Unified Modeling Language

The Design Phase with UML2

Why Design?

- We saw the phases of S/W Engg
- Every software development process has the design phase
- For the *hotshot* programmers, this may not seem important
  - Fortunately those hotshots will never lead big projects
  - Unfortunately, most of the work done by those hotshots will be a footnote of history (e.g.: Netscape Navigator)
  - We are not hotshots ☺
- Simply put; good design → good software
- Good design also means
  - Faster development
  - Easier maintenance
  - Lesser reliance on a static team
  - Much easier to respond to requirements changes

History

- Man has always wanted to design
  - Proof → Pyramids (is anyone going to tell me they weren't designed before construction?)
- So have software developers
  - Sadly, here, the success-story is less prevalent
- Earlier methods for were quite primitive
  - Flowcharts (only algorithmic info)
  - Entity-relationship diagrams (only artifacts)
  - DFDs (good, but limited to showing dataflows through the system)

OOP? What?

- OOP = Object Oriented Programming
- OOD = Object Oriented Design
- Object?
  - Data
  - Operations
- Class?
  - Template or *type* of an object
  - int is a class, int a means that a is an *instance* of int
- You have all seen OOP (Java)
- System = Collection of cooperating objects
- Application design
  - Decompose system into objects that correspond to real-world objects
  - Figure out which interfaces they present to each other
  - Compose them together

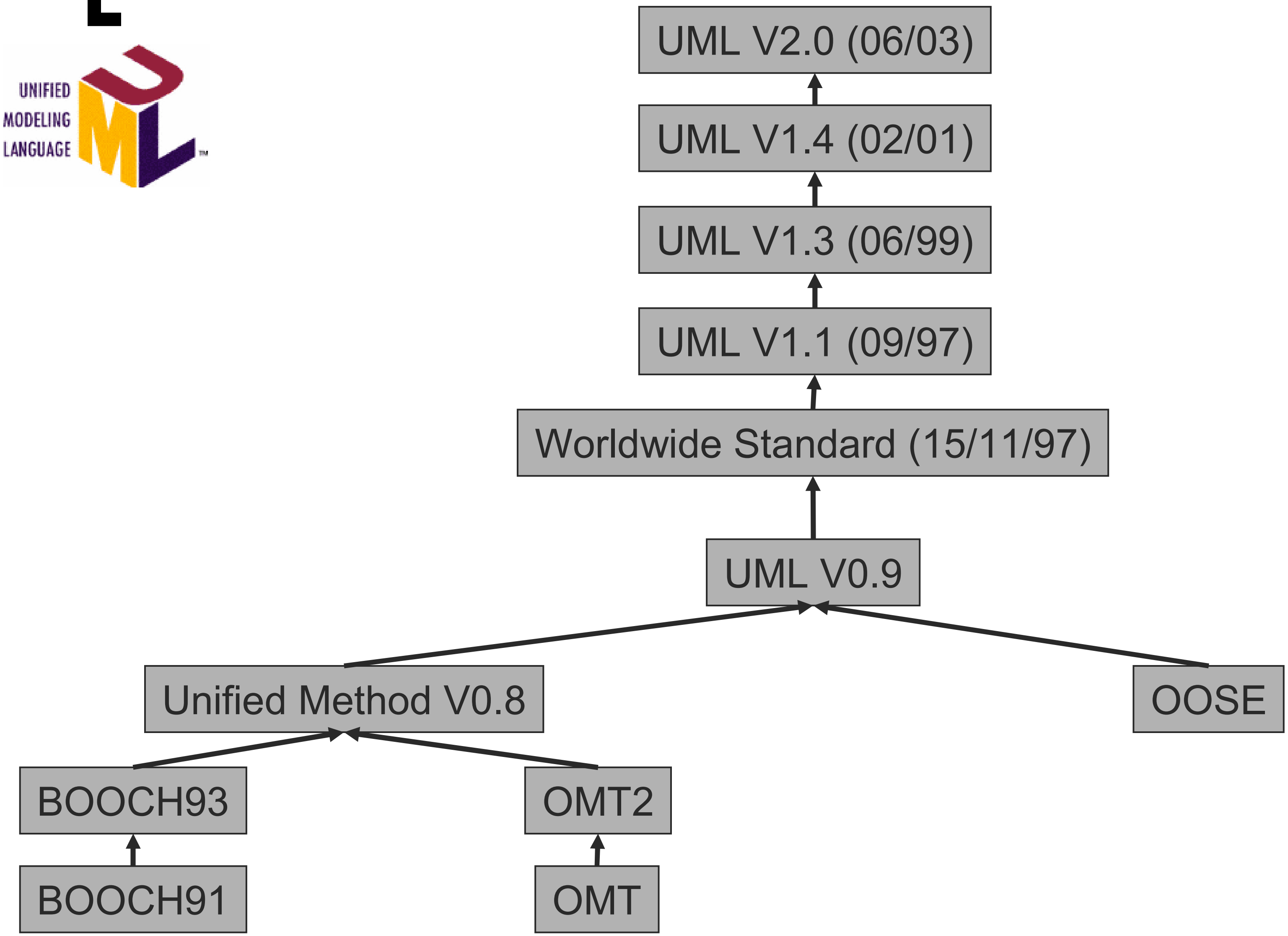
Early Days of OOP

- OOP was first introduced in mid-70s in a programming language called Smalltalk, since then many others have become available
  - Java
  - C++
  - Ada95
- Since then OOD has taken many forms
- In early to mid-90s there was OMT
  - Functional model
    - Showcases functionality from the user's point-of-view
  - Object model
    - Showcases the object topology of the system
  - Dynamic model
    - Showcases the behavioral aspects of the system

Early Days of OOP

- Object Oriented programming languages appeared early
  - 1973: Smalltalk
  - 1983: C++
  - 1989: Eiffel
  - 1995: Java, Ada95
- Object oriented analysis and design methods appeared later
  - 1987: Hierarchical Object Oriented Design (HOOD)
  - 1991: Object Oriented Analysis (OOA)
  - 1991: Object Modeling Technique (OMT)
  - 1991: Object Oriented Design (OOD)
  - Sept 1997: Unified Modeling Language (UML)
- 1989: Creation of the OMG (Object Modeling Group), which is a standards body which manages CORBA, UML, …, etc.

