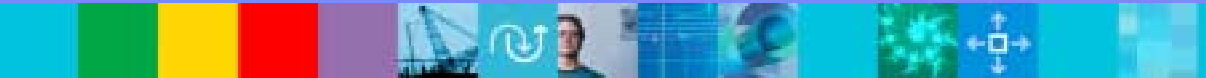
IBM Software Group

Rational. software

Brass Bubbles: An Overview of UML 2.0 (and MDA)

John Hogg
hogg@ca.ibm.com

@business on demand software



Version 3.2

2003-11-21

IMPORTANT DISCLAIMER!

The technical material described here is still under development. It is an “Adopted Specification” but will continue to evolve to an “Available Specification”.

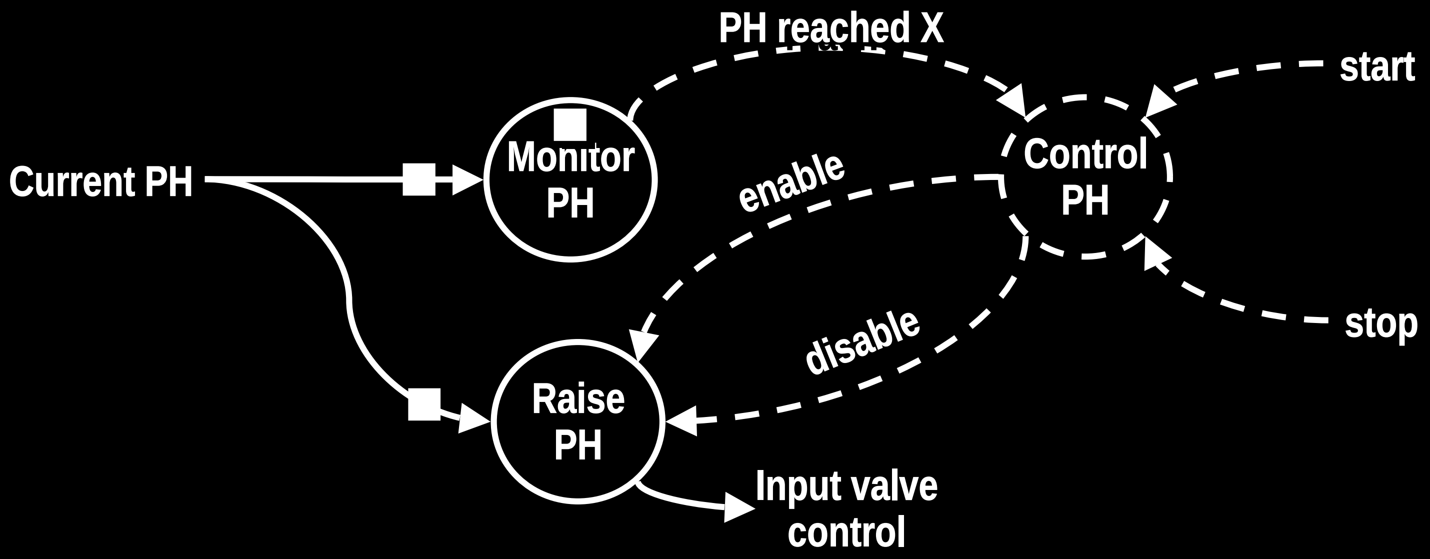
Tutorial Objectives

1. To explain the essential features of model-driven development (based on UML)
2. To clarify the design intent and rationale behind UML 2.0
3. To introduce major new features of UML 2.0

Tutorial Overview

- Introduction: modeling and software
- Model-Driven Development
- A Critique of UML 1.x
- Requirements for UML 2.0
- Foundations of UML 2.0
- Architectural Modeling Capabilities
- Dynamic Semantics
- Interaction Modeling Capabilities
- Activities and Actions
- State Machine Innovations
- Other New Features
- Summary and Conclusion

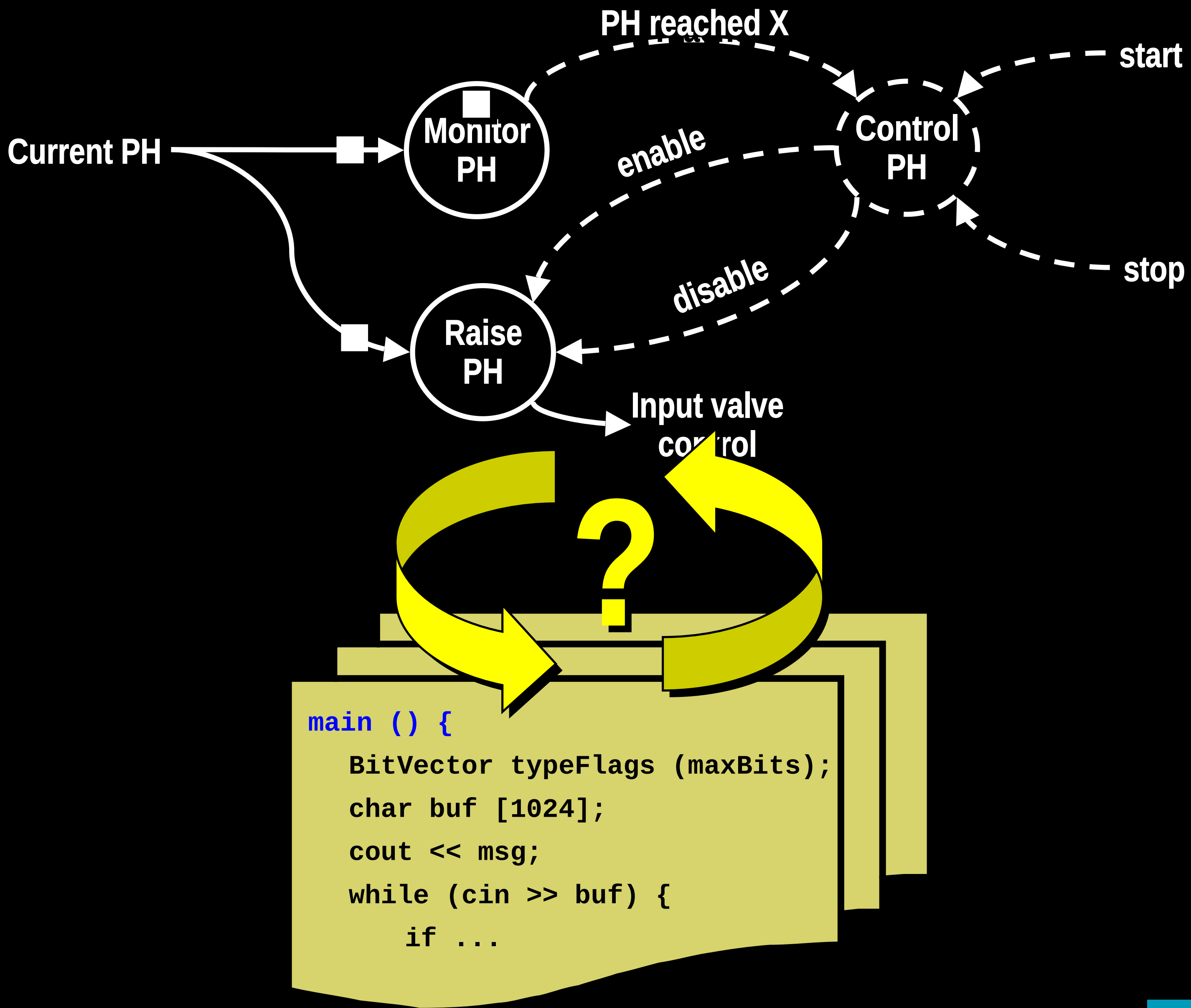
A Skeptic's View of Software Models...



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

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

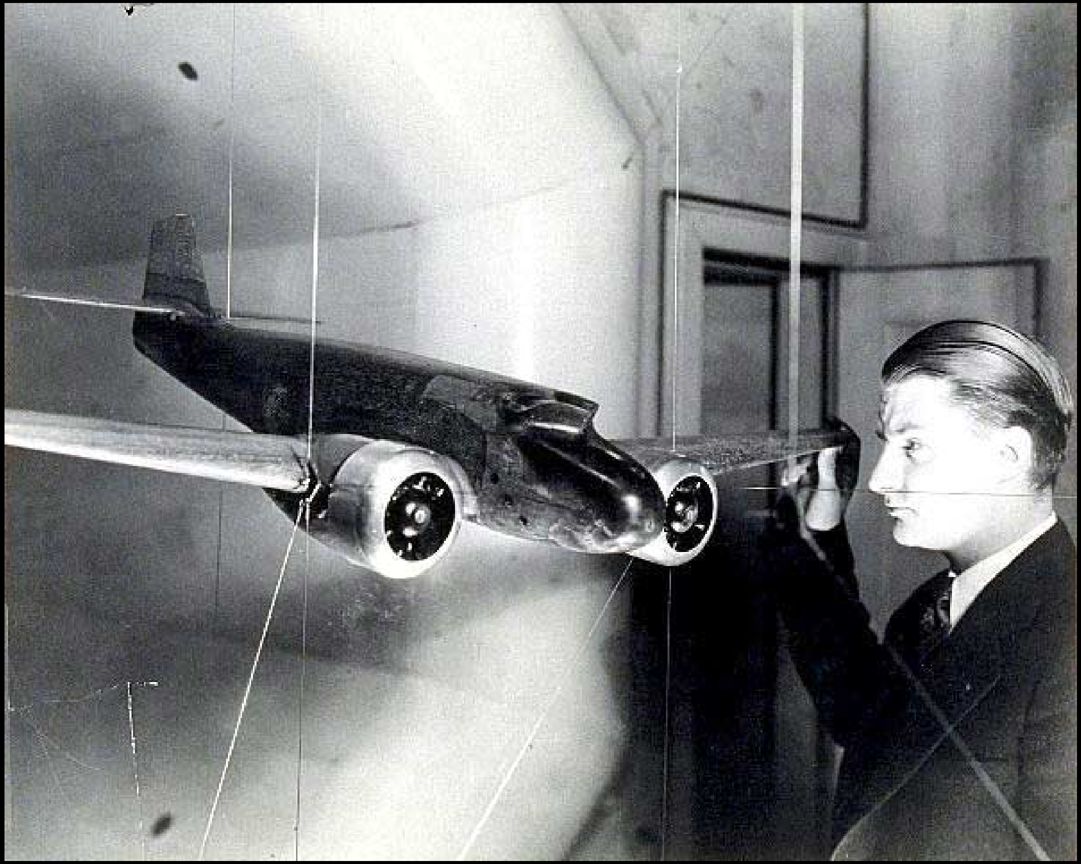
The Problem with Bubbles...



```
main () {
   BitVector typeFlags (maxBits);
   char buf [1024];
   cout << msg;
   while (cin >> buf) {
      if ...
```

Models in Traditional Engineering

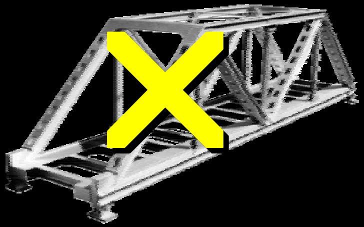
- As old as engineering (e.g., Vitruvius)
- Traditional means of reducing engineering risk

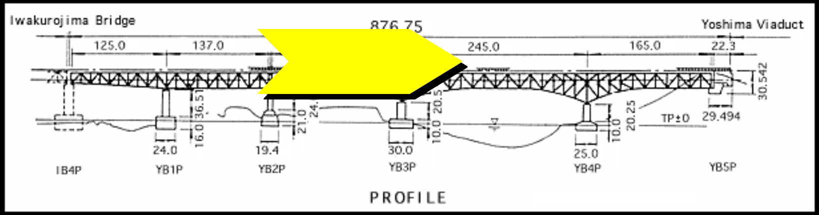


What Engineers Do

- Before they build the real thing...

...they first build models...and then learn from them

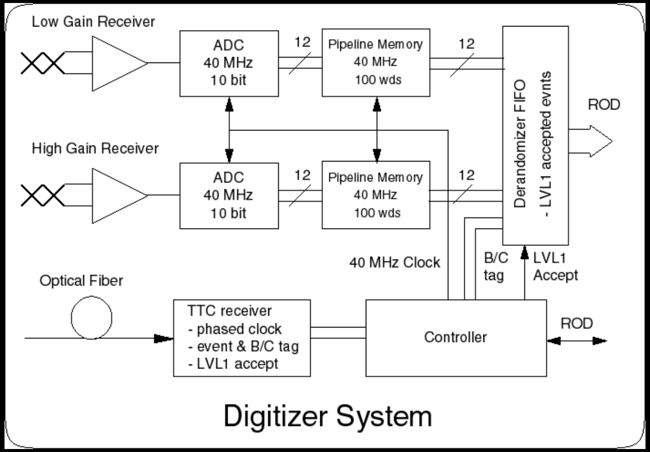




Engineering Models

- Engineering model:
  A reduced representation of some system

Modeled system


Low Gain Receiver
ADC 40 MHz 10 bit
12
Pipeline Memory 40 MHz 100 wds
12
Derandomizer FIFO - LVL1 accepted evnts
ROD
High Gain Receiver
ADC 40 MHz 10 bit
12
Pipeline Memory 40 MHz 100 wds
12
40 MHz Clock
B/C tag
LVL1 Accept
Optical Fiber
TTC receiver - phased clock - event & B/C tag - LVL1 accept
Controller
ROD
Digitizer System

Model

- Purpose:
  To help us understand a complex problem or solution
  To communicate ideas about a problem or solution
  To drive implementation

Characteristics of Useful Models

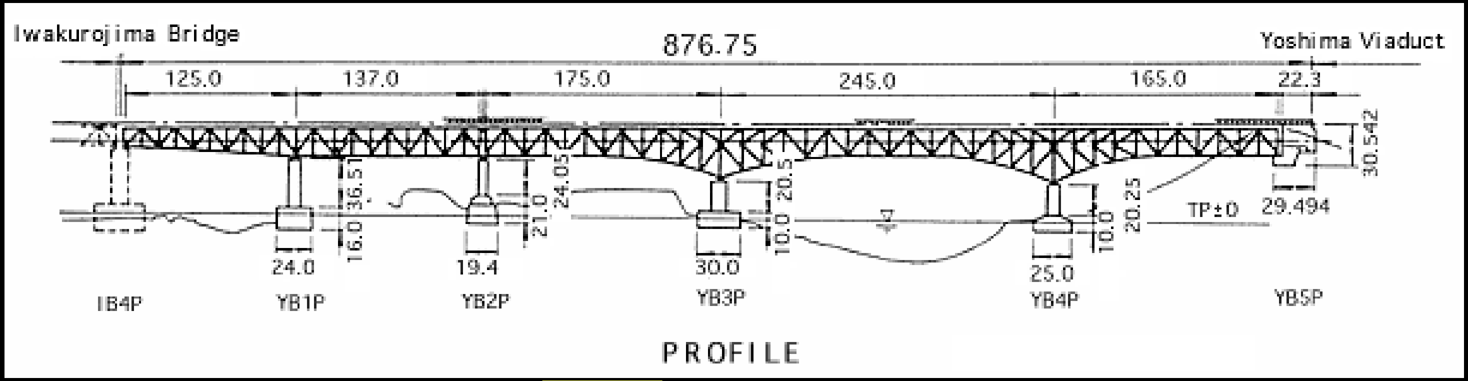
- Abstract
  - Emphasize important aspects while removing irrelevant ones
- Understandable
  - Expressed in a form that is readily understood by observers
- Accurate
  - Faithfully represents the modeled system
- Predictive
  - Can be used to derive correct conclusions about the modeled system
- Inexpensive
  - Much cheaper to construct and study than the modeled system

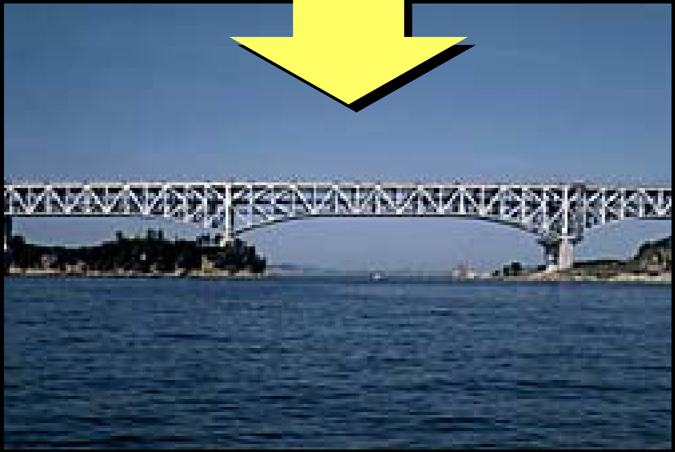
To be useful, engineering models must satisfy <u>all</u> of these characteristics!

How Models are Used

- To detect errors and omissions in designs before committing full resources to full implementation
  - Through (formal) analysis and experimentation
  - Investigate and compare alternative solutions
  - Minimize engineering risk
  - *Make mistakes cheaply*
- To communicate with stakeholders
  - Clients, users, implementers, testers, documenters, etc.
- To drive implementation

A Problem with Models





Semantic Gap due to:

- Idiosyncrasies of actual construction materials
- Construction methods
- Scaling effects
- Skill sets
- Misunderstandings
- Inaccurate models

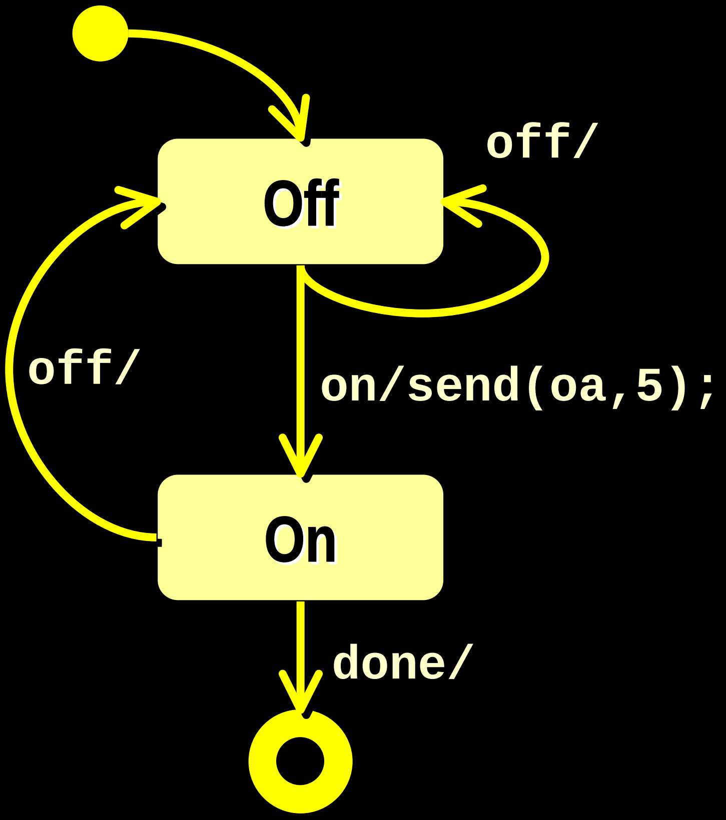
Can lead to serious errors and discrepancies in the realization

Models of Software

- A description of the software which
  - Abstracts out irrelevant detail
  - Presents the software using higher-level abstractions

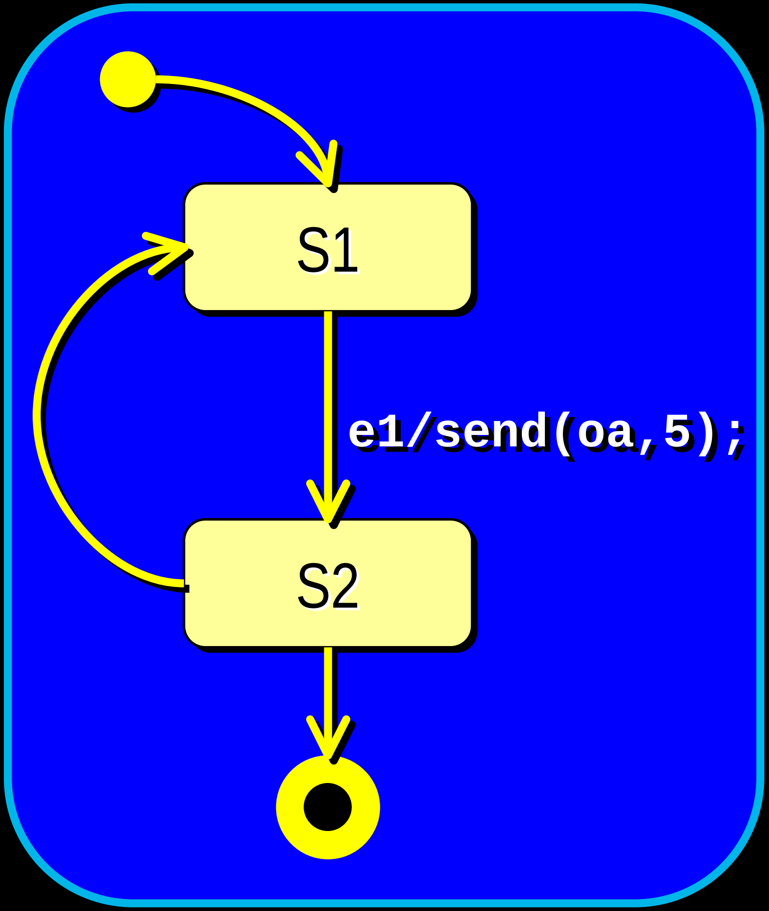
```
case mainState of
   initial: send(“I am here”);
            end
   Off:     case event of
              on: send(oa,5);
                  next(On);
                  end
              off: next(Off);
                  end

            end
    On:     case event of
              off: next(Off);
                  end
              done: terminate;
                  end
            end
   end
```



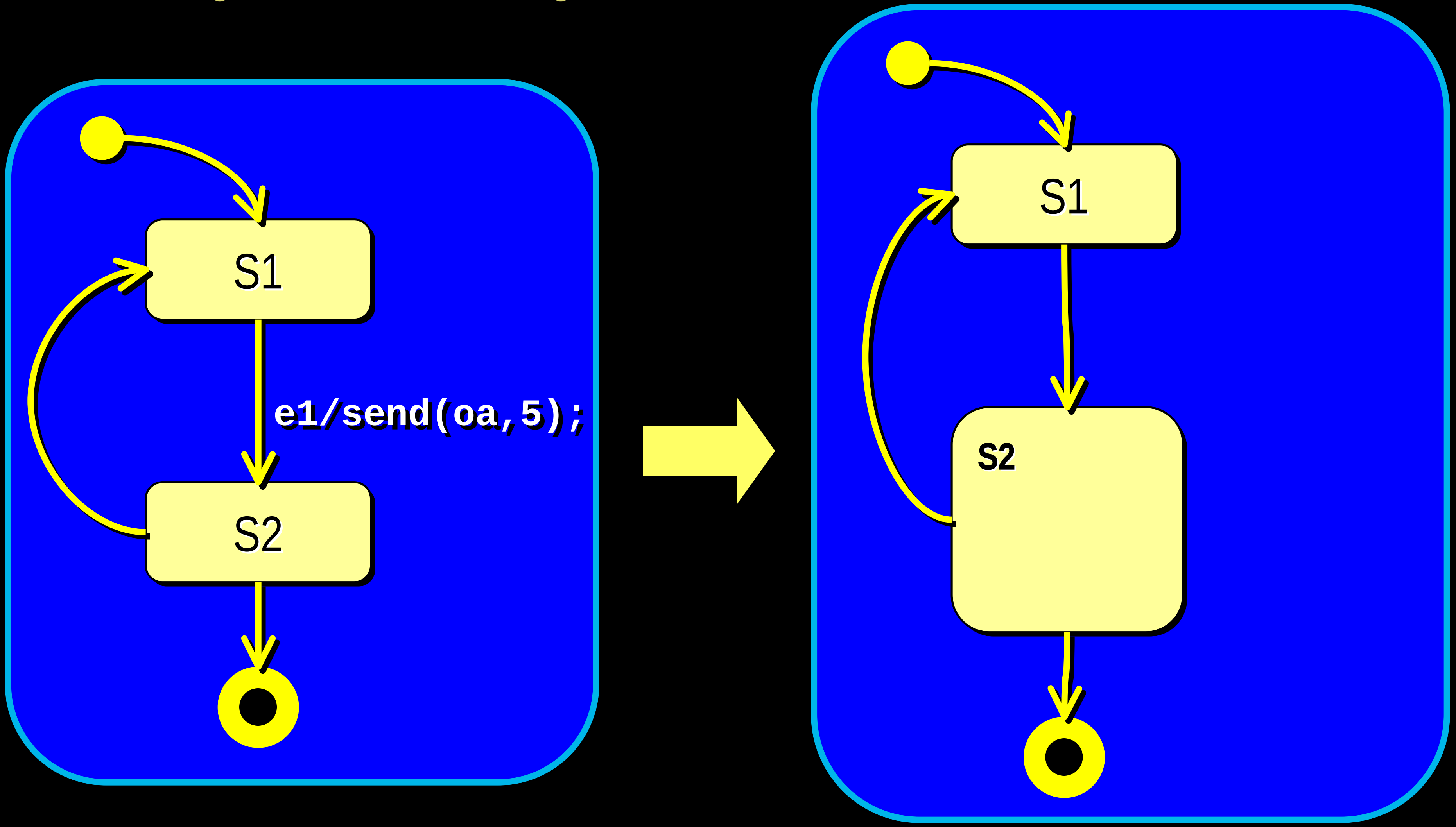
Evolving Models

- Adding detail to a high-level model:



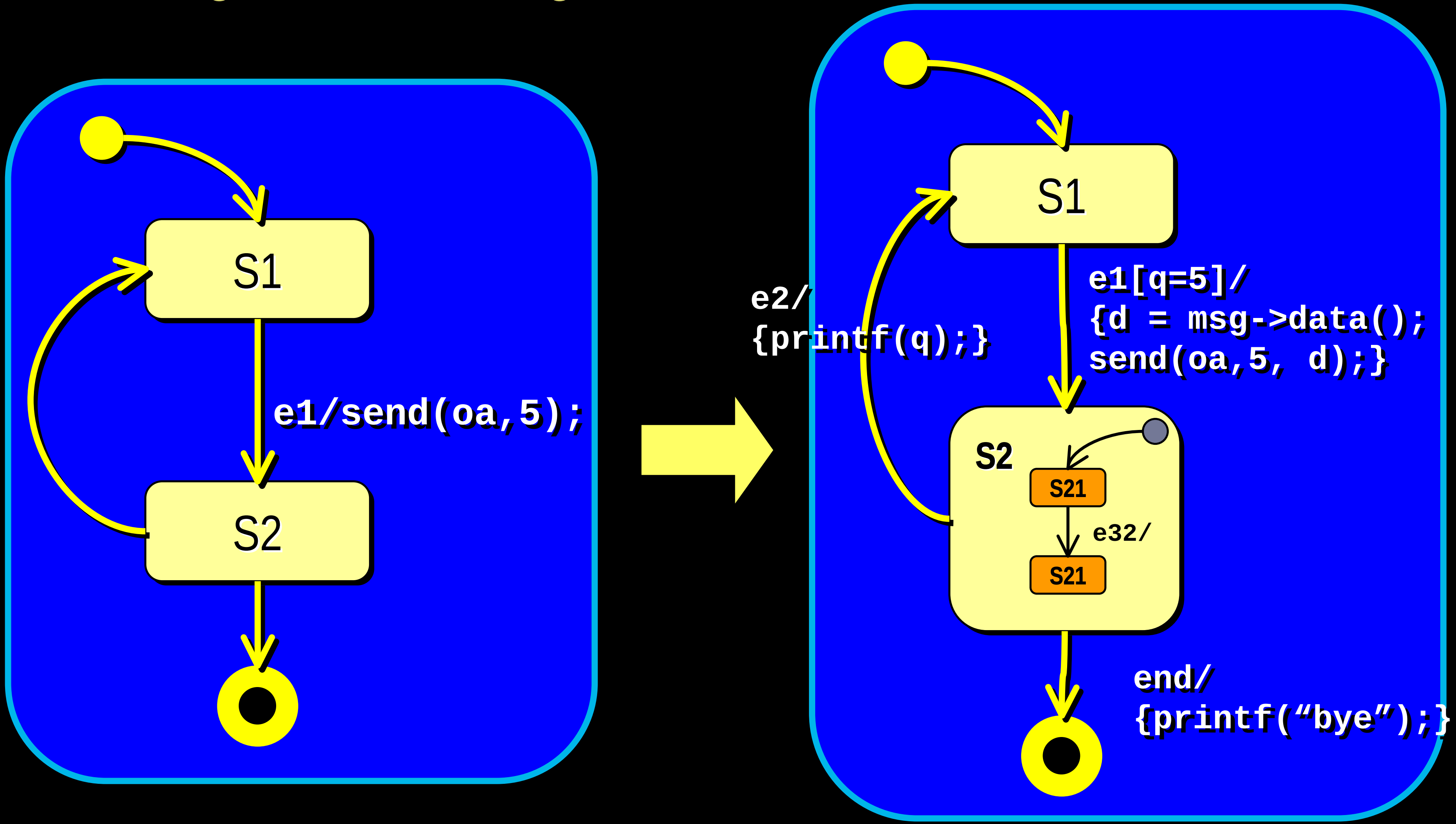
Evolving Models

- Adding detail to a high-level model:



Evolving Models

- Adding detail to a high-level model:



The Remarkable Thing About Software

Software has the rare property that it allows us to directly evolve models into full-fledged implementations without changing the engineering medium, tools, or methods!

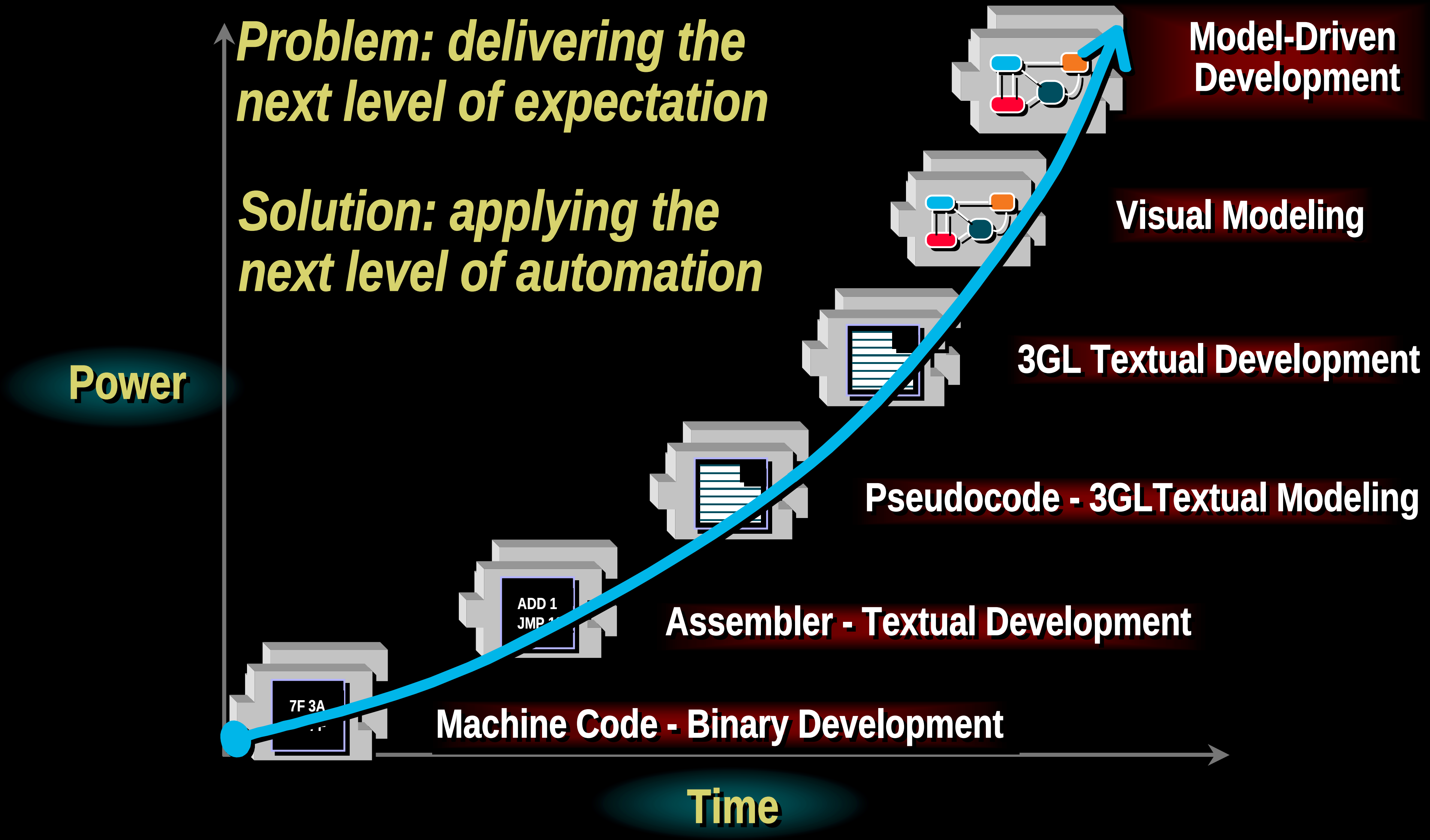
⇒ This can ensure perfect accuracy of software models, since the model and the system that it models are the same thing

The model <u>is</u> the implementation

The Evolution of Software Development

Problem: delivering the next level of expectation

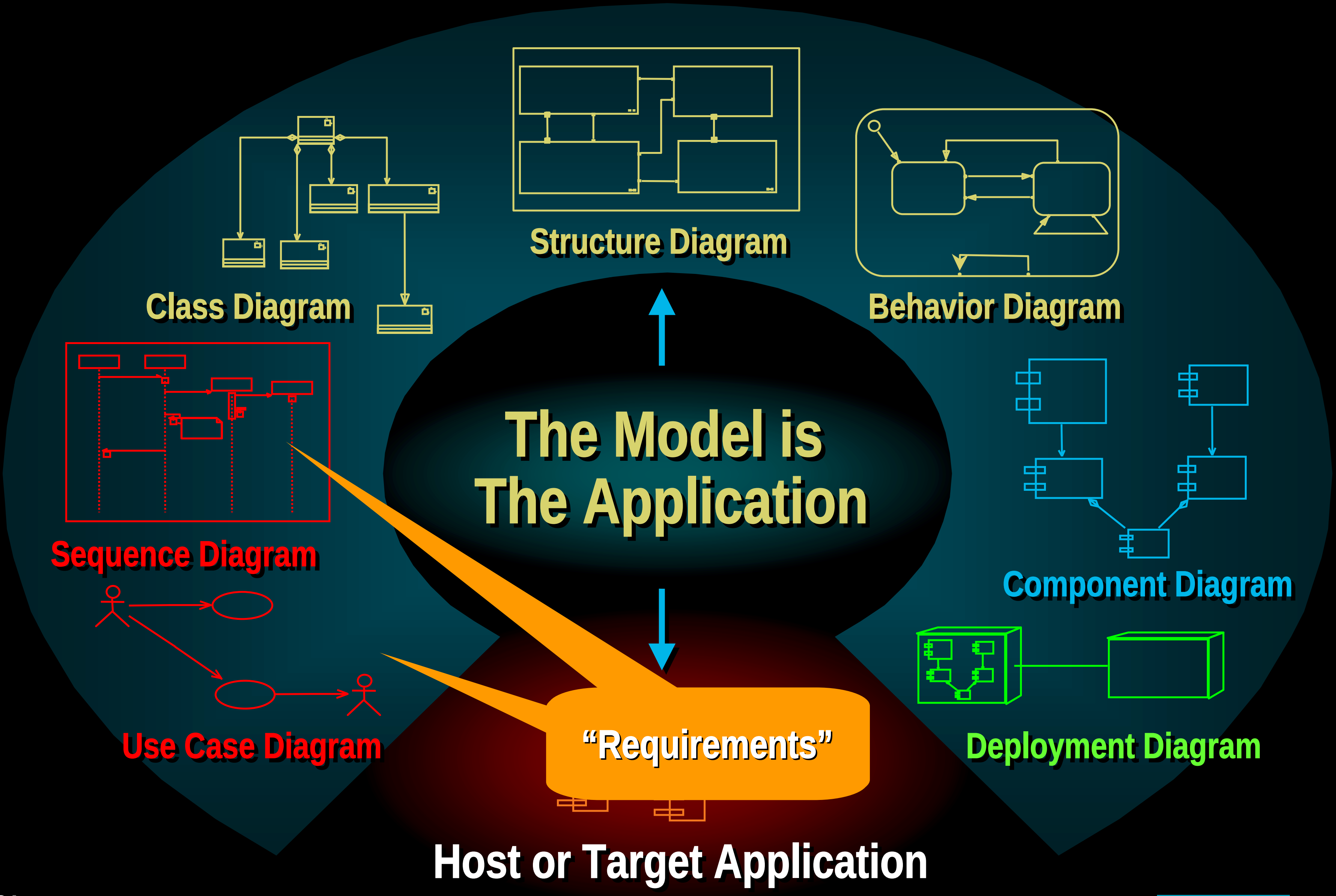
Solution: applying the next level of automation


Model-Driven Development
Visual Modeling
3GL Textual Development
Pseudocode - 3GLTextual Modeling
Assembler - Textual Development
Machine Code - Binary Development
ADD 1
JMP 1
7F 3A
Power
Time

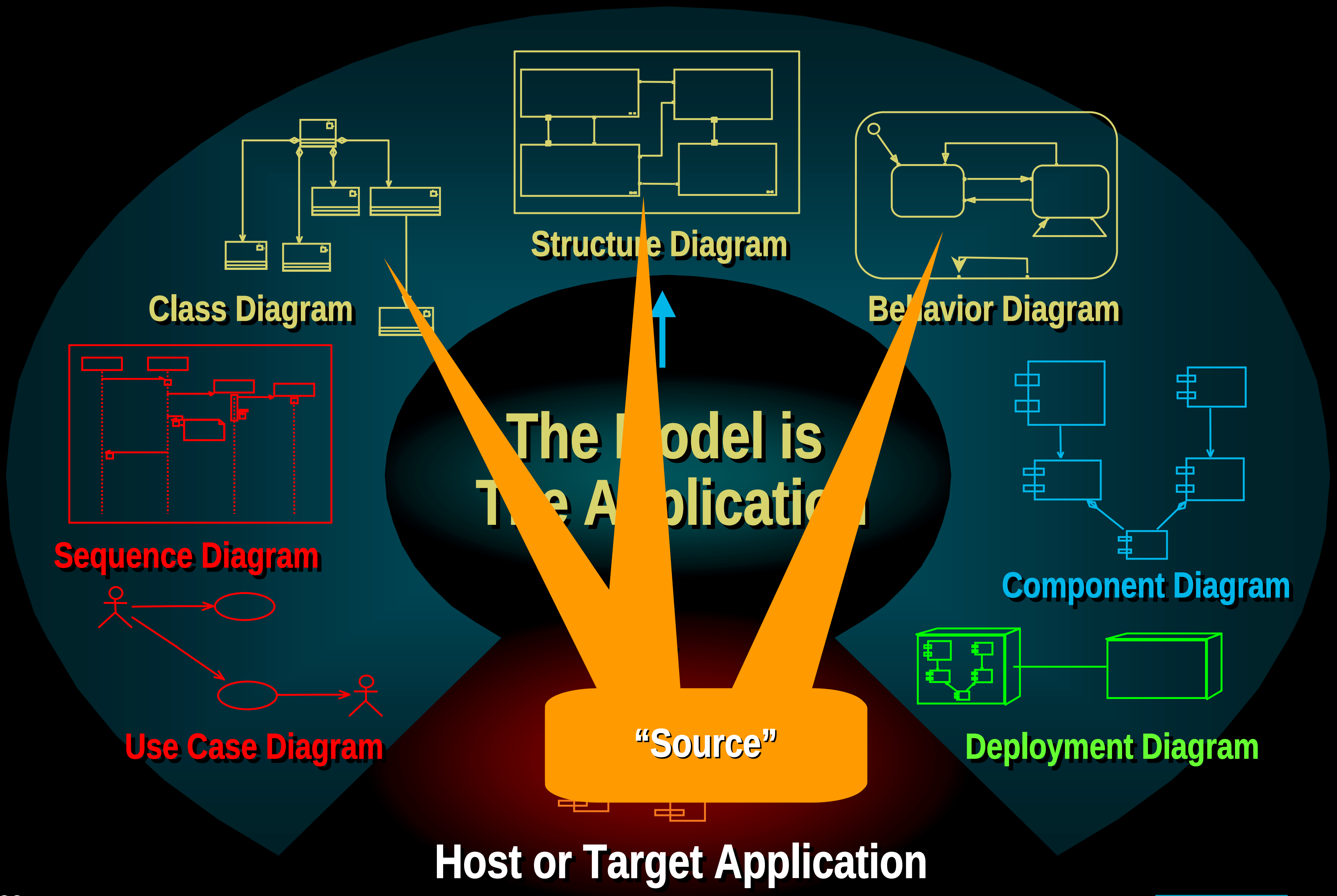
Model-Driven Style of Development (MDD)

