CHAPTER 3 - MDSE Use Cases

Overview

1 Introduction

1.1 Human Cognitive Processes
1.2 Models
1.3 Model Engineering

2 MDSE Principles

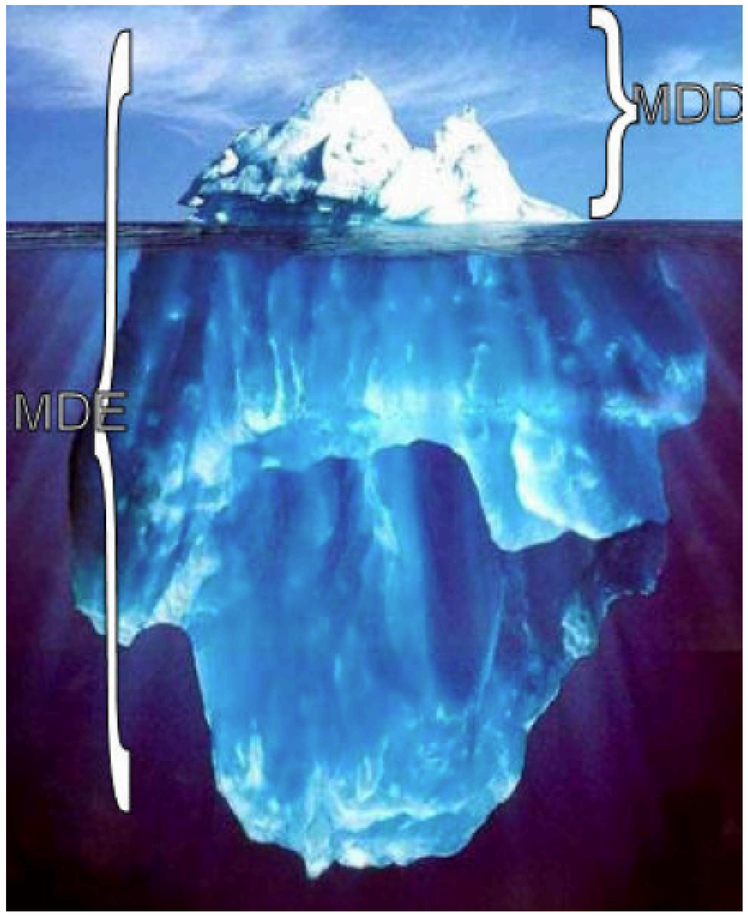
2.1 MDSE Basics
2.2 The MD* Jungle of Acronyms
2.3 Modeling Languages and Metamodeling
2.4 Overview Considered Approaches
2.5 Tool Support
2.6 Criticisms of MDSE

3 MDSE Use Cases

3.1 MDSE applications
3.2 USE CASE1 – Model driven development
3.3 USE CASE2 – Systems interoperability
3.4 USE CASE3 – Model driven reverse engineering

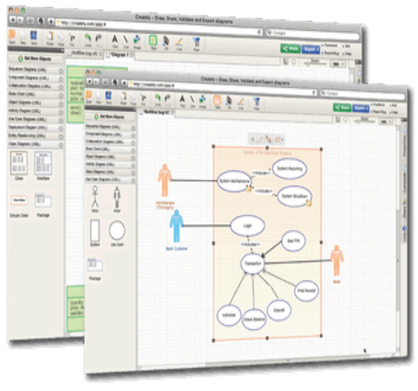
MDSE has many applications

- MDSE goes far beyond code-generation
- MDD is just the tip of the iceberg
  - » And MDA a specific “realization” of MDD when using OMG standards

