Architecture of Digital Systems II

Appendix

A Formalizing System Design With UML

Why Formal System Descriptions?

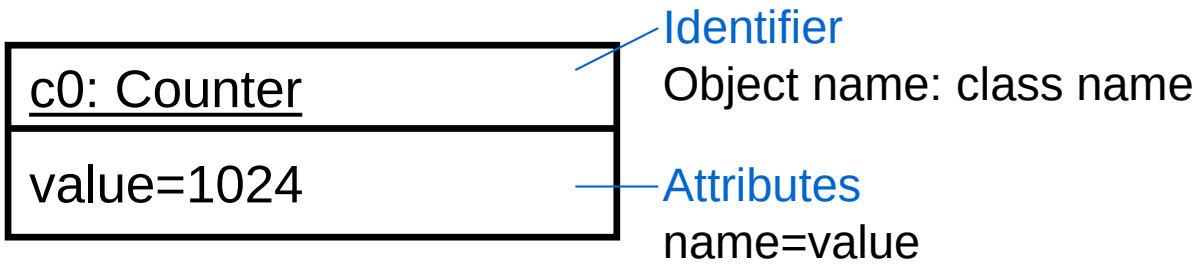
System design is a complex procedure involving different design tasks at different levels of abstraction: creating requirements and specifications, creating an architecture, implementing hardware and software and creating tests. For the lower-level tasks, formal specification languages such as programming languages and hardware description languages allow to precisely define the system's structure and behavior. Automatic tools such as compilers for SW and synthesis tools for HW use these descriptions to generate the hardware and software on the bit level.

On higher levels of abstraction, formal languages help to unambiguously describe and communicate system structure and behavior between developers, customers, designers and manufacturers. Examples for higher-level formal system description languages are SDL, Petri nets, Statecharts, UML.
In the following, we take a (superficial) look at UML.

Unified Modeling Language (UML)

The Unified Modeling Language (UML) is an object-oriented, general-purpose, graphical language for modeling systems. It is maintained and standardized by the *Object Management Group* (a consortium of over 800 companies from the IT and other branches). With respect to object orientation, it is similar to a programming language. With respect to describing structure, it is similar to hardware description languages like VHDL.

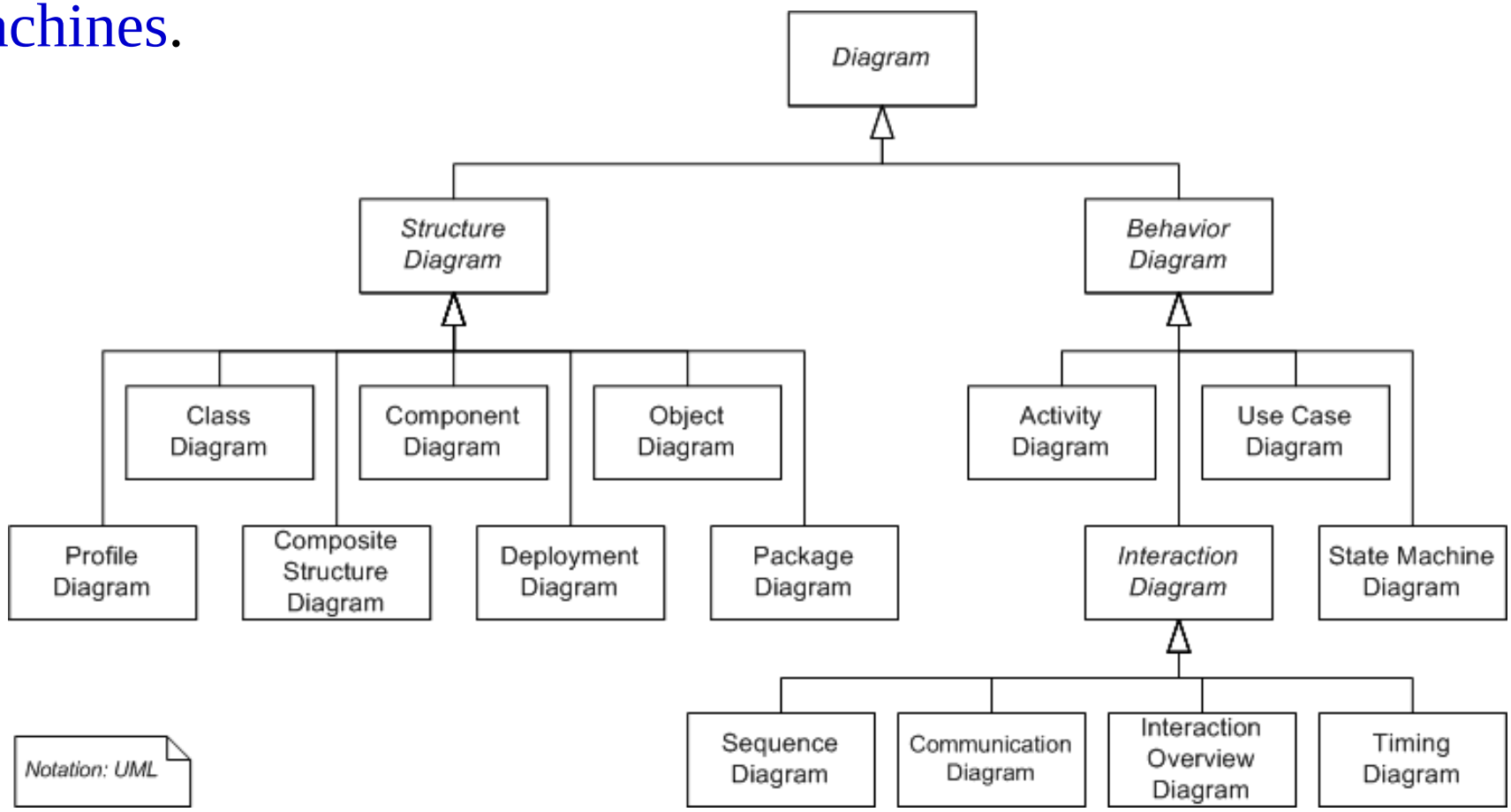
The basic element of object-oriented design is the object. An object includes a set of attributes defining its internal state.



