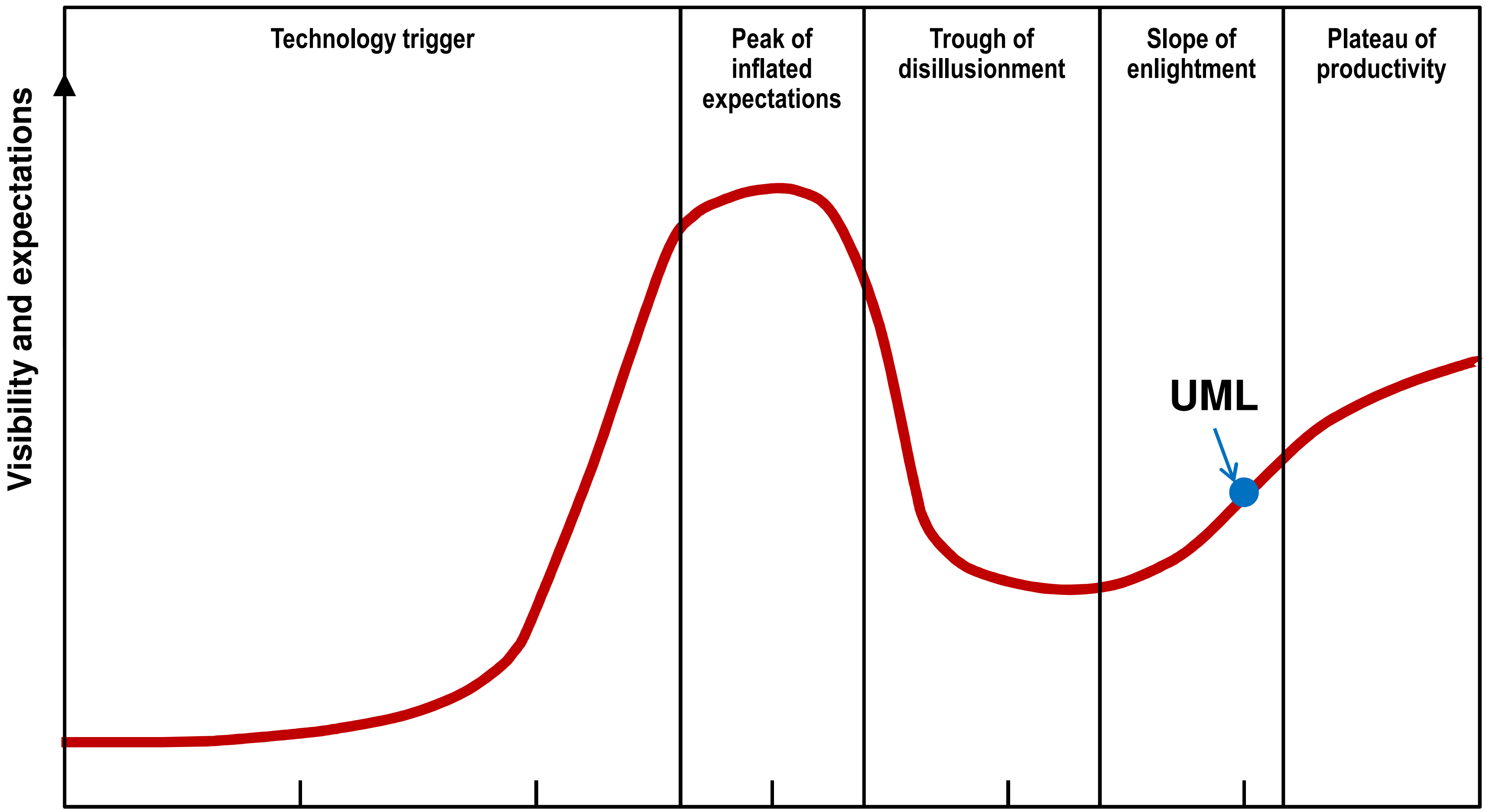
Elements of Model-Based Engineering with UML 2:
What They Don't Teach You About UML

Bran Selić

Malina Software Corp., Canada
Simula Research Labs, Norway
Zeligsoft (2009) Ltd., Canada
University of Toronto, Canada
Carleton University, Canada
University of Sydney, Australia

selic@acm.org

The Software Technology Hype Cycle

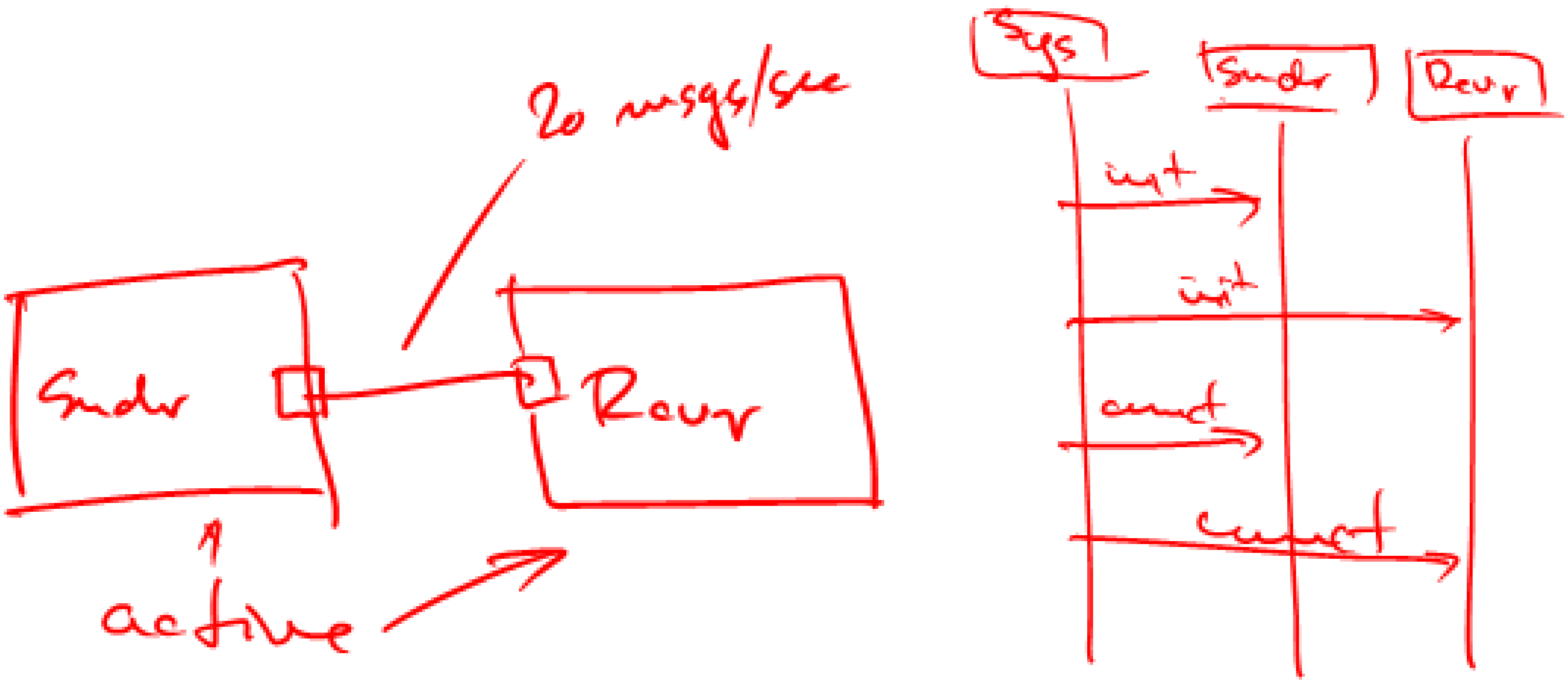


Lecture Overview

- About Model-Based Engineering (MBE)
- A Short Primer on Modeling Language Design
- The Unified Modeling Language
  - Semantics
  - UML as a DSL tool
- UML as an architectural description language

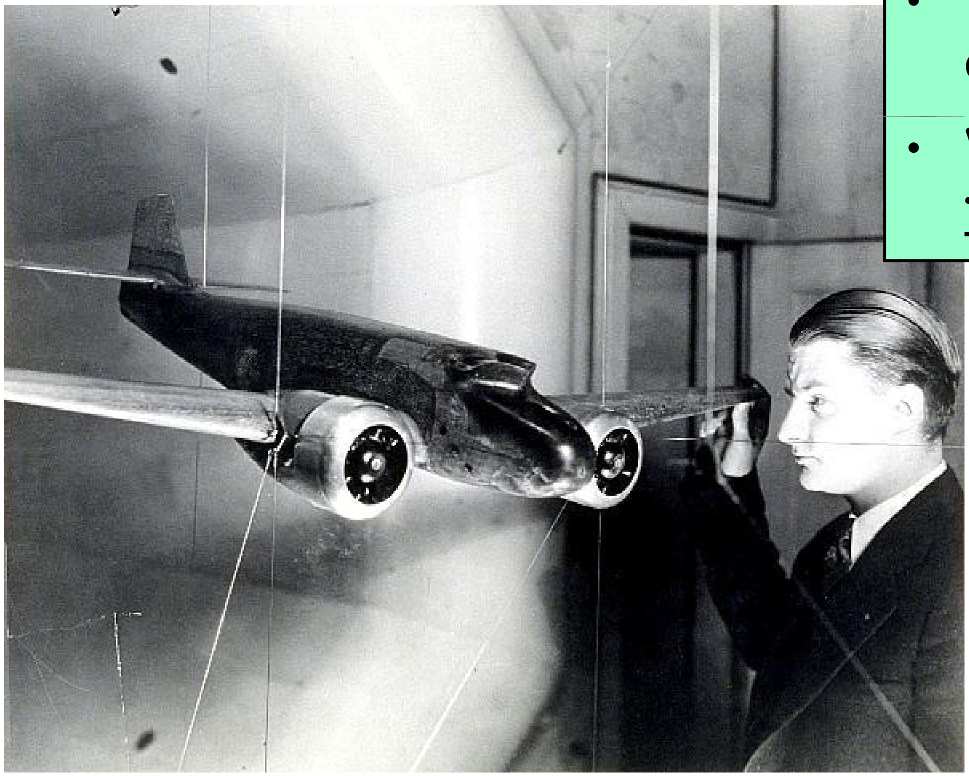
A Common Attitude to Modeling Software

- We don't do modeling here...it's a waste of time
- But...



Engineering Models

- **ENGINEERING MODEL:** *A selective representation of some system that specifies, 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

Why Do Engineers Build Models?

- To understand
  - ...problems and solutions
  - Knowledge acquisition
- To communicate
  - ...understanding and design intent
  - Knowledge transfer
- To predict
  - ...the interesting characteristics of system under study
  - Models as surrogates
- To specify
  - ...the implementation of the system
  - Models as "blueprints"

Types of Engineering Models

- **Descriptive**: models for understanding, communicating, and predicting
  - E.g., scale models, mathematical models, qualitative models, documents, etc.
  - Tend to be highly abstract (detail removed)
- **Prescriptive**: models as specifications
  - E.g., architectural blueprints, circuit schematics, state machines, pseudocode, etc.
  - Tend to be sufficienbtly detailed so that the intended system can be implemented

What about applying modeling to software?

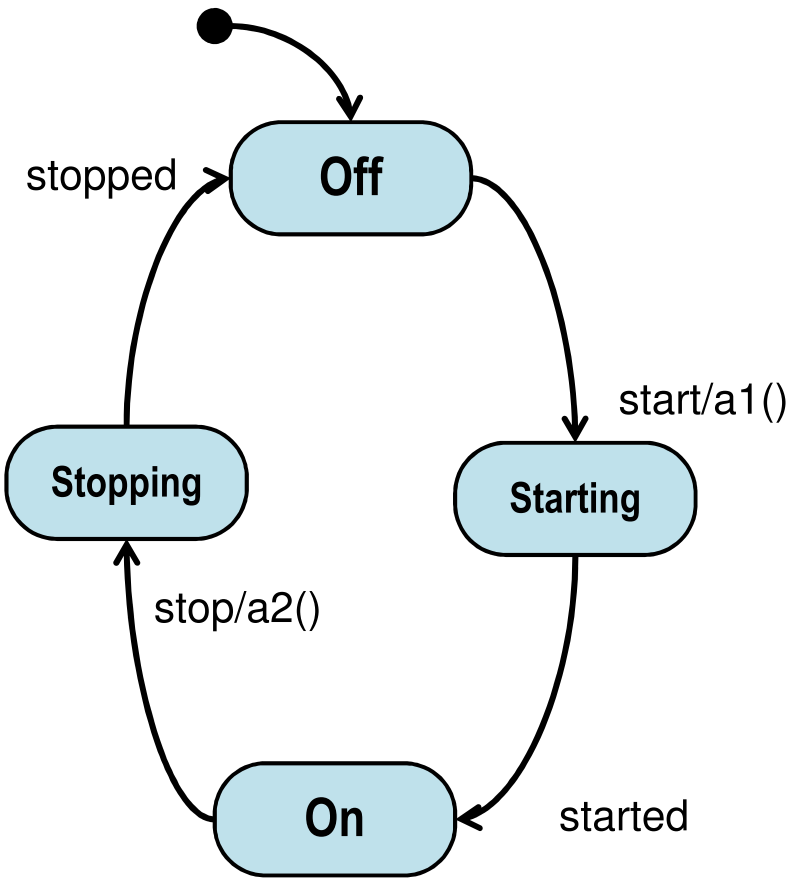
Footnote: On the Use of Graphics

- "Whenever someone draws a picture to explain a program, it is a sign that something is not understood." – E. Dijkstra*
- "Yes, a picture is what you draw when you are trying to understand something or trying to help someone understand." – W. Bartussek*

* Quoted in D.L. Parnas, "Precisely Annotated Hierarchical Pictures of Programs", McMaster U. Tech Report, 1998.

Footnote: On the Effectiveness of Graphics

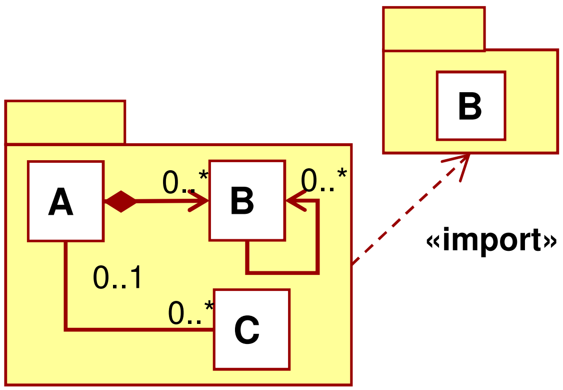
```
State: Off, On, Starting, Stopping;
Initial: Off;
Transition:
    {source: Off;
     target: Starting;
     trigger: start;
     action: a1();}
Transition:
    {source: Starting;
     target: On;
     trigger: started;}
Transition:
    {source: On:
     target: Stopping;
     trigger: stop;
     action: a2();}
Transition:
    {source: Stopping;
     target: Off;
     trigger: stopped;}
```



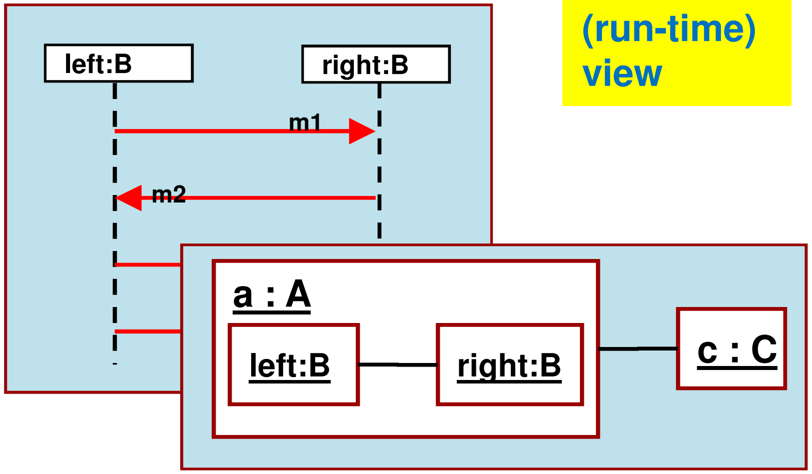
Modeling Software

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

Repository view (structure of the software specification)



Execution (run-time) view

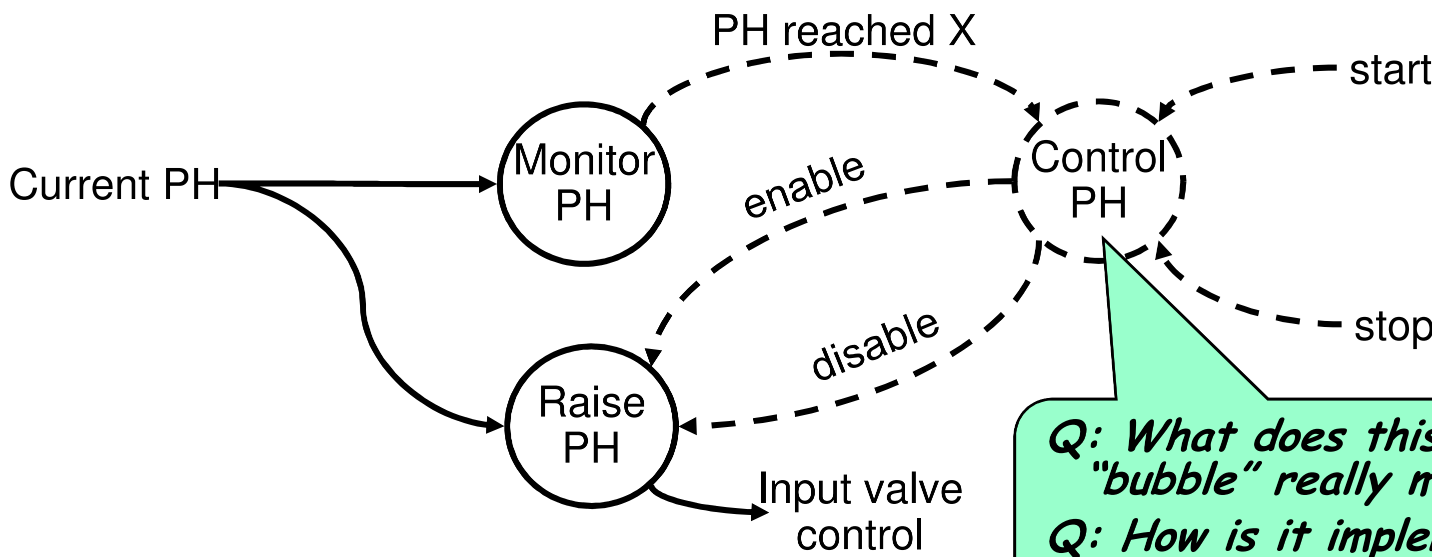


Example of the descriptive role of modeling languages (not generally present in programming languages)

What's a Modeling Language?

- **MODELING LANGUAGE:** A computer language intended for constructing models of systems and the contexts in which these systems operate
- Examples:
  - AADL, Matlab/Simulink, Modelica, SDL, SysML, UML, etc.

Early SW Modeling: SA/SD



Q: What does this "bubble" really mean?

Q: How is it implemented in code?

"…bubbles and arrows, as opposed to programs, …never crash"

-- B. Meyer
"UML: The Positive Spin"
American Programmer, 1997

Modeling languages have yet to recover from this "debacle"

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

"Classical" Software Modeling Languages

- **Flow charts, SA/SD, 90's OO notations (Booch, OMT, OOSE, UML 1)**
- **Most of them were designed almost exclusively for constructing descriptive models**
  - Informal "sketching" [M. Fowler]*

  ⇒ No perceived need for high-degrees of precision

  ⇒ Languages are ambiguous and open to interpretation ⇒ source of undetected miscommunication

***http://martinfowler.com/bliki/UmlAsSketch.html**

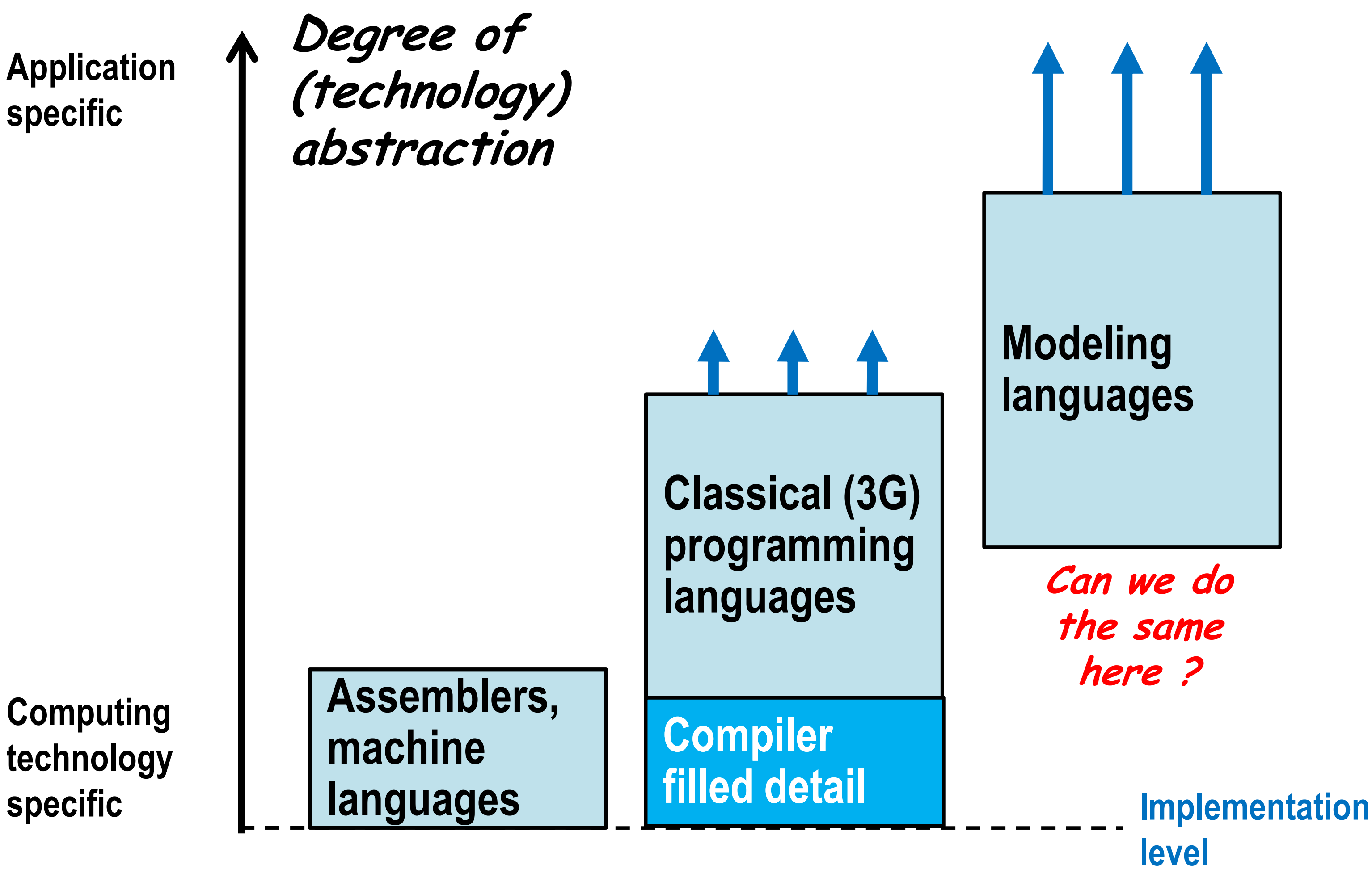
New Generation of Modeling Languages

- Formal languages designed for modeling
  - ⇒ Support for both descriptive and prescriptive models
  - ...sometimes in the same language
- Key objectives:
  - Well-understood and precise semantic foundations
  - Can be formally (i.e., mathematically) analyzed (qualitative and quantitative analyses)
  - And yet, can still be used informally ("sketching") if desired

Modeling vs. Programming Languages

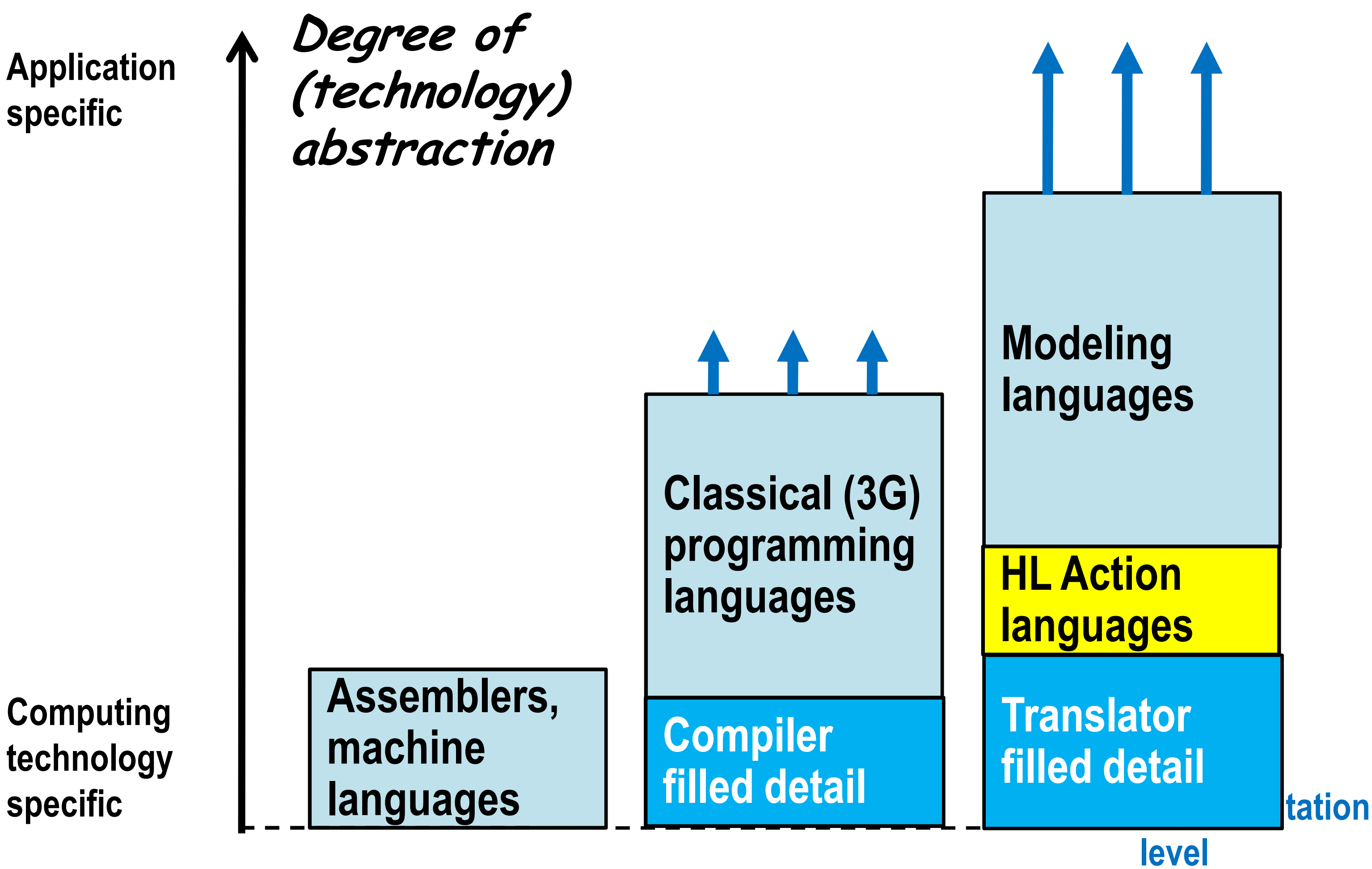
- **The primary purpose and focus of programming languages is <u>implementation</u>**
  - The ultimate form of <u>prescription</u>
  - ⇒ Implementation requires total precision and "full" detail
  - ⇒ Takes precedence over description requirements
- **A modeling language <u>must</u> support <u>description</u>**
  - I.e., <u>communication</u>, <u>prediction</u>, and <u>understanding</u>
  - These generally require omission of "irrelevant" detail such as details of the underlying computing technology used to implement the software

The Evolution of Computer Languages



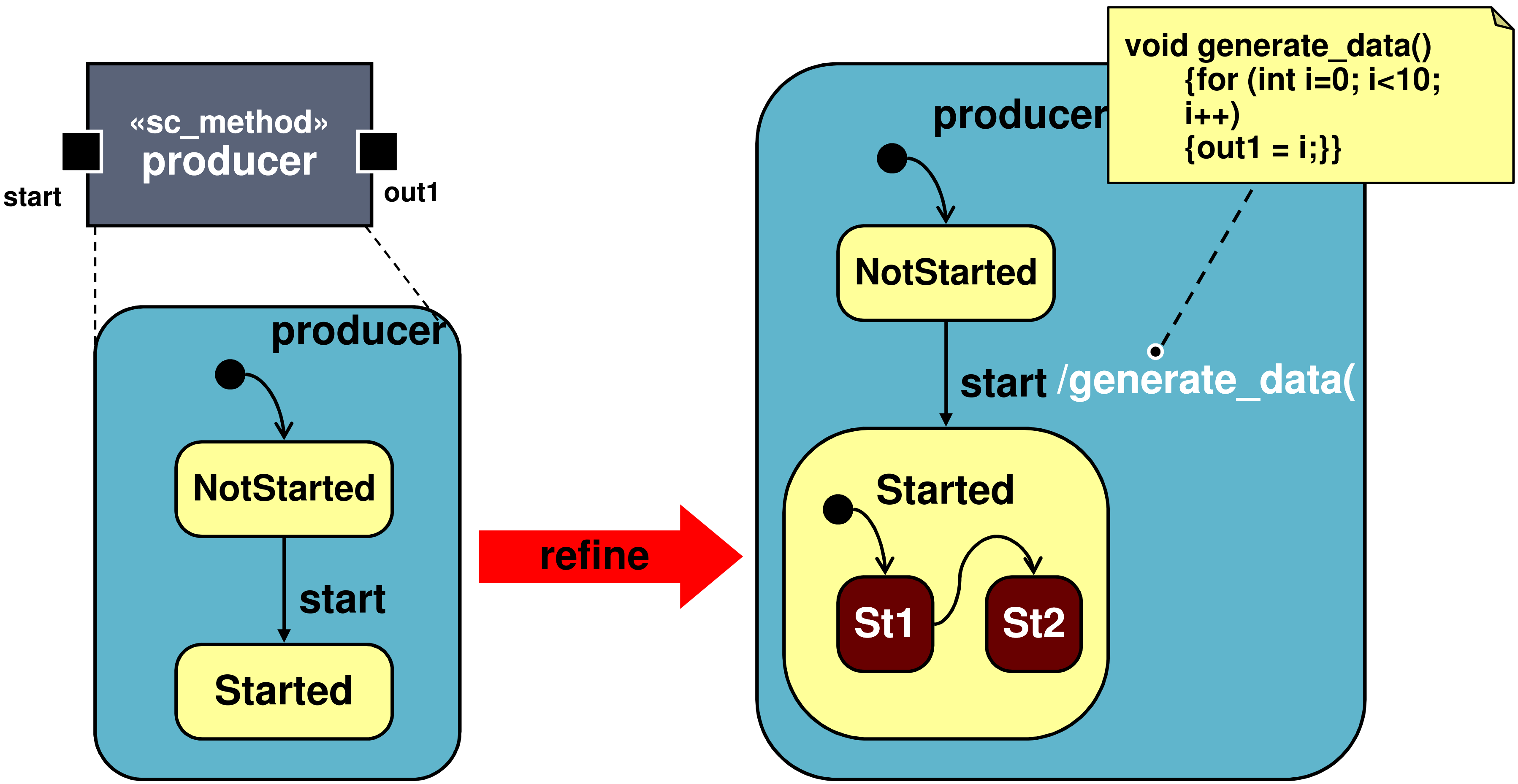
Filling the Gap

- Combination of approaches



Modern MBE Development Style

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



A Unique Feature of Software

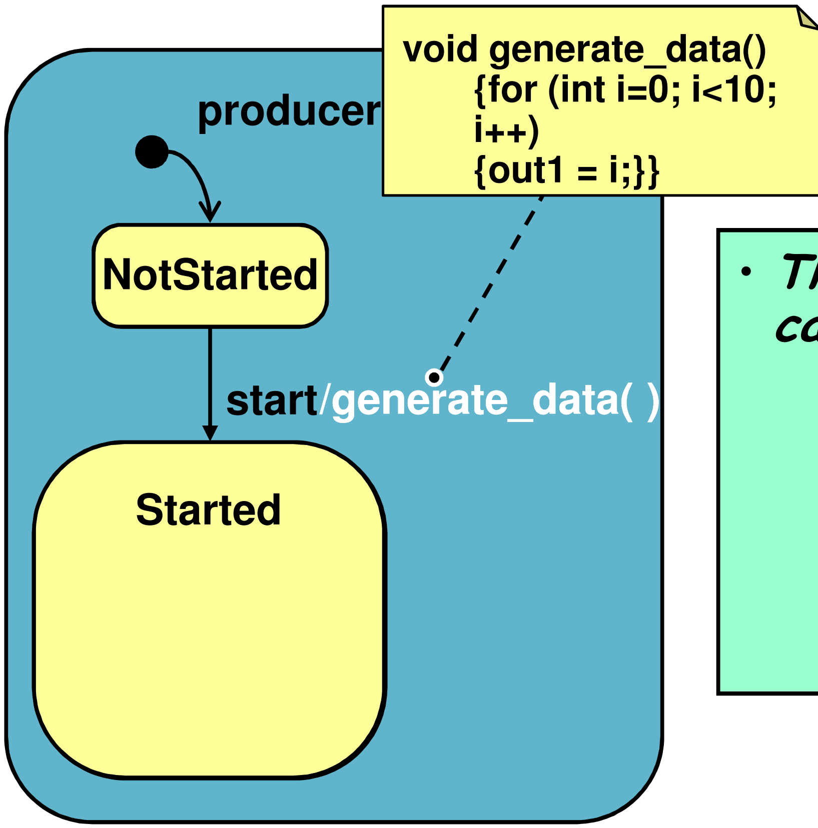
- A software model and the software being modeled share the same medium—the computer
  - Which also happens to be our most advanced and most versatile automation technology

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

But, if the Model is the System…

- …do we not lose the abstraction value of models?