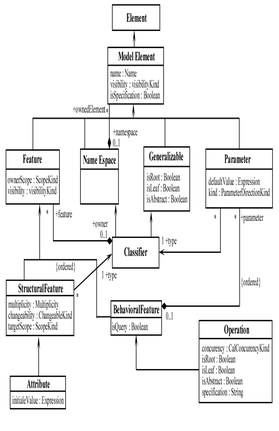


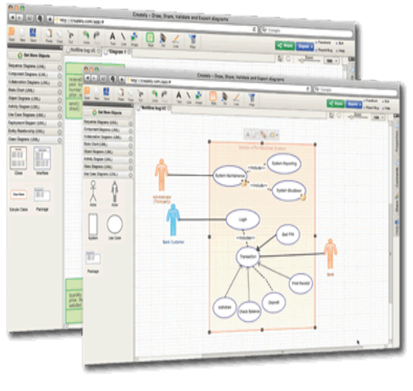
Three killer MDSE applications

Code Generation



Software Modernization



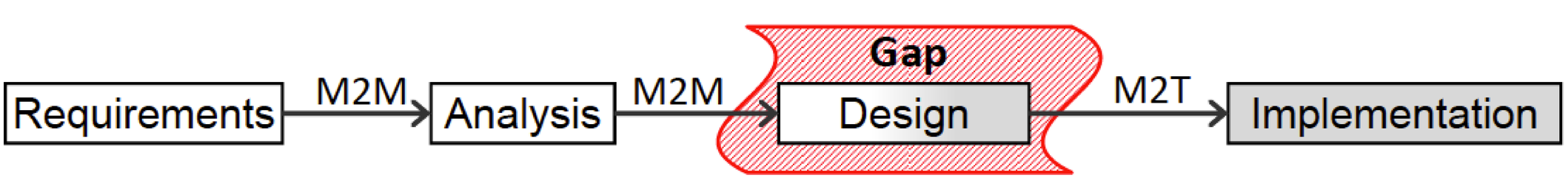


Systems interoperability

Use Case1 – Model driven development

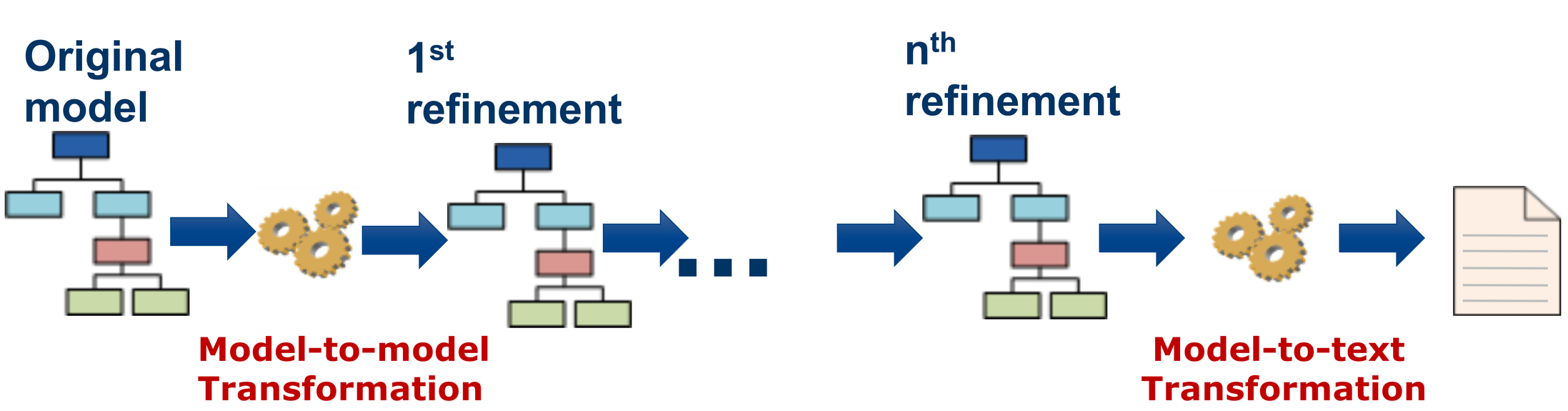
MDD contribution: Communication

- Models capture and organize the understanding of the system within a group of people
- Models as lingua franca between actors from business and IT divisions



MDD contribution: Productivity

- MDD (semi)automates software development
- In MDD, software is derived through a series of model-to-model transformations (possibly) ending with a model-to-text transformations that produces the final code



Model Driven Architecture

- The Object Management Group (OMG) has defined its own comprehensive proposal for applying MDE practices to system's development:

MDA (Model-Driven Architecture)

Four principles of MDA

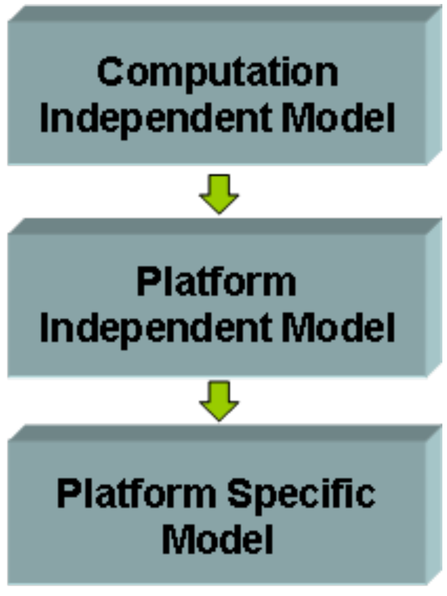
- Models must be expressed in a well-defined notation, so as to enable effective communication and understanding
- Systems specifications must be organized around a set of models and associated transformations
  - » implementing mappings and relations between the models.
  - » multi-layered and multi-perspective architectural framework.
- Models must be compliant with metamodels
- Increase acceptance, broad adoption and tool competition for MDE

Definitions according to MDA

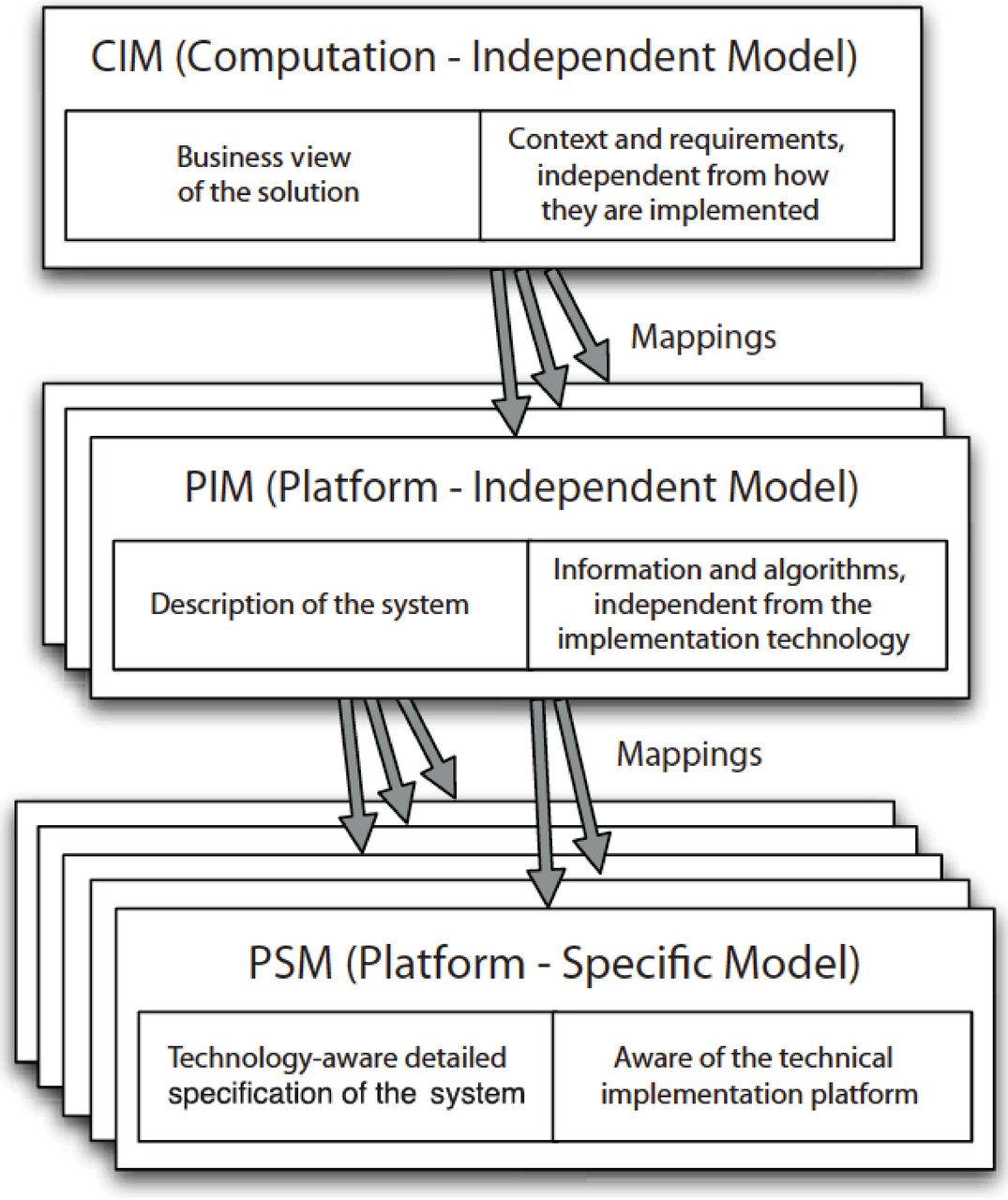
- System: The subject of any MDA specification (program, computer system, federation of systems)
- Problem Space (or Domain): The context or environment of the system
- Solution Space: The spectrum of possible solutions that satisfy the reqs.
- Model: Any representation of the system and/or its environment
- Architecture: The specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors
- Platform: Set of subsystems and technologies that provide a coherent set of functionalities for a specified goal
- Viewpoint: A description of a system that focuses on one or more particular concerns
- View: A model of a system seen under a specific viewpoint
- Transformation: The conversion of a model into another model

