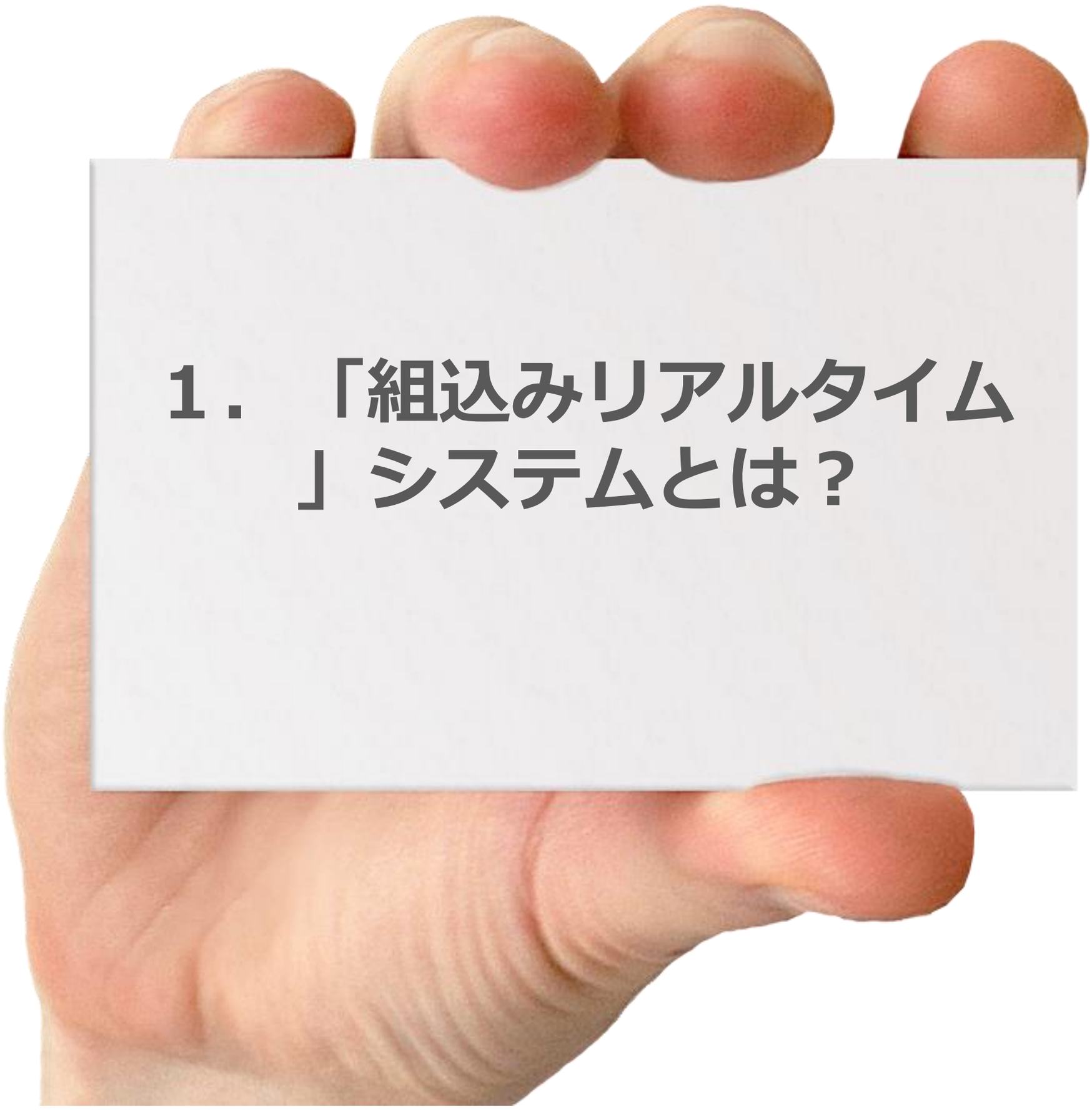
ITRON入門（中級編）

中級者向けのRTOSを使ったリアルタイムシステム開発手法入門

2016年度

YRP Ubiquitous Networking Lab.
TRON Forum

目次



１．「組込みリアルタイム」システムとは？

組込みシステムの定義

- センサやアクチュエータ、他の機械システム等と協調して動作するコンピュータシステム

- 例
  - 家電製品の制御システム
  - ファックスやコピー機の制御
  - 自動車の制御システム
  - 携帯電話

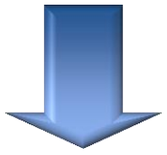
など…

組込みシステムの要件（従来からの要件）

- **リアルタイムシステム**
  - 計算処理よりも、入出力処理、通信処理が中心
  - モノを制御するため、高い応答性能が要求される
- **性能、サイズのチューニング**
  - （特にハードウェア）コストを極限まで下げる
  - 結果として厳しいリソース制約上でソフトウェア開発
  - コンパクトな実装
- **専用化されたシステム**
  - 必要のない機能を削除することでチューニング可能
- **高い信頼性**
  - 組込みシステムは、クリティカルな応用の場面も多い
  - ネットワークアップデートなどの仕組みがないものは、システムの改修に多大なコスト

組込みシステムの要件（新しい要件①）

- ユビキタス時代を迎え、組込み機器もネットワーク接続され、単体では動作しない。



- ネットワーク通信機能
  - サーバとの連携で、機能を実現
  - 他の製品やサービスとの連携
- セキュリティ
  - 暗号通信
  - 認証通信

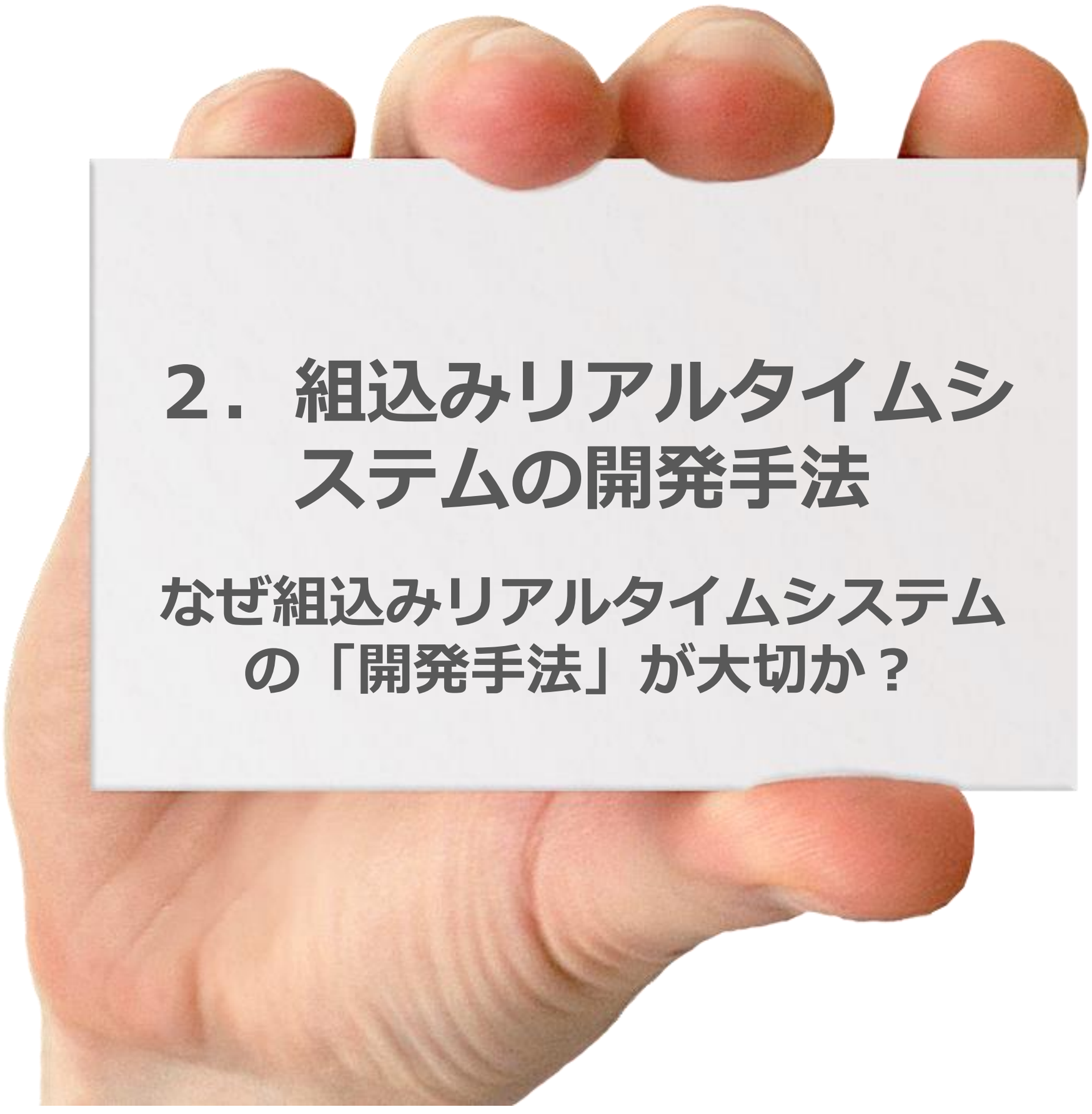
組込みシステムの要件（新しい要件②）

- **省資源・省電力**
  - 増えるモバイル型の組込み機器
  - バッテリーの持続時間は、製品競争力上重要
  - 世界的な省エネルギー意識の高まりから、重要な省電力

- **使いやすく、やさしい利用者インターフェース**
  - 幼児からお年寄りまで。

組込み型コンピュータで特に重要な実時間性

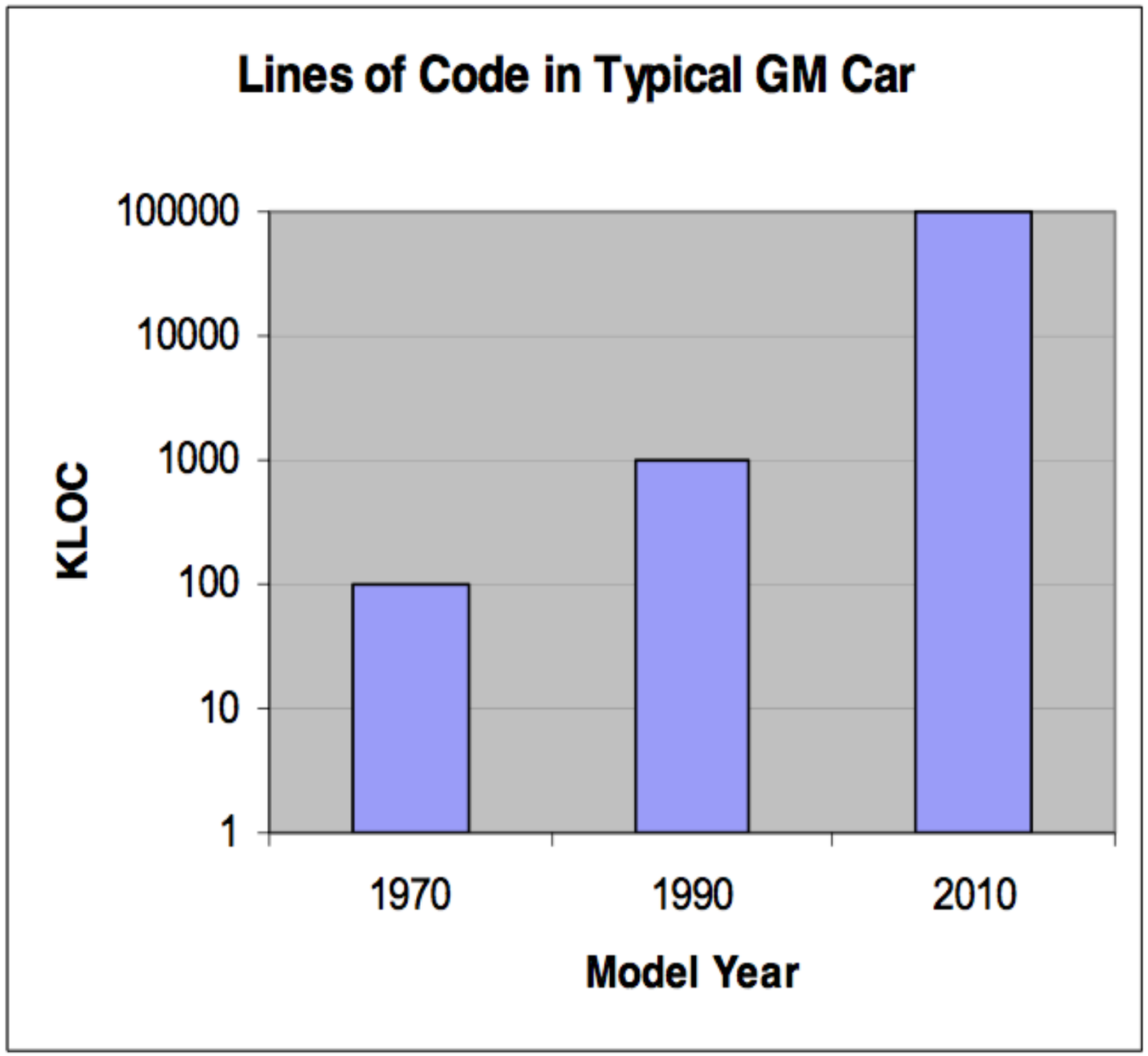
- **身の回りの「組込みコンピュータ」の仕事は？**
  - 給料計算や数値計算をするわけではない。
- **実世界の動きにあわせ、人間にサービス**
  - 実世界の時間に合わせて動作する
- **「リアルタイム（実時間）システム」（Real-time System）**



