Object oriented modeling

2006/2007

Part I

UML

Modeling

- Modeling is important for creating high quality software
- We model in order to:
  - understand the system
  - specify desired structure and behaviour
  - describe the architecture and be able to alter it
  - improve risk management

Model

- Model is a simplification of reality
- The simplification allows to skip unimportant (at a particular moment) details
- At the same time it allows to emphasize important aspects

Modeling principles

- Model choice influences the solution of the problem - both the method and quality
- Every model can have various levels of detail
- The model should reflect reality
- Usually, one model is not enough. Several independent models are the best solution of the system is not trivial

Modeling paradigms

- Structural. It evolved from structural programming languages and spawned a large number of different approaches. The discrepancies between approaches severely limited usefulness of structural modeling Two attempts at unification:
  - CRIS (Comparative Review of Information Systems Methodologies) workgroup
  - EuroMethod

Modeling paradigms

- Object oriented. Resulted from increased interest in object-oriented languages. During '89-'94 period more than 50 different solutions were active, however, unlike the structural approach, they converged into one.

Modeling paradigms

- The most important methods that constituted the final approach were:
  - OMT (Object Modeling Technique), Rumbaugh 1991
  - OOAD (Object Oriented Analysis and Design), Booch 1991
  - OOSE (Object Oriented Software Engineering), Jacobson 1992

Towards UML

- Work on UML started in 1994, when Rumbaugh and Booch, both employed by Rational Software Corporation, started work on unification of OMT and OOAD. The result, Unified Method (UM) 0.8, was presented in '95. At the same year Jacobson joined Rational and enhanced UM with elements of his OOSE, which resulted in UM 0.9 and UM 0.91 (both in '96). From this point the language is known as UML.

UML development

- The efforts of Rational were quickly backed by some important players: IBM, DEC, HP, Oracle, Unisys, Microsoft- among others. This led to further developments and version 1.0 in 1997. This version was later passed to Object Management Group (OMG). Version 1.1 followed in the same year. This version was the official one up till 2001 (version 1.4). Version 1.5 became the official one in 2003.

UML 2.0

- Version 2.0 was introduced in 2003. It is the first major revision of the standard, introducing many new diagrams and modeling categories.

UML diagrams

- Model in UML is a graphical representation of the system
- The representation consists of logically interconnected diagrams
- Version 2.0 contains 13 types of diagrams
- An important concept is that of a **classifier** - abstract category that generalizes a collection of instances having the same features, and **instance** - a realization of classifier

UML views

- System design requires collaboration of a number of persons, having different competences and responsibilities (managers, designers, programmers, clients etc.)
- Each person sees the system from different point of view
- UML addresses this problem by employing 5 different views of the system

UML views

- use case view - most important, defines scope and expected functionality of the system
- dynamic view - describes behaviour (dynamics) of instances in the systems
- logical view - describes statics of the system
- implementation view - mostly used by programmers, describes components of the system
- deployment view - describes hardware required by the components

Extension mechanisms

- Although UML contains broad spectrum of concepts and elements, it may not suit a particular modeling domain
- For that reason extension mechanisms are incorporated into UML
- There are three types of extension mechanisms:
  - stereotype
  - constraint
  - tagged value

Stereotypes

- Allow to introduce new modeling categories based on existing ones
- Stereotypes can be:
  - textual - the name is surrounded by << >> quotes and placed on the stereotyped element
  - graphical - specific graphical symbol is placed on the stereotyped element
- A large number of standard stereotypes is recommended by OMG

Constraints

- Constraint is an expression describing condition applied to the constrained element
- It can be expressed in natural language, as a mathematic formula or in OCL (Object Constraint Language) - special language for object constraints
- Constraints are placed in { } parentheses, next to the constrained element

Tagged values

- Tagged values allow to define new properties
- They are expressed as name-value pairs
- They are placed in { } parentheses

Use case diagrams

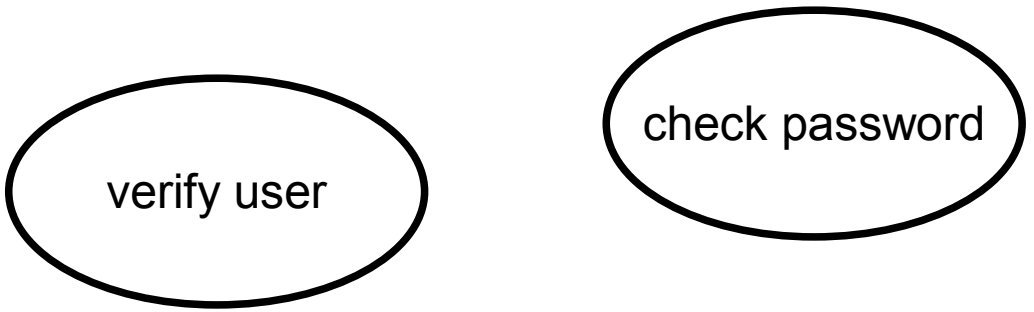
- They allow to:
  - identify and document requirements
  - analyze scope of applicability
  - communicate between developers, owners, clients etc.
  - develop project of the future system
  - develop testing procedures for the system
- There are two types of use case diagrams:
  - business use case diagrams
  - system use case diagrams

Use case diagrams

- They contain:
  - use cases
  - actors
  - relationships

Use cases

- Specification of sequence of actions (and their variants) which the system can perform through the interaction with actors of that system
- Use case is a coherent fragment of system functionality
- Its name a curt order to perform particular function, expressed in imperative. The namie is placed inside an oval

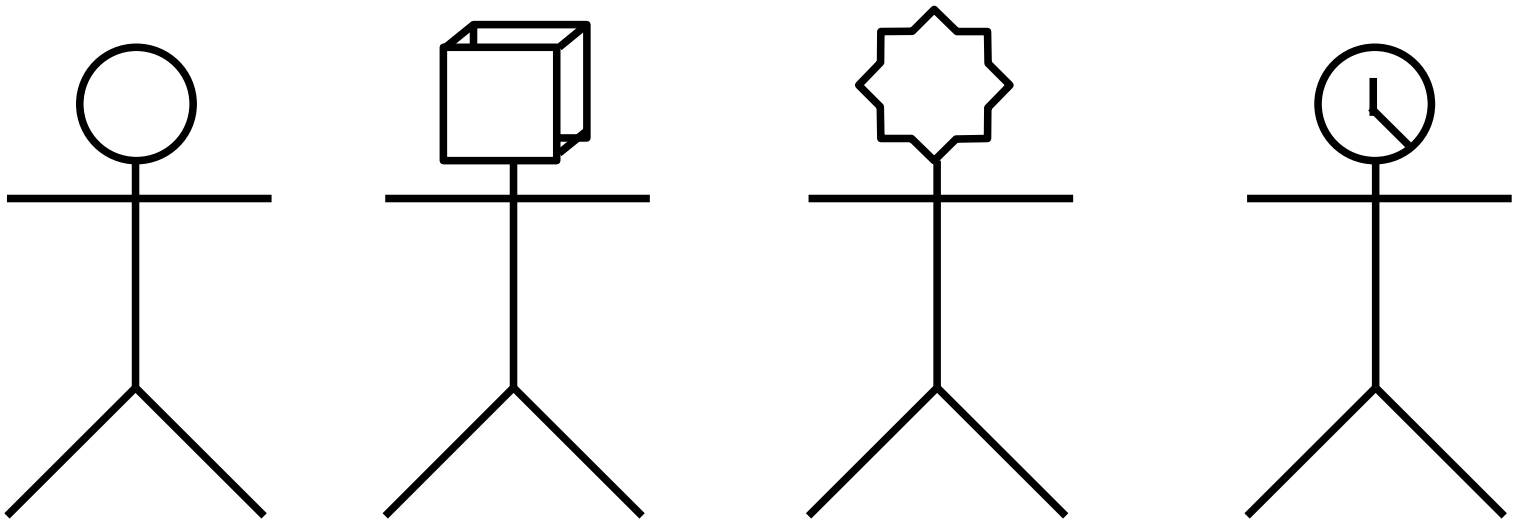


Actor

- Actor is a coherent collection of roles played by users of the use case, during interaction with this use case
- Actors can be
  - persons (single person, group, organisation etc.)
  - external systems (software or hardware)
  - time
- Name is a noun reflecting the role played in the system
- Actor can use more than one use case, an use case can interact with one or more actors

Actor stereotypes

- The classic symbol of actor can be stereotyped to distinguish between various types of actors



classic / human

external system

device

time

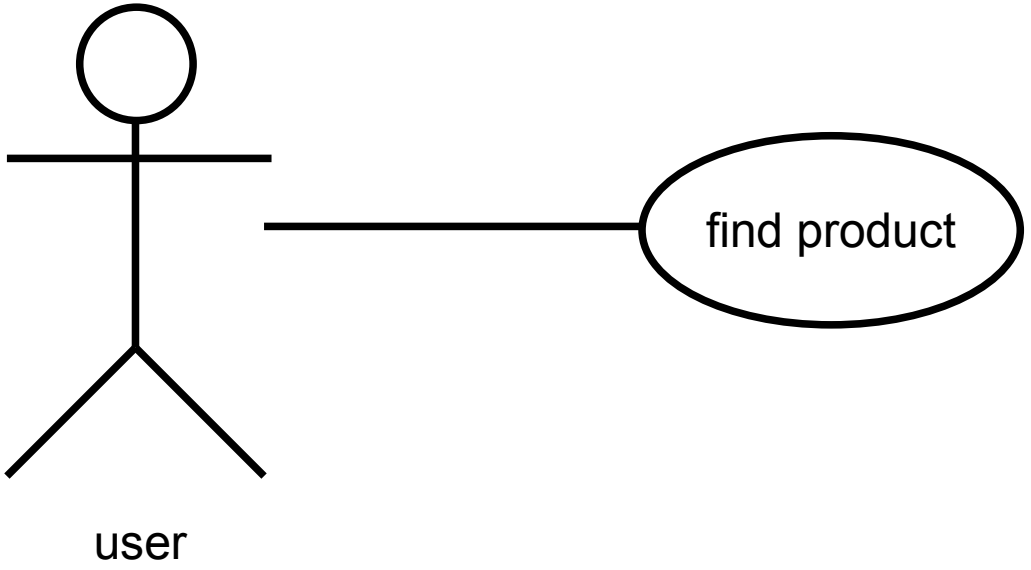
Relationship

- Ties elements of the diagram (e.g. actors and use cases)
- There are 4 kinds of relationship:
  - association
  - generalisation
  - dependence
  - realisation

Association

- Association describes ties between instances of classifiers (two or more)
- In the use case diagram it represents bidirectional communication between an actor and a use case
- It is depicted as a solid line
- Usually they do not have names

Association



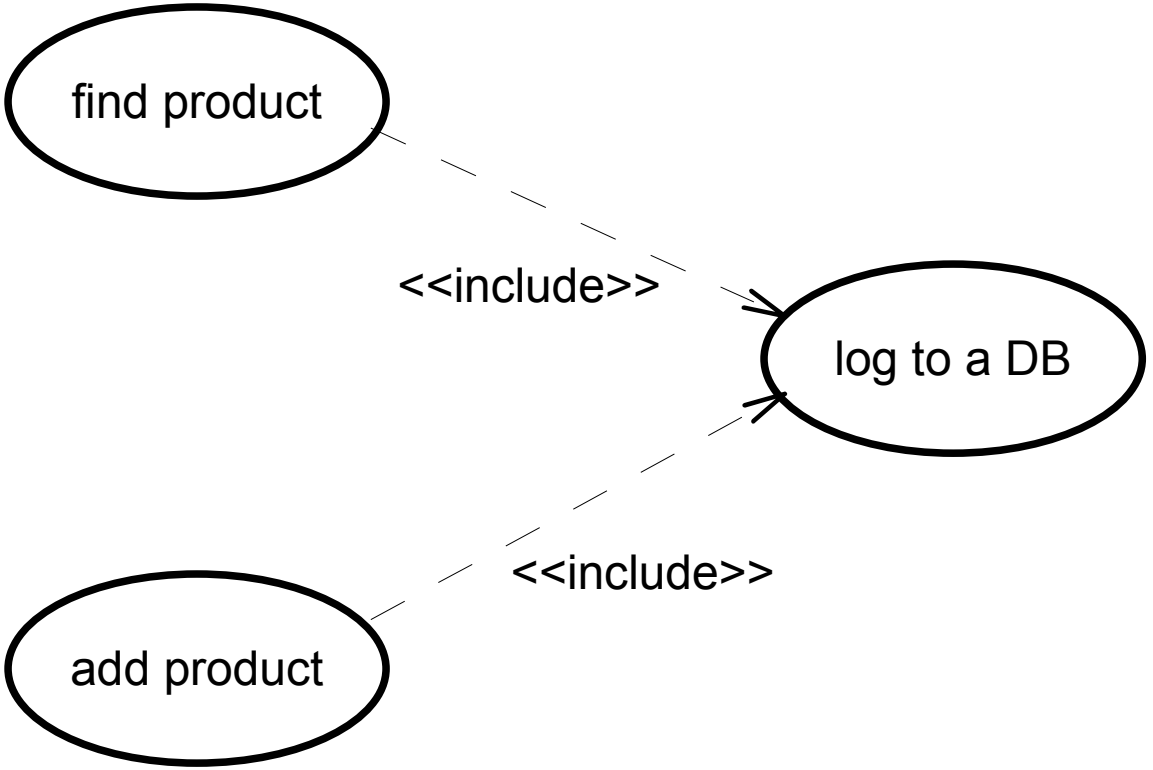
Dependency

- Dependency is a relationship between two model elements where change in one element (independent one) influences the second element (dependent one)
- It is depicted as a dashed arrow
- In use case diagrams dependency is stereotyped into:
  - <<include>> dependency
  - <<extend>> dependency

<<include>> dependency

- Relationship between the containing case and contained case
- The contained case is executed always when the containing case is executed - and only then
- It is useful when several use cases contain the same part
- The arrow points from the containing case to the contained case

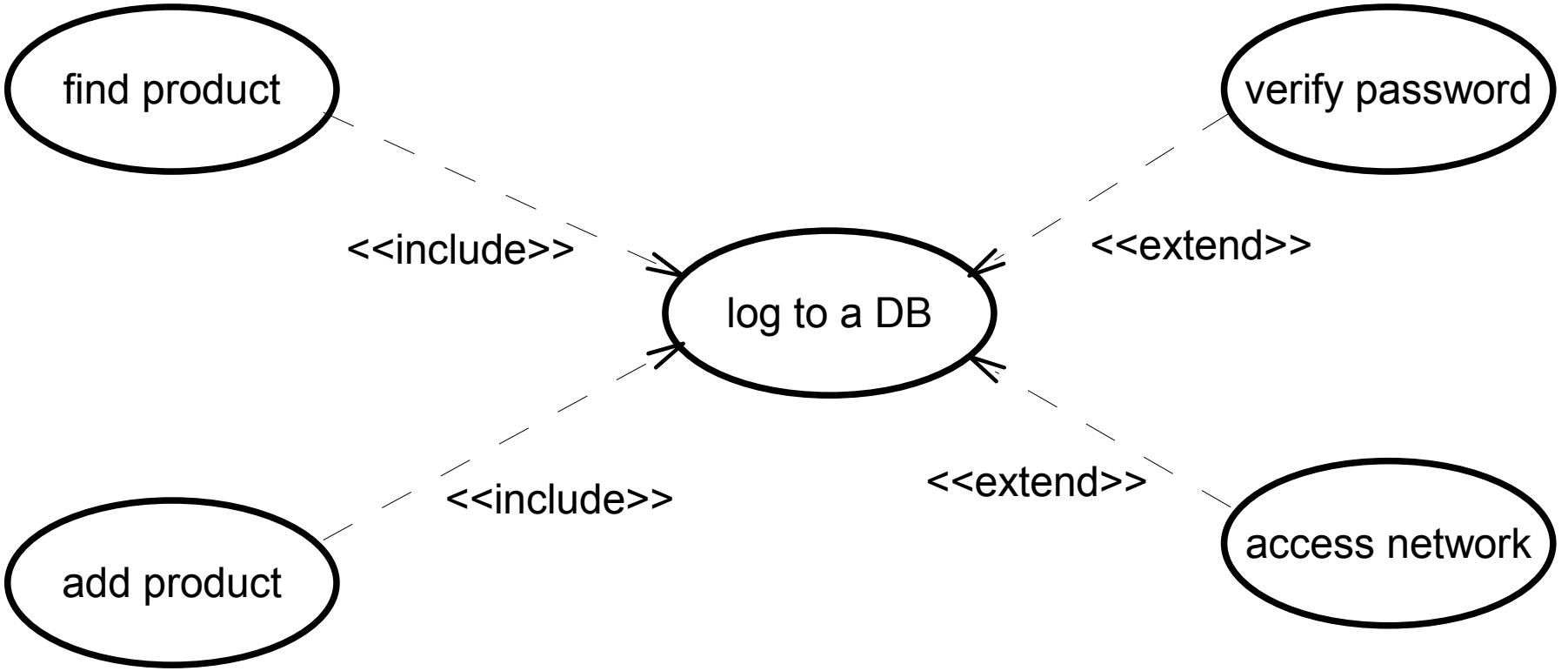
<<include>> dependency



<<extend>> dependency

- Relationship between base case and a case that optionally may introduce additional functionality to the base case
- It is useful when a case may, under certain conditions, rely on some other cases
- The arrow points from the extension to the base

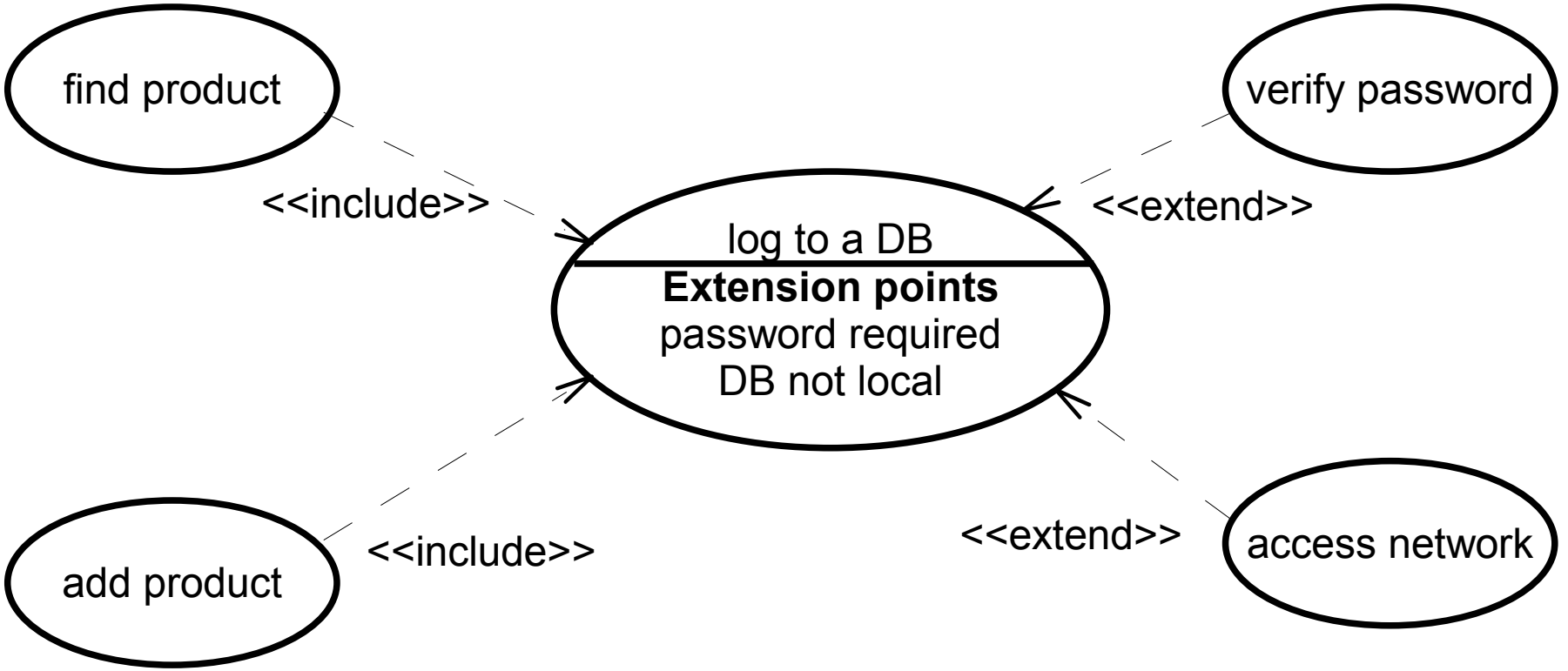
<<extend>> dependency



Extension points

- It is possible to specify situations/conditions under which the extending cases must be included
- They are listed in the extended case, under a horizontal line

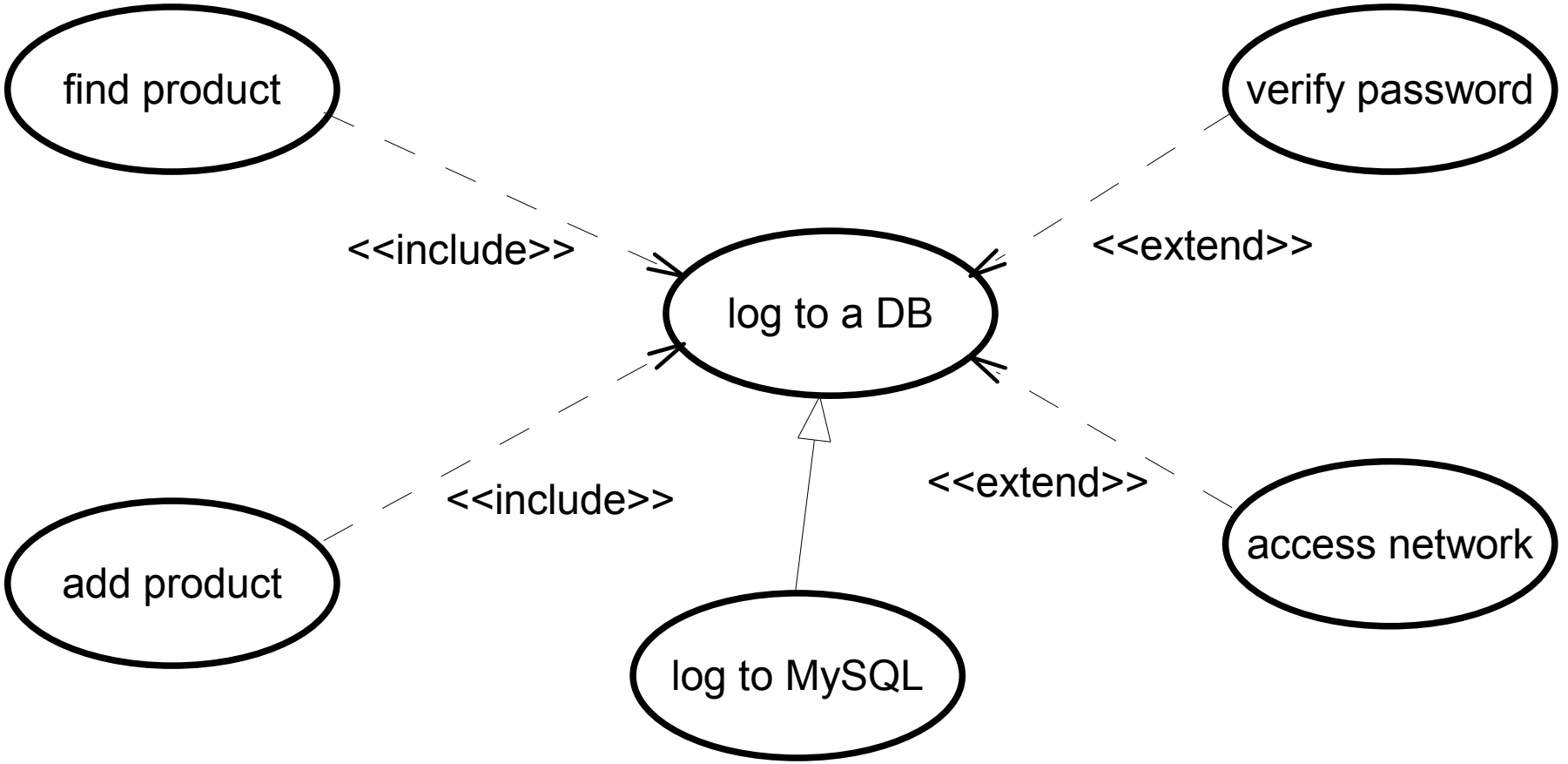
Extension points



Generalisation

- Generalization is a taxonomic relation between general and specialised classifier
- Specialised classifier inherits all features of the general classifier
- It is depicted by a solid-line arrow with triangle head, pointing towards the general classifier

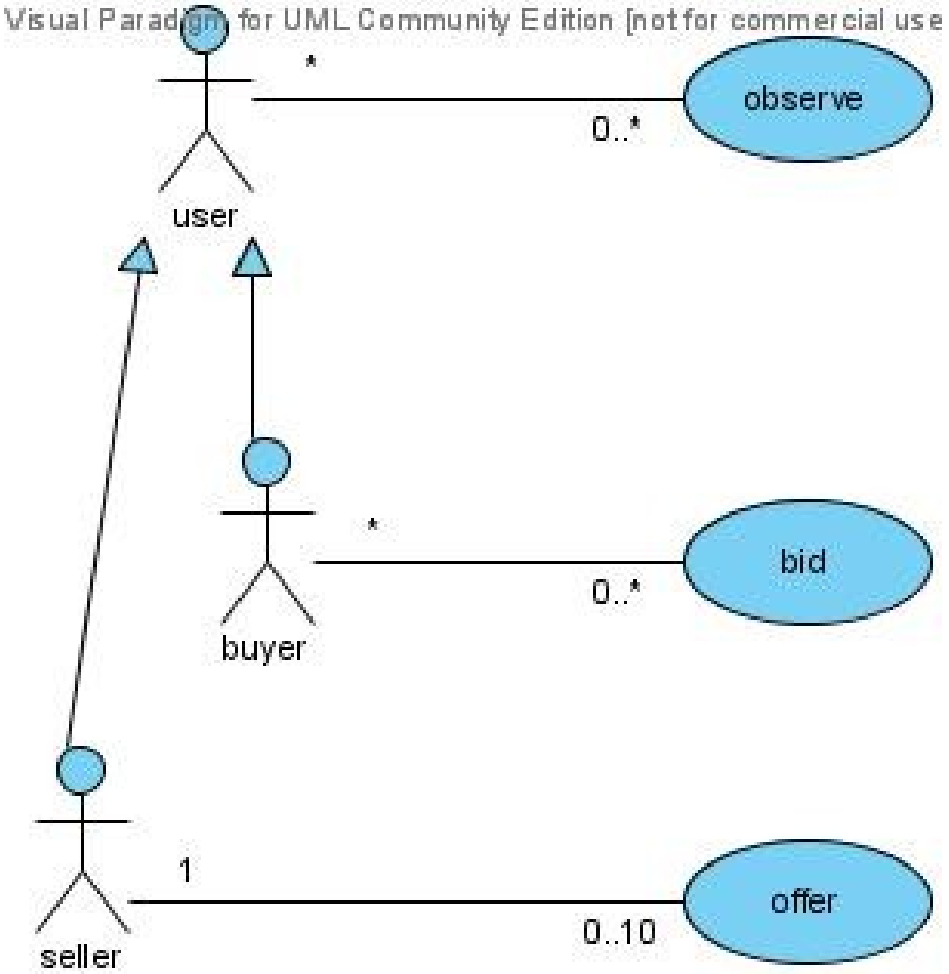
Generalisation



Multiplicity

- Allows to specify the number of items at each end of the association that take place in the association
- Possible cases:
  - n (n > 0) exactly n
  - n..* (n ≥ 0) n or more
  - n..m (m > n ≥ 0) between n and m
  - * many (unknown number)
  - n, m, o..p, q (q > p...) list

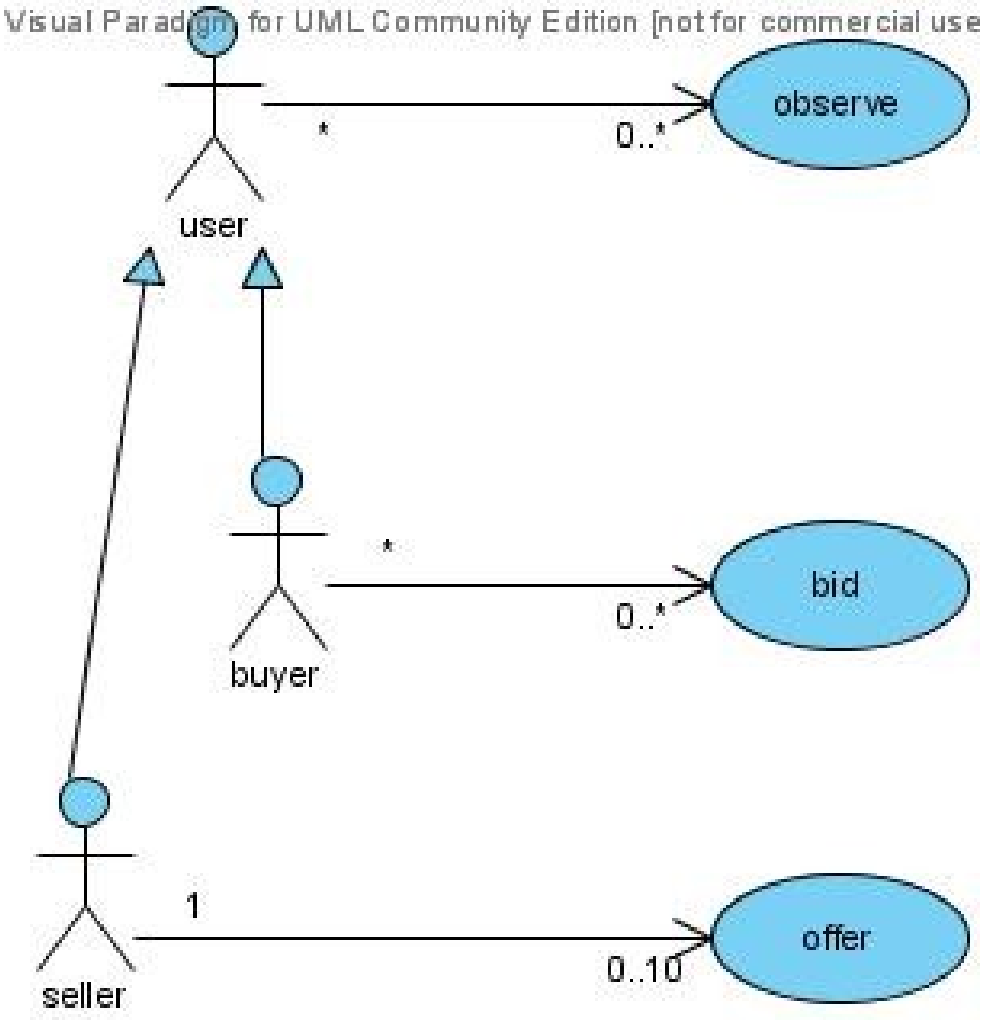
Multiplicity



Navigability

- Bidirectional association can be enhanced by providing information about the side that initiates communication
- This is depicted using an arrow
- The arrow does not indicate direction of data flow

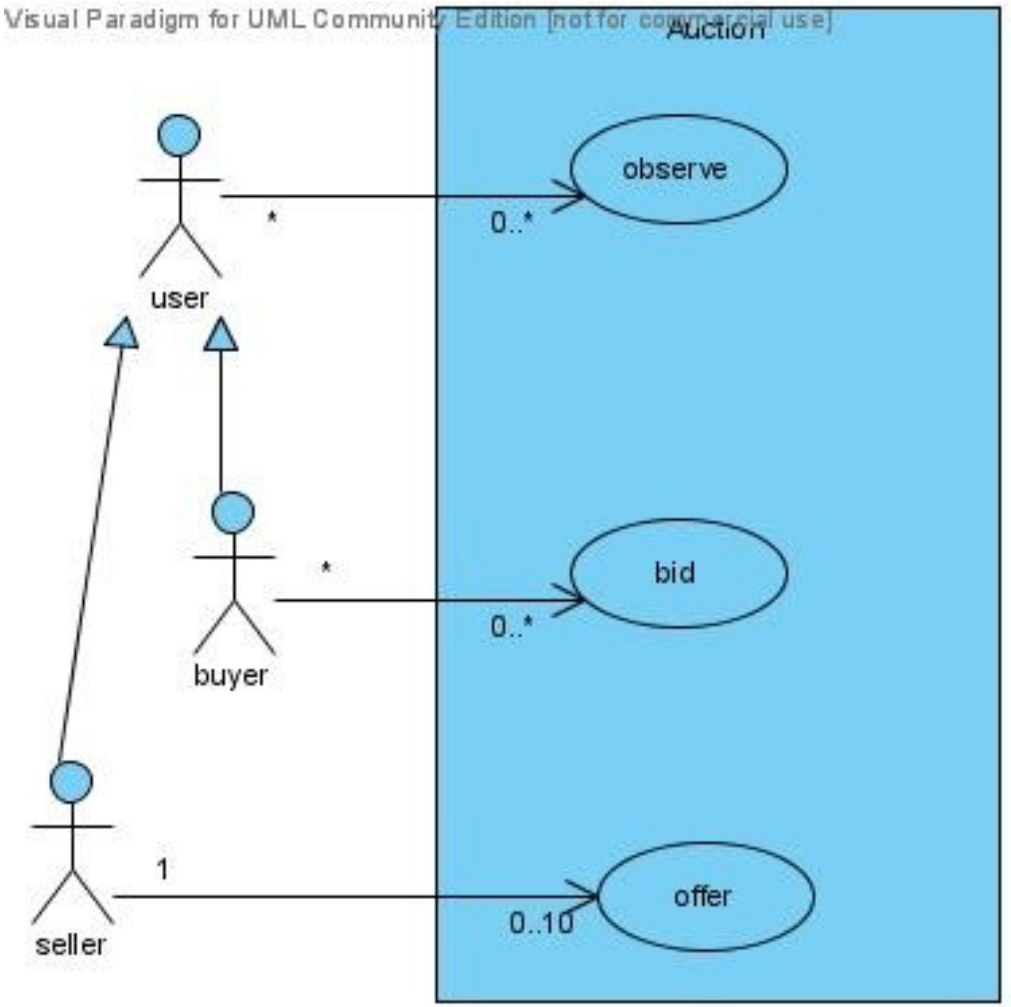
Navigability



System

- The use cases can be grouped to form a complete system
- The grouping is depicted by placing them inside a named rectangle

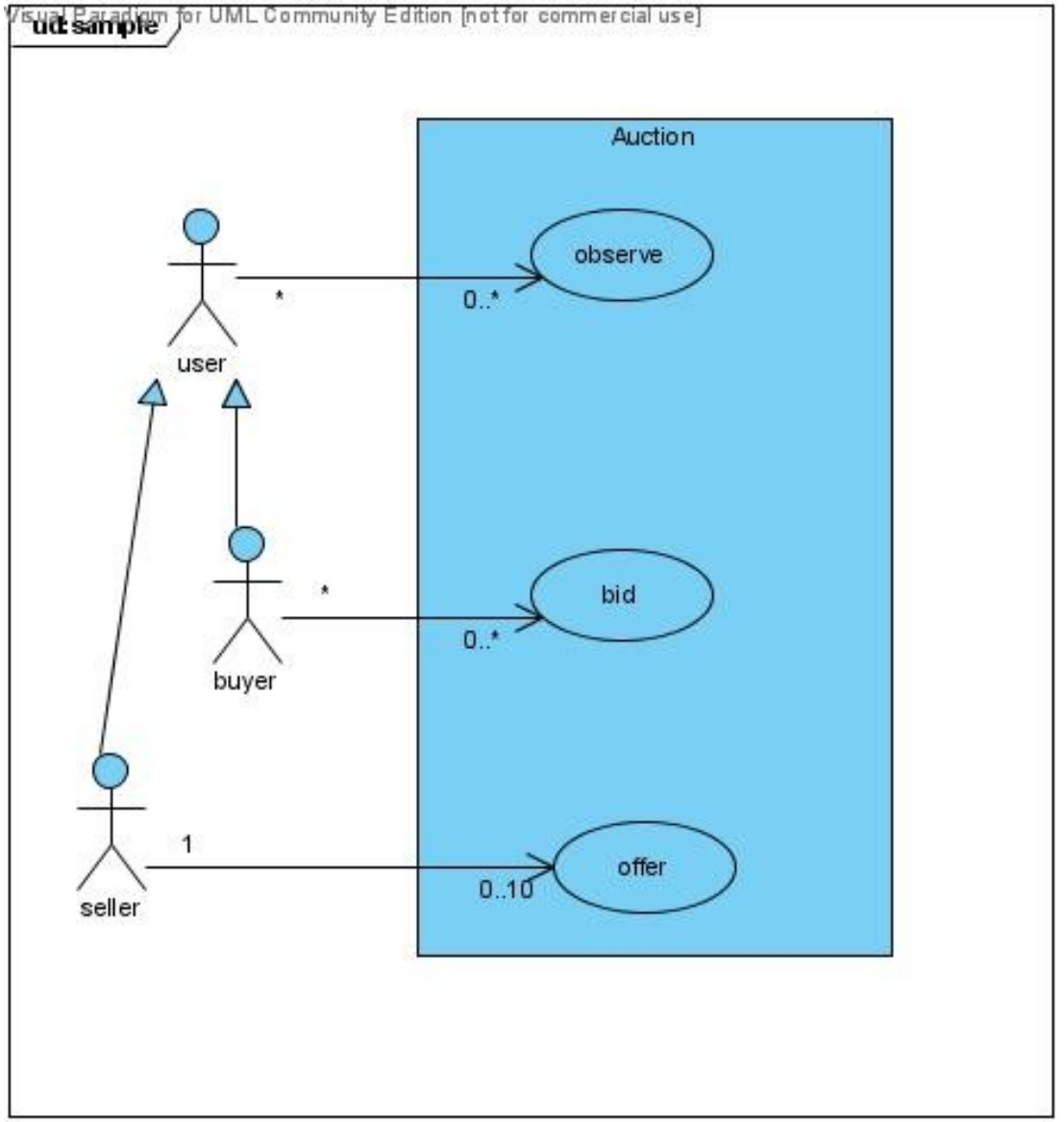
System


Visual Paradigm for UML Community Edition [not for commercial use]
Auction
observe
0..*
*
user
bid
0..*
*
buyer
1
offer
0..10
seller

Frame

- The whole diagram (any type) can be placed inside a frame
- The frame has a header (at the left-top corner) that contains information about diagram name and, optionally, its type and parameters
- Framing improves clarity of documentation when the project is large

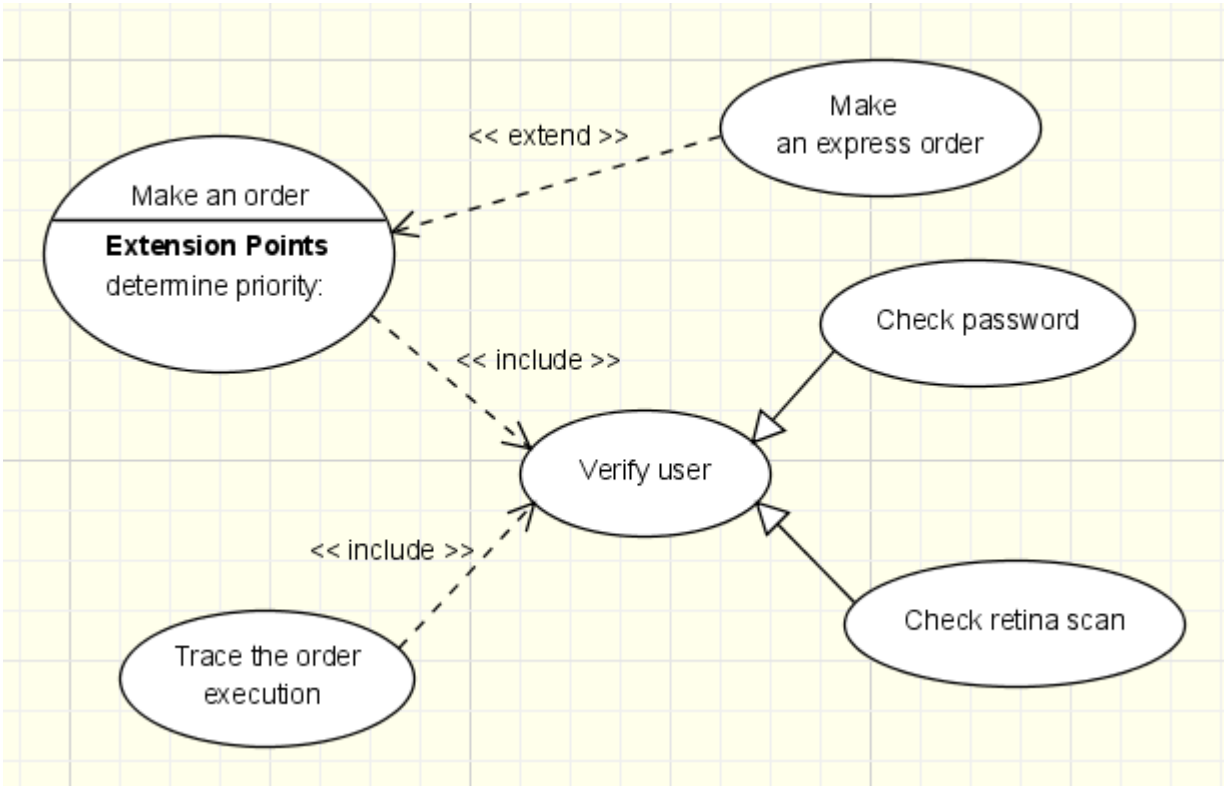
Frame


Visual Paradigm for UML Community Edition [not for commercial use]
uc sample
Auction
observe
0..*
user
*
buyer
*
bid
0..*
seller
1
offer
0..10

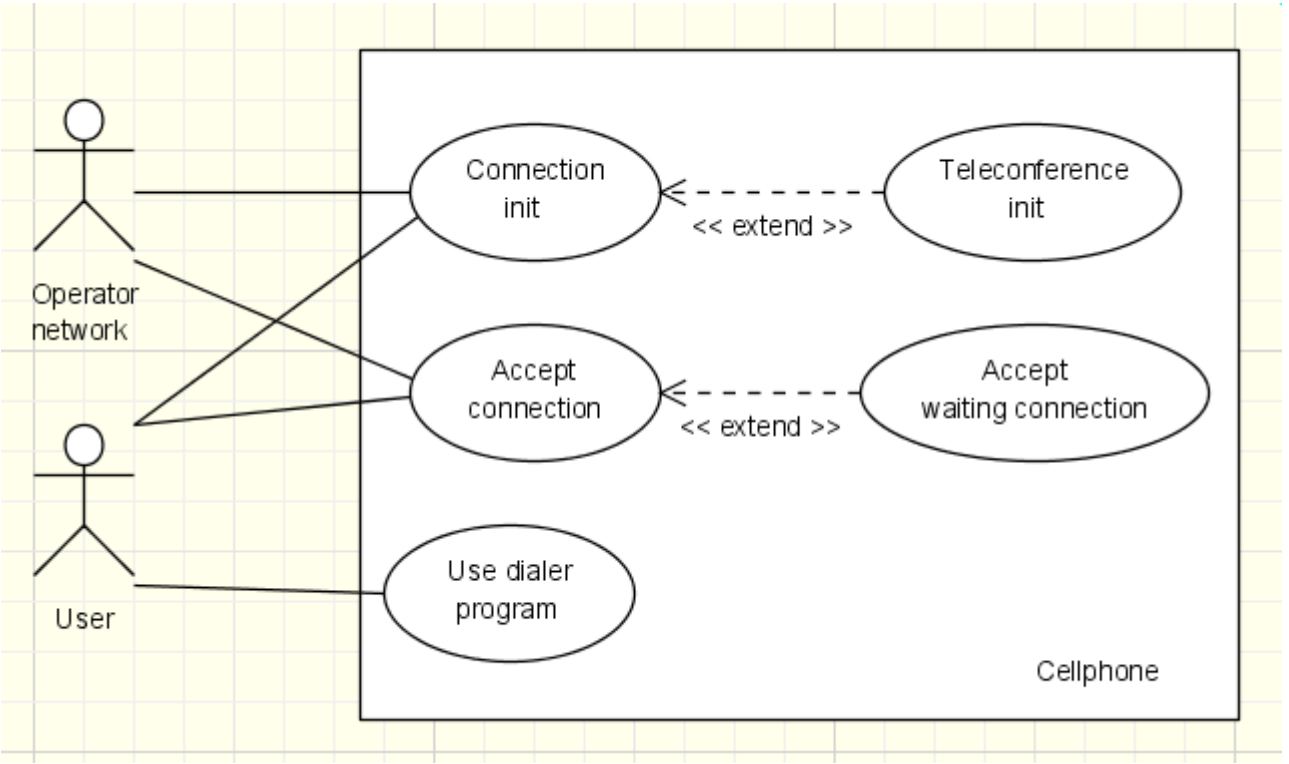
Documenting use cases

- Use case diagram is very general
- In order to clearly define intended behaviour of the system, each use case should have additional information, called scenario
- Scenario is a sequence of actions documenting behaviour
- In complex cases it is possible to define main scenario and alternative scenarios
- Scenario can be written down as natural language text, pseudo-code, table etc.