An object without name is called *anonymous object*

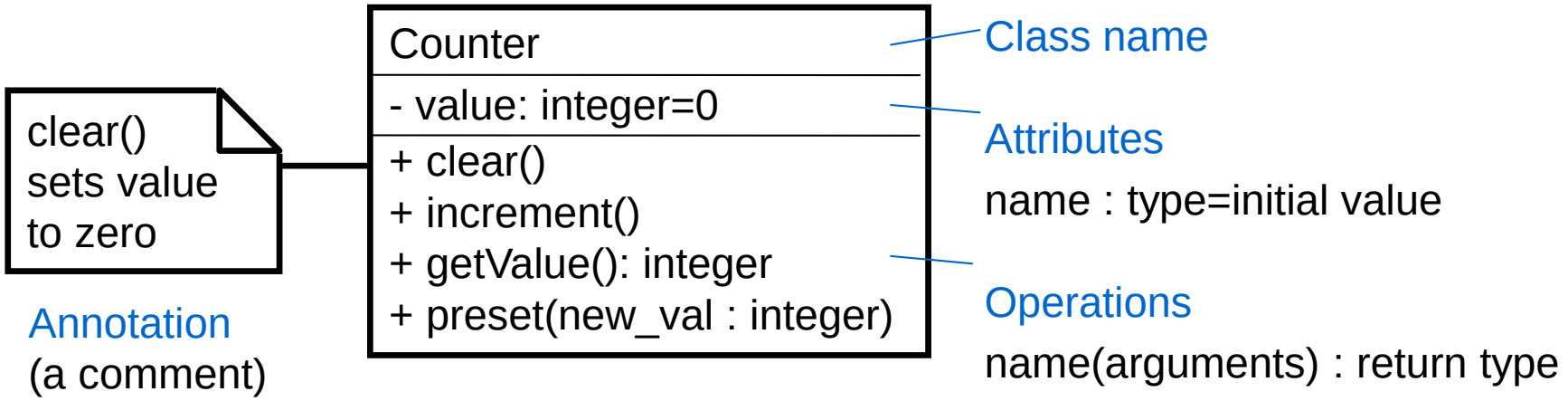
UML Diagrams

UML 2.2 has 14 types of diagrams for modeling structure and behavior of systems.
As an example of a structure diagram, we will look at classes.
As an example of a behavior diagram, we will look at state machines.



UML Structural Diagram Example: Class

A class serves as a type definition for objects. All objects of the same class have the same characteristics, however, each object has its own internal state (attribute values).
The class defines the operations that an object may perform to interact with its environment.
Attributes marked with '+' are visible from the outside. Attributes marked with '-' can only be accessed class-internally.