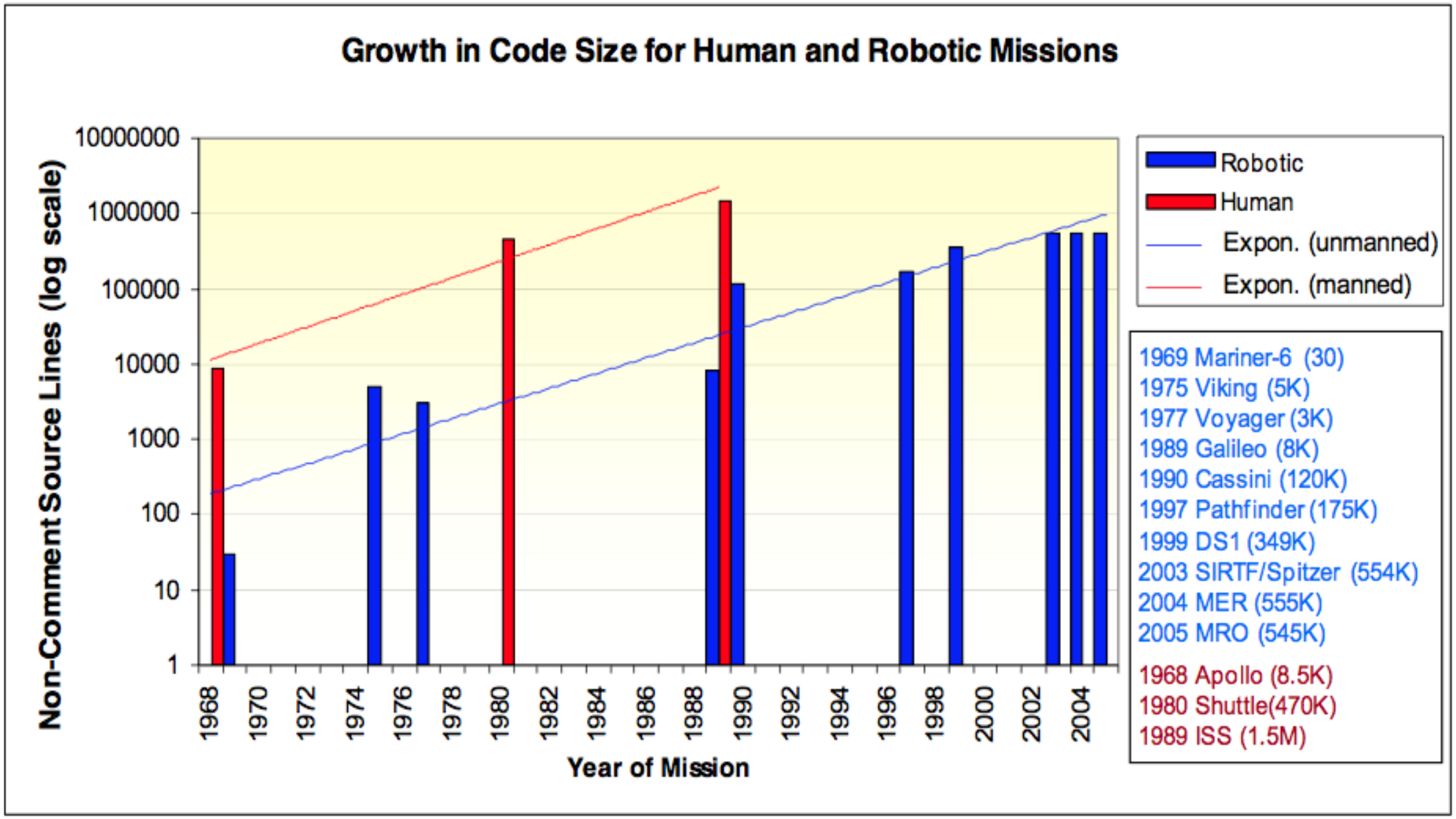
２．組込みリアルタイムシステムの開発手法

なぜ組込みリアルタイムシステムの「開発手法」が大切か？

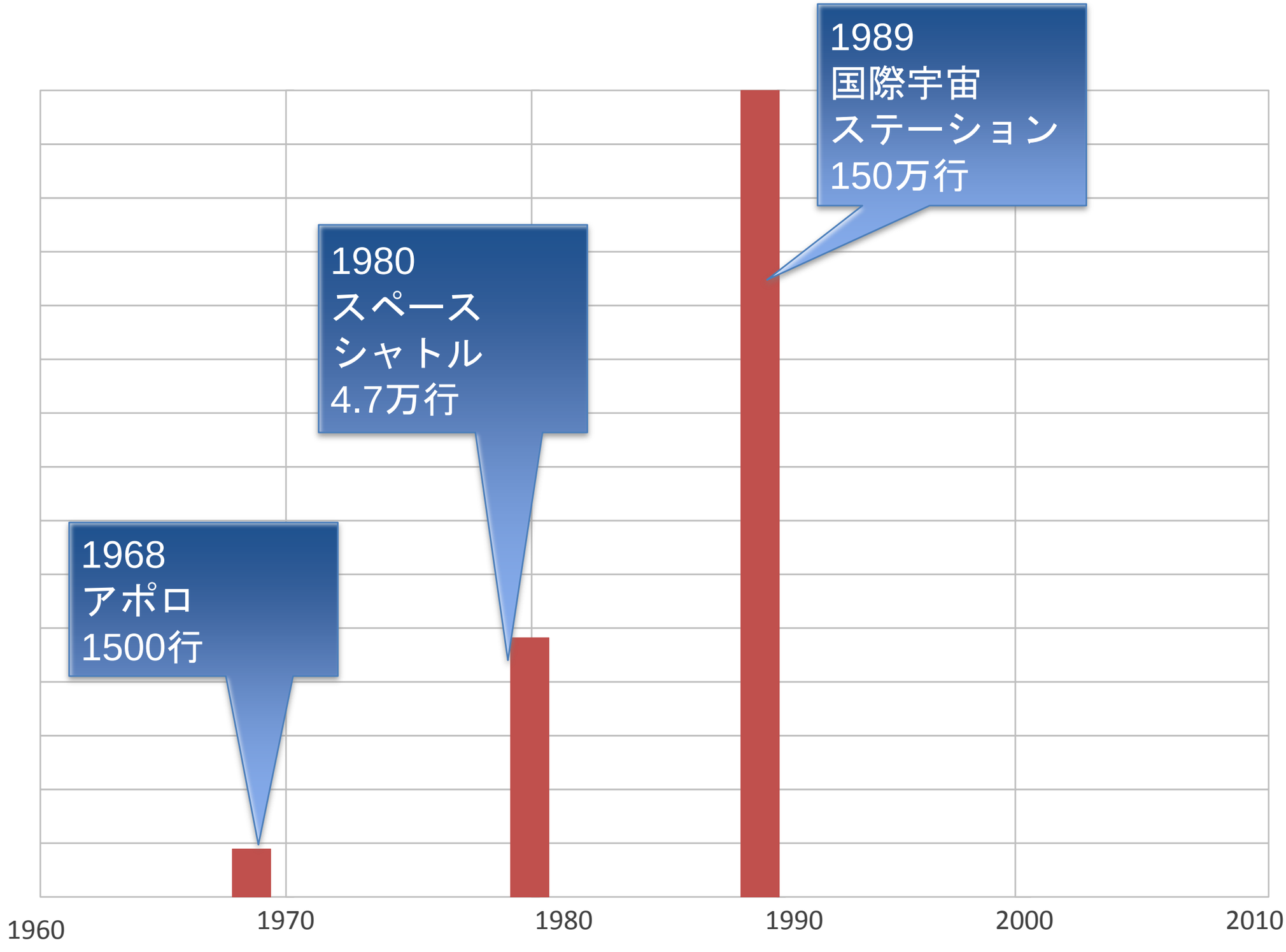
GMの自動車の制御ソフトウェア規模の増大



宇宙船のソフトウェアコードサイズの増大



宇宙船のソフトウェアコードサイズの増大（リアルスケール）

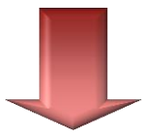


なぜ、組込みソフトウェア開発か？

- **組込機器の増大と多品種化、複雑化、大規模化**
  - 市場が大きくなったのはよいが、製品開発に求められる技術はますます高度化。

- **時代とともに、様々なニーズ**
  - 安全性・品質・信頼性、ユーザインタフェース（利用者）、国際性、コスト、省エネルギー、セキュリティー、など

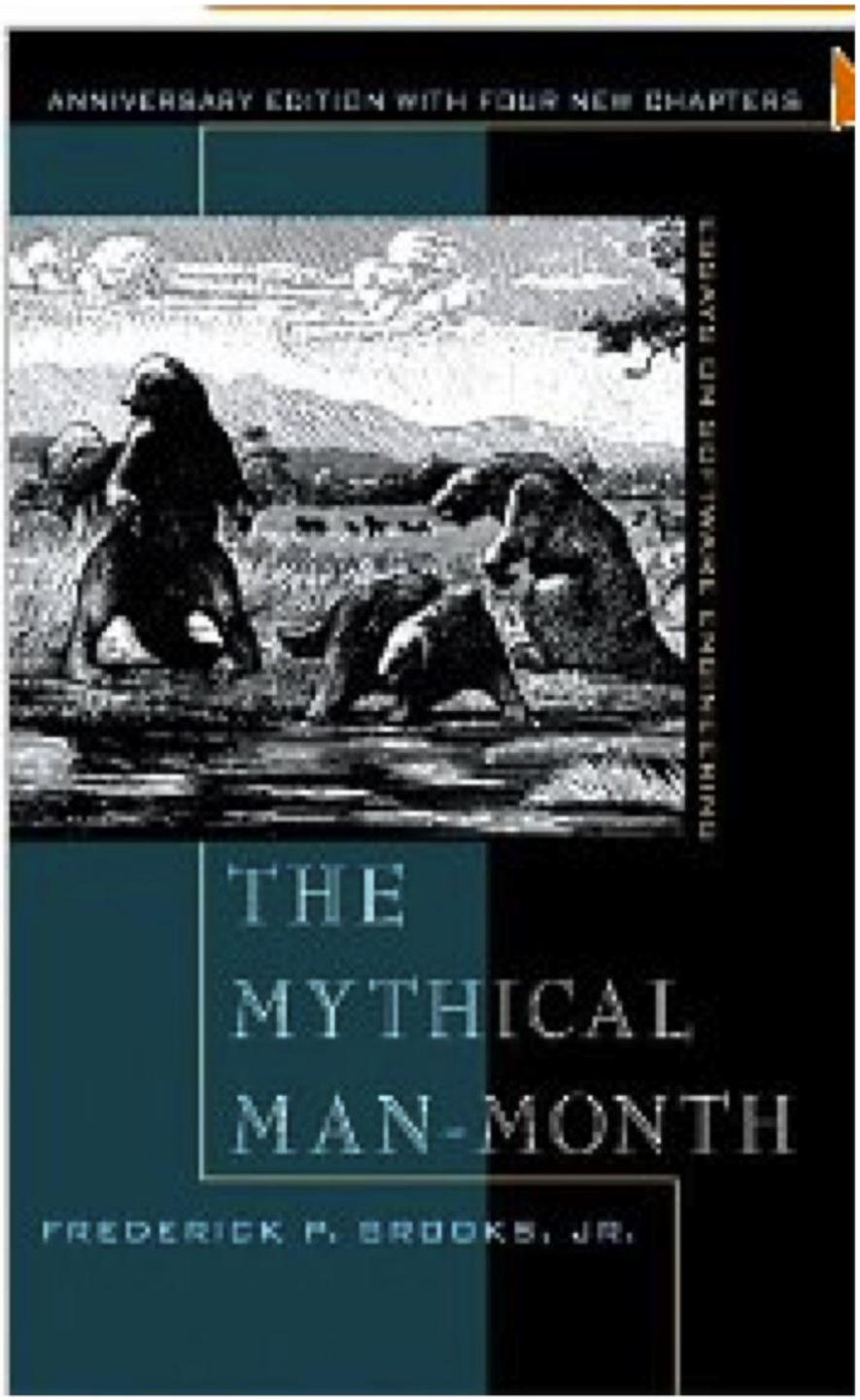
- **それにもかかわらず、開発サイクルも短縮化、製品価格の下落（開発費に圧縮）**



- **開発プロセスが重要**

“No silver bullet!” by Fred Brooks 1986

F.P.Brooks: “The Mythical Man-Month: Essays on Software Engineering”



“No silver bullet!”

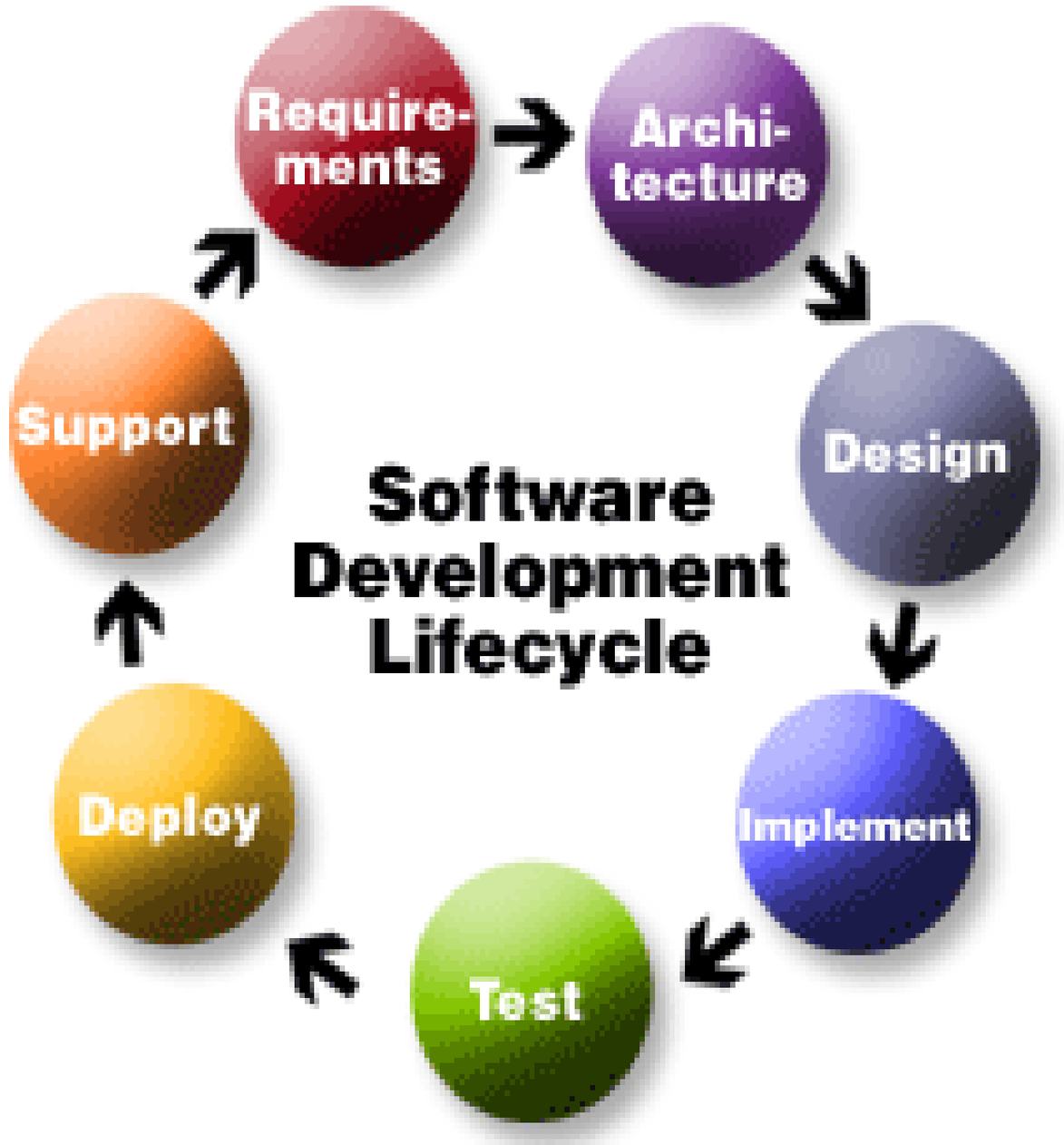
- **「ソフトウェアの生産性をひとりでにもたらすようなプログラミング技法は今後10年間は登場しない」**
  - 【出典】F. P. Brooks: “No Silver Bullet: Essence and Accidents of Software Engineering”, Addison-Wesley, 1995.
  - Silver Bullet=万能の方法論のこと
  - 狼男は通常の武器ではダメージを与えられず、銀で作成された武器だけが有効であるという伝説に由来

３．組込みソフトウェアの開発プロセス

3-1. ソフトウェア ライフサイクル

Software Life Cycle（ソフトウェアライフサイクル）

- ソフトウェアへの要求が発生してから、製作、修正、改良を経て、破棄されるまでの過程



ソフトウェアライフサイクルの工程

- **要求分析**
  - 発注者・顧客がソフトウェアで行いたい業務を明確にし、ソフトウェアが満たすべき機能を決定する工程
- **システム設計**
  - 要求を機能としてどのように実現するかを決定する工程
    - ユーザインタフェースはこの段階で決定されることが多い。
- **プログラム設計**
  - 機能をプログラムとしてどのように実現するかを決定する工程
- **コーディング／実装**
  - 設計に従ってプログラムを作成し、実行可能コードを生成する工程
    - マニュアル等のドキュメント類も、このとき同時に作成する。
- **テスト**
  - 顧客が意図とおりのプログラムができたか、およびプログラムが意図したとおりに動作するか検査する工程
    - 単体でのテストのほかに、他のプログラムと組み合わせたときのテストを行う必要もある。
- **保守**
  - 運用中に発見された不具合の修正や、機能・性能・使い易さの向上のためのシステム改良の工程

ソフトウェア開発全体の流れが重要

- **ソフトウェア開発の中での労力の比として…**
  - 要求分析・システム・プログラム設計：＿＿＿＿1／3
  - 実装・コーディング：＿＿＿＿＿＿＿＿＿＿＿＿1／6
  - テスト（単体試験）：＿＿＿＿＿＿＿＿＿＿＿＿1／4
  - テスト（結合試験）：＿＿＿＿＿＿＿＿＿＿＿＿1／4

  と経験的に言われている。

- **いろいろな言い方がある。**
  - 開発プロセスにおいて、テストの割合が予想外に大きいことを主張するケースが多い。

3-2. 組込み開発の特徴

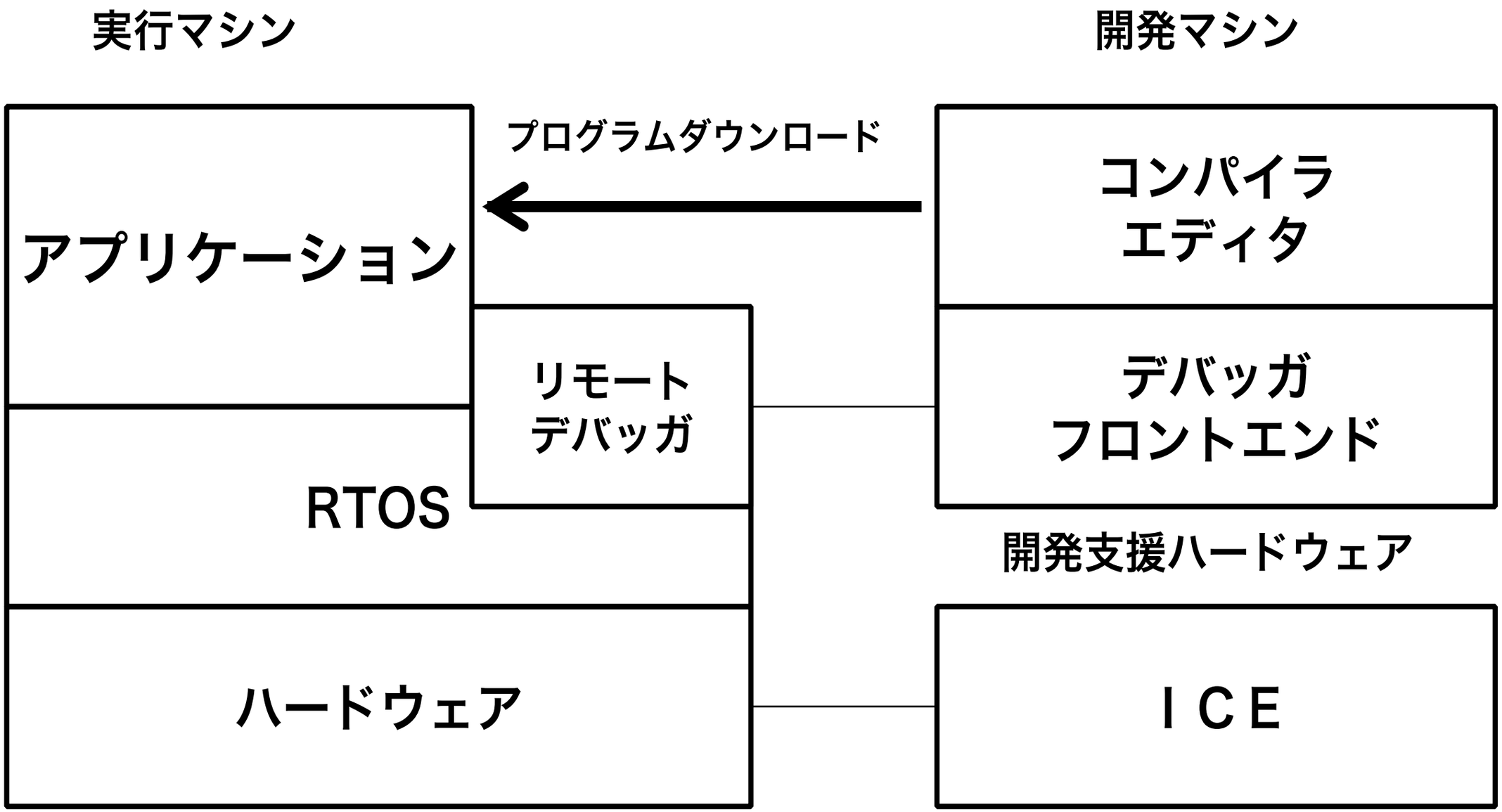
組込みシステム開発の特徴

- **ハードウェアとソフトウェアを同時並行で開発**
  - パソコン上でプログラムを書くときのように、完全にデバッグが終わったマシン上でプログラムを書くわけではない。
  - トラブル発生時 ➔ハードが原因？ソフトが原因？
- **プログラムを作るマシンと、作ったプログラムを走らせるマシンが、別々**
  - プログラムを作るマシン＝エディタやコンパイラを動かすマシン
  - プログラムを走らせるマシン＝作ったプログラムを動かすマシン
    ➔ これらが違うマシンなので、一旦プログラムを作ったら、実行マシンにダウンロードしてから動作

