UML Reference Sheets

© 2013, 2014 Michael Marking; all rights reserved, including moral rights.
Web site: *http://www.tatanka.com/*

Revision Information

This document was last revised 2014.03.02. The current version should be available at *http://www.tatanka.com/engineering/miscellany/umldocs/*.

License and Terms of Use

This document may be redistributed freely, but only if distribution is done without changes, abridgement, or amendment. Give it away free, charge for it, print it, or post it on the Internet, as long as you comply with these terms. Proper attribution is made by leaving the entire document intact and without modification. Specifically, this work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License. To view a copy of this license, visit *http://creativecommons.org/licenses/by-nd/3.0/* or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Contents:

Relationships — Page 2
Dependency Relationships — Page 3
Classifiers — Page 5
Interfaces — Page 6
Interfaces, Use Case, Actor, State — Page 7
States — Page 8
State Diagrams — Page 9

UML is so large that, for someone who doesn't use it frequently, it's easy to forget the notation. The goal of this document is to provide a reference to the major parts of the language. I'm not trying to teach how to use UML, merely to jog one's memory. There is often more than one way to describe a model in UML. Clarity is by far the most important goal. The specification is colour-blind, so use colour any way you want, to be more clear.

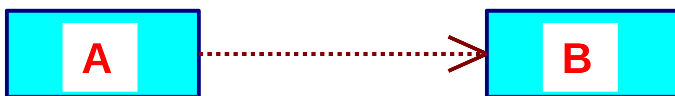
Note: Some elements from SysML, the UML extension for system modelling, are incorporated into these diagrams. For the official definitions, refer to the Object Management Group web site (*http://www.omg.org*); specifications are freely downloadable from there. Please let me know of any errors, ambiguities, shortcomings, or other problems by writing to *marking@tatanka.com*.

association: There is a relationship among instances of A and instances of B.

navigability: A can use B in an expression to navigate to B, or to express constraints.

non-navigability: A cannot navigate to B.

generalization: B generalizes A; B is a superclass of A; A is a subclass of B. A specializes B. Generalization is a *relationship*, whereas inheritance is a *process* which uses generalization to create descriptions, although sometimes this notation is wrongly described as "A inherits from B".



dependency: A depends on B. *(More details on next page.)*

aggregation: A is a constituent of B. However, A has an existence independent of B.

composition: A is exclusively a part of B, not a part of anything else, and with a coincident lifetime.

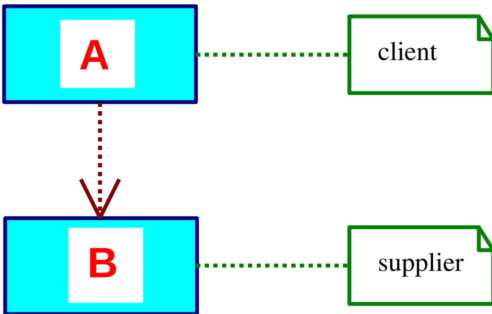
membership or containment: A is a member of package B; B contains A.

ownership of an association end: A owns the end at B of the association between A and the classifier B.

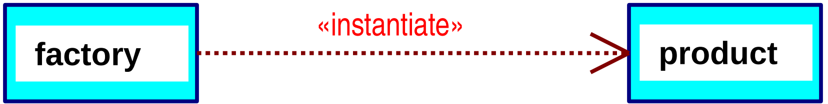
realization: A realizes, or provides an implementation for, B. *(More details under Dependency Relations, on next page.)*

©2013, 2014 Michael Marking. All rights reserved.

A **dependency** relationship is written as a dashed line with an arrow. The arrowhead is open, except that, in a **realization** (one special kind of dependency relationship), the arrowhead is closed and hollow. The element at the tail of the arrow is the **client**; the element at the arrowhead is the **supplier**. The client depends on the supplier. The semantics of the client are not complete without the supplier.



Keywords or stereotype names in guillemets may be used to clarify the type of dependency relationship. For example, the «instantiate» keyword describes the relationship between a factory and its product.



