Unified Modelling Language (UML)

Marek Rychlý
rychly@fit.vutbr.cz

Brno University of Technology
Faculty of Information Technology
Department of Information Systems

Information Systems Analysis and Design (AIS)
10 October 2019

Outline

UML Class Diagram

- Visualisation of classes and interfaces.
  (including their internal structure and relationships to other classes/interfaces)
- Class is a set of objects sharing a common structure&behaviour.
  (i.e., the set of objects possibly differing in state but not behaviour)
- Features of the classes: attributes and operations.

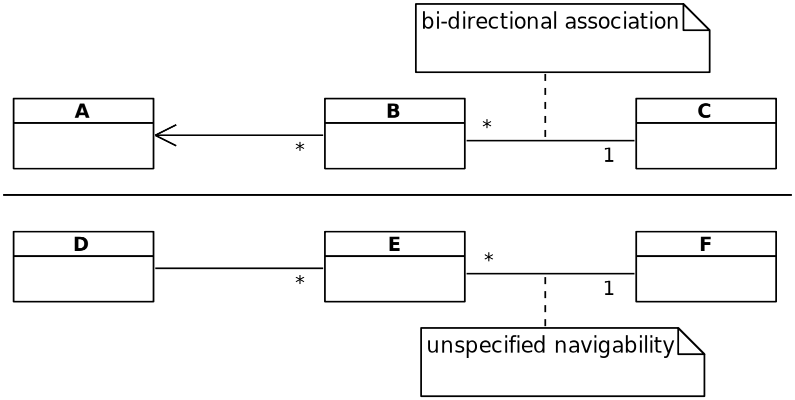
attribute	is a representation of an object's internal state (i.e., what the objects have)
operation	is a specification of behaviour of the object (i.e., what the objects do)
method	is an implementation of the operation (i.e., how the objects do the operations)

Relationships of Classes in UML

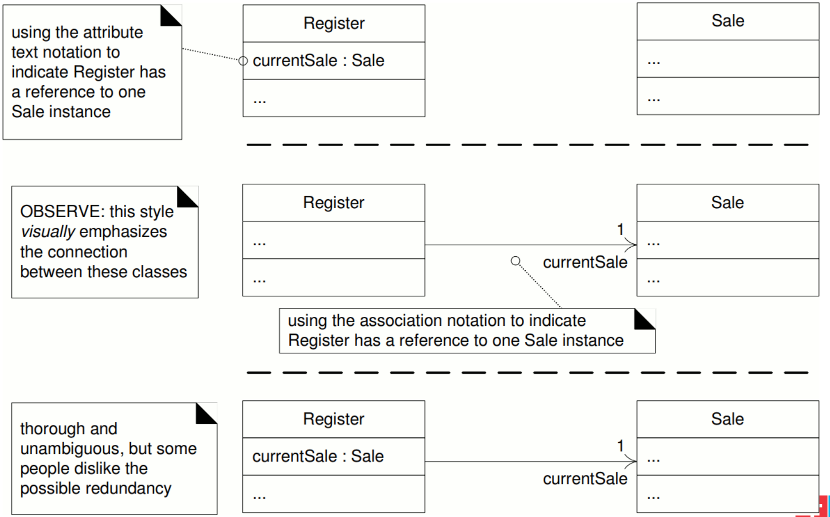
- Association is described by participants and
  - a single name of the relationship
    (it can be unspecified)
  - roles of participants
    (they can be unspecified)
  - multiplicity of participants
    (one of 0, 0..$*$, 0..1, 1, 1..$*$, and $*$, or unspecified, for each participant)
  - navigability to participants
    (true "$\rightarrow$", false "$\times$", or unspecified for each participant)
- There are no implicit values for missing multiplicity/navigability.
  (unspecified multiplicity is not "1", navigability is "unspecified" or "bi-directional")
- Two special types of associations:
  (they are forms of containment, for parent-child relationships)

aggregation	for modelling of a separable part of a whole (the child can exist independently of the parent in this case)
composition	for modelling a non-separable part of the whole (the child cannot exist independently of the parent in this case)

Different Semantics of Navigability



Association and Attribute


using the attribute text notation to indicate Register has a reference to one Sale instance
Register
currentSale : Sale
...
Sale
...
...
OBSERVE: this style *visually* emphasizes the connection between these classes
Register
...
...
Sale
...
...
1
currentSale
using the association notation to indicate Register has a reference to one Sale instance
thorough and unambiguous, but some people dislike the possible redundancy
Register
currentSale : Sale
...
Sale
...
...
1
currentSale

(adopted from “Applying UML and Patterns” by Craig Larman)

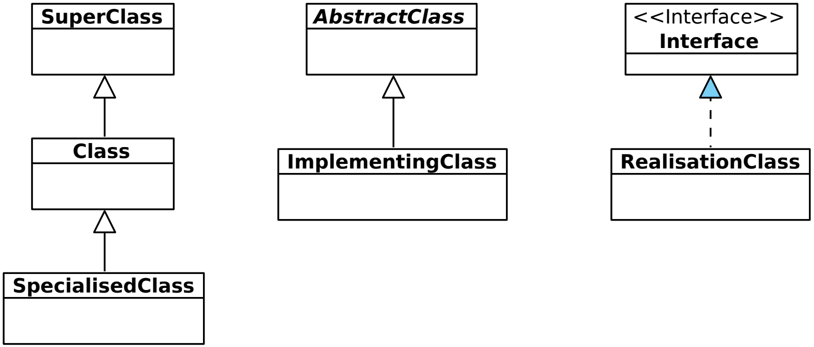
Generalisation/Specialisation

- The relationship between a general and a specific classifier.
  (instance of the specific classifier is also an indirect instance of the general one)

superclass	is the more general classifier
subclass	is the more specific classifier (it is a subtype)

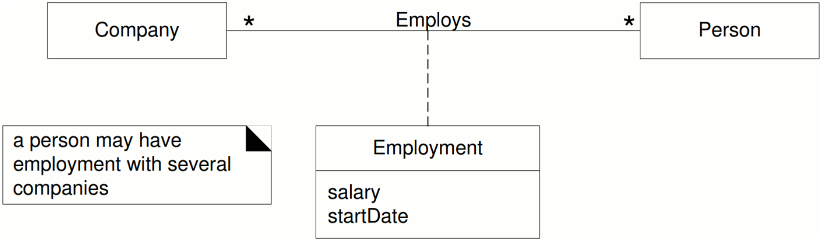
- Subtype Requirement, aka "Liskov substitution principle":
  Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T. *— Liskov/Wing (1994)*
- The generalisation with "Liskov substitution principle" means that
  - a subclass inherits attributes, operations, relationships, constraints
  - there is a strong dependency of a subclass to its superclass
- There are constraints in UML to describe "covering" and "disjoint"
  - {complete}, or {incomplete}, constraint
    (if every superclass is also an instance of at least one of its subclasses)
  - {disjoint}, or {overlapping}, constraint
    (if no instance of any subclass may also be an instance of another subclass)

Generalisation and Realisation



Association Class

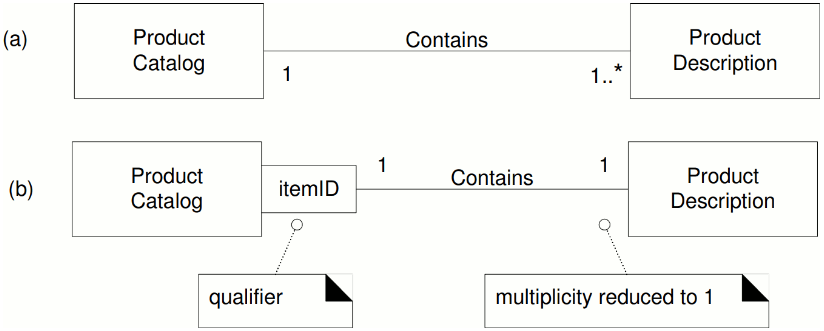
when an association is a class



(adopted from “Applying UML and Patterns” by Craig Larman)

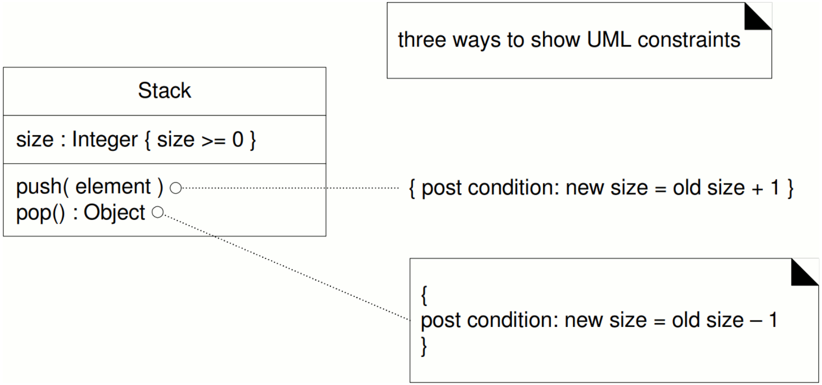
Qualified Association

when a qualifier/key reduces an association multiplicity (*a* and *b* model the same association)



(adopted from “Applying UML and Patterns” by Craig Larman)

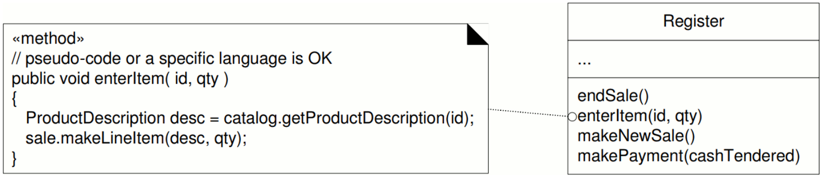
Constraints in Class Diagrams


three ways to show UML constraints
Stack
size : Integer { size >= 0 }
push(element)
pop() : Object
{ post condition: new size = old size + 1 }
{
post condition: new size = old size – 1
}

(adopted from “Applying UML and Patterns” by Craig Larman)

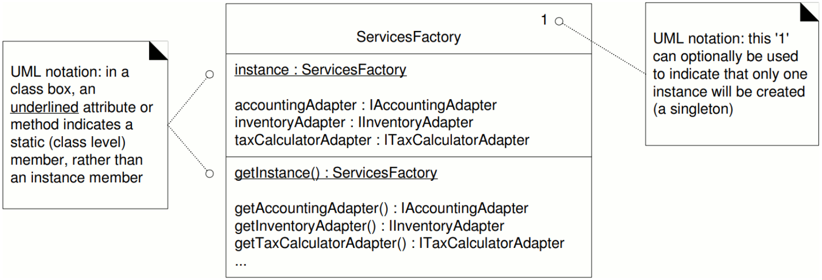
Annotations in Class Diagrams

to outline methods of classes


«method»
// pseudo-code or a specific language is OK
public void enterItem(id, qty)
{
ProductDescription desc = catalog.getProductDescription(id);
sale.makeLineItem(desc, qty);
}
Register
...
endSale()
enterItem(id, qty)
makeNewSale()
makePayment(cashTendered)

(adopted from “Applying UML and Patterns” by Craig Larman)

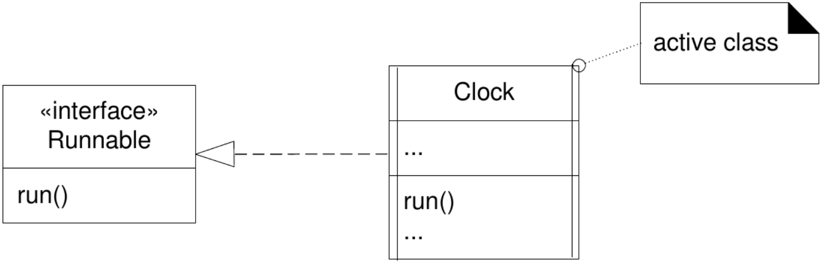
Singleton Classes in Class Diagrams


UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member
1
ServicesFactory
instance : ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getInstance() : ServicesFactory
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...
UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)

(adopted from “Applying UML and Patterns” by Craig Larman)

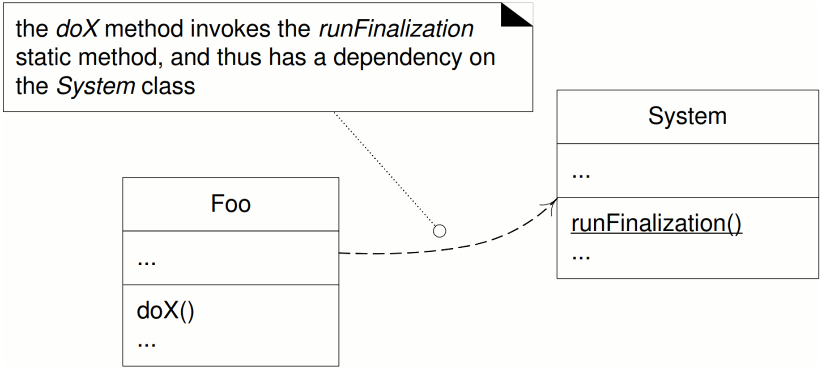
Active Classes in Class Diagrams

(Clock class is a realisation of Runnable interface, not its specialisation)



(adopted from "Applying UML and Patterns" by Craig Larman)

Dependencies in Class Diagrams

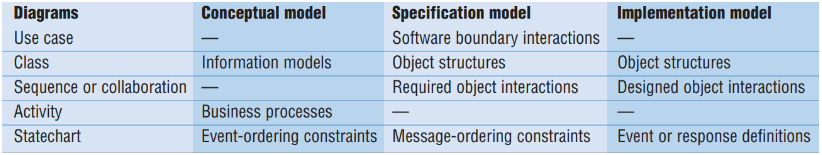

the *doX* method invokes the *runFinalization* static method, and thus has a dependency on the *System* class
System
...
runFinalization()
...
Foo
...
doX()
...

