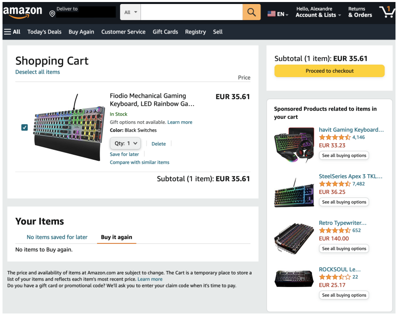
RAMDE - Engenharia Orientada a Requisitos e Modelos
Mestrado em Engenharia de Sistemas Computacionais Críticos
Lecture 01
Introduction

Alexandre Bragança atb@isep.ipp.pt

Dep. de Engenharia Informática – ISEP

2022/2023

Motivation: Requirements, Models and Critical Computing Systems (1/5)



Shopping Cart
Deselect all items
Price

Fiodio Mechanical Gaming Keyboard, LED Rainbow Ga… EUR 35.61
In Stock
Gift options not available. Learn more
Color: Black Switches
Qty: 1 | Delete | Save for later | Compare with similar items

Subtotal (1 item): EUR 35.61

Your Items
No items saved for later | Buy it again
No items to Buy again.

The price and availability of items at Amazon.com are subject to change. The Cart is a temporary place to store a list of your items and reflects each item's most recent price. Learn more
Do you have a gift card or promotional code? We'll ask you to enter your claim code when it's time to pay.

Subtotal (1 item): EUR 35.61
Proceed to checkout

Sponsored Products related to items in your cart

havit Gaming Keyboard…	4,146	EUR 33.23	See all buying options
SteelSeries Apex 3 TKL…	7,482	EUR 36.25	See all buying options
Retro Typewriter…	652	EUR 140.00	See all buying options
ROCKSOUL Le…	22	EUR 25.17	See all buying options

Consider the following functional requirement for an e-commerce system:

"While buying, when my basket is not empty, if I checkout, after paying, the system shall generate a receipt"

Problem: How can I verify if the system includes this functionality?

Motivation: Requirements, Models and Critical Computing Systems (2/5)

Consider the following functional requirement for an e-commerce system:

"While buying, when my basket is not empty, if I checkout, after paying, the system shall generate a receipt"

Problem: How can I verify if the system includes this functionality?

Possible/Usual Solution: Usually, I could **perform a test of the system, simulating this scenario, and check if the result is what is expected**.

However... This will only show that for the specific case (and data) tested, the system performed as expected in the requirement. There are no guarantees that the system will perform as required in all possible cases. And it is unfeasible/impossible to test all the possible scenarios!

Maybe this is acceptable, and not having a receipt is not a critical problem. We can always generate the missing receipts later...

Motivation: Requirements, Models and Critical Computing Systems (3/5)

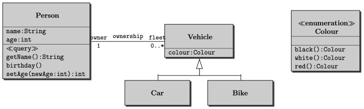
Now consider the following functional requirement for a car:

"While driving, when the driver hits the braking pedal, the braking system should be activated to start decelerating the car"

Problem: If, as previously, I only test for a specific scenario, will this be enough? **Will I consider my system reliable?...**

Maybe we need some guarantee or proof that this requirement will always stand!

Motivation: Requirements, Models and Critical Computing Systems (4/5)



"All cars of a person are black":

```
context   Person
inv:      self.fleet->forAll(v | v.colour = #black)
```

Models, and modeling, allows to construct abstractions of systems with only the specific aspects required for the aim of the model. This reduces the domain of possible scenarios to test, from infinite to something that is manageable.

Models are also a formal way to specify a system and its requirements. This enables the use of mathematical constructs to verify and prove properties of the system (requirements of the system). It is no longer required to test all possible scenarios.

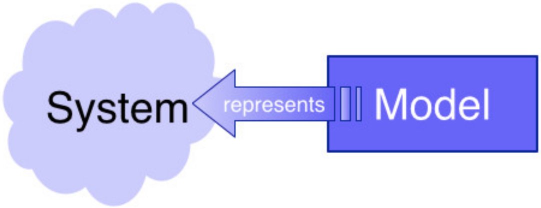
There are several approaches, but it is usual to use some kind of **state machine to model the behavior** of systems and some kind of **logic to model its requirements or constraints**.