Modeling Levels
CIM, PIM, PSM

- Computation independent (CIM): describe requirements and needs at a very abstract level, without any reference to implementation aspects (e.g., description of user requirements or business objectives);
- Platform independent (PIM): define the behavior of the systems in terms of stored data and performed algorithms, without any technical or technological details;
- Platform specific (PSM): define all the technological aspects in detail.

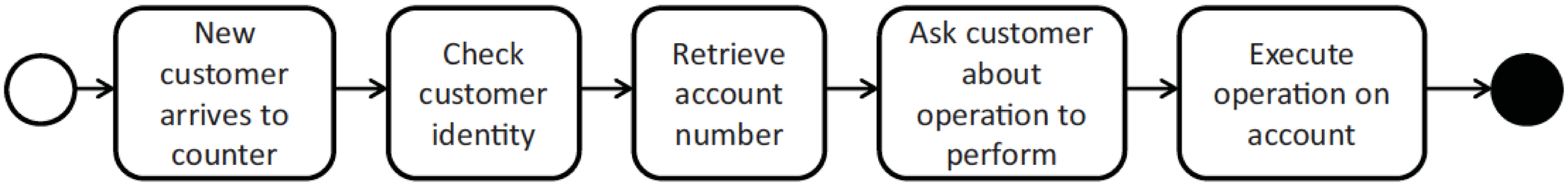


CIM, PIM and PSM



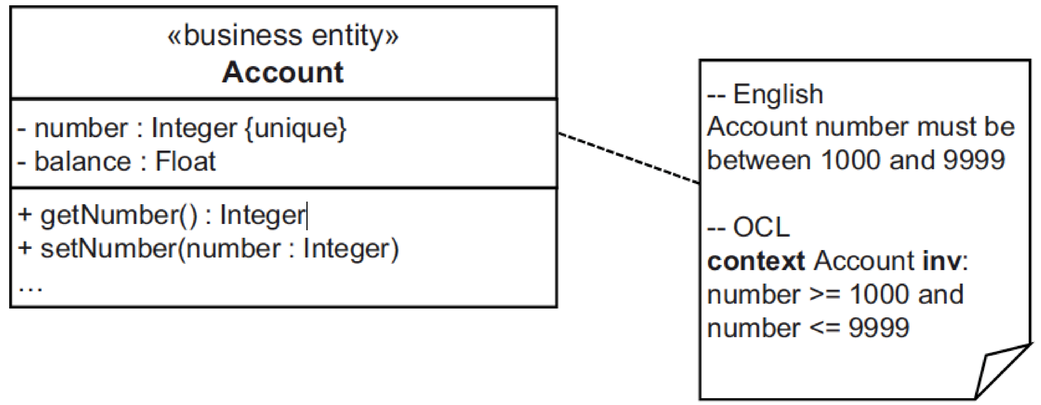
CIM

- MDA Computation Independent Model (CIM)
  - » E.g., business process



PIM

- specification of structure and behaviour of a system, abstracted from technological details