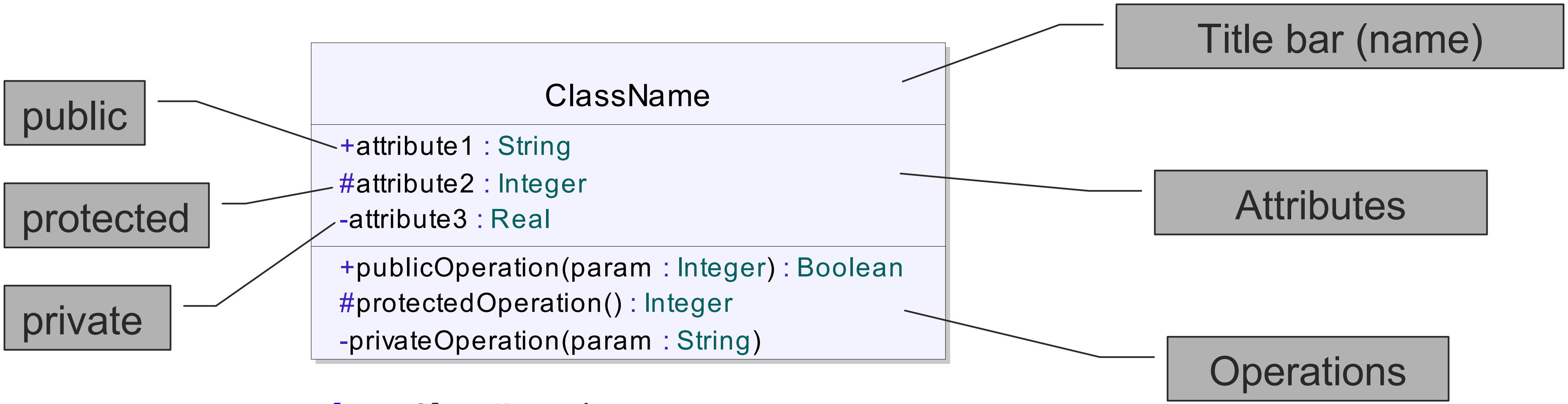
History of the UML



Objectives

- The designers of the UML had the following objectives
  - To represent systems by *object concepts*, such as classes and associations
  - To take into account the scale of complexity in large systems
  - To create a notation easily readable by humans, and manipulable by automated tools
  - To establish a coupling between the design and the execution of software

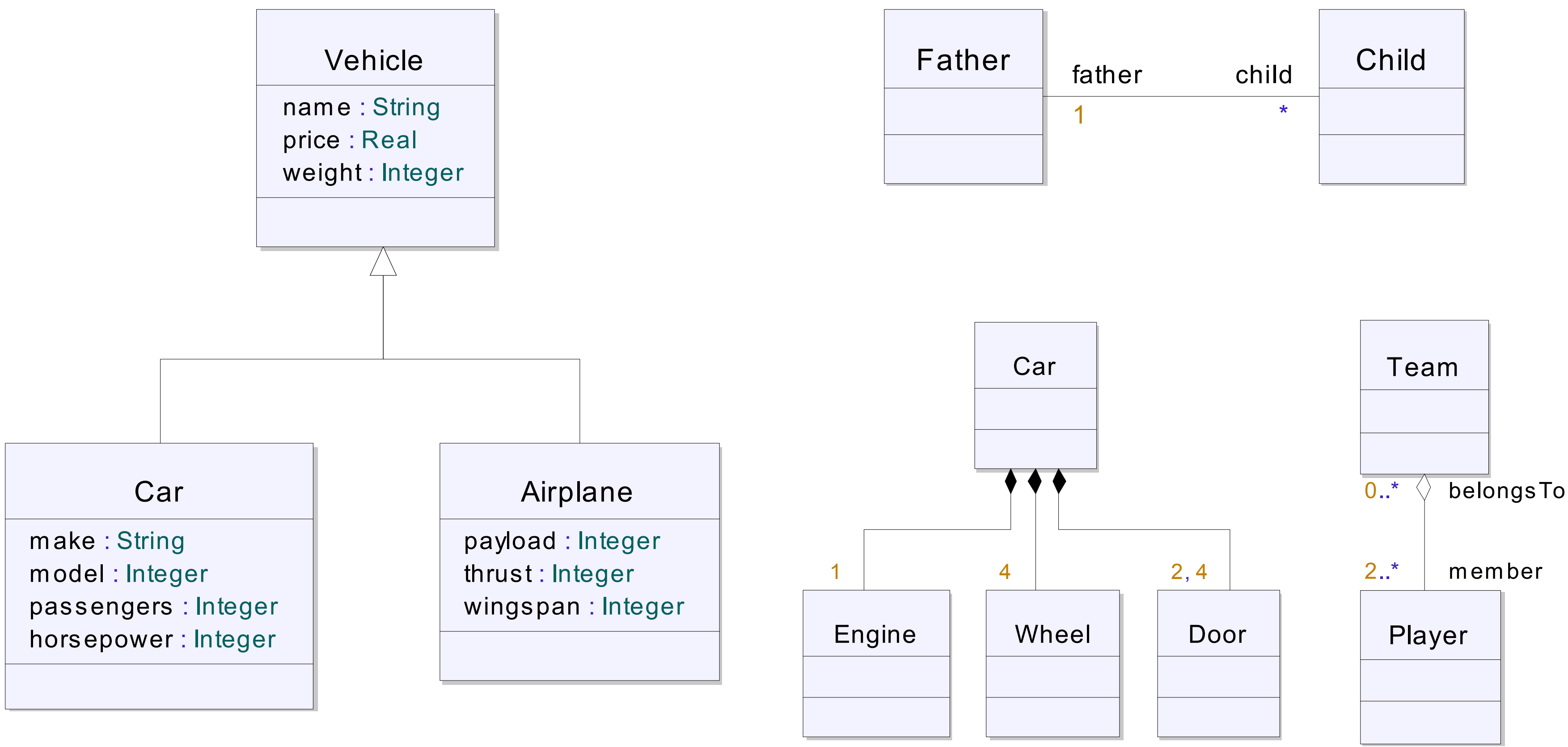
Class



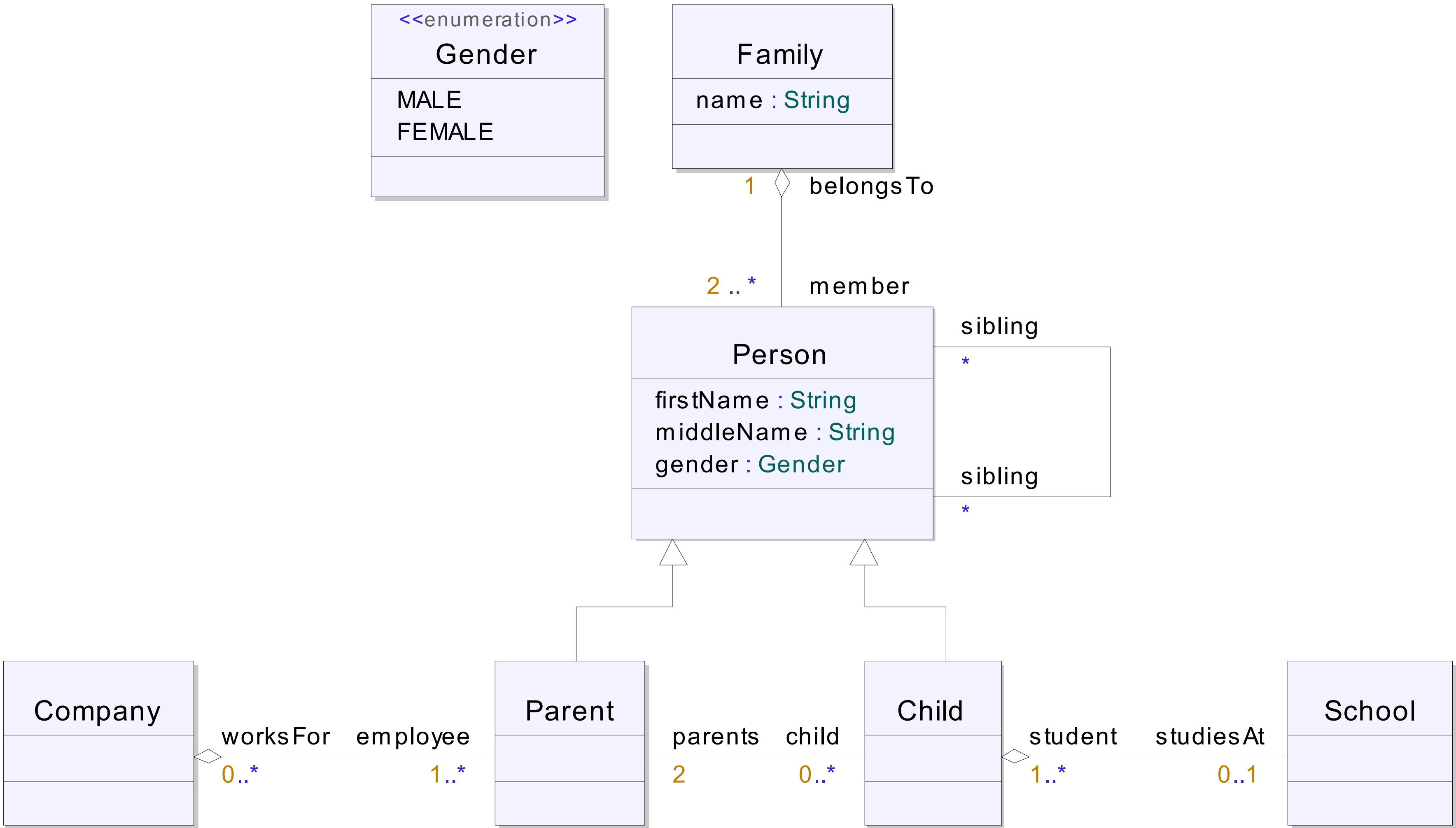

```
class ClassName {
public
    String attribute1;
protected
    Integer attribute2;
private
    Real attribute3;

public
    Boolean publicOperation(Integer param);
protected
    Integer protectedOperation();
private
    void privateOperation(String param);
};
```

