第16回パターン勉強会

非機能要求とアーキテクチャ

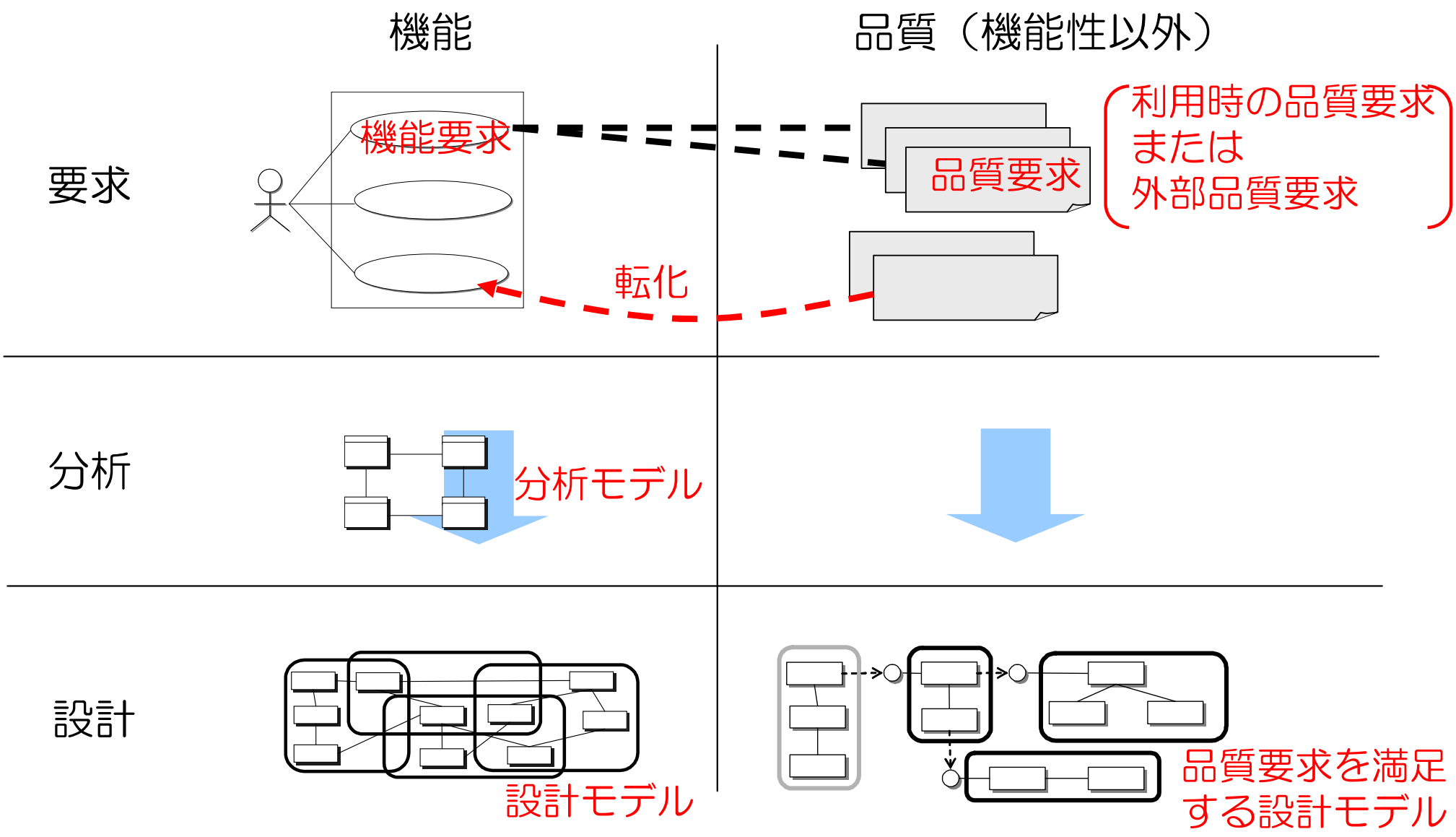
2008年8月6日

鷲崎 弘宜
早稲田大学 / 国立情報学研究所
washizaki@waseda.jp

機能要求と非機能/品質要求

- 機能要求（Functional requirements）
  - 入力に対する出力の規定
  - 必要な出力を得るための入力への処理
- 非機能要求（Non-functional requirements）
  - 機能要求以外のあらゆる開発に課せられる要求
  - 品質要求, コスト, 納期, 開発上の制約など
- 品質要求（Quality requirements）
  - 品質上、満たさなければならない事柄
  - 「測定」可能な形で定義されることが望ましい
- 様々な品質要求
  - 機能要求に付随するもの
  - システム全体に対するもの
  - 機能要求に転化するものもある（例: セキュリティ → 認証機能）

機能と品質の関係


機能
品質（機能性以外）
要求
機能要求
品質要求
利用時の品質要求
または
外部品質要求
転化
分析
分析モデル
設計
設計モデル
品質要求を満足
する設計モデル

品質要求の記述: CMU/SEI手法群

- 品質要求の記述手法の一種として、品質シナリオがある
  - CMU/SEIアーキテクチャ手法群における共通の記述方法
- CMU/SEIのアーキテクチャ手法群
  - Attribute Driven Design（ADD）：品質シナリオに基づくアーキテクチャ設計手法 [ADD]
  - Scenario-Based Architectural Analysis（SAAM）：品質シナリオに基づくアーキテクチャ評価手法 [SAAM]
  - Architecture Trade-off Analysis Method (ATAM): アーキテクチャ設計上の（特に品質要求に関する）判断/選択/決定の結果を評価する手法 [ATAM]
  - Attribute-Based Architectural Style（ABAS）：品質特性について評価済みのアーキテクチャパターンのカタログ [ABAS]

CMU/SEIアーキテクチャ手法群における品質モデル

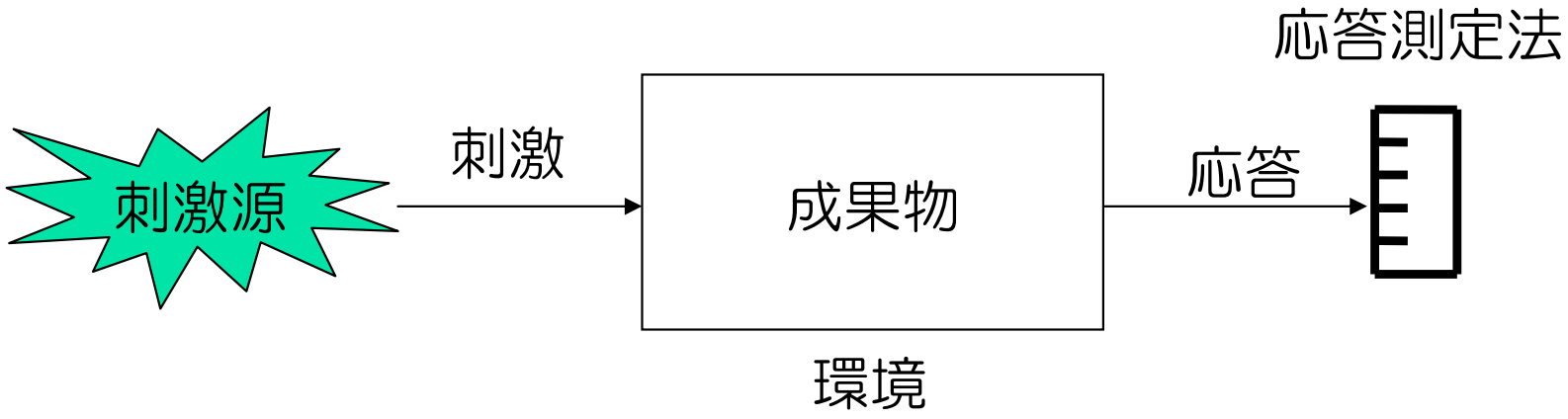
- 可用性
  - 障害を遮断あるいは修復し、故障無く動作する能力
- 変更容易性
  - 要求や環境の変化時に必要な変更コスト・時間が小さい能力
- 性能
  - 時間的制約を満たす能力
- セキュリティ性
  - 外部からの攻撃に抵抗しデータ・情報を保護する能力
- テスト容易性
  - 拡張・修正したソフトウェアのテストによる妥当性確認を容易にできる能力
- 使いやすさ
  - 利用者が必要な作業を容易に達成できる能力

シナリオ [Chaudron][Bass05]

- シナリオ
  - 要求を具体的なストーリーとして記述したもの
  - 利害関係者の意図/興味の表現
  - 機能要求や品質要求の表現
- シナリオの種類
  - 利用シナリオ（ユースケースシナリオ）：予期される利用方法
  - 発展シナリオ: 予期される変更
  - 試験シナリオ: システムへの予期せぬ”力”
- 品質特性と結び付けられたシナリオを「品質シナリオ（品質特性シナリオ）」と呼ぶ

シナリオの構成要素 [Chaudron][Bass05]

- 刺激源: 刺激を生み出す人やシステム、装置など
- 刺激（stimulus）：システムに達した場合に考慮する必要がある事柄・状況
- 環境（environment）：刺激が発生する状況
- 成果物（artifact）：刺激を受ける対象
- 応答（response）：刺激の到着後に開始される活動
- 応答測定法（response measure）：応答の測定方法。要求の検証のため。



シナリオの例 [Chaudron]

- 利用シナリオ:
  - 端末の利用者は、*ピーク時* に [報告をWeb経由で要求] し、5秒以内に報告を受理する。
- 成長シナリオ:
  - *シナリオXの待ち時間を1～2.5秒まで減少させるために*、新サーバを 2人週以内で [追加する]。
- 調査シナリオ:
  - *通常の運用時* に 半数のサーバが [停止] しても、システム全体の稼働率に影響が及ばない。

