モデル駆動開発とドメイン特化言語

2013年10月16日

九州大学
大学院システム情報科学研究院
鵜林 尚靖

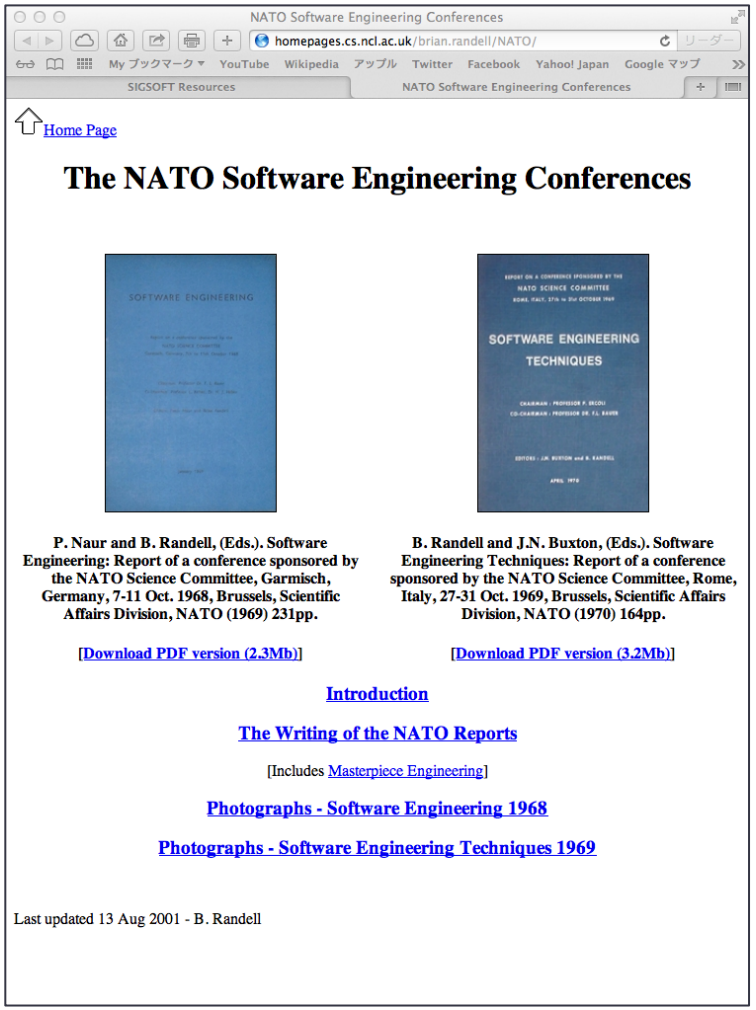
九州大学
KYUSHU UNIVERSITY

内容

- 第一部　ソフトウェア工学とモデル駆動開発（MDD）
- 第二部　MDD概論
  - MDDとは
  - コード生成の仕組み
  - ドメイン特化開発とDSL
- 第三部　研究事例　（私たちの研究紹介）
  - 事例1：組込みソフトウェア開発のための外部環境分析
  - 事例2：Yet Another MDD ～コード生成を仮定しないMDD～
- 第四部　今後の動向

第一部　ソフトウェア工学とモデル駆動開発（MDD）

アイスブレーク: NATO会議覚えていますか？



http://homepages.cs.ncl.ac.uk/brian.randell/NATO/

報告書の目次

4 5

CONTENTS

HIGHLIGHTS . . . 3
CONTENTS . . . 5
PREFACE . . . 9
1. BACKGROUND OF CONFERENCE . . . 13
2. SOFTWARE ENGINEERING AND SOCIETY . . . 19
3.1. THE NATURE OF SOFTWARE ENGINEERING . . . 19
3.2. SOFTWARE ENGINEERING MANAGEMENT AND METHODOLOGY . . . 24
3.3. DESIGN AND PRODUCTION IN SOFTWARE ENGINEERING . . . 32
4. DESIGN . . . 35
4.1. INTRODUCTION . . . 35
4.1.1. Sources of techniques . . . 35
4.1.2. Need for hardware based on program structure . . . 35
4.1.3. Relation to mathematics . . . 37
4.2. DESIGN CRITERIA . . . 38
4.2.1. Genera design criteria . . . 38
4.2.2. User requirements . . . 40
4.2.3. Reliability and design . . . 44
4.2.4. Logical completeness . . . 44
4.3. DESIGN STRATEGIES AND TECHNIQUES . . . 45
4.3.1. Sequencing the design process . . . 45
4.3.2. Structuring the design . . . 50
4.3.3. Feedback through monitoring and simulation . . . 53
4.3.4. High-level languages . . . 55
4.4. COMMUNICATION AND MANAGEMENT IN DESIGN . . . 59
5. PRODUCTION . . . 65
5.1. INTRODUCTION . . . 65
5.1.1. The problems of scale . . . 65
5.1.2. The problems of reliability . . . 70
5.2 PRODUCTION — MANAGEMENT ASPECTS . . . 72
5.2.1. Production planning . . . 72
5.2.2. Personnel factors . . . 83
5.2.3. Production control . . . 86
5.2.4. Internal communication . . . 89
6
5.3. PRODUCTION — TECHNICAL ASPECTS . . . 93
5.3.1. Tools . . . 93
5.3.2. Concepts . . . 96
5.3.3. Performance monitoring . . . 99
6. SERVICE . . . 103
6.1. INTRODUCTION . . . 103
6.1.1. The virtue of realistic goals . . . 103
6.1.2. Initial system release . . . 103
6.1.3. Frequency of releases . . . 104
6.1.4. Responsibility for modified systems . . . 106
6.2. REPLICATION, DISTRIBUTION AND MAINTENANCE . . . 107
6.2.1. Replication . . . 107

6.2.2. Distribution . . . 109
6.2.3. Maintenance . . . 110
6.3. SYSTEM EVALUATION . . . 112
6.3.1. Acceptance testing . . . 113
6.3.2. Performance monitoring . . . 114
6.4. FEEDBACK TO MANUFACTURERS FROM USERS . . . 115
6.5. DOCUMENTATION . . . 116
6.6. REPROGRAMMING . . . 117
7. SPECIAL TOPICS . . . 119
7.1. SOFTWARE: THE STATE OF THE ART . . . 119
7.1.1. Introduction . . . 119
7.1.2. Problem areas . . . 120
7.1.3. The underlying causes . . . 122
7.1.4. Possible solutions . . . 124
7.1.5. Summary . . . 125
7.2. EDUCATION . . . 125
7.3. SOFTWARE PRICING . . . 129
7.3.1. Introduction . . . 129
7.3.2. Arguments in favour of separate pricing . . . 150
7.3.3. Arguments against separate pricing . . . 131
7.3.4. Splitting software and hardware production . . . 132
7.3.4.1. The argument in favour . . . 132
7.3.4.2. The argument against . . . 133
7
8. INVITED ADDRESSES . . . 135
8.1. KEYNOTE SPEECH by A.J. Perlis . . . 135
8.2. 'MASS PRODUCED' SOFTWARE COMPONENTS, by M.D. McIlroy . . . 138
8.2.1. Discussion . . . 151
9. WORKING PAPERS . . . 157
R.W. Bemer, A.G. Fraser, A.E. Glennie, A. Opler, and H. Wiehle: *Classification of subject matter* . . . 160
R.W. Bemer: *Checklist for planning software system production* . . . 165
E.W. Dijkstra: *Complexity controlled by hierarchical ordering of function and variability* . . . 181
S. Gill: *Thoughts on the sequence of writing software* . . . 186
A.I. Llewelyn and R.F. Wickens: *The testing of computer software* . . . 189
T.B. Pinkerton: *Performance monitoring and systems evaluation* . . . 200
B. Randell: *Towards a methodology of computing system design* . . . 204
F. Selig: *Documentation standards* . . . 209
APPENDICES . . . 213
A 1 Participants . . . 213
A 2 Conference Schedule . . . 218
A 3 Original list of topics that were to be discussed . . . 219
A 4 Addresses of welcome . . . 224
CONTRIBUTOR INDEX . . . 226
SUBJECT INDEX . . . 228

色々と議論されていた. . .

設計に関する議論

4.3. DESIGN STRATEGIES AND TECHNIQUES

4.3.1. SEQUENCING THE DESIGN PROCESS

The problem of the proper order in which to do things during design is currently a subject for research in software engineering. This was reflected in the extensive discussions during the conference.

Naur: In the design of automobiles, the knowledge that you can design the motor more or less independently of the wheels is an important insight, an important part of an automobile designer's trade. In our field, if there are a few specific things to be produced, such as compilers, assemblers, monitors, and a few more, then it would be very important to decide what are their parts and what is the proper sequence of deciding on their parts. That is really the essential thing, what should you decide first. The approach suggested by Christopher Alexander in his book: *Notes on the Synthesis of Form*, is to make a tree structu
together those decisions that hang most closely together, and
final design. Then you move up one step and combine them
kind, as to which design decisions are related to one anothe
consider this a very promising approach.

David: (from *Some thoughts about the production of large softwar*

»Begin with skeletal coding: Rather than aiming at finished
at exploring interfaces, sizes of critical modules, complexity
items should be checked out, preferably on the hardware if it
The contributions of this step should be insight and experien

Kolence: (from *On the interaction between software design techni*

»Consider how the understanding of the types of structures
affect the manager's decision on how to begin the developn
specifications. The most important concepts which must be

4.3.4. HIGH-LEVEL LANGUAGES

The use of high-level languages in writing software systems was the subject of a debate.

d'Agapeyeff: (from *Reducing the cost of software*)

»In aiming at too many objectives the higher-level languages have, perhaps, proved to be useless to the layman, too complex for the novice and too restricted for the expert.« I maintain that high-level programming languages have, to this extent, failed.

Fraser: Software is generally written in a low-level language. Has anyone written low-level software in a high-level language? Would you do it again?

David: (from *Some thoughts about the production of large software systems (2)*)

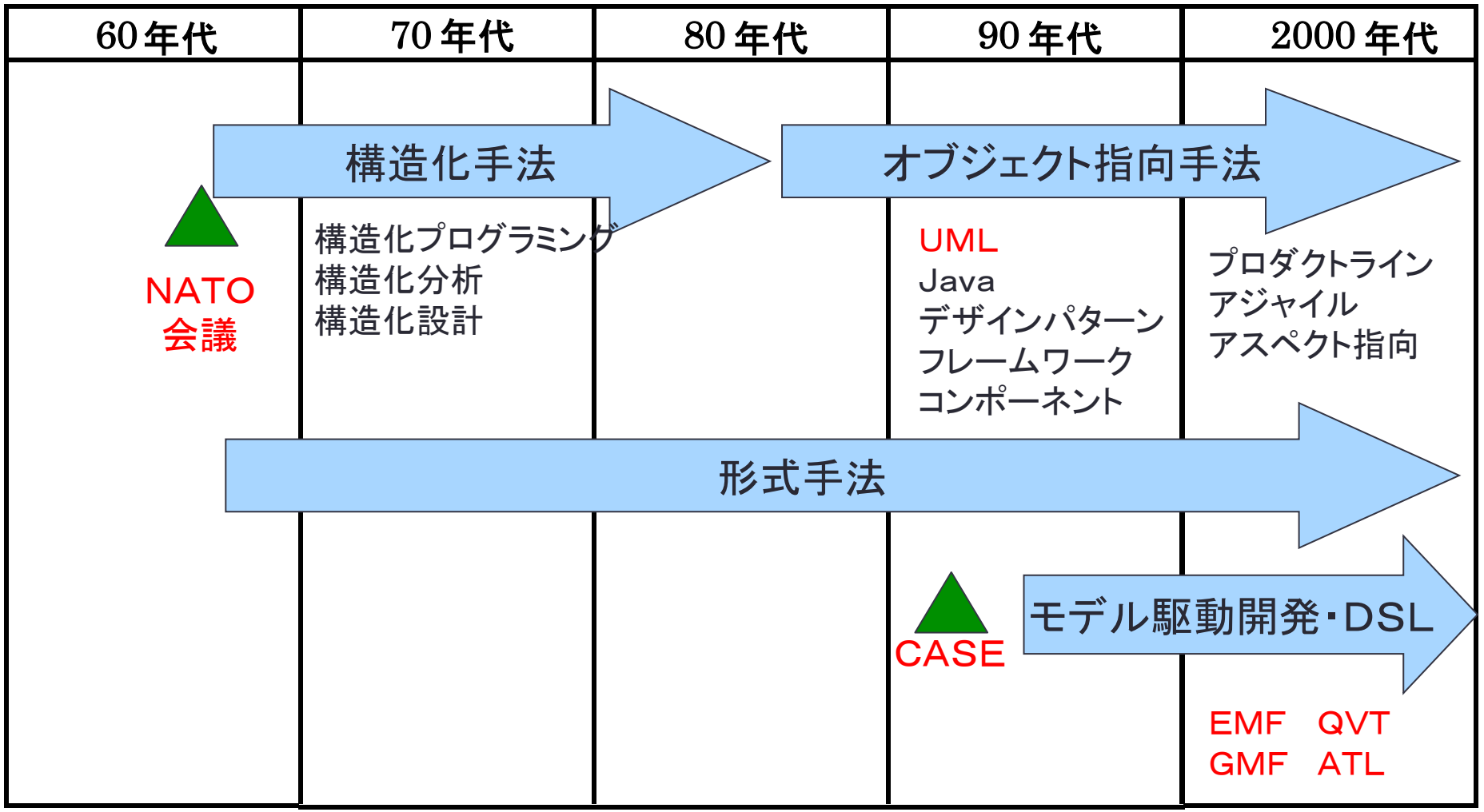
»Few large systems h
penalties in size and
In my opinion, this vi
these penalties. Secon
to a high level with t
to performance can b
mance indicated earlier may mean that it is difficult to obtain better size-performance software using machine code written by an army of programmers of lesser average calibre.

