IBM Software Group

Introduction to UML 2.0

Terry Quatrani

UML Evangelist

ON DEMAND BUSINESS

Session Objectives

- After completing this session, you should be able to:
  - Identify the different UML diagrams
  - Describe the purpose of each diagram
  - State where diagrams can be used

Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML

Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML

The Importance of Modeling

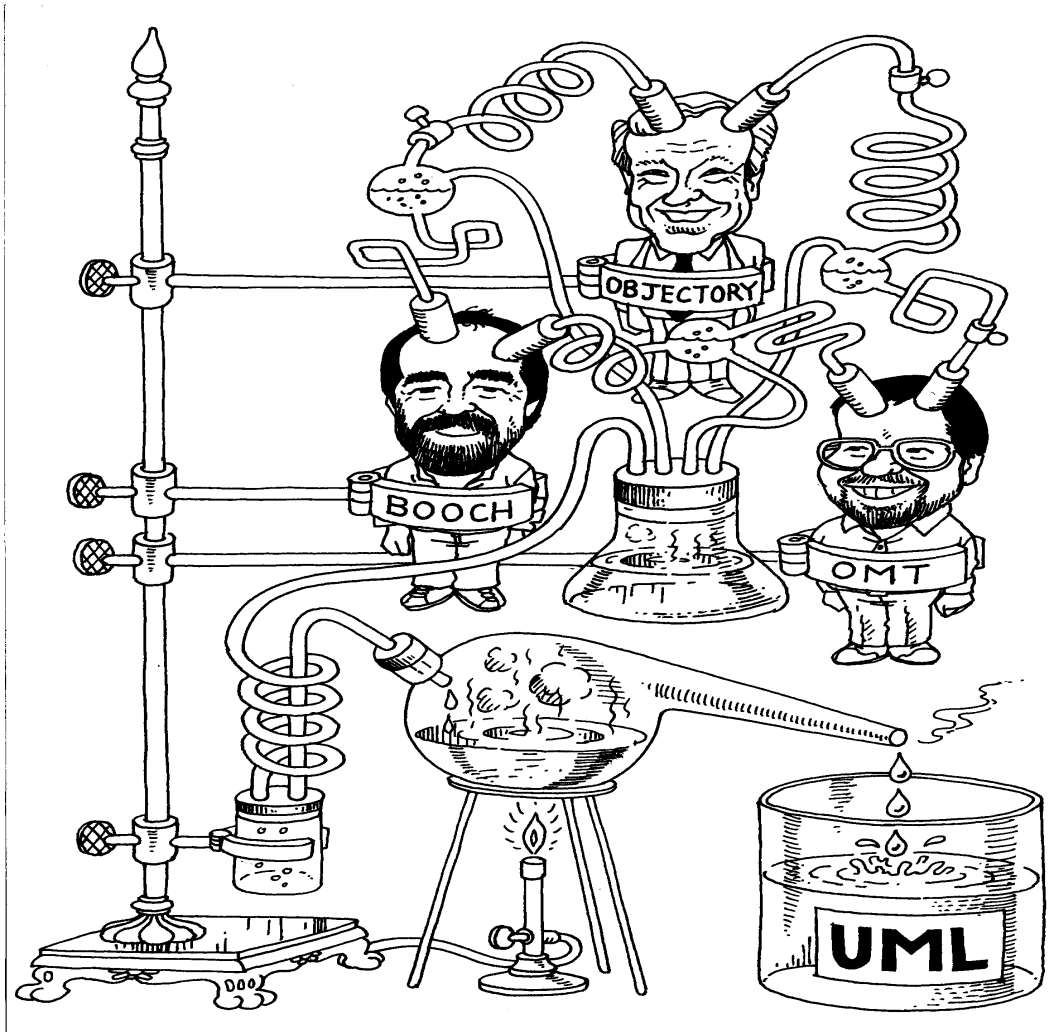
Why do we model?

- To manage complexity
- To detect errors and omissions early in the lifecycle
- To communicate with stakeholders
- To understand requirements
- To drive implementation
- To understand the impact of change
- To ensure that resources are deployed efficiently

Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML

The Unified Modeling Language



- The UML is the standard language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system

Blobs with writing in their hair
And small adornments in the air
And has-relations everywhere
I've looked at clouds that way.
But now I've purged them from my Sun
Eliminated every one
So many things I would have done
To drive the clouds away.
I've looked at clouds from both sides now
Both in and out, and still somehow
It's clouds' delusions I appall
I really can't stand clouds at all…
Balls for multiplicity
Black and white for clarity
And data flows arranged in trees
Were part of OMT.
But now I've had to let them go
We'll do it differently, you know
'Cause Grady said, they've got to go
We can't use OMT.

Approach to Evolving UML 2.0

- 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

Formal RFP Requirements

- Infrastructure – UML internals
  - More precise conceptual base for better MDA support
  - MOF-UML alignment
- Superstructure – User-level features
  - New capabilities for large-scale software systems
  - Consolidation of existing features
- OCL – Constraint language
  - Full conceptual alignment with UML
- 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)

OCL Requirements

- Define an OCL metamodel and align it with the UML metamodel
  - OCL navigates through class and object diagrams ⇒ must share a common definition of Class, Association, Multiplicity, etc.
- New modeling features available to general UML users
  - Beyond constraints
  - 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)

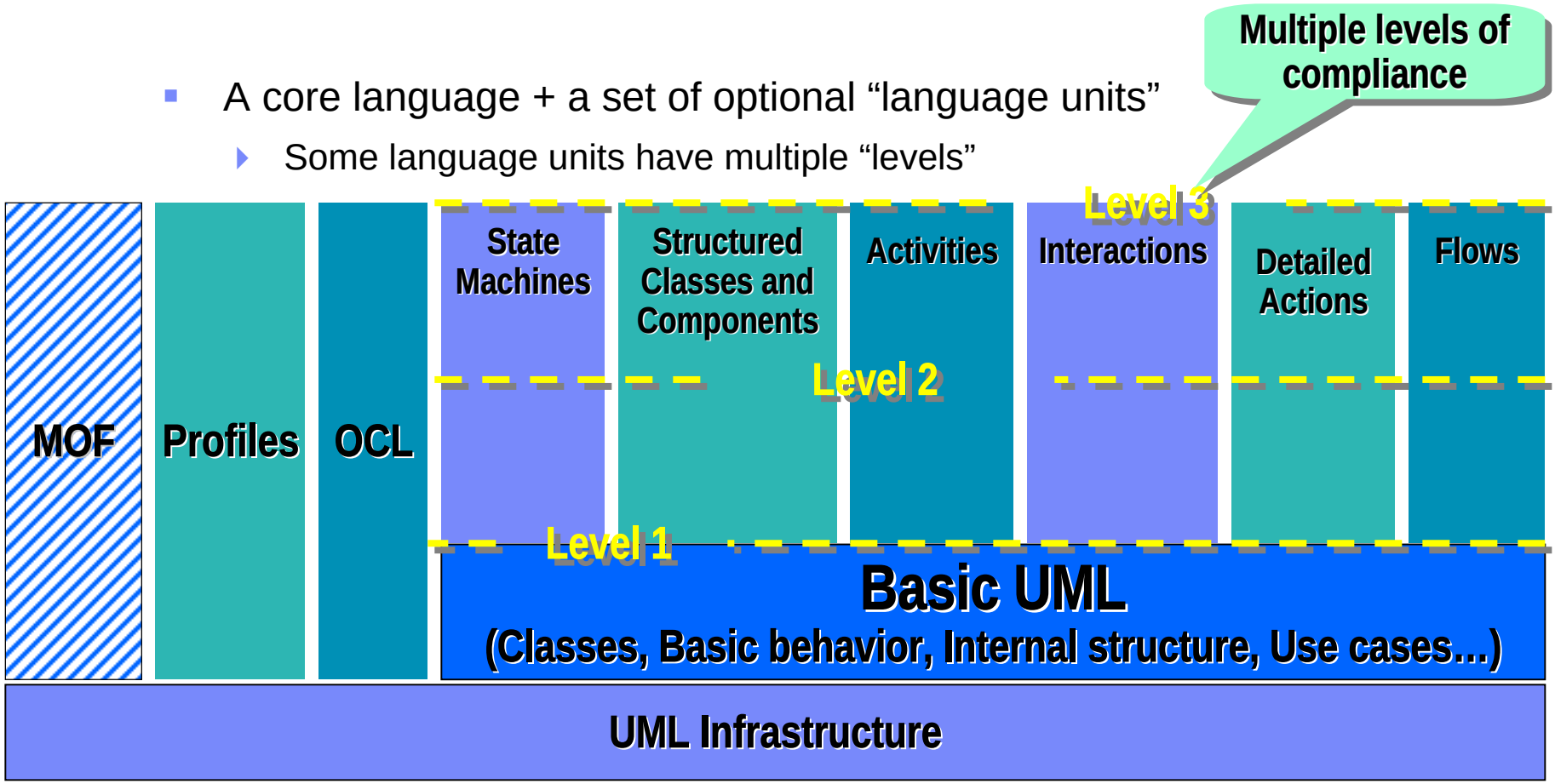
- More direct support for architectural modeling
  - Based on existing architectural description languages (UML-RT, ACME, SDL, etc.)
  - Reusable interaction specifications (UML-RT protocols)
- Behavior harmonization
  - Generalized notion of behavior and causality
  - Support choice of formalisms for 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

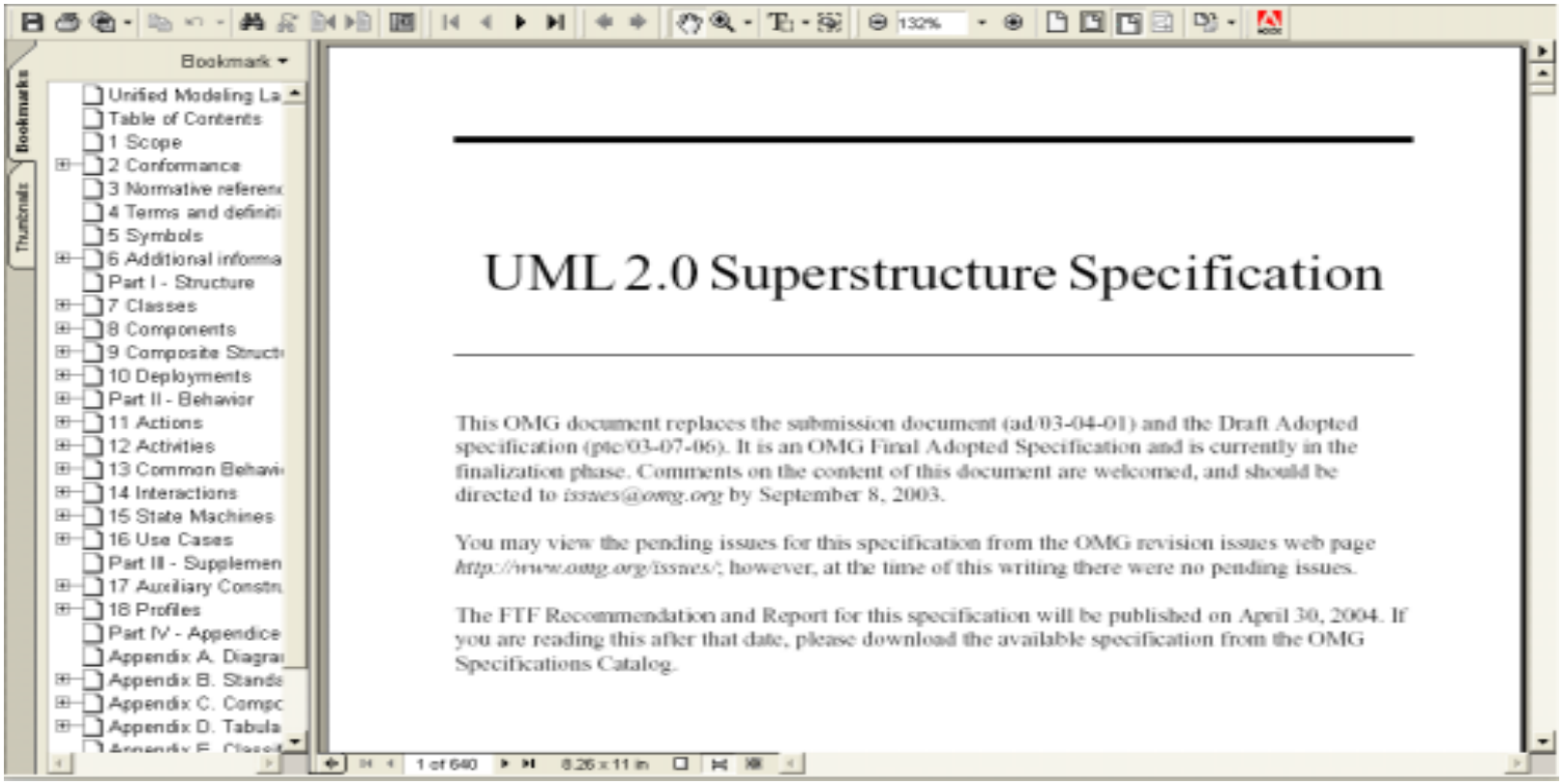
Language Architecture

- A core language + a set of optional “language units”
  - Some language units have multiple “levels”



The UML 2.0 Specification

- Can be downloaded from http://www.omg.org/cgi-bin/doc?ptc/2004-10-02

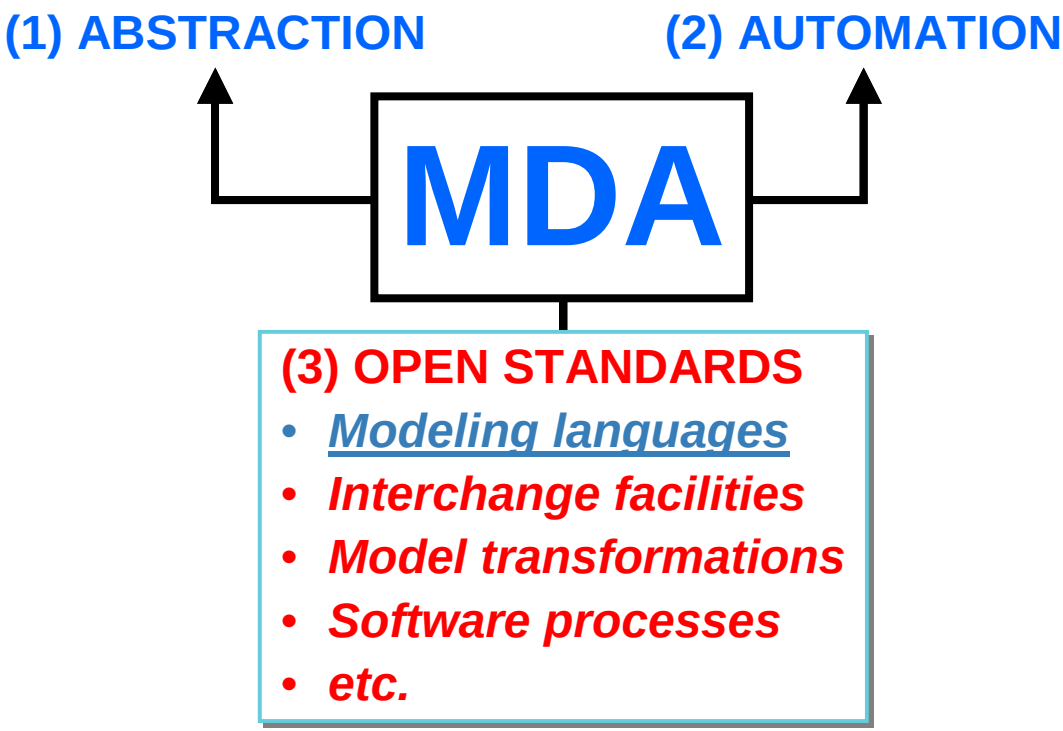


Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML

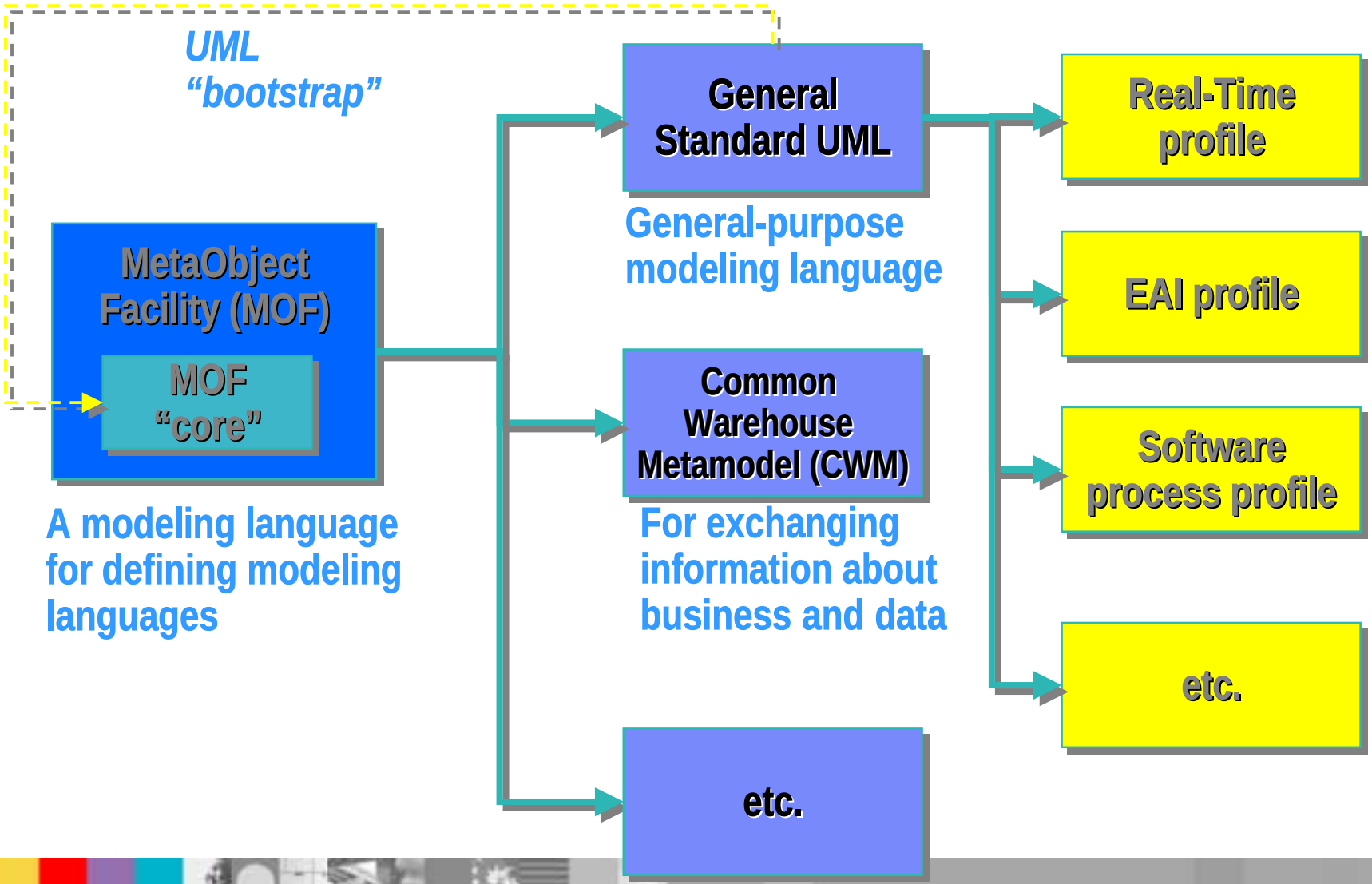
Model-Driven Architecture (MDA)

- An OMG initiative to support model-driven development through a series of open standards



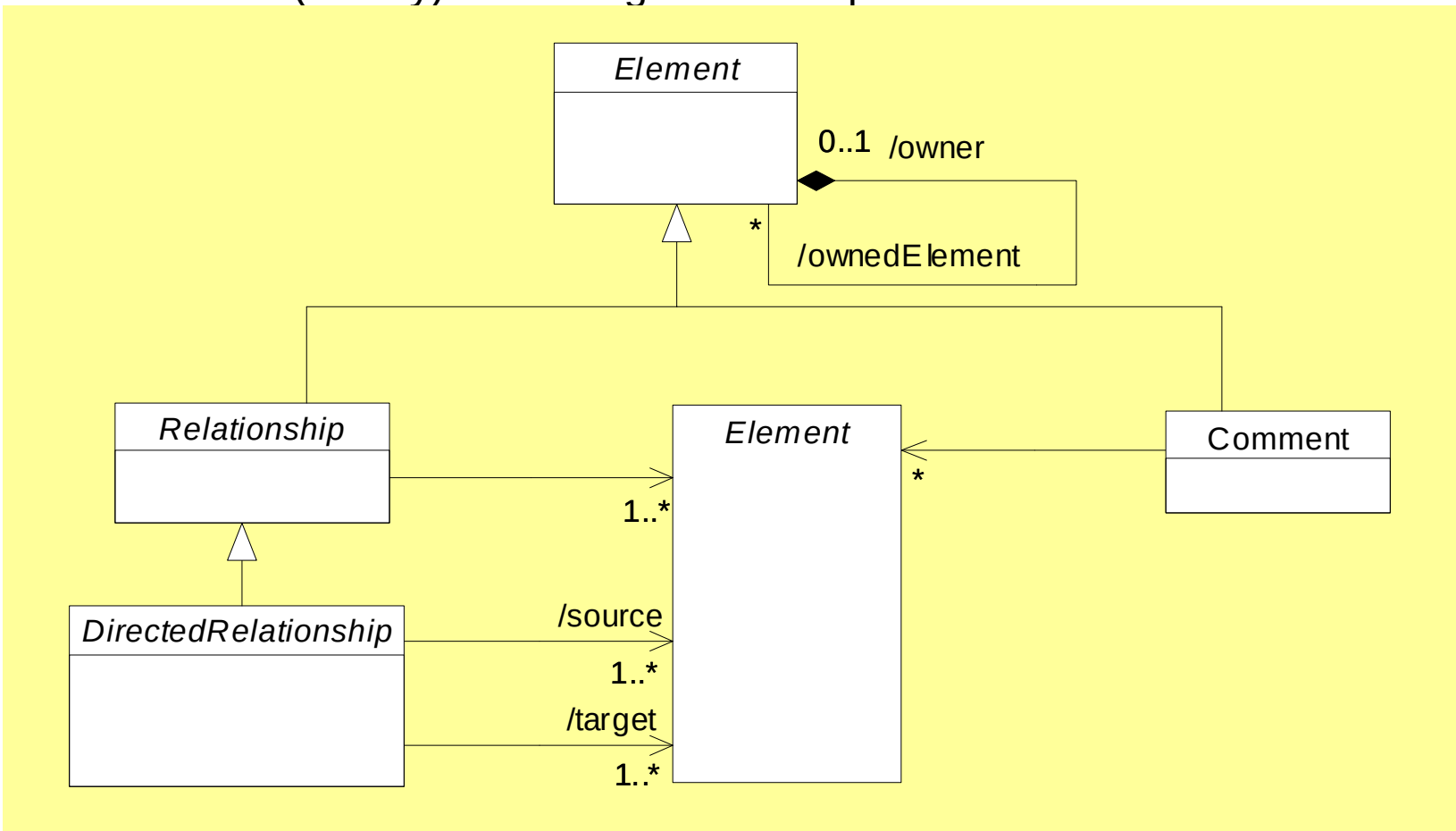
- Set of modeling languages for specific purposes

The Languages of MDA



MOF (Metamodel) Example

- Uses (mostly) class diagram concepts to define

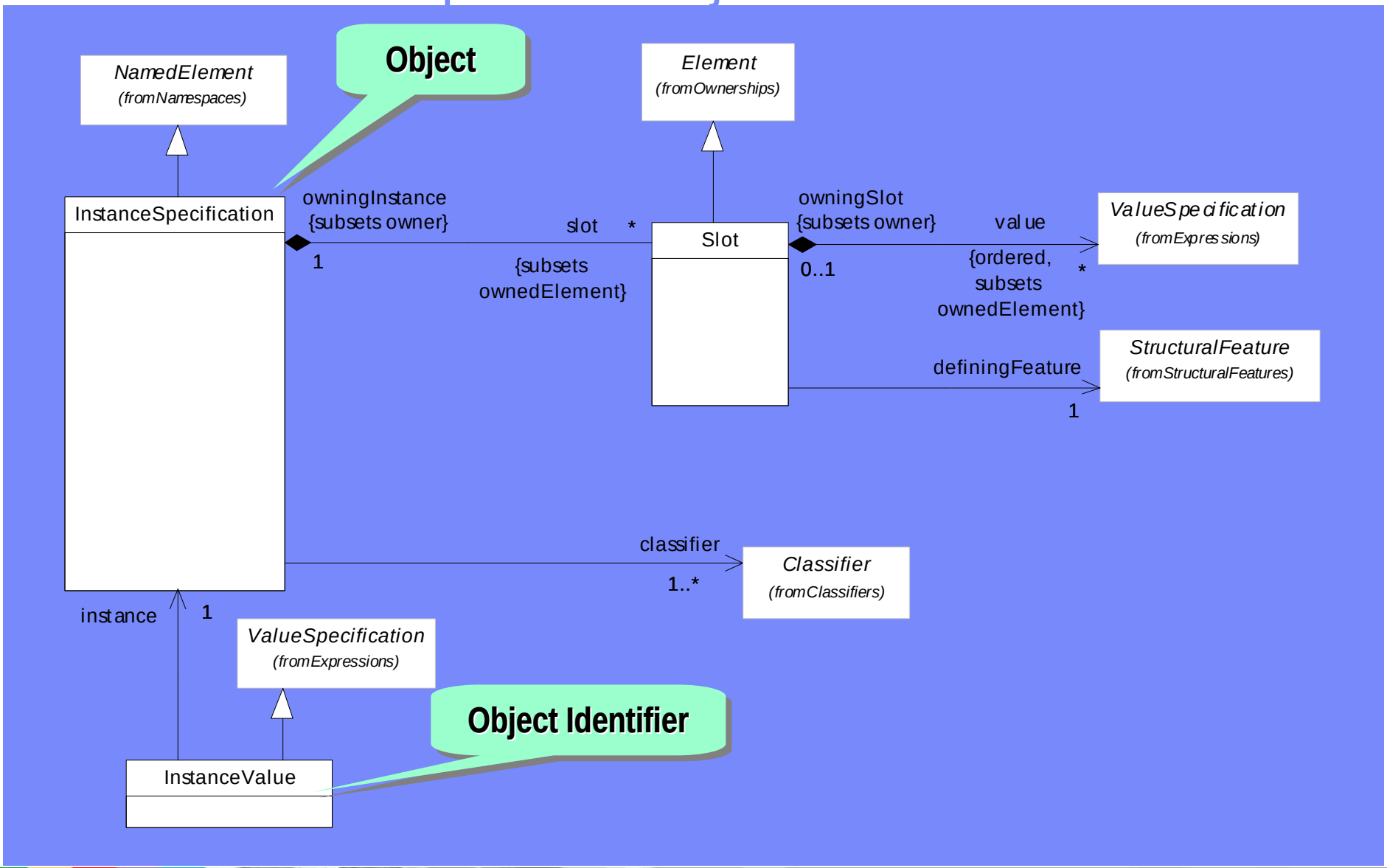


Agenda

- The Importance of Modeling
- The Unified Modeling Language
- Model-Driven Architecture
- UML Diagrams
- Extending the UML

Meta-models and Notations

Metamodel Description of Objects



Metamodel Structure

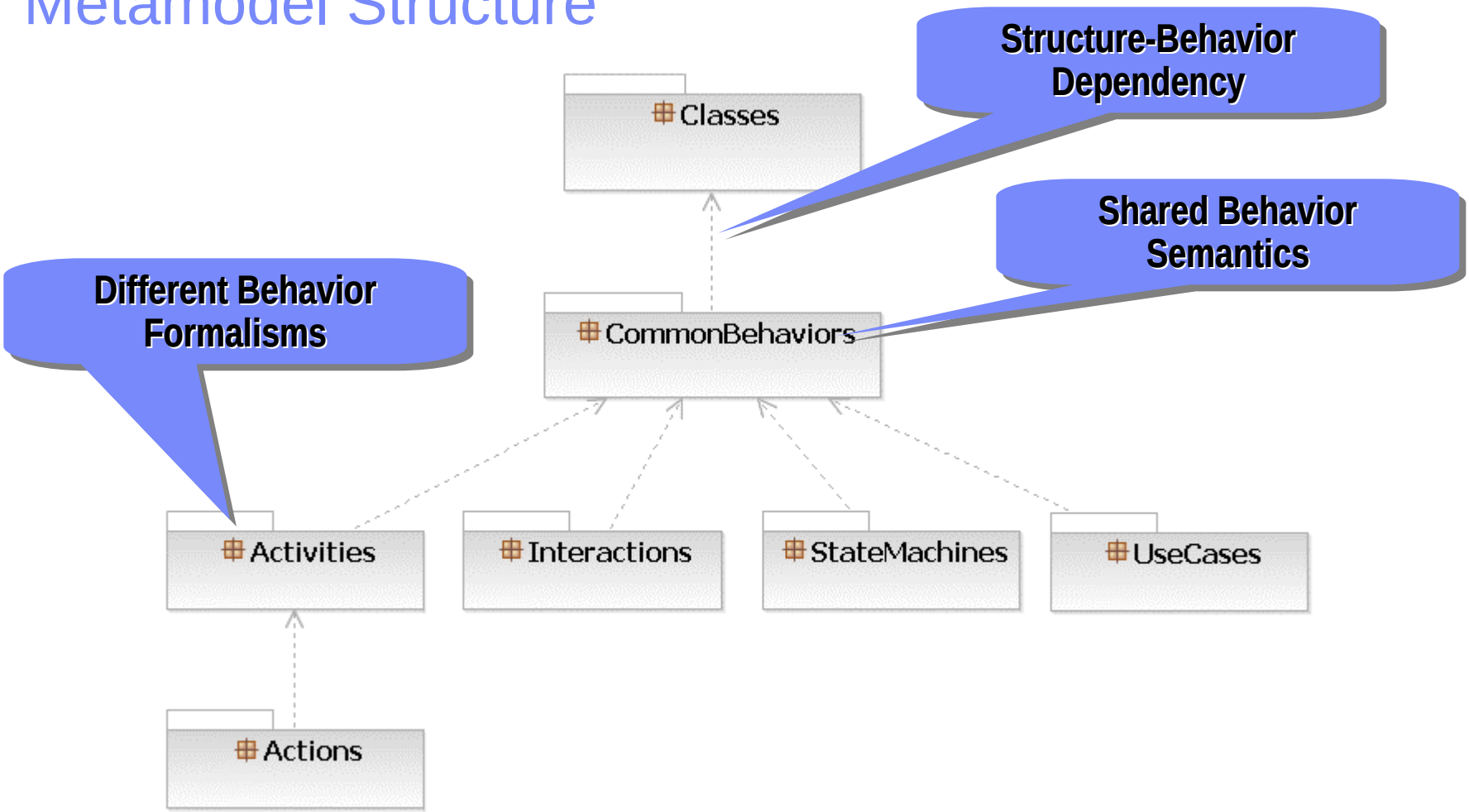
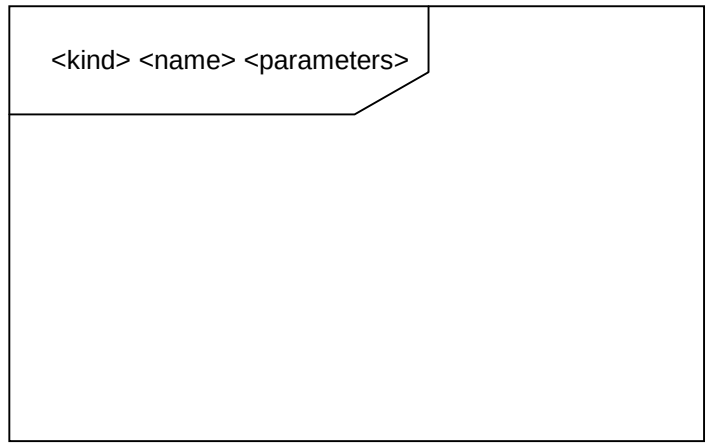
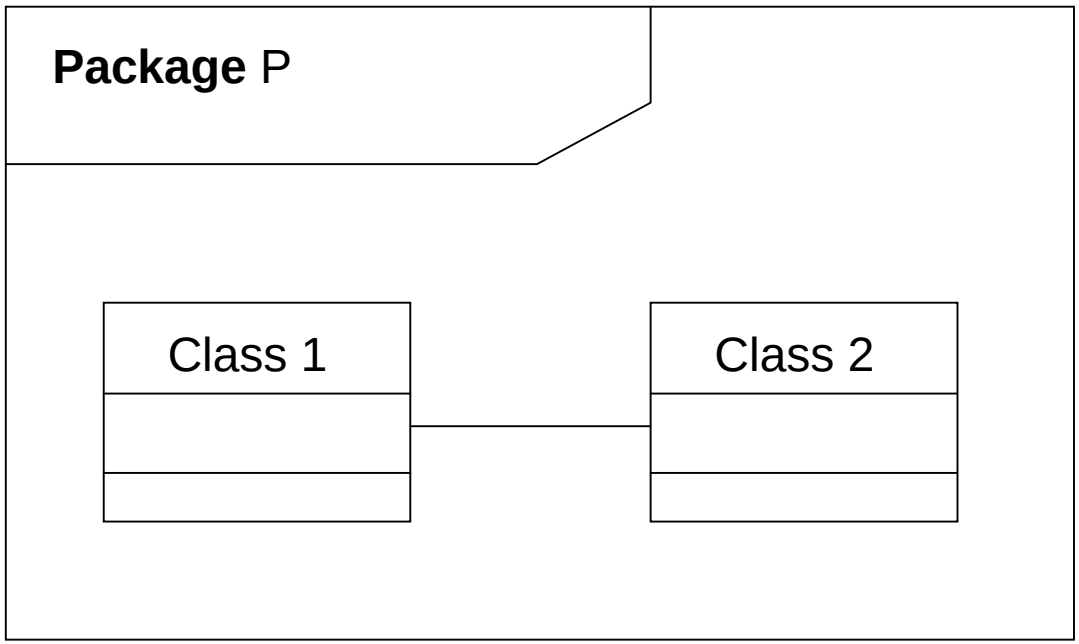


