UNIT -2

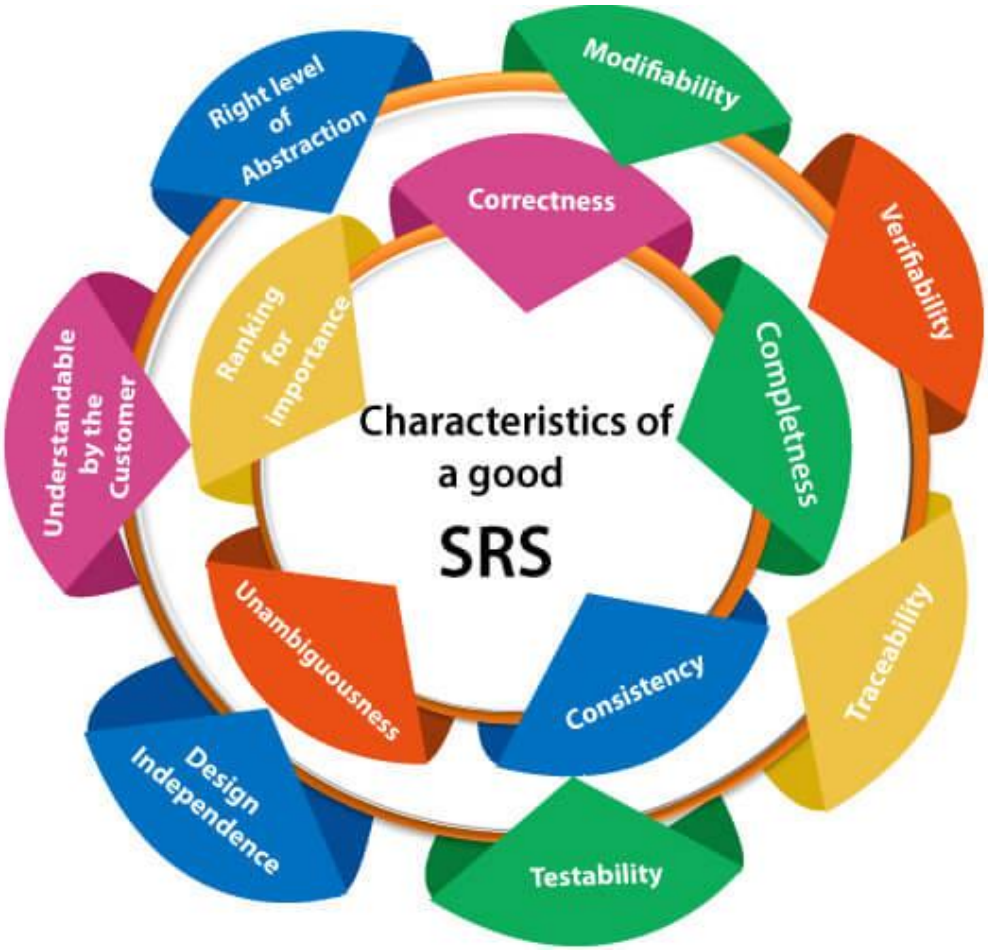
Software Requirement Specification

A software requirements specification (SRS) is a detailed description of a software system to be developed with its functional and non-functional requirements. The SRS is developed based the agreement between customer and contractors. It may include the use cases of how user is going to interact with software system. The software requirement specification document consistent of all necessary requirements required for project development. To develop the software system we should have clear understanding of Software system. To achieve this we need to continuous communication with customers to gather all requirements.

A good SRS defines the how Software System will interact with all internal modules, hardware, communication with other programs and human user interactions with wide range of real life scenarios. Using the *Software requirements specification* (SRS) document on QA lead, managers creates test plan. It is very important that testers must be cleared with every detail specified in this document in order to avoid faults in test cases and its expected results.

It is highly recommended to review or test SRS documents before start writing test cases and making any plan for testing. Let's see how to test SRS and the important point to keep in mind while testing it.

Characteristics of good SRS



Following are the features of a good SRS document:

1. Correctness: User review is used to provide the accuracy of requirements stated in the SRS. SRS is said to be perfect if it covers all the needs that are truly expected from the system.

2. Completeness: The SRS is complete if, and only if, it includes the following elements:

(1). All essential requirements, whether relating to functionality, performance, design, constraints, attributes, or external interfaces.

(2). Definition of their responses of the software to all realizable classes of input data in all available categories of situations.

> ***Note: It is essential to specify the responses to both valid and invalid values.***

(3). Full labels and references to all figures, tables, and diagrams in the SRS and definitions of all terms and units of measure.

3. Consistency: The SRS is consistent if, and only if, no subset of individual requirements described in its conflict. There are three types of possible conflict in the SRS:

(1). The specified characteristics of real-world objects may conflicts. For example,

(a) The format of an output report may be described in one requirement as tabular but in another as textual.

(b) One condition may state that all lights shall be green while another states that all lights shall be blue.

(2). There may be a reasonable or temporal conflict between the two specified actions. For example,

(a) One requirement may determine that the program will add two inputs, and another may determine that the program will multiply them.

(b) One condition may state that "A" must always follow "B," while other requires that "A and B" co-occurs.

(3). Two or more requirements may define the same real-world object but use different terms for that object. For example, a program's request for user input may be called a "prompt" in one requirement's and a "cue" in another. The use of standard terminology and descriptions promotes consistency.

4. Unambiguousness: SRS is unambiguous when every fixed requirement has only one interpretation. This suggests that each element is uniquely interpreted. In case there is a method

used with multiple definitions, the requirements report should determine the implications in the SRS so that it is clear and simple to understand.

5. Ranking for importance and stability: The SRS is ranked for importance and stability if each requirement in it has an identifier to indicate either the significance or stability of that particular requirement.

Typically, all requirements are not equally important. Some prerequisites may be essential, especially for life-critical applications, while others may be desirable. Each element should be identified to make these differences clear and explicit. Another way to rank requirements is to distinguish classes of items as essential, conditional, and optional.

6. Modifiability: SRS should be made as modifiable as likely and should be capable of quickly obtain changes to the system to some extent. Modifications should be perfectly indexed and cross-referenced.

7. Verifiability: SRS is correct when the specified requirements can be verified with a cost-effective system to check whether the final software meets those requirements. The requirements are verified with the help of reviews.

8. Traceability: The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each condition in future development or enhancement documentation.

There are two types of Traceability:

1. Backward Traceability: This depends upon each requirement explicitly referencing its source in earlier documents.

2. Forward Traceability: This depends upon each element in the SRS having a unique name or reference number.

The forward traceability of the SRS is especially crucial when the software product enters the operation and maintenance phase. As code and design document is modified, it is necessary to be able to ascertain the complete set of requirements that may be concerned by those modifications.

9. Design Independence: There should be an option to select from multiple design alternatives for the final system. More specifically, the SRS should not contain any implementation details.

10. Testability: An SRS should be written in such a method that it is simple to generate test cases and test plans from the report.

11. Understandable by the customer: An end user may be an expert in his/her explicit domain but might not be trained in computer science. Hence, the purpose of formal notations and symbols should be avoided too as much extent as possible. The language should be kept simple and clear.

12. The right level of abstraction: If the SRS is written for the requirements stage, the details should be explained explicitly. Whereas,for a feasibility study, fewer analysis can be used. Hence, the level of abstraction modifies according to the objective of the SRS.

Properties of a good SRS document

The essential properties of a good SRS document are the following:

Concise: The SRS report should be concise and at the same time, unambiguous, consistent, and complete. Verbose and irrelevant descriptions decrease readability and also increase error possibilities.

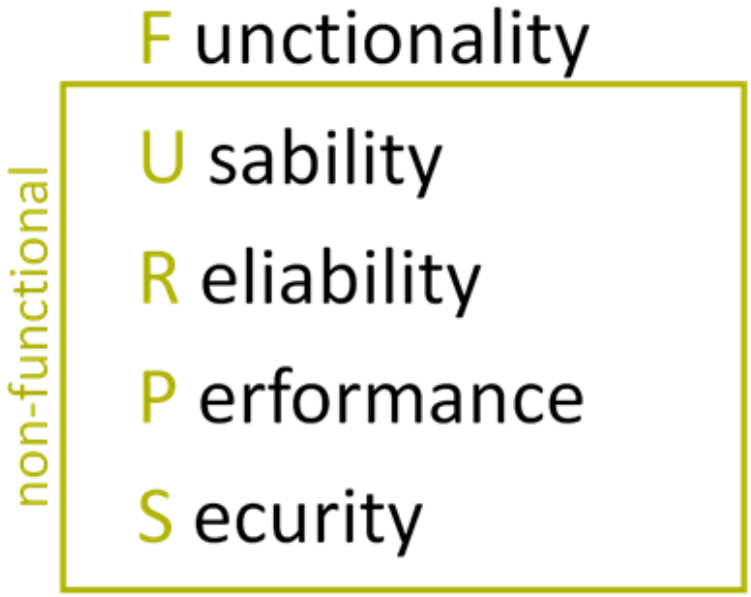
Structured: It should be well-structured. A well-structured document is simple to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the user requirements. Often, user requirements evolve over a period of time. Therefore, to make the modifications to the SRS document easy, it is vital to make the report well-structured.

Black-box view: It should only define what the system should do and refrain from stating how to do these. This means that the SRS document should define the external behavior of the system and not discuss the implementation issues. The SRS report should view the system to be developed as a black box and should define the externally visible behavior of the system. For this reason, the SRS report is also known as the black-box specification of a system.

Conceptual integrity: It should show conceptual integrity so that the reader can merely understand it. Response to undesired events: It should characterize acceptable responses to unwanted events. These are called system response to exceptional conditions.

Verifiable: All requirements of the system, as documented in the SRS document, should be correct. This means that it should be possible to decide whether or not requirements have been met in an implementation.

FURPS:-



FURPS is an acronym representing a model for classifying requirements.

Usability - UX, Human Factors, Aesthetics, Consistency, Documentation

Reliability - Availability, Robustness/Durability, Recoverability, Stability, Accuracy

Performance - Speed, Efficiency, Resource Consumption

The IEEE standard for requirements documents

The most widely known requirements document standard is (IEEE, 1998). This IEEE standard suggests the following structure for requirements documents:

1. Introduction
1.1 Purpose of the requirements document
1.2 Scope of the product
1.3 Definitions, acronyms and abbreviations
1.4 References
1.5 Overview of the remainder of the document

2. General description
2.1 Product perspective
2.2 Product functions
2.3 User characteristics
2.4 General constraints
2.5 Assumptions and dependencies

3.**Specific requirements**, covering functional, non-functional and interface requirements. This is obviously the most substantial part of the document but because of the wide variability in organisational practice, it is not appropriate to define a standard structure for this section. The requirements may document external interfaces, describe system functionality and performance, and specify logical database requirements, design constraints, emergent system properties and quality characteristics.

4.**Appendices**

5.**Index**

Although the IEEE standard is not ideal, it contains a great deal of good advice on how to write requirements and how to avoid problems. It is too general to be an organisational standard in its own right. It is a general framework that can be tailored and adapted to define a standard geared to the needs of a particular organisation.

Useful link -

Data Dictionaries

A data dictionary is a file or a set of files that includes a database's metadata. The data dictionary hold records about other objects in the database, such as data ownership, data relationships to other objects, and other data. The data dictionary is an essential component of any relational database. Ironically, because of its importance, it is invisible to most database users. Typically, only database administrators interact with the data dictionary.

The data dictionary, in general, includes information about the following:

- Name of the data item
- Aliases
- Description/purpose
- Related data items
- Range of values
- Data structure definition/Forms

The **name of the data item** is self-explanatory.

Aliases include other names by which this data item is called DEO for Data Entry Operator and DR for Deputy Registrar.

Description/purpose is a textual description of what the data item is used for or why it exists.

Related data items capture relationships between data items e.g., total_marks must always equal to internal_marks plus external_marks.

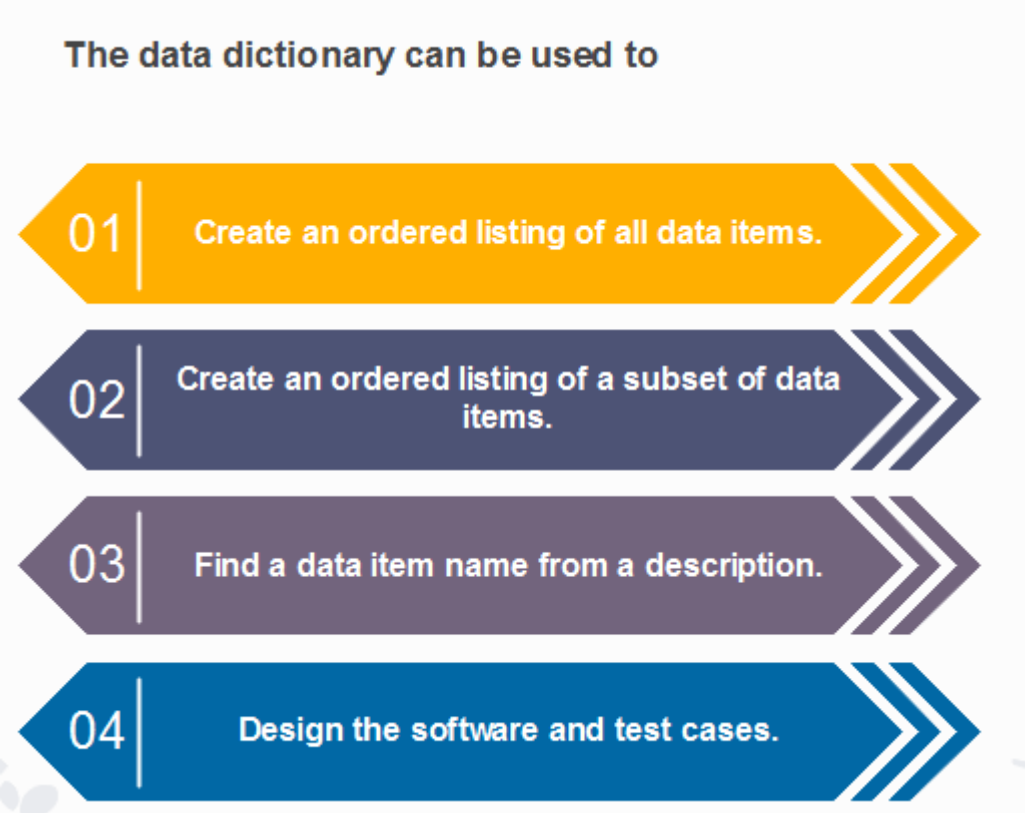
Range of values records all possible values, e.g. total marks must be positive and between 0 to 100.

Data structure Forms: Data flows capture the name of processes that generate or receive the data items. If the data item is primitive, then data structure form captures the physical structures of the data item. If the data is itself a data aggregate, then data structure form capture the composition of the data items in terms of other data items.

The mathematical operators used within the data dictionary are defined in the table:

Notations	Meaning
x=a+b	x includes of data elements a and b.
x=[a/b]	x includes of either data elements a or b.
x=a x	includes of optimal data elements a.

x=y[a]	x includes of y or more occurrences of data element a
x=[a]z	x includes of z or fewer occurrences of data element a
x=y[a]z	x includes of some occurrences of data element a which are between y and z.



Unified Modeling Language

History of UML

UML was developed by Grady Booch, Ivar Jacobson and James Rumbaugh (**The Three Amigos**)

UML versions

- **0.8** Booch & Rumbaugh 1995
- **0.9** The Three Amigos 1996
- **1.1** OMG 1997
- **1.2-1.5** OMG refinements 1998-2003
- **2.0** OMG major review and redesign 2003-2005
- **2.1.1, 2.1.2** OMG 2007
- **2.2** OMG 2009
- **2.3** OMG 2010

- **2.4.1** OMG 2011
- **2.5** OMG 2015 current version

UML specification: http://omg.org/spec/UML/Current

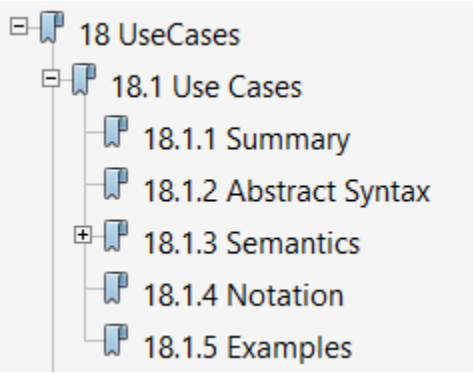
Overview

- OMG Standard
- The Unified Modeling Language (UML)
- Data modeling, business modeling (work flows), object modeling, and component modeling
- UML aims to be a standard modeling language which can model concurrent and distributed systems
- UML is a de-facto industry standard
- UML models may be automatically transformed to other representations (e.g. Java, PHP)
- UML is extensible, through **profiles**, **stereotypes** and **tagged values**
- A lot of other languages are based on UML (SysML, SoaML)
- Usually default language for a lot of architecture frameworks (MoDAF, DoDAF, NAF)

UML 2.5 Specification

- *This specification has been extensively re-written from its previous version to make it easier to read by removing redundancy and increasing clarity.*
- The UML Infrastructure no longer forms part of the UML specification.
- Package Merge is not used within the specification.
- The compliance levels L0, L1, L2, and L3 have been eliminated, because they were not found to be useful in practice. A tool either complies with the whole of UML or it does not.

Parts of UML 2.5 Specification



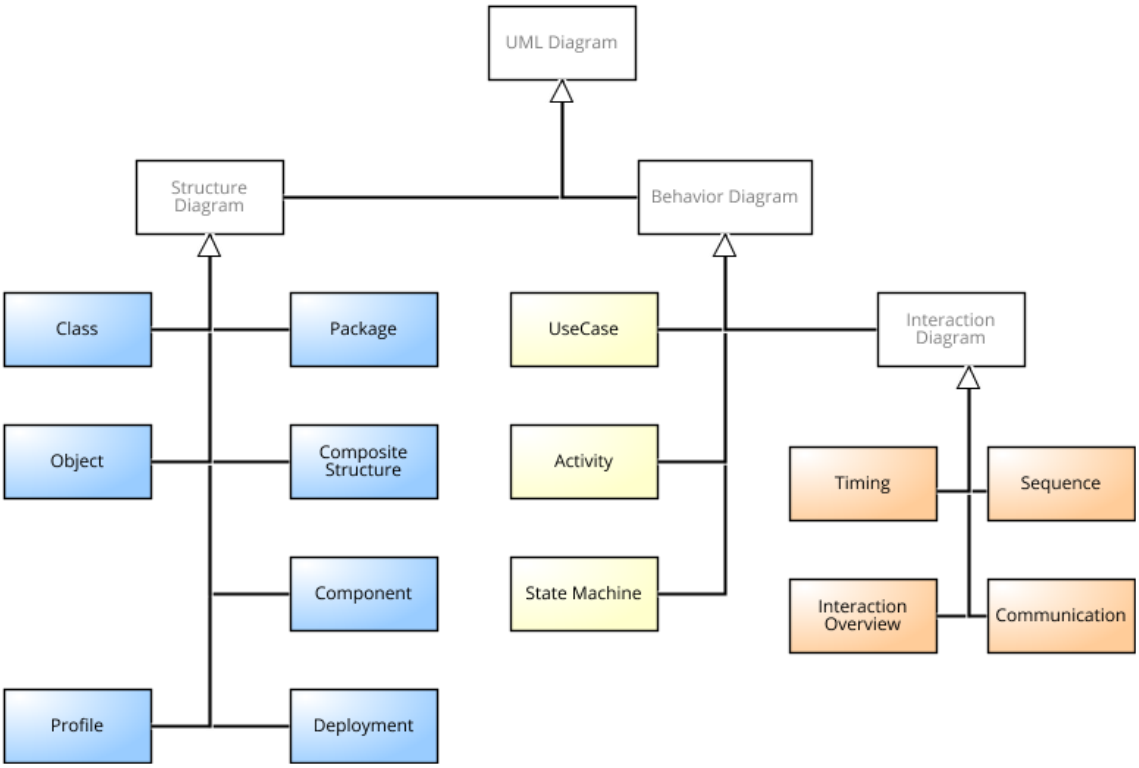
- **Abstract Syntax** contains one or more diagrams that define that capability in terms of the UML metamodel
- **Semantics** specifies the semantics (meaning) of all of the concepts.
- **Notation** specifies the notation corresponding to all of the concepts defined in the sub clause. Only concepts that can appear in diagrams will have a notation specified.