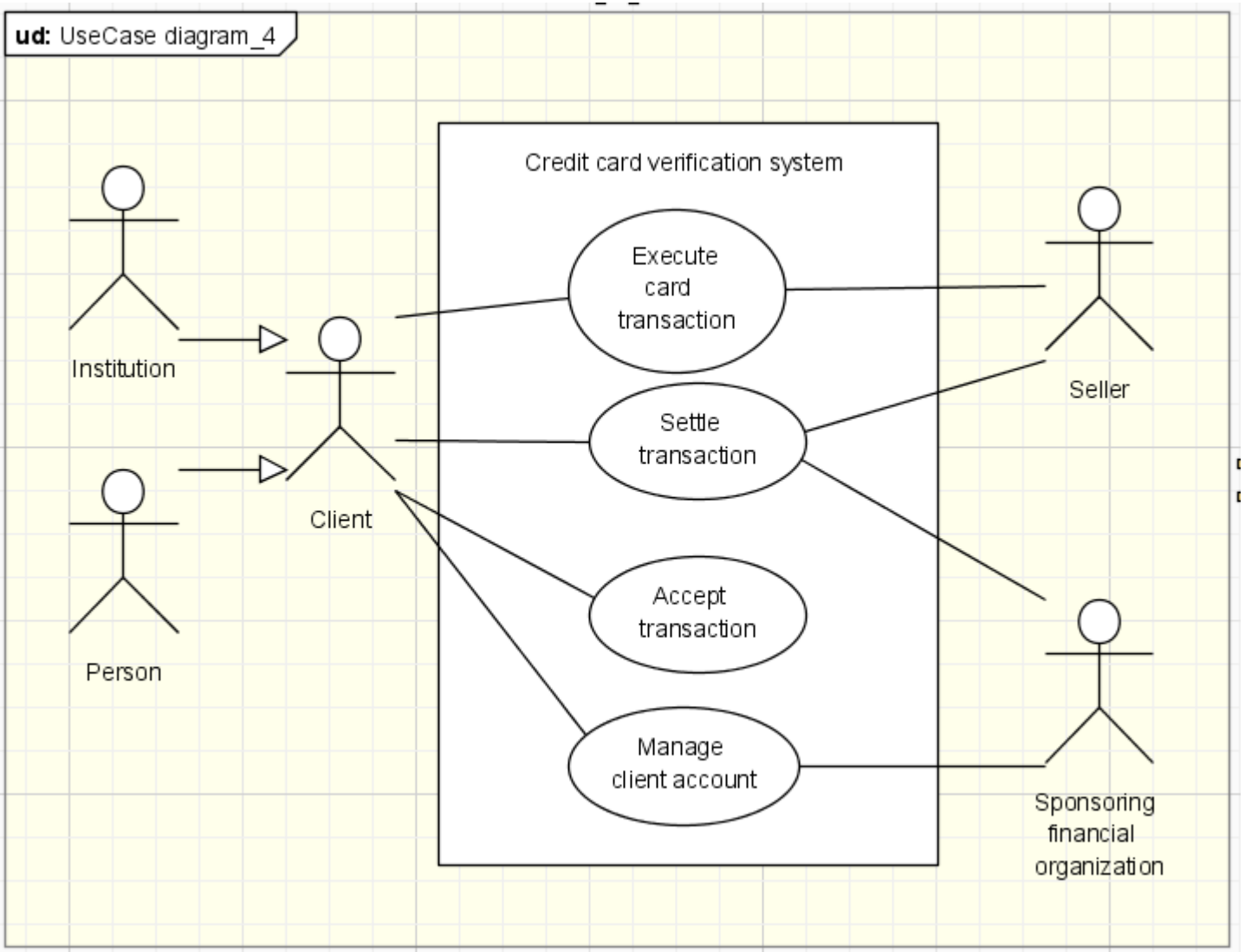
Sample diagrams



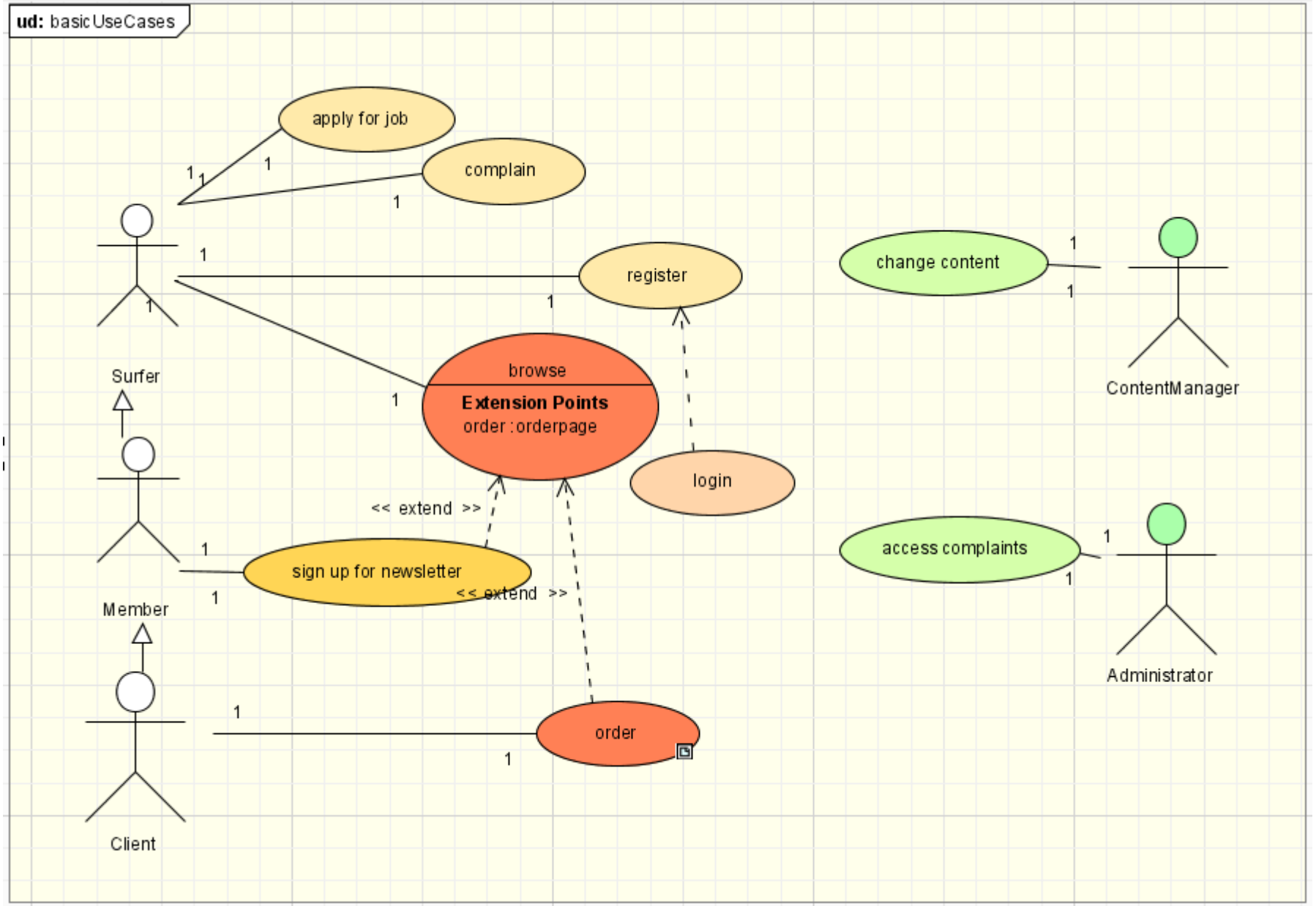
Sample diagrams



Sample diagrams



Sample diagrams



Class diagrams

- Contain information about static elements (classes) and relationships between them
- They are very closely related to the object-oriented programming technique
- Are among the most important UML diagrams

Class symbol

- The symbol of a class is a rectangle, usually divide by horizontal lines into three sections:
  - name
  - attributes
  - operations
- If needed, this can be expanded with additional sections (e.g. exceptions)

Class symbol

- For complex classes displaying all attributes and operations may take too much space
- Possible solutions are:
  - displaying only class name without attributes and operations sections
  - displaying only class name with empty attributes and operations sections
  - displaying part of attributes and operations sections, denoting continuation of the list by ellipsis
  - hiding (some) operations / attributes

Visibility of class members

- It is possible to specify visibility of attributes and operations
- Visibility translates to access control of object oriented languages
- Possibilities are:
  - \+ public
  - \- private
  - # protected
  - ~ packet

- Other possibilities, better suited for some languages, can be used

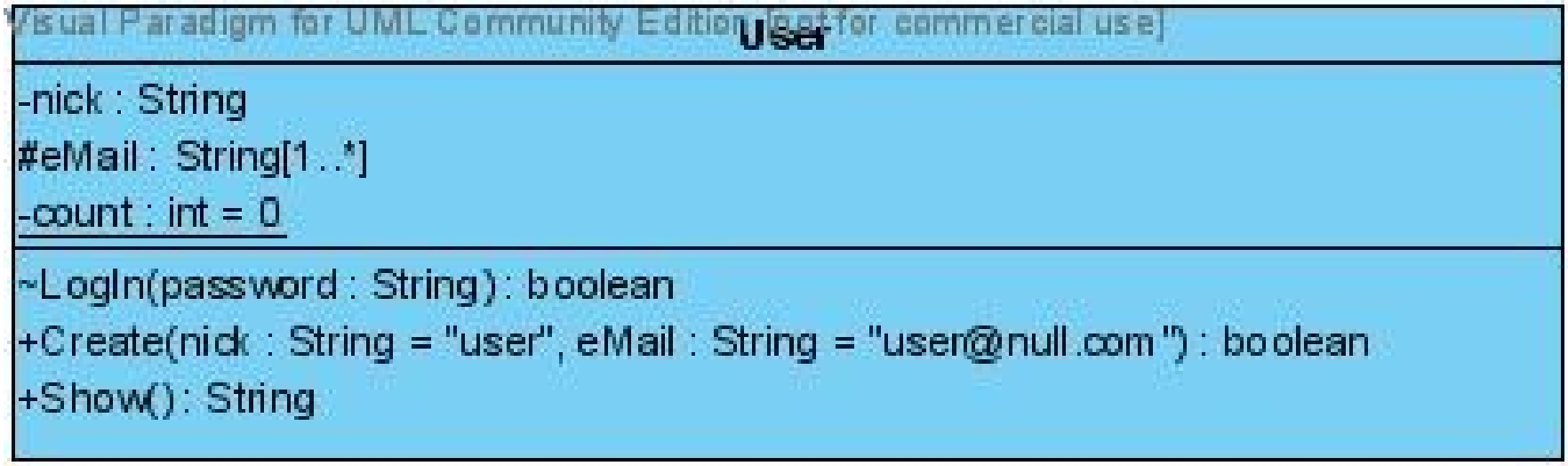
Static members

- Members can be declared as static
- Concept is identical to the idea of static members in object-oriented programming
- Static members are depicted by underline

Members specification

- For attributes it is possible to specify:
  - type. The type is placed after attribute name, separated by a colon
  - count
  - initial value
- For operations it is possible to specify:
  - return type. The type is placed after operation name, separated by a colon
  - arguments. Each argument can be specified just as attribute, with the addition of direction of passing the argument (“in” is the default direction)

Members specification



Relationship

- All 4 types of relationship are used
- The main type is association
- Association can have the following features (bold are new compared to use cases):
  - **name**
  - **roles**
  - navigability
  - multiplicity
  - **aggregation**

Association name

- It is possible to name an association in order to provide more detailed information
- The name can also contain direction

Association roles

- Roles are another way to provide more detailed information about association
- Role of a class is described by text placed close to the class symbol
- It is possible to specify both association name and class roles

Association navigability

- Default navigability is bidirectional
- To specify unidirectional navigability, the association is depicted with an arrow at the end
- In the class diagram unidirectional navigability means the **communication** is unidirectional (cf. use case diagrams)

Association multiplicity

- The same meaning as in the use case diagrams

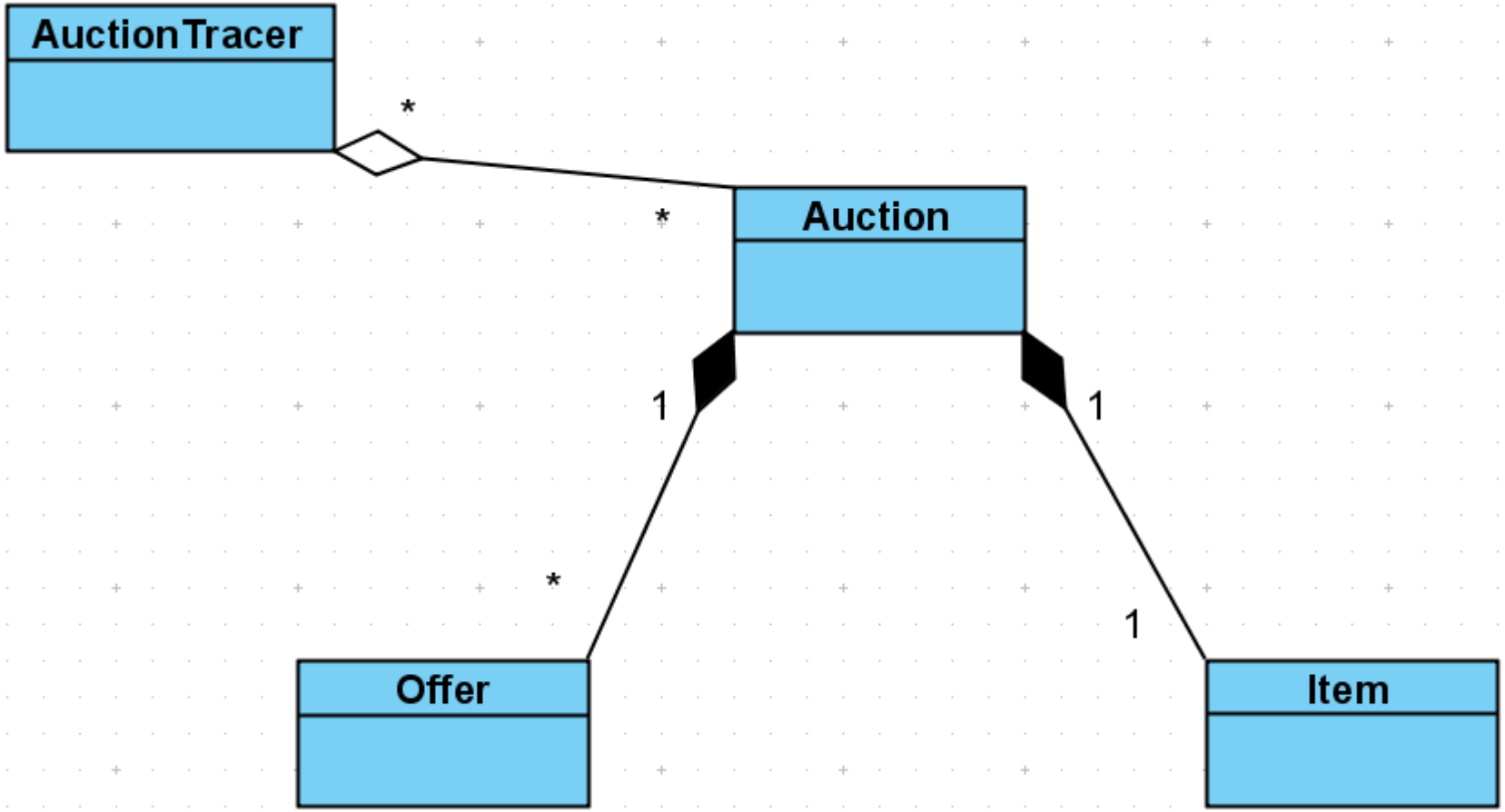
Association aggregation

- Aggregation describes relationship between the whole and the part
- There are two kinds of aggregation:
  - complete (composition, strong aggregation)
  - partial (aggregation, weak aggregation)
- Aggregation is depicted by a parallelogram placed next to the symbol representing the whole
- Strong aggregation is depicted by a solid parallelogram, weak - by hollow

Strong and weak aggregation

- In case of strong aggregation the contained objects cannot exist if the containing object is removed
- The concept is identical to the situation in object-oriented language when one class contains objects of another class
- In case of weak aggregation the contained objects can exist without the containing object
- Furthermore, one object can be contained by many other objects
- This concept is identical to the situation when one class contains pointer (reference) to object of another class

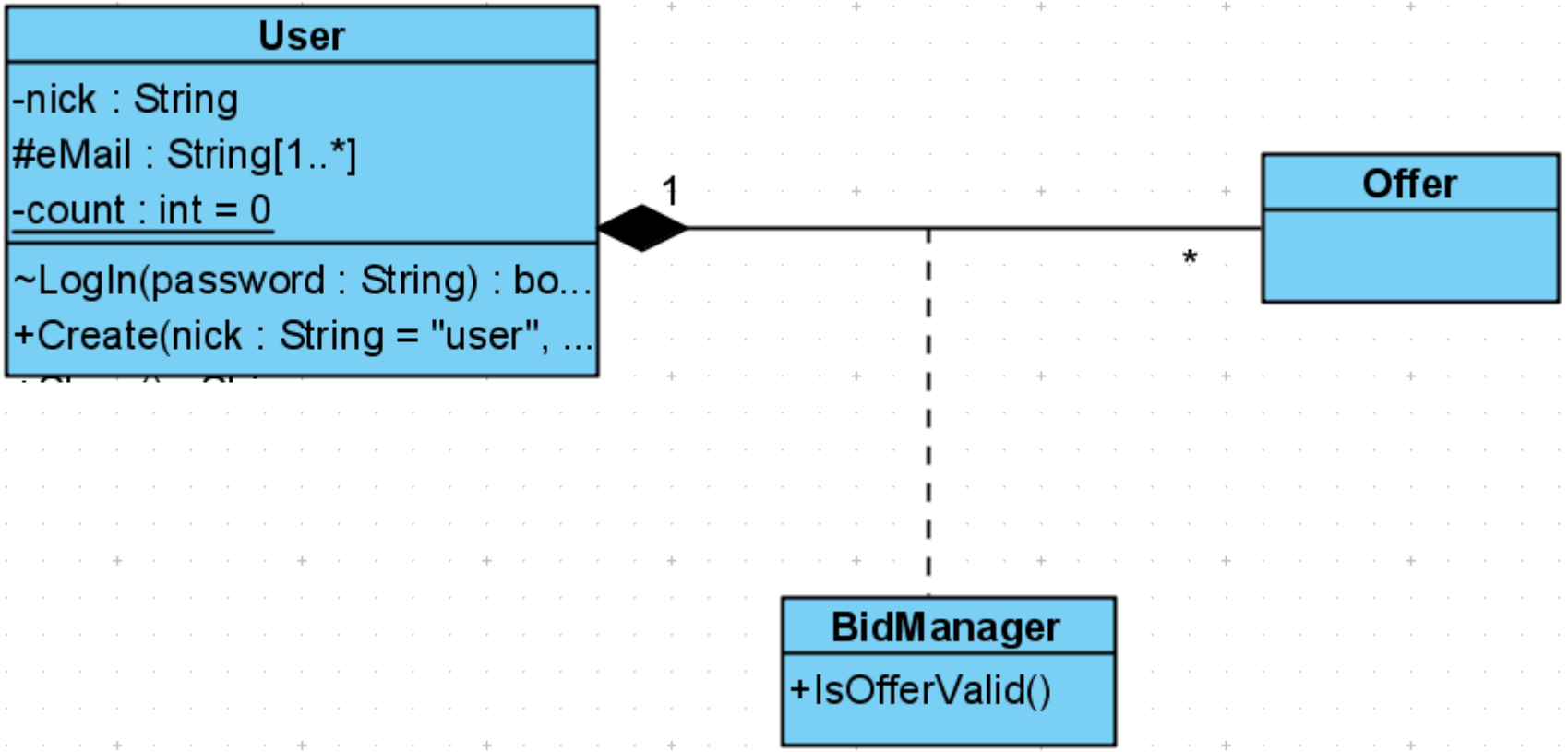
Strong and weak aggregation



Association class

- Can be used to precisely describe relationship between classes
- It is depicted by a class placed close to the association and connected wit the association by a dashed line

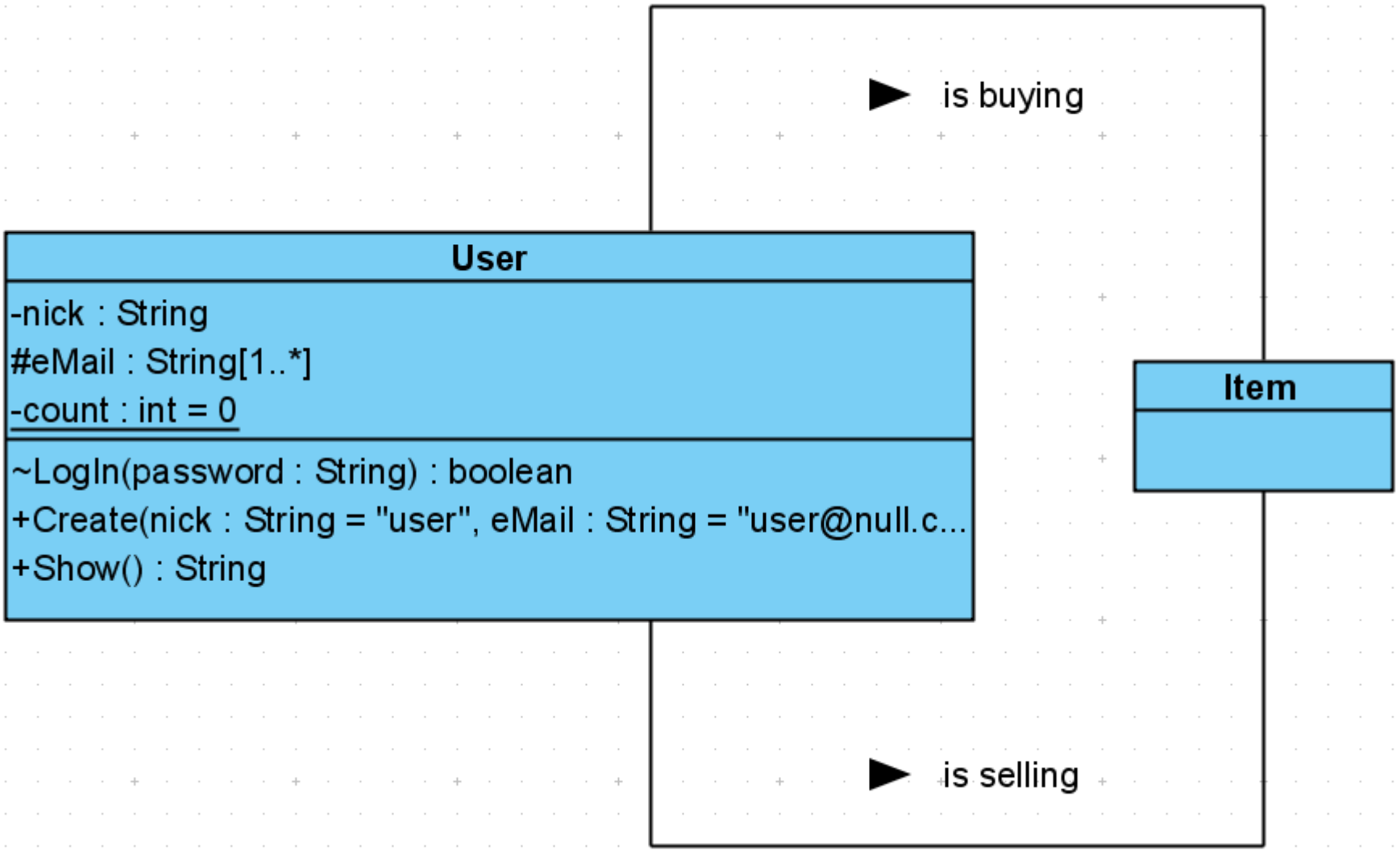
Association class



Multiple associations

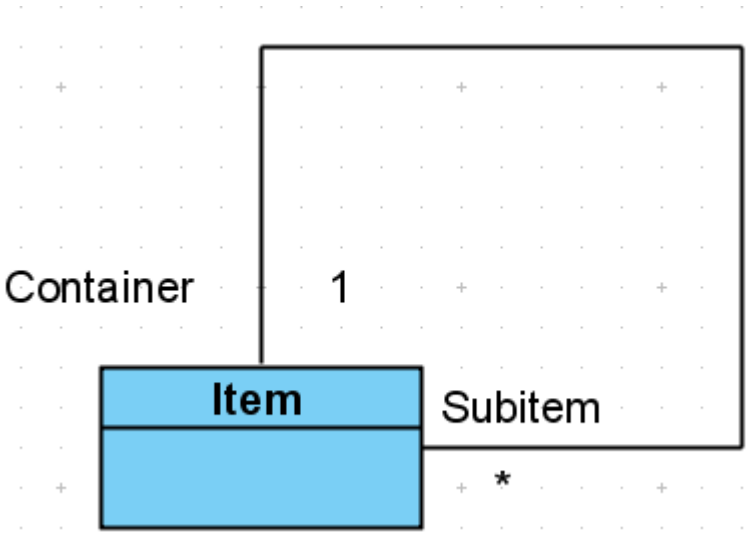
- Two classes can be differently related to each other in different contexts
- This may result to more than one association between classes
- Every one of the multiple associations should be named

Multiple associations



Self association

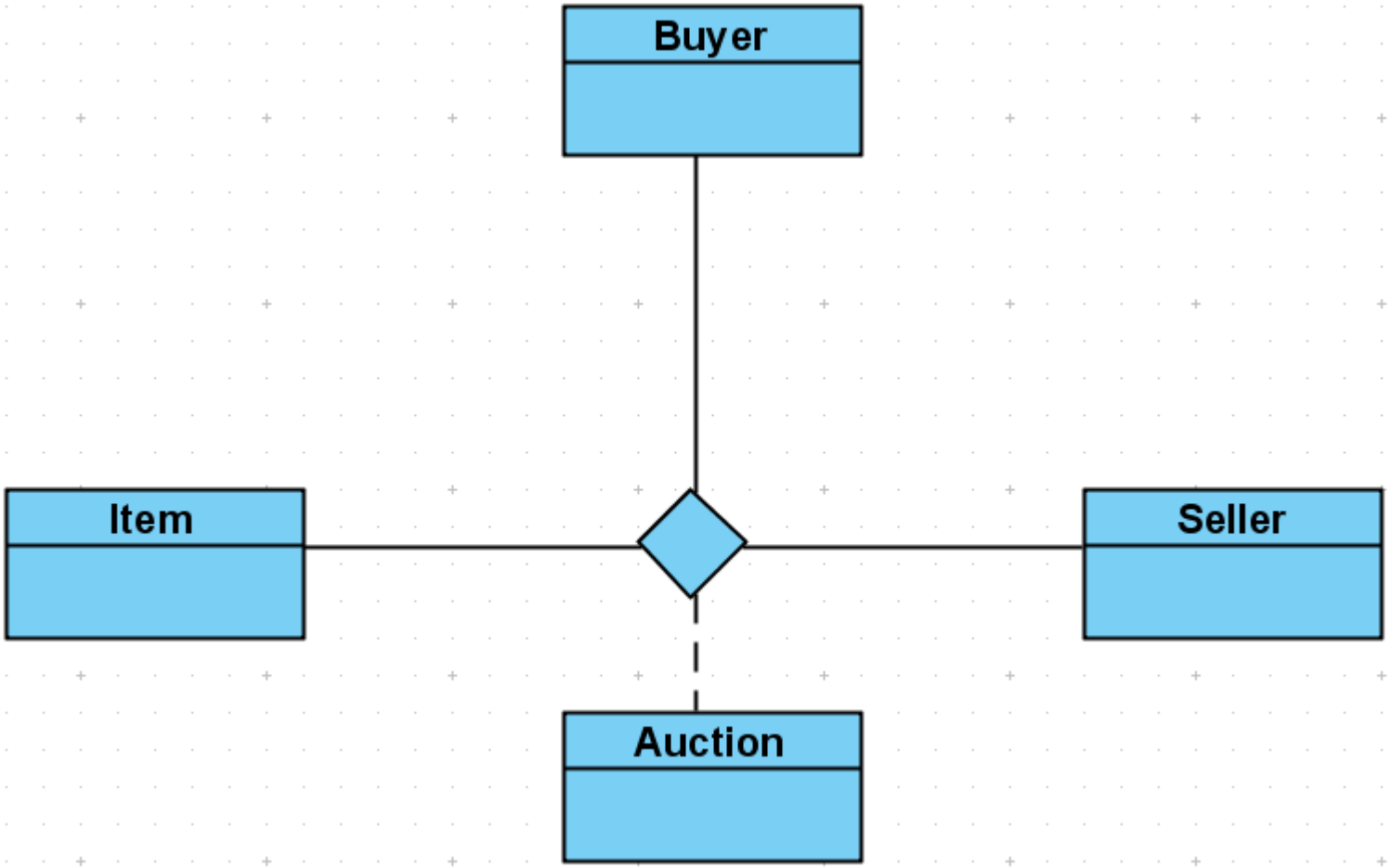
- It is possible to make an association that relates the class to itself



N-ary association

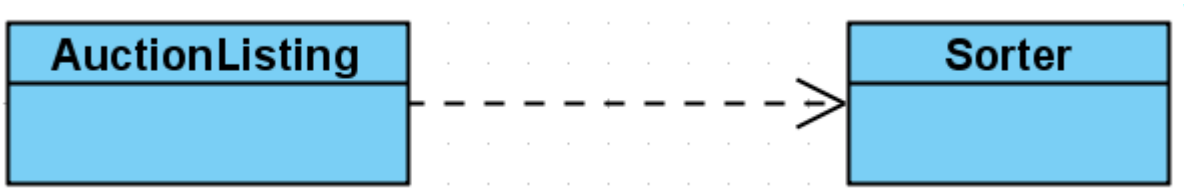
- Is is possible to specify association between more than two classes
- Such an association may also contain an association class
- It should not be confused with multiple association

N-ary association



Dependency

- This relationship means that one class (client) makes (some sort of) use of another class (supplier)
- It is depicted using a dashed line with arrow pointing from the client to the supplier



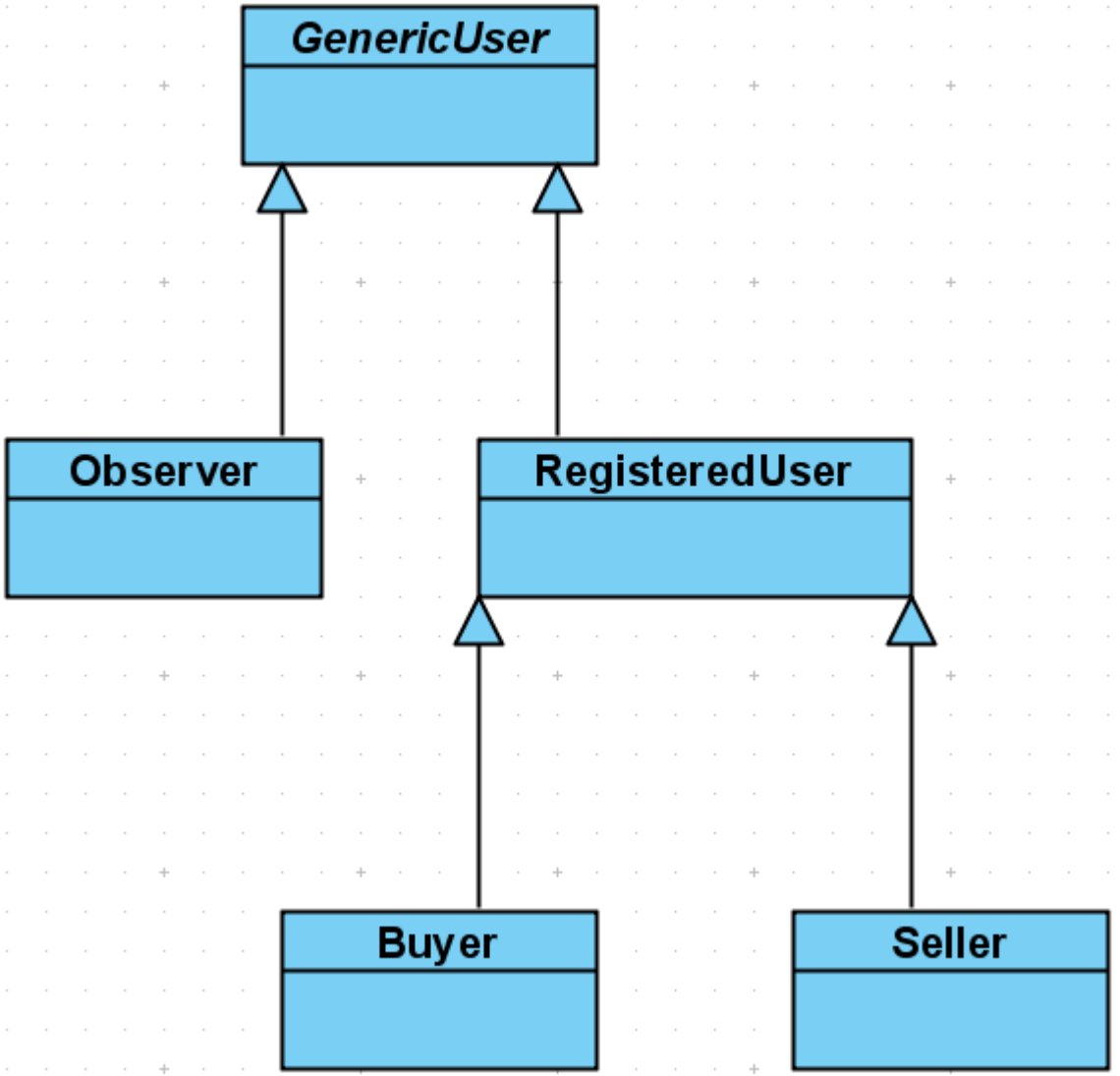
Generalisation

- Generalisation is commonly used in class diagrams
- There are several concepts that enhance the idea of generalisation to precisely describe this relationship

Abstract classes

- Abstract classes do not have instances (objects)
- The concept is identical to the abstract classes in object-oriented programming
- Abstract classes are denoted by names in italics

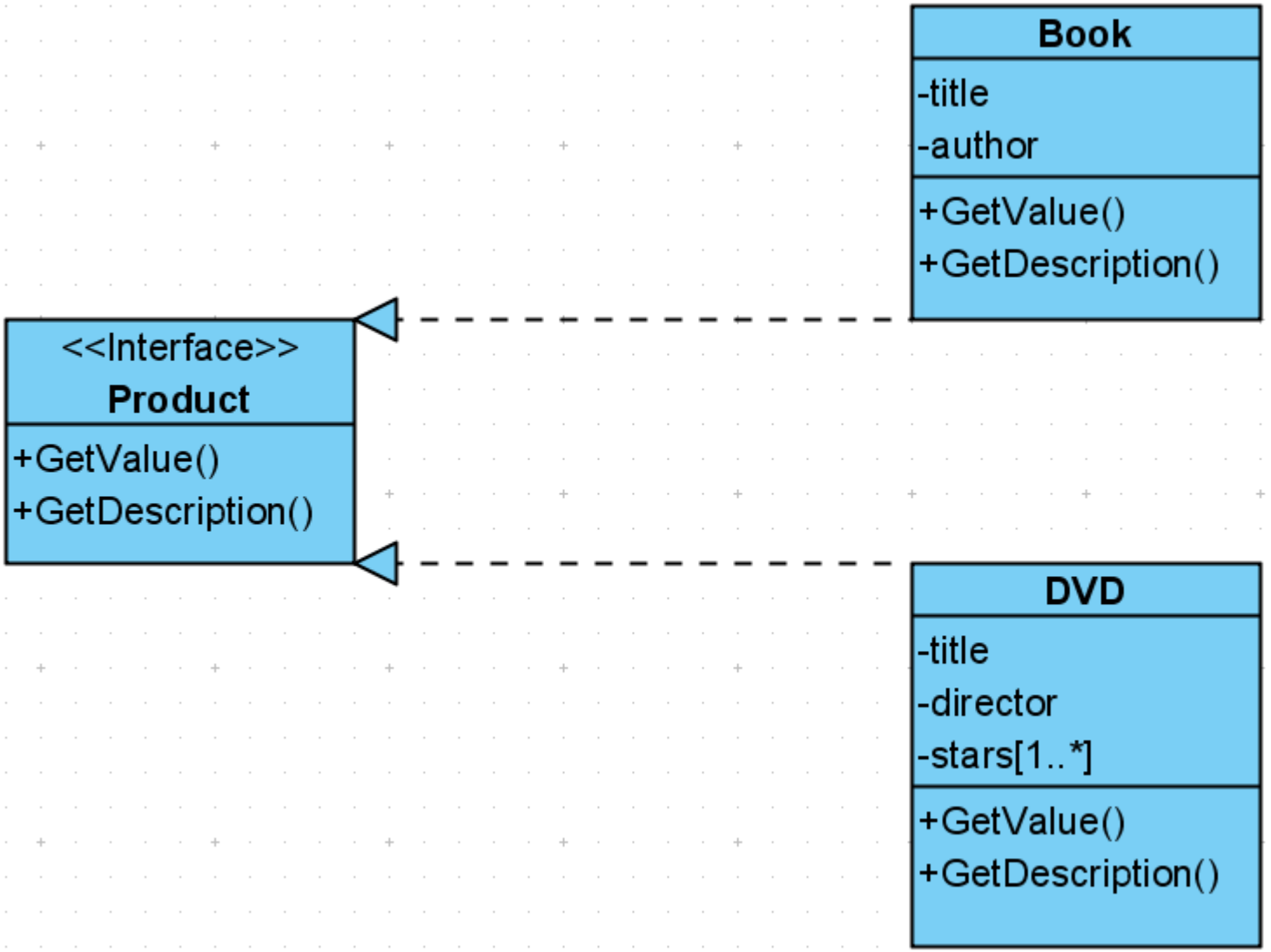
Abstract classes



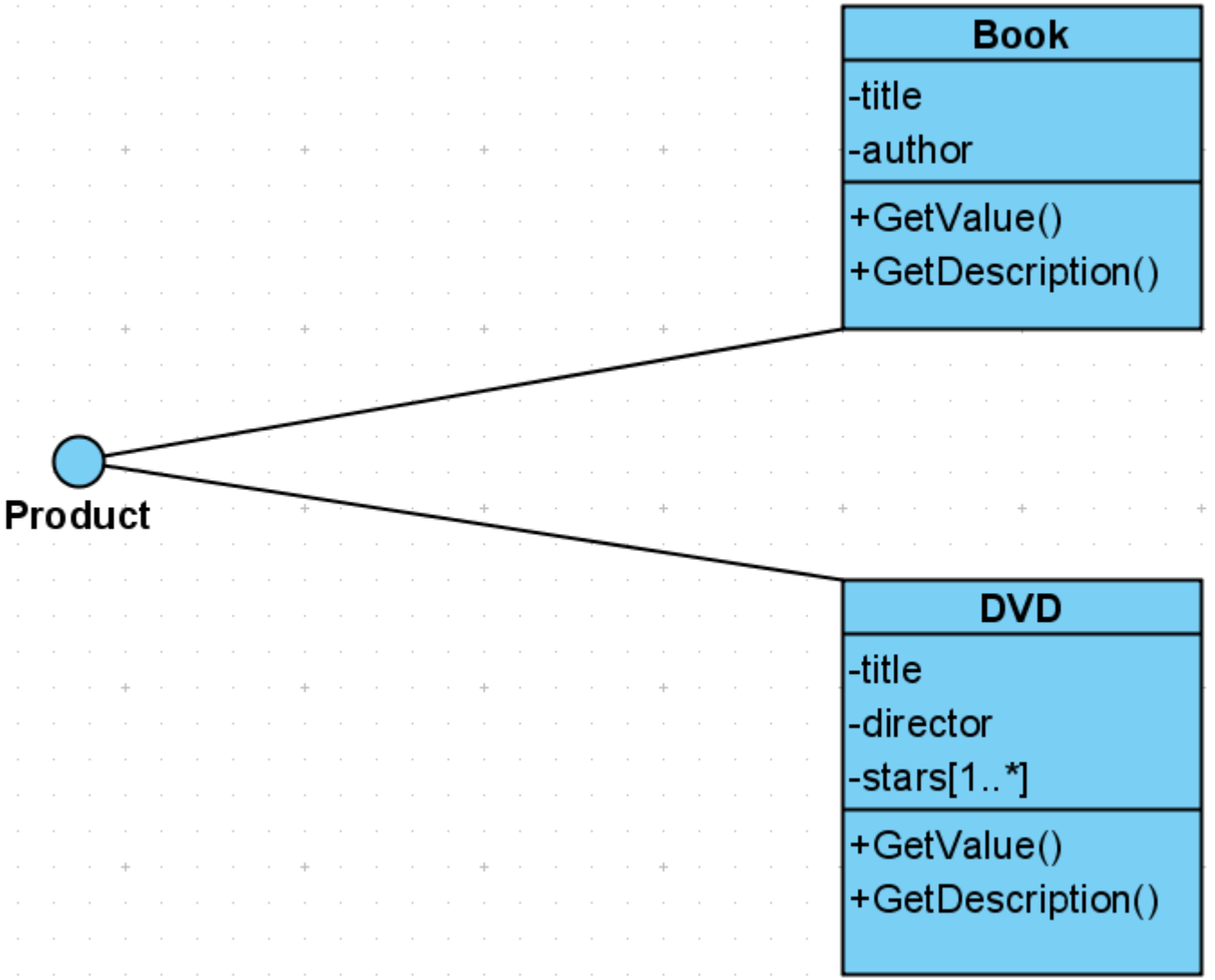
Realisation

- In class diagram this is a relationship between an interface and its implementation
- This is identical to the concept used in object-oriented languages
- Realisation is depicted by a dashed line with hollow arrow pointing from the class to the interface
- The interface can be displayed either as a rectangle with operations (similar to the class), or as a ball