Relationships between Classes

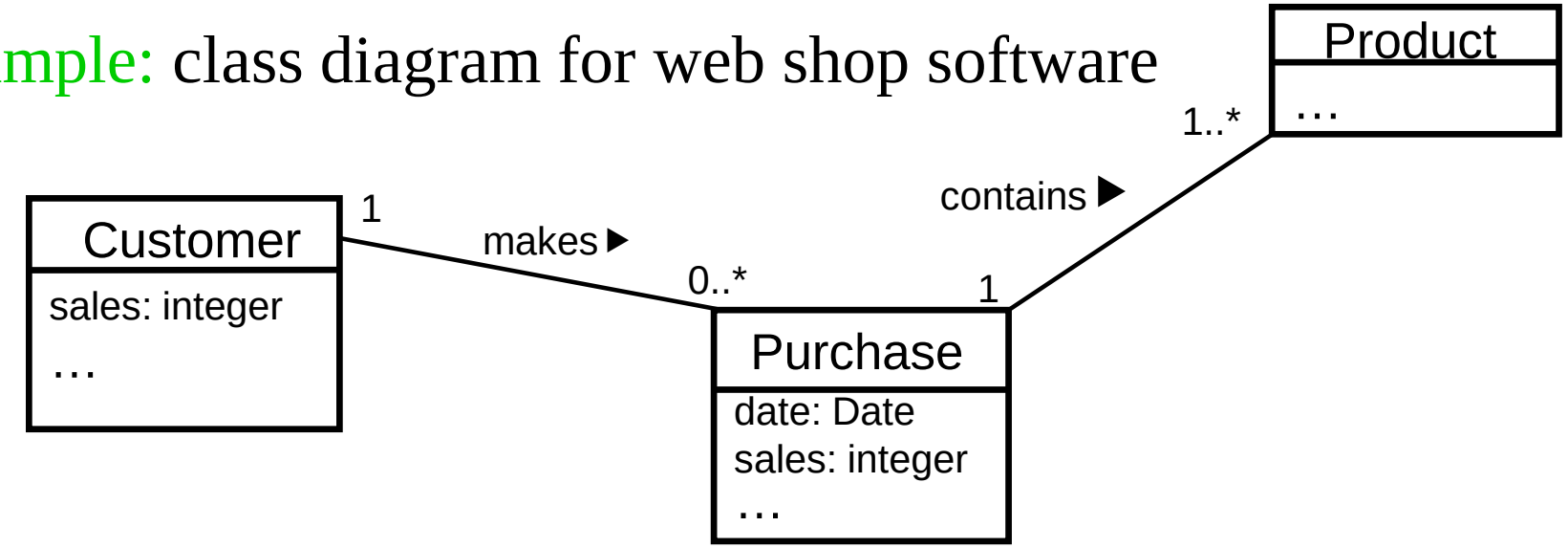
In UML, we can model several types of relationships between classes:

- Association is a simple relationship between objects that expresses communication but not ownership.
- Generalization allows to define one class in terms of another (inheritance, derived classes).
- Aggregation describes how an object is made of smaller objects.
  - Composition is aggregation in which access to the component objects is prohibited.

Association

Association models a relation between classes. It can be attributed with cardinality constraints and a label defining the relation between objects of the classes.

Example: class diagram for web shop software

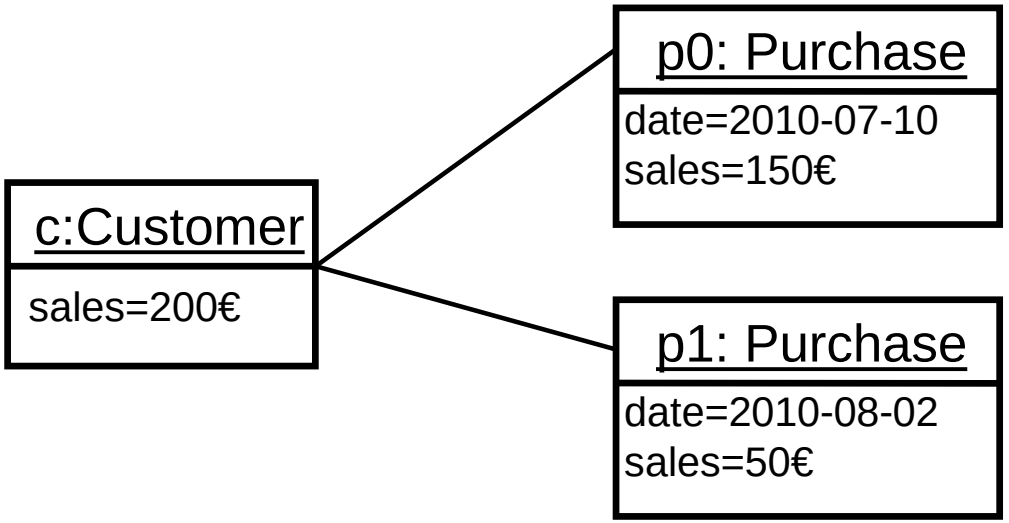


- Every time a customer buys something a Purchase object is created
- A customer may visit the shop several times (or never)
- A customer may buy several products at once

Links

A link models a relationship between actual objects. "Link" is to "association" as "object" is to "class".