ソフトウェアとハードウェアのCo-design

- **応用に応じた専用ハードウェア**
  - 組込みシステムは専用システム
  - 応用にマッチするハードウェアに作り込むことで、コンパクトで低コストの製品
- **ASIC等のカスタムLSI技術の普及**
  - 今までよりも少量生産でも十分低コストな専用ハードウェアを構築可能

開発環境の概要

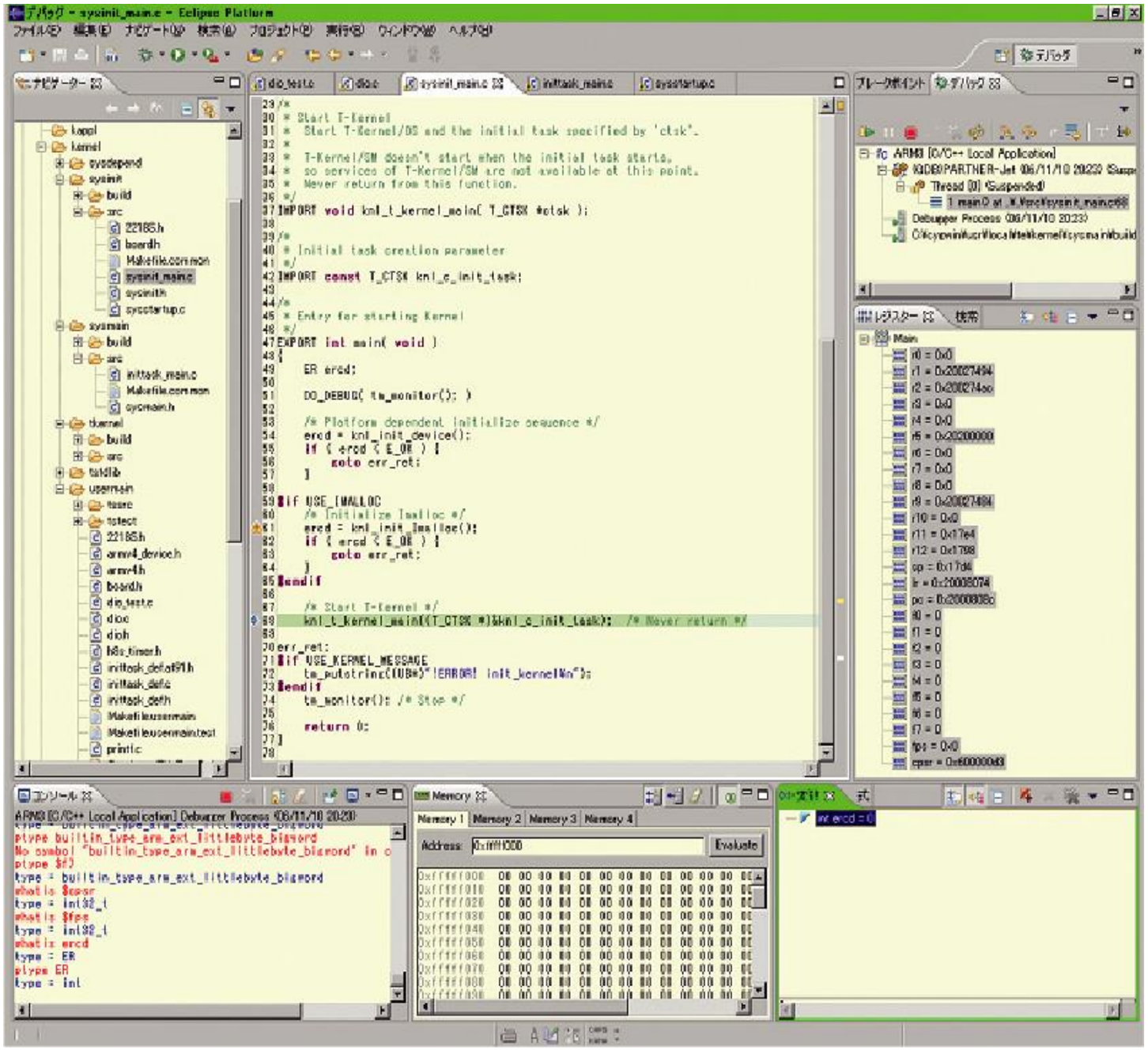


実行マシン例：Spansion FM3 USBボード

ICE: In Circuit Emulator

GUI統合開発環境の例

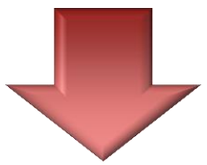
Eclipse の画面（デバッグ時）



４．開発全体の流れ

ソフトウェアにもいろいろある

- デバイスドライバやグラフィックルーチンのような部品
- 銀行のオンラインシステムのATM端末のように、業務がはっきりしているもの
  - しかも顧客は単体、締切厳格
- オペレーティングシステムのように、技術チャレンジがあり、試行錯誤するもの
  - しかも顧客は不特定多数、はっきりした締切はない
- パッケージソフトのように、不特定多数向けソフト



- 適切なプロセスは違う。

4-1. Waterfall Model

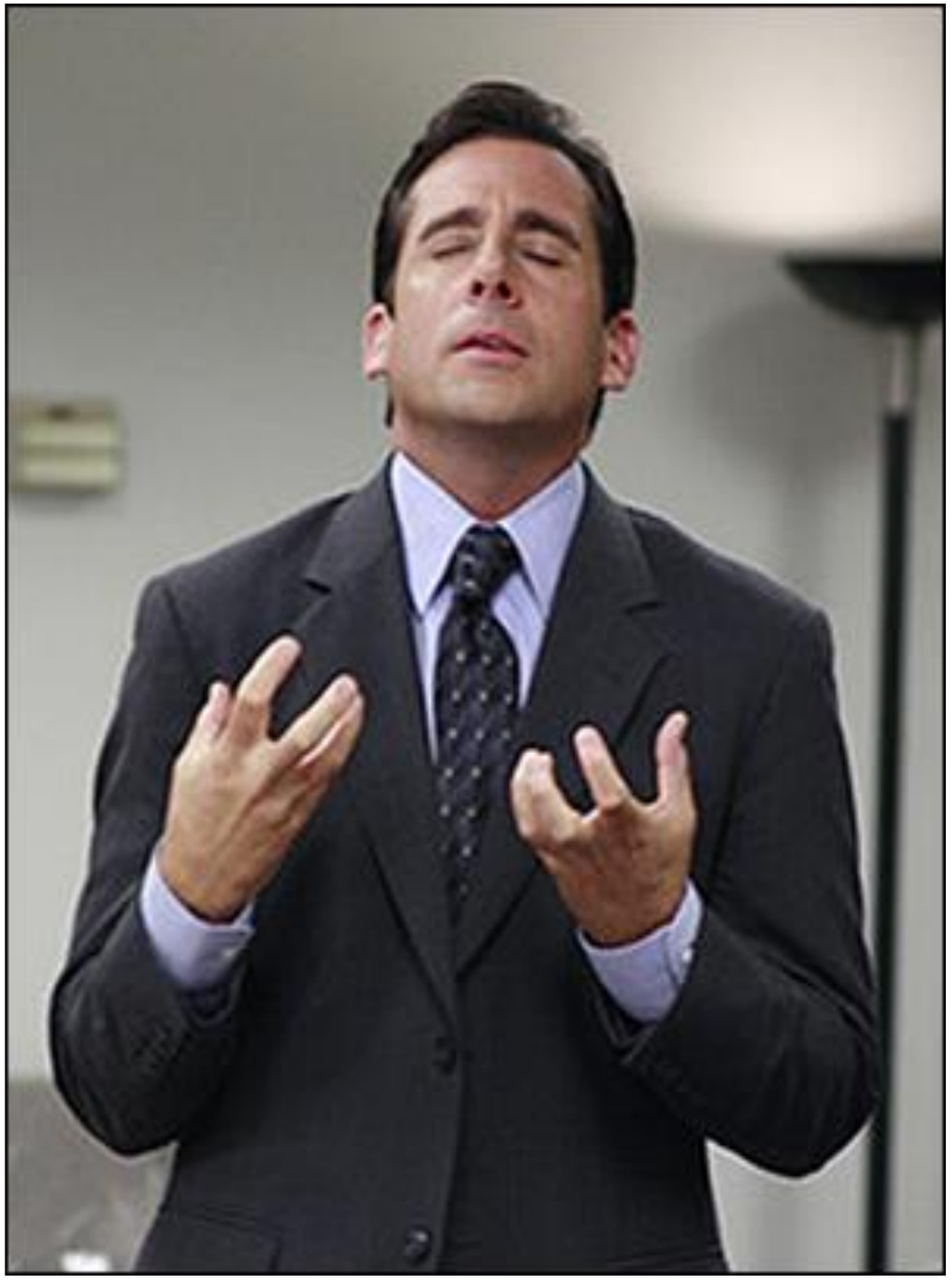
ウォーターフォールモデル

Waterfall Model [W. W. Royce, 1970]

- 滝から水が落ちるように、上流工程から下流工程へ実行が進化して、ソフトウエア製品が作成される。
- 古典的かつ基本的な考え方。

ウォーターフォールモデルは管理者好み

- マイルストーンが見やすい
- 一直線上に進むので、同時に一種類の作業が行われているので、わかりやすい
- 進捗チェックがしやすい
  - 例えば、、
  - 90%コーディングが完了した
  - 20% はテストが完了した
  - など



Waterfall Modelの問題点

- いくつかの点で実際のソフトウェア開発のプロセスと整合しない。

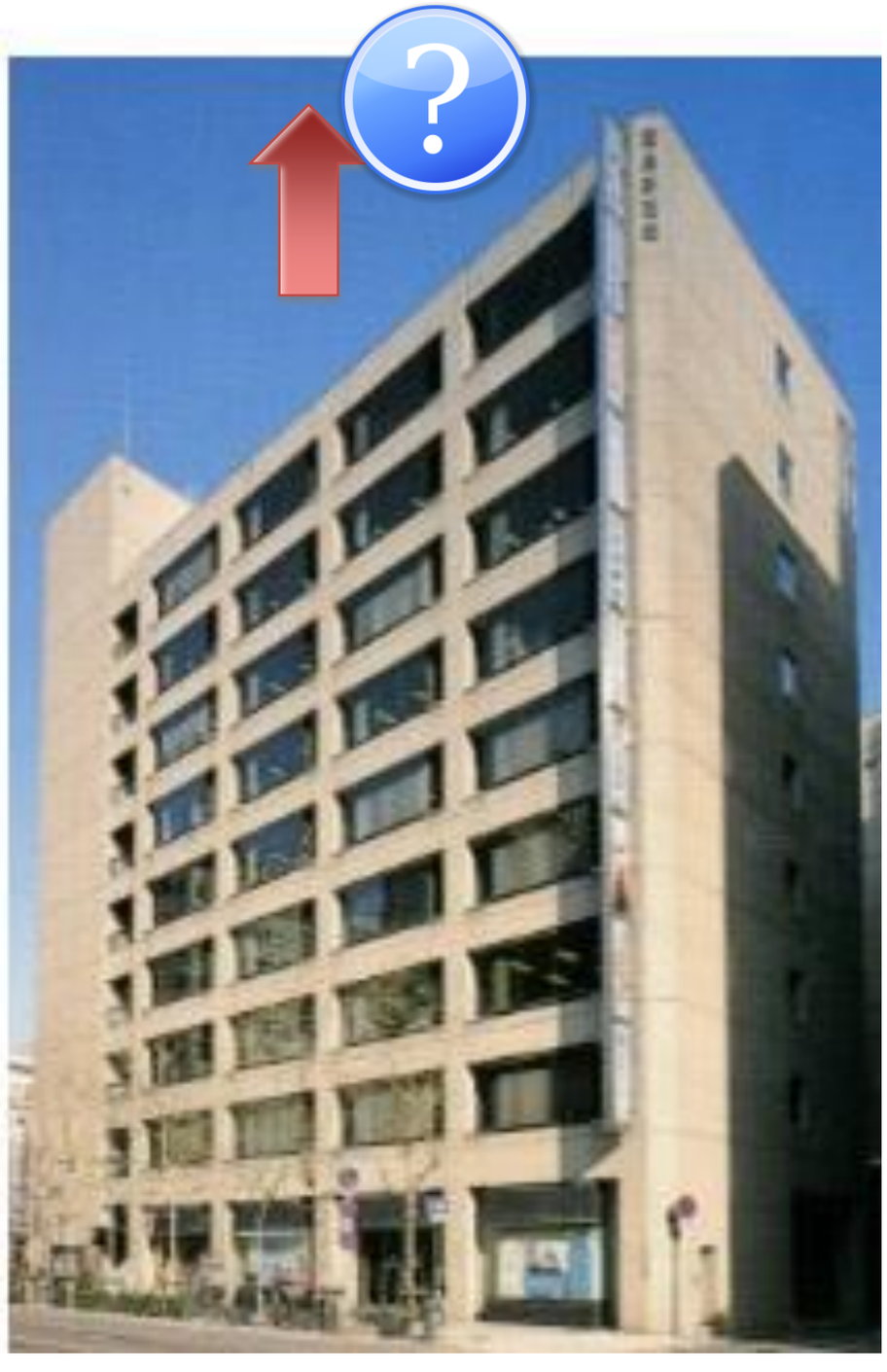
- プログラムが完成するまでテストが行われない。
- 設計、コーディングなどの途中で要求に不十分な点が見つかると、最初からやり直さなければならない。

「コーヒーブレイク」

- **ソフトウェアと建築における開発過程を比較すると、、**

- **新しい建物を建てる時、、**
  - 最初の計画＝ 9階建て
  - 9階建てで設計する
  - 工事が開始される。
  - 施主：やはり10階建てにしたい

- **実際の建物の建設において、こんなことはおこらない**
- **しかし、ソフトウェア開発では、しばしばおこる。**
  - なぜおこるのか？…結局できてしまうからか？