Motivation: Requirements, Models and Critical Computing Systems (5/5)

So, thats it, **Model-Driven Engineering (MDE)** solves everything?... Not so far, it helps, **but several issues are still there**, for instance:

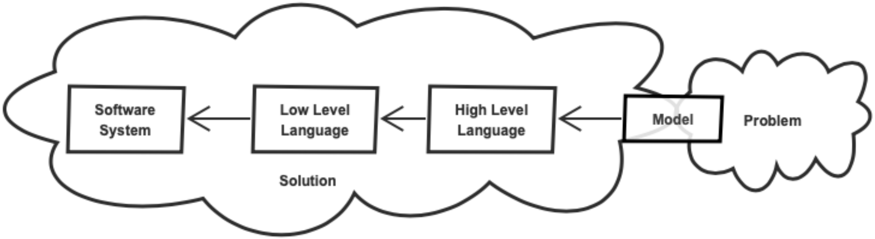
- **Sometimes proving the behavior of some abstraction model is not enough or not possible**. For instance, in the previous example, the behavior of the car can be impacted by the environment (i.e., weather, state of the tires, quality of the road). More specific models and tests may be required for building a reliable system
- **Models must be converted to code**. This can be a manual, semi-automatic, or full-automatic task. Some features of MDE, such **treating code as models**, can help in this context
- **Eliciting and specifying requirements is also a problem**, since stakeholders and software engineers may not understand each other. MDE and **Domain-Specific Languages (DSL)** can help in this context

Base Concepts: What is a Model?...



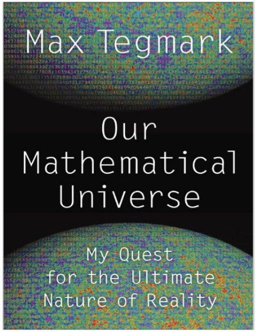
- **Model**: a simplified or partial representation of reality, defined in order to accomplish a task or to reach an agreement.

Base Concepts: A Model in Model-Driven Engineering



- **Model**: a representation of a software system, using high level concepts, enabling a connection between problem and solution spaces.

Base Concepts: Side Thought, What is Reality?


Max Tegmark
Our Mathematical Universe
My Quest for the Ultimate Nature of Reality

"Our Mathematical Universe", Max Tegmark, Alfred A. Knopf, 2014
Max Erik Tegmark is a cosmologist. Tegmark is a professor at the Massachusetts Institute of Technology

*When we look at reality through the equations of physics, we find that they describe patterns and regularities. But to me, mathematics is more than a window on the outside world: in this book, I'm going to argue that **our physical world not only is described by mathematics, but that it is mathematics: a mathematical structure, to be precise.***

Goal

This course addresses **model-driven engineering (MDE) for Critical Systems** by focusing on subjects such as

- metamodeling and domain-specific languages
- modeling standards
- requirements elicitation and analysis
- model-based verification and validation tools, and testing
- safety standards for critical systems

Outcomes

By successfully passing this course, the student must be able to understand the principles of MDE. Specifically, by the end of the course, the student must be able to:

- **Understand the fundamental concepts of MDE**, its role in a critical system's development and safety assessment processes, and apply the acquired knowledge in practical use-cases;
- **Identify and write functional and non-functional requirements for critical systems**, using the best practices and ensuring that they respect the expected quality standards (verifiable, traceable, unambiguous, correct, etc.)
- Show the capability to **understand domain specific standards common in critical systems' development**, and how to **apply them when using an MDE approach**;
- **Acquire hands-on experience by using tools** that support requirement specification and management, system modeling and its safety assessment, using model checking and testing.

Contents

Contents

- Fundamental concepts and principles of MDE, and its specificities when applied to critical systems
- Fundamentals of requirement engineering
- UML and SysML as software and system modeling languages for enabling critical systems MDE
- Formal specification and model checking of critical systems: principles and tools
- Software development standards for critical systems
- Fundamentals of testing and fault analysis, and its usage in MDE
- Case studies / Demonstrations

Organization: 2 modules/sections

- **MDE** (from week 1 to 8) Alexandre Bragança (atb@isep.ipp.pt)
- **Model Verification** (from week 9 to 11) David Pereira (dmp@isep.ipp.pt)

Bibliography and Documents

- **Lecture Handouts** (pdf)
- Sanford Friedenthal, Alan Moore, Rick Steiner, "**A Practical Guide to SysML: The Systems Modeling Language, 2nd Edition**", Elsevier, 2015
- Alexander Kossiakoff, William N. Sweet, Samuel J. Seymour, Steven M. Biemer. "**Systems Engineering Principles and Practice 3rd Edition**", Wiley, 2022
- Klaus Pohl, "**Requirements Engineering: Fundamentals, Principles, and Techniques**", 2010th Edition, Springer
- Christel Baier and Joost-Pieter Katoen, "**Principles of Model Checking**", MIT Press, 2008