Diagram Elements

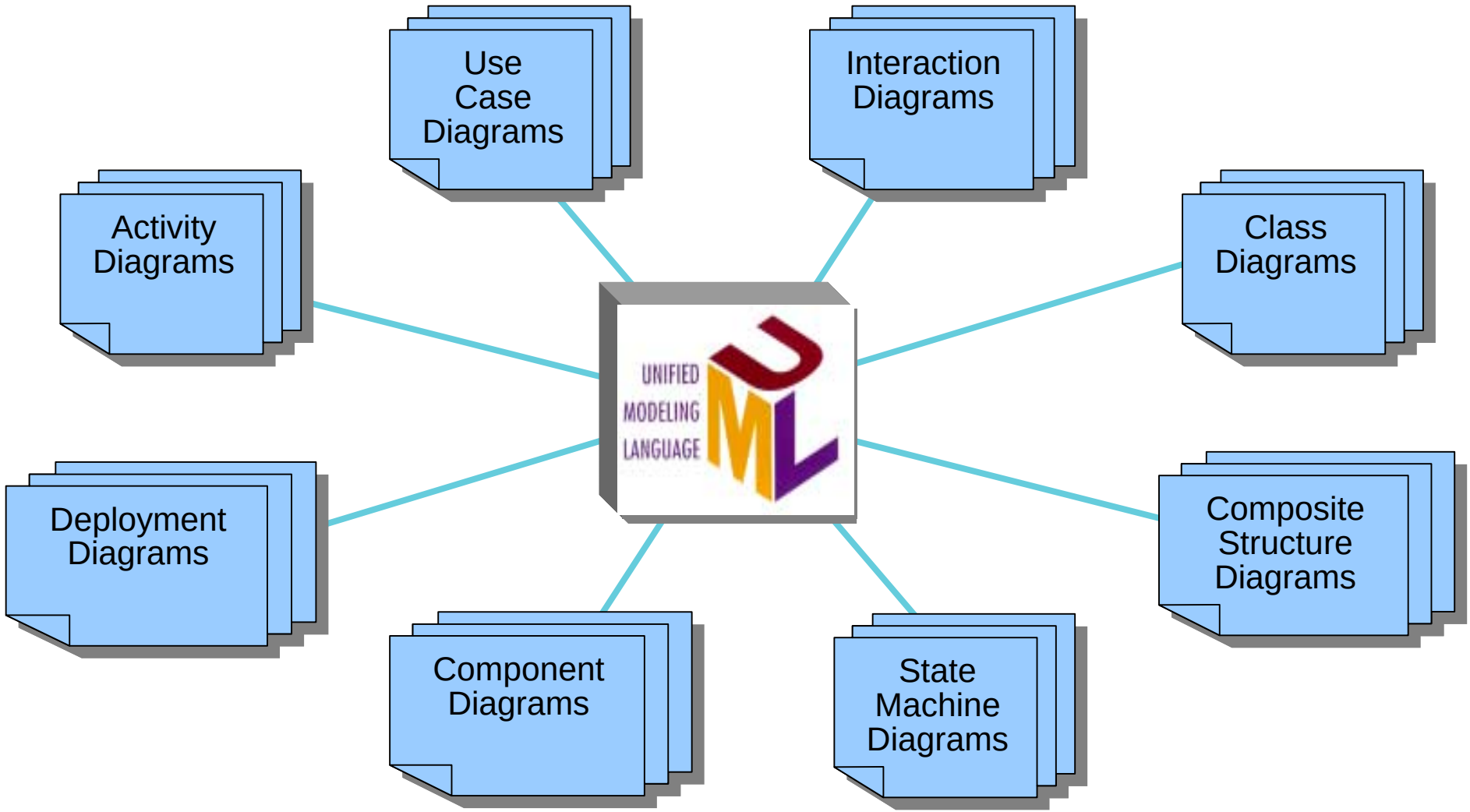
- Each diagram has a frame, a content area and a heading
- The frame is a rectangle and is used to denote a border
  - Frame is optional
- The heading is a string contained in a name tag which is a rectangle with cut off corners in the upper left hand corner of the frame
  - Format [<kind>] [<name>] [<parameters>]
  - <kind> can be activity, class, component, interaction, package, state machine, use case



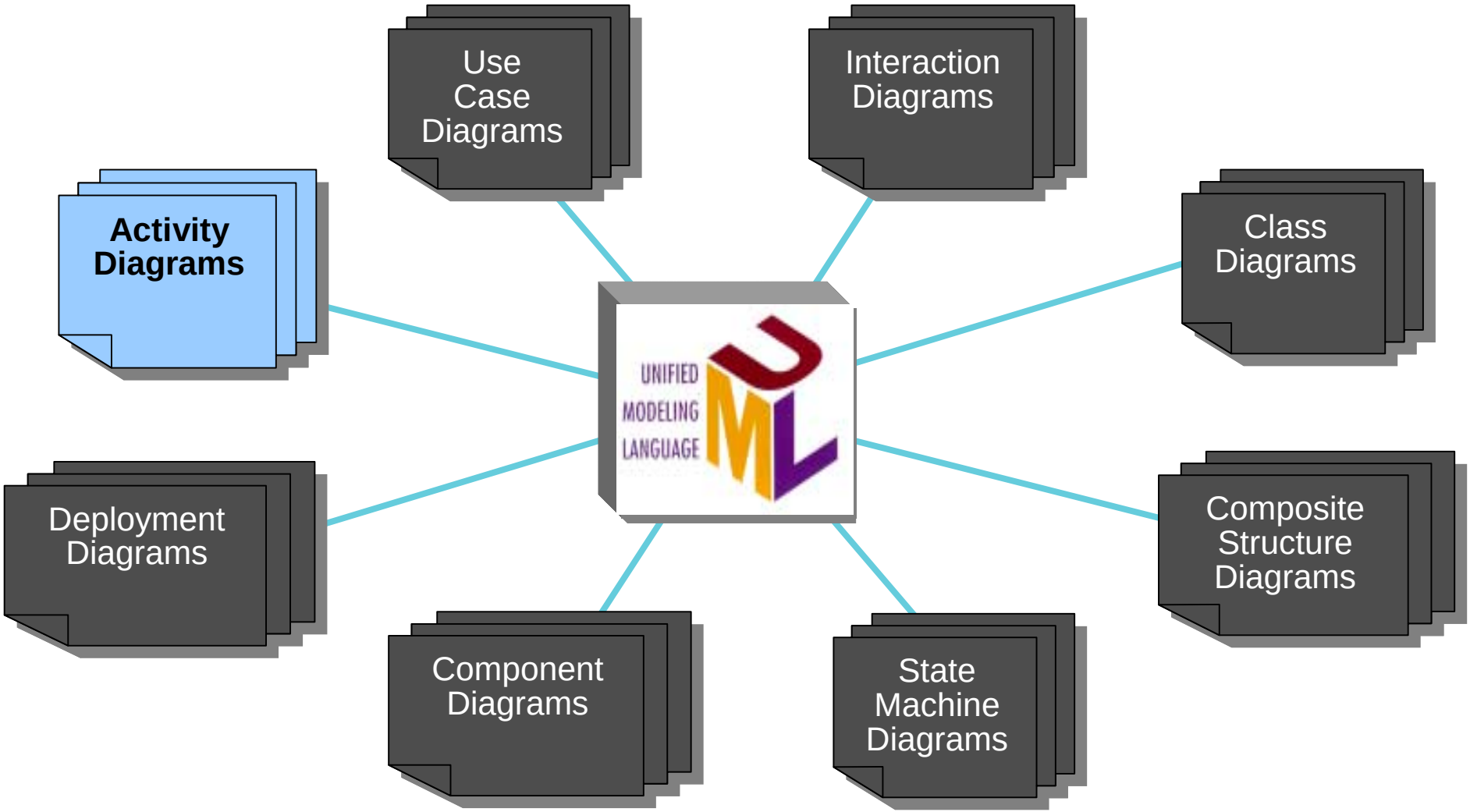
Frame Example



UML 2.0 Diagrams



UML 2.0 Diagrams

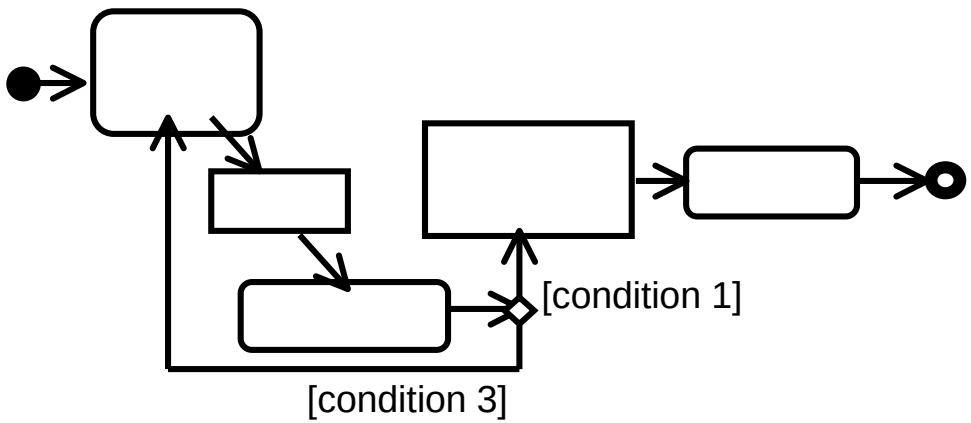


Activities

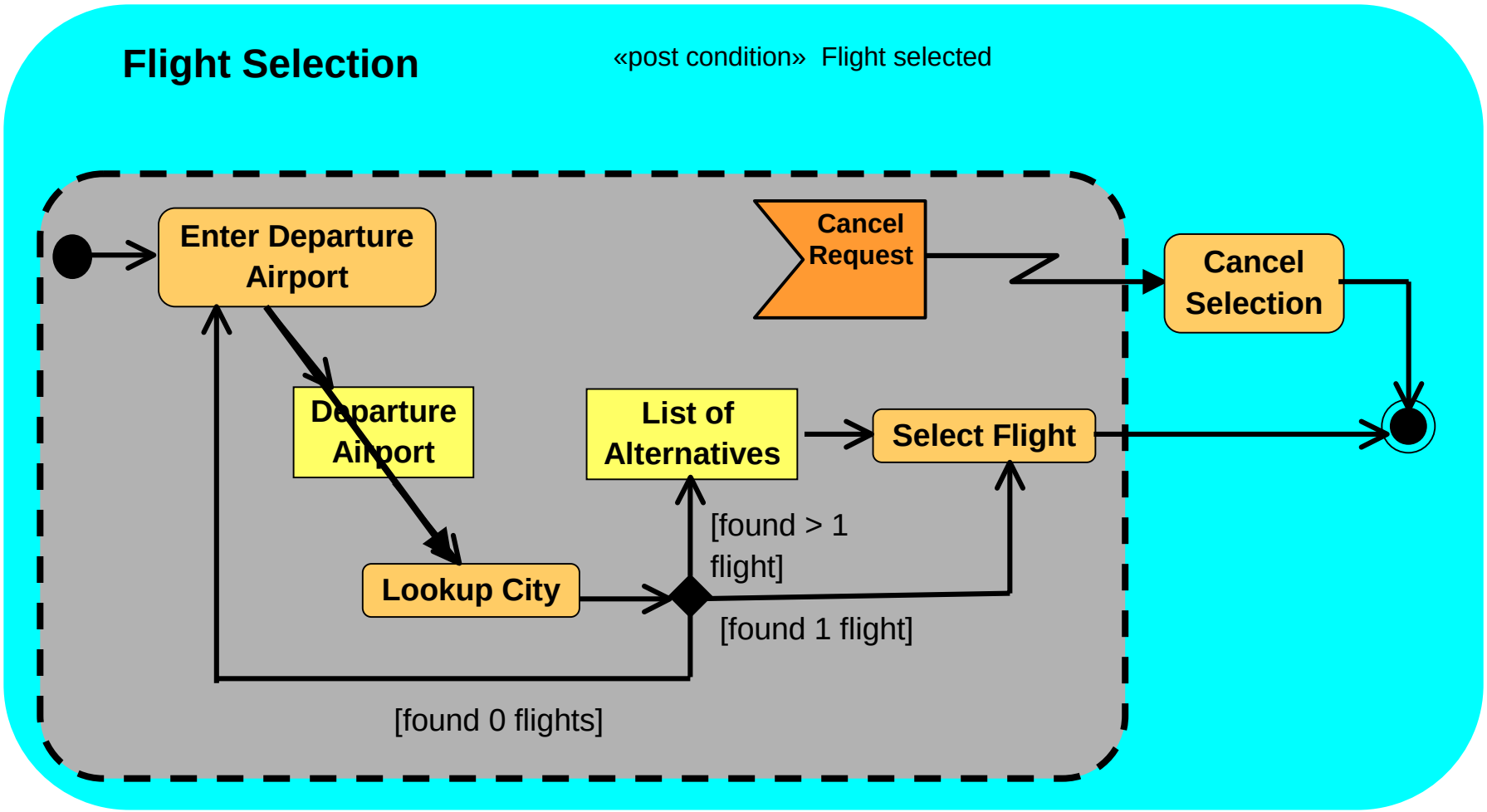
- Significantly enriched in UML 2.0 (relative to UML 1.x activities)
  - More flexible semantics for greater modeling power (e.g., rich concurrency model based on Petri Nets)
  - Many new features
- Major influences for UML 2.0 activity semantics
  - Business Process Execution Language for Web Services (BPEL4WS) – a de facto standard supported by key industry players (Microsoft, IBM, etc.)
  - Functional modeling from the systems engineering community (INCOSE)

Activity Diagram

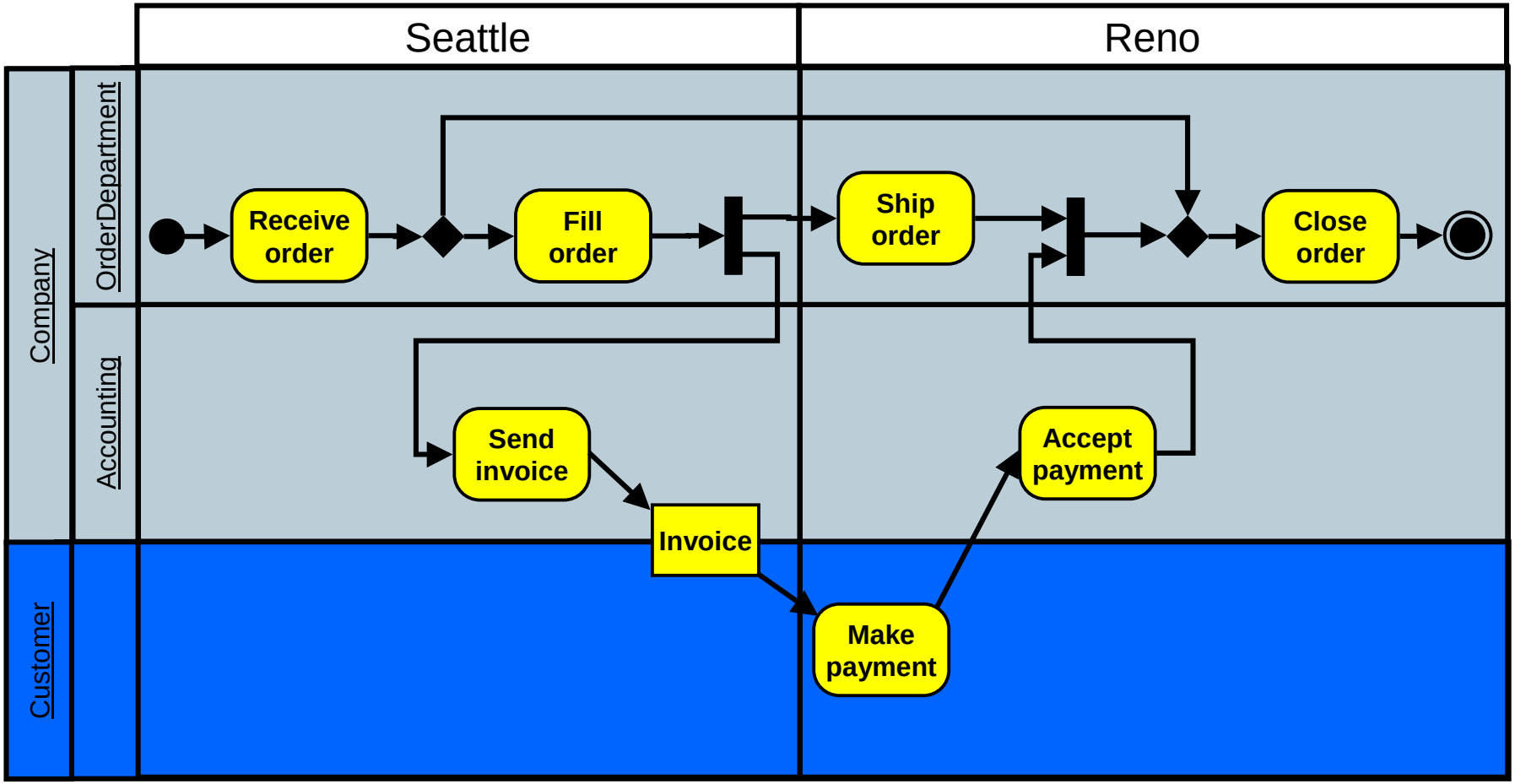
- **Activity diagrams** show flow of control and data flow
- Typically used to model
  - Business process workflow
  - Flow within a use case
  - Business rules logic
  - Functional processes
  - UI screen flows



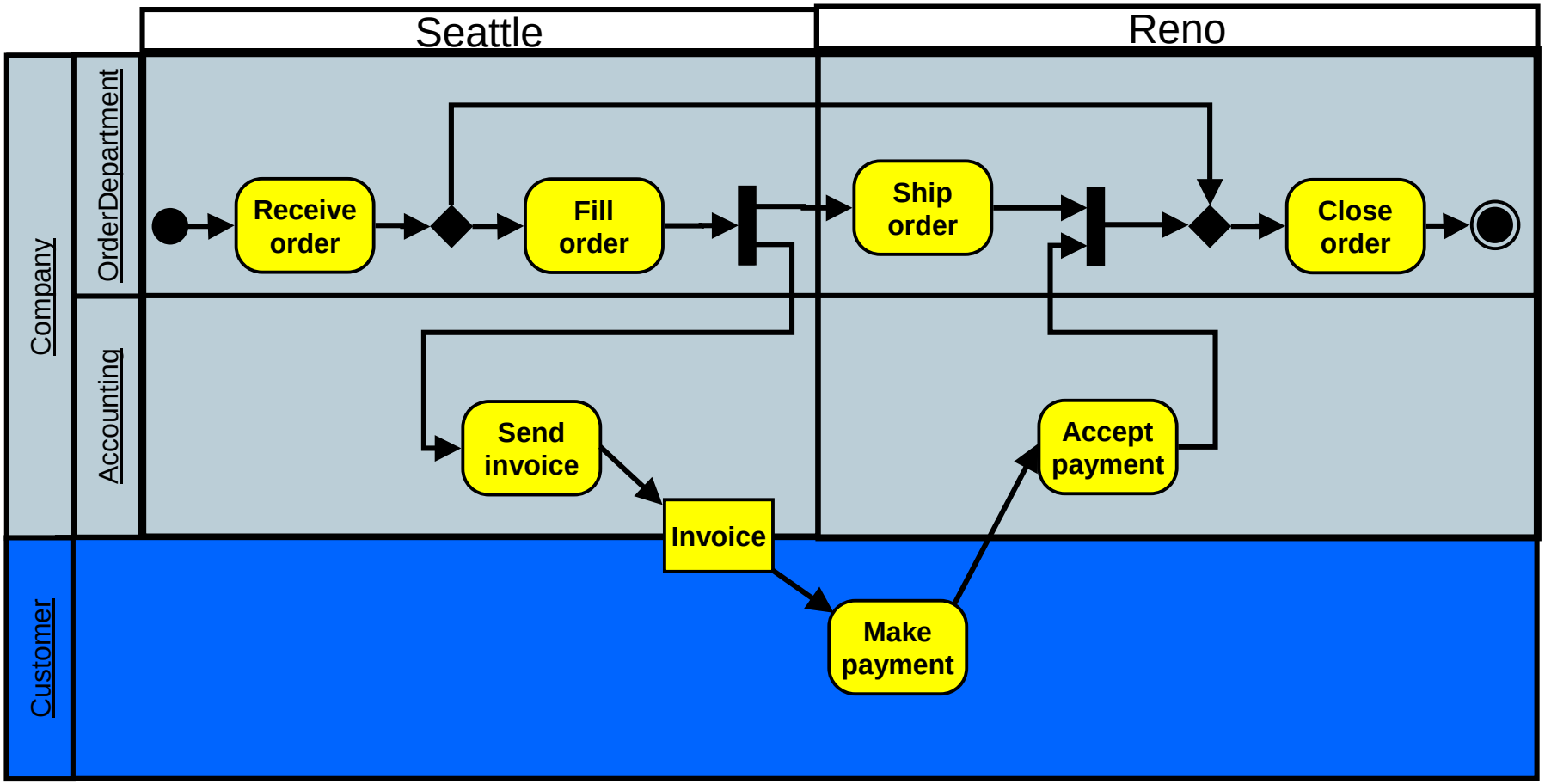
Activity Diagram



Partitioning capabilities



Partitioning capabilities



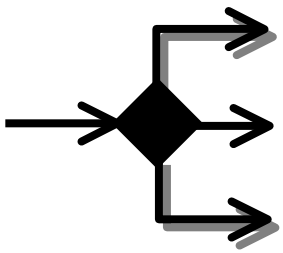
Activities: Basic Notational Elements

Control/Data Flow

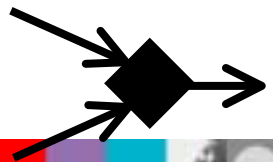
Activity or Action

Object Node
(may include state)

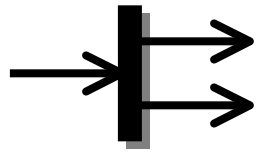
Pin (Object)



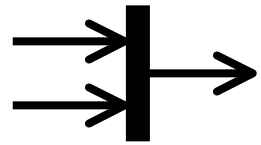
Choice



(Simple) Join



Control Fork



Control Join

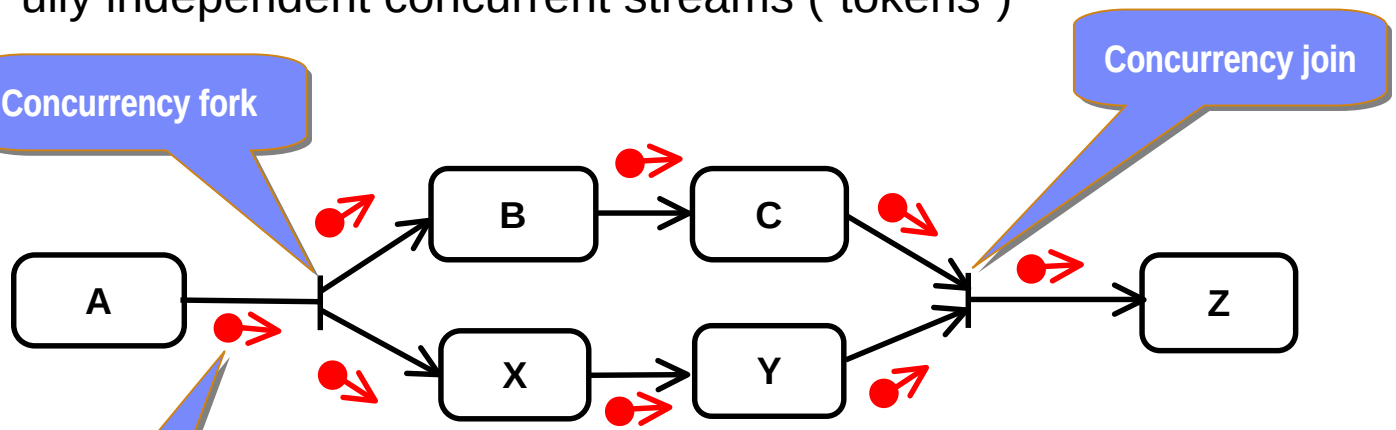
Initial Node

Activity Final

Flow Final

Extended Concurrency Model

- Fully independent concurrent streams (“tokens”)



“Tokens” represent individual execution threads (executions of activities)

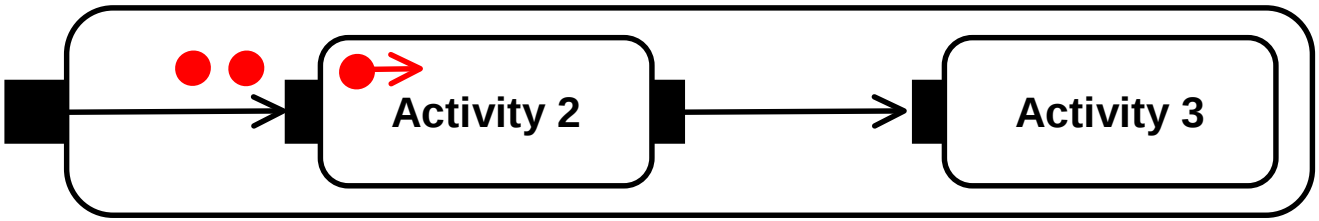
NB: Not part of the notation

Valid output streams:

ABCXYZ
AXYBCZ
ABXCYZ
AXBYCZ
ABXYCZ
AXBCYZ

Activities: Token Queuing Capabilities

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



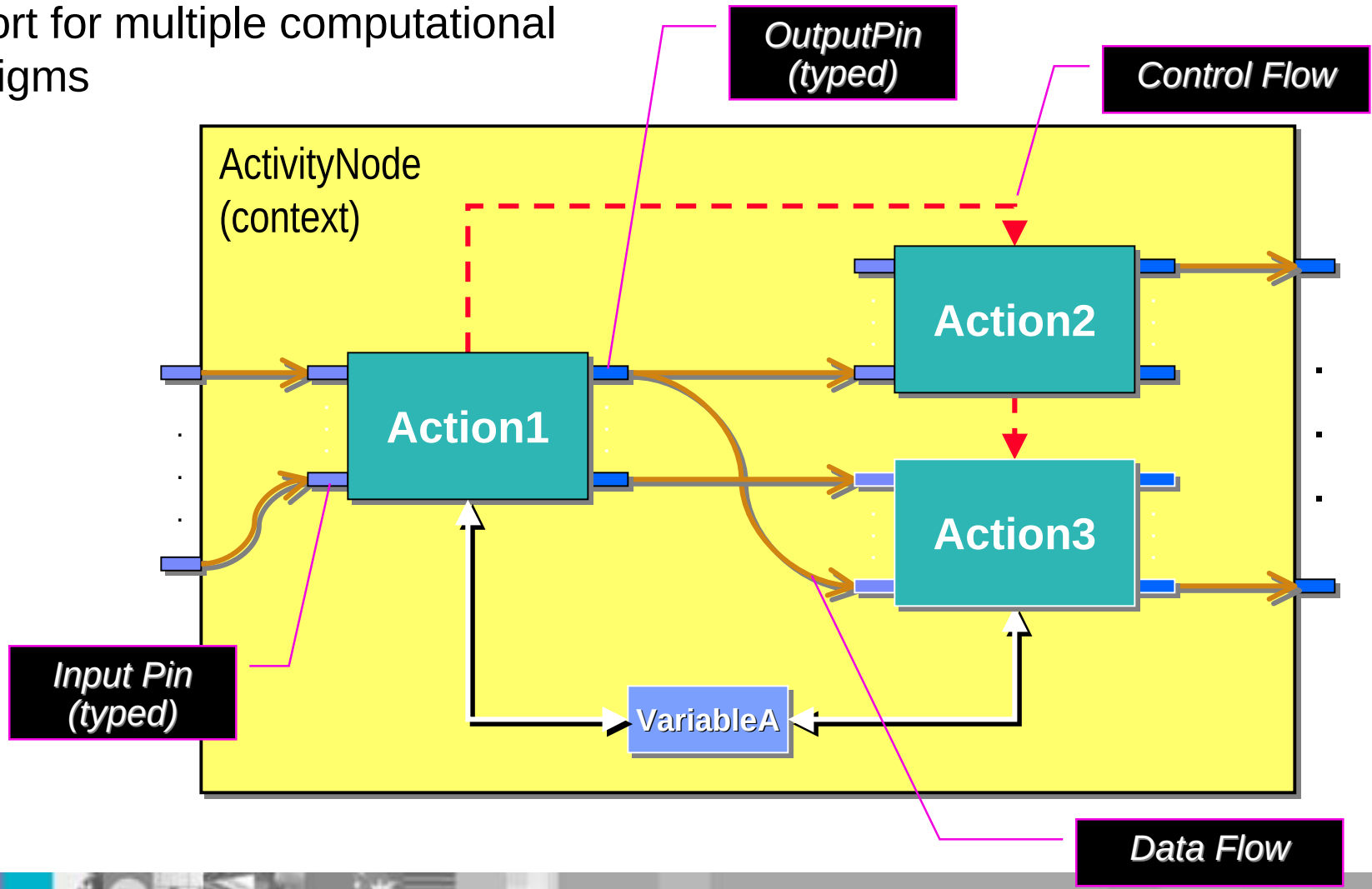
- …or, they can flow through continuously
  - taken as input while behavior is executing
  - given as output while behavior is executing
  - identified by a {stream} adornment on a pin or object node

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 semantics for those actions
    - i.e. what happens when the actions are executed
    - Pre- and post-condition specifications (using OCL)
  - No concrete syntax for individual kinds of actions (notation)
    - Flexibility: can be realized using different concrete languages
- In UML 2, the metamodel of actions was consolidated
  - Shared semantics between actions and activities (Basic Actions)