DSLの議論に置き換えたらどうであろうか？

高級言語に関する議論

The advantages of coding in a high-level language lie in increased programmer productivity, fewer bugs in the code, fewer programmers required, increased flexibility in the product, 'readability' of the source code ('self-documentation') and some degree (unspecified) of machine independence (better called 'portability'). Many of these advantages have a face validity, which is fortunate since they are difficult to support with hard evidence. There is evidence, however, on the flexibility issue drawn from the Multics experience. Multics is coded almost entirely in a subset of PL/I known as EPL. The early versions of Multics were large and slow. Major improve-

ソフトウェア開発技術の変遷



モデリングに関する主な会議

モデリング専門

- MODELS
  - International Conference on Model Driven Engineering Languages and Systems
- MiSE
  - Modelling in Software Engineering @ ICSE
- ECMFA
  - European Conference on Modeling Foundations and Applications
- DSM
  - Workshop on Domain-Specific Modeling @ SPLASH
- MODELSWARD
  - International Conference on Model-Driven Engineering and Software Development
- UML&FM
  - International workshop UML and Formal Methods

ソフトウェア工学・言語

- ICSE
  - International Conference on Software Engineering
- FSE
  - ACM SIGSOFT Symposium on the Foundations of Software Engineering
- ASE
  - International Conference on Automated Software Engineering
- GPCE
  - International Conference on Generative Programming: Concepts & Experiences
- SLE
  - International Conference on Software Language Engineering
- FOSD
  - International Workshop on Feature-Oriented Software Development

現在の研究テーマは

研究

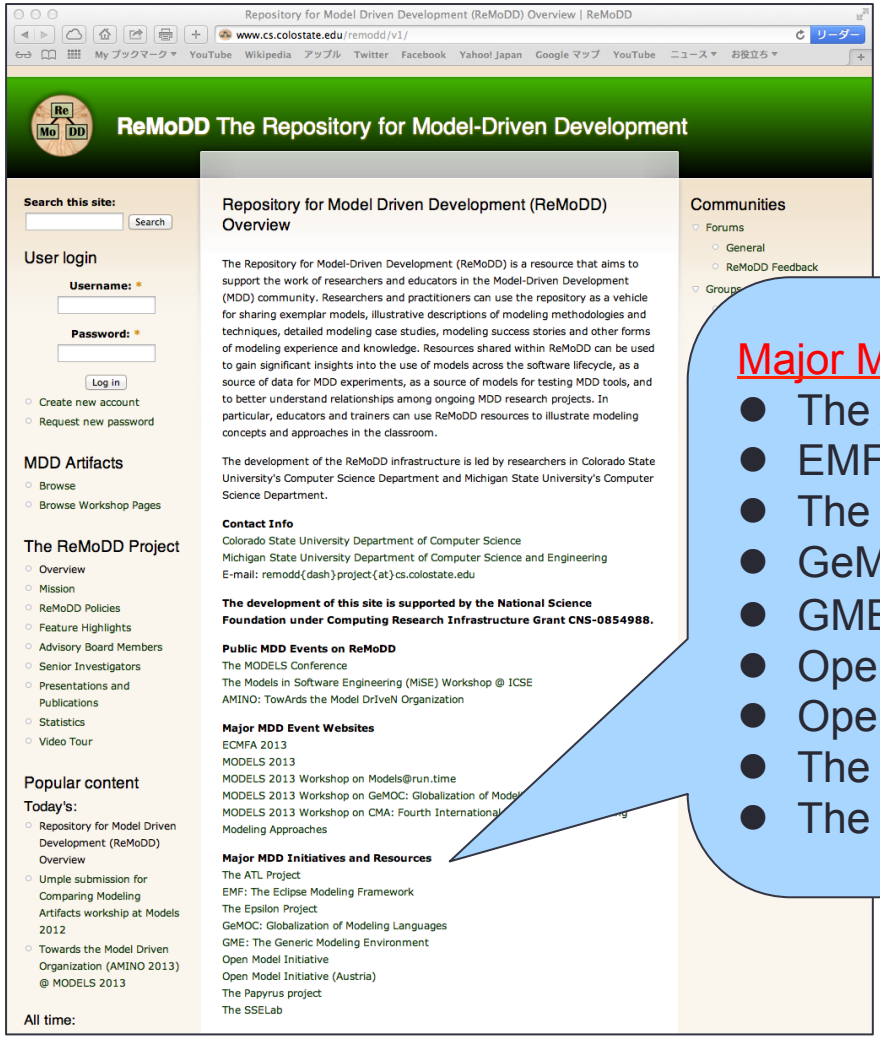
- Model-driven development and its proven effectiveness
- Development and use of domain-specific languages
- Evolution of general-purpose modeling languages and related standards
- Definition of the syntax and semantics of modeling languages
- Tools and meta-tools for modeling languages and model-driven development
- Model composition and metamodel composition
- Definition and usage of model transformations and generative approaches
- Support for legacy issues and evolution of models
- Proposals for new model quality assurance techniques: analysis, testing, model checking, validation and verification
- Model-driven development and cyber-physical systems
- New methodologies/frameworks/processes for model-driven development
- Development of systems engineering and modeling-in-the-large concepts
- Integration of modeling languages and tools: hybrid multi-modeling approaches
- New modeling paradigms and formalisms
- Modeling in, and for, the Cloud

応用

- Introducing model-driven approaches into organizations
- Application, technology and methodology case studies (successful and unsuccessful)
- Managing evolution of models
- Industrial scale model management (model size, user group size, viewpoint size etc.)
- Limitations, gaps and mismatches in current modeling standards
- Economic issues of model driven development
- Achieving industrial quality benchmarks with model driven development
- Industrial requirements for domain specific modeling
- Visions for industrial strength model driven engineering
- Empirical studies about model-based development

青色：活発な研究テーマ
赤色：新しいテーマ

MDDのためのリポジトリ ReMoDD



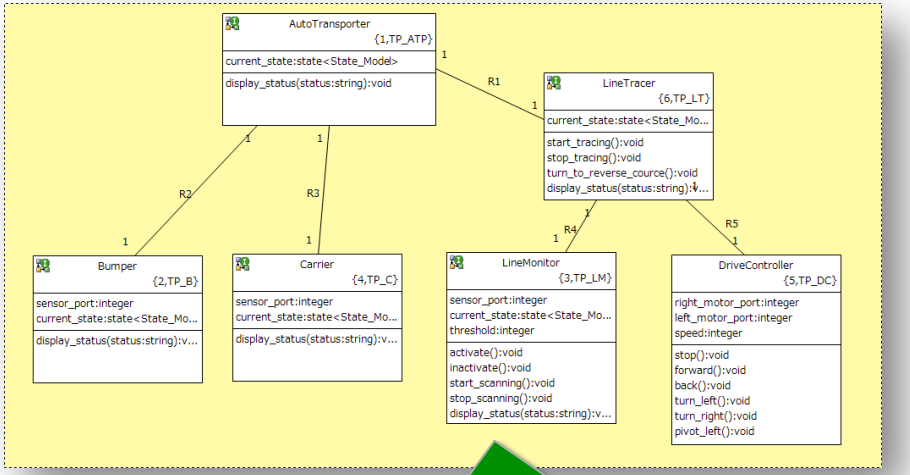
http://www.cs.colostate.edu/remodd/v1/

第二部　MDD概論

MDDとは

モデル駆動開発　MDD: Model Driven Development

設計モデル



モデルを動作させてテストできないの？

ソースコード

```
/*
 * Structural representation of application analysis class:
 *   AutoTransporter  (TP_ATP)
 */
struct Transporter_TP_ATP {
  Escher_StateNumber_t current_state;
  /* application analysis class attributes */

  /* relationship storage */
  Transporter_TP_LT * TP_LT_R1;
  /* Note:  No storage needed for TP_ATP->TP_B[R2] */
  /* Note:  No storage needed for TP_ATP->TP_C[R3] */
};

#define Transporter_TP_ATP_MAX_EXTENT_SIZE 10
extern Escher_Extent_t pG_Transporter_TP_ATP_extent;
extern void Transporter_TP_ATP_op_display_status( Transporte

extern void Transporter_TP_ATP_R1_Link( Transporter_TP_LT *,
/* Note:  TP_LT<-R1->TP_ATP unrelate accessor not needed */
extern void Transporter_TP_ATP_R2_Link( Transporter_TP_B *,
/* Note:  TP_B<-R2->TP_ATP unrelate accessor not needed */
extern void Transporter_TP_ATP_R3_Link( Transporter_TP_C *,
/* Note:  TP_C<-R3->TP_ATP unrelate accessor not needed */
```

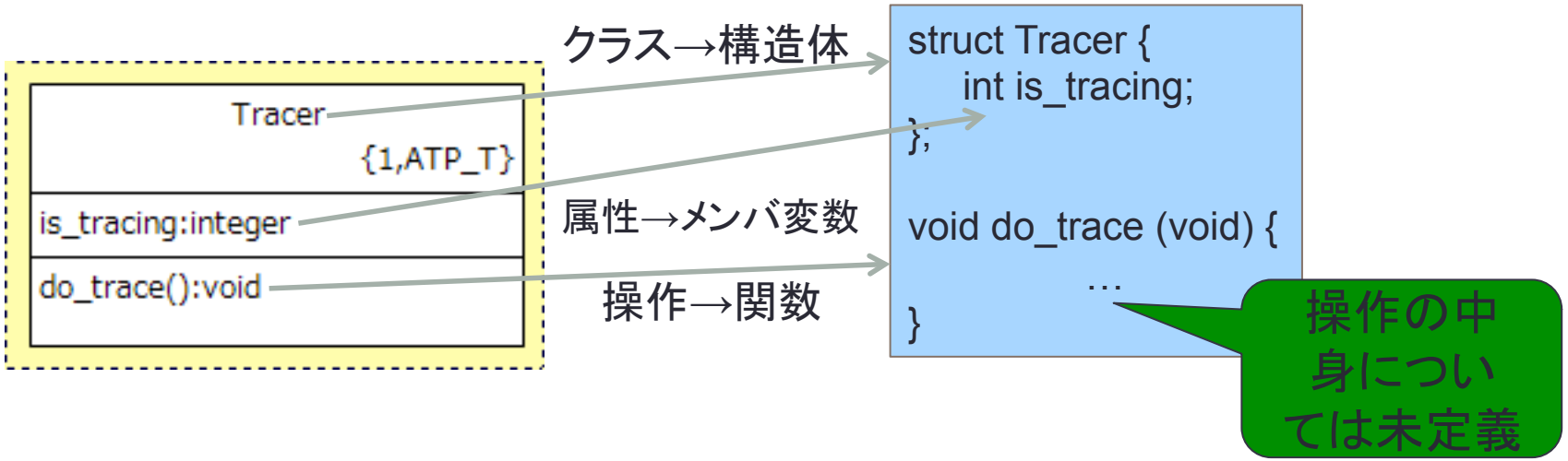
（半分でもいいから）自動化できないの？

モデルを使って開発を楽にする方法

モデルからコードへの変換～人間が実行

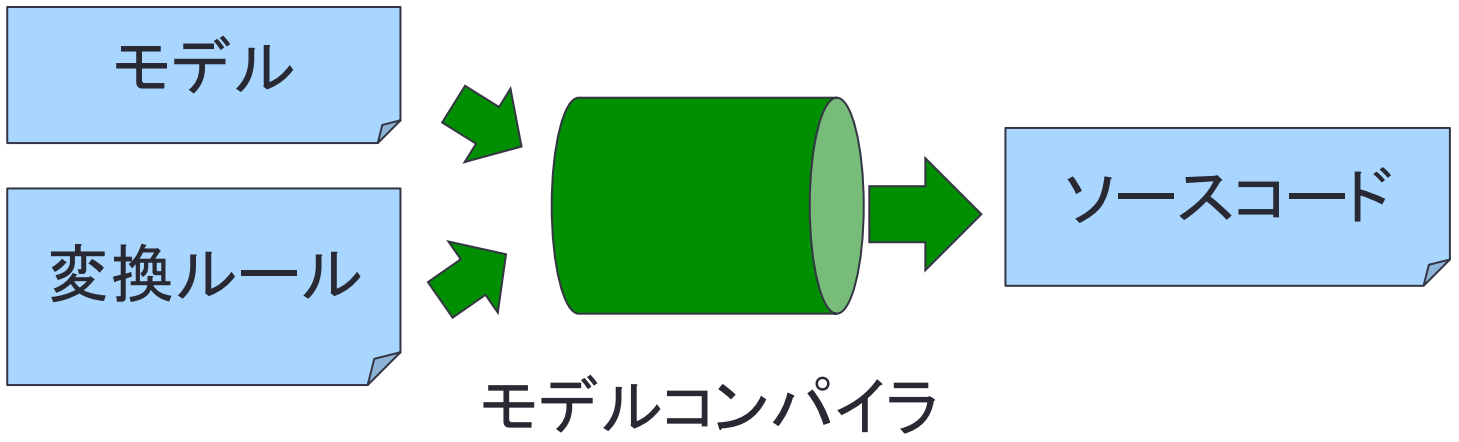
- クラスのソースコードへの変換を考えると・・・
- 決まりきったソースコードのパターンがある

