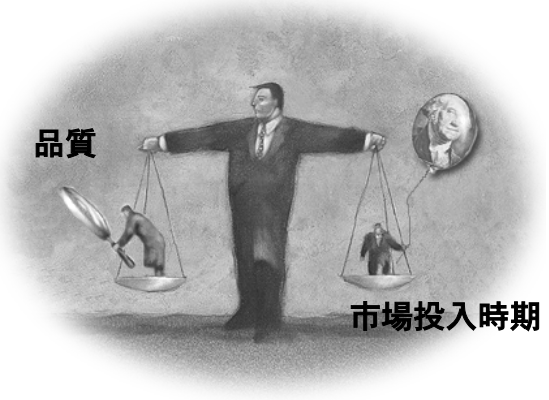
IBM Software Group

UMLに惑わされないコンポーネントベース開発

Rational. software

e business on demand software

真実?

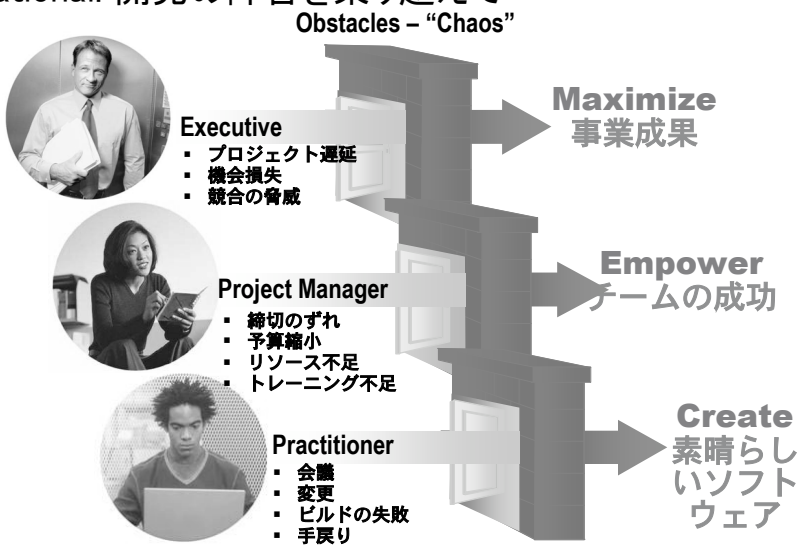


組み込みソフトウェア開発の課題

アプリケーションの複雑性	環境の複雑性	プロセスの複雑性
▪ 強力なタイミング制約 ▪ 少ないメモリー ▪ 並列、分散、ネットワーク化 ▪ イベント駆動	▪ 多くのRTOS ベンダー ▪ 多くのチップベンダー ▪ 複数のIDE ▪ 限定されたホスト、ターゲット接続性 ▪ 低いビルトインデバッグ能力	▪ 移ろう要求 ▪ 設計の変換エラー ▪ 共通理解の欠如 ▪ 困難な保守 ▪ 低いパフォーマンス ▪ アーキテクチャ上の強制 ▪ 不十分なテスト ▪ 信頼できないビルド ▪ 統合されていない ▪ 発見の遅れ ▪ テスト、デバッグが困難

Obstacles to success

Rational: 開発の障害を乗り越えて



なにがおこっているのか?

- 組み込みプロジェクトの50%以上が数ヶ月スケジュールが遅延している1
- 組み込みプロジェクトの25%が放棄された2
- 設計の44%だけが、予想の20%程度におさまった1
- 全開発作業の50%以上がテストに費やされている

[1]Electronics Market Forecasters, April 2001
[2]Embedded Developer Systems Survey, Summer 2001

課題

- 組み込みソフトウェア開発における課題…
  - 品質の維持／向上
  - 高い保守性
  - 開発効率の向上
  - 開発期間の短縮

課題を乗り越えるために

- 品質の維持／向上
  - 要求管理
  - 回帰テスト
  - 構成変更管理
- 高い保守性
  - 要求管理
  - コンポーネントに基づく開発（CBD）
  - 構成変更管理
- 開発効率の向上
  - ビジュアルモデリング(UML)
  - コンポーネントに基づく開発（CBD）
- 開発期間の短縮
  - ツールによる自動化
  - 反復プロセスによるリスク管理