Object Behavior Basics

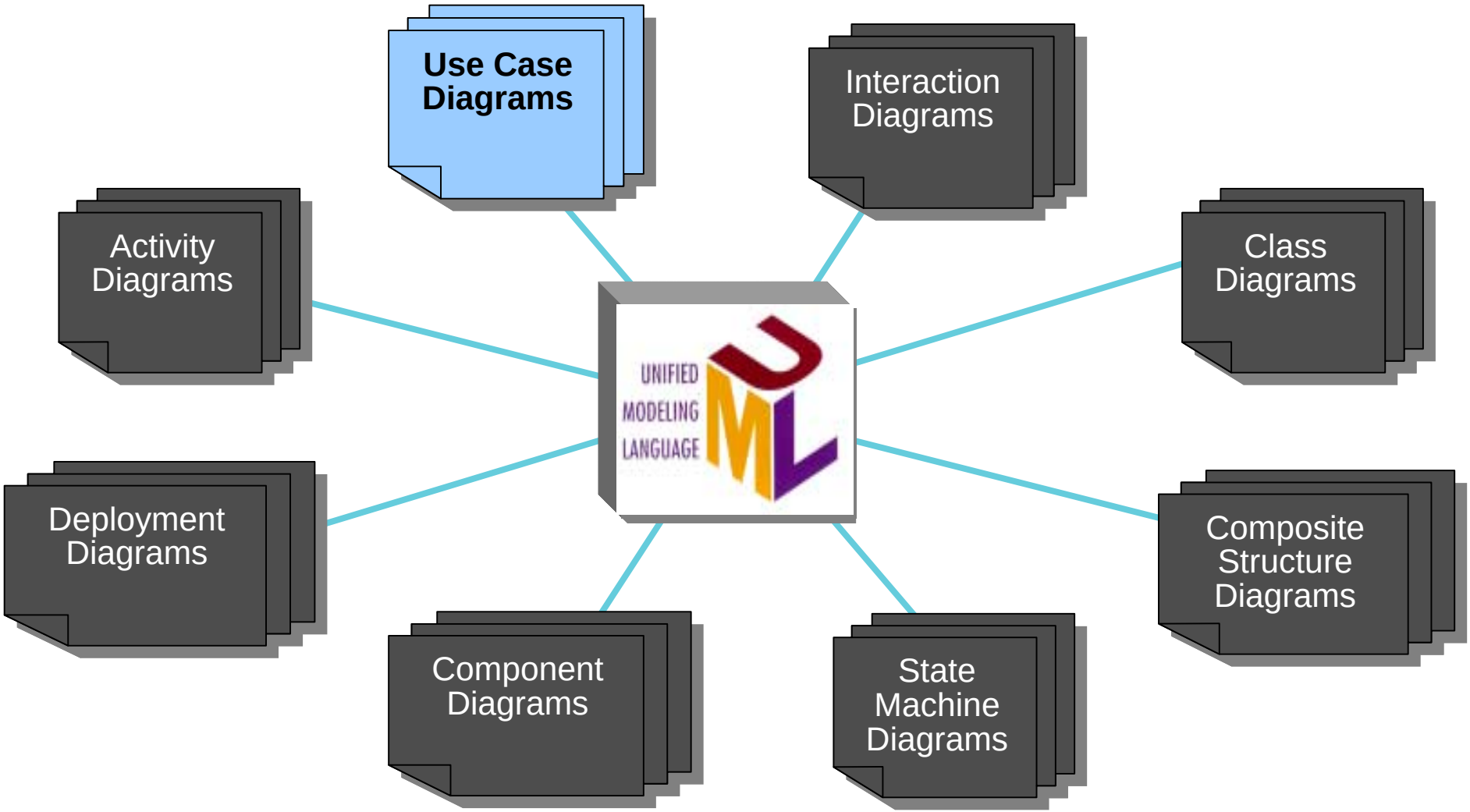
- Support for multiple computational paradigms



Categories of Actions

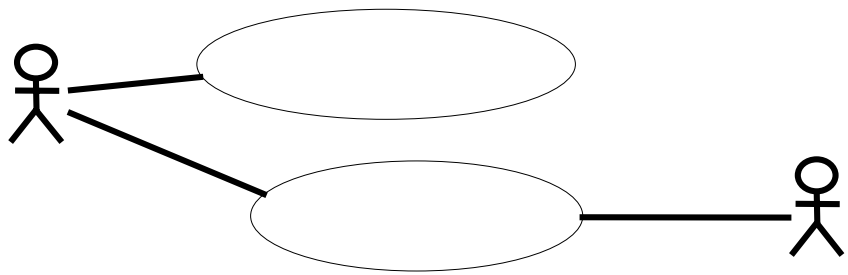
- Communication actions (send, call, receive,…)
- Primitive function action
- Object actions (create, destroy, reclassify,start,…)
- Structural feature actions (read, write, clear,…)
- Link actions (create, destroy, read, write,…)
- Variable actions (read, write, clear,…)
- Exception action (raise)

UML 2.0 Diagrams



Use Case Diagram

- **Use case diagrams** are created to visualize the relationships between actors and use cases
- Use cases are a visualization the functional requirements of a system



Actors

- An **actor** is someone or some thing that must interact with the system under development

Passenger

Bank

Use Cases

- A **use case** is a pattern of behavior the system exhibits
  - Each use case is a sequence of related transactions performed by an actor and the system in a dialogue
- Actors are examined to determine their needs
  - Passenger – Search for Flights, Make Reservation, Pay for Flight
  - Bank -- receive payment information from Pay for Flight

Search for Flights

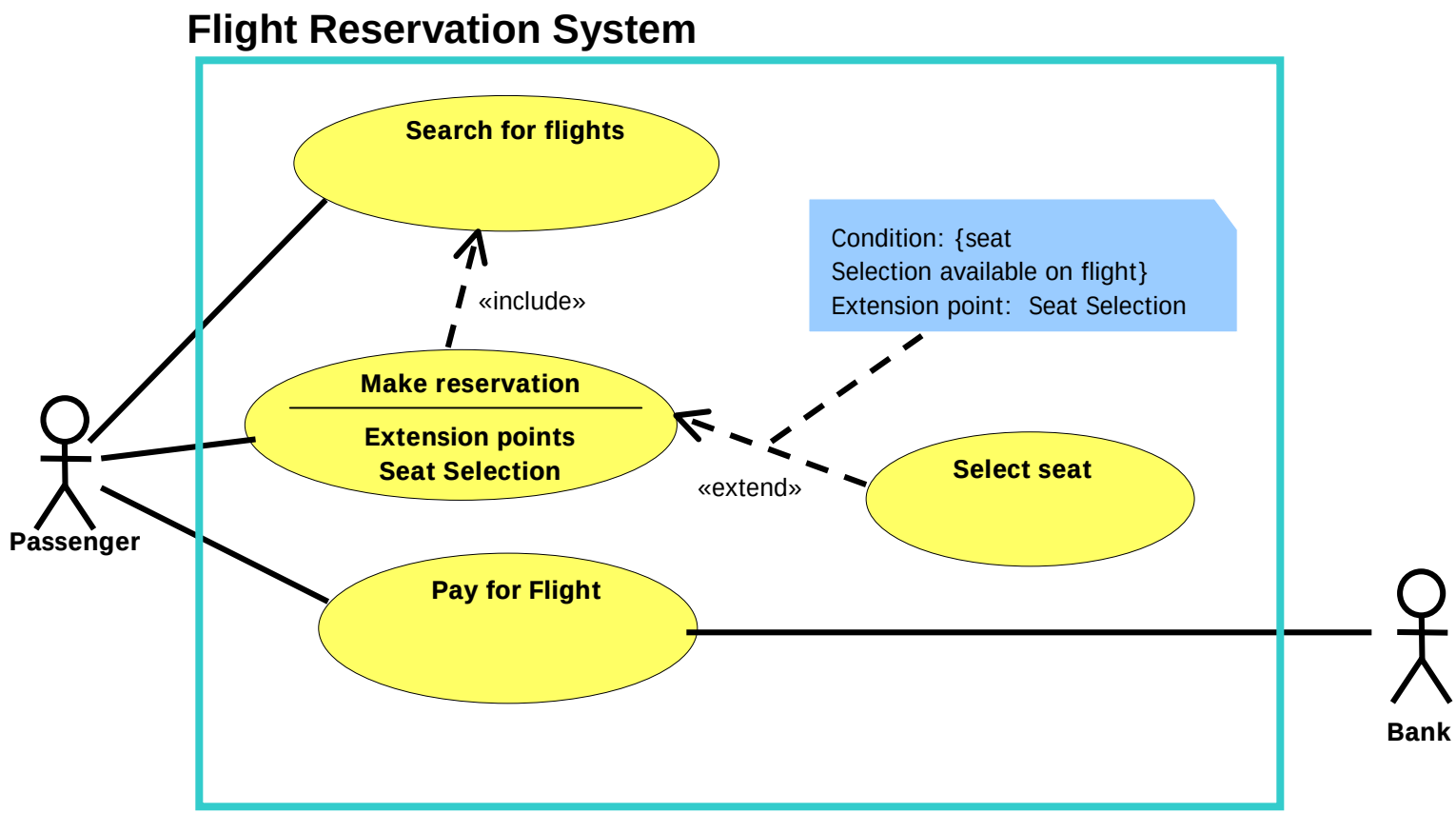
Make Reservation

Pay for Flight

Documenting Use Cases

- A **use case specification** document is created for each use cases
  - Written from an actor point of view
- Details what the system must provide to the actor when the use cases is executed
- Typical contents
  - How the use case starts and ends
  - Normal flow of events
  - Alternate flow of events
  - Exceptional flow of events

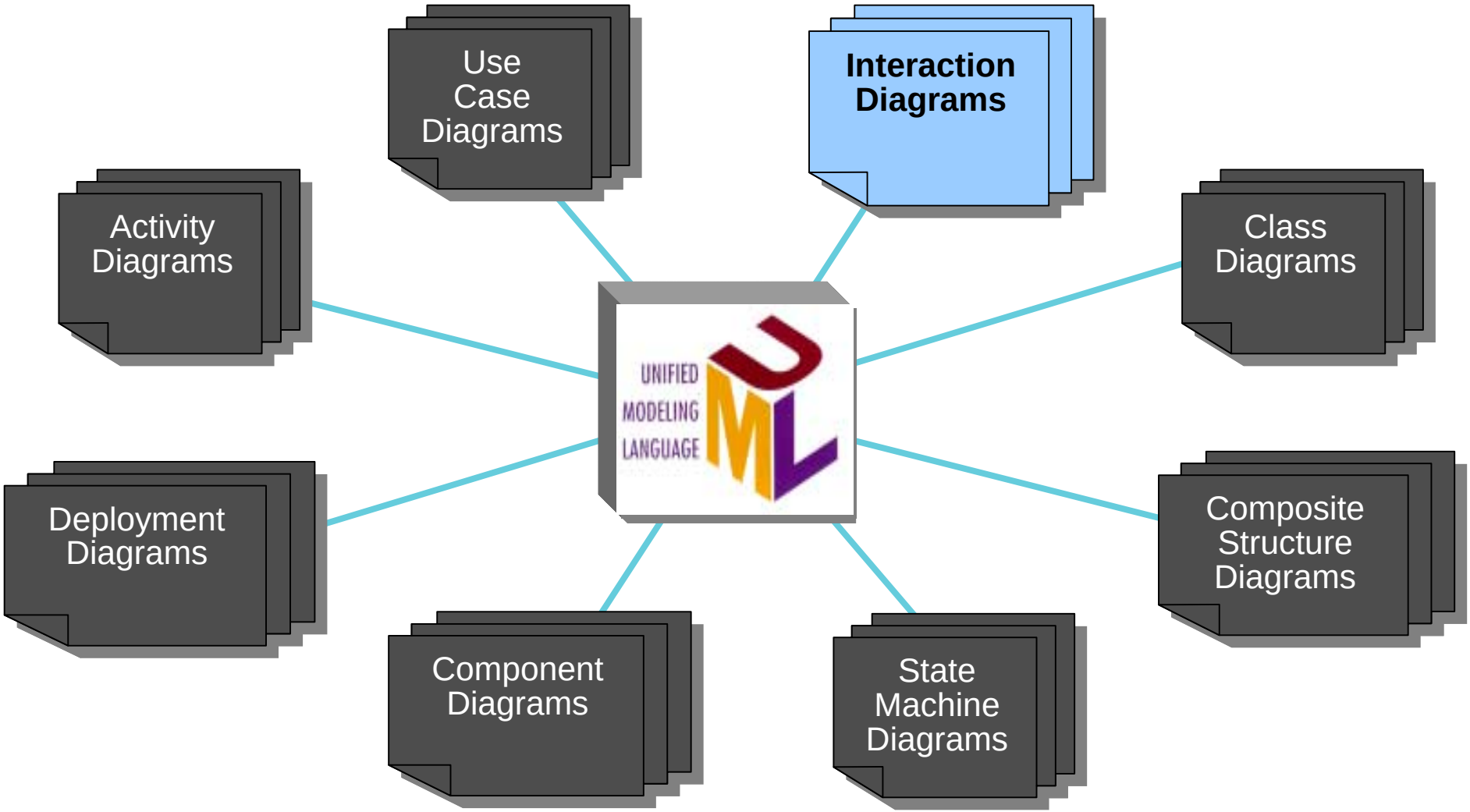
Use Case Diagram



UML 2.0 Changes

- No notational changes for use case diagrams in UML 2.0

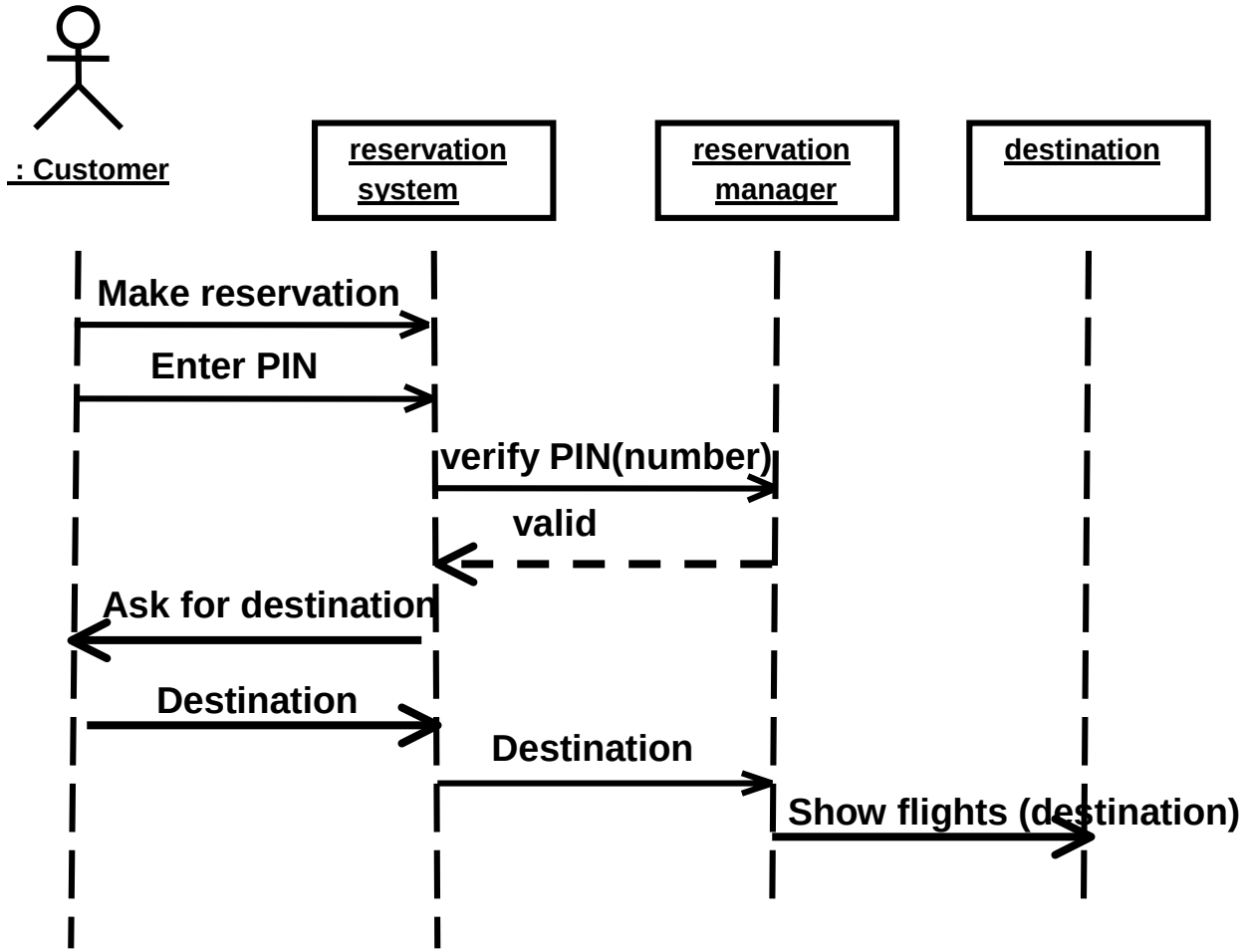
UML 2.0 Diagrams



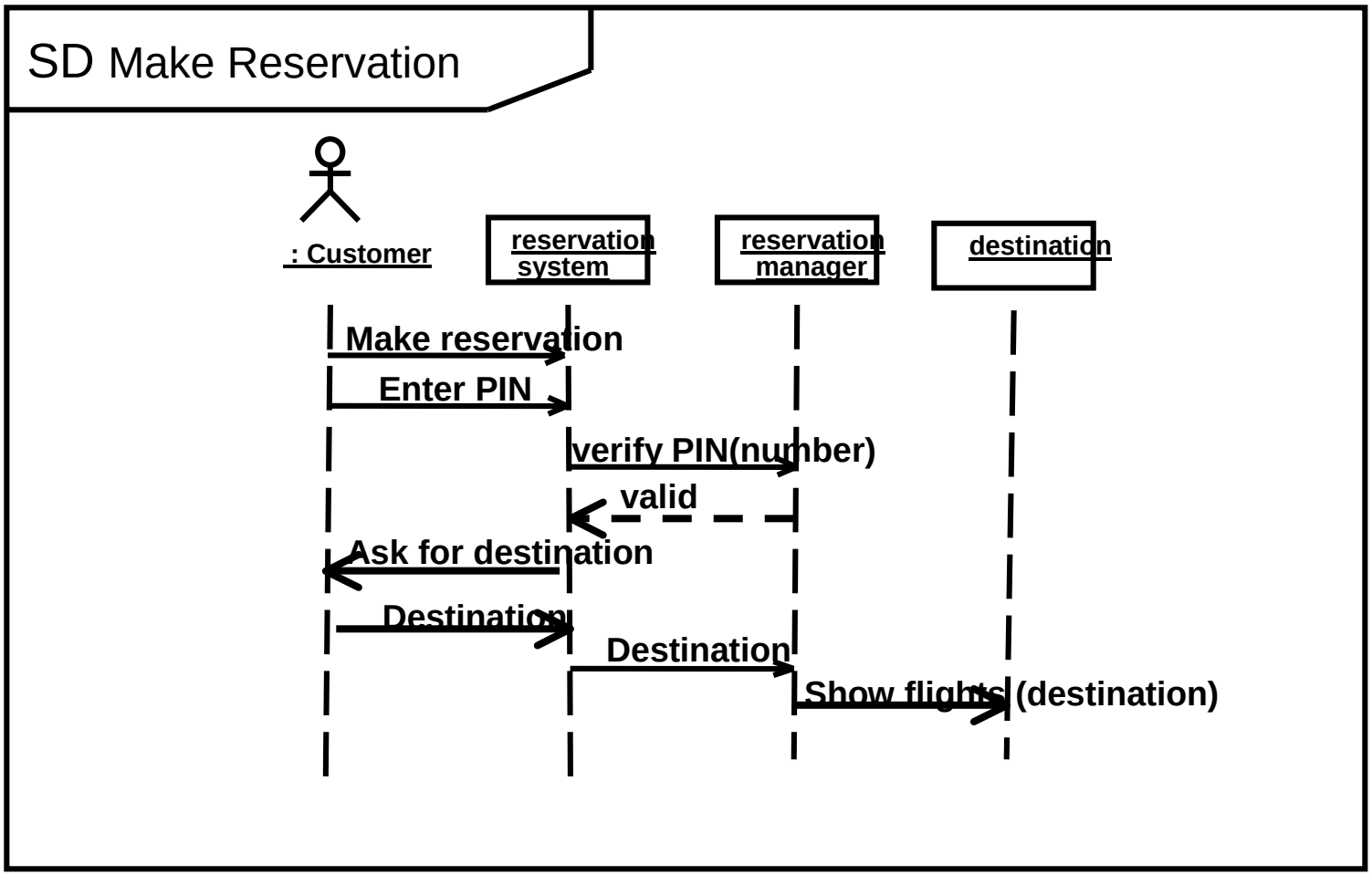
Interaction Diagrams

- **Interaction diagrams** show the communication behavior between parts of the system
- Four types of diagrams
  - Sequence diagram
    - Emphasis on the sequence of communications between parts
  - Communication diagram
    - Emphasis on structure and the communication paths between parts
  - Timing diagram
    - Emphasis on change in state over time
  - Interaction overview diagram
    - Emphasis on flow of control between interactions

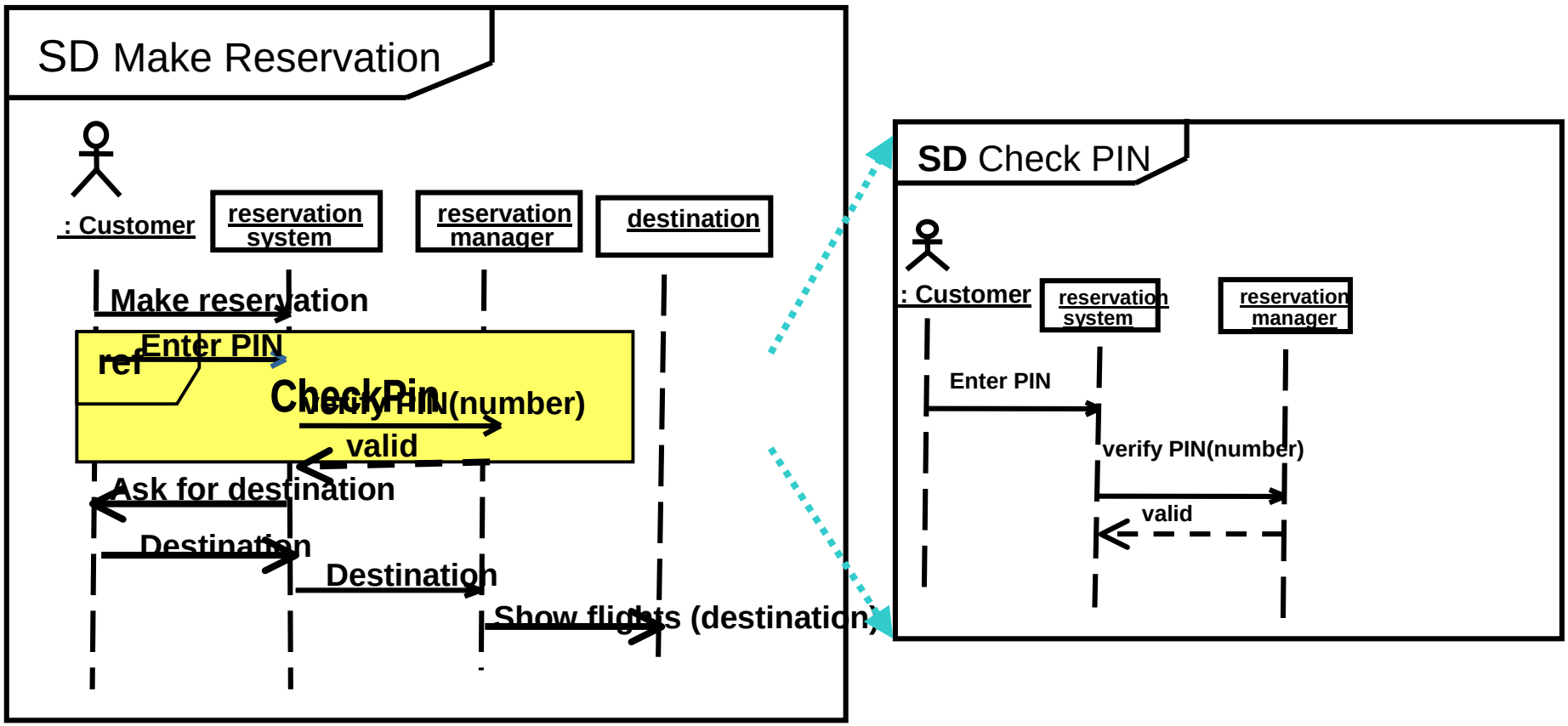
Sequence Diagram



Framed Sequence Diagram



Composite Diagrams



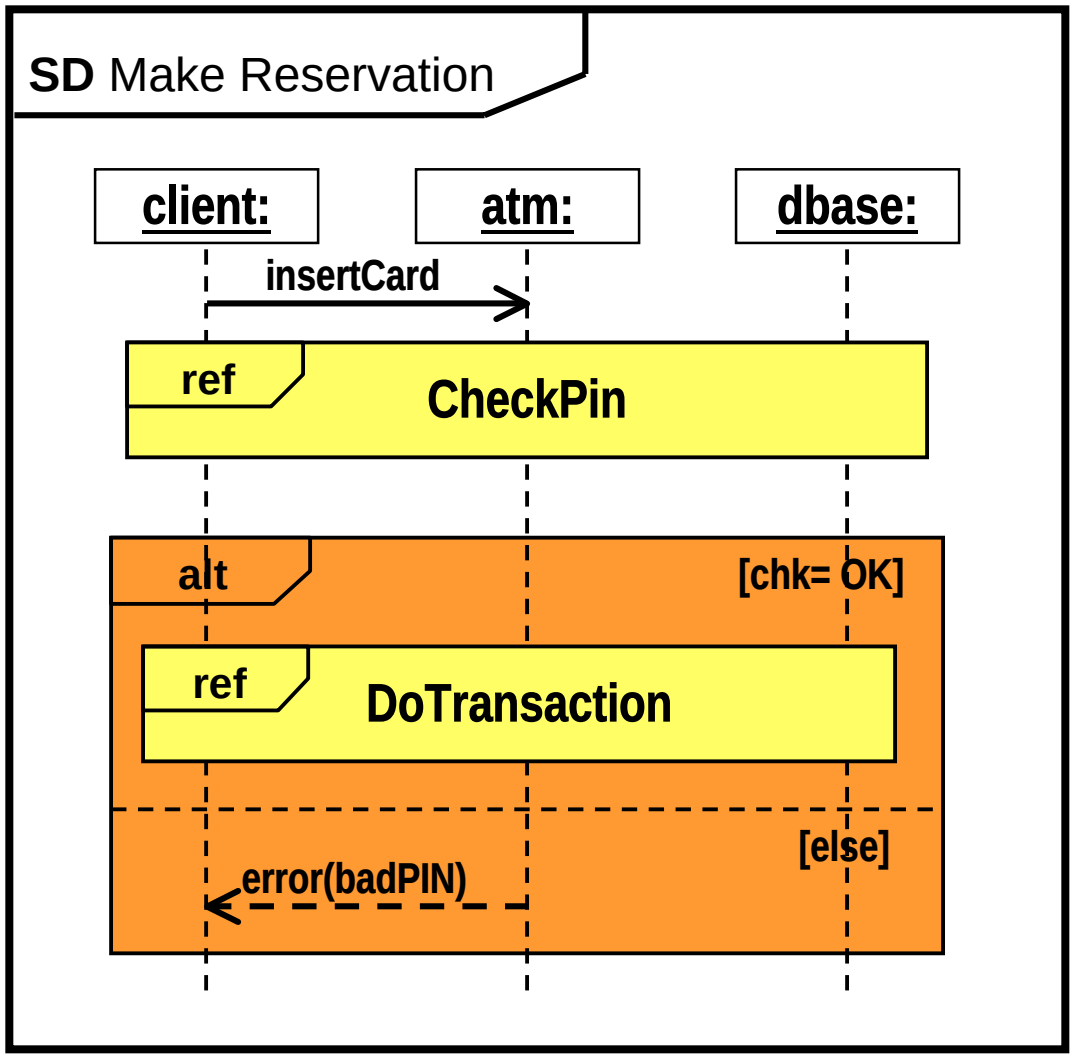
Combined Fragment Types

- 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
- Loop (loop)
  - Optional guard: [<min>, <max>, <Boolean-expression>]
  - No guard means no specified limit
- Break (break)
  - Represents an alternative that is executed instead of the remainder of the fragment (like a break in a loop)

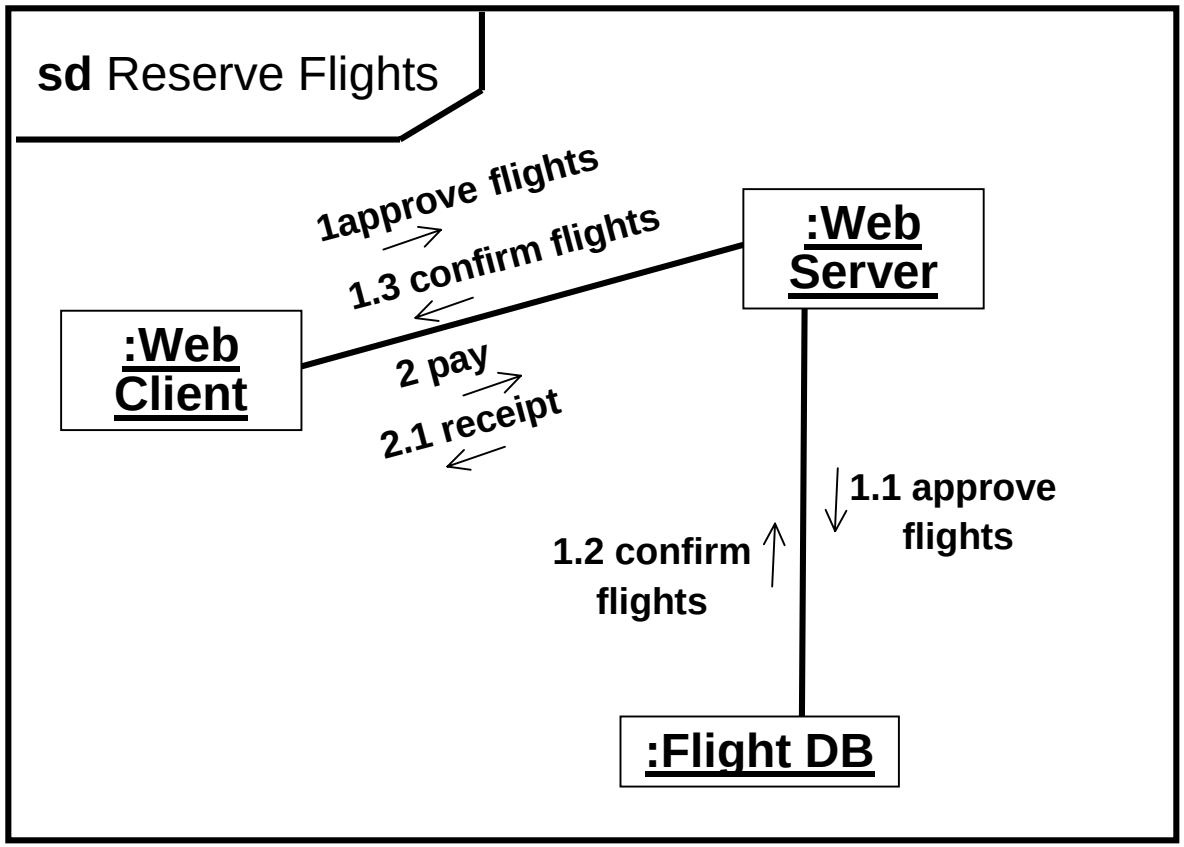
Combined Fragment Types

- Parallel (par)
  - Concurrent (interleaved) sub-scenarios
- Negative (neg)
  - Identifies sequences that must not occur
- Assertion (assert)
  - This must happen
- Critical Region (region)
  - Traces cannot be interleaved with events on any of the participating lifelines