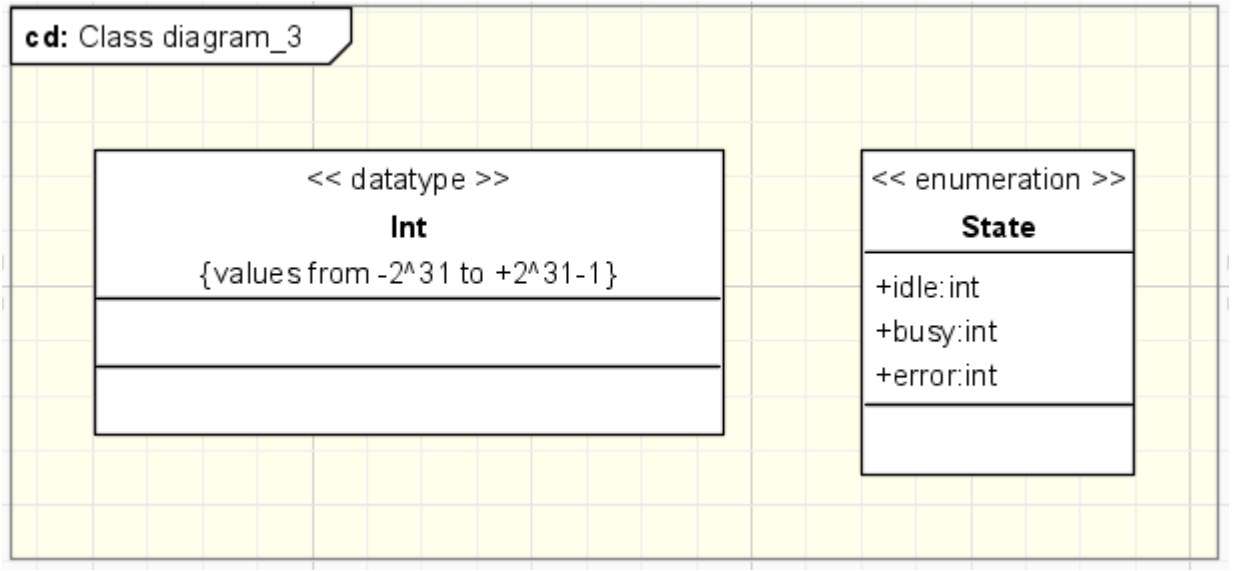
Realisation



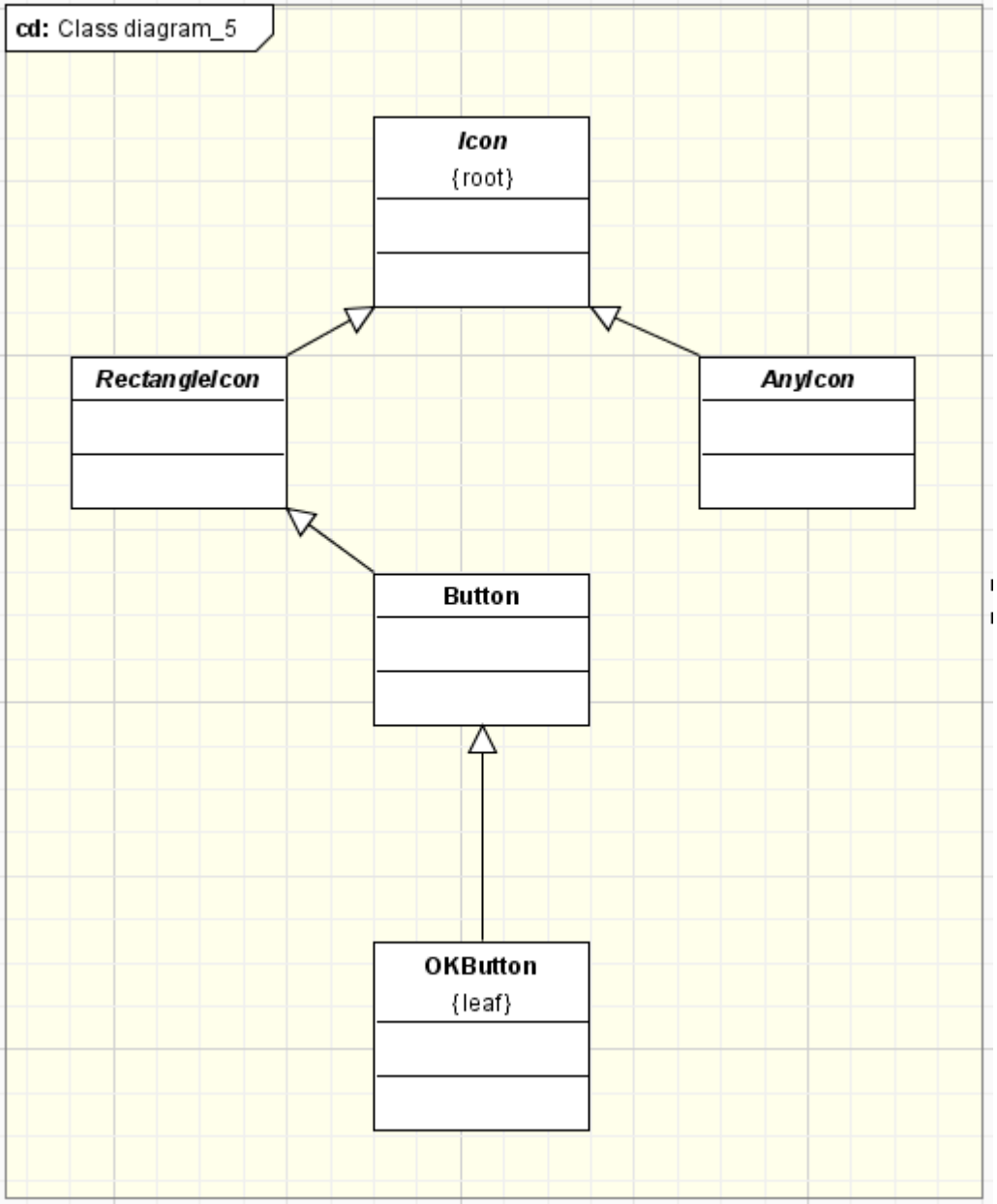
Realisation



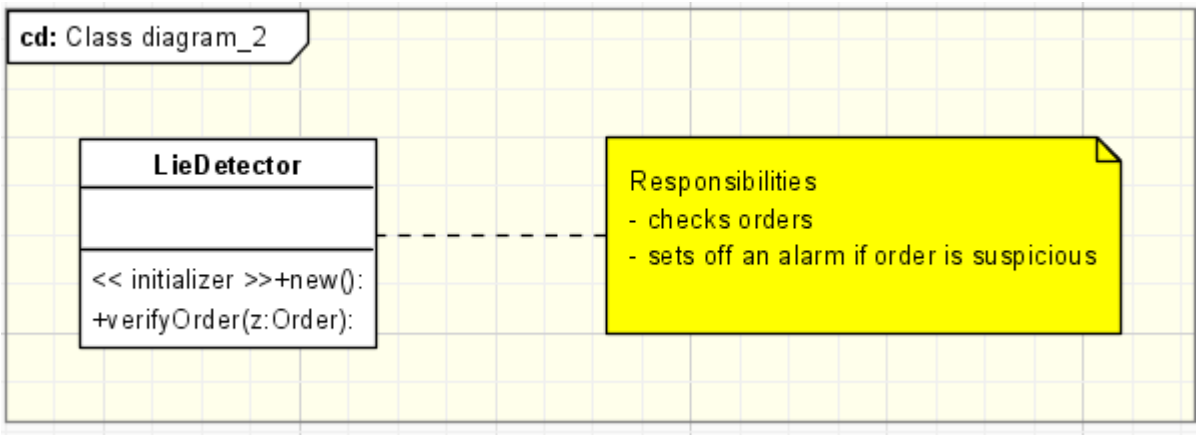
Sample diagrams



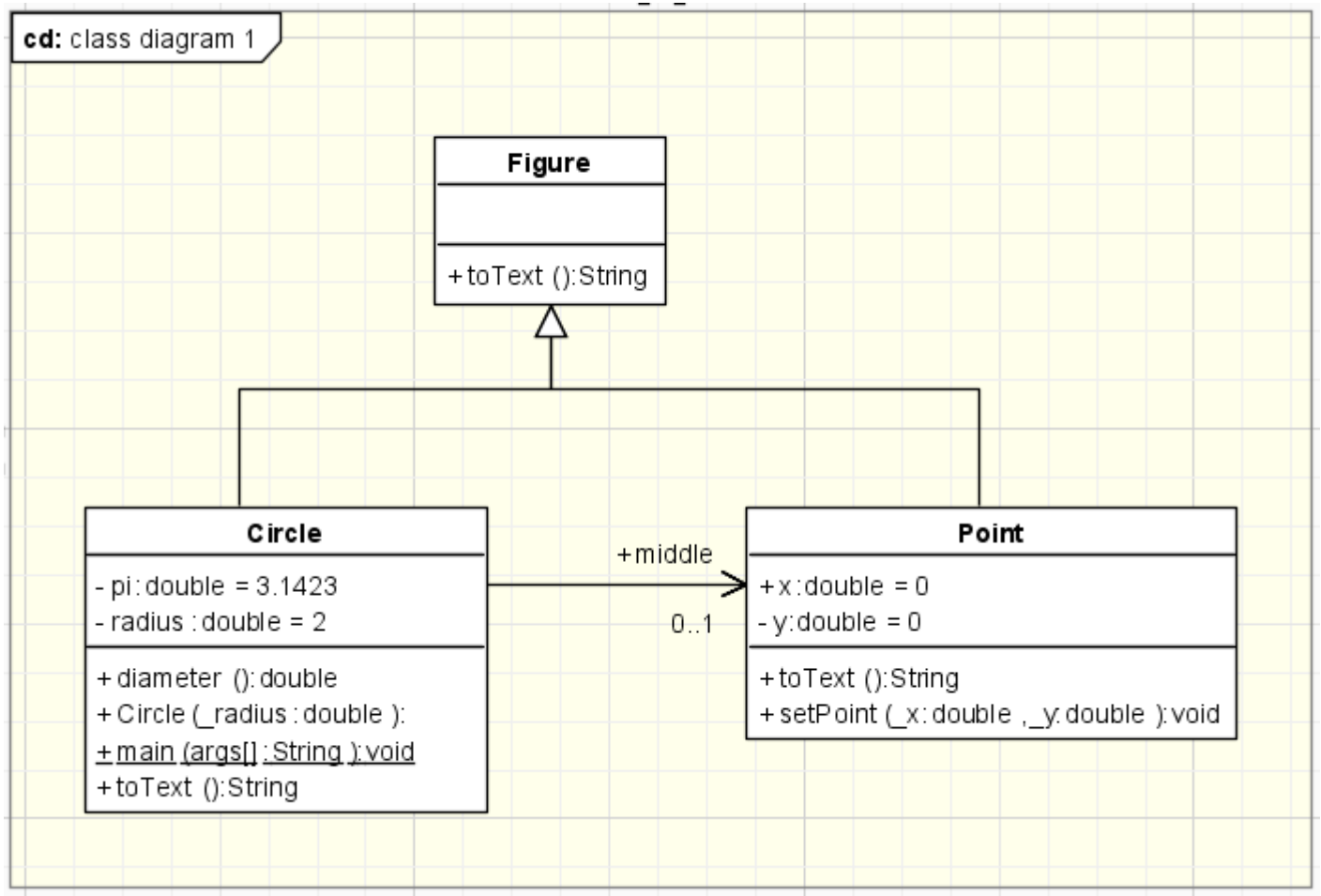
Sample diagrams



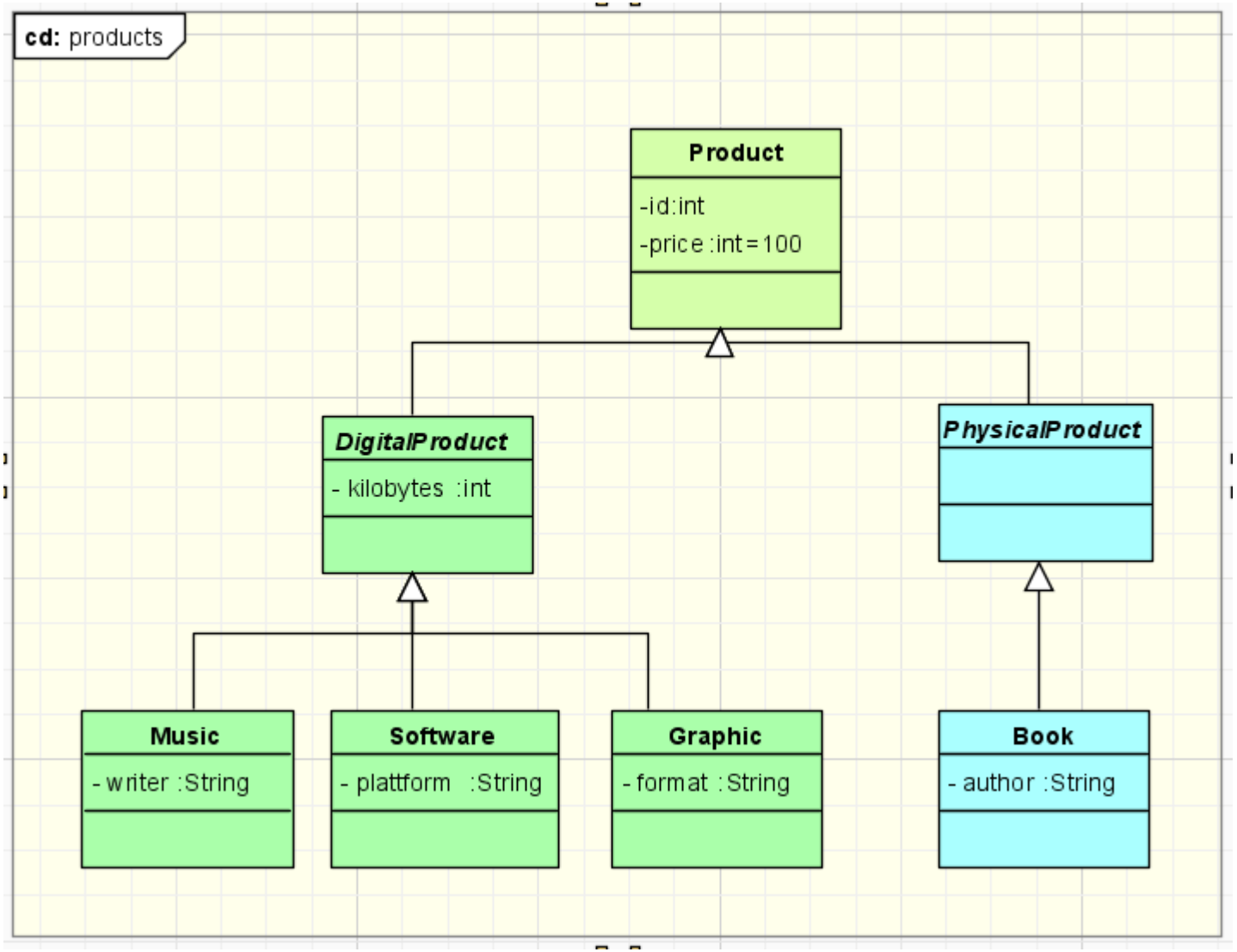
Sample diagrams



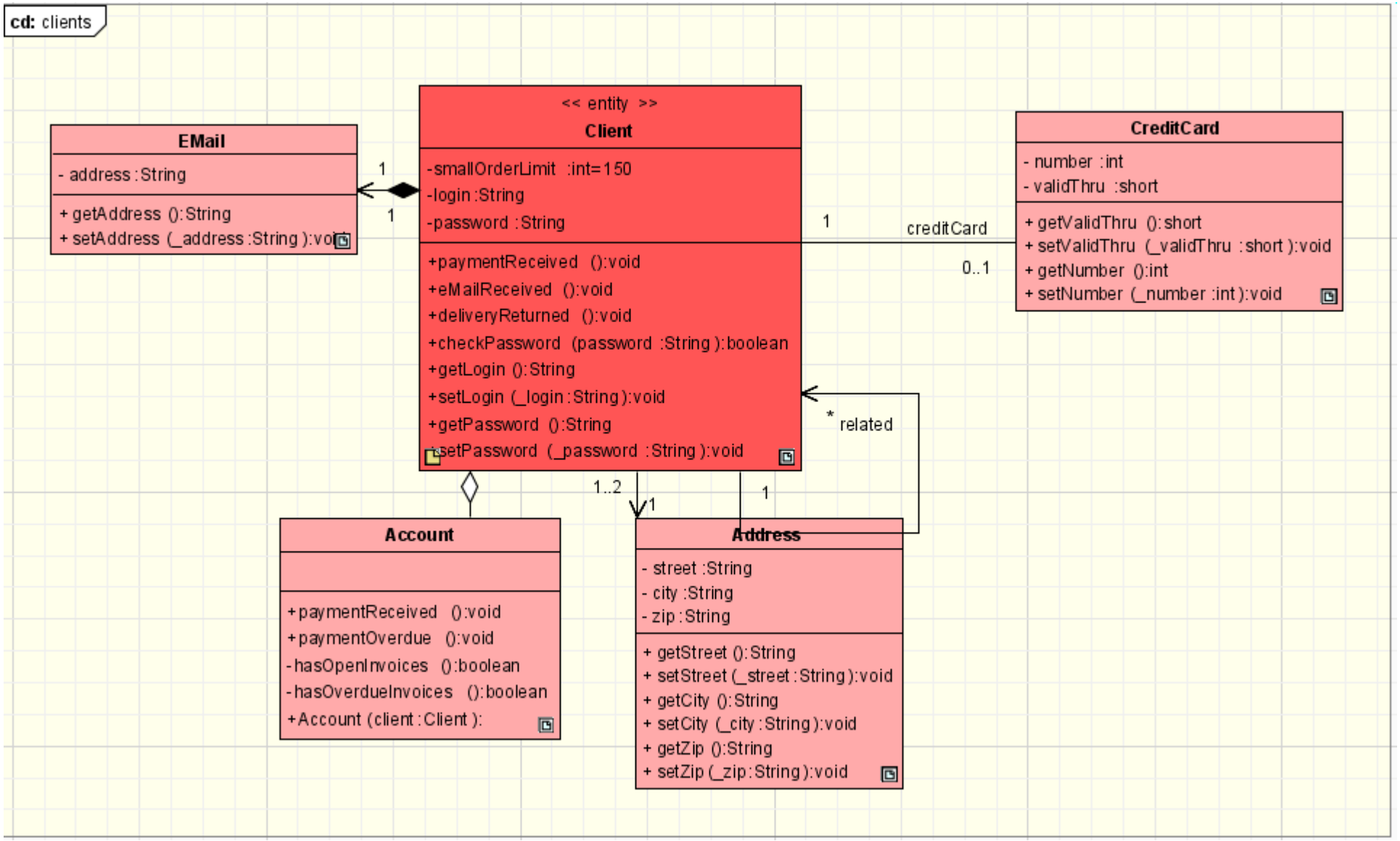
Sample diagrams



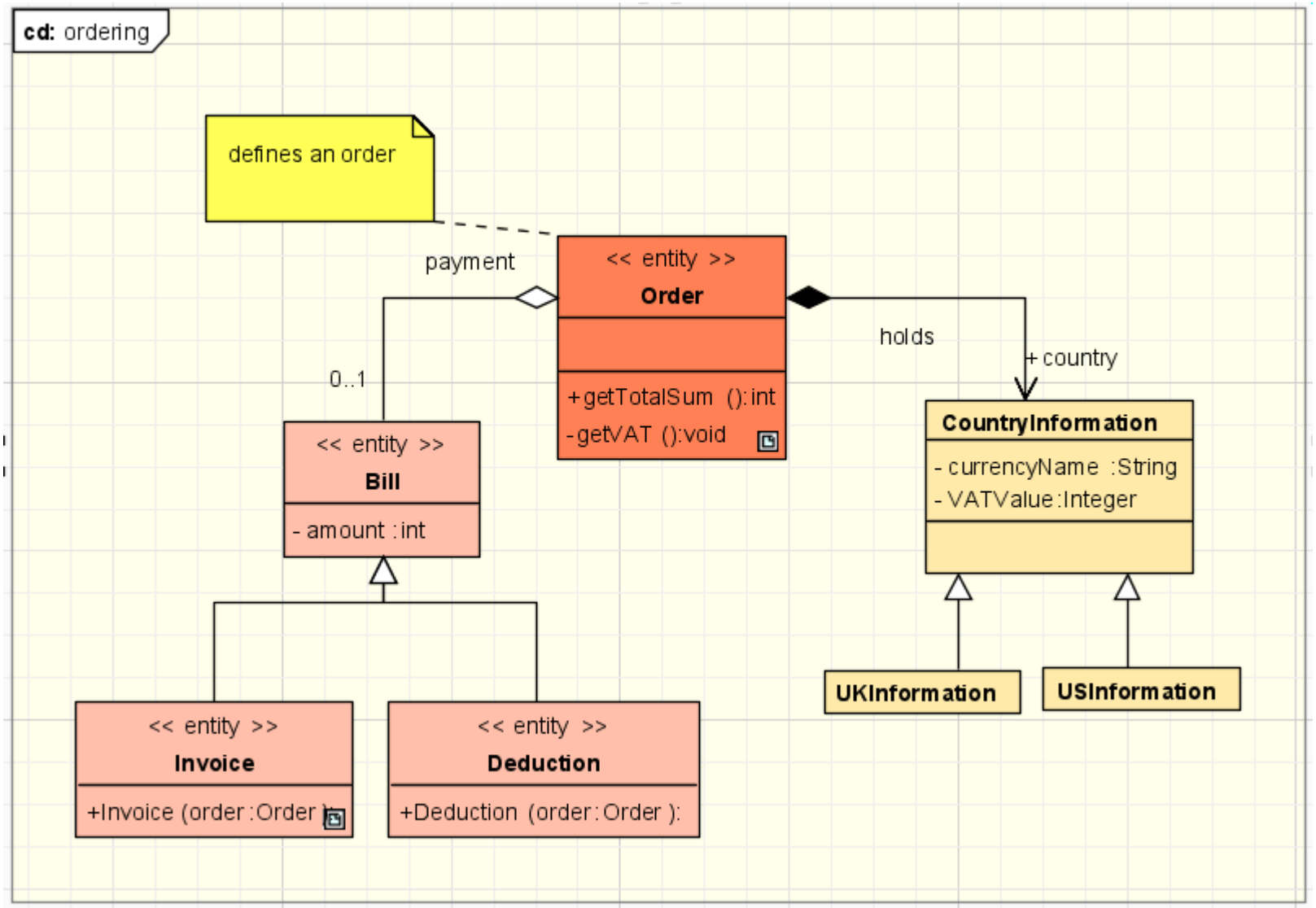
Sample diagrams



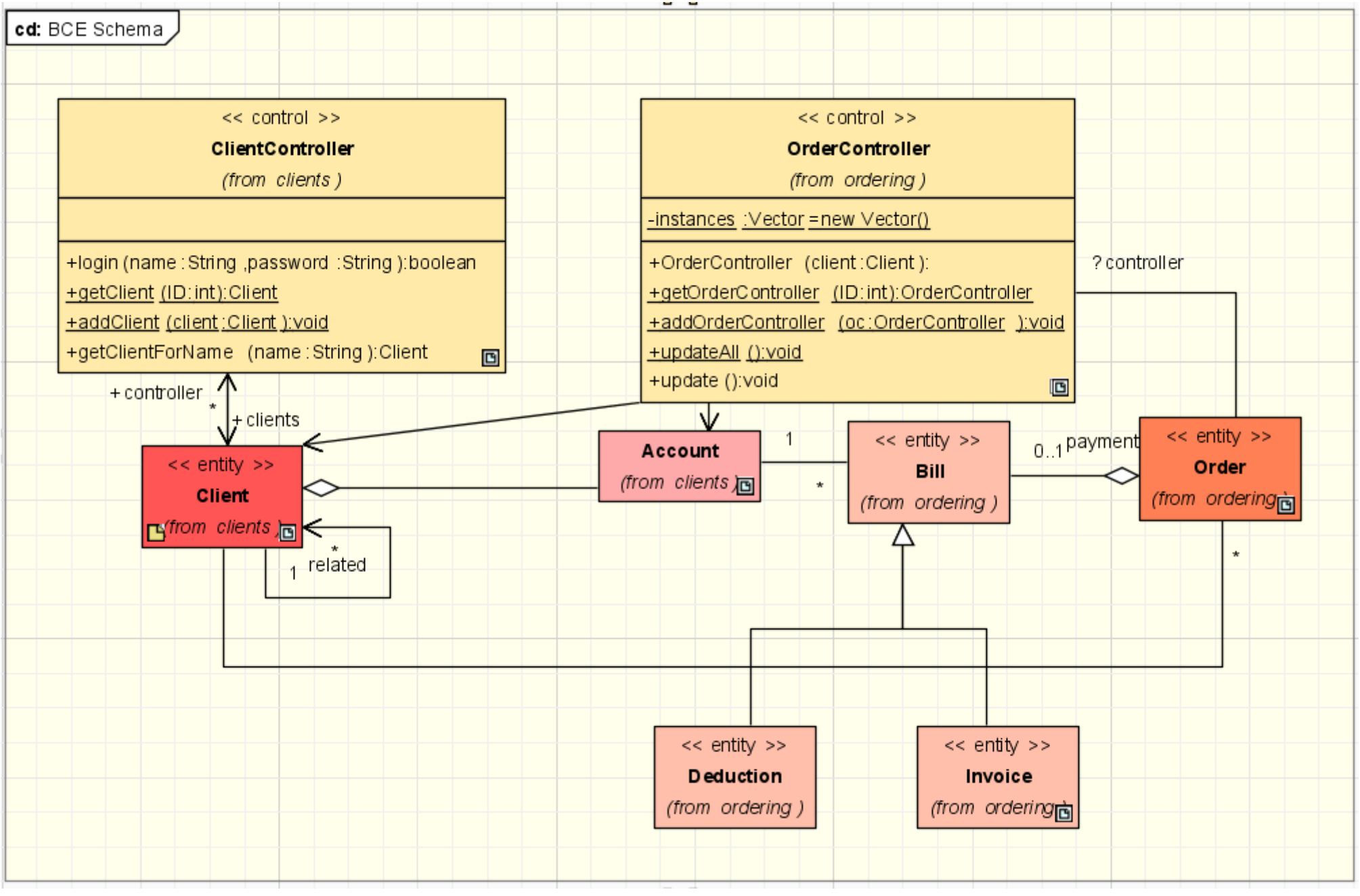
Sample diagrams



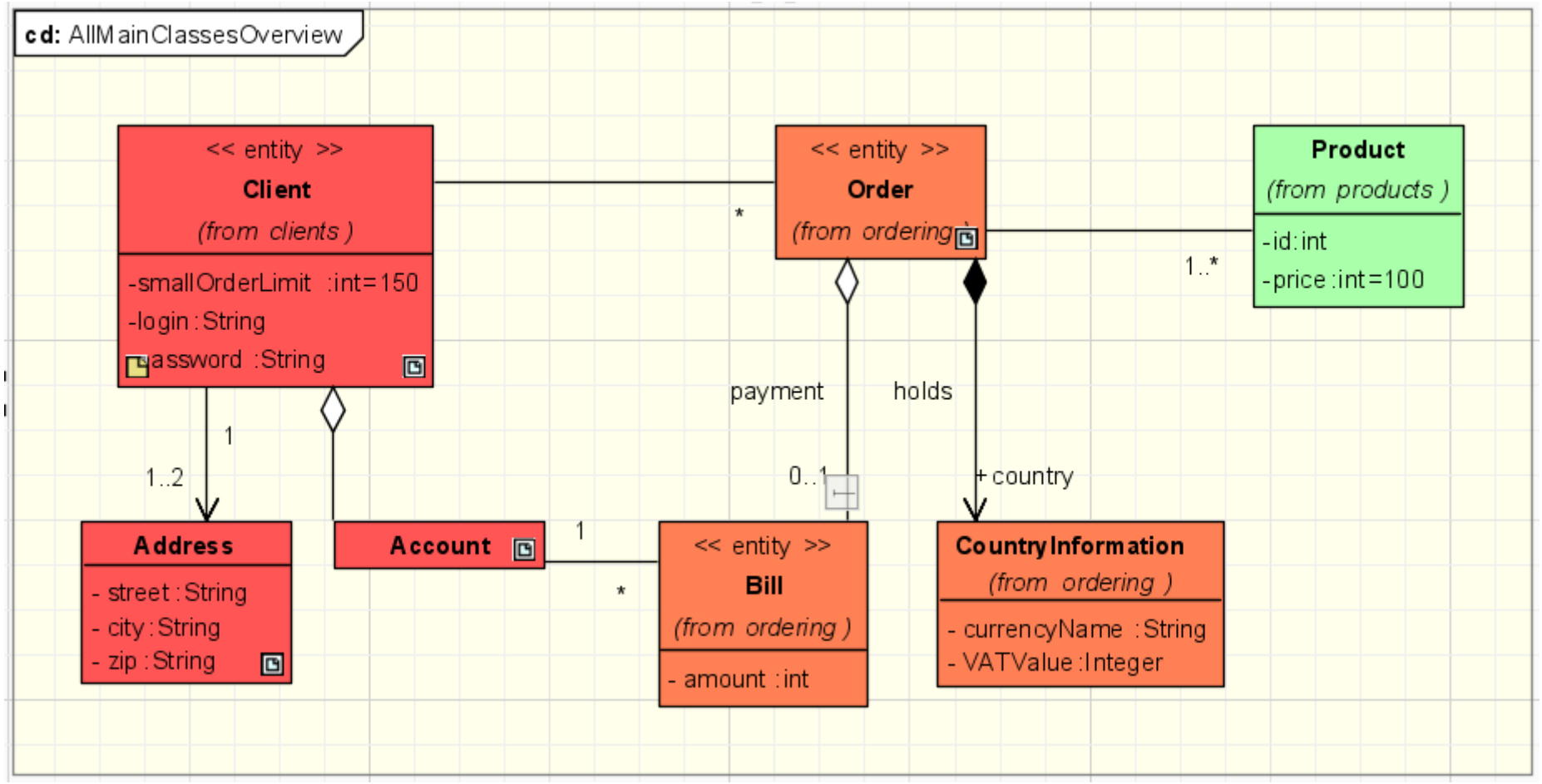
Sample diagrams



Sample diagrams



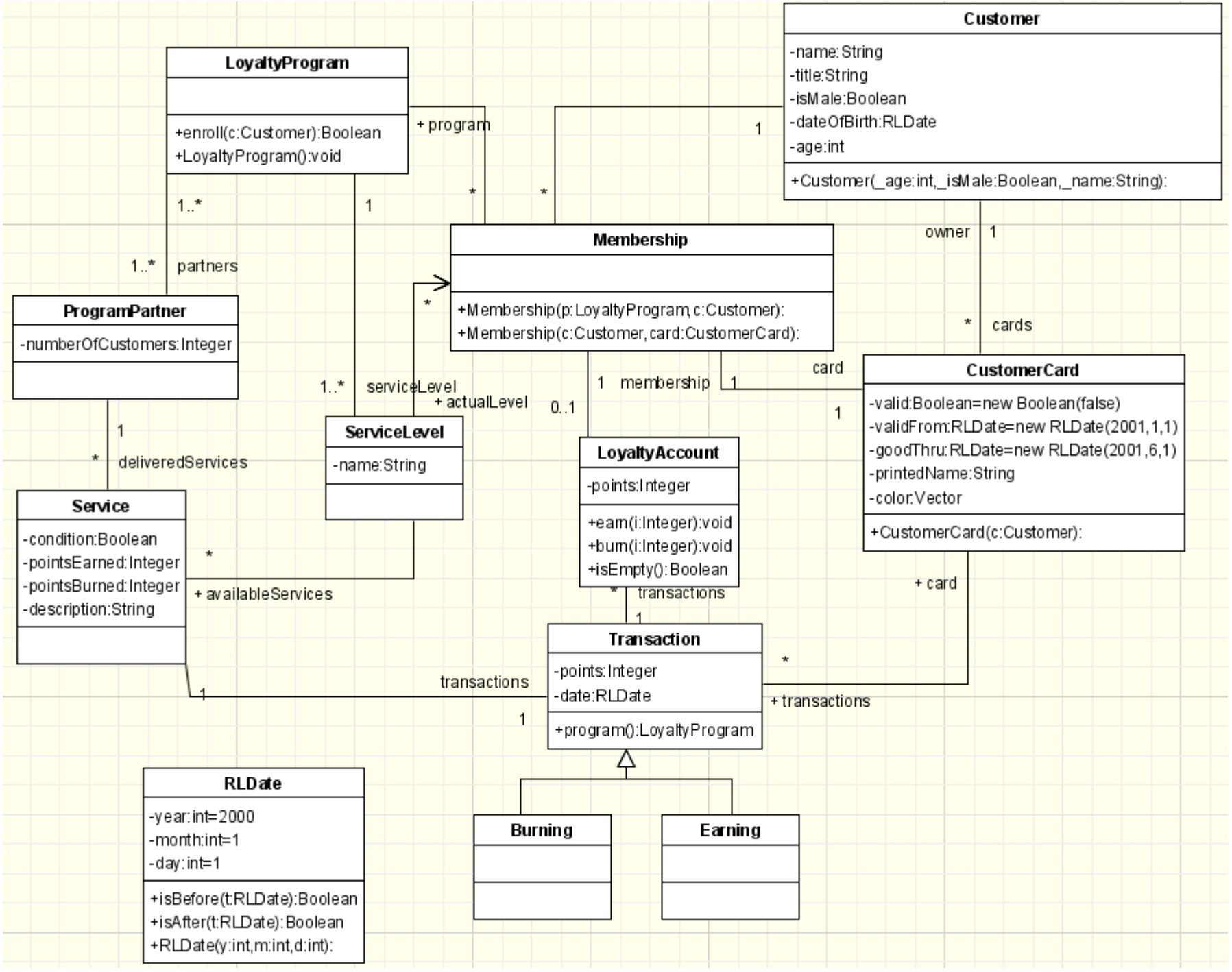
Sample diagrams



Code generation

```
public class Client {
    private products.int smallOrderLimit = 150;
    private String login;
    private String password;
    public clients.CreditCard creditCard;
    public clients.EMail eMail;
    public java.util.Collection address = new java.util.TreeSet();
    public java.util.Collection client = new java.util.TreeSet();
    public ordering.void paymentReceived() {
        return null;
    }
    public ordering.void eMailReceived() {
        return null;
    }
    public ordering.void deliveryReturned() {
        return null;
    }
    public boolean checkPassword(String password) {
        return false;
    }
       /*...*/
    public java.util.Collection order = new java.util.ArrayList();
    clients.Account account;
    public clients.ClientController controller;
 }
```

Sample diagrams



Associations revisited

- Association – a relationship exists between two classes (student – teacher, seller – buyer)
- Weak aggregation – one class belongs to another, but the part can exist without a whole (order – products, library – books)
- Strong aggregation (composition) – one class belongs to another, the part cannot exist without a whole (polygon – its vertices, order – shipping address)

Activity diagrams

- Describe dynamics of the system (cf. class diagrams)
- They graphically represent sequential and concurrent control and data flows
- They can be used for modeling:
  - business processes
  - **use cases scenarios**
  - algorithms
  - operations

Main elements of activity diagrams

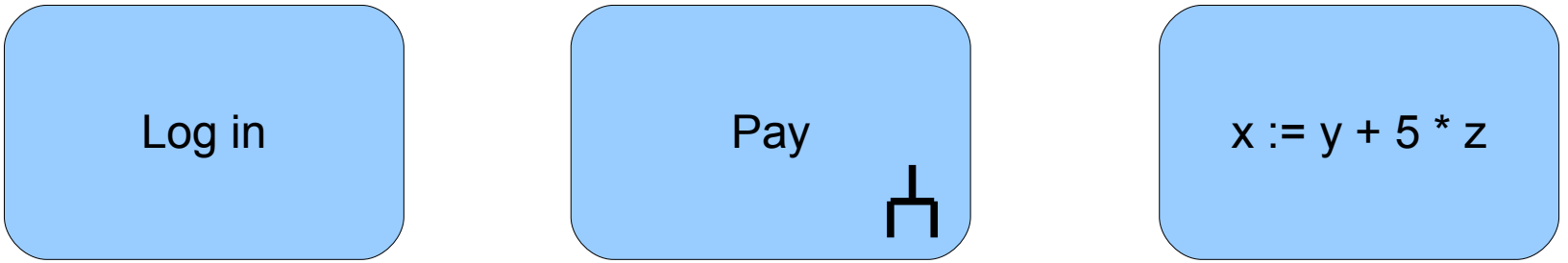
- activity
- action
- control flow
- initial node
- activity final
- flow final

Activity

- Activity may represent complex processes and algorithms
- In order to improve readability, not all elements of the process/algorithm are represented
- Instead, activity can be decomposed into other activities (using a separate diagram), creating a hierarchical structure
- The decomposition can be performed till we reach the level of *actions* - elementary entities describing dynamics of the system

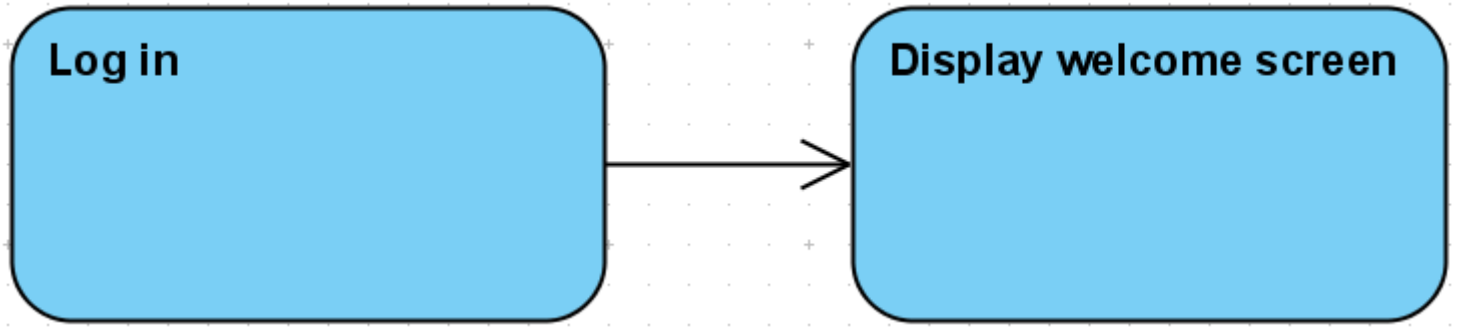
Activity and action

- Activities are represented by rectangles with rounded corners
- Decomposable activities may have special mark denoting this fact placed in right lower corner
- Actions are depicted in the same manner as activities (but cannot have the “decomposable” symbol)



Control flow

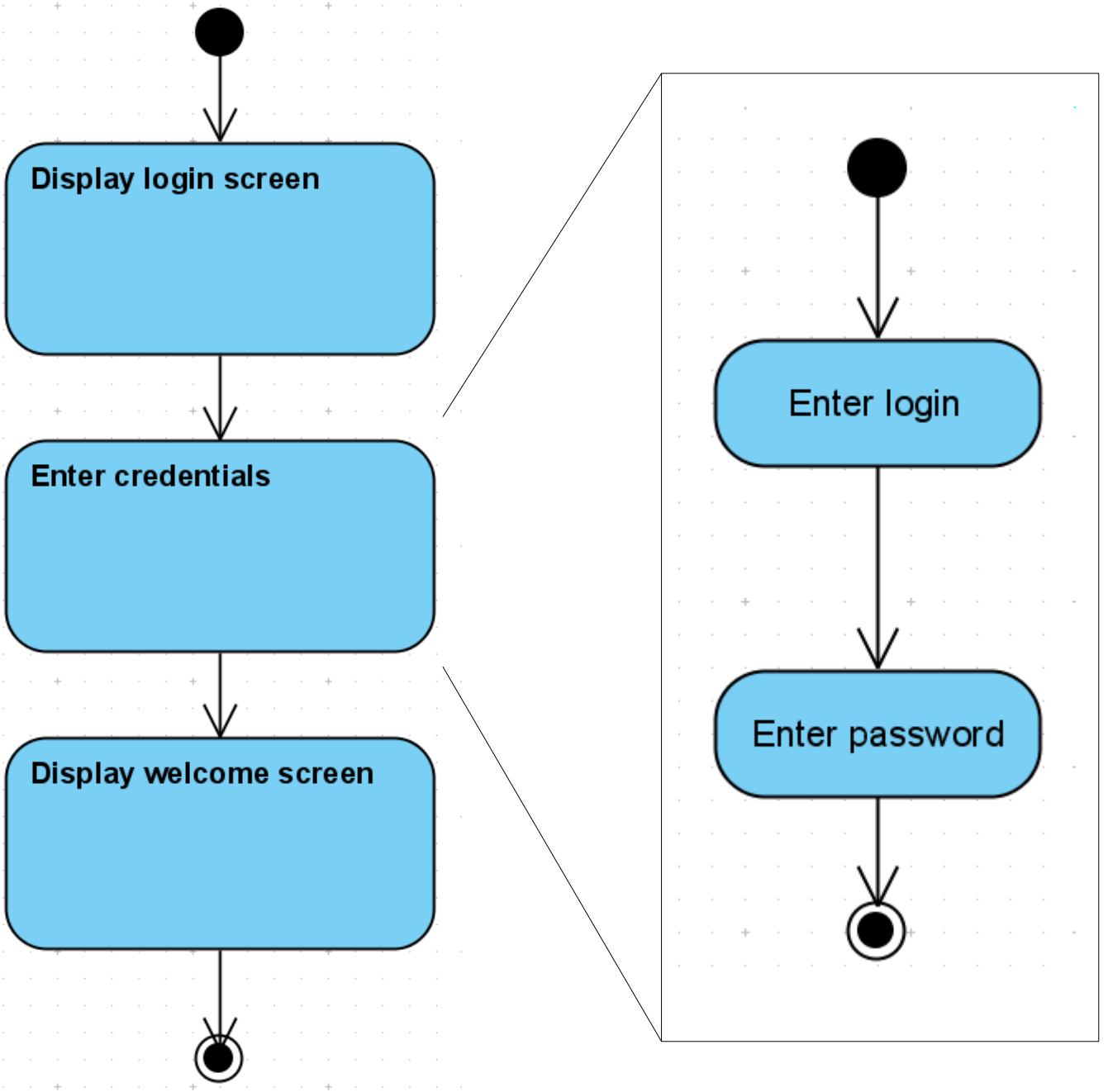
- Control flow is a relation between two activities/actions denoting that after completion of one activity/action the control will be passed to the other
- It is denoted by an arrow



Initial node, activity final, flow final

- Initial node indicates beginning of control flow(s). Usually one per diagram. Denoted by a solid circle
- Activity final indicates stop of **all** flows in the diagrams. May be more than one. Denoted by a small solid circle inside a bigger hollow one
- Flow final indicates stop of **one** flow. May be more than one. Denoted by a hollow circle with a cross (X)

Simple diagram



Decision and merge nodes

- More complicated diagrams require means of representing decisions and alternate control paths
- This can be achieved by using decision and merge nodes
- Decision node has one input flow and two or more output flows. Only one output can be selected at a time
- Merge node has may inputs and only one output
- Symbol of both nodes is a diamond, they are distinguished on the basis of the number of inputs and outputs

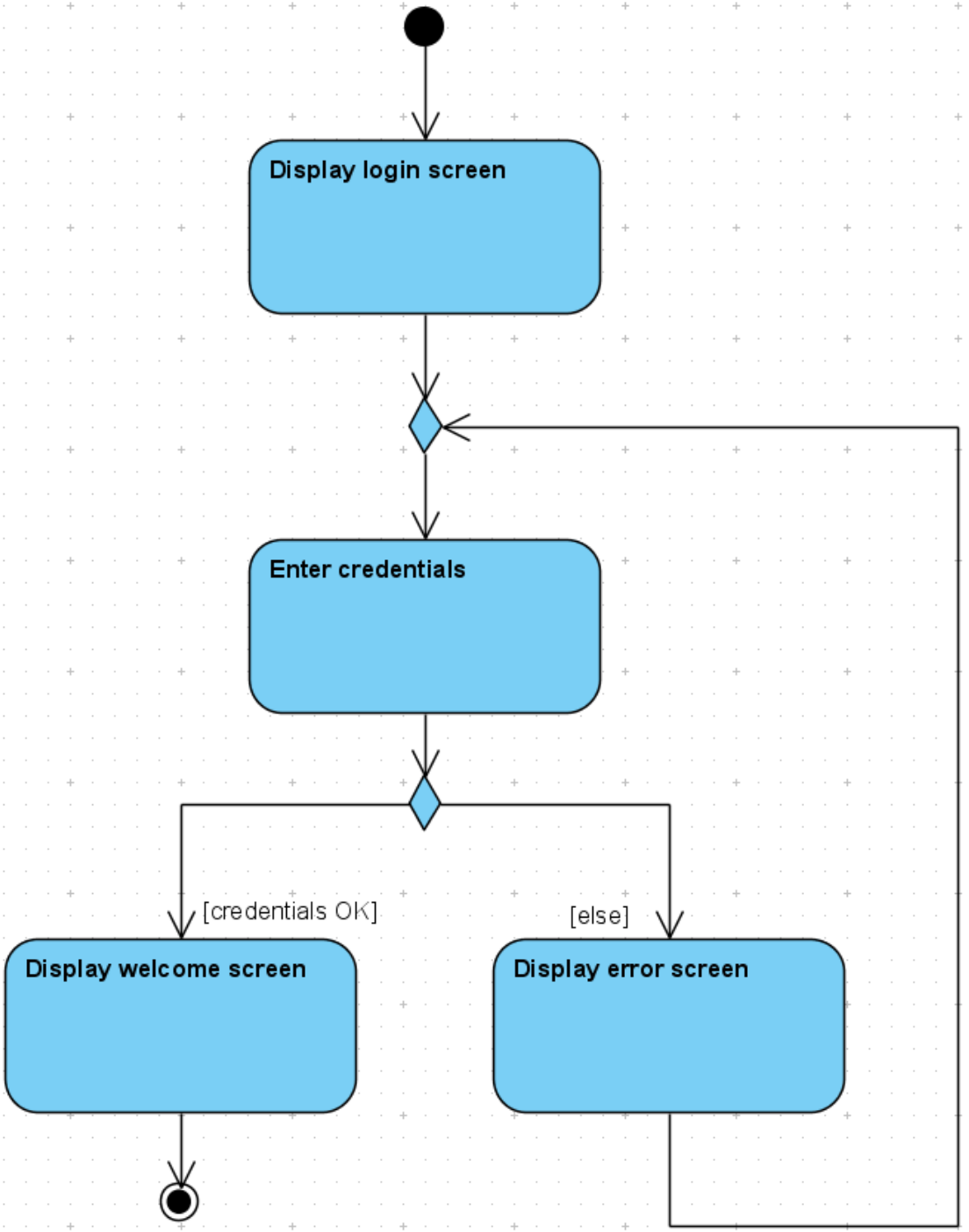
Decision node

- Output selection is performed on the basis of a guard condition
- Guard condition is placed in rectangular parentheses close to the output
- All guard conditions must be mutually exclusive
- One of the gurad conditions can be replaced by the keyword *else* (also placed in the square parentheses)

Merge node

- Does not perform any synchronisation functions - every flow that reaches the merge node will be immediately forwarded to the output

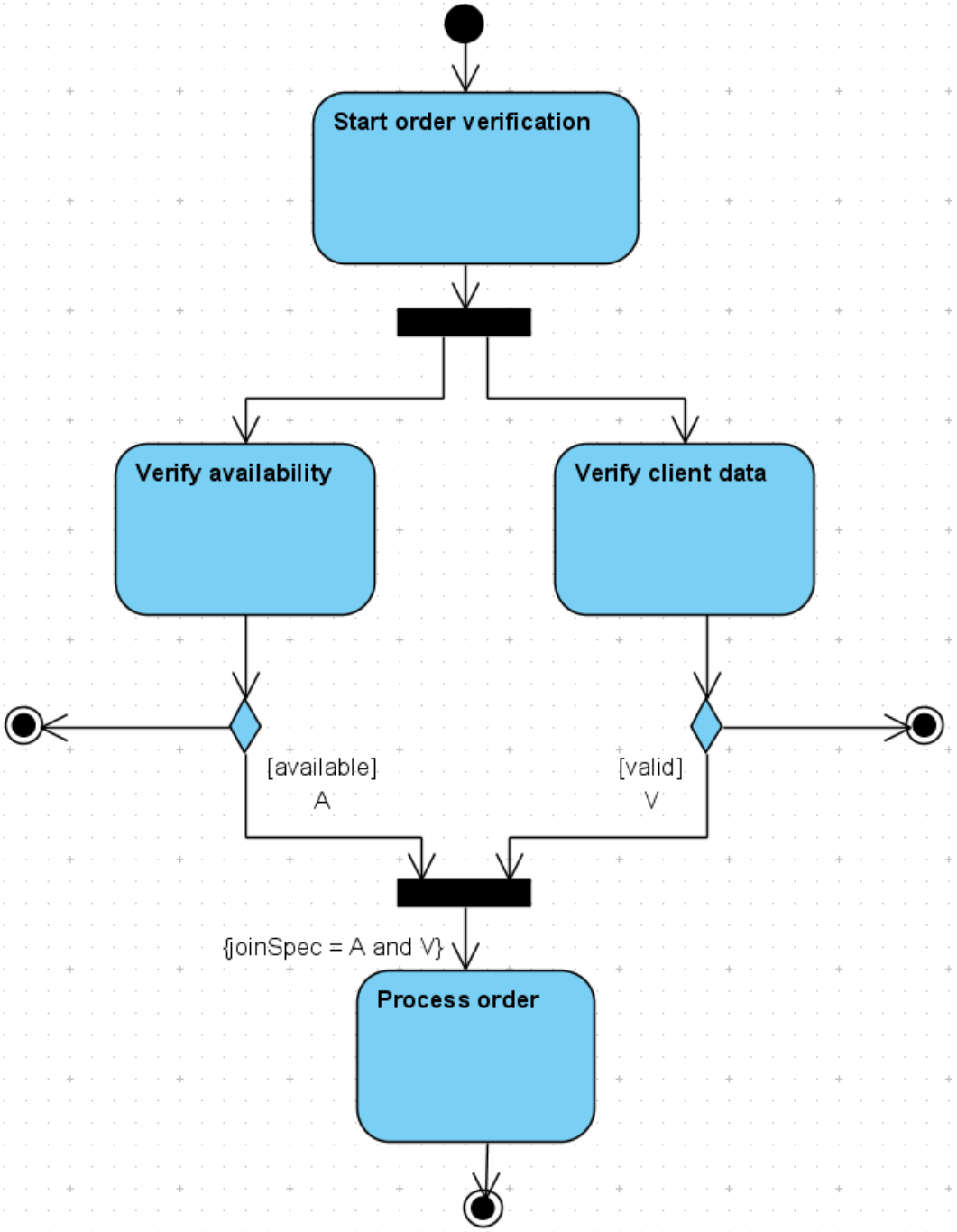
Sample diagram



Concurrent flows

- It is possible to specify flows that execute in parallel (concurrently)
- In order to model this functionality, fork node and join node are defined
- Fork node has one input and two or more outputs; flows entering fork node are split
- Join node has two or more inputs and one output. It can be used to synchronise flows. It is possible to use *join specification* - boolean condition that specifies that the flow is passed to the output (true) or destroyed (false)

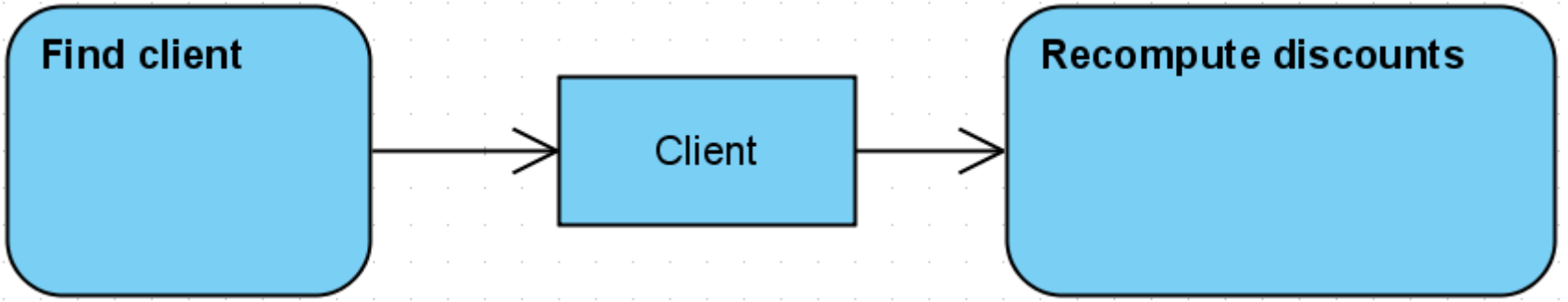
Sample diagram

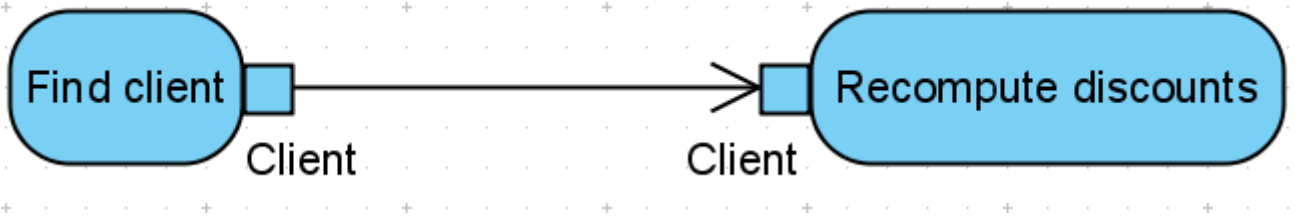


Data flow

- As a supplement to the control flow, is is possible to describe data flow (flow of objects)
- It is useful when:
  - Actual object flow takes place
  - State of an object is changed
- The object has to be connected with an activity or action
- The object is depicted by a rectangle with the object name
- Alternatively, object flow can be depicted using input/output pins