UMLは課題克服のための一つの手段に過ぎない！

なぜモデリングするのか？

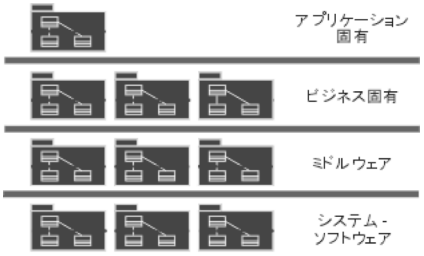
- モデルはシステムを単純化して、特定の観点からシステムの重要な要素を示し、次の目的のために役立ちます
  - 複雑なシステムを理解するための支援
  - 設計の代替案の経済的な調査と比較
  - 実装用の基盤の形成
  - 要求の正確な把握
  - 決定事項の明確な伝達
- 組込み開発では…
  - 複雑化するアプリケーション、システムを扱う
  - 並列性、並行性を扱う
  - （コンポーネント）アーキテクチャの可視化

UMLモデルの使い方を正しく理解しましょう

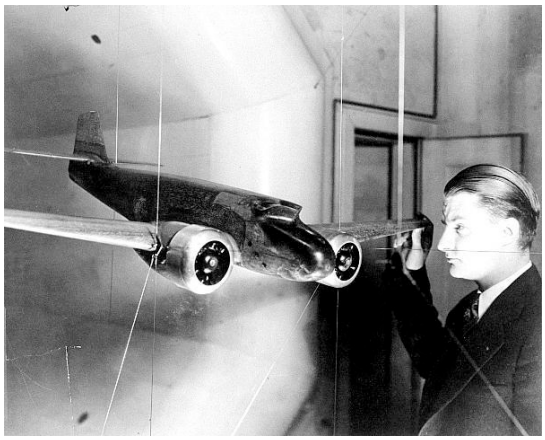
コンポーネントに基づくアーキテクチャを使用すべし

- コンポーネントに基づくアーキテクチャは、置き換え可能なコンポーネントに基づくアーキテクチャで、独立していて置き換えが可能なモジュール コンポーネントを使用するため、複雑なアーキテクチャを管理してコンポーネントの再利用を促進するのに役立ちます
  - Rational Unified Process(RUP)の実践原則のひとつ

※レイヤを使用したコンポーネントに基づくアーキテクチャ



UMLはアーキテクチャを可視化する！



エンジニアリング・モデル

なぜ、エンジニアはモデルを用いるか？

- 複雑なシステムをすべて理解するために…
  - モデルはシステムの割引いたバージョンで、本質を強調し、無関係なものをぼやかす ⇒ 抽象化
- …リスクの最小化:
  - 全リソースをコミットする前に要求や設計のエラーや手抜かりを見つける
- …利害関係者とのコミュニケーション
  - 顧客、利用者、実装者、テスター、文書作成者、他
  - 要求と設計のトレードオフについて
- …実装するため
  - 実装の青写真をモデルを用いる
  - ソフトウェアシステムにとって特別な重要性を持つ

有用なモデルの特徴

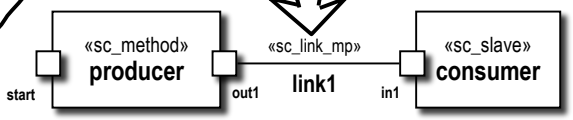
- 抽象化
  - 不要なものを隠蔽／排除するための重要な見方を強調
- 理解しやすい
  - オブザーバーが直感的に理解できる形式で記述
- 正確
  - モデル化したシステムを忠実に表す
- 予測性
  - モデル化したシステムに関するQ&Aとして用いることができる
- 廉価
  - モデル化したシステムを構築、理解するのに非常に安上がり

