The “Theory” and Practice of Modeling Language Design (for Model-Based Software Engineering)

Bran Selić

Malina Software Corp.

Zeligsoft (2009) Ltd.
Simula Research Labs, Norway
University of Toronto,
Carleton University

selic@acm.org

© Copyright Malina Software

Perhaps, a more appropriate title for this tutorial is:

"The Modeling Language Designer's Grimoire"??

Grimoire: *noun* a manual of black magic (for invoking spirits and demons)

© Copyright Malina Software

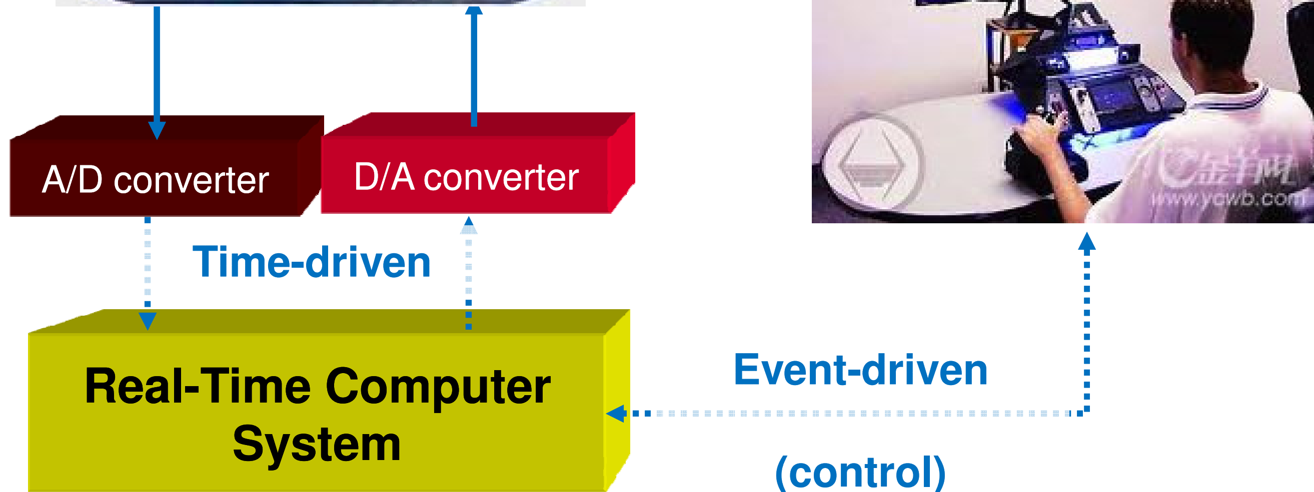
Tutorial Outline

- On Models and Model-Based Software Engineering
- The Key Dimensions of Modeling Language Design
- Defining a Modeling Language
- Case Study: UML
- Language Refinement: UML Profiles
- Model Transformations

© Copyright Malina Software

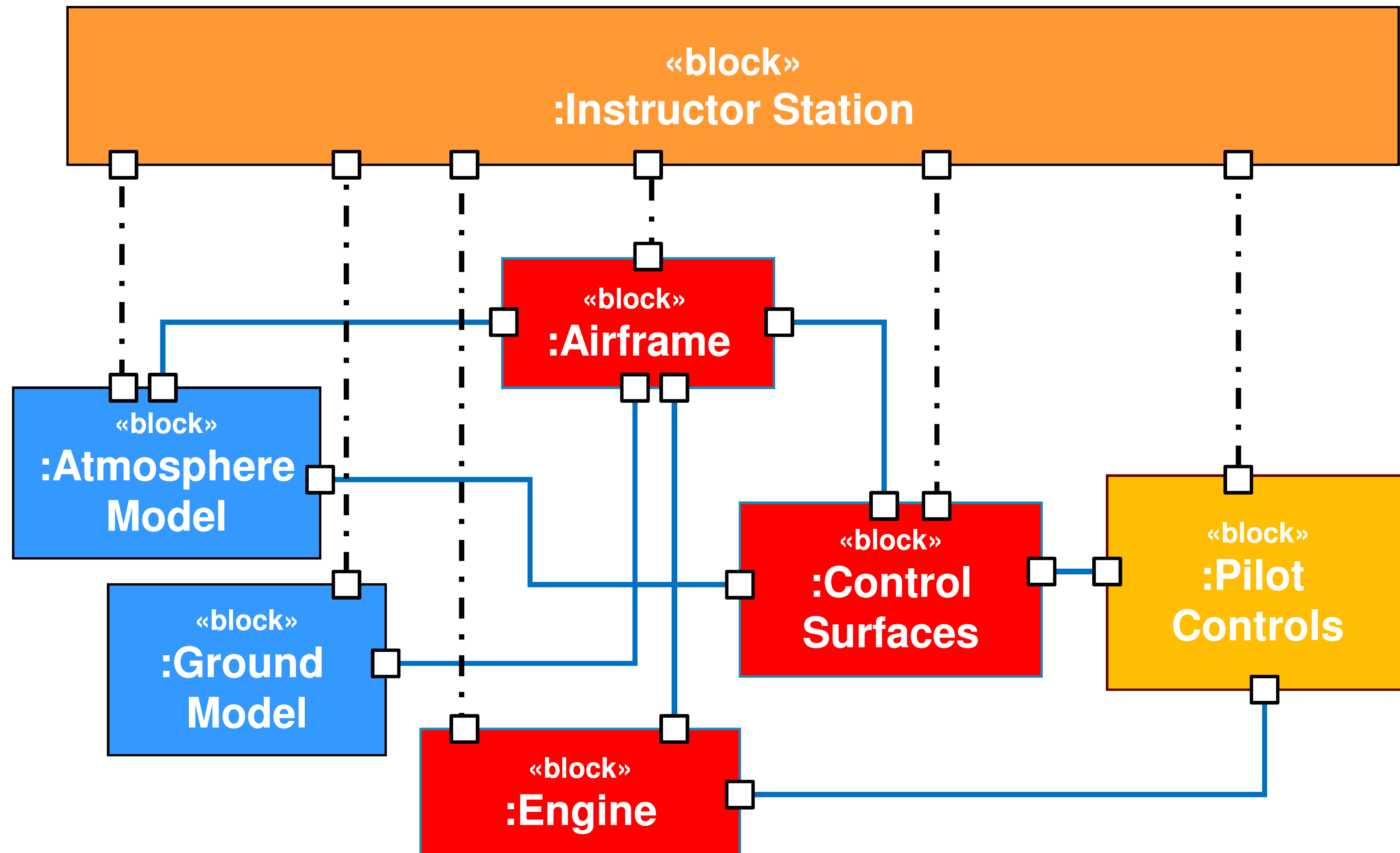
Example System: Aircraft Simulator Software

- **Typical embedded software system**
  - Software that must interact with the physical world in a timely fashion



© Copyright Malina Software

The Logical Structure of the Software*

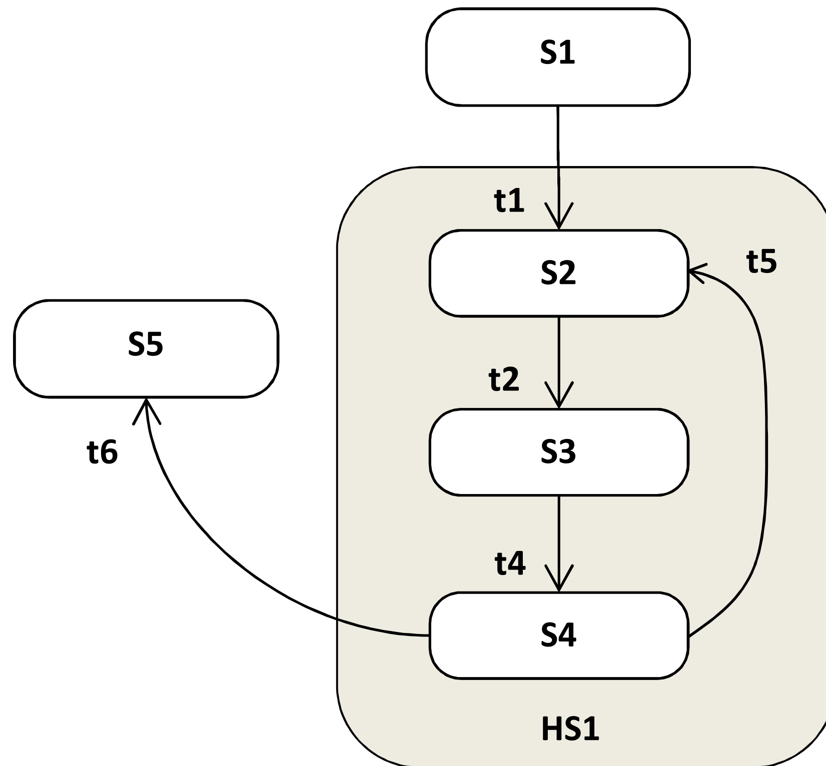


*(simplified representation)

© Copyright Malina Software

Behaviour as Specified

Control behaviour (event driven)



Physical simulation (time driven)

$$vx(t) = vx(t-1) + \Delta vx(t)$$

$$vy(t) = vy(t-1) + \Delta vy(t)$$

$$vz(t) = vz(t-1) + \Delta vz(t)$$

$$\Delta vx(t) = (x(t) - x(t-1)) / \Delta t$$

$$\Delta vy(t) = (y(t) - y(t-1)) / \Delta t$$

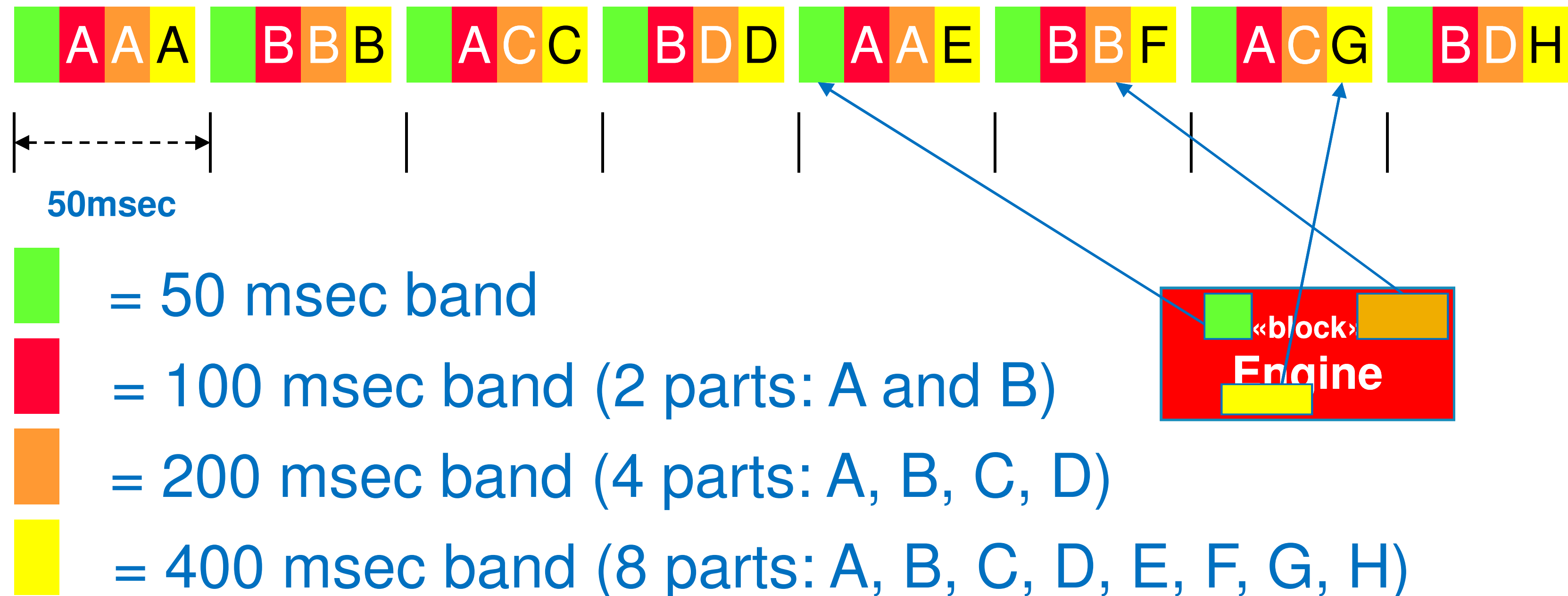
$$\Delta vz(t) = (z(t) - z(t-1)) / \Delta t$$

But, the implementation code corresponding to the behaviour and structure looks very different

© Copyright Malina Software

Simulator Software: As Implemented

- Behaviour sliced according to rate of change
- Structural relationships represented by references in code



The semantic gap between the way we think about the problem/solution and its realization in software adds significant complexity and poses major impediments to design analysis and software maintenance

© Copyright Malina Software

On Types of Complexity

- Essential complexity
  - Immanent to the problem
  - ⇒ *Cannot be eliminated by technology or technique*
  - e.g., solving the "traveling salesman" problem
- Accidental complexity
  - Due to technology or methods chosen to solve the problem
  - e.g., building a house without power tools
  - ⇒ *Complex problems require correspondingly powerful tool*

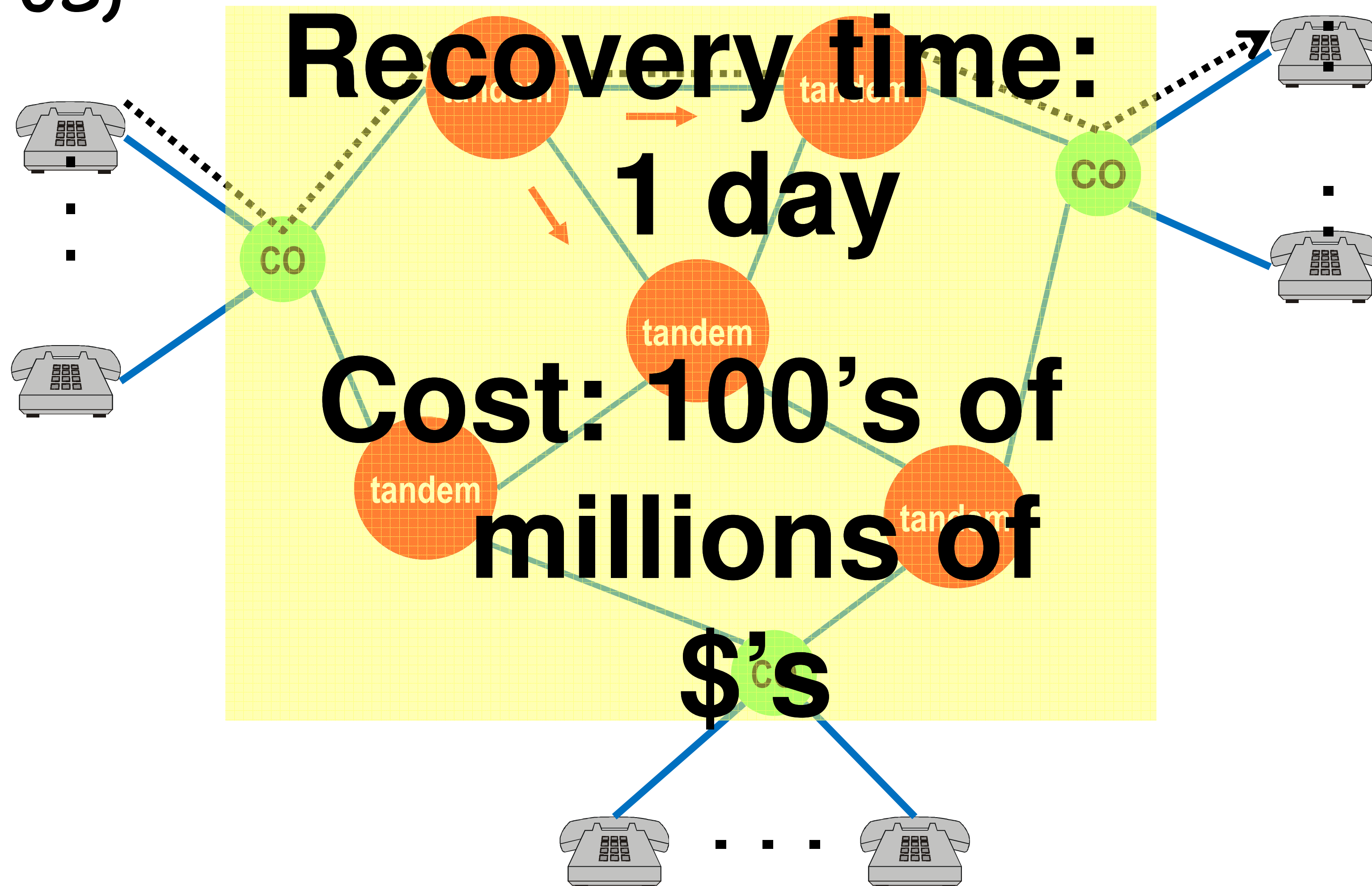
The best we can do is to try and minimize accidental complexity!

Thesis: "Modern" mainstream programming languages (e.g., C++, Java) abound in accidental complexity and are machine-centric

© Copyright Malina Software

The Case of the Tandem Switches Tango…

- 1990: AT&T Long Distance Network (Northeastern US)



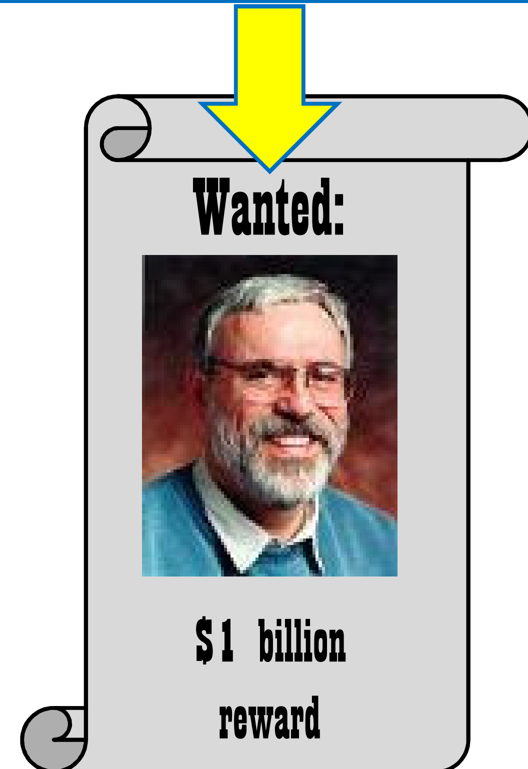
© Copyright Malina Software

The Hidden Culprit

- The (missing) "break" that broke it

```
. . .;
switch (...) {
      case a : ...;
            break;
      case b :...;
            break;
            . . .
      case m : ...;
      case n : ...;
            . . .
};
```

...and, it's all HIS fault!



Ooops! Forgot the "break"...

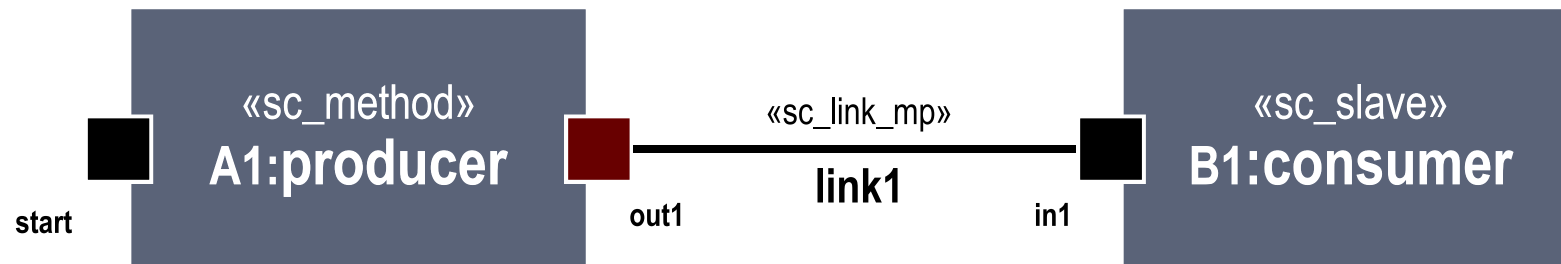
© Copyright Malina Software

A Bit of Modern Software…

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
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);}};
```


© Copyright Malina Software

…and its (UML 2) Model



© Copyright Malina Software

Automatic Code Generation from Models

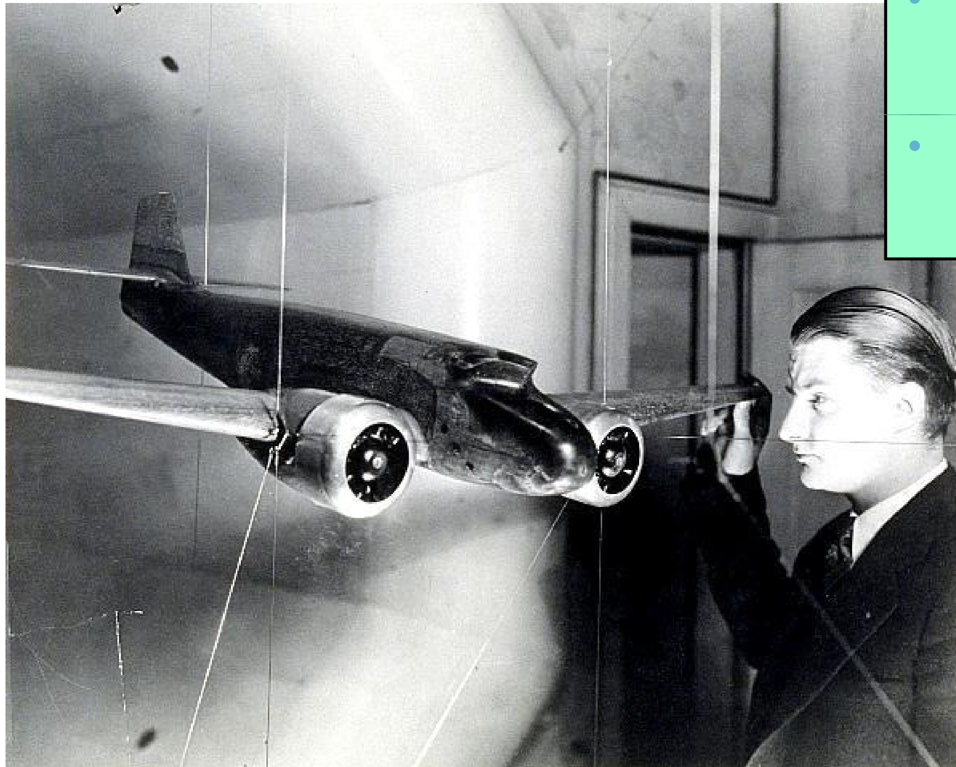
```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
}
```