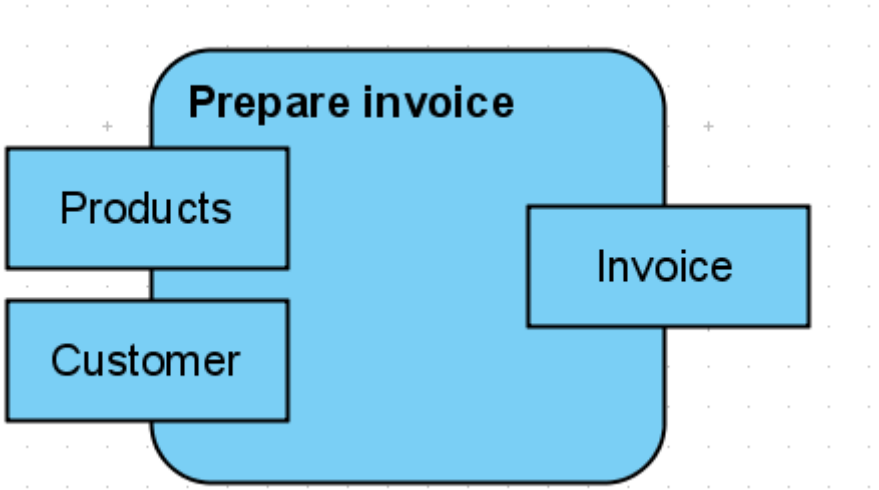
Simple data flow





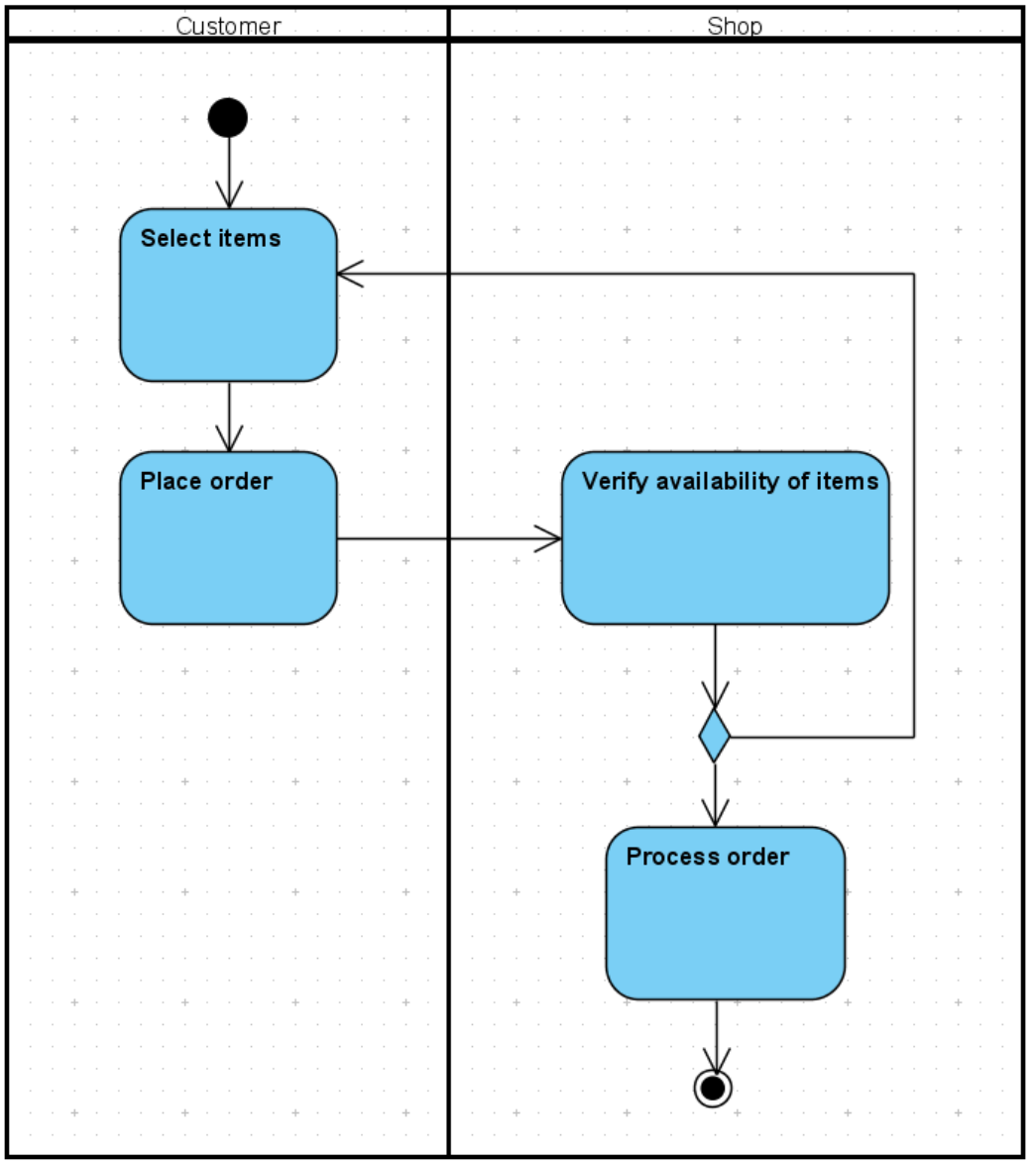
Activity parameter

- It is possible to specify that an object is a parameter of an activity
- This is depicted by the object rectangle placed on the border of an activity



Partitions

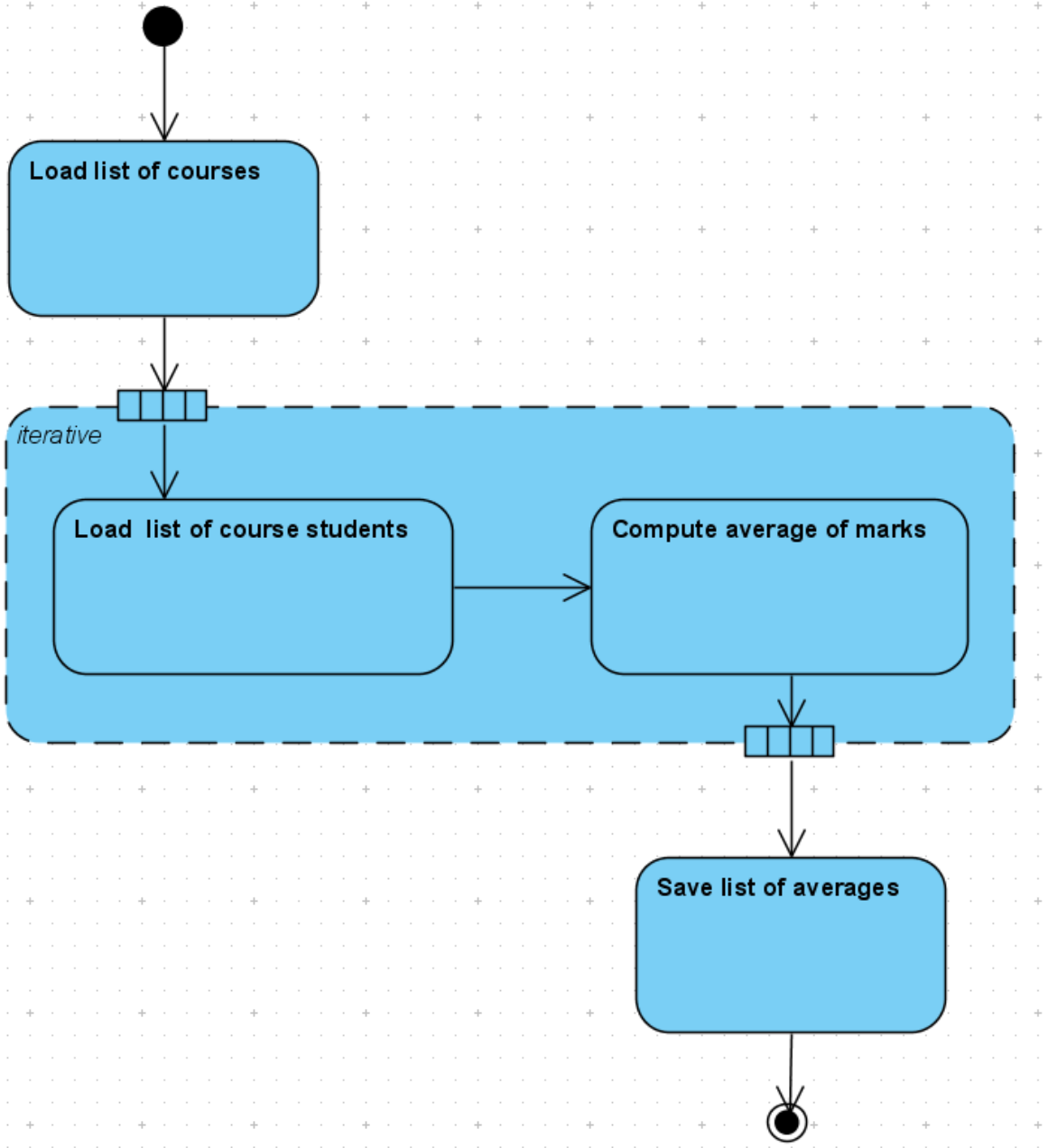
- Activities, actions and objects can be grouped into partitions



Expansion region

- Expansion region allows to specify a part of a diagram that is executed many times, depending on the number of elements on its input
- The inputs and outputs of the expansion region are called expansion nodes
- The mode of execution is specified by a string in italics, placed inside the expansion region, and can be:
  - iterative
  - parallel
  - streaming

Sample expansion region



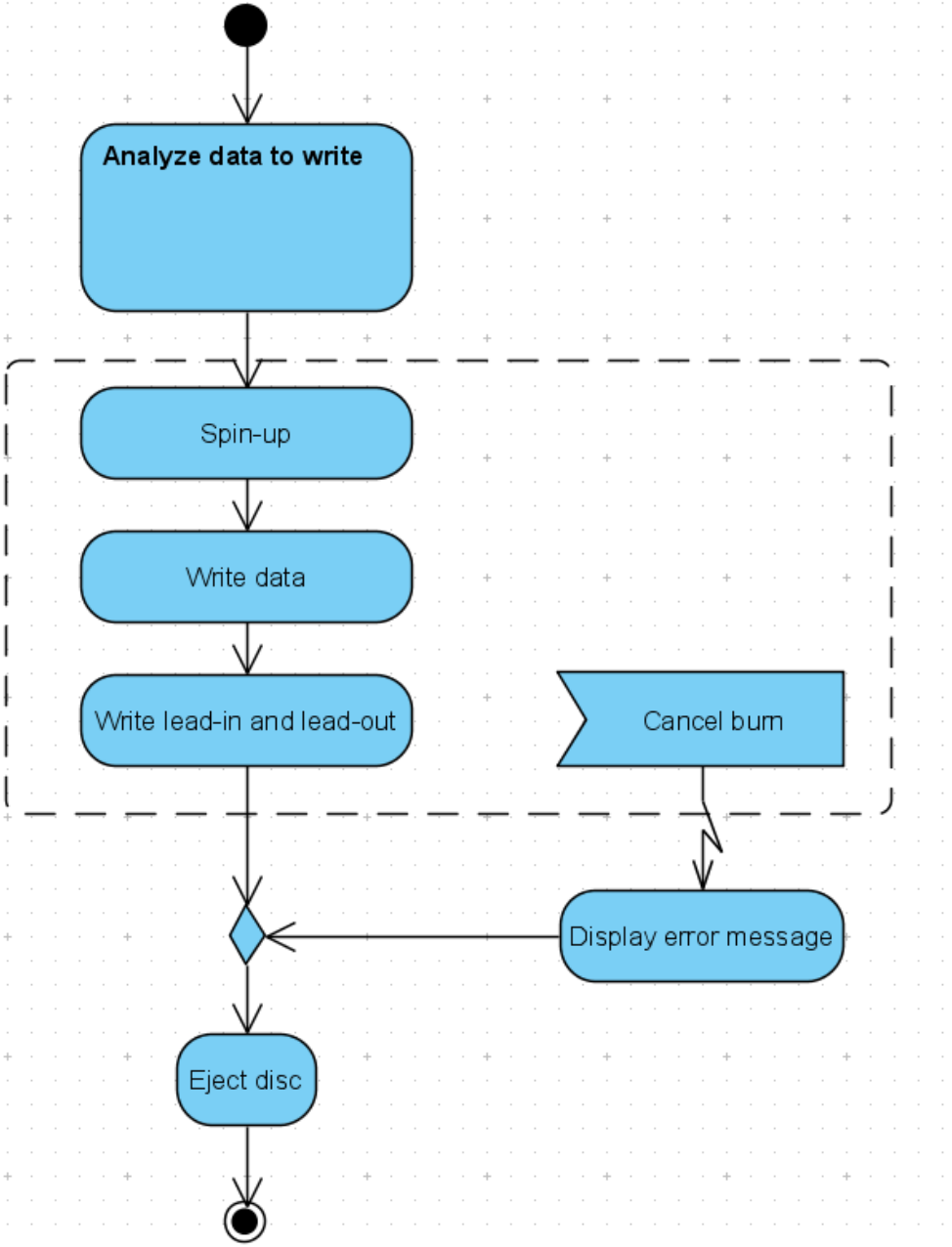
Interruptible activity region

- Allows to specify a part of the diagram where execution can be immediately interrupted by an external condition
- In case of an interruption all flows are terminated except the interrupting edge
- The interrupting edge always starts inside of the interruptible activity region and ends outside
- For this functionality signals notation can be useful

Signals

- Signals can be used to represent asynchronous processing
- It is possible to:
  - send signal
  - accept signal

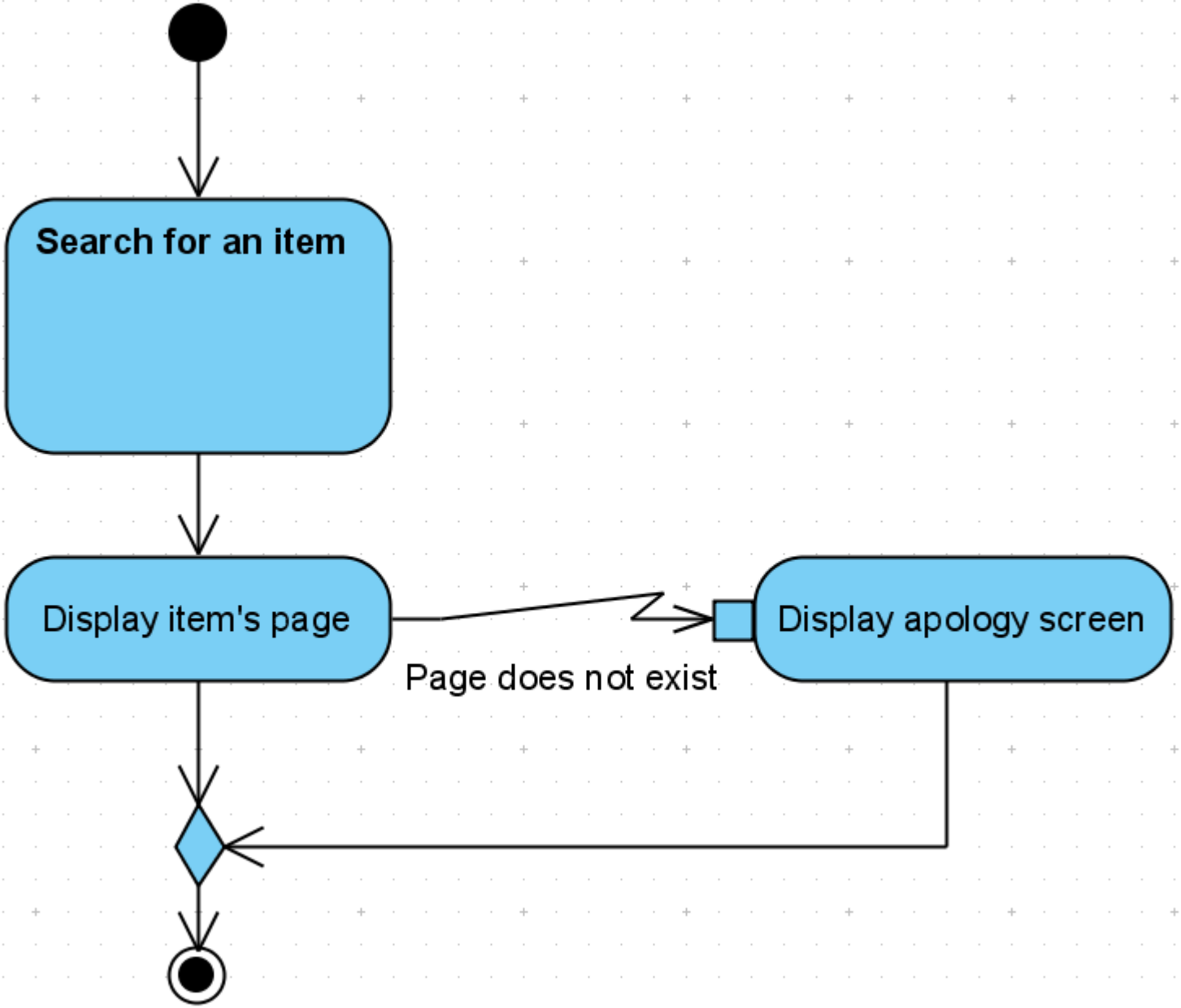
Sample interruptible activity region



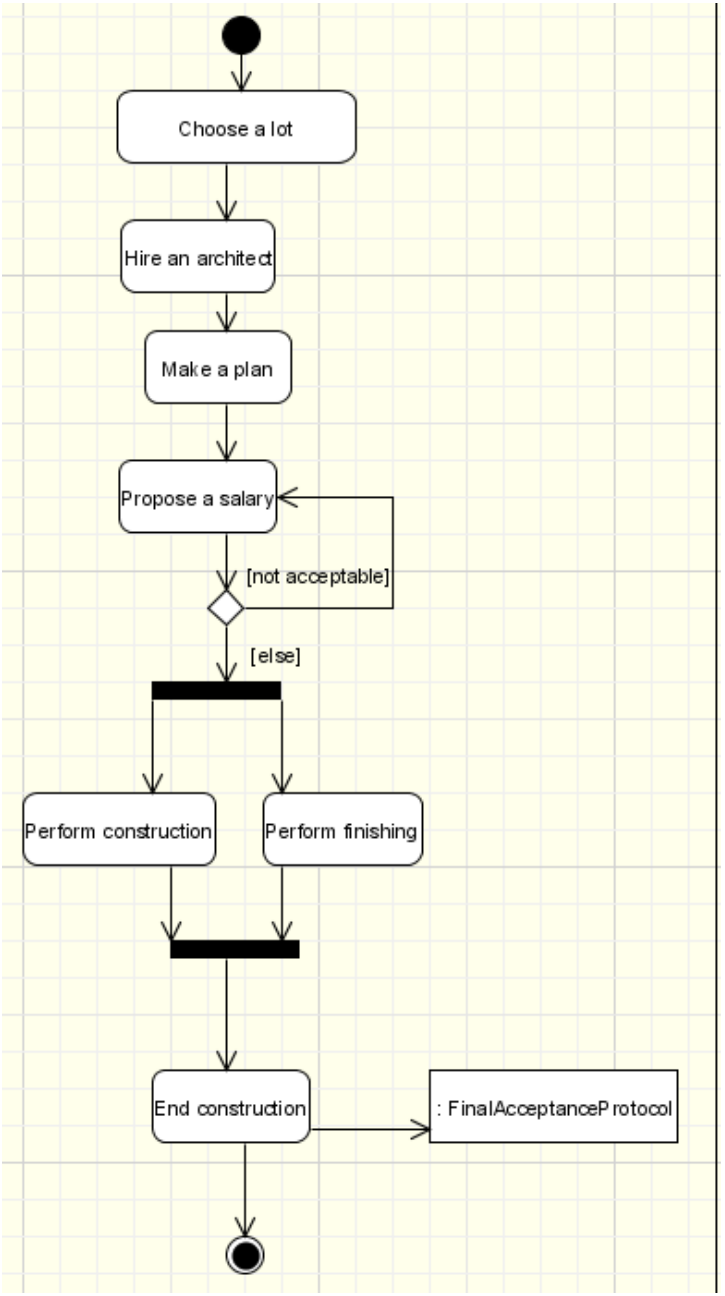
Exception handlers

- Allow to model an activity performed in error situation
- In case of an exception condition, control is passed to the handling activity
- The handler must be named

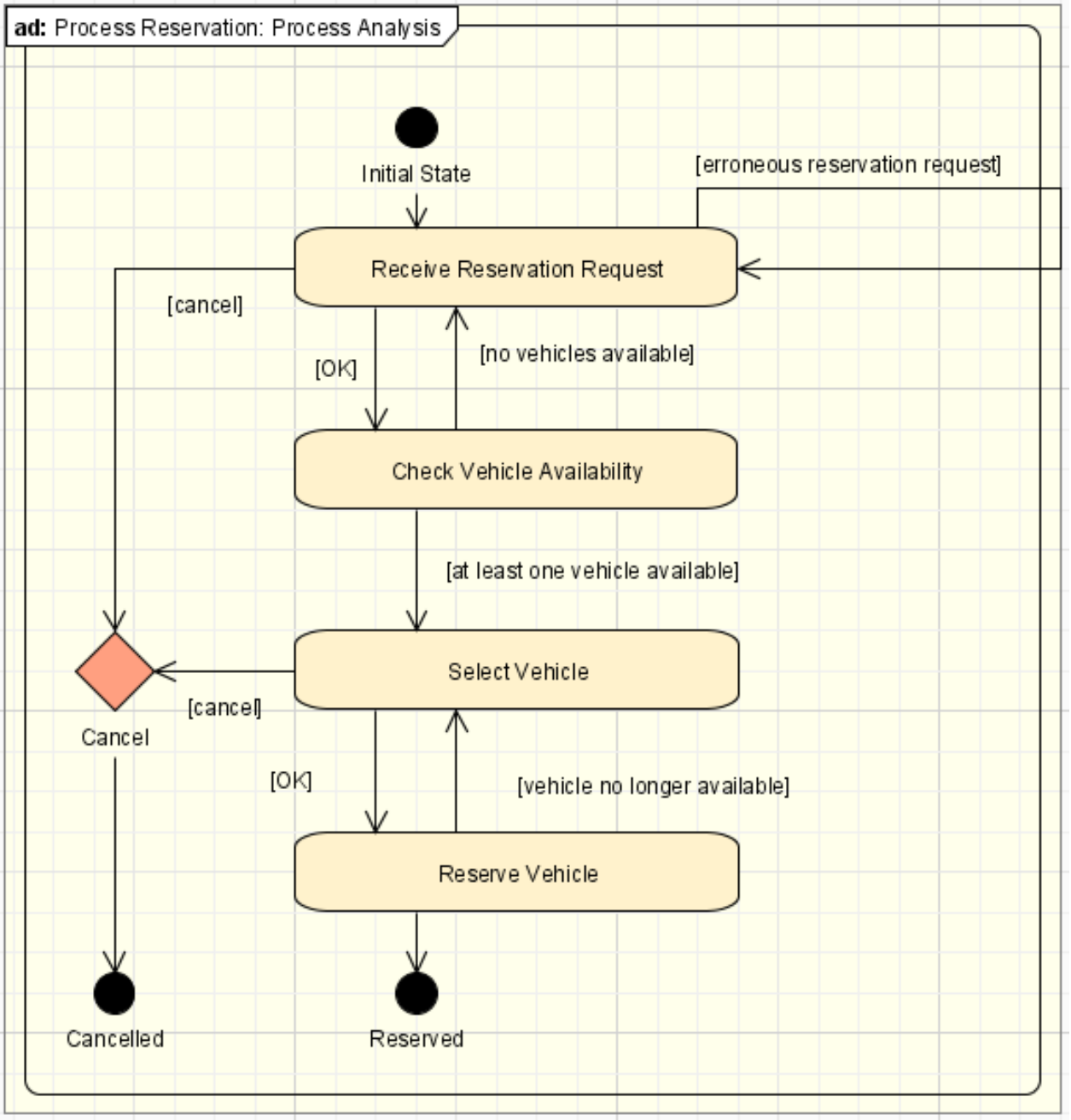
Sample exception handler



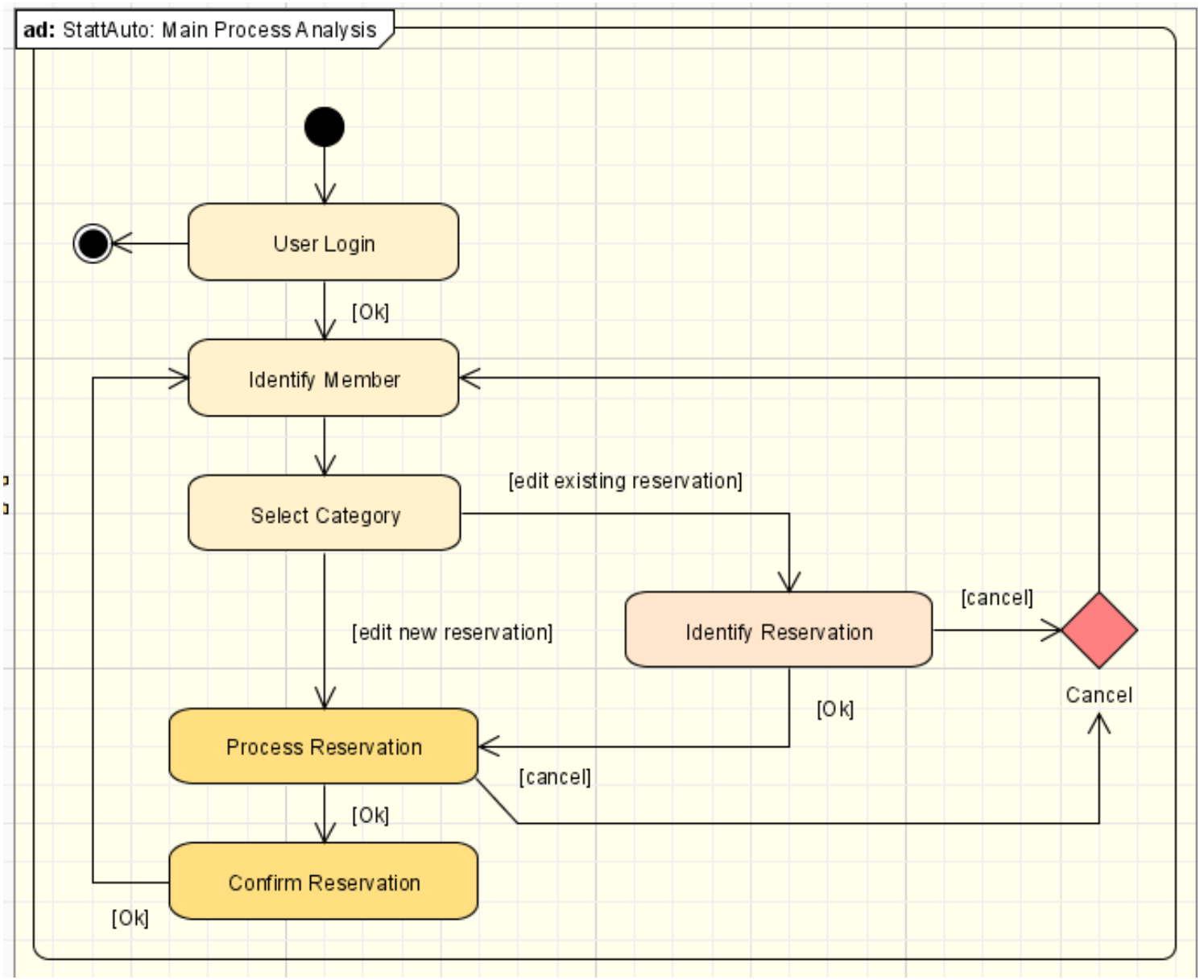
Sample diagrams



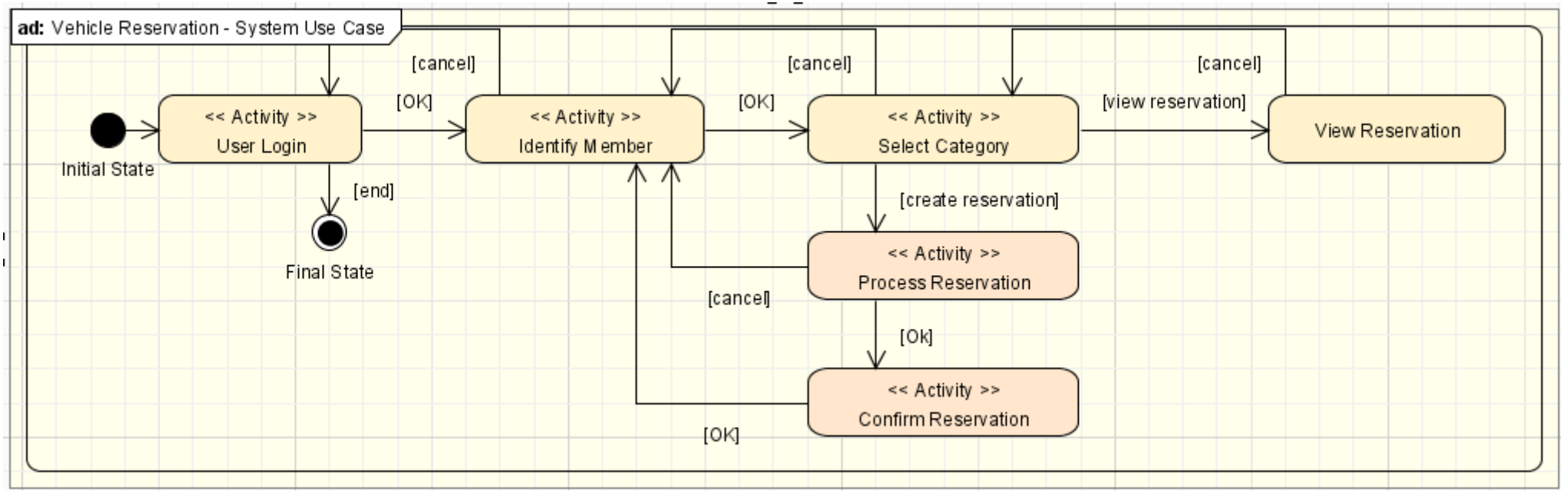
Sample diagrams



Sample diagrams



Sample diagrams



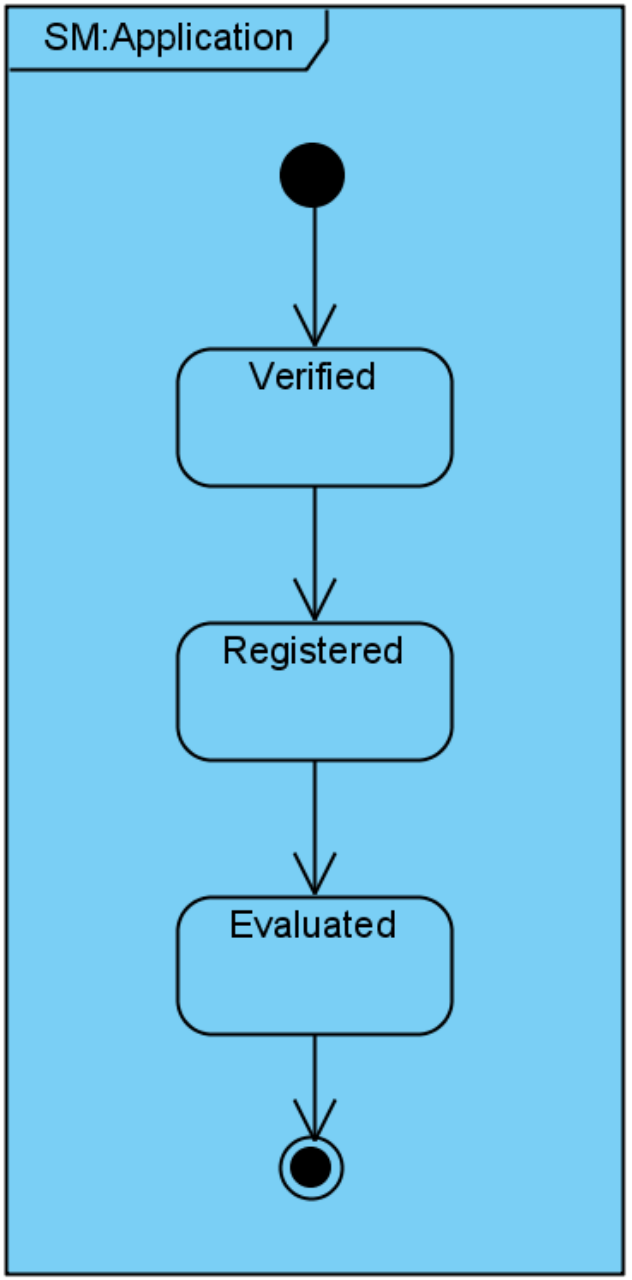
State machine diagrams

- State machine diagram describes graphically discrete behaviour (state-transition) of finite systems
- Describe states of ojects
- Can be directly used to generate programming language code
- Are constructed using elements introduced in the activity diagrams
- Main elements are:
  - State
  - Transition
  - Initial and final states

State machine diagrams

- State is a condition of an object. It may be performing some actions, waiting for an event, fulfilling a condition
- Transition indicates that an object being in the first state will perform some actions and transit to the second state whenever particular conditions are met

Sample state machine



Elements of the state

- The state in a machine can be divided into sections:
  - Name
  - Internal activities
  - Internal transitions
  - Decomposition
- The sections are separated using horizontal lines

State name

- Should in an unambiguous way define state of an object. Unlike in activity diagrams, where names were imperatives ordering to perform some action, here the names describe condition the object is in

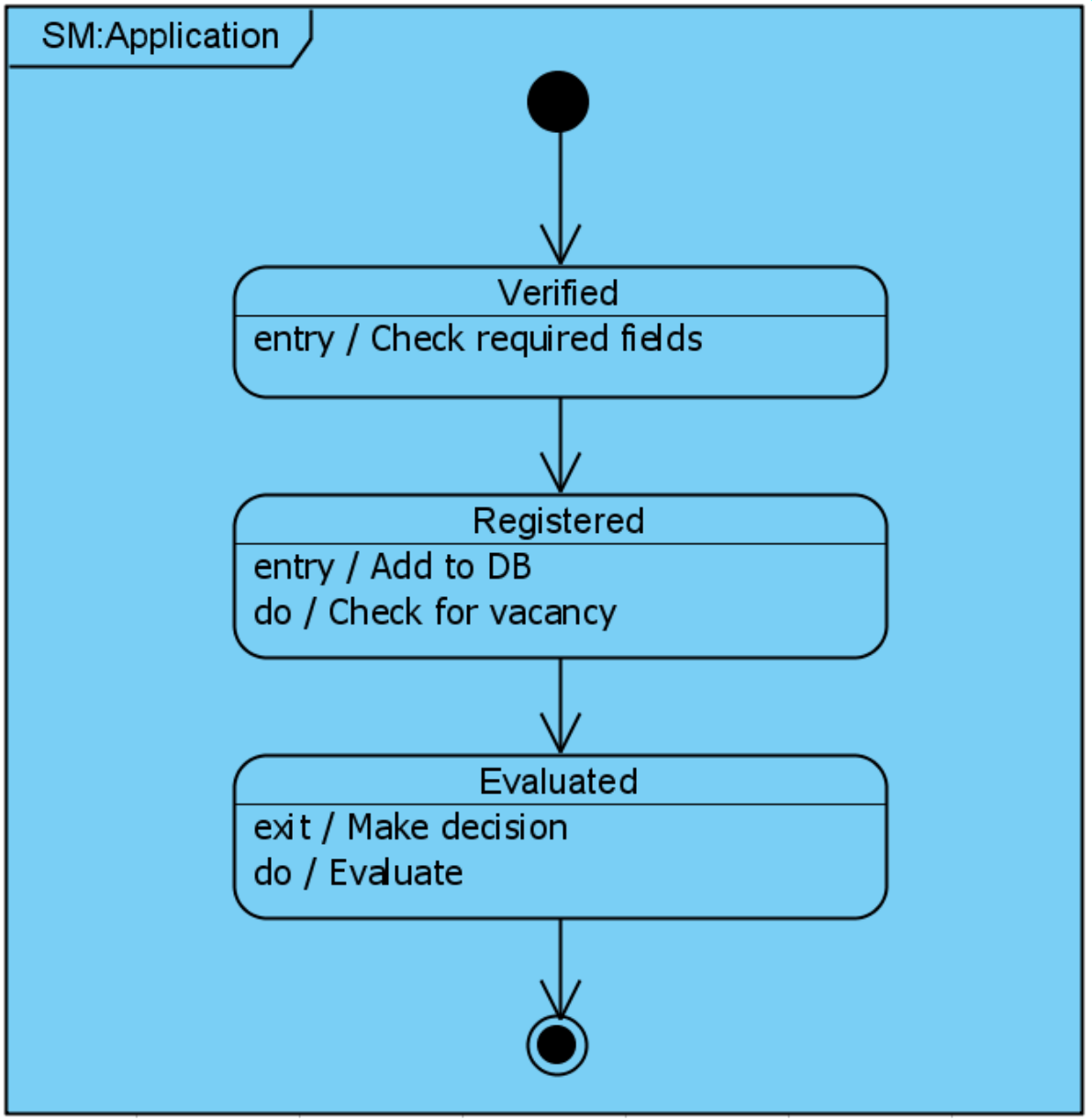
Internal activities

- Describe activities performed in connection with a state. Three kinds are possible:
  - Entry – activity performed when an object enters particular state
  - Do – activity performed while an object is in a particular state
  - Exit – activity performed when an object leaves a particular state
- It is possible to define only one activity of “entry” and “exit” kind, and an unlimited number of “do” kind

Internal transitions

- Indicate transitions that start and end in the same state
- Unlike external transition that starts and ends in the same state, this transition does not trigger the “enter” and “exit” activities

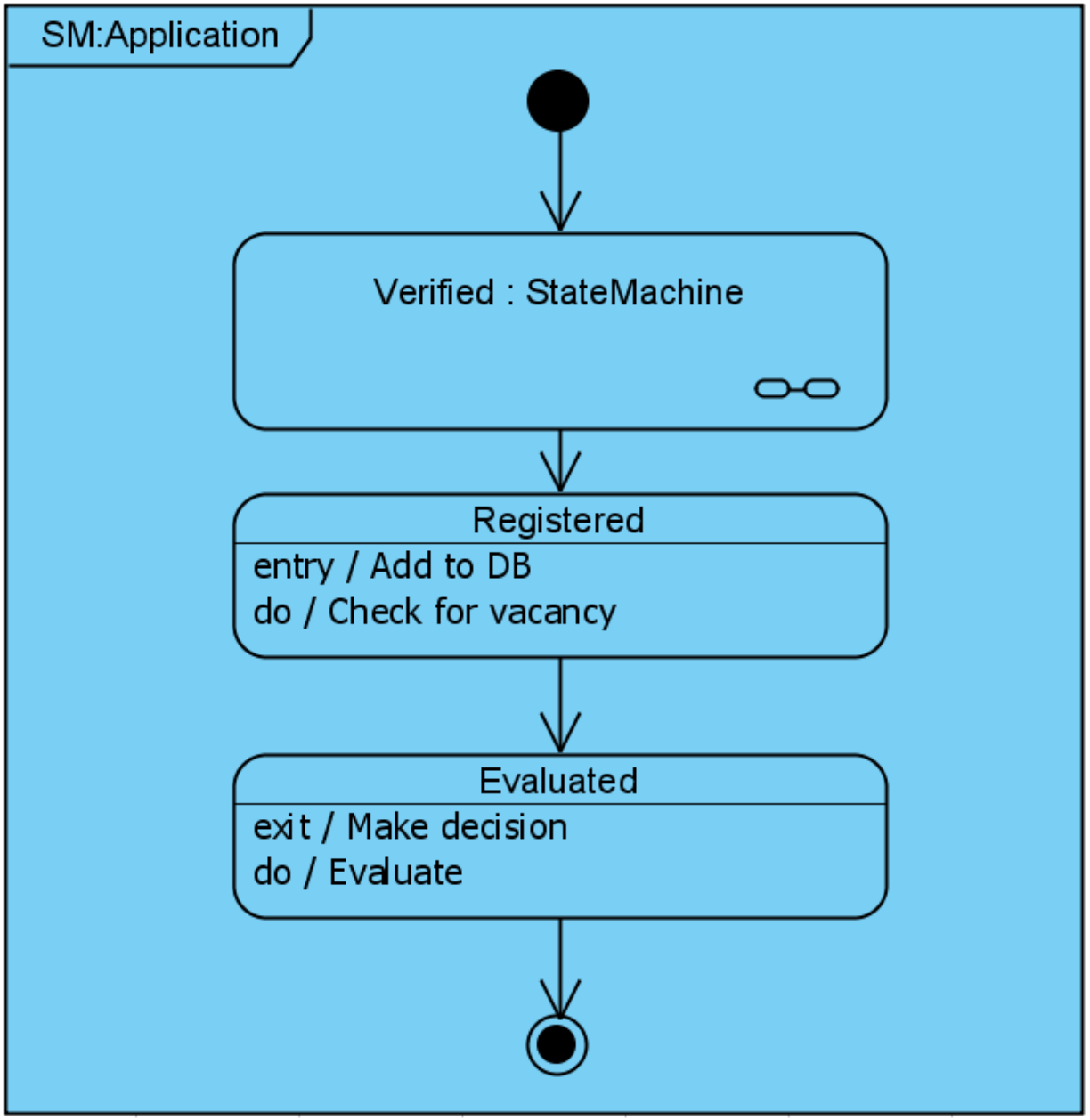
Sample state machine

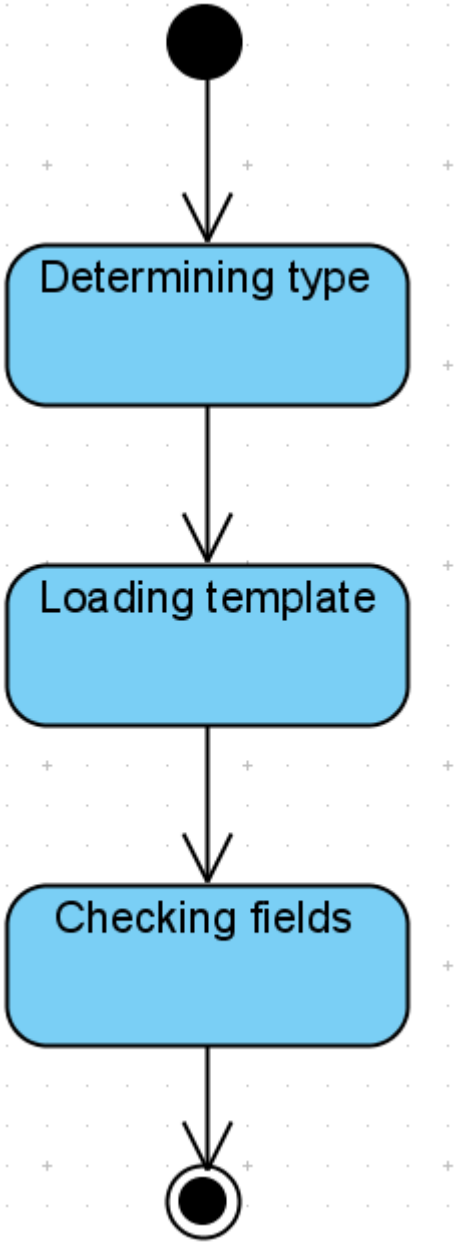


Composite state

- Is used to provide more details about a (complex) state
- It can contain a submachine
- Submachine is a normal state machine diagram
- The state containing a submachine is depicted using a submachine symbol in the lower right corner

Composite state - submachine

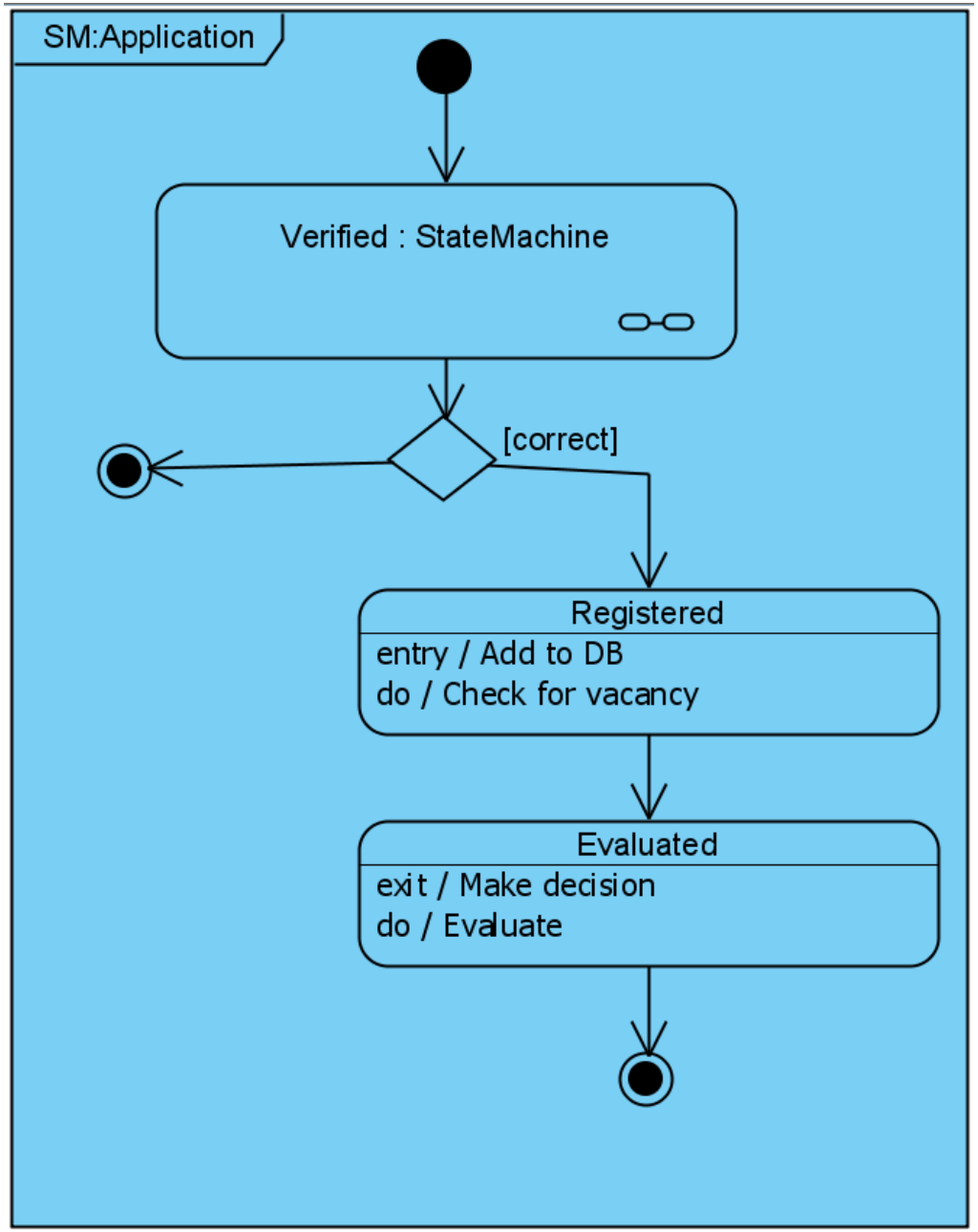




Choice, junction, fork and join nodes

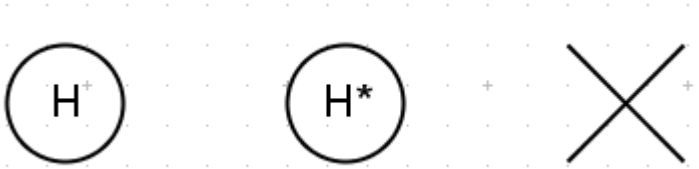
- It is possible to use nodes similar to the activity diagrams:
  - Choice (similar to decision node)
  - Junction (similar to decision and merge nodes)
  - Fork
  - Join

