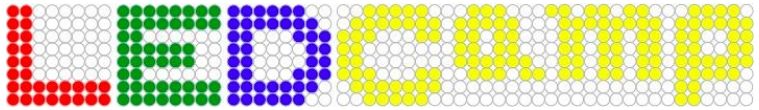
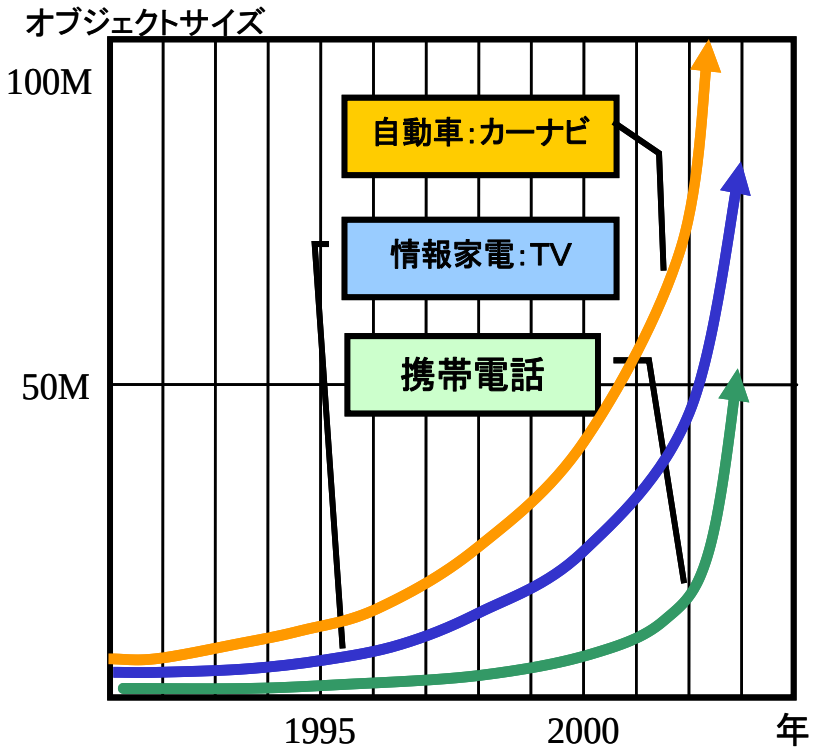