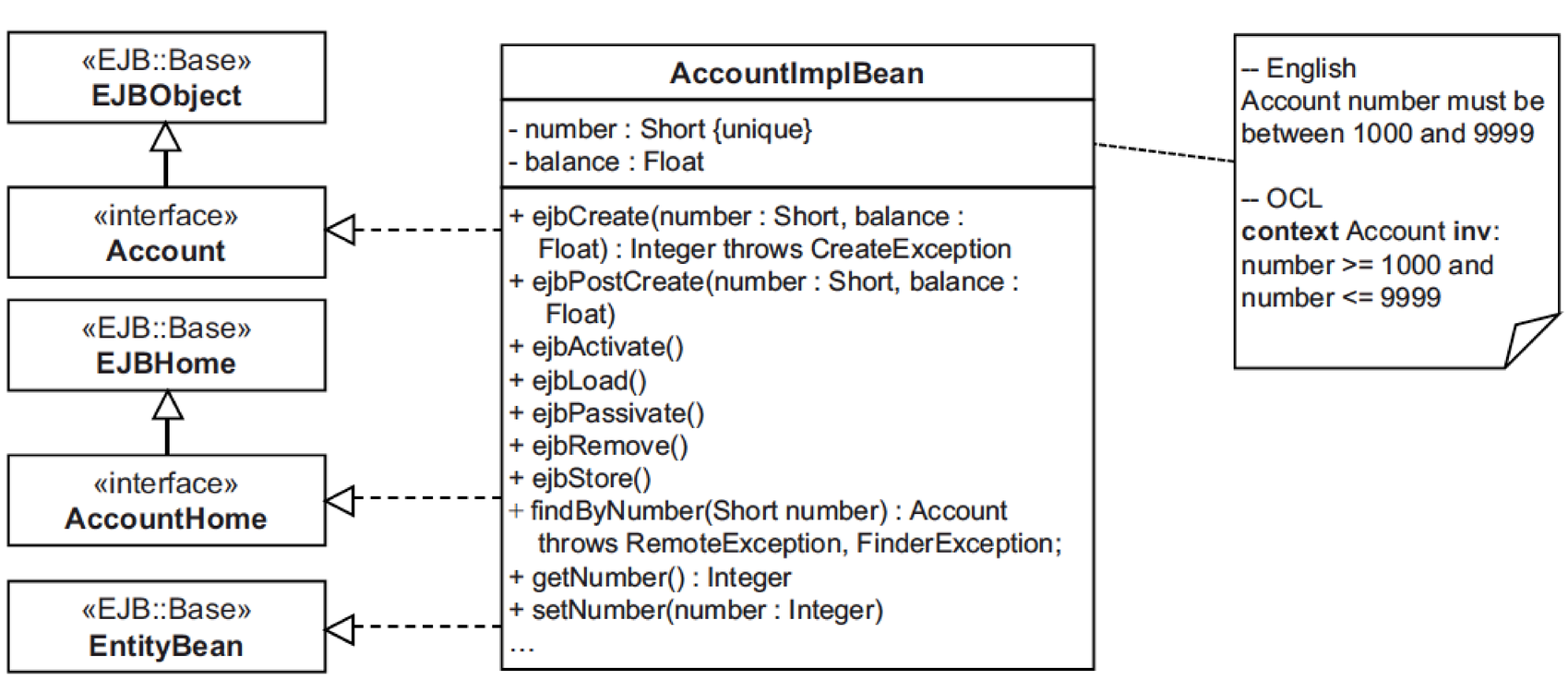


- Using the UML (optional)

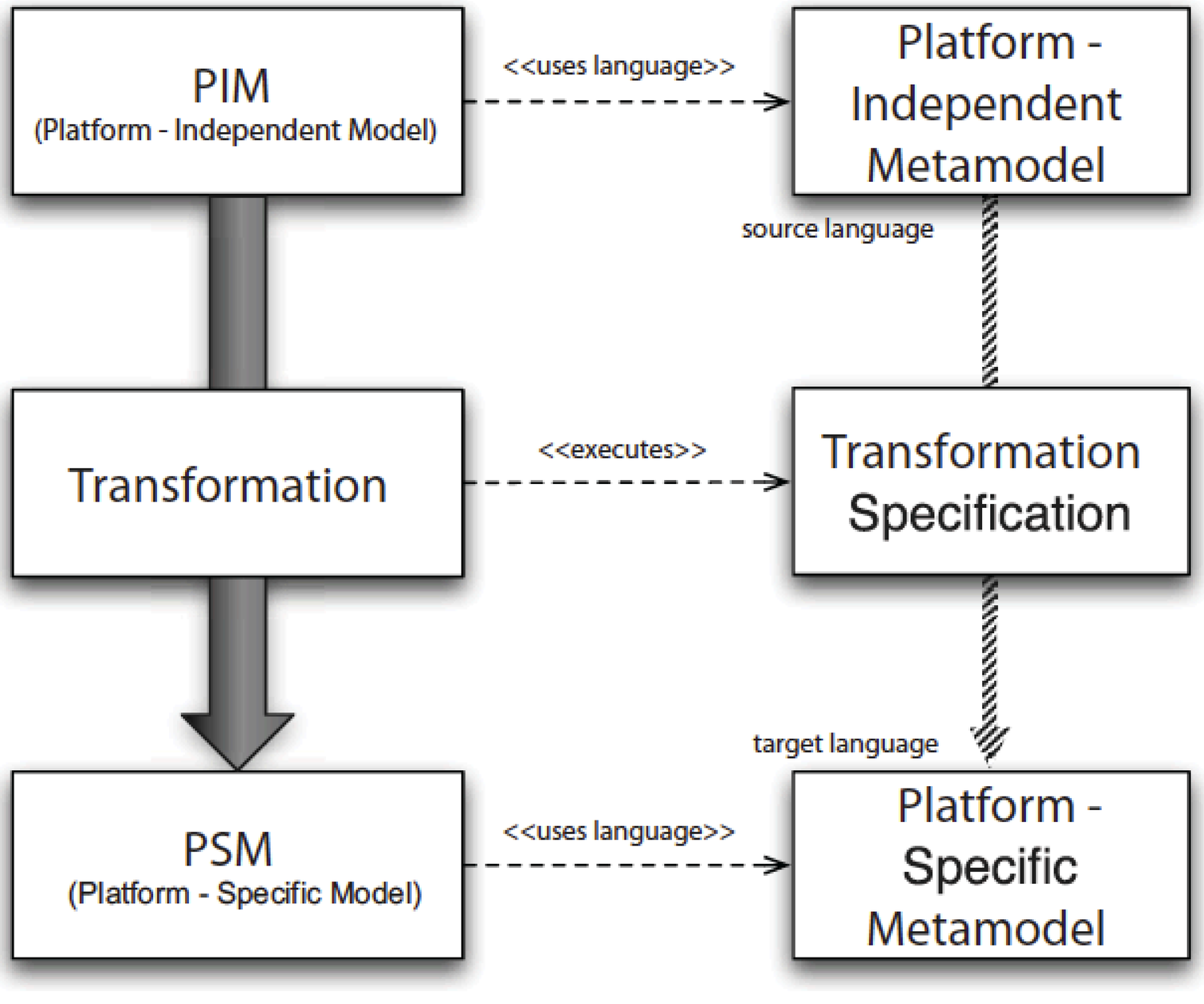
- Abstraction of structure and behaviour of a system with the PIM simplifies the following:
  - » Validation for correctness of the model
  - » Create implementations on different platforms
  - » Tool support during implementation

PSM

- Specifies how the functionality described in the PIM is realized on a certain platform
- Using a UML-Profile for the selected platform, e.g., EJB