- 良いシナリオの条件: [刺激]、*環境*、（測定可能な）応答が明白なこと
  - 刺激源、成果物、応答測定法があるとなおよい

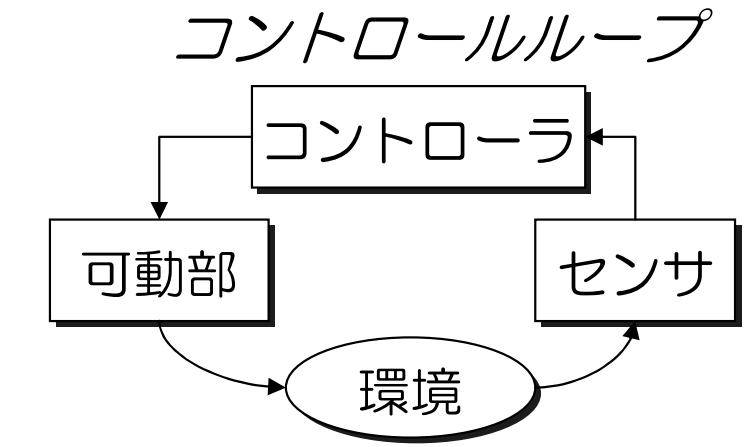
アーキテクチャ設計

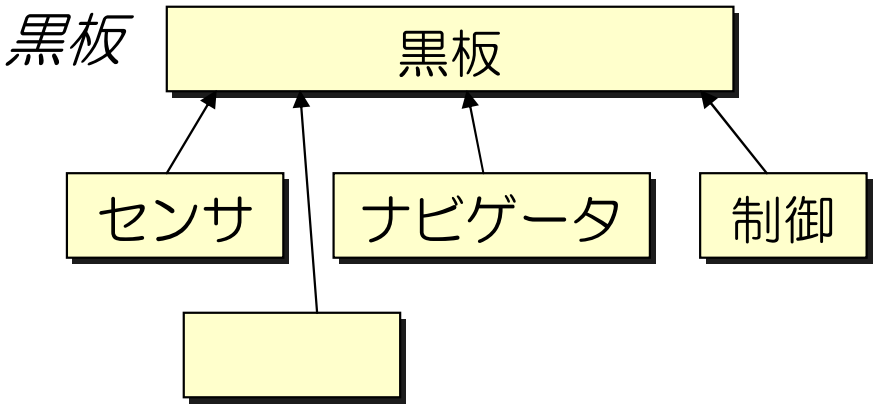
設計とその分類

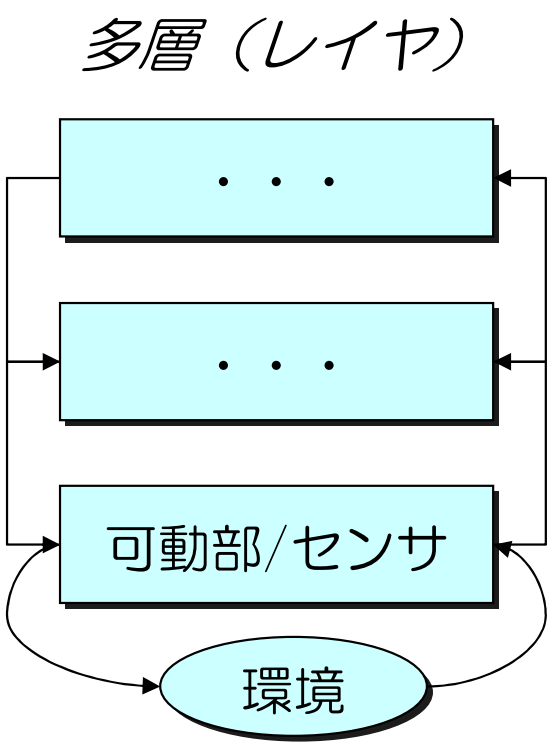
- 要求を実現するための方策を検討し、決定する工程
  - 「何を(What)」から「如何に(How)」へ。
  - 技術的課題とその方策、リスクを明らかに
- アーキテクチャ設計
  - 別名：基本設計、方式設計
  - システムを適度な大きさのコンポーネントに分割することが主たる内容
  - 機能要求＋非機能要求（開発基盤・環境を含む）に基づくソフトウェアアーキテクチャの決定
- 詳細設計
  - 別名：モジュール設計
  - コンポーネントの内部をどのようにすればよいかの決定が主たる内容

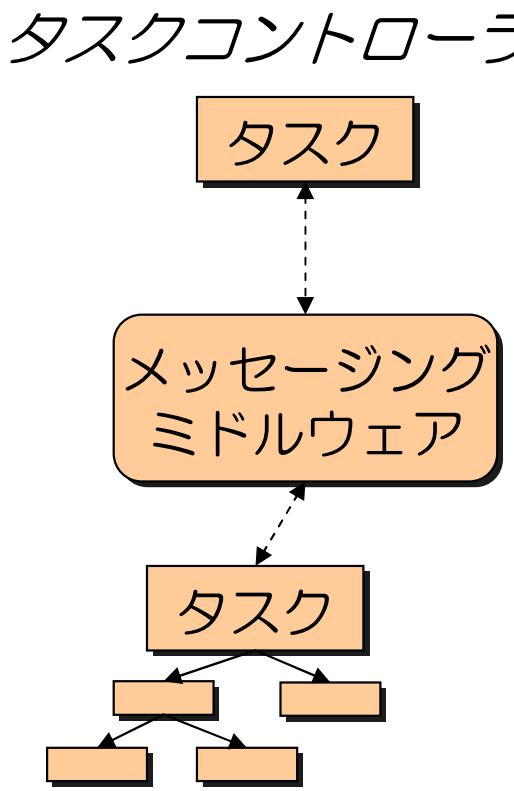
アーキテクチャとは

- 構成要素や関係を大局的に理解可能なモデル
  - システム開発の関係者全員のコミュニケーションを可能に
  - 早い段階での設計上の決定を明らかにする









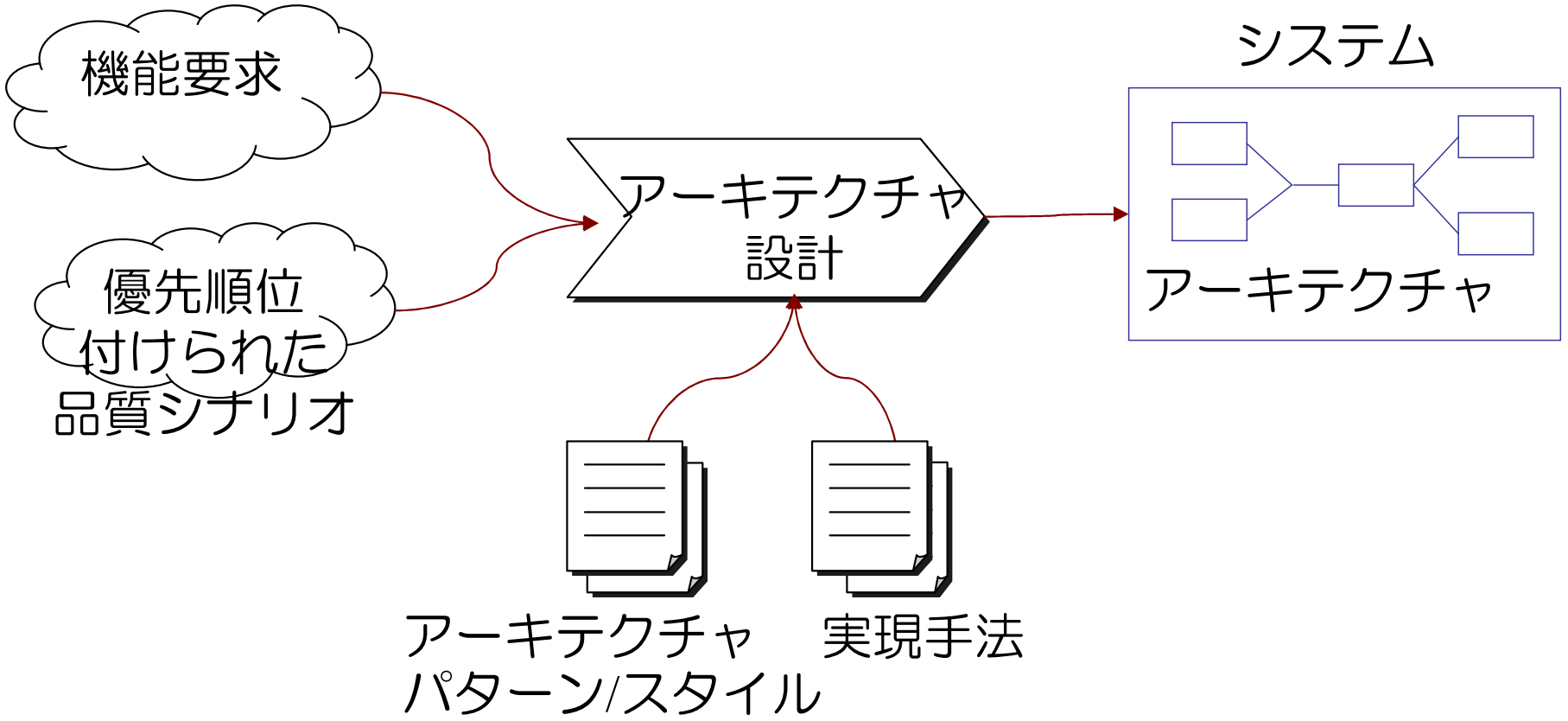
アーキテクチャ設計

- データベース、ネットワーク、分散化、信頼性確保対策等、システムの基盤となる設計
  - 機能要求＋非機能要求を満足するための方式
  - 利用できるものを再利用、新規開発を削減
- 経緯
  - ‘70～’80: Dijkstraの階層構造、Parnasの情報隠蔽
  - ‘80～’90: 大規模・複雑化による初期アーキテクチャの重要性
  - ’90～: 分散化・オープン化によるアーキテクチャ共通化

品質駆動型アーキテクチャ設計 ADD [Matthews03]

品質駆動型アーキテクチャ設計: ADD

- Attribute Driven Design （ADD）
- CMU/SEIのアーキテクチャ手法群の一種
- 機能要求と品質要求の両方をサポートするアーキテクチャを設計する１つの手法



アーキテクチャ設計の開始時の前提条件

- 要求が必要
  - ただし全ての要求がそろっていなくとも構わない
- アーキテクチャは以下より形成される
  - 機能要求
  - 品質要求
  - ビジネス要求
  - アーキテクトの経験
- それらのうちで、特にアーキテクチャ形成について支配的なものを「アーキテクチャドライバ」と呼ぶ

出典: [Matthews03]

例: 車庫システム

- 家庭内情報システムと連動する車庫扉開閉機のプロダクトラインアーキテクチャ（プロダクトシリーズ・ファミリに共通なアーキテクチャ）の設計
- ADDへの入力: 要求群
  - ユースケースとしての機能要求
  - 実現上の制約（プラットフォーム制約など）
  - システムに特化した品質シナリオとしての品質要求
- 例: 車庫システムに対する品質シナリオ群
  - プロダクトライン中のプロダクトごとに機器や制御方法が異なる。
  - プロセッサも異なる可能性がある。
  - 扉を降ろす際に障害物を感知すると1秒以内に停止しなければならない。
  - 家庭内情報システムから監視・制御のためにアクセス可能な必要がある。

