実践的モデリング論

- 難しいことを考えずにモデリングを実践するには -

2014年11月7日

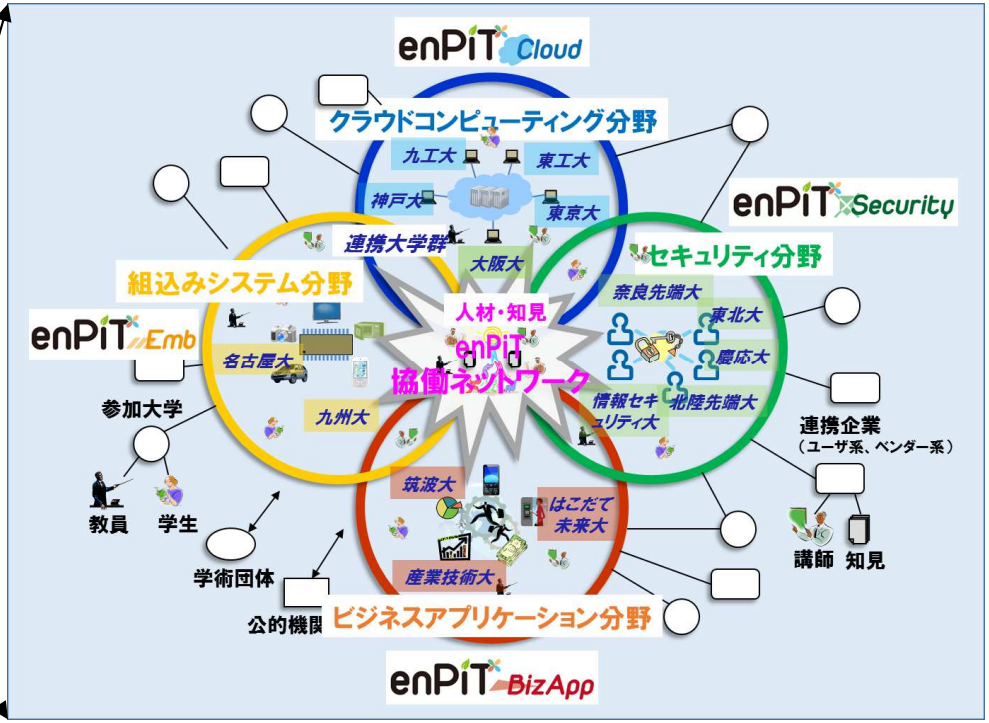
大阪大学大学院情報科学研究科
特任教授　春名　修介

自己紹介

- 1977年松下電器産業株式会社入社
  - 4ビットμプロセッサのクロス環境の開発から組込み業界へ
- 研究開発部門において，プログラミング言語処理系，ソフトウェア開発環境，リアルタイムOS，プロセッサアーキテクチャの研究開発及び製品開発に従事
  - アセンブリ言語からC言語への移行を推進．全国行脚
    - 》一時期，ハード設計の経験も
  - C++，Javaのデジタル家電応用を推進．組込みソフトの大規模化の中へ
- 2001年よりエンジニアリング部門において，ソフトウェア開発力向上のための全社施策の企画に従事
  - 特に，組込みソフトウェア向けのアーキテクチャ設計方法，モデル駆動開発手法の普及び社内教育を推進
- 対外活動
  - 組込みシステム産業振興機構 システムアーキテクト育成カリキュラム「組込み適塾」の企画・講師
  - JEITA ソフトウェア事業基盤専門委員会委員（2005～2013.8.31）
- 2013年9月より大阪大学にて，情報教育ネットワークenPiT参画

enPiTとは

- 全国15大学が連携した，PBLを主体にした実践的情報教育プロジェクト（文部科学省事業）．



大学院生向けに1年間実施．単位認定

2013年度修了者　308名

2016年度の修了者目標：1200名程度

本日の内容

- モデルとは
  - 目的が重要 -
- アーキテクチャとは
  - 目的の一例として -
- 組込みソフトウェア・アーキテクチャ設計におけるモデル
  - 何を表現しなければならないか -
- 表記法の事例
  - 対象・目的に合った表現とは -
- まとめ：モデリング成功のポイント

モデルとは

- ある目的のため必要な部分のみを残した抽象的な表現
  - ソフトウェアの場合，コードより抽象度が高い
  - 目的外の内容は，省略されている
- モデルに共通した考え方（詳細を省略した後，残るものは？）


と

間の関係を表現


と表現方法は，目的により異なる

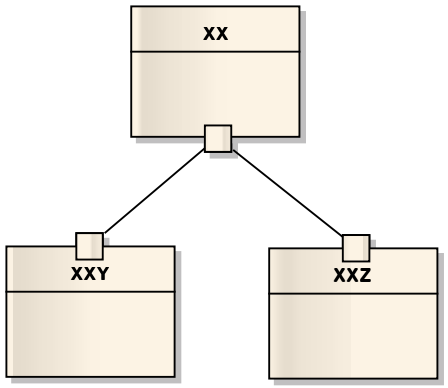
他者に全体像を伝えるための手段．関係者間での分かり易さも重要な観点

- 例
  - 構造モデル ⇒ 機能（責務）の利用関係を記述
  - 要求モデル ⇒ 要求間の関係を記述
  - フィーチャモデル ⇒ 構成機能の固定・変動の関係を記述
  - 状態遷移モデル ⇒ 状態間の遷移関係を記述
  - ビジネスモデル ⇒ ビジネスの諸要素（人・もの・金）の関係を記述