- An approach to software development in which the focus and primary artifacts of development are models (as opposed to programs)
  - Implies automatic generation of programs from models
  - Using modeling languages directly as implementation tools
  - “The model is the implementation”

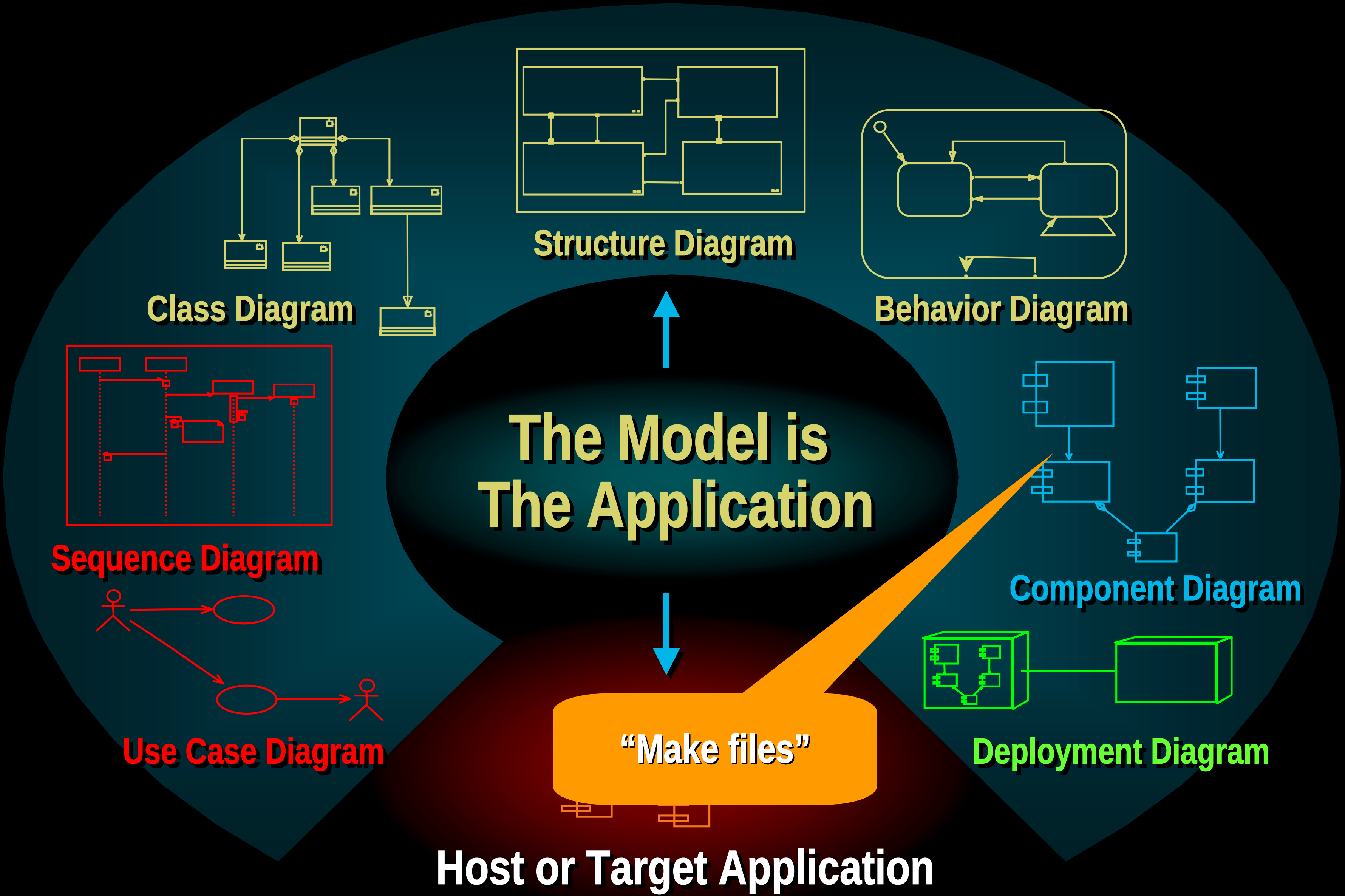
The UML Model Is The Application



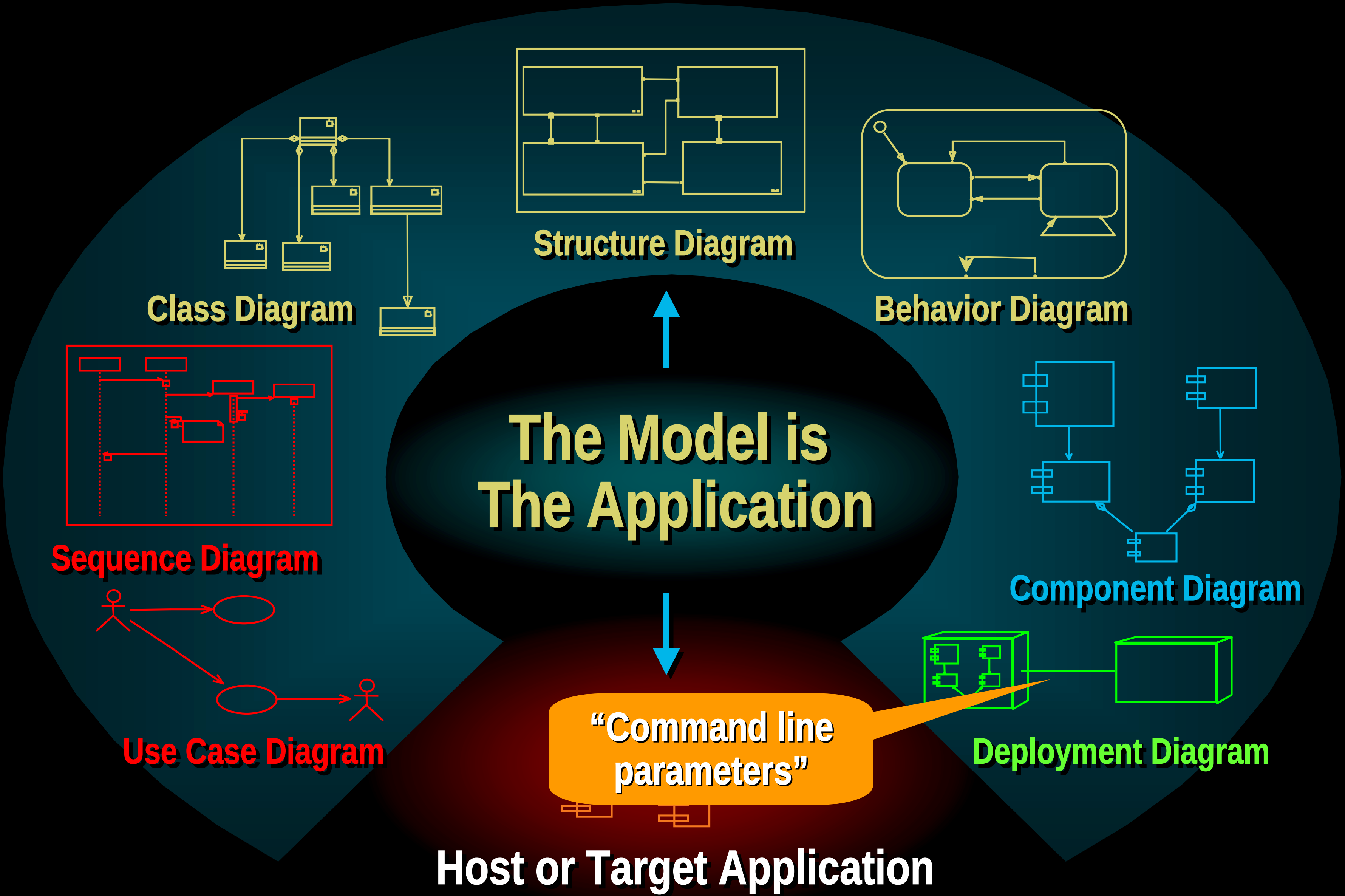
The UML Model Is The Application



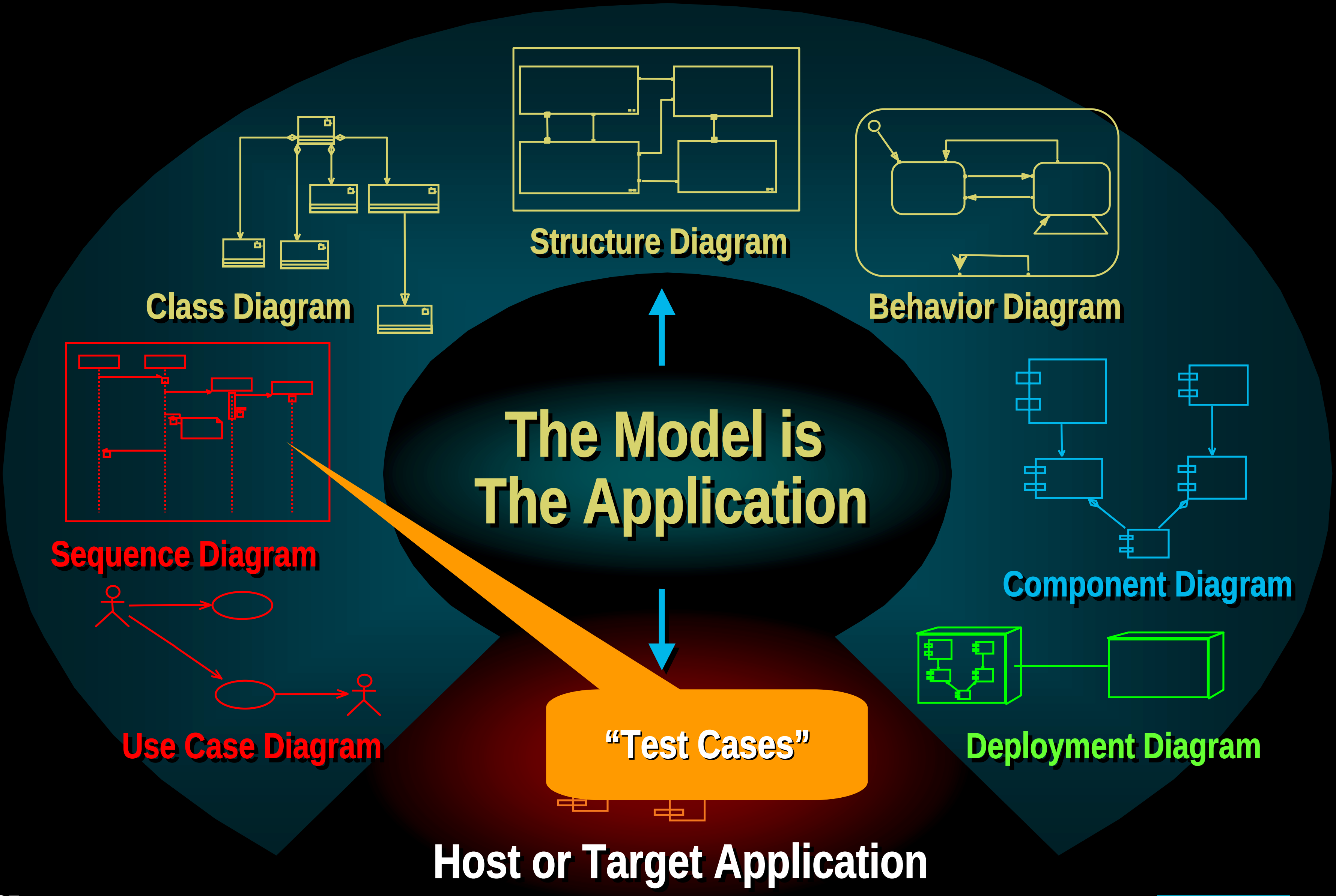
The UML Model Is The Application



The UML Model Is The Application

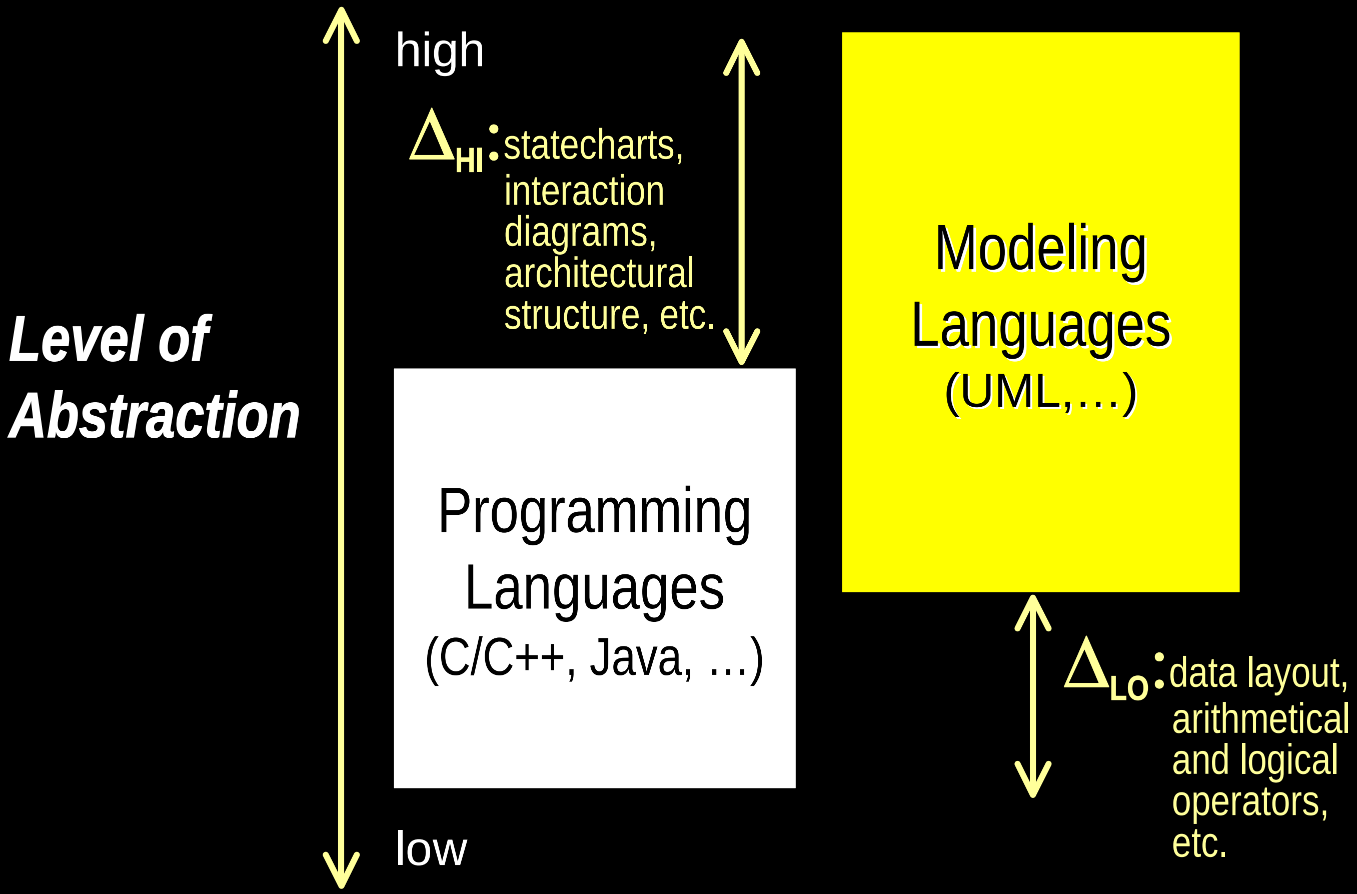


The UML Model Is The Application



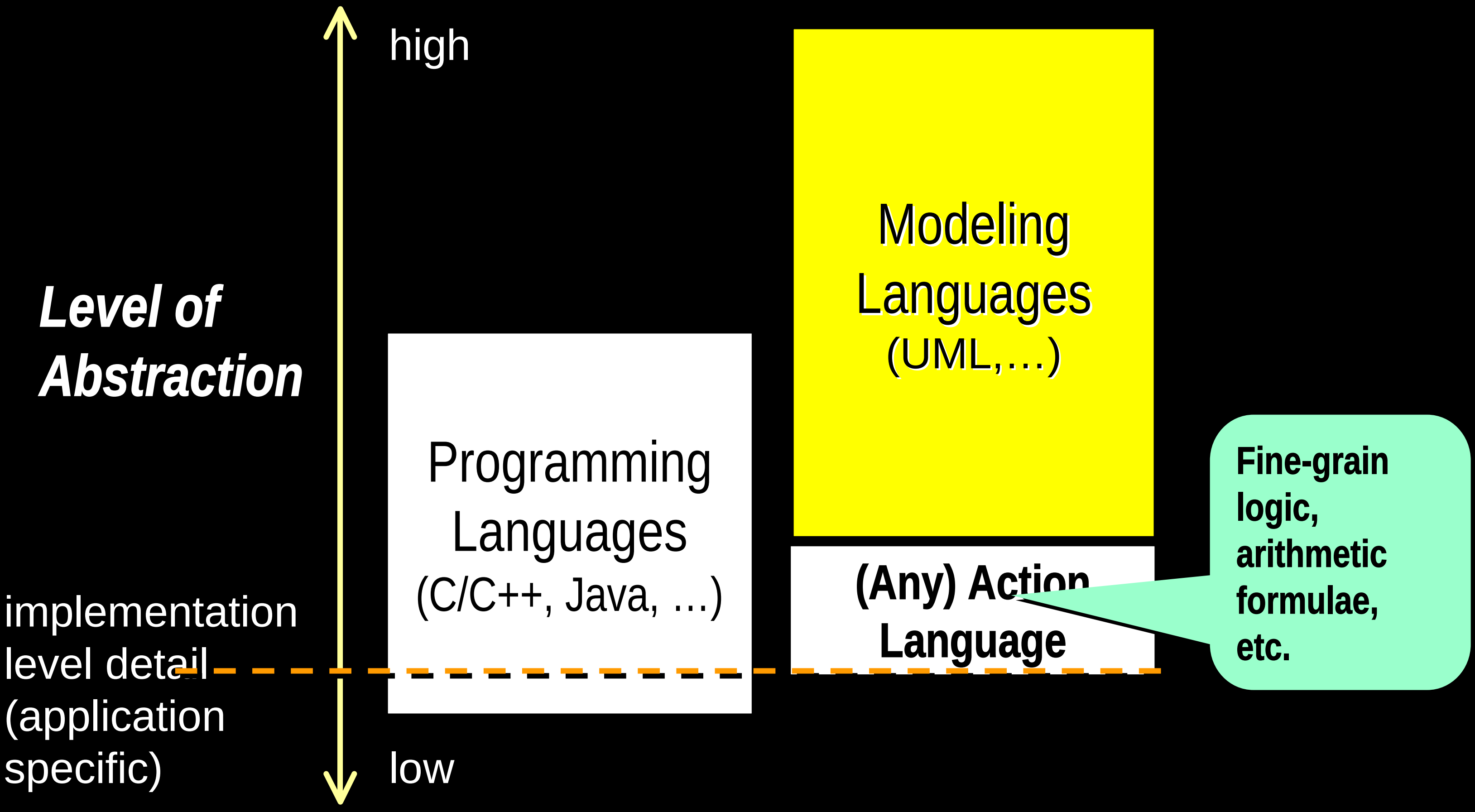
Modeling versus Programming Languages

- Cover different ranges of abstraction


Level of Abstraction
high
low
Δ_{HI}: statecharts, interaction diagrams, architectural structure, etc.
Modeling Languages (UML,…)
Programming Languages (C/C++, Java, …)
Δ_{LO}: data layout, arithmetical and logical operators, etc.

Covering the Full Range of Detail

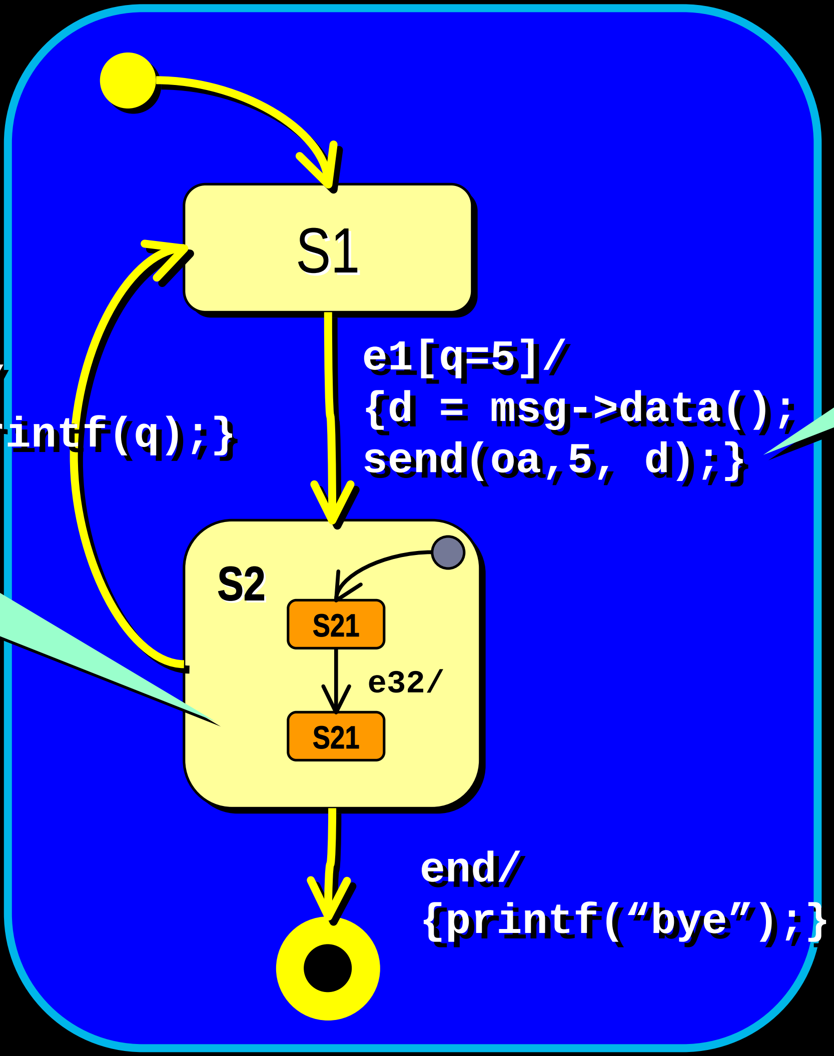
- "Action" languages (e.g., Java, C++) for fine-grain detail



high
Level of Abstraction
Programming Languages (C/C++, Java, …)
Modeling Languages (UML,…)
(Any) Action Language
Fine-grain logic, arithmetic formulae, etc.
implementation level detail (application specific)
low

Example Spec

- Appropriate languages for each abstraction level



High-level parts described using high-level abstractions

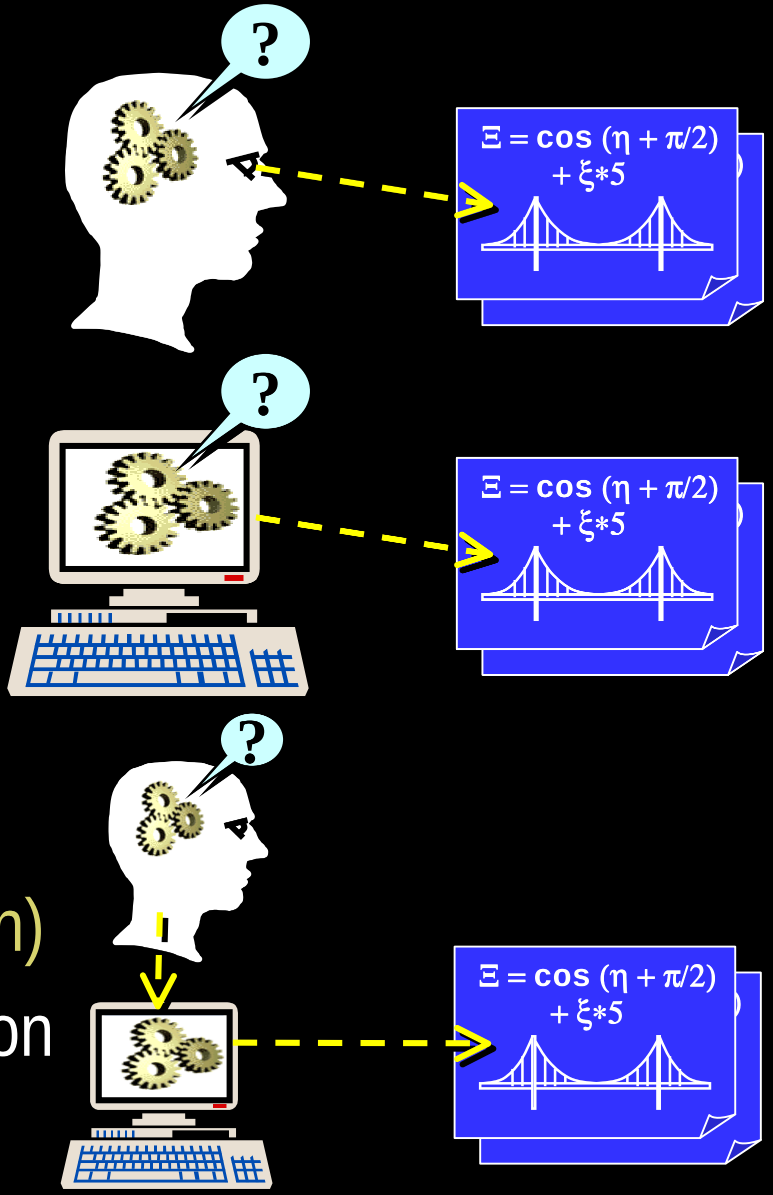
Fine-grain logic in a traditional 3G language

Advantage: exploits

- Existing tools
- Code libraries
- Developer experience

How We Learn From Models

- By inspection
  - mental execution
  - unreliable
- By formal analysis
  - mathematical methods
  - reliable (theoretically)
  - *formal analysis answers very narrow questions!*
- By experimentation (execution)
  - more reliable than inspection
  - direct experience/insight


?
$\Xi = \cos(\eta + \pi/2) + \xi * 5$
?
$\Xi = \cos(\eta + \pi/2) + \xi * 5$
?
$\Xi = \cos(\eta + \pi/2) + \xi * 5$

MDD Implications

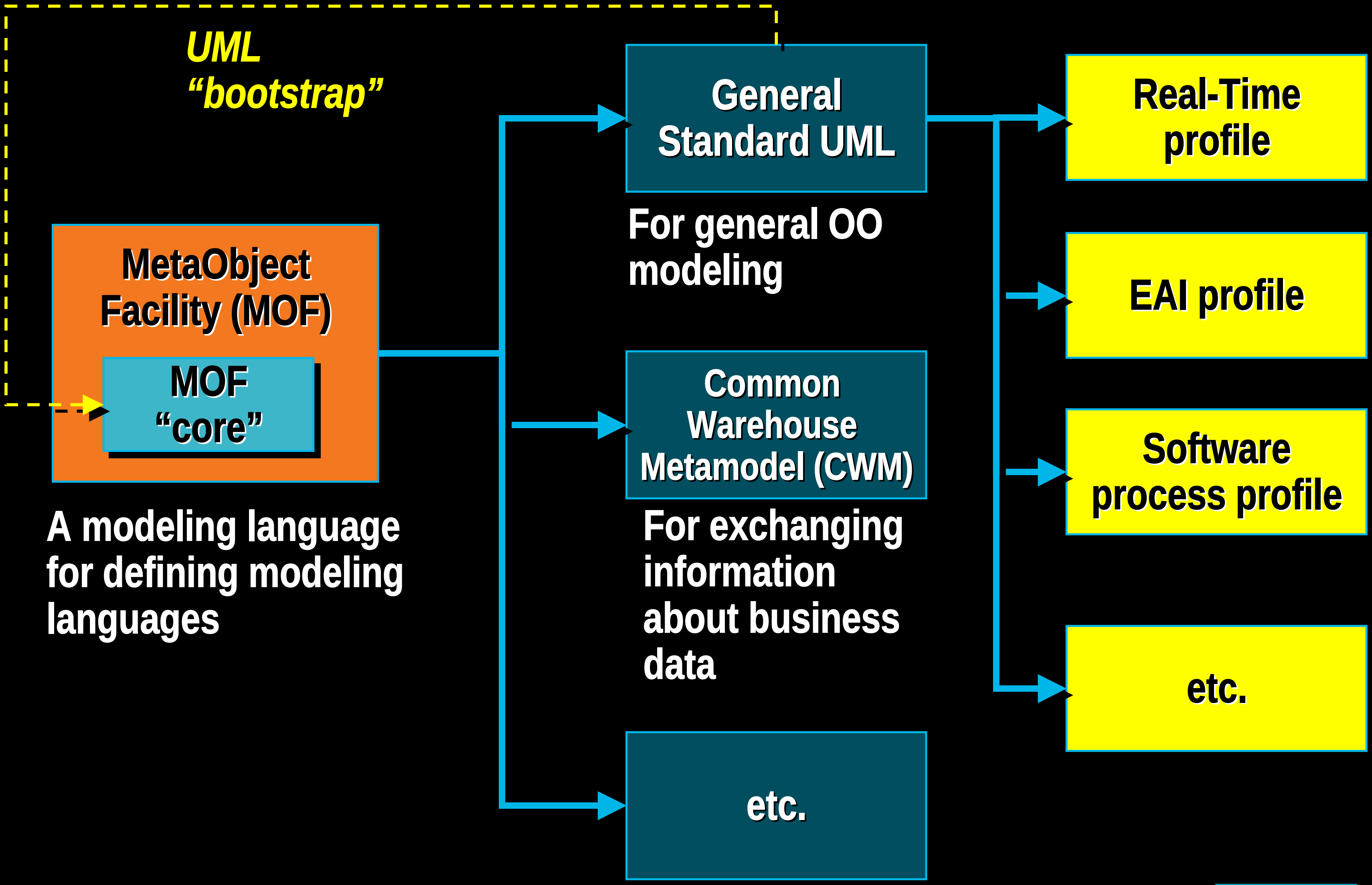
- Ultimately, it should be possible to:
  - Execute models
  - Translate them automatically into implementations
  - …possibly for different implementation platforms
  - ⇨ Platform independent models (PIMs)
- Modeling language requirements
  - The semantic underpinnings of modeling languages must be precise and unambiguous
  - It should be possible to easily specialize a modeling language for a particular domain
  - It should be possible to easily define new specialized languages

Model-Driven Architecture

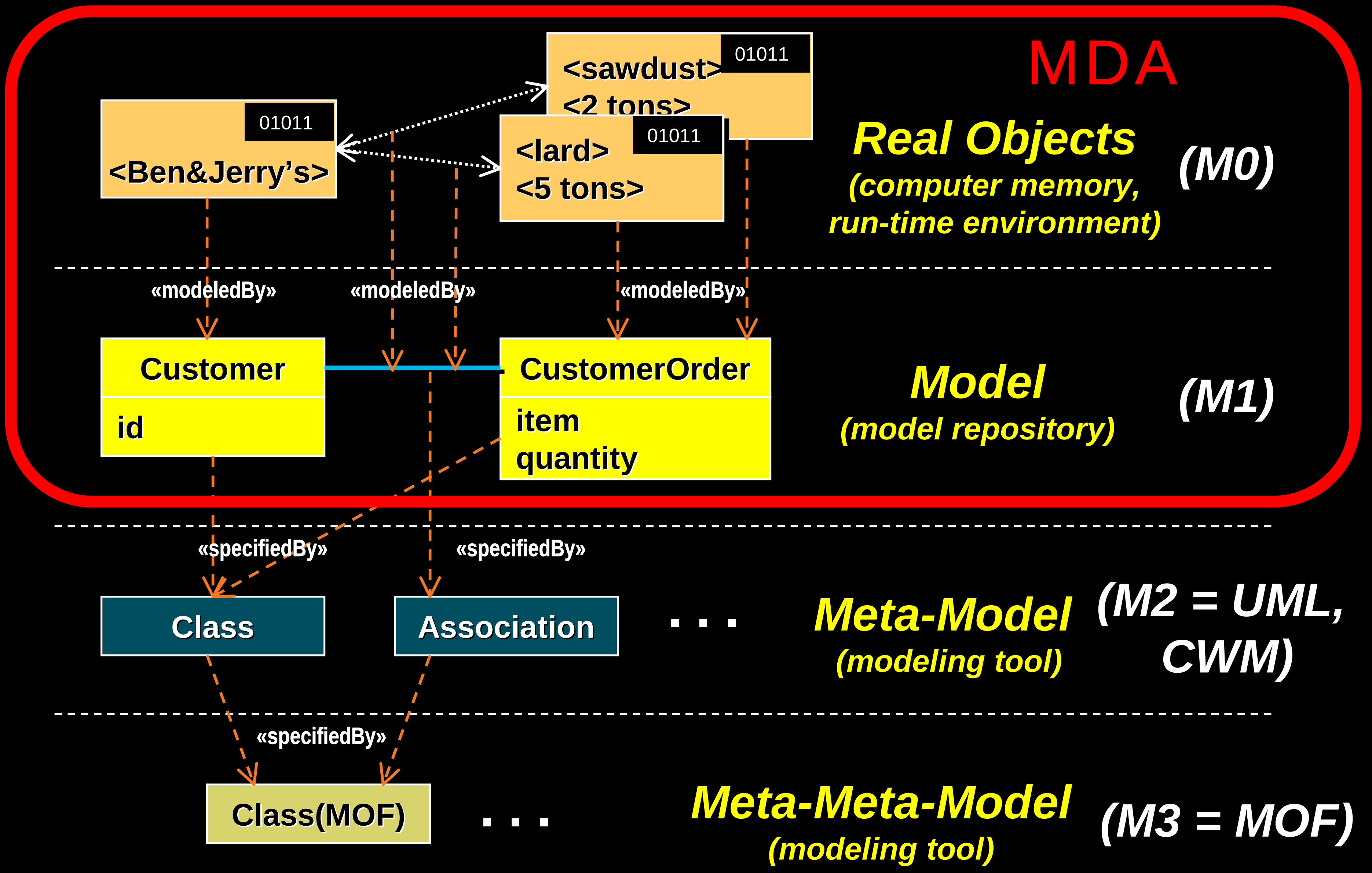
- An OMG initiative
  - A framework for a set of standards in support of MDD
- Inspired by:
  - The widespread public acceptance of UML
  - The availability of mature MDD technologies
  - OMG moving beyond middleware (CORBA)
- Purpose:
  - Enable inter-working between complementary tools
  - Foster specialization of tools and methods
  - Provide guidance for MDD
- Overview papers:
  - http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01
  - http://www.omg.org/cgi-bin/doc?omg/2003-05-01

The Languages of MDA

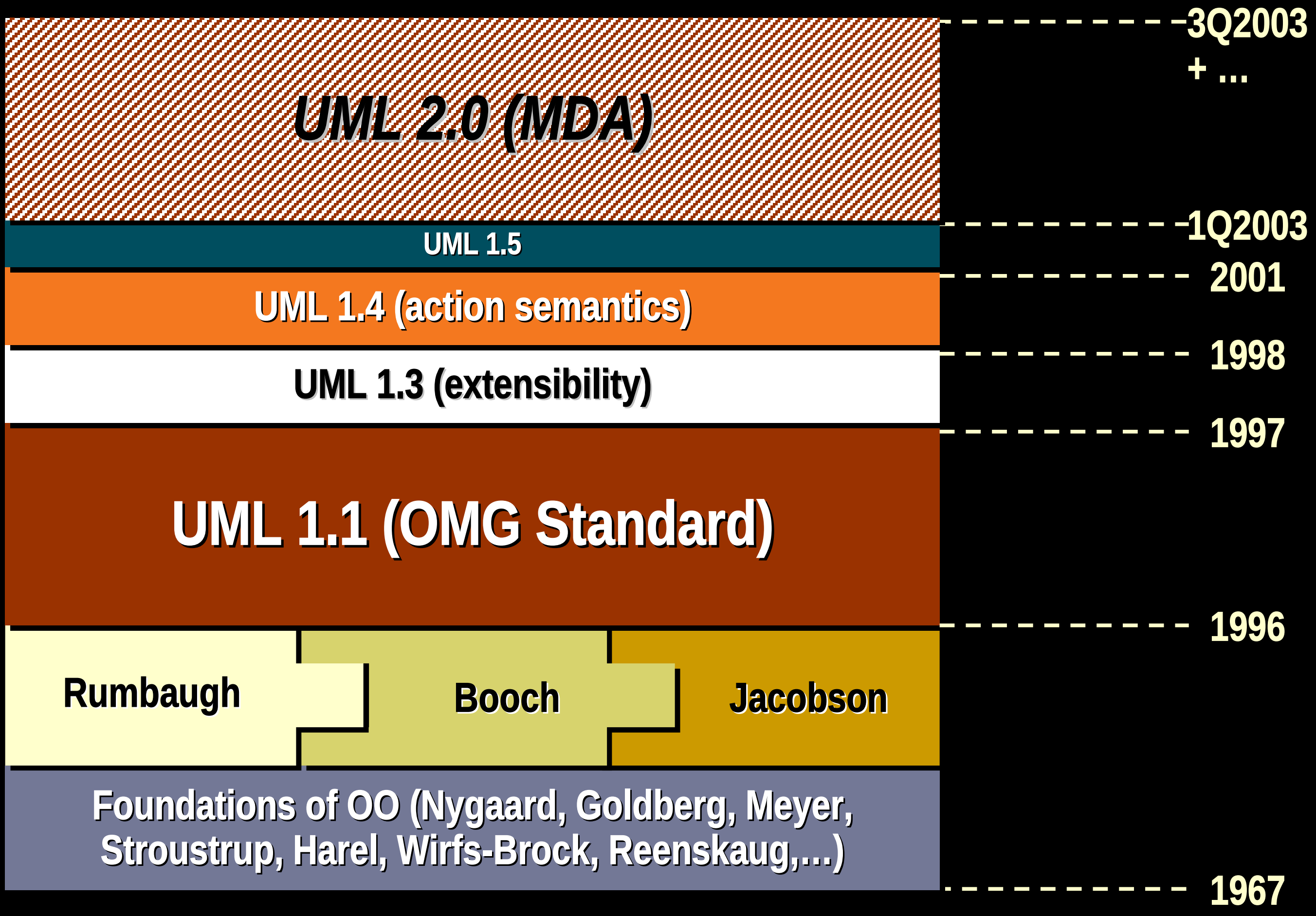
- Set of modeling languages for specific purposes


UML
"bootstrap"
MetaObject Facility (MOF)
MOF "core"
A modeling language for defining modeling languages
General Standard UML
For general OO modeling
Common Warehouse Metamodel (CWM)
For exchanging information about business data
etc.
Real-Time profile
EAI profile
Software process profile
etc.