出典: [Matthews03]

ADDの手順

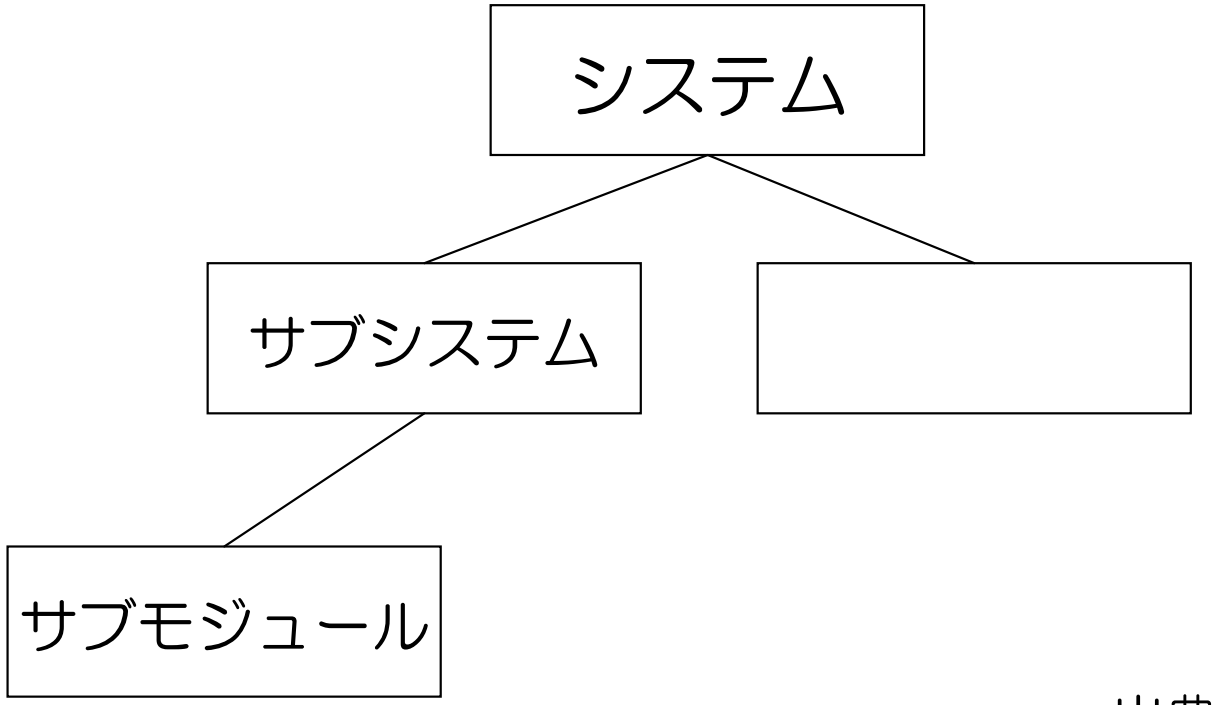
1. 分解するモジュールの選択
   - システム全体から開始
   - 選択モジュールへの入力（制約、機能・品質要求）が得られる必要がある
2. モジュールの洗練
   a. 分解に適したアーキテクチャドライバ群の選択
   b. それらを満足するアーキテクチャ的手法（実現手法・アーキテクチャパターン）の選択
   c. モジュールを具体化、ユースケースで表される機能を割り当て
   d. 子モジュール群のインタフェースの定義
   e. ユースケースと品質シナリオの検証と洗練
3. 全てのモジュールについて必要な限り分解を繰り返す
4. アーキテクチャの複数ビューからの記述

出典: [Matthews03]

1. 分解するモジュールの選択

- システム ➔ サブシステム ➔ サブモジュール

1. システム全体をモジュールとして開始
2. モジュールの選択
3. 分解・洗練の必要な限り繰り返し



出典: [Matthews03]

2a. モジュールの洗練

- 品質シナリオ・機能要求からアーキテクチャドライバを選択
  - 該当モジュールのための優先順位の高い要求群
  - 例: 4つのシナリオがアーキテクチャドライバ
- 品質特性ごとに再構成
  - リアルタイムパフォーマンスの要求
  - 変更容易性の要求
- 注意: 要求は等価値ではない
  - 優先順位の低い要求は、優先順位の高い要求を満足する過程で（制約を伴って）満足させる

出典: [Matthews03]

2b. アーキテクチャパターンの選択と構成

- 確認: アーキテクチャパターンの内容
  - 構成要素, 関連を明示した構造上のレイアウト, 制約, 相互作用の仕組み
  - アーキテクチャパターンは、個々の細かな実現手法（原始的なアーキテクチャ設計手段）群を実現した結果である
- 利用可能なアーキテクチャパターンカタログと実現手法
  - 実現手法（Tactics）[Bass05]
  - 品質特性ベース・アーキテクチャスタイルAttribute-Based Architecture Styles（Layers ABAS、Publish-Subscribe ABASなど）[ABASlein99]
  - アーキテクチャスタイル（Batch Sequentialパターン、Repositoryパターン、Interpreterパターンなど）[Shaw96]
  - POSAアーキテクチャパターン（Layersパターン、Model-View-Controller: MVCパターン、Pipes and Filtersパターンなど）[POSA00]
- 各品質要求について、満足するための実現手法群およびそれらを実現した結果としてのアーキテクチャパターンを特定する
  - ただし、各実現手法・アーキテクチャパターンは複数の品質特性に影響をもたらすため、そのバランスをとる必要がある（例: 保守性 <-> 効率性 のトレードオフ）

出典: [Matthews03]

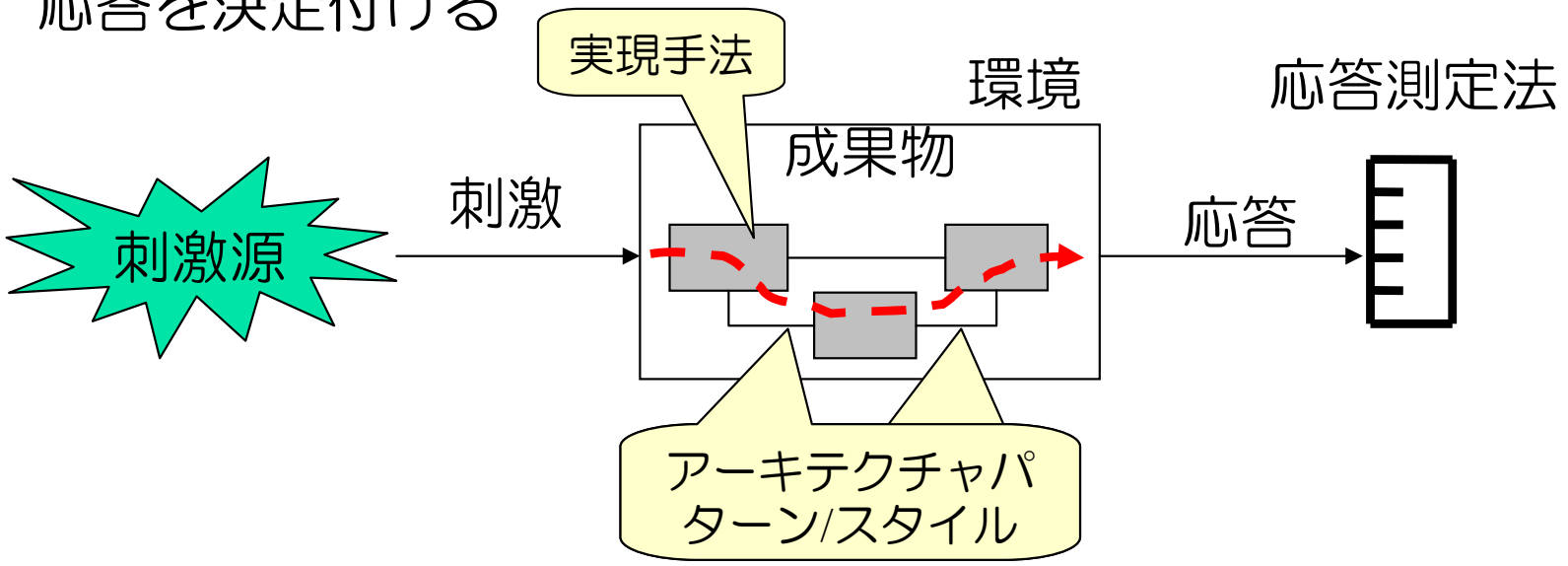
2b: アーキテクチャパターン選択と構成（つづき）

- 該当モジュールについて全体的なアーキテクチャパターンを選択・構成する
  - アーキテクチャドライバを満足すること
  - 同満足に有効な実現手法を組み合わせて得られていること
  - ある実現手法の選択にあたり、他の決定済み実現手法の副作用を考慮すること
- 例:
  - 変更容易性の達成に有効な実現手法「モジュールの一般化」は、アーキテクチャパターン「Interpreter（インタプリタ）」につながる
  - Interpreterの実現例: HTMLレンダリング, マクロ処理
  - 注意: Interpreterは性能（特に時間効率性）に影響するため、部分的に使用すべき可能性あり

