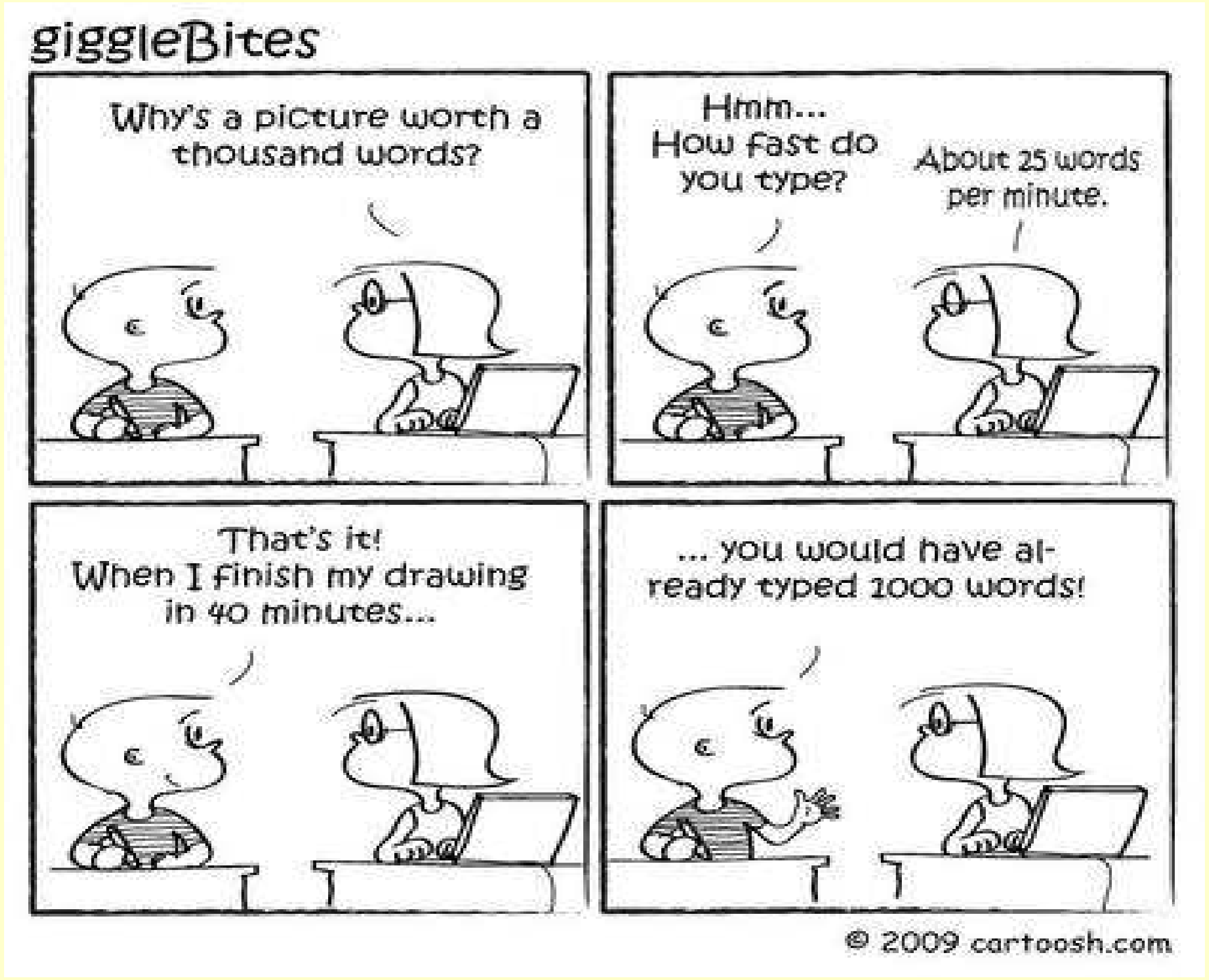
UML - Introduction



giggleBites
Why's a picture worth a thousand words?
Hmm... How fast do you type?
About 25 words per minute.
That's it! When I finish my drawing in 40 minutes...
... you would have al-ready typed 1000 words!
© 2009 cartoosh.com

Overview: Edited From Wikipedia, the free encyclopedia

The **Unified Modeling Language (UML)** is a standardized specification language for object modeling.

UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system, referred to as a *UML model*.

UML is officially defined at the Object Management Group (OMG) by the UML metamodel, a Meta-Object Facility metamodel (MOF).

UML was designed to specify, visualize, construct, and document software-intensive systems.

UML is not restricted to modeling software. UML is also used for business process modeling, systems engineering modeling, and representing organizational structures.

Overview: Edited From Wikipedia, the free encyclopedia

The Systems Modeling Language (SysML) is a Domain-Specific Modeling language for systems engineering that is defined as a UML 2.0 profile.

UML has been a catalyst for the evolution of model-driven technologies, which include Model Driven Development (MDD), Model Driven Engineering (MDE), and Model Driven Architecture (MDA).

By establishing an industry consensus on a graphic notation to represent common concepts like classes, components, generalization, aggregation, and behaviors, UML has allowed software developers to concentrate more on design and architecture.

UML models may be automatically transformed to other representations (e.g. Java) by means of transformation languages, supported by the OMG.

UML is extensible, offering the following mechanisms for customization: profiles and stereotype. The semantics of *extension by profiles* has been improved with the UML 2.0 major revision.

UML Founding Fathers: The *three amigos*

James Rumbaugh: **OMT** – ‘best’ for OOA

James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen: *Object-Oriented Modeling and Design*, Prentice Hall, ISBN 0-13-629841-9

Grady Booch: **Booch method** – ‘best’ for OOD

(1993). *Object-oriented Analysis and Design with Applications*, 2nd ed., Redwood City: Benjamin Cummings. ISBN 0-8053-5340-2.

Ivar Jacobson: **OOSE method** - first object-oriented design methodology to employ use cases.

Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley Professional (1992). ISBN 0201544350

UML Founding Fathers: The *three amigos*

Rational Machines was founded by Paul Levy and Mike Devlin in 1981 to provide tools to expand the use of modern software engineering practices, particularly explicit modular architecture and iterative development.

Grady Booch was a chief scientist at Rational, working on graphical notations

Rational Software Corporation hired **James Rumbaugh** from General Electric in 1994, the company became the source for the two most popular object-oriented modeling approaches of the day (OMT and Booch Method)

Together Rumbaugh and Booch attempted to reconcile their two approaches and started work on a Unified Method.

Joined by **Ivar Jacobson**, the creator of the OOSE method, in 1995, after his company, Objectory, was acquired by Rational.

The three methodologists were collectively referred to as the *Three Amigos*, since they were well known to argue frequently with each other regarding methodological preferences. (Jacobson: "What's the difference between a terrorist and a methodologist? You can negotiate with a terrorist".)

Rational was sold for $2.1B to IBM on February 20, 2003

Development History

Under the technical leadership of the Three Amigos, an international consortium called the UML Partners was organized in 1996 to complete the *Unified Modeling Language (UML)* specification, and propose it as a response to the OMG RFP.

The UML Partners' UML 1.0 specification draft was proposed to the OMG in January 1997. During the same month the UML Partners formed a Semantics Task Force, chaired by Cris Kobryn and administered by Ed Eykholt, to finalize the semantics of the specification and integrate it with other standardization efforts.

The result of this work, UML 1.1, was submitted to the OMG in August 1997 and adopted by the OMG in November 1997.

Development History

As a modeling notation, the influence of the OMT notation dominates (e.g., using rectangles for classes and objects).

Though the Booch "cloud" notation was dropped, the Booch capability to specify lower-level design detail was embraced.

The use case notation from Objectory and the component notation from Booch were integrated with the rest of the notation, but the semantic integration was relatively weak in UML 1.1

This was not really fixed until the UML 2.0 major revision.

Development History

The Unified Modeling Language is an international standard:
ISO/IEC19501:2005 Information technology -- Open Distributed Processing -- Unified Modeling Language (UML) Version 1.4.2.

UML has matured significantly since UML 1.1. Several minor revisions (UML 1.3, 1.4, and 1.5) fixed shortcomings and bugs with the first version of UML, followed by the UML 2.0 major revision, which is the current OMG standard.

Development History

The first part of UML 2.0, the Superstructure which describes the new diagrams and modeling elements available, was adopted by the OMG in October 2004. Other parts of UML 2, notably the infrastructure, the Object Constraint Language (OCL) and the diagram interchange were ratified later.

The final UML 2.0 specification has been declared *available* and has been added to OMG's formal specification library. The other parts of the UML specification, the UML 2.0 infrastructure, the UML 2.0 Diagram Interchange, and UML 2.0 OCL specifications have been *adopted*.

UML version 2.1 revision is being developed, and should be available in the form of an XMI 2.1 version of the UML 2.1 version. The corresponding XMI 2.1 file will be made available from the OMG ADTF group.

Most of the commercially successful UML tools now support most of UML 2.0

Modeling

It is very important to distinguish between the UML model and the set of diagrams of a system.

A diagram is a partial graphical representation of a system's model. The model also contains a "semantic backplane" — documentation such as written use cases that drive the model elements and diagrams.

There are three prominent parts of a system's model:

Functional Model
: Showcases the functionality of the system from the user's Point of View.
: Includes Use case diagrams.

Object Model
: Showcases the structure and substructure of the system using objects, attributes, operations, and relationships.
: Includes Class Diagrams.

Dynamic Model
: Showcases the internal behavior of the system.
: Includes sequence diagrams, activity diagrams and state machine diagrams.

Models can be exchanged among UML tools by using the XMI (XML Metadata Interchange) format.

UML Diagrams

In UML 2.0 there are **13 types of diagrams**. To understand them, it is sometimes useful to categorize them hierarchically:

Structure Diagrams emphasize what things must be in the system being modeled:

* Class diagram

* Component diagram

* Composite structure diagram

* Deployment diagram

* Object diagram

* Package diagram

UML Diagrams continued …

Behavior Diagrams emphasize what must happen in the system being modeled:

* Activity diagram

* State Machine diagram

* Use case diagram

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

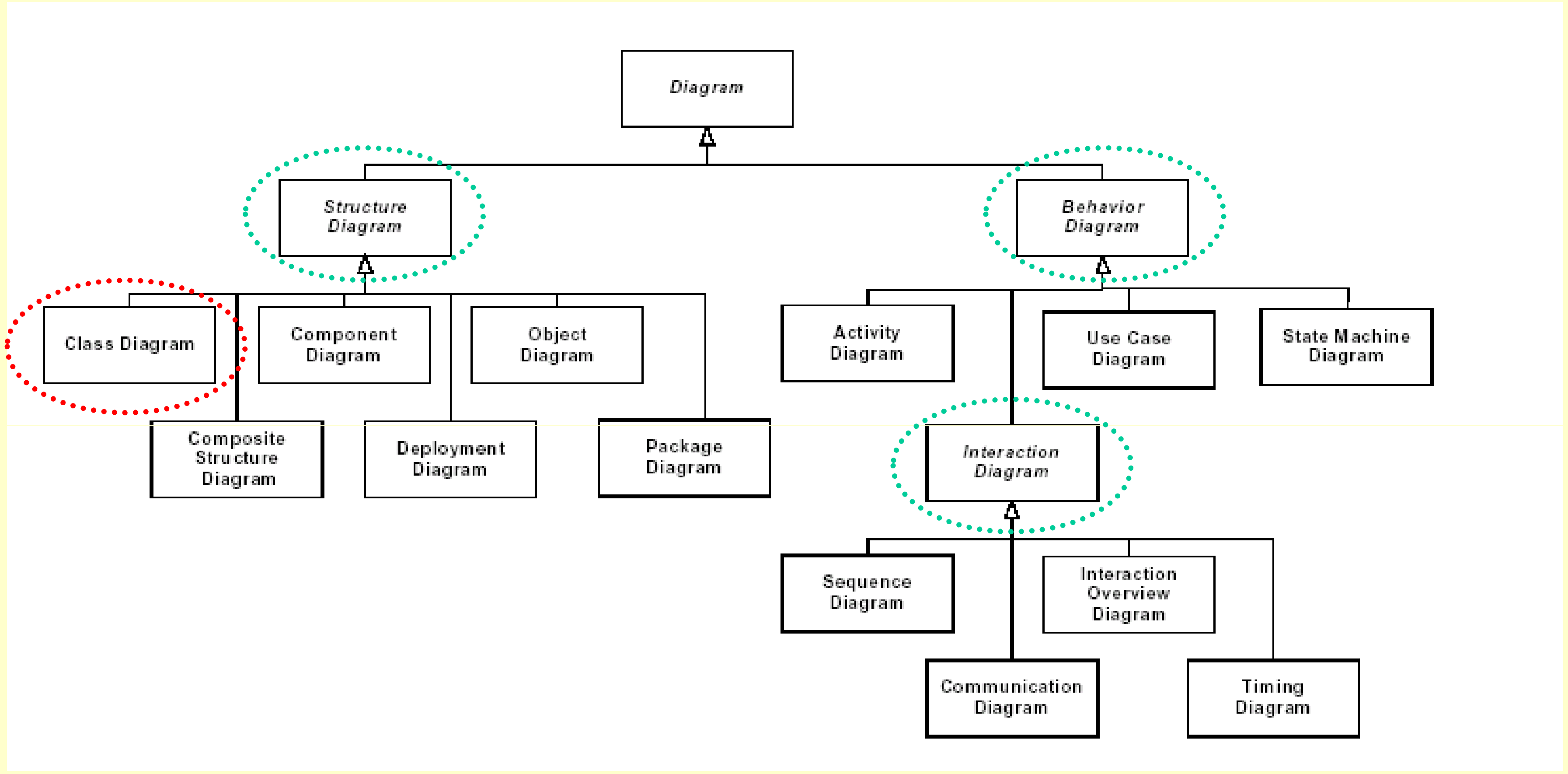
* Communication diagram

* Interaction overview diagram (UML 2.0)

* Sequence diagram

* UML Timing Diagram (UML 2.0)

A (UML Class) Diagram of the 3 main categories of UML diagram



UML Diagrams: Learning Priorities

Typical

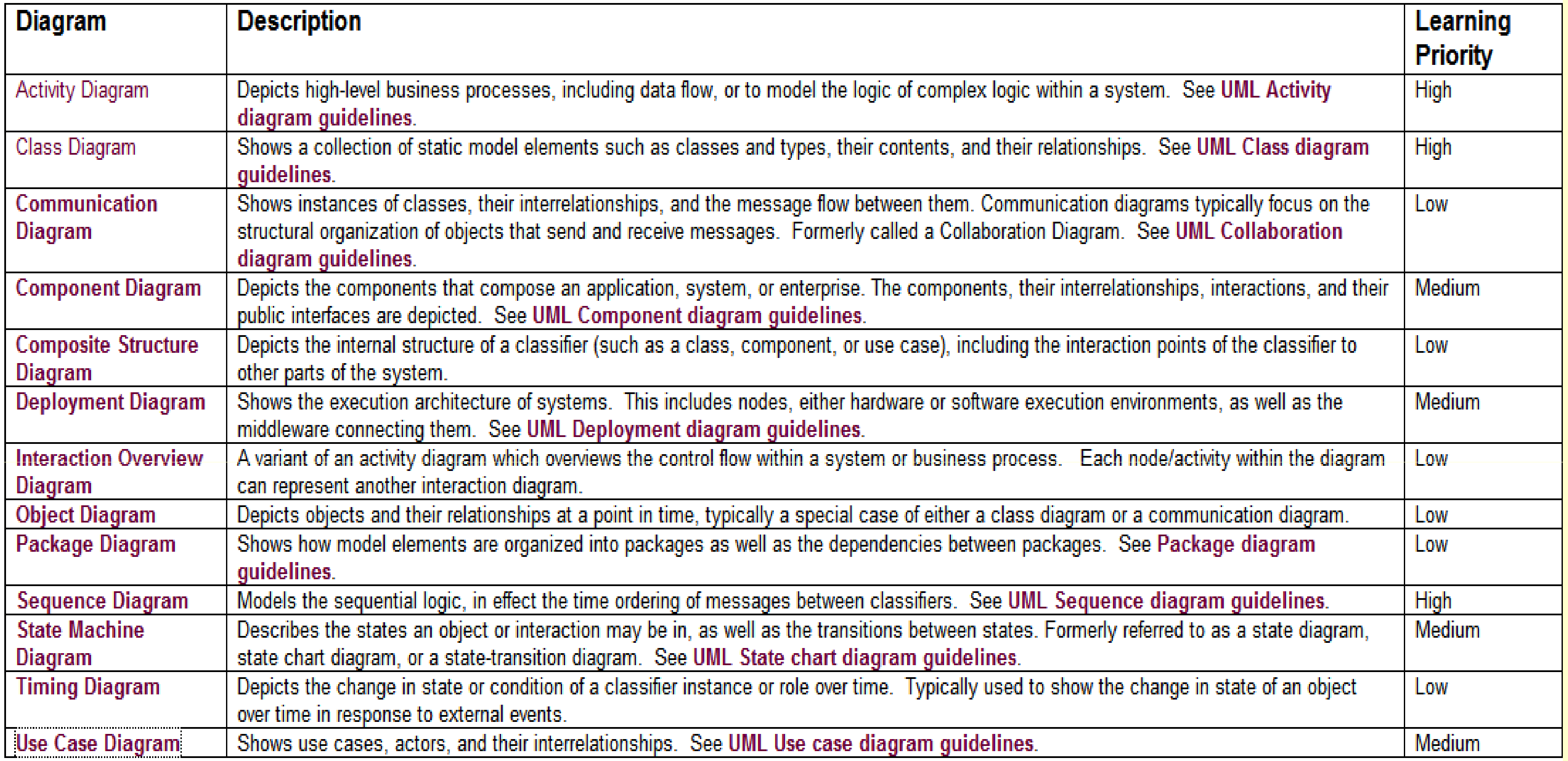


Diagram	Description	Learning Priority
Activity Diagram	Depicts high-level business processes, including data flow, or to model the logic of complex logic within a system. See UML Activity diagram guidelines.	High
Class Diagram	Shows a collection of static model elements such as classes and types, their contents, and their relationships. See UML Class diagram guidelines.	High
Communication Diagram	Shows instances of classes, their interrelationships, and the message flow between them. Communication diagrams typically focus on the structural organization of objects that send and receive messages. Formerly called a Collaboration Diagram. See UML Collaboration diagram guidelines.	Low
Component Diagram	Depicts the components that compose an application, system, or enterprise. The components, their interrelationships, interactions, and their public interfaces are depicted. See UML Component diagram guidelines.	Medium
Composite Structure Diagram	Depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system.	Low
Deployment Diagram	Shows the execution architecture of systems. This includes nodes, either hardware or software execution environments, as well as the middleware connecting them. See UML Deployment diagram guidelines.	Medium
Interaction Overview Diagram	A variant of an activity diagram which overviews the control flow within a system or business process. Each node/activity within the diagram can represent another interaction diagram.	Low
Object Diagram	Depicts objects and their relationships at a point in time, typically a special case of either a class diagram or a communication diagram.	Low
Package Diagram	Shows how model elements are organized into packages as well as the dependencies between packages. See Package diagram guidelines.	Low
Sequence Diagram	Models the sequential logic, in effect the time ordering of messages between classifiers. See UML Sequence diagram guidelines.	High
State Machine Diagram	Describes the states an object or interaction may be in, as well as the transitions between states. Formerly referred to as a state diagram, state chart diagram, or a state-transition diagram. See UML State chart diagram guidelines.	Medium
Timing Diagram	Depicts the change in state or condition of a classifier instance or role over time. Typically used to show the change in state of an object over time in response to external events.	Low
Use Case Diagram	Shows use cases, actors, and their interrelationships. See UML Use case diagram guidelines.	Medium

Depending on what you need/want to learn

Criticisms

Although UML is a widely recognized and used modeling standard, it is frequently criticized for the following deficiencies -

Language bloat: UML is often criticized as being gratuitously large and complex.

Imprecise semantics: Since UML is specified by a combination of itself (abstract syntax), OCL (well-formedness rules) and English (detailed semantic), it lacks the rigor of a language precisely defined using formal language techniques.

Problems in learning and adopting: Language bloat and imprecise semantics make learning and adopting UML problematic, especially when management forces UML upon engineers lacking the prerequisite skills.

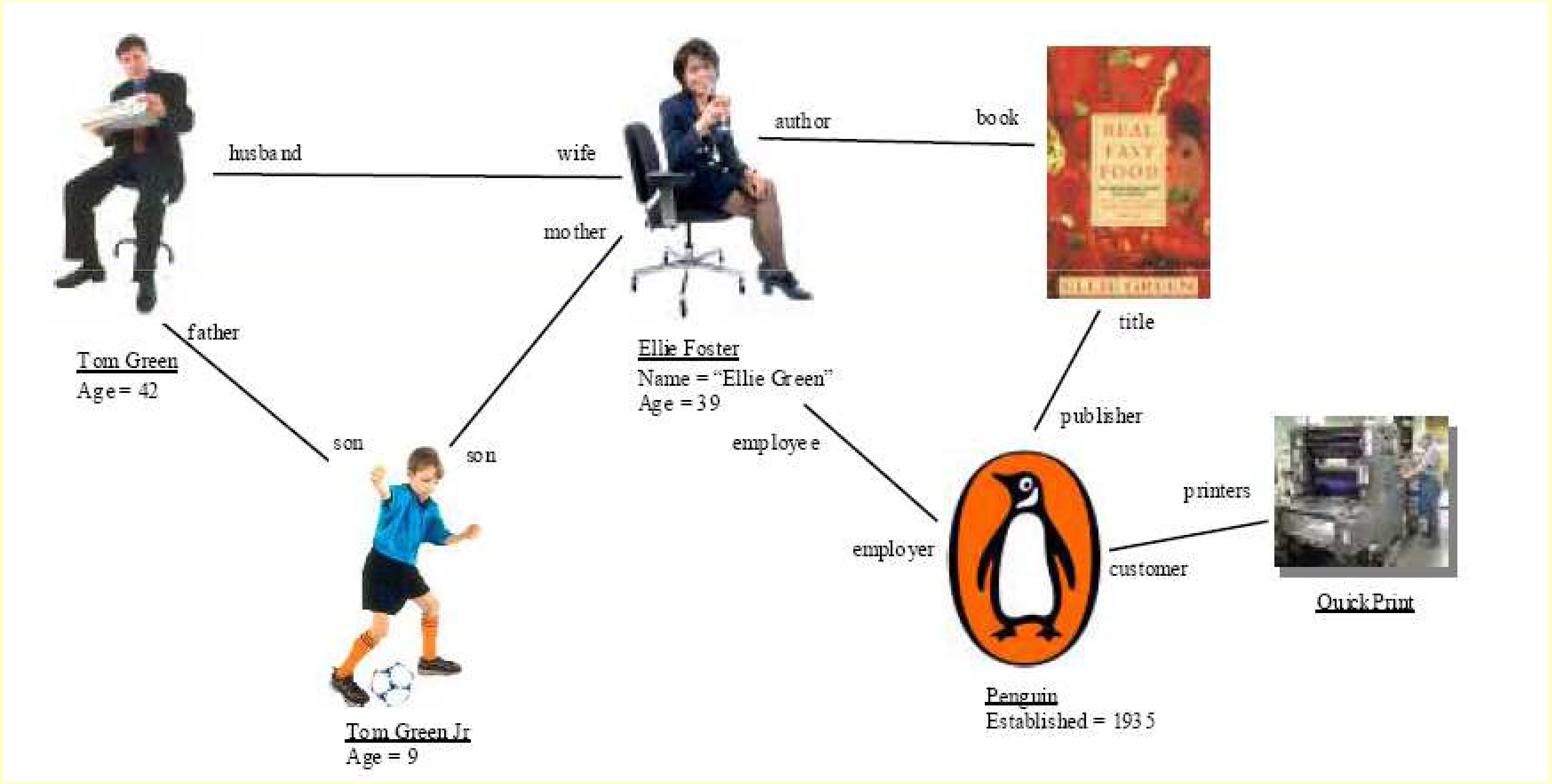
Only the code is in sync with the code: UML has value in approaches that compile the models to generate code. This however, may still not be sufficient since UML as a language does not exhibit Turing completeness, and any generated source or executable code would be limited to what a UML interpreting tool can discern or assume.

Cumulative Impedance/Impedance mismatch: As with any notational system, UML is able to represent some systems more concisely or efficiently than others.

Tries to be all things to all people: UML is a general purpose modeling language, which tries to achieve compatibility with every possible implementation language.

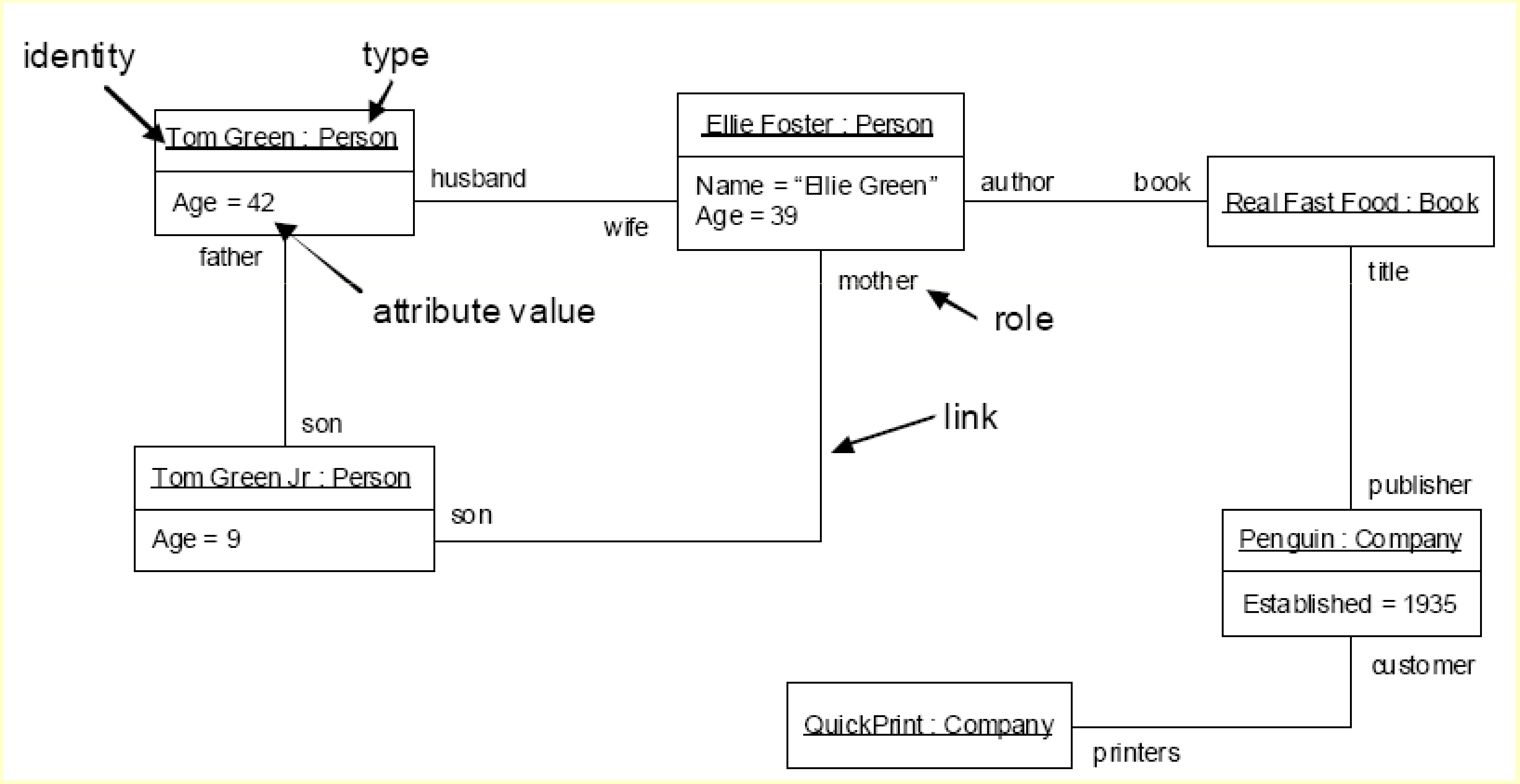
Introducing Notation: Object Diagrams

Unsurprisingly, as the basic currency of object orientation is objects, we can use UML to describe objects, their attributes - the facts that we can know about them - and the relationships between objects at some point in time.



Introducing Notation: Object Diagrams

More formally in UML:



Introducing Notation: Object Diagrams

It is important to note that **objects have a unique identity that remains unchanged throughout their lifetimes**. Ellie Foster is still the same person, even after she has changed her name to Ellie Green.

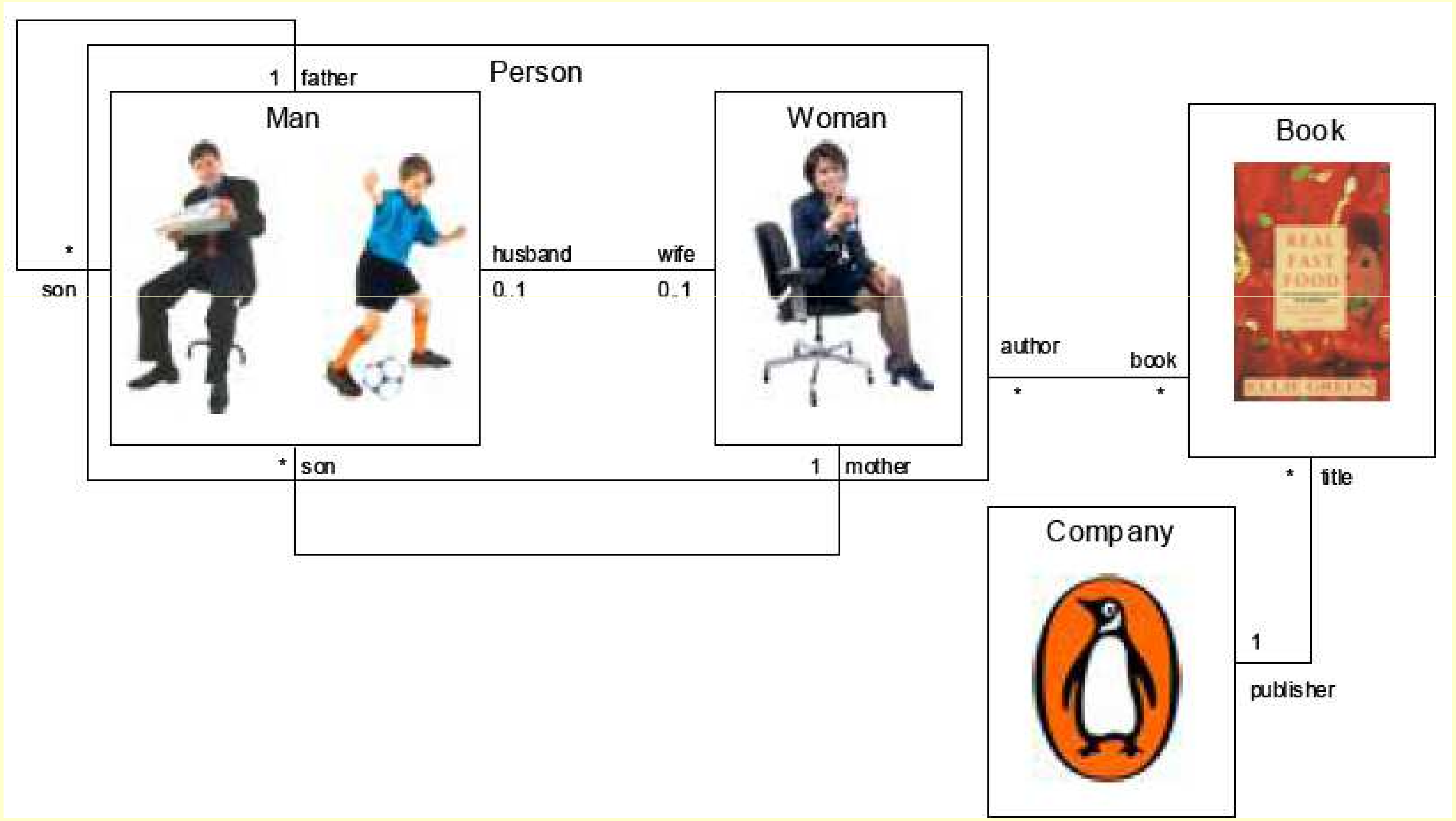
Another important thing to note about **objects** is that they **can play different roles** in respect of each other. They can play more than one role at the same time. For example, Ellie Foster is a wife to Tom Green, a mother to Tom Green Jr and the author of Real Fast Food.

As well as describing objects, **we can use UML to model types of objects**, that is, sets of similar objects that share the same characteristics. Types, or classes, as they are more commonly known in object oriented software development, tell us what attribute values any instance of that type is allowed to have, what roles objects of that type are allowed to play, and how many objects are allowed to play the same role with respect to the same object.

We use the term **multiplicity** to refer to the number of objects that are allowed to play the same role at the same time with respect to another object. For example, in the relationship mother->son, the role of son can be played by zero or more objects of type Man at the same time with respect to the same Woman, so the multiplicity of the role son is zero or more (or 0..*, or just *, in UML). In the reverse, only one Woman can play the role of mother with respect to the same Man, so the multiplicity of the role mother is exactly one (or simply 1 in UML).

Introducing Notation: Object Diagrams

Multiplicity (informally):



The relationship between objects and types (or classes).

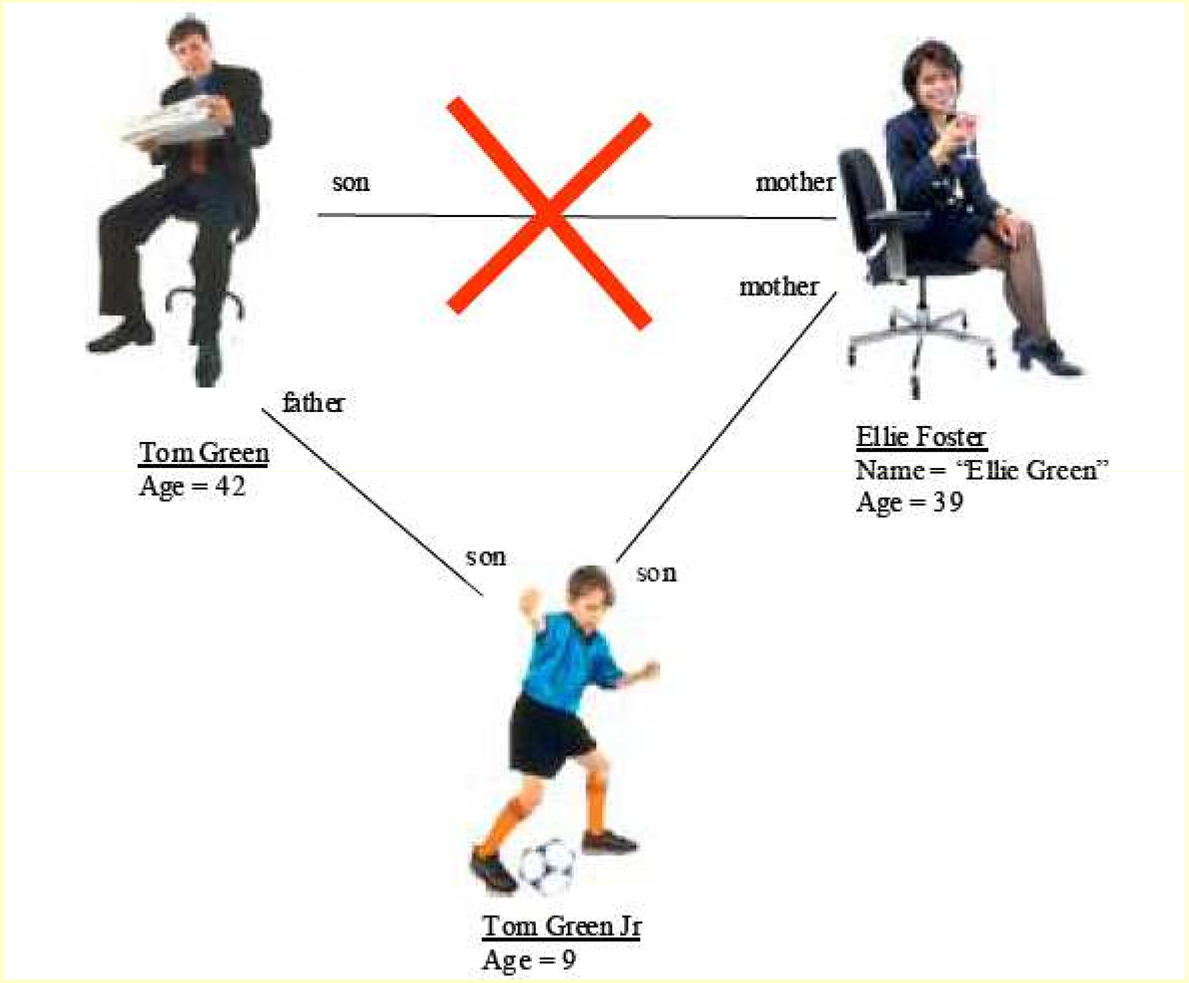
An object is said to be an instance of a type - a specific example of that type.

Every instance of a type has the characteristics defined by that type, and must obey any rules that apply to it. If our type model tells us that every Man has exactly one mother, for example, then an object of type Man with no mother, or with two mothers, does not conform to its type.

So a type, or class, model tells us the rules about what instances of objects, their attributes and the relationships between them are allowed. But quite often they do not tell us all of the rules. Sometimes rules about types can be more complex and subtle than, eg, a Man must have exactly one mother.

For example, how can we model the fact that a Woman cannot have a son who is also the father of any of her other sons? We can use object models to illustrate scenarios that might break these subtle rules.

The relationship between objects and types (or classes).