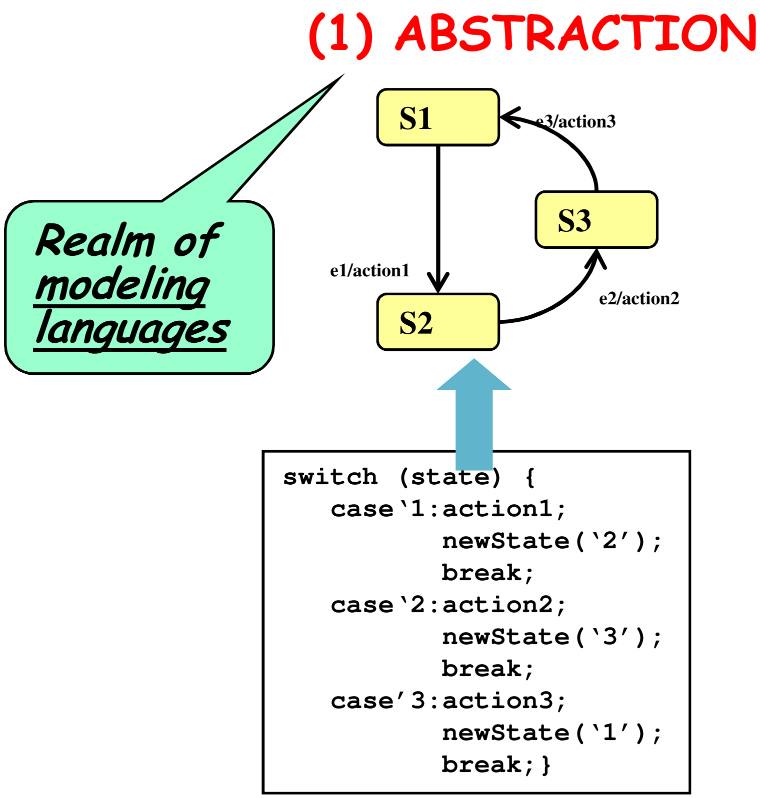
- *The computer offers a uniquely capable abstraction device:*

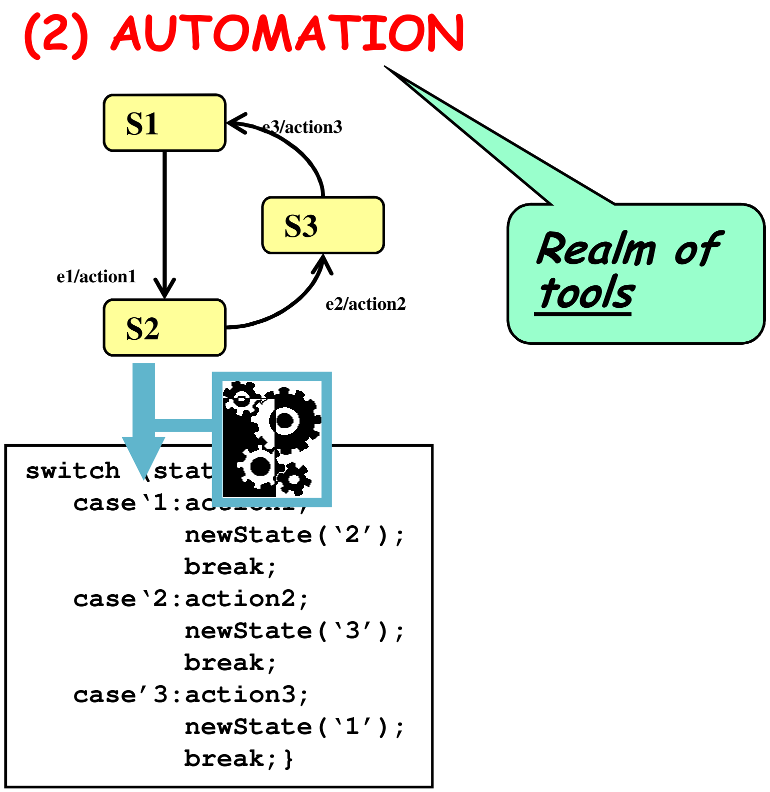
  Software can be represented from any desired viewpoint at any desired level of abstraction

  The abstraction is inside the system and can be extracted automatically

Summary: What MBE IS

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





Addendum: What MBE is NOT

- **It is not a paradigm shift**
  - Uses existing computing/programming paradigms
  - ...but, at higher levels of abstraction
- **It is not about graphical vs. textual syntax**
  - ...but syntax becomes a more significant factor
  - Whatever is appropriate to support understanding and communication
- ***(WARNING: Contentious statement!)*** **It is not about PIM vs. PSM**
  - "Platform" is a relative concept and independence is a matter of degree
  - Not a helpful distinction, but can be dangerously misinterpreted

Lecture Overview

- About Model-Based Engineering (MBE)
- A Short Primer on Modeling Language Design
- The Unified Modeling Language
  - Semantics
  - UML as a DSL tool
- UML as an architectural description language

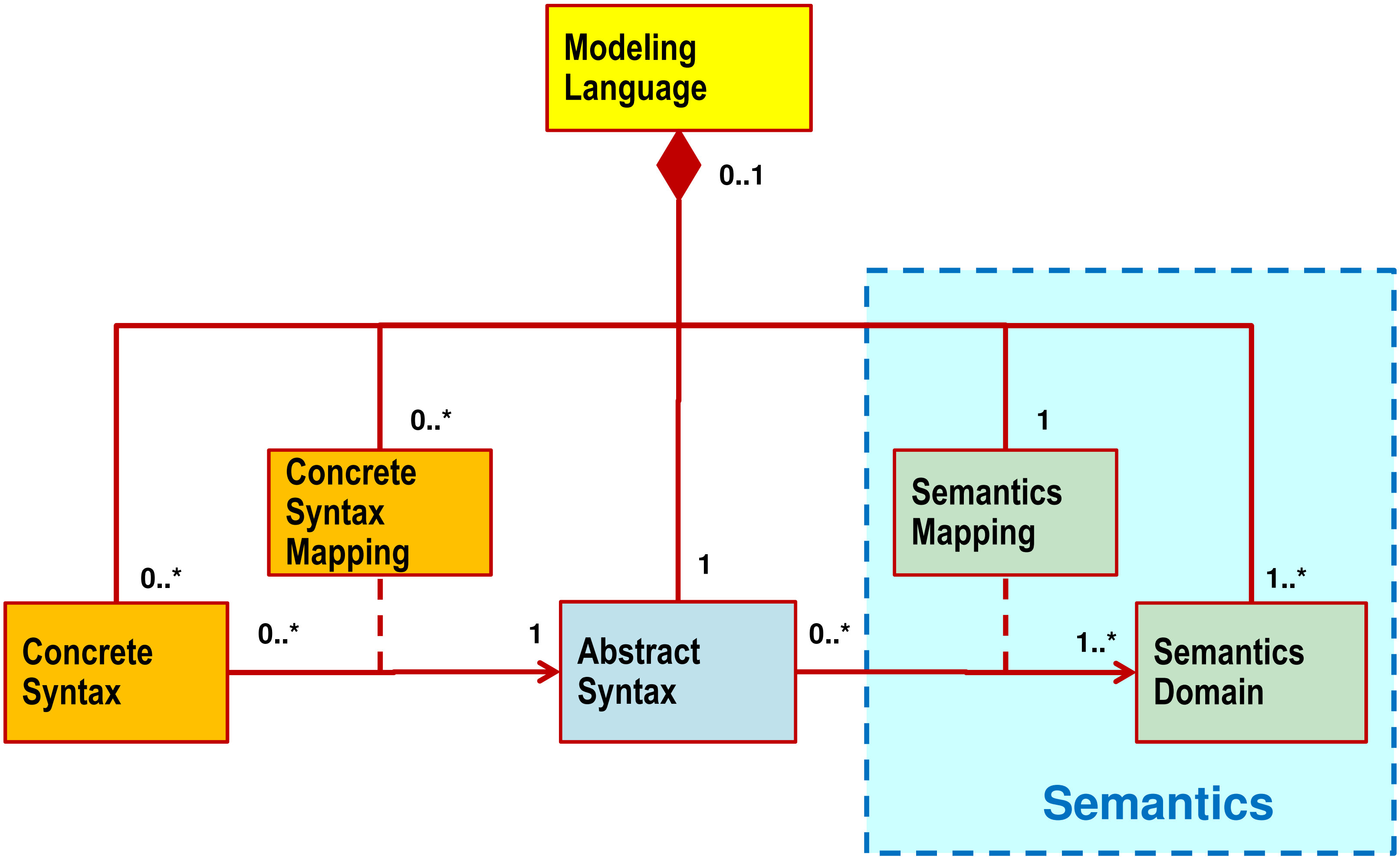
Components of a Modeling Language

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

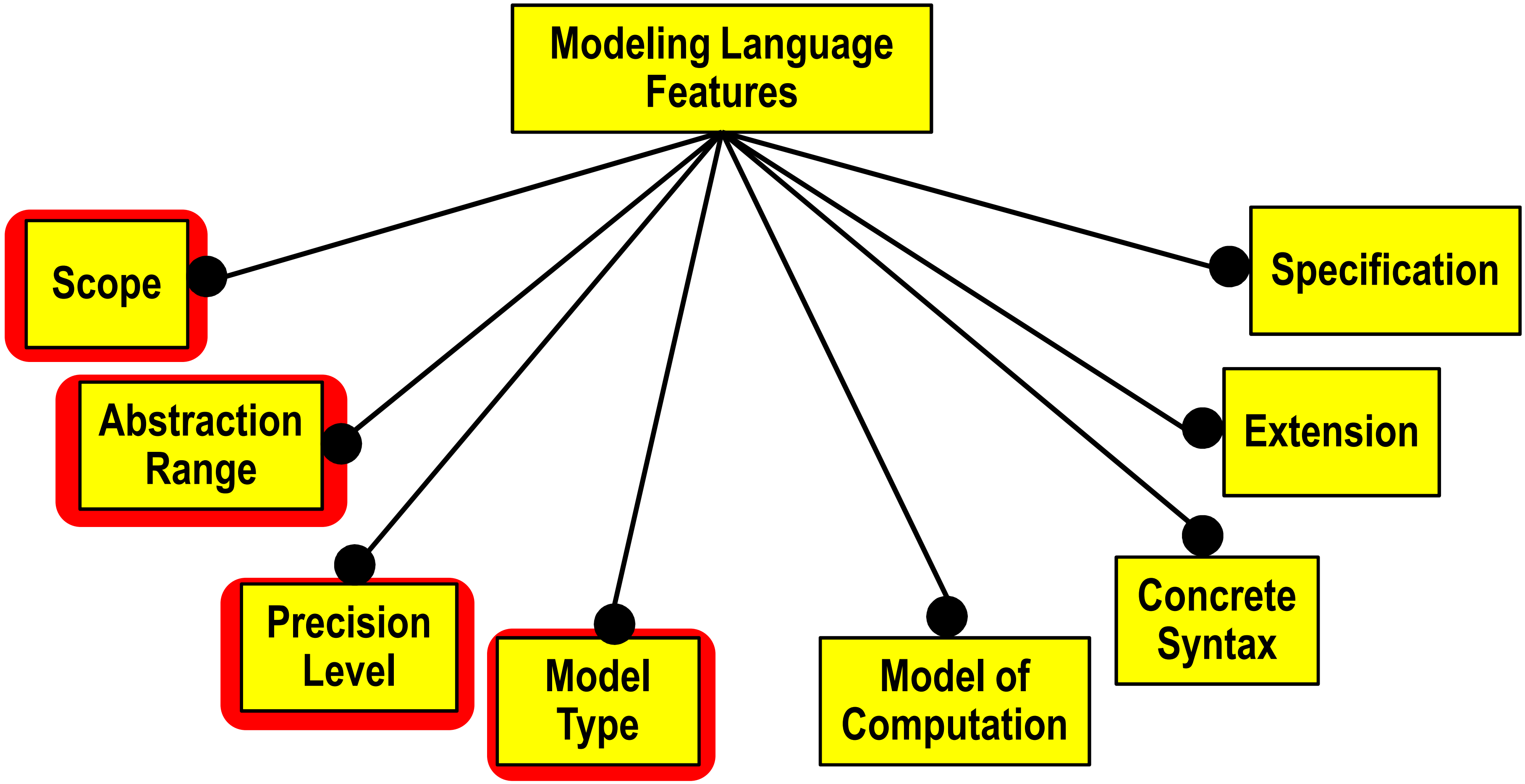
  ABSTRACT SYNTAX

  - CONCRETE SYNTAX (notation/representation)
    - e.g., keywords, graphical symbols for concepts
    - Mapping to abstract syntax concepts
  - SEMANTICS: the *meaning* of the language concepts
    - Comprises: Semantic Domain and Semantic Mapping (concepts to domain)

Elements of a Modeling Language



Key Modeling Language Design Issues



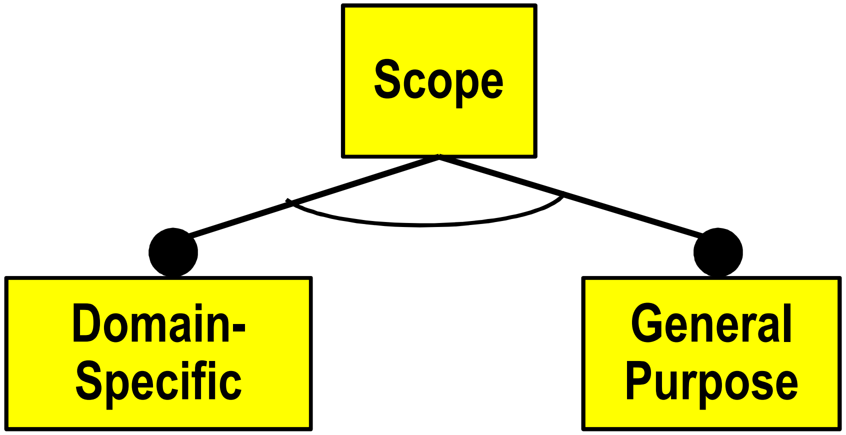
Some choices are inter-dependent

Selecting Language Scope

- A common opinion:

  "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 well-suited to any domain?"

- Which suggests:



But, this may be an oversimplification

Scope: How General/Specific?

- **Generality often comes at the expense of expressiveness**
  - <u>Expressiveness</u>: 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

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...**

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?**

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

Success Criteria for a Language (2)

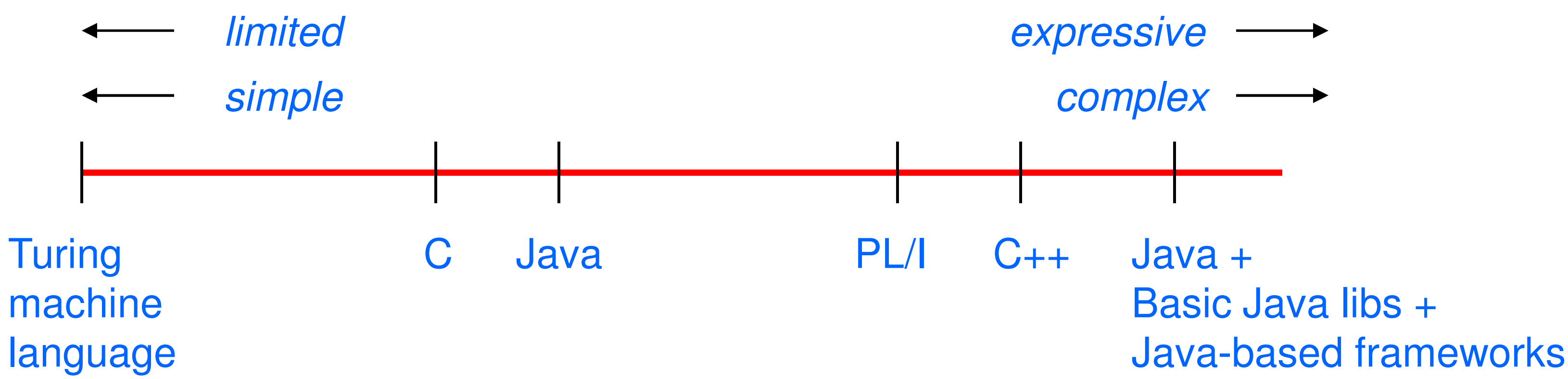
- **Language Support & Infrastructure:**
  - Availability of necessary **tooling**
  - Effectiveness of tools (reliability, quality, usability, customizability, interworking ability)
  - Availability of skilled practitioners
  - Availability of teaching material and training courses
  - Availability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

Sidebar: Basic Tooling Capabilities

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

Language Complexity

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



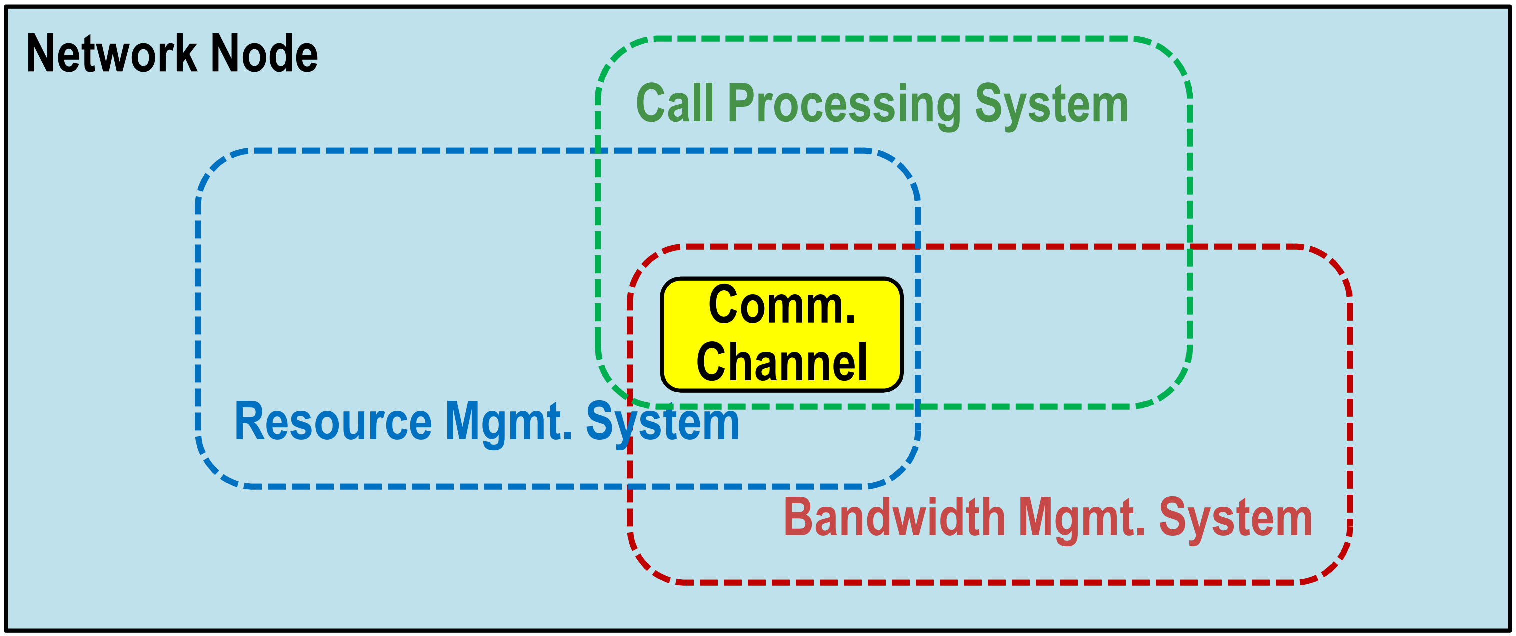
- **The art of computer language design lies in finding the right balance between expressiveness and simplicity**
  - Need to minimize accidental complexity while recognizing and respecting essential complexity
  - Small languages solve small problems
  - No successful language has ever gotten smaller

Practical Issues of Scope

- **Practical systems often involve multiple heterogeneous domains**
  - Each with its own ontology and semantic and dedicated specialists
- **Example: a telecom network node system**
  - Basic bandwidth management
  - Equipment and resource management
  - Routing
  - Operations, administration, and systems management
  - Accounting (customer resource usage)
  - Computing platform (OS, supporting services)

The Fragmentation Problem

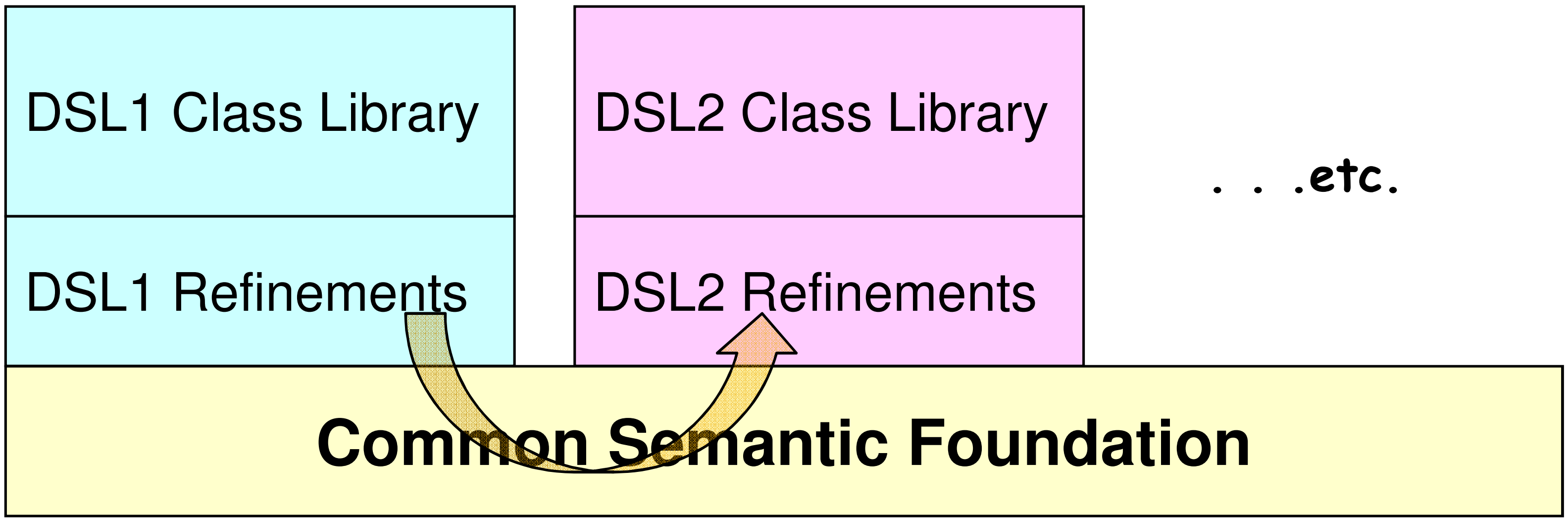
- **FRAGMENTATION PROBLEM:** combining overlapping independently specified domain-specific subsystems, specified using different DSLs, into a coherent and consistent whole (i.e., single implementation)



Sadly, there are no generic composition (weaving) rules – each case has to be handled individually

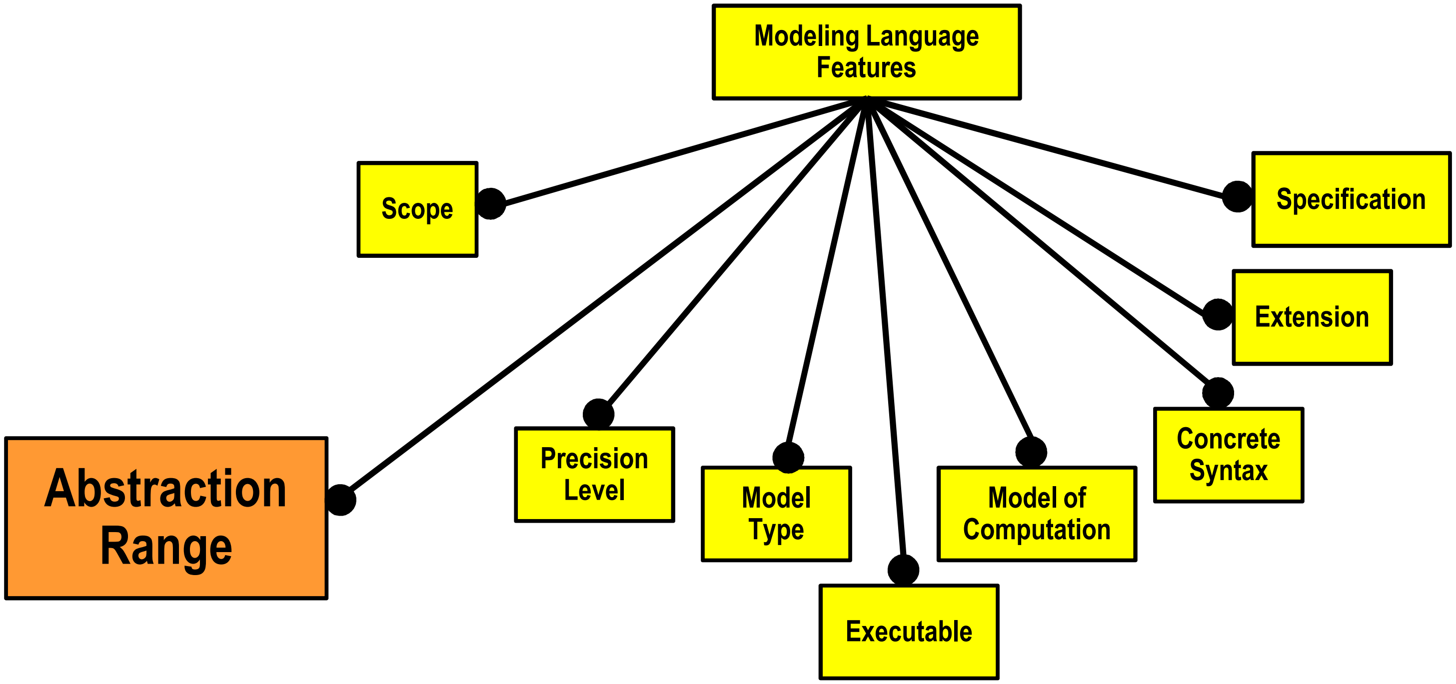
Approach to Dealing with Fragmentation

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



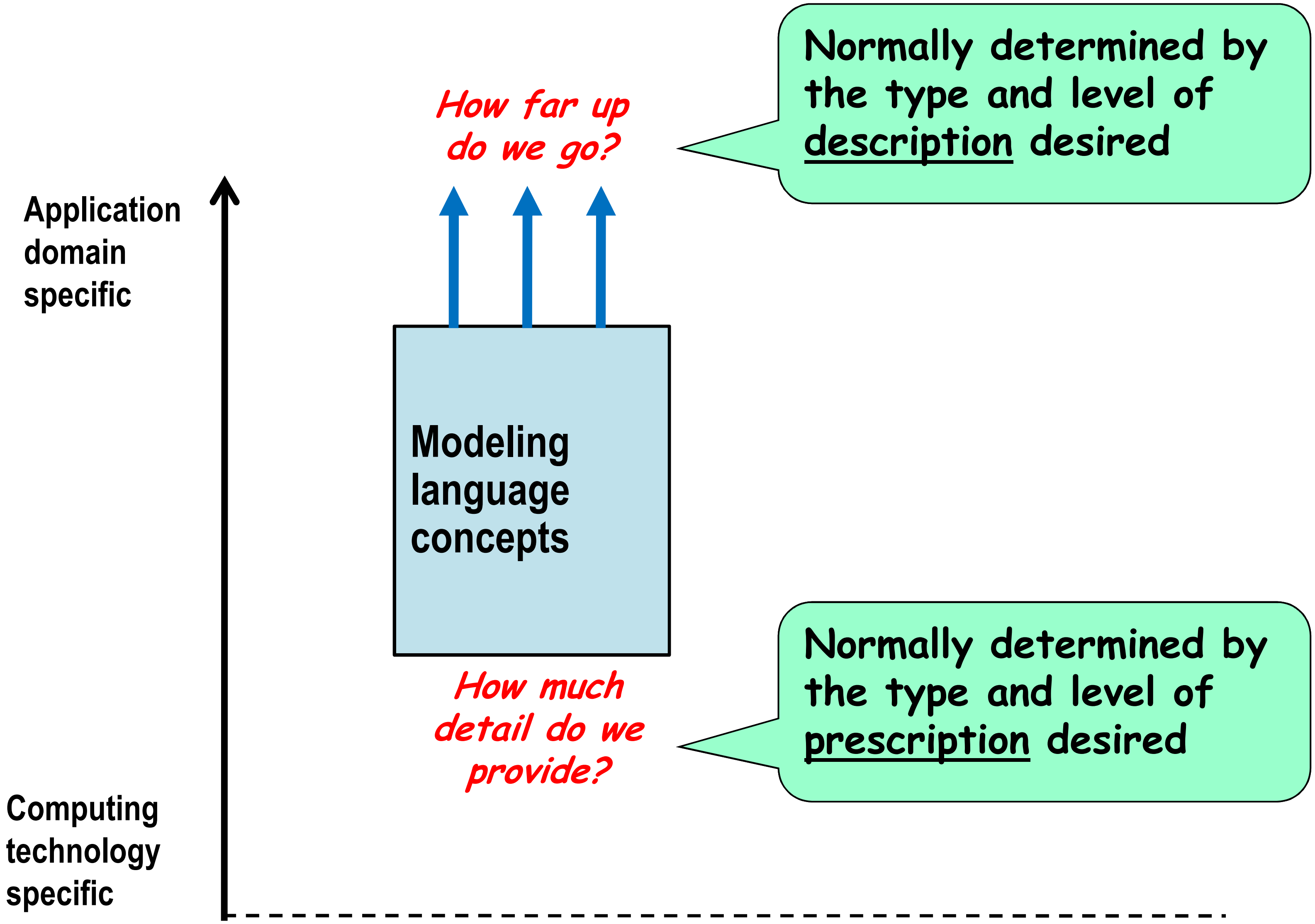
- NB: Same divide and conquer approach can be used to modularize complex languages
  - Core language base + independent sub-languages (e.g., UML)

Selecting An Abstraction Range

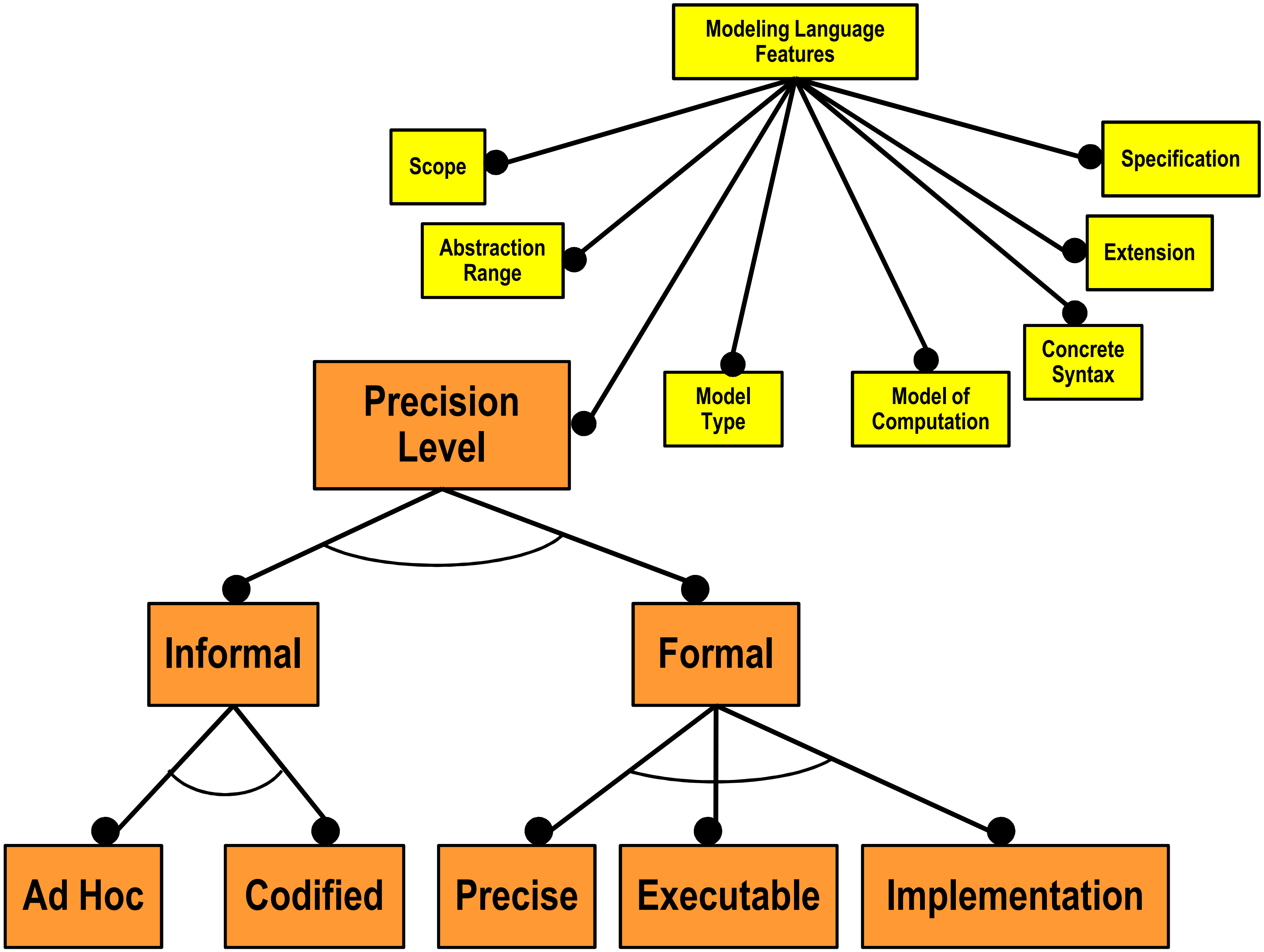


- This decomposes into two separate questions:
  - What is a suitable level of abstraction of the language?
  - How much (implementation-level) detail should the language concepts include?
- The answers depend on other design choices

Abstraction Range of Computer Languages



Selecting a Precision Level



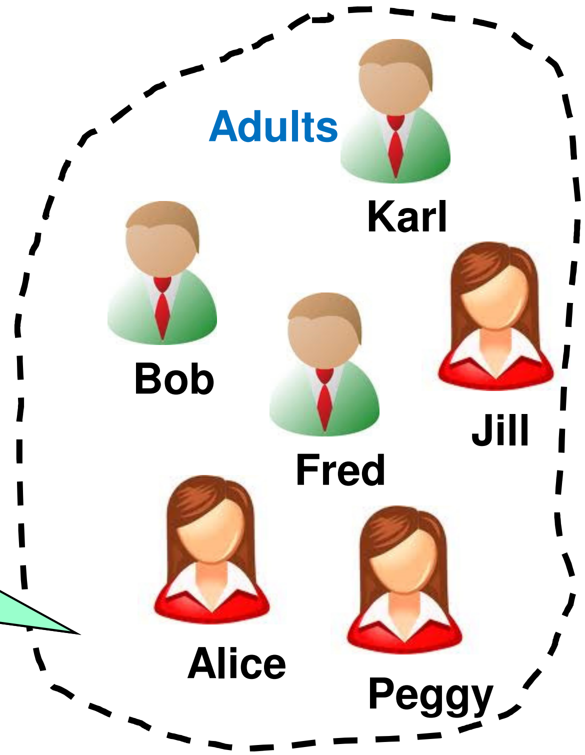
Formality

- **Based on a well understood mathematical theory with existing analysis tools**
  - E.g., automata theory, abstract state machines, Petri nets, temporal logic, process calculi, queueing theory, Horne clause logic
  - NB: precise does not necessarily 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

Precision vs. Detail

- **A specification can be precise but still leave out detail:**
  - E.g., we can identify a set very precisely without necessarily specifying the details associated with its members

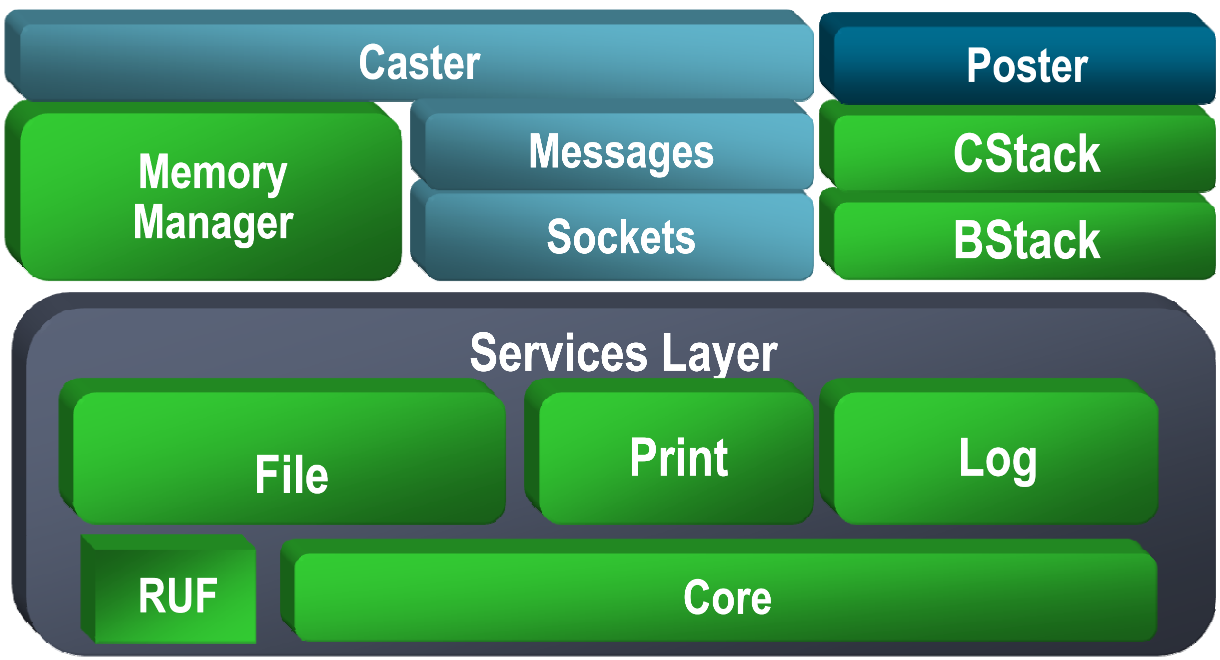
We state very precisely as to what constitutes <u>the set of Adults</u> of some population (age ≥ 21) without being specific about details such as names or genders of its members



back

Ad Hoc "Languages"

- Mostly notations created for specific cases (not intended for reuse)
- Used exclusively for descriptive purposes
- No systematic and comprehensive specification of syntax or semantics
  - *Appeal to intuition*



Codified Languages

- **Example: UML 1.x, OMT, SysML, ...**
- **Characteristics:**
  - Defined: An application-independent language specification exists
  - Some aspects of the language are fully defined (usually: concrete syntax, semantics)
  - Semantics usually based on natural language and other informal specification methods
  - Designed primarily for descriptive modeling
  - But, may also be used partly for specification (e.g., partial code generation/code skeletons)

Precise Languages

- Examples: Object Constraint Language (OCL), Layered Queueing Networks (LQN)
- Formally defined semantics (domain and mapping)
- High level of abstraction but typically cover relatively small range
  - I.e., lacking detail for execution or implementation
  - Often declarative
- Mostly designed for description (e.g., prediction and analysis), but may also be used for specification

On Specifying Semantics

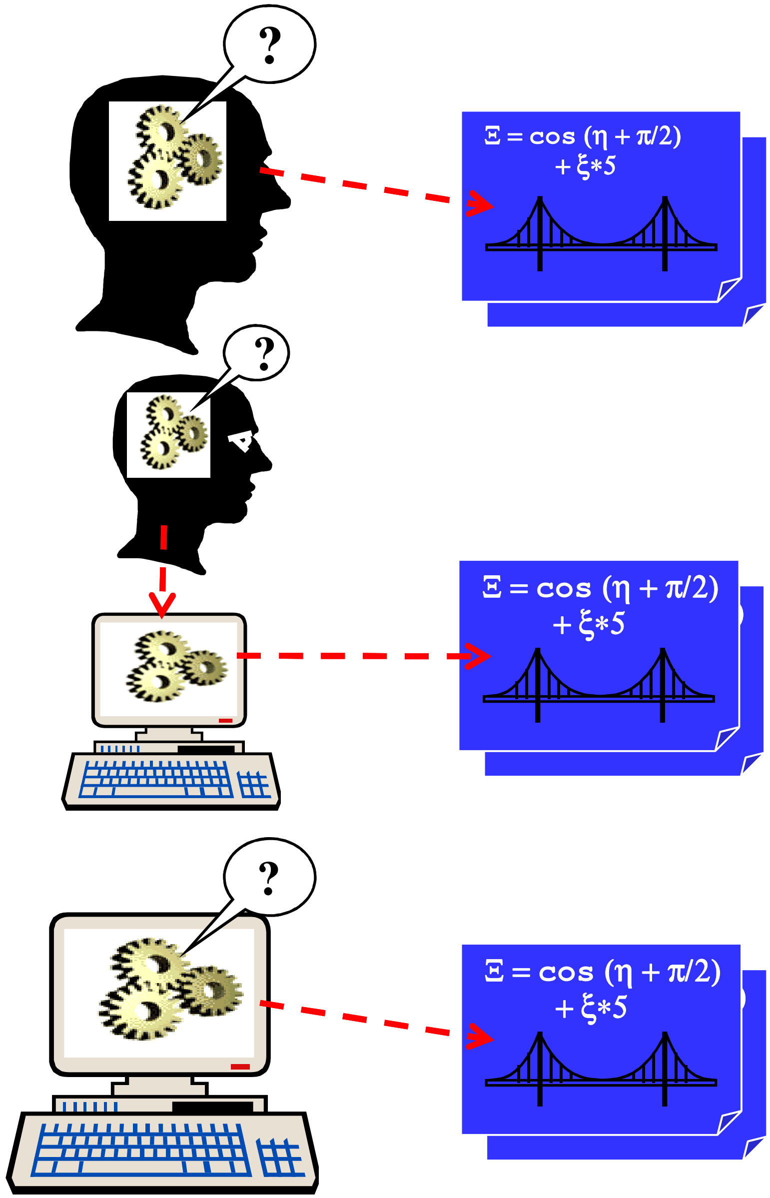
- Semantics are specified using a language whose semantics are already defined
- Numerous approaches to defining run-time semantics of computer languages
  - Informal natural language description are the most common way
  - Denotational, translational, operational, axiomatic, etc.