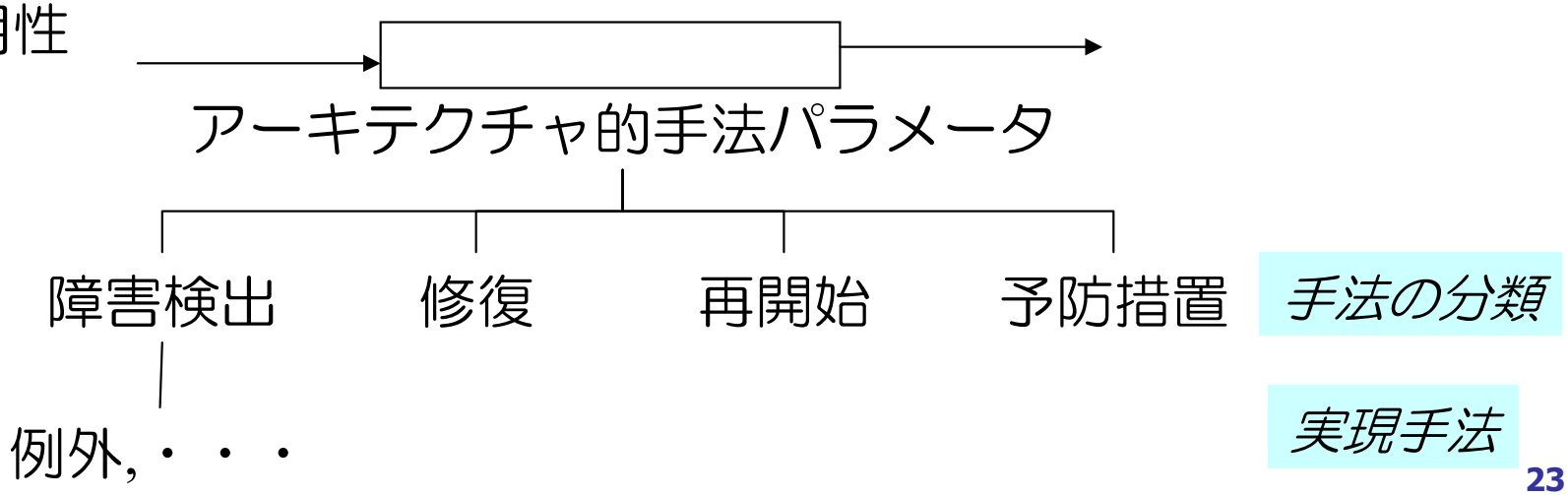
出典: [Matthews03]

実現手法・パターンと刺激の関係

- 実現手法やアーキテクチャパターン/スタイルが、刺激に対する応答を決定付ける



- 実証済みの実現手法やアーキテクチャパターン/スタイルが、アーキテクチャ的手法パラメータとして品質特性ごとに整理されている
- 例: 可用性



2c. モジュールの具体化、機能割り当て

- 2b において実現したモジュール（タイプ）群を具体化する
  - 機能割り当ては、一般的はオブジェクト指向設計における責務割り当てと同様に実施
- 例: 扱う機能
  - 通常の扉昇降 – 性能に非重大（non-critical）
  - 障害物の検知と対処 – 性能に重大（critical）
  - センサ群の仮想化
  - 通信の仮想化

出典: [Matthews03]

2c. モジュールの具体化、機能割り当て（つづき）

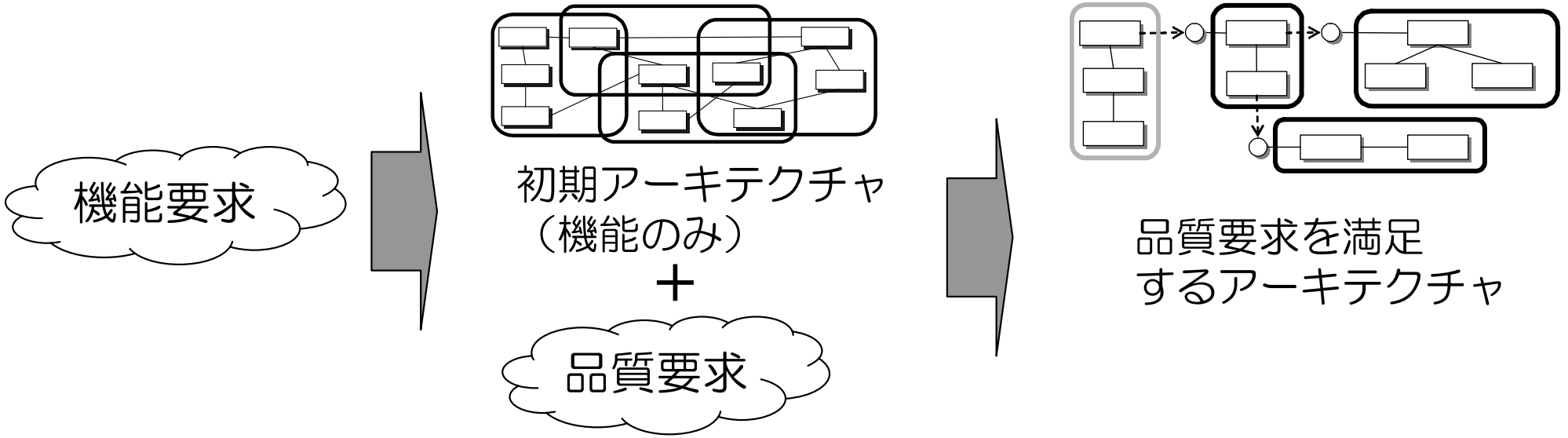
- 求められる機能群を達成していることを検証する
  - 各ユースケースについて、割り当てたモジュールと子モジュール群の責務集合により達成されることを確認
  - 親・子モジュール間の相互作用の決定
  - この段階では、入出力情報そのものは重要ではない
  - 相互作用の決定には実現手法、デザインパターン等を用いることができる
- 求められる品質群を達成していることを検証する
  - 品質シナリオを達成できることを確認
  - 例: シナリオ「車庫扉を降下中に障害物を検出した場合、0.1秒以内に停止しなければならない」は、スケジューラと障害物検知モジュールに委譲される。具体的にはスケジューラは・・・。

出典: [Matthews03]

逆アプローチ: Bosch法 [Bosch00]

2b と 2c の順が逆

1. 機能に基づいたアーキテクチャ設計
2. 品質特性の評価
3. アーキテクチャの変換
   - アーキテクチャスタイル/パターンの適用
   - デザインパターンの適用
   - 品質要求の機能への変換



4. アーキテクチャの複数ビューからの記述

- 論理ビュー ＜必須＞
  - 構造、各機能の制約
  - 形式: クラス図など
  - 機能、制約、他の品質の全考慮結果
- 実行時ビュー ＜必須、シナリオやユースケースあたり＞
  - 実行時の振る舞い、協調関係の記述
  - シーケンス図、コミュニケーション図、アクティビティ図など
  - 性能、可用性など
- プロセス・並行性ビュー
  - 並行作業・動作、同期・非同期の記述
  - ユースケースマップ、シーケンス図、コミュニケーション図、アクティビティ図など
  - 性能、可用性など
- 開発時・実装ビュー
  - パッケージ構成、ファイル間の依存関係
  - パッケージ図、クラス図、コンポーネント図など
  - 変更容易性など
- 配置ビュー
  - 計算機ノードへの配置構成など
  - クラス図、配置図など
  - 変更容易性など
- その他: 一貫性の求められる局所構造など

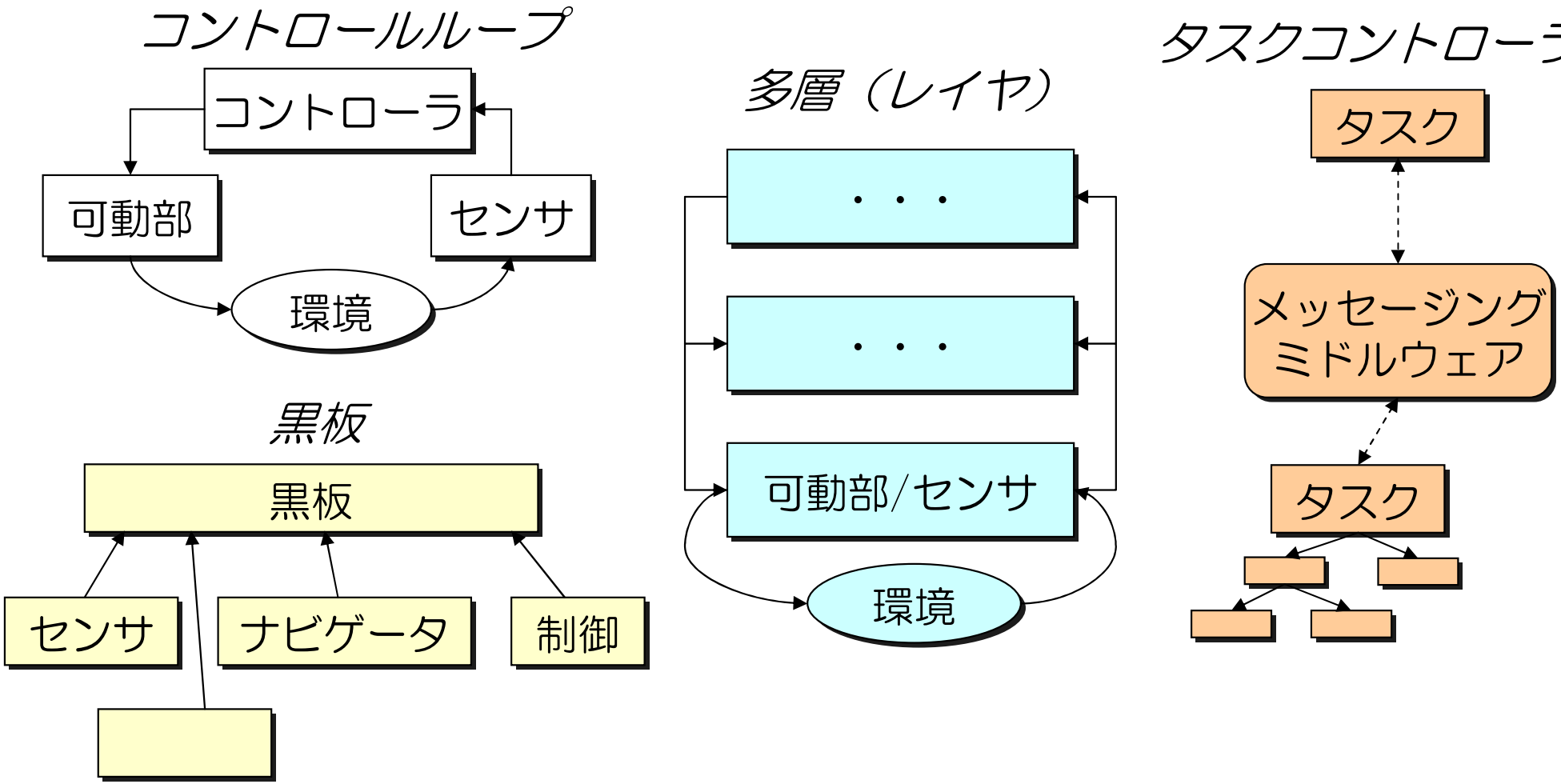
アーキテクチャ設計のまとめ

- 設計（Design）では、Howを明らかにする
  - トレードオフを伴う決定の連続
  - 分析（Analysis）とは本質的に異なる
  - 最初に非機能要求を扱う重要な工程
- ADD: アーキテクチャの品質駆動型設計
  - アーキテクチャパターン・スタイルと品質シナリオを併用した堅実な手順と方針
- アーキテクトの経験によるところが大きい。
  - 最低限、主要なアーキテクチャパターン・スタイルを抑えておく。
  - Layers, Black board, MVCなど