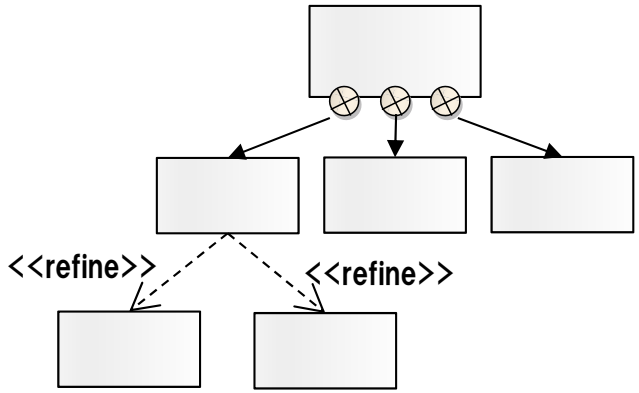
モデルの例

■ 色々なモデル表現

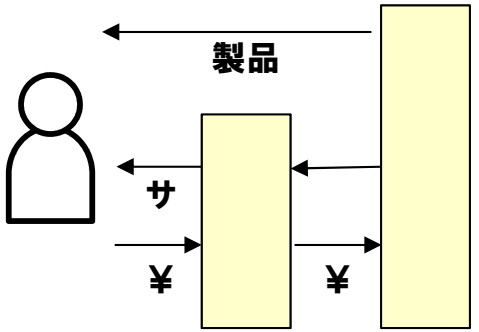
【構造図】
SysML内部ブロック図
コンポーネント間の関係を表現



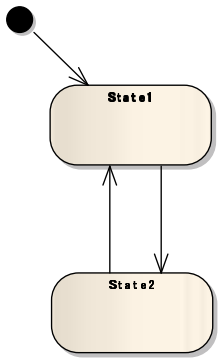
【要求図】
SysML要求図
要求間の関係を表現



【ビジネスフロー図】
ピクト図
￥，人，ものの流れを表現



【状態遷移図】
SysML状態遷移図
状態間の遷移関係を表現



皆さんも記法に関わらず普段から行われていること
（何も難しいことはないハズ？）

アーキテクチャとは
- 目的の一例として -

ソフトウェア・アーキテクチャの定義

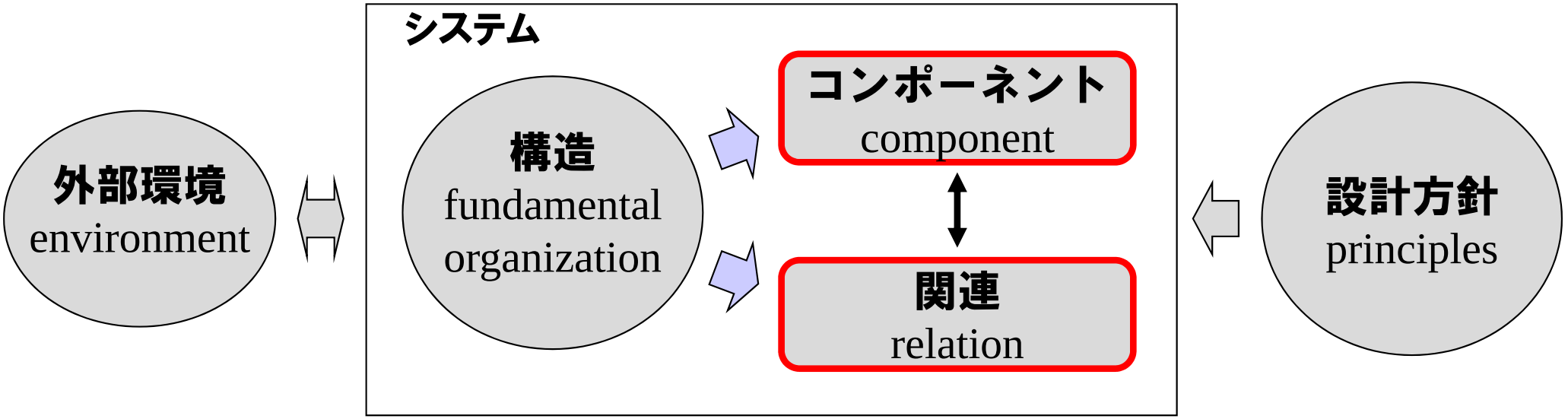
■ 国際規格：※IEEE STD 1471-2000に見るアーキテチャの定義

※IEEE std 1471-2000:
IEEE Recommended Practice for Architectural Description of Software-Intensive Syste

説明も抽象的？

システム内のコンポーネントとそれらの関連や外部環境との関連で表現される基本構造及び設計とその進化の指針となる方針

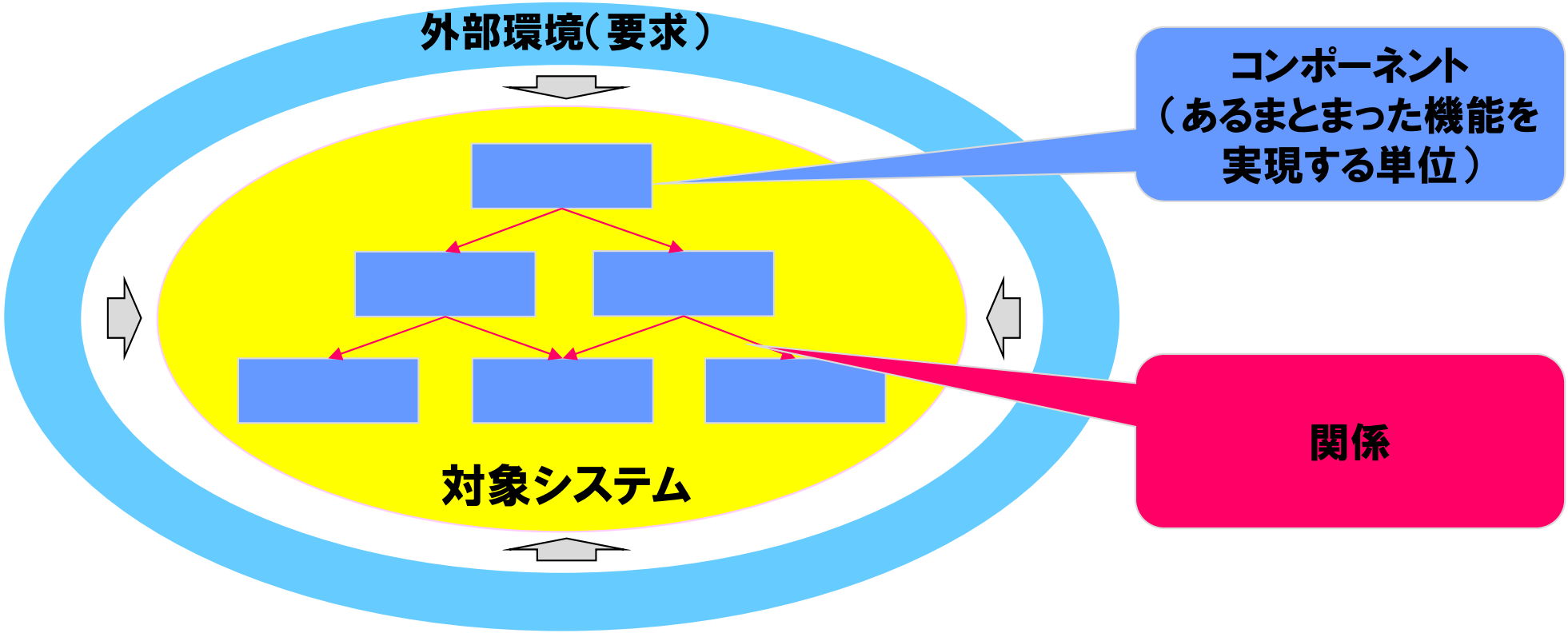
3.5 architecture: The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.



ソフトウェア・アーキテクチャの私ないの解釈

- ある要求を実現するソフトウェアを設計するに当たり，
- システムをコンポーネントに分割，
- コンポーネントとその関係を規定し，
- 分割の考え方・要求との対応などと共に表現されたもの

分割の視点は色々（View）



視点(View)とは

- 全ての関係を一度に表現することは不可能
- 関心事毎にモデルを用いて表現
- 機能分担の視点：静的構造
- 動作の視点　　：動的構造
  - 状態遷移もこの範疇
- ネットワーク，分散
- 横断的関心