```
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
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);}};
```

© Copyright Malina Software

Engineering Models

- **ENGINEERING MODEL:** *A selective representation of some system that captures accurately and concisely all of its essential properties of interest for a given set of concerns*



- We don't see everything at once
- What we do see is adjusted to human understanding

© Copyright Malina Software

Why Do Engineers Build Models?

- To <u>understand</u>
  - ...the interesting characteristics of an existing or desired (complex) system and its environment
- To <u>predict</u>
  - ...the interesting characteristics of the system by analysing its model(s)
- To <u>communicate</u>
  - ...their understanding and design intent (to others and to oneself!)
- To <u>specify</u>
  - ...the implementation of the system (models as blueprints)

© Copyright Malina Software

Characteristics of Useful Engineering Models

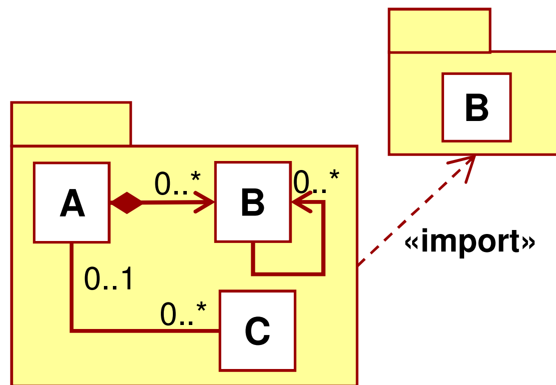
- **Purposeful:**
  - Constructed to address a specific set of concerns/audience
- **Abstract**
  - Emphasize important aspects while removing irrelevant ones
- **Understandable**
  - Expressed in a form that is readily understood by observers
- **Accurate**
  - Faithfully represents the modeled system
- **Predictive**
  - Can be used to answer questions about the modeled system
- **Cost effective**
  - Should be much cheaper and faster to construct than actual system

To be useful, engineering models must satisfy at least these characteristics!

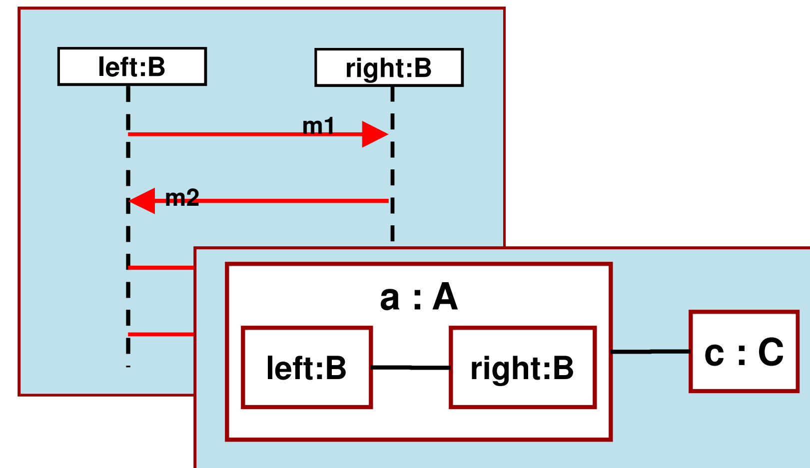
© Copyright Malina Software

What's a Software Model?

- **SOFTWARE MODEL:** An engineering model of a software system from *one or more viewpoints* specified using one or more modeling languages
  - E.g.:



Structural view



Execution view

© Copyright Malina Software

What's a Modeling Language?

- **(SOFTWARE SYSTEM) MODELING LANGUAGE:** A computer language intended for constructing models of software programs and the contexts in which they operate
  - Can range from very abstract to complete ("the map is the territory")

© Copyright Malina Software

Modeling vs Programming Languages

- **The primary purpose and focus of programming languages is implementation**
  - The ultimate form of specification
  - ⇒ Implementation requires total precision and "full" detail
  - ⇒ Takes precedence over understandability
- **The purpose of modeling also includes communication, prediction, and understanding**
  - These generally require omission of "irrelevant" detail
- **However...**

© Copyright Malina Software

The Unique Nature of Software

- Software is a unique in that a program and its model share the same medium – the computer

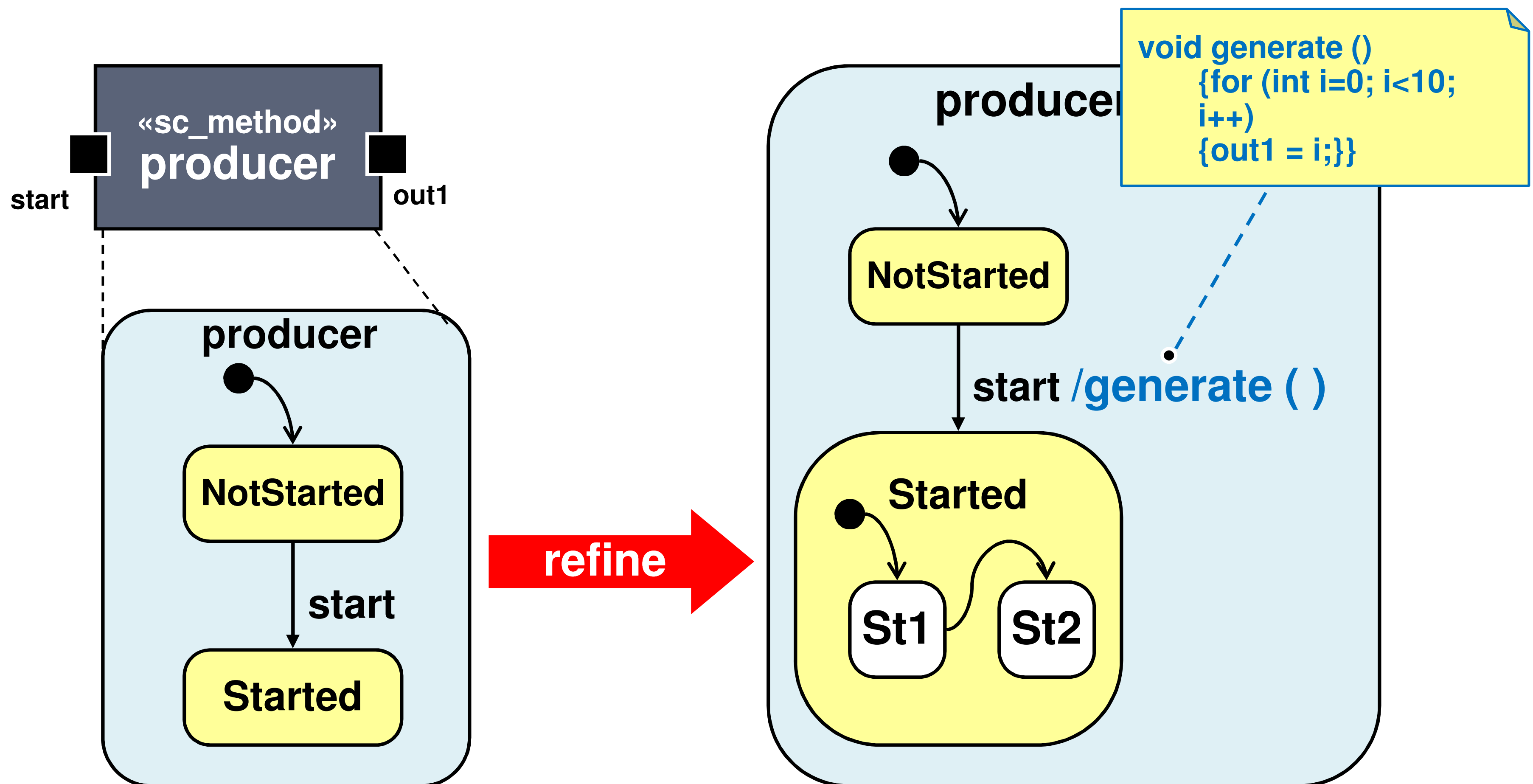
⇒ The two can be formally linked to each other

⇒ This formal linkage can be realized by automated transformations implemented on a computer

© Copyright Malina Software

Modern MBSE Development Style

- Models can be refined continuously until the application is fully specified ⇒ in the extreme case ***the model can become the system that it was modeling!***



© Copyright Malina Software

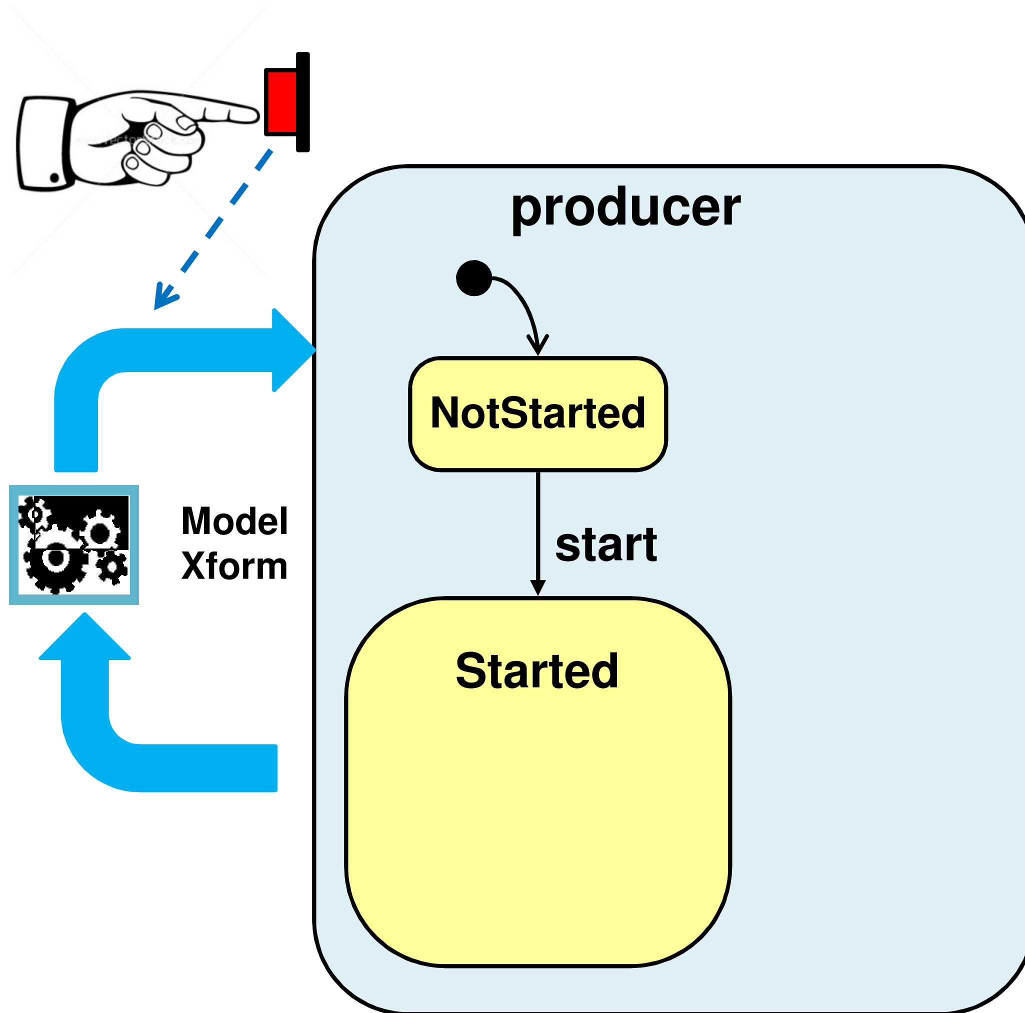
But, if the Model is the System...

- ...are we not losing the key abstraction characteristic of models?

- *The computer offers a uniquely capable abstraction device:*

  Software can be represented from any desired viewpoint and at any desired level of abstraction

  The abstraction resides within the model and can be extracted automatically



© Copyright Malina Software

A Unique Feature of Software

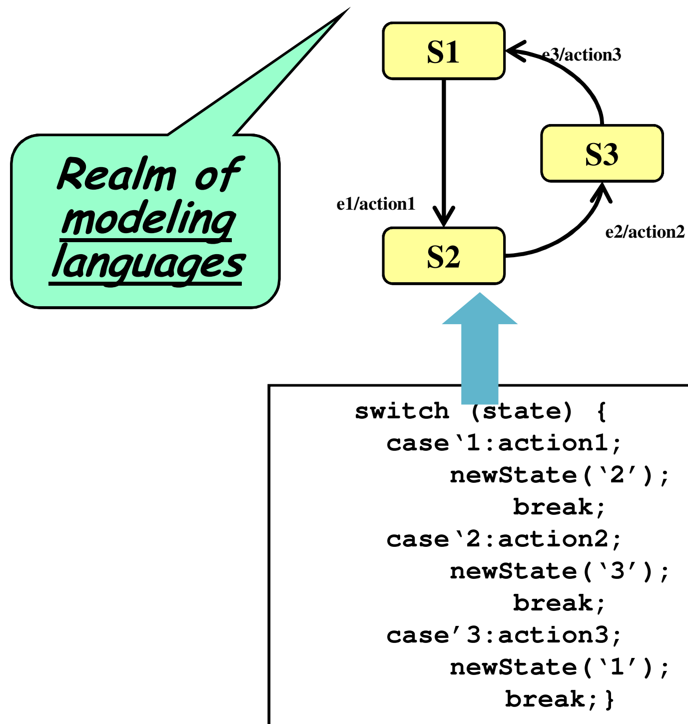
Software has the unique property that it allows us to directly evolve models into implementations <u>without fundamental discontinuities in the expertise, tools, or methods</u>!

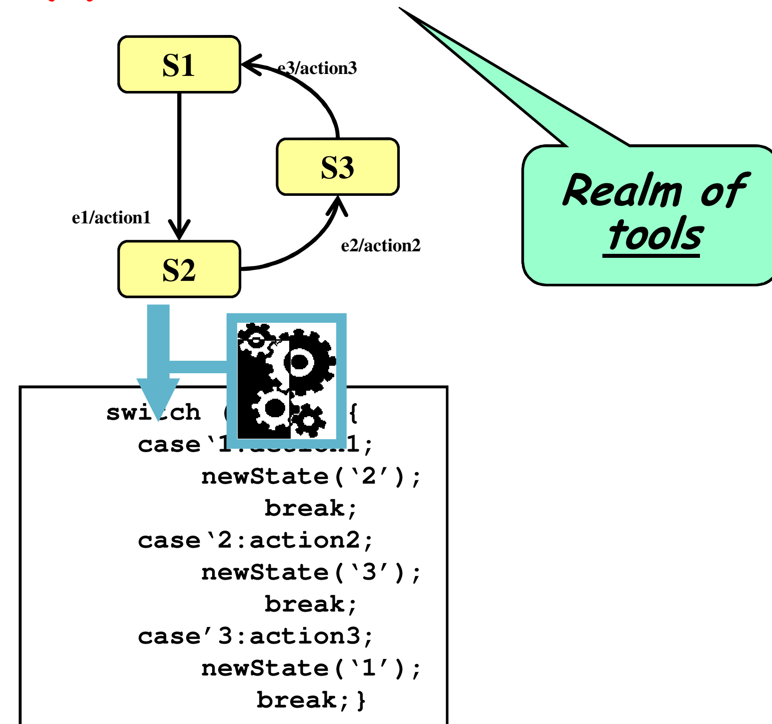
⇒ High probability that key design decisions will be preserved in the implementation and that the results of prior analyses will be valid

© Copyright Malina Software

The Model-Based Engineering (MBE) Approach

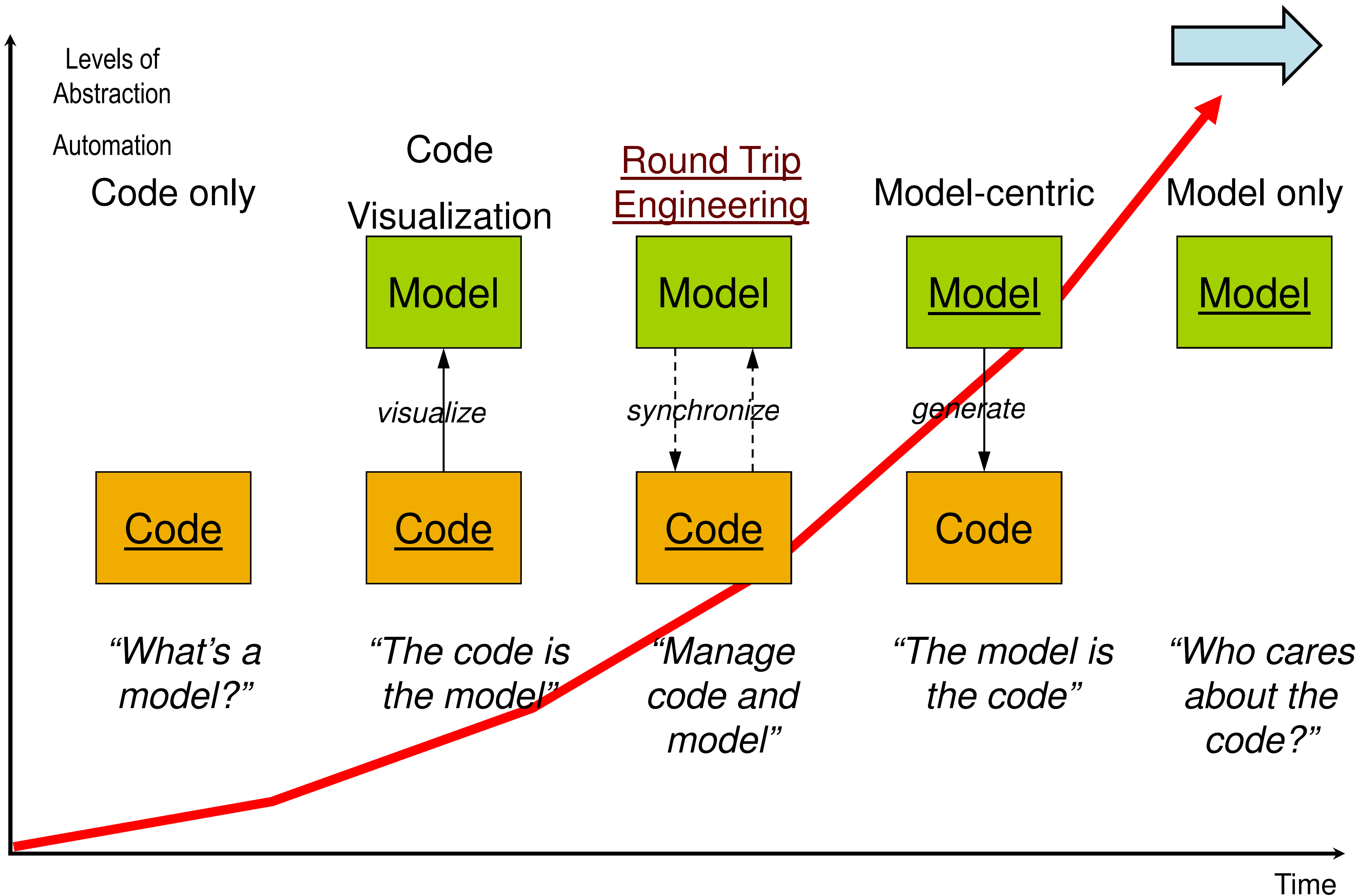
- An approach to system and software development in which software models play an indispensable role
- Based on two time-proven ideas:





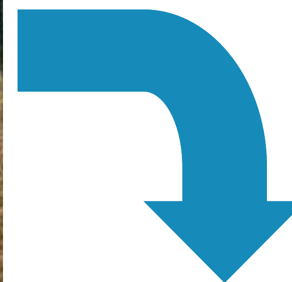
© Copyright Malina Software

Styles of MBSE

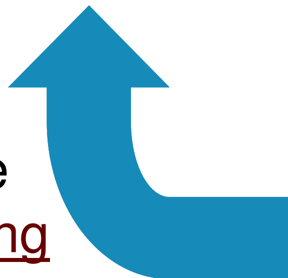


© Copyright Malina Software

Roundtrip Engineering



Implementation transformation



Reverse engineering

NB: Slide idea borrowed from an **itemis AG** presentation

© Copyright Malina Software

Automatic Code Generation

- A form of model transformation (model to text)
  - To a lower level of abstraction
- State of the art:
  - All development done via the model (i.e., no modifications of generated code)
  - Size: Systems equivalent to ~ 10 MLoC
  - Scalability: teams involving hundreds of developers
  - Performance: within ±5-15% of equivalent manually coded system

© Copyright Malina Software

Major Telecom Equipment Manufacturer

- MBE technologies used
  - UML, Rational Technical Developer, RUP
- Example 1: Radio Base Station
  - 2 Million lines of C++ code (87% generated by tools)
  - 150 developers
- Example 2: Network Controller
  - 4.5 Million lines of C++ code (80% generated by tools)
  - 200 developers

Benefits

80% fewer bugs

30% productivity increase

© Copyright Malina Software

..and a Few Extreme Cases

- **Major Equipment Manufacturer 1:**
  - Code production rate went from 40 LoC/day to 250 Loc/day (>600% improvement)
- **Major Equipment Manufacturer 2:**
  - Code production rate went from 200 LoC/week to 950 Loc/week (~500% improvement)
  - 6-person team developed 120 KLoC system in 21.5 weeks compared to planned 40 weeks (~100% improvement)
  - Fault density (per line of code) reduced 17-fold (1700%)

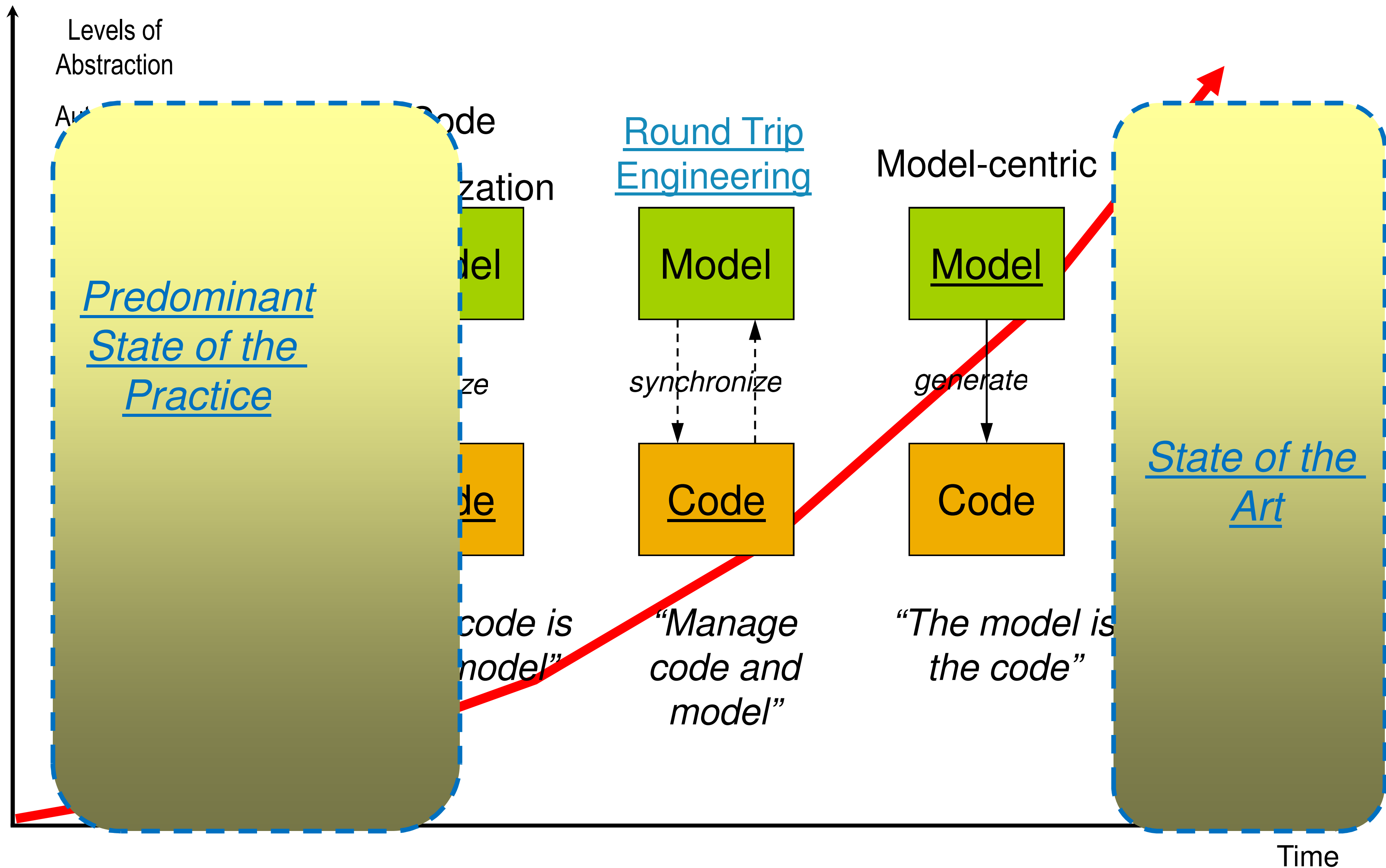
© Copyright Malina Software

Sampling of Successful MBE Products

Automated doors, Base Station, Billing (In Telephone Switches), Broadband Access, Gateway, Camera, Car Audio, Convertible roof controller, Control Systems, DSL, Elevators, Embedded Control, GPS, Engine Monitoring, Entertainment, Fault Management, Military Data/Voice Communications, Missile Systems, Executable Architecture (Simulation), DNA Sequencing, Industrial Laser Control, Karaoke, Media Gateway, Modeling Of Software Architectures, Medical Devices, Military And Aerospace, Mobile Phone (GSM/3G), Modem, Automated Concrete Mixing Factory, Private Branch Exchange (PBX), Operations And Maintenance, Optical Switching, Industrial Robot, Phone, Radio Network Controller, Routing, Operational Logic, Security and fire monitoring systems, Surgical Robot, Surveillance Systems, Testing And Instrumentation Equipment, Train Control, Train to Signal box Communications, Voice Over IP, Wafer Processing, Wireless Phone

© Copyright Malina Software

Where We Stand at Present



© Copyright Malina Software

Q: If this stuff is so good, why isn't everybody doing it?

© Copyright Malina Software

Root Causes of Low Adoption Rate

- **Social/Cultural issues**
  - Conservative mindset of many practitioners
- **Economic factors**
  - Retraining
  - Retooling
  - Reorganizing development
  - Integration with legacy
- **Technical issues**
  - Immaturity of tools
  - Lack of systematic theoretical underpinnings

© Copyright Malina Software

Tutorial Outline

- On Models and Model-Based Software Engineering
- The Key Dimensions of Modeling Language Design
- Defining a Modeling Language
- Case Study: UML
- Language Refinement: UML Profiles
- Model Transformations

© Copyright Malina Software

Current "Hot" Topic of Controversy

"Surely it is better to design a small language that is highly expressive, because it focuses on a specific narrow domain, as opposed to a large and cumbersome language that is not particularly well-suited to any particular domain?"

This is a high-level design issue, but not the only one by any means...

© Copyright Malina Software

Key Modeling Language Design Dimensions

- Scope?
  - Broad (general) or narrow (domain specific)?
- Formal or informal? (executable?)
- Semantics?
  - Static: Ontology (concepts and relationships)?
  - Dynamic: Model of computation (how do things happen?)
- New language or an extension or refinement of an existing one?
- Concrete syntax?
  - Graphical? Textual? Heterogeneous?
- Extensible?
- Method of language specification?

© Copyright Malina Software

Scope: How General/Specialized?

- **Generality often comes at the expense of expressiveness**
  - Expressiveness: the ability to specify *concisely* yet *accurately* a desired system or property
  - Example:
    - UML does not have a concept that specifies mutual exclusion devices (e.g. semaphore) ⇒ to represent such a concept in our model, we would need to combine a number of general UML concepts in a particular way (e.g., classes, constraints, interactions)
  - ...which may(?) be precise, but not very concise
- **It also comes at the cost of detail that is necessary to:**
  - Execute models
  - Generate complete implementations

© Copyright Malina Software

Specialization: Inevitable Trend

- **Constant branching of application domains into ever-more specialized sub-domains**
  - As our knowledge and experience increase, domain concepts become more and more refined
    - E.g., simple concept of computer memory → ROM, RAM, DRAM, cache, virtual memory, persistent memory, etc.
- **One of the core principles of MBE is raising the level of abstraction of specifications to move them closer to the problem domain**

- **This seems to imply that domain-specific languages are invariably the preferred solution**
- **But, there are some serious hurdles here...**

© Copyright Malina Software

The Case of Programming Languages

- Literally hundreds of domain-specific programming languages have been defined over the past 50 years
  - Fortran: for scientific applications
  - COBOL for "data processing" applications
  - Lisp for AI applications
  - etc.
- Some relevant trends
  - Many of the original languages are still around
  - More often than not, highly-specialized domains still tend to use general-purpose languages with specialized domain-specific program libraries and frameworks instead of domain-specific programming languages
  - In fact, the trend towards defining new domain-specific programming languages seems to be diminishing
- Why is this happening?

© Copyright Malina Software

Success* Criteria for a Language (1)

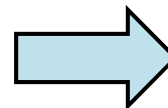
- Technical validity: absence of major design flaws and constraints
  - Ease of writing correct programs
- Expressiveness
- Simplicity: absence of gratuitous/accidental complexity
  - Ease of learning
- Run-time efficiency: speed and (memory) space
- Familiarity: proximity to widely-available skills
  - E.g., syntax

* "Success" ⇒ language is adopted by a substantive development community and used with good effect for real-world applications

© Copyright Malina Software

Success Criteria for a Language (2)

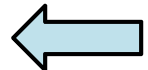
- **Language Support & Infrastructure**:
  - Availability of necessary tooling
  - Effectiveness of tools (reliability, quality, usability, customizability, interworking ability)
  - Availability of skilled practitioners
  - Availability of teaching material and training courses
  - Availability of program libraries
  - Capacity for evolution and maintenance (e.g., standardization)



© Copyright Malina Software

Basic Tooling Capabilities

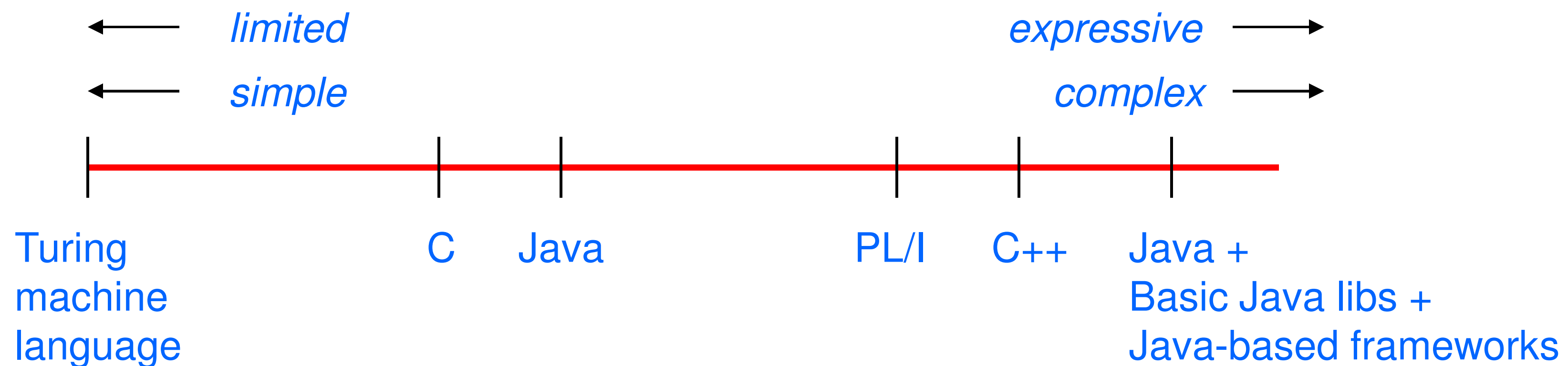
- **Essential**
  - Model Authoring
  - Model validation (syntax, semantics)
  - Model export/import
  - Document generation
  - Version management
  - Model compare/merge
- **Practical**
  - Code generation
  - Model simulation/debug/trace
  - Model transformation
  - Model review/inspection
  - Collaborative development support
  - Language customization support
  - Test generation
  - Test execution
  - Traceability



© Copyright Malina Software

Design Challenge: Simplicity (Scope)

- ***How complex (simple) should a language be to make it effective?***



- **The art of computer language design lies in finding the right balance between expressiveness and simplicity**
  - Need to minimize accidental complexity while recognizing and respecting essential complexity
  - Small languages solve small problems
  - No successful language has gotten smaller

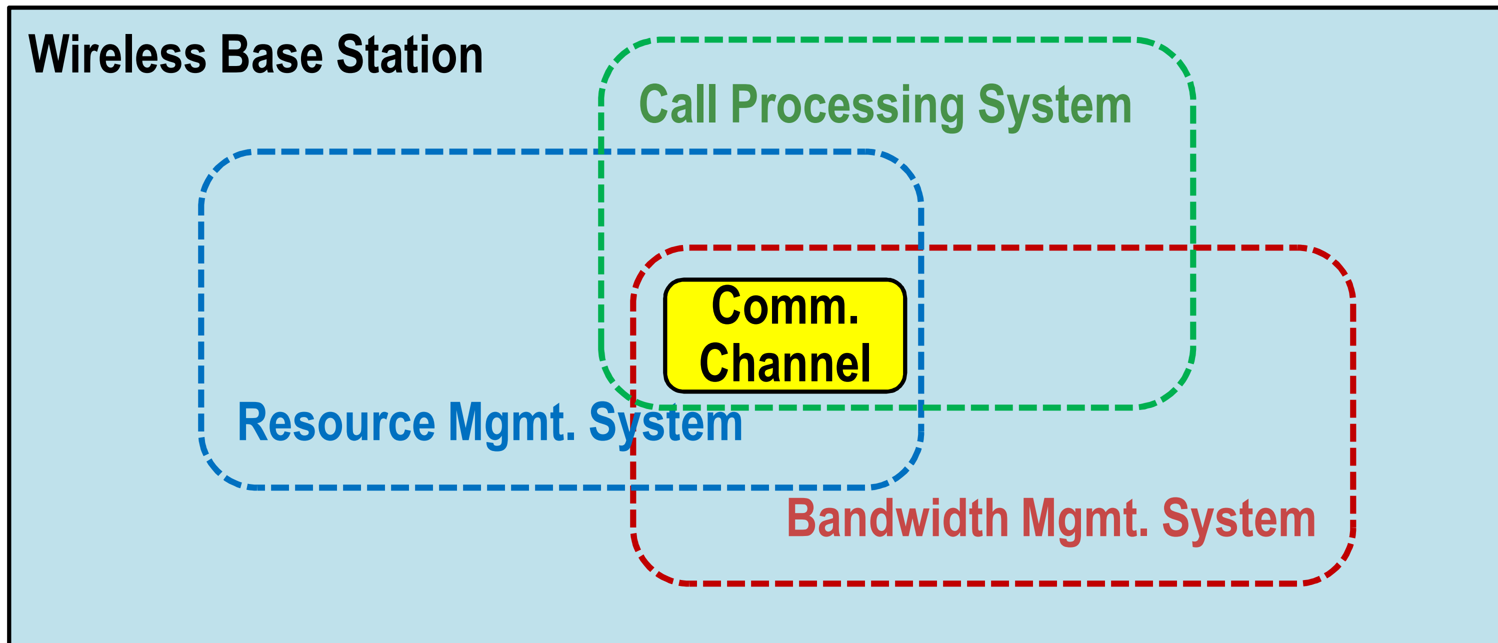
© Copyright Malina Software

Design Challenge: Scope

- Real-world systems often involve multiple heterogeneous domains
  - Each with its own ontology and semantic
- Example: wireless telecom system
  - Basic bandwidth management
  - Equipment and resource management
  - Operations, administration, and systems management
  - Accounting (customer resource usage)
  - Computing platform (OS, supporting services)

© Copyright Malina Software

The Fragmentation Problem

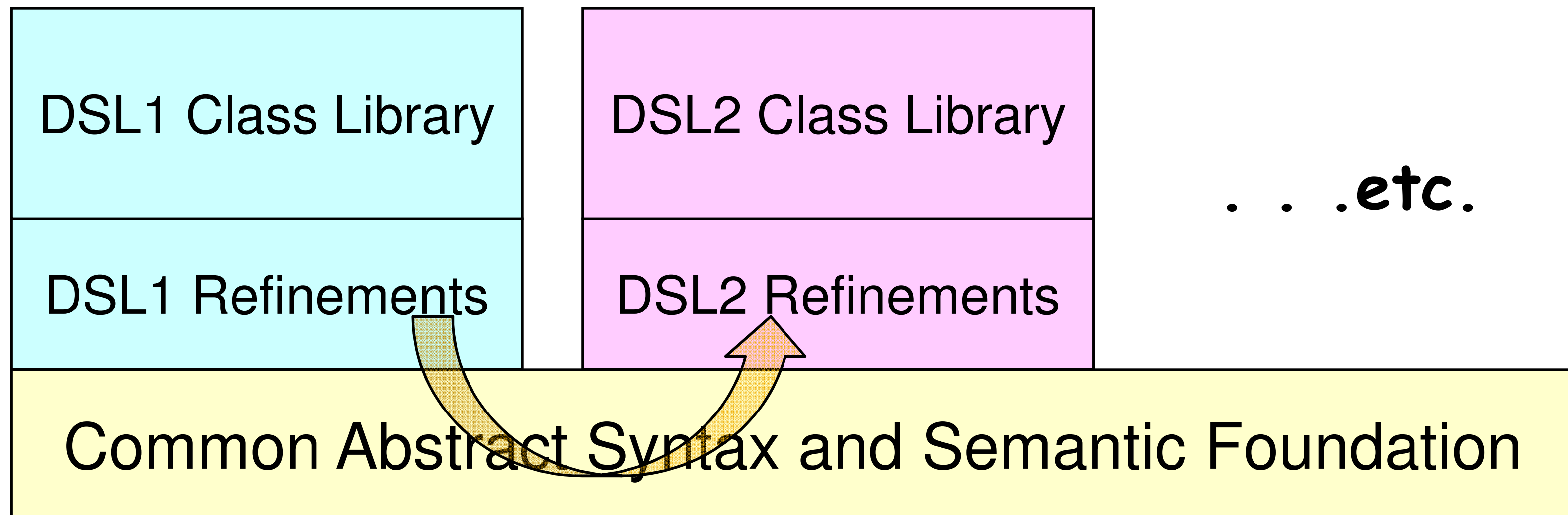


- **FRAGMENTATION PROBLEM:** combining independently specified domain-specific subsystems specified using different DSLs into a coherent and consistent whole

© Copyright Malina Software

Approach to Dealing with Fragmentation

- Having a common syntactic and semantic foundations for the different DSLs seems as if it should facilitate specifying the formal interdependencies between different DSMLs



- NB: Same divide and conquer approach can be used to modularize complex languages
  - Core language base + independent sub-languages (e.g., UML)

© Copyright Malina Software

Key Modeling Language Design Dimensions

- Scope?
  - Broad (general) or narrow (domain specific)?
- Formal or informal? (executable?)
- Semantics?
  - Static: Ontology (concepts and relationships)?
  - Dynamic: Model of computation (how do things happen?)
- New language or an extension or refinement of an existing one?
- Concrete syntax?
  - Graphical? Textual? Heterogeneous?
- Extensible?
- Method of language specification?

© Copyright Malina Software

Formal vs. Informal: Categories

- **Based on degree of "formality"**
  - Precision of definition, internal consistency, completeness, level of detail covered

Category	*Characteristics*	*Primary Purpose*
IMPLEMENTATION	Defined, formal, consistent, complete, detailed	Prediction, Implementation
EXECUTABLE	Defined, formal, consistent, complete	Analysis, Prediction
FORMAL	Defined, formal, consistent	Analysis, Prediction
CODIFIED	Defined, informal	Documentation, Analysis
AD HOC	Undefined, informal	Documentation, Analysis

© Copyright Malina Software

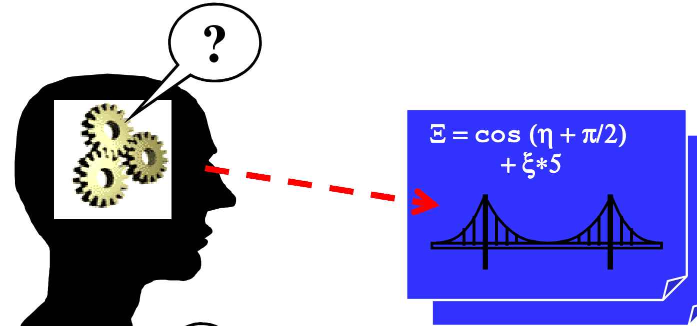
Formality

- **Based on a well understood mathematical theory with existing analysis tools**
  - E.g., automata theory, Petri nets, temporal logic, process calculi, queueing theory, Horne clause logic
  - NB: precise does not always mean detailed
- **Formality provides a foundation for automated validation of models**
  - Model checking (symbolic execution)
  - Theorem proving
  - However, the value of these is constrained due to scalability issues ("the curse of dimensionality")
- **It can also help validate the language definition**
- **But, it often comes at the expense of expressiveness**
  - Only phenomena recognized by the formalism can be expressed accurately

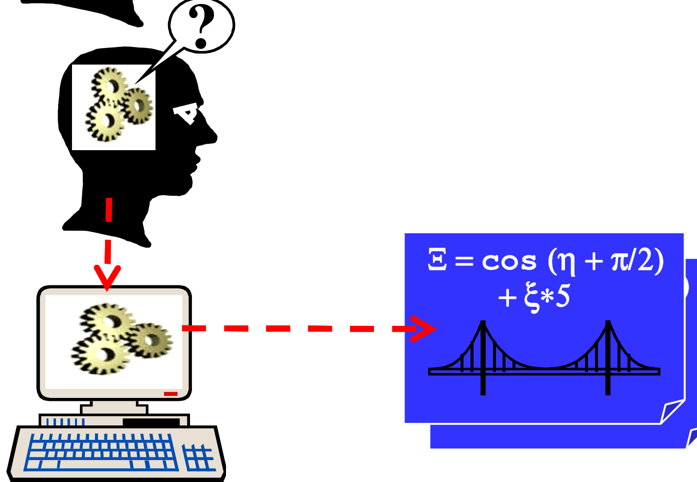
© Copyright Malina Software

How We Can Learn From Models

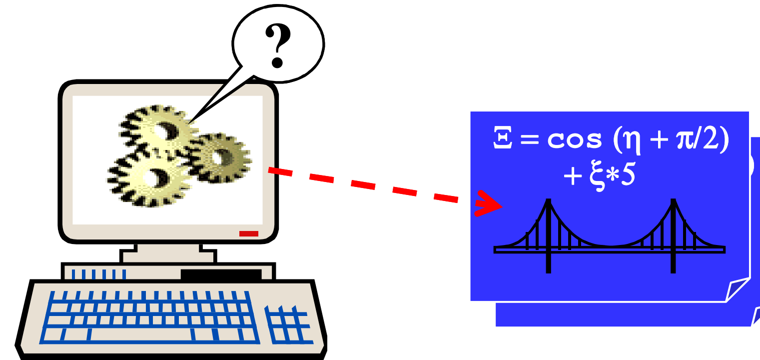
- By inspection
  - mental execution
  - unreliable



- By execution
  - more reliable than inspection
  - direct experience/insight



- By formal analysis
  - reliable (provided the models are accurate)
  - formal methods do not scale very well



© Copyright Malina Software

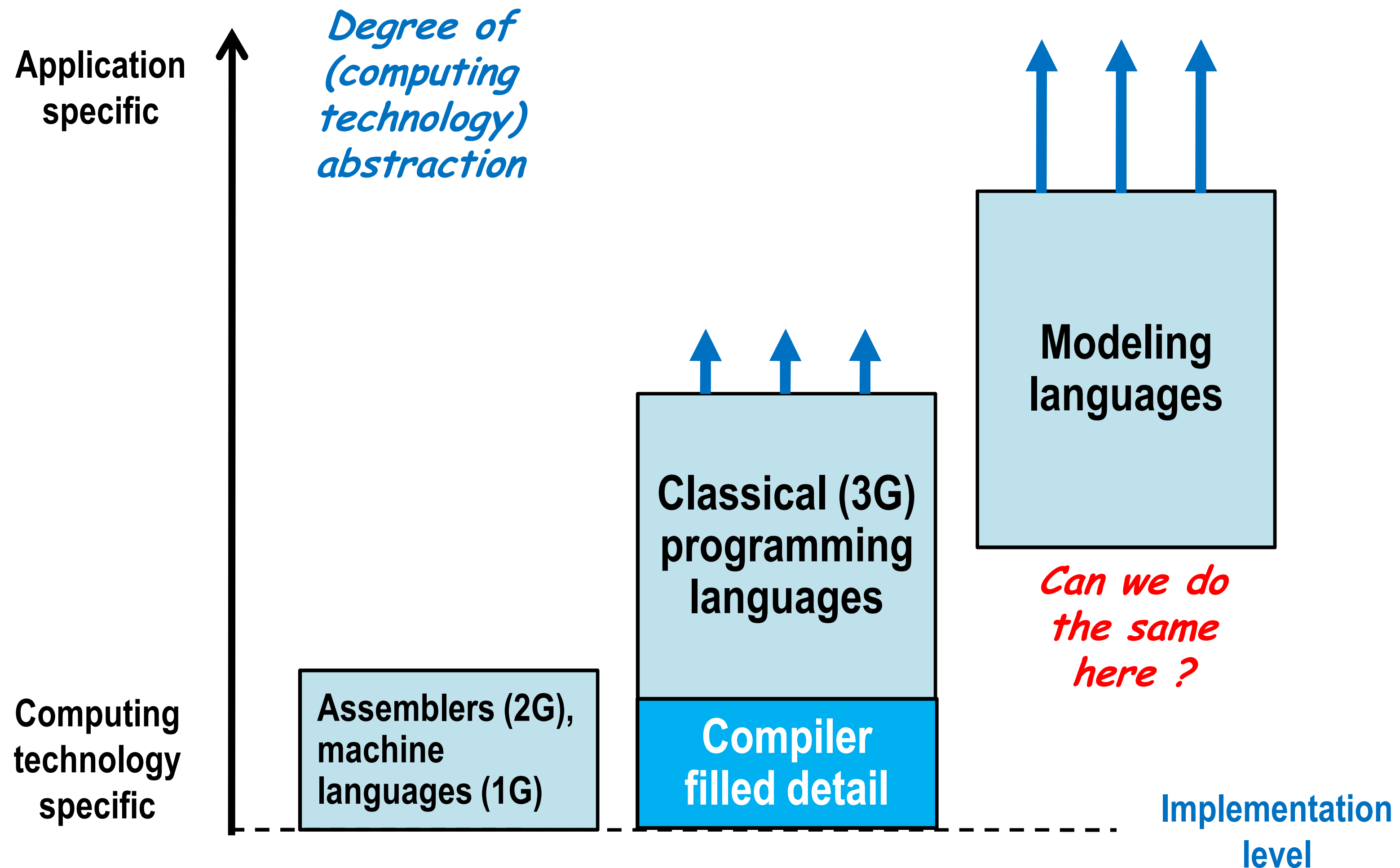
Executable Models

- **Ability to execute a model on a computer and observe its behavior**
  - With possible human intervention when necessary
- **D. Harel: "Models that are not executable are like cars without engines"**
  - However, not all models need be executable
- **Key capabilities**
  - Controllability: ability to start/stop/slow down/speed up/drive execution
  - Observability: ability to view execution and state in model (source) form
  - Partial model execution: ability to execute abstract and incomplete models

© Copyright Malina Software

Modeling Languages for Implementation

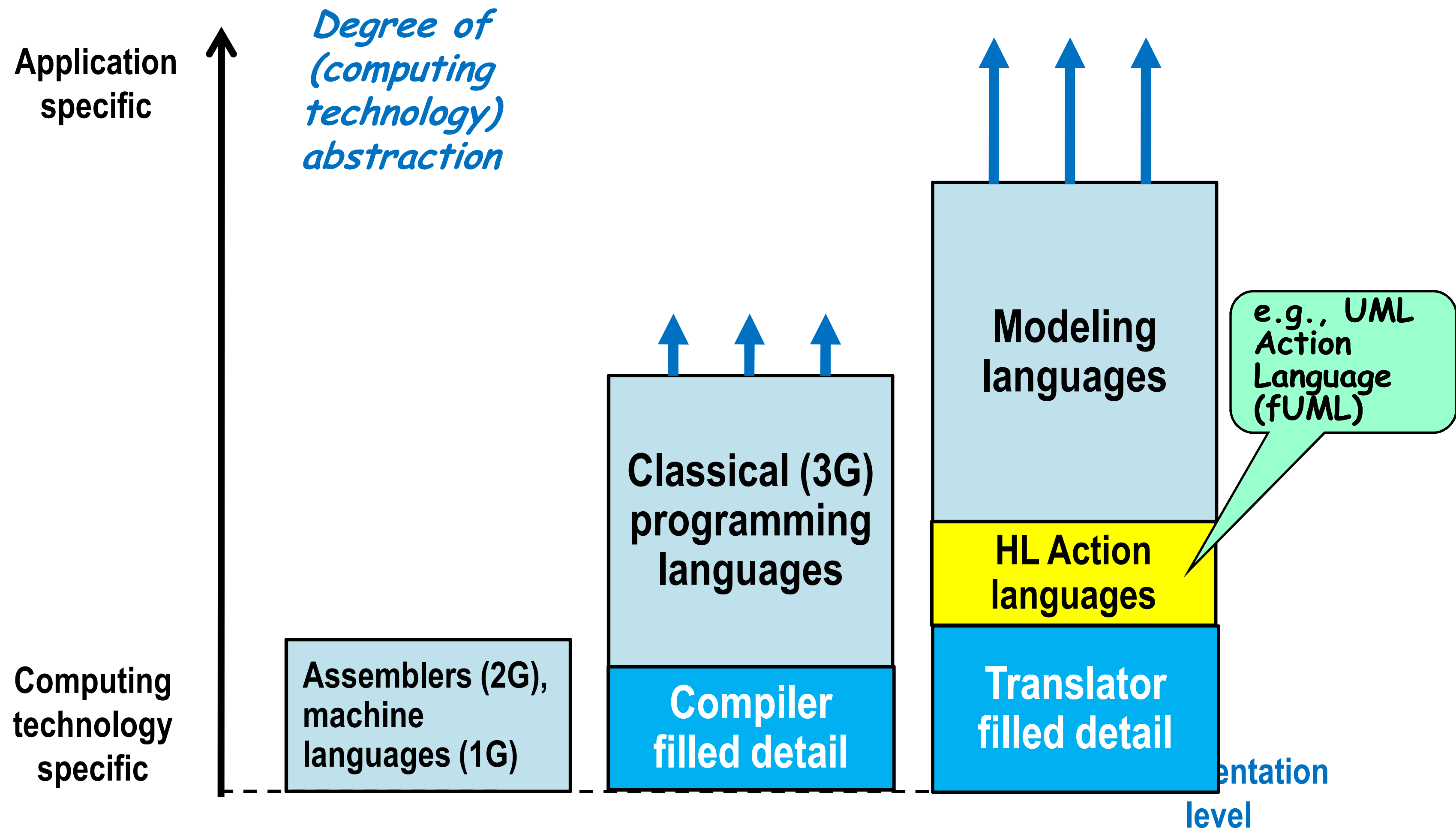
- Much of the evolution of computer languages is motivated by the need to be more human-centric



© Copyright Malina Software

Modeling Languages for Implementation

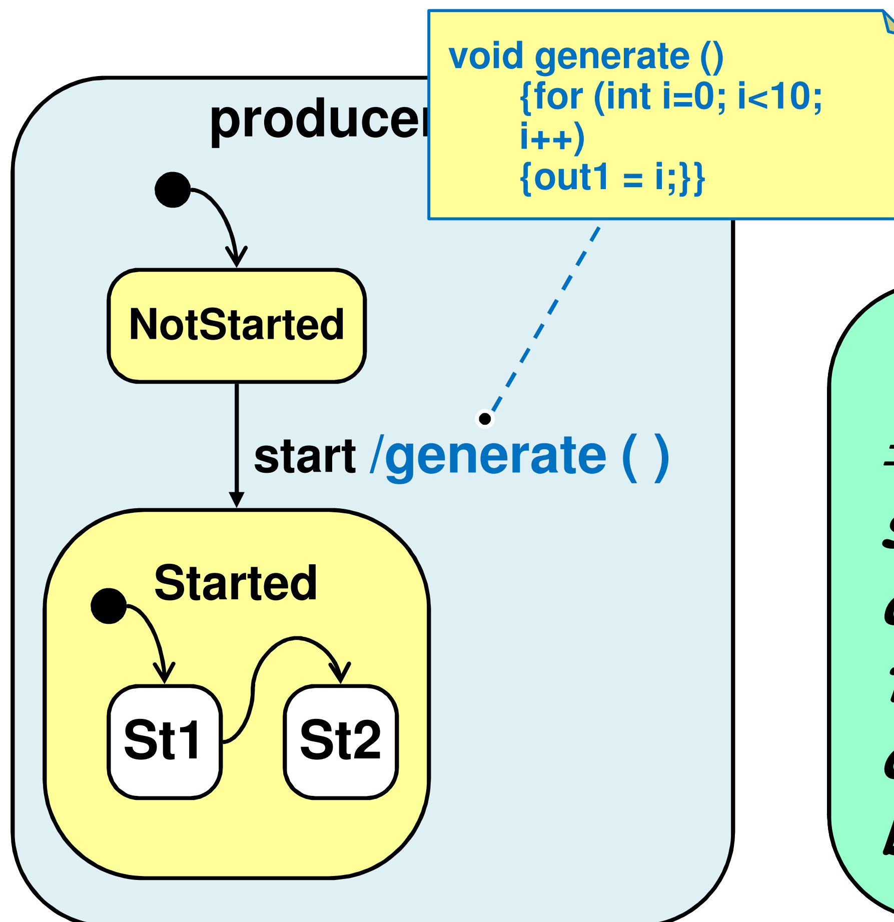
- A number of "descriptive" modeling languages have evolved into fully-fledged implementation languages



© Copyright Malina Software

Implementation Modeling Languages

- Typically a heterogeneous combination of syntaxes and executable languages with different models of computation



Too detailed:
⇒ needs to be supplemented with abstraction (model) transformations to obtain the full benefits of models

© Copyright Malina Software

Key Modeling Language Design Dimensions

- Scope?
  - Broad (general) or narrow (domain specific)?
- Formal or informal? (executable?)
- Semantics?
  - Static: Ontology (concepts and relationships)?
  - Dynamic: Model of computation (how do things happen?)
- New language or an extension or refinement of an existing one?
- Concrete syntax?
  - Graphical? Textual? Heterogeneous?
- Extensible?
- Method of language specification?

© Copyright Malina Software

Semantics: Ontology (Information Science)

- "A formal representation of knowledge as a set of concepts within a domain and the relationships between those concepts" [Wikipedia]
- In modeling language design:
  - The set of primitive concepts that represent (model) the phenomena in a domain
  - The rules for combining these concepts to construct valid (well-formed) statements in the language
- Example [UML]:
  - Concepts: Class, Association, Dependency, Attribute, etc.
  - Relationships: Attributes are owned by Classes
- Requires a specialized language (e.g., OWL, MOF)

© Copyright Malina Software

Semantics: Model of Computation (MoC)

- Model of Computation: A conceptual framework (paradigm) used to specify how a (software) system realizes its prescribed functionality
  - Where and how does behavior (computation) take place
- Selecting the dominant MoC(s) is a primary language design decision
  - Based on characteristics of the application and requirements
  - Closely coupled to the chosen formalism

© Copyright Malina Software

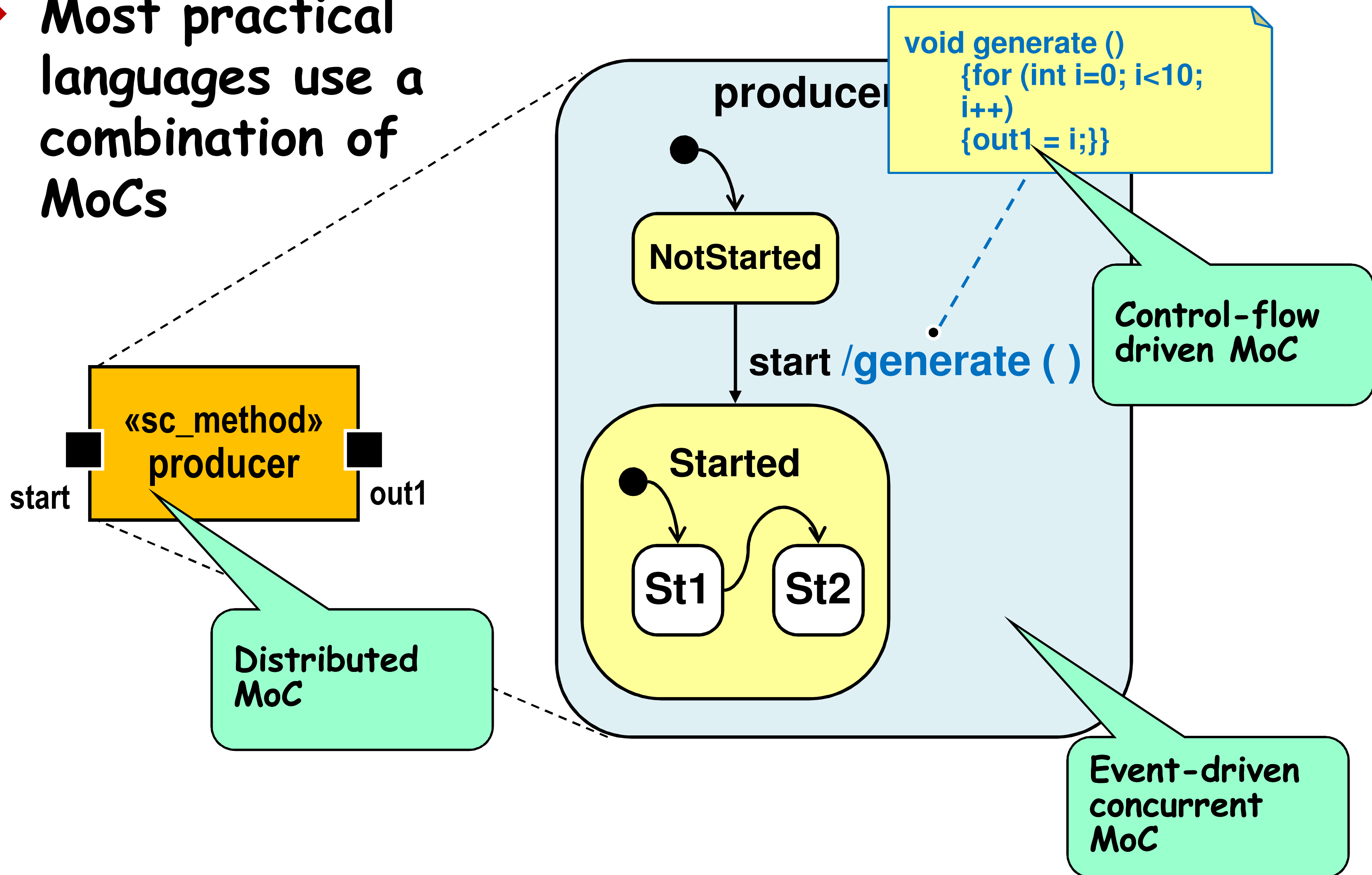
MoC Dimensions

- Concurrency paradigm: does computation occur sequentially (single thread) or in parallel (multiple threads)?
- Causality paradigm: what causes behavior
  - event driven, control driven, data driven (functional), time driven, logic driven, etc.
- Execution paradigm: nature of behavior execution
  - Synchronous (discrete), asynchronous, mixed (LSGA)
- Interaction paradigm: how do computational entities interact
  - synchronous, asynchronous, mixed
- Distribution paradigm: does computation occur in a single site or multiple?
  - Multisite (⇒ concurrent execution) vs. single site
  - If multisite: Coordinated or uncoordinated (e.g., time model, failure model)?

© Copyright Malina Software

Nesting MoCs

- Most practical languages use a combination of MoCs



© Copyright Malina Software

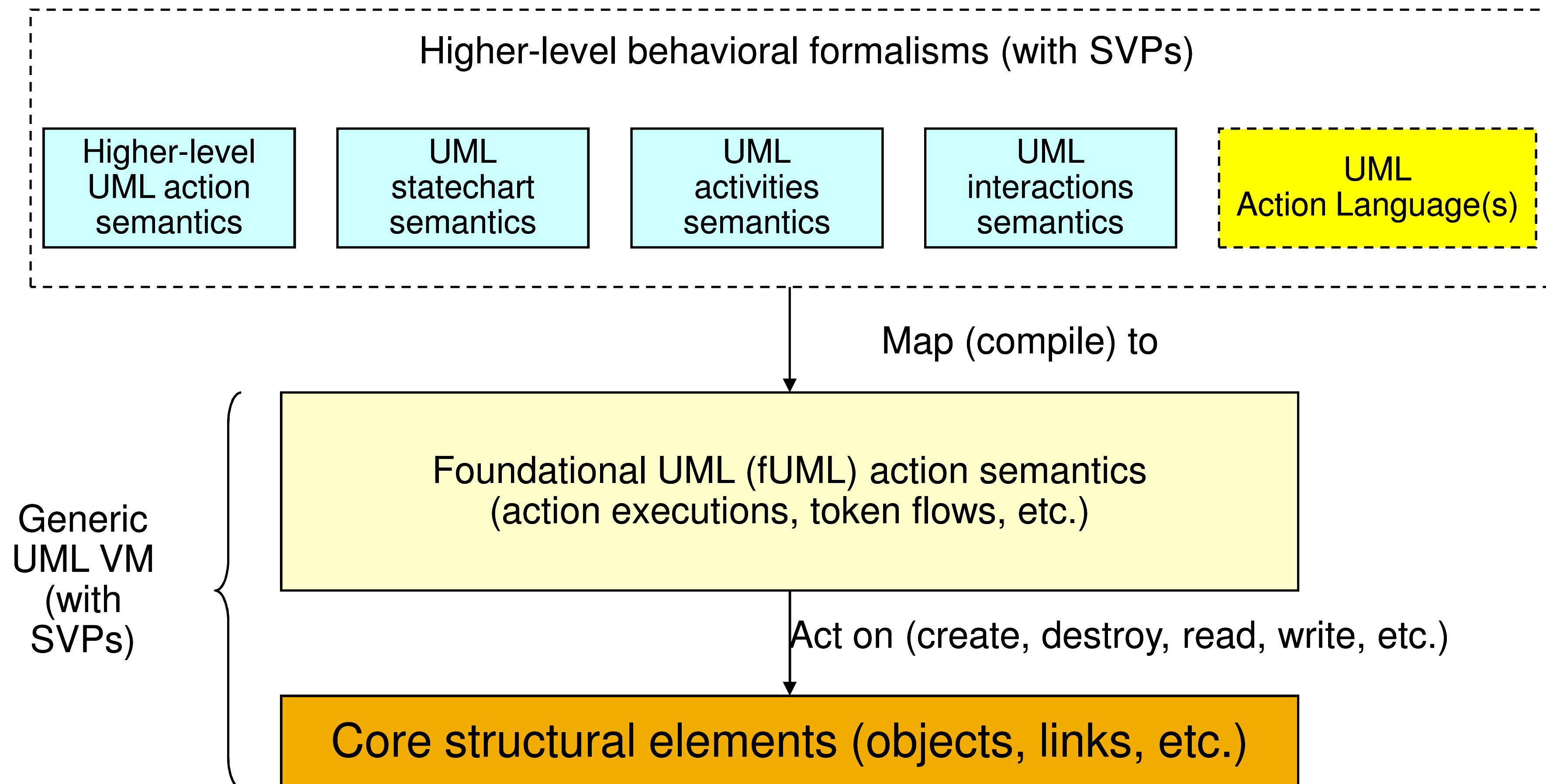
On Specifying Semantics

- Semantics are specified using a language whose semantics are already defined
- Numerous approaches to defining run-time semantics of computer languages
  - Informal natural language description are the most common way
  - Denotational, operational, axiomatic
- The "Executable UML Foundation" specification provides a standard for defining semantics
  - Defines the dynamic semantics for a subset of standard UML concepts that have a run-time manifestation
  - Semantics of modeling languages can be specified as programs written using a standardized executable modeling language (operational (interpretive) approach)
  - The semantics of Executable UML itself are defined axiomatically

© Copyright Malina Software

OMG Approach to Specifying UML Semantics

- **UML semantics hierarchy**
  - As defined by the Executable UML Foundation proto-standard

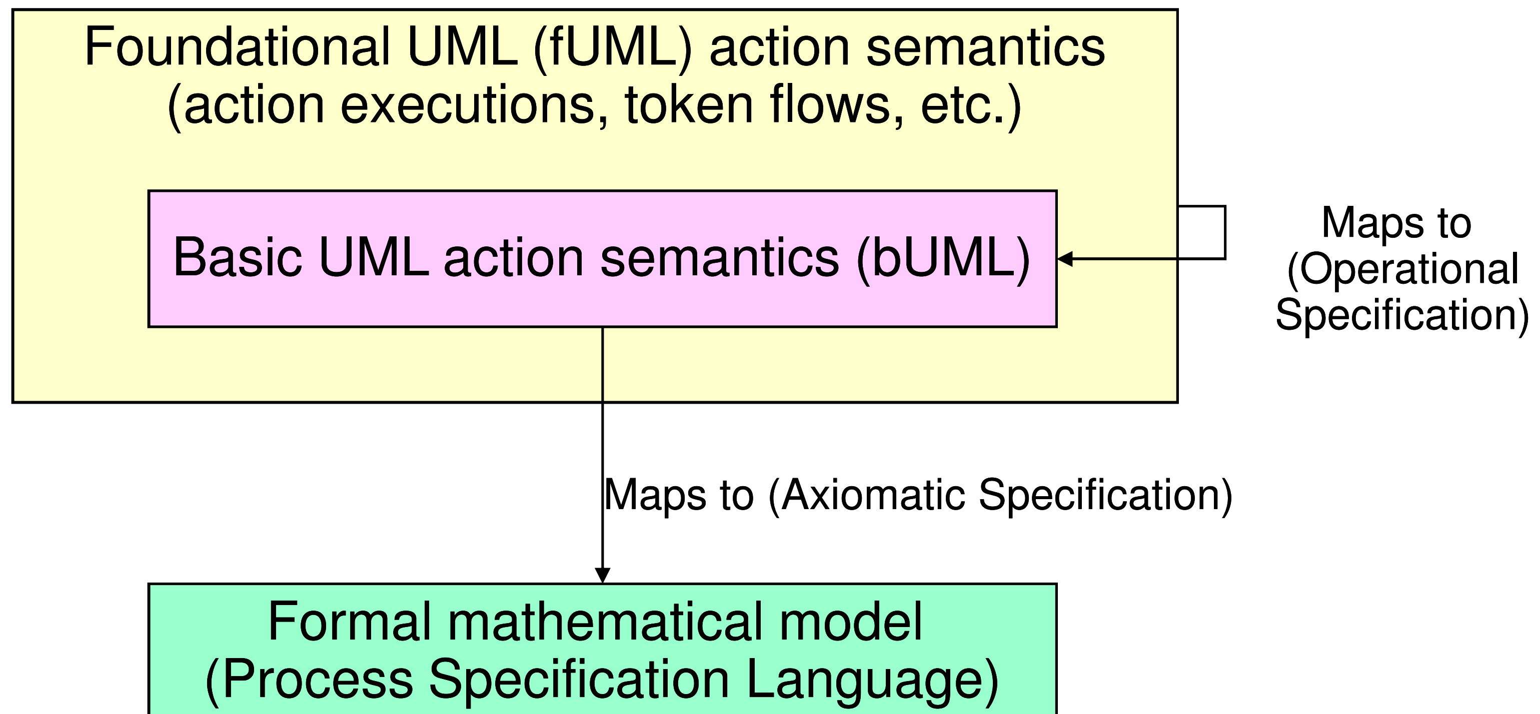


SVP = Semantic Variation Point

© Copyright Malina Software

Foundational UML (fUML) and Basic UML (bUML)

- A subset of fUML actions is used as a core language (Basic UML) that is used to describe fUML itself



Basis for a formalization of UML

© Copyright Malina Software

Key Modeling Language Design Dimensions

- Scope?
  - Broad (general) or narrow (domain specific)?
- Formal or informal? (executable?)
- Semantics?
  - Static: Ontology (concepts and relationships)?
  - Dynamic: Model of computation (how do things happen?)
- New language or an extension or refinement of an existing one?
- Concrete syntax?
  - Graphical? Textual? Heterogeneous?
- Extensible?
- Method of language specification?

© Copyright Malina Software

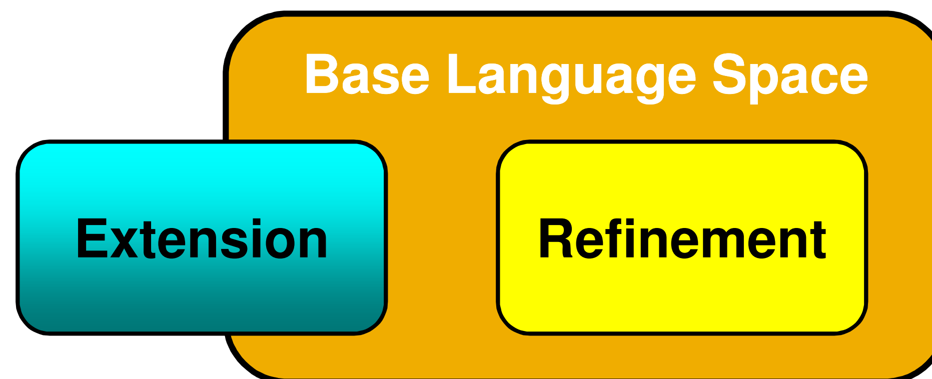
Approaches to DSML Design

1. Define a completely new language from scratch
2. Extend an existing language: add new domain-specific concepts to an existing (base) language
3. Refine an existing language: specialize the concepts of a more general existing (base) language

© Copyright Malina Software

Refinement vs Extension

- Semantic space = the set of all valid programs that can be specified with a given computer language
- Refinement: subsets the semantic space of the base language (e.g., UML profile mechanism)
  - Enables reuse of base-language infrastructure
- Extension: intersects the semantic space of the base language



© Copyright Malina Software

Comparison of Approaches

Approach	Expressive Power	Ease of Lang.Design	Infrastructure Reuse	Multimodel Integration
New Language	*High*	*Low*	*Low*	*Low*
Extension	*Medium*	*Medium*	*Medium*	*Medium*
Refinement	*Low*	*High*	*High*	*High*

© Copyright Malina Software

Refine, Define, or Extend?

- **Depends on the problem at hand**
  - Is there significant semantic similarity between the UML metamodel and the DSML metamodel?
    - Does every domain concept represent a semantic specialization of some UML concept?
    - No semantic or syntactic conflicts?
  - Is language design expertise available?
  - Is domain expertise available?
  - Cost of maintaining a language infrastructure?
  - Need to integrate models with models based on other DSMLs?
- **Example: Specification and Description Language (SDL: ITU-T standard Z.100)**
  - DSML for defining telecommunications systems and standards
  - First defined in 1970
  - Currently being redefined as a UML profile (Z.109)

© Copyright Malina Software

Key Modeling Language Design Dimensions

- Scope?
  - Broad (general) or narrow (domain specific)?
- Formal or informal? (executable?)
- Semantics?
  - Static: Ontology (concepts and relationships)?
  - Dynamic: Model of computation (how do things happen?)
- New language or an extension or refinement of an existing one?
- Concrete syntax?
  - Graphical? Textual? Heterogeneous?
- Extensible?
- Method of language specification?

© Copyright Malina Software

Concrete Syntax Design

- Two main forms:
  - For computer-to-computer interchange (e.g., XMI)
  - For human consumption – "surface" syntax
- Designing a good surface syntax is the area where we know least
  - If a primary purpose of models is communication and understanding, what syntactical forms should we use for a given language?
- Requires multi-disciplinary skills
  - Domain knowledge
  - Computer language design
  - Cognitive science
  - Psychology
  - Cultural Anthropology
  - Graphic design
  - Computer graphics

© Copyright Malina Software

Concrete Syntax Design Dimensions

- **Graphical (visual) or textual?**
  - …or both?
- **If visual: what are the primary and secondary metaphors (for visual languages)?**
  - Consistency, intuitive appeal
- **Multiple viewpoints**
  - Which ones?
  - How are they represented?
  - How are they linked?
- **How is the syntax defined?**
  - Examples or some formalism (which one?)?
  - Mapping to abstract syntax?
- **How is the syntactical information stored in the model?**
  - We may want to use different notations for the same model depending on viewpoint
- **Interchange format**
  - Human readable as well? (e.g., XML based)

© Copyright Malina Software

Key Modeling Language Design Dimensions

- Scope?
  - Broad (general) or narrow (domain specific)?
- Formal or informal? (executable?)
- Semantics?
  - Static: Ontology (concepts and relationships)?
  - Dynamic: Model of computation (how do things happen?)
- New language or an extension or refinement of an existing one?
- Concrete syntax?
  - Graphical? Textual? Heterogeneous?
- Extensible?
- Method of language specification?

© Copyright Malina Software

Key Language Design Questions

- **Who are the primary users?**
  - Authors / readers? (i.e., primary use cases)
- **What is its primary purpose?**
  - Documentation, analysis, prediction, implementation?
- **Context**
  - What is the context in which models defined in the language will have to fit?
    - What is the dominant model of behavior ⇒ MoC

© Copyright Malina Software

Tutorial Outline

- On Models and Model-Based Software Engineering
- The Key Dimensions of Modeling Language Design
- Defining a Modeling Language
- Case Study: UML
- Language Refinement: UML Profiles
- Model Transformations

© Copyright Malina Software

Defining a Modeling Language

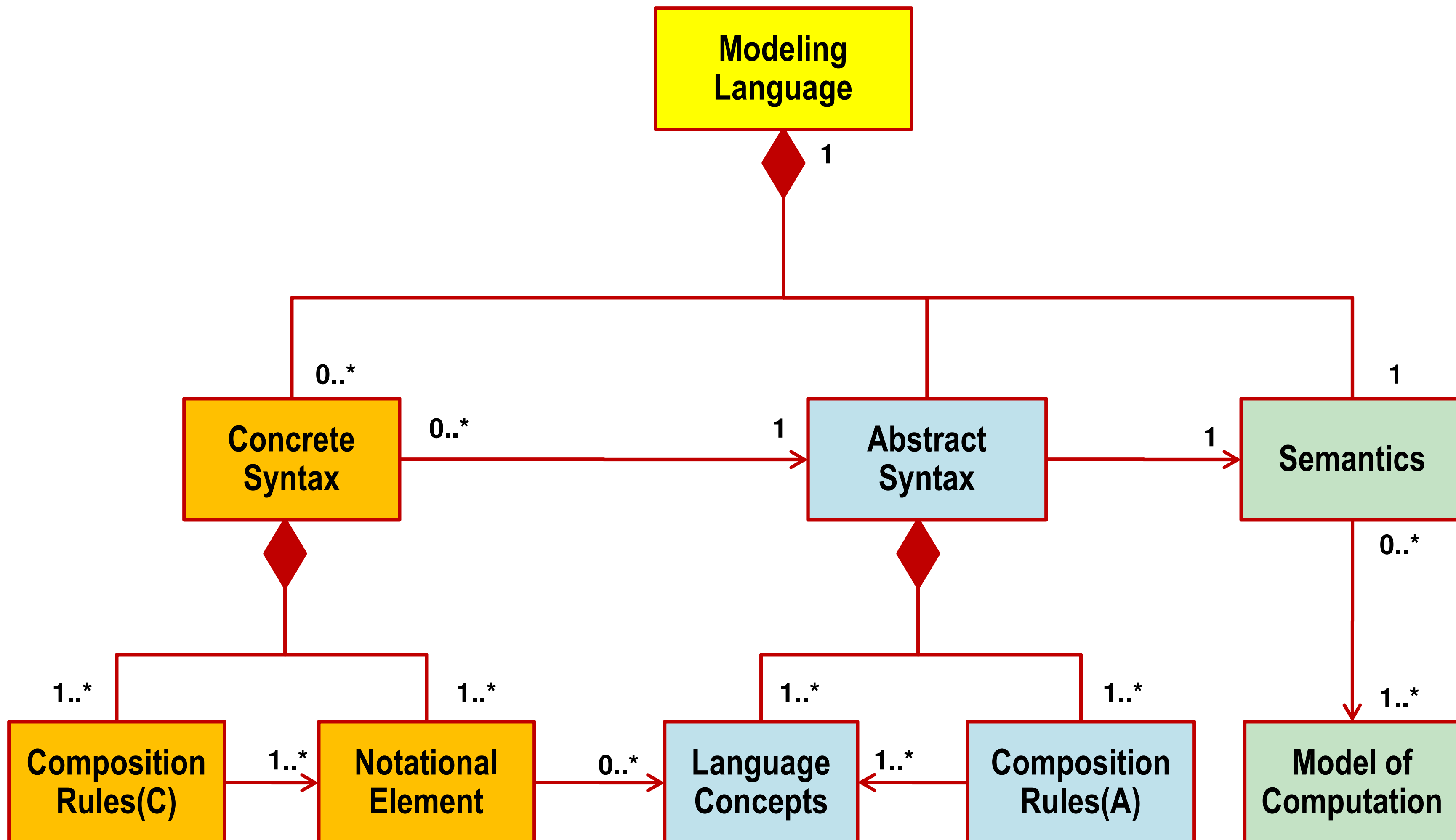
- **The definition of a modeling language consists of:**

ABSTRACT SYNTAX

- Set of language concepts/constructs ("ontology")
  - e.g., Account, Customer, Class, Association, Attribute, Package
- Rules for combining language concepts (well-formedness rules)
  - e.g., "each end of an association must be connected to a class"
- CONCRETE SYNTAX (notation/representation)
  - e.g., keywords, graphical symbols for concepts
- SEMANTICS: the *meaning* of the language concepts
  - i.e., what real-world artifacts do the concepts represent?

© Copyright Malina Software

The Key Elements of a Modeling Language

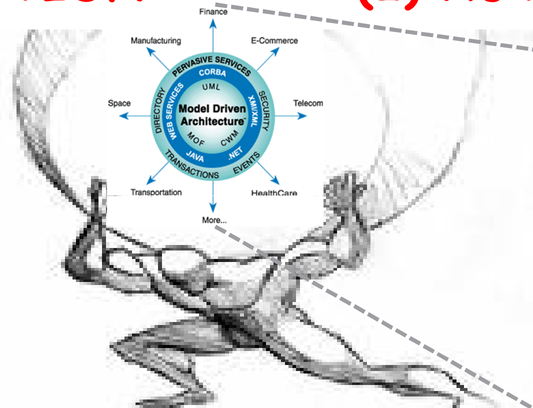


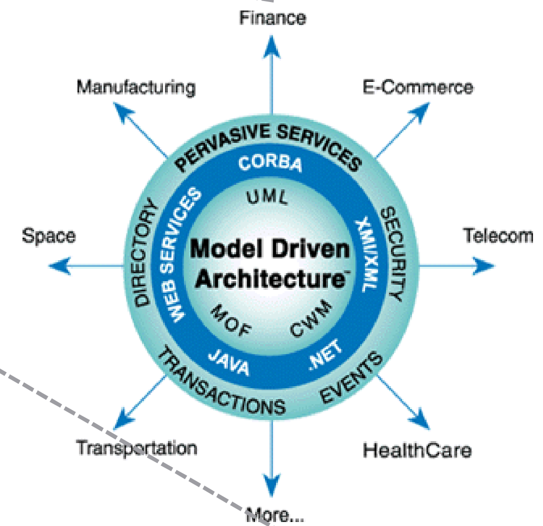
© Copyright Malina Software

Model-Driven Architecture (MDA)™

- In recognition of the increasing importance of MBE, the Object Management Group (OMG) is developing a set of supporting industrial standards

(1) ABSTRACTION (2) AUTOMATION





(3) INDUSTRY STANDARDS

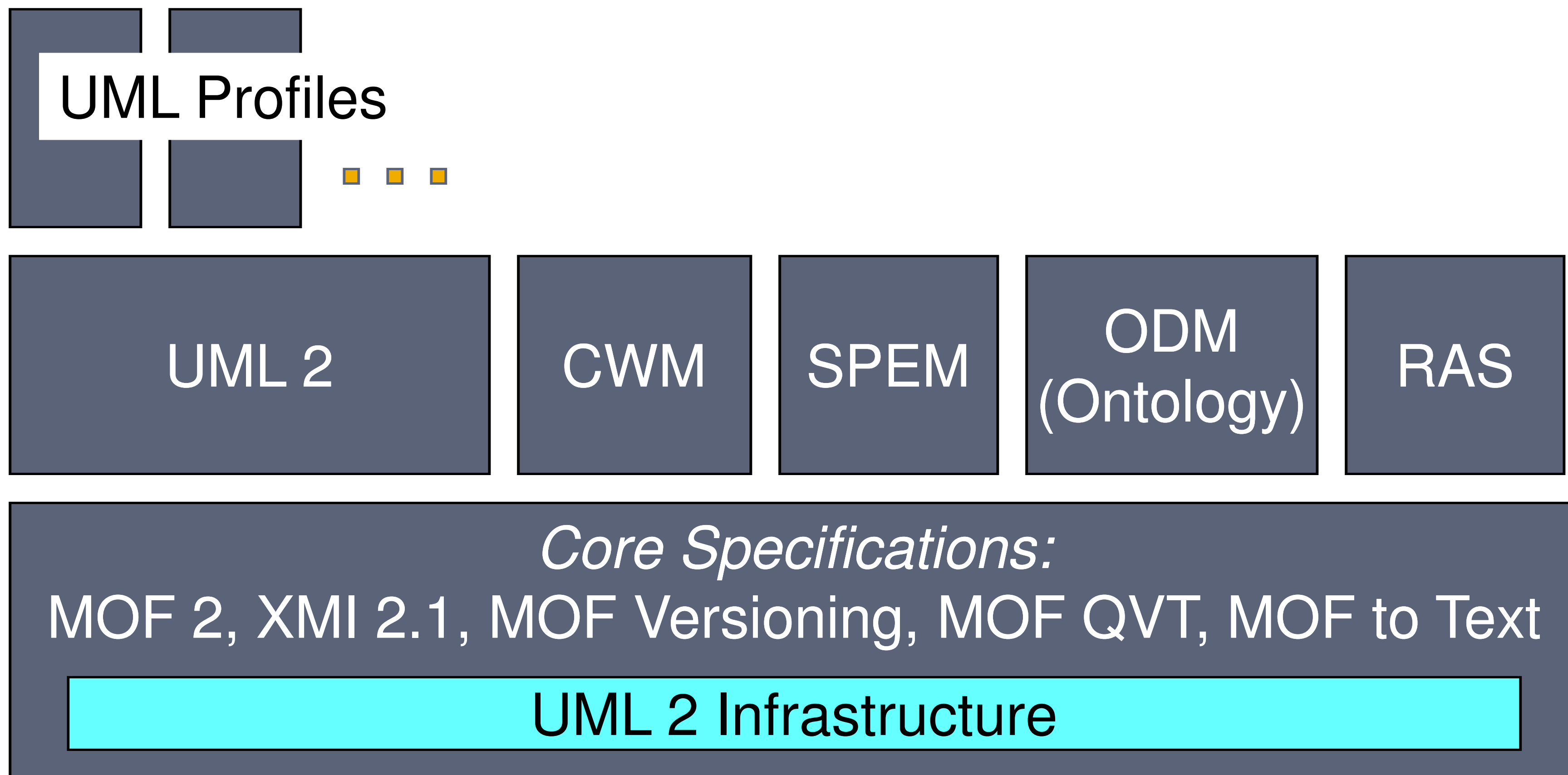
- UML 2
- OCL
- MOF
- SysML
- SPEM
- …etc.

http://www.omg.org/mda/

© Copyright Malina Software

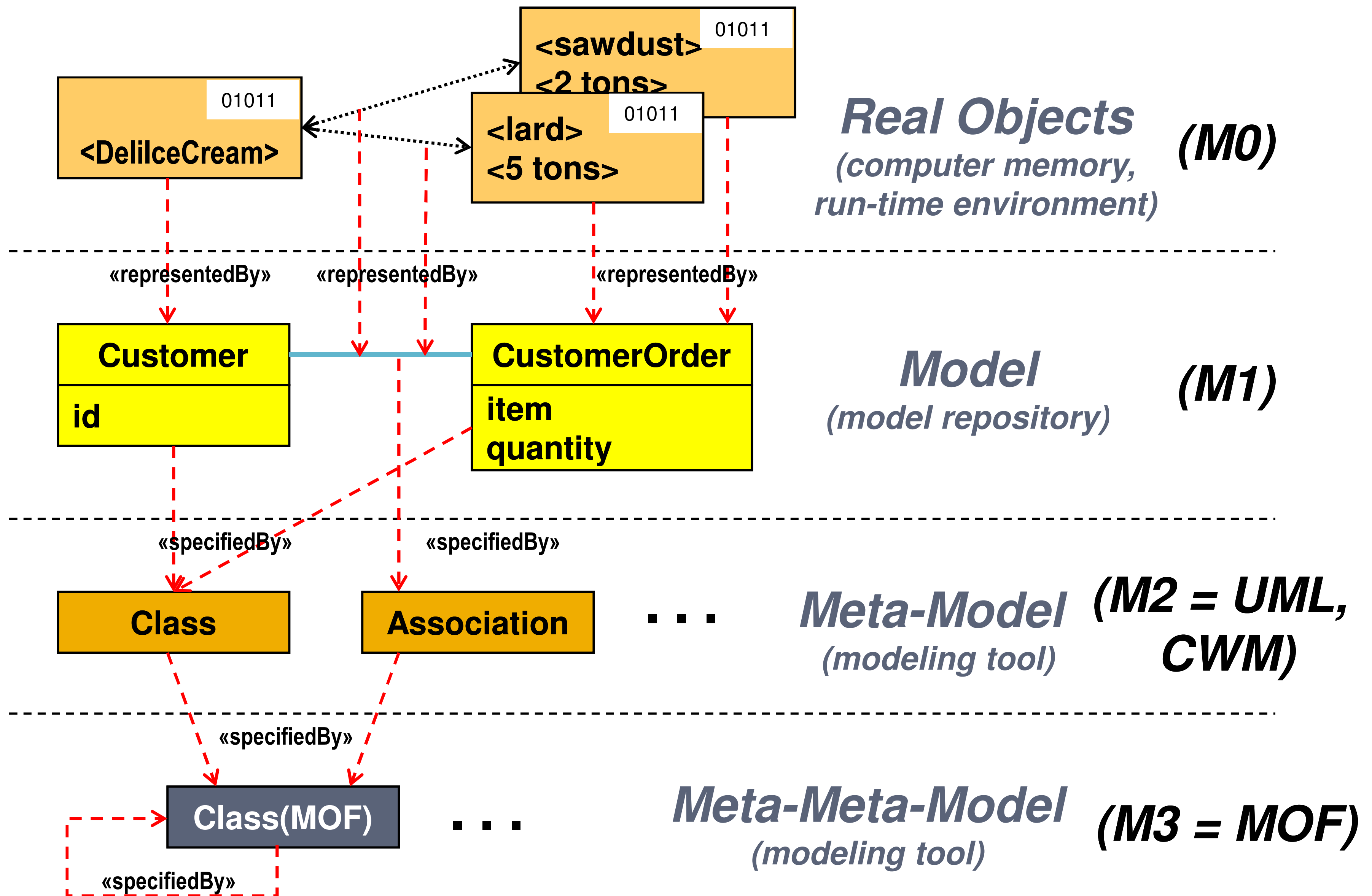
MDA Languages Architecture

- MDA = OMG's initiative to support model-based engineering with a set of industry standards:



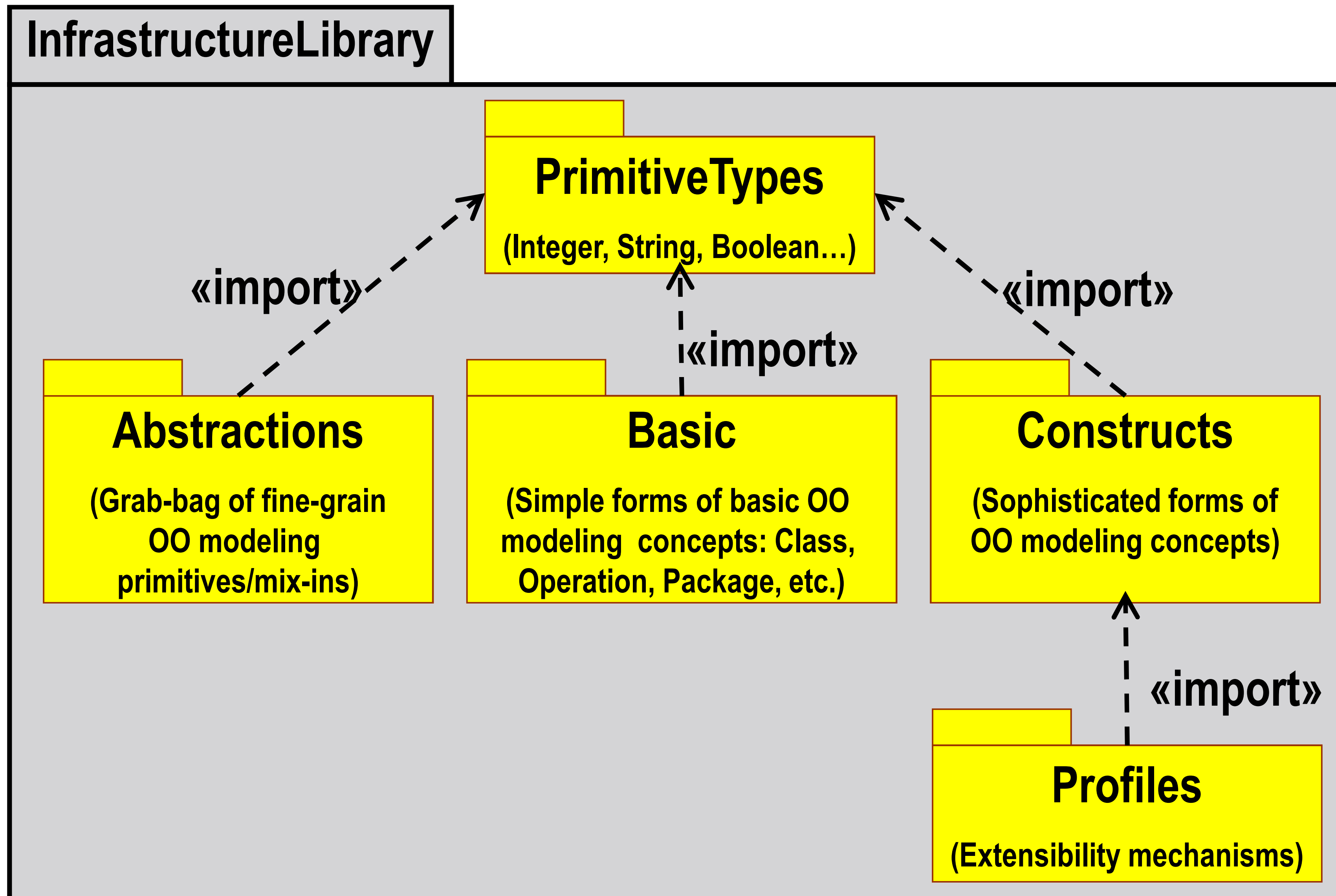
© Copyright Malina Software

The OMG 4-Layer "Architecture"



© Copyright Malina Software

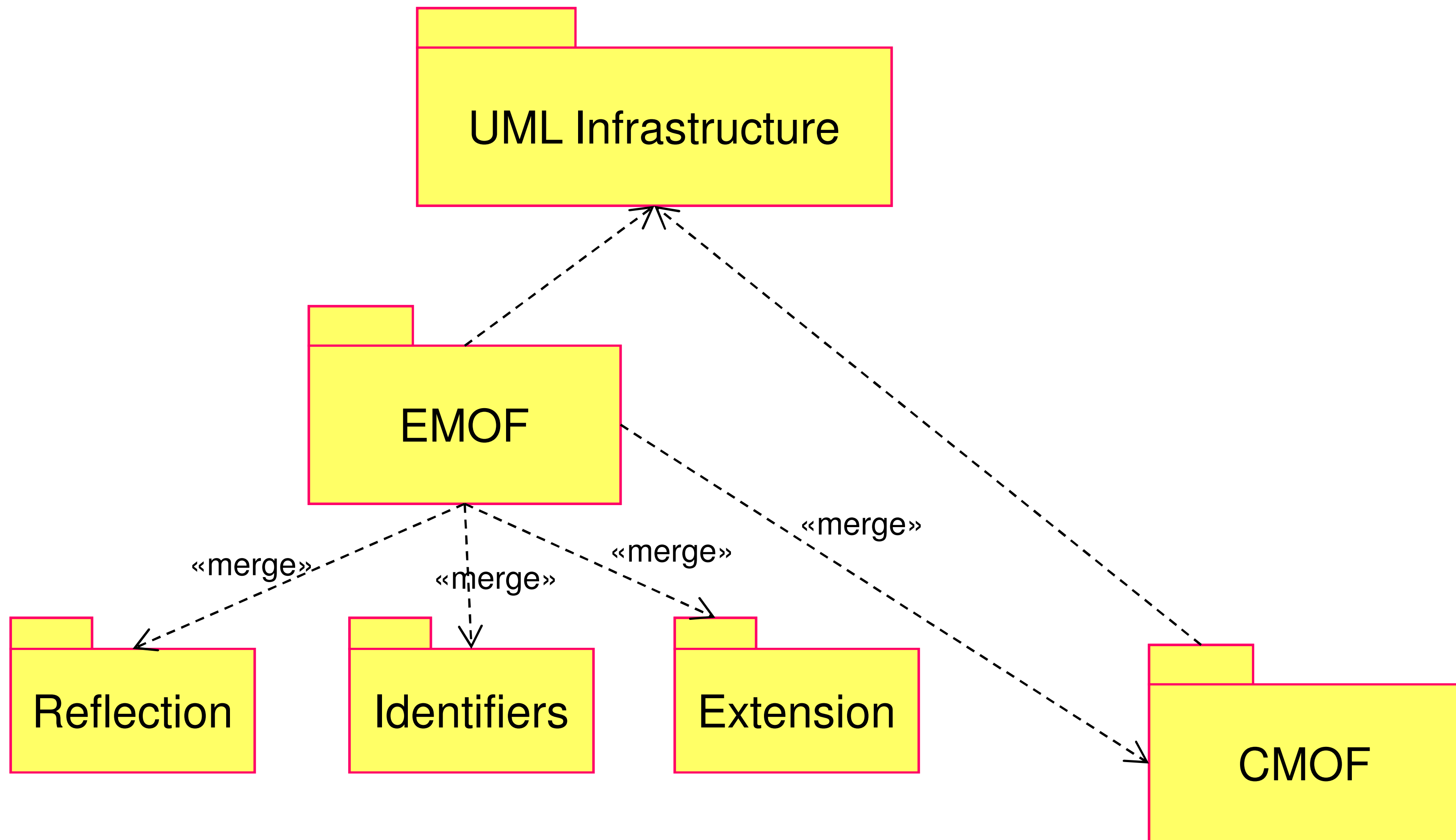
Infrastructure Library - Contents



© Copyright Malina Software

© Copyright Malina Software

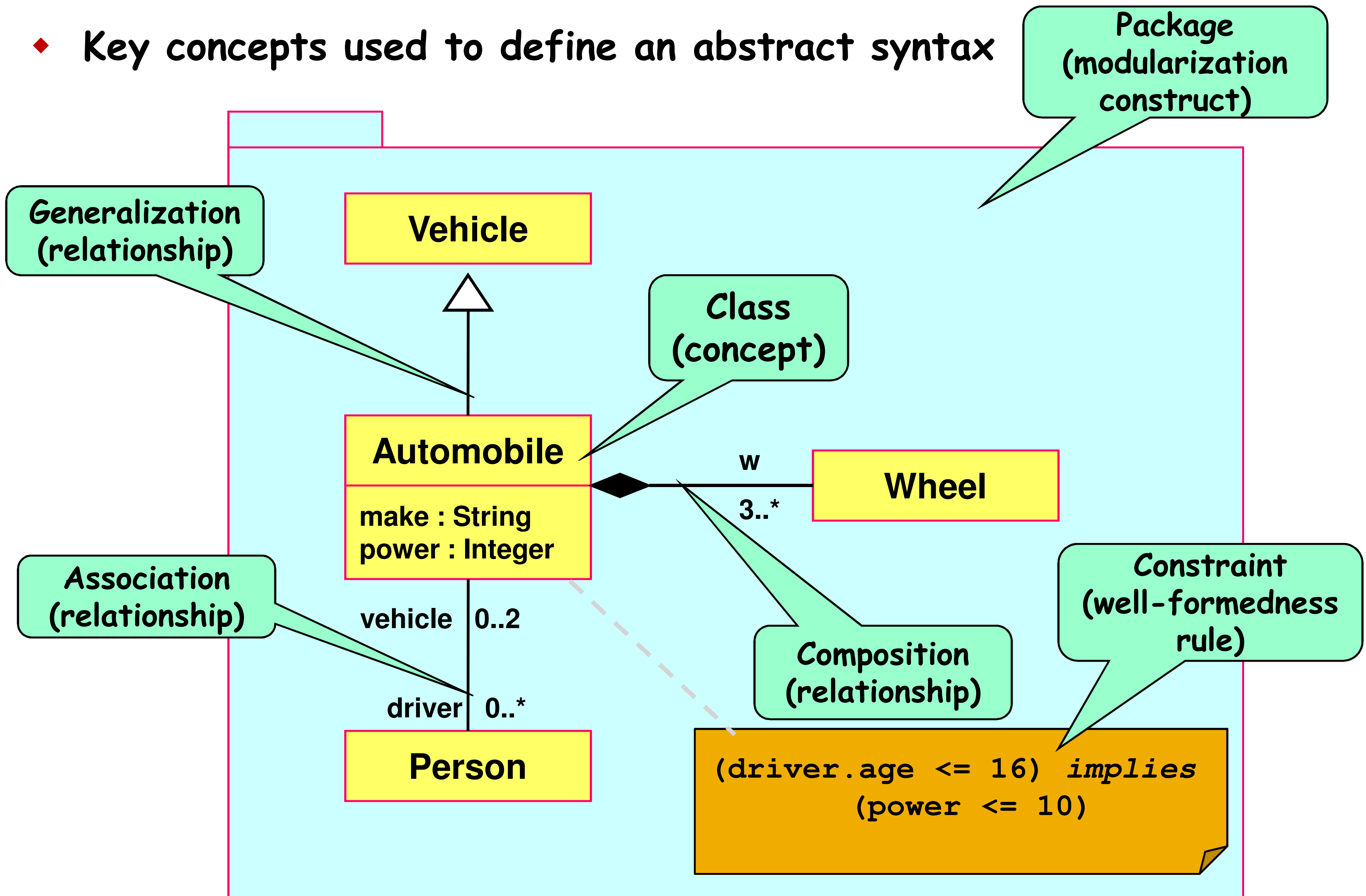
The Structure of MOF (simplified)



© Copyright Malina Software

Essential MOF Concepts (Example)

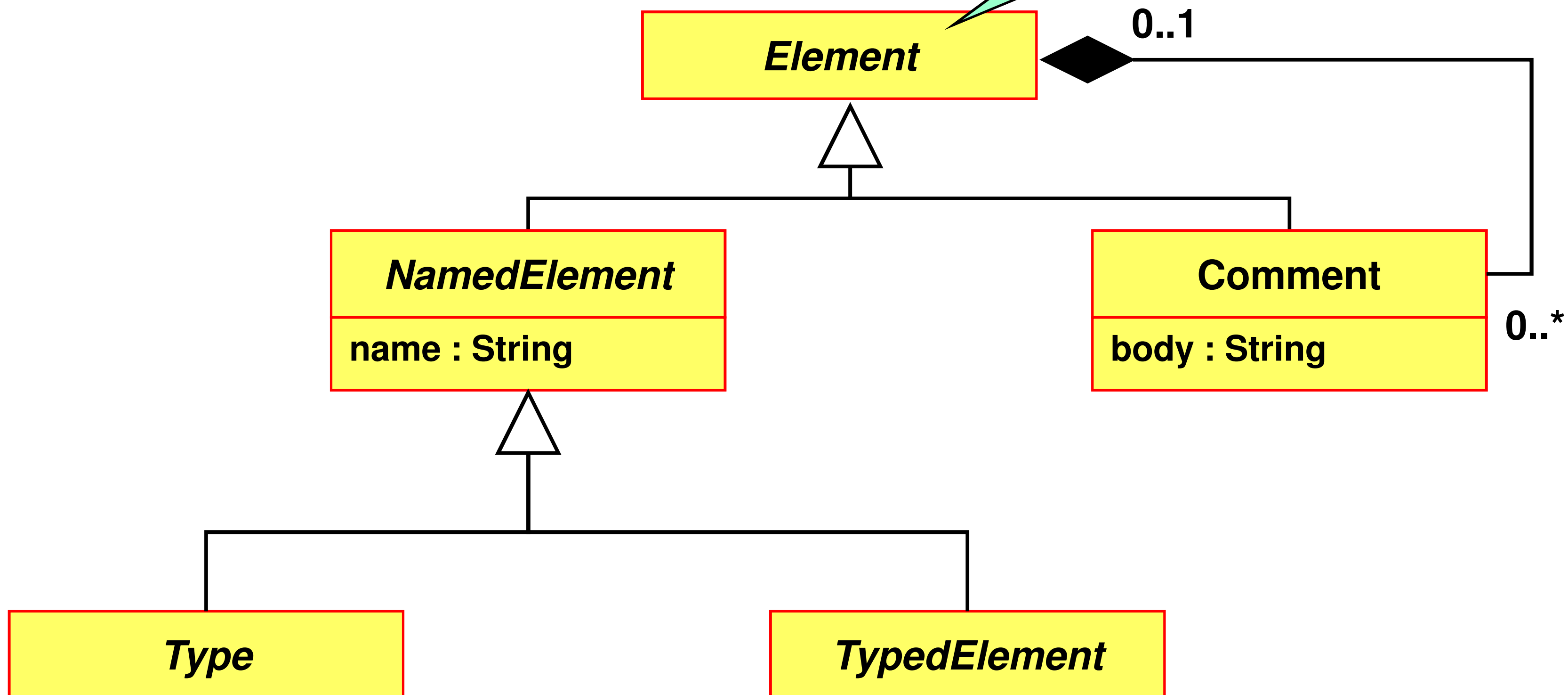
- Key concepts used to define an abstract syntax



© Copyright Malina Software

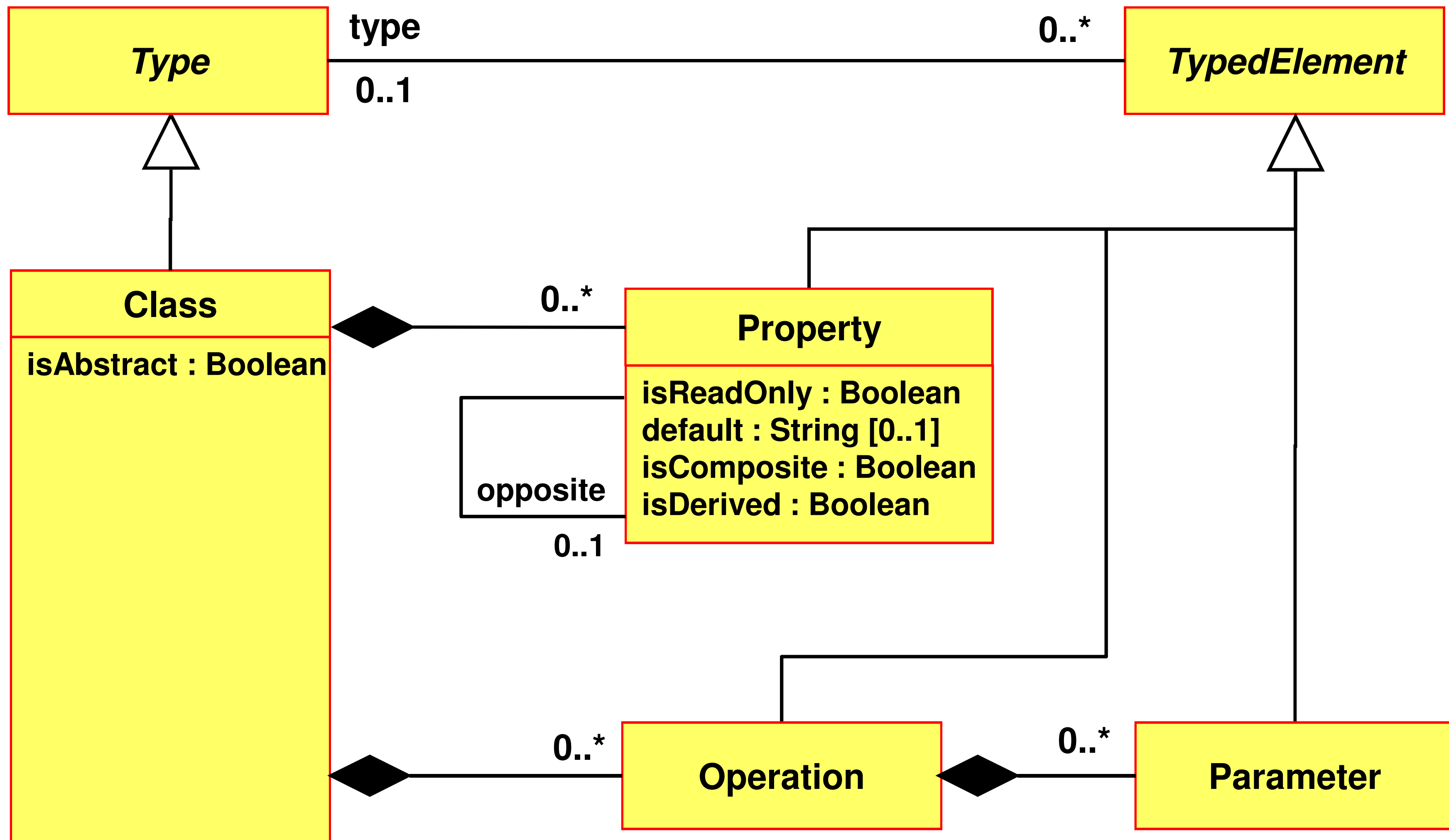
Example: EMOF Metamodel (Root)

- Using MOF to define (E)MOF



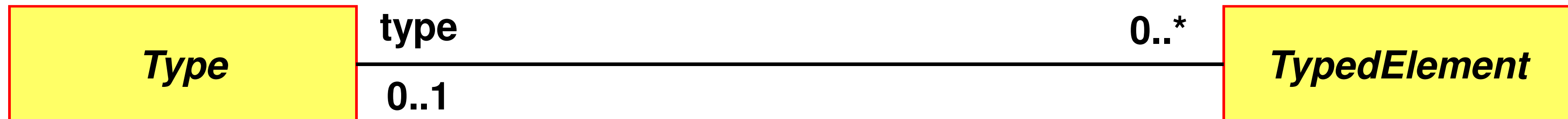
© Copyright Malina Software

Example: EMOF Metamodel (simplified)



© Copyright Malina Software

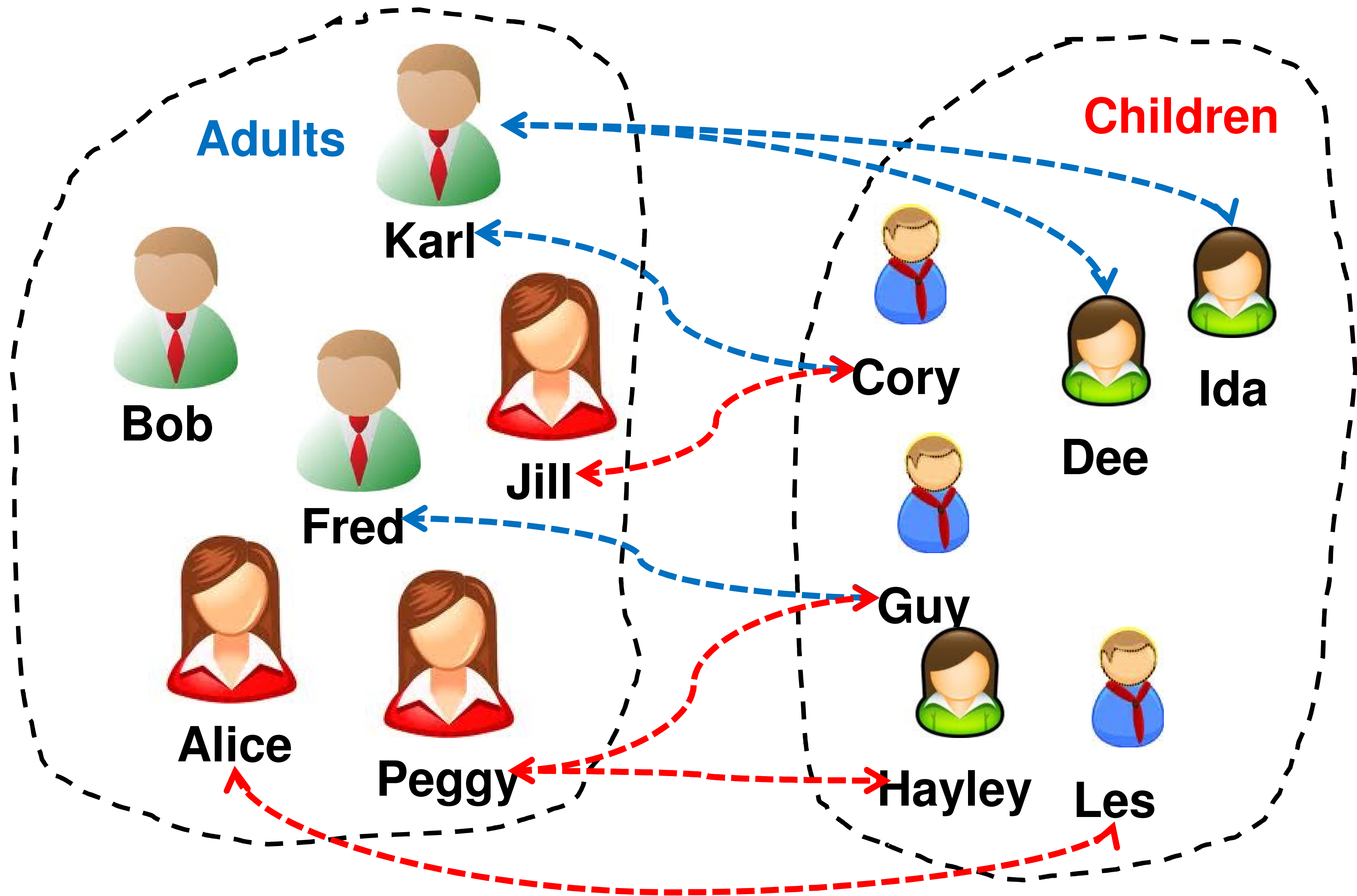
The Meaning of Class Models



- A key element of abstract syntax definition
- What does this model represent?
  - There is much confusion about the meaning of this type of diagram (model)
- Questions:
  - What do the boxes and lines represent?
  - How many Types are represented in this diagram?
  - What does "type" mean?
  - What does "0..*" mean?

© Copyright Malina Software

Example: Two Populations with Relationships



© Copyright Malina Software

Some Facts We Can State

- **Particular statements:**
  - Bob has no children (in the set Children)
  - Karl is the father of Dee and Ida
  - Karl and Jill are the parents of Cory
  - Fred and Peggy are the parents of Guy
  - Peggy is the mother of Hayley
  - Alice is the mother of Les
- **General statements (through abstraction):**
  - Children can have one or two Adult parents
  - Some Adults are parents of Children
  - Every Person has a name and a gender

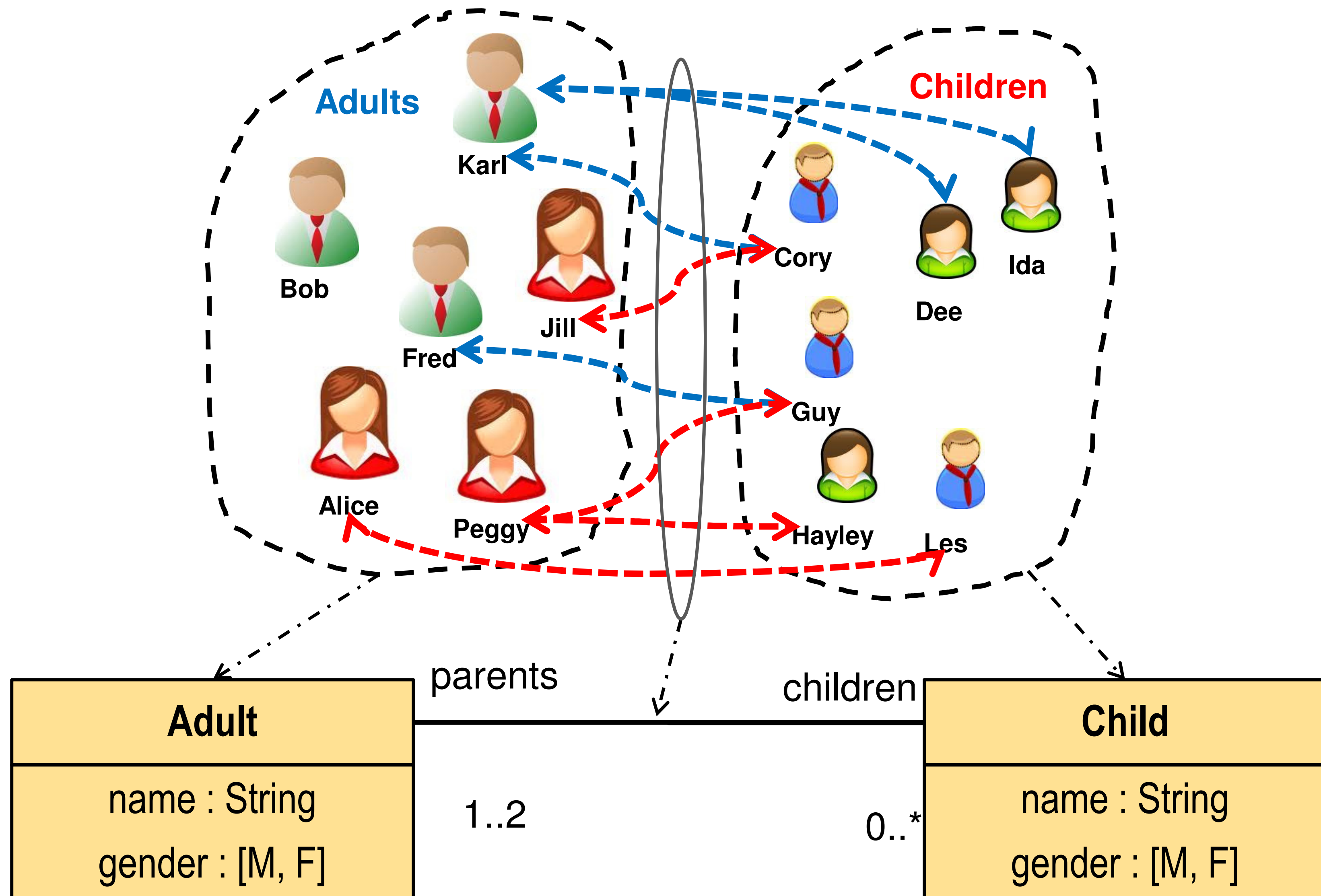
© Copyright Malina Software

Classes and Associations in MOF (and UML)

- Class: A *specification* of a *collection* of object instances characterized by posessing the same set of features and behaviours
- Association: A *specification* of a *collection* of links whose ends conform respectively to the same type

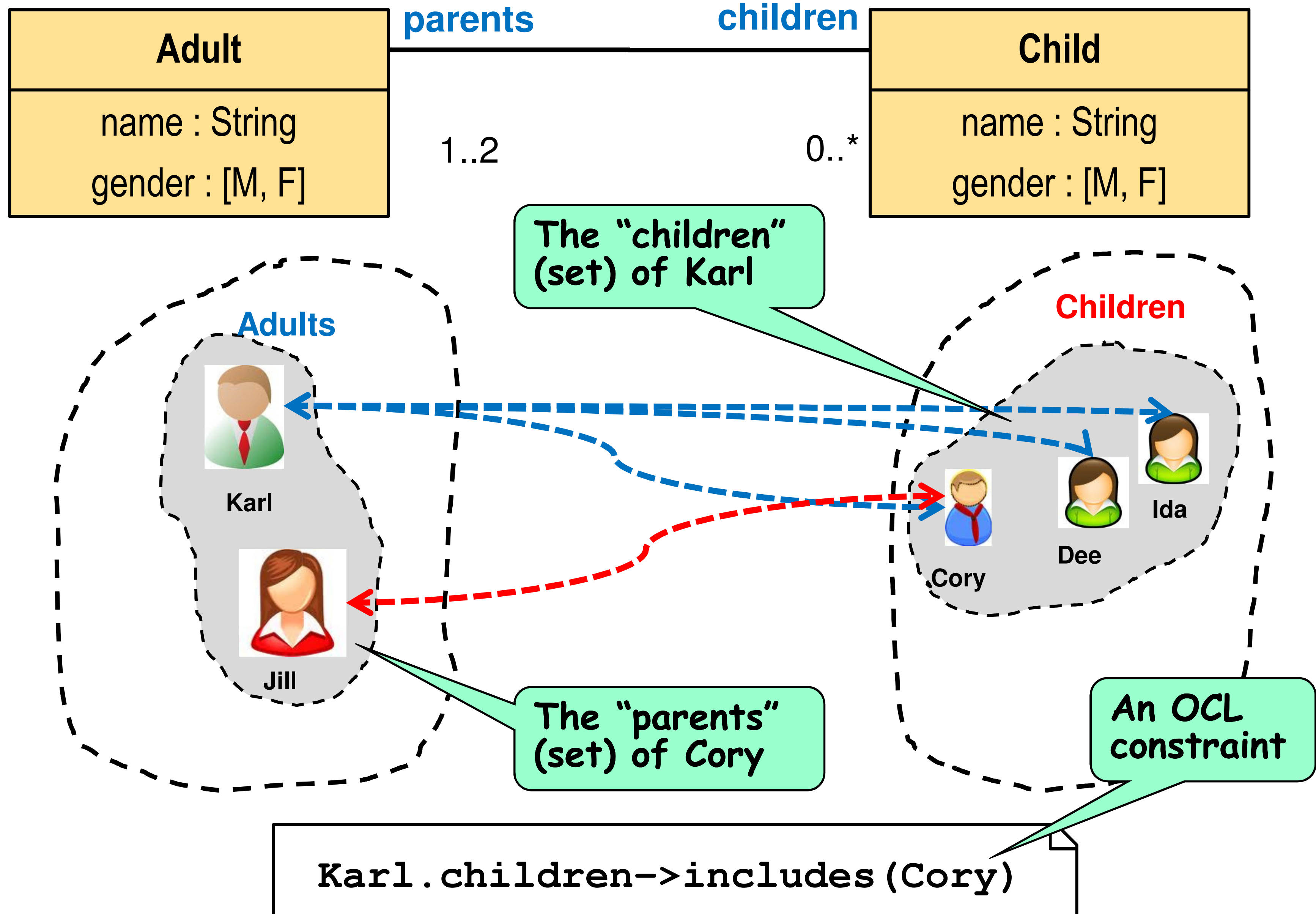
© Copyright Malina Software

Classes and Association Classifiers



© Copyright Malina Software

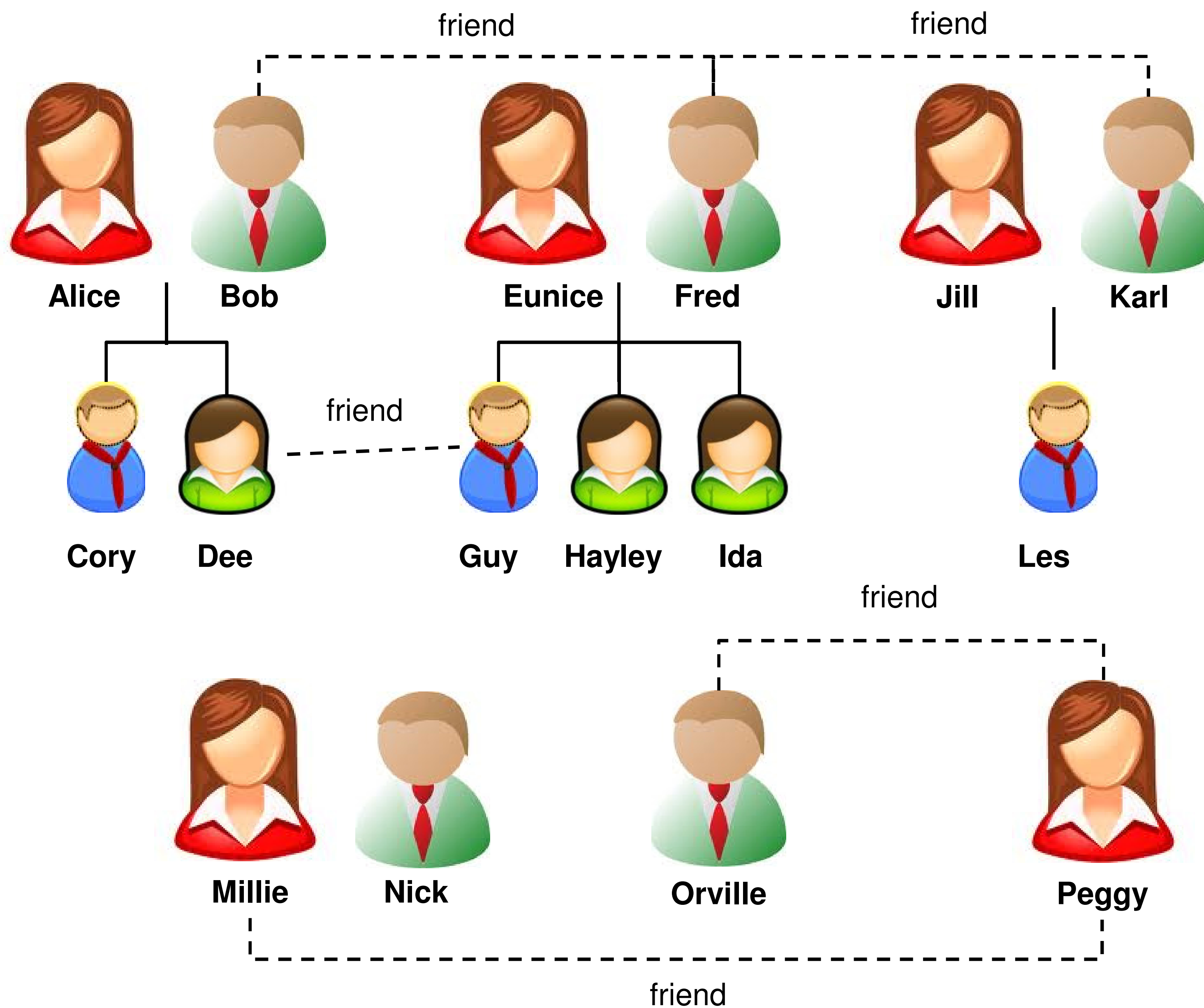
Interpreting Association Ends



© Copyright Malina Software

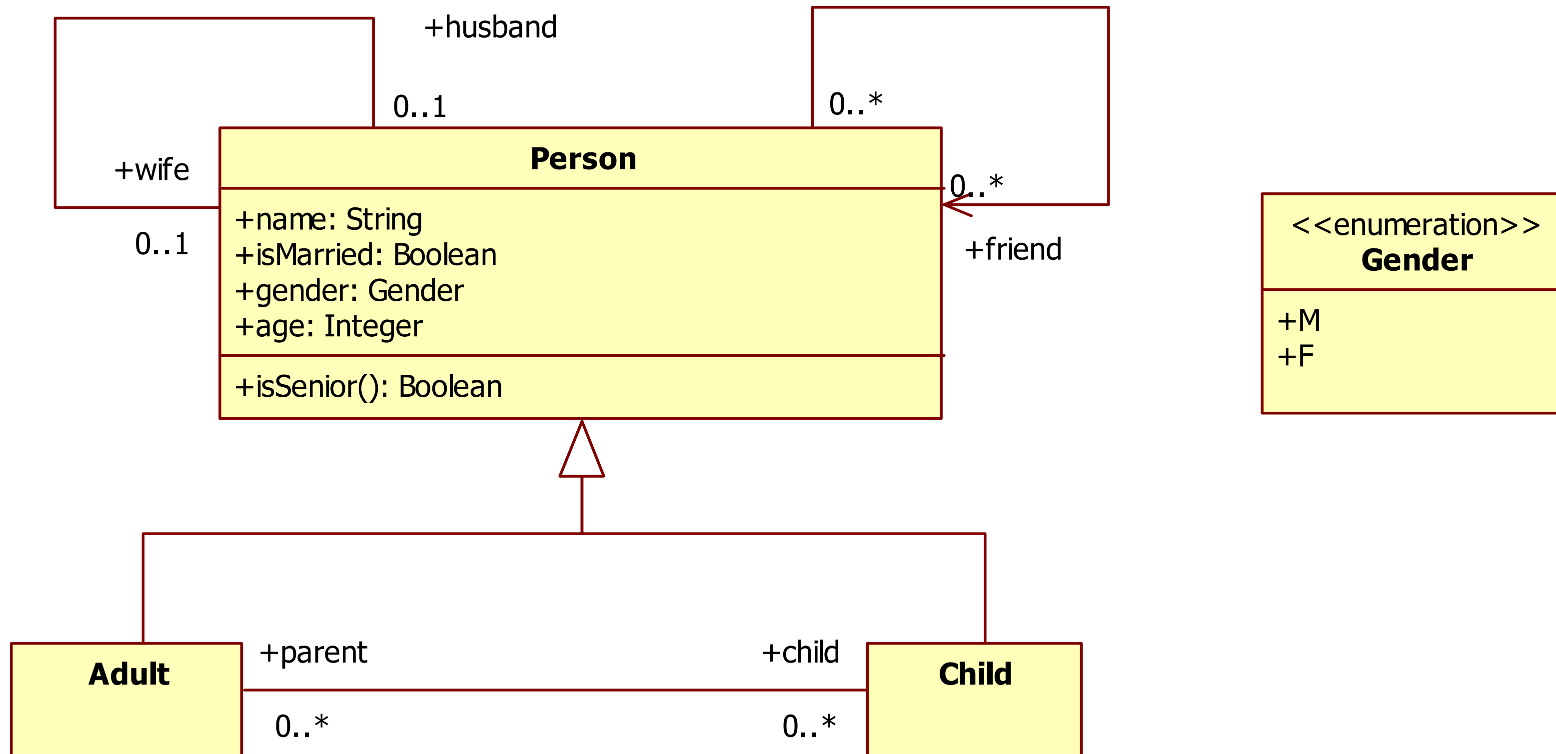
© Copyright Malina Software

Example: A Community



© Copyright Malina Software

A Corresponding Class Diagram



© Copyright Malina Software

Propositions, Predicates, and Constraints

- Proposition: A statement that is either True of False
  - Bob is an Adult
  - Les and Ida are friends
- Predicates: Propositions that involve variables; e.g.:
  - There is at least one Adult with the name "Bob"
  - All Adults are married
  - Every Child has at least one Adult parent
- A predicate require a range for its variables
- Constraints: predicates that, by design, must evaluate to True; e.g.:
  - Only Adults can have Children
  - An Adult who is married must have a spouse

© Copyright Malina Software

First-Order (Predicate) Logic(s)

- Used to reason about predicates
- Basic operators of FOL:
  - The usual Boolean operators
    - AND ($\wedge$)
    - OR ($\vee$)
    - NOT ($\neg$)
  - Conditional:
    - If <predicate-1> is True (hypothesis) then <predicate-2> (conclusion) must be True
    - <predicate-1> $\rightarrow$ <predicate-2>
  - Existential quantifier ($\exists$):
    - There exists at least one member of the domain such that predicate <predicate> is True
    - $\exists$ a $\in$ Dom | <predicate>
  - Universal quantifier ($\forall$):
    - For all members of the specified domain <predicate> is True
    - $\forall$ a $\in$ Dom | <predicate>

© Copyright Malina Software

FOL Examples and OCL Equivalents

- There is at least <u>one Adult</u> with the name "Bob"
  - $\exists\ a \subset Adult \mid (name(a) = \text{"Bob"})$
- All <u>Adults</u> are married
  - $\forall\ a \subset Adult \mid (married(a) = True)$
- Every <u>Child</u> has at least one Adult parent
  - $\forall\ c \subset Child \mid (size(parents(c)) \geq 1)$
- ...and their OCL equivalents
  - `exists (a:Adult| a.name = "Bob")`
  - `forAll (a:Adult | a.isMarried)`
  - `forAll (c:Child | parents->size() >= 1)`

© Copyright Malina Software

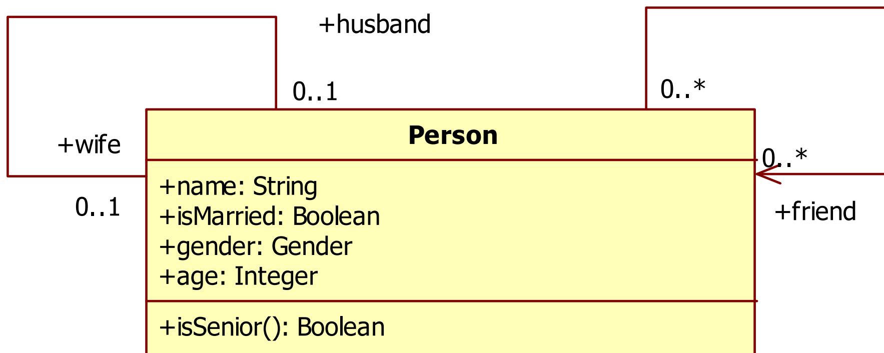
The Object Constraint Language (OCL)

- An OMG standardized language for specifying constraints and queries for UML and MOF <u>classifiers and objects</u>
  - http://www.omg.org/spec/OCL/2.2/PDF
  - Declarative side-effect-free language
- Primarily used in conjunction with MOF to specify the abstract syntax of modeling language constructs
  - Example:

```
context Child inv:
self.parent->size() <= 2
```


© Copyright Malina Software

OCL Basics - Contexts and Constraints



- **Context: identifies the class (or object) to which the constraint (also called an invariant) applies**

```
context Person inv: ...
```

- **Class constraints are written from the perspective of a generic member of the context class**
  - ...which means that they apply to all members of the class

```
((self.isMarried) and (self.gender = #M))
   implies((self.wife->size() = 1) and
         (self.husband->size () = 0))
```


© Copyright Malina Software

OCL Basics - Data Types Used in OCL

- **Reuses basic UML/MOF primitive types**
  - Boolean, Integer, String
  - Adds type Real
  - Support all common arithmetic and logic operators
- **Collection types**
  - Set, OrderedSet, Bag, Sequence
- **Model types**
  - Modeler-defined application-specific classes
  - E.g., Person, Adult, Child, Gender
- **OclType = the type of all types (metatype)**
  - Useful operation on any type: allInstances()

```
Person.allInstances() -- returns the set of all
                      -- instances of Person
```


© Copyright Malina Software

OCL Basics - Standard Arithmetic Operators

- <numeric-expr-1> **<** <numeric-expr-2>
- <numeric-expr-1> **>** <numeric-expr-2>
- <numeric-expr-1> **<=** <numeric-expr-2>
- <numeric-expr-1> **>=** <numeric-expr-2>
- <numeric-expr-1> **+** <numeric-expr-2>
- <numeric-expr-1> **-** <numeric-expr-2>
- <numeric-expr-1> ***** <numeric-expr-2>
- <numeric-expr-1> **/** <numeric-expr-2>
- <numeric-expr-1>**.mod**(<numeric-expr-2>**)**
- ...

© Copyright Malina Software

OCL Basics - Common Logic Operators

- **not** <Boolean-expr>
- <Boolean-expr-1> **or** <Boolean-expr-2>
- <Boolean-expr-1> **and** <Boolean-expr-2>
- <Boolean-expr-1> **xor** <Boolean-expr-2>
- <expression-1> **=** <expression-2>
- <expression-1> **<>** <expression-2>
- <Boolean-expr-1> **implies** <Boolean-expr-1>
- **if** <Boolean-expr-1> **then** <expression-2> [**else** <expression-3> **endif**]

© Copyright Malina Software

OCL Basics - OclAny

- Supertype of all types
  - Not to be confused with OclType (which is a metatype)
    - The type of OclAny is OclType
- Defines a useful set of operations that can be applied to any primitive or user-defined object

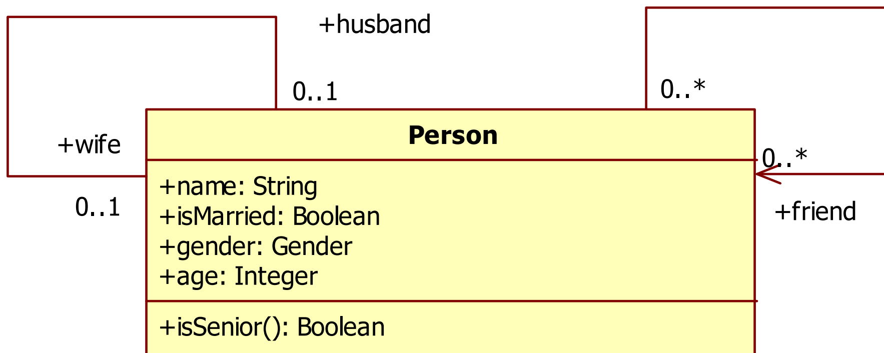
```
oclType()           -- returns the type of an object

oclIsTypeOf(<type>) -- returns True if object is of
                       -- type <type>

oclIsKindOf (<type>)-- returns True if object is of
                       -- type <type> or its subtype
```


© Copyright Malina Software

OCL Basics - Accessing Properties



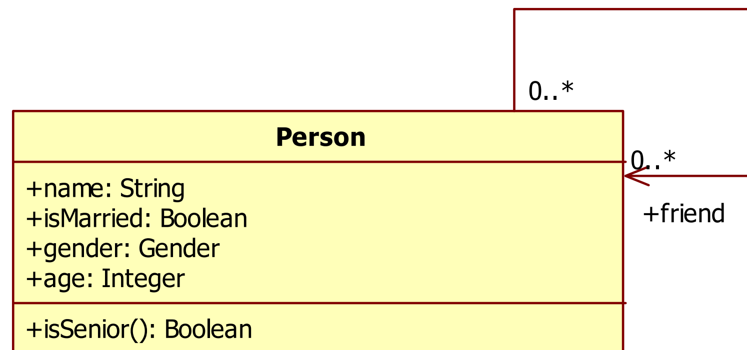
- Classifier (and object) attributes and operations are accessed by the dot (.) operator

```
self.name          -- returns name String
self.isSenior()    -- returns True or False
self.friend        -- returns a collection
                   -- persons who are friends
                   -- of "self"
```


© Copyright Malina Software

OCL Basics - Association/Link Navigation

- Association ends are accessed like all other Properties of Classifiers
- OCL can navigate from a Classifier context to any outgoing association end...regardless of navigability

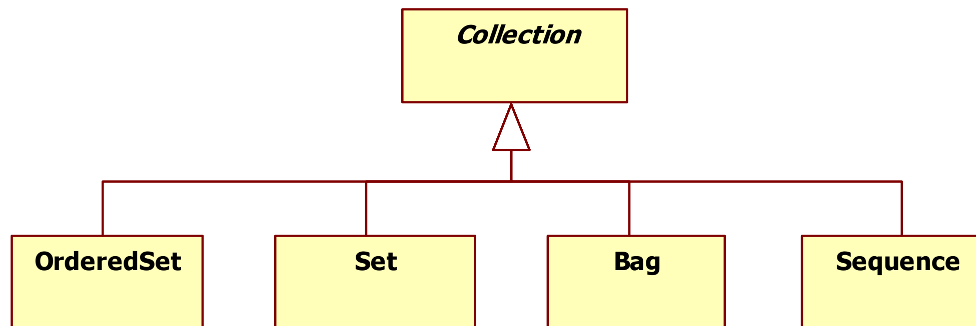