UML Features

- Does not define method (see Unified Process)
- UML defines both:
  - UML model (contains documentation and all relations)

    - UML diagrams (partial graphic representation of a system's model)
- UML can model both views of the system:
    - Static (structural)
    - Dynamic (behavioral)
- Contains 14 different diagrams

UML Diagrams



Structure diagrams

Structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.

- **Class diagram**: describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
- **Component diagram**: describes how a software system is split up into components and shows the dependencies among these components.
- **Composite structure diagram**: describes the internal structure of a classifier and the collaborations that this structure makes possible.
- **Deployment diagram**: describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
- **Object diagram**: shows a complete or partial view of the structure of an example modeled system at a specific time.
- **Package diagram**: describes how a system is split up into logical groupings by showing the dependencies among these groupings.
- **Profile diagram**: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype.

Behavior diagrams

Behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

- **Activity diagram**: describes the business and operational step-by-step workflows of components in a system.
- **Use Case diagram**: describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.
- **State machine diagram**: describes the states and state transitions of the system.

Interaction diagrams⌘

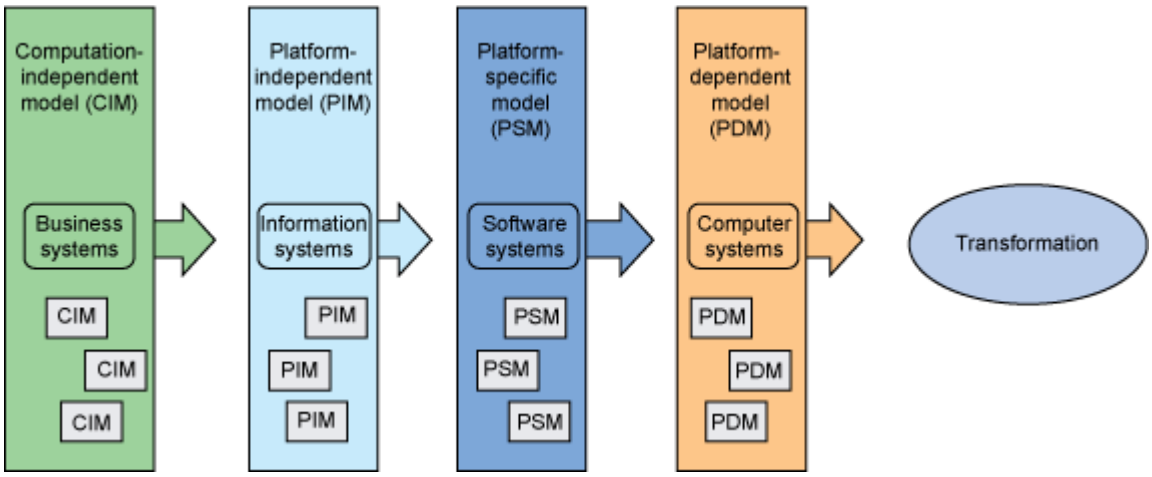
Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled:

- **Sequence diagram**: shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.
- **Communication diagram**: shows the interactions between objects or parts in terms of sequenced messages.
- **Interaction overview diagram**: provides an overview in which the nodes represent communication diagrams.
- **Timing diagrams**: a specific type of interaction diagram where the focus is on timing constraints.

MDA

- **Model Driven Architecture** (MDA) is a software design approach
- specification is expressed as computer model
- created by OMG in 2001
- http://www.omg.org/mda/

MDA Process



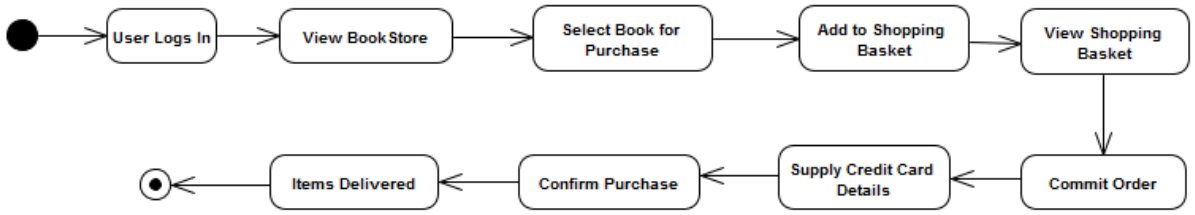
MDA Process

- Define system functionality using a platform-independent model (PIM)
  - domain-specific language DSL
  - e.g. UML, BPMN, SoaML, etc...
- Translate PIM to one or more executable platform-specific model(s) (PSMs)
  - uses different DSLs (e.g. MVEL)
  - or programming languages (e.g Java, C#, etc...)
- Transformation is done by automated tool

Related Standards

- The Meta Object Facility (MOF)
  - http://www.omg.org/mof/
- The Unified Modeling Language (UML)
  - http://www.omg.org/spec/UML/Current/
- The Common Warehouse Metamodel (CWM)
  - http://www.omg.org/spec/CWM/

Activity Diagram



- Describes the workflow behavior of a system, focuses on flows.
- A Process describes a sequence or flow of Activities in an organization with the objective of carrying out work.

Activity, Action

Activity

- describes a **sequence of actions** based on control models and object flow models
- contains edges and activity nodes (e.g. actions)
- represented by a rectangle with rounded corners

Action

- is a fundamental unit of executable functionality contained within an Activity
- represents a **single step** within an activity

Activity

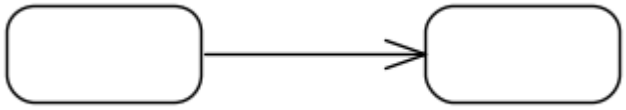
Activity contains **nodes**:

- Action
- Object
- Control Node

and **edges**:

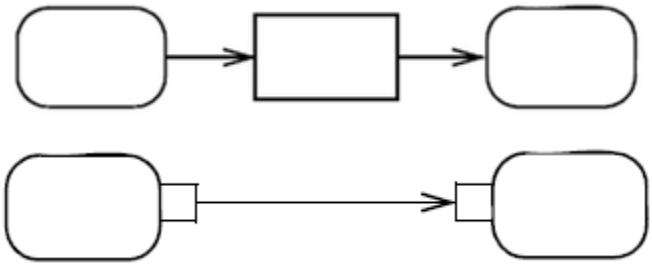
- Control Flow
- Object Flow

Control Flow

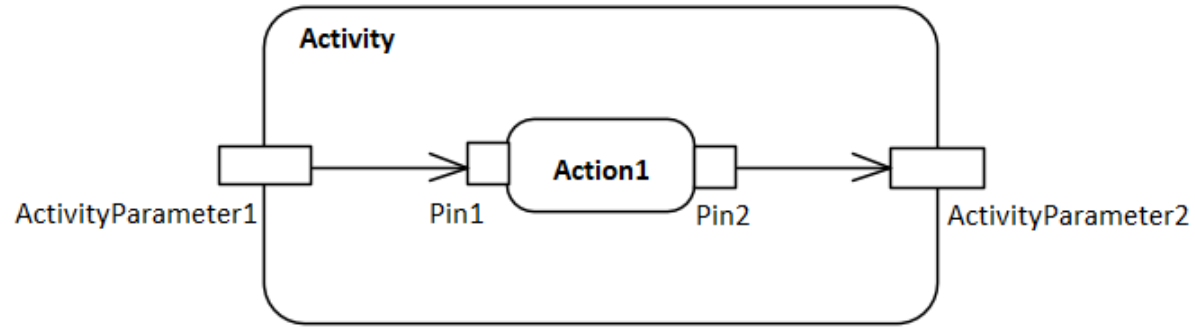


- A control flow is an edge that starts an activity node after the previous one is finished

Object Flow

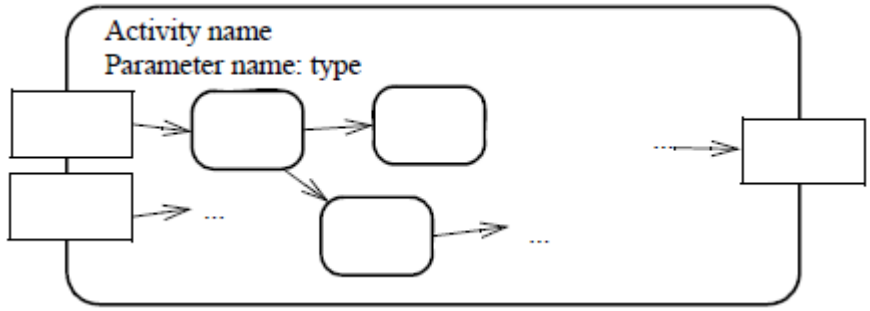


OR

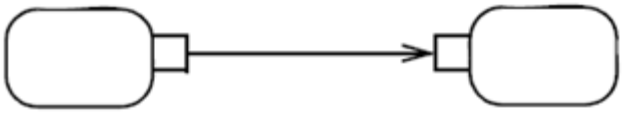


- An object flow is an activity edge that can have objects or data passing along it

Objects



- Incoming or outgoing objects are parameters
- Placed (as rectangles) on the border

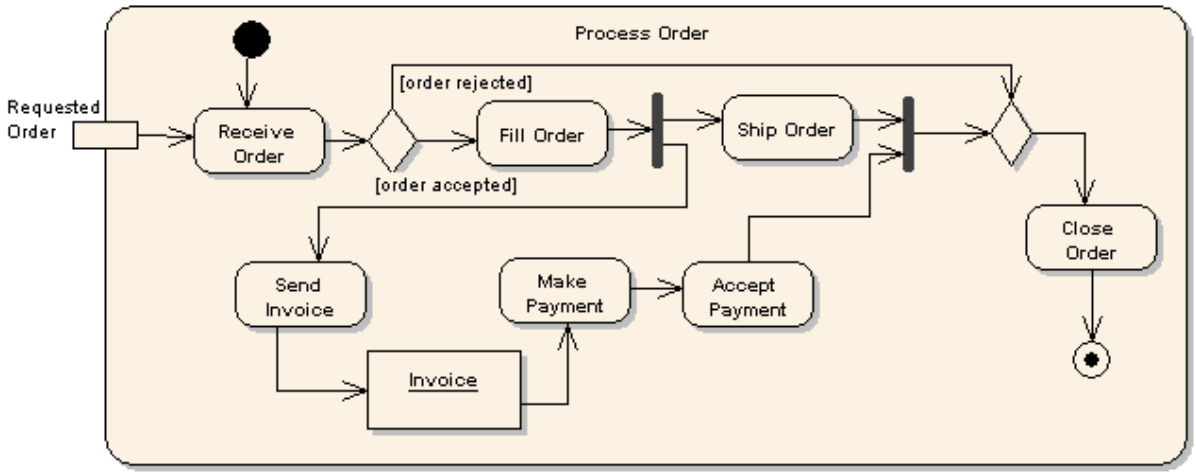


- Or as small rectangle, called a pin (**input** or **output**)

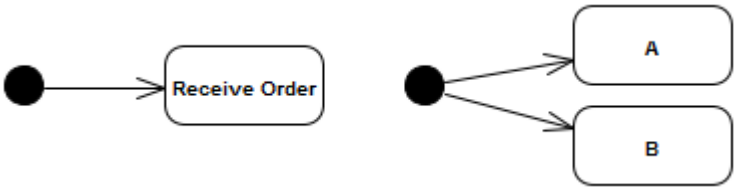
Token

- The semantics of an activity is based on a token flow
- Control tokens and object tokens flow **between** nodes **over** edges

Activity Example

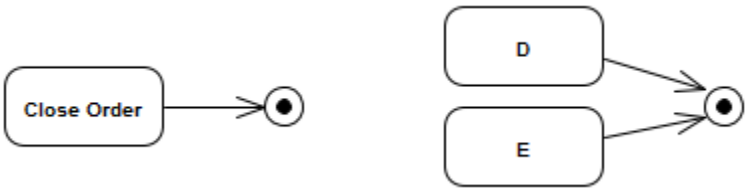


Initial Node



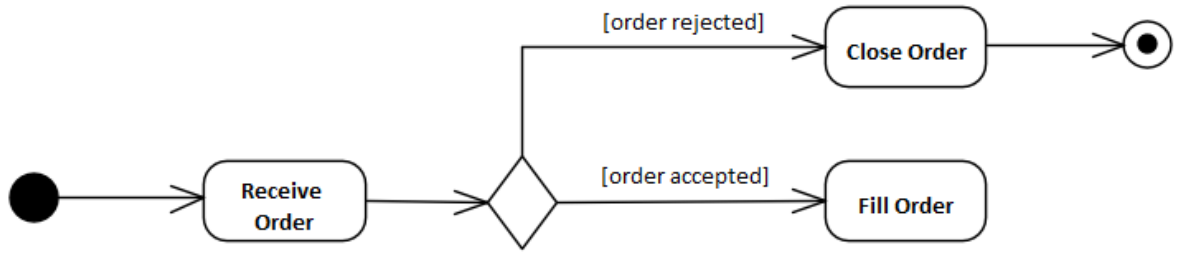
- It has outgoing edges but no incoming edges
- An activity may have more than one initial node
  - Each generates a concurrent flow
- An initial node can have more than one outgoing edge
  - Semantics: a control token at each outgoing edge
- Activity diagrams do not have to have initial nodes
- Notation: a filled circle

Final Node



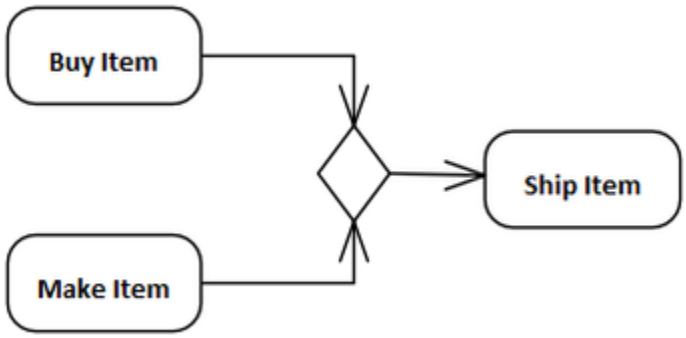
- An activity may have more than one activity final node
- The first one reached stops all flows in the activity
  - Regardless of the number of tokens in activity
- At least one incoming edge and no outgoing edges
- If several incoming edges only one must carry a token
- Activity diagrams do not have to have final nodes
- Notation: a filled circle with an outer ring

Decision



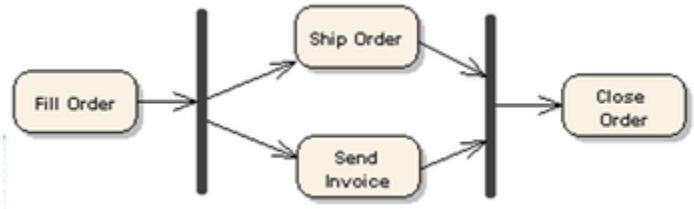
- is a control node that chooses between outgoing flows
- One incoming edge and several outgoing edges
- When a token is supplied guards on outgoing edges are evaluated
  - Token goes to first true outgoing
- Notation: a rhombus

Merge



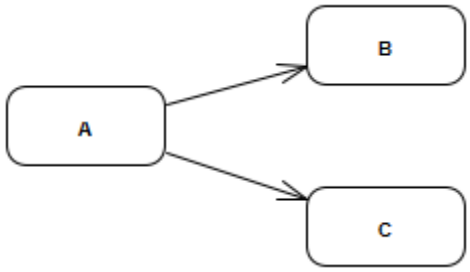
- A merge node is a control node that brings together multiple alternate flows
  - It is not used to synchronize concurrent flows
- Several incoming edges, one outgoing edge
- Nothing calculated, nothing expected
- Notation: an empty rhombus

Fork, Join



- **Fork** – one incoming edge and multiple outgoing edges
- **Join** – multiple incoming edges and one outgoing edge

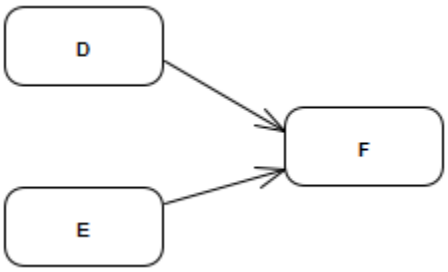
Implicit splitting



Once action A terminates

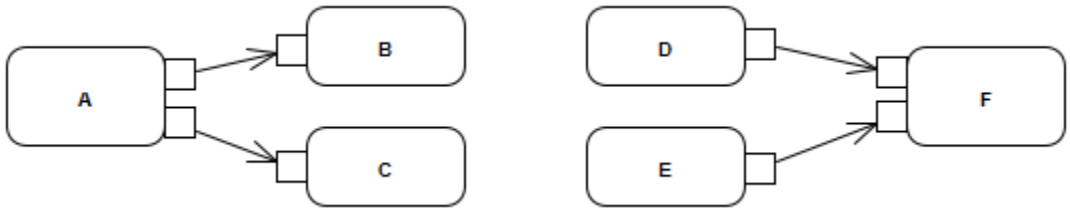
- a control token is available at both outgoing edges
- actions B and C start concurrently

Implicit synchronization



Action F doesn't start until tokens are available at both incoming edges - actions D and E have to terminate