Teaching/Learning Tools

- Moodle (`https://moodle.isep.ipp.pt`)
- Git (`https://git-scm.com`)
- Bitbucket (`https://bitbucket.org`)
- Eclipse (`http://www.eclipse.org`)
  - Eclipse SysML plugin (Papyrus - `https://www.eclipse.org/papyrus/`)
- MPS (`https://www.jetbrains.com/mps/`)
  - Mbeddr (`http://mbeddr.com`)
  - Fasten (`https://sites.google.com/site/fastenroot/`)
- Others, as indicated by teacher during classes

Previous knowledge assumed as acquired

- Extended knowledge and **experience in programming in the java language** (LEI: APROG, PPROG, LPROG).
- Software engineering knowledge and experience, mainly **analysis and design best practices with object oriented paradigm and UML** (LEI: ESOFT, EAPLI).
- Basic knowledge and **experience with version control systems** (LEI: LAPR2, LAPR3, LAPR4 and LAPR5).

Classes - Lectures

Lectures

- Lecture classes will introduce and motivate the relevance of MDE in the development process of any critical system.
- In these classes, the theoretical foundations, principles and techniques associated with the process of modeling a system using an MDE approach will be introduced. Special focus will be given to using models for requirements specification
- Model verification and testing will also be addressed in terms of their principles and explored with concrete examples and case studies.
- In each of the topics addressed in these lectures, practical examples and exercises will be given to students

Classes - TP

TP

- TP classes will be dedicated to complement necessary extra technical content of the lecture classes that is fundamental to enable students to use appropriate software tools to validate the knowledge they acquired in practical MDE exercises
- During these classes, there will also be time allocated for the lecturer to mentor the class projects being developed in teams.
- This mentoring will focus on providing early feedback to help the teams to mitigate possible flaws on their models, guiding them on the maintenance of requirements, consistency of the models, and defining verification and testing efforts.
- The project, being iterative and incremental, will contribute to all learning outcomes, since it will lead the students to bridging the gap between what they learn in the lecture classes and the software skill being acquired in the tutorial classes, in a continuous and goal oriented way.

Evaluation

Evaluation during the semester (project, 100%) - no final written exam.

- Students must form teams of 2 elements to work during the semester
- Each group will work within a specific Bitbucket repository
- Each team should propose a working topic for the project (something related to critical systems, like home automation system or automatic warehouse systems)
- All artifacts produced during the project (including documentation) are required to be committed to the repository.
- **Students who are free from attending classes must develop the project outside classes**. They must comply with all the other rules (e.g., deadlines).

Evaluation - Project

Project [100%]

- Project to be developed during the semester and divided into 2 parts
  - Part 1: Model-Driven Engineering (MDE) for critical systems (70%)
  - Part 2: Model Verification (30%)
- **The requirements for each component of each part of the project are presented during classes.**
- Documents for each part will also be available in Moodle
- Specific details regarding the project and its rules will be provide in Moodle.
- Minimum score of 9,5/20,0.
- **Attention: Evaluation is individual**
  - Although it is a group project each student is required to answer for all tasks/components of the project and the work effort should be divided between members.
  - Each part includes a mandatory individual assessment session with each student after submission
  - The performance of the student in the session is taken into account in the evaluation

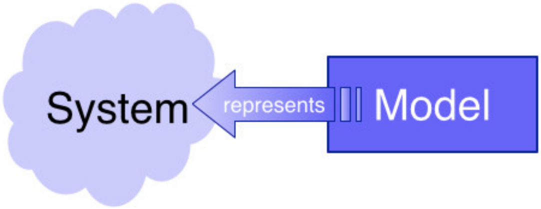
Evaluation Rubric

A performance scale will be available to the students so that they can be aware of their own performance (and it will be used in the evaluation/assessment of each component of each part of the project):

- **0- nothing**. No submission or irrelevant.
- **1- tentative (4/20)**. Do not achieve any of the requirements
- **2- developing/insufficient (8/20)**. Achieves only a subset of the requirements
- **3- acceptable (12/20)**. Achieves the requirements but with major faults
- **4- good (16/20)**. Achieves the requirements with minor faults; justifies options
- **5- excellent (20/20)**. Achieves completely the requirements; justifies options; analyzes alternatives in a quantified and rigorous way (e.g., implementing alternative(s))

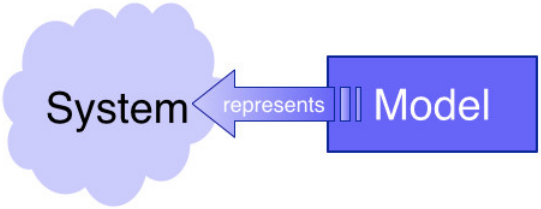
Attention: A grade below 10/20 means you fail.