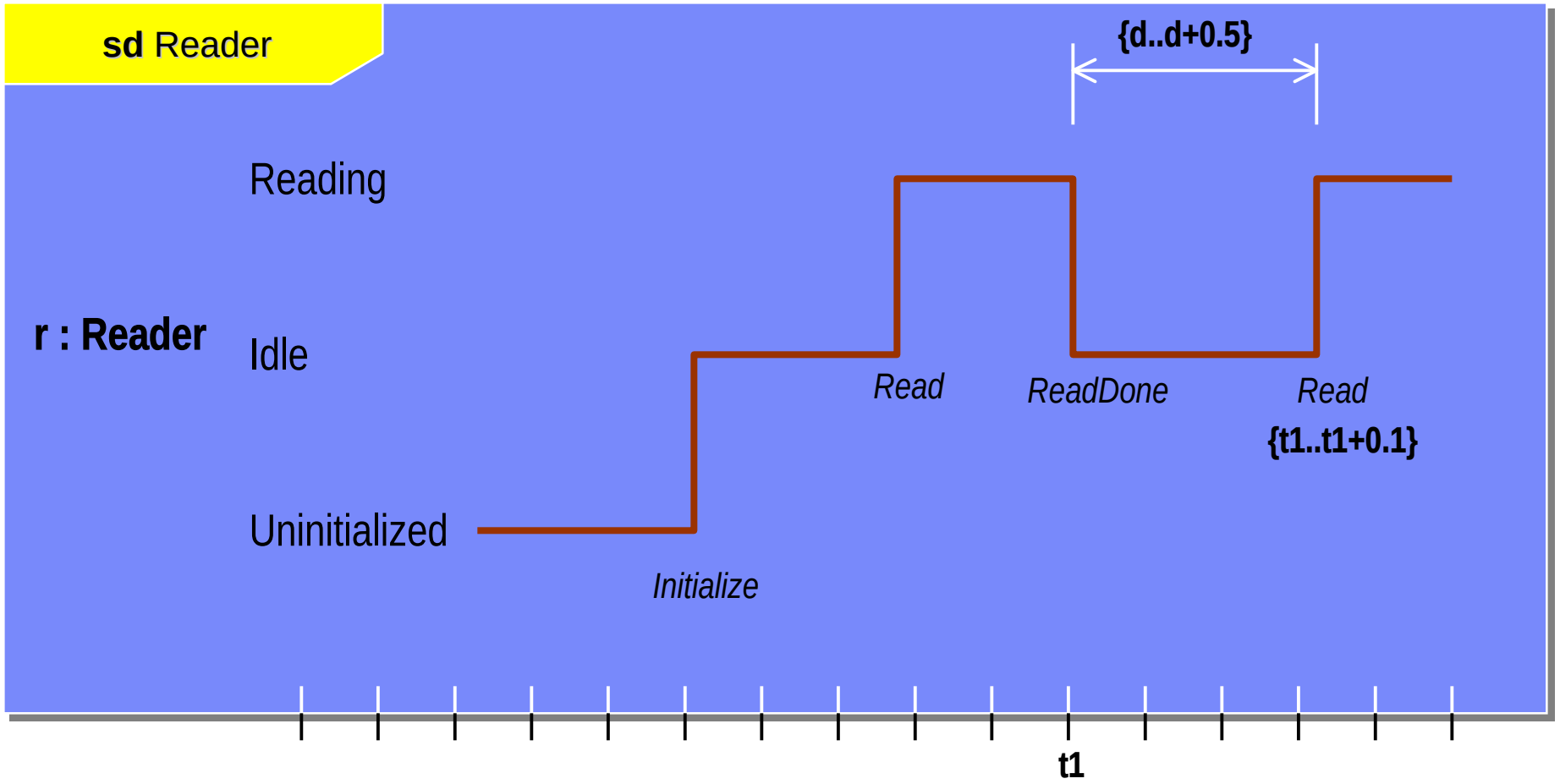
Combined Fragments Diagram



Communication Diagram Example

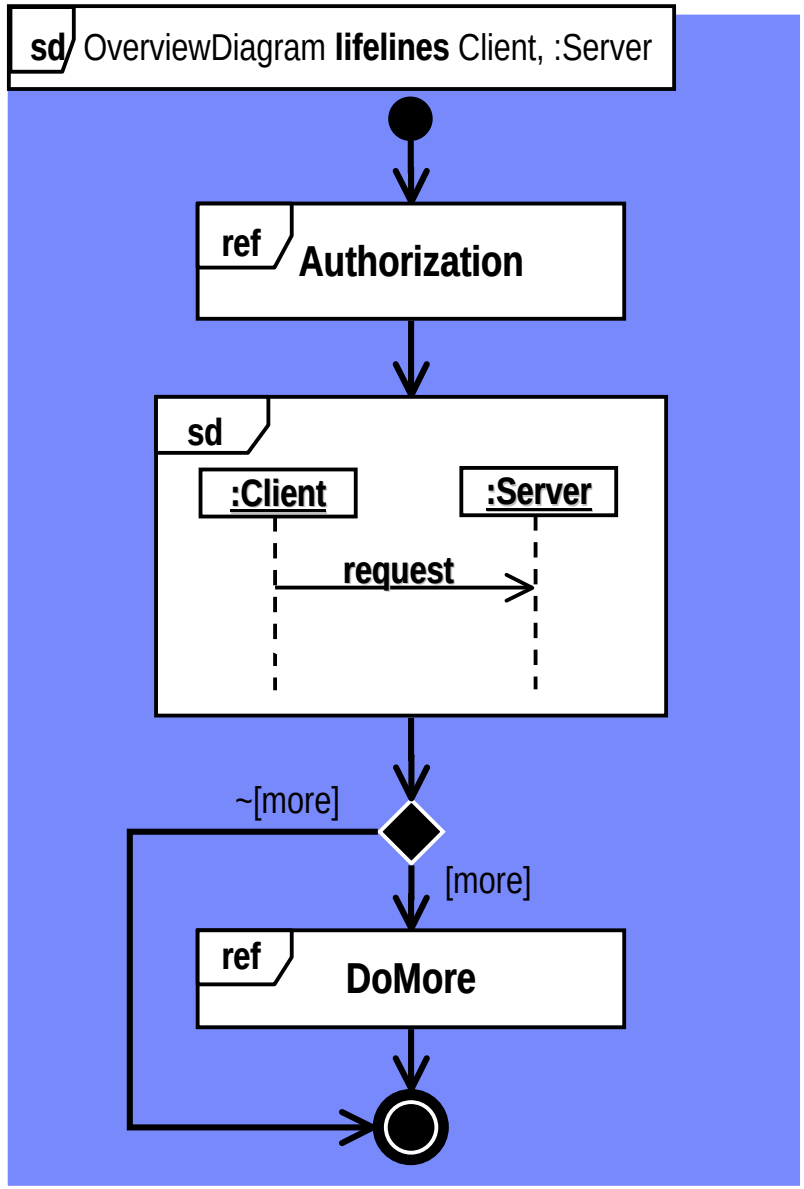


Timing Diagram Example



Interaction Overview Example

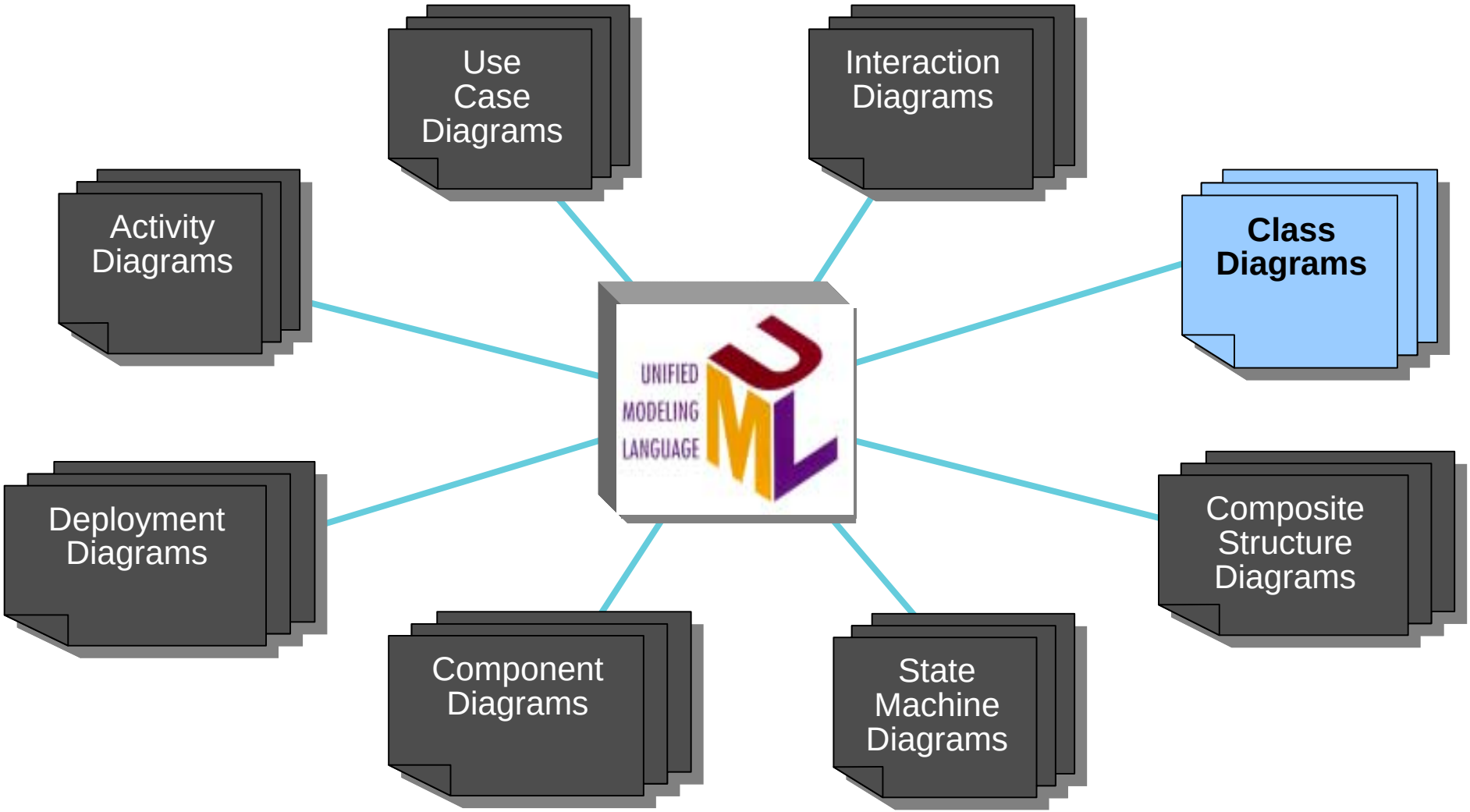
- Like flow charts
  - Use activity graph notation for control constructs
- Better overview of complex interactions
  - Alternatives, options etc.
- Multiple diagram types could be included/referenced



UML 2.0 Changes

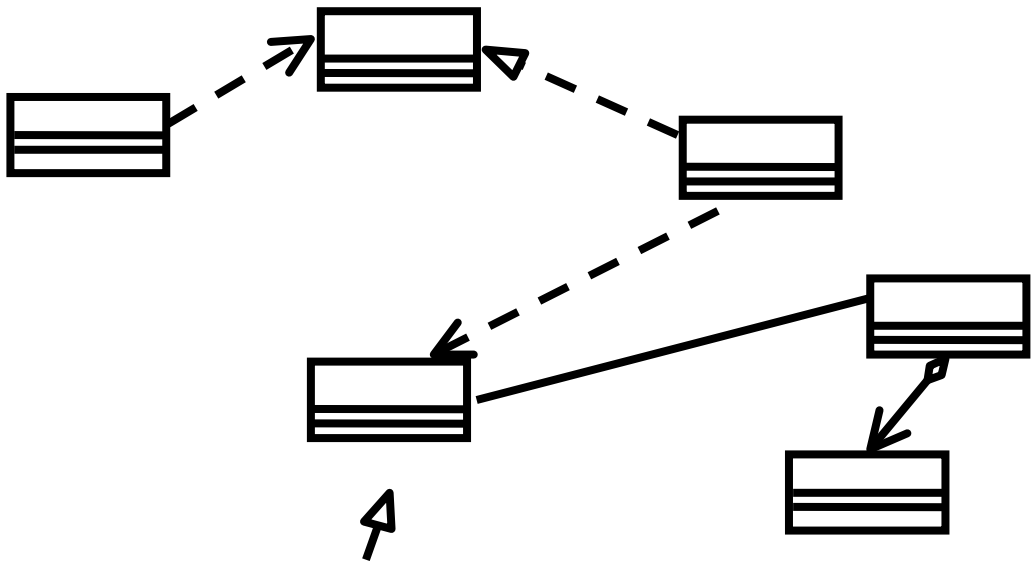
- Interaction occurrences and combined fragments added
- Communication diagrams created to replace collaboration diagrams used to show interactions
- New diagrams: Timing Diagram and Interaction Overview Diagram

UML 2.0 Diagrams



Class Diagram

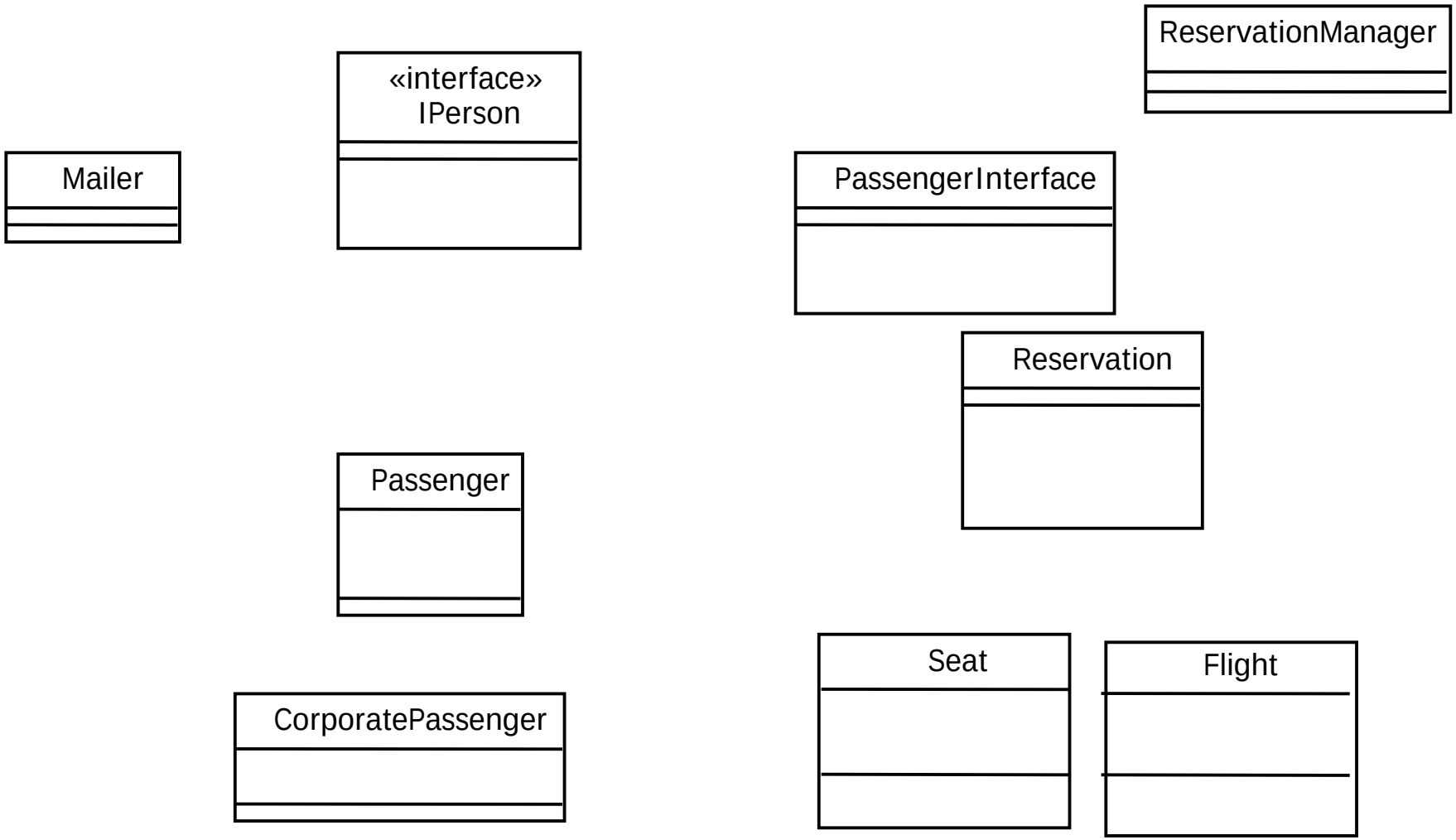
- **Class diagrams** show static structure
  - This is the diagram that is used to generate code



Classes

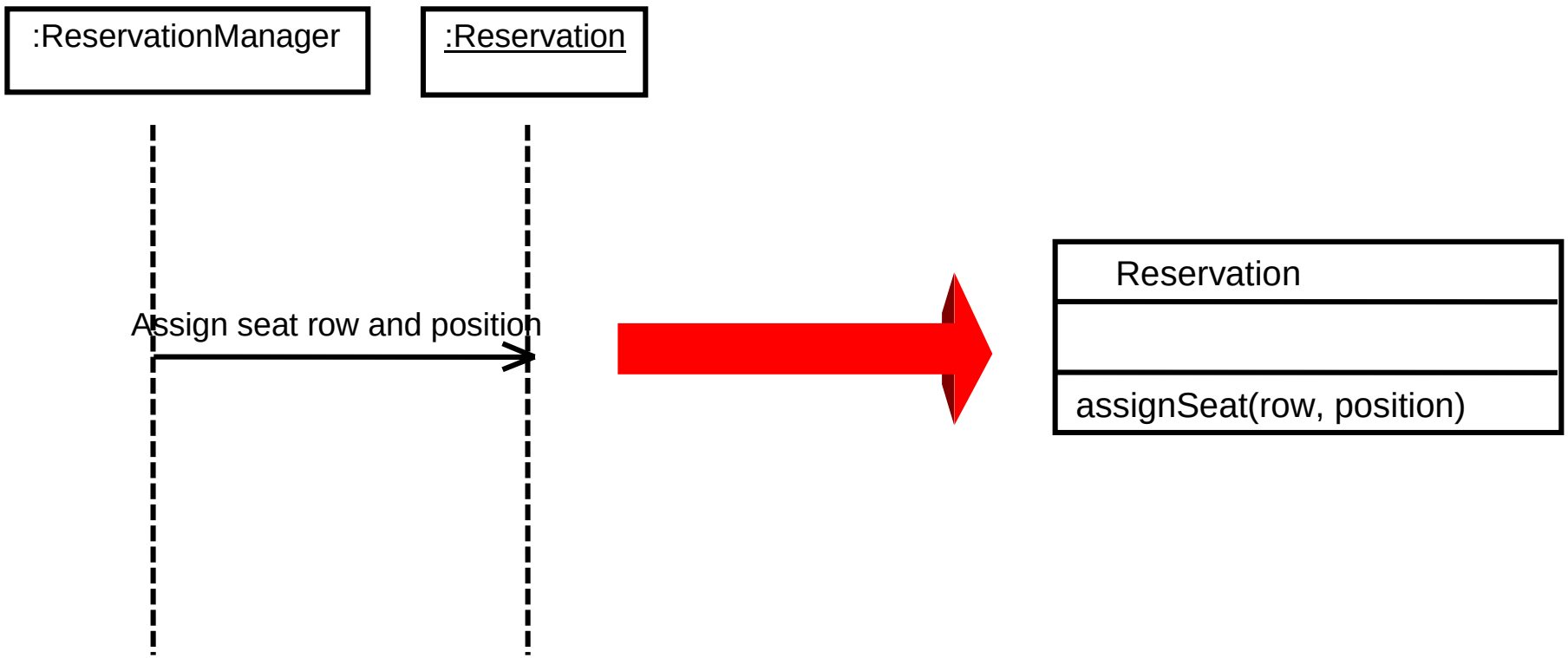
- A **class** defines a set of objects with common structure, common behavior, common relationships and common semantics
- Classes can be “discovered” by examining the objects in sequence and collaboration diagram
- A class is drawn as a rectangle with three compartments
- Classes should be named using the vocabulary of the domain
  - Naming standards should be created
  - e.g., all classes are singular nouns starting with a capital letter

Classes


ReservationManager
«interface»
IPerson
Mailer
PassengerInterface
Reservation
Passenger
Seat
Flight
CorporatePassenger

Operations

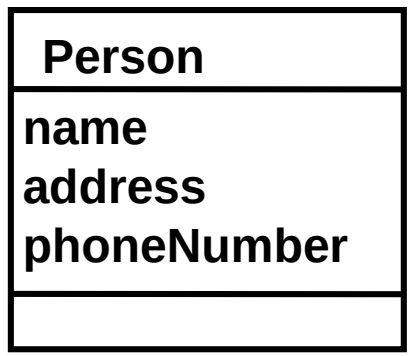
- The behavior of a class is represented by its **operations**
- Operations may be found by examining interaction diagrams



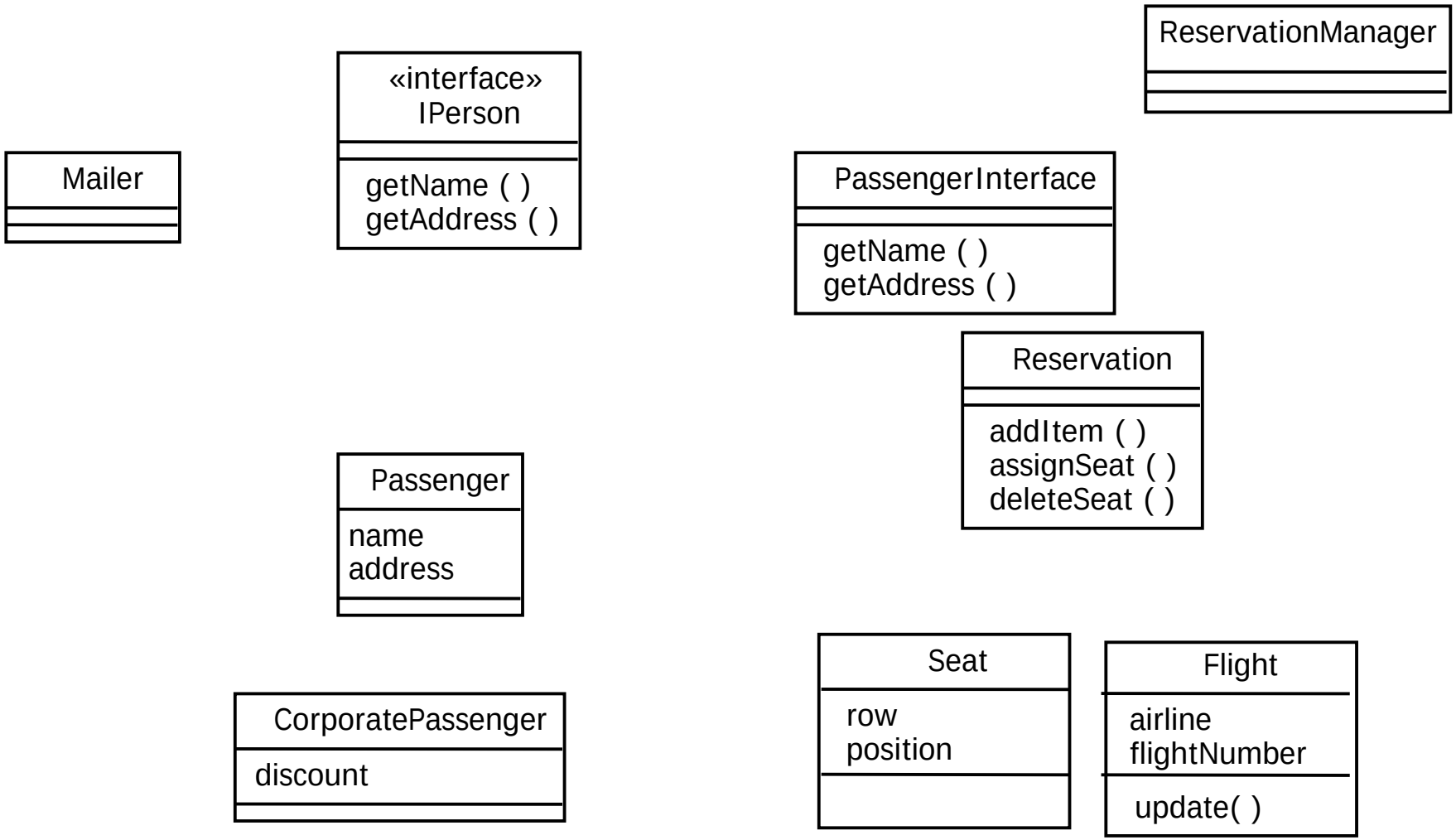
Attributes

- The structure of a class is represented by its **attributes**
- Attributes may be found by examining class definitions, the problem requirements, business rules and by applying domain knowledge

The name, address and phone number for each person is needed before a reservation can be made



Classes with Operations and Attributes



Relationships

- Relationships provide a pathway for communication between objects
- Sequence and/or communication diagrams are examined to determine what links between objects need to exist to accomplish the behavior -- if two objects need to “talk” there must be a link between them
- Relationship types
  - Association
  - Aggregation
  - Composition
  - Dependency

Relationships

- An **association** is a bi-directional connection between classes

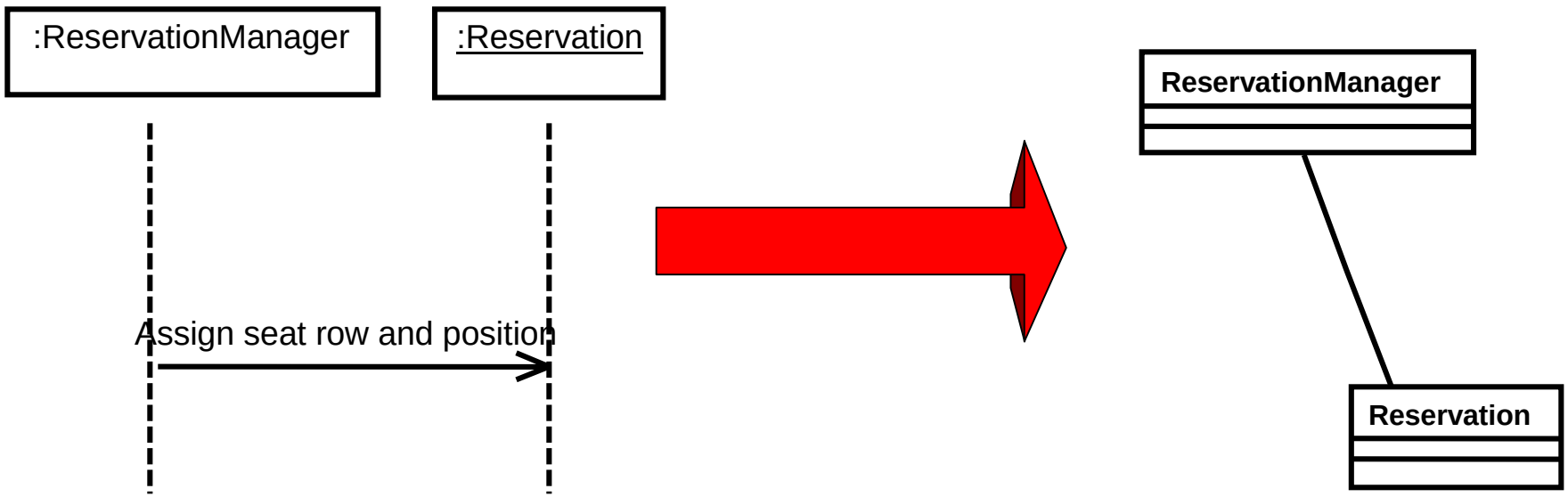
- An **aggregation** is a stronger form of association where the relationship is between a whole and its parts

- A **composition** is a stronger form of aggregation where the part is contained in at most one whole and the whole is responsible for the creation of its parts

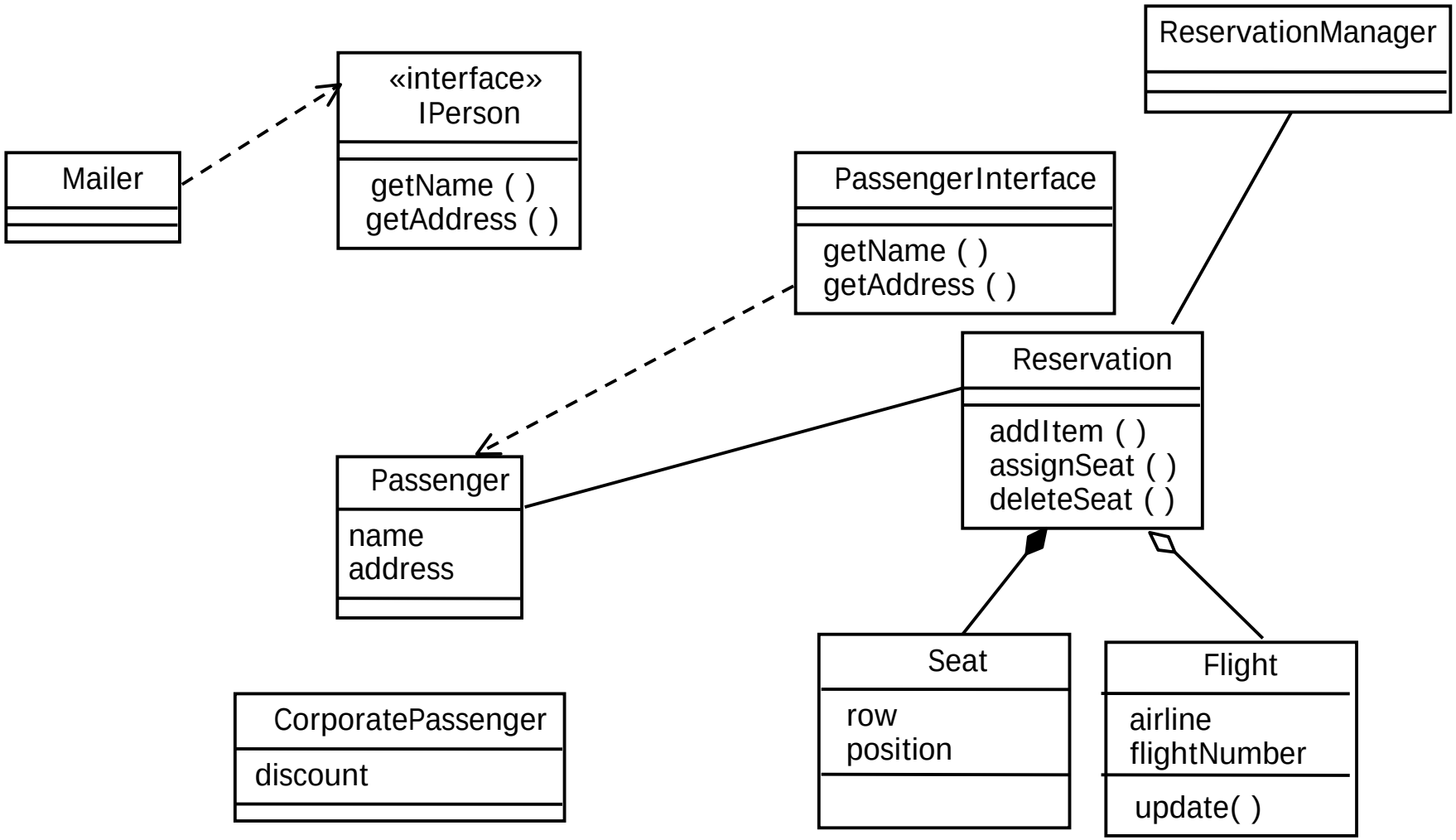
- A **dependency** is a weaker form of relationship showing a relationship between a client and a supplier where the client does not have semantic knowledge of the supplier

Finding Relationships

- Relationships are discovered by examining interaction diagrams
  - If two objects must “talk” there must be a pathway for communication