Class Relations



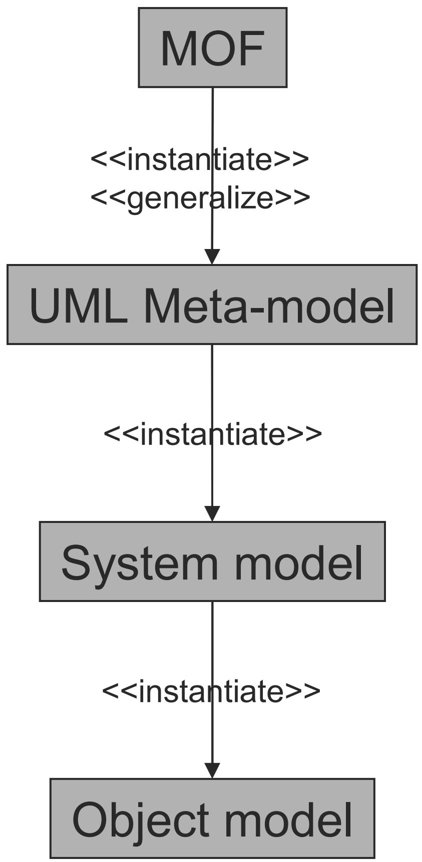
Class Relations



Semantics of UML

- The semantics of UML are defined semi-formally
- Legality rules are given through the UML *meta-model*
- Meta-model → The model of the model
- Everything in the UML meta-model is a class
  - Attributes are classes
  - Operations are classes
  - Associations are classes
- UML itself is defined in a 4 layer meta-model architecture
- Don’t worry if this sounds very complicated
- It is ☺
- UML has a semantic model (for architecture and for executable code) that maps well to most OO languages
- But does not *require* the use of a specific language

4-Layer Meta-model

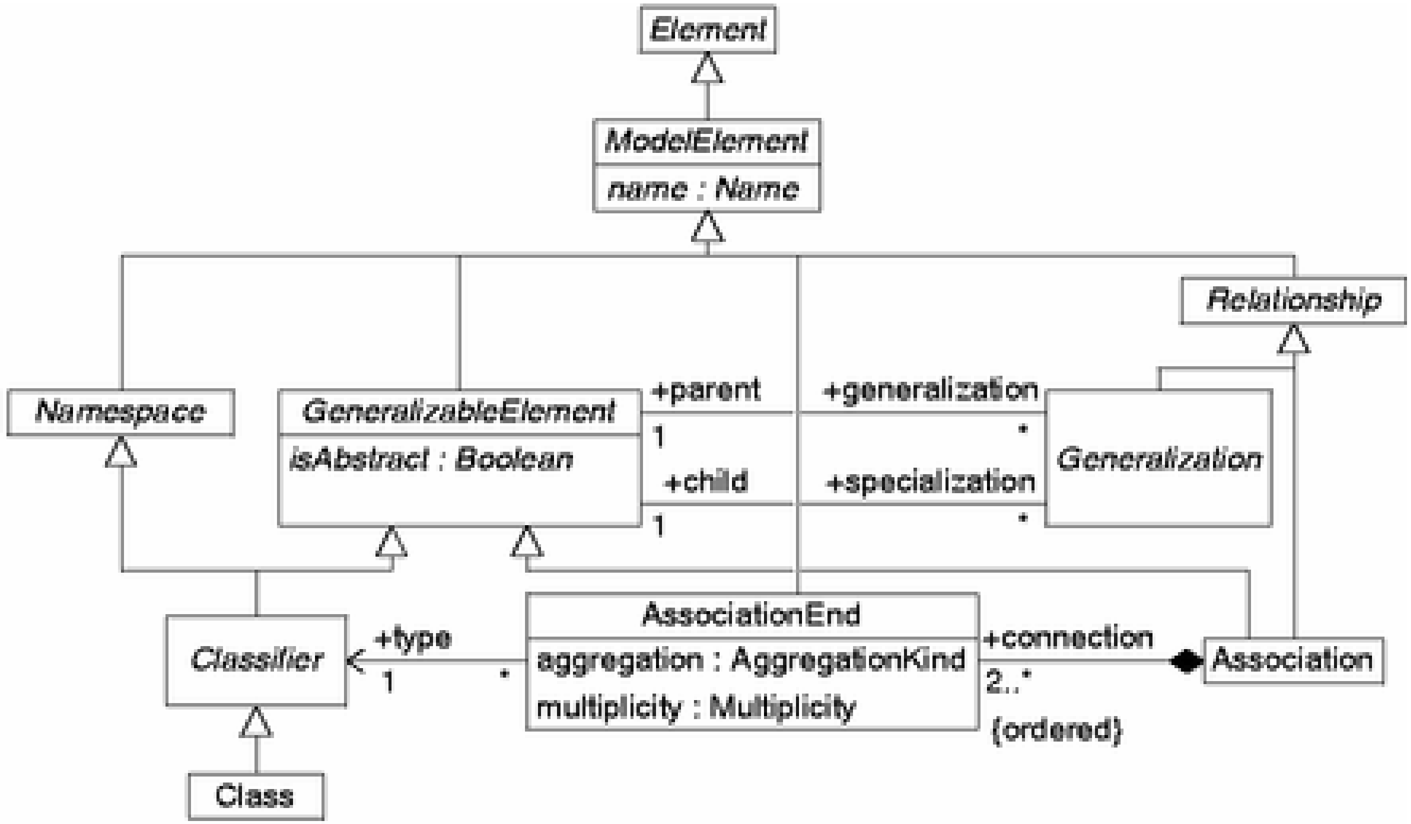


Meta meta-model	MOF	MetaAttribute, MetaClass
Meta-model	UML Specification	Class, Attribute, Operation, Component
System model	Project using UML	ATM, accountNumber, WithdrawTransaction( )
Object model	Running system	[Account:Name="John"; Account:Balance = 400;]