Motivation: Abstractions and Models



- The human mind continuously re-works reality by applying cognitive processes
- **Abstraction**: capability of finding the commonality in many different observations:
  - generalize specific features of real objects (**generalization**)
  - classify the objects into coherent clusters (**classification**)
  - aggregate objects into more complex ones (**aggregation**)
- **Model**: a simplified or partial representation of reality (i.e., an abstraction), defined in order to accomplish a task or to reach an agreement.

Models: What is a Model?



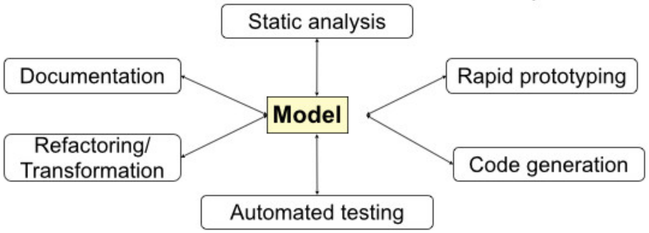
Mapping Feature	A model is based on an original (=system)
Reduction Feature	A model only reflects a (relevant) selection of the original‘s properties
Pragmatic Feature	A model needs to be usable in place of an original with respect to some purpose

Purposes:

- descriptive purposes
- prescriptive purposes

What is Model Engineering?

Model as the **central artifact** of software development



Related terms:

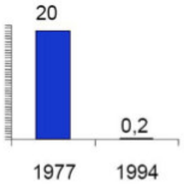
- Model Driven Engineering (MDE),
- Model Driven [Software] Development (MDD/MDSD),
- Model Driven Architecture (MDA)
- Model Integrated Computing (MIC)

Why Model Engineering?

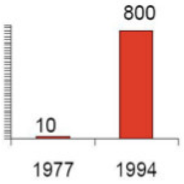
- **Increasing complexity of software**
  - Increasing basic requirements, e.g., adaptable GUIs, security, network capabilities, . . .
  - Complex infrastructures, e.g., operating system APIs, language libraries, application frameworks
- **Software for specific devices**
  - Web browser, mobile phone, navigation system, video player, etc.
- **Technological progress . . .**
  - Integration of different technologies and legacy systems, migration to new technologies
- **. . . leads to problems with software development**
  - Software finished too late
  - Wrong functionality realized
  - Software is poorly documented/commented
  - and can not be further developed, e.g., when the technical environment changes, business model/ requirements change, etc.

Why Model Engineering?

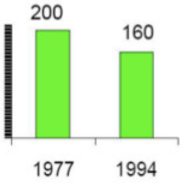
Quality problems in software development



Number of bugs per 1000 LOC



Program size (1000 LOC)



Resulting absolute bug count

Real quality improvements are only possible if the increase in program complexity is **overcompensated** !

(Average values, from Balzert 96)

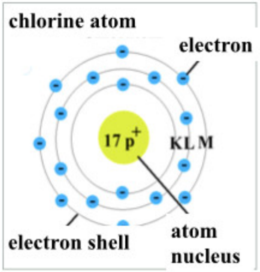
How are we compared to 1994?

Why Model Engineering?

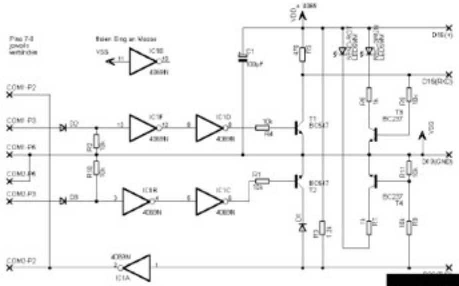
- **Traditional** usage of models in software development
  - **Communication** with customers and users (requirement specification, prototypes)
  - Support for software design, capturing of the **intention**
  - **Task specification** for programming
  - **Code visualization**
- What is the **difference** to Model Engineering?

