UML Start-Up Training

UB1

WILLERT.
pioneers in embedded software engineering

UML

This training course is designed with the intention to teach UML in not longer than a day. Yes UML is a complicated language with a lot of diagrams, model elements, a complex syntax and semantic; and most people claim that it takes a long time to learn that language.

But:
if we concentrate our efforts to the most important part of the language you will be able to become familiar with it in a day.

And:
use it frequently, then you'll get the less important things by use and by reading a good book: the standard (visit the web pages from OMG and you'll manage to download the standard, it's free of charge).

Index

History

The "OO tower" of Babel

In the early Nineties there was a vast amount of competing object oriented modeling approaches together with corresponding kinds of diagrams and CASE environments which, sometimes more and sometimes less, differ from each other.

Examples:

- Object-Oriented Analysis (OOA): Coad-Yourdon 1991
- Object-Oriented Design (OOD): Booch 1991
- Object Modeling Technique (OMT): Rumbaugh 1991
- OO Software Engineering (OOSE): Jacobson 1992
- Fusion: Coleman 1994

The Idea: Unified Modeling Language

The idea of Grady Booch:

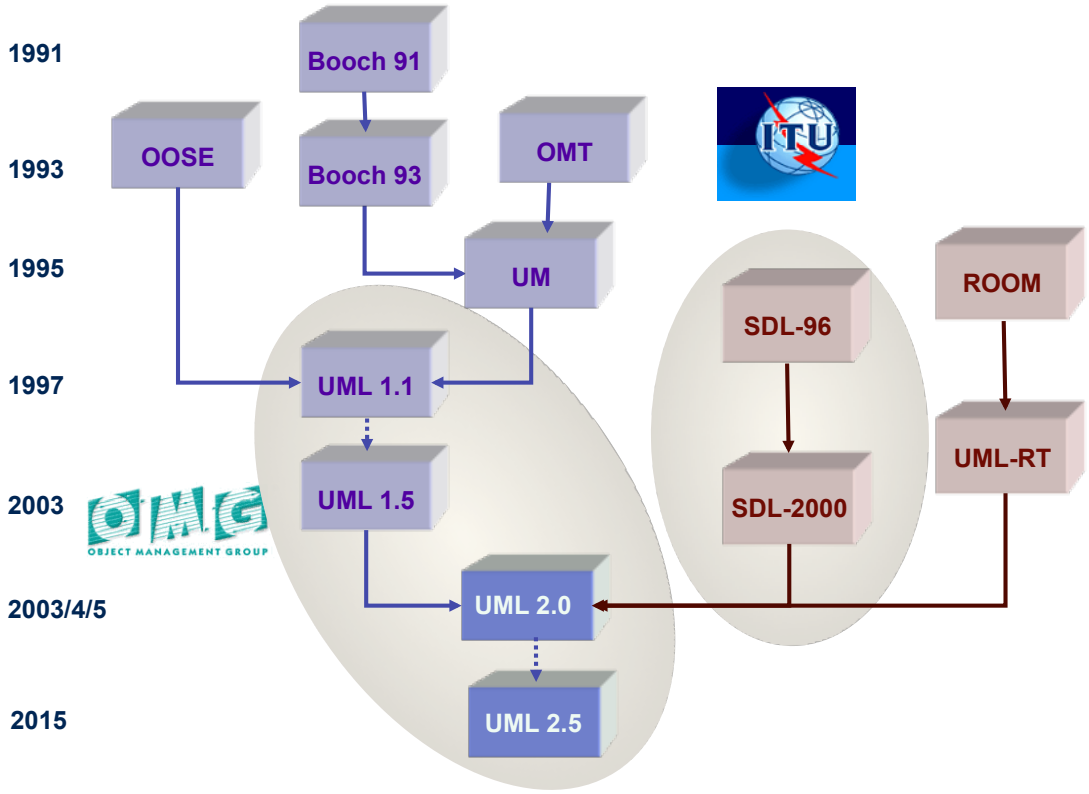
A number of OO modeling dialects is replaced by a standardized language in order to bury unnecessary ditch (and religious) wars in the OO environment unite advantages of different modeling approaches relieve potential users of being spoilt for choice provide for the preconditions for data exchange between OO tools enable close integration of OO tools reduce dependencies of users of one supplier release tool manufacturers of the support of many approaches.

Some Milestones

Things started happening very fast

– or – A short story about the „Unified Modeling Language"…*

- October 1994: Rumbaugh changes from GE to Rational (Booch)
- October 1995: Unified Method 0.8 = OMT (Rumbaugh) + OOD (Booch)
- Autumn 1995: Jacobson changes from Objective to Rational – this means the "Three amigos" (Rumbaugh, Booch, Jacobson) are united
- June 1996: Unified Modeling Language (UML) 0.9 with Use-Cases of Jacobson and support by DEC, HP, IBM, Microsoft, Oracle, …
- January 1997: UML 1.0 is presented for standardization at OMG = Object Management Group
- September 1997: UML 1.1 with extended description and supplementations, especially Object Constraint Language (OCL) = predicate logical conditions as annotations regarding class diagrams
- November 1997: OMG accepts reviewed UML 1.1 as standard
- May 2001: Version UML 1.4 agreed
- March 2003(!): last version UML 1.5 (working version for 2.0)
- July 2005: formal adoption of UML 2.0
- June 2015: last released version 2.5



The idea: Unified Modeling Language

There are a lot of different definitions of the word „modeling", depending from the context you use it. Modeling in mathematics is different from modeling in solids or modeling in the fashion. When it comes to modeling in computer science there is always the principle of abstraction, it allows to capture the important aspects of a system while omitting the rest.

The idea: Unified Modeling Language

UML is a language and consequently serves for communication!

The aim is to achieve a common understanding of the system functionalities for all persons involved (from the user via the manager up to the developer) by using the UML at any time!

The words of this language are called model elements, the paper where the words can be written down and can be read from are the diagrams. Each diagram is just a view of part of the system together, all diagrams provide a complete picture: the model. But bear in mind: also model elements that do not appear on a diagram belong to the model. As for every other language there is just a well defined grammar and semantics in UML, but no guidelines how to use it, here is just the most important suggestion: keep it simple, or to quote Grady Booch:

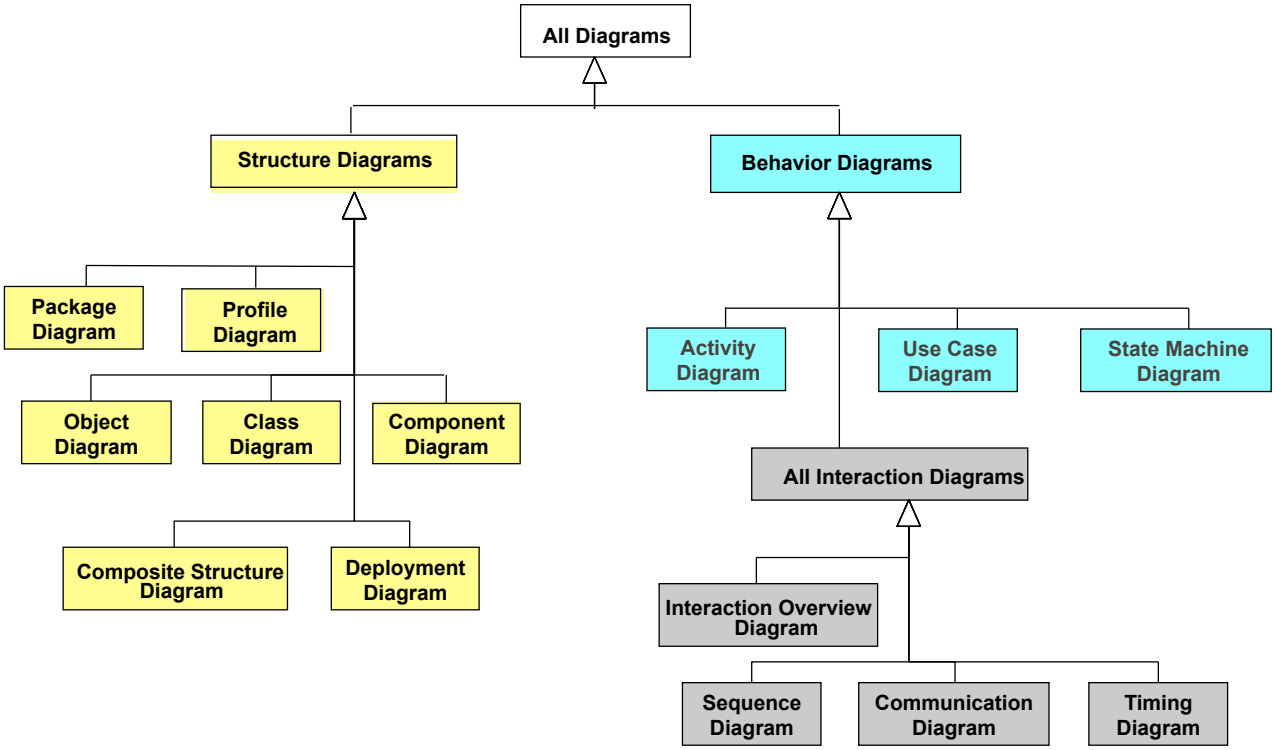
Best Practice Tip

"80% of most problems can be modeled with 20% of UML."

UML - Overview Diagrams

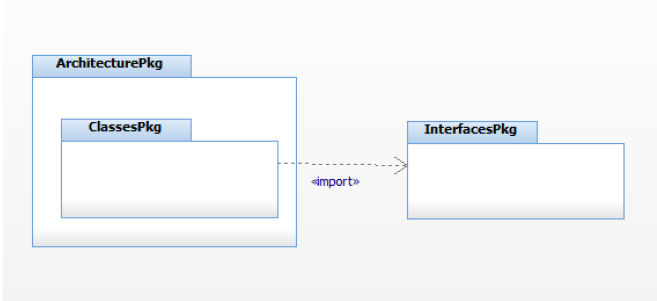
What does the language consist of? There are model elements which are the „words“ of the language and there are diagrams. The diagrams provide the user with the „paper“ to write the words down. Usually a model has a lot of diagrams, each is just a special view to a special aspect of the model. All diagrams together provide a picture of the problem.