CIM – PIM – PSM mappings



Modeling language specification

- MDA's core is UML, a standard general-purpose software modeling language

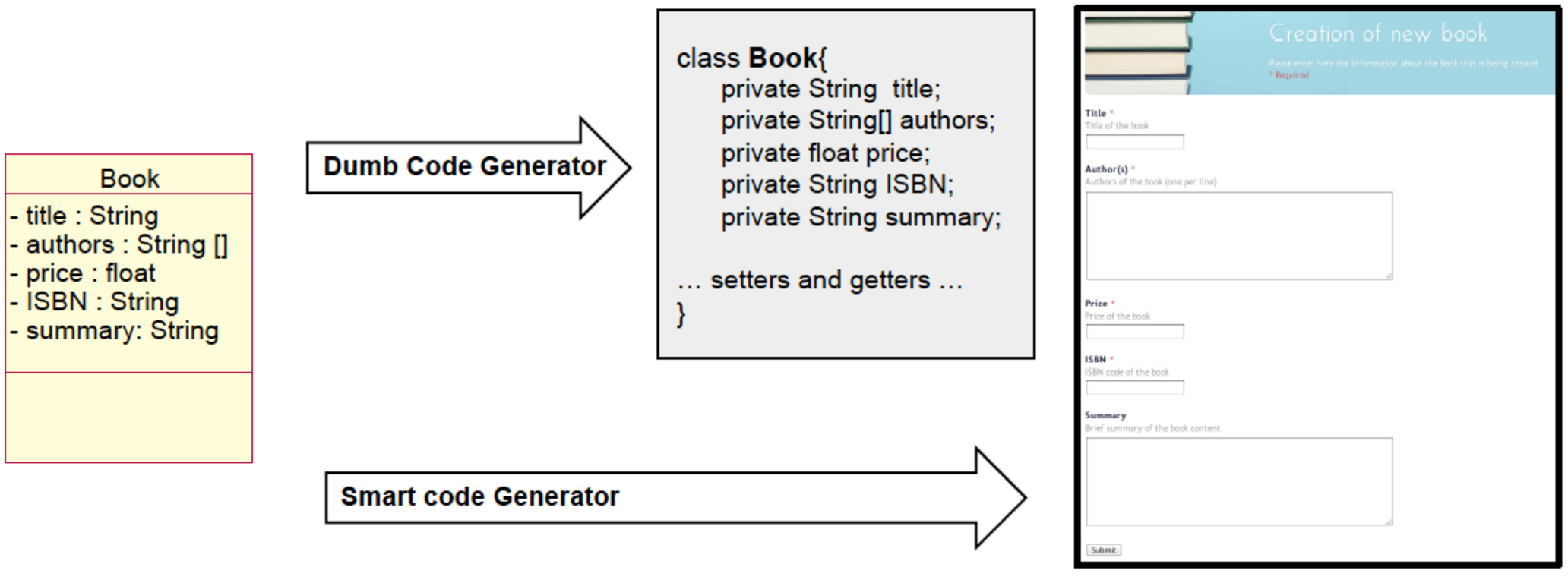
- Two options for specifying your languages:
  - (Domain-specific) UML Extensions can be defined through UML Profiles
  - Full-fledged Domain-specific languages (DSMLs) can be defined by MOF

Executable models

- An executable model is a model complete enough to be executable
- From a theoretical point of view, a model is executable when its operational semantics are fully specified
- In practice, the executability of a model may depend on the adopted execution engine
  - » models which are not entirely specified but that can be executed by some advanced tools that are able to fill the gaps
  - » Completely formalized models that cannot be executed because an appropriate execution engine is missing.

Smart vs. dumb execution engines

- CRUD operation typically account for 80% of the overall software functionality
- Huge spared effort through simple generation rules



Executable models

- Most popular: Executable UML models
- Executable UML development method (xUML) initially proposed by Steve Mellor
- Based on an action language (kind of imperative pseudocode)

- Current standards
  - » Foundational Subset for Executable UML Models (fUML)
  - » Action language is the Action Language for fUML (Alf)
    - basically a textual notation for UML behaviors that can be attached to a UML model

Executable models: 2 main approaches

- Code generation: generating running code from a higher level model in order to create a working application
  - » by means of a rule-based template engine
  - » common IDE tools can be used to render the source code produced

- Model interpretation: interpreting the models and making them run

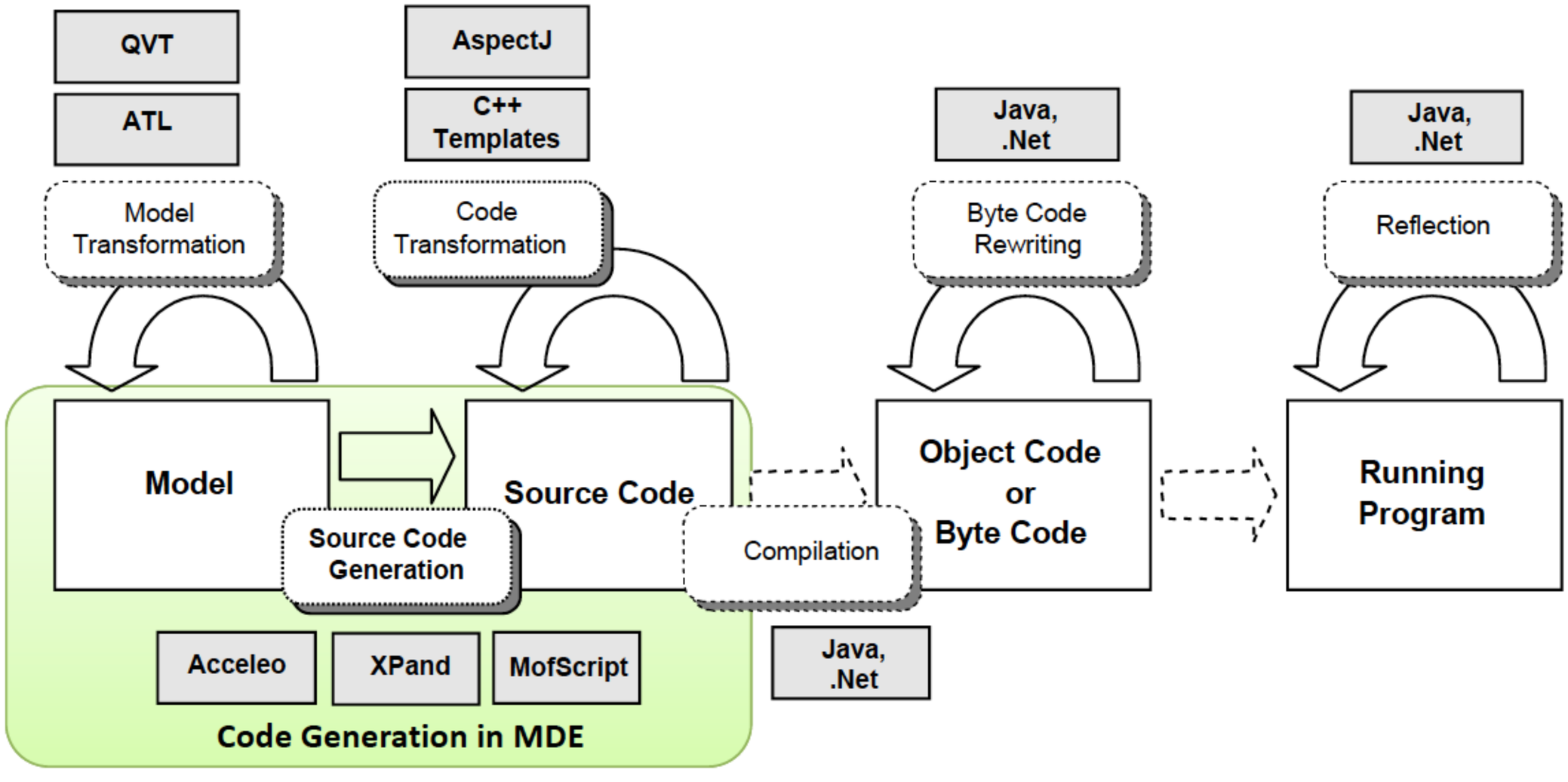
  - Non-empty intersection between the two options