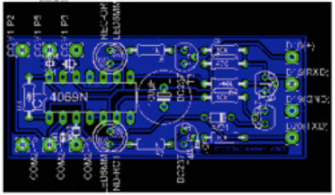
Usage of Models

- Do not apply models as long as you have not checked the underlying **simplifications** and evaluated its **practicability**.
- Never mistake the **model** for the **reality**.
  - Attention: abstraction, abbreviation, approximation, visualization, . . .



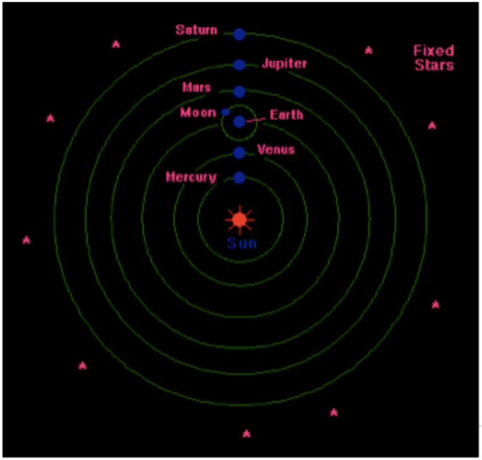
Constructive models (Example: Electrical Engineering)





Declarative models (Example: Astronomy)

Heliocentric model by Kopernikus

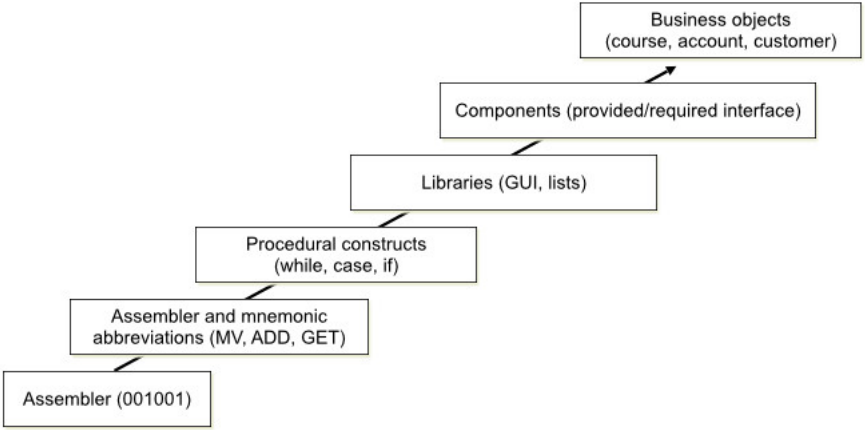


Application area of modeling

- **Models as drafts**
  - Communication of ideas and alternatives
  - Objective: modeling per se
- **Models as guidelines**
  - Design decisions are documented
  - Objective: instructions for implementation
- **Models as programs**
  - Applications are generated automatically
  - Objective: models are source code and vice versa

Increasing abstraction in software development

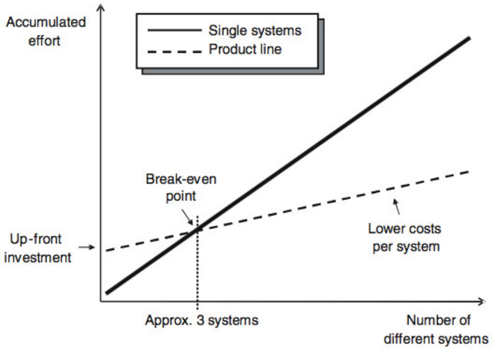
The **used artifacts of software development** slowly converge to the concepts of the **application area**. We have also being increasing how we are able to **reuse constructs in our software solutions**.
Such concepts can be **realized as domain-specific languages (DSL)**


Business objects (course, account, customer)
Components (provided/required interface)
Libraries (GUI, lists)
Procedural constructs (while, case, if)
Assembler and mnemonic abbreviations (MV, ADD, GET)
Assembler (001001)

Industrial Reuse -> Software Product Lines

A software product line is a set of software-intensive systems ***sharing a common, managed set of features*** *that satisfy the specific needs of a* ***particular market segment or mission*** *and that are developed from a* ***common set of core assets in a prescribed way***[1].