モデル駆動開発
～理論編～

大栄　豊

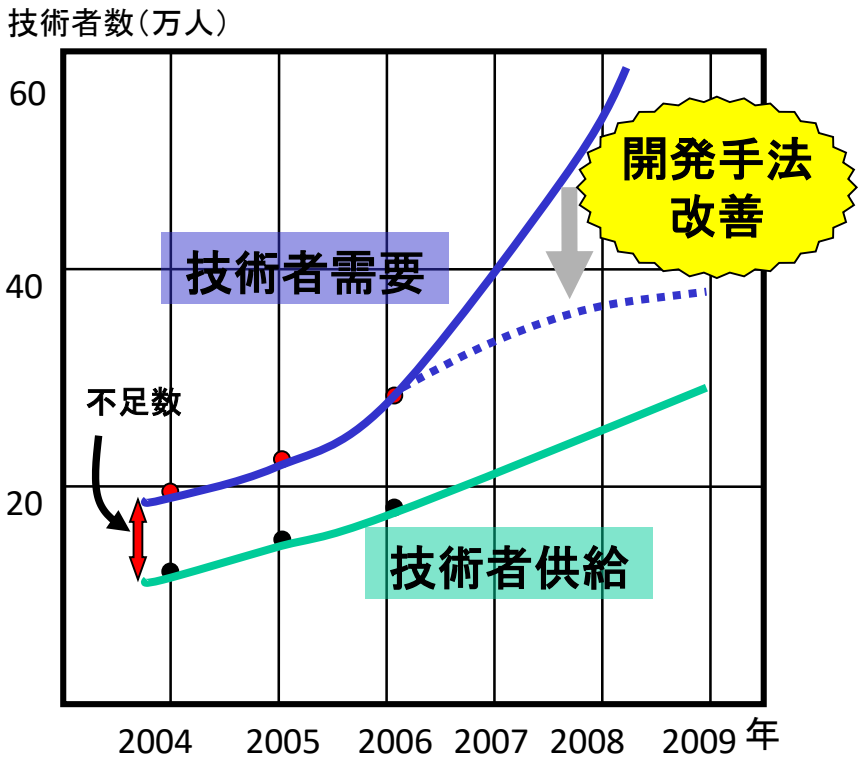
組込みシステム開発の現状(1)
組込みソフトウェア開発の危機

組込みソフトウェア開発量の指数的増大



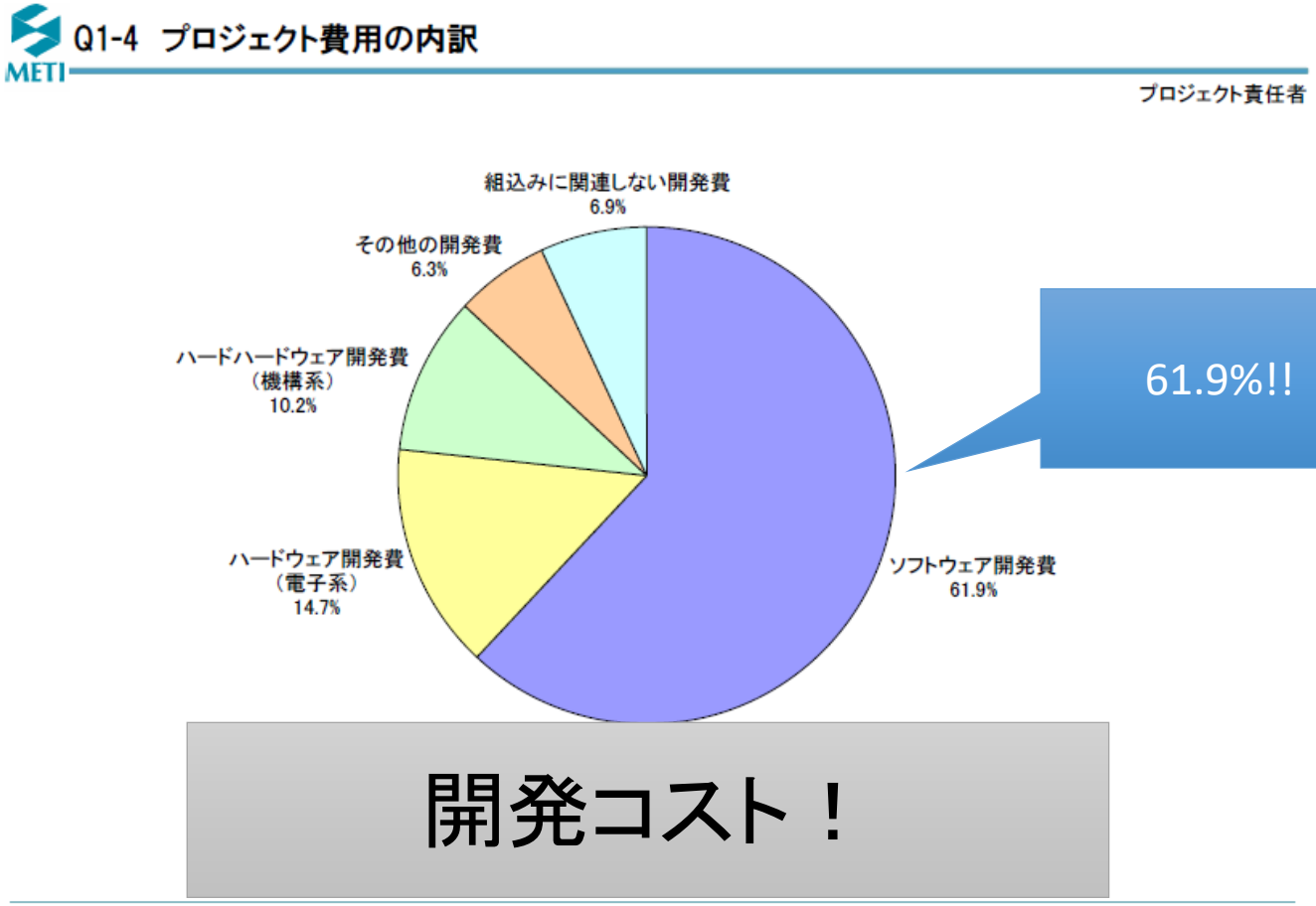
出典: 組込みソフトウェア開発力強化推進フォーラム(2004年6月)
日経エレクトロニクス 2000 9-1(no.778) をベース

組込みソフトウェア技術者需要数の指数的増大

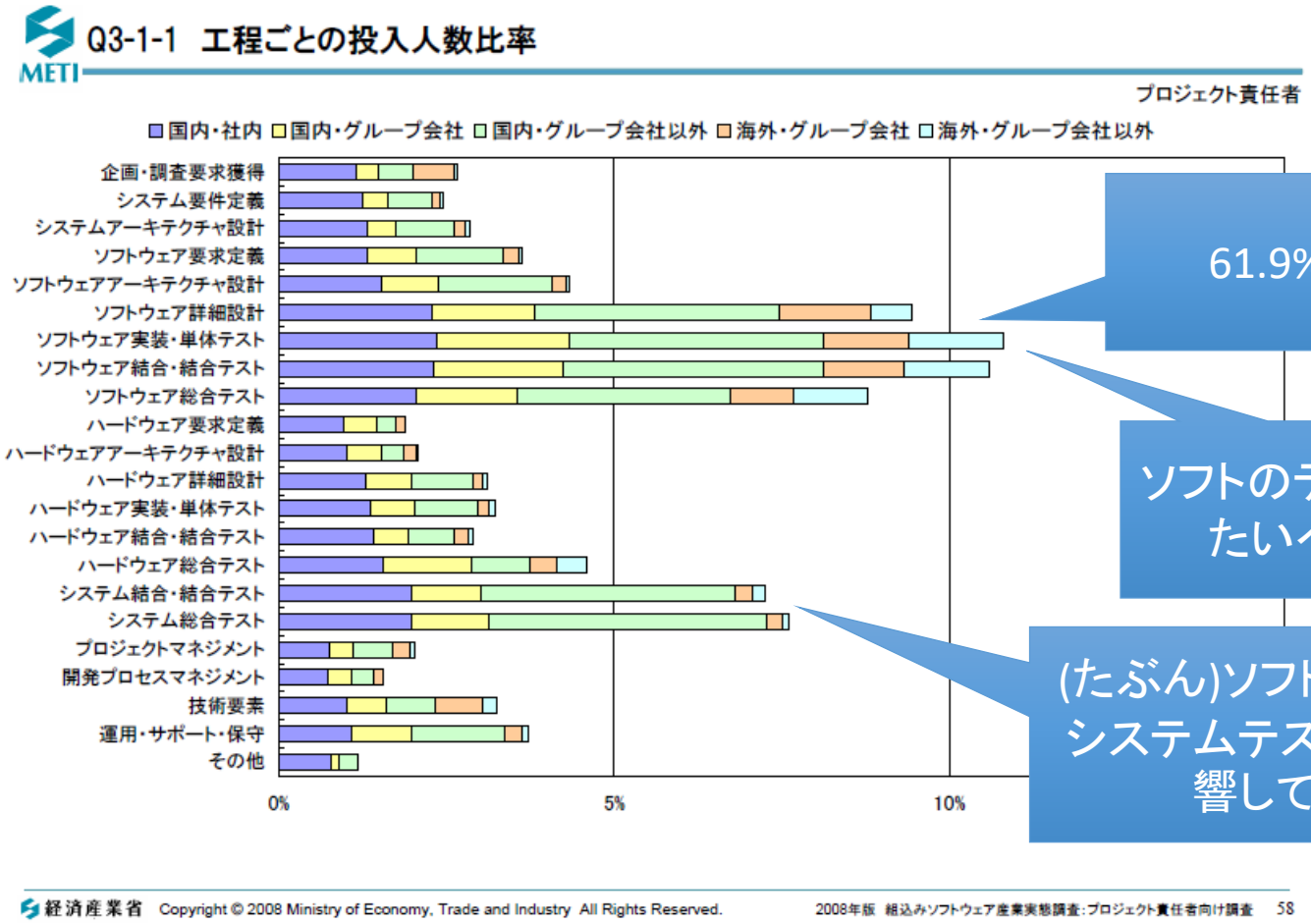


出典: 経済産業省組込みソフトウェア産業実態調査(2006年度版)

組込みシステム開発の現状(2)



組込みシステム開発の現状(3)



ソフトウェアとモデリング

- ソフトウェアは超複雑.
- 100万行のコードだと，単純換算で400pの文庫本×100冊．複数人で一切の誤字なく，不整合なく，書き上げる必要がある.