```
self.friend
self.person      -- uses default naming rule for
                 -- unnamed ends
```

- Unless the multiplicity is exactly 1, the result of the navigation is a collection

© Copyright Malina Software

OCL Basics – OCL Collections and Operations



- Collections represent groups of values (e.g., Classes)
- Collections can be manipulated using special collection operations
  - `size()` -- the size of the collection (Integer)
  - `count(<value>)` -- the number of occurrences of <value> (Integer)
  - `includes(<value>)` -- True if collection includes <value>
  - `includesAll(<collection>)` -- True if collection includes <collection>
  - `isEmpty()` -- True if collection is empty
  - `notEmpty()` -- True if collection is not empty
  - `exists(<expression>)` -- True if <expression> is True for at least 1 element
  - `forAll(<expression>)` -- True if <expression> is True for all elements
  - ...

© Copyright Malina Software

OCL Basics – Applying Operations to Collections

- The application of an operation to a collection is indicated by the use of the right-arrow (->) operator
  - `self.friend->size()` -- number of friends of self
  - `self.friend->isEmpty()` -- checks if set of -- friends is empty

© Copyright Malina Software

OCL Basics – Universal and Existential Quantifiers

- "exists" and "forAll" operations are used to specify predicates over collections
  - exists = ∃ (first-order logic existential quantifier operator)
    - `self.friend->exists (f:Person | f.name = 'Bob')`
      `-- at least one friend must be named 'Bob'`
  - forAll = ∀ (first-order logic universal quantifier operator)
    - `Person.allInstances()->forAll(p:Person| p.name <> '')`
      `-- the name of a Person cannot be an empty string`
  - Avoids confusing mathematical symbols (+ avoids need for special typesetting)

© Copyright Malina Software

OCL Basics - Select and Collect Operations

- Special iteration operations for deriving useful new collections from existing ones
- Select provides a subset of elements that satisfy a predicate
  - `<collection>->select (<element> : <type> | <expression>)`
  - `Person->select (p:Person | p.isMarried)`
- Collect returns a new collection of values obtained by applying an operation on all of the elements of a collection
  - `<collection>->collect (<element> : <type> | <expression>)`
  - `Person->collect (p:Person | p.name)`

© Copyright Malina Software

OCL Basics - Other Useful Collection Operations

- For Sets, Bags, Sequences
  - ```
    <Coll-1>->union (<Coll-2>)    -- returns the union of
                                  -- <Coll-1> and <Coll-2>
    ```
  - ```
    <Coll-1>->intersection(<Coll-2>) -- returns the
                                  -- intersection of
                                  -- <Coll-1> and <Coll-2>
    ```
  - ```
    <Coll-1> - (<Coll-2>)         -- returns a collection of
                                  -- elements in <Coll-1>
                                  -- that are not in <Coll-2>
    ```
- For OrderedSets and Sequences
  - `<Coll>->at(i)     -- access element at position i`
  - `<Coll>->append(<object>) -- add <object> to end of <Coll>`
  - `<Coll>->first()   -- return first element in <Coll>`
  - `<Coll>->last()    -- return last element in <Coll>`

© Copyright Malina Software
    ```

OCL Basics - Pre- and Post-Conditions

- "Design by contract" for Operations, Receptions, and UML Behaviors
  - Pre-conditions: Conditions that must hold before an operation
  - Post-conditions: Conditions that must hold after an operation has completed
- Syntax:
  - `pre: <Boolean-expr>`
  - `post: <Boolean-expr>`
  - Also can use `<attribute>@pre` in a post-condition to refer to the value of `<attribute>` prior to the execution of the operation

© Copyright Malina Software

OCL Basics - Defining Operations

- **Syntax**

```
context <operation-name>(<parameters>) : <return-
  type>
   [pre: <Boolean-expr>]
   [post: <Boolean-expr>]
   [body: <expression>]   -- must evaluate to
                          -- a kind of <return-type>
```

- **Example**

```
context Person::isSenior() : Boolean
    pre: age >= 0
    post: age = age@pre  -- age is unchanged
                         -- after operation
                         -- completes
    body: (age >= 65)
```

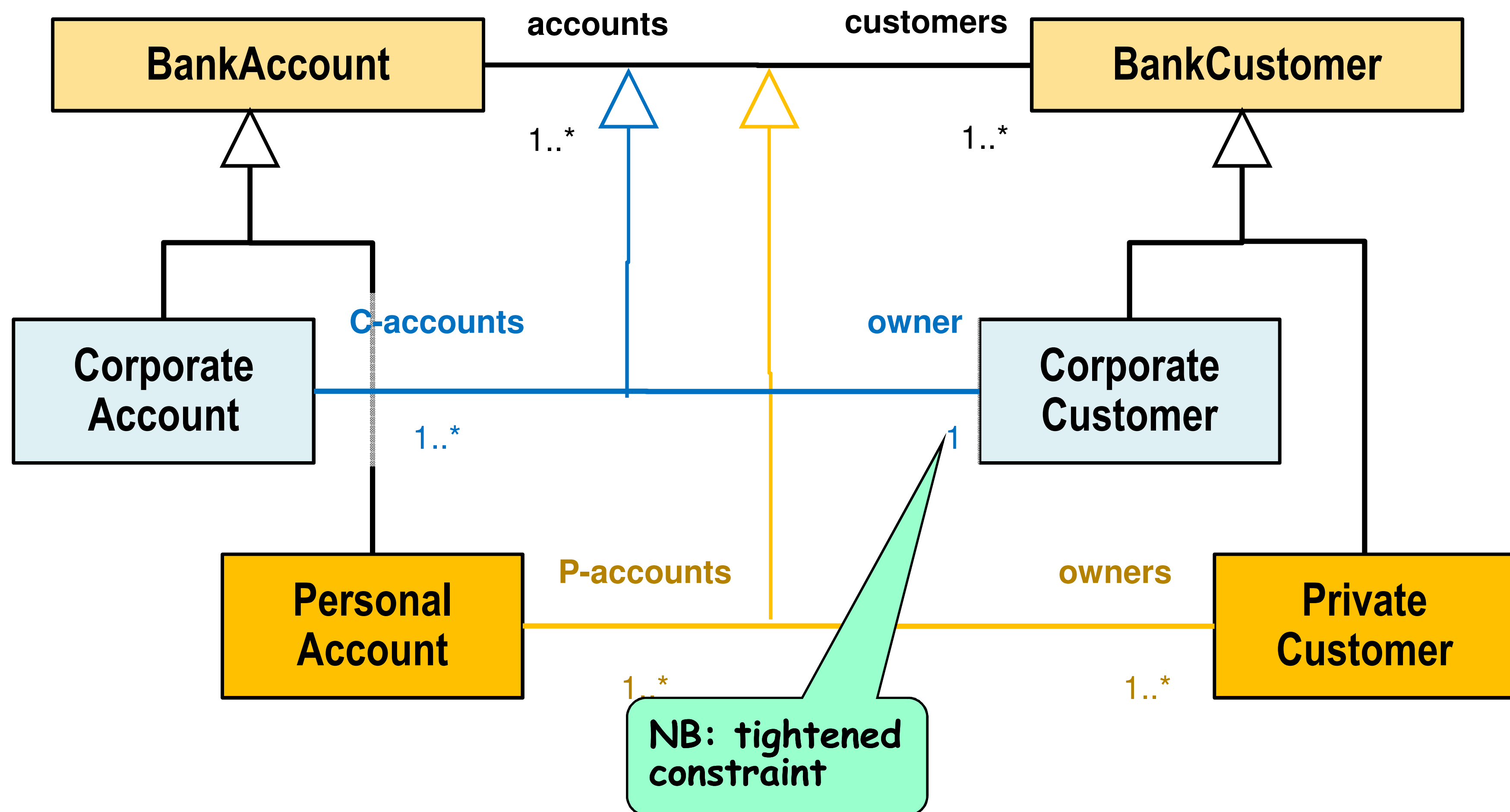

© Copyright Malina Software

Summary: OCL

- OCL constraints are one of the means to define well-formedness (i.e., syntactic) rules for MOF-based models
  - Complement class (meta)models
- It is based on basic first-order logic and set theory and operates on class (and instance) diagrams
  - Since class diagrams deal with relationships between sets and elements of sets
- Defines primarily static semantics but can also be used to specify dynamic semantics (e.g., through pre- and post- statements on operations)

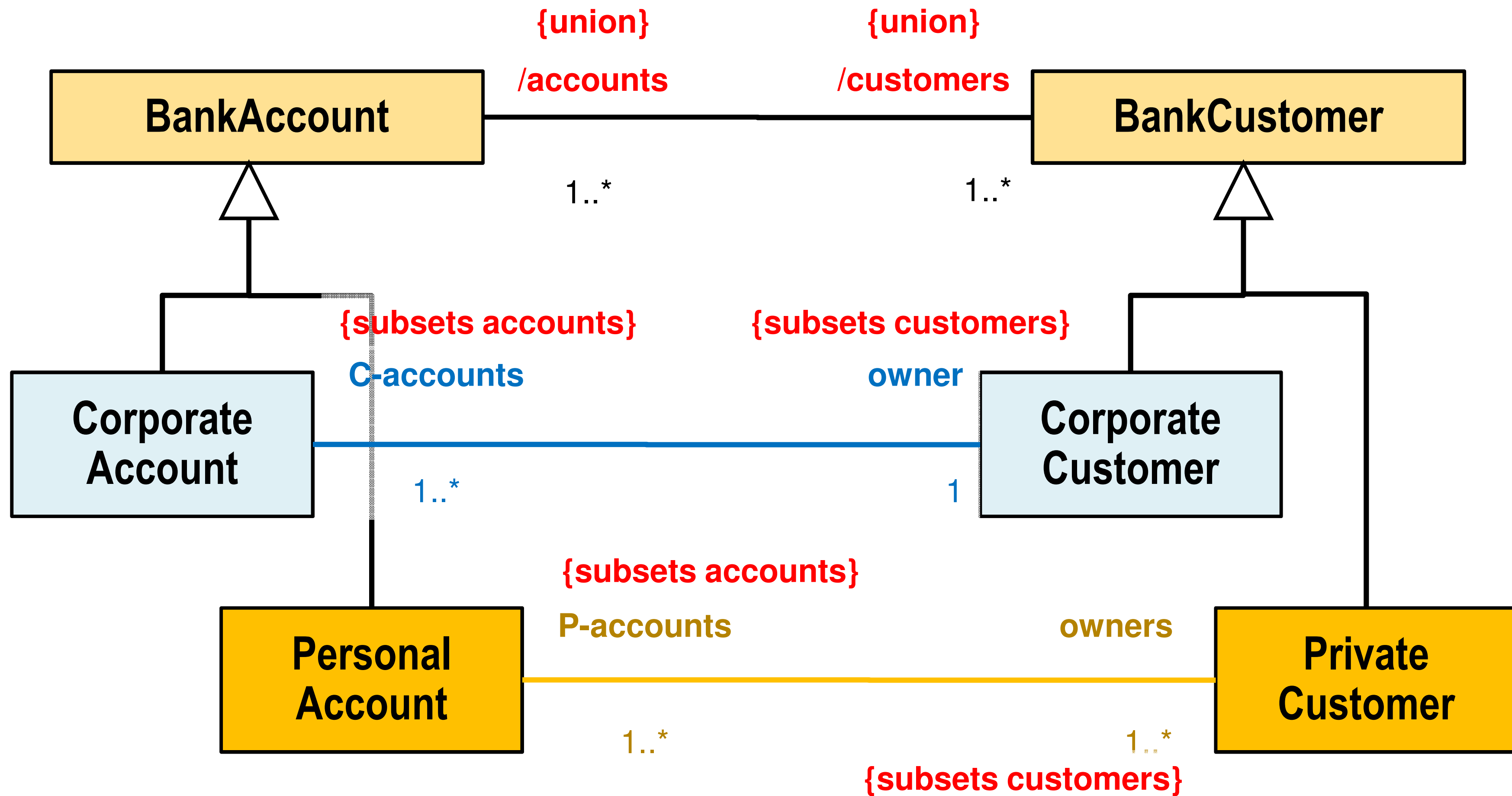
© Copyright Malina Software

MOF Mechanism: Association Specialization



© Copyright Malina Software

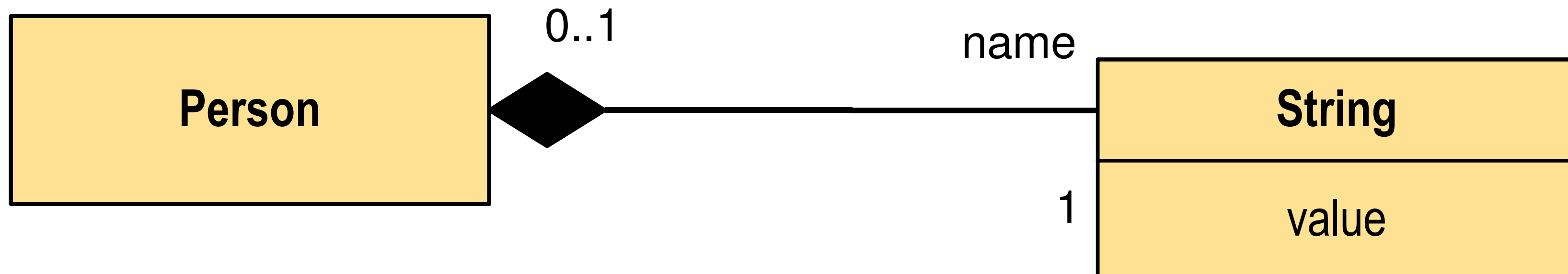
More Refined Specification



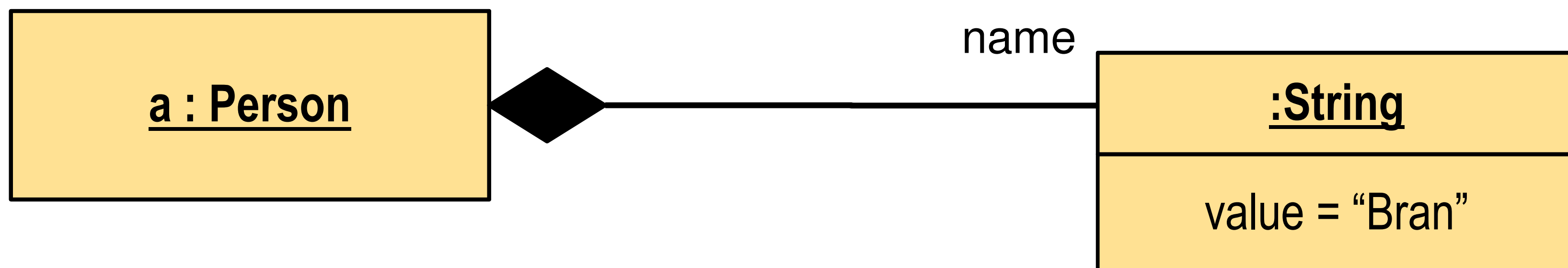
© Copyright Malina Software

Composition ("Black Diamond") Associations

- **Implies "ownership" of one element by another**
  - i.e., An instance of Person owns an instance of String that specifies that person's "name"



- **Semantics: "Deletion" semantics**
  - When the owner is removed, all its owned elements are also removed with it



© Copyright Malina Software

Key MOF Abstraction: Names and Namespaces

- **Names are specified by Strings**
  - No pre-defined character set or size limit
  - E.g.: "Alice", "R2->D2", "4 Sale", "Селић", "多音字", ""
  - NB: An empty name is a valid name
- **A namespace is a model element that owns a collection of model elements (that may be) distinguished by their names**
  - The features (attributes, operations, etc.) of a Class
  - Used as a basis for specifying other MOF concepts: Package, Class, Operation, etc.
- **General rules (may be further constrained in a profile)**
  - Names in a namespace are not necessarily unique (but is preferred)
    - E.g. Two operations may have the same name but different parameter sets
    - E.g.: an operation and an attribute of a class can have the same name
  - Namespaces can contain other namespaces ⇒ hierarchical (qualified) names
    - Use of double-colon (::) to separate names in a qualified name
    - E.g.: "System::CoolingSubsystem::ACUnit"

© Copyright Malina Software

The Concept of Visibility

- Named elements owned by a MOF namespace have a defined visibility:
  - The capacity for an element to be referenced by other model elements (possibly in other namespaces)
- Pre-defined visibility kinds:
  - public (+) – named element is visible to all elements that can access its namespace
  - private (-) – visible only within the namespace that owns it
  - protected (#) – visible only to elements in its *extended* namespace (i.e., its namespace and the namespaces of all specializing classifiers – for classifier type namespaces only)
    - E.g., a Class attribute visible to all subclasses of that Class
  - package – (~) visible only to other elements in the same package
    - e.g., a Class attribute visible to all elements in the same Package as the Class

© Copyright Malina Software

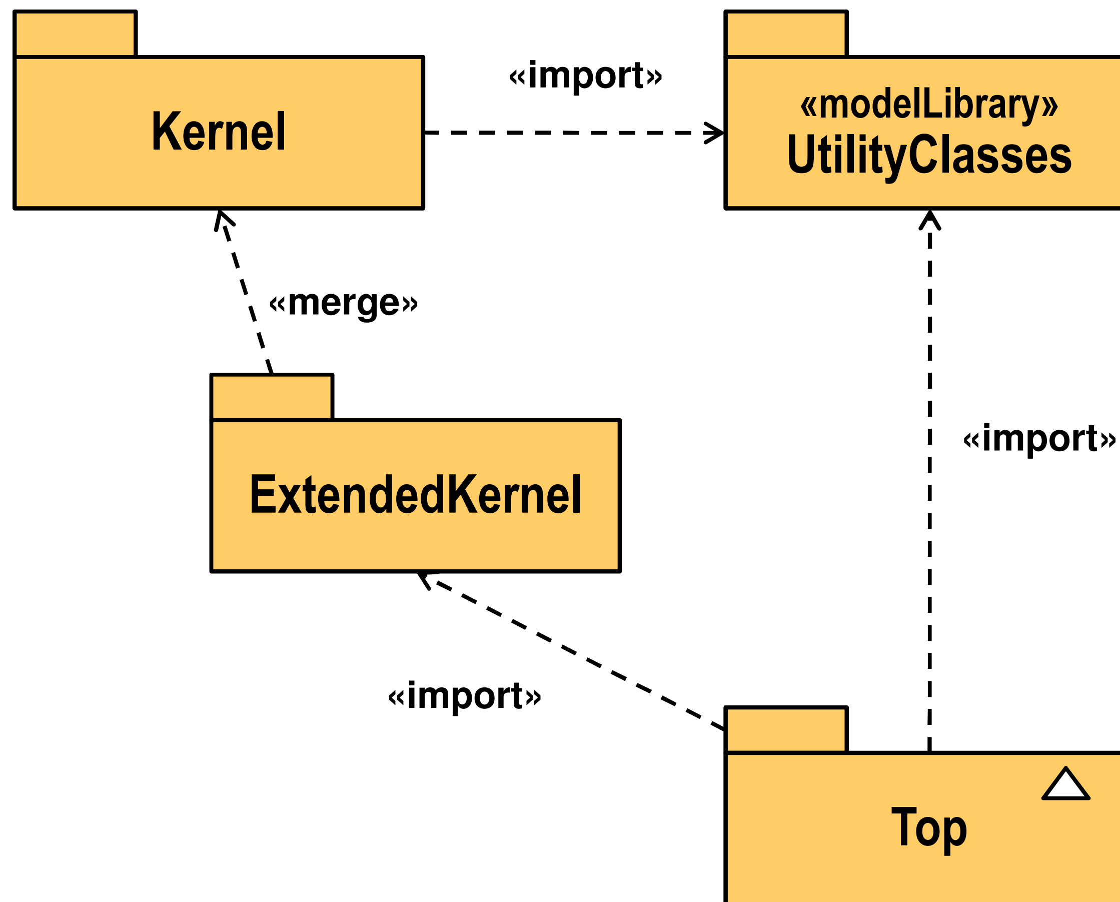
MOF Packages

- A Package is a means for grouping the parts of a (meta)model
  - Packages are (typically) not intended to model anything
  - Analogous to file system folders
- A package is a kind of namespace
  - Public elements from other packages can be imported into the package namespace (analogous to "external" declarations in programming)

© Copyright Malina Software

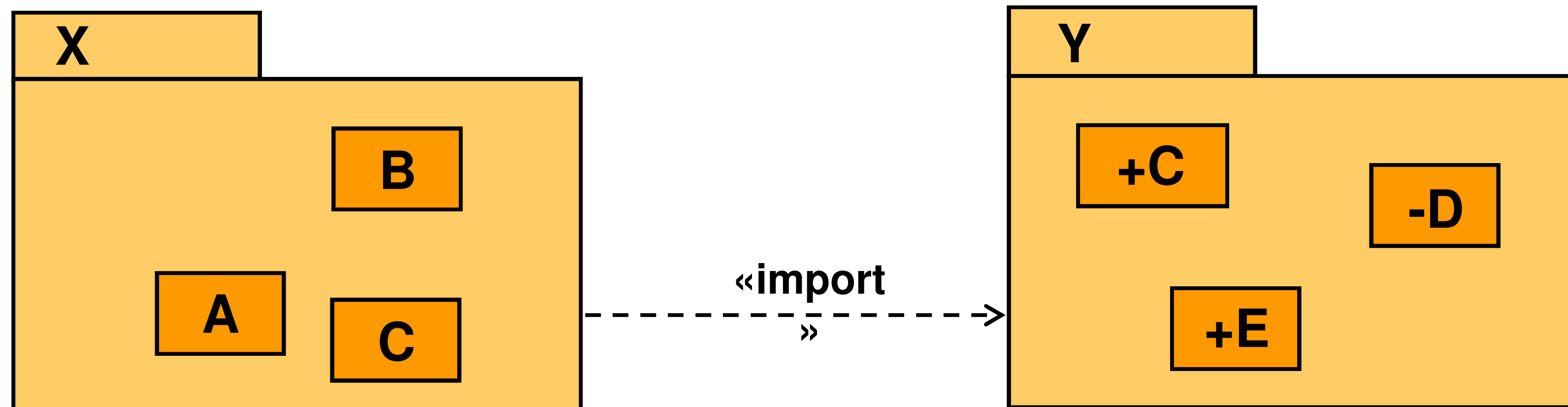
Package Diagrams

- **Show relationships (import, merge) between packages in a model**
  - *A design-time view*



© Copyright Malina Software

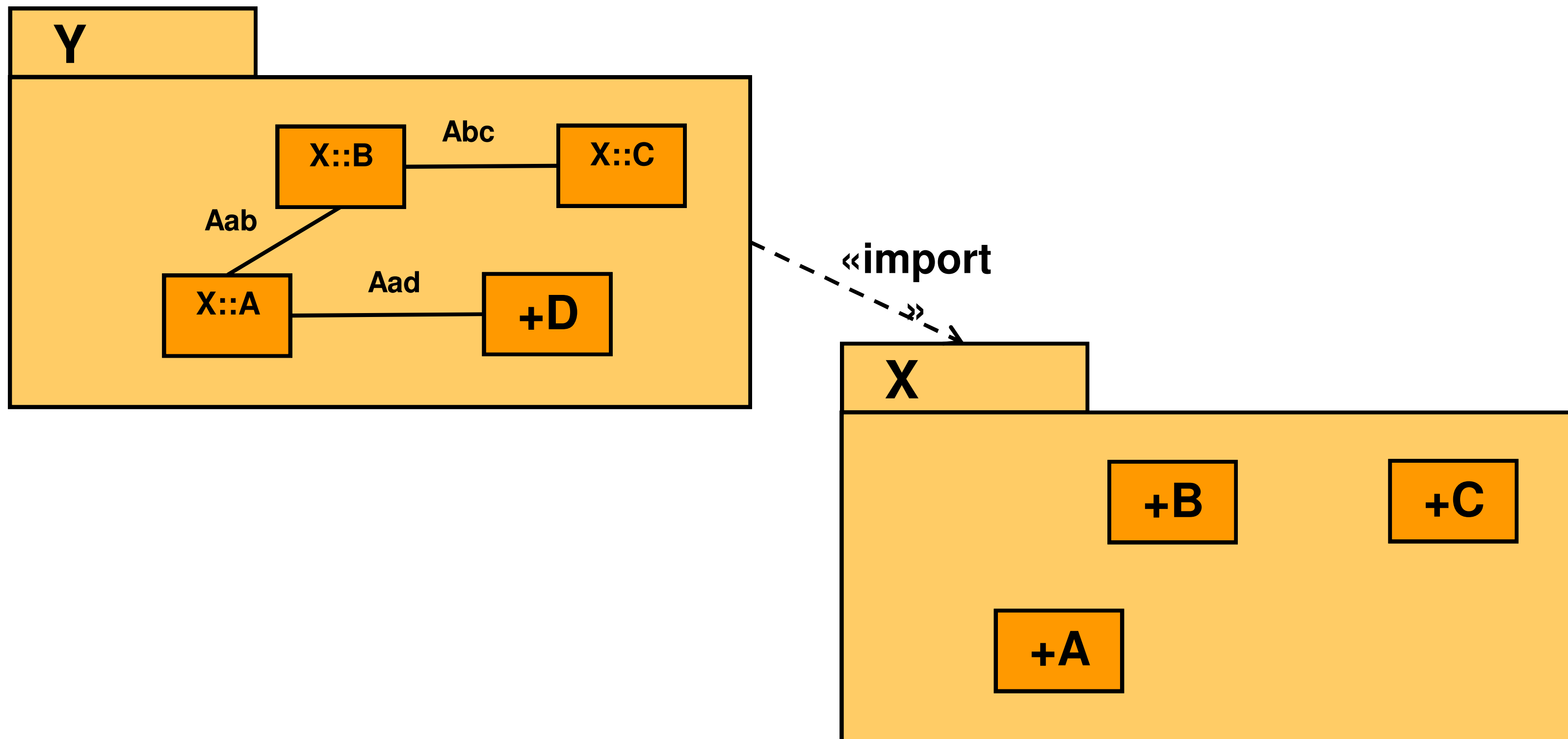
Package Import Semantics



- Following importing, namespace X contains the following names:
  - A, B, C, Y::C, E
  - ...but not D
- However, Y::C and E are not owned by X

© Copyright Malina Software

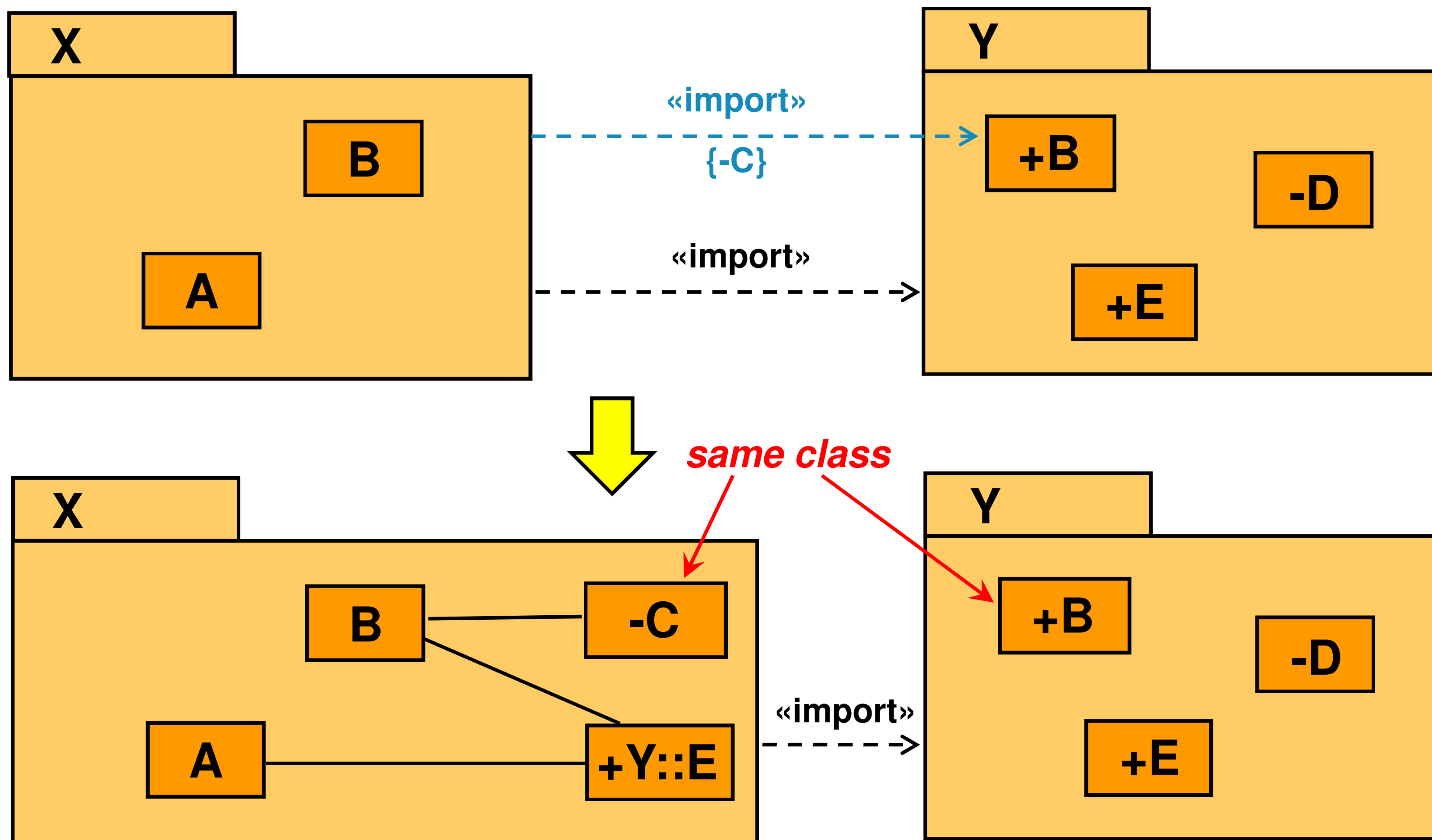
Package Import Semantics (cont'd)



- Y owns D and the associations

© Copyright Malina Software

Element Import

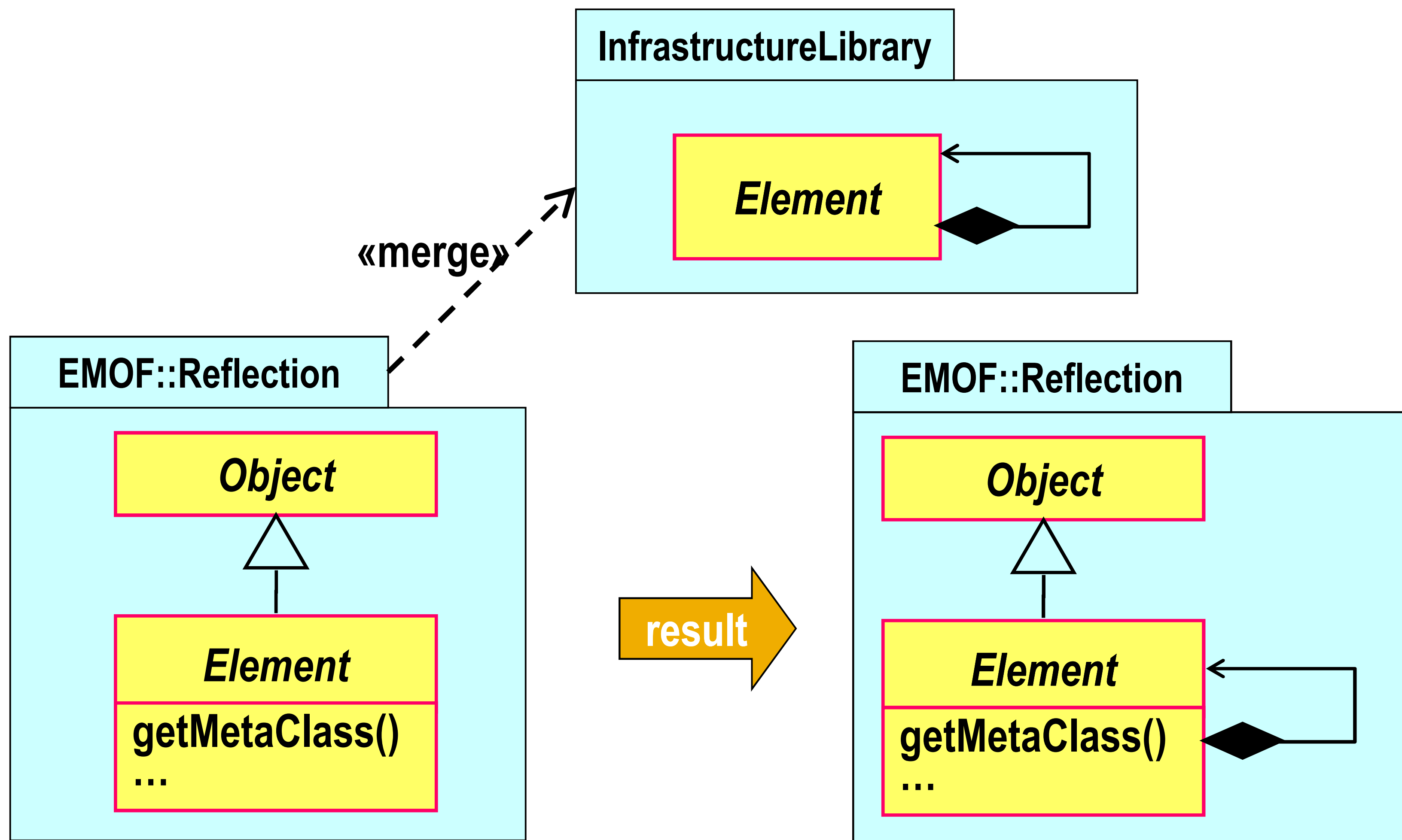


An imported element can be given a local alias and a local visibility

© Copyright Malina Software

Package Merge

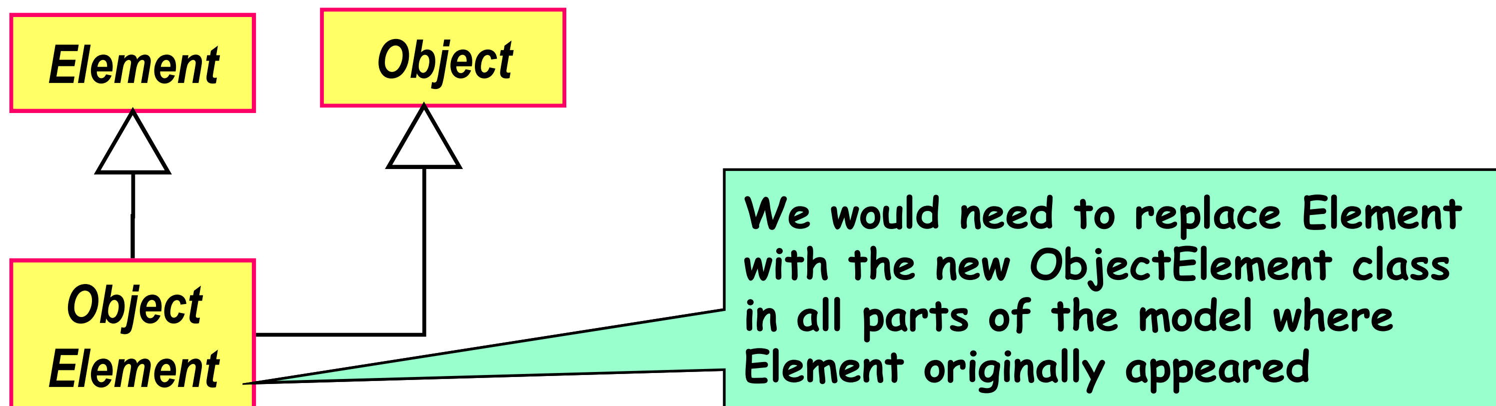
- Allows selective incremental concept extension



© Copyright Malina Software

Package Merge Semantics

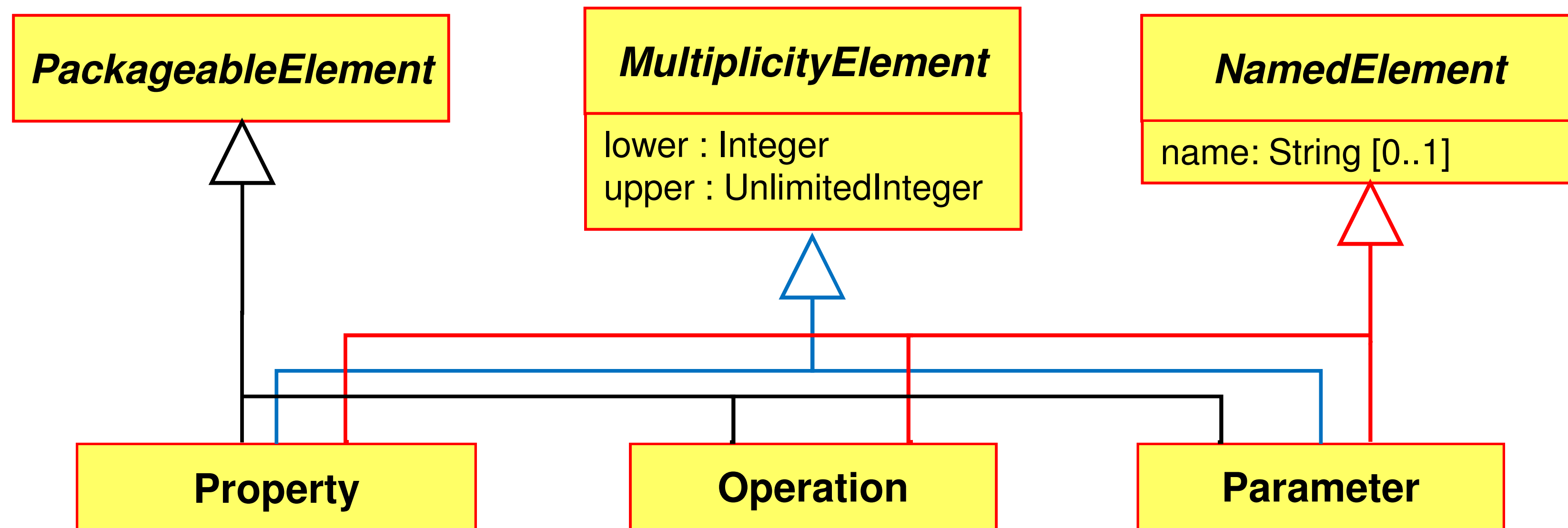
- A graphically-specified operation on packages and their contents
  - Extends definition and scope of a concept through an increment
- Why not just use subclassing?
  - The additional specifications apply to the original concept wherever it was used



© Copyright Malina Software

EMOF Metamodel: Modular (Mixin) Generalizations

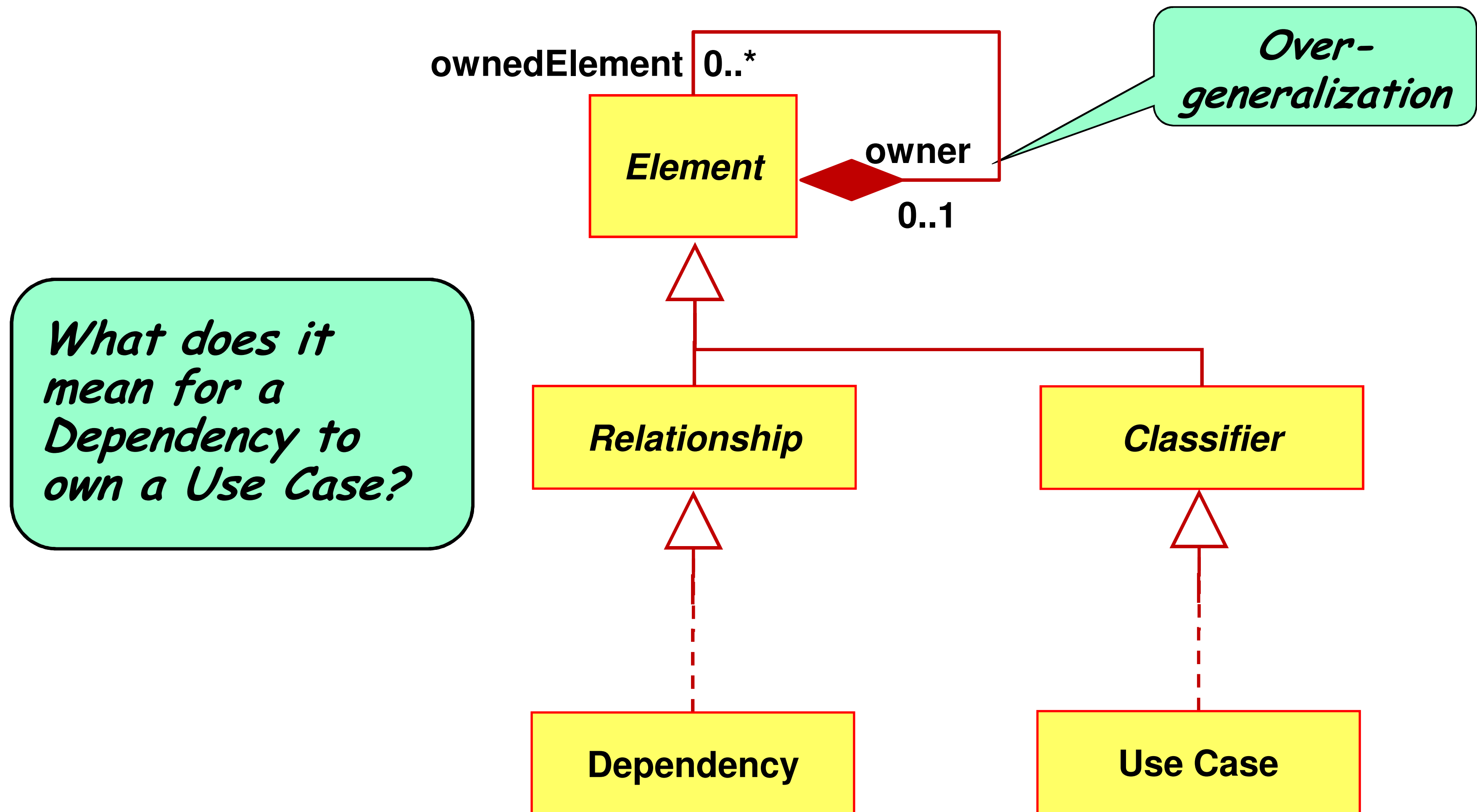
- A meta-modeling pattern for adding capabilities using specialized superclasses
  - Each increment adds a well-defined primitive capability
  - To be used with extreme caution because it can lead to semantic conflicts and overgeneralization problems
    - Especially if the mixin classes have associations



© Copyright Malina Software

Meta-Modeling Trap: Overgeneralization

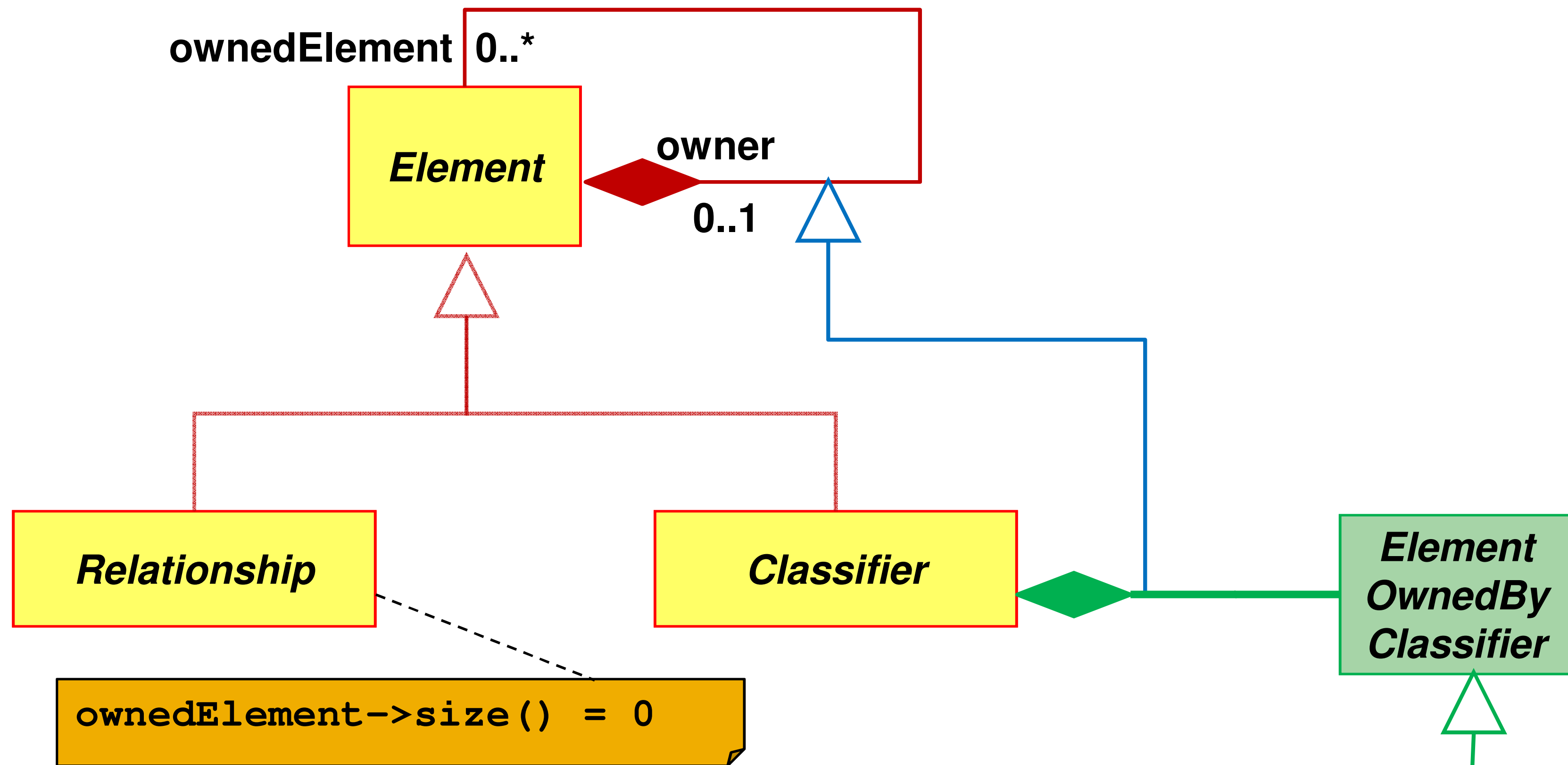
- Fragment of the UML 2 metamodel (simplified):



© Copyright Malina Software

Dealing with Overgeneralization

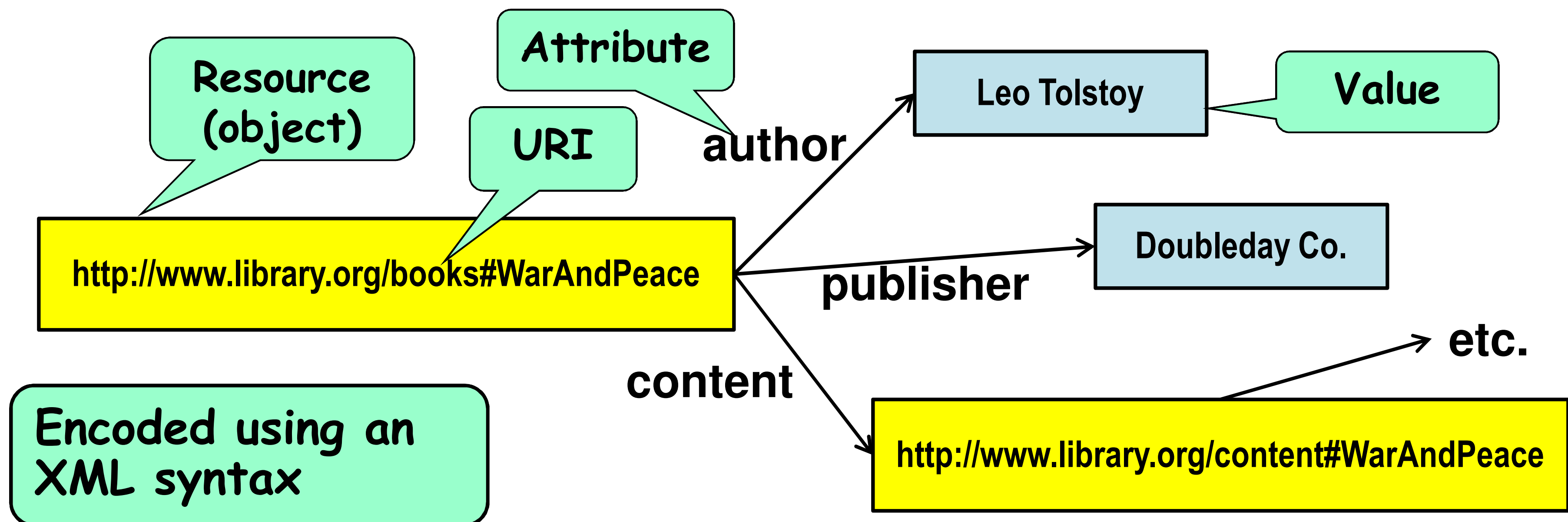
- Caused by "abstract" associations
- Can sometimes be avoided using association specialization (covariance) or constraints



© Copyright Malina Software

RDFS – An Alternative to MOF

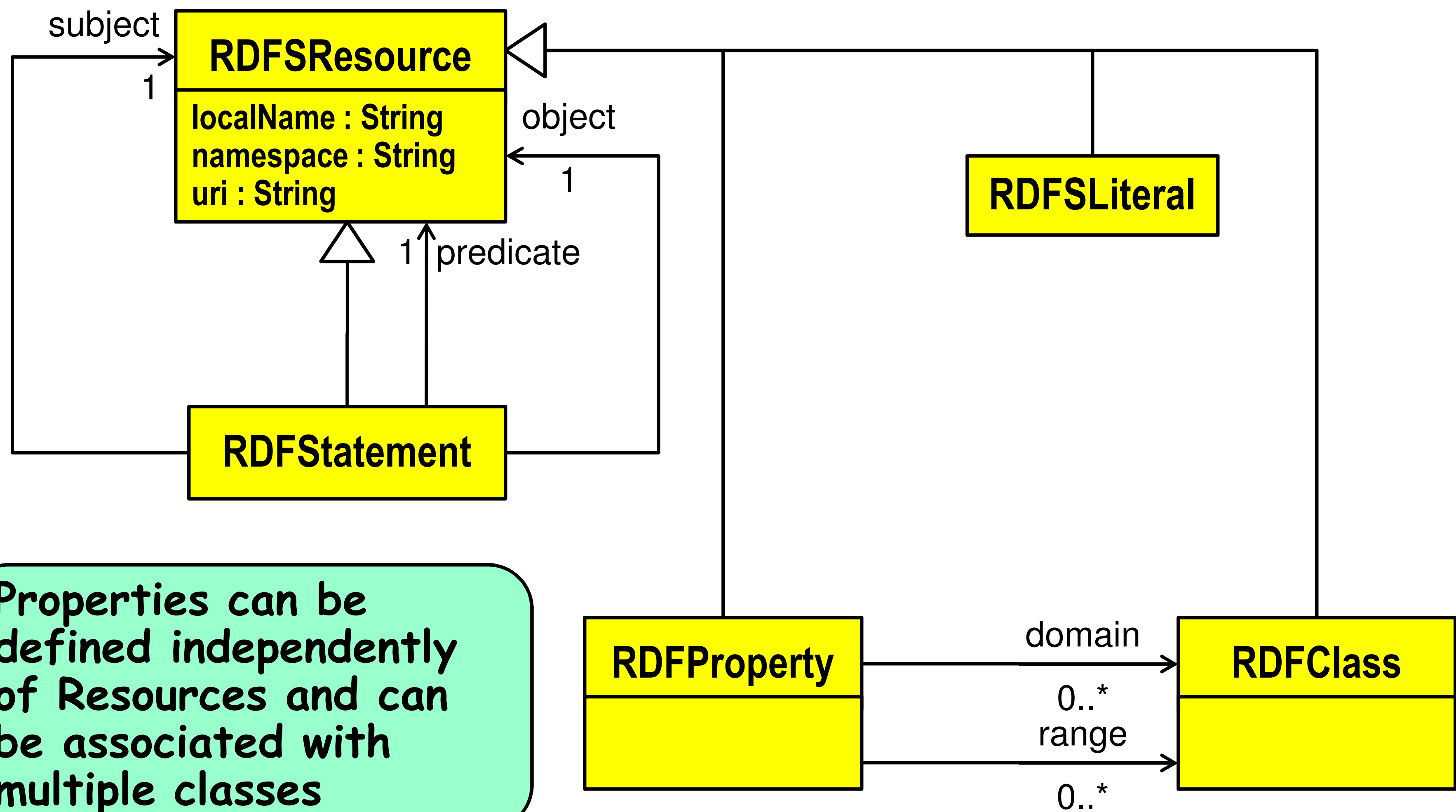
- Resource Description Framework Schema (RDFS)
  - A standardized format for knowledge (ontology) representation and interchange on the World-Wide Web
  - Standardized by the W3C consortium
  - http://www.w3.org/standards/techs/rdf
- Based on simple <object-attribute-value> paradigm:



© Copyright Malina Software

RDFS Metamodel (simplified)

- Everything is a resource (identified by a URI)



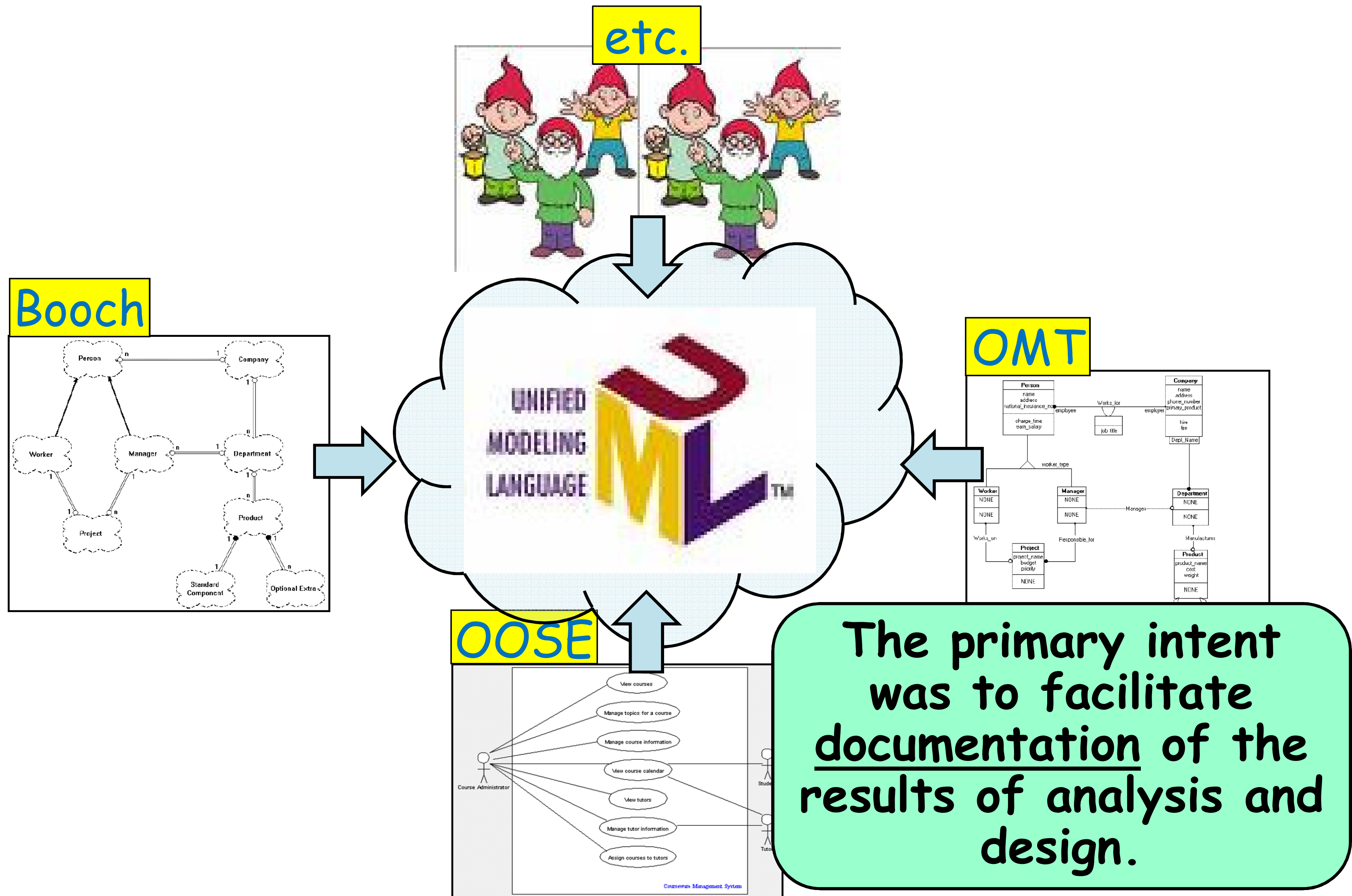
© Copyright Malina Software

Tutorial Outline

- On Models and Model-Based Software Engineering
- The Key Dimensions of Modeling Language Design
- Defining a Modeling Language
- Case Study: UML
- Language Refinement: UML Profiles
- Model Transformations

© Copyright Malina Software

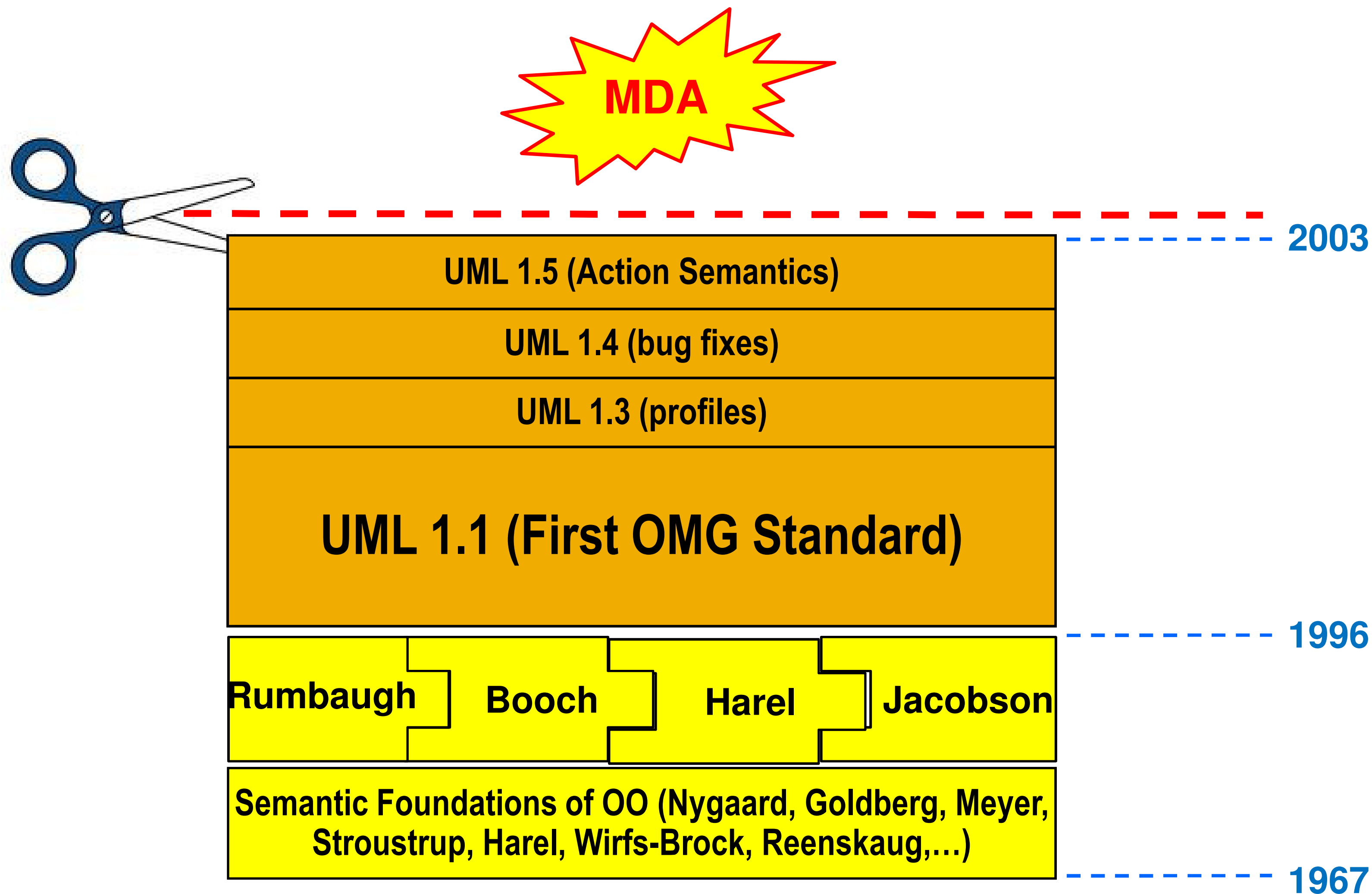
UML 1: The First Cut



The primary intent was to facilitate documentation of the results of analysis and design.