Figure: Economics of software product line engineering[2]



[1] From "Software Product Lines: Practices and Patterns", Paul Clements, Linda Northrop, Addison-Wesley, 2001

[2] From "Software Product Lines in Action", Frank van der Linden et al, Springer, 2007

Fundamentals of Software Product Lines

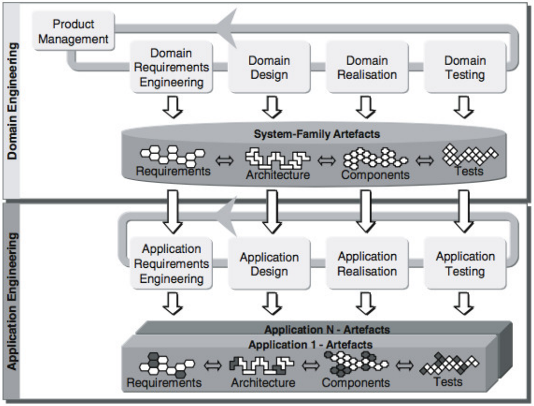


Figure: The two-life-cycle model of software product line engineering[3]

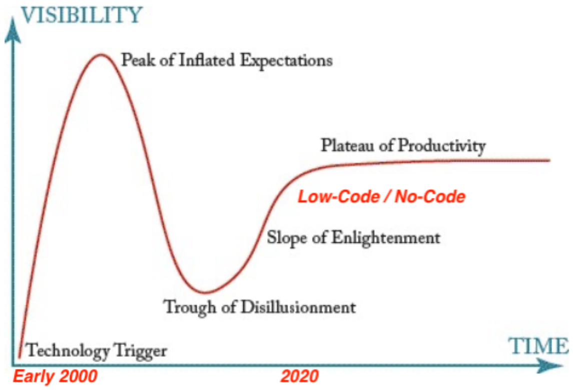
[3] From "Software Product Lines in Action", Frank van der Linden et al, Springer, 2007

Software Product Lines Fundamental Principles

- **Variability management**: individual systems are considered as variations of a common theme. This variability is made explicit and must be systematically managed.
- **Business-centric**: software product line engineering aims at thoroughly connecting the engineering of the product line with the long-term strategy of the business.
- **Architecture-centric**: the technical side of the software must be developed in a way that allows taking advantage of similarities among the individual systems.
- **Two-life-cycle approach**: the individual systems are developed based on a software platform. These products – as well as the platform – must be engineered and have their individual life-cycles.

DSL, MDE, SPL: Are Not These Old Solutions from 2000?

Figure: Technology Hype Cycle



Low-Code / No-Code seems to be the democratization of the approach and the really **enabler of MDE productivity for the masses...**

DSL, MDE, SPL: Who is using these approaches globally?

The **Software Product Line Conference**[4] has nominated a series of systems for its hall of fame. Included in this list are companies such as:

- Boing
- Bosch
- Ericsson
- General Motors
- HP
- Lucent
- Nokia
- Philips
- Siemens
- Toshiba
- US Army

Note: These are all systems that have been documented and presented at the SPLC conference.

[4] `http://splc.net/hall-of-fame/`

DSL, MDE, SPL: What tools to use?

DSLs, MDE and SPLs do not require specific tools. It is possible to adopt any of these approaches **using general and widely used software engineering tools**.

However, there are also specific tools oriented to support such approaches. Some of the major tools are: [5]

- EMF - Eclipse Modeling Framework (`https://www.eclipse.org/modeling/`). The reference tool solution for MDE and DSL development. Several tools built around EMF:
    - ATL - ATL Transformation Language (`http://www.eclipse.org/atl/`)
    - Acceleo (`http://www.eclipse.org/acceleo/`)
    - Xtext (`http://www.eclipse.org/Xtext/`)
    - Sirius (`https://www.eclipse.org/sirius/`)
- MPS - Meta Programming System (`https://www.jetbrains.com/mps/`)
- Modeling SDK for Visual Studio - Domain-Specific Languages (`https://docs.microsoft.com/en-us/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages?view=vs-2017`)

[5] This list is not complete.

What's Next?