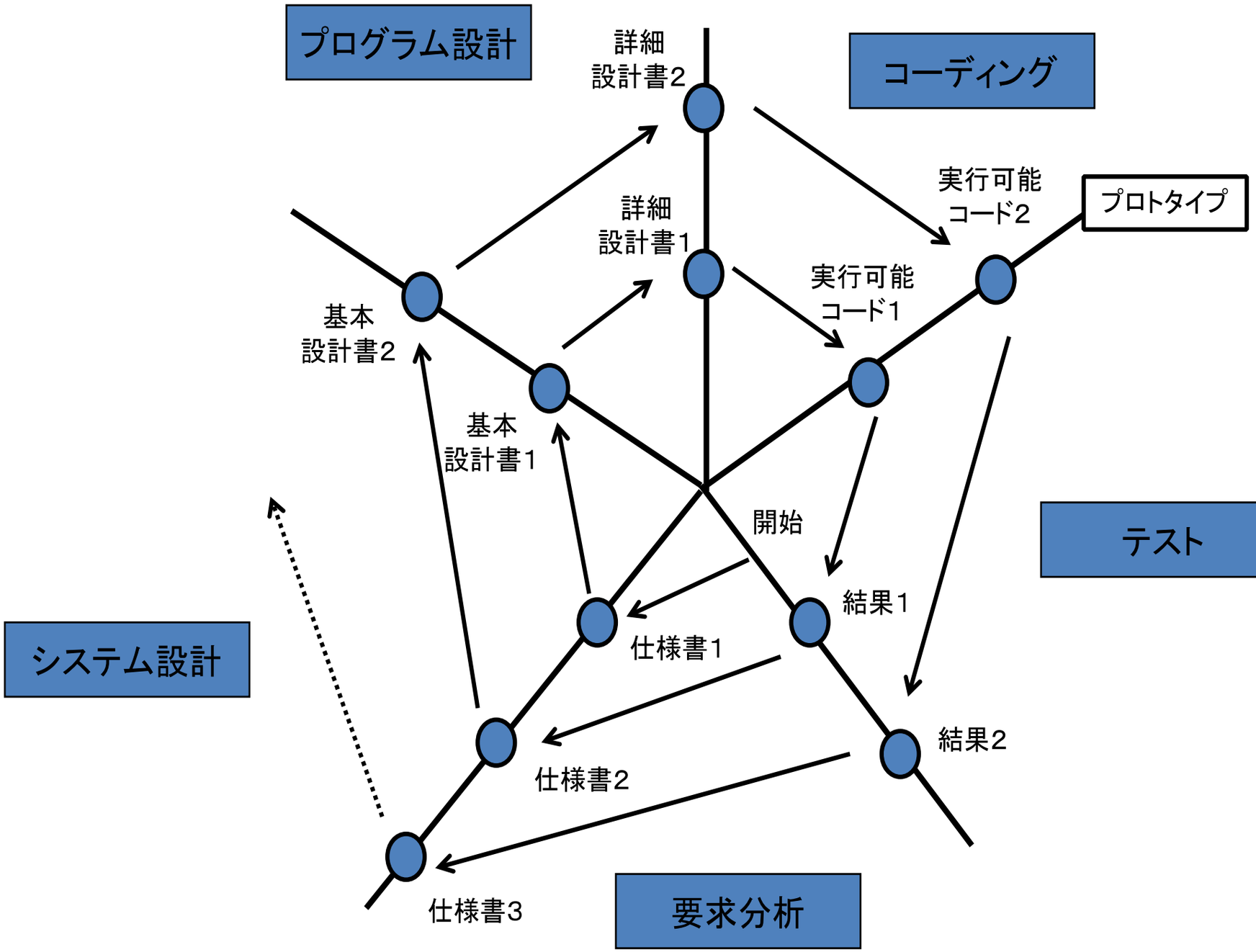
4-2. Spiral Model

スパイラルモデル

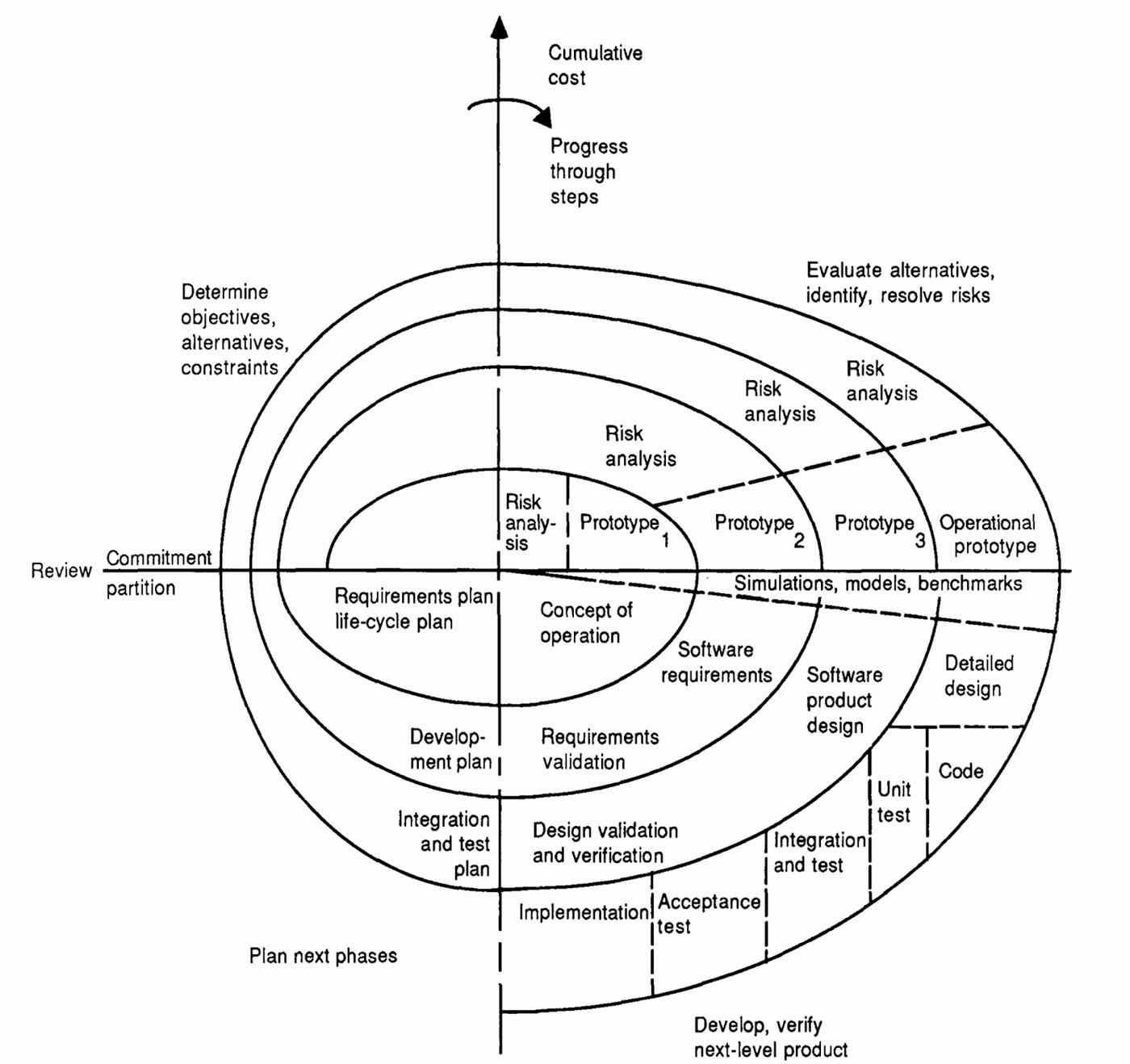
Spiral Model　[B. W. Boehm, 1988]

- **複雑な要求を全部満たすソフトウェアを、最初から作ることはしない。**
  - 要求調査をし、本質的な部分・実現が容易な部分に絞った要求（仕様書1）を作成して分析設計を行う（プロトタイピング／Prototyping）
  - これを反復して精度をあげる。
  - 想定している反復のループ期間は、6ヶ月から2年。
- **米軍のFuture Combat Systemの開発などに適用**

Spiral Model



Original Spiral Model（全体プロセス）



Spiral Modelの利点・欠点

- **利点**
  - プログラムの規模やスケジュールが予測しやすい。
  - ソフトウェア開発につきものの、要求仕様の変更に対応しやすい。
  - 設計工程がのびて実装に費やせる期間が短くなるということがおきにくい。
- **欠点**
  - 徐々に構築されるシステムなので、管理者にとっては全体の工程を管理しづらい。
  - システム開発全体のリスクを管理しにくい。

4-3. Unified Process

UP: Unified Process

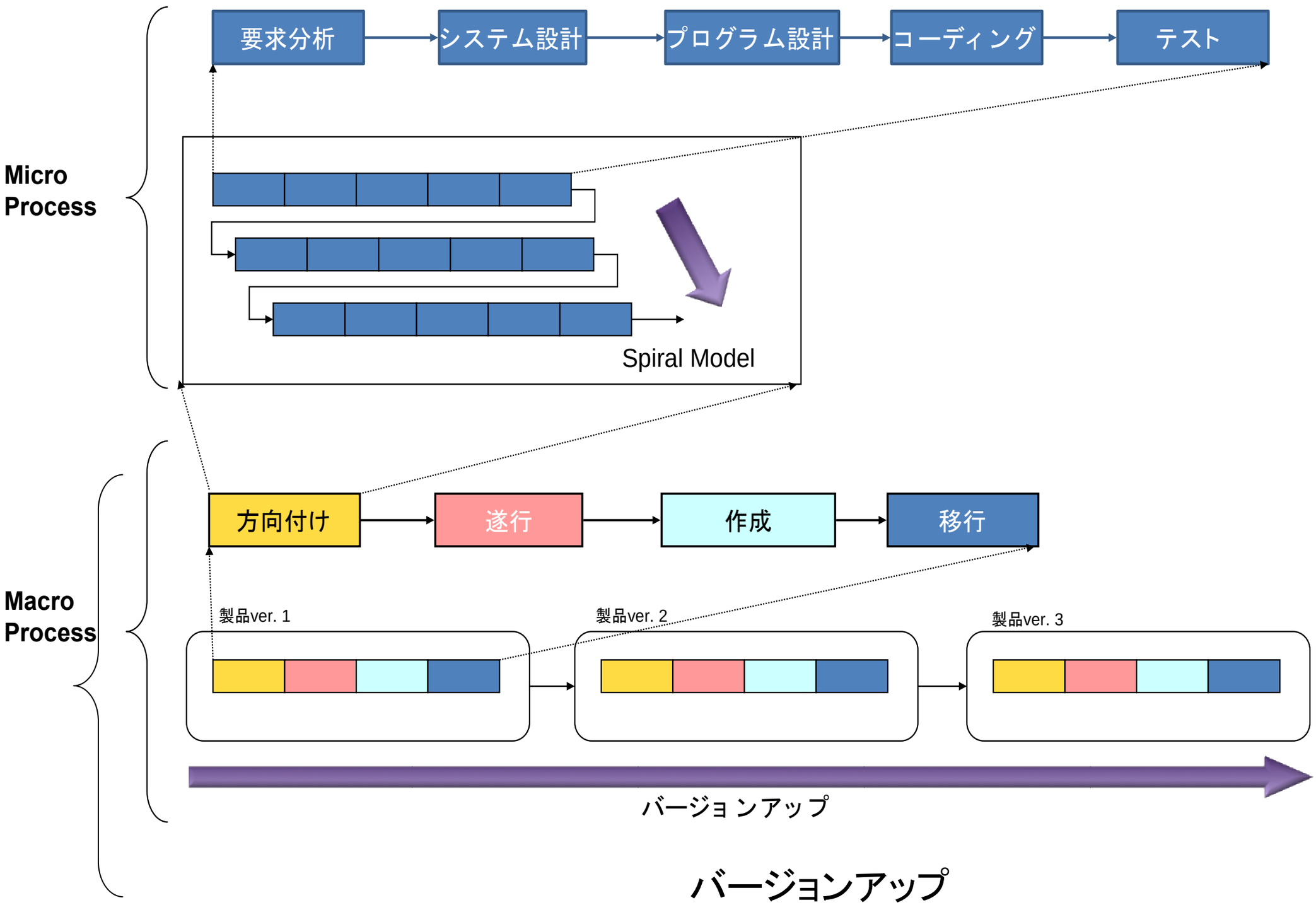
- マクロプロセスとマイクロプロセスからなる。

- マイクロプロセス
  - スパイラルモデルを採用
  - 要求分析・システム分析・設計・実装・テストのプロセスを繰り返す。
- マクロプロセス
  - フェーズと呼ばれる4つの工程を管理者の立場から管理する。
    - 方向付けフェーズ（Inception Phase）
    - 遂行フェーズ（Elaboration Phase）
    - 作成フェーズ（Construction Phase）
    - 移行フェーズ（Transition Phase）

UPのマクロプロセス

- **方向付けフェーズ（Inception Phase）**
  - どのようなソフトウェアを作成するか概念化を進める。
  - アイデアをプロトタイプを作成し、評価する。
- **遂行フェーズ（Elaboration Phase）**
  - システム構築の核となるアーキテクチャベースラインを作る。
  - 骨組と黄金ルート（必ず通る基本ルート）を作る。
- **作成フェーズ（Construction Phase）**
  - システムとして仕上げる。
  - もっとも工数をかける工程
  - β版のリリースが目標
- **移行フェーズ（Transition Phase）**
  - β版のリリースとフィールドでの評価結果の反映
  - 出荷の準備など、製品化のための作業を行う。

Unified Process



バージョンアップ

4-4. Agile Software Development

アジャイル型開発

Agile Software Development／アジャイル型ソフトウェア開発

- ソフトウェア工学において、迅速かつ適応的にソフトウェア開発を行う、軽量な開発手法群の総称。
- 開発対象を多数の小さな機能に分割。一つの反復（iteration）で一つの機能を開発する。
  - 反復では、1～4週間程度の短い期間単位を採用することで、リスクを最小化する。
- 各反復では、従来作成した成果物に更にもう一つの機能を追加する。
- 一つの反復の中で、計画・要求分析・設計・実装・テスト文書化といったソフトウェア開発の全工程を行う。

Agile Software Development適用が難しいケース

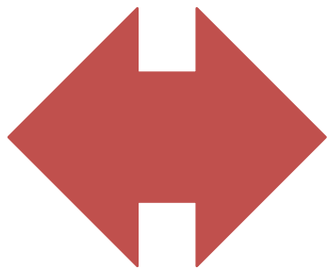
- 20人以上の大規模チームでの開発
- 地理的に分散した開発チームでの開発
- ミッションクリティカルなシステム、人命にかかわるシステムの開発
- 指示と管理を社風とする企業

4-5. まとめ

計画重視か？
適応重視か？

計画重視か、適応重視か？

- **計画重視型（ウォーターフォール型）が得意なケース**
  - クリティカルなシステム
  - 経験の少ない開発者が多数
  - 仕様変更があまりない
  - 開発者が多い
  - 秩序を重視する組織的文化



- **適応重視型（アジャイル型）が得意なケース**
  - クリティカルでない。
  - 熟練した開発者が参加
  - 頻繁に仕様変更がある
  - 開発者が少ない
  - 混沌とした状況への対処能力が高いチーム

5．要求分析

Structured Analysis（構造化分析）

- **ライフサイクルモデルの最上流における分析方法の一つ**
  - 要求要件を分析し、ソフトウェアアーキテクチャへ転換していく

- **データフロー図(Data Flow Diagram: DFD)**
  - システムが提供する機能をデータの流れをたどって直感に把握したすく記述する。
- **コンテキストダイアグラム（Context Diagram, 全体文脈図）**
  - プロセスが一つだけの特殊なDFD
  - 構造化分析に最初の段階で行う
- **データ辞書（Data Dictionary: DD）**
  - データの中身の記述
  - 連接（＋）、選択（｜）、繰り返し（＊）を用いてテキスト形式で記述したり、図で記述したりする。

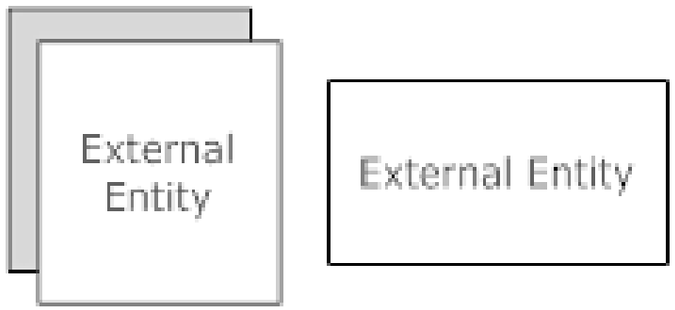
- **更に、階層的分解（Divide and Conquer）、モジュール分割などを行う**

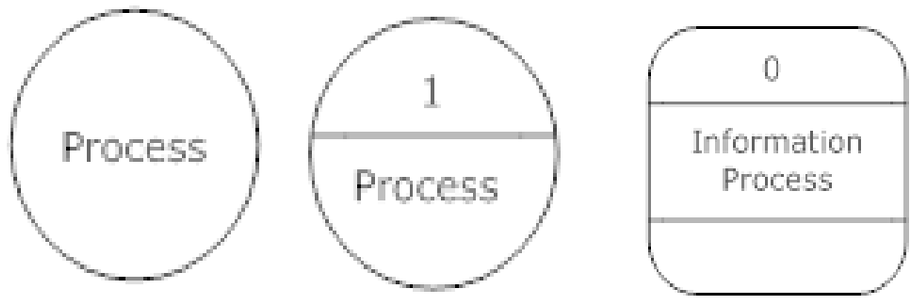
データフロー図（Data Flow Diagram）

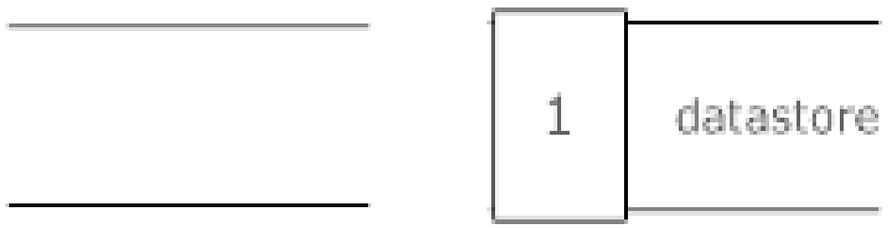
- **概要**
  - データフロー図は、情報システムのデータ（Data）の流れ（Flow）をグラフィカルに表現する図（Diagram）である。
  - データフロー図はデータ処理の可視化にも使われる。
- **使い方**
  - 一般に設計者は最初にシステムとシステム外部とのやり取りをコンテキストレベルのDFDで描く。その後コンテキストレベルのDFDを「詳細化」してシステムの細部をモデル化していく。
  - プロジェクトの出資者やエンドユーザーは、システム開発の全段階で概要説明や相談をうけるものである。その時には、以下のようにDFDを使うと効果がある。
    - データフロー図を使い、システムがどう実装され、何をどのように実行するのかをユーザーが眼で見でわかるように示す。
    - 従来システムのデータフロー図と新システムのデータフロー図を描いて比較し、どこが効率化されたかを即座にわからせる。
    - データフロー図を使って、エンドユーザーは自分たちが具体的にどういうデータをどこで入力するのかを思い描かせることができる。

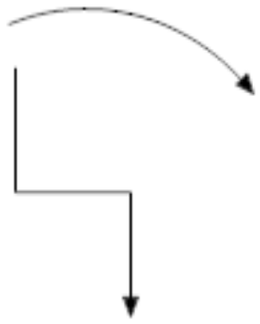
データフロー図（Data Flow Diagram）

- **外部実体／ターミネーター**
  - モデル化されるシステム外部に存在するもの。外部の情報源や外部への情報の出力先を意味する。
  - 四角形または楕円で表す（右図のInput/Output）
- **プロセス**
  - 入力データを処理してなんらかの加工を加え、結果となるデータを出力するもの。
  - 円または角を丸めた多角形で表される（右図のFunction）
- **データストア**
  - データの保管場所を表現したもの。
  - 2本の平行線で表される（右図のFile/Database）
- **データフロー**
  - 他の部品間のデータの移動経路を描いたもの。
  - 矢印線で表される（右図のFlow）