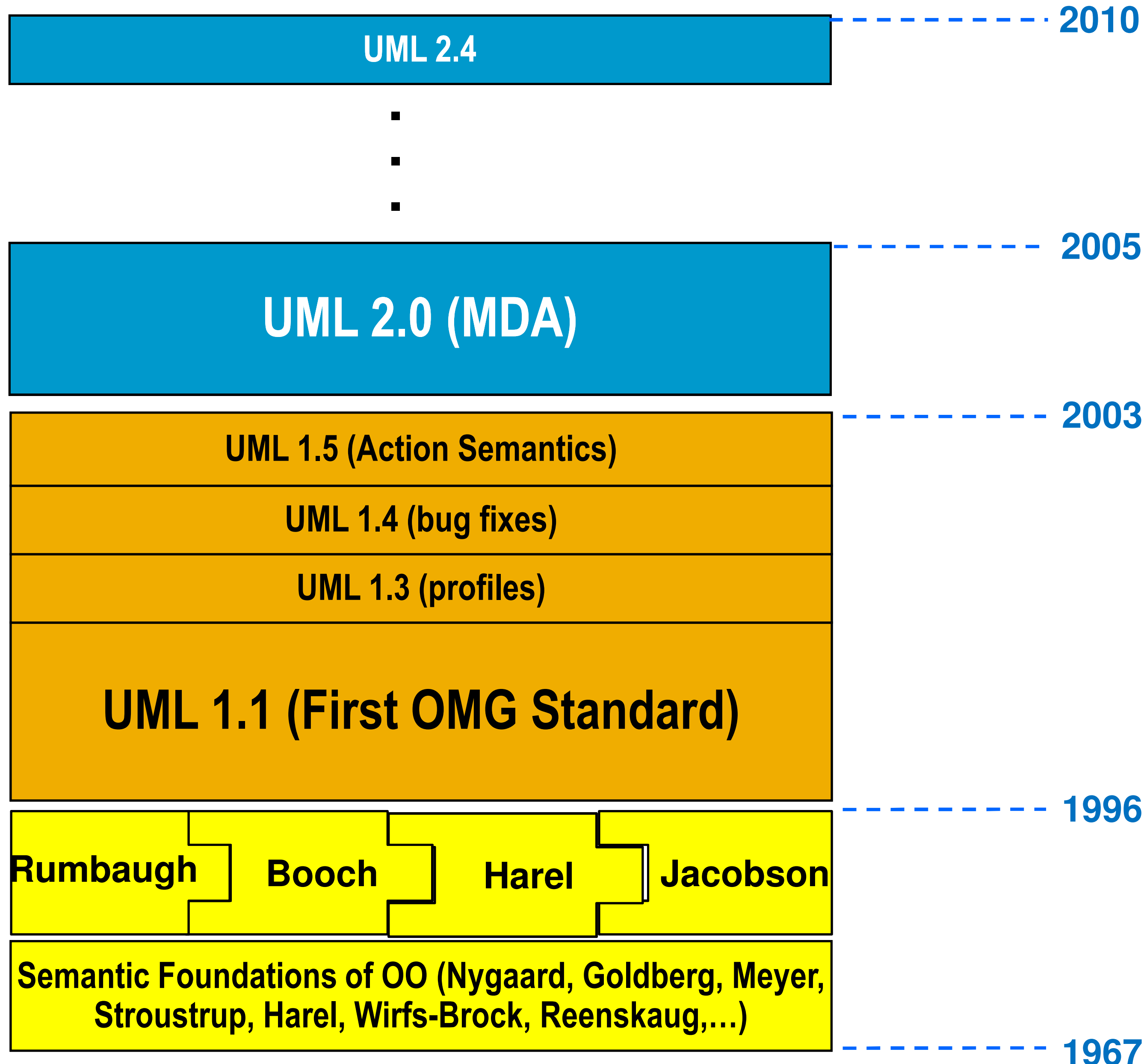
© Copyright Malina Software

UML Roots and Evolution: UML 1



© Copyright Malina Software

UML Roots and Evolution: UML 2



© Copyright Malina Software

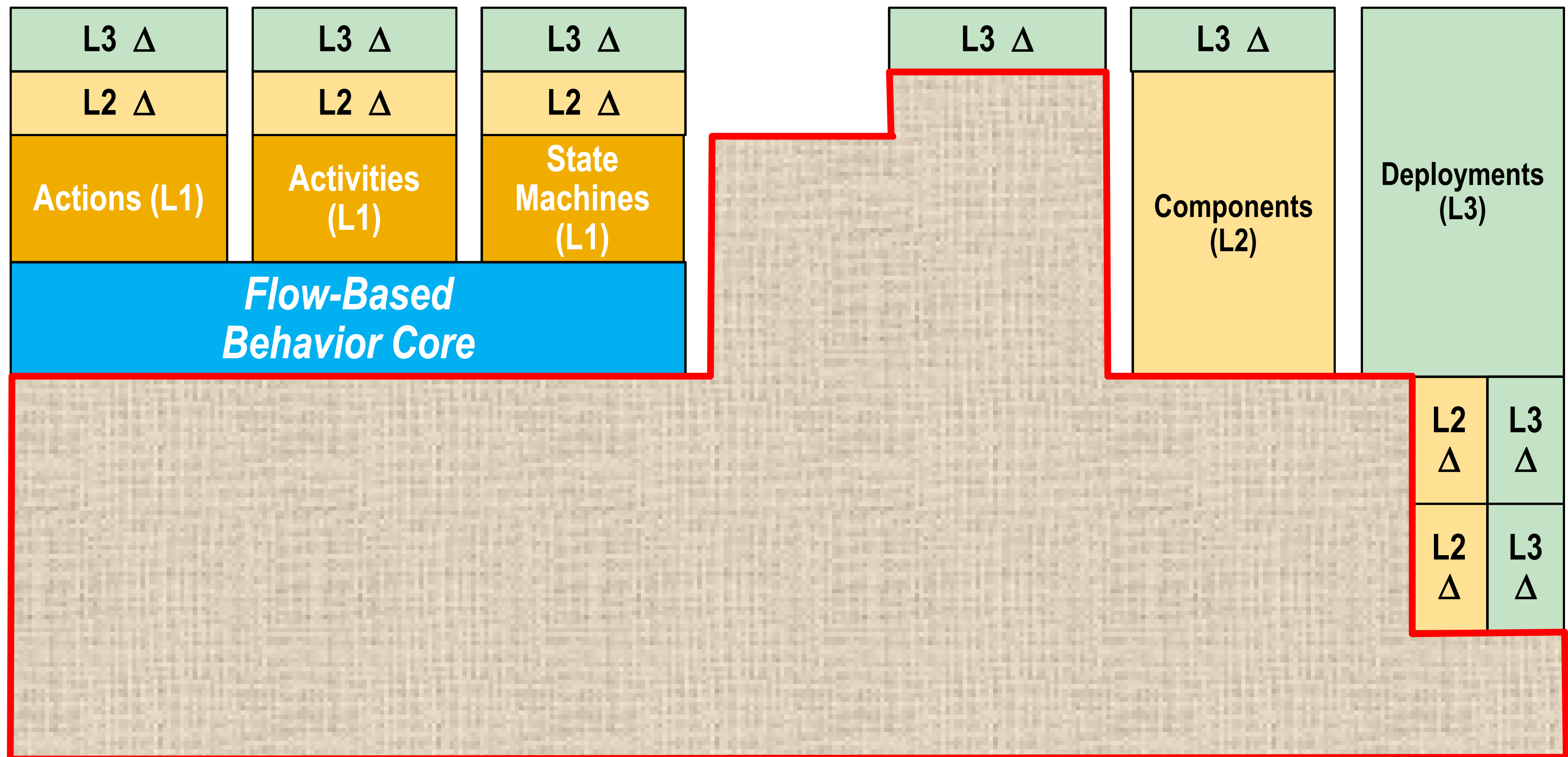
UML 1 vs UML 2

- **UML 1 was intended primarily as a design and documentation tool**
  - Informal and semi-formal definition
  - Overlapping and ambiguous concepts
- **UML 2 is intended to support MDA**
  - Much more formal and precise definition
    - *Executable UML Foundation*
  - Opens up potential to use UML as an implementation language
- **UML 2 added capabilities**
  - Highly-modular language architecture
  - Improved large-system modeling capability
    - Interactions, collaboration (instance) structures, activities all defined recursively for scalability
  - More powerful language extensibility capability

© Copyright Malina Software

UML 2 Language Architecture

- A user-selectable collection of different languages for different needs based on a set of shared conceptual cores
- Organized into user-selectable increments of increasing sophistication



© Copyright Malina Software

UML Language Compliance Levels

- **4 levels of compliance (L0 – L3)**
  - compliance(Lx) ⇒ compliance (Lx-1)
- **Dimensions of compliance:**
  - Abstract syntax (UML metamodel, XMI interchange)
  - Concrete syntax
    - Optional Diagram Interchange compliance
- **Forms of compliance**
  - Abstract syntax
  - Concrete syntax
  - Abstract and concrete syntax
  - Abstract and concrete syntax with diagram interchange
- ***However, this architecture has been deemed as too complex for implementation and the compliance levels and the use of package merge in the definition of UML are being eliminated in UML 2.5 (due in 2012)***

© Copyright Malina Software

UML Language Specification Format

- **Abstract Syntax**
  - Concepts, their features, and mutual relationships
  - Described by a MOF metamodel + additional constraints (OCL or English)
- **Concrete Syntax**
  - Notation (diagrams, text, tables, graphical representation)
  - UML concrete syntax definition is incomplete and informally defined
  - XML-based Interchange format (XMI)
- **Language Semantics**
  - The meaning of UML models and the concepts used to express them
  - English (+ fUML model + mathematical model)

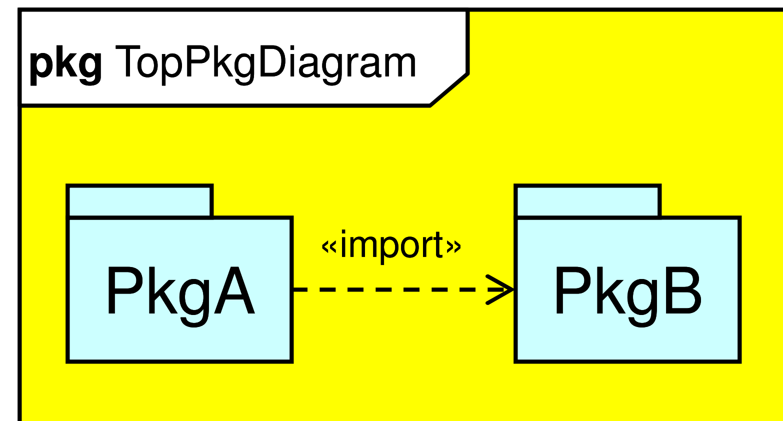
© Copyright Malina Software

© Copyright Malina Software

General Diagram Format

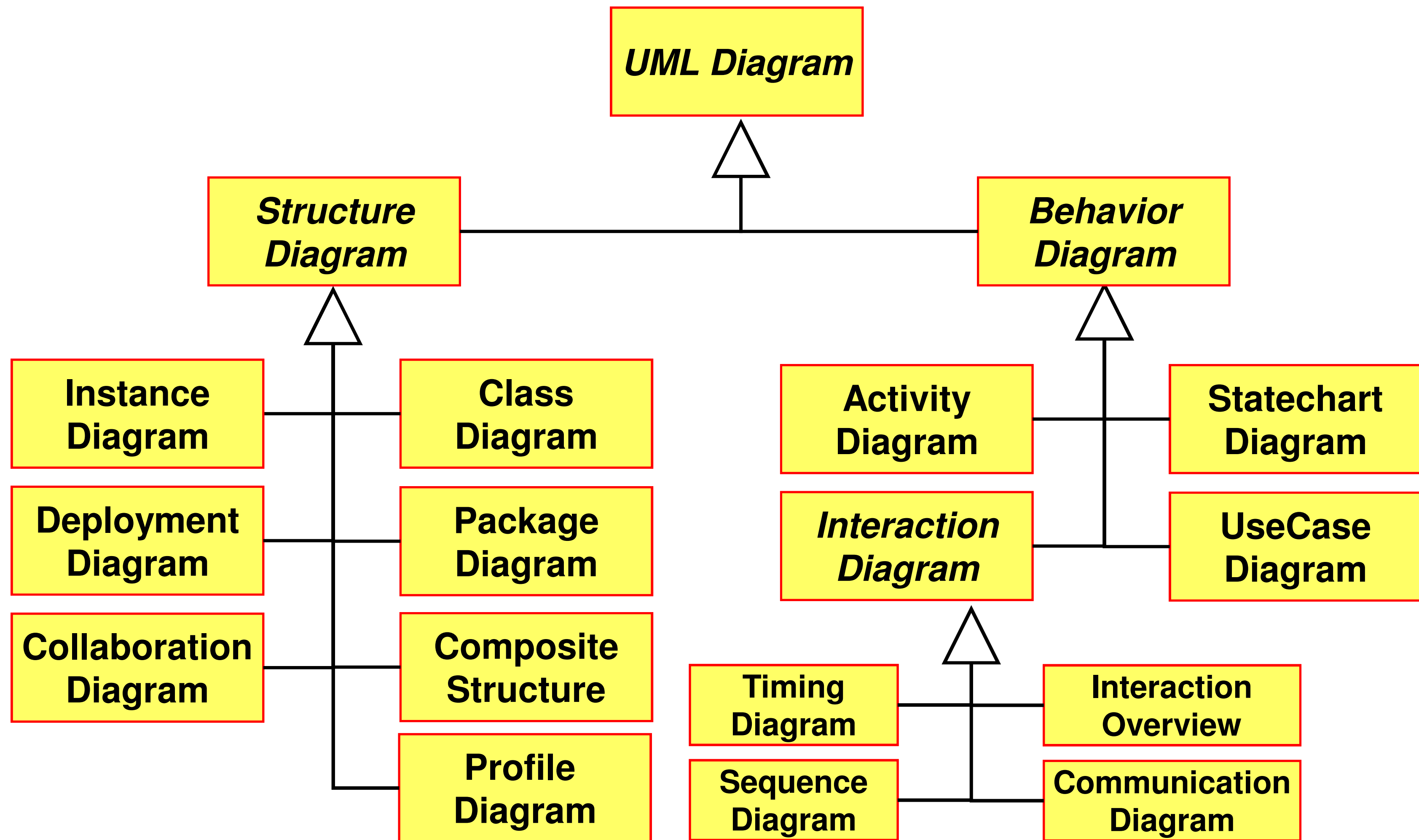
- An optional structured form for UML diagrams

act
class
cmp
dep
sd
pkg
stm
uc



© Copyright Malina Software

UML Concrete Syntax: Diagram Types



© Copyright Malina Software

UML Concrete Syntax

- **Incomplete and informally defined**
  - No definitive rules on what is a valid syntactical construction
  - Creates difficulties in model interchange and confuses users
- **The relationship between concrete and abstract syntax is not fully defined**
  - The "Diagram Interchange" specification defines a one-way link from the concrete syntax elements of a model (e.g., diagrams) to the abstract syntax elements
  - Allows for multiple different representations of a given model (e.g., textual, graphical)
- **A standard "Diagram Definition" specification has recently been adopted by the OMG**
  - Can serve as a basis for a precise and complete specification of the concrete syntax of UML and other MDA languages

© Copyright Malina Software

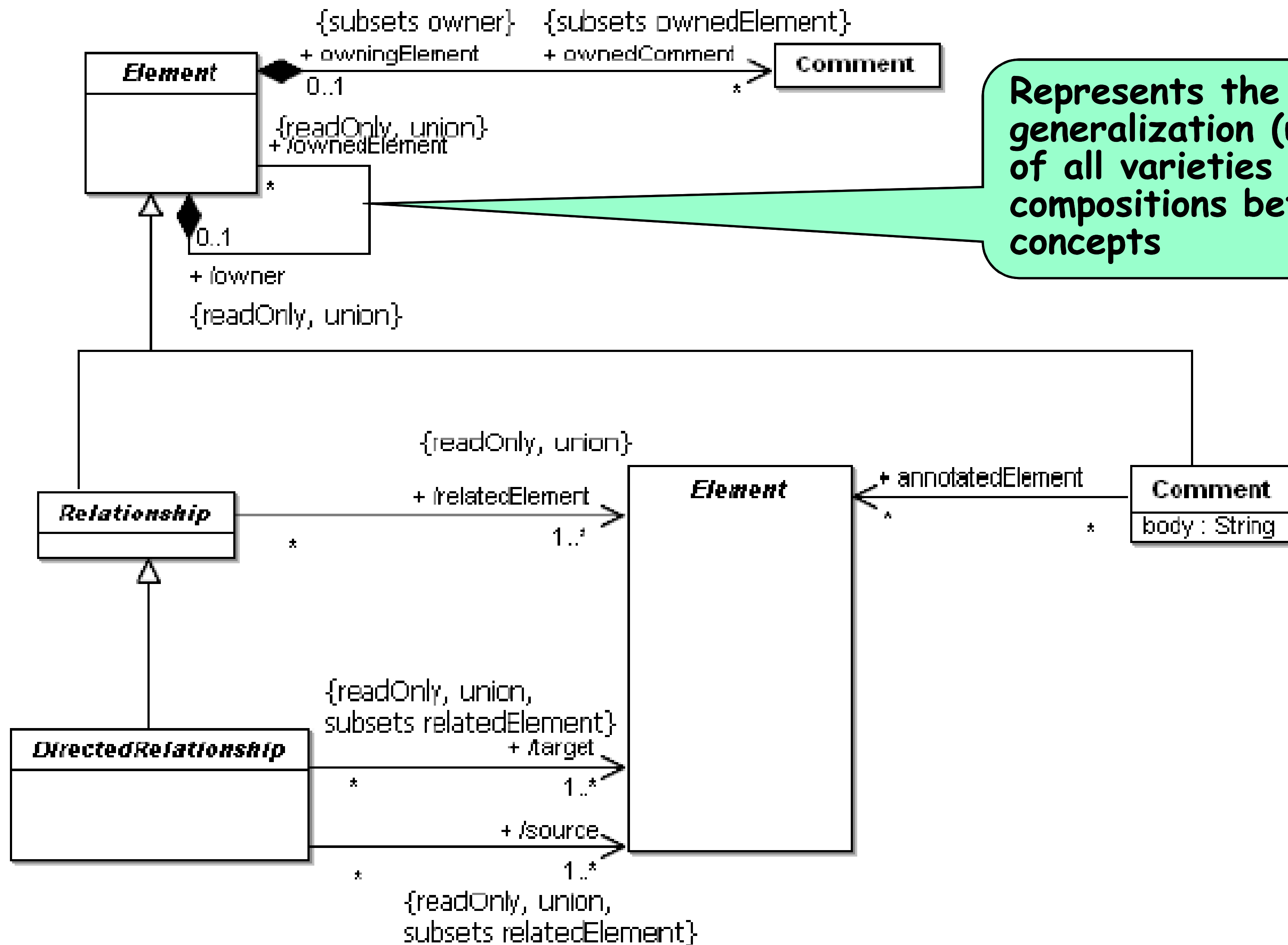
© Copyright Malina Software

New Approach to the UML Metamodel

- **In UML 2, a more incremental approach was used to define the abstract syntax (compared to UML 1)**
  - Based on a core of fine-grained abstract concepts (e.g., namespace, multiplicity, redefinition)
  - More complex concepts derived using mixin-based composition of core concepts and package merge
  - More refined semantics through the use of association specialization
- **Enables:**
  - Cleaner definition of core semantics (concepts are isolated from each other)
  - More flexible definition of complex concepts
  - Simpler evolution and extension

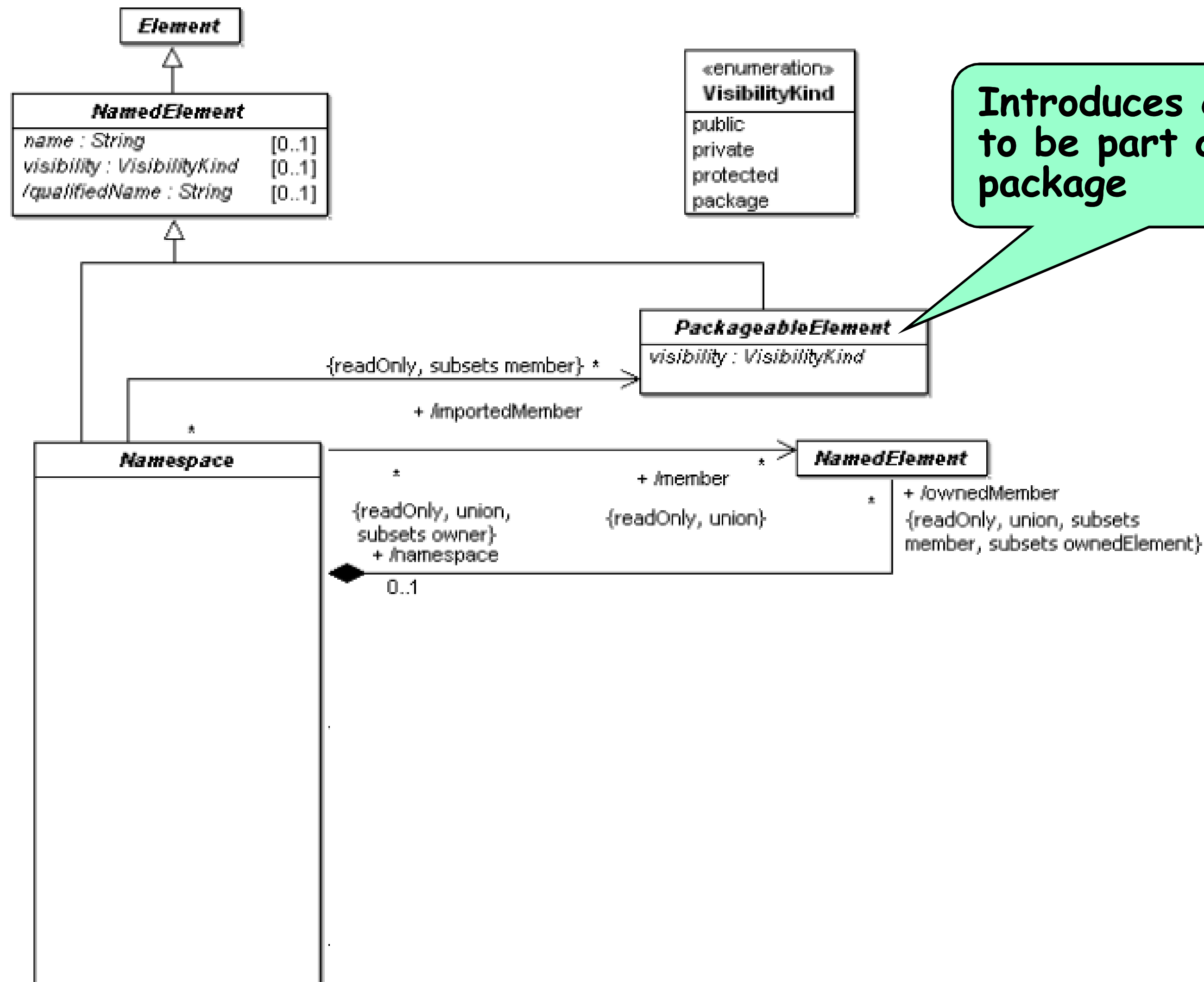
© Copyright Malina Software

The Root of the UML Metamodel



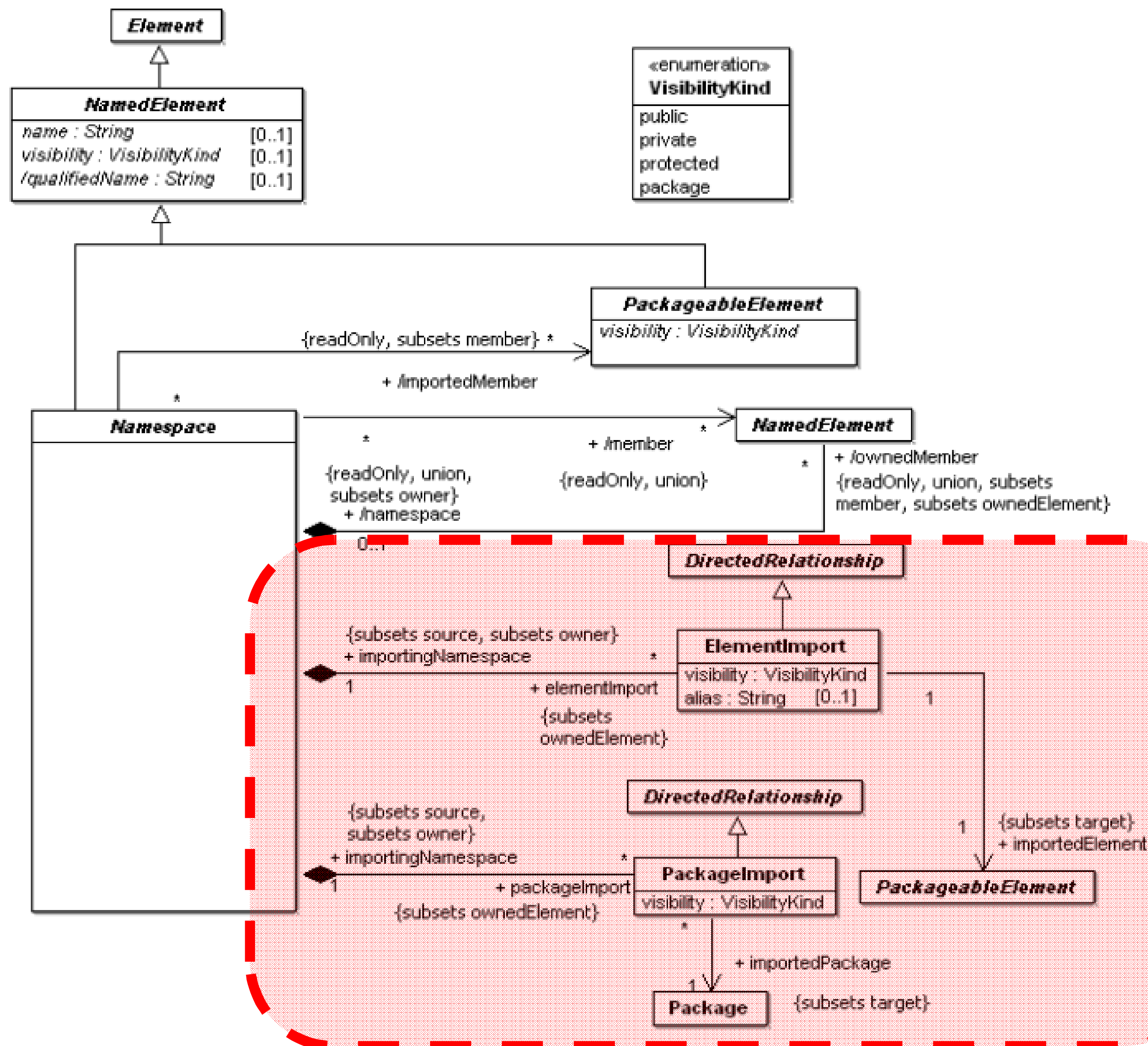
© Copyright Malina Software

The Namespaces Metamodel



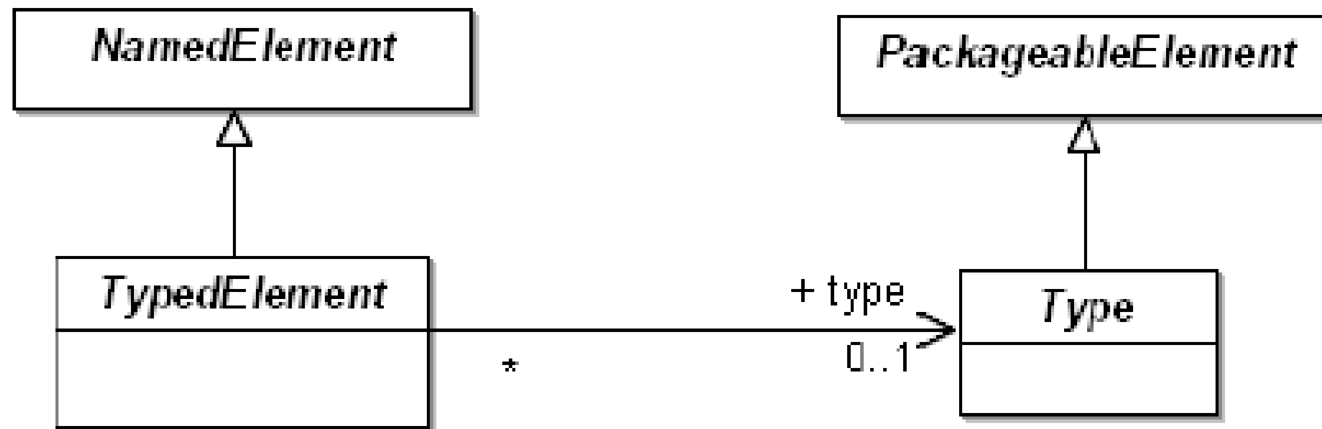
© Copyright Marina Software

Adding Namespace Relationships



© Copyright Malina Software

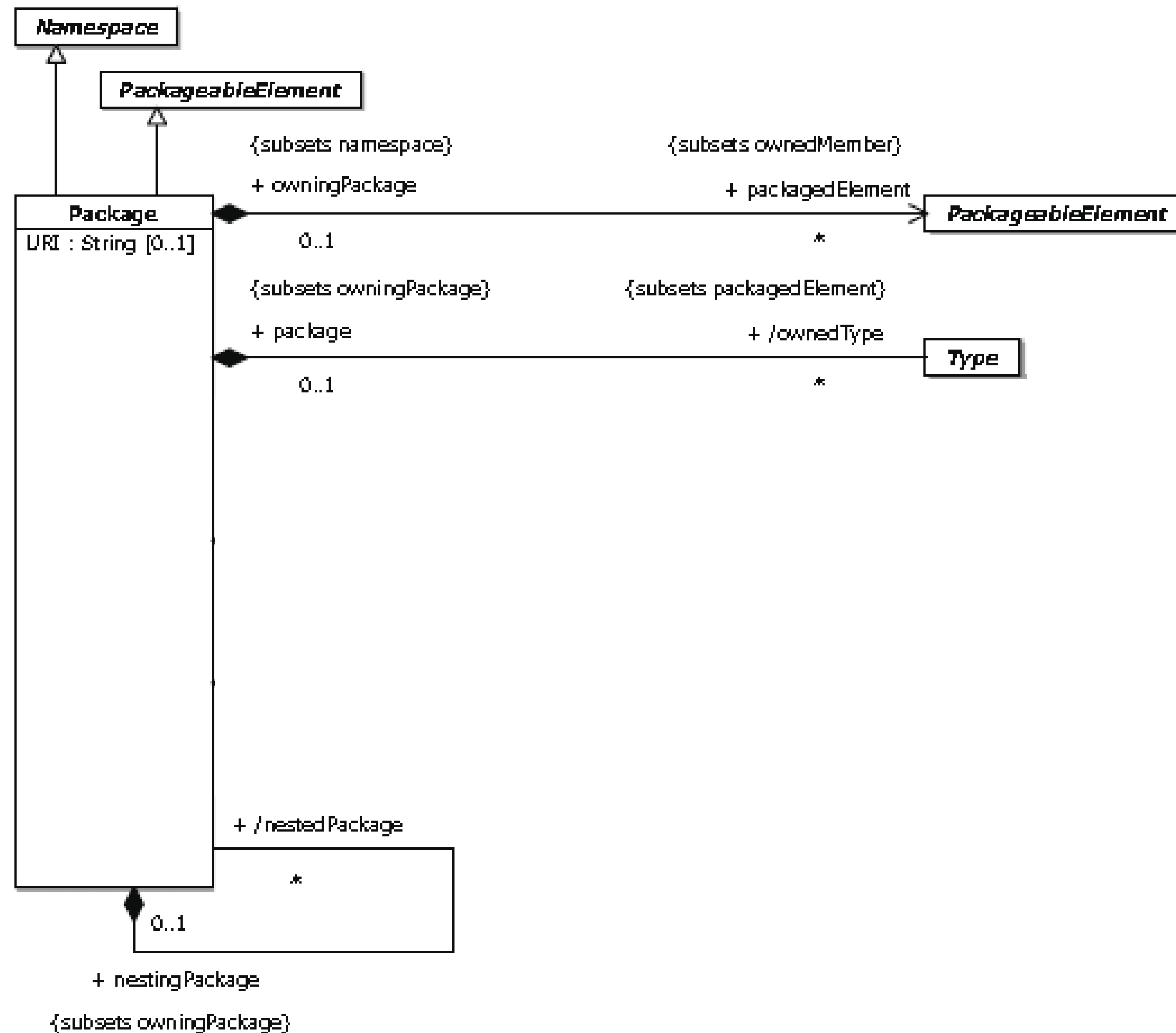
Types Metamodel



- **NB: typed elements may not actually have a type declared (lower bound of TypedElement::type = 0)**
  - To support incomplete models

© Copyright Malina Software

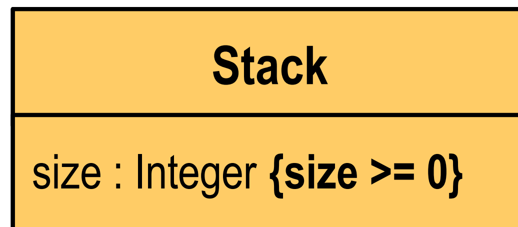
Packages Metamodel

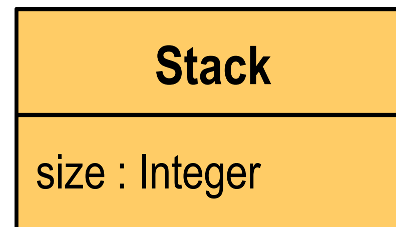


© Copyright Malina Software

Constraints

- Captures some semantics for elements of a namespace
- Expressed in either formal or natural language
  - E.g., Object Constraint Language (OCL), English, Australian
- Notation choices:





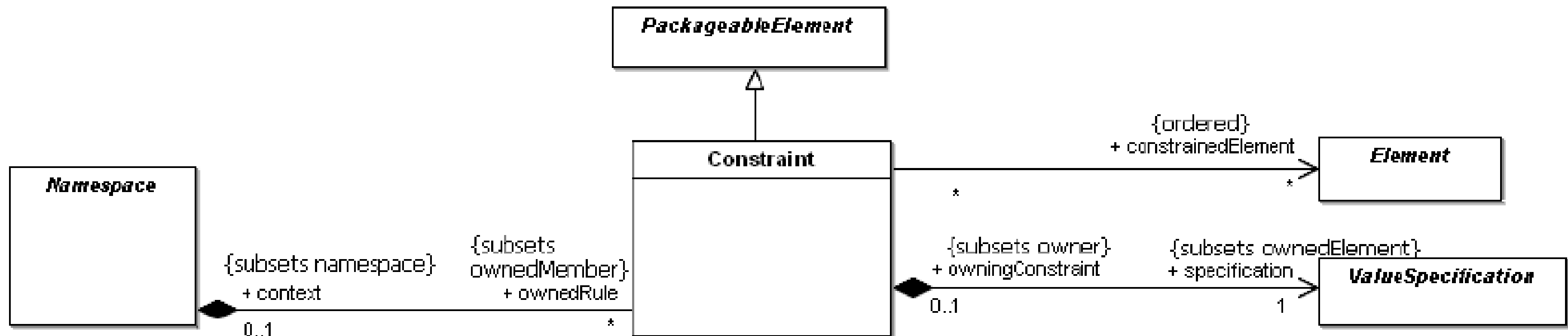
{size >= 0}

IN LINE

ATTACHED NOTE

© Copyright Malina Software

Constraints Metamodel

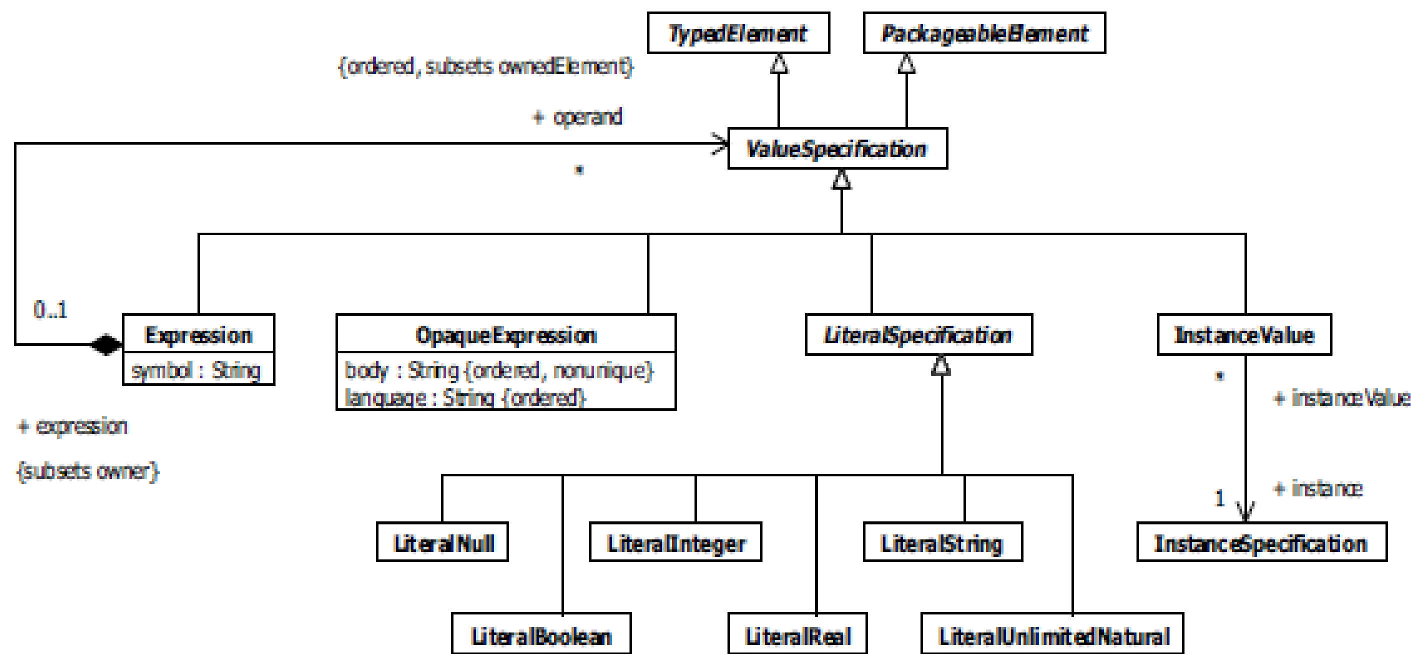


- ValueSpecification: modeling element that contains an expression which evaluates to a Boolean value (true/false)
- The namespace can be a Class, Package, Interface, etc.

© Copyright Malina Software

Core Structural Concepts: Values

- **Value: universal, unique, and unchanging**
  - Numbers, characters, strings
  - Represented by a symbol (e.g., 7, VII, "a", "fubar")
  - Includes object identifiers ("InstanceValue")
  - Represented by ValueSpecification in the UML metamodel



© Copyright Malina Software

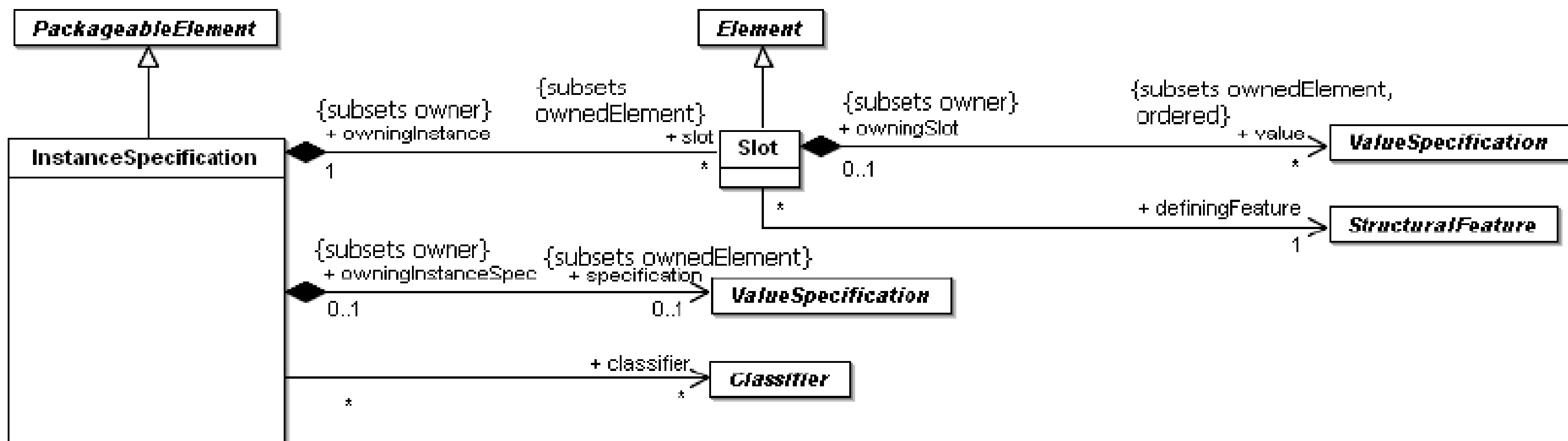
Core Structural Concepts: Objects

- "Value containers": a (physical) space for storing values
  - Can be created and destroyed dynamically
  - Limited to storing only values conforming to a given type
  - The stored value may be changed (e.g., variables, attributes)
  - Multiple forms in UML: variables, attribute slots, objects
- Object: a complex value container with a unique identity conforming to a Class type
  - May own attribute slots (containers for attribute values)
  - May be capable of performing behaviours (operations, etc.)
  - Identity does not change during its lifecycle
  - ...but its type could change over type (type reclassification)

© Copyright Malina Software

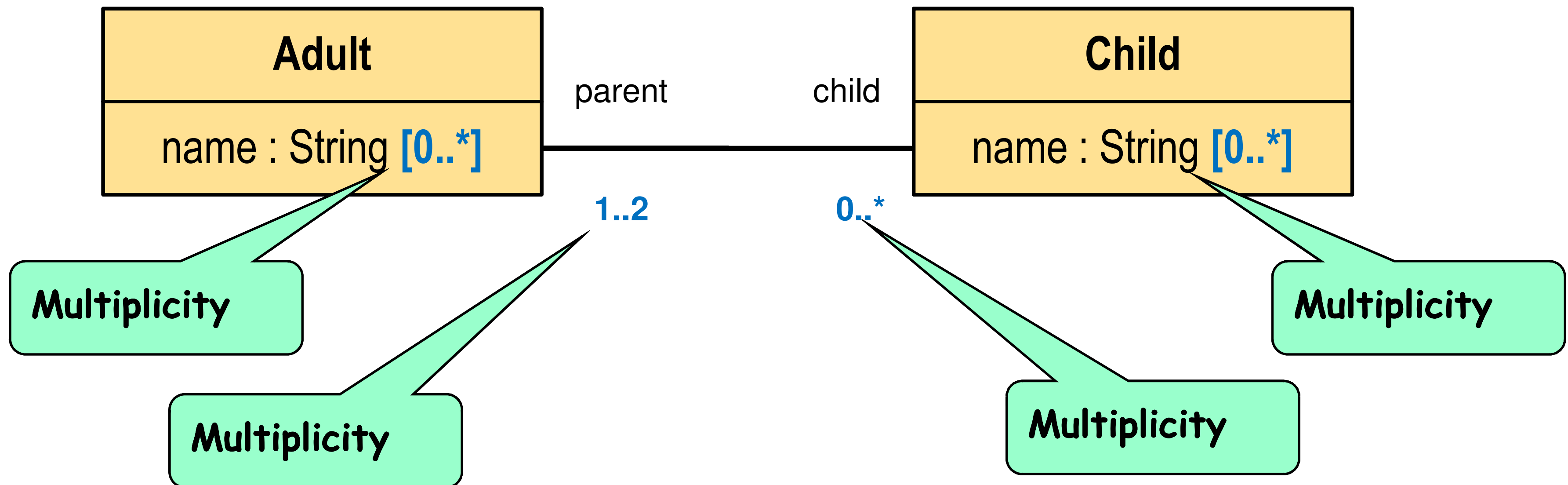
Specifying Instances in UML

- InstanceSpecification: a generic facility for modeling objects and other things that occupy memory space
- The metamodel



© Copyright Malina Software

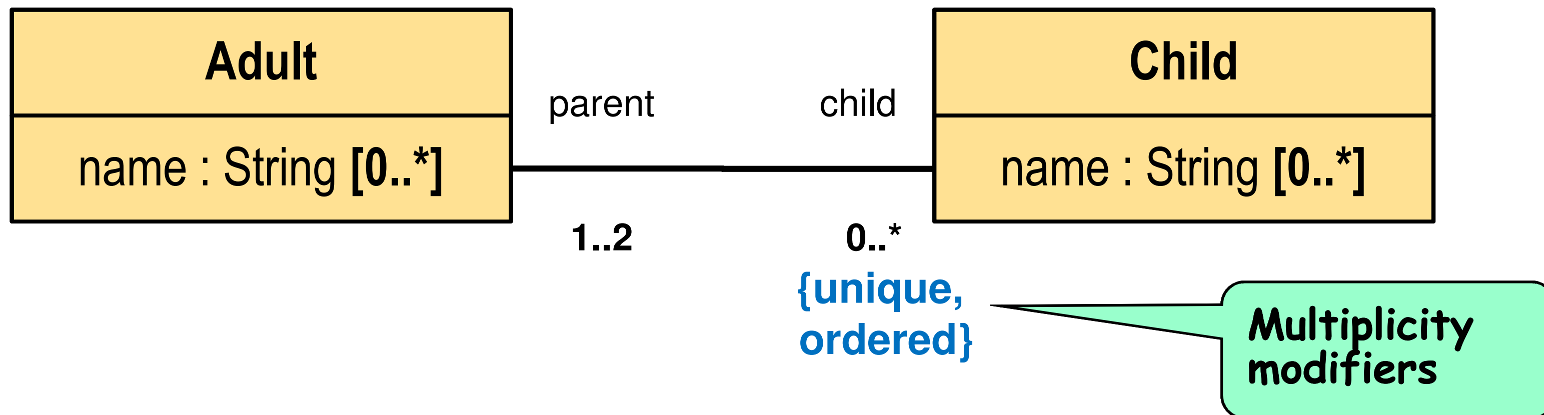
Multiplicity Concept



- For specifying usages that are collections
  - Attributes, association ends, parameters, etc.
- Can specify ranges (upper and lower bounds)
  - 0 = absence of elements
  - * (or [0..*]) = open-ended upper bound

© Copyright Malina Software

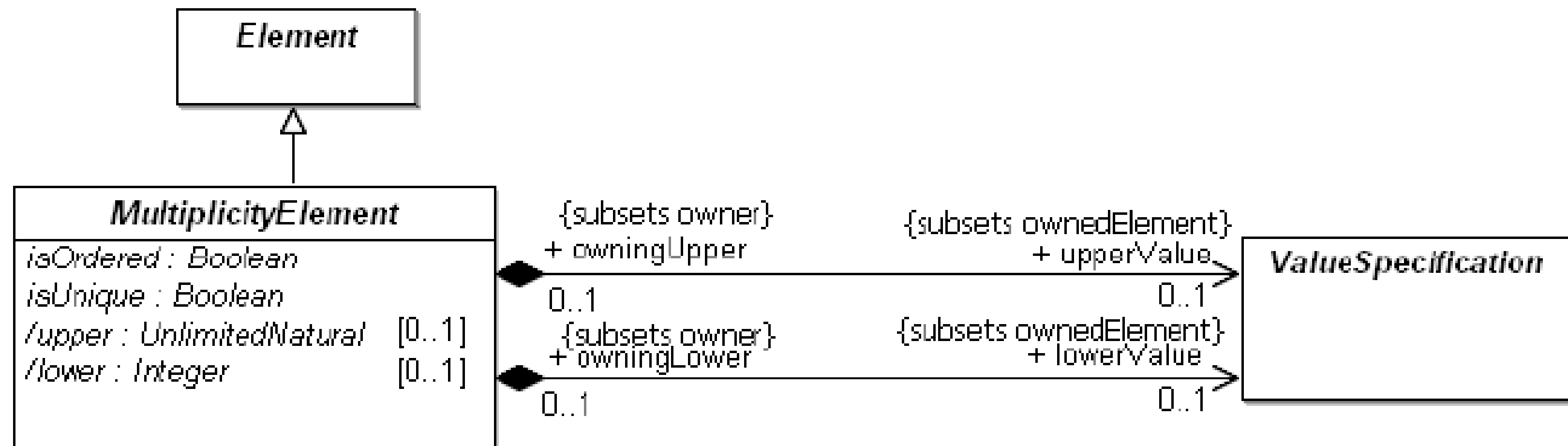
Multiplicity Modifiers



- {unique} = no two elements of the collection can have the same value
  - Default value: *true*
  - {nonunique} if *false*
- {ordered} = the elements in the collection are ordered in some appropriate manner (⇒ an ordered collection)
  - Default value: *false* = {unordered}

© Copyright Malina Software

Multiplicities Metamodel



© Copyright Malina Software

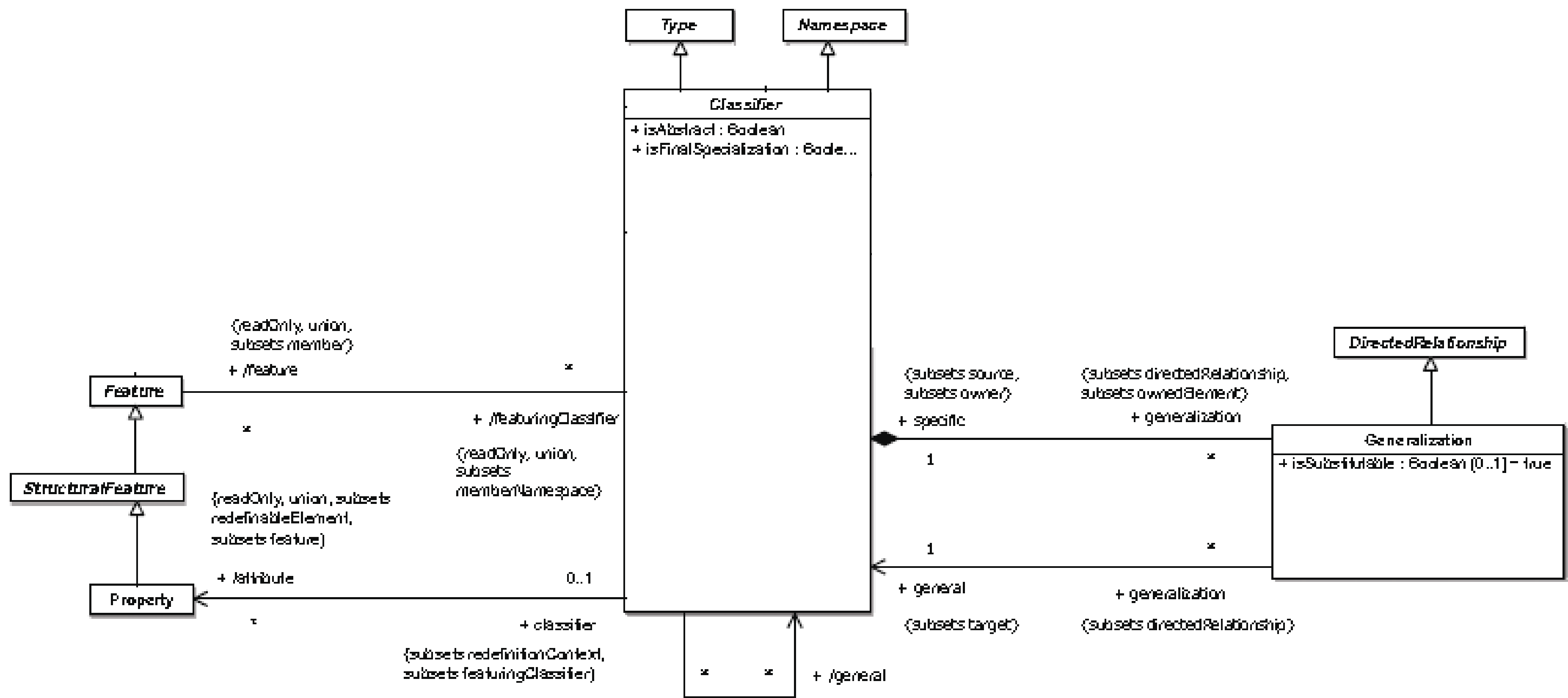
The UML Concept of Classifier

- A classifier is a specification of a collection of entities (things, concepts, etc.) that
  1. Share a set of characteristics (features) and which
  2. Can be classified (divided) into sub-collections according to the nature of some of their characteristics
- E.g., people can be classified based on their gender into men and women
  - . . . or, they may be classified according to their age into adults and children
- Kinds of Classifiers in UML:
  - Classes, Associations, AssociationClasses, DataTypes, Enumerations, Interfaces, Behaviors (Activities, Interactions, StateMachines), Signals, UseCases, Collaborations, Components, Nodes, etc.