Executable Languages

- "Models that are not executable are like cars without engines", [D. Harel]
- Examples: Modelica, Matlab
- A subcategory of precise languages covering a range that includes sufficient detail for creating executable models
  - But, may be missing detail required for automatic generation of implementations
  - Often based on operational semantics that may not be easily analyzed by formal methods (due to scalability issues)
- Rationale:
  - Enables early detection of design flaws
  - Helps develop engineering intuition and confidence

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



Executable Modeling Tool Requirements

- **Ability to execute a model on a computer and observe its behavior**
  - With possible human intervention when necessary
- **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

Implementation (Modeling) Languages

- **Computer languages that:**
  - Provide concepts at high levels of abstraction suitable for descriptive purposes, and also
  - Include detailed-level concepts such that the models can provide efficient implementations through either automatic code generation or interpretation
- **Examples: UML-RT, Rhapsody UML, etc.**

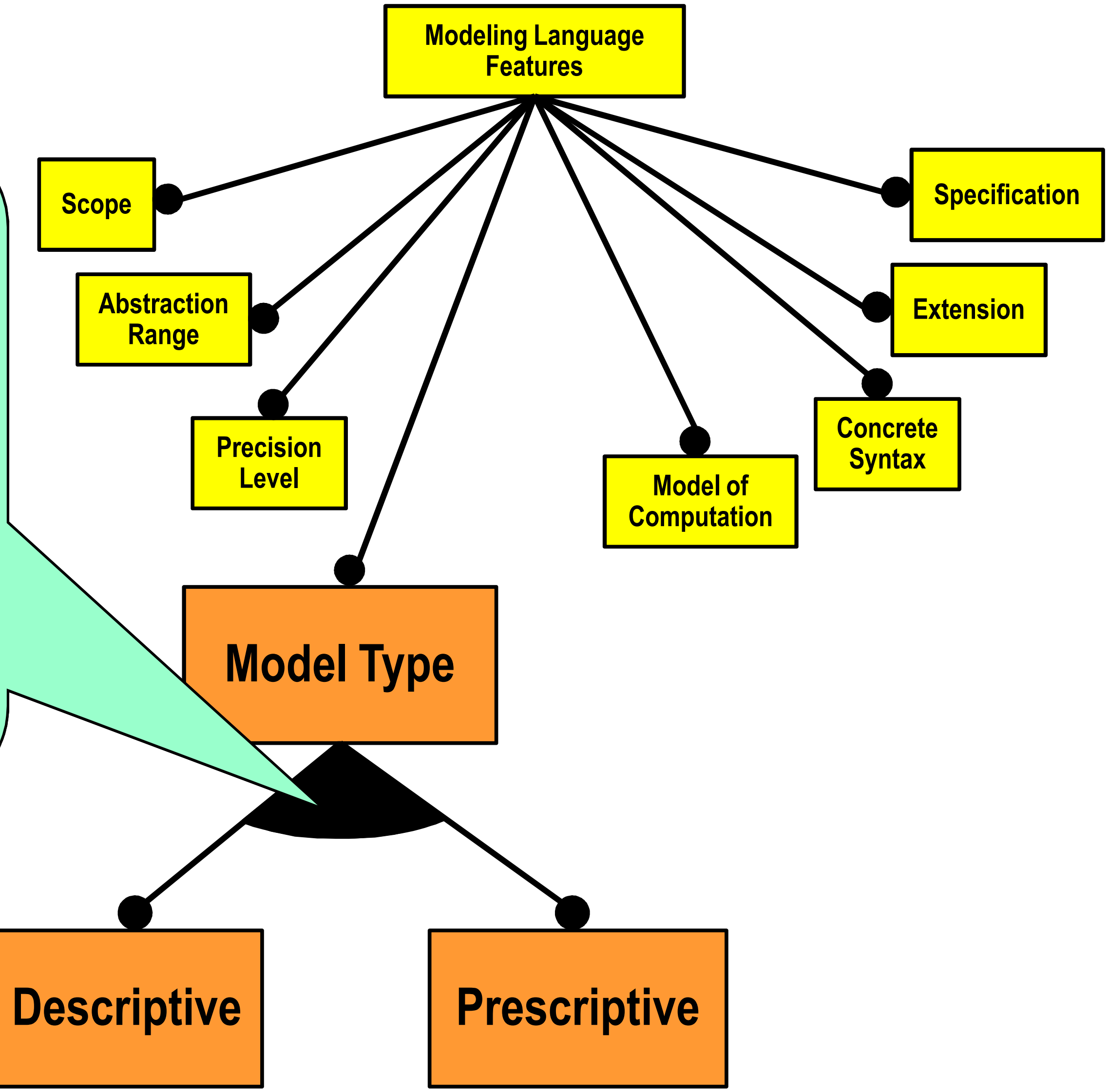
Precision Level Categories

- **A more refined categorization 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
PRECISE	Defined, formal, consistent	Analysis, Prediction
CODIFIED	Defined, informal	Documentation, Analysis
AD HOC	Undefined, informal	Documentation, Analysis (no reuse)

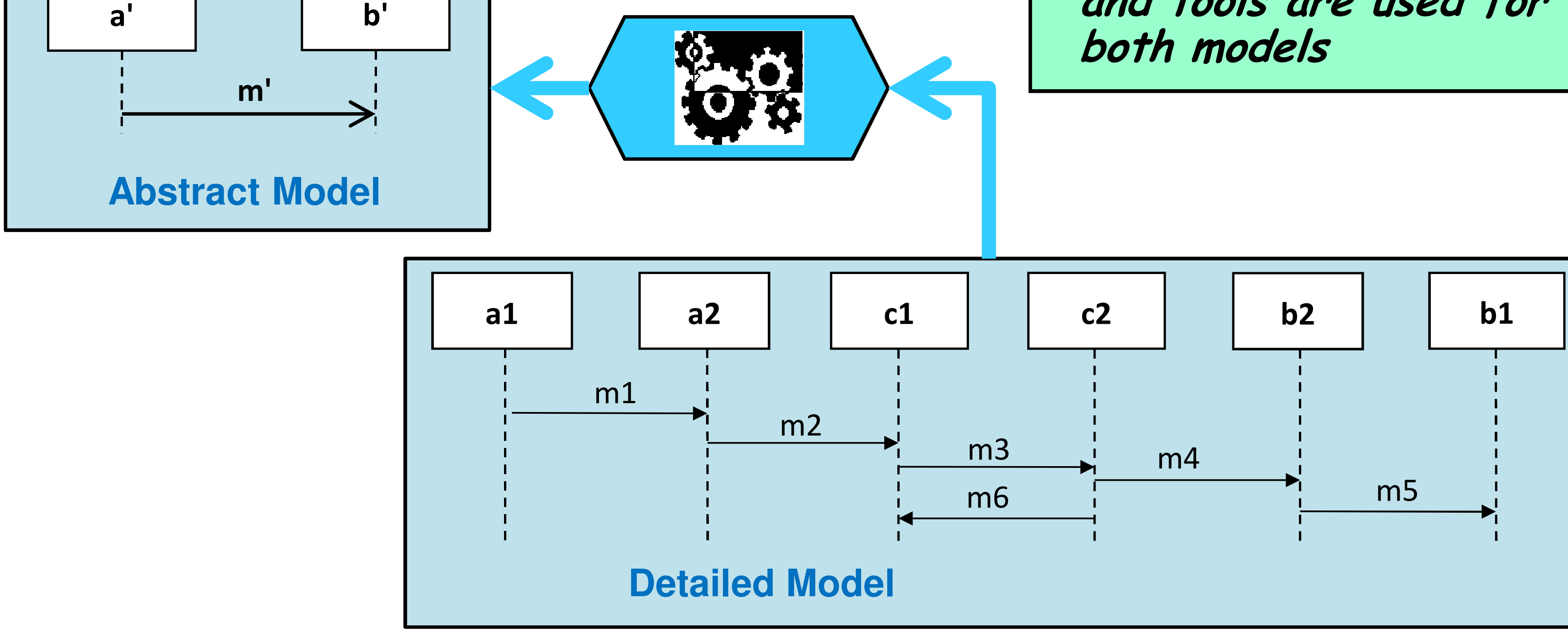
Selecting a Model Type

With the appropriate choice of Abstraction Range and Precision Level in combination with suitable model transforms, it is possible to define languages that support both types of models



Pragmatics: Multiple Models Needed

- In reality, it is generally <u>not practical to have a single model</u> that covers all possible levels of abstraction
- But, it is possible to formally (i.e., electronically) couple different models via <u>persistent model transforms</u>



NB: The same language and tools are used for both models

Summary

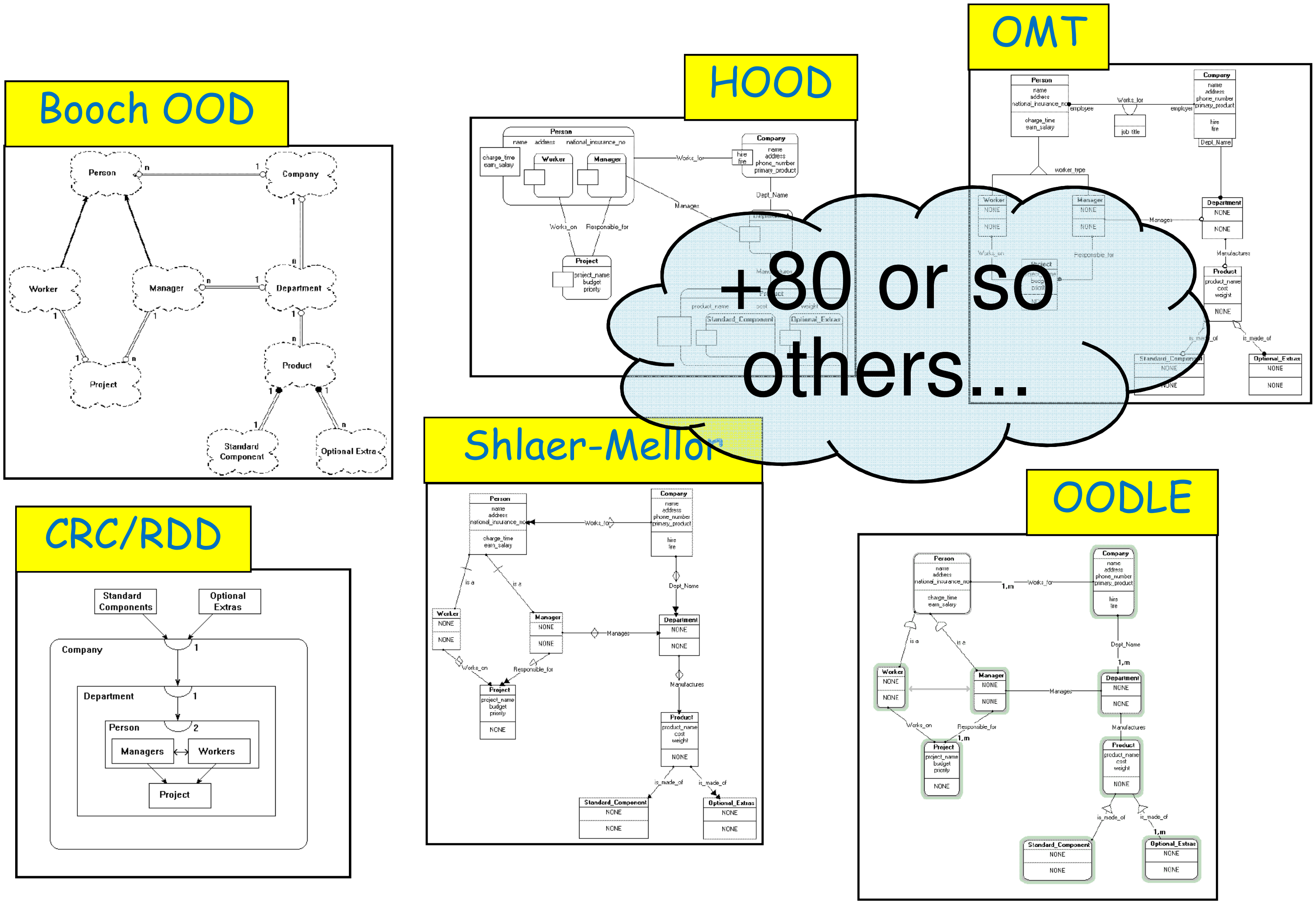
- **The more diverse nature of models (compared to programs) creates new challenges for language designers**
  - Support for descriptive models
  - Opportunity to combine descriptive and prescriptive applications
- **Our understanding of modeling language design is still evolving**
  - Need a sound theoretical underpinning

Lecture Overview

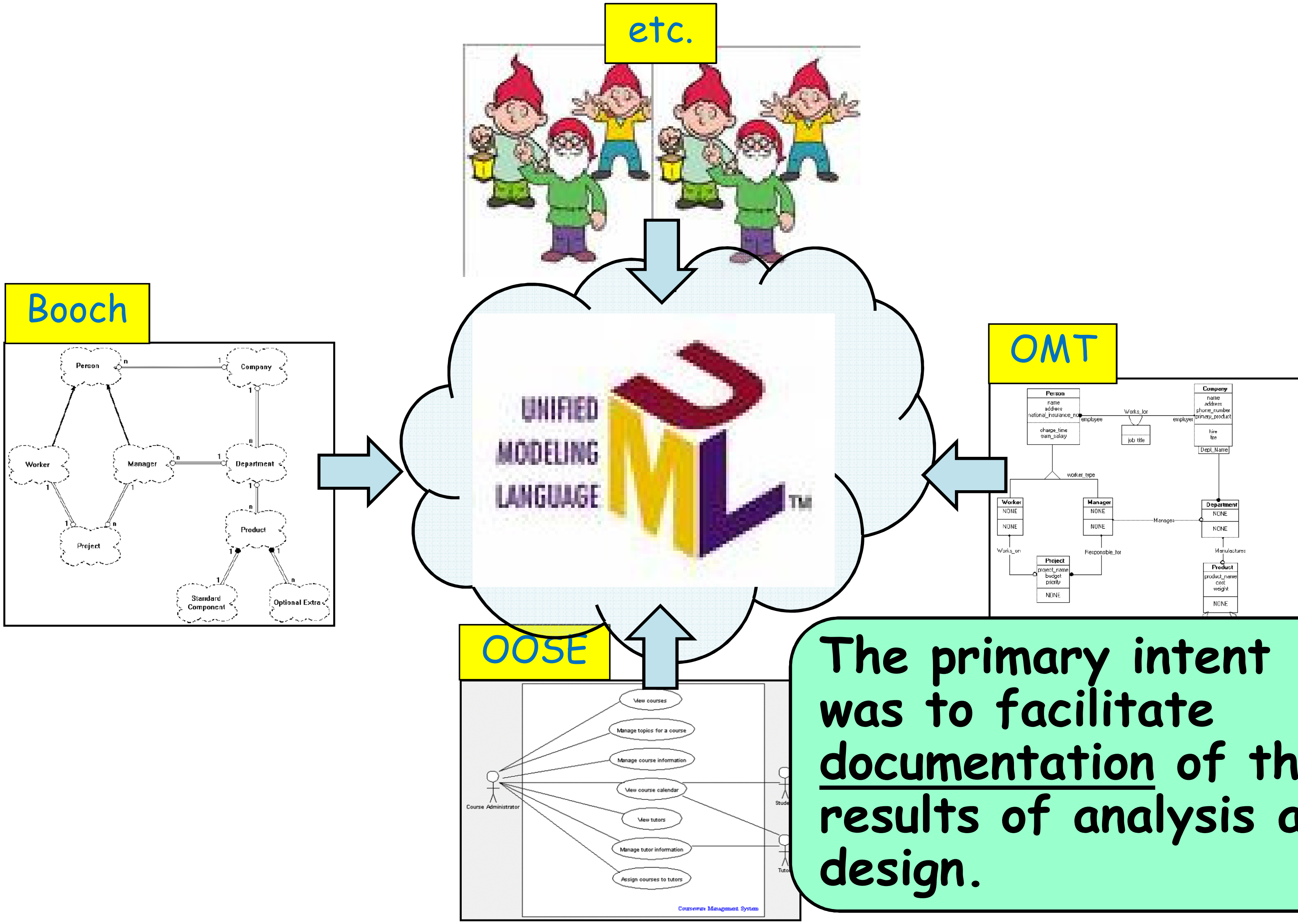
- About Model-Based Engineering (MBE)
- A Short Primer on Modeling Language Design
- The Unified Modeling Language
  - Semantics
  - UML as a DSL tool
- UML as an architectural description language

In the Beginning…

- **1987-1996:** Time of the *Method and Notation Wars*

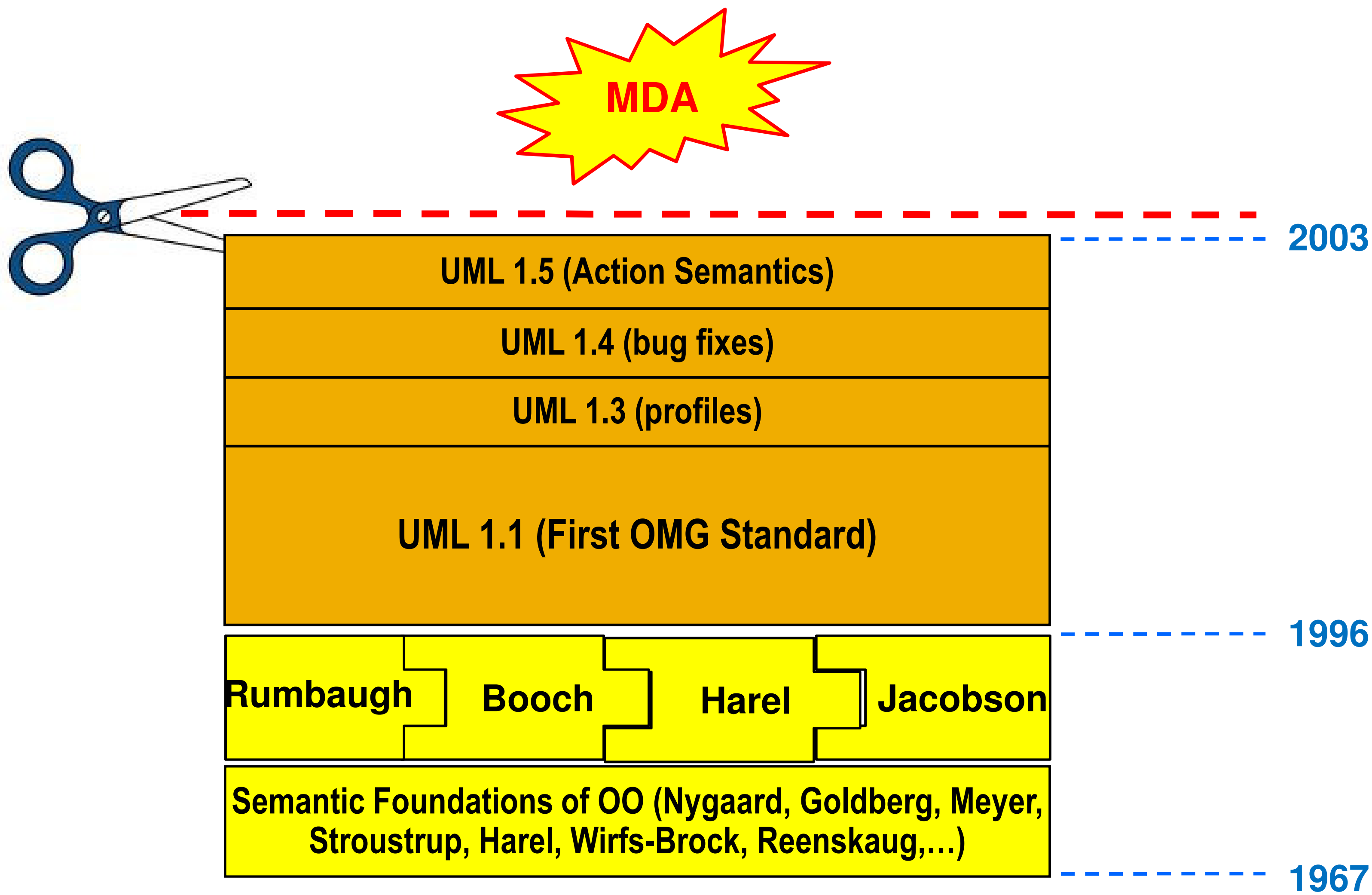


UML 1: The First Cut

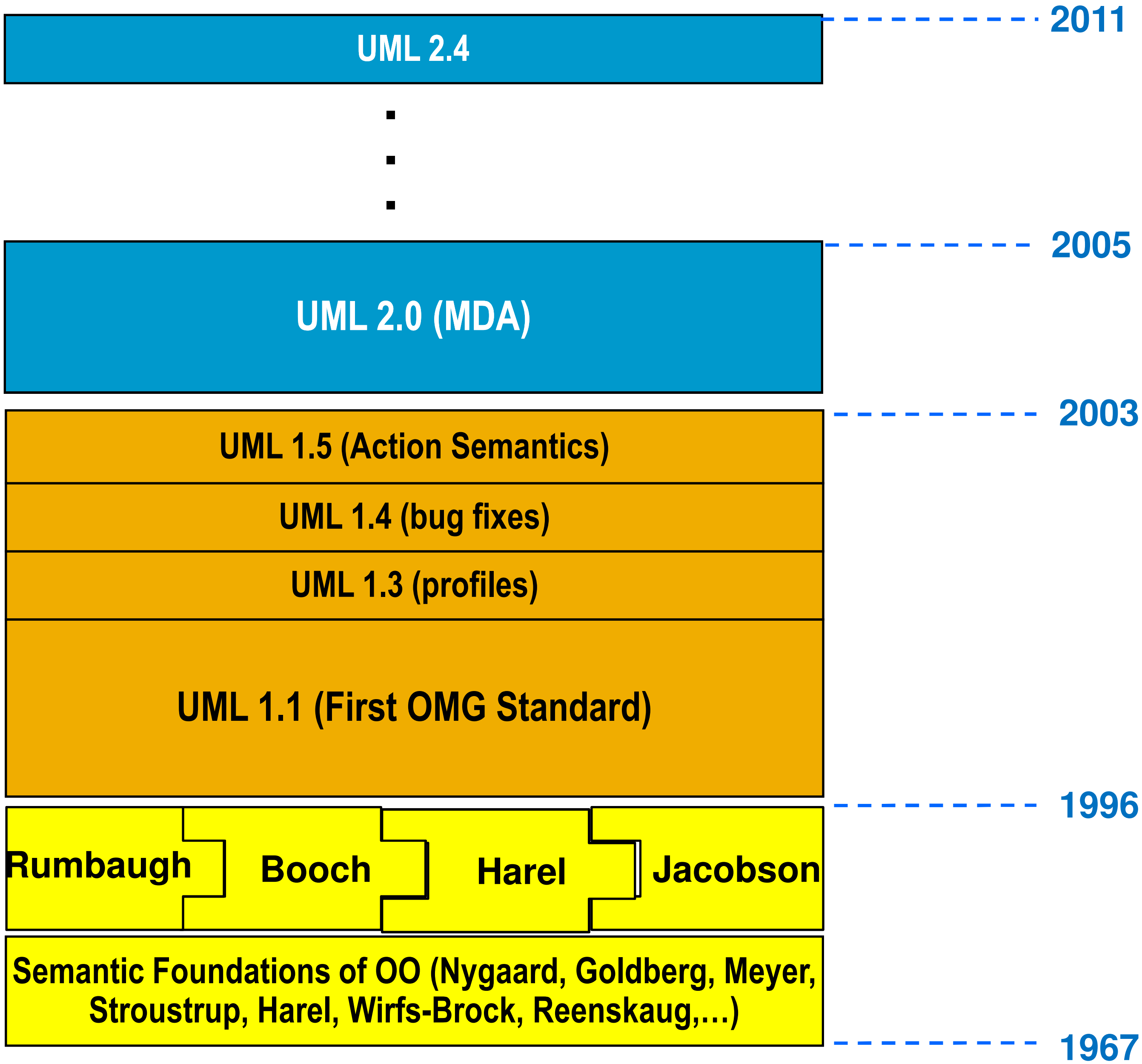


The primary intent was to facilitate <u>documentation</u> of the results of analysis and design.

UML Roots and Evolution: UML 1



UML Roots and Evolution: UML 2



Lecture Overview

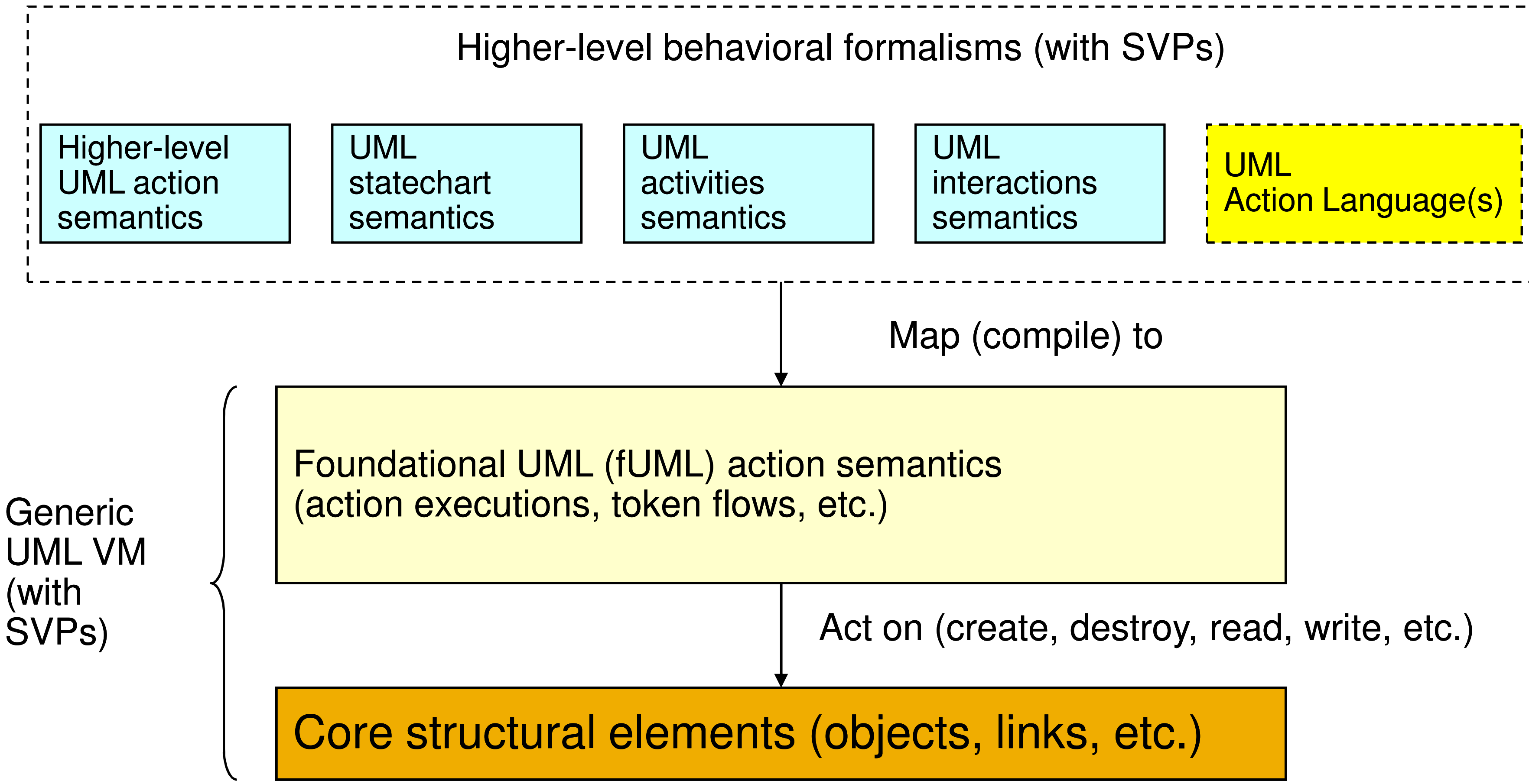
- About Model-Based Engineering (MBE)
- A Short Primer on Modeling Language Design
- The Unified Modeling Language
  - Semantics
  - UML as a DSL tool
- UML as an architectural description language

On Specifying Semantics

- **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
- **http://www.omg.org/spec/FUML/**

OMG Approach to Specifying UML Semantics

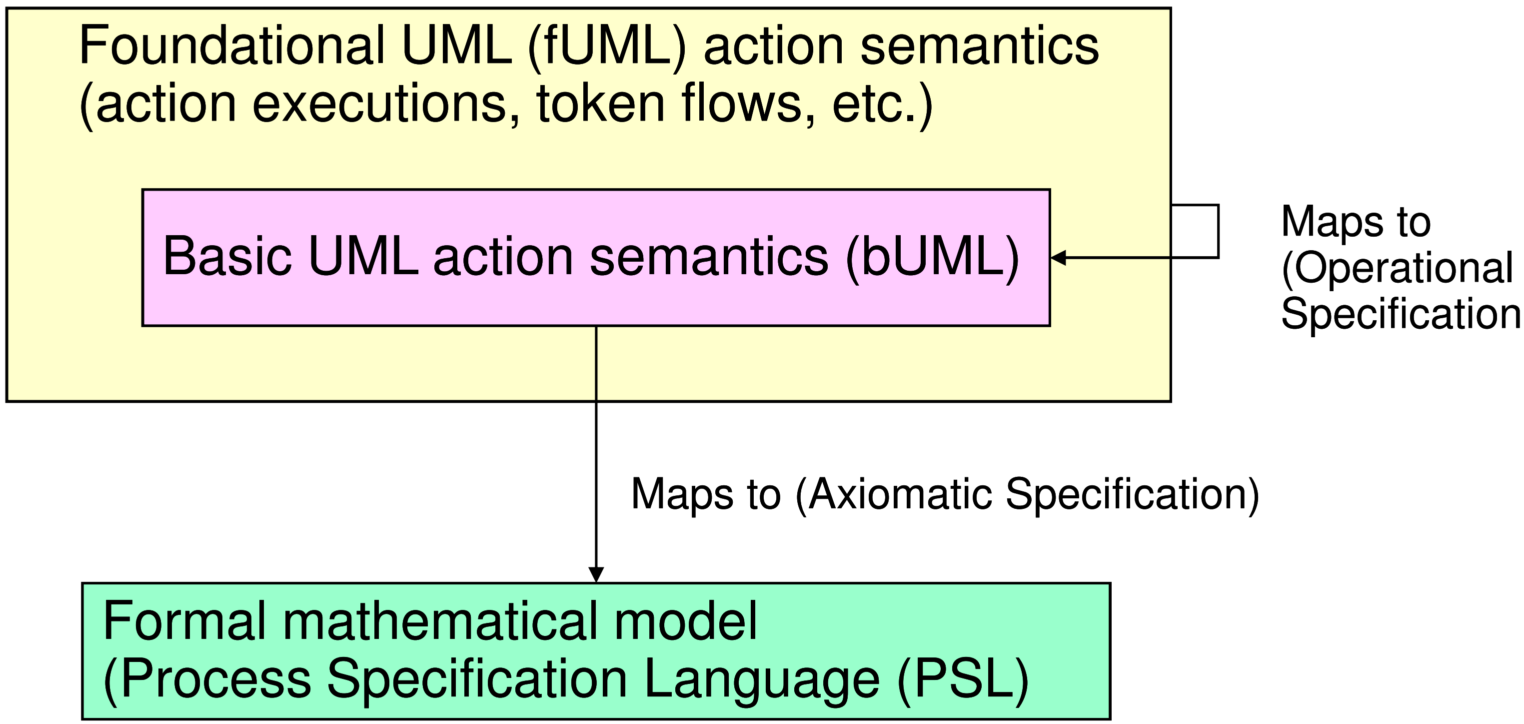
- **UML semantics hierarchy**
  - As defined by the Executable UML Foundation proto-standard



SVP = Semantic Variation Point

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

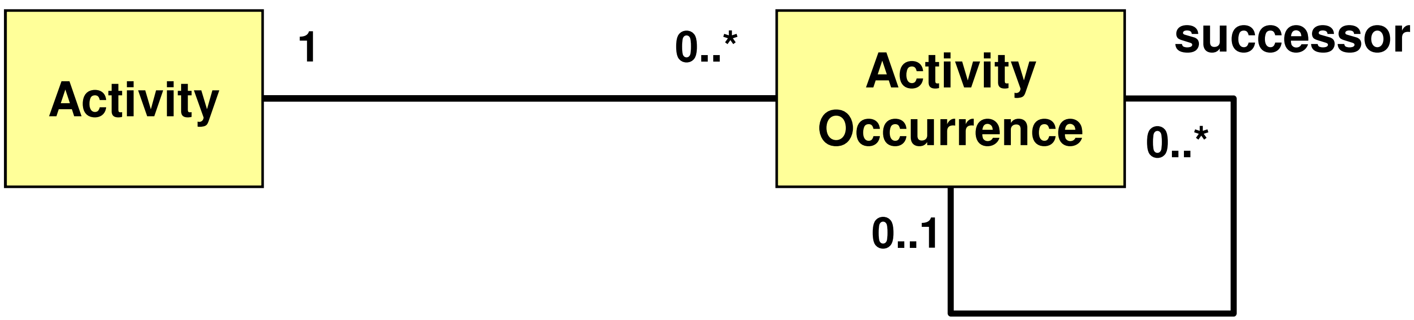
PSL: Process Specification Language

- An axiomatization of processes expressed using Common Logic Interchange Format (CLIF)
  - ISO 24707, http://standards.iso.org/ittf/PubliclyAvailableStandards/c039175_ISO_IEC_24707_2007(E).zip.
  - ISO 18629, see http://www.conradbock.org/#PSL
- Example:

Property values must be classified by the type of the property.

```
(forall (p c occ f o v)
      (if (and (buml:type p c)
               (psl:occurrence occ)
               (psl:legal occ)
               (psl:prior f occ)
               (form:property-value o p v f))
      (exists (f2)
               (and (psl:prior f2 occ)
                    (form:classifies c v f2))))))
```

PSL Model of Execution

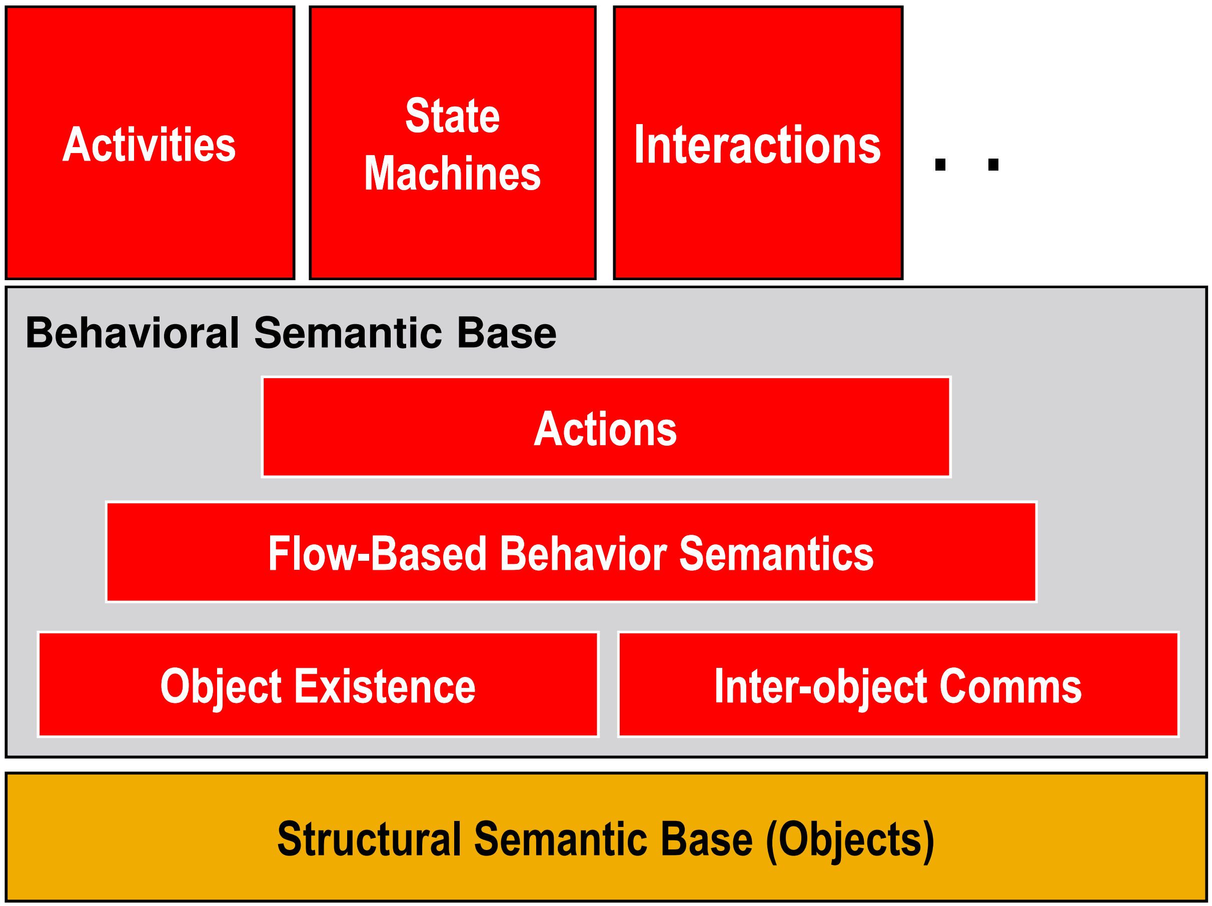


- Results in a tree of possible executions
- PSL rules are constraints on valid executions (tree pruning)

UML Model of Computation

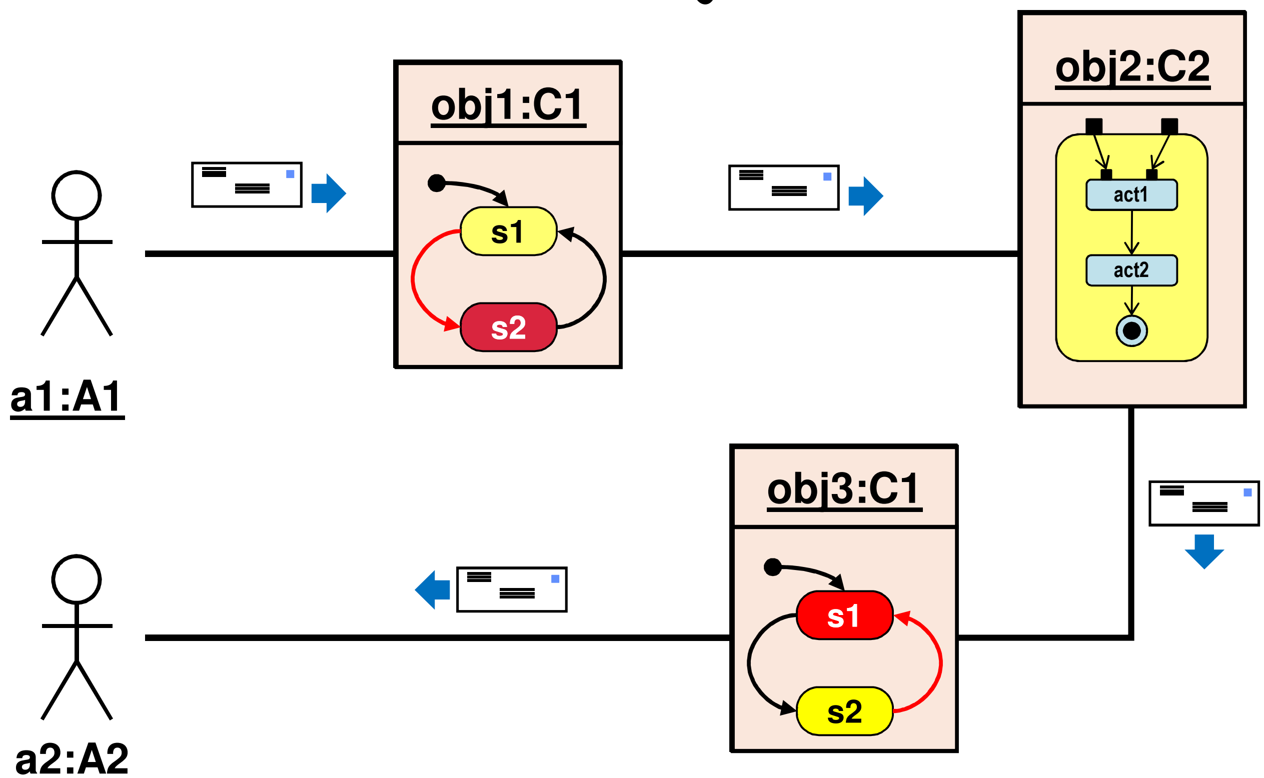
- **Object oriented**
  - All behavior stems from (active) objects
- **Distributed**
  - Multiple sites of execution ("localities")
- **Concurrent**
  - Active objects $\Rightarrow$ multiple threads of execution
- **Heterogeneous causality model**
  - Event driven at the highest level
  - Data and control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

UML Run-Time (Dynamic) Semantics Architecture



UML Model of Causality (How Things Happen)

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



How Things Happen in UML

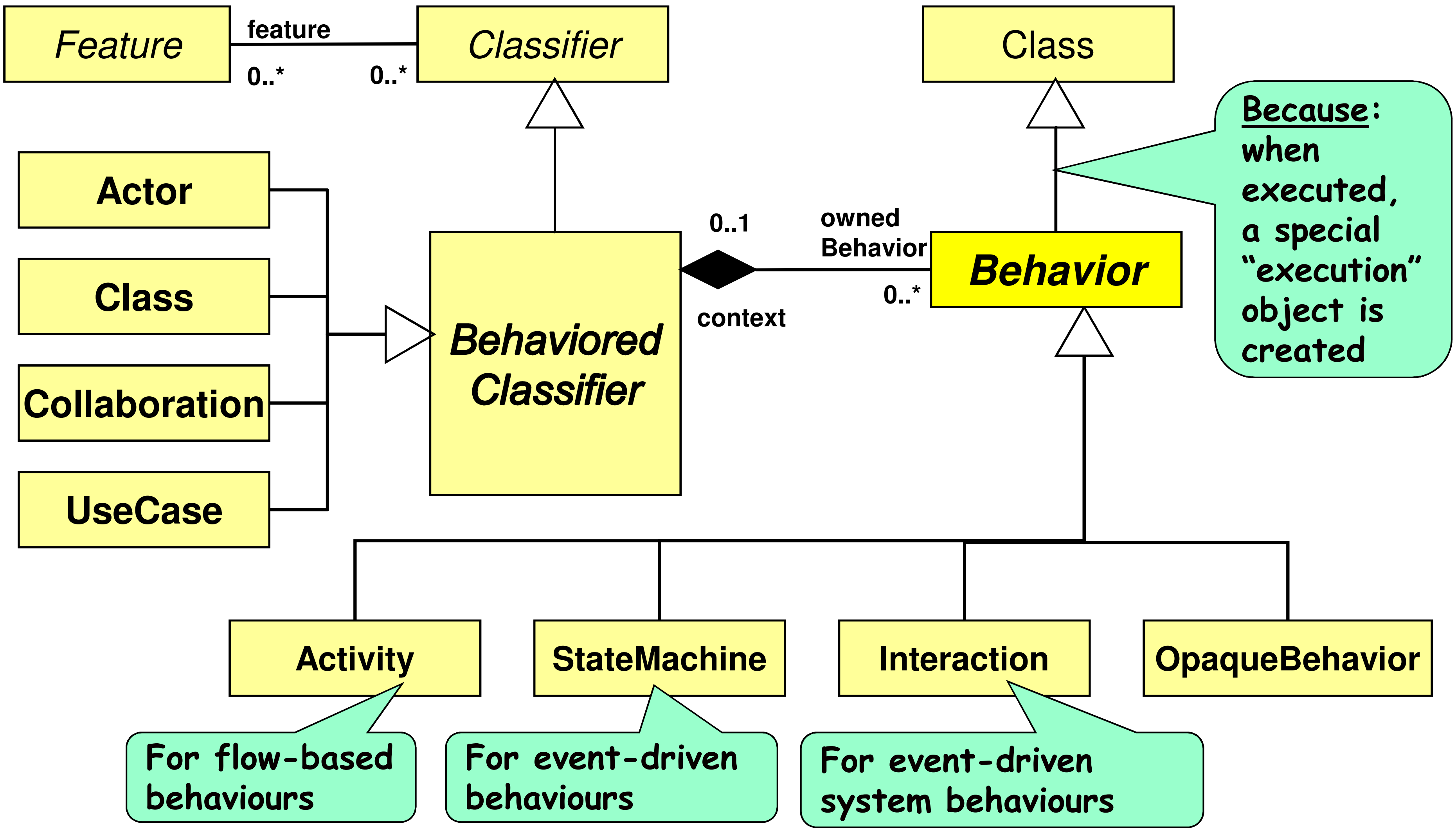
- **An action is executed by an object**
  - May change the contents of one or more variables or slots
  - If it is a communication ("messaging") action, it may:
    - Invoke an operation on another object
    - Send a signal to another object
    - Either one will eventually cause the execution of a 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