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

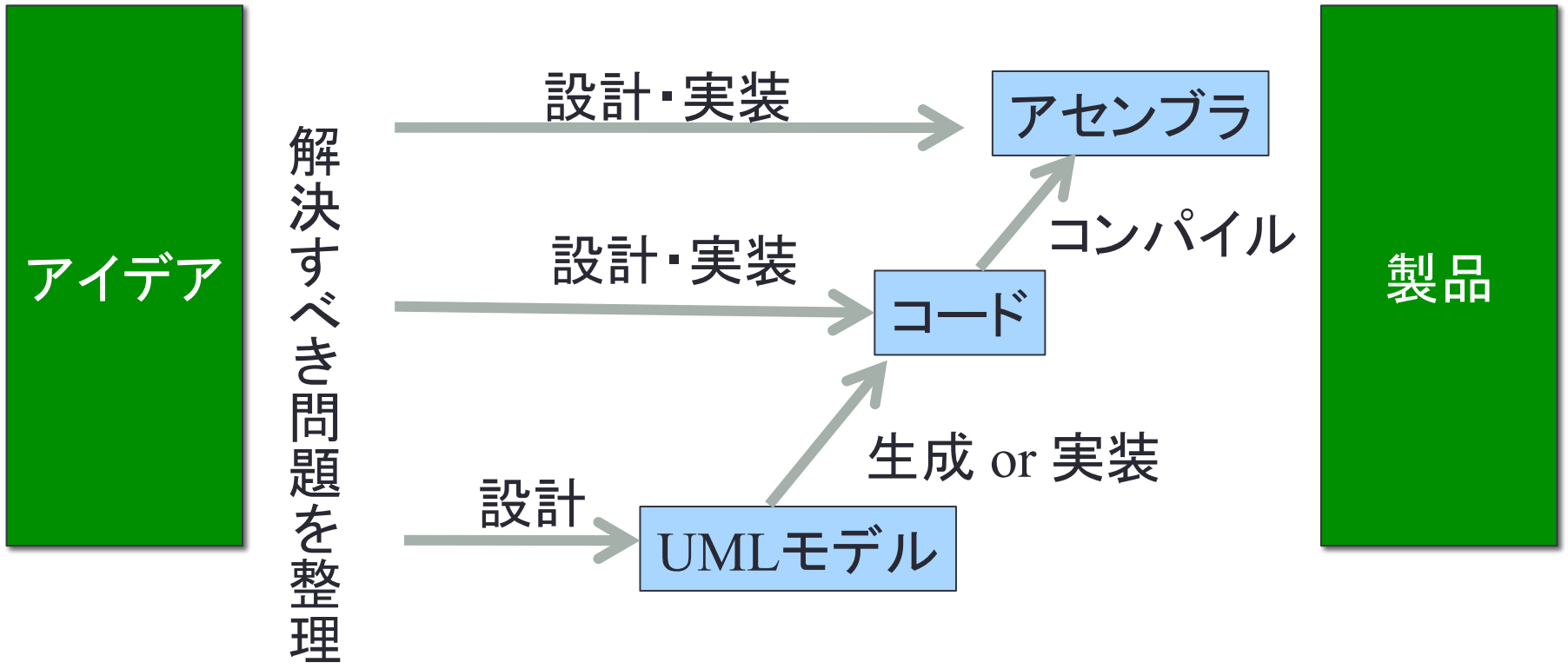


モデルからコードへの変換 ～コンピュータが実行

- 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



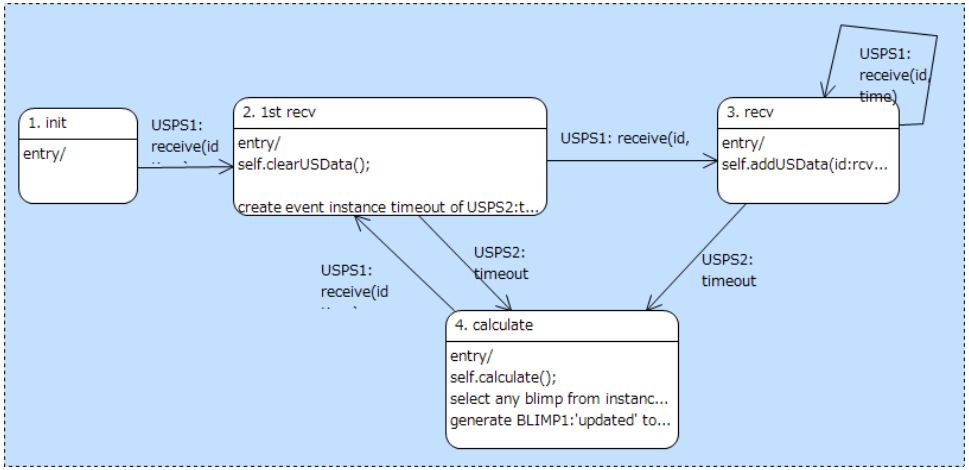
モデルの抽象度と生産性



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

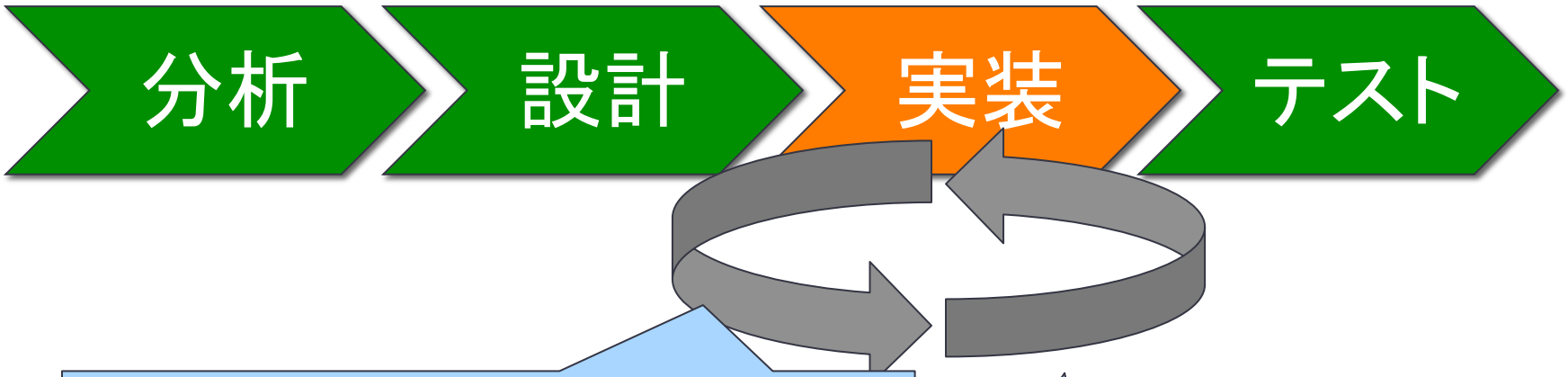
- 形式的な（= コンピュータが解釈可能な）モデルを利用



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

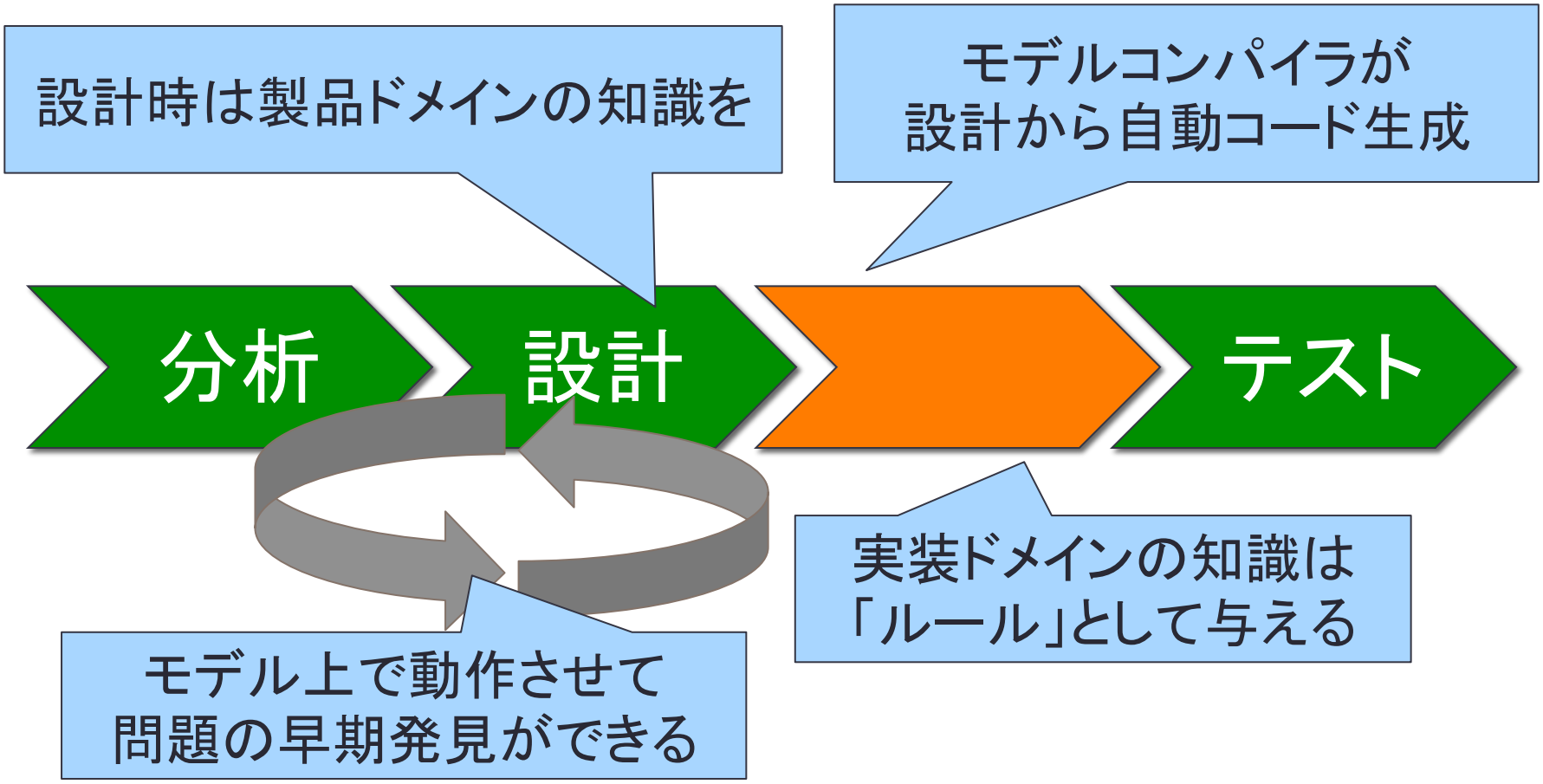
開発スタイルの変化: モデル駆動開発以前



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！

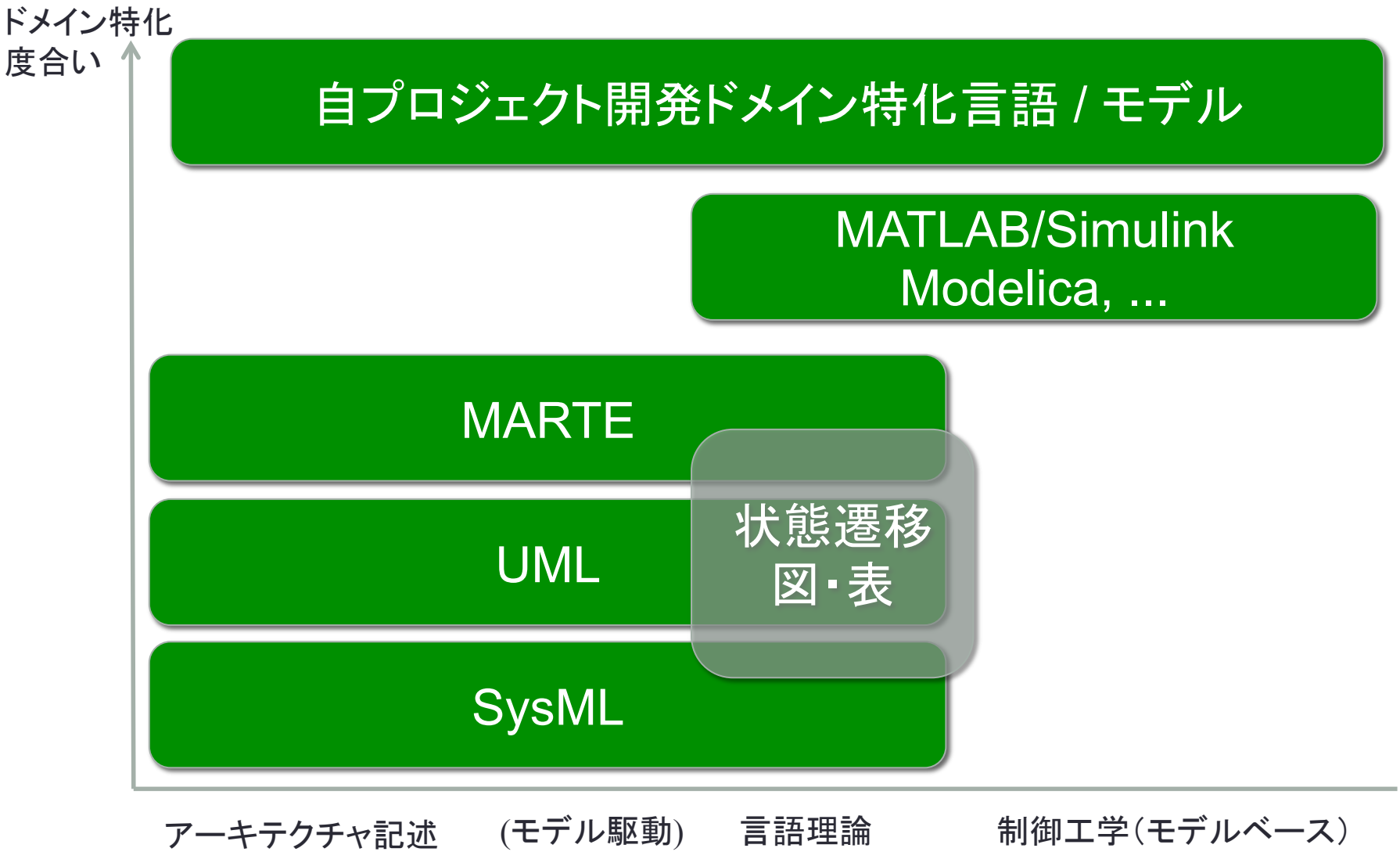
製品ドメインの知識
実装ドメインの知識
開発者は両者ともに必要

開発スタイルの変化: モデル駆動開発以降



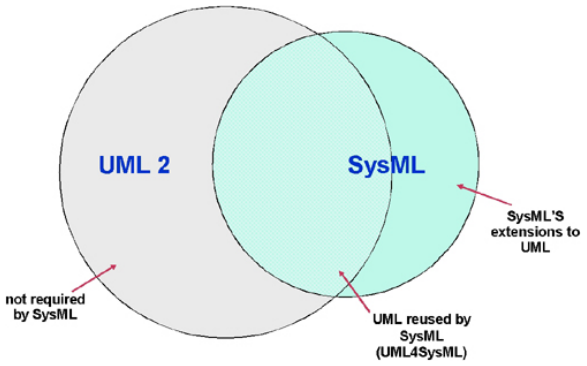
関心の分離ができる！

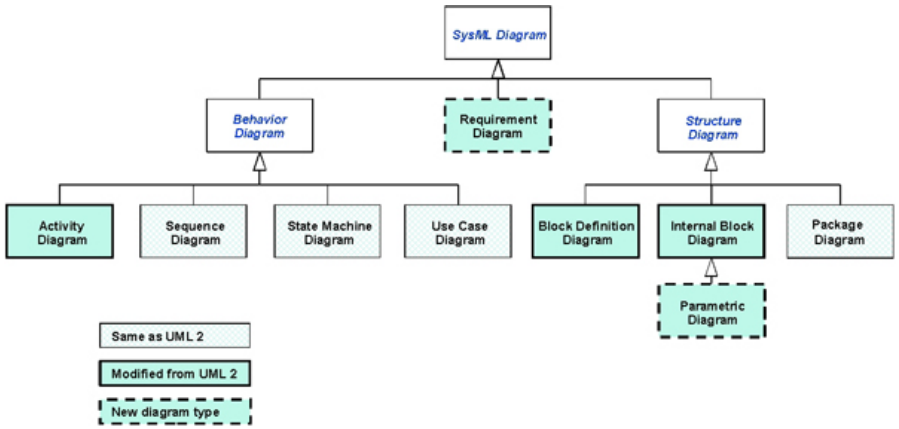
モデル地図


ドメイン特化
度合い
自プロジェクト開発ドメイン特化言語 / モデル
MATLAB/Simulink
Modelica, ...
MARTE
UML
SysML
状態遷移
図・表
アーキテクチャ記述
(モデル駆動)
言語理論
制御工学(モデルベース)

システムモデリング言語SysML

- SysML（Systems Modeling Language）は、システムをモデリングするための言語である。UMLはソフトウェアの分析、設計に用いられるが、組込みシステムのようにハードウェアとソフトウェアから構成される場合には若干力不足の側面がある。たとえば、ハードウェア制約などを記述するのが容易ではない。
- SysMLの言語仕様は、UMLの仕様を再利用した部分と新たに拡張した部分から構成される。