The "4-Layer" Architecture



UML: The Foundation of MDA

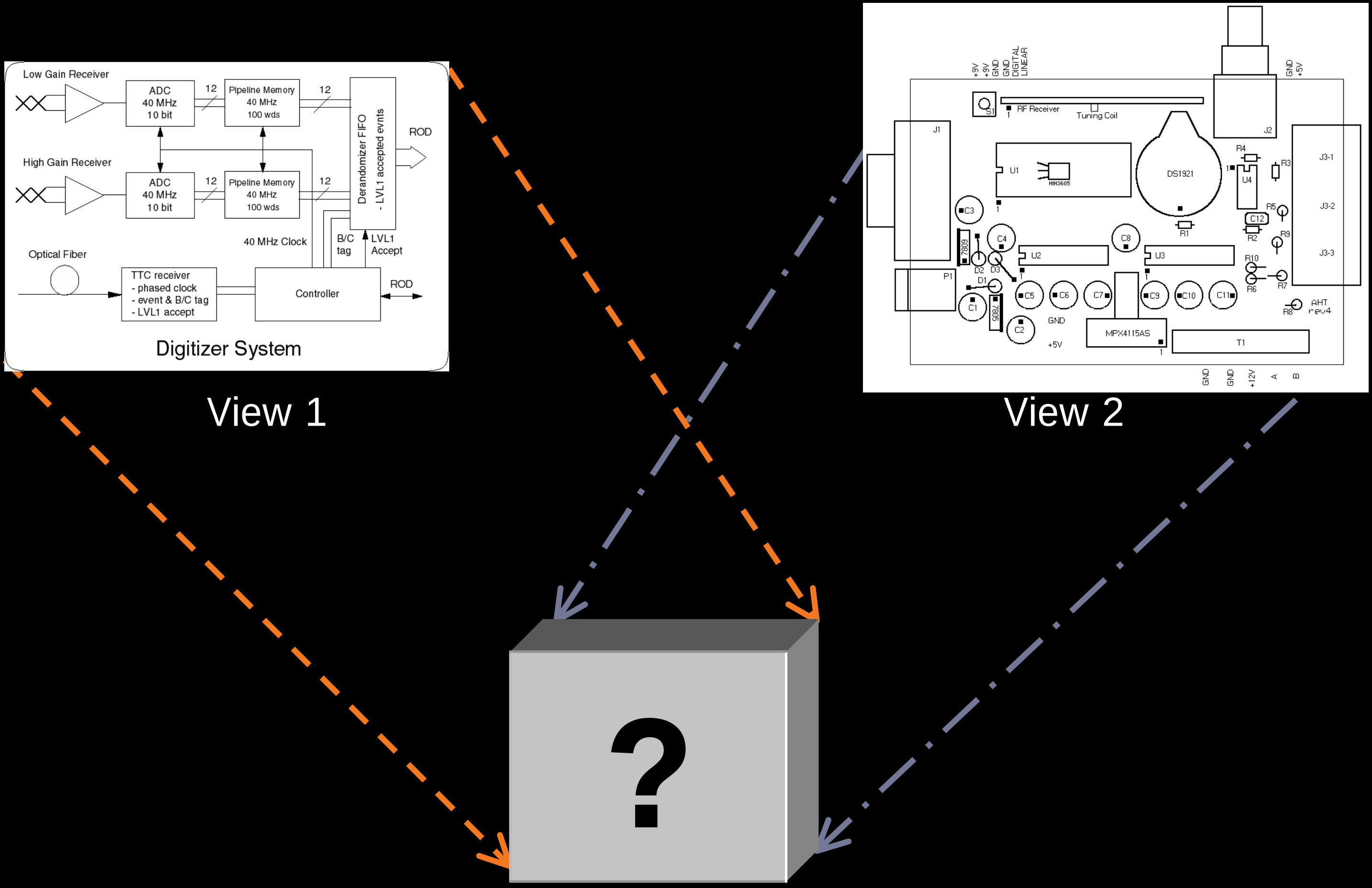


UML 1.x: What Went Right

- Timeliness (meeting a real need)
- Emphasis on semantics as opposed to notation
  - model-based approach (versus view-based)
  - detailed semantic specifications
- Higher-level abstractions beyond most current OO programming language technology
  - state machines and activity diagrams
  - support for specifying inter-object behavior (interactions)
  - use cases
- Customizability (extensibility)

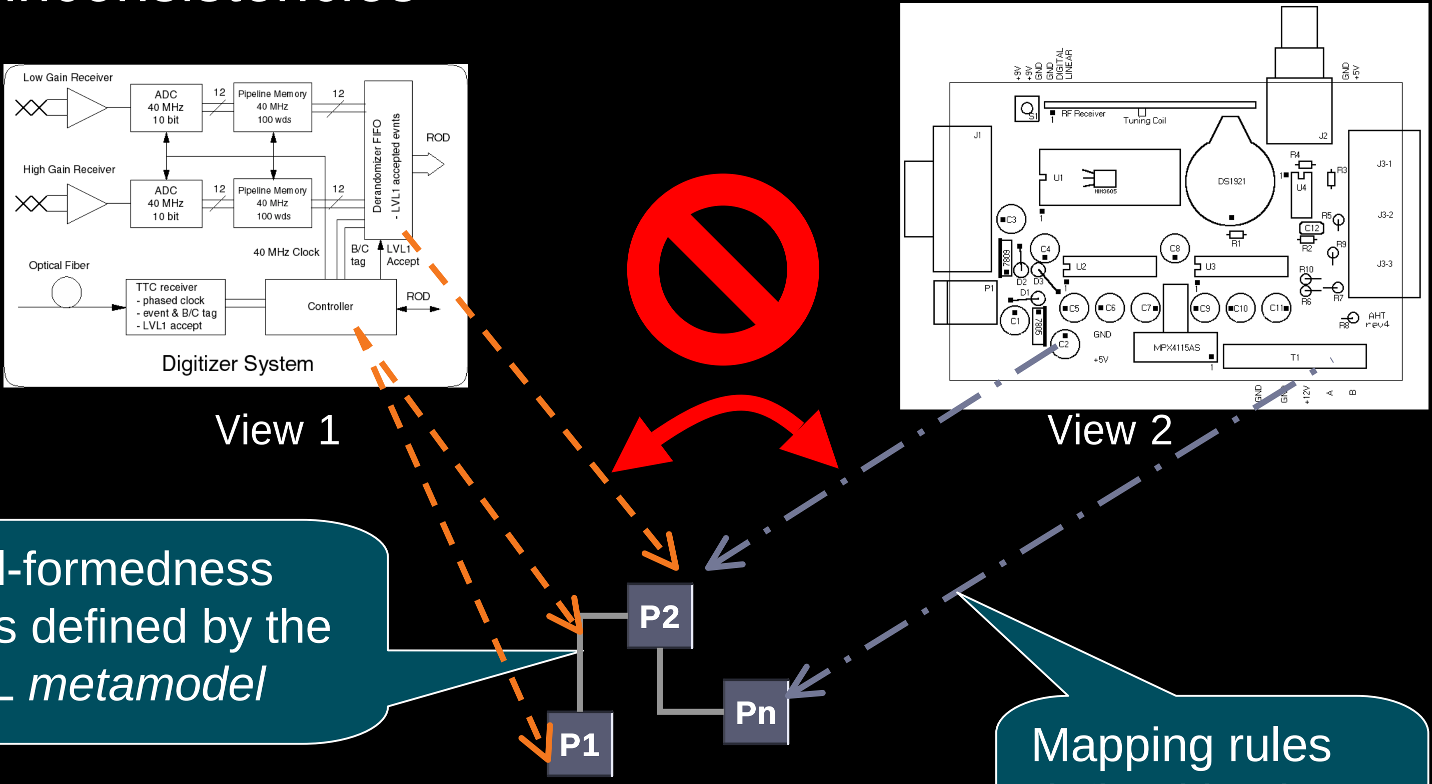
Traditional Approach to Views in Modeling

- Multiple, informally connected views
  - Combined in the final (integration) phase of design



UML Approach: Single Model

- Views are projections of a complete model
  - Continuous integration of views with dynamic detection of inconsistencies

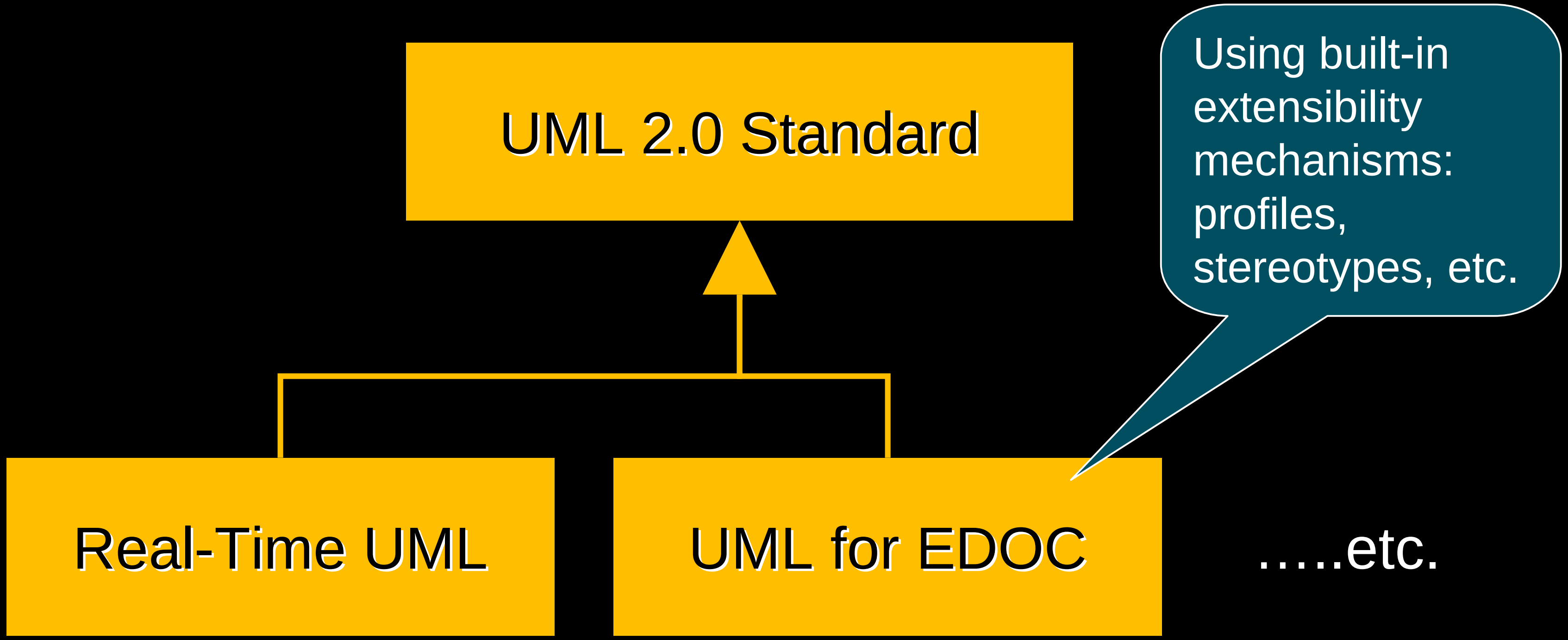


Well-formedness rules defined by the UML *metamodel*

Mapping rules defined by the UML *spec*

Specializing UML

- Goal: avoid the second-system syndrome ("language bloat")
  - UML standard as a basis for a "family of languages"


UML 2.0 Standard
Using built-in extensibility mechanisms: profiles, stereotypes, etc.
Real-Time UML
UML for EDOC
…..etc.

UML 1.x: What Went Wrong?

- Does not fully exploit MDD potential of models
  - E.g., "C++ in pictures"
- Inadequate modeling capabilities
  - Business and similar processes modeling
  - Large-scale systems
  - Non-functional aspects (quality of service specifications)
- Too complex
  - Too many concepts
  - Overlapping concepts
- Inadequate semantics definition
  - Vague or missing (e.g., inheritance, dynamic semantics)
  - Informal definition (not suitable for code generation or executable models)
- No diagram interchange capability
- Not fully aligned with MOF
  - Leads to model interchange problems (XMI)

- Introduction: modeling and software
- Model-Driven Development
- A Critique of UML 1.x
- Requirements for UML 2.0
- Structure of UML 2.0
- Architectural Modeling Capabilities
- Dynamic Semantics
- Interaction Modeling Capabilities
- Activities and Actions
- State Machine Innovations
- Other New Features
- Summary and Conclusion

Sources of Requirements

- MDA (retrofit)
  - Semantic precision
  - Consolidation of concepts
  - Full MOF-UML alignment
- Practitioners
  - Conceptual clarification
  - New features, new features, new features…
- Language theoreticians
  - My new features, my new features, my new features…
  - Why not replace it with my modeling language instead?
- Dilemma: avoiding the “language bloat” syndrome

Formal RFP Requirements

1) Infrastructure – UML internals
   - More precise conceptual base for better MDA support
2) Superstructure – User-level features
   - New capabilities for large-scale software systems
   - Consolidation of existing features
3) OCL – Constraint language
   - Full conceptual alignment with UML
4) Diagram interchange standard
   - For exchanging graphic information (model diagrams)

Infrastructure Requirements

- Precise MOF alignment
  - Fully shared “common core” metamodel
- Refine the semantic foundations of UML (the UML metamodel)
  - Improve precision
  - Harmonize conceptual foundations and eliminate semantic overlaps
  - Provide clearer and more complete definition of instance semantics (static and dynamic)
- Improve extension mechanisms
  - Profiles, stereotypes
  - Support “family of languages” concept

OCL Requirements

- Define an OCL metamodel and align it with the UML metamodel
  - OCL navigates through class and object diagrams $\Rightarrow$ must share a common definition of Class, Association, Multiplicity, etc.
- New modeling features available to general UML users
  - Beyond constraints
  - Ability to express business rules
  - General-purpose query language

Diagram Interchange Requirements

- Ability to exchange graphical information between tools
  - Currently only non-graphical information is preserved during model interchange
  - Diagrams and contents (size and relative position of diagram elements, etc.)

Superstructure Requirements (1 of 2)

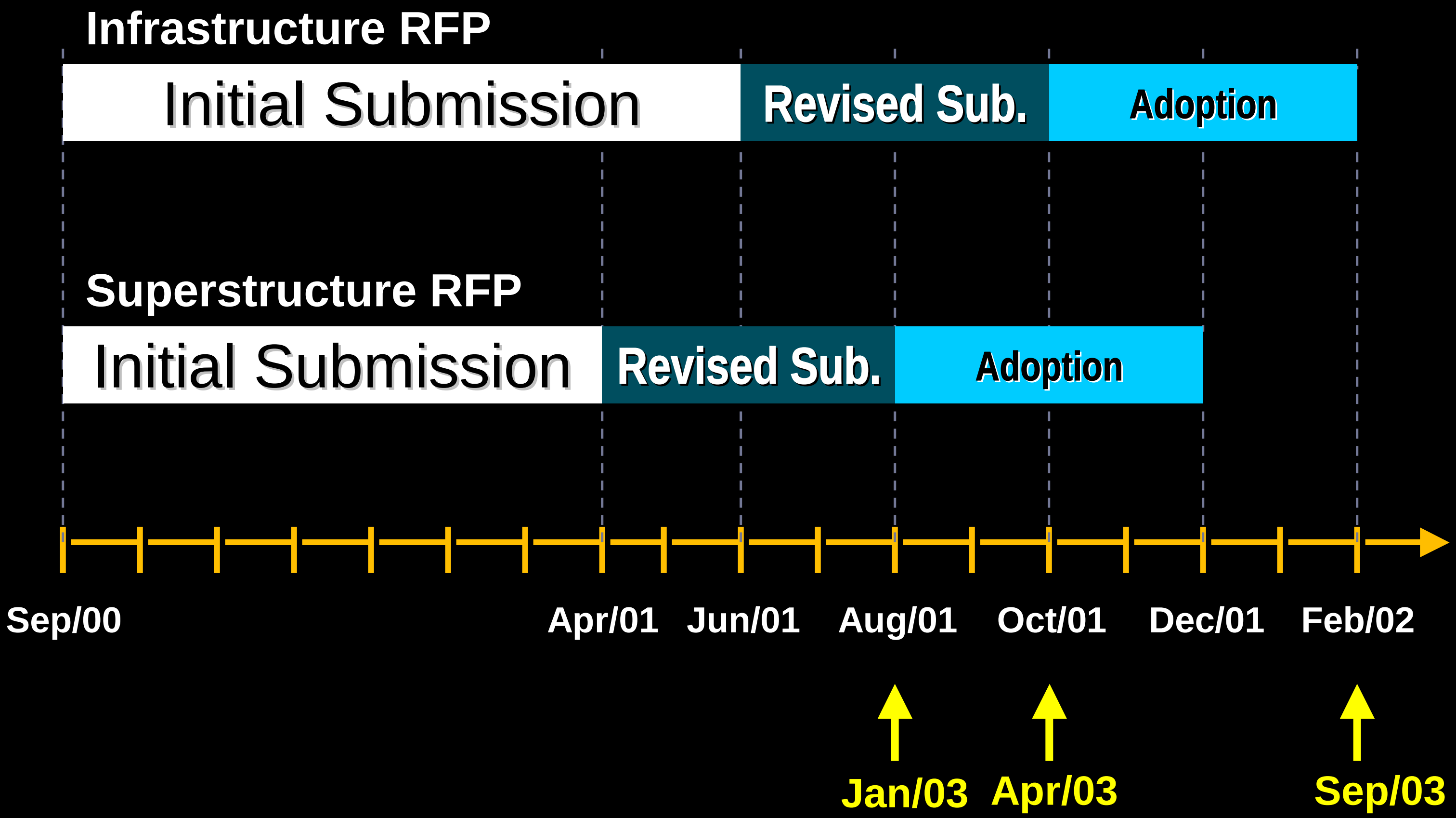
- More direct support for architectural modeling
  - Based on existing architectural description languages (UML-RT, ACME, etc.)
  - Reusable interaction specifications (UML-RT protocols)
- Behavior harmonization
  - Generalized notion of behavior and causality
  - Support choice of formalisms for specifying behavior
- Hierarchical interactions modeling
- Better support for component-based development
- More sophisticated activity graph modeling
  - To better support business process modeling

Superstructure Requirements (2 of 2)

- New statechart capabilities
  - Better modularity
- Clarification of semantics for key relationship types
  - Association, generalization, realization, etc.
- Remove unused and ill-defined modeling concepts
- Clearer mapping of notation to metamodel
- Backward compatibility
  - Support 1.x style of usage
  - New features only if required

UML 2.0 Schedule

- Single 2.0 standard at the end:



UML 2.0 Standardization Sequence

- Complex adoption process
  - Step 1: Endorsement by OMG architecture board (June 2003)
  - Step 2: OMG membership vote (September 2003)
  - Step 3: OMG BoD endorsement (October 2003)
    - Spec becomes “Adopted Specification”
- UML 2 articles, books, best-guess tools are now appearing
  - Step 4: UML 2.0 Finalization Task Force (FTF) (June 2004?)
  - Step 5: OMG membership vote (September 2004?)
  - Step 6: OMG BoD endorsement (October 2004?)
    - Spec becomes “Available Specification” (i.e., a standard)
- When is UML 2.0 an “official” standard?
  - Two main phases: October 2003 and October 2004
  - It will change up to October 2004 (currently >300 open issues)

- Introduction: modeling and software
- Model-Driven Development
- A Critique of UML 1.x
- Requirements for UML 2.0
- Foundations of UML 2.0
- Architectural Modeling Capabilities
- Dynamic Semantics
- Interaction Modeling Capabilities
- Activities and Actions
- State Machine Innovations
- Other New Features
- Summary and Conclusion

First, the Politics

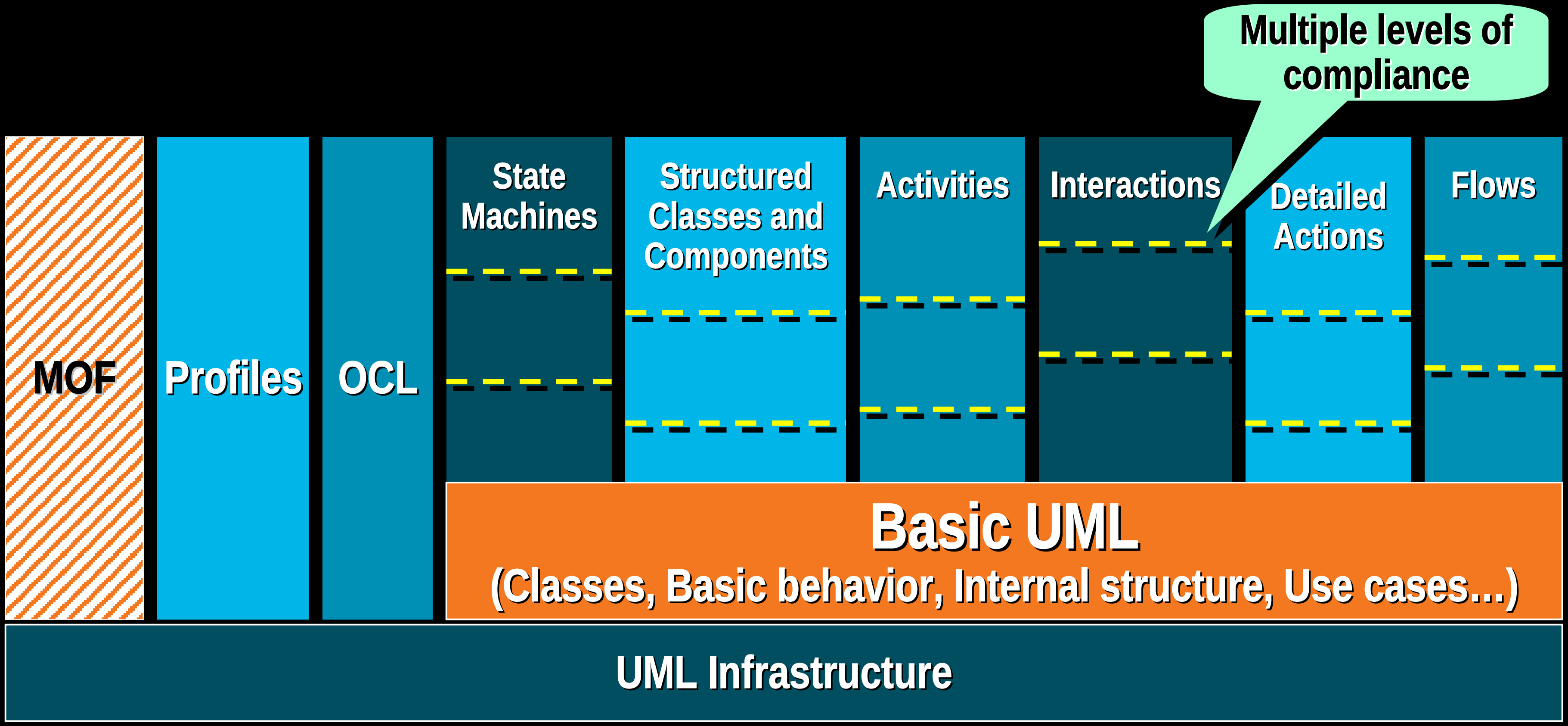
- Multiple competing submissions (5)
  - Most involved consortia of companies representing both UML tool vendors and UML users
  - One prominent (800-lb gorilla) submission team (“U2P”) with most of the major vendors (Rational, IBM, Telelogic, ...) and large user companies (Motorola, HP, Ericsson…)
- Over time:
  - Some submissions lapsed
  - Some submissions were merged into the U2P
  - Only one final submission
  - This submission was adopted

U2P Submission Approach

- Evolutionary rather than revolutionary
- Improved precision of the infrastructure
- Small number of new features
- New feature selection criteria
  - Required for supporting large industrial-scale applications
  - Non-intrusive on UML 1.x users (and tool builders)
- Backward compatibility with 1.x

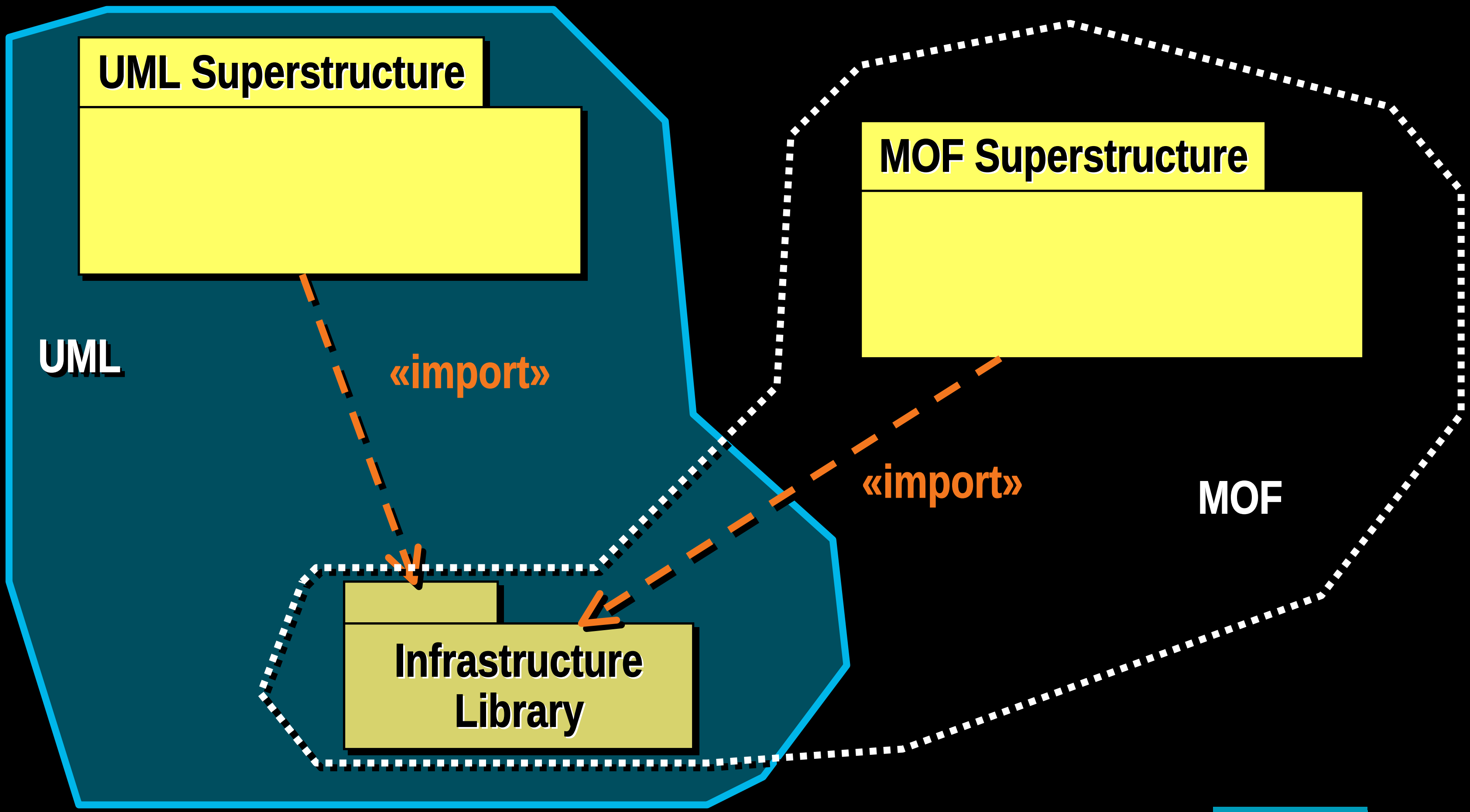
General Language Structure

- A core language + optional “sub-languages”
  - Enables flexible subsetting for specific needs
  - Users can “grow into” more advanced capabilities


Multiple levels of compliance
MOF
Profiles
OCL
State Machines
Structured Classes and Components
Activities
Interactions
Detailed Actions
Flows
Basic UML
(Classes, Basic behavior, Internal structure, Use cases…)
UML Infrastructure

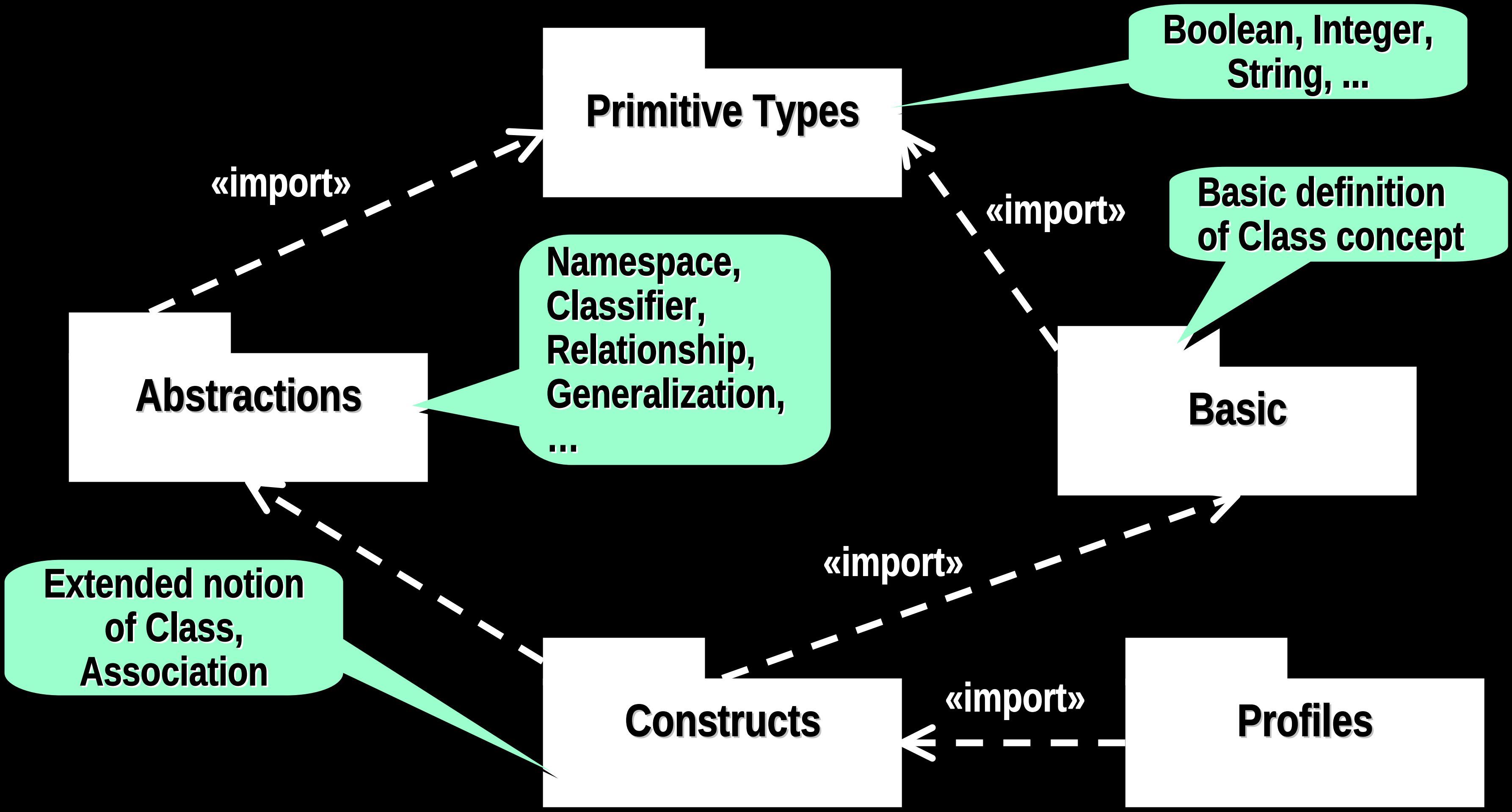
UML-MOF Alignment

- Shared conceptual base
  - MOF: language for defining modeling languages
  - UML: general purpose modeling language



Infrastructure Library

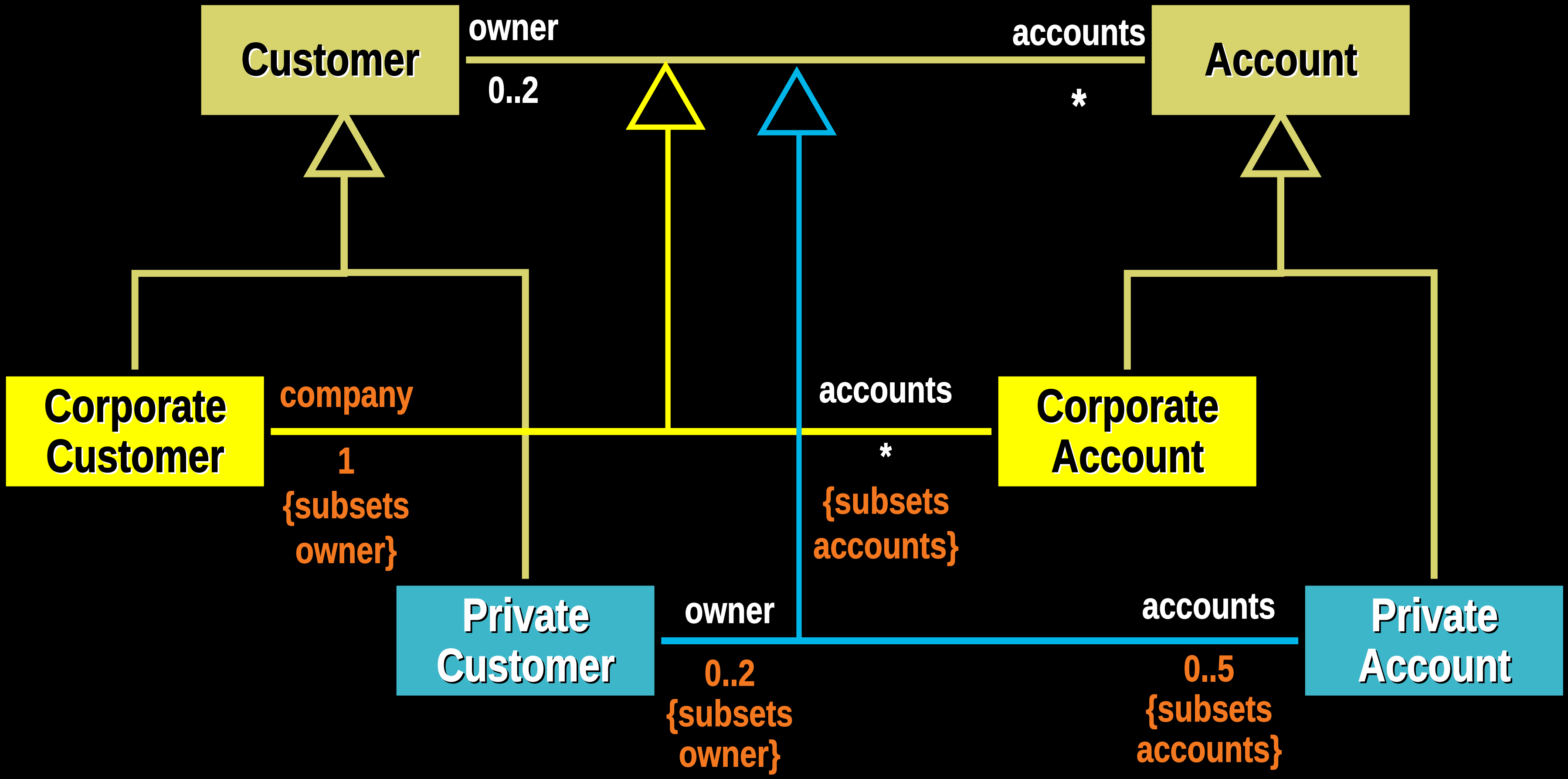
- Shared between MOF and UML



Primitive Types
Boolean, Integer, String, ...
«import»
«import»
Basic definition of Class concept
Namespace, Classifier, Relationship, Generalization, ...
Abstractions
Basic
«import»
Extended notion of Class, Association
Constructs
«import»
Profiles

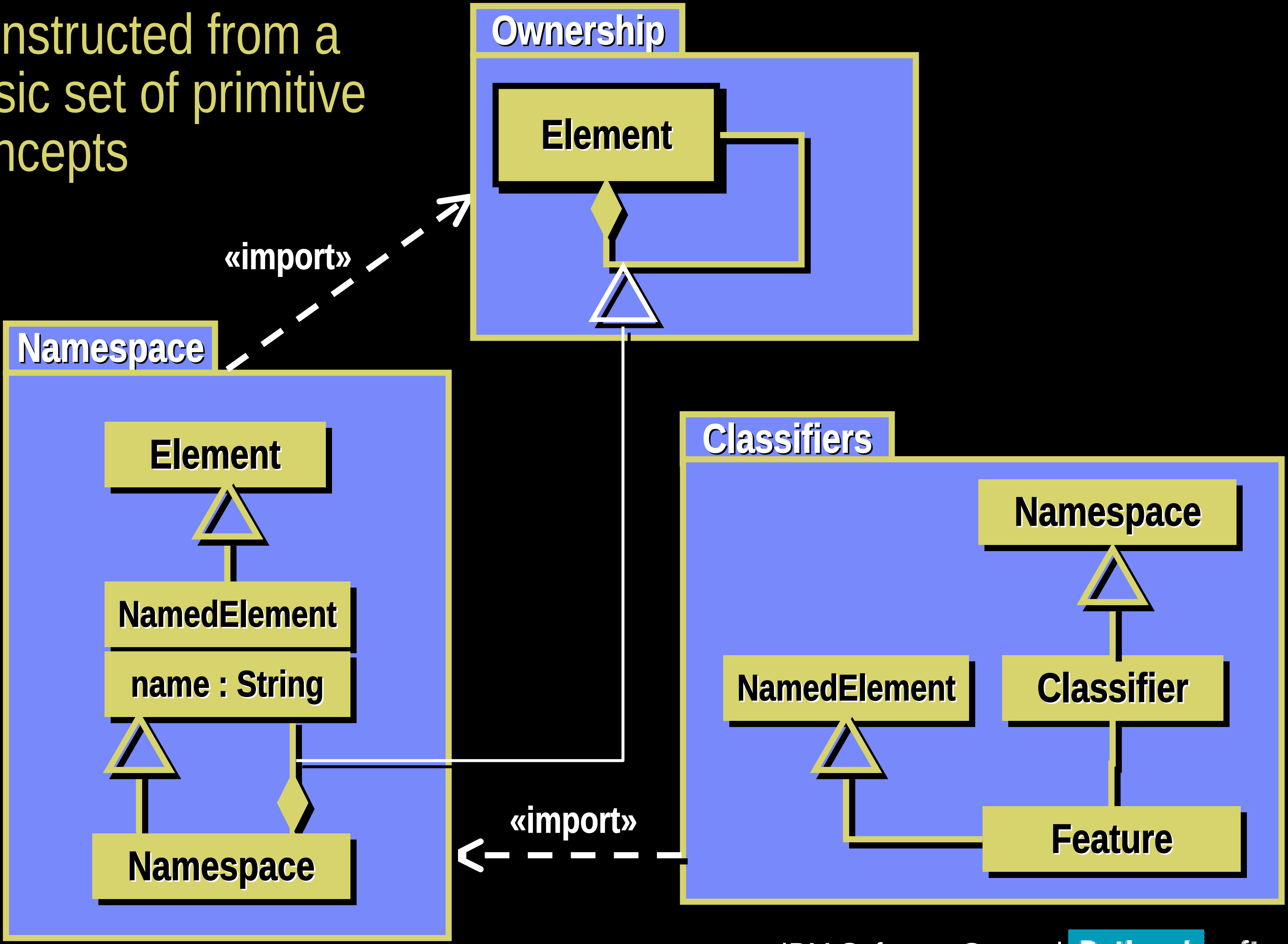
Association Specialization

- Also used widely in the definition of the UML metamodel
  - Avoids covariance problems



Example: Classifier Definition

- Constructed from a basic set of primitive concepts

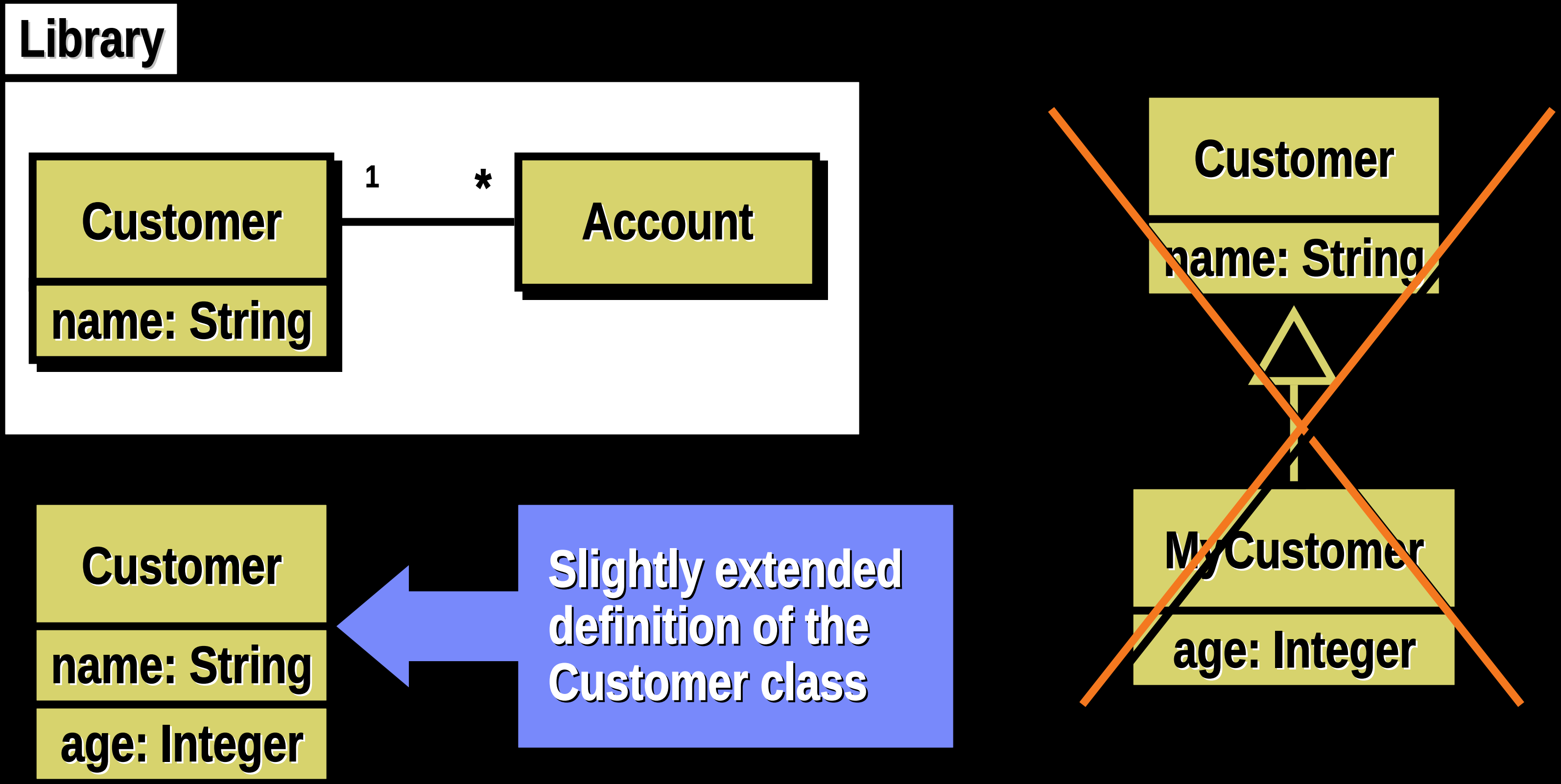


Infrastructure: Consolidation of Concepts

- The re-factoring of the UML metamodel into fine-grained independent concepts
  - Eliminates semantic overlap
  - Provides a better foundation for a precise definition of concepts and their semantics
  - Conducive to MDD