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

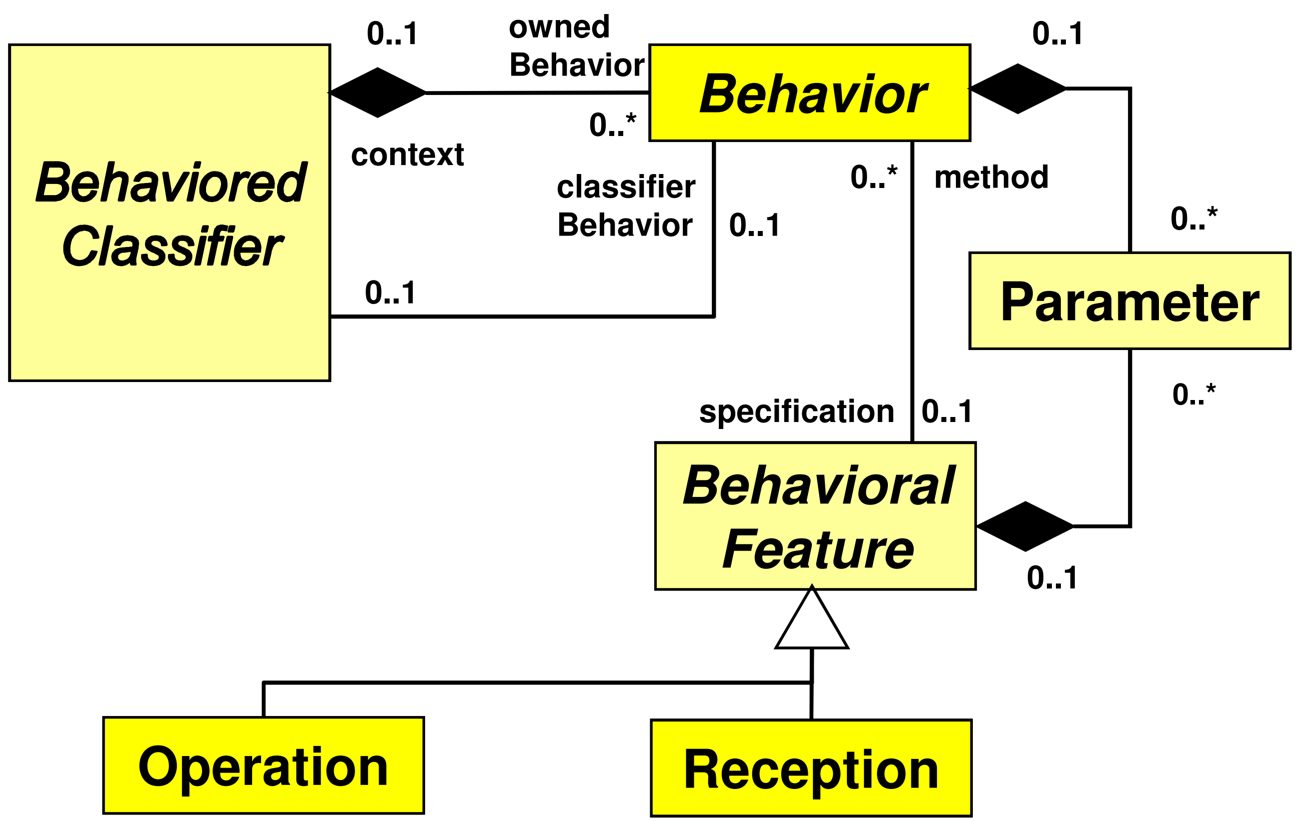
Relationship Between Structure and Behaviour

- From the UML metamodel:



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



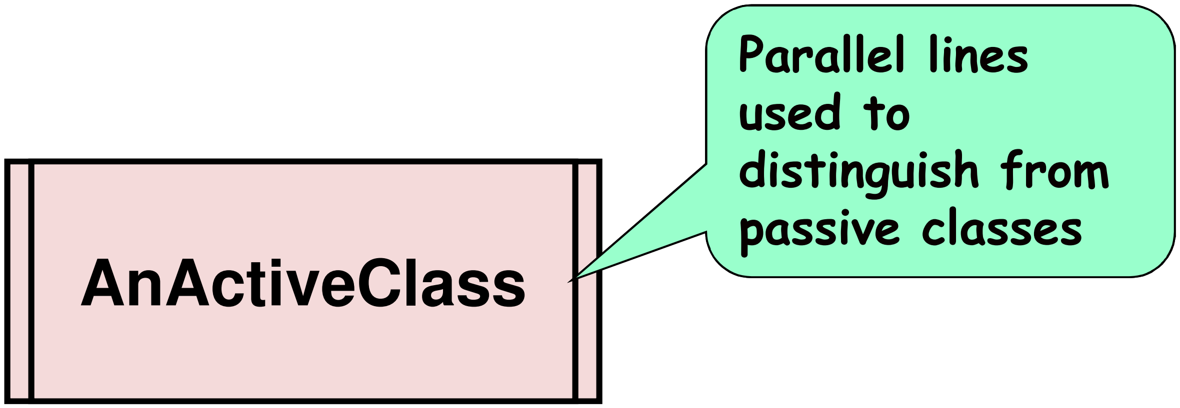
Active Object Definition

- **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.

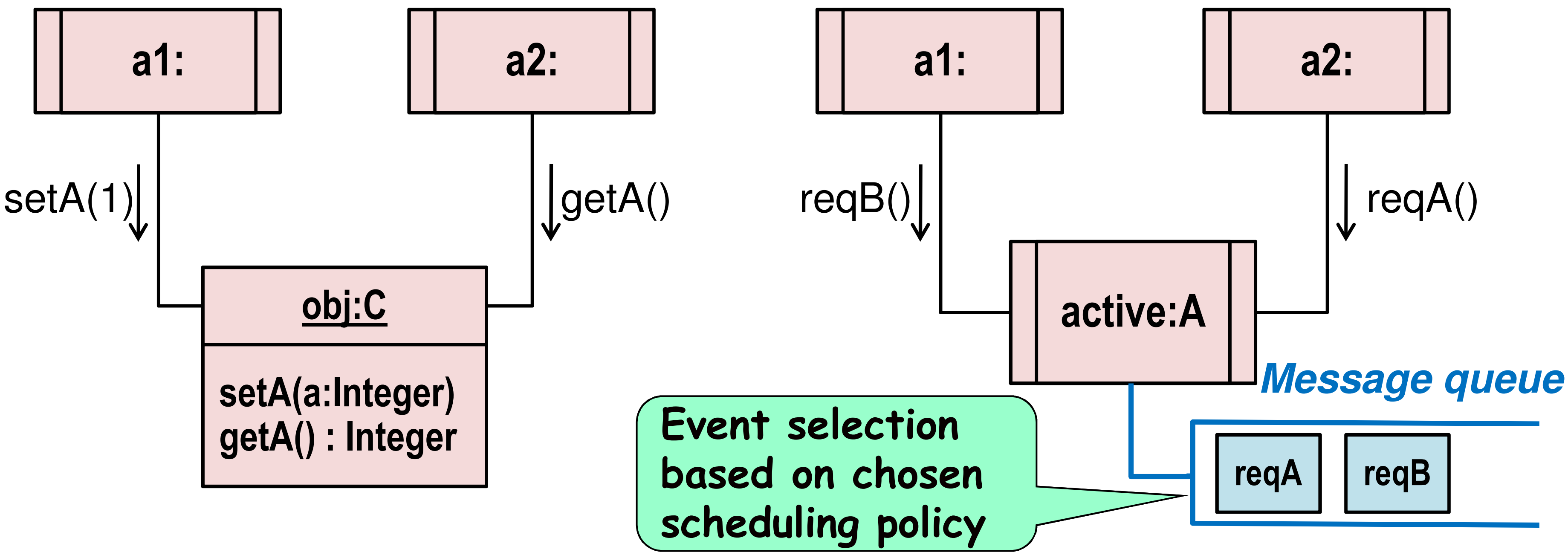
- **Also:**

  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...



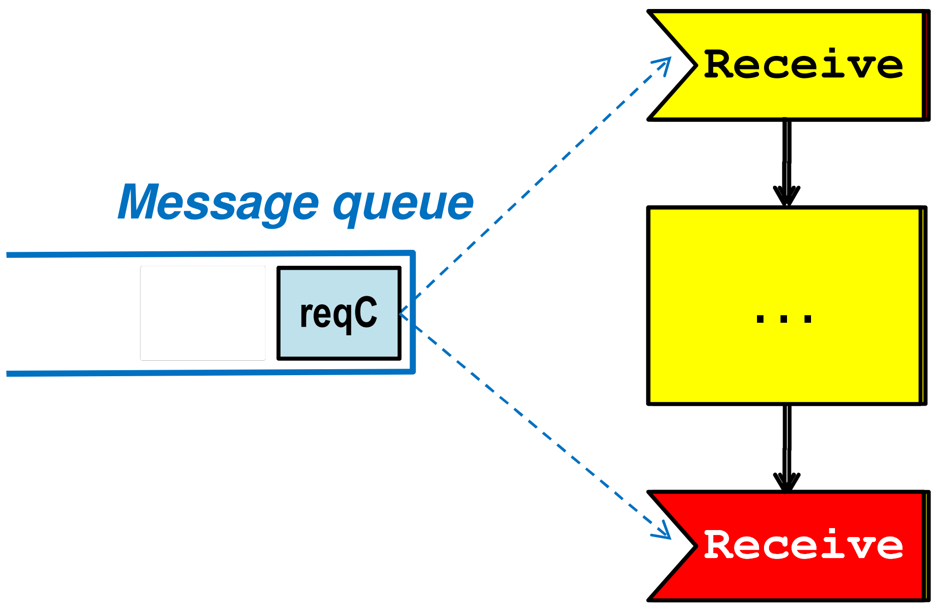
Passive vs. Active Objects

- 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



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

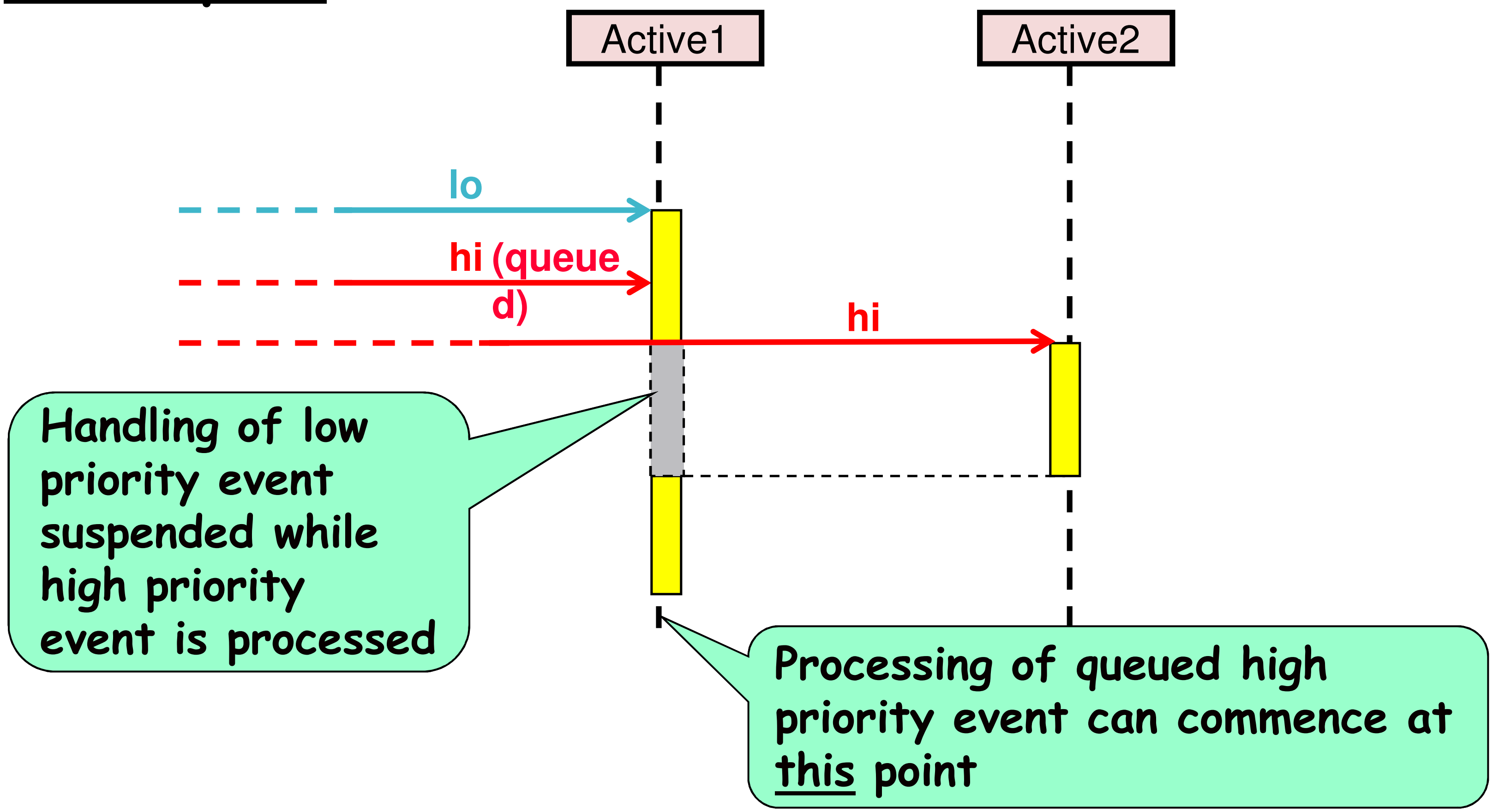


The Problem with RTC

- <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)

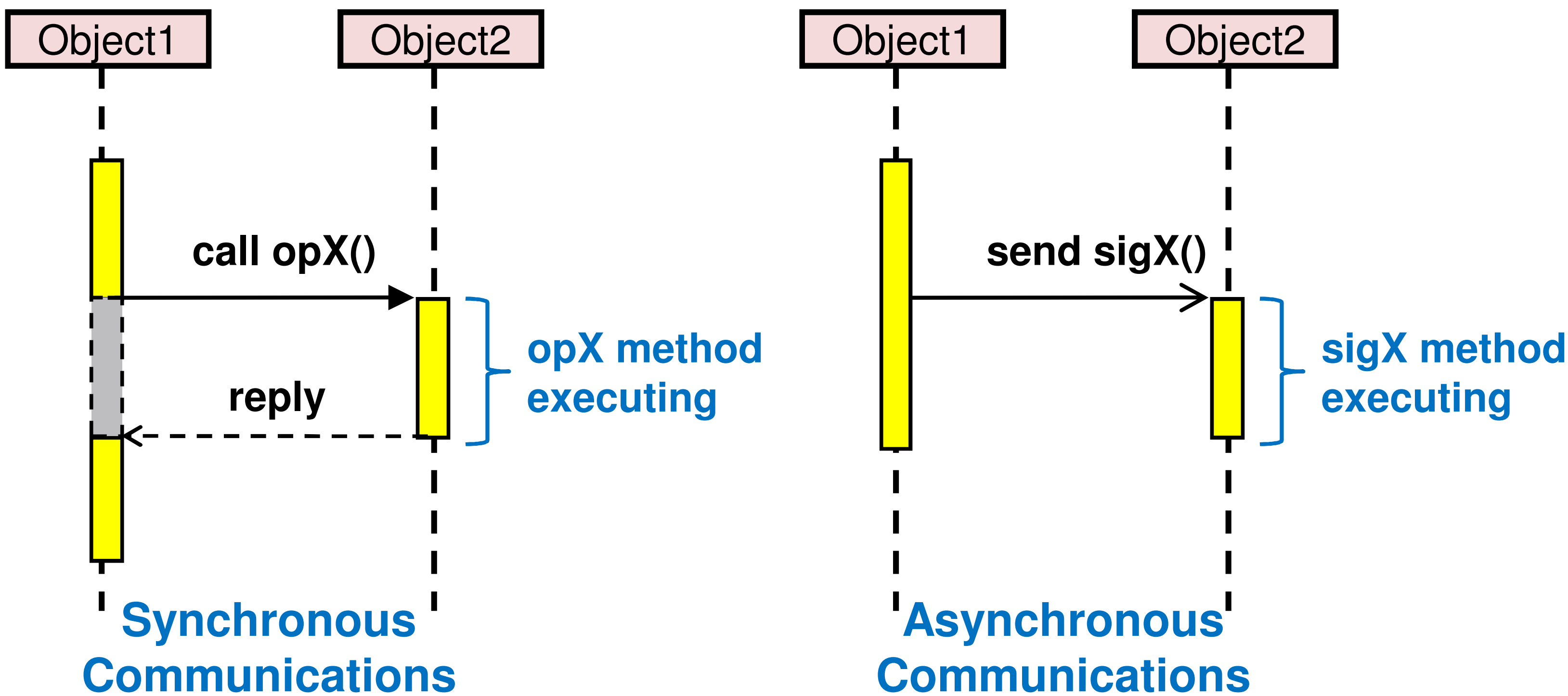
RTC Semantics

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



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)



Purpose of UML Actions

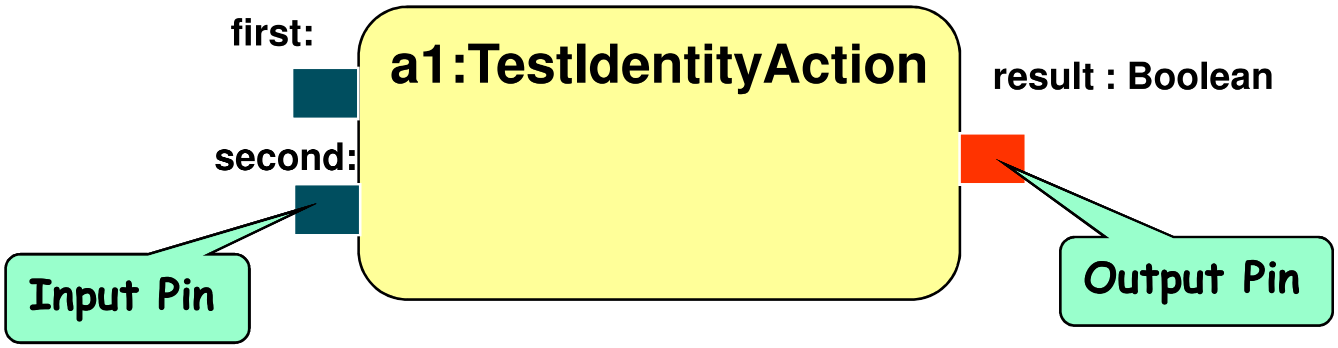
- **For modelling fine-grained behavioural phenomena which manipulates and accesses UML entities (objects, links, attributes, operations, etc.)**
  - E.g. create link, write attribute, destroy object
  - A kind of UML “assembler”
- **The UML standard defines:**
  - A set of actions and their semantics (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

Categories of UML Actions

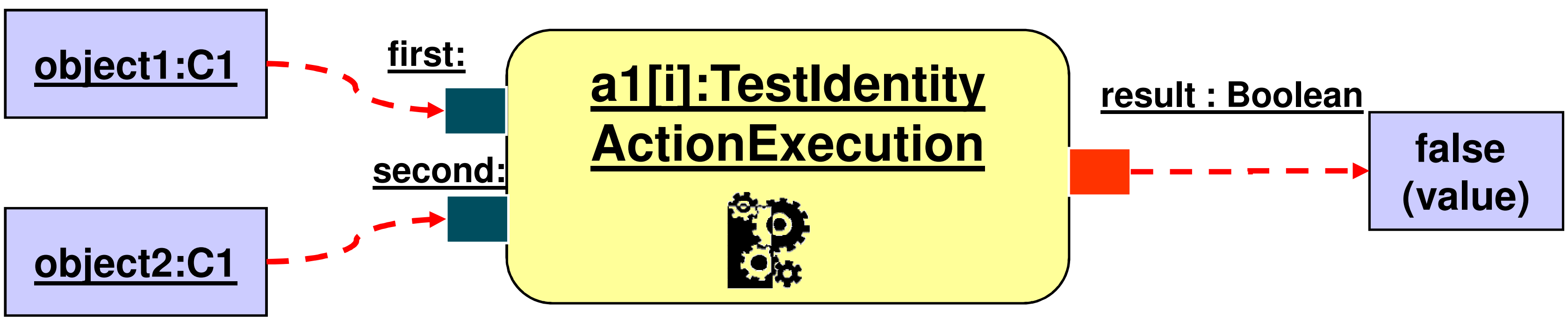
- **Capabilities covered**
  - Communication actions (send, call, receive,…)
  - Primitive function action
  - Object actions (create, destroy, reclassify,start,…)
  - Structural feature actions (read, write, clear,…)
  - Link actions (create, destroy, read, write,…)
  - Variable actions (read, write,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

Action Specifications and Action Executions

Action Specification (a design-time specification)



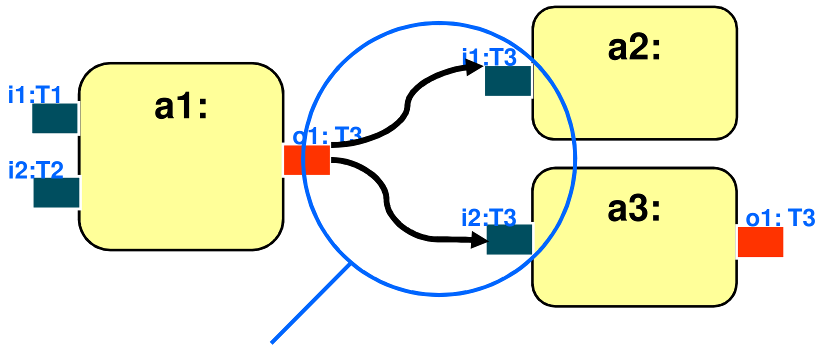
Action Execution (a run-time concept)



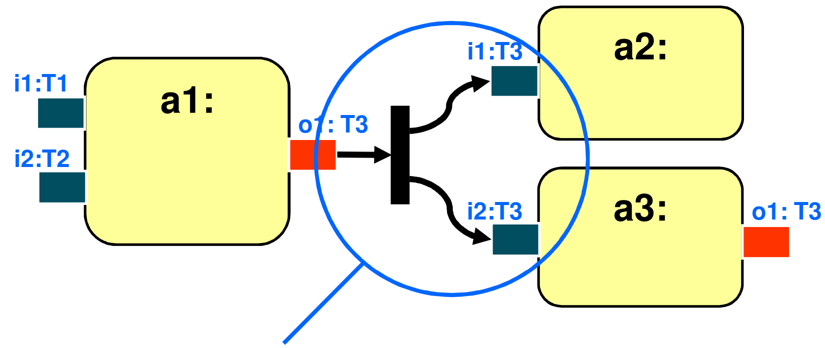
NB: each time action a1 needs to be executed, a new action execution is created

Combining Actions

- Data flow MoC: output to input connections

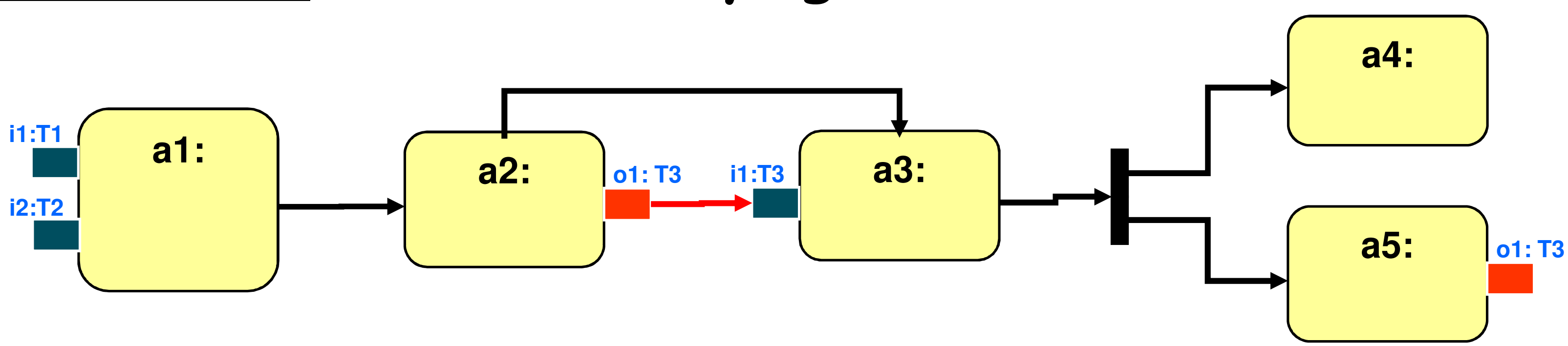


Contention (a2 and a3)



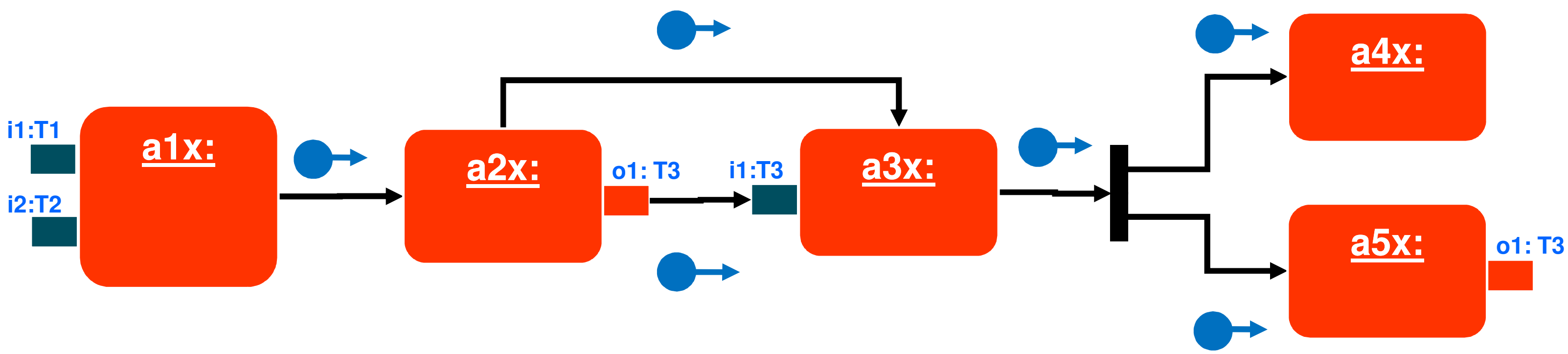
Data replication

- Control flow MoC: identifying successor actions



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

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

Lecture Overview

- About Model-Based Engineering (MBE)
- A Short Primer on Modeling Language Design
- The Unified Modeling Language
  - Semantics
  - UML as a DSL tool
- UML as an architectural description language

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)**

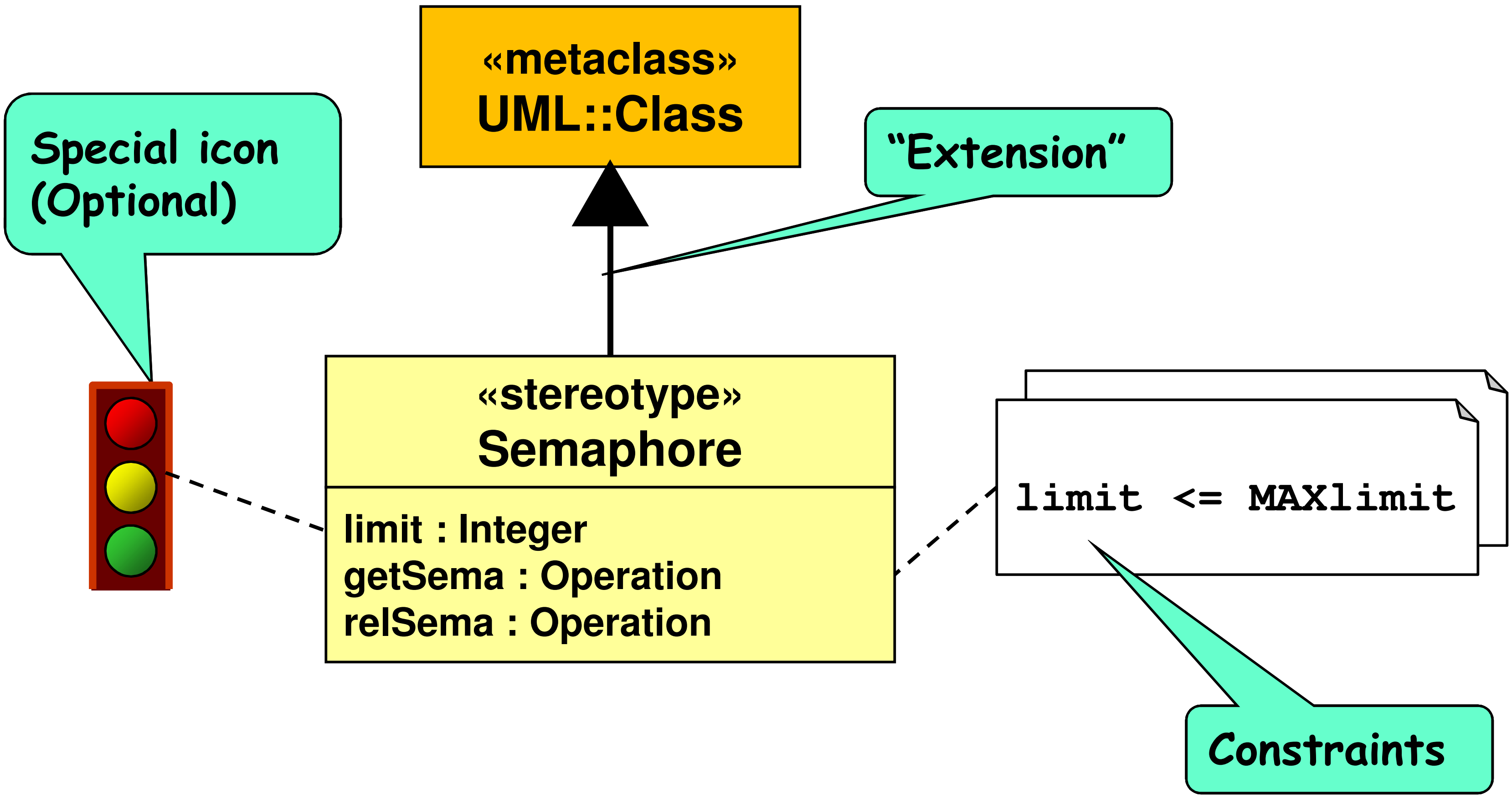
Example: Adding a Semaphore Concept to UML

- **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

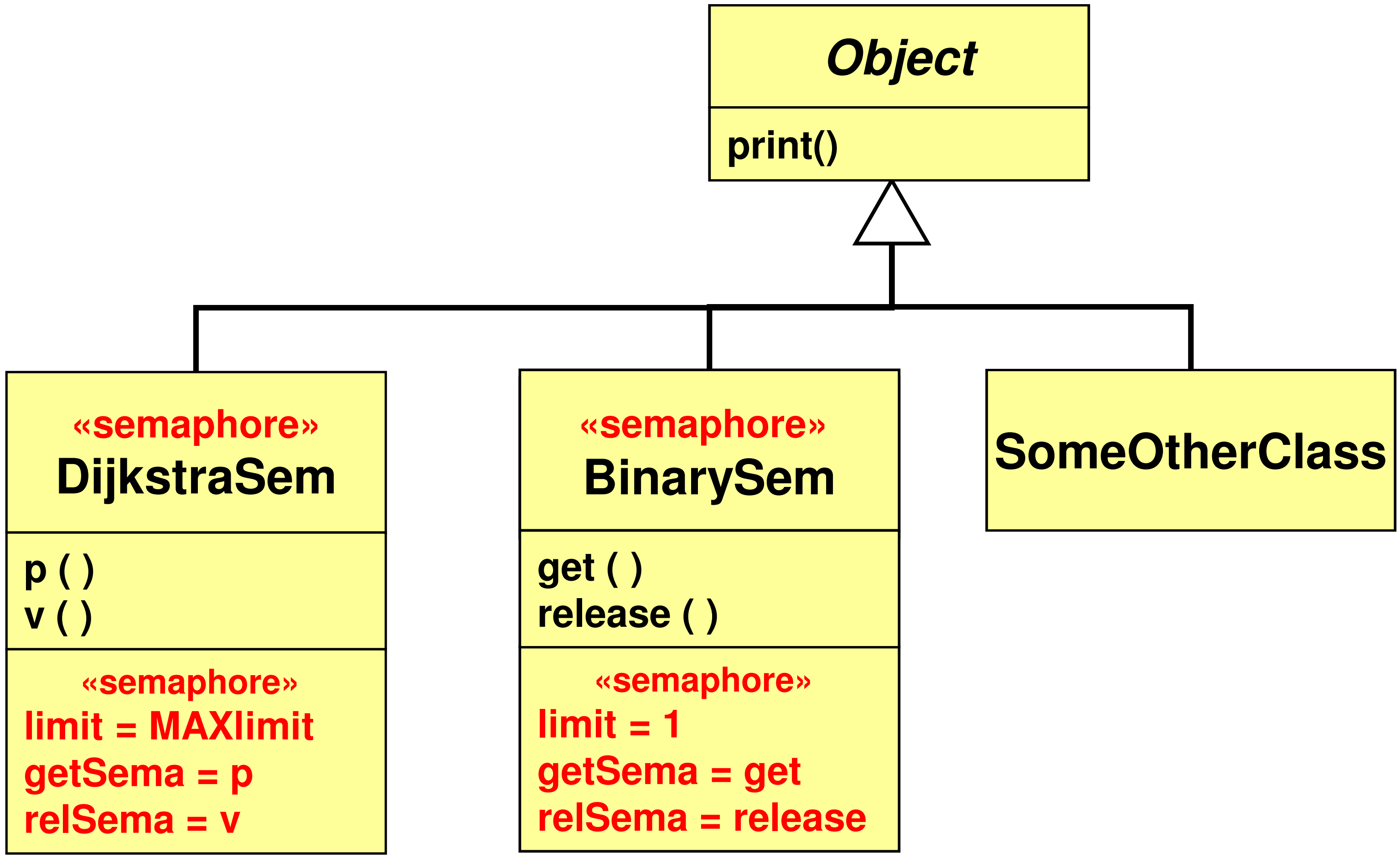
Example: The Semaphore Stereotype

- **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?)

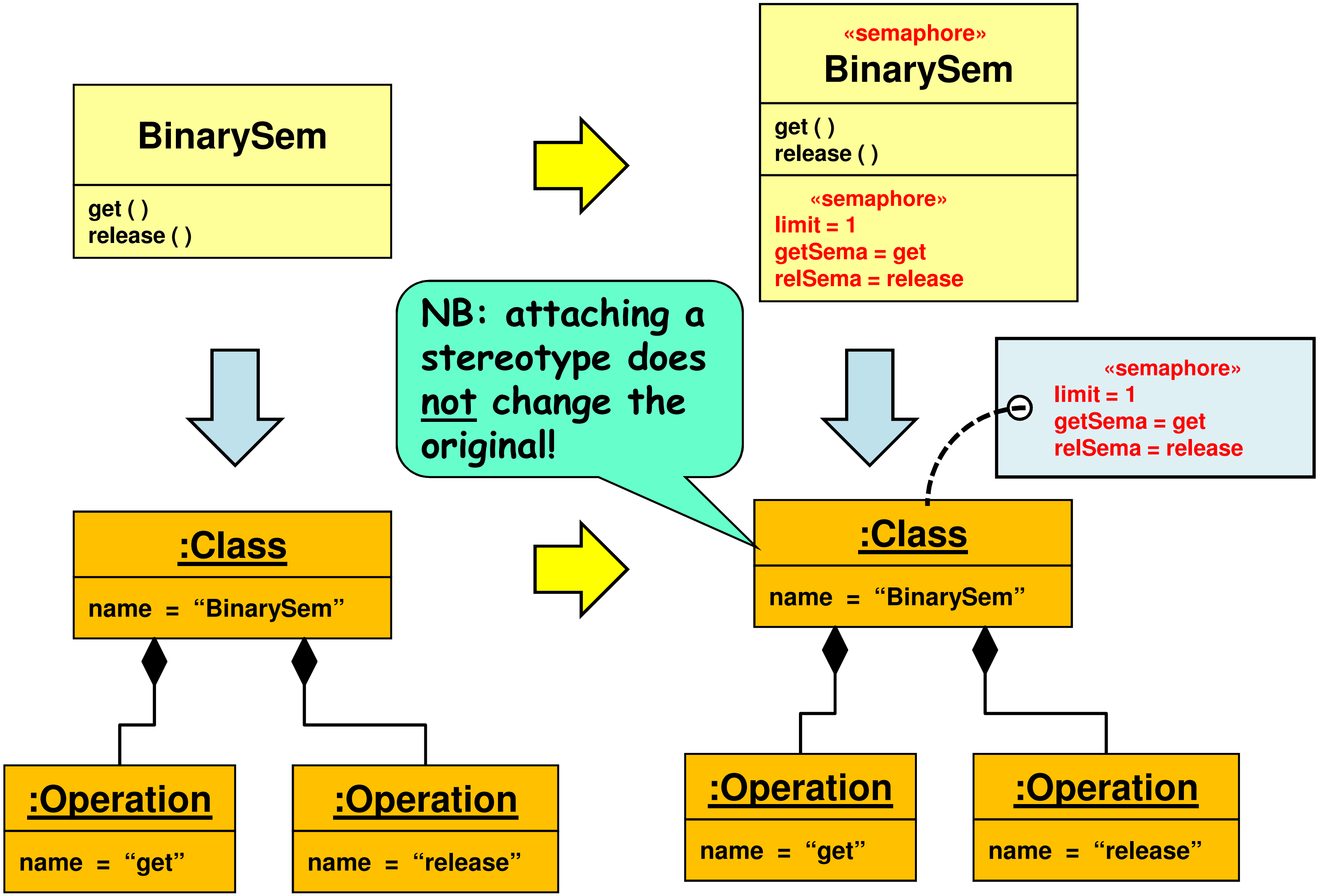
Example: Graphical Definition of the Stereotype



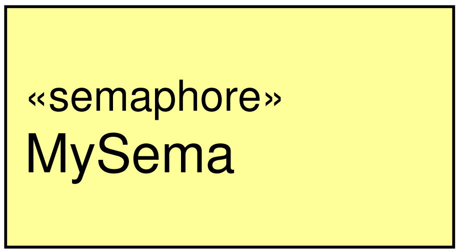
Example: Applying the Stereotype



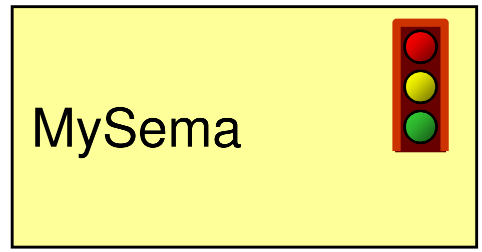
The Semantics of Stereotype Application



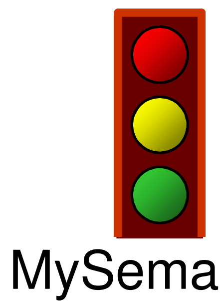
Stereotype Representation Options



(a)



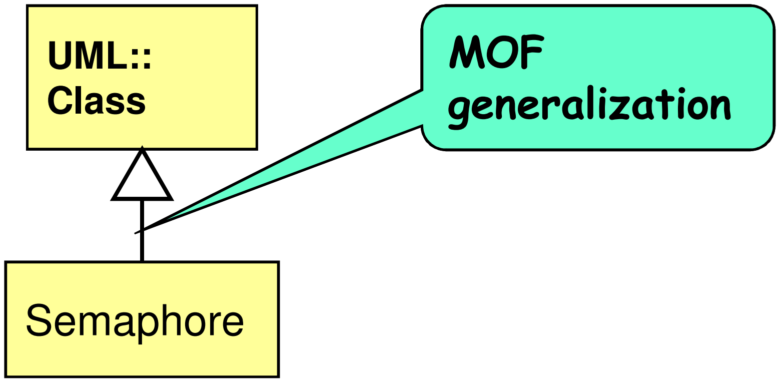
(b)



(c)

Why are Stereotypes Needed?