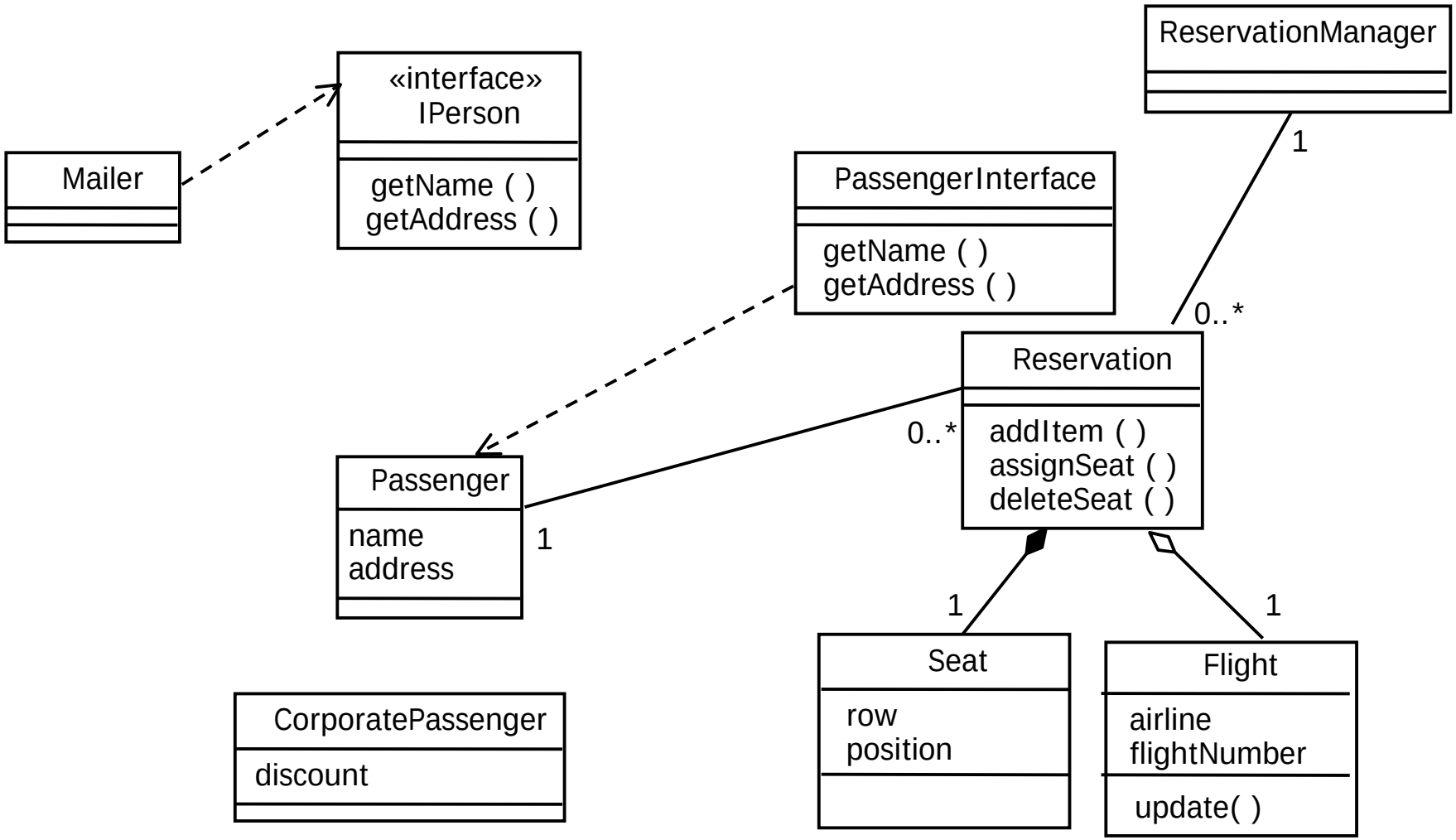
Relationships



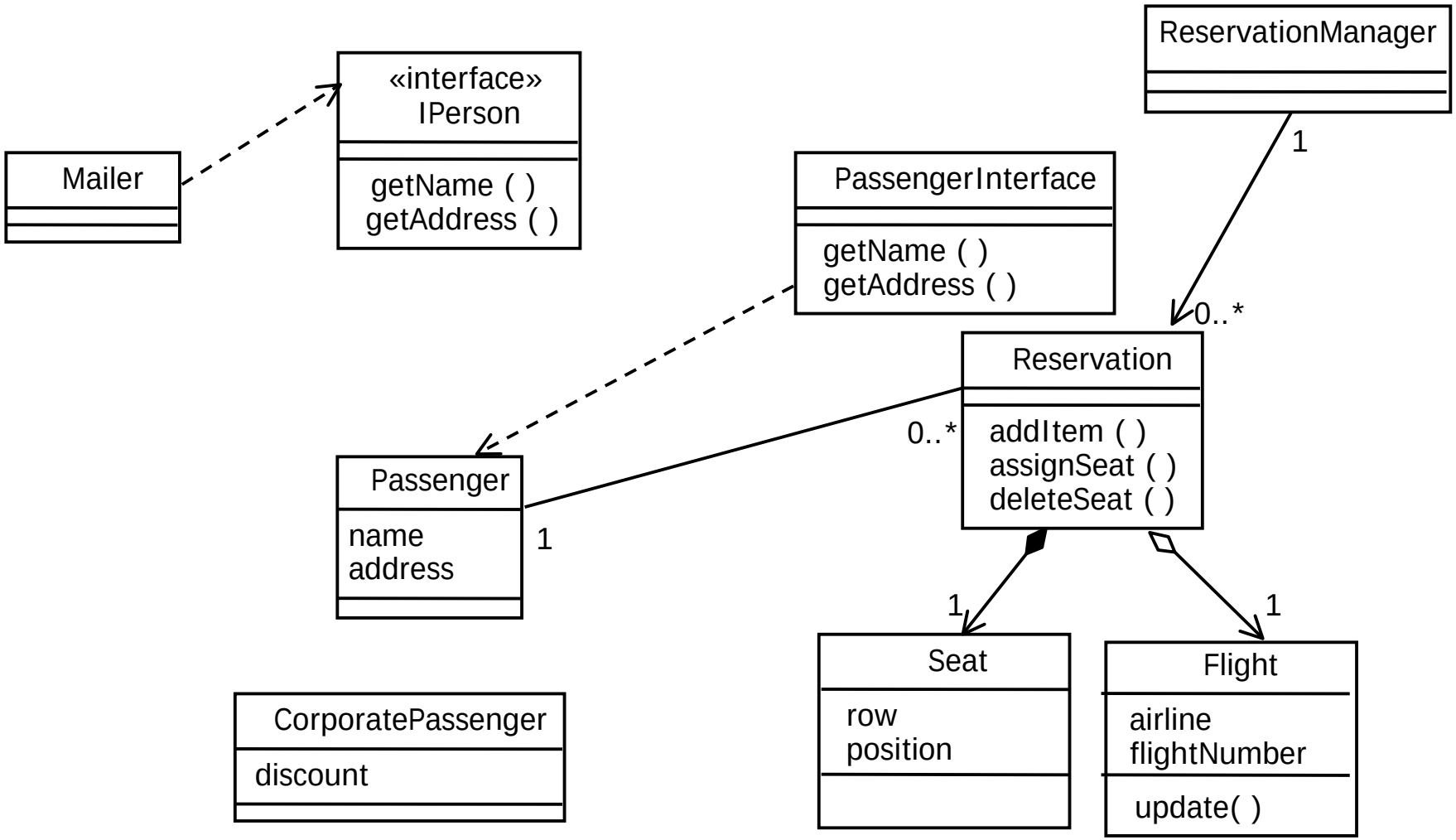
Multiplicity and Navigation

- **Multiplicity** defines how many objects participate in a relationships
  - Multiplicity is the number of instances of one class related to ONE instance of the other class
  - For each association and aggregation, there are two multiplicity decisions to make: one for each end of the relationship
- Although associations and aggregations are bi-directional by default, it is often desirable to restrict **navigation** to one direction
  - If navigation is restricted, an arrowhead is added to indicate the direction of the navigation

Multiplicity



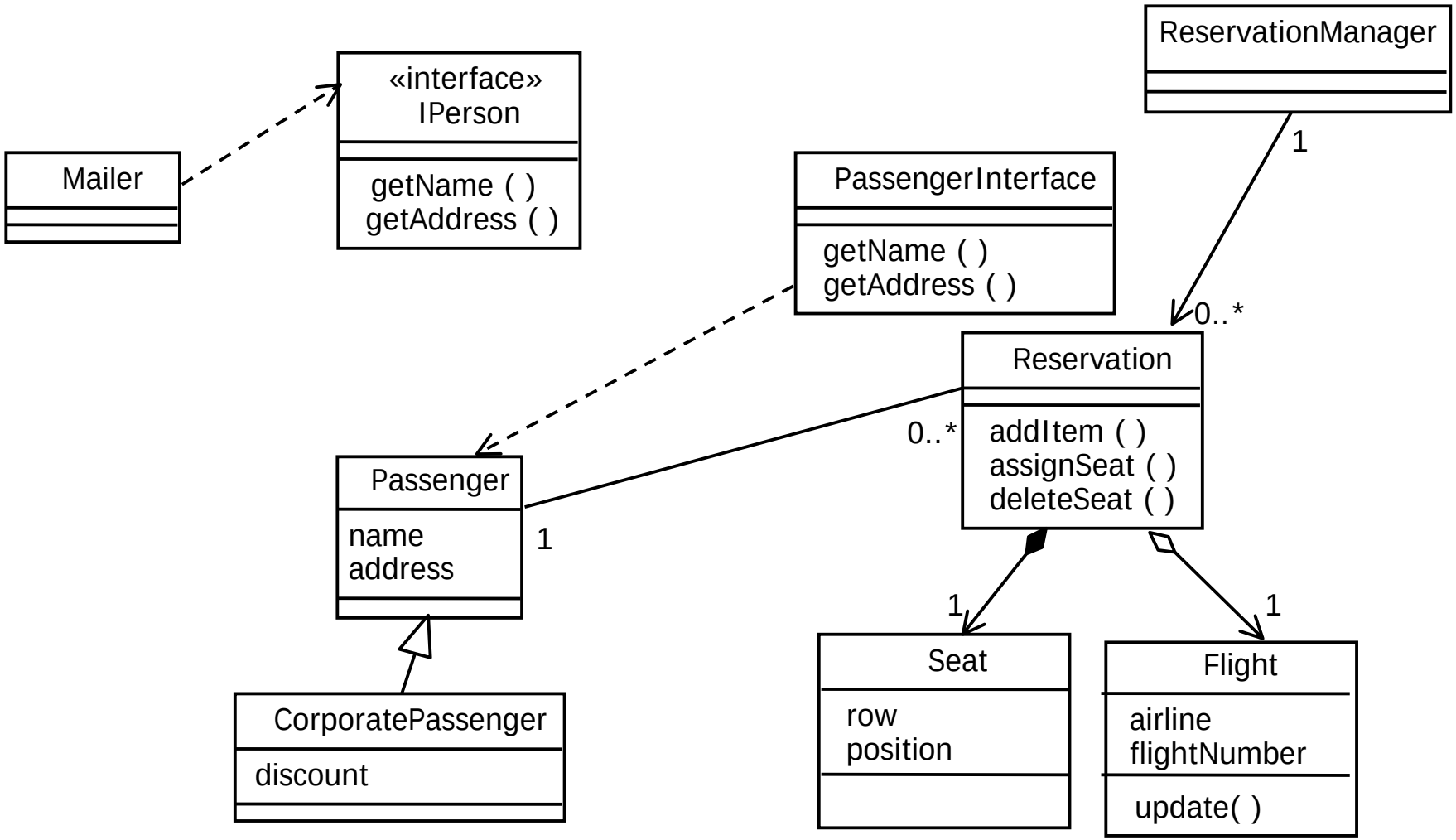
Navigation


ReservationManager
«interface»
IPerson
getName ( )
getAddress ( )
Mailer
PassengerInterface
getName ( )
getAddress ( )
1
0..*
Reservation
addItem ( )
assignSeat ( )
deleteSeat ( )
0..*
Passenger
name
address
1
1
1
Seat
row
position
Flight
airline
flightNumber
update( )
CorporatePassenger
discount

Inheritance

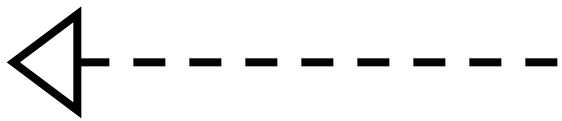
- **Inheritance** is a relationships between a superclass and its subclasses
- There are two ways to find inheritance:
  - Generalization
  - Specialization
- Common attributes, operations, and/or relationships are shown at the highest applicable level in the hierarchy

Inheritance

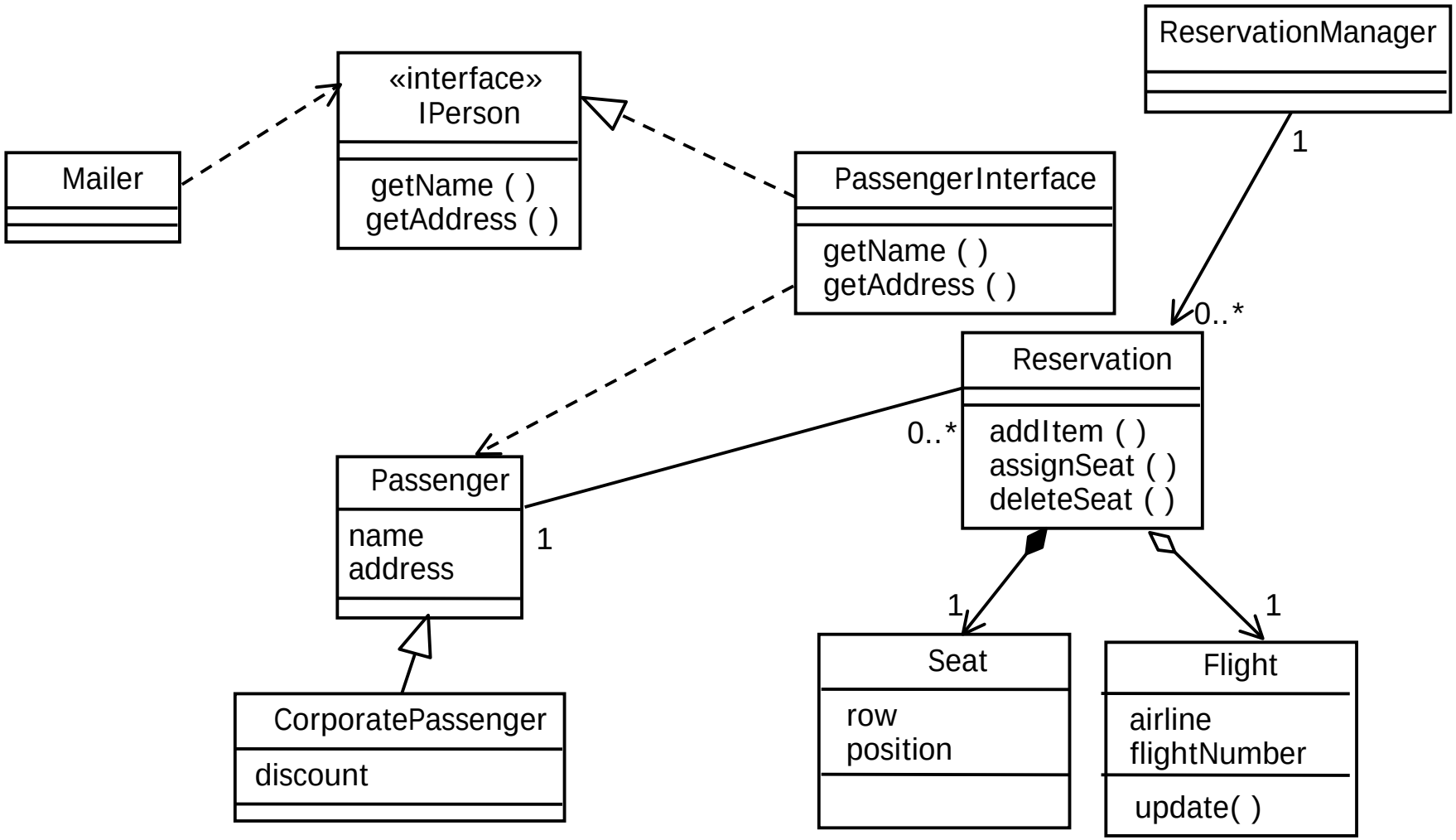


Realization

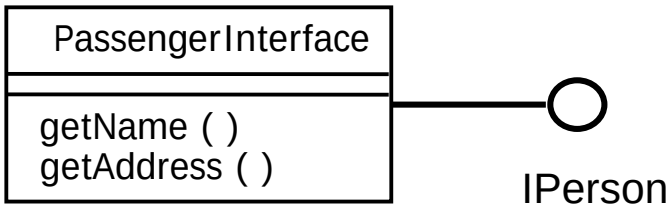
- **Realization** is a relationship between a specification and its implementation



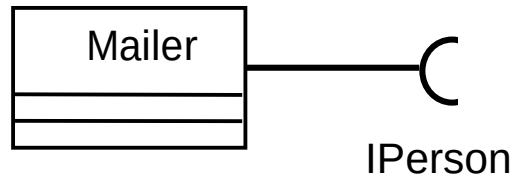
Realization



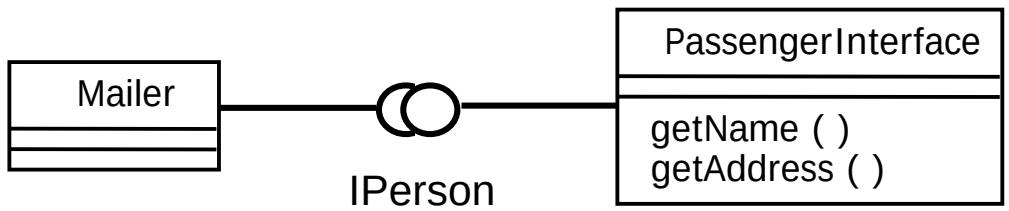
UML 2 Interface Notation



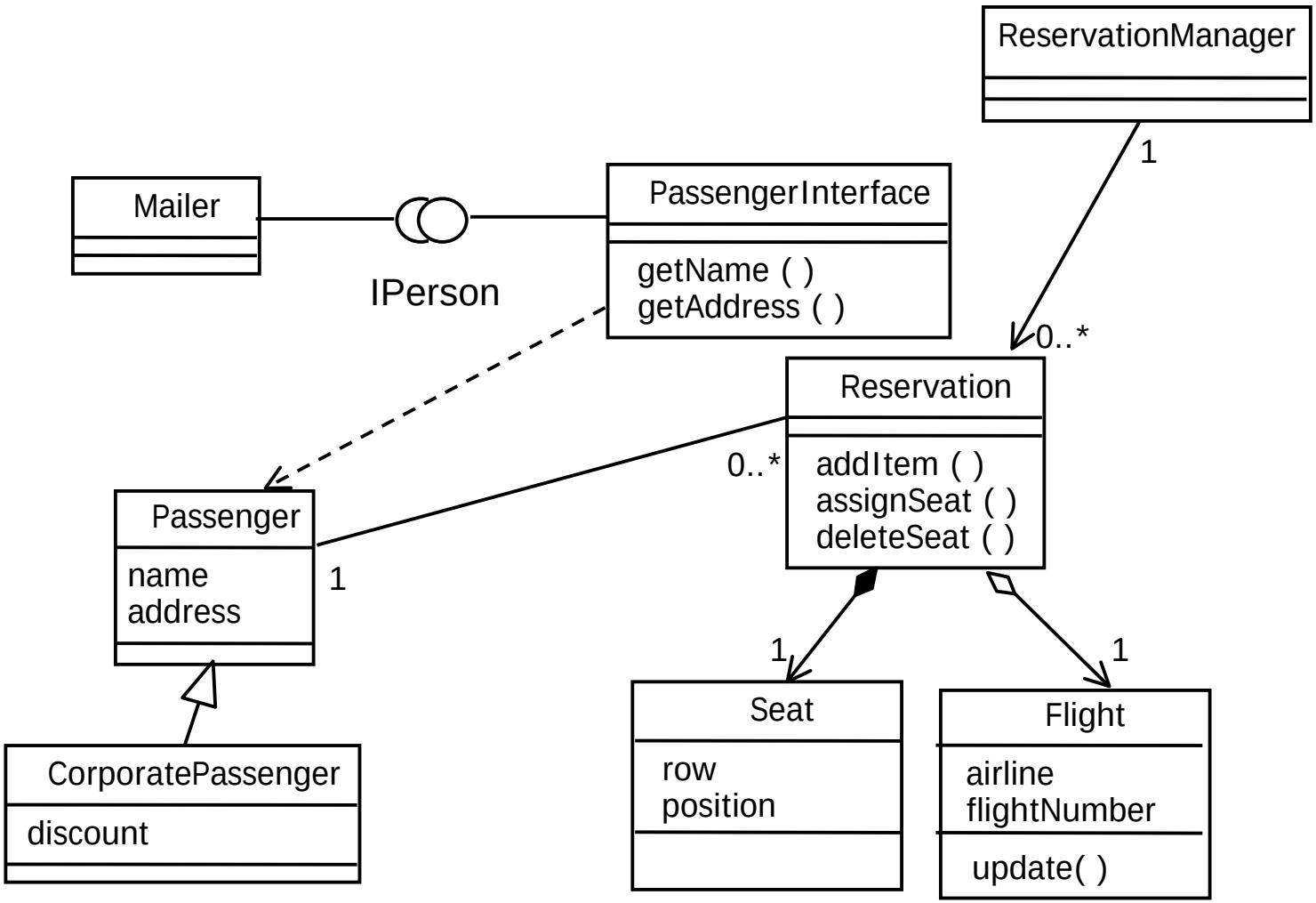
PassengerInterface is the *implementation* IPerson interface



Mailer *uses* the IPerson interface



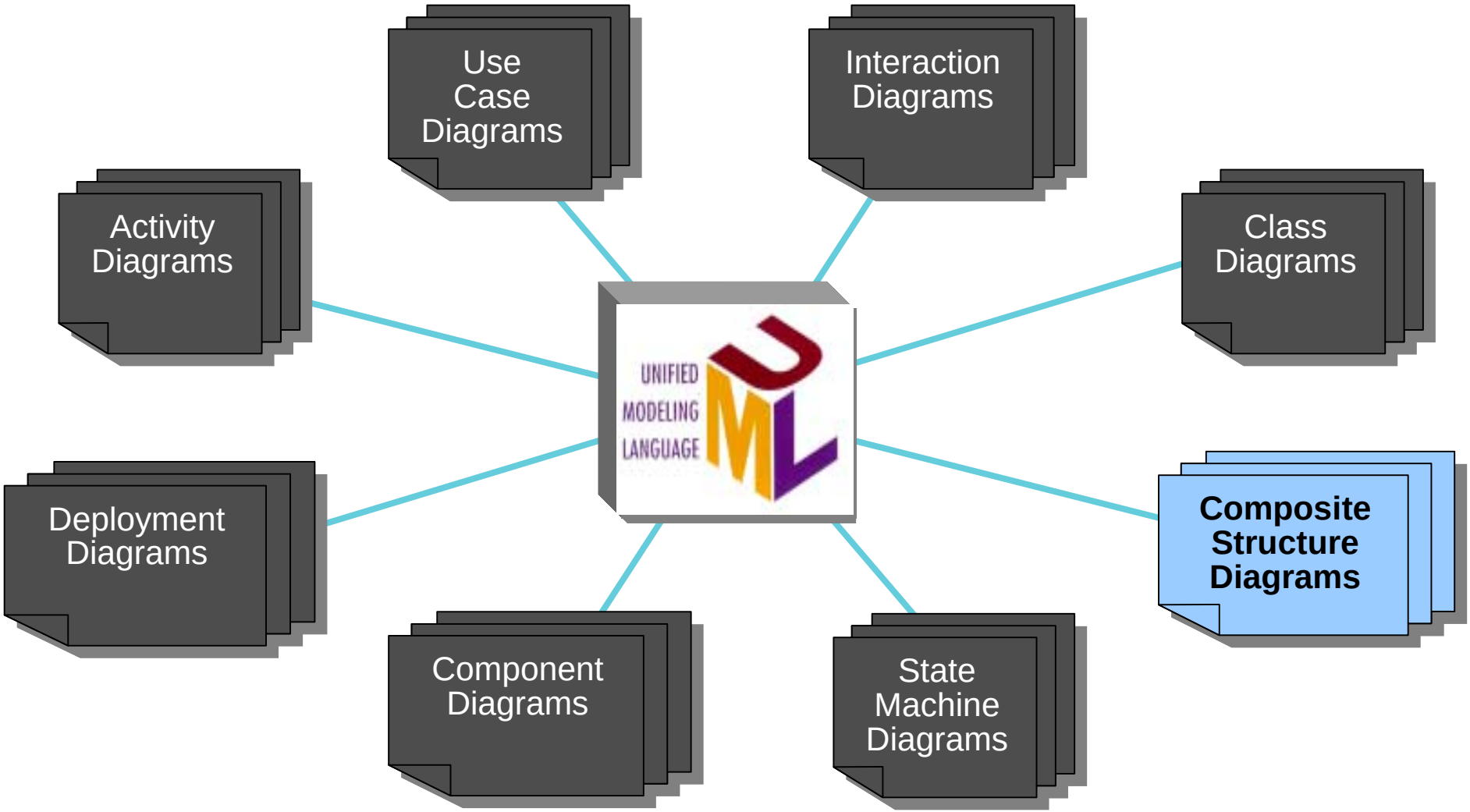
Realization



UML 2.0 Changes

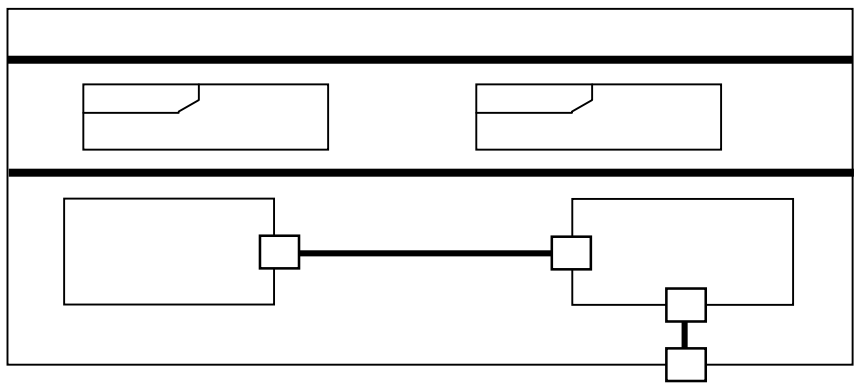
- Notation for a required interfaces () added

UML 2.0 Diagrams



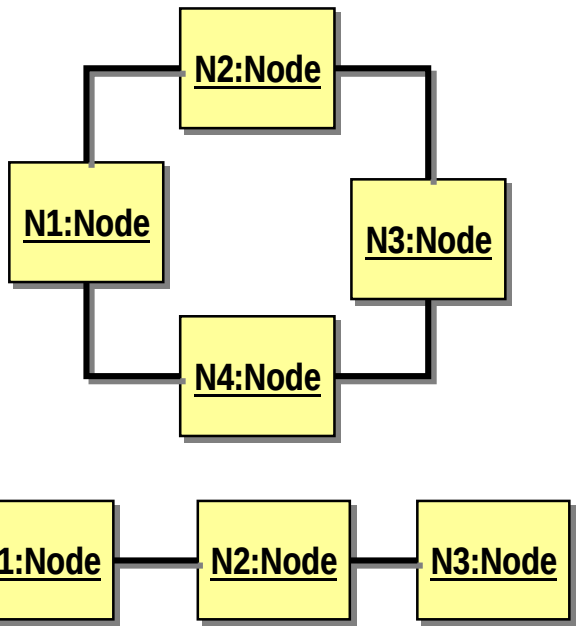
Composite Structure Diagrams

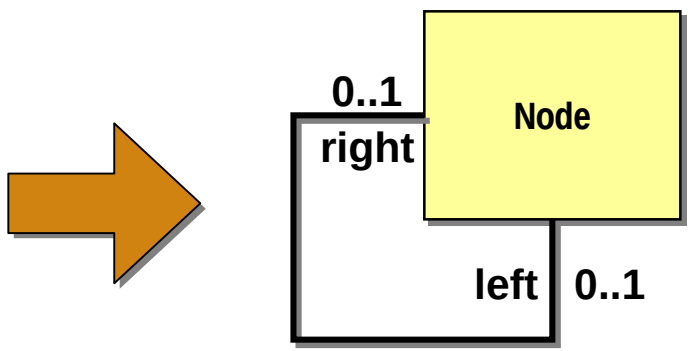
- **Composite Structure diagrams** show the internal structure of a classifier and its interaction points to other parts of the system
  - Instance view of the world
- They show how the contained parts work together to supply the behavior of the container



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

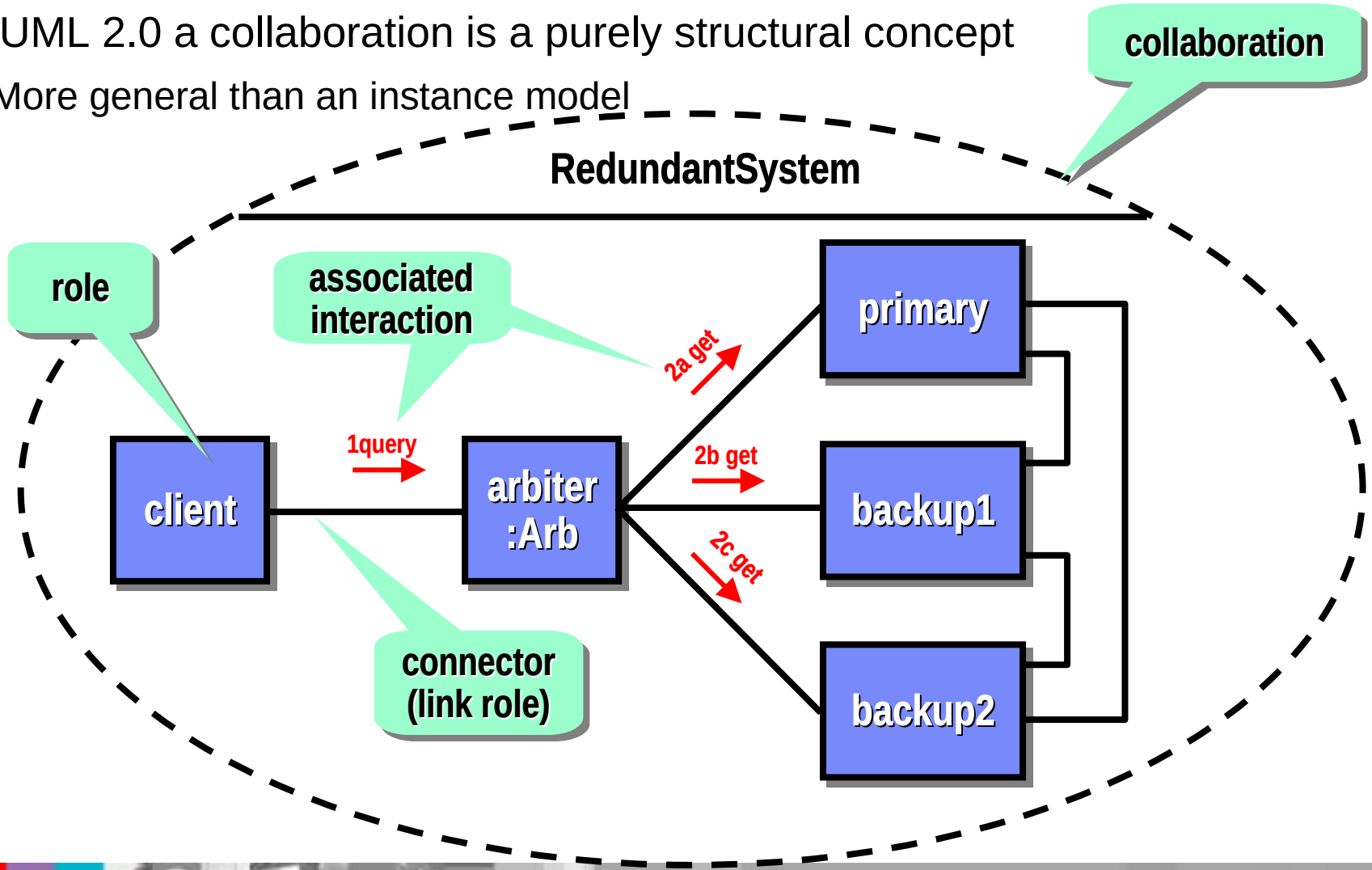




Same class diagram describes both systems!