Assignment 01

First assignment of MDE - Modeling and the Eclipse EMF

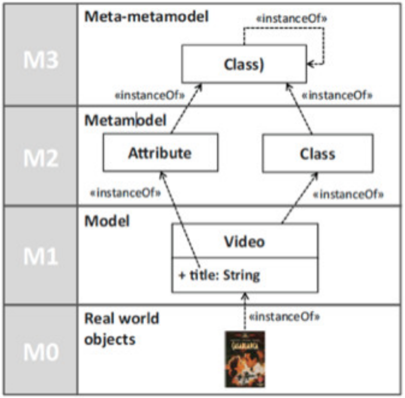
Start in the TP Class (and complete outside classes)
Overview

- Install Eclipse and the EMF - Eclipse Modeling Framework (`https://www.eclipse.org/modeling/`)
- Create a metamodel for state machines
- Use the previous metamodel to create state machine models
- Explore model validation and queries with OCL
- Commit the code to your repository. Add a readme file in Markdown that describes and explains the assignment.

Preamble: Models and Metamodels (1/2)

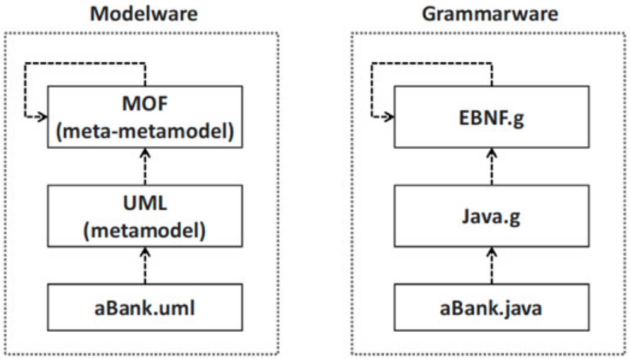
In MDE:

- A **Model** is defined following the "constraints/rules" of its **Metamodel**
- **Metamodel** = yet another abstraction, highlighting properties of the model itself
- A metamodel is also a model
- Every model has its metamodel
- Metamodels can be used for:
  - defining new languages
  - defining new properties or features of existing information (metadata)

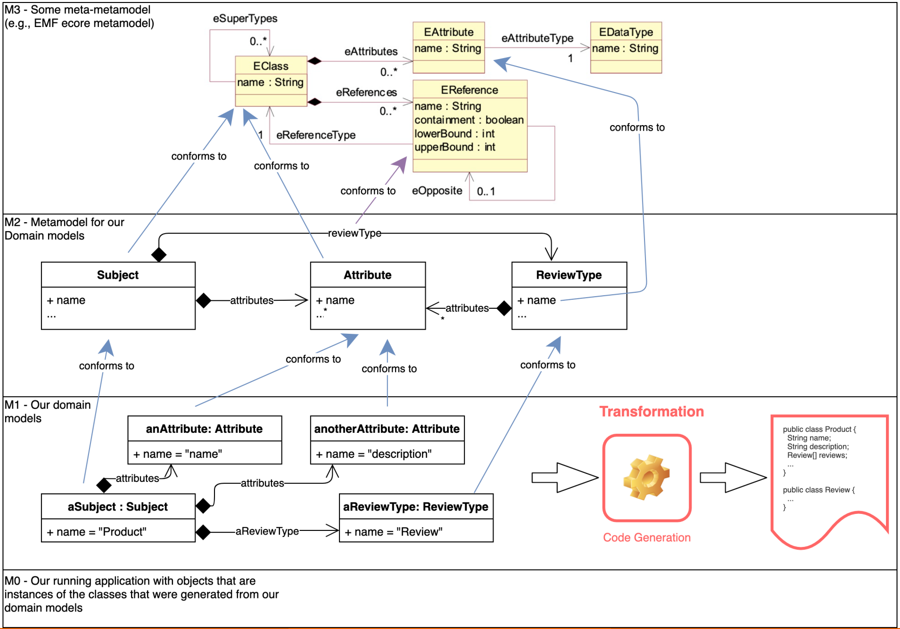


Preamble: Models and Metamodels (2/2)

- MDE concepts are "similar" to concepts of programming languages and grammars
- The relation between metamodel and models is similar to the relation between a program in a programming language and the grammar of that programming language



The Modeling Levels



```
public class Product {
  String name;
  String description;
  Review[] reviews;
  ...
}

public class Review {
  ...
}
```

Meta-Metamodel: Ecore - EMF Metamodel: Core Concepts

To model the M2 model we use the M3 modeling concepts of Ecore (the EMF Metamodel).