Choice node

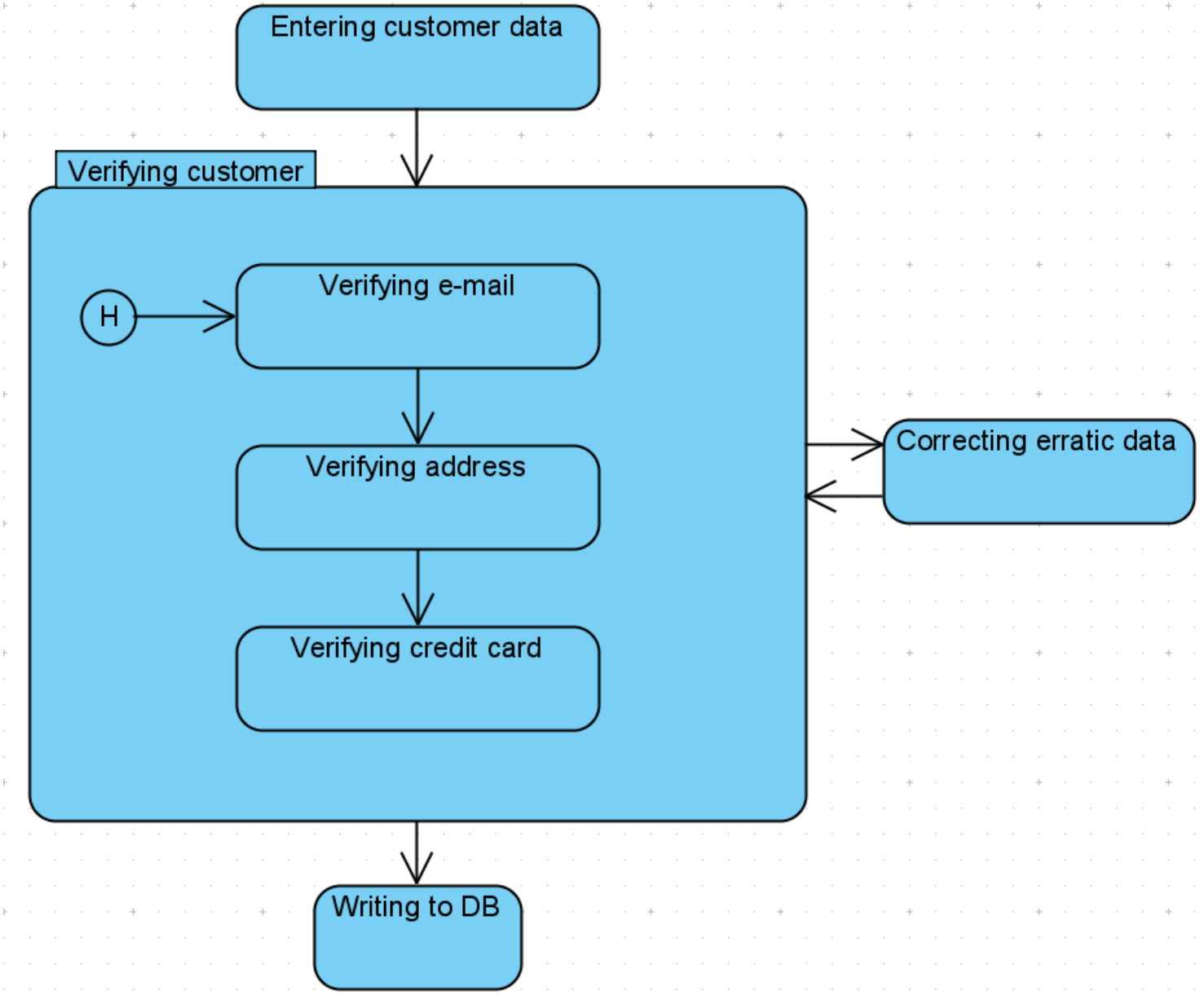

SM:Application
Verified : StateMachine
[correct]
Registered
entry / Add to DB
do / Check for vacancy
Evaluated
exit / Make decision
do / Evaluate

Shallow history, deep history, termination

- History enables to save information about complex state upon leaving this state
  - Shallow history saves a pointer to the substate that was active
  - Deep history saves also information about all substates (i.e. pointers to the active substates in substates, and so on)
- Termination indicates end of state machine processing due to destruction of the object being processed



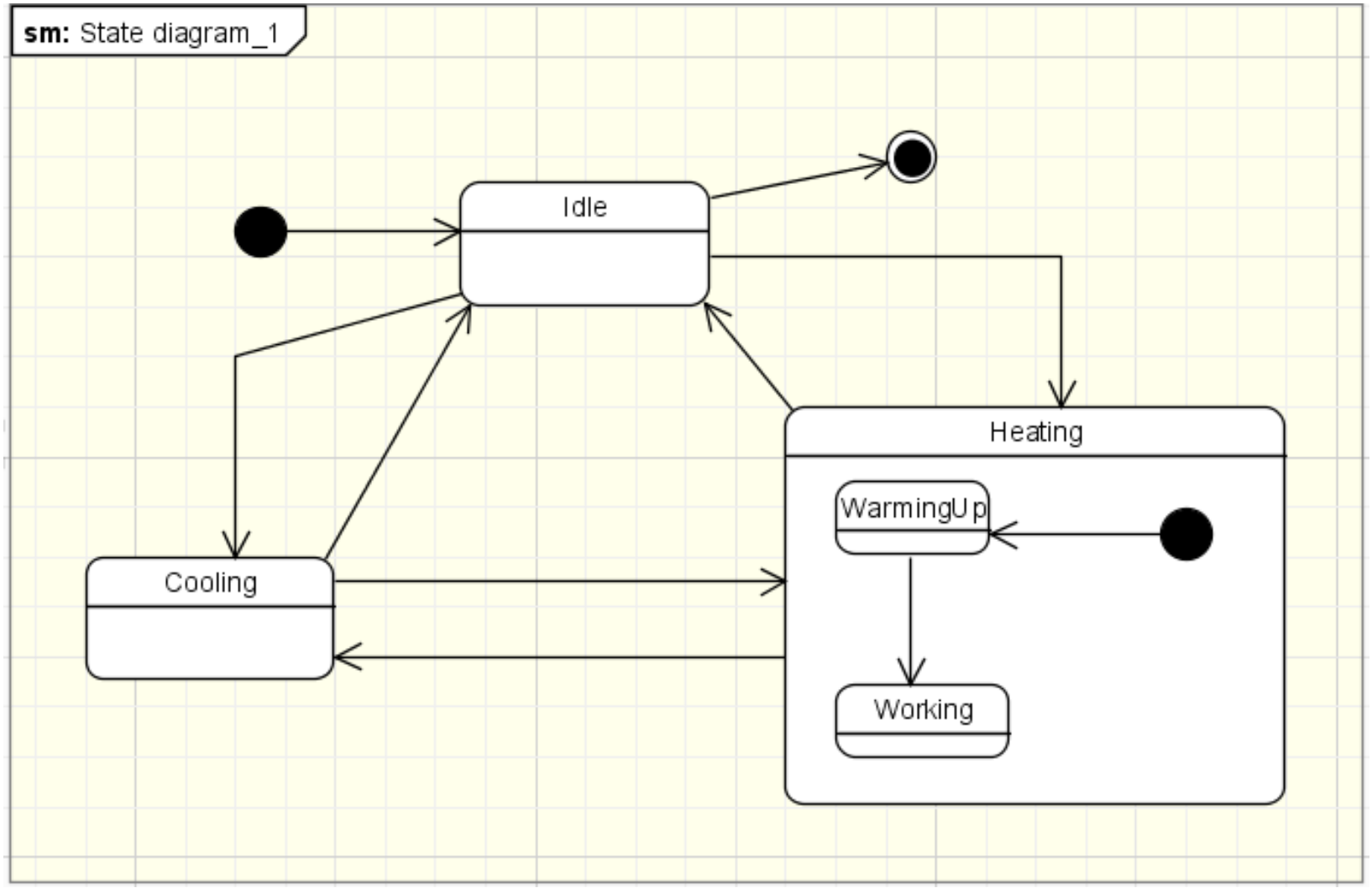
History sample



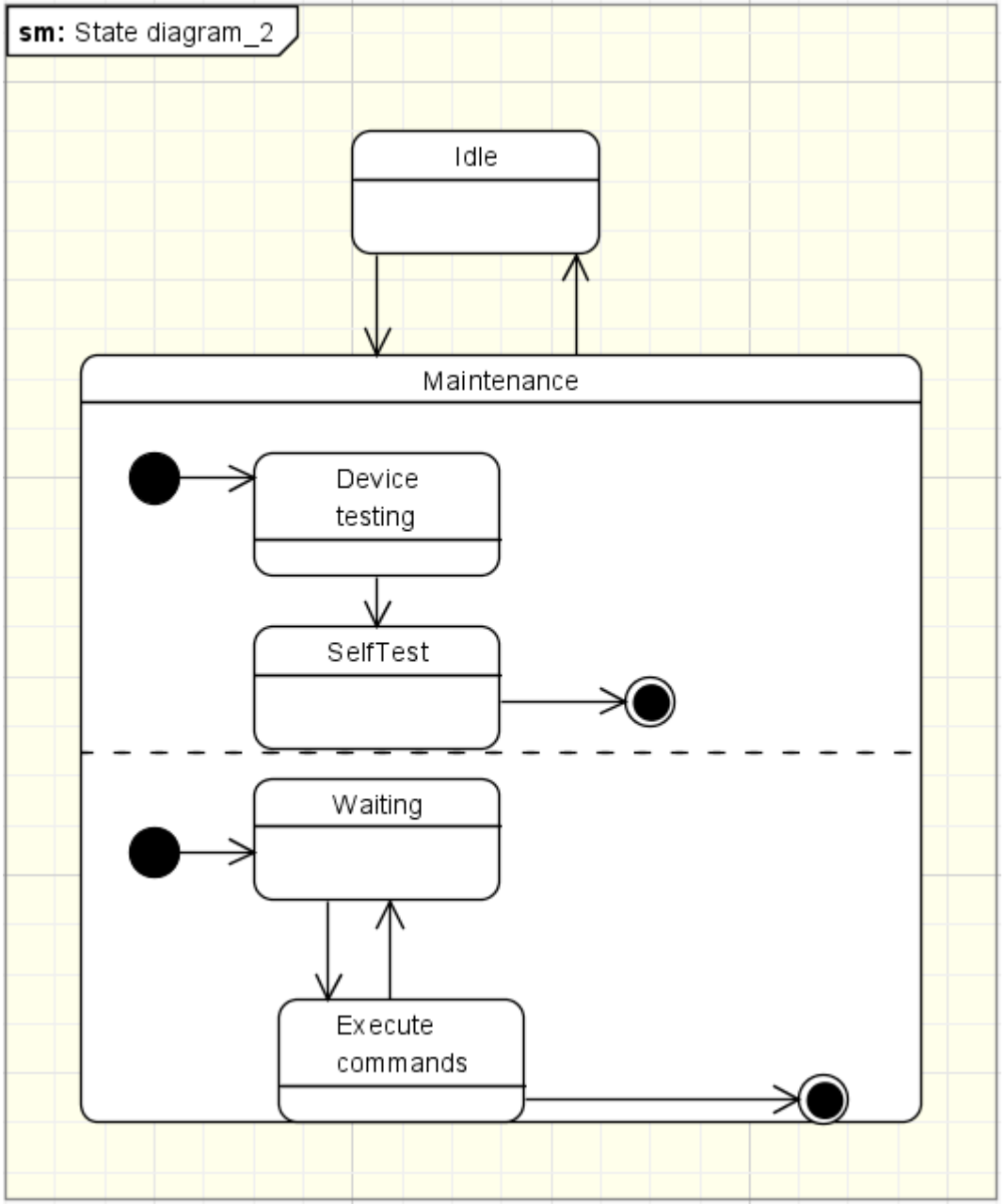
Events

- Events trigger transition from one state to another. Possible events are:
  - Signal. Asynchronous.
  - Call event. Similar to function call.
  - Time event. Happen after certain amount of time. Defined by *after* keyword
  - Change event. Happen when condition is met. Defined by *when* keyword

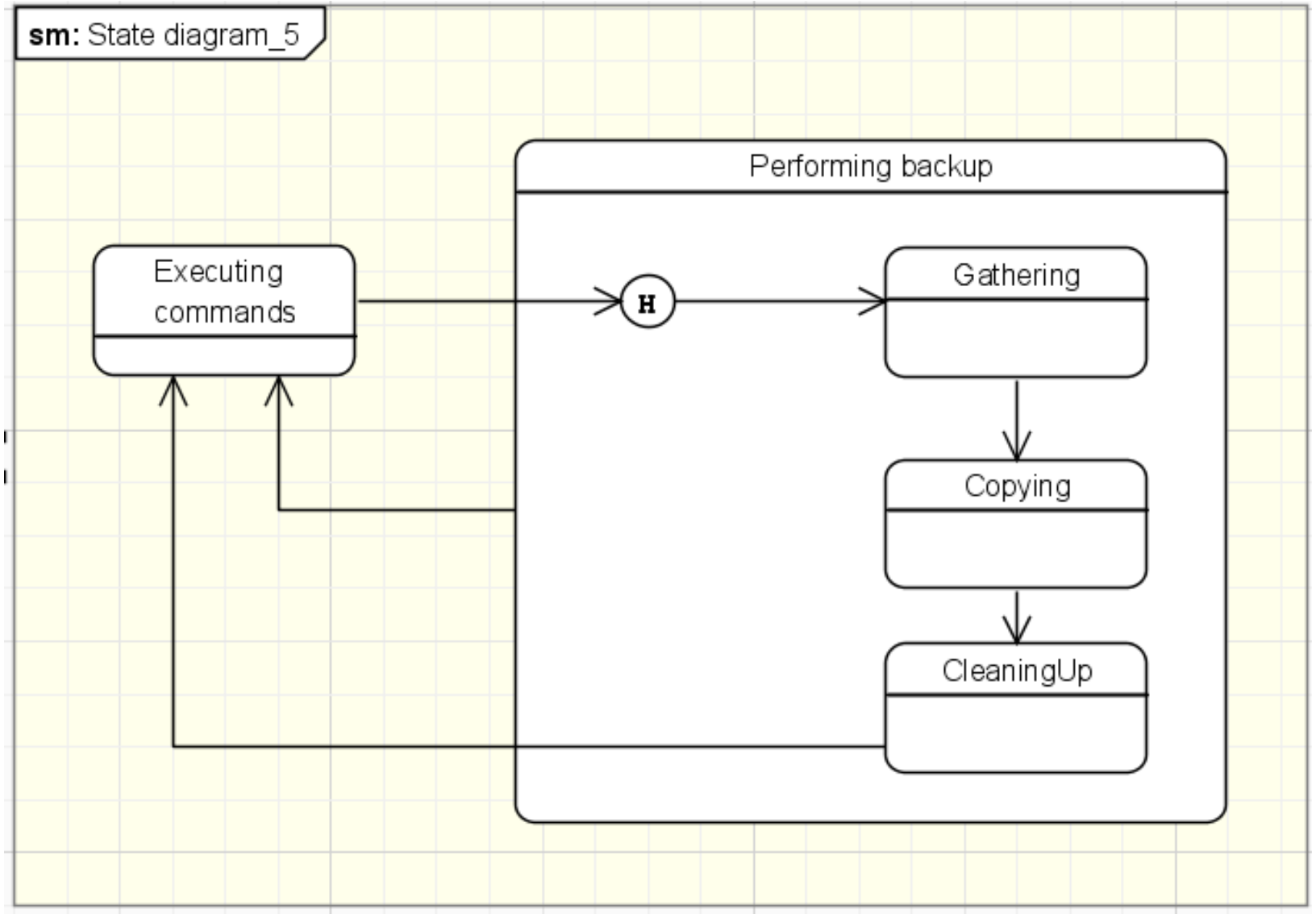
Sample diagrams



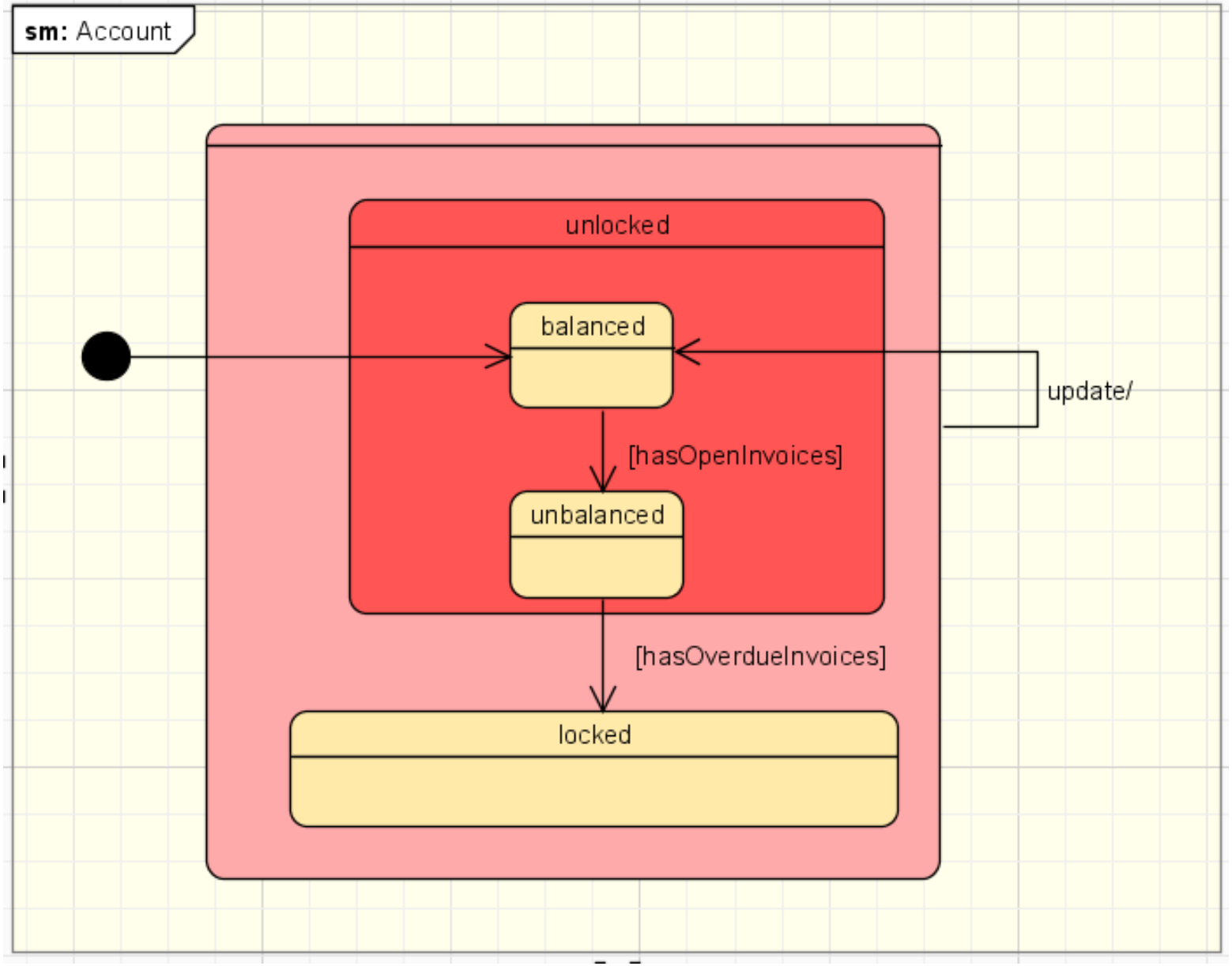
Sample diagrams



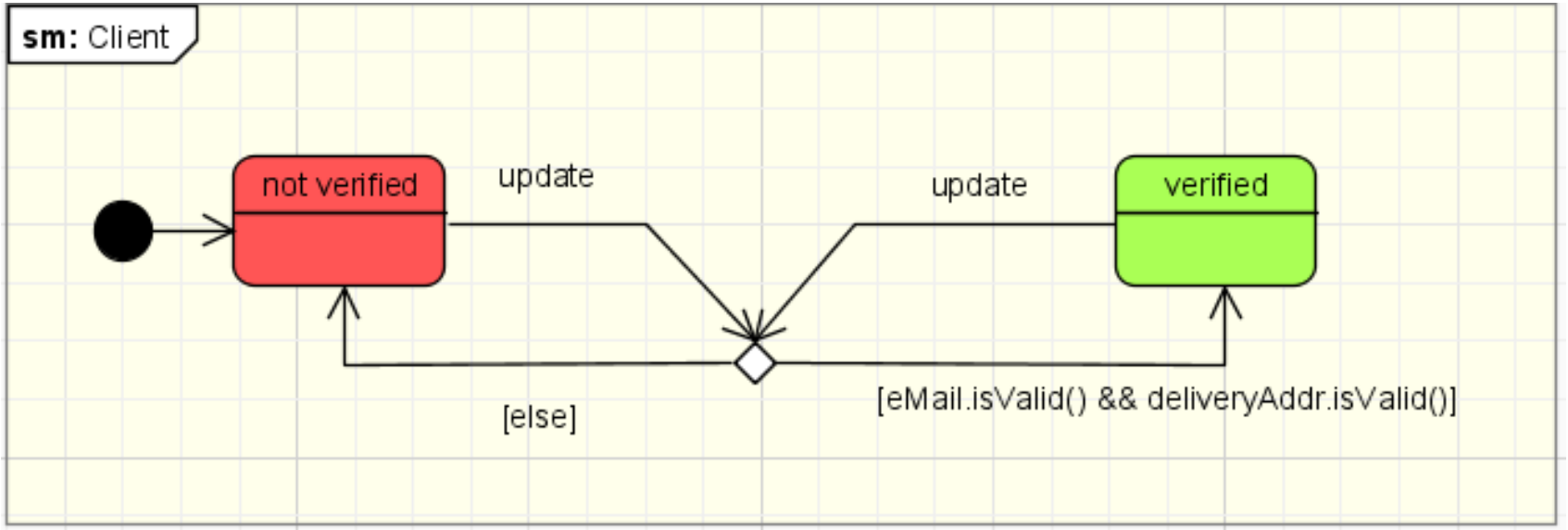
Sample diagrams



Sample diagrams



Sample diagrams



Sequence diagrams

- Describes interactions between objects as a sequence of messages
- Good for documenting use cases
- Two dimensions are used in the sequence diagrams:
  - Horizontal is used to indicate the objects taking part in the communication
  - Vertical is used to indicate the time sequence of interation

Sequence diagrams

- Three types are possible:
  - Conceptual. Only basic elements are used. Suitable for first sketches, and managers
  - Generic. Much more detailed, employs all elements and concepts. When documenting use cases, include the main scenario and the alternative ones. Are the basis for (automatic) code generation
  - Instance. Describes single scenario from the main and alternative ones

Main elements of the diagram

- Object
- Lifeline
- Message
- Execution specification

Object

- Sequence diagrams typically describe interactions between objects of classes
- Thy can also include instances of other classifiers: use cases, actors, signals etc.
- They are depicted using a rectangle with a name (sometimes underlined)
- In trivial diagrams they are all placed along the top line of the diagram

Lifeline

- Represents life span of the object
- Is depicted by a dashed line going from the object downwards

Messages

- Message represents information exchange between objects. It is an order from one object to another to perform some operation(s)
- The complete syntax is as follows:
  `predecessor/sequence_expression signature`
- Only signature is compulsory (and only part of it)

Messages - predecessor

- Predecessor is the number of message that has to appear (some time before) in order for this message to be executed
- When more that one message has to appear before the current one, they can be all listed, comma separated

Message – sequence expression

- May contain:
  - Message identification (number or name)
  - Message recurrence or iteration sequence
- The above fields are separated by colon
- Message recurrence is expressed as `[actualCondition]`
- Message iteration sequence is expressed as `*[iterationSpecification]`
- Examples: `1.2:[x>15]`
  `initial:*[i:=1..15]`

Message - signature

- May consist of:
  - Name (compulsory)
  - Arguments list
  - Return value

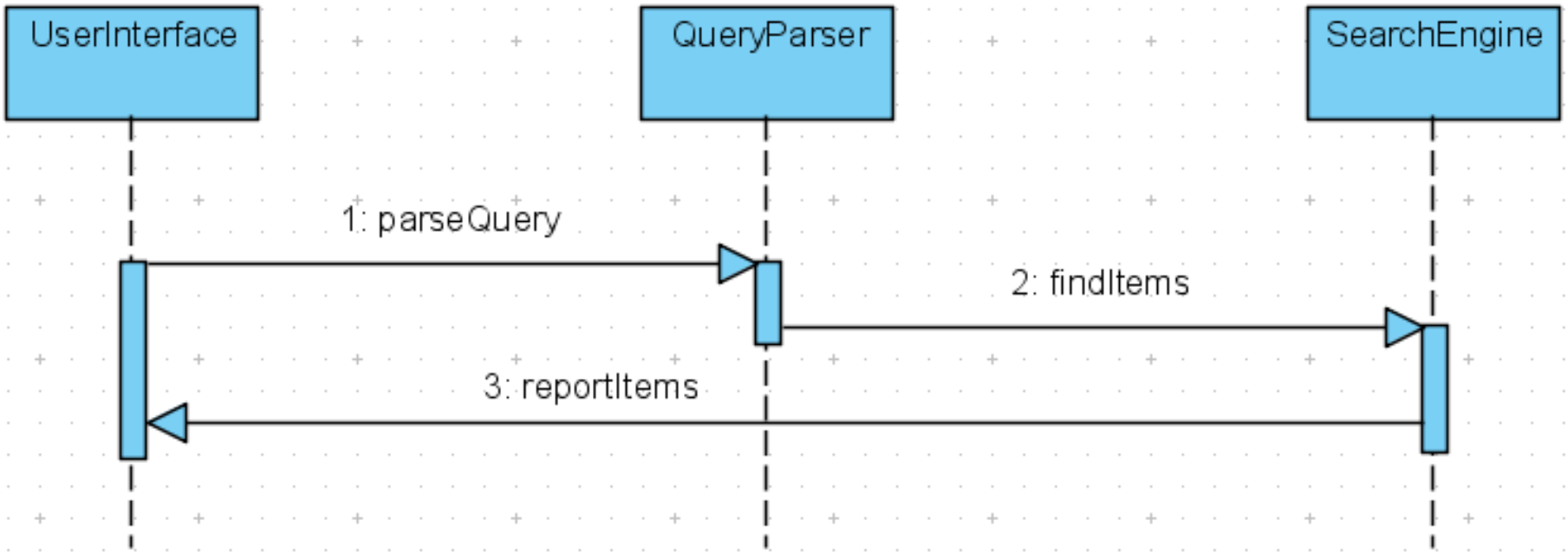
Message - signature

- Name indicates the operation that will be performed by the receiver of the message
- Arguments list and return value may be specified similarly as in class specification, but represent actual, not formal parameters (and types are not included)
- Return value makes only sense in messages that result in passing data to the caller
- Example:
  `findItem(name):itemsList`

Execution specification

- Represents activity period of an object (computations, message passing from/to an object)
- Is depicted by a rectangle placed on the life line, the height representing timspan of the activity
- The beginning is a result of an activation (often result of received message), end – of a deactivation

Sample diagram



Message types

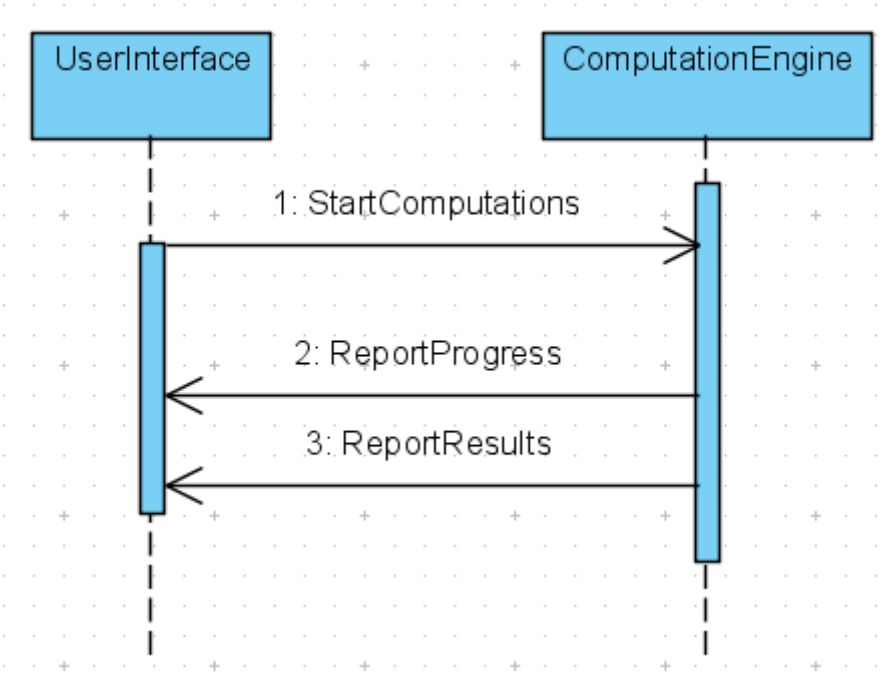
- Synchronous
- Asynchronous
- Return
- Lost
- Found

Synchronous message

- Control is passed to the called object
- Control flow of the sender is suspended until called action is executed
- Is depicted by a solid arrow
- This translates to a typical function call

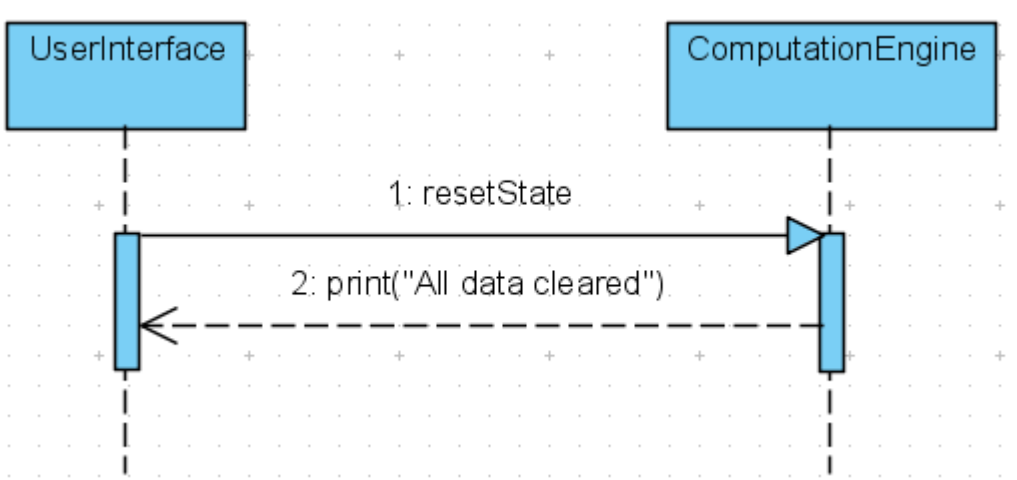
Asynchronous message

- Control flow of the caller is not interrupted
- Is depicted using “open” arrow
- Is possible when the caller and callee are not in the same thread



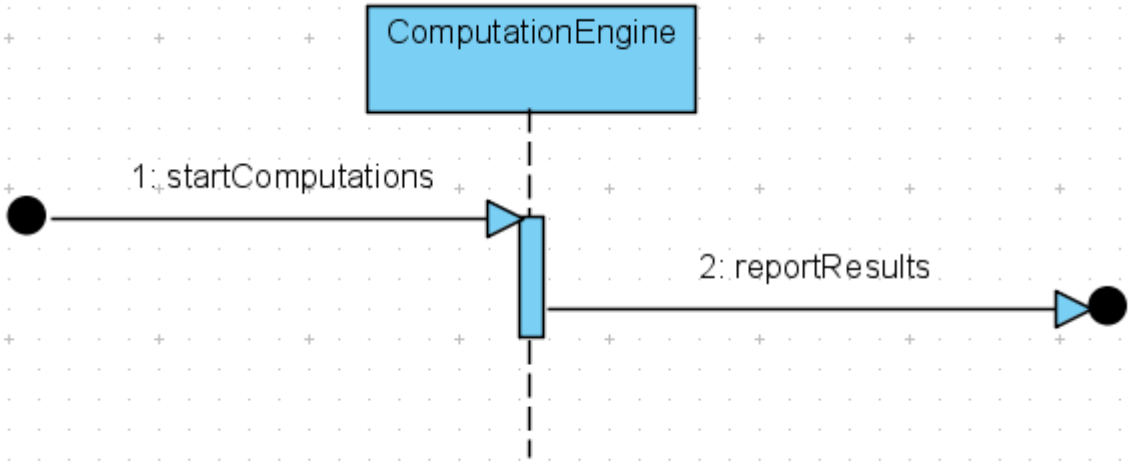
Return message

- Indicates return of the control flow to the caller of previous message (makes sense in case of synchronous messages)
- Is not required
- Also indicates that a certain operation in the caller of the previous message is started
- Is depicted using dashed arrow



Lost & found messages

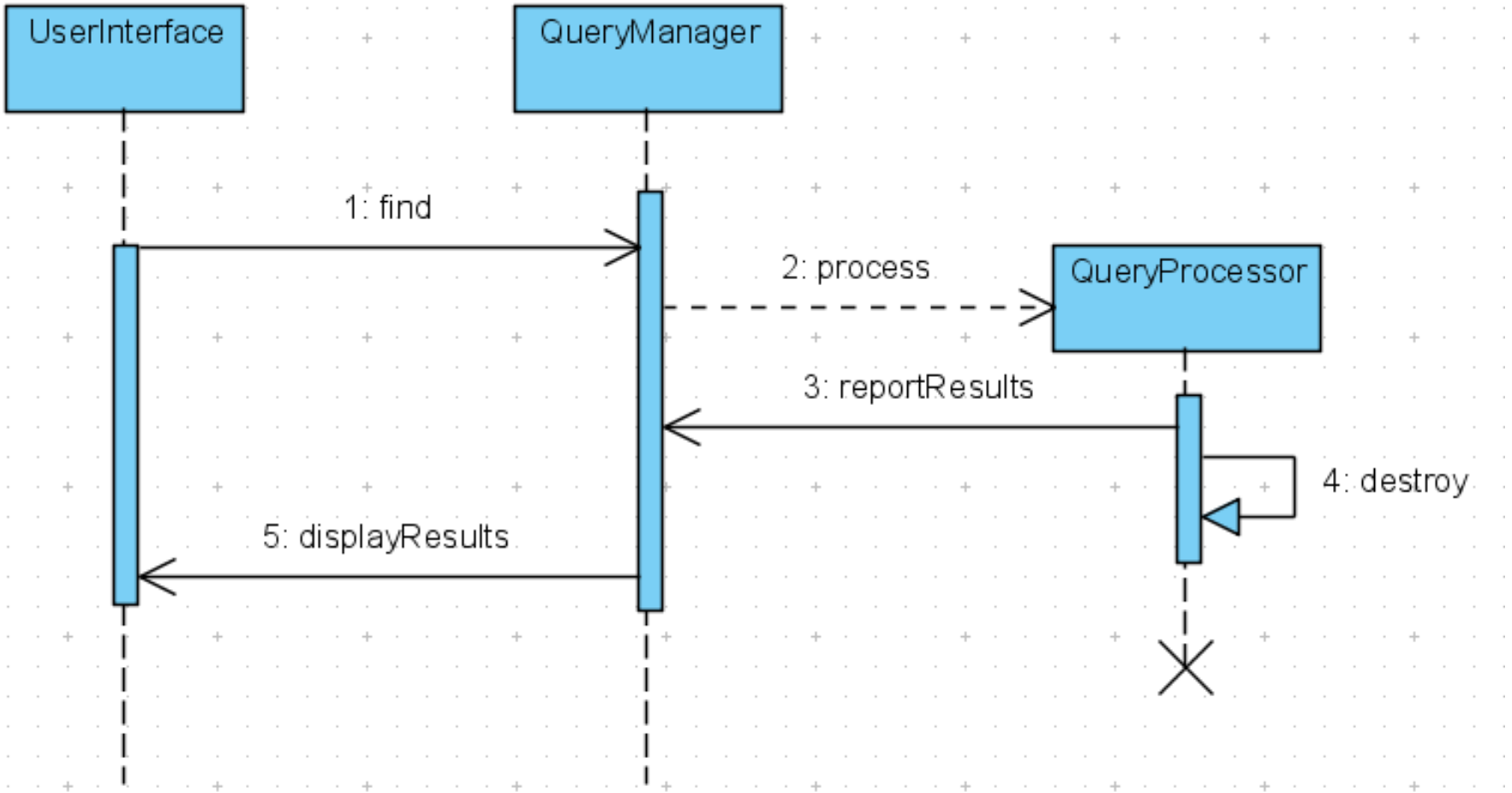
- Lost & found messages are useful when the caller (found) or callee (lost) are not known during creation of the diagram
- This is common in large system
- These messages are depicted by placing a solid circle in place of the unknown object



Creating and destroying objects

- Creation and destruction of objects can be depicted in the diagrams
- This is marked by adding **create** or **destroy** stereotypes to appropriate messages
- At the end of **create** message a new object has to be placed (which results in its placement being lower than of the “typical” objects)
- After callee receives **destroy** message, its lifeline is terminated. This is indicated by an X placed at the end of lifeline

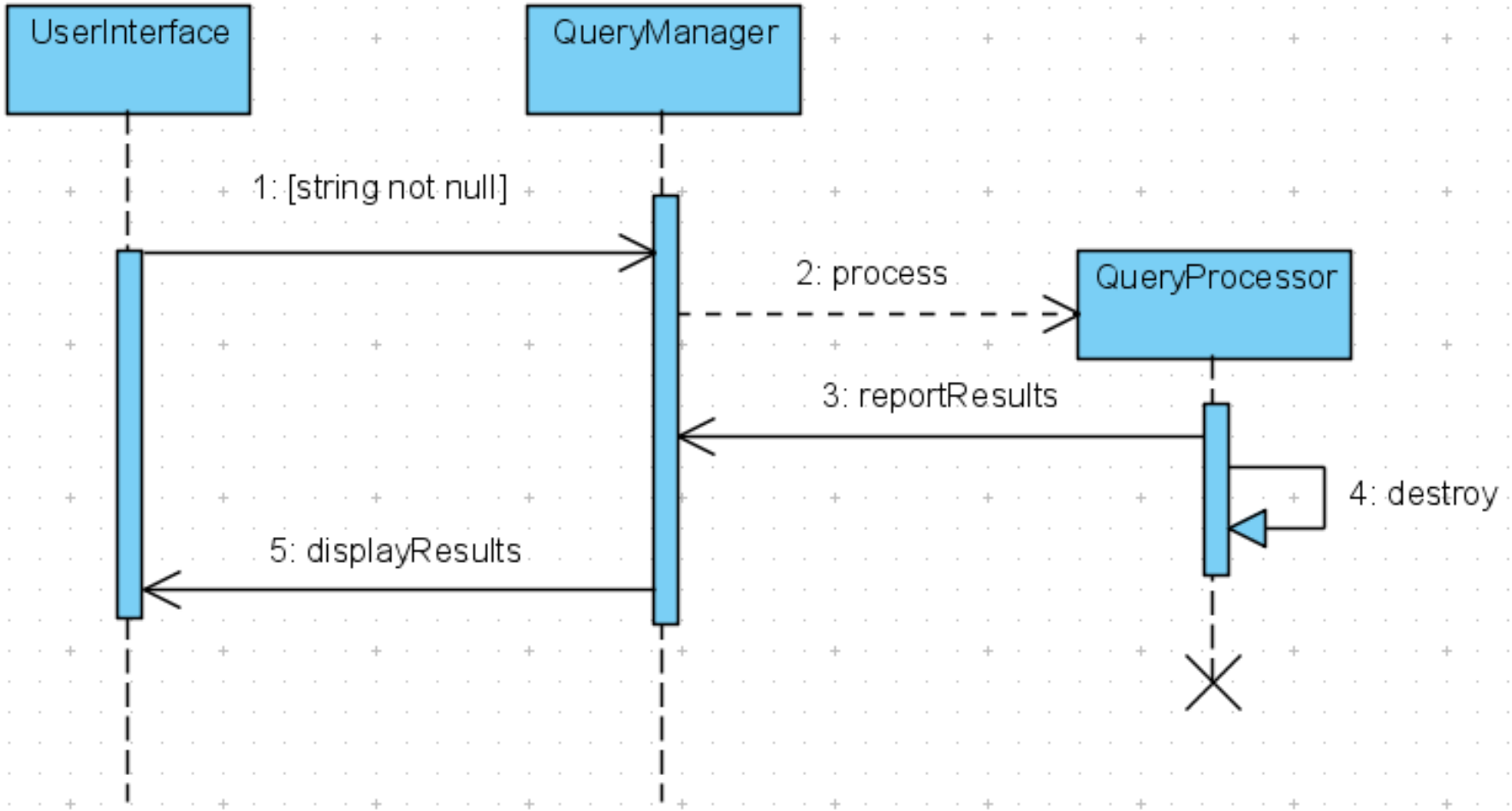
Sample creation and destruction



Conditional messages

- It is possible to specify the guard condition(s) under which the message is passed
- They are placed in square parentheses before the message specification
- It is possible to specify more than one condition
- If the condition(s) is/are not met, the message is not passed and the operation in the callee is not performed

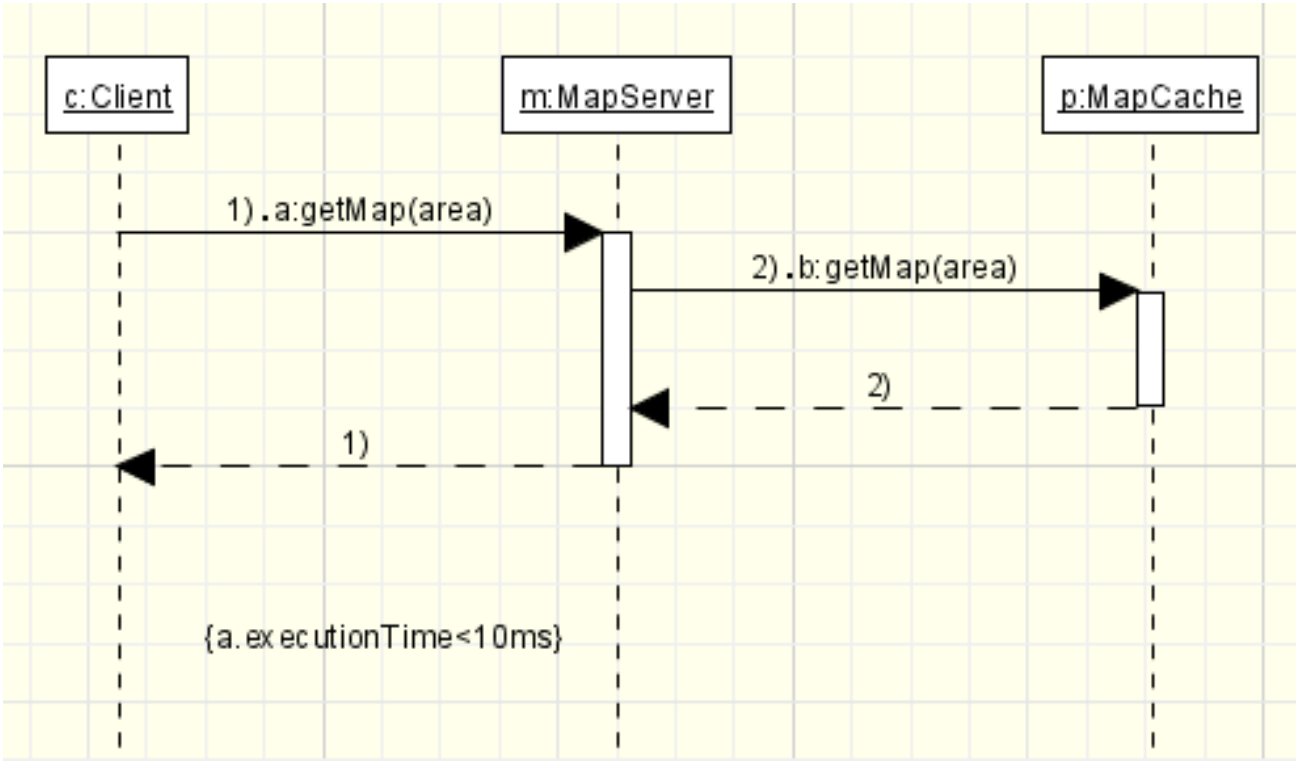
Sample conditional message



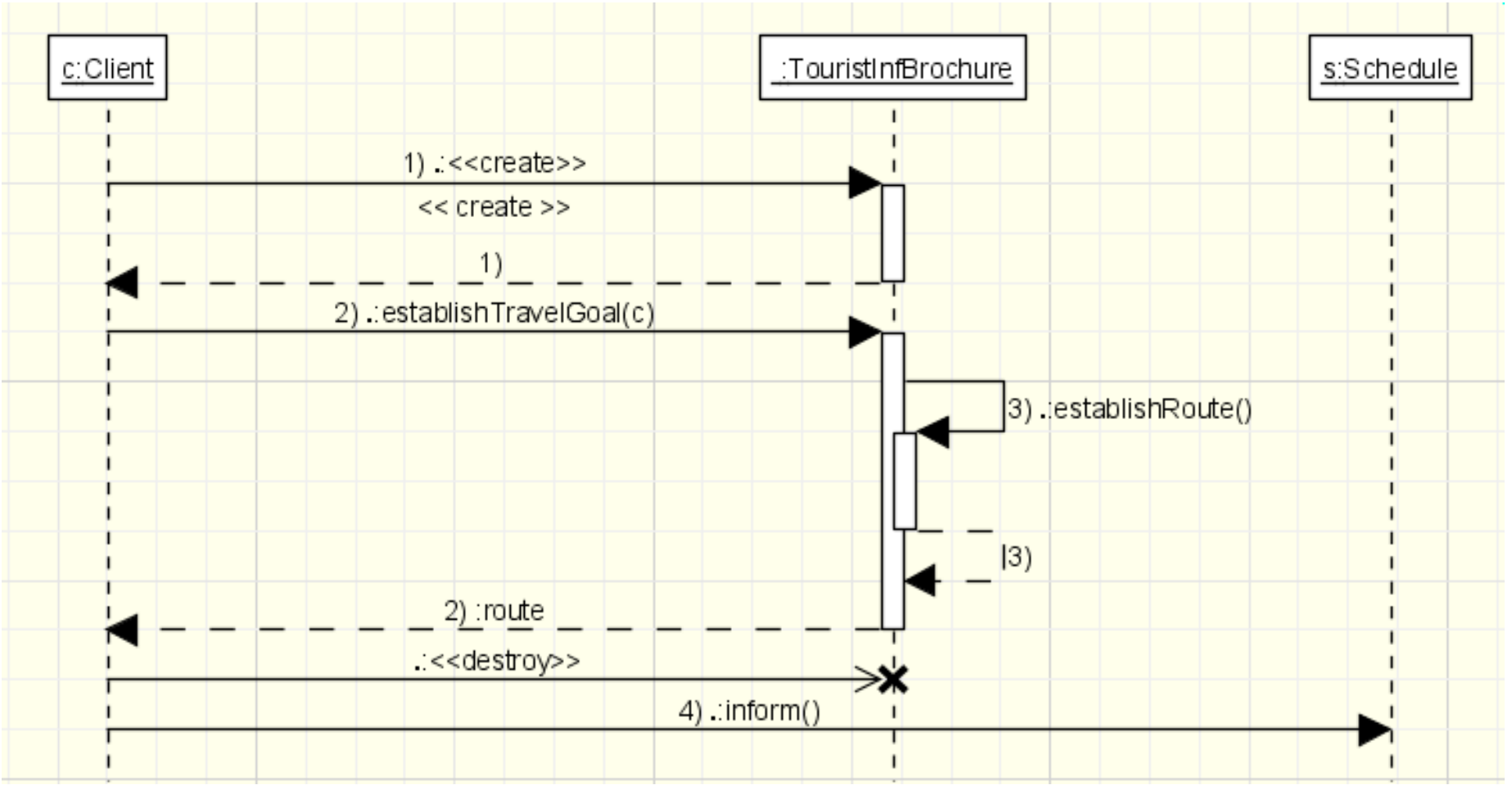
Branch

- Conditional messages allowed to model the situations when the message is either passed, or not passed
- Another approach is required when two or more different messages can be passed, depending on some condition
- It is possible to branch message depending on a guard condition
- The alternative messages may be passed to different object or to the same object – in the latter case the lifeline of the calee is split

