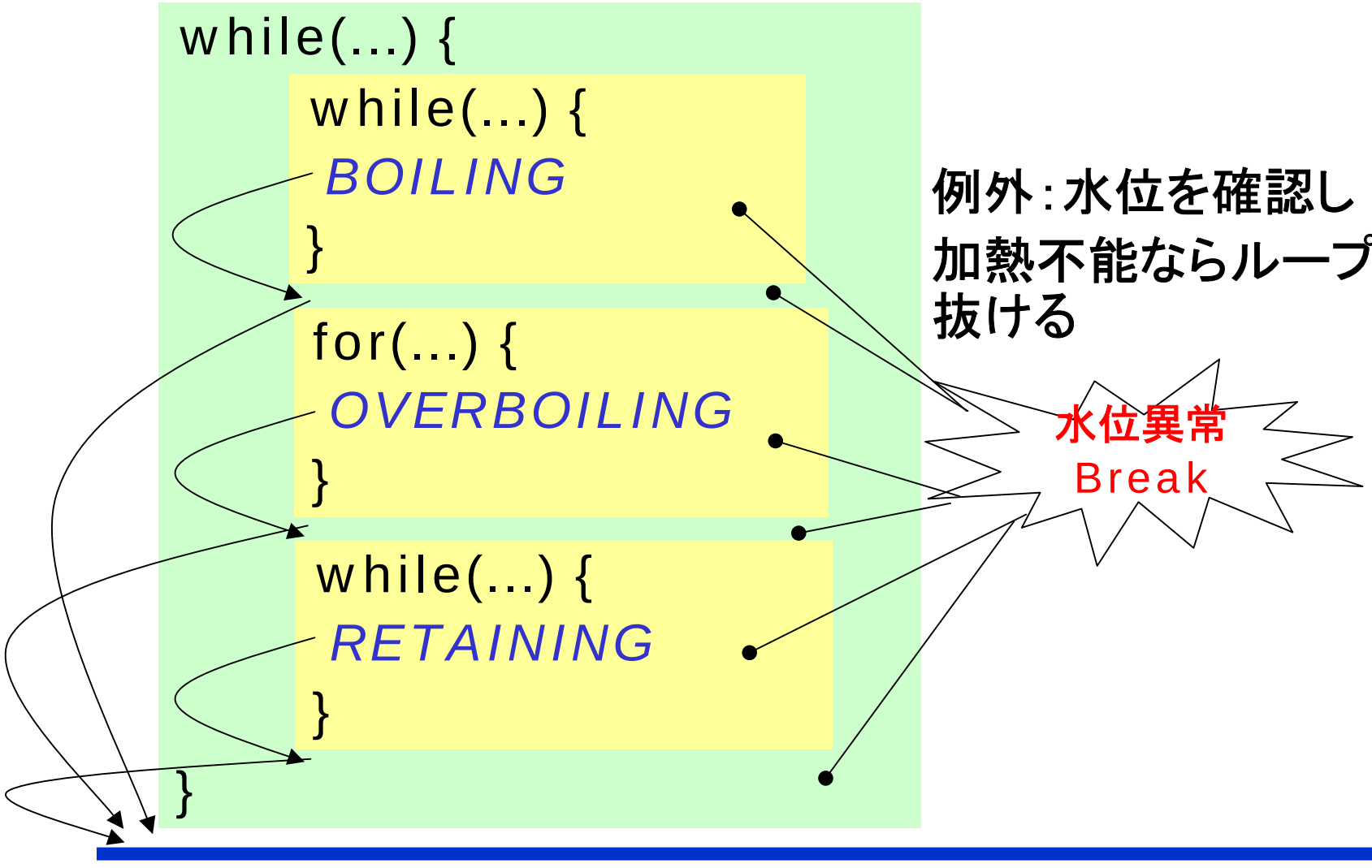
開発課題と失敗事例の解説

担当：富士ゼロックス情報システム　須田　泉

1. SESSAMEの紹介およびコースの概要
2. 開発課題と失敗事例の解説
3. 組込み向け構造化分析の例
4. 組込み向け構造化分析・設計の概要
5. 組込み向け構造化設計
6. プログラミング
7. ソフトウェアテストの概要
8. 話題沸騰ポットに対するテストの実践
9. 大規模開発に向けての注意点

失敗事例1　基本動作でコーディング





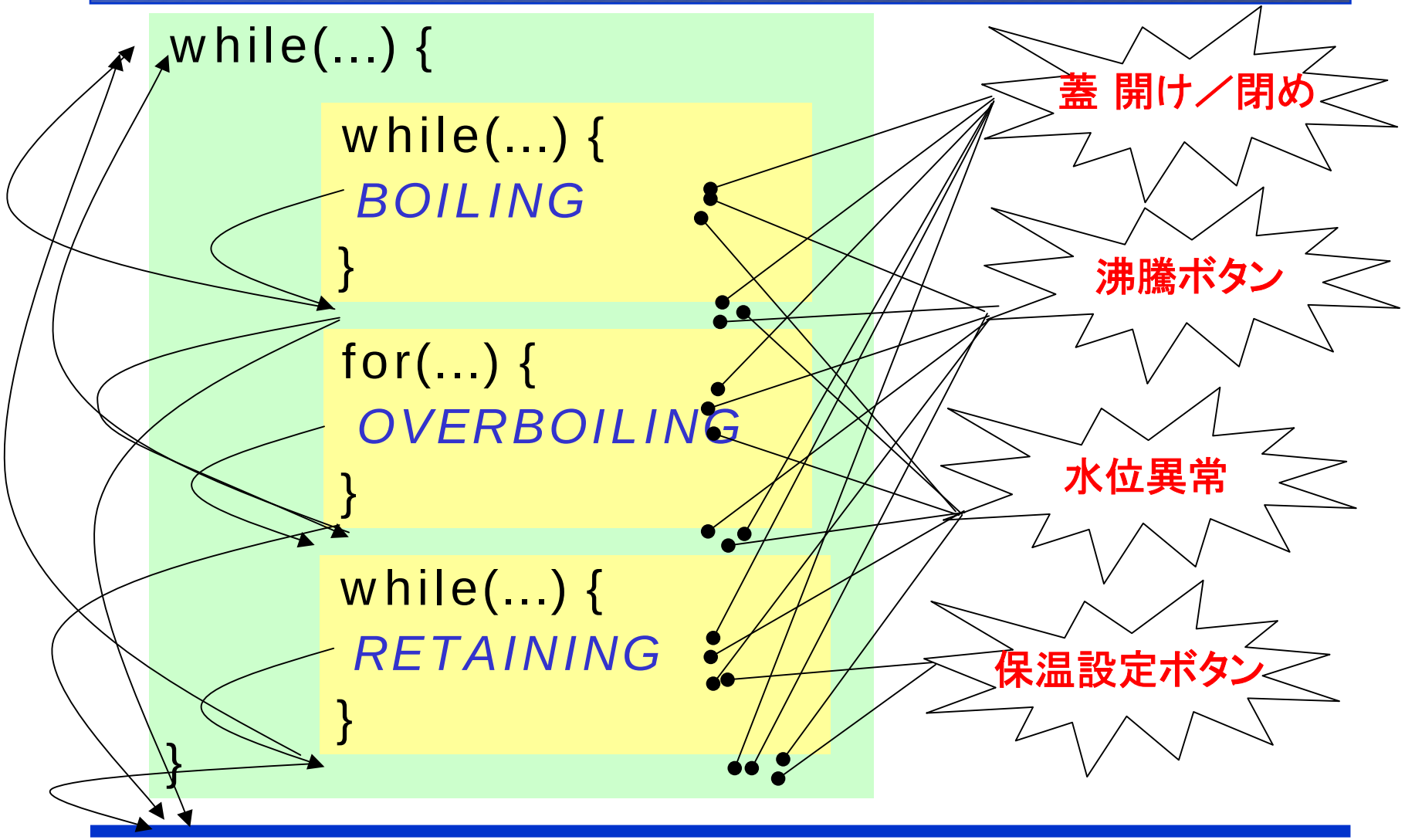
ユーザ操作によるイベントを追加すると

１、蓋の開け閉め→再沸騰

２、沸騰中に沸騰ボタン（沸騰中断）→保温

３、保温中に沸騰ボタン→再沸騰

４、保温中に保温設定変更→目標温度変更



要求仕様の曖昧さに気付かない？

１、ミルクモードで沸騰完了すると100度で保温？

２、沸騰中に沸騰中断したら保温？

過渡期に表示パネルに矛盾が生じる

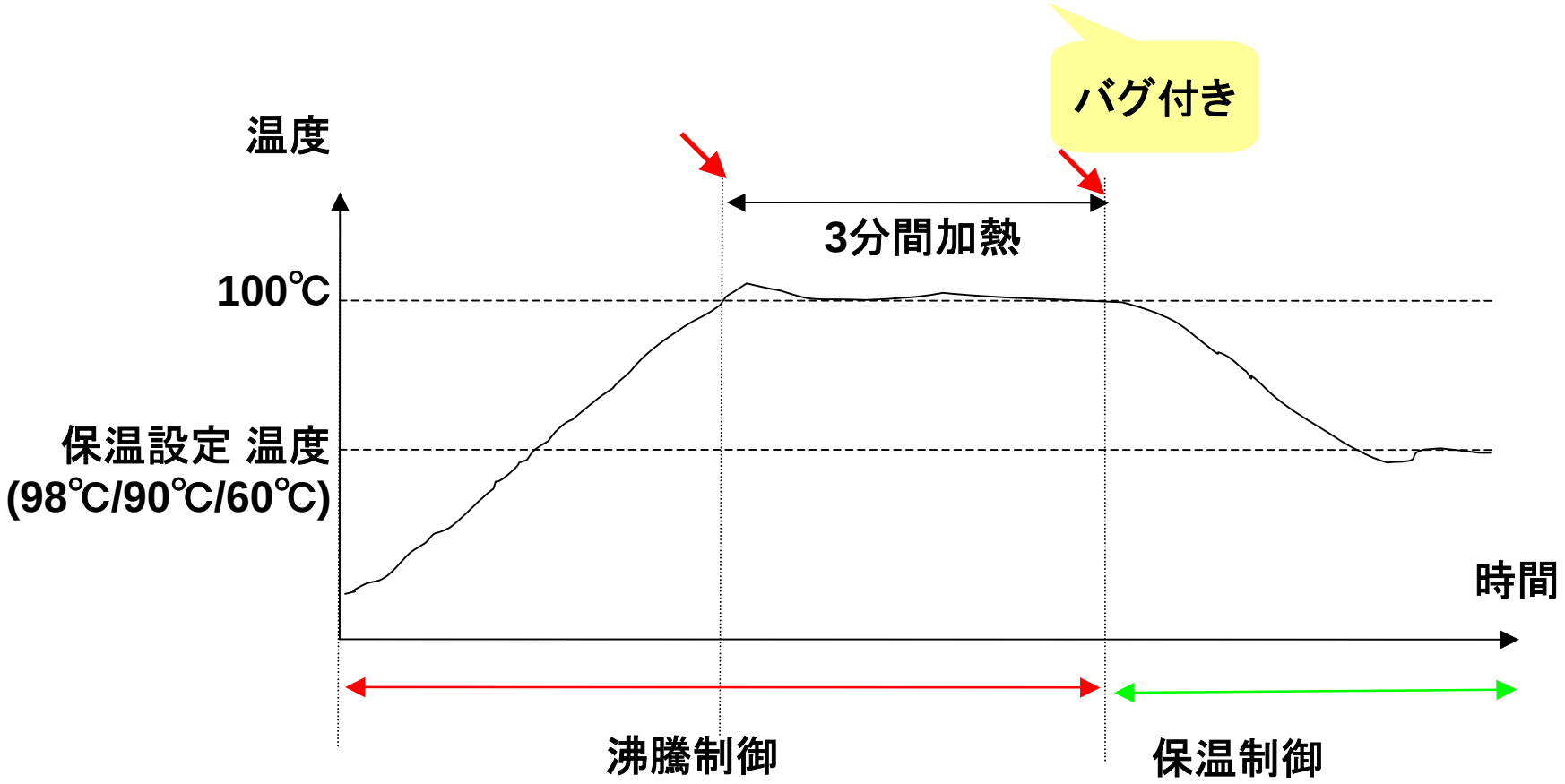
３、蓋が開いた時、沸騰ボタン、沸騰・保温ランプ以外の表示パネルはどうする？

４、蓋が閉まった直後の保温設定は前回を憶えておく？

５、タイマーは何分まで設定できる？

６、タイマーキャンセル方法は？

失敗事例２　5章をさらっと読んでコーディング



```
while(...) {
    if (Portが変化) {
        Eventの解析
    }
    if (沸騰制御) {
        if (3分加熱終了した) {
            保温制御に遷移
        } else if (沸騰100℃に達した) {
            3分タイマースタート
        }
    }
    ヒーター制御（PID制御）
}
```

- 蓋（Open→Close）が変化
- 沸騰ボタンが押された
- 保温設定ボタンが押された

- OnOff制御？

ポットの状態

- bool retain = False　/* 状態を保温制御/沸騰制御*/

　　→状態が追加できない(過渡期、エラー発生)

仕様の誤解

- 沸騰ボタンがトグル制御になっていない
- 温度制御方式に漏れがある(沸騰時はOnOff制御)
- 蓋が開く、水が無くなる、ポットが倒れる

　→ハード的にスイッチが切れると想定している

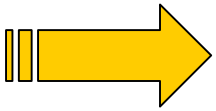
原価削減→ソフトに仕様変更(よくある話)

失敗事例のようなプログラムを作ってしまうと

仕様変更で破綻（作り直し）
保守性が低い
再利用できない

トライ＆エラー
アプローチ

要求仕様曖昧
設計者の理解も曖昧



品質の低下

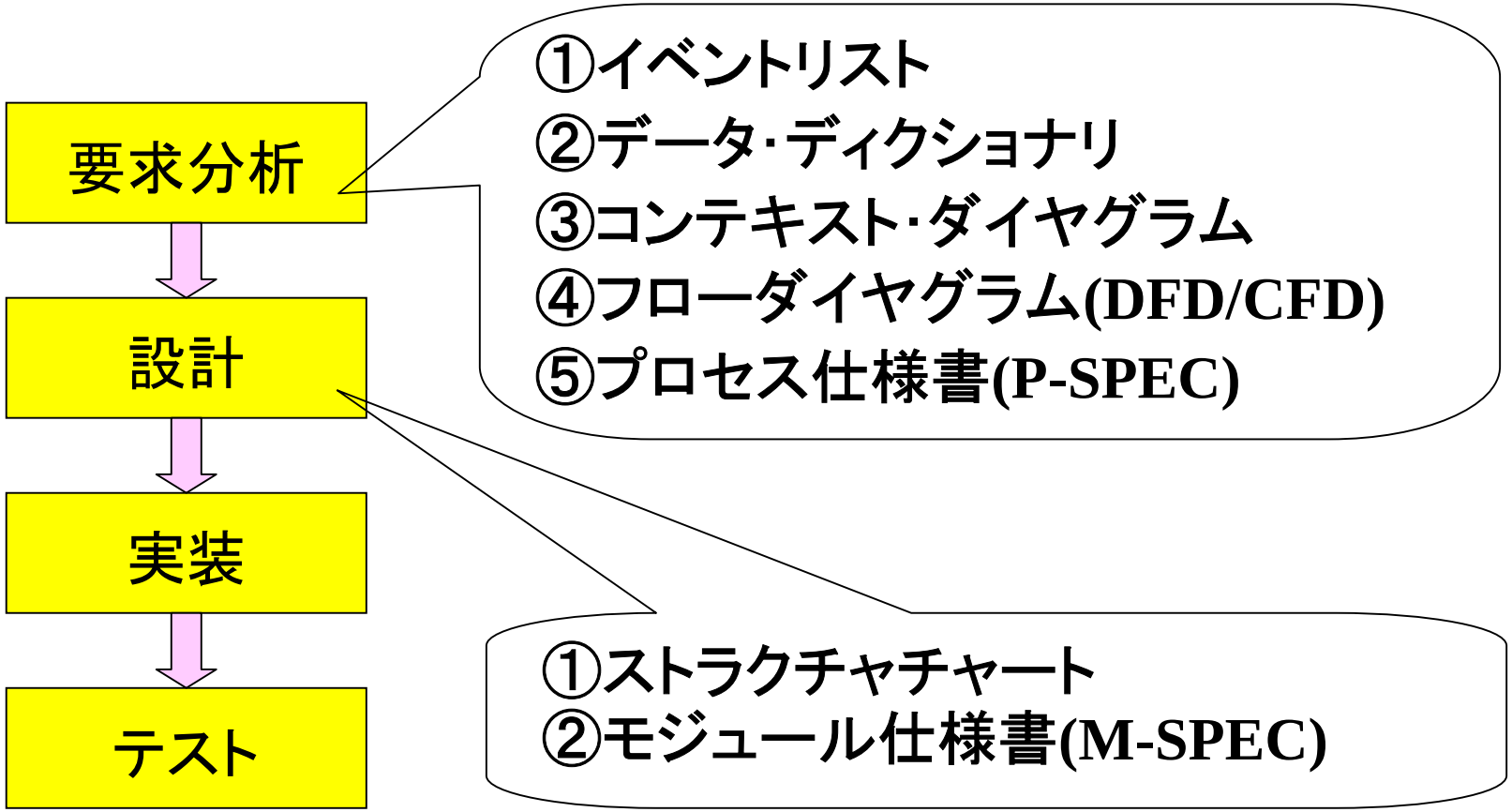
分析していますか？

組込み向け構造化分析の例

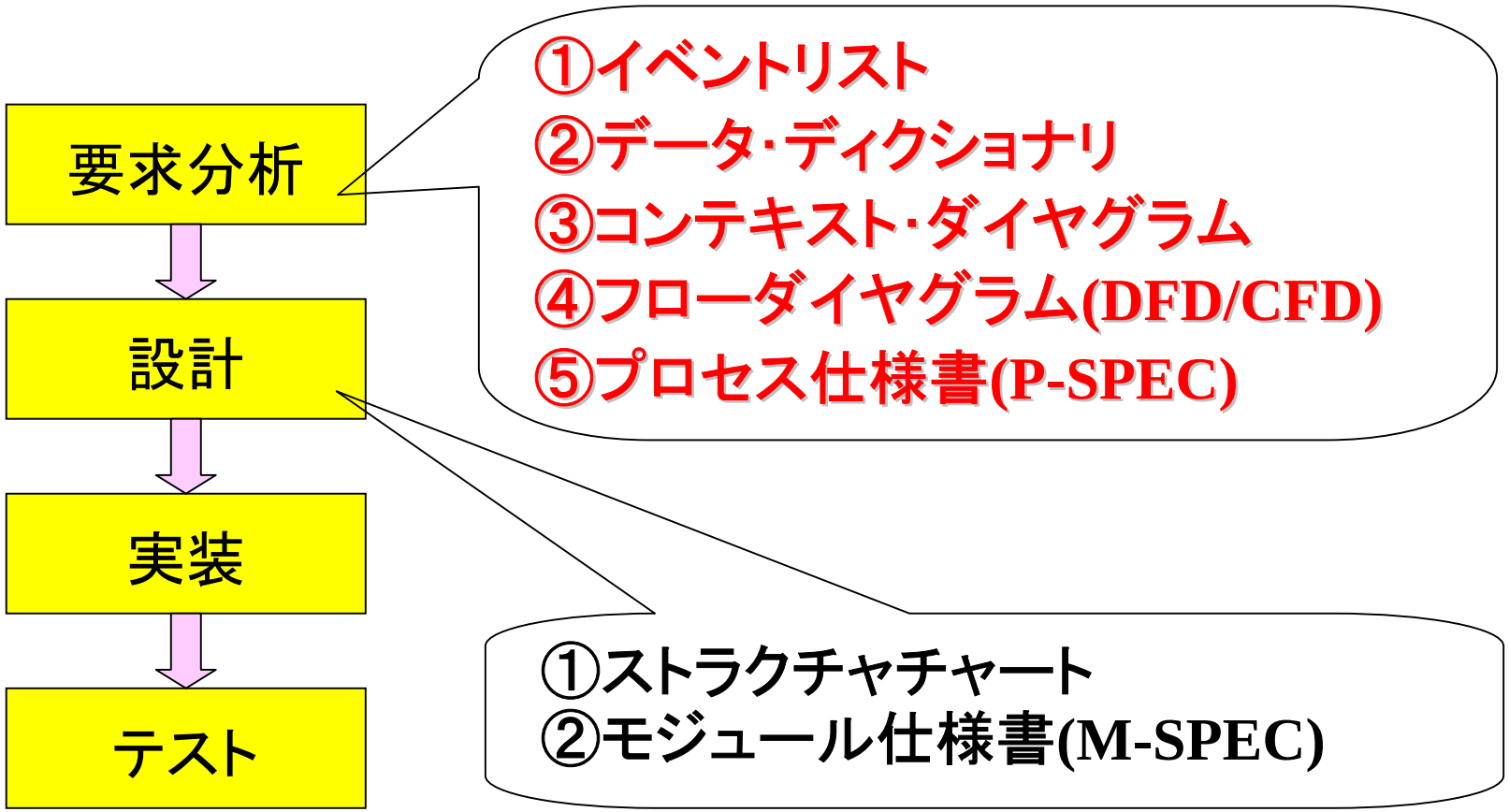
担当：富士写真フイルム株式会社　鈴木圭一

1. SESSAMEの紹介およびコースの概要
2. 開発課題と失敗事例の解説
3. 組込み向け構造化分析の例
4. 組込み向け構造化分析・設計の概要
5. 組込み向け構造化設計
6. プログラミング
7. ソフトウェアテストの概要
8. 話題沸騰ポットに対するテストの実践
9. 大規模開発に向けての注意点

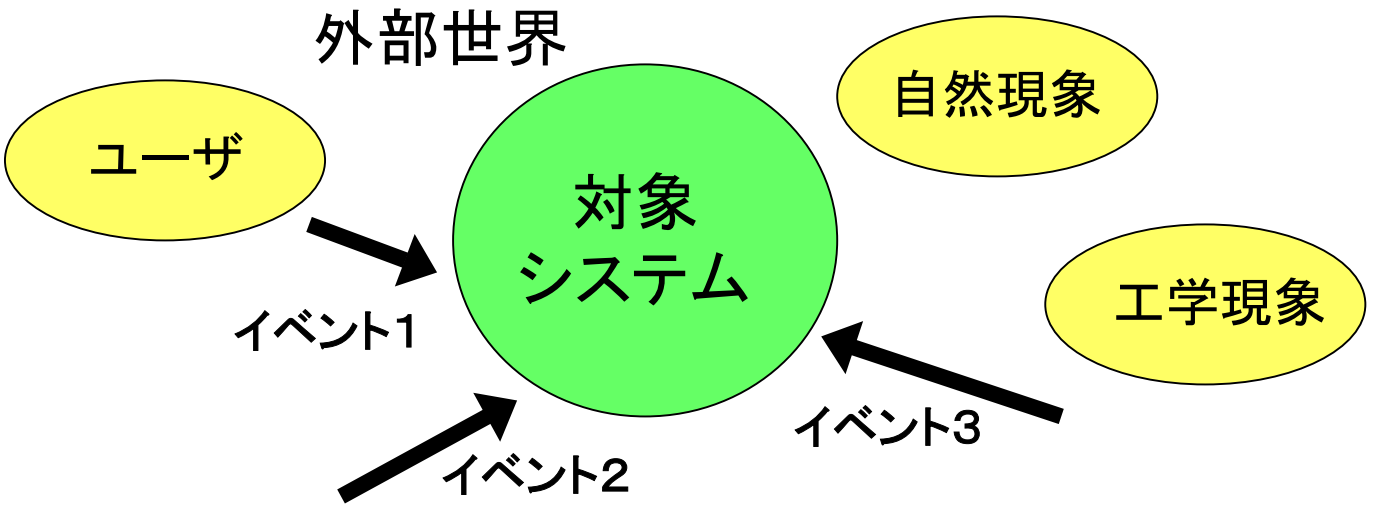
はじめに　－構造化分析/設計の成果物－



アジェンダ



①イベントリスト（１）



イベントリスト

イベント	スティミュラス	アクション	レスポンス
イベント1			
イベント2			
イベント3			
： ：			

①イベントリスト（２）

- ***イベント***
  対象システムの外で生じ、その発生をシステムが制御できない事象。
- ***スティミュラス***
  イベントが発生したことを伝える情報入力。
- ***アクション***
  イベントが発生した時の対象システムの果たすべき機能。
- ***レスポンス***
  イベントが発生した時の対象システムの外部への反応。

＜イベントリスト初版＞

イベント	スティミュラス	アクション	レスポンス
タイマボタン押下	・現在の状態(タイマが押されているかいないか)	・タイマ起動 ・タイマ１分追加 ・ブザーを３回鳴らす	・タイマ残り時間表示
保温設定ボタン	・現在の保温設定状態と押された回数	・保温モード(高温、節約)	・保温モード表示
ボタン	・現在のロック状態(ロック / 解除) ・蓋センサーの状態(開 / 閉)	・給	・ロックランプ点灯 / 消灯
ボタン押下	・水位検出		・給湯口より水排出 ・水位メータで水位を表示
給湯ボタン離上		・給湯停止	
沸騰ボタン押下	・現在の状 (沸騰中 / それ以外) ・沸騰終	・沸騰中止→保温	・沸騰ランプ消灯 ・沸騰ランプ点滅 ・ブザーを３回鳴らす
蓋が開けられる			・沸騰ボタン操作不可 ・沸騰ランプ消灯
蓋が閉じられる		・温度制御可能な水位ならば沸騰状態に移行 ・沸騰終了後、保温状態に移行	・沸騰ランプ点滅 ・沸騰ランプ消灯

実装に関わる具体的な名前(how)は使わない

ユーザにわかる記述内容で表現する

イベントの発生を伝えるデータを記述する

＜イベントリスト改訂版＞

イベント	スティミュラス	アクション	レスポンス
タイマ開始	タイマ　時間	(1)タイマを起動する (2)指定時間を経過したらユーザーに知らせる	(1)・タイマ残り時間を表示する ・操作受付を通知する (2)・タイムアップを警告する ・タイマ残り時間を表示する
タイマ時間追加	タイマ　時間	・現在の残り時間に指定された時間を追加する	(1)・タイマ残り時間を表示する ・操作受付を通知する (2)・タイムアップを警告する ・タイマ残り時間を表示する
保温モード指示	保温モード	・保温モードを設定する ・温度制御を変更する	・ 操作受付を通知する ・温度 / モードを表示する
給湯口のロック	（なし）	・給湯口をロックする	・ 操作受付を通知する ・ロック中を　表示 する
給湯口のロック解除	（なし）	・給湯口のロックを解除する	・ 操作受付を通知する ・ロック解除を　表示 する
給湯開始	（なし）	・ポット内の水を給湯する	・ 操作受付を通知する ・給湯口から水を排出する
給湯終了	（なし）	・ポット内の水の給湯を停止する	・給湯口からの水の排出を停止する
沸騰開始	（なし）	(1)水を沸騰させる (2)沸騰が終了したらユーザーに知らせる	(1)・沸騰中を　表示 する ・保温解除を　表示 する (2)・沸騰終了を警告する ・沸騰解除を　表示 する ・保温中を　表示 する
沸騰中断	（なし）	・水を保温状態にする	・沸騰解除を　表示 する ・保温中を　表示 する
水位の変化	水位	・水温を保温温度にする	・水位表示を更新する
満水	（なし）	・温度制御を停止する	・沸騰解除を　表示 する ・保温解除を　表示 する
水なし	（なし）	・温度制御を停止する	・沸騰解除を　表示 する ・保温解除を表示　する
蓋を閉じる	（なし）	・温度制御可能な水位ならば沸騰を開始する	・沸騰　中を 表示する ・保温　解除を　表示 する
蓋を開ける	（なし）	・温度制御を停止する	・沸騰解除を　表示 する ・保温解除を表示　する
温度異常	（なし）	・温度制御を停止する ・ブザーを鳴らし警告する	・異常を通知する

②データ・ディクショナリ

- 構造化分析で用いる各種図表の中で使用するデータの名前と定義を一定の順序で列記した一覧表。（随時作成）

記号	意味	説明
=	～から成り立っている ～に等しい	左辺の構成要素を右辺に示す
+	～に加えて ～と～	構成要素を並列に並べる（その順序に意味はない）
{ }	～の繰り返し ～の反復	{}で囲んだ構成要素を任意の回数で繰り返す。繰り返しの回数（上限や下限）を{ }外に書くこともある。
[|]	～の中から１つを選択する	[ ]内の構成要素から１つを選択
()	オプション	( )で囲んだ構成要素からの任意選択
" "	文字列	引用符で囲んだ文字列が、そのまま構成要素の内容となる

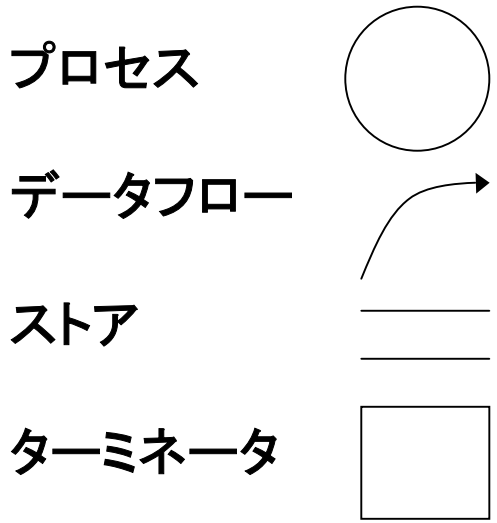
＜データ･ディクショナリ改訂版＞

（イベントリスト改訂版時点）

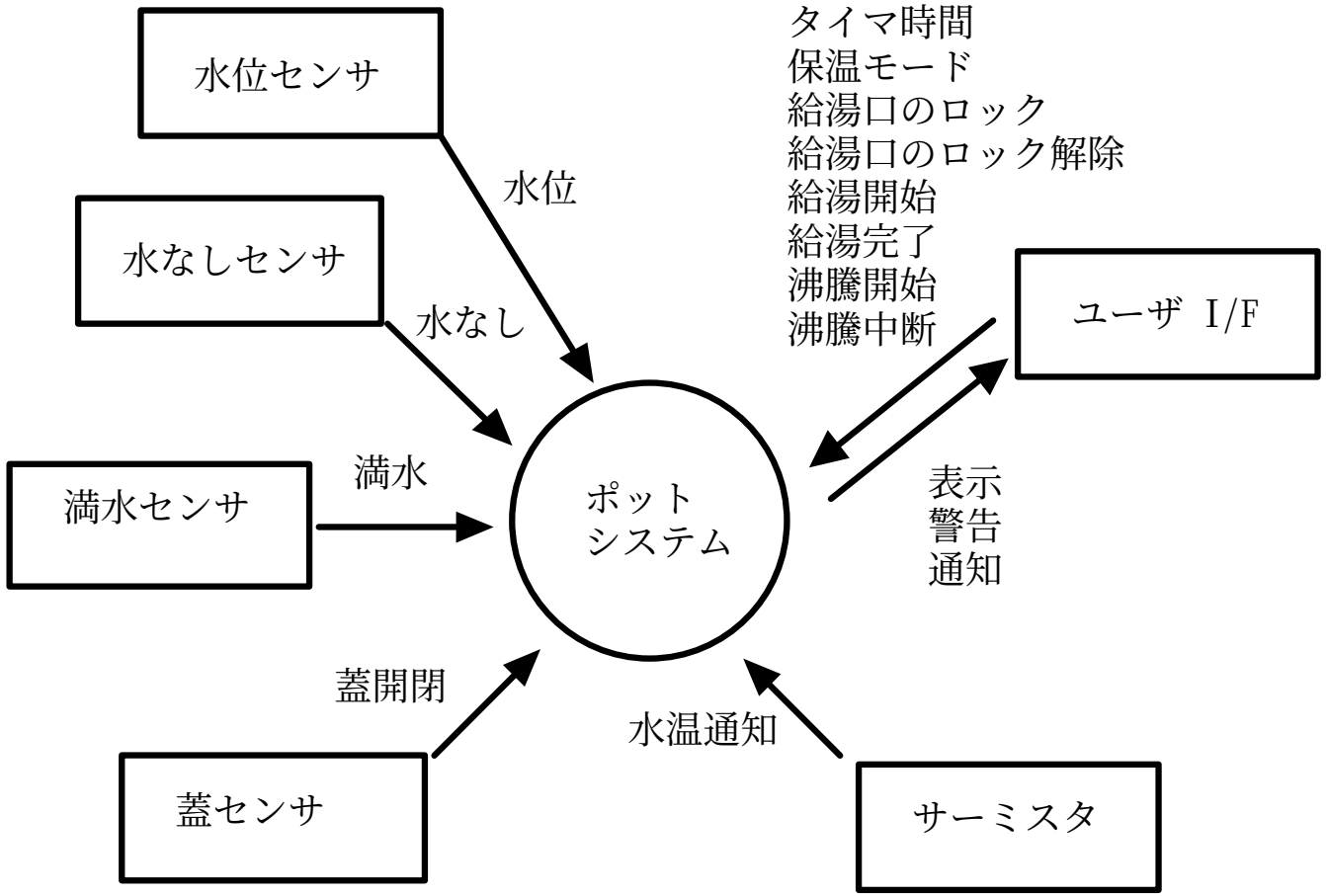
- **タイマ残り時間**＝整数　下限：0 上限：9　単位：分
- **保温モード**＝[高温モード｜節約モード｜ミルクモード]
- **温度異常**＝[高温異常｜温度上がらず異常]
- **水位**＝整数　下限：0 上限：4　単位：なし
- **水温**＝単位：℃
- **高温モード**＝沸騰とほとんど同じように使うための保温モード
- **節約モード**＝消費電力を節約するための保温モード
- **ミルクモード**＝乳児の粉ミルク調乳用として使うためのモード
- **高温異常**＝水温が一定以上に上がった場合
- **温度上がらず異常**＝水温が上がらなくなった場合

③コンテキスト･ダイヤグラム

- システムの機能要求をネットワーク図として表したものをデータ･フロー･ダイアグラム(DFD)と呼ぶ。そのDFDの中で、対象システムのシステムと外部環境との間にあるデータの境界を記述するDFDを特にコンテキスト･ダイアグラムと呼ぶ。つまりコンテキスト･ダイアグラムはDFDの最上位層である。