過去のソフトウェアモデルの多くこれらの見方を間違っている！

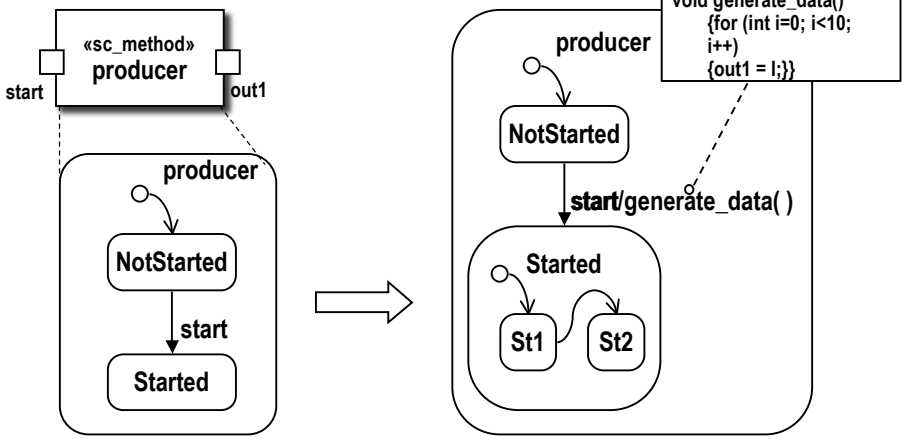
ソフトウェアモデル

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // to kick-start the producer
void generate_data ()
{
for(int i =0; i <10; i++).{
out1 =i ; //this will invoke the slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // declare as a module state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
```

```
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize the accumulator}};
SC_MODULE(top) // structural module
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```



ソフトウェアモデルの改良

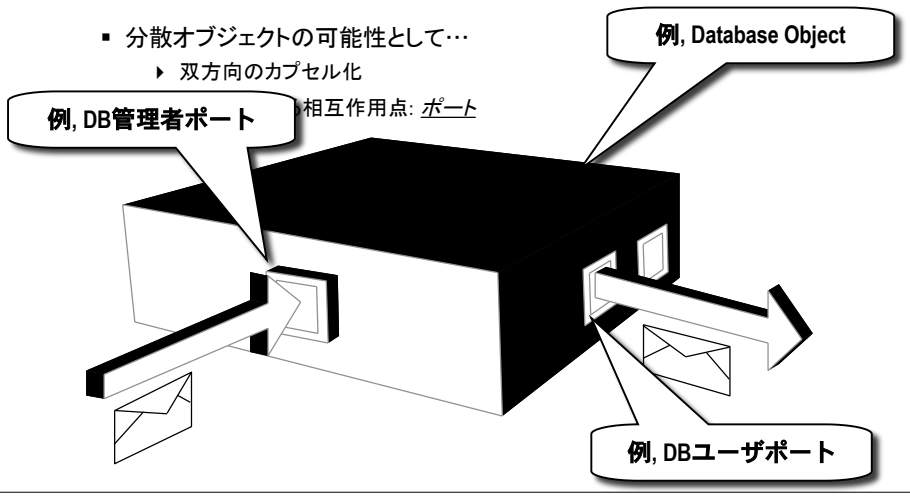


- モデルは、システムが完成するまで継続してリファインできる!