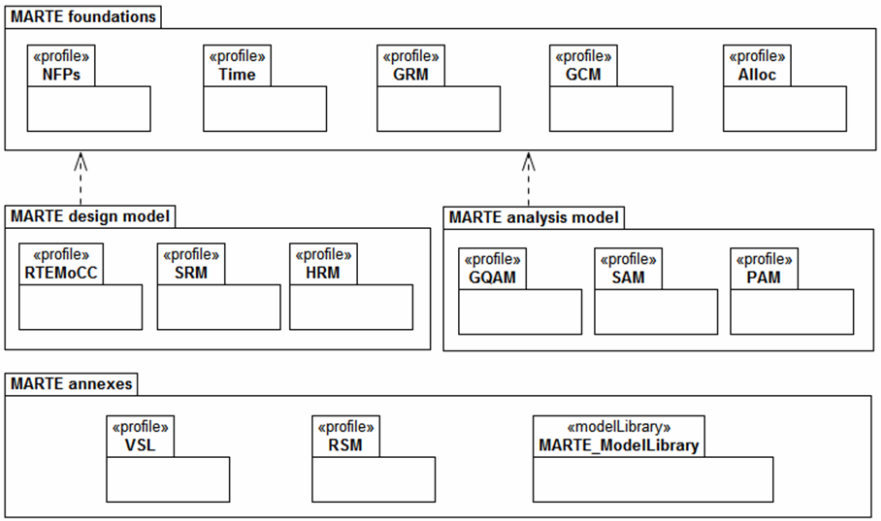




http://www.omgsysml.org

UML profile for MARTE

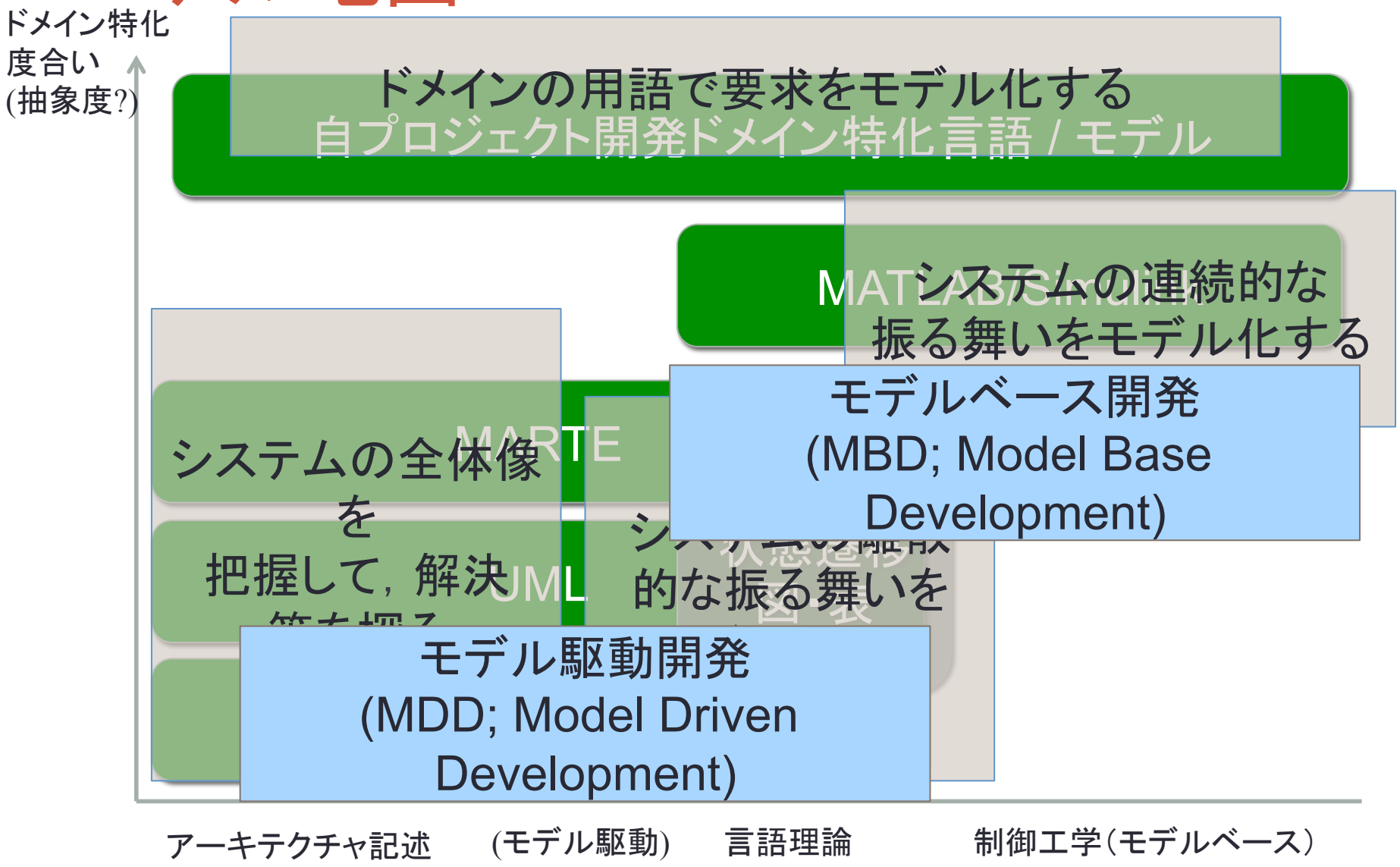
- このUMLプロファイル仕様は、モデル駆動のリアルタイム組み込みシステム（Real Time and Embedded Systems=RTES）開発に必要な機能をUMLに追加するためのものである。
- UML profile for MARTE（略称 MARTE）と呼ばれるこの拡張は、仕様記述、設計および確認・検証の各段階をサポートする。



NFPs = Non-Functional Properties (非機能属性)
GRM = Generic Resource Modeling (総称的リソースモデリング))
GCM = Generic Component Model (総称的コンポネントモデル)
Alloc = Allocation modeling (割当てモデリング)
RTEMoCC = RTE Model of Computation & Communication (RTE計算通信モデル)
SRM = Software Resource Modeling (ソフトウェア・リソースモデリング)
HRM = Hardware Resource Modeling (ハードウェア・リソースモデリング)
GQAM = Generic Quantitative Analysis Modeling (総称的定量分析モデリング)
SAM = Schedulability Analysis Modeling (スケジューラビリティ分析モデリング)
PAM = Performance Analysis Modeling, (パフォーマンス分析モデリング)
VSL = Value Specification Language (値定義言語)
RSM = Repetitive Structure Modeling (反復的構造モデリング)

http://www.object-report.jp/omginfo/technology/omg-marte/ より転載

モデル地図

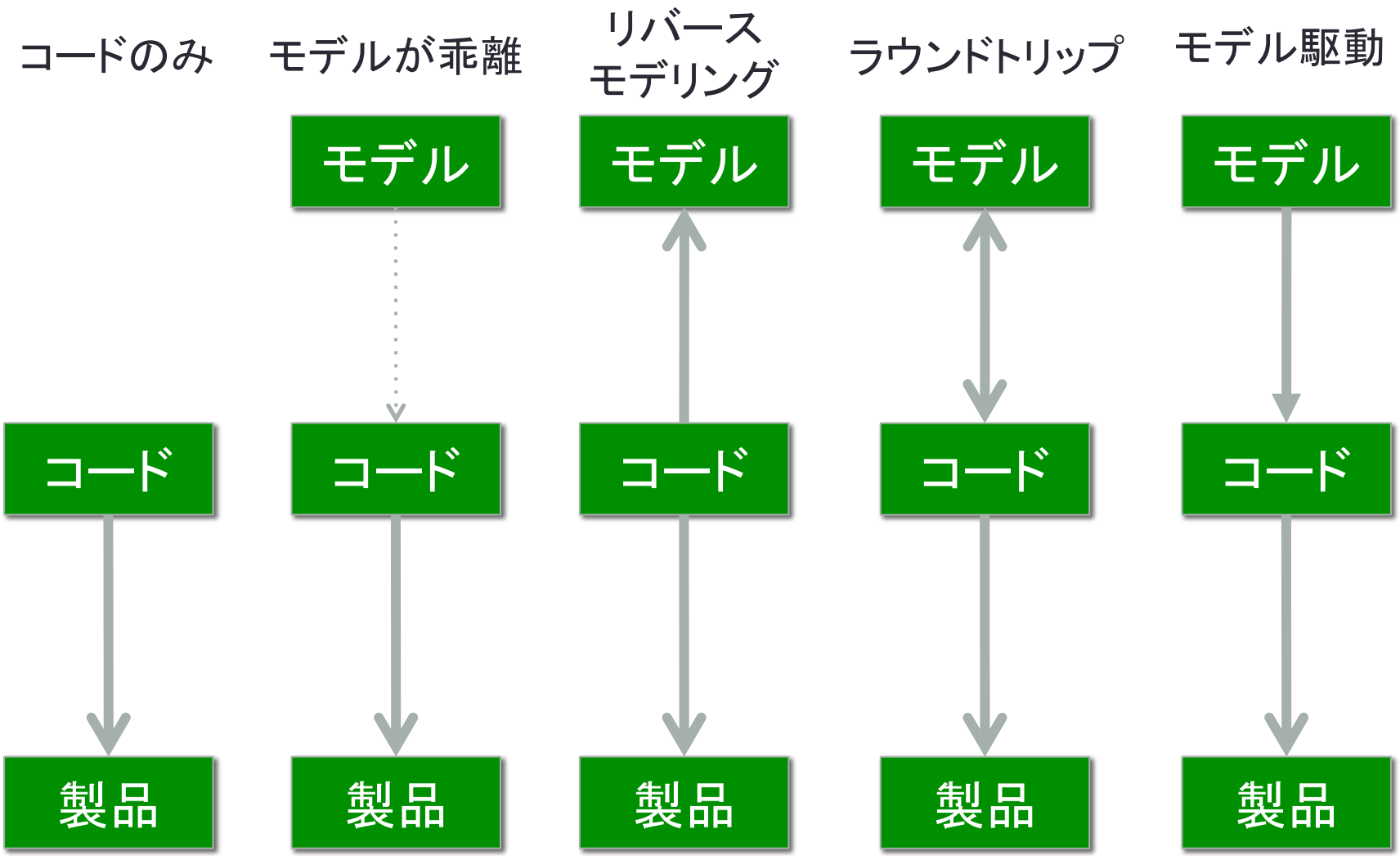

ドメイン特化
度合い
(抽象度?)
ドメインの用語で要求をモデル化する
自プロジェクト開発ドメイン特化言語 / モデル
システムの連続的な
振る舞いをモデル化する
モデルベース開発
(MBD; Model Base
Development)
システムの全体像
を
把握して，解決
MARTE
UML
モデル駆動開発
(MDD; Model Driven
Development)
アーキテクチャ記述
(モデル駆動)
言語理論
制御工学(モデルベース)

モデルの様々な利用方法

Martin Fowlerは以下の3つに分類

- スケッチとしてのUML（UML as sketch）
  - コミュニケーションの道具
- 設計図としてのUML（UML as blueprint）
  - 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（UML as programming language）
  - コード生成
  - シミュレーションによる検証

様々なモデルの利用


コードのみ
モデルが乖離
リバース
モデリング
ラウンドトリップ
モデル駆動
モデル
モデル
モデル
モデル
コード
コード
コード
コード
コード
製品
製品
製品
製品
製品

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。

いつか来た道. . .

NATO会議での高級言語に関する議論

4.3.4. HIGH-LEVEL LANGUAGES

The use of high-level languages in writing software systems was the subject of a debate.

d'Agapeyeff: (from *Reducing the cost of software*)

»In aiming at too many objectives the higher-level languages have, perhaps, proved to be useless to the layman, too complex for the novice and too restricted for the expert.« I maintain that high-level programming languages have, to this extent, failed.

Fraser: Software is generally written in a low-level language. Has anyone written low-level software in a high-level language? Would you do it again?

David: (from *Some thoughts about the production of large software systems* (2))

»Few large systems have been written in high-level languages. This is not surprising since there are inevitable penalties in size and performance of compiled code, and these factors have been paramount in people's minds. In my opinion, this view is no longer appropriate in many instances since techniques are available to overcome these penalties. Secondary memories can take the squeeze out of the size issue, and performance can be brought to a high level with the aid of a traffic analysis of flow of control in the system. Thus, those modules crucial to performance can become 56 the subject of special attention. Indeed, the vast range of programmer performance indicated earlier may mean that it is difficult to obtain better size-performance software using machine code written by an army of programmers of lesser average calibre.

The advantages of coding in a high-level language lie in increased programmer productivity, fewer bugs in the code, fewer programmers required, increased flexibility in the product, 'readability' of the source code ('self-documentation') and some degree (unspecified) of machine independence (better called 'portability'). Many of these advantages have a face validity, which is fortunate since they are difficult to support with hard evidence. There is evidence, however, on the flexibility issue drawn from the Multics experience. Multics is coded almost entirely in a subset of PL/I known as EPL. The early versions of Multics were large and slow. Major improve-

MDDに利用できるツール

- Clooca
  - モデリング言語のオンラインプラットフォーム
- BridgePoint
  - Executable UML、組込み向けの高品質コード生成
  - クラス図、ステートチャート図、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