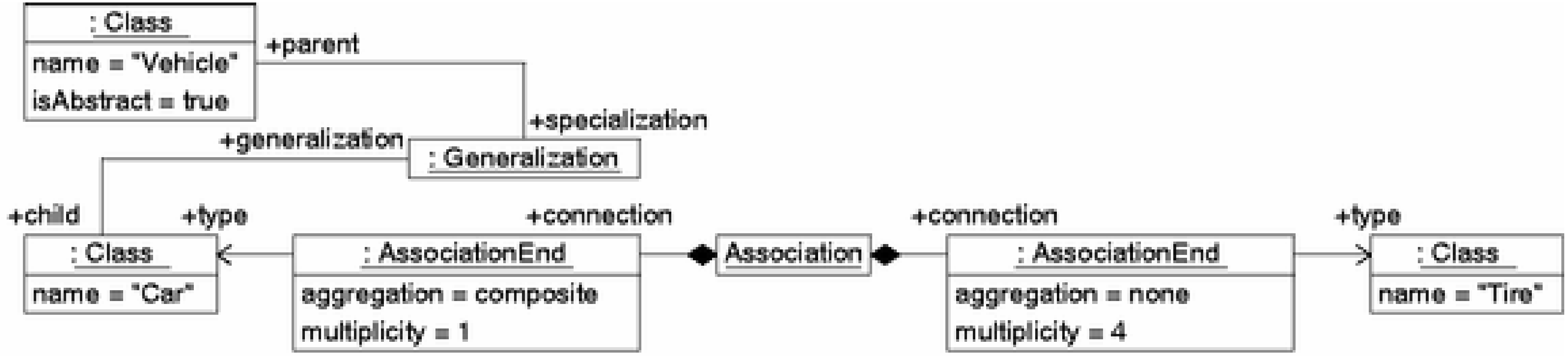
The Meta-model

- The meta-model levels (MOF as well as UML meta-model) are defined with the concepts and notions of classes
- In addition, the concepts and notions of classes themselves are defined in MOF (the elementary ones) and the UML meta-model (the advanced ones)
- So in other words, MOF *auto-defines* itself
- We also say that MOF *bootstraps* itself

Meta-model Example



Meta-model → Model



UML V2.0 Diagrams

- Structural modeling diagrams
  - Package diagrams
  - Class diagrams
  - Object diagrams
  - Composite structure diagrams
  - Component diagrams
  - Deployment diagrams
- Behavioral modeling diagrams
  - Use case diagrams
  - Activity diagrams
  - State-machine diagrams
  - Communication diagrams
  - Sequence diagrams
  - Timing diagrams
  - Interaction overview diagrams

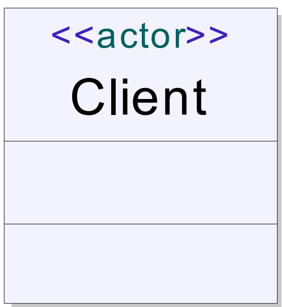
UML V2.0 Diagrams

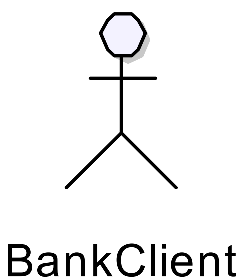
- Each stage of the software development process is accomplished using one or more diagrams
- So a diagram represents a *view* on a certain part of the system
- A diagram is based on certain concepts
  - The class diagram is based on classes, associations
  - The state-chart diagram is based on states, regions, transitions
  - The composite structure diagram is based on components, ports and interfaces
- A diagram has certain construction rules, these rules define what is allowed in a certain diagram and what is not
- These rules are given in the UML V2.0 meta-model

Use-case Diagrams

- Represent the behavior of a system from the point-of-view of the user
- System utilization specification → represent the total usage scenarios
- Very useful in the analysis phase to capture principal entities and flows of use
- A system should be described by at most a dozen use-cases (moderate sized system)
- Exceptions and errors should not be a separate use-case
- Correspond to a functional decomposition of the system, but always from a user perspective

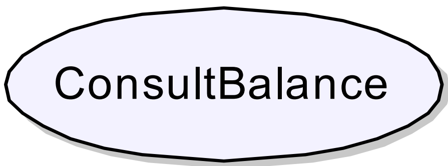
Use-case Diagrams (Actors)





- Class stereotyped <<actor>>
- Represented as a scarecrow in a use-case diagram
- Is the external element interacting with the system
- Can be
  - Principal actor (client of bank)
  - Secondary actor (info system of bank)
- Legality rule
  - Every actor *must* communicate with the system

Use-case Diagrams (Use-case



- A set of scenarios (nominal and non-nominal)
- Legality rule
  - A use-case must represent a unit of micro or macro functionality of the system
- No stereotype