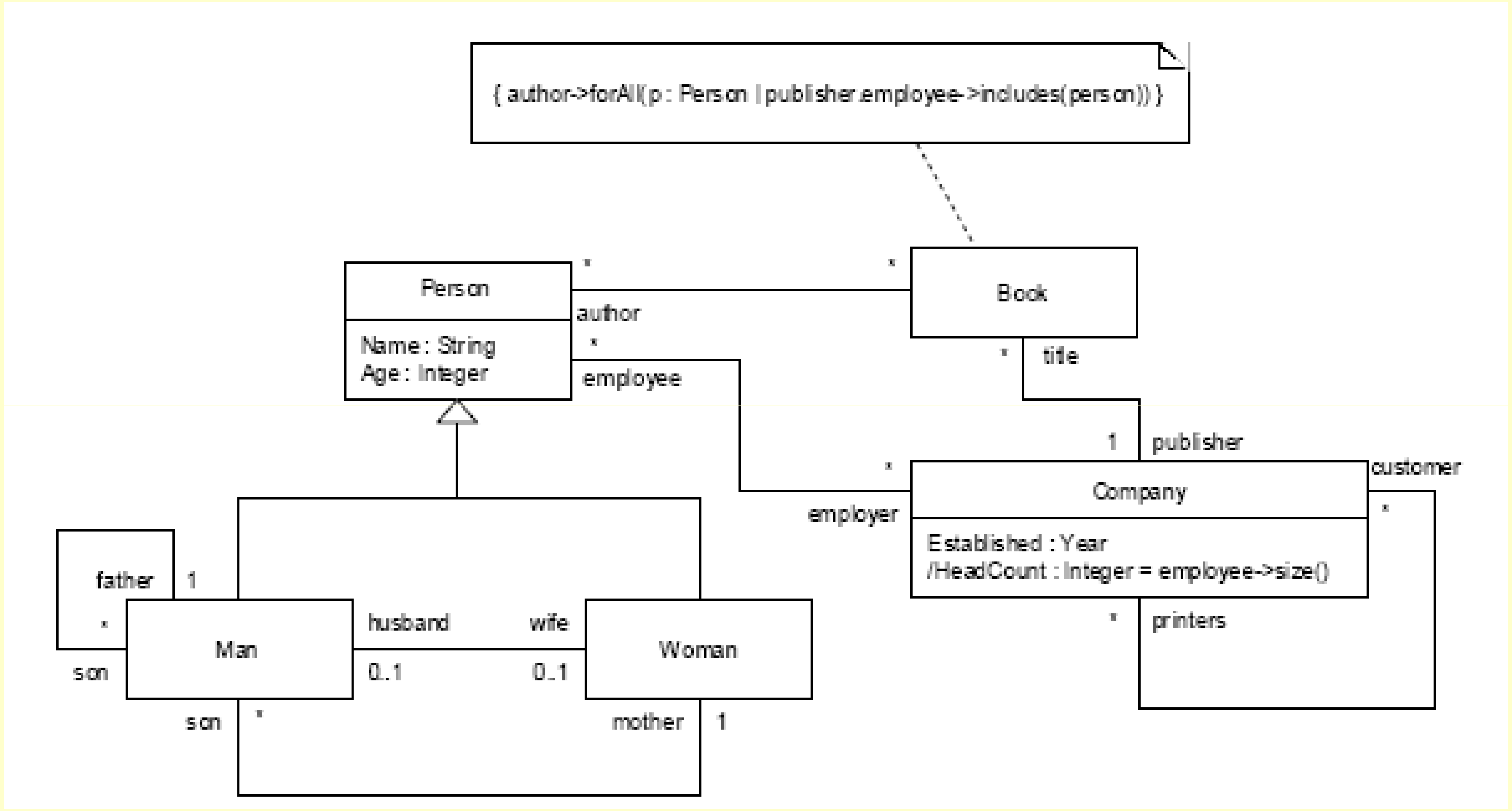


Ellie Foster cannot be the mother of Tom Green because he is the father of her son, Tom Green Jr.

This can be added to the model as a rule (or constraint):

{ **son** cannot be **father** of any other **sons** }

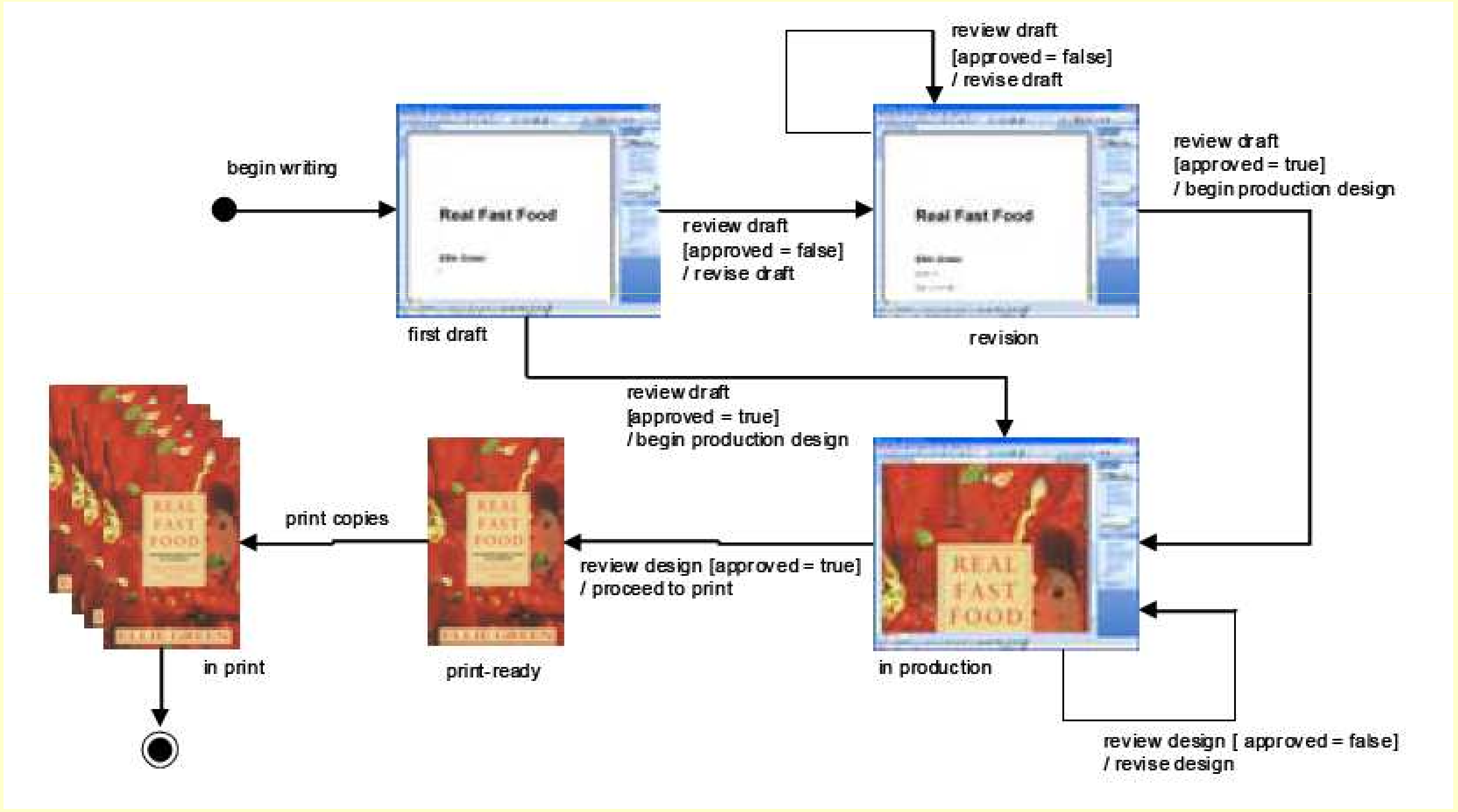
More formal class diagrams using OCR?



We can also combine UML with more formal models (in Event-B, eg)

Modelling Behaviour

State transition models allow us to model object lifecycles and event-driven behaviour



Modelling Behaviour: adding constraints

We can use constraints that apply to transitions to show how a certain event triggers a certain transition from one state to another only when some condition is true.

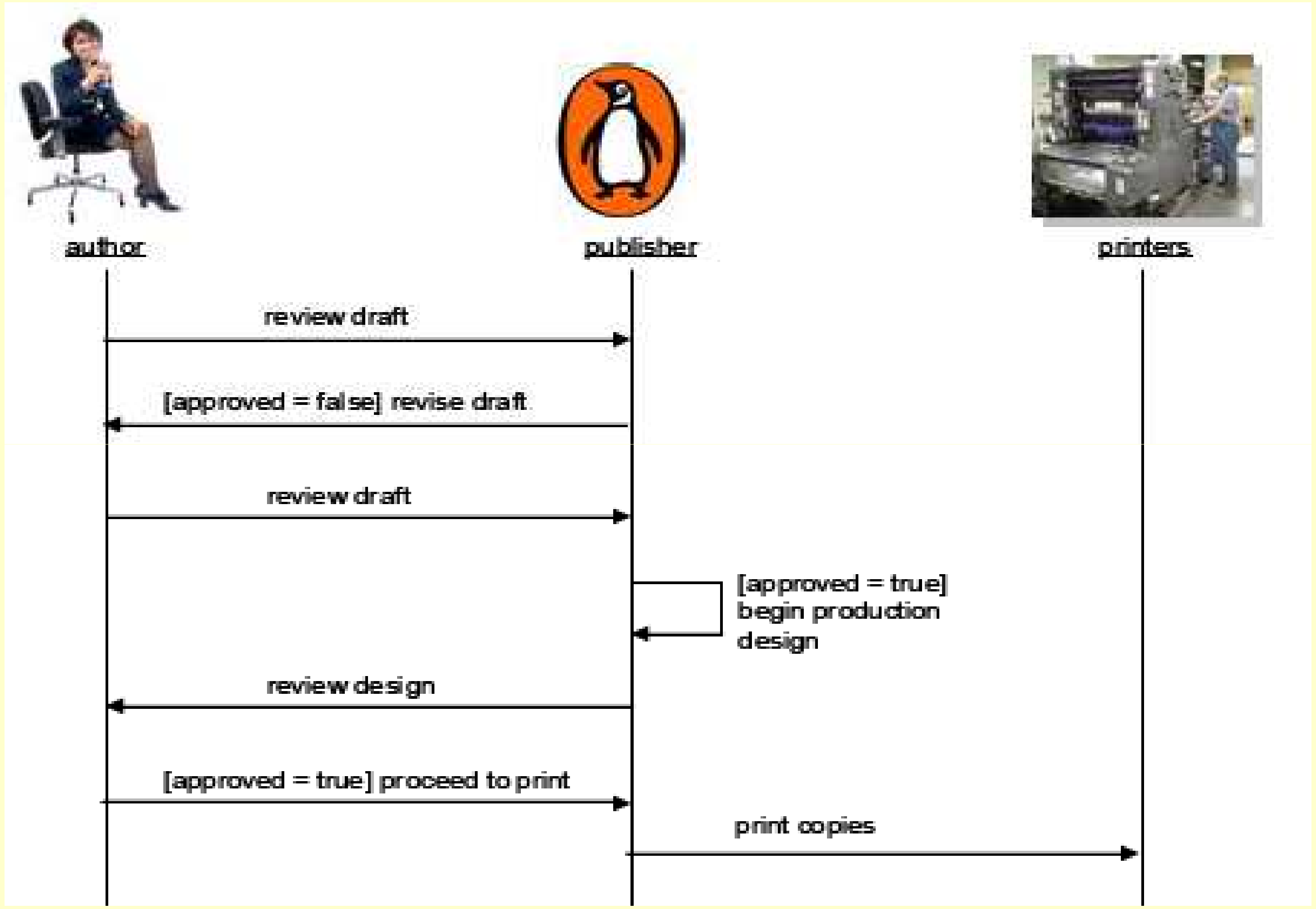
For example, when the publisher reviews the draft of a book in development, it could take the book into production design, but only if the publisher has approved the draft.

We call constraints on transitions guard conditions. In UML, guard conditions are written in square brackets after the event.

Optionally, we can also show how some action is executed as a result of a transition, eg: we might want to show that the designer should revise his design for a book if his last design was not approved when the publisher reviewed it.

Actions appear after that event and the guard - if there is one - for a transition.

Modelling interactions with sequence diagrams



Modelling interactions with sequence diagrams

Once we have identified the objects involved and their relationships, we must now decide which object is taking responsibility for what action in the execution of a process.

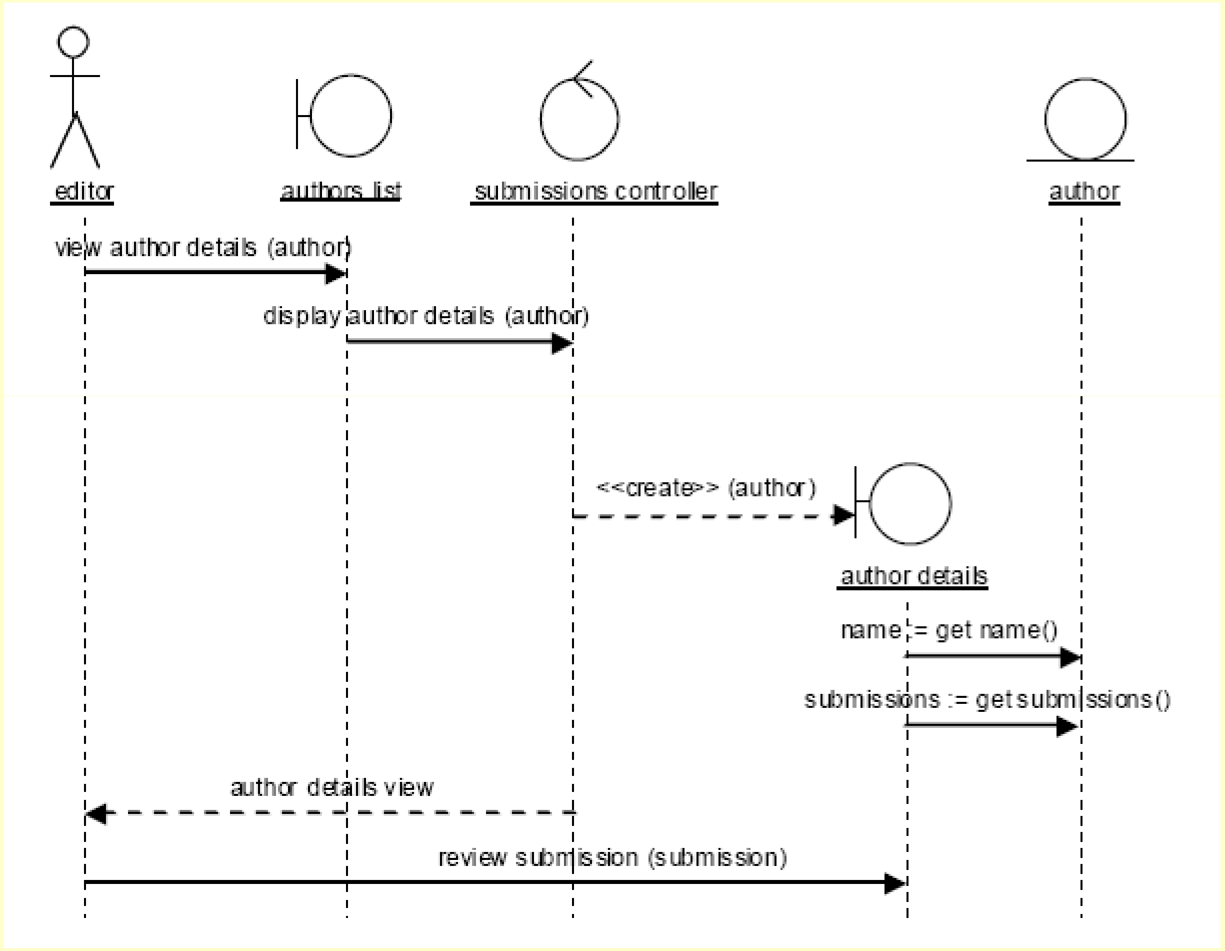
Sequence models describe how objects send messages to each other - through well- defined interfaces - asking them to do some useful piece of work. Each type of object has responsibility for providing a set of services, and the interface for each type public face through which other objects can request those services. In UML, we call those services operations.

A sequence model shows how, over time - and for a specific scenario (a specific instance of a process - or a pathway through that process): the objects involved use each others’ operations to get the job done.

Assigning these responsibilities is a key part in the object oriented thought design process, and we will see how these models can be used in a well-defined and rigorous object oriented development process.

Modelling interactions with sequence diagrams

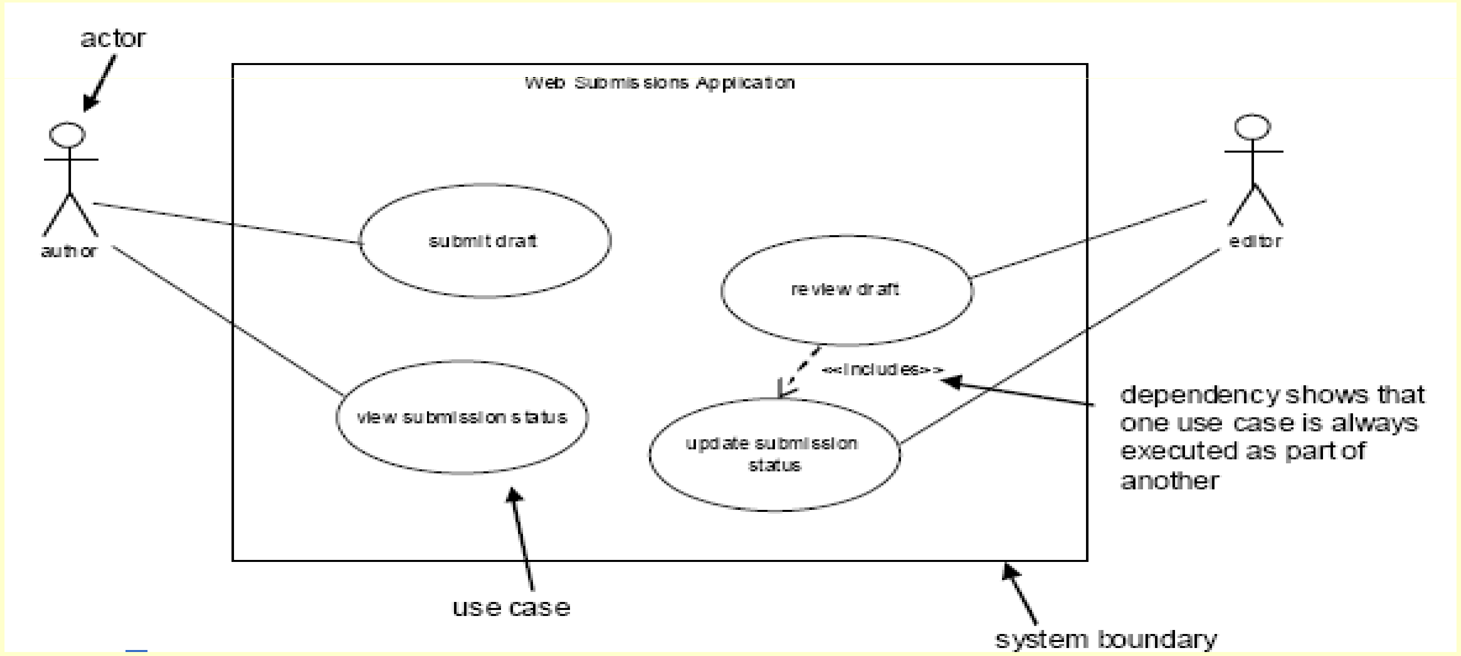
Objects, like an author, can be found in many different interaction diagrams:


editor
authors list
submissions controller
author
view author details (author)
display author details (author)
<<create>> (author)
author details
name := get name()
submissions := get submissions()
author details view
review submission (submission)

Use case diagrams: special type of sequence diagram?

Use case diagrams show the different classes of user and the goals they can achieve using the system

Some software development processes are said to be use case-driven, in so much as they are driven by requirements



Example Activity Diagram

An activity diagram models the flow of actions in some process or workflow.

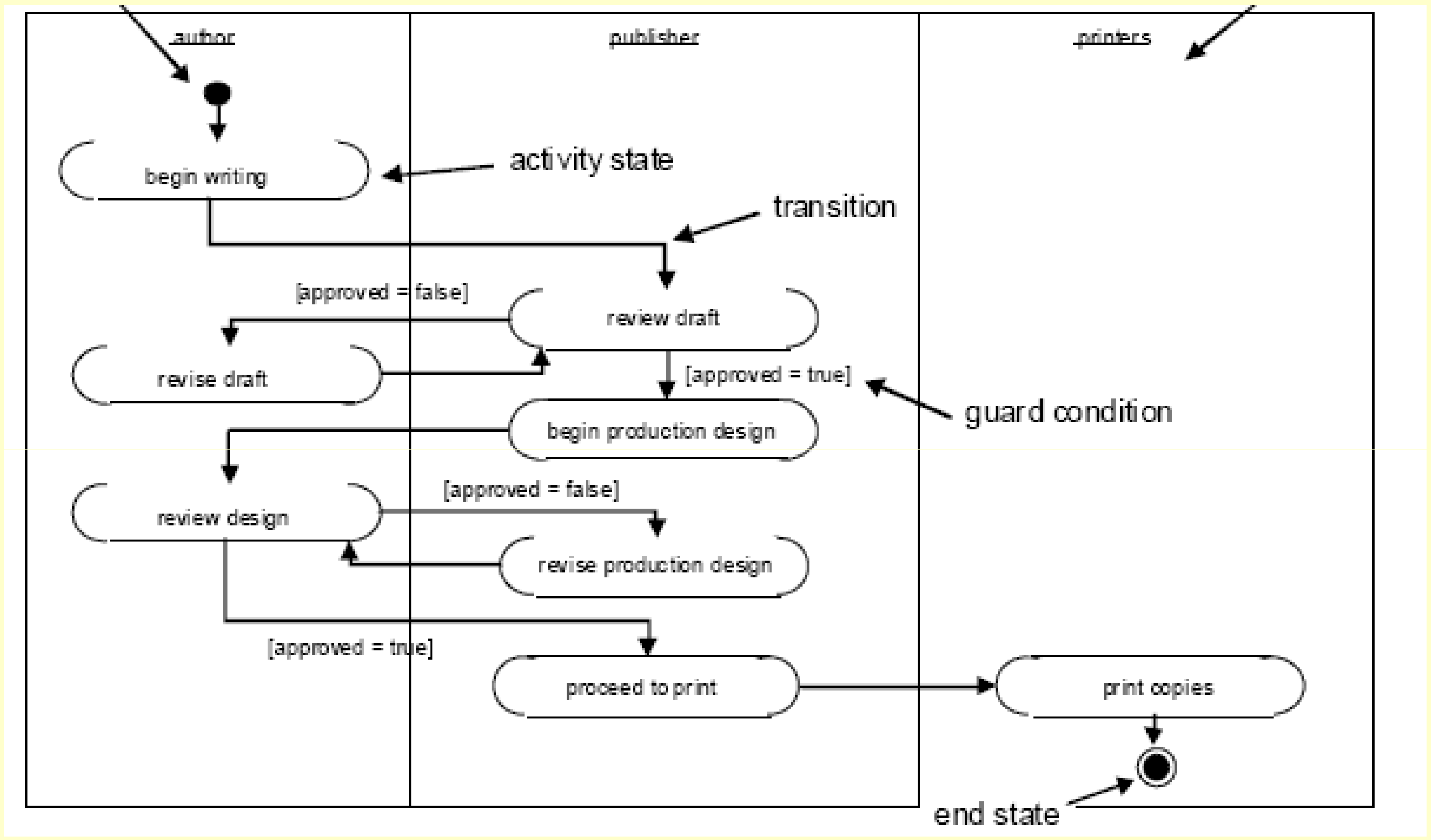
It could be a business process, or it could be the control flow of program code.

An activity diagram shows sequences of activity states - or actions -where when one action is complete the flow immediately moves on to the next action.

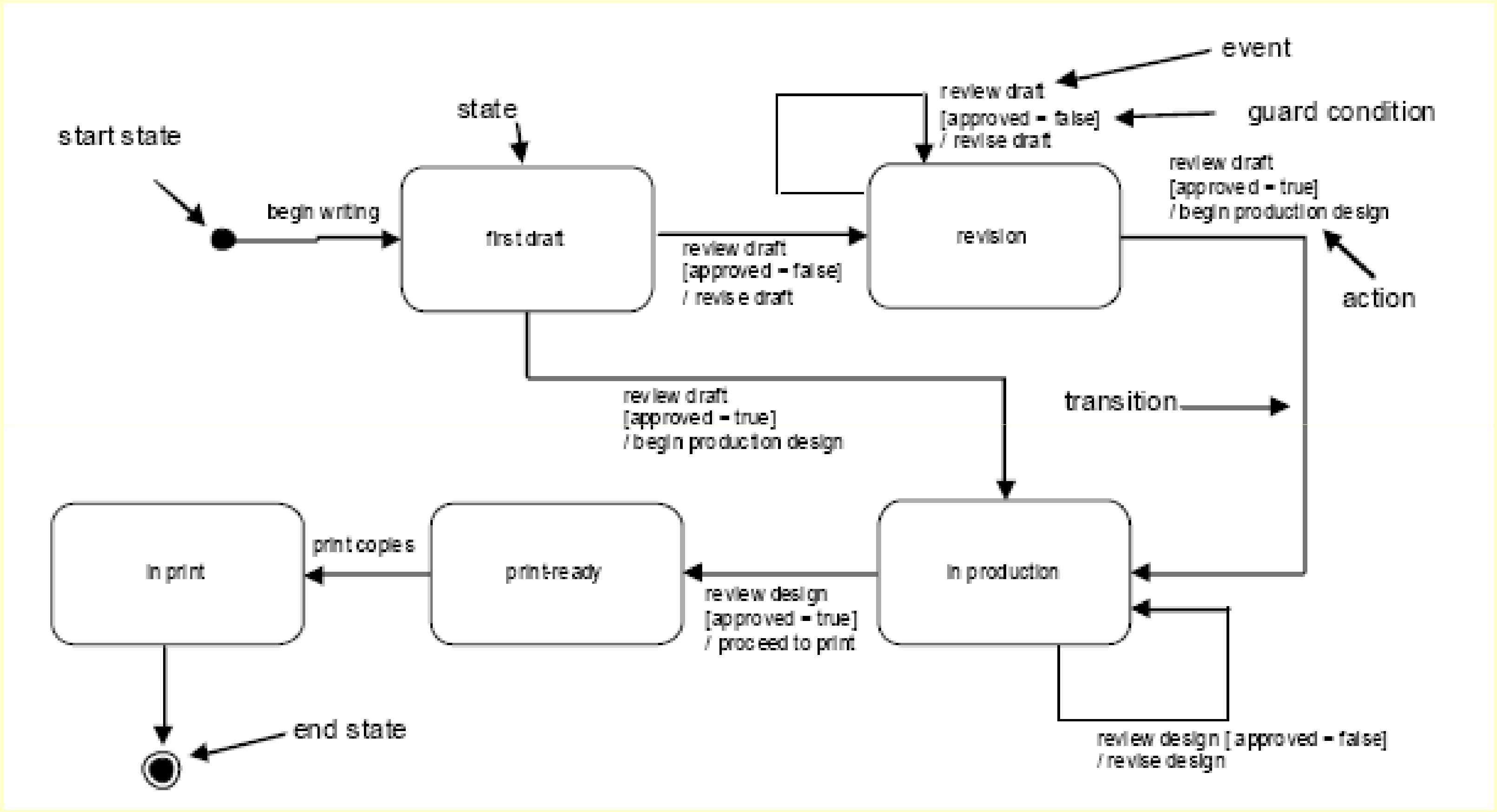
This automatic transitioning from one activity state to the next is what distinguishes activity diagrams from state transition diagrams.

Example Activity Diagram

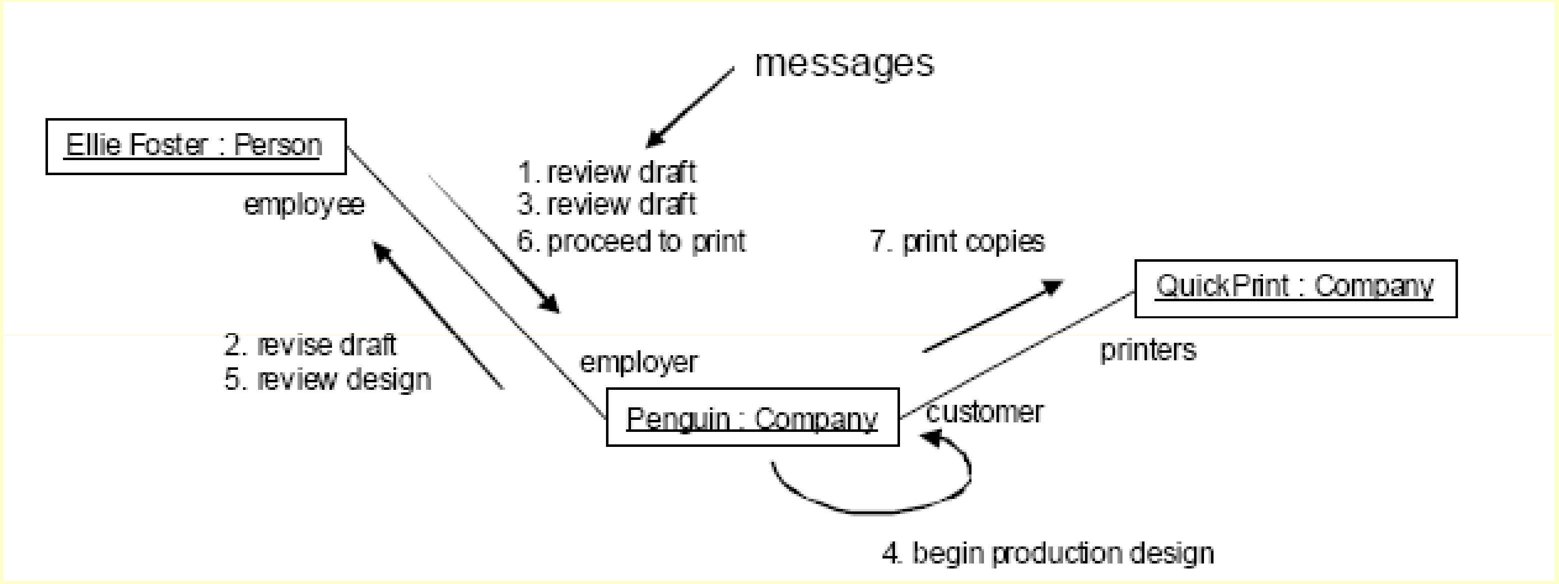
Start state



Why not use a state transition diagram?



… **Or a collaboration diagram?**

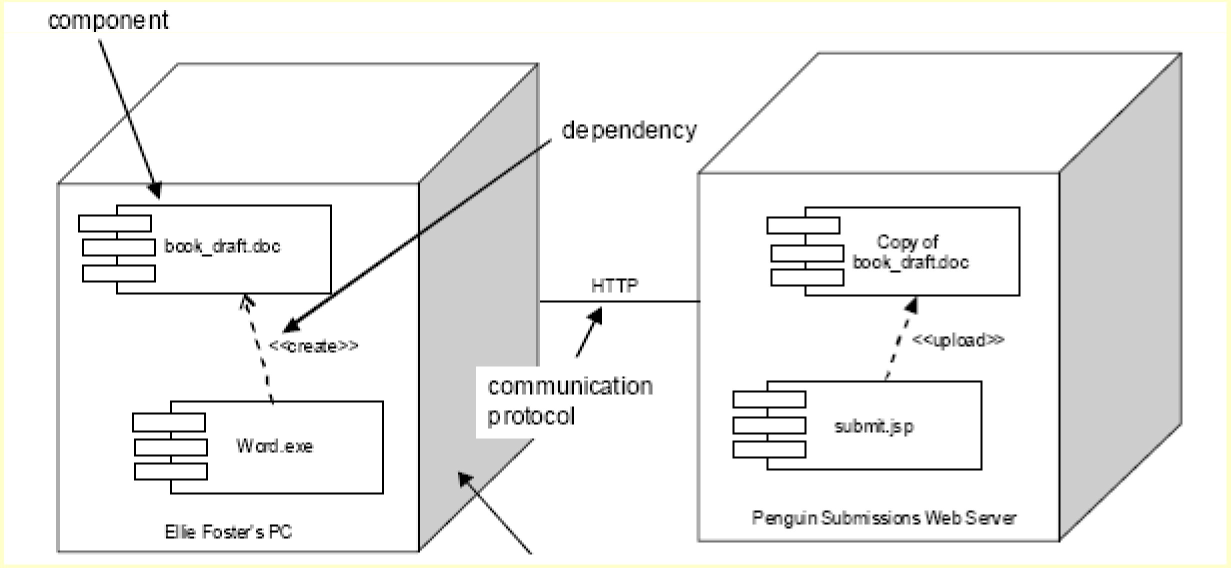


Or ... Etc …?

When modelling becomes questionable: component diagrams

UML has special notations for representing physical components, their interfaces, thedependencies between them, and their deployment architectures.

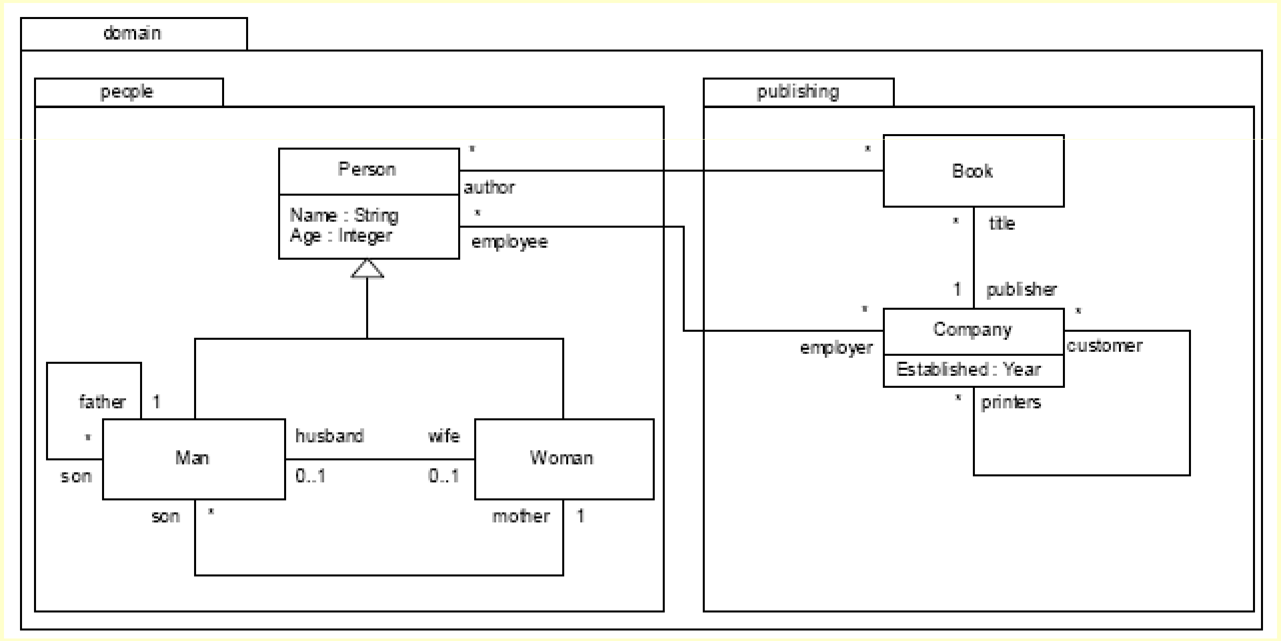
Its worth nothing,though, that at this low level of abstraction the benefits of modeling start to become questionable.



When modelling becomes questionable: packages

Packages & Model Management

Just as we can group files on our computers into folders to make them easier to find,we can break large models down into packages that group related model elements together.



Learning UML: different, complementary approaches

- **By memorising:**
  Diagram by Diagram

- **By analysing:**
  Case Studies

- **By doing:**
  OO Analysis, Requirements and Design

- **By having large scale problems:**
  Software Engineering Process – co-ordinating diagrams

- **By using tools:**
  Validation, Verification, Code Generation, Reverse Engineering

<u>**Does UML meet your needs?**</u>

What requirements do you have of your modeling language?

What sort of things should it be able to represent elegantly?

Repeat

1. Think about a problem/case study/project that you have done
2. What sort of things were you reasoning about?
3. What sort of notation did you use?

Until you think you have a complete understanding of these things

How do you do these in Event-B (if at all)?

Does UML meet your/our needs?

Extending the UML

Sometimes it is necessary to convey more information than vanilla UML is able to describe.

We have already seen one mechanism for adding extra information to our models - using constraints.

The UML standard provides two other mechanisms for extending the language: stereotypes and tagged values.

Additional Reading Material

- *UML3.0 and the future of modeling*, Cris Kobryn, 2004
- *Death By UML Fever*, Alex E Bell, 2004
- *UML Fever: Diagnosis and Recovery*, Alex E Bell, 2005
- *The UML as a Forml Modeling Notation*, France, Evans, Lano and Rump, 1998
- *On formalizing the UML object constraint language OCL*, Mark Richters and Martin Gogolla, 1998
- *Teaching UML is Teaching Software Engineering is Teaching Abstraction*, Gregor Engels, Jan Hendrik Hausmann, Marc Lohmann, Stefan Sauer, 2005

Useful Resources

My Little UML (Tools) Page: Michael W. Godfrey's pointers to useful tools

Violet UML Editor: standalone editor with minimal functionality (multi-platform, jre)

ArgoUML : Eclipse Plugin (or standalone app)
ArgoUML reference manual

Fujaba4Eclipse: From UML to Java and back again (Eclipse plugin)

TopCased:Open Source Toolkit for critical systems

TO DO: Email me links/details concerning the UML tools that you currently use (or have used)

SOME UML NOTATION

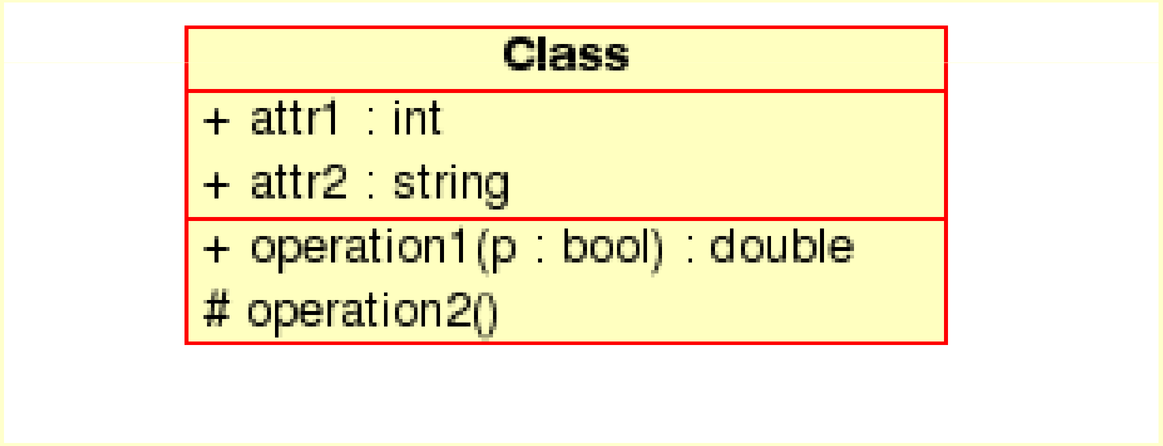
Class Diagrams

A Class defines the attributes and the methods of a set of objects.
All objects of this class (instances of this class) share the same behaviour, and have the same set of attributes (each object has its own set).
The term “Type” is sometimes used instead of Class, but it is important to mention that these two are not the same, and Type is a more general term.