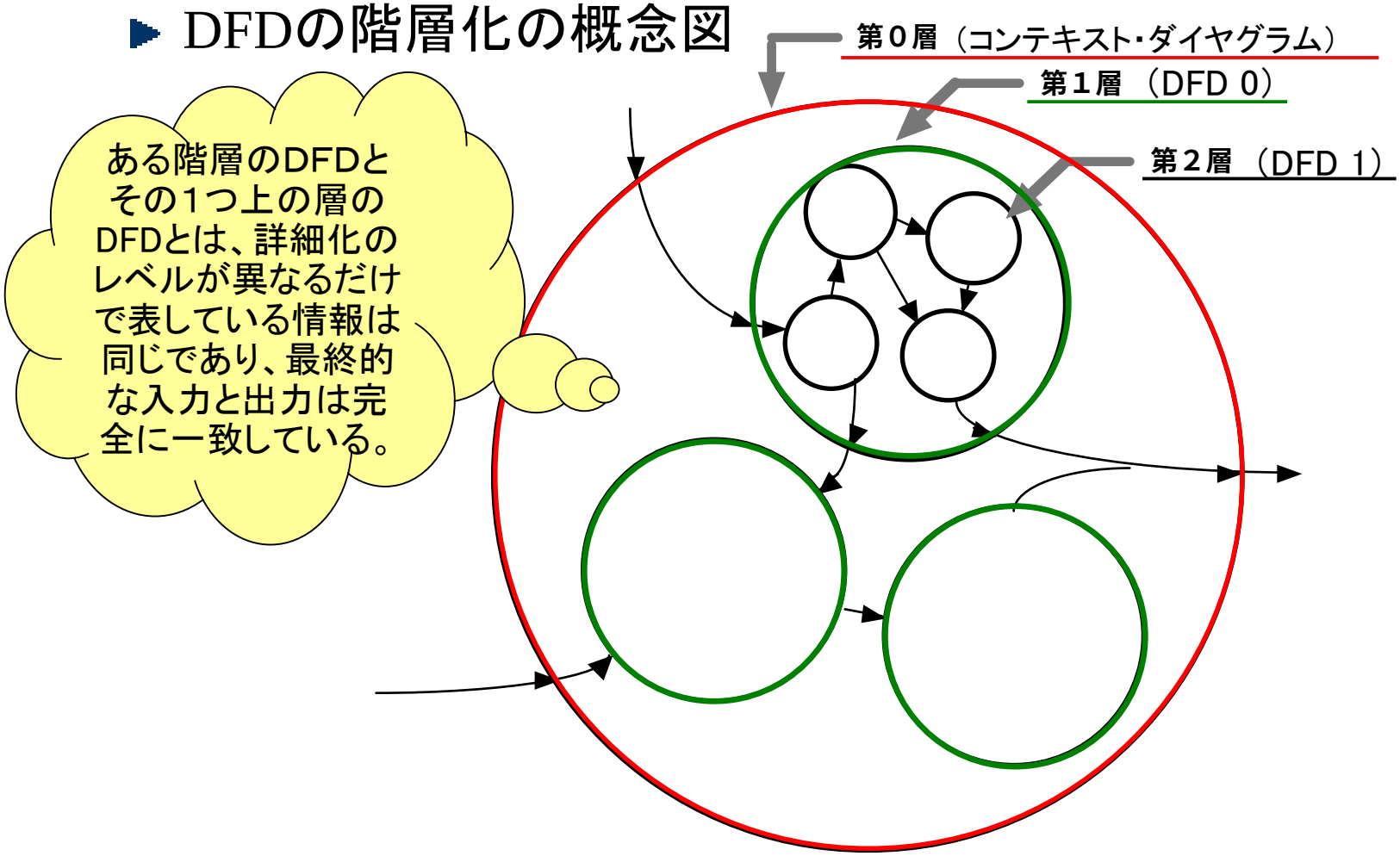


＜コンテキスト･ダイヤグラム改訂版＞

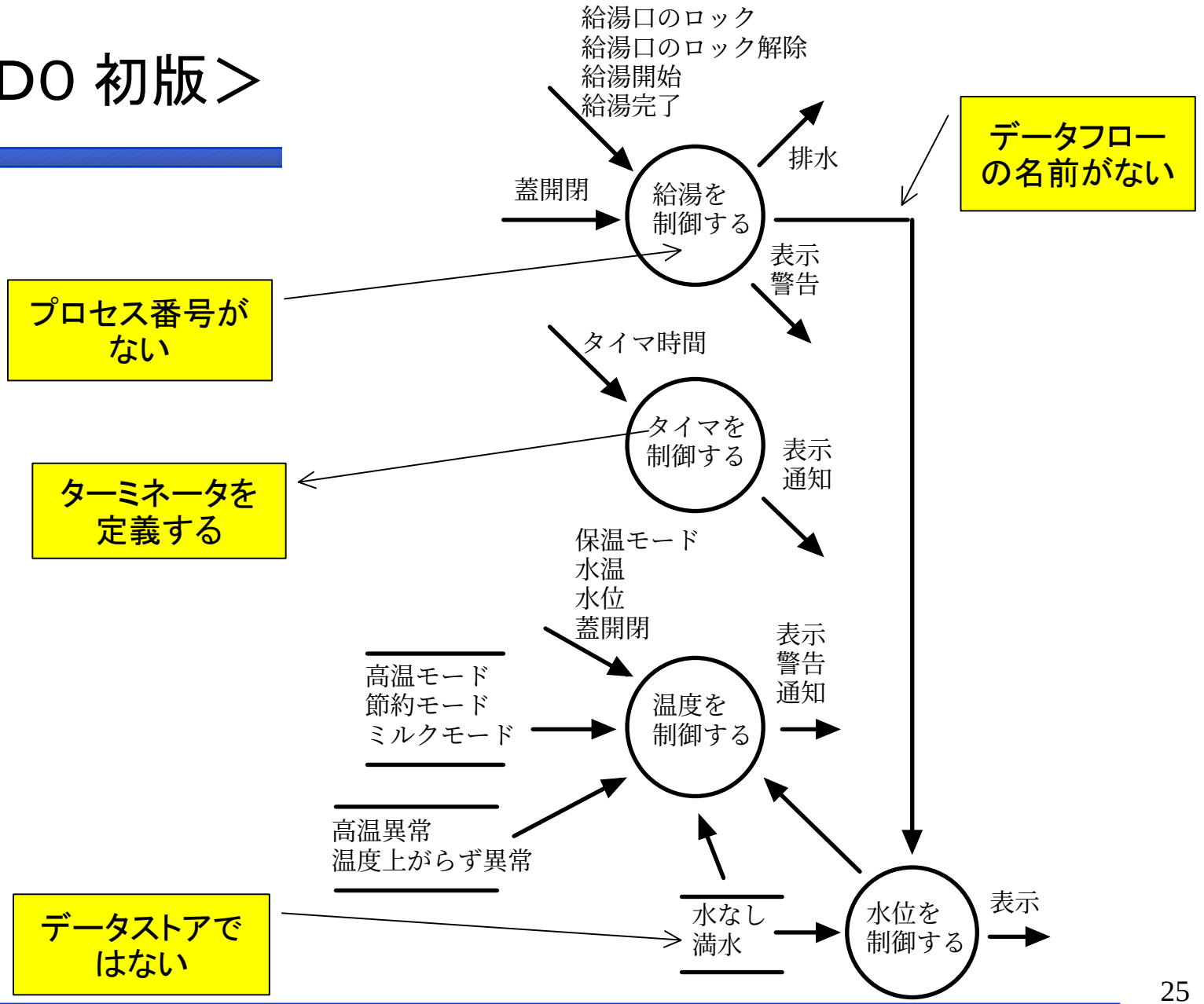


④フロー・ダイヤグラム（DFD/CFD）

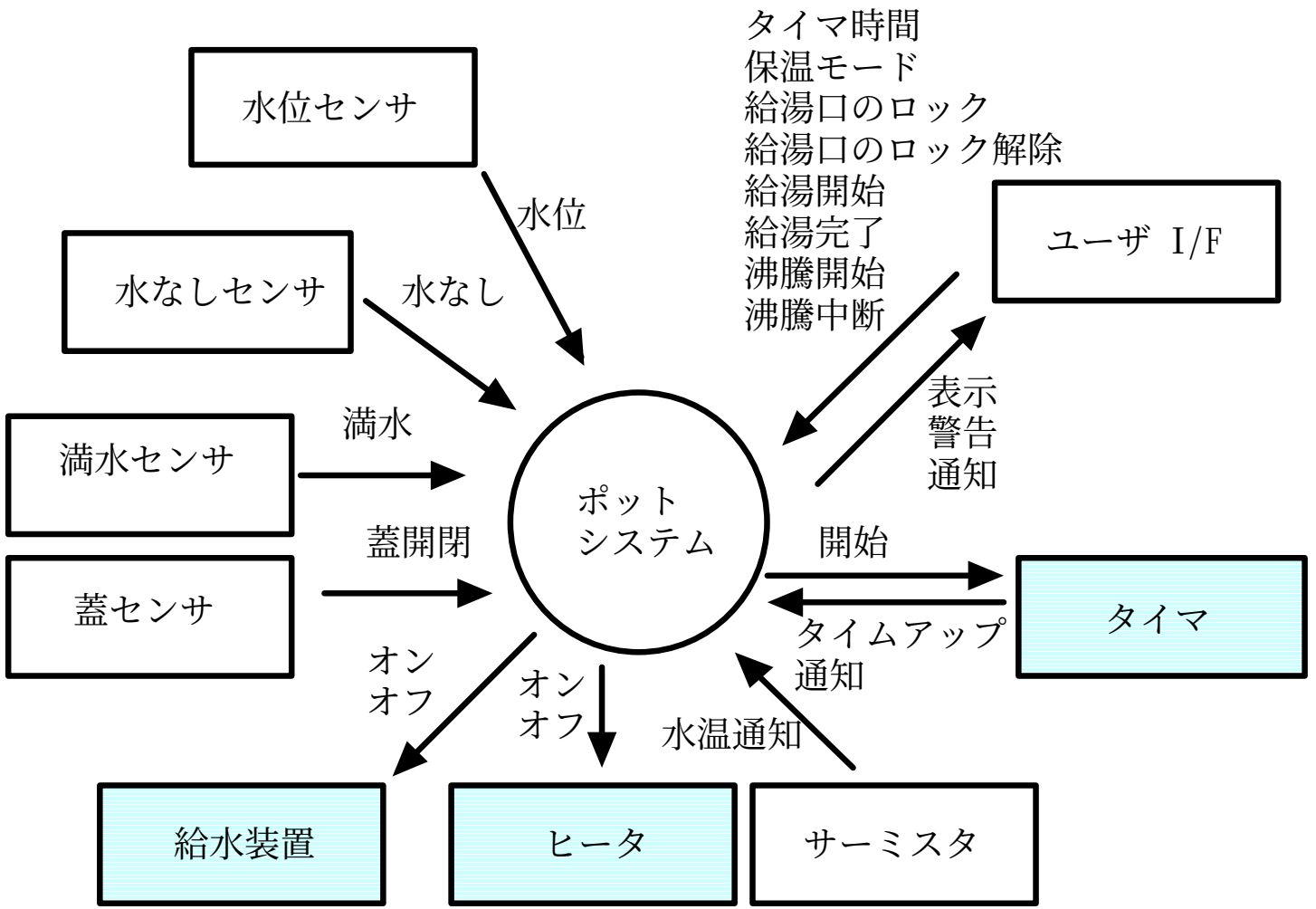
- DFDの階層化の概念図



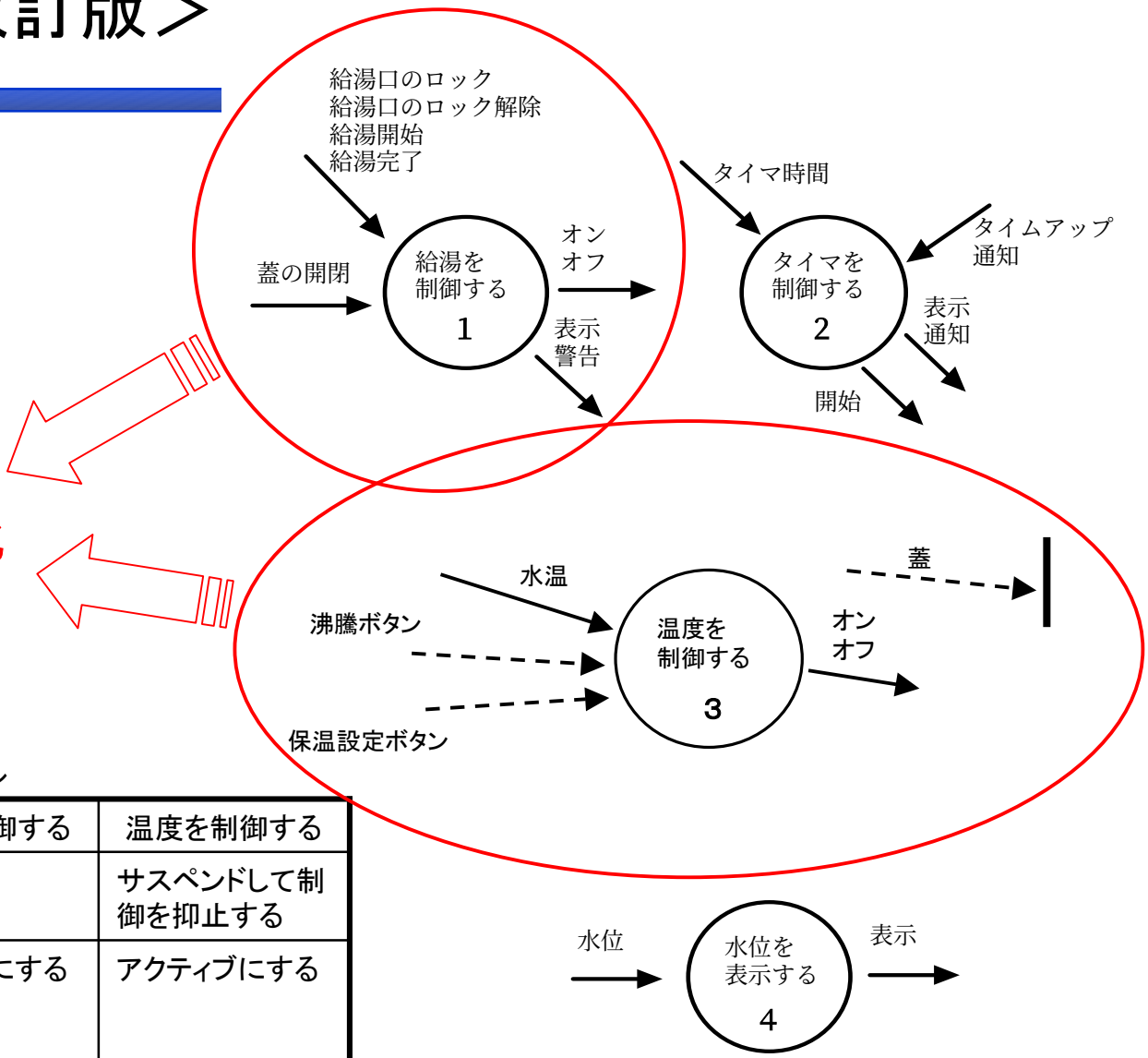
＜DFD0 初版＞



＜コンテキスト･ダイヤグラム再改訂版＞



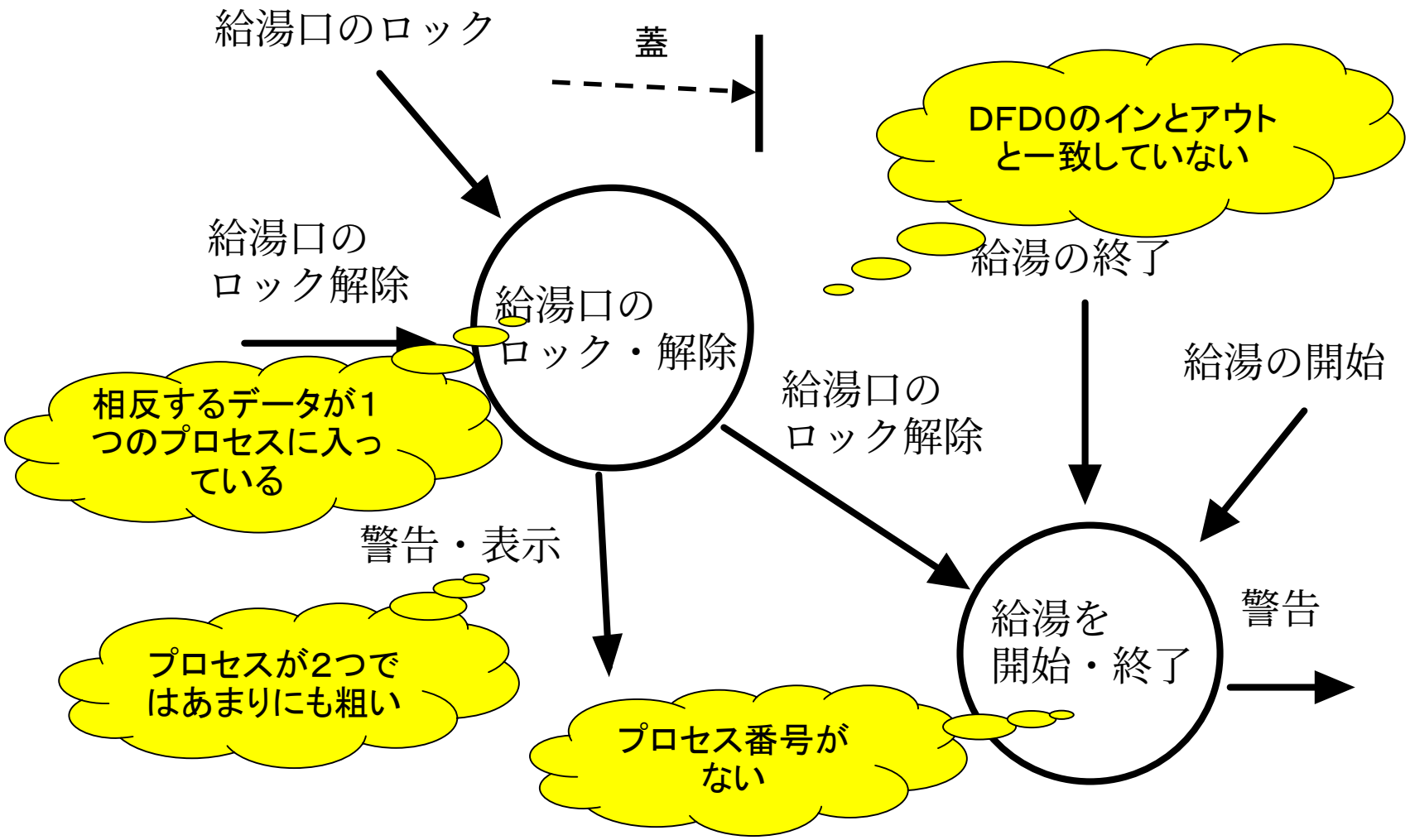
＜DFD0 改訂版＞



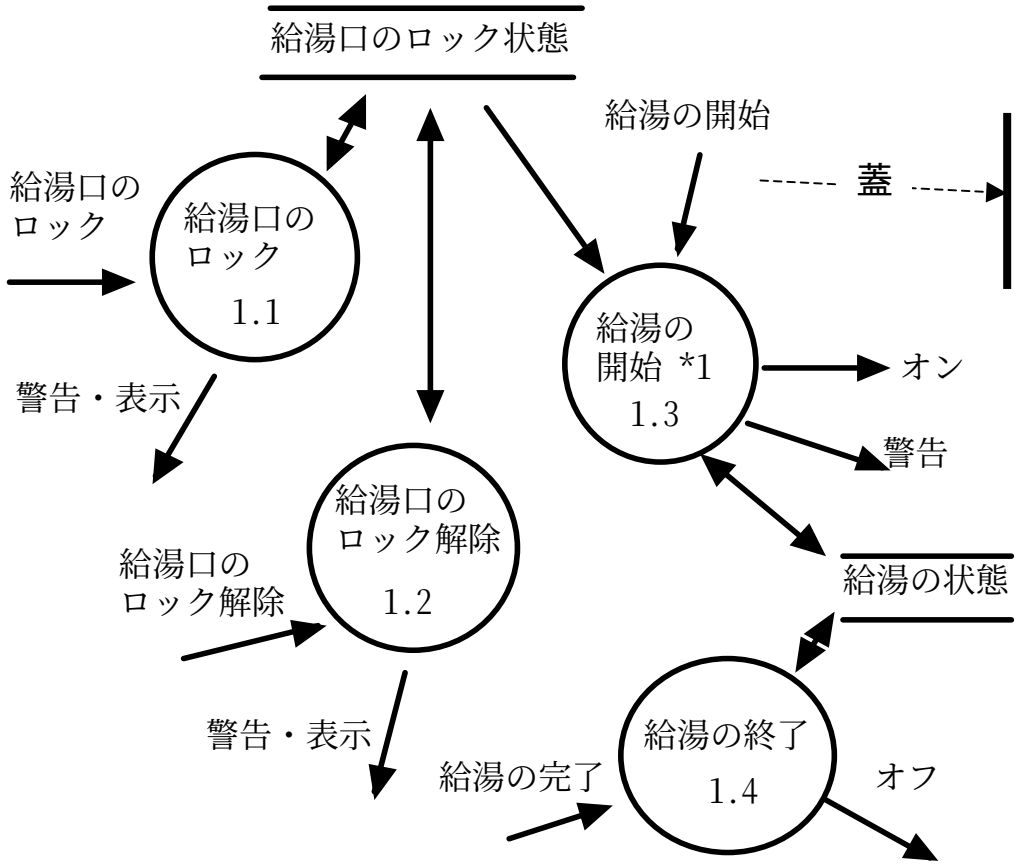
プロセス起動テーブル

蓋	給湯を制御する	温度を制御する
“開” (=off)	停止する	サスペンドして制御を抑止する
“閉” (=on)	アクティブにする	アクティブにする

＜プロセス1のDFD1初版＞



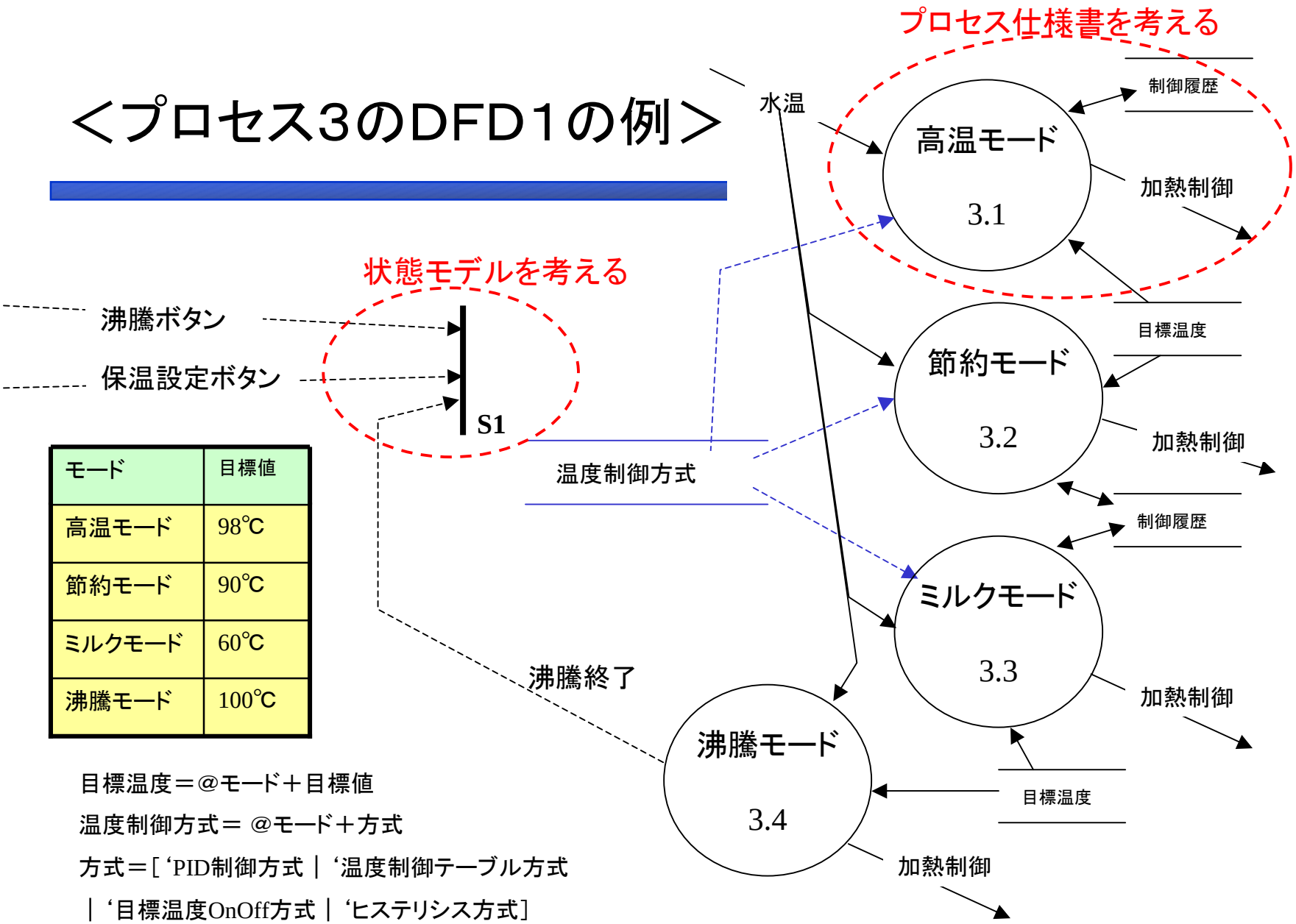
＜プロセス1のDFD1改訂版＞



蓋	給湯の開始
“開” (=off)	停止する
“閉” (=on)	アクティブにする

*1 水位と関係なく実施

＜プロセス3のDFD1の例＞



モード	目標値
高温モード	98℃
節約モード	90℃
ミルクモード	60℃
沸騰モード	100℃

目標温度＝@モード＋目標値

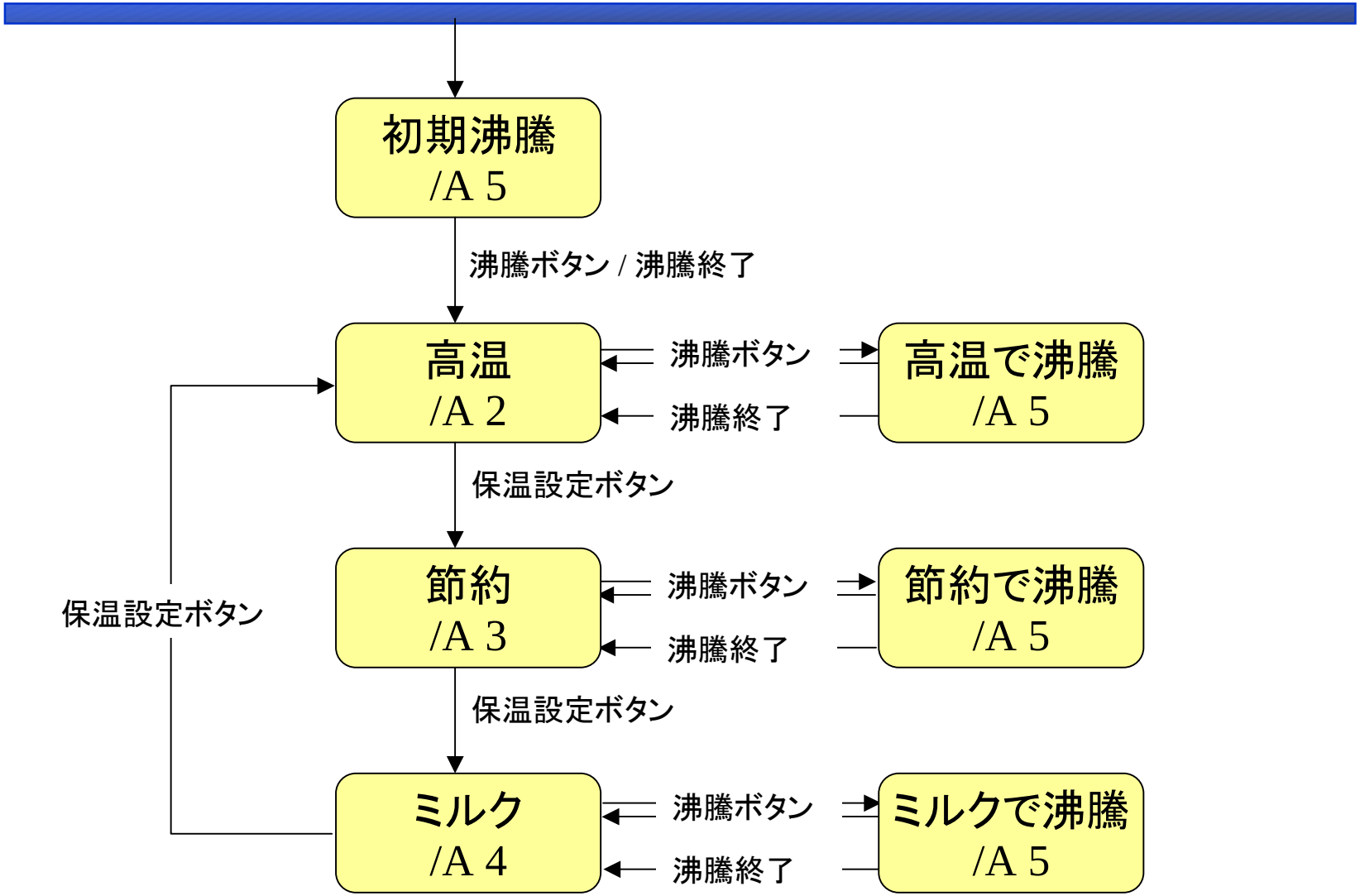
温度制御方式＝ @モード＋方式

方式＝[‘PID制御方式｜‘温度制御テーブル方式

｜‘目標温度OnOff方式｜‘ヒステリシス方式]

制御履歴＝前回の加熱率＋T2+T1+T0

＜S1状態モデルの例＞



⑤プロセス仕様書（P-SPEC）＜プロセス3.1の例＞

高温モードでPID制御する

制御周期ごとに、

1. //水温履歴を更新する

$T_2=T_1$

$T_1 = T_0$

$T_0 =$ 水温

2. //目標温度を得る

Tgを目標温度. モード＝高温モードである
目標温度. 目標値に設定する

3. //PIDでの制御量を計算する

$\Delta M=Kp(T_1-T_0)+Ki(T_g-T_0)+Kd(2T_1-T_0-T_2)$

今回の加熱率＝前回の加熱率＋ΔM　//加熱率が100%以上になることは今回、考慮しない。

4. //ヒータ通電期間を制御する

制御期間×今回の加熱率　の期間中は、

加熱制御=“on”

この期間が終了するタイミングで

//ヒータを切る

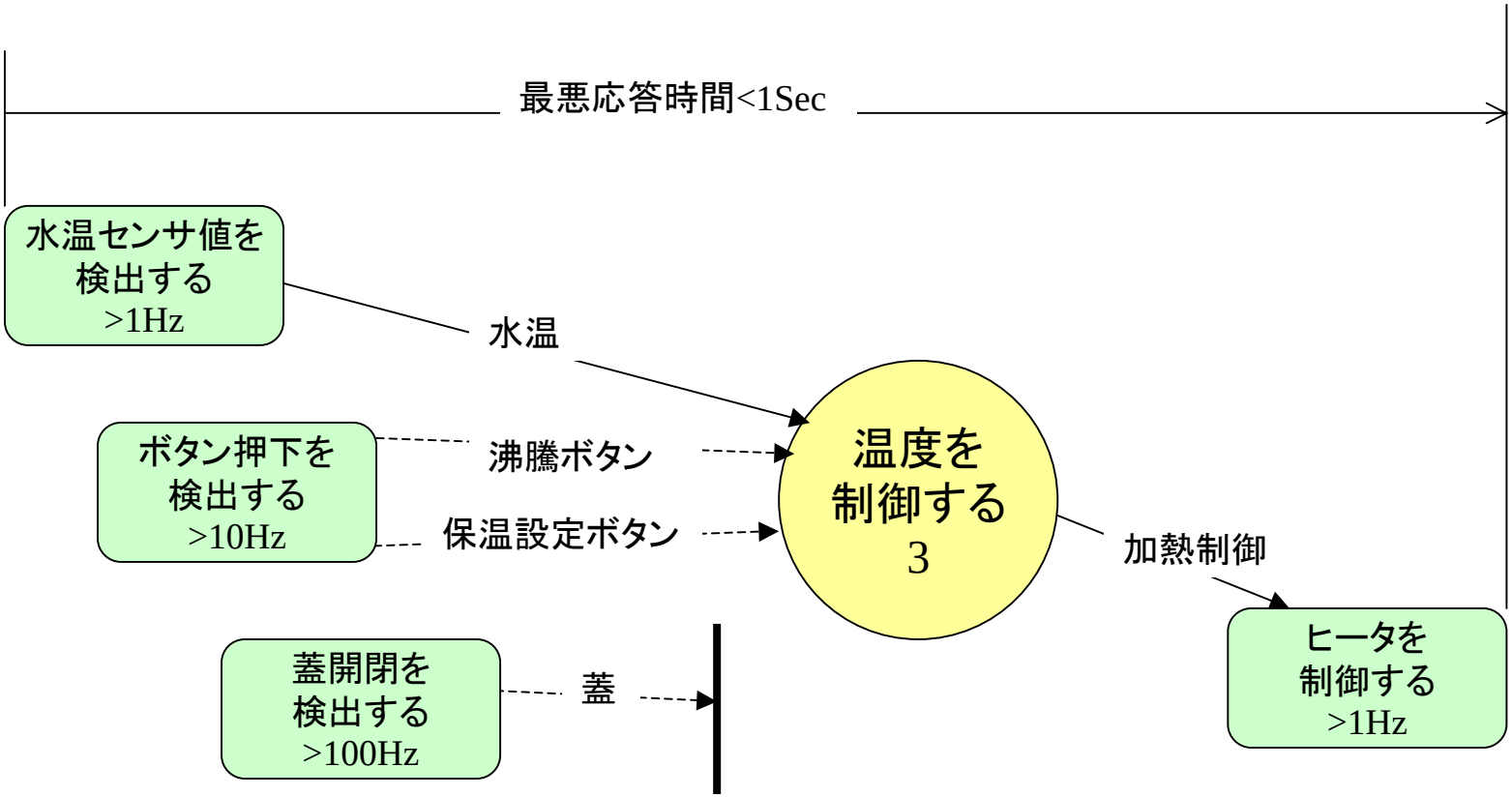
加熱制御=“off”

//加熱率の履歴を更新する

前回の加熱率＝今回の加熱率　//更新

設計の準備として

- プロセス3のDFDのI／O部を深くスケッチする



要求分析の例として、
組込み向け構造化分析手法の分析手順を
簡単に紹介させていただきました。

次の坂本さんの講義
「組込み向け構造化分析と構造化設計の概要」で、
詳しい解説を聞いていただきます。

組込み向け構造分析・設計の概要

担当：ルネサスソリューションズ　坂本　直史

1. SESSAMEの紹介およびコースの概要
2. 開発課題と失敗事例の解説
3. 組込み向け構造化分析の例
4. 組込み向け構造化分析・設計の概要
5. 組込み向け構造化設計
6. プログラミング
7. ソフトウェアテストの概要
8. 話題沸騰ポットに対するテストの実践
9. 大規模開発に向けての注意点