少しだけ Clooca の宣伝

九州大学大学院システム情報科学府情報知能工学専攻社会情報システム工学コース（QITO）の学生が起業し開発

http://www.clooca.com/

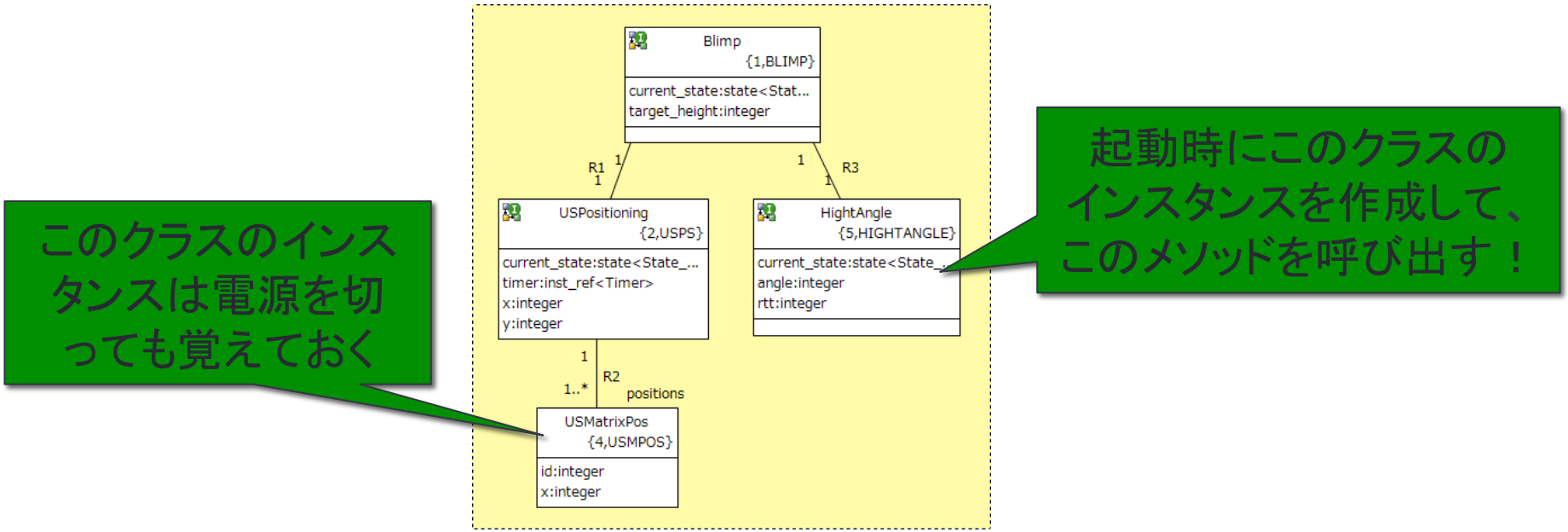
コード生成の仕組み

モデルからコードへの変換～様々なレベル

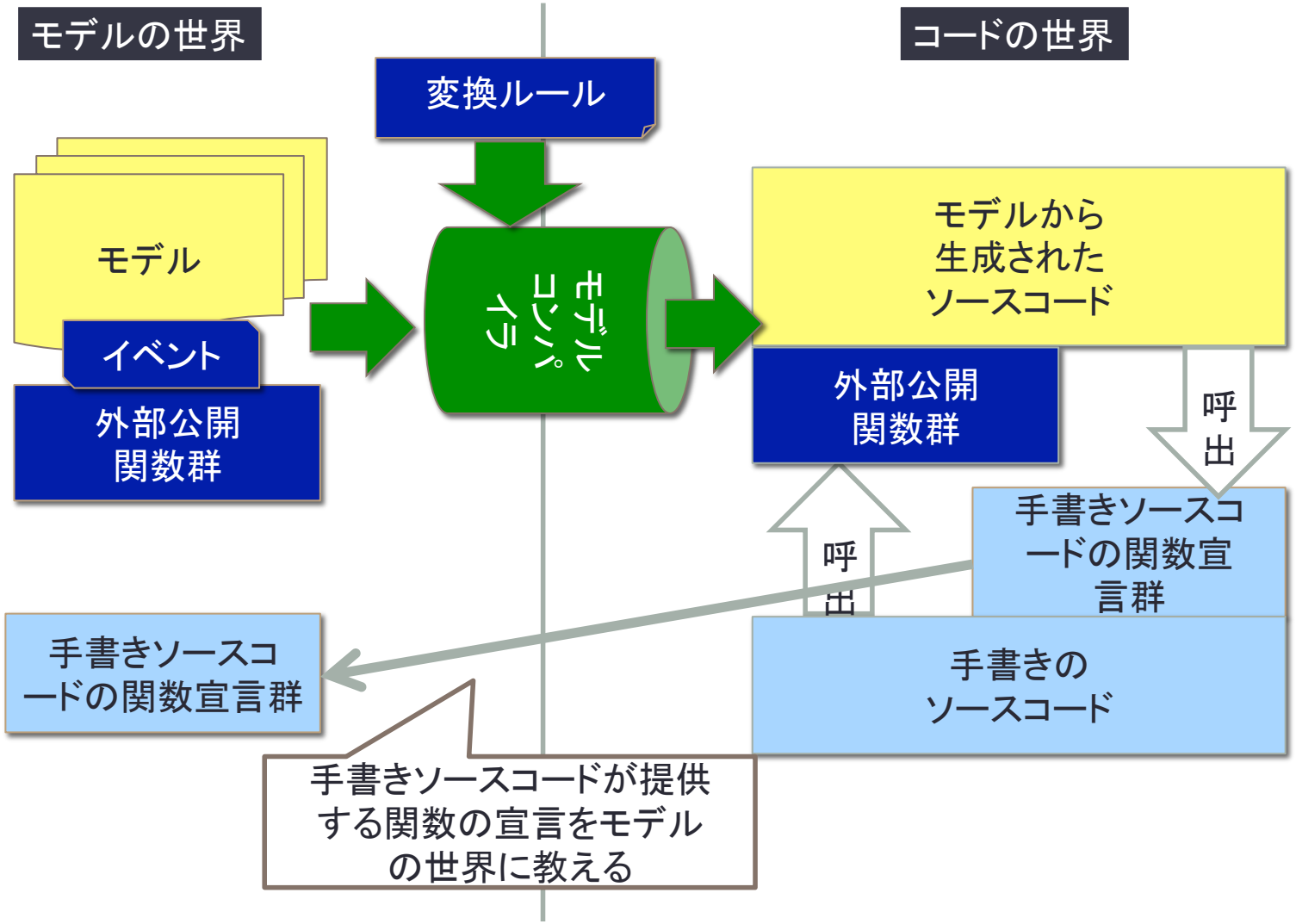
- コード生成（というかモデリング）には構造と振る舞いの両面が必要
- クラス図などからスケルトンコードの生成
  - 詳細な振る舞いはソースコード中に手で記述
  - →簡単に出来そう。一貫性保持が困難。
- クラス図にコード断片を埋込みコード生成
  - 振る舞いはコード断片の形でクラス図に埋込み
  - 完全なコード生成が可能
  - →抽象度が上がっていない。
- クラス図とステートマシン図からコード生成
  - 構造はクラス図、振る舞いはステートマシン図で記述
  - より詳細な振る舞いはコード断片としてステートマシン図に埋込み
  - 完全なコード生成が可能

変換ルールの与え方: マーク

- 指定したモデル要素のみに適用するルール
  - 「指定」のことを「マーク」と呼ぶ
- 特別扱いしたい要素にマークを付ける
  - 永続的/一時的オブジェクト、リモートオブジェクト
  - 起動時、モデル記述以外の世界とのインタフェース



モデルの世界 vs. コードの世界
モデルとコードの世界の関係



厳密なモデル定義とモデル変換定義

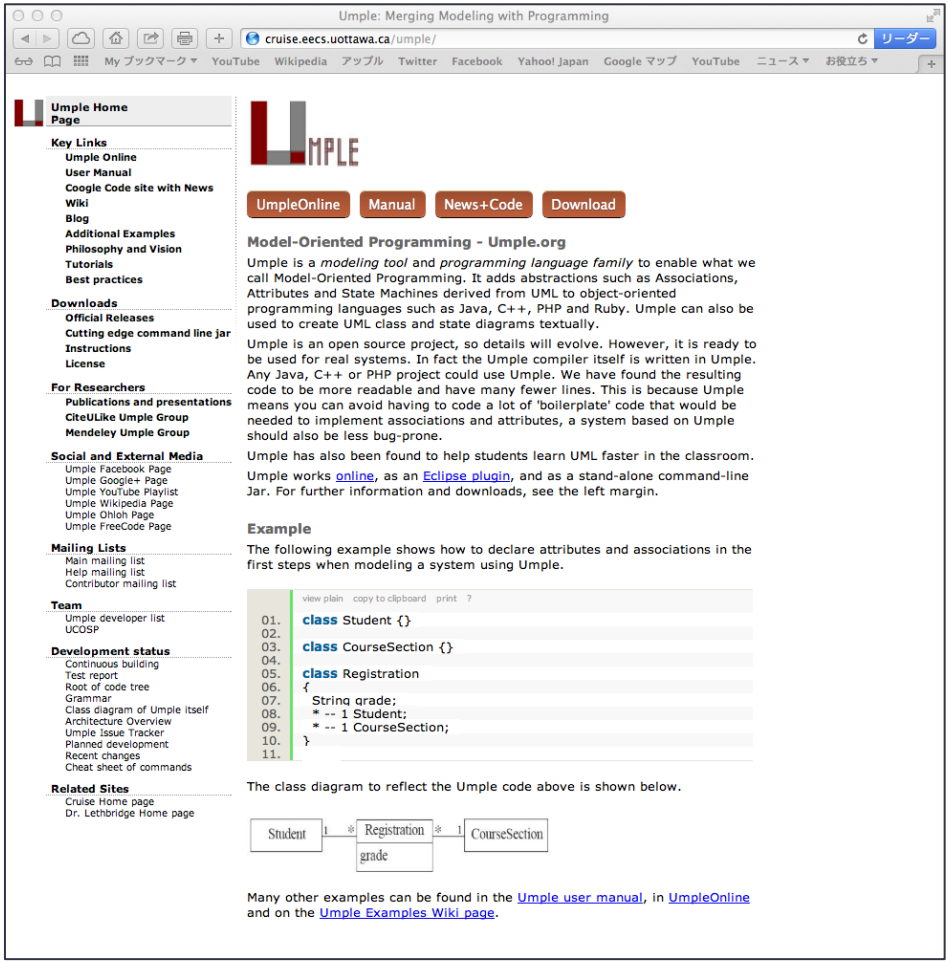
- 厳密なモデル表記 （MOF、OCL）
- 厳密なモデル変換定義 （QVT）

QVTの例

```
mapping Simple_Class_To_Java_Class
  refine Simple_Class_And_Java_Class {
  domain {(SM.Class)[name = n, attributes = A] }
  body {
    (JM.Class)[
      name = n,
      attributes = A->iterate(a as ={} |
        as +  Simple_Attribute_To_Java_Attribute(a))
    ]
  }
}
```

MOF: Meta Object Facility
OCL: Object Constraint Language
QVT: Queries, Views, and Transformations
ATL: ATLAS Transformation Language（QVTのRFPを受けてINRIAが開発）

Model-Oriented Programming



http://cruise.eecs.uottawa.ca/umple/

ArchJava を彷彿

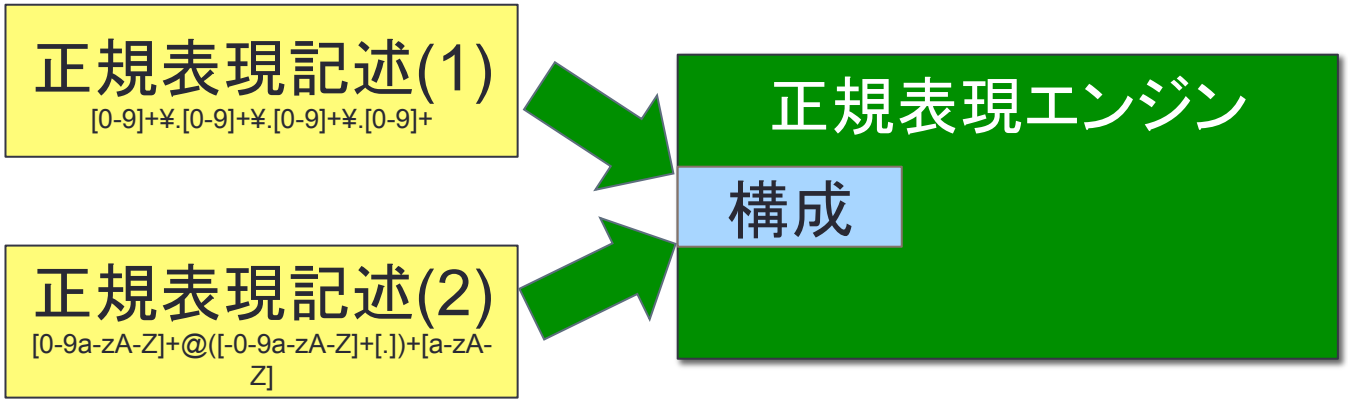
ドメイン特化型開発とDSL

ドメイン特化型開発とは(1)

特定目的向けの言語を設計することによってソフトウェア開発における多くの問題をより簡単に解決する

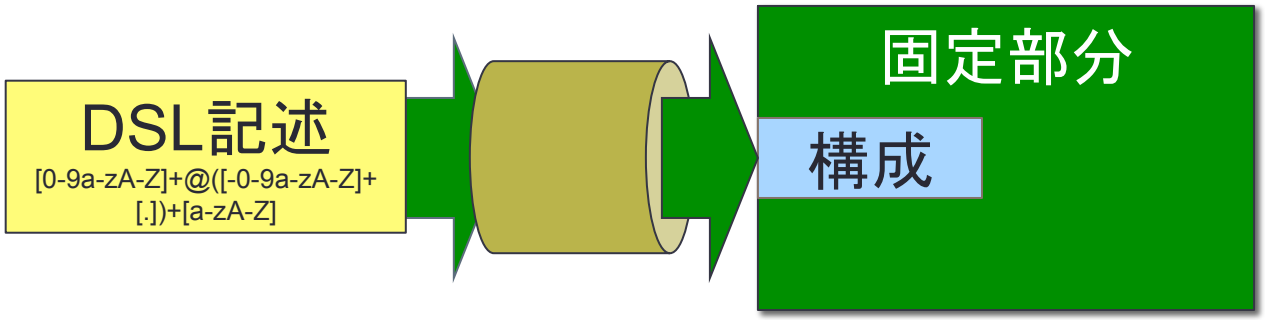
～Steve Cook他: ドメイン特化型開発[1]～

- 例: 正規表現と正規表現エンジン
  - テキストのパターンに対してある処理を行う場合
    - 正規表現がなければ毎回パーサを実装
      - 生産性低い、品質の問題も
    - 正規表現を定義、正規表現エンジンとして再利用
      - 「特定目的向けの言語」→「多くの問題を簡単に解決」



ドメイン特化型開発とは(2)

- プロダクトラインソフトウェア工学の実現手段のひとつ
  - 相違部
    - ドメイン特化型言語(Domain-Specific Language; DSL)で記述
    - DSL記述から構成情報などを生成
  - 共通部
    - 従来手法で開発
- モデル駆動開発のひとつ
  - 要求や設計をドメインに特化したモデル(DSM / DSML)で記述
  - 汎用のモデリング言語よりも記述が容易



DSLのカテゴリ

- 外部DSL
  - ホスト言語とは異なる言語で開発されたDSL
  - 例：Latex、SQL など
- 内部DSL（or 組込みDSL）
  - ホスト言語がもつ拡張機構を用いて構築したDSL
  - DSL部分以外はホスト言語の機能がそのまま使用できる
  - 拡張機構をもつ言語としてScalaなどがある