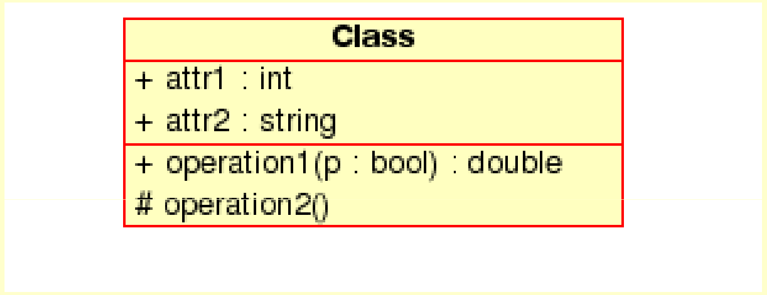
In UML, Classes are represented by rectangles, with the name of the class, and can also show the attributes and operations of the class in two other “compartments” inside the rectangle.

Class Diagrams

Graphical Notation, example:



Class Diagrams

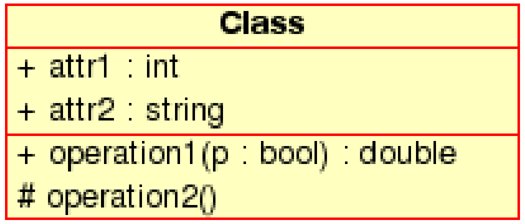


Attributes

In UML, Attributes are shown with at least their name, and can also show their type, initial value and other properties. Attributes can also be displayed with their visibility:

+	public attributes
#	protected attributes
-	private attributes

Class Diagrams



Operations

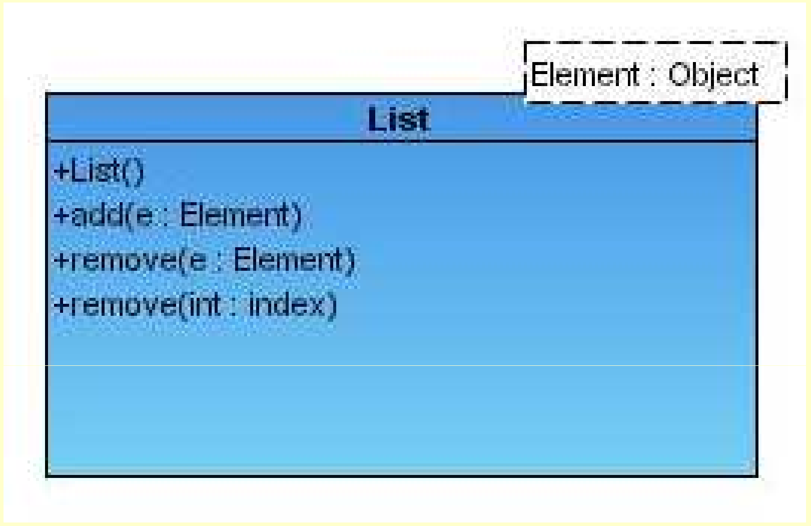
Operations (methods) are also displayed with at least their name, and can also show their parameters and return types. Operations can, just as Attributes, display their visibility:

+ *public* operations

# *protected* operations

- *private* operations

Class Diagrams



Templates

Classes can have templates, a value which is used for an unspecified class or type.

The template type is specified when a class is initiated (i.e. an object is created). Templates exist in C++ and were introduced in Java 1.5 where they are also called Generics.

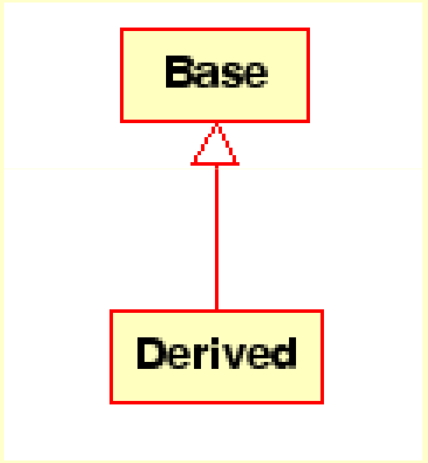
Class Relations

Generalisation

Inheritance is one of the fundamental concepts of Object Orientated programming, in which a class “gains” all of the attributes and operations of the class it inherits from, and can override/modify some of them, as well as add more attributes and operations of its own.

In UML, a *Generalisation* association between two classes puts them in a hierarchy representing the concept of inheritance of a derived class from a base class.

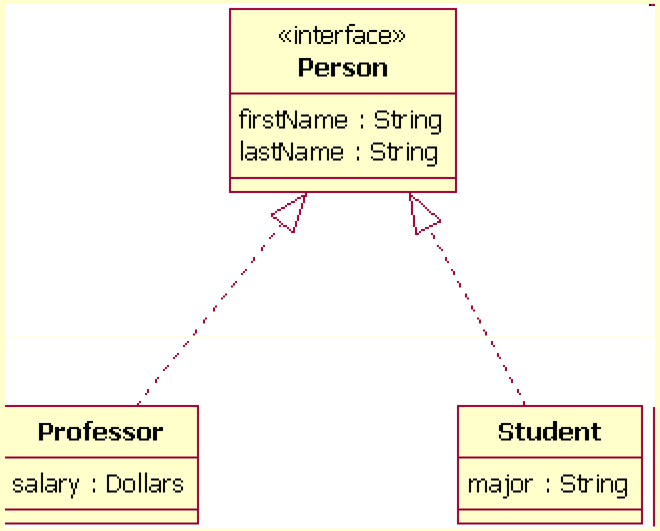
In UML, Generalisations are represented by a line connecting the two classes, with an arrow on the side of the base class.



Class Relations

Interface – implementing (a contract)

In the diagram both the Professor and Student classes implement the Person interface and do not inherit from it. We know this for two reasons:
1) The Person object is defined as an interface — it has the "«interface»" text in the object's name area, and we see that the Professor and Student objects are class objects because they are labeled according to the rules for drawing a class object (there is no additional classification text in their name area).
 2) We know inheritance is not being shown here, because the line with the arrow is dotted and not solid.



Class Relations

Associations

An association represents a relationship between classes, and gives the common semantics and structure for many types of “connections” between objects.

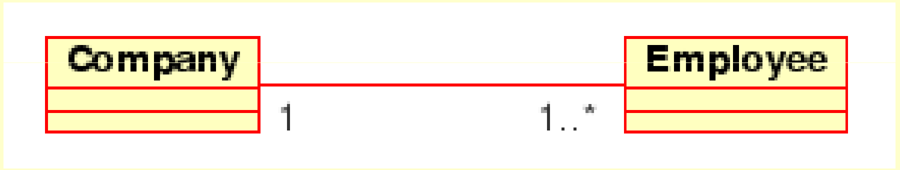
Associations are the mechanism that allows objects to communicate to each other. It describes the connection between different classes (the connection between the actual objects is called object connection, or *link*.)

Associations can have a role that specifies the purpose of the association and can be uni- or bidirectional (indicates if the two objects participating in the relationship can send messages to the other, of if only one of them knows about the other). Each end of the association also has a multiplicity value, which dictates how many objects on this side of the association can relate to one object on the other side.

Class Relations

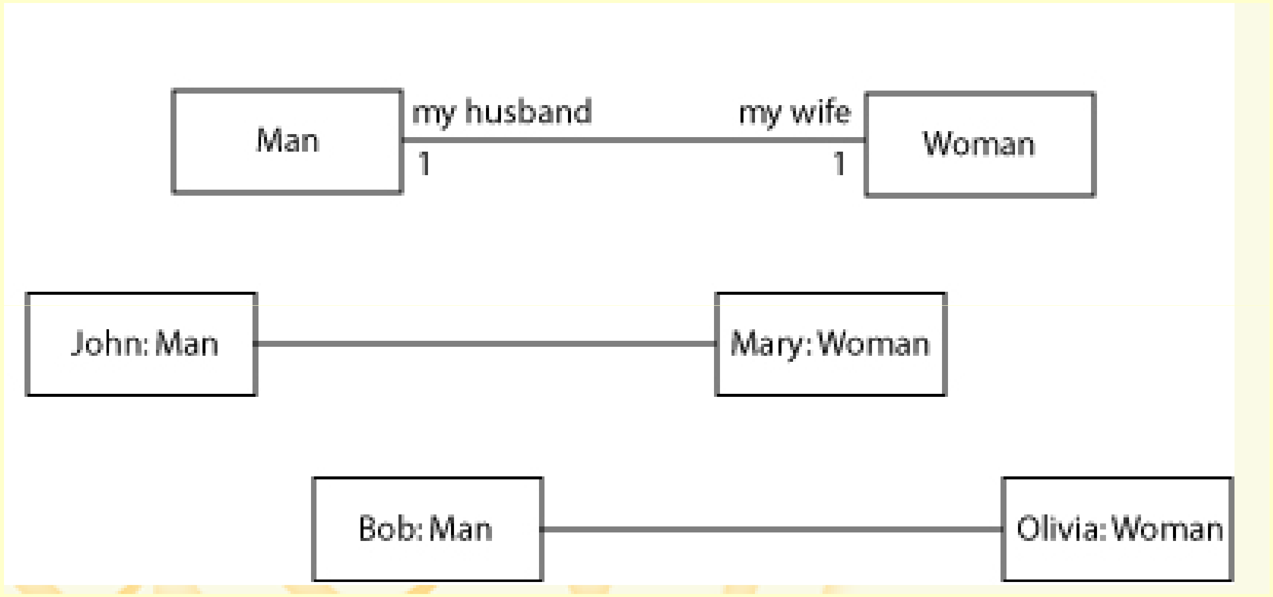
Associations

In UML, associations are represented as lines connecting the classes participating in the relationship, and may also show the role and the multiplicity of each of the participants. Multiplicity is displayed as a range [min..max] of non-negative values, with a star (*) on the maximum side representing infinite.



Class Relations

Example: monogamous marriage



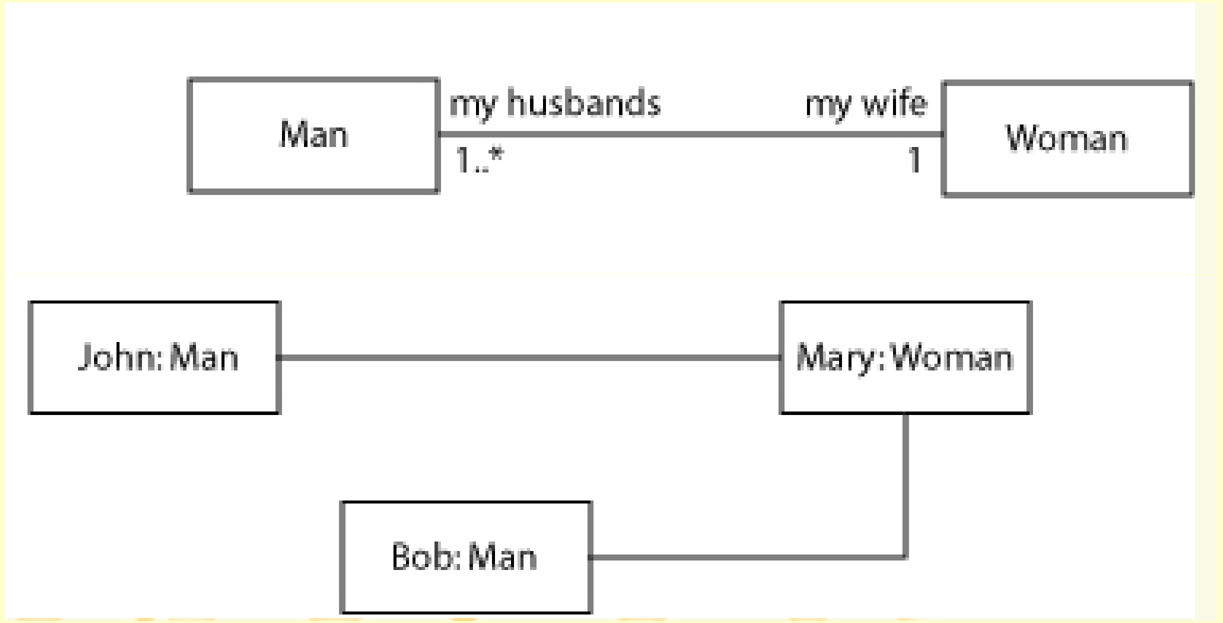
Class Relations abacus rules

Any two objects that are linked through an association are "married," just like John and Mary. Such a marriage has a number of characteristics, which we call the ABACUS rules. The characteristics that make up ABACUS (and compose the acronym) are:

•***A****wareness*: Both objects are aware of the fact that the relationship exists.
•***B****oolean existence*: If the partners agree to a divorce, the relationship (in UML called the link) is dissolved completely.
•***A****greement*: Both objects have to agree with the relationship; they need to say "I do."
•***C****ascaded deletion*: If one of the partners dies, the link is dissolved as well.
•***US****e of rolenames*: An object may refer to its partner using the role name provided with the association: "my husband" or "my wife."

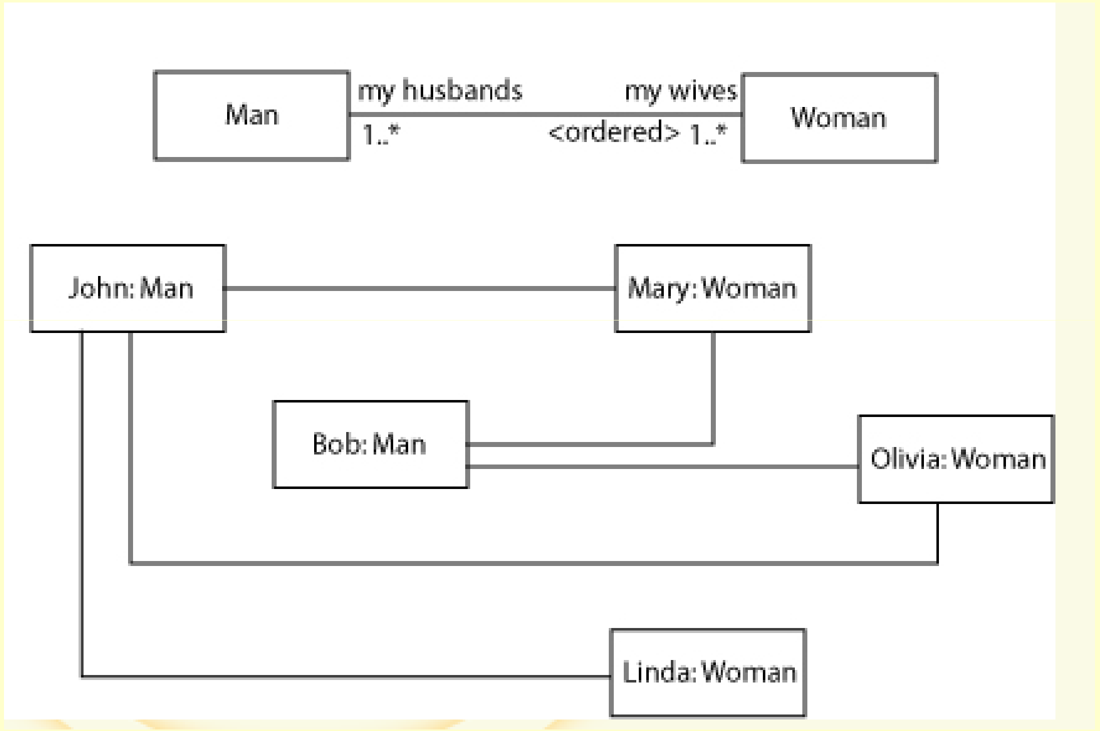
Class Relations

Example: polygamous marriage



Class Relations

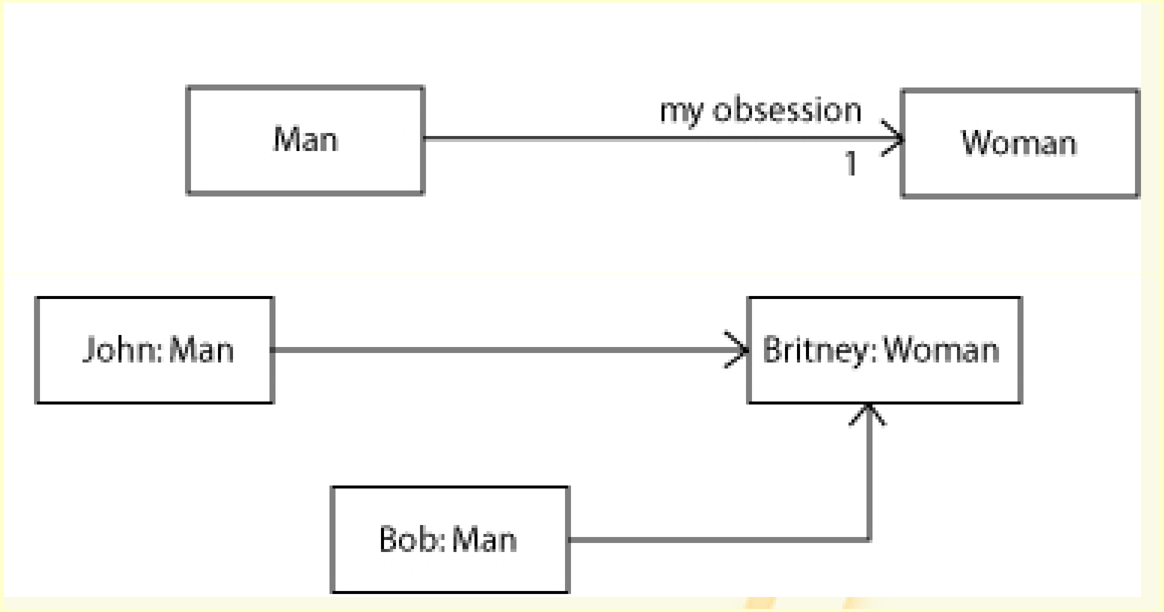
Example: group marriage



<ordered> is non-default collection

Class Relations

Example: one way association



<u>Class Relations</u>

Sets, Ordered Sets, Bags, and Sequences

The UML offers a choice in the type of collection used for an association end with a multiplicity greater than one.

You may choose one of the following four types:

Set: Every element may be present in the collection only once. This is the <u>default</u> collection type for association ends with a multiplicity larger than one.

Ordered Set: A set in which the elements are ordered. There is an index number for each element. Note that the elements are not sorted, that is, an element with a lower index number is not in any way larger or smaller than one with a higher index. An association end with this type is indicated in the diagram by adding the marking **<ordered>.**

Class Relations

Sets, Ordered Sets, Bags, and Sequences cont …

Bag: An element may be in the collection more than once. In the marriage example this means that, for instance, Mary may be married to John twice. Note that if one end of an association has this type, than the other end must be either a bag or a sequence. The type is indicated in the diagram by **<bag>.**

Sequence: A bag in which the elements are ordered. It is indicated in the diagram by **<sequence>** or **<seq>.** If one end of an association has this type, than the other end must be either a bag or a sequence.

Class Relations

Sets, Ordered Sets, Bags, and Sequences cont …

What are the effects of choosing a different collection type for an association end?

Let us first stress that the ABACUS rules must hold, whatever type you choose.

The type of the association end is relevant only when an element is added to the relationship, and this is very important for the way associations are implemented.

In the diagram for the group marriage (Figure 3) the end is marked "my wives" as an ordered set.

This means that the implementer of the addWife operation in the class Man that adds a woman to the ordered set must decide how to order the set of wives.

Also, the implementer of addWife must make sure that the new wife is not already present in the list of wives. The OrderedSet type does not allow this.

Class Relations

Implementing Associations

The most important thing about implementing associations is to make sure that the two-way connection is always correct.

It should always abide to the ABACUS rules. This means that whenever there is a change in the links, the implementation needs to take care of readjusting the references at both sides of the association.

Let's look at the simplest case: the one-to-one association.

Class Relations

Implementing 1-1 Associations

One-to-one associations can be implemented using two fields (pointers) in the two associated classes.

In class Man the type of the field should be Woman, and vice versa.

Because the association is symmetrical, the same implementation can be used in both classes.

Next to the field we need get and set operations: getMyWife and setMyWife.

The body of the getMyWife operation is simple; it just returns the value of the field

Class Relations

Implementing 1-1 Associations (in Java)

```
public Woman getMyWife() {
     return f_myWife;
}
```

```
public void setMyWife(Woman element) {
     if ( this.f_myWife != element ) {    // prevent infinite loop
          if ( this.f_myWife != null ) {    // there is a previous wife!
               // remove the link with the previous wife
               this.f_myWife.z_internalRemoveFromMyHusband( (Man)this );
          }
          this.f_myWife = element;       // set the field to the new value
          if ( element != null ) {
               // make the new wife aware of the link
               element.setMyHusband( (Man)this );
          }
     }
}
```

Class Relations

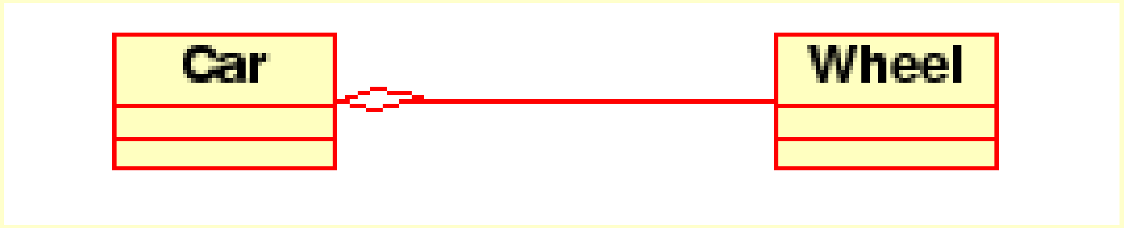
Aggregation

Aggregations are a special type of associations in which the two participating classes don't have an equal status, but make a “whole-part” relationship.

An Aggregation describes how the class that takes the role of the whole, is composed (has) of other classes, which take the role of the parts.

For Aggregations, the class acting as the whole always has a multiplicity of one.

In UML, Aggregations are represented by an association that shows a rhombus on the side of the whole.



Thus, aggregations can be structured in a tree but not in a graph (with circuits)

Class Relations

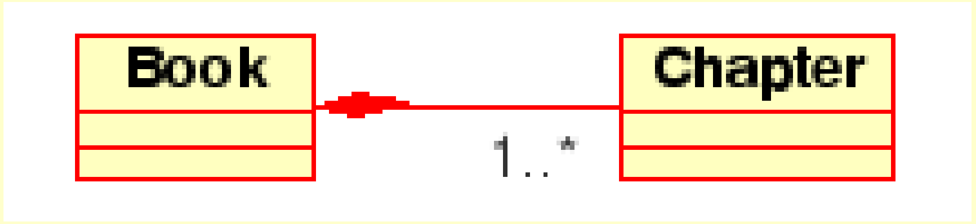
Composition

Compositions **are associations** that represent *very strong* aggregations.

This means, Compositions form whole-part relationships as well, but the relationship is so strong that the parts cannot exist on its own.

They exist only inside the whole, and if the whole is destroyed the parts die too.

In UML, Compositions are represented by a solid rhombus on the side of the whole.



Other Class Diagram Items (for later)

Class diagrams can contain several other items besides classes:

Interfaces

Interfaces are abstract classes which means instances can not be directly created of them. They can contain operations but no attributes. Classes can inherit from interfaces (through a realisation association) and instances can then be made of these classes.

Datatypes

Datatypes are primitives which are typically built into a programming language. Common examples include integers and booleans. They can not have relationships to classes but classes can have relationships to them.

Enums

Enums are a simple list of values. A typical example is an enum for days of the week. The options of an enum are called Enum Literals. Like datatypes they can not have relationships to classes but classes can have relationships to them.

Packages

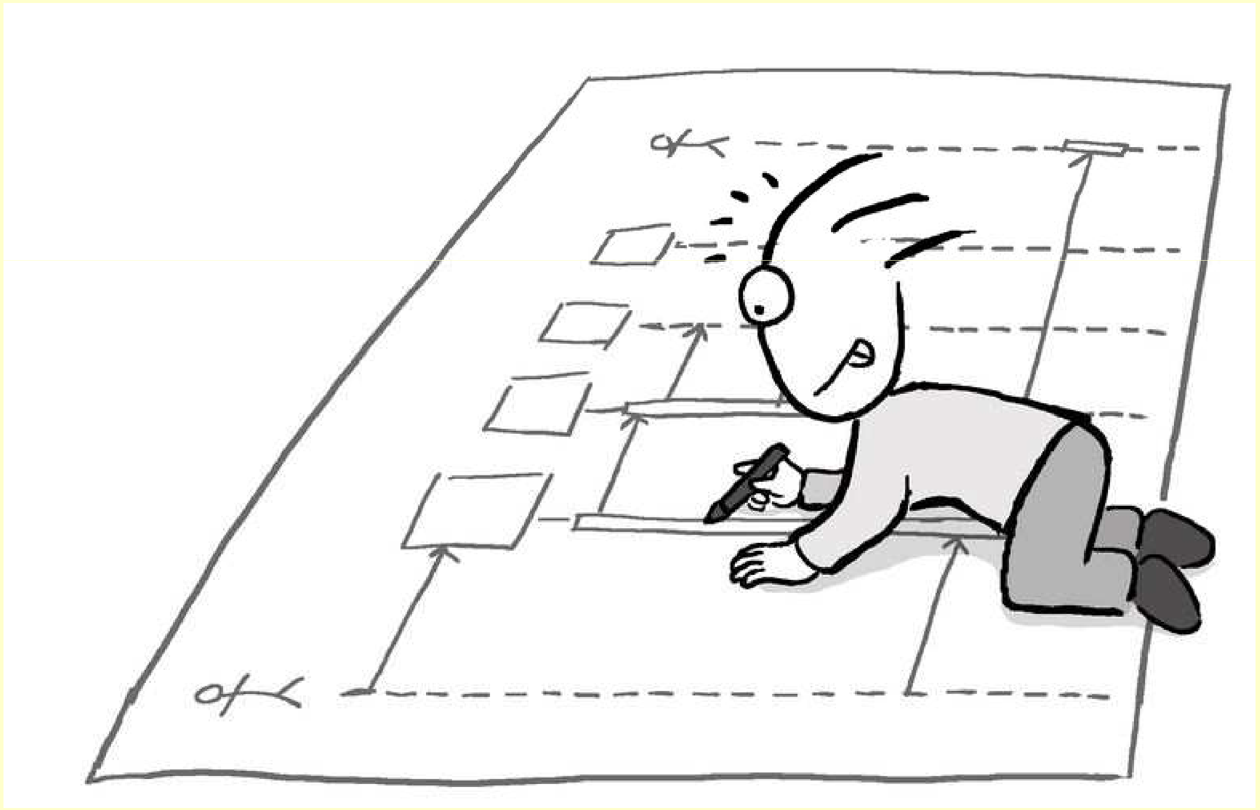
Packages represent a namespace in a programming language. In a diagram they are used to represent parts of a system which contain more than one class, maybe hundreds of classes.

UML Class Diagram Modelling Problem

Identify the key classes in a lift control system and model their relationships in a class diagram

STUCK? HAVE YOU TRIED SHUTTING OFF THE POWER, WAITING TWO MINUTES, AND REBOOTING?
Copyright 2008 John Crowther

Sequence Diagrams



Sequence Diagrams

Sequence Diagrams show the message exchange (i.e. method call) between several Objects in a specific time-delimited situation.

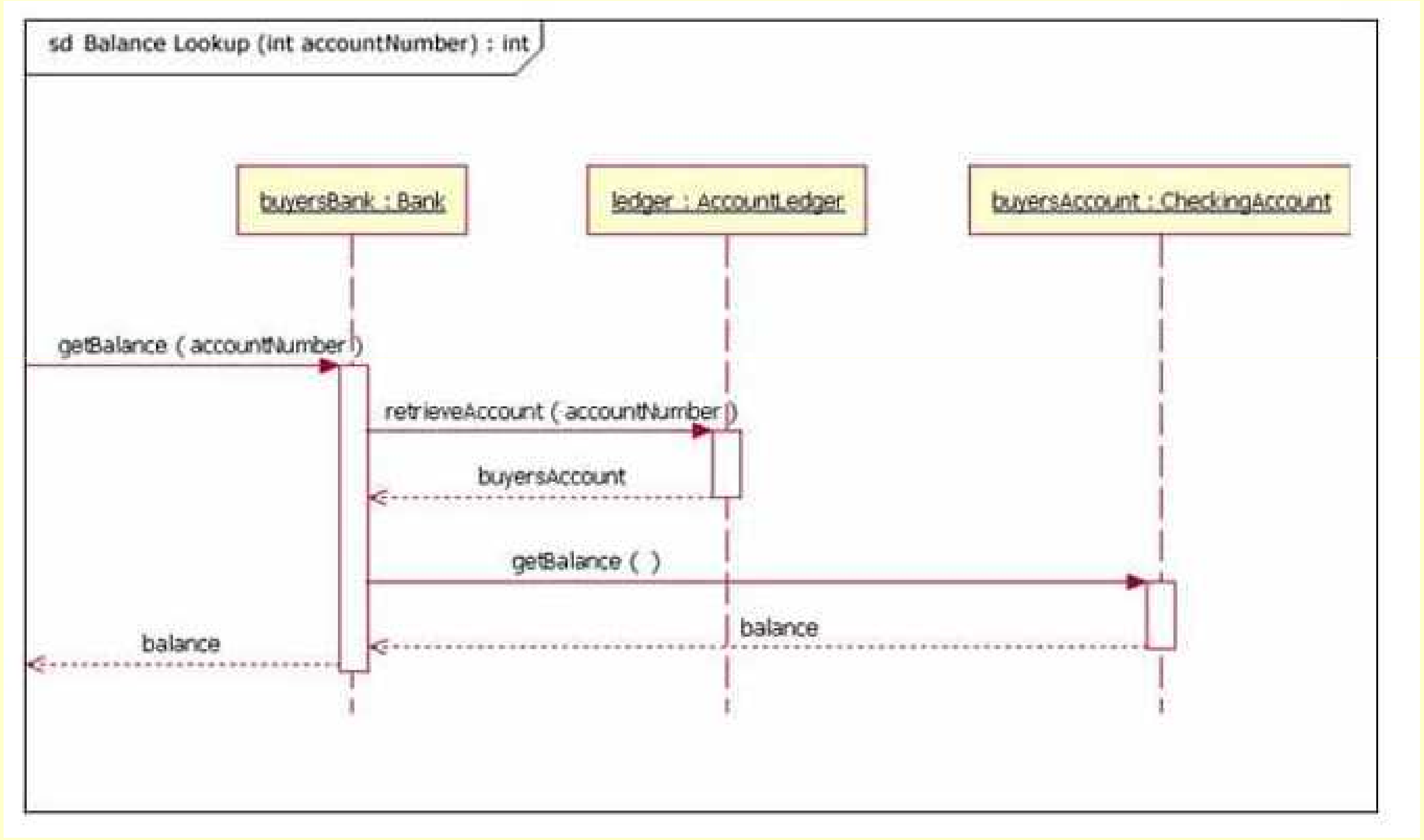
Objects are instances of classes.

Sequence Diagrams put special emphasis in the order and the times in which the messages to the objects are sent.

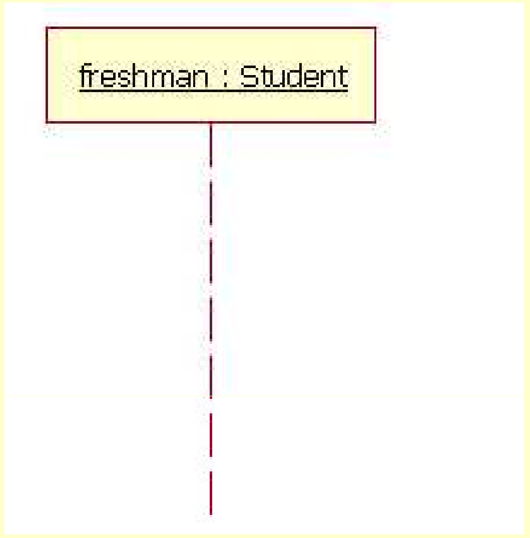
In Sequence Diagrams objects are represented through vertical dashed lines, with the name of the Object on the top.

The time axis is also vertical, increasing downwards, so that messages are sent from one Object to another in the form of arrows with the operation name and optionally the actual parameter values.

Sequence Diagrams: example


sd Balance Lookup (int accountNumber) : int
buyersBank : Bank
ledger : AccountLedger
buyersAccount : CheckingAccount
getBalance (accountNumber)
retrieveAccount (accountNumber)
buyersAccount
getBalance ( )
balance
balance

Sequence Diagrams: the basic elements



An example of the Student class used in a lifeline whose instance name is freshman

Lifelines

When drawing a sequence diagram, lifeline notation elements are placed across the top of the diagram.

Lifelines represent either roles or object instances that participate in the sequence being modeled.

Lifelines are drawn as a box with a dashed line descending from the center of the bottom edge

The lifeline's name is placed inside the box.

Sequence Diagrams: the basic elements

Messages

The first message of a sequence diagram always starts at the top and is typically located on the left side of the diagram for readability.

Subsequent messages are then added to the diagram slightly lower then the previous message.

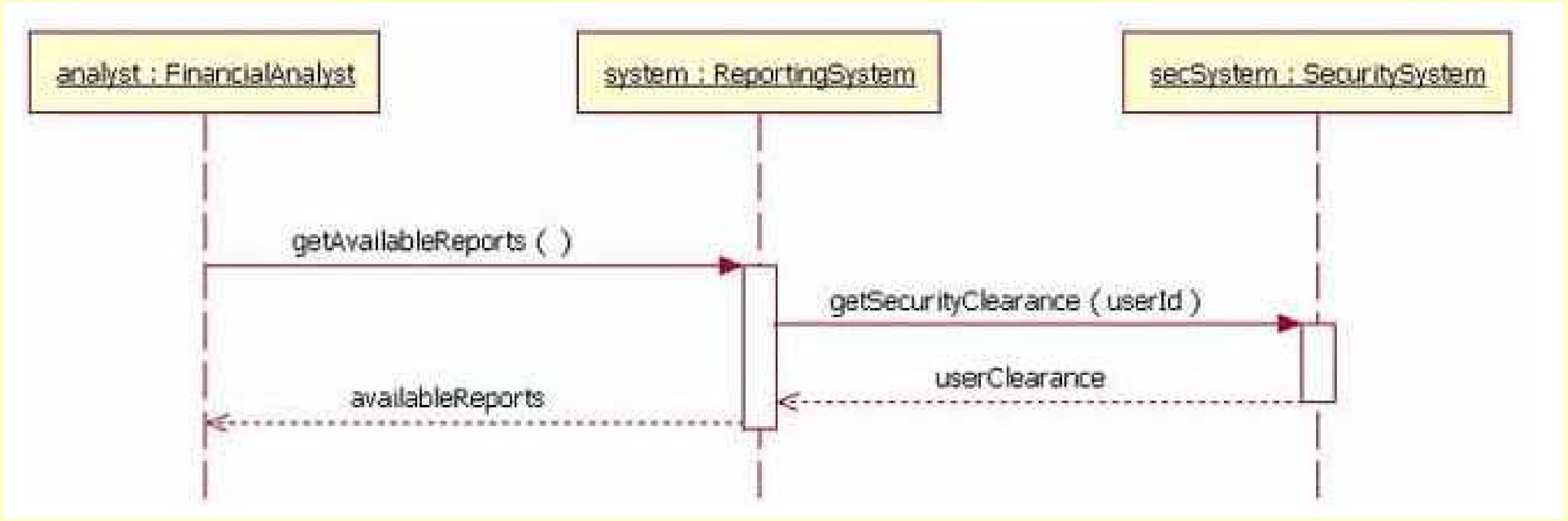
To show an object (i.e., lifeline) sending a message to another object, you draw a line to the receiving object with a solid arrowhead (if a synchronous call operation) or with a stick arrowhead (if an asynchronous signal).

The message/method name is placed above the arrowed line. The message that is being sent to the receiving object represents an operation/method that the receiving object's class implements.

Sequence Diagrams: the basic elements

Messages

In the example, the **analyst** object makes a call to the **system** object which is an instance of the **ReportingSystem** class. The **analyst** object is calling the system object's **getAvailableReports** method. The system object then calls the **getSecurityClearance** method with the argument of **userId** on the **secSystem** object, which is of the class type **SecuritySystem**.


analyst : FinancialAnalyst
system : ReportingSystem
secSystem : SecuritySystem
getAvailableReports ( )
getSecurityClearance (userId)
userClearance
availableReports

Sequence Diagrams: the basic elements

Return Messages

Besides just showing message calls on the sequence diagram, the diagram includes return messages.

These return messages are optional; a return message is drawn as a dotted line with an open arrowhead back to the originating lifeline, and above this dotted line you place the return value from the operation.

The **secSystem** object returns **userClearance** to the system object when the **getSecurityClearance** method is called. The system object returns **availableReports** when the **getAvailableReports** method is called.

Again, the return messages are an optional part of a sequence diagram.

The use of return messages depends on the level of detail/abstraction that is being modeled.

Sequence Diagrams: the basic elements

Self Messaging:

When modeling a sequence diagram, there will be times that an object will need to send a message to itself.

When does an object call itself?

A purist would argue that an object should never send a message to itself.