アジェンダ

1．何故、分析を？

2．構造化分析と構造化設計

3．話題沸騰ポットにおける構造化分析

4．構造化設計概要

5．まとめ

アジェンダ

1．何故、分析を？

2．構造化分析と構造化設計

3．話題沸騰ポットにおける構造化分析

4．構造化設計概要

5．まとめ

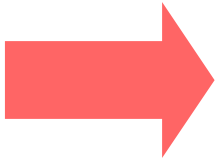
仕様書はもらったけれど・・・・

・もらった仕様書通りに作ったけど？

・そんなことどこにも書いてないのに？

・そんな変更の可能性はちゃんと言っておいて欲しい！

・あの人の仕様書は、・・・・

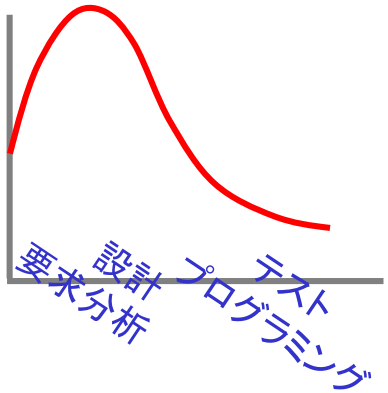


みんなが経験する。でも、
- ・何故そんなことが起こるか？
- ・どうすれば、被害を最小限に？

仕様書を作る立場から

・客先から仕様がでてこない

・仕様を細かく書いている時間がない



・変更はつきもの。それより開発を進めたい。

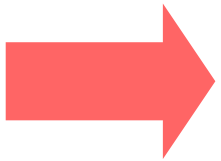
・昔の自分を思い出そう
　いつか通った道のはず！
・後工程に影響が大きいから、経験があり
　給料の高い人がやってるはず！

仕様書を読む立場から

・仕様が少し曖昧でも、わかる範囲でコーディング

・早く動かしたい

・仕様変更はいつものこと。指示があったら考える。



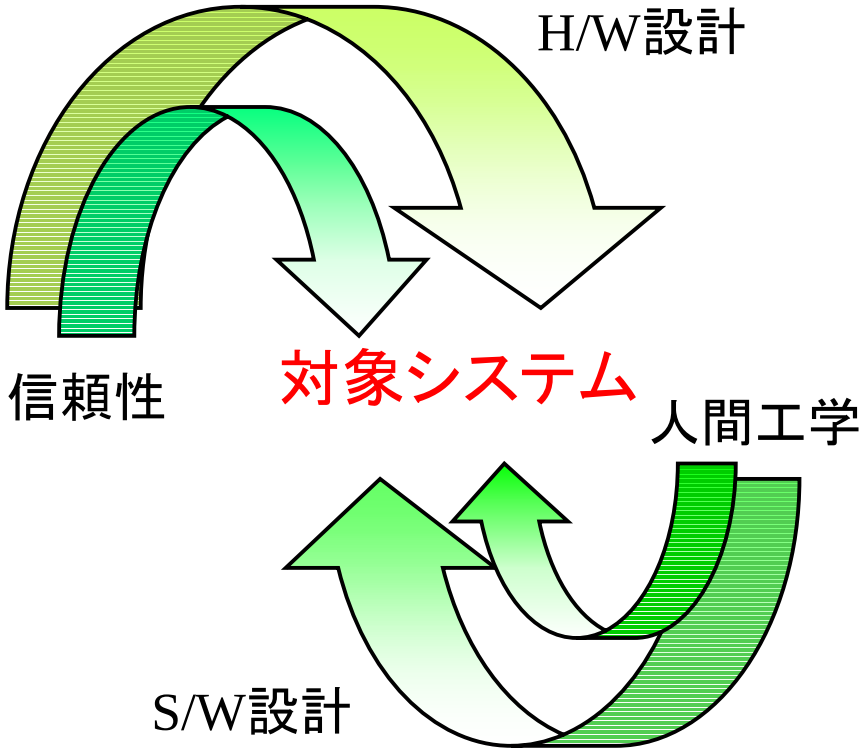
・拡張性や保守性を考えておくことが
自分を守る
・OJTだけでなく新しい技術の取り込むことで
知見を広くし、一段上のエンジニアへ

要求分析

・開発対象システムの本質を考察
・仕様の漏れ、抜け、改善点
　→　目指すシステムと
　　有るべき姿の共有

・何をどのようにする？
　→　“What”を明確に

・いろいろな視点からの
　分析が必要



アジェンダ

1．何故、分析を？

2．構造化分析と構造化設計

3．話題沸騰ポットにおける構造化分析

4．構造化設計概要

5．まとめ

構造化分析と構造化設計

・S/W開発の大規模化 → 品質低下 / 保守

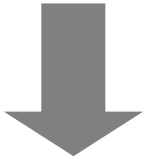
この問題の解決を目指す

設計手法であり、コミュニケーション手段である

構造化分析

・”What”を明確にする

・ユーザとシステム開発者、システム開発者同士のコミュニケーション向上



構造化設計

・構造化分析の結果 → どのように作るか

“What”から”How”の世界へ

・分析で掘り起こしたモジュール間の特性を考慮

保守性の良いプログラム構造

構造化分析

・基本はDeMarcoに始まる構造化分析
　リアルタイム拡張としてHatleyらによる試み

・開発手順

イベントリスト

⇩

コンテキスト・ダイヤグラム

⇩

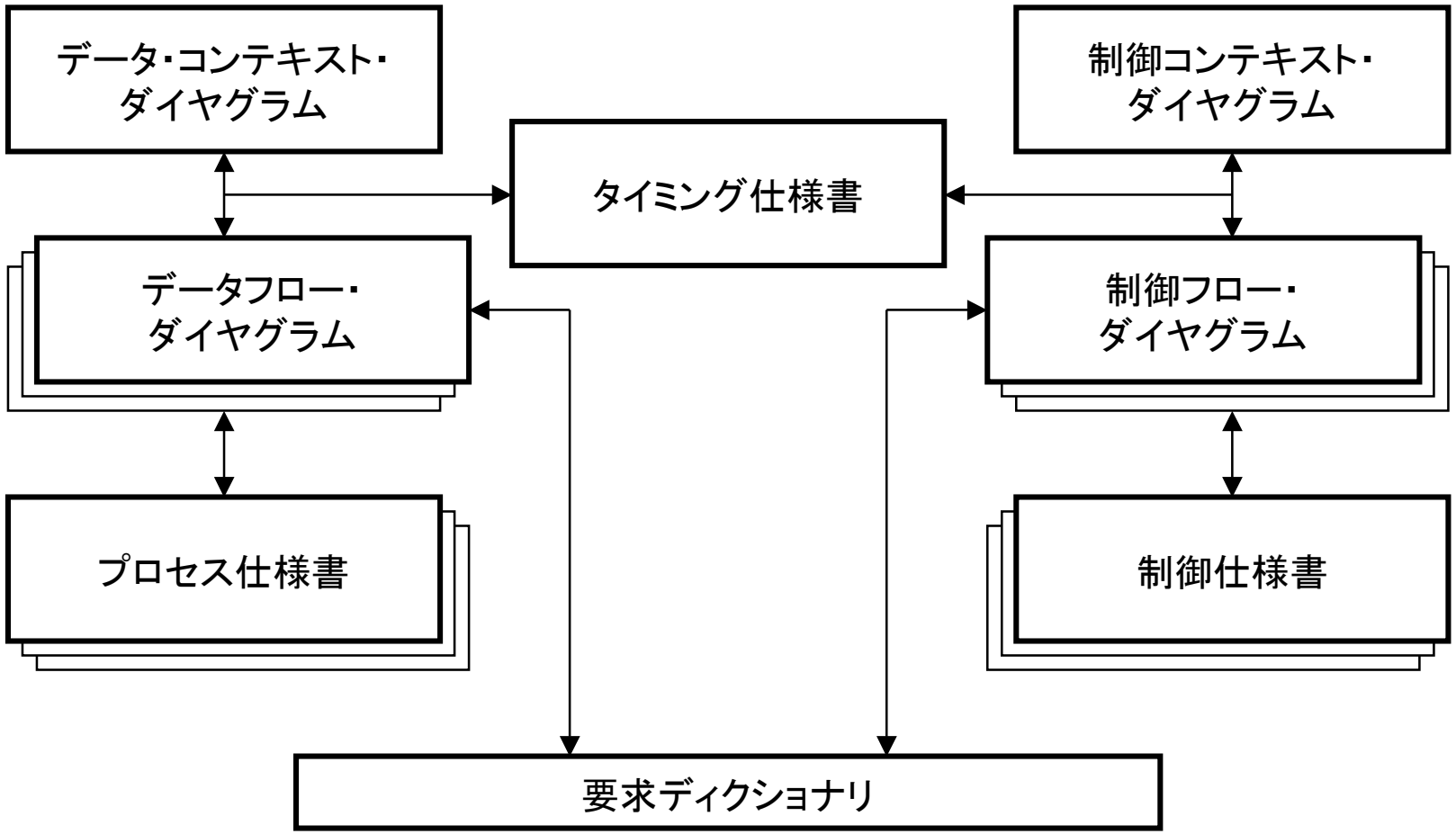
フロー・ダイヤグラム

⇩

プロセス仕様書

データ・ディクショナリ（随時）

Hatley/Pirbhaiによる要求モデルの構成要素



アジェンダ

1．何故、分析を？

2．構造化分析と構造化設計

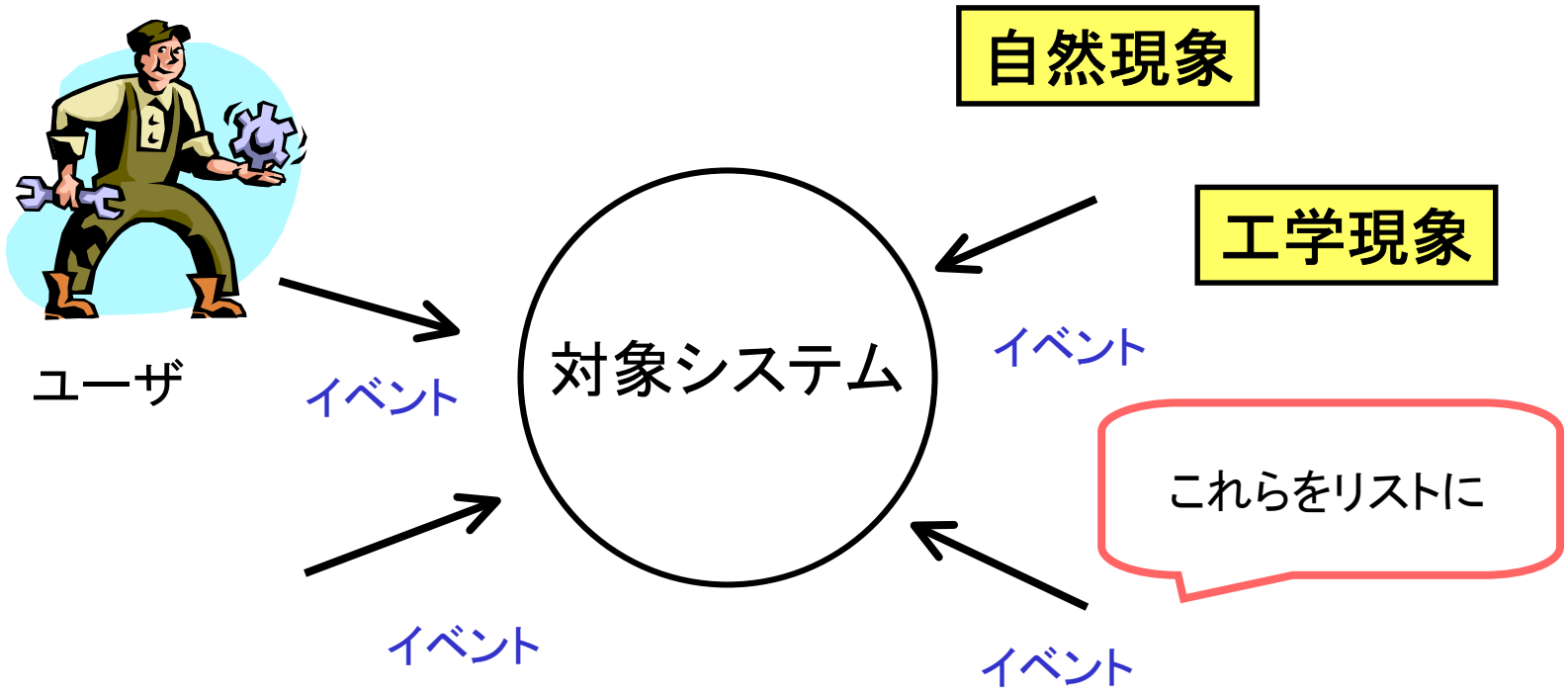
3．話題沸騰ポットにおける構造化分析

4．構造化設計概要

5．まとめ

イベントリスト

・対象システムの外部で発生する様々な事象から
システムで対応しようとするものの一覧

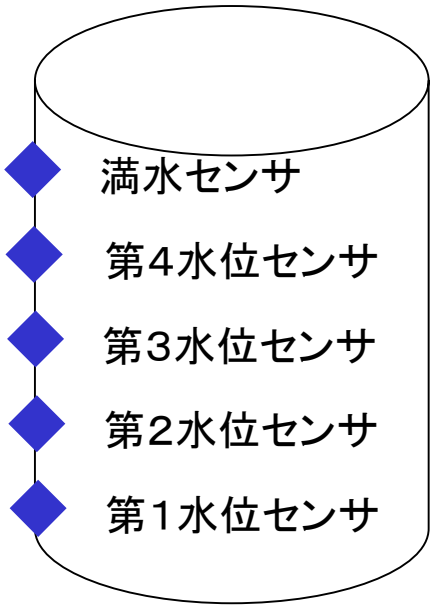


話題沸騰ポットのイベントリスト（１）

・水位の変化をどう捉えるか？

・水位の変化に関係しそうなイベント
　　給湯
　　給水
　　満水の検出

・イベント候補
　　水位センサ毎のオン/オフ
　　給湯や給水
　　　水位の変化は副作用



話題沸騰ポットのイベントリスト（2）

・対象システムの外部で発生する様々な事象から
システムで対応しようとするものの一覧

イベント	スティミュラス	アクション	レスポンス
例） 保温モード 指示	保温モード	・保温モードを 設定する ・温度制御を 変更する	操作受付を通知する 温度/モードを表示する

システムが 発生を制御 できない事象	イベントが 発生したことを 伝える情報入力	イベントが 発生した時に 果たすべき機能	イベントが 発生した時の 外部への反応

話題沸騰ポットのイベントリスト（３）

・水位の変化に関連するイベントを考えてみましょう

第n水位センサ
　水位を検出。各センサはonの時、その位置よりも水位が高い。

アクション？

満水センサ
　水位がこのポットの許容上限を越えていないかどうかを検出。
　センサがonの時、水位が許容上限を越えている。

イベント？

レスポンス？

給湯ボタン
　ボタンを押すと給湯口から水を排出する。押している間中は
　給湯を行う。手を離すと給湯を停止する。

スティミュラス？

水位メータ
　ポット内の水位を表示

話題沸騰ポットのイベントリスト（４）

・お湯を使いたい　　給湯する

給湯口を開く	解除ボタン	給湯可能にする
給湯口を閉じる	解除ボタン	断熱する
湯を注ぐ	給湯ボタン = “on”	湯が出る
湯を注ぐ終える	給湯ボタン = “off”	湯が止まる

・ラーメンを食べたい　　調理時間を計る

３分待とう	タイマボタン	最初の調理時間を設定する
３分では足りない	タイマボタン	調理時間延長
やめた	？？？	キャンセル

人間の「心の状態」から分析する
→　リアクティブ・オートマトン法
　武蔵工業大学　松本先生

話題沸騰ポットのイベントリスト（5）

・例えば、こんな分析結果が出たとします

用語統一

実装にかかわり“How”

ステ	アクション	レスポンス
給湯スタート	給湯ボタン押下 水位検出	給湯スタート	給湯口より水排出 水位メータで水位を表示
給湯終了	給湯ボタン離上	給湯停止	給湯口より水排出停止
給水	水位センサの オン→オフ	水位センサの値により、水位メータの表示を更新する	水位メータの上昇

・どうですか？

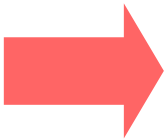
イベント発生を伝える「データ」

外部から見える変化はレスポンス

話題沸騰ポットのイベントリスト（６）

・水位センサ毎のオン/オフを１つのイベントに
　　仕様変更に追従できない
　　粒度が細かく問題の本質が把握しにくい

・給湯や給水をイベントに
　　給水はポットからは検出しにくい
　　→　給水は根本的には水位の変化
　　　満水や水温低下は給水に伴う副作用



「水位変化」をイベントとする
「満水」を「水位変化」と別イベントとするか？
→　システム上重要なことはイベントとして抽出

話題沸騰ポットのイベントリスト（7）

イベント	スティミュラス	アクション	レスポンス
給湯開始	（なし）	ポット内の水を給湯	操作受付を通知する 給湯口から水を排出する
給湯終了	（なし）	ポット内の水の給湯を停止する	給湯口からの水の排出を停止する
水位の変化	水位	水温を保温温度にする	水位表示を更新する
満水	（なし）	温度制御を停止する	沸騰解除を表示する 保温解除を表示する
水なし	（なし）	温度制御を停止する	沸騰解除を表示する 保温解除を表示する

データディクショナリ

・設計で用いる用語を登録　プログラム言語の予約語
　　用語（データ名）は誰にでもわかる言葉を
　　設計を通して用語を統一

・設計の各フェーズでこまめにアップデート

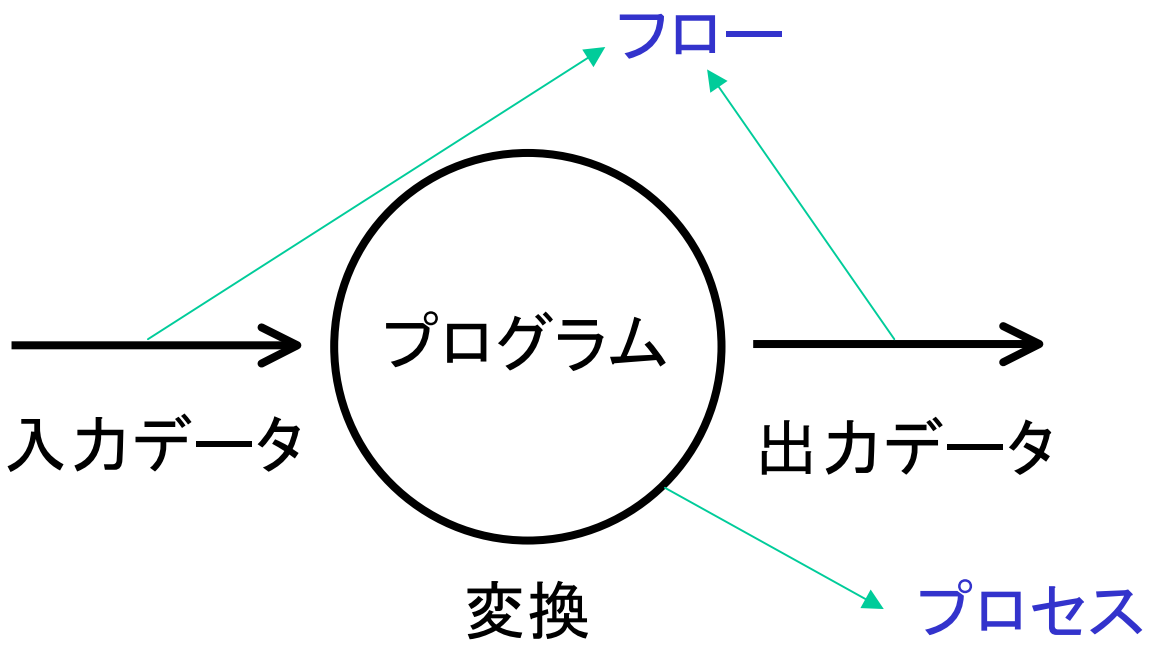
タイマ残り時間＝整数　下限：0　上限：9　単位：分
保温モード　＝[高温モード|節約モード|ミルクモード]
温度異常　＝[高温異常|温度上がらず異常]
水位　＝整数　下限：0　上限：4　単位：なし
高温モード　＝沸騰とほとんど同じように使うための保温モード
節約モード　＝消費電力を節約するための保温モード　ミルクモード　＝乳児の粉ミルク調乳用として使うためのモード
高温異常　＝水温が一定以上に上がった場合
温度上がらず異常　＝水温が上がらなくなった場合

タイミング仕様書

入力	イベント	出力	イベント	対応時間
水温	110度を超える	ブザー音	高温エラー	最大1秒
	前回検出した水温より低い	ブザー音	ヒータ動作異常	1分

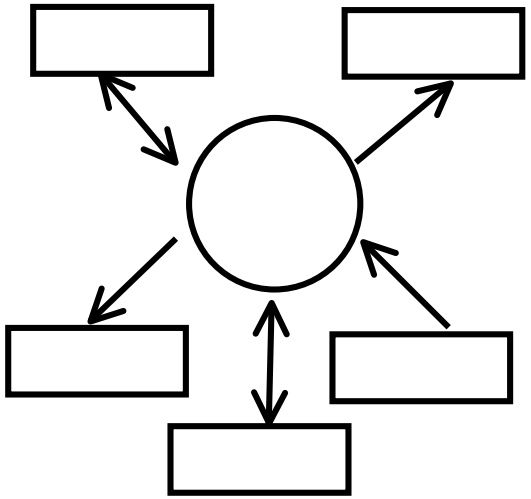
データフロー・ダイヤグラムによる要求分析

・プログラムをデータの変換と捉える

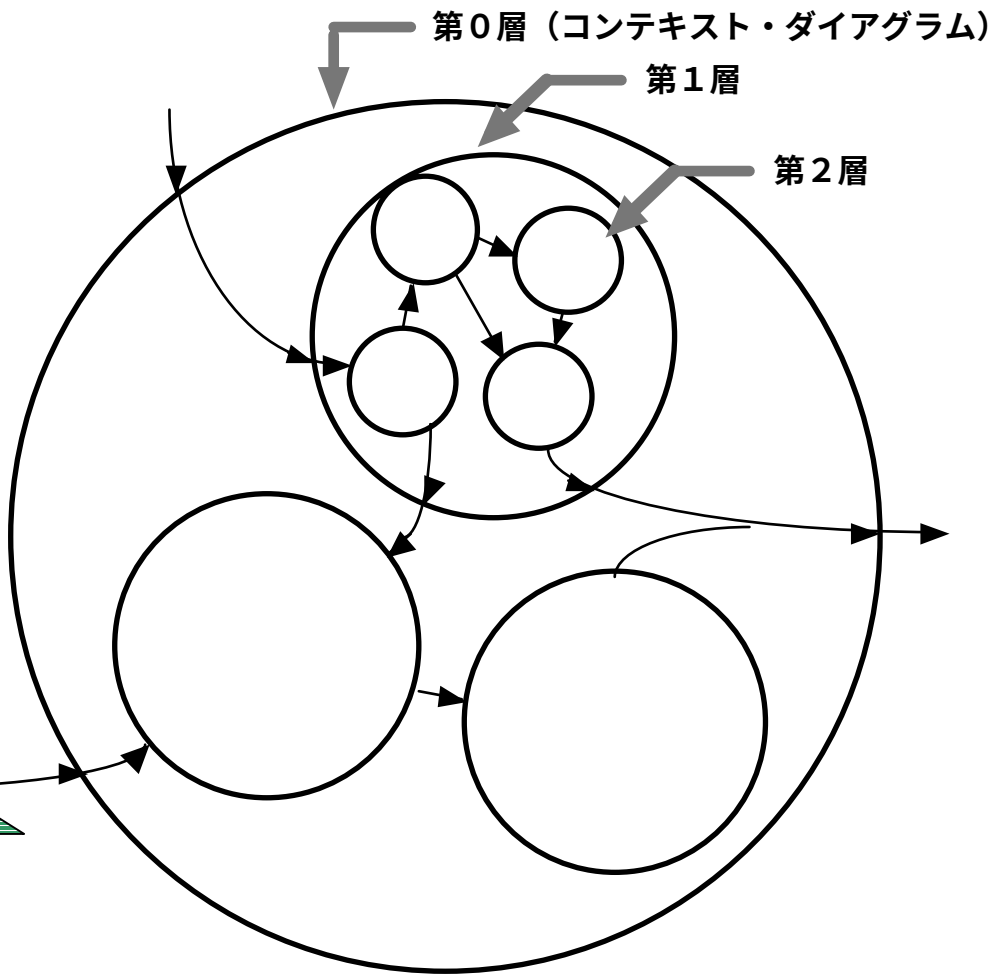


データフロー・ダイヤグラム　＝　DFD

分析の過程



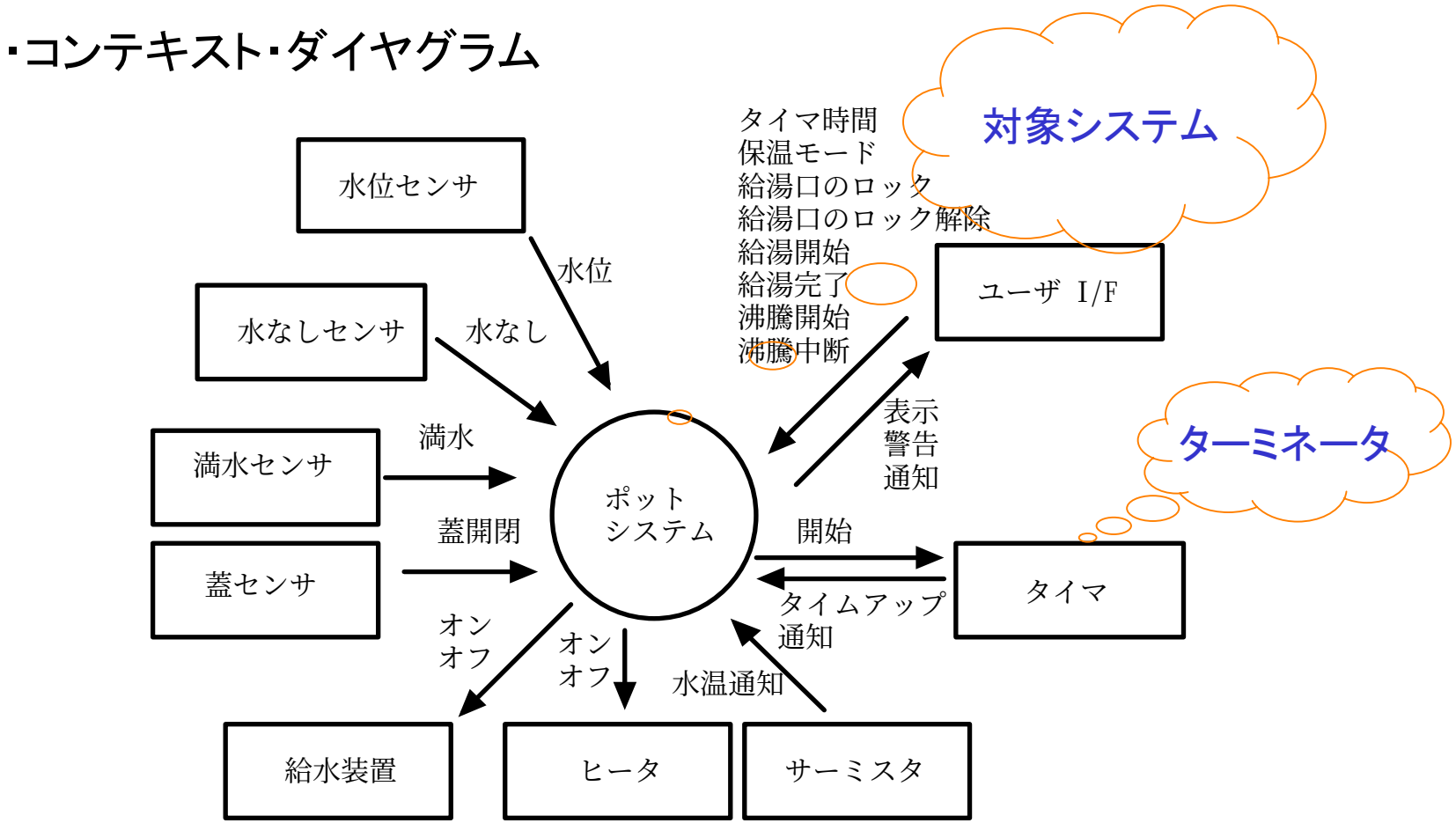
コンテキスト・ダイヤグラム



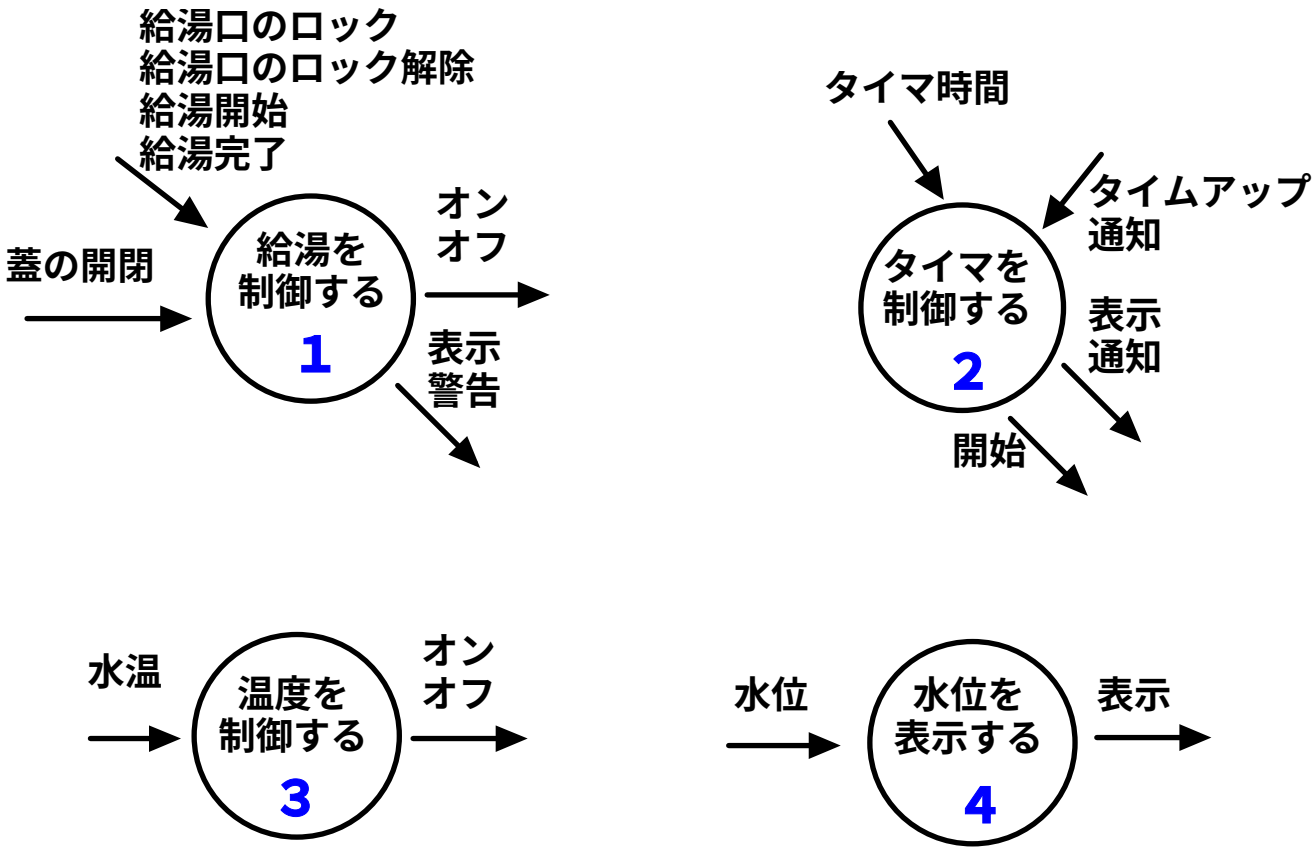
分析スタート！

分析の入り口

・コンテキスト・ダイヤグラム



分析　ステップ1　～DFD0～



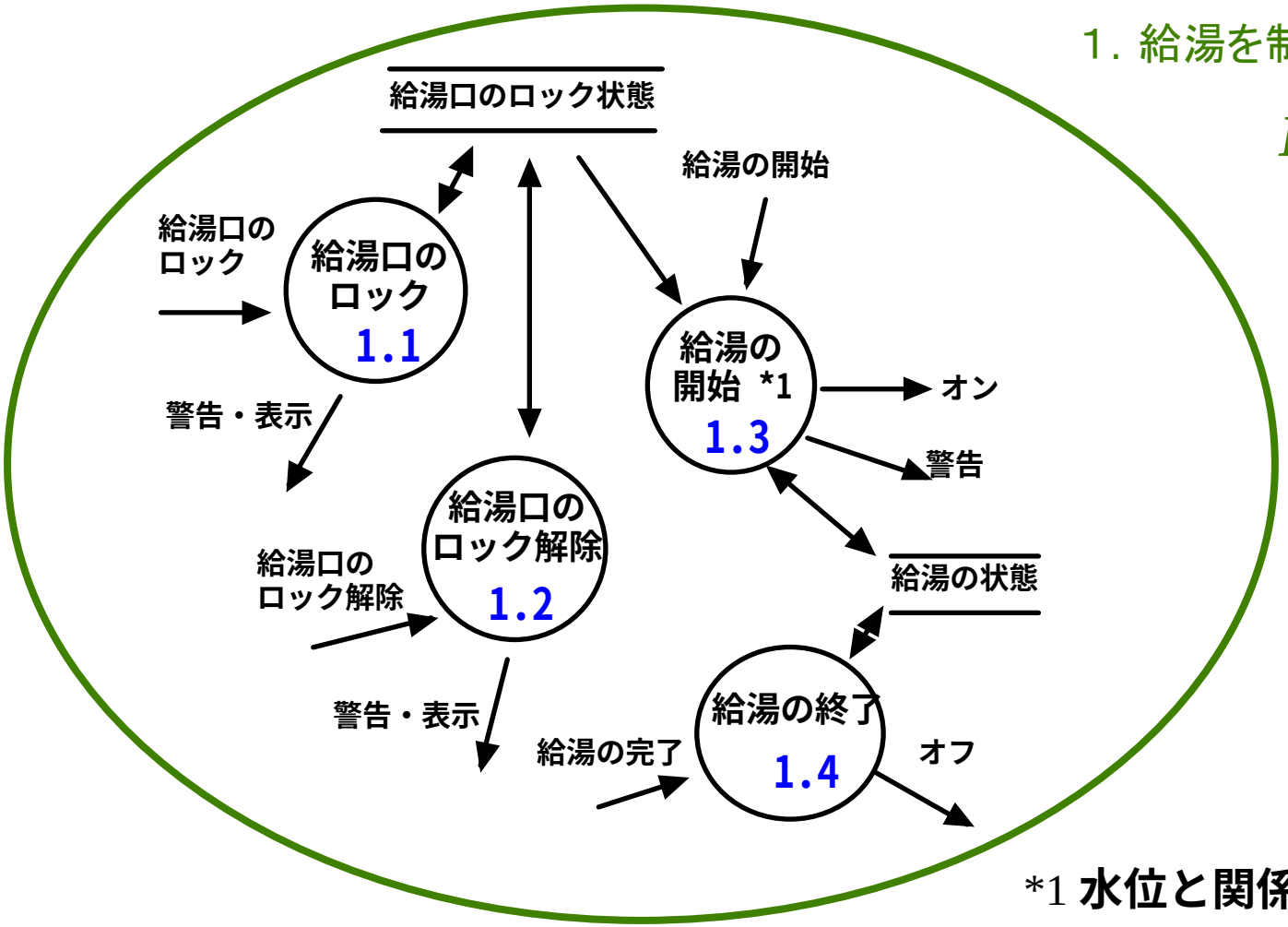
DFD作成の基本ルール

- 粒度と曖昧な言葉に注意
- 曖昧な言葉を使わない
  「処理する」「制御する」「データ」

- DFD0でプロセスが多すぎると、まとめ直しを
- ７±２の理論
  イベントリストの完成度が低い
  イベントリストにある１つのリストに対して
  プロセスを作っていないか？