ATAMによるアーキテクチャ定性的評価[Chaudron]

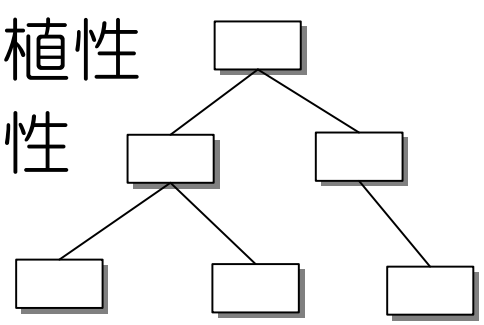
アーキテクチャ評価の理由

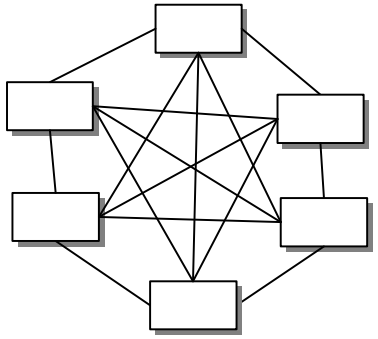
- 実装してからでは遅すぎる
  - 実装を待ってから評価を始めると、コストやリスクが増大
- あらゆる設計はトレードオフを伴う
  - アーキテクチャ設計は、開発中で初めて高いリスクを伴う決定



定量的 <-> 定性的

- 定量的評価: 測定による How much
  - 目的指向、結果が即得られるが結果の解釈は難しい。測れないものあり。
  - モデルから: 機能性、保守性、移植性
  - 実機から: 信頼性、効率性、利用性





- 定性的評価: シナリオによる What if
  - あらゆる注目する品質特性に集中できるが, 精度は評価者の経験に依存する。
  - 信頼性の例: *通常走行中に* 複数センサから矛盾入力値があると1秒以内に停止して再読み取りする

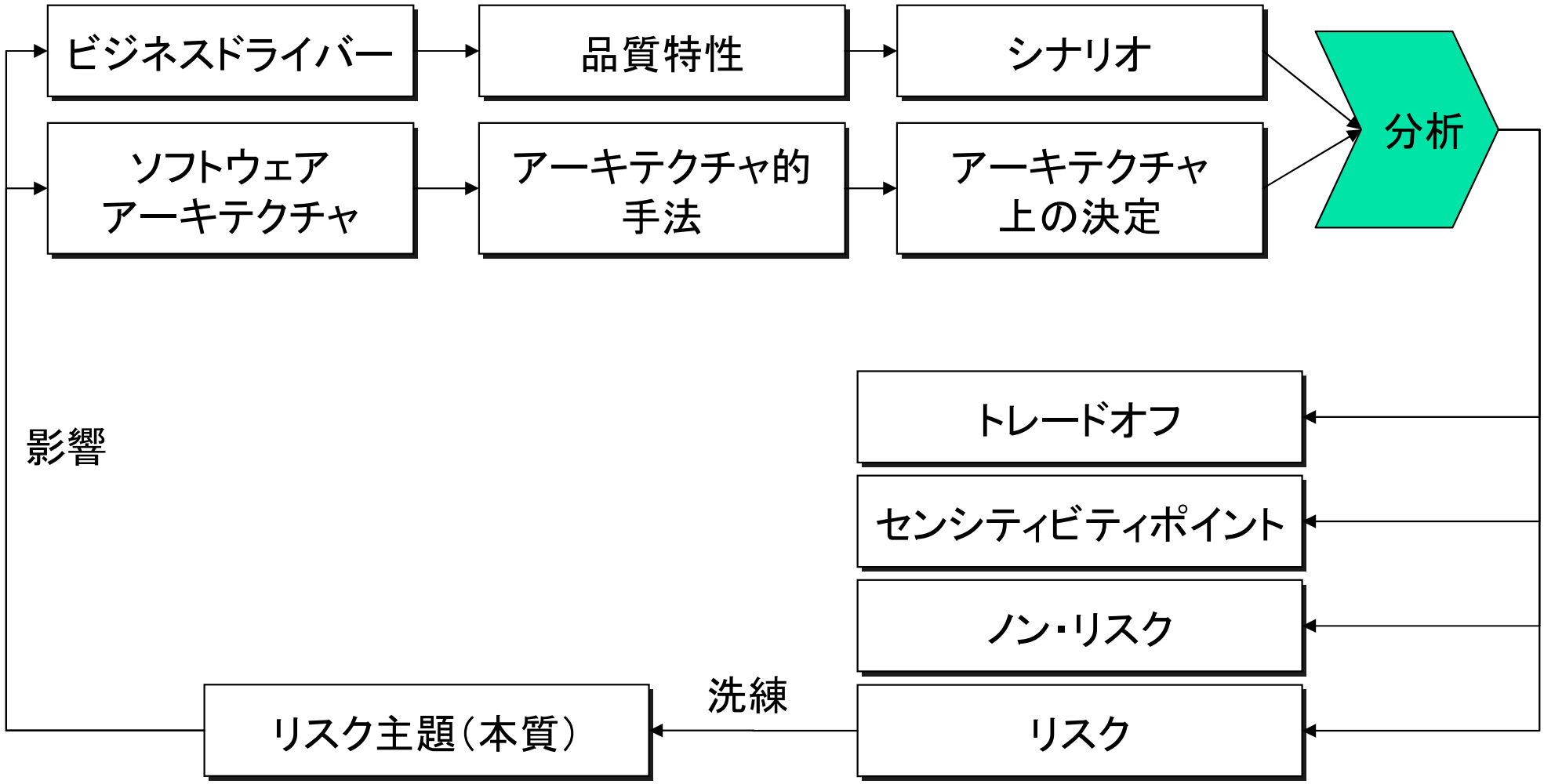
アーキテクチャ	シナリオ１	シナリオ２	シナリオ３
コントロールループ	＋	＋	－
多層（レイヤ）	－		＋

Architecture Trade-off Analysis Method (ATAM)とは

- アーキテクチャ設計上の（特に品質要求に関する）判断/選択/決定の結果を評価する手法
  - •判断/選択/決定が品質要求を十分に扱うかを評価
  - •品質特性を予測する試みではない
- 事例多数
  - •米ボーイング
  - •国内システム [石田06]
- リスクの発見: 品質特性について将来問題を生じる可能性のある選択肢
- センシティビティーポイント（敏感な箇所）の発見: 僅かな変化が品質特性に有意な差をもたらす選択肢
- トレードオフの発見: 複数の品質特性に影響する決定

出典: [Chaudron]

ATAMの全体像［ATAM］



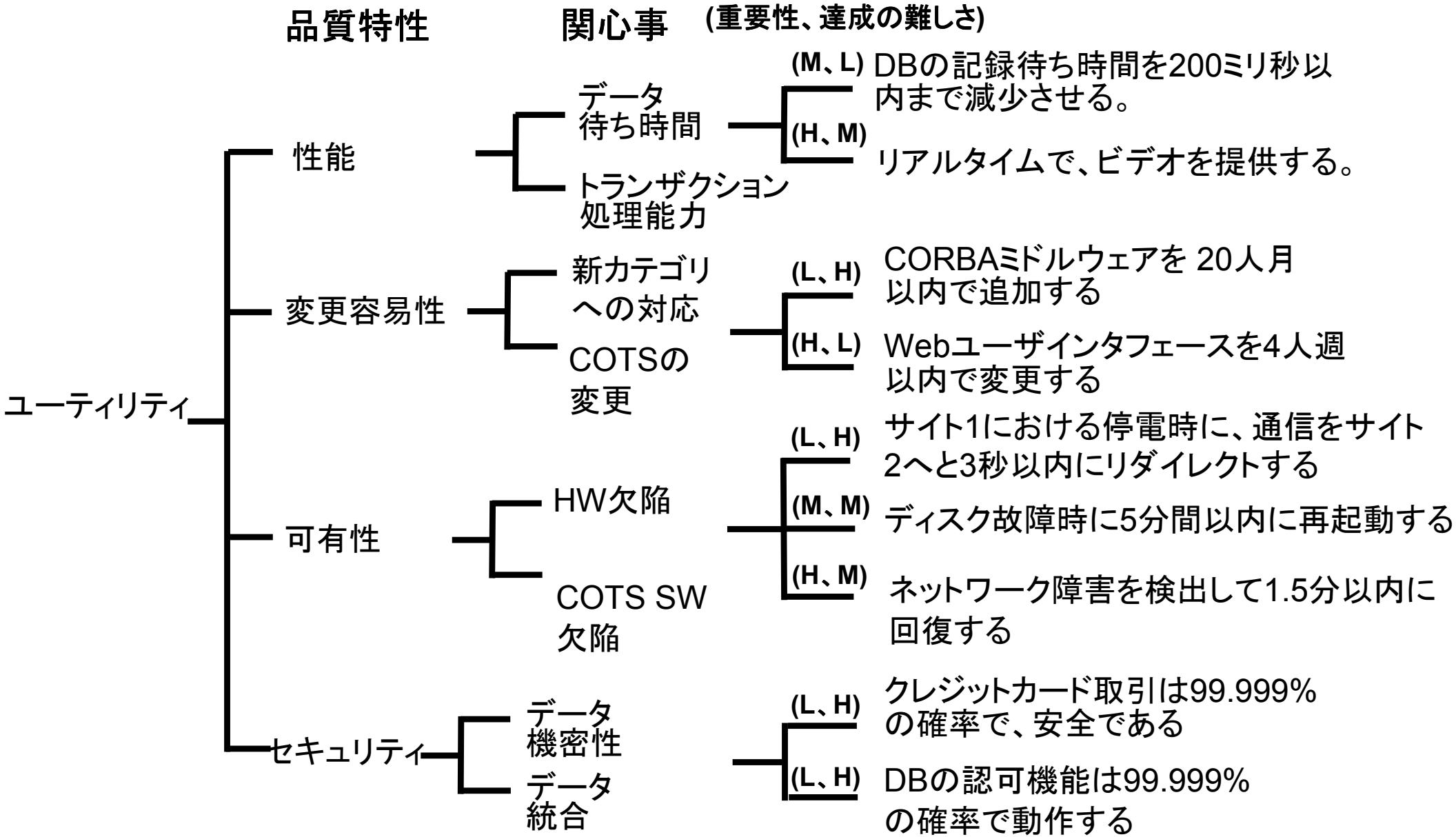
ATAMの手順

1. ATAMの提案と理解
2. ビジネスドライバーの明確化
3. 最初のアーキテクチャ提案
4. アーキテクチャ的手法の特定
    1. 品質要求を実現するための主要な箇所
    2. 支配的なスタイル/パターン、実現手法、原則など
5. 品質特性ユーティリティツリーの作成
6. アーキテクチャ的手法の分析
    1. 設計アプローチを品質特性に関して調査
    2. リスク、センシティビティポイント、トレードオフ
7. ブレインストーミングとシナリオの優先順位付け
    1. 利害関係者を含めた議論、新シナリオ（あれば）
8. アーキテクチャ的手法の再分析
9. 結果のまとめ