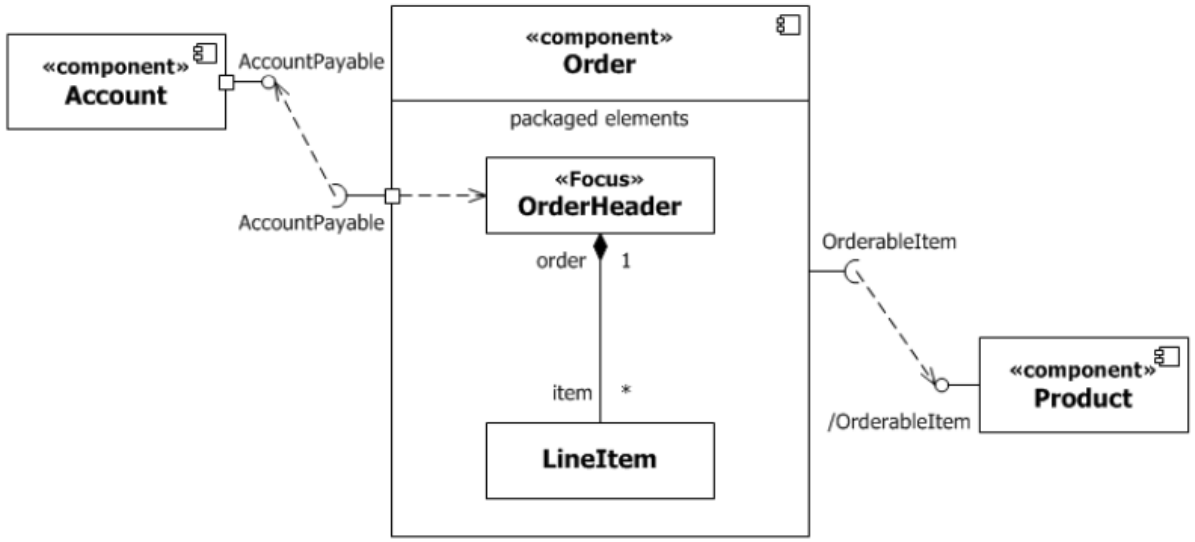
Implicit split/synchronization - object flow



AND semantics for object flows

- Once action A terminates it provides two object nodes
- Action F doesn't start until object tokens are available at both incoming edges - actions D and E have to terminate

Component Diagram

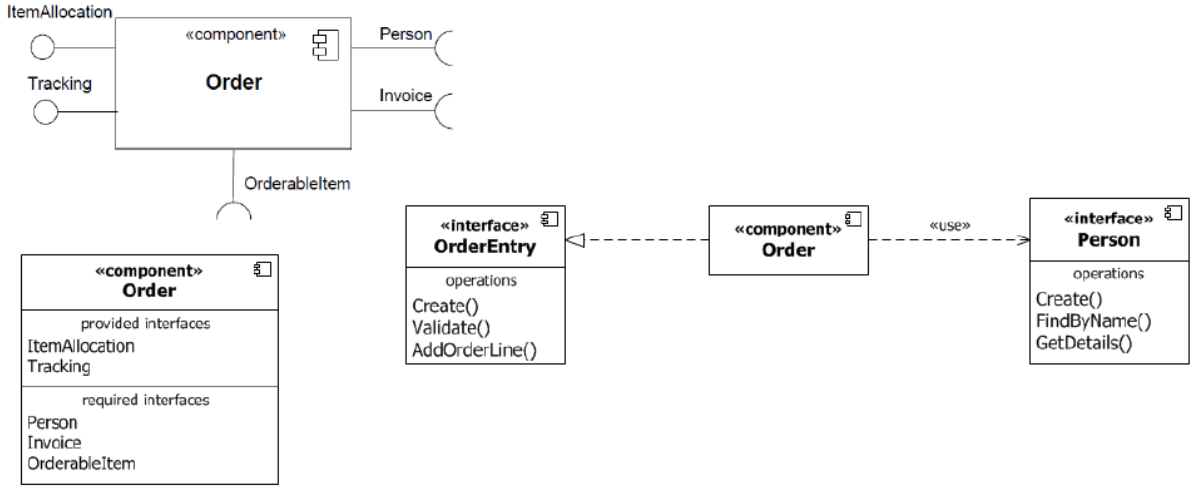


- Describes how a software system is split up into components and shows the dependencies among these components.

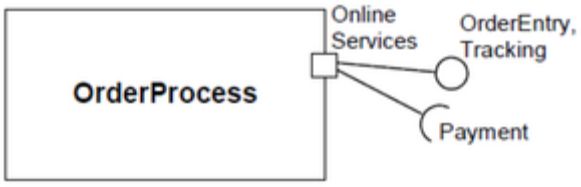
Component

- A Component represents a modular part of a system that encapsulates its contents.
- A Component is a self-contained unit that encapsulates the state and behavior of a number of Classifiers.
- A Component specifies a formal contract of the services that it provides to its clients and those that it requires from other Components
- A Component is a substitutable unit that can be replaced at design time or run-time by a Component that offers equivalent functionality.

Interfaces⌘

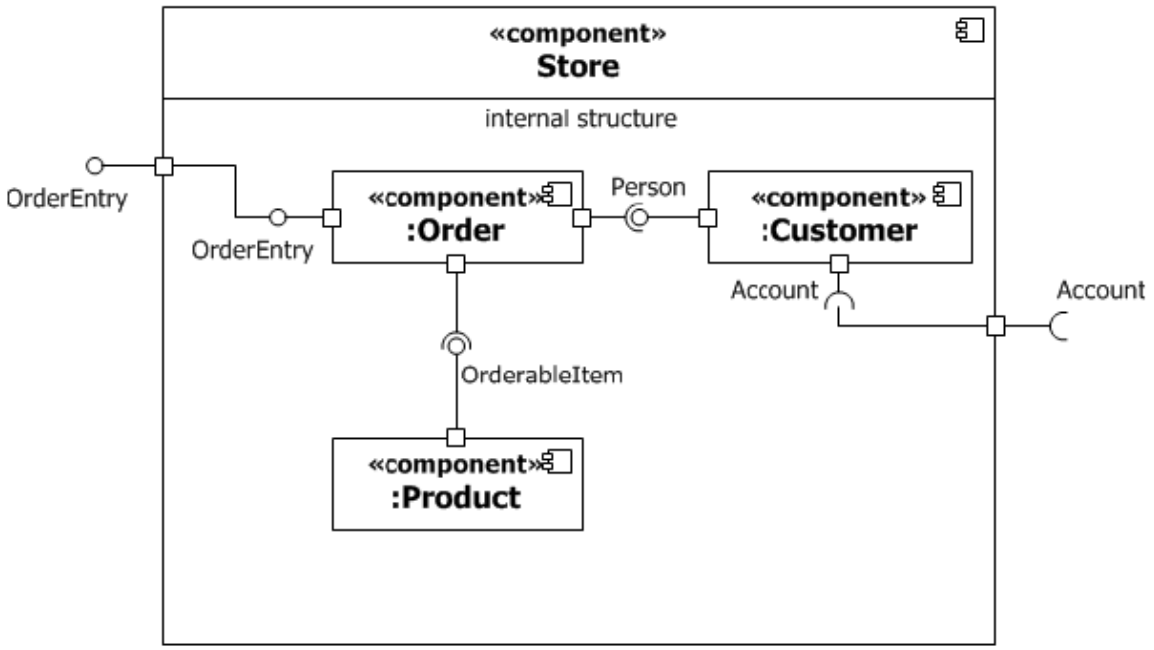


Port



- Ports represent interaction points through which an Classifier communicates with its environment.
- A Port may specify the services an Classifier provides (offers) to its environment as well as the services that an Classifier expects (requires) of its environment.

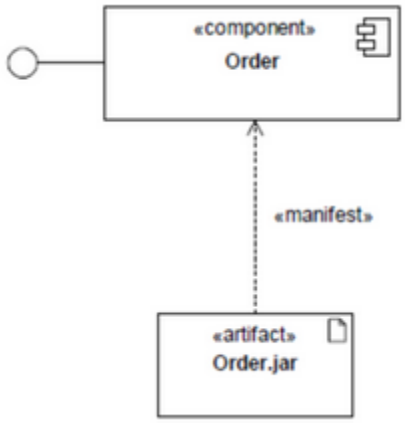
Connector



- Connector specifies a link that enables communication between two (or more) instances.

- Two types of connectors:
  - delegation (connect externally provided interfaces to the parts that realize or require them, represents the forwarding of events)
  - assembly (connector between two or more parts that defines that one part provides the services that other part use)

Manifestation

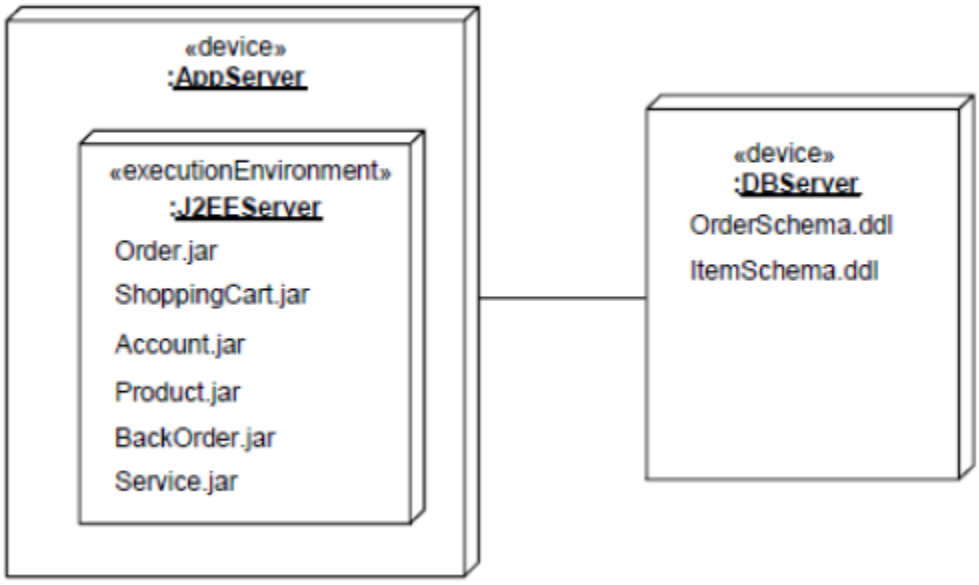


- An Artifact may embody, or manifest, a number of model elements.
- Manifestation represents implementation of one or more model elements by an artifact.

Deployment Diagram

- Describes the hardware used in system implementations, execution environments and artifacts deployed on the hardware.

Node⌘



- A Node is computational resource upon which Artifacts may be deployed.
- Nodes may have complex internal structure (can only consist of other nodes)
- Two types of nodes:

- Device (physical machine components) - e.g. application server, client workstation, mobile device
- Execution Environment (software systems) - e.g. OS, workflow engine, database system, J2EE container

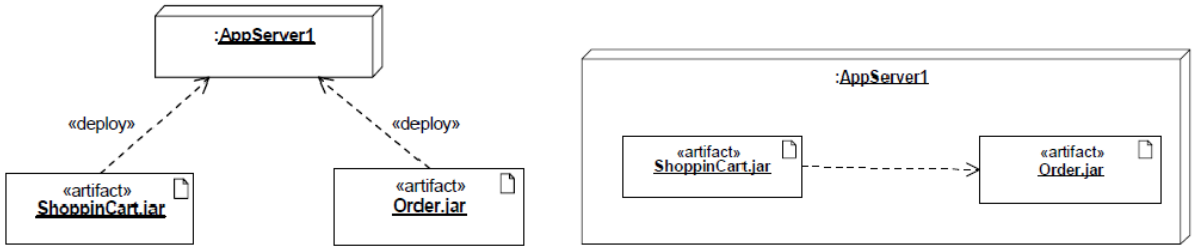
Artifact

- An **artifact** represents some item of information that is used or produced by a software development process or by operation of a system.
- An **artifact** represents concrete elements in the physical world.
- Examples of artifacts: model files, source files, scripts, executable files, database tables, word-processing documents, and mail messages.

Communication Path

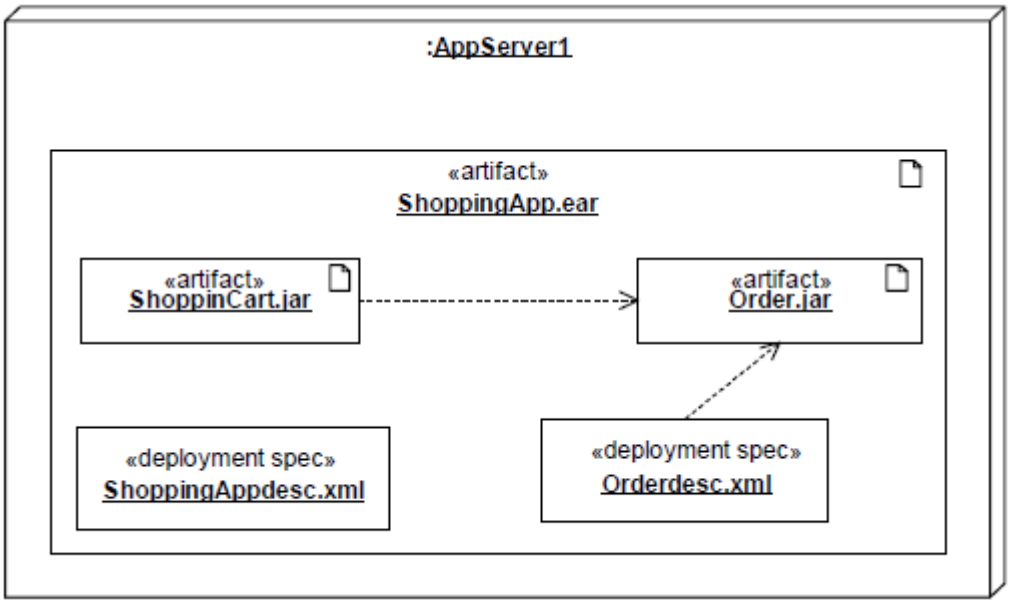
- A communication path is an association between two nodes, through which they are able to exchange signals and messages.
- It usually represents physical connection between devices or some protocol between execution environments.

Deployment



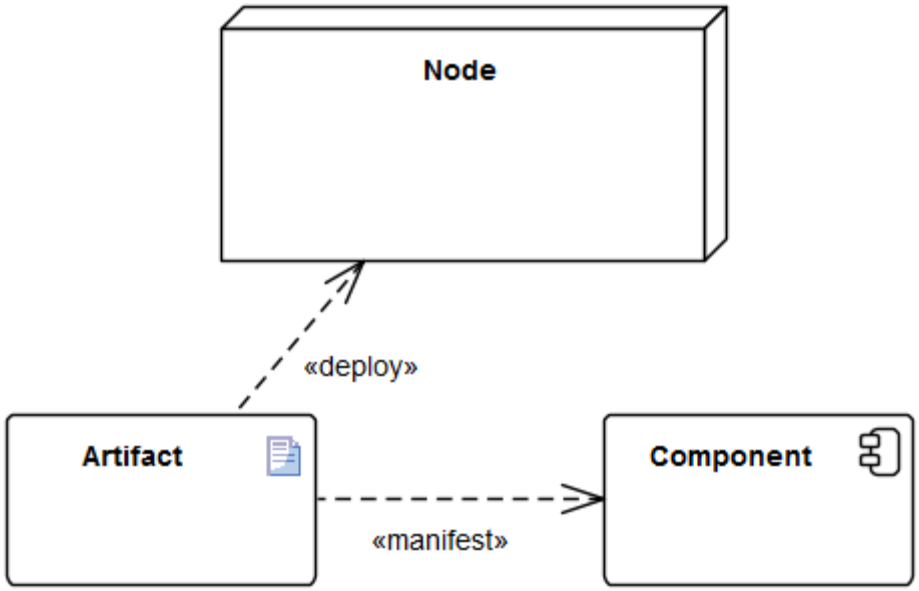
- A deployment is a dependency relationship which describes allocation (deployment) of an artifact to a deployment target.

Deployment Specification



- A deployment specification specifies a set of properties that determine execution parameters of a component artifact that is deployed on a node.

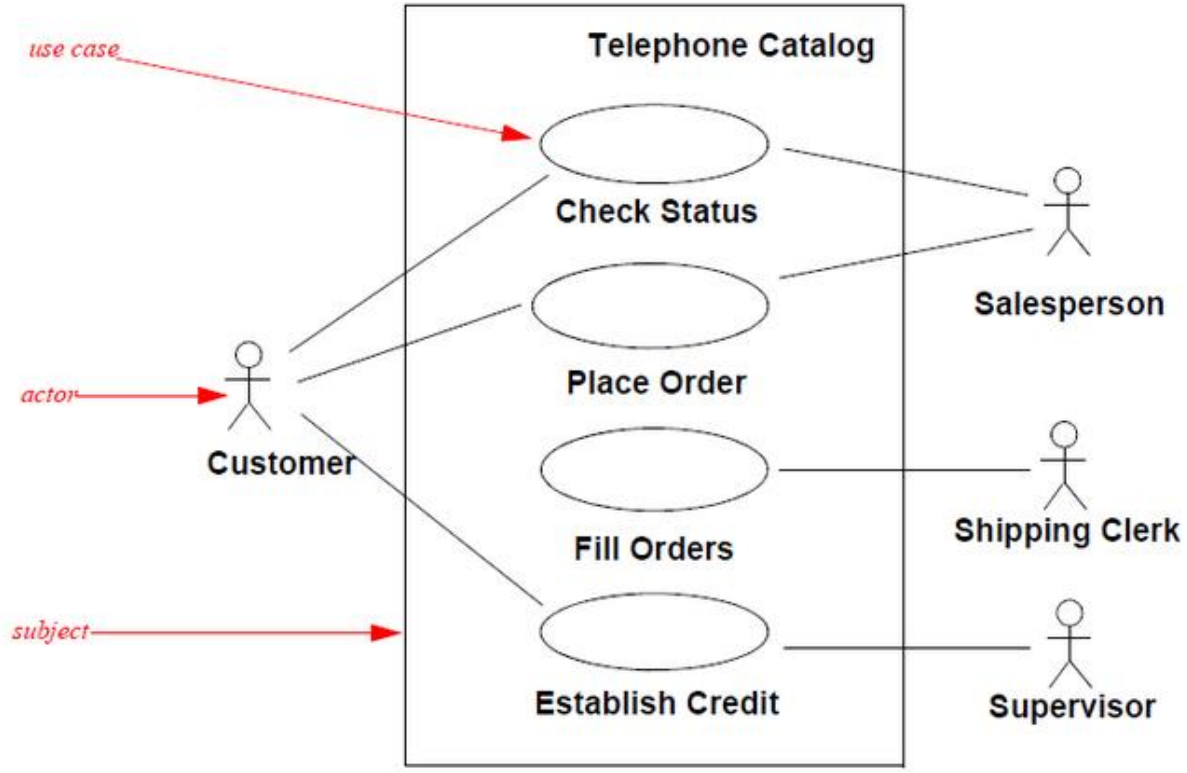
Connection between Node, Component and Artifact



Use Case Basics

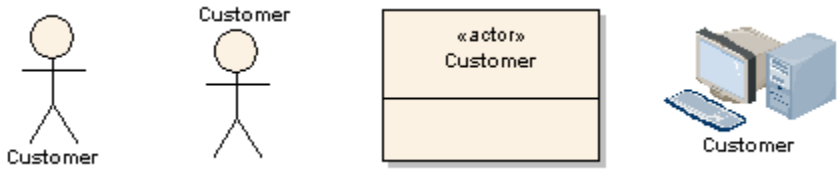
- **Use cases** are used to model the **requirements** of a system (what a system is supposed to do)
- The key concepts associated with use cases are **actors**, **use cases**, and the **subject**.