(adopted from “Applying UML and Patterns” by Craig Larman)

Different Purposes of Modelling

according to “Modeling with a sense of purpose” by John Daniels, 2002)

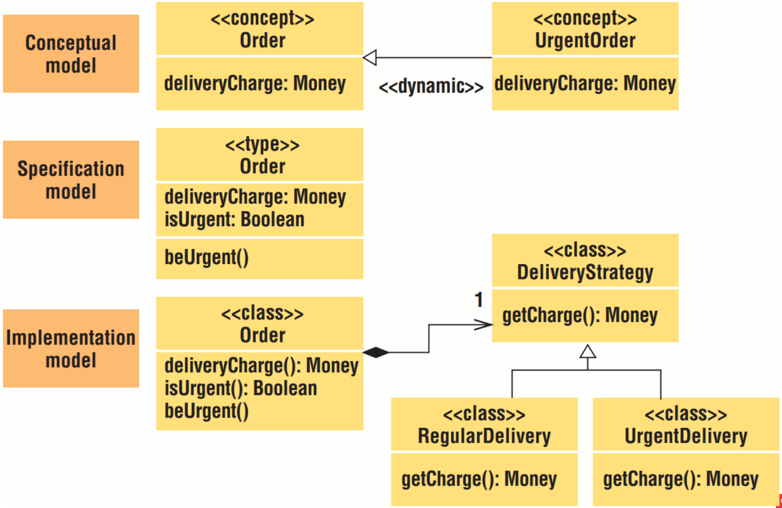
- **Conceptual** models describe a situation of interest in the world, such as a business operation or factory process. They say nothing about how much of the model is implemented or supported by software.
- **Specification models** define what a software system must do, the information it must hold, and the behaviour it must exhibit. They assume an ideal computing platform.
- **Implementation models** describe how the software is implemented, considering all the computing environment’s constraints and limitations.



Diagrams	**Conceptual model**	**Specification model**	**Implementation model**
Use case	—	Software boundary interactions	—
Class	Information models	Object structures	Object structures
Sequence or collaboration	—	Required object interactions	Designed object interactions
Activity	Business processes	—	—
Statechart	Event-ordering constraints	Message-ordering constraints	Event or response definitions

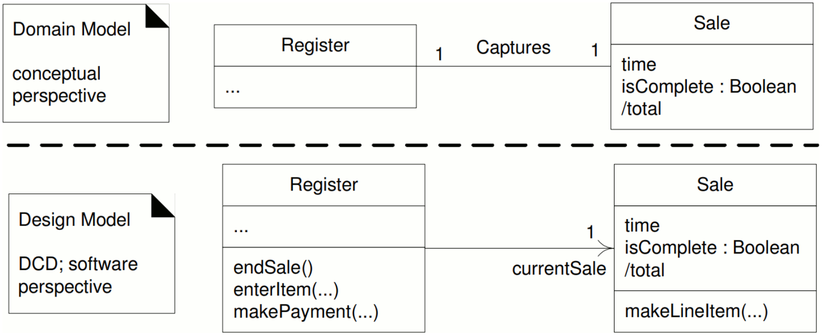
(adopted from “Modeling with a sense of purpose” by John Daniels, 2002)

One Notation, Three Models



(adopted from “Modeling with a sense of purpose” by John Daniels, 2002)

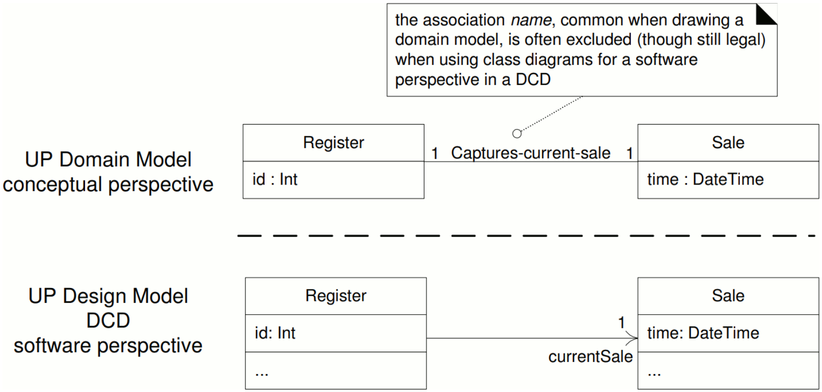
Domain Model and Design Model



(adopted from “Applying UML and Patterns” by Craig Larman)

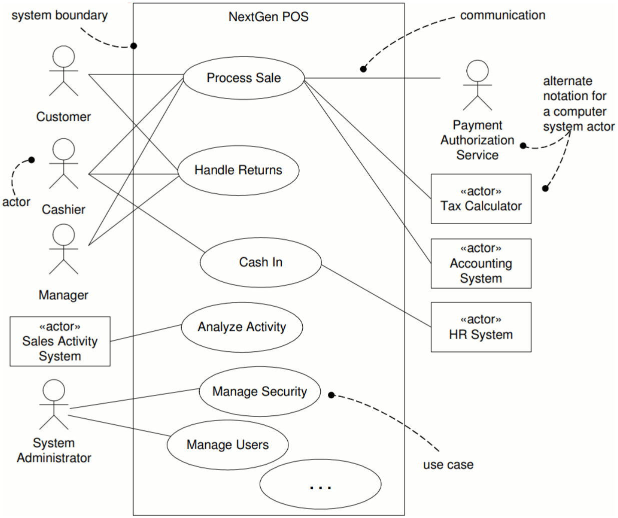
Domain Model and Design Model

and association names


the association *name*, common when drawing a domain model, is often excluded (though still legal) when using class diagrams for a software perspective in a DCD
UP Domain Model conceptual perspective
Register
id : Int
1 Captures-current-sale 1
Sale
time : DateTime
UP Design Model DCD software perspective
Register
id: Int
...
1
currentSale
Sale
time: DateTime
...

(adopted from “Applying UML and Patterns” by Craig Larman)

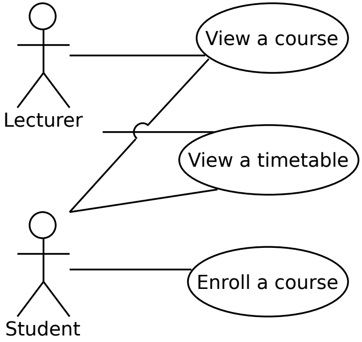
UML Use Case Diagram

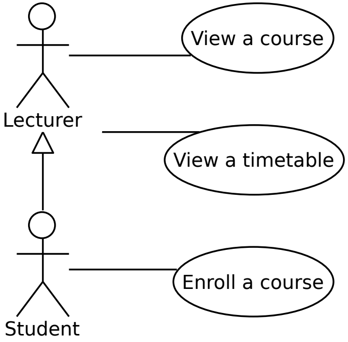


(adopted from “Applying UML and Patterns” by Craig Larman)

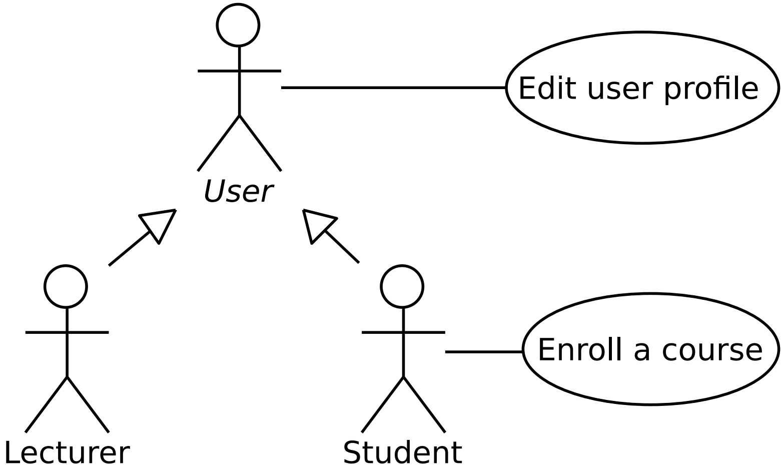
Generalisation/Specialisation of Actors

can be utilised only for identical use-cases



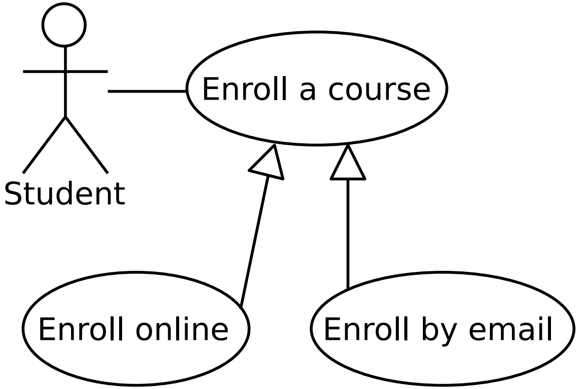


Abstract Actors



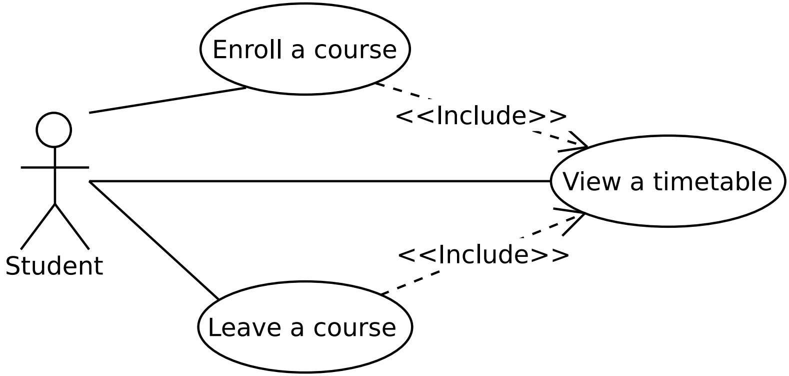
Generalisation/Specialisation of Use Cases

to inherit, extend, or modify a more general use case



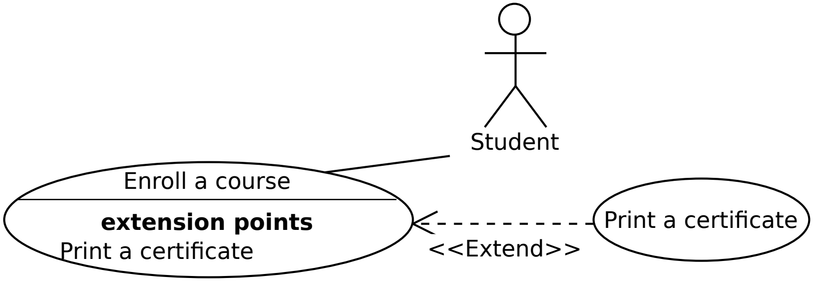
Include Relationship of Use Cases

(the included use-case is always performed in the including use-case)

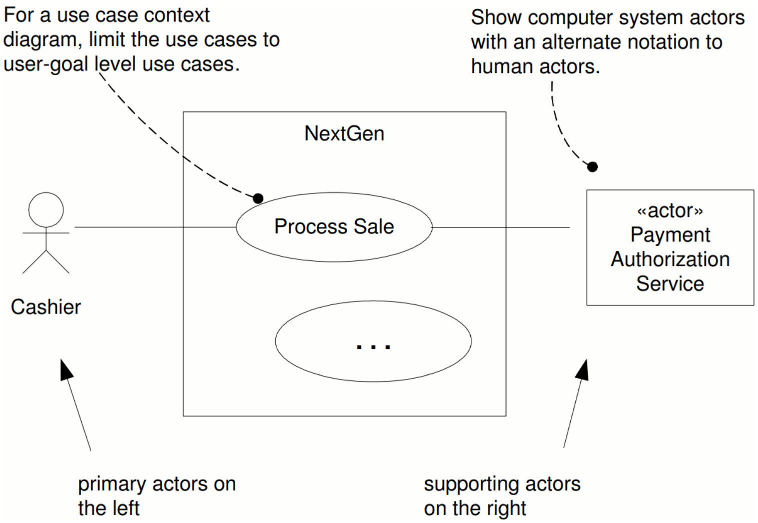


Extend Relationship of Use Cases

(the extended use-case may require extending use-case in the extension point)



Primary and Supporting/Secondary Actors


For a use case context diagram, limit the use cases to user-goal level use cases.
Show computer system actors with an alternate notation to human actors.
NextGen
Process Sale
...
Cashier
«actor» Payment Authorization Service
primary actors on the left
supporting actors on the right

(adopted from "Applying UML and Patterns" by Craig Larman)

Structured Use Cases

- UML lacks detail on the structure of use cases.
  (the use cases are just things in the Use Case diagram)
- It just recommends to describe use cases “in plain text, using operations, in activity diagrams, by a state-machine, or by other behaviour description techniques, such as pre-and post conditions; the interaction between the use case and the actors can also be presented in collaboration diagrams”.
- Typically, use cases are described in structured text, in a template.
  (see templates by Coleman at HP, by Oracle, and others)
- The template should provide description for:
  name, id, annotation, primary/secondary actors, pre-/post-conditions, scenario (steps), alternate flows, exception, issues, etc.