ドメイン特化型開発の利点

生産性

- モデルと思考のミスマッチが減少
- アプリケーションの自動生成

品質

- 設計と実装の一貫性保持
- ドメイン特化最適化
- 汎用言語での記述エラーを最小化

コミュニケーション

- 様々なステークホルダがモデルを利用しやすくなる

ドメイン特化型開発の欠点

コスト

- 多品種展開できなければ無意味

教育

- 独自の教育・教育コンテンツが必要
- 汎用言語よりは導入が容易だといわれている

ドメイン特化型開発のプロセス

ドメインエンジニアリング

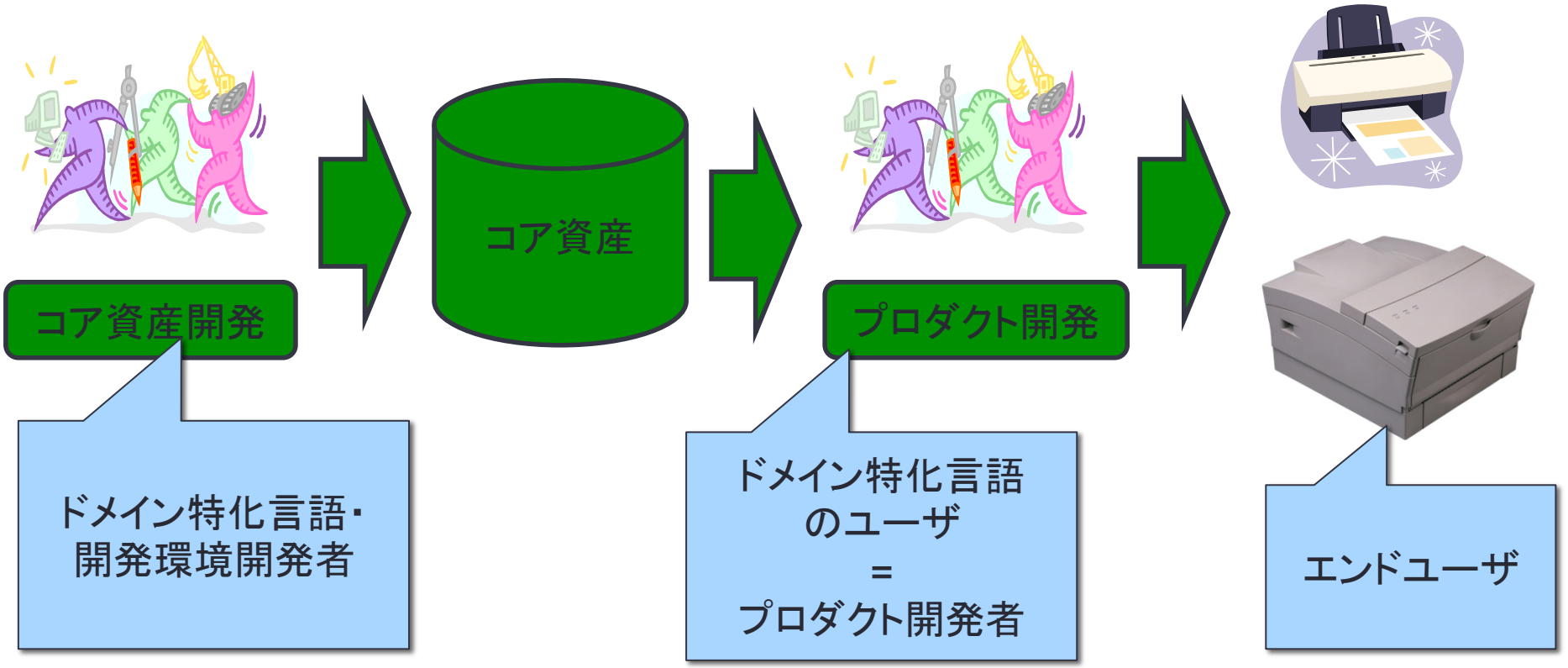
- チームビルディング
- スコーピング
- 可変性の特定
- ドメイン特化モデルの発見
- ドメイン固有記法の定義
- 共通アーキテクチャ設計と実装コンポーネントの確定
- コードジェネレータの開発
- ドメインテスト

アプリケーションエンジニアリング

- ドメイン特化言語によるアプリケーションの記述
- アプリケーションテスト

ドメイン特化型開発のロール

- エンドユーザ
- ドメイン特化言語のユーザ
- ドメイン特化言語・開発環境の開発者



DSL開発環境・ツール

- Microsoft DSL Tools
  - Visual Studio Professional Edition以上
    - VS上で開発できる独自言語を作り出すことが出来る。
  - モデルエディタ
    - メタモデル・図表現の開発支援
  - ジェネレータの開発支援
    - C#, VB + テンプレートによるコード生成
- EMF/GMF
  - Eclipse上で動作
  - DSL Toolsとほぼ同等
- MetaEdit+
  - モデルエディタ
    - メタモデル・図表現・表の開発支援
  - ジェネレータの開発支援
    - MEALスクリプト言語によるコード生成
- Enterprise Architect
- テキスト系
  - Xtext@Eclipse

モデル駆動開発の導入10個のtips

1. ドメインをしっかりと狭く保つ
   - モデル駆動開発を適用する
2. 十分に理解できているドメインに適用する
3. MDDをクリティカルパスに適用する
   - パイロットプロジェクトでは十分な注意と資源が割かれない・・・
4. MDDは最初から適用すると一番うまくいく
   - MDDに適したソフトウェアアーキテクチャがあり，後から変更することは難しい
5. ほかの場所で相殺される利益に注意するべし
6. コード生成ばかりに気を取られない
   - 上流でのシミュレーションによる品質向上，良いソフトウェアアーキテクチャの獲得など
7. 抽象的に考えられる人ばかりではない
8. 多くのプロジェクトは規模の拡大時に失敗する
9. 人やものの考え方にツールやプロセスを適合させる，逆ではなく
10. そう，たった9つだけだ

By 久住先生（九州大学）

第三部　研究事例

事例1：組込みソフトウェア開発のための外部環境分析

Naoyasu Ubayashi, Yasutaka Kamei, Masayuki Hirayama, and Tetsuo Tamai: A Context Analysis Method for Embedded Systems ---Exploring a Requirement Boundary between a System and Its Context, 19th IEEE International Requirements Engineering Conference (RE 2011), IEEE Computer Society, pp.143-152 (2011).

自動車 CC（クルーズコントロール）操作マニュアルからの抜粋

- 次のような状況のときはCCを使用しないでください。使用すると思わぬ事故につながるおそれがあります。
  - 交通量の多い道や急カーブのある道
  - 凍結路や積雪路などのすべりやすい路面
  - 急な下り坂

どうして問題になるのか？

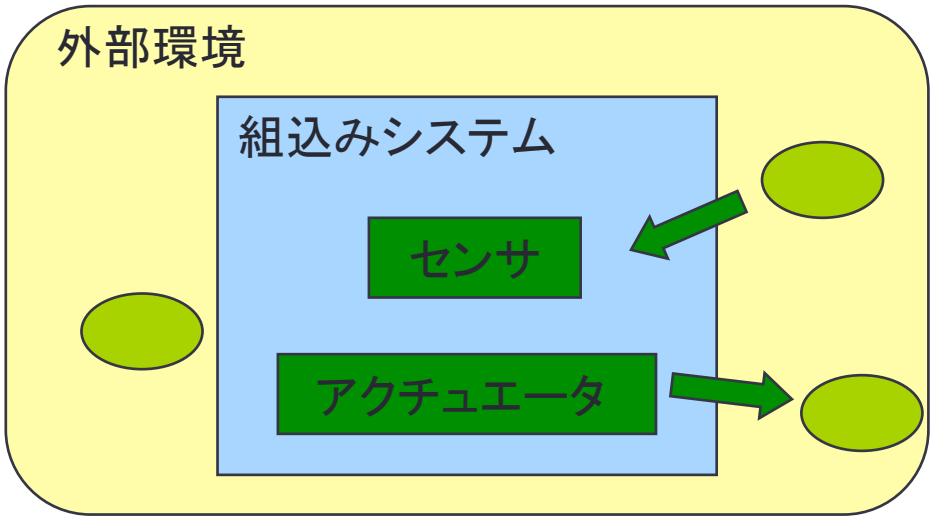
- 交通量の多い道や急カーブのある道
  - 道路の状況にあった速度で走行できないため事故につながるおそれがある
- 凍結路や積雪路などのすべりやすい路面
  - タイヤが空転し、車のコントロールを失うおそれがある
- 急な下り坂
  - エンジンブレーキが十分効かないため、セットした速度を超えてしまい、思わぬ事故につながるおそれがある

安全性と考慮すべき外部環境の境界

- 自動車開発において「安全性」は最も重要なアイテムの一つである
- しかし、すべての外部環境下において本当に安全性を確保できるのか？
  - もしも走行中に、突然、落石があったらどうする？
  - もしも走行中に、橋が崩壊したらどうする？
  - 結局、考えれば切りがない（フレーム問題）
- 考慮すべき（製品として保証すべき）外部環境の境界を決定することが必要

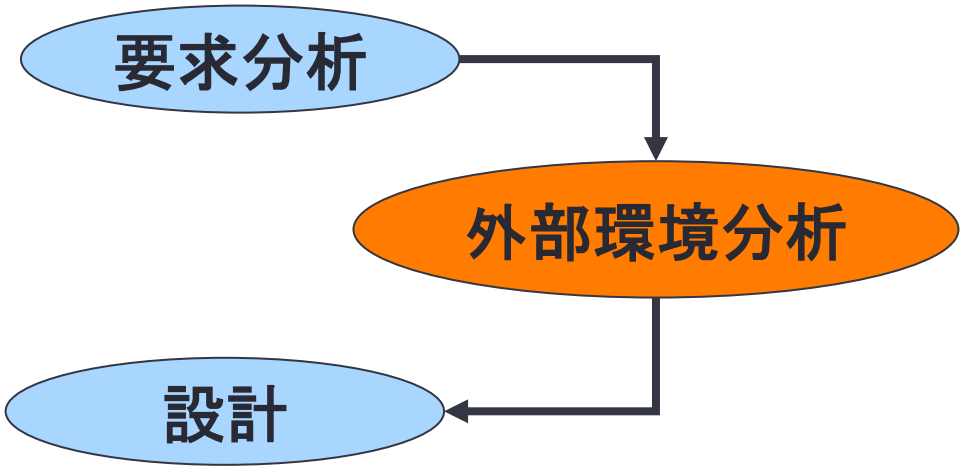
組込みシステムとは

- 機器に組込まれて動作するシステム
  - 車載システム、携帯電話、作業用ロボット
- センサ、アクチュエータを介して外部環境とインタラクション



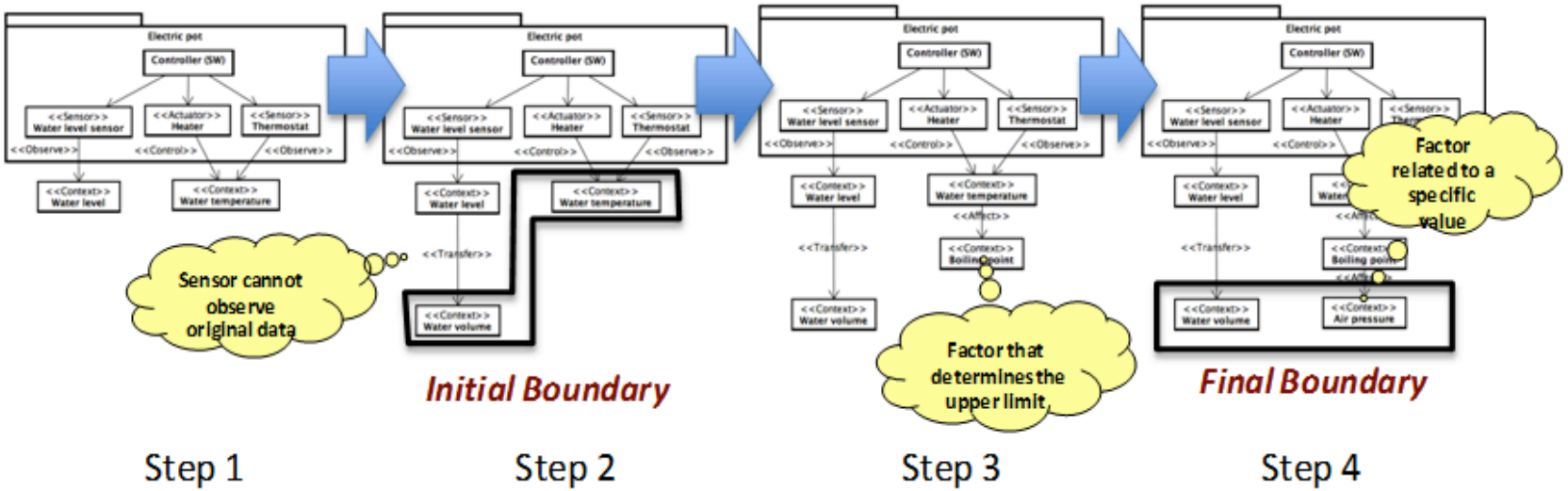
外部環境分析手法 CAMEmb

- CAMEmb（Context Analysis Method for Embedded Systems）
- 要求分析終了時に外部環境を分析する工程を置く
- 外部環境に起因する不具合を減らすことができる



Naoyasu Ubayashi, Yasutaka Kamei, Masayuki Hirayama, and Tetsuo Tamai: A Context Analysis Method for Embedded Systems ---Exploring a Requirement Boundary between a System and Its Context, 19th IEEE International Requirements Engineering Conference (RE 2011), IEEE Computer Society, pp.143-152 (2011).

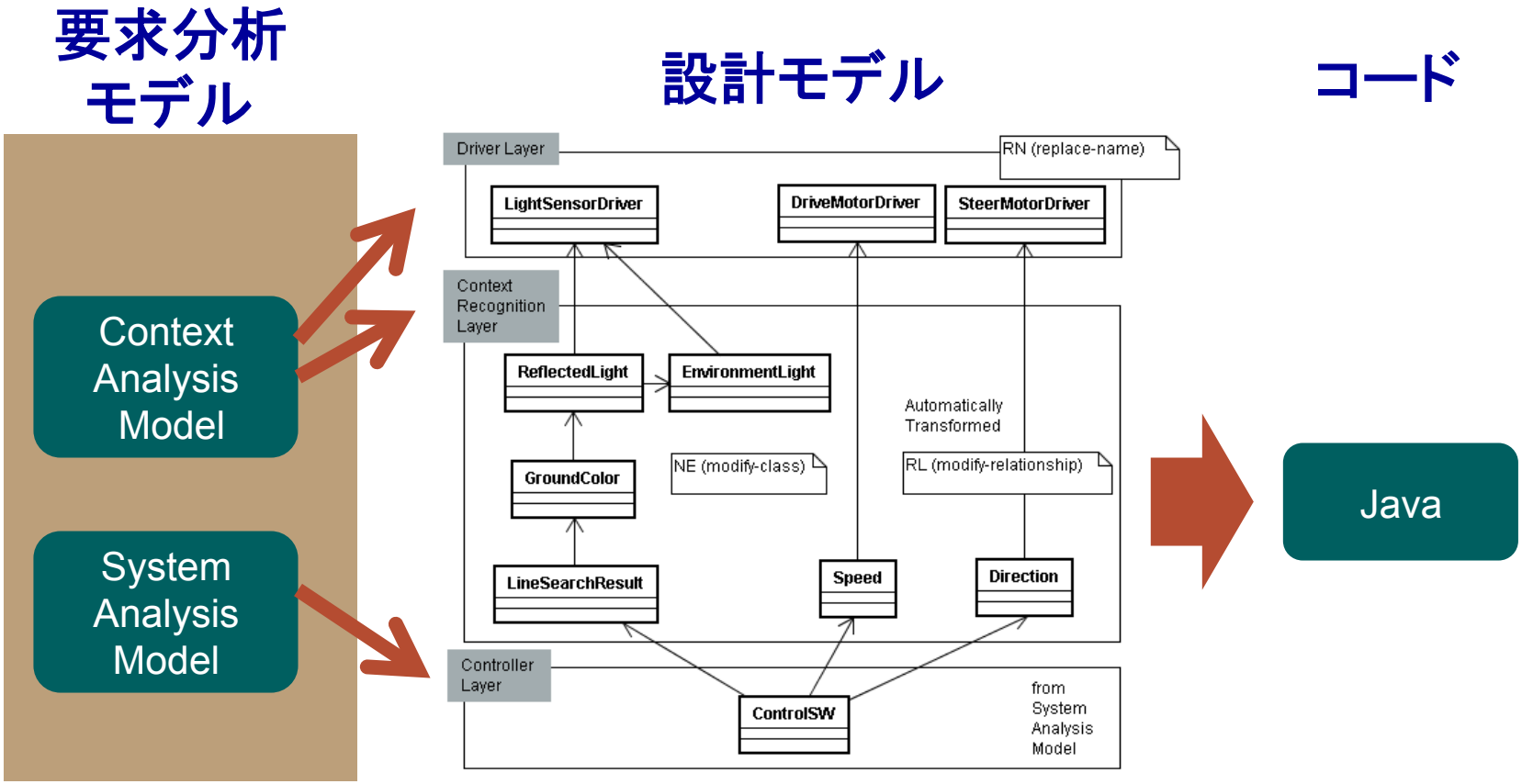
CAMEmbによる外部環境分析



外部環境分析のためのガイドワード

No.	Category of ≪ *Affect* ≫	Guide word
1.	physical phenomena	factor that determines the upper limit
2.	physical phenomena	factor that determines the lower limit
3.	physical phenomena	factor related to a specific value
4.	influence to sensing	factor that interferes with the observation
5.	influence to actuation	factor that interferes with the control