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

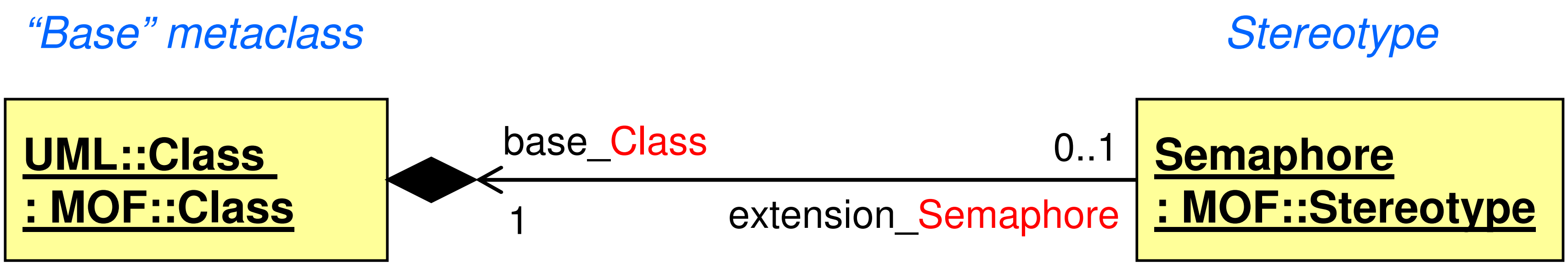


Rationale:

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)

The MOF Semantics of UML Extension

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

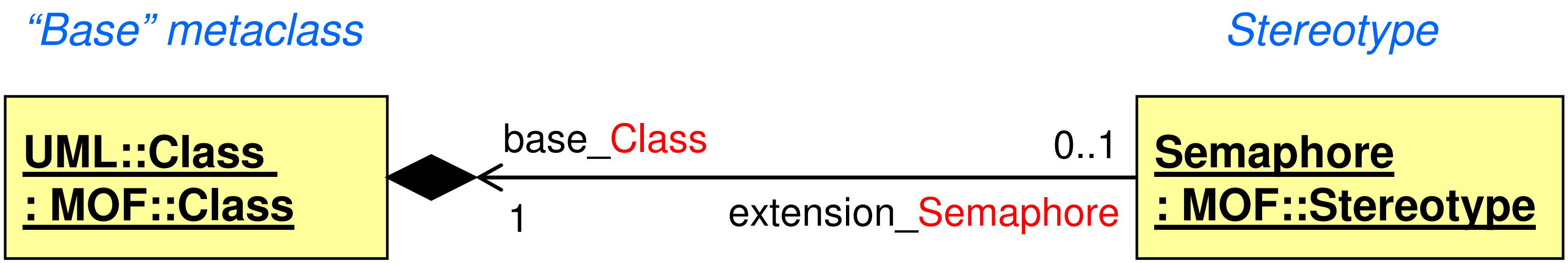


- *Association ends naming convention:*

```
base_<base-class-name>
extension_<stereotype-name>
```

- **Required for writing correct OCL constraints for stereotypes**

Example: OCL Constraint for a Stereotype

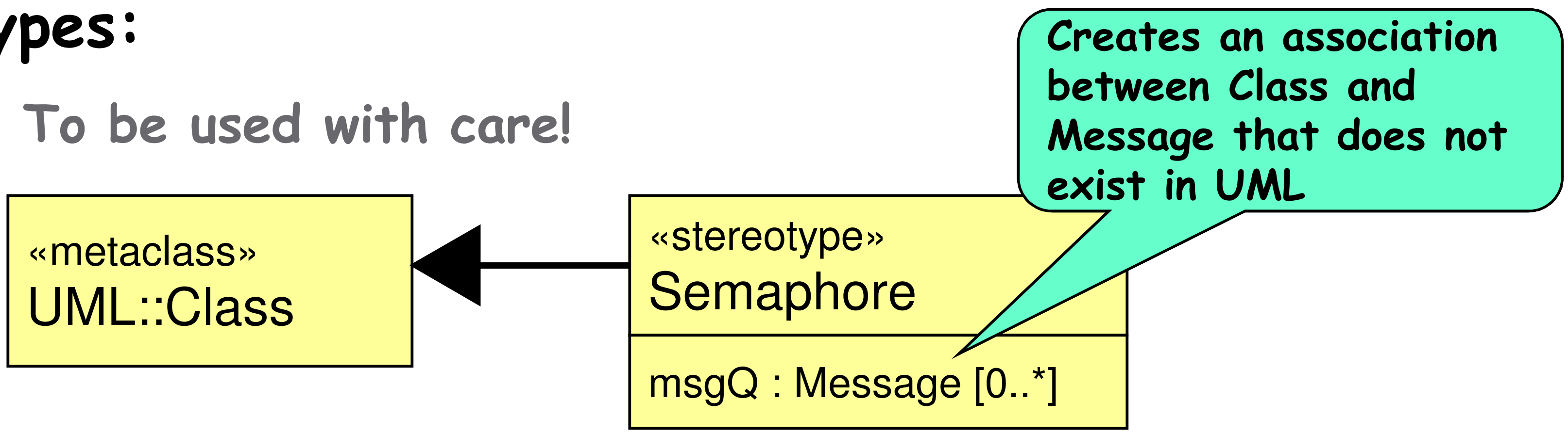


- **Semaphore constraint:**
  the base Class must have an owned ordered attribute called "msgQ" of type Message

```
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))
```

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 capability added via stereotype attribute types:**
  - To be used with care!

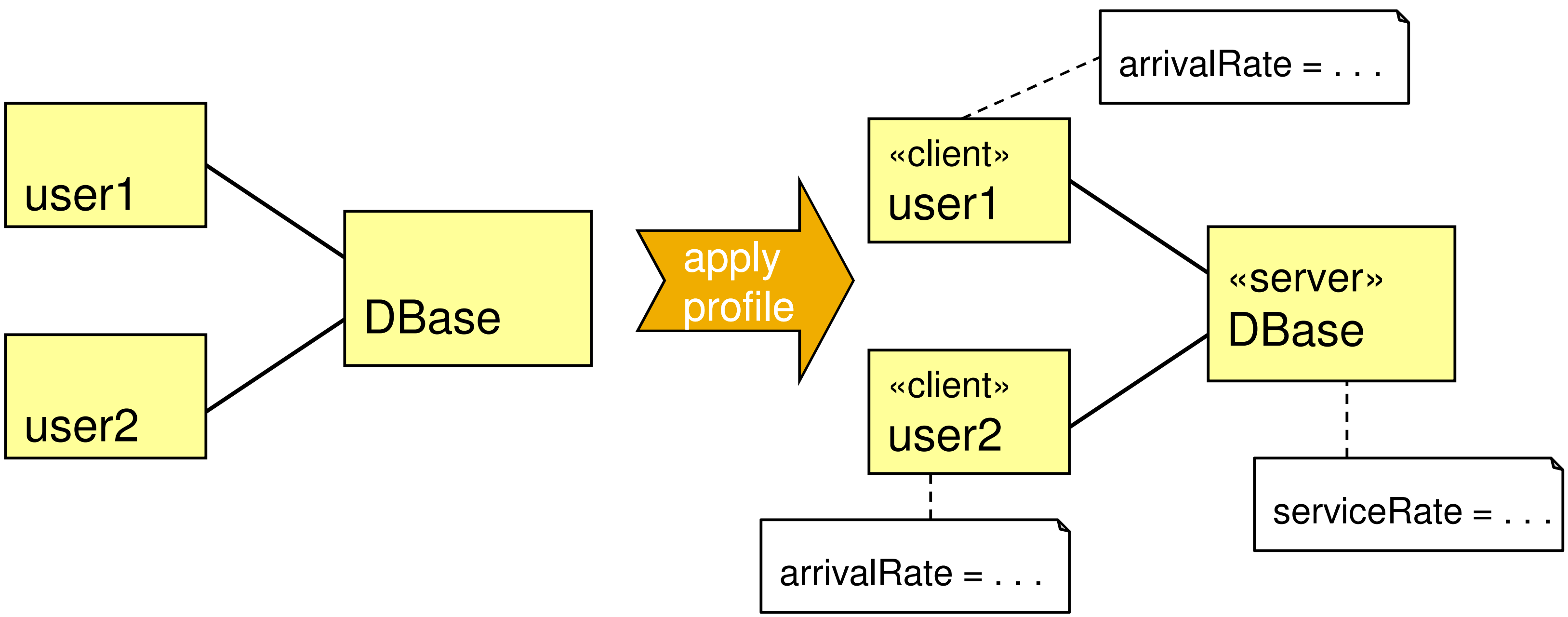


UML Profiles

- **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)

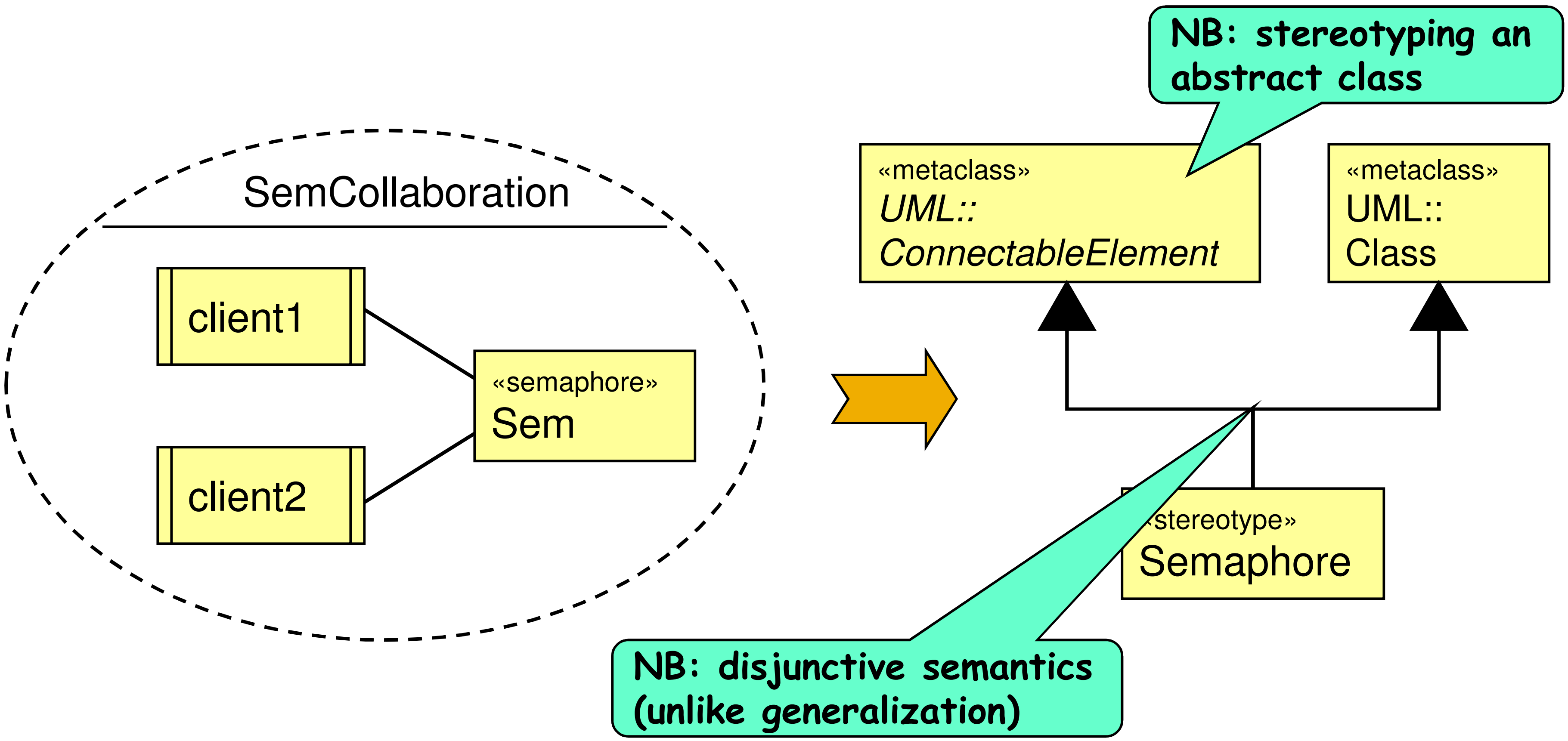
Re-Casting Models Using Profiles

- **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**



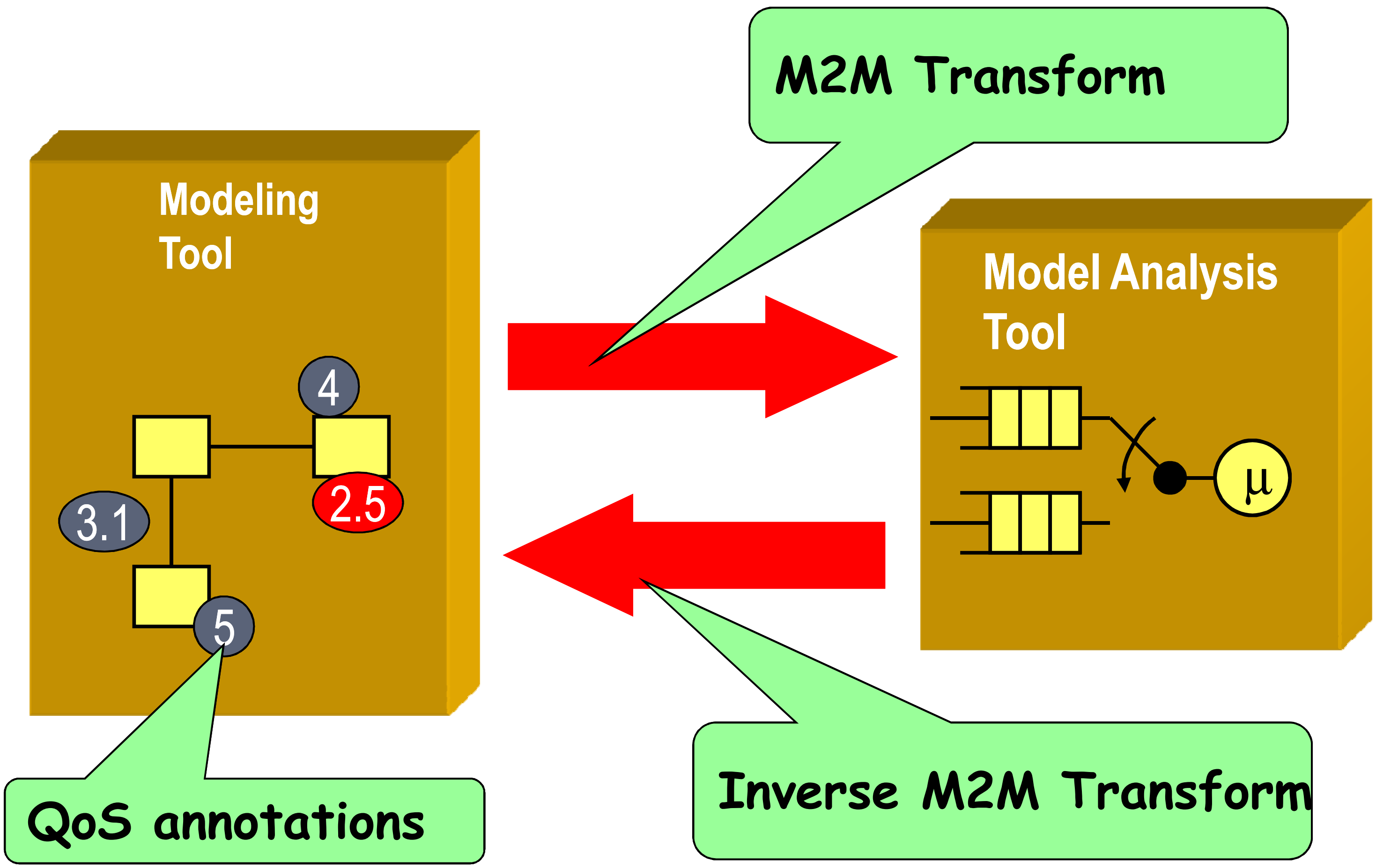
Multi-Base Stereotypes

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



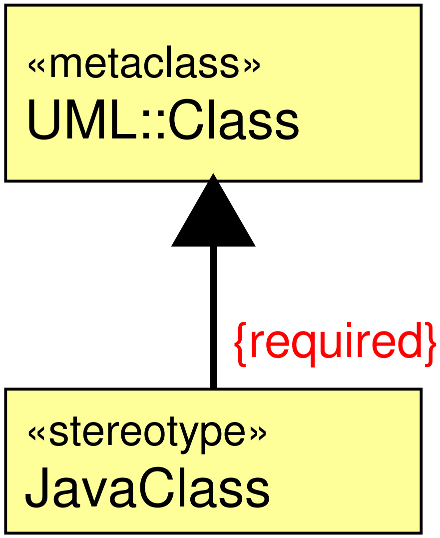
Analysis with Cast Profiles

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



"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



Strict Profile Application

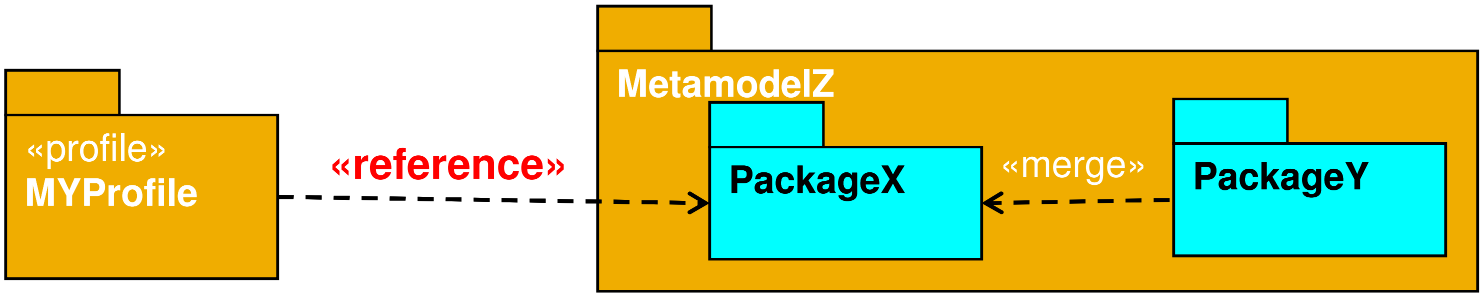
- A *strict* application of a profile will hide from view all model elements that do not have a corresponding stereotype in that profile
  - Convenient for generating views
- Strictness is a characteristic of the profile application and not of the profile itself
  - Any given profile can be applied either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

Metamodel Subsetting with Profiles (1)

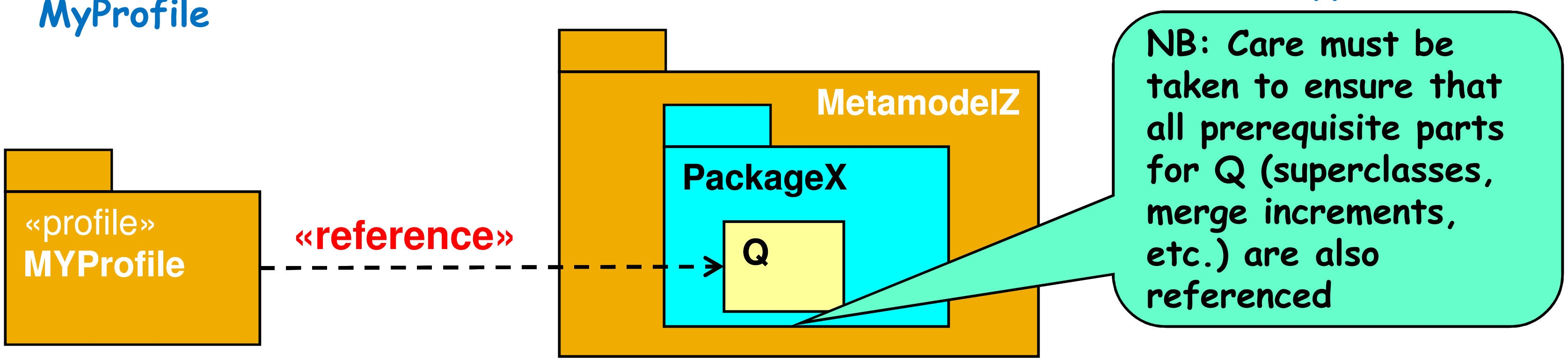
- **It is often useful to remove segments of the full UML metamodel resulting in a minimal DSML definition**
  - NB: Different mechanism from strict profile application – the hiding is part of the profile definition and cannot be applied selectively
- **The UML 2.1 profile mechanism adds controls that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

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

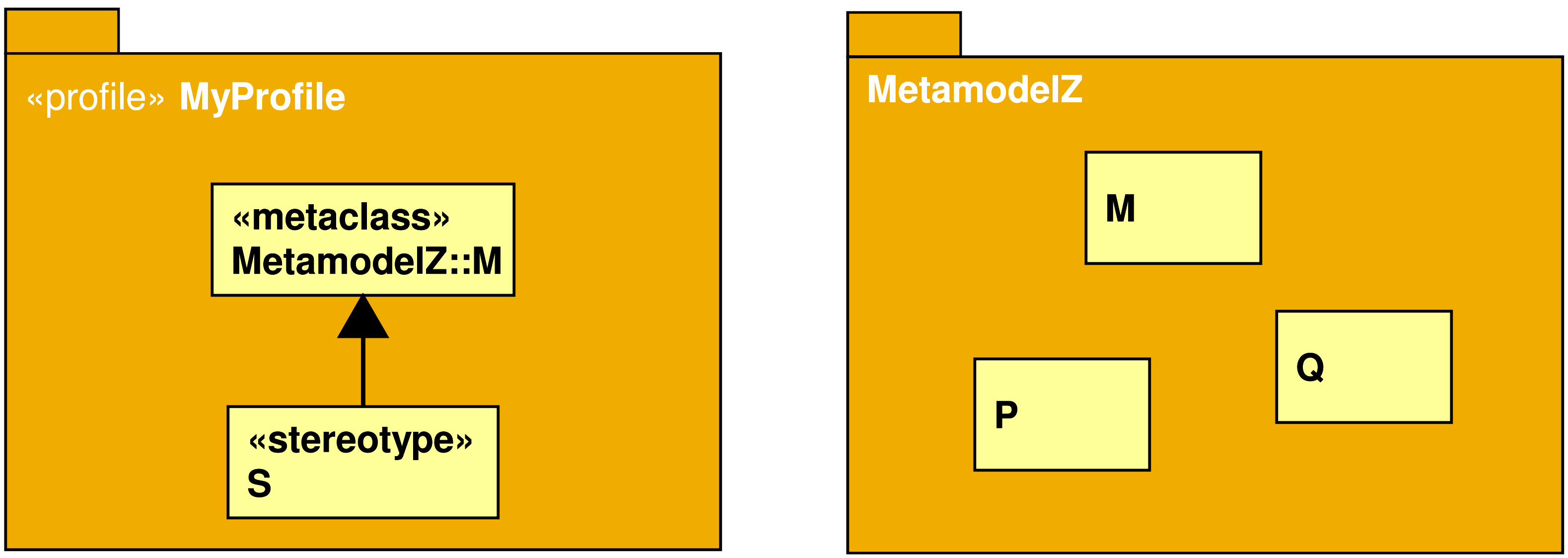


- **Case 2: Explicit Metaclass Reference**
  - Metaclass Q is visible and can serve as a base metaclass for stereotypes in MyProfile



Metamodel Subsetting with Profiles (3)

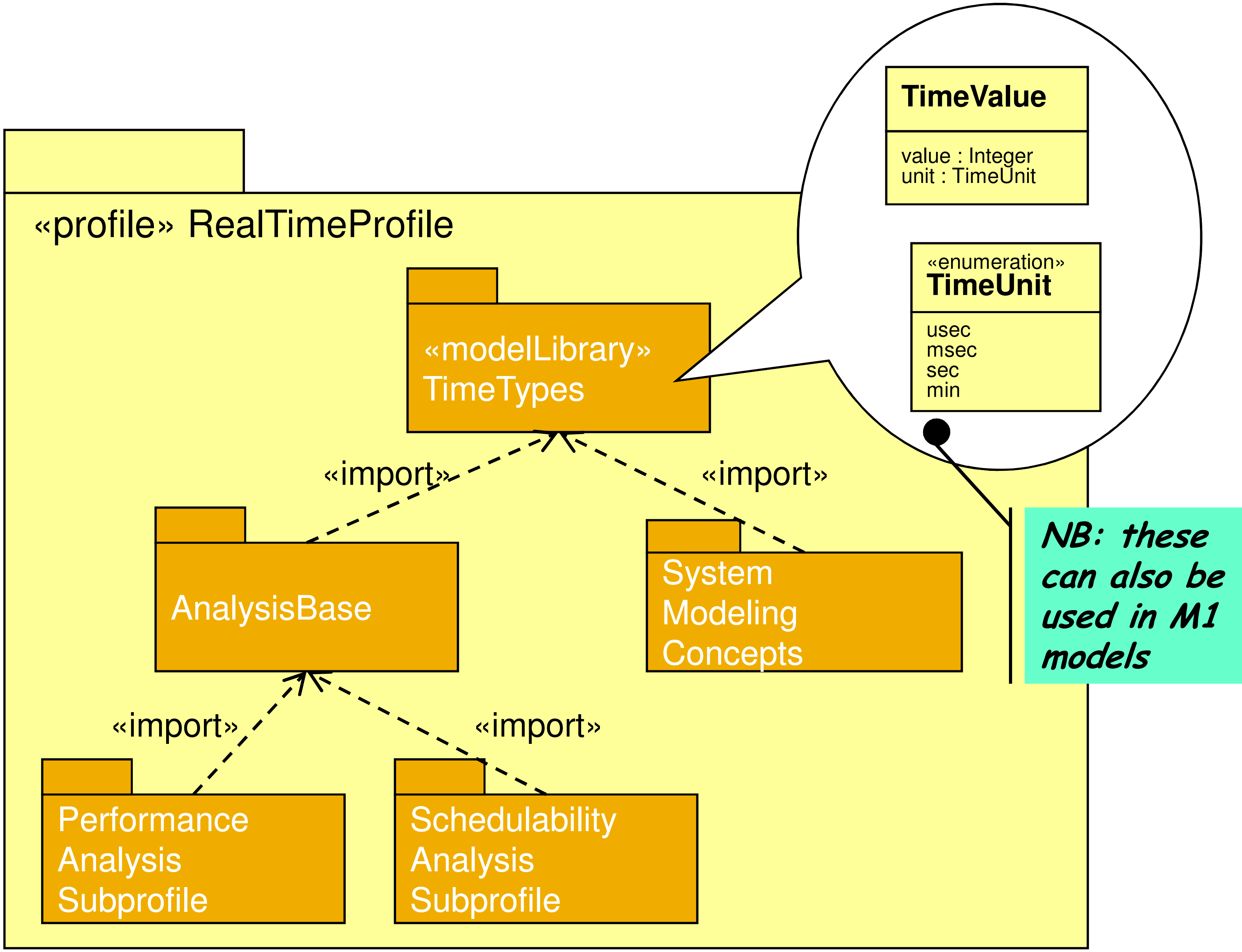
- **Case 3: Implicit metaclass reference**
  - Metaclass M is visible



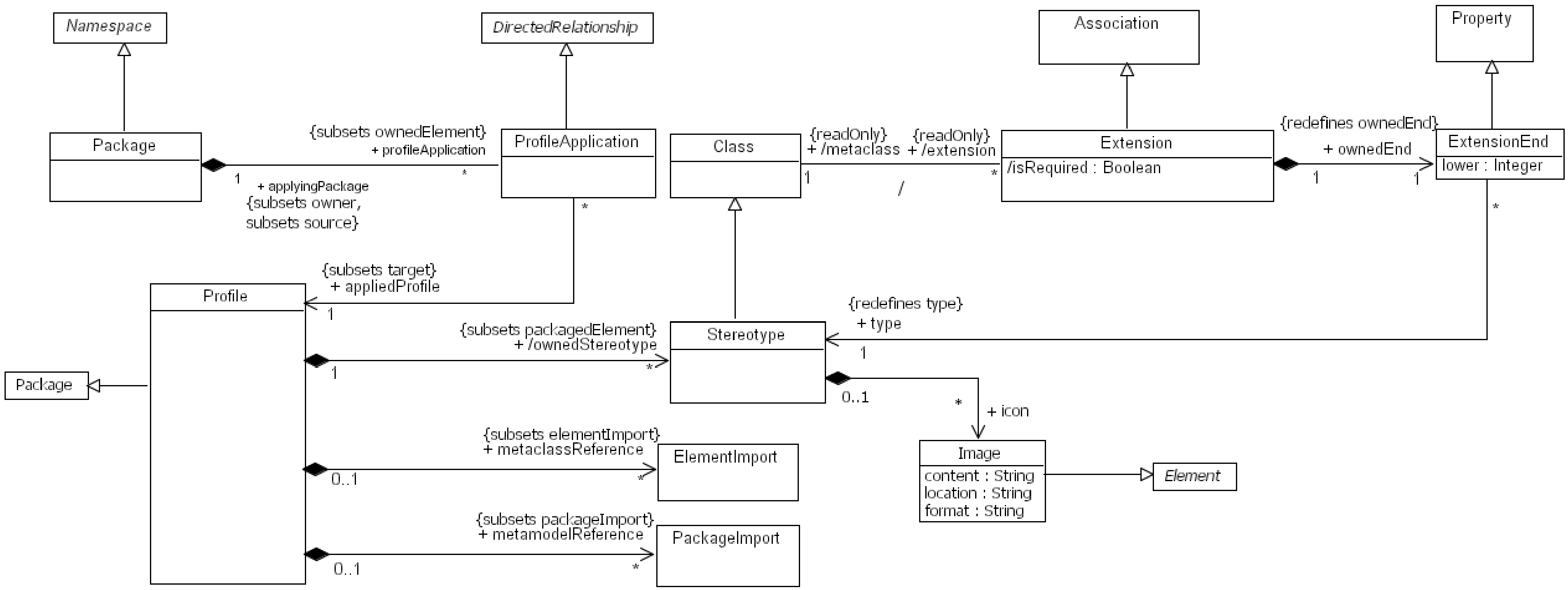
Model Libraries

- **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**

Example: Model Library



The UML Profile Metamodel



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

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 confliciting meta-associations
    - These too can be eliminated sometimes by constraints

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**

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)

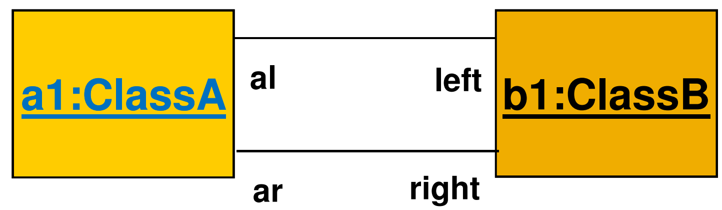
Lecture Overview

- About Model-Based Engineering (MBE)
- A Short Primer on Modeling Language Design
- The Unified Modeling Language
  - Semantics
  - UML as a DSL tool
- UML as an architectural description language

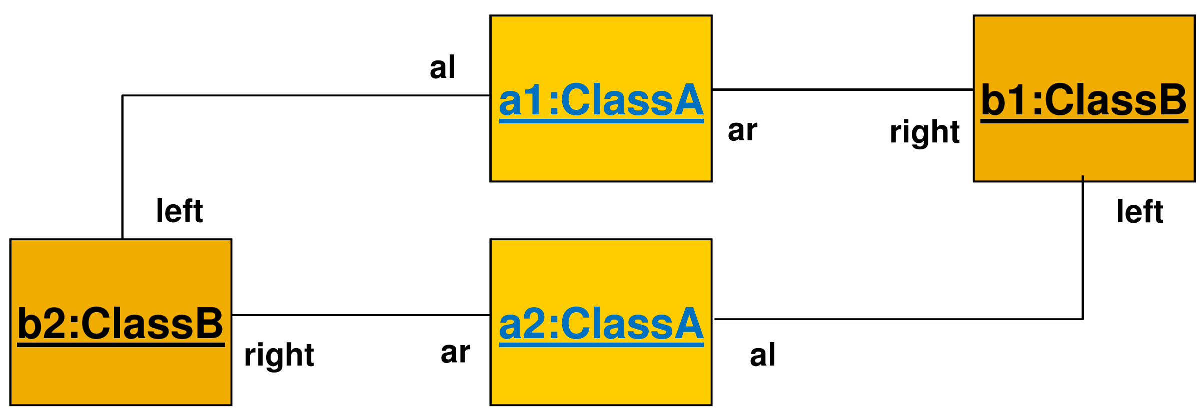
Class Specifications and Run-Time

Q: How many different run-time configurations are described by this class diagram?

(1)



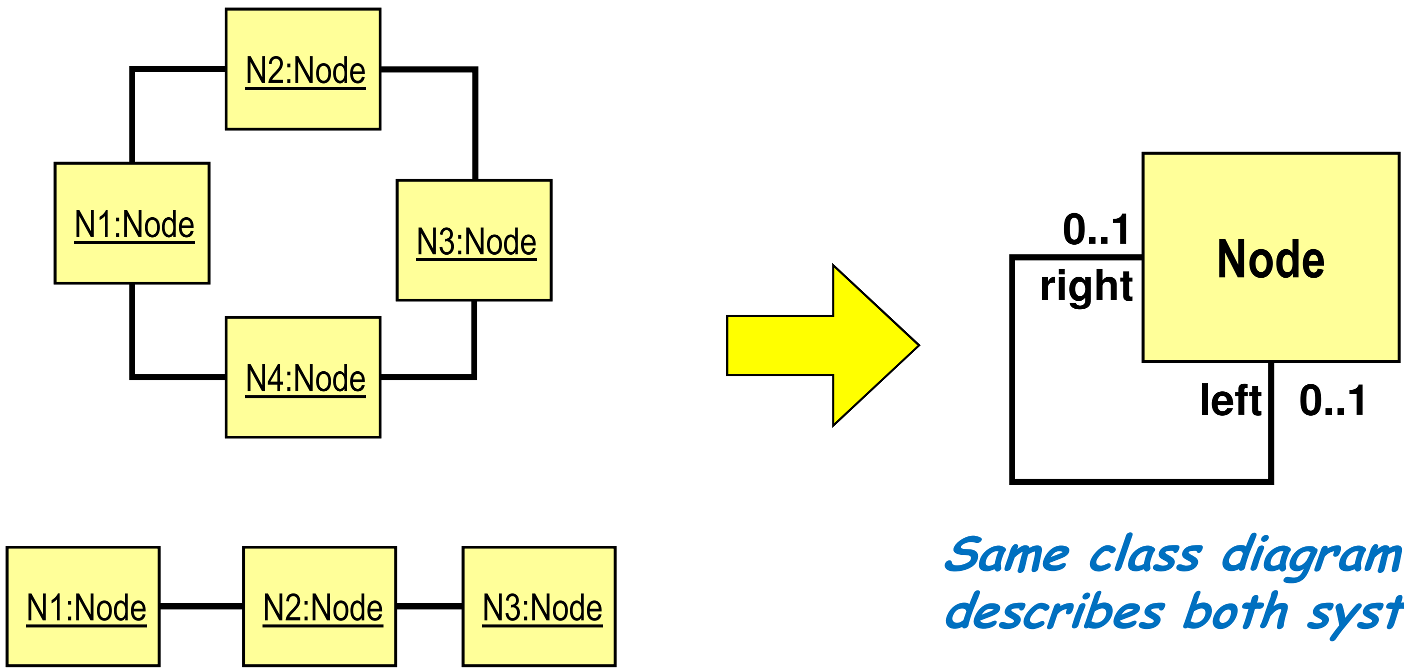
(2)



(3) **etc.**

Modeling Software Structures

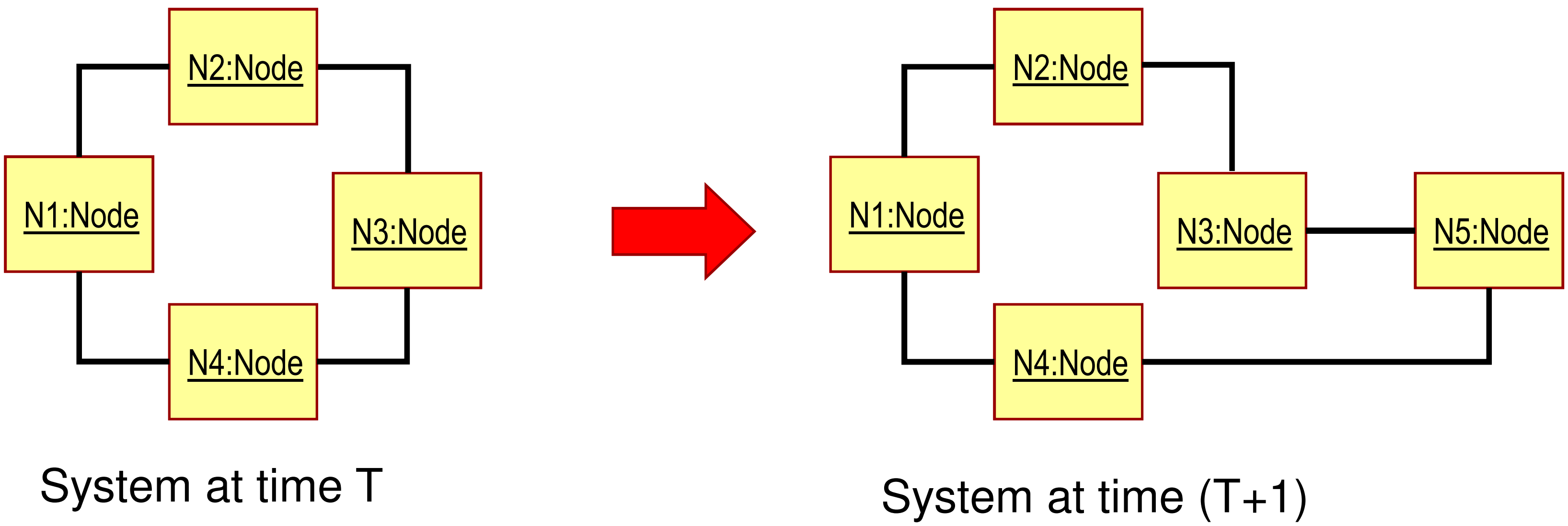
- Class diagrams are not always sufficient for precise representation of run-time structures
- Some structures need to be represented at the instance level



Same class diagram describes both systems!

Object Diagrams Perhaps?