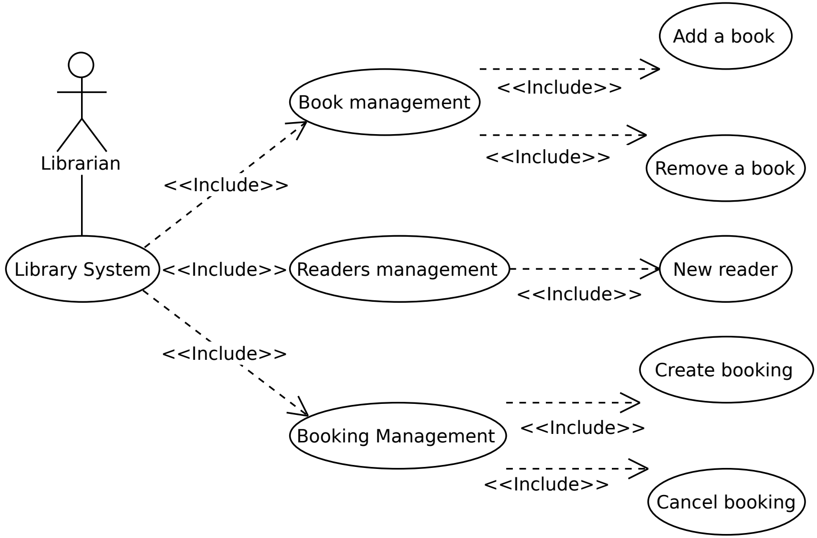
Use Cases in Practice

(observations and recommendations by Jim Arlow)

- After a brief introduction, users are able to understand use-cases.
- The understanding of actor generalisation is more difficult.
- Frequent “include” makes the understanding more difficult.
  (the users must keep in mind all including and included use-cases)
- “Extend” relationship is difficult to understand.
  (even after a thorough introduction and not only for users, also for developers)
- Specialisation of non-abstract use cases should not be used.
  (the specialised use cases are more difficult to understand; “extend” is better)

A use case diagram should be:
simple, with structured description, no more than two levels, no structural/functional decomposition (for the sake of analysis)

An Example of Improper Use Case Modelling



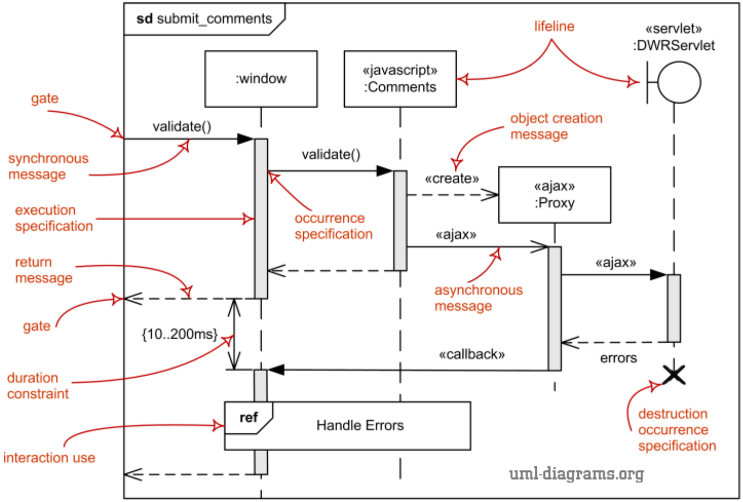
UML Interaction Diagrams

- One of main artefact of object-oriented analysis and design.
  (together with class diagrams that describe a static structure)
- An interaction of cooperating classes for a particular purpose.
- By four different diagrams/view in UML 2.
  (Sequence, Communication, Timing, and Interaction overview diagrams)

UML Sequence Diagram

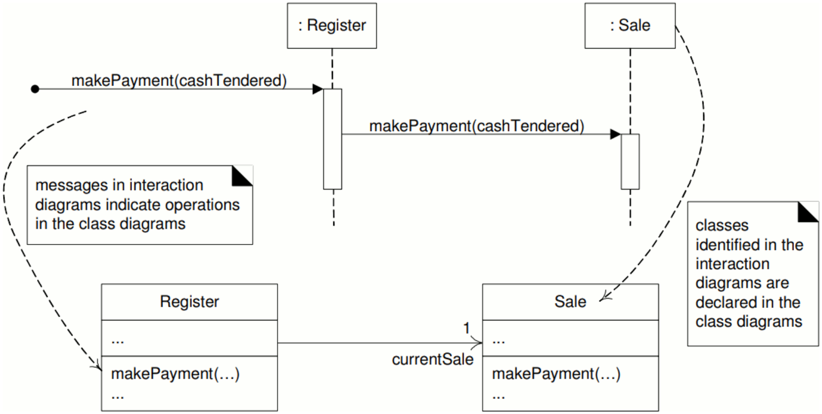
- Description of a sequence of message calls in two dimensions:
    1. **lifelines** of participant as classifier instances
       (objects, actors, classes implementing interfaces, components, etc.)
    2. **tail** of timeline where activities of participants occur
- The participants communicate by **message passing** (calls).
- Only instances can participate the interactions.
  (the instance names are not underlined in this case, unlike in the other diagrams)

An Example of Sequence Diagram



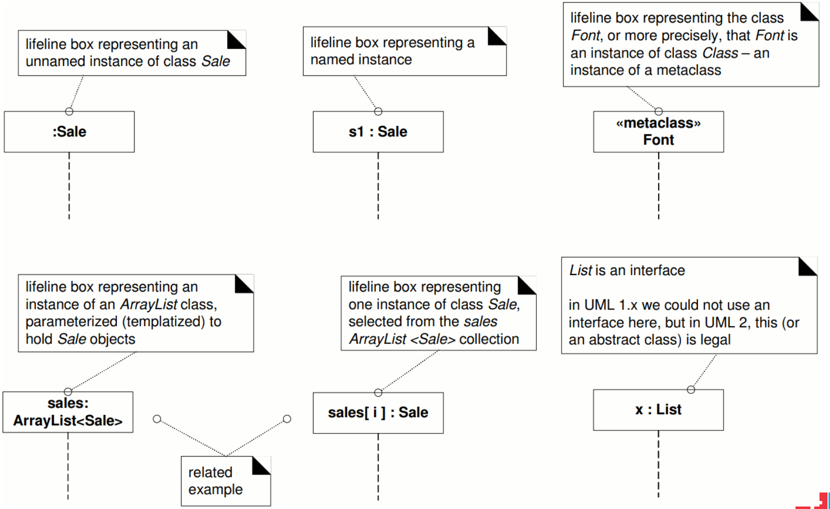
(adopted from “UML Sequence Diagrams” by uml-diagrams.org)

Sequence Diagram in the Context of Class Diagram


: Register
: Sale
makePayment(cashTendered)
makePayment(cashTendered)
messages in interaction diagrams indicate operations in the class diagrams
classes identified in the interaction diagrams are declared in the class diagrams
Register
...
makePayment(...)
...
Sale
...
makePayment(...)
...
1
currentSale

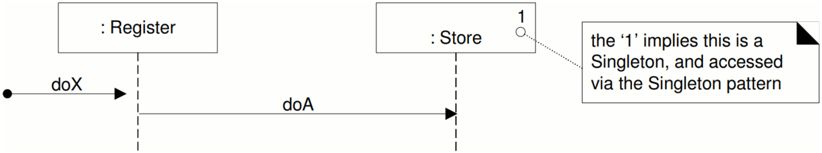
(adopted from “Applying UML and Patterns” by Craig Larman)

Lifelines Notation in Sequence Diagrams


lifeline box representing an unnamed instance of class *Sale*
:Sale
lifeline box representing a named instance
s1 : Sale
lifeline box representing the class *Font*, or more precisely, that *Font* is an instance of class *Class* – an instance of a metaclass
«metaclass» Font
lifeline box representing an instance of an *ArrayList* class, parameterized (templatized) to hold *Sale* objects
sales: ArrayList<Sale>
related example
lifeline box representing one instance of class *Sale*, selected from the *sales* *ArrayList <Sale>* collection
sales[i] : Sale
List is an interface
in UML 1.x we could not use an interface here, but in UML 2, this (or an abstract class) is legal
x : List

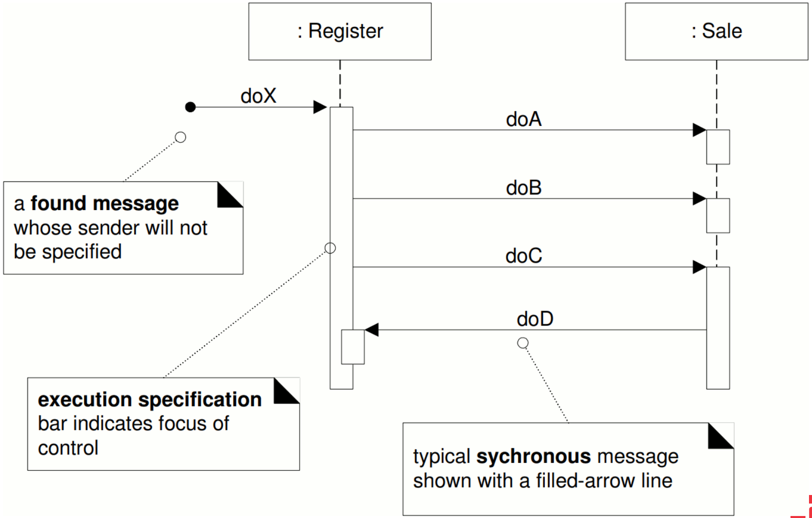
(adopted from “Applying UML and Patterns” by Craig Larman)

Singleton in Sequence Diagrams


: Register
: Store
1
doX
doA
the ‘1’ implies this is a Singleton, and accessed via the Singleton pattern

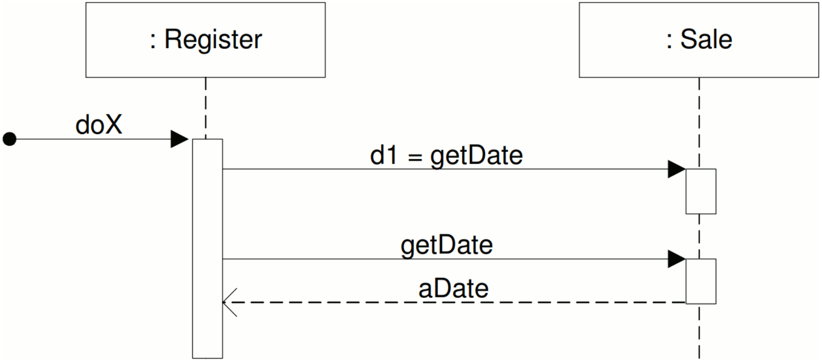
(adopted from “Applying UML and Patterns” by Craig Larman)

Messages and Activities in Sequence Diagrams


: Register
: Sale
doX
doA
doB
doC
doD
a **found message** whose sender will not be specified
execution specification bar indicates focus of control
typical **sychronous** message shown with a filled-arrow line

(adopted from “Applying UML and Patterns” by Craig Larman)

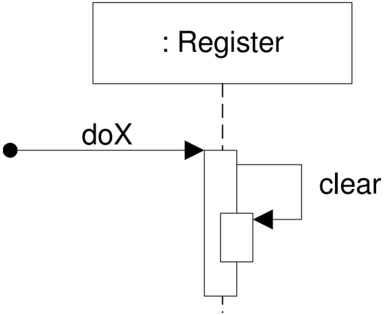
Reply Messages in Sequence Diagrams



(adopted from “Applying UML and Patterns” by Craig Larman)

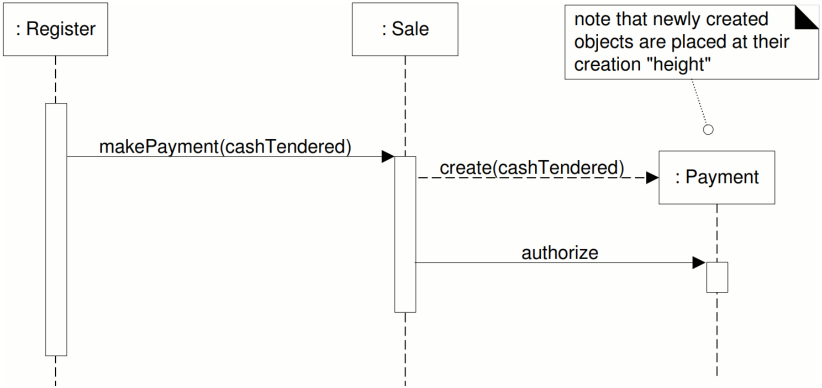
Self (Recursive) Messages in Sequence Diagrams

(the resulting activity may or may not be modelled in the diagram)



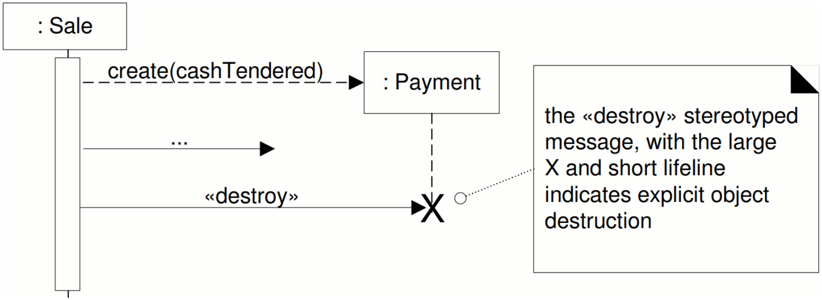
(adopted from “Applying UML and Patterns” by Craig Larman)