出典: [Chaudron]

ユーティリティツリー

- 詳細化され優先順位付けられた品質要求



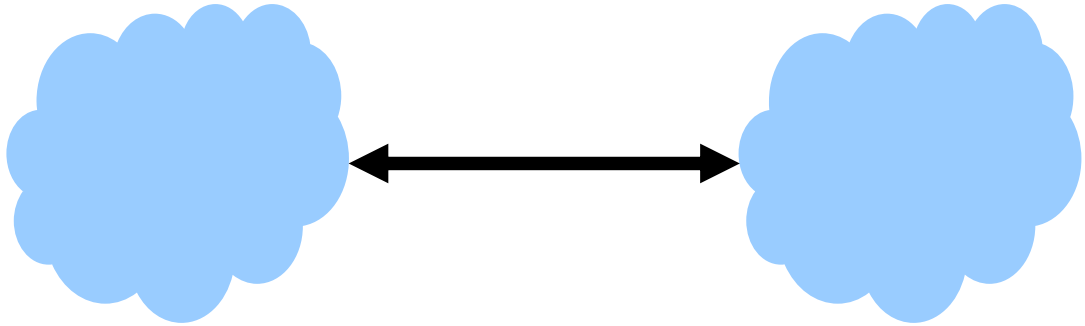
出典: [Chaudron]

センシティビティポイント

対象とする品質特性について肯定的に
作用するアーキテクチャ的手法

例:

- 性能



スループットが1つのチャネルに依存する場合

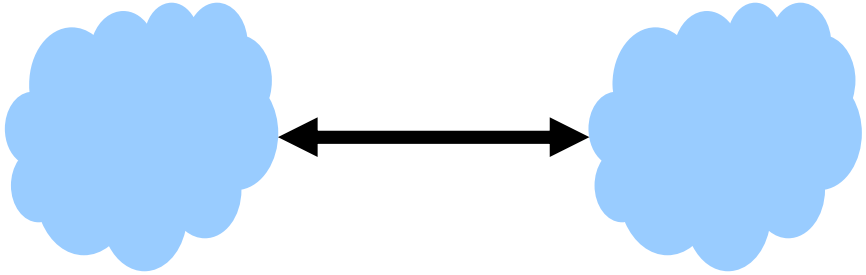
チャネルスピードの向上　⇒　性能の向上

出典: [Chaudron]

トレードオフポイント

複数の品質特性について異なる方向に影響する事柄

- 性能
- 信頼性
- セキュリティ



チャネルスピードの向上 ⇒ 性能の向上 ↑ & 信頼性の低下 ↓

暗号化の強化 ⇒ セキュリティの向上 ↑ & 性能の低下 ↓

出典: [Chaudron]

リスクとノン・リスク

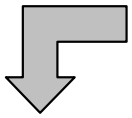
- リスク: 品質要求の観点から見て望ましくない結果を引き起こす可能性のあるアーキテクチャ的手法
- ノン・リスク: 分析の結果、問題ないと見なされる決定

⇒トレードオフポイント（およびセンシティビティポイント）は、リスクの候補

出典: [Chaudron]

品質シナリオ単位の評価例

	シナリオ1			シナリオ2		
	S	T	R	S	T	R
手法1	Y	-	Y			
手法2	-	Y	Y			



シナリオ	S1, メインスイッチのハードウェア故障を検出して復旧すること		
品質特性	可用性		
環境	通常動作		
刺激	CPUの1つが故障する		
応答	スイッチの可用性0.99999を達成する		
アーキテクチャ上の決定	**センシティビティ**	**トレードオフ**	**リスク**
A1: CPUのバックアップ	SE1: 異なるハードウェアとOSを使って普通のモードが故障しないことを確実にする。		R1: 復旧の実現のためにバックアップは必須である。
A2: データチャネルはバックアップし	SE2: ・・・	T1: S11における性能の達成に貢献するが、S12における可用性を下げる可能性があ	R2: データチャネルのバックアップが無いので、可用性の要求にはリスクがある。
A3: ウォッチドッグ	SE3: ・・・		
A4: ハートビート	SE4: ・・・		
A5: フェイルオーバールーティング	SE5: ・・・		

出典: [Chaudron]

ATAMのまとめ

- 品質についてアーキテクチャ設計上の決定を評価
- 利点
  - 期間の厳しい開発において、アーキテクチャについて熟考し合意する時間を設定
  - 早期のリスク特定、利害関係者にとって本質的な機能/特性に集中
  - アーキテクチャ文書/モデルの改善
- 留意
  - 参加者の経験に依存する主観的な判定である
  - 明確化された品質特性上の要求が必須
  - 利害関係者、アーキテクトのアクティブな参加が必須
  - 既知の実現手法やアーキテクチャの十分な理解が必須
- 応用
  - 実装への適用可能性
  - コスト分析との併用・統合

付録: 実現手法、アーキテクチャパターン

実現手法: 可用性

障害を遮断あるいは修復し、故障無く動作する能力

- 障害検出
  - ピン（ping）／エコー: ピンを発信して精査対象から時間内にエコーを受信
  - ハートビートクラスタリング: 定期的な生存メッセージの発信と受信
  - 例外: 例外の発行と受け取り
- 修復準備および修復
  - 投票: 冗長なモジュール群に同一入力を与えて、出力群から多数決などで決定、障害モジュールの検出
  - 動作中の冗長性: バックアップを並行実行させて障害発生時に切り替え
  - 受動的な冗長性: 障害発生時に予備モジュールに通知してバックアップ再開
  - 予備: 予備のシステム・モジュールを用意し、本システムの状態変化ログから状態を定期的に同期させて、障害発生時に移行・交換
- 再開始
  - シャドウオペレーション: 問題のあるモジュールを隠して動作させ、問題なく振舞うことを確認するようになってから戻す
  - 状態の再同期: 冗長構成時の状態の更新
  - ロールバック: 定期的な状態記録と障害時の復旧
- 予防措置
  - サービスからの除去: メモリリークなどを防ぐためのモジュールの一時的退避・再起動
  - トランザクション: 一度に戻すことのできる連続した手順群のまとめ
  - プロセス監視: 動作プロセスの監視と機能不具合時の削除、新プロセスインスタンス生成、初期化

実現手法: 変更容易性

要求や環境の変化時に必要な変更コスト・時間が小さい能力

- 変更の局所化
  - 意味の一貫性: 意味に一貫性のある責務群の同一モジュールへの割り当て
  - 起こりうる変更の予測: 変更を予測し修正が必要なモジュールを制限するように責務割り当て
  - モジュールの汎化: モジュールを一般化し、変更要求を入力の調整で実現
  - 選択肢の制限: 可能な選択肢を制限して変更の影響を下げる
  - 共通サービスの抽出: 共通の処理・サービスを特別なモジュール群にまとめあげる
- 波及効果の抑制
  - 情報隠蔽: プライベート、パブリックの選択
  - 既存インタフェースの維持: インタフェースの安定化
  - 通信経路の制限: あるモジュールとデータを共有する他のモジュール群の制限
  - 仲介者の使用: リポジトリ、Façade, Bridge, Mediator, Strategy, Proxy, Factory,Broker,、ネームサーバなど
- 束縛時期の延期
  - 実行時の登録: プラグ＆プレイ動作のサポート、Publish-Subscribeなど
  - 構成ファイル: 起動時のパラメータ
  - ポリモルフィズム: オブジェクト指向のメソッド動的束縛
  - コンポーネント交換: ロード時の束縛
  - 定義プロトコルの厳守: 独立プロセスの実行時の束縛

実現手法: 性能

時間的制約を満たす能力

- リソース要求
  - 演算処理効率の向上: アルゴリズム改良による待ち時間減少
  - オーバーヘッドの低減: リソース要求の制限
  - イベント発生率の管理: 外部からのイベント到着の規制
  - サンプリング頻度の抑制: 少ない頻度でイベント監視・サンプリング
  - 実行時間の制限
  - キューサイズの制限
- リソース管理
  - 並行性の導入: 並行、並列処理
  - 複数のコピーを確保: データや処理のコピー保持
  - 使用可能なリソースの増強
- リソース調停
  - スケジューリング方針: FIFO, 固定優先順位、動的優先順位、静的

実現手法: セキュリティ性

外部からの攻撃に抵抗しデータ・情報を保護する能力

- 攻撃の防御
  - ユーザ認証
  - ユーザ権限: 権限を与える、アクセス制御
  - 機密性保持: 暗号化など
  - 完全性の維持: チェックサムなど
  - 公開の制限: データ、サービスの公開を制限
  - アクセスの制限: ファイヤウォールなど
- 攻撃の検知
  - 侵入検知: トラフィックパターンの既知攻撃パターンへの整合など
- 攻撃から回復
  - 復旧（可用性と同様）
  - 識別・監査証跡: トランザクションの記録など

実現手法: テスト容易性

拡張・修正したソフトウェアのテストによる妥当性確認を容易にできる能力

- 入出力の管理
  - 記録と再生: 入出力を記録してテストに活用
  - 実装とインタフェースを分離: テスト用の実装の置き換え
  - 特殊なアクセス経路／インタフェース: テスト用インタフェース
- 内部監視
  - 内臓モニター: 状態、容量などの情報へのアクセスを可能に。イベント記録。

実現手法: 使いやすさ

利用者が必要な作業を容易に達成できる能力

- ユーザインタフェースを分離: ユーザインタフェースを変更できるように分離設計、MVC, PACなど
- ユーザ主導を支援
  - キャンセル
  - アンドゥ
  - 集約: 複合コマンド
  - マルチビュー
- システム主導を支援
  - ユーザモデルの維持: 利用者の特徴（例えば知識）に応じた対処
  - システムモデルの維持: システムの期待される振る舞いに応じた対処
  - タスクモデルの維持: タスクモデルを用いて文脈から利用者の意図を推測、対処