Note: The "direction" of relationship dependencies may, in some cases, seem perverse. One might think it more logical that a product would depend on its factory, rather than the other way around. These are, however, relationships of meaning and function, not necessarily chronological relationships. If the supplier changes, then the meaning or function of the client must, logically, change. Put another way, the factory was built and exists in order to build the product, not the converse, so the product is the controlling logical entity. We say "toy factory" or "appliance factory", not usually the "factory-built toy". In a very few cases the opposite logic might apply.

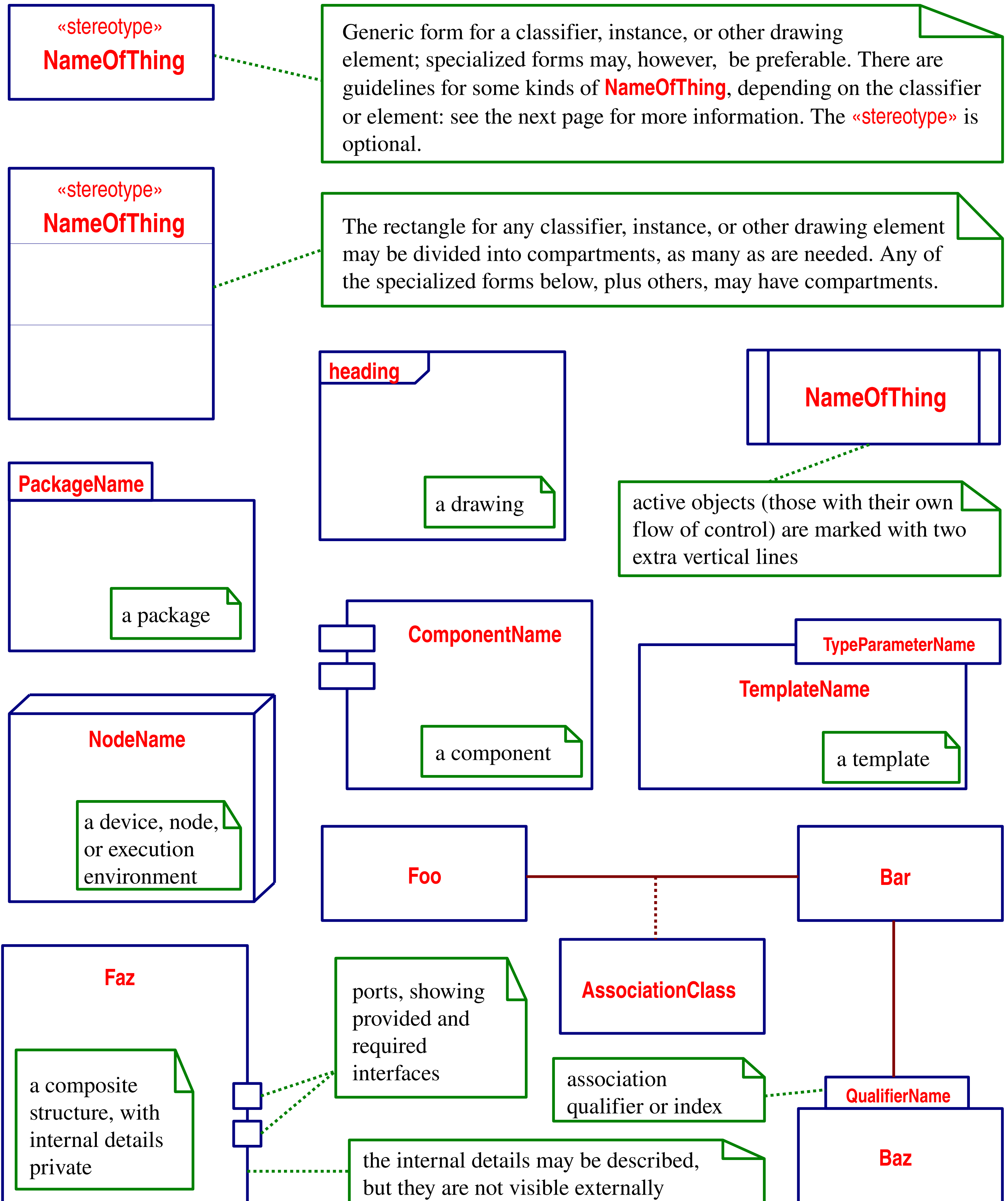
Keywords to be used with dependency relationships:

Keyword	Client	Supplier	Description
«access»	package	package	Names are imported from supplier (source) to client (target), with private visibility.
«apply»	model or package	profile	Make some or all metaclasses from the supplier profile available for use in the client model or package.
«call»	classifier	classifier	The source operation, or an operation within the client class, invokes the target operation, or an operation within the supplier class.

©2013, 2014 Michael Marking. All rights reserved.

Keyword	Client	Supplier	Description
«conform»	view	viewpoint	Properties of interest to stakeholders in «view» package conform to those of «viewpoint» package.
«copy»	requirement	requirement	Client copies requirement from supplier.
«create»	classifier	classifier	The client creates instances of the supplier.
«derive»	element	element	The client may be computed from the supplier.
«deriveReqt»	requirement	requirement	The client requirement is based upon analysis of the supplier (source) requirement; that is, it satisfies the source requirement in the client context.
«extend»	use case	use case	Client adds functionality to supplier.
«import»	package	package	Names are imported from supplier (source) to client (target), with public visibility.
«include»	use case	use case	Client includes functionality of supplier.
«instantiate»	classifier	classifier	Operations on the client create instances of the supplier.
«realize»	implement-ation	specific-ation	Client provides implementation of a supplier specification. Usually, instead of the keyword, this dependency relationship is represented by a dashed line with a closed, hollow arrowhead.
«reference»	profile	metaclass or metamodel	The client profile imports from the supplier metaclass or metamodel.
«refine»	element	element	In a mapping between two different semantic levels, the supplier is the base for the more-developed client.
«responsibility»	element	element	Client has obligation or responsibility under contract to supplier.
«satisfy»	element	requirement	The client element satisfies the supplier requirement.
«send»	operation	signal	Signal is sent by client.
«trace»	element	element	Client represents concept in one model related to supplier in another model. May relate model elements to documents.
«use»	classifier	classifier	The supplier classifier's presence is required for the correct functioning of the client.
«verify»	element	requirement	The client element, perhaps a test case, establishes that the supplier requirement is satisfied.

©2013, 2014 Michael Marking. All rights reserved.



©2013, 2014 Michael Marking. All rights reserved.

The names of objects should be underlined, and take one of the forms Instance or Class:Instance.

Class names, including interface names, are centred, begin with a capital letter, and are written in boldface: **ClassName**. The names of abstract classes are italicized.

«interface»

InterfaceName

+ Attribute1 : Type1
– Attribute2 : Type2

+ Operation1 (...) : ...
– Operation2 (...) : ...

The general syntax of an attribute is:
<visibility> <derived marker> <attribute name>
<attribute type> <multiplicity> <default value>
<properties> <constraints>

<visibility> is one of:

+	public	**#**	protected
–	private	**~**	package

The *<derived marker>*, if present, is a slash or virgule (**/**)

<multiplicity> is a value or range, in square brackets, with the values separated by **..**, and ***** indicating an unbounded value (Examples: [0..*], [*], [3])

The *<default value>* is preceded by an equals sign, **=**

The *<properties>* are a list of comma-separated strings or names, enclosed in curly braces, **{ }**

<constraints> are statements, also enclosed in curly braces

The general syntax of an operation is:
<visibility> <operation name> **(** *<attribute type>* **)**
: *<return type> <properties>*

<visibility> and*<properties>* are as for attributes

The parameter specifications are separated by commas; each has the syntax
<direction> <parameter name> **:** *<parameter type>*
<multiplicity> <default value> <properties>