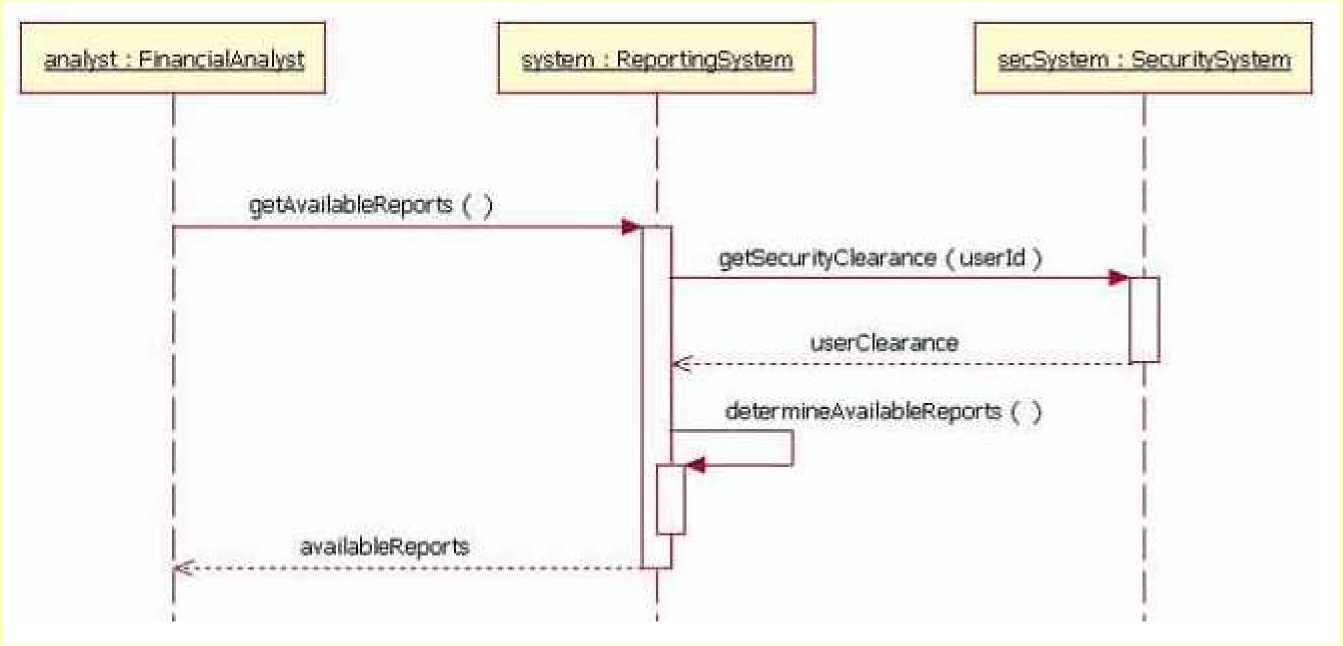
However, modeling an object sending a message to itself can be useful in some cases.

To draw an object calling itself, you draw a message as you would normally, but instead of connecting it to another object, you connect the message back to the object itself.

Sequence Diagrams: the basic elements

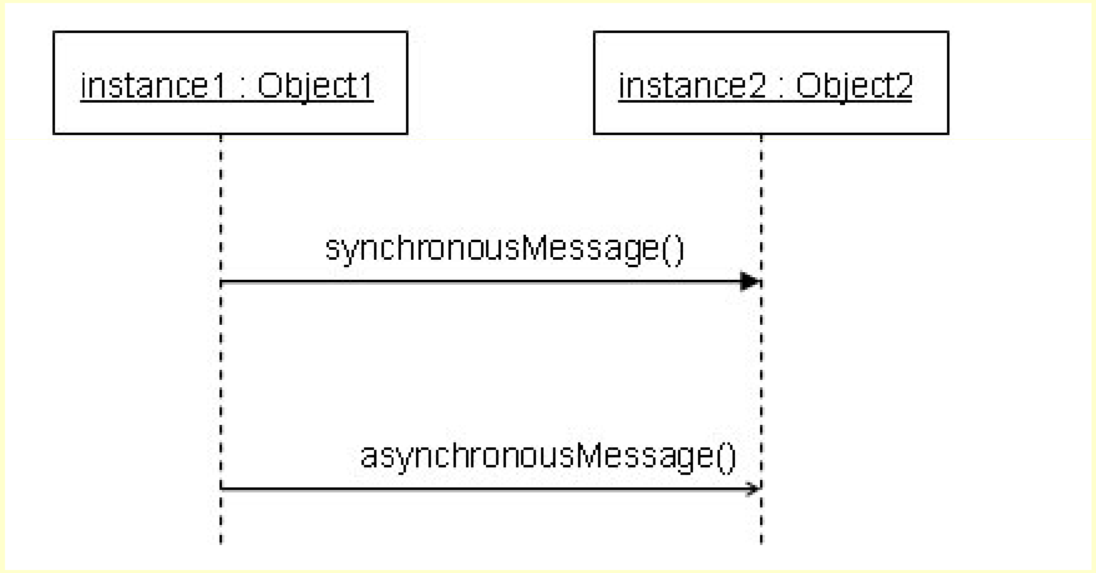
Self Messaging:

The Figure shows the system object calling its **determineAvailableReports** method. By showing the system sending itself the message "**determineAvailableReports**," the model draws attention to the fact that this processing takes place in the system object.


analyst : FinancialAnalyst
system : ReportingSystem
secSystem : SecuritySystem
getAvailableReports ( )
getSecurityClearance (userId)
userClearance
determineAvailableReports ( )
availableReports

Sequence Diagrams: the basic elements

Messages: synchronous or assynchronous



Sequence Diagrams: the basic elements

Messages: Guards

When modeling object interactions, there will be times when a condition must be met for a message to be sent to the object.

Guards are used throughout UML diagrams to control flow.

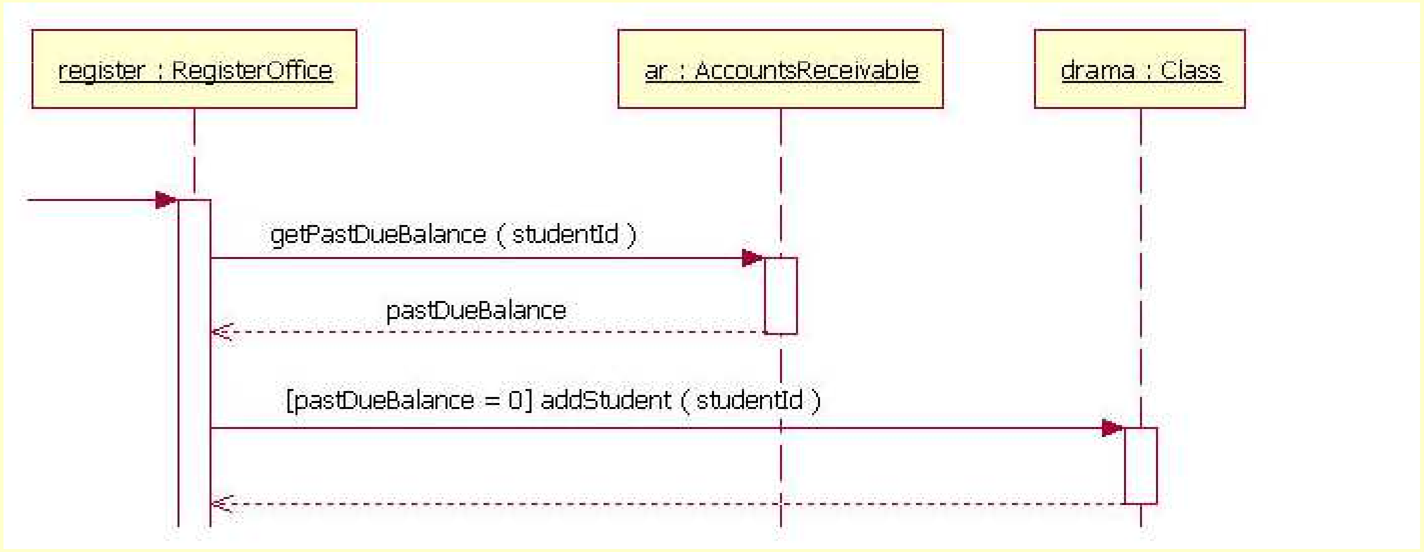
Note: In UML 1.x, a guard could only be assigned to a single message.

To draw a guard on a sequence diagram in UML 1.x, you placed the guard element above the message line being guarded and in front of the message name.

Sequence Diagrams: the basic elements

Messages: Guards

The figure shows a fragment of a sequence diagram with a guard on the message **addStudent** method.


register : RegisterOffice
ar : AccountsReceivable
drama : Class
getPastDueBalance (studentId)
pastDueBalance
[pastDueBalance = 0] addStudent (studentId)

Sequence Diagrams: the basic elements

Alternatives

Alternatives are used to designate a mutually exclusive choice between two or more message sequences.

Alternatives allow the modeling of the classic "if then else" logic (e.g., **if** I buy three items, **then** I get 20% off my purchase; **else** I get 10% off my purchase).

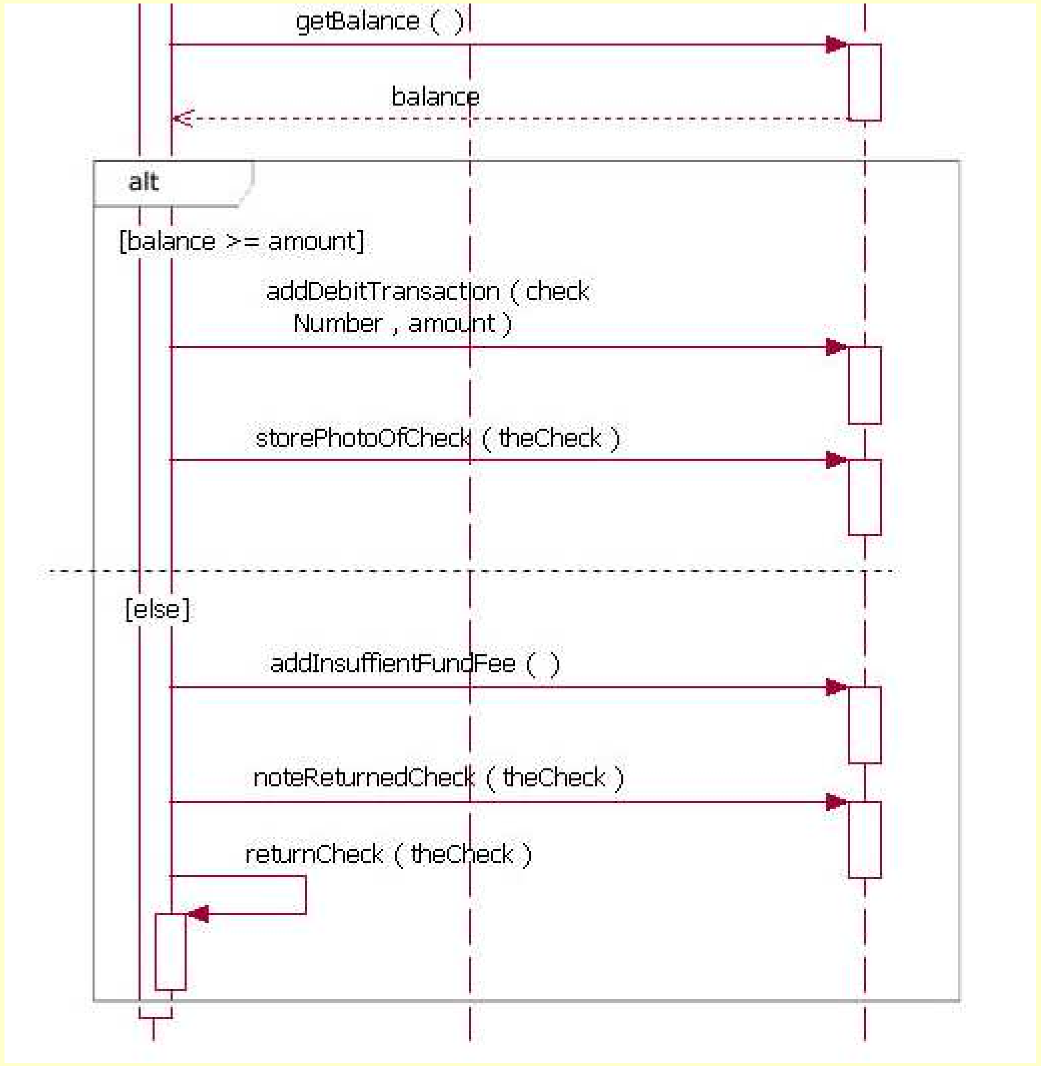
An alternative combination fragment element is drawn using a frame. The word "alt" is placed inside the frame's namebox. The larger rectangle is then divided into what UML 2 calls operands.

Operands are separated by a dashed line. Each operand is given a guard to test against, and this guard is placed towards the top left section of the operand on top of a lifeline.

If an operand's guard equates to "true," then that operand is the operand to follow.

Sequence Diagrams: the basic elements

Alternatives: example


getBalance ( )
balance
alt
[balance >= amount]
addDebitTransaction (check Number , amount)
storePhotoOfCheck (theCheck)
[else]
addInsuffientFundFee ( )
noteReturnedCheck (theCheck)
returnCheck (theCheck)

Sequence Diagrams: the basic elements

Options

The option combination fragment is used to model a sequence that, given a certain condition, will occur; otherwise, the sequence does not occur.

An option is used to model a simple "if then" statement (i.e., if there are fewer than five donuts on the shelf, then make two dozen more donuts).

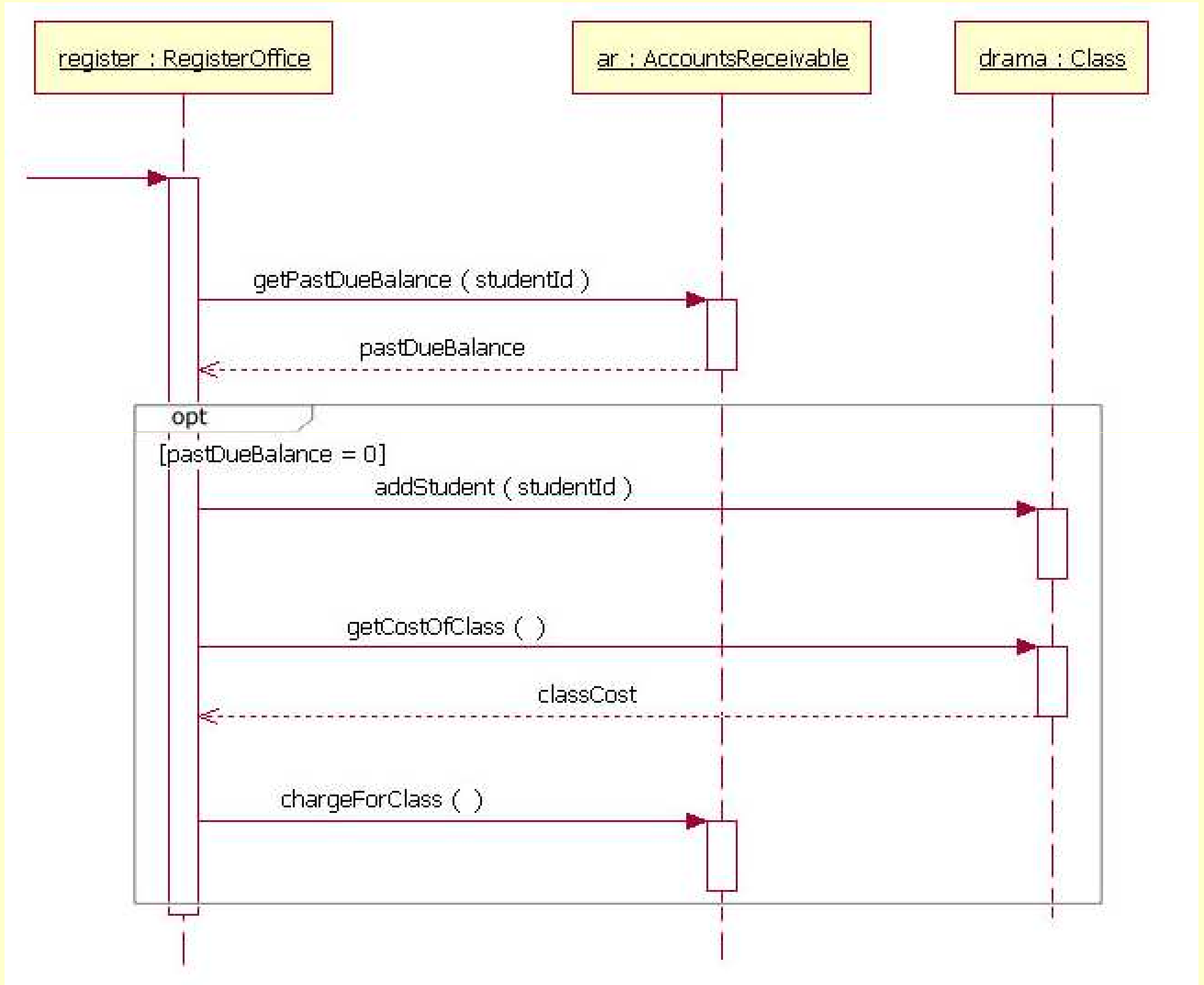
The option combination fragment notation is similar to the alternation combination fragment, except that it only has one operand and there never can be an "else" guard (it just does not make sense here).

To draw an option combination you draw a frame.

The text "opt" is placed inside the frame's namebox, and in the frame's content area the option's guard is placed towards the top left corner on top of a lifeline. Then the option's sequence of messages is placed in the remainder of the frame's content area.

Sequence Diagrams: the basic elements

Options: example



Sequence Diagrams: the basic elements

<u>Loops (special iterative options)</u>

Occasionally you will need to model a repetitive sequence.

In UML 2, modeling a repeating sequence has been improved with the addition of the loop combination fragment.

The loop combination fragment is very similar in appearance to the option combination fragment.

You draw a frame, and in the frame's namebox the text "loop" is placed. Inside the frame's content area the loop's guard is placed towards the top left corner, on top of a lifeline.

Then the loop's sequence of messages is placed in the remainder of the frame's content area.

In a loop, a guard can have two special conditions tested against in addition to the standard Boolean test.

Sequence Diagrams: advanced concepts

Referencing another sequence diagram

When doing sequence diagrams, developers love to reuse existing sequence diagrams in their diagram's sequences.

Starting in UML 2, the "Interaction Occurrence" element was introduced.

The addition of interaction occurrences is arguably the most important innovation in UML 2 interactions modeling.

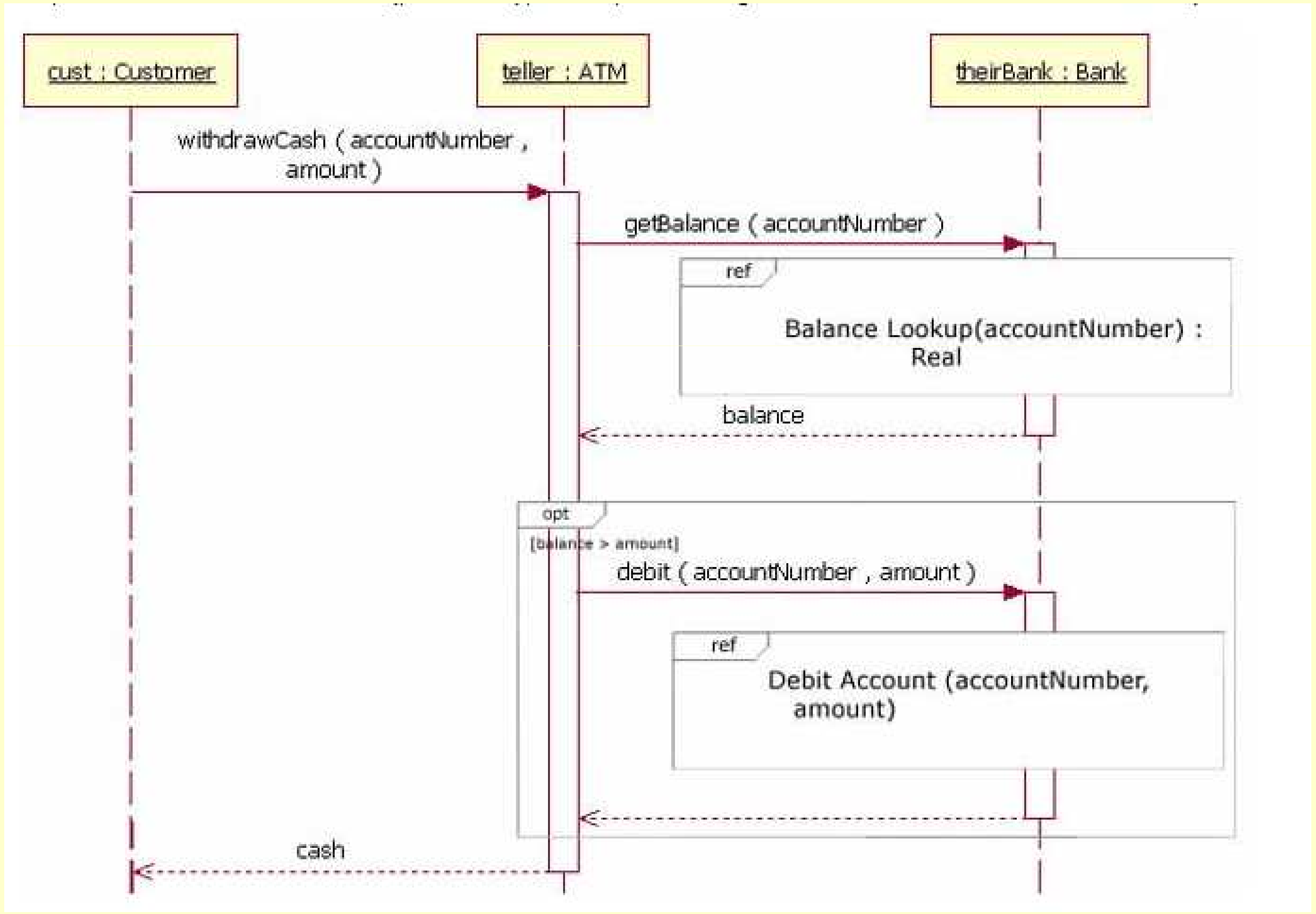
Interaction occurrences add the ability to compose primitive sequence diagrams into complex sequence diagrams.

With these you can combine (reuse) the simpler sequences to produce more complex sequences.

This means that you can abstract out a complete, and possibly complex, sequence as a single conceptual unit.

Sequence Diagrams: advanced concepts

Referencing another sequence diagram , example:


cust : Customer
teller : ATM
theirBank : Bank
withdrawCash (accountNumber , amount)
getBalance (accountNumber)
ref
Balance Lookup(accountNumber) : Real
balance
opt
[balance > amount]
debit (accountNumber , amount)
ref
Debit Account (accountNumber, amount)
cash

Sequence Diagrams: advanced concepts

Parallel

Today's modern computer systems are advancing in complexity and at times perform concurrent tasks.

When the processing time required to complete portions of a complex task is longer than desired, some systems handle parts of the processing in parallel.

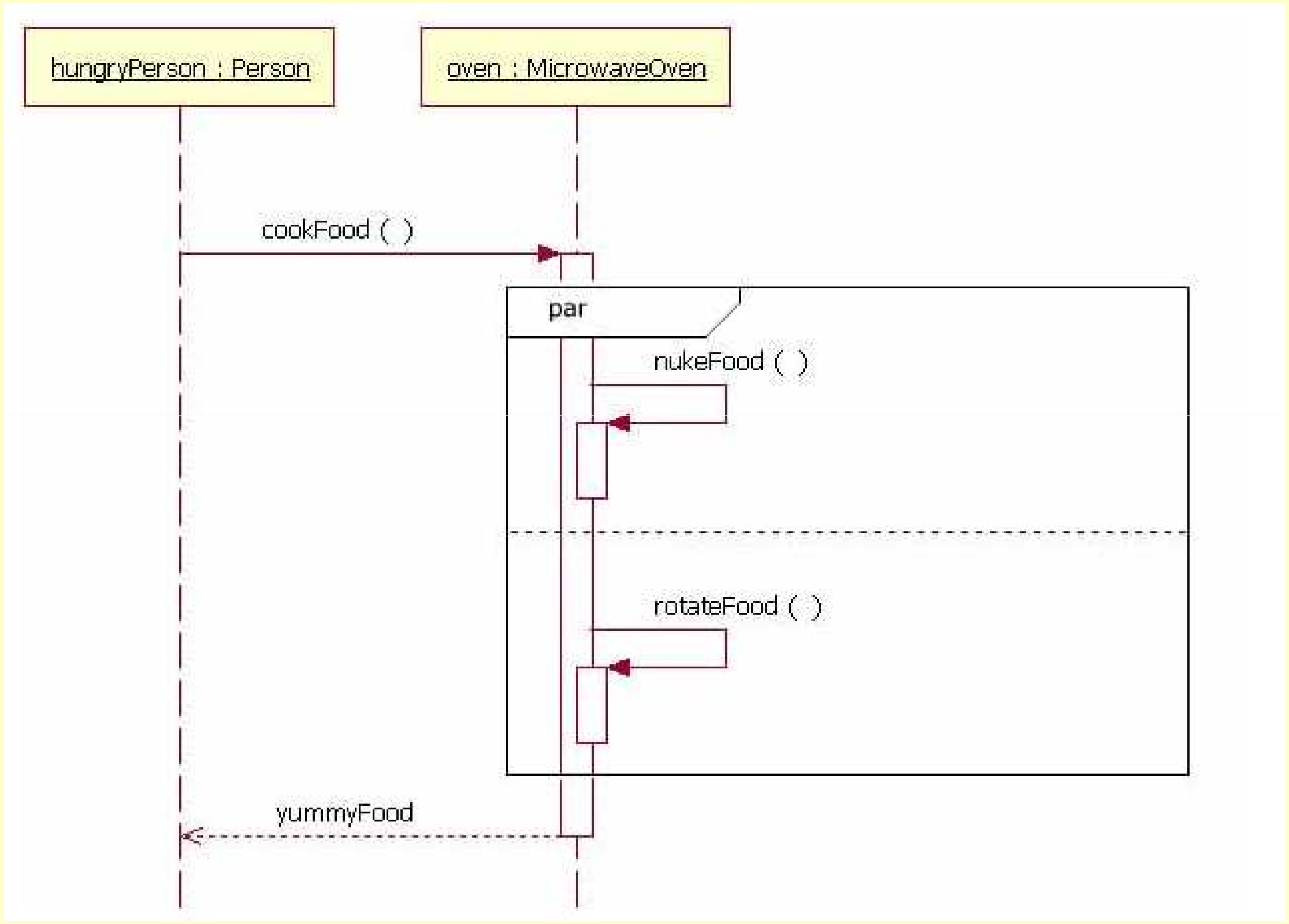
The parallel combination fragment element needs to be used when creating a sequence diagram that shows parallel processing activities.

The parallel combination fragment is drawn using a frame, and you place the text "par" in the frame's namebox.

You then break up the frame's content section into horizontal operands separated by a dashed line. Each operand in the frame represents a thread of execution done in parallel.

Sequence Diagrams: advanced concepts

Parallel example:



Sequence Diagrams: more lift design decisions

Draw a sequence diagram showing the communication between components of the lift when a user presses a button to request the lift at a floor

Collaboration Diagrams

Collaboration Diagrams show the interactions occurring between the objects participating in a specific situation.

This is more or less the same information shown by Sequence Diagrams but there the emphasis is put on how the interactions occur in time while the Collaboration Diagrams put the relationships between the objects and their topology in the foreground.

In Collaboration Diagrams messages sent from one object to another are represented by arrows, showing the message name, parameters, and the sequence of the message.

Collaboration Diagrams are specially well suited to showing a specific program flow or situation and are one of the best diagram types to quickly demonstrate or explain one process in the program logic.

Collaboration Diagrams

A Collaboration diagram is easily represented by modeling objects in a system and representing the associations between the objects as links.

The interaction between the objects is denoted by arrows. To identify the sequence of invocation of these objects, a number is placed next to each of these arrows.

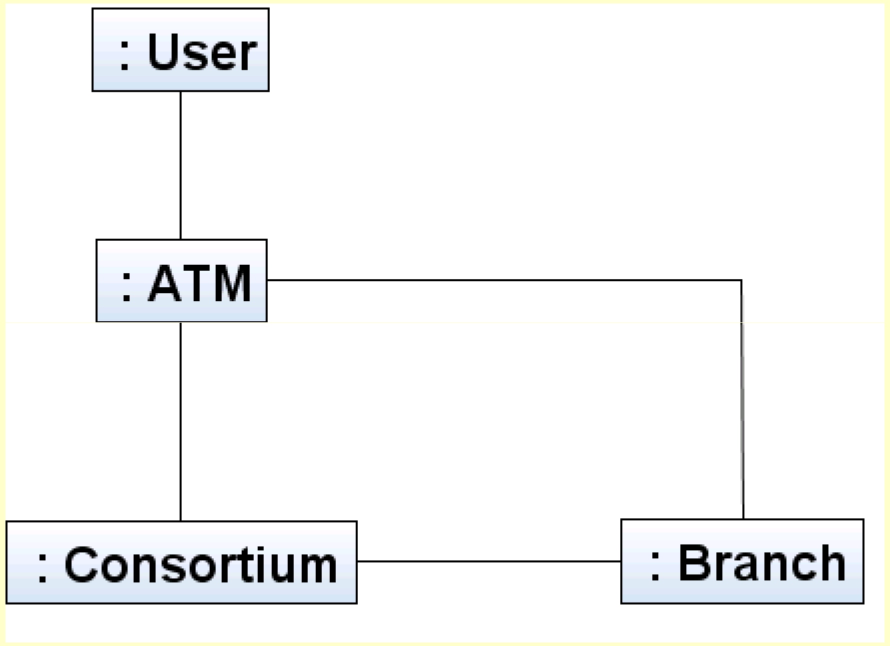
Defining a Collaboration diagram

Sophisticated modeling tools can easily convert a collaboration diagram into a sequence diagram and the vice versa.

Hence, the elements of a Collaboration diagram are essentially the same as that of a Sequence diagram.

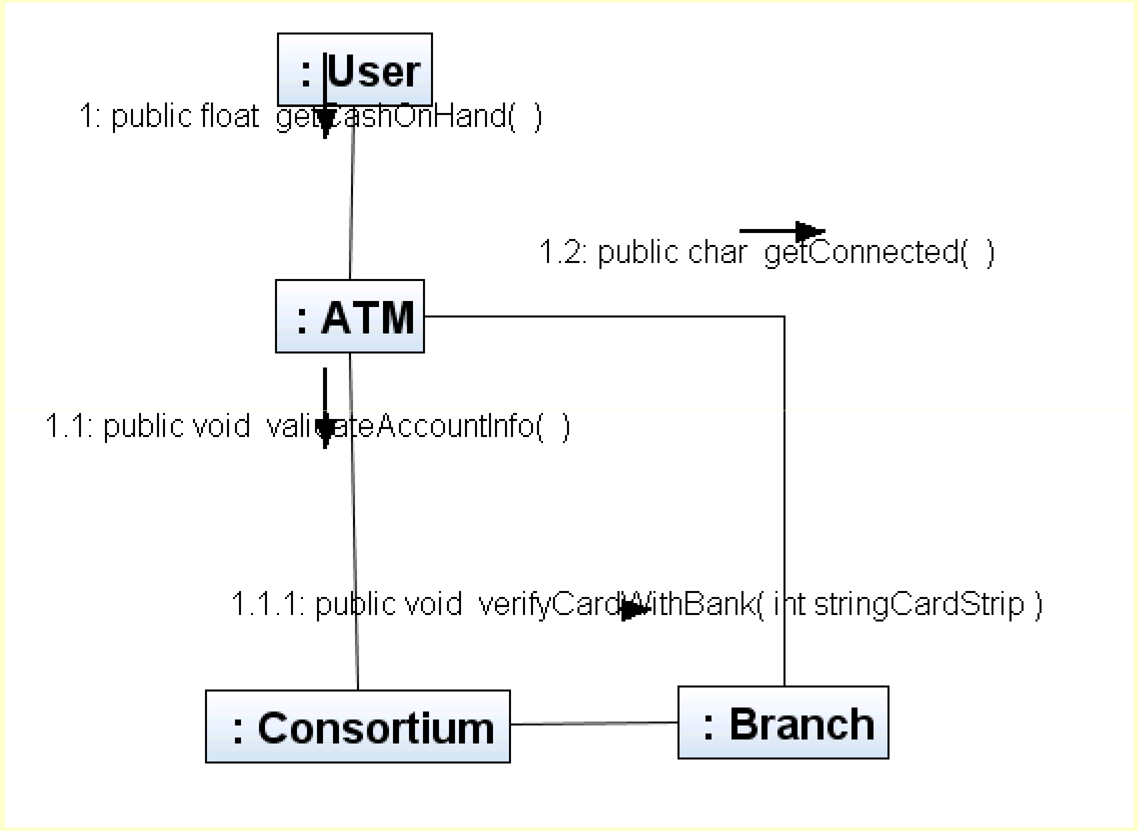
Collaboration Diagrams

Example Structure:



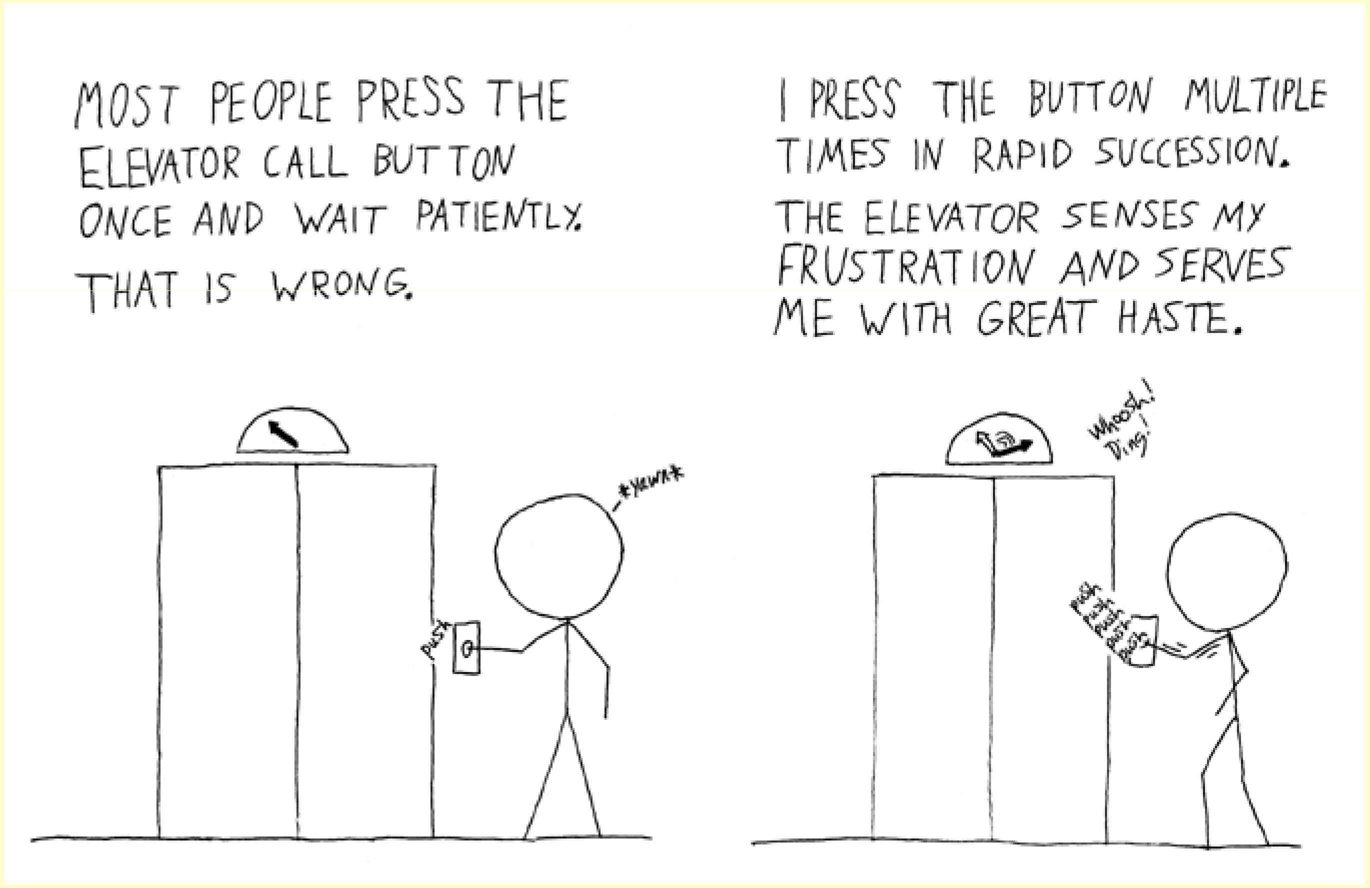
Collaboration Diagrams

Example Structure: add method information


: User
1: public float getCashOnHand( )
1.2: public char getConnected( )
: ATM
1.1: public void validateAccountInfo( )
1.1.1: public void verifyCardWithBank(int stringCardStrip)
: Consortium
: Branch

Collaboration Diagrams

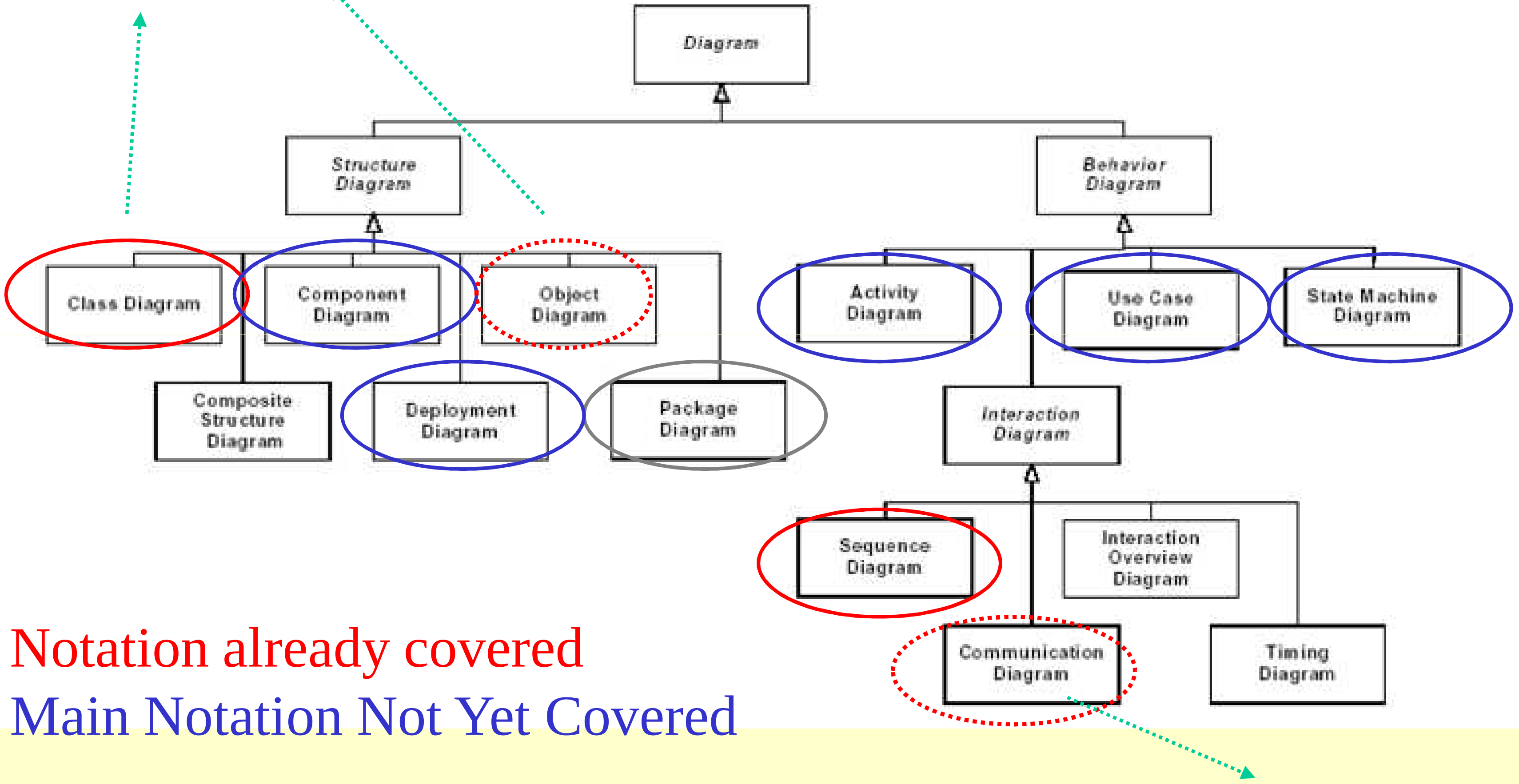
Transform the previous sequence diagram into a collaboration diagram


MOST PEOPLE PRESS THE ELEVATOR CALL BUTTON ONCE AND WAIT PATIENTLY.
THAT IS WRONG.
PUSH
YAWN
I PRESS THE BUTTON MULTIPLE TIMES IN RAPID SUCCESSION.
THE ELEVATOR SENSES MY FRUSTRATION AND SERVES ME WITH GREAT HASTE.
Whoosh! Ding!
PUSH PUSH PUSH PUSH

Some More UML Notation

Overview of UML2.0 diagrams (from wikipedia)

Object diagram is an instance of a class diagram



Notation already covered

Main Notation Not Yet Covered

Simpler form of *collaboration* diagram

Diagrams already seen:

Class Diagrams (and some) object diagrams: associations, aggregation, composition, generalization, multiplicity, visibility, stereotypes…

Interaction Diagrams: sequences, (and some) collaborations (communication diagrams)

More diagrams:

Use Cases, Activities, State Machines (and StateCharts)

Components, Deployment,

Generic Diagram(s) for structuring: **Packages**

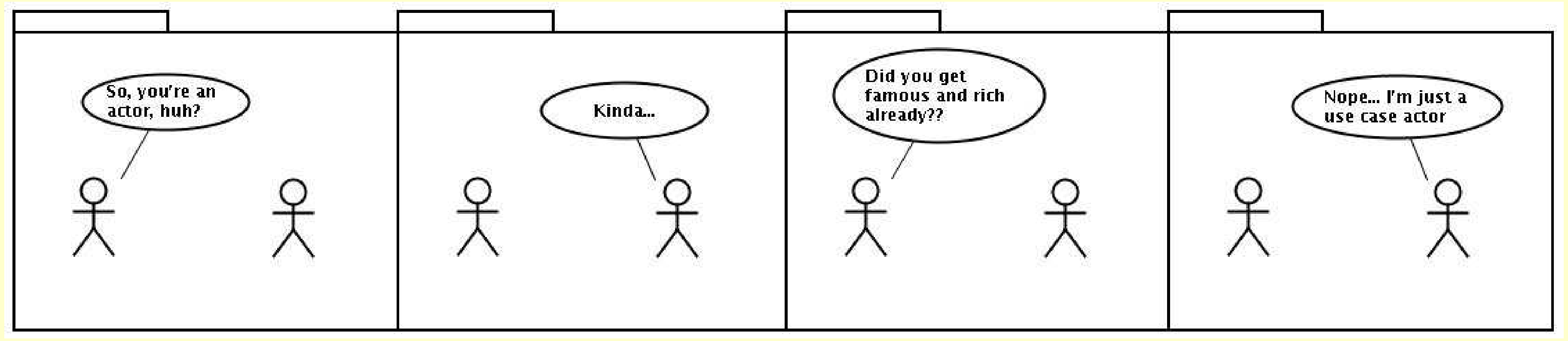
Advanced concepts**:** Events and signals, Processes and Threads, Time, Interface, Datatype, Subsystems, Patterns and Frameworks

Use Case Diagrams

Use case diagrams overview the usage requirements for a system.