構造化クラス:
UML2.0においてアーキテクチャを扱う

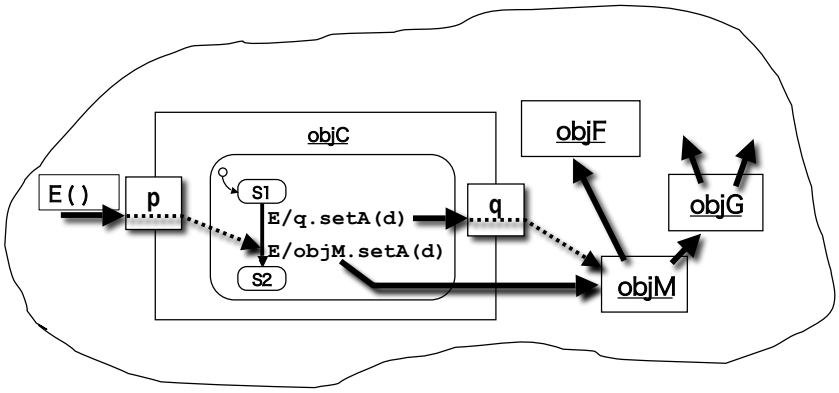
構造化クラス: 外部ビュー

- 分散オブジェクトの可能性として…
  - 双方向のカプセル化
  - 相互作用点: *ポート*



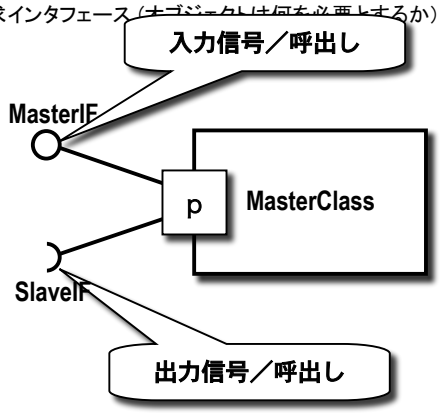
ポート

- 別々の異なる相互作用（並列も可）
- 環境とオブジェクト内部は完全に分離



ポートの表記とセマンティクス

- ポートは複数のインタフェース仕様をサポートできる
  - 提供インタフェース（オブジェクトは何ができるか）
  - 要求インタフェース（オブジェクトは何を必要とするか）



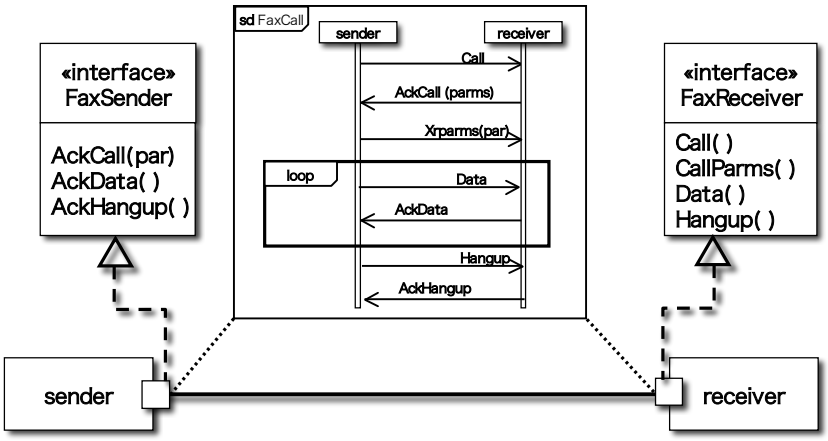
通信オブジェクトの組み立て

- ポートはコネクターで接続され、構造化クラスで構成された一対のコラボレーションを作る

コネクターのモデル通信チャネル
コネクターはプロトコルの制約をうける
静的ルールの適用 (プロトコル互換)