Creating Instances in Sequence Diagrams


: Register
: Sale
note that newly created objects are placed at their creation "height"
makePayment(cashTendered)
create(cashTendered)
: Payment
authorize

(adopted from “Applying UML and Patterns” by Craig Larman)

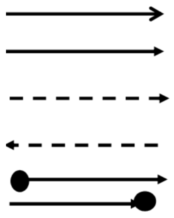
Destroying Instances in Sequence Diagrams


: Sale
create(cashTendered)
: Payment
...
«destroy»
X
the «destroy» stereotyped message, with the large X and short lifeline indicates explicit object destruction

(adopted from “Applying UML and Patterns” by Craig Larman)

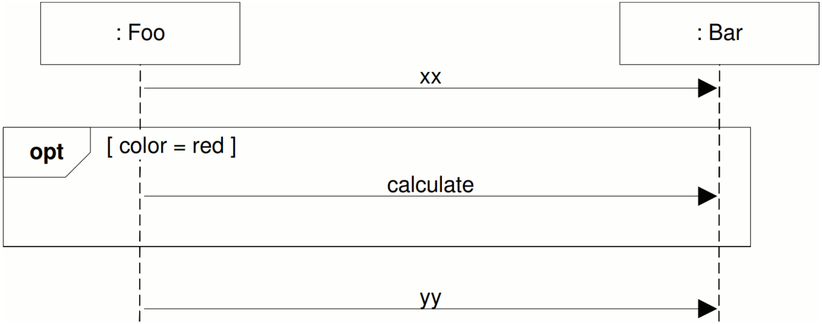
Message Types in Sequence Diagrams

- asynchronous – a sender remains active
- synchronous – a sender passes control to a receiver
- creation of an object
- reply – the end of message processing by its receiver, optionally with a return value
  (modelled only if necessary, otherwise should be omitted)
- found/lost – to enter/leave a sequence diagram



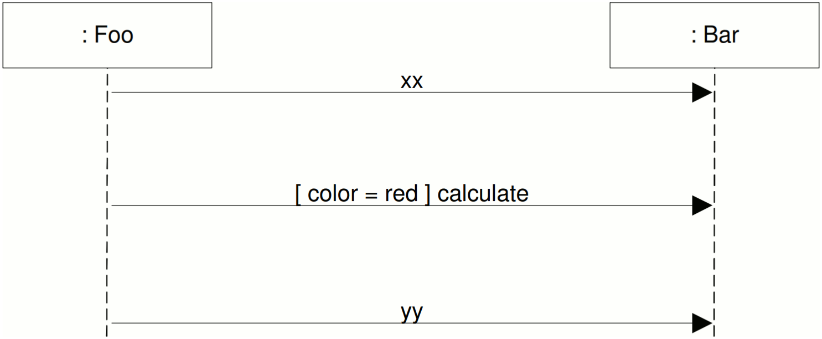
- A message can be passed conditionally, repeatedly, etc.
- For such constraints, messages can be enclosed into frames.
  (so called “combined fragment”, e.g., opt, alt, loop, break, par, etc.)

“Option” Fragment in UML 2



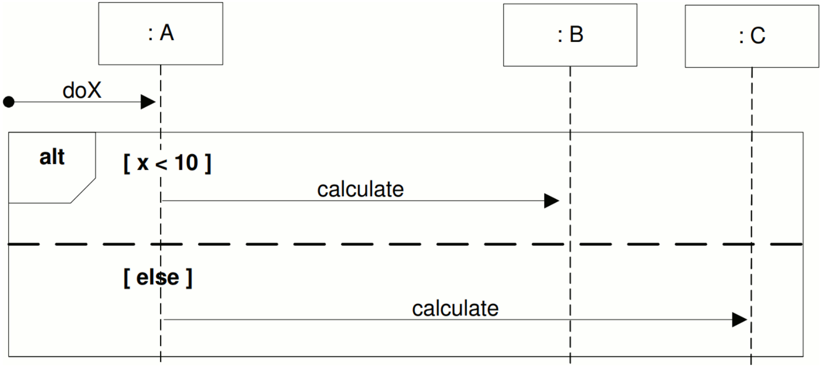
(adopted from “Applying UML and Patterns” by Craig Larman)

Conditional Messages in UML 1



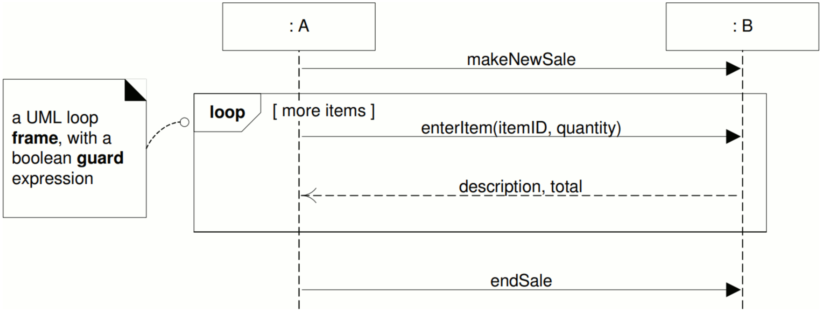
(adopted from “Applying UML and Patterns” by Craig Larman)

“Alternatives” Fragment



(adopted from “Applying UML and Patterns” by Craig Larman)

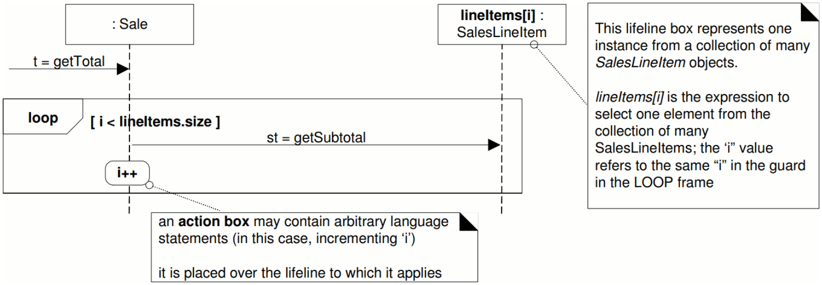
“Loop” Fragment with an Iteration Condition


: A
: B
makeNewSale
a UML loop **frame**, with a boolean **guard** expression
loop
[more items]
enterItem(itemID, quantity)
description, total
endSale

(adopted from “Applying UML and Patterns” by Craig Larman)

“Loop” Fragment with an Explicit Iterator

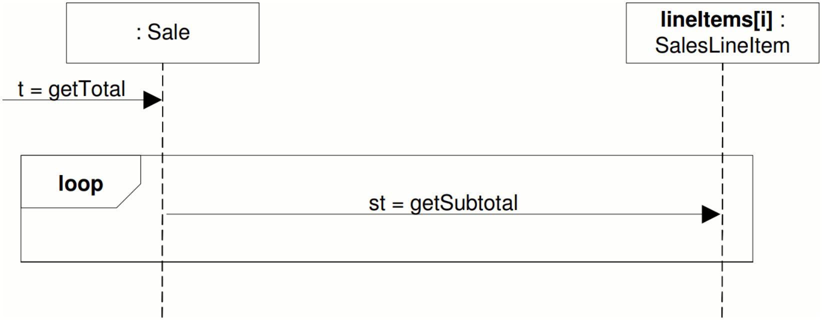
for iterating through a collection


: Sale
lineItems[i] : SalesLineItem
t = getTotal
loop
[i < lineItems.size]
st = getSubtotal
i++
This lineline box represents one instance from a collection of many SalesLineItem objects.
lineItems[i] is the expression to select one element from the collection of many SalesLineItems; the ‘i” value refers to the same “i” in the guard in the LOOP frame
an action box may contain arbitrary language statements (in this case, incrementing ‘i’)
it is placed over the lifeline to which it applies

(adopted from “Applying UML and Patterns” by Craig Larman)

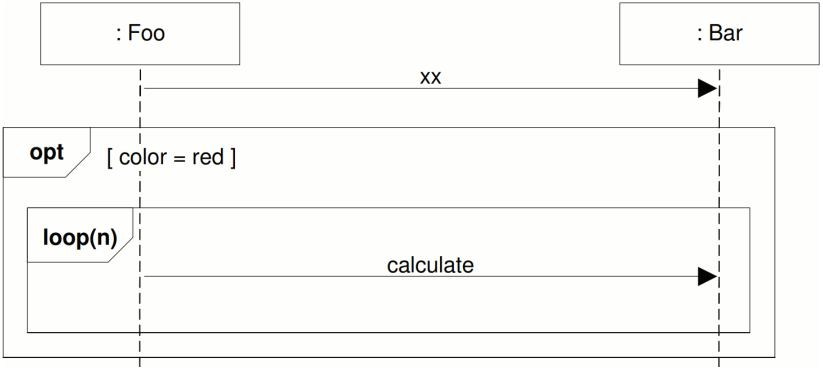
“Loop” Fragment with an Implicit Iterator

for iterating through a collection



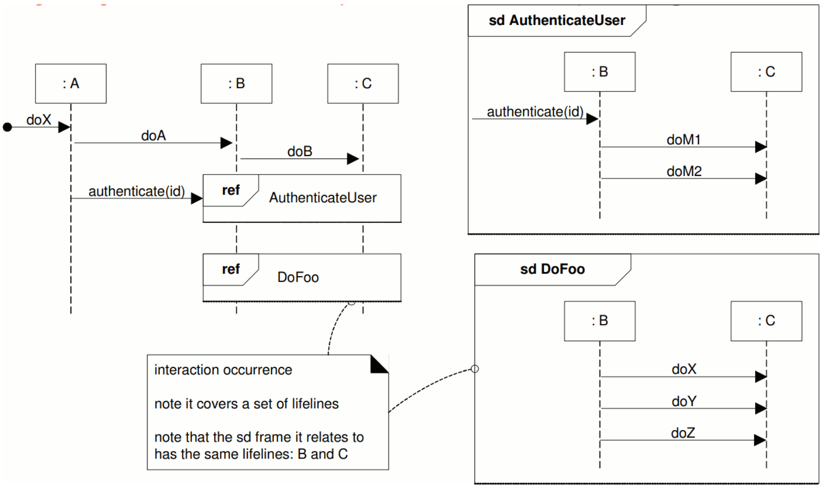
(adopted from “Applying UML and Patterns” by Craig Larman)

Nested Fragments



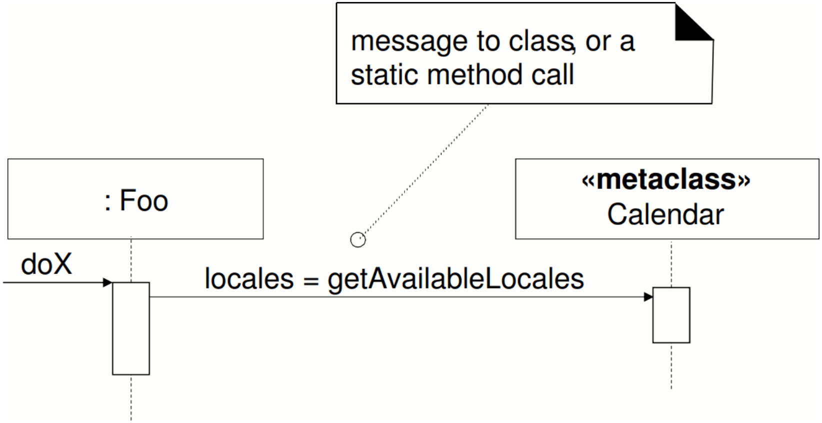
(adopted from “Applying UML and Patterns” by Craig Larman)

Referring to Another Interaction Diagram



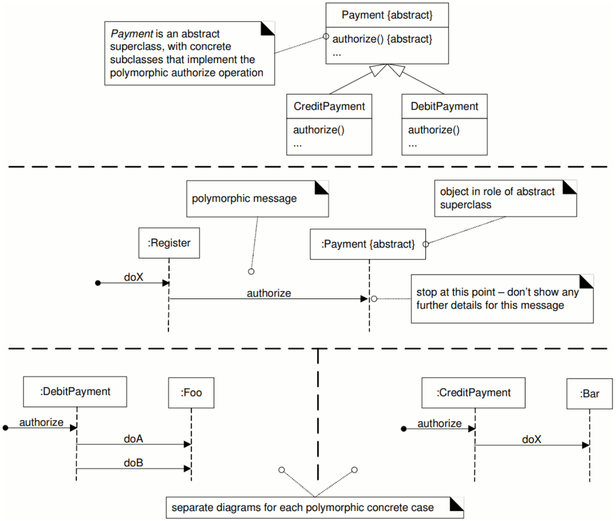
(adopted from “Applying UML and Patterns” by Craig Larman)

Static Method Calls in Sequence Diagrams


message to class, or a static method call
: Foo
«metaclass»
Calendar
doX
locales = getAvailableLocales

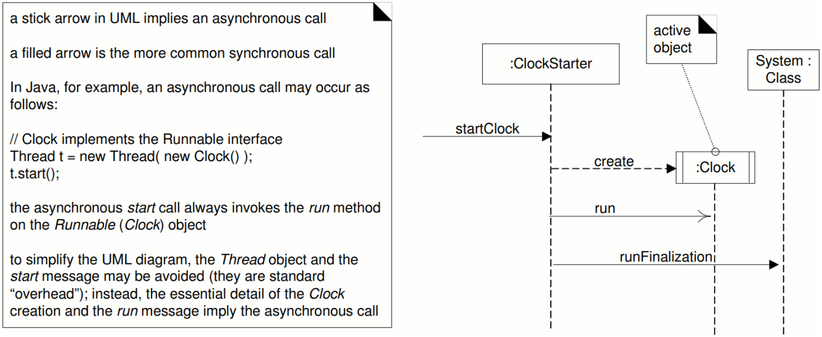
(adopted from “Applying UML and Patterns” by Craig Larman)