⋮

代表的な視点はあるが固定化されたものではない．設計するシステムを表現するために必要な視点を決めればよい．

よくできた概念モデルです

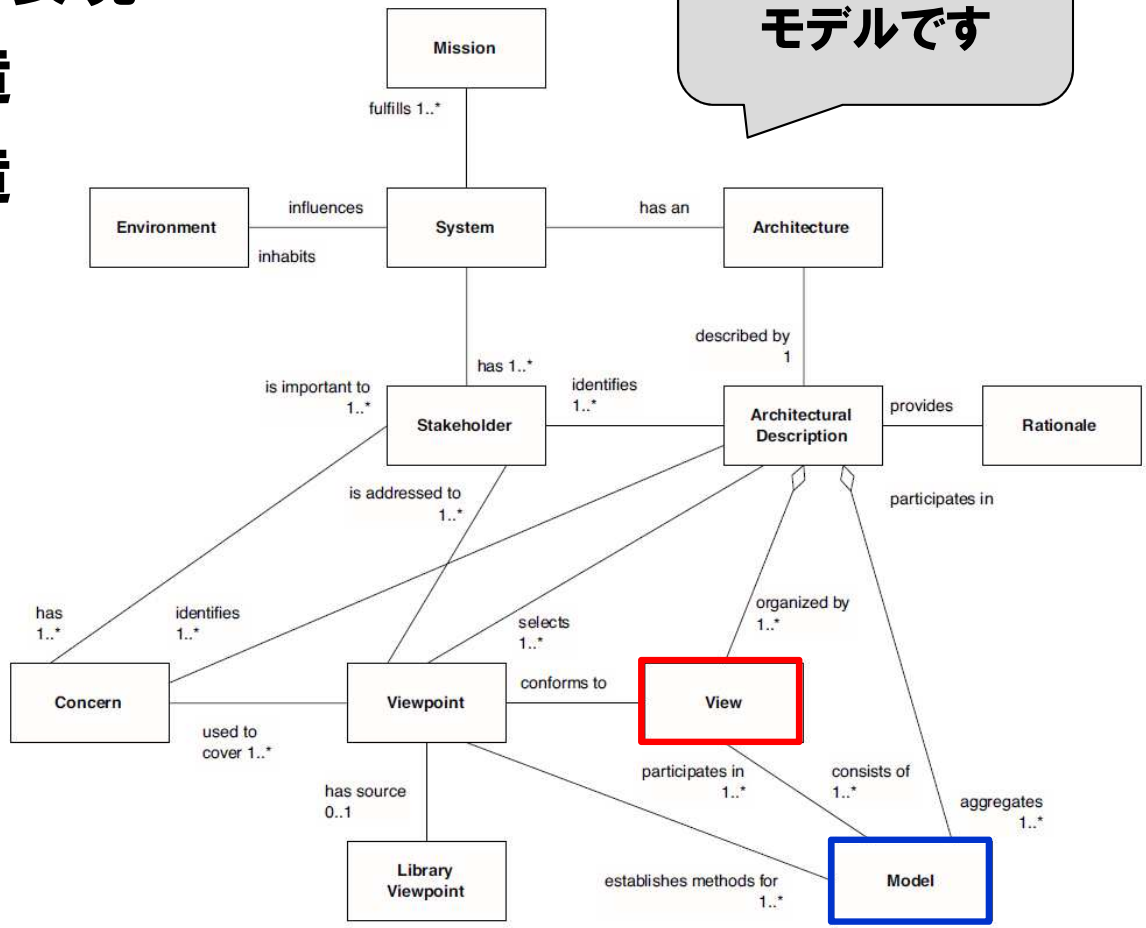


Figure 1—Conceptual model of architectural description

IEEE std 1471-2000より引用

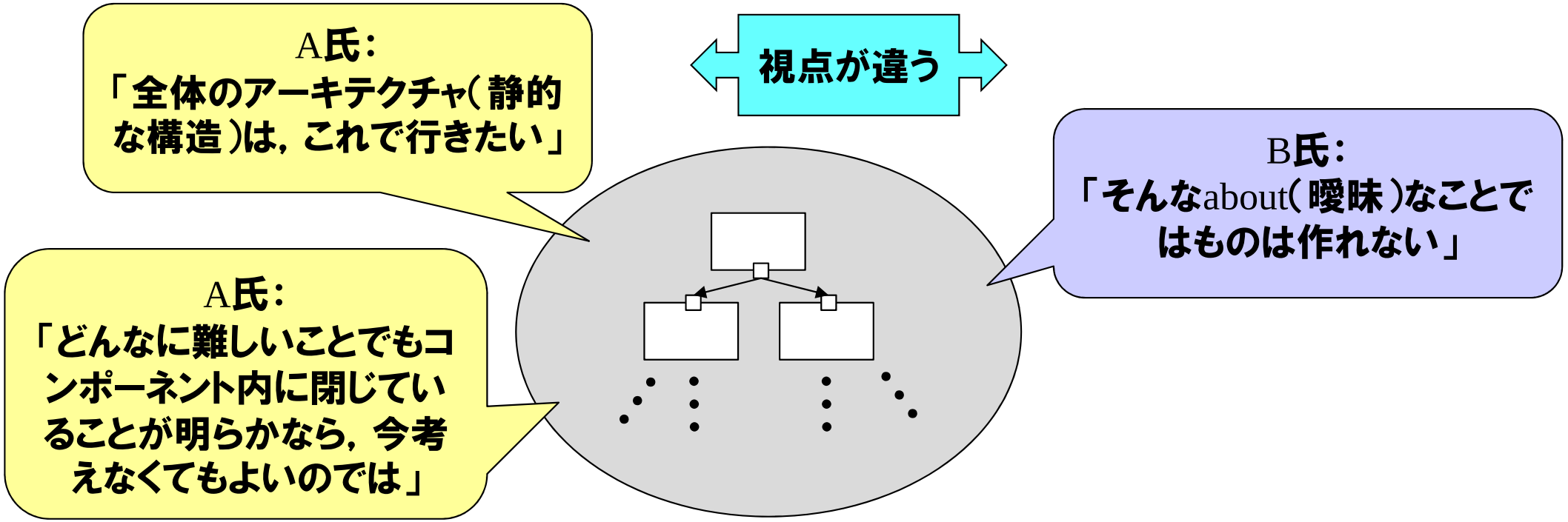
アーキテクチャ設計の目的

- **目的**
  - 全体構造を可視化
    - » システムをどのように分割し，分割単位：コンポーネント間の関係を規定するか
    - » コンポーネント内に閉じた詳細には言及しない ⇒ モデル化
  - ステークホルダ間で開発するシステムの全体像を共有する

- **効果（もっと色々あります）**
  - 設計制約を与えることにより，担当者間の擦り合わせを減らす
  - 設計漏れによる大きな手戻りを減らす
  - 再利用など，設計時に考慮するべき非機能用件への対応が明確になる
  - 修正時に，修正の影響範囲が明らかになる
  - 揉め事が起こったとき，論理的に正しい方向に議論を進めることができる
  - 次の機能進化の議論のベースとなる　などなど

事例①：アーキテクトA氏の憂鬱

■ 抽象化と曖昧さ



抽象化することと，曖昧になることとは違う

コンポーネント内の詳細は欠落するが，コンポーネント間の関係は厳密
細かいことが解決しないと先に進めないタイプの人は，アーキテクトには向いていない？

組込みソフトウェア・アーキテクチャ設計における
モデル

{
・組込みソフトウェアのアーキテクチャ設計では
何を表現しなければならないか
・どのようなViewがあるのか
}

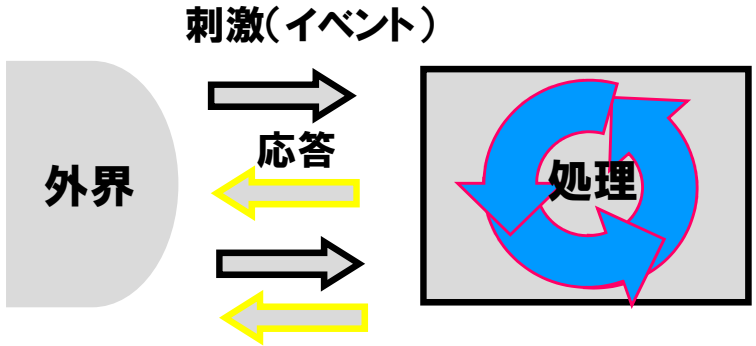
組込みソフトウェアの特徴：釈迦に説法ですが

① 入力出力型

・入力と出力に時間関係／制約はない
・処理は，逐次的に実行
・処理が完了するまで，次の入力はない
・一回の入力毎に処理が完結
例：数値計算など
　アルゴリズムの教科書に出てくる処理