- コードだけ見て全体の関係や流れを把握するのは難しい．というか無理.

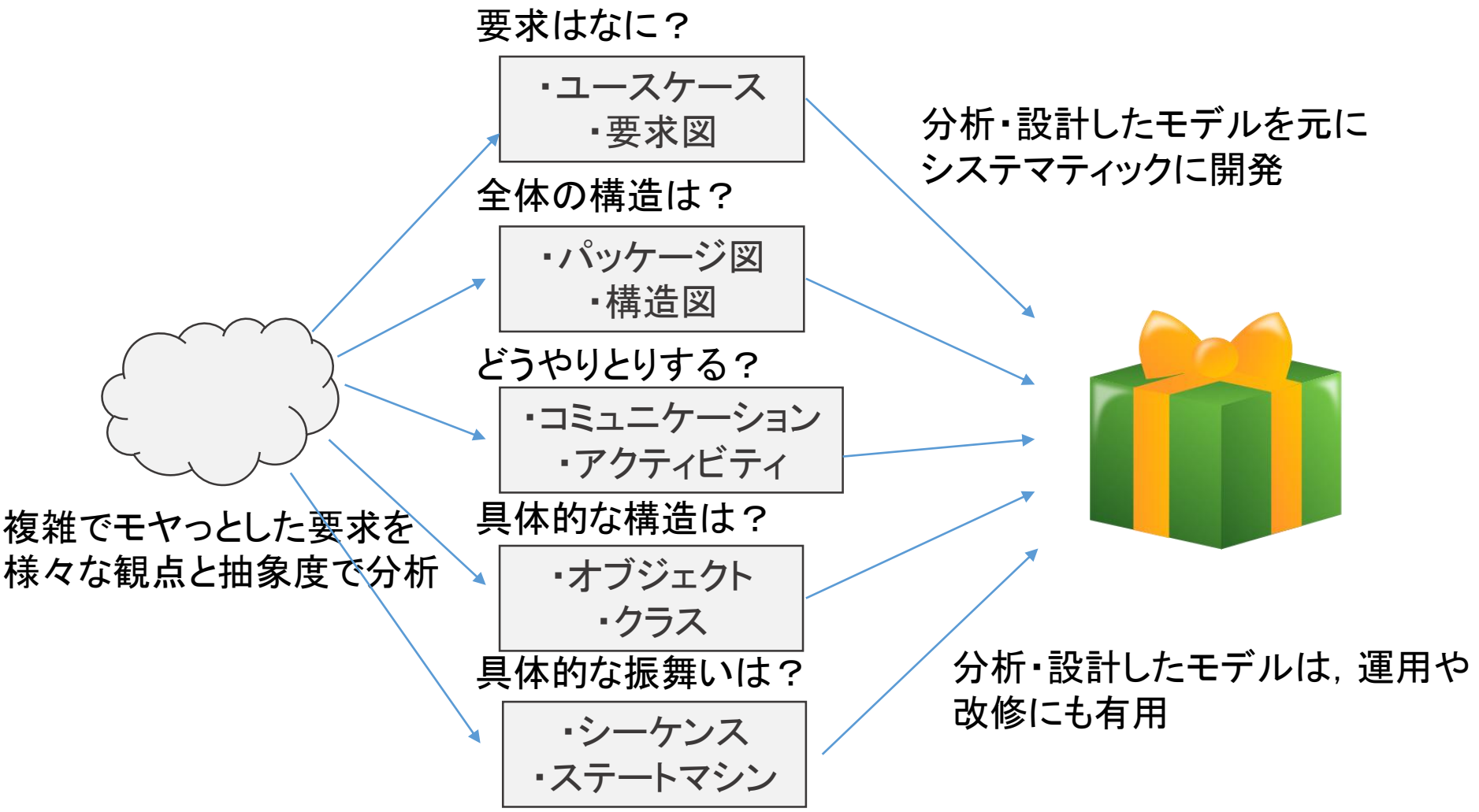
- 様々な観点と抽象度からソフトウェアを分析・整理することが必要.

設計せずに建築する事はない

- ソフトウェア開発は本質的に難しいもの.
- それを数十人で何ヶ月も掛けて作成する．設計無しではありえない.
- その後何年にも渡ってメンテナンスされる.

http://freeyork.org/art/deconstruction-of-buildings-by-michael-jantzen

観点と抽象度



UML図

- ソフトウェア開発で使いやすい図をまとめたもの.
- すべての図を使う必要はない.

- 観点と抽象度の枠組み.

- 個人のスケッチレベルであれば，好きに描いてよい.
- 複数人での開発であれば，表記ルールは守ること.　（共通言語の意味がなくなる.）
- ステークホルダーの合意が取れているのであれば，独自形式でもよい.　ー＞DSL(*1)など.