- DFD0の上位レベルでは、いくつかのプロセスを
  まとめたもので、名前は少し曖昧でもよい

分析　ステップ２　～DFD1～



DFD作成の指針

- 先ずは書き始める
  書いたものを見て改善
  頭の中でパッとまとまりゃ苦労しない

- プロセスの兵糧攻め
  プロセスに出入りするフロー数を制限
- プロセスへの均等なフローの割り当て
  整理する→フローの掌握、抽象化
  哲学者
- 名前は自ずから
  感性の世界
  芸術家

DFDを用いた分析

- 分析とは

  かみ砕き、分類し、整理し、抽象的概念で再構築する作業

  芸術家の感性と哲学者の思考

  → 良いソフトウェア開発者に要求される資質と同じ

- DFDを用いた分析

  データの変換に沿っての分析

  → ソフトウェア開発者には馴染みのフィールド

  図形表現でのビジュアル化

  WhatとHowを切り離して設計

  → 巨大化・複雑化するシステムへの対応

制御フロー・ダイヤグラム（CFD）

・制御フローの流れを規定

フロー上をデータが流れればいいのか
条件の組み合わせで活性化されるプロセスが違うのか

○上位層のシステムの状態を制御するロジックの記述
×プリミティブ・プロセス同士の相互作用の詳細の記述

・でも、制御は控えめに

DFDを最大限に、制御は最小限に

×ついつい実装を意識　　モデルを抽象的なものに

・1つのDFDに対して制御フローを作成

データの流れと制御の流れを分離し明確に

CFD作成指針

・CFD作成順序

(1) DFD　　できるだけ

(2) 組み合わせコントロール　　仕方なし

(3) 順次処理コントロール　　最悪

・道具立て

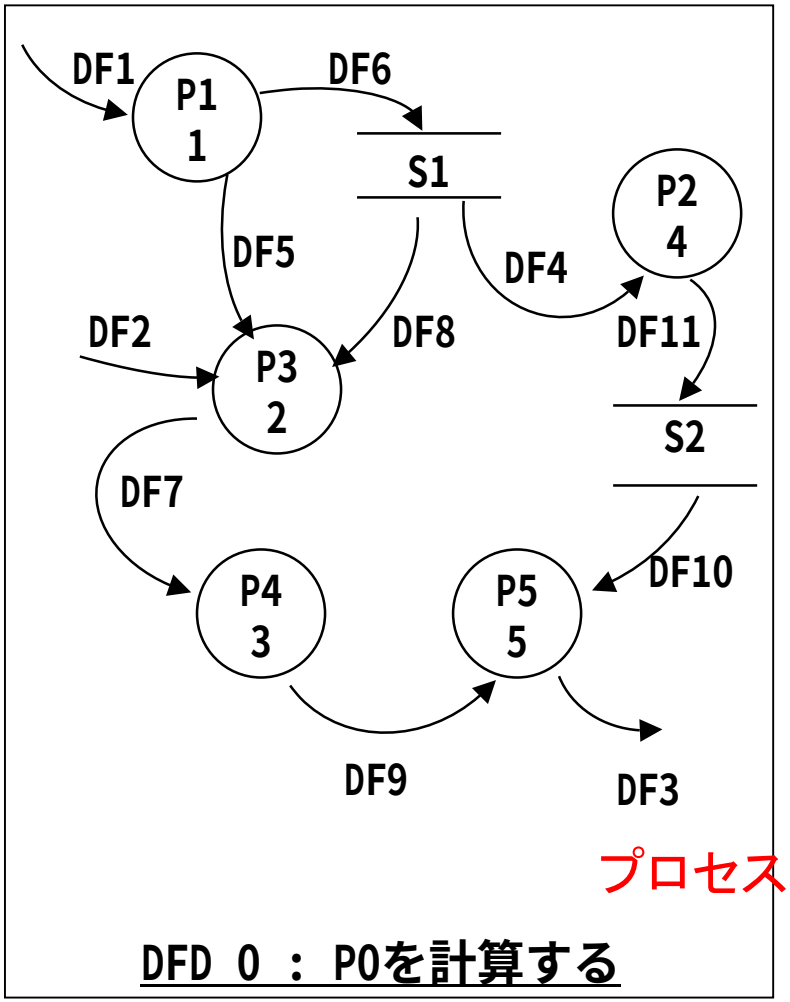
組み合わせコントロール

デシジョンテーブル（真理値表）

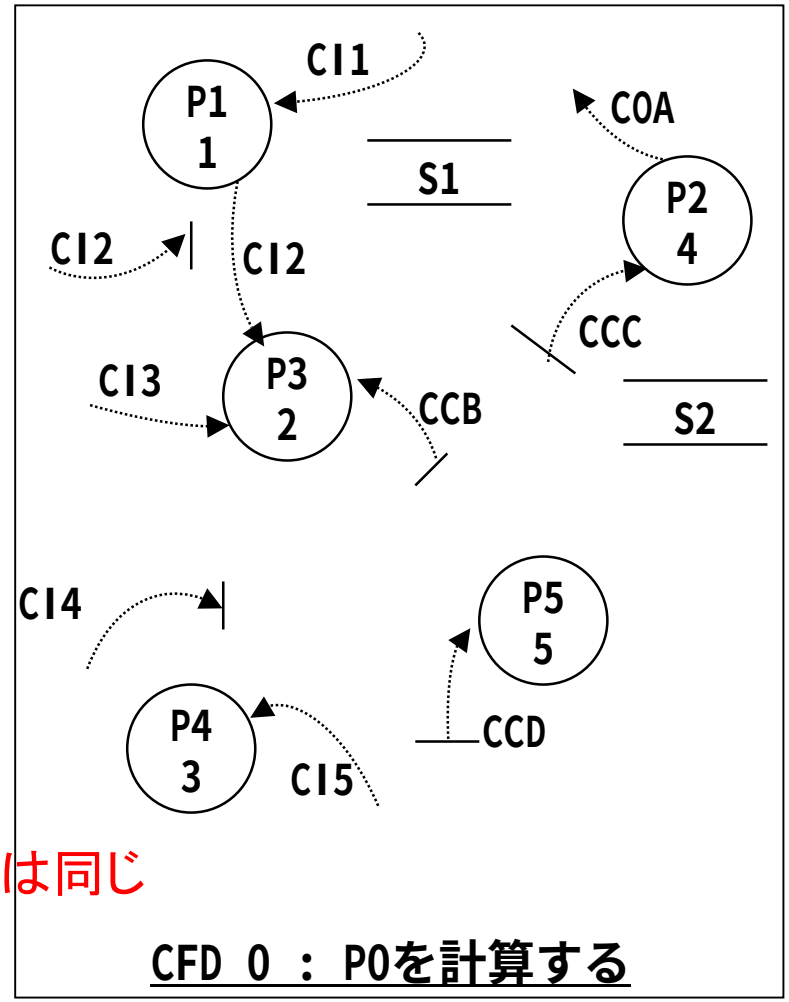
順次処理コントロール

状態遷移図

DFDとCFD



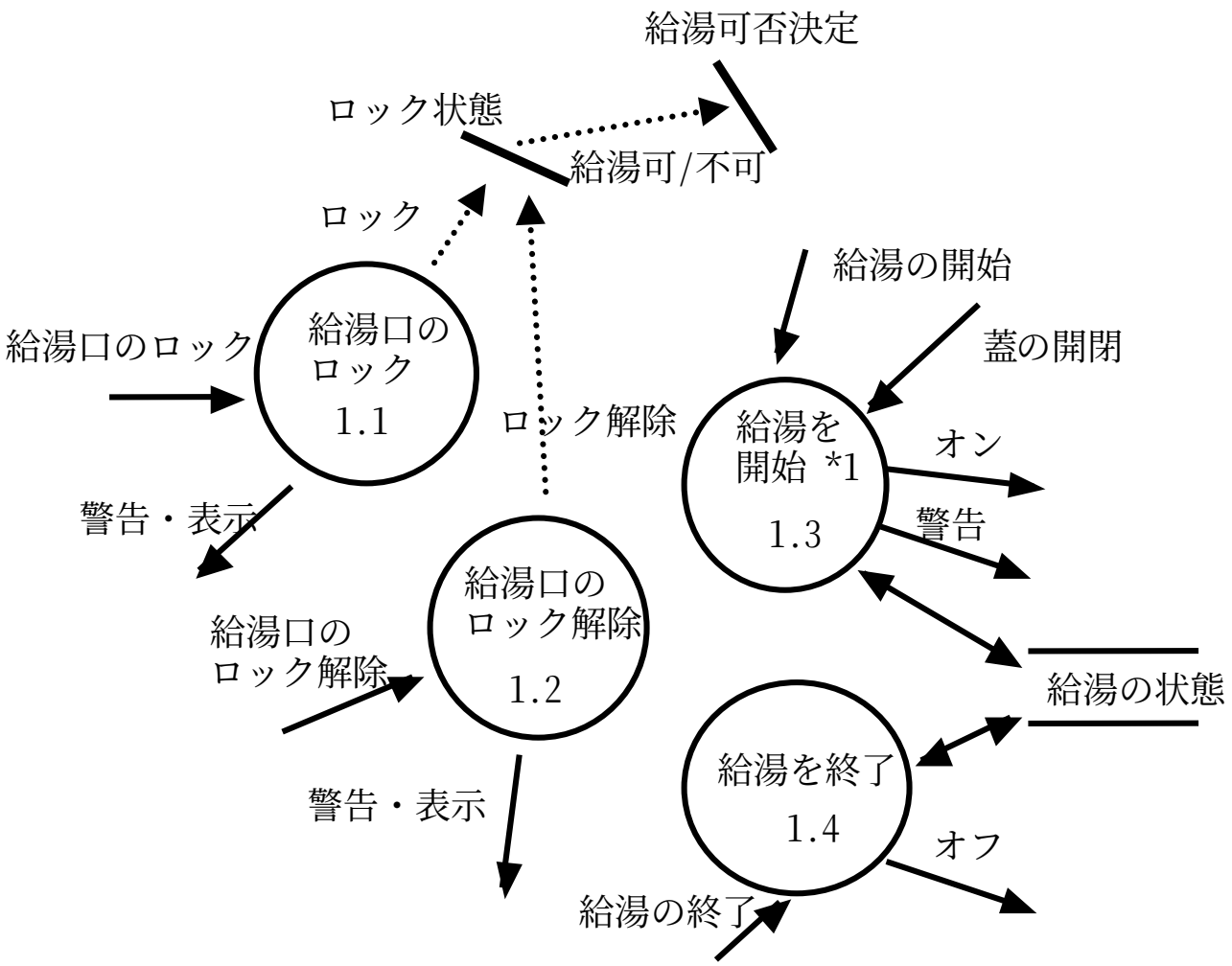
DFD 0 ： P0を計算する



CFD 0 ： P0を計算する

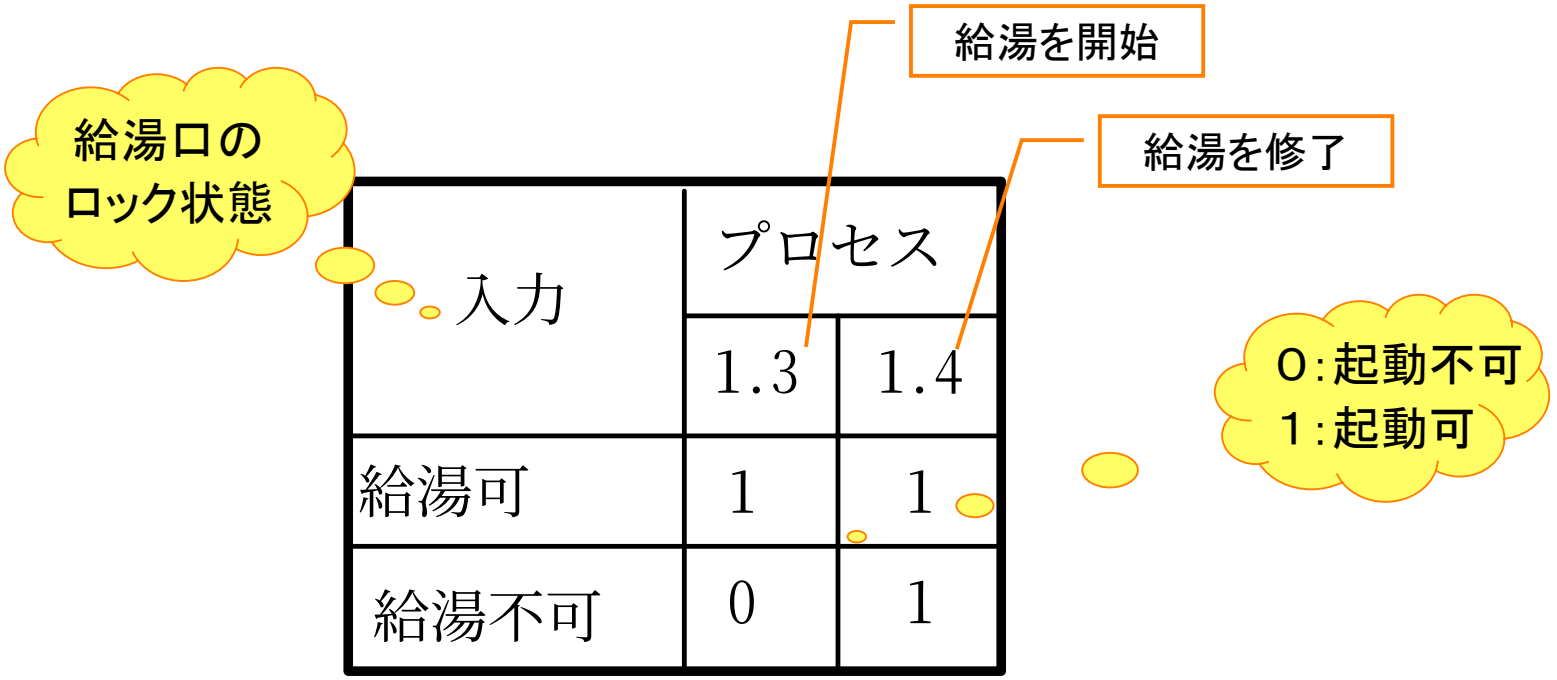
プロセスは同じ

CFD



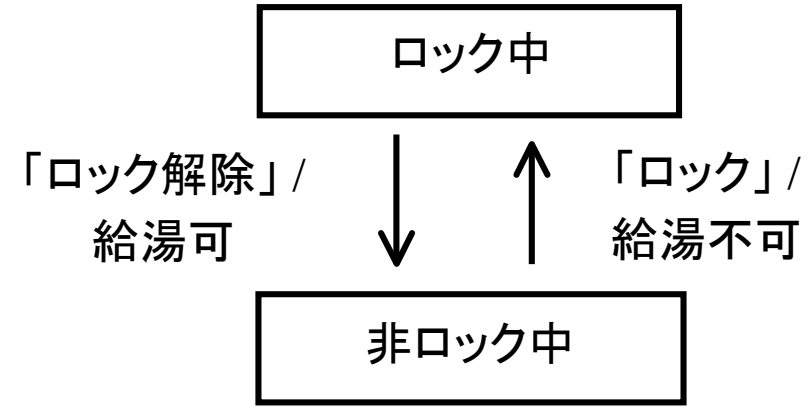
組み合わせコントロール

- 入出力データの全てが2値ならブール式
- 入出力データのどれかが多値ならデシジョンテーブル



順次処理コントロール

- 状態遷移図、状態遷移テーブル、状態遷移マトリックス
- 状態遷移図　７±２の理論



- 状態遷移テーブル

現在の状態	イベント	アクション	次の状態
ロック中	ロック解除	給湯可	非ロック中
非ロック中	ロック	給湯付加	ロック中

プロセス仕様書

- リーフのプロセスに対して作成
- 入力から出力への変換がなすべきことを記述

  → まさにデータフロー
- 記述方法は目的にかなっていれば自由

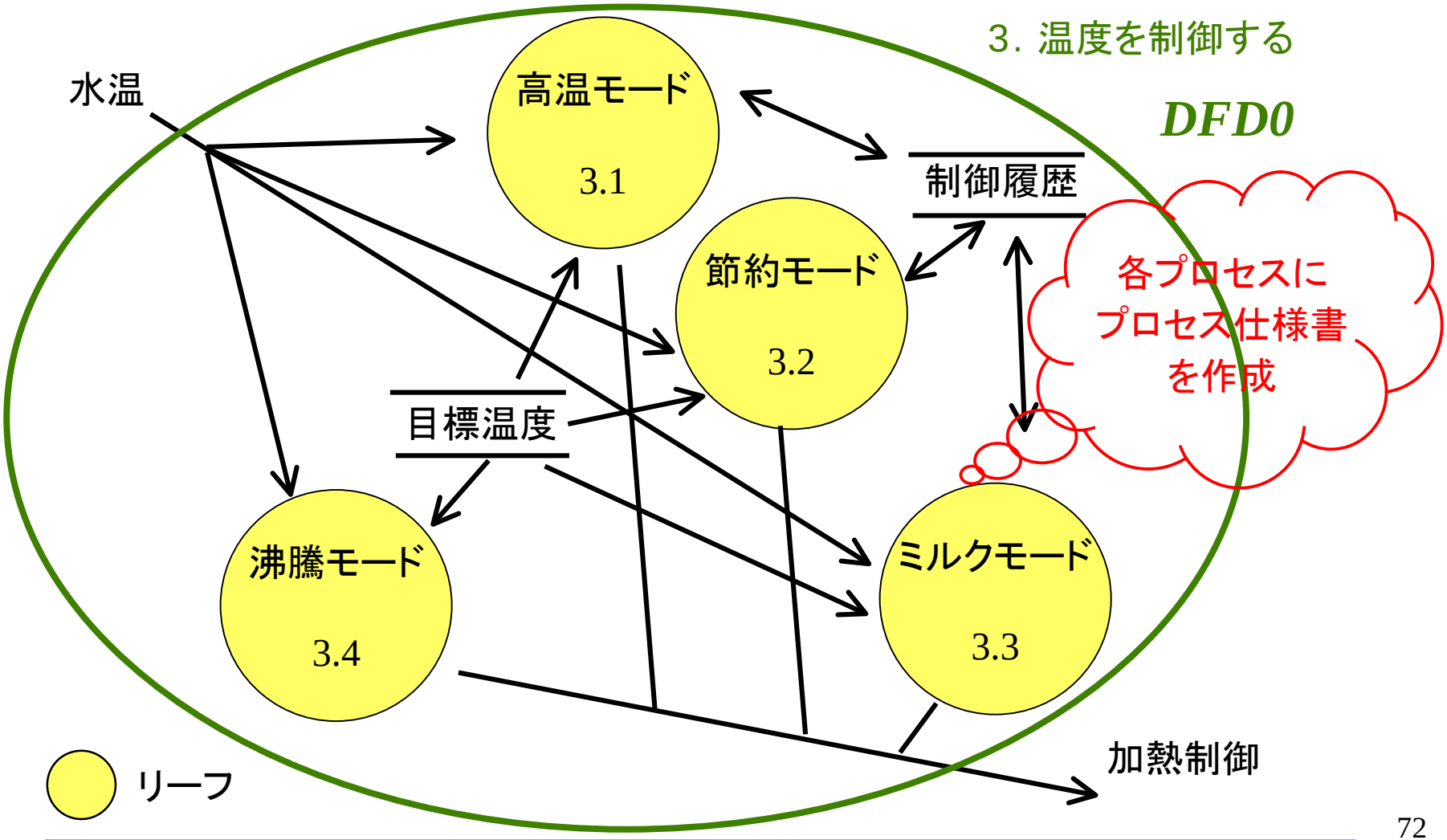
  文章 / 数式 / 表や図
- しかし

  疑似コードは×

  文章形式は構造化言語で

  （使用する言葉と構文を決めておき、冗長性を排除）
- 局所的な記述なら、デシジョン・テーブル / 状態遷移図等もOK

温度制御とプロセス仕様書



プロセス仕様書 P3.1　～高温モードでPID制御する～

制御周期ごとに、

1. // 水温履歴を更新する

$T_2=T_1$　$T_1=T_0$　T_0 = 水温

2. // 目標温度を得る

Tg を目標温度　モード＝高温モードである、

目標温度:目標値に設定する

3. // PIDでの制御量を計算する

$\Delta M=Kp(T_1-T_0)+Ki(T_g-T_0)+Kd(2T_1-T_0-T_2)$

今回の加熱率＝前回の加熱率＋ΔM　// 加熱率が100%以上になることは今回考慮しない

4. // ヒータ通電期間を制御する

制御期間×今回の加熱率の期間中は、

加熱制御=“on”

この期間が終了するタイミングで

// ヒータを切る

加熱制御=“off”

// 加熱率の履歴を更新する

前回の加熱率＝今回の加熱率　// 更新

プロセス仕様書 P3.4　～沸騰モード～

// 沸騰させてから3分間煮沸する

１. // 沸点まで連続的に加熱する

　　加熱制御 = “on”

　　水温 ≧ 100℃　となるまで待つ　// 異常の監視はしない

２. 以下を制御周期単位で3分間継続実行する

　　if　水温 ≧ 100℃　加熱制御 ＝ “off”

　　else　加熱制御 ＝ “on”

アジェンダ

1．何故、分析を？

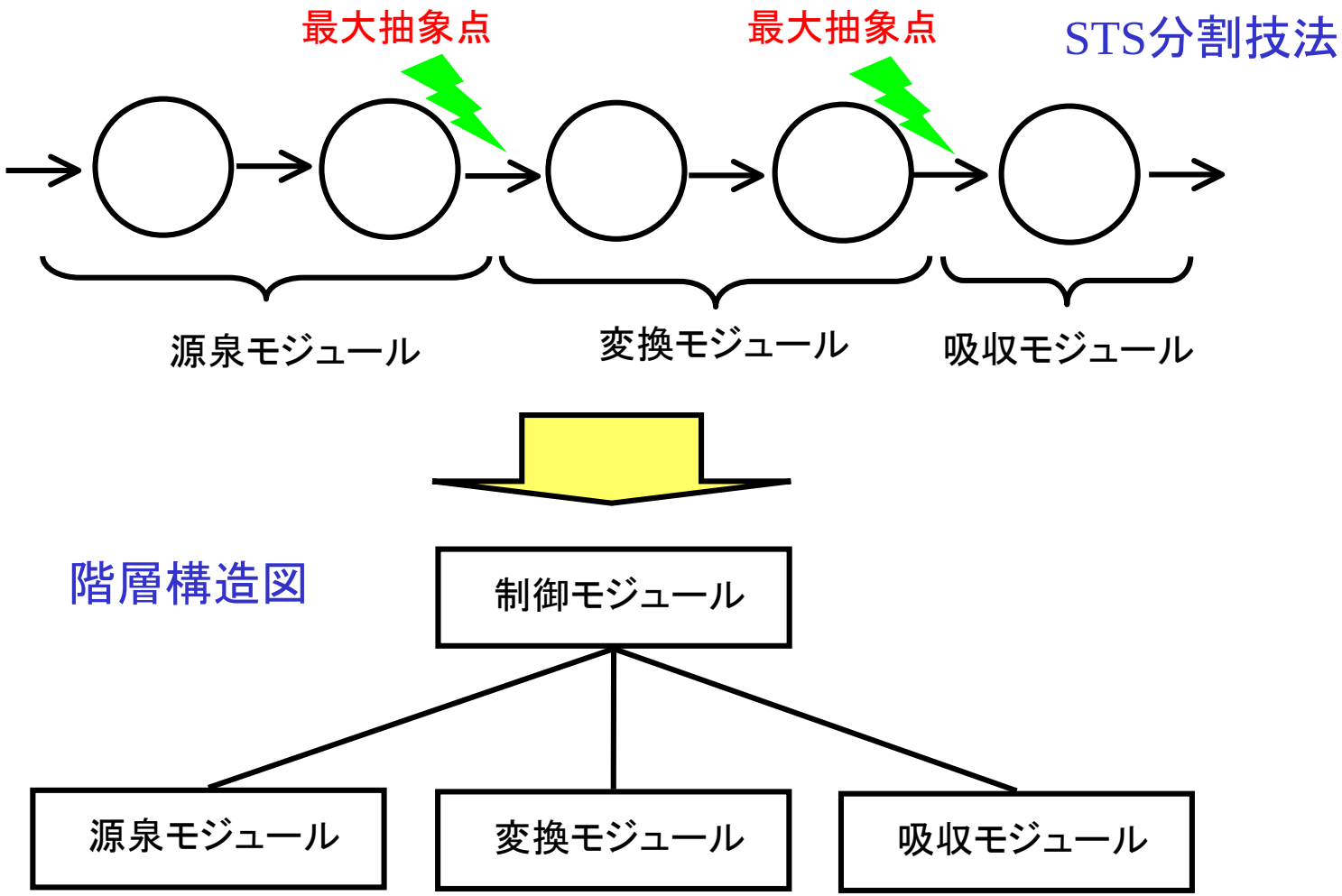
2．構造化分析と構造化設計

3．話題沸騰ポットにおける構造化分析

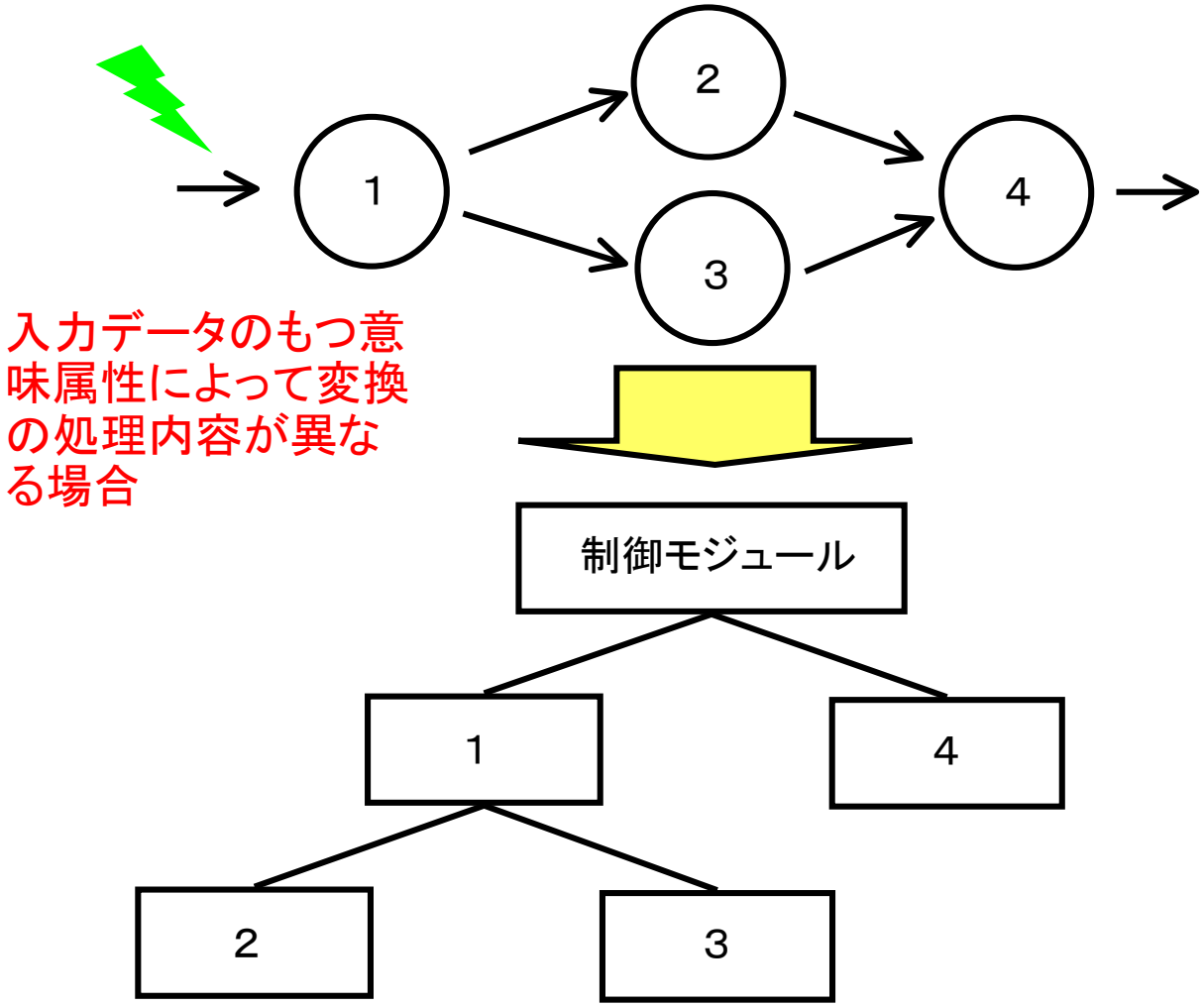
4．構造化設計概要

5．まとめ

構造化分析から構造化設計へ（1）



構造化分析から構造化設計へ（2）



まとめ

・分析しましょ！

かみ砕き、分類し、整理し、抽象的概念で再構築する作業

芸術家の感性　　哲学者の思考

→　良いソフトウェア開発者に
　　要求される資質と同じ

・手法は何であれ、芸は身を助く！

分析でプロジェクトに潜む病魔をチャッチ
→　あなたの分析がプロジェクト全体を救う

一段上のエンジニア目指して、***Let's analyze!***

参考

・Hatley and Pirbhai：“リアルタイム・システムの構造化分析”,
日経BP社, 1989

・リアクティブ・オートマトン法
http://www.sft.cs.musashi-tech.ac.jp/~yhm/index.html

・坂本：“ソフトウェア高信頼化のアプローチ”,
第５回組込みシステム開発技術専門セミナー, ES-6, pp. 37-69 (2002)

組込み向け構造化設計

担当：リコー　山田 大介

1. SESSAMEの紹介およびコースの概要
2. 開発課題と失敗事例の解説
3. 組込み向け構造化分析の例
4. 組込み向け構造化分析・設計の概要
5. 組込み向け構造化設計
6. プログラミング
7. ソフトウェアテストの概要
8. 話題沸騰ポットに対するテストの実践
9. 大規模開発に向けての注意点

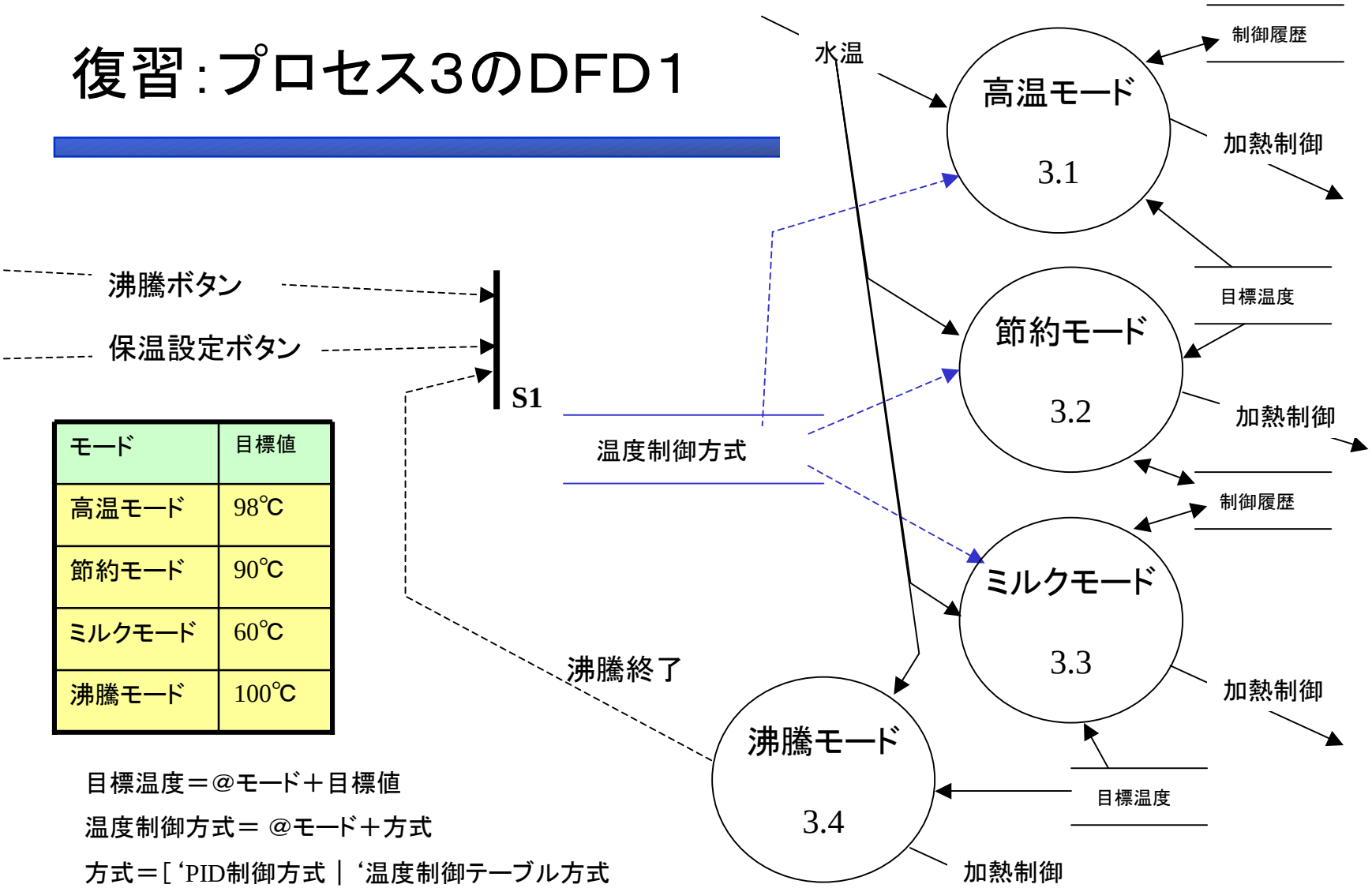
アジェンダ

1. 構造化設計
   - 分析モデルから設計の構造図へ
2. 設計の品質
   - モジュール分割、凝集度と結合度
3. アーキテクチャ設計
   - 非機能要件の設計
4. モジュール仕様
   - 構造図からモジュール仕様へ
5. 仕様変更への対応
   - 長持ちする設計