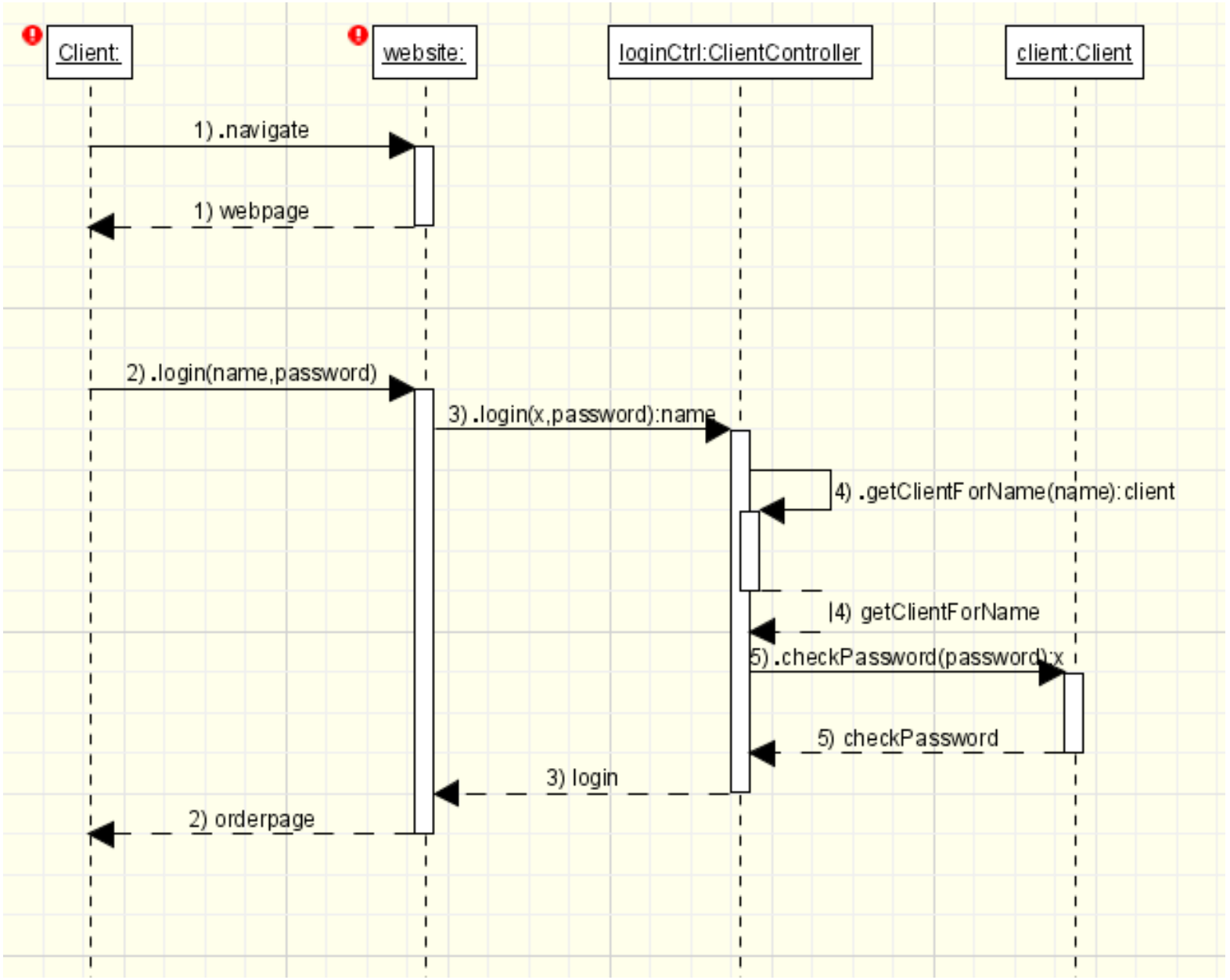
Sample diagrams



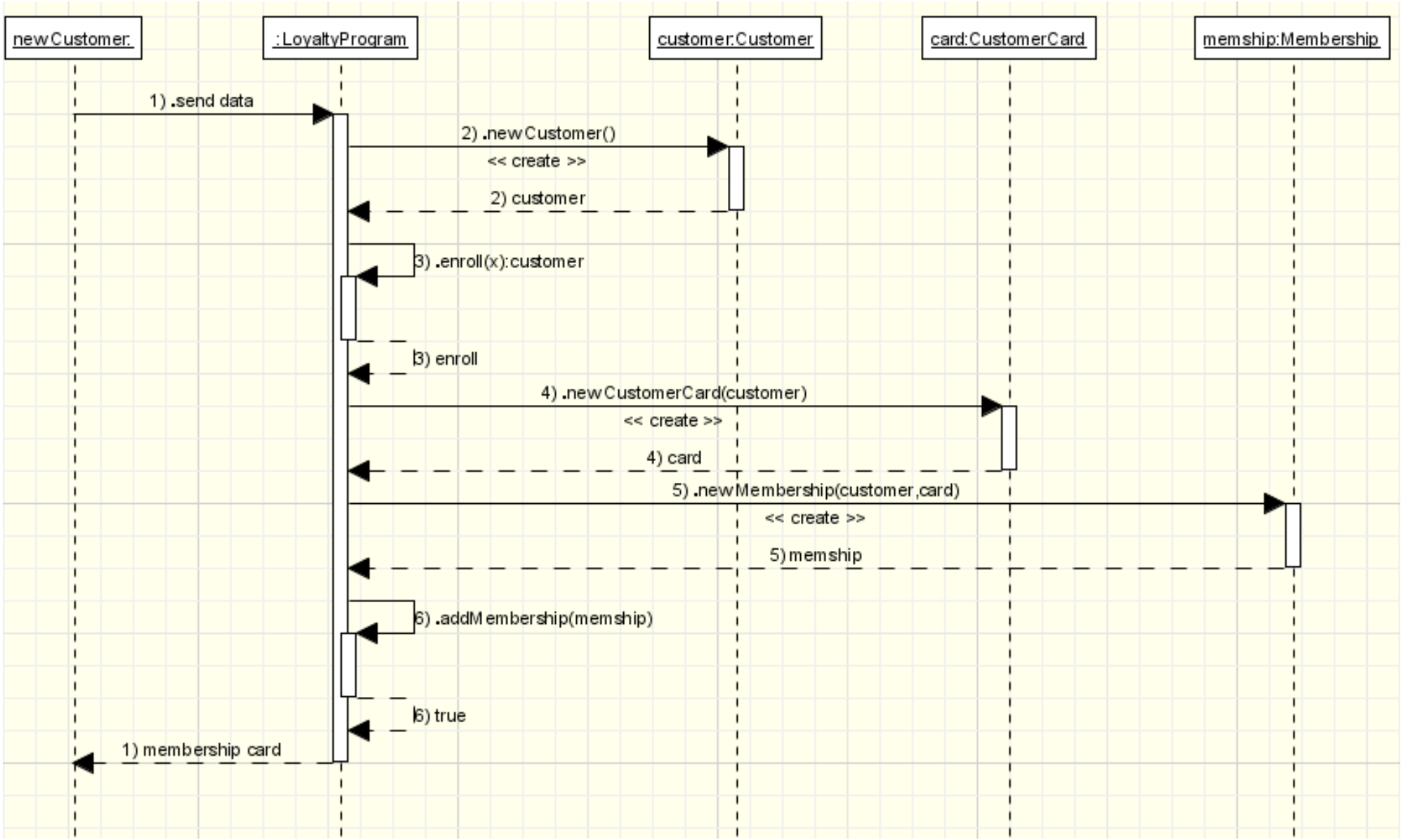
Sample diagrams



Sample diagrams



Sample diagrams



Combined fragments

- More complex concepts can be expressed using combined fragments
- These are selected parts of the diagram, characterised by interaction operator
- Graphically they are depicted in a similar fashion to the diagram documentation – by a frame encompassing a region with a header in the left top corner
- The header contains interaction operator and, optionally, parameters
- For some operators several operands, i.e. fragments, are present, separated by dash-dot line

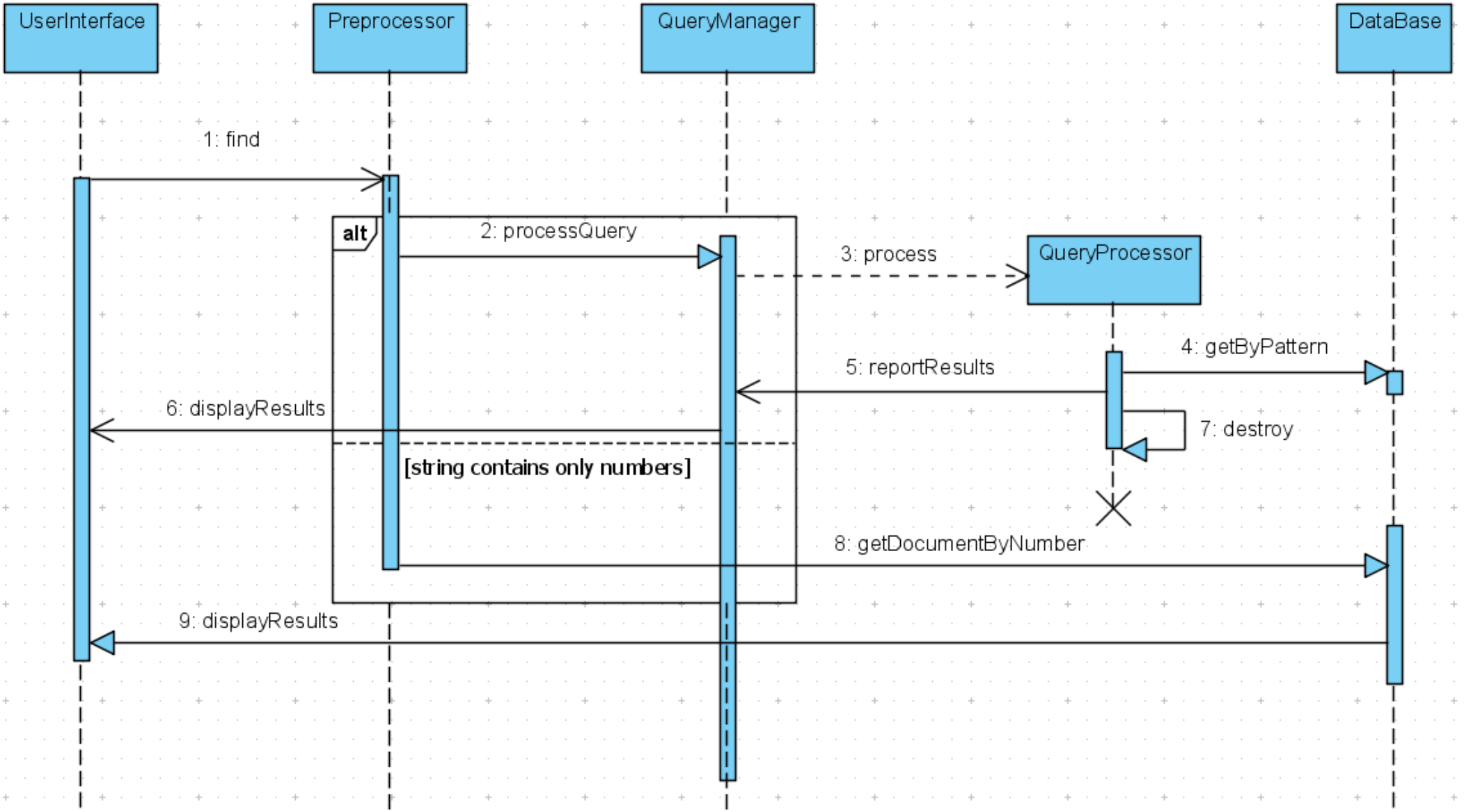
Combined fragments interaction operators

- Alt
- Opt
- Break
- Loop
- Neg
- Par
- Critical
- Assert
- Consider
- Ignore

Alt operator - alternative

- Indicates that only one of the fragment's operands (subfragments) can be selected
- Which operand is selected depends upon conditions placed in the operands (in square parentheses)
- The operand without a condition is the default one
- This concept can be used instead of branching

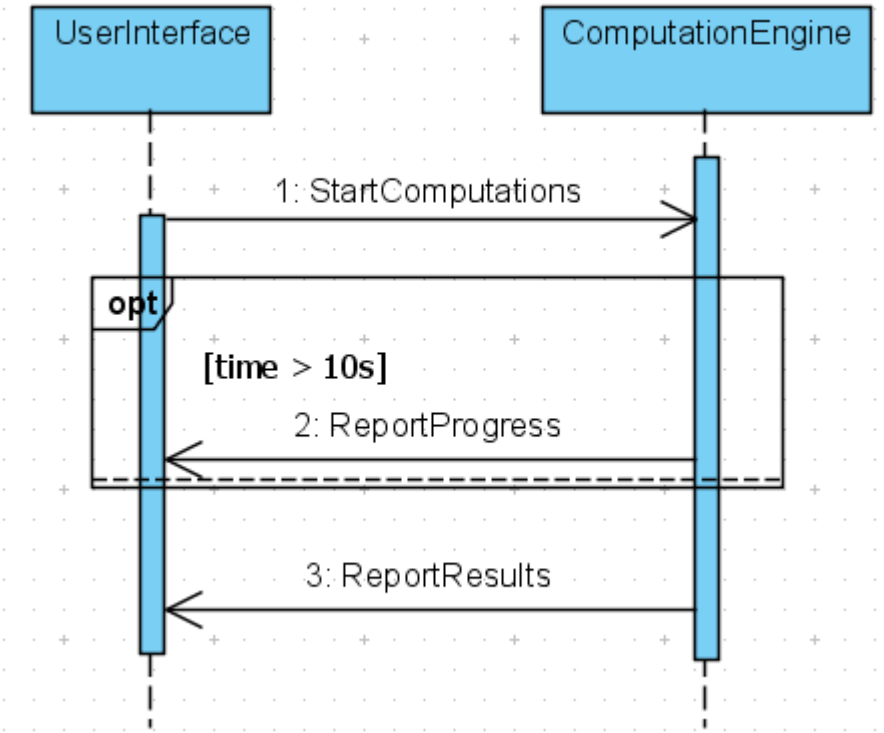
Sample alt diagram



Opt operator - option

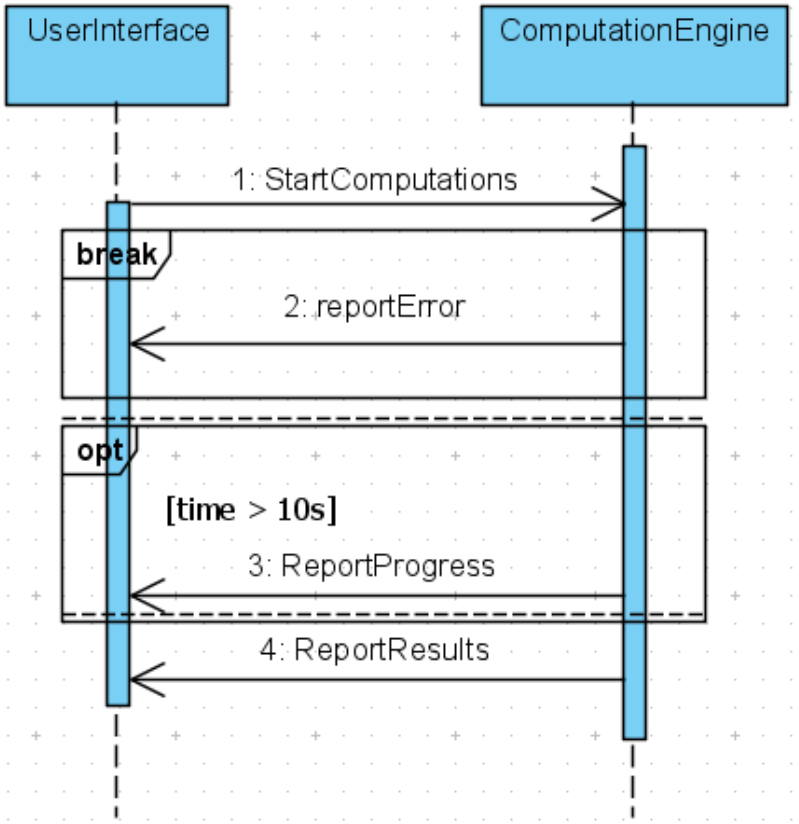
- Indicates that part of the diagram will be executed optionally, depending on the condition
- The condition is placed, is square parentheses, in the fragment in question
- This can be used instead of message condition

Sample opt diagram



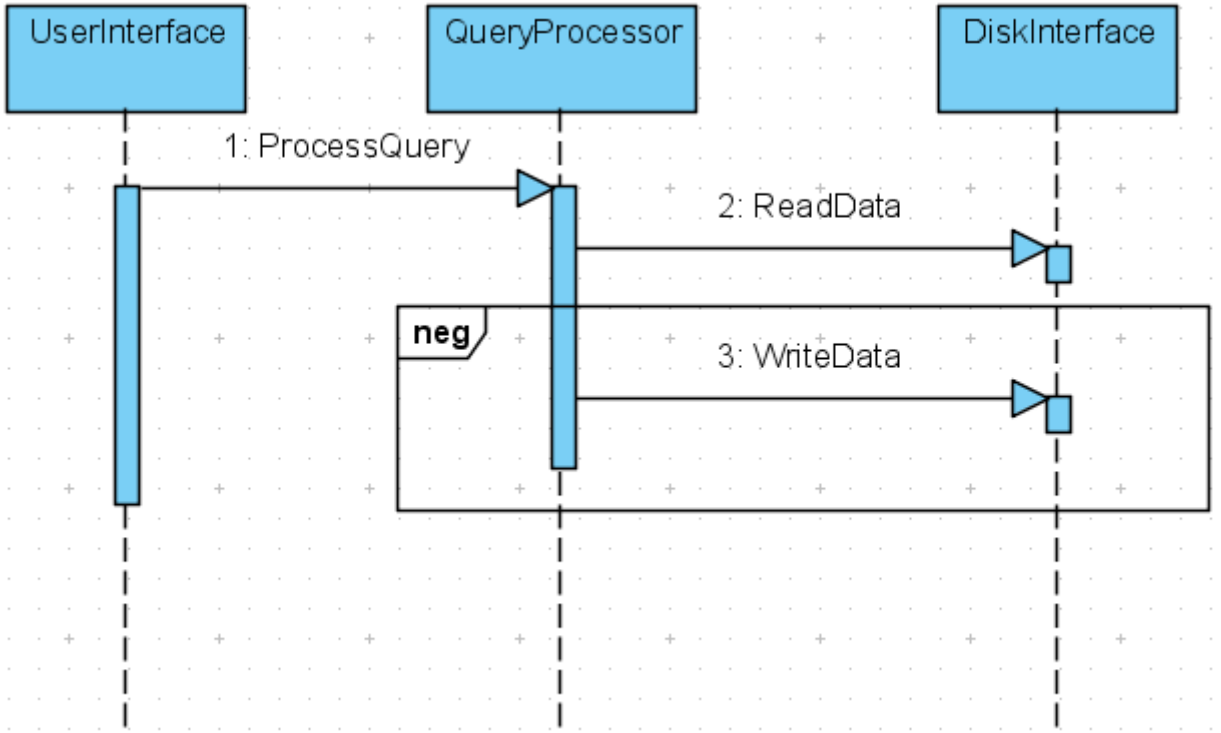
Break operator – execution interruption

- Break allows to define a fragment that will be performed in case a condition is met
- If the fragment is performed, the rest of the execution specification is skipped



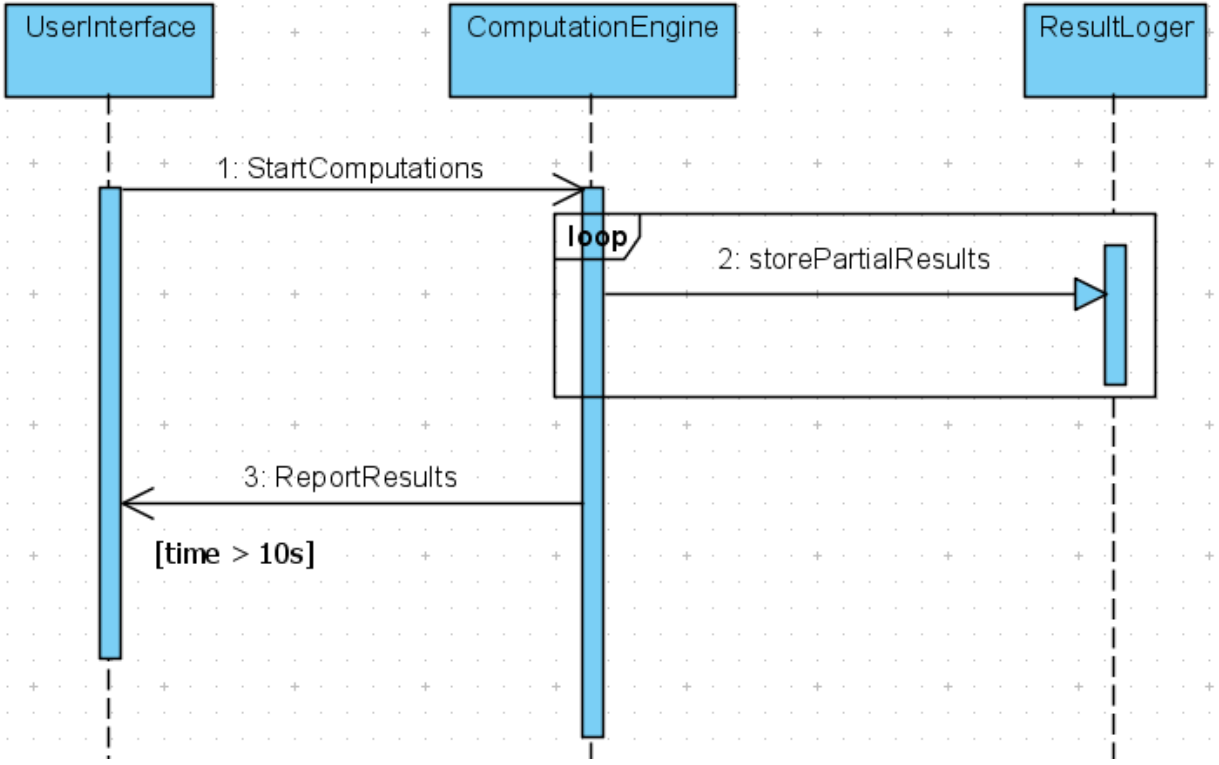
Neg operator – erratic behaviour

- Neg indicates fragment that should not be performed (if it **is** performed, it is treated as erratic behaviour)



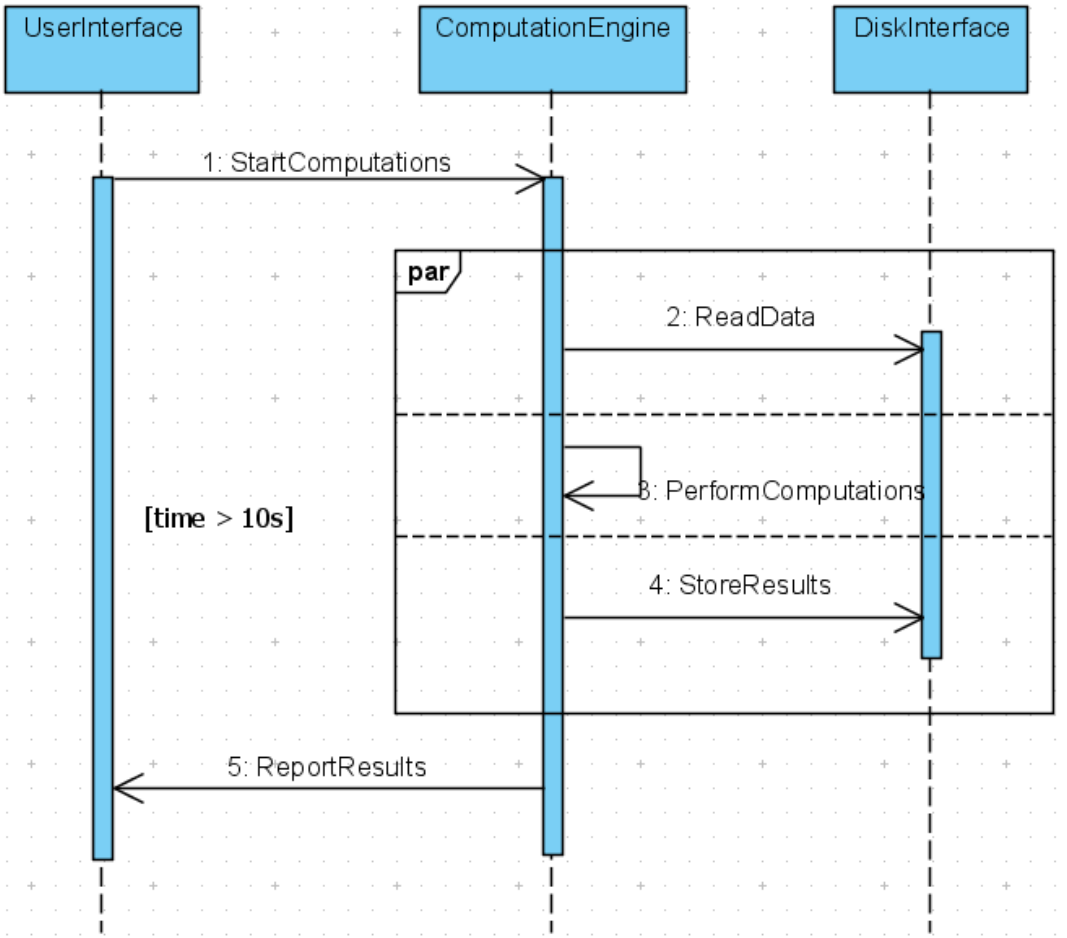
Loop operation - iteration

- Allows to repeat specified fragment a number of times
- The number of iterations can be specified as a parameter of the operator



Par operator – parallel (concurrent) execution

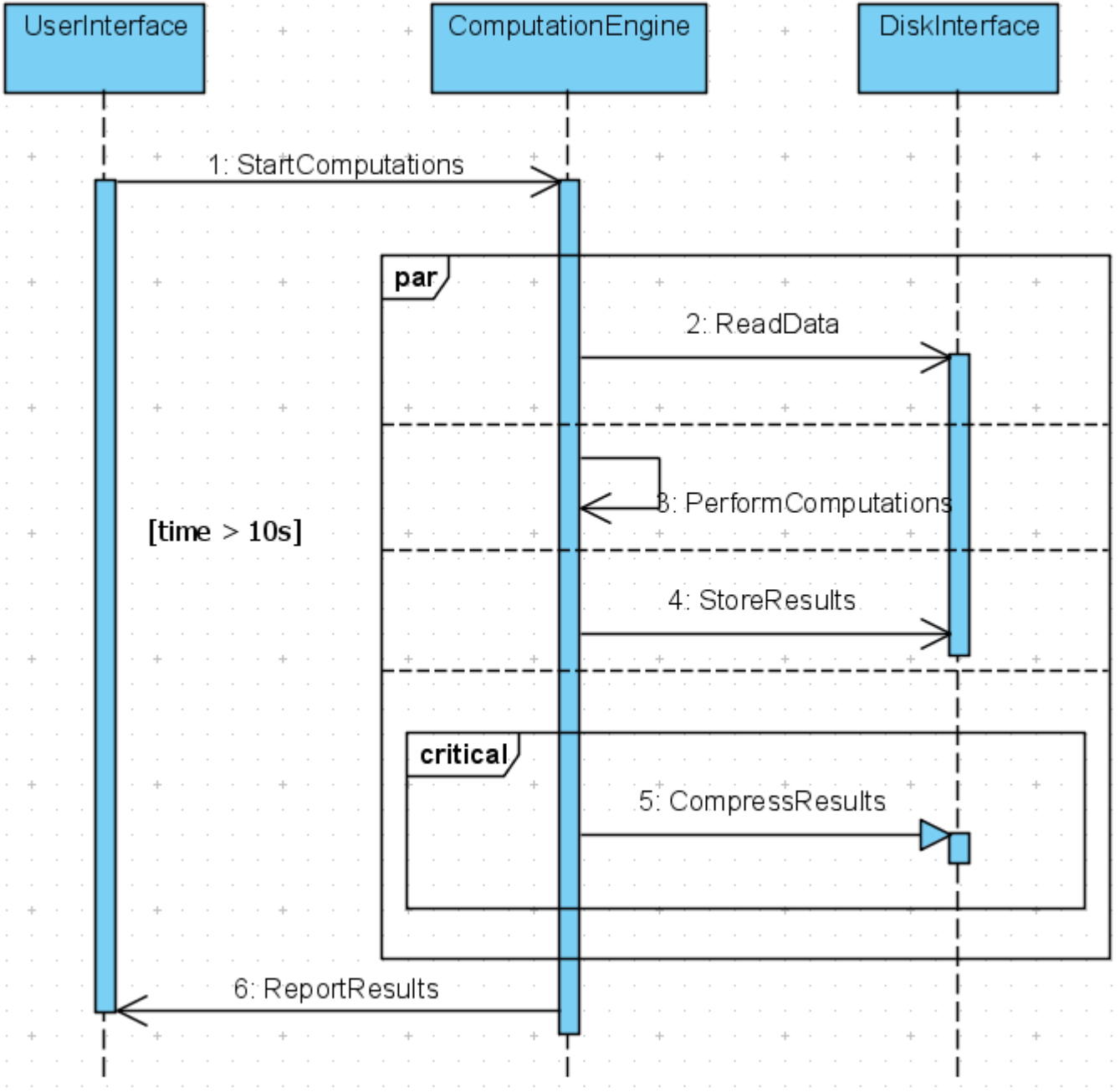
- Indicates that all operators of the region are performed concurrently



Critical operator – high priority fragment

- Indicates part of the diagram that, when executed, will block the objects that are included in the operator until the critical operation is finished
- Operations that involve other objects can be carried on

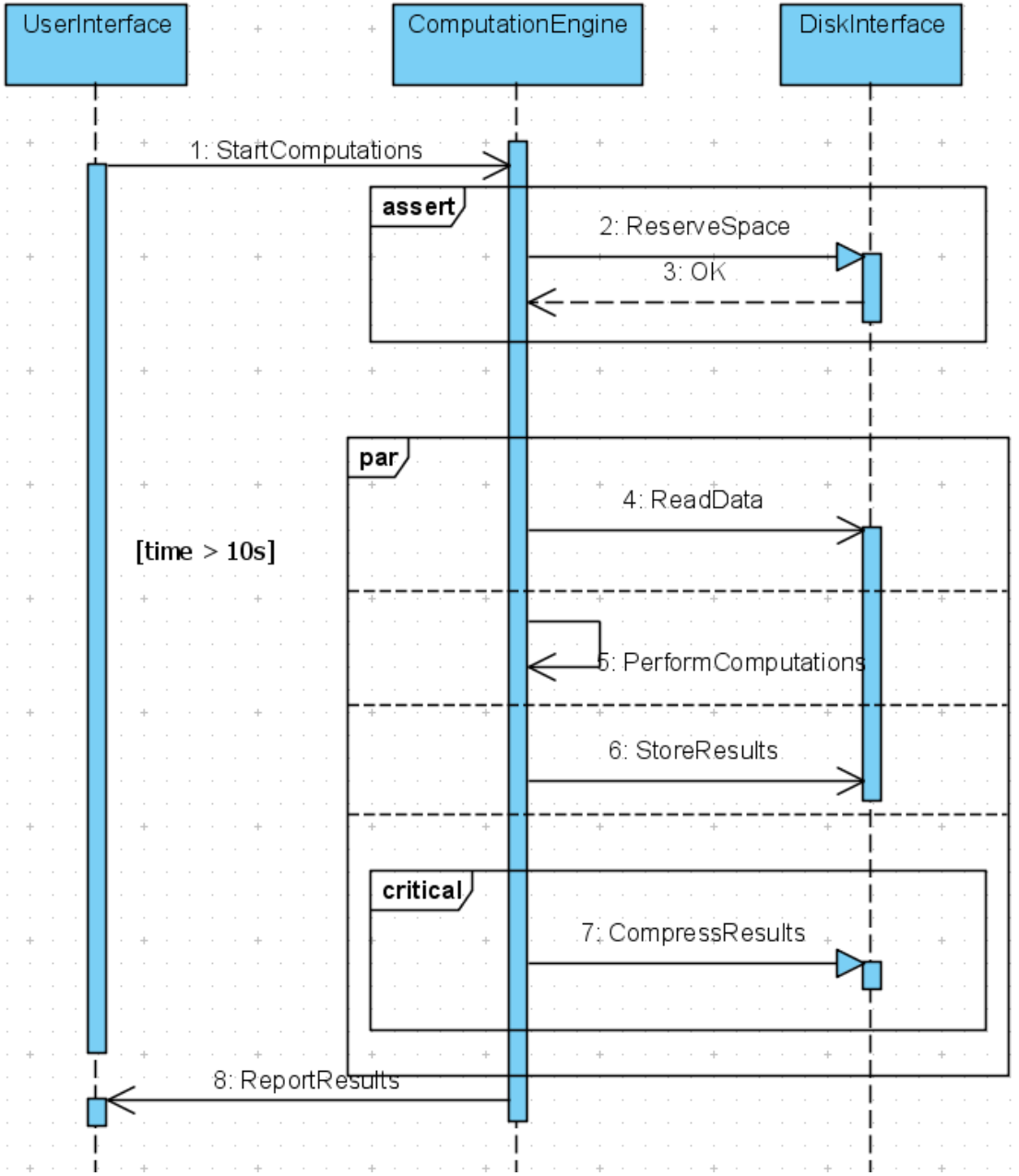
Sample critical operation



Assert operator – required sequence

- Allows to specify a sequence of messages that has to appear in the system exactly as indicated. In other words, this sequence is required, absence of it would be an error condition (cf. neg operator)

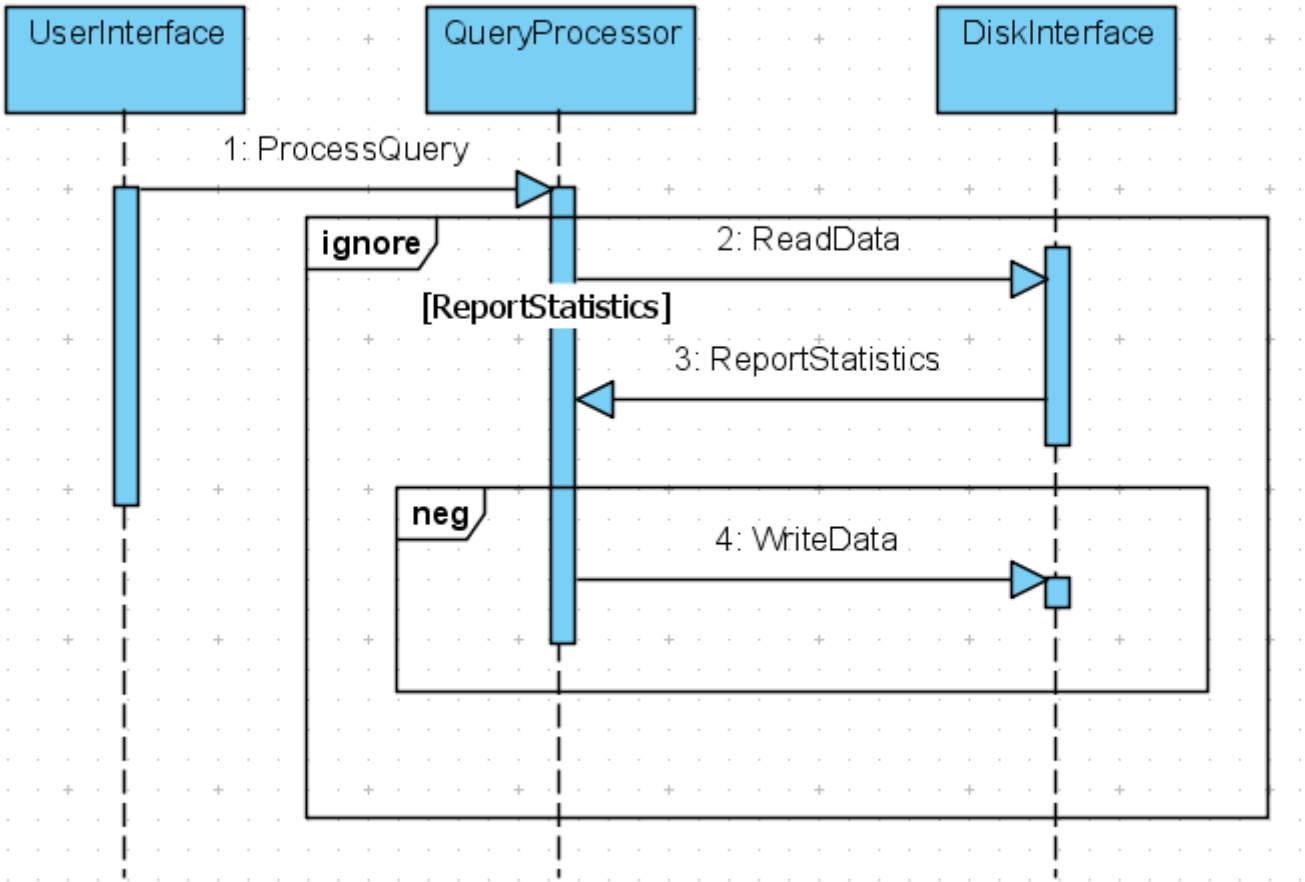
Sample assert operation



Ignore and consider operators

- Ignore operator allows to indicate operations (messages) that are not important to the execution process (they are often omitted in the diagram)
- Consider operator indicates operations that are important to the execution (it is the same as specifying all other messages as 'ignore')

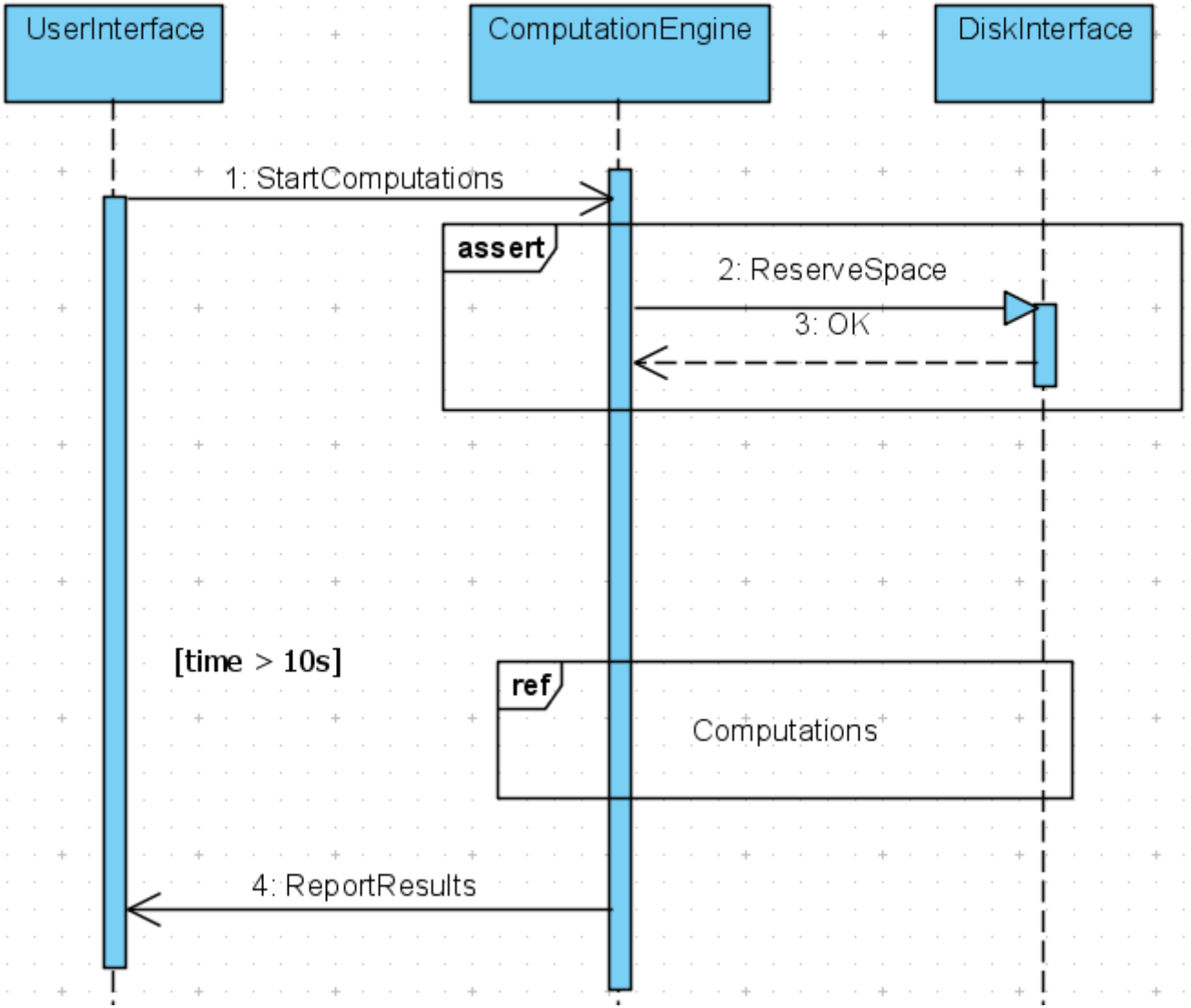
Sample ignore operator



Large diagrams

- In large systems it is impossible to place all interactions in one diagram
- It is possible to split the diagrams
- For example, it is possible to create a “main” diagram, containing only top-level interactions, and a number of sub-diagrams, containing details
- The sub-diagrams can be represented in the “main” diagram as interaction occurrence (depicted by ref region)

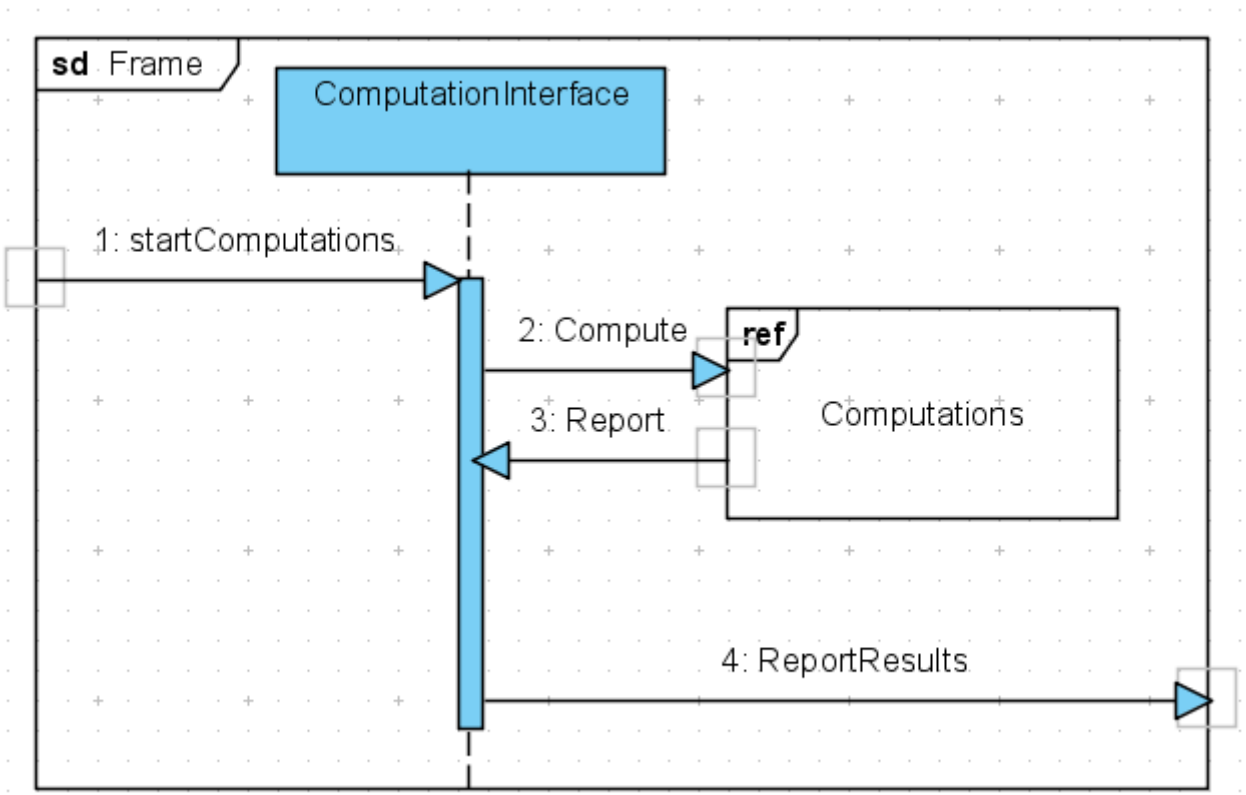
Sample ref region



Gates

- Are means of communication between (fragments of) diagrams
- Are represented by small squares placed on the edge of (fragment of) a diagram
- Allow important in large (fragmented) diagrams