They are useful for presentations to management and/or project stakeholders

For actual development you will find that use cases provide significantly more value because they describe "the foundations" of the actual requirements and can drive development of other models


So, you're an actor, huh?
Kinda...
Did you get famous and rich already??
Nope... I'm just a use case actor

Use Case Diagrams

Use Case Diagrams describe the relationships and dependencies between a group of *Use Cases* and the Actors participating in the process.

It is important to notice that Use Case Diagrams are not suited to represent the design, and cannot describe the internals of a system.

Use Case Diagrams are meant to facilitate the communication with the future users of the system, and with the customer, and are specially helpful to determine the required features the system is to have.

Use Case Diagrams tell, *what* the system should do but do not — and cannot — specify *how* this is to be achieved.

Use Case Diagrams

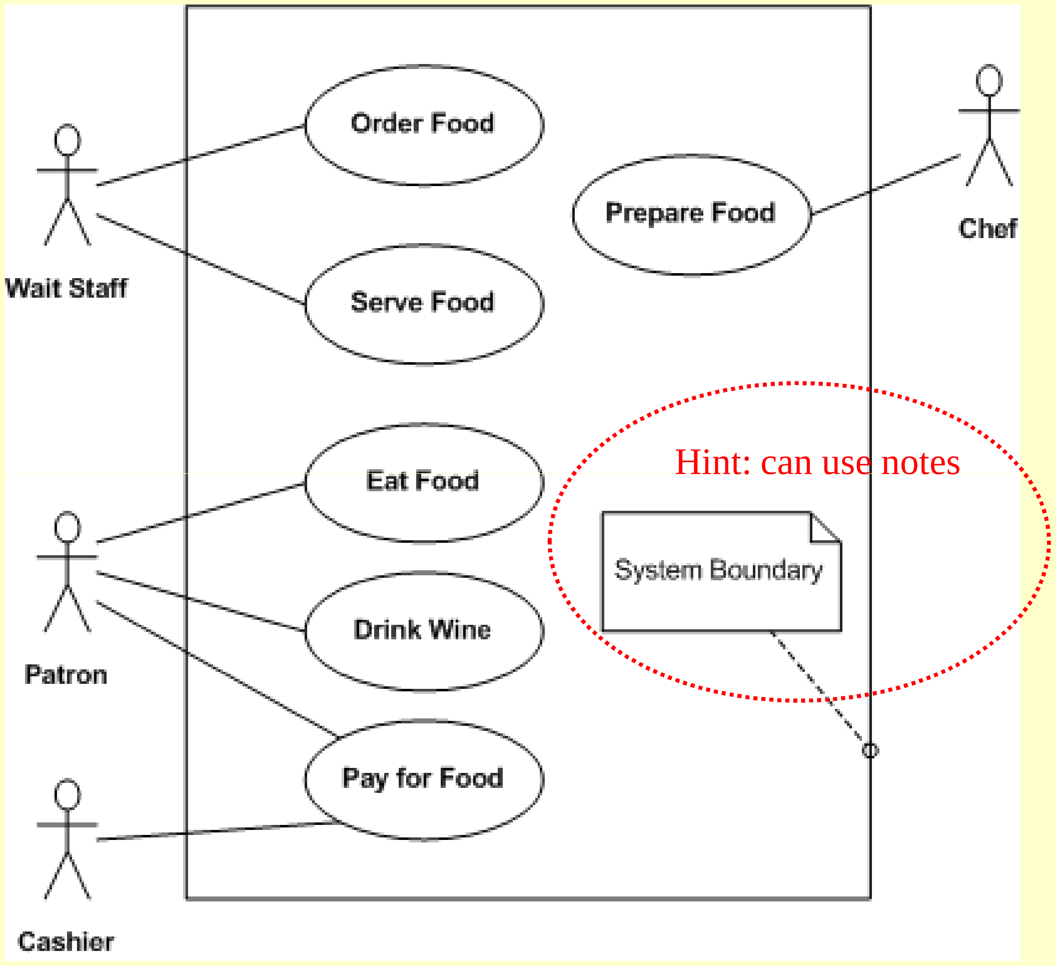
A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a **horizontal ellipse**.

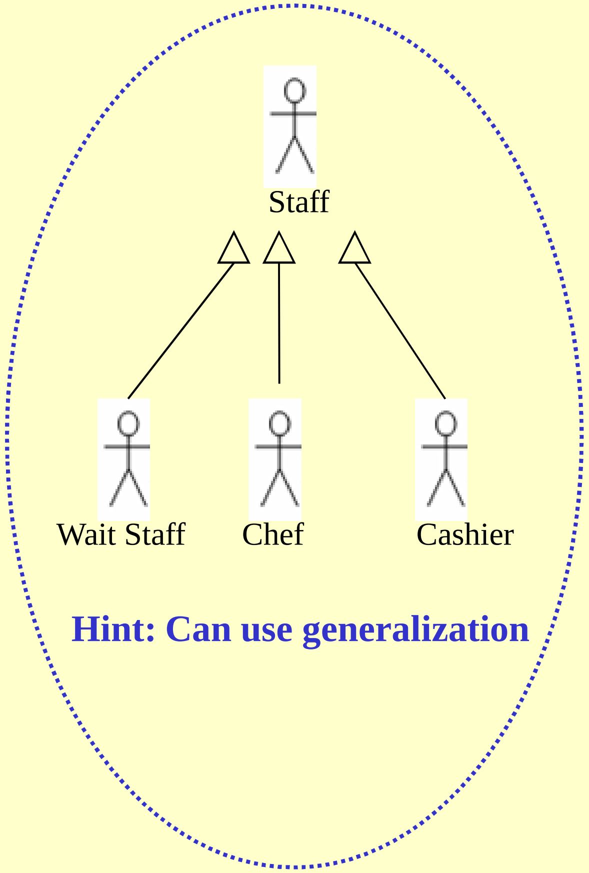
An actor is a person, organization, or external system that plays a role in one or more interactions with your system. Actors are (normally, by default) drawn as stick figures.

Associations between actors and use cases are indicated in use case diagrams by **solid lines**. An association exists whenever an actor is involved with an interaction described by a use case. Associations are modeled as lines connecting use cases and actors to one another, with an **optional arrowhead on one end of the line**. The arrowhead is often used to indicating the direction of the initial invocation of the relationship or to indicate the primary actor within the use case.

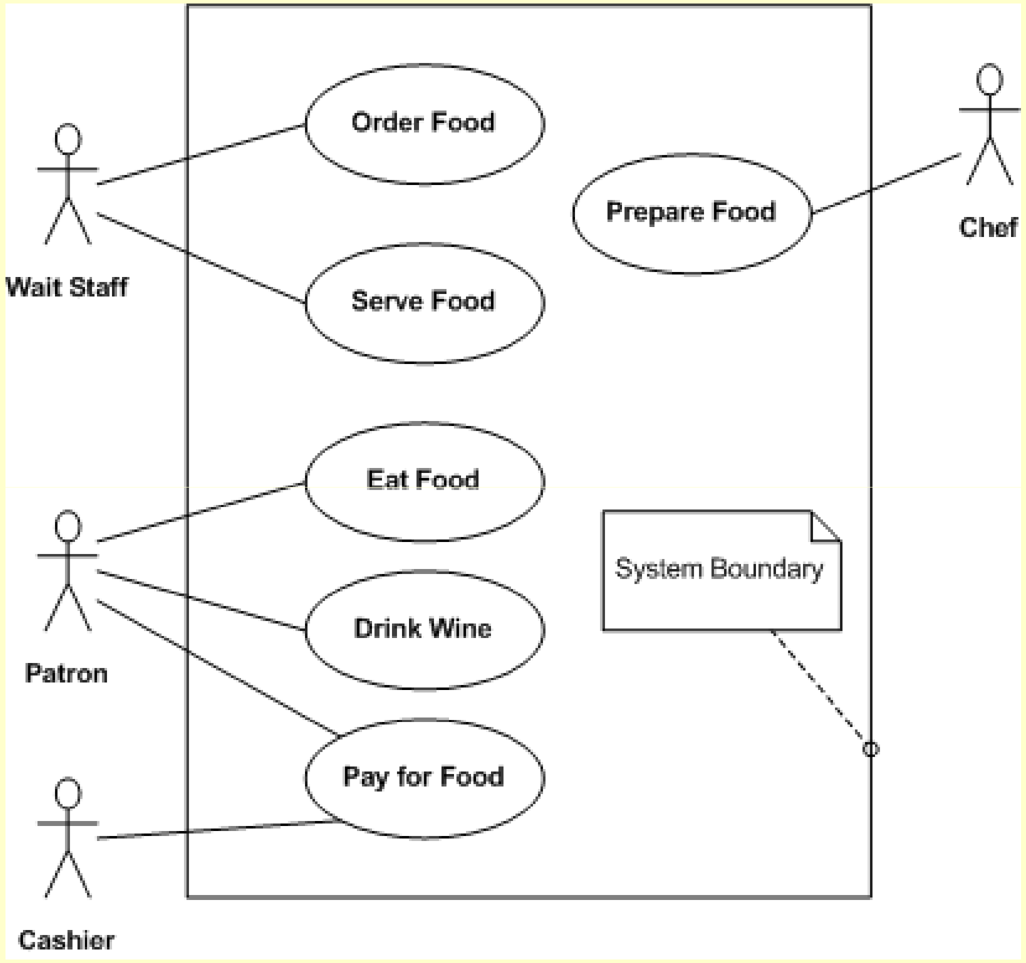
Hint: The arrowheads are typically confused with data flow and so their use is sometimes best avoided.

Use Case Diagram Example: a restaurant





Use Case Diagram Example: a restaurant

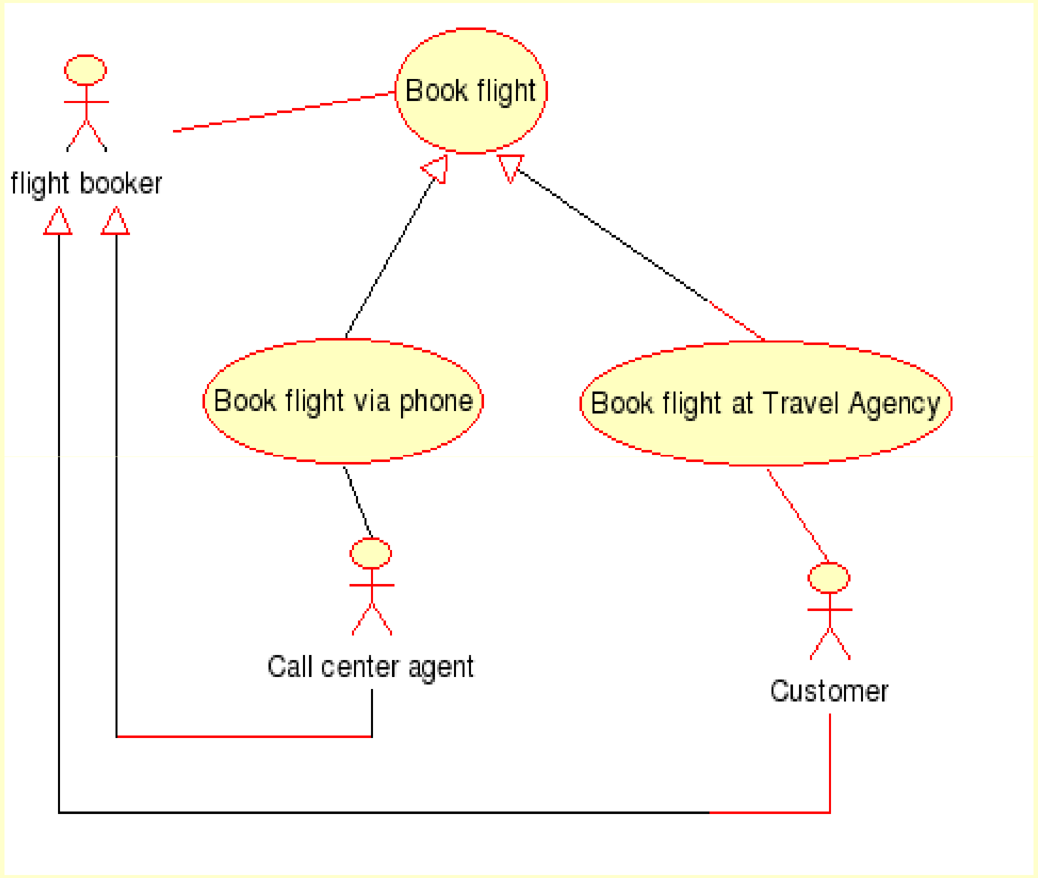


Interaction among actors is not shown on the use case diagram. If this interaction is essential to a coherent description of the desired behaviour, perhaps the system or use case boundaries should be re-examined.

Alternatively, interaction among actors can be part of the assumptions used in the use case.

However, note that actors are a form of role, a given human user or other external entity may play several roles. Thus the Chef and the Cashier may actually be the same person.

Use Case Diagram Example: a flight reservation system



An actor is an external entity (outside of the system) that interacts with the system by participating (and often initiating) a Use Case. Actors can be in real life people (for example users of the system), other computer systems or external events.

Actors do not represent the physical people or systems, but their role. This means that when a person interacts with the system in different ways (assuming different roles) he will be represented by several actors.

Use Case Diagrams

System boundary boxes (optional). You can draw a rectangle around the use cases, called the system boundary box, to indicates the scope of your system. Anything within the box represents functionality that is in scope and anything outside the box is not. System boundary boxes are rarely used,

Hint: such boxes can be usefully applied to identify which use cases will be delivered in each major release of a system.

Packages (optional). Packages are UML constructs that enable you to organize model elements (such as use cases) into groups. Packages are depicted as file folders and can be used on any of the UML diagrams, including both use case diagrams and class diagrams.

Hint: use packages only when diagrams become unwieldy, which generally implies they cannot be printed on a single page, to organize a large diagram into smaller ones.

Use Case Diagrams

It is vital to remember that when we describe the flow of a use case scenario, we are actually describing the interaction design of the software.

Many people make the mistake of treating these descriptions as requirements, and feel un-empowered to change them when necessary, leading to some pretty unusable systems.

Use case flows are not requirements. The functional goals of use case scenarios are the actual requirements. How users interact with the software to achieve those goals is the beginnings of the software design.

Use Case Diagrams

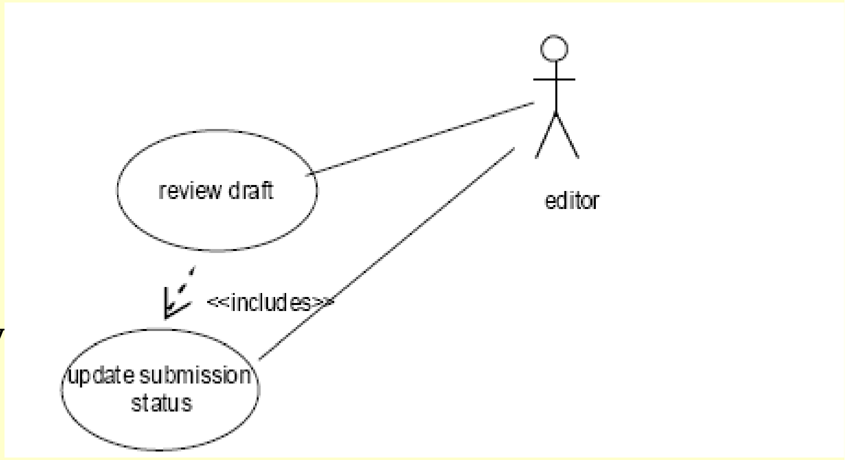
Use Case Relationships

Three major relationships among use cases are supported by the UML standard, which describes graphical notation for these relationships:

- **Include**
- **Extend**
- **Generalization**

Use Case Diagrams

Include



In one form of interaction, a given use case may *include* another.

The first use case often depends on the outcome of the included use case.

This is useful for extracting common behaviours from multiple use cases into a single description.

The notation is a dashed arrow from the including to the included use case, with the label "«include»".

This usage resembles a macro expansion where the included use case behaviour is placed inline in the base use case behaviour. There are no parameters or return values.

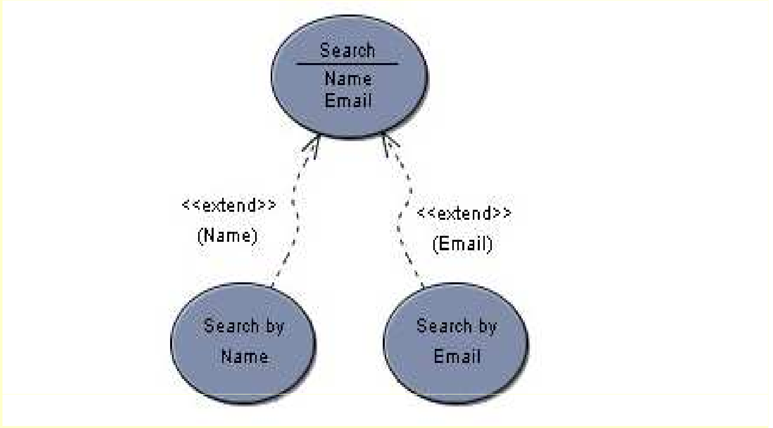
Use Case Diagrams

Extend

In another form of interaction, a given use case, (the extension) may *extend* another. This relationship indicates that the behaviour of the extension use case may be inserted in the extended use case under some conditions.

The notation is a dashed arrow from the extension to the extended use case, with the label «extend».

This can be useful for dealing with special cases, or in accommodating new requirements during system maintenance and extension.



search by name is said to extend *search* at the *name* extension point. The extends link is more controlled than the generalization link in that functionality can only be added at the extension points.

Use Case Diagrams

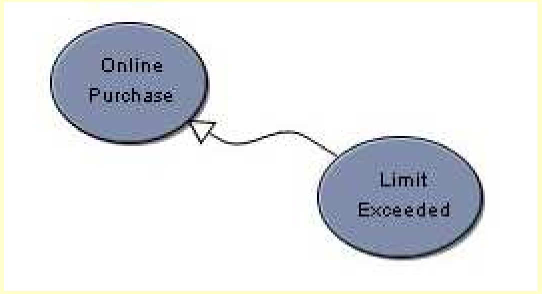
Generalization

In the third form of relationship among use cases, a *generalization/specialization* relationship exists.

A given use case may be a specialized form of an existing use case.

The notation is a solid line ending in a hollow triangle drawn from the specialized to the more general use case.

This resembles the object-oriented concept of sub-classing, in practice it can be both useful and effective to factor common behaviors, constraints and assumptions to the general use case, describe them once, and deal with *same as except* details in the specialized cases.



The use case *limit exceeded* describes a situation in which the usual scenario of *online purchase* is not performed. Use cases that generalize another use case should only specify an alternative, even exceptional, scenario to the use case being generalized. The overall goal of the use cases should be the same.

Creating Use Case Diagrams

Hint: there are two common approaches -

The “all actors first” approach

Some like to start by identifying as many actors as possible. You should ask how the actors interact with the system to identify an initial set of use cases. Then, on the diagram, you connect the actors with the use cases with which they are involved. If an actor supplies information, initiates the use case, or receives any information as a result of the use case, then there should be an association between them. As you begin to notice similarities between use cases, or between actors, start modeling the appropriate relationships between them.

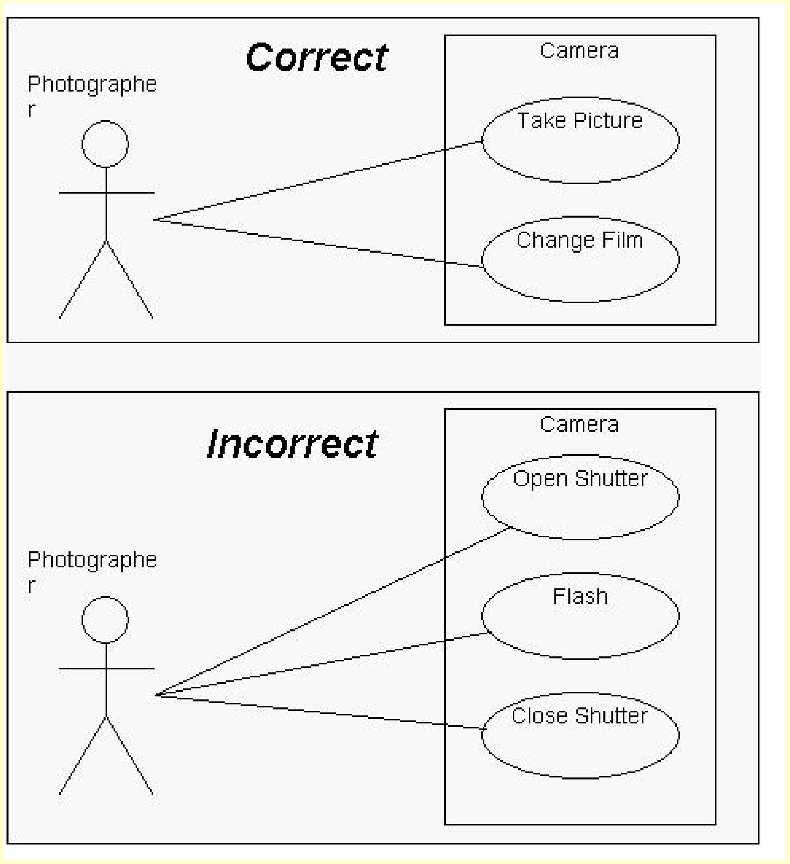
The “one actor at a time” approach

Others like to start by identifying one actor and the use cases that they’re involved with first and then evolve the model from there. Both approaches work. The important point is that different people take different approaches so you need to be flexible.

Creating Use Case Diagrams

Hint: what to include ??

In the diagram we would like to represent the use cases for a camera. Suppose we choose "Open Shutter", "Flash", and "Close Shutter" as the top-level use cases. Certainly these are all behaviours that a camera has, but no photographer would ever pick up their camera, open the shutter, and then put it down, satisfied with their photographic session for the day. The crucial thing to realize is that these behaviours are not done in isolation, but are rather a **part** of a more high-level use case, "Take Picture"



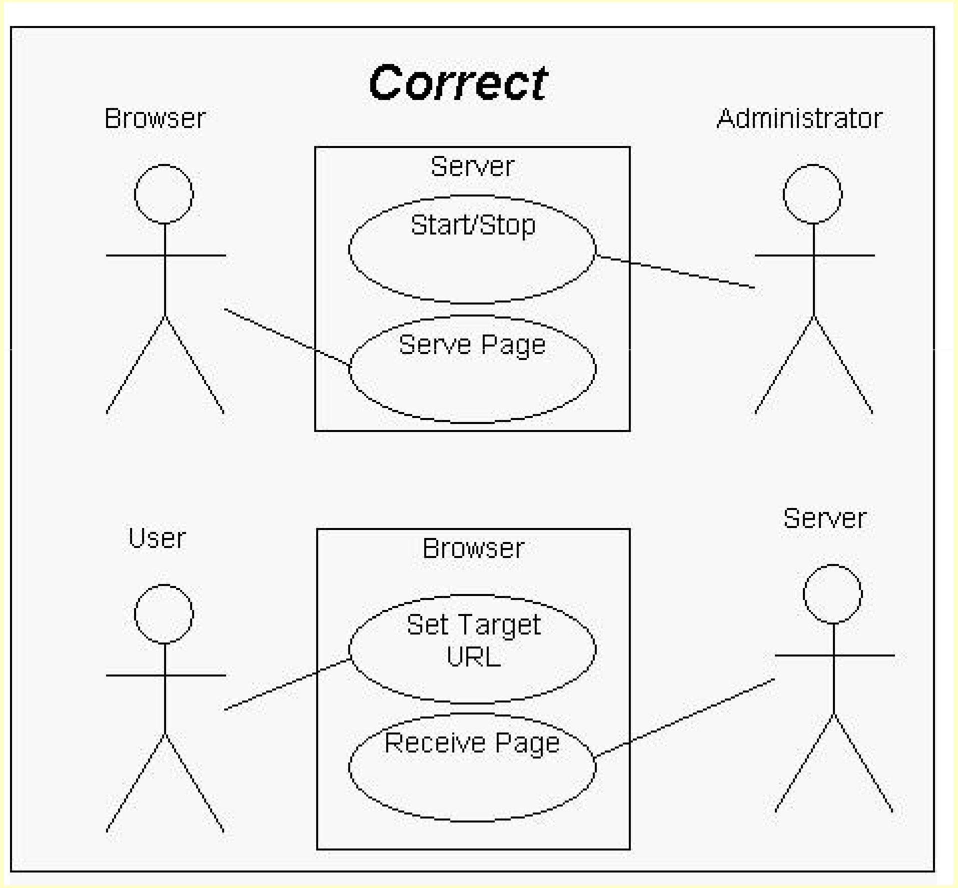
Creating Use Case Diagrams

The actors in my diagram have interactions. How do I represent them?
If there are interactions between the actors in your system, you cannot represent those interactions on the same diagram as your system. What you can do instead is draw a separate UCD, treating one of the actors itself as a system, and your original system (along with the other actors) as actors on this new diagram.

Example: Suppose you wanted to diagram the interactions between a user, a web browser, and the server it contacts. Since you can only have one system on the diagram, you must choose one of the obvious "systems", such as the server. You might then be tempted to draw interaction lines between the actors, but this is a problem because it isn't clear what the interaction means, so it isn't helpful to show it here. A more useful solution would be to draw two diagrams, showing all of the interactions.

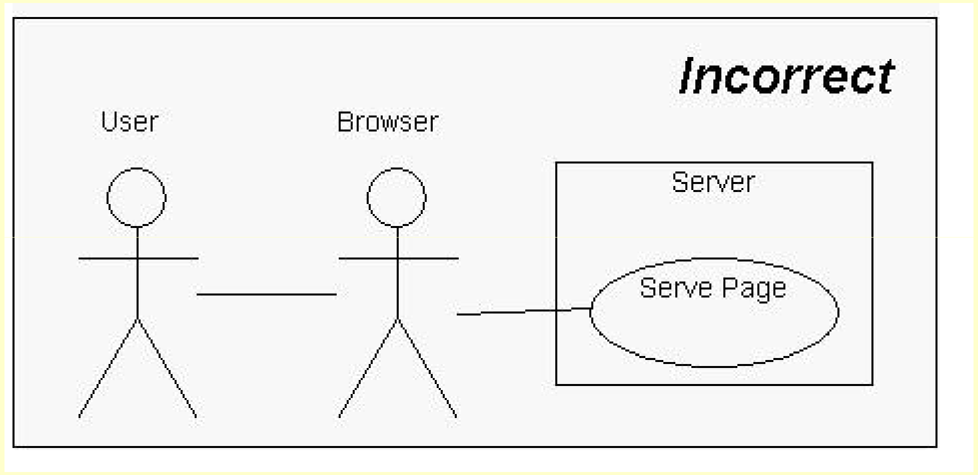
Creating Use Case Diagrams

The actors in my diagram have interactions. How do I represent them?

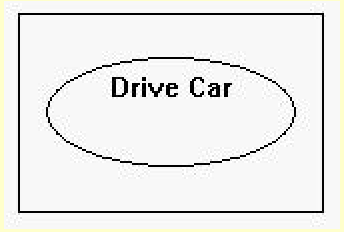


Creating Use Case Diagrams

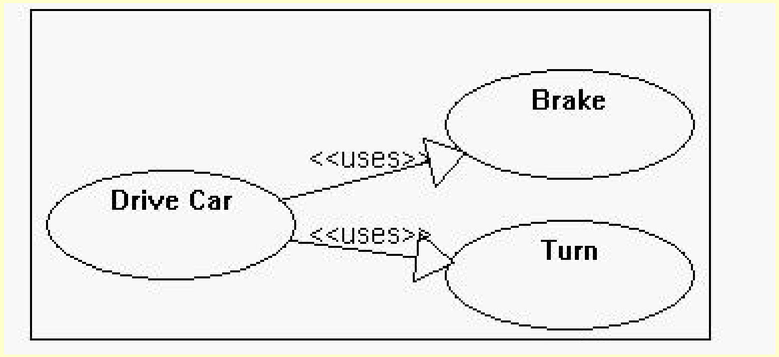
The actors in my diagram have interactions. How do I represent them?



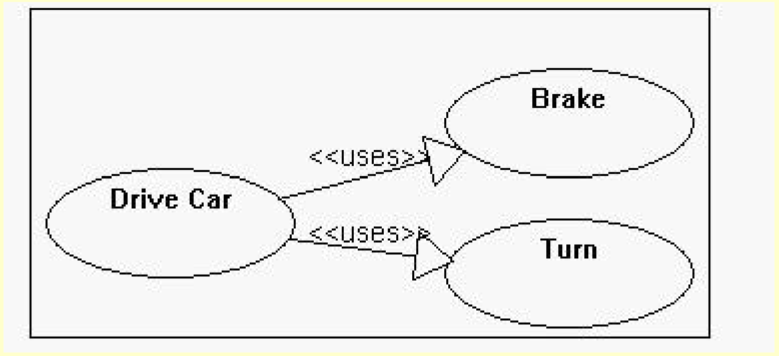
Evolution of use case diagrams



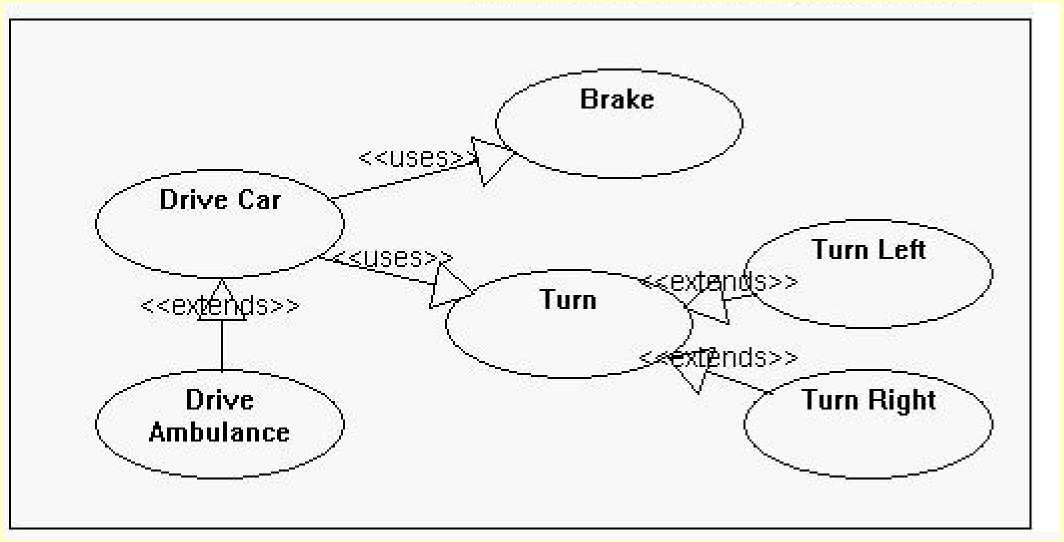
becomes



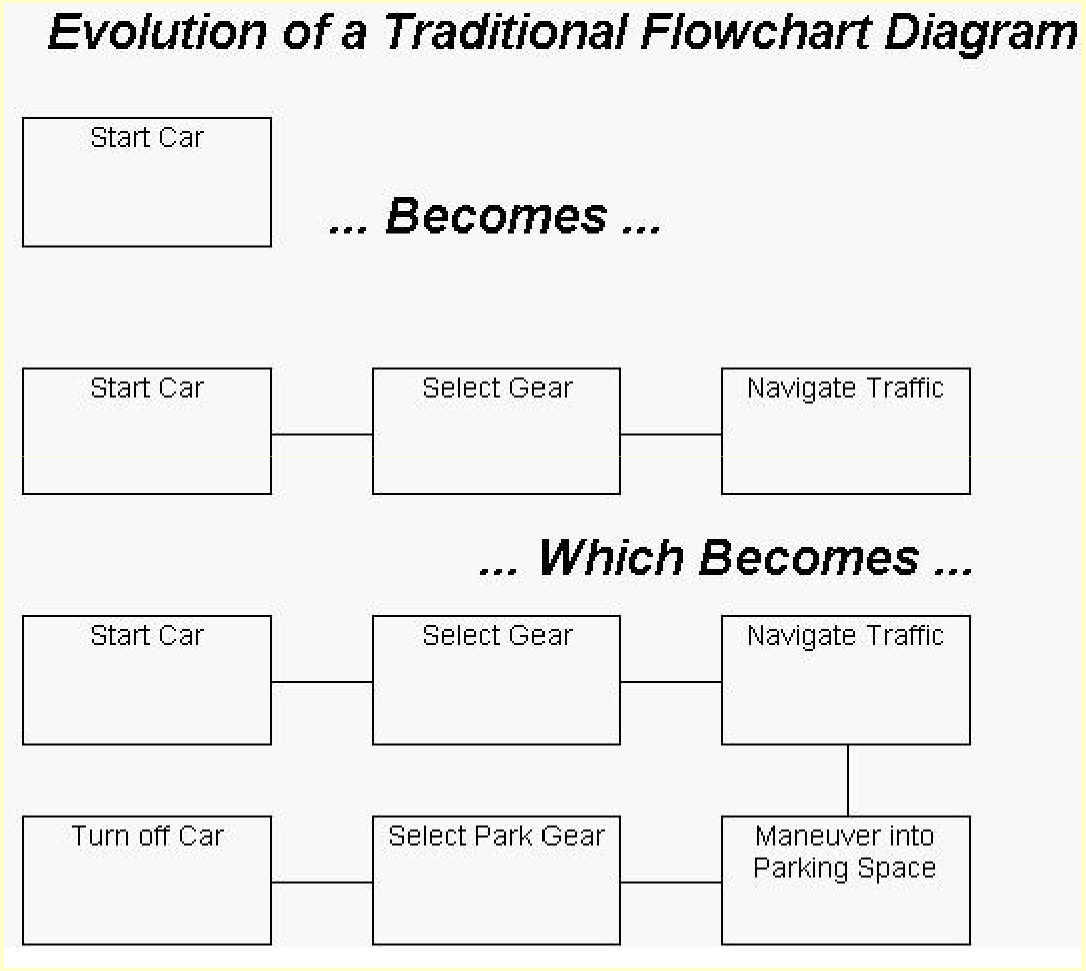
Evolution of use case diagrams



becomes



Evolution of use case diagrams vs evolution flow charts

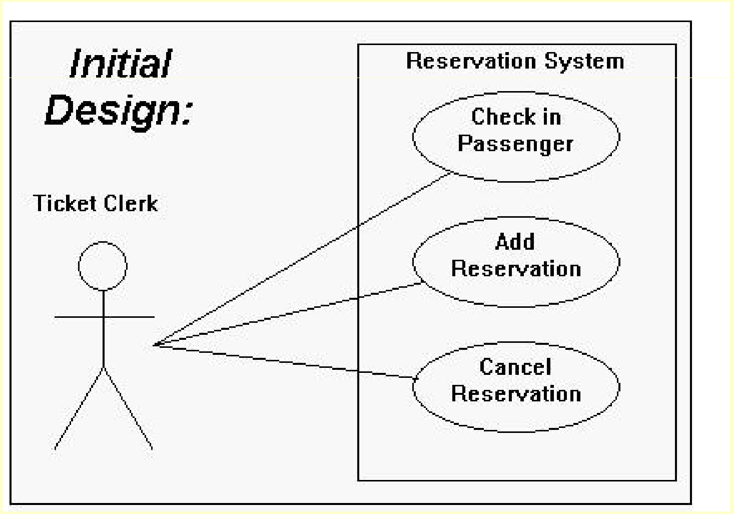


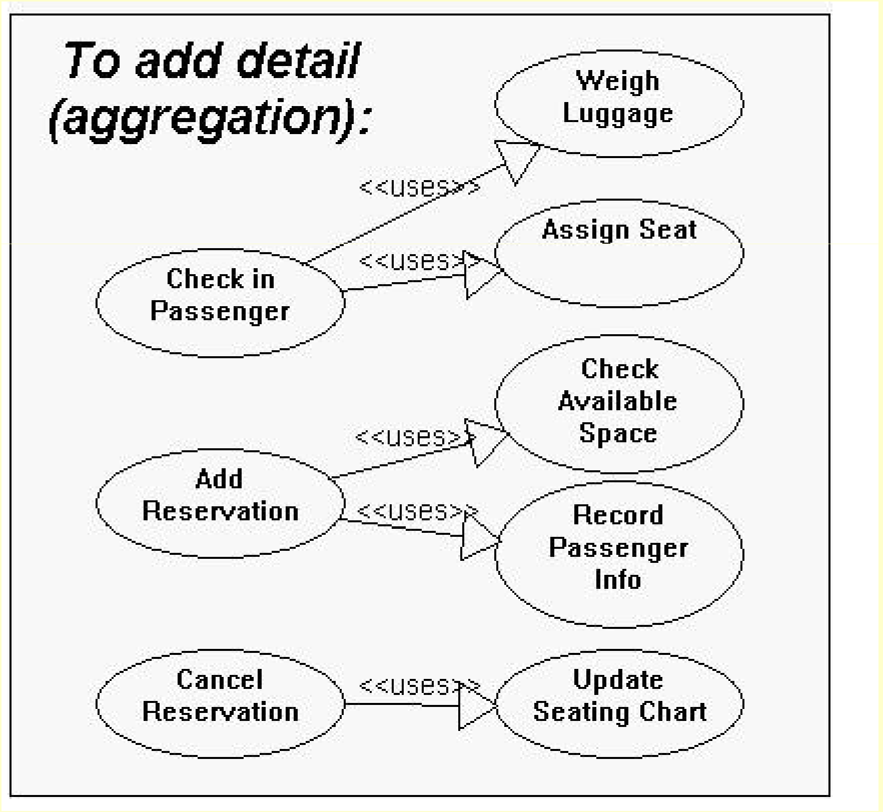
UCDs represent functionality in a top-down way, whereas flow charts represent behavior in a linear, time-based way.