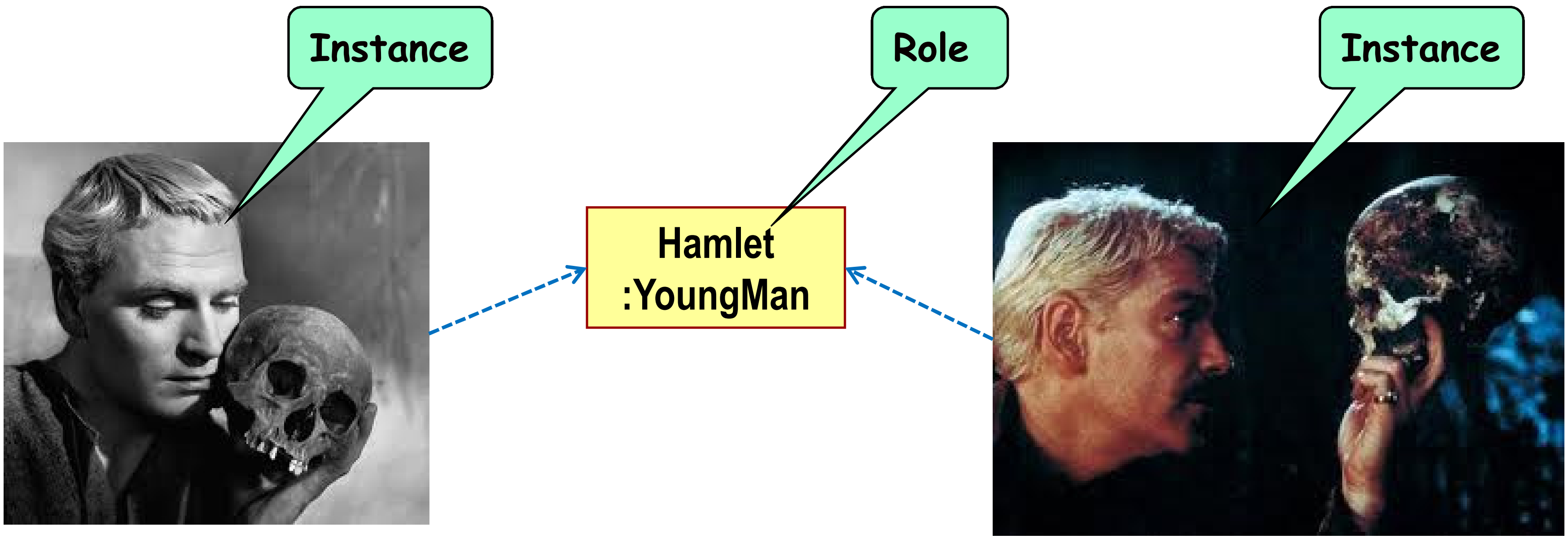
- **Not necessarily…**
  - Object diagrams represent "snapshots" of some specific system at some point in time
  - They can only serve as examples and not as general architectural specifications (unless we define a profile)



- **Need a way of talking about "prototypical" instances across time**

Roles: Prototypical Instances

- **<u>Role</u>: representation of an instance with a specific responsibility in some broader generalized context (<u>collaboration</u>)**
  - E.g., the role of Hamlet in the play "Hamlet"
  - Can be filled by different actual individuals (instances)

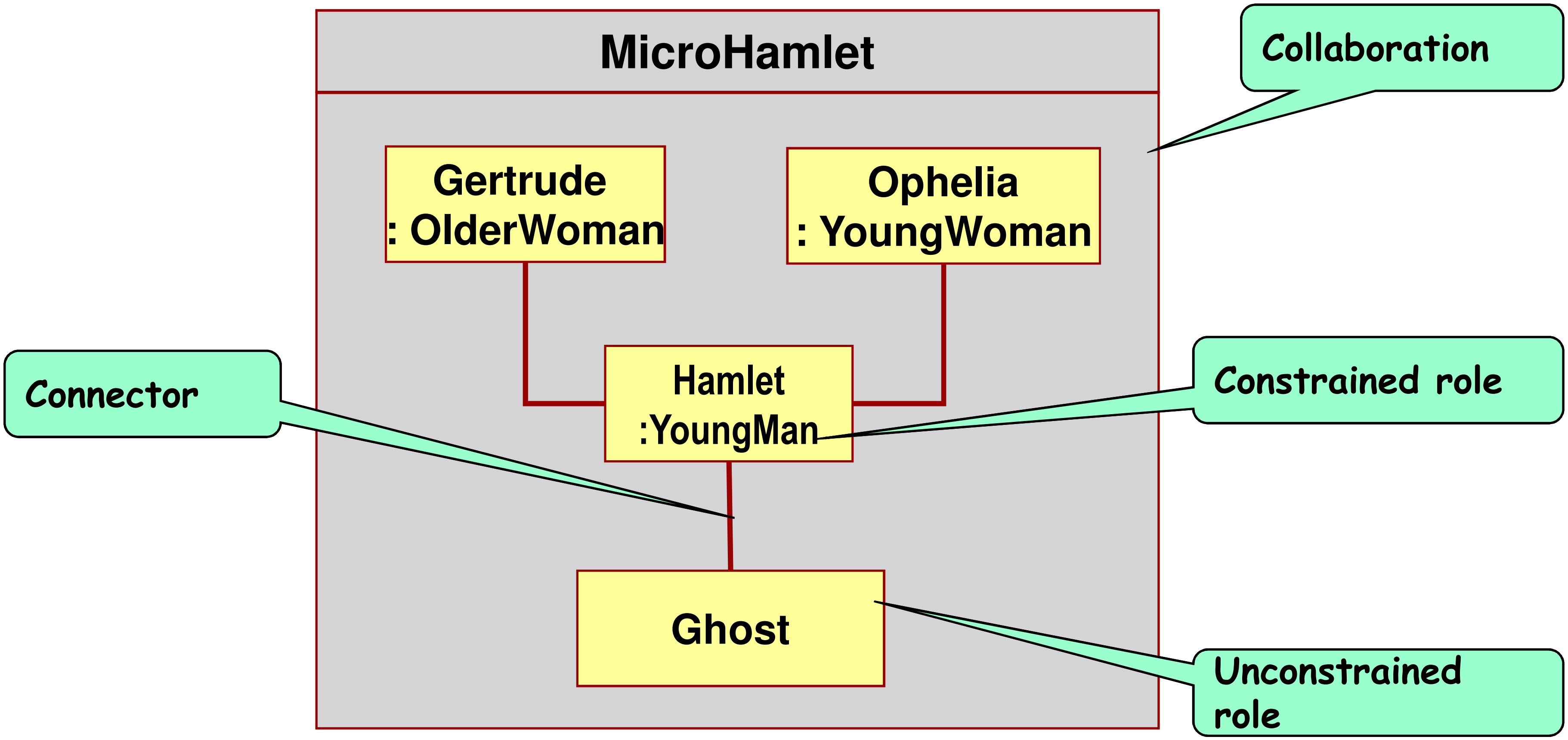


Laurence Olivier

Kenneth Branagh

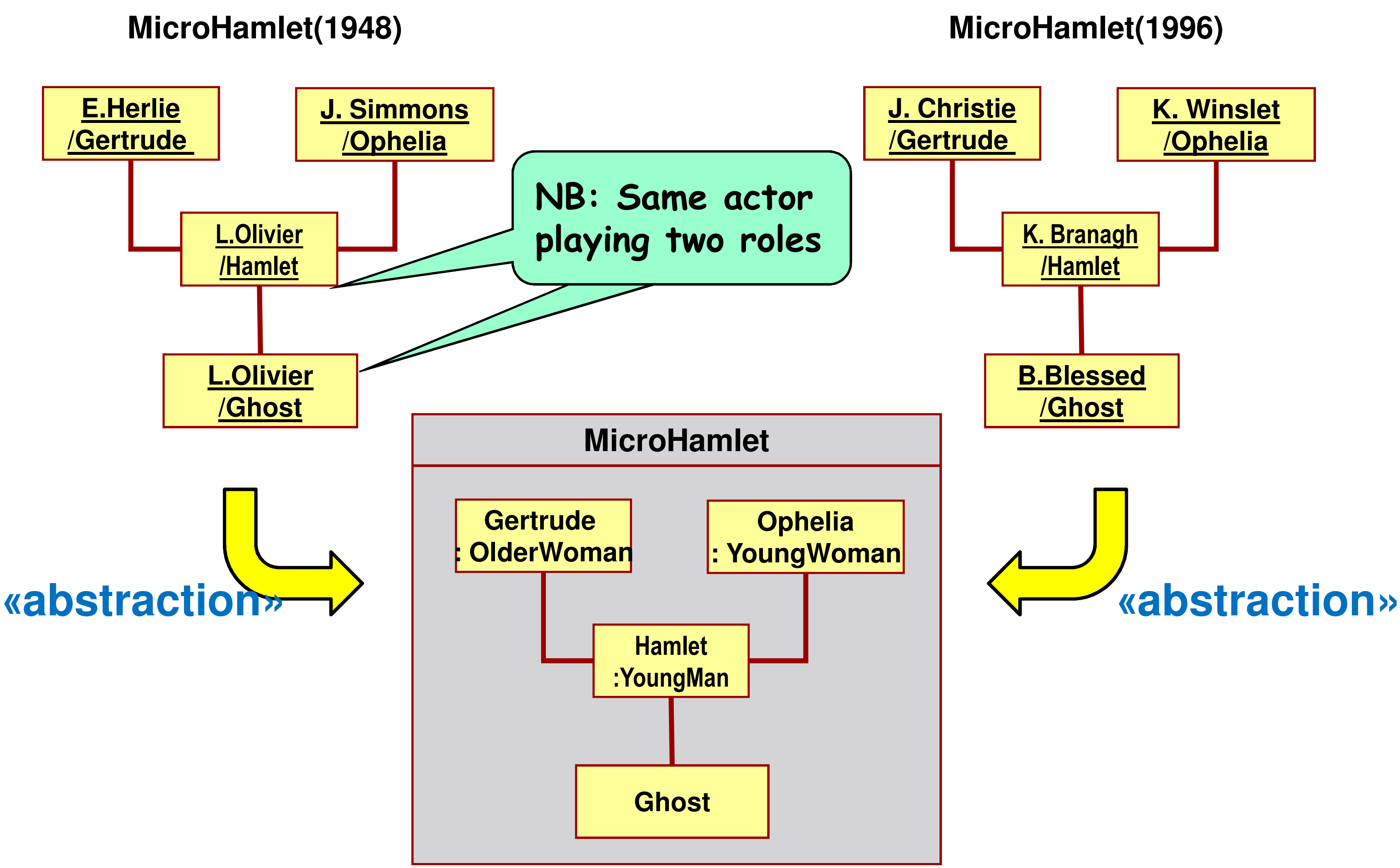
Collaborations

- Describes a set of roles communicating using connectors
- A role can represent an instance or something more abstract



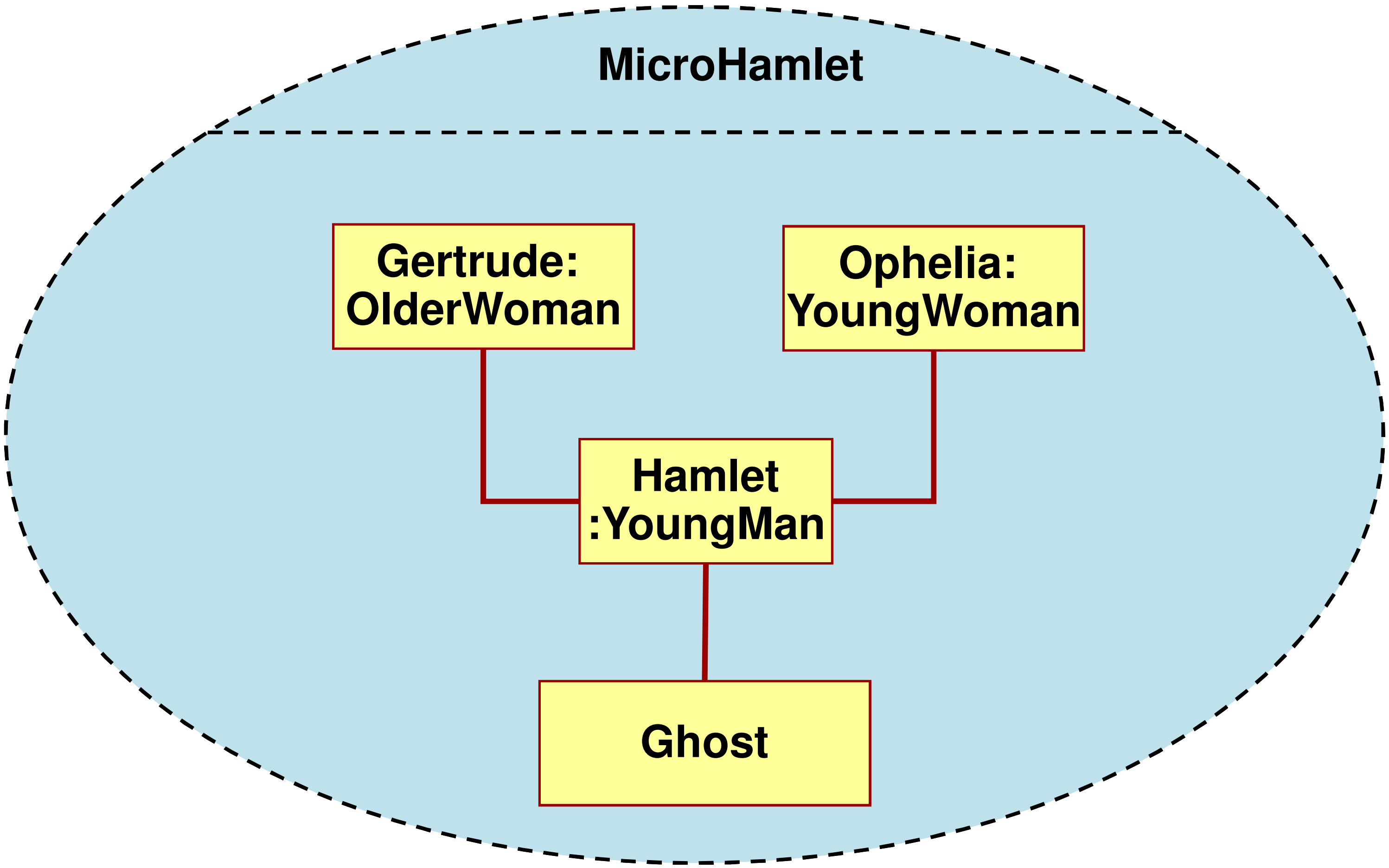
Collaborations and Roles

- Collaborations represent a network of cooperating object instances whose identities have been abstracted away (roles)



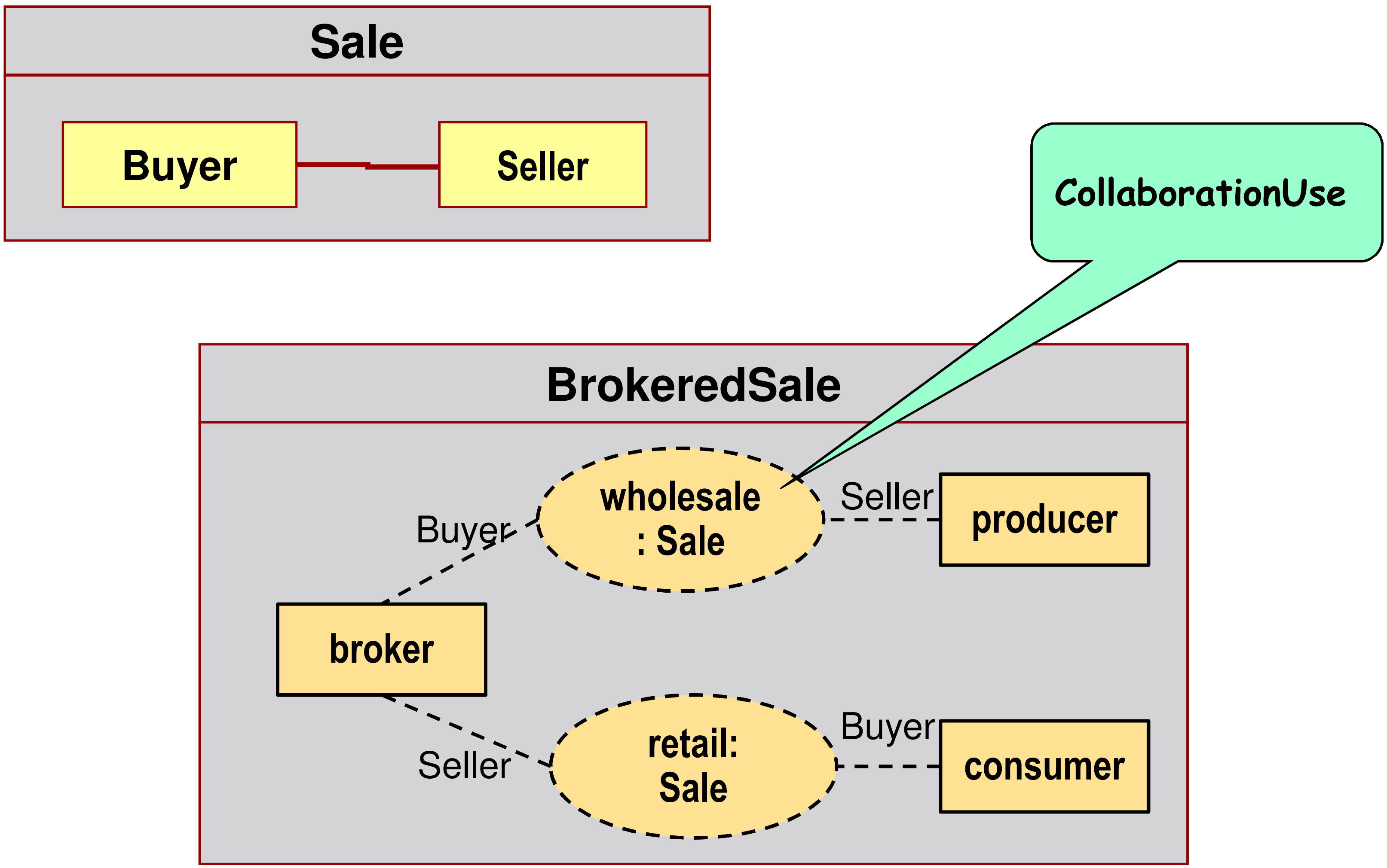
Alternative Notation

- **Common in textbooks – but not very practical**
  - Inefficient use of space; rectangle notation recommended



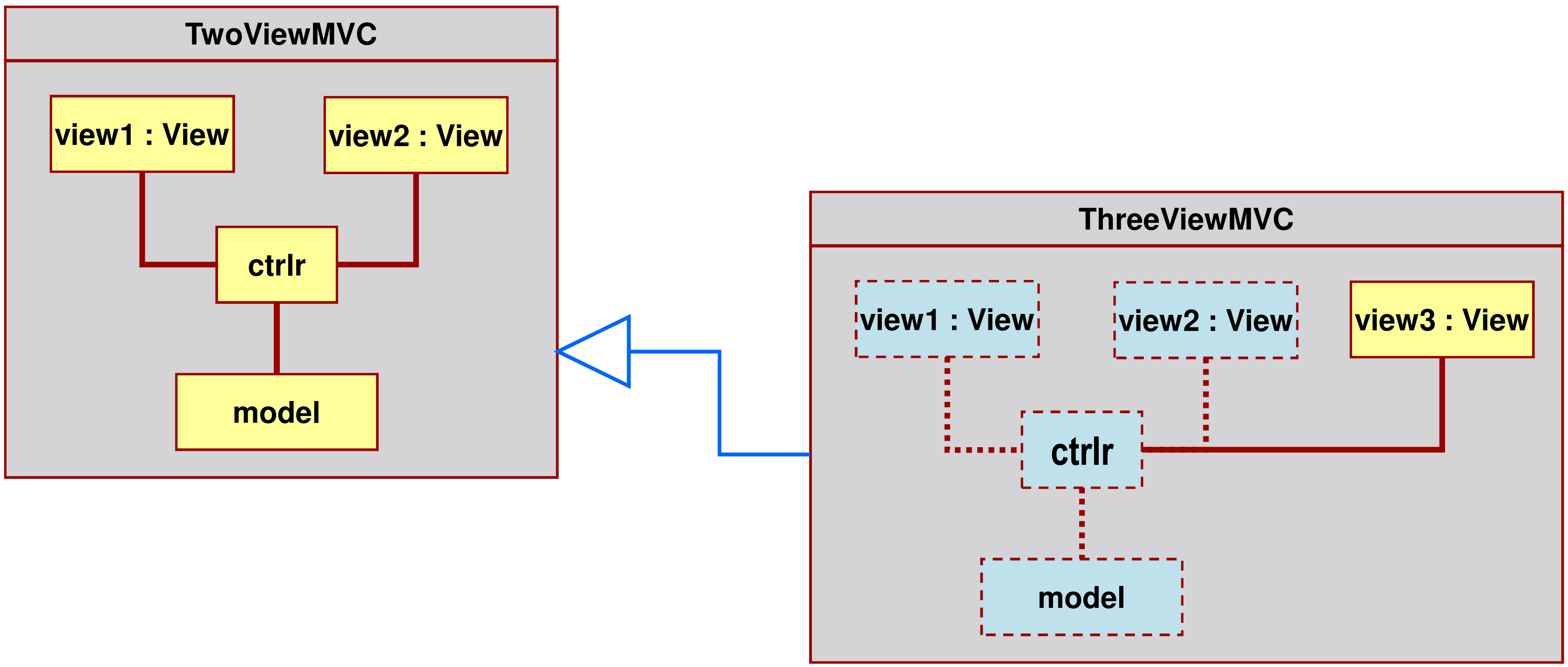
Collaboration Uses

- Applying a collaboration specification



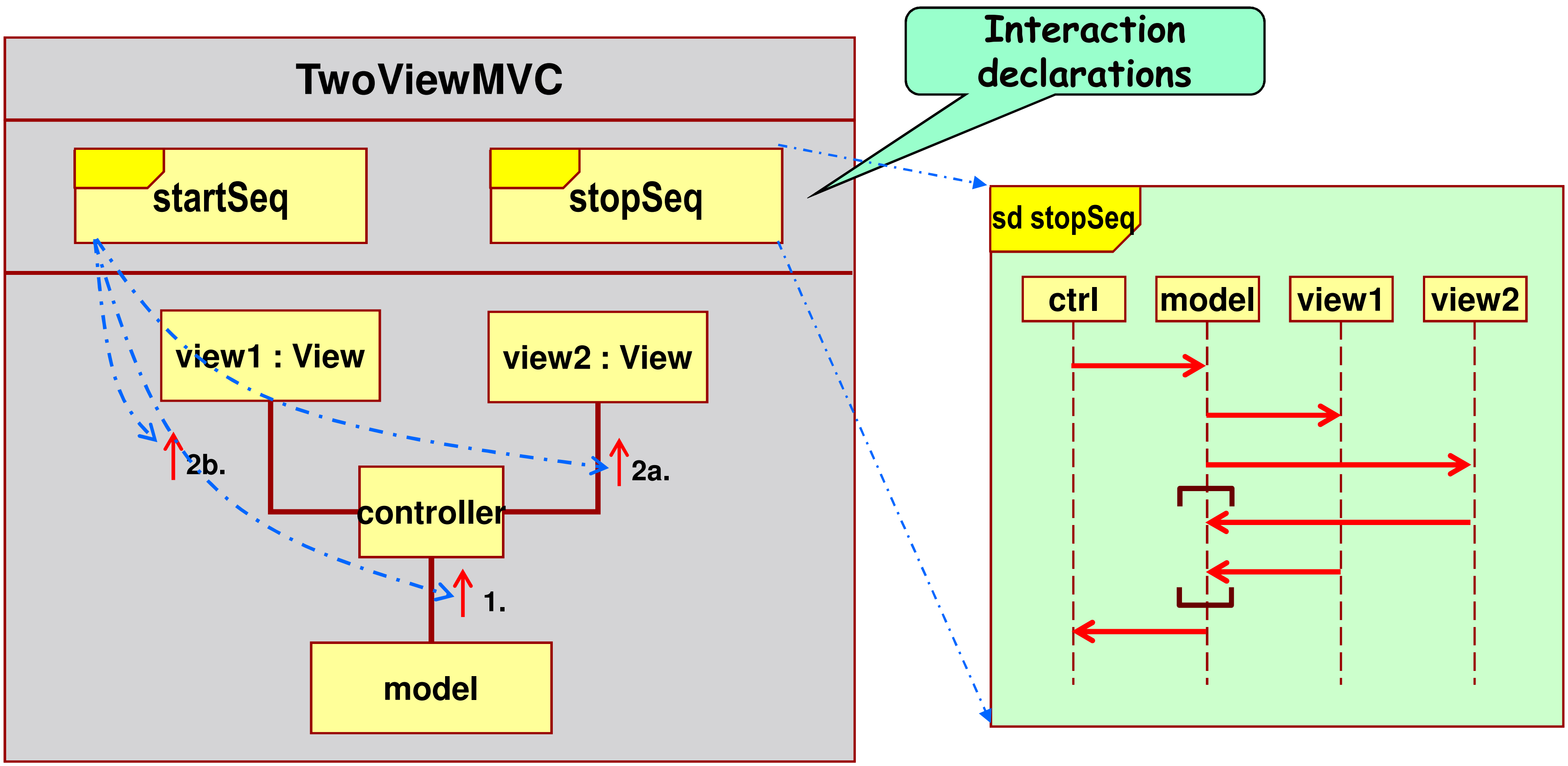
Collaborations are Classifiers

- Collaborations can be refined using inheritance
  - Possibility for defining generic architectural structures



Collaborations and Behavior

- **One or more behavior specs can be attached to a collaboration**
  - To show interesting interaction sequences within the collaboration

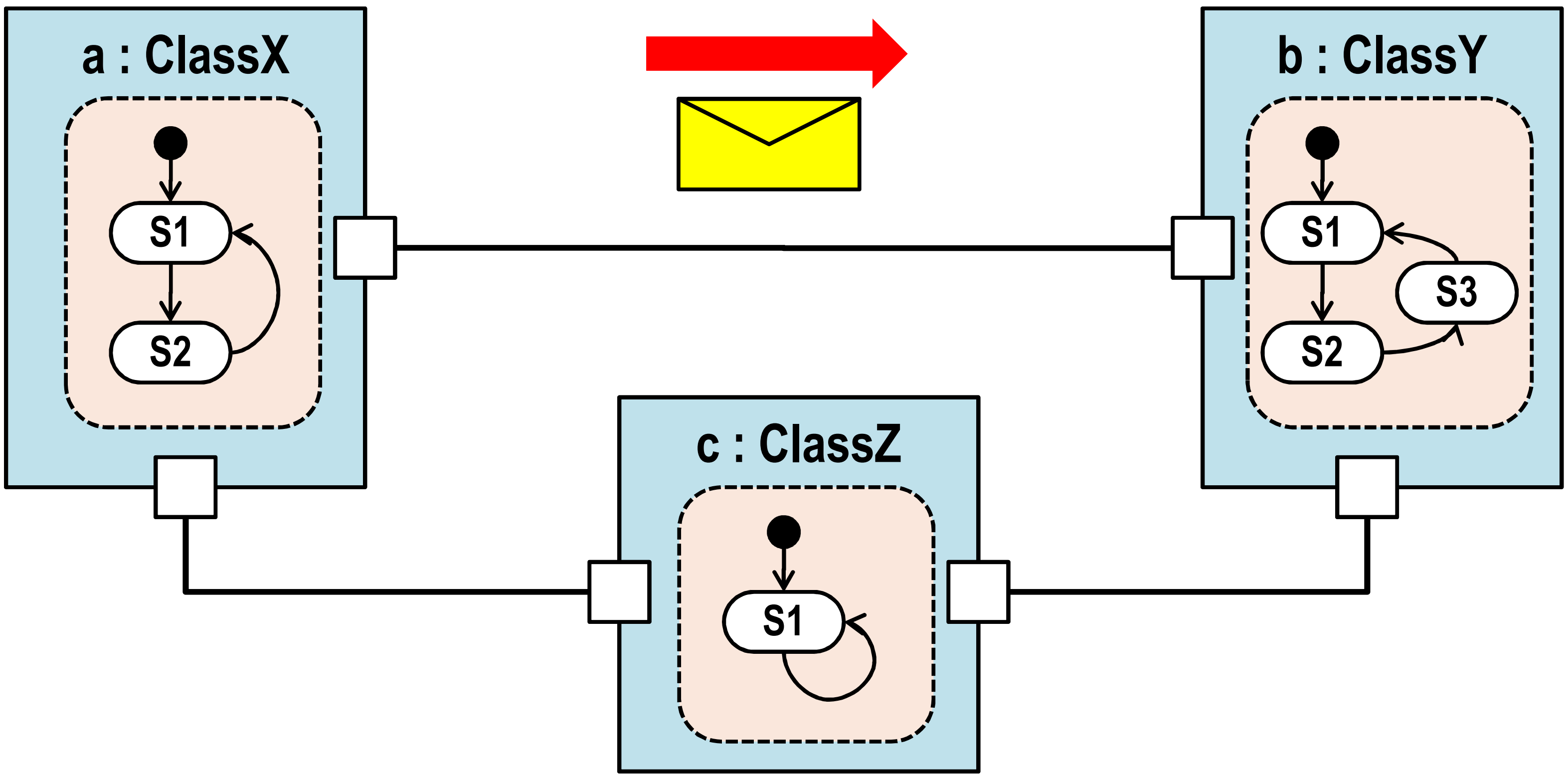


Structured Classes

- Classes with
  - External structure (port interaction points)
  - Internal (collaboration) structure
- Primarily intended for architectural modeling
- Heritage: architectural description languages (ADLs)
  - UML-RT profile: Selic and Rumbaugh (1998)
  - ACME: Garlan et al.
  - SDL (ITU-T standard Z.100)

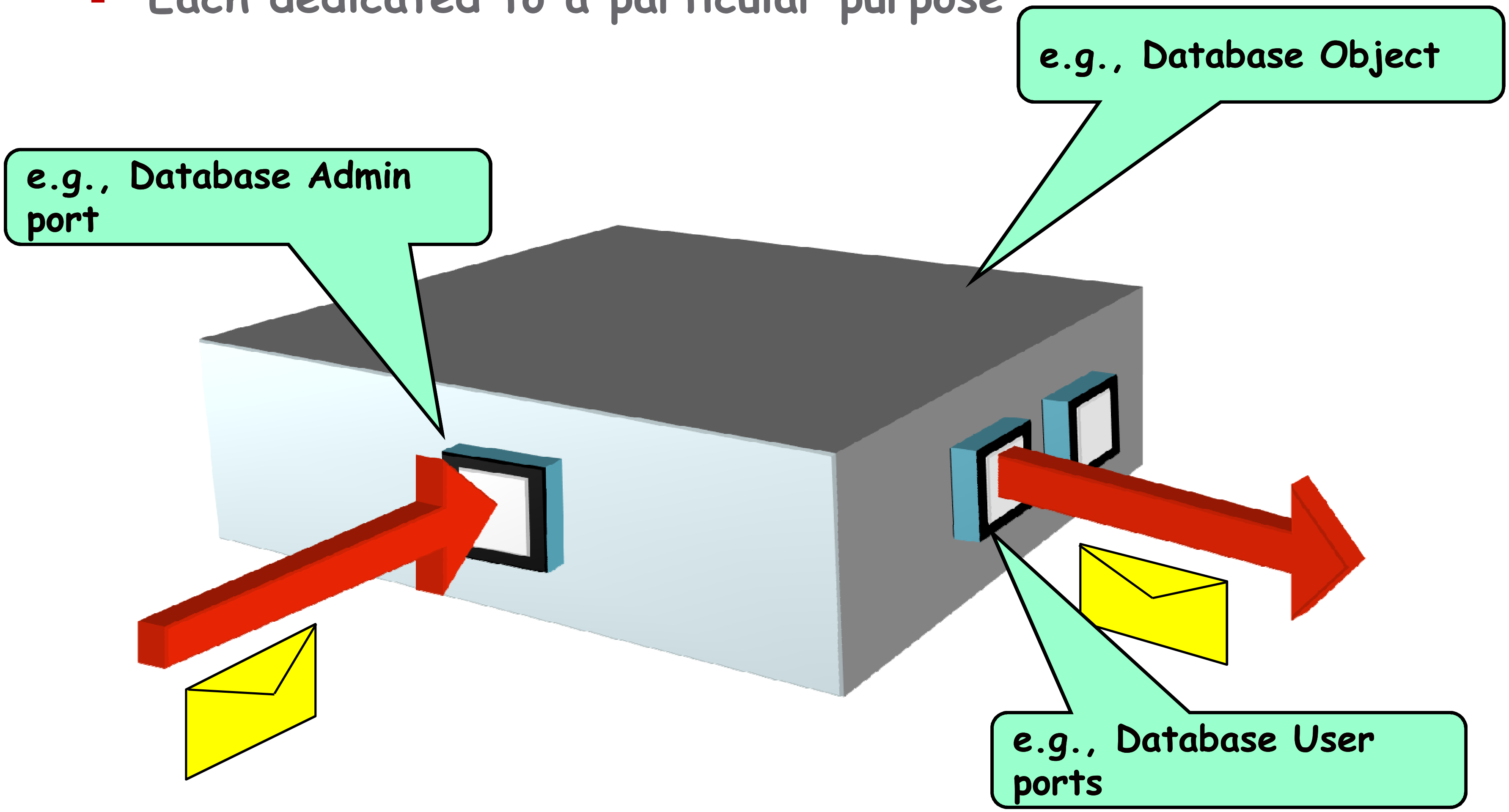
Basic Model of Computation

- Networks (collaborations) of specialized objects that realize a system's functionality by executing their behaviours triggered by the arrival of incoming messages



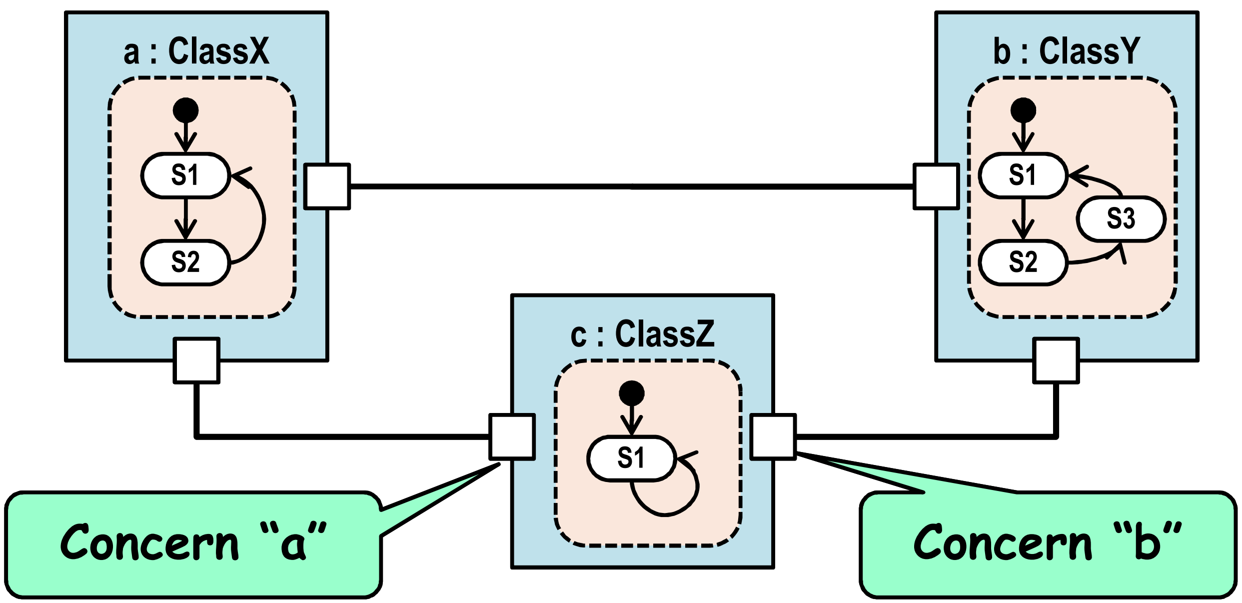
Structured Objects: Ports

- **Multiple points of interaction**
  - Each dedicated to a particular purpose



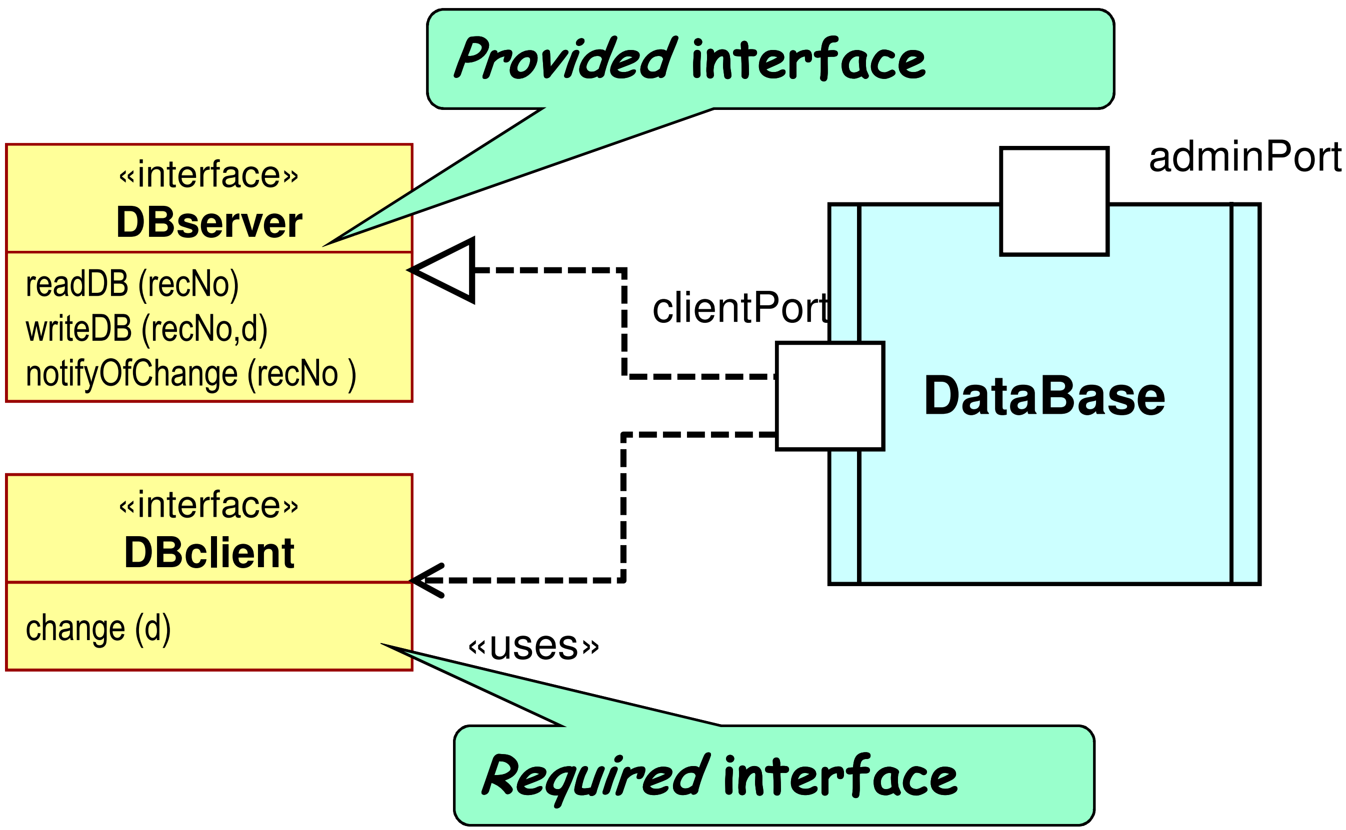
The Primary Rationale for Ports

- Full decoupling between the internals of an object (i.e., its implementation) and its environment
  - Greatly increases reusability potential of components
- Separation of multiple collaborations of an object
  - To support: safety, security, separation of concerns (aspects)

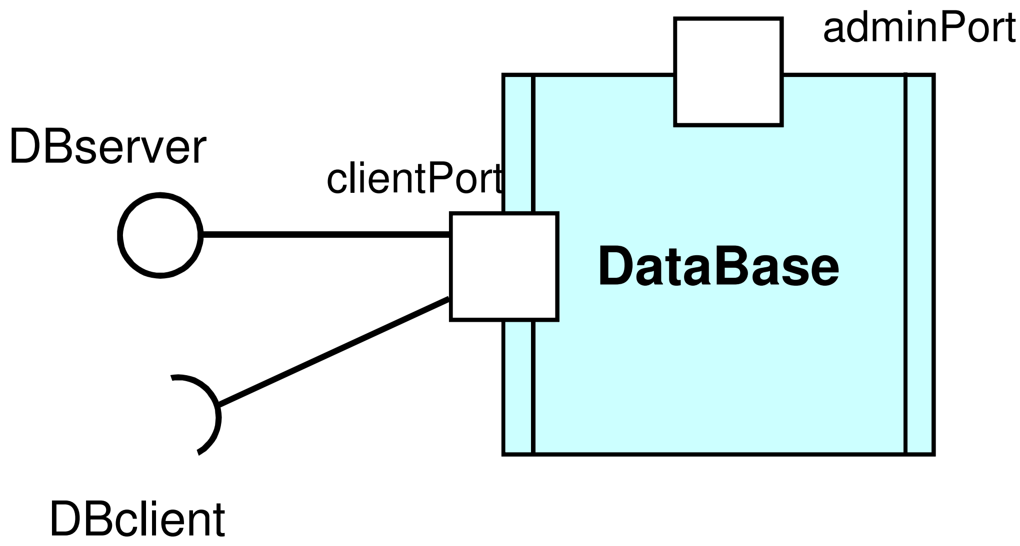


Ports and Interfaces

- **Ports can provide and/or require Interfaces**
  - General case: both required and provided
  - Uni-directional ports are also common

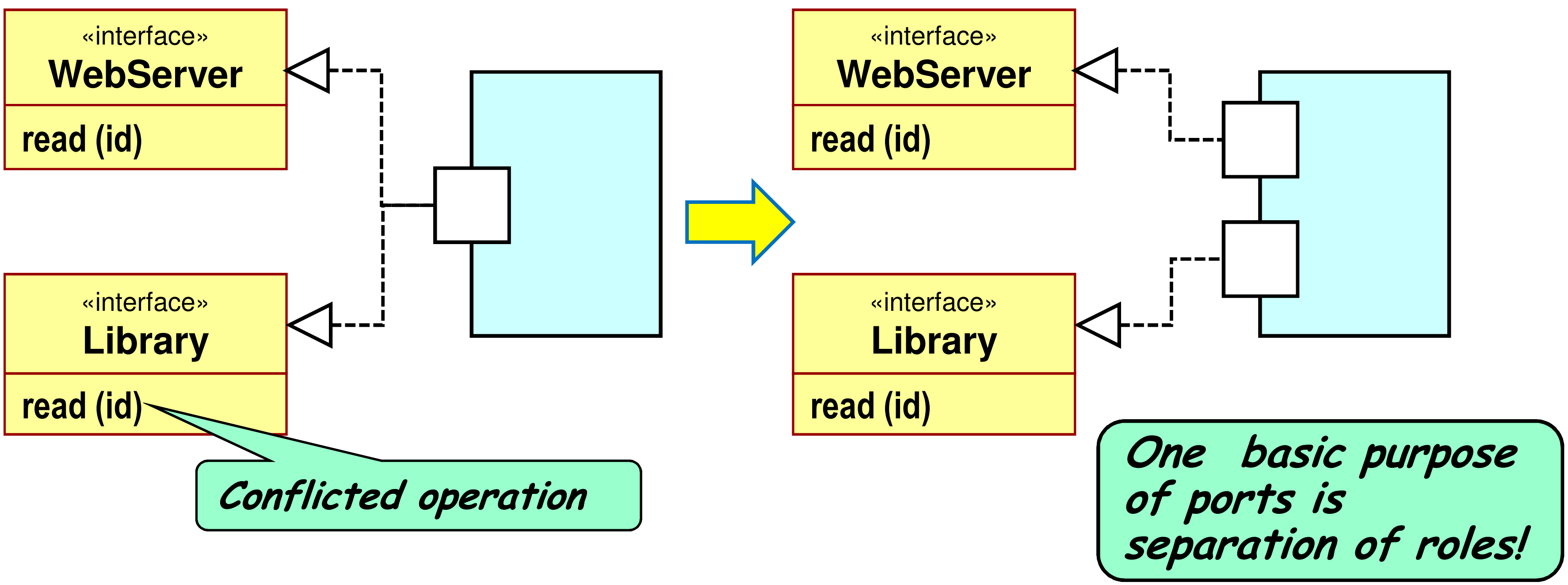


Alternative Shorthand Notation



Multiple Interfaces per Port?