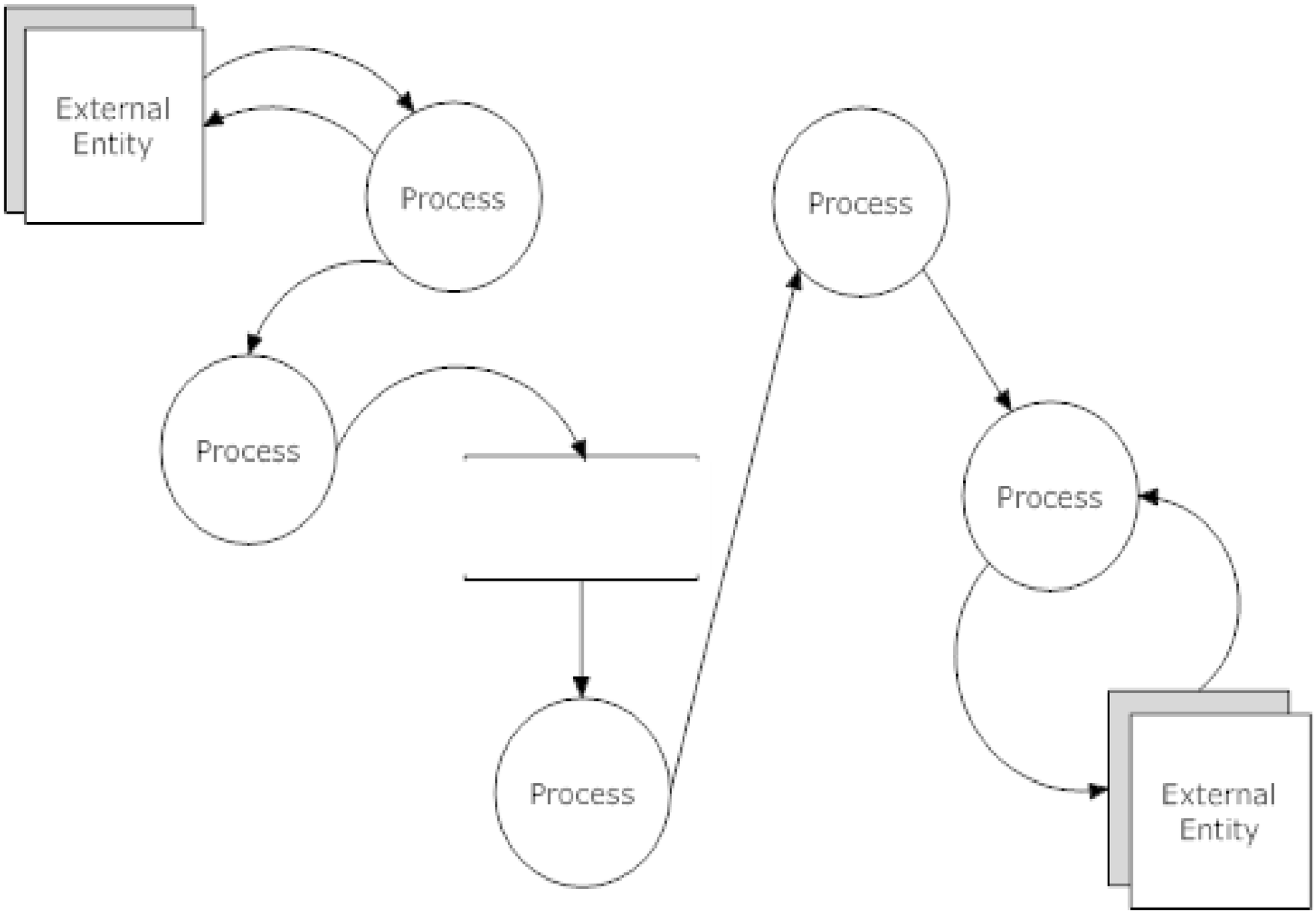




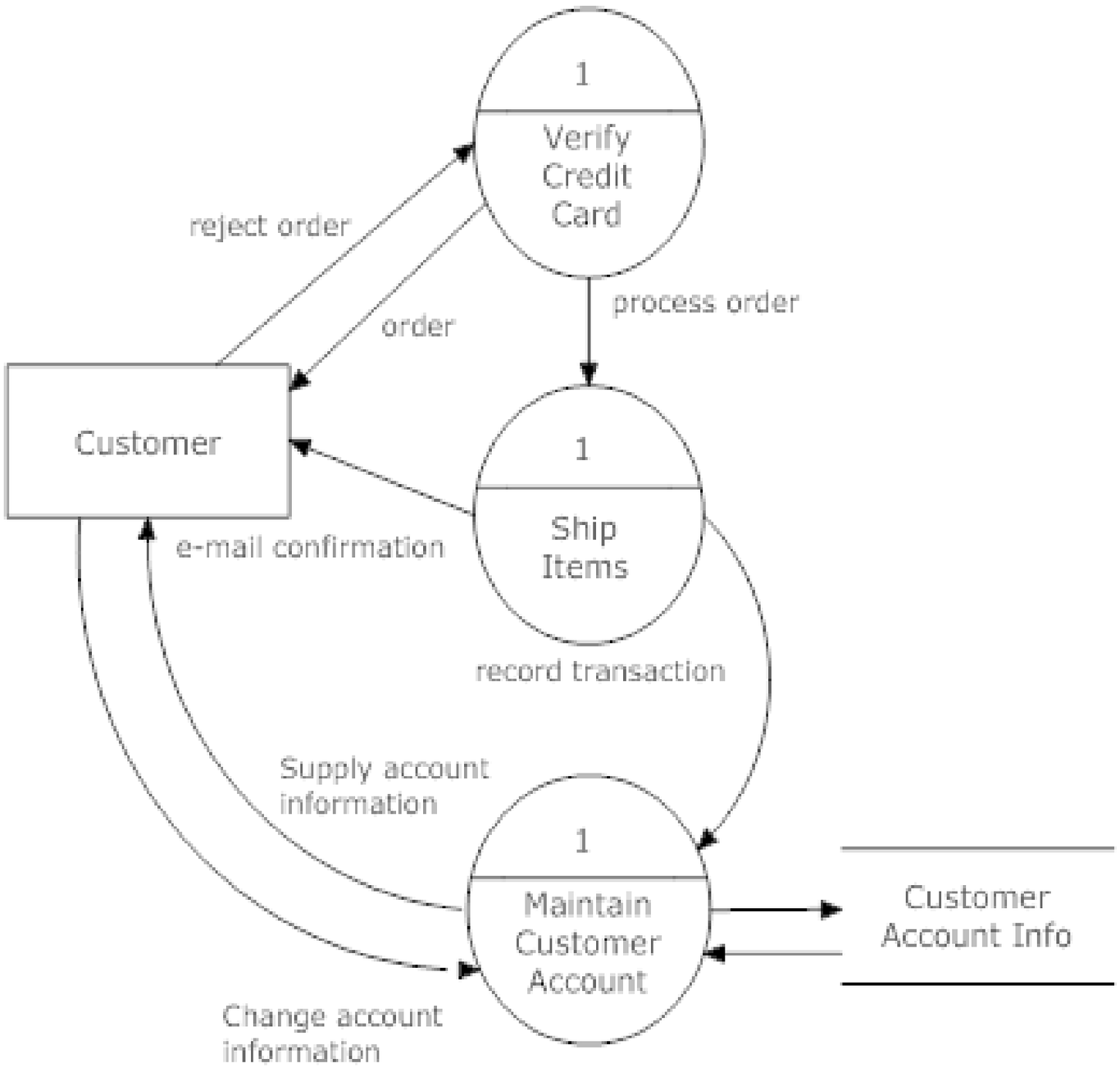




データフロー図（例１）



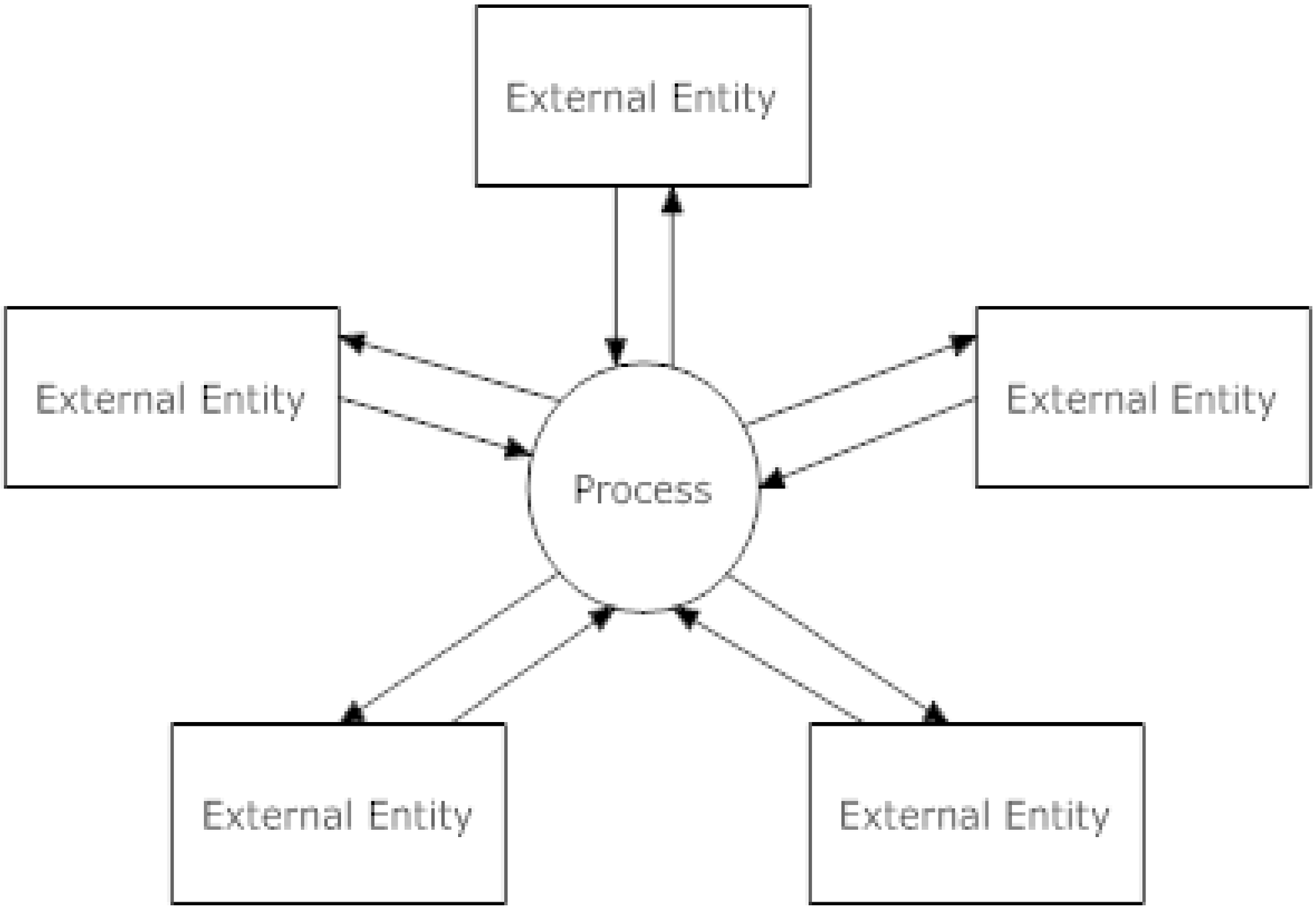
データフロー図（例2）



コンテキストダイヤグラム（Context Diagram）

- **概要**
  - 「トップレベル」・データ・フロー・ダイアグラムともいい，作ろうとしているシステムが何をしようとしているのかを明らかにした図。
- **使い方**
  - プロセスが一つだけの特殊なDFD
  - 構造化分析に最初の段階で行う

コンテキストダイヤグラムの例



データ辞書（Data Dictionary）

- **データ辞書（data dictionary）、「意味、他のデータとの関係、起源、用途、フォーマットなどのデータに関する情報を集中的に保管したもの」である。**
  - データベースやデータベース群を解説した文書
  - DBMSの構造を決定するのに必要な必須コンポーネント
  - DBMS固有のデータ辞書を拡張または代替するミドルウェア

データ辞書（正規表現した例）

name = courtesy-title + first-name + (middle-name) + last-name

courtesy-title = [Mr. | Miss | Mrs. | Ms. | Dr. | Professor]

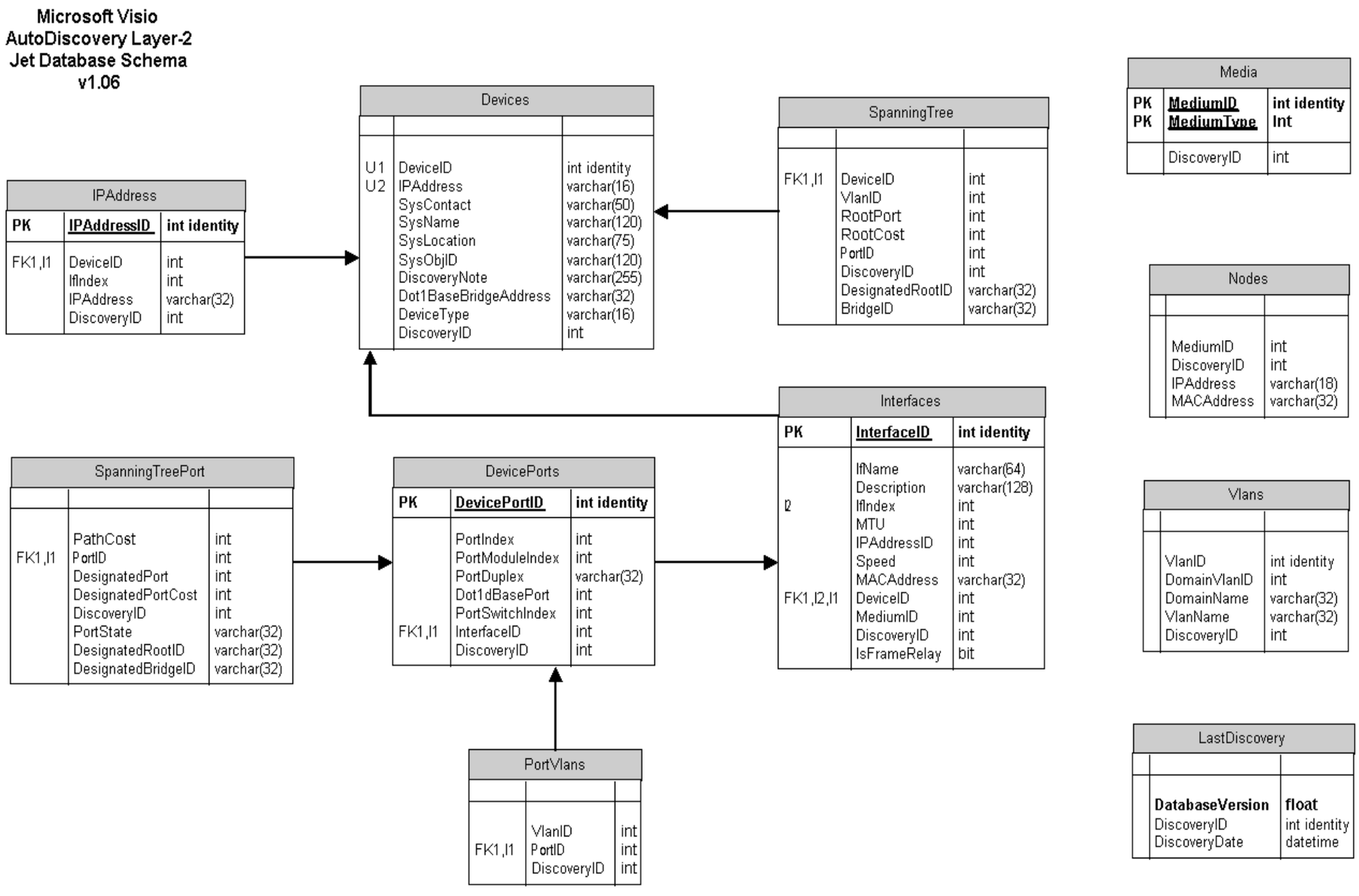
first-name = {legal-character}

middle-name = {legal-character}

last-name = {legal-character}

legal-character = [A-Z|a-z|0-9|'|-| |]

データ辞書（図形表現した例）



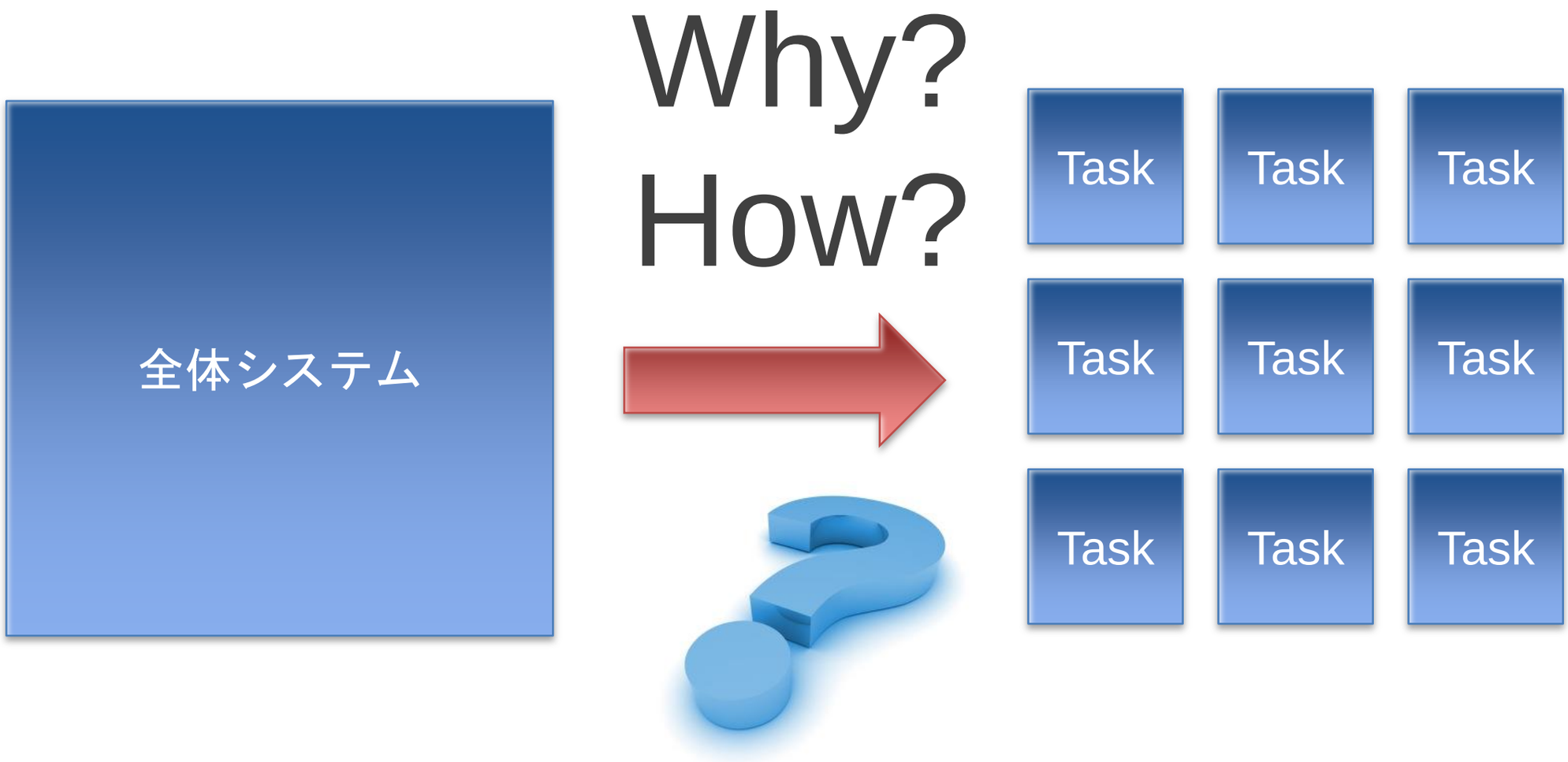
6．設計

設計手法の内容

- 要求定義内容の確認
- 業務のサブシステム化
- 入力設計、出力設計
- コード設計　→ “モジュール化”
- 論理データ設計
- 外部設計書の作成
- ユーザインタフェース設計