アジェンダ

1. 構造化設計
2. 設計の品質
3. アーキテクチャ設計
4. モジュール仕様
5. 仕様変更への対応

復習：プロセス3のDFD1



モード	目標値
高温モード	98℃
節約モード	90℃
ミルクモード	60℃
沸騰モード	100℃

目標温度＝@モード＋目標値

温度制御方式＝ @モード＋方式

方式＝[‘PID制御方式｜‘温度制御テーブル方式

｜‘目標温度OnOff方式｜‘ヒステリシス方式]

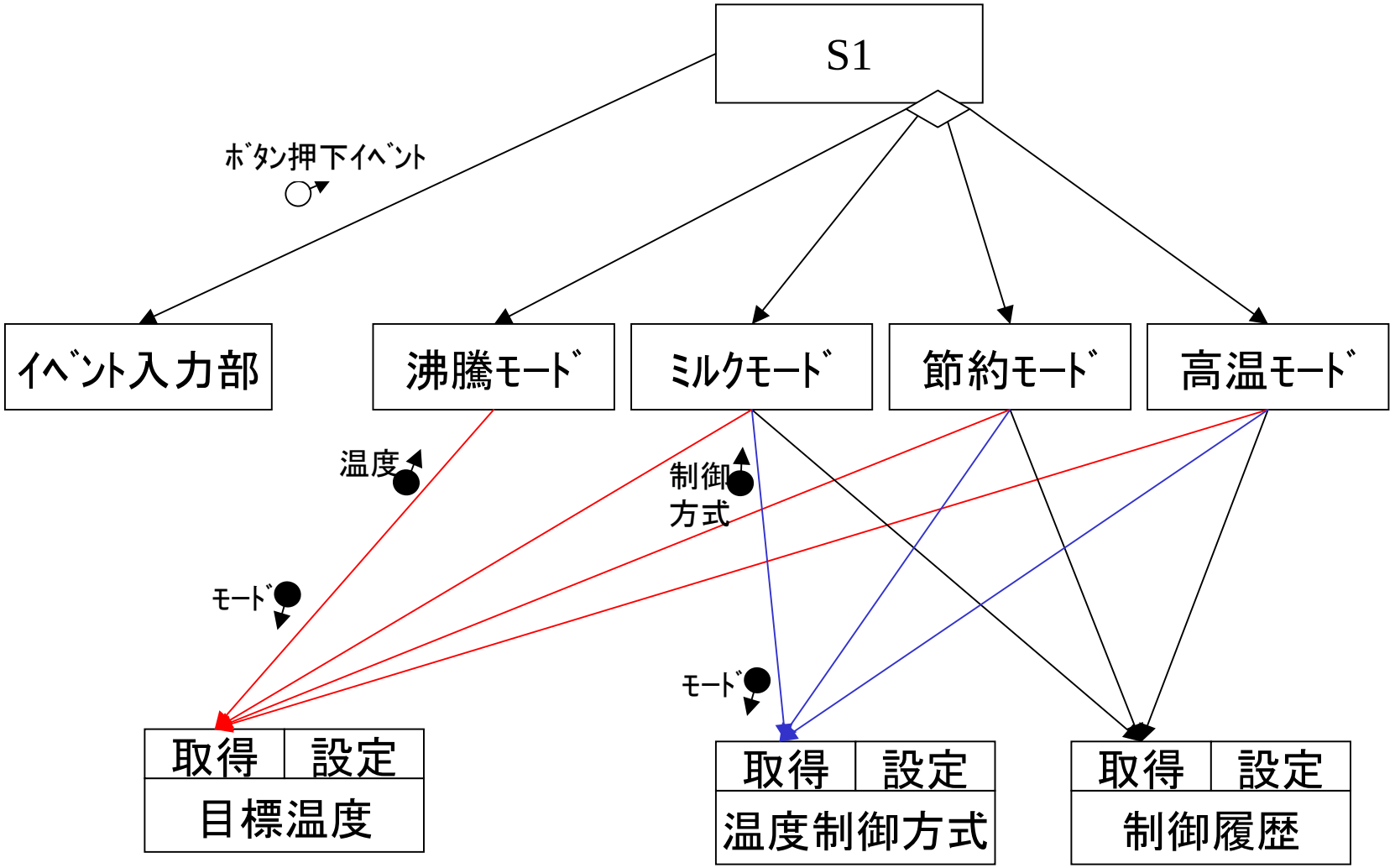
制御履歴＝前回の加熱率＋T2+T1+T0

構造化設計の手順

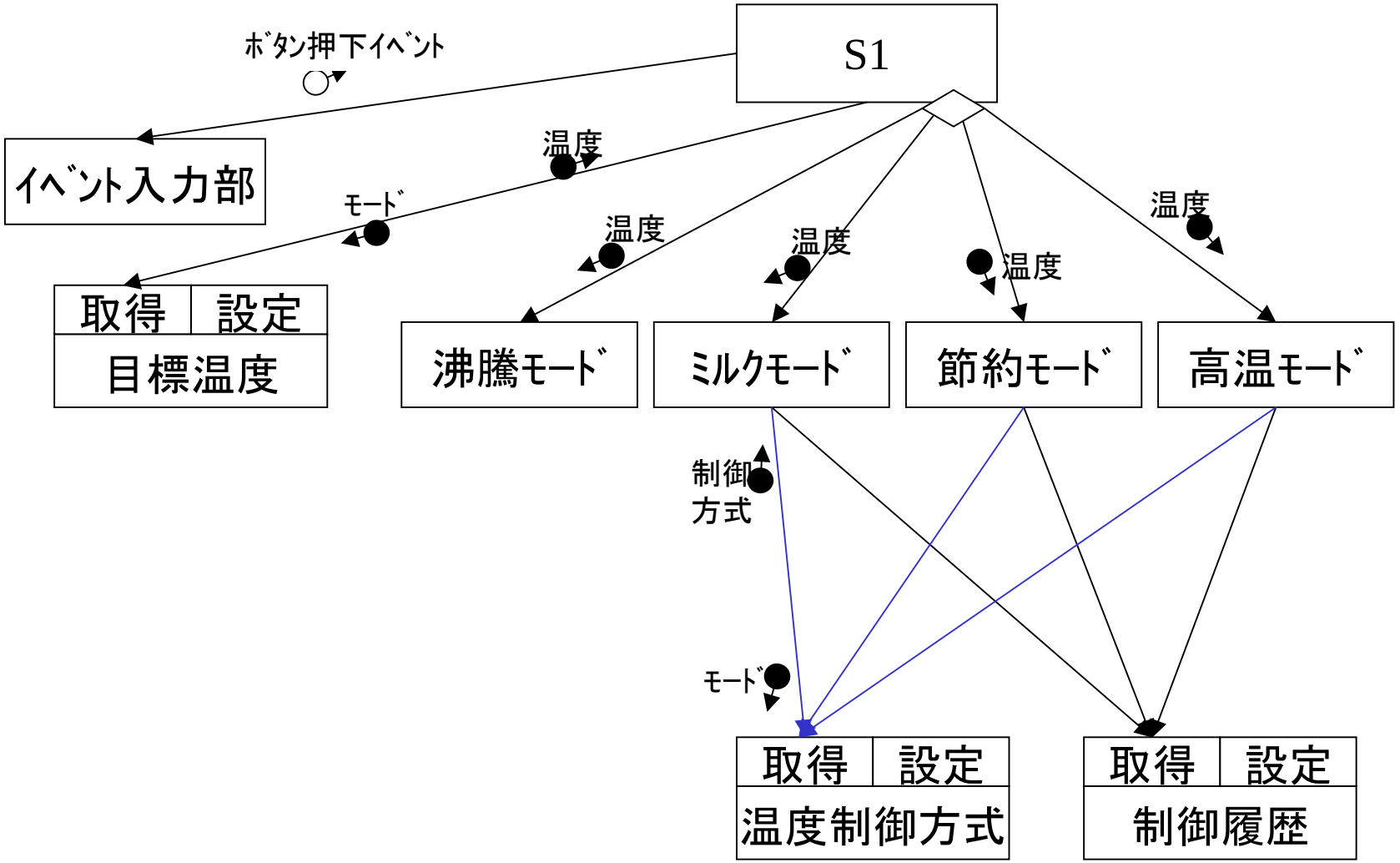
- 構造化分析のDFDに基づき、
- 最大抽象点を見つけて、源泉－変換－吸収スタイルを作る
  - その際に、ボスとして制御モジュールを作る
- 制御バーをモジュールに置き換える
  - ディスパッチャになる
- その他のバブルをモジュールに置き換える
  - そのバブルの下に、下の階層のバブルがぶら下がっていく
- 外部データの入力モジュールを追加する
- データの入出力を考える
  - データのアクセスにはデータ隠蔽モジュール

次に示す2つの案が簡単に設計できる

案1



案2



構造図の要点

- 構造図は、分析時のDFDからある程度機械的に作成できる
  - 最大抽象点を見つけることにテクニックは必要
    - 最も本質的な形で入力を表現しているデータフローと、最も本質的な形で出力を表現しているデータフローで囲まれた部分

- あとは、次に示す設計基準（凝集度と結合度）により、モデルの選択と、若干の構造見直しを行う

アジェンダ

1. 構造化設計
2. 設計の品質
3. アーキテクチャ設計
4. モジュール仕様
5. 仕様変更への対応

設計の品質

- モジュール分割
- モジュールの凝集度（コヒージョン）
- モジュール間の結合度（カップリング）

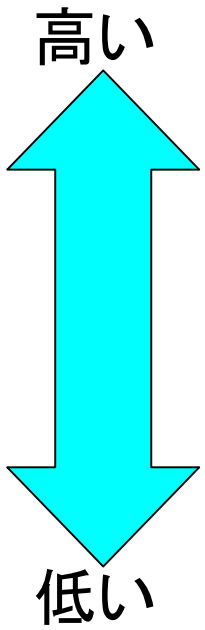
モジュール分割

- モジュールサイズの適正化
  - 1ページの半分程度(30行)
  - 実装が単純になる
- システムの明解さ
  - 均衡型システム(左右均衡、上下は論理ー物理)
  - モジュールの関係だけでシステムが理解できる
- 重複の極小化
  - 同じ機能を複数のモジュールに持たせない
- 情報隠蔽
  - モジュールで取り扱うデータを隠蔽する

凝集度（コヒージョン）

- 凝集度は「高い」ほど保守性が良い

1. 機能的
2. 逐次的
3. 通信的
4. 手順的
5. 一時的
6. 論理的
7. 偶発的



良い凝集度

1. 機能的
   - 単一の目的を持った機能、情報、責務。
2. 逐次的
   - 同一のデータに対する、
   - 関連が強く、必然的な順番を有す機能集合。
3. 通信的
   - 同一のデータに対する、
   - 複数目的の機能集合。

以上が、凝集度が高く、保守しやすいモジュール

中程度の凝集度～悪い凝集度

4. 手順的
   - 関連の薄い複数データに対する、順次機能。
5. 一時的
   - 同一時間に発生する、互いに関係のない機能集合。
6. 論理的
   - 外部から利用する時に便利な機能の寄せ集め。
7. 偶発的
   - まったく相関のない機能の寄せ集め。
   - 実装制約（HOW項目）を意識してしまうと、、、

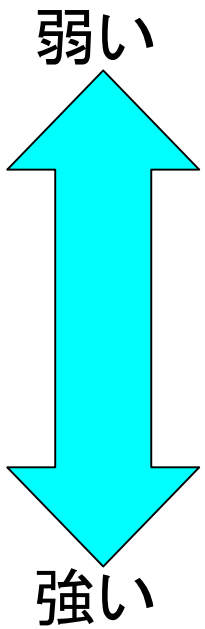
時の経過とともに、

モジュールの存在理由が曖昧になっていく

結合度（カップリング）

- 結合度は「弱い」ほど保守性が良い

1. データ結合
2. 構造体結合
3. バンドリング結合
4. 制御結合
5. ハイブリッド結合
6. 共通結合
7. 内容結合



良い結合度

1. データ結合
   - 構造を持たない単一のデータ。
2. 構造体結合
   - 同一の目的を持つデータメンバの集合。
3. バンドリング結合
   - 複数のフィールドから構成される構造体。
   - 使い方自由の万能構造体は良くない。

以上が、結合度が弱く、

理解しやすく、波及作用が限定的なモジュール間結合

中程度の結合度～悪い結合度

4. 制御結合
   - 相手のモジュールを制御するフラグ。
5. ハイブリッド結合
   - 値の範囲により意味の異なるデータやフラグ。
6. 共通結合
   - グローバルデータ。
7. 内容結合
   - 相手のモジュールの内部を参照。(アセンブラ)

変更すると、思わぬところに副作用が、、、

分析の効用

分析を行うことにより、

- 要求の曖昧個所や矛盾点に気づくだけでなく
- ある程度機械的に設計が可能となる
- さらに、「凝集度」が高く、「結合度」が弱い、という高品質な設計も手に入る

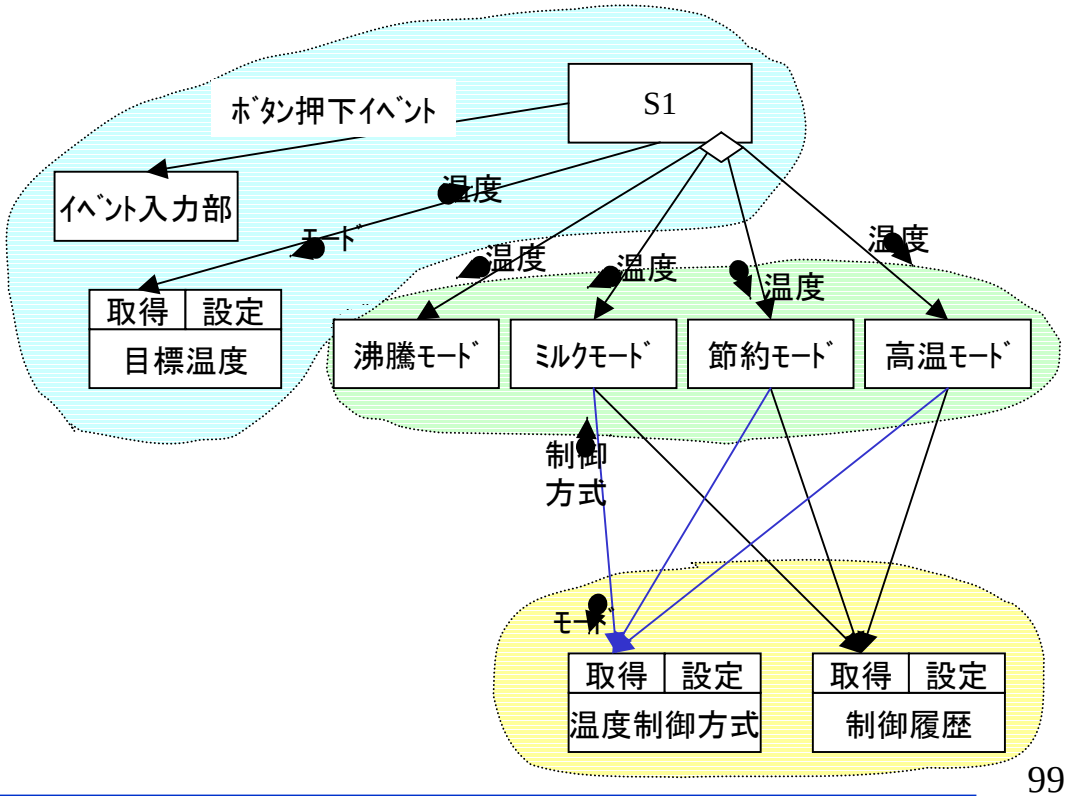
上流工程で品質を作り込もう

アジェンダ

1. 構造化設計
2. 設計の品質
3. アーキテクチャ設計
4. モジュール仕様
5. 仕様変更への対応

アーキテクチャ設計

- システム外部からのイベント
  - 制御スレッドの検討
  - 割り込みの利用
- 並行処理
  - パッケージ分割
    - 構造図で破線囲い
  - サブシステム分割
    - 人員配置にも



非機能要件

- 時間的制約
  - 応答時間
    - ハードリアルタイム／ソフトリアルタイム
- リソースの制約
  - ROM／RAM
  - CPUパワー
- 環境上の制約
  - 開発環境、工場検査、市場メンテナンス
- フォールトトレランス
  - 過負荷、フェイルセーフ

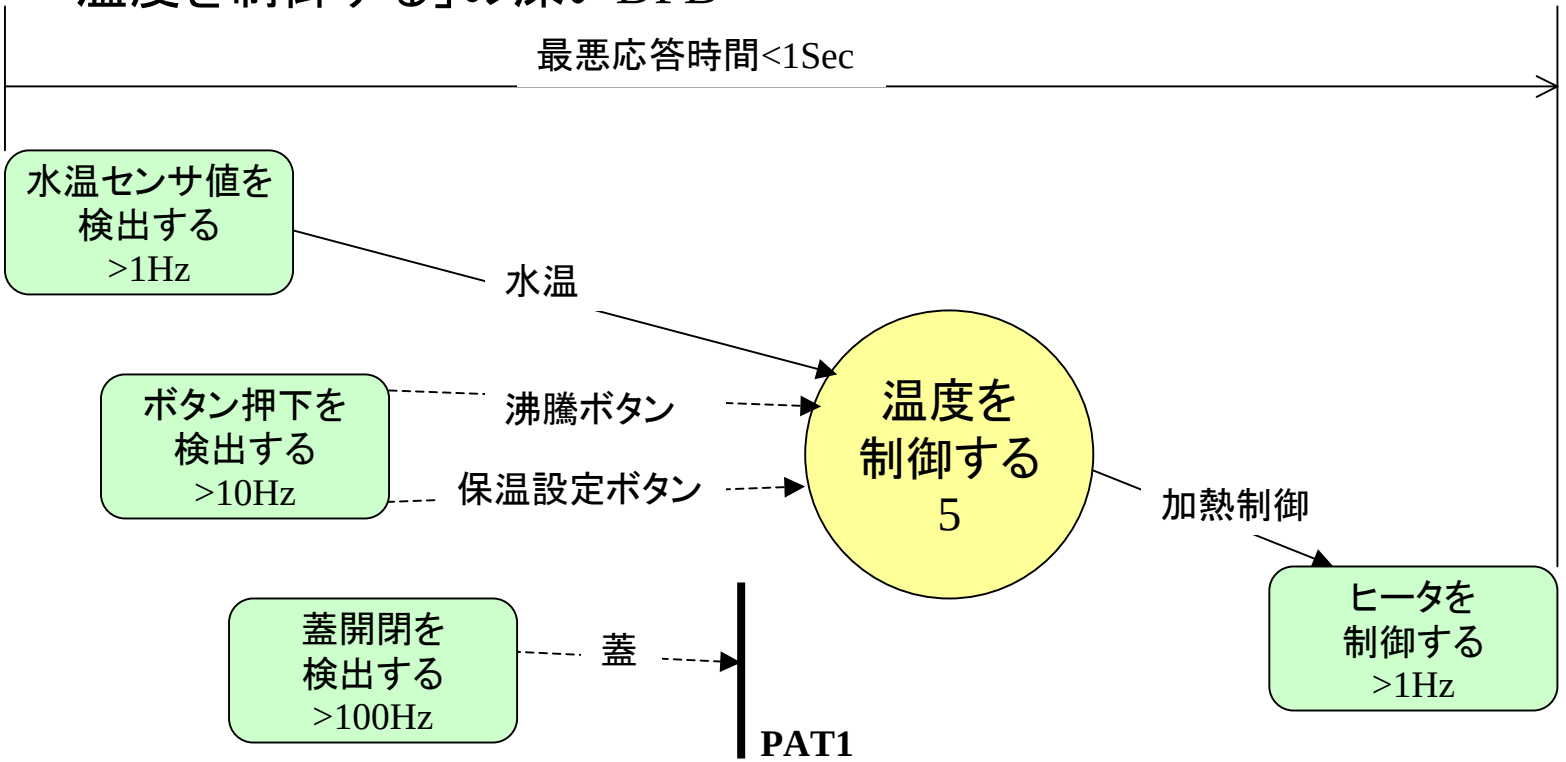
非機能要件：ポットの例

- レートモノトニック要求
  - ヒータのOnOff制御（PID制御は、1秒毎）　精度±20mSec　ラインクロック
- イベントドリブン要求
  - 蓋の開閉検出　必須　ハード割り込み
  - 温度異常検出　不用
  - 満水と水なし検出　不用
- データ構造要求
  - 温度履歴の保持方法　3点までのリニアバッファ
  - 復帰のメカニズム　invocationで対応
- 製品対応要求
  - ROM実装量を最小化＋保守容易化　SCM*環境で開発
- エラー処理メカニズム要求
  - エントリ引数チェックとトラップメカニズム　コーディングルールで対応

*SCMとは、ソフトウエア構成管理

非機能要件：ポットの例

- 「温度を制御する」の深いDFD

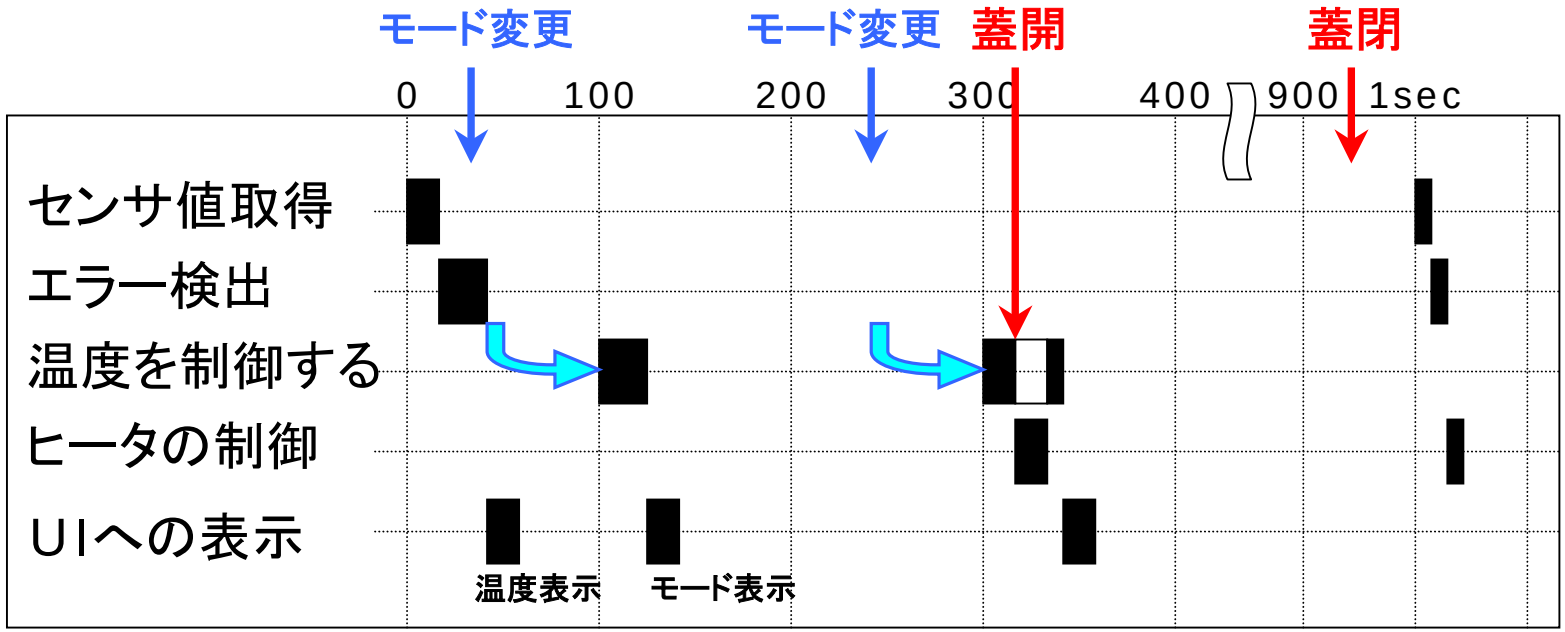


蓋の開閉を割り込み処理

ボタン押下と水温検出を周期制御

タイミングと並行性の検討

- センサ値取得が、1sec周期
- ボタン押下（モード変更）の検出が、100msec周期
- 蓋の開閉は、割り込み処理



タイミングと並行性の検討

- タイムリー性
  - ワーストケースを積み上げてみる
  - ハードリアルタイム　　ハードウエア処理の遅延がないか？
  - ソフトリアルタイム　　ユーザへの応答性は大丈夫か？

- 並行性
  - 処理時間がかかるもの、ウエイトが発生するもの、の抽出
  - スケジューリング方式を検討
    - イベントドリブン、レートモノトニック
  - 共有するリソースやデータが存在するか？
    - 何らかの排他処理

ここあたりに経験に基づく暗黙の知識が？

- スケジューリング方式の選定
- パフォーマンスチューニング
- ROMサイズのオプティマイズ
- RAMのセクション設定、スタック量の見積もり
- フォールトトレランス

など？

皆さんが普段留意している事項を書きとめてください。
可能であれば失敗事例もお願いします。

アジェンダ

1. 構造化設計
2. 設計の品質
3. アーキテクチャ設計
4. モジュール仕様
5. 仕様変更への対応

モジュール仕様

- モジュール単位のインターフェイス仕様と機能仕様
- インターフェイス仕様
  - そのモジュールが「何をすべきか」をインターフェイス仕様で定義
  - インターフェイス契約
- 機能仕様
  - そのモジュールが「どのように実現するか」を擬似コードで定義
  - 分析時のP-SPECを利用可能

モジュール仕様一覧の例

- 割り込み処理
  - 蓋開で、ヒーター制御を停止
- 制御クロック算出
  - ラインクロックでの制御周期計算
- 「温度を制御する」のディスパッチャ
- 「温度を制御する」のイベント入力部
- 「温度を制御する」の高温モード
- 「温度を制御する」のミルクモード
- 「温度を制御する」の沸騰モード

モジュールのインターフェイス仕様の例

名　称:「温度を制御する」の高温モード

目　的:高温で保温するときの温度制御

引　数:なし

戻り値:なし

モジュールの機能仕様の例

- 擬似コード
  - P-Specより抜粋

制御周期ごとに、

1. //水温履歴を更新する

$T_2=T_1$

$T_1 = T_0$

$T_0 =$ 水温

2. //目標温度を得る

Tgを目標温度. モード＝高温モードである
目標温度. 目標値に設定する

3. //PIDでの制御量を計算する

$\Delta M=Kp(T_1-T_0)+Ki(T_g-T_0)+Kd(2T_1-T_0-T_2)$

4. //ヒータ通電期間を制御する

制御期間×今回の加熱率　の期間中は、

加熱制御=“on”

この期間が終了するタイミングで

//ヒータを切る

加熱制御=“off”

//加熱率の履歴を更新する

前回の加熱率＝今回の加熱率　//更新

ここにはちょっとした罠が、、、

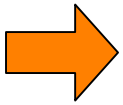
アジェンダ

1. 構造化設計
2. 設計の品質
3. アーキテクチャ設計
4. モジュール仕様
5. 仕様変更への対応

仕様変更：ポットの例、例えば、

- ミルクモードのときは、「62度を上限値、58度を下限値」という仕様変更
- いきなりプログラミングした場合、
  - プログラムの流れを順を追って読み、
  - ソースコード上で変更個所を特定する
  - スパゲッティになってしまっている場合は、
  - 変更箇所の特定すらできないことも
  - また変更による副作用も恐い
- 設計図がある場合、
  - 構造図から変更モジュールが特定できる
  - 外部のインターフェイスを変えないように、内部の処理を変えよう

62度を超えたぞ、
ミルクモードだから、
一度沸騰させているし、
ヒータのエラーにもなっていないから、
ここでヒータを停止させよう。
でもこの手前の分岐は何かな？？
今ヒータ止めてもいいのかな？？



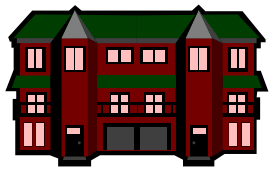
変更箇所の特定は素早く、修正は局所的

仕様変更への対応

- プログラムの流れを追って修正（増築）
  - ソースコードを見ないと修正個所が特定できない
  - やがてスパゲッティ化
  - ソースコードだけが信頼できる

老舗温泉旅館

- 設計図から修正個所を特定
  - 修正個所が局所化される
  - 制御された修正



工学的建築
（基礎がしっかり）

仕様変更への対応

- 「長持ちする設計」が良い設計
  - 固定変動分離
    - 外部へのインターフェースを固定
    - 内部処理は変動容易
    - 固定部から変動部を呼び出して、変動部だけを修正
- 設計図でレビューしましょう
  - ソースコードを追わないと分からない
    - 作った人しか分からない
    - さらに、副作用に関しては未知
  - 抽象度の高い設計図でのレビューが効果的

まとめ

1. 構造化設計
   - 分析さえしっかりすれば設計は簡単。分析しましょう！
2. 設計の品質
   - 高い凝集度と弱い結合度で、保守しやすいソフトウエアへ！
3. アーキテクチャ設計
   - いろいろ暗黙の知識がありそうですね。
   - 失敗事例などの知識を共有化する仕組みを作りましょう！
4. モジュール仕様
   - 構造図さえあればここも簡単。但し、状態やモードに注意。
5. 仕様変更への対応
   - 「長持ちする設計」を目指し、設計図でレビューしましょう！

以上が、設計でした。
次はもちろん（やっと？）プログラミングです。

その設計は、変更に耐えられますか？

参考

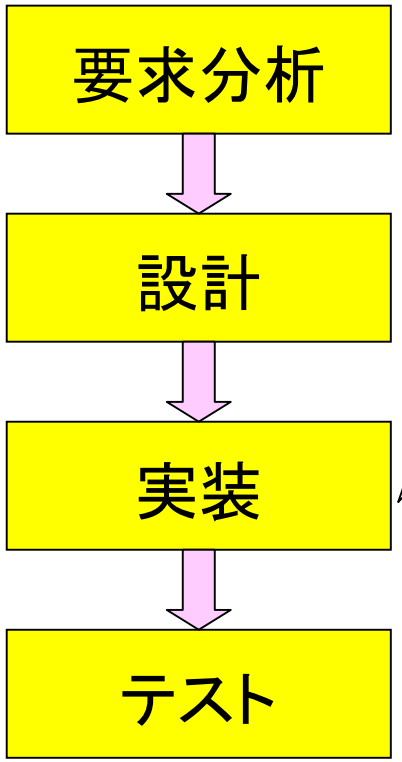
- Meilir Page-Jones：”構造化システム設計への実践的ガイド“，近代科学社，1991

プログラミング

担当：（株）富士通コンピュータテクノロジーズ　　上原慶子

1. SESSAMEの紹介およびコースの概要
2. 開発課題と失敗事例の解説
3. 組込み向け構造化分析の例
4. 組込み向け構造化分析・設計の概要
5. 組込み向け構造化設計
6. プログラミング
7. ソフトウェアテストの概要
8. 話題沸騰ポットに対するテストの実践
9. 大規模開発に向けての注意点

はじめに　ープログラミングー



第一章　プログラミング例

第二章　良い組み込みプログラムとは

第三章　C言語の書き方の留意点

1. プログラミング例

- 状態遷移マトリックス例
- プログラム例その1
- プログラム例その2
- プログラム例の特徴

1.1 状態遷移マトリクス例

状態番号	状態／事象	保温設定ボタン	沸騰ボタン	初期沸騰検出	沸騰終了
1	水位不足		変化なし		
2	加熱途中		変化なし	→3	×
3	初期沸騰	→4？？？	？？？	×	→4
4	高温保温	→7	→5	×	×
5	高温沸騰前	→8	変化なし	→6	×
6	高温沸騰中	→9	→4	×	→4
7	節約保温	→10	→8	×	×
8	節約沸騰前	→11	変化なし	→9	×
9	節約沸騰中	→12	→7	×	→7
10	ミルク保温	→4	→11	×	×
11	ミルク沸騰前	→5	変化なし	→12	×
12	ミルク沸騰中	→6	→10	×	→10

1.2 プログラム例その1(switch文)

各状態と事象をそれぞれ, 判断文で判定し, 関数を呼び出す

```
switch (status){
   case  RETAINING :  /* 高温保温中 */
      if ((cur_switch & BOIL_SWITCH) != 0) {/* 沸騰*/
         status = boiling; /* 沸騰モードへ遷移 */
         prboil( ); /* 沸騰処理呼び出し */
         }
      else  if ((cur_switch & RETAIN_SWITCH) != 0){/* 保温 */
         status = economy; /* 節約モードへ遷移 */
         preco( ); /* 節約処理呼び出し */
         }
...
```

1.2 プログラム例その2(関数ポインタの定義)

各状態と事象をそれぞれ, 二次元の配列で考え, 関数ポインタを使って呼び出す

```
struct  functbl {
    void   (*fp)(void); /* 呼びだす処理関数のアドレス */
    mode_type  status; /* 遷移する状態 */
};
mode_type   status;
#define   MAXSTATUS  12
#define   MAXEVENT    5
void   functbl[MAXSTATUS][MAXEVENT] = {
                { {prboil , boiling} , {preco, economy},  ・・・
```

1.3 プログラム例その2(関数ポインタによる呼び出し)

各状態と事象をそれぞれ, 二次元の配列で考え, 関数ポインタを使って呼び出す

```
void   (*callp)(void);
for (; ;) /* 無限ループ */
{
   /*イベント待ち処理およびevent領域へのイベント番号の設定 省略 */
   callp = functbl[status][event].fp; /* 呼び出す関数の設定 */
   nextstatus = functbl[status][event].status; /* 次の状態の設定*/
   (*callp)();  /* 状態とイベントに対応した処理の呼び出し */
   status = nextstatus; /* 状態の変更 */
}
```

1.4 プログラム例の特徴

1.switch文のネストは二つまでで, 状態遷移マトリックスに対応

2. 判断文が簡単

→追加, 変更がしやすい

組み込みプログラムの場合は, ハード開との並行開発となるため, プログラムの仕様変更が多い.