The 14 Diagrams



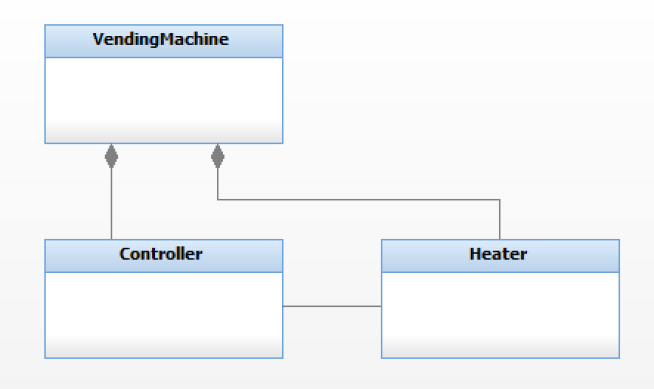
The Structure Diagrams

Package / Profile Diagram



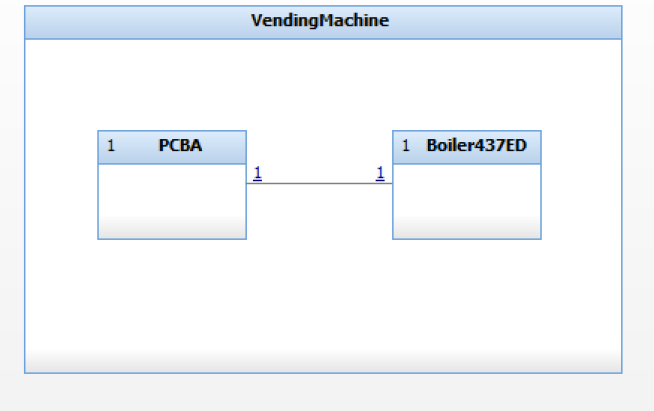
The Package as well as the Profile Diagram show the structure of a model. They use packages (respectively profiles) and dependencies among them. A dependency is stereotyped with „import“ or „access“.

Class / Component / Object Diagram



These 3 diagrams show classes, respectively components or objects and dependencies between them. They provide a static view into the system and emphasize the resources of the parts of the model.
The example here is a class diagram showing (only a small part) of the model of vending machine.

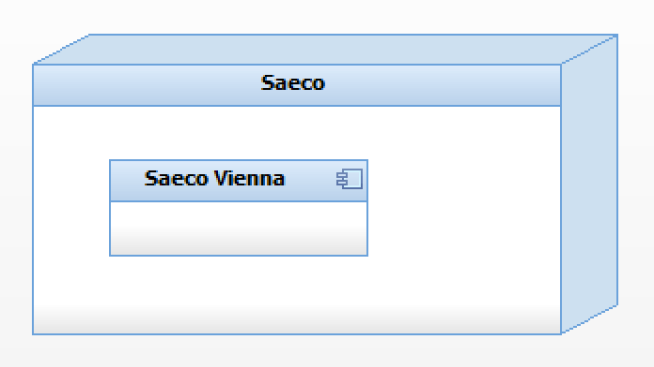
Composite Structure Diagram



A structured class or component contains a collaboration of parts. The Composite Structure Diagram defines in which way the parts are connected and with whom they communicate. Unlike the 3 diagrams that show the resources of a model (Class / Component / Object Diagram) this diagram shows the current usage inside the architecture of the system.

Again: only a small part of the system is shown in this screenshot.

Deployment Diagram

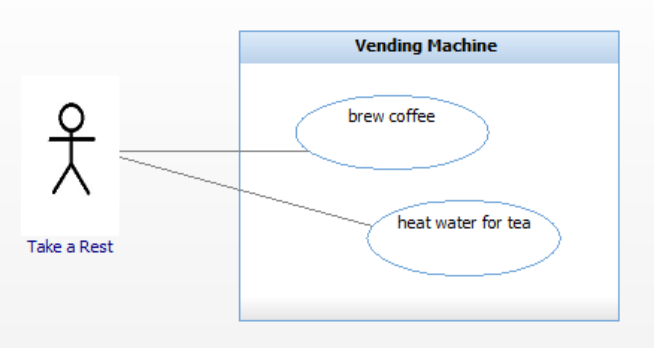


A Deployment Diagram specifies a set of artifacts deployed onto a set of interconnected nodes.

Here we have only one node and one component, due to the fact that the model of the coffee machine is not very complicated.

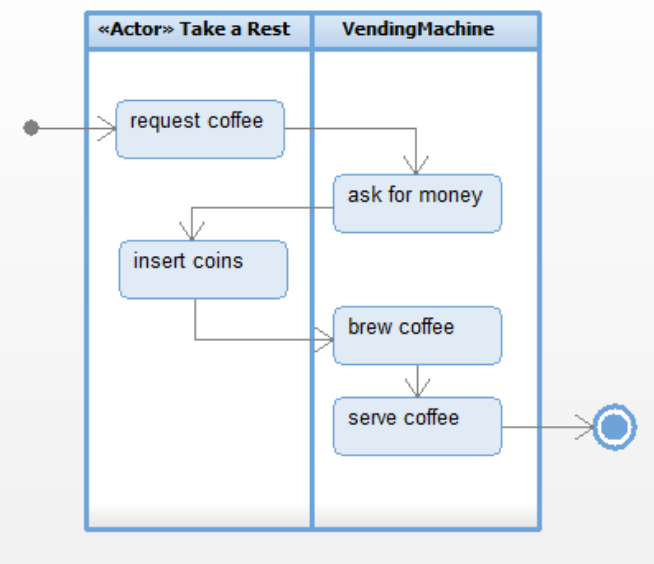
The Behavior Diagrams

Use Case Diagram



The Use Case Diagram describes the functionality a stakeholder wants the system to do. Important model elements are actors, use cases, associations and include and extend dependencies.

Activity Diagram

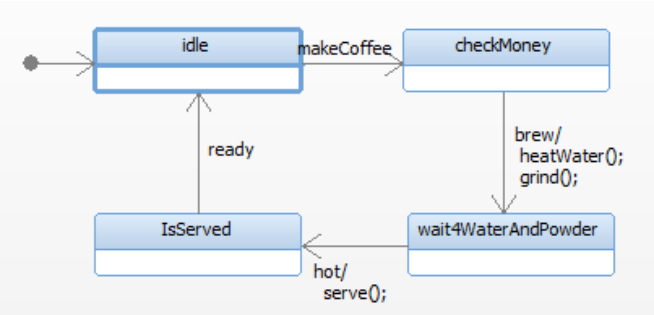


An Activity Diagram describes the flow of actions. Important model elements are action and activity nodes, object nodes, pins, join and fork control nodes and swim lanes.

Compared to Sequence Diagrams they emphasize actions and activities, but not the messages that are used for communication.

Again the example is a simple one, reminder: the coffee machine isn't very complicated.

State Machine Diagram



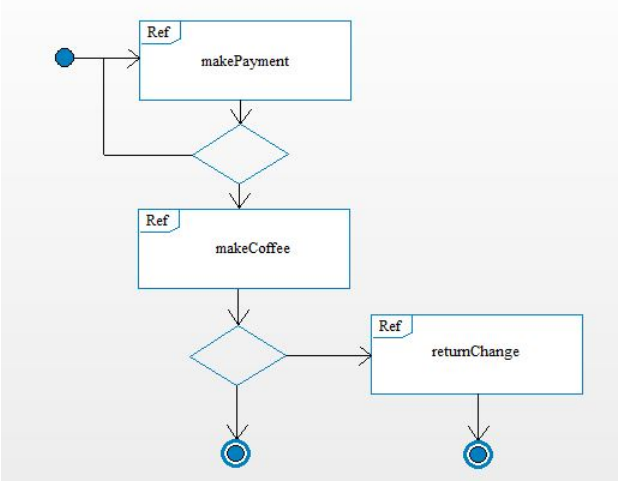
A State Machine Diagram describes the behavior of the system, or the components of the system in a form of Extended Temporal Finite State Machine.

Reminder:
the example is a simple one, the coffee machine isn't very complicated.

Advantage of state machines: they are formally defined, consequently they can easily be proved.

The Interaction Diagrams

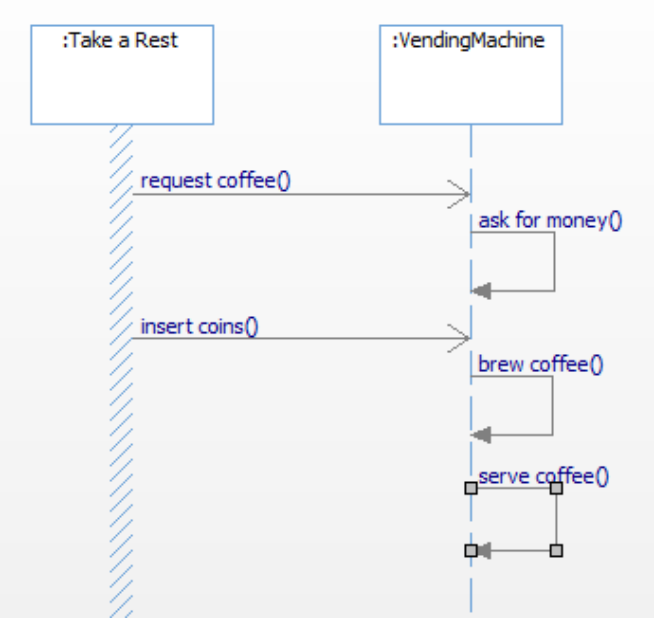
Interaction Overview Diagram



An Interaction Overview Diagram describes the flow of interaction diagrams. Important model elements are decision nodes and combined fragments.

Compared to Sequence or Activity Diagrams they emphasize other interaction diagrams of the model, but not messages or actions of the system.

Sequence Diagram



A Sequence Diagram describes the order of messages in time. Important model elements are lifelines, messages and combined fragments.