組込み設計手法
モジュール分割／タスク分割

モジュール分割



モジュール分割の尺度

- **モジュールサイズ**
  - ソースコードサイズ（たとえば、100行程度とか）
- **モジュールの遮断度合い**
  - モジュールの内部設計が外部から見えない度合い
- **モジュール間結合度**
  - モジュール間でデータの共有度合い
- **モジュール強度**
  - モジュール間で機能を必要とする度合い

モジュール化の原則（上ほど、結合強度が強い）

- **機能的結合**
  - 一つの機能を実行するために関連し合っている要素を結合
- **情報的結合**
  - 一つの情報（データ構造）を扱う複数の機能を結合（情報の局所化、オブジェクト指向）
- **連絡的結合**
  - 同じデータを参照したり、データの受け渡しがある要素を結合
- **手順的結合**
  - 問題を処理するために関係している複数の機能のいくつかを結合
    - フローチャートの一部を結合する、など
- **時間的結合**
  - 実行される時期が同じ要素を結合
    - 初期設定モジュール、終了モジュール、など
- **論理的結合**
  - 論理的・抽象的に関係ある要素の結合
    - 入出力用モジュール、編集モジュール、など
- **暗合的結合**
  - 特別な関係のない要素を結合。
    - たまたま重複したコードを結合する、など
    - チューニングする際などでみられる

モジュール分割手法

分割方法	技法名	方法
データの流れに着目した分割技法（処理中心）	STS分割	データの入力処理（Source）、データの変換処理（Transform）、データの出力処理（Sink）の3種類のモジュールに分割してモジュールの構造化を行う。
	トランザクション分割	モジュール分割技法のひとつ。データの種類により実行する処理（トランザクション）が決まるような場合に、トランザクションの種類ごとにモジュール分割を行う技法。
データ構造に着目した分割技法（データ中心）	ジャクソン法	基本、連続、繰り返し、選択の4つの図式を用いて、データ構造とプログラム構造を階層化する。
	ワーニエ法	入出力データの構造に着目して、データが「いつ、どこで、何回」使われるかをもとに、順次・選択・繰り返しの制御構造でプログラムを展開する。
オブジェクトという単位に着目した分割技法	オブジェクト指向設計	

タスク分割ガイドライン

タスク分割の考え方

- タスク分割方法は、設計者によっていろいろ考え方がある。
- 考え方の例を示すが、これが唯一の方法ではない。
- とにかく柔軟に考え、いろいろな分割候補の中から最適なものを選ぶようにすることが必要。
  - どれが最適かは、設計者が判断しなくてはならない。

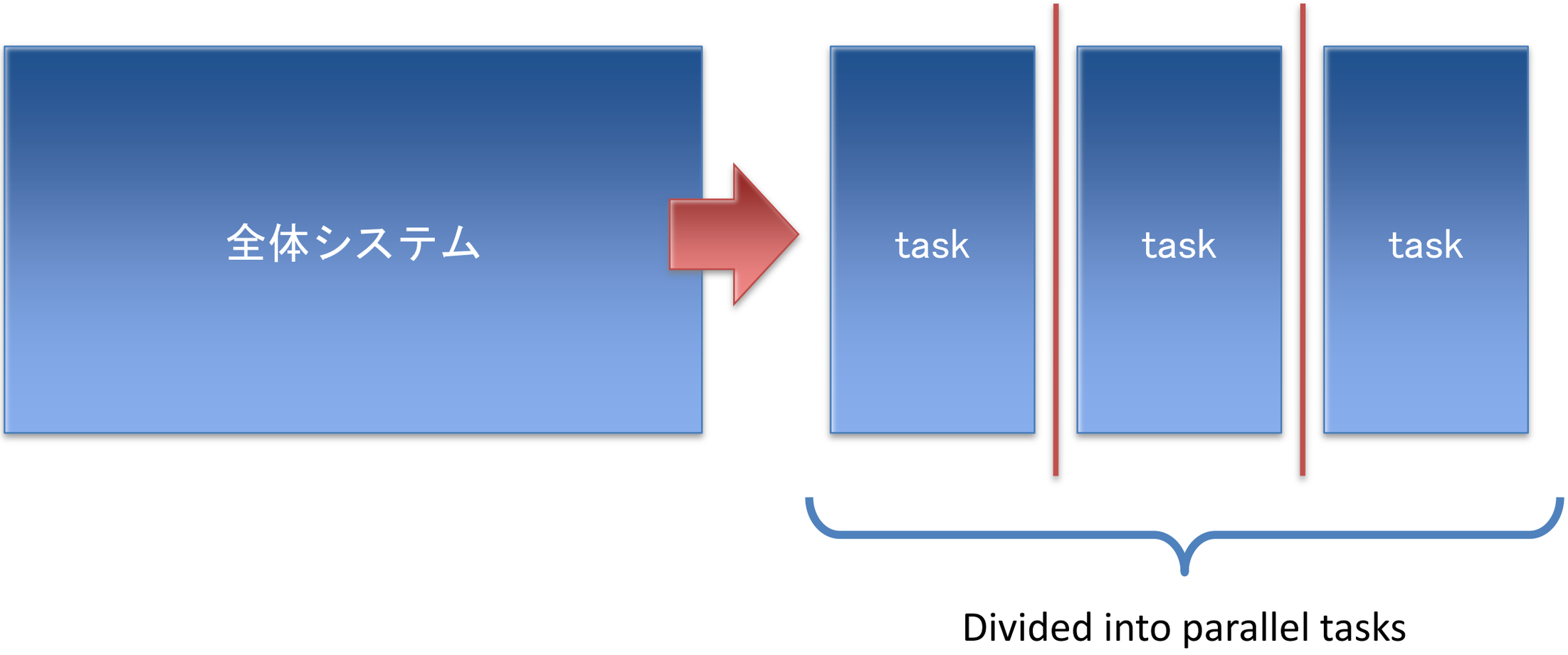
タスク分割は役割分割

- **タスク分割作業(設計作業)は、どの機能をどの処理で行うかを決めるための作業**
  - それぞれの役割を決めることによって、全体が動作する
  - 必要な割込みハンドラは、ハードウェアの構成で決まってしまう
- **どの機能をタスクで行い、どの機能をハンドラで行うのかも一緒に考える**
  - 処理の構成や目的によって異なるが、ハンドラで処理を完結させたほうが良い場合もある

並行動作に着目

- **まずは並行して動作すべき機能に着目**
  - 並行して動作すべき機能が、まずはタスクの候補
- **機能の種類が違う場合には、並行して動作する必要がない場合もある**
  - 別のタスクにまとめられるかどうかを検討する
  - シーケンシャルに動作させた方が良いものもある
- **並行動作や優先動作の検討をもとに、タスクとして明確化する＝設計作業**
  - 文書化しておく。
- **あいまいなままだと、その後の実装やテストがうまくいかない場合が出てくる。**
- **機能によっては、タスクで動作させずに、関数ライブラリでよい場合もある**

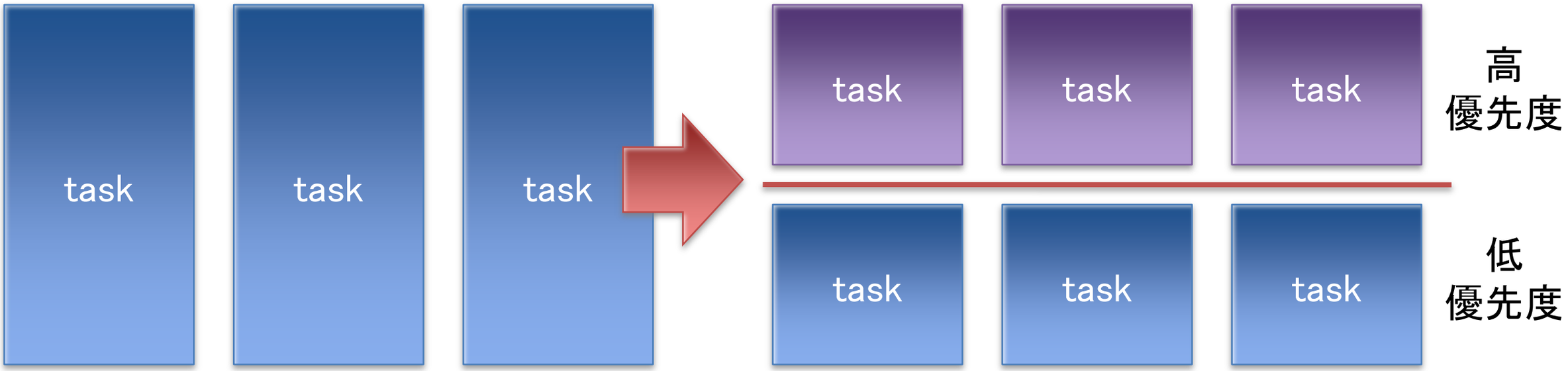
(1) 並行動作するタスクに分割



優先機能に着目

- **次に優先的に処理すべき機能に着目**
  - 非常停止や通信などで、システムに求められている機能をもとに検討する
- **優先すべき機能は、システムによって要求が違うので、注意が必要**
  - たとえば同じネットワーク機器でも、ルータとネットワークプリンタでは、優先すべき処理がちがう

(2) 優先度の観点からタスク分割する



タスク間の情報伝達に着目

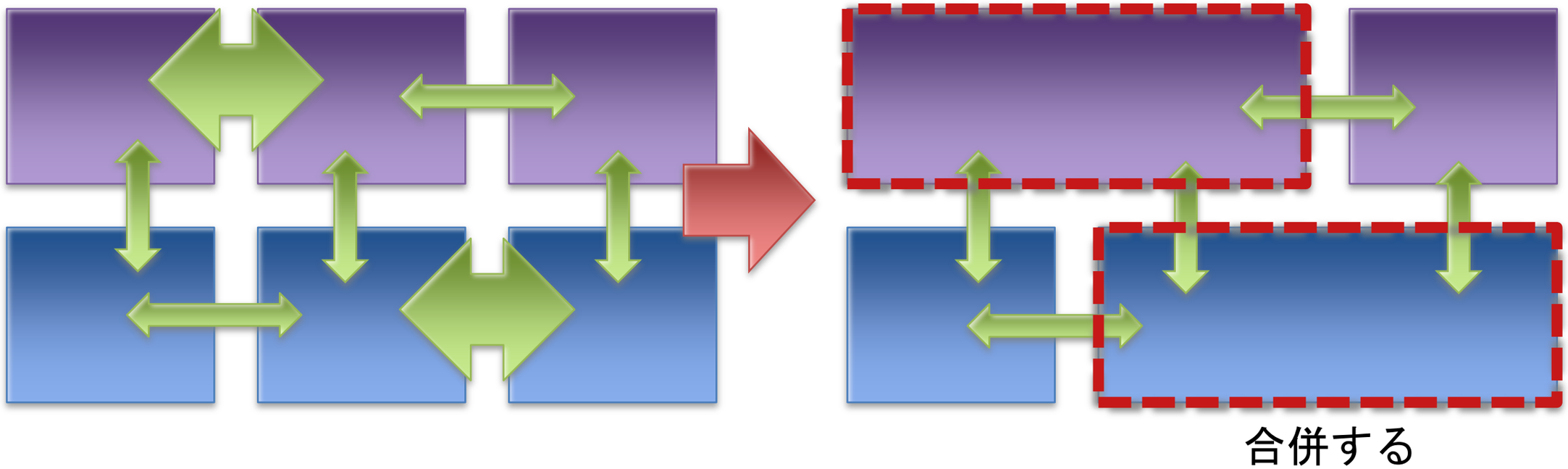
- **タスク間で、どのような機能を使って情報伝達(同期や通信)を行うのか検討する。**
  - RTOSの同期通信機能を使うのか？
  - グローバル領域を使うのか？
  - ハードウェアの機能を使うのか？
- **選択した情報伝達手段で、問題が無いかどうかを検討する。**
  - 複数の機能を同時に使えない場合もある。

(3) 強く結合しているタスクを合併する

通信分析

タスク間通信が多いタスクを発見する

合併する

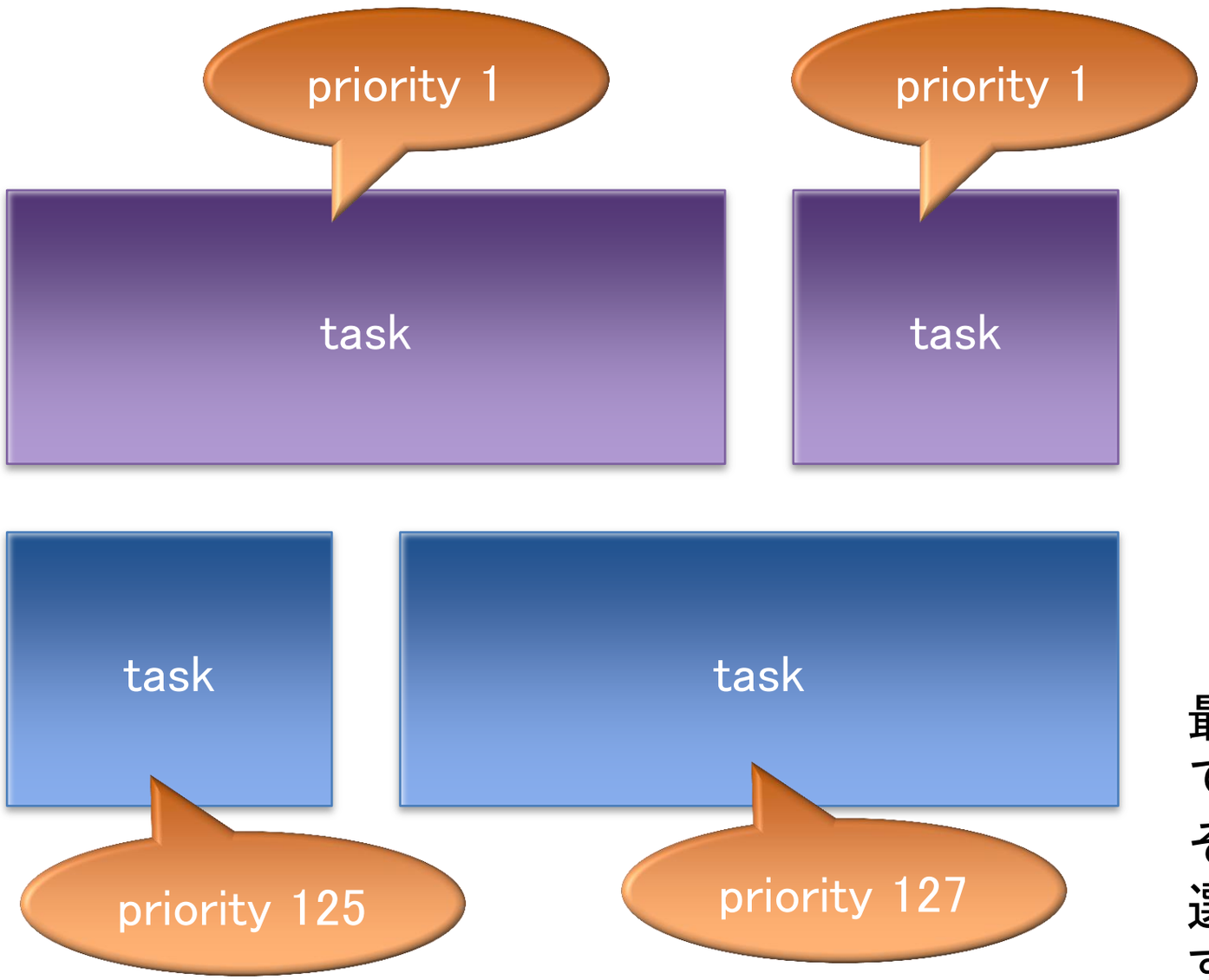


合併する

優先度の設定はあとで

- **まずすべてのタスクを同一優先度とする。**
  - 基準をそろえる。
- **優先的に処理すべきタスクの優先度を高くする。**
  - すべてのタスクが「優先的」という考えでは、うまくいかない。
- **「犠牲」にしてよいタスクの優先度を低くする。**
  - アイドルタスクやバックグラウンド処理タスクなど。

(4) 優先度を与える



最初は、同じ優先度をすべてのタスクに対等に与える
それから、特別なタスクを選んで、高い優先度を付与する。

考え方についての注意

- **考え方(着目点)の違いで、タスク分割方法も異なる。**
  - いわゆる「正解」はない。
- **OSやミドルウェアが持っている機能によっては、タスク分割の考え方が異なる場合もある。**
  - 例えば、BSDソケットのタスク(プロセス)分割の考え方はUNIX向きだが、小さな組込みシステムにはあまり向かない。

適度な大きさと量に注意

- **あまり細かくタスク分割しない。**
  - タスク切替えにも、時間がかかる。
- **タスク規模(コード量や処理機能量)をそろえた方が見通しが良い。**
  - 管理しやすくなる。
- **一概にこの程度とは言えないので、難しい問題ではある。**

タスクはイベントドリブンで

- **タスクはできるかぎり、イベントが発生してから動作を開始するように設計する。**
  - イベントが無い間は待ち状態
- **どのイベントが発生した時に、どのタスクが動作するのか、整理しておく。**
  - 同時にイベントが発生した時には、RTOSが優先度を使って判断する。