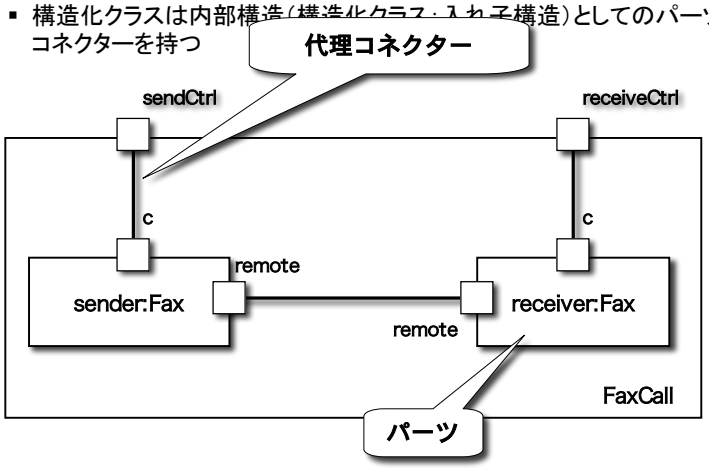
プロトコルとコネクター

- 再利用可能な動的コントラクトによる
  コネクターを通じた正当な相互作用



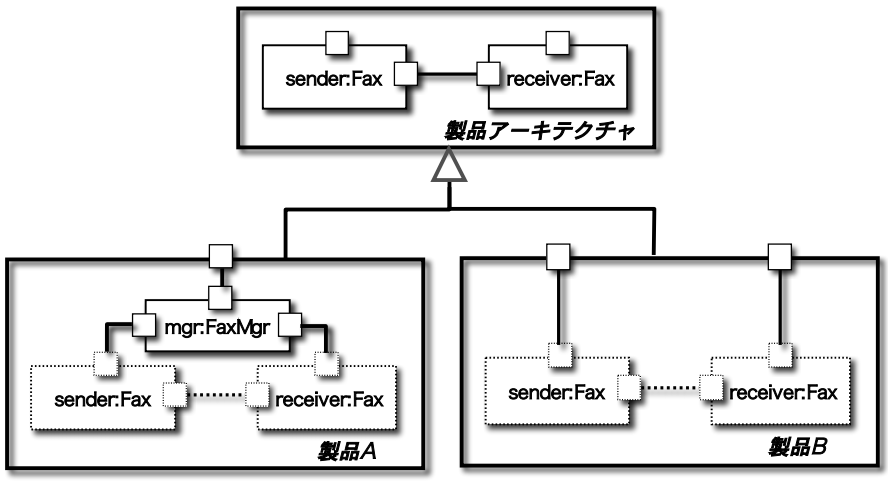
構造化クラス: 内部構造

- 構造化クラスは内部構造(構造化クラス・入れ子構造)としてのパーツとコネクターを持つ

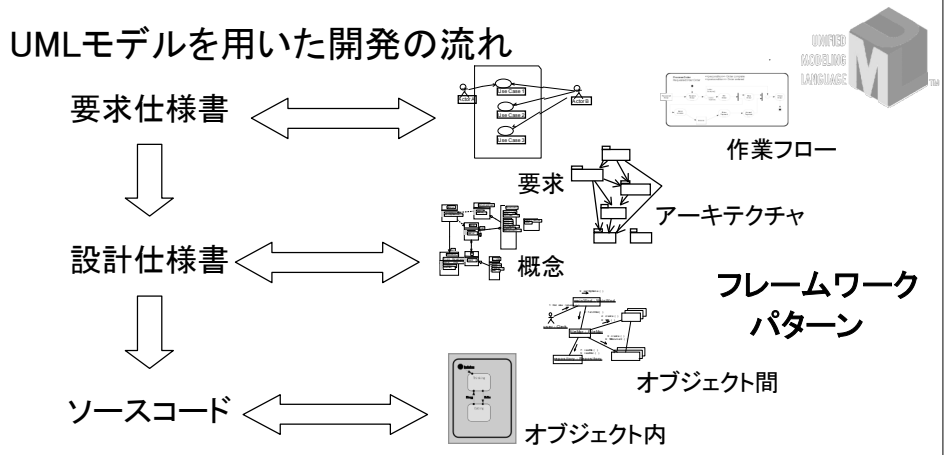


製品ファミリーのアーキテクチャ

- 標準継承メカニズムの使用 (違いによるデザイン)



UMLモデルを用いた開発の流れ



UMLによるモデル記述

従来の自然言語と
ソースコードの世界

モデルが成果物に加わる。
上位の設計情報を共有・再利用可能になる

構造化クラス：Rational Rose TechnicalDeveloper(RT)

RoseRT.exe

- 二つの業界標準ツールから:
  - Rational Rose ™
    - 最も利用されているビジュアルモデリングツール
    - 業界標準モデリング言語: UML
  - ObjecTime Developer ™
    - 組込みリアルタイムシステム用設計手法 ROOM
    - 実績のある自動コード生成技術
- その結果出来たのが:
  - 完全な実行イメージの生成 – UML model is the source
  - モデルを実行し、モデルのままでデバッグも行う
  - リアルタイム向きアプリケーションフレームワーク
  - ネイティブ開発 / クロス開発 どちらも可能

AUTOSAR:
車載組込みソフトウェアアーキテクチャ標準へ

AUTOSARとは？

- 目的
  - to run the SW-Component in an AUTOSAR compliant environment.
    - AUTOSAR RTE(Runtime Environment)
    - AUTOSAR interface + SW component
  - 自動車向け組込みソフトウェアの標準となる実行環境、コンポーネントアーキテクチャーの提供