Modelling of Polymorphism in Sequence Diagrams


Payment is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation
Payment {abstract}
authorize() {abstract}
...
CreditPayment
authorize()
...
DebitPayment
authorize()
...
polymorphic message
object in role of abstract superclass
:Register
:Payment {abstract}
doX
authorize
stop at this point – don't show any further details for this message
:DebitPayment
:Foo
authorize
doA
doB
:CreditPayment
:Bar
authorize
doX
separate diagrams for each polymorphic concrete case

(adopted from “Applying UML and Patterns” by Craig Larman)

Asynchronous Calls and Active Classes



a stick arrow in UML implies an asynchronous call

a filled arrow is the more common synchronous call

In Java, for example, an asynchronous call may occur as follows:

```
// Clock implements the Runnable interface
Thread t = new Thread( new Clock() );
t.start();
```

the asynchronous *start* call always invokes the *run* method on the *Runnable* (*Clock*) object

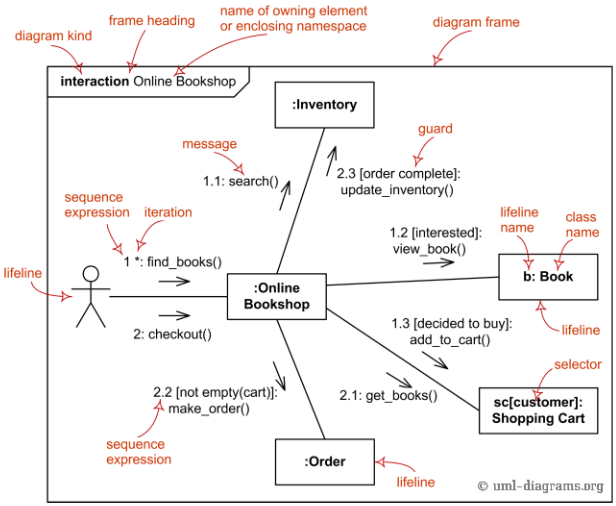
to simplify the UML diagram, the *Thread* object and the *start* message may be avoided (they are standard “overhead”); instead, the essential detail of the *Clock* creation and the *run* message imply the asynchronous call

(adopted from “Applying UML and Patterns” by Craig Larman)

UML Communication Diagram

- Description of an interaction with focus on a static structure.
- The notation is similar to the sequence diagram.
  (however, there is only one dimension, no timeline)
- The ability to model conditional messaging and loop, etc.
  (however, it is more difficult to understand such interactions)
- Easy to draw, therefore more suitable for whiteboard sketching.
  (it is important in the case of agile modelling)

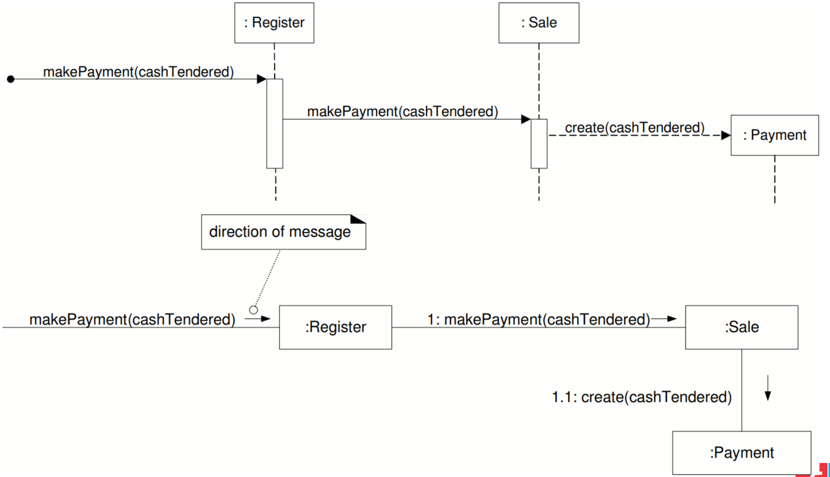
An Example of Communication Diagram



(adopted from “UML Communication Diagrams Overview” by uml-diagrams.org)

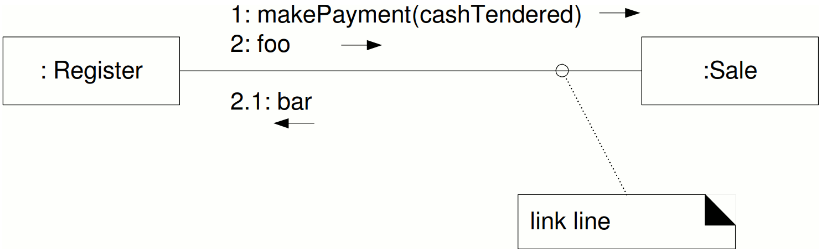
Sequence and Communication Diagrams

(two different views of the same interaction)


: Register
: Sale
makePayment(cashTendered)
makePayment(cashTendered)
create(cashTendered)
: Payment
direction of message
makePayment(cashTendered)
:Register
1: makePayment(cashTendered)
:Sale
1.1: create(cashTendered)
:Payment

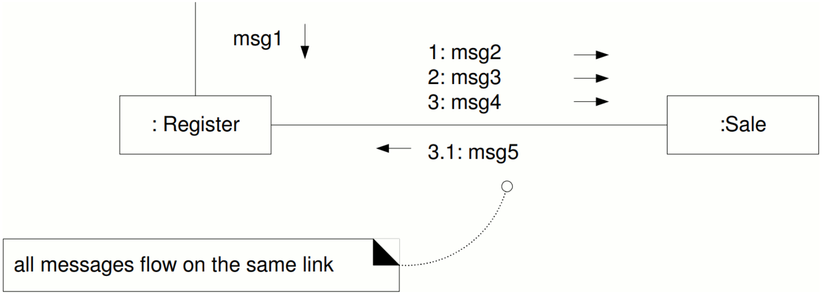
(adopted from “Applying UML and Patterns” by Craig Larman)

Links between Interaction Participants



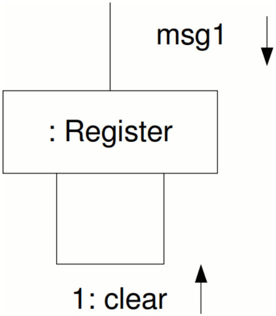
(adopted from “Applying UML and Patterns” by Craig Larman)

Messages and Links in Communication Diagrams



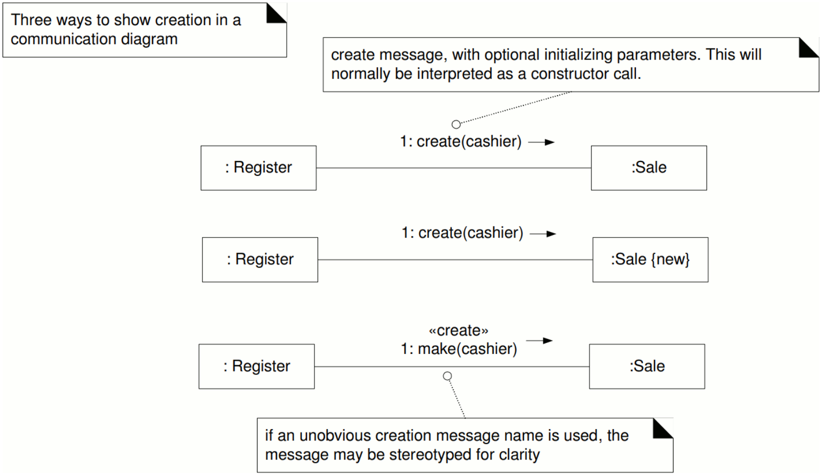
(adopted from “Applying UML and Patterns” by Craig Larman)

Self Messages in Communication Diagrams



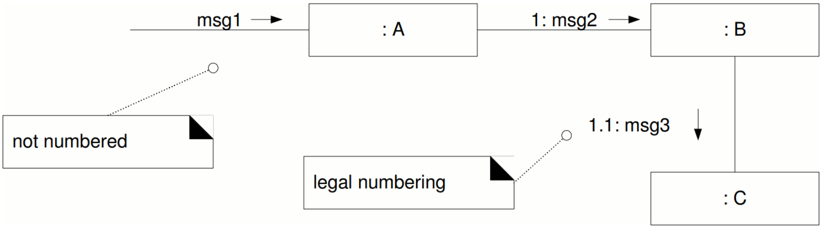
(adopted from “Applying UML and Patterns” by Craig Larman)

Creating Instances in Communication Diagrams


Three ways to show creation in a communication diagram
create message, with optional initializing parameters. This will normally be interpreted as a constructor call.
1: create(cashier)
: Register
:Sale
1: create(cashier)
: Register
:Sale {new}
«create»
1: make(cashier)
: Register
:Sale
if an unobvious creation message name is used, the message may be stereotyped for clarity

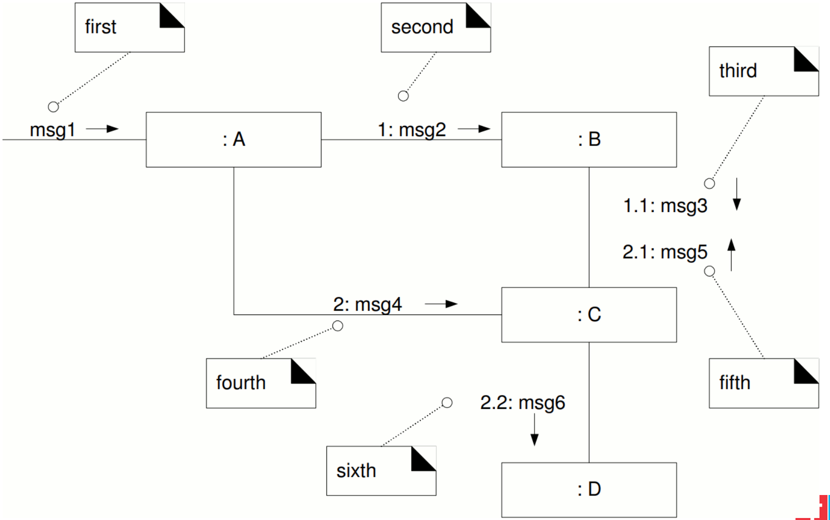
(adopted from “Applying UML and Patterns” by Craig Larman)

Message Numbering during Interactions



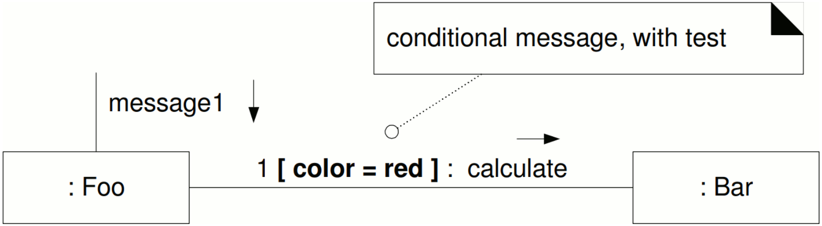
(adopted from “Applying UML and Patterns” by Craig Larman)

Message Numbering – A More Complex Example



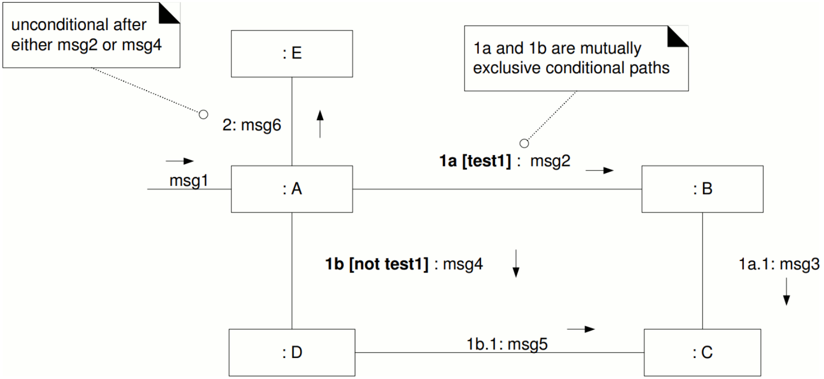
(adopted from "Applying UML and Patterns" by Craig Larman)

Conditional Messages in Communication Diagrams



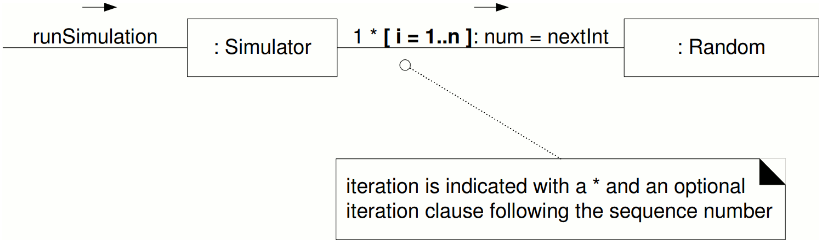
(adopted from “Applying UML and Patterns” by Craig Larman)

Mutually Exclusive Messages


unconditional after either msg2 or msg4
: E
1a and 1b are mutually exclusive conditional paths
2: msg6
1a [test1] : msg2
msg1
: A
: B
1b [not test1] : msg4
1a.1: msg3
: D
1b.1: msg5
: C