② イベント駆動型

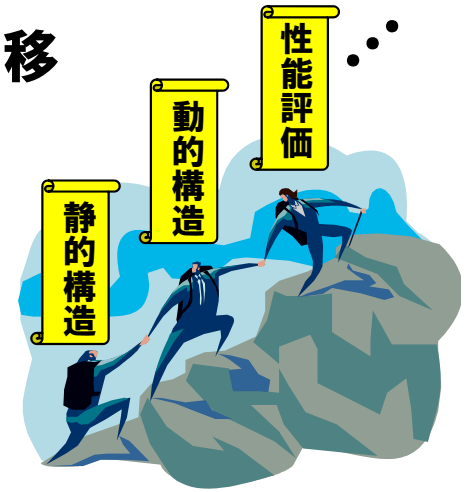
組込みソフト

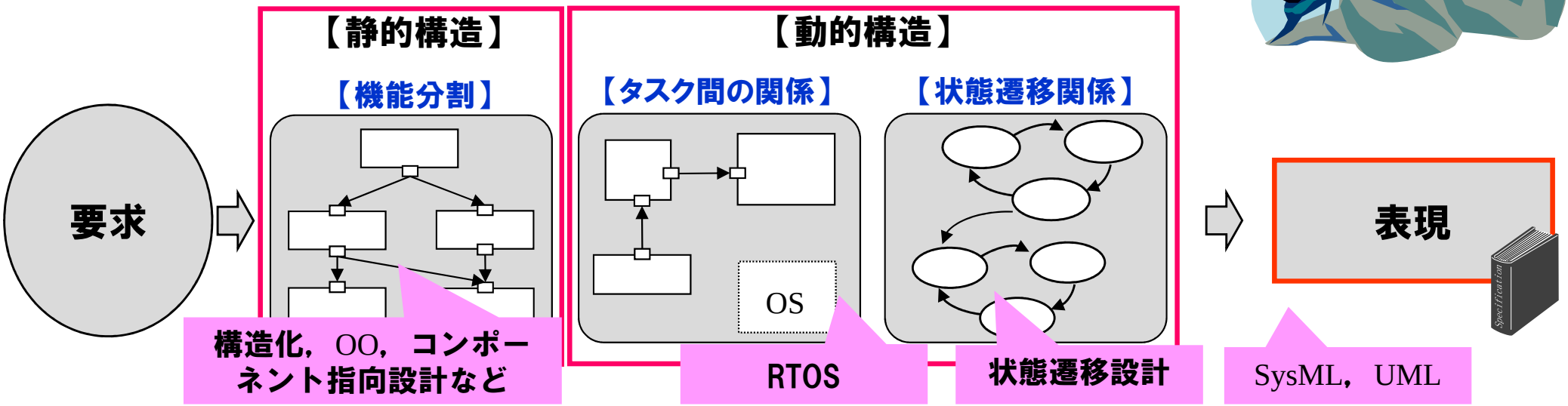


・入力と出力に時間関係／制約がある
・処理は，並行的に実行
・1つイベントの処理中にも，
　別のイベントの処理をする必要がある
・中断，処理の変更は例外ではなく設計項目
例：ハード制御，通信制御，・・・・

構造設計の3つの視点（View）

- **静的視点と動的視点の両面を表現する必要がある**
  ＜組込みソフトウェアは，静的な構造設計で，ひと山越えるのみ＞
  - 静的構造：①機能分割（責務分割）
  - 動的構造：②並行動作（タスク間の関係），③状態遷移
- **表現することも同時に重要**
  - 設計力と表現力は表裏一体
  - 関係者が理解し易い表記法の選択も重要





表記法の事例
- 対象・目的に合った表現とは -

代表的な構造の表記方法

- **国際規格:IEEE STD 1471-2000に見る静的構造の表現要素例**

C.1 The structural viewpoint in software architecture

付録Appendix C.1

Viewpoint language

The structural viewpoint depends on the following entities:

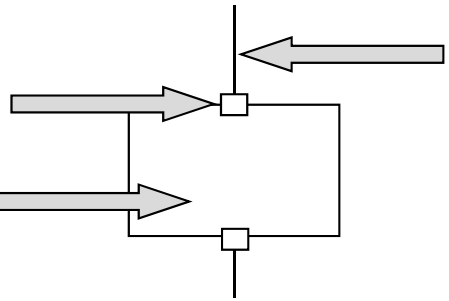
- *Components* (individual units of computation)
- Components have *ports* (interfaces)
- *Connectors* (represent interconnections between components for communication and coordination)
- Connectors have *roles* (where they attach to components)

I/F：Port

固有の処理：

Component

（独立性の高い）



関係：Connector

C&C（Componet＆Connector）**モデル**
ADL（Architecture Description Language）**の**
図的表現として著名
通信業界のSDL，SysML**の内部ブロック図，**
UML2.0**のコンポジット構造図もこの流れ**