Use Case Diagram⌘



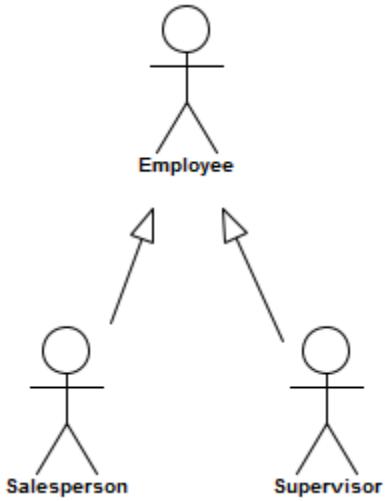
- describes the relationships among a set of use cases and the actors participating in these use cases
- it does NOT describe behavior or flows (**WHO** and **WHAT**, not **HOW**)

Actor



- a **role** played by an **external entity** - a user or any other system that interacts with the subject
- may represent roles played by human users, external hardware, or other subjects
- Notation: "stick man" icon (with the name above or below) or rectangle

Relationships Between Actors

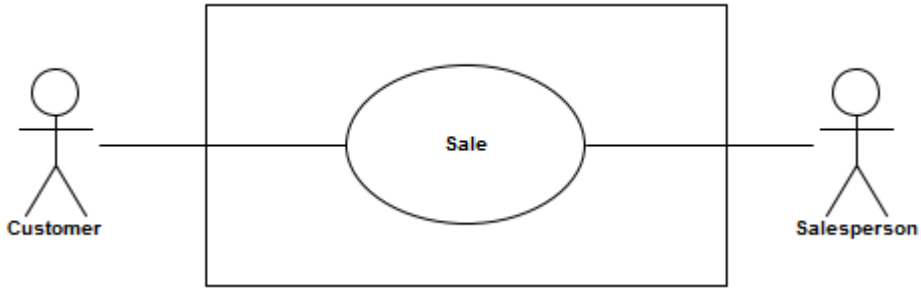


- Generalization
- Specialization

UseCase

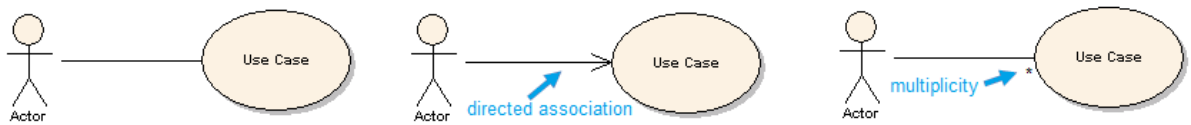
- the specification of a set of actions performed by a system and produce a **result**
- a **complete** set of actions that perform a function for a particular actor
- Notation: an ellipse (with the name inside or below) or rectangle with stereotype

Subject



- a system (subject) is a box containing use cases
- defines boundaries and responsibilities
- actors are **always external** to a system

Relationships Between Actors and Use Cases



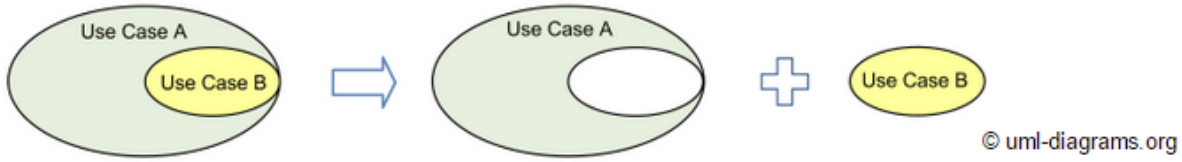
- an **association** expresses a communication path between the actor and the use case
- **directed association** from the active to the passive element (who initiated the communication)
- association with **multiplicity** like 0..*, 2..*, *, etc
  - **at the actor end** - more than one actor instance is involved in initiating the use case
  - **at the use case end** - an actor can be involved in multiple use cases of that type

Relationships Between Use Cases

Three types of relationship:

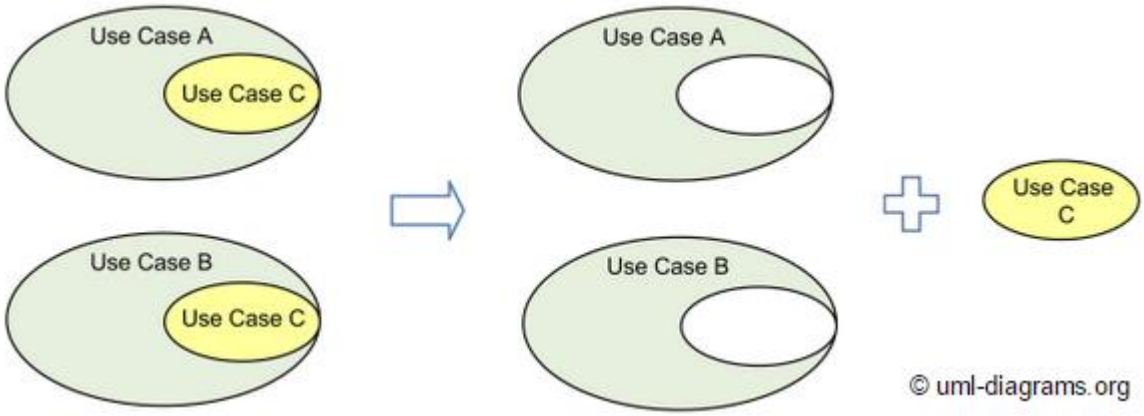
- «include»
- «extend»
- generalization

Include



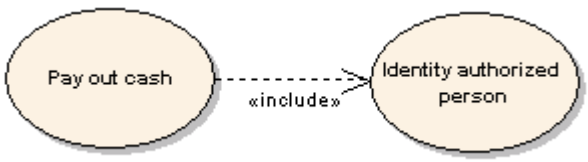
- defines that a use case contains the behavior defined in another use case
- behavior of the included use case is inserted into the behavior of the including use case
- included use case is **not optional**, and is **always required**

Include - common part



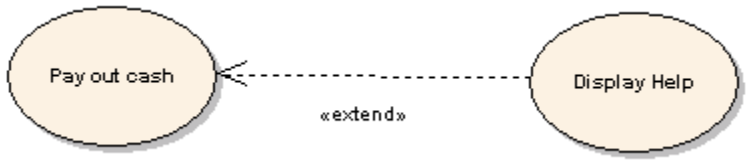
- include relationship is intended to be used when there are **common** parts of the behavior of two or more use cases
- this common part is then extracted to a separate use case, to be included by all the base use cases having this part in common

Include Example



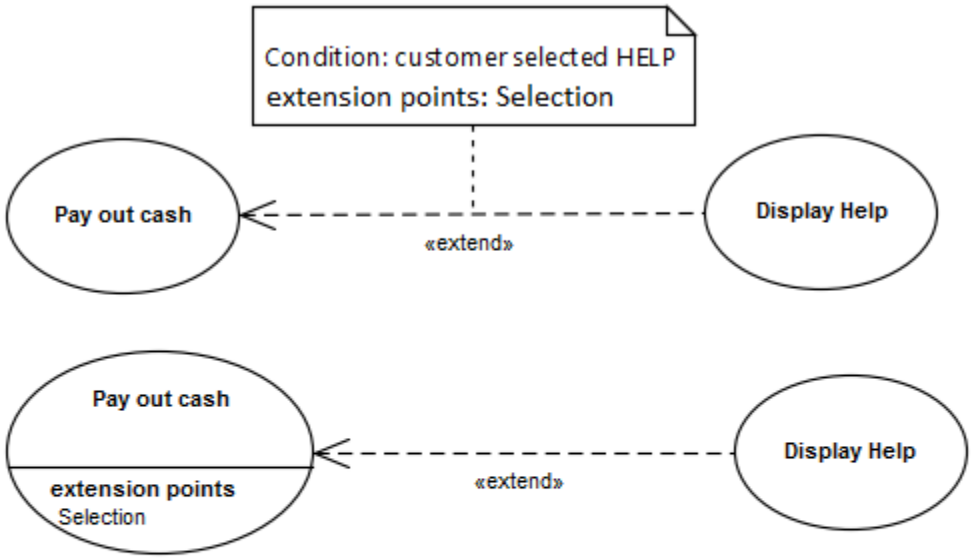
- basic flow of events is **incomplete** without the inclusion

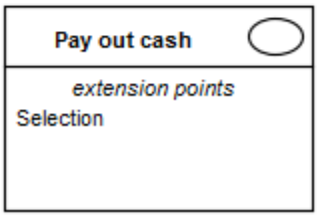
Extend⌘



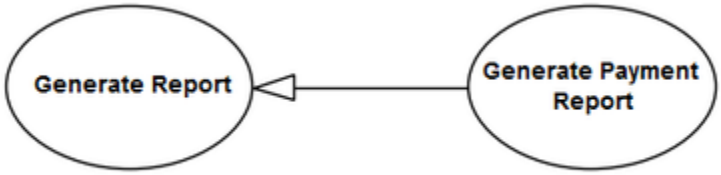
- a relationship from an extending use case to an extended use case
- specifies how and when the behavior defined in the extending use case can be inserted into the extended use case
- the extension takes place at **extension point**
- basic flow of events should still be **complete**

Extension Point



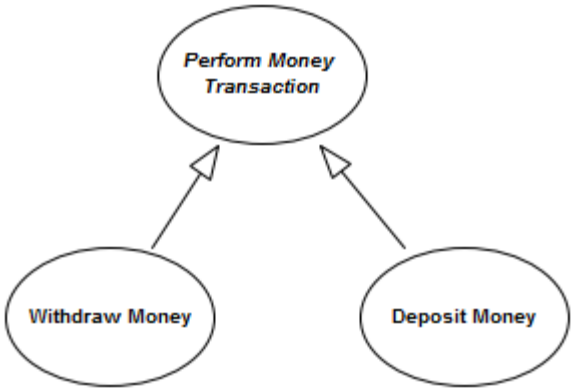


Generalization



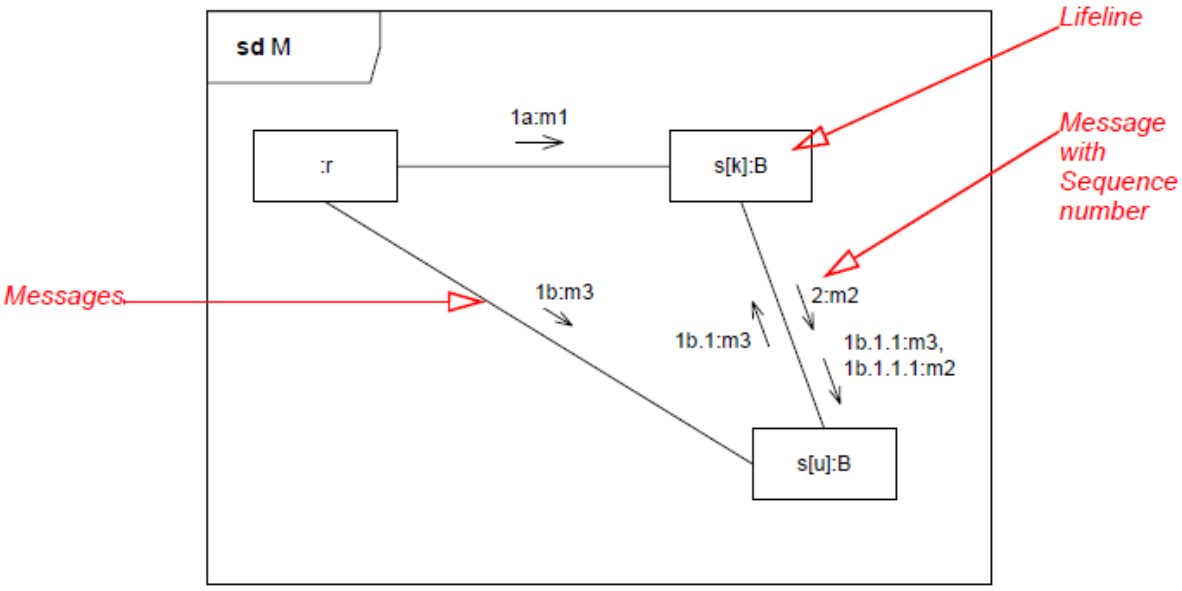
- the specific use case inherits the features of the more general use case

Abstract Use Case



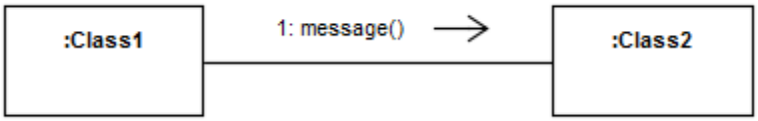
- does not provide a complete declaration and can typically not be instantiated
- is intended to be used by other use cases, e.g., as a target of generalization relationship
- the name of an abstract Use Case is shown in *italics*

Communication Diagram



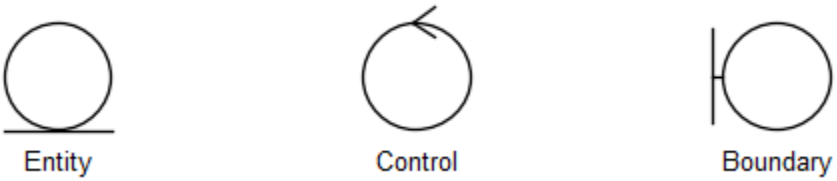
- Describes interactions between objects.
- The sequencing of Messages is given through a sequence numbering scheme.
- Communication Diagrams correspond to simple Sequence Diagrams.
- Lifeline represents an individual **participant** in the interaction.

Message



- A message is shown as a line from the sender to the receiver.
- The send and receive events may both be on the same lifeline.
- The form of the line or arrowhead reflects properties of the message:
  - Synchronous message (send message and suspend execution while waiting for response)
  -  Asynchronous message (send message and proceed immediately without waiting)
  -  Reply message (reply to synchronous message)

Entity-Control-Boundary Pattern

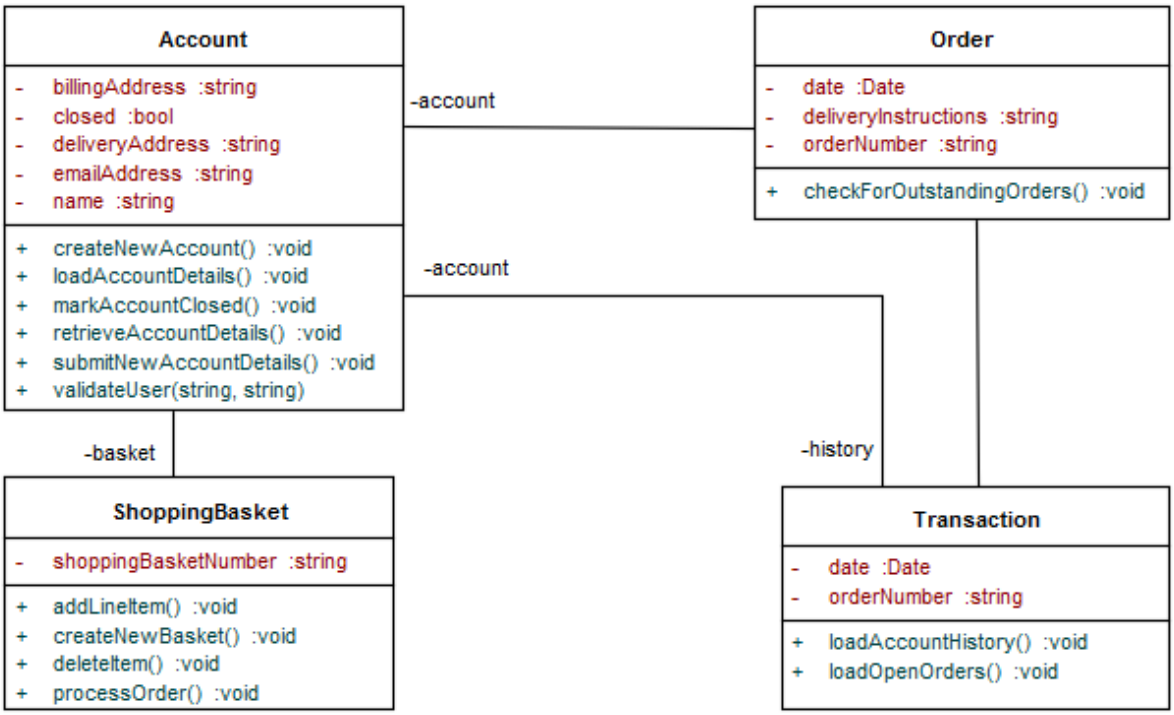


- **Entity**: objects representing system data
- **Control**: objects that manages the flow of interaction of the scenario (the logic of a system)
- **Boundary**: objects that interface with system actors

Entity-Control-Boundary Links

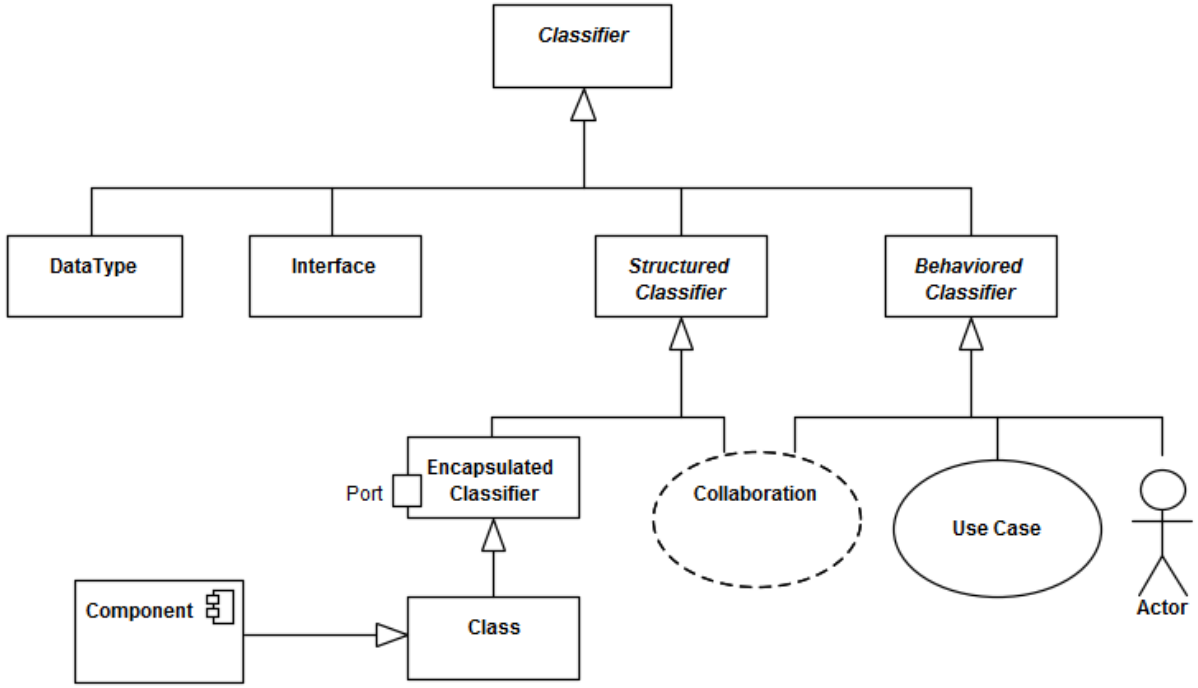
	Actor	**Boundary**	**Control**	**Entity**
Actor		OK		
Boundary	OK		OK	
Control		OK	OK	OK
Entity			OK	OK