ケーススタディ

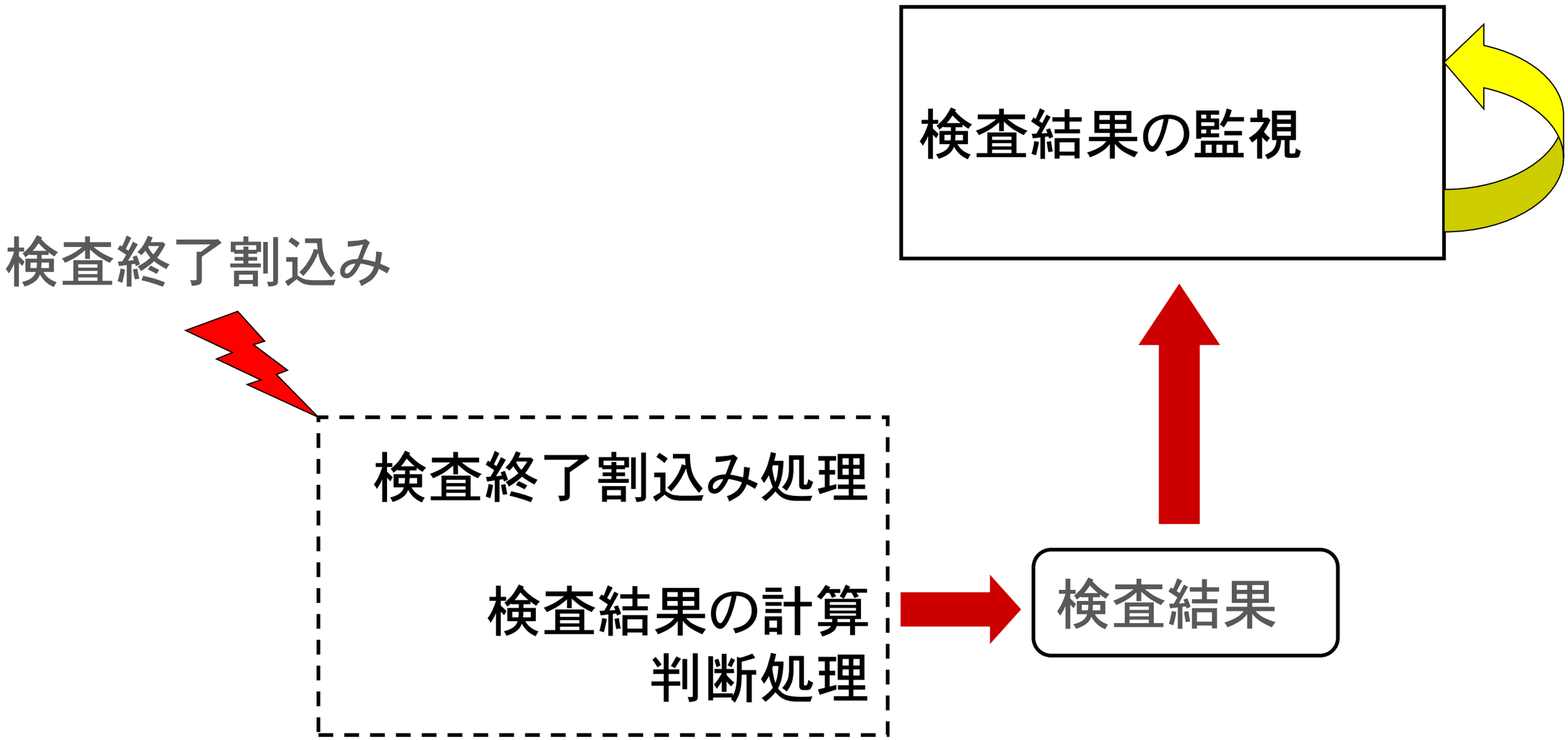
①処理の目的と内容

- 工場の製造ラインで、製造物の検査を行う機器
- 検査結果によって、製造物の合否を判断する。
- 検査結果全体のばらつき(標準偏差)が大きい時は、製造ラインを停止する。
- 試作を行った後、別途実機用のプログラムを開発する形で、プロジェクトを進める。

②Task分割（役割分割）の悪い例

- 標準偏差の計算や判断は、検査終了割込みがきっかけになるので、すべて割込み処理で行った。
- メインタスクは、計算結果および判断段結果を受け取るのみ。
- 試作段階では、これでも問題はなかった。

②Task分割（役割分割）の悪い例



③問題点は？

- 計算処理に時間がかかるため、他の割込みを取りこぼす。
- 優先度が高いタスクに切替わるタイミングが遅い。

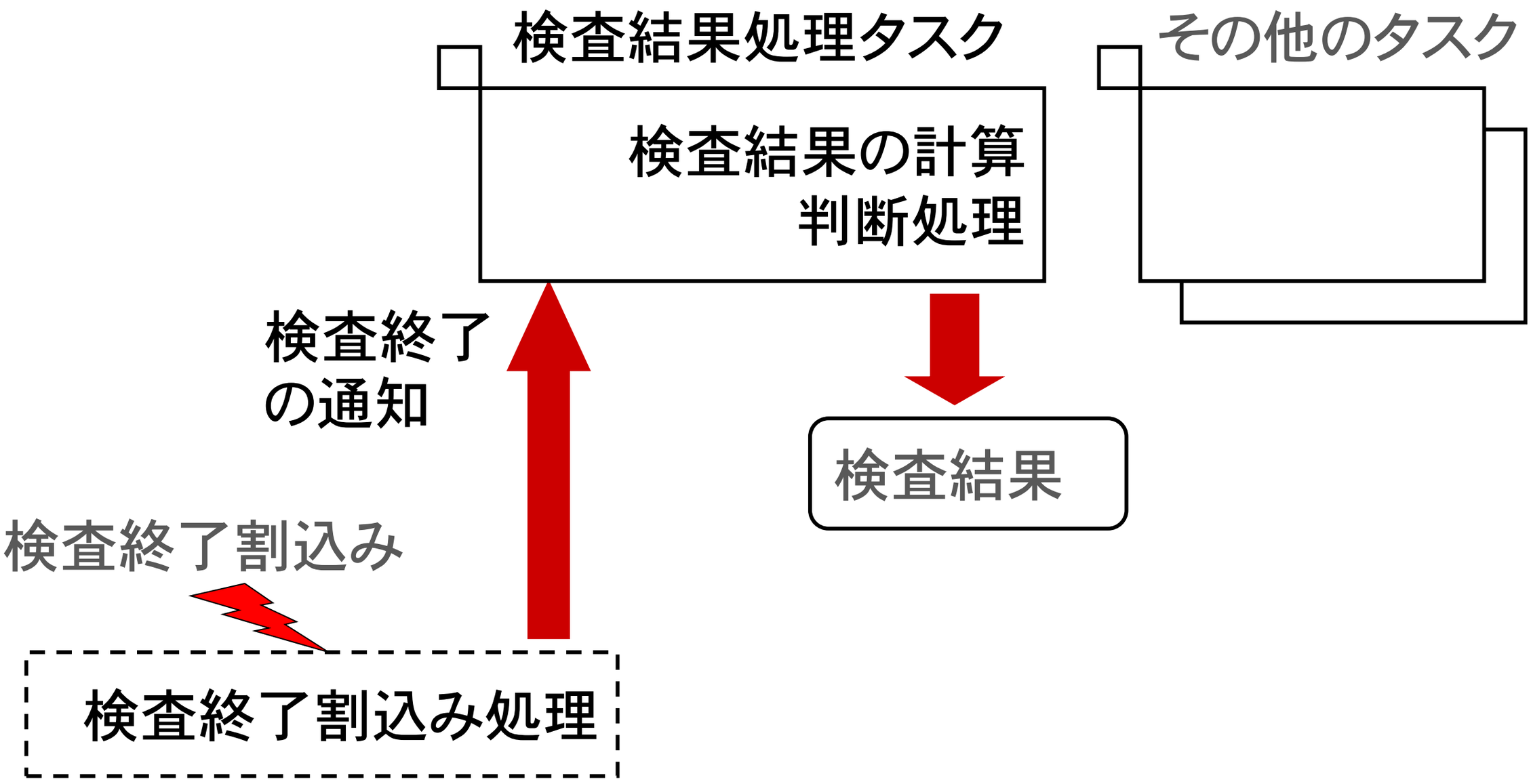
④改善のポイント

- 割込み処理では、複雑な計算や操作を行わない。
- 時間がかかる計算を行うタスクは、優先度を下げる。

⑤Task分割(役割分割)の良い例

- 割込み処理は、検査終了をメインタスクに伝えるだけにする。
- 判断および計算処理を検査結果処理タスクで行い、優先度を下げる。

⑤Task分割(役割分割)の良い例



⑥まとめ

- **プログラムを試作しながらプロジェクトを進める場合（プロトタイピング手法）では、試作後のプログラムの扱いを最初から考慮しておく。**
  - 骨格として使用するのか
  - 部分的に採用するのか
  - 使い捨てるのか
- **試作プログラムは、最終的な動作環境も考慮する。**
  - 他に動作するタスクなども考慮する。

7．コーディング／実装

実装に関連する技術項目

- **様々な資源の扱い**
  - プロセス管理、同期・通信、時間管理、メモリ管理、時間管理、割込処理
  - 大型データストレージ（ファイル、データベース、…）
  - ネットワーク通信（インターネット、近距離通信、…）
  - デバイス制御（センサー、アクチュエーター、…）
- **ユーザインタフェース**
  - GUI、音声認識、画像認識、…
- **マルチメディア（信号処理）**
  - 動画、音声、…
- **暗号・認証、セキュリティー**
- **省電力**
- **信頼性**

実装に関連するツール

- プログラミング言語
- 開発環境
- 組込みリアルタイムOS
- ミドルウエア
- ネットワーク外部サービス
- デバイス（インタフェース）

7-1. リアルタイムOSを用いた開発

リアルタイムシステムは複雑

- **実世界とのかかわり**
  - ランダムに起きる事象への対応
  - 実時間を扱う必要性
- **複雑な処理が要求される。**
- **そのための技術**
  - 並行処理（タスク、スケジューリング）
  - 同期・通信
  - 実時間処理
  - 記憶管理

なぜRTOSが必要か？

- **これをすべてユーザプログラムで実現できるか？**
  - 実現できても、共通機能として常駐システム化したほうが楽なものも多い。
  - 実現できないものもある。

- **何らかの汎用的なシステムでサポート**

➔ リアルタイムＯＳ（Real-time OS: RTOS）

RTOSとは何か？

- リアルタイム・組込みシステム開発において、
  共通に使用される管理プログラム
- リアルタイムシステム向きの機能を持つ
  （各イベントに対して高速に応答できる）
- タスク切換時間、各サービスコール時間があらかじめ予測できる
- コンピュータの持つ資源を仮想化し、効率利用できる（再利用性）

RTOSが提供する機能

1. タスク管理機能
2. タスク同期管理機能
3. 同期通信機能
4. メモリ管理機能
5. 時間管理機能
6. 割込み管理機能

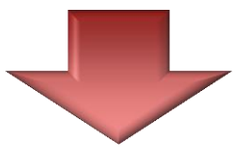
RTOS導入のメリット

- **プログラム作成が容易に←リアルタイム処理の基本処理を扱ってくれる**
  - 並行処理・スケジューリングのサポート
    ➔ タスク分割設計が容易に、効率的なI/Oも簡単に実現
  - 同期処理のサポート
    ➔ タスク間の通信・同期の実現が容易に
  - 記憶管理のサポート
    ➔ 記憶管理の細部に関らないでもすむ。
- **プログラムの再利用性の向上**
- **保守性・拡張性の向上**

RTOSが提供しない機能

- 入出力管理機能
- 通信・ネットワーク機能
- ファイル管理機能
- ユーザインターフェース機能（GUI)
- 音声認識、音声合成
- 画像圧縮伸張

など

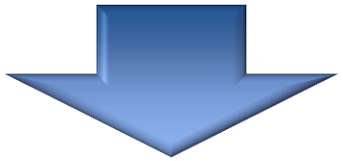


- 上位のミドルウェア／デバイスインタフェースでサポート

7-2. 組込みリアルタイムシステムの機能

リアルタイムシステムは複雑

- **実世界とのかかわり**
  - ランダムに起きる事象への対応
  - 実時間を扱う必要性



- **複雑な処理が要求される。**
- **そのための技術**
  - 並行処理（タスク、スケジューリング）
  - 同期・通信
  - 実時間処理
  - 記憶管理

なぜRTOSが必要か？

- **これをすべてユーザプログラムで実現できるか？**
  - 実現できても、共通機能として常駐システム化したほうが楽なものも多い。
  - 実現できないものもある。

- **何らかの汎用的なシステムでサポート**

➔ リアルタイムOS（Real-time OS: RTOS）

RTOSとは何か？

- リアルタイム・組込みシステム開発において、
  共通に使用される管理プログラム
- リアルタイムシステム向きの機能を持つ
  （各イベントに対して高速に応答できる）
- タスク切換時間、各サービスコール時間があらかじめ予測できる
- コンピュータの持つ資源を仮想化し、効率利用できる（再利用性）

RTOS導入のメリット

- **プログラム作成が容易に←リアルタイム処理の基本処理を扱ってくれる**
  - 並行処理・スケジューリングのサポート
    ➔ タスク分割設計が容易に、効率的なI/Oも簡単に実現
  - 同期処理のサポート
    ➔ タスク間の通信・同期の実現が容易に
  - 記憶管理のサポート
    ➔ 記憶管理の細部に関らないでもすむ。
- **プログラムの再利用性の向上**
- **保守性・拡張性の向上**

リアルタイムシステムの必要な機能とRTOS

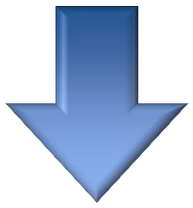
RTOSが提供する機能

- タスク管理機能
- タスク同期管理機能
- 同期通信機能
- メモリ管理機能
- 時間管理機能
- 割込み管理機能

RTOSが提供しない機能

- 入出力管理機能
- 通信・ネットワーク機能
- ファイル管理機能
- ユーザインターフェース機能（GUI）
- 音声認識、音声合成
- 画像圧縮伸張

など



- 上位のミドルウェアでサポート

組込みリアルタイムOSの機能

- 複数のタスクやハンドラの間の協調動作を管理する

- 具体的には…
- タスク（スレッド、プロセス）管理
  - スケジューリング、ディスパッチング
- タスク間同期
- タスク間通信
- 資源（記憶）管理
- 時刻／時間管理

7-3. リアルタイム?

Real-time ?

リアルタイムシステムとは？

- 一般的には、次々に起こる実世界の事象（イベント）に合わせて素早く処理することが求められるようなシステム

- 入出力処理
- 機器類の制御
- 実時間の進行に追従した処理が重視

- きちんとした定義は難しい（後のスライド…）

リアルタイムシステムを考える…

- **根本的には、「素早い」応答、実世界の実時間に追従**

- **では、計算処理性能が高いコンピュータであれば、リアルタイムシステムか？**
  - 半分あたり、半分はずれ

- **実世界の実時間に応答・追従のための「時間制約」を満たすために、コンピュータが十分「速い」ことは必要条件**
  - 計算処理性能が高いコンピュータであるにも係らず、時間制約を満たせてないケースもある（つまり、十分条件ではない）
  - リアルタイム向きにできていないことが原因（次のスライド）
  - どこまで短い時間制約にまで対応できるかも、重要な指標（後述）

リアルタイムシステムの定義

- **リアルタイムシステムは、利用できる計算機資源（resource）に限りがある中で、故障のような厳しい結果をもたらす応答時間違反を行なわないシステム**
  - 「故障」＝システム仕様での要求を満たせないこと
- **リアルタイムシステムとは、その論理的正当性が、アウトプットの正確性とその時刻の両方に依存するシステム**
  - （例）システムへの要求＝「127＋382の答えを求めなさい。答えは3分後までに出しなさい（現在：12時23分）」
    - （答） 509（12時30分）
      ➔ リアルタイムシステムでは計算失敗の例となる
    - （答） 509（12時24分）
      ➔ リアルタイムシステムでも計算成功の例

ソフトリアルタイムとハードリアルタイム

- **ソフト・リアルタイム・システム**
  - 「ソフトリアルタイムシステム」とは、時間制約を満たす事に失敗した時に、性能低下はあるがシステム故障は起こさない
- **ハード・リアルタイム・システム**
  - 「ハードリアルタイムシステム」とは、時間制約を満たす事に失敗することが一回でもあると、完全に破壊的なシステム故障につながるかもしれないシステム
- **ファーム・リアルタイム・システム**
  - ファームリアルタイムシステムとは、いくつかの時間制約ミスではトータルな故障にはならないが、多くの時間制約ミスがあると完全に破壊的なシステム故障につながるかもしれないシステム

！ソフトとハードは、求められる時間制約の長短によって区別されるものではない。

時間制約を守るための方法

- **方針１**
  - デッドライン（deadline）が近い処理を先に実行する
    - 直感的にも自然な方法
    - 一定の条件のもとでは最適な方法であることが理論的にも証明されている（RMS: Rate Monotonic Scheduling）