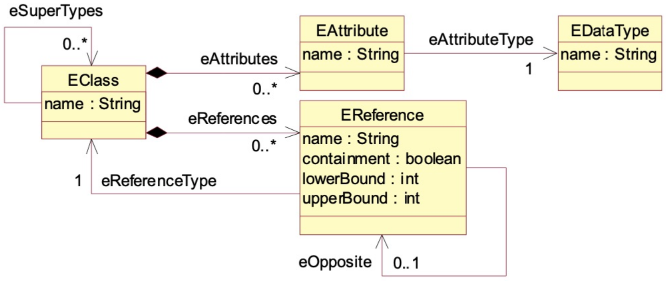


Figure: Excerpt of the Ecore Metamodel

- These are the main concepts used in Ecore to specify all possible models!
- Ecore is, in fact, an implementation of the EMOF metamodel from OMG (EMOF = Essential Meta Object Facility)
- For a full diagram of the Ecore Metamodel see
`http://www.kermeta.org/docs/org.kermeta.ecore.documentation/build/html.chunked/Ecore-MDK/ch02.html`

Metamodel of UML State Machines - Maybe too complex?...

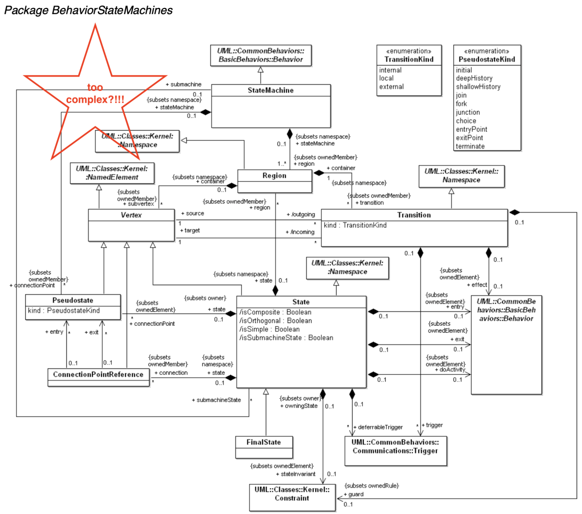

Package BehaviorStateMachines
too complex?!!!
UML::CommonBehaviors::BasicBehaviors::Behavior
StateMachine
«enumeration» TransitionKind
internal
local
external
«enumeration» PseudostateKind
initial
deepHistory
shallowHistory
join
fork
junction
choice
entryPoint
exitPoint
terminate
+ submachine
{subsets namespace} + stateMachine
UML::Classes::Kernel::Namespace
UML::Classes::Kernel::NamedElement
Region
{subsets namespace} + container
{subsets ownedMember} + region
{subsets ownedMember} + subvertex
Vertex
+ source
+ target
+ /outgoing
+ /incoming
Transition
kind : TransitionKind
{subsets ownedMember} + transition
+ container
Pseudostate
kind : PseudostateKind
{subsets ownedMember} + connectionPoint
+ connectionPoint
+ entry
+ exit
ConnectionPointReference
{subsets ownedMember} + connection
{subsets namespace} + state
{subsets owner} + state
State
/isComposite : Boolean
/isOrthogonal : Boolean
/isSimple : Boolean
/isSubmachineState : Boolean
+ submachineState
{subsets owner} + owningState
FinalState
{subsets ownedElement} + effect
{subsets ownedElement} + entry
{subsets ownedElement} + exit
{subsets ownedElement} + doActivity
UML::CommonBehaviors::BasicBehaviors::Behavior
+ deferrableTrigger
+ trigger
UML::CommonBehaviors::Communications::Trigger
{subsets ownedElement} + stateInvariant
UML::Classes::Kernel::Constraint
{subsets ownedRule} + guard

Figure: Excerpt of the UML Metamodel (M2) - See `https://www.omg.org/spec/UML/2.5.1/About-UML`

Solution: Using Eclipse EMF

We can use Eclipse EMF to create our own specific metamodel for state machines...

Assignment 1: MDE - Modeling and the Eclipse EMF

Requirements

- Base your solution in what is described in: `https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines?hsLang=de`
- Create a metamodel for Moore machines and another for Mealy machines
- Use the metamodels to create state machine models
- Explore model validation and queries with OCL
- Commit the code into your repository. Add to the readme file (in Markdown) a small report for the assignment.

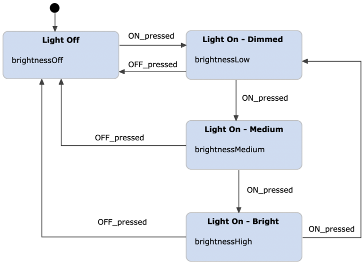


Figure: Simple State Machine