第二章 よい組み込みプログラムとは

1.品質のよいプログラムとは

2. よいプログラムを作る注意点

3. McCabeの複雑度

2.1 品質のよいプログラム

- ISO/IEC9126の品質特性で★がプログラミング時の注意点

機能性

★効率性‥実行時間が短く、資源の使用が少ない

★信頼性‥プログラムの障害発生率が低い

★保守性‥障害修正や機能追加変更が容易

使用性

★移植性‥他の環境でも動作可能

2.2 品質のよいプログラムにするためには

おもな注意点

効率性・・書き方によるオブジェクトサイズの違い
を習得する。(コンパイラにも依存)

信頼性・・コーディングミスを起しにくい文体

保守性・・わかりやすいプログラム

移植性・・標準的な書き方とコンパイラ依存事項の
知識取得

2.3 McCabeの複雑度

- 判定文が多いとプログラムの理解に時間がかかる.
- プログラムの複雑度とはプログラムがどの程度複雑でわかりにくいか? を数値で表す.
- プログラムの複雑度の計量法でよく使われているのが,
- McCabeの閉路複雑度

> McCabeの複雑度=判定数 + 1

McCabeの閉路複雑度 <=20 わかりやすい
McCabeの閉路複雑度 >20 理解に時間がかかる
McCabeの閉路複雑度 >50 作成者以外にはほとんど理解不可能

複雑度を下げるために関数分割をするのはよく考えてから

第三章　C言語の書き方の留意点

1. C言語の特徴
2. 優先順位の注意点
3. if文の書きかたのヒント
4. わかりやすいプログラムの文体
5. 効率性向上のヒント
6. 移植性のまとめ

.

3.1 C言語の特徴

1. 簡潔性

演算子により, 処理を簡潔に書ける. i = i + 1; i += 1; i++;

2. 自由度

一つの処理を, いろいろな書き方ができる.

3.不定(コンパイラ依存)

同じソースコードであってもターゲットによって動作が異なる場合有

4.アセンブラに近い

高級言語であるが, アセンブラにかなり近い言語.

生成されたオブジェクトコードも他の高級言語に比べ小さい.

コンパイラによってコード生成も異なる.

→コーディング規約が必要

3.2 優先順位の注意点の例

例)ビット演算(誤)
if (byte0 & 0X08 = = 0)
　この例ではbyte0 という1バイトデータと0x08のビット演算を行った結果を0 と比較しようとしている
= =と& では= =の方が優先順位が高い

0x08と0 を比較して, 等しくないので, 0 となる
↓
0 とbyte0 のビット演算をする.

カッコが必要
例）ビット演算（正）
if ((byte0 & 0X08) = = 0)

3.3 if文の書きかたのヒント

if文の判定文は障害を起こしやすい箇所

主な誤り

1. = =と!=を逆に書くことがある
2. &&や| |と &や | を混同する
3. 優先順序の勘違いで正しく判定されない

→カッコを付けたり, 字下げをして見やすくする

→判定をきちんと書く

→マクロを使う

例)

if (sub1( )) { ‥(処理1) }

if (sub1( ) != 0) { ‥‥(処理1) }

3.4 わかりやすいプログラム

1. わかりやすい変数名
   外部変数と内部変数の違いが変数名からわかる
   外部変数(グローバル)の変数は名前から使用目的がわかる
2. わかりやすいマクロ名 (#define)
3. わかりやすい関数名
   例) 動詞+名詞
4. 構造がわかるインデント(繰り返しや判断など)
5. 自然な判定式(not!の使用を制限する)
6. 一貫した書き方. 気まぐれはやめる.
7. わかりやすいコメント
   関数と外部変数には必須

まとめ

仕様変更に耐えられるプログラムとは
1.プログラムの作法が対応している.
2.プログラムの構造が対応している.

組み込みプログラムの場合
さらに, 仕様変更のありそうな部分を予測する能力が必要
ハードに関する知識も重要

ソフトウェアテストの概要

担当：西 康晴

1. SESSAMEの紹介およびコースの概要
2. 開発課題と失敗事例の解説
3. 組込み向け構造化分析の例
4. 組込み向け構造化分析・設計の概要
5. 組込み向け構造化設計
6. プログラミング
7. ソフトウェアテストの概要
8. 話題沸騰ポットに対するテストの実践
9. 大規模開発に向けての注意点

目次

- テストは大事
  - バグの影響はとんでもなく大きいのだ
  - テストが上手になるとバグの無い開発ができる
- テストの進め方
  - テストのフェーズとプロセス
  - テストに必要な視点
- テストの設計
  - 単体テスト・結合テスト・機能テスト・システムテスト
- テストしやすい開発をしよう
  - テストが上手になるとバグの無い開発ができる

バグの影響はとんでもなく大きいのだ

- 300万行のうち1行余計だったせいで、
  AT＆Tは11億ドルの損害
  - エラー回復コードがバイパス
  - 長距離電話が9時間に渡り話し中になった
- ブレーキシステムのバグで、GMにリコール？
  - 停車距離が15～20m伸びてしまった
  - 350万台がリコール、数百億ドルの損害
- ARIANE5ロケットは爆発
  - オーバーフローが原因
  - 4億ドルの損害

テストは簡単？

- テストなんて、仕様を確認するだけじゃないか
- ちゃんと作れば、テストなんていらない
- テスターは、力の無い奴にやらせればいい
- テスターはあら探しをするからムカつく

テストとデバッグは同じじゃないの？

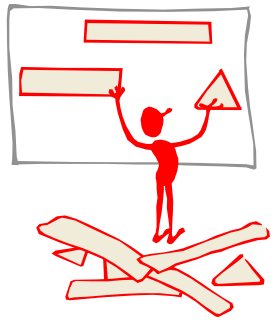
- デバッガ上で動かしてみるのはテストではない
  - バグを「いぶり出す」ために知恵を捻って設計すべし
  - 動きやすいテストをしてはダメよ
  - バグが見つかって始めてテストは成功なのだ

テストが上手になってくると
そもそもバグのない開発ができるようになる

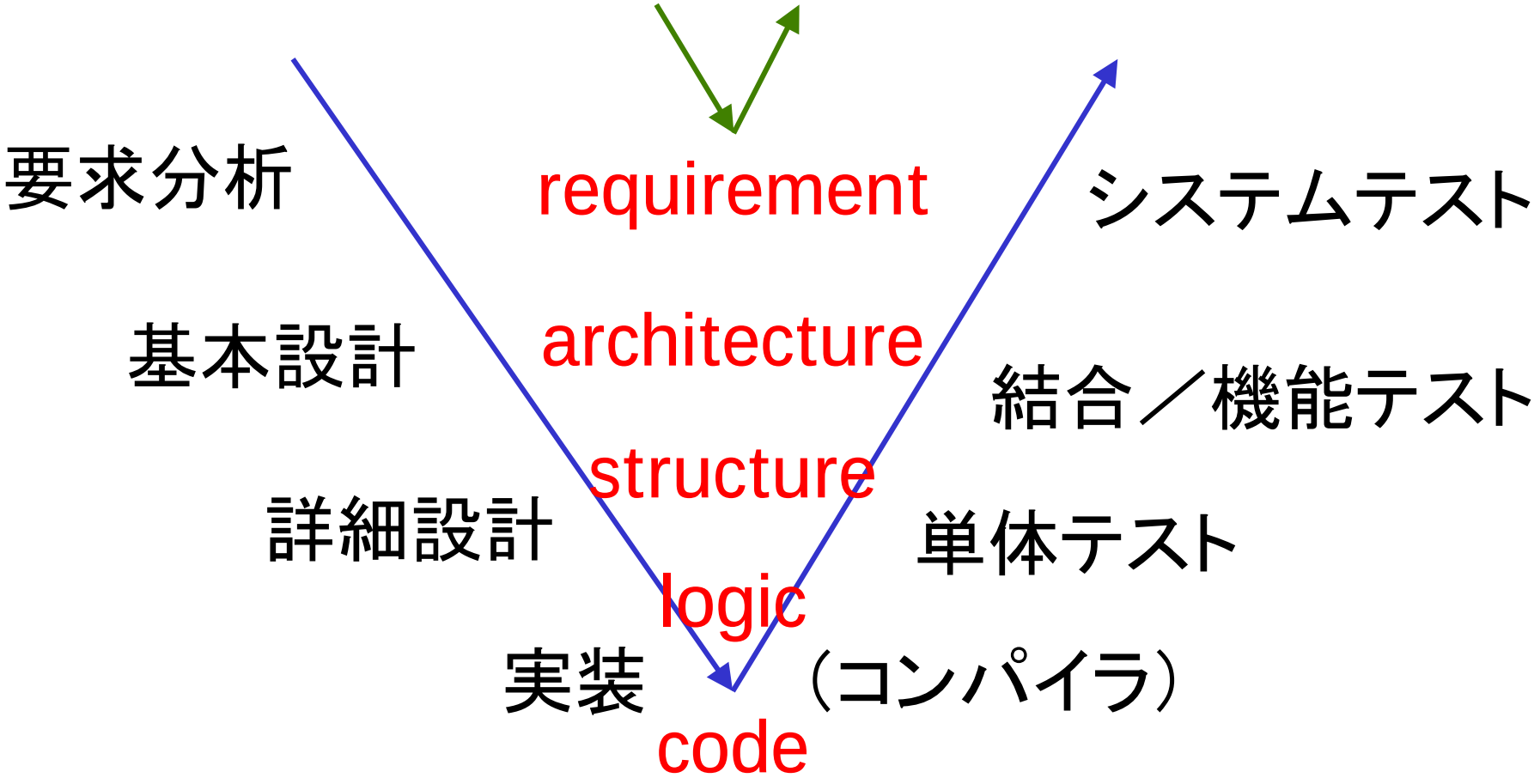
テストはどうやって進めればいいの？

- 組込みソフトのテストのフェーズ
  - 下位Vモデル
    - 単体テスト／結合テスト／機能テスト／システムテスト
  - 上位Vモデル
    - シミュレータテスト／実機テスト／シミュレータテスト
  - リグレッションテスト（回帰テスト）
    - 修正や変更の副作用で入り込んだ新たなバグも
      きちんと見つけて取り除こう

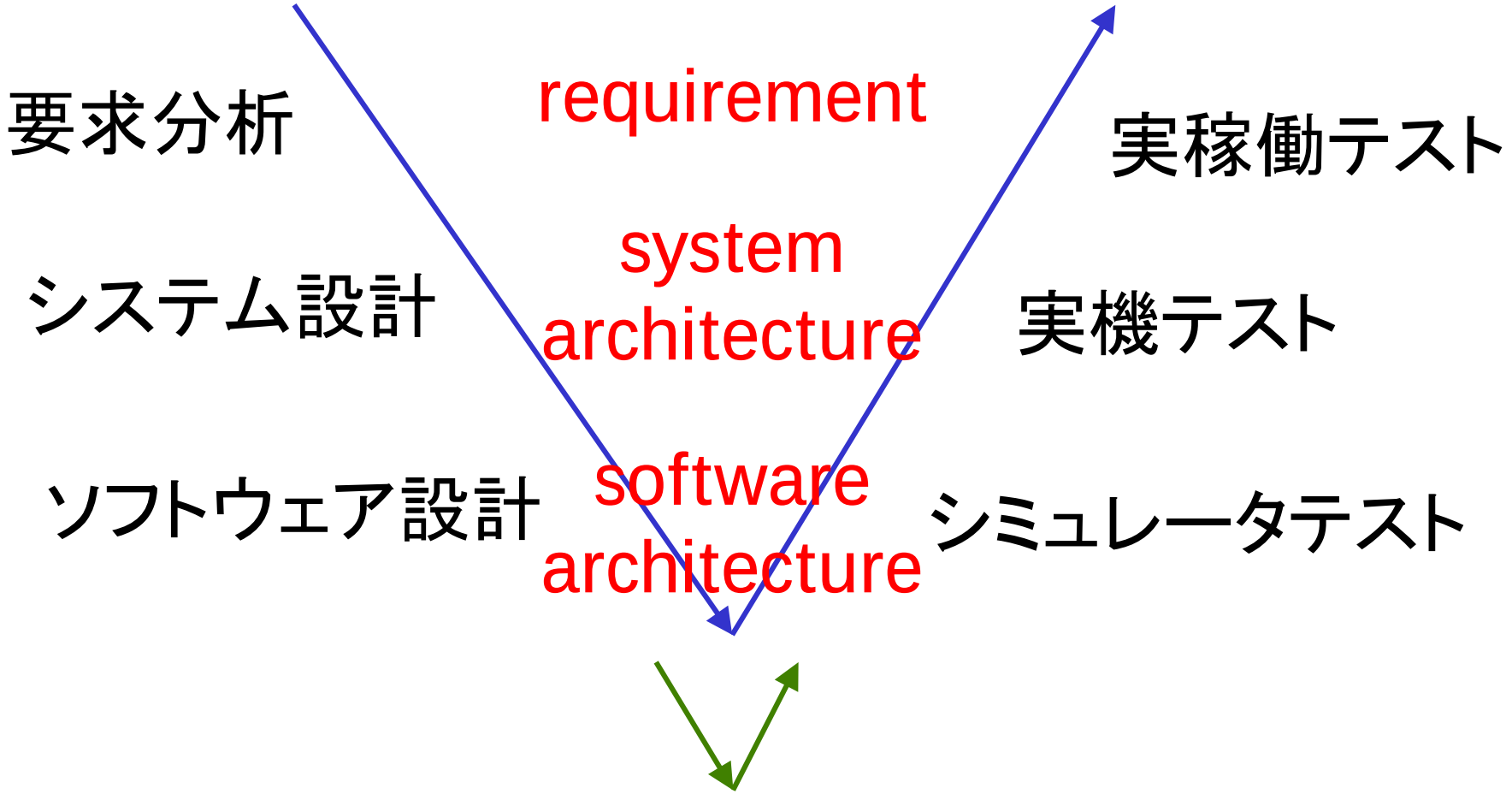
行き当たりばったりではなく
きちんと「設計」しよう



組込みソフトのテストのフェーズ：下位Vモデル



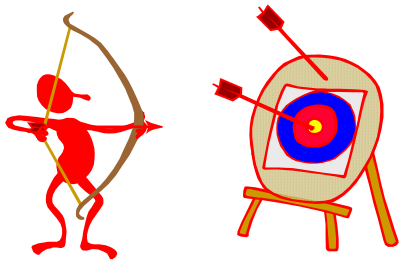
組込みソフトのテストのフェーズ：上位Vモデル



テストを設計する時には何を意識すればいいの？

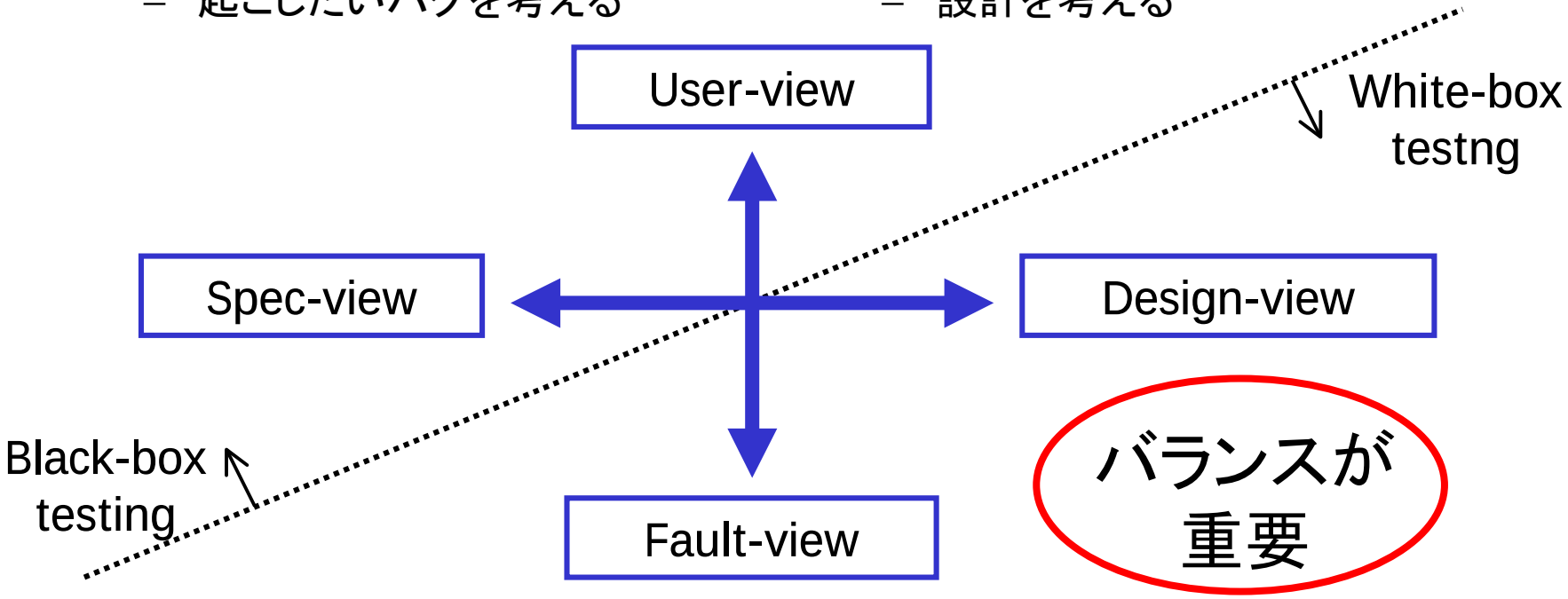
- 網羅
  - テスト漏れにはバグが潜んでいるかもしれない
    - どれくらい網羅したかを測る：カバレッジ
      - 制御パステスト
      - 機能網羅テスト

- ピンポイント
  - バグの多そうなところを狙う
    - 先人の知恵
      - 境界値テスト
      - ストレス系テスト
    - 過去の経験
      - バグの分析をして自分の弱いところを把握しよう



テスト設計に必要な視点

- User-view
  - ユーザが何をするかを考える
- Fault-view
  - 起こしたいバグを考える
- Spec-view
  - 仕様を考える
- Design-view
  - 設計を考える



モジュールをテストしよう：単体テスト

- どんなテスト？
  - 開発者が行うテスト
  - モジュールレベルのテスト
  - 詳細設計やプログラミングに着目したテスト

- どんな手法でテストするの？
  - 境界値テスト
    - モジュールの引数などに与えるテストデータを
      はじっこの値（境界値）にするテスト
  - 制御パステスト
    - モジュール内のロジックを全て通すように
      テストデータを与えるテスト

両方とも
必要！

境界値にはバグが多い

- 境界値や限界値の周りにはバグがたくさん潜んでいる
  - 条件文の間違い：どんな間違いが多い？

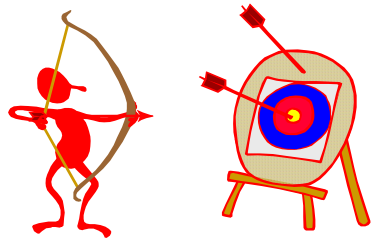
    例） ○：if (a＞0) then　　×：if (a≧0) then
    のように間違えることが多い。
    境界値であるa=0でテストすればバグが見つかる
    - if文などの条件文やwhile文などのループには、境界ずれや一つ違いのバグが多い
  - リソースのリミット：テスト文字列の長さは？

    例） 10バイトの固定長文字列を入力するプログラムの場合
    とても長い文字列を入力すれば落ちるかもしれない
    - 大きさを持ったデータ構造には、バッファオーバーフローなど容量の限界値ギリギリの処理を忘れているバグが多い

バグが多いところを狙おう：境界値テスト

- 同値クラスを見落とすと、ゴソッとテストがモレる
  - 境界値や限界値を抽出する範囲を「同値クラス」と呼ぶ
    - モジュールの引数、返値の境界値
    - if文の条件式など内部の境界値
    - テンポラリファイルの容量などI/Oの境界値
  - 正常境界値だけでなく異常境界値もテストする
  - 境界値を考えるのをサボっちゃダメよ

どんなときでも
境界値を考えよう



境界値テストの練習をしてみましょう

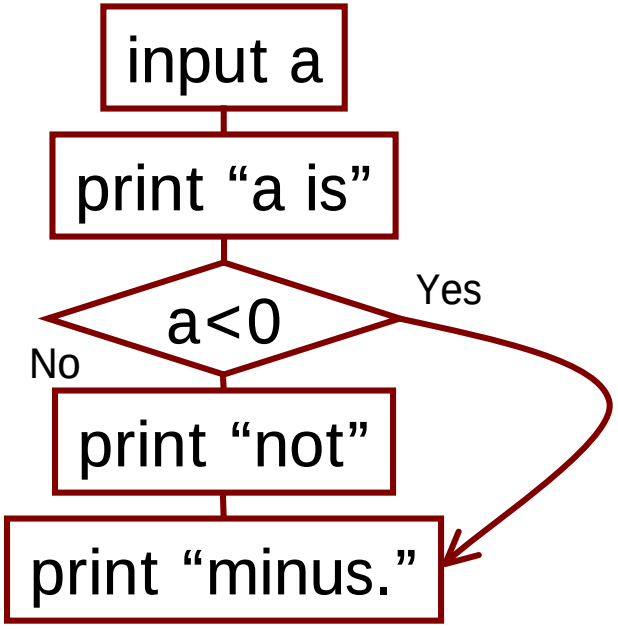
- 以下のモジュールのテストを設計してみましょう
  - int型の引数aがあります
  - モジュール内にはif (a<0) と if (7<a)という２つの条件文があります

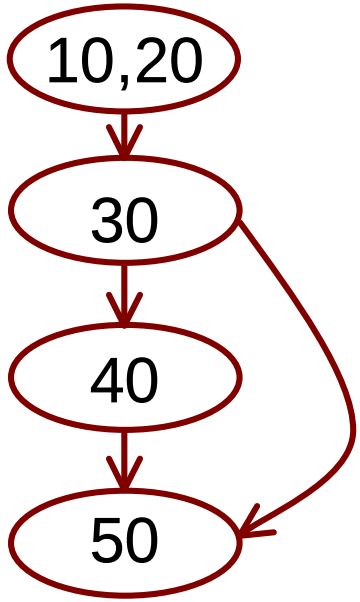
- まず同値クラスを挙げてみましょう。
- 次に同値クラスの境界値を挙げてみましょう
- 最後に、どんなバグが見つかるかを考えてみましょう

ロジックを網羅しよう：制御パステスト

- ロジックを網羅して全てきちんとモレなくテストしよう
  - フローグラフを描いてロジックを抜き出す
    - フローグラフ：ノードとリンクで書かれた図
    - フローチャートや状態遷移図はフローグラフ

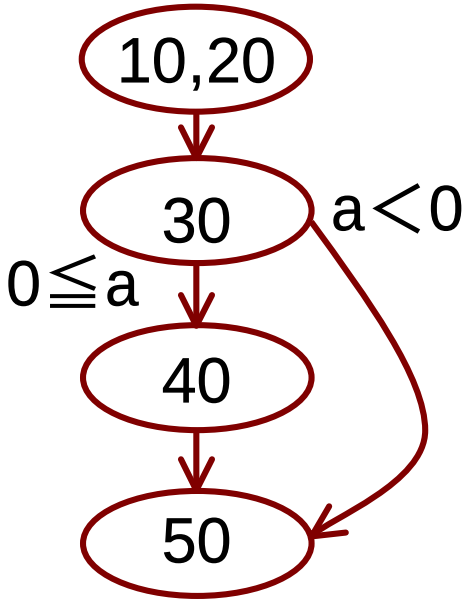
```
10  input a
20  print “a is”
30  if (a<0) goto 50
40  print “not”
50  print “minus.”
```





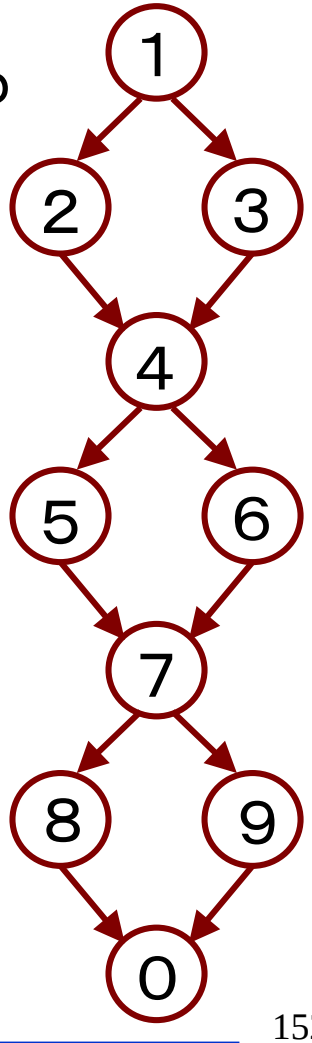
制御パステストの設計：パスの網羅

- パスとは？
  - フローグラフ上の経路
  - プログラムのロジック
  - パスの数がテストの数に比例する
- パスを全て網羅するように
  テストを設計する
  - まずパスの一覧表を作る
    - 10→20→30→40→50
    - 10→20→30→50
  - 次にパスを通るデータを実装する
    - 10→20→30→40→50　　: a=0
    - 10→20→30→50　　: a=-1



どこまでテストすればいいのかな？

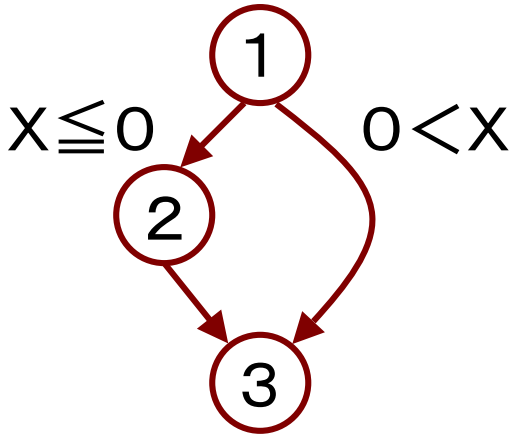
- 全てのパスを組み合わせようとすると膨大になる
  - if文の数の累乗で増えていく
  - if文に含まれるandやorの数の累乗でも増えていく
    - 右のグラフでは8本のパスが必要となる
- 最低でもif文の両側の分岐は1度テストしよう
  - 条件網羅（C1基準）
- 組み合わせが増えないように気を付けて開発する必要がある
  - KISS: Keep It Simple, Stupid!



参考：命令網羅（C0基準）

- どれだけ網羅できているか、を「カバレッジ」と呼ぶ
  - フローグラフのカバレッジ
  - 機能カバレッジ
- 全ての命令を一度実行すればいいような気がする
  - 命令網羅／ノード網羅／C0基準
- 右のフローグラフでは
  テストケースは1つでよい？

  A：1→2→3
  X＝0



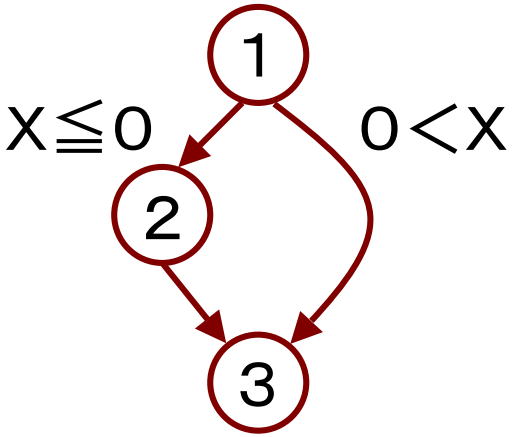
- 明らかにテストが足りない
  - if (9<x) とタイプミスしていても分からない

参考：条件網羅（C1基準）

- 少なくとも条件文のtrueとfalseは網羅しなくてはならない
  - 条件網羅／リンク網羅／C1基準
  - 右のフローグラフでは、
    テストケースは以下の2つになる

A：1→2→3
X＝0

B：1→3
X＝1



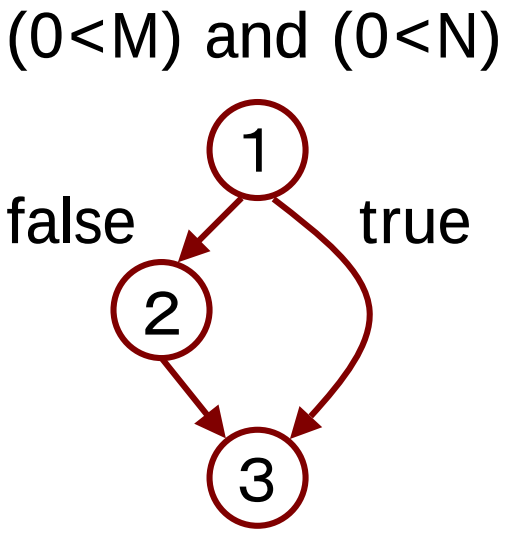
  - if (9<x) とタイプミスしていても分かる
    - 両方とも左に分岐するのでバグであると分かる

参考:複合条件網羅(C2基準)

- 組み合わせていない単一の条件式を
  すべて網羅しないとテストできたことにならない
  - 右のフローグラフでは、
    テストケースは以下の4つになる

A:1→2→3 (falseの場合)
(M=-1, N=-1)
(M= 1, N=-1)
(M=-1, N= 1)

B:1→3 (trueの場合)
(M= 1, N= 1)



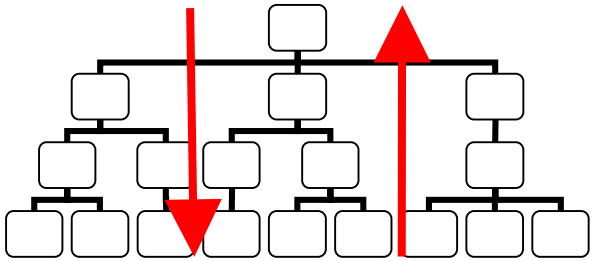
設計をテストしよう：結合テスト

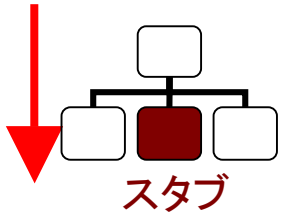
- どんなテスト？
  - 開発者が行うテスト
  - モジュールレベルのテスト
  - 概要設計や構造設計に着目したテスト
    - 状態遷移設計やモジュール間構造設計に着目したテスト

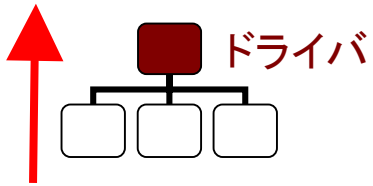
- どんな手法でテストするの？
  - トップダウンテスト／ビッグバンテスト
    - モジュール間の結合のミスを探すテスト
  - 状態遷移パステスト／状態遷移マトリクステスト
    - 状態遷移図や状態遷移マトリクスのミスを探すテスト
  - など

モジュール間のインターフェースのテスト：結合テスト

- モジュールをつなげる時のテスト
  - モジュールの結合順序を決める
  - ドライバやスタブを作成する
  - インターフェースをテストする
- トップダウンテスト
  - 上位のモジュールから先に作成し、結合していく
  - スタブが必要
- ボトムアップテスト
  - 下位のモジュールから先に作成し、結合していく
  - ドライバが必要
- ビッグバンテストは禁止である

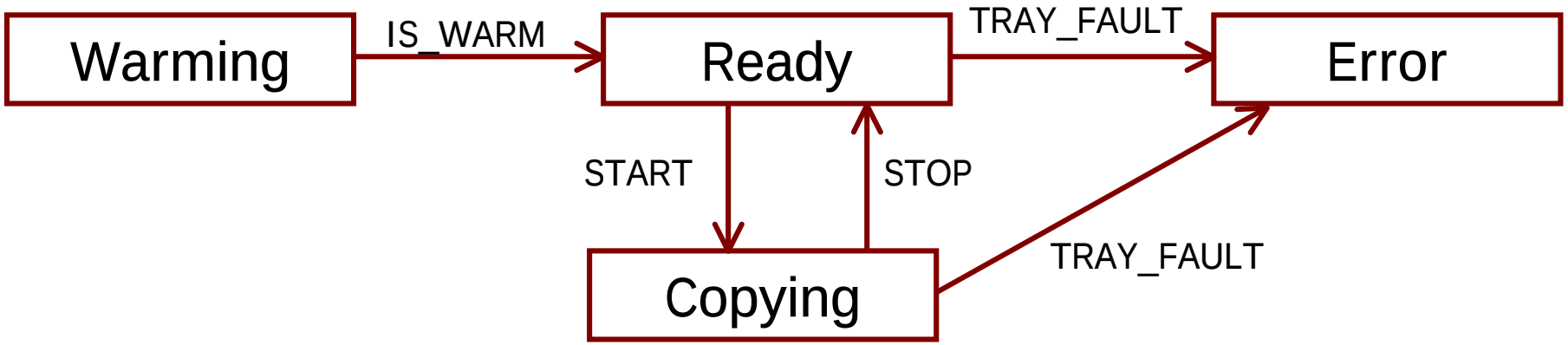


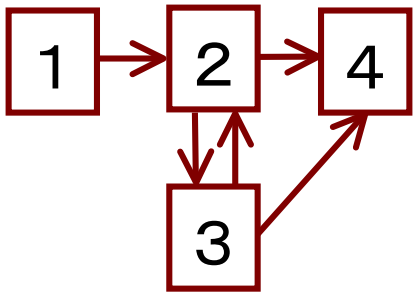




イベントを順番に起こそう：状態遷移パステスト

- イベントを順番に起こしてテストすることで
  状態遷移が正しく実現されているかをチェックしよう





パスの数は何本必要？

- 1→2→4
- 1→2→3→2→4
- 1→2→3→2→3→2→4
- 1→2→3→4
- 1→2→3→2→3→4
- 1→2→3→2→3→2→3→4

イベントと状態を網羅しよう：状態遷移マトリクステスト

- 状態遷移マトリクスをモレなくテストすることで
  状態遷移が正しく実現されているかをチェックしよう

	S / E	起動処理中 (Warming)	スタンバイ中 （Ready）		
		0	1		
スタンバイ完了 (IS_WARM)	0	→Ready	×	×	×
コピーボタン押下 (START)	1	／	→Copying	?	／
コピー完了 (STOP)	2	×	×	→Ready	×
エラー発生 (TRAY_FAULT)	3	?	→Error	→Error	×

何が起こるか分からない
↓
バグの可能性が高い

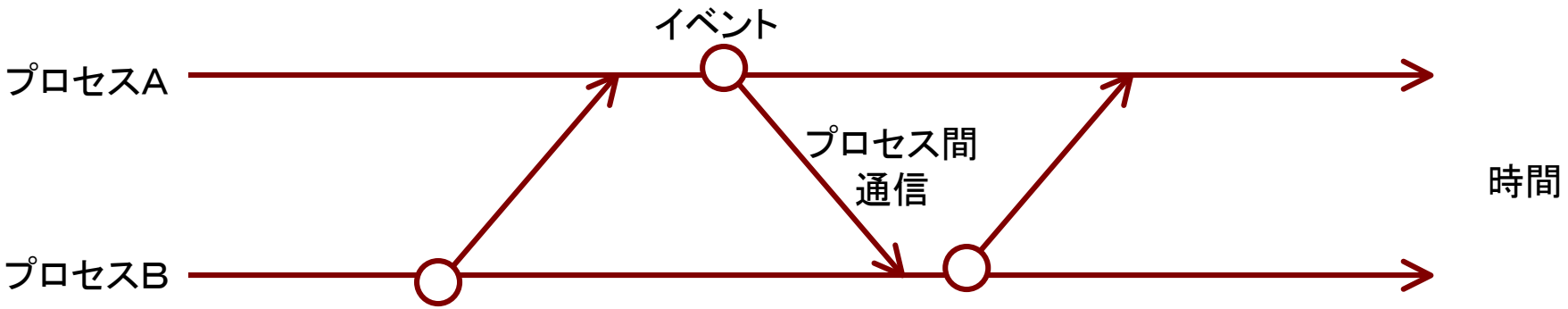
→：遷移
／：無視
×：ありえない

– 状態遷移マトリクステストを設計すると、
  矛盾や抜けのある遷移もレビューできる

状態遷移テストの注意点

- 状態遷移パステストと状態遷移表テストは使い分ける
  - リンク網羅ならばカバレッジとしては等価
  - 状態遷移パステストは、ユーザの操作などと対応させてテストしやすい
  - 状態遷移表テストは、異常なイベントへの対応が網羅的にテストできる
- 状態遷移図そのものの間違いに気を付ける
  - 仕様書をよく読んで検討する
    - 状態や遷移のモレ／あいまいな遷移条件
- 時間に関する状態には気を付ける
  - ある一定時間だけ初期化処理の状態になる、など
  - その状態でいられる最小時間と最大時間を明確にする
- モデルと実装の差異に気を付ける
  - モデル上では瞬時に遷移することになっているが、実際にはほんの少しでも時間がかかるため、ありえないイベントが発生する