C&Cモデルの現実的な意味

- 特別なものではなく，あるモジュールの持つ基本要素を表現したもの

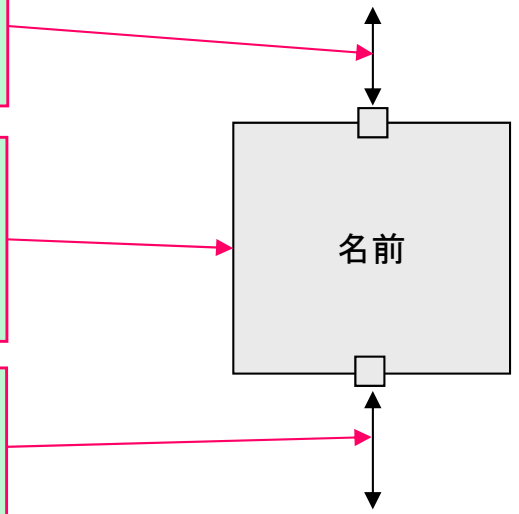
名前（モジュール名）

提供API：
外部に提供する機能のためのインタフェース

実現する機能（責務）

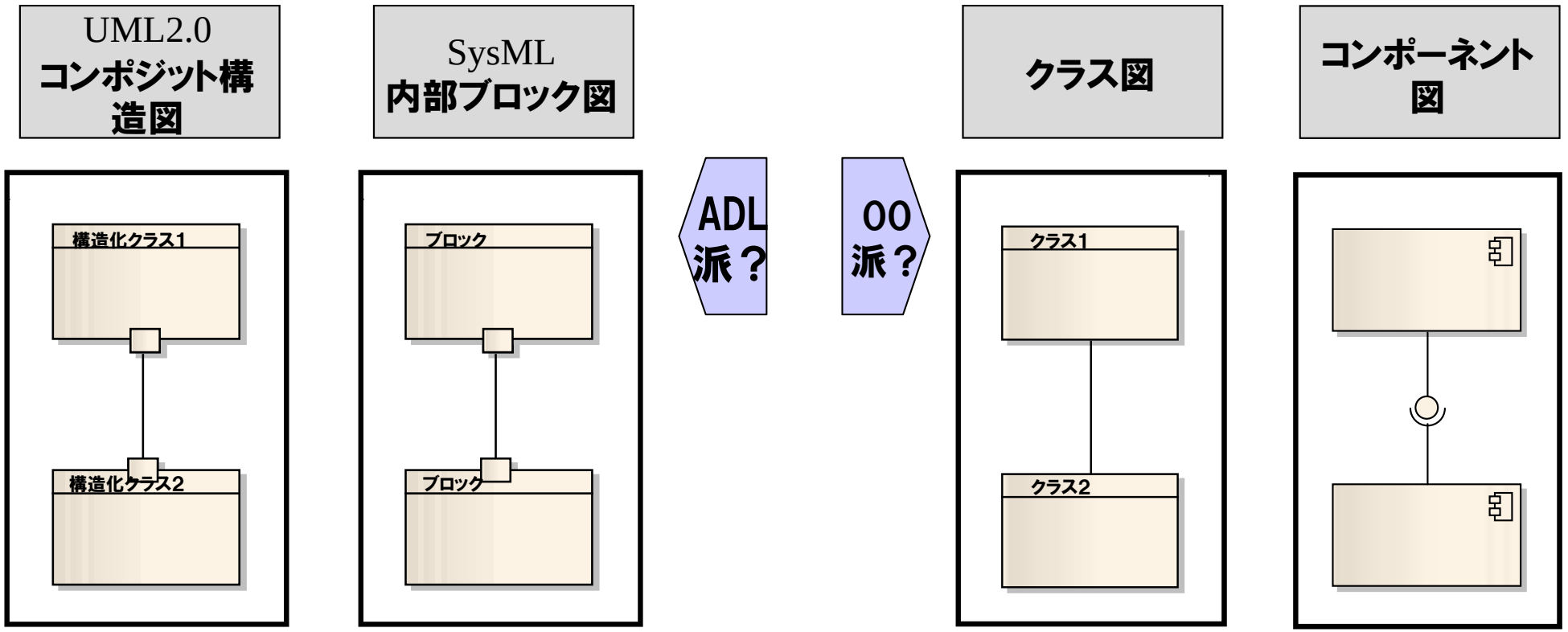
参照API：
下位に委譲する機能のためのインタフェース

C&Cモデル
ADL
SysML内部ブロック図 など



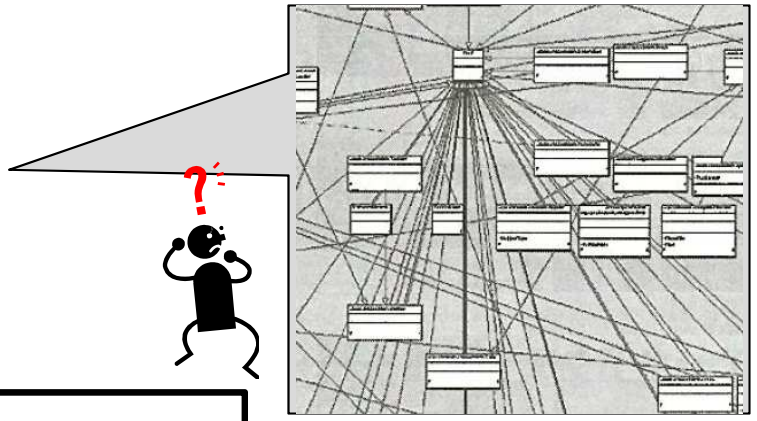
その他の構造表記方法

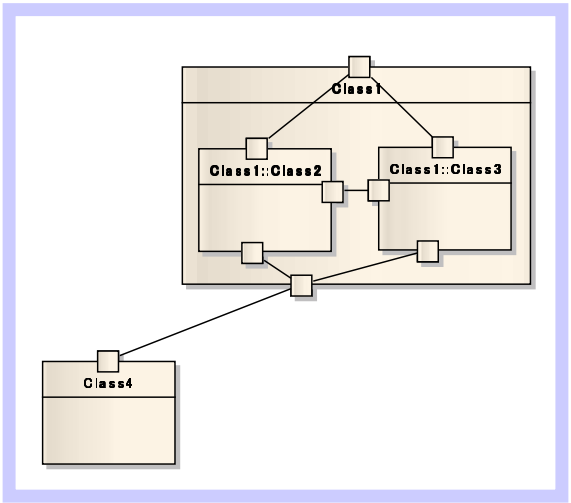
- コンポーネントとその関係の表現方法は色々あります
  - モデル駆動開発を視野に入れると異なりますが，宗教論争になります
  - それぞれ，利害得失がありますので，
  - 使用ツールや開発実態に合わせて決めればよいと思います



事例②：蜘蛛の巣クラス図

- クラス図の苦手な点
  - クラス図は階層の表現が苦手
  - 少し大きい図になると蜘蛛の巣状態に

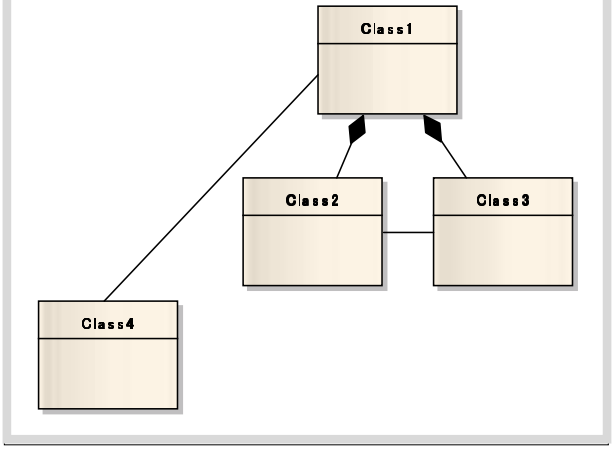




SysML内部ブロック図
UML2.0 コンポジット構造図

クラス図では

階層構造がわかり難い



オブジェクト指向言語の仕様を反映
実装に近い
今ほど，大規模ではない頃に生まれた

クラス図の ♦ ◊ ▲ △ は，
要注意
クラス自体の定義と
I/Fとしての関連とは
区別して考える

SysMLでは
ブロック自体の定義と
ブロック間の関係の記述を
分離

アーキテクチャ記述としてのUML見直しの動き

- 2000年位から，欧米ではアーキテクチャ記述の観点からUMLの見直しが始まる　→　SysML，AutoSarへの動き

http://www.spaconference.org/spa2006/sessions/session22.html

Spa 2006

Software practice advancement 2006
26-29 March 2006
The Robinson Centre, Bedfordshire, England

Technology
Practice
Process
People

SPA Conference session: Describing Information Systems: Moving Beyond UML

One-line description: A workshop that will attempt to identify the requirements and likely form of a visually oriented language for describing the architecture of information systems.

Session materials

Session format: Workshop [read about the

Abstract:

Part of the job of the soft
architectural description s
communicated to and understood by interested stakeholders (including the architect himself).

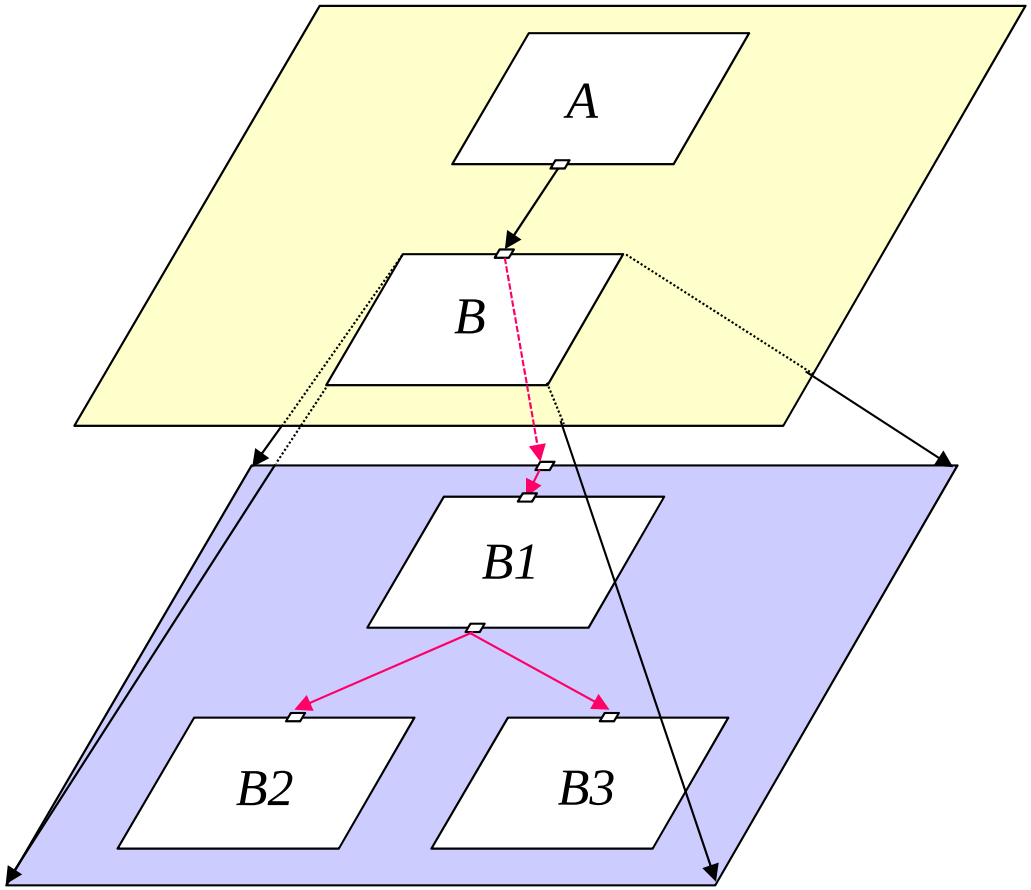
UML is the defacto standard language for describing software models, butit is primarily an object-oriented software design notation and many architects find it difficult to express their ideas using it.