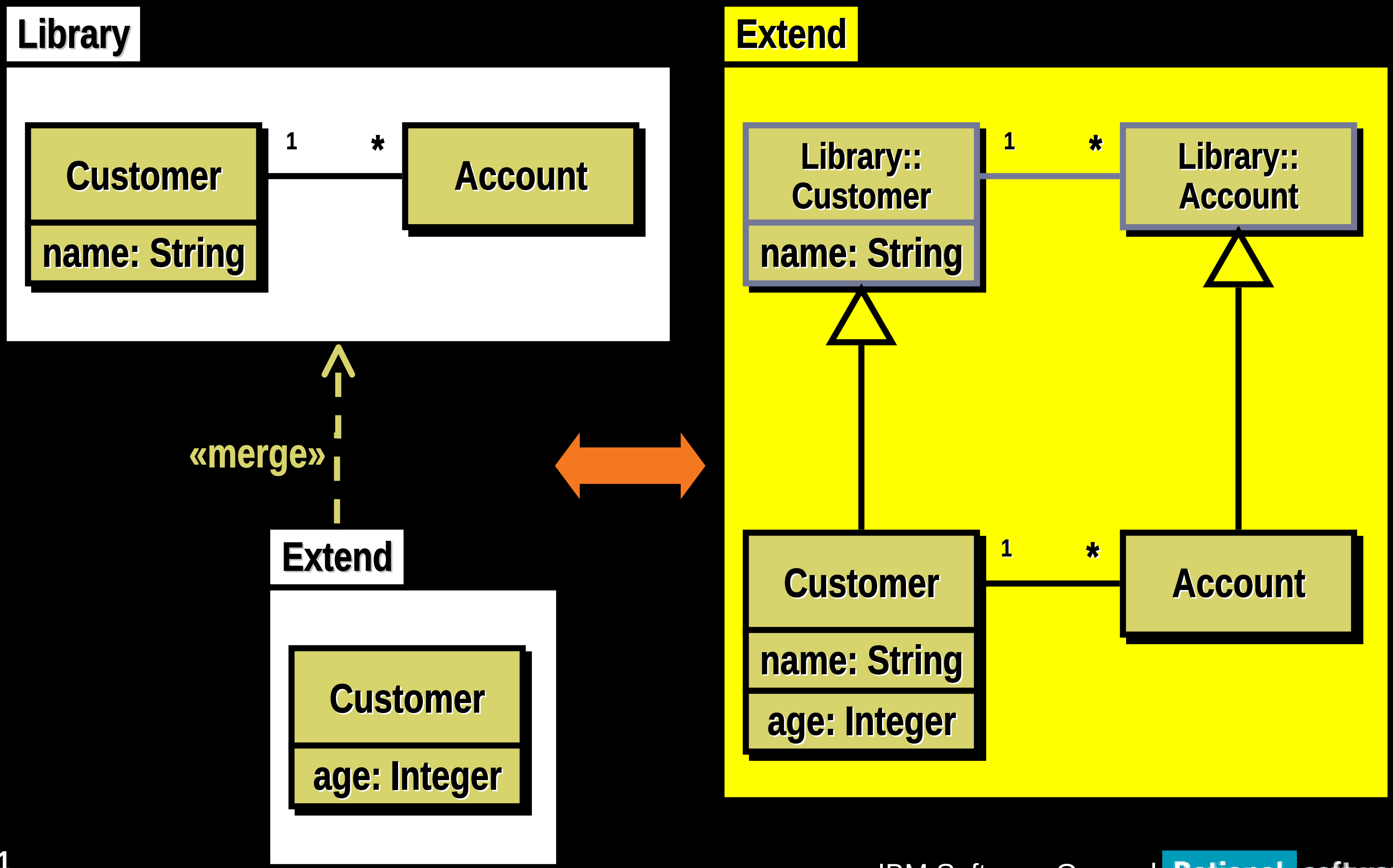
Package Merge: Motivation

- In some cases we would like to modify a definition of a class without having to define a subclass
  - To retain all the semantics (relationships, constraints, etc.) of the original



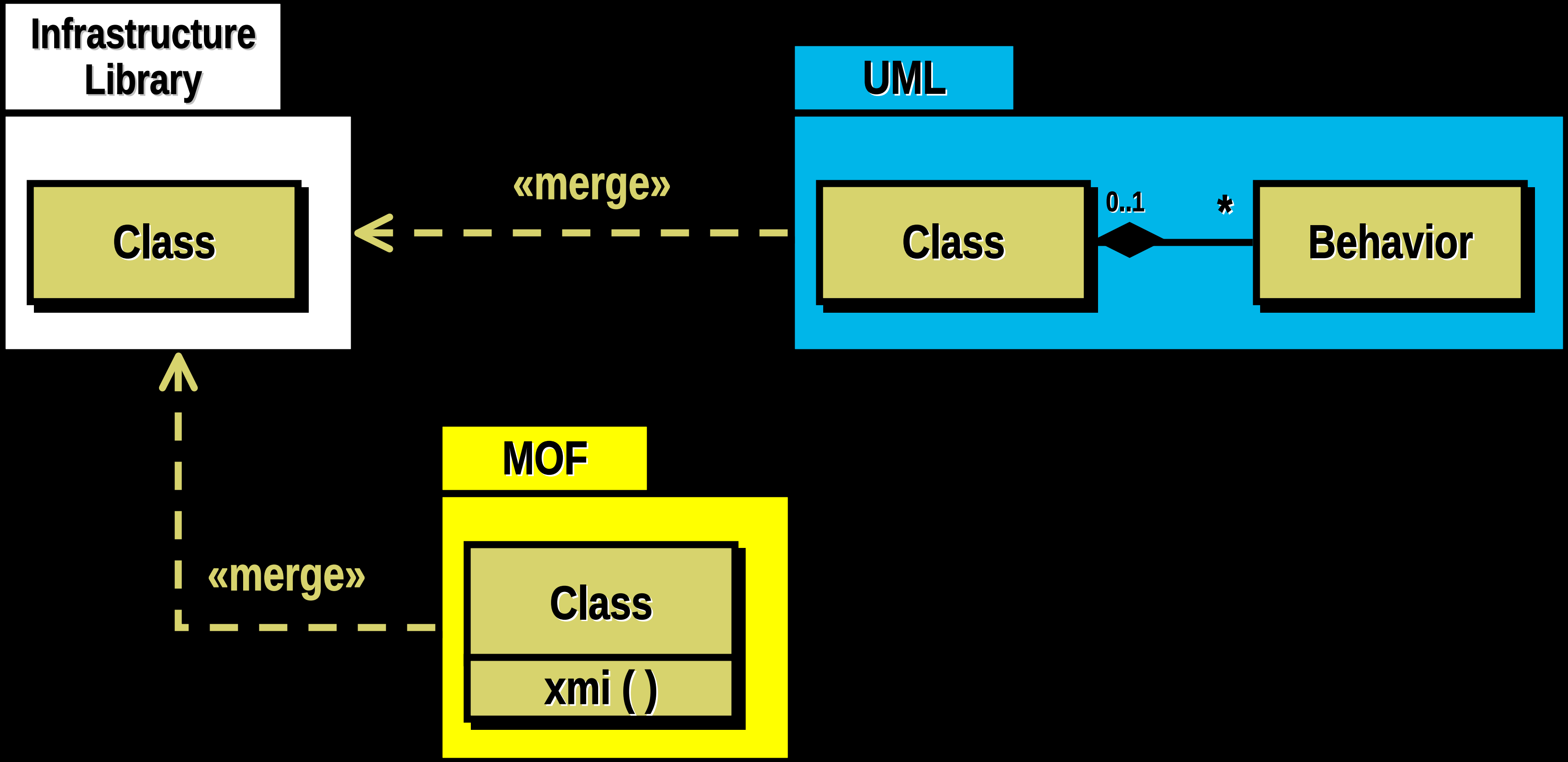
Package Merge: Semantics

- Incremental redefinition of concepts



Package Merge: Metamodel Usage

- Enables common definitions for shared concepts with the ability to extend them according to need
  - E.g. MOF and UML definitions of Class



Summary: Foundations

- The language has been restructured and modularized
  - Set of specialized languages
  - Multiple levels of sophistication
- There have been significant “under the hood” changes to the UML metamodel
  - More precise language definition (suitable for MDD)
  - Much semantic overlap eliminated

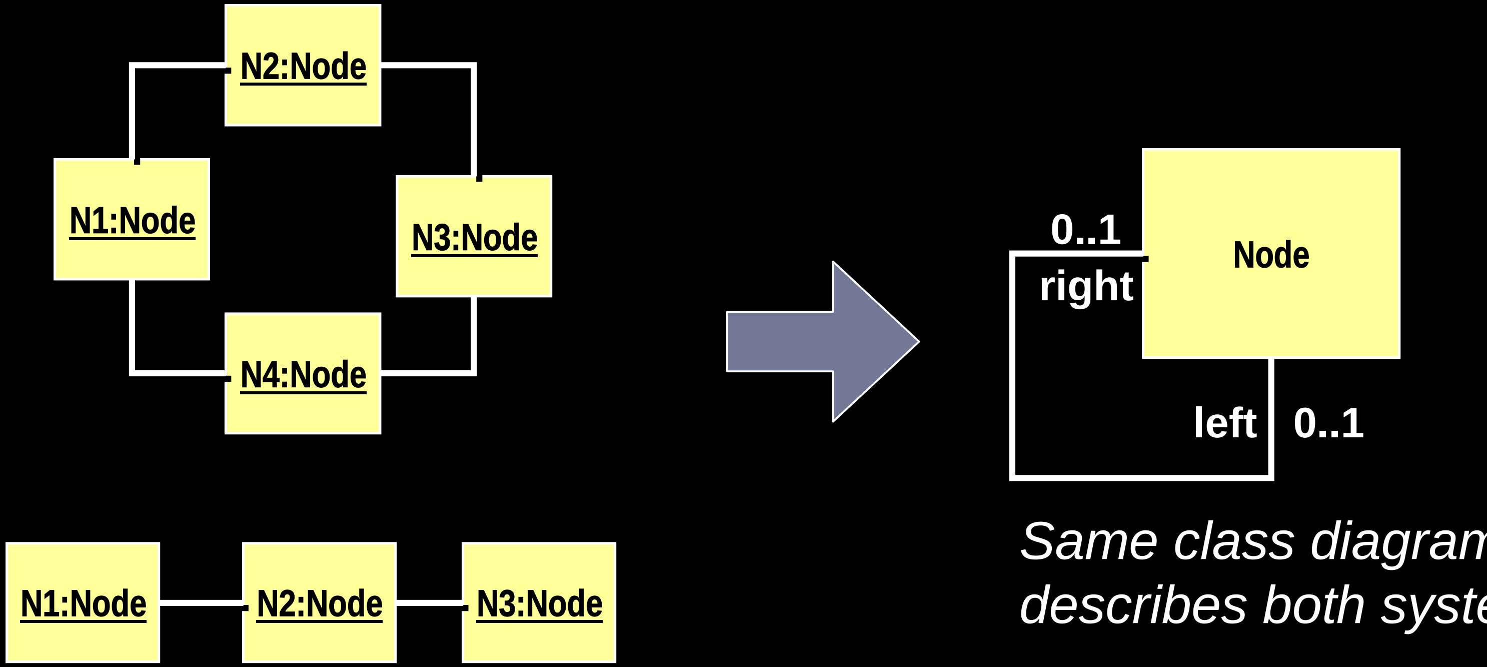
- Introduction: modeling and software
- Model-Driven Development
- A Critique of UML 1.x
- Requirements for UML 2.0
- Foundations of UML 2.0
- Architectural Modeling Capabilities
- Dynamic Semantics
- Interaction Modeling Capabilities
- Activities and Actions
- State Machine Innovations
- Other New Features
- Summary and Conclusion

Structured Classes: Background

- Intended for architectural modeling
  - Concept of objects with internal and external structure (architectural objects)
  - Used primarily for modeling complex systems/subsystems
- Desired structure is asserted rather than constructed
  - Class constructor automatically creates desired structures
  - Class destructor automatically cleans up
  - Major boost to expressiveness, product reliability, developer productivity
- Heritage: architectural description languages (ADLs)
  - UML-RT profile: Selic and Rumbaugh (1998)
    - ROOM: Selic et al. (1994)
  - ACME: Garlan et al. (1997)
  - SDL (ITU-T standard Z.100)

Aren't Class Diagrams Sufficient?

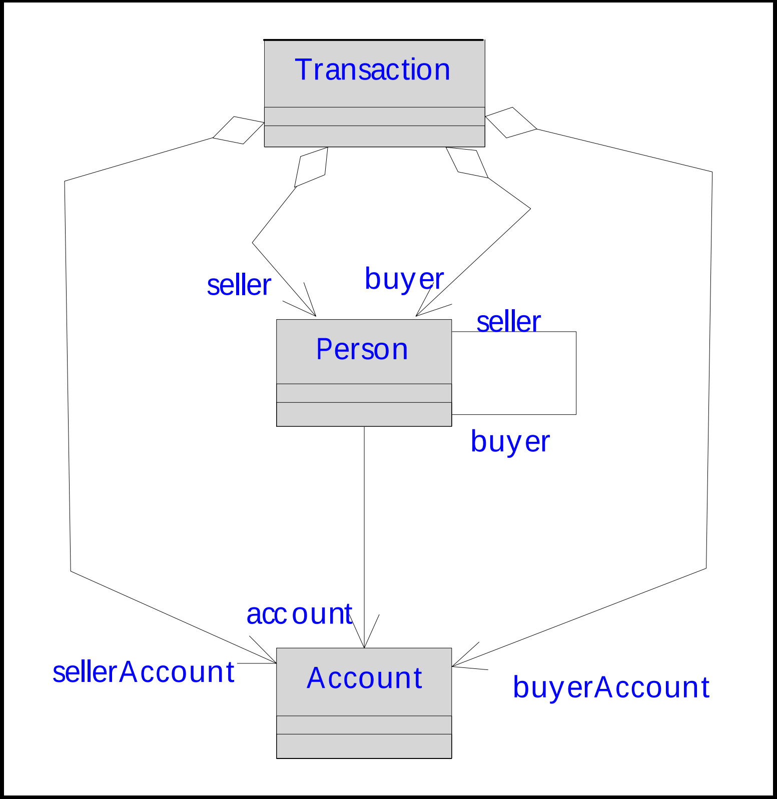
- No!
  - Because they abstract out certain specifics, class diagrams are not suitable for performance analysis
- Need to model structure at the instance/role level

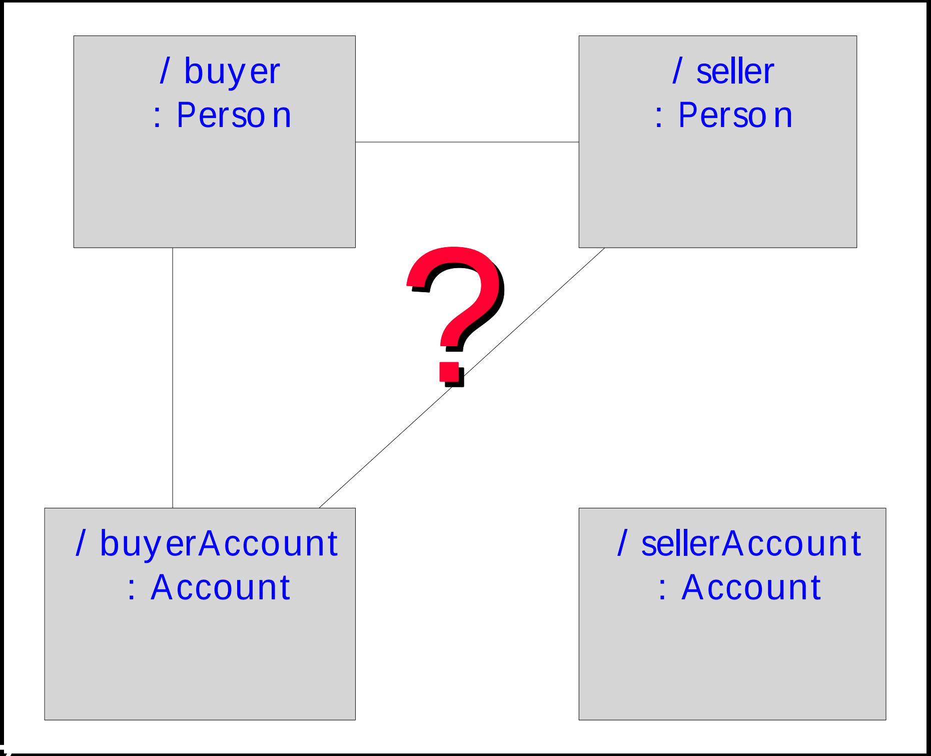


Same class diagram describes both systems!

Structure Modelling Is Not Class Modelling

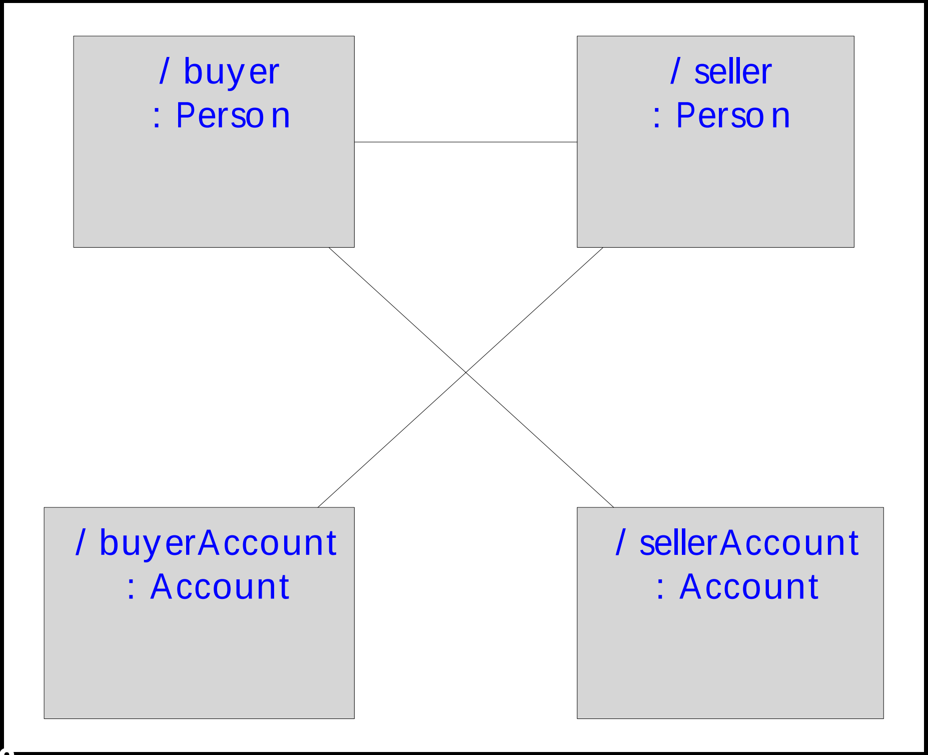
- Transaction example:
  - Two Persons
  - Each Person has an Account
  - Who owns which Account?

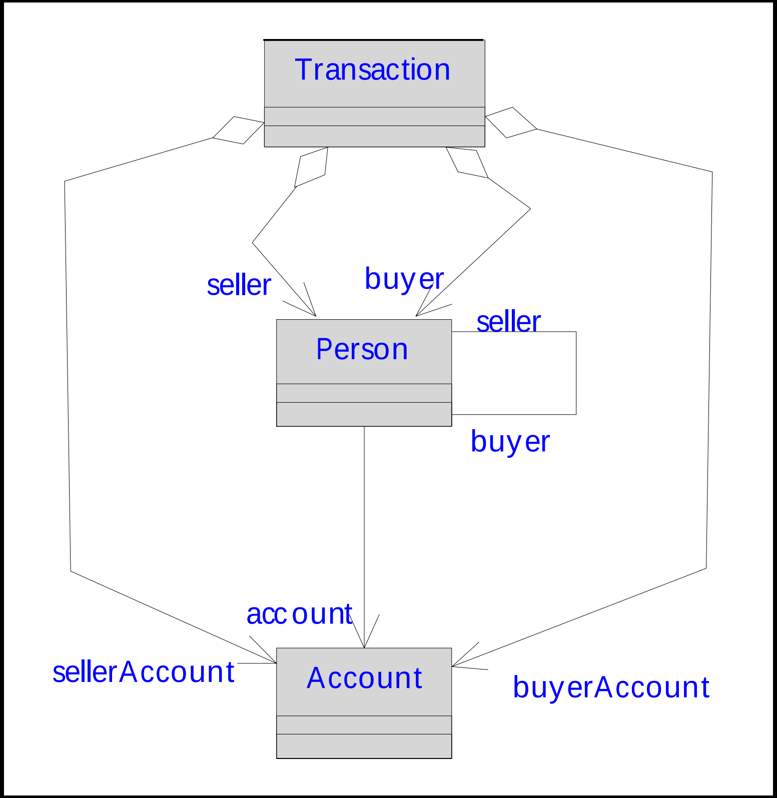




Structure Modelling Is Not Class Modelling

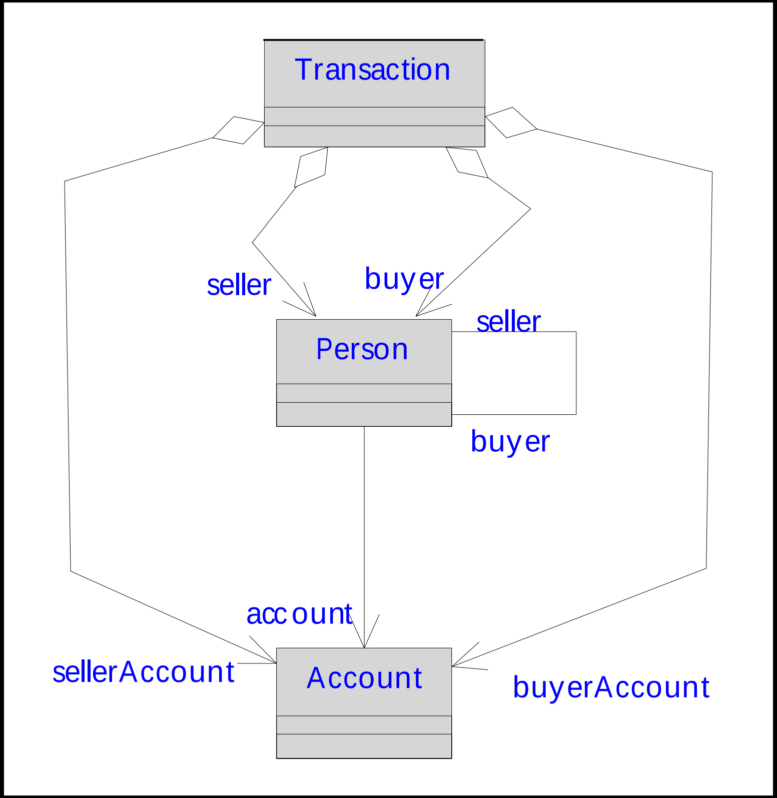
- Transaction example:
  - Two Persons
  - Each Person has an Account
  - Who owns which Account?

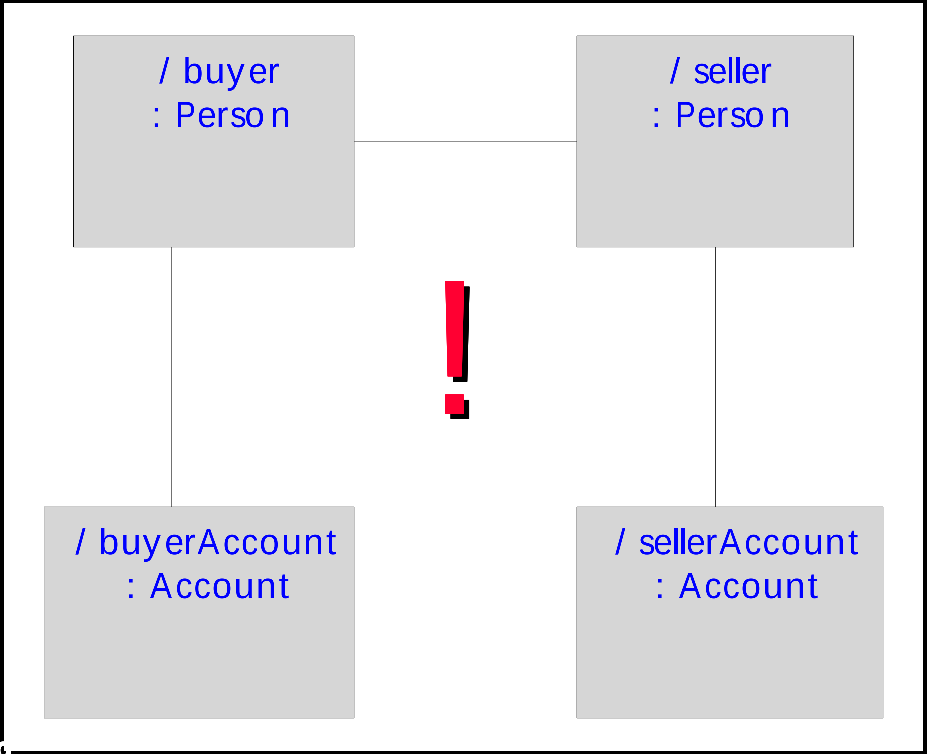




Structure Modelling Is Not Class Modelling

- Transaction example:
  - Two Persons
  - Each Person has an Account
  - Who owns which Account?

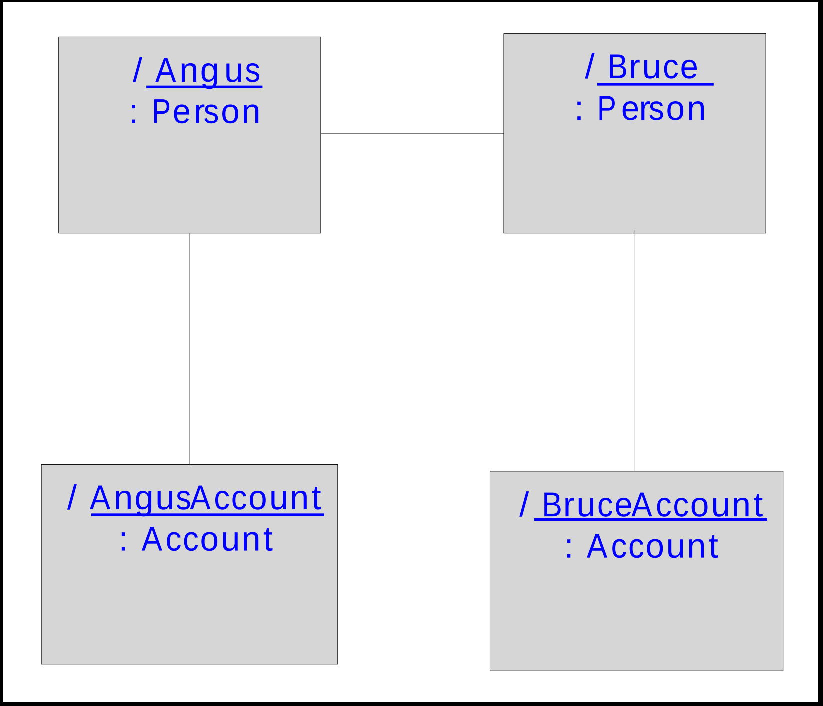




- Class models show "for all" properties
- Structure models show architectural relationships

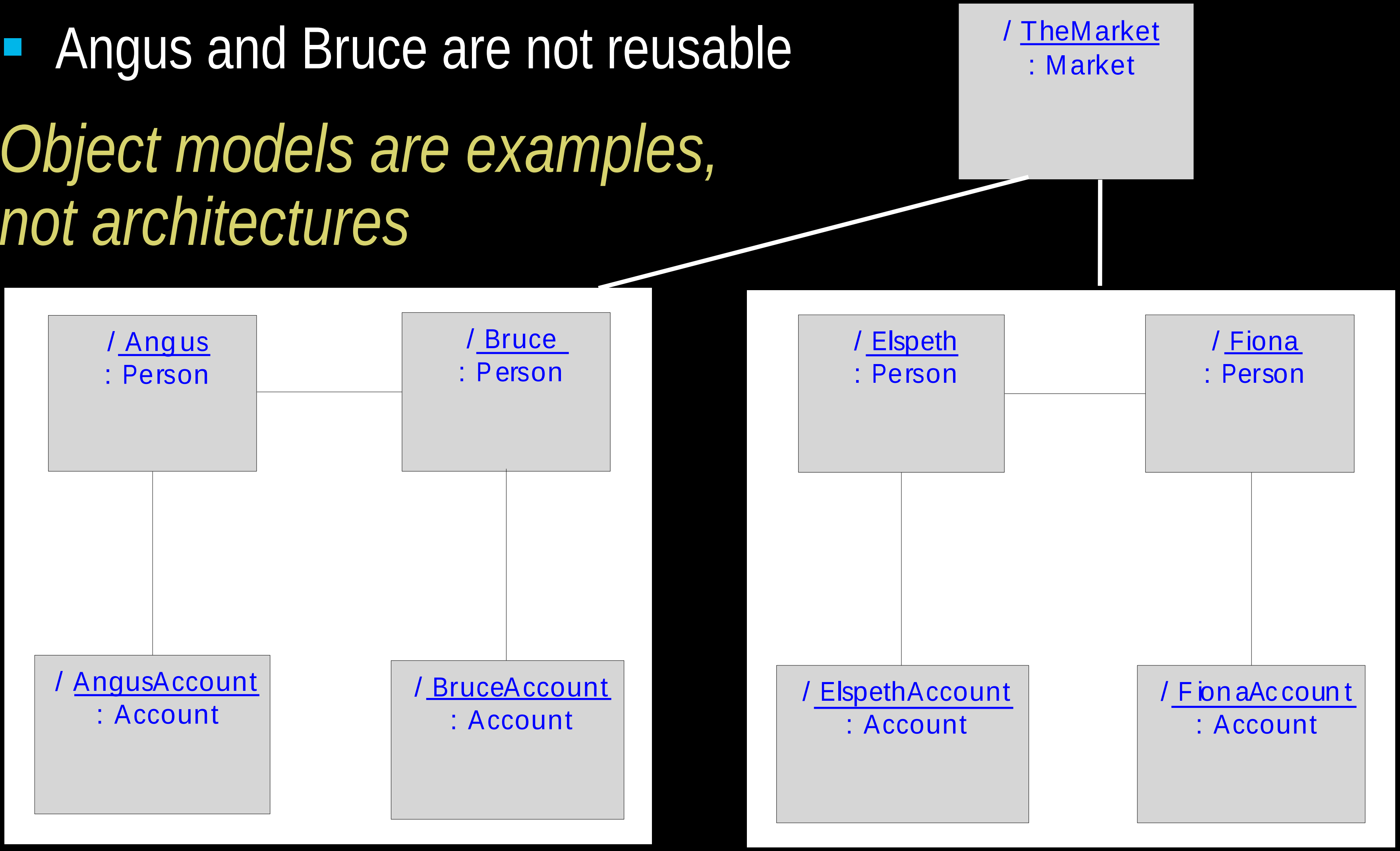
Structure Modelling Is Not Object Modelling

- Object models show completely reified objects
  - Angus and Bruce are not reusable



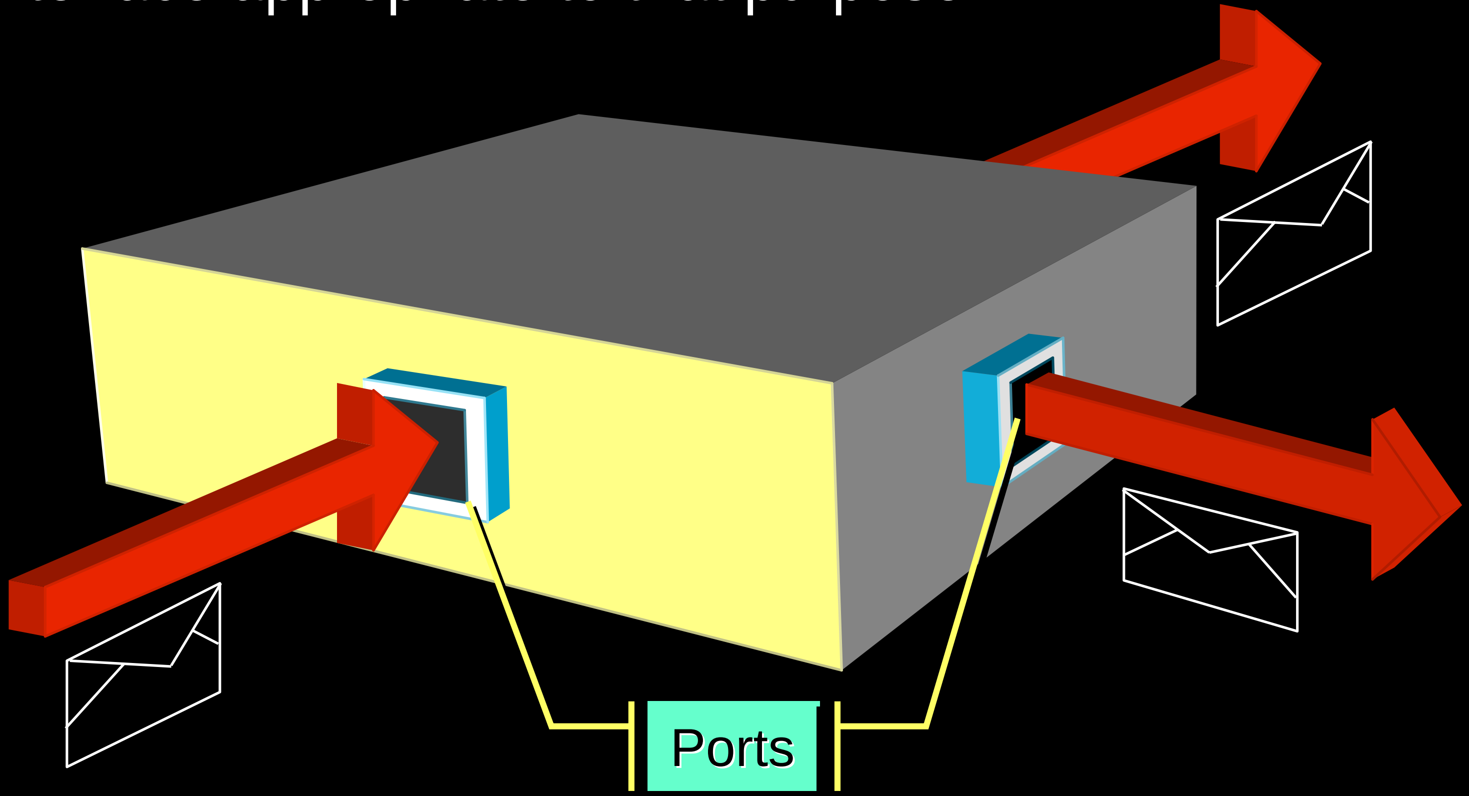
Structure Modelling Is Not Object Modelling

- Object models show completely reified objects
  - Angus and Bruce are not reusable
- *Object models are examples, not architectures*



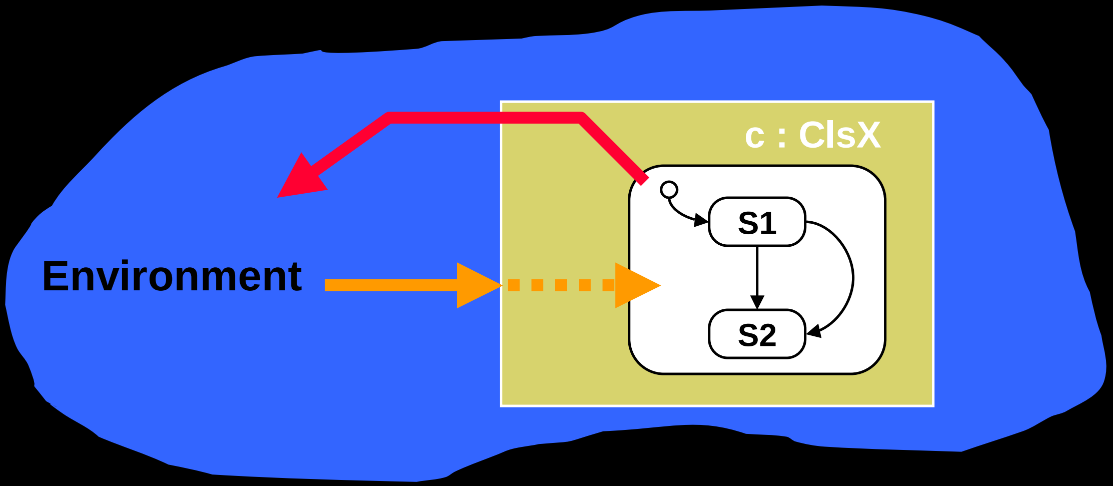
Structured Classes: External Structure

- Complex objects with multiple "faces"
  - Multiple interaction points: *ports*
  - Each port is dedicated to a specific purpose and presents the interface appropriate to that purpose



Ports: Two-Way Encapsulation

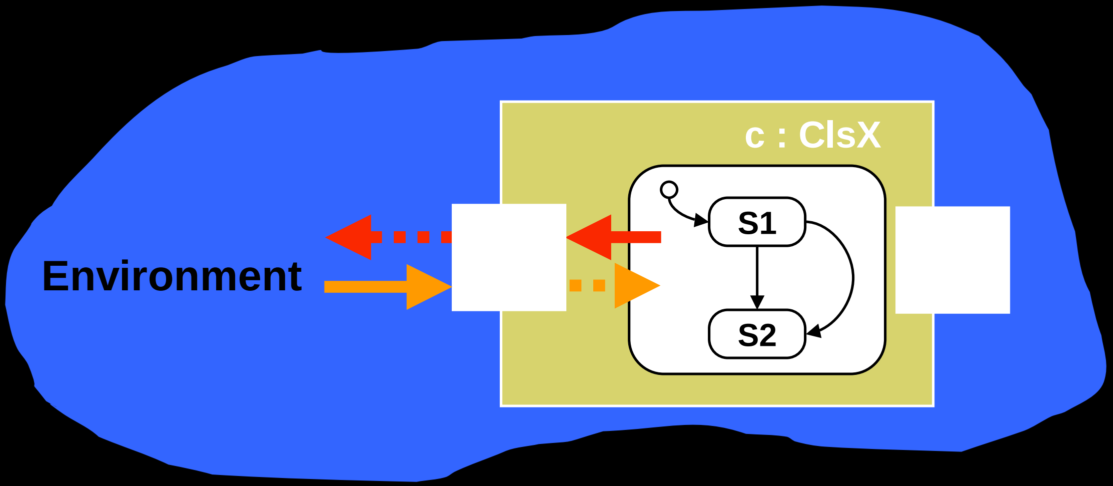
- Boundary objects that
  - help separate different (possibly concurrent) interactions
  - fully isolate an object's internals from its environment



"There are very few problems in computer science that cannot be solved by adding an extra level of indirection"