© Copyright Malina Software

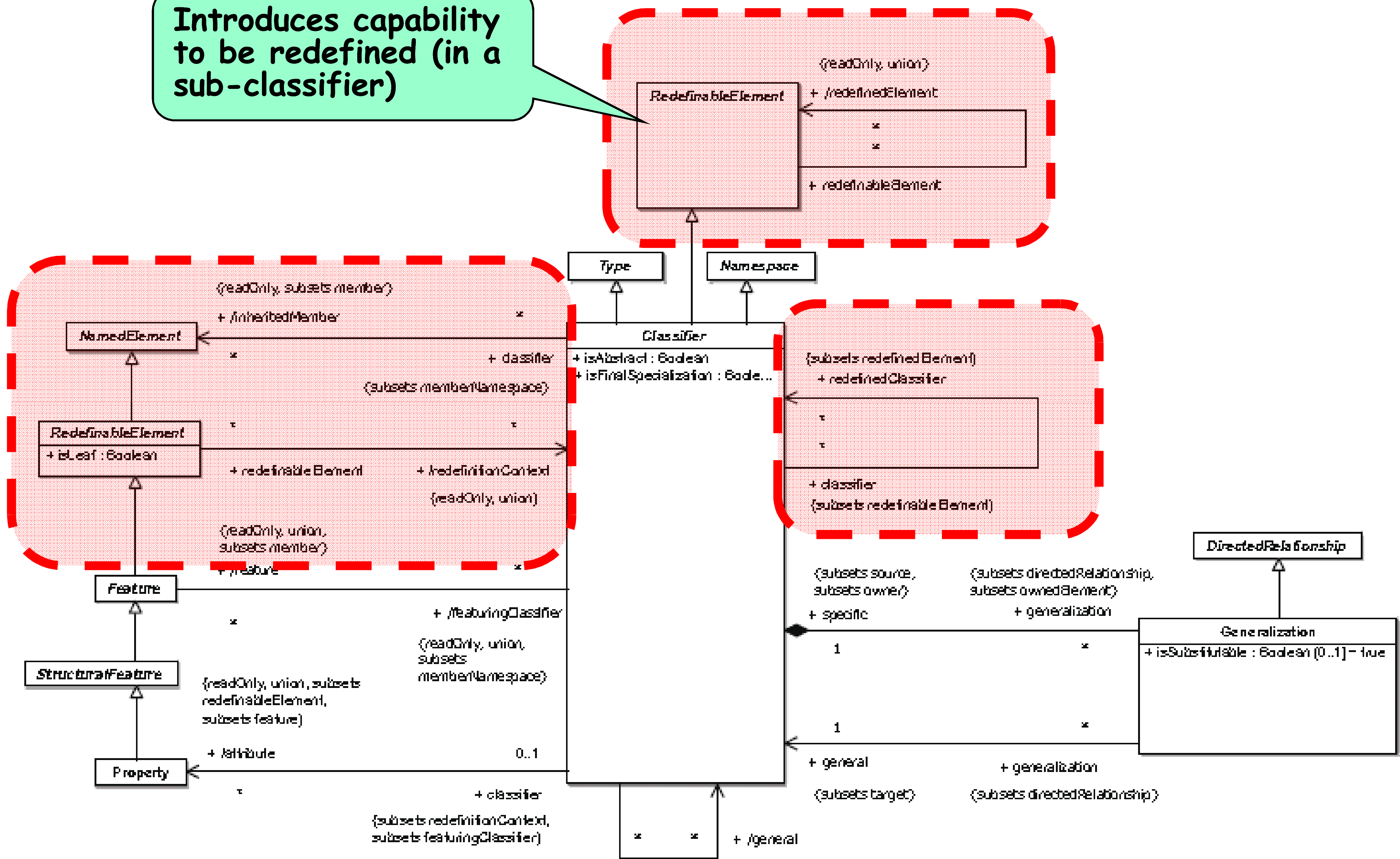
UML Classifier Concept

- **Introduces concepts that can be:**
  - Classified based on the nature of their features
  - Generalized/specialized



© Copyright Malina Software

Redefinition in UML



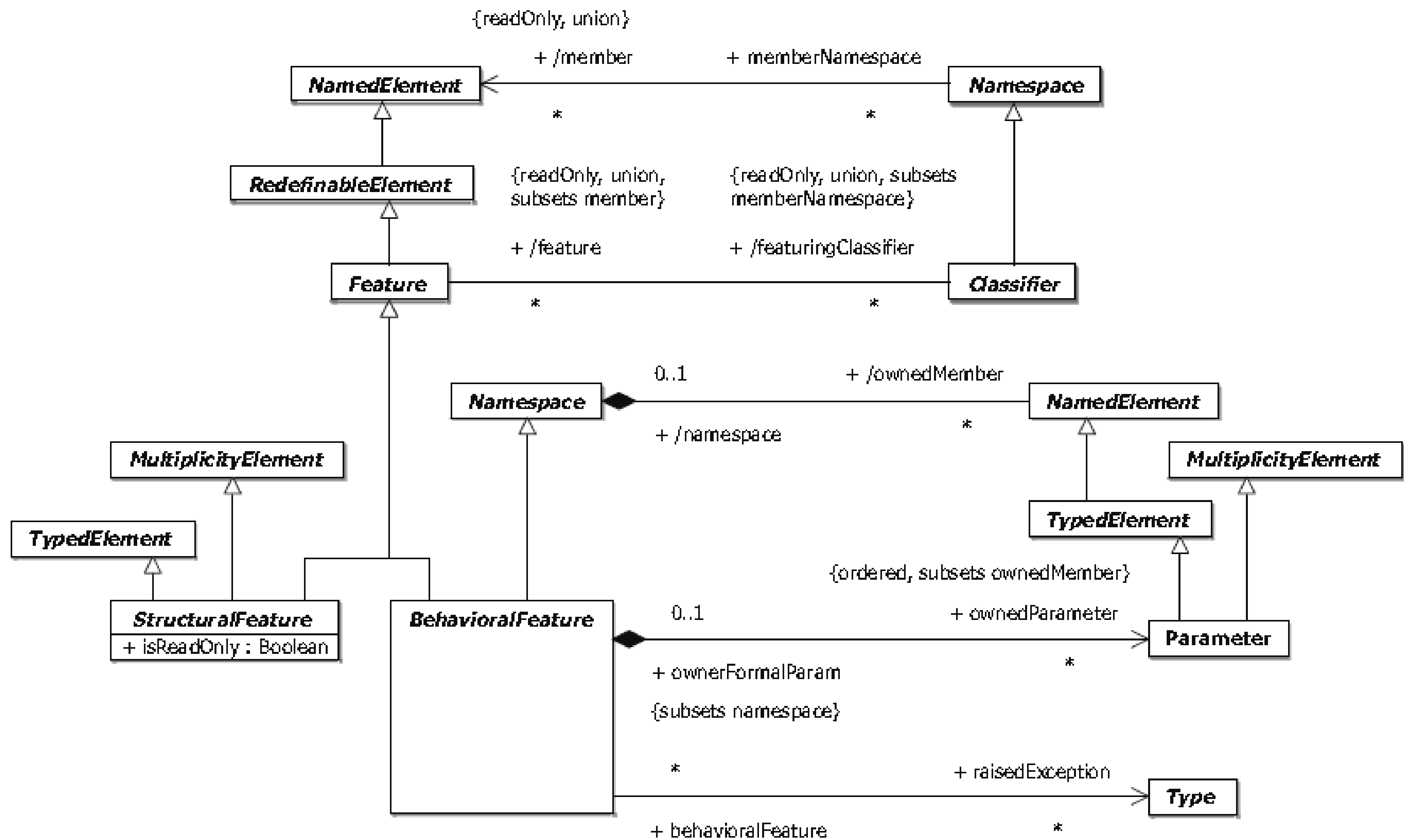
© Copyright Malina Software

Classifier Features

- Two basic kinds:
  - Structural features (e.g., attributes, associations)
  - Behavioural features (e.g., operations, receptions)
- Structural features are type usages and have:
  - Multiplicity
  - Visibility
  - Scope [static(classifier), instance]
    - Static features are singletons and apply to the classifier as a whole rather than individual instances
- Behavioural features have
  - Visibility and scope

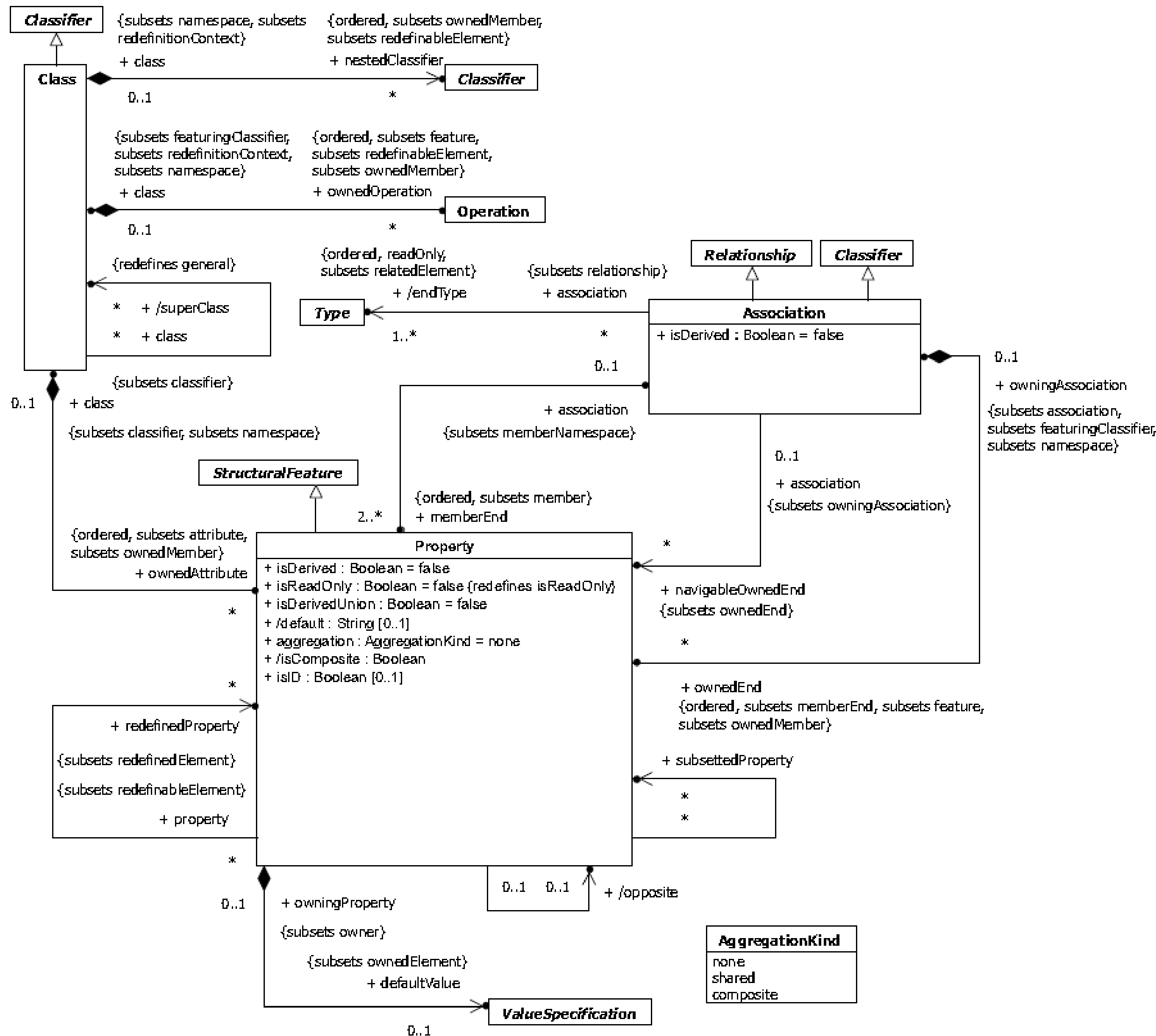
© Copyright Malina Software

Features Metamodel



© Copyright Malina Software

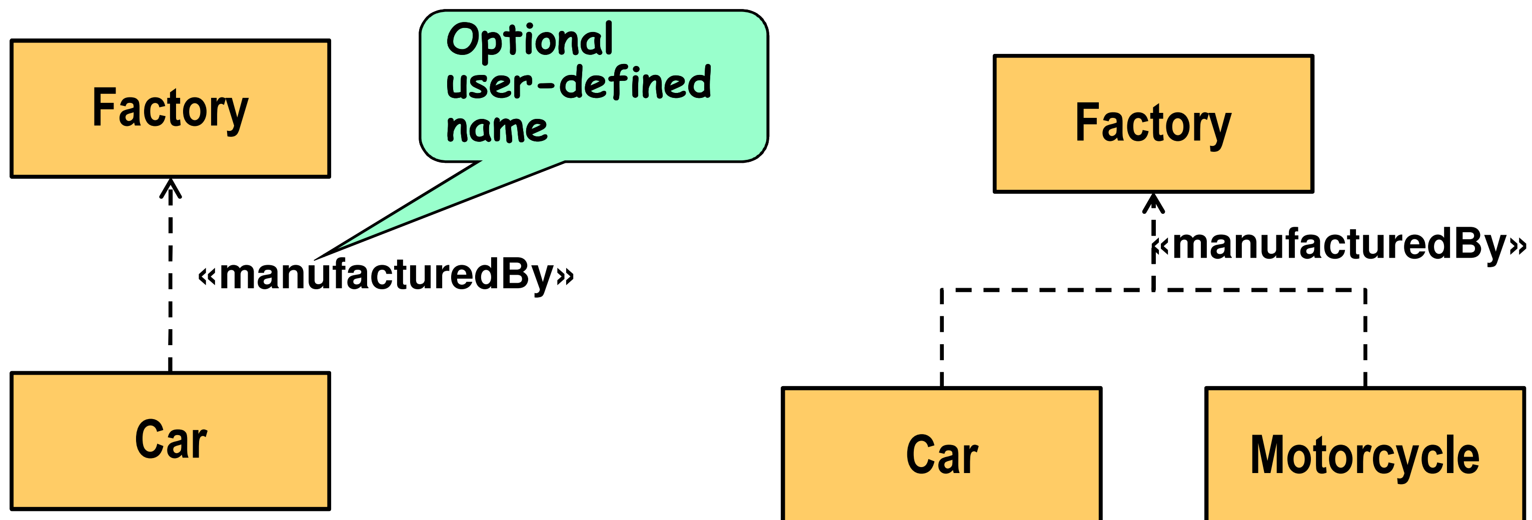
Class and Association Metamodel



© Copyright Malina Software

Dependencies

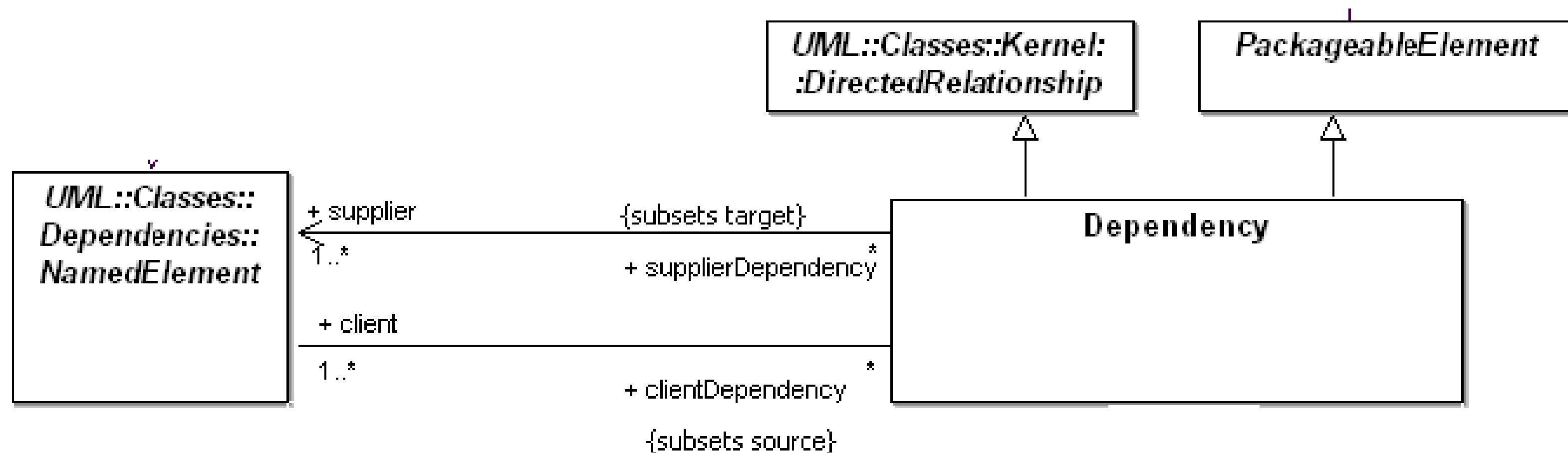
- An <u>informal</u> statement that one or more "client" items require one or more "supplier" items <u>in some way</u>
  - A change in a supplier affects the clients (in some way)
  - Semantics are very open (loose), but can be specialized



© Copyright Malina Software

Dependencies Metamodel

- In general, dependencies can be drawn between any two named elements in the model

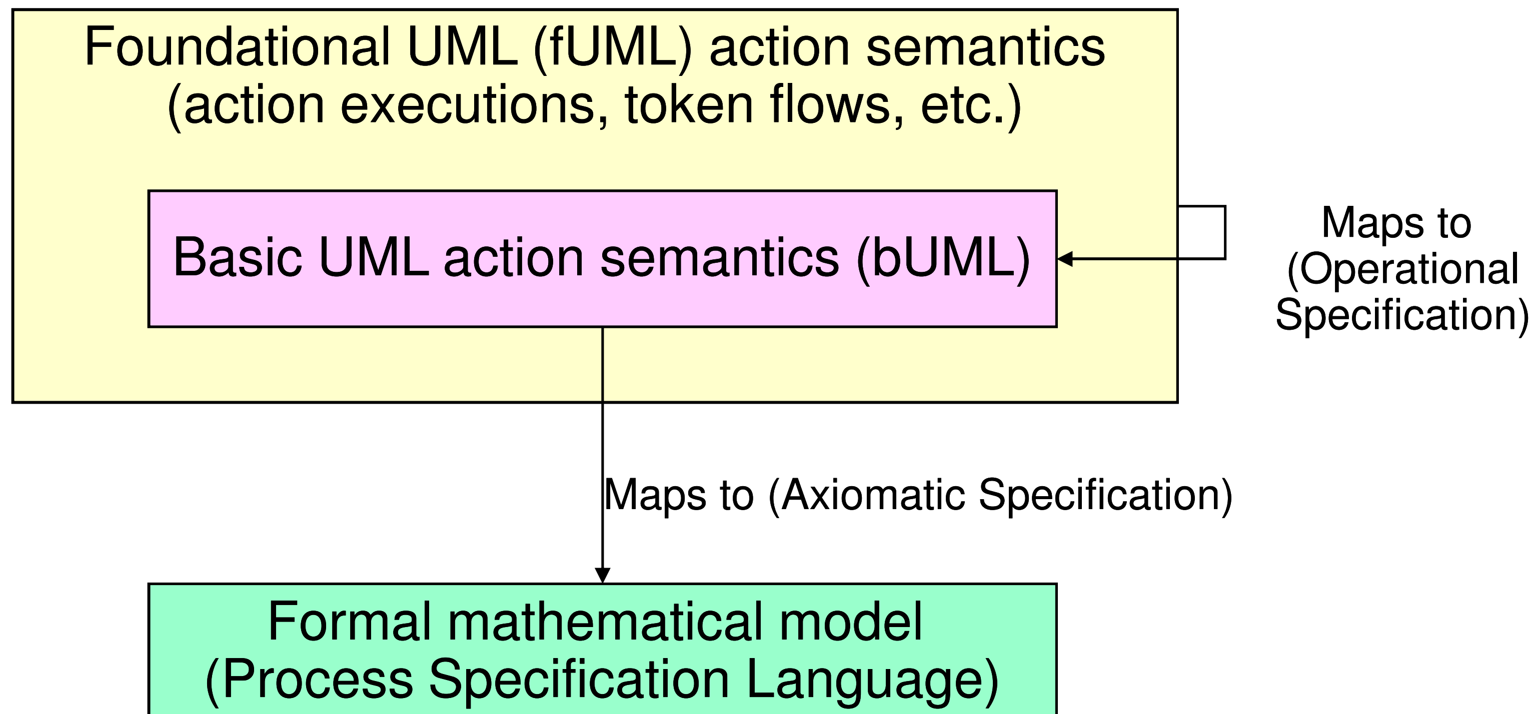


© Copyright Malina Software

© Copyright Malina Software

Foundational UML (fUML) and Basic UML (bUML)

- A subset of fUML actions is used as a core language (Basic UML) that is used to describe fUML itself



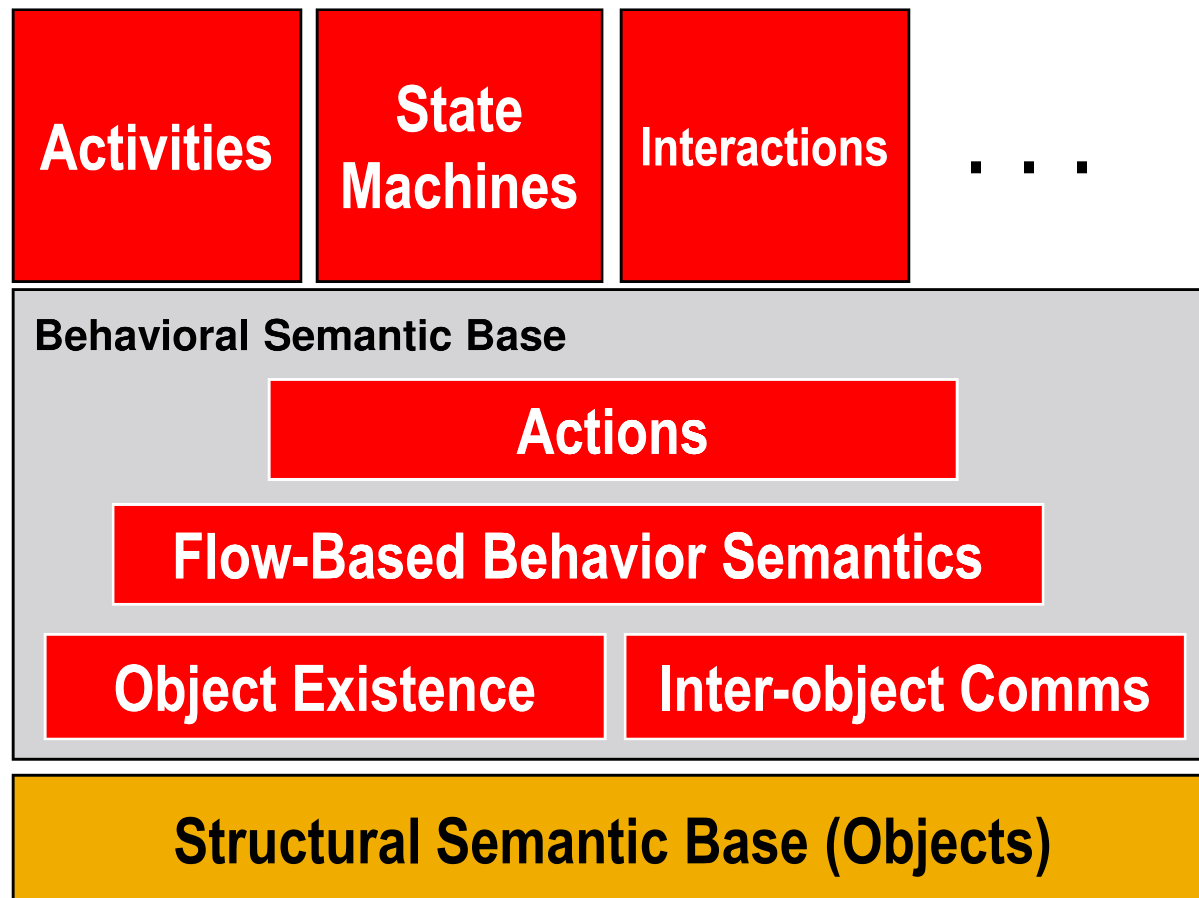
© Copyright Malina Software

UML Model of Computation

- Structure dominant
  - All behavior stems from (active) objects
- Distributed
  - Multiple sites of execution ("localities")
- Concurrent
  - Active objects $\Rightarrow$ multiple threads of execution
- Heterogeneous causality model
  - Event driven at the highest level
  - Data and control flow driven at more detailed levels
- Heterogeneous interaction model
  - Synchronous, asynchronous, mixed

© Copyright Malina Software

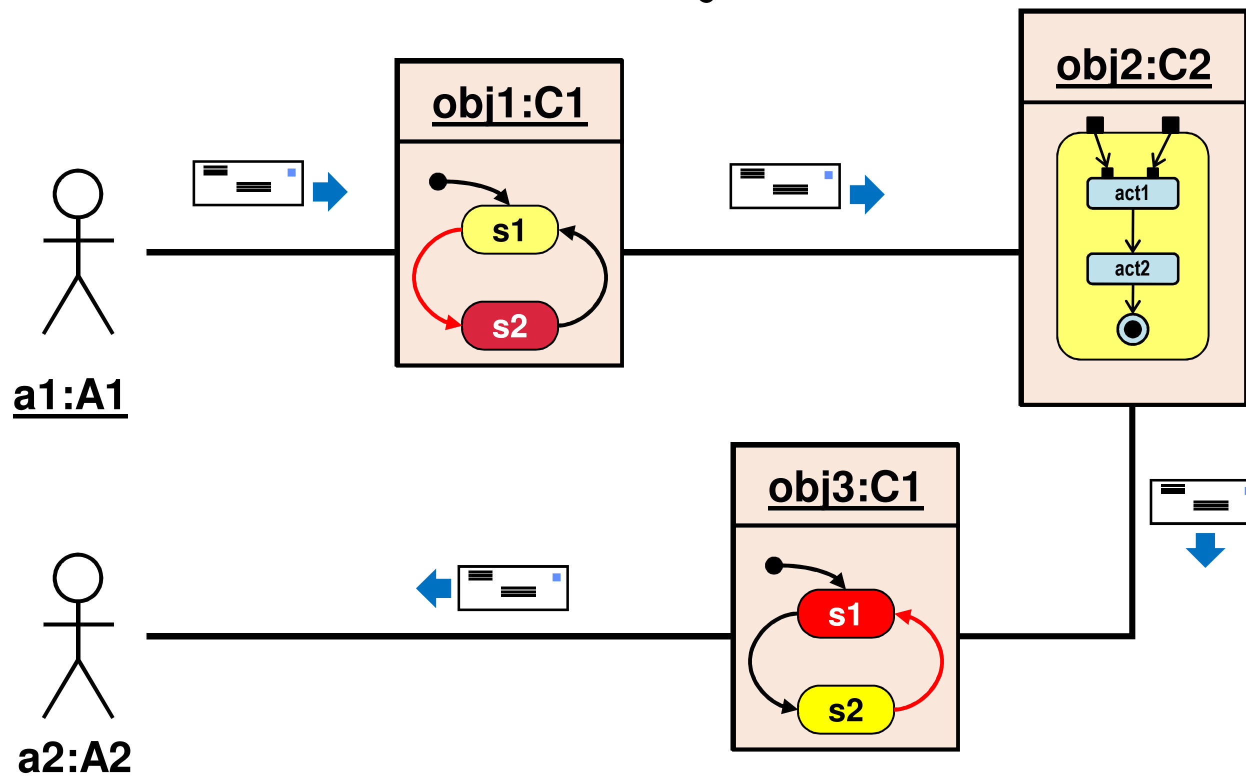
UML Run-Time (Dynamic) Semantics Architecture



© Copyright Malina Software

UML Model of Causality (How Things Happen)

- A discrete event-driven model of computation
  - Network of communicating objects
- All behaviour stems from objects



© Copyright Malina Software

How Things Happen in UML

- **An action is executed by an object**
  - May change the contents of one or more variables or slots
  - If it is a communication ("messaging") action, it may:
    - Invoke an operation on another object
    - Send a signal to another object
    - Either one will eventually cause the execution of a procedure on the target object…
    - …which will cause other actions to be executed, etc.
  - Successor actions are executed
    - Determined either by control flow or data flow

© Copyright Malina Software

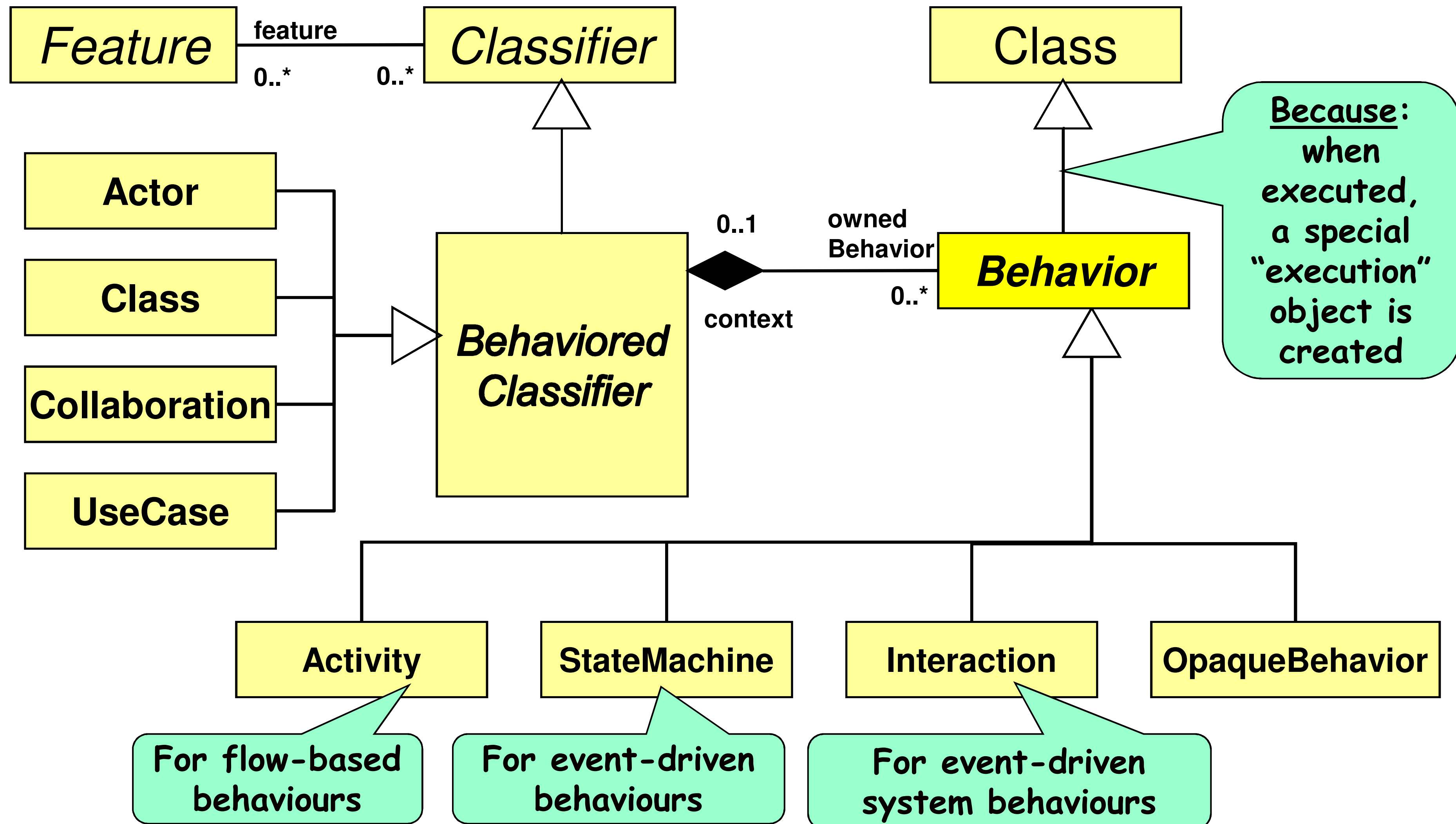
Basic Structural Elements

- **Values**
  - Universal, unique, constant
  - E.g. Numbers, characters, object identifiers ("instance value")
- **"Cells" (Slots/Variables)**
  - Container for values or objects
  - Can be created and destroyed dynamically
  - Constrained by a type
  - Have identity (independent of contents)
- **Objects (Instances)**
  - Containers of slots (corresponding to structural features)
  - Just a special kind of cell
- **Links**
  - Tuples of object identifiers
  - May have identity (i.e., some links are objects)
  - Can be created and destroyed dynamically

© Copyright Malina Software

Relationship Between Structure and Behaviour

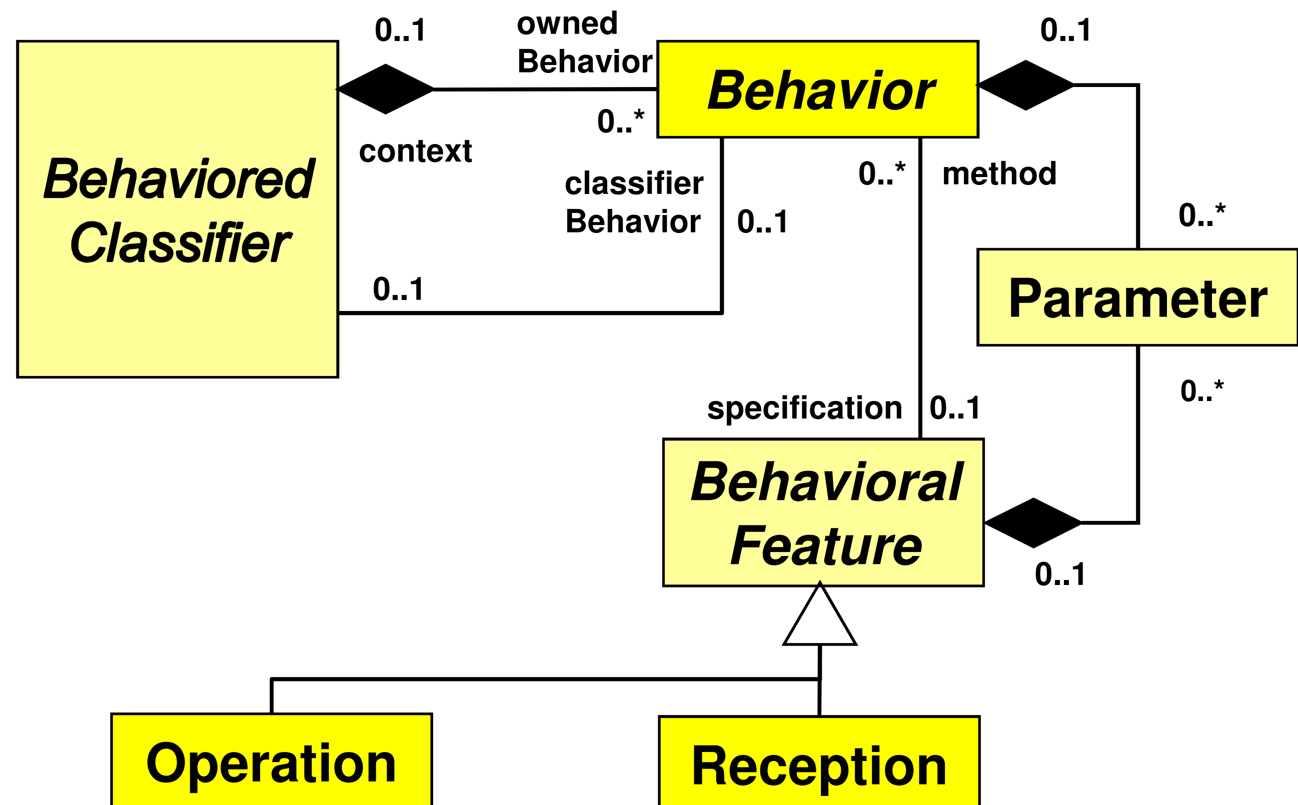
- From the UML metamodel:



© Copyright Malina Software

Classifier Behaviours vs. Methods

- **Methods**: Intended primarily for passive objects
  - Can be synchronous (for operations) or asynchronous (for receptions)
- **Classifier behaviour**: Intended primarily for active objects
  - Executed when the object is created



© Copyright Malina Software

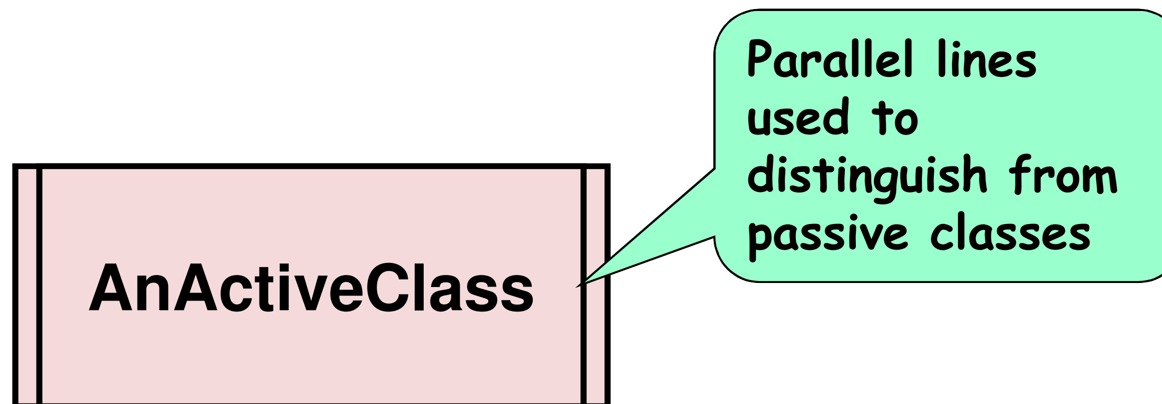
Active Object Definition

- Active object definition:

  An active object is an object that, as a direct consequence of its creation, commences to execute its classifier behavior, and does not cease until either the complete behavior is executed or the object is terminated by some external object.

- Also:

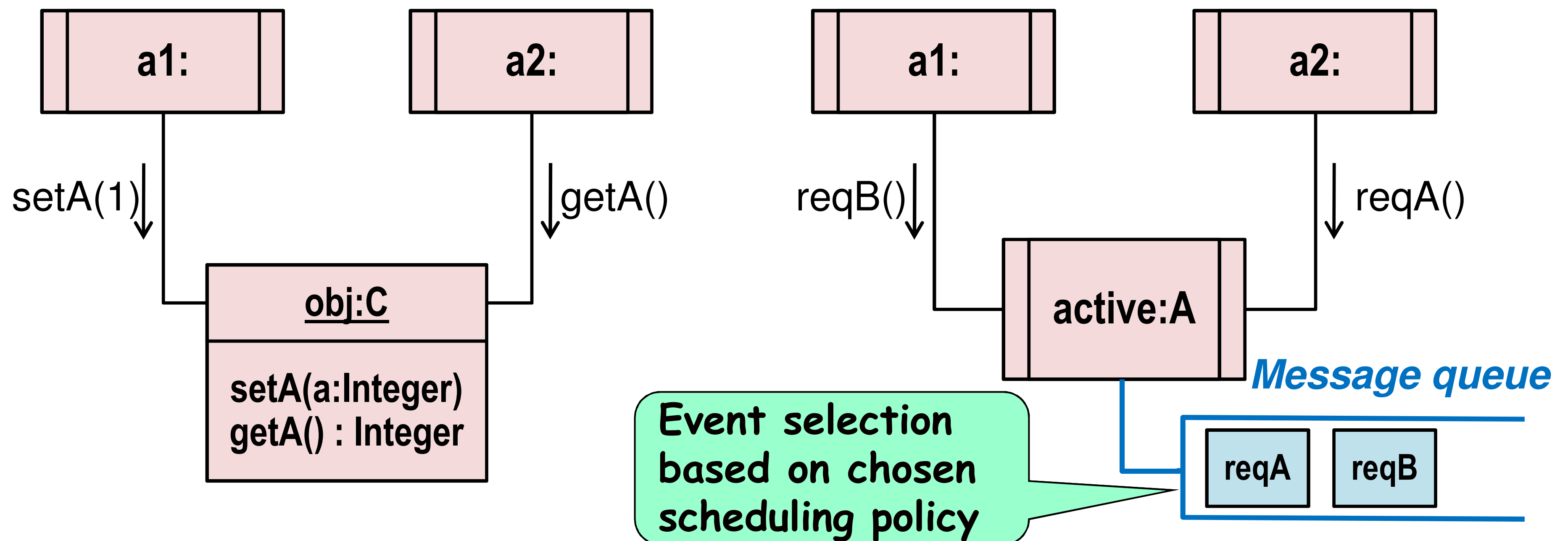
  The points at which an active object responds to [messages received] from other objects is determined solely by the behavior specification of the active object...