Class Diagram



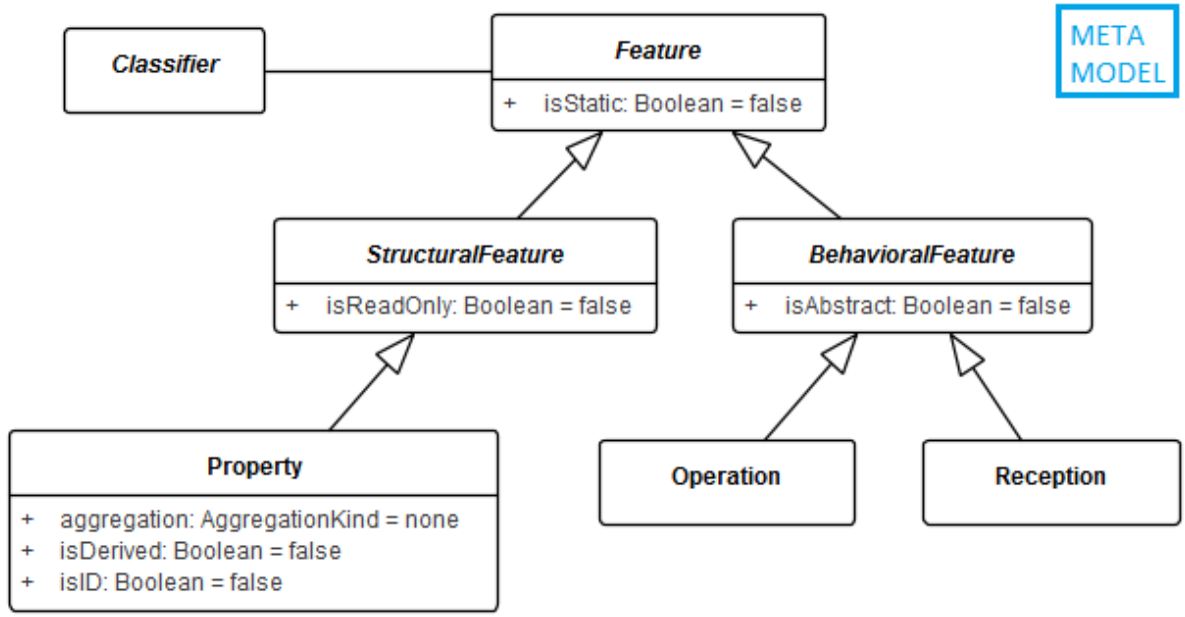
Describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.

Classifier



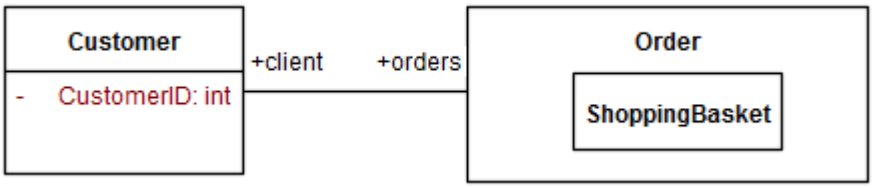
- an abstract base class
- describes a set of instances that have features in common

Feature



- declares a behavioral or structural characteristic of instances of classifiers
- **structural feature** describes a structure of an instance of a classifier (e.g. property)
- **behavioral feature** specifies an aspect of the behavior of classifier's instances (e.g. operation)

Property



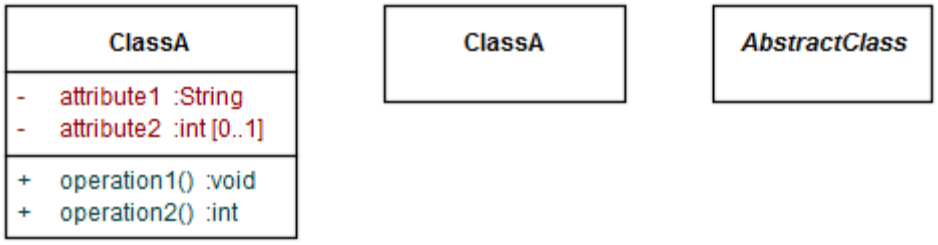
- Properties are Structural Features that represent:
  - the attributes of Classifiers (when a property is owned by a Classifier other than an Association)
  - the member Ends of Associations
  - the parts of Structured Classifiers.

Operation

- a behavioral feature of a classifier, which specifies name, type, parameters, and constraints
- can have preconditions and postconditions
- can have a type (the type of the return parameter)
- example:

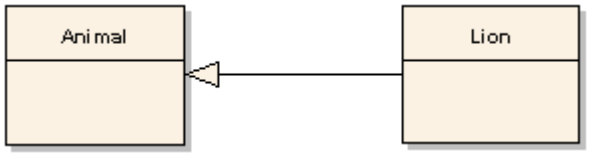
```
+ createWindow (location: Coordinates): Window
```

Class



- describes a set of objects that share the same specifications of features (attributes and operations)
- is a special classifier
- an object is an instance of a class

Generalization



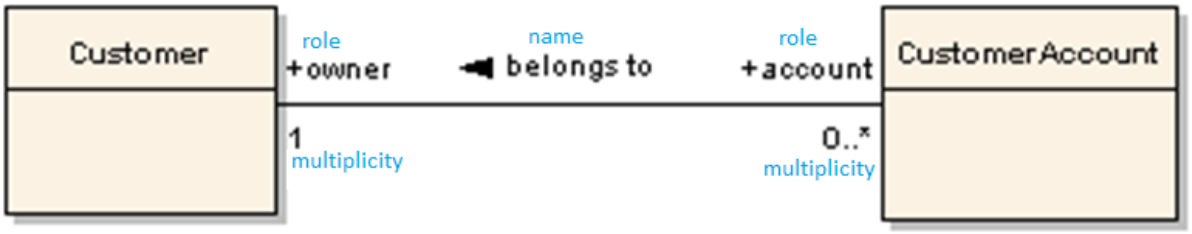
- relationship between a more general classifier and a more specific classifier
- each instance of the specific classifier is also an indirect instance of the general classifier
- the specific classifier inherits the features of the more general classifier

Generalization Example

BIRD

PIGEON

Association



- specifies a relationship between instances
- describes a set of tuples whose values refer to typed instances
- declares that there can be links between instances of the associated types

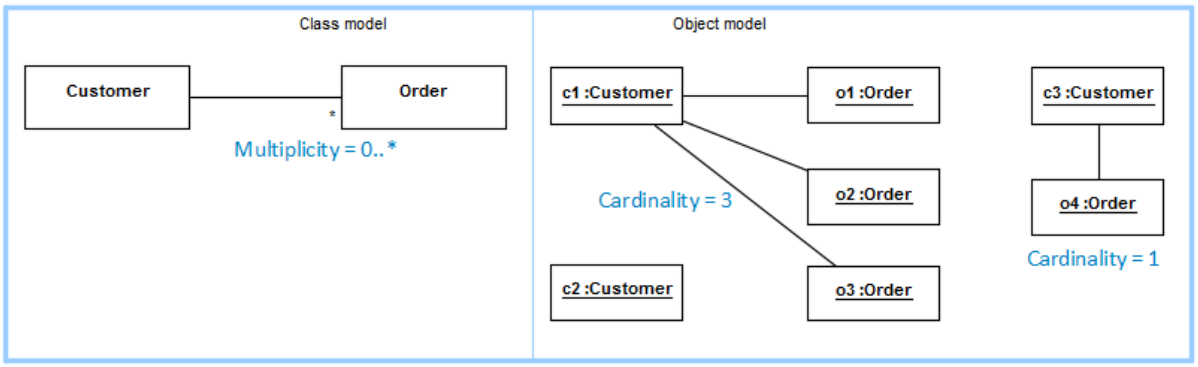
Multiplicity

- specifies how many objects of the opposite class an object can be associated with
- is a range of the minimum and maximum values
- syntax: **number** or **min..max**

Multiplicity	**Notation**
Zero	**0** or **0..0**
exactly one	**1**
zero or one	**0..1**

zero or more	**0..*** or *****
one or more	**1..***

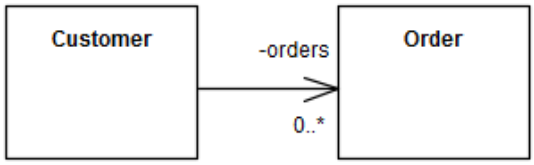
Multiplicity example

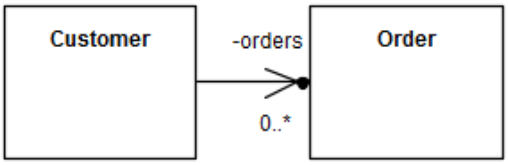


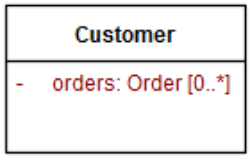
Multiplicity - order and uniqueness

- Multiplicity defines a specification of order and uniqueness of the collection elements.
- This option can specify whether the values should be **unique** and/or **ordered**.
  - **ordered**: the collection of values is sequentially ordered (default: not ordered)
  - **unique**: each value in the collection of values must be unique (default: is unique)

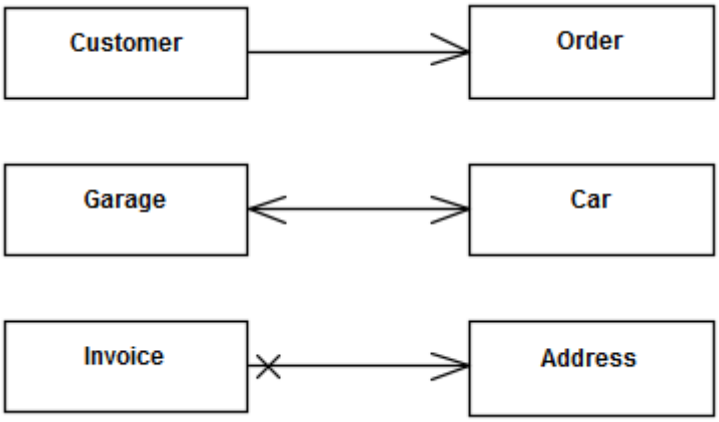
Property = End or Attribute





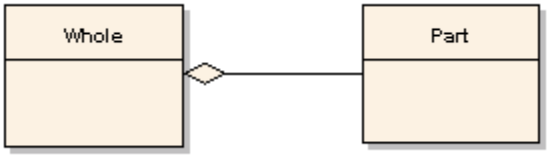


Navigability



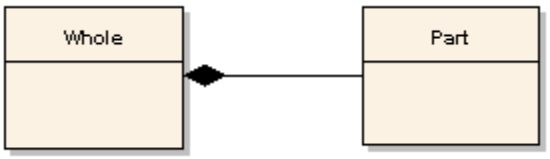
- specifies whether one object can be accessed directly from another

Aggregation



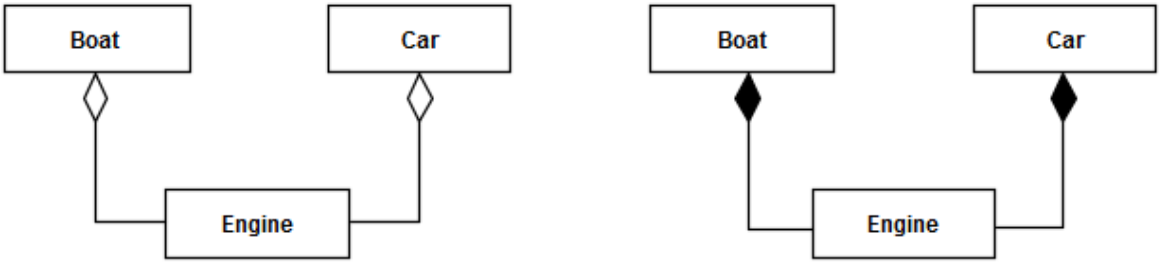
- shows how something (**whole**) is composed of **parts**
- parts can exist separately - can be **shared**
- precise semantics of aggregation varies by application area and modeler :)

Composition

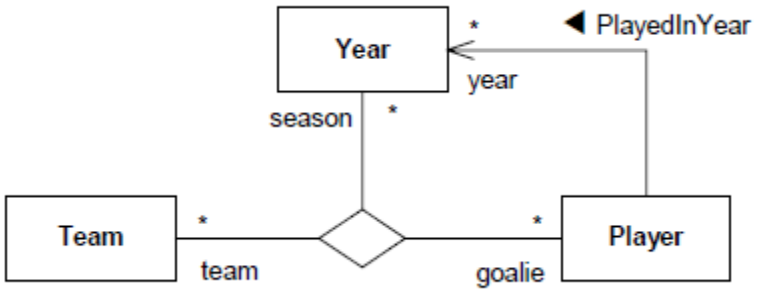


- a strict form of aggregation
- the whole is the owner of its parts
- parts **can not** be shared
- the existence of its parts depends on the whole

Aggregation / Composition Example

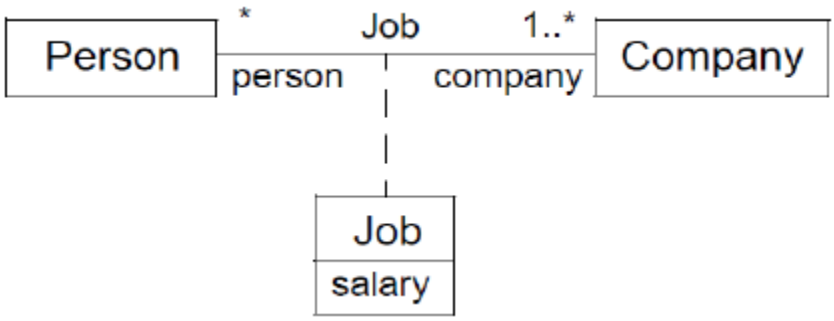


N-ary association



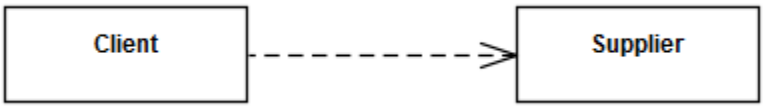
- if an association has more than two end points (here: ternary association)
- Notation: a rhombus is used as a connection point

Association Class



- Association Class ia a model element that has both Association and Class properties.
- It not only connects a set of Classifiers but also defines a set of Features that belong to the Association itself.

Dependency

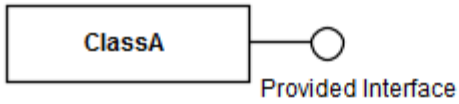


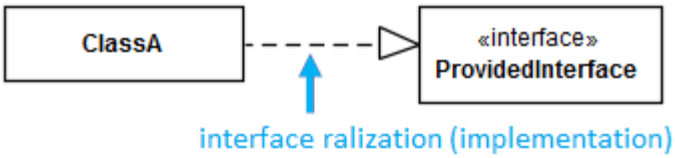
- a relationship that signifies that a model element(s) requires other model element(s) for their **specification** or **implementation**
- the complete semantics of the depending elements is dependent on the definition of the supplier element(s)
- the modification of the **supplier** may impact the **client** model elements
- the semantics of the **client** is not complete without the **supplier**
- the type of dependency can be specified by using a keyword or stereotype

Interface

- is a kind of classifier that represents a **declaration** of a set of coherent public features and obligations
- an interface specifies a **contract**; any instance of a classifier that realizes the interface must satisfy that contract
- it is just declaration - so it is **not instantiable**

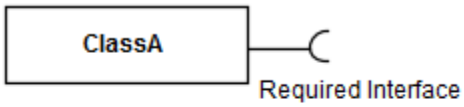
Provided Interface





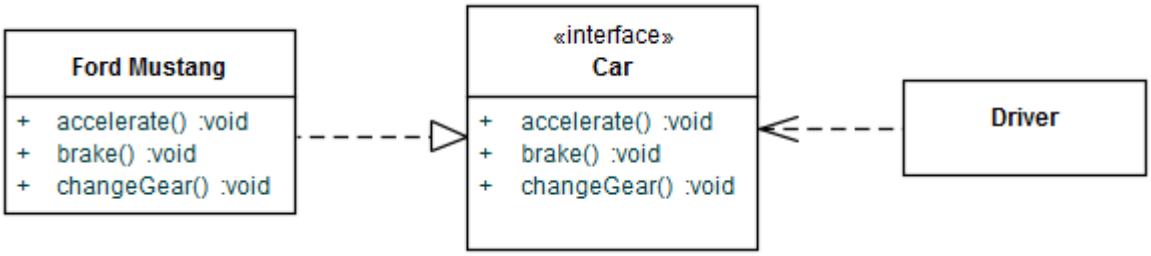
- an interface realized by a classifier is its **provided interface**
- represent the obligations that instances of that classifier have to satisfy

Required Interface



- specify services that a classifier **needs** in order to perform its function

Interface Example



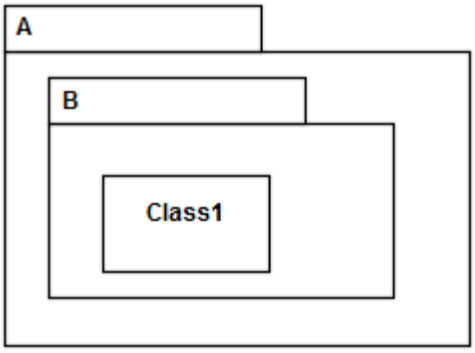
Named Element

- represents an element that **may** have a name and a visibility

Visibility Kind	Notation
public	+

private	-
protected	#
package	~

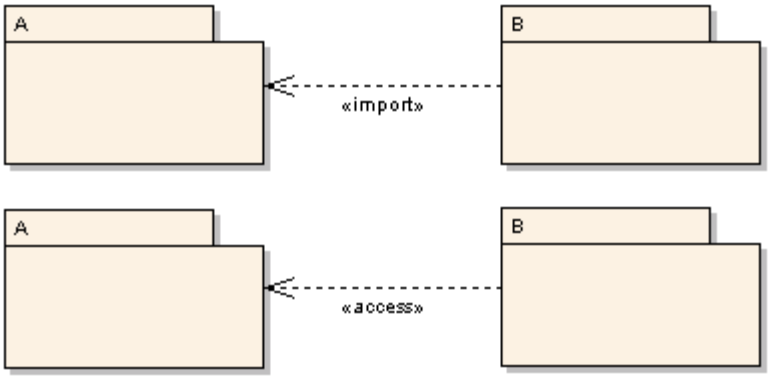
Package



- used to group elements, and provides a namespace for the grouped elements
- qualified name:

```
package name::element name
```

Package Import, Access

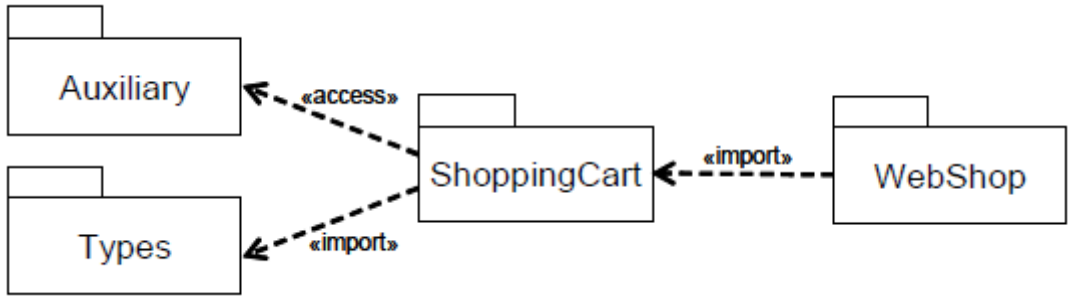


A package import is defined as a directed relationship that identifies a package whose members are to be imported by a namespace.