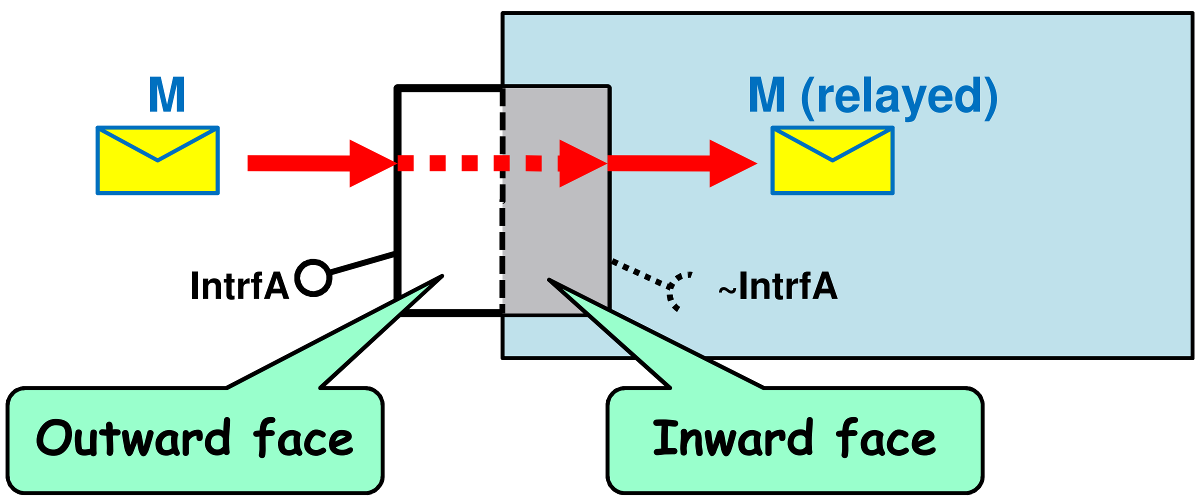
- **Possible: but should be avoided**
- **Because:**
  - Creates potential conflicts (name clashes) and requires some kind of disambiguation mechanism
  - Unclear semantics if interfaces have protocol state machines attached
  - Much less problematic to simply define separate ports for each interface



The Essential Nature of Ports

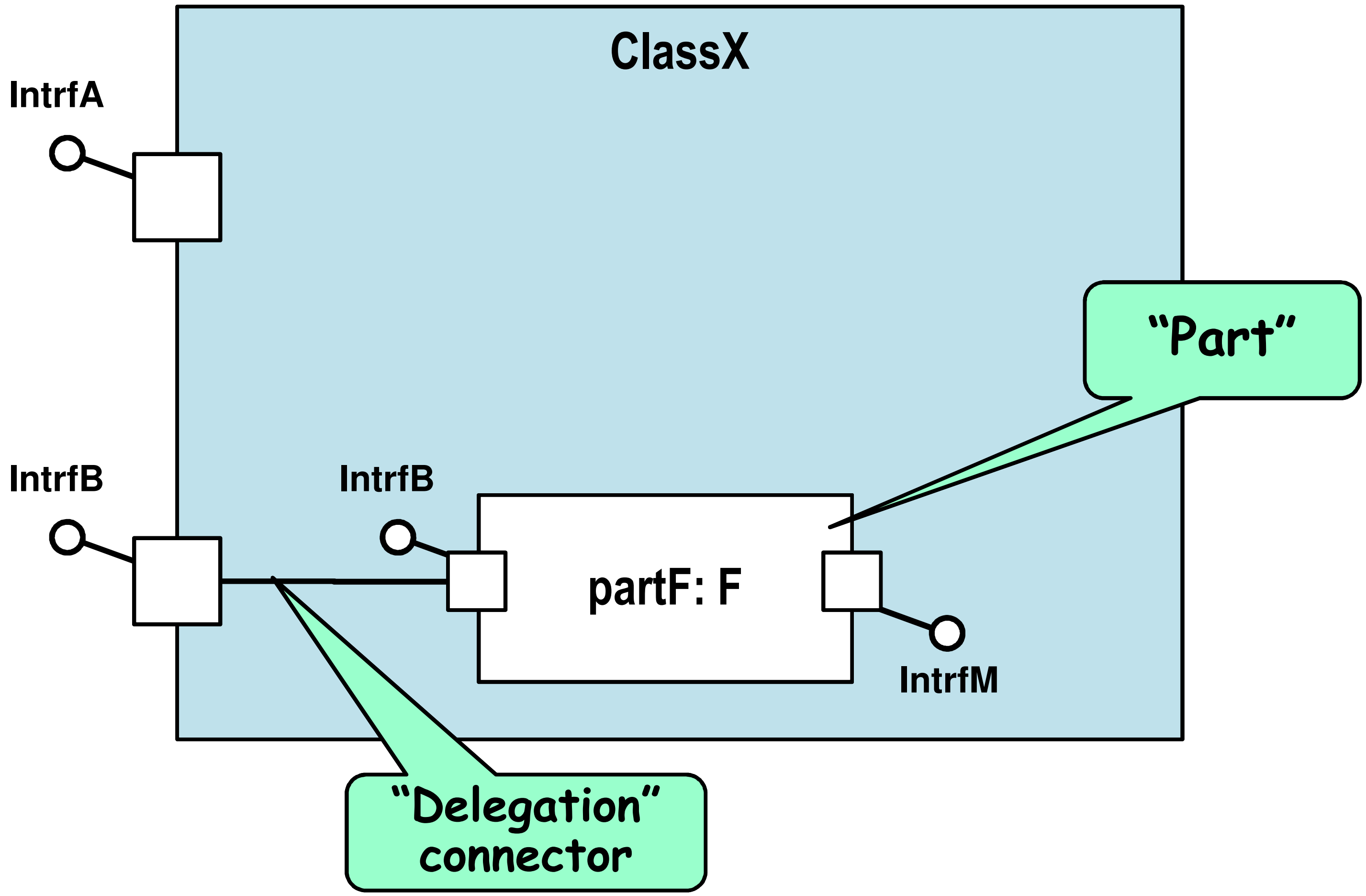
- **Ports sit on the boundary of an object and act as a *message relay* between the inside and outside**
  - Whatever message arrives on the outside is relayed inwards
  - ...and vice versa

⇒ **Conceptually, there are two faces to every port (*inward* and *outward*)**
  - The two are complements (conjugates) of each other
  - The inward face is implicit (i.e., we do not declare it)



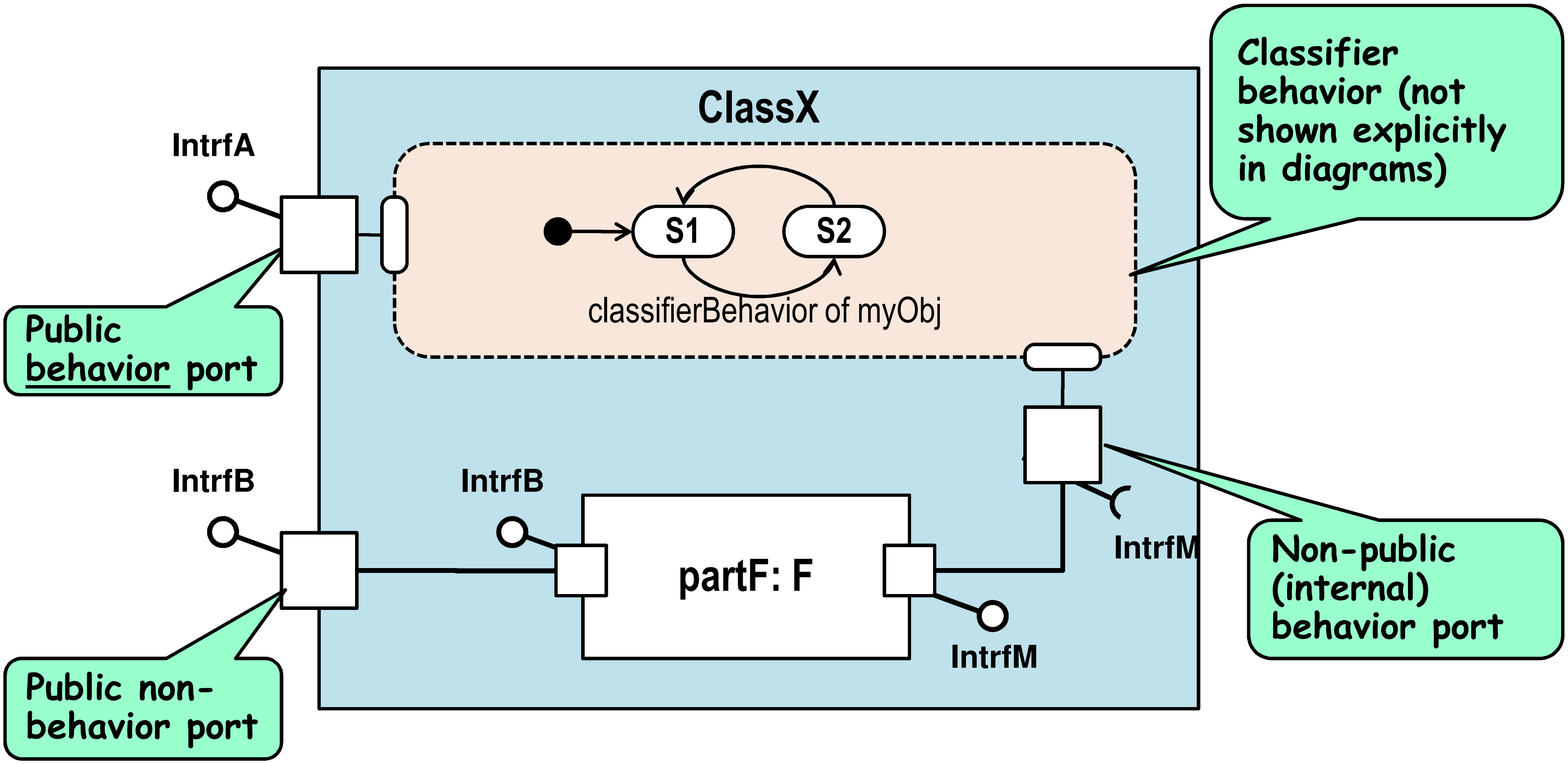
Internal Structures

- **Structured classes can contain collaboration structures comprising <u>parts</u> that are usages of other structured (or basic) classes**



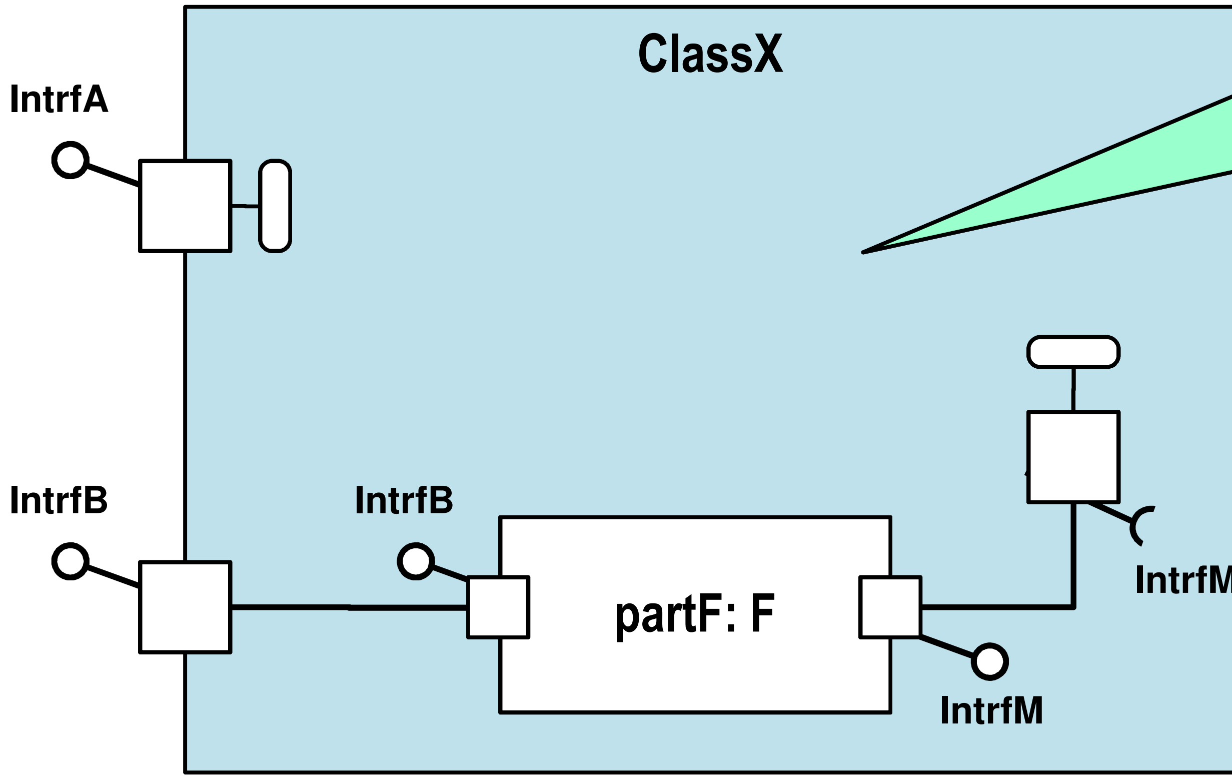
Ports and Behaviours

- Behavior ports: ports that are connected to the classifier behaviour of an object



Ports and Behaviours

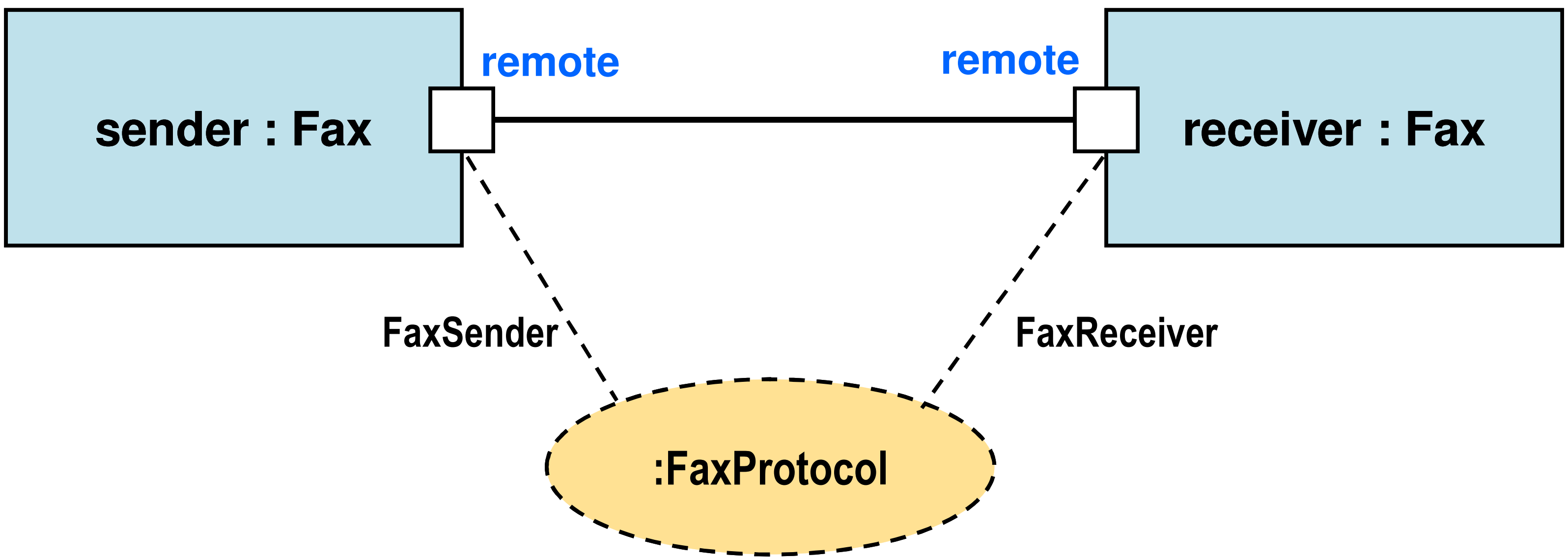
- Behavior ports: ports that are connected to the classifier behaviour of an object



Actual notation does not show the classifier behavior ⇒ implied by behavior ports

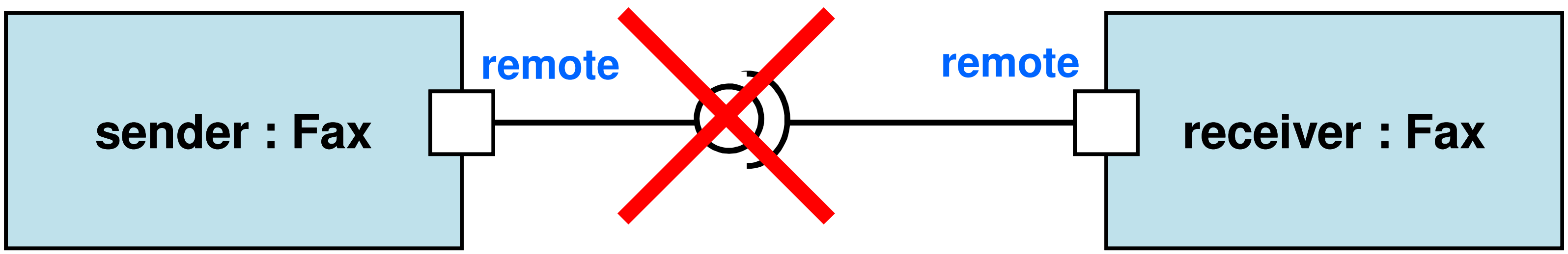
Assembling Structured Objects

- Ports can be joined by connectors
- These connections can be constrained to a protocol
  - Static checks for dynamic type violations are possible
  - Eliminates "integration" (architectural) errors



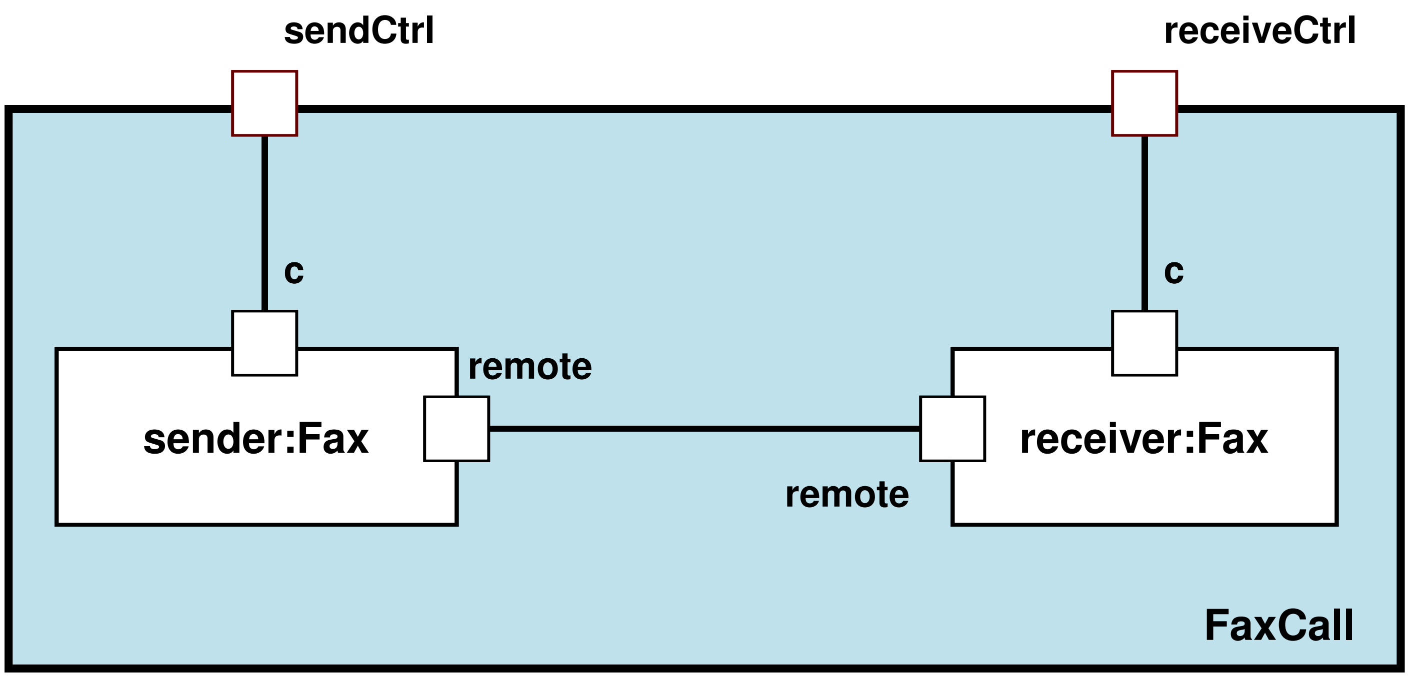
A Comment on Notation

- **Avoid the ball-in-socket notation for connectors**
  - No added meaning relative to a simple line
  - Creates the false impression that the connector connects only particular interface types



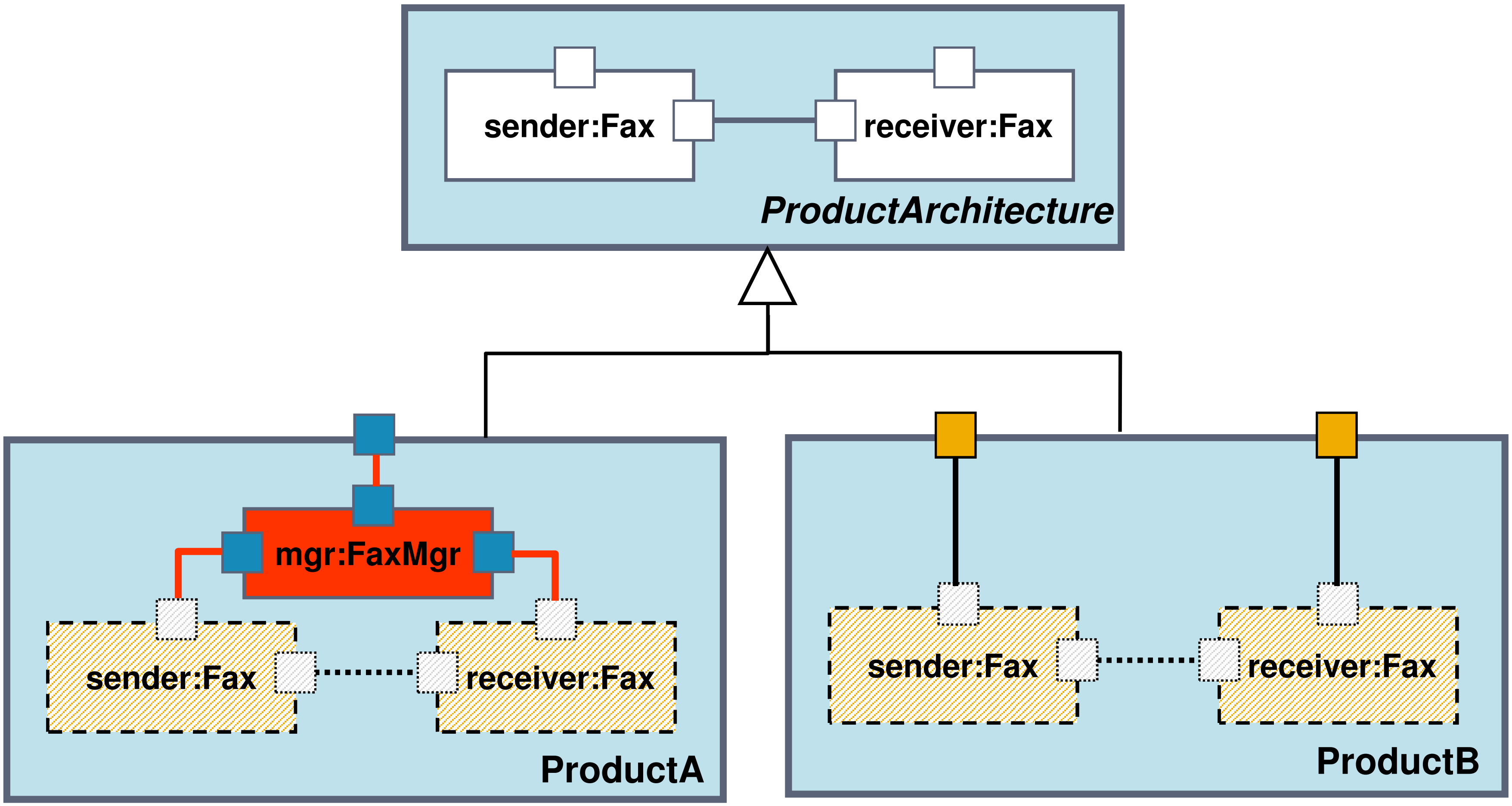
Using Structured Classes

- Structured classes can be used to capture and complex architectural structures as a unit
- Which can be created and destroyed as a unit



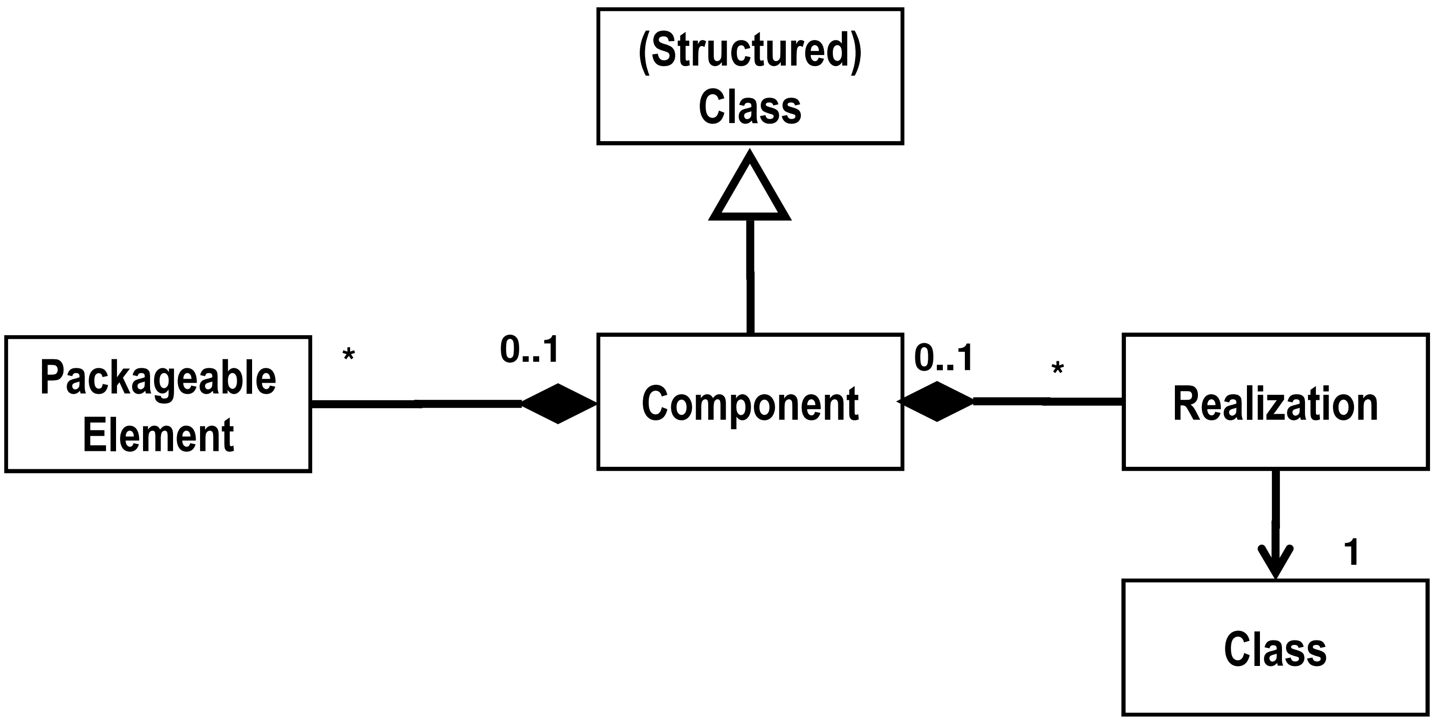
Structure Refinement Through Specialization

- **Reuse at the architectural level**
  - Useful for representing product families



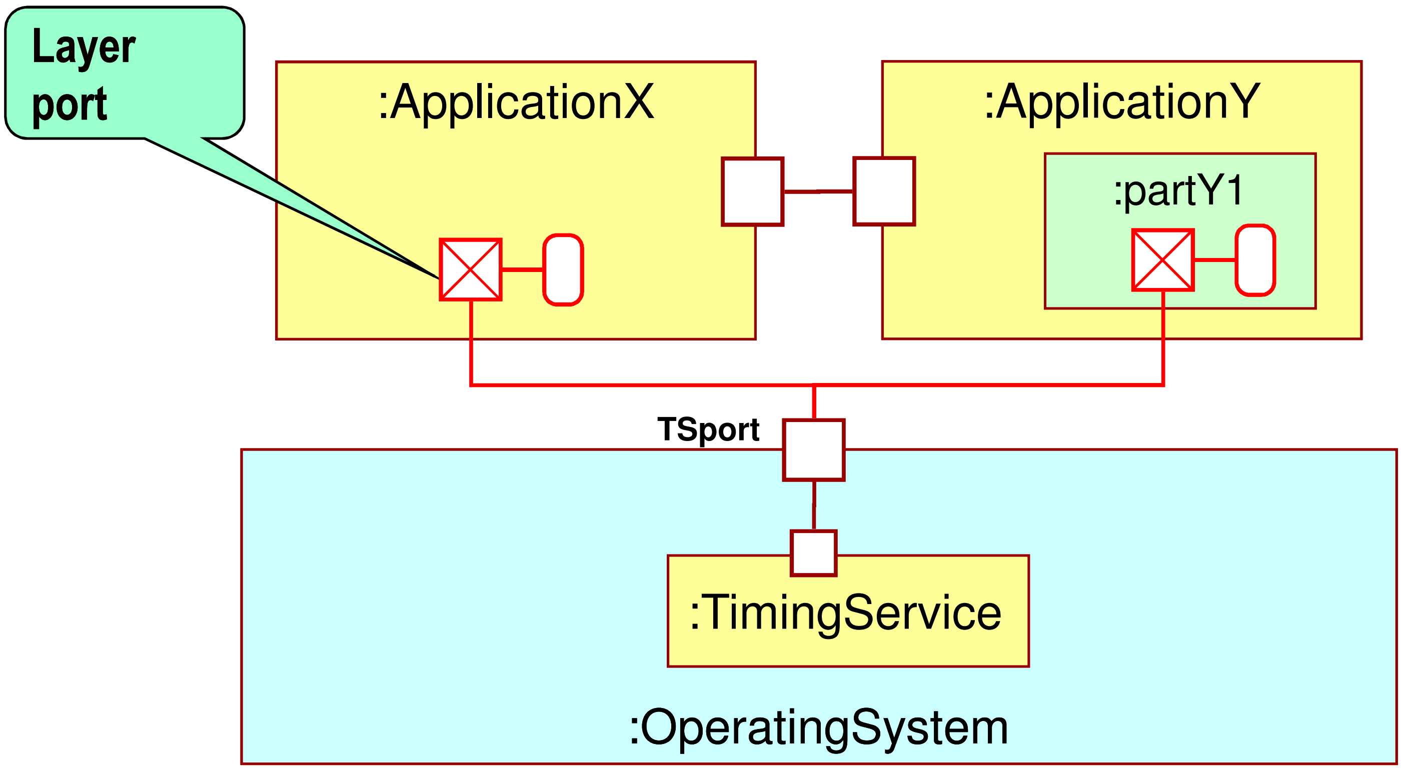
Components

- **A kind of structured class whose specification**
  - May be realized by one or more implementation classes
  - May include any other kind of packageable element (e.g., various kinds of classifiers, constraints, packages, etc.)



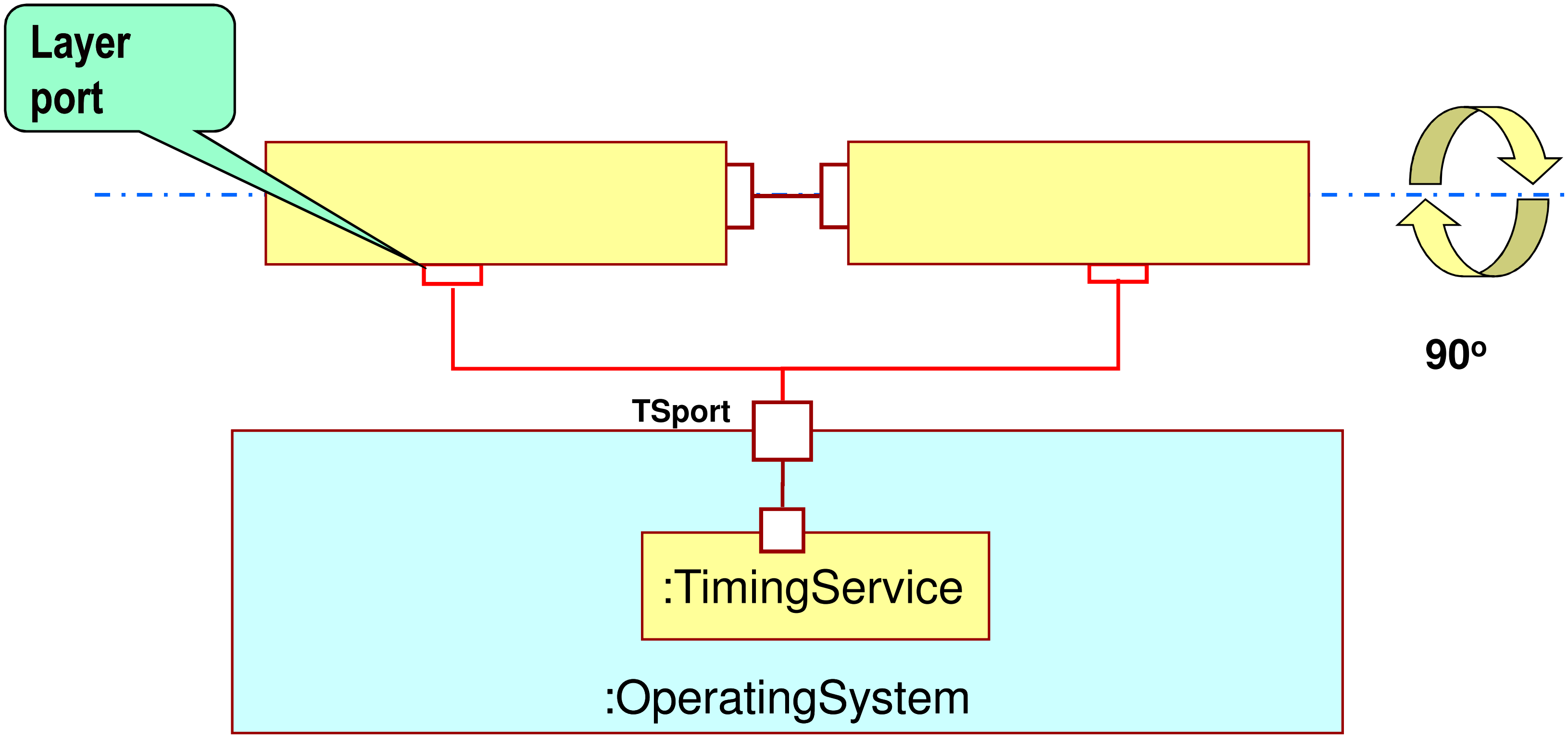
Modeling Layers in UML

- Layer ports (SAPs) are represented by a port whose "isService" attribute is set to FALSE



Modeling Layers (cont.)