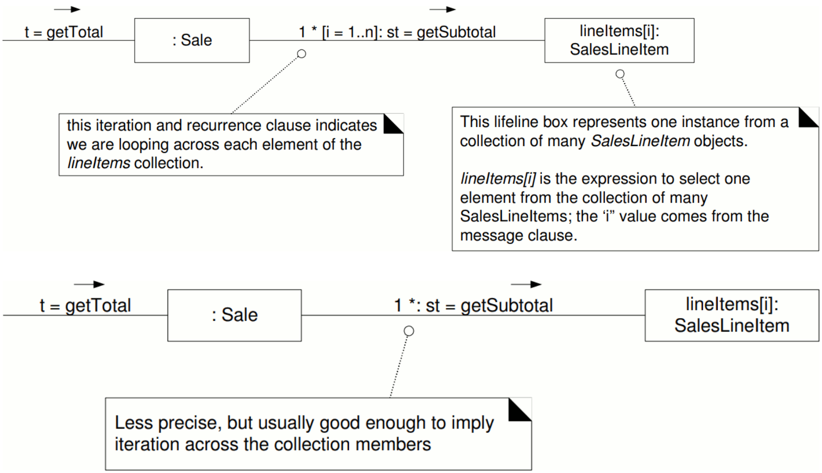
(adopted from “Applying UML and Patterns” by Craig Larman)

Iteration by Condition in Communication Diagrams


runSimulation
: Simulator
1 * [i = 1..n]: num = nextInt
: Random
iteration is indicated with a * and an optional iteration clause following the sequence number

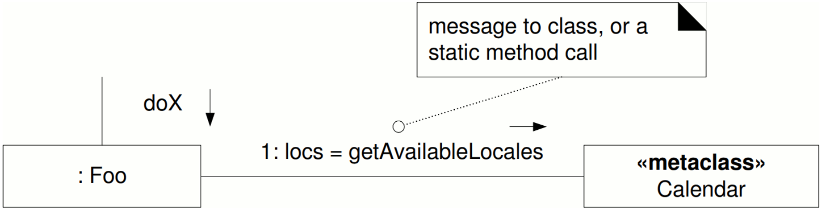
(adopted from “Applying UML and Patterns” by Craig Larman)

Iteration through Collections


t = getTotal
: Sale
1 * [i = 1..n]: st = getSubtotal
lineItems[i]:
SalesLineItem
this iteration and recurrence clause indicates we are looping across each element of the *lineItems* collection.
This lineline box represents one instance from a collection of many *SalesLineItem* objects.
lineItems[i] is the expression to select one element from the collection of many SalesLineItems; the 'i" value comes from the message clause.
t = getTotal
: Sale
1 *: st = getSubtotal
lineItems[i]:
SalesLineItem
Less precise, but usually good enough to imply iteration across the collection members

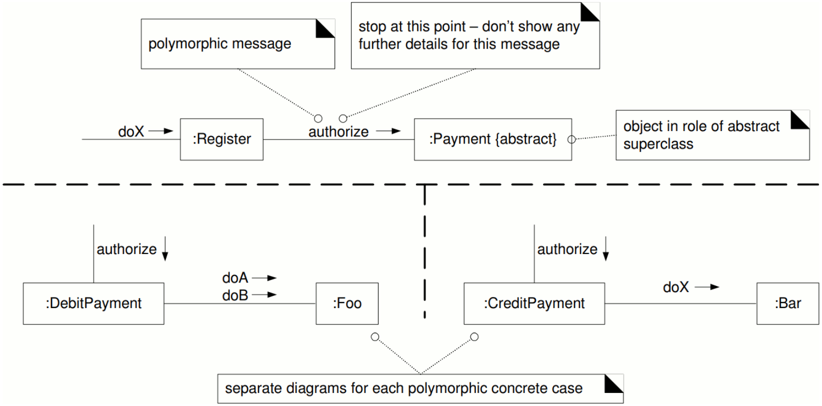
(adopted from "Applying UML and Patterns" by Craig Larman)

Static Method Calls in Communication Diagrams


message to class, or a static method call
doX
1: locs = getAvailableLocales
: Foo
«metaclass»
Calendar

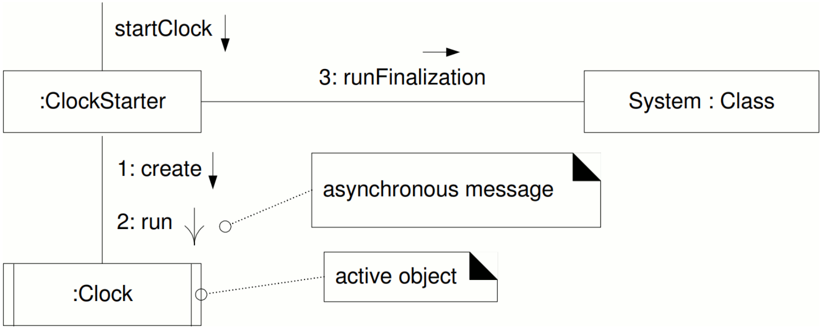
(adopted from “Applying UML and Patterns” by Craig Larman)

Modelling of Polymorphism


polymorphic message
stop at this point – don't show any further details for this message
doX
:Register
authorize
:Payment {abstract}
object in role of abstract superclass
authorize
:DebitPayment
doA
doB
:Foo
authorize
:CreditPayment
doX
:Bar
separate diagrams for each polymorphic concrete case

(adopted from "Applying UML and Patterns" by Craig Larman)

Asynchronous and Synchronous Calls

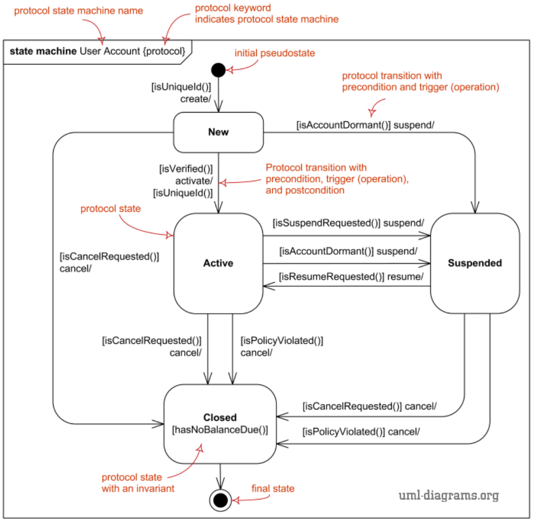


(adopted from “Applying UML and Patterns” by Craig Larman)

UML State Machine Diagram

- A behavioural diagram to describe
  - internal “states” of objects,
  - “activities” of the objects in the states,
  - “events” triggering transitions between the states,
  - “action” related to those transitions.
- The internal state of an object is defined by
  - values of its attributes,
  - existing relationships to other objects.
- The ability to model conditional messaging and loop, etc.
  (however, it is more difficult to understand such interactions)
- Easy to draw, therefore more suitable for whiteboard sketching.
  (it is important in the case of agile modelling)

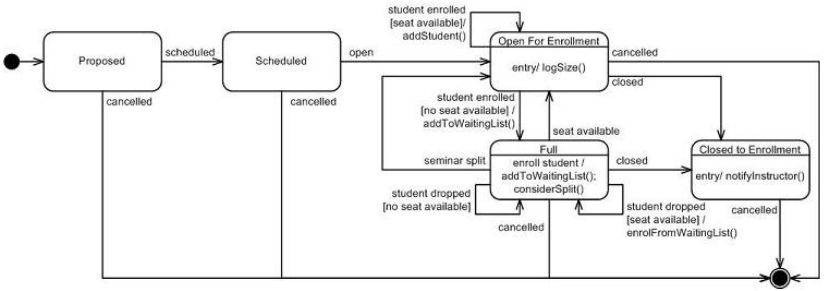
An Example of (Protocol) State Diagram


protocol state machine name
protocol keyword indicates protocol state machine
state machine User Account {protocol}
initial pseudostate
[isUniqueId()] create/
protocol transition with precondition and trigger (operation)
[isAccountDormant()] suspend/
New
[isVerified()] activate/ [isUniqueId()]
Protocol transition with precondition, trigger (operation), and postcondition
protocol state
[isSuspendRequested()] suspend/
[isAccountDormant()] suspend/
[isResumeRequested()] resume/
[isCancelRequested()] cancel/
Active
Suspended
[isCancelRequested()] cancel/
[isPolicyViolated()] cancel/
[isCancelRequested()] cancel/
[isPolicyViolated()] cancel/
Closed [hasNoBalanceDue()]
protocol state with an invariant
final state
uml-diagrams.org

(adopted from “UML Communication Diagrams Overview” by uml-diagrams.org)

Another Example of State Diagram

(with actions on transitions and activities in states)



(adopted from “UML 2 State Machine Diagrams: An Agile Introduction” by Scott W. Ambler)

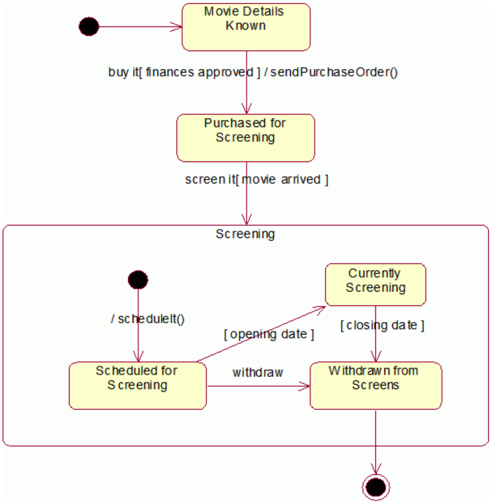
Events, Actions, and Activities in State Diagrams

- An event can be of various types:

call	to perform a particular operation (the event triggers a transition on a method call)
signal	sent/received asynchronously (the event triggers a transition on a particular signal)
change	of value a particular boolean expression (the event triggers a transition on the false-to-true turn)
time	in a particular time (the event is generated in a particular time)

- An action is immediate, interminable, triggered on particular event.
  (there can be "entry/", "exit/", or internal "xxx/" actions)
- An activity is long-term, terminable by an event, and always related to a particular state.
  (it this case, the activity is prefixed by "do/")
- The actions and the activities are usually operations of an object.
  (and transitions are in form "event [condition] / action")

Yet Another Example of State Diagram



(adopted from “Practical Software Engineering” by Maciaszek&Liong)

UML Activity Diagram

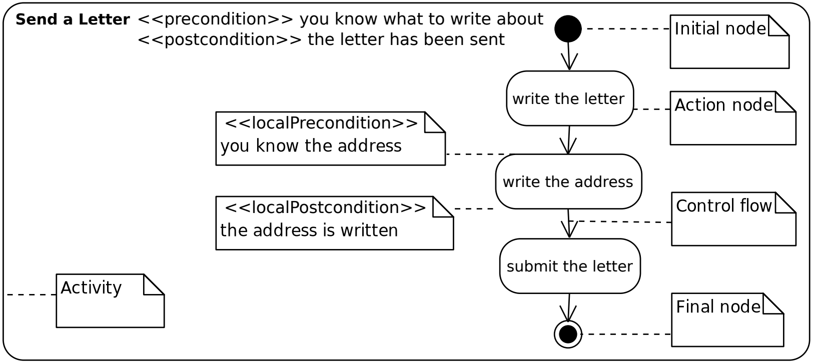
- Based on work-flow modelling and on Petri nets.
  (the Petri nets semantics in UML 2 replaced a state machine semantics in UML 1)
- Independent on a system's static structure, on classes/objects.
  (contrary to the UML machine state diagram)
- Modelling of an activity by sequential/parallel flows of nodes.
  (modelling of business processes, work/data-flows, operations, behaviour, etc.)
- Three types of nodes:

action nodes	are individual atomic actions in the activity
control nodes	to control flows through the modelling activity
object nodes	are objects utilised/operated by the activity

- Two types of flows between the nodes:

control flow	through the modelling activity to pass the control
object flow	to pass the objects utilised/operated by the activity

An Example of Activity Diagram


Send a Letter
<<precondition>> you know what to write about
<<postcondition>> the letter has been sent
Initial node
write the letter
Action node
<<localPrecondition>> you know the address
write the address
<<localPostcondition>> the address is written
Control flow
submit the letter
Activity
Final node

Modelling of Use Cases by Activity Diagrams

1. Pay VAT

ID: UC_VAT_1

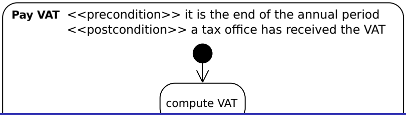
Pay VAT to a tax office at the end of the annual period.

Primary Actors	Time
Supporting Actors	Tax Office
Level	User
Complexity	Medium
Use Case Status	Base
Implementation Status	Scheduled
Preconditions	it is the end of the annual period
Post-conditions	a tax office has received the VAT
Author	N/A
Assumptions	N/A

1.1. Scenarios

1.1.1. Scenario

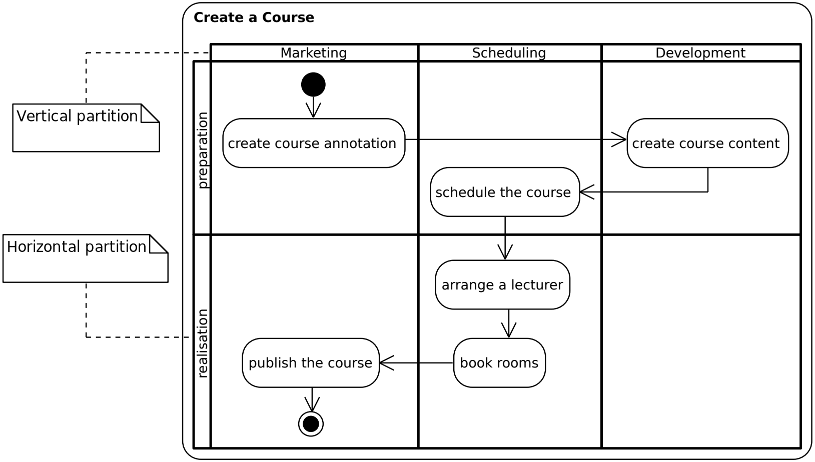
1. The use case is executed at the end of the annual period.
2. SYSTEM Compute the VAT which has to be sent to the tax office.
3. SYSTEM Make a wire transfer of the VAT to the tax office.