代表的なアーキテクチャパターン

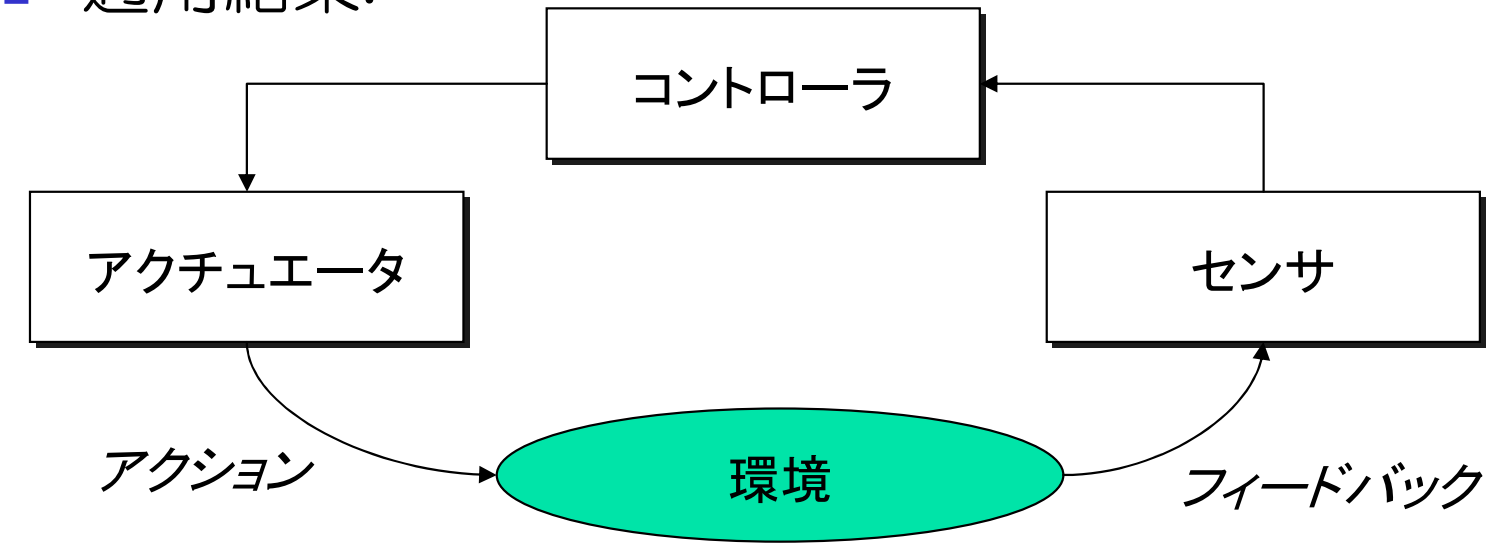
- 典型的な優れたアーキテクチャ設計を導出する過程をパターンとして表したもの
- 下記については[オブジェクトハンドブック] を参照すると良い

- 基本的アーキテクチャ
  - 多層（Layers）アーキテクチャ
- ドメインに応じた選択的アーキテクチャ
  - GUIドメイン
    ⇒ MVC（Model-View-Controller）アーキテクチャ
  - 分散ネットワークドメイン
    ⇒ ブローカ（Broker）アーキテクチャ
  - 小規模コマンドドメイン
    ⇒ パイプ&フィルタ（Pipe & Filter）アーキテクチャ

[オブジェクトハンドブック] 永和システムマネジメント, オブジェクトハンドブック2001, http://www.objectclub.jp/technicaldoc/object-orientation/handbook

Control Loop

- 扱う問題: 環境へのリアクティブな対応
- 構造: コントローラ、アクチュエータ、センサ
- 振る舞い: 以下の繰り返しで目的を達成する
  - センサは環境から入力を受けて、コントローラに渡す
  - コントローラは入力値を解釈して次のプランを決定し、アクチュエータに指示する
  - アクチュエータは環境に対して動作する（アクションをおこす）
- 適用結果:



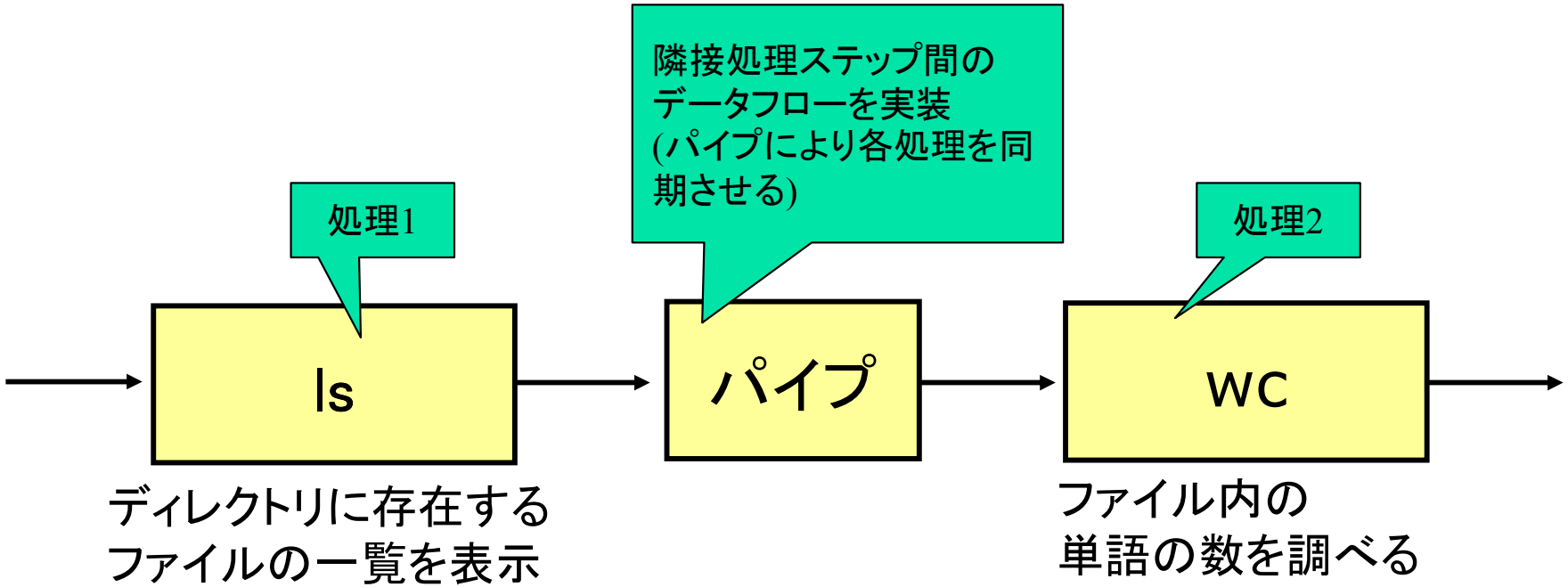
Pipes&Filters(1/2)

- 名前：Pipes & Filters(パイプ&フィルタ)
  - あるモジュールの出力が、他のモジュールの入力となる設計
- 状況：入力データストリームを処理/変換しなければいけないシステムを構築。
- 問題：システムに柔軟性を持たせる必要がある。
- フォース
  - ステップの多重処理(ステップを並列/擬似並列に実行)を除外できない。
  - 処理ステップの交換やその組み合わせ変更によって、将来のシステム拡張が可能であるべき。
  - 1つの処理ステップが小さいことが望ましい。
  - 非隣接処理ステップは、情報を共有するべきではない。
  - 入力データの発生源が複数存在する。
  - さまざまな方法で、最終結果を表示したり保存したりできるべき。
  - 今後の処理のために中間結果をファイルに明示的に格納すると、ディレクトリが煩雑になり、誤りが生じやすい。

[POSA00]より

Pipes&Filters(2/2)

- 例：Unixのシェル(shell)で書かれたプログラム
  - コマンド例：　ls | wc



Layers(1/3)

- 名前： Layers(レイヤ)
  - 全体を変化のしやすさに応じて多層（レイヤ）に分割し、変化しにくい層から順に上へ積み上げたアーキテクチャ
  - 層間の直接的な呼び出し（メソッド実行など）を、変化しやすい上層から、変化しにくい直下層に制限
  - 直接的な依存関係を上から下への方向に固定
  - 変更の影響は依存の方向とは逆方向に伝わるため、保守性が高くて変更に強いアーキテクチャを得たいという非機能要求を実現
- 状況：「抽象度の異なる要素が混合する」という主要な特徴を持ったシステムを構築。抽象度の高い要素は、それよりも抽象度の低い要素が提供する操作だけを使用する。