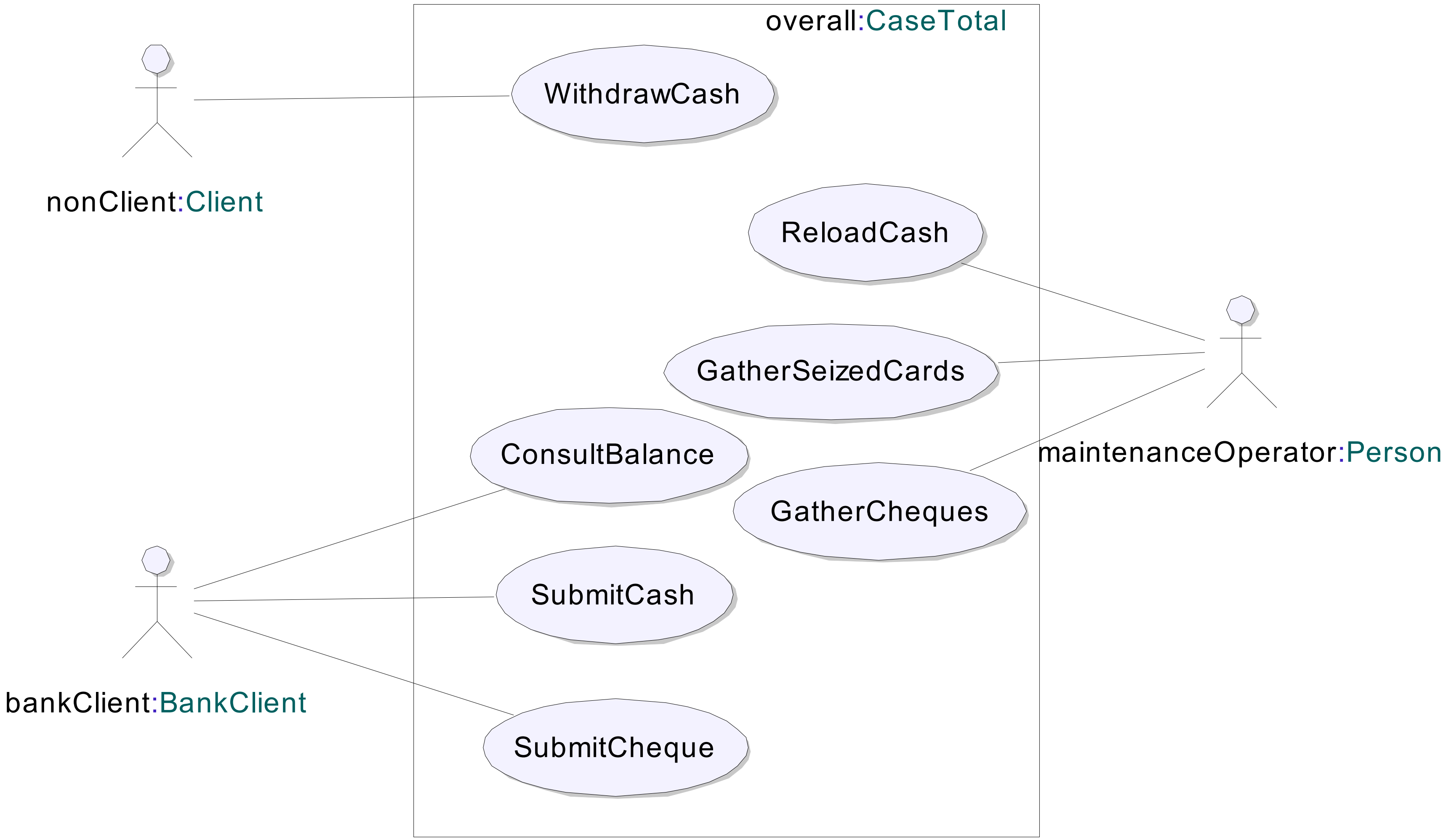
Use-case Diagrams

- Links between actors and use-cases represent communications
  - Each use-case represents a sequence of exchanged messages
  - Usually this corresponds to a *transaction*
- The communication can be
  - Directional or non-directional
  - Tagged (for additional info) or not
- May represent a continuous data-flow

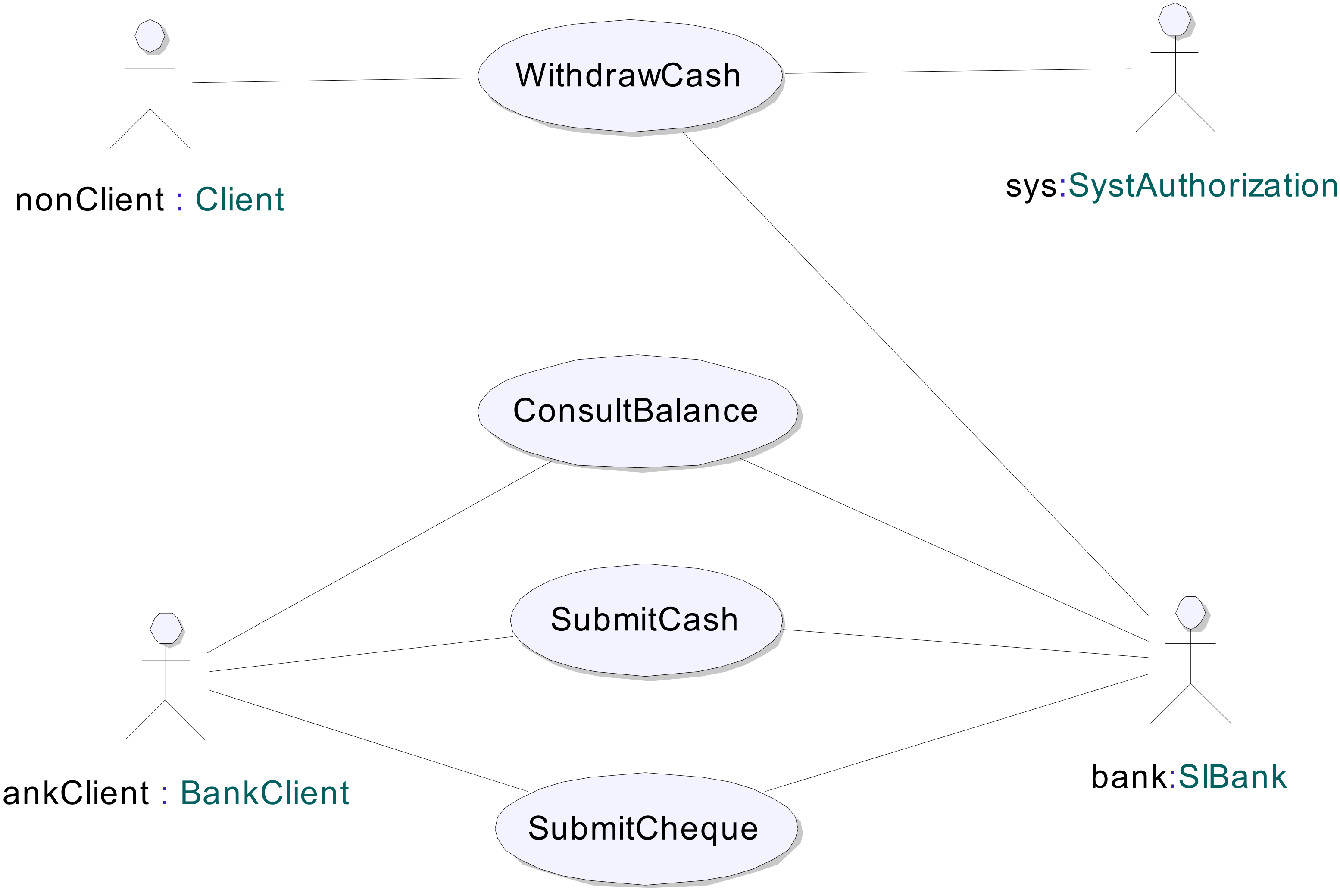
Capturing Requirements

- We will consider the famous ATM example
- The ATM offers the following service
  - Distribution of money to holders of credit cards, using a card reader and a bill distributer
  - Consultation of the account balance, submission of cash and submission of cheques for clients that have a credit card of the same bank as the ATM
- Other requirements
  - Transactions are secure
  - The distributer must be refilled from time to time

Overall Use-case



Secondary Actors



Text Description of Use-cases

- Title: WithdrawCash
- Summary: Someone with a credit card who is not a client of this bank can withdraw cash if his credit allows
- Description of scenarios
  - He has credit
  - He doesn’t have credit
- Non-functional requirements
  - Maximum amount of time taken
- Requirements of the HCI
  - …

Text Description of Use-cases

- Description of scenarios
  - Preconditions
    - The ATM should have money
    - There should be no card in the card reader
    - No transaction should currently be under way
  - Nominal scenario
    - …
  - Alternative scenarios
    - …
  - Error chains
    - …
  - Postconditions
    - The user should have his cash
    - His account should be debited
    - Card should be ejected

Text Description of Use-cases

- Nominal scenario
  1. The card holder puts his card into the reader
  2. The ATM verifies that it is a credit card
  3. The ATM demands the authorization code
  4. The card holder enters code
  5. The ATM compares code with that on chip
  6. The card holder enters amount required
  7. The ATM consults the SystAuthorization
  8. The SystAuthorization answers with decision
- Alternative scenarios
  1. A1: The code is provisionally incorrect => NS4
  2. A2: Amount requested > balance => NS8
- Error chains
  1. E1: Invalid card
  2. E2: Invalid code after 3rd try
  3. E3: Withdrawal not allowed
  4. …
- Postconditions
  1. Cash ejected = cash deficit in distributer
  2. Card is ejected
  3. Receipt is printed and ejected