Ports: Two-Way Encapsulation

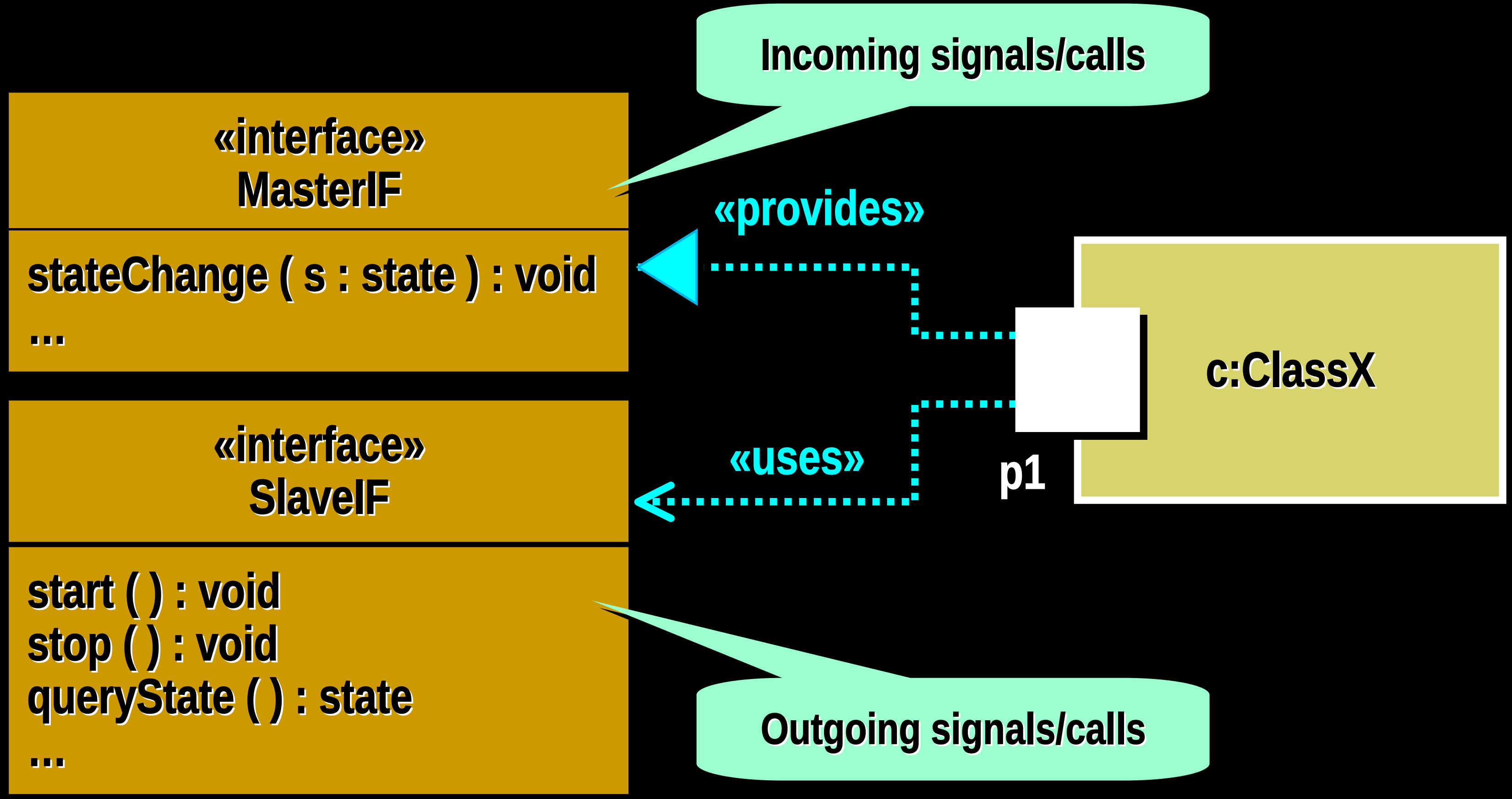
- Boundary objects that
  - help separate different (possibly concurrent) interactions
  - fully isolate an object's internals from its environment



"There are very few problems in computer science that cannot be solved by adding an extra level of indirection"

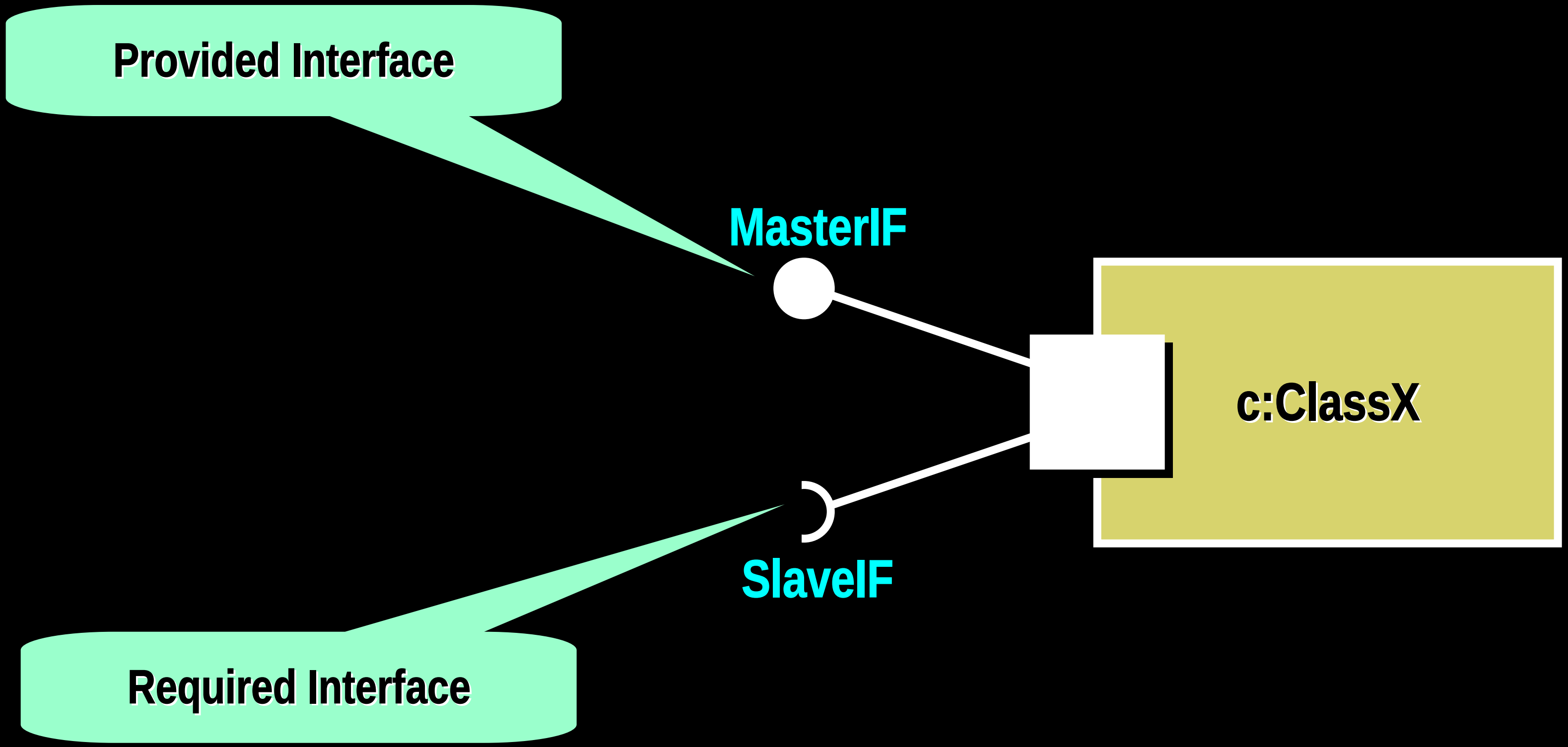
Port Semantics

- A port can support multiple interface specifications
  - Provided interfaces (what the object can do)
  - Required interfaces (what the object needs to do its job)



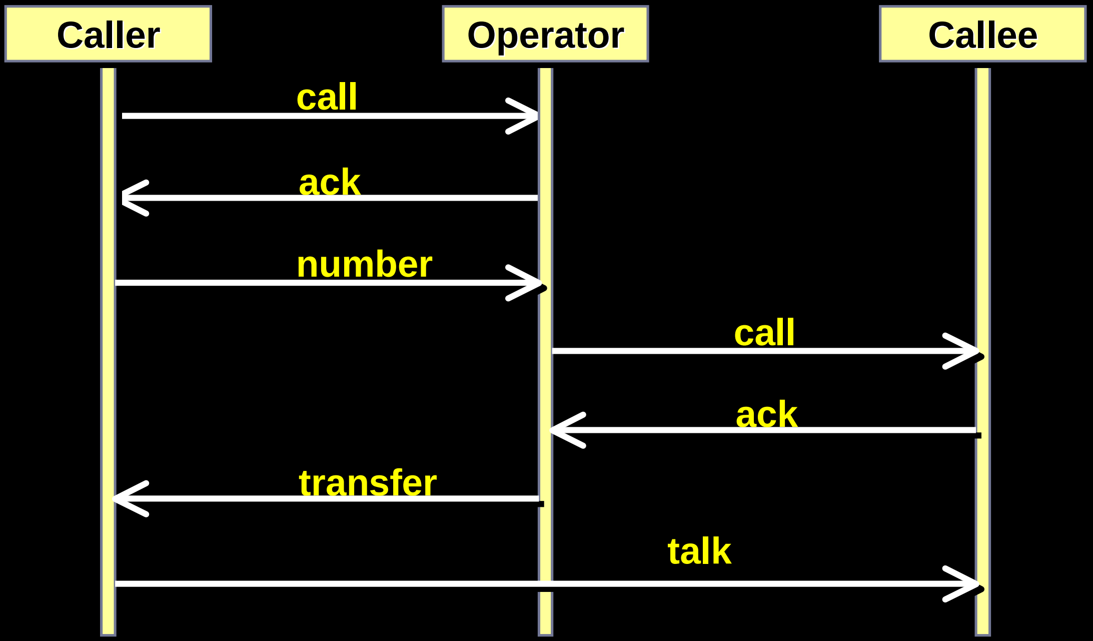
Ports: Alternative Notation

- Shorthand “lollipop” notation with 1.x backward compatibility



Protocols: Reusable Interaction Sequences

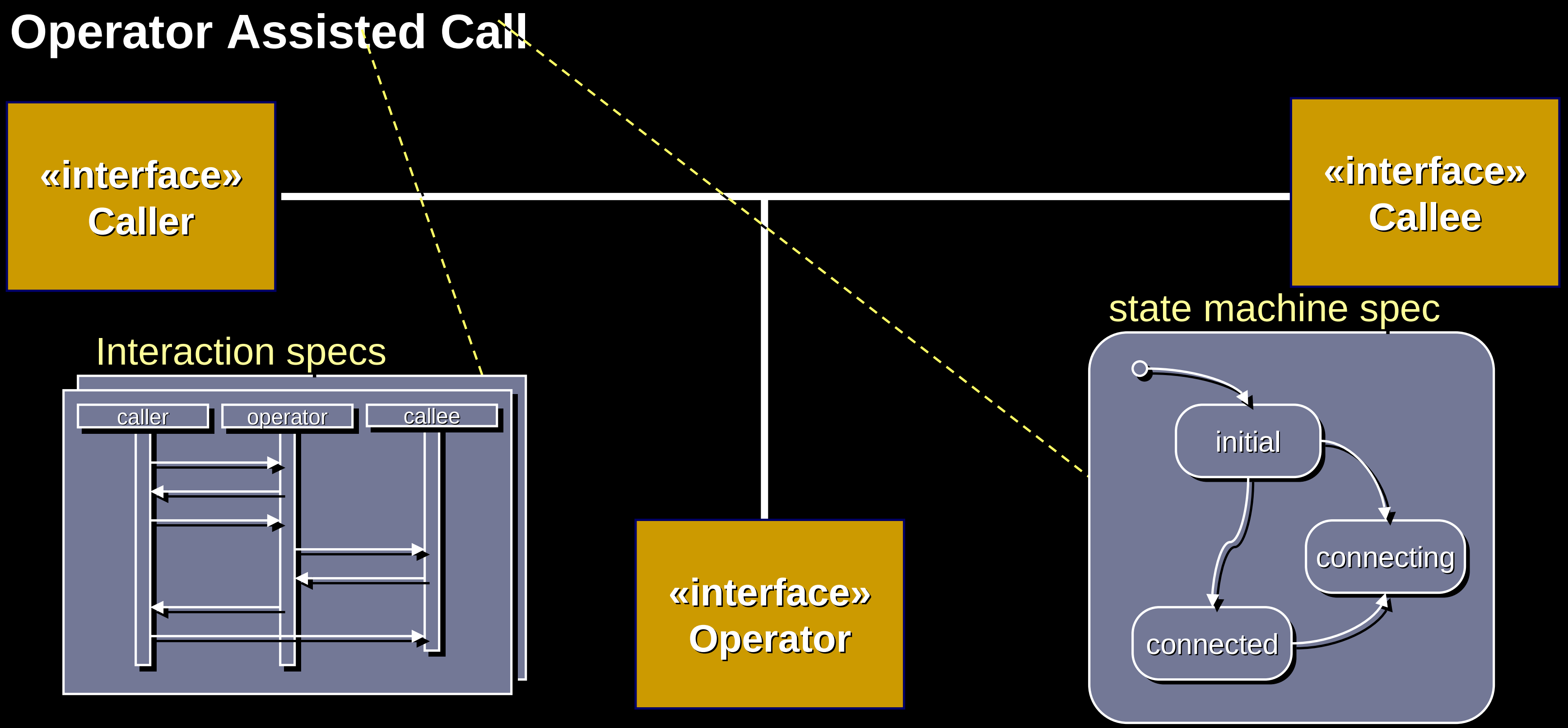
- Communication sequences that
  - always follow a pre-defined dynamic order
  - occur in different contexts with different specific participants



- Important architectural tool
  - Defines valid interaction patterns between architectural elements

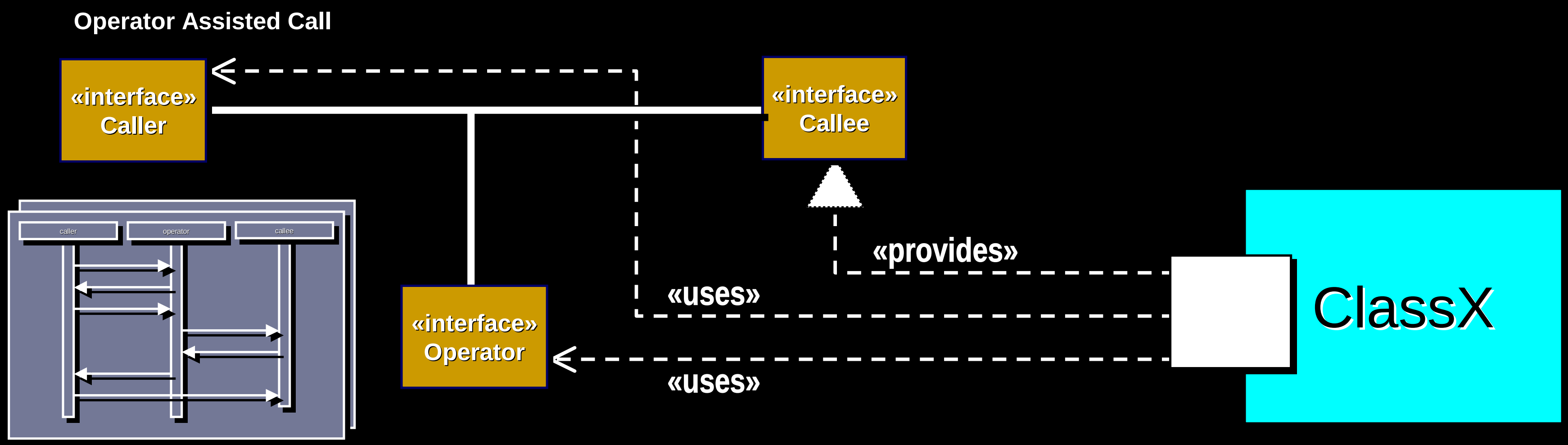
Modeling Protocols with UML 2.0

- Modeled by a set of interconnected interfaces whose features are invoked according to a formal behavioral specification
  - Based on the UML collaboration concept
  - May be refined using inheritance



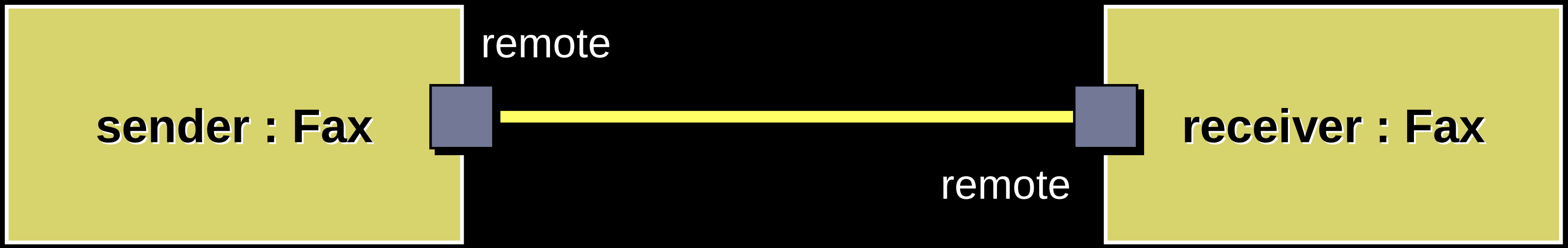
Associating Protocols with Ports

- Ports play individual protocol roles
  - Ports assume the protocol roles implied by their provided and required interfaces



Assembling Communicating Objects

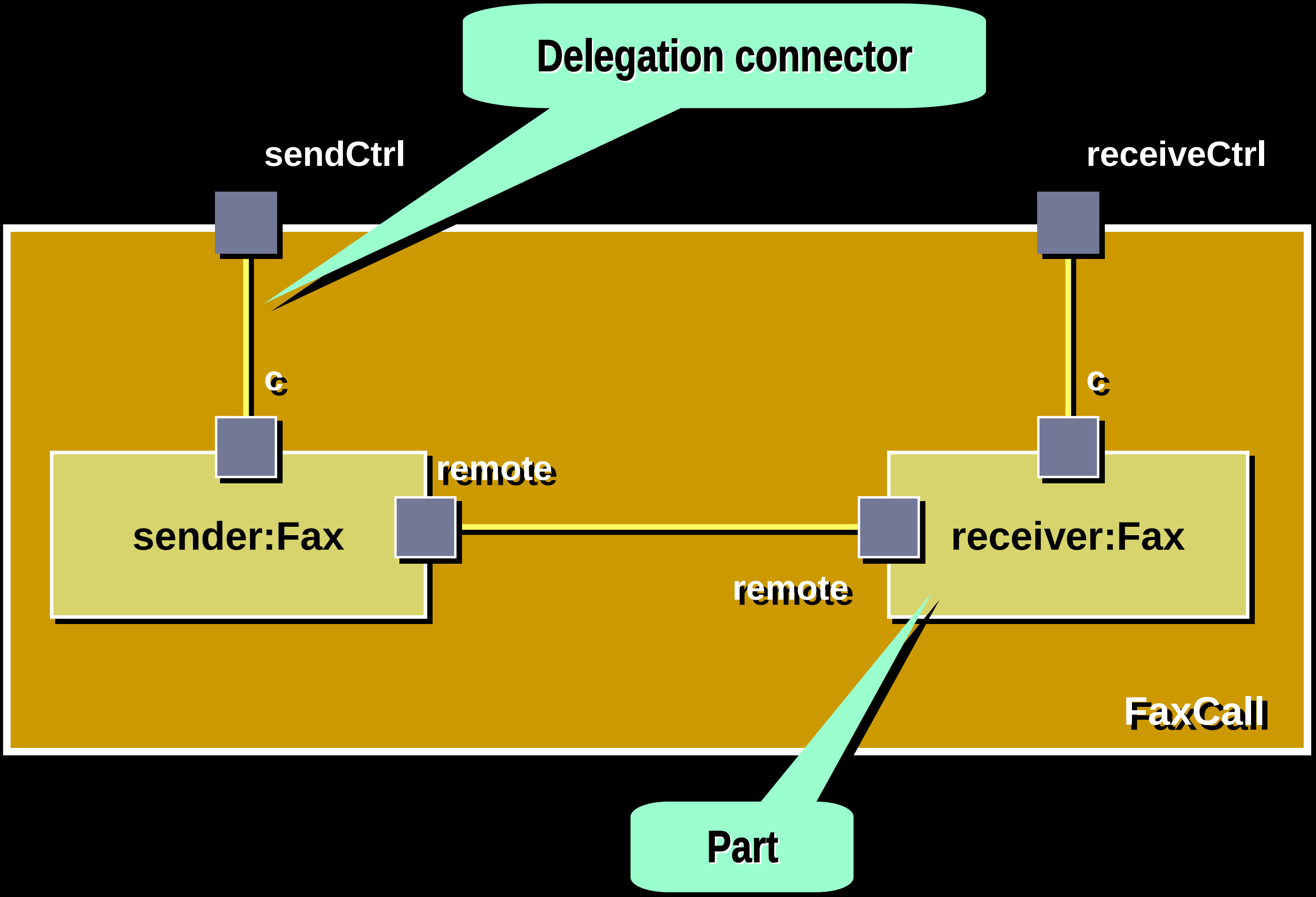
- Ports can be joined by *connectors* to create peer collaborations composed of structured classes



Connectors model communication channels
A connector is constrained by a protocol
Static typing rules apply (compatible protocols)

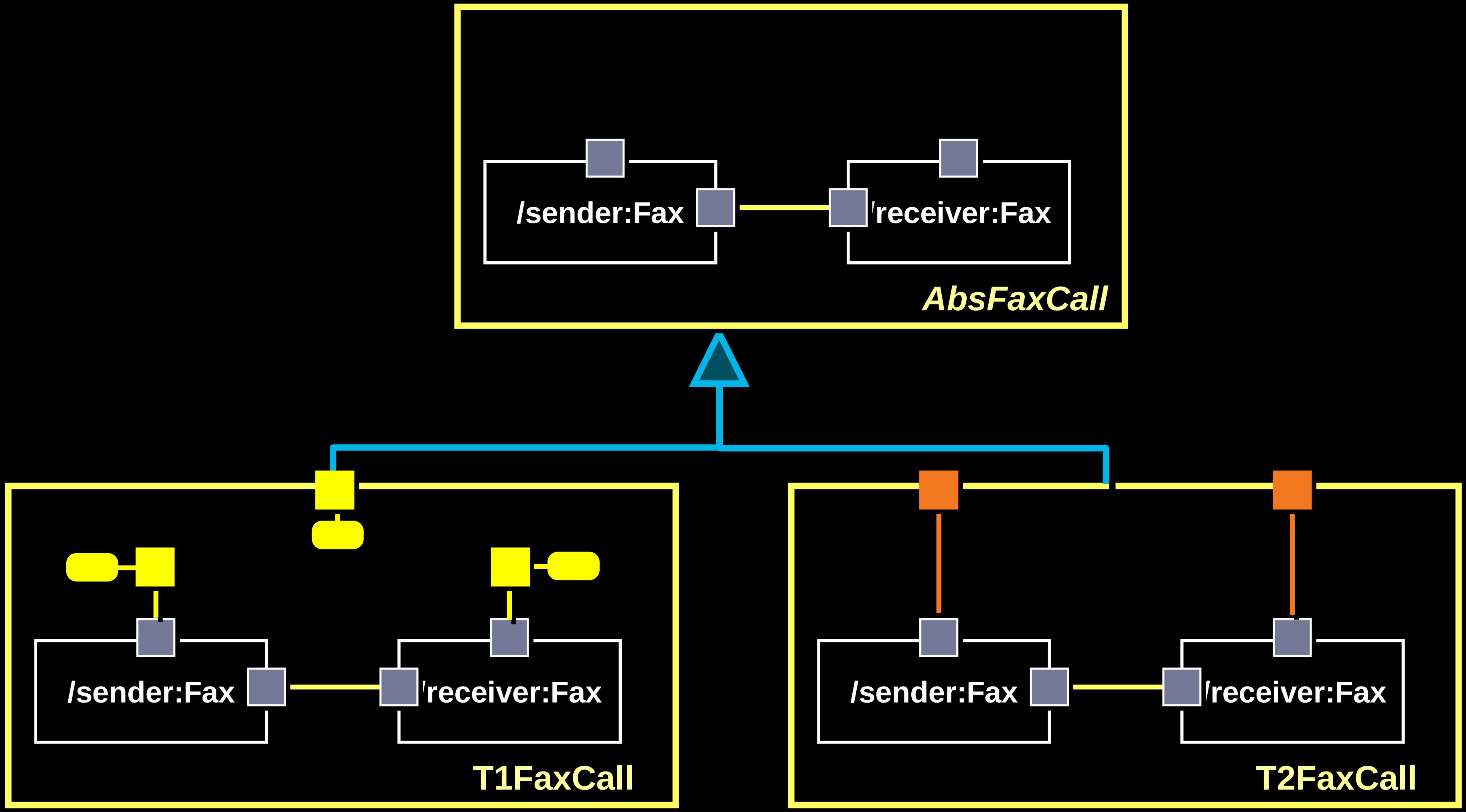
Structured Classes: Internal Structure

- Structured classes may have an internal structure of (structured class) parts and connectors

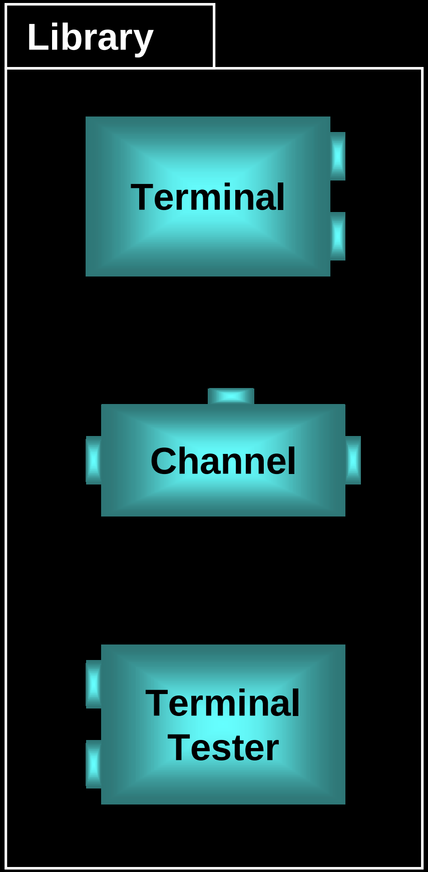


Structure Refinement Through Inheritance

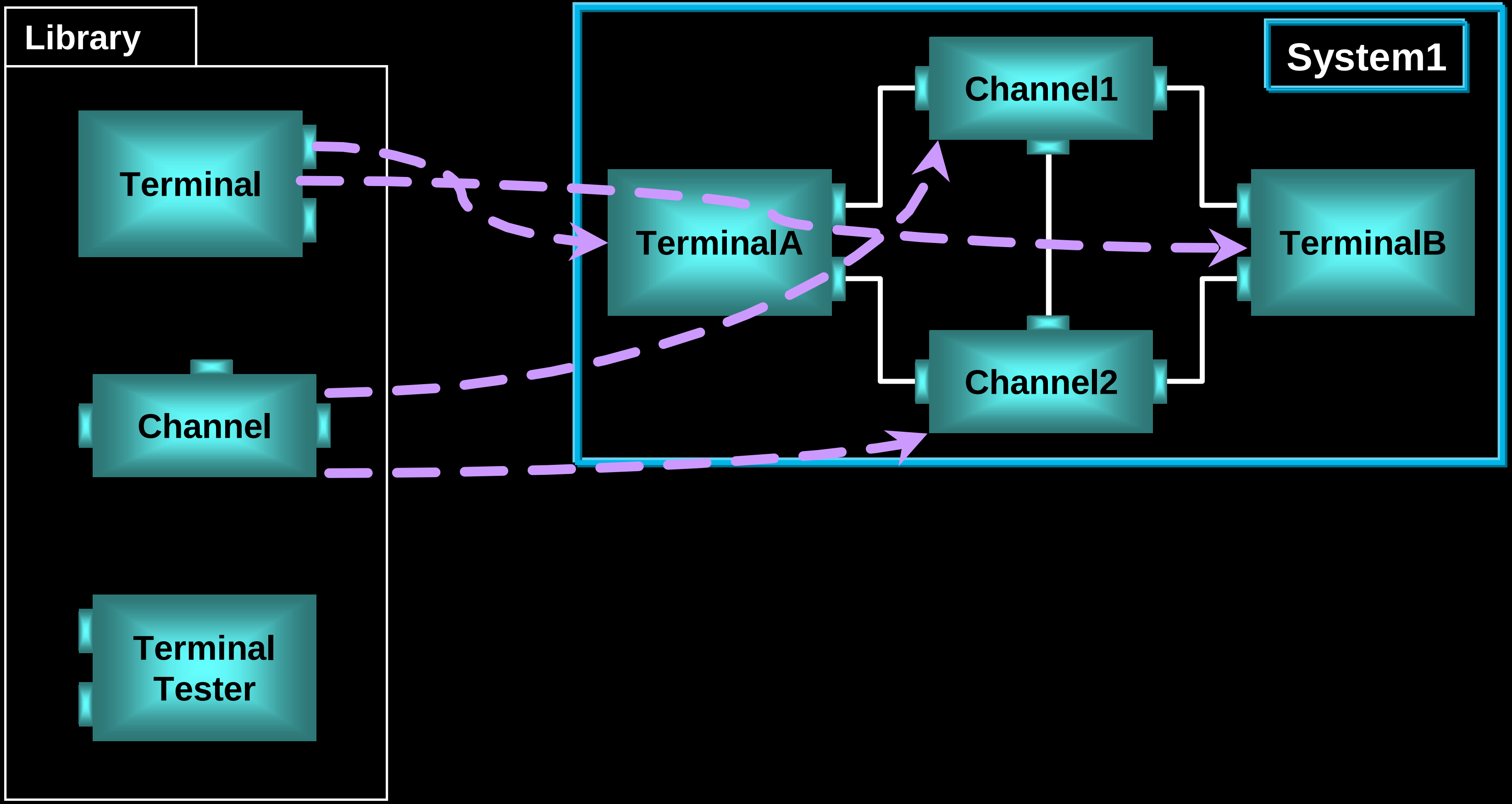
- For product families with a common architecture



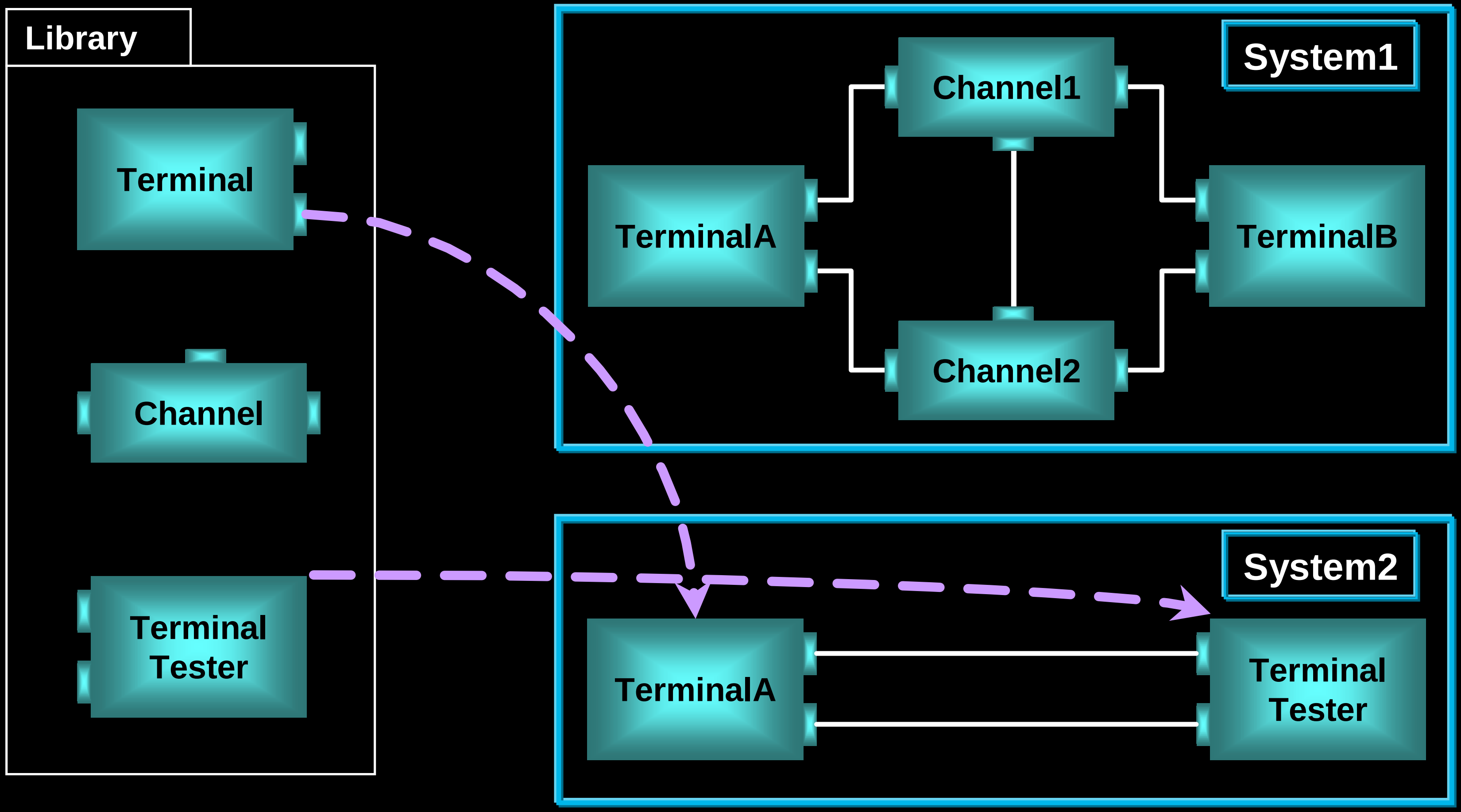
Structured Class Reuse: “Software Lego®”



Structured Class Reuse: “Software Lego®”

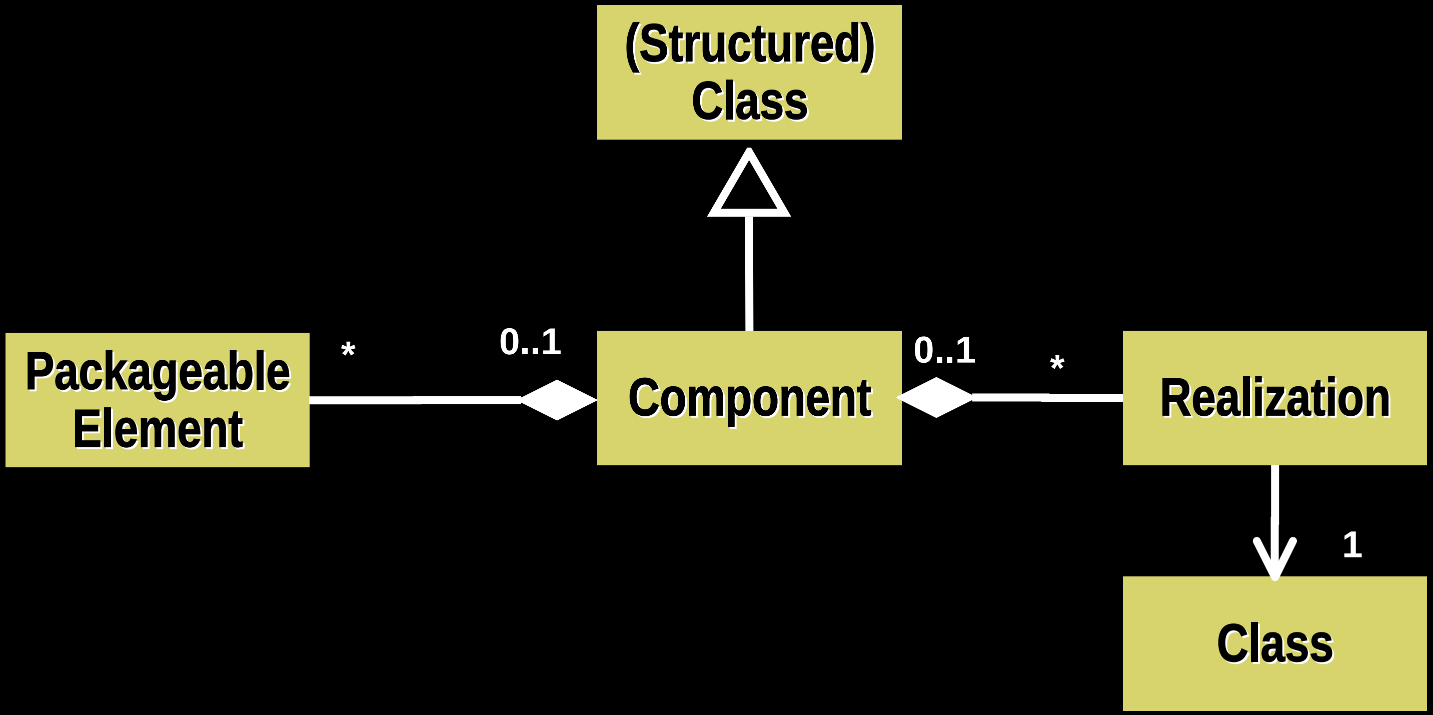


Structured Class Reuse: "Software Lego®"



Components

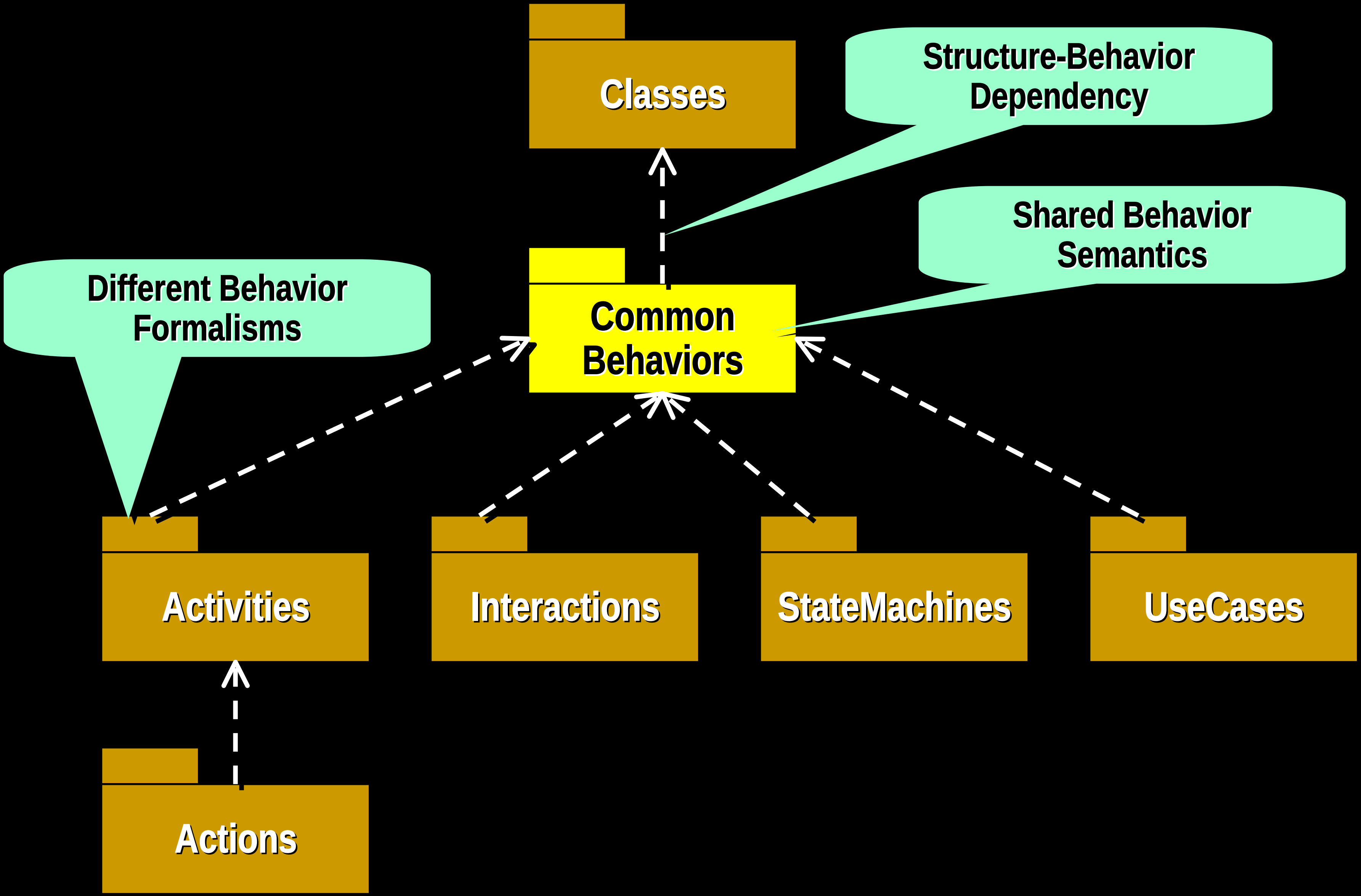
- A kind of structured class whose specification
  - May be realized by one or more implementation classes
  - May include any other kind of packageable element (e.g., various kinds of classifiers, constraints, packages, etc.)