Sample gates



Communication Diagrams

- Allow functionality similar to sequence diagrams
- Use different approach:
  - the messages are not explicitly ordered in time by the vertical axis,
  - the objects can placed anywhere in the diagram
- The only way to denote order of the messages is their numbering
- Communication diagrams can be translated (also automatically) to sequence diagrams, and vice-versa

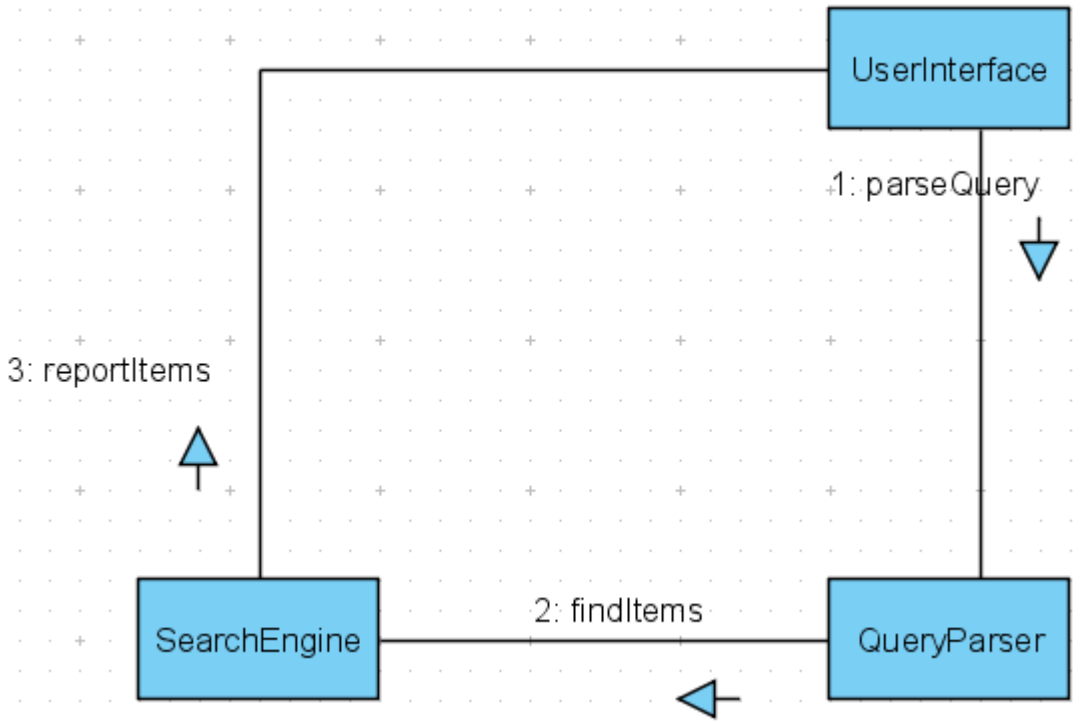
Limitations

- Not all concepts from the sequence diagrams can be presented in communication diagrams
- Missing concepts are:
  - Lost and found messages
  - Combined fragments
  - Gates

Elements of communication diagram

- Object – similar to the object in sequence diagram, but with no lifeline
- Link – shows that two objects communicate (exchange messages). Does not denote any actual message passing. Depicted as a line between two objects
- Message – similar to message in sequence diagram. Depicted as a short arrow with message description next to the link

Sample communication diagram



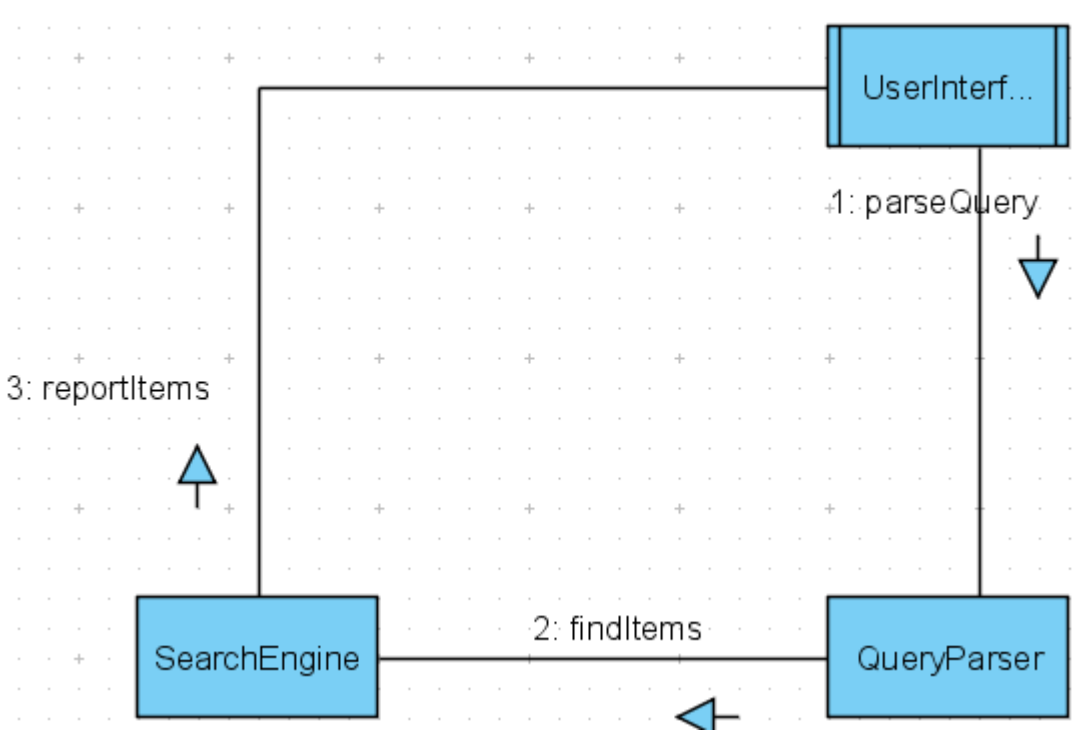
Messages

- When many messages of the same type are passed between two objects, they may be represented by a single arrow with multiple descriptions
- Numbering with dots (1.1, 1.2 and so on) may be used to group messages
- By using the same predecessor for a number of messages parallel processing can be documented

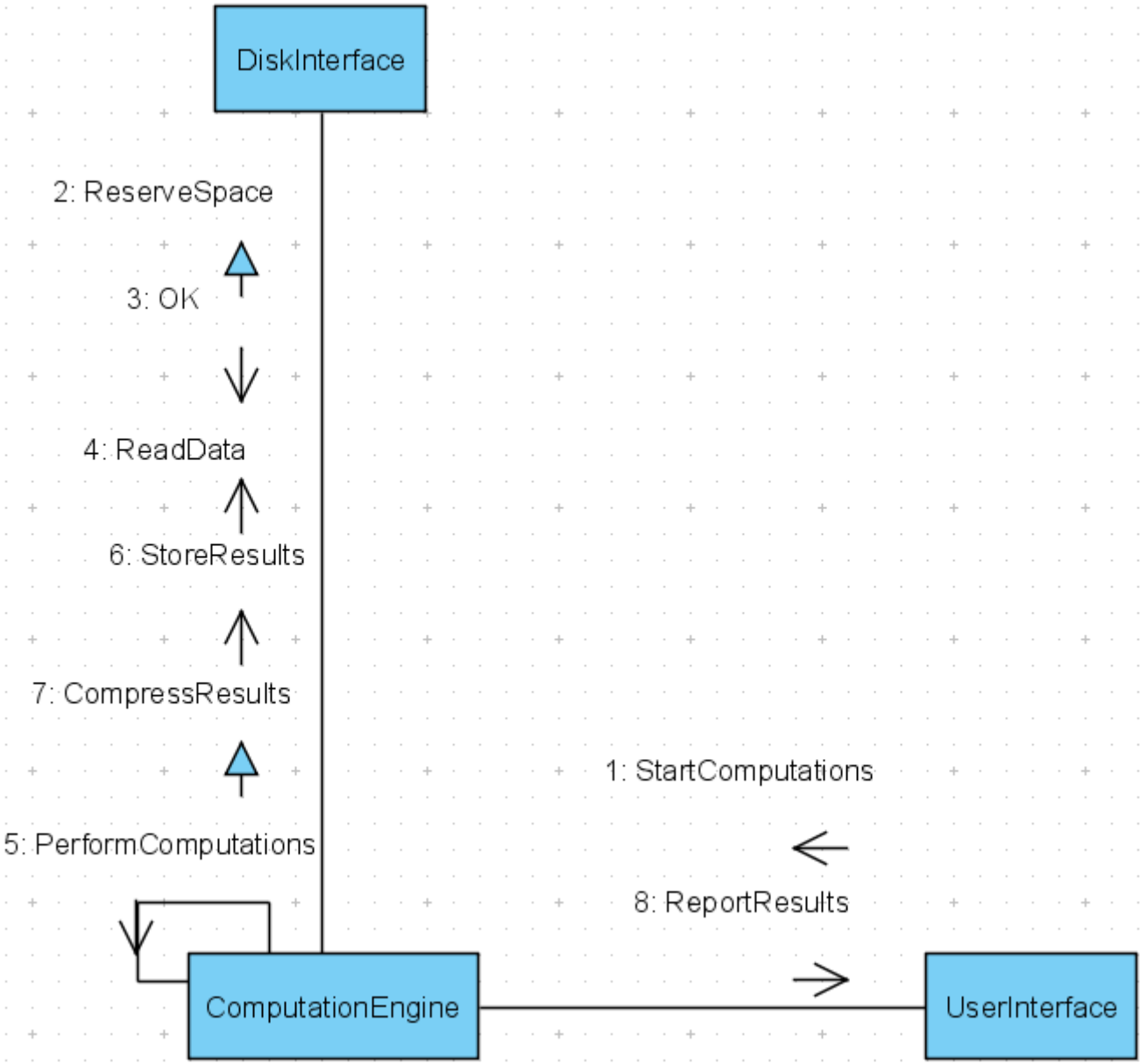
Objects

- When message is passed to a all objects of the same class, this can be shown by replacing the typical object symbol with multiple objects symbol (three rectangles placed in a pile)
- A special type of object is that belonging to an active class. Such object can initiate message-passing sequence. It is depicted by double vertical edges

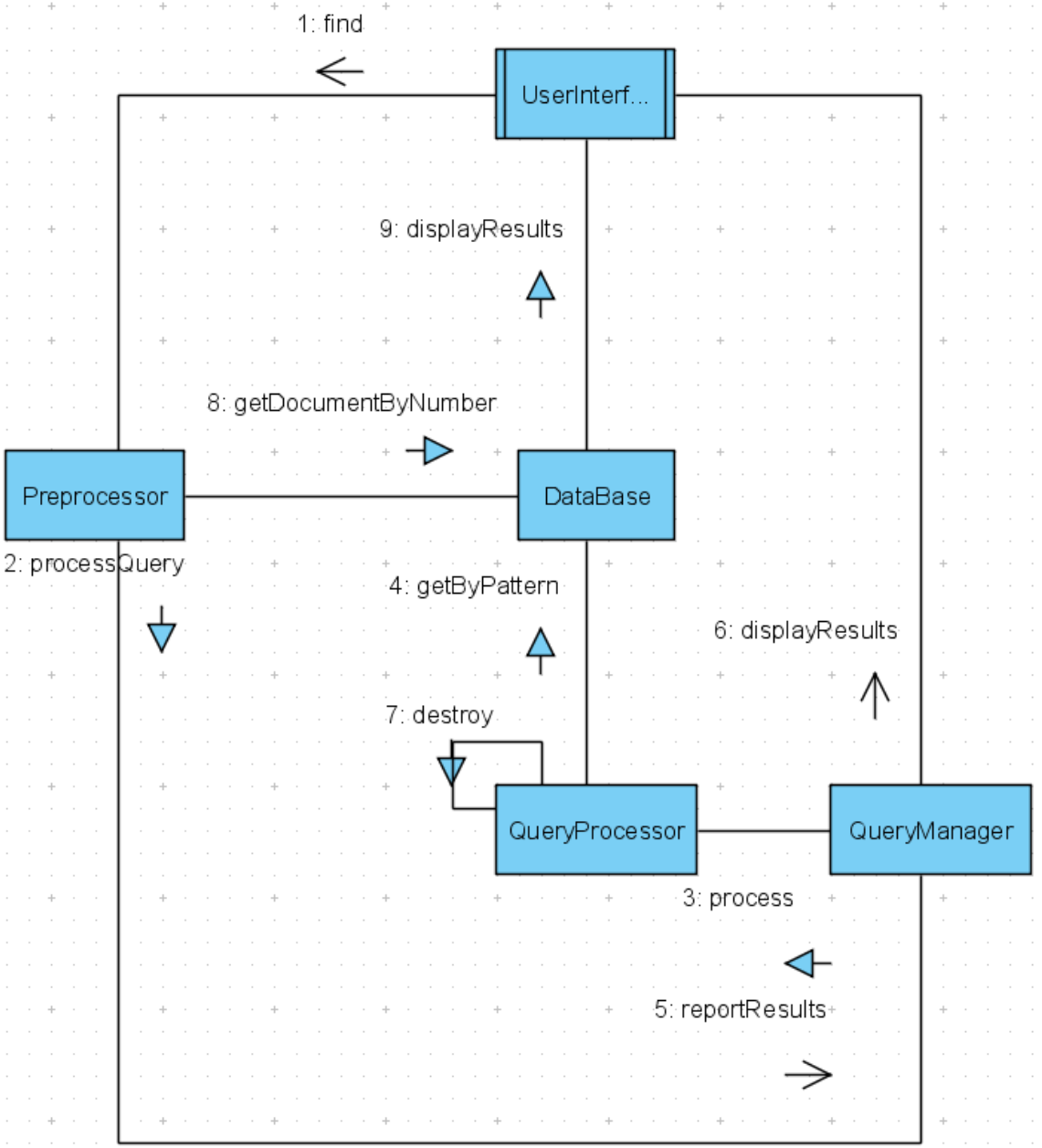
Sample active class object



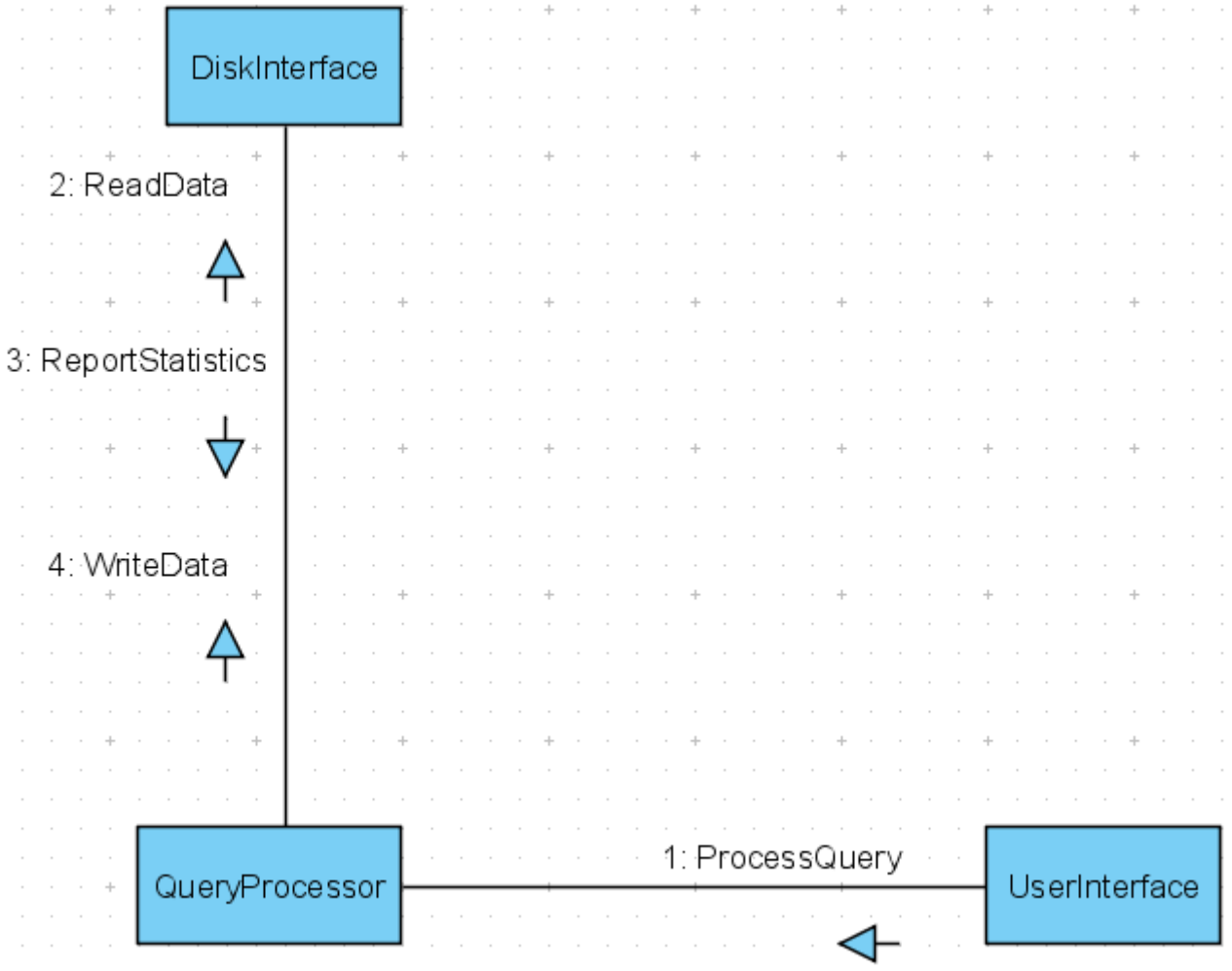
Sample communication diagram



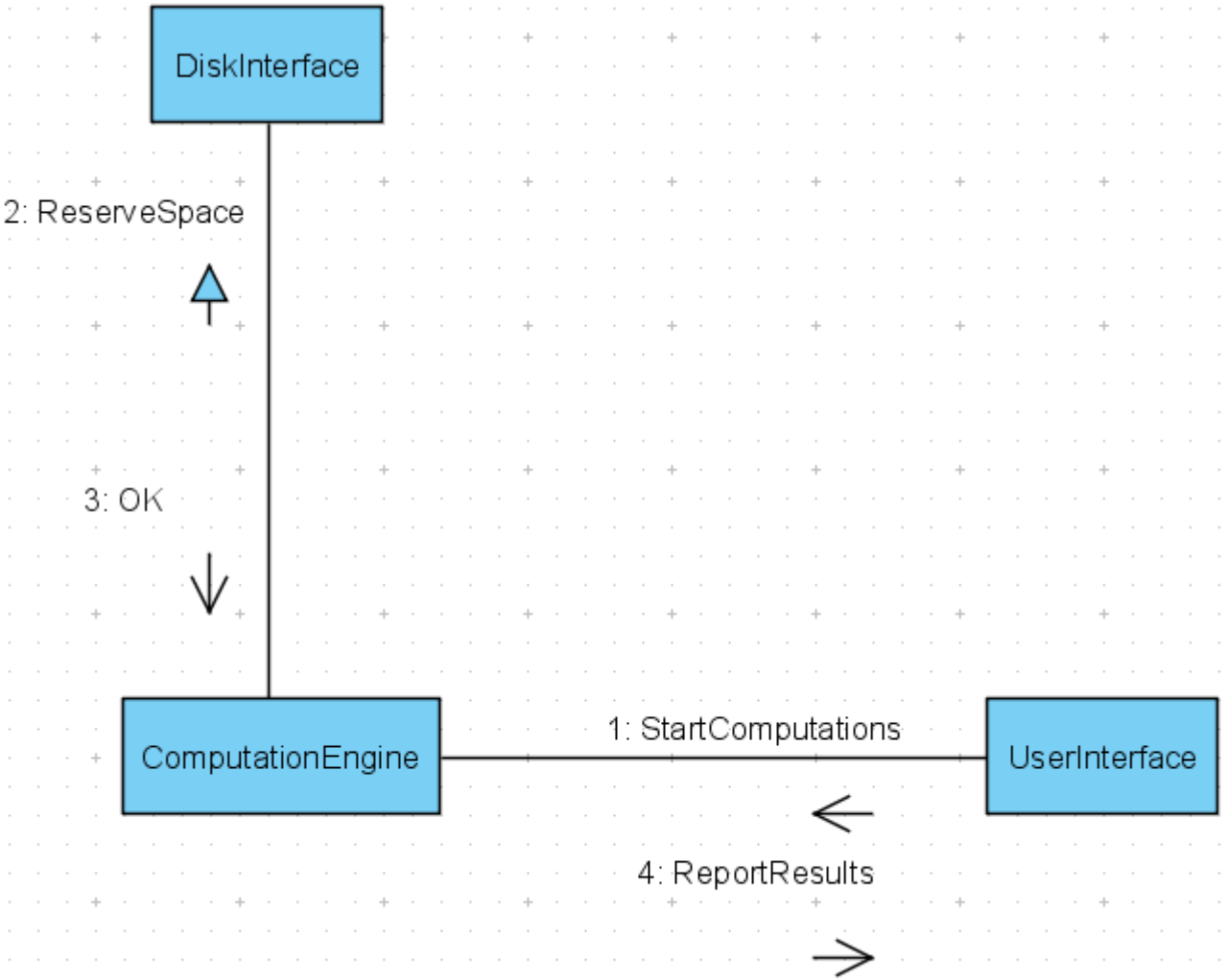
Sample communication diagram



Sample communication diagram



Sample communication diagram



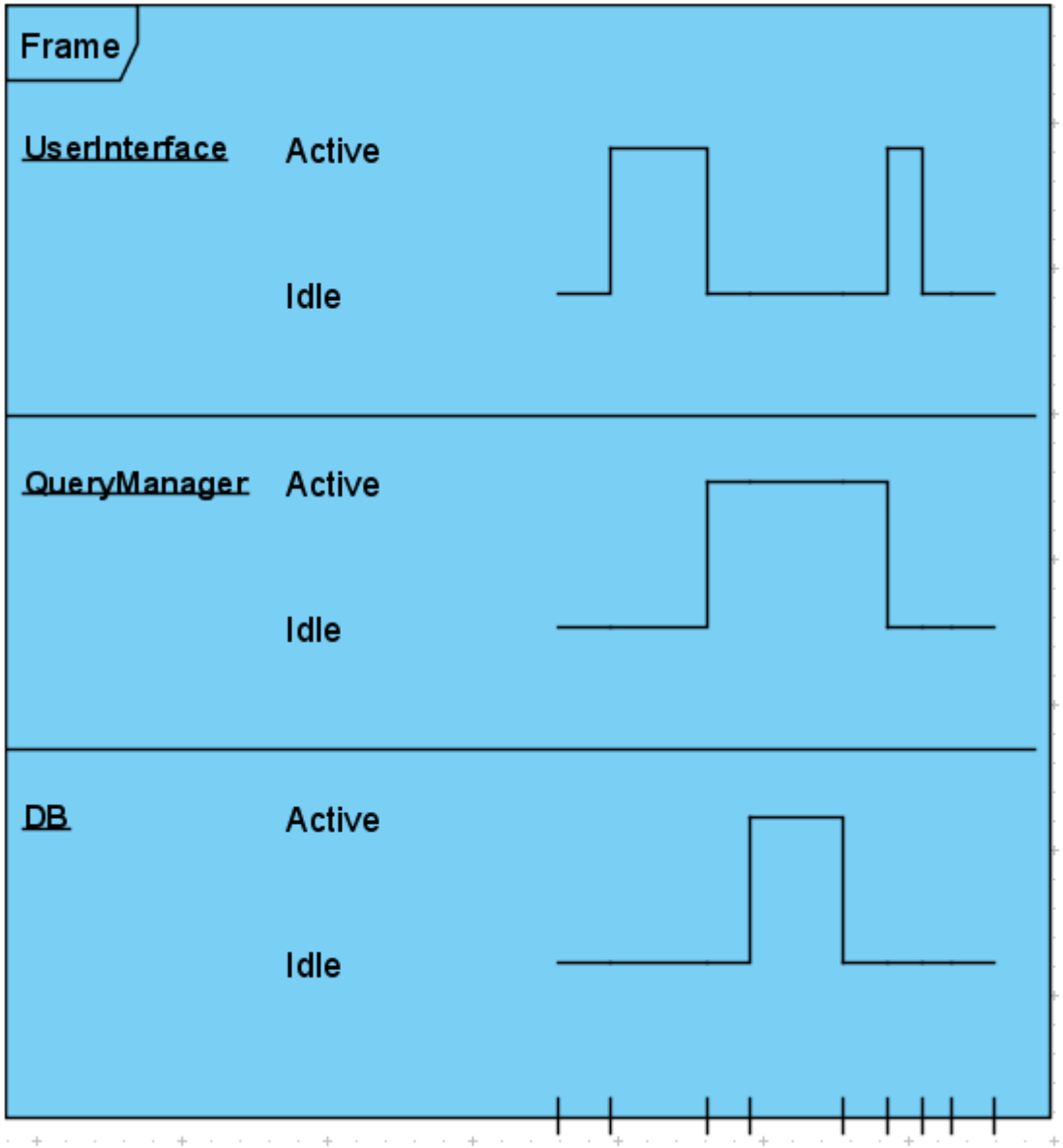
Timing diagram

- Used to describe state changes of object(s)
- The state changes are described in strict relation to **time**
- They are especially useful for systems where timing of operations is crucial (multimedia, real-time applications)
- They are related to machine state diagrams and sequence diagrams

Timing diagrams

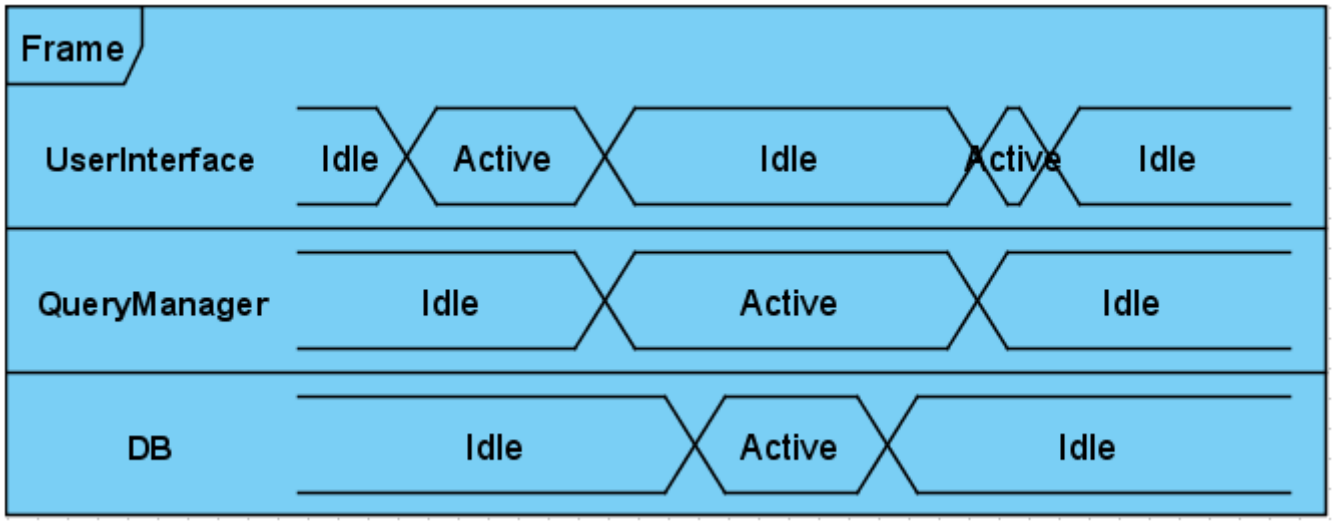
- Basic diagram consists of a timing frame, containing:
  - Time scale (horizontal axis, lower edge)
  - Names of classifiers for which the states are depicted
  - Names of states (for each classifier). Typical states are: idle, active, waiting, computing etc.
  - Lifelines (for each classifier) showing the state changes in time

Sample timing diagram



Alternative notation

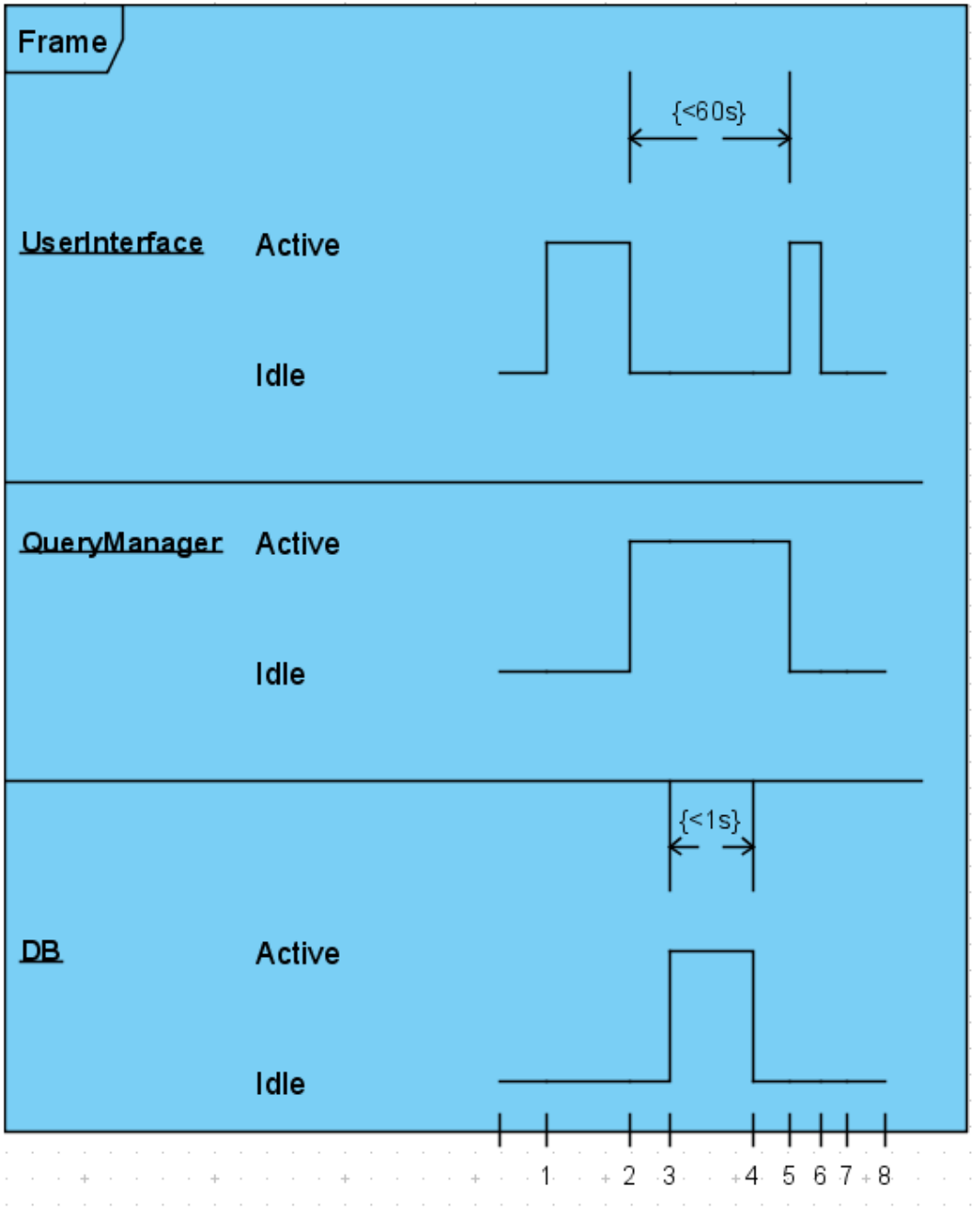
- Timing diagrams can be presented in alternative, compact notation
- Not all elements are possible in this notation



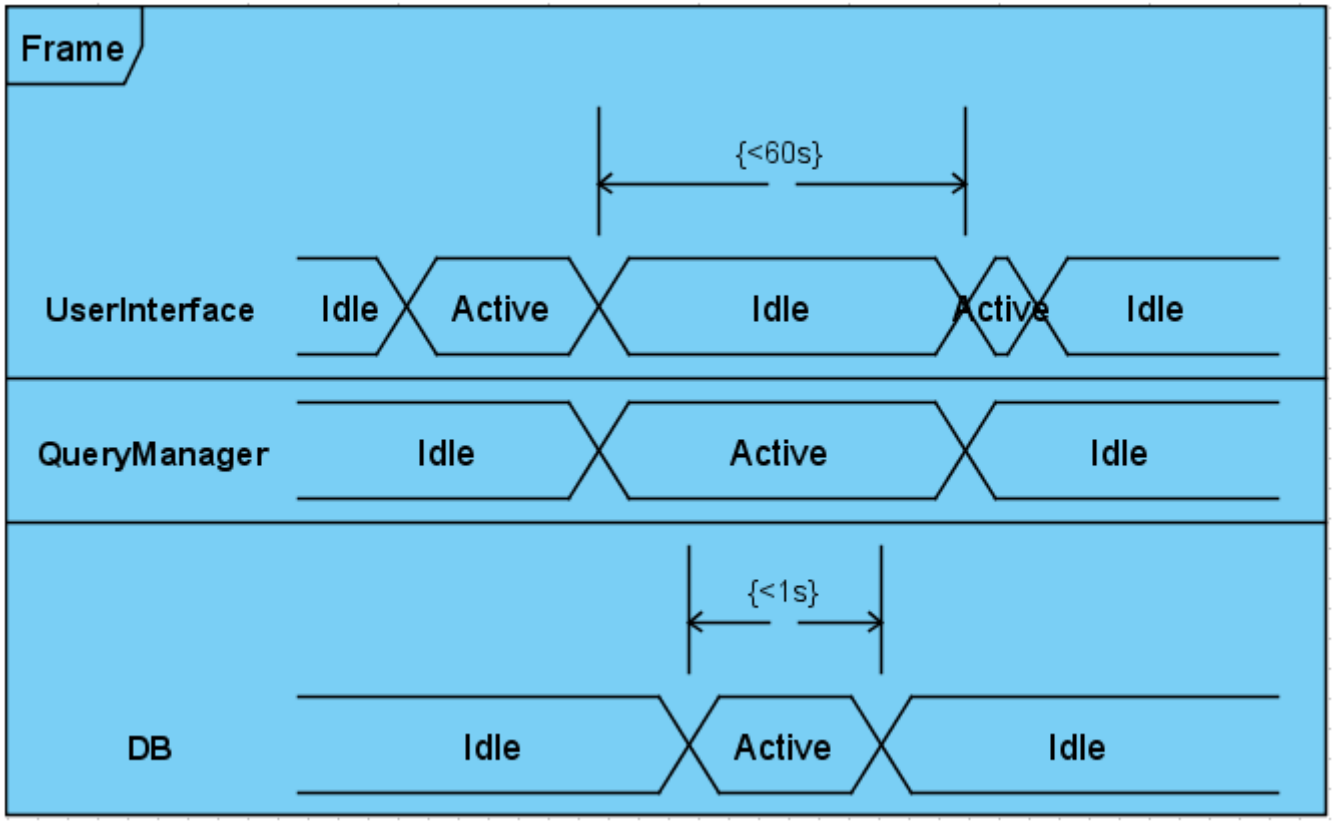
Time constraints

- It is possible to specify constraints that precisely control duration of the states
- The constraints can contain any kind of expression, they are placed above appropriate lifeline
- They can be displayed in both full and compact modes

Sample time constraints



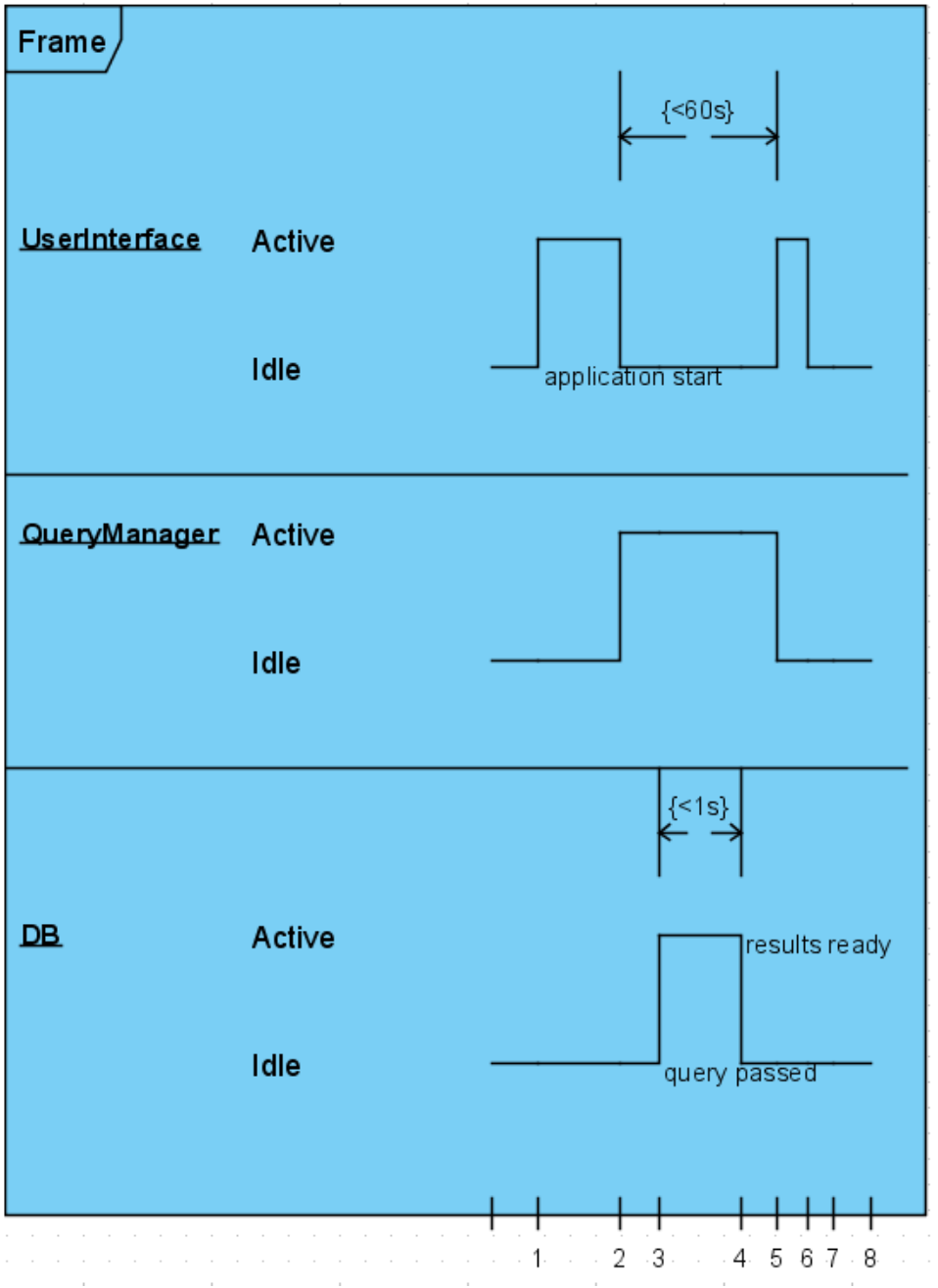
Sample time constraints



Stimuli

- It is possible to describe events that result in state changes
- They are simply displayed as expressions next to the point of state change
- They are not visible in the compact mode

Sample stimuli



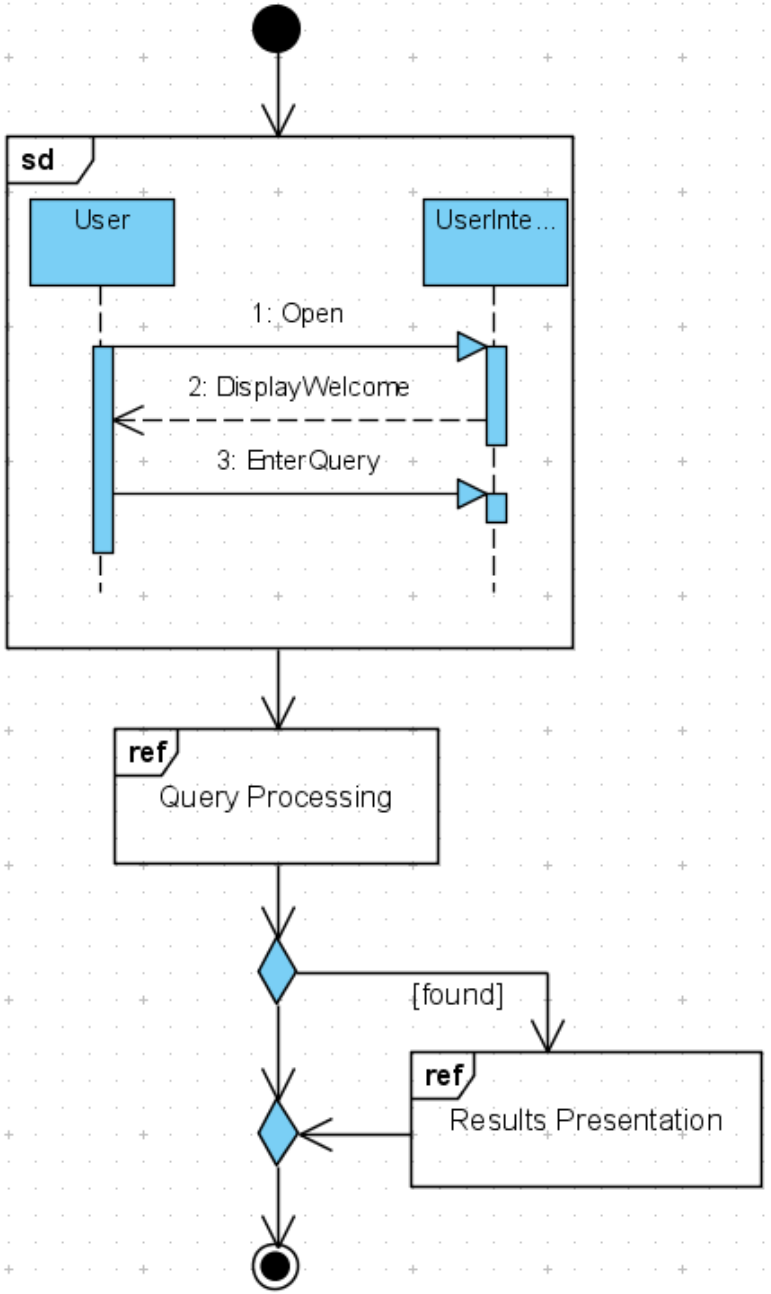
Interaction overview diagram

- Allows to tie a number of sequence, communication and timing diagrams using a notation inherited from activity diagrams
- Are suitable for large systems, where interactions and connections between a large number of mentioned diagram have to be shown

Interaction overview diagrams

- The sd, cd and td diagrams can be represented in two ways:
  - As a REF region pointing to another diagram
  - As a frame containing complete specification of a particular diagram
- Both representations can be mixed in one diagram
- Apart from this, elements from activity diagrams are present: control flows, initial and final nodes, decision, merge, for and join nodes

Sample IOD diagram



Component diagrams

- Allow to describe interactions between components (modules) of the system
- Component is a hermetic part of the software system that interacts with other components through interfaces and may be related to interfaces through dependency or realisation
- It is closely related to the concept of reusability
- Components are depicted by a symbol resembling class symbol, sometimes with two “pins” over the left edge. They may be stereotyped

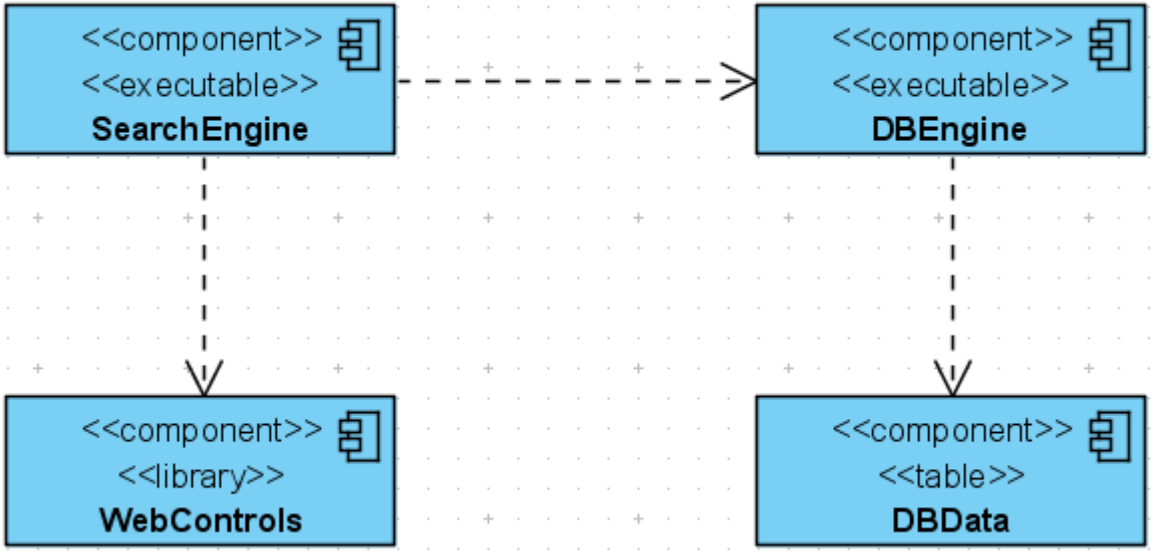
Component diagrams

- Typical components are:
  - Executables (including dynamic link libraries)
  - Libraries
  - Databases
  - Subsystems
  - Services
- Stereotypes are provided for these (and more) components

Dependency

- In the simple approach it is possible to describe connections between components using dependency relationship
- This is depicted by a dashed-line arrow pointing from the dependent component (i.e. the one that utilises some services provided by the other component)