Alternatives to UML have been proposed in both the academic community (in the form of ADLs like ACME, xADL and Wright) and the industrial community(for example Gregor Holpe's "gregorgrams" notation used in his EAI patterns book).

UML is the defacto standard language for describing software modules, but it is primarily an object oriented software design notation and many architects find it difficult to express their ideas using it.

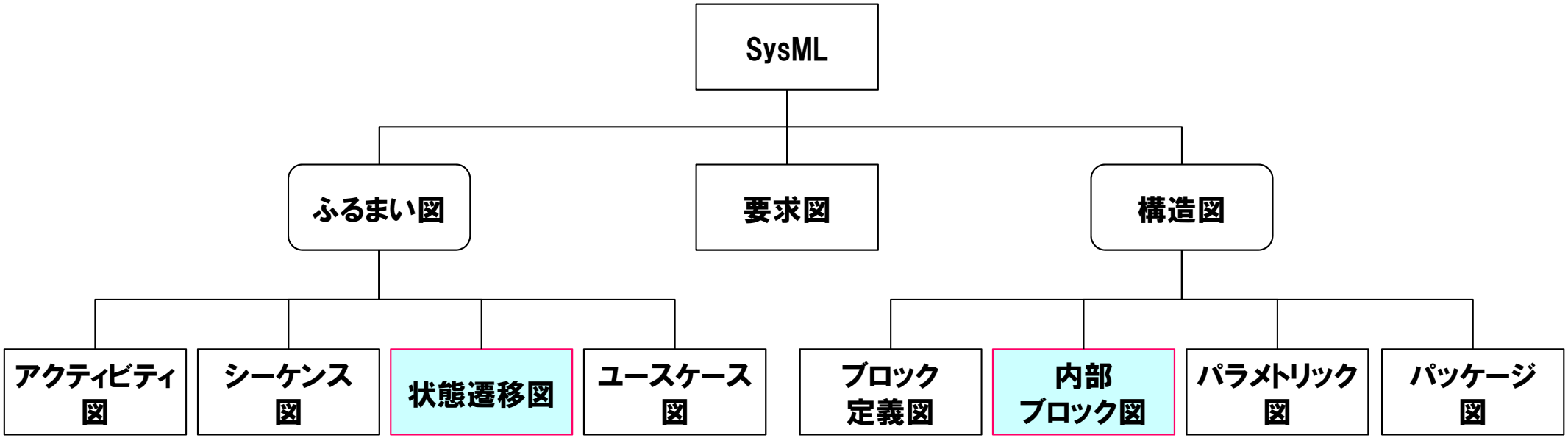
一番のポイント：詳細化構造の表現

- 規模が大きくなると，コンポーネントの粒度も大きくなり，各コンポーネントの内部構造を示す必要が生じる
- 構造の表現には階層構造（詳細化構造）の表記が不可欠



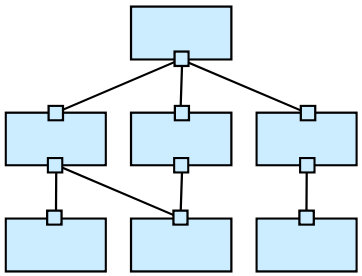
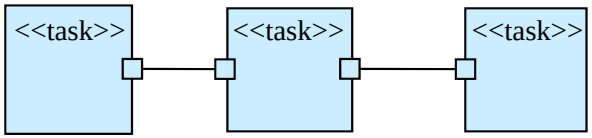
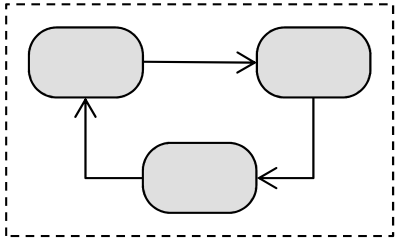
表記法の候補

- SysML
  - UML2.0からの7種類の図に要求図とパラメトリック図を加えた9種類の図
  - 階層構造の記述が可能な内部ブロック図に着目



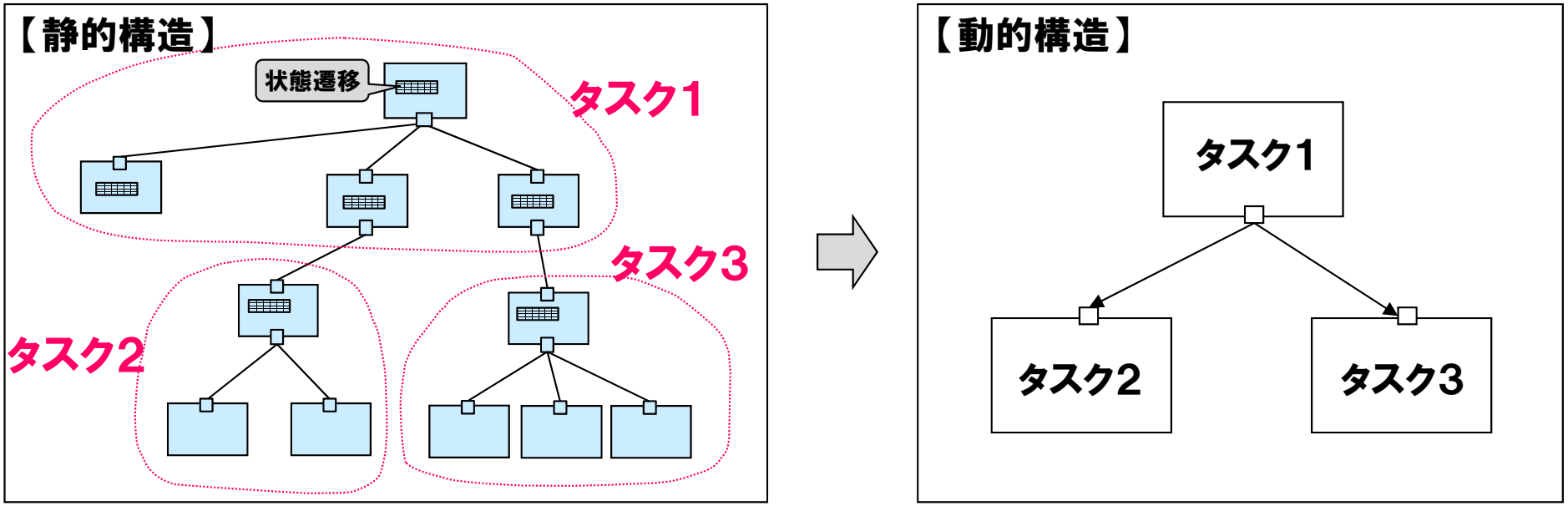
選択した表記法

- 何でもよいので，開発プロジェクトの中で統一することが重要
  - プロジェクトメンバーが親しんでいる・ツールのサポートがある・・・
  - 組込みソフトウェアの場合，最低限，静的構造・動的構造・状態遷移は決めておく