*1) DSL = Domain Specific Language

モデルとコードの乖離

要求はなに？

・ユースケース
・要求図

全体の構造は？

・パッケージ図
・構造図

どうやりとりする？

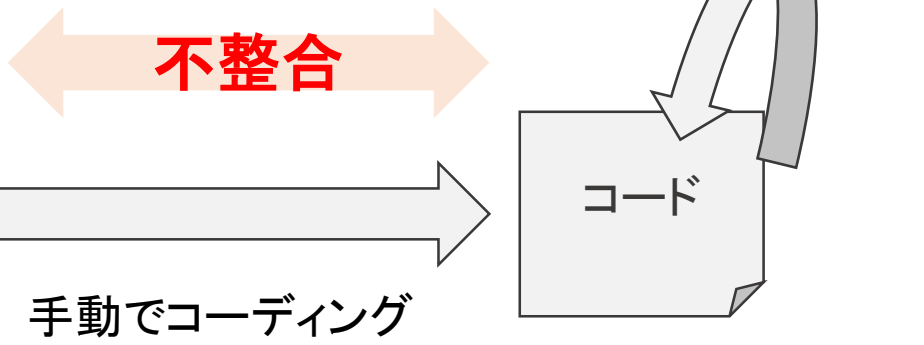
・コミュニケーション
・アクティビティ

具体的な構造は？

・オブジェクト
・クラス

具体的な振舞いは？

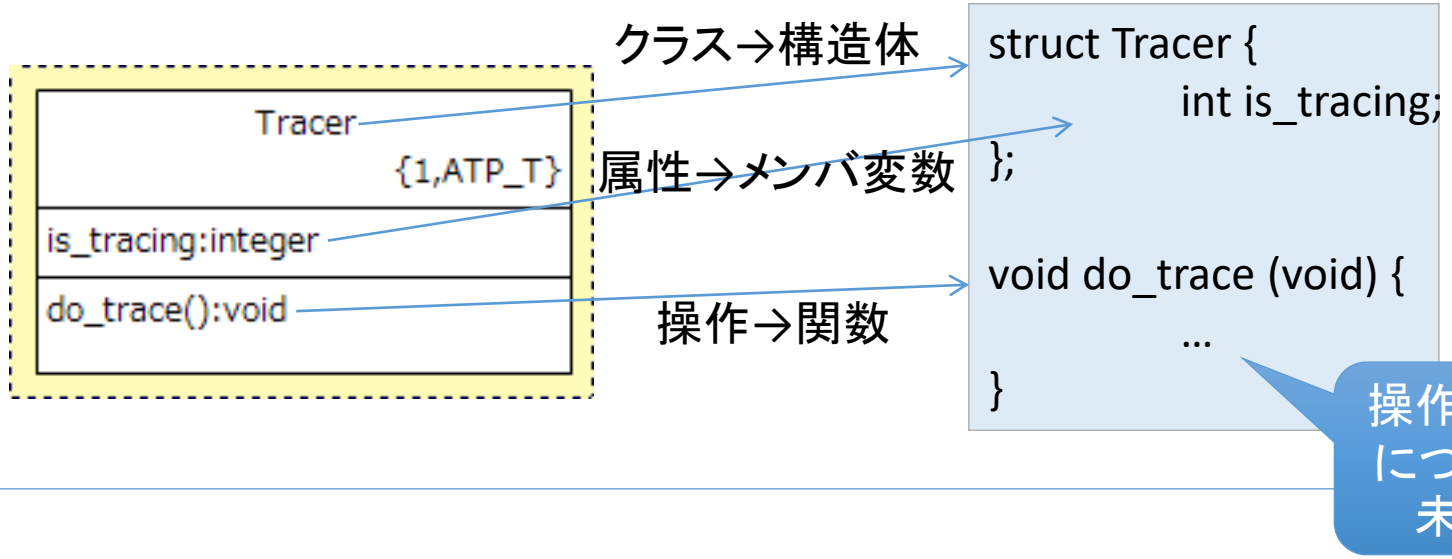
・シーケンス
・ステートマシン



モデルからコードへの変換～人間が実行（１）

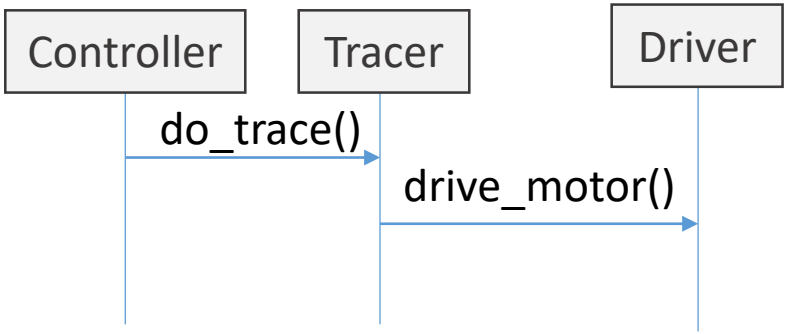
- クラスのソースコードへの変換を考えると・・・
  - 決まりきったソースコードのパターンがある.

モデル要素	ソースコード
クラス	構造体
属性	構造体のメンバ変数
操作	関数



モデルからコードへの変換～人間が実行（２）

- 振舞いの部分の実装は・・
- シーケンス図から



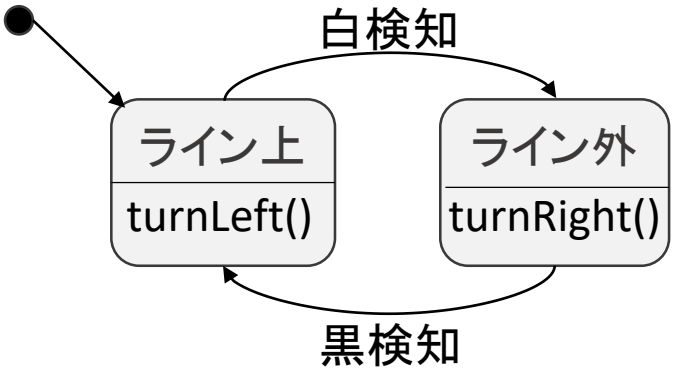
Controller class

```
void main(void) {
        Tracer::do_trace();
}
```