Also, the way you develop them is all-together different.

Extends or uses?

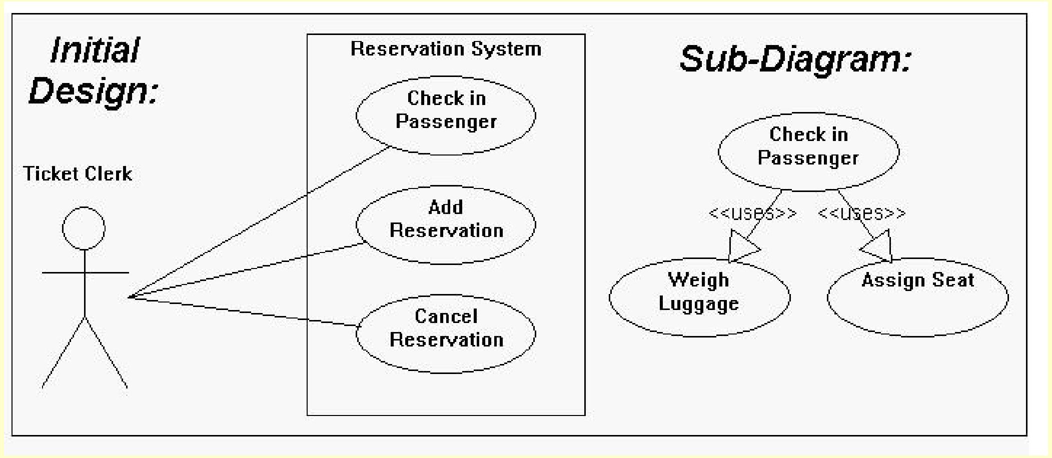
The *uses arrow* is drawn from a use case X to another use case Y to indicate that *the process of doing X always involves doing Y at least once*

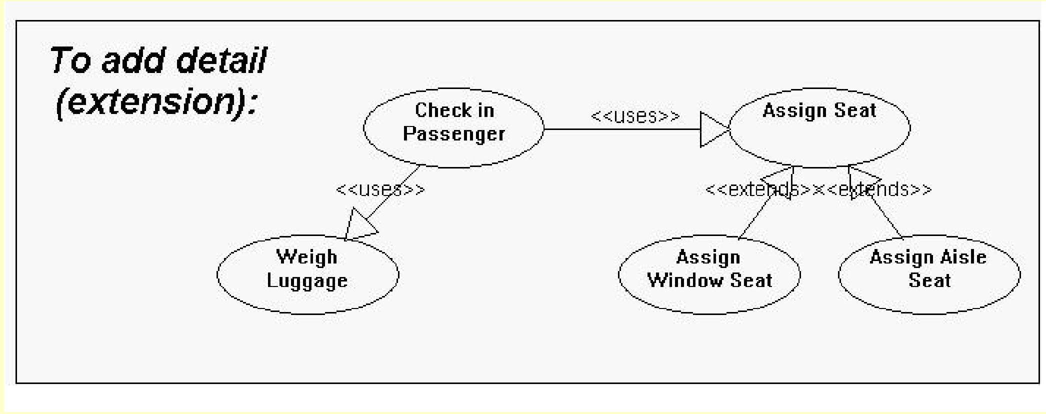




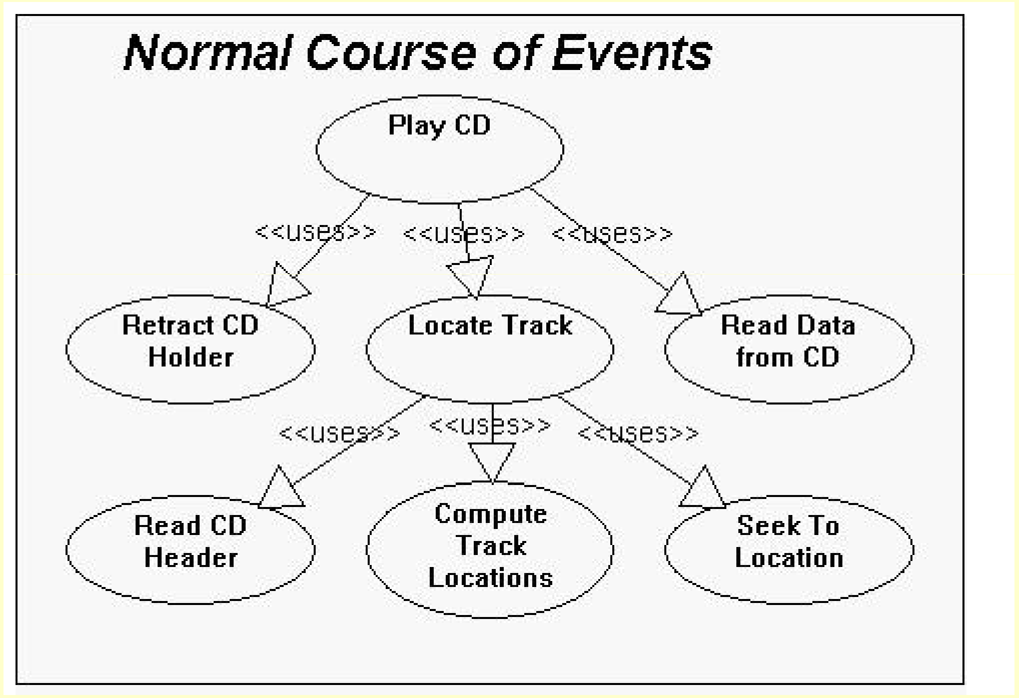
Extends or uses?

The *extends arrow* (or *extends edge*) is drawn from a use case X to a use case Y to indicate that the process X is a special case behaviour of the same type as the more general process Y. You would use this in situations where your system has a number of use cases (processes) that all have some subtasks in common, but each one has something different about it that makes it impossible for you to just lump them all together into the same use case.

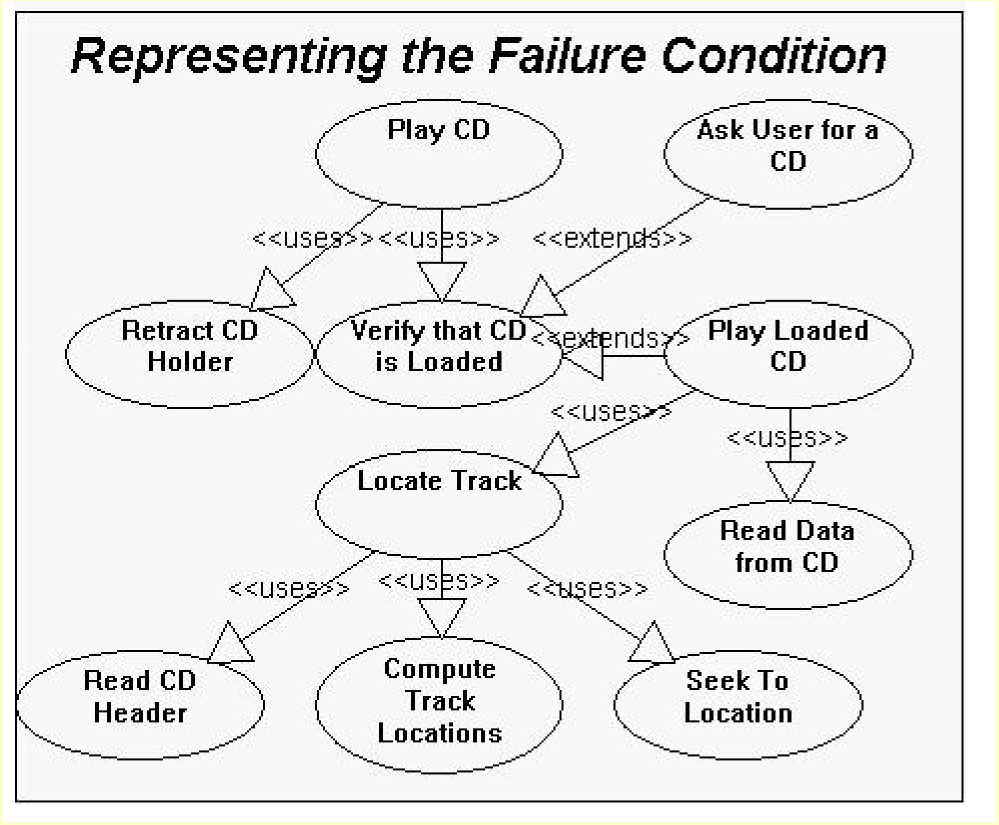




The scenario I want to describe branches into several possible outcomes, or has some error conditions. How can I represent that with Use Case Diagrams?



The scenario I want to describe branches into several possible outcomes, or has some error conditions.



Representing failure and branching conditions is often best done with a Sequence Diagram or flow chart, but there are some grey-area cases when it isn't clear whether or not a Use Case Diagram is appropriate.

A rule of thumb: if in representing the branching actions in the Use Case Diagram you must add significantly more use case ovals, and the resulting diagram is muddy or confusing, *consider using a different diagramming style.*

Activity and state diagrams

Interaction diagrams demonstrate the behaviour of several objects when executing a single use case.

When you want to show the sequence of events on a broader scale use **activity** and **state diagrams.**

An **activity** is the execution of a task whether it be a physical activity or the execution of code. Simply put, the **activity diagram** shows the sequence of activities.

Like the simple flow chart, activity diagrams have support for conditional behaviour, but has added support for parallel execution as well.

A **state diagram** shows the change of an object through time. Based upon events that occur, the state diagram shows how the object changes from start to finish.

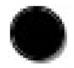
Activity and state diagrams

Activity diagrams are used to show workflow in parallel and conditionally. They are useful when working out the order and concurrency of a sequential algorithm, when analyzing the steps in a business process and when working with threads.

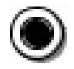
State diagrams show the change of an object over time and are useful when an object exhibits interesting or unusual behaviour - such as that of a user interface component.

As always, use these diagrams only when they serve a purpose. Don't feel that you have to draw a state diagram for every object in your system and an activity diagram for every process. Use them where they add to your design. You may not even include these diagrams in your design, and your work may still be complete and useful.

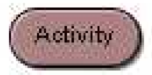
Activity diagrams



Start: each activity diagram has one start (above) at which the sequence of actions begins.



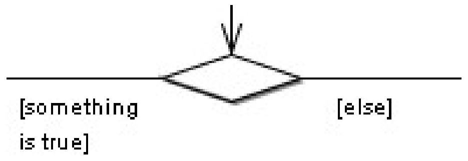
End: each activity diagram has one finish at which the sequence of actions ends



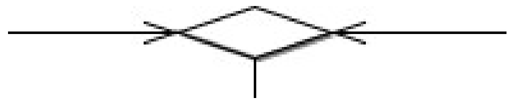
Activity: activities are connected together by transitions. Transitions are directed arrows flowing from the previous activity to the next activity. They are optionally accompanied by a textual label of the form:

[guard] label

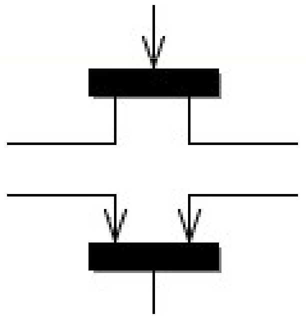
The guard is a conditional expression that when true indicates that the transition is taken. The label is also optional and is freeform.



Activity diagrams

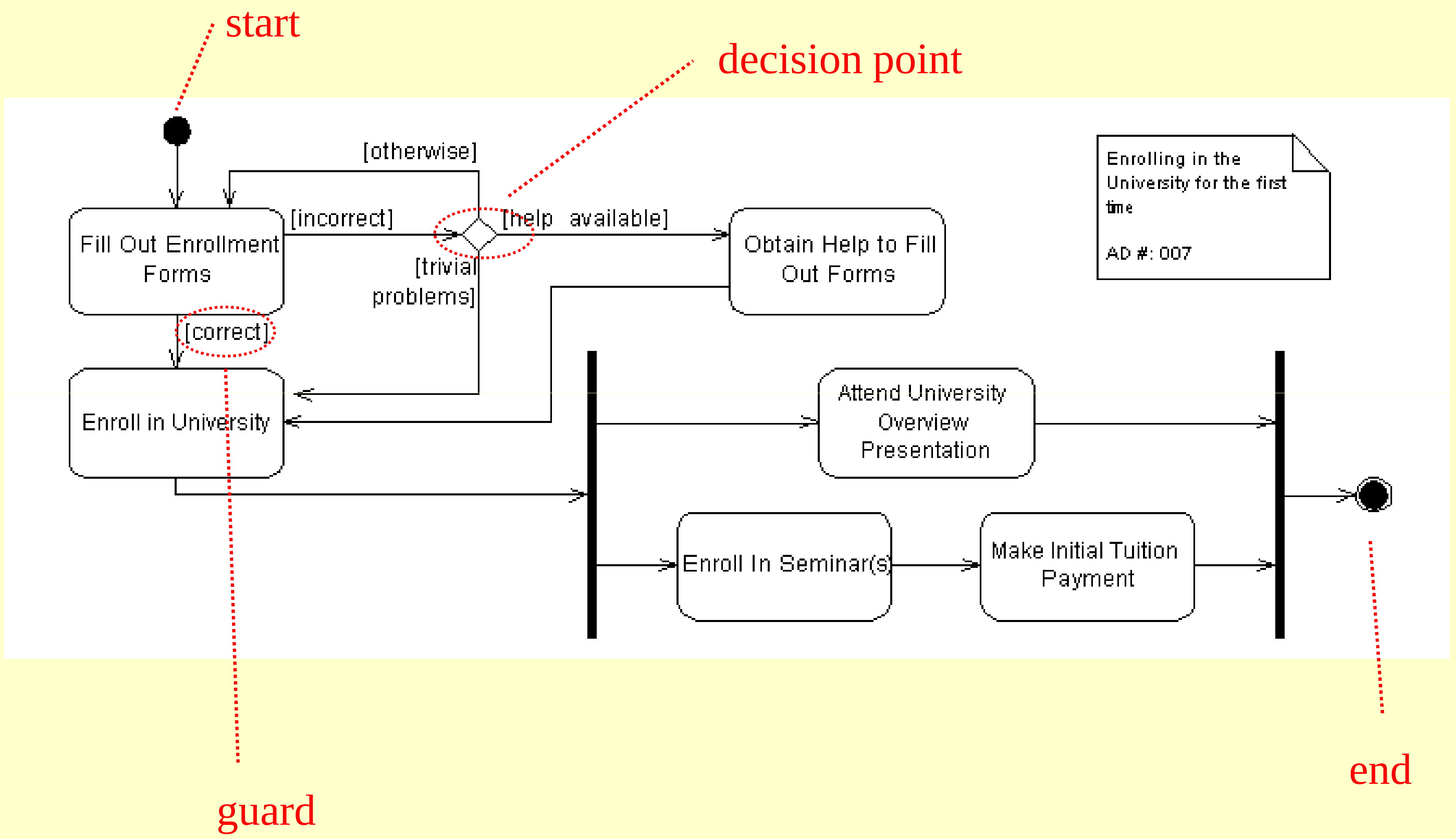


To show conditional behaviour use a branch and a merge. The top diamond is a branch and has only one transition flowing into it and any number of mutually exclusive transitions flowing out. That is, the guards on the outgoing transitions must resolve themselves so that only one is followed. The merge is used to end the conditional behaviour. There can be any number of incoming, and only one outgoing transition.

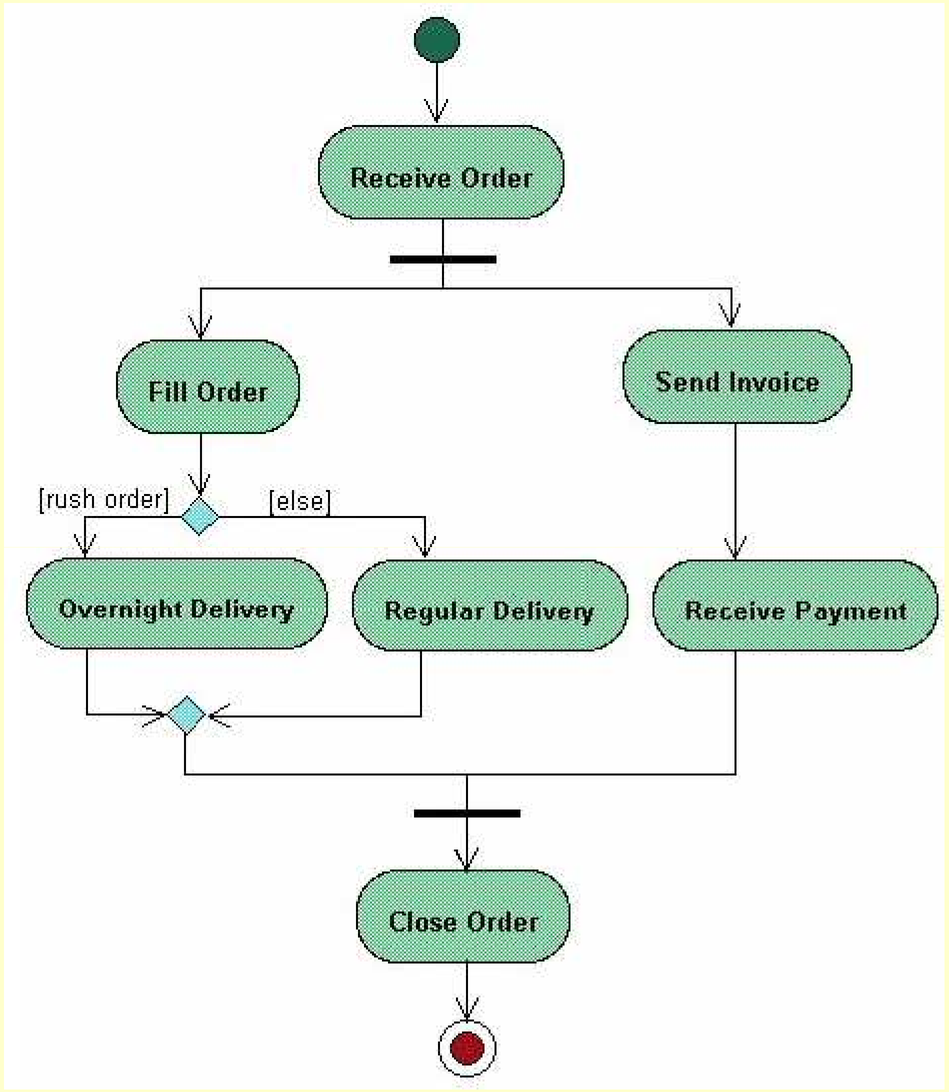


To show parallel behaviour use a fork and a join. The fork (top) has one transition entering and any number of transitions exiting, all of which will be taken. The join (bottom) represents the end of the parallel behaviour and has any number of transitions entering, and only one leaving.

Activity diagrams: a student enrolement example


start
decision point
[otherwise]
[incorrect]
[help available]
Fill Out Enrollment Forms
Obtain Help to Fill Out Forms
[trivial problems]
[correct]
Enrolling in the University for the first time
AD #: 007
Enroll in University
Attend University Overview Presentation
Enroll In Seminar(s)
Make Initial Tuition Payment
guard
end

Activity diagrams: an order example



Once the order is received the activities split into two parallel sets of activities.

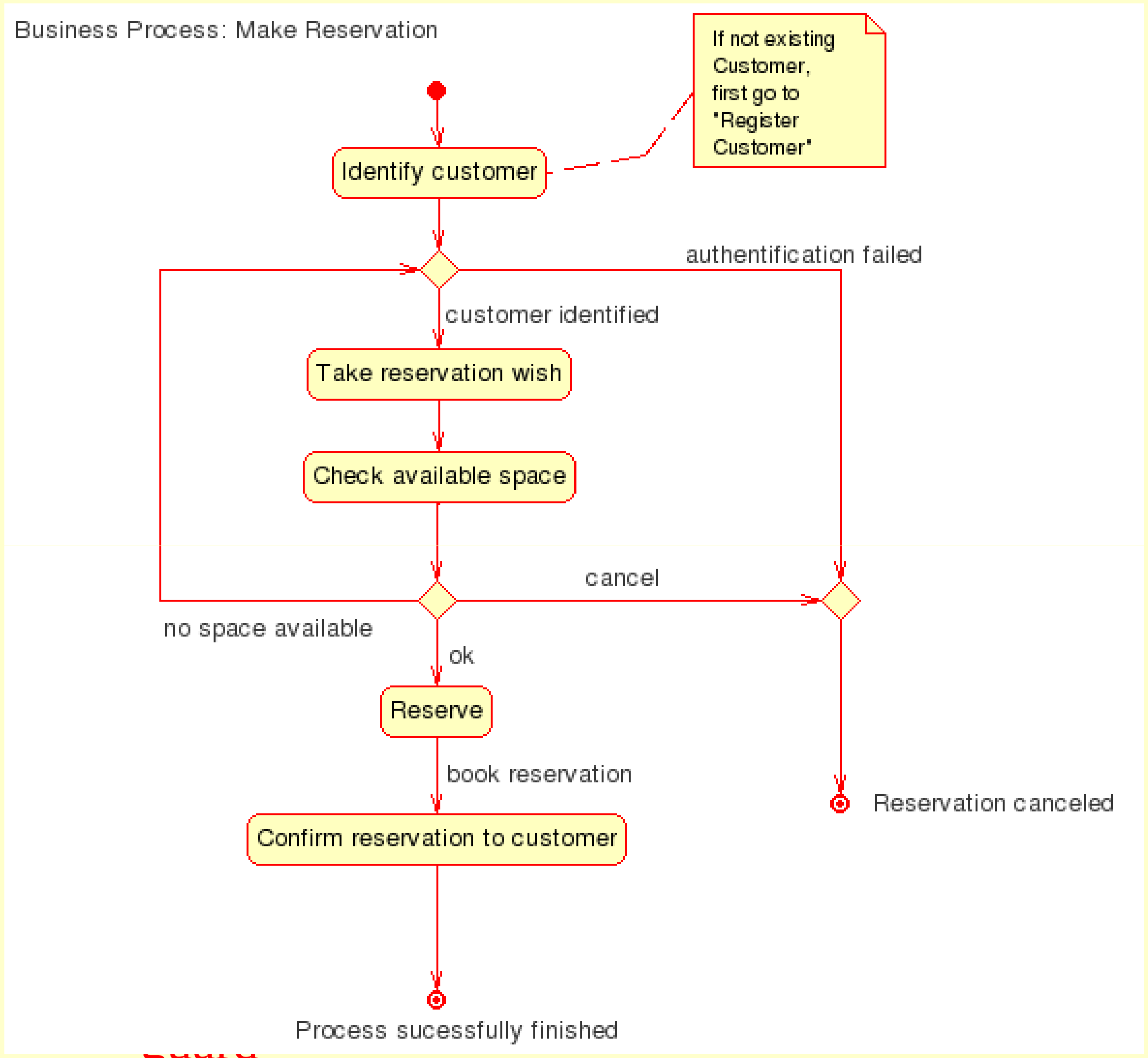
One side fills and sends the order while the other handles the billing.

On the Fill Order side, the method of delivery is decided conditionally.

Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.

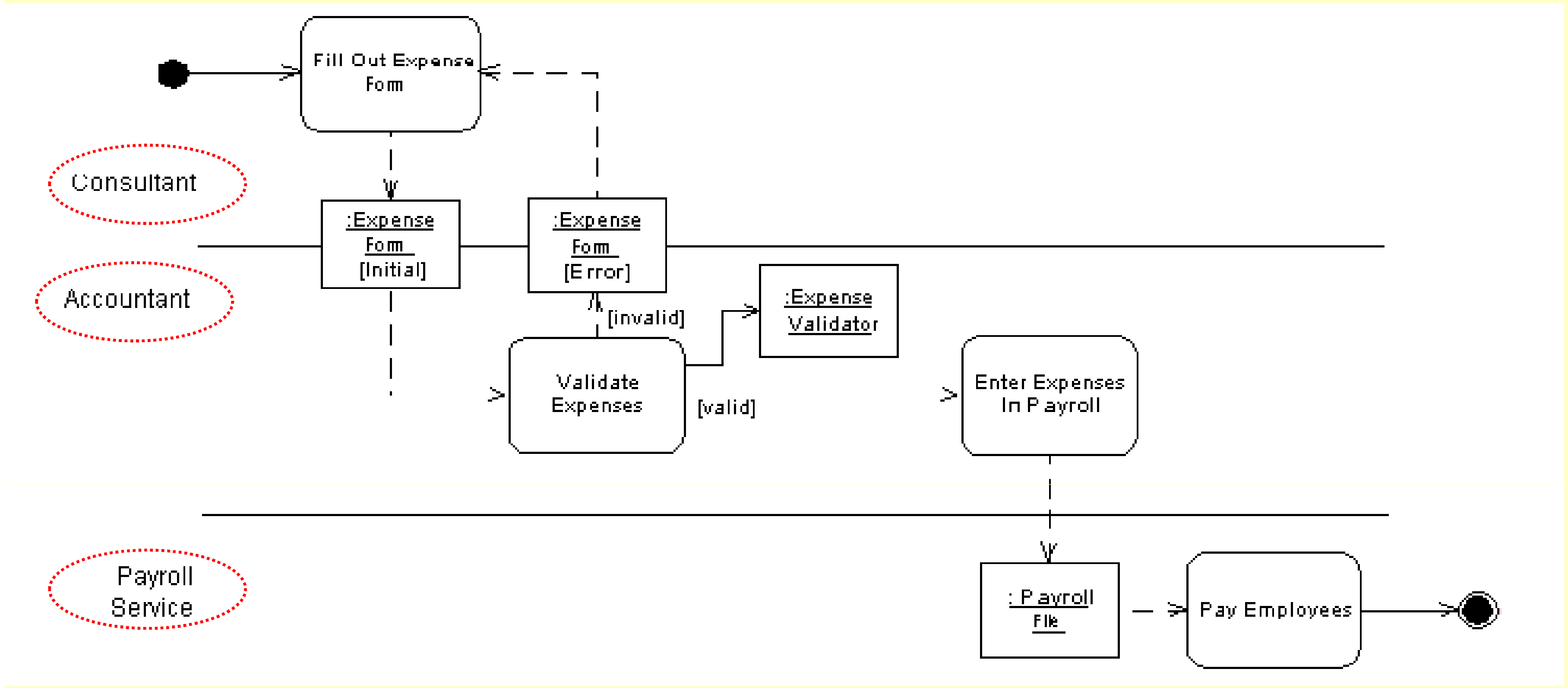
Finally the parallel activities combine to close the order.

Activity diagrams: a reservation example



guard

Activity diagrams: a (swimlane) expenses example



A swimlane is a way to group activities performed by the same actor on an activity diagram or to group activities in a single thread. The example includes three swimlanes, one for each actor.

Activity diagrams or state diagrams?

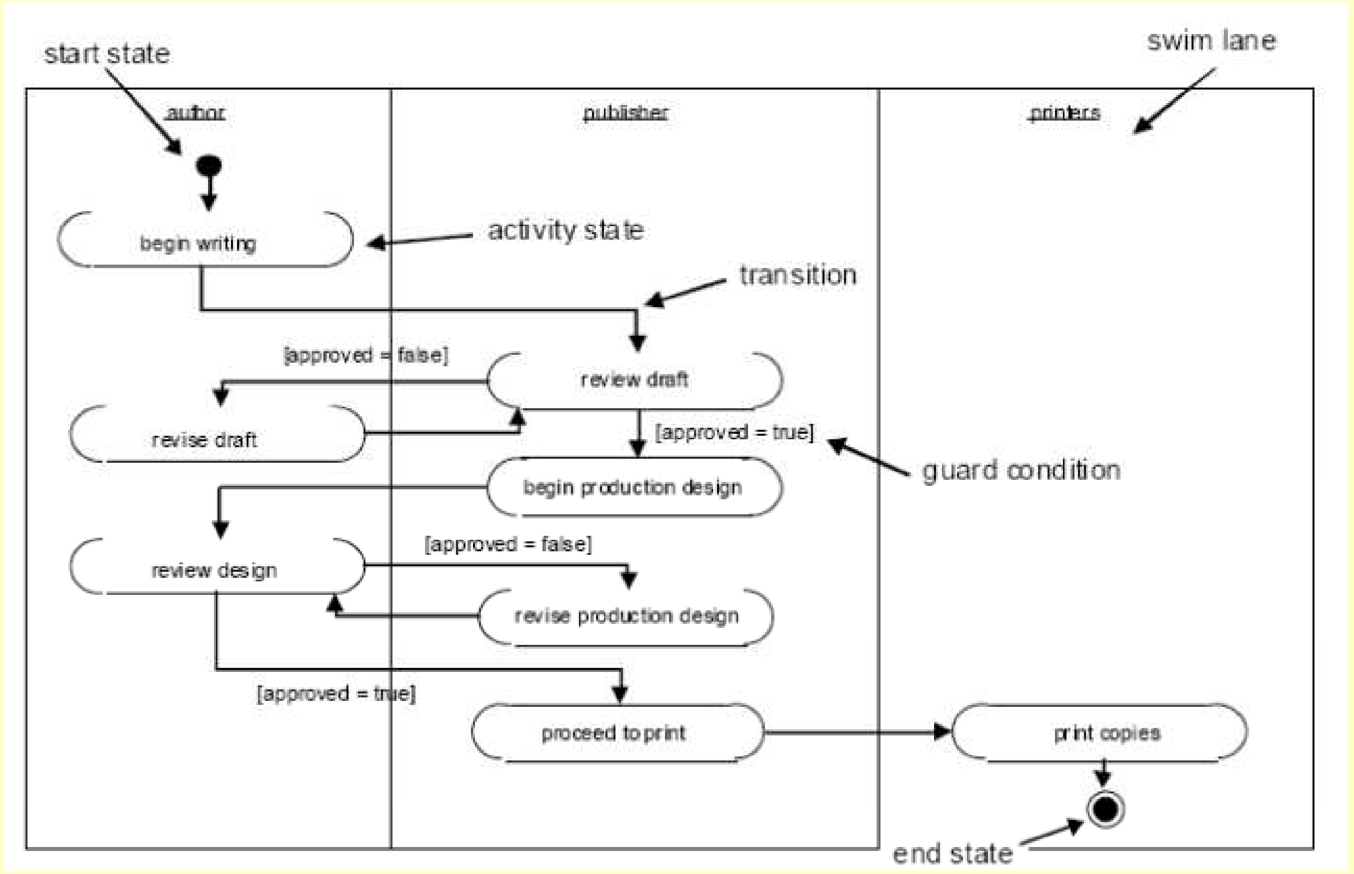
An activity diagram shows sequences of activity states (actions to you and me) where when one action is complete the flow immediately moves on to the next action.

This automatic transitioning from one activity state to the next is what distinguishes activity diagrams from their close cousin, state transition diagrams. In state transition diagrams, transitions from one state to another occur as the result of events, and don’t happen automatically.

It is perfectly legal in UML to include state transition elements in an activity diagram: for example, to show how, after completing a sequence of actions, a system waits for user input before moving on to the next step in the flow.

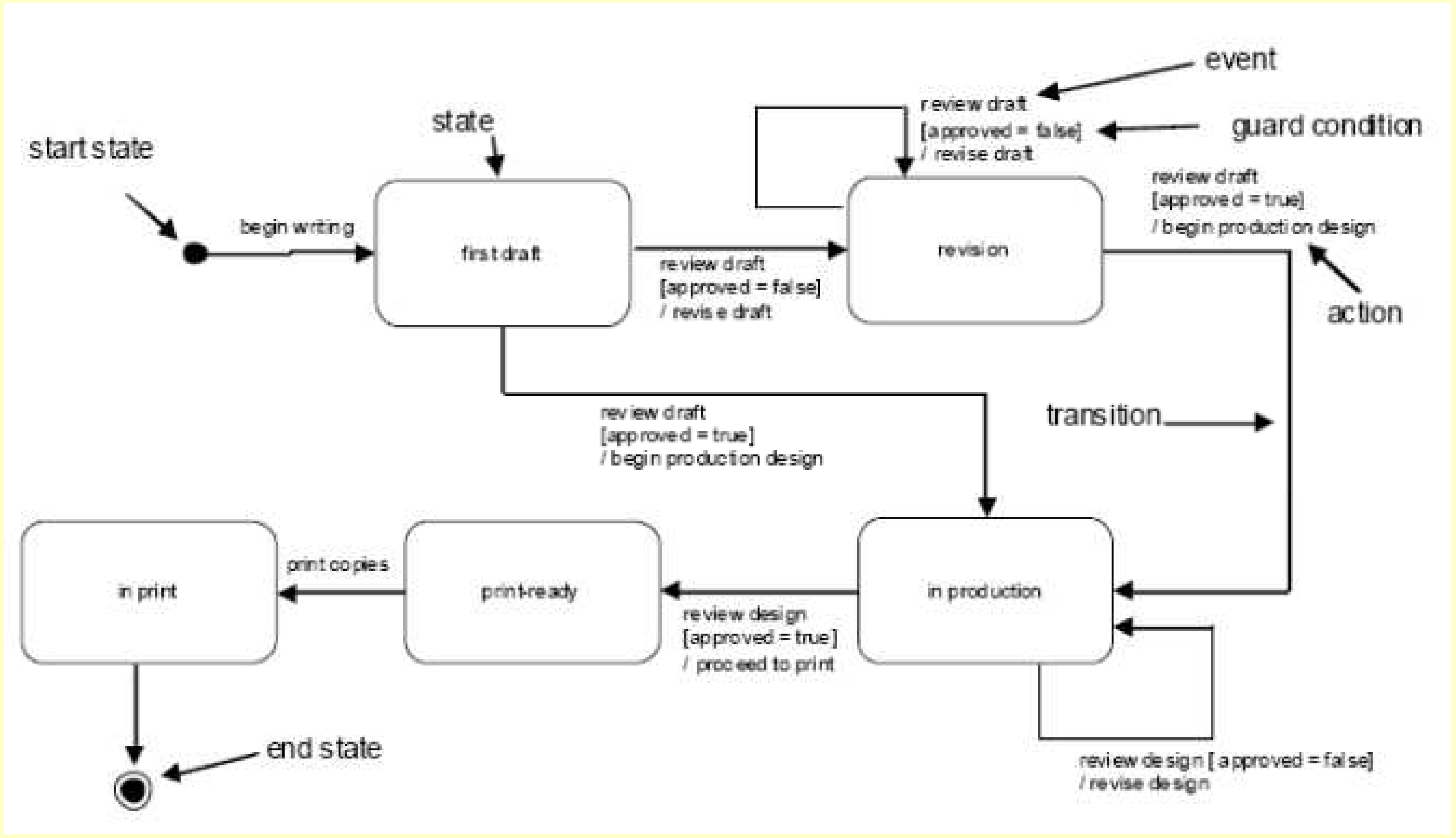
Activity diagrams or state diagrams?

An activity diagram describes the flow of a business process or program code.



Activity diagrams or state diagrams?

State Transition diagrams model object lifecycles and event-driven processes



State (machine) diagrams

State machine diagrams depict the various states that an object may be in and the transitions between those states.

UML 2 state machine diagrams were formerly called state chart diagrams in UML 1.x

In fact, in other modeling languages, it is common for this type of a diagram to be called a state-transition diagram or even simply a state diagram.

A state represents a stage in the behavior pattern of an object, and like UML activity diagrams it is possible to have initial states and final states.

An initial state, also called a creation state, is the one that an object is in when it is first created, whereas a final state is one in which no transitions lead out of.

A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object.