Pay VAT
<<precondition>> it is the end of the annual period
<<postcondition>> a tax office has received the VAT
compute VAT

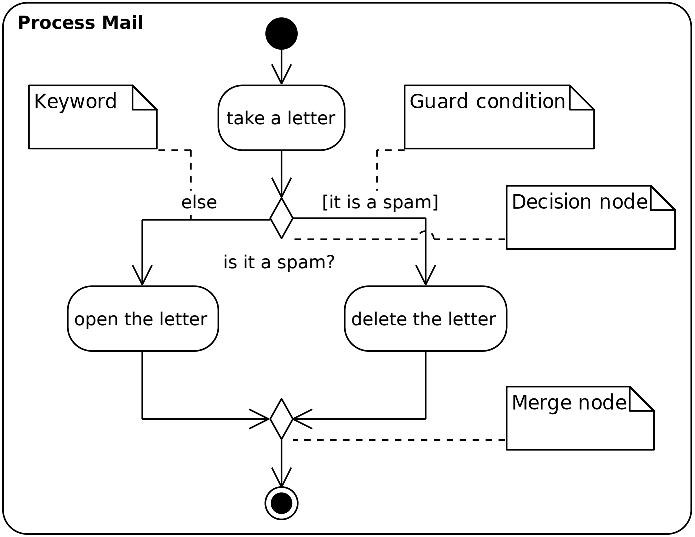
Semantics of Activity Diagram

- In UML 2, activity diagram is based on Petri nets.
  (movement of tokens between places; a token can be a control, object, or data)
- The state of a modelled activity $=$ the actual placement of tokens.
- An action in the activity diagram is triggered if and only if
  - there are tokens available on all input edges of the action
  - and a local precondition is met.
- After the action, tokens are sent through all output edges of the action if and only if a local postcondition is met.
- The flow of tokens is also controlled by conditions on edges.

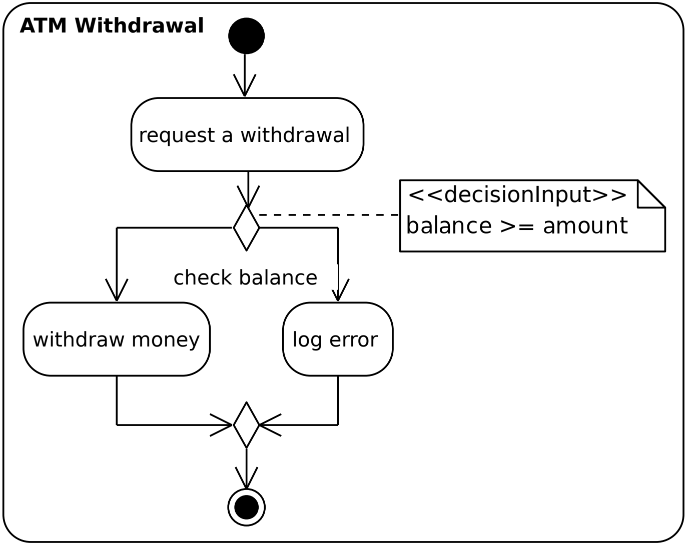
Partitions/Swim-Lanes in Activity Diagrams



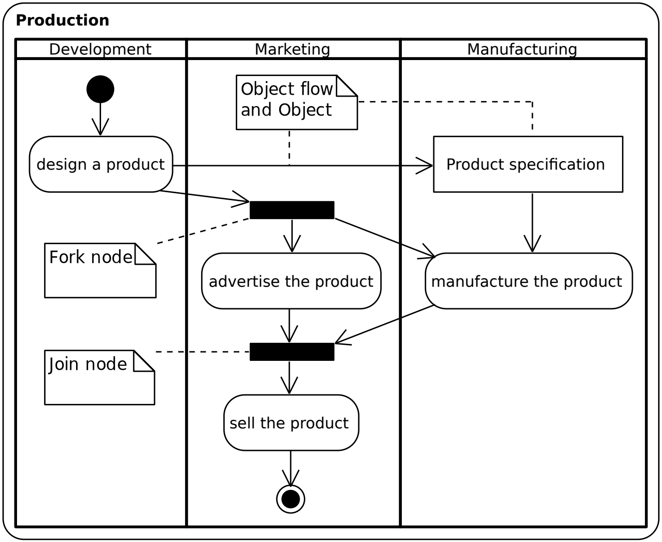
Decision and Merge Control Nodes



A decision Condition in the Control Node



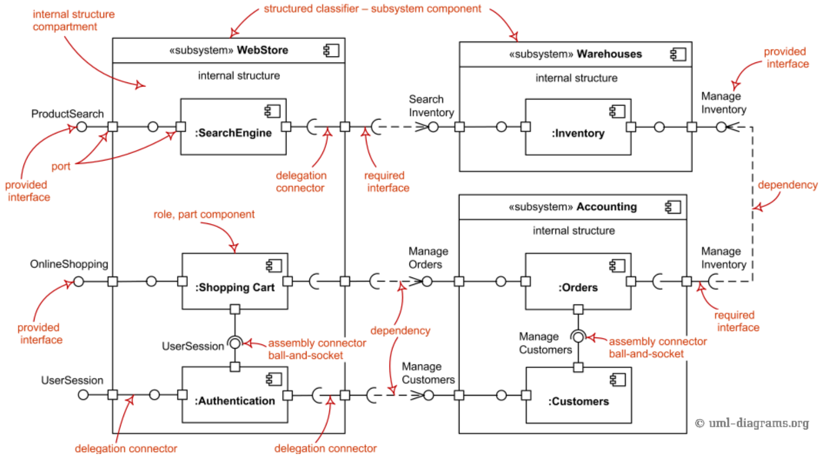
Fork and Join Control Nodes in Activity Diagrams



UML Component Diagram

- For implementation modelling, the diagram describe SW components.
  (including their classifiers, e.g., classes; artefacts, e.g., source-code files; etc.)
- The components are system modules encapsulating their content behind a particular API/interfaces.
  (they are black-boxes, or grey-boxes if decomposable to other components)
- Behaviour of a component is fully specified by its interfaces.
  (provided interfaces and required interfaces)
- Components (their behaviour) can be customisable by parameters.
- A system is composed of its interconnected component.
  (interconnected in accordance with the system's architecture)

An Example of Component Diagram

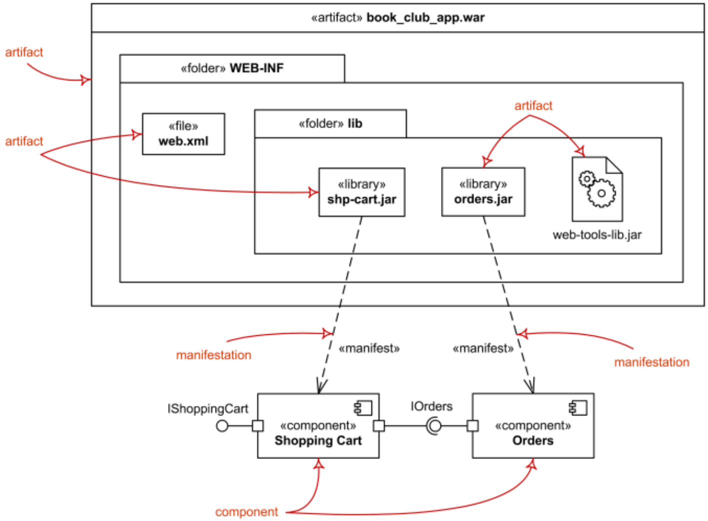


(adopted from “UML Component Diagrams” by uml-diagrams.org)

UML Deployment Diagram

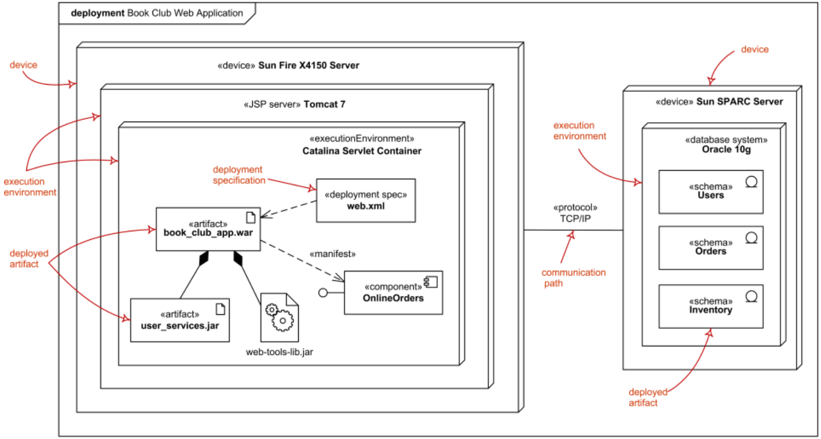
- For implementation modelling, it describes a physical deployment.
  (of software components/artefacts to a physical IT infrastructure)
- It allows to model a topology of the deployment.
  (nodes and communication links between the nodes)
- Useful for communicating an overall system architecture between developer themselves and with a customer.
  (both software and hardware/infrastructure architecture)
- Two types of nodes:
  - device node – physical/virtual hardware
  - execution environment node – software instances

An Example: Manifestation of Artefacts in Deployment



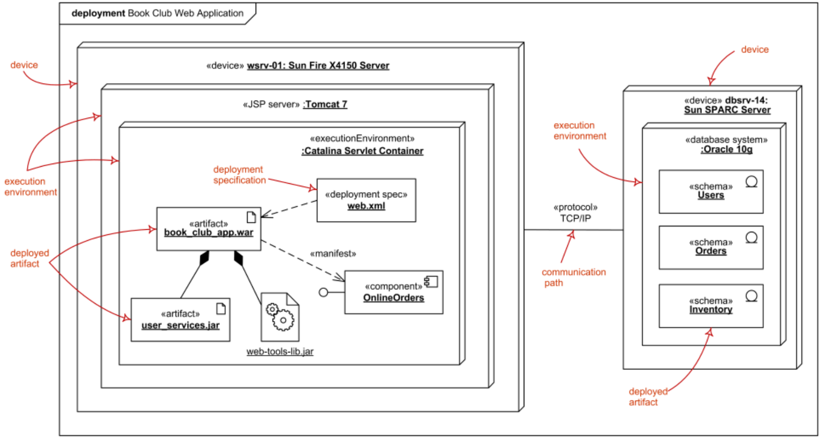
(adopted from “Deployment Diagrams Overview” by uml-diagrams.org)

An Example: Specification Level of Deployment



(adopted from “Deployment Diagrams Overview” by uml-diagrams.org)

An Example: Instance Level of Deployment

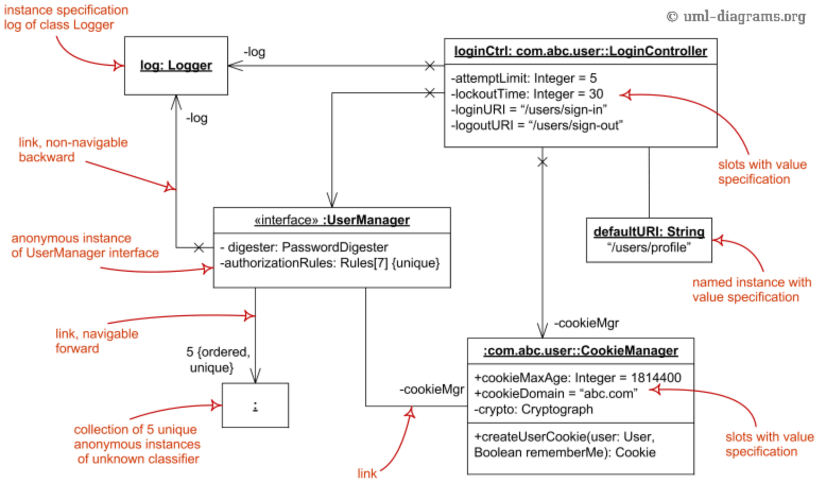

deployment Book Club Web Application
device
«device» wsrv-01: Sun Fire X4150 Server
«JSP server» :Tomcat 7
«executionEnvironment» :Catalina Servlet Container
execution environment
deployment specification
«deployment spec» web.xml
«artifact» book_club_app.war
deployed artifact
«manifest»
«component» OnlineOrders
«artifact» user_services.jar
web-tools-lib.jar
«protocol» TCP/IP
communication path
execution environment
device
«device» dbsrv-14: Sun SPARC Server
«database system» :Oracle 10g
«schema» Users
«schema» Orders
«schema» Inventory
deployed artifact

(adopted from “Deployment Diagrams Overview” by uml-diagrams.org)