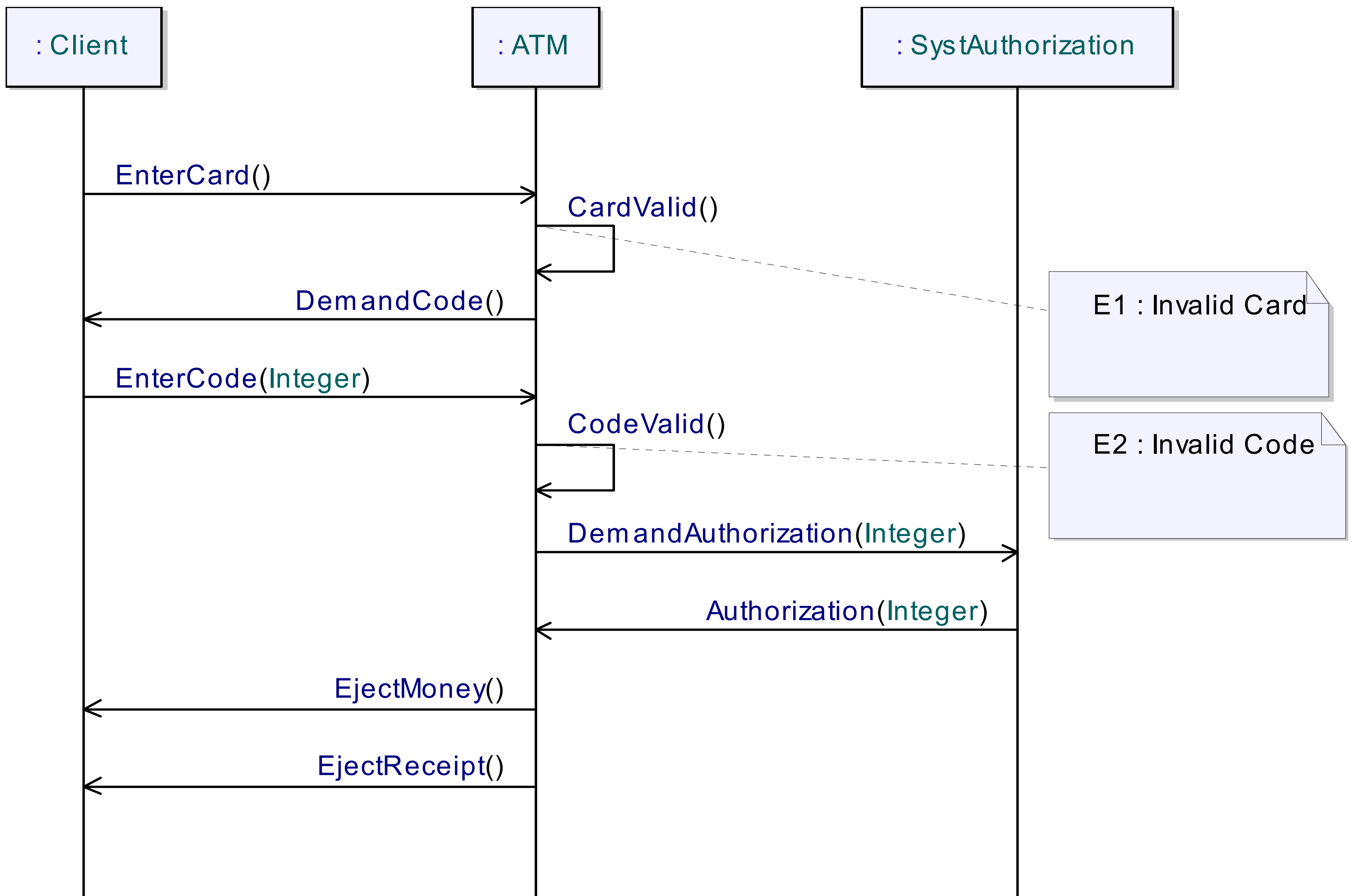
Sequence Diagrams

- Show the exact sequence of messages exchanged between different entities of a system model
- Objects
  - Dedicated object: A named object of a certain class of the system
  - Anonymous object: Any object of a class of the system
- Supports data flow and different types of synchronization
- The horizontal axis represents a conceptual or function difference (different objects of the system)
- The vertical axis represents an increase of time (time increases as we go down)
- Horizontal lines represent exchanged signals or operations

Sequence Diagrams

- Show the interaction between objects in a temporal sequence
  - Objects participating in an interaction are given along with their *swimline*
  - The messages exchanged between them are shown as horizontal lines between the vertical swimlines
- We can describe all the message sequences possible for an object
- One sequence diagram per use-case (usually)
- There are two kinds of interactions
  - Asynchronous: Signal transmission and reception
  - Synchronous: Operation calls on objects