Code Generation

- Goal: generating running code from higher level models
  - » Like compilers producing executable binary files from source code
  - » Also known as model compilers

- Once the source code is generated state-of-the-art IDEs can be used to manipulate the code

Code Generation: Scope



Code Generation: Benefits

- Intellectual property
- Separation of modeling and execution
- Multi-platform generation
- Generators simpler than interpreters
- Reuse of existing artefacts
- Adaptation to enterprise policies
- Better performances

Code Generation: Partial Generation

- Input models are not complete & code generator is not smart enough to derive or guess the missing information
- Programmers will need to complete the code manually
- Caution! Breaking the generation cycle is dangerous

- Solutions:
  - » Defining protected areas in the code, which are the ones to be manually edited by the developer
  - » Using round-trip engineering tools (not many available)
  - » Better to do complete generation of parts of the system instead of partial generation of the full system

Code Generation: Turing test

- A human judge examines the code generated by one programmer and one code-generation tool for the same formal specification. If the judge cannot reliably tell the tool from the human, the tool is said to have passed the test

Model interpretation

- A generic engine parses and executes the model on-the-fly using an interpretation approach

- Benefits
  - » Faster changes & Transparent (re)deployment
  - » Better portability (if the vendor supports several platforms)
  - » The model is the code. Easier model debugging
  - » No deployment
  - » Updates of the model at runtime
  - » Higher level abstraction of the system (implemented by the interpreter)
  - » Updates in the interpreter may result in automatic improvements of your software

- Danger of becoming dependent of the application vendor. Limited influence in the –ities of the SW

Generation and interpretation

- Can be used together in the same process
  - » Interpretation at early prototyping / debugging time
  - » Generation for production and deployment
- Hybrid solutions are possible:
  - » Model interpretation based on internal code generation implementation
  - » Code generation that relies on predefined, configurable components / framework at runtime. The generated code is e.g., XML descriptor / configurations of the components