CAMEmbベースのモデル駆動開発



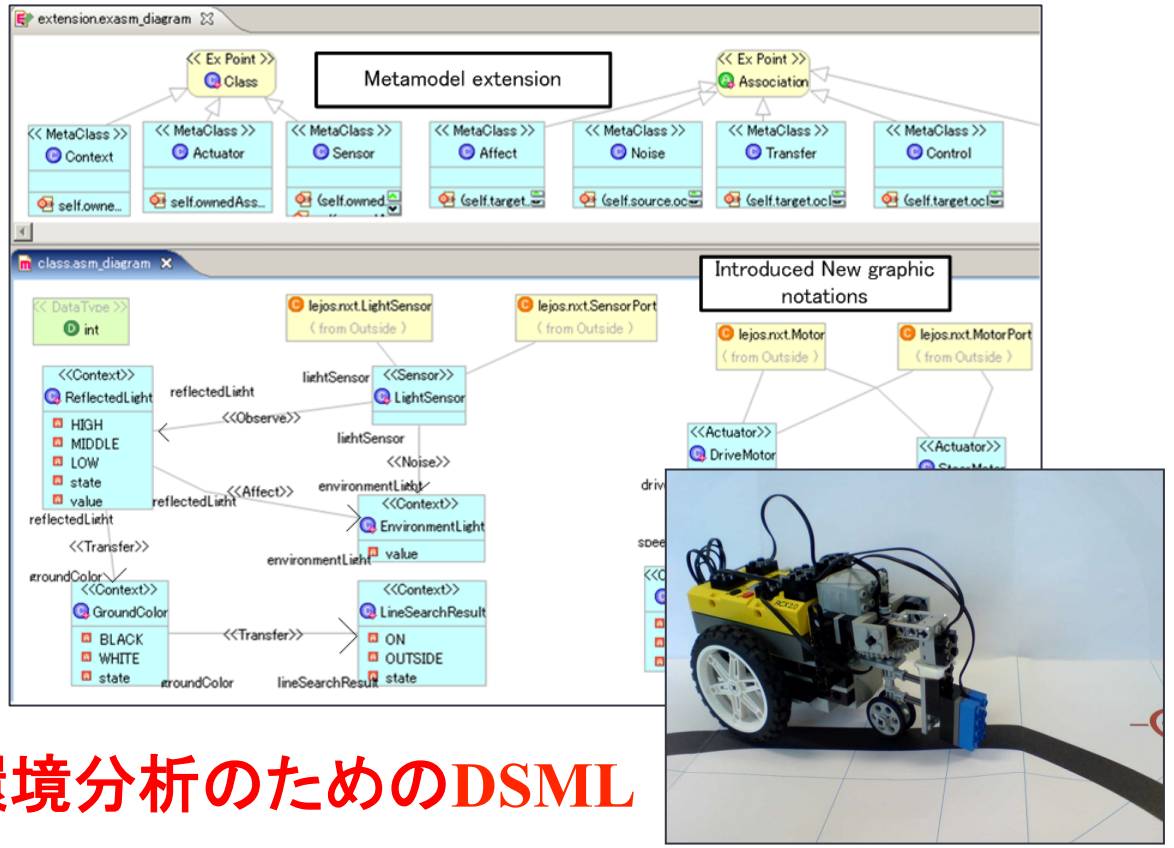
外部環境分析結果をコードに反映（自動化）

開発環境 CAMEmbModeler

- モデルエディタ
  - EMF
  - GMF
- モデルコンパイラ
  - アスペクト指向（AspectM）
  - Prolog

拡張可能なアスペクト指向モデリング言語AspectMを使用してDSLを構築

ライントレーサの外部環境モデル



外部環境分析のためのDSML

Ubayashi, N., Otsubo, G., Noda, K., and Yoshida, J.:
An Extensible Aspect-oriented Modeling Environment, CAiSE 2009, pp.17-31, 2009.

事例2：YET ANOTHER MDD

コード生成を仮定しないMDD

Naoyasu Ubayashi, Jun Nomura, and Tetsuo Tamai, Archface: A Contract Place Where Architectural Design and Code Meet Together, 32rd ACM/IEEE International Conference on Software Engineering (ICSE 2010), ACM PRESS, pp.75-84 (2010).

MDDの限界

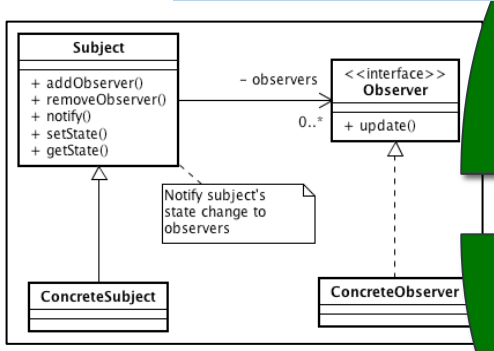
- DSLが提供される場合はMDDは有効。
- その一方、DSLが提供できない一般的な開発では、コード生成をベースとするMDDには限界がある。たとえば、UMLでモデリングし、Javaで実装するような場合。
- しかし、このような開発形態は多々見られる。これに対する支援はできないものか？

- モデルとは本当にコードの生成元のためのみに存在するのであろうか？
- モデルにはコードとは異なった役割があるのではないか？
- 鍵となるのは抽象化。

詳細化モデル vs. 抽象化モデル

コードの完全生成

詳細なモデル



抽象的なモデル

```
public class Subject{
    private Vector observers = new Vector();
   private String state = “”;

    public void addObserver(Observer o){ … }
    public void removeObserver(Observer o){ … }
    public void notify(){
        Iterator i = observers.iterator();
        while(i.hasNext() ){
            Observers o = (Observer)i.next();
            o.update( this );
        }
    }
   public void setState(String s){ state = s; }
   public String getState() {return state; }
}
```

抽象化は？

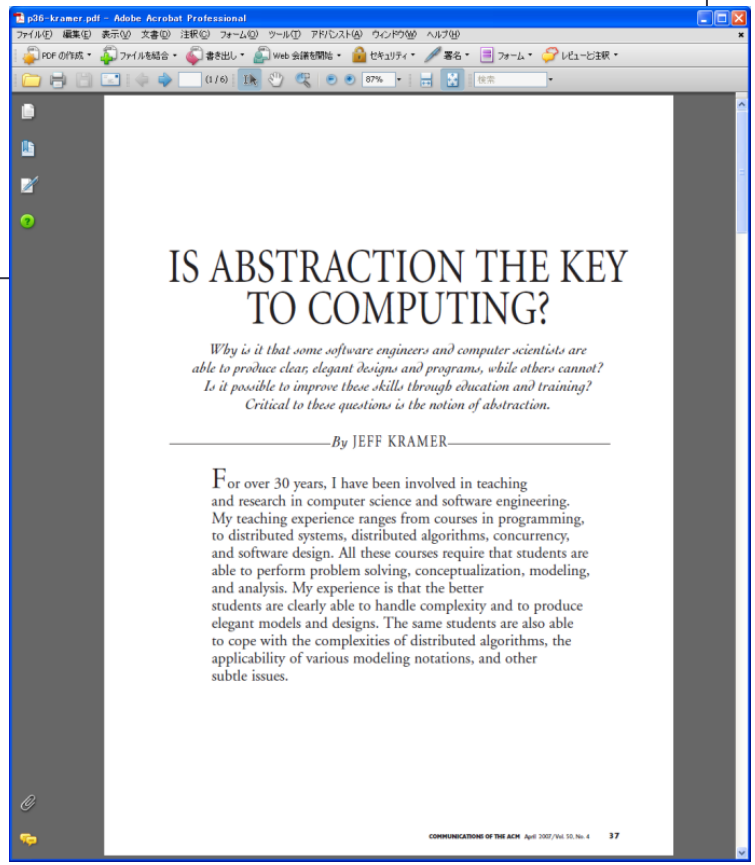
トレーサビリティは？

スケルトンコードの生成
ハンドコーディング

抽象化, Abstraction, …

- 明快でエレガントな設計やプログラミングができるエンジニアとそうでないエンジニアがいるのは何故であろうか？
- その答えは抽象化能力にある。

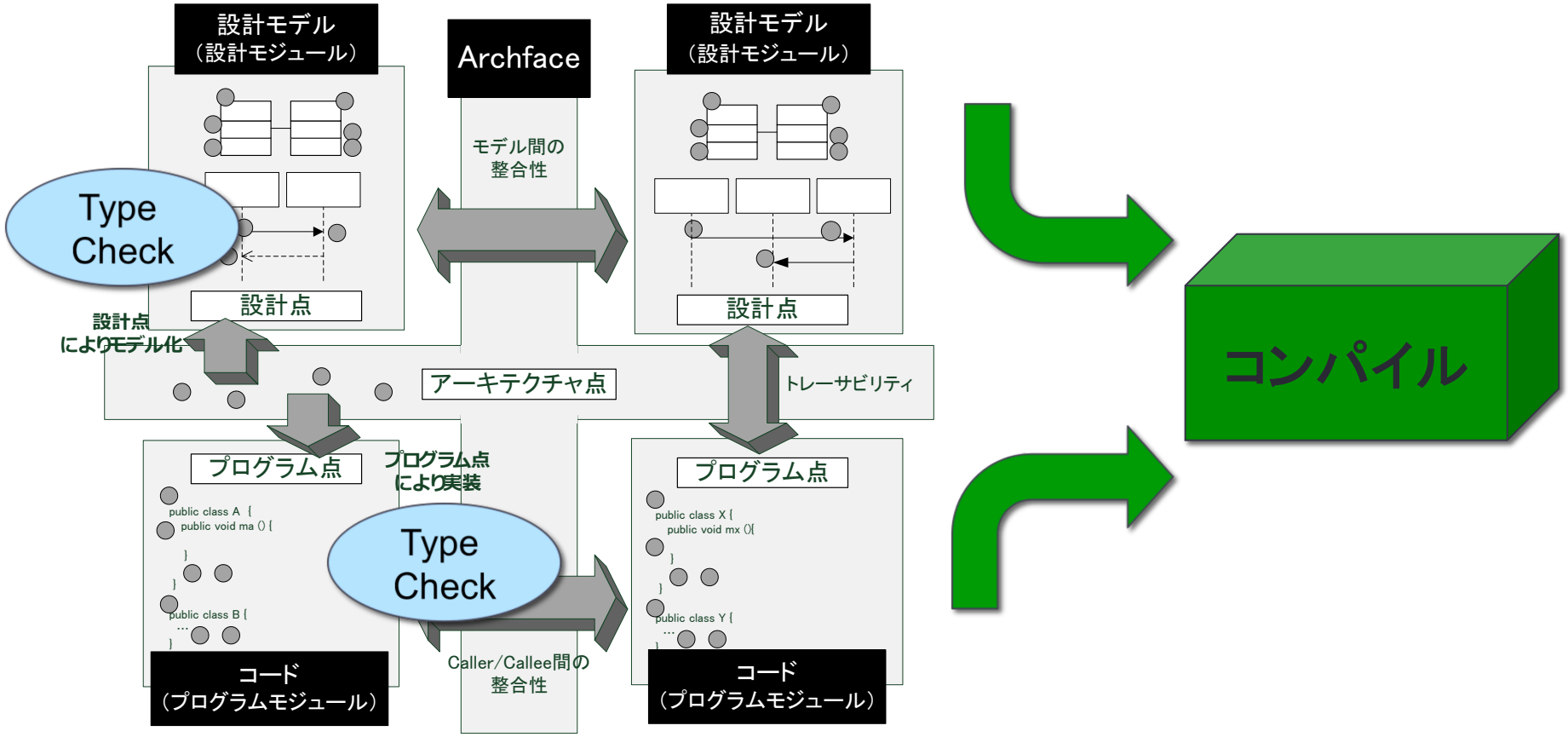
Kramer, J., Is Abstraction the Key to Computing ?
Communications of the ACM,
Vol. 50, Issue 4, pp.36-42, 2007.



我々のアプローチ

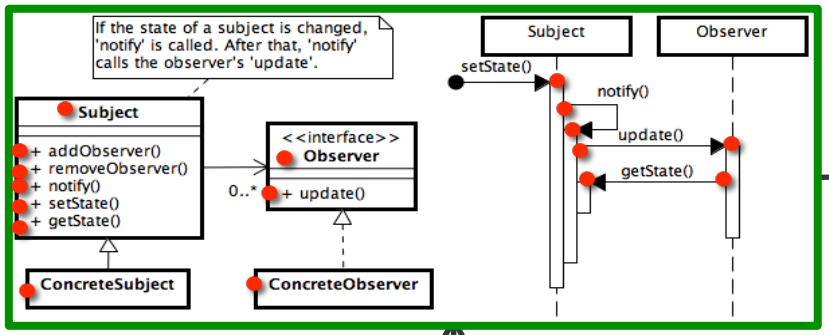
- モデルを抽象化記述のためのSWモジュールと見なす
- モデルとコードはコンパイラの対象
- コンパイラをパスすればモデルとコードのトレーサビリティは保証

インタフェース機構 Archface



Naoyasu Ubayashi, Jun Nomura, and Tetsuo Tamai, Archface: A Contract Place Where Architectural Design and Code Meet Together, 32rd ACM/IEEE International Conference on Software Engineering (ICSE 2010), ACM PRESS, pp.75-84 (2010).