Two types:

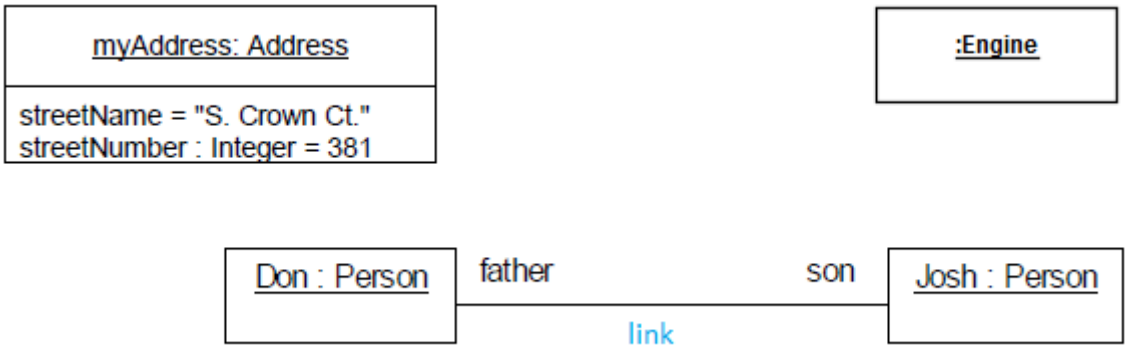
- «import» for a **public** package import
  - transitive: if A imports B and B imports C then A indirectly imports C
- «access» for a **private** package import
  - intransitive

Package Import Example



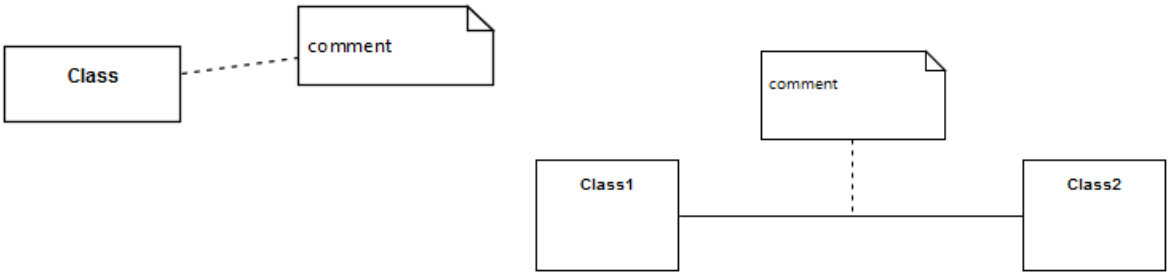
- elements in Types are imported to ShoppingCart, and then further imported to WebShop
- elements of Auxiliary are only accessed from ShoppingCart, and cannot be referenced from WebShop

Instance



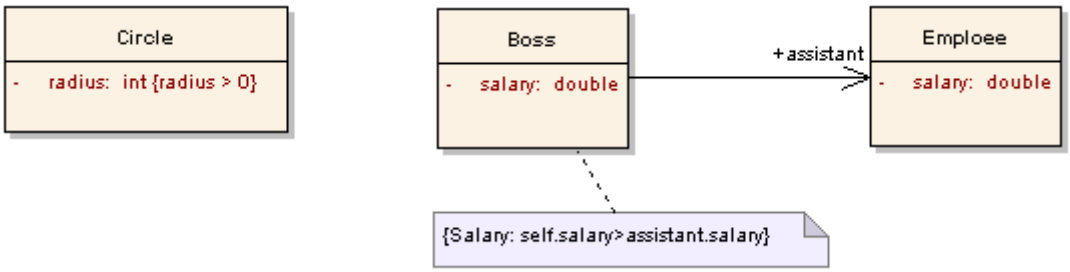
- is a concrete instance in the modeled system
- instance = object

Comment



- a textual annotation that can be attached to an element
- may contain information that is useful to a modeler
- can be attached to more than one element

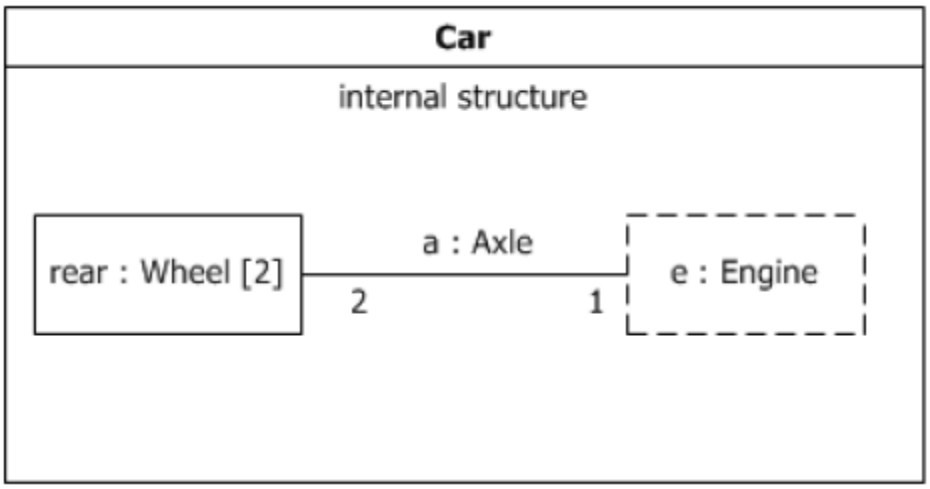
Constraint



- condition or restriction related to an element
- it must always be true
- can be in a formal (OCL) or in a human language
- syntax:

{ [name :] boolean expression **}**

Composite Structure Diagram



- Describes the internal structure of a classifier and the collaborations that this structure makes possible.
- **Structured Classifiers** may contain an internal structure of connected elements each of which plays a role.
- **Encapsulated Classifier** extends Structured Classifier with the ability to own Ports, a mechanism for isolating an Encapsulated Classifier from its environment.

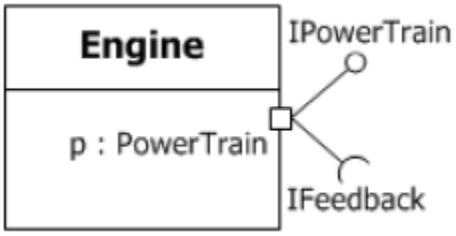
Role and Part

- Role represents a participant within the internal structure of a Structured Classifier.
- Part is special kind of Role (owned using composition).
- A part may be shown by graphical nesting of a box symbol with a **solid** outline representing the part.
- A role that is not a composition may be shown by graphical nesting of a box symbol with a **dashed** outline.

Connector

- A Connector specifies links between two or more instances playing roles within a structured classifier.

Port

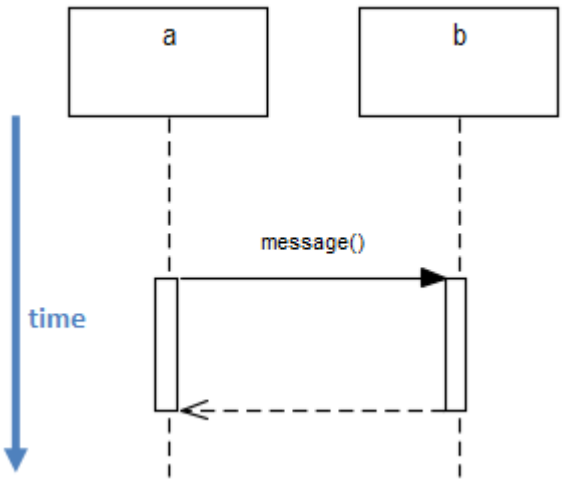


- A **Port** represents interaction point through which an Encapsulated Classifier communicates with its environment.

Interaction

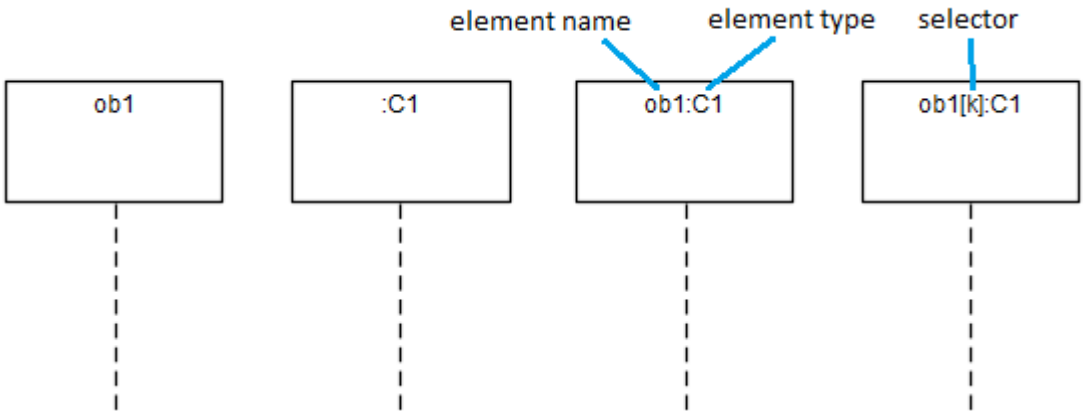
- An **interaction** is a unit of behavior that focuses on the visible exchange of information between participants (limited in time).
- 4 interaction diagrams in UML 2.0:
  - **Sequence diagrams** - sequence in which messages are exchanged
  - **Communication diagrams** - relationship between the participants
  - **Timing diagrams** - state changes of the participants relative to the time
  - **Interaction overview diagrams** - order of interactions in an activity-like notation

Sequence Diagram



- Participants are represented by a rectangle and a dashed line (lifeline)
- Messages are represented by arrows between lifelines
- Time runs from top to bottom
- The entire diagram represents one interaction

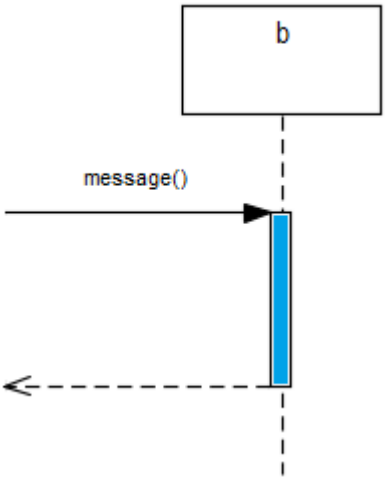
Connectable Element



Connectable Element

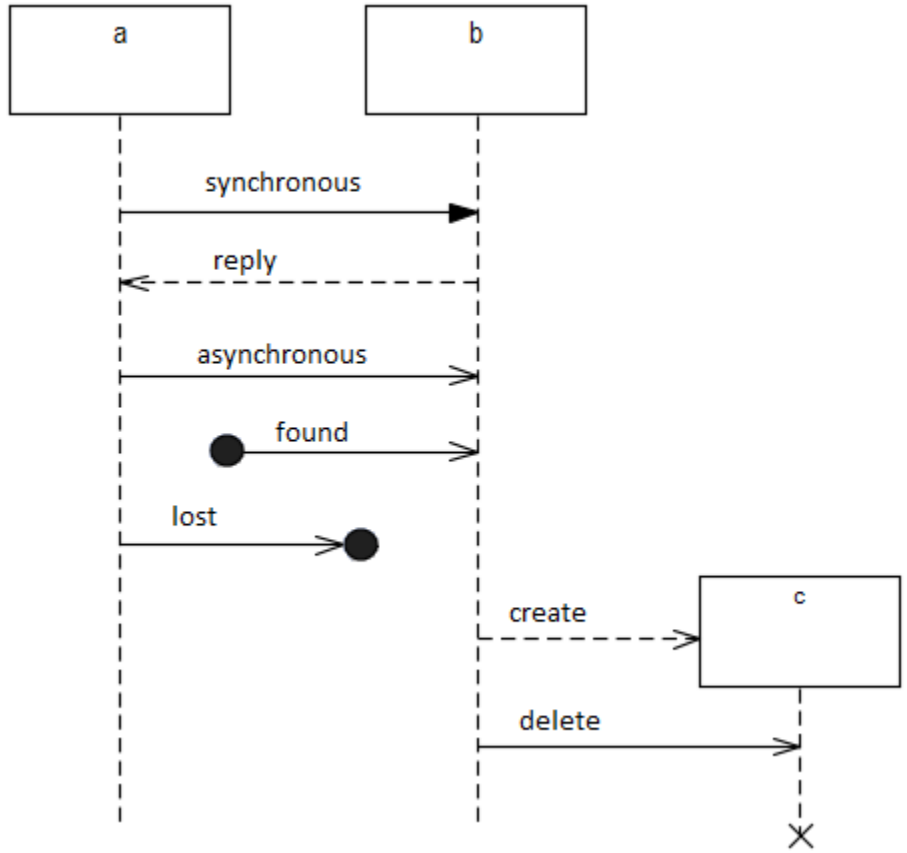
- a set of instances
- can be considered as objects of the specified type
- lifelines (participants) represent connectable elements

Execution Specification (Execution Occurrence)



- Specification of the execution of a unit of behavior within the lifeline.
- Sending and receiving messages determine start and end of execution specification.

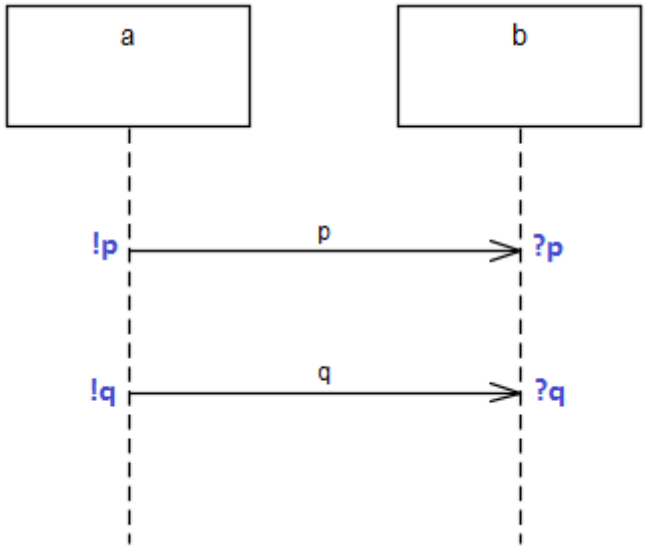
Message



Message Types

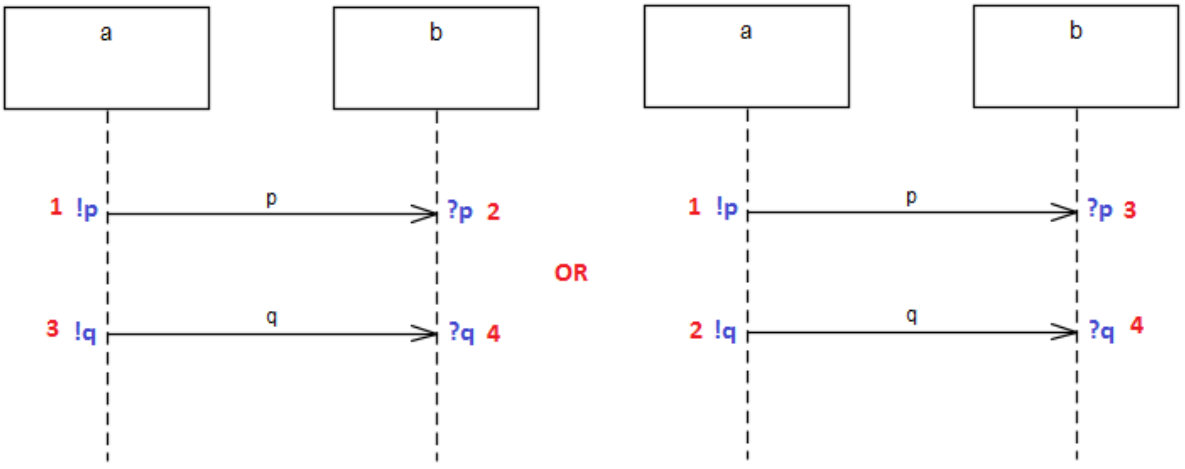
- **Synchronous message** (filled arrowhead):
  - caller waits until called behavior terminates
  - **reply message** is represented by dashed line with either an open or filled arrow head
- **Asynchronous message** (open arrowhead):
  - caller doesn't wait but continues after call, no reply
- **Found message** (open arrowhead originating from a filled circle):
  - receiver known, sender not known
- **Lost message** (open arrowhead pointing to a filled circle)
  - sender known, receiver not known
- **Create message** (dashed line with open arrowhead pointing to header of lifeline):
  - new lifeline is created at this point in the diagram
- **Delete message** (open arrowhead):
  - object destruction, stop (destruction of an object is represented by a cross on the lifeline)

Event Occurrences



- send event is denoted by an exclamation mark (**!p**)
- receive event is denoted by question mark (**?p**)
- A valid sequence for the interaction: **< !p, ?p, !q, ?q >**
- **It's not the only valid sequence!**