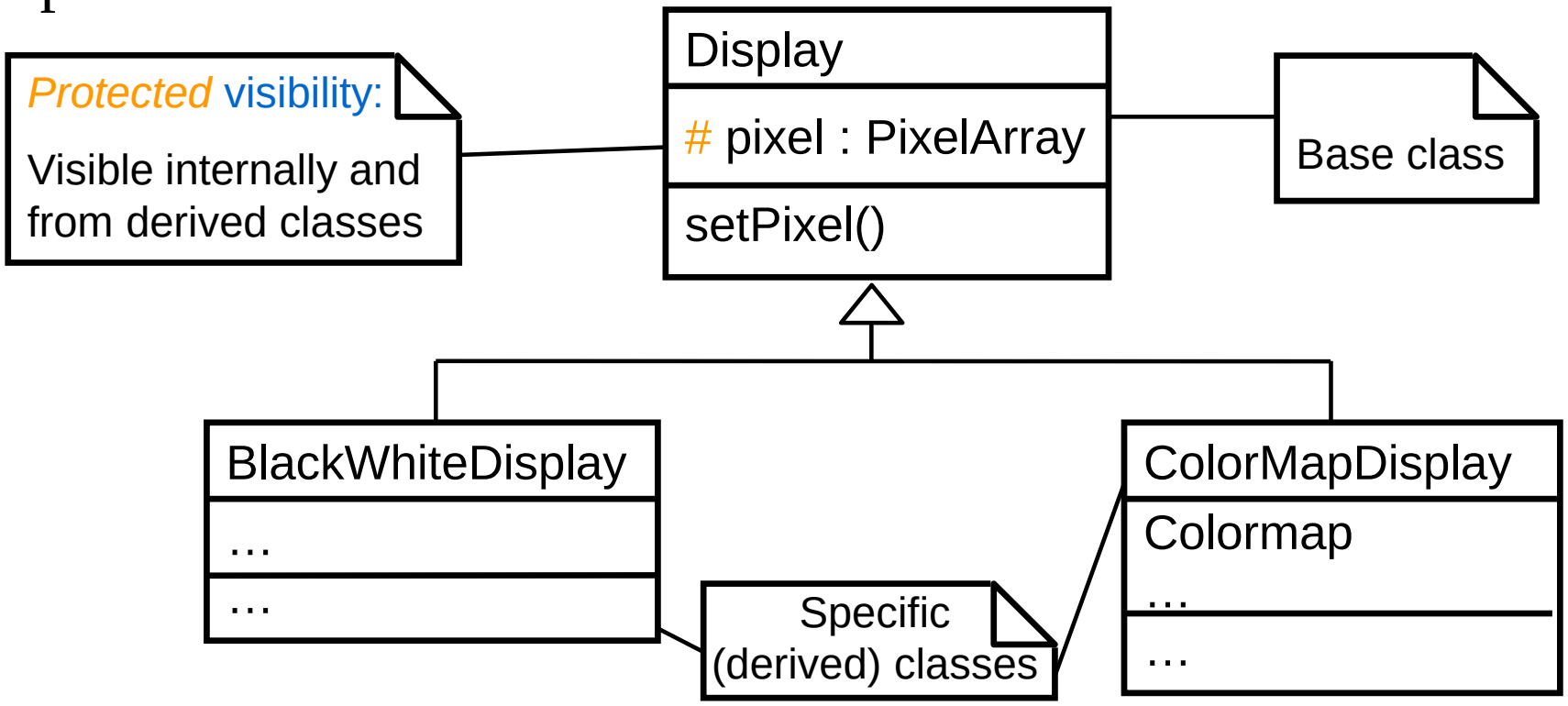
Example: object diagram for web shop software



Generalization

Generalization allows to define one class in terms of another class. Inheritance is a form of generalization. A derived class inherits all the attributes and operations from its base class.
Generalization is symbolized by an arrow with a hollow triangle-shaped head.

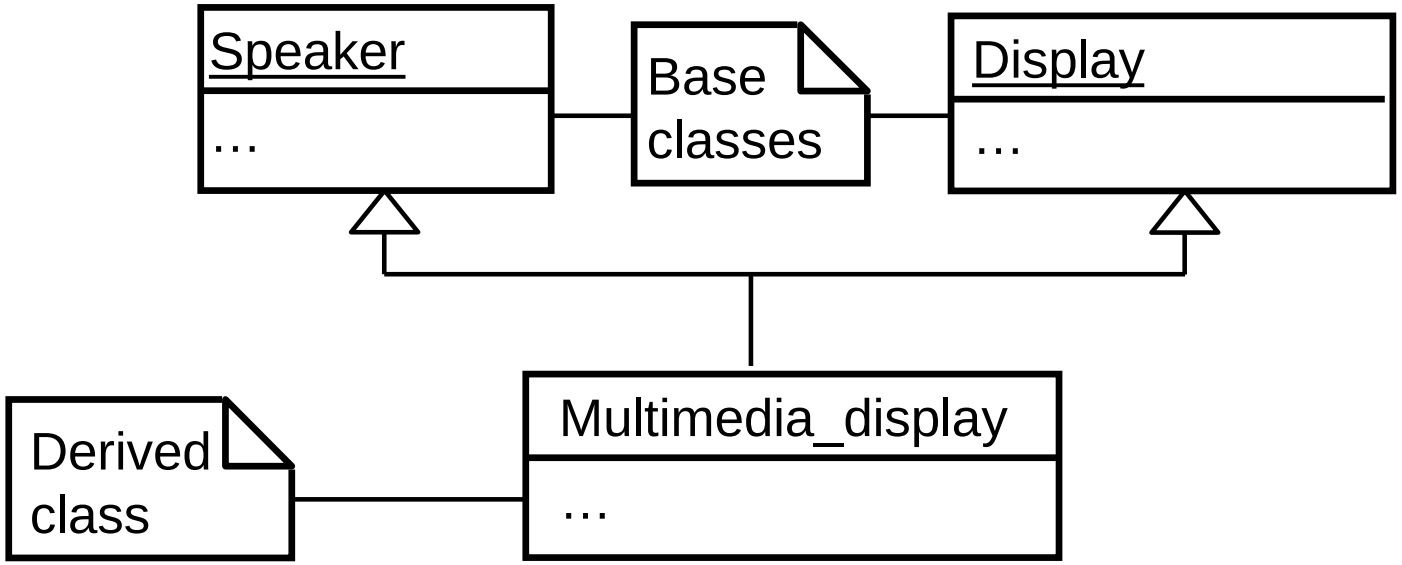
Example:



(Generalization)

UML allows multiple inheritance. A class may be derived from several base classes. It inherits all attributes and operations from every base class.