Tracer class

```
void do_trace (void) {
        Driver::drive_motor();
}
```

- ステートマシン図から



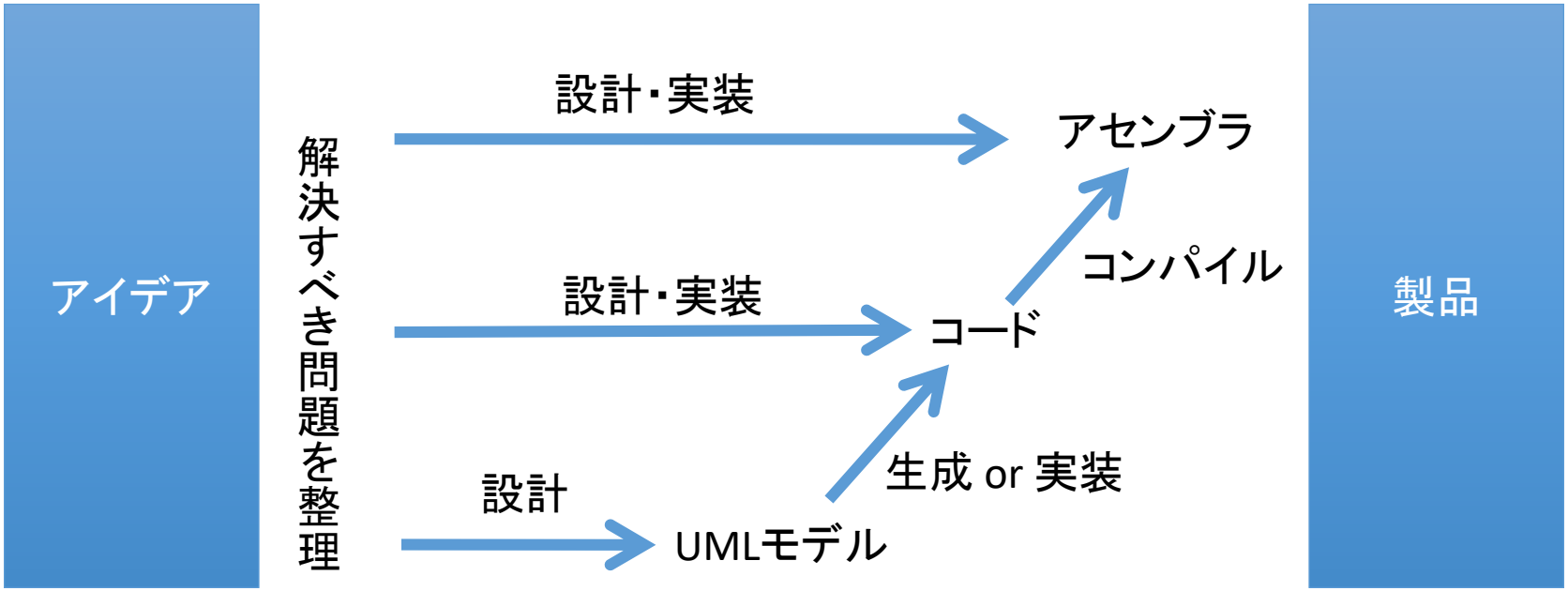

```
enum State{ OnLine, OutOfLine}
State state = OnLine;
void transition(Event e){
  switch(state){
     case OnLine:
        if(e == White){
          turnLeft();
          state = OutOfLine; ・・・
```

モデルからコードへの変換～コンピュータが実行

- 決まり切ったパターンがあるならば、コンピュータに実行させることができるのでは？
  - 人間が行う「実装のパターン」を「変換ルール」としてとらえる.
  - 「変換ルール」が十分に形式的 (= プログラムにできる)であれば、機械化できる.
  - 「変換ルール」が汎用的ならば、再利用できる.

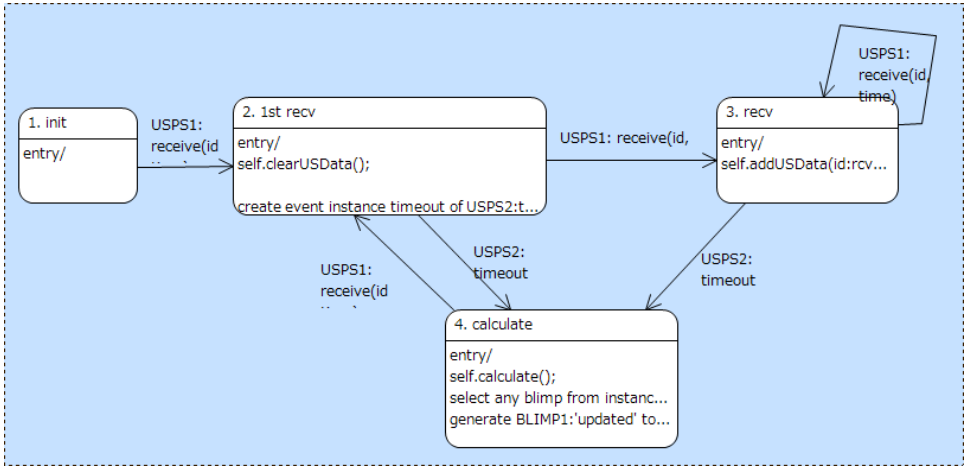
- どこかで聞いたことのあるような話のような・・・

モデルの抽象度と生産性



モデルを動作させてテスト

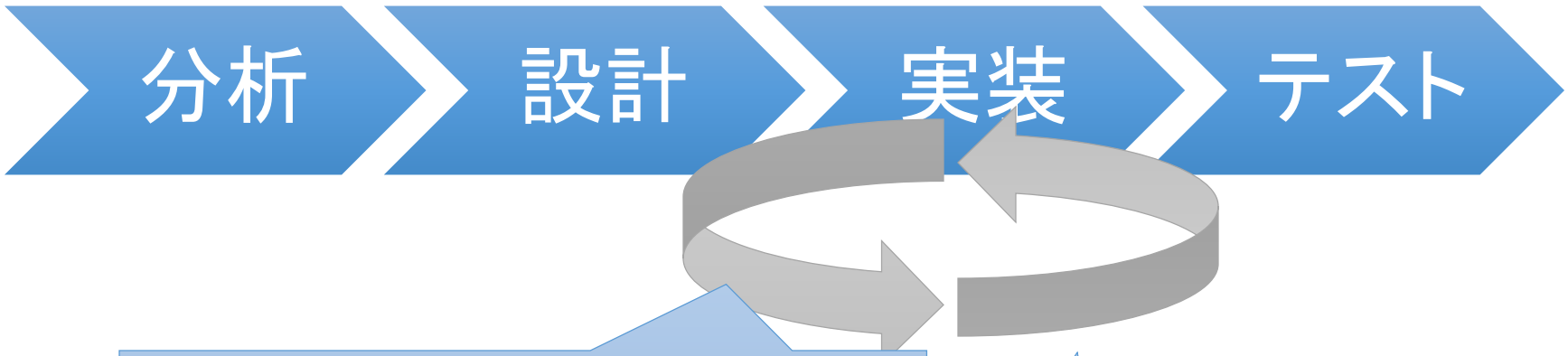
- 設計段階で動かすことはできないか？
  - 実装する前にモデル上で振る舞いをシミュレーションしてテストできれば、問題の早期発見につながる.
    - 実装を待たなくてもテストが可能.
- 形式的な(= コンピュータが解釈可能な)モデルを利用.