- **方針２**

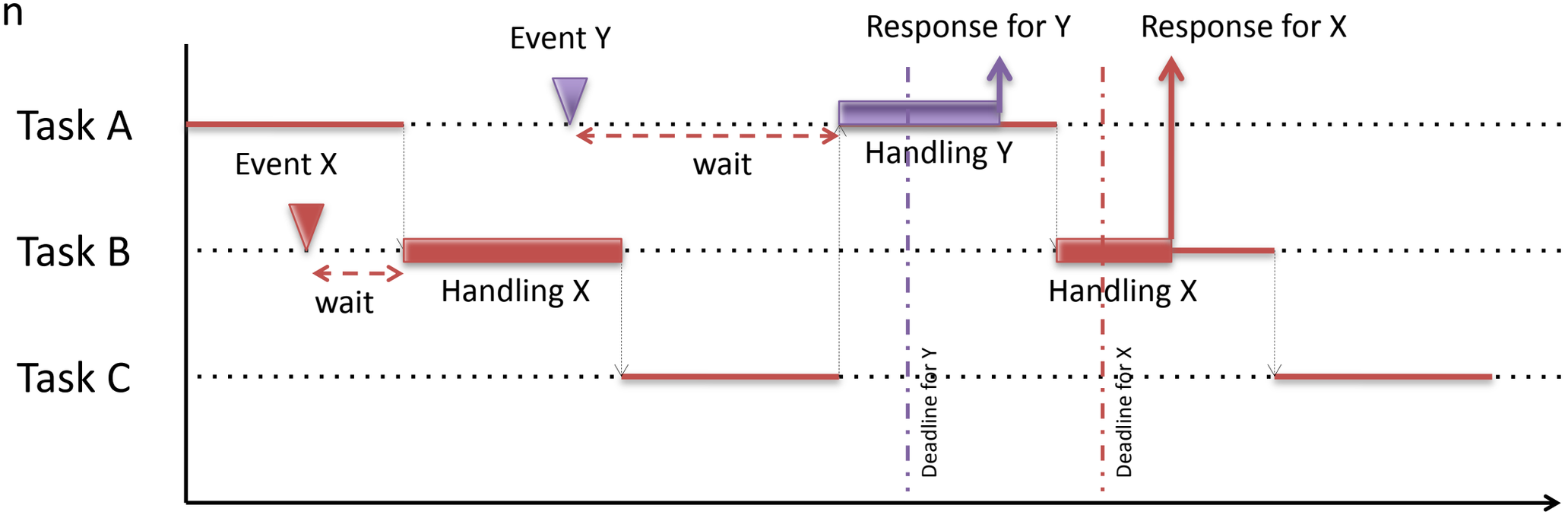
  ①各処理それぞれの実行時間の予測

  ②処理時間が予測できれば、すべての時間制約を守るように処理の順番を調整（スケジューリング）

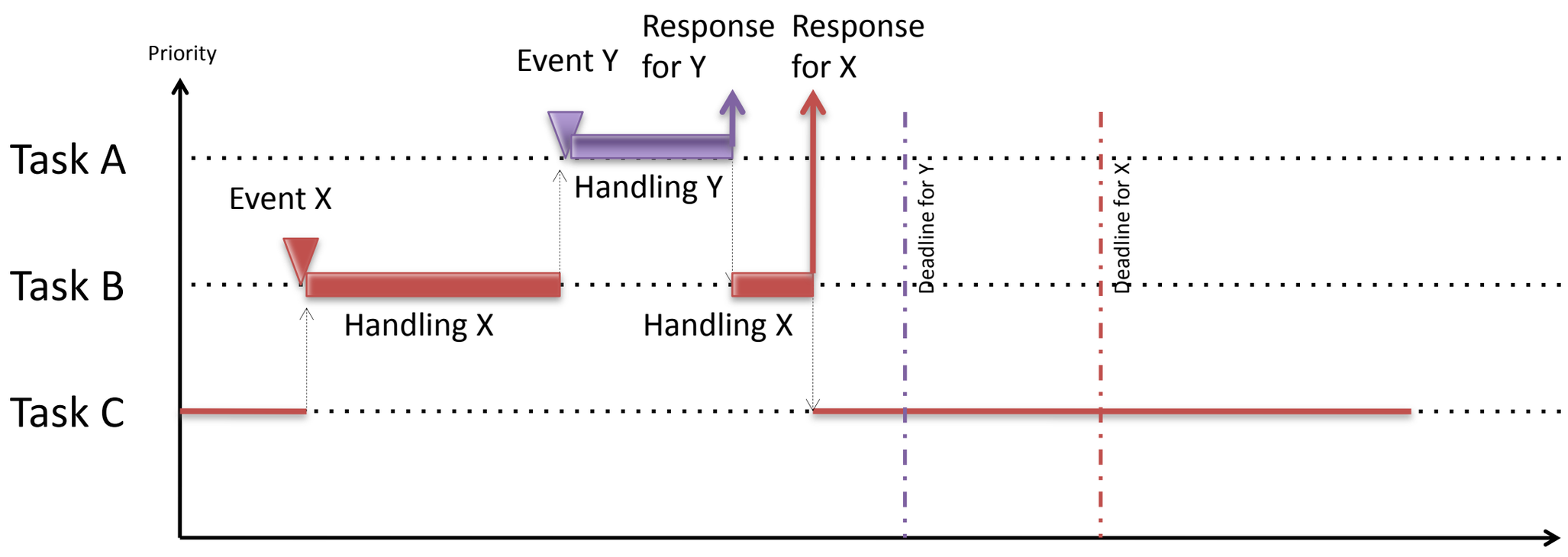
  ③処理時間が予測できない部分は、最悪処理時間を保証できる仕組みを入れる（タイムアウト）

優先度ベーススケジューリングと ラウンドロビンスケジューリングの違い

Round Robin Scheduling



Prioritized Preemptive Scheduling



7-4. コーディング ガイドライン

コーディングルール

- ソースコードをコーディングするとき、どのような書き方をするべきか、のノウハウを整理し、開発メンバー内で参照・遵守。それによってソースコードレベルの品質を均質化することを目的。
- 代表的なコーディングルール
  - MISRA-C
  - ESCR

MISRA-C

- **MISRA　=　Motor Industry Software Reliability Association**
- **1994　“Development Guideline for Vehicle Based Software” を発表**
  - ソフトウェア開発プロセス全体に言及
  - 2000：ISO/TR 15479として発表
- **1998　“Guidelines for the Use of the C Language in Vehicle Based Software”を発表**
  - C言語でプログラミングする工程にスポットを当てる
  - MISRA-Cと呼ばれている。
- **内容**
  - 127のルールから成り立っている
  - 別紙参照
  - 【参照】MISRA-C研究会:「組込み開発者におくるMISRAーC：組込みプログラミングの高信頼化ガイド」,日本規格協会, 2004年

MISRA-Cのルールの例

- **ルール10（推奨）**
  - コード部はコメントアウトしてはならない
- **解説**
  - コードをコメントアウトすることで可読性を損ない、プログラマの勘違いを誘発する可能性がある。またコメントの入れ子が発生する可能性もあるため、コードのコメントアウトはすべきではない。

元コード

```
extern SI_16 func1(void)
{
   SI_16 si16_var = 0;
   si16_var = func1();
   if(si16_var != 0)
   {
      /* エラー処理 */
   }
}
```

コメントアウト例（ルール違反）

```
extern SI_16 func1(void)
{
   SI_16 si16_var = 0;
   /* コメントアウト
   si16_var = func1();
   if(si16_var != 0)
   {
      /* エラー処理 */
   }
   */
}
```

回避策

```
extern SI_16 func1(void)
{
  SI_16 si16_var = 0;
#if 0
  si16_var = func1();
  if(si16_var != 0)
  {
    /* エラー処理 */
  }
#endif
}
```

MISRA-Cのルールの例

- **ルール１４（必要）**
  - char型は常にunsigned charかsigned charで定義されなければならない。
- **解説**
  - char型の扱いに関してより安全なソースコードを記述するために、処理系により扱いの異なる可能性のあるchar型を使用させないためのルールである。
  - C906.2.1.1項に、「“単なる”Char型を符号付きとして扱うか否かは、処理系定義とする。」とある。
  - 従って、char型は使用せずに、unsigned char型かsigned char型を使用する。

MISRA-Cのルールの例

- **ルール17（必要）**
  - typedef名は再使用してはならない.
- **解説**
  - ルール17は，typedef名の使用を制限するためのルールである。
  - typedefで一度定義した名前と同じ名前を再使用した場合、可読性を損なうだけでなく、再使用のされ方によってはtypedef名が隠蔽され使用できなくなる可能性もある。

MISRA-Cのルールの例

- **ルール21（必要）**
  - 識別子を隠してしまうことになるため、内部有効範囲の識別子は、外部有効範囲の識別子と同じ名前で使用してはならない。

例1：

```
SI_16 si16_var;
{
  SI_16 si16_var;
  si16_var = 3;
}
```

例2：

```
#define func() {SI_16 si16_var; si16_var
 = 0;}
{
  SI_16 si16_var;
  func();
  si16_var = 3;
}
```

MISRA-Cのルールの例

- **ルール30（必要）**
  - すべての自動変数は使用する前に値を代入しなければならない。
- **解説**
  - 値を設定していない自動変数の使用を禁止している。
  - C906.5.7項に、「自動記憶域期間をもつオブジェクトを明示的に初期化しない場合、その値は不定とする。」とある。
  - 自動変数は宣言時の初期化を含めて使用する前に値を代入し、自動変数が不定の値にならないようにすべきである.

ESCR

- **IPAが作成した組込みソフトウェアのソースコード品質向上を目的として定めたコーディングルール集**
  - ソフトウェアの信頼性、保守性、移植性、効率性の4つの視点
- **コーディング作法**
  - 言語非依存なメタルール
- **サンプルルール**
  - C言語のソースコードに適用する際のルール

- 【参考】IPA:「組込みソフトウェア開発向けコーディング作法ガイド/C言語版」, 翔泳社, 2007.

8．レビュー／テスト

8-1. コードレビュー手法

レビューの意義

- **背景**
  - ソフトウェア開発の周期が短期間化
  - 低価格化、短納期開発、高信頼性要求
- **問題**
  - 欠陥除去作業は、下流工程になってからのほうが修正コストが大きい。
- **解決アプローチ**
  - できるだけ早い段階の上流工程で、欠陥除去をする。
  - 一つ一つのシステムや機能を手離れよくするために、最初から品質の高いものを作ることが不可欠。
  - そのために、できるだけ上流工程のプロセスを改善、レビューの力を入れることが重要。

レビューの分類

- **プロジェクト管理を主眼としたレビュー**
  - マネージメントレビュー
  - ステータスレビュー
  - プロジェクト終了レビュー
- **作業成果物の品質向上を主眼としたレビュー**
  - 契約前に実施するレビュー
    - 契約レビュー
    - 説明書原稿レビュー
  - 開発時に実施するレビュー
    - デザインレビュー
    - コードインスペクション

作業成果物の欠陥・改善のためのレビューは、「ピアレビュー」とも呼ばれる

ピアレビューの種類と概要

レビューの種類	概要
アドホックレビュー	必要に応じて対応する即席形のレビュー。記録を作成するなど、管理的要素は含まない。
パスアラウンド	電子メール等を利用して、作成成果物のコピーを配布し、複数のコメントを依頼する。
ペアレビュー	作成者とレビューア（１名）がペアを組んで、作成成果物を確認。
ウォークスルー	作成者が作業成果物をチーム構成員に説明し、コメントを求める。
チームレビュー	複数のメンバーが参加し確認する。計画的に実施し、定義された手順に従って実施する。
インスペクション	最も体系的に参加者に特定の役割を与えた厳格なレビュー。作成者以外の参加者が会議を主導し、チェックリストの使用、分析の実施なども実施する。

レビューとテストの違い

	レビュー	テスト
確認範囲	文書、モデルやコードの静的な評価のみ	想定環境下での動作を実際に評価する
障害発見タイミング	間接的な症状だけではなく、問題点を直接指摘する。そのため、問題点に対する改善を即検討し、実施することが可能	障害、つかりシステムが期待通りの挙動を示さないという実行結果の一つにでくわして初めて原因となる欠陥を究明。 原因究明後にその対応を検討し、実施する。
潜在障害対応効率	テストでは検出が遅れる可能性のある欠陥も一度に確認し、問題を顕在化することが可能。 問題の顕在化時期が早い場合には、修正が比較的楽であるため、対応が容易	障害を再現させるためのデータや、環境を用意する必要がある。 またテストで大きな変更必要事項を発見しても、時間的な問題や作業担当者の心理的な問題で、通常は抜本的な対策はとりがたく、暫定的対応になりがち。 その結果対応が遅れる。

8-2. テスト手法

テスト手法

- **ホワイトボックステスト**
  - プログラムの構造に着目したソフトウェアテスト。
  - 着目する構造には、命令や分岐がある。
- **ブラックボックステスト**
  - プログラムの入力と出力に注目したソフトウェアテスト。
  - 仕様とおりにプログラムが動作するかどうかテストする。
  - プログラムの入力が単一のときは、同値分割や限界値分析をする。
  - プログラムの入力が複数あり、相互に影響を与えるときは、デシジョン表や原因結果グラフなどを用いてテスト入力値を決定する。

ホワイトボックステスト（制御フローテスト）

- **命令網羅基準（C0）によるテスト**
  - モジュールに含まれるすべての命令を実行するテスト。
- **分岐網羅基準（C1）によるテスト**
  - すべての分岐において、すべての分岐の方向を実施するテスト。
- **条件網羅基準（C2）によるテスト**
  - 複数条件で起こりうる真偽と分岐の組み合わせ経路を実行するテスト。

ブラックボックステスト

- **同値分割テスト**
  - それぞれの値から代表的な値を入力として選んで行うテスト
- **限界値分析テスト**
  - 入力の範囲を想定される出力毎に分割し、それぞれの範囲の境界を入力として選んでテストを行う。
- **デシションテーブル**
  - 入力が複数のパラメーターから構成されるときに、入出力の関係を表で表したもの。
  - 表の各列がテストケースをあらわしている。

様々なテスト手法

- **単体テスト**
  - 関数やメソッドなどの小さい単位で行うテスト。
  - ホワイトボックステストで行うことが多い。
- **結合テスト**
  - 単体テストが完了したモジュールを組合わせるテスト
  - トップダウンテスト
    - 結合テストのうちで、単体テストが完了モジュールのうち、上位モジュールから順に結合させてテストを行う。
    - 仕様的振る舞いを決定する上位モジュールからテストできることで、機能漏れ、仕様の認識違いを開発の早い段階でチェックできる。
    - 開発とテストを並行して進めにくい。
  - ボトムアップテスト
    - 結合テストのうちで、単体テストが完了モジュールのうち、トップダウンテストとは逆に、下位モジュールから順に結合させてテストを行う。
    - 独立性の高い下位モジュールから順にテストを行うことで、開発とテストを実施できる。
    - 後からテストする上位モジュールに不具合があった場合、下位モジュールに影響が及ぶ。

テスト手法

- **システムテスト**
  - ソフトウェア単独ではなく、他のプログラムやハードウェア、ネットワーク、データベースなどと組み合わせて実施するテスト。
  - 組込みでは、実際のハードウェアにソフトウェアを組み込んで行うテストに相当。
- **αテスト・βテスト**
  - 完成前のソフトウェアを一般ユーザに使ってもらって欠陥を発見するテスト。
- **ストレステスト**
  - ソフトウェアシステムに高い負荷をあたえ、データの破壊や致命的問題が発生しないかテストすること。
  - 高負荷状態でしか起きない、発生頻度の低い不具合の発見ができる。
- **パフォーマンステスト**
  - ソフトウェアの性能を測り、規定の性能が出るか確かめること。
- **回帰テスト（レグレッションテスト）**
  - プログラムを修正・変更したときに、修正前の他の機能が動作することを確認するために行うテスト。

９．信頼性

プラント事故

社会の安心・安全を脅かす事故例及び製品安全問題

- 日航機ジャンボ機御巣鷹山墜落事故
- 姫路花火歩道橋事故
- クレーム隠し、データ改ざん（自動車）
- 株式誤発注問題
- ＪＲ福知山線列車脱線事故
- 耐震強度偽装問題
- 食品（菓子消費期限、再利用、肉…）
- 大型自動回転ドア挟まれ事故
- 石油温風暖房機一酸化中毒事故
- エレベータ挟まれ事故
- 製品安全マーク
- ガス瞬間湯沸かし器一酸化炭素中毒事故
- シュレッダー幼児指切断事故
- リチウム電池発火事故
- ジェットコースタ脱線事故
- 扇風機火災事故
- ガス機器ＣＯ中毒で２１年間で３５５人死亡

企業を取り巻く社会的潮流：CSR

- Corporate Social Responsibility 企業の社会的責任
  - 法令遵守
  - 説明責任
  - 情報開示
  - 誠実
  - 人材尊重
  - 環境保全
  - グローバル市場に参加するための設計、生産、販売での的確な行動

「組込みシステムも、
信頼性・安全性は
極めて重要なテーマ」

ISO 9126：ソフトウェアの品質特性

- **機能性：全体としてのユーザニーズの充足度**
  - 合目的性･正確性･接続性･整合性･セキュリティー
- **信頼性：所定の条件下で正しく稼動している割合**
  - 成熟性･障害許容性･回復性
- **使用性：理解、習得、運用の容易さの度合い**
  - 理解性･習得性･操作性
- **効率性：所定の機能を実現する処理性能と必要資源の効率性**
  - 実行効率性･資源効率性
- **保守性：修正と改善の容易さの度合い**
  - 解析性･変更作業性･安定性･試験性
- **移植性：他の環境へ移す場合の容易さの度合い**
  - 環境適応性･移植作業性･規格準拠性･置換性

参考文献

参考文献

- 社団法人トロン協会 著, 坂村 健 監修：「組込みシステム実践プログラミングガイド―ITRON仕様OS/T-Kernel対応」, 技術評論社, 2008年.
- 組込みソフトウェア管理者・技術者育成研究会（SESSAME）：「Open SESSAME Seminar/組込みソフトウェア技術者・管理者向けセミナー：初心者向けテキスト」, http://www.sessame.jp/.
- 富士通ラーニングメディア ：「標準テキスト 組込みプログラミング《ソフトウェア基礎》 」、技術評論社 .
- 杉浦 英樹, 橋本 隆成：「組込みソフトウェア開発 基礎講座 (組込みエンジニア教科書)」 , 翔泳社.
- 情報処理学会組込みシステム研究会監修：「組込みシステム概論 (組込みシステム基礎技術全集 vol. 1)」, CQ出版.
- MISRA-C研究会: 「組込み開発者におくるＭＩＳＲＡ－Ｃ：組込みプログラミングの高信頼化ガイド」,日本規格協会, 2004年.
- IPA：「組込みソフトウェア開発向けコーディング作法ガイド/C言語版」, 翔泳社, 2007.
- F. P. Brooks: “No Silver Bullet: Essence and Accidents of Software Engineering”, Addison-Wesley, 1995.

【実習】ITRON中級テキスト「 RTOS（μITRON）を使ったリアルタイムシステム開発手法入門」

著者　TRON Forum