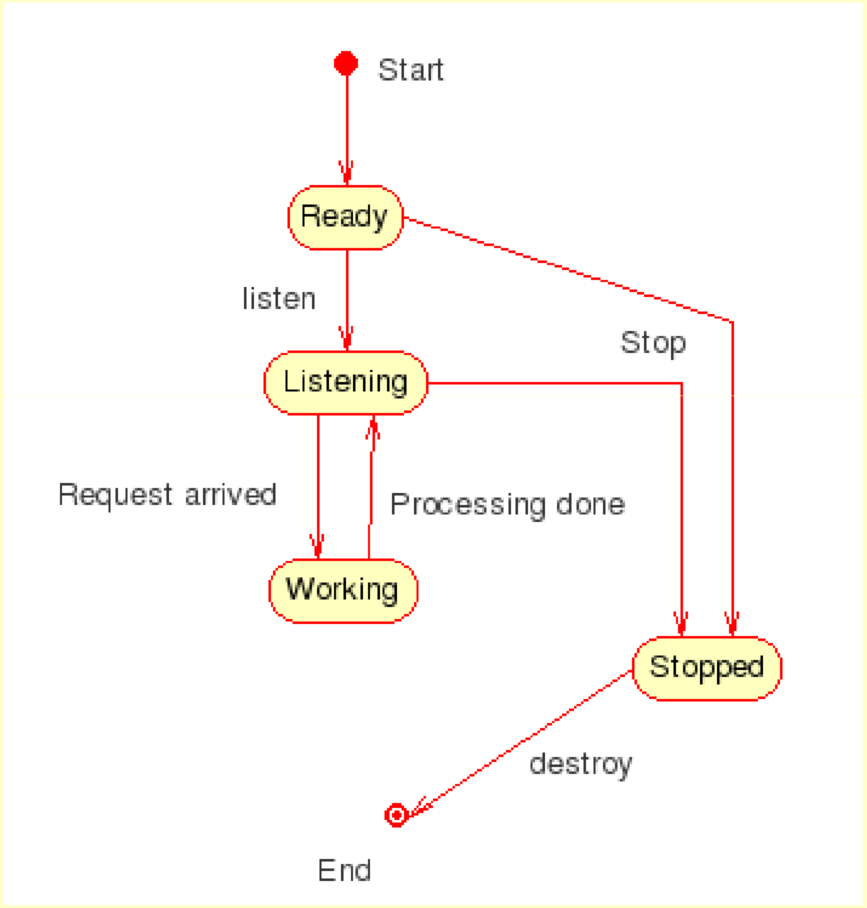
<u>State (machine) diagrams</u>

A UML State describes the internal state of an object of one particular class.

Note that not every change in one of the attributes of an object should be represented by a State but only those changes that can significantly affect the workings of the object.

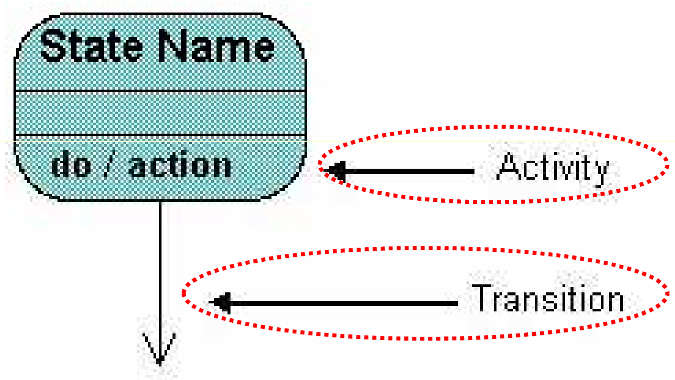
There are two special types of States: Start and End. They are special in that there is no event that can cause an Object to return to its Start state, in the same way as there is no event that can possibly take an Object out of its End state once it has reached it.

Server Example:



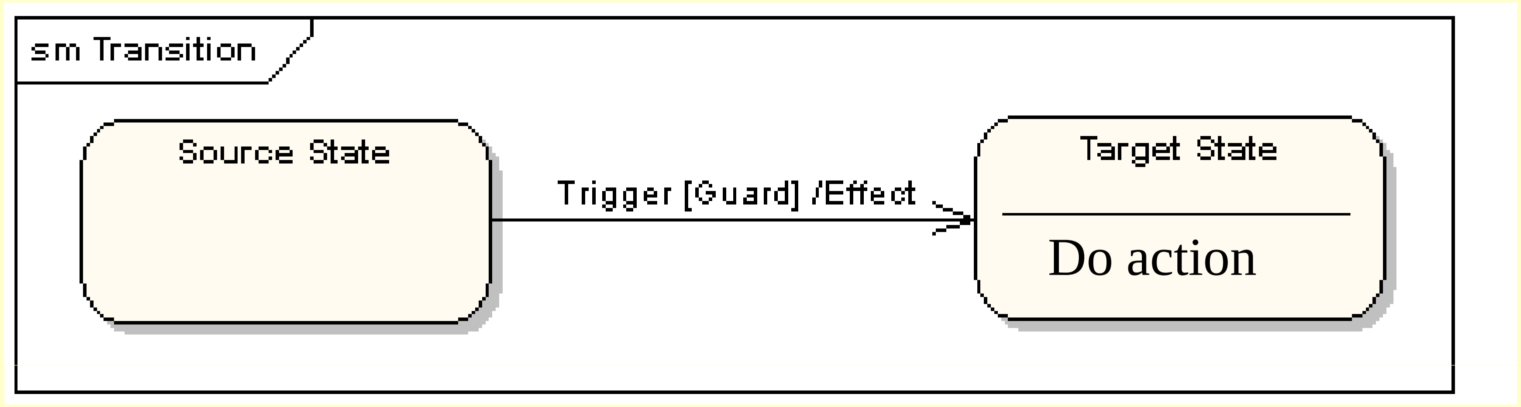
State (machine) diagrams: how to draw

State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state.



State (machine) diagrams: how to draw

Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect,



"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time.

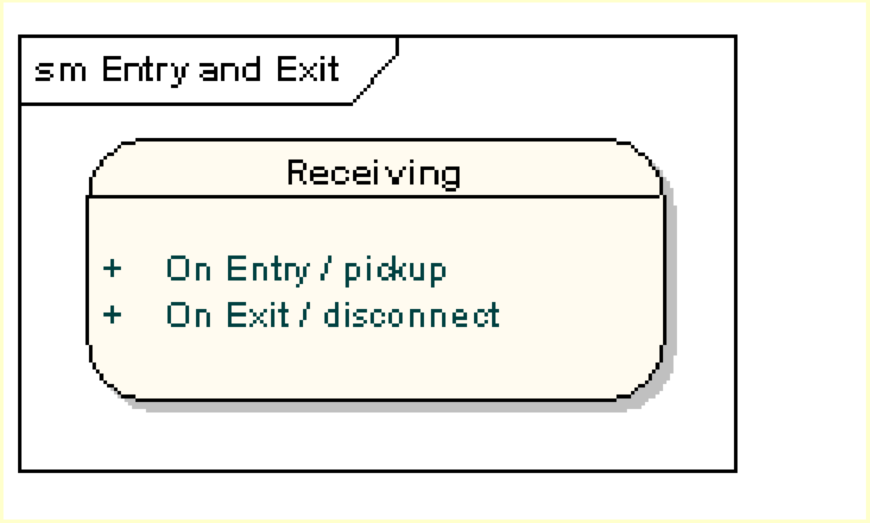
"Guard" is a condition which must be true in order for the trigger to cause the transition.

"Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

State (machine) diagrams: how to draw

State Actions

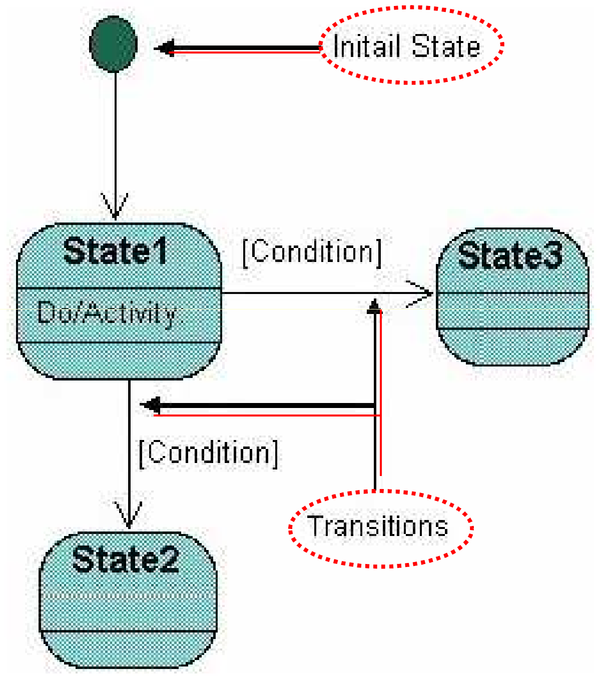
Typically, an effect is associated with a transition. If a target state has many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action.



It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of actions of each type.

State (machine) diagrams: how to draw

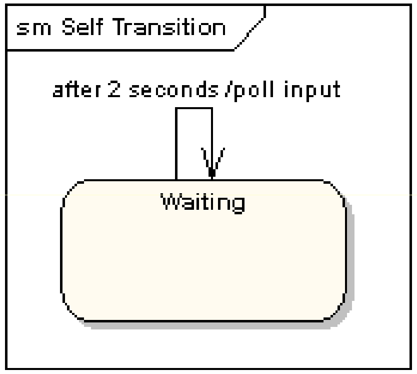
All state diagrams being with an initial state of the object. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.



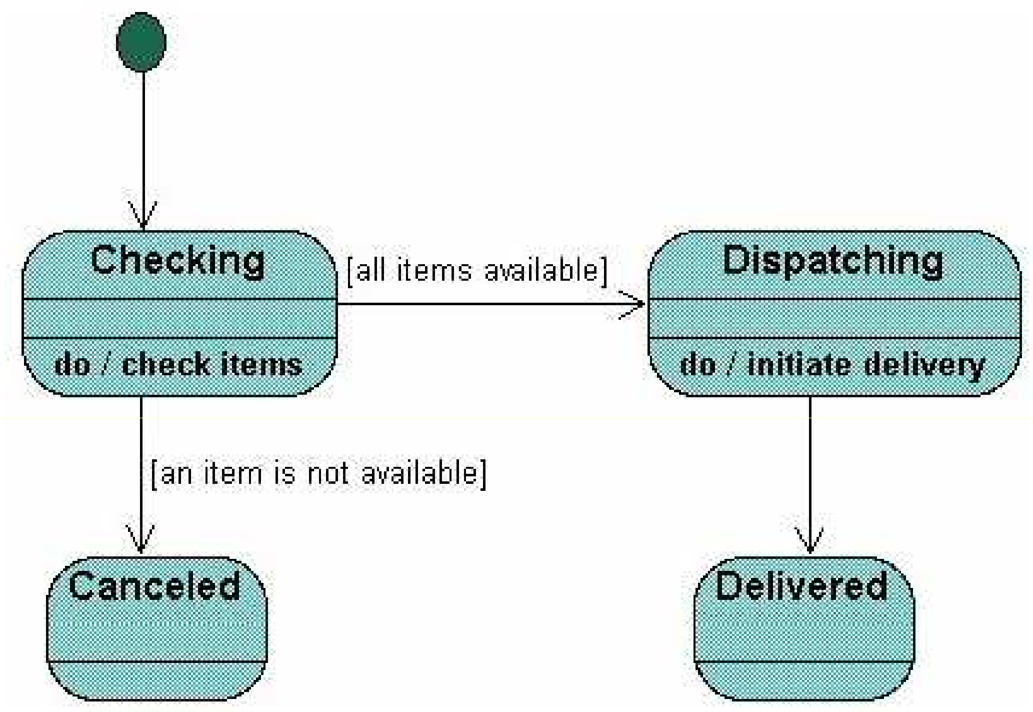
State (machine) diagrams: how to draw

Self-Transitions

A state can have a transition that returns to itself, as in the following diagram. This is most useful when an effect is associated with the transition.



State (machine) diagrams: Order Object Example



When the object enters the Checking state it performs the activity "check items."

After the activity is completed the object transitions to the next state based on the conditions [all items available] or [an item is not available].

If an item is not available the order is canceled.

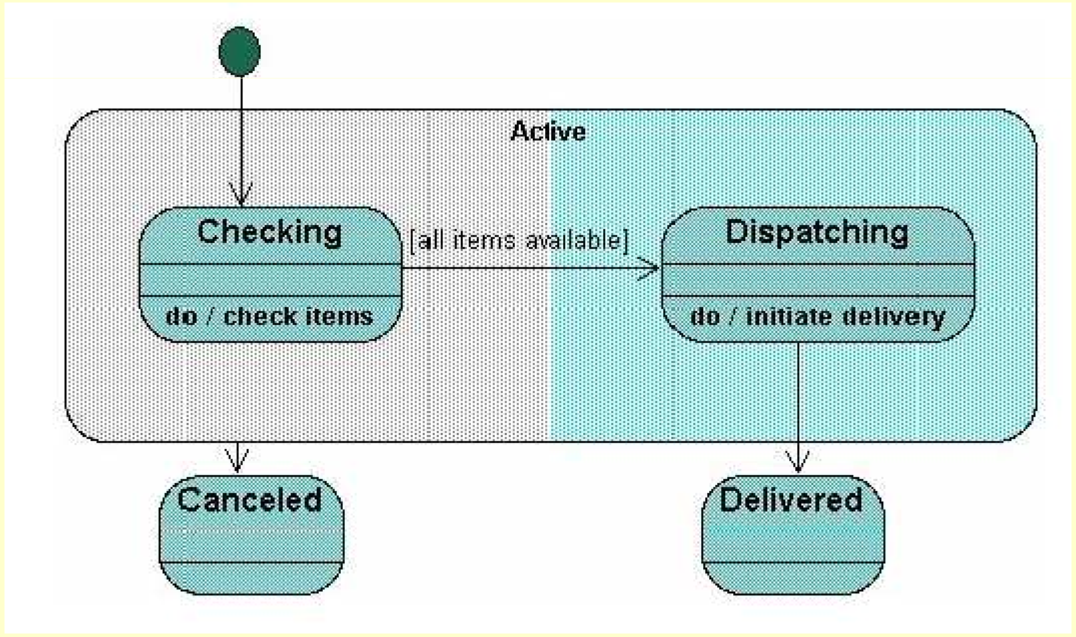
If all items are available then the order is dispatched.

When the object transitions to the Dispatching state the activity "initiate delivery" is performed.

After this activity is complete the object transitions again to the Delivered state.

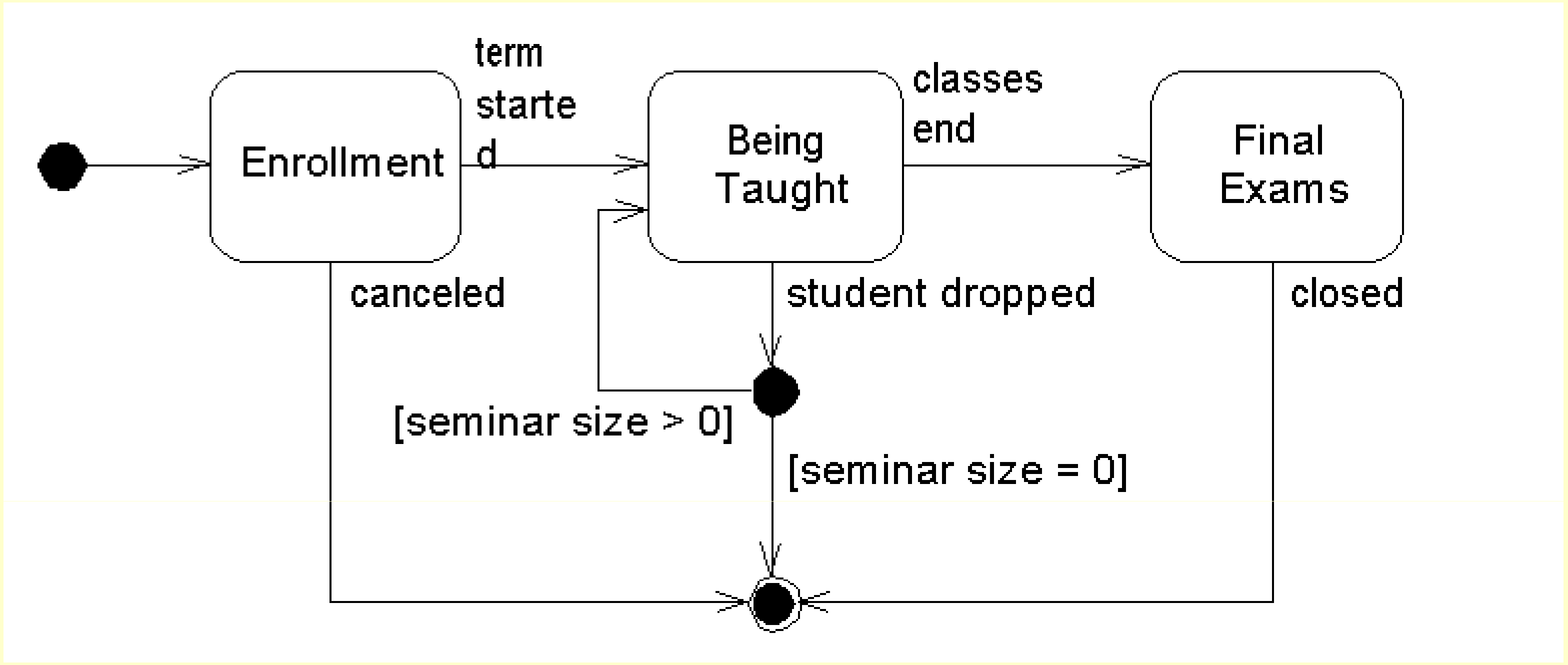
State (machine) diagrams: super states

State diagrams can also show a super-state for the object. A super-state is used when many transitions lead to the a certain state. Instead of showing all of the transitions from each state to the redundant state a super-state can be used to show that all of the states inside of the super-state can transition to the redundant state. This helps make the state diagram easier to read.



Here, both checking and dispatching can lead directly to canceled

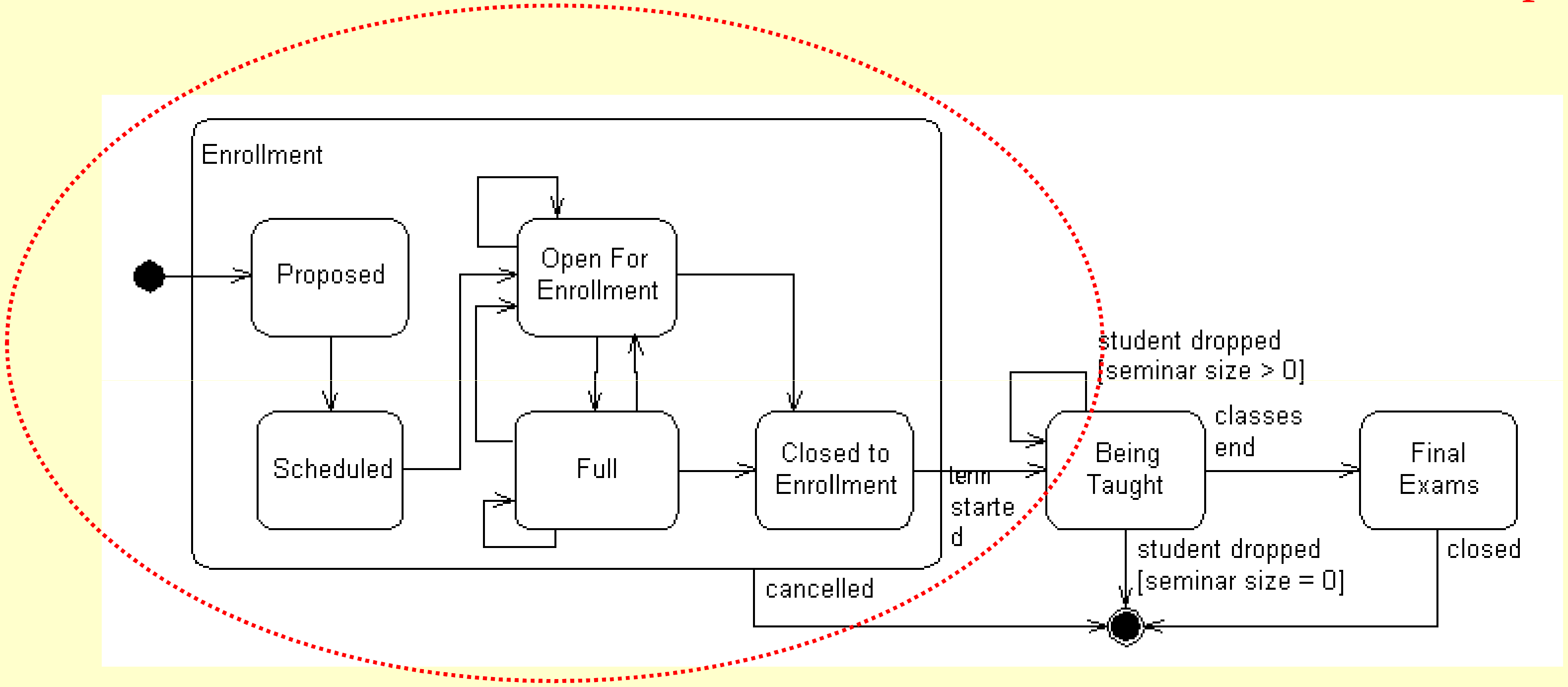
State machine diagram composition



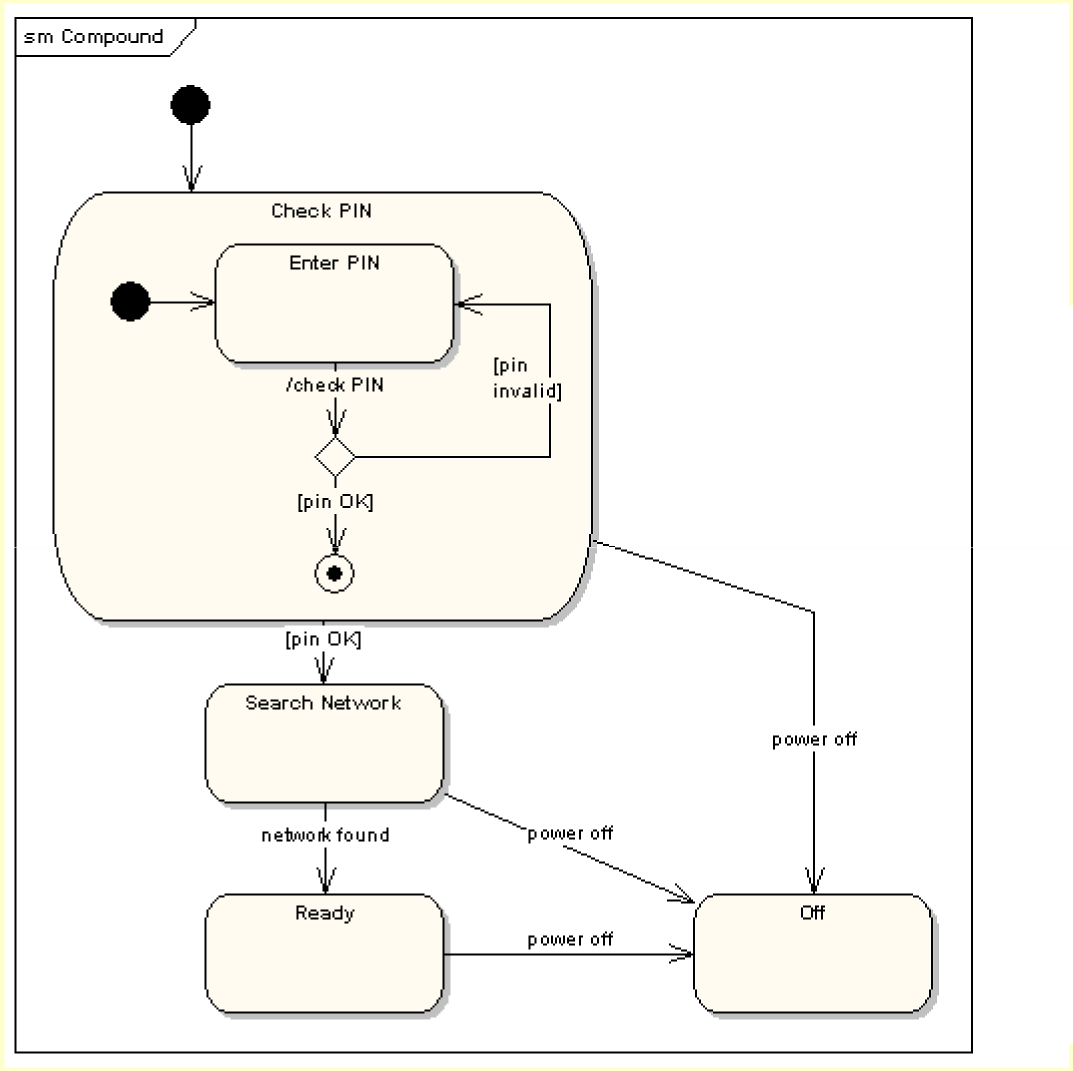
We wish to Compose the Enrollment state (activity) into substates with more detail

State machine diagram composition

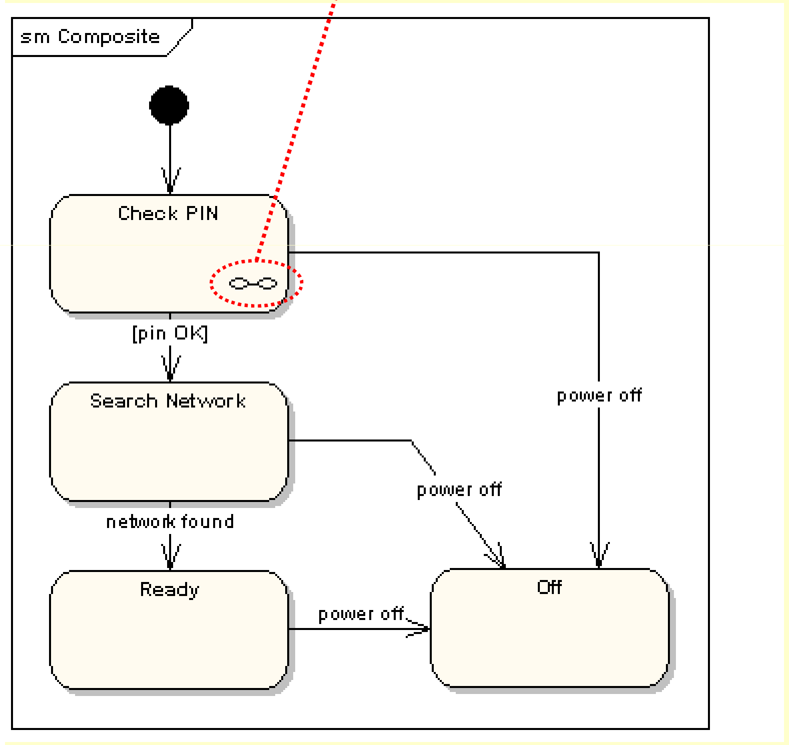
Enrollment has been further decomposed



Composition: alternative ways of viewing compound states

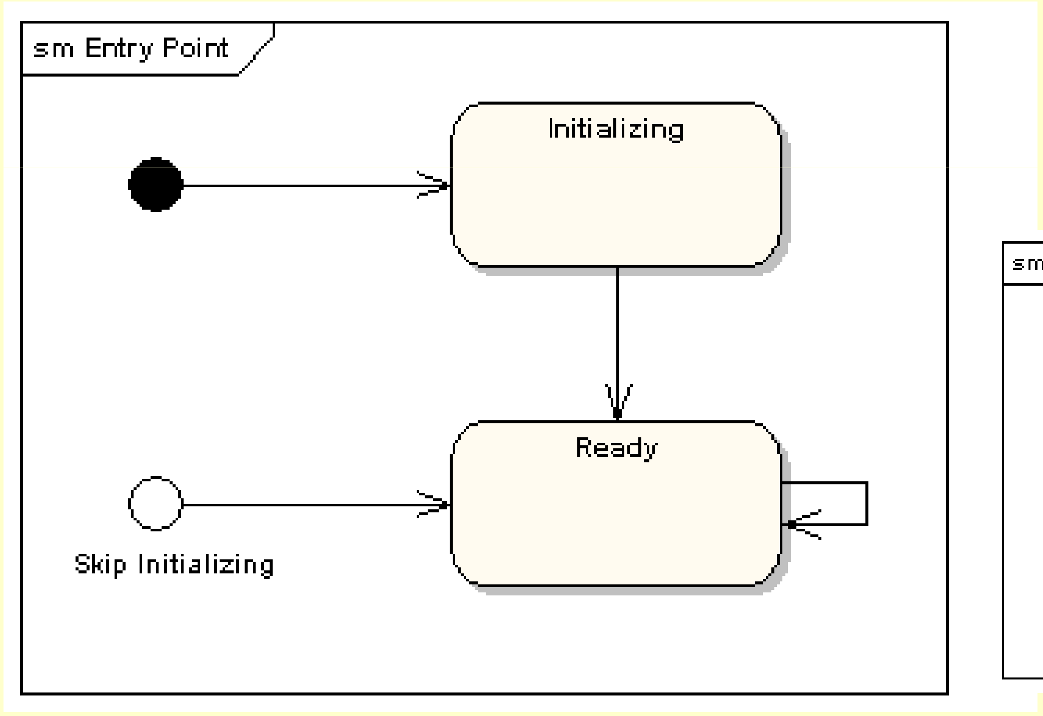


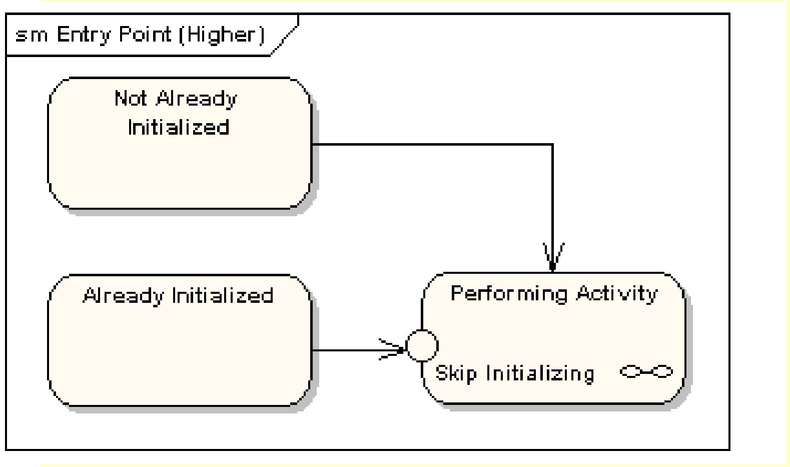
Details of the Check PIN sub-machine are shown in a separate diagram.



Composition: entry points

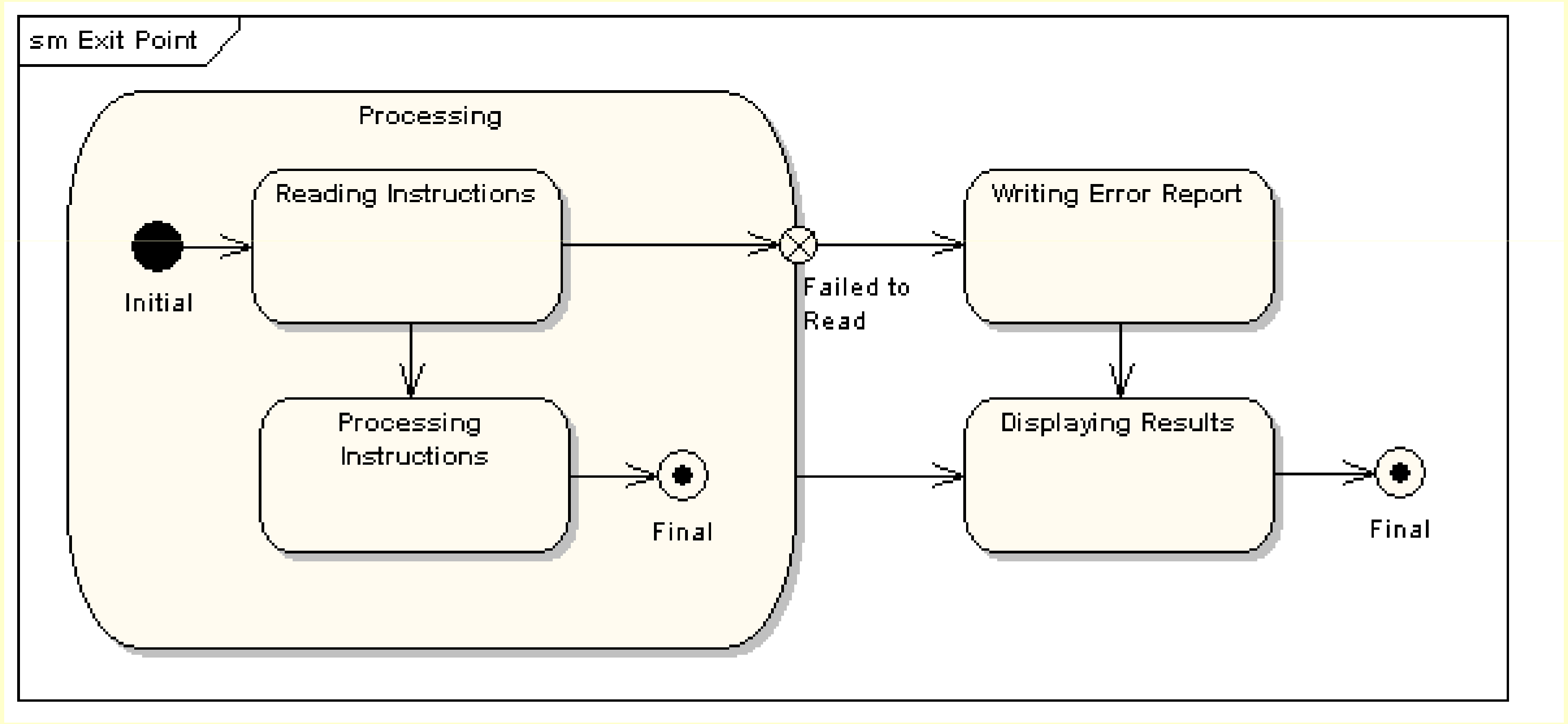
Sometimes you won’t want to enter a sub-machine at the normal initial state. For example, in the following sub-machine it would be normal to begin in the "Initializing" state, but if for some reason it wasn’t necessary to perform the initialization, it would be possible to begin in the "Ready" state by transitioning to the named entry point.





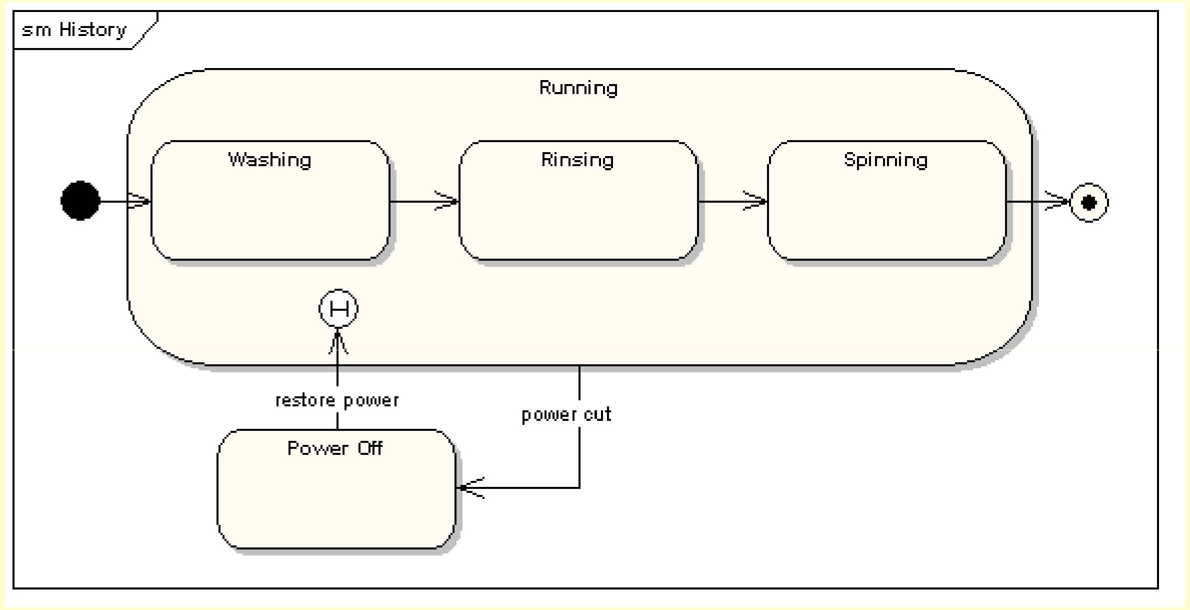
Composition: exit points

In a similar manner to entry points, it is possible to have named alternative exit points. The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.



Composition: history states

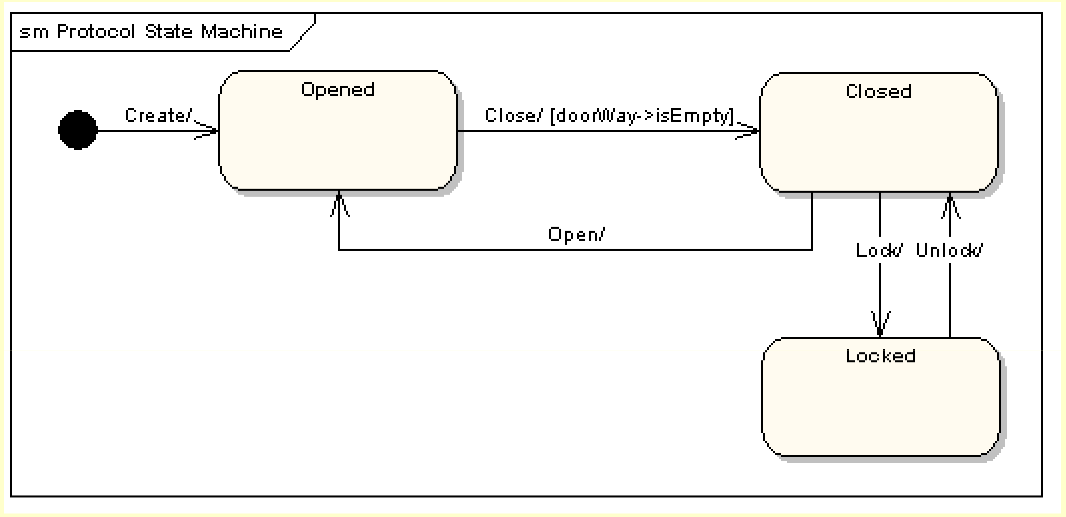
A history state is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.