Subsystems

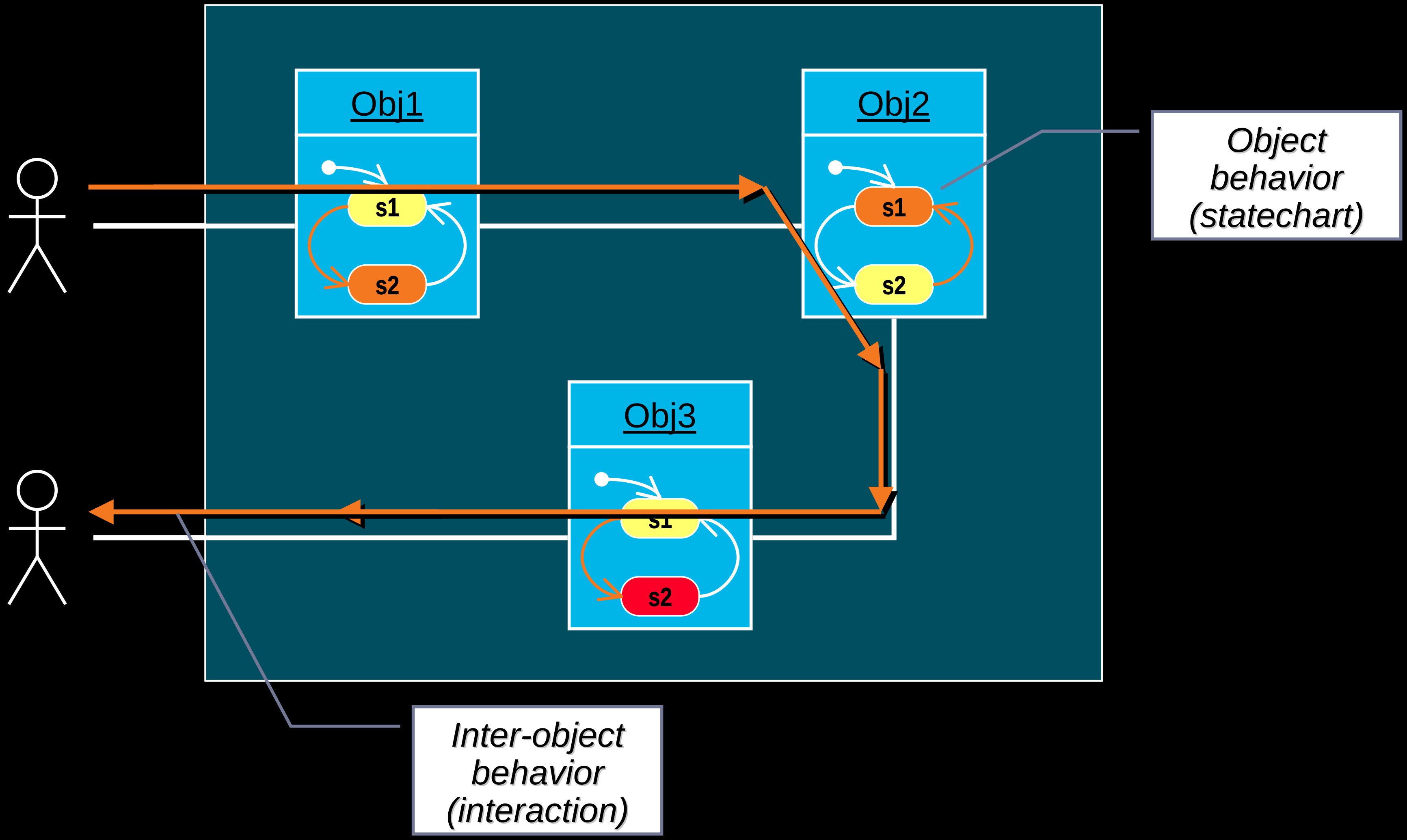
- A system stereotype of Component («subsystem») such that it may have explicit and distinct specification («specification») and realization («realization») elements
  - Ambiguity of being a subclass of Classifier and Package has been removed (was intended to be mutually exclusive kind of inheritance)
  - Component (specifications) can contain any packageable element and, hence, act like packages

Dynamic Modeling Concepts


Classes
Structure-Behavior Dependency
Shared Behavior Semantics
Different Behavior Formalisms
Common Behaviors
Activities
Interactions
StateMachines
UseCases
Actions

Structure and Behavior

- Structure is the context for all behavior:

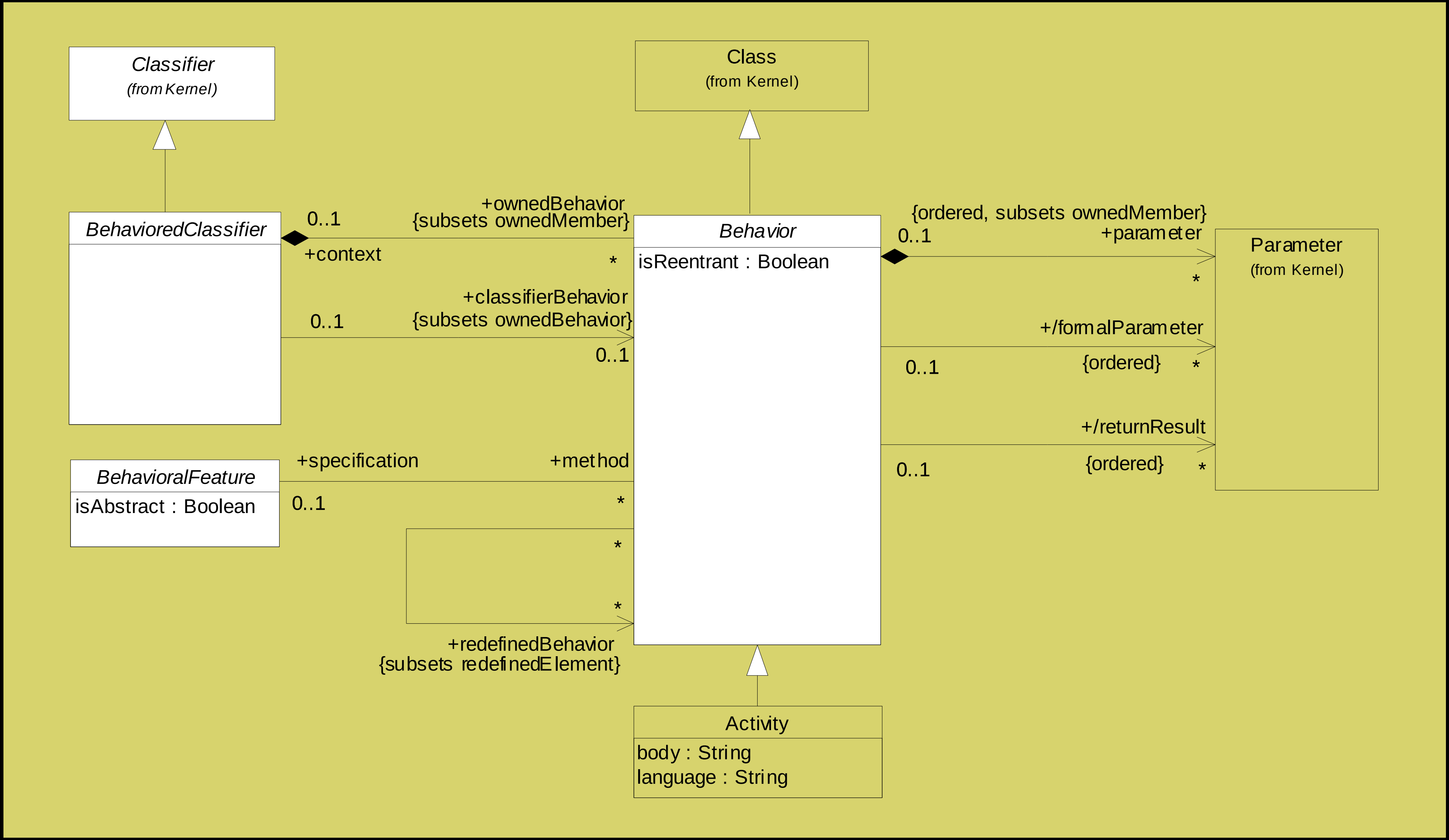


Obj1
s1
s2
Obj2
s1
s2
Obj3
s1
s2
Object behavior (statechart)
Inter-object behavior (interaction)

How Things Happen in UML

- An action is executed
  - May change the value of one or more variables or object attributes
  - If it is a “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
    - May be controlled by control flow

Common Behavior Metamodel

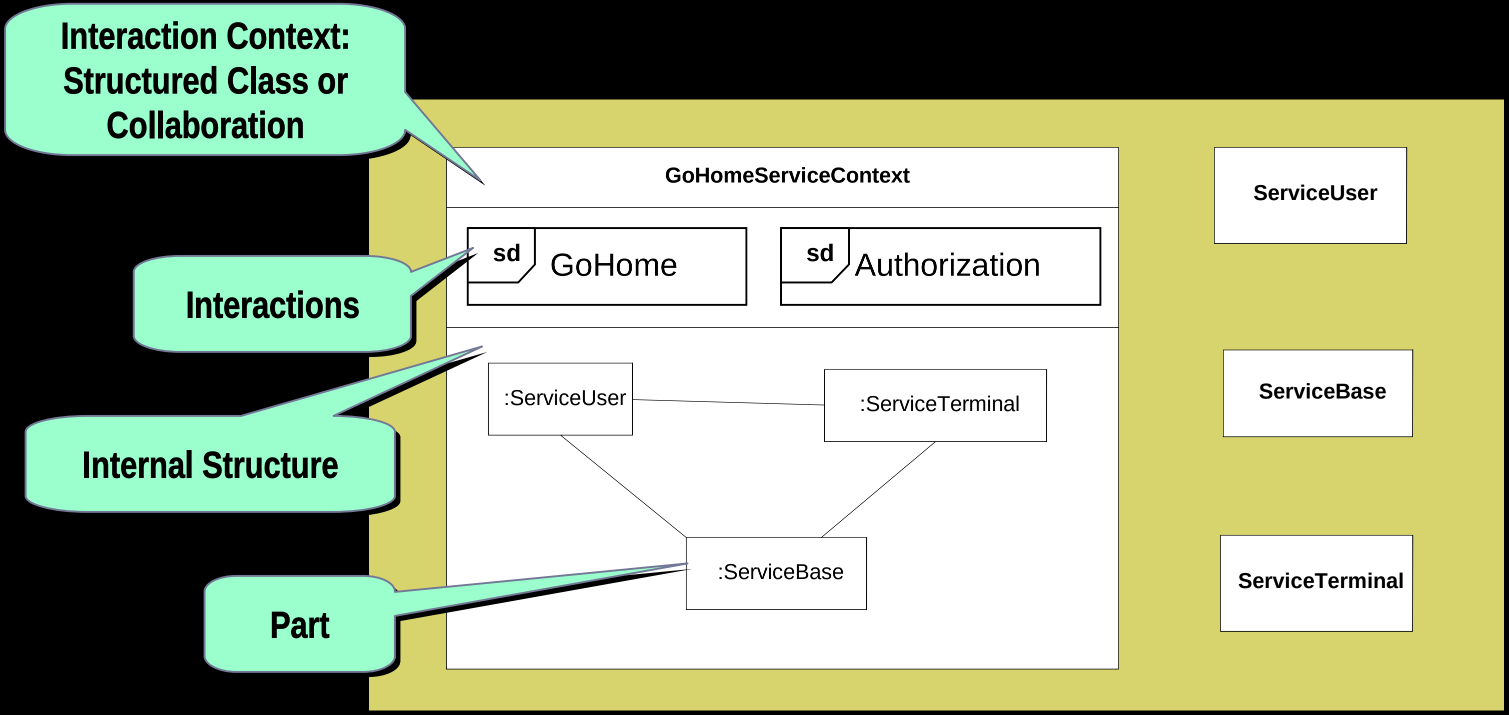


Overview of New Features

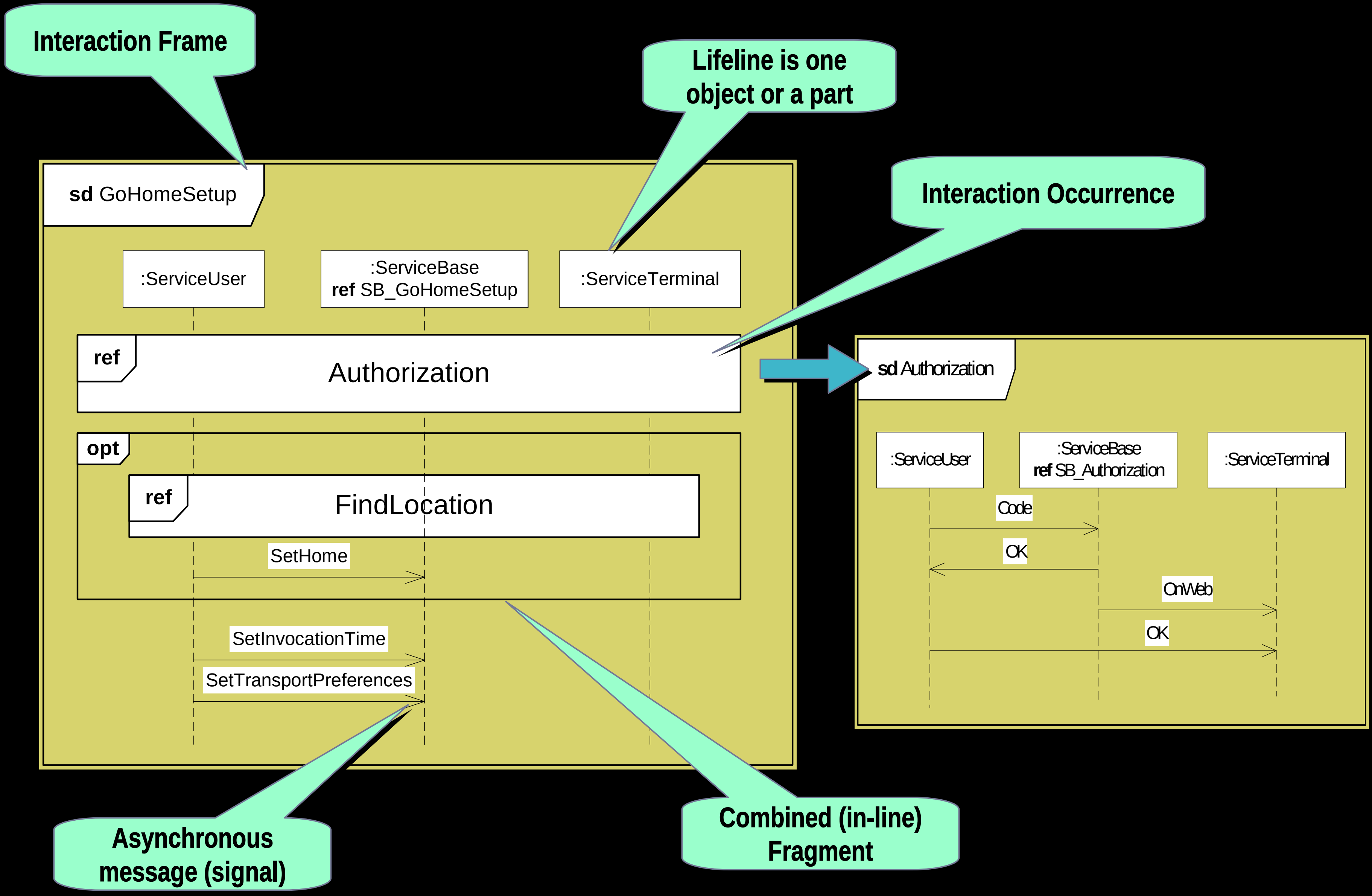
- Interactions focus on the communications between collaborating instances communicating via messages
  - Both synchronous (operation invocation) and asynchronous (signal sending) models supported
- Multiple concrete notational forms:
  - sequence diagram
  - communication diagram
  - interaction overview diagram
  - activity diagram
  - timing diagram
  - interaction table

Example: Interaction Context

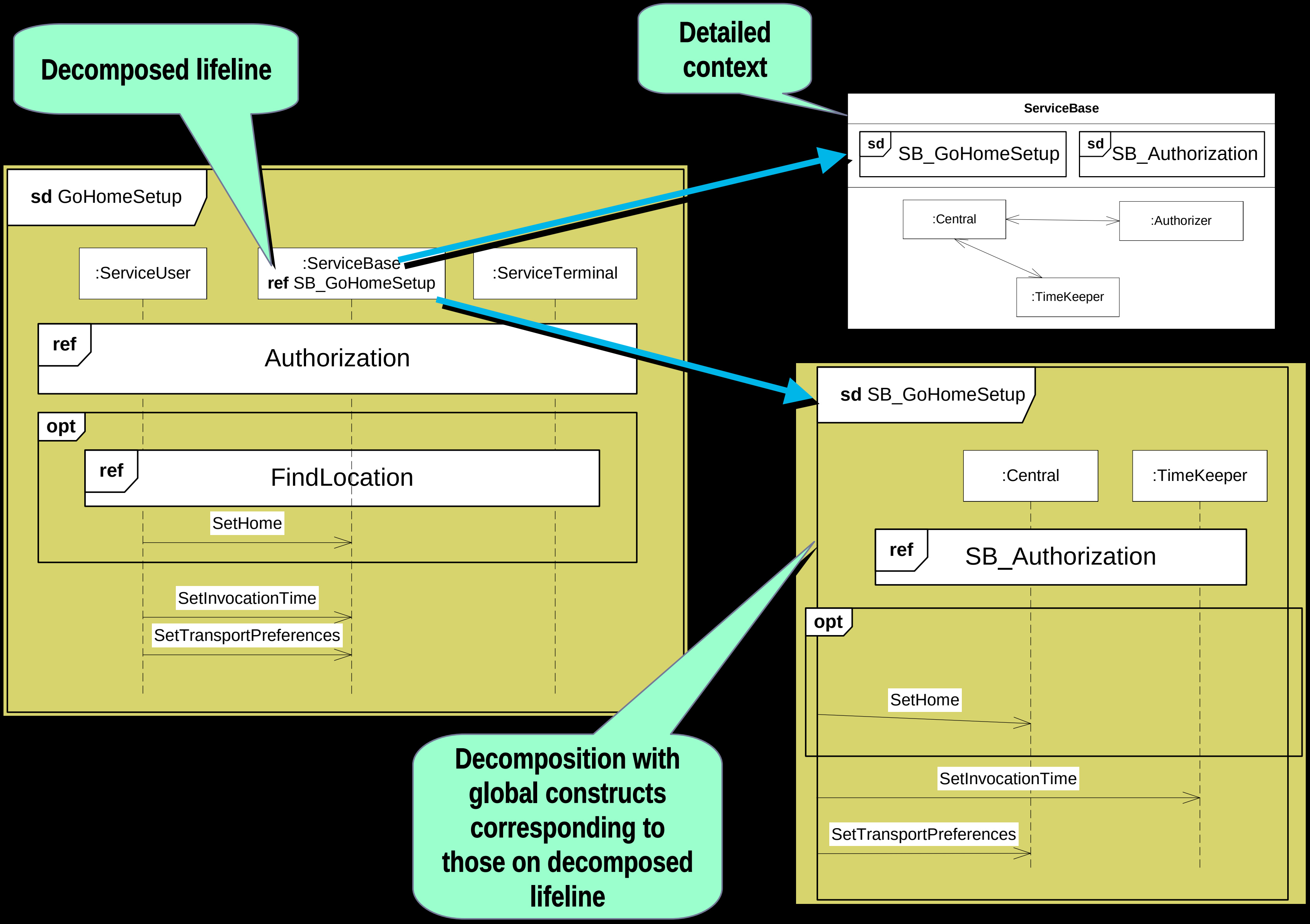
- All interactions occur in structures of collaborating parts
  - the structural context for the interaction



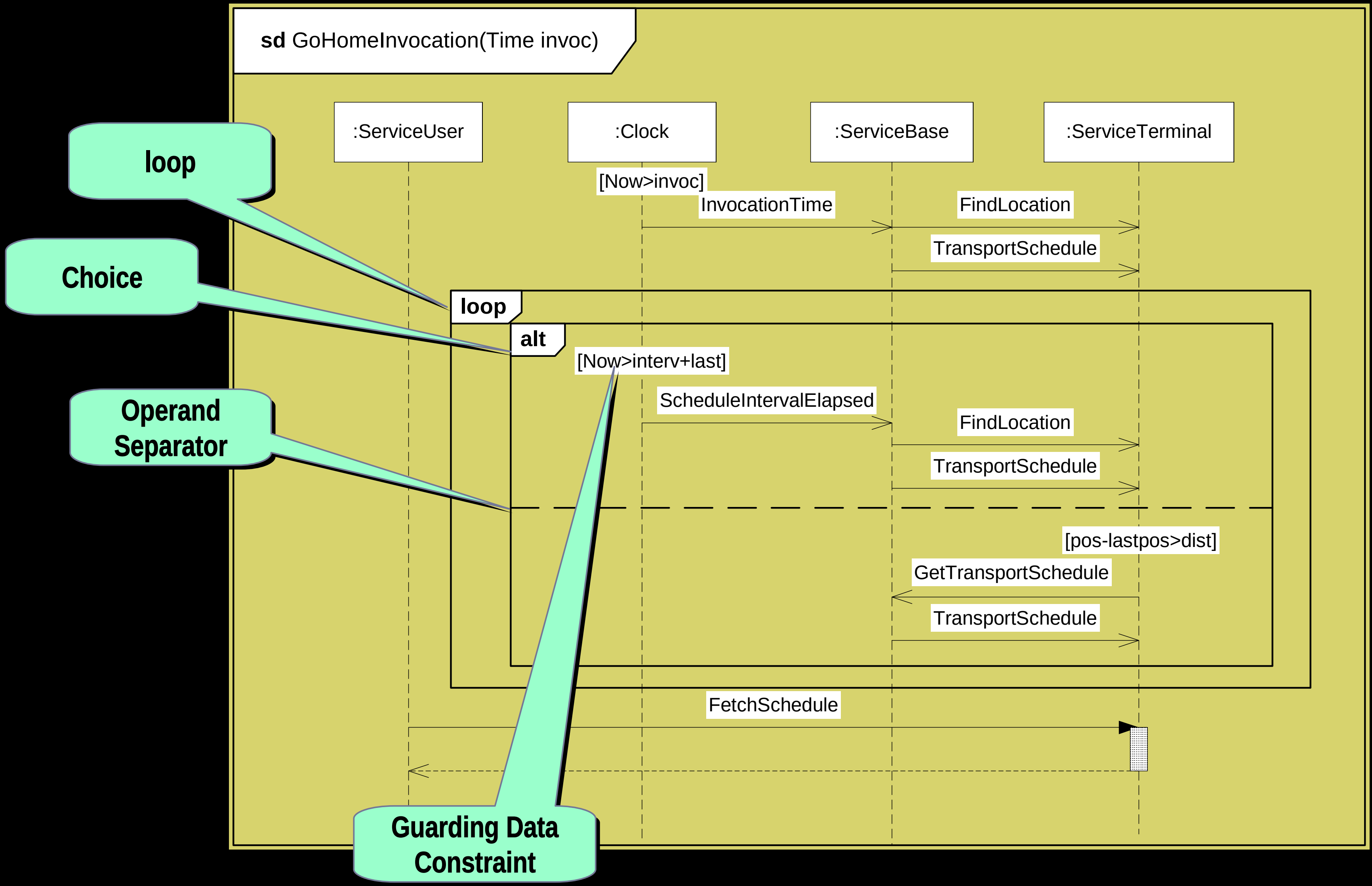
Interaction Occurrences



Structural Decomposition in Sequence Diagrams



Combined Fragments and Data



Combined Fragment Types (1 of 2)

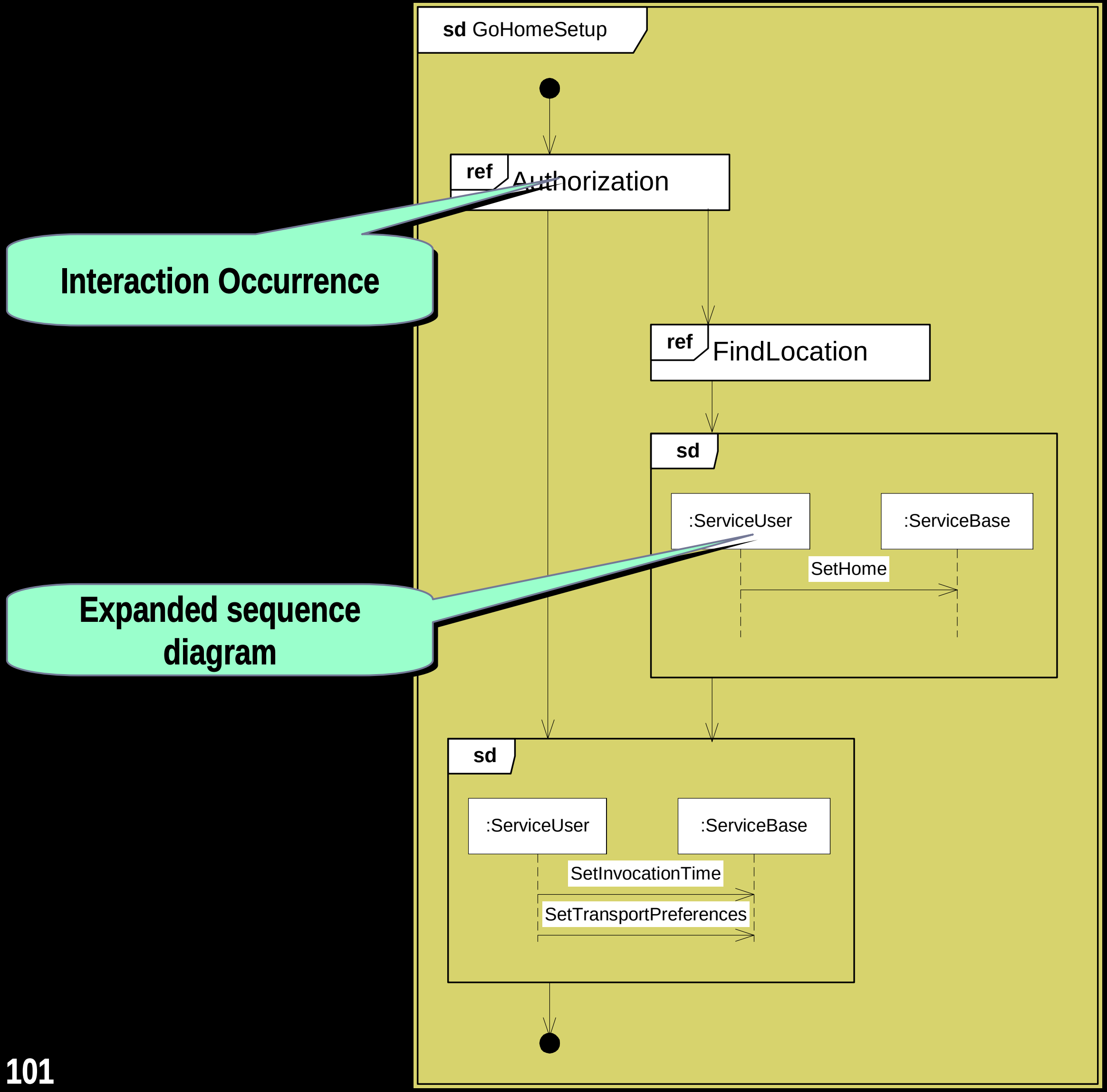
- Alternatives (**alt**)
  - choice of behaviors – at most one will execute
  - depends on the value of the guard (“else” guard supported)
- Option (**opt**)
  - Special case of alternative
- Break (**break**)
  - Represents an alternative that is executed instead of the remainder of the fragment (like a break in a loop)
- Parallel (**par**)
  - Concurrent (interleaved) sub-scenarios
- Negative (**neg**)
  - Identifies sequences that must <u>not</u> occur

Combined Fragment Types (2 of 2)

- Critical Region (**critical**)
  - Traces cannot be interleaved with events on any of the participating lifelines
- Assertion (**assert**)
  - Only valid continuation
- Loop (**loop**)
  - Optional guard: [<min>, <max>, <Boolean-expression>]
  - No guard means no specified limit

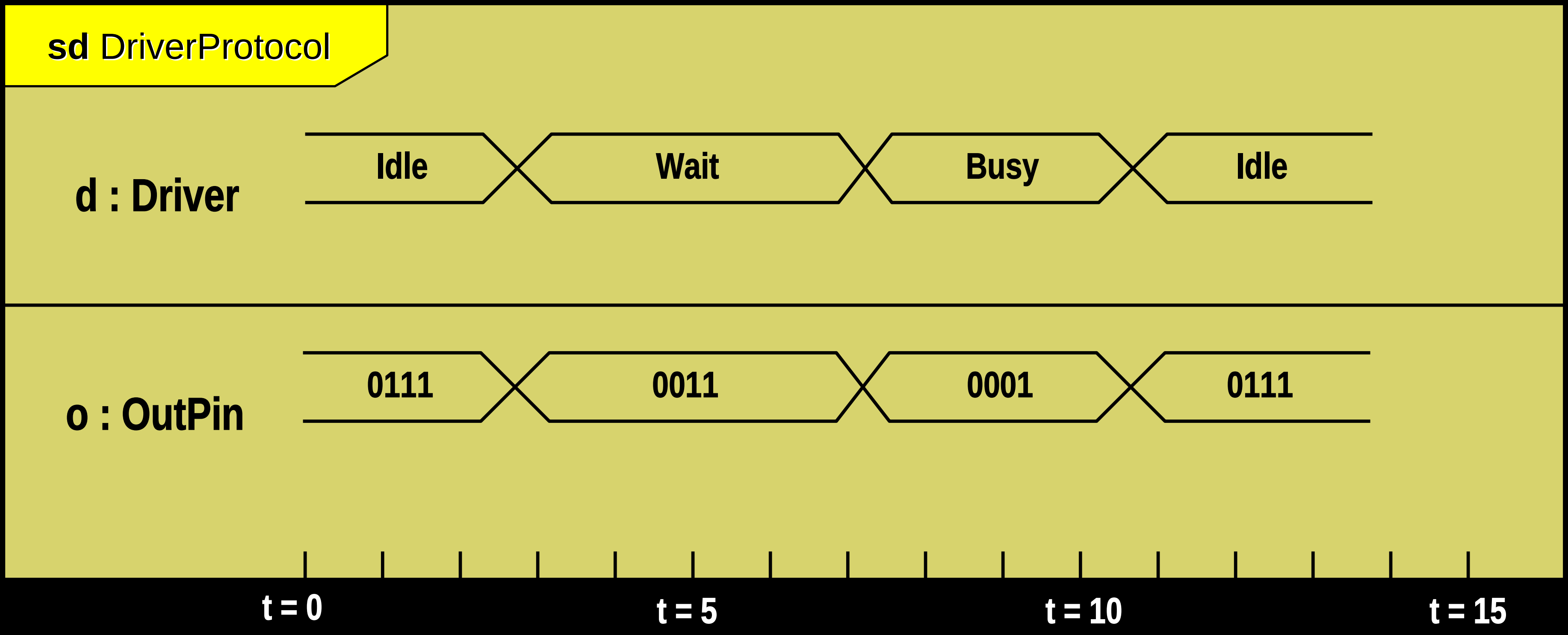
Interaction Overview Diagram

- An interaction with the syntax of activity diagrams



Timing Diagrams

- Can be used to specify time-dependent interactions
  - Based on a simplified model of time (use standard “real-time” profile for more complex models of time)

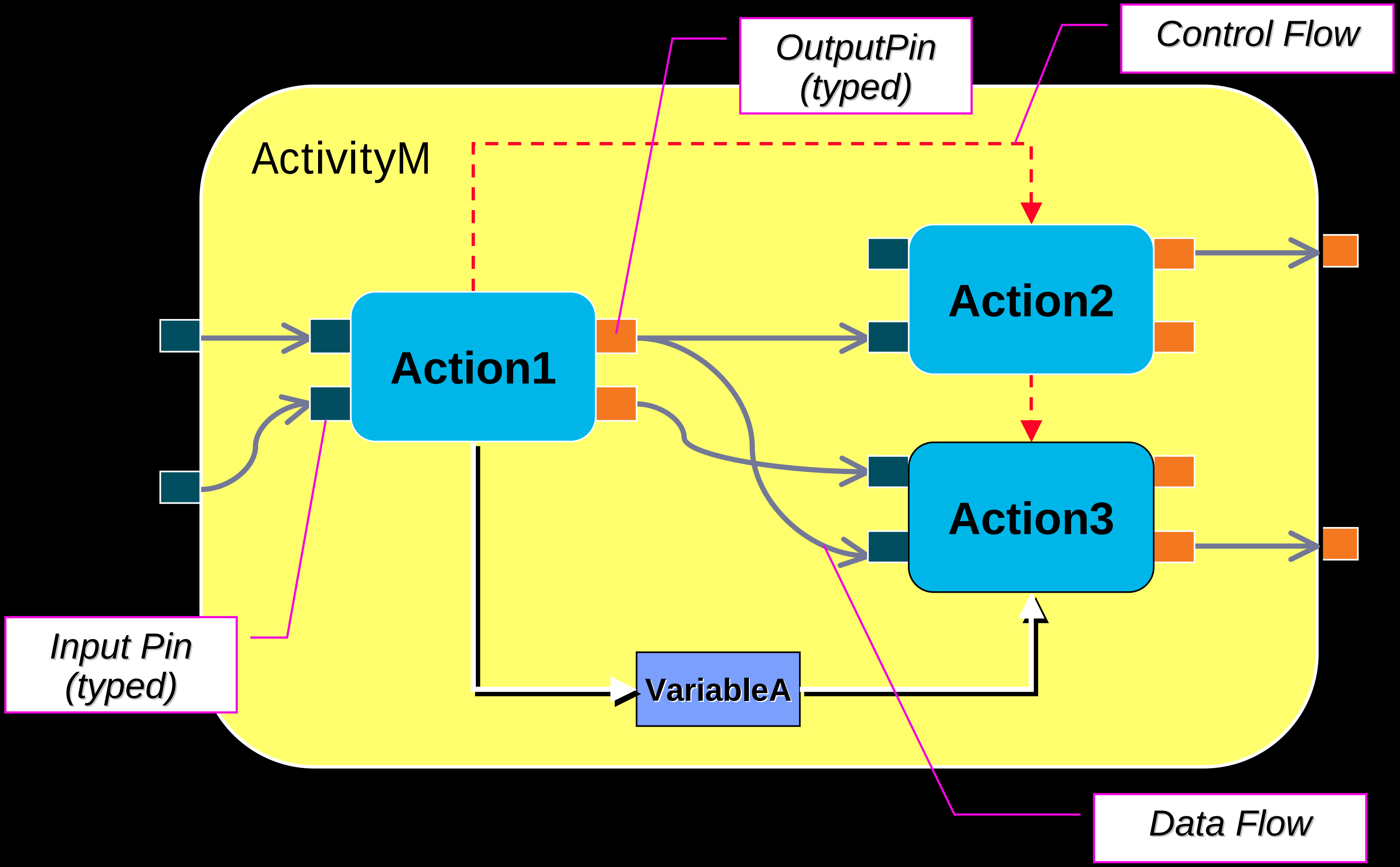


Actions in UML

- Action = fundamental unit of behavior
  - for modeling fine-grained behavior
  - Level of traditional programming languages
- UML defines:
  - A set of action types
  - A <u>semantics</u> for those actions
    - i.e. what happens when the actions are executed
  - In general, no specific standard notation for actions
    - a few exceptions, e.g., “send signal”
  - This provides a flexibility to use any language to realize the semantics
    - There is **no** “Action Semantics Language” in UML
- In UML 2, the metamodel of actions was consolidated
  - Shared semantics between actions and activities (Basic Actions)

Shared Action/Activity Semantics

- Data/control flow foundations for maximal implementation flexibility

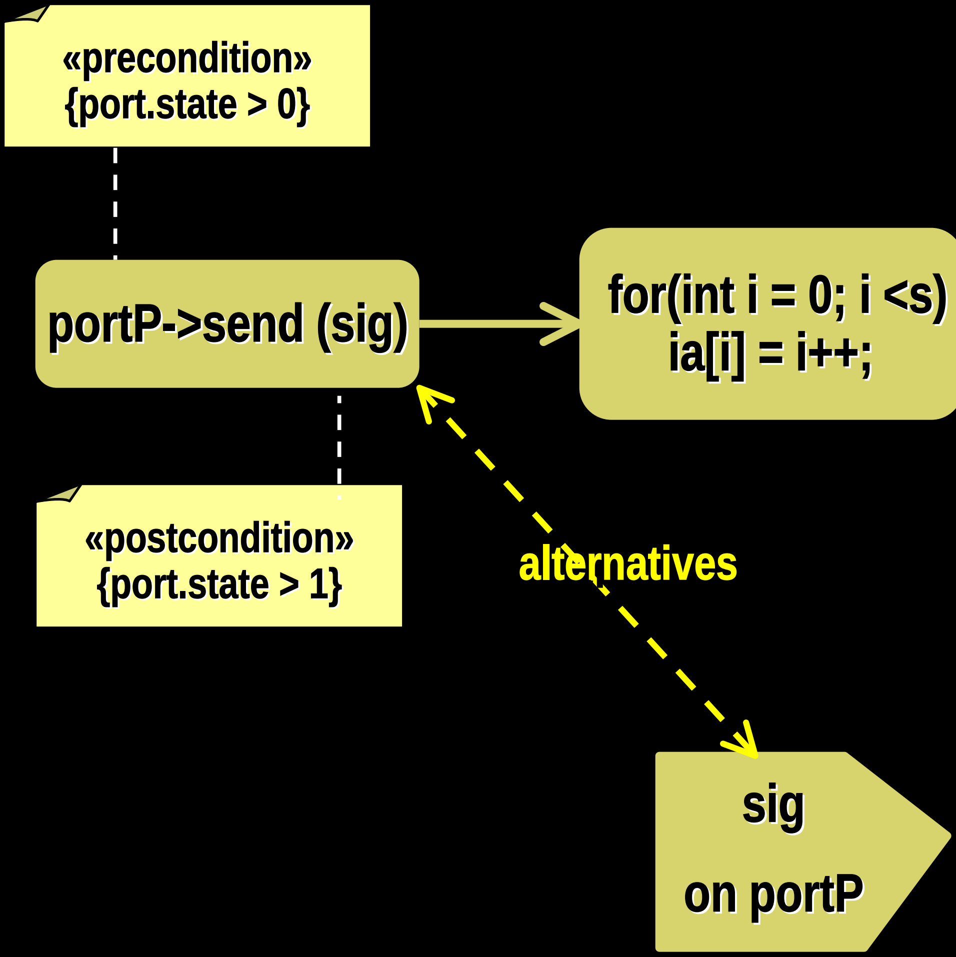


Categories of Actions

- 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)