パステストの応用：シーケンス図のテスト



- C0基準：イベント網羅
  - 一つ一つのイベントがちゃんと発生するか
- C1基準：イベント間通信網羅
  - 一つ一つのプロセス間通信がちゃんと受け取れるか

- イベント発生可能性時刻の同定
  - 境界値テストで明らかにする
- フローグラフで表現できる設計は全てパステストが可能

コンパイル後に改めてテストしよう：機能テスト

- どんなテスト？
  - 開発者だけでなく、テスト担当者や品質保証部門も行うテスト
  - 組み上がったソフトウェアレベルのテスト
  - 機能仕様に着目したテスト
- どんな手法でテストするの？
  - 機能網羅テスト
    - 機能一覧を作って網羅するテスト
    - 簡単そうだが意外にやらない
  - 境界値テスト
    - 機能のパラメータに着目した境界値テスト
    - 仕様のバグが見つかる

機能網羅テスト

- ソフトウェアの持つ機能を漏れなくテストする
  - 表を作って機能を全て書き、OKになったらチェックする
    - 基本中の基本だが面倒がって行わないことが多い

機能	*チェック欄*
機能1	レ
機能2	
機能3	レ
機能4	
機能5	レ
機能6	

マトリクス網羅テスト

- 2次元の組み合わせを網羅する場合のテスト
  - （機能×データ）、（状態×イベント）など
    - 行と列の組み合わせが網羅されるかをチェック

	条件A	条件B	条件C	条件D
条件1	レ			
条件2			レ	
条件3	レ			
条件4				レ
条件5		レ		
条件6				

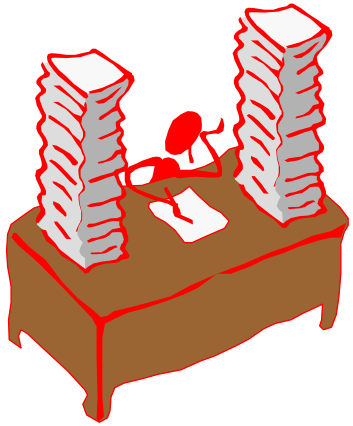
    - 「ありうる組み合わせ」が網羅されるかをチェック

	条件A	条件B	条件C	条件D
条件1	○ レ			
条件2	○	○	○ レ	
条件3	○	○ レ		
条件4	○	○	○ レ	○
条件5		○	○	○ レ
条件6				○

境界値とバグ

- 境界値や限界値の周りにはバグがたくさん潜んでいる
- バグが多いところを狙ってテストしよう：境界値テスト
- 仕様の境界値をどんどんテストしよう
  - 境界値を考えると、仕様があいまいなことに気付く
    - ４月１日生まれは早生まれ？遅生まれ？
      - 民法第１４３条：満年齢は起算日に応当する前日をもって満了する
      - 学校教育法第２２条：満６歳に達した日の翌日以降における最初の学年

境界値テストは
なるべく上流で設計して
仕様のあいまいな部分の
レビューに使おう



色々いじめてみよう:システムテスト

- どんなテスト？
  - 開発者だけでなく、テスト担当者や品質保証部門も行うテスト
  - ソフトウェアを使うユーザレベルのテスト
  - 要求仕様に着目したテスト
- どんな手法でテストするの？
  - ストレス系のシステムテスト
    - ソフトウェアに負荷をかけるテスト
    - よくバグが見つかるので手抜きしないでテストする
  - 環境系のシステムテスト
    - 相性や互換性の問題を見つけるテスト
    - データだけでなく、電気や熱の流れなどにも気を付ける
  - 評価系のシステムテスト
    - どれくらい堅牢か、どれくらいユーザが満足するかの評価
    - ユーザに使わせるだけでなく、きちんと考えて設計する

参考：ストレス系のシステムテスト

- ボリュームテスト
  - 大きなデータ、たくさんのデータを与えるテスト
    - 巨大な表のあるページで某Webブラウザがクラッシュ
- ストレージテスト
  - ディスクやメモリなどを残り少ない状態で使うテスト
    - メモリ12Mで動かしたら某OSがブルーサンダー
- 高頻度テスト
  - 短時間にたくさん処理させたり同時に処理させるテスト
    - ある種のパケットを大量に投げると某Webサイトが停止
- ロングランテスト
  - 長時間実行させるテスト
    - 某ターミナルエミュレータは長時間の使用でブルーサンダー

参考：環境系のシステムテスト

- 構成テスト
  - 一緒に利用している他のソフトやハード「から」悪影響を与えられる問題を検出するテスト
    - 某ビデオドライバをインストールするとテスト対象であるWindowsが起動しなくなる
- 両立性テスト
  - 一緒に利用している他のソフトやハード「に」悪影響を与える問題を検出するテスト
    - テスト対象である某Webブラウザをインストールすると某ワープロが起動しなくなる
- 互換性テスト
  - 他のソフトやハードとデータなどの交換をさせるテスト
    - 某ワープロで枠を作ると、異なるバージョンでは位置ズレを起こす

参考：評価系のシステムテスト

- 障害対応性テスト
  - 電源を抜くなどの障害を起こして復旧性などを評価するためのテスト
    - 某PCは電源を抜いてスリープさせるとHDDがクラッシュ
- セキュリティテスト
  - 機密保護などの穴を突いてセキュリティを評価するためのテスト
    - 某ウィルスチェッカは、特定のURLで管理機能を奪取できる
- ユーザビリティテスト
  - 操作性や視認性などを評価するためのテスト
    - 某統計ソフトのインストーラは、OKとCancelが逆の場合があり、インストール不能だとユーザに判断されてしまう
- ユーザ操作テスト
  - ユーザが満足しているかを評価するためのテスト

Testability の高い開発を意識しよう

- テストが上手になると、そもそもバグのない開発ができるようになる
  - テストを設計すると、構造を見直したり曖昧なところを明確にすることになるので、気付かなかったバグに気付くようになる
  - テスト設計が簡単なプログラムは、設計や実装もシンプルなので、バグが入り込みにくくなる
  - テストが実施しやすいプログラムは、実施できるテストの量も増えるので、バグが見つかりやすい
- テストが容易な製品設計を、テスト容易化設計と呼ぶ
  - Testability DesignやDFT (Design For Test)とも呼ぶ
  - プログラムの内部に依存関係があると、その分だけ組み合わせが必要になるので、きちんとモジュール化を行う
  - プログラムの出力や内部状態の変化が分からないとバグを見逃してしまうので、なるべく簡単に分かるようにしておく

まとめ

- テストをする時に意識すべきこと
  - 網羅：モレなくテストする
  - ピンポイント：バグが多いところを狙ってテストする
- テストとは
  - 知恵を絞って設計すべし
  - バグが見つかって初めてテストは成功なのだ
  - いろいろなテスト手法がある
- テストが上手になってくると、
  そもそもバグのない開発ができるようになる

テストを意識して開発することで
はじめからバグの無いソフトを作ろう

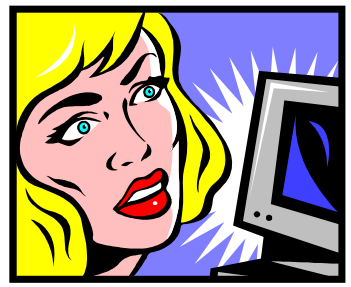
話題沸騰ポットに対するテストの実践

担当：日立ソフトサービス　大野　晋

1. SESSAMEの紹介およびコースの概要
2. 開発課題と失敗事例の解説
3. 組込み向け構造化分析の例
4. 組込み向け構造化分析・設計の概要
5. 組込み向け構造化設計
6. プログラミング
7. ソフトウェアテストの概要
8. 話題沸騰ポットに対するテストの実践
9. 大規模開発に向けての注意点

テストは難しい

- 通常、プログラム開発の工数の半分以上をテストに費やしている
- しかもその大半をテストの実施ではなく、デバッグに費やす
- テスト時間のほとんどは様々な原因のテストのやり直し
- 体系的にテストを計画し、実践することで工数の大半を削減できる
- しかし、実際には行き当たりばったりにテストが実施され、時間が浪費される



立場によってもテストの内容が違ってくる

- プログラム作成者のテスト：意図した通りかどうかの確認
- テスターのテスト：バグのたたき出し
- 品質保証（QA）のテスト：品質の確認、完成の最終確認

プログラム作成者のテスト

設計者としてできたものの動作を保証するためにしていたテストは。

（1）共用ルーチン、マクロなど共有物（部品）のテスト

（2）モジュールレベルの単体テスト（C0,C1：100%はマナー！）

（3）モジュールを組みたてた組み合わせテスト（機能仕様書、詳細仕様書レベル）

（4）全部組み立てた機能テスト（機能の動作確認）

（5）パラメタを組み合わせた組み合わせ条件のテスト

（6）境界条件テスト

（7）エラーケースのテスト

（8）限界テスト

（9）他のプログラムなどとの組み合わせテスト

（10）使用条件を考えたユーザイメージの機能評価

テスターのテスト

デバッグを目的としたテスト
（1）全体の機能に対する機能テスト
（2）パラメタを組み合わせた組み合わせ条件のテスト
（3）境界条件テスト
（4）エラーケースのテスト
（5）限界テスト
（6）他のプログラムなどとの組み合わせテスト
（7）使用条件を考えたユーザイメージの機能評価

品質保証（QA）としてのテスト

限られた時間内でプログラムの品質を保証するための検査
（1）主力機能をサンプリングした機能テスト（機能の確認）
（2）パラメタを組み合わせた組み合わせ条件のテスト
（3）境界条件テスト
（4）エラーケースのテスト
（5）限界テスト
（6）他のプログラムなどとの組み合わせテスト
（7）使用条件を考えたユーザイメージの機能評価
で、特に（3）（4）（5）（6）（7）は大切。
しかし、本当に大切なのは、バグが出た場合の要因分析！

さて、テストを始めよう：テストの準備

・まず、計画（戦略）を立てる

・そして、テストの準備をする
　⇒例えば、ポットのテストにはこんなものが必要！

1. ポットのシミュレータ
2. ポットへの温度計設置
3. センサの作動状態がわかるような仕掛け
4. ポットのデバッグ環境

テストケースを作る（1）

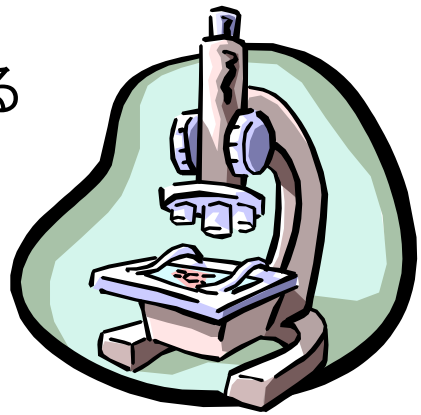
- 機能の一覧を作成する
- それぞれの機能の同値クラス分析と境界値分析を行う

同値クラスとは…

「2つのテストを実行して同じ結果を期待するとき、2つは同値であるという。」（Kaner他）

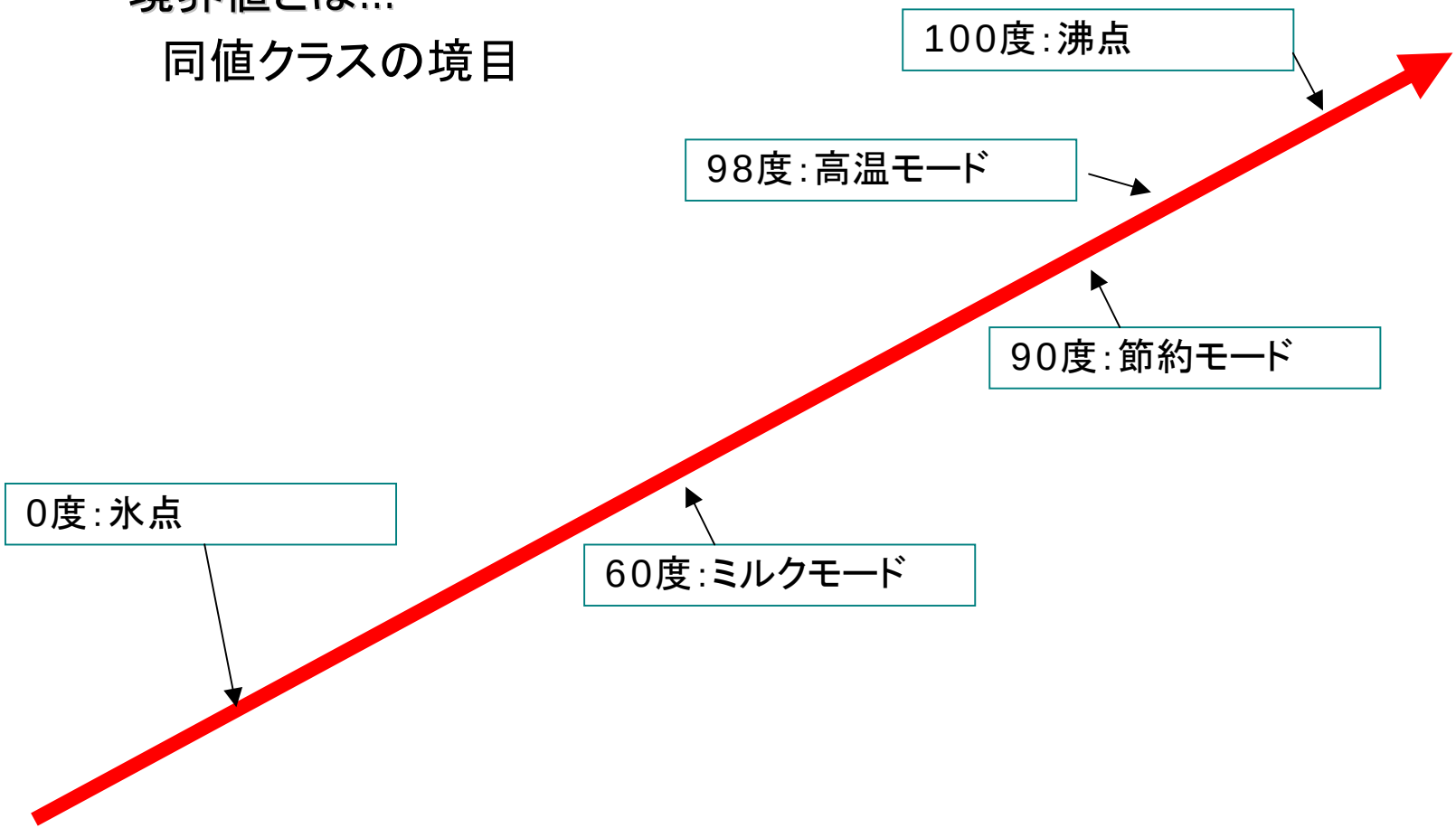
例えば、温度制御のテストを考える場合

・水温：10度、20度、36度、50度は同値クラスになる

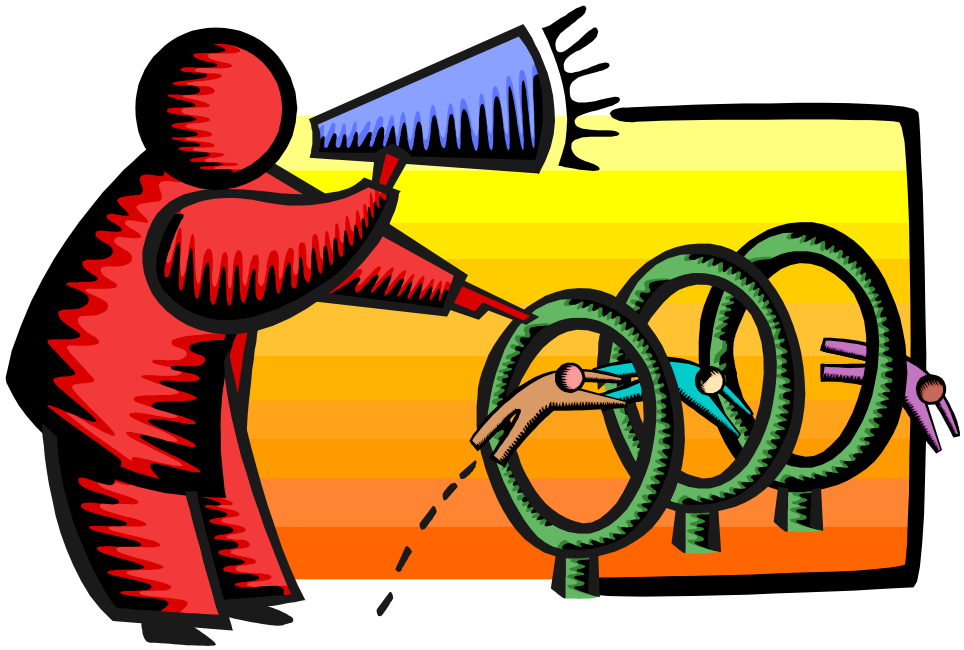


テストケースを作る（2）

境界値とは…
　同値クラスの境目



さあ、各モードのテストケースを挙げてみよう



テストケースを作る（3）

- 保温モード：高温
  - 100度
  - 98度
  - 95度
  - 80度
  - 0度
- 保温モード：節約
  - 100度
  - 98度
  - 90度
  - 80度
  - 0度
- 保温モード：ミルク
  - 100度
  - 98度
  - 60度
  - 30度
  - 0度

ほんの1例。

どこまで、確認するかでリストアップする温度も増えていく！

テストケースを作る（４）

テストケースを作るには

- 機能を同値分析、境界分析する
- 抽出した機能のテスト値を組み合わせてテストケースを作成する
- 実験計画法の直交表を用いることで、テストケースを最適化できることもある

テストケースを作る（５）

機能から視点を変えてテストケースを考える

- エラー処理
- 性能
- 割り込み
- I／O
- 温度
- スイッチの種類とタイミング
- 過去の失敗

など

これらを同値分析、境界値分析などを行ってテストケースを作成する

テストを測る

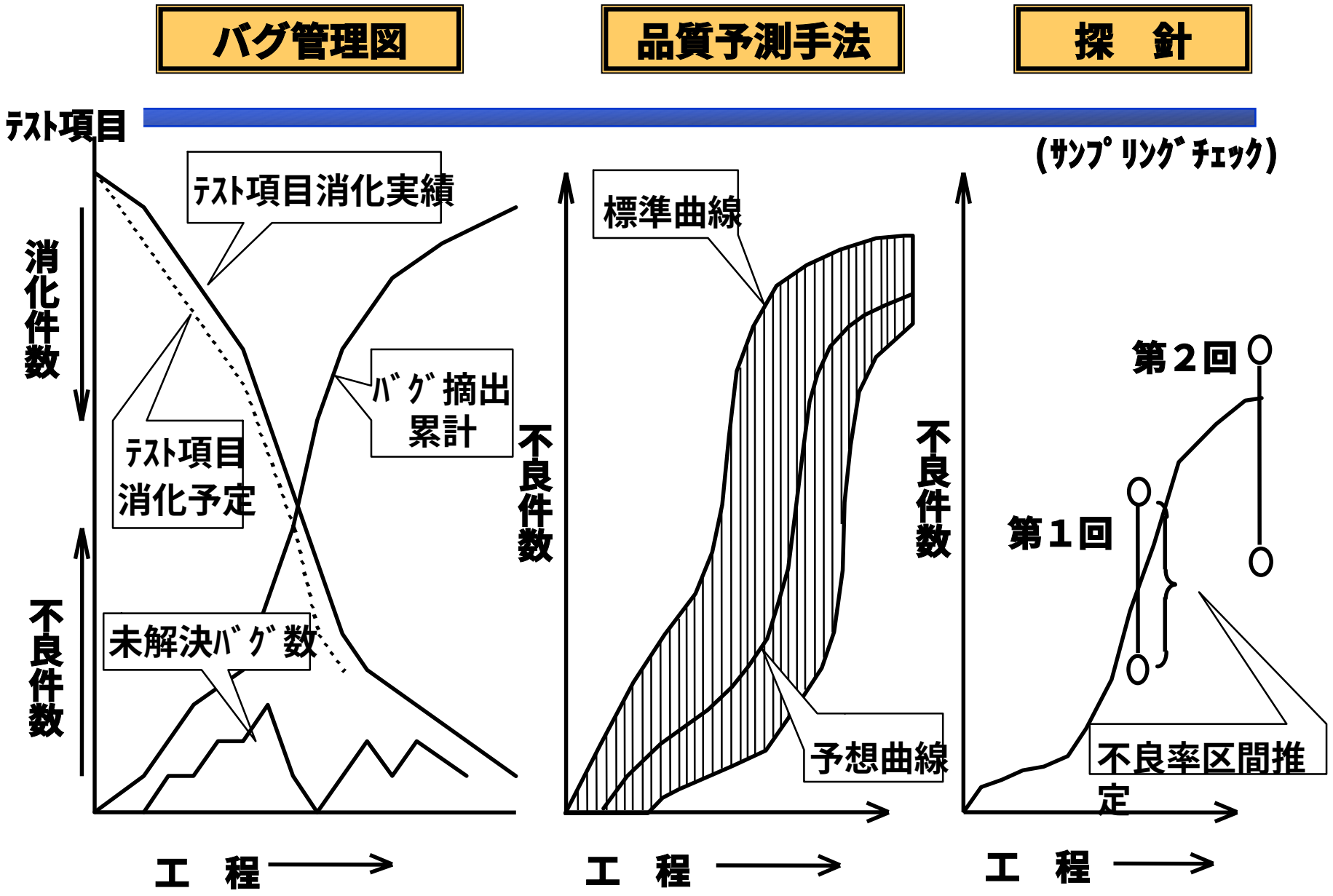
- テストの結果をまとめることでいろいろなことが見えてくる
- テストのまとめ方には

✧ バグを1件ずつ分析するためのレポートを書く

✧ レポートの一覧をもとにバグの傾向を分析する

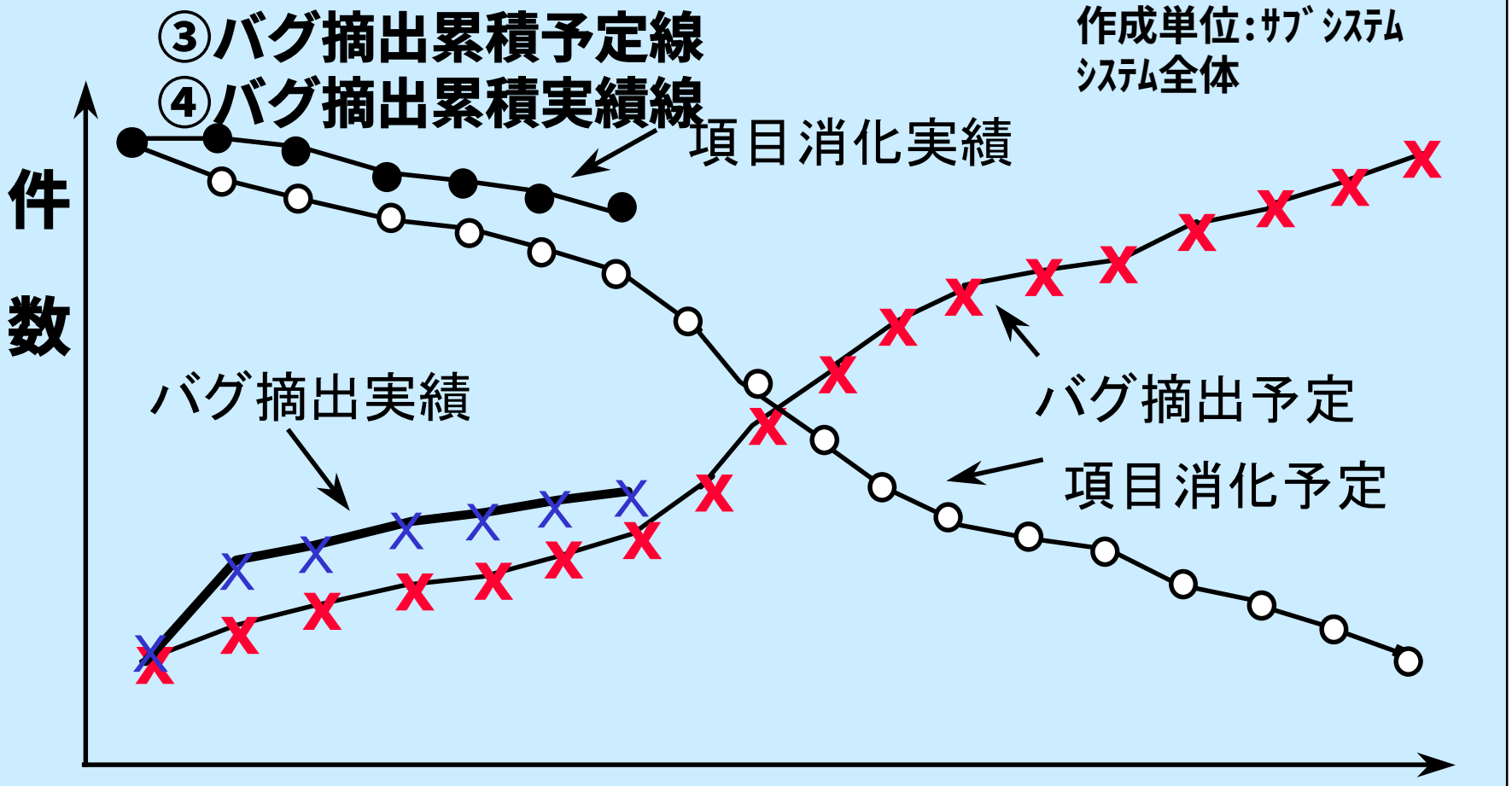
✧ バグの摘出とテストの進捗からプログラムの品質を分析する

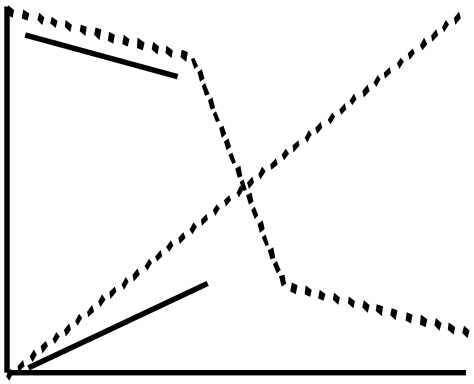
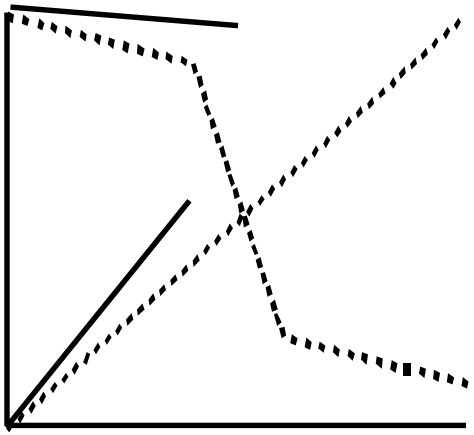
⇒次にテストの進捗傾向の見方の例を示そう



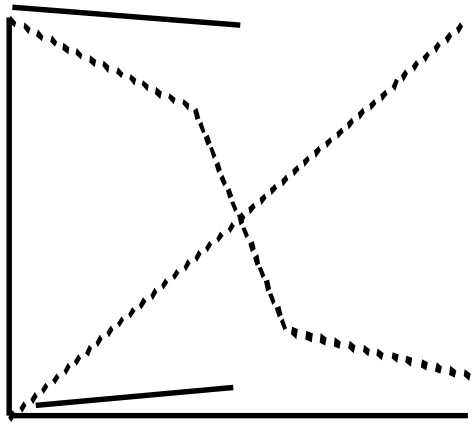
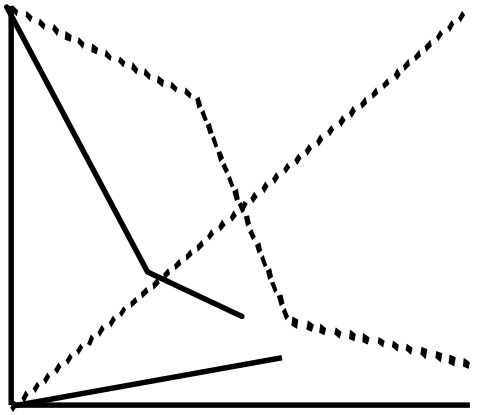
バグ管理図とは、次の４つの線で予実績を示す。

①テスト項目の消化予定線
②テスト項目の消化実績線
③バグ摘出累積予定線
④バグ摘出累積実績線



項番	組合せ		バグ管理図	考えられる問題要因
	項目消化	バグ摘出		
1	Ｐ１ 正常	Ｂ１ 正常		特になし ＊架空の報告の可能性あり
2	Ｐ２ 消化停滞	Ｂ３ バグ多発		①バグの作りこみ量が多い ②仕様変更、中途追加でバグを作り込み

バグ管理図の見方（その１）

項番	組合せ		バグ管理図	考えられる問題要因
	項目消化	バグ摘出		
3	Ｐ２ 消化 停滞	Ｂ２ 摘出 不足		①デバッグ／テスト環境の不備 ②開発要員不足
4	Ｐ３ 急激 消化	Ｂ２ 摘出 不足		①テスト項目の質が低い ②テストの実施が一部の機能に偏っている。 ③品質が良い（？）

バグ管理図の見方（その２）

テスターの視点（1）

普通のテストはこのように進んでいく。
⇒ただ、バグを良く見つける人と見つけられない人がいる

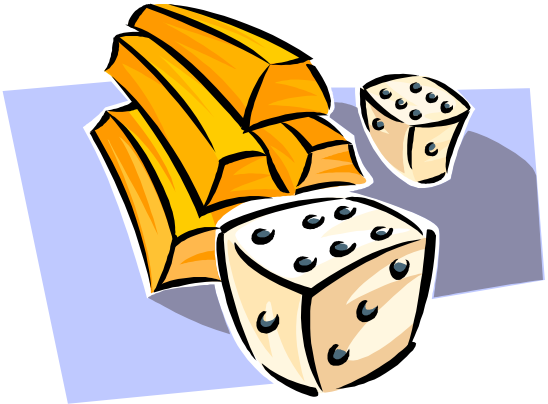
- 良いテストができるひと
  - 気のつく人？
  - 気配りのできる人？
  - よく仕様を知っている人？
  - 品質保証部の人？
  - お客さん？
  - 心配性の人？

テスターの視点（2）

単純に「目に見える仕様」をテストしてもバグはなかなか見つからない！

⇒例えば、「話題沸騰ポット」をテストするとしたら、どのようなポイントをテストするだろうか？

仕様書からこんな点をテストしたいと思うポイントを挙げてみよう！



テスターの視点（3）

（1）気圧が低く、沸点が下がっている場合
（2）高温の水を沸かす
（3）低温の水（氷）を沸かす
（4）水量センサーの誤作動：空瓶センサーがoff
（5）センサー面での水のゆれ
（6）蓋を開けるタイミング：ヒーターの切り替えと蓋の開け閉め
（7）加熱中の水量変化
（8）水以外の液体
（9）外気温の影響
（10）ボタンの同時押し
（11）周波数による違い
⇒本来、考えられるべき事項（文章にならない仕様）
⇒良いテストとは、文章以外の仕様を見つけ、それを確かめること
⇒特に繊細な制御をするプログラムの場合、制御に抜けが生じやすい

テスターの視点（4）

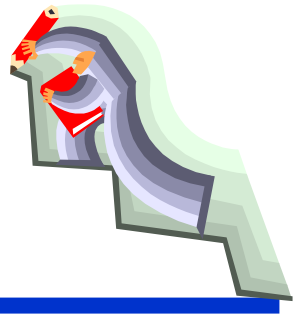
- 過去のバグを見直すことで、考えておくべき視点を知ることができる
- バグレポートは知識の宝庫！
- 蓄積がない場合には、書籍の付録のバグ一覧が有効！
- 「書いてある仕様」でプログラムが間違っていることは本当に少ない！
- 「書いていない仕様」で問題を抱えていることは多いが、バグなのか、その通りで良いのか、の判断は難しい！

まとめ

- ソフトウェアのテストのやり方は単一のパターンではなく、目的によってやり方が異なる
- テストの視点とは、設計者と同じ視点である

テスターの言い分：

- 良いテスターの視点を設計の早い時期に役立てることで効率よく、品質の良いプログラムが出来上がる
- だから、テストでバグ出しをするよりもレビューや机上見直しが効率的！



大規模開発に向けての注意点

担当：(株)豆蔵　福富三雄

1. SESSAMEの紹介およびコースの概要
2. 開発課題と失敗事例の解説
3. 組込み向け構造化分析の例
4. 組込み向け構造化分析・設計の概要
5. 組込み向け構造化設計
6. プログラミング
7. ソフトウェアテストの概要
8. 話題沸騰ポットに対するテストの実践
9. 大規模開発に向けての注意点

アジェンタ

- コースの目的
- 組み込みソフトウェアの現状
- 大規模開発の状況
- 問題点
- 原因
- 解決方法
- 開発者へのメッセージ
- 最後に

コースの目的

組込ソフトウェア開発の初級技術者の皆さんに、

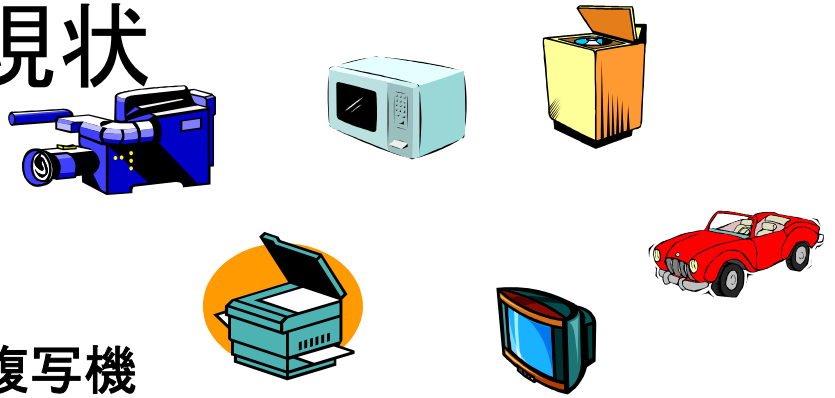
◆大規模開発におけるソフトウェア開発の問題点、解決策を理解して頂く
◆今後、大規模開発に参加する時に何の技術を習得すれば良いかヒントを得る

組み込みソフトウェアの現状

今、いったいどうなっているの？

組み込みソフトウェアの現状

我々の身の回りの製品には
組み込みシステムが多く使われている



携帯電話、PDA、自動車、ビデオ、テレビ、複写機
FAX、ゲーム機器、エアコン、電子レンジ、洗濯機
カーナビ、デジタルカメラ、心臓ペースメーカ.....