© Copyright Malina Software

Passive vs. Active Objects

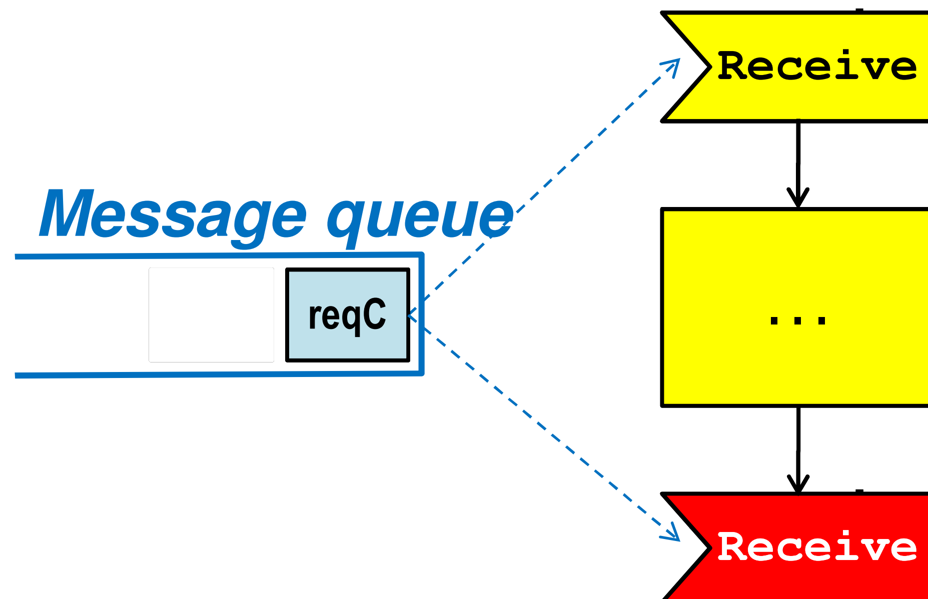
- Passive objects respond whenever an operation (or reception) of theirs is invoked
  - NB: invocations may be concurrent ⇒ conflicts possible!
- Active objects run concurrently and respond only when they execute a “receive” action



© Copyright Malina Software

Run-To-Completion (RTC) Semantics

- **Any messages arriving <u>between</u> successive "receive" actions are queued and only considered for handling on the next "receive" action**
  - Simple "one thing at a time" approach
  - Avoids concurrency conflicts



© Copyright Malina Software

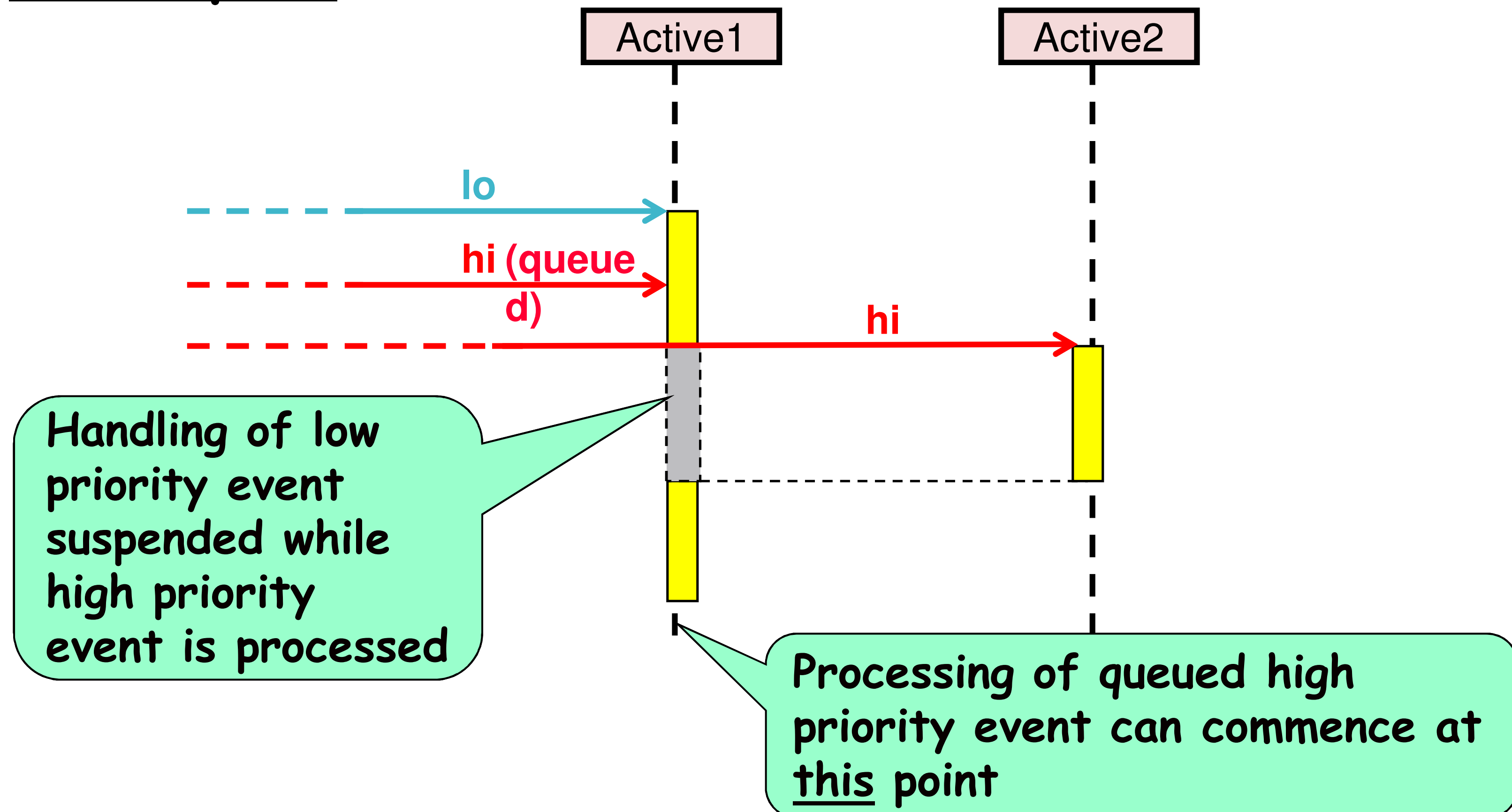
The Problem with RTC

- <u>Message (event) priority</u>: in some systems (e.g., real-time systems) messages may be assigned different priorities
  - To differentiate important (high priority) events from those that are less so and to give them priority handling (e.g., interrupting handling of a low priority message)
- <u>Priority inversion</u>: The situation that occurs when a high priority message has to wait for a low priority message
- *The RTC approach is susceptible to priority inversion*
  - But, it is limited to situations where the high-priority and low-priority events are being handled by the same object (rather than the system as a whole)

© Copyright Malina Software

RTC Semantics

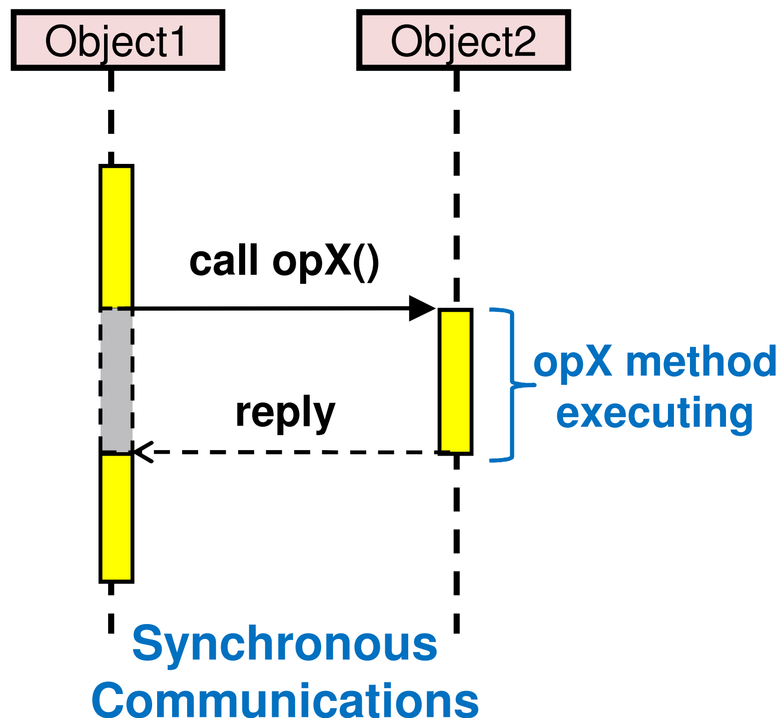
- If a high priority event arrives for an object that is ready to receive it, the processing of any low priority events by other active objects can be interrupted



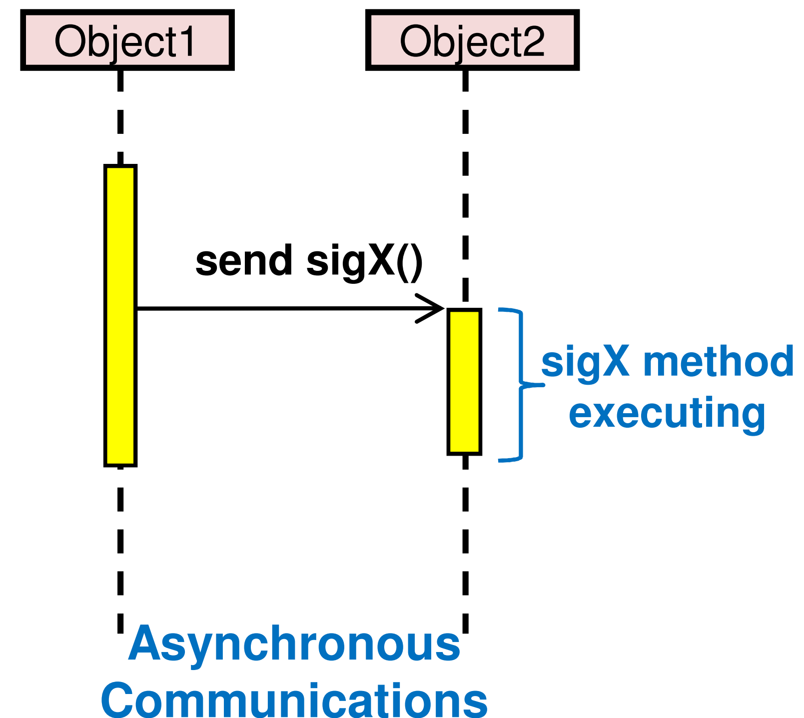
© Copyright Malina Software

UML Communications Types

- **Synchronous communications: (Call and wait)**
  - Calling an operation synchronously
- **Asynchronous communications: (Send and continue)**
  - Sending a signal to a reception
  - Asynchronous call of an operation (any replies discarded)



Synchronous Communications



Asynchronous Communications

© Copyright Malina Software

Purpose of UML Actions

- For modelling fine-grained behavioural phenomena which manipulates and accesses UML entities (objects, links, attributes, operations, etc.)
  - E.g. create link, write attribute, destroy object
  - A kind of UML "assembler"
- The UML standard defines:
  - A set of actions and their semantics (i.e., what happens when the actions are executed)
  - A method for combining actions to construct more complex behaviours
- The standard does not define:
  - A concrete syntax (notation) for individual kinds of actions
  - Proposal exists for a concrete semantics for UML Actions

© Copyright Malina Software

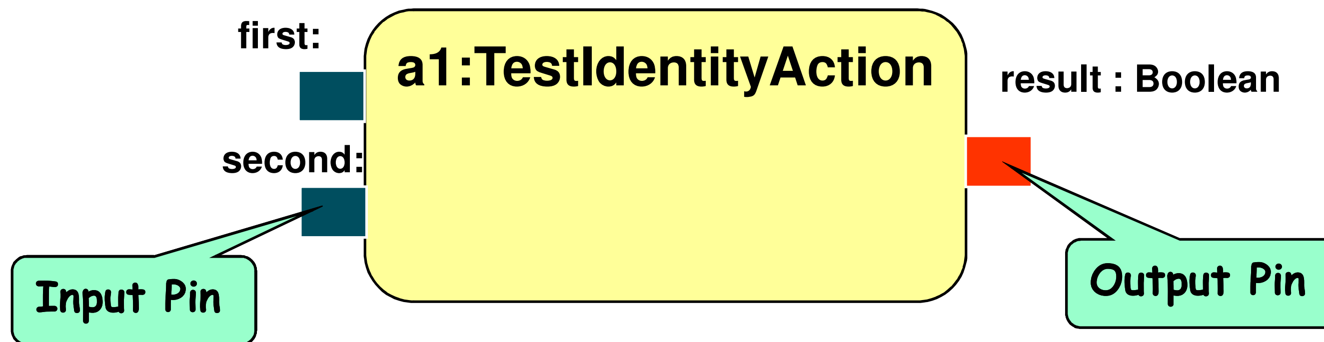
Categories of UML Actions

- **Capabilities covered**
  - Communication actions (send, call, receive,…)
  - Primitive function action
  - Object actions (create, destroy, reclassify,start,…)
  - Structural feature actions (read, write, clear,…)
  - Link actions (create, destroy, read, write,…)
  - Variable actions (read, write, clear,…)
  - Exception action (raise)
- **Capabilities not covered**
  - Standard control constructs (IF, LOOP, etc. – handled through Activities)
  - Input-output
  - Computations of any kind (arithmetic, Boolean logic, higher-level functions)

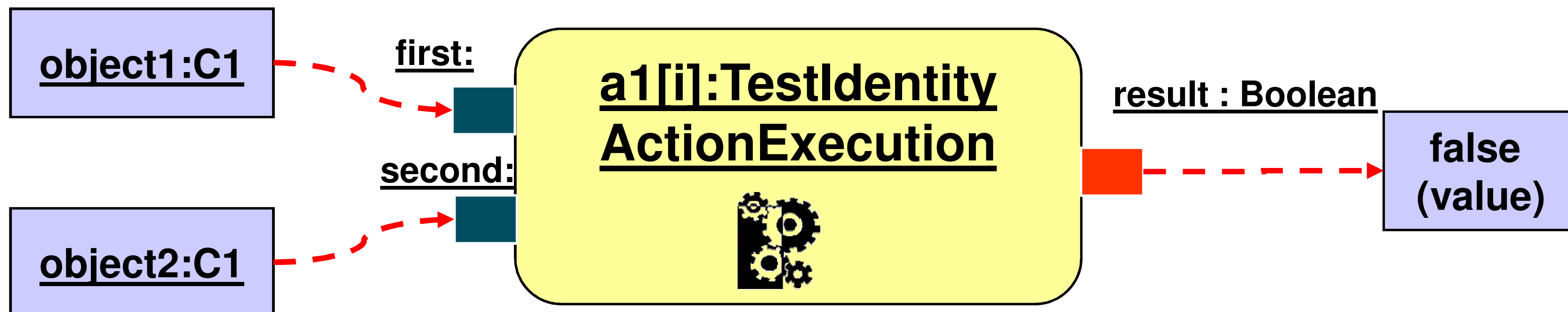
© Copyright Malina Software

Action Specifications and Action Executions

Action Specification (a design-time specification)



Action Execution (a run-time concept)

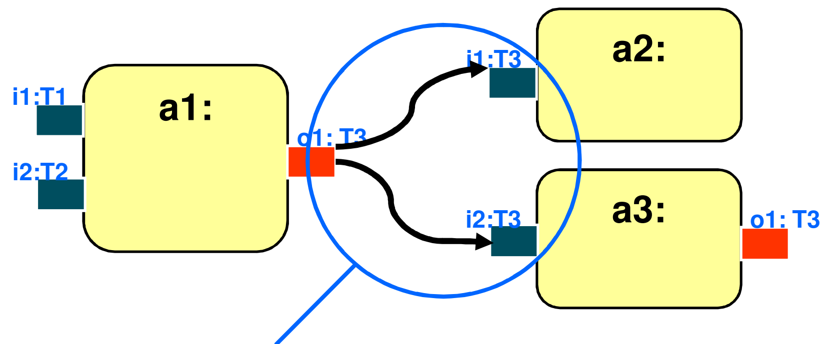


NB: each time action a1 needs to be executed, a new action execution is created

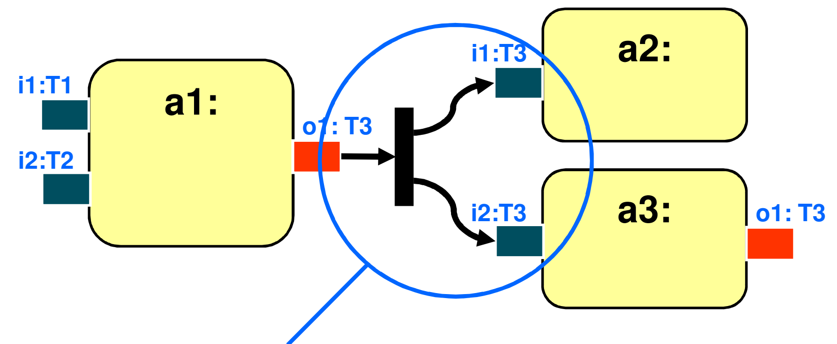
© Copyright Malina Software

Combining Actions

- Data flow MoC: output to input connections

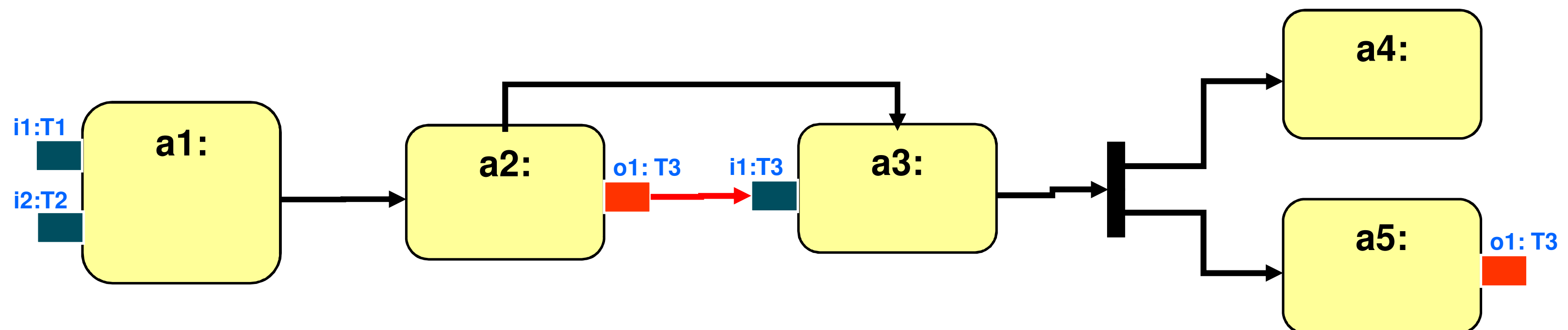


Contention (a2 and a3)



Data replication

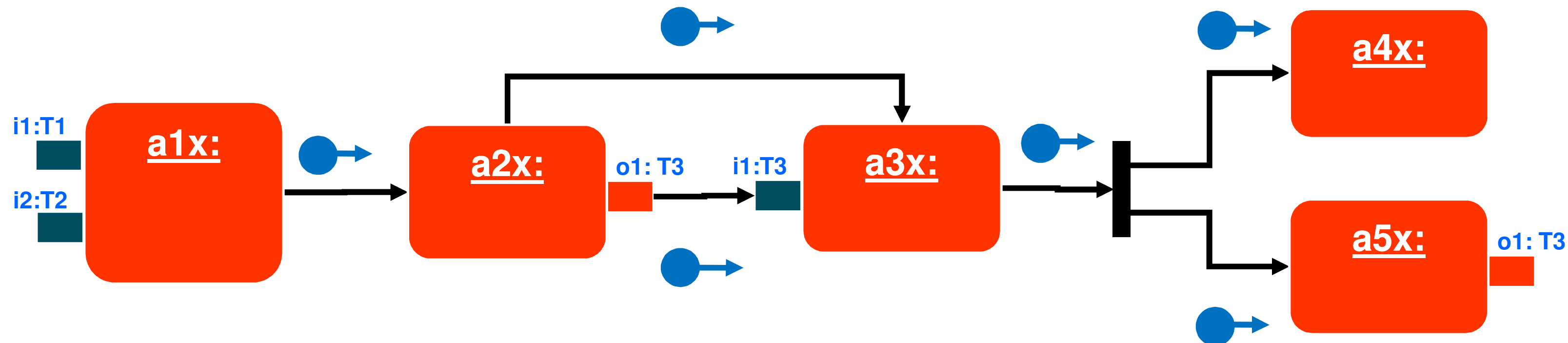
- Control flow MoC: identifying successor actions



© Copyright Malina Software

Controlling Execution: Token Passing

- Execution order can be modeled as an exchange of data/control "tokens" between nodes



- General execution rules:
  - All tokens have to be available before actions execute
  - Tokens are offered only after action execution completes

© Copyright Malina Software

Summary: UML Semantics

- **The UML model of computation is:**
  - Structure dominant
  - Distributed
  - Concurrent
  - Event driven (at the highest level)
  - Data and control flow driven (at finer grained levels)
  - Supports different interaction models
- **The core part of the UML semantics is defined formally**
  - Provides an opportunity for automated formal analyses

© Copyright Malina Software

Tutorial Outline

- On Models and Model-Based Software Engineering
- The Key Dimensions of Modeling Language Design
- Defining a Modeling Language
- Case Study: UML
- Language Refinement: UML Profiles
- Model Transformations

© Copyright Malina Software

UML as a Platform for DSMLs

- **DSML = Domain-Specific Modeling Language**
- **Designed as a "family of modeling languages"**
  - Contains a set of semantic variation points (SVPs) where the full semantics are either unspecified or ambiguous
  - SVP examples:
    - Precise type compatibility rules
    - Communications properties of communication links (delivery semantics, reliability, etc.)
    - Multi-tasking scheduling policies
  - Enables domain-specific customization
- **Open to both extension ("heavyweight" extension) and refinement ("lightweight" extension)**

© Copyright Malina Software

Example: Adding a Semaphore Concept to UML

- **Semaphore semantics:**
  - *A specialized object that limits the number of concurrent accesses in a multithreaded environment. When that limit is reached, subsequent accesses are suspended until one of the accessing threads releases the semaphore, at which point the earliest suspended access is given access.*
- **What is required is a special kind of object**
  - Has all the general characteristics of UML objects
  - …but adds refinements

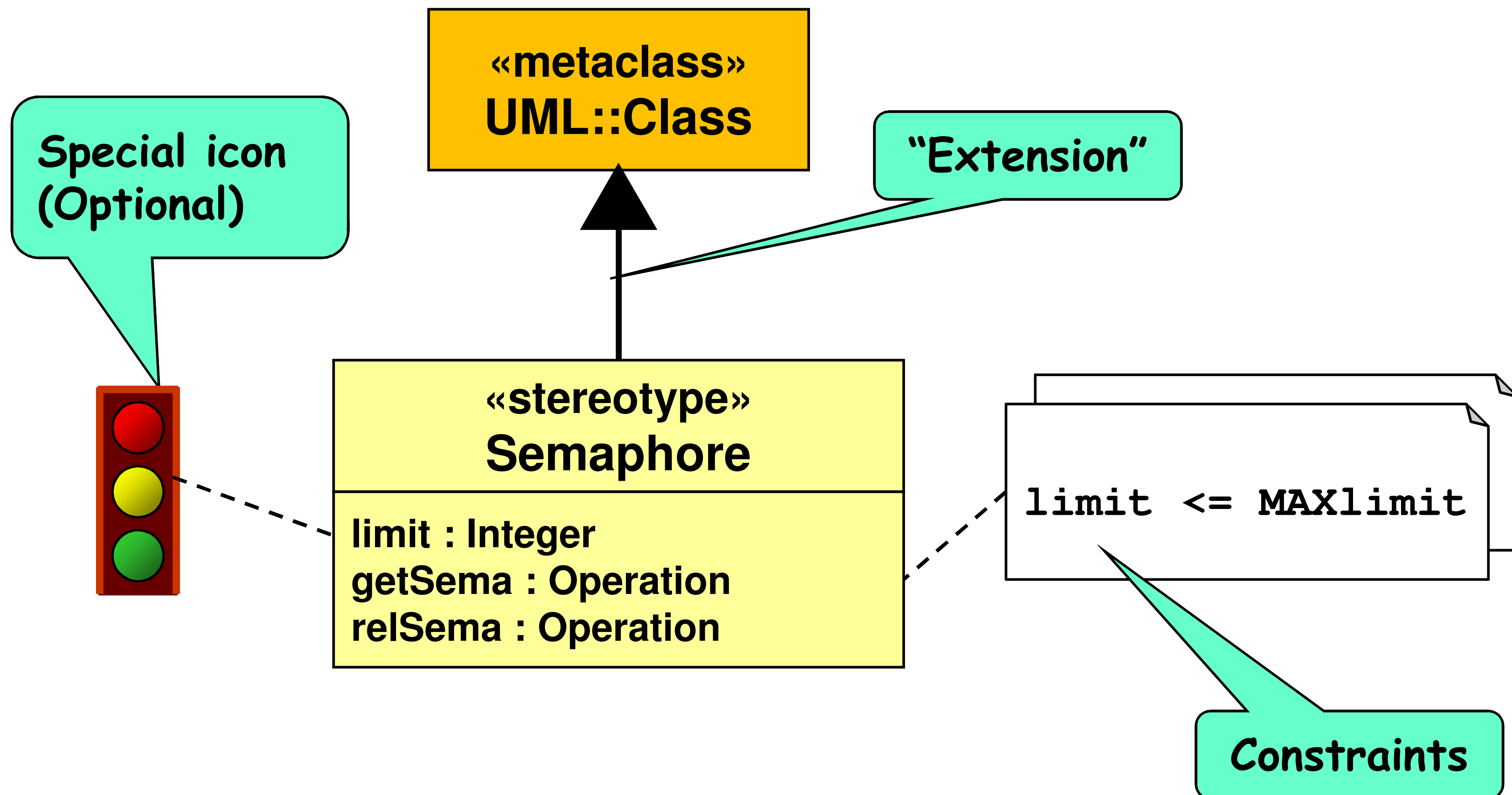
© Copyright Malina Software

Example: The Semaphore Stereotype

- **Design choice: Refine the UML Class concept by**
  - "Attaching" semaphore semantics
    - Done informally as part of the stereotype definition
  - Adding constraints that capture semaphore semantics
    - E.g., when the maximum number of concurrent accesses is reached, subsequent access requests are queued in FIFO order
  - Adding characteristic attributes (e.g., concurrency limit)
  - Adding characteristic operations (getSemaphore (), releaseSemaphore ())
- **Create a new "subclass" of the original metaclass with the above refinements**
  - For technical reasons, this is done using special mechanisms instead of MOF Generalization (see slide Why are Stereotypes Needed?)

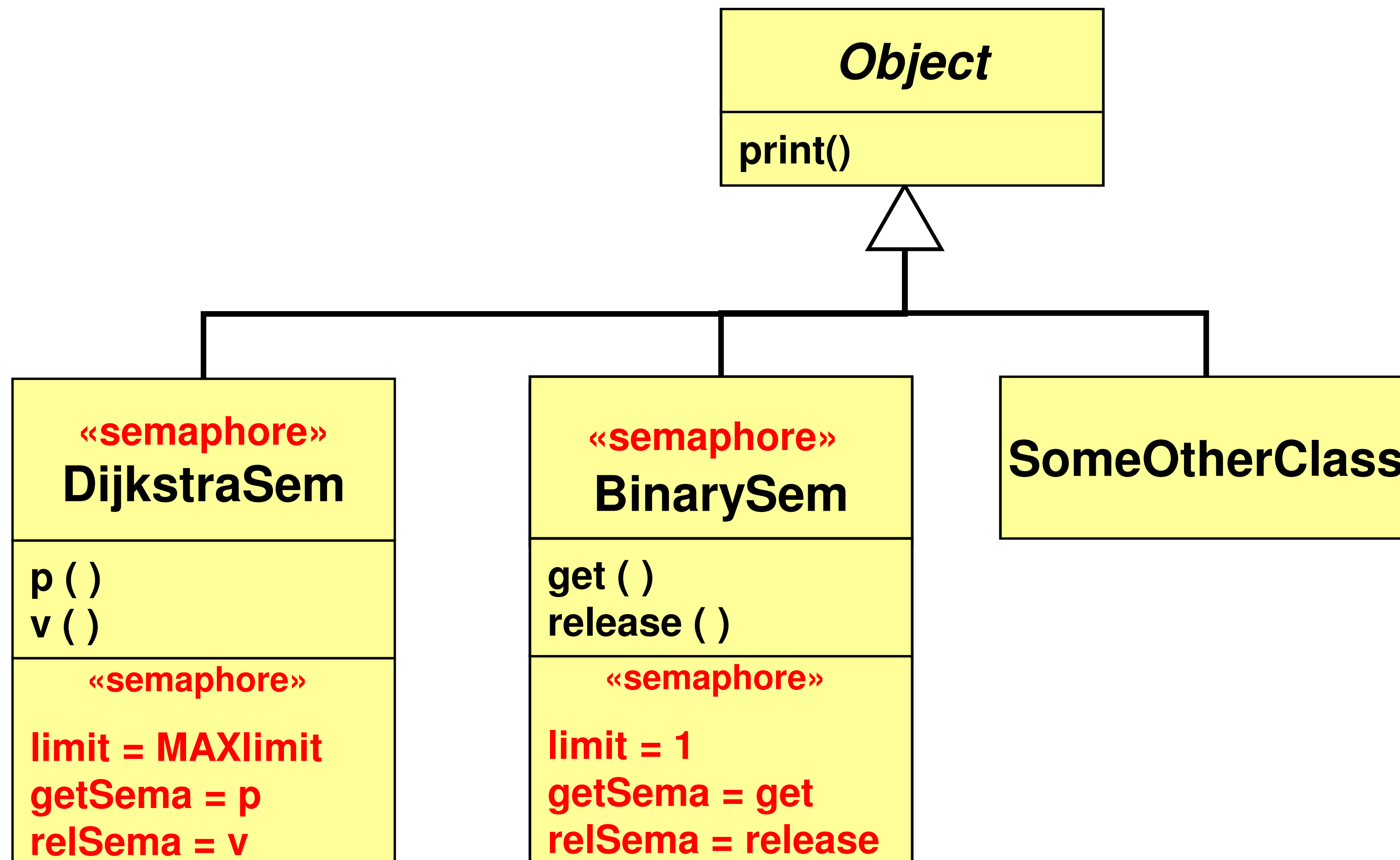
© Copyright Malina Software

Example: Graphical Definition of the Stereotype



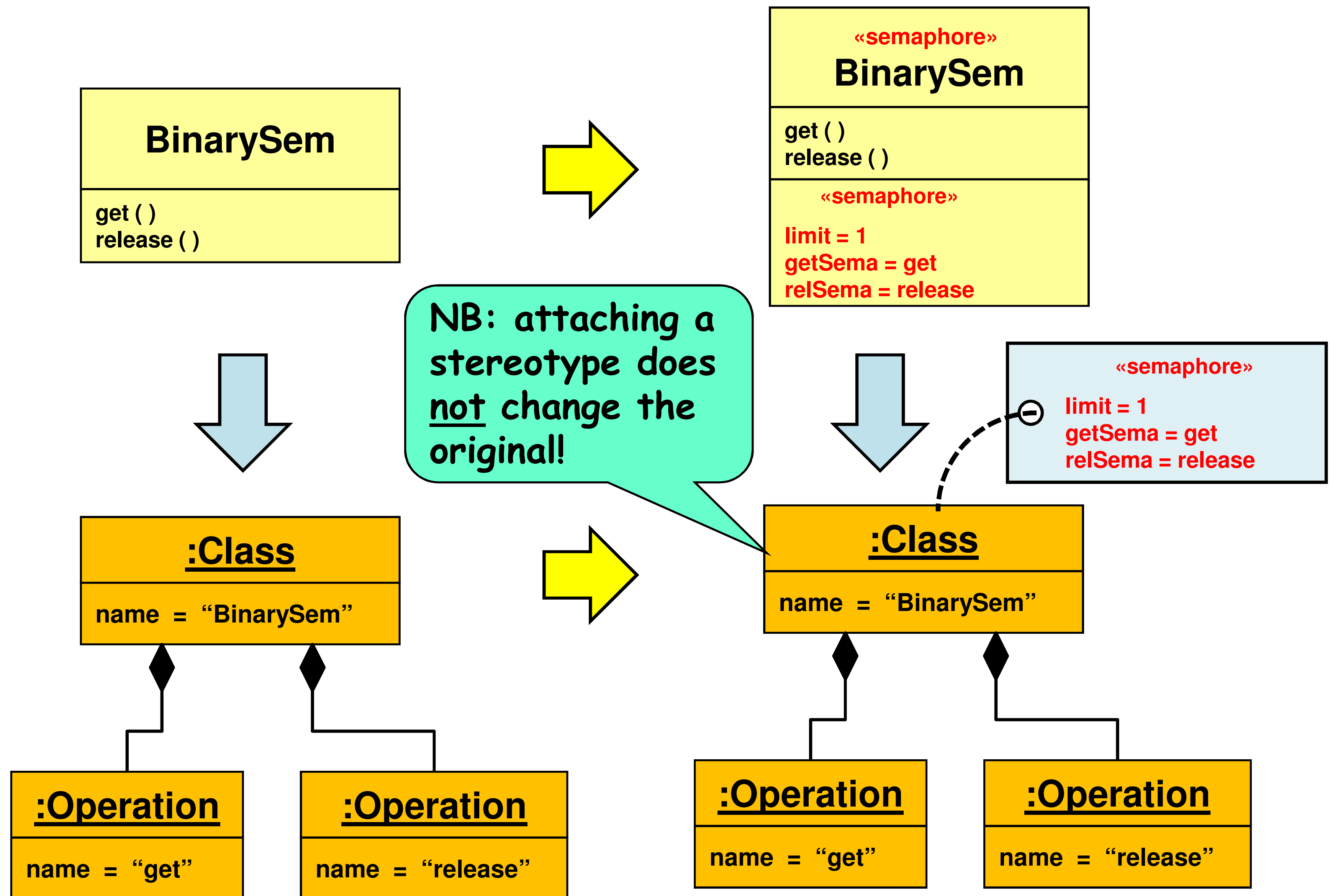
© Copyright Malina Software

Example: Applying the Stereotype



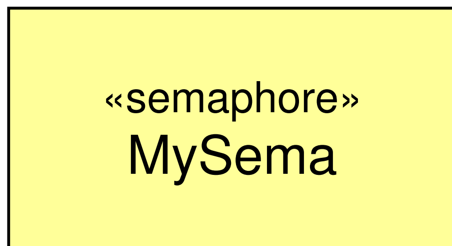
© Copyright Malina Software

The Semantics of Stereotype Application

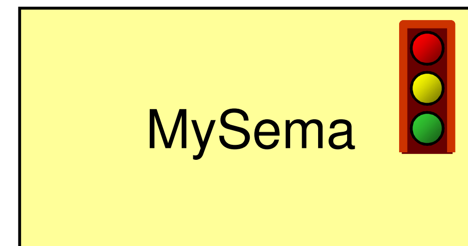


© Copyright Malina Software

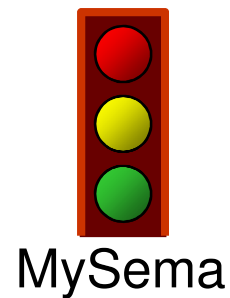
Stereotype Representation Options



(a)



(b)



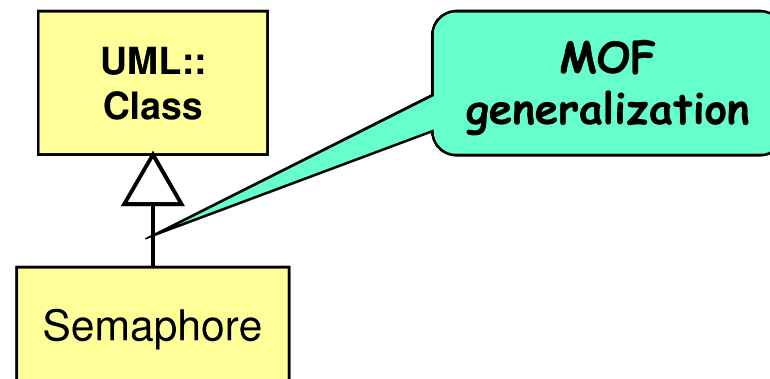
MySema

(c)

© Copyright Malina Software

Why are Stereotypes Needed?

- **Why not simply create a new metaclass?**



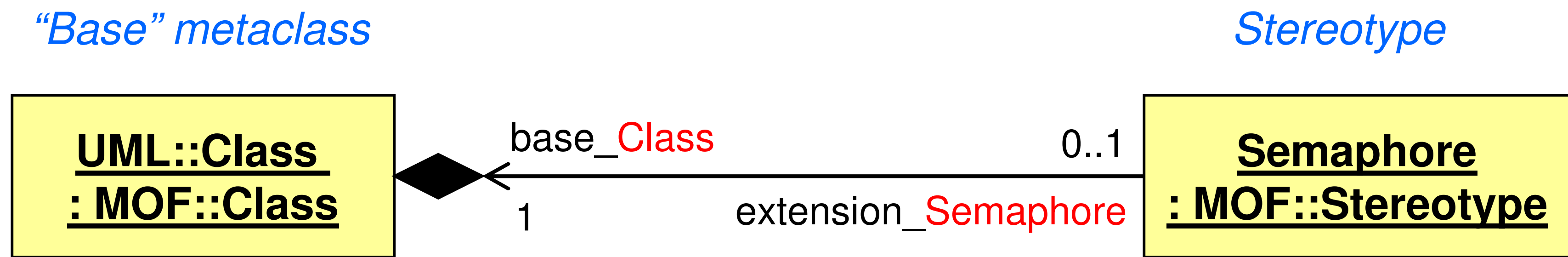
Rationale:

1. Not all modeling tools support meta-modeling ⇒ need to define (M2) extensions using (M1) models
2. Need for special semantics for the extensions:
   - multiple extensions for a single stereotype
   - extension of abstract classes (applicable to all subclasses)

© Copyright Malina Software

The MOF Semantics of UML Extension

- How a stereotype is attached to its base class within a model repository:



- Association ends naming convention:

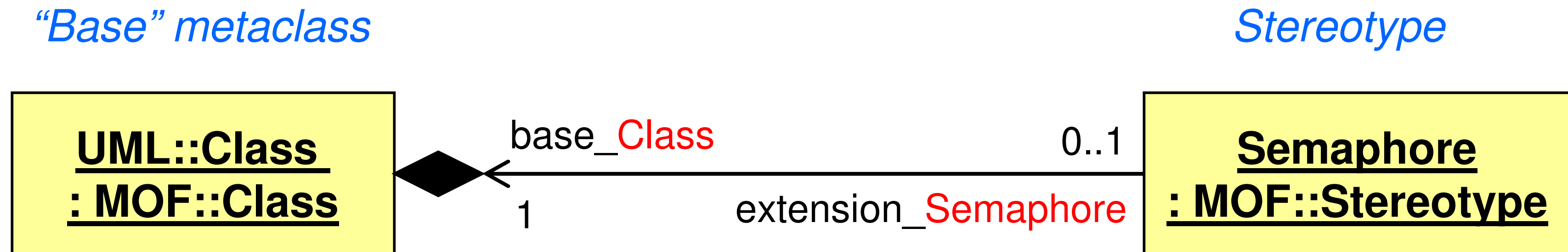
  base_<base-class-name>

  extension_<stereotype-name>

- Required for writing correct OCL constraints for stereotypes

© Copyright Malina Software

Example: OCL Constraint for a Stereotype



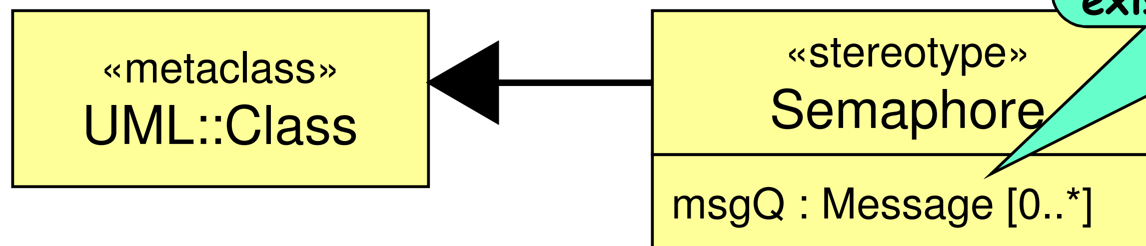
- **Semaphore constraint:**
  the base Class must have an owned ordered attribute called "msgQ" of type Message

```
context Semaphore inv:
   self.base_Class.ownedAttribute->
    exists (a | (a.name = 'msgQ')
          and (a.type->notEmpty())
          and (a.type = Message)
          and (a.isOrdered)
          and (a.upperValue = self.limit))
```


© Copyright Malina Software

Adding New Meta-Associations

- **This was not possible in UML 1.x profiles**
  - Meta-associations represent semantic relationships between modeling concepts
  - New meta-associations create new semantic relationships
  - Possibility that some tools will not be able to handle such additions
- **UML 2.0 capability added via stereotype attribute types:**
  - To be used with care!



Creates an association between Class and Message that does not exist in UML

© Copyright Malina Software

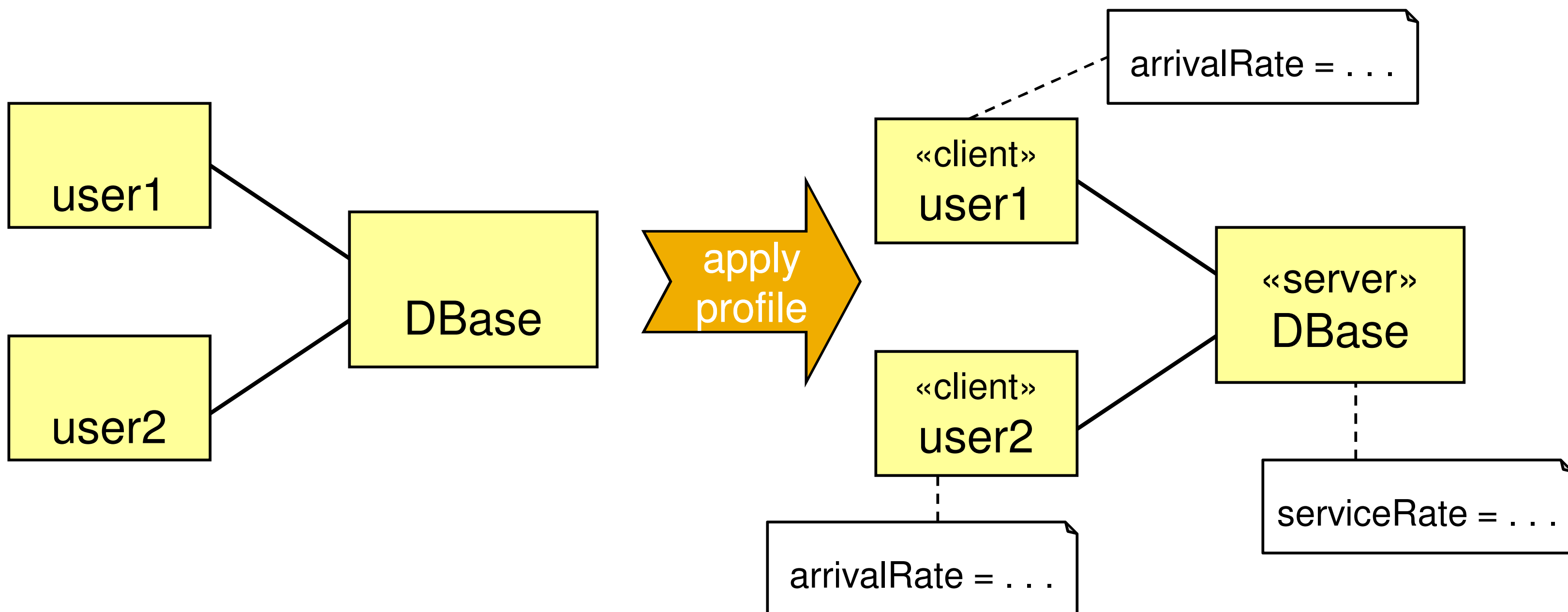
UML Profiles

- **Profile**: ***A special kind of package containing stereotypes and model libraries that, in conjunction with the UML metamodel, define a group of domain-specific concepts and relationships***
  - The profile mechanism is also available in MOF where it can be used for other MOF-based languages
- **Profiles can be used for two different purposes:**
  - To define a domain-specific modeling language
  - To define a domain-specific viewpoint (*cast* profiles)

© Copyright Malina Software

Re-Casting Models Using Profiles

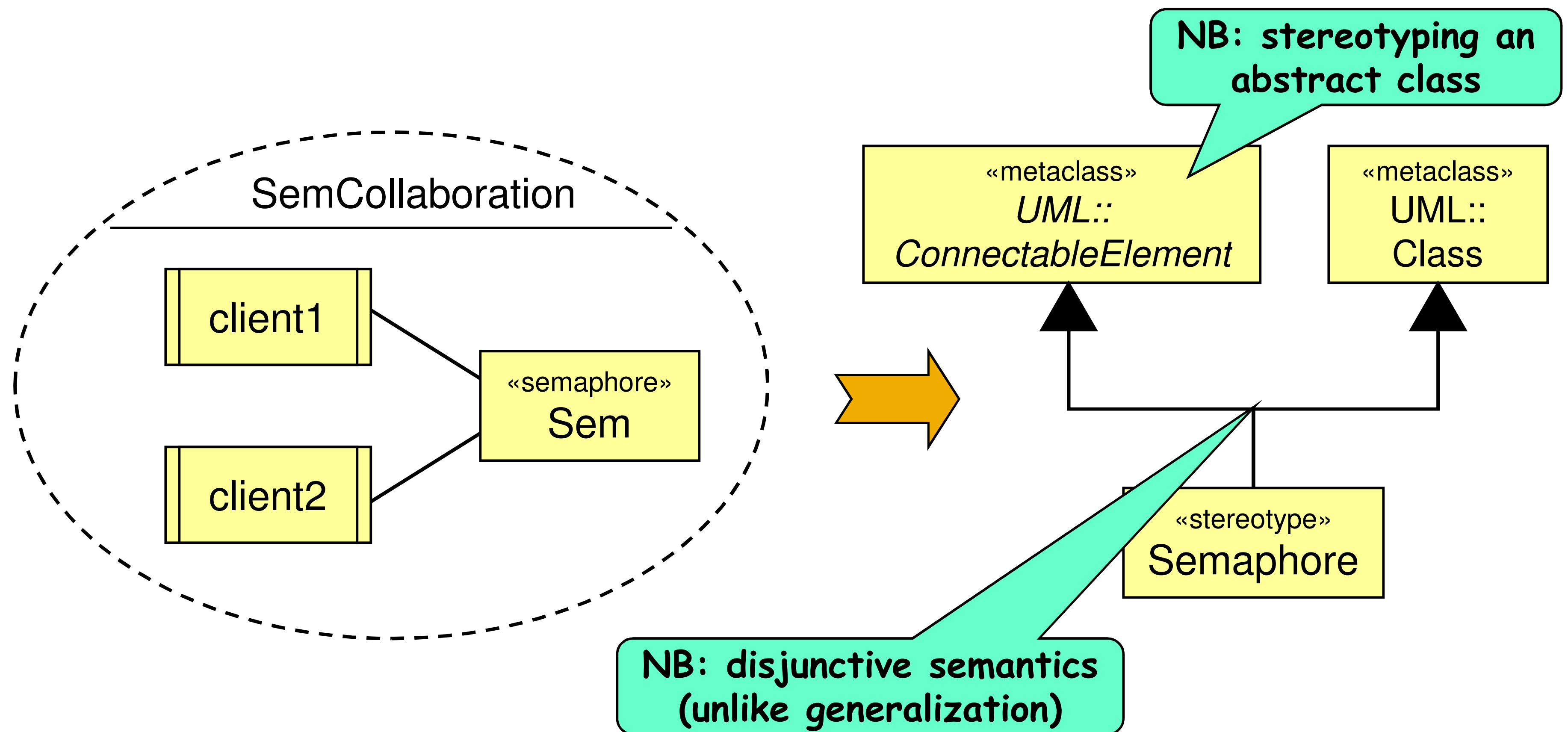
- A profile can be dynamically applied or unapplied to a given model
  - Without changing the original model
  - Allows a model to be interpreted from the perspective of a specific domain
- Example: viewing a UML model fragment as a queueing network



© Copyright Malina Software

Multi-Base Stereotypes

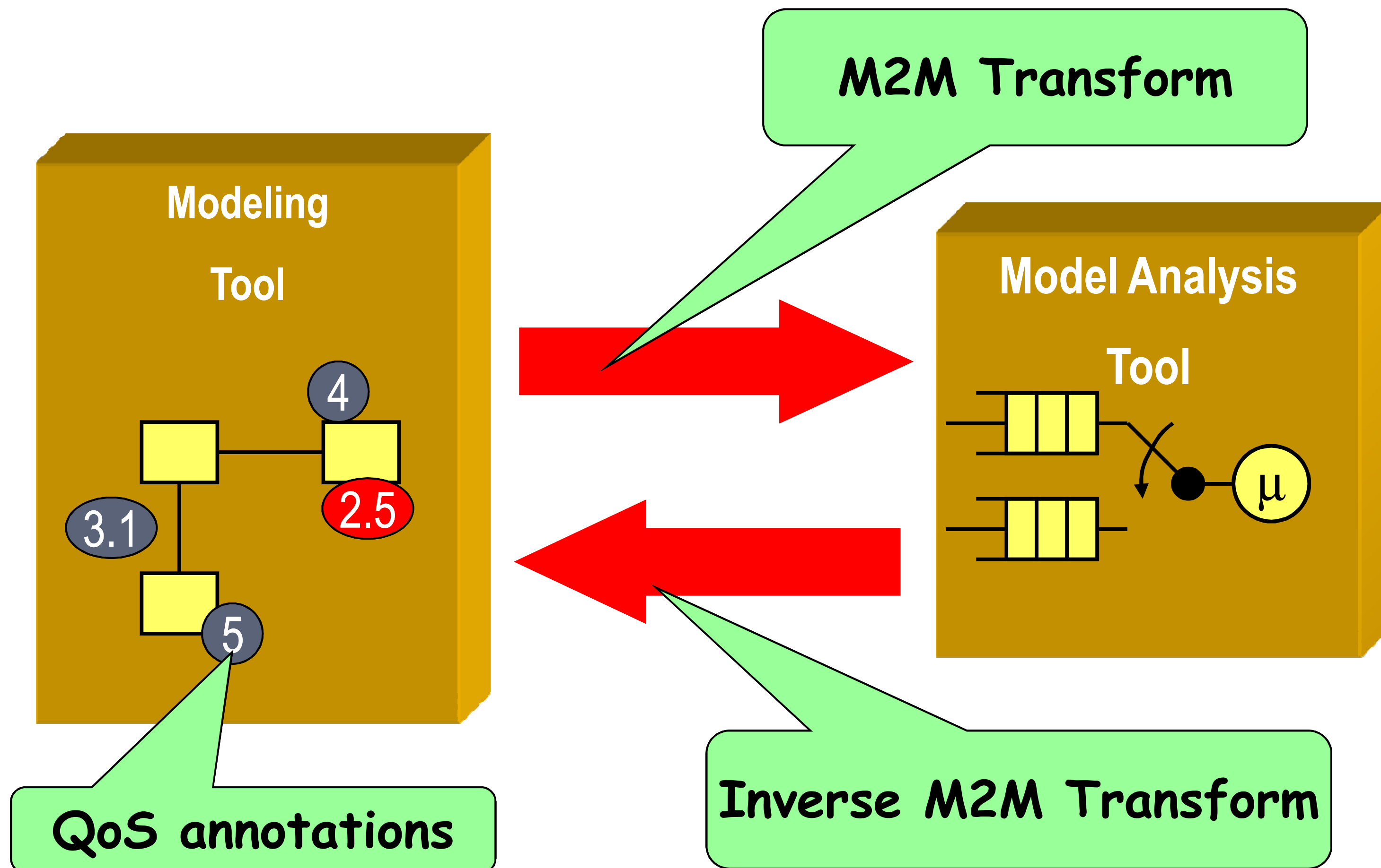
- A domain concept may be a specialization of more than one base language concept



© Copyright Malina Software

Analysis with Cast Profiles

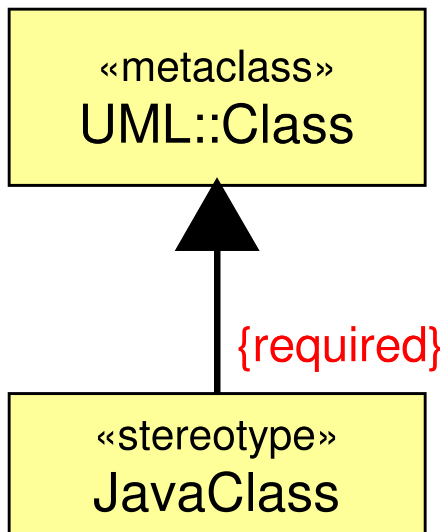
- E.g., recast a model as a queueing network model



© Copyright Malina Software

"Required" Extensions

- **An extension can be marked as "required"**
  - Implies that every instance of the base class will be stereotyped by that stereotype
    - Used by modeling tools to autogenerate the stereotype instances
  - Facilitates working in a DSML context by avoiding manual stereotyping for every case
  - E.g., modeling Java



© Copyright Malina Software

Strict Profile Application

- A *strict* application of a profile will hide from view all model elements that do not have a corresponding stereotype in that profile
  - Convenient for generating views
- Strictness is a characteristic of the profile application and not of the profile itself
  - Any given profile can be applied either way

© Copyright Malina Software

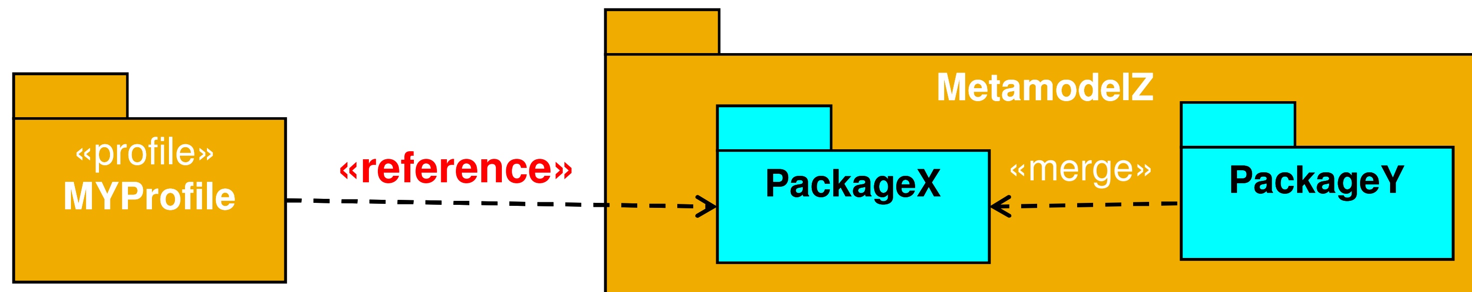
Metamodel Subsetting with Profiles (1)

- **It is often useful to remove segments of the full UML metamodel resulting in a minimal DSML definition**
  - NB: Different mechanism from strict profile application – the hiding is part of the profile definition and cannot be applied selectively
- **The UML 2.1 profile mechanism adds controls that define which parts of the metamodel are used**
  - Based on refinement of the package import and element import capabilities of UML

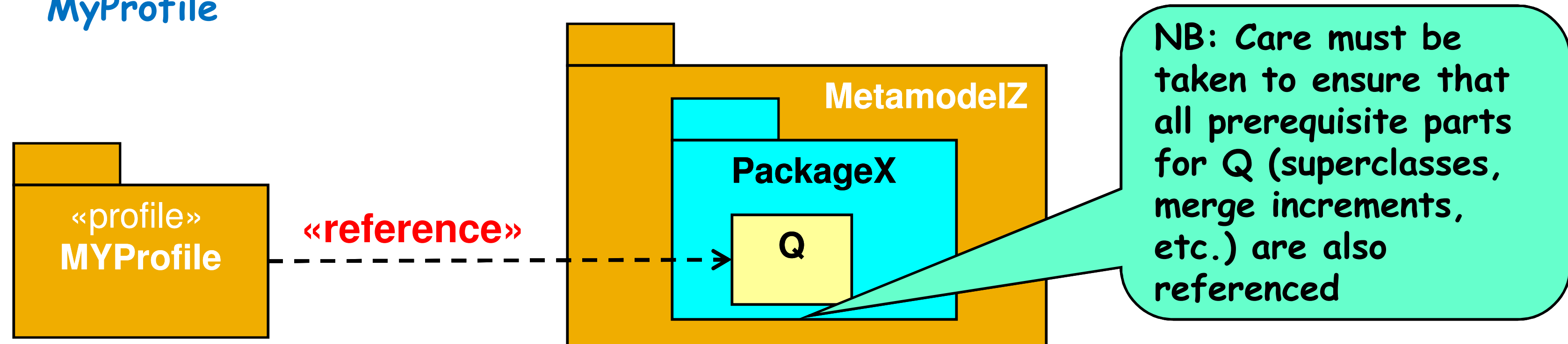
© Copyright Malina Software

Metamodel Subsetting with Profiles (2)

- **Case 1: Metamodel Reference**
  - All elements of the referenced MOF package (PackageX) are visible (but not the elements of PackageY)
  - These elements can also serve as the base metaclasses for stereotypes in MyProfile