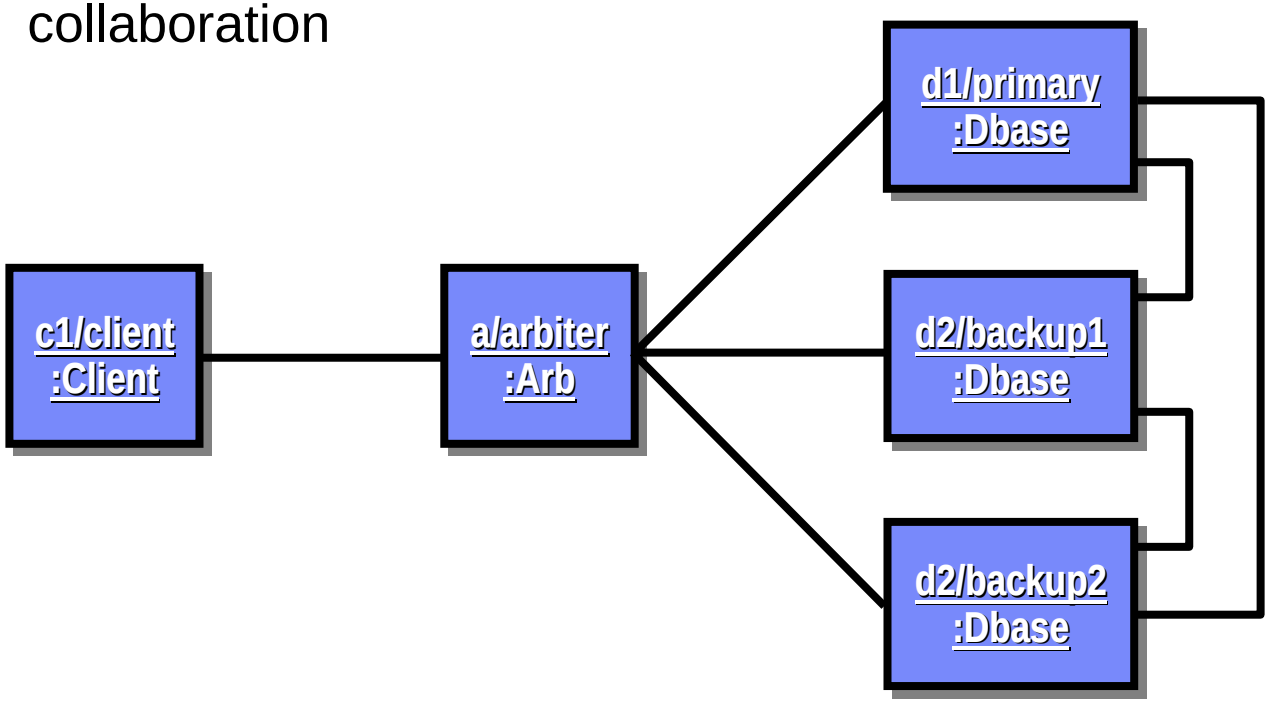
Collaborations

- In UML 2.0 a collaboration is a purely structural concept
  - More general than an instance model



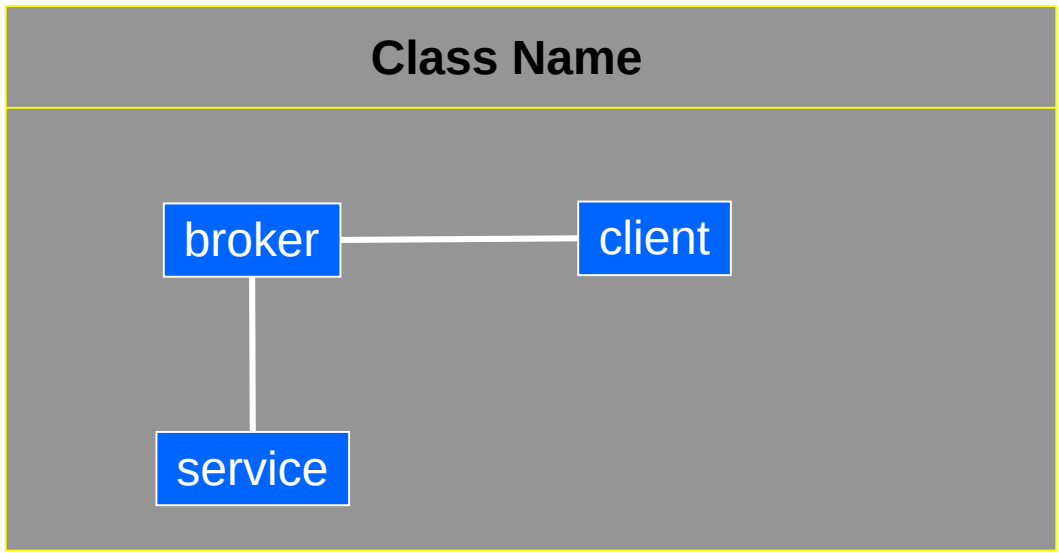
Roles and Instances

- Specific object instances playing specific the roles in a collaboration



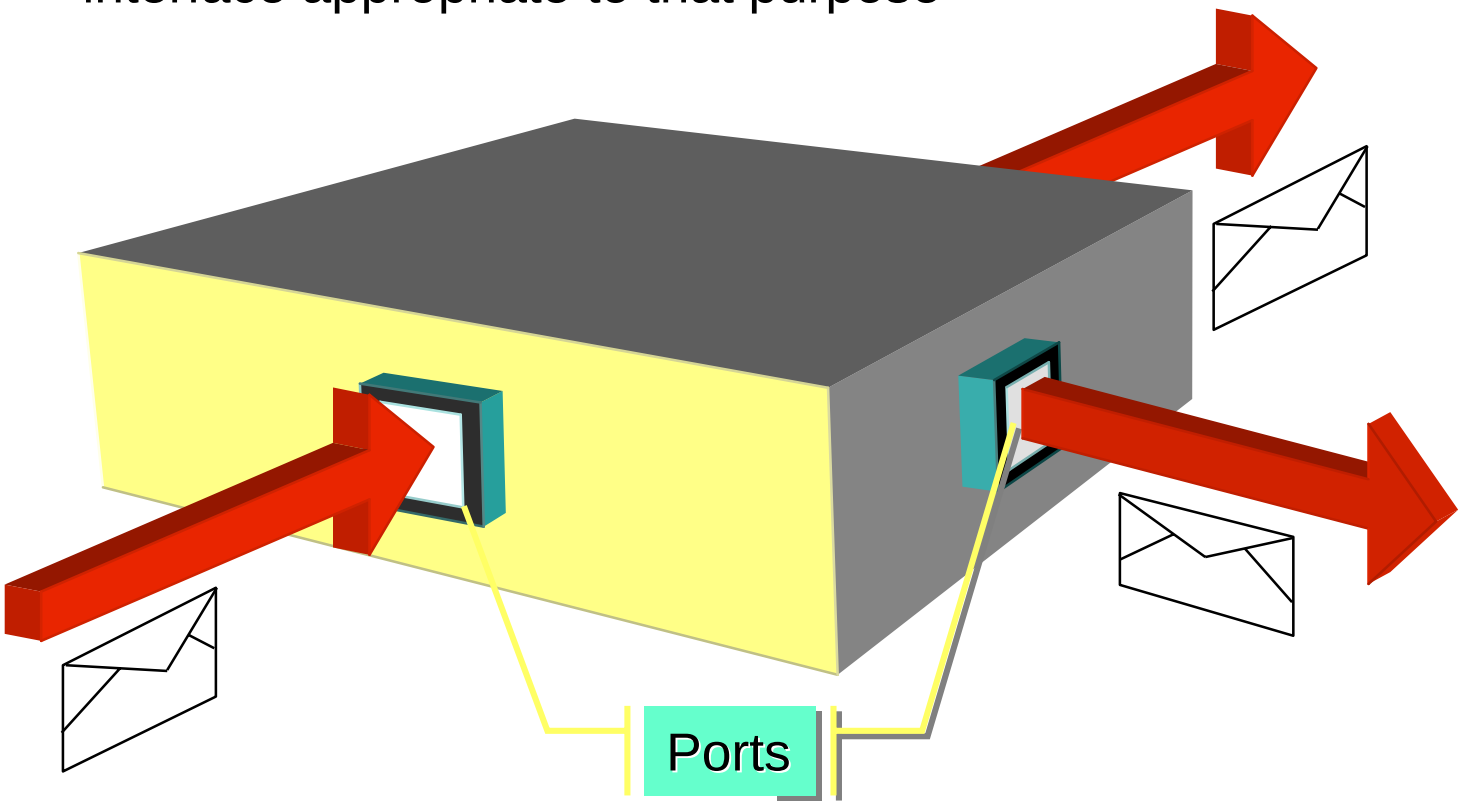
Structured Class

- A complex class comprised of internal “parts”
- Desired structure is asserted rather than constructed
  - Class constructor automatically creates desired structures
  - Class destructor automatically cleans up



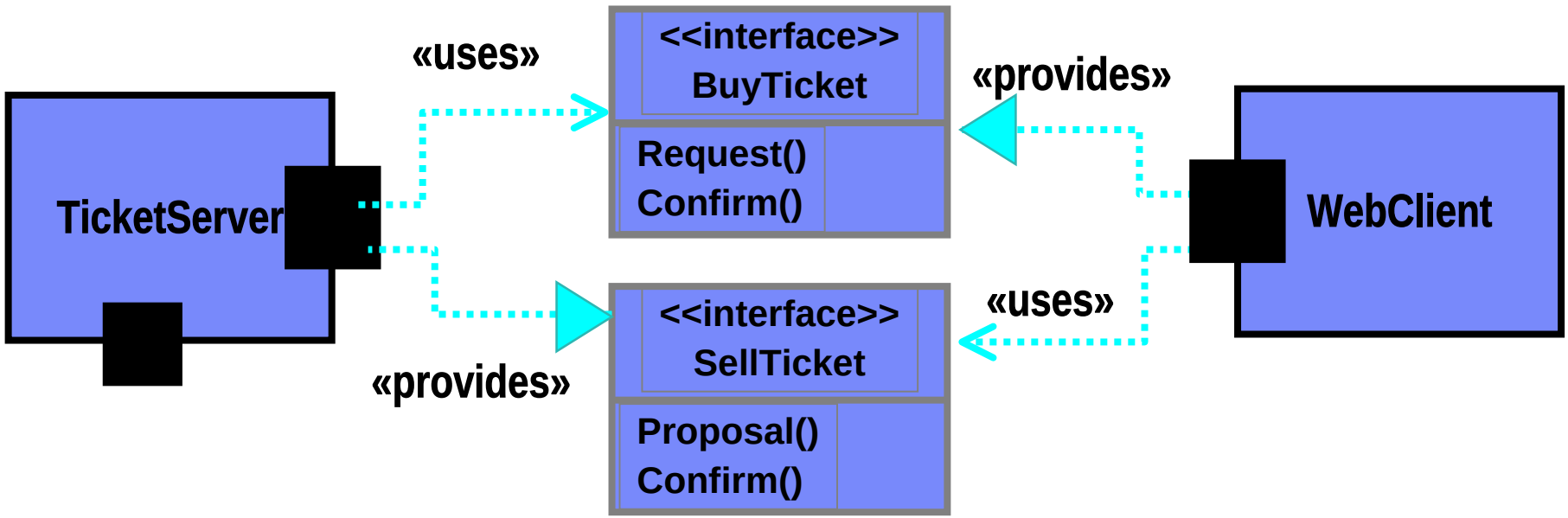
Ports

- Interaction points
- Each port is dedicated to a specific purpose and presents the interface appropriate to that purpose



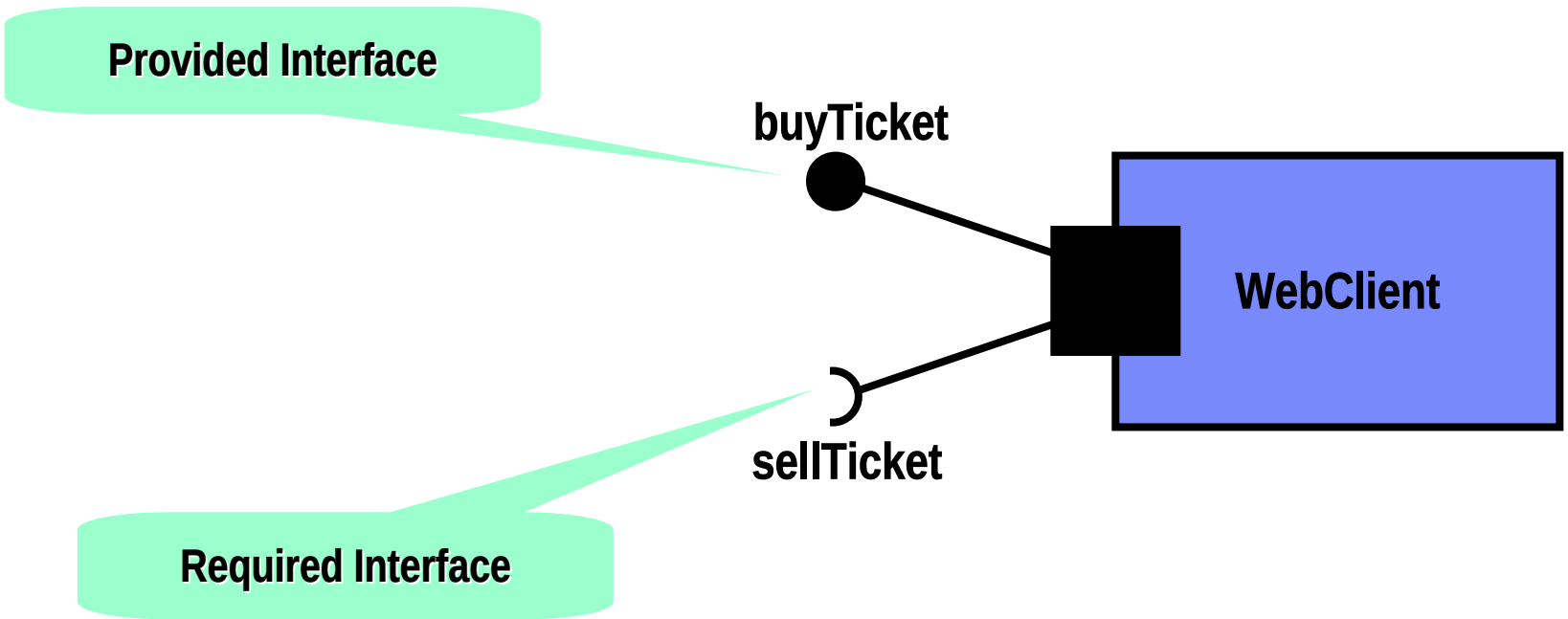
Port Semantics

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



Ports: Alternative Notation

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



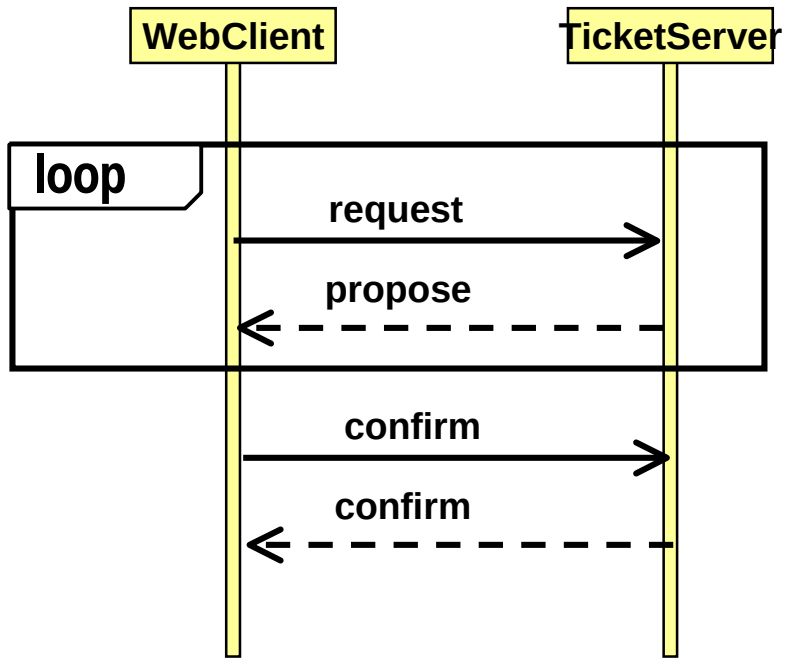
Assembling Communicating Objects

- Ports can be joined by connectors to model communication channels
  - At runtime, the WebClient is linked to the TicketServer

A connector is constrained by a protocol
Static typing rules apply (compatible protocols)

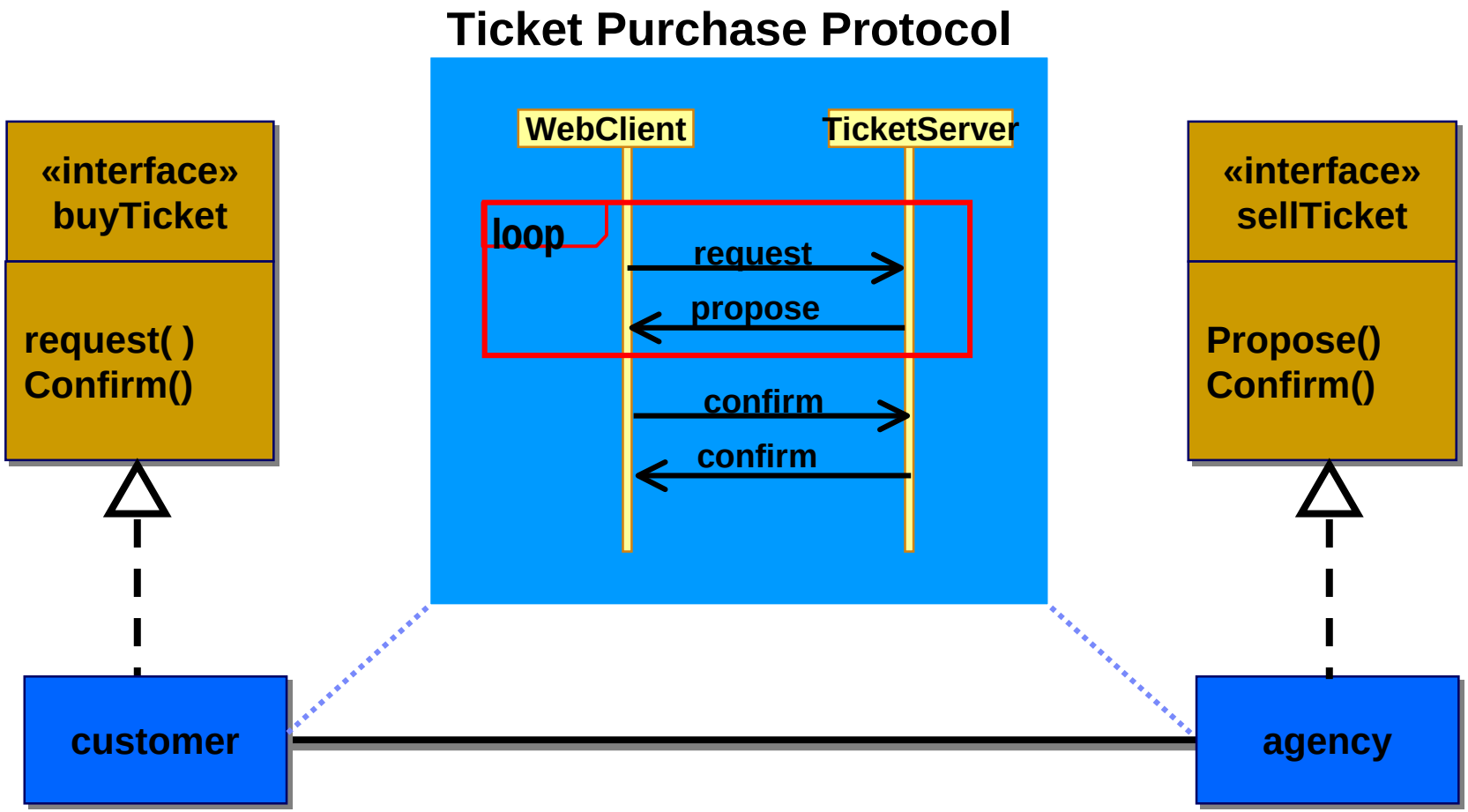
Protocols: Reusable Interaction Sequences

- Communication sequences that
  - Conform to a pre-defined dynamic order
  - Are defined generically in terms of role players
  - E.g., ticket purchase protocol

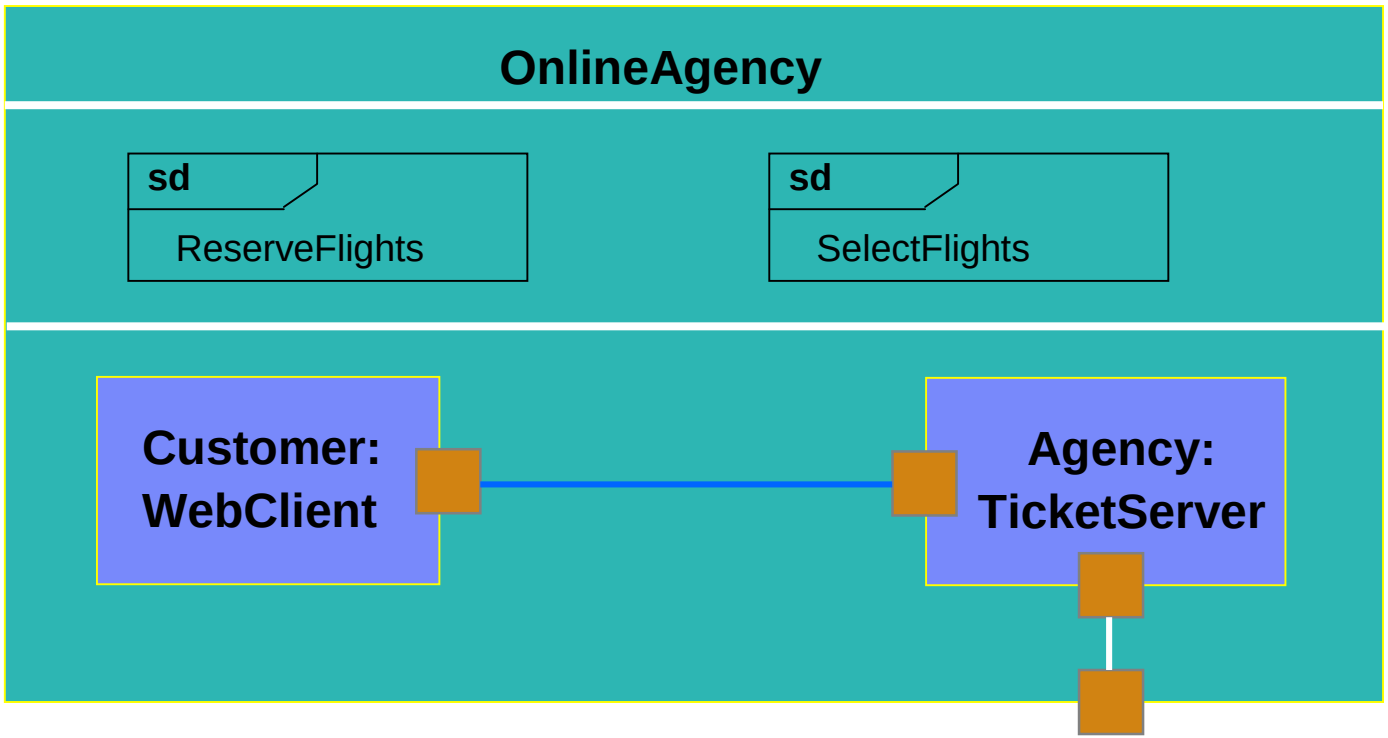


Modeling Protocols with UML 2.0

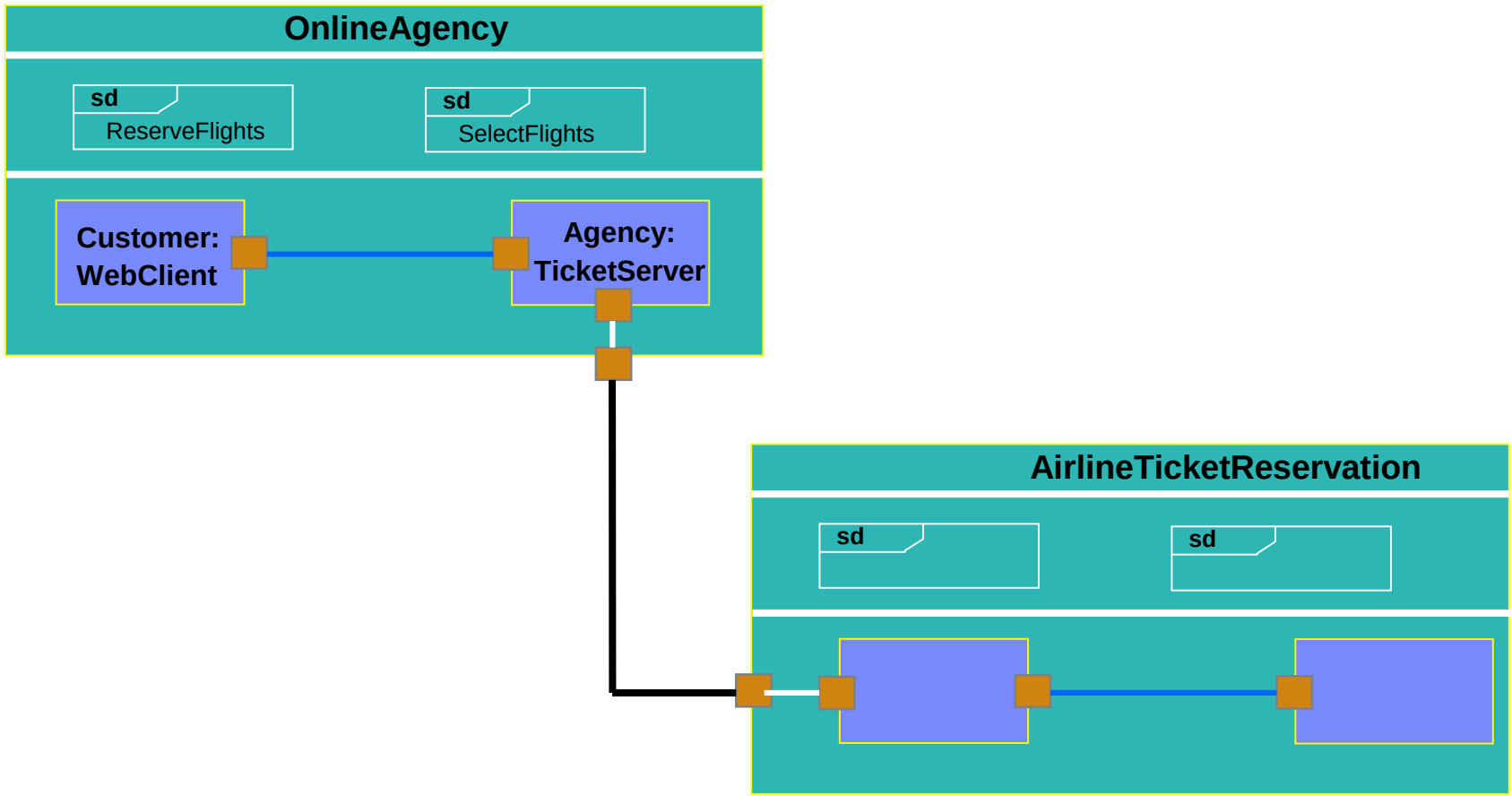
- A collaboration structure with interactions



Structured Classes: Putting Them Together



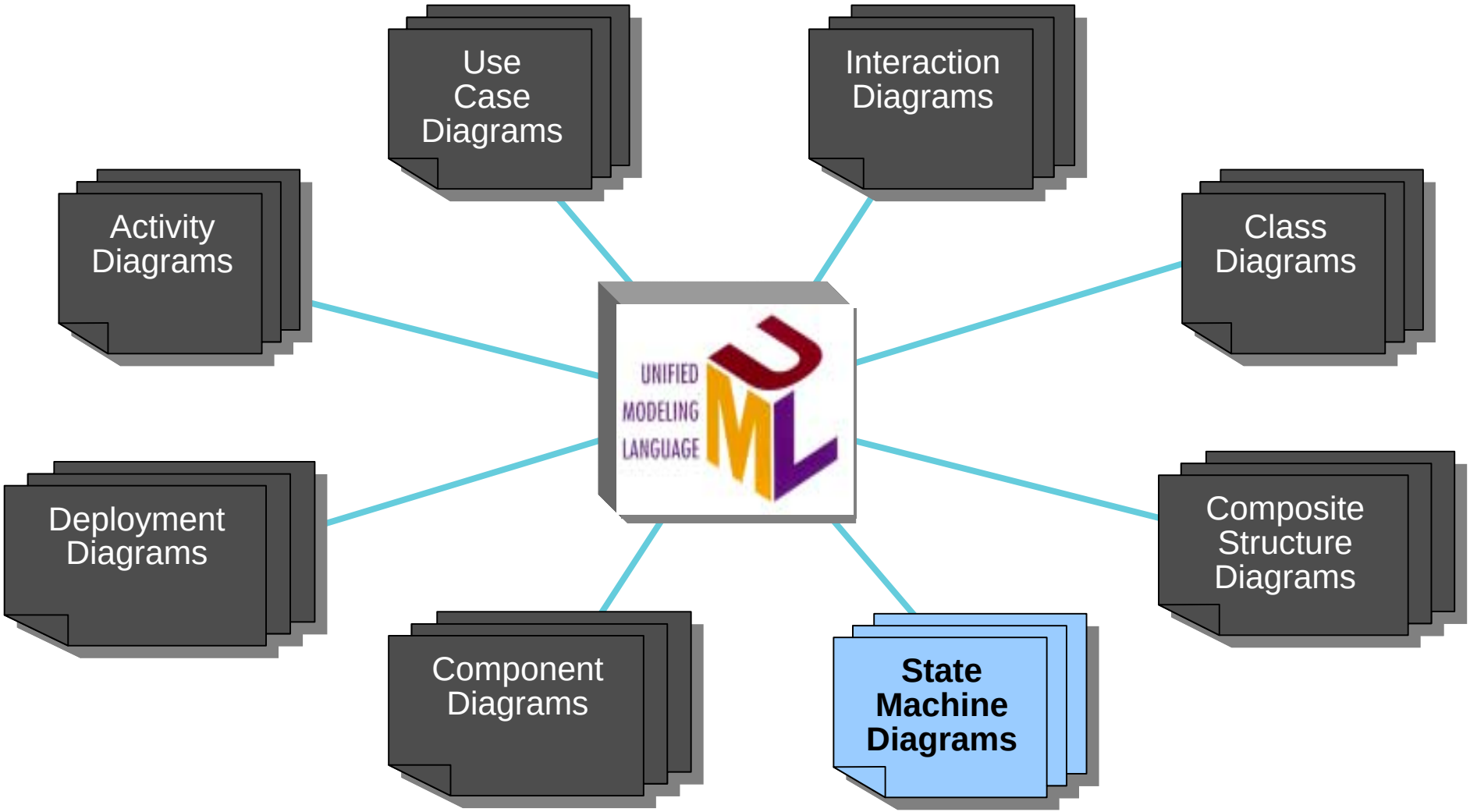
Structured Classes: Putting Them Together



UML 2.0 Changes

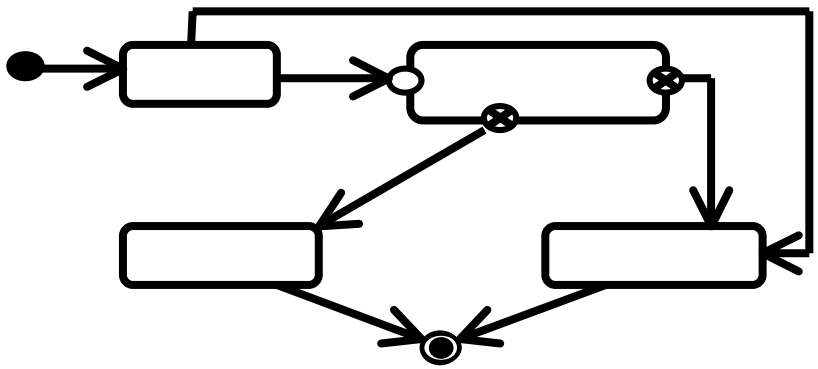
- Composite structure diagrams, structured classes, ports and connectors are new