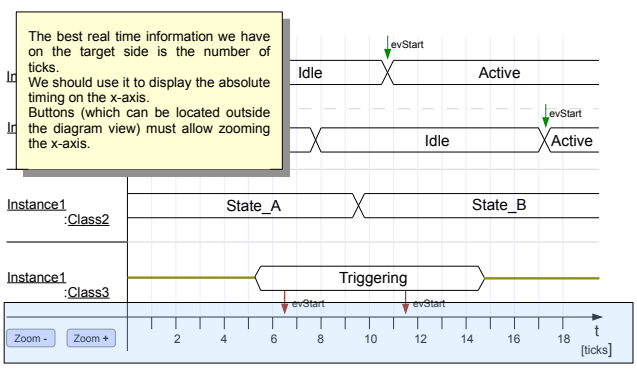
Compared to Activity Diagrams they emphasize communication between entities, instead of actions or activities.

This example looks similar to the appropriate Activity Diagram. This is done intentionally. In a normal model the content of Sequence and Activity Diagrams is different.

Best Practice Tip

A recommendation to the question: „Activity and Sequence Diagrams have a similar purpose, both show an order of something, so: when shall I use an Activity and when I shall use a Sequence Diagram?“
Answer:
if you want to put the focus on actions, use the Activity Diagram;
if the important information is the communication between some entities, use a Sequence Diagram.

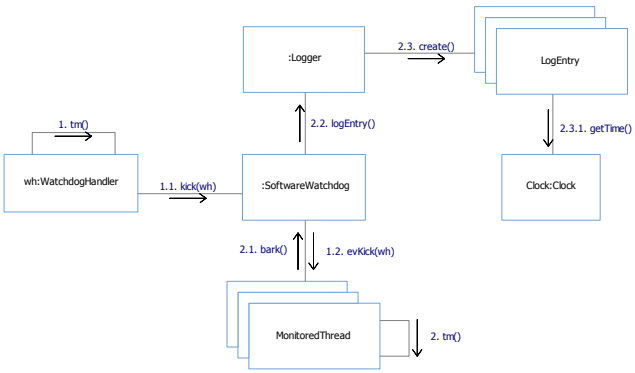
Timing Diagram



This diagram shows instances and states of classes in run-time. The timing axis is a real-time axis and the time in this example is divided in system ticks. Also shown are the events that trigger the transition of states.

Ideally this diagrams is used in software for the Real Time World, a model for the coffee machine does not need such a diagram. Consequently, this screenshot does not show parts of the coffee machine.

Communication Diagram



Communication Diagrams have the same purpose as Sequence Diagrams, they show the sequence of messages, that are used for the communication between entities. Entities can be classes, components, object, actors or the environment of a system. Communication Diagrams are rarely used, probably as it is more difficult to read them because the sequence of the messages are shown by numbers.

Best Practice Tip:

Do not use Communication Diagrams, use Sequence Diagrams instead. Communication Diagrams are only for historical reasons still in UML.

Note:

The important diagrams - they should always be present in any model - are:

- Package Diagrams
- Use Case Diagrams
- Sequence and/or Activity Diagrams
- Class Diagrams
- Composite Structure Diagrams
- State Machine Diagrams

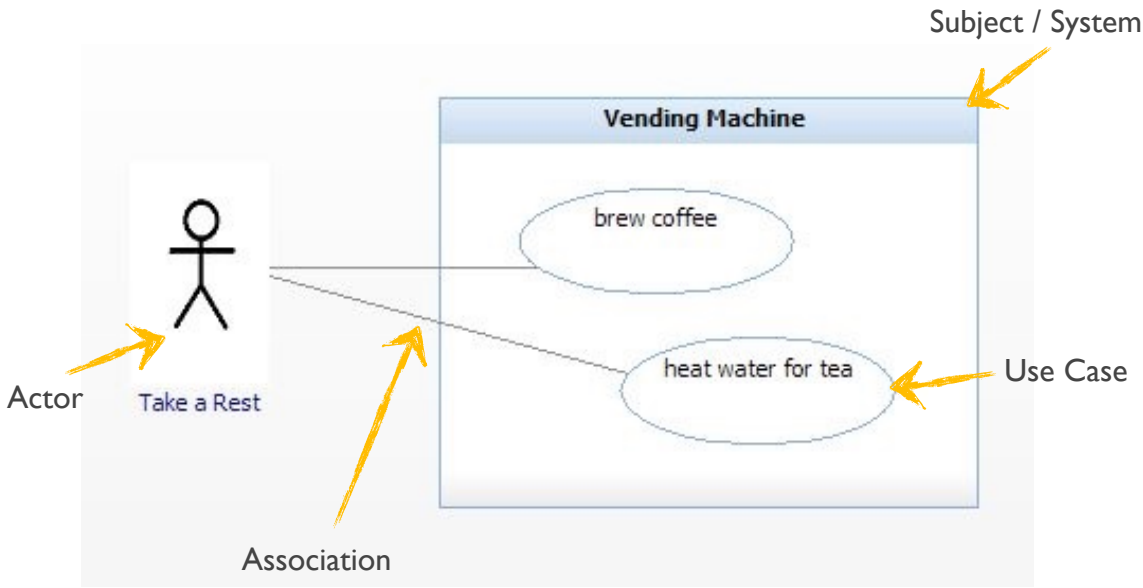
Use Case Modeling

The starting point in modeling any system is deciding exactly:

- Describe **WHAT** the system will do at a high-level
- User focus

  Capture requirements from user's perspective

  Users are involved



The Diagram

The Elements

Actor

An external entity that interacts with the system.

- Users
- External systems
- System buses (e.g. CAN, Ethernet)
- Sensors, motors, devices, …

An actor represents a role. One physical person can play several roles in interacting with the system and these

Best Practice Tip

There is **never** an actor **not** associated with any Use Case

Identifying Actors

There are several ideas to find the relevant actors for a system. Maybe ask the following questions:

- Who will use the main functionality of the system?
- Who will need support from the system to do their daily tasks?
- Who will need to maintain, administrate, and keep the system working?
- Which hardware devices does the system need to handle?
- With which other systems does the system need to interact?
- Who or what has an interest in the results that the system produces?

Best Practice Tip
Use a **noun** for an Actor's name.

Use Case

- A Use Case is a functionality that the system shall offer to an actor
- The Use Case describes the interaction between one or more actors and the system
- The Use Case symbol contains text which is the goal of a related actor

Whilst the name of a Use Case specifies the main functionality only, one also needs to describe the details of the Use Case. Usually they are depict in scenarios, shown inside the model in textual way or in a graphical way with:

- Sequence Diagrams
- Activity Diagrams
- State Machine Diagrams

Information recorded by the detail of a Use Case will include the basic course of action ('sunny day' sequence of events) as well as some alternative courses (include also failure situations)

Best Practice Tip
Use a **verb** for a Use Case's name

Identifying Use Cases

Put yourself in the situation of an actor. Ask yourself: "What do I want to do with the system?"

The Use Case shall represent a complete functionality for the actor. Don't divide it into parts that must be used together to add some value to the actor.

Best Practice Tip
A Use Case should satisfy a **major** goal for the actor

Subject / System

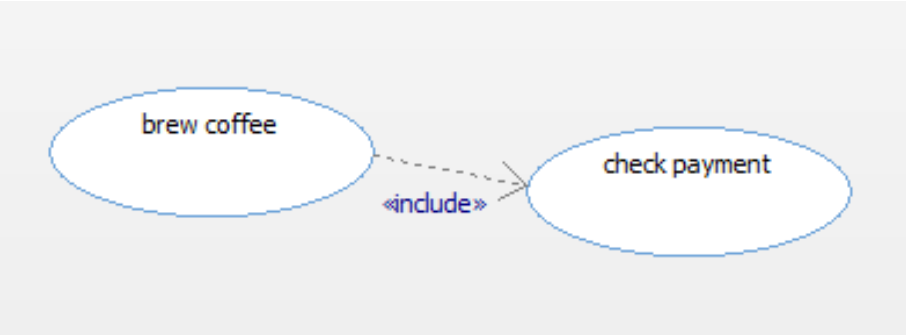
Indicates the system boundary. This is now an important phase of developing a model:
Here is the definition what is part of the system and what is NOT part of the system. Represents the system that shall be developed; all entities that interact with the system are outside it.

Association

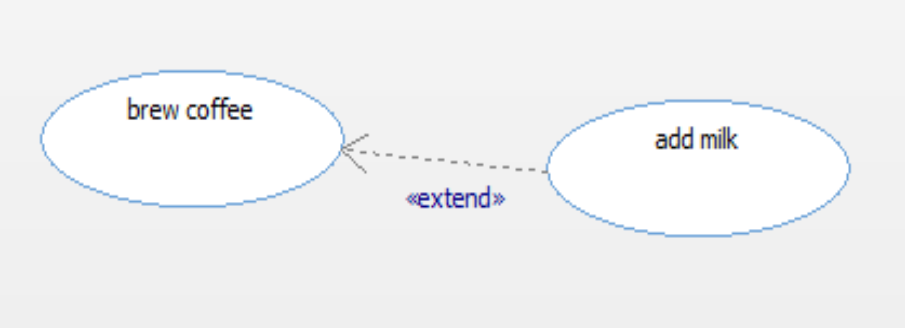
Drawn between an Actor and a Use Case. It represents the communication between the actor and the system. The communication can be initiated from the Actor but also from the Use Case. Consequently the association is NOT directed. The Associated can be named, but normally this does not provide any benefit. Supposed, all the Actors and Use Cases are defined, the Associations between them show the most important advantage: in case an Association is not present, one can be sure, that

- an Actor is superfluous or
- a Use Case is superfluous or
- some requirements are missing.

Dependencies between Use Cases



An include relationship between Use Cases is shown by a dashed arrow with an open arrowhead from the base Use Case to the included Use Case. The arrow is labeled with the keyword «include».



An extend relationship between Use Cases is shown by a dashed arrow with an open arrowhead from the Use Case providing the extension to the base Use Case. The arrow is labeled with the keyword «extend».