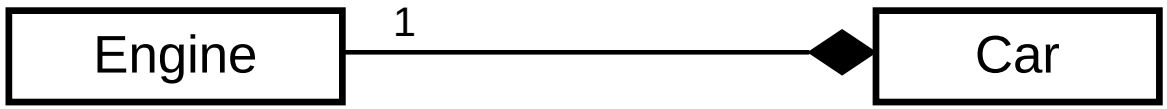
Example:



Composition

The composition relation is represented by a line with a filled diamond-shaped head.
The related objects are bound to each other – when the owner is destroyed, the components are destroyed, also.

Example:



- Interpretation: an object of class Engine is a component of an object of class Car.
- An Engine object may be assigned only to one Car object via the composition relation.

Aggregation

The aggregation relation is a more general "contains" relation. The aggregated classes are not tightly bound to each other as in composition. Aggregation is symbolized by a line with a hollow diamond-shaped head.

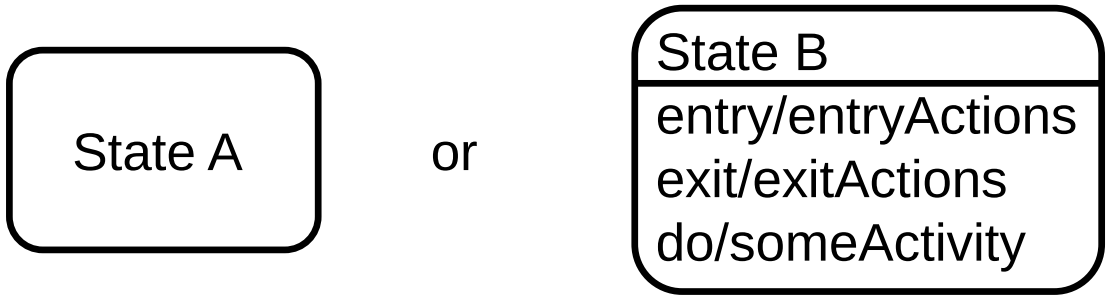
Example:

- Interpretation: several students and one teacher are part of a class
- But: students and teacher are not bound to the class (when the class is over, students and teacher still exist).
- Students and teachers may each be assigned to several classes.

UML Behavioral Diagram Example: State Machine

In UML, behavior of a system can be described using state machines. A state describes a situation in the life of an object. In this situation some activities may take place. Activities are marked by "*entry*" (activity on entering the state), "*exit*" (activity on leaving the state) or "*do*" (activity while being in the state).

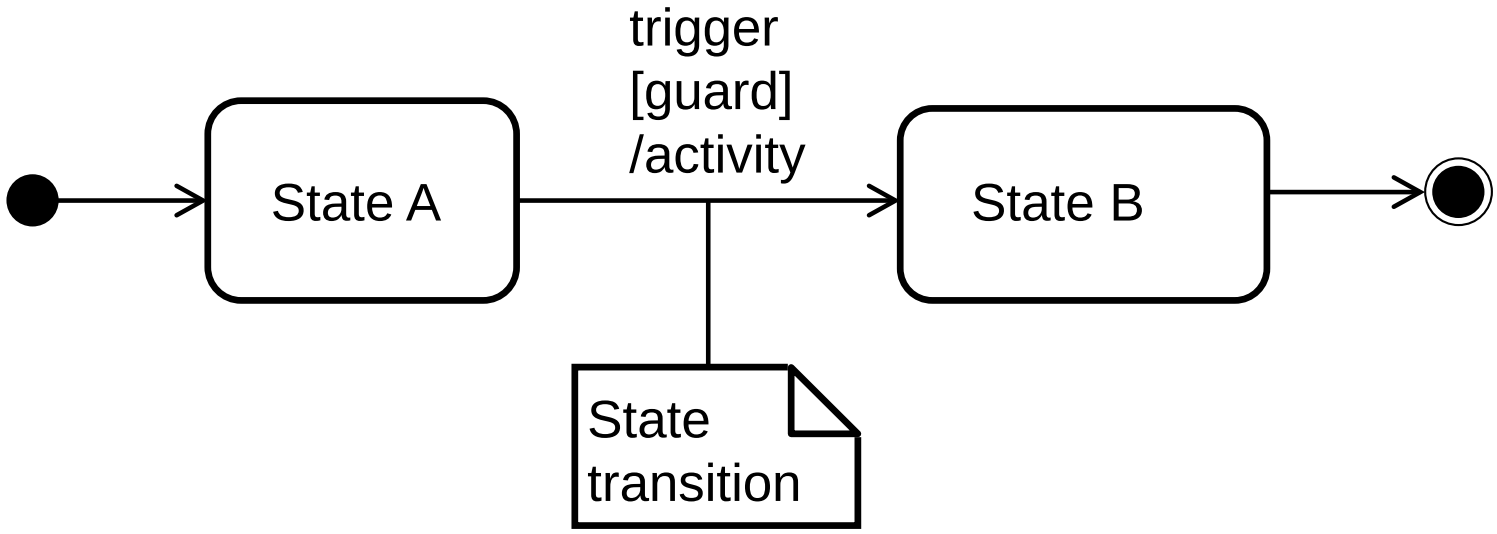
UML symbol for state:



(Behavioral Modeling)

The transition from one state to the next is triggered by an event (not a clock as in hardware). The symbol for a transition is a skeleton arrow. A transition may be labeled with the trigger event, with a guard condition that must be fulfilled for the transition to actually take place, and with an activity being performed during the transition.

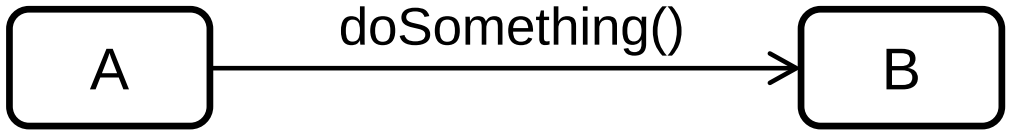
There are special symbols for the start state and the end state (pseudo states).



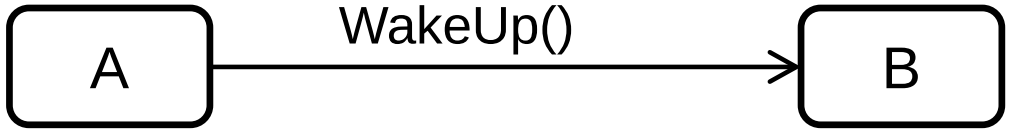
Events

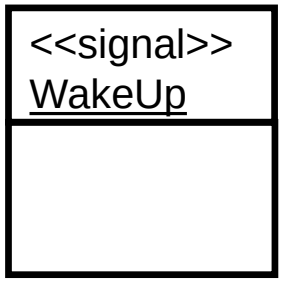
UML defines several kinds of events that can trigger a state transition.

A call event is similar to a procedure call. An operation is invoked on the object; this triggers the state transition.



A signal event is an asynchronous reception of a signal. The signal itself is represented in the diagram as a special object labeled as <<signal>>.

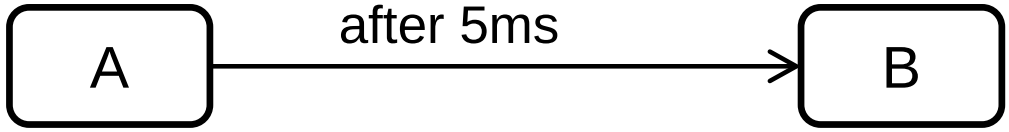




(Events)

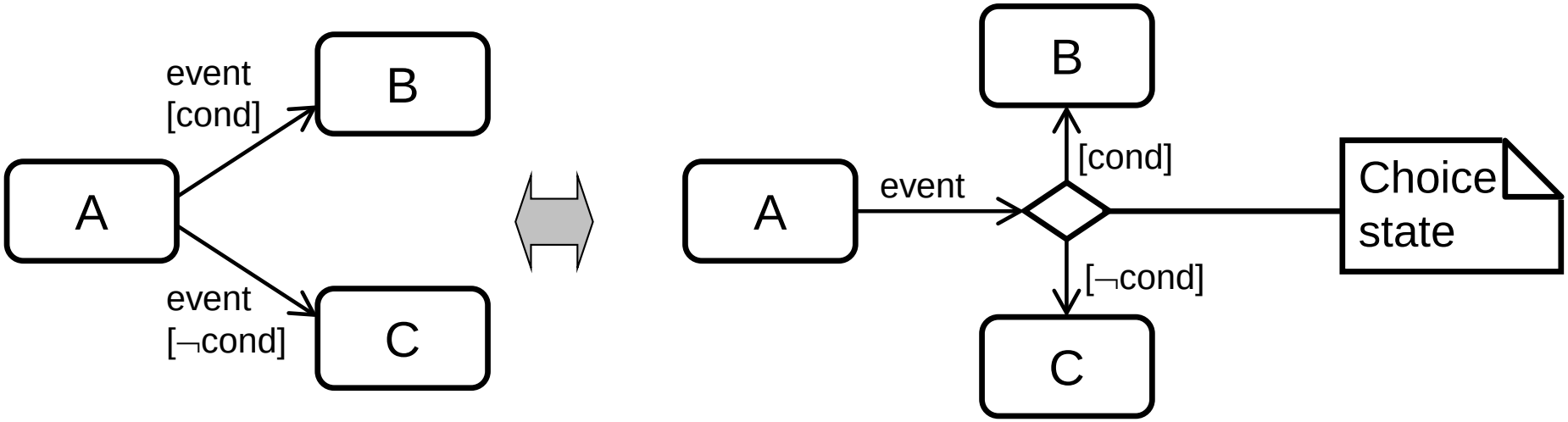
A change event occurs when some condition becomes true.