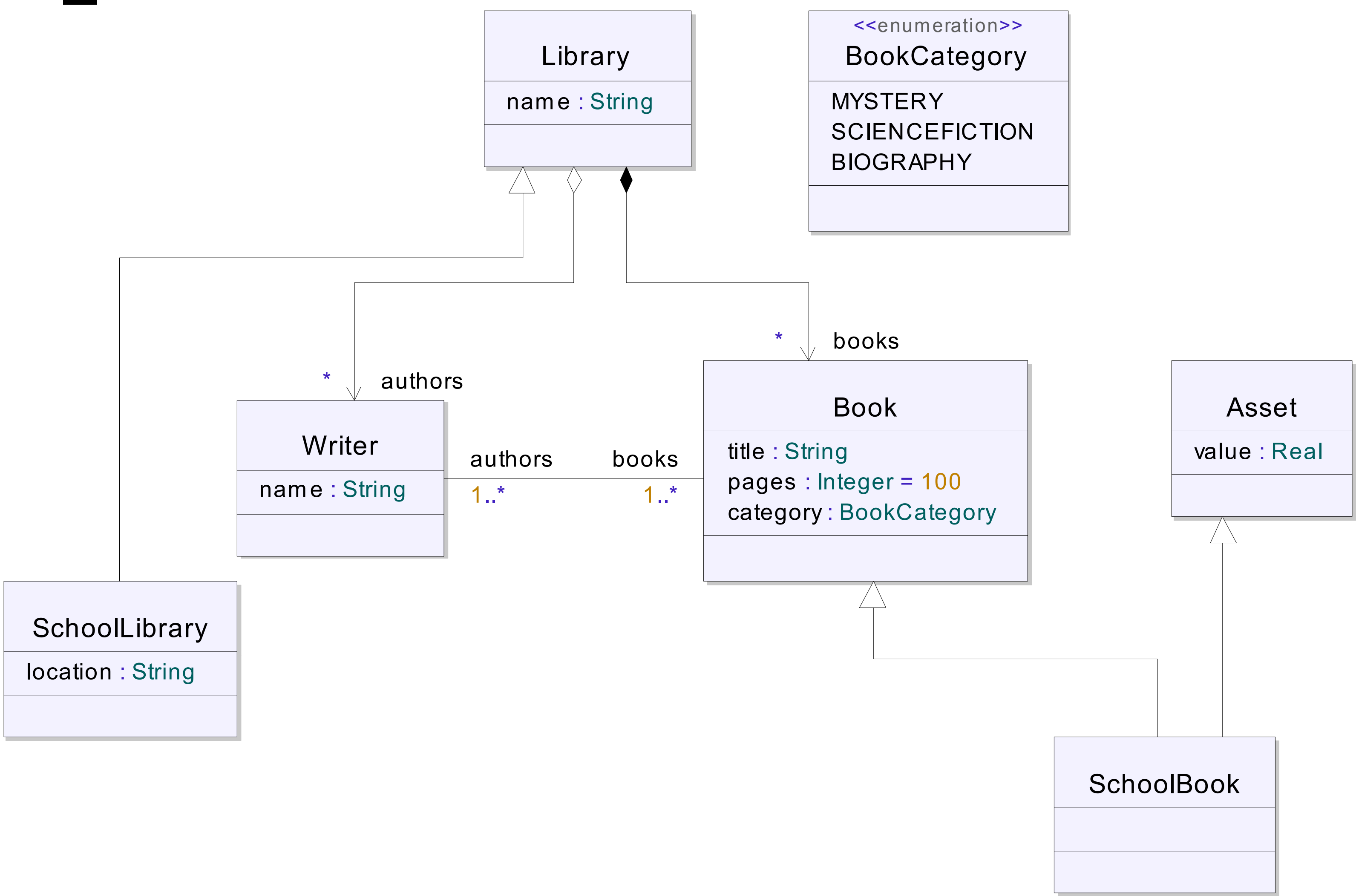
Sequence Diagram



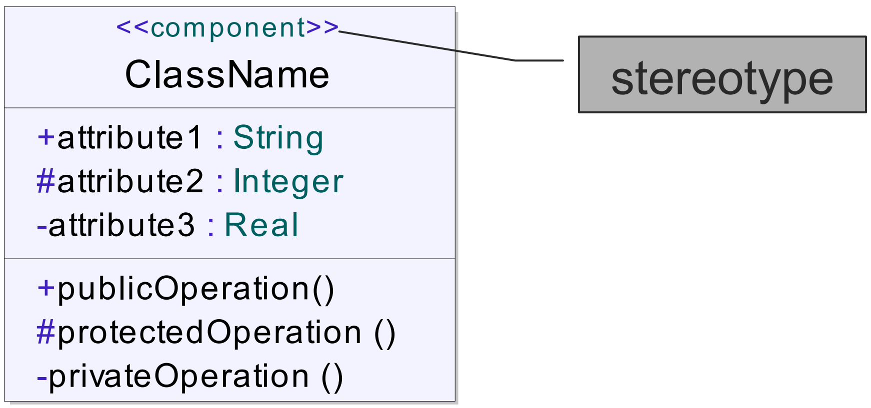
Class Diagrams

- Show classes and their structure
- Show associations between classes
- Show signal definitions
- Show ports and interfaces on classes
- Show inheritance relations
- We saw class diagrams in the beginning
    - The family domain model class diagram
    - The vehicle, car, and airplane class diagram

Class Diagrams



Class Diagrams

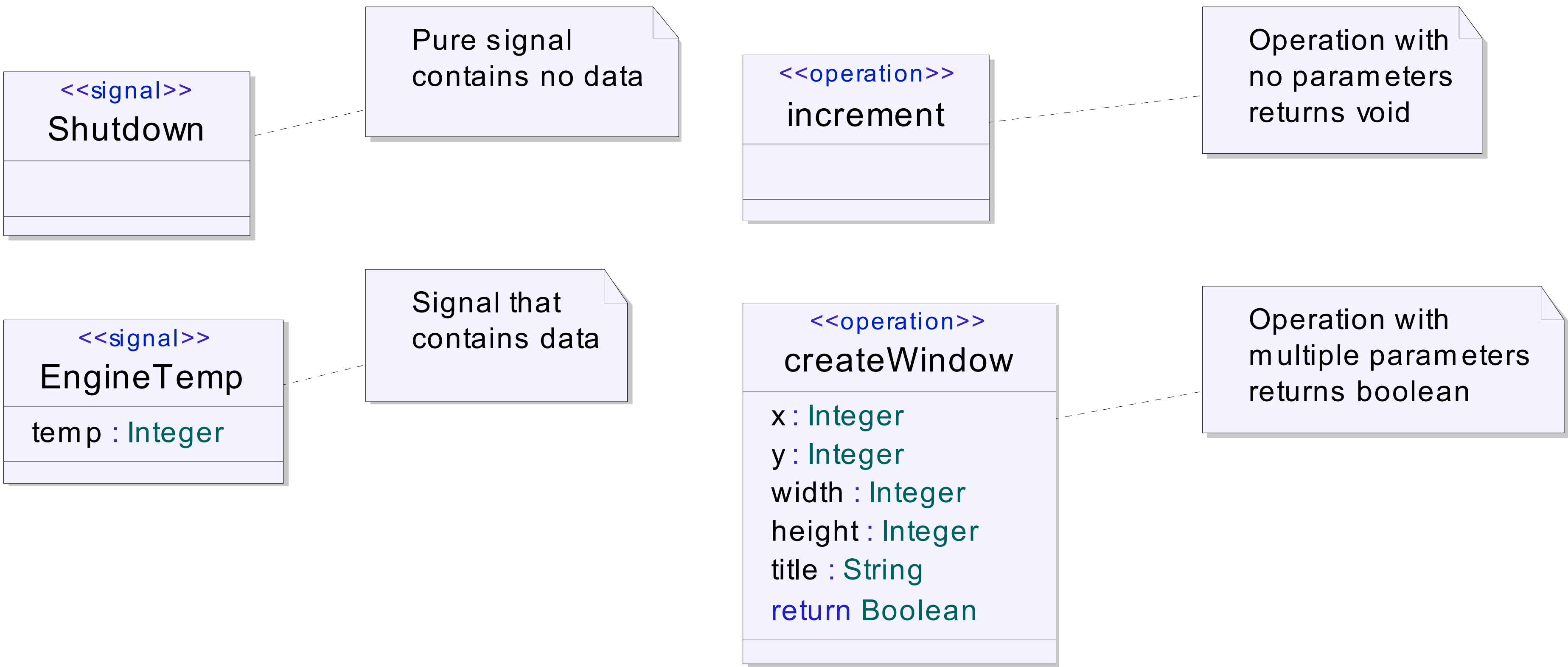


- A stereotype adds extra information to a model element
- May be used for
  - Identifying distinct types of artifacts
  - Adding meta-attributes to the UML meta-model

Signals and Operations

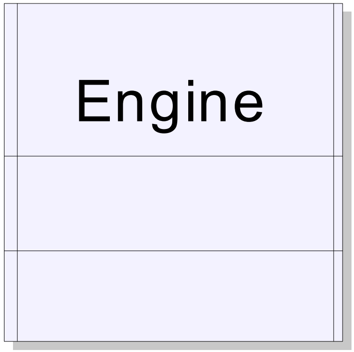
- UML provides for two basic kinds of interaction between objects
  - Operation calls (synchronous)
  - Signal transmission (asynchronous)
- Operation calls
  - Like member function calls in Java/C++
  - Have a wait-for-return semantic
  - The target class *receives* the invoking class' thread
- Signal transmission
  - Asynchronous transmission of message
  - Sender continues its own processing after signal transmission
  - The target class does its processing in its own thread
- Presentation
  - Signals are represented as a class box with stereotype <<signal>>
  - Operations are represented as a class box with stereotype <<operation>>
- Both signals and operations may have parameters, they are given in the attributes section of the class box