Generalization between Use Cases or Actors

This relationship is in general possible, as both model elements are "classifiers" in the meta model of UML. Classifiers may have a "Generalization"-relation between them. Nevertheless from the perspective of user's style using UML is does not make sense to use it. If 2 Use Cases are similar and differ in some points, it can make sense to use the Include- and/or Extend-Dependency.

Best Practice

Some Recommendations

Use Case Diagrams and their model elements give an answer to the question "**What**" does the system do.

Avoid to describe solutions.

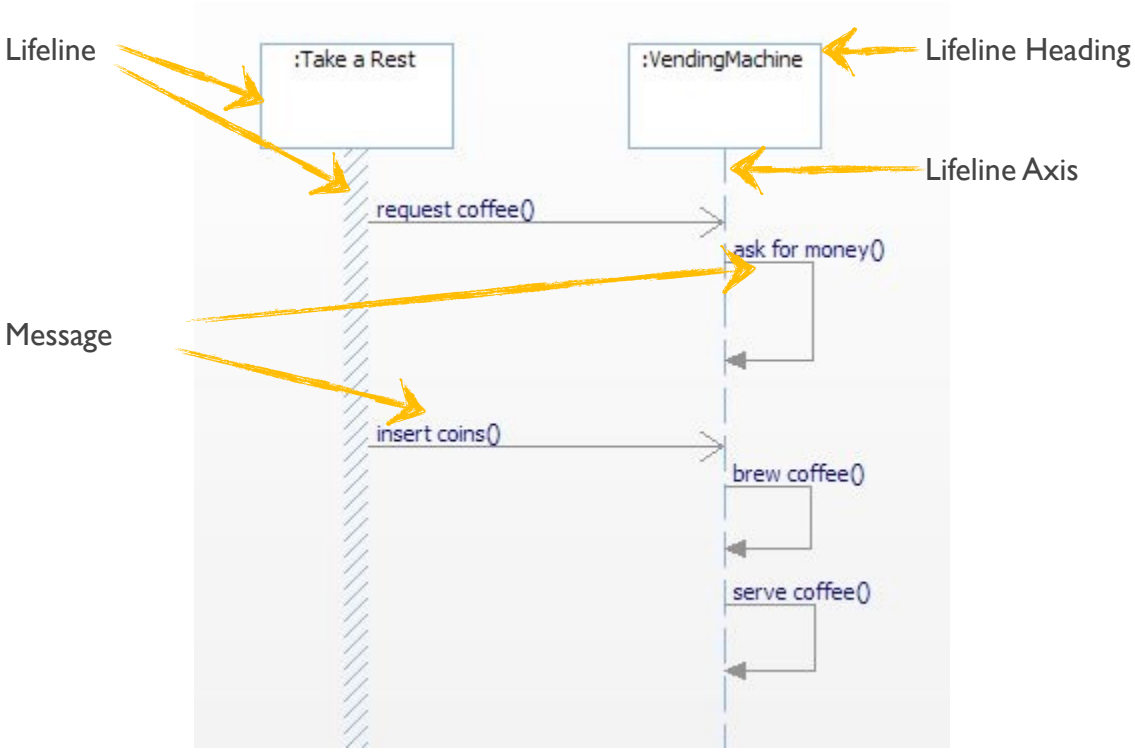
Use Case Diagrams and their model elements give an answer to the question "**What**" is **Inside** and "**What**" is **Outside** of the system.

A Use Case should satisfy a **major** goal for an Actor, i.e. do not specify too many Use Cases in a model.

A Use Case is **not** a Requirement

The model is still viewed as a **Black Box.**

Sequence Diagram



Lifeline

A Lifeline represents an individual participant in an interaction, it can be an Object, a Class, a Component, an Actor, etc.. While Classes or Objects or other structural features can have a multiplicity greater than 1, a Lifeline represents exactly 1 entity.

Note that Rhapsody shows different axis for Lifelines, a Lifeline representing an Actor is shown in small diagonally broken lines (e.g. "Take a Rest") a Lifeline representing other model elements has a dashed vertical axis. The UML standard requires that the axis is a vertical line (which may be dashed).

The syntax requires that the Lifeline Heading contains a name.

The semantics of a Lifeline: the order of the specifications on a Lifeline denote the order of their occurrence. It means the order of messages or other specifications is the only import information. The distance between the specifications is irrelevant, neither is there any time information corresponding to the axis of the Lifeline.

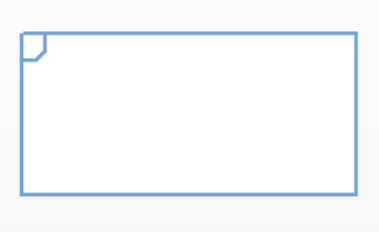
Message

A Message represents communication between Lifelines. A Message must have a name, it can have arguments. A Message has exactly one sender and one receiver, i.e. there is no broadcast. The sender and the receiver can be the same Lifeline. It is also allowed to avoid showing the sender of a Message (found Message) or the receiver (lost Message).

A Message can represent asynchronous communication or synchronous communication. The UML wording for an asynchronous Message is Signal (Rhapsody uses Event) and Operation for a synchronous Message. The Message line can be drawn horizontally or diagonally downwards for both kinds. The kinds differ in the arrow head: an asyncSignal has an open arrow head ("request_coffee"), a syncOperation has a filled arrow head.

More kinds of Messages are replyMessage (the return result of a syncOperation), a createObjectMessage or a deleteObjectMessage. (They are of minor importance and should only be used in diagrams in the model implementation phase)

Interaction Fragment

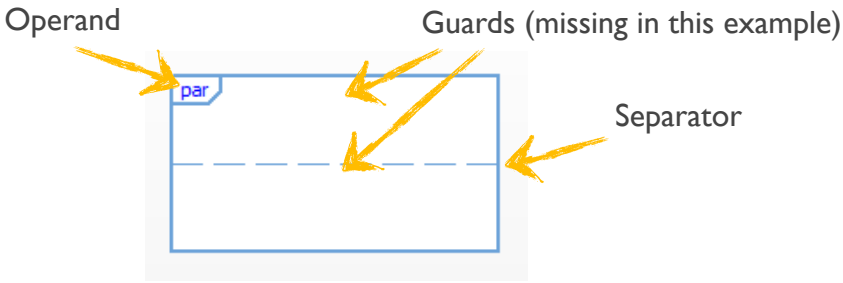


InteractionFragment is an abstract notion of the most general interaction unit. An interaction fragment is a piece of an interaction. Each interaction fragment is conceptually like an interaction by itself.

Interaction Operand

Usually an Interaction Fragment comes in combination with an Interaction Operand or with an Interaction Use. An Interaction Operand is given as text in a small compartment in the upper left corner of the CombinedFragment frame and is one entry of the list:

- alt
- opt
- par
- loop
- critical
- neg
- assert
- strict
- seq
- ignore
- consider



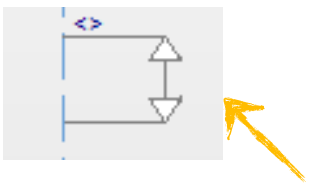
Optionally the operand can be used together with a guard (if the guard condition is empty, it is taken to mean a true guard). Here we have parallel interaction, both conditions are true, the fragment is divided by a Separator.

Interaction Use

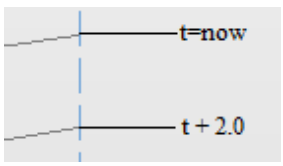
In this case the fragment has the keyword "Ref" in the compartment in the upper left corner. An Interaction Use refers to another interaction, in most cases this is another Sequence Diagram.

Duration / Time Constraint

The only way in UML to specify time or duration information. In Rhapsody only the Duration Constraint is available and has the name "Time interval".



Duration Constraint / Time Interval



Time Constraint / Time Observation
(not available in Rhapsody)

Best Practice

There are a lot more model elements that can be used in Sequence Diagrams. But we strongly believe in the maxim: use at most 20% of the UML.
Consequently we are sure, that you can solve all Sequence Diagram related questions with the model elements described in this document. If not we recommend you have a look into the standard instead of buying some books about UML.

Some Recommendations

Sequence Diagrams and their model elements give an answer to the question "**Who**" communicates with "**Whom**".

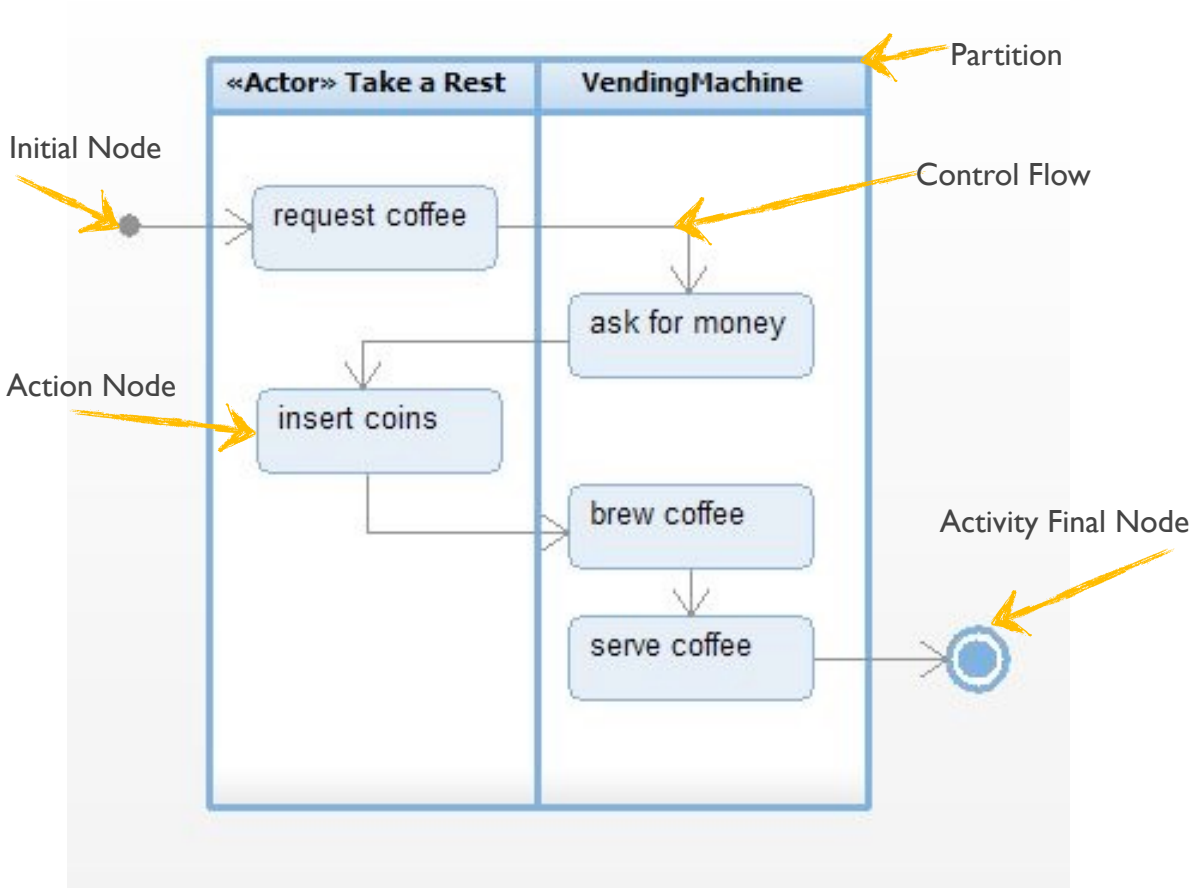
Sequence Diagrams and their model elements give an answer to the question "**Which Messages**" are used for the communication.