従来型開発とモデル駆動開発の比較

- 従来型開発 (aka ソースコード中心の開発)
  - 要求仕様書、設計文書をもとにソースコードを開発
  - 上流での要求仕様書や設計文書の正しさを担保するのはレビュー
  - 実際の製品の品質保証をするのは「テスト」
    - 開発の半分以上がテスト・・・
  - 要求仕様書・設計文書とソースコードの一貫性を保つのは困難
- モデル駆動開発
  - 要求レベル、設計レベルのモデルを作成
  - モデルから（ある程度は）自動的にソースコードを生成
  - 上流での検証が比較的容易
    - 機械可読なモデルがあるのでシミュレーションが可能
  - 自動的にソースコードを生成するので、モデルとコードの一貫性保持が容易

開発スタイルの変化: モデル駆動開発以前



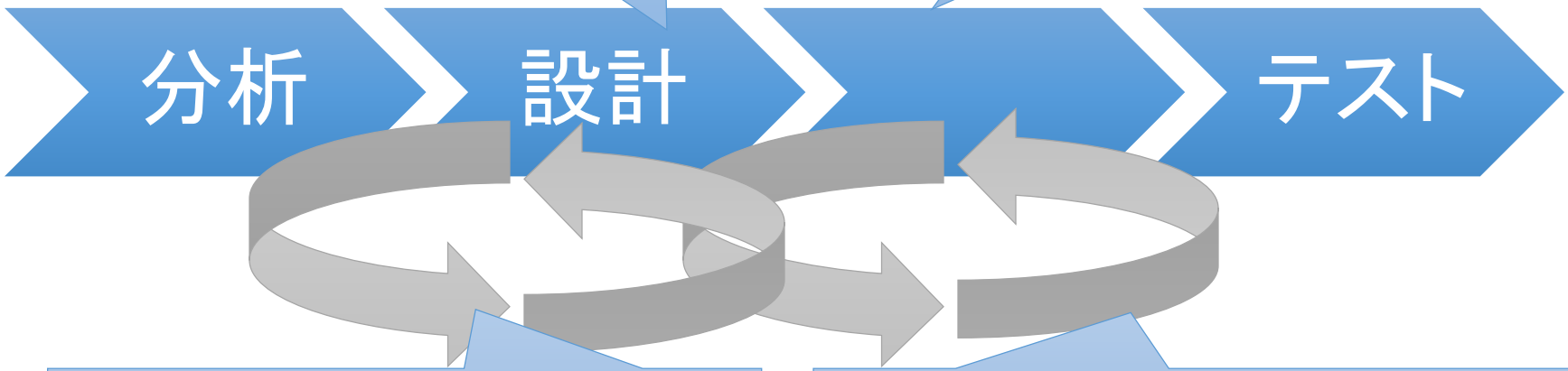
不具合を見つける度に、設計、実装、テストを「手動で」回さなければならない！

製品ドメインの知識
実装ドメインの知識
開発者は両者ともに必要

開発スタイルの変化: モデル駆動開発以前以降

設計時は製品ドメインの知識を

モデルコンパイラが
設計から自動コード生成



モデル上で動作させて
問題の早期発見ができる

実装ドメインの知識は
「ルール」として与える

関心の分離ができる！

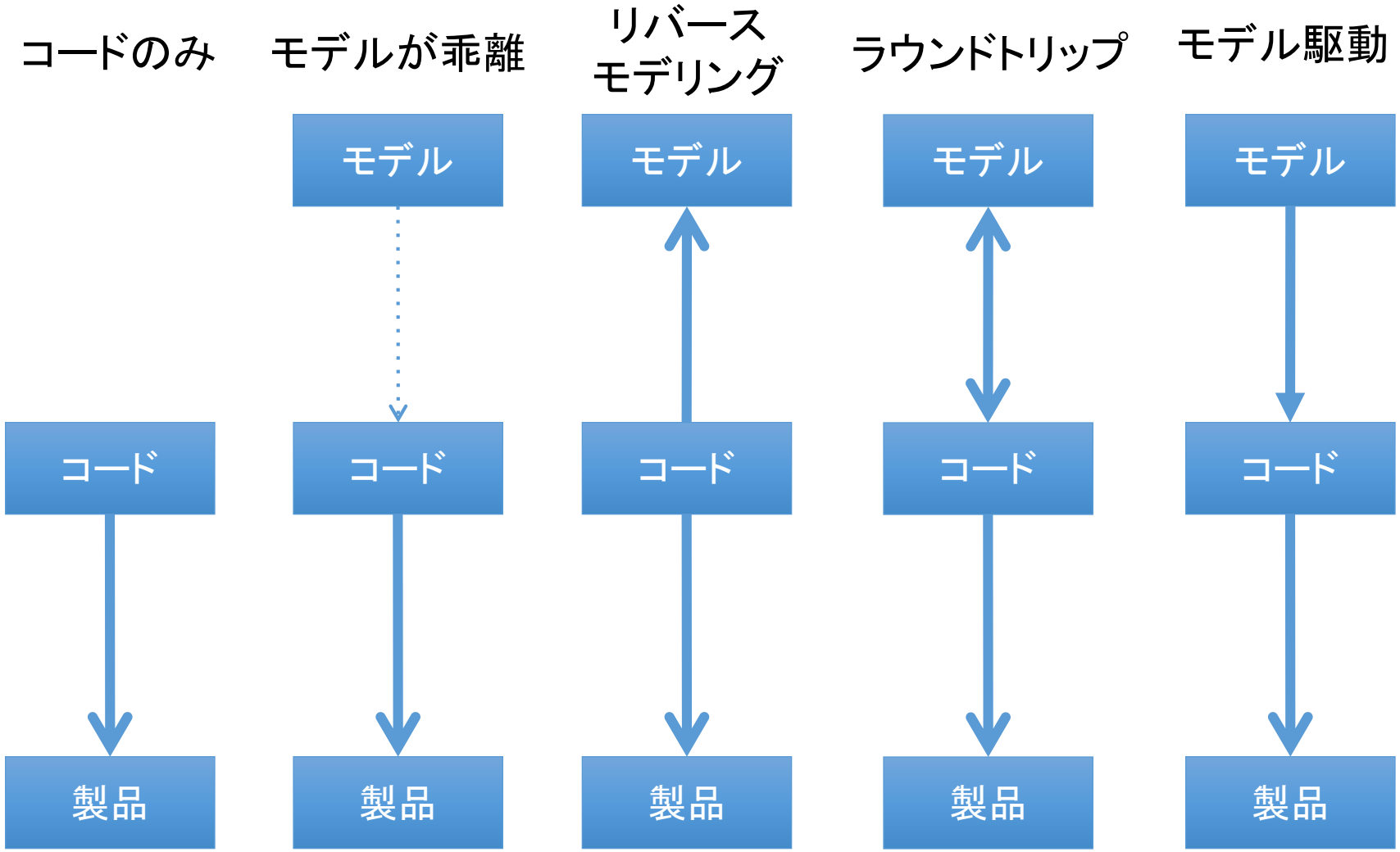
モデルの様々な利用方法

- スケッチとしてのUML
  - コミュニケーションの道具
- 設計図としてのUML
  - 設計 → スケルトン生成
  - コードの可視化
- プログラミング言語としてのUML