Signals and Operations



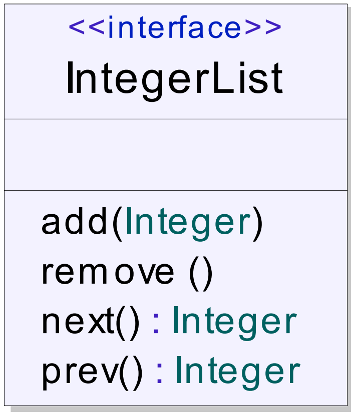
Active Classes

- An active class in UML is one that
  - Starts execution of its behavior as soon as an object of it is created
  - Does not cease until either
    - The behavior defined for it is completed
    - It is terminated by another object
  - So it is also referred to as having its own *thread of control*
- The points at which an object of an active class responds to communication is determined solely by its behavior and not by the invoking object
- Presentation
  - An active class is shown by a class box with additional vertical bars on the sides



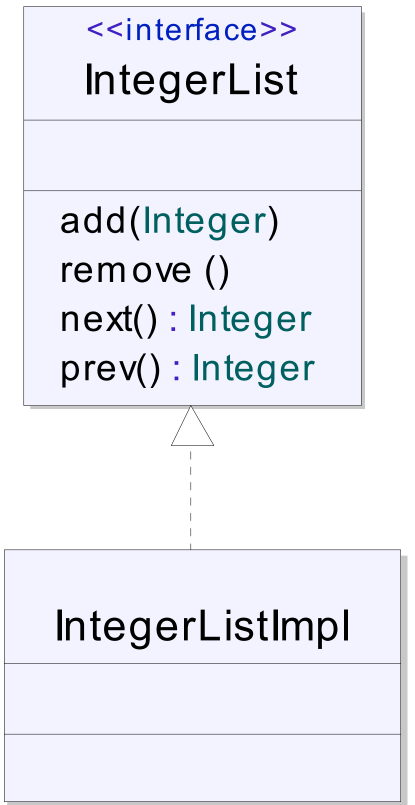
Interfaces

- An interface represents a declaration of a set of public features and obligations (ref. Java interfaces)
- Interfaces are not instantiable
- An interface is *implemented* by an instance of a class (an object)
- A given class may implement a number of interfaces
- An interface may be implemented by a number of classes
- Presentation
  - Class boxes in a class diagram
  - With stereotype <<interface>>
  - Can only contain public operations or signals



Realizing an Interface

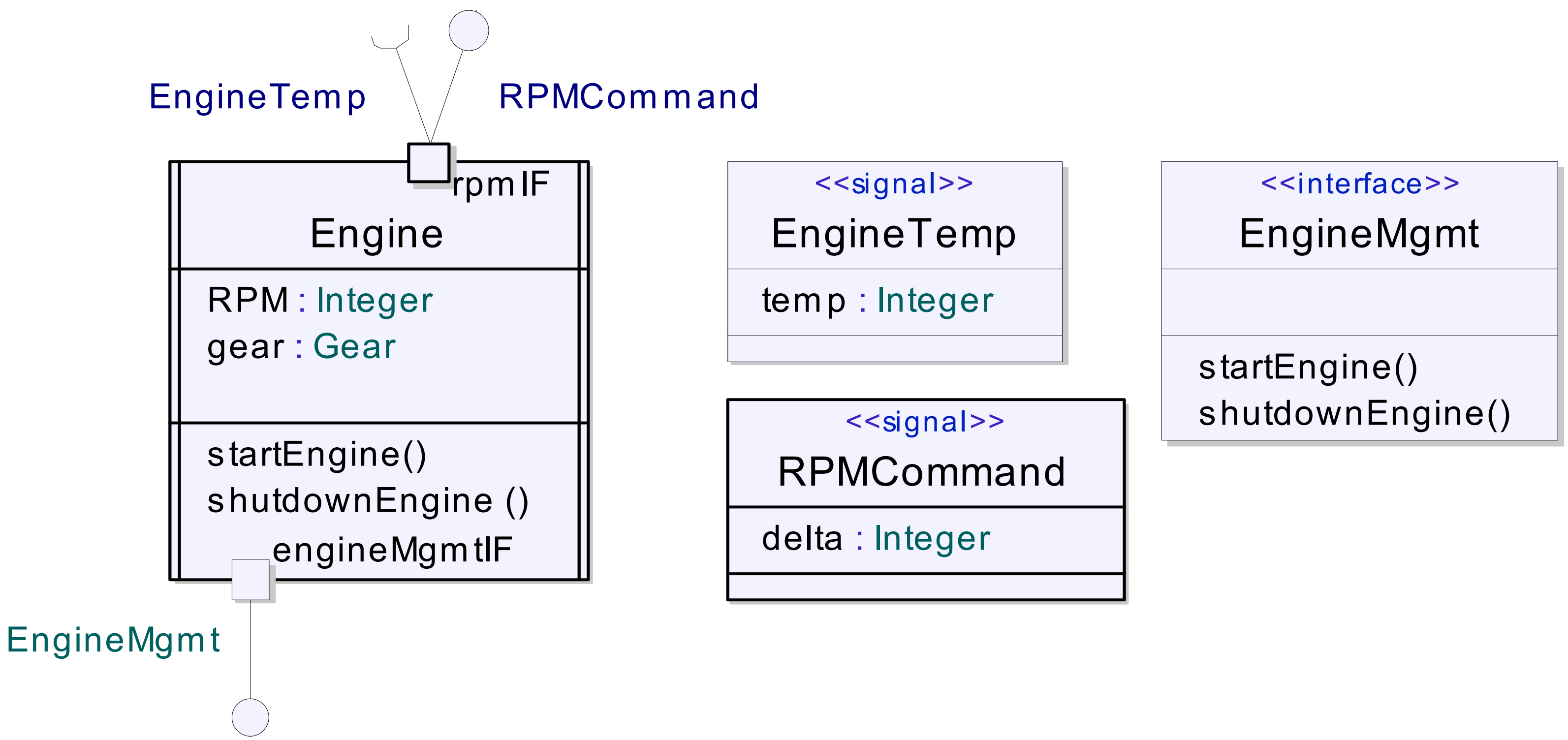
- An implementation class (normal class) can inherit from an interface
- In this case we say that the implementation class realizes the interface



Ports

- A port is a distinct interaction point
  - Between an object and its environment
  - Between an object and its internal parts
- Ports are connected to other ports through *connectors* through which requests can be made to invoke the behavioral features of a class
- A port may specify
  - The services an object provides (its provided interface)
  - The services an object expects (its requested interface)
- Presentation
  - Ports are shown as small solid rectangles on the boundary of classes
  - Provided interfaces are shown as a line extending out of the port rectangle and terminating with a solid circle
  - Required interfaces are shown similarly but with a crescent

Ports


EngineTemp
RPMCommand
rpmIF
Engine
RPM : Integer
gear : Gear
startEngine()
shutdownEngine ()
engineMgmtIF
EngineMgmt
<<signal>>
EngineTemp
temp : Integer
<<interface>>
EngineMgmt
startEngine()
shutdownEngine()
<<signal>>
RPMCommand
delta : Integer