USE CASE2 – Systems interoperability

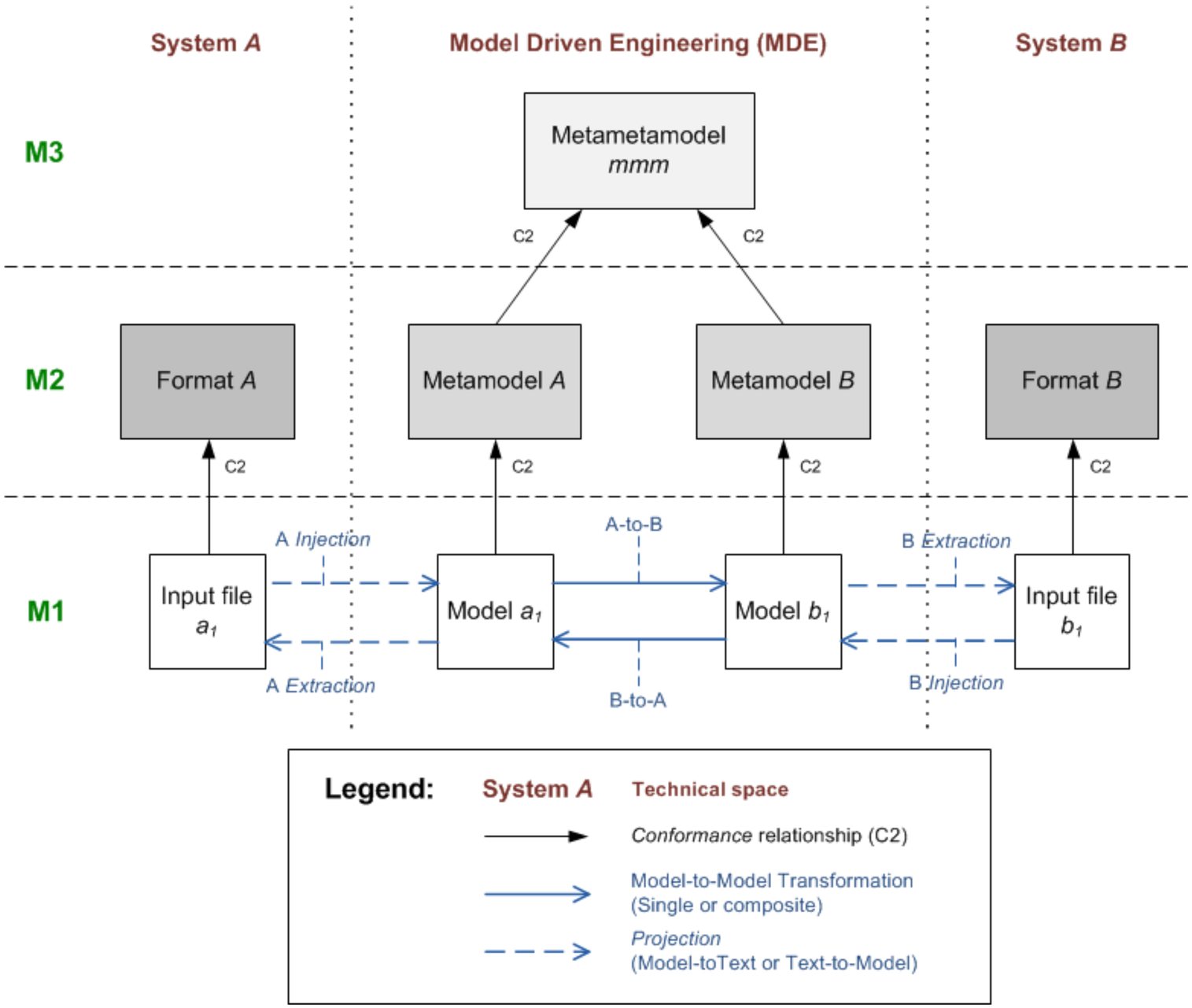
Interoperability

- Ability of two or more systems to exchange information (IEEE)
- Needed for collaborative work (e.g. using different tools), tool and language evolution, system integration…
- Interoperability must be done at the syntactic and semantic levels

Model-Driven Interoperability

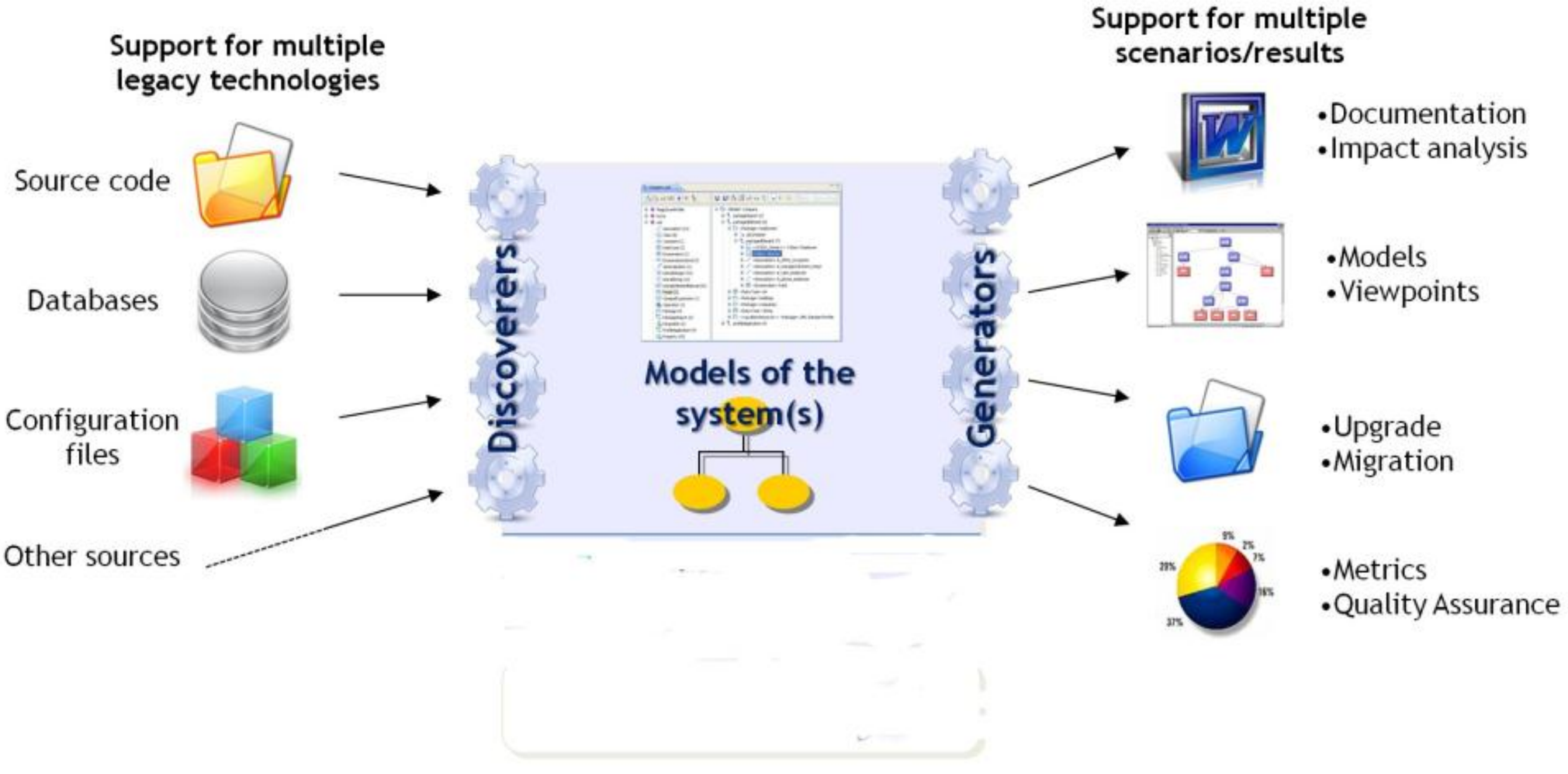
- MDSE techniques to bridge the interoperability gap
- The metamodels (i.e. “schemas”) of the two systems are made explicit and aligned
- Transformations follow the alignment to move information
  - » Injectors (text-to-model) represent system A data as a model (syntactic transformation)
  - » M2M transformation adapts the data to system B metamodel (semantic transformation)
  - » Extractors (model-to-text) generate the final System B output data (syntactic transformation).