- The layer ports are in a different "dimension" than peer ports



Summary: Complex Structure Modeling

- UML has added the notion of structured classifiers primarily for architectural modeling
- Two basic types
  - Collaborations
  - Structured classes (and components)
- Collaborations are used to capture patterns of co-operating objects
  - Roles, connectors, etc.
  - No encapsulation shell – cannot be instantiated as objects
- Structured classes are mostly used for architectural components with
  - An external structure (ports)
  - An internal structure (parts, connectors, etc.)
  - Can be created and destroyed as a unit

Conclusions

- **UML 2 has added some important modeling capabilities**
  - Support for architectural specification
- **UML is gradually evolving into a "modern" modeling language**
  - An "implementation" language with precisely defined semantics (fUML)
- **Intended to support both descriptive and prescriptive uses**
  - Facilitates "agile" development
- **Capable of supporting towards domain-specific languages via the profile mechanism**
  - ...within intended design limits
  - Part of the original design intent
  - Approach has major technical advantages (avoids integration problem)
  - A better theoretical understanding and foundation of profile mechanism is needed

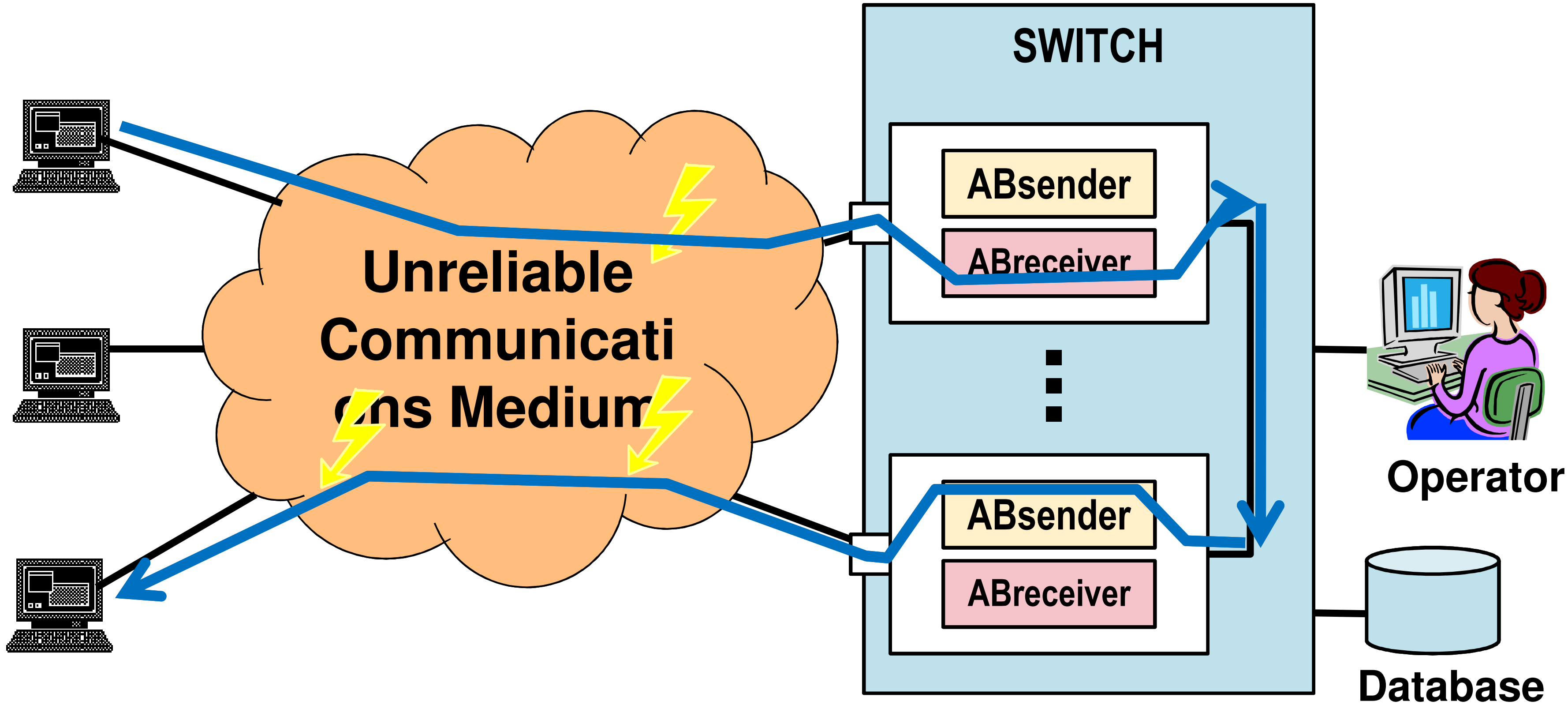
Bibliography/References

- D. Milicev, "Model-Driven Design with Executable UML", WROX, 2009.
- A. Kleppe, "Software Language Engineering", Addison-Wesley, 2009
- T. Clark et al., "Applied Metamodeling – A Foundation for Language Driven Development", (2nd Edition), Ceteva, http://www.eis.mdx.ac.uk/staffpages/tonyclark/Papers/
- S. Kelly and J.-P. Tolvanen, "Domain-Specific Modeling: Enabling Full Code Generation," John Wiley & Sons, 2008
- J. Greenfield et al., "Software Factories", John Wiley & Sons, 2004
- D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of 'Semantics'", IEEE Computer, Oct. 2004.
- E. Seidewitz, "What Models Mean", IEEE Software, Sept./Oct. 2003.
- T. Kühne, "Matters of (Meta-)Modeling, Journal of Software and Systems Modeling, vol.5, no.4, December 2006.
- 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)

APPENDIX: A Worked Example of Architectural Design and Modeling

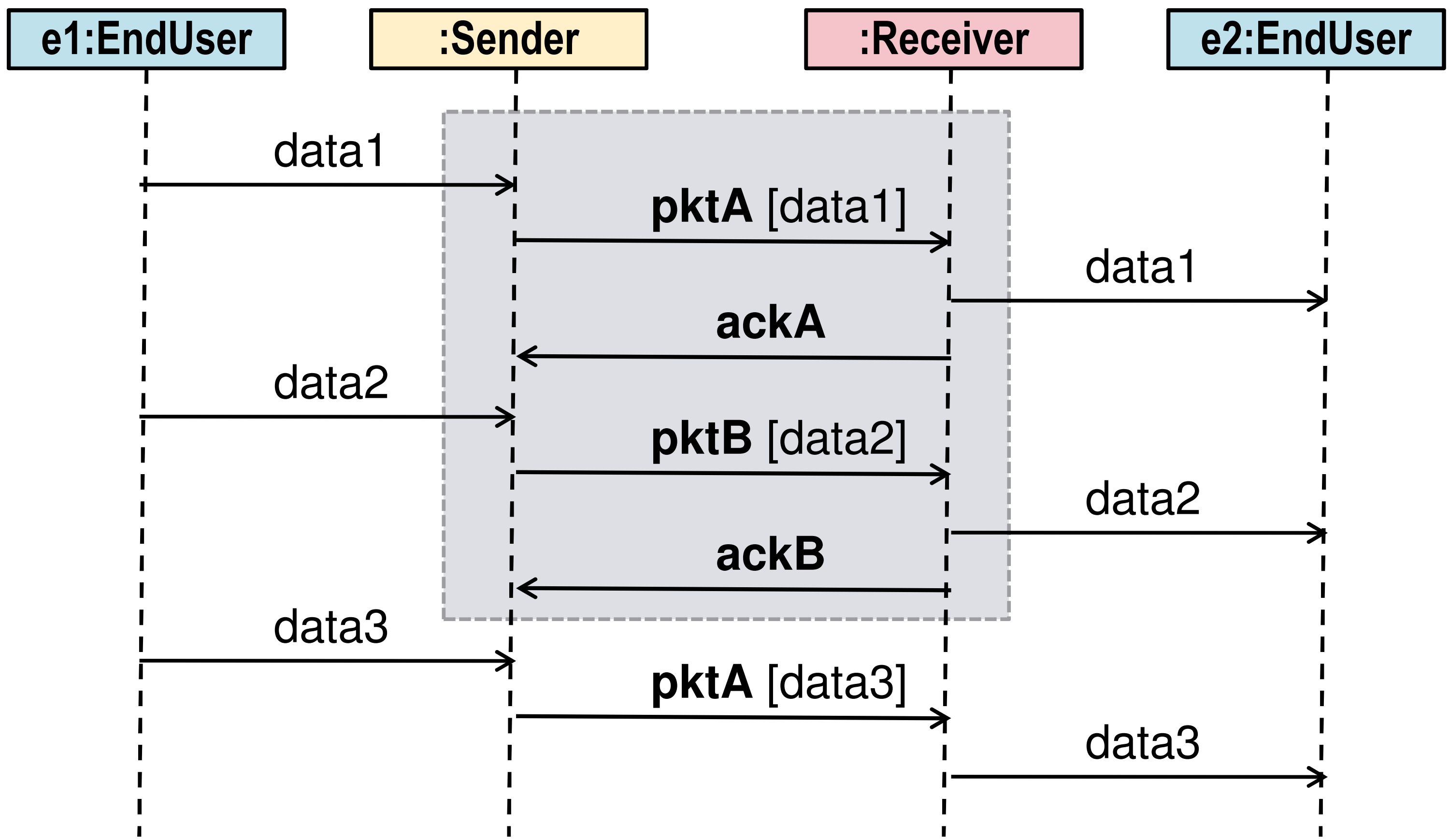
Example Problem

- Design the software architecture of data transmission switch that supports multiple users using the alternating-bit protocol



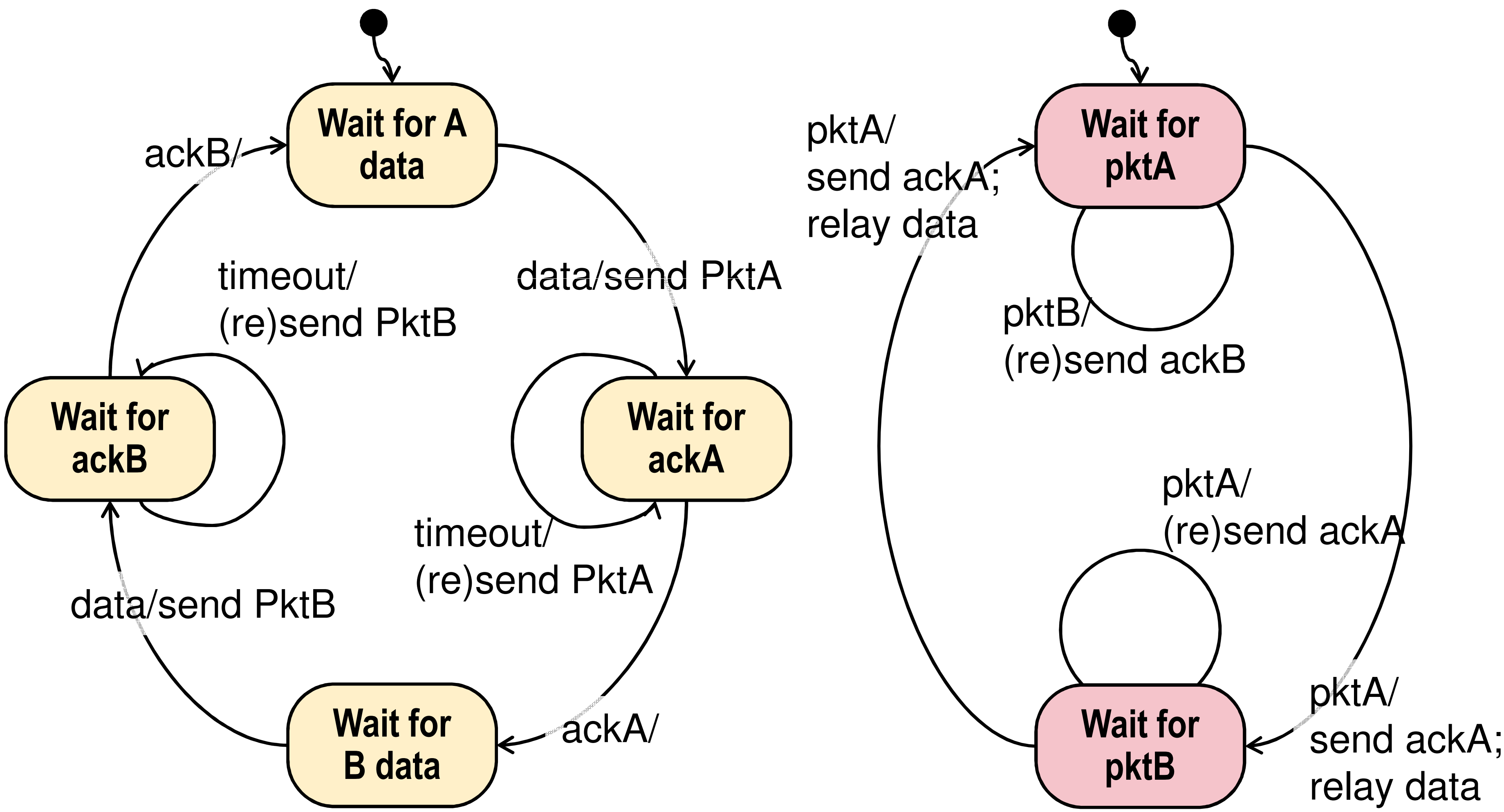
The Alternating Bit Protocol: Spec 1

- A simple positive acknowledgement protocol with retransmission

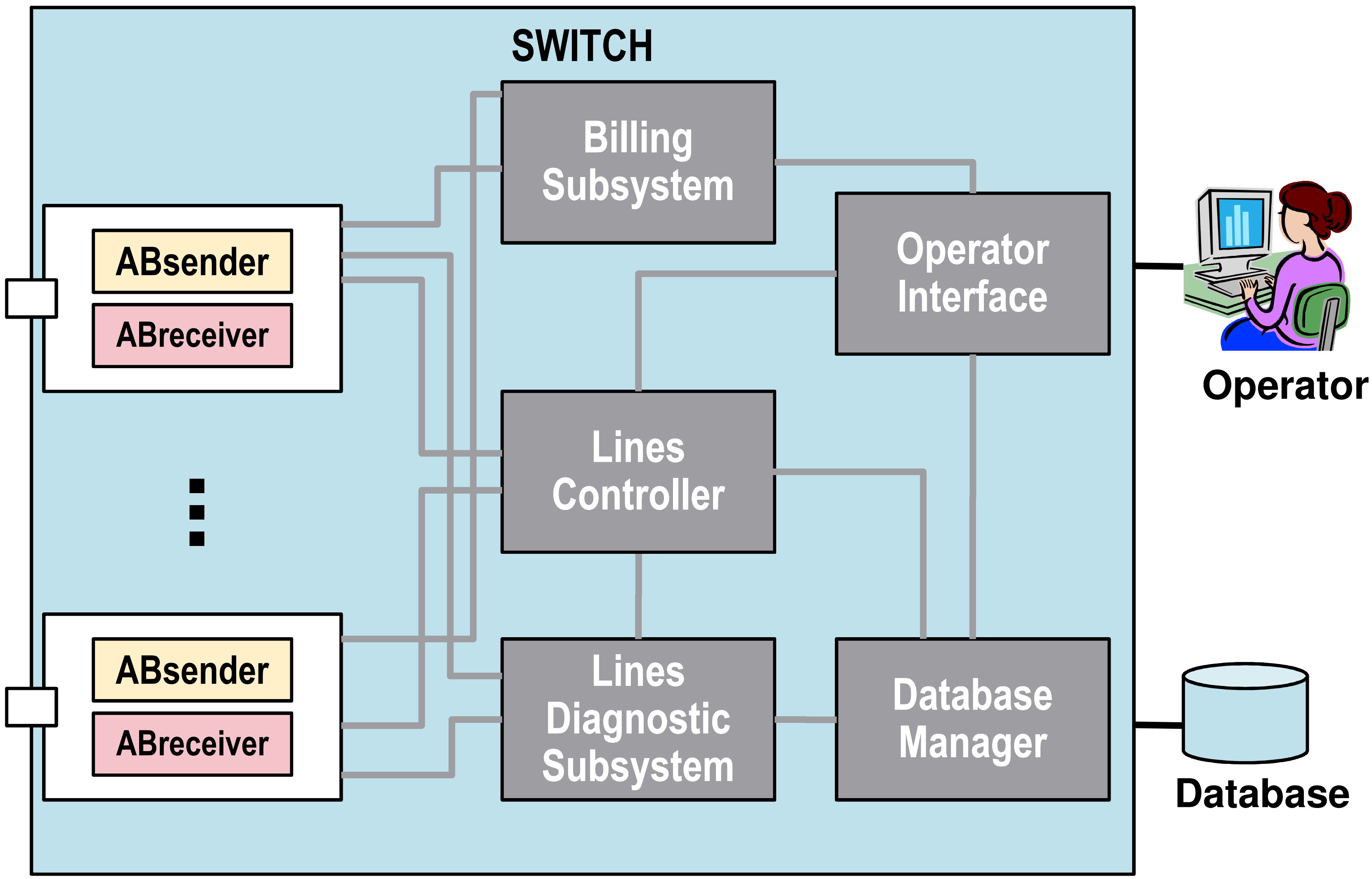


The Alternating Bit Protocol: Spec 2

- **State machines of sender and receiver ends**
  - *Define the primary functionality of the switch*



A (Simplified) Architecture



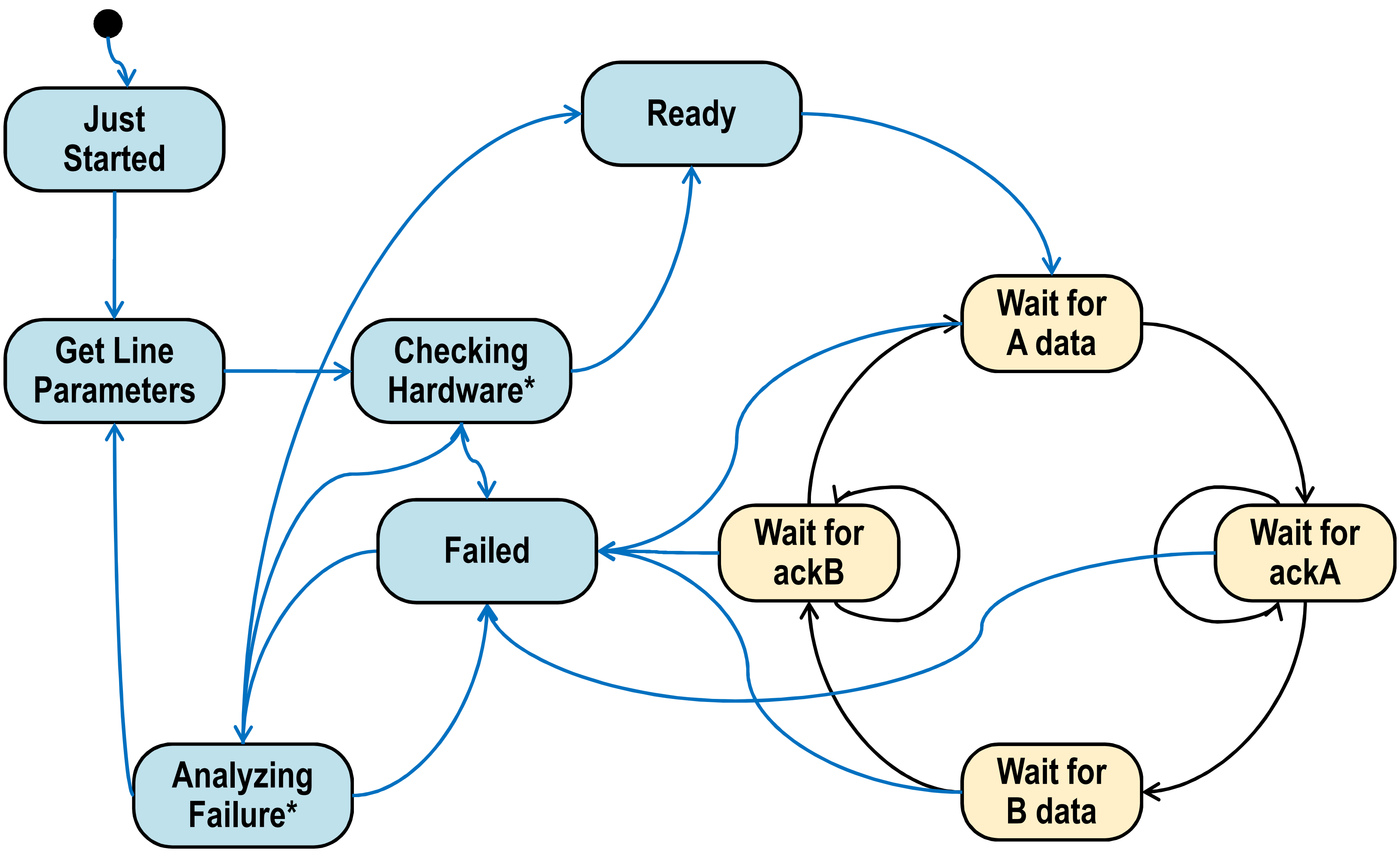
Control

The set of (additional) mechanisms and actions required to bring a system into the desired operational state and to maintain it in that state in the face of various planned and unplanned disruptions

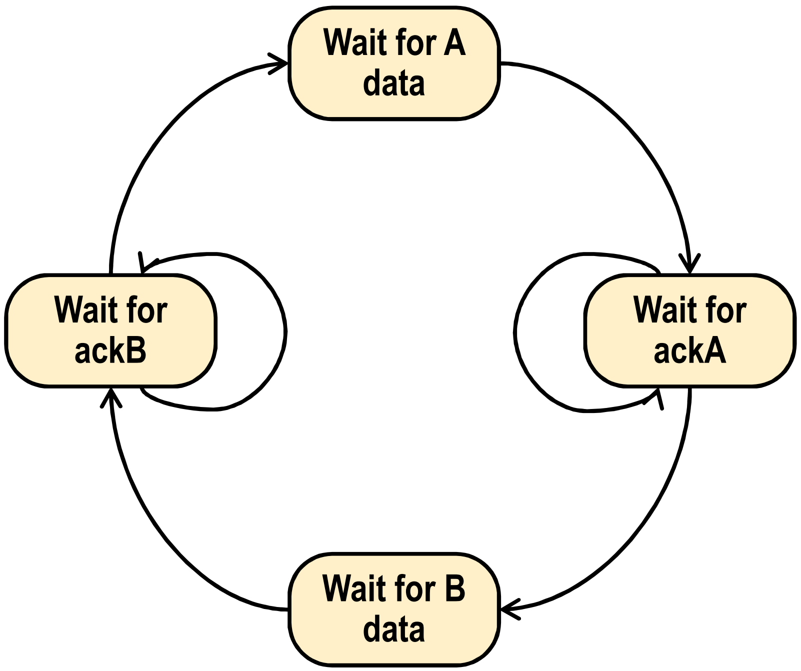
- For software systems this includes:
  - system/component start-up and shut-down
  - failure detection/reporting/recovery
  - system administration, maintenance, and provisioning
  - (on-line) software upgrade

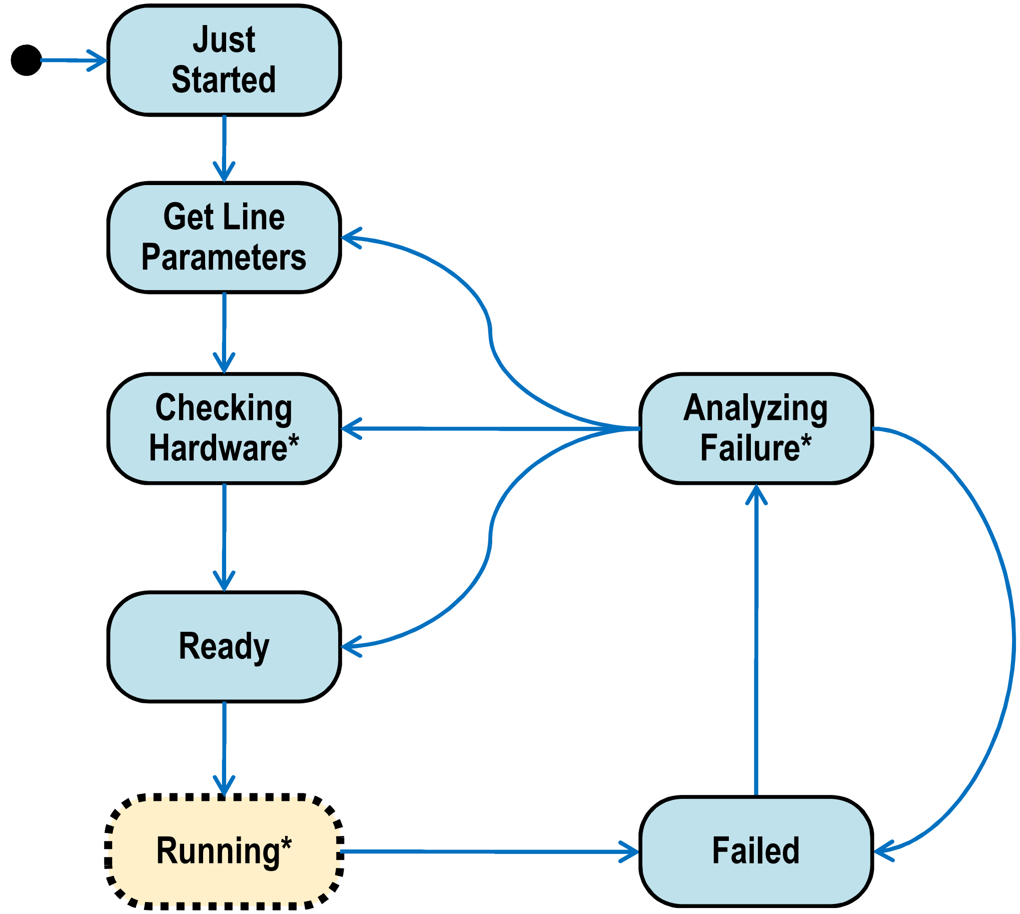
Implementing the Sender

- Sounds simple enough...



The Automata Isolated





If we could find a way of separating them, our job would be much simpler

Control versus Function

- **Control behavior is often treated in an ad hoc manner, since it is not part of the primary system functionality**
  - typically retrofitted into the framework optimized for the functional behavior
  - can lead to controllability and stability problems
- ***However, in highly-dependable systems as much as 80% of the system code is dedicated to control behavior!***

Some Important Observations

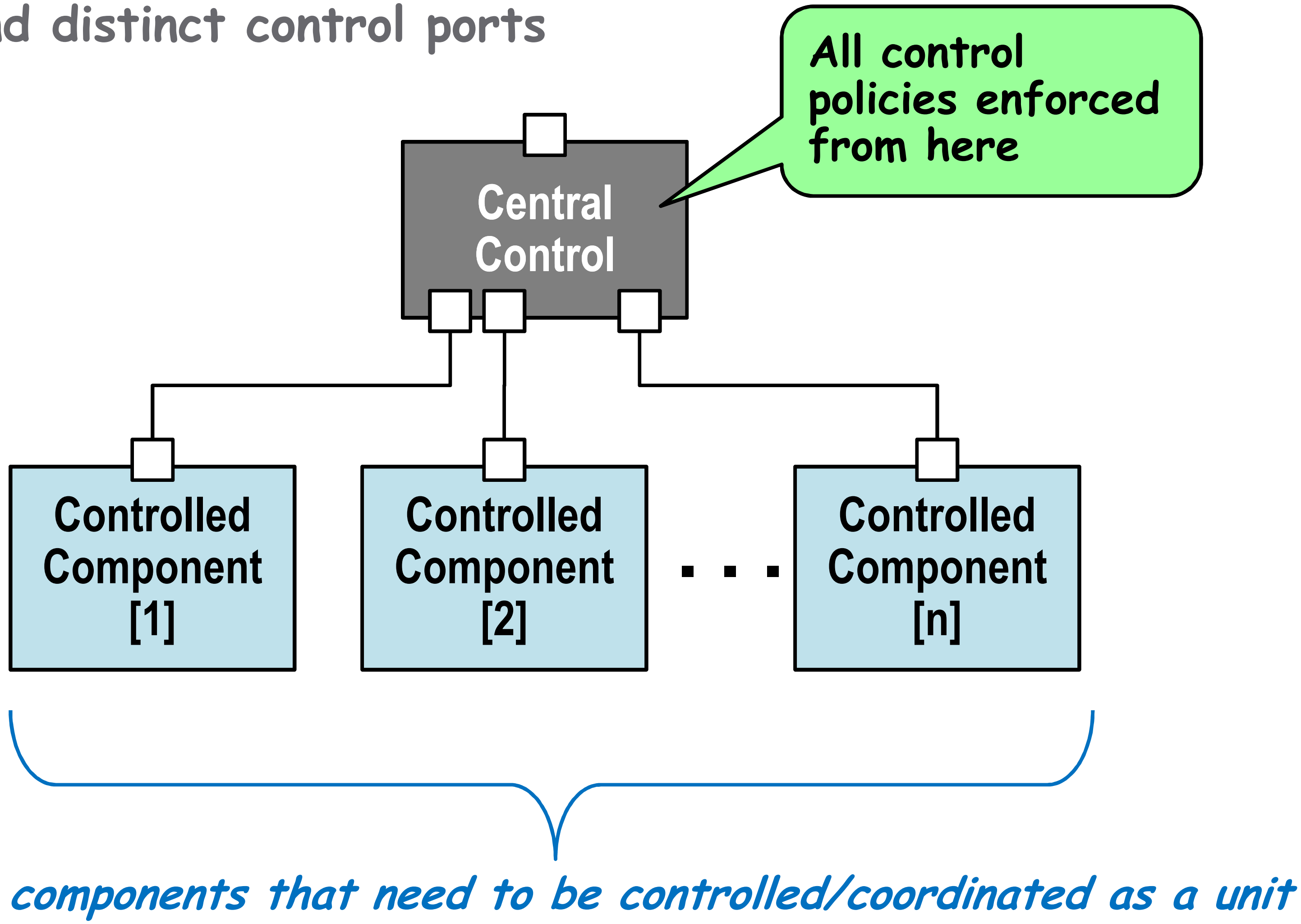
- ***Control predicates function***
  - before a system can perform its primary function, it first has to reach its operational state
- ***Control behavior is often independent of functional behavior***
  - the process by which a system reaches its operational state is often the same regardless of the specific functionality of the component

Basic Design Principles

- ***Separate control from function***
  - separate control components from functional components
  - separate control from functional interfaces
  - imbed functional behavior within control behavior
- ***Centralize control (decision making)***
  - if possible, focus control in one component
  - place control policies in the control components and control mechanisms inside the controlled components

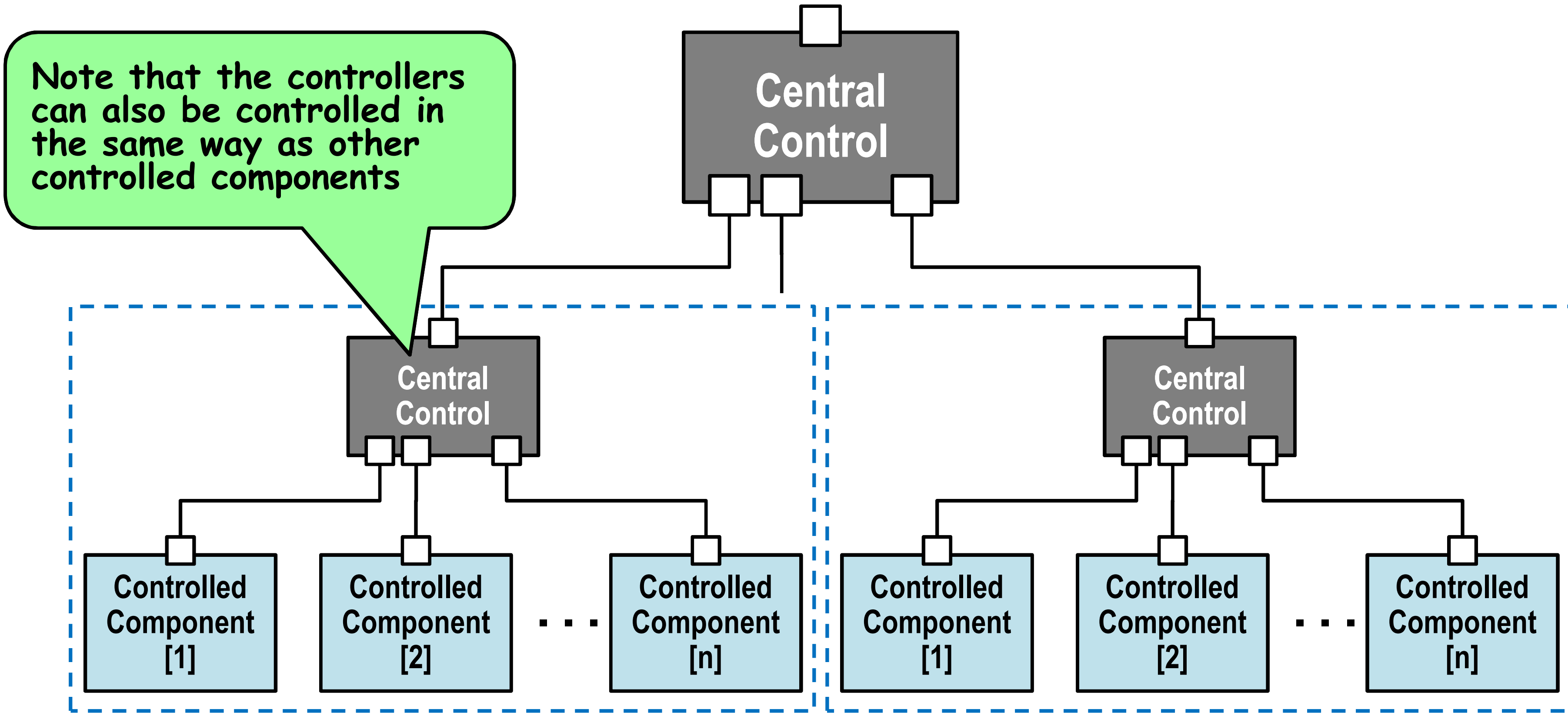
The Core Architectural Pattern

- **A star-like pattern with control in the centre**
  - ...and distinct control ports



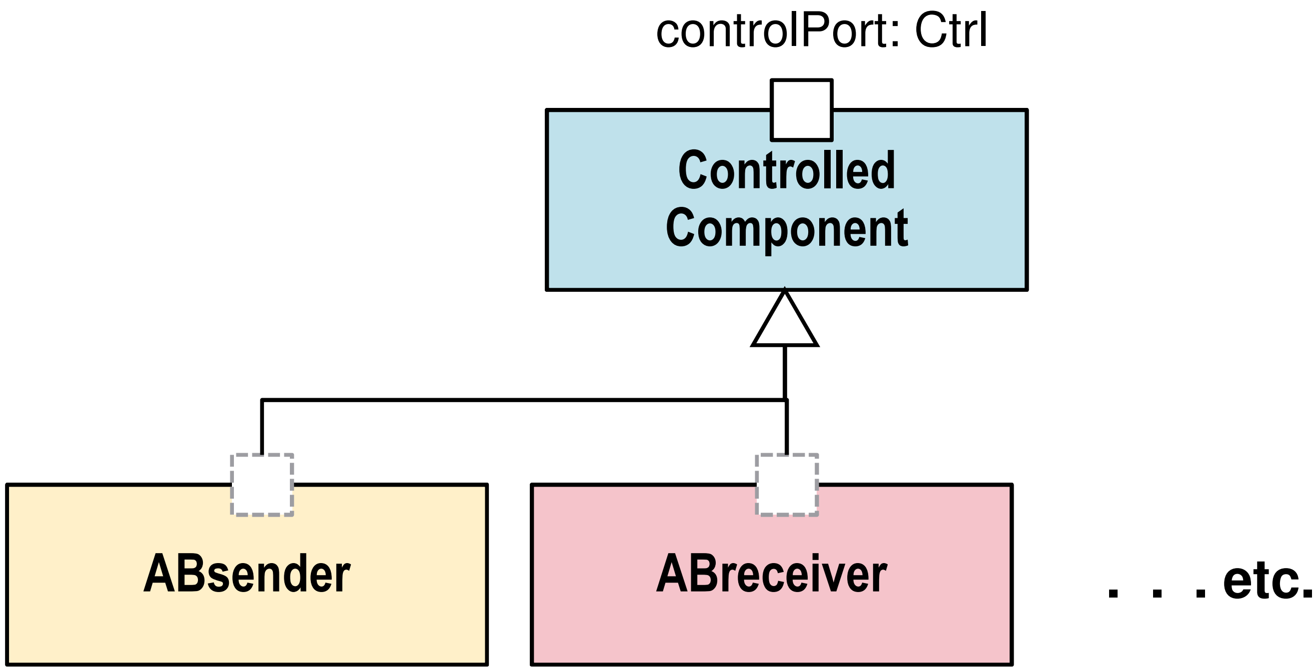
Recursive/Hierarchical Application

- ***The Recursive Control Pattern***
  - Scales easily to very large systems
  - Simple, but ensures consistent and highly controllable dynamic software systems



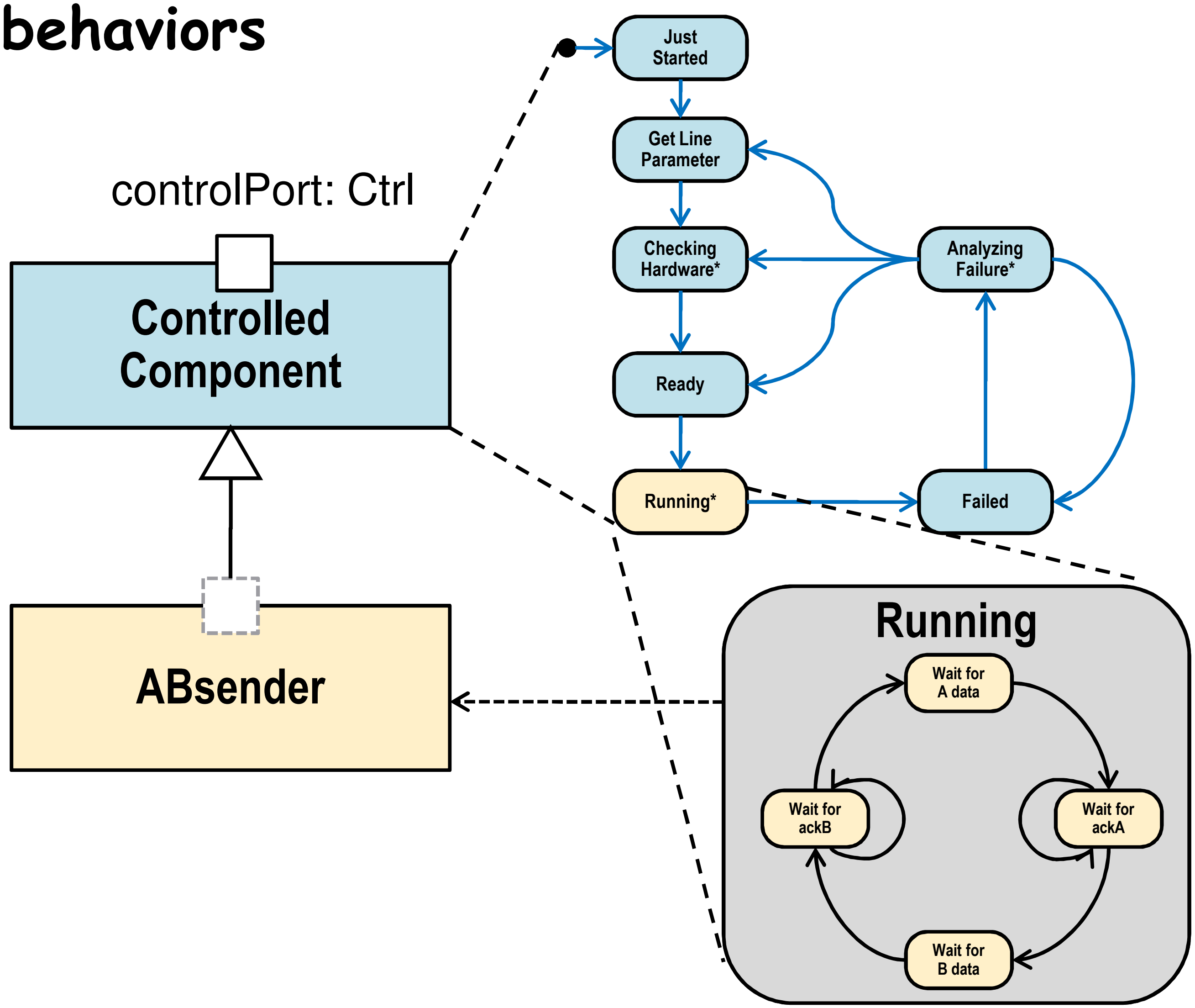
Using the Abstract Component Pattern

- **All controlled components that share the same control automaton can be subclasses of a common abstract class**



Weaving Using Hierarchical States

- **Achieves clean separation of control and functional behaviors**



Summary

- The Recursive Control architectural pattern provides a general architecture for software applications that need to operate continuously
- It partitions the problem into
  - Control part – dealing with the "care and feeding" of the software application (e.g., maintenance, failure recovery, update, etc.)
  - Functional part – dealing with the application itself
- The control part of the architecture is typically not very application specific
- The modeling of this pattern can be greatly simplified if a suitable modeling language is used