The *<direction>* is one of in, out, or inout

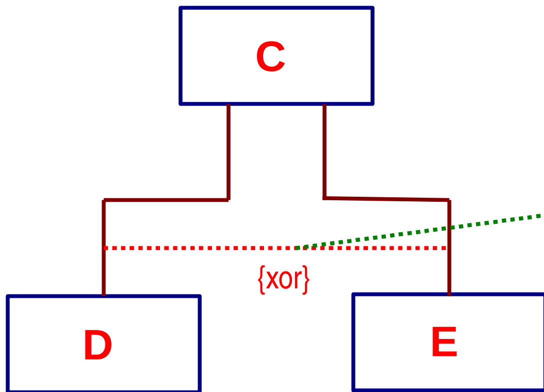
Other parts of the definition are as for attributes

The descriptions of static operations (those with class scope) are underlined.

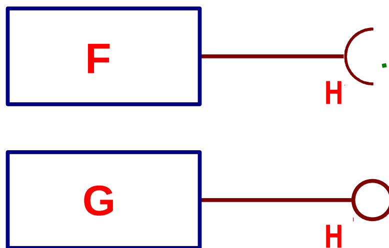
Additional compartments may be added as needed. The most common use for additional compartments in interfaces is to describing possible exceptions.

©2013, 2014 Michael Marking. All rights reserved.

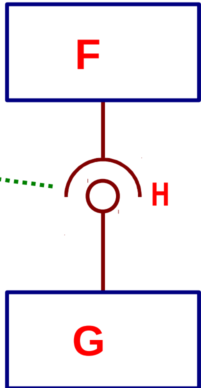
Alternative notation for attributes: Attribute1 of A associates B with A. (Usually, Attribute1 is a pointer or name.)

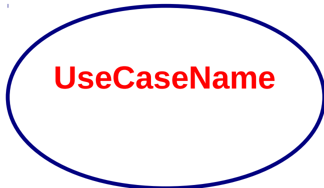


The {xor} annotation indicates that one, but not both, of the associations is possible at any given time.



F requires interface H, and G provides interface H. These may be written separately, as to the left, or together, as to the right. The icons may be attached directly to a classifier, or they may be attached to a port on the classifier.



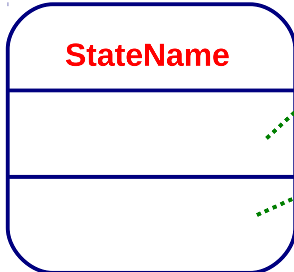


Actor. You may also use a rectangle with the <<actor>> stereotype.

Compartment for listing internal activities. Each has the syntax

<label> / <expression>

where <expression> may be written in pseudocode or any other suitable form. Three labels are reserved: Entry, Exit, and Do, but other labels may be used.

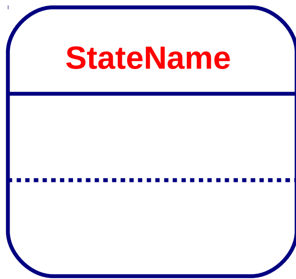


Compartment for listing internal transitions. Each has the syntax

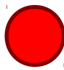
<event> (<attribute list>) [<guard condition>] / <effect>

The <guard> conditions evaluate to boolean values, and enable or disable the trigger effect of the <event>. The <attribute list> is an optional list of parameters.

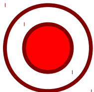
©2013, 2014 Michael Marking. All rights reserved.



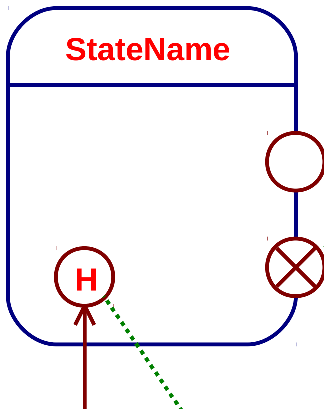
State with two regions; any number of regions is permitted. Each region contains a separate substate machine. The substate machines operate in parallel, they are entered simultaneously when the superstate is entered. Unless specified otherwise, the superstate exits when all substate machines have completed or terminated.