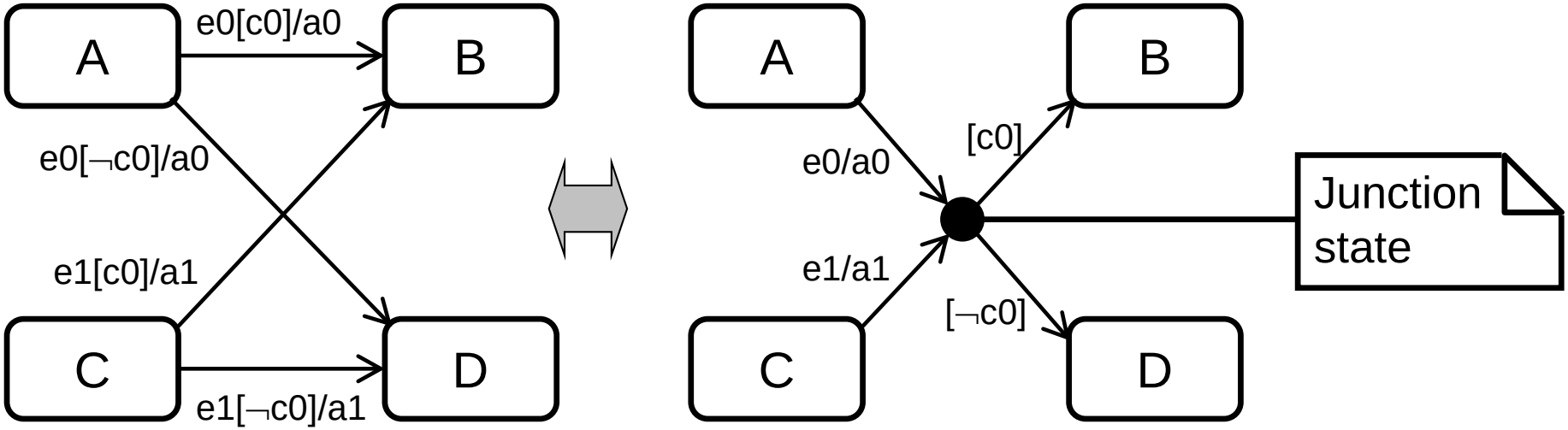
A time event causes the state machine to leave a state after a certain amount of time. UML defines the keyword "*after*" to label time events.



Choice and Junction States

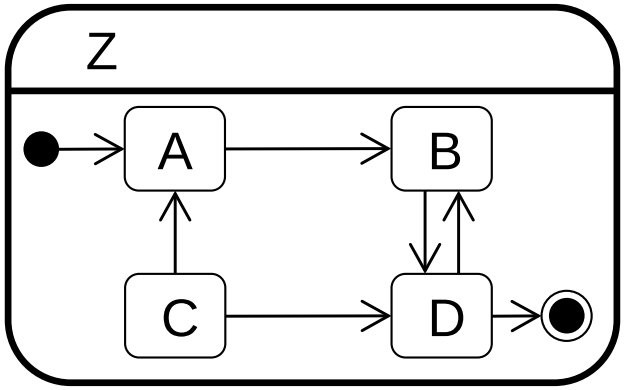
The choice state and the junction state are pseudo states that help to unclutter state diagrams.



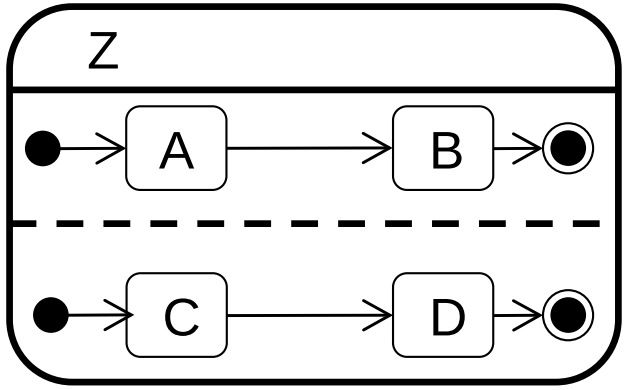


Composite States

States may be composed of sub-state-machines. A composite state can be of OR-type (sequential sub-states), or of AND type (concurrent sub-states).
[This is similar to *Statecharts*.]