  - コード生成
  - シミュレーションによる検証

モデルからコードへの変換～様々なレベル

- コード生成（というかモデリング）には構造と振る舞いの両面が必要.
- クラス図などからスケルトンコードの生成.
  - 詳細な振る舞いはソースコード中に手で記述.
  - →簡単に出来そう．一貫性保持が困難.
- クラス図にコード断片を埋込みコード生成.
  - 振る舞いはコード断片の形でクラス図に埋込み.
  - 完全なコード生成が可能.
  - →抽象度が上がっていない.
- クラス図とステートマシン図からコード生成.
  - 構造はクラス図、振る舞いはステートマシン図で記述.
  - より詳細な振る舞いはコード断片としてステートマシン図に埋込み.
  - 完全なコード生成が可能.

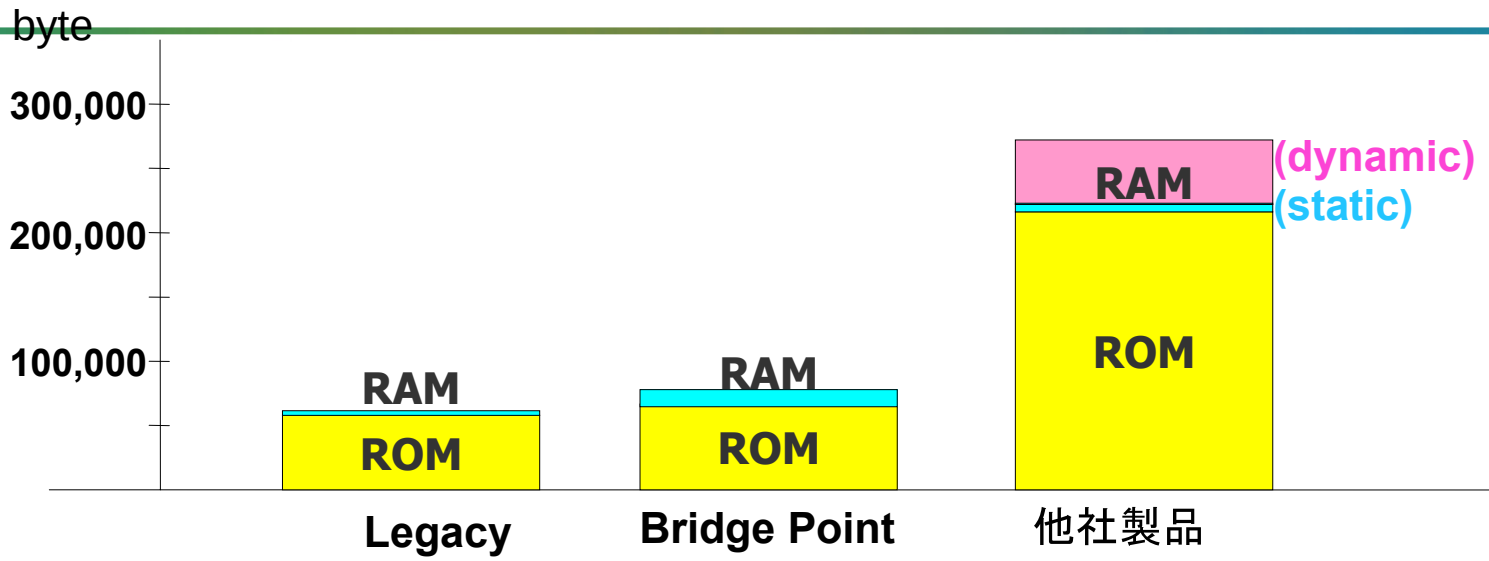
様々なモデルの利用



FAQ

- 性能が大幅に低下するんじゃないの？
  - ほとんど性能低下はありません.
  - Cでの記述とほぼ同等の性能が得られます.
  - パレートの法則.
- メモリを無駄に消費するんじゃないの？
  - 確かに増えますが大規模ではほとんど問題になりません.
    - たいてい同様のコードを手でコーディングしてます.
  - 大規模開発では手でのコーディングよりもサイズダウンという事例もあるようです.
  - 小規模開発にはちょっときついかもしれません.
- デバッグはできるの？コンパイラのバグに対応出来るの？
  - 手でのコード開発とほぼ同等の可読性が得られます.
  - MDDしている8割の会社はモデルコンパイラに手を入れてません.