The initial pseudo-state. This is the starting point of the state machine, unless the entry pseudo-state is used.



The final pseudo-state. When the state machine reaches this state, there are no transitions to any other state. The machine remains in the final state until the machine is destroyed, and internal activities associated with that state will continue. A machine may have more than one final state.



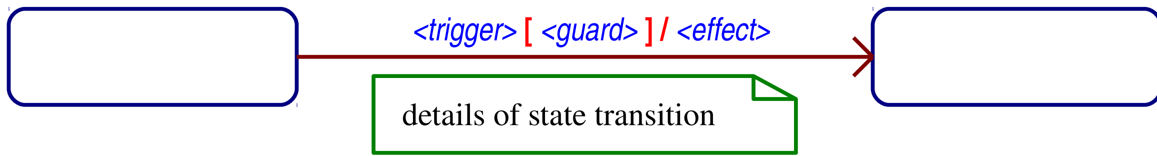
The entry (above) and exit (below) pseudo-states. These are placed on the boundaries of a state machine, and allow (using arrows) transitions between the internal states of the machine and external states or events. A transition inward from the entry pseudo-state may be used in lieu of using the initial state.

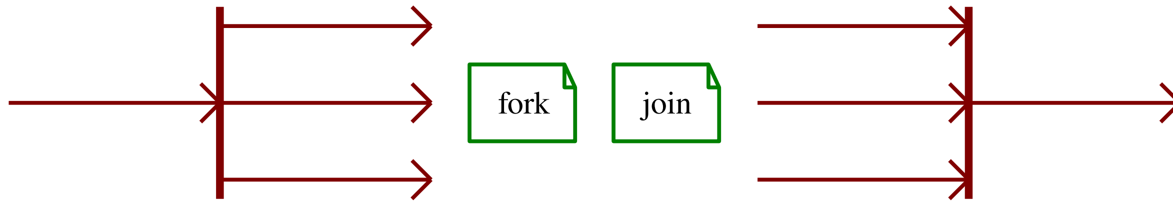
The shallow history pseudo-state. When this sub-state is entered from the outside, it is like a "return": the sub-state machine picks up from where it was when it was last exited. Thus, it may have no outgoing arrows, and may appear to go nowhere, but it jumps to the last sub-state the machine was in. If there is an outgoing arrow, only one is permitted, and it indicates the default sub-state, in case the super-state has never before been entered.

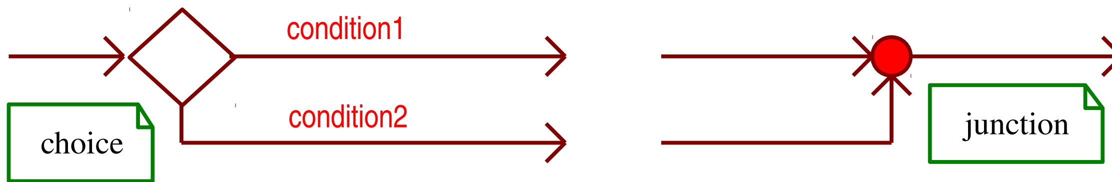
The deep history pseudo-state. Similar to the shallow history pseudo-state, except that a transition to the deep history pseudo-state causes not only a resumption from the last sub-state, but also causes resumption of all contained sub-sub-state machines beginning at their respective sub-sub-states.

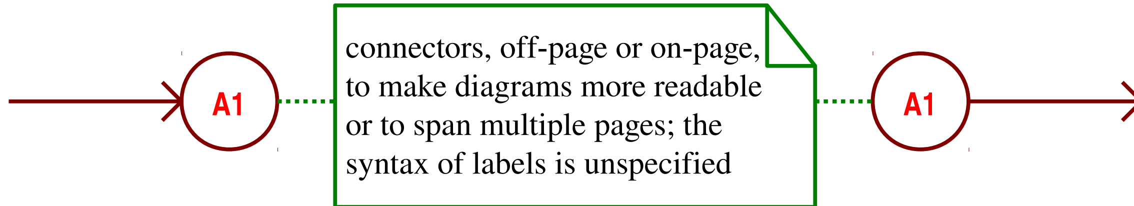
The terminate pseudo-state. Unlike the final state, entry into the terminal state causes the machine to halt completely; there will be no activity and the instance of the machine is effectively destroyed.