Event Occurrences valid trace



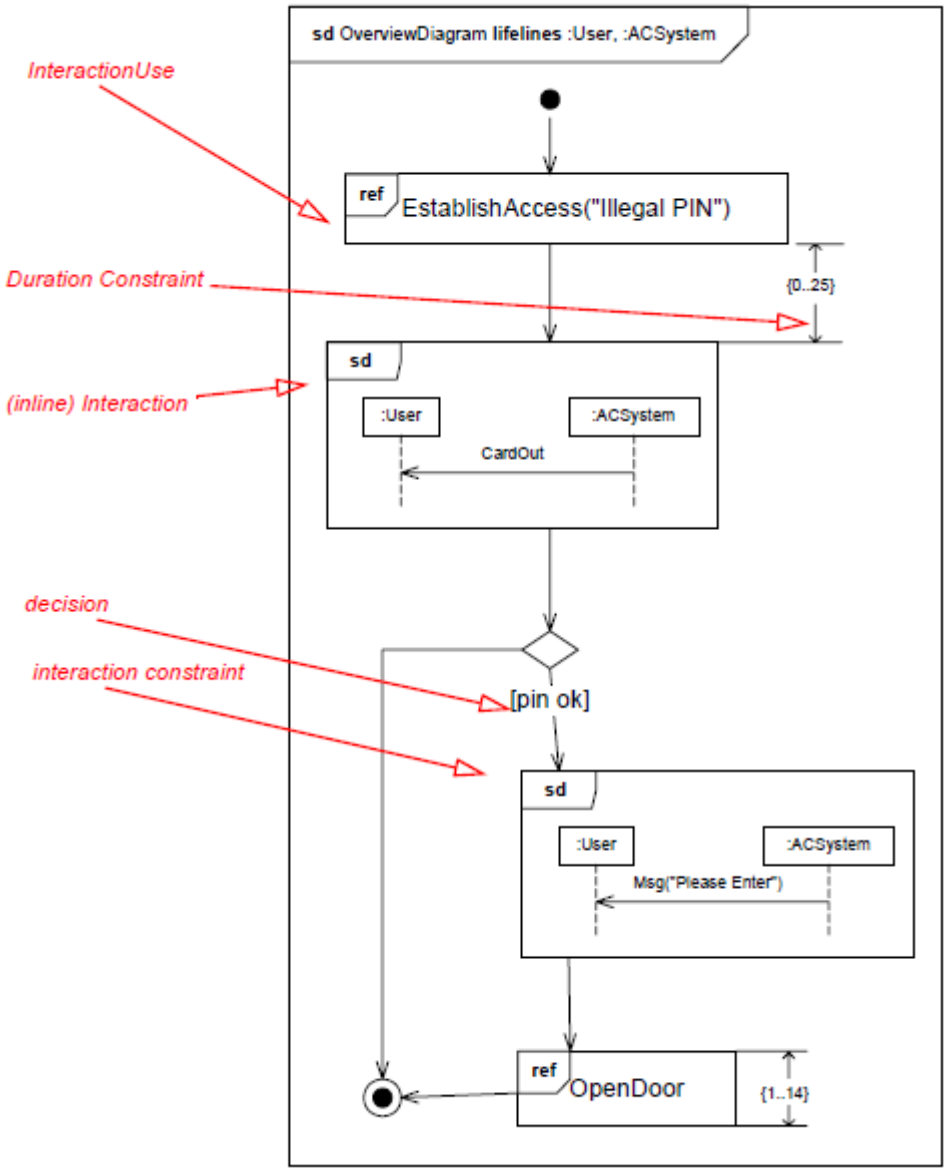
Event Occurrences Rule

- **Send** event **before receive** event
- The order of events along one lifeline is **relevant**.
- The position of one event relative to an event on another lifeline is **insignificant**.

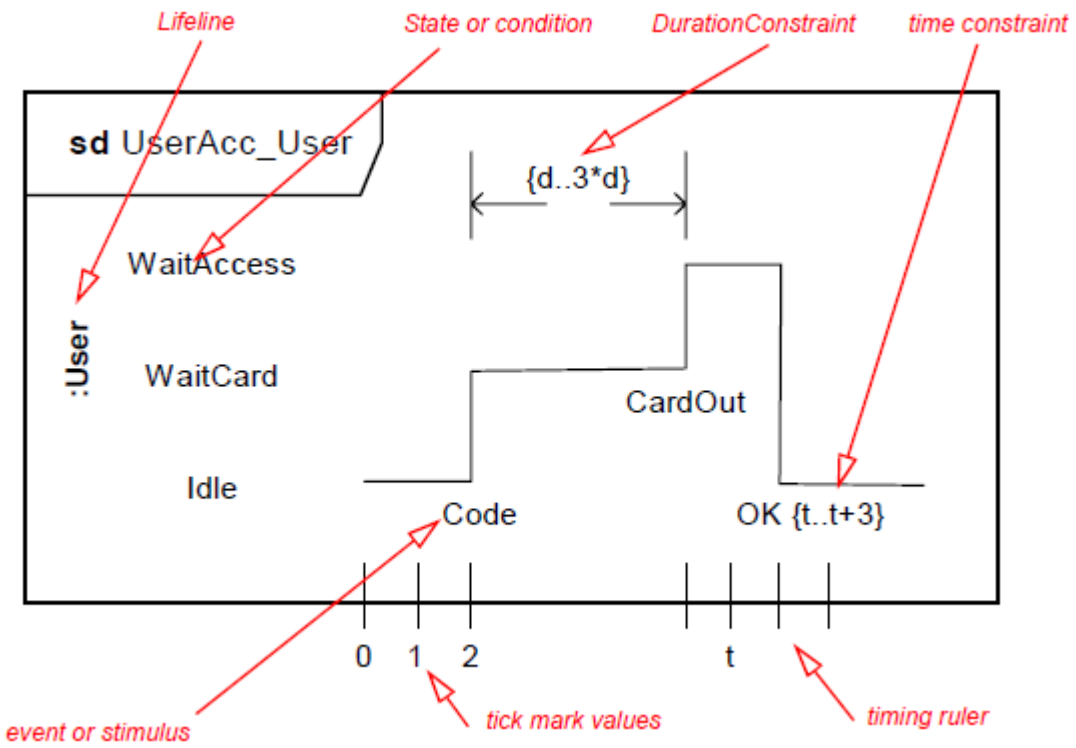
Interaction Overview Diagram

- Interaction Overview Diagram is a variant of Activity Diagram.
- Interaction Overview Diagrams focus on the overview of the flow of control where the nodes are **Interactions** or **InteractionUses**.

Interaction Overview Diagram

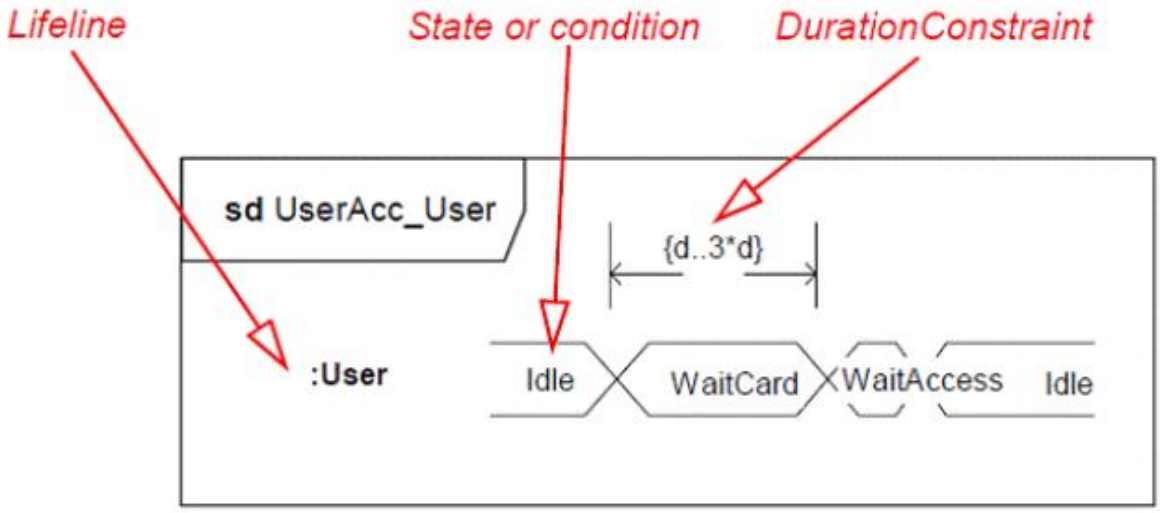


Timing Diagram



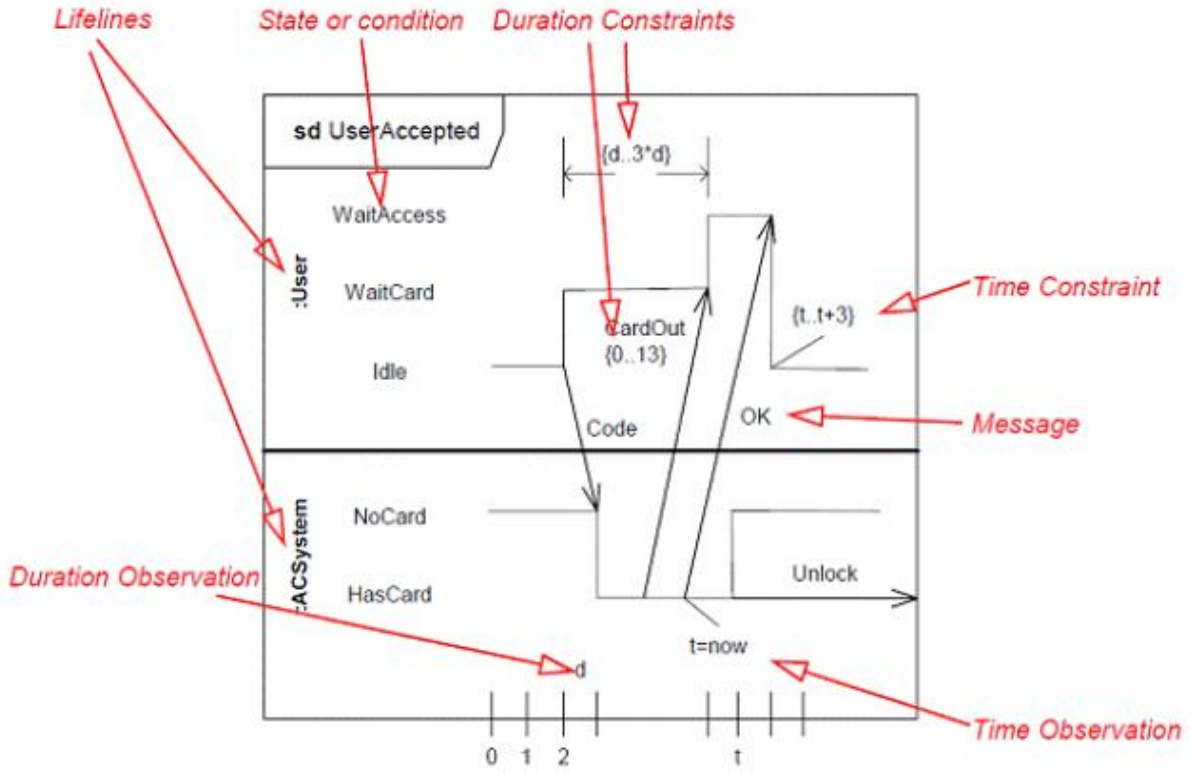
Timing diagrams show change in state of a structural element over time.

General Value Lifeline

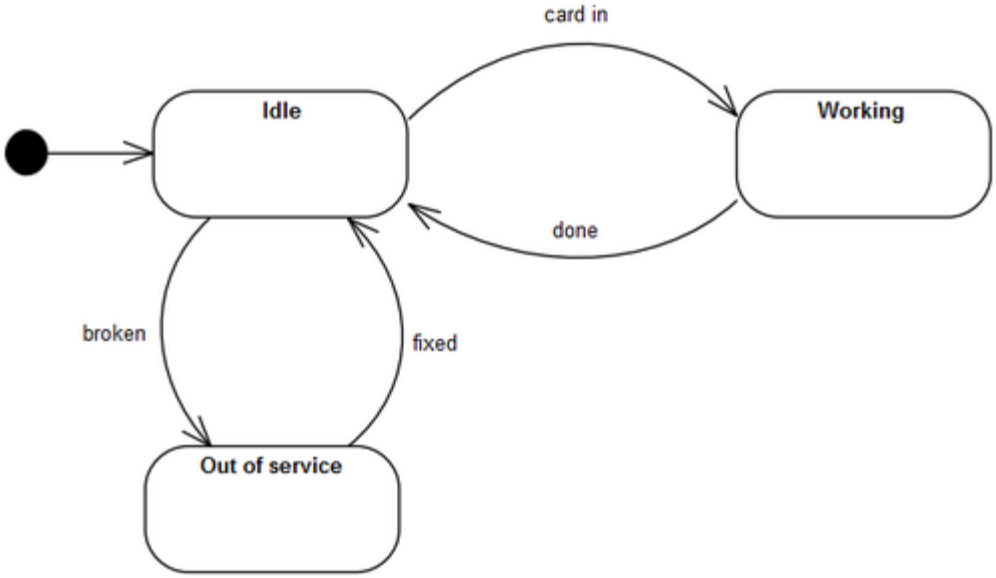


- Shows the value of the connectable element as a function of time.
- Value is explicitly denoted as text.
- Crossing reflects the event where the value changed.

More Lifelines



State Machine Diagram



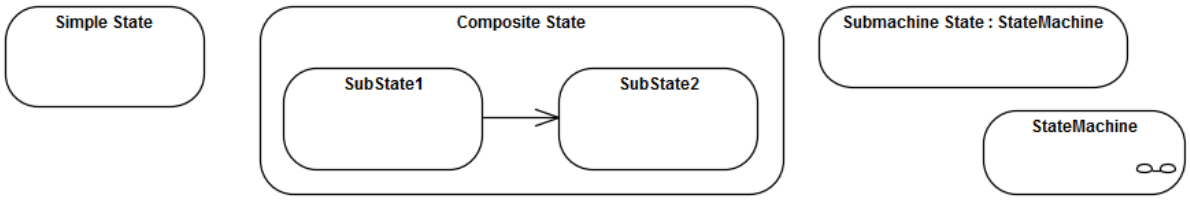
- State machine diagram is a behavior diagram which shows **discrete behavior** of a part of designed system through finite state transitions.
- State machine diagrams can also be used to express the valid interaction sequences (**protocols**) for parts of a system.
- Two kinds of state machines:
  - behavior state machine,
  - protocol state machine.

Behavior State Machine

Vertex

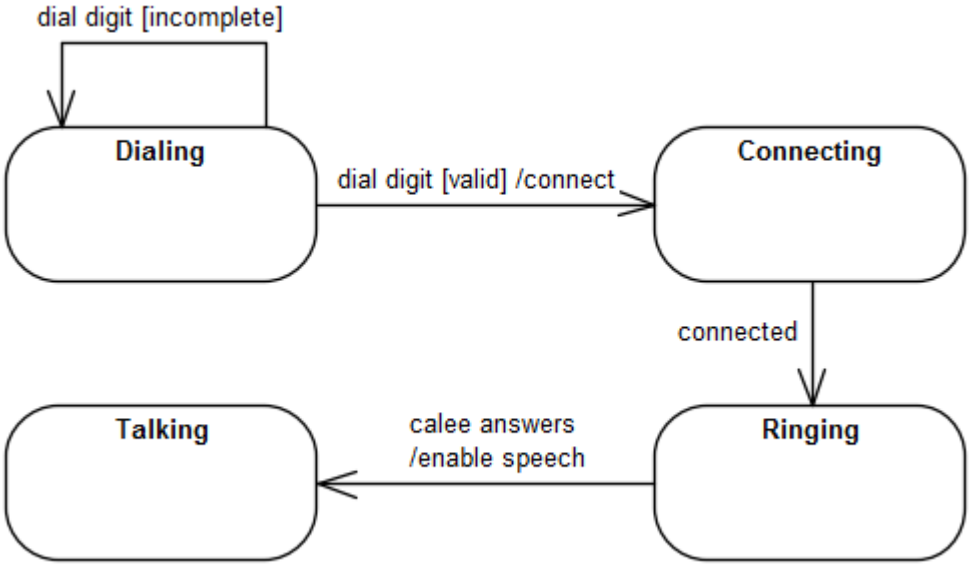
- Vertex is named element which is an abstraction of a node (State or Pseudostate) in a state machine graph.
- It can be the source or destination of any number of transitions.

State⌘



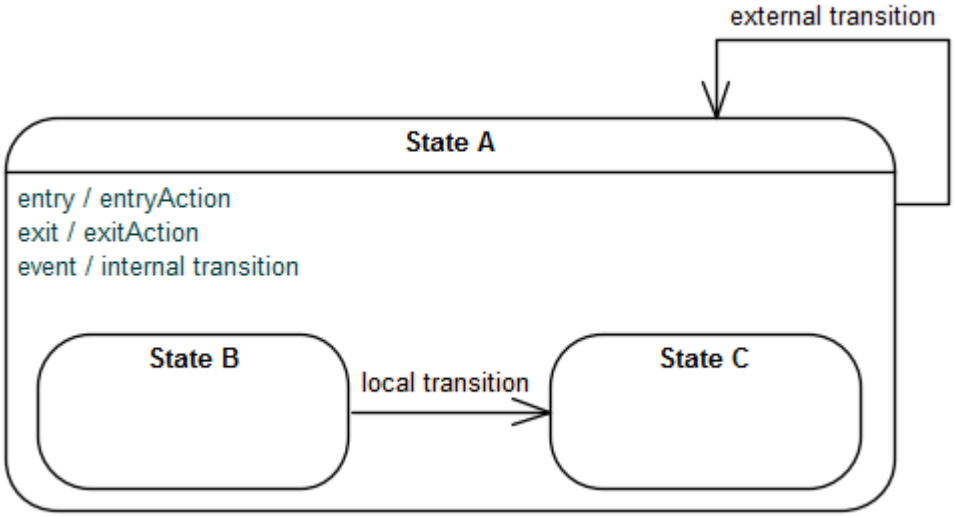
- State models a situation during which some invariant condition holds.
- In most cases this condition is not explicitly defined.
- Kinds of States:
  - Simple State (has no internal vertices or transitions)
  - Composite State (contains at least one region)
  - Submachine State (refers to an entire State Machine)

Transition⌘



- Directed relationship between a source vertex and a target vertex.
- Syntax: **[<trigger> [‘,’ <trigger>]* [‘[‘ <guard>’]’] [‘/’ <behavior-expression>]]**
  - **trigger** specifies events that may cause state transition,
  - **guard** is a boolean expression,
  - optional **behavior-expression** is an expression specifying the effect (is executed when the transition fires)

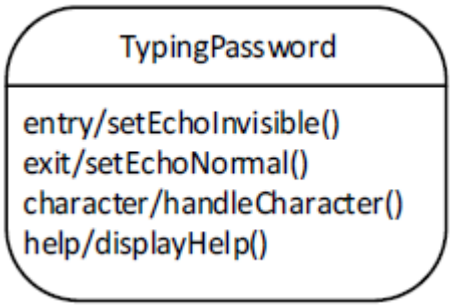
Transition Kind



The semantics of a Transition depend on its relationship to its source Vertex:

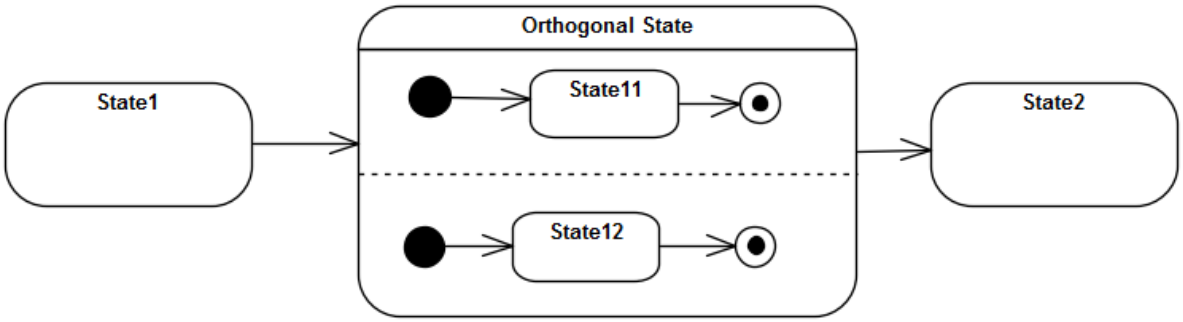
- **external** - transition exits its source vertex (executes exit action of source state)
- **local** - transition does not exit its containing State (exit action of the containing state will not be executed)
  - Target vertex may be different from its source Vertex.
  - Local transition can only exist within a composite State.
- **internal** is a special case of a local transition with the same source and target states,
  - the state is never exited (and not re-entered)
  - no exit or entry actions are executed

Internal Behaviors



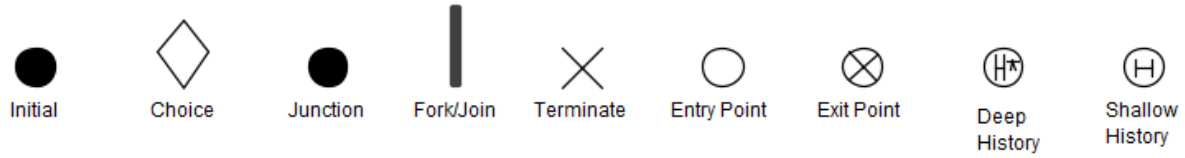
- **entry** - a Behavior which is performed upon entry to the State (entry Behavior).
- **do** - an ongoing Behavior (doActivity Behavior) that is performed as long as the modeled element is in the State or until the computation specified by the expression is completed.
- **exit** - a Behavior that is performed upon exit from the State (exit Behavior).