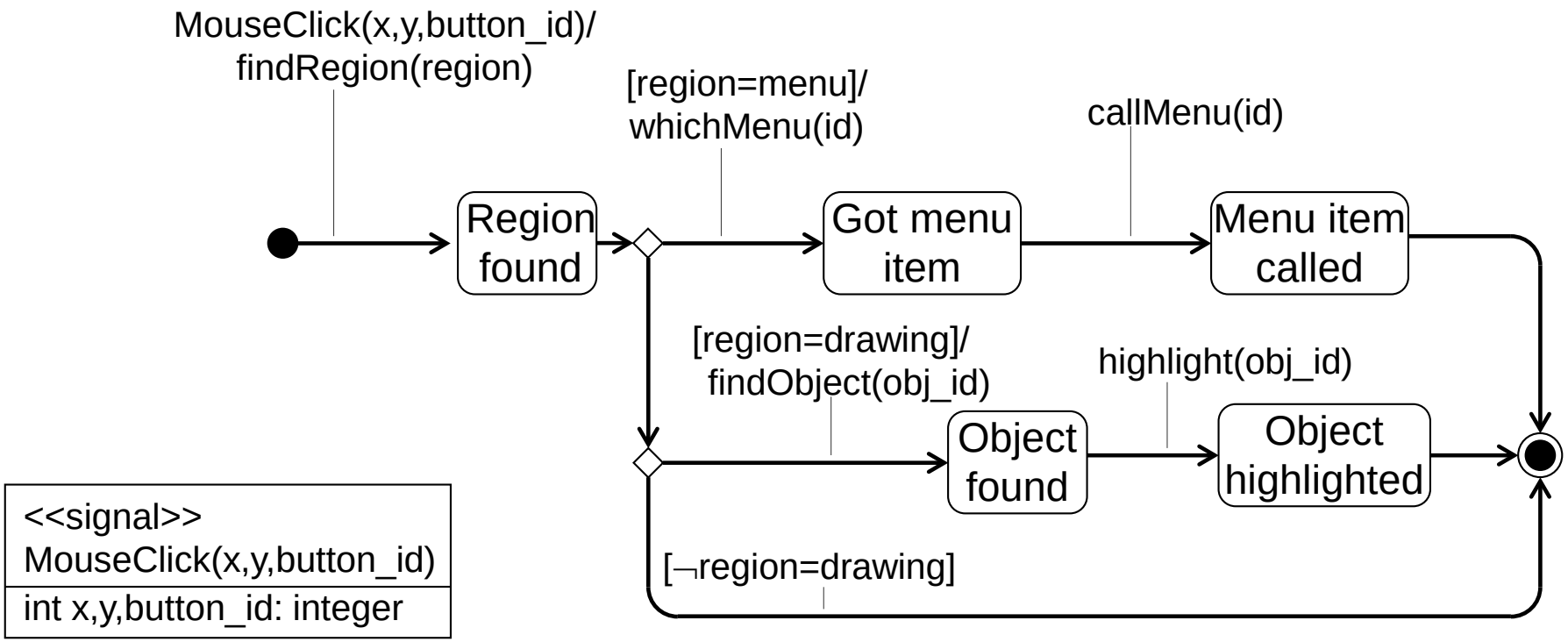
Example of OR composite state:
While in Z, the machine can be in either one of the sub-states A, B, C or D.



Example of AND composite state:
While in Z, the machine can be simultaneously in either (A and C), (A and D), (B and C) or (B and D).

UML State Machine Example

Example of a state diagram describing the behavior of a display+mouse user interface.

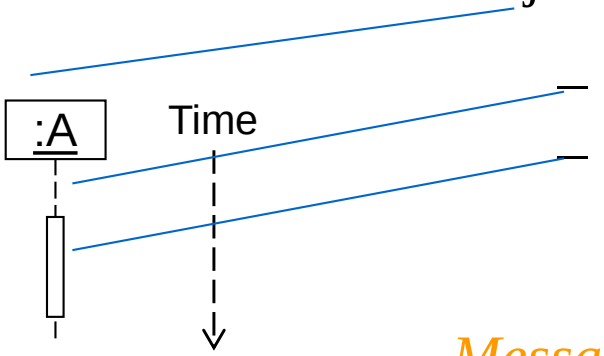


Sequence Diagrams

In UML, Sequence Diagrams are a special form of Interaction Diagrams. They are used to model the communication between objects by showing a particular scenario of behavior with a choice of events happening and messages being exchanged between objects.

Sequence diagrams are similar to timing diagrams in hardware, however time flows vertically (from top to bottom).

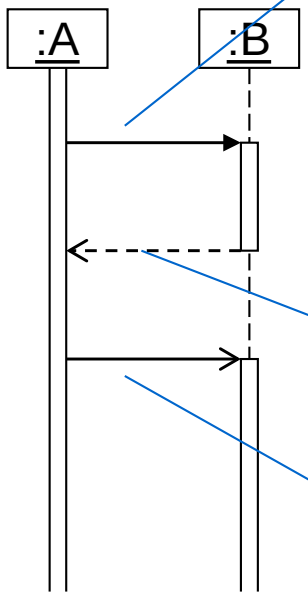
Sequence Diagrams – Elements



Objects involved in the scenario; for each object a

- *Life line*: shows when object exists
- *Execution specification (Activity line)*: denotes execution of a sequence of atomic actions (e.g. when object is active)

Messages being exchanged:



A message may represent

- Operation call and start of an execution (call event)
- Sending and reception of a signal (signal event)

UML distinguishes between

- *Synchronous message*

  Before proceeding the sending object waits for receiving a

- *Reply message*

  Note: Sender remains active while waiting for reply

- *Asynchronous message*

  Sending object continues execution; may be used to start a new (concurrent) thread of control