静的構造	動的構造	
静的構造図	タスク関連図	状態遷移図
SysMLの内部ブロック図 または ｺﾝﾎﾟｼﾞｯﾄ構造図(UML2.0)  機能ブロック(コンポーネント)とその利用関係を記述	SysMLの内部ブロック図 または ｺﾝﾎﾟｼﾞｯﾄ構造図(UML2.0)  メッセージ，割込み，共有メモリ，などの関係(I/F)を記述	状態遷移図(SysML，UML2.0) または 状態遷移表  静的構造の機能ブロック単位で記述

視点(View)間の整合

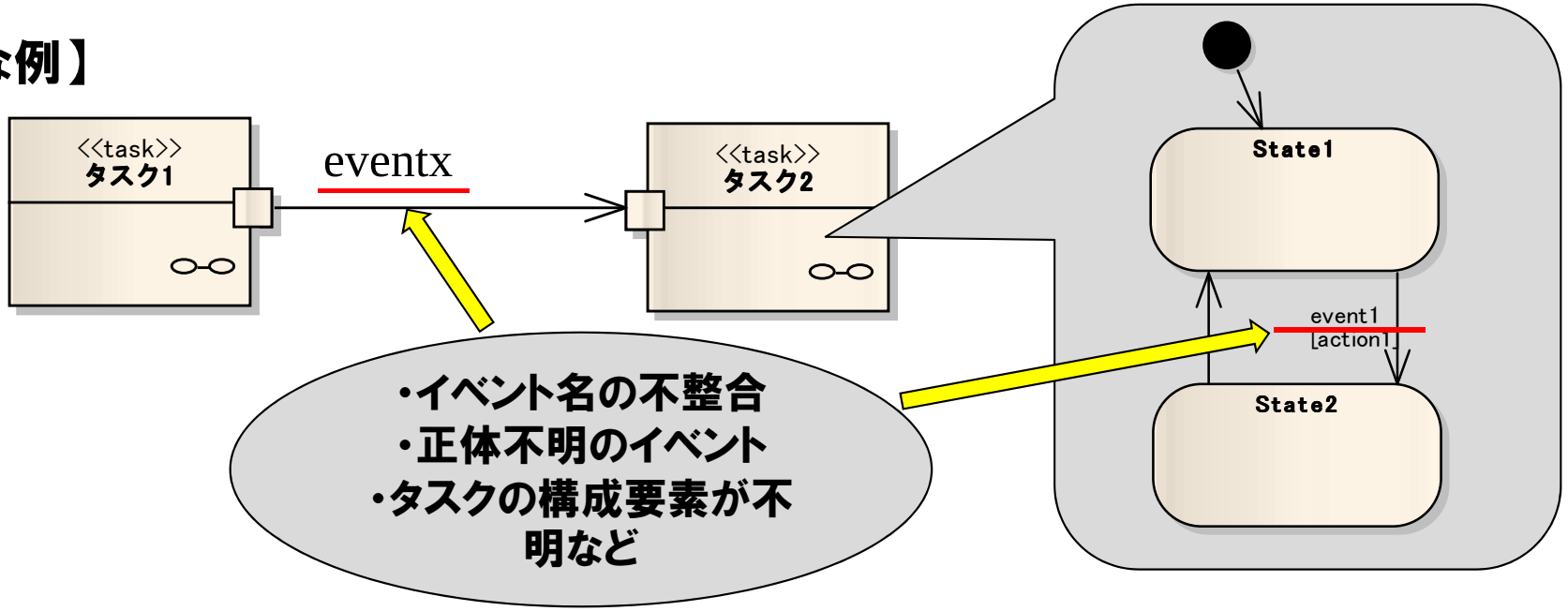
- 各視点毎のモデルに一貫性がなければ，全体として設計の意図が理解できない
  - View間の整合の方法を考えておく必要がある
- 定石
  - 並行に動作しない静的構造の各コンポーネントをタスクにまとめる
    - » 逆を言えば，並行に動作するコンポーネントは，タスクに分ける
  - 状態遷移は静的構造の単位で考え，構造と対応を明確にする



事例③：本人にしか分からないモデル

- View毎のモデル記述では，各View間の整合が取れていないと，読み手は理解できない
  - また，実現可能性が？（絵に描いた餅）の可能性が大

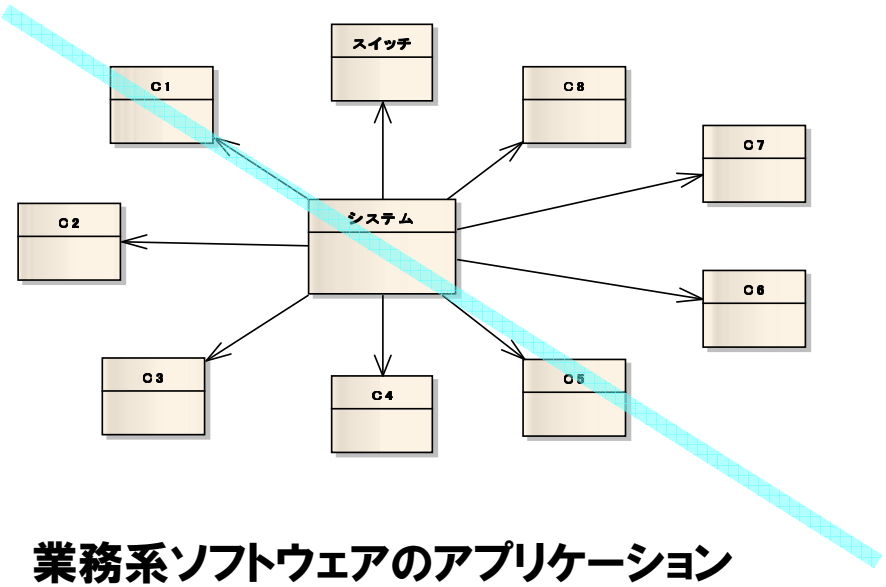
【簡単な例】



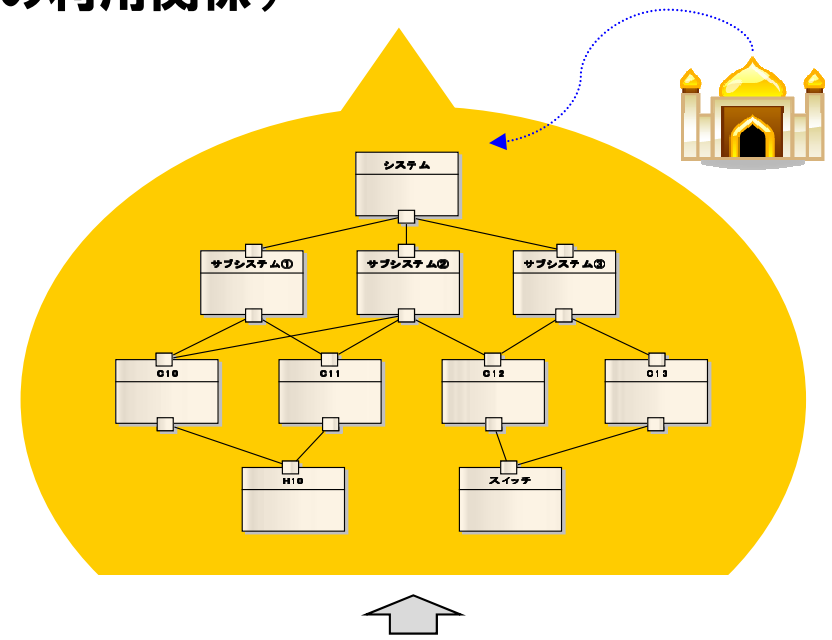
- 一般的な傾向
  - 自分が苦労したことを書きたがり，当たり前と思っていることは，書かない
  - しかし，当たり前と思っていることこそが第三者から見ると重要なことが多い

事例④：放射線状のクラス図

- システム階層の変化が表現されないと，設計意図が伝わらない
- 特に，システム階層の全てが開発対象の組込みシステムでは．
  - 一般常識
    - » 上がユーザに近い（抽象度が高い）．下がハードウェアに近い（具体的）
    - » 上が下に処理を依頼する（機能分割時の利用関係）