UML 2.0 Diagrams

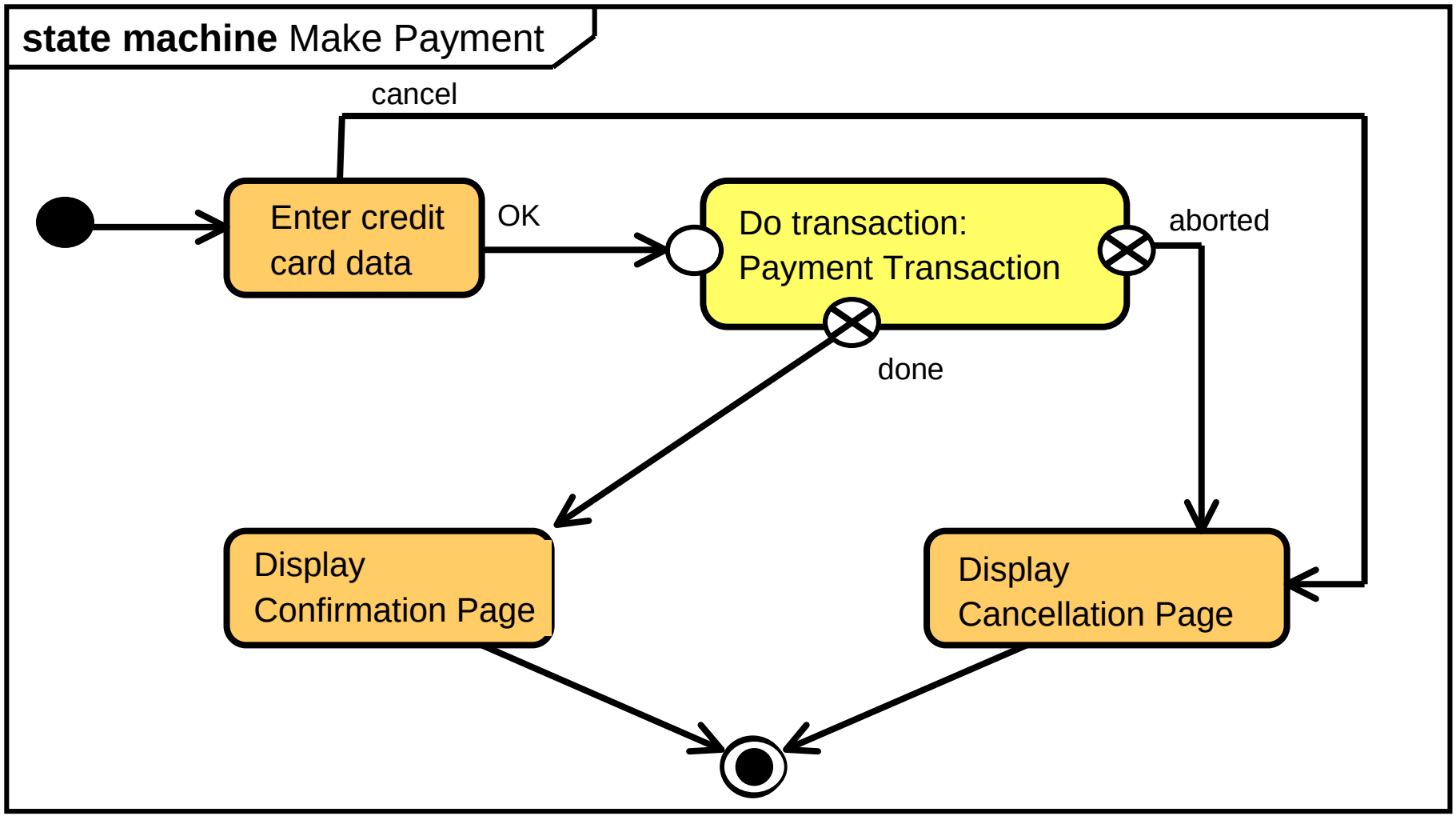


State Machine Diagram

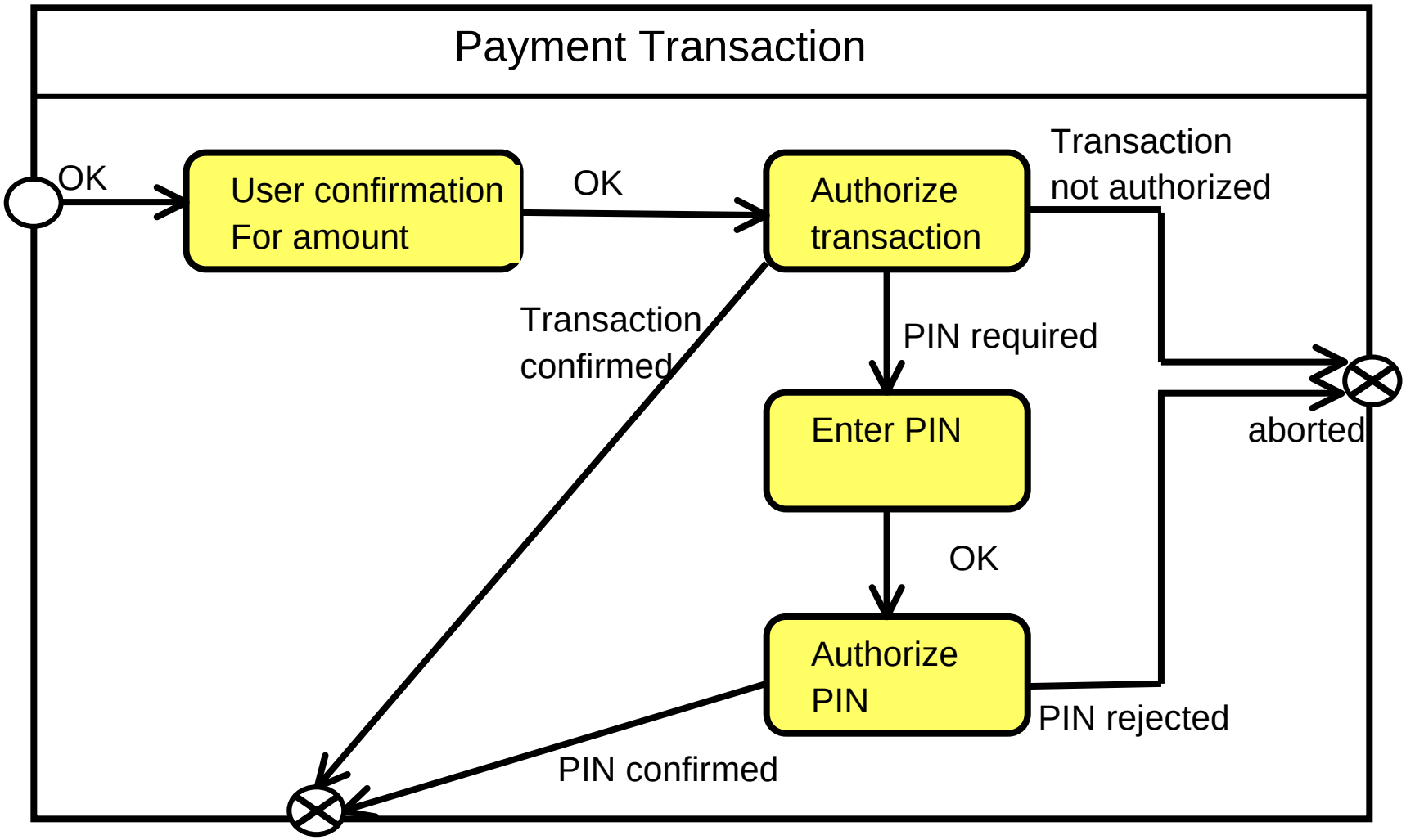
- A **state machine diagram** shows
  - The life history of a given class
  - The events that cause a transition from one state to another
  - The actions that result from a state change
- State machine diagrams are created for objects with significant dynamic behavior



State Diagram

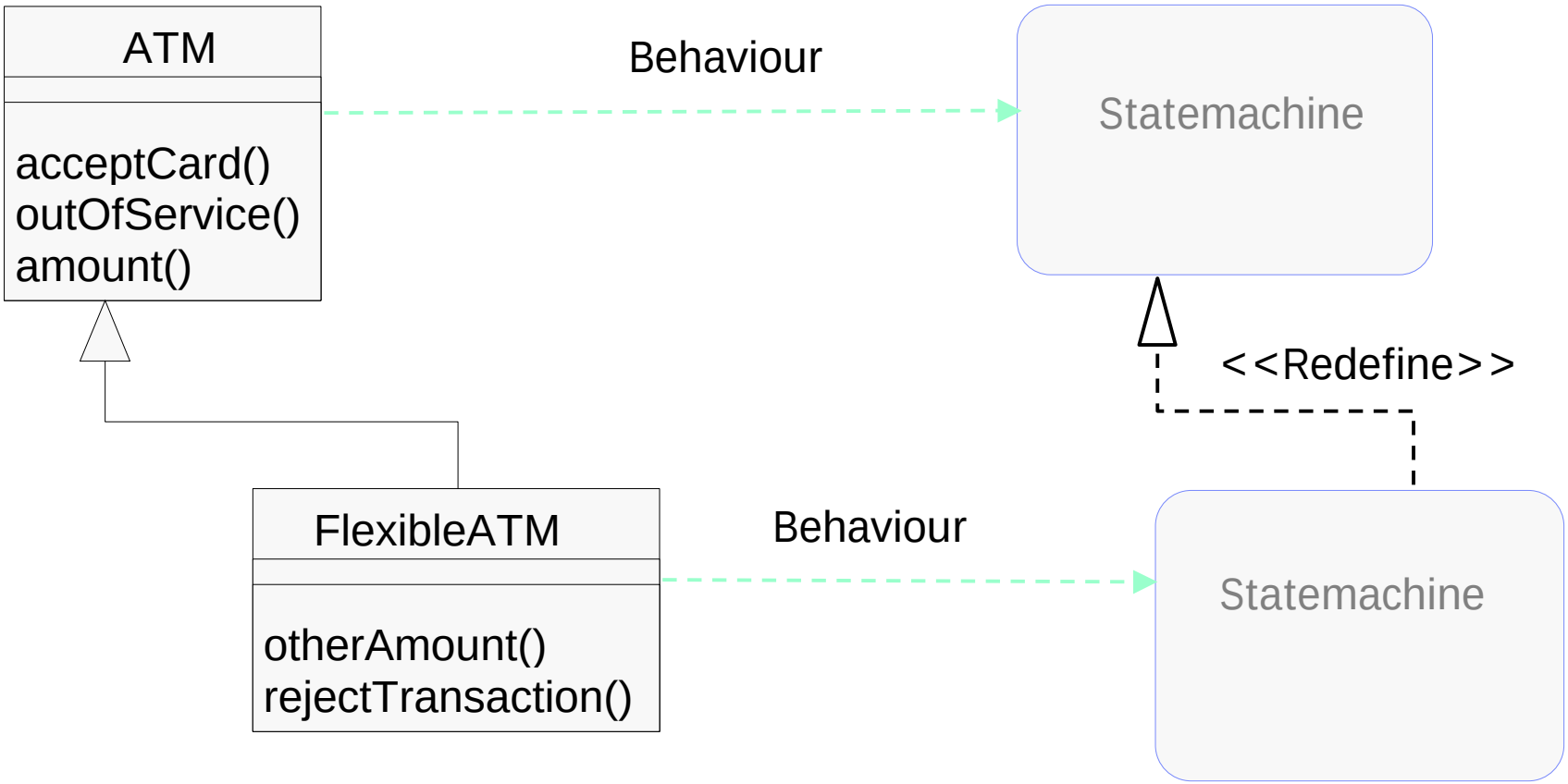


Submachine

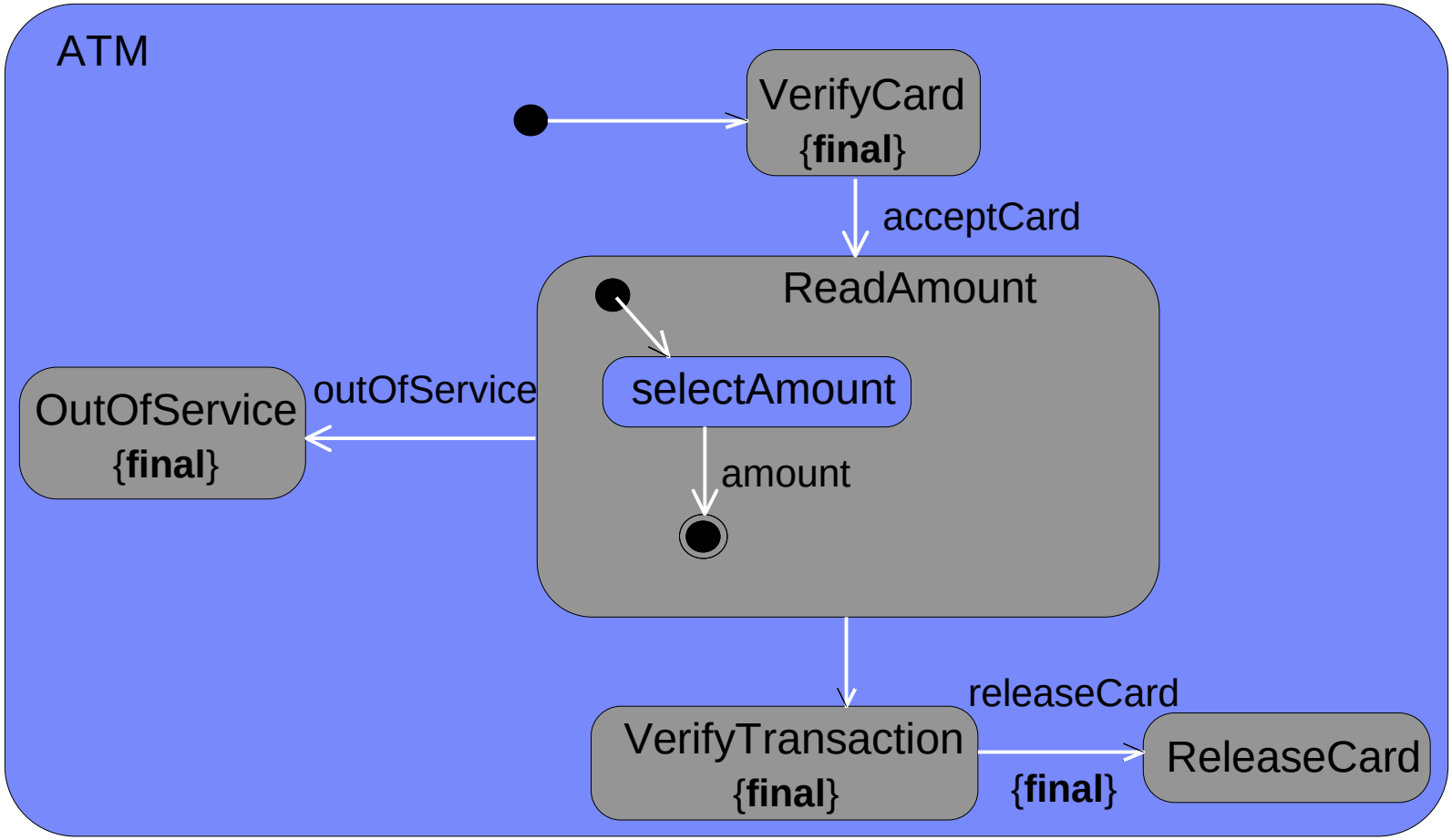

Payment Transaction
OK
User confirmation
For amount
OK
Authorize
transaction
Transaction
not authorized
Transaction
confirmed
PIN required
Enter PIN
aborted
OK
Authorize
PIN
PIN rejected
PIN confirmed

Specialization

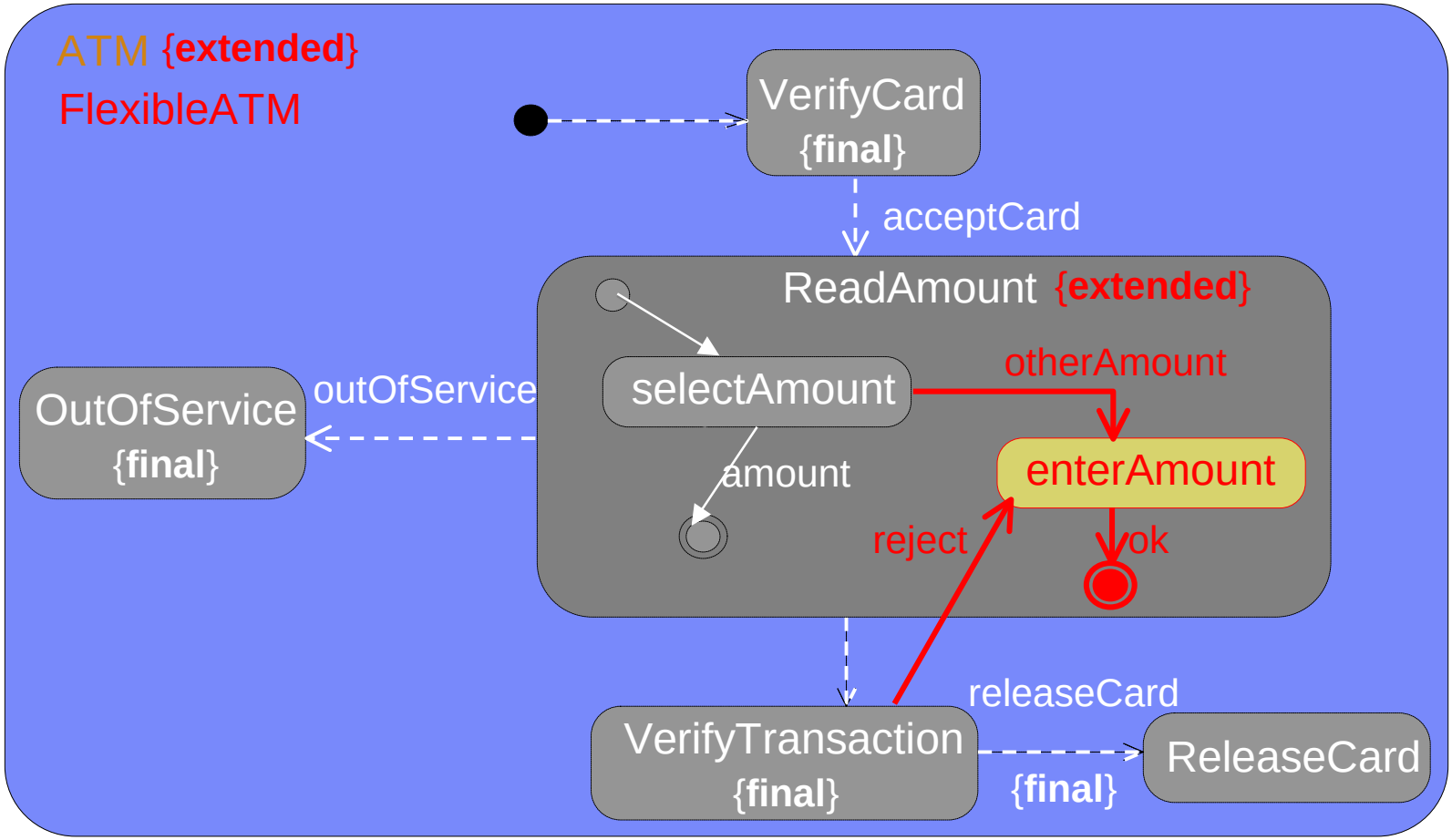
- Redefinition as part of standard class specialization



Example: State Machine Redefinition



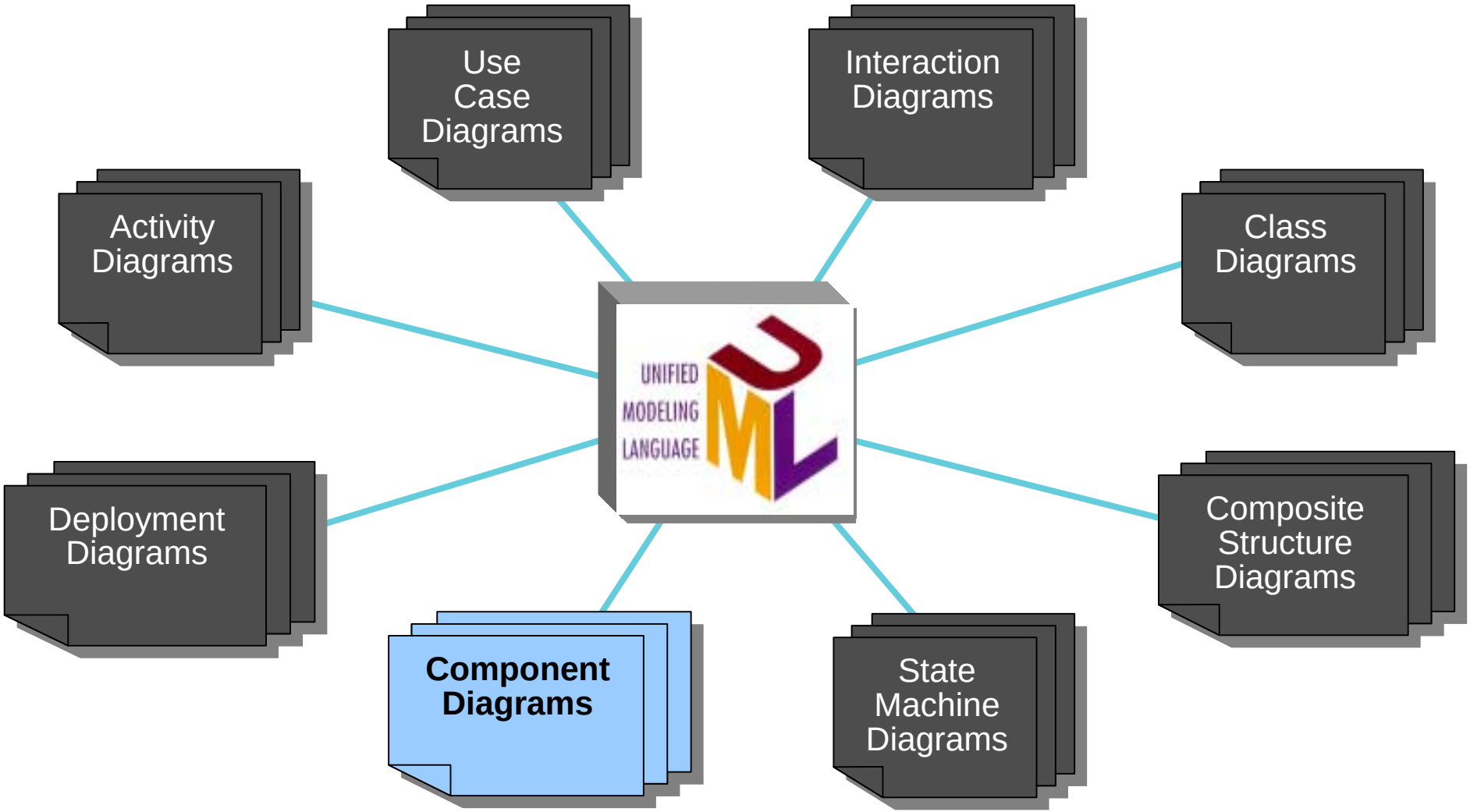
State Machine Redefinition



UML 2.0 Changes

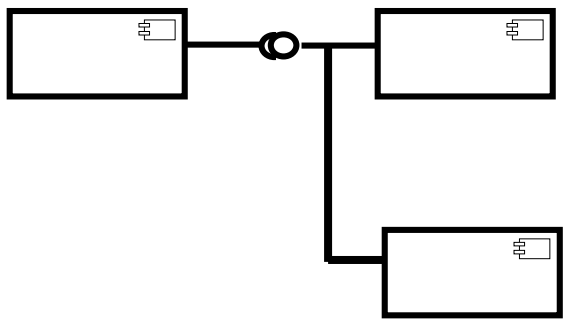
- Protocol state machines added

UML 2.0 Diagrams

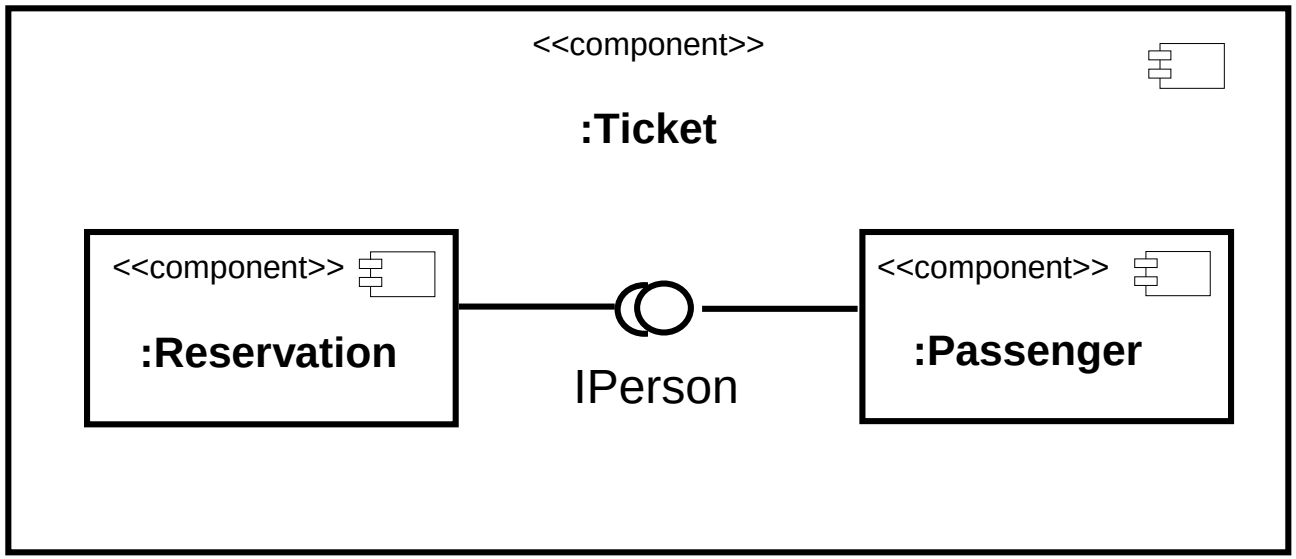


Component Diagram

- A **component** is a modular unit with well defined interfaces that is replaceable within its environment
- Components can be logical or physical
- Logical components
  - Business components, process components etc.
- Physical components
  - EJB components, .NET components, COM components etc.



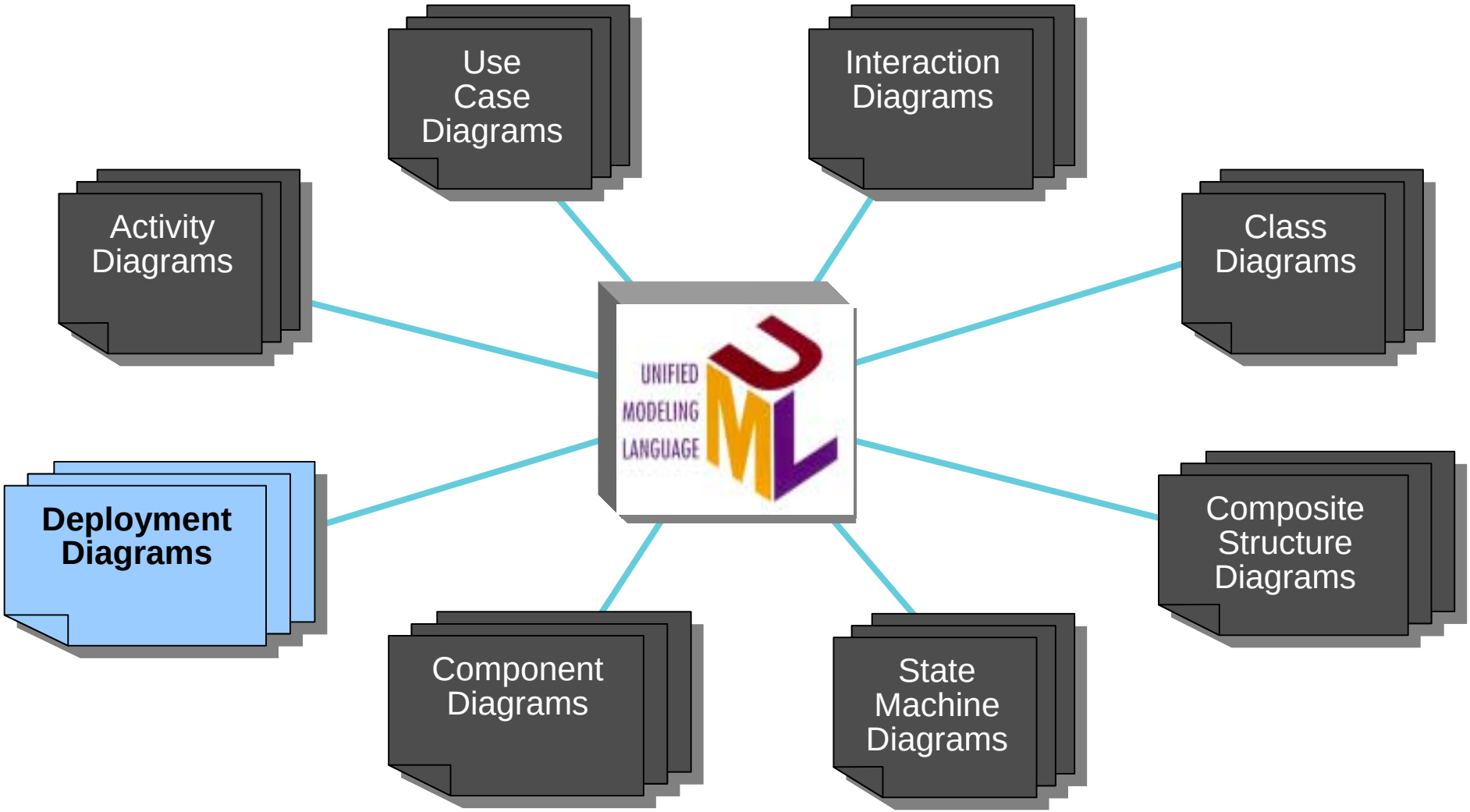
Component Diagram



UML 2.0 Changes

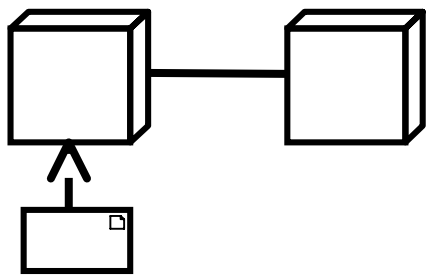
- Notation for a component changed
- Component may have ports

UML 2.0 Diagrams



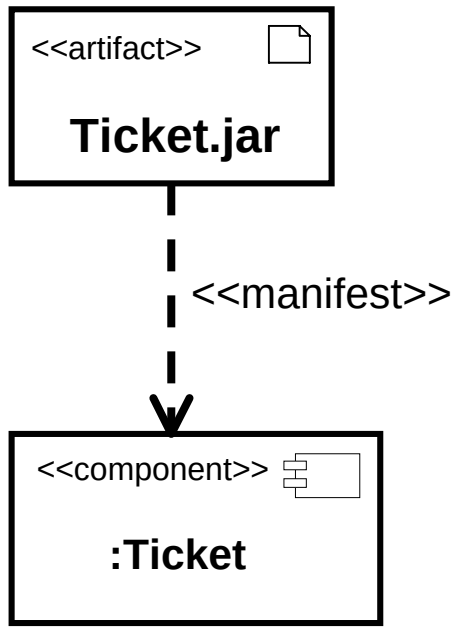
Deployment Diagram

- **Deployment diagrams** show the execution architecture of systems
- Nodes are connected by communication paths to create network systems

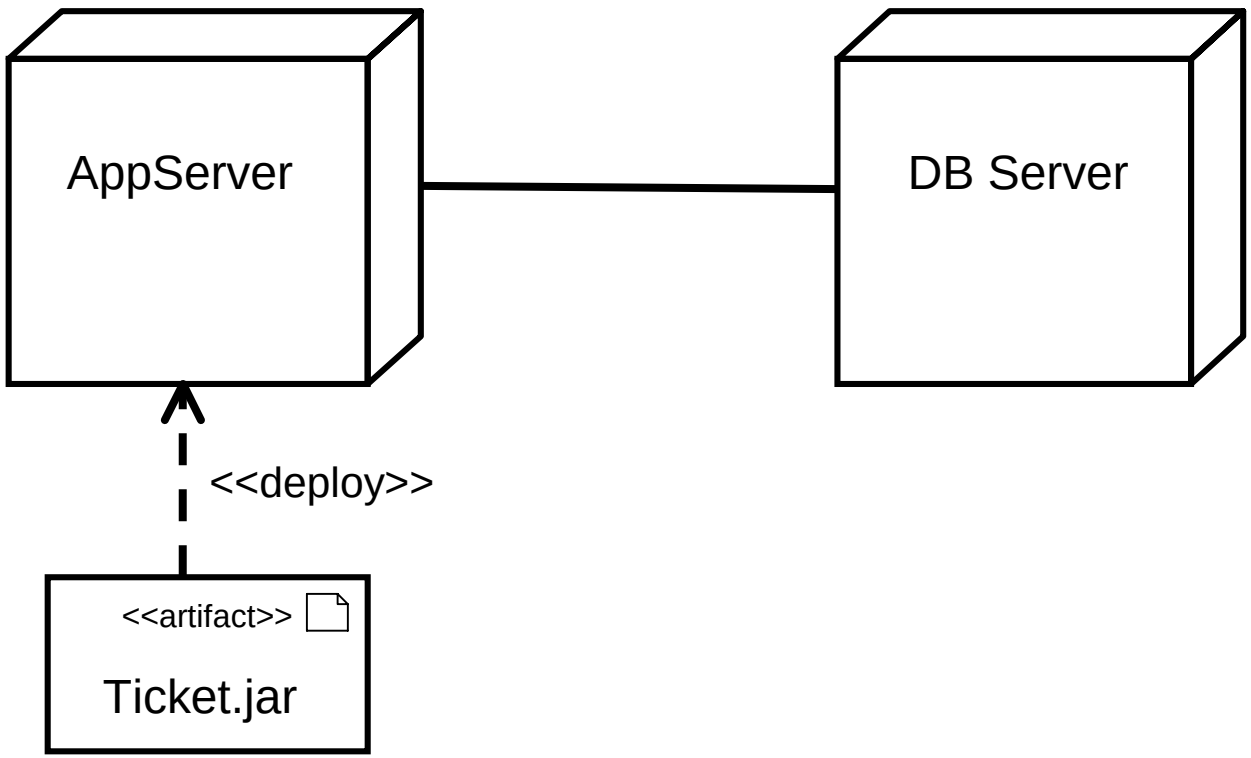


Artifacts

- An **artifact** represents a physical entity
- Examples
  - Model files, source files, scripts, binary executable files



Deployment Diagram



UML 2.0 Changes

- Artifacts and deployment specifications are new
- New Node types introduced
  - Execution environment (virtual machine, app server, and other “middleware”)

Agenda

- The Importance of Modeling
- The Unified Modeling Language
- UML Diagrams
- Extending the UML

Extending the UML

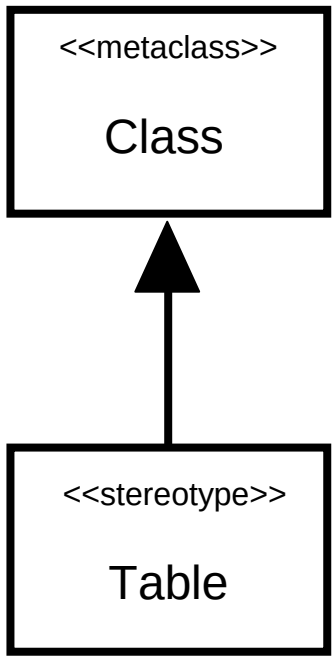
- In order to model something effectively, the language that you use to model it must be rich and expressive enough to do so
- “Out of the box” UML is sufficient for modeling object-oriented software systems
- BUT… there are many more models that are useful in the software development process
- UML can easily be extended to add more semantics to cover other modeling situations
  - Database models
  - Business process
  - Web pages
  - On and on….