Sequence Diagrams and their model elements give an answer to the question "**What is the Order of the Messages**".

Avoid to use other elements like actions or states; they are the key elements in different diagrams, not in a Sequence Diagram.

In the requirements analysis phase use one Sequence Diagram for one scenario, do not use fragments with optional or alternative communication. And concentrate on the important scenarios, so that you do not get lost too much in details.

Activity Diagram



Partition

Partitions divide an activity. They often correspond to organizational units in a model. In this example all actions from the actor are grouped into the corresponding partition, and similarly the actions from the Vending Machine. Partitions are often also called "Swim Lanes".

Activity Nodes

are Action Nodes, Object Nodes and Control Nodes.

Action Node

is an element that is the fundamental unit of executable functionality. An action must have a name.

Control Flow

A control flow starts an activity node after the previous one is finished. This is done automatically without the need to get any trigger from the model or from the environment.

Control Nodes

are Initial, Activity Final, Flow Final, Decision, Merge, Fork and Join Node.

Initial Node

The starting point for an activity. An activity may have more than one starting point, in that case multiple flows are started when the activity is initiated.

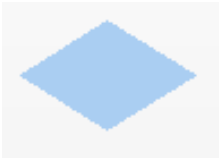
Activity Final Node

An activity final node is a final node that stops all flows in an activity.

Flow Final Node

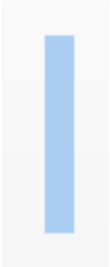
A flow final node stops the current flow only.

Decision / Merge Node



A Decision has one incoming and several outgoing flows, the Merge has several incoming and one outgoing flow. The outgoing flows of a Decision need to be guarded so that a choice between the outgoing flows can be made.

Fork / Join



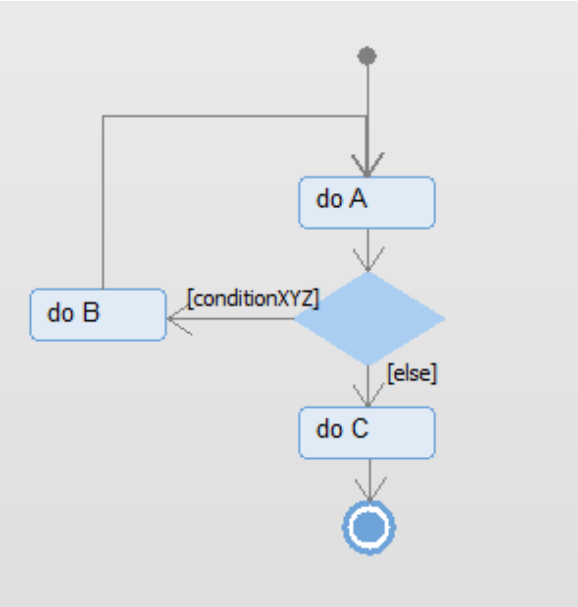
The Fork is similar to a Decision; but here the flow is split into several parallel flows. The Join is similar to a Merge, but here are multiple flows joined into one.

Semantic

While Sequence Diagrams emphasize an order of communication between entities, Activity Diagrams emphasize a sequence of actions. They do not put value on messages. Activity Diagrams are in principle executable, that requires the UML to provide a semantic.

It can in easiest way explained with "tokens". A token is put onto the initial node and follows the control flow. Any action is executed automatically after all possible tokens (maybe an action has more than one possible incoming control flow) arrived. After the action run to completion tokens are delivered on all possible outgoing flows.

Examples

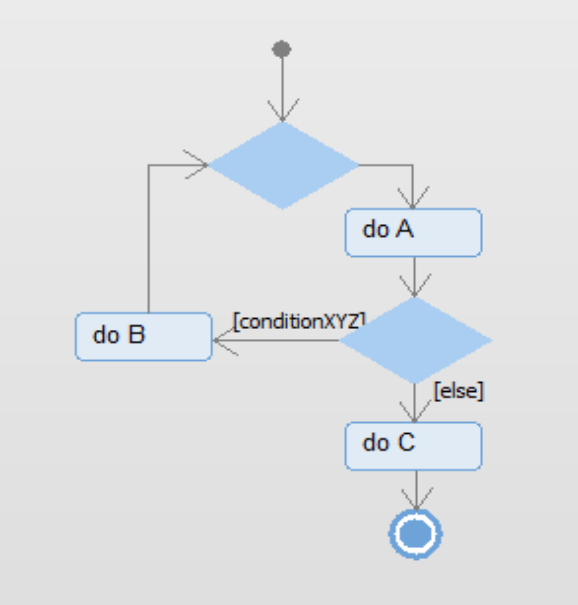


What will happen in this example after the Activity is initiated?

Answer: nothing at all.

Reason:

The condition for the Action Node "do A" to be executed is that all possible tokens must be present. But only the token from the Initial Node can arrive, the token sent from Action Node "do B" will NEVER arrive.



What will happen in this example after the Activity is initiated?

Answer: it works as expected.

Reason:

The token semantic for a Merge Node states, that only one token must be present to deliver the outgoing token onto the Control Flow to the Action Node "do A".

Best Practice

There are a lot more model elements that can be used in Activity Diagrams. But we strongly believe in the maxim: use at most 20% of the UML.
Consequently we are sure, that you can solve all Activity Diagram related questions with the model elements described in this document. If not, we recommend you to have a look into the UML-Standard instead of buying some books.

Some Recommendations

Activity Diagrams and their model elements give an answer to the question "**What is the Order of Activities / Actions**".

Avoid to use messages;
they are the key elements in Sequence Diagrams.

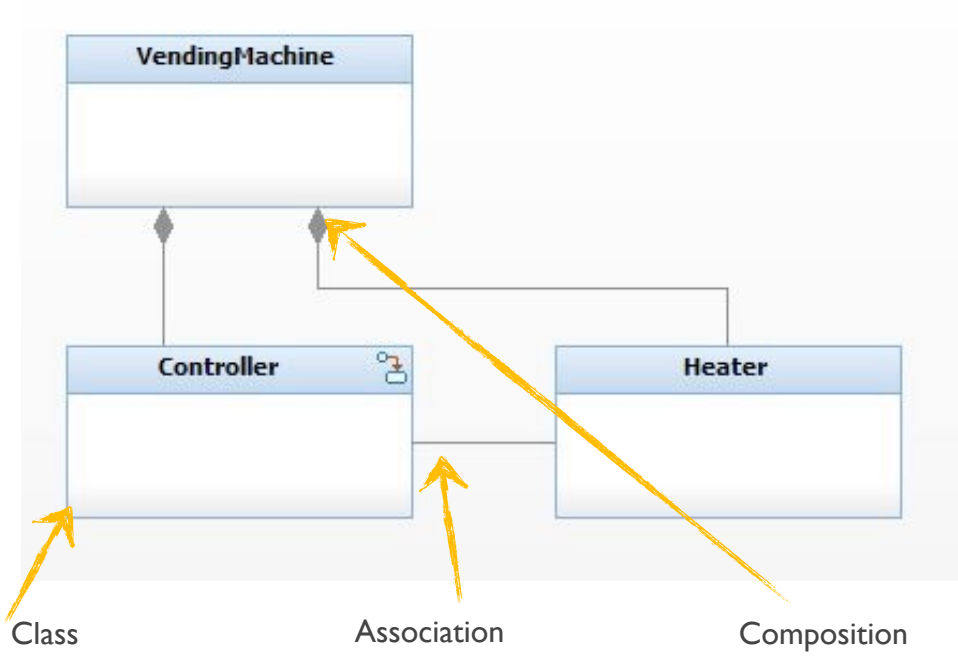
Activity Diagrams support the Requirements Analysis phase of a model. As they are executable, one can simulate the behavior of a Use Case with their help.

Activity Diagrams support the System Design phase of a model, they ease to find operations. Partitions inside the System Design Activities serve for the "allocation" of operations to classes or components.

During the Implementation phase of a model they allow the code generation for classes.

Inside the Testing world they can build the execution for test cases.

Class Diagram



Model Elements

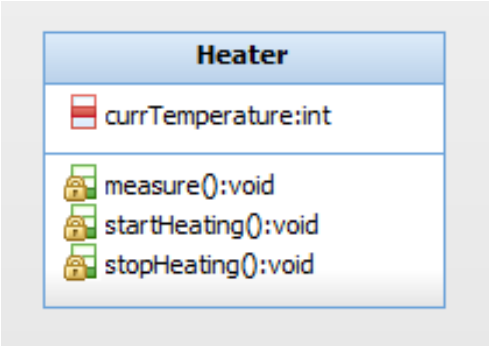
Classes, Interfaces, Relationships between Classes

Class

A class describes a set of objects that share the same specifications of features, constraints, and semantics.