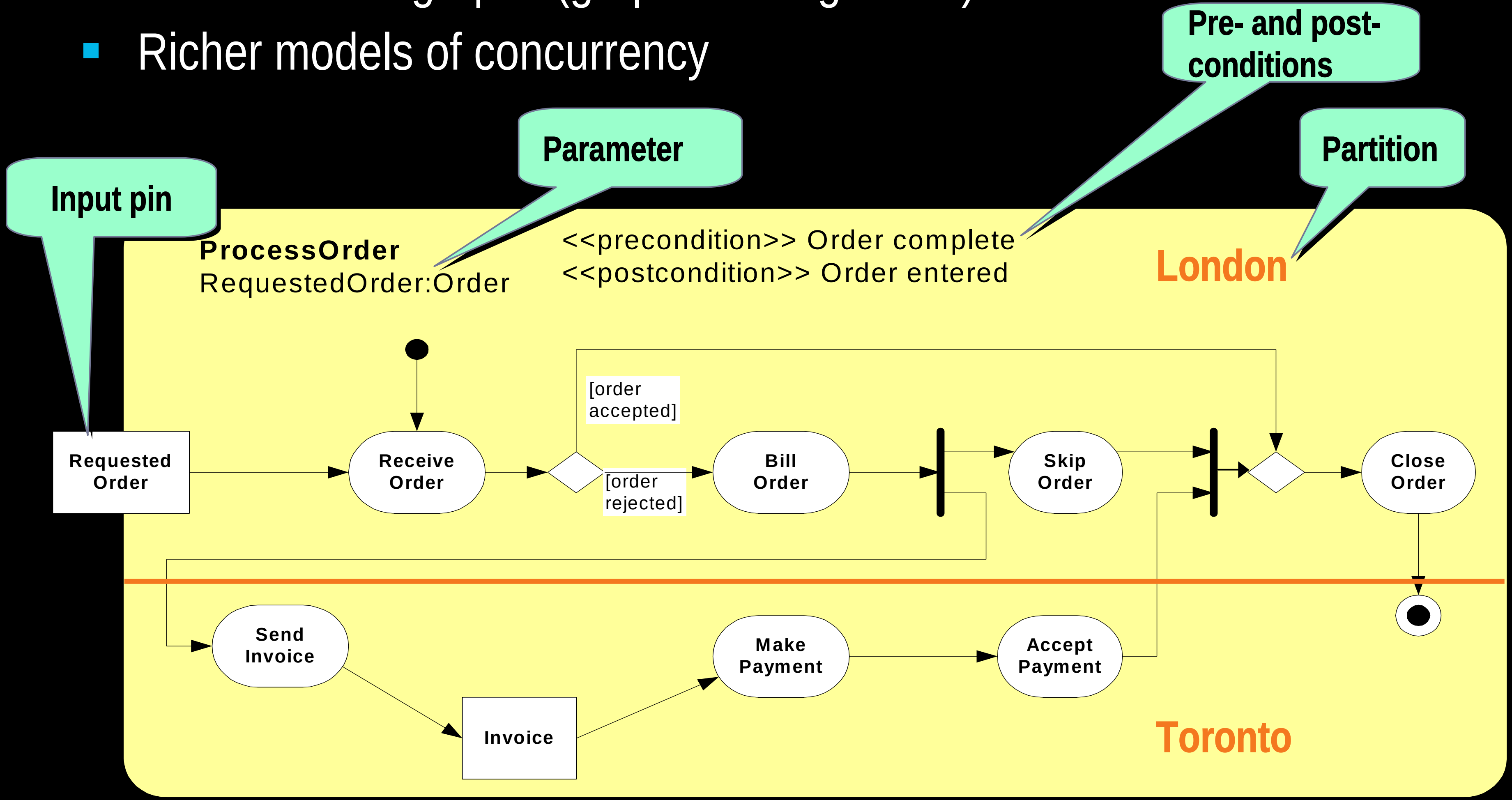
General Notation for Actions

- No specific symbols (some exceptions)

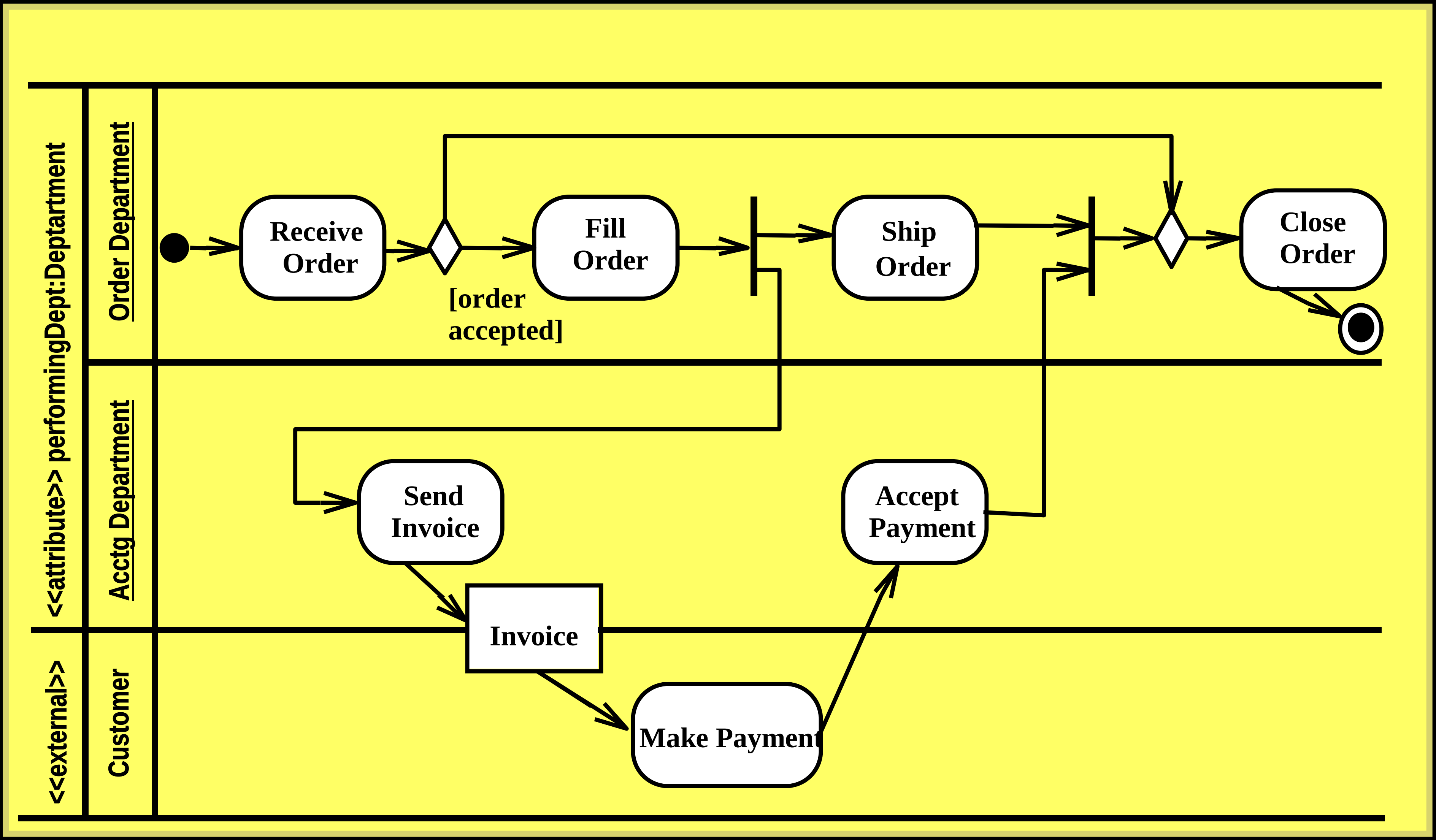


Activities: New Semantic Foundation

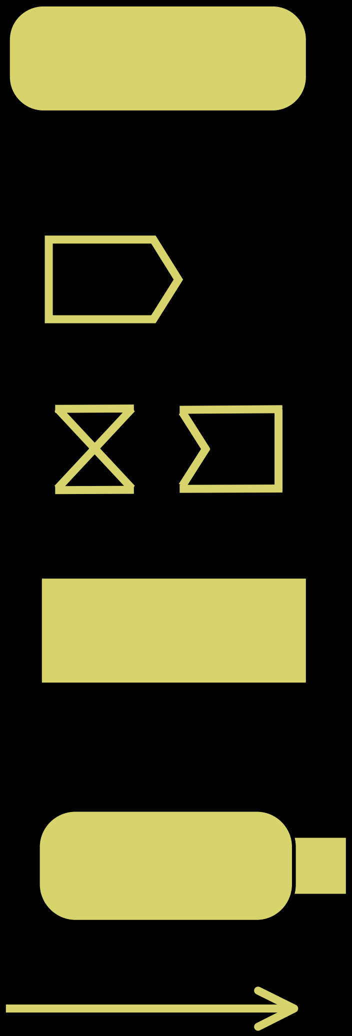
- Petri Net-like foundation (vs. statecharts) enables
  - Un-structured graphs (graphs with “go-to’s”)
  - Richer models of concurrency



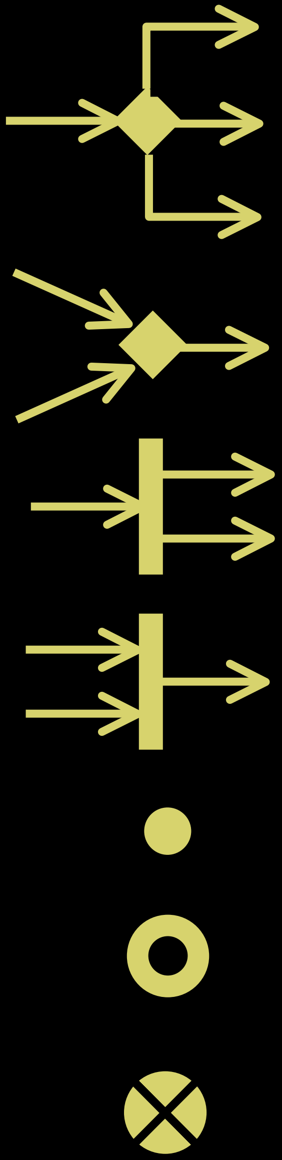
Hierarchical Partitions



Activities: Basic Notational Elements



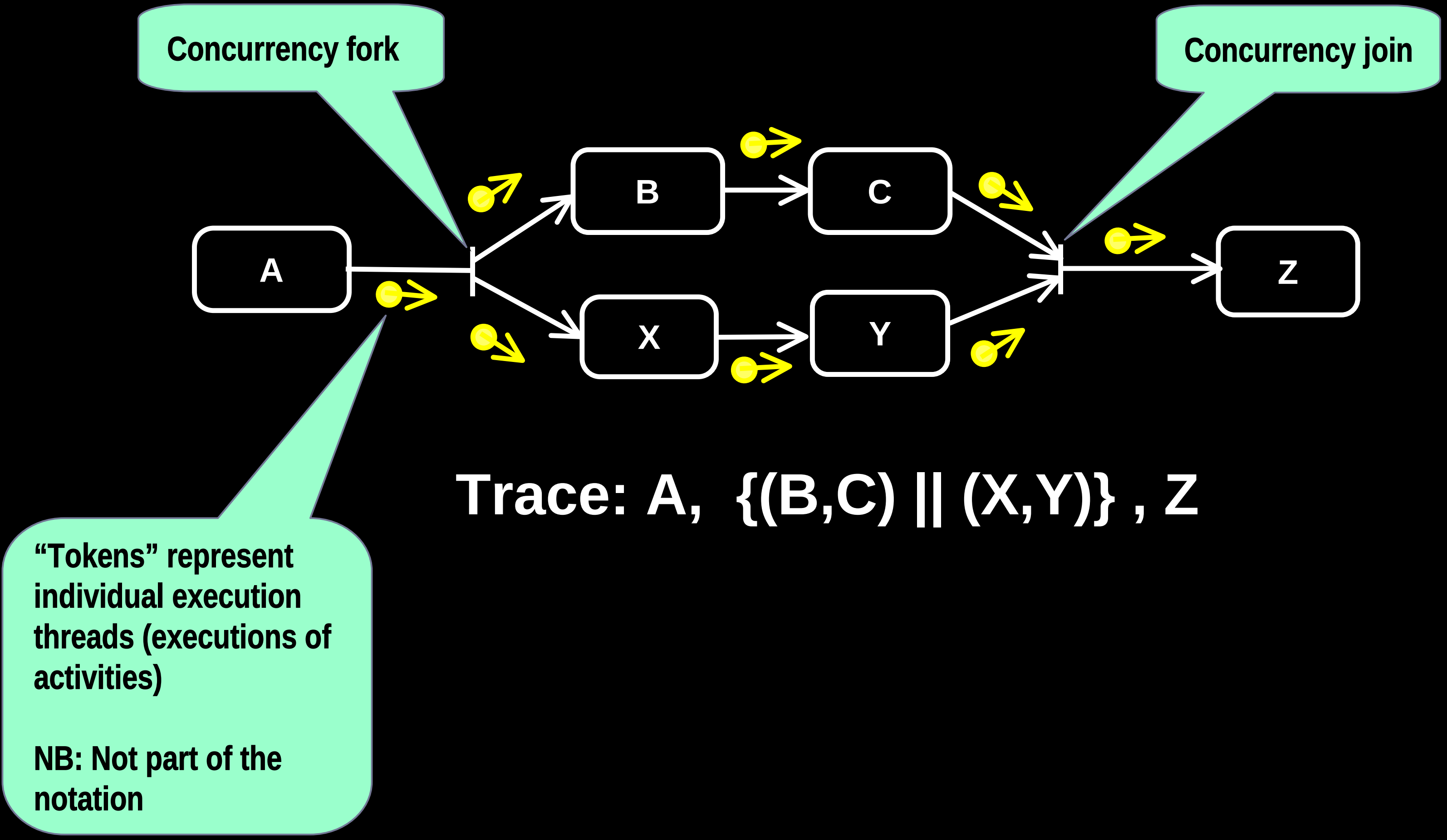
- Activity or Action
- Send Signal Action
- Accept Event Action
- Object Node
  (may include state)
- Pin (Object)
- Control/Data Flow



- Choice
- (Simple) Join
- Control Fork
- Control Join
- Initial Node
- Activity Final
- Flow Final

Extended Concurrency Model

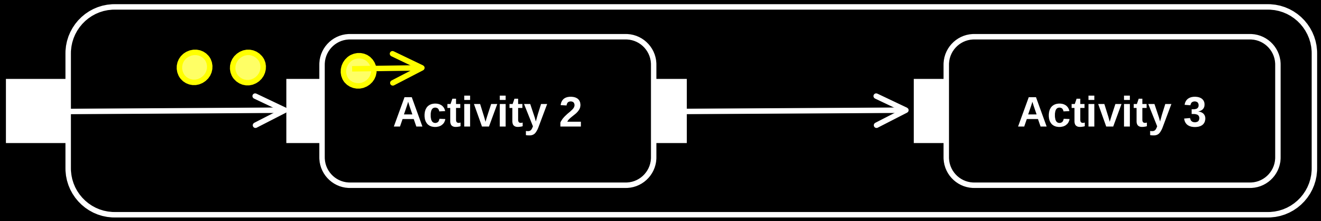
- Fully independent concurrent streams (“tokens”)


Concurrency fork
Concurrency join
A
B
C
X
Y
Z
“Tokens” represent individual execution threads (executions of activities)
NB: Not part of the notation

Trace: A, {(B,C) || (X,Y)} , Z

Activities: Token Queuing Capabilities

- Tokens can
  - queue up in “in/out” pins.
  - back up in network.
  - prevent upstream behaviors from taking new inputs.

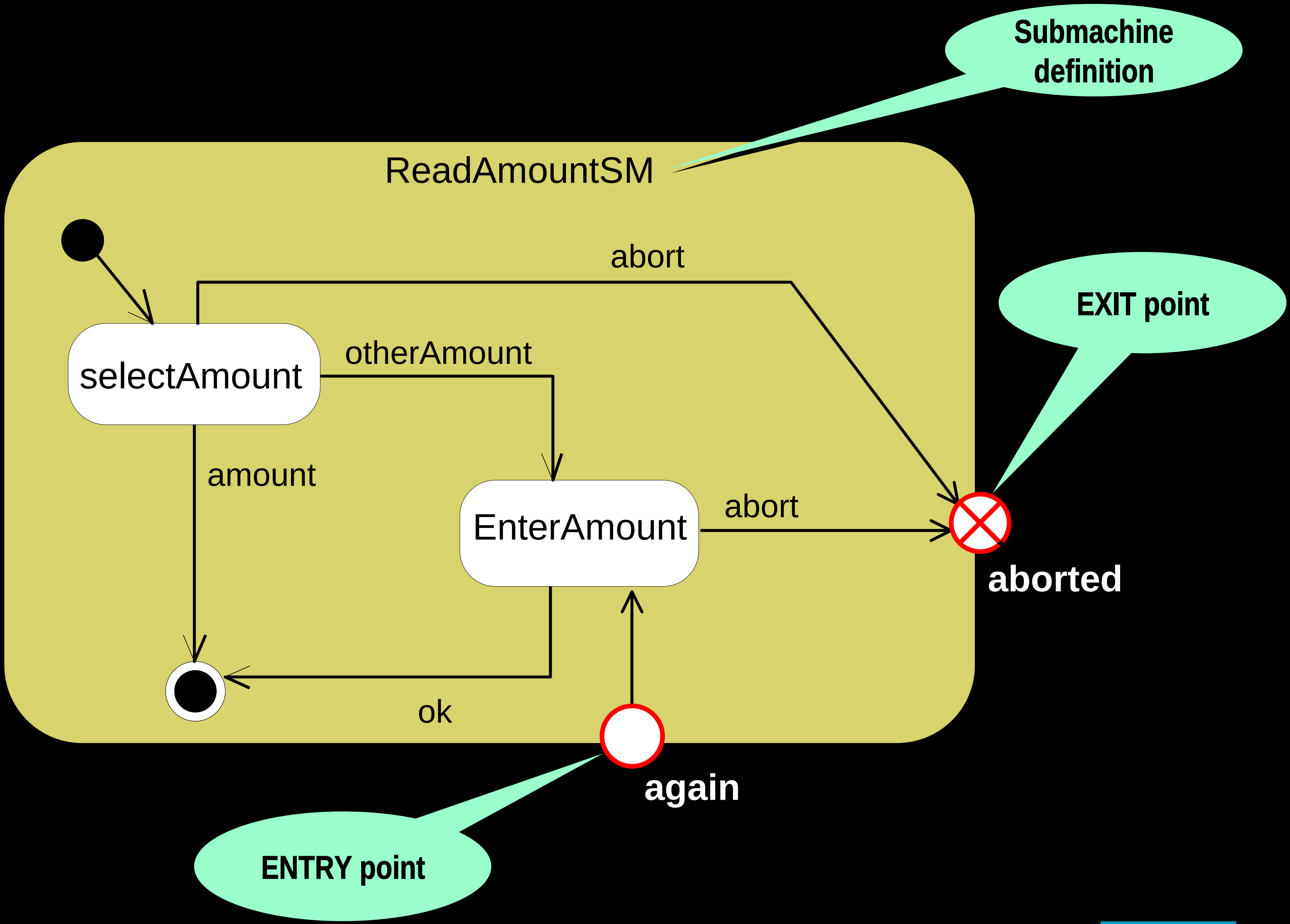


- …or, they can flow through continuously
  - taken as input while behavior is executing
  - given as output while behavior is executing
  - identified by a {stream} adornment on a pin or object node

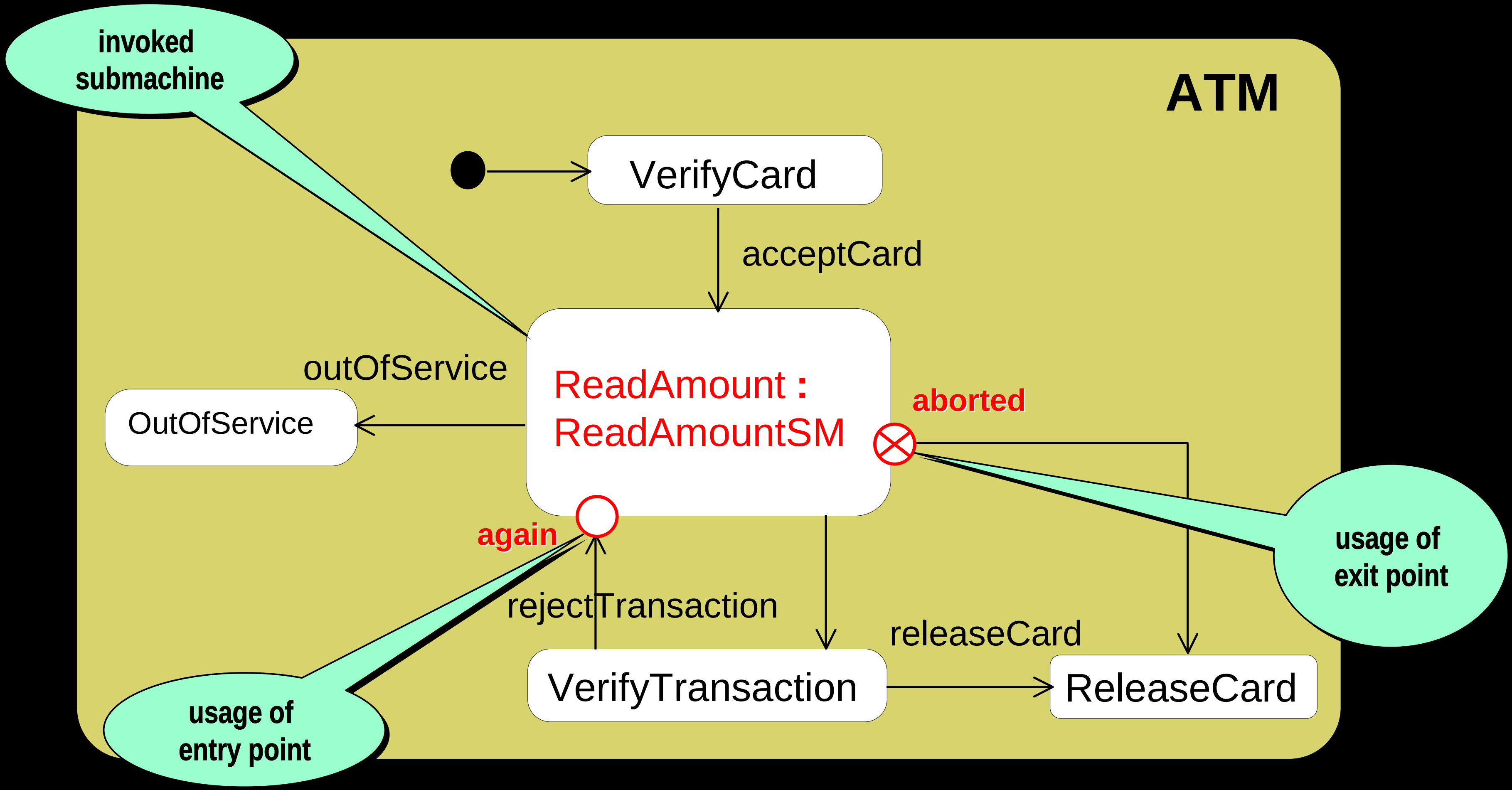
State Machine Improvements

- New modeling constructs:
  - Modularized submachines
  - State machine specialization/redefinition
  - State machine termination
  - “Protocol” state machines
    - transition pre/post conditions
    - protocol conformance
- Notational enhancements
  - action blocks
  - state lists

Modular Submachines: Definition

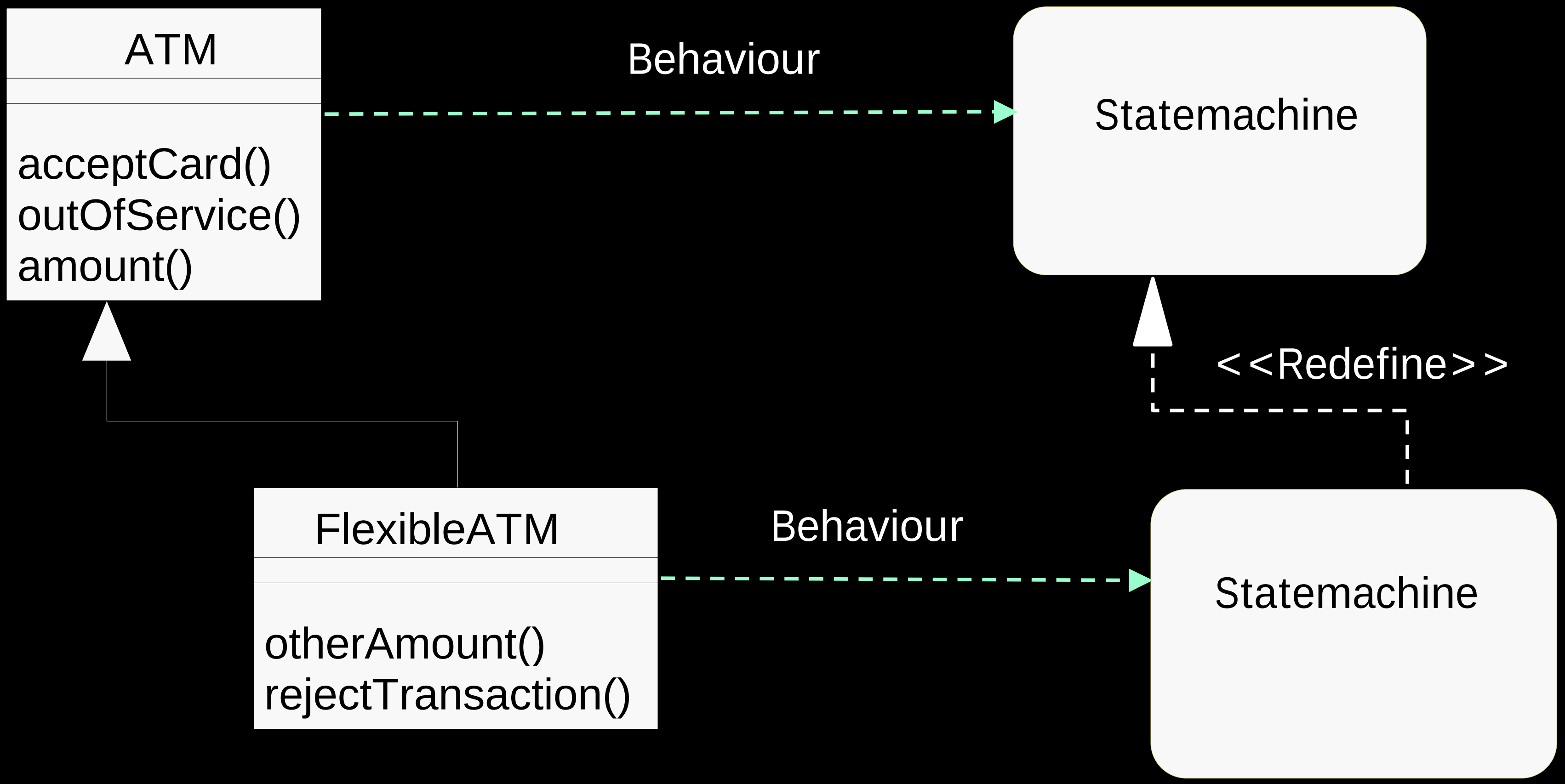


Modular Submachines: Usage



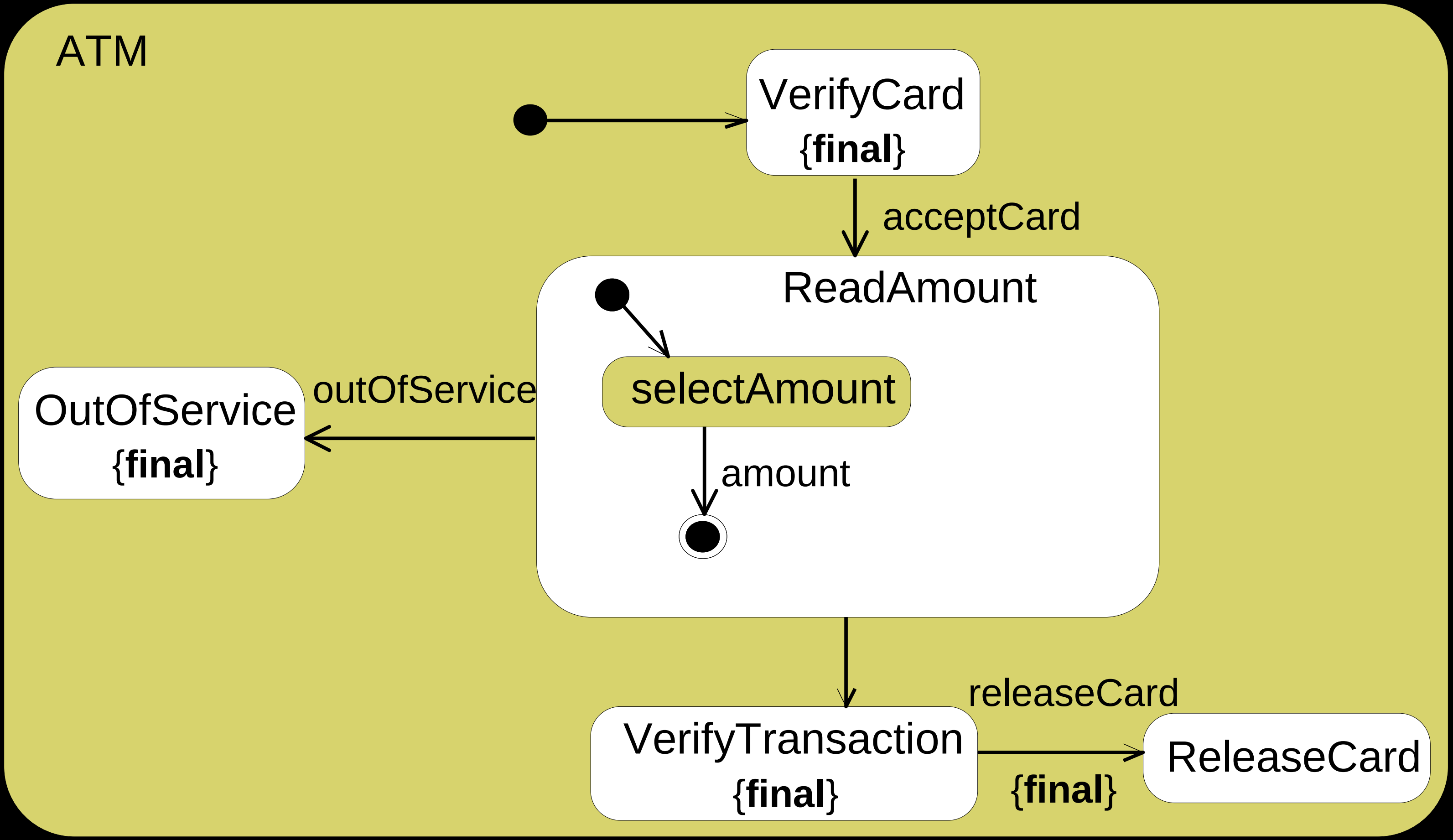
Specialization

- Redefinition as part of standard class specialization

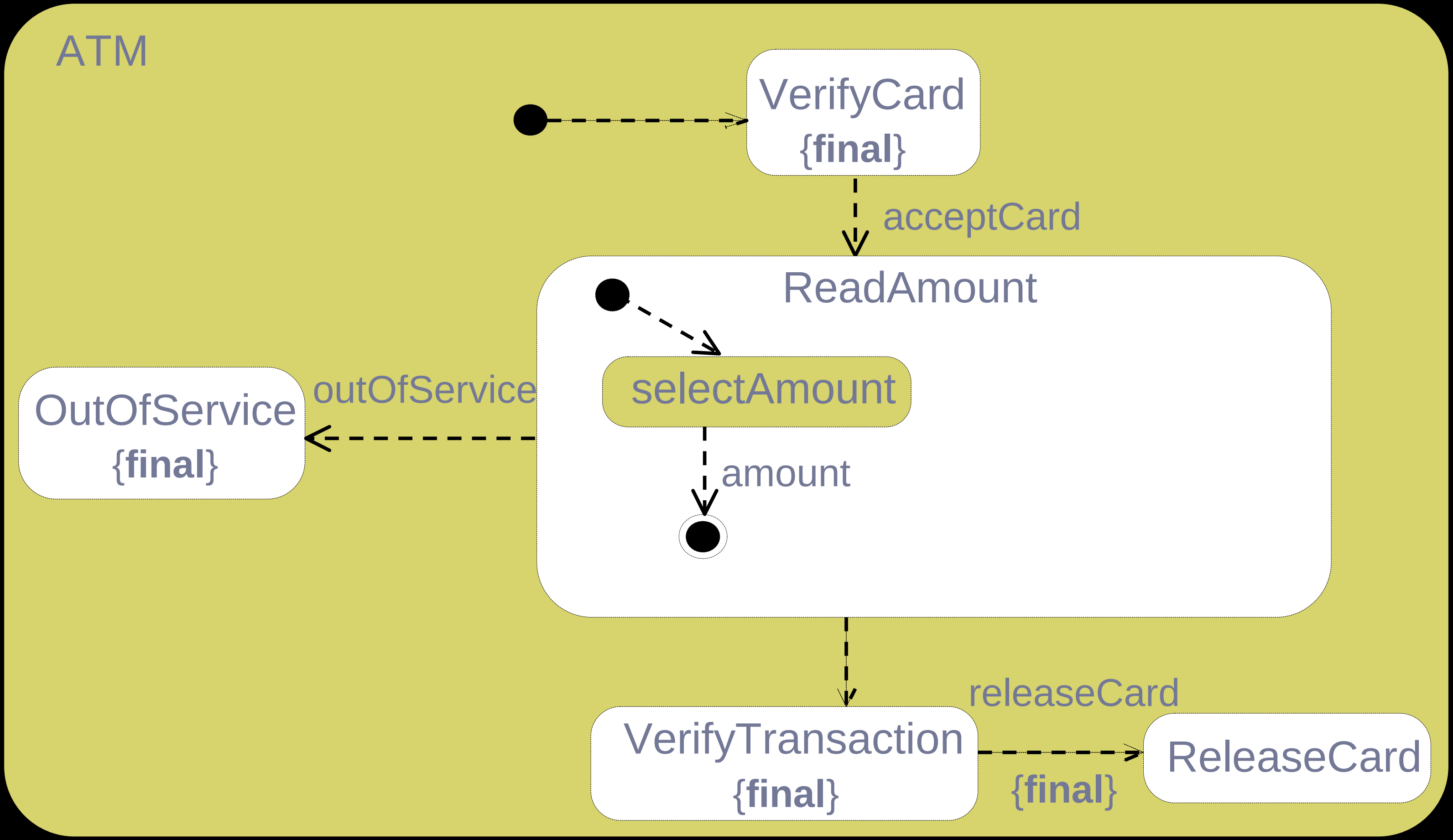


Example: State Machine Redefinition

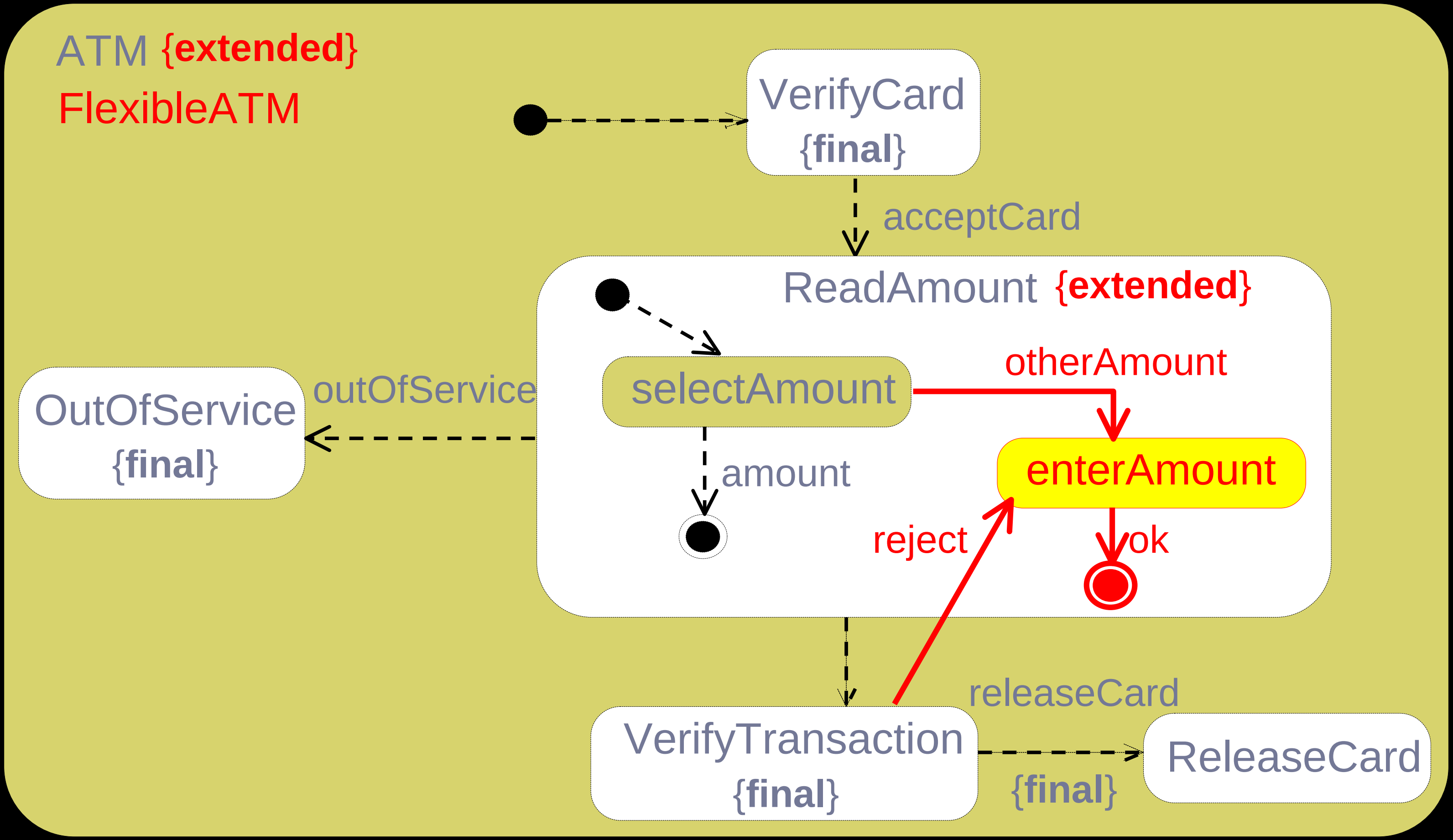
- State machine of ATM to be redefined



State Machine Redefinition

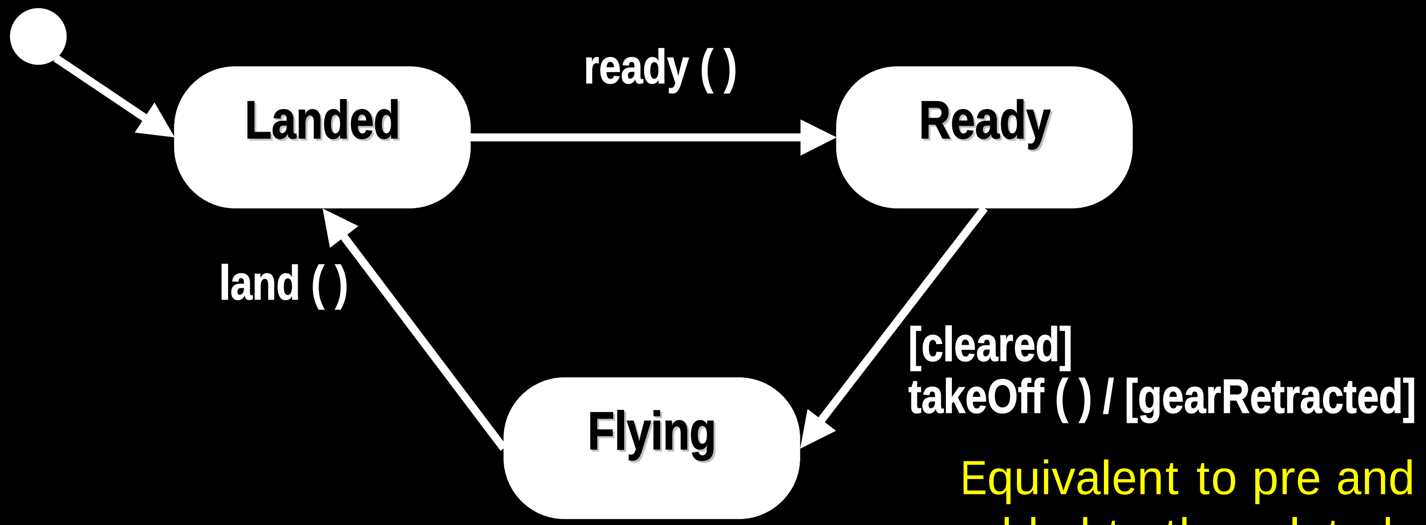


State Machine Redefinition



Protocol State Machines

- For imposing sequencing constraints on interface usage
  - (should not be confused with multi-party protocols)



Equivalent to pre and post conditions added to the related operations:

takeOff()

<u>Pre</u>

-in state "Ready"