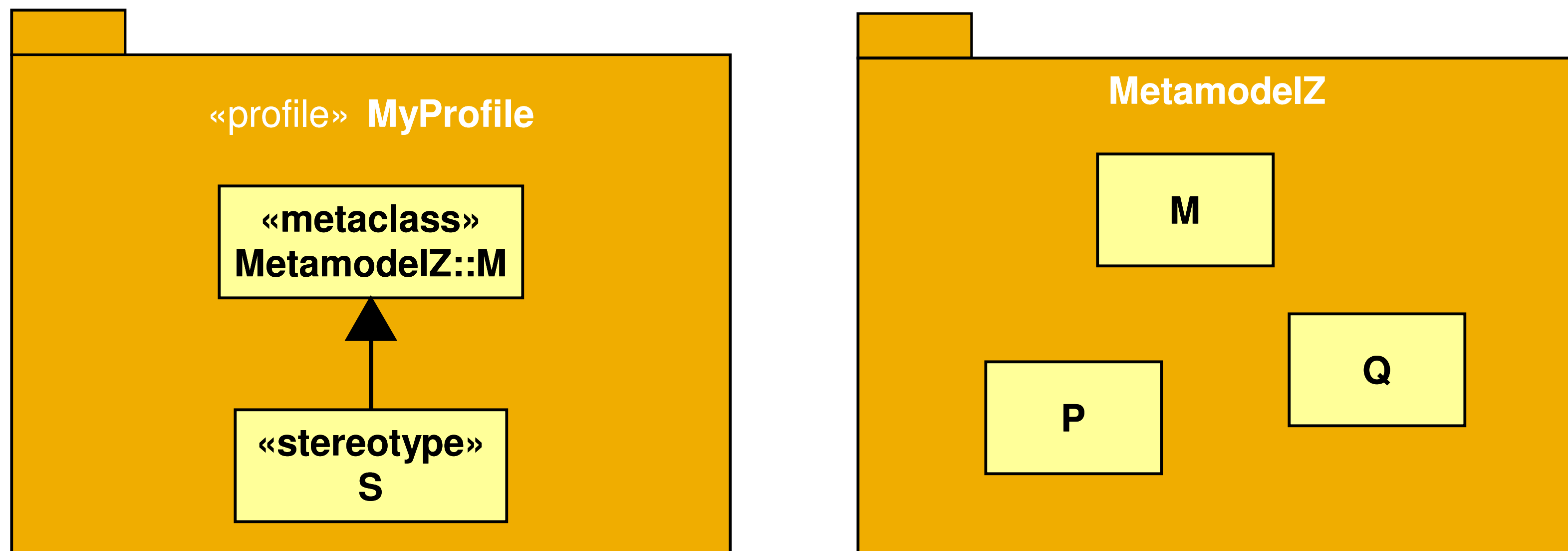
- **Case 2: Explicit Metaclass Reference**
  - Metaclass Q is visible and can serve as a base metaclass for stereotypes in MyProfile



© Copyright Malina Software

Metamodel Subsetting with Profiles (3)

- **Case 3: Implicit metaclass reference**
  - Metaclass M is visible



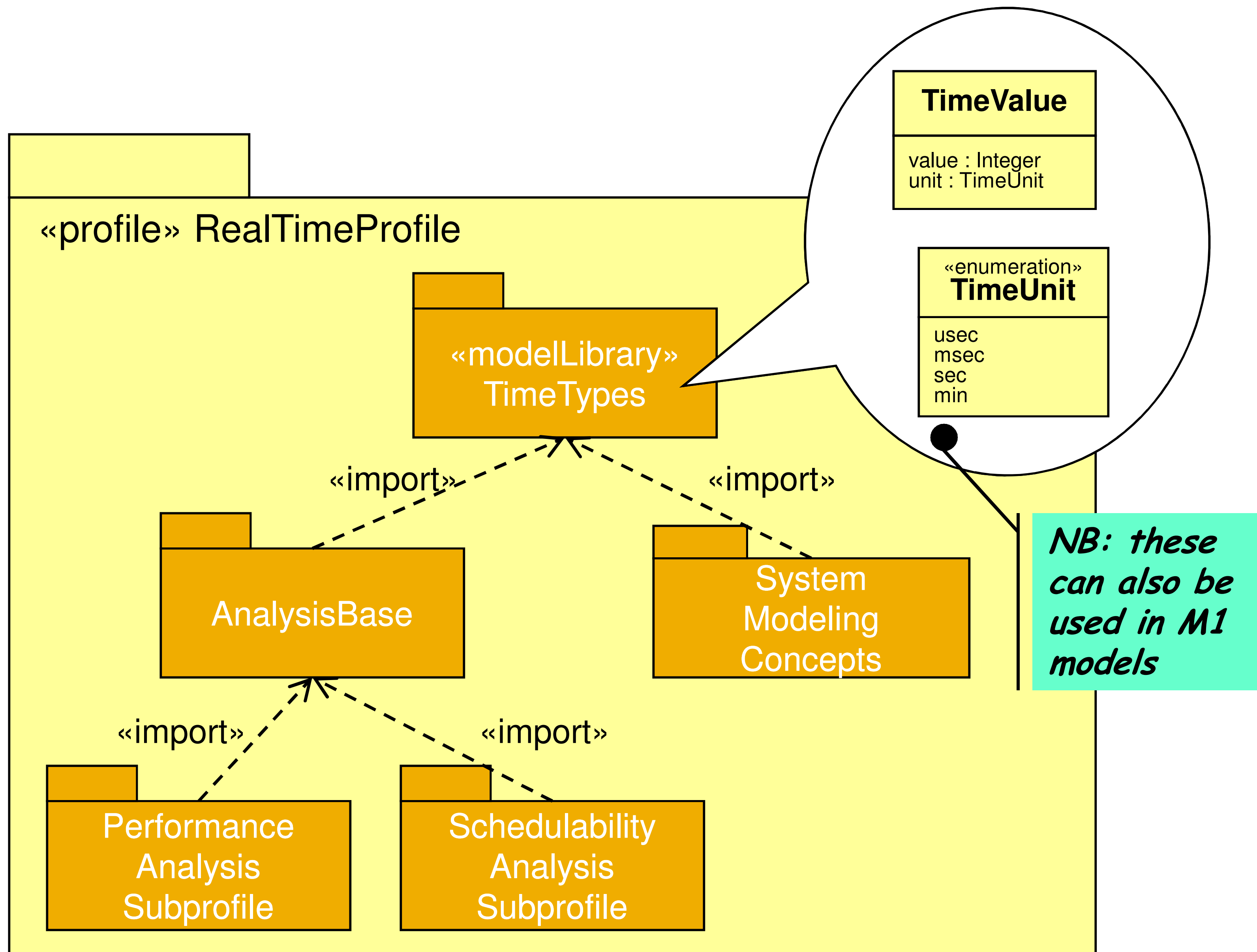
© Copyright Malina Software

Model Libraries

- M1 level model fragments packaged for reuse
  - Identified by the «modelLibrary» standard stereotype
- Can be incorporated into a profile
  - Makes them formally part of the profile definition
    - E.g., define an M1 "Semaphore" class in a library package and include the package in the profile
  - The same implicit mechanism of attaching semantics used for stereotypes can be applied to elements of the library
  - Overcomes some of the limitations of the stereotype mechanism
  - Can also be used to type stereotype attributes
- However, it also precludes some of the advantages of the profiling mechanism
  - E.g., the ability to view a model element from different viewpoints
- Model libraries should be used to define useful types shared by two or more profiles or profile fragments as well as by models at the M1 level

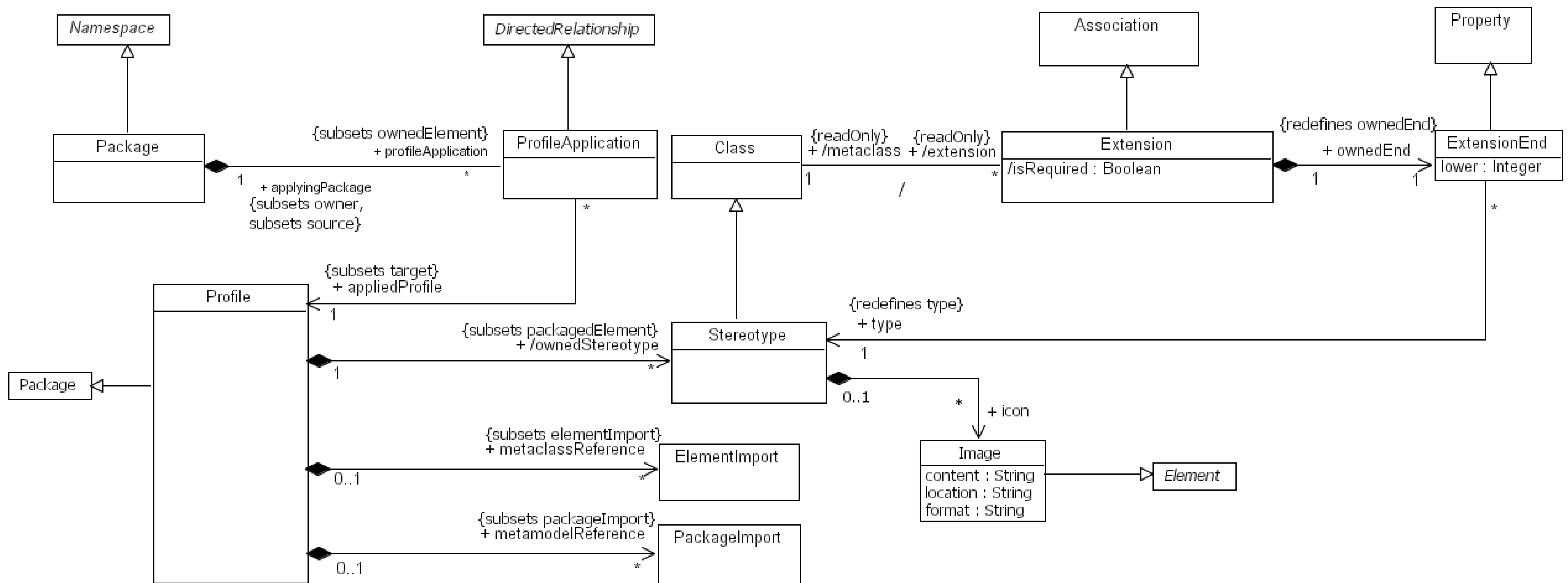
© Copyright Malina Software

Example: Model Library



© Copyright Malina Software

The UML Profile Metamodel



© Copyright Malina Software

Guidelines for Defining Profiles

- **Always define a pure domain model (using MOF) first and the profile elements second**
  - Allows separation of two different concerns:
    - What are the right concepts and how are they related?
    - How do the domain-specific concepts map to corresponding UML concepts?
  - Mixing these two concerns often leads to inadequate profiles
- **For each domain concept, find the UML concept(s) that most closely match and define the appropriate stereotype**
  - If no matching UML concept can be found, a UML profile is probably unsuitable for that DSML
  - Fortunately, many of the UML concepts are quite general (object, association) and can easily be mapped to domain-specific concepts

© Copyright Malina Software

Matching Stereotypes to Metaclasses

- **A suitable base metaclass implies the following:**
  - Semantic proximity
    - The domain concept should be a special case of the UML concept
  - No conflicting well-formedness rules (OCL constraints)
  - Presence of required characteristics and (meta)attributes
    - e.g., multiplicity for domain concepts that represent collections
    - New attributes can always be added but should not conflict with existing ones
  - No inappropriate or conflicting characteristics or (meta)attributes
    - Attributes that are semantically unrelated to the domain concept
    - These can sometimes be eliminated by suitable constraints (e.g., forcing multiplicity to always have a value of 1 or 0)
  - Presence of appropriate meta-associations
    - It is possible to define new meta-associations
  - No inappropriate or conflicciting meta-associations
    - These too can be eliminated sometimes by constraints

© Copyright Malina Software

Beware of Syntactic Matches!

- Avoid seductive appeal of a syntactic match
  - In particular, do not use things that model M1 entities to capture M0 elements and vice versa
    - Example: using packages to represent groupings of run-time entities
    - Example: using connector and part structures to capture design time dependencies (e.g., requirements dependencies)
- This may confuse both tools and users

© Copyright Malina Software

Catalog of Adopted OMG Profiles

- UML Profile for CORBA
- UML Profile for CORBA Component Model (CCM)
- UML Profile for Enterprise Application Integration (EAI)
- UML Profile for Enterprise Distributed Object Computing (EDOC)
- UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms
- UML Profile for Schedulability, Performance, and Time
- UML Profile for System on a Chip (SoC)
- UML Profile for Systems Engineering (SysML)
- UML Testing Profile
- UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)
- UML Profile for DoDAF/MoDAF (UPDM)

© Copyright Malina Software

Bibliography

- OMG UML Profiles specifications
  - http://www.omg.org/technology/documents/profile_catalog.htm

© Copyright Malina Software

Tutorial Outline

- On Models and Model-Based Software Engineering
- The Key Dimensions of Modeling Language Design
- Defining a Modeling Language
- Case Study: UML
- Language Refinement: UML Profiles
- Model Transformations

© Copyright Malina Software

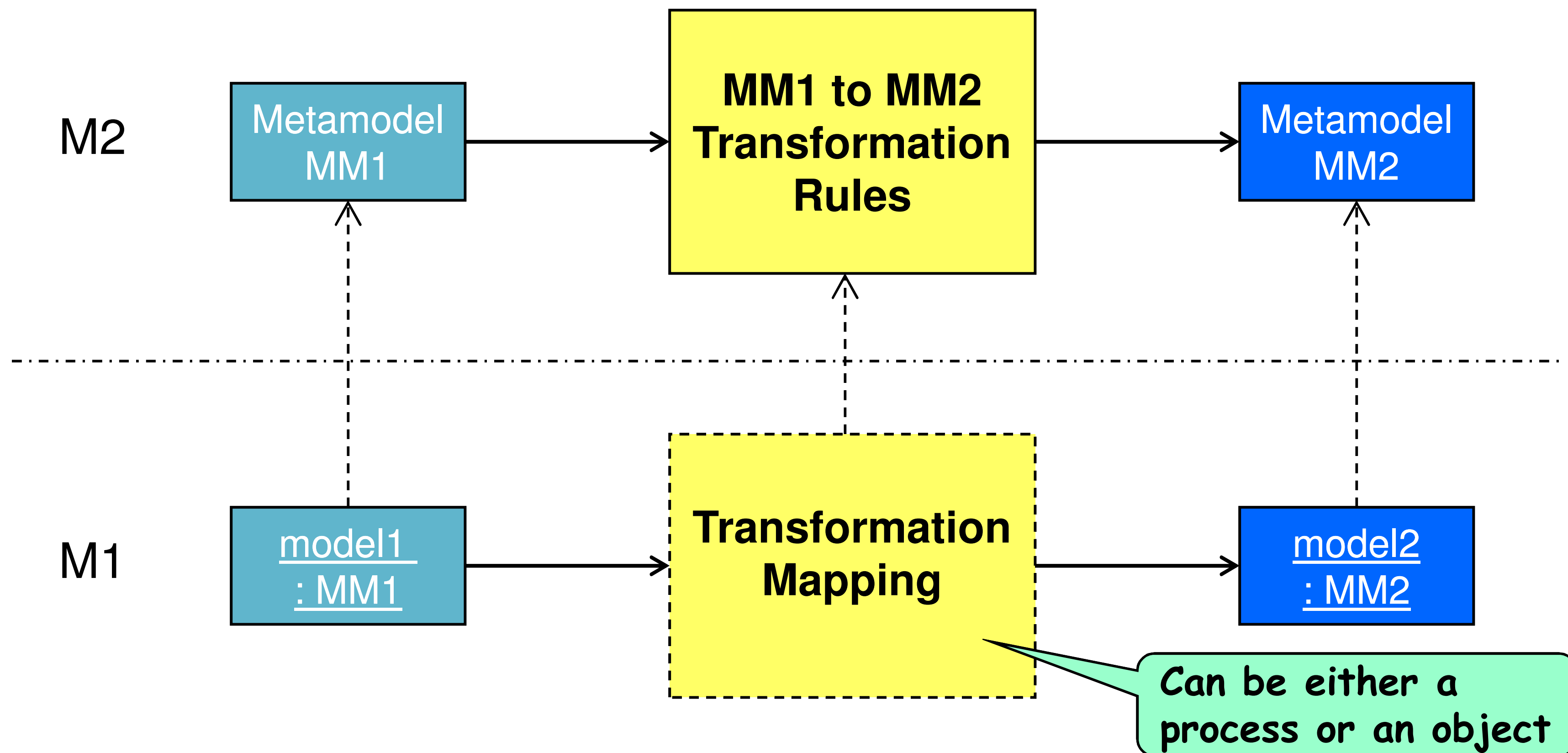
Model Transformations: Purpose

- **Generating a new model from a source model—according to formally defined rules—to:**
  1. Extract an interesting subset of the source model (Query)
     - Example: Find all classes that have multiple parents
  2. Generate a new model, based on a different metamodel, that is "equivalent" to the source model (Transformation)
     - Example: Create a queueing network model of a UML model to facilitate performance analysis
     - Example: UML to Java transformation for code generation
     - Definition of "equivalence" depends on the particular case
  3. Represent the source model from a particular viewpoint (View)
     - In effect, just a special case of Transformation

© Copyright Malina Software

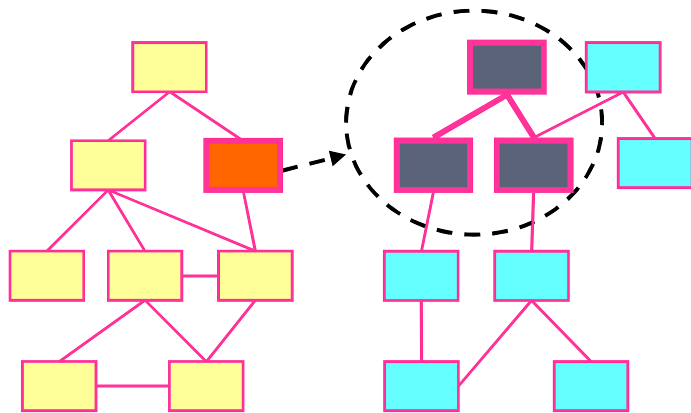
A Basic Representation of Model Transformation

- **Source to target mapping based on pre-defined transformation rules**
  - Conceptually similar to source code compilation

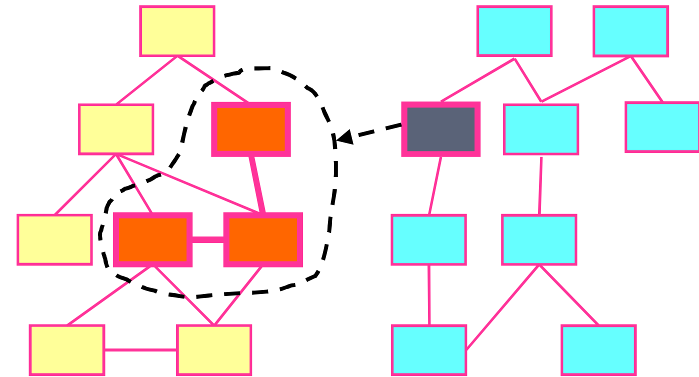


© Copyright Malina Software

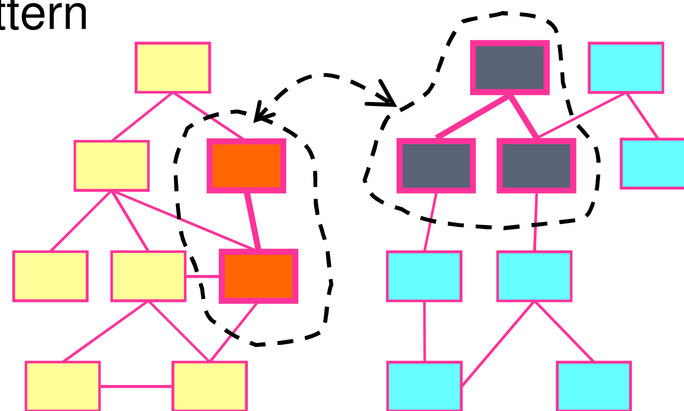
Model Transformations: Styles



(1) Source element
to target pattern



(2) Source pattern
to target element



(3) Source pattern
to target pattern

© Copyright Malina Software

The Traditional Template Approach

- A basic "element-to-pattern" style
  - Generate a target pattern for each element of the source model
  - Example (S. Mellor: "archetype" language that generates a Java class):

```
.for each object in O_OBJ
 public class ${object.name} extends StateMachine
         private StateMachineState currentState;
.select many attributes related by object->O_ATTR[R_105]
.for each attribute in attributes
 private ${attribute.implType} ${attribute.name};
.end for
 .
 .
 .
.select many signals related by object->SM_SM[R301]->SM_EVT[R303]

.for each signal in  signals
 protected void ${signal.name} () throws ooadException;
 .
 .
 .
.end for }
.emit to file ${object.name}.java
.end for
```

- Open source example: Jave Emitter Templates (JET) in Eclipse

© Copyright Malina Software

Some Drawbacks of the Template Approach

- Primarily intended for model-to-code transforms
  - Optimized for working with strings
  - Model-to-model transforms possible but may be difficult to specify
- Cons:
  - E.g., no dedicated support for a system-level optimization pass
  - Unidirectional: no built-in support for inverse mapping
  - Localized perspective: no support for incremental transformation ("recompile the world" for every change, regardless of scope)
  - Serialized transformation process (like Cfront): No ability to exploit application level knowledge for optimization

© Copyright Malina Software

MDA Transformation Standards

- MOF to XMI
- MOF to JMI
- MOF 2 Queries/Views/Transformations
- MOF to Text

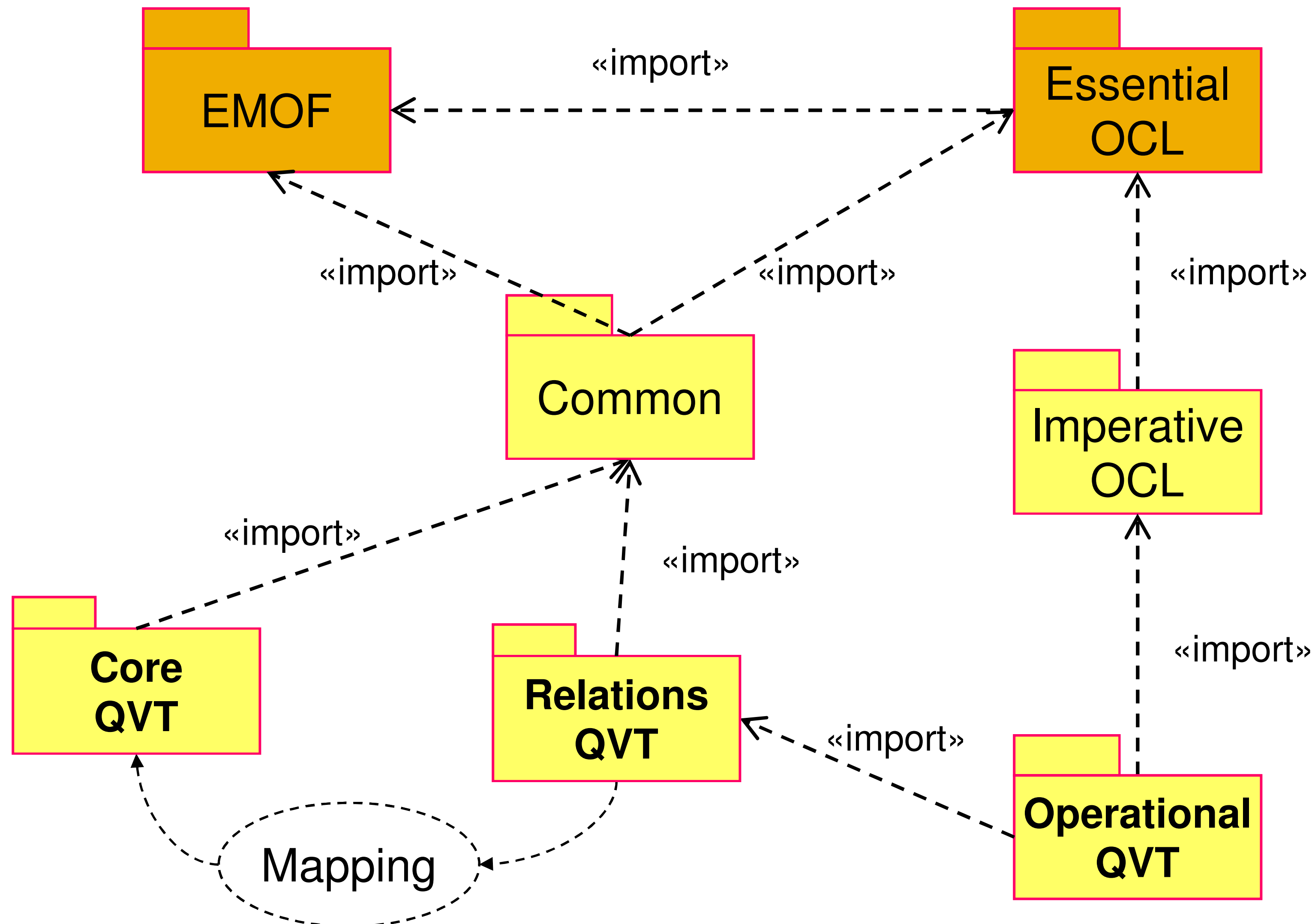
© Copyright Malina Software

QVT Basics

- Three types of transformations defined:
  - Core
  - Relations
  - Operational Mappings
- The first two forms are declarative the third is imperative (procedural)
  - Core is a "minimal" form based on minor extensions to OCL and EMOF
  - Relations is more user-friendly but may be less efficient
  - The standard defines a formal mapping from Relations to Core
- Operational style provides more flexibility but is more complex
- All allow invocation of external ("black box") transforms

© Copyright Malina Software

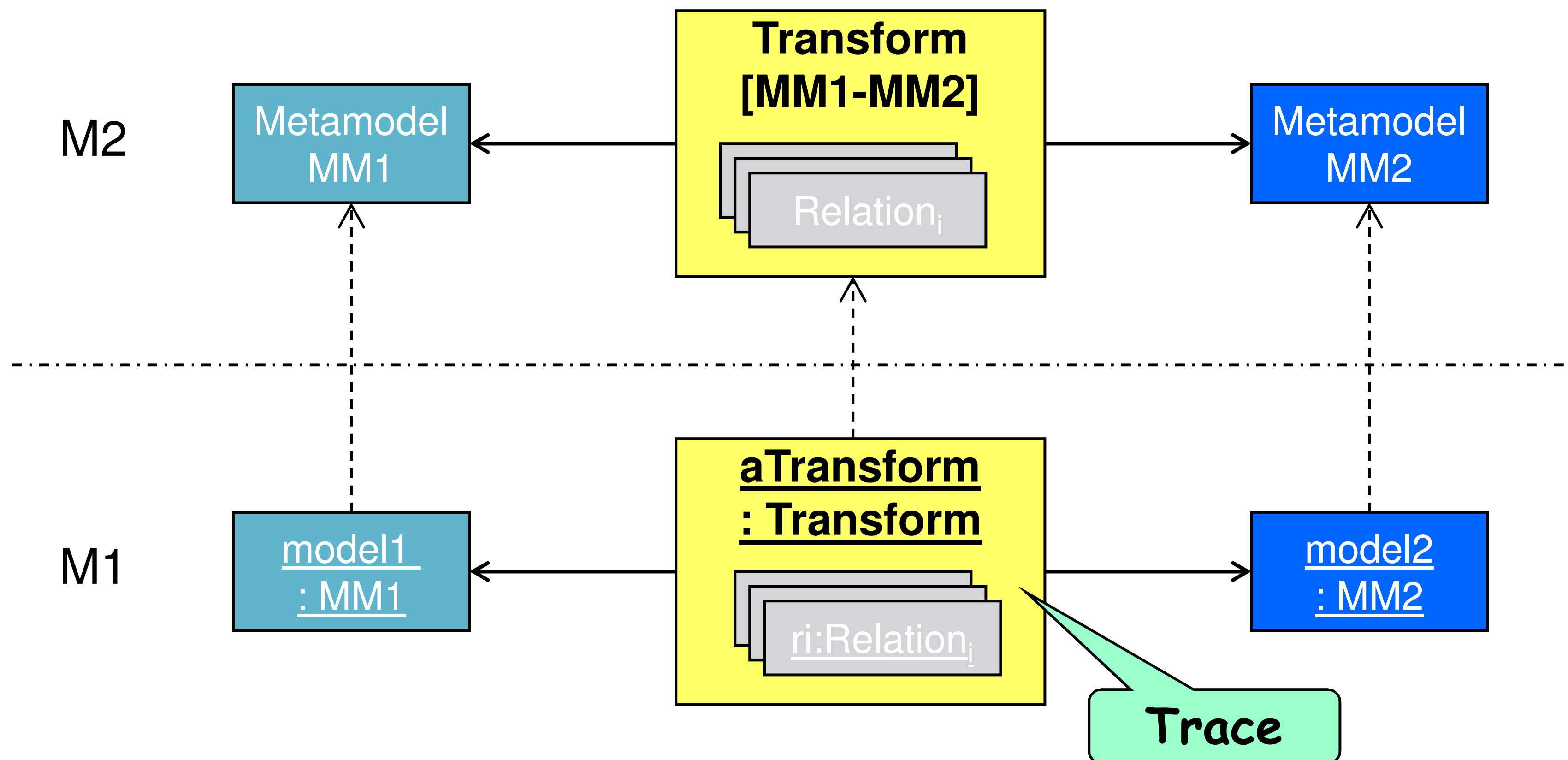
QVT Metamodel Structure (simplified)



© Copyright Malina Software

A Generalized Model of QVT Model Transformations

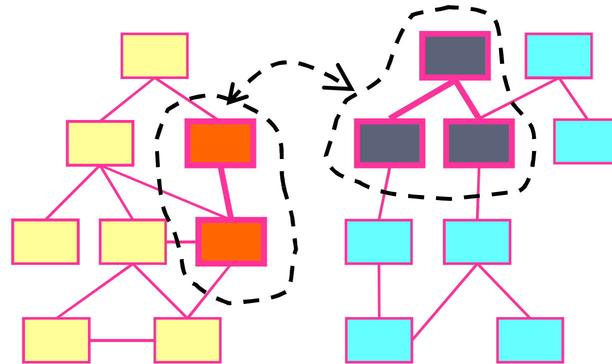
- Transformations can be viewed as an instantiable collection of relations between metamodel patterns



© Copyright Malina Software

Relations and Transformations

- A relation specifies a matching between two (or more) model patterns in respective models



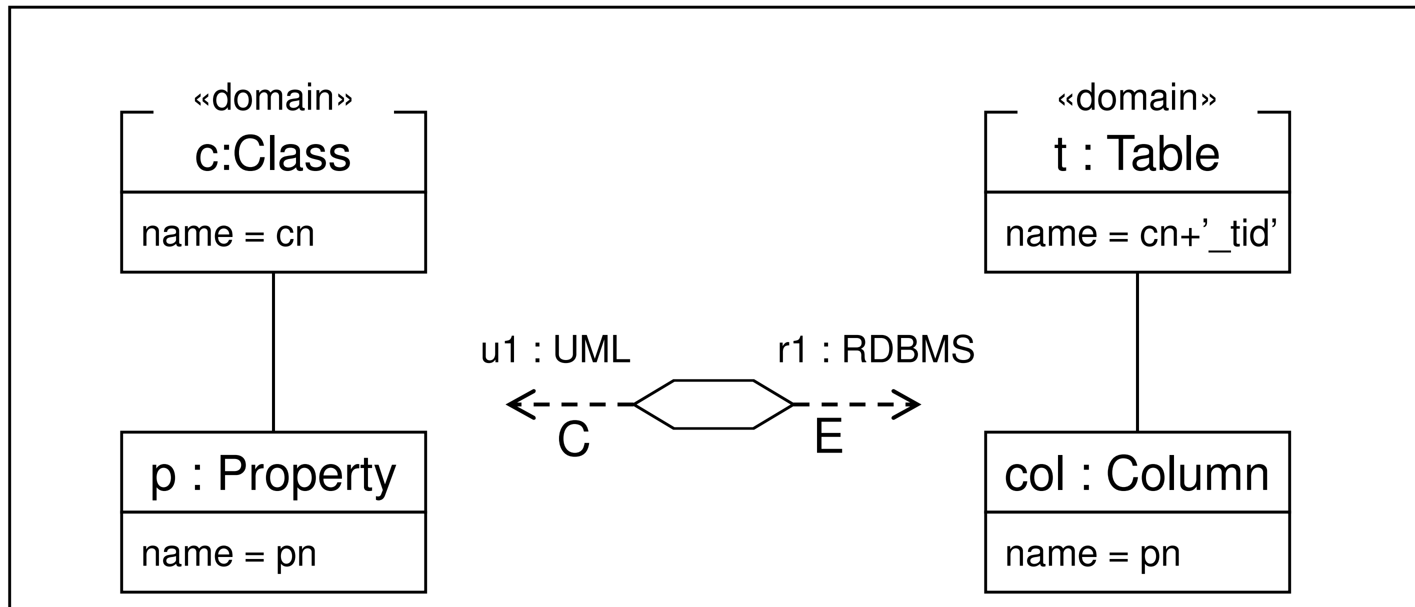
  - A pattern (domain) in a relation can be designated as
    - "checkonly" – only detects and flags violations of relation
    - "enforced" – modifies corresponding model to assure relation holds
  - A relation may invoke (depend on) other relations
  - A transformation that must hold consists of one or more "top-level" relations that must hold
    - …and a set of relations invoked (transitively) by top-level relations

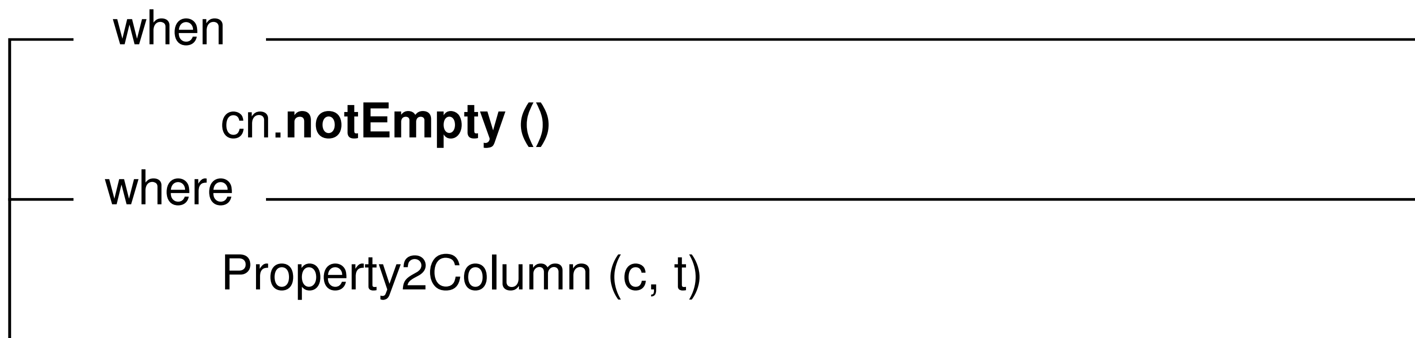
© Copyright Malina Software

Relational Transforms: Graphical Syntax

- Only practical for binary relations

Class2Table





© Copyright Malina Software

Relational Transforms: Textual Syntax

- Equivalent textual specification:

```
relation Class2Table {
   checkonly domain u1:UML c:Class {
        name = cn, p:Property {name = pn}}
   enforce domain r1:RDBMS t:table {
        name = cn + '_tid',
        col:Column {name = pn }}
   when { cn.notEmpty() }
   where { Property2Column (c, t) }}
```

- Any number of domains (patterns) can be included in a relation

© Copyright Malina Software

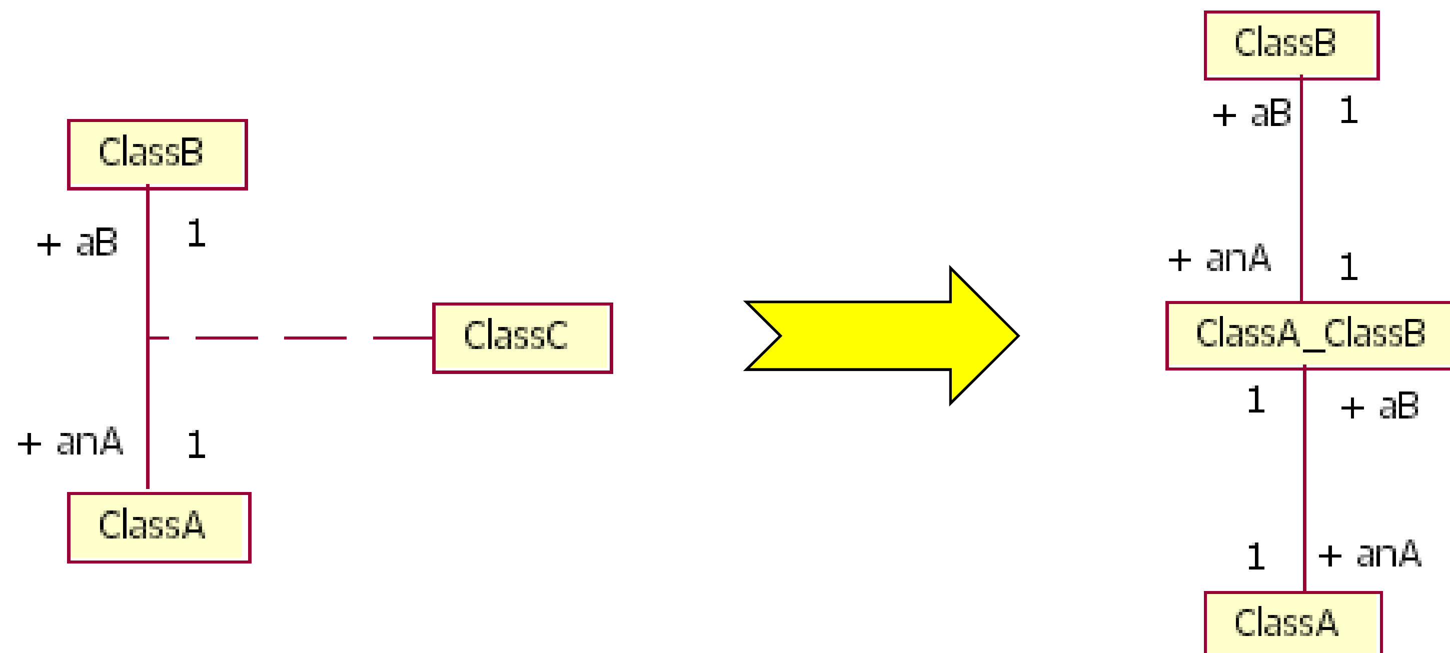
Transformation Relations as Objects

- Model transformation can be viewed as enforcement of pre-defined formal relations between two or more models
- This can be achieved by instantiating a set of "relation" objects (traces) that enable continuous transformation
  - If one model changes, trace objects react to ensure that the declared relations between models always hold (for enforced domains)
  - Only the necessary changes are made $\Rightarrow$ incremental change model

© Copyright Malina Software

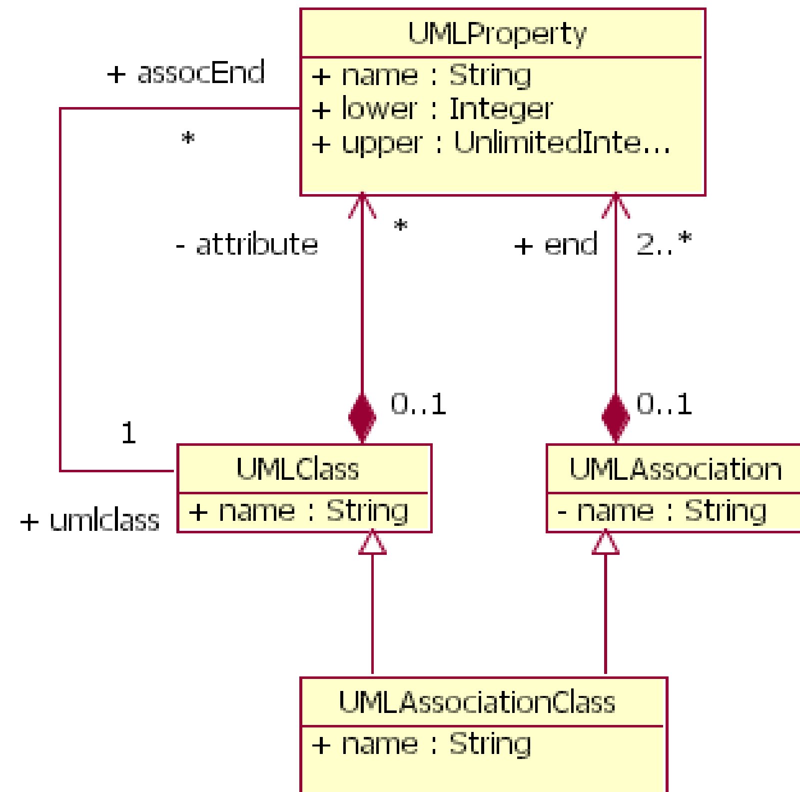
Example: UML to MOF Model Transform

- **Translate a UML-based metamodel into a proper MOF metamodel**
  - e.g., MOF does not support association classes



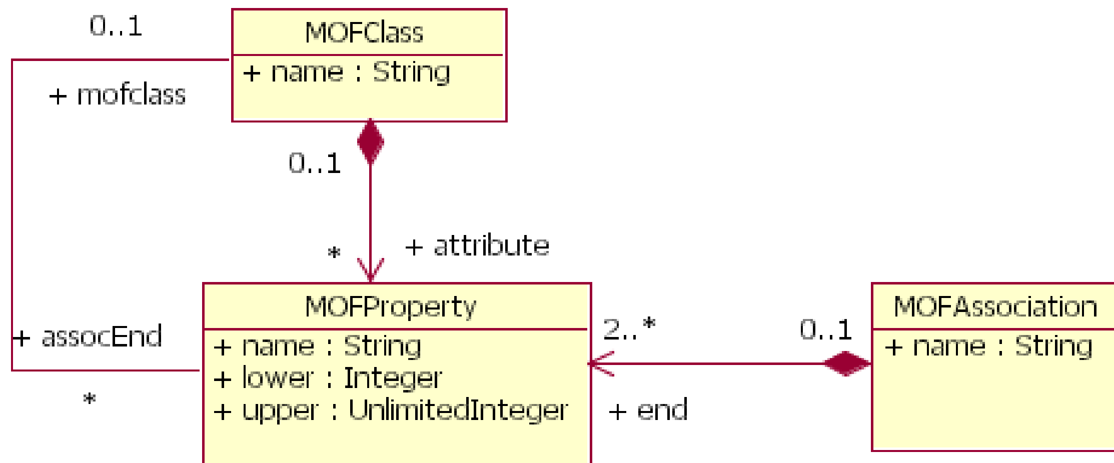
© Copyright Malina Software

Example: UML Metamodel (Simplified Fragment)



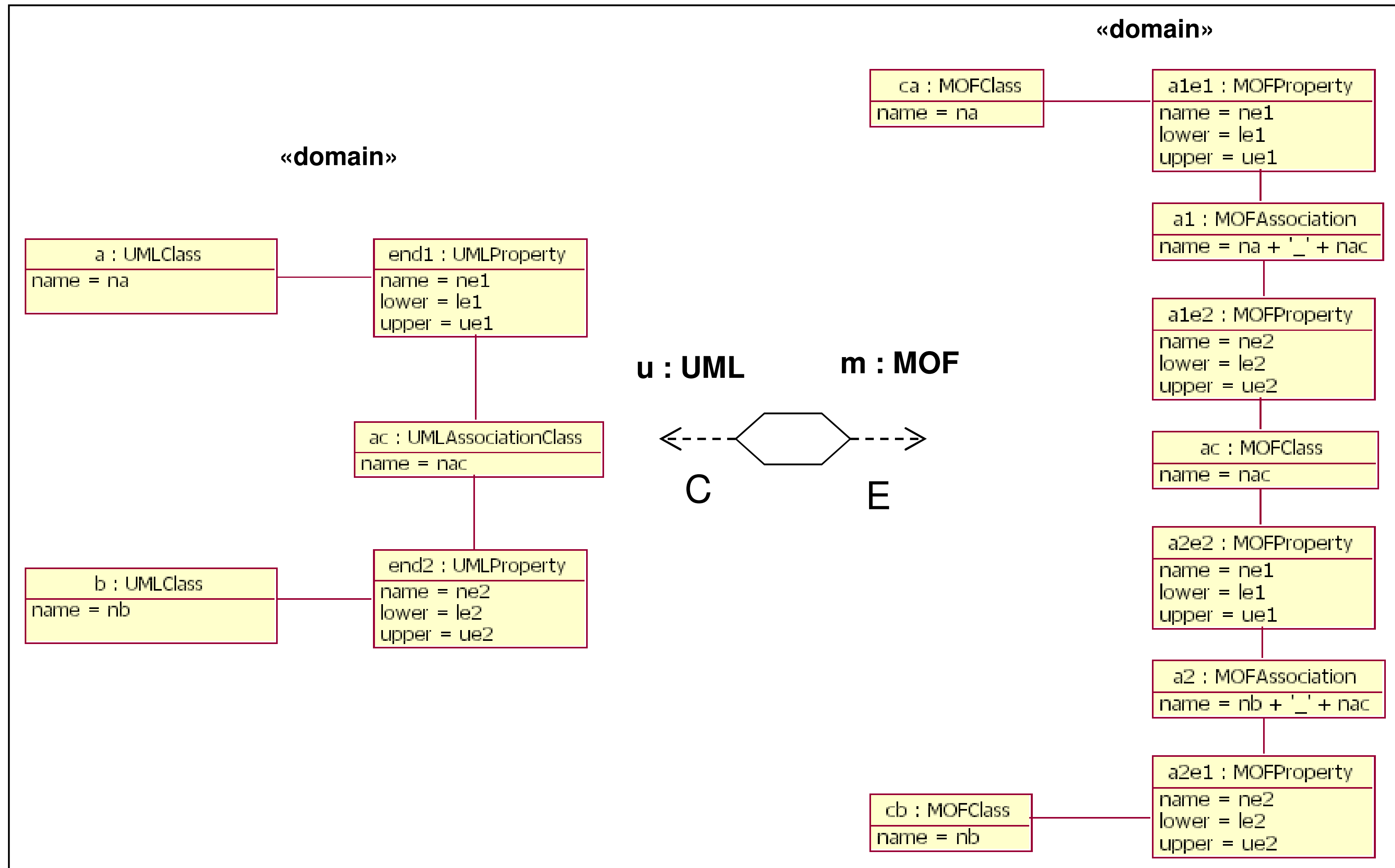
© Copyright Malina Software

Example: MOF Metamodel (Simplified Fragment)



© Copyright Malina Software

Example: Relationship Definition



© Copyright Malina Software

Alternative: Operational Mappings Approach

- A "how" (vs "what") approach to transformation
  - Unidirectional: source and target clearly identified
  - However: support for incremental transforms
    - Uses concept of "trace" objects that incarnate a transformation instance
- Provides explicit control over the entire transformation process
  - E.g., can specify sub-transformations that can be executed in parallel
- Extends OCL with imperative statements and side-effects
  - E.g.: assignment statement, imperative "if-then-else", loops, etc.
  - Used to specify transformation procedure
  - Includes a "standard library" of OCL operations

© Copyright Malina Software

Operational Mapping: Basics

- **Some conceptual overlap with Relations**
  - Source is always "checkonly" and target is always "enforced"
  - Transformations are reified as objects
  - Each "transformation instance" (trace) ensures continuous updating of the target model in the presence of ongoing modifications of the source model
- **General format:**
  - transformation <name> (in im:MM1, out om:MM2)
    main () {-- <imperative transformation description> }
- **A transformation occurs by creating an instance of the appropriate transformation and invoking its "transform()" operation**
  - The results can be checked for success or failure
  - Exceptions can be handled explicitly, etc.

© Copyright Malina Software

Operational Mapping: Mapping Specification

- **Each mapping is defined as an operation on the appropriate metamodel element**
  - E.g. (see slide xxx):
  - ```
     mapping AssociationClass::UMLAssocClass2MOF(){
    // this will create an end object if it does not exist
    object ca : MOFClass
          {name := self.end[0].umlclass.name;};
    object cb : MOFClass
          {name := self.end[1].umlclass.name;};
    object a1e1 : MOFProperty
          {name := self.end[0].name;
           lower := self.end[0].lower;
           upper := self.end[0].upper;}
    object a1e2 : MOFProperty
          {name := self.end[1].name;
           lower := self.end[1].lower;
           upper := self.end[1].upper;}
    ...
    object ac : MOFAssociation
          {name := ca.name + '_' + cb.name; }
    }
    ```


© Copyright Malina Software
    ```

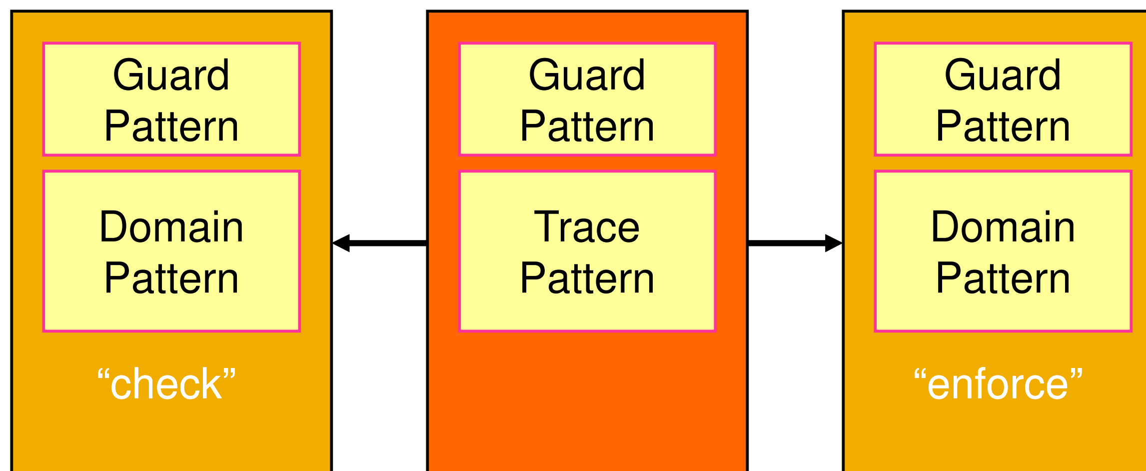
Operational Mapping: Invoking the Mappings

```
transformation UML2MOF (in um:UML, out mm:MOF)
main (){
im.objectsOfType(AssociationClass)->
     map UMLAssocClass2MOF ()}
mapping AssociationClass::UMLAssocClass2MOF() {
     ...}
```


© Copyright Malina Software

Alternative: The Core Language Approach

- **Similar to Relations but simpler**
  - Trace classes need to be specified explicitly
  - Mappings tend to be more verbose




```
map <mappingName> {
      [check] [enforce] <metamodelName> (<guardPattern>) {<domainPattern>}
      [check] [enforce] <metamodelName> (<guardPattern>) {<domainPattern>}
      ...
      where (<guardPattern>) {<tracePattern>} }
```


© Copyright Malina Software

Summary: MDA Transformations

- A key element of MDA
- An operation on MDA models to
  - Convert models into equivalent models suitable for analysis or viewing
  - Refine or generalize models (e.g., PIM to PSM, or PSM to PIM)
  - Generate code from models
- OMG provides a technology-neutral standard for defining transformations
  - Declarative style (Core and Relations)
  - Imperative style (Operational Mappings)
- Work on model transformations is in its infancy and more research is required to deal with scalability, performance, optimization, etc.

© Copyright Malina Software

Summary (1)

- **The definition of a modeling languages comprises a concrete syntax, an abstract syntax, and semantics**
  - Greater emphasis on communication/understanding aspects compared to most programming languages
    - E.g., multiple DSMLs, each chosen for its expressiveness
- **We have neither a deep understanding nor a systematic approach to modeling language design**
  - A discipline lacking theoretical underpinnings $\Rightarrow$ but definitely not lacking in controversy
  - But, a critical discipline to help us contend with the growing complexity of modern software

© Copyright Malina Software

Summary (2)

- **Designing a useful domain-specific computer language (modeling or programming) is challenging and requires diverse and highly-specialized skills**
  - Domain expertise
  - Modeling language design expertise
    - No established and reliable theory of modeling language design to guide the designer
  - Dealing with the fragmentation problem
- **And remember: <u>if the support infrastructure is inadequate, the language may not be viable</u>**
  - Despite its potential technical excellence

© Copyright Malina Software

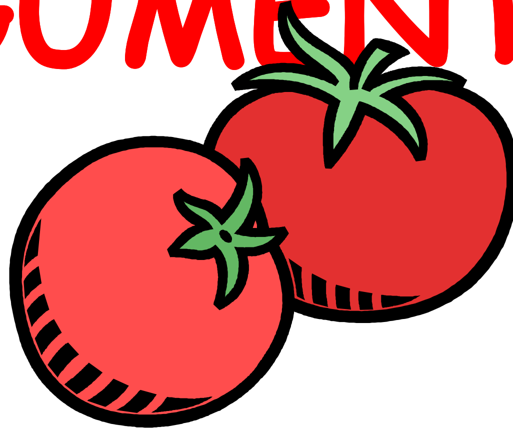
Bibliography/References

- A. Kleppe, “Software Language Engineering”, Addison-Wesley, 2009
- Kelly, S. and Tolvanen, J-P., "Domain-Specific Modeling: Enabling Full Code Generation," John Wiley & Sons, 2008
- J. Greenfield et al., “Software Factories”, John Wiley & Sons, 2004
- Kermeta Workbench (http://www.kermeta.org/)
- OMG’s Executable UML Foundation Spec (http://www.omg.org/spec/FUML/1.0/Beta1)
- UML 2 Semantics project (http://www.cs.queensu.ca/~stl/internal/uml2/index.html)
- ITU-T SDL language standard (Z.100) (http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf)
- ITU-T UML Profile for SDL (Z.109) (http://www.itu.int/md/T05-SG17-060419-TD-WP3-3171/en)

© Copyright Malina Software

- THANK YOU -

QUESTIONS, COMMENTS, ARGUMENTS...



© Copyright Malina Software