Class is a kind of classifier whose features are attributes and operations.

The graphical symbol for a class is a rectangle. The rectangle can be divided in compartments. A class is often shown with three compartments. The middle compartment holds a list of attributes while the bottom compartment holds a list of operations. Attributes or operations may be presented grouped by visibility. A visibility keyword or symbol can then be given once for multiple features with the same visibility. Additional compartments may be supplied to show other details, such as constraints, or to divide features.



For demonstration reasons we filled the "Heater" class with an attribute and 3 operations. Rhapsody uses special icons to differentiate attributes and operations as well as the private and public visibility. The attribute "currTemperature" is publicly visible, means everything outside and inside the class can read and write this attribute. The 3 operations have a private visibility, only the heater class itself has access to the operation. UML uses the characters '+' (visibility: public) and '-' (visibility: private).

Features of Classes

Attributes

An Attribute is an element to store data.

Notation:

visibility name : data-type [multiplicity] = value {constraint}

Name is mandatory, all other parts of the notation are optional.

Operations

An Operation is a behavioral feature of a class that specifies the name, type, parameters, and constraints for invoking an associated behavior.

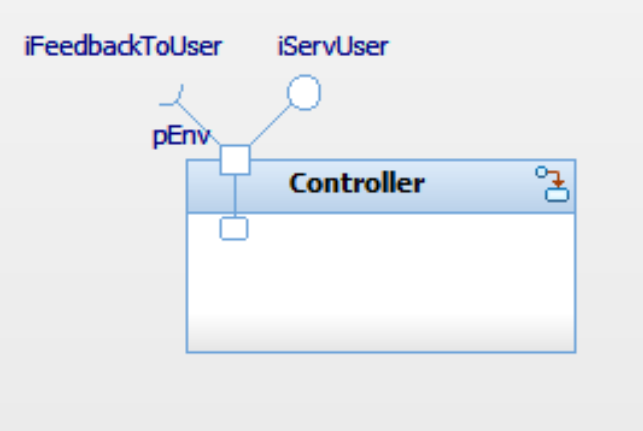
Notation:

visibility name (parameter-list) : return-type [multiplicity]

Name is mandatory, all other parts of the notation are optional.

Ports

Ports represent interaction points between a class and its environment. The interfaces associated with a port specify the nature of the interactions that may occur over a port. The required interfaces of a port characterize the requests that may be made from the class to its environment through this port. The provided interfaces of a port characterize requests to the class that its environment may make through this port.
A port has the ability to specify that any requests arriving at this port are handled by the behavior of the class; rather than being forwarded to any contained parts. Notation:



The Port is shown as a small box on the border of the class symbol. In this example the port "pEnv" is a behavior port, that states that any requested behavior is to be defined inside the class "Controller".

The behavior is specified in the Interface "iServUser". It has a Realized-Dependency (or ball notation) to the class "Controller". The Usage-Dependency (half-circle or socket notation) states that this Interface is needed to be served or realized by something else.

Using this kind of interaction point for a class result in decoupling the internals of the class from its environment, ports allow a classes to be defined independently of its environment, making that class reusable in any environment that conforms to the interaction constraints imposed by its ports.

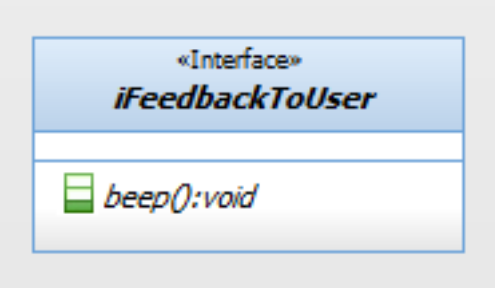
Interface

An interface declares a set of public features or service offered by a class or component. An interface does not specify how it is to be implemented, but merely what needs to be supported by realizing instances.

Because an interface is merely a declaration, it is not an model element that can be instantiated; that is, there are no instances of interfaces at run time.

The set of interfaces realized by a class or component are its provided interfaces. They represent the duties that this class / component have to their clients.

Interfaces may also be used to specify required interfaces, which are specified by a usage dependency between the classifier and the corresponding interfaces. Required interfaces specify services that a class / component needs in order to perform its function and fulfill its own duties to its clients.

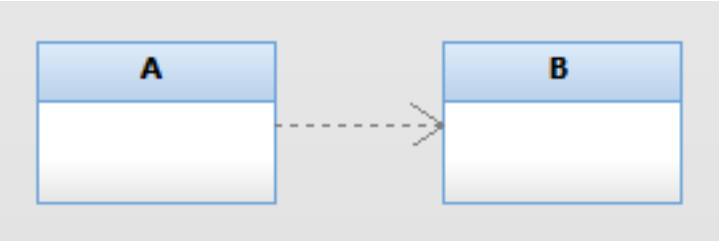


Relationships between Classes

Dependency, Association, Aggregation, Composition, Generalization, (Realization, Extension)

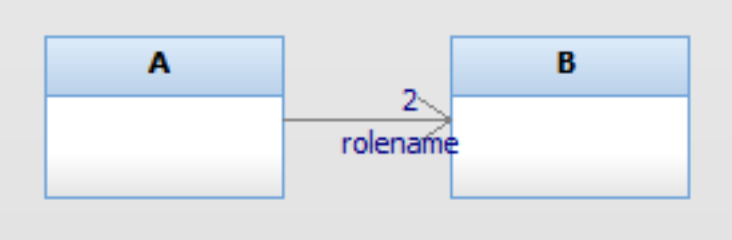
Dependency

A dependency signifies a supplier/client relationship between model elements where the modification of the supplier may impact the client model elements. A dependency implies the semantics of the client is not complete without the supplier. Notation is a dotted line with an open arrow head. The arrow points to the dependent element.



Association

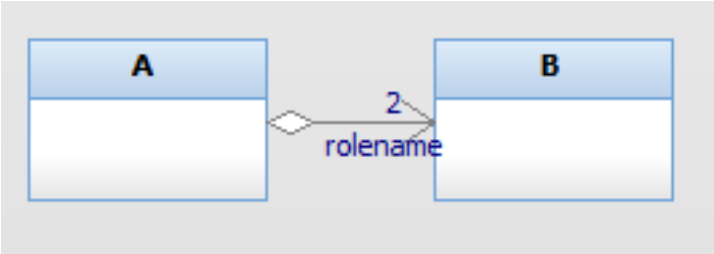
An association specifies a semantic relationship that can occur between classifier (Classes, Actors, Use Cases,...). The notation is a simple line. An example:



Semantic for this example: 1 instance of A is somehow related to 2 instances of B, one can navigate from A to B using the name "rolename", but there is no definition on navigability from B to A.

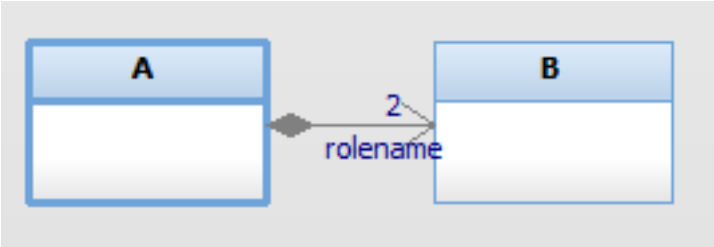
Aggregation

This is an Association with additional whole / part property. Now B is part of A.



Composition

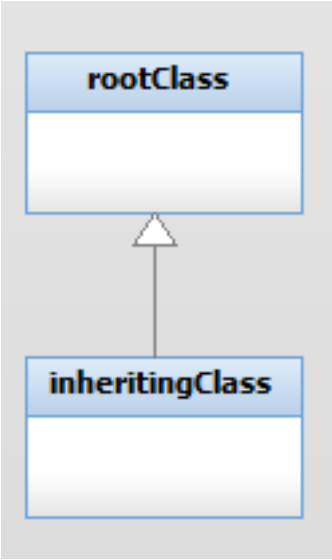
This is an Aggregation with additional responsibility to the whole. If an instance of A is deleted, the composition parts of B are deleted as well.



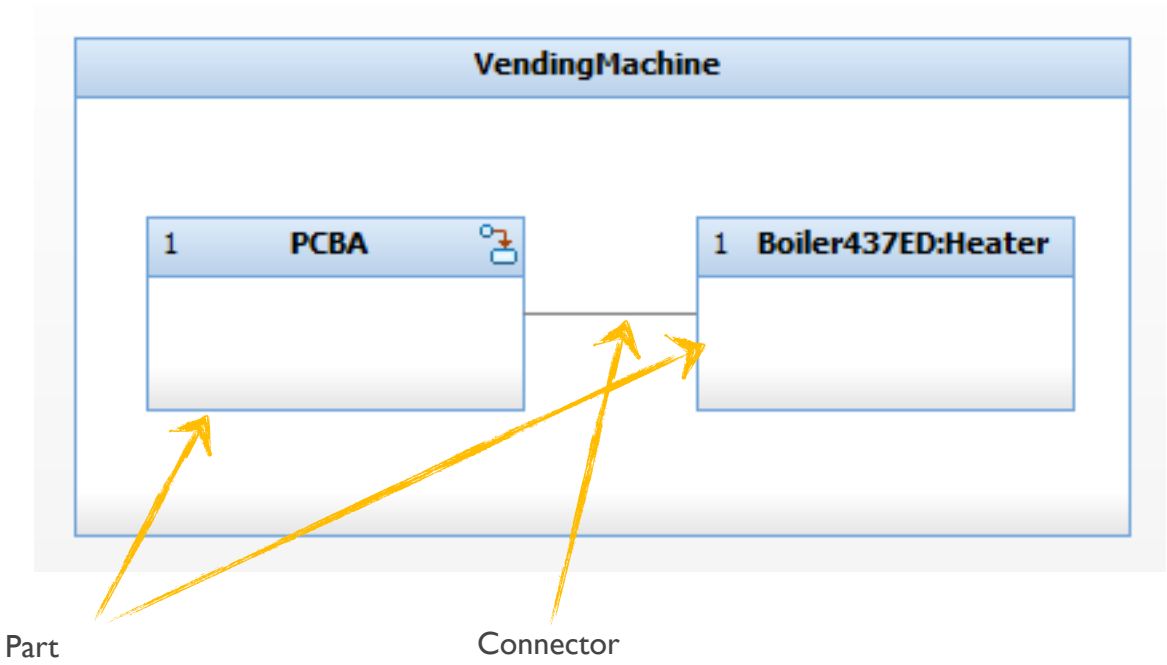
Generalization

A generalization relates a specific classifier to a more general classifier, and is owned by the specific classifier. Multiple Generalization is possible, i.e. a classifier can inherit from more than one root class at the same time. Root classes that do not allow to be instantiated are abstract classes. Notation for abstract classes: the class name is shown in *italics*.