Memory サイズの比較

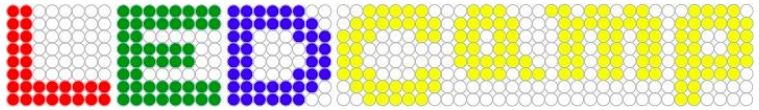


	Legacy	Bridge Point	他社製品
ROM	57,232byte	62,071byte	214,289byte
RAM(static)	1,176byte	12,676byte	1,934byte
RAM(dynamic)			approx. 45,000byte

MDDに利用できるツール

- Clooca
- xtUML(旧BridgePoint)
  - Executable UML、組込み向けの高品質コード生成
  - クラス図、ステートマシン図、アクション言語による完全なるMDD
- Enterprise Architect
  - Professonal版:コード埋込みクラス図
  - Ultimate版: コード埋込みクラス図/ステートチャート図
  - 体験版あり
- Rhapsody
  - コード埋込みクラス図/ステートチャート図

- astah*
  - コード生成プラグインを利用

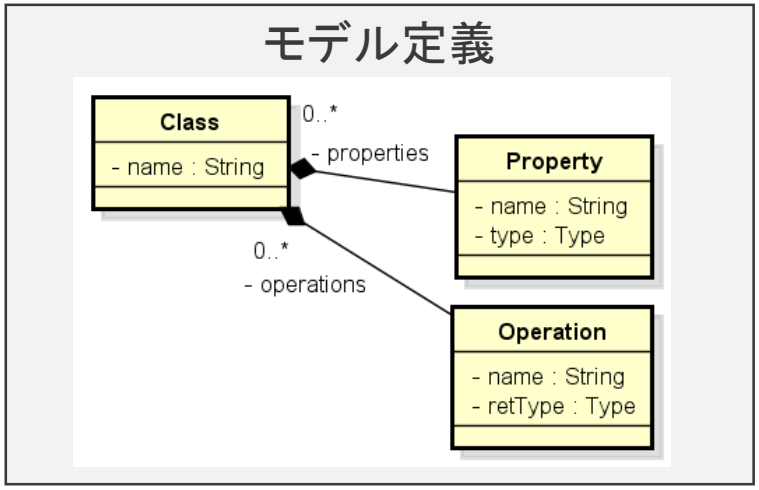


コード生成の仕組み

astah* m2t

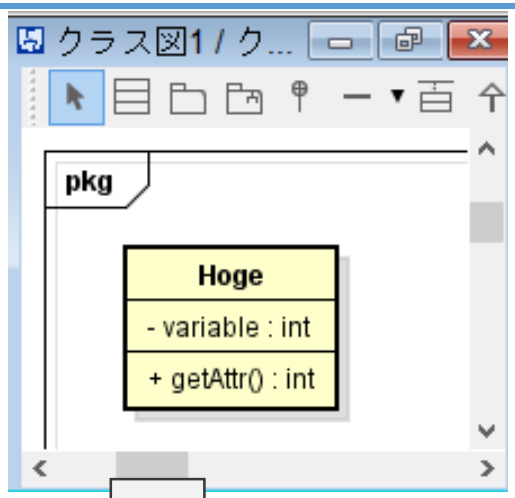
https://github.com/s-hosoai/astahm2t

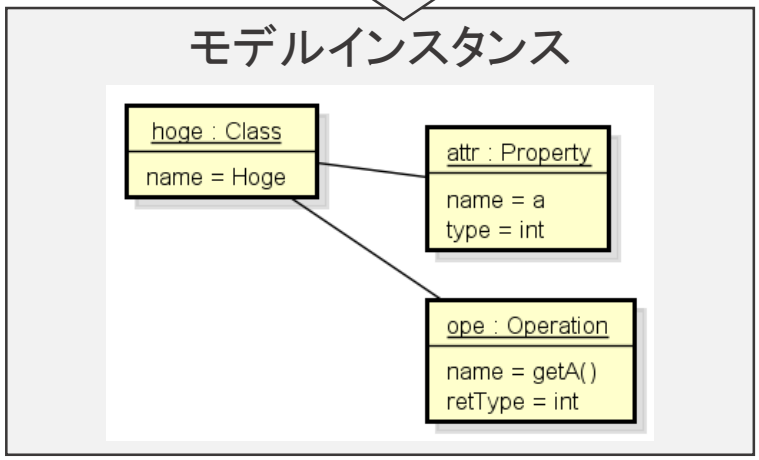
astah*のモデルの扱い



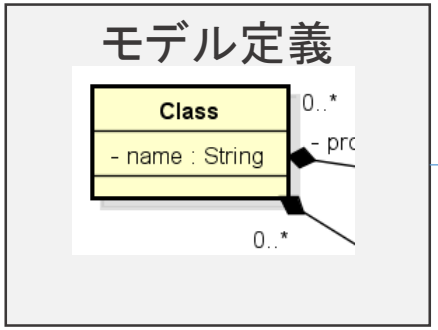
astah*は内部にUMLのモデル定義を持っている．例えば上記はクラス図のモデル定義の一部
（実際のクラス名や構造は異なる）

例えば右のようなクラス図を描くと内部では，下図のようなインスタンスを保持している





モデルからコードへの変換



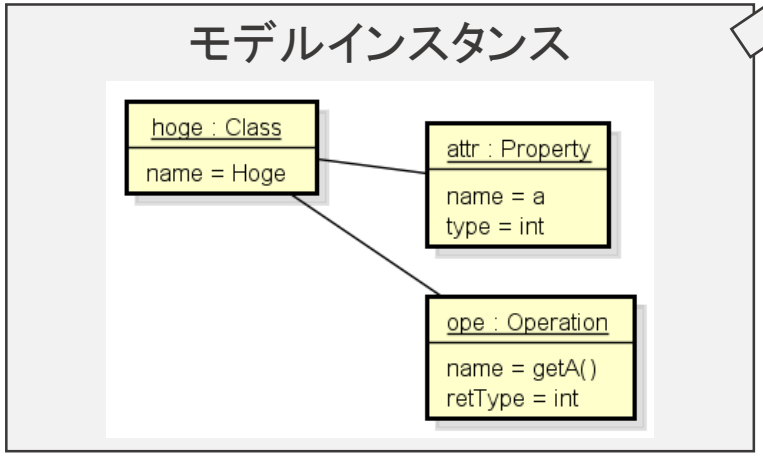
モデル定義を用いた
テンプレートを作成

コードテンプレート(C++用)

```
class ${Class.name} {

}
```

${}の部分は，インスタンスの値が展開される



インスタンス
をテンプレート
に適用し，
コードを生成

生成コード

```
class Hoge {

}
```

コード生成の仕組み