Archface =
設計 + プログラム インタフェース



アーキテクチャ点をどう選択するかにより抽象度を決定

設計点としてモデリング

Archface
（アーキテクチャ点）

プログラム点として実装

```
public class Subject{
    private Vector observers = new Vector();
    private String state = “”;
...
}
```

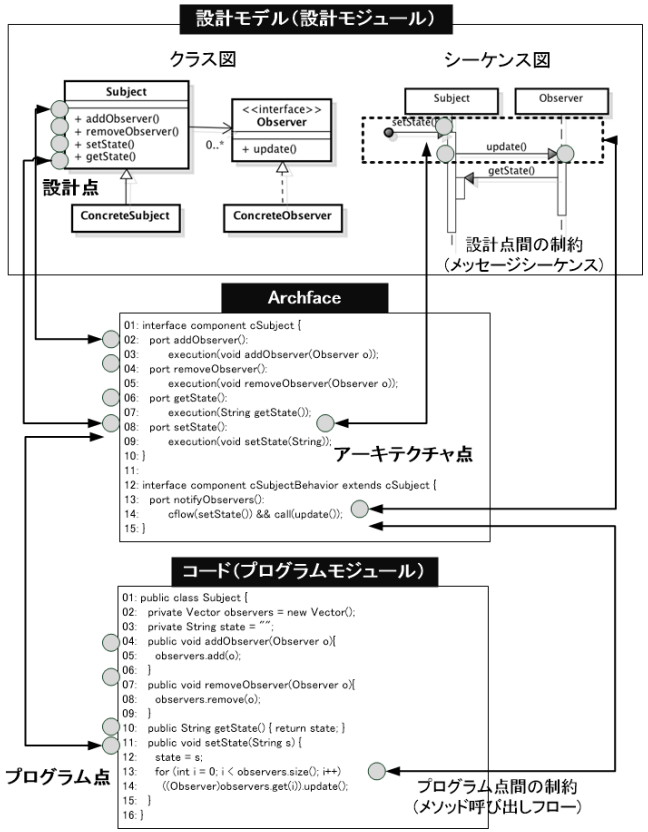
トレーサビリティチェック

設計は
Archfaceを
モデル化しているか？

Type Check
SMT Solver

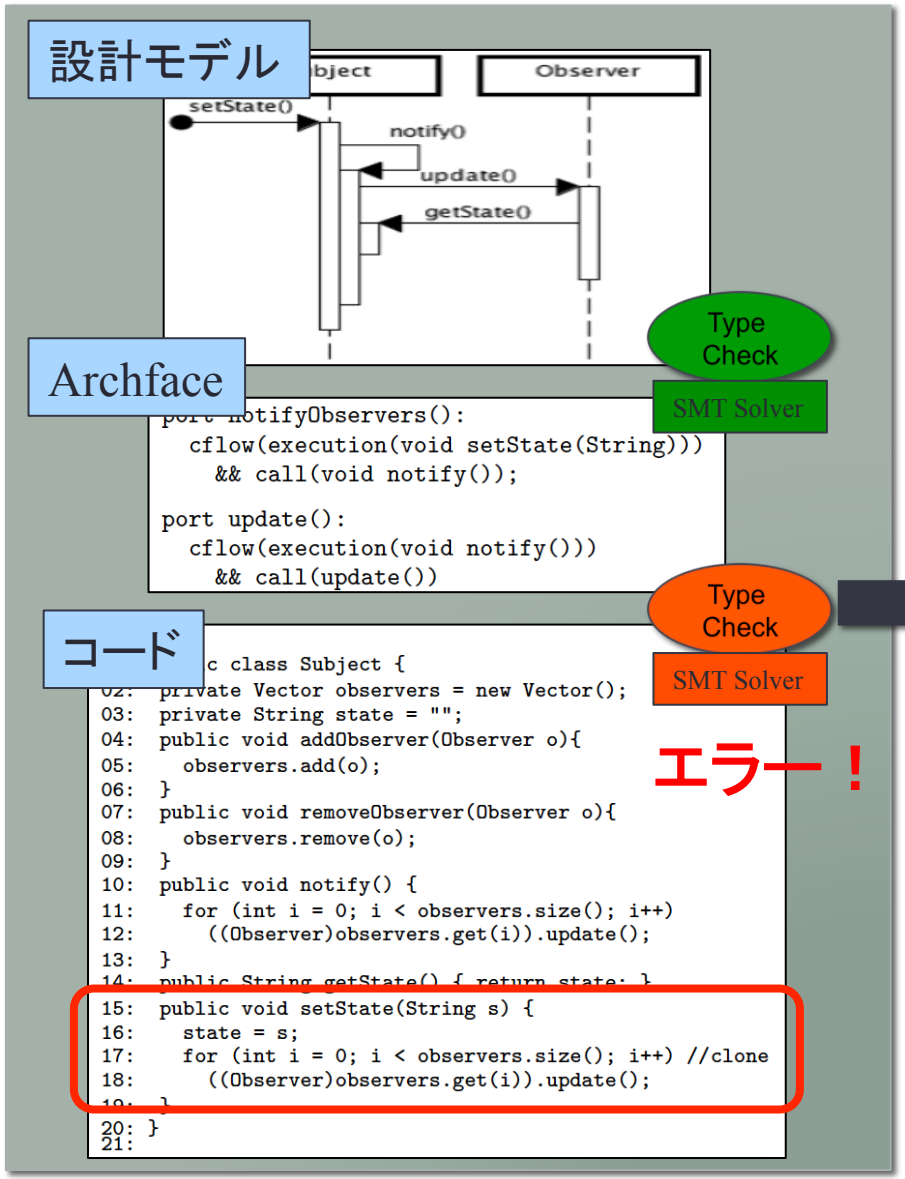
コードは
Archfaceを実装してい
るか？

Type Check
SMT Solver



Ubayashi, N. and Kamei, Y., Verifiable Architectural Interface for Supporting Model-Driven Development with Adequate Abstraction Level, MiSE 2012.

抽象化洗練の例



エラーへの対処方法

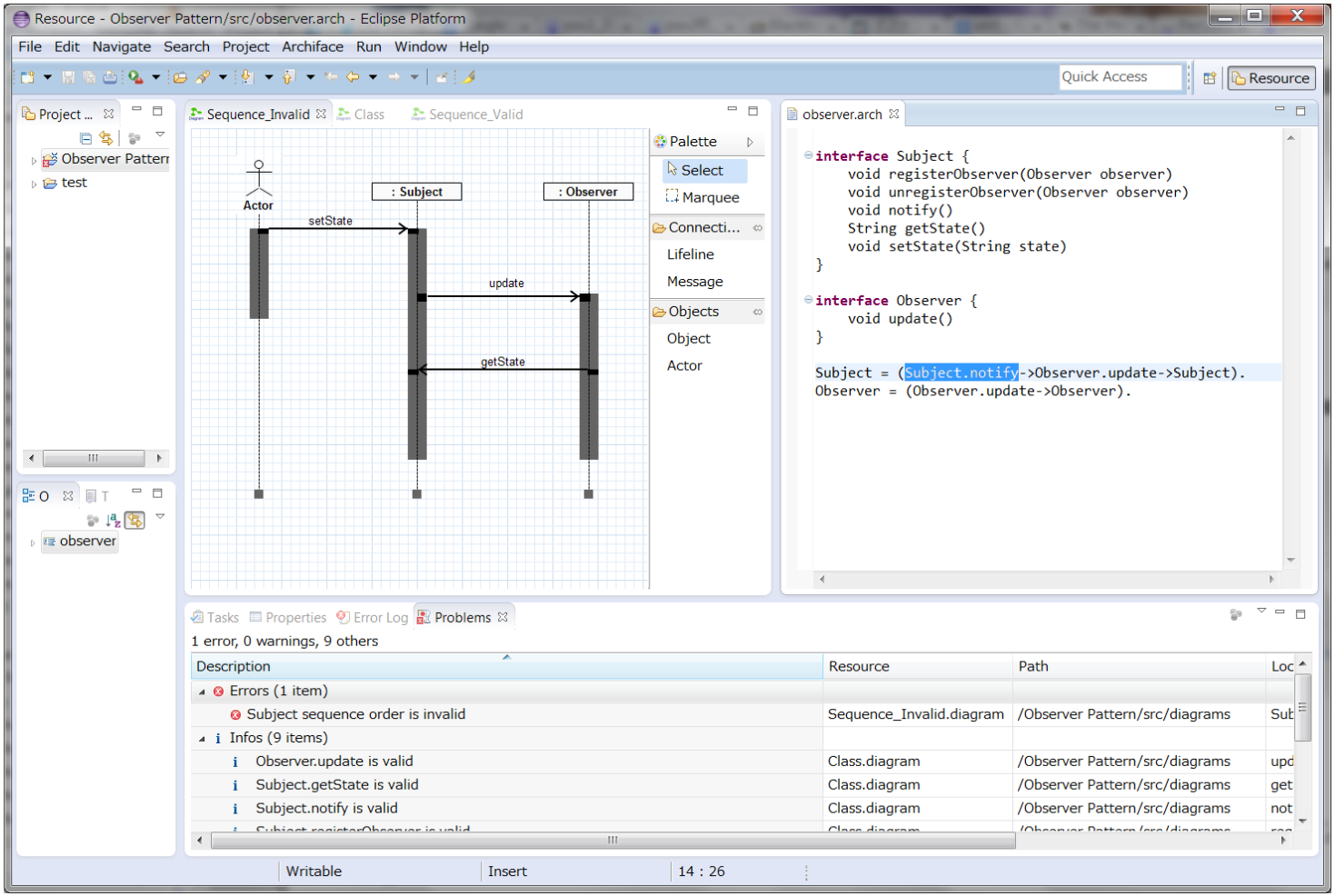
1. 設計の方が正しいと判断しコードを設計モデルに合わせる
2. コードの方が正しいと判断し設計モデルをコードに合わせる
3. 抽象レベルの設定に問題があると判断し設計モデルの抽象度を変更する

```
port notifyObservers():
  cflow(execution(void setState(String)))
  && call(update());
```

Type Check
SMT Solver

これに伴い設計モデルも変更

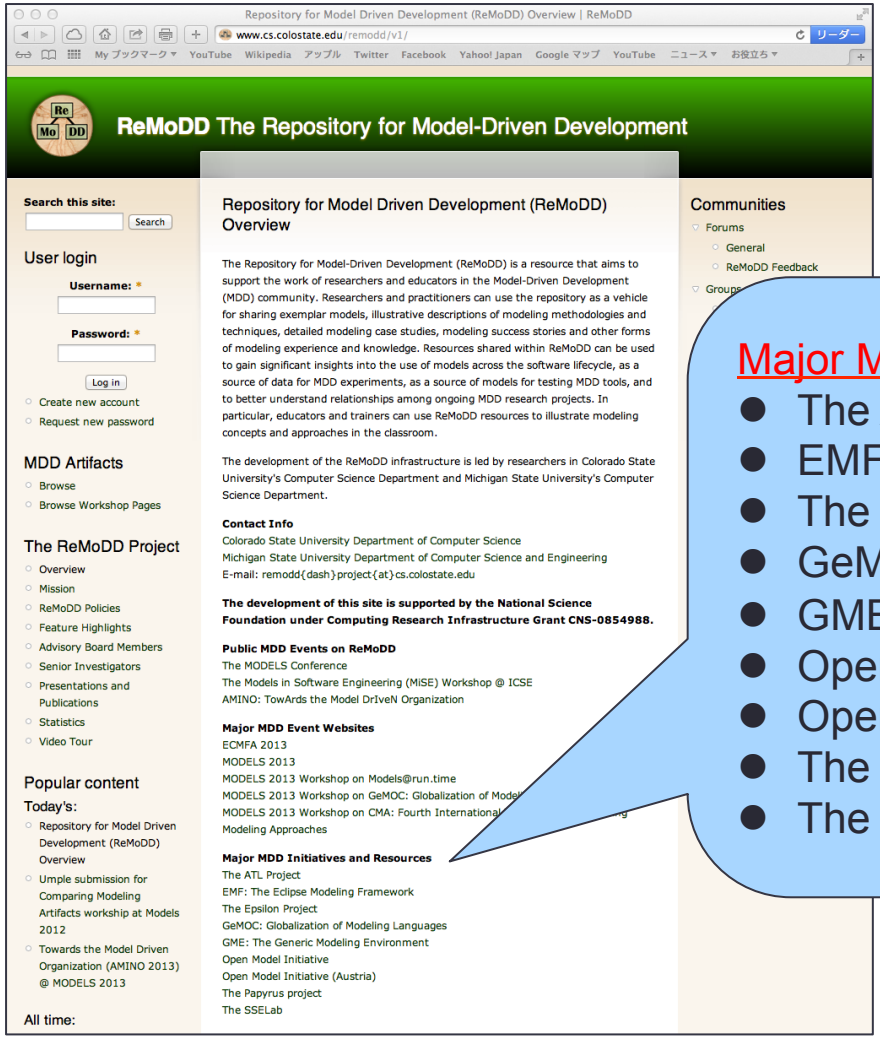
iArch: Archface中心のIDE



EMF + Graphiti + Xtext で開発

第五部　今後の動向

MDDのためのリポジトリ ReMoDD



Major MDD Initiatives and Resources

- The ATL Project
- EMF: The Eclipse Modeling Framework
- The Epsilon Project
- GeMOC: Globalization of Modeling Languages
- GME: The Generic Modeling Environment
- Open Model Initiative
- Open Model Initiative (Austria)
- The Papyrus project
- The SSELab

http://www.cs.colostate.edu/remodd/v1/

今後のモデリングコミュニティ

- オープン化
  - The Open Model Initiative（http://www.openmodels.org）
    - 誰もが、コピー、利用、修正、配布できるオープンな参照モデルを共同で開発するプロジェクト
    - オープンソースコミュニティのモデリング版
- DSL開発の容易化加速
  - 様々なプロジェクト
    - EMF（Eclipse Modeling Framework）
    - GMF（Graphical Modeling Framework）
    - Graphiti（Graphical Tooling Infrastructure）
    - Xtext（Framework for Development of Programming Languages and Domain Specific Languages）
    - ATL（ATL Transformation Language）
    - Papyrus

ホットな研究トピック

- スケッチとしてのモデリングを支援する環境
- 不確かさを含むモデル駆動開発
- トレーサビリティ
- 双方向変換
- Model@Runtime
- LOP（Language-oriented programming）
- Model-Oriented Programming – Umple.org

- 参考資料
  - チュートリアル「モデル駆動工学の原理と応用」（日本ソフトウェア科学会誌「コンピュータソフトウェア」）

おわり

謝辞：本資料作成にあたり、九州大学 久住憲嗣准教授に
多大な協力をいただきました。