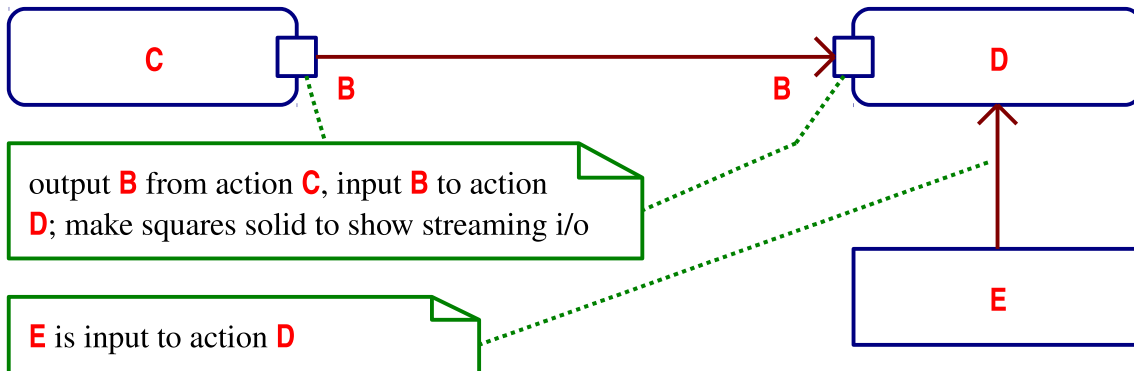
©2013, 2014 Michael Marking. All rights reserved.

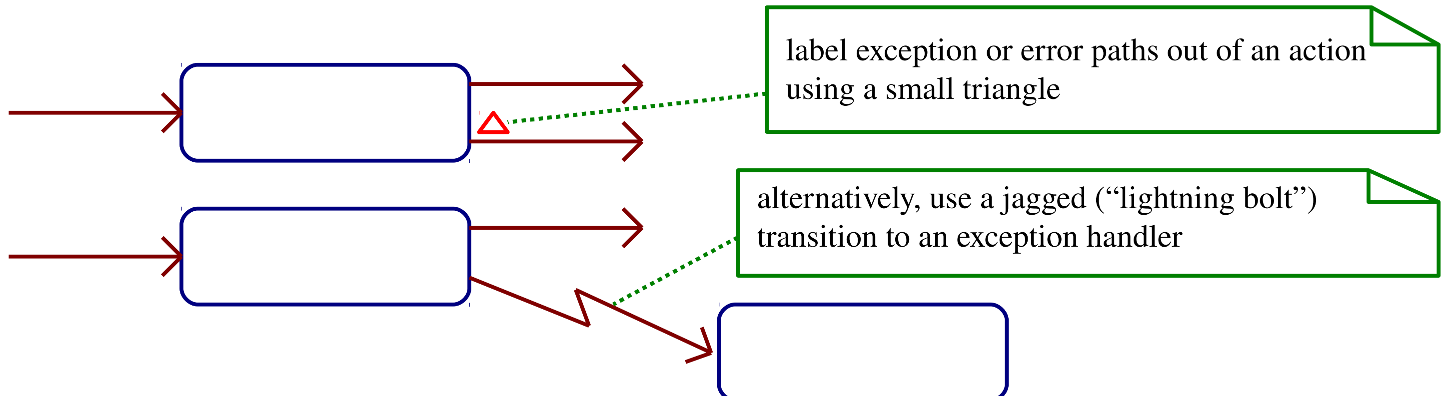












©2013, 2014 Michael Marking. All rights reserved.