In this state machine, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning". If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

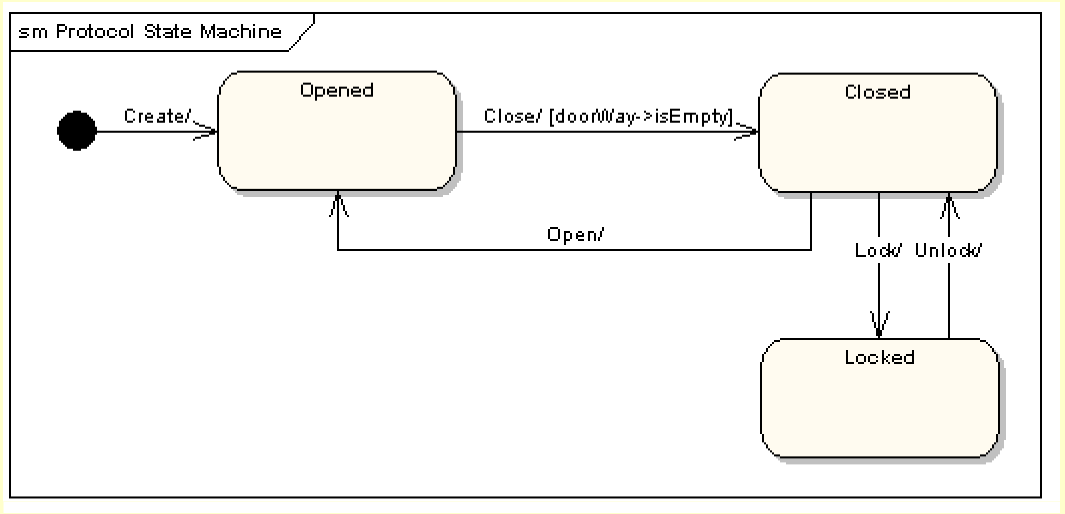
State machine diagram composition: not just compound (sub)states?

Consider a simple diagram for a door



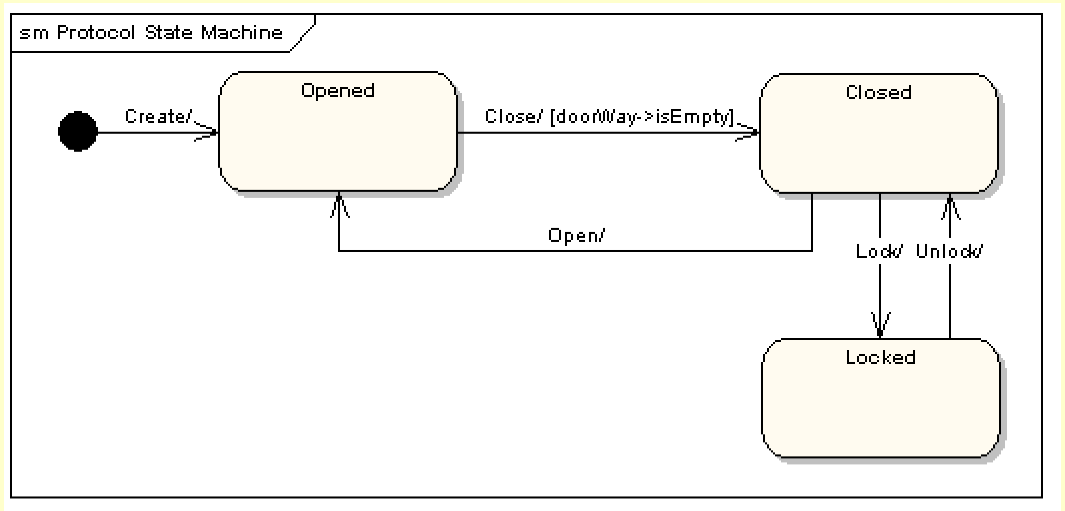
How could we compose an object from 2 doors?

State machine diagram composition



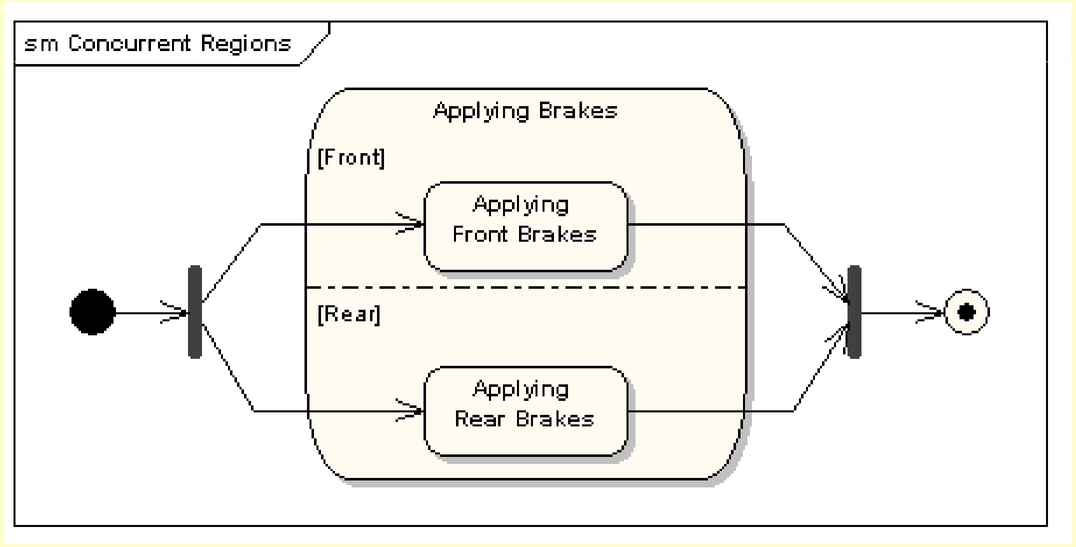
+ ??

How to compose in parallel?



State Machine: Concurrent Regions

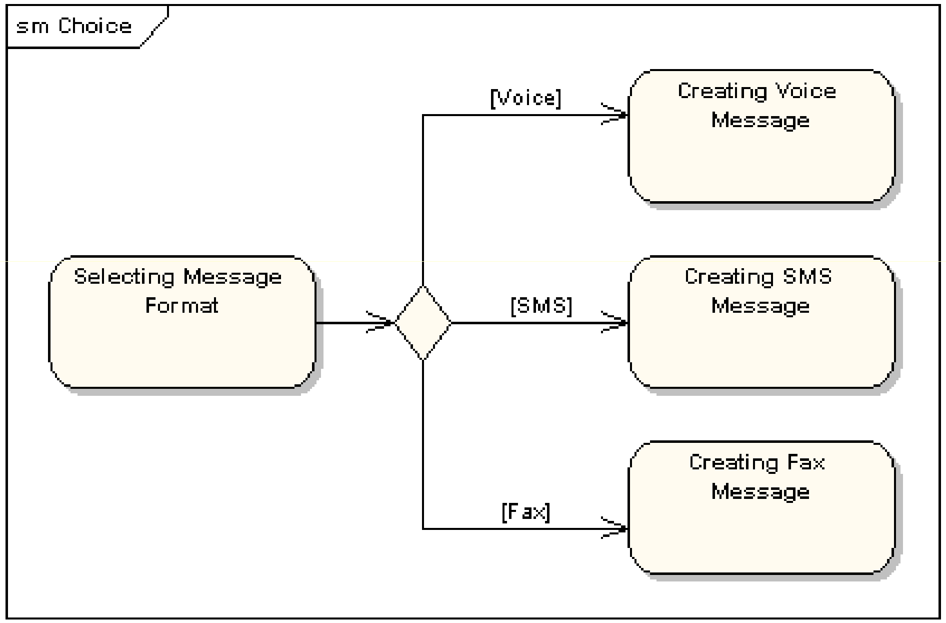
A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.



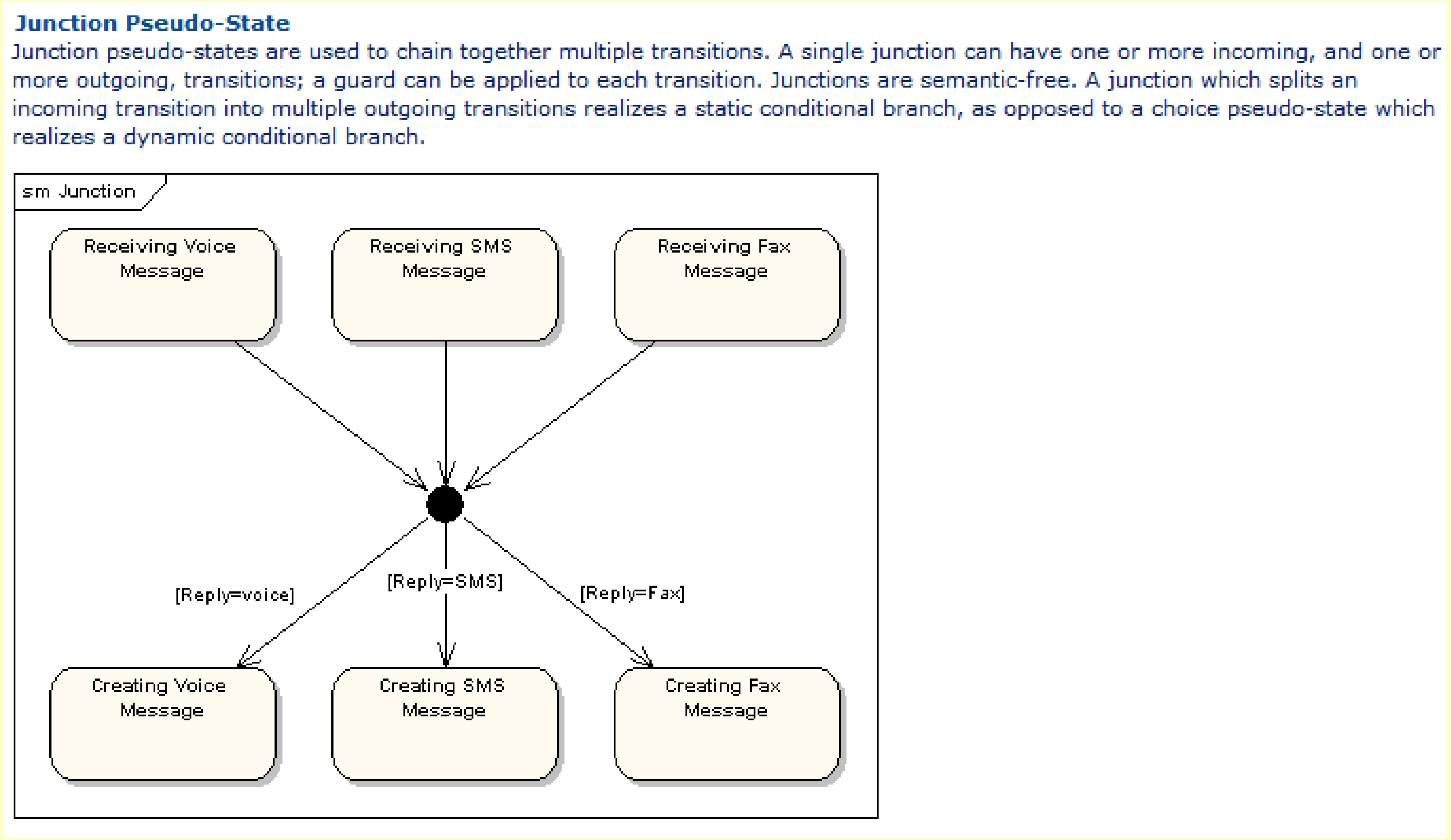
State Machine: Pseudo states

Choice Pseudo-State

A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving. The following diagram shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.



State Machine: Pseudo states



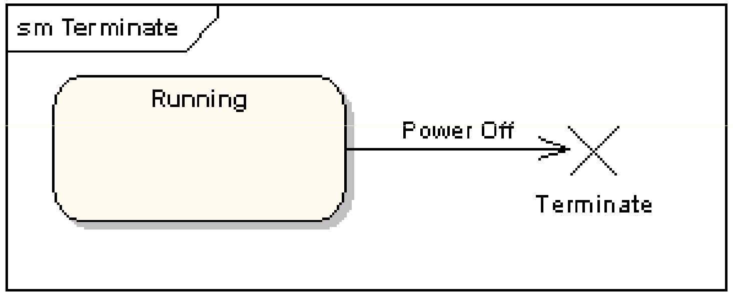
Junction Pseudo-State

Junction pseudo-states are used to chain together multiple transitions. A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition. Junctions are semantic-free. A junction which splits an incoming transition into multiple outgoing transitions realizes a static conditional branch, as opposed to a choice pseudo-state which realizes a dynamic conditional branch.

State Machine: Pseudo states

Terminate Pseudo-State

Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is notated as a cross.



State Machine: More Lift Design

Draw a state machine diagram for a lift showing how it decides whether to stay at its current floor, move up or move down.

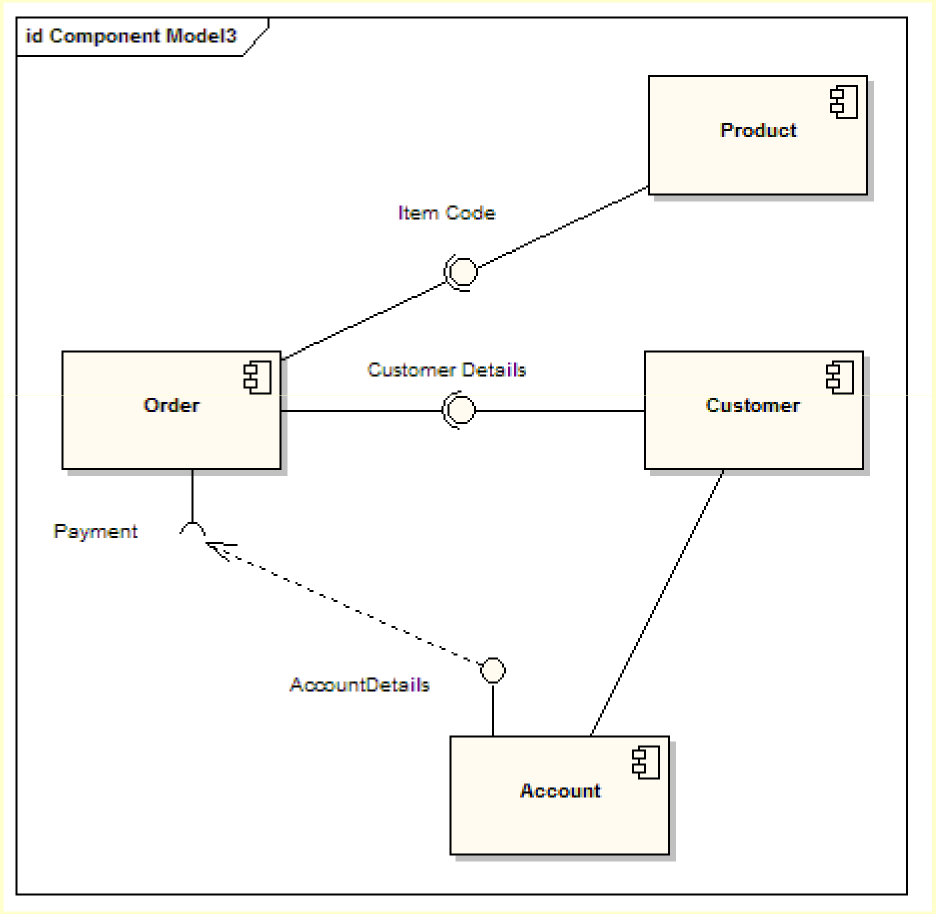
Component Diagrams

Component diagrams illustrate the pieces of software, embedded controllers, etc., that will make up a system. A component diagram has a higher level of abstraction than a Class Diagram - usually a component is implemented by one or more classes (or objects) at runtime. They are building blocks so a component can eventually encompass a large portion of a system.

Components are similar in practice to package diagrams, as they define boundaries and are used to group elements into logical structures. The difference between package diagrams and component diagrams is that Component Diagrams offer a more semantically rich grouping mechanism. With component diagrams all of the model elements are private, whereas package diagrams only display public items.

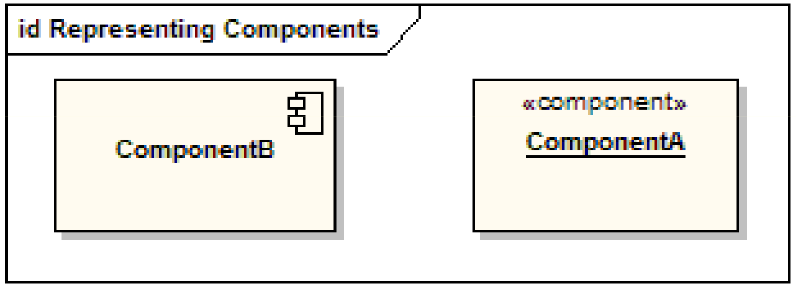
Component Diagrams: order example

The diagram demonstrates some components and their inter-relationships. Assembly connectors "link" the provided interfaces supplied by "Product" and "Customer" to the required interfaces specified by "Order". A dependency relationship maps a customer's associated account details to the required interface; "Payment", indicated by "Order".



Component Diagrams: notation

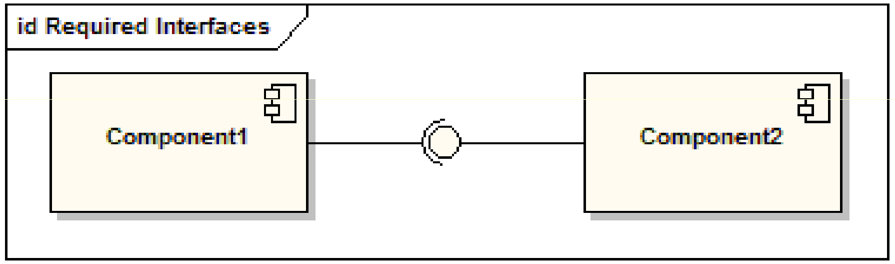
Components are represented as a rectangular classifier with the keyword «component»; optionally the component may be displayed as a rectangle with a component icon in the right-hand upper corner.



Component Diagrams: notation

Assembly Connector

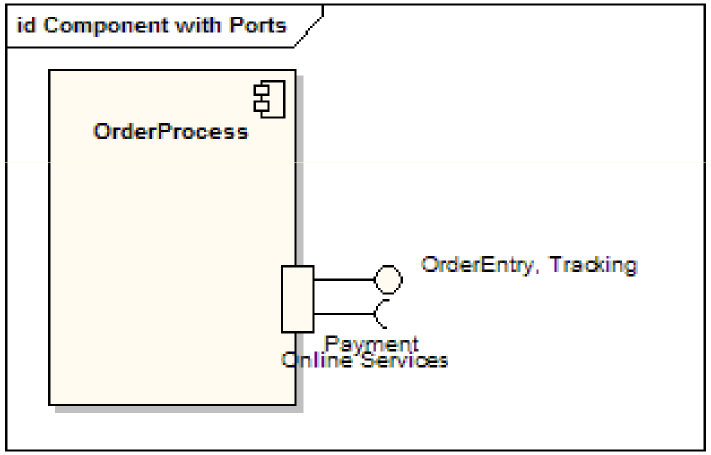
The assembly connector bridges a component's required interface (Component1) with the provided interface of another component (Component2); this allows one component to provide the services that another component requires.



Component Diagrams: notation

Components with Ports

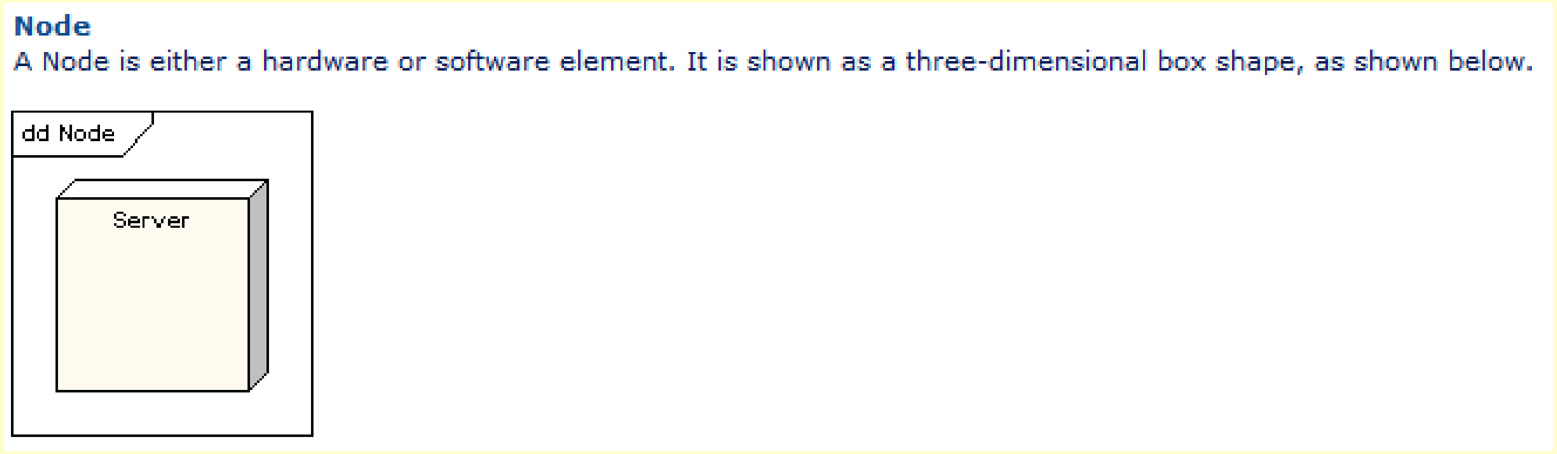
Using Ports with component diagrams allows for a service or behavior to be specified to its environment as well as a service or behavior that a component requires. Ports may specify inputs and outputs as they can operate bi-directionally. The following diagram details a component with a port for online services along with two provided interfaces order entry and tracking as well as a required interface payment.



Deployment Diagrams

Deployment diagrams show the physical disposition of significant artefacts within a real-world setting.

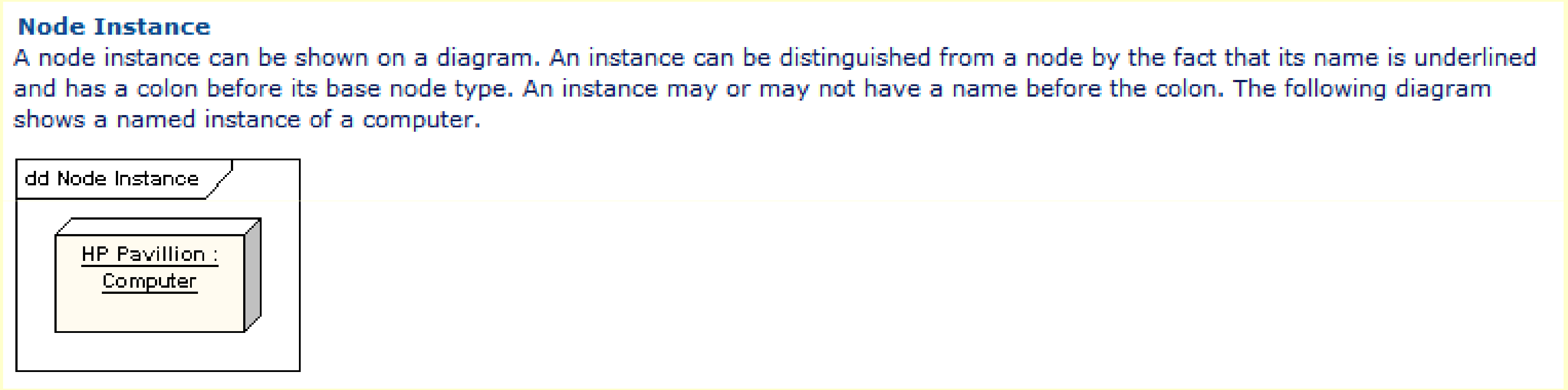
A deployment diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artefacts are mapped onto those nodes.



Node

A Node is either a hardware or software element. It is shown as a three-dimensional box shape, as shown below.

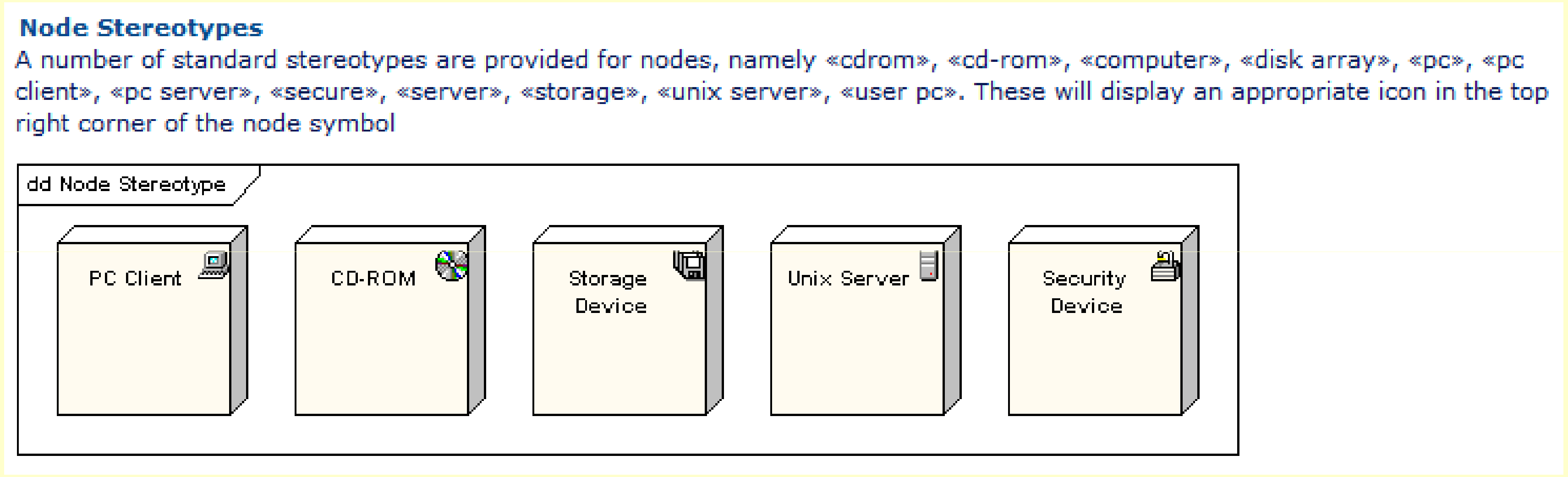
Deployment Diagrams



Node Instance

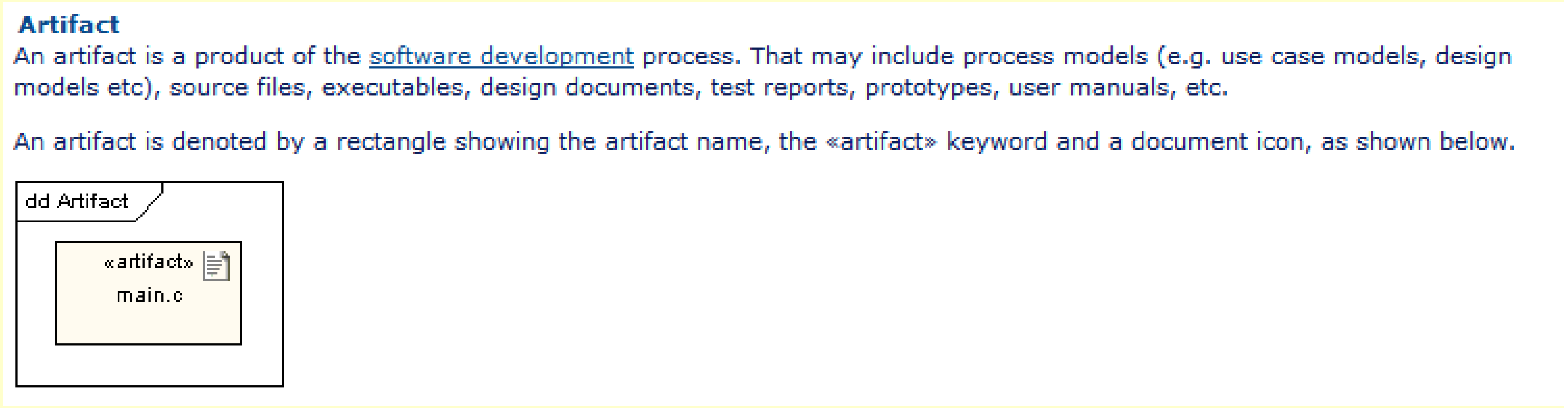
A node instance can be shown on a diagram. An instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon. The following diagram shows a named instance of a computer.

Deployment Diagrams


Node Stereotypes

A number of standard stereotypes are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc». These will display an appropriate icon in the top right corner of the node symbol

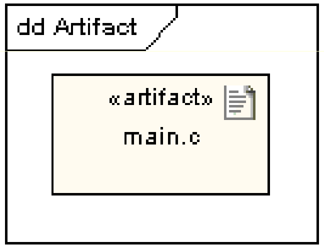
Deployment Diagrams



Artifact

An artifact is a product of the software development process. That may include process models (e.g. use case models, design models etc), source files, executables, design documents, test reports, prototypes, user manuals, etc.

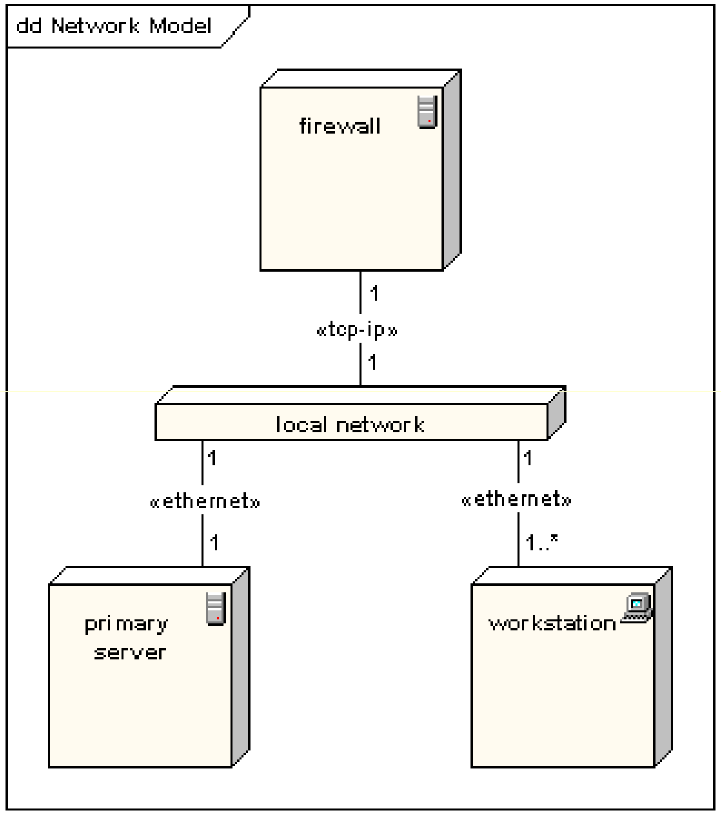
An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword and a document icon, as shown below.



Deployment Diagrams

Association

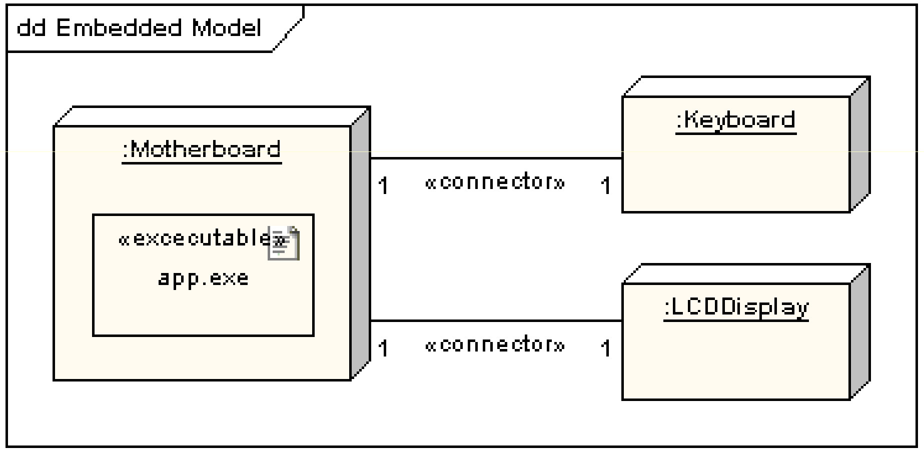
In the context of a deployment diagram, an association represents a communication path between nodes. The following diagram shows a deployment diagram for a network, depicting network protocols as stereotypes, and multiplicities at the association ends.



Deployment Diagrams

Node as Container

A node can contain other elements, such as components or artifacts. The following diagram shows a deployment diagram for part of an embedded system, depicting an executable artifact as being contained by the motherboard node.



Package Diagrams

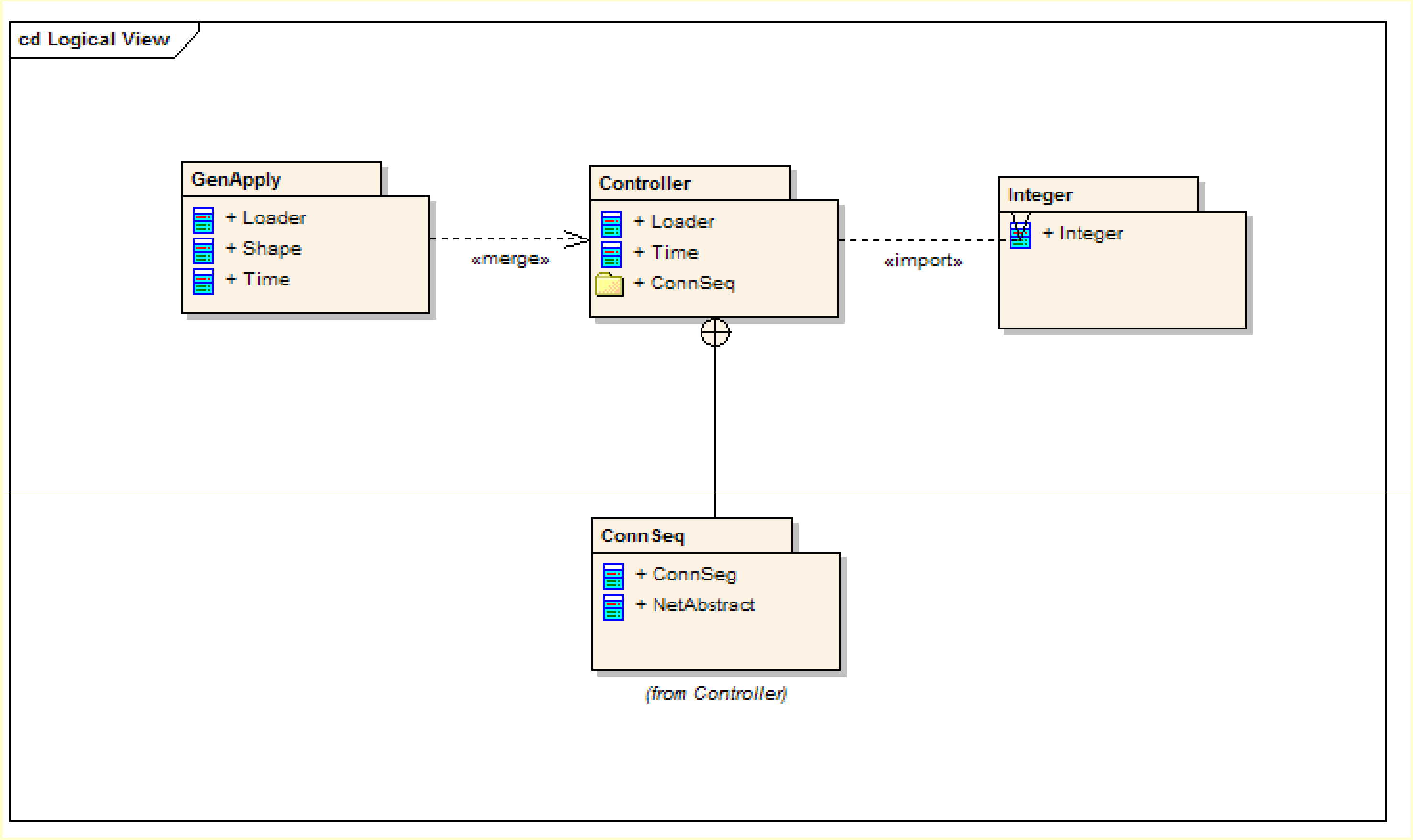
Package diagrams are used to divide the model into logical containers, or 'packages', and describe the interactions between them at a high level.

Package diagrams are used to reflect the organization of packages and their elements.

When used to represent class elements, package diagrams provide a visualization of the namespaces.

The most common use for package diagrams is to organize use case diagrams and class diagrams, although the use of package diagrams is not limited to these UML elements.

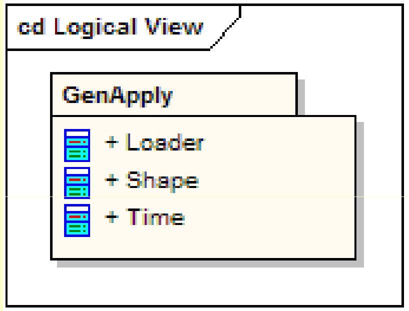
Package Diagrams: Example



Elements contained in a package share the same namespace. Therefore, the elements contained in a specific namespace must have unique names.

Package Diagrams: Notation

Packages can be built to represent either physical or logical relationships. When choosing to include classes in specific packages, it is useful to assign the classes with the same inheritance hierarchy to the same package. There is also a strong argument for including classes that are related via composition, and classes that collaborate with them, in the same package.



Packages are represented in UML 2.1 as folders and contain the elements that share a namespace; all elements within a package must be identifiable, and so have a unique name or type. The package must show the package name and can optionally show the elements within the package in extra compartments.

Package Diagrams: Notation

Package *Merge*
A «merge» connector between two packages defines an implicit generalization between elements in the source package, and elements with the same name in the target package. The source element definitions are expanded to include the element definitions contained in the target. The target element definitions are unaffected, as are the definitions of source package elements that don’t match names with any element in the target package.
Package *Import*
The «import» connector indicates that the elements within the target package, which in this example is a single class, use unqualified names when being referred to from the source package. The source package's namespace gains access to the target classes; the target's namespace is not affected.
Nesting *Connectors*
The nesting connector between the target package and source packages shows that the source package is fully contained in the target package.