[POSA00]より

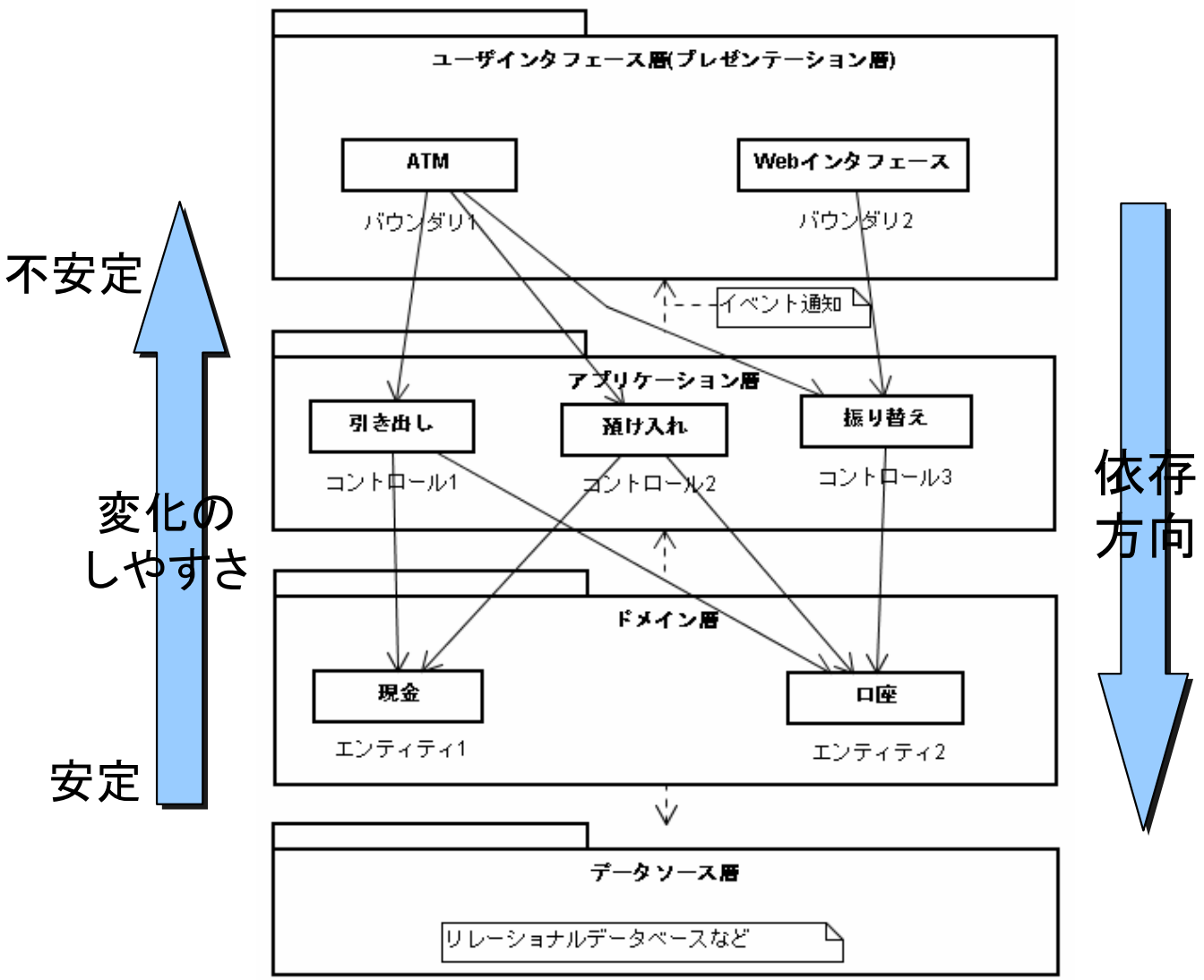
Layers(2/3)

- 問題：水平的な構造化が必要である。
- フォース
  - コンポーネント境界をはさんでデータのやり取りが行われると、パフォーマンスの低下を招くことがある。
  - ソースコードを変更しても、システムに影響を及ぼすべきではない。
  - インタフェースの変更があってはならない。
  - システムの部分要素(part)は交換可能であるべきである。他のシステム要素に影響を及ぼすことなく、コンポーネントの実装を別の実装で置き換え可能であることが望まれる。
  - 現在開発中のシステム用に設計された下位レベルの要素が、将来、別システムの開発で利用される可能性がある。
  - 解読性と保守性を向上させるために、システム上の類似した責務をグループ化するべきである。
  - コンポーネント粒度に「標準」が存在しない。
  - 複合コンポーネントに対して、さらに分割を行う必要がある。
  - システムは、プログラマのチームによって実装されるものである。

[POSA00]より

Layers(3/3)

- 適用事例
  - 仮想マシン
  - 一般的な情報システム
- アーキテクチャ設計への適用例



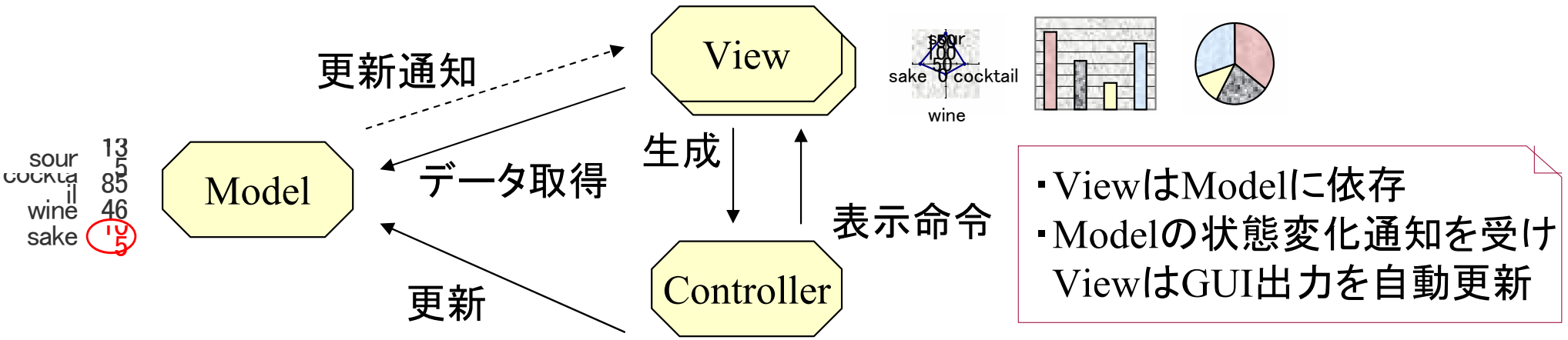
MVC(1/2)

- 状況: 同一データについて多様なGUIを持つ対話型ソフトウェアの設計。データ更新に基づいて複数GUIにおける表示に同期をとりたい。
- 問題:
  - ユーザインタフェースは、要求仕様の変更による影響度が大きい。
  - 複数ユーザが互いに矛盾するユーザインタフェースを要求することがある。
  - ユーザインタフェースと機能中核部分の結合度が高い場合には、誤りを潜在的に持つ可能性が高くなる。
- フォース:
  - 同一情報を別々のウィンドウに異なる形式で表示する必要がある。
  - データに対して行われた捜査が、直ちにアプリケーションの表示と動作に反映されなければならない。
  - ユーザインタフェースの変更を容易に行うことができる。また、実行時においても、その変更を行うことができることが望ましい。
  - 異なるルック＆フィールド標準のサポートやそれらの間の移植が、アプリケーションの中核となるコードに影響してはならない。

[POSA00]より

MVC(2/2)

- 解決: 以下の構造として設計する。
  - Model ：核となるデータと機能をカプセル化
  - View ：情報をユーザに提示
  - Controller：入力装置からユーザ入力を受取る
- 適用事例:
  - Smalltalk
  - MFC
  - Java/Swing



参考文献

- [徳本07]徳本: UTM アプライアンスのログ情報解析ツールの開発, NII/TopSE 2007年度公開シンポジウム, 2007, http://www.topse.jp/events/symposium07/tokumoto.pdf
- [繁在家08]繁在家、鷲崎: 品質駆動型設計によるWEBシステムの開発,第160回 ソフトウェア工学・第9回 組込みシステム　合同研究発表会IPSJ SIGSE-160, 2008.
- [Bass05] L. Bass, et al. 前田ほか訳: 実践ソフトウェアアーキテクチャ（Software Architecture in Practice 2nd Ed.）, 日刊工業新聞社, 2005
- [ATAM] CMU/SEI, The Architecture Tradeoff Analysis Method (ATAM), http://www.sei.cmu.edu/architecture/ata_method.html
- [ABAS] CMU/SEI, Attribute-Based Architectural Style (ABAS), http://www.sei.cmu.edu/architecture/abas.html
- [ADD] CMU/SEI, Attribute-Driven Design Method, http://www.sei.cmu.edu/architecture/add_method.html
- [SAAM] CMU/SEI, Scenario-Based Analysis of Software Architecture, http://www.sei.cmu.edu/architecture/scenario_paper/
- [Chaudron] Michel Chaudron, Software Architecting, 2003-2004, http://www.win.tue.nl/~mchaudro/swads/
- [岸05] 岸、野田、深澤: ソフトウェアアーキテクチャ, 共立出版, 2005
- [Bosch00] J. Bosch: Design and use of software architectures, Addison-Wesley, 2000

参考文献（つづき）

- [井上03] 井上健, "ソフトウェアパターン入門", IPSJ/SIGSEパターンワーキンググループ設立記念セミナー発表資料, 2003, http://patterns-wg.fuka.info.waseda.ac.jp/ および、羽生田栄一 監修, パターンワーキンググループ著, ″ソフトウェアパターン入門～基礎から応用へ～″, ソフトリサーチセンター, 2005.
- [羽生田05] 羽生田栄一, "ソフトウェアパターン講義", SRCセミナー, 2005.
- [細谷03] 細谷, "形から入らないパターン活動", IPSJ/SIGSEパターンワーキンググループ設立記念セミナー発表資料, 2003
- [Jacana] Godfrey Jackson, "Jacana: A Pattern Language", http://www.jacana.org.uk/pattern/
- [JavaWorld0501] 榊原彰 他, "ジャバワールド 2005年1月号",IDGジャパン,2005.
- [山本06] 山本理枝子, "ソフトウェアパターンの導入事例", TopSE第12回講義予定, 2006.
- [鷲崎05] 鷲崎弘宜 他, "ソフトウェアパターン研究の発展経緯と最近の動向", IPSJ第147回ソフトウェア工学研究会, 2005.
- [Meszaros97] Gerard Meszaros and Jim Doble, ″A Pattern Language for Pattern Writing,″ in Pattern Languages of Program Design, Vol.3, 1997. http://hillside.net/patterns/writing/patternwritingpaper.htm
- [Rising99] Linda Rising: Patterns Mining, in Handbook of Object Technology, CRC Press (1999)
- [Gabriel] R. Gabriel: A Timeless Way of Hacking, in Core J2EE Patterns, Pearson Education, 2001
- [Tracz88] Will Tracz: RMISE Workshop on Software Reuse Meeting Summary, In Software Reuse: Emerging Technology, IEEE CS (1988)
- [JIS0701] 日本工業規格: JIS X 0701 ドキュメンテーション用語（基本概念）(1989)
- [Hagge05] L. Hagge and K. Lappe, ″Sharing Requirements Engineering Experience Using Patterns,″ IEEE Software, Vol.22, No.1, 2005.
- [紫合05] 紫合治 他, ″ウィンターワークショップ2005・イン・伊豆参加報告″, IPSJ第148回ソフトウェア工学研究会, 2005.
- [久保05] 久保淳人 他, ″パターンマイニングによるソフトウェア要求の獲得知識の記述″, ソフトウェア工学の基礎ワークショップ (FOSE2005), 2005.
- [Coplien98] James Coplien, "Software Development as Science, Art, and Engineering." In Linda Rising (collected). The Patterns Handbook, Cambridge University Press, 1998.
- [Martin04] Robert C. Martin（著）, 瀬谷啓介（訳）, ″アジャイルソフトウェア開発の奥義: オブジェクト指向開発の秘伝書″, ソフトバンクパブリッシング, 2004.