-cleared for take off

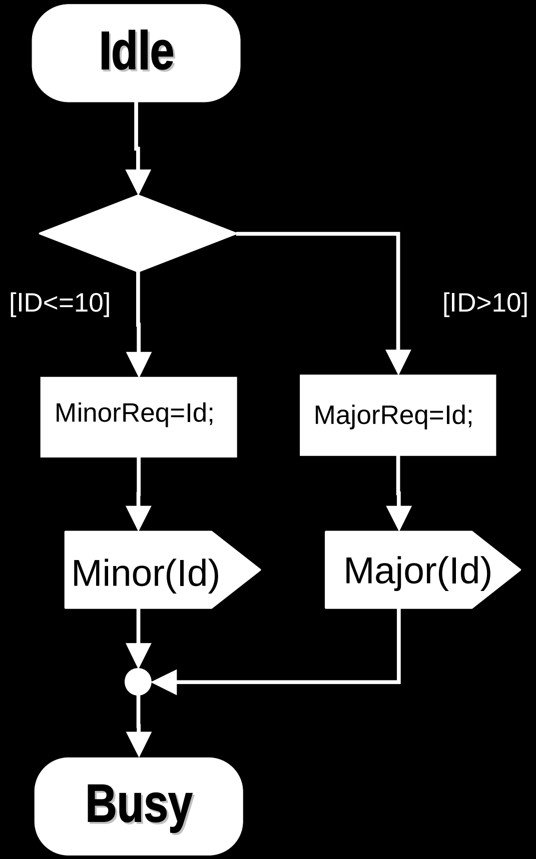
<u>Post</u>

-landing rear is retracted

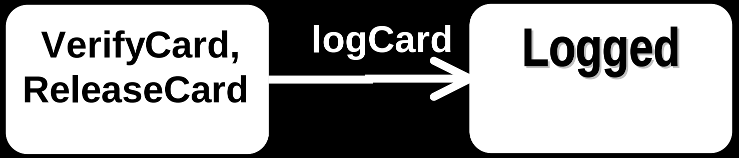
-in state "Flying"

Notation Enhancements

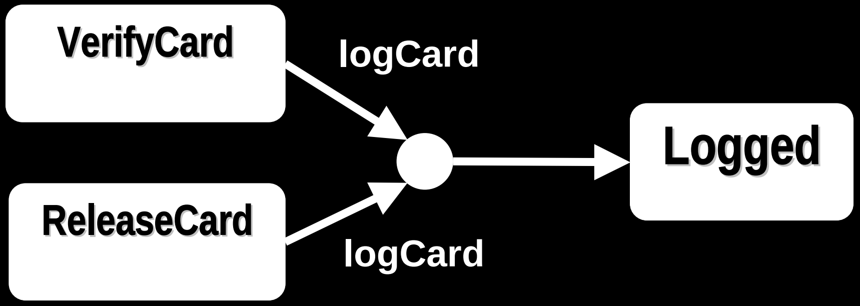
- Alternative transition notation



- State lists

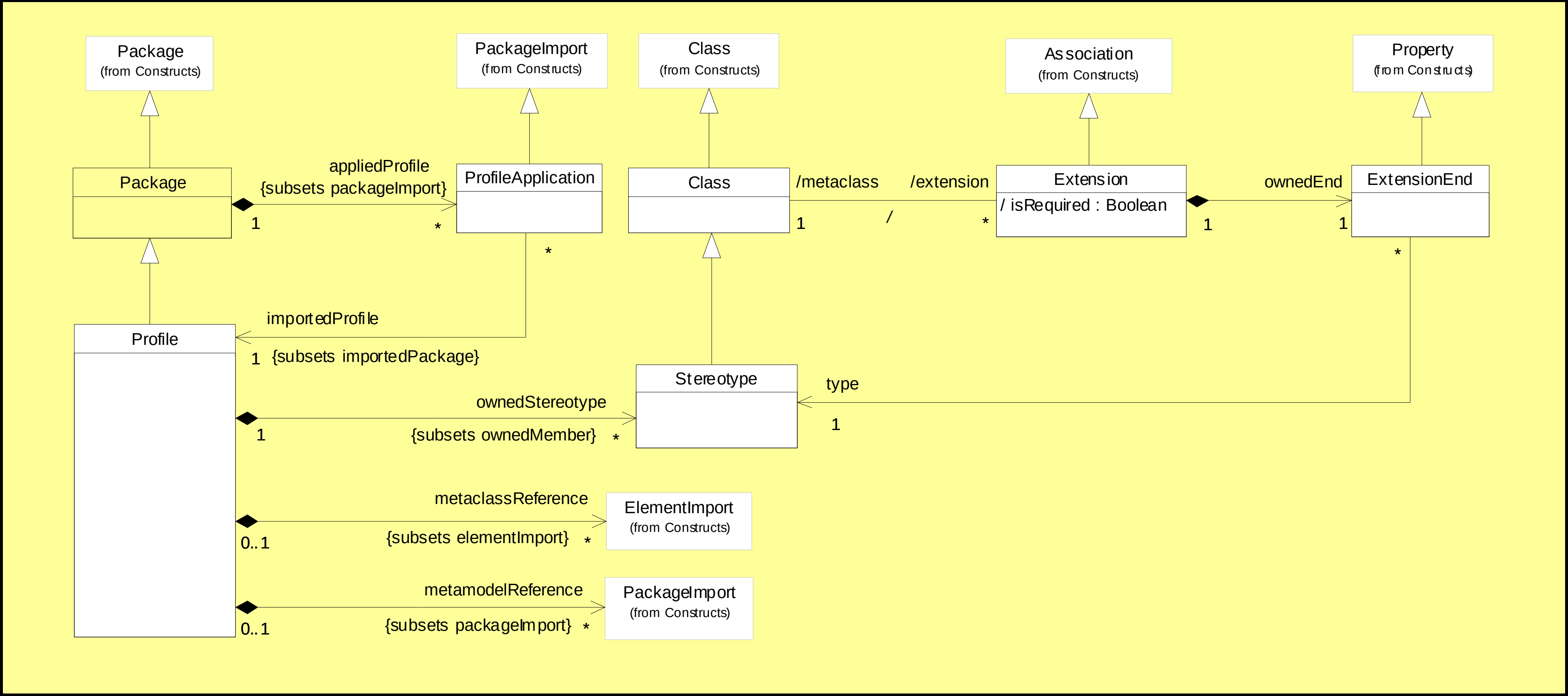


Is a notational shorthand for



Profiles: Metamodel

- Semantically equivalent to 1.x from a user's perspective
  - But, new notation introduced
  - Better fit to the new UML metamodel

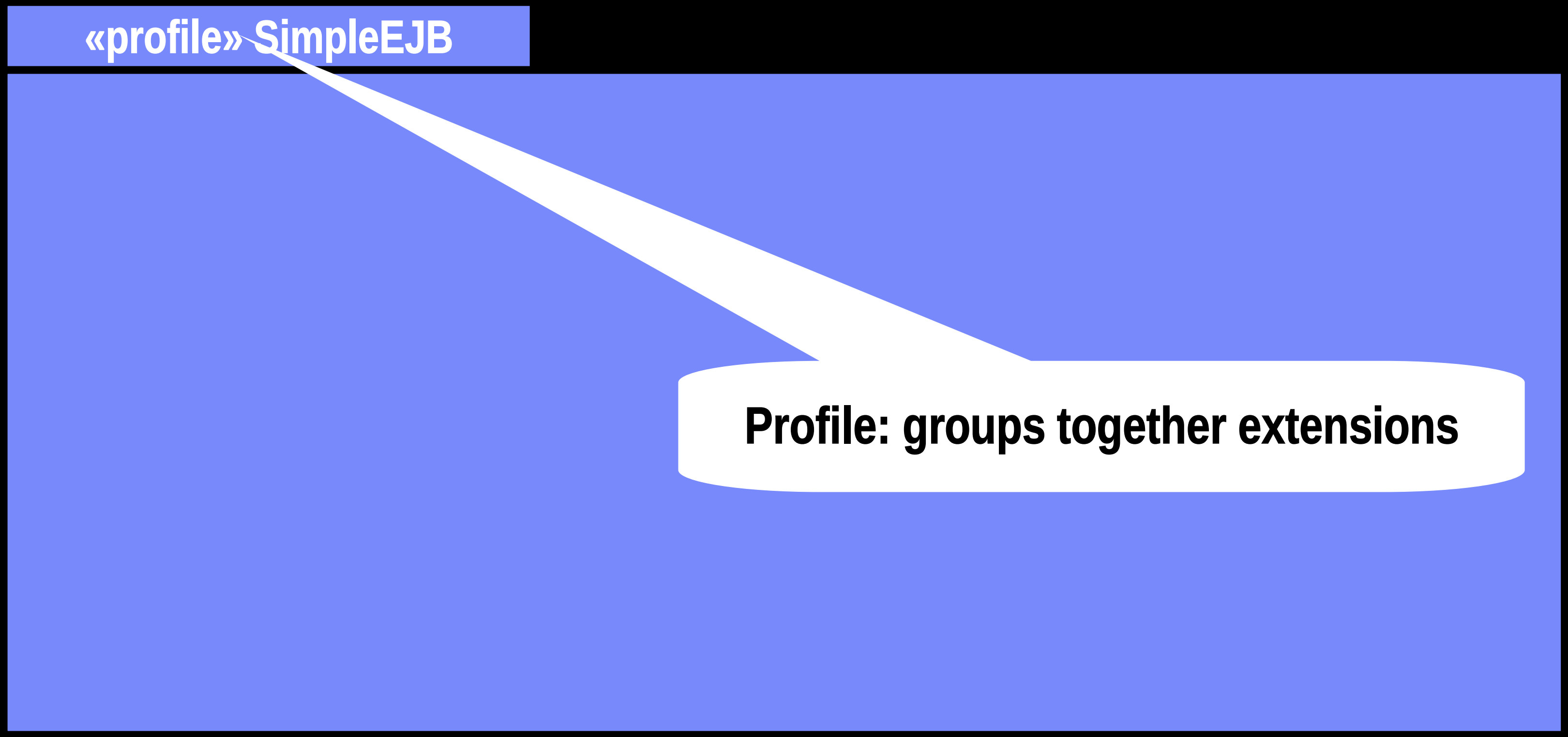


Profiles: Specializing UML

- Profiles specialize UML for specific domains
- Examples:
  - OMG standards:
    - EAI: Enterprise Application Integration
    - EDOC: Enterprise Distributed Object Computing
    - CORBA
    - Schedulability, Performance and Time
  - Proprietary:
    - UML-RT: UML for Real Time
- Metamodel concept
  - Defined on metamodel
  - Used on model

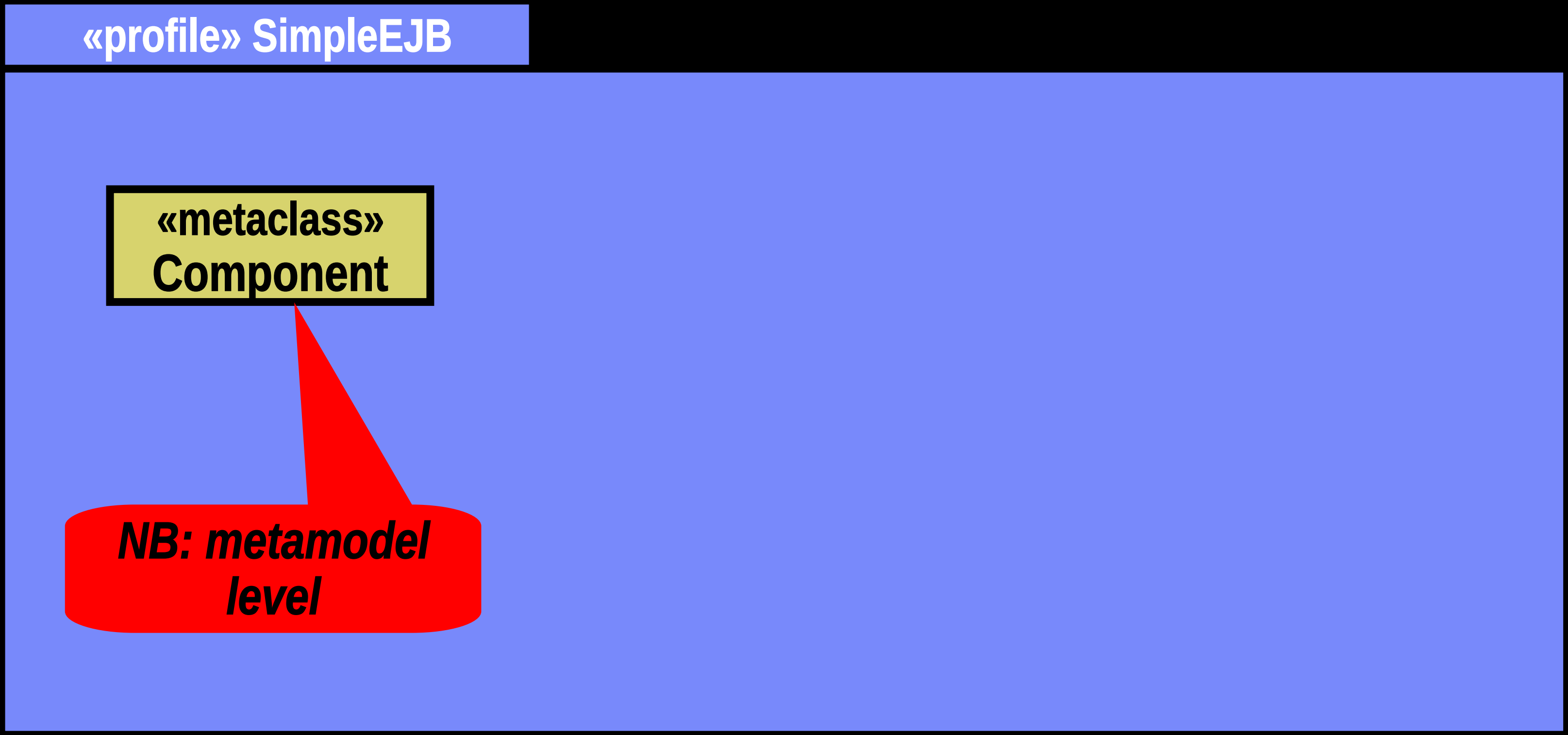
Profiles: Example

- *Profiles* group together *stereotypes*, *properties* and *constraints*



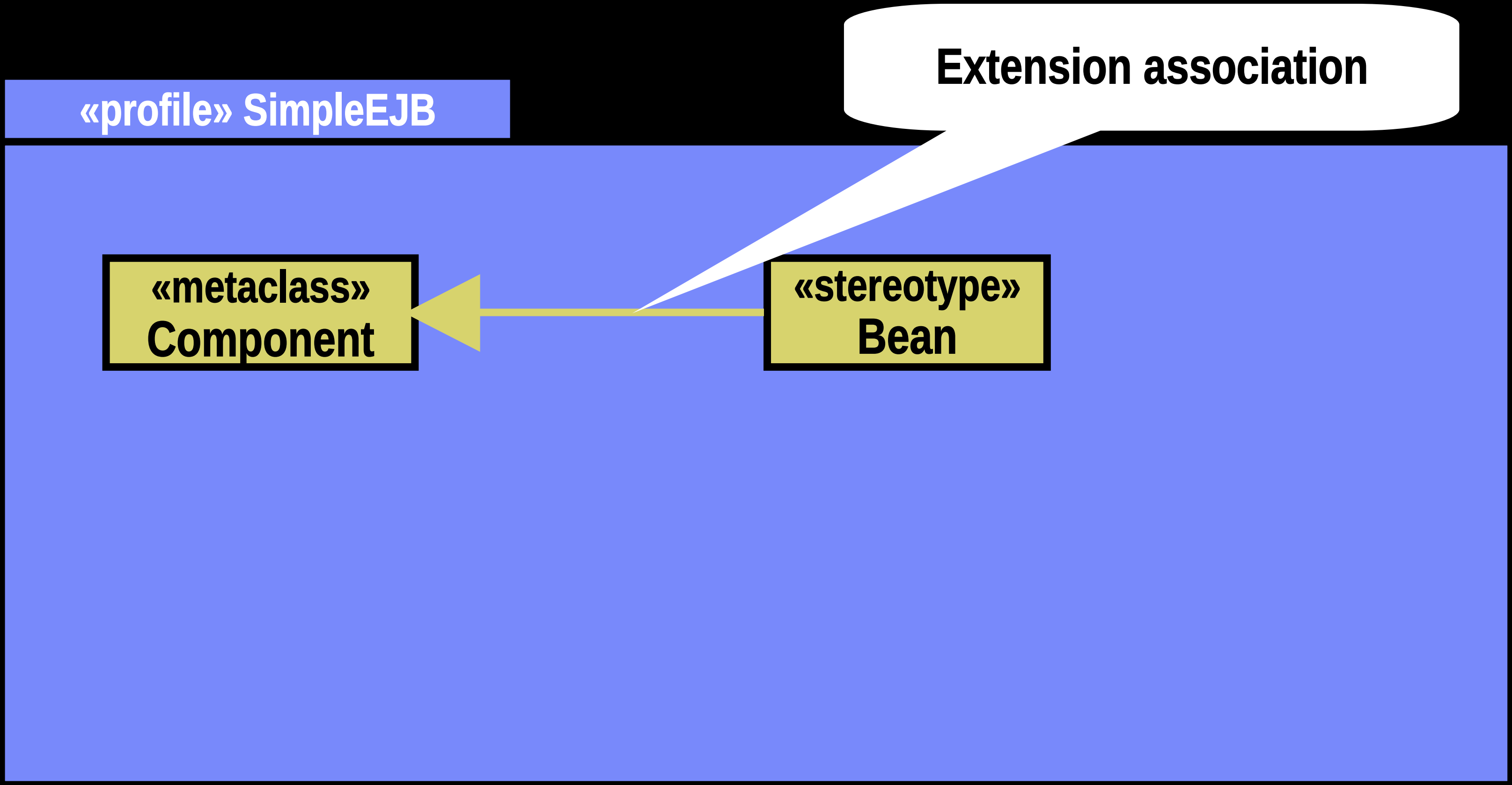
Profiles: Example

- *Profiles* group together *stereotypes*, *properties* and *constraints*



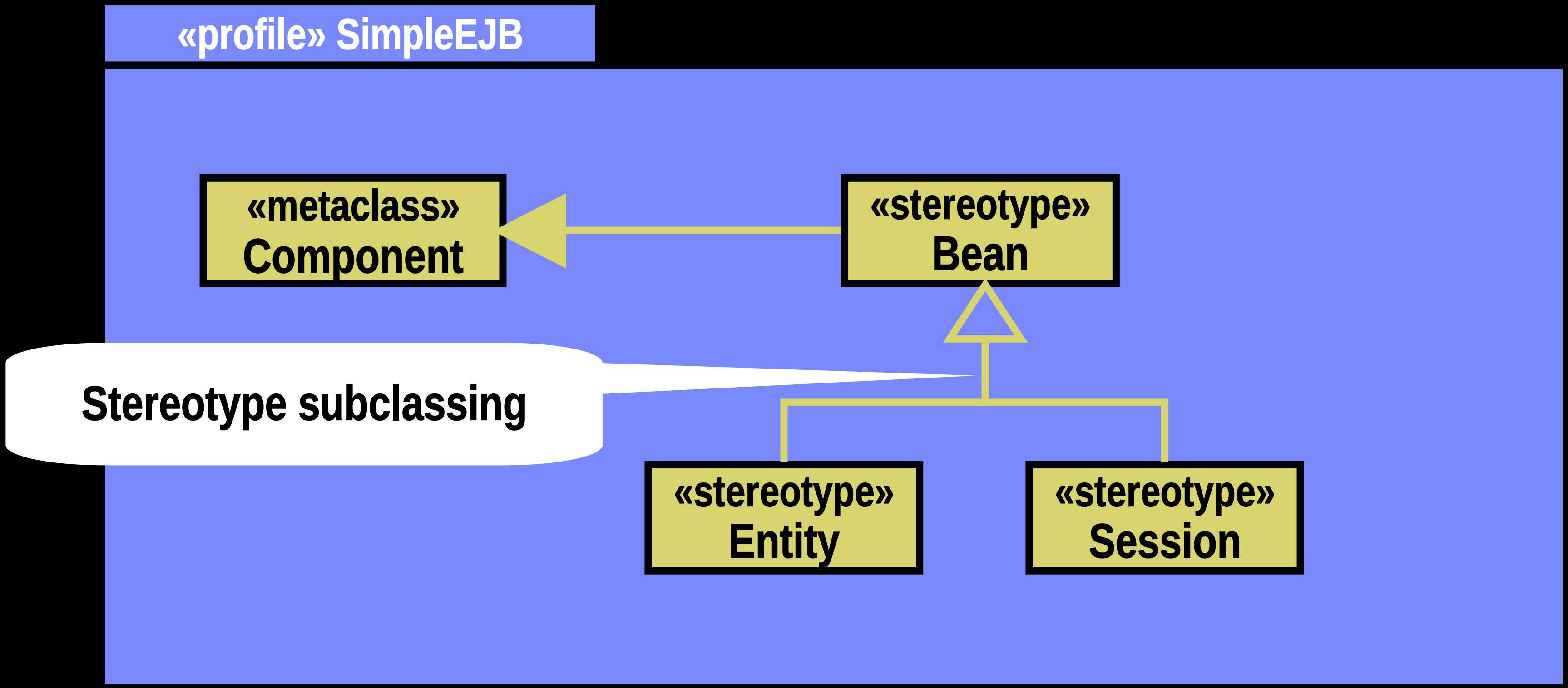
Profiles: Example

- *Profiles* group together *stereotypes*, *properties* and *constraints*



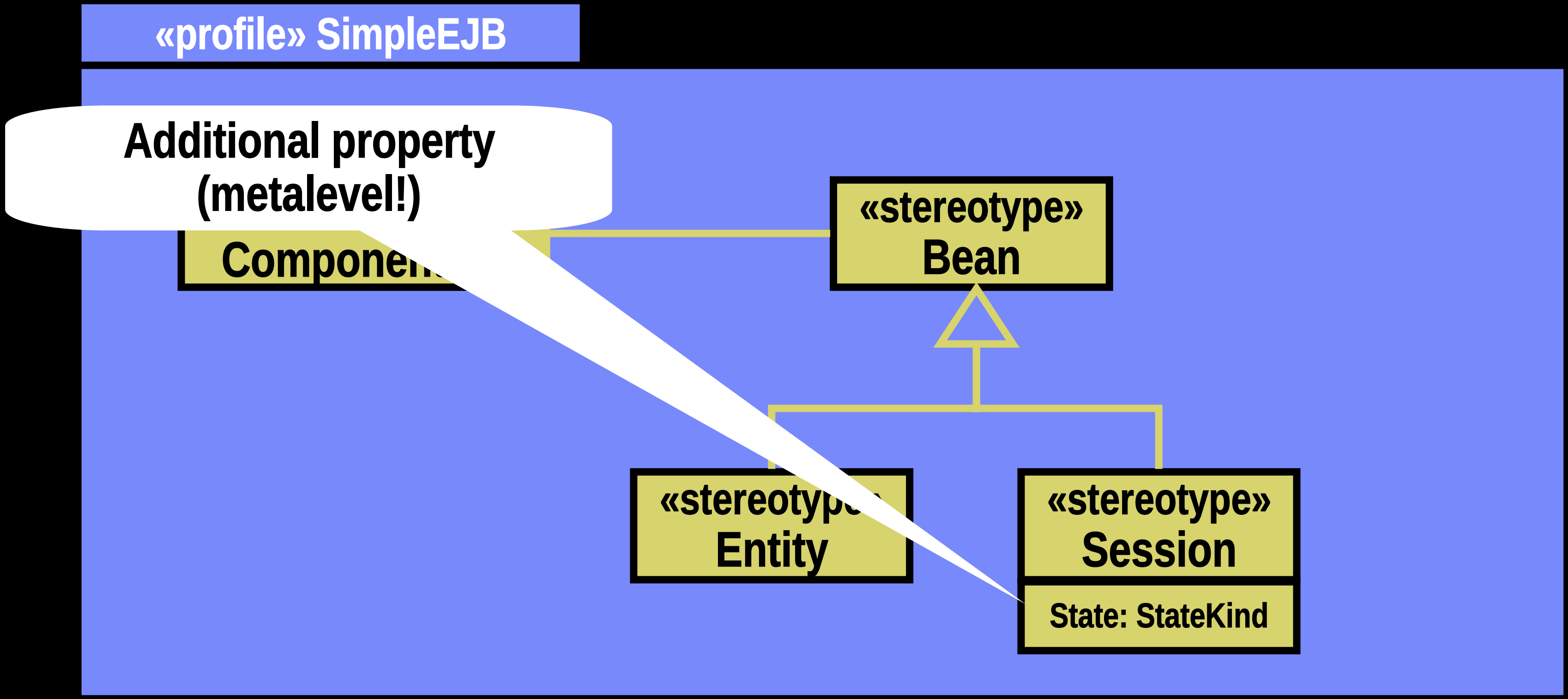
Profiles: Example

- *Profiles* group together *stereotypes*, *properties* and *constraints*



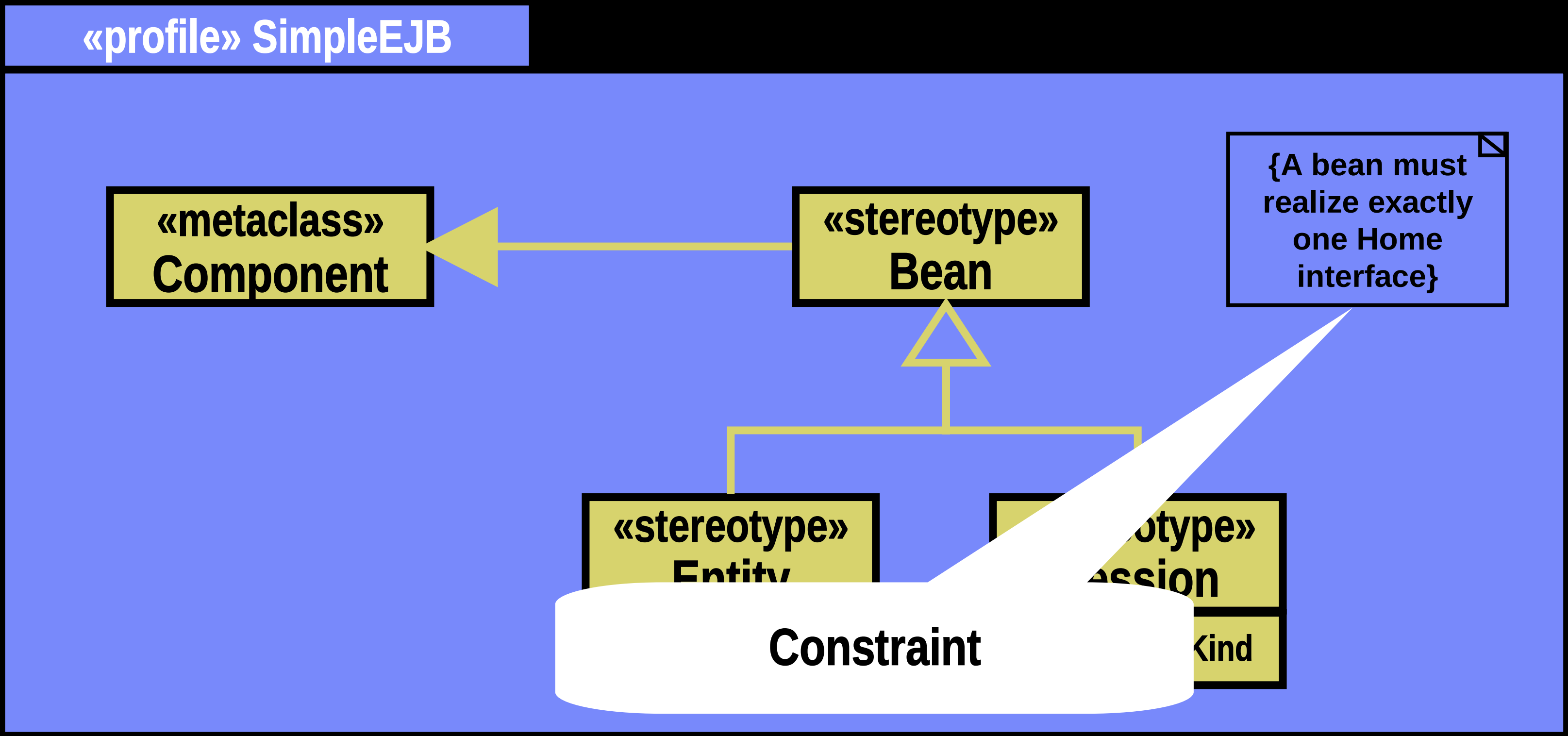
Profiles: Example

- *Profiles* group together *stereotypes*, *properties* and *constraints*



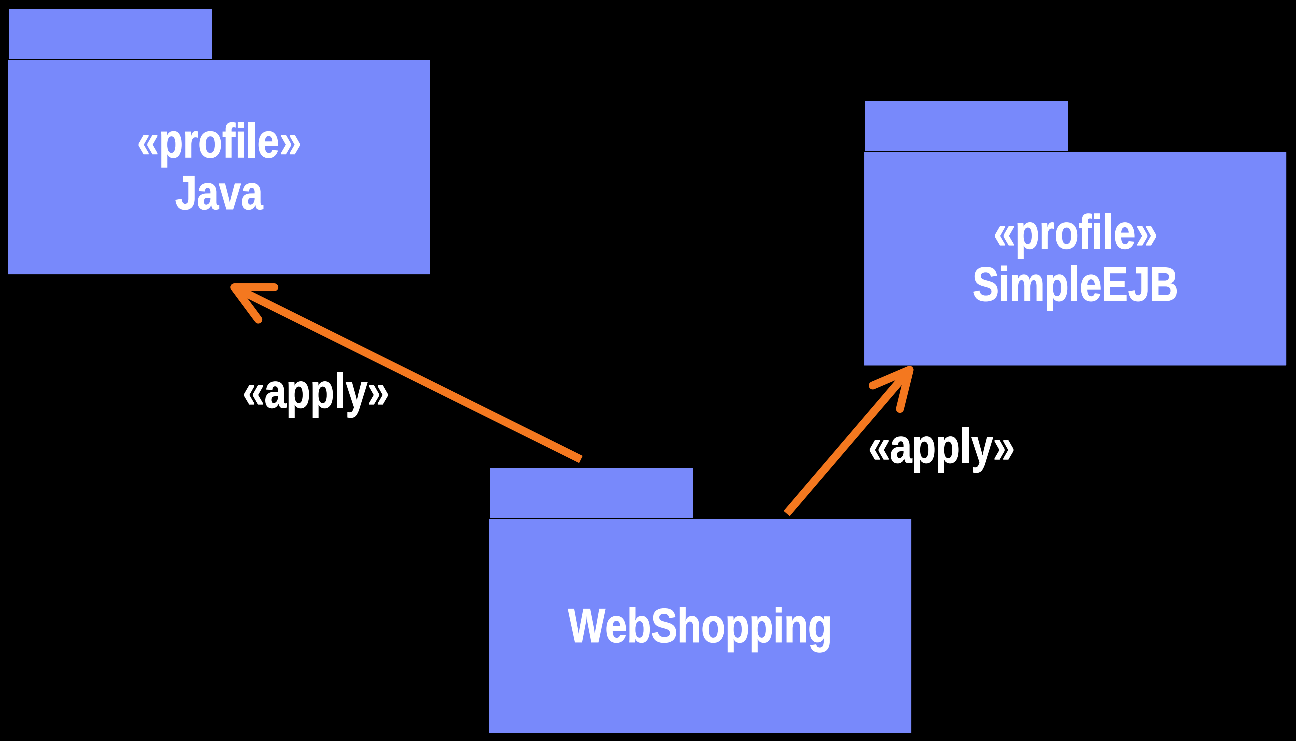
Profiles: Example

- *Profiles* group together *stereotypes*, *properties* and *constraints*



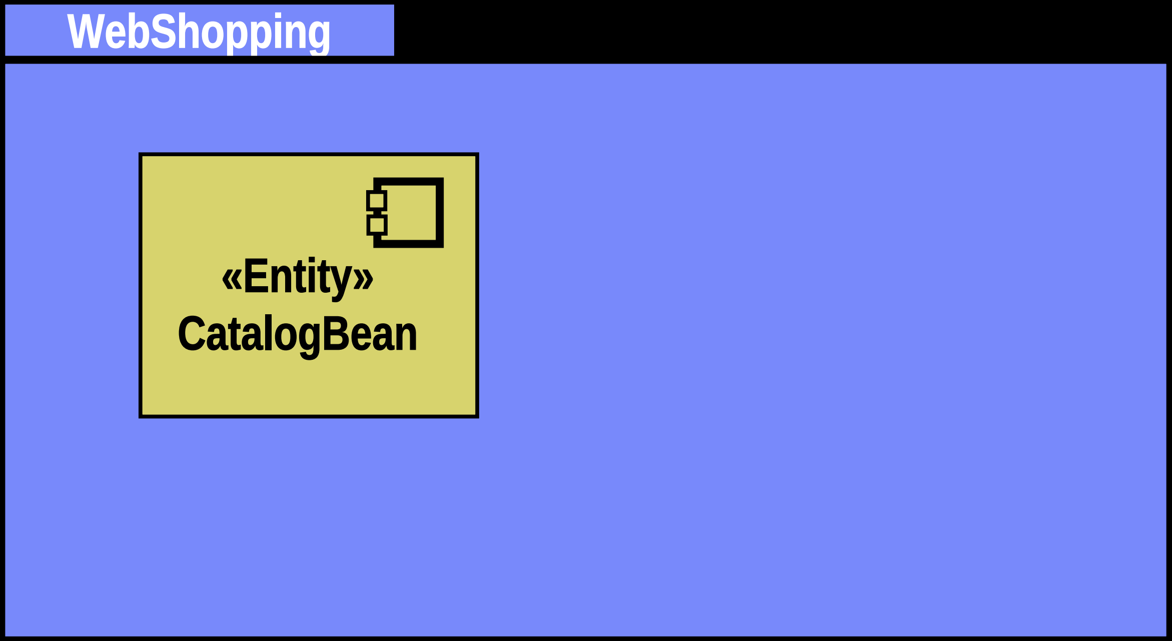
Applying the Profile (1)

- Apply the profile defined at the *metamodel* level to a *model*
  - As a profile…



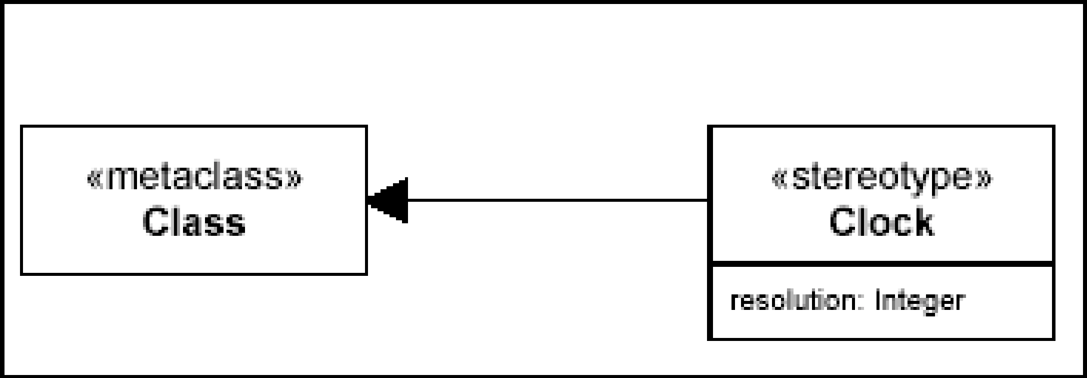
Applying the Profile (2)

- Apply the profile defined at the *metamodel* level to a *model*
  - As a profile…
  - …and to individual classes



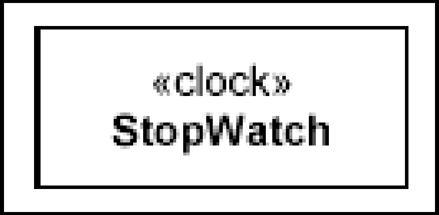
You Will Understand Metamodelling When..

- You understand how the metamodel view…



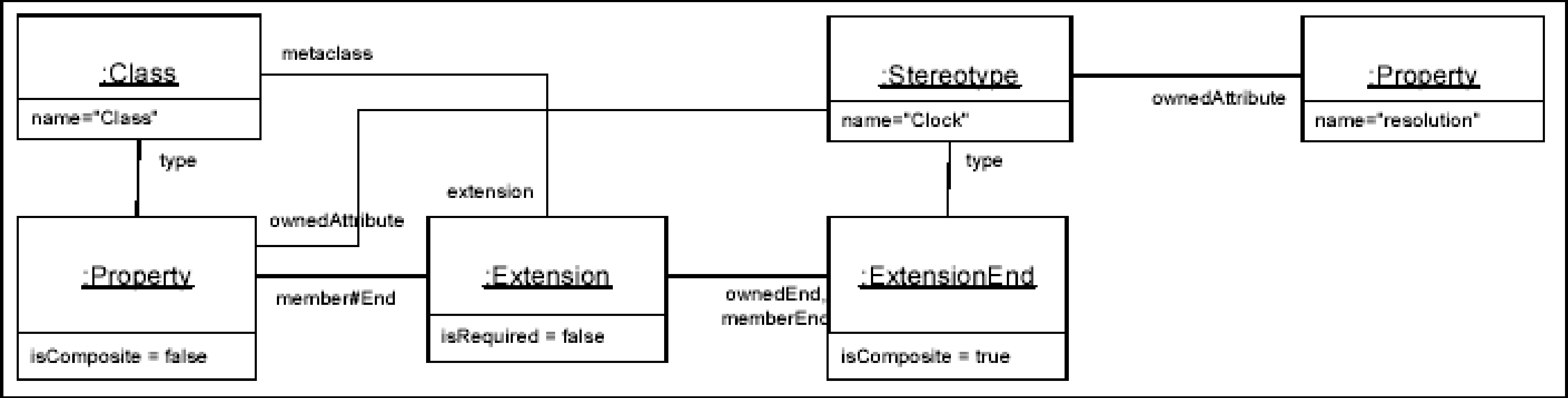
You Will Understand Metamodelling When..

- You understand how the metamodel view…
- …relates to the model view…



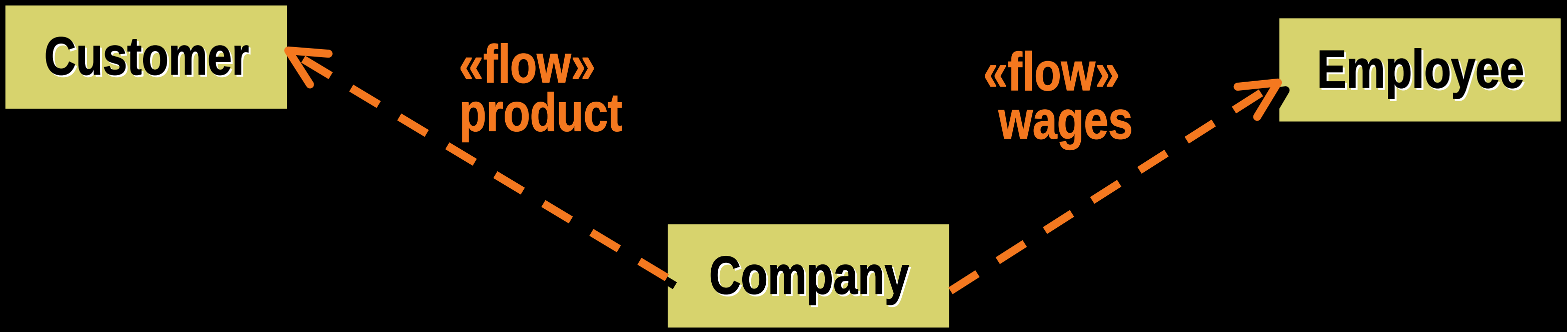
You Will Understand Metamodelling When..

- You understand how the metamodel view…
- …relates to the model view…
- …and the metamodel instance view



Information Flows

- For specifying exchanges of information items between active entities at a very abstract level
  - Do not specify details of the information (e.g., type)
  - Do not specify how the information is relayed
  - Do not specify the relative ordering of information flows



Summary: Model-Driven Development

- Software has a unique advantage when it comes to using engineering models for development
  - Seamless progression from design to product
- MDD has already indicated that it can significantly improve the reliability and productivity of software development
  - Proven technologies
  - Dedicated standards
  - Increased use of automation
- The OMG has responded to this potential with the MDA initiative
- MOF and UML are two core OMG standard technologies that are part of MDA

Summary: UML 2.0

- First major revision of UML
- Original standard had to be adjusted to deal with
  - MDD requirements (precision, code generation, executability)
- UML 2.0 characterized by
  - Small number of new features + consolidation of existing ones
  - Scaleable to large software systems (architectural modeling capabilities)
  - Modular structure for easier adoption (core + optional specialized sub-languages)
  - Increased semantic precision and conceptual clarity
  - Suitable foundation for MDA (executable models, full code generation)

QUESTIONS?

(hogg@ca.ibm.com)