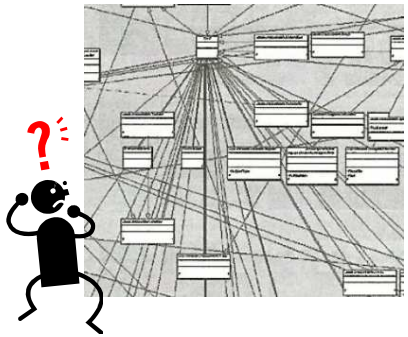
業務系ソフトウェアのアプリケーション
層の中では出てくることもあるかも？



古来からの定石：モスク型は理解し易い
（その他：STS分割，DFDの活用など）

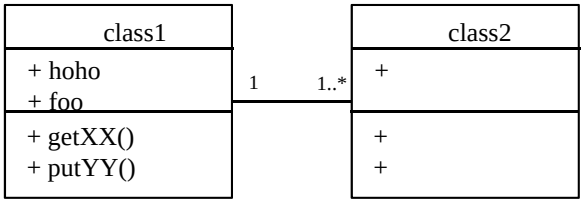
事例⑤：MDDに向けての注意

- 「全体を俯瞰するモデル」と「コード生成可能なモデル」は，観点が異なることがある
  - 「コード生成可能なモデル」の発想で，全体を考えて行くと蜘蛛の巣状態に



どうしても
コンポーネントの粒度
が小さくなる

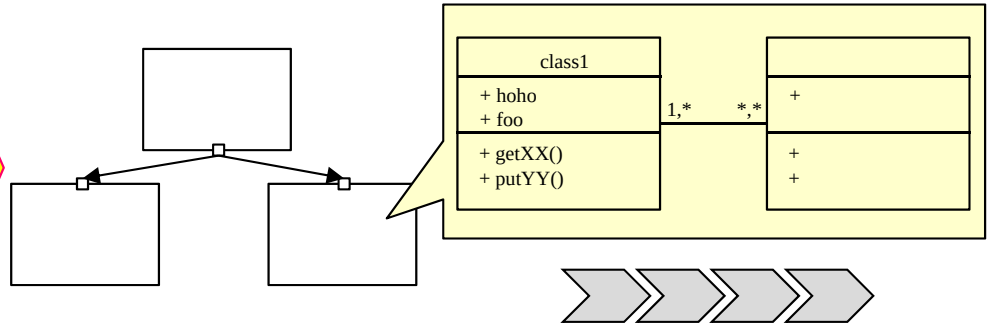
【コード生成可能なモデル】



コードがモデル
に変わっただけ
スパゲッティモ
デル？

- 全体構造を表現するモデルから詳細化してコンポーネント内のコード生成可能なモデルとの整合を取る

アーキテクチャ
全体に影響を与える重要
な関係を押さえる
（アーキテクトの仕事）



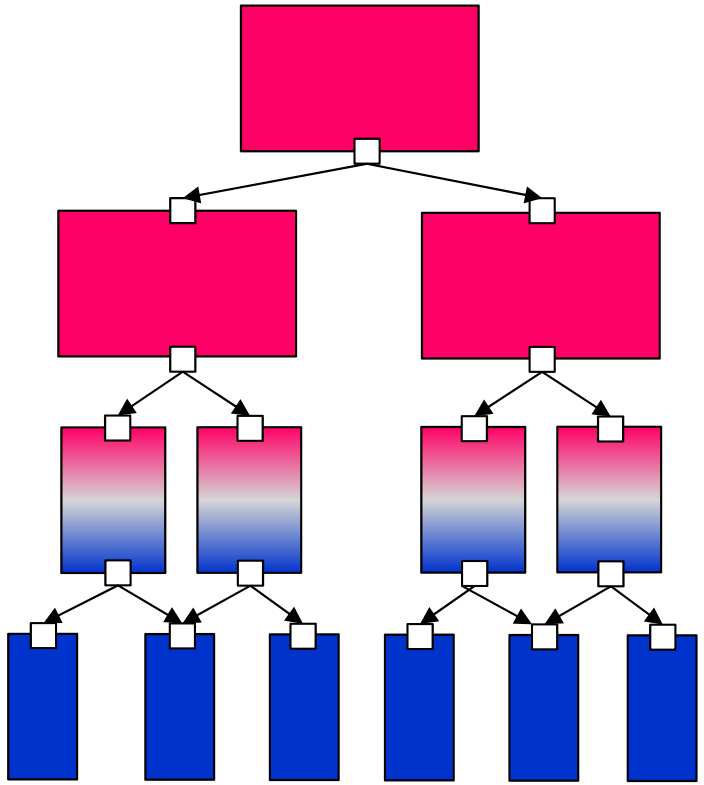
コンポーネント内
コード生成可能なモデル
を作成
（モデラの仕事？）

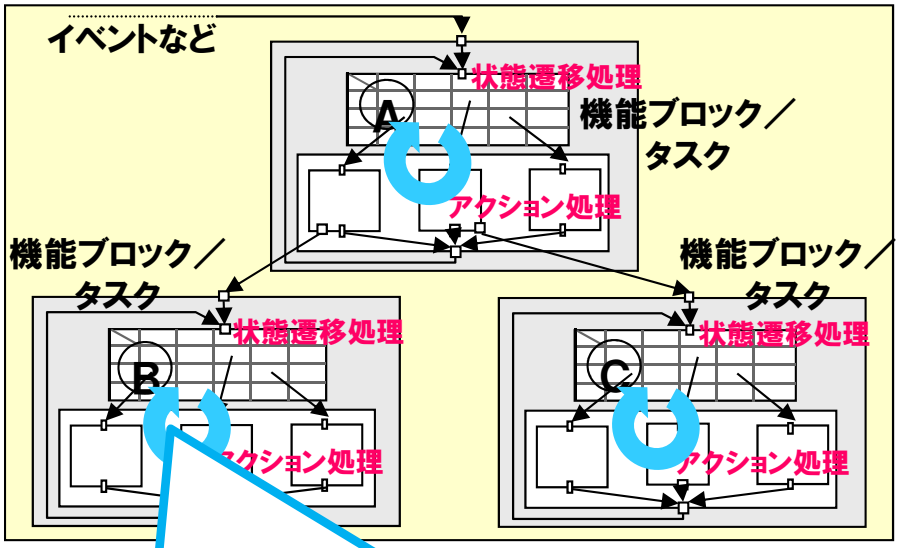
組込みソフトウェアの基本構造

- 3視点(静的構造，動的構造，状態遷移)の融合設計
  - RTOS・状態遷移を組み合わせて，非同期処理を小さな同期処理(順序処理)の集合体に分解する

状態遷移的
非同期

シーケンス的・
順序処理的
同期





状態遷移を単独で考えるのではなく
構造と対応付ける

RTC：Run To Completion

○イベント待ちは一箇所

○アクション処理の中では待たない
- 走りきる -

まとめ：モデリング成功のポイント

- 対象・目的を明らかにする
- 対象・目的に適したモデルを選択する
  - 一つのモデルに複数視点を盛り込まない．View毎にモデルを記述．
  - View間の整合を取ることが設計力．
  - 表記法の細かい文法に囚われると本末転倒．
- 昔からあるソフトウェアの基本を忘れず
  - 静的構造：構造化設計，DFD(Data Flow Diagram)の活用 など
  - 動的構造：RTOS，状態遷移，走りきり(RTC) など
- 抽象度をコントロールする
  - コードの延長ではなく，全体を俯瞰できる抽象度の発見が成功のポイント

細かすぎず，一般化しすぎず

一番難しい

ご清聴ありがとうございました

106人ワークショップでの積極的なご議論をお願いいたします