情報家電、ユビキタス社会
（どこでもコンピュータ）

システムのネットワーク化

ユーザニーズの多様化

多品種、
高機能化、
製品サイクル
短期化

組込みソフトウェアの
需要が飛躍的に増加

大規模開発の状況

今、どんな状態になっているの？

大規模開発の状況

大規模開発例

- カーナビゲーション開発
- 携帯電話開発
- PDA開発
- デジタルテレビ開発
- カラー複写機開発

PC上のソフトウェアを開発するのと同等の規模になってきている

ソフトウェア開発者は
１００人以上が普通に

大規模開発の状況

ユビキタス社会、
情報家電に向けて

ハード制御中心 ⇒ GUI / 画像処理 / ネットワーク

ソフトウェアの制御対象が変化

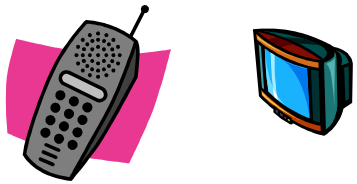
ソフトウェアが製品の中心になってきている

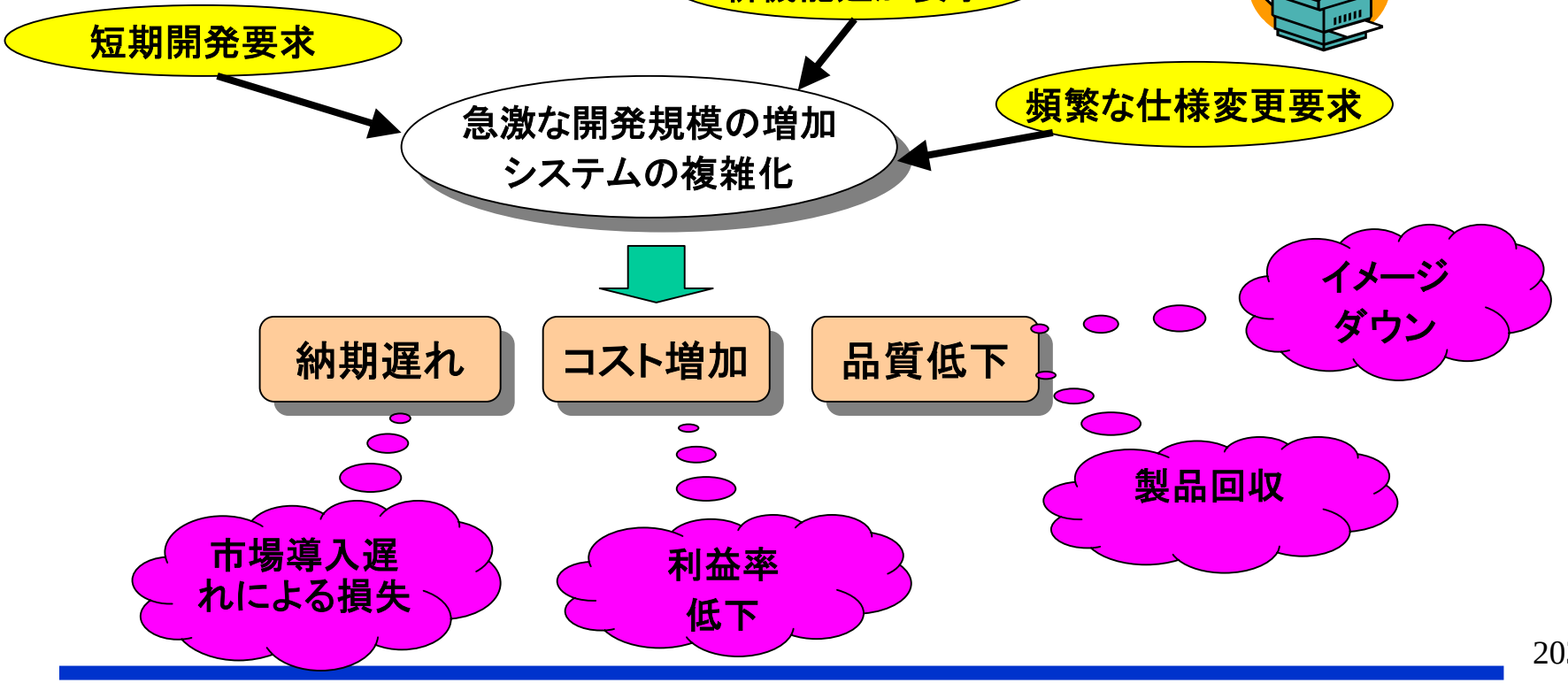
年々開発規模は
増大し続けている

必然的にプロジェクトに
多くの開発者が参加

大規模開発の状況

組み込みシステム開発の課題





大規模開発の例（携帯電話開発）

携帯電話への機能追加要求

自動車電話 ⇒ ショルダーホーン ⇒ デジタル方式 ⇒ 留守番サービス

⇒ メール ⇒ WEB JAVAアプリ ⇒ デジタルカメラ ⇒ 動画 GPS

開発期間短縮

メーカは、ドコモ製品だけでも年2回新機種投入

大規模開発の例（携帯電話開発）

複数機種対応

DOCOMO（FOMA）
DOCOMO
au
J-PHONE
Tu-Ka

→

504
211
651
302

多品種開発の要求

大規模開発の例（携帯電話開発）

短期開発、高機能化・高付加価値化、仕様変更の結果

開発規模、検証項目は増えているが，開発期間が
短く、複数機種を投入しなくればならない

品質を確保できない

検証項目 ⇒ 2～3倍増加

iモード以前　3万～5万件
現在　10万件

ソフト規模 ⇒ 3～4倍増加

ソフトバグによる端末回収騒ぎ

503i　数十万台回収

顧客の信頼失う

企業に大きな損害を与える

問題点

今、どんな問題があるのでしょう？

大規模開発の問題点

大規模開発と小規模開発の差は？

多数の技術者で開発
開発拠点が複数
協力会社の活用
システムの複雑化

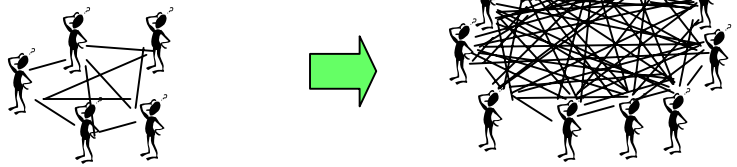
→

個人から
組織による開発へ

小規模ならみんなの頑張りがあれば
成功できた（エースがいれば乗り切れた）

人が多いためコミュニケーション、情報共有が難しい

開発の依存関係が多い



大規模開発の問題点（マネージメント）

ソフトウェア開発が可視化できていない
（見えないものは制御できない）

（ハードウェア開発と異なる）

◆どのような計画で実施しているか分からない

（利害関係者と合意していない、非現実的な計画、勘による見積り、組織体制が不明確）

プロジェクト計画が不十分で
失敗する事例が多い

プロジェクトの成否の
80%は計画で決まる
と言われている

よく聞く話

→「プロジェクト計画は作っても直ぐ変更になるから意味がない」

「時間がないから作らない」

しかし、大きなプロジェクトでは、何がおきるかわからない

リスクがいたるところで待っている

充分準備していても、後で不十分であったことを知らされる時がある

大規模開発の問題点（マネージメント）

計画不十分で発生した問題

（例）

- ✓必要な時期に成果物が入手できなく、作業待ちになった
- ✓明日までにソフトウェアが急に欲しいと言われ徹夜した
- ✓予定していた開発者が別プロジェクトに取られた
- ✓開発を開始したら、考えていた技術より難しかった
- ✓規模が予定より大きく、残業、休出が続いた
- ✓先週までは予定通りの進捗と思っていたら、突然3ヶ月も遅れることがわかった

計画不十分で発生した問題は何かありますか？

二日目のセミナーで考えましょう

大規模開発の問題点（マネージメント）

◆何を開発しようとしているのか分からない（要件管理）

（検討されない要件、複雑な仕様、文書化されない要件、承認無しの変更、読み難い仕様書）

要求の変化、要求があいまいで、要件数が大きいため管理が難しい

◆プロジェクトがどのような状況なのか把握できない（進捗管理）

（把握する対象が不明確、リリース間際で大きな問題が発見される、是正措置が取れない）

管理対象が多く、開発拠点の違い、また協力会社管理があるため難しい

大規模開発の問題点（マネージメント）

◆協力会社にまかせっきり（協力会社管理）

（依頼する作業が不明確、適切な選定が出来ていない、定期的な進捗が確認出来ていない、品質保証体制がない、ノウハウが流出する）

コストの安い会社へ依頼しなければならない

◆作業成果物がどのようになっているかわからない（変更管理、構成管理）

（どれを変更して良いかわからない、いつどこを変更したかわからない、
どれを使用するかわからない、どれに影響するかわからない、
どのバージョンのものが組み込まれているのかわからない）

対象成果物、開発者が多く、開発拠点が違うため管理が難しい

大規模開発の問題点（マネージメント）

◆予測した問題を踏まえて開発していない（リスク管理）

（思いがけない予定変更、技術変更、予算変更に対応できない）

規模が大きいため問題
が発生した時の影響大

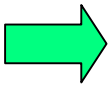
◆コミュニケーションを保つことが難しい（コミュニケーション管理）

（情報が溢れている、情報が正しく伝わらない、必要な情報が見つからない、
会議が多い、電子メールが多い）

ユーザ、開発拠点、協力会社
の複雑な関係があり難しい

大規模開発の問題点（マネージメント）

プロジェクトが成功しない理由は、技術よりマネージメントの問題が大きい



技術が優秀な集団であってもプロジェクトは成功しない

大規模開発の問題点（開発プロセス、技術）

開発者は他人のソフトウェアおよびシステム全体を理解できない

→ 成果物が外在化されていない
（例えば、コードしか残っていない、アーキテクチャドキュメント（ソフトウェア構造）がない）

生産性が上がらない

→ 開発に無駄が多い
（同じようなコードを毎回作成する、再利用されていない、開発者によって成果物のバラツキが大きい、ゼロからの開発）

変更箇所が大、拡張が難しい

→ アーキテクチャ設計が弱い（基本設計なし）
骨格無しに追加に追加を重ねている
（サブシステム分割、レイヤー化、サブシステムI/F、フレームワーク設計）

大規模開発の問題点（開発プロセス、技術）

技術が組織に残らない → 技術が個人に依存
（同じトラブルを繰り返す、協力会社に蓄積）

リリース後に欠陥が発生する → レビュー、テストが不十分である
（テスト時間が確保できない）

原因

何故問題が発生してきているのでしょう？

原因

今までの数人の職人的なエンジニアが開発してきた(実験的、研究室的）方法と変わっていないのでは、

◆試行錯誤を行いながら、困難を乗り越えてきた経験を元に開発を進めている

◆ソフトウェア工学について知らない、実践できない
（プログラミング重視、ソフトウェア工学軽視）

◆ソフトウェアを開発するための必要な教育(大学/企業）をしていない
（ソフトウェアはプログラミング言語を知っていれば開発できると思っている）

原因

◆人海戦術によるプログラミング開発

⇒　大量の無駄

◆納期のプレッシャによる突貫工事

犬小屋を作る方法で大きなビルを建てるような事をしている

設計図面を描かず、基礎工事なしで開発を進めている

原因

◆プロジェクトをマネージメントできる人がいない

プロジェクトの管理ができない

（リソースの割り当てや管理ができない、顧客や開発者との調整ができない）
（プロジェクトが正しい目標に向かえるようにするための全般的な管理および
リーダシップが行えない）

◆アーキテクトがいない

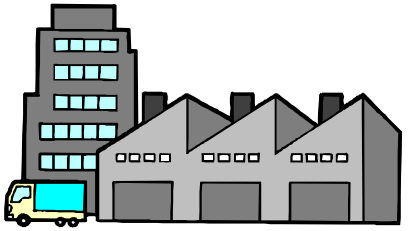
システムの基本設計（アーキテクチャ設計）ができない

（ドメイン全体の知識不足、技術方向性・方針が出せない、技術判断が出来ない）
（サブシステム間調整ができない、担当者毎に別方向に進む（統一が取れない））

解決方法

何をどのように解決していくのでしょう？

改善の必要性



メーカは、付加価値のある製品を競合他社より先に販売し、売上利益を確保したい

組み込みシステムソフトウェアエンジニアリング改善が急務

ソフトウェアエンジニアリング改善が必要

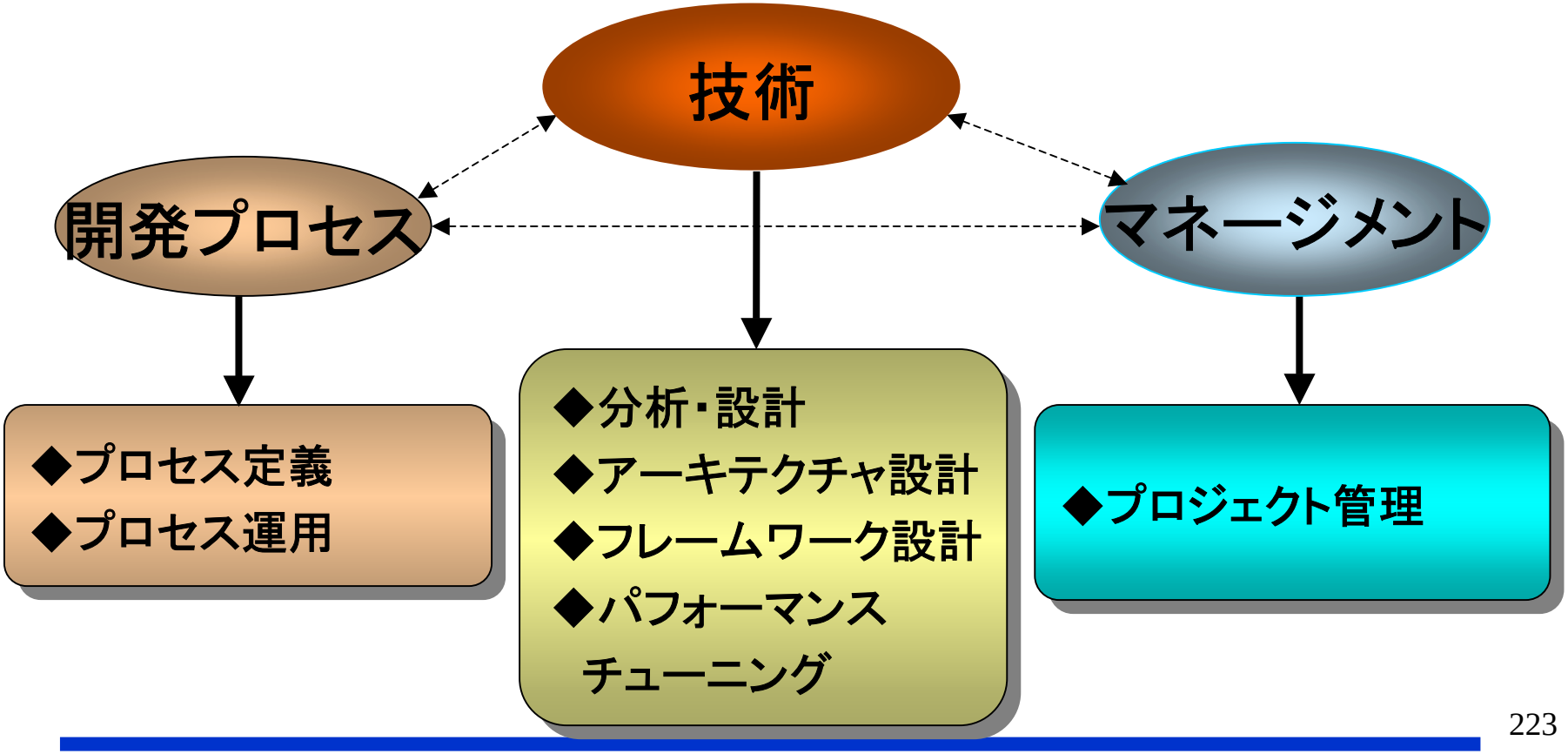
- ✓生産性向上
- ✓品質向上
- ✓開発期間短縮

ソフトウェア工学実践

- ・プロセス/手法（構造化手法、オブジェクト指向）
- ・プロセス/プロジェクト管理（CMM、PMBOK）
- ・教育（プロジェクト管理、分析、設計手法、テスト）
- ・再利用（フレームワーク）
- ・ツール（構成管理、要件管理、CASE）

3つの視点によるバランス良い解決方法が必要

ソフトウェア工学に基づいて開発する



解決方法（マネージメント）

プロジェクトの可視化

（何をしようとしているのか、状況はどのようになっているかを
プロジェクト関係者で見えるようにする）

例えば、CMM、PMBOK等のプロジェクト管理のフレームワークを参考にして、
組織の状況に合わせた改善を行う

解決方法（マネージメント）

全てを実施せず、基本から段階的に行いましょう

- ◆プロジェクト計画 ---- スコープ記述
  （範囲、達成目標(品質、予算、納期)）
- ◆組織体制 ---- 役割、責務の明確化
- ◆定量的進捗管理 ---- 進捗状況の可視化とタイムリーな是正措置の実施
- ◆リスク管理 ---- 技術リスク、政治的リスク対応
- ◆コミュニケーション ---- 利害関係者間の情報共有
  （いつ、誰が、どのように必要か）
  情報共有ツールの利用
- ◆構成管理 ---- ソフトウェア成果物の制御を実施する

詳細は二日目のセミナーで！

解決方法（開発プロセス）

◆要件を良く理解（分析）する （何を作ろうとしているのか考えましょう）

◆保守性、拡張性、再利用性を考慮して設計する

（一人で開発しているのではなく、みんなに使われる事を意識する）

◆レビュー、テストをしっかり行う

開発者が多いので誰が何をどのように作るのかを明確化する

＜プロセスの要素＞

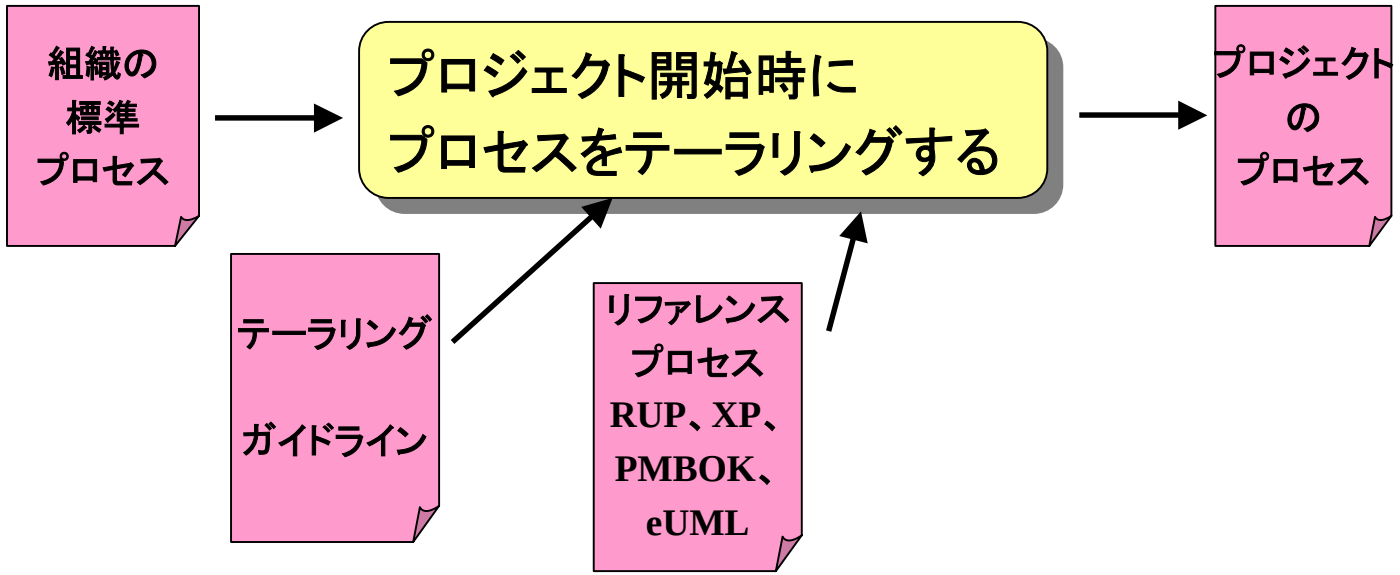
ワーカ（Workers）:	誰が行うのか
作業(Activities):	どのように行うのか
成果物（Artifacts）:	何を作るのか
ワークフロー（Workflow）:	いつ行うのか

解決方法（開発プロセス）

◆プロセスを定義してはどうですか

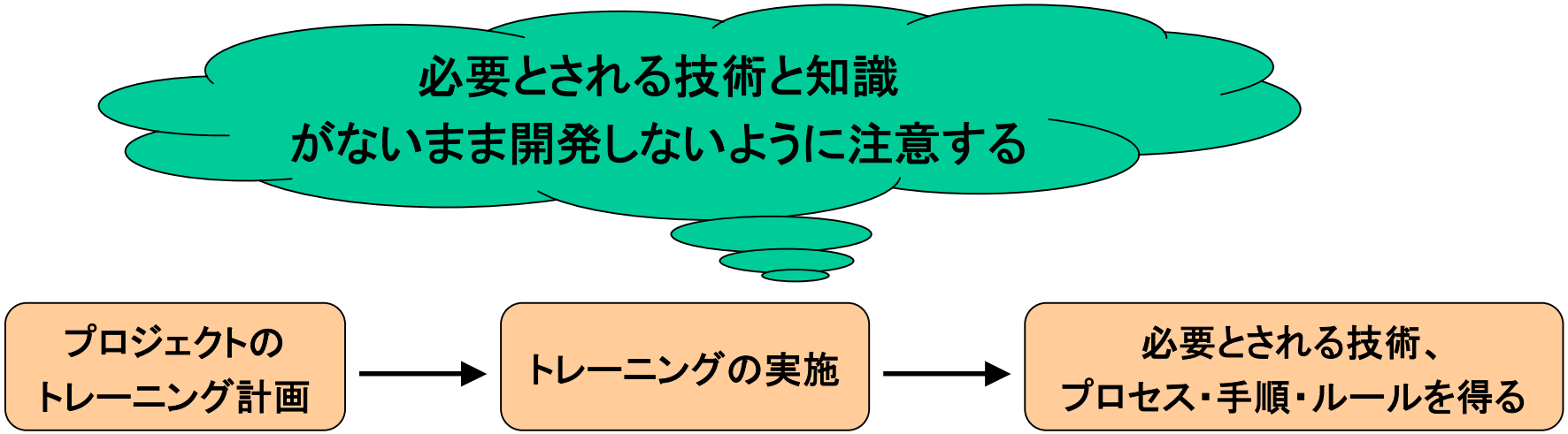
ソフトウェアの規模（開発者の人数）、開発者のスキル、ソフトウェアの種類（組込み系、業務系）、開発期間など各要因によってプロセスは異なる

開発チームは、提供された標準プロセスに従い、チーム毎にテーラリングする必要がある



解決方法（開発プロセス）

◆プロセスが定義できたら、プロセストレーニングを実施する



- ✓予算と工数の確保
- ✓役割に応じたどのような
  トレーニング必要か
- ✓いつ必要か

解決方法(技術)

◆分析、設計手法 ⇒ ソフトウェア（分析、設計）の可視化 ＝ モデリング

（例えば）

オブジェクト指向 ⇒ システムの抽象化、局所化により、
再利用、拡張性、変更容易性を高める

構造化手法 ⇒ 本日の前半のセミナー内容を復習
しましょう

モデリング ⇒ UML
共通モデリング言語によるシステムの理解しやすさ、
レビューのしやすさ（コミュニケーションツール）

◆アーキテクチャ設計 ⇒ 複雑なシステムを作成するには基本設計が必要

◆フレームワーク設計 ⇒ ゼロから開発するのではなく差分での開発

解決方法

大規模開発では、下記役割が重要

◆プロジェクトマネージャ

◆アーキテクト

育成には時間がかかるため、資質のある候補者を選んで必要なスキルを身に付けさせるための専門教育を実施する（発掘、育成）

開発者へのメッセージ

では、みなさんが今後大規模開発に
参加するときに何に注意すれば
良いのでしょう？

開発者へのメッセージ

大規模開発に入っても良いように

マネージメント

◆担当部分を見積り、計画を立てましょう
◆計画にに基づいた実績（進捗）を報告しましょう
◆計画と実績が乖離したら勇気を持って再計画しましょう
◆メトリクスデータを取りましょう（作業時間、成果物サイズ、欠陥数）
◆問題を早くプロジェクトマネージャに報告し、一緒に考えましょう
◆プロジェクト終了時に評価をしましょう

開発者へのメッセージ

大規模開発に入っても良いように

開発プロセス/技術

◆要求仕様を良く理解（分析）してから開発しましょう
（あいまいな要求を減らす、勘違いを減らす）

◆コーディングを始める前に設計をしましょう
（変更が容易なように、追加が容易なように、同じ物を作らないように、）

◆軽い気持ちでレビューをしましょう
（習慣化する、他人から成果物を評価してもらう、他人の成果物を評価する、
良い成果物を見る、人は評価しない）

開発者へのメッセージ

大規模開発に入っても良いように

開発プロセス/技術

◆読みやすいプログラムを書きましょう
(他人に読まれることを意識する、良いコードを見ましょう)

◆良いやり方を学びましょう
(継続した学習(2ヶ月に1冊は専門書を読む)をしましょう)

最後に

基本からしっかりやっていきましょう（ABC)

現状にとどまることなく、常に改善しなければという
危機感、緊張感を持ち続けましょう
改善すべき問題を自覚しましょう

本日のセミナーで使用した用語がわからなかった人は勉強しましょう

付録: 話題沸騰ポットのシミュレーション

担当：ノキア・ジャパン株式会社　山崎辰雄

アジェンダ

1. なぜ、ソフトウエア・シミュレータ?
2. どこまでシミュレートするの?
3. どうなっているの?
4. なにがいいの?
5. 気をつけなければならないこと
6. デモンストレーション
7. つぎはどこへ?
8. まとめ

なぜ、ソフトウエア・シミュレータ? (1)

SESSAMEでポットを作ることになりました。

はじめは、**LEGO　マインド・ストーム**を使うつもりでした。

・理由

　開発環境がそろっている

　比較的安価 (約4万円)

　くじけても、子供のおもちゃになる

・買ってはみたけれど

　オプションで、温度センサはあるけど、ヒーターがない

　水にはつけられないよね

　やっぱり、お札が飛んでいくのはイヤ

なぜ、ソフトウエア・シミュレータ? (2)

ワンボード・マイコンで作ることにしました。

・理由

メンバーに提供していただきました

作り込めば、なんでもできる

・作業に取りかかりましたが

ハード仕様書を書いた

回路図もできた

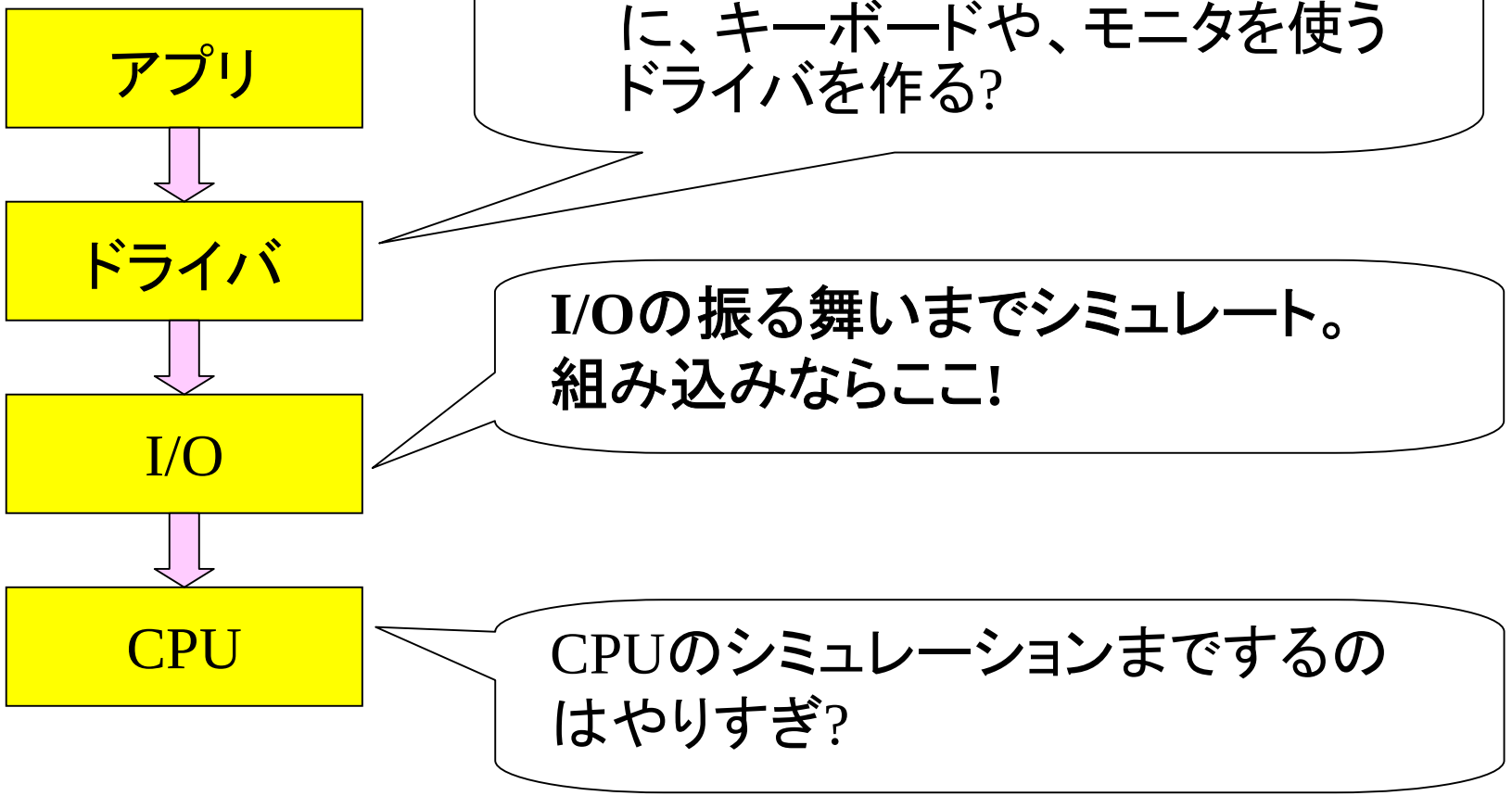
でも、なかなか形にならない

ちょっと違うマイコンのCも勉強しなくちゃ

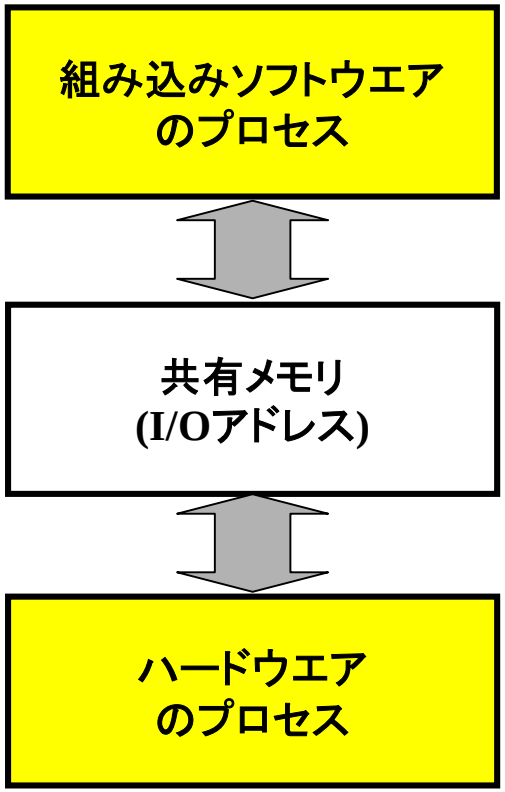
このセミナーの日程が迫ってきましたぁ

← 仕事で苦しんでるのそのまんま

どこまでシミュレートするの?



どうなっているの?



共有メモリを用意して、
二つのプロセスから読み書きする。(I/O空間のつもり)

片方のプロセスは、
ハードウエアをシミュレート
もう一つのプロセスが、
組み込みソフト

なにがいいの?

- ハードウエアがなくても開発開始できる
- ハード屋さんに頼まなくても自分で拡張できる
- お財布に優しい
- 本当のハードウエア同様にI/Oアクセスできる
- ハードウエアが手に入ったら、ポインタの変更だけで対応できる
- シミュレータは別プロセスなのでリンク不要
- 組み込みソフトの変数も、共有メモリにおけば他のプロセスを使ってモニタできる。

でも、いいことばかりじゃないよ。

気をつけなければならないこと

- シミュレートしたところまでしか確認できません
- CPUまで、シミュレートしていません。つまりマイコンのCの仕様と、シミュレータのCの仕様は違うかも
- 実時間処理はやっぱり違う
- 外部環境も何らかの形でシミュレートしなければ。。
- ソフトはもちろん、ハードも設計できる人でないとよいモデルが作れない
- 物理現象の知識もいる。ポットだと、水にヒーターを通して電力をある時間与えると何度温度が上昇するのかな????

デモンストレーション

動かすのにUnixが必要です。

- 共有メモリ
- シグナル
- ノン・ブロックI/O (BSD)
- マルチ・プロセス
- cursesライブラリ
- マルチ・ターミナル

レシピはSESSAME Webサイト

http://blues.tqm.t.u-tokyo.ac.jp/esw/

をごらんください。

つぎはどこへ

- ***商品企画***

  プロトタイプ作って動かして商品を提案しましょう

- ***協調設計***

  ソフト屋自らソフトの作りやすいハードを提案しましょう

- ***ハードウエア知識***

  シミュレータを作ることでハードの立場からも考えることができるようになります。組み込みは、ハード&ソフトです

まとめ

シミュレータの例として、
I/Oレベルでハードウエアをシミュレートする手法を
簡単に紹介させていただきました。

工夫次第で、より快適な開発&テスト環境が得られます。
組み込みを極めるため
“ハードウエア”にも手を出してみられませんか。