UML Object Diagram

- Captures objects and their status in a particular point of time.
  (a snapshot of the detailed state of a system at a point in time)
- Helps to explain a complex structures of class instances.
  (as a graph of instances, including objects and data values)
- Objects described also in component/deployment diagrams.
  (they are special kinds of object diagrams if containing only instance specifications)
- Object diagram elements:
  - named and anonymous instance specifications for objects,
    (instances of classes from the corresponding class diagram)
  - slots with value specifications,
    (attribute values of the classes in their particular instances)
  - and links between the objects.
    (instances of associations from the corresponding class diagram)

An Example: Object Diagram

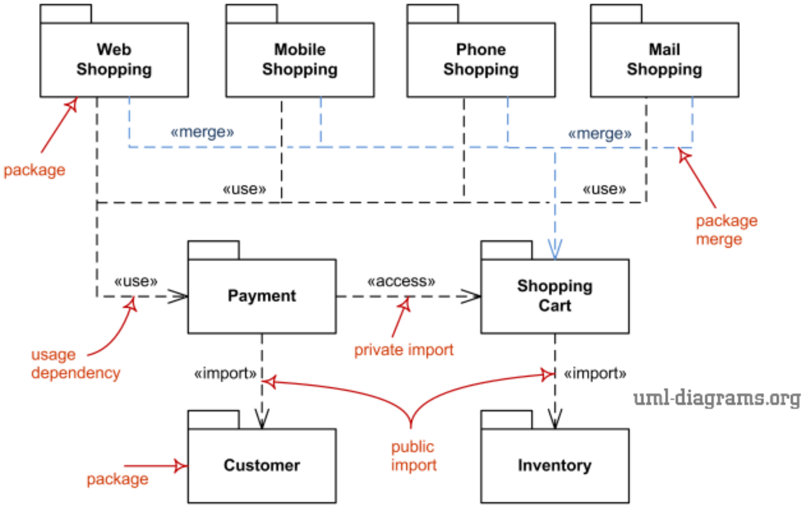


(adopted from "UML Class and Object Diagrams Overview" by uml-diagrams.org)

UML Package Diagram

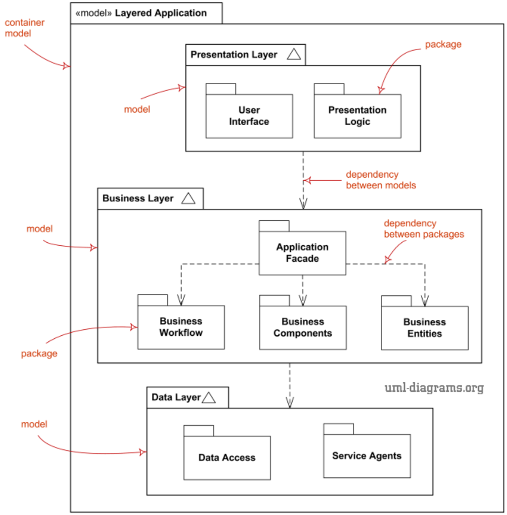
- Package is a namespace used to group together related elements.
  (to organize elements into groups to provide better structure for a system model)
- Elements with unique names within the enclosing package.
  (different types of elements can have the same name in the same package)
- Any package could be also a member of other package.
  (a system or a model is hierarchically decomposed by the packages)
- In the package diagram, members of a package may be shown
  - within the boundaries of the package,
    (suitable for shallow hierarchies)
  - or outside of the package by a package composition relationship.
    (branching lines from the package with a plus sign to the members)
- The diagram allows also to describe dependencies and generalisations of the packages.

An Example: Package Diagram for Software



(adopted from “UML Package Diagrams Overview” by uml-diagrams.org)

An Example: Package Diagram for Models

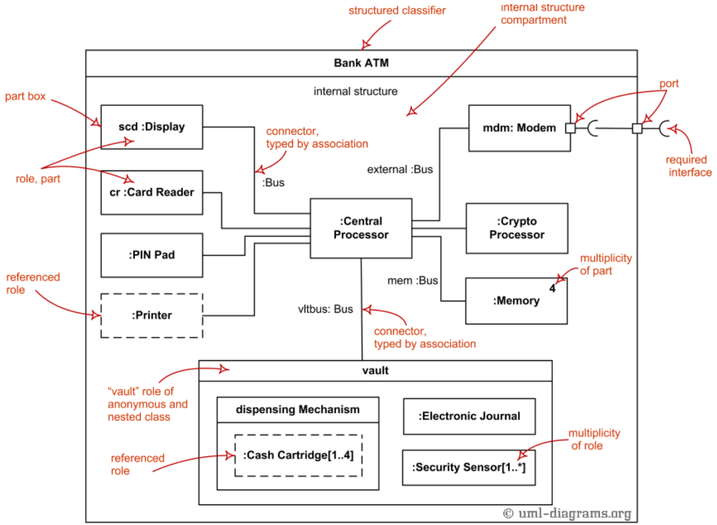


(adopted from “UML Package Diagrams Overview” by uml-diagrams.org)

UML Composite Structure Diagram

- A specific composition of interconnected run-time elements.
  (run-time instances of classifiers collaborating over communications links to achieve some common objectives in a software system)
- A composite structure diagram can describe
  - an internal structure of a classifier,
    (a decomposition of a classifier into its properties, parts and relationships)
  - collaboration of cooperating classifiers,
    (a cooperation of a set of classifiers by identifying their specific roles to play)
  - collaboration use in a specific situation.
    (a pattern application described by a collaboration to a specific situation)
- Only a static structure is captured, without any messaging.
  (contrary to the interaction diagrams in UML)

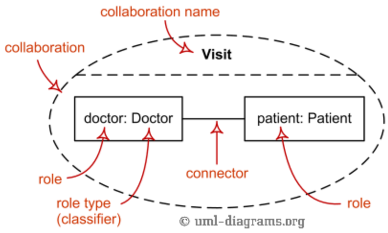
An Example: Internal Composite Structure

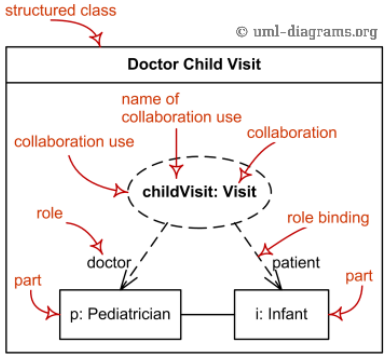


(adopted from "UML Composite Structure Diagrams" by uml-diagrams.org)

An Example: Collaboration and Collaboration Use

(on the left and on the right side, respectively)



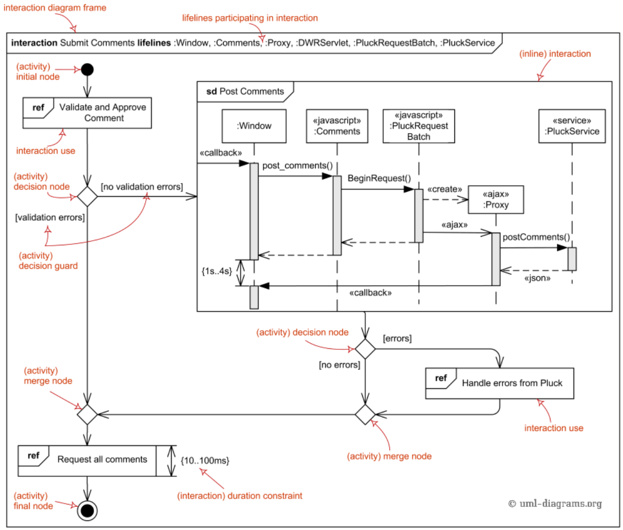


(adopted from “UML Composite Structure Diagrams” by uml-diagrams.org)

UML Interaction Overview Diagram

- An overview of the flow of control through control nodes.
  (like an activity diagram without invocation actions)
- The nodes of the flow are
  - inline interactions
    (inline interaction diagrams as special forms of call behaviour actions)
  - interaction uses
    ("ref" fragments allowing to use/call another interaction)
- Use various elements from both
  - the activity diagram
    (initial, flow final, activity final, decision, merge, fork, and join nodes)
  - the interaction diagram.
    (interaction, interaction use, duration constraint, time constraint elements)
- Branching and joining of branches should be properly nested.
  (this is more restrictive than in activity diagrams and could be quite difficult to obey)

An Example: Interaction Overview Diagram



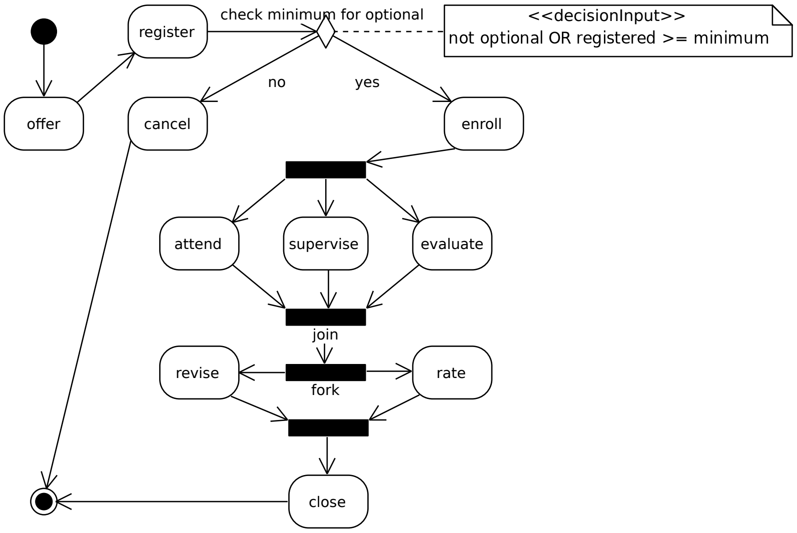
(adopted from “UML Interaction Overview Diagrams” by uml-diagrams.org)

An Example: Course Registration Information System

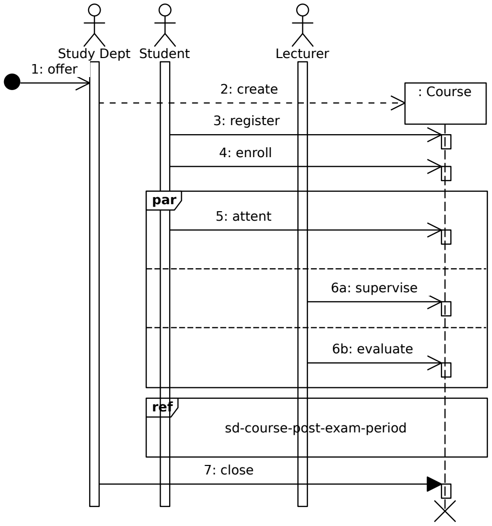
When analysing the faculty's IS, your were consulting with the teachers' representative about the requirements for the study part of the system. Specifically, you discussed the required behaviour of the system in relation to the courses. You have found that a course object appears in the list of courses **offered** to students. The student then can **register** to the course. In the case of **an optional course**, the course is **opened iff** at least the specified minimum number of students is registered into the course, otherwise the course is **cancelled**. During an enrolment period, students confirm their registration and are **enrolled**, or, exceptionally, they can cancel the registration and enrol another free course. After the course starts, the enrolled students **attend** the course, however, the Study Department may exclude from the course of a student who has ceased to study. Teachers **supervise** the course and may define and **evaluate assessments**. At a given date after the exam period, the course is closed for classification. After that, there is a post-exam period in which teachers can **revise** the classification if necessary and students can **rate** the course. By the end of this period, the course is definitely **closed**.

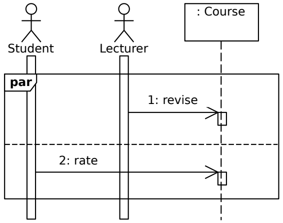
Describe the scenario above and a life-cycle of the Course object by appropriate UML diagrams.

The Example: Activity Diagram

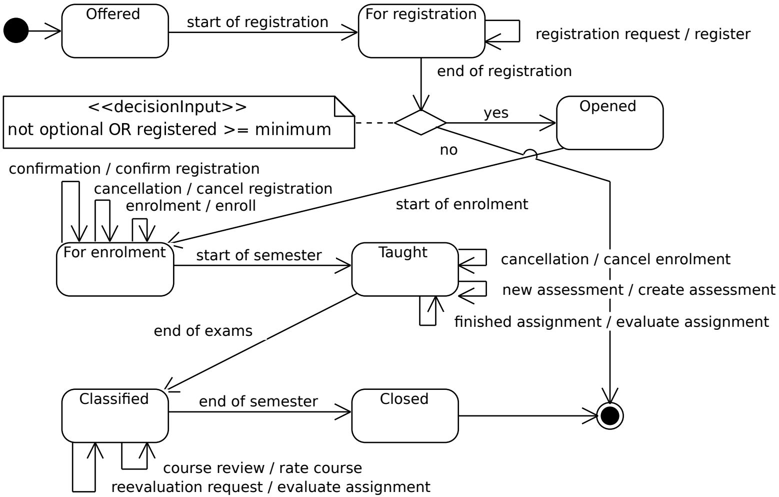


The Example: Sequence Diagram





The Example: State Machine Diagram


Offered
start of registration
For registration
registration request / register
end of registration
<<decisionInput>>
not optional OR registered >= minimum
yes
Opened
no
confirmation / confirm registration
cancellation / cancel registration
enrolment / enroll
start of enrolment
For enrolment
start of semester
Taught
cancellation / cancel enrolment
new assessment / create assessment
finished assignment / evaluate assignment
end of exams
Classified
end of semester
Closed
course review / rate course
reevaluation request / evaluate assignment

Thank you for your attention!

Marek Rychlý
<rychly@fit.vutbr.cz>