Properties of the root class can be used in the inheriting class as they are, but they can also be overridden inside the inheriting class.



Composite Structure Diagram



The term "structure" in this clause refers to a composition of interconnected elements, representing run-time instances collaborating over communications links.

Model Elements

Parts, Connectors

Part

A part is an instance of a class or component. The part is shown as a rectangle with at least the name of the part. Additional information can be show by a number indicating the multiplicity of the part and the type name. In the example above we have the parts PCBA and Boiler437ED of class Heater.

Connector

Specifies a link that enables communication between two or more instances. This link may be an instance of an association, or it may represent the possibility of the instance to communicate via ports.

State Machine Diagram

A state machine is composed of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition; this is called a transition. To define a state machine one needs a list of its states, and the triggering condition for each transition. A state machine can be defined in a class, a component, an actor or in a use case.

Benefits of State Machines

- State machines are formally defined, they form executable models.
- State machines can be executed and visualized at the design level of abstraction.
- State machines can be animated -- their dynamic behavior shown graphically .
- You can focus on the abstract behavior.
- State Machines provide for easy testing.

State

Identifies a condition, the system is in. The combination state name - trigger is unique inside the owner of the state machine. A trigger is necessary to get the state machine from one state into another. That is called "transition".

Transition

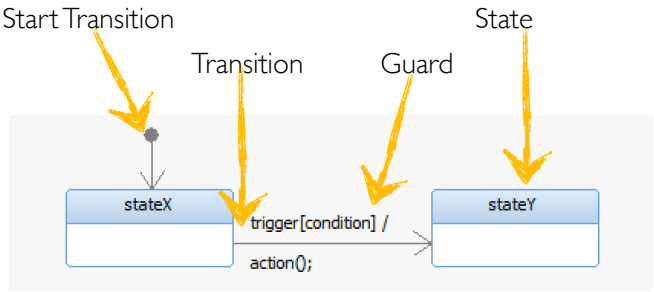
The step from one state into another. A transition is initiated from a trigger, it is the active phase of the state machine. In this phase one or more actions can be executed. They run to completion with the result, that at the end of the actions the state machine is again in a condition (i.e. waiting for another trigger). The symbol is a solid line with an arrow. Trigger, guard and list of actions may be denoted textually on the transition line. Graphical symbols are also possible.

Guard

The trigger can have a guard. It contains a condition that evaluates to true, otherwise no transition will occur.

Pseudostate Initial State (Start Transition)

This state including the transition to the next state is present exactly once in each state machine. The start transition executes automatically and without being triggered.
During the transition any initializing action may be invoked.



Different Types of Trigger

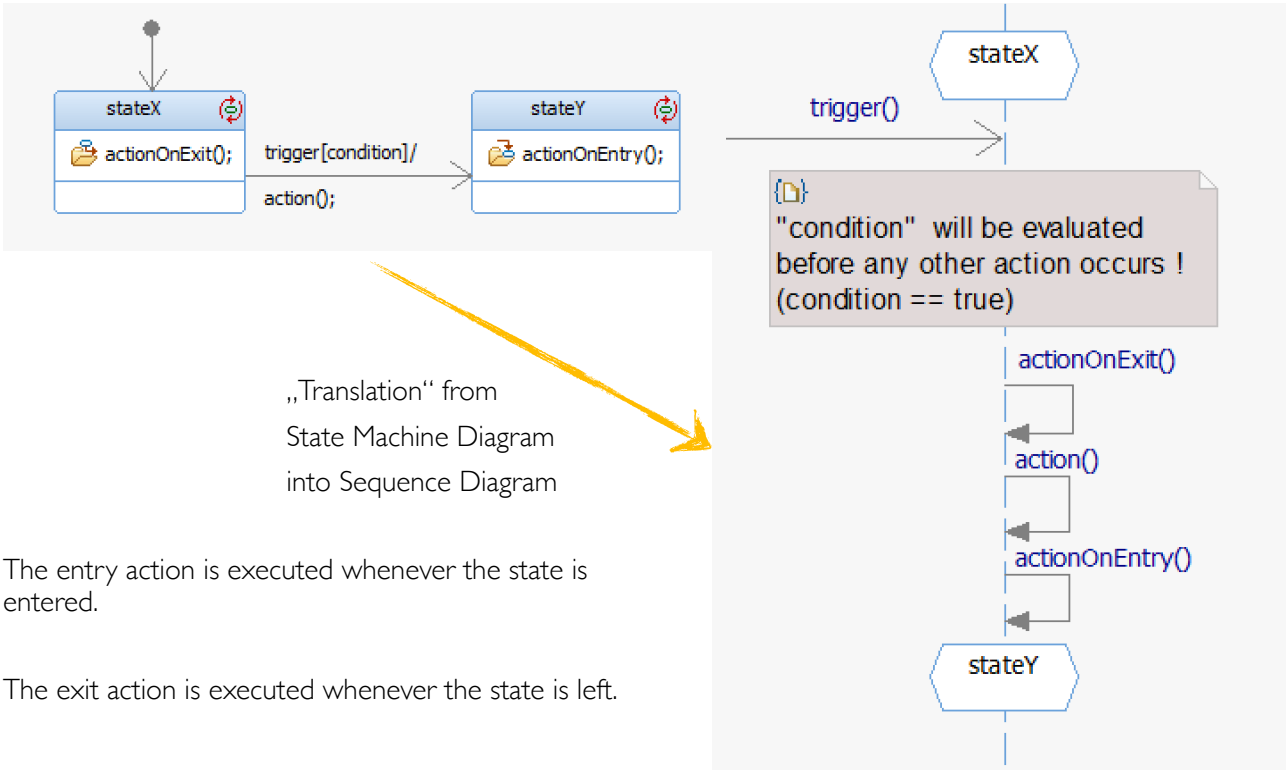
A trigger can be:

- An asynchronous message, sometimes called event, sometimes also called signal
- A synchronous message, usually an operation of a classifier (block / use case / …)
- A timeout
- A change event, an attribute of the owning block has changed.

Note: The change event trigger is not implemented in Rhapsody.

Entry and Exit Actions

More elements: entry and exit actions, they are executed „inside“ a state:



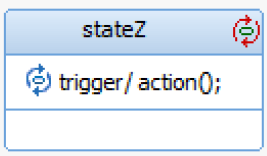
The entry action is executed whenever the state is entered.

The exit action is executed whenever the state is left.

Reaction in State

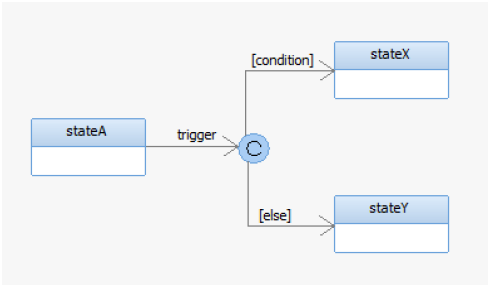
More elements: reaction in state, it is executed „inside“ a state:

If „stateZ“ is the current state and „trigger“ arrives, „action()“ will be executed, but no transition occurs. Consequently exit or entry actions are also not executed.



Choice

A choice or sometimes also called decision or condition connector is a pseudo state.
A „trigger" causes the „move" to the choice. There all specified guards are evaluated. The one that evaluates to true is executed.



Note: Responsibility for the correctness (only one guard is allowed to be true) is up to the user of the state machine.

Note: The guards are evaluated before the trigger is consumed.

State Machine Semantic

If the state machine is in a „stateX" and a „trigger" is sent and the guard (if any) on that transition evaluates to „true", that transition is handled. It will execute any actions associated with that transition.

Events are quietly discarded if:

- A transition is triggered, but the transition's guard evaluates to „false"
- A transition to a conditional pseudostate is triggered, but all exiting transition guards evaluate to „false"
- There is no transition line with the event as trigger.

Actions run to completion. Normally actions take an insignificant amount of time to perform. They may be interrupted by another thread execution, but that object will complete its action list before doing anything else. Actions are implemented via

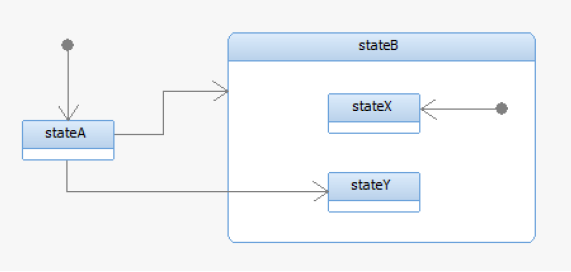
- A classifier's operations
- Externally available functions.

Hierarchical State Machines

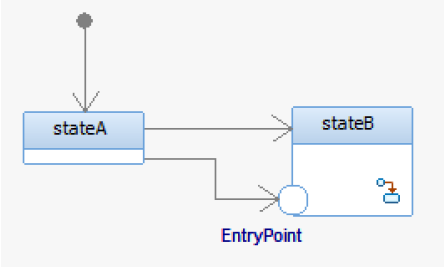
It is a state machine inside a state. All statements from above are still valid, each state machine follows the mentioned rules. Additionally one can specify entry and exit points, and transitions to the super-state as well as transitions to nested states inside the super-state.