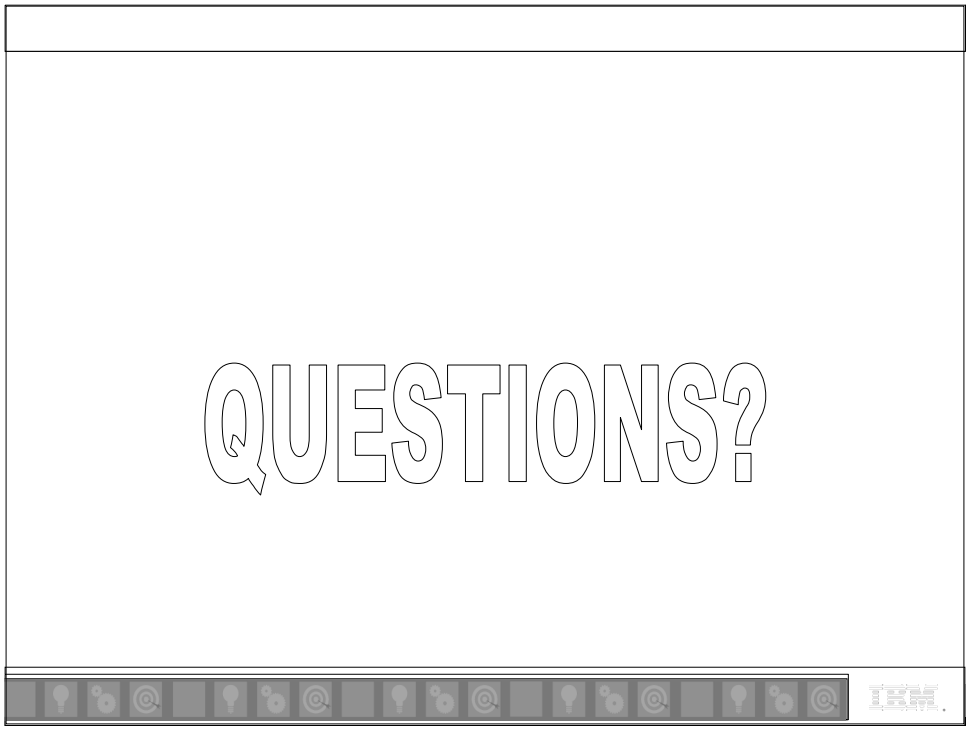
※「トヨタと日産自、車載電子制御システムのソフトウェアやネットワークの標準化及び共通利用を目的とした取組みを開始」、2004/9/9付、日経新聞

※　参照　http://www.autosar.org/

なぜコンポーネントアーキテクチャが重要なのか？

- ソフトウェア アーキテクチャに含まれるもの:
  - ソフトウェアシステムの構成についての重要な決定
  - 構造的な要素の選択と、システムを構成するインターフェイスおよびこれらの要素間のコラボレーションとして規定される振る舞い
  - 構造的要素と振る舞い的要素を、徐々により大きなサブシステムにするためのコンポジション。このような構成、要素、インターフェイス、コラボレーションおよびそれらのコンポジションをガイドするためのアーキテクチャのスタイル
- ソフトウェア アーキテクチャは、構造や振る舞いだけでなく、使い方、機能、性能、障害許容力、再利用、わかりやすさ、経済性と技術的制約のトレードオフ、美的な問題も扱います
- ソフトウェア アーキテクチャに焦点を当てると、コンポーネントが相互作用する基本的なメカニズムとパターンを含めた構造（コンポーネントと、コンポーネントを統合する方法）を明確にすることができます
  - プロジェクト管理の計画立案の側面をサポートし、並行して開発できるコンポーネントと順番に開発するコンポーネントをコンポーネントの依存関係から決定することができます。
- パッケージ、サブシステム、レイヤなどの概念は、分析と設計の段階でコンポーネントを編成してインターフェイスを指定するために使用します。

UMLを使ってコンポーネントとアーキテクチャ中心の開発へ！




Thank You!