Extension Mechanisms

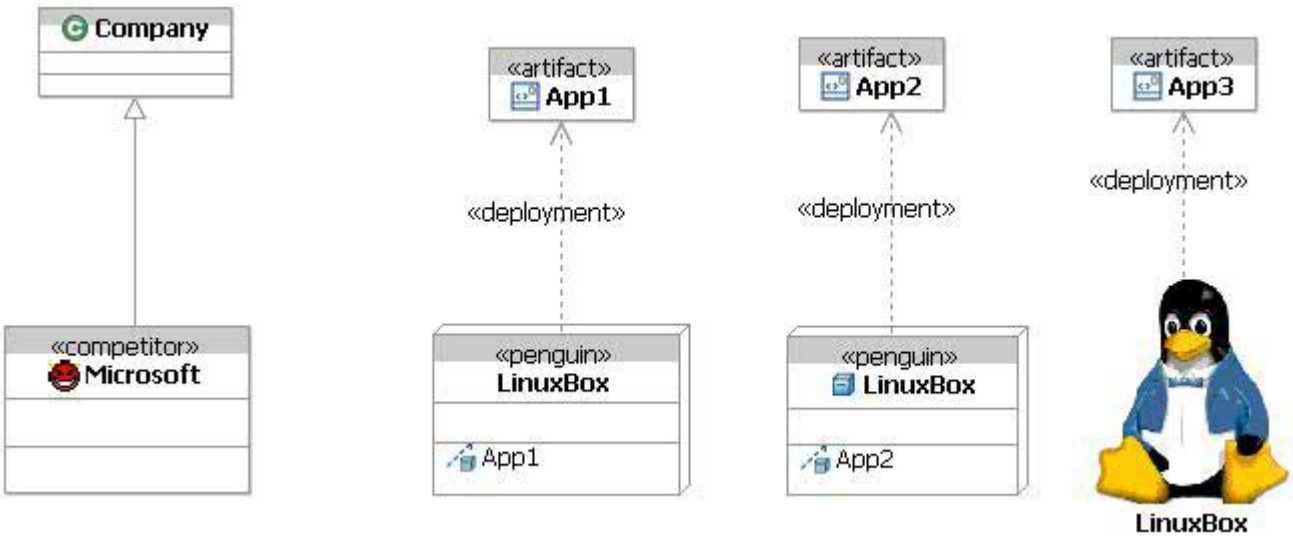
- Stereotypes
- Tag definitions and tagged values
- Constraints

Stereotypes

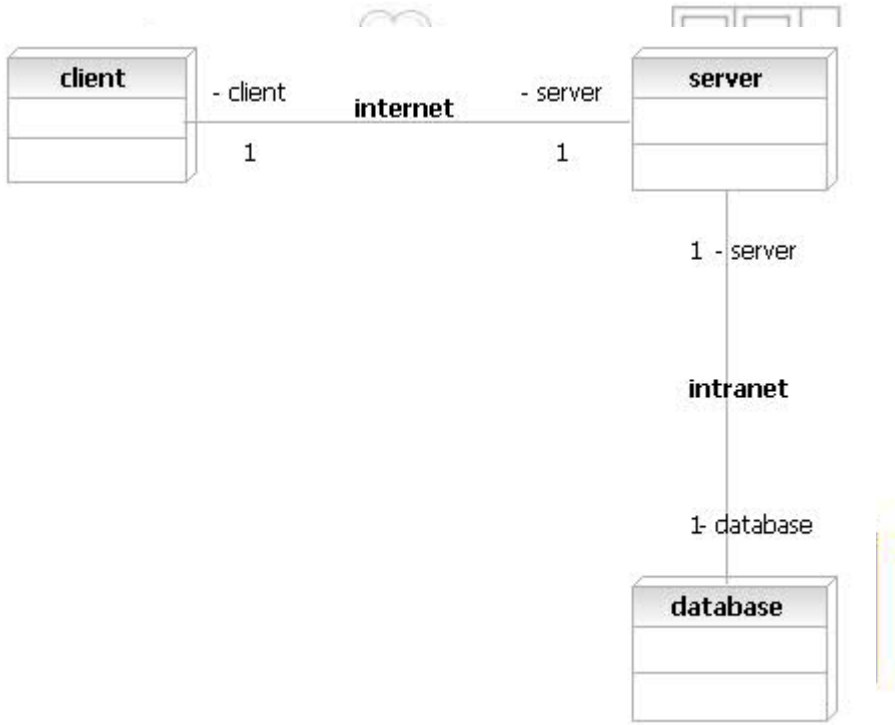
- A more refined semantic interpretation for a model element



Stereotype Examples



Deployment Diagram



Tag Definitions and Tagged Values

- A **tag definition** the ability to associate extra information with a modeling element
  - Defines the name and the type of information
- Example
  - A database table could be modeled as a stereotyped class
  - Columns could be modeled as stereotyped attributes
  - Tag definition for a column could be “NullsAllowed”
- A **tagged value** is the actual instance of a tag definition with a value that is associated to the modeling element
- Example
  - Create a table Person
  - Add a column, you MUST supply a value for “NullsAllowed” (true or false)

Constraints

- A **constraint** is a rule that is applied to a modeling element
  - Represented by curly braces in UML
- Used to evaluate if a modeling element is “well-formed”
- Example
  - The name of a column cannot exceed the maximum length for column names for the associated database
- The language of constraints is Object Constraint Language (OCL)

OCL

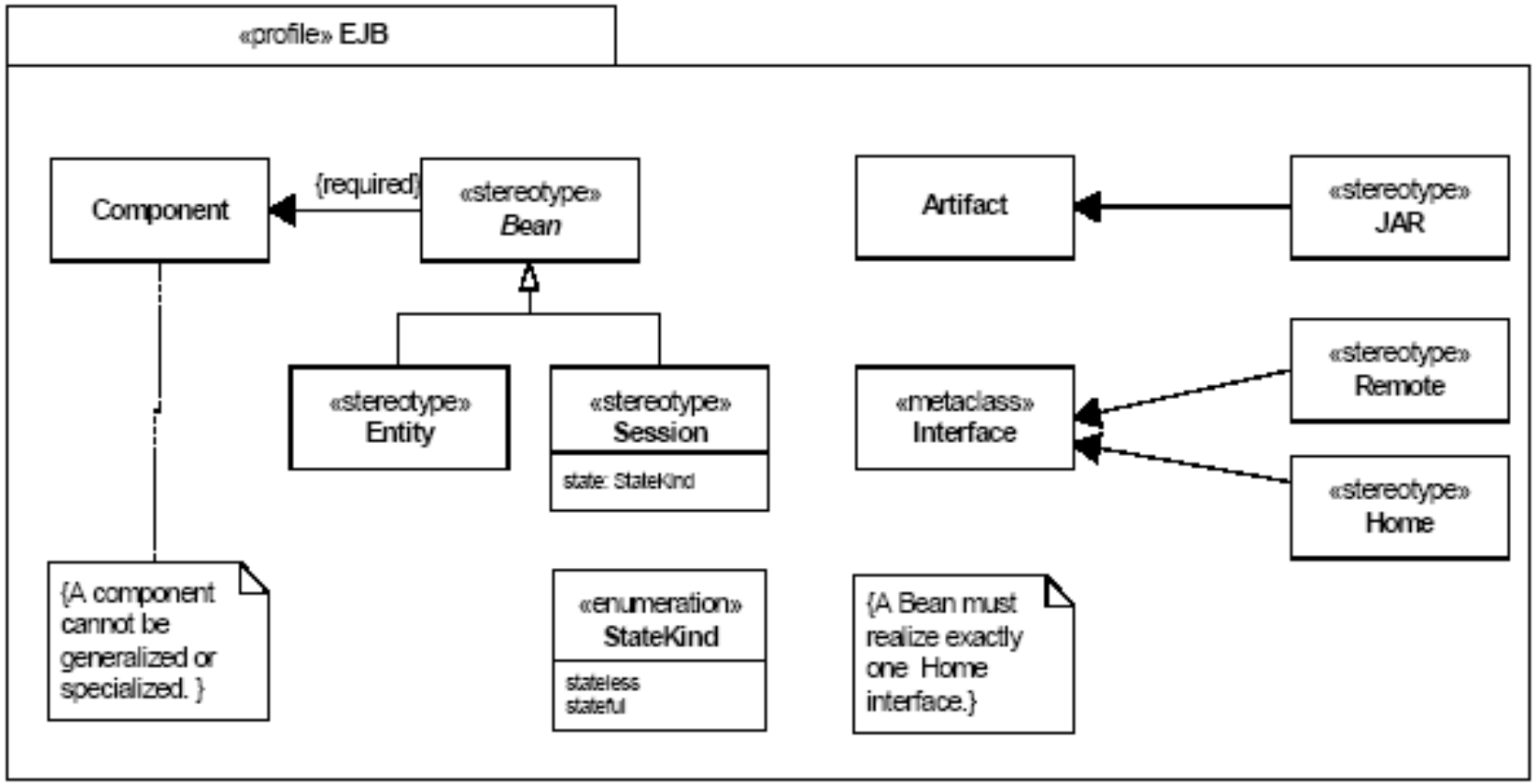
- OCL is another OMG specification
- Defines an object-oriented language that is similar to Smalltalk, Java and C++
- Formal way to express model ” well formedness” rules

```
{let x = self.model().OwnedProfile->any(Name='Data Modeler').
TaggedValueSet->any(Name='Preferences').TagDefinition-
>any(Name='maximumidentifierlength').DefaultValue.oclAsType(UML::Int
egerTaggedValue).Value in self.Name.size() <= x}
```

Profiles

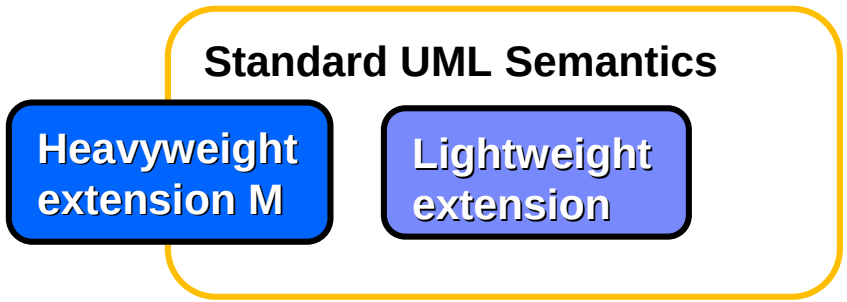
- A **profile** is a collection of steretoypes, tag definitions and constraints that work together to define new semantics for a model
- Example
  - Data modeling profile
  - Business modeling profile
  - EJB profile

EJB Profile


«profile» EJB
Component
{required}
«stereotype»
Bean
Artifact
«stereotype»
JAR
«stereotype»
Entity
«stereotype»
Session
state: StateKind
«metaclass»
Interface
«stereotype»
Remote
«stereotype»
Home
{A component cannot be generalized or specialized. }
«enumeration»
StateKind
stateless
stateful
{A Bean must realize exactly one Home interface.}

Specializing UML

- Lightweight extensions
  - Extend semantics of existing UML concepts by specialization
  - Conform to standard UML (tool compatibility)
  - Profiles, stereotypes
- Heavyweight (MOF) extensions
  - Add new non-conformant concepts or
  - Incompatible change to existing UML semantics/concepts

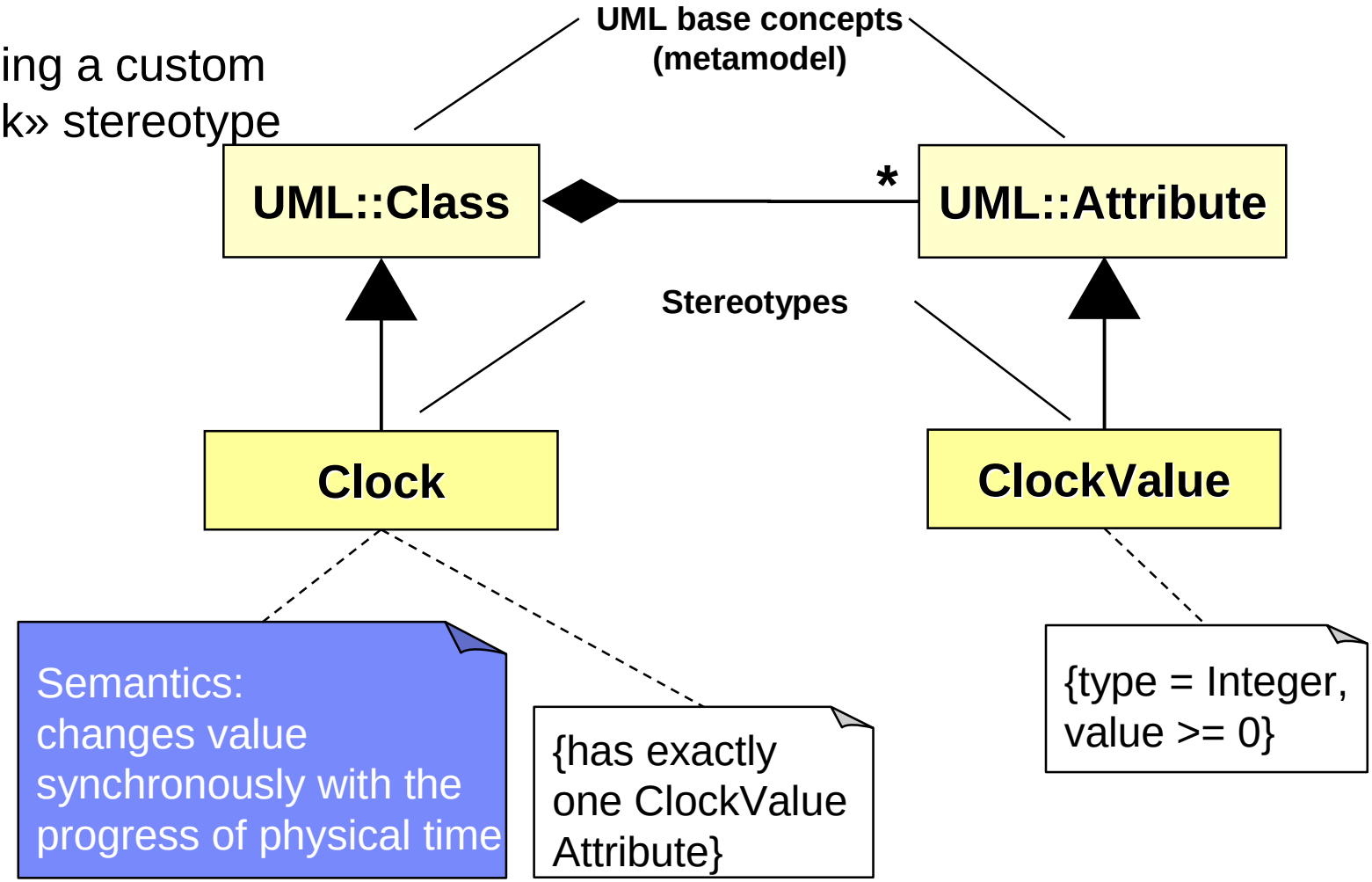


The Profile-Based Approach to DSLs

- Profile = a compatible specialization of an existing modeling language by
  - Adding constraints, characteristics, new semantics to existing language constructs
  - Hiding unused language constructs
- Advantages:
  - Supported by the same tools that support the base language
  - Reuse of base language knowledge, experience, artifacts
- Example: ITU-T standard language SDL (Z.100)
  - Modeling language used in telecom applications
  - Now defined as a UML profile (Z.109)

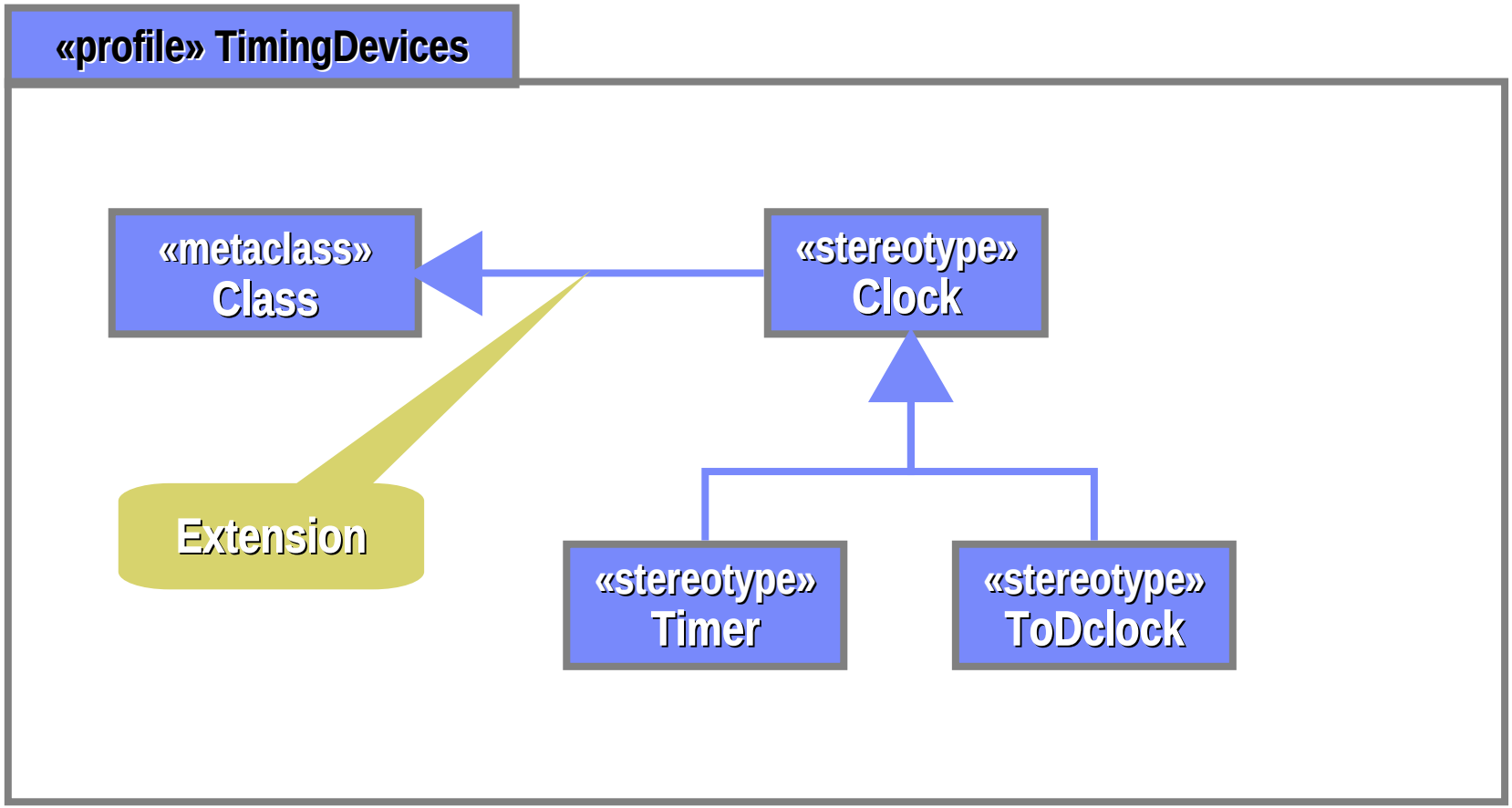
UML Profile Example

- Defining a custom «clock» stereotype



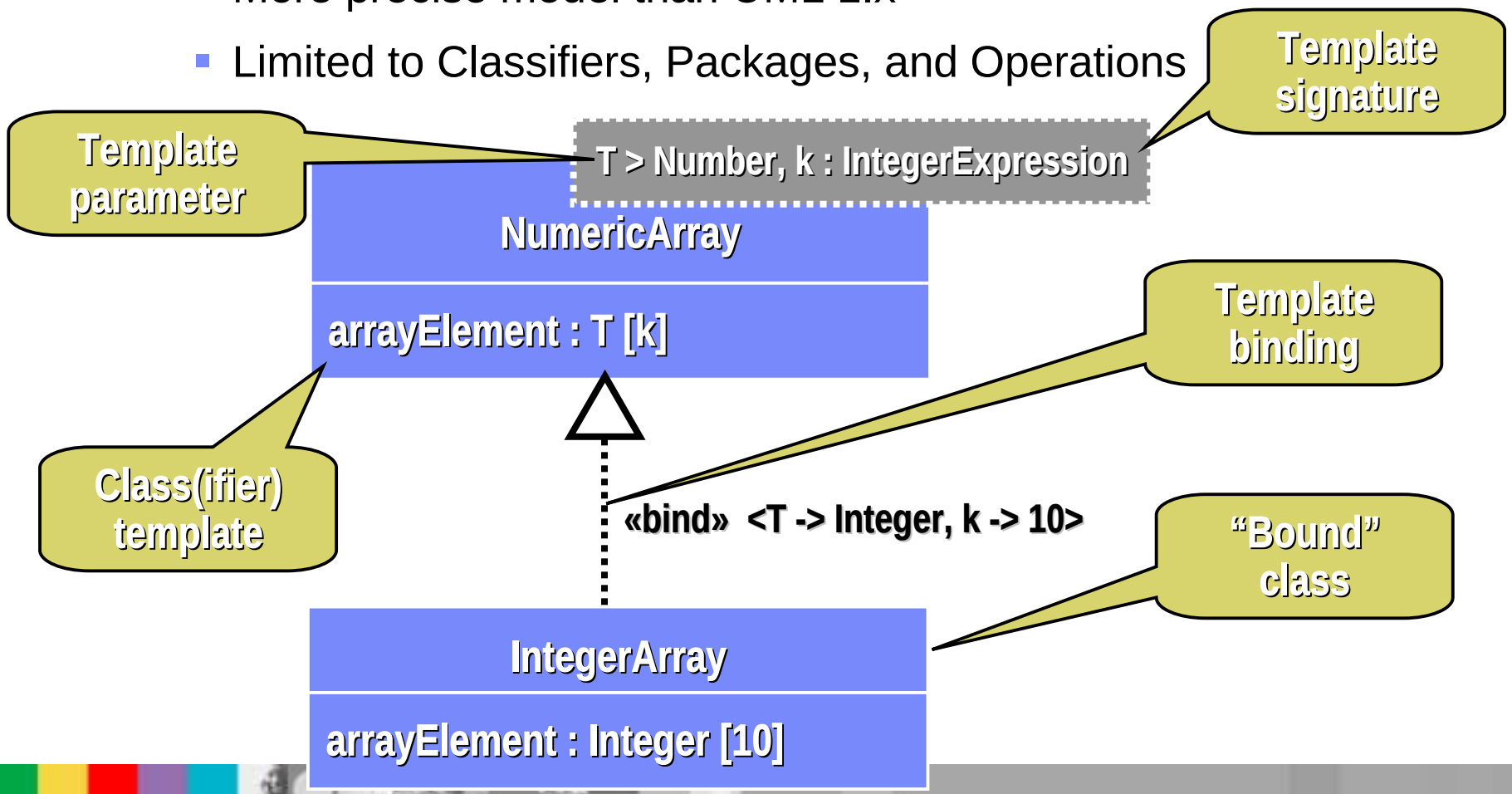
Profiles: Notation

- E.g., specializing the standard Component concept



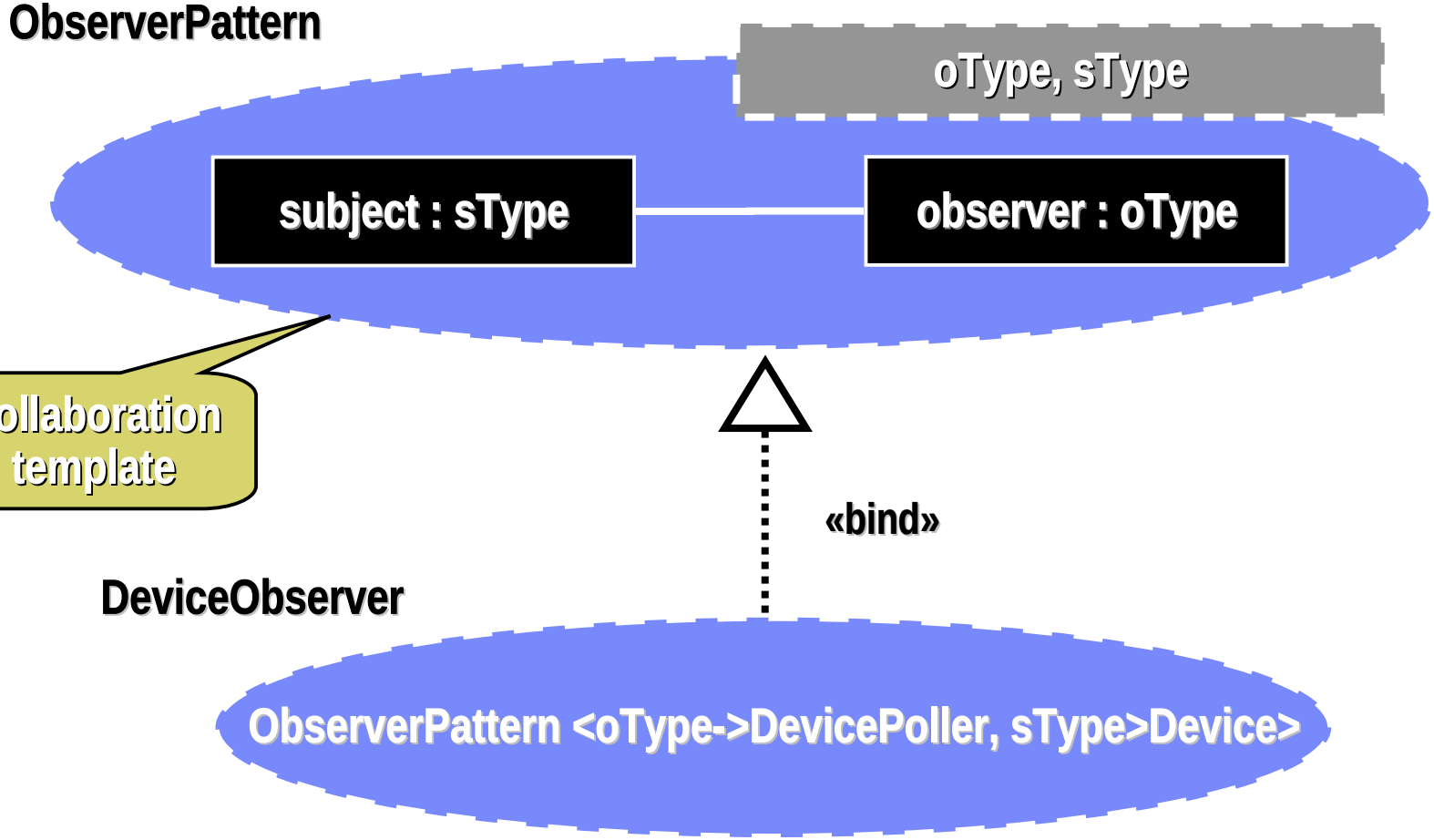
Templates

- More precise model than UML 1.x
- Limited to Classifiers, Packages, and Operations



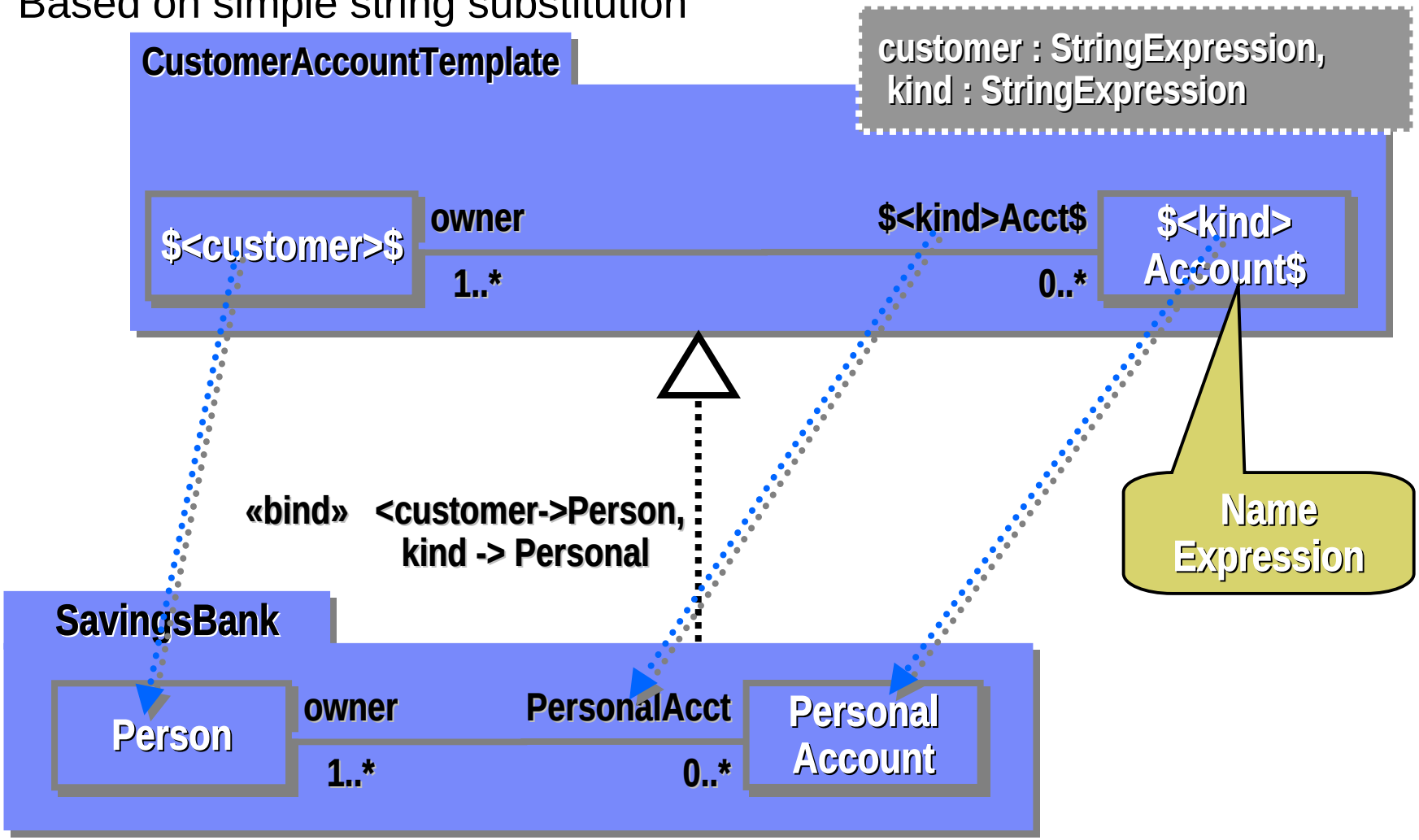
Collaboration Templates

- Useful for capturing design patterns



Package Templates

- Based on simple string substitution



Summary: UML 2.0 Highlights

- Greatly increased level of precision to better support MDD
  - More precise definition of concepts and their relationships
  - Extended and refined definition of semantics
- Improved language organization
  - Modularized structure
  - Simplified compliance model for easier interworking
- Improved support for modeling large-scale software systems
  - Modeling of complex software structures (architectural description language)
  - Modeling of complex end-to-end behavior
  - Modeling of distributed, concurrent process flows (e.g., business processes, complex signal processing flows)
- Improved support for defining domain-specific languages (DSLs)
- Consolidation and rationalization of existing concepts

Session Summary

- Now that you have completed this session, you should be able to:
  - Identify the different UML diagrams
  - Describe the purpose of each diagram
  - Use the different diagram and model elements


Questions


Thank
You