Completion Transitions, completion events



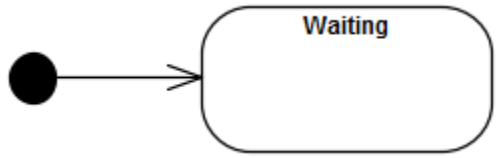
- Completion transition has an implicit trigger (may have a guard).
- Completion event enables this trigger if all actions associated with the source state have completed execution:
  - In simple states: a completion event is generated when the associated entry and doActivity actions have completed executing.
  - In composite or submachine states: a completion event is generated when:
    - all internal activities have completed execution, and
    - if the state is a composite state, all its orthogonal regions have reached a final state, or
    - if the State is a submachine State, the submachine State Machine execution has reached a final state.

Pseudo State



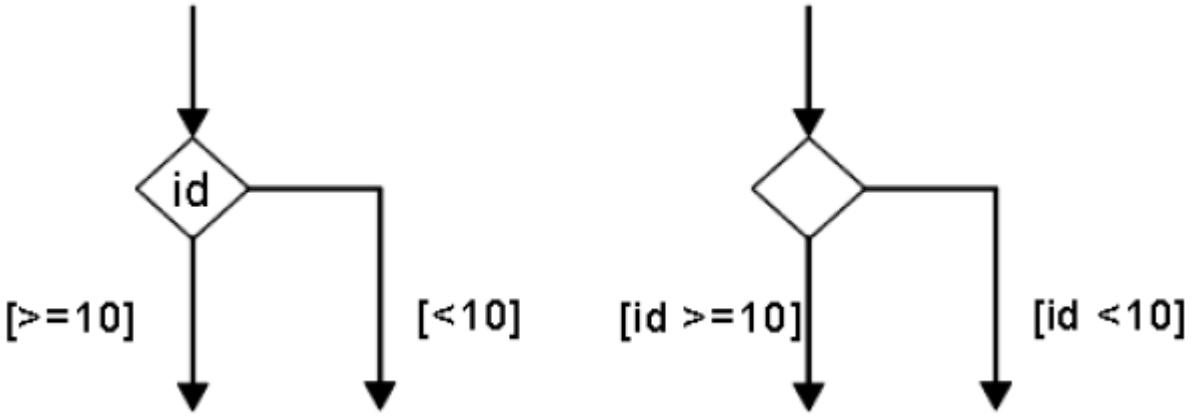
- Pseudostate is an abstract vertex that describes different types of transient vertices in the state machine graph.
- Pseudostates are transitive - a compound transition execution simply passes through them, arriving on an incoming Transition and leaving on an outgoing Transition **without pause**.

Initial ⌘



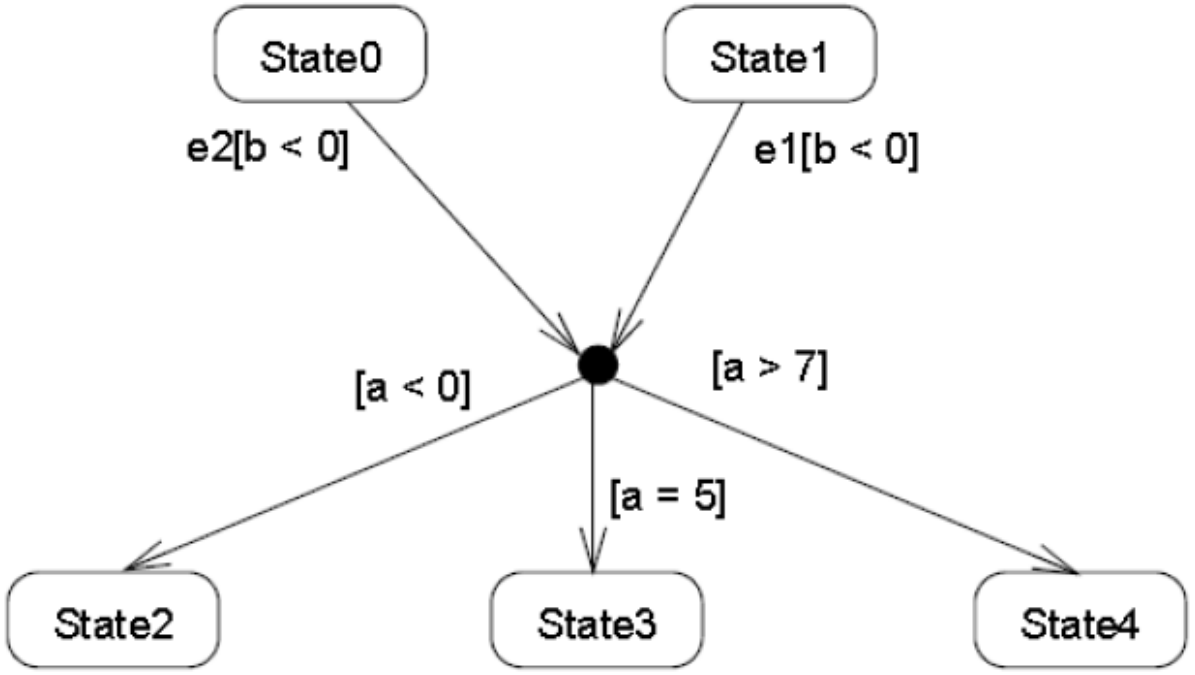
- Initial Pseudostate represents a starting point for a Region.
- It is the source for at most one Transition, which may have an associated effect Behavior, but not an associated trigger or guard.
- There can be at most one initial Vertex in a Region.

Choice



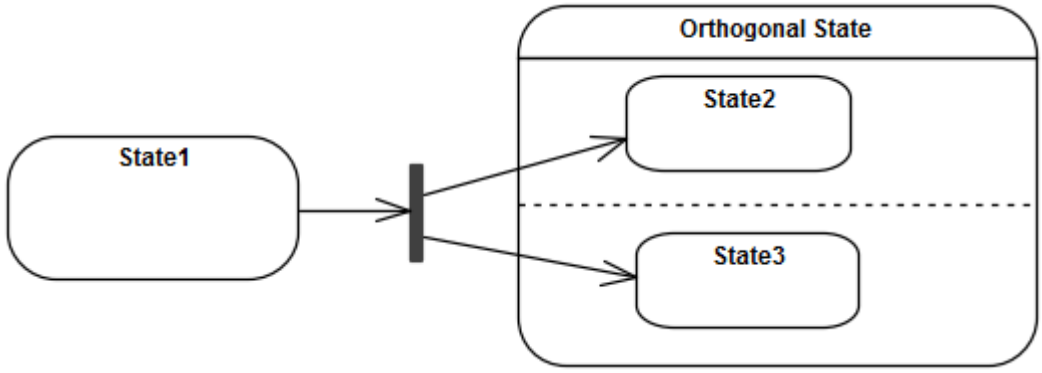
- Choice is used to realize a dynamic conditional branch.
- It allows splitting of compound transitions into multiple alternative paths.
- The decision on which path to take may depend on the results of Behavior executions performed in the same compound transition prior to reaching the choice point.
- If more than one guard evaluates to true, one of the corresponding Transitions is selected.
- If none of the guards evaluates to true, then the model is considered ill formed

Junction ⌘



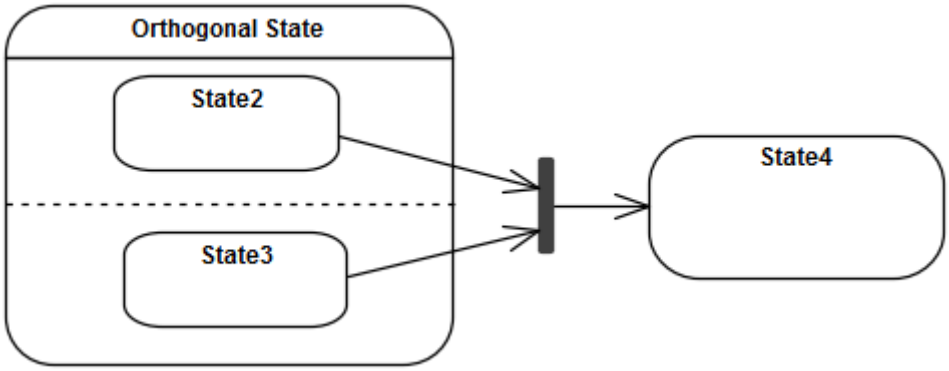
- Junction can be used to merge multiple incoming Transitions into a single outgoing Transition or,
- it can be used to split an incoming Transition into multiple outgoing Transition segments with different guard Constraints.
- Such guard Constraints are evaluated before any compound transition containing this Pseudostate is executed, which is why this is referred to as a static conditional branch.

Fork



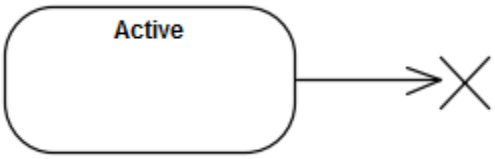
- Splits an incoming Transition into two or more Transitions terminating on Vertices in orthogonal Regions of a composite State.
- The Transitions outgoing from a **Fork** cannot have a guard or a trigger.

Join



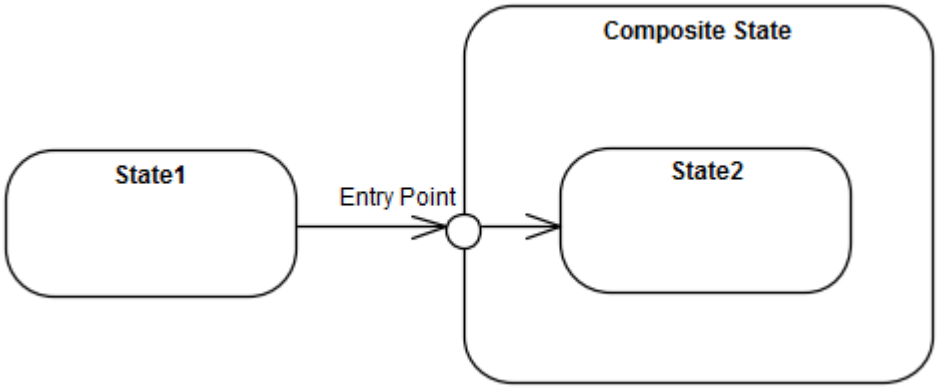
- Common target Vertex for two or more Transitions originating from Vertices in different orthogonal Regions.
- Transitions terminating on a **Join** cannot have a guard or a trigger.

Terminate



- Execution of the State Machine is terminated immediately.
- The State Machine does not exit any States nor does it perform any exit Behaviors. Any executing doActivity Behaviors are automatically aborted.

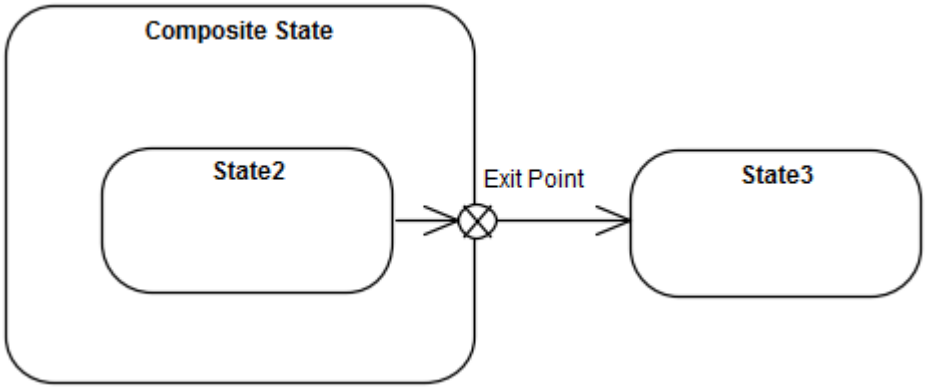
Entry Point



- Represents an entry point for a State Machine or a composite State that provides encapsulation of the insides of the State or StateMachine.
- In each Region of the StateMachine or composite State owning the Entry Point, there is at most a single Transition from the entry point to a Vertex within that Region.

If the owning State has an associated entry Behavior, this Behavior is executed before any behavior associated with the outgoing Transition. If multiple Regions are involved, the entry point acts as a fork Pseudostate.

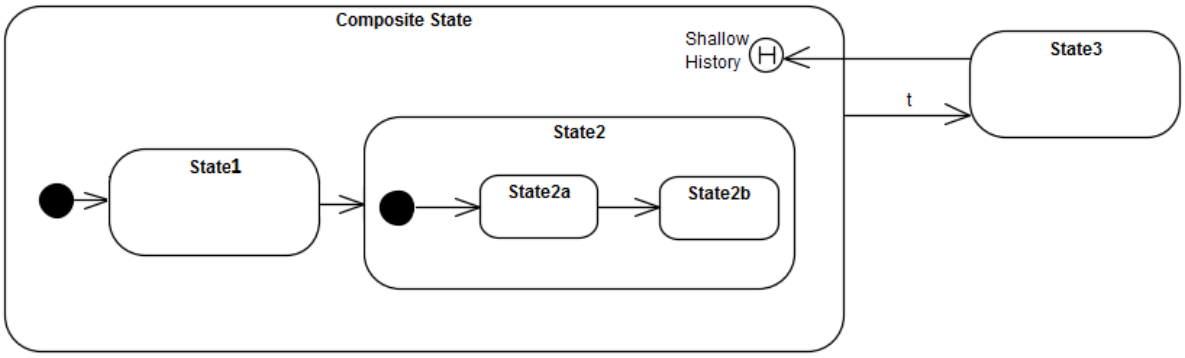
Exit Point



- Represents an exit point of a StateMachine or composite State that provides encapsulation of the insides of the State or State Machine.
- Transitions terminating on an exit point within any Region of the composite State or a State Machine referenced by a submachine State implies exiting of this composite State or submachine State (with execution of its associated exit Behavior).

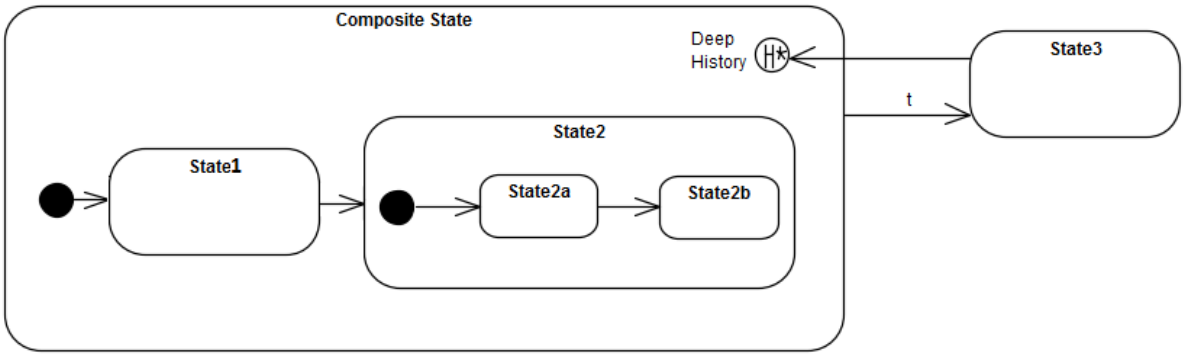
If multiple Transitions from orthogonal Regions within the State terminate on this Pseudostate, then it acts like a join Psuedostate.

Shallow History



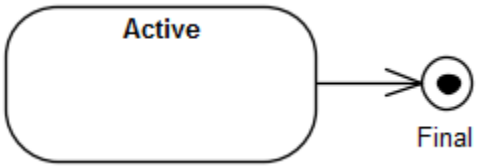
- Represents the most recent **active substate** of its containing Region, but not the substates of that substate.
- Transition terminating on this Pseudostate implies restoring the Region to that substate.
- A single outgoing Transition from this Pseudostate may be defined terminating on a substate of the composite State.
- Shallow History can only be defined for composite States and, at most one such Pseudostate can be included in a Region of a composite State.

Deep History



- Represents the most recent **active state configuration** of its owning Region.
- Transition terminating on this Pseudostate implies restoring the Region to that same state configuration.
- The entry Behaviors of all States in the restored state configuration are performed in the appropriate order starting with the outermost State.
- Deep History Pseudostate can only be defined for composite States and, at most one such Pseudostate can be contained in a Region of a composite State.

Final State



- **Final State** is a special kind of State (not Pseudo State) signifying that the enclosing Region has completed.

- Transition to a Final State represents the completion of the behaviors of the Region containing the Final State.

Event Processing for State Machines

The run-to-completion paradigm (in general)

- State Machine will perform its initialization during which it executes an initial transition, after which it enters a wait point.
- A wait point is represented by a stable state configuration. It remains thus until an Event stored in its event pool is dispatched.
- This Event is evaluated and a single State Machine step is executed.
- A step involves executing a transition and terminating on a stable state configuration (i.e., the next wait point).
- This cycle then repeats until the State Machine completes its Behavior.
- Event occurrences are detected, dispatched, and processed by the State Machine execution, one at a time.