MDI: Global schema



USE CASE3 – Model driven reverse engineering

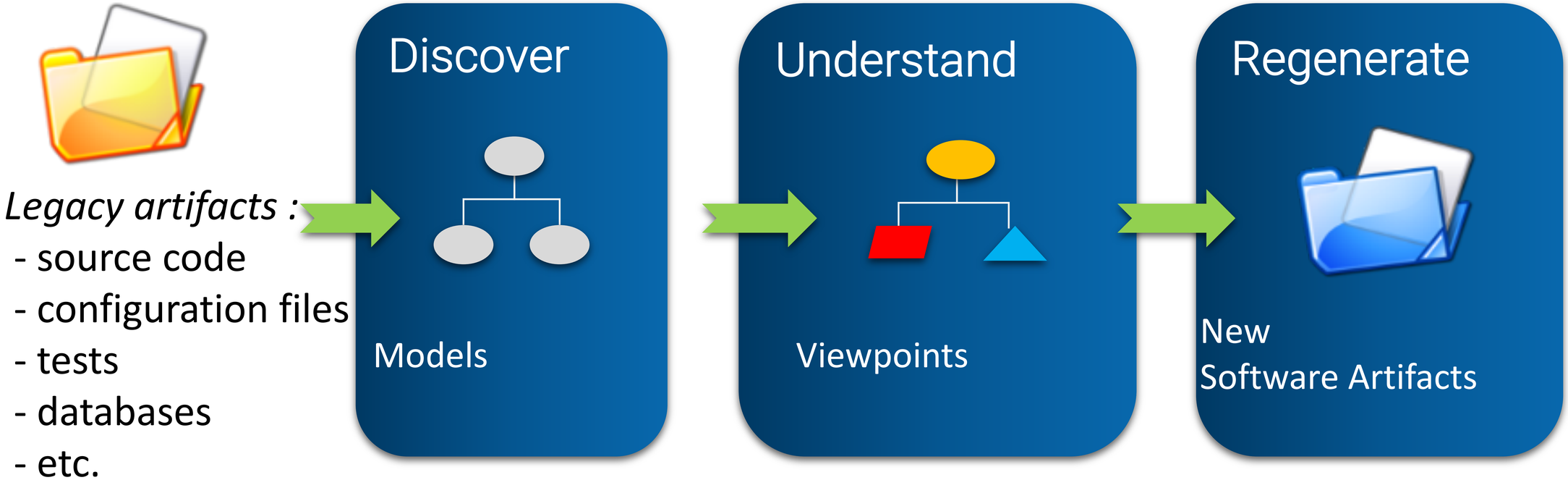
Need for reverse engineering



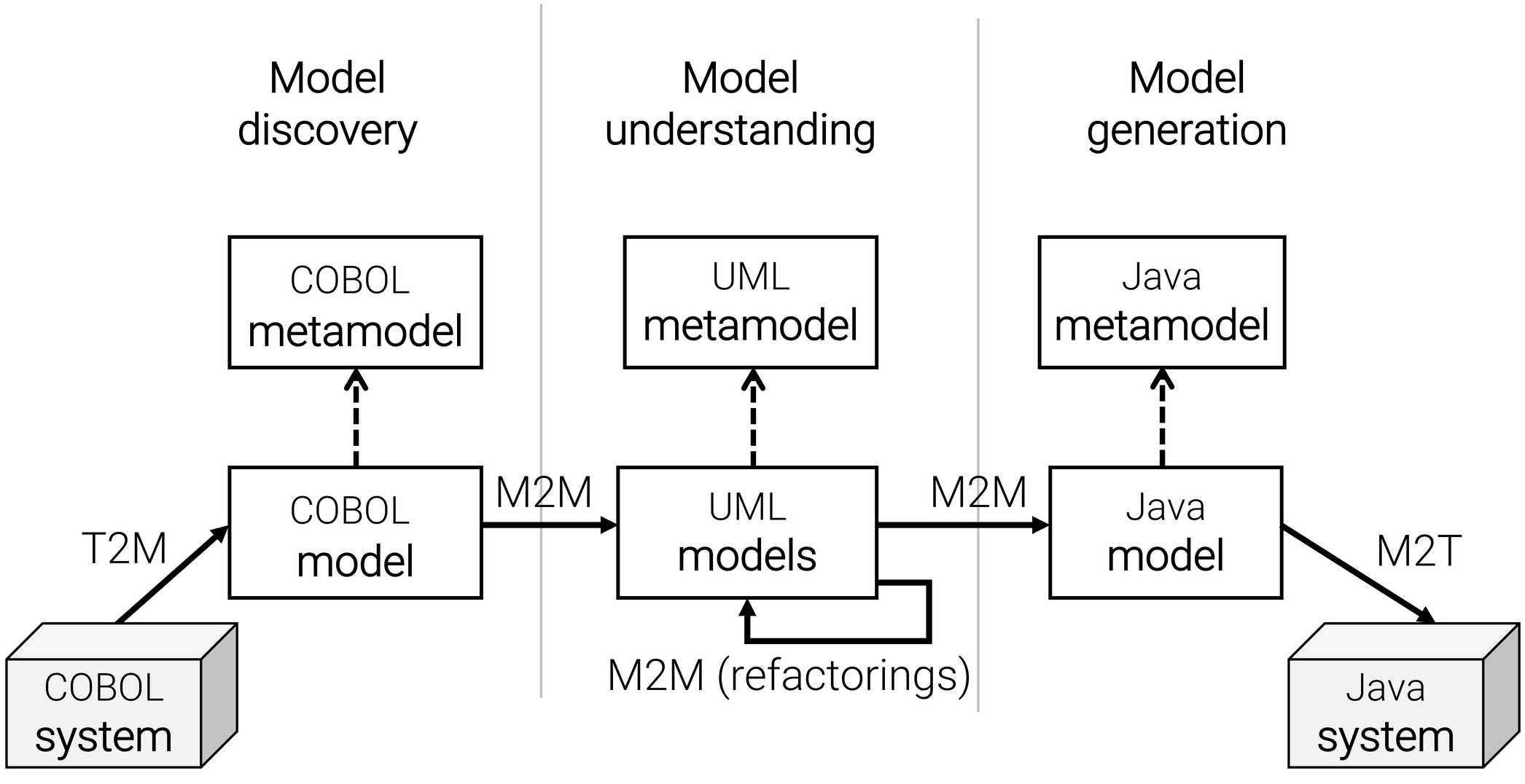
Model-driven reverse engineering

- Why? Models provide an homogeneous and interrelated representation of all legacy components.

  No information loss: initial models have a 1:1 correspondance with the code



Model-Driven Interoperability: Example



ADM

- ADM (Architecture-Driven Modernization) is addressing the problem of system reverse engineering
- It includes several standards that help on this matter
- The Knowledge Discovery Metamodel (KDM): An intermediate representation for existing software systems that defines common metadata required for deep semantic integration of lifecycle management tools. Based on MOF and XMI
- The Software Measurement Metamodel (SMM): A meta-model for representing measurement information related to software, its operation, and its design.
- The Abstract Syntax Tree Metamodel (ASTM): A complementary modeling specification with respect to KDM, ASTM supports a direct mapping of all code-level software language statements into low-level software models.

MDA vs. ADM – the MDRE process