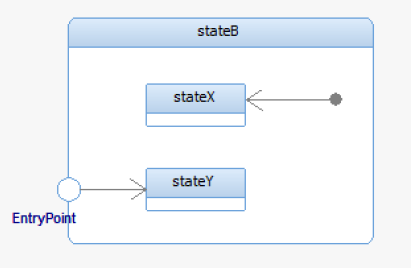
Examples:

State „stateB" contains a state machine with 2 states and a start transition. There is a transition from „stateA" to „stateB" which results in execution of the start transition to „stateX". A second transition goes from „state" directly to „stateY".



Now the sub-state machine is hidden in an own diagram. An entry point is necessary to allow the transition from „stateA" directly to „stateY".





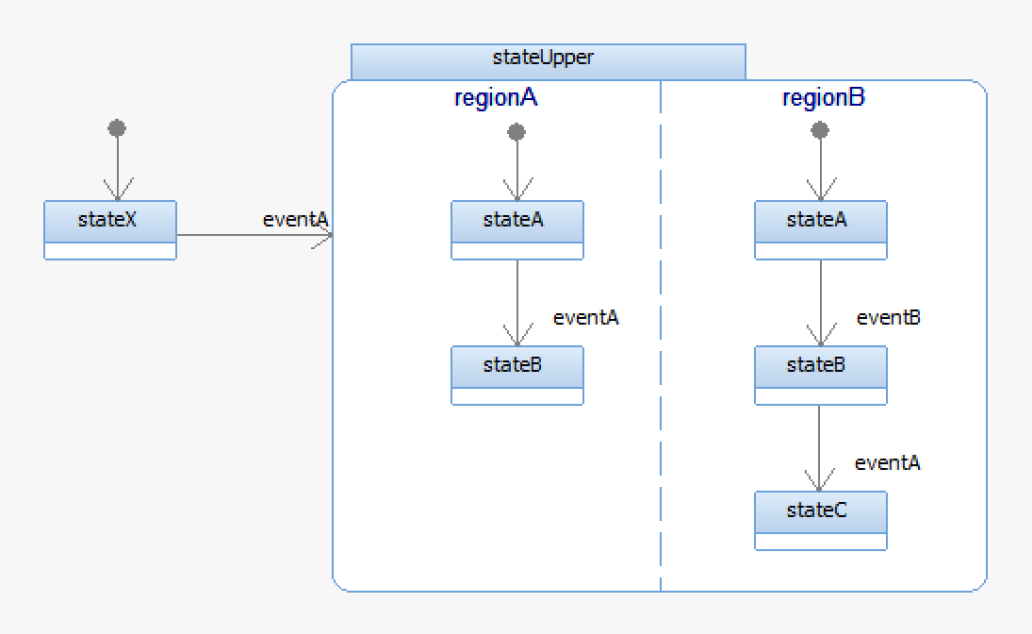
Parallel State Machines

A state can be divided into multiple state machines inside parallel regions. Again: All statements from above are still valid, each state machine follows the mentioned rules. Additional rules:

- A trigger that is sent to the „upper" state is handled by the state machines of all regions of the state.
- All regions use the properties of the owning classifier (block / use case / …).

Questions:

- Assume the current state is „stateX" . What will happen, if „eventA" is sent?
- Assume the current state is „stateUpper" and there „stateA" inside „regionA" and „stateA" inside „regionB". What will happen if „eventA" is sent?



Package / Profile Diagram

Model Elements

Packages, Profiles, Dependencies, Stereotypes, Tags, Extensions.

Package

A package is used to group elements, and provides a namespace for the grouped elements.

The public contents of a package are always accessible outside the package through the use of qualified names. In case you want to use the name only (instead of the qualified name) you must specify an Import- or an Access-Dependency as shown in the above example.

Profile

The Profile mechanism has been defined specifically for providing a lightweight extension mechanism to the UML standard. In UML 1.*, stereotypes and tagged values were used as string-based extensions that could be attached to UML model elements in a flexible way. In UML 2.* it is defined as a specific meta-modeling technique, as an example see the extension between the stereotype "TestComponent" and the classifier "Class" from the meta-model.

In addition it is possible to interchange profiles between tools, together with models to which they have been applied, by using the UML XMI interchange mechanisms. A profile usually is defined as an interchangeable UML model.

Profiles are applied to a given package.

Dependencies

Only an "Import" or an „Access"-Dependency allows the access to the namespace of the independent package. An "Apply"-Dependency is drawn from a package to a profile, it specifies that elements inside the package can use the extensions specified in the profile.

Stereotypes

Stereotypes defines how an existing element of the metamodel is extended. It means a stereotype cannot be used by itself in a model, it can only be used in conjunction with the extended element. A Stereotype can have properties, these are tagged values. A stereotype can extend all elements of the metamodel, but not the element "Stereotype". When a stereotype is applied to a model element, the syntax of that element cannot be changed.

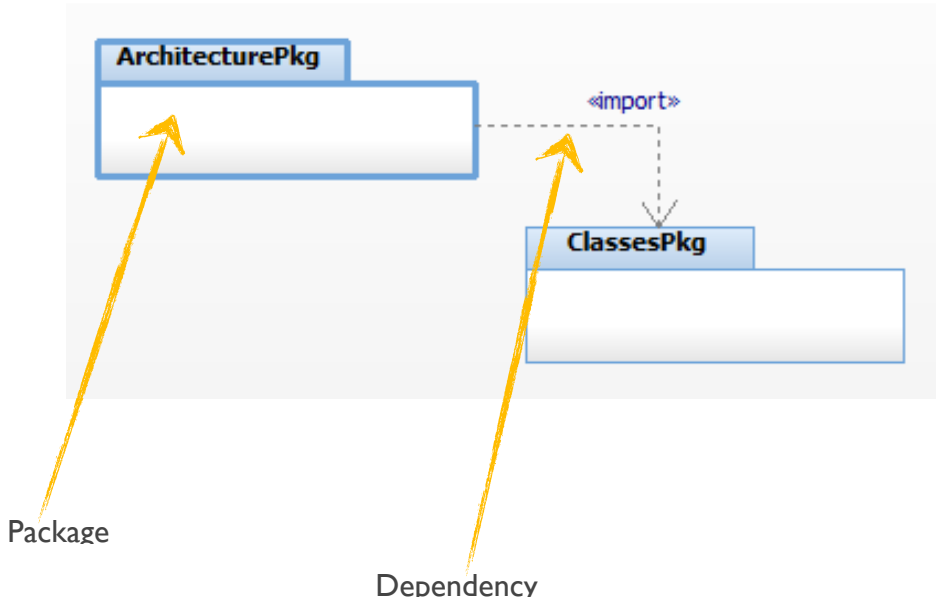
Extension

Extension is a special kind of a relation. An extension is used to indicate that an element of the metaclass are extended through a stereotype, and gives the ability add stereotypes to classes. Notation:

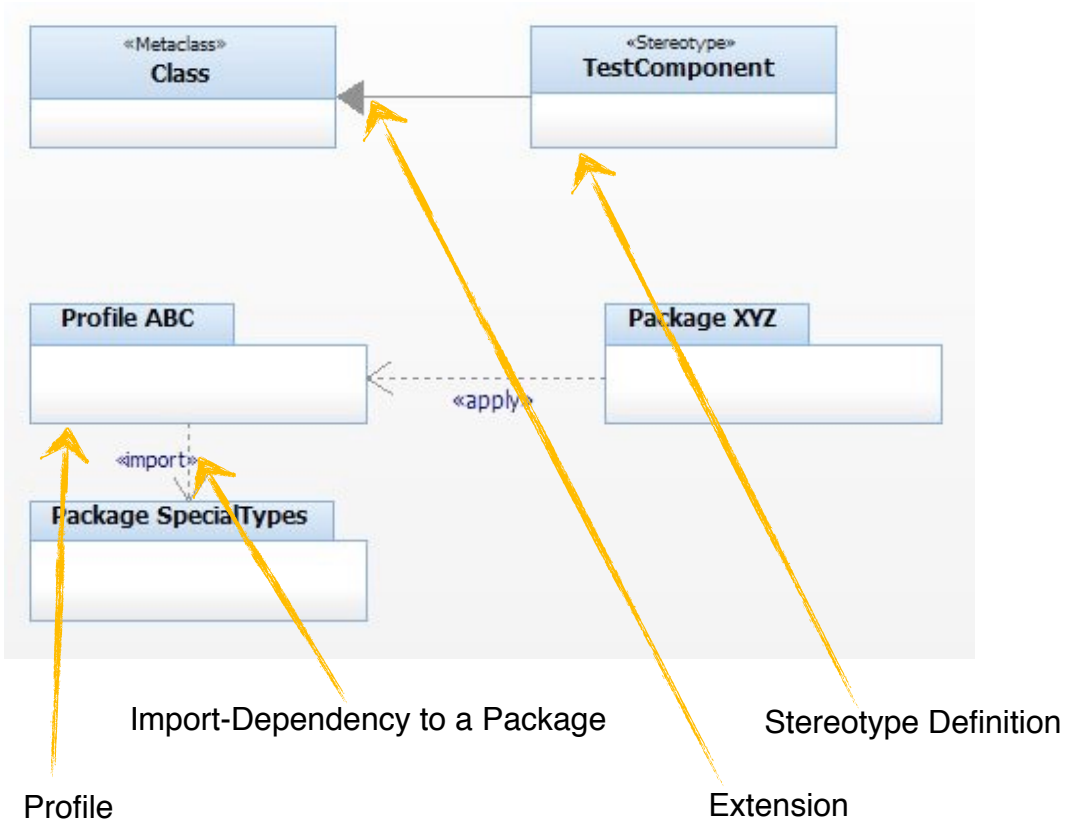
Tag

A tagged value can only be represented as an attribute defined on a stereotype. Therefore, a model element must be extended by a stereotype in order to be extended by tagged values. A Tag is just a literal without having a type.

Package Diagram Example



Profile Diagram Example



Notes

UMLforum.de

Product:

IBM® RATIONAL® RHAPSODY® START-UP TRAINING

Author:

Wolfgang Sonntag

Editor:

WILLERT SOFTWARE TOOLS GMBH

Hannoversche Straße 21
DE - 31675 Bückeburg
Phone: +49 5722 9678 - 60
info@willert.de
www.willert.de

Premier Business Partner