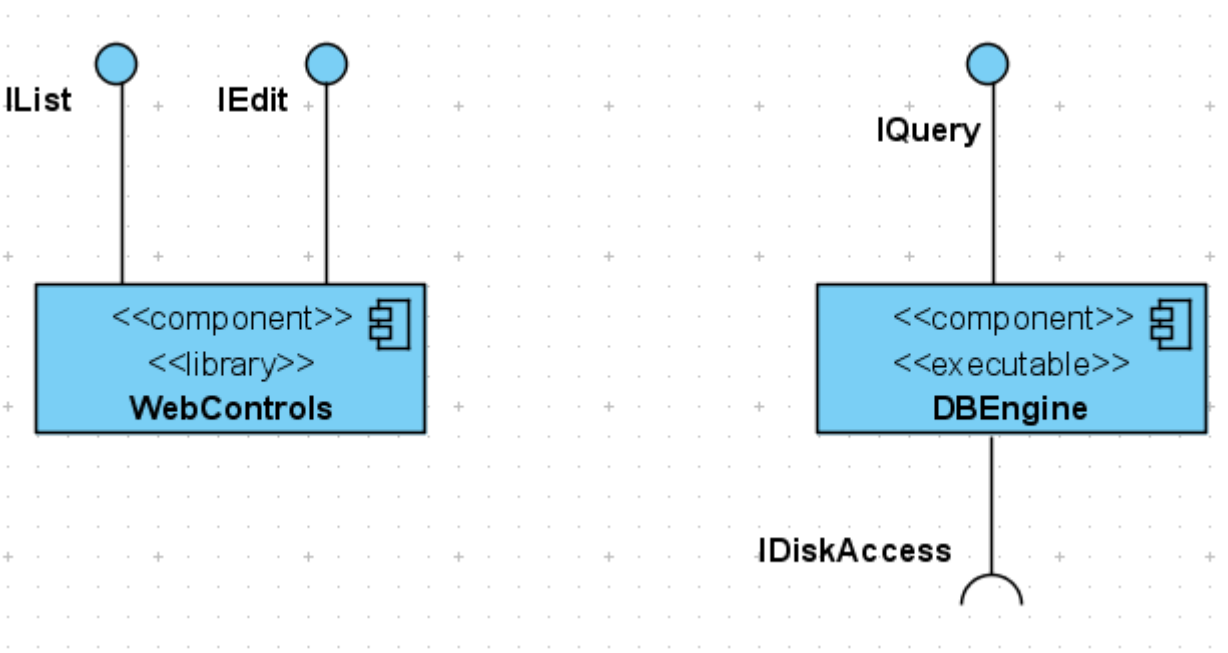
Sample component diagram



Interfaces

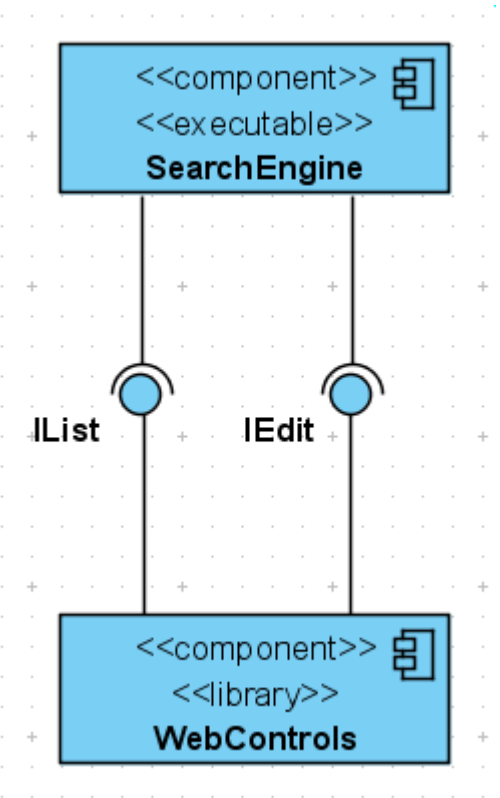
- More details about components may be provided by describing the interfaces
- Two distinct situations are possible:
  - Component realizes an interface. This means the component implements functionality required by the interface and can offer this functionality to other components. This is call provided interface and is depicted by a ball
  - Component depends on interface. This means that components needs service of another component, that implements the interface. This is called required interface and is depicted by a socket

Sample interfaces



Interfaces

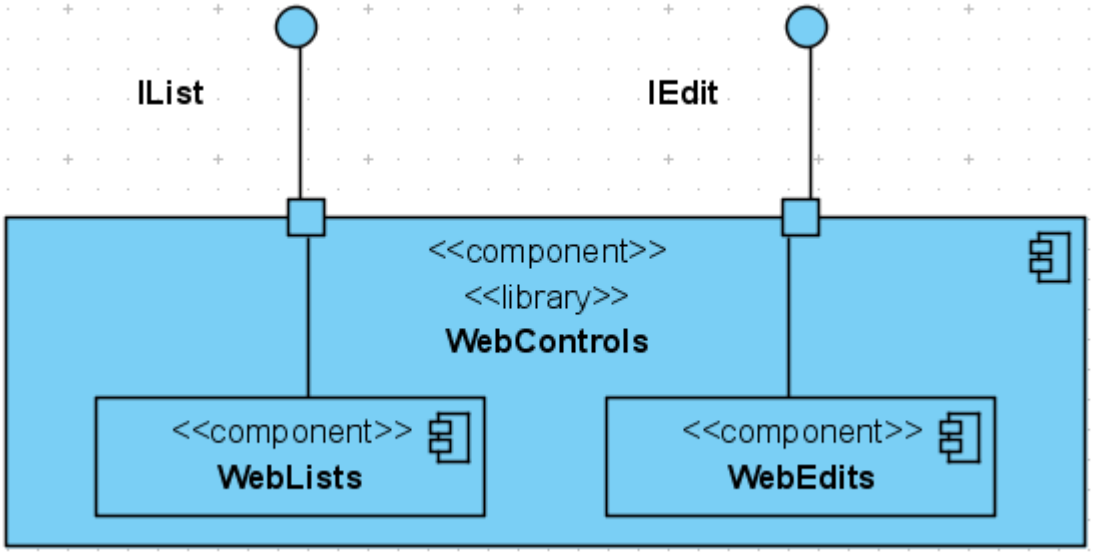
- The ball-socket connections can be used to precisely describe dependency



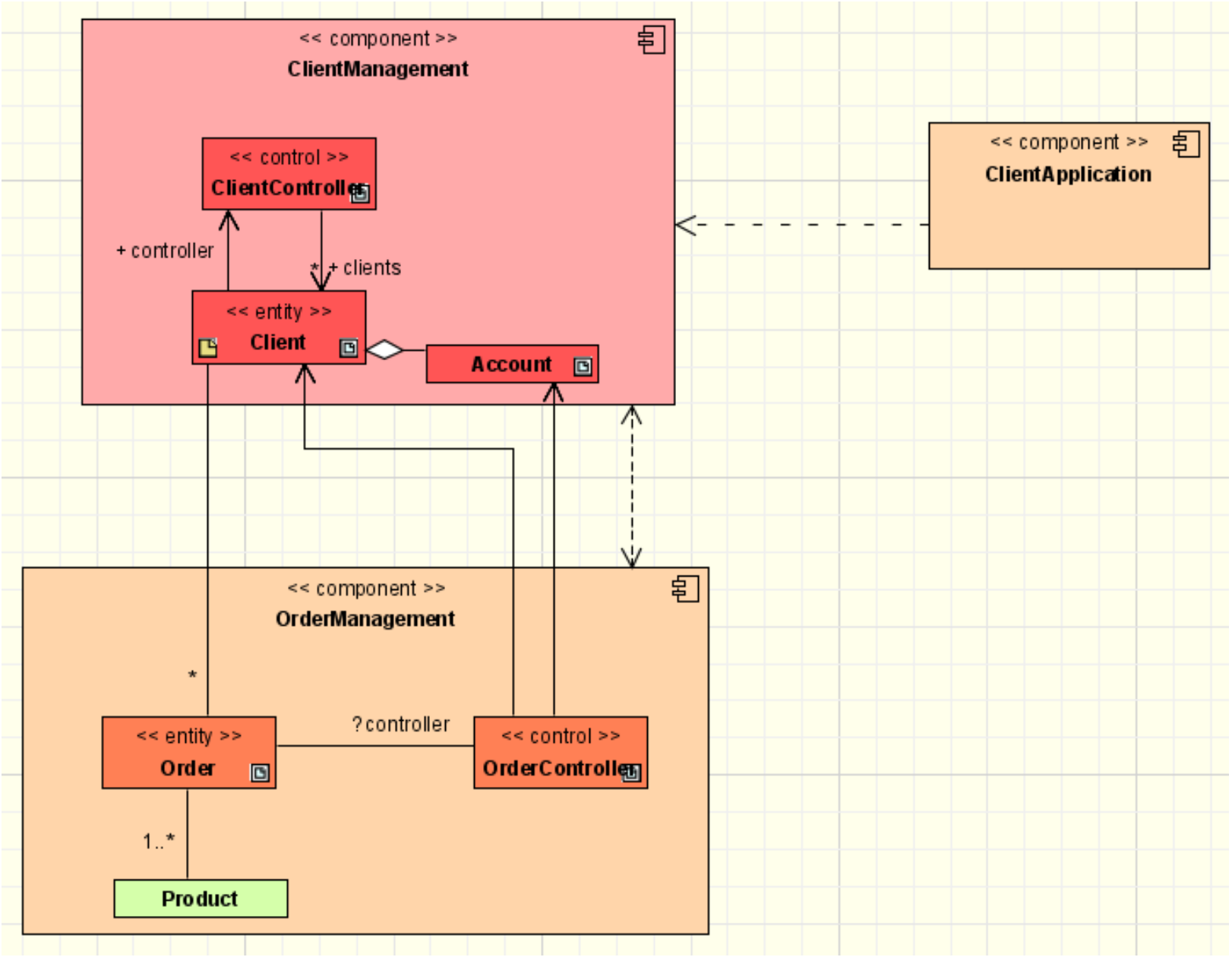
Internal structure

- It is also possible to provide more details about internal structure of the component by showing its sub-components
- In this case it may be useful to indicate which sub-components use interfaces of the parent component
- This may be done by using ports
- They are depicted as small rectangles on the border of the component

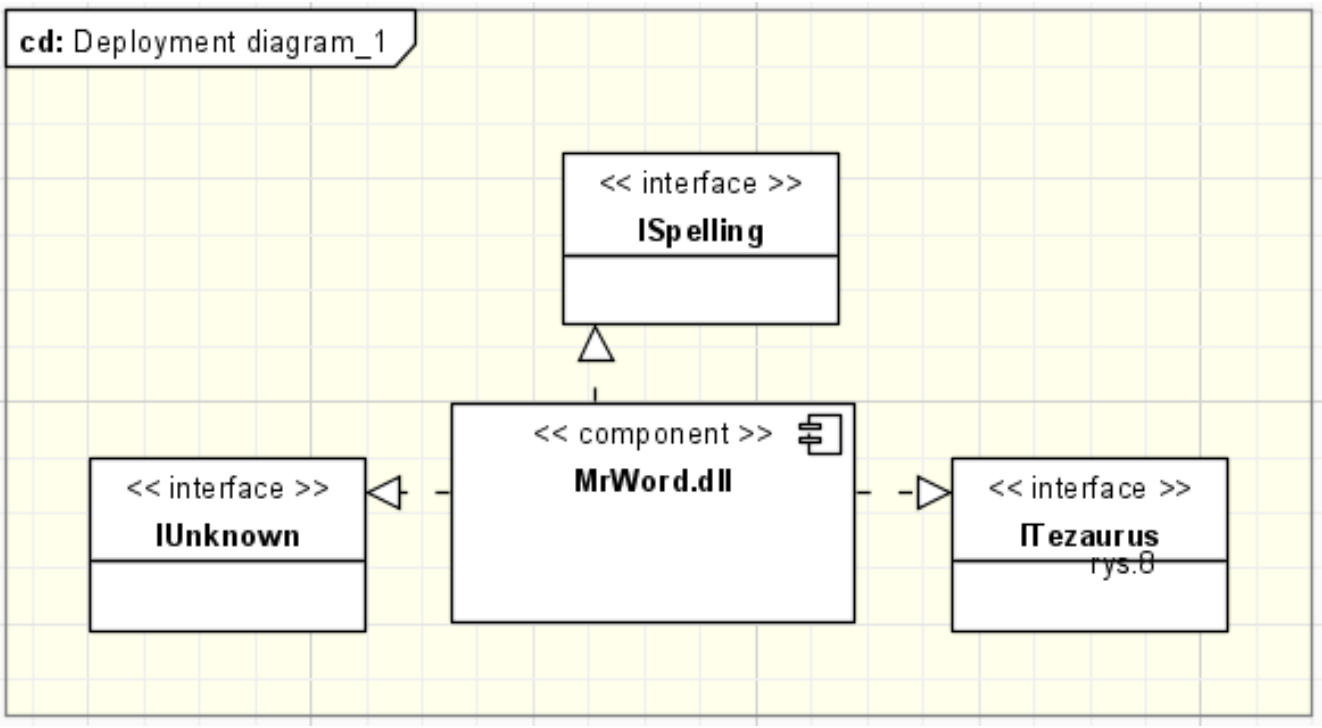
Sample component with ports



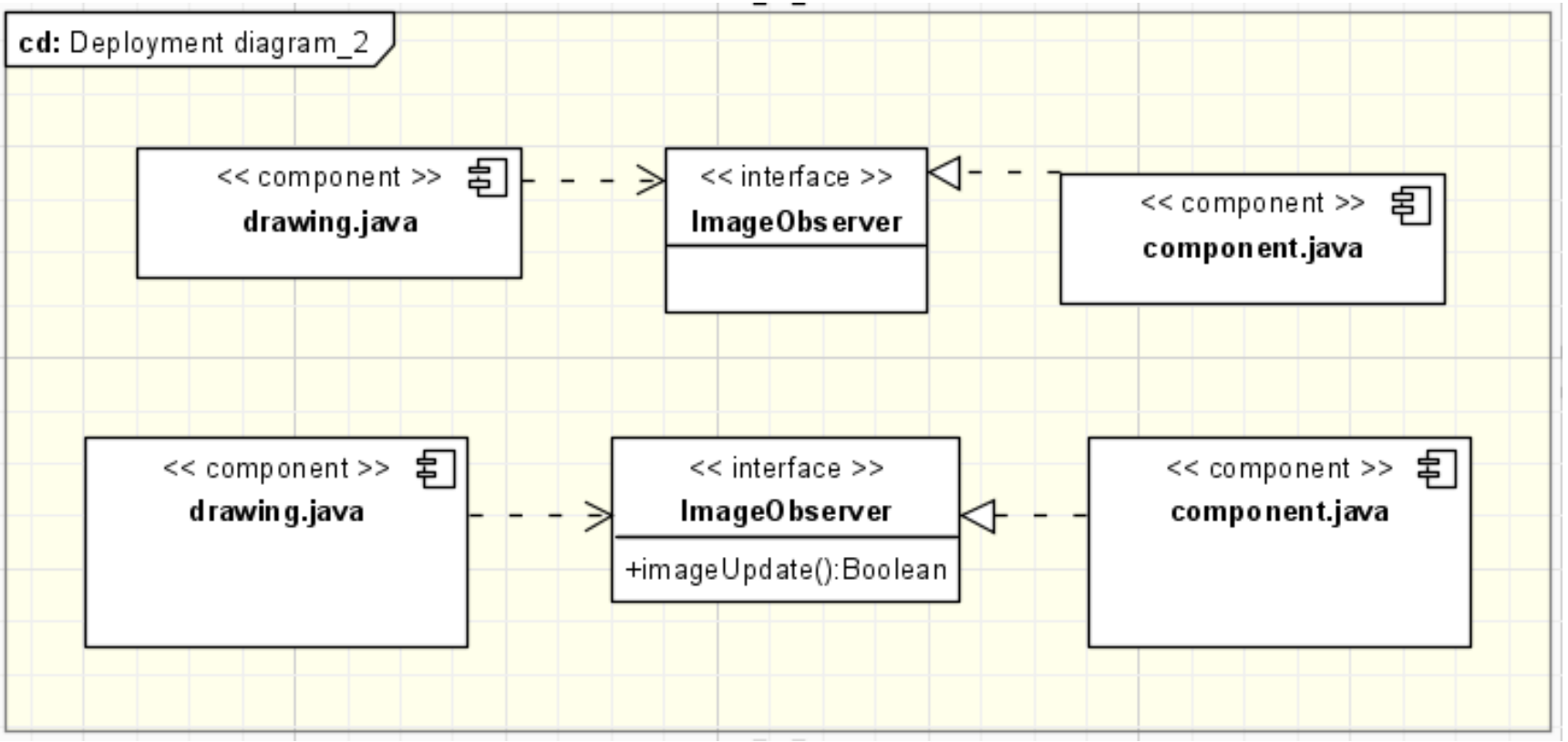
Sample diagram



Sample diagram



Sample diagram



Deployment diagrams

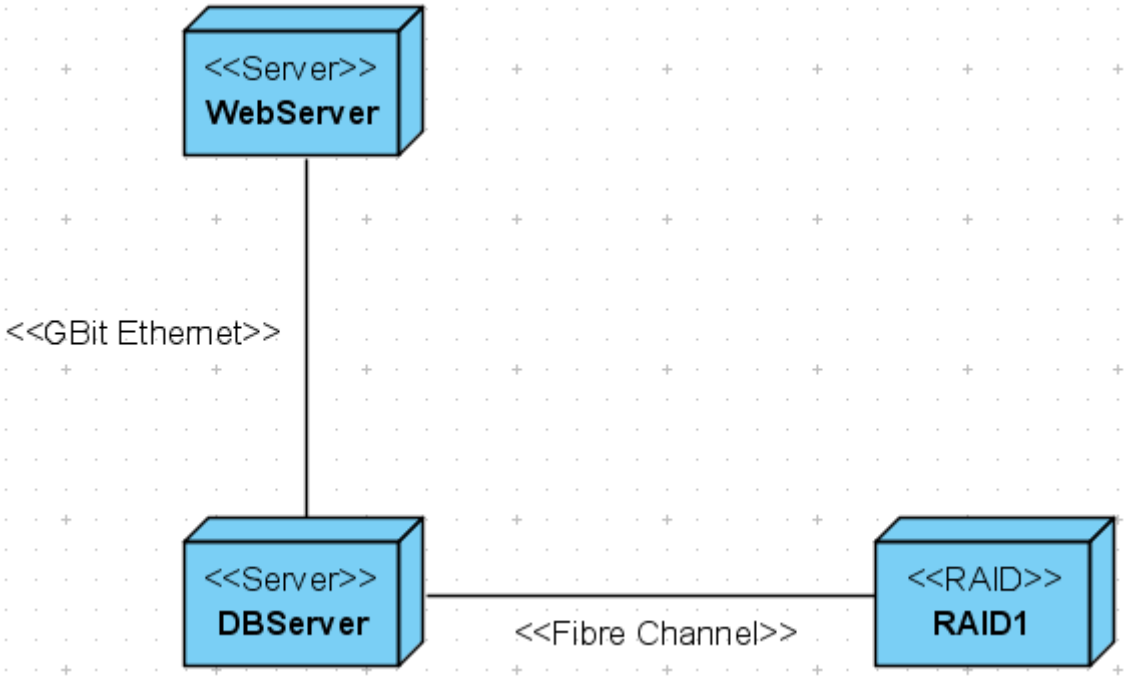
- Allow to describe the hardware components of the system and tie them to previously created elements of the model
- They use the concept of an artifact
- Artifact is any part of the software system. This can be a class, file, diagram, model, database, document and so on. Artifact is a physical item in the sense it is composed from a sequence of bits
- The artifacts are placed in nodes
- Nodes can be connected

Nodes

- Nodes represent physical or logical entities that provide means to “store” artifacts
- There are two different types of nodes:
  - Devices. These are physical appliances, such as computers, printers, switches etc.
  - Execution environments. These include software systems, such as operating systems, database systems etc.
- Node types are distinguished by stereotypes. Further details can be provided by more precise stereotypes (server, printer, Debian etc.)

Communication paths

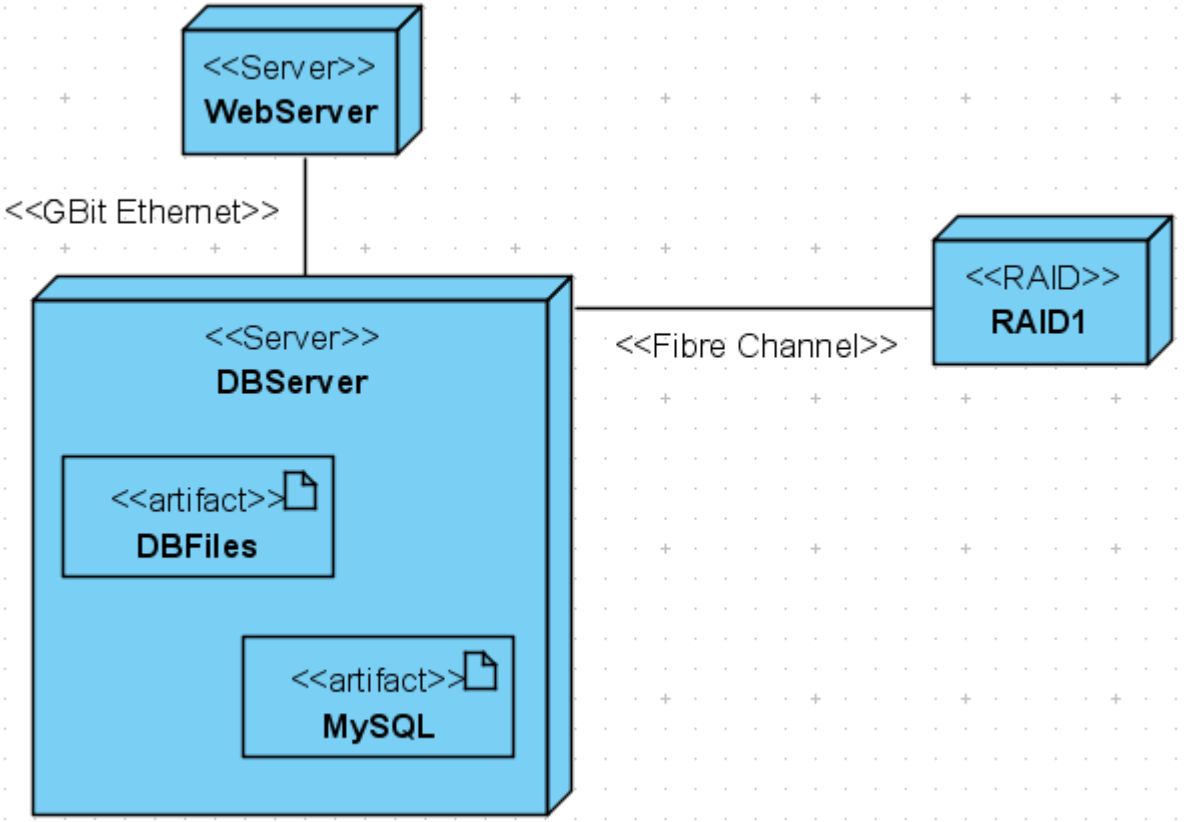
- Represent connections between nodes
- They often represent network connections
- Are depicted using associations



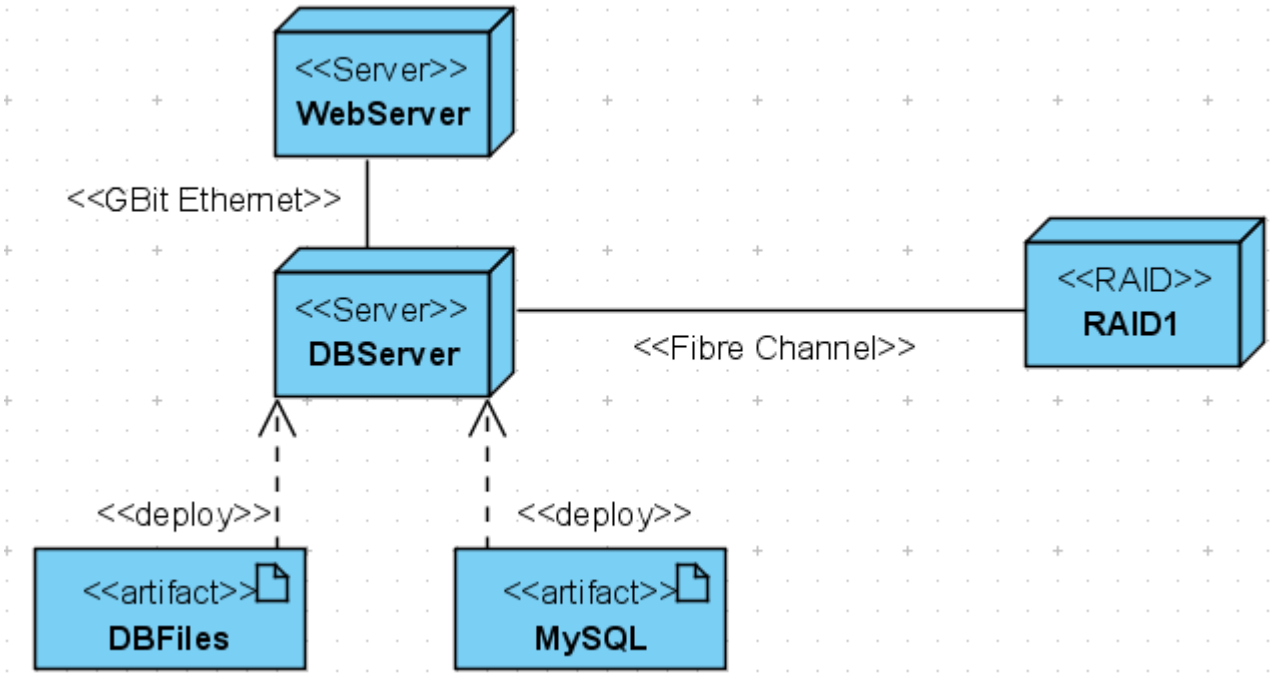
Artifacts in nodes

- Artifacts are embedded in appropriate nodes
- There are two possible ways of representing this fact:
  - By placing artifact inside of the node symbol
  - By connecting the node and artifact with dependency having <<deploy>> stereotype

Sample diagram



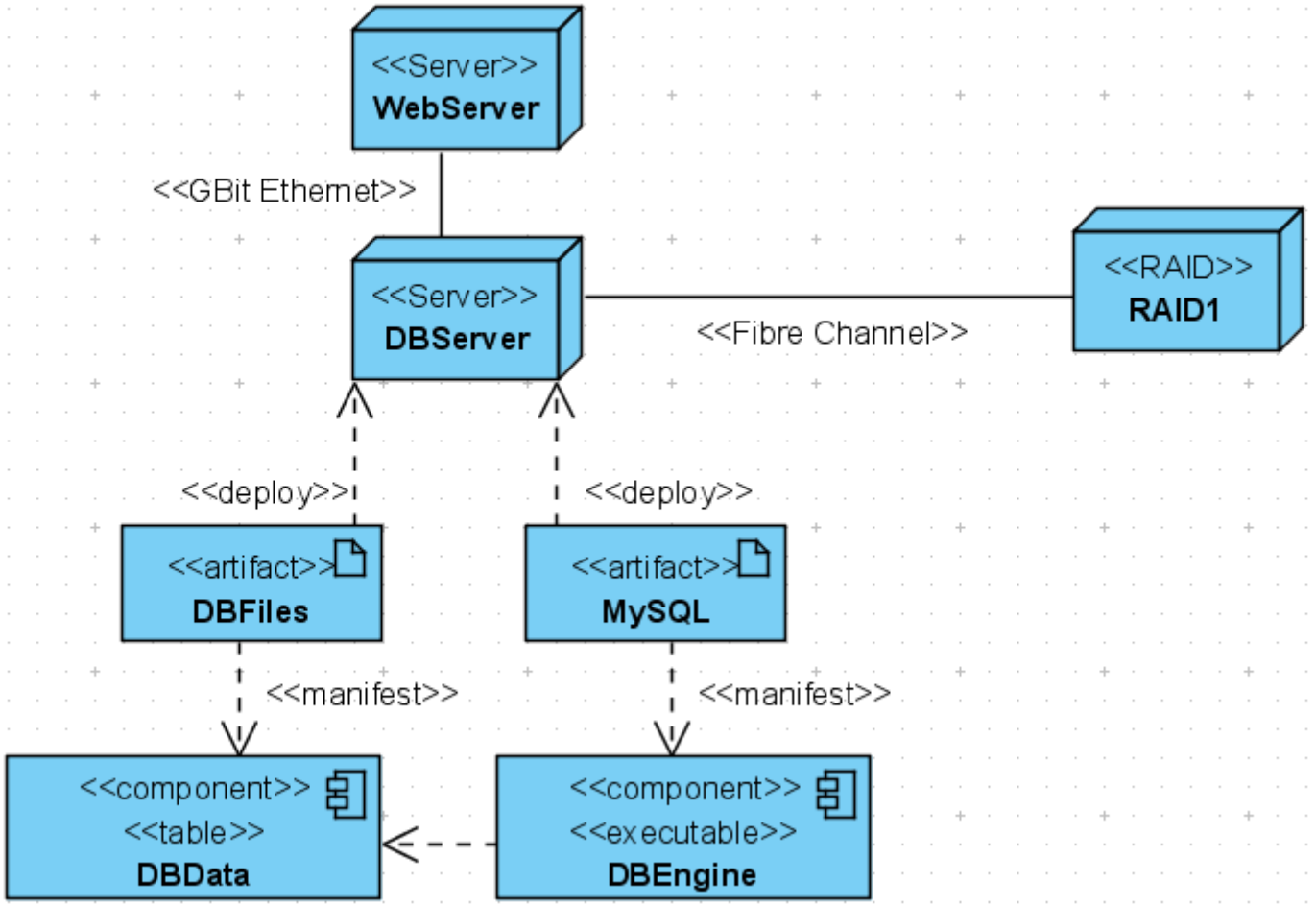
Sample diagram



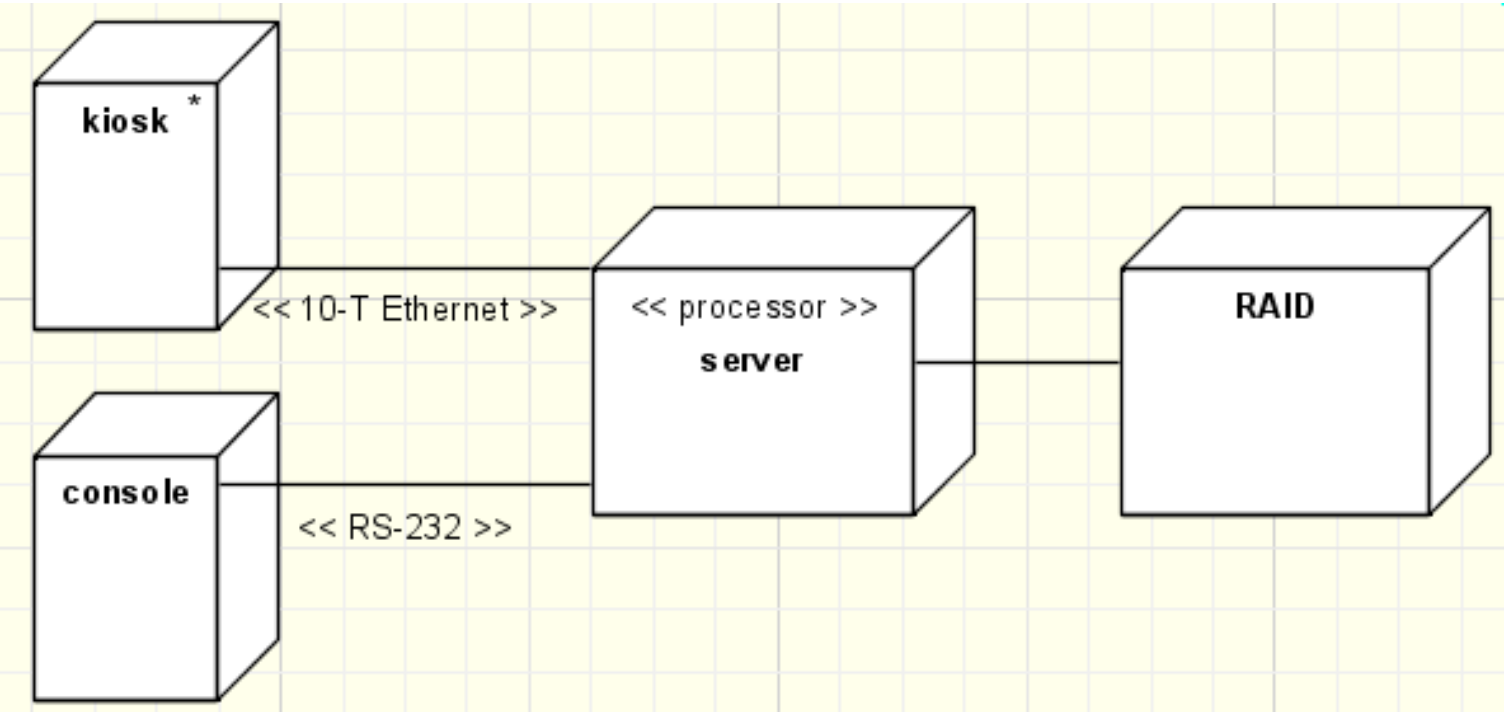
Manifestation

- The artifacts can manifest components, i.e. Indicate that they contain a particular component and therefore it is availabale at the node they are placed
- This is depicted using dependency with <<manifest>> stereotype

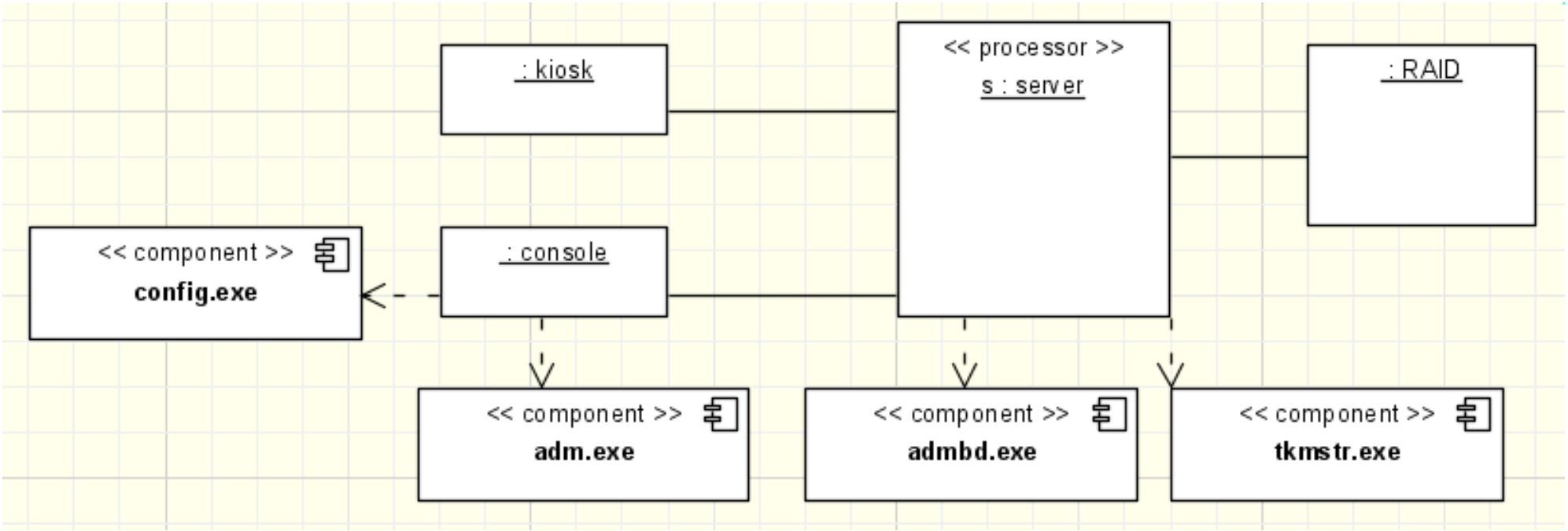
Sample manifestation



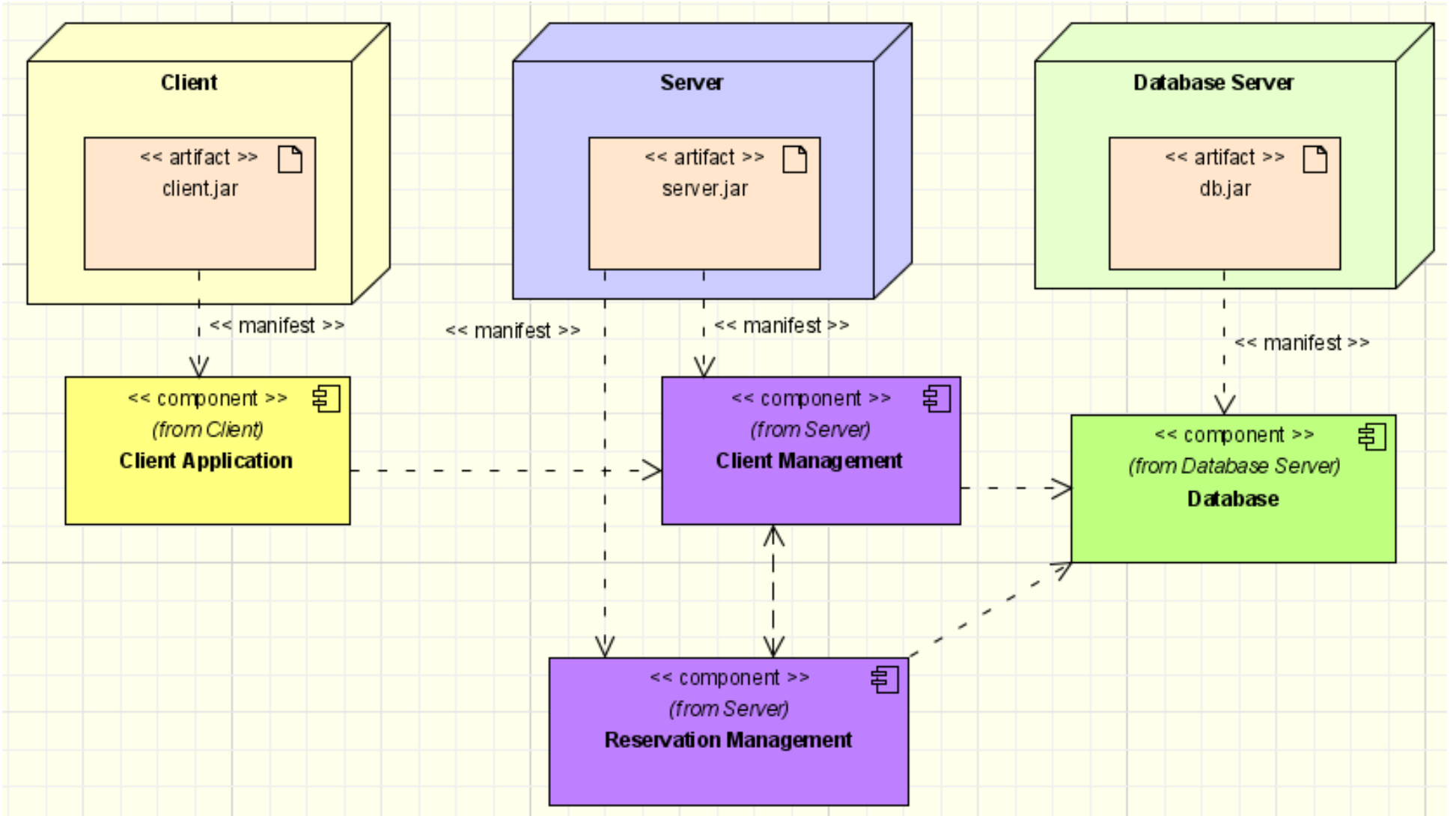
Sample diagram



Sample diagram



Sample diagrams



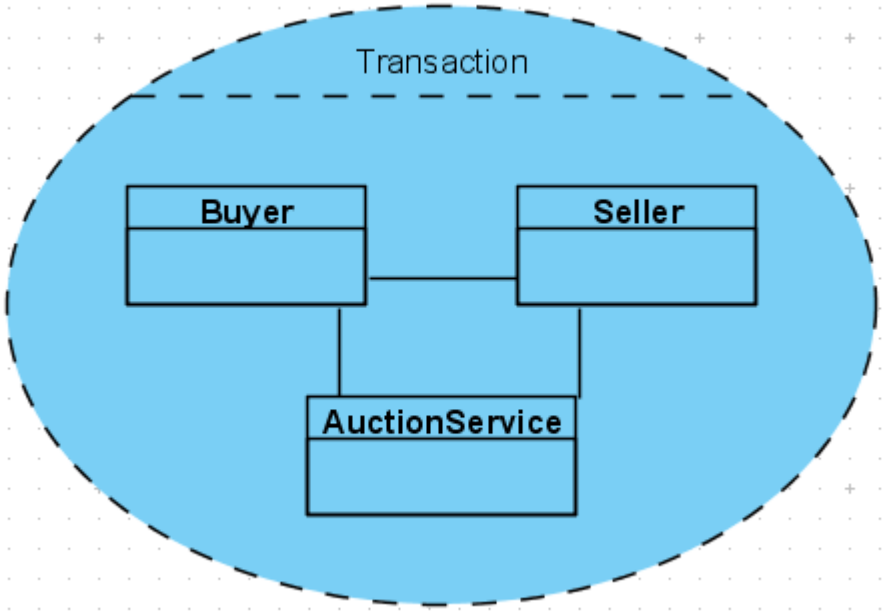
Composite structure diagram

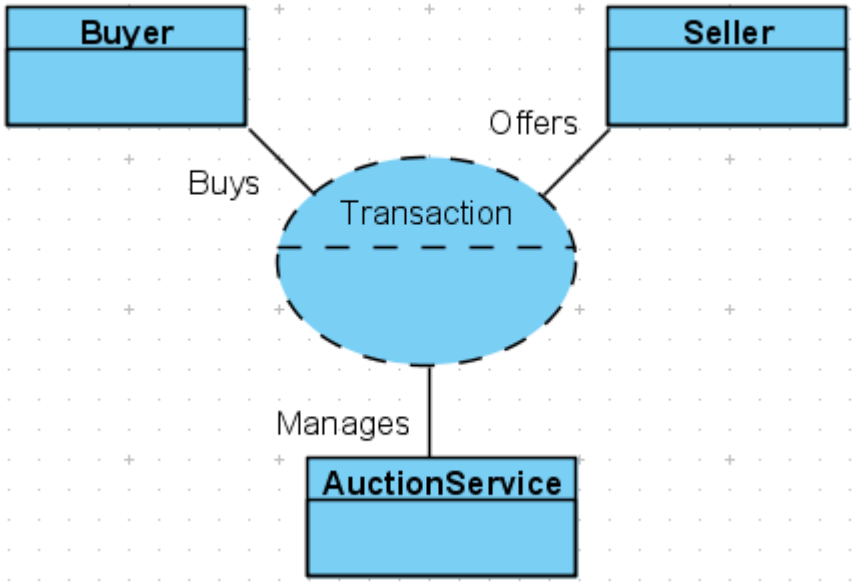
- Allows to present collaborations between classifiers
- The important element is collaboration symbol, an oval with dashed-line border
- The collaboration depicts the classifiers that “work” together for some reason
- The diagram should present only the classifiers that take part in the collaboration, and for these classifiers only the functionality (interfaces) that are relevant

Composite structure diagram

- The classifiers may be placed inside the collaboration oval
- Alternatively, the classifiers can be placed outside of the oval and connected with it using associations
- The associations should be named to describe the roles in the collaboration

Composite structure diagram





Package diagram

- Allows to arrange and group model elements
- Main elements are packages, dependencies and containment of packages
- Any model elements can be placed inside package
- The name of package is placed inside the package symbol, in the package symbol tab (if elements are placed in the package symbol) or in the frame tab (if frame is used to describe package)

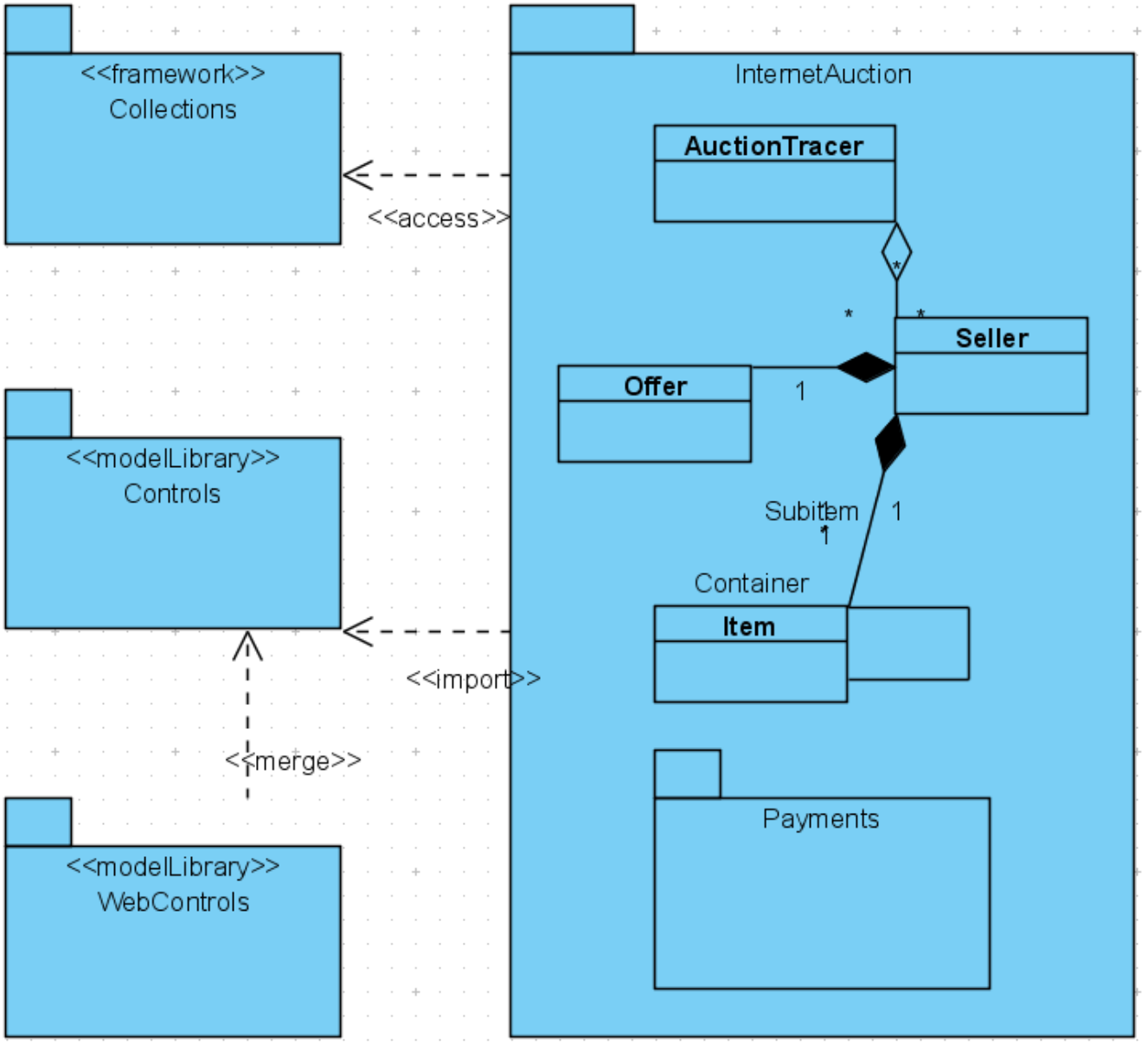
Package diagram

- The packages can be connected by dependencies
- There are three possibilities, denoted by stereotypes
  - Import. Means that elements of the source package can use elements of the target package by means of unqualified names
  - Merge. Generalisation between elements of the source package and elements with the same names of the target package
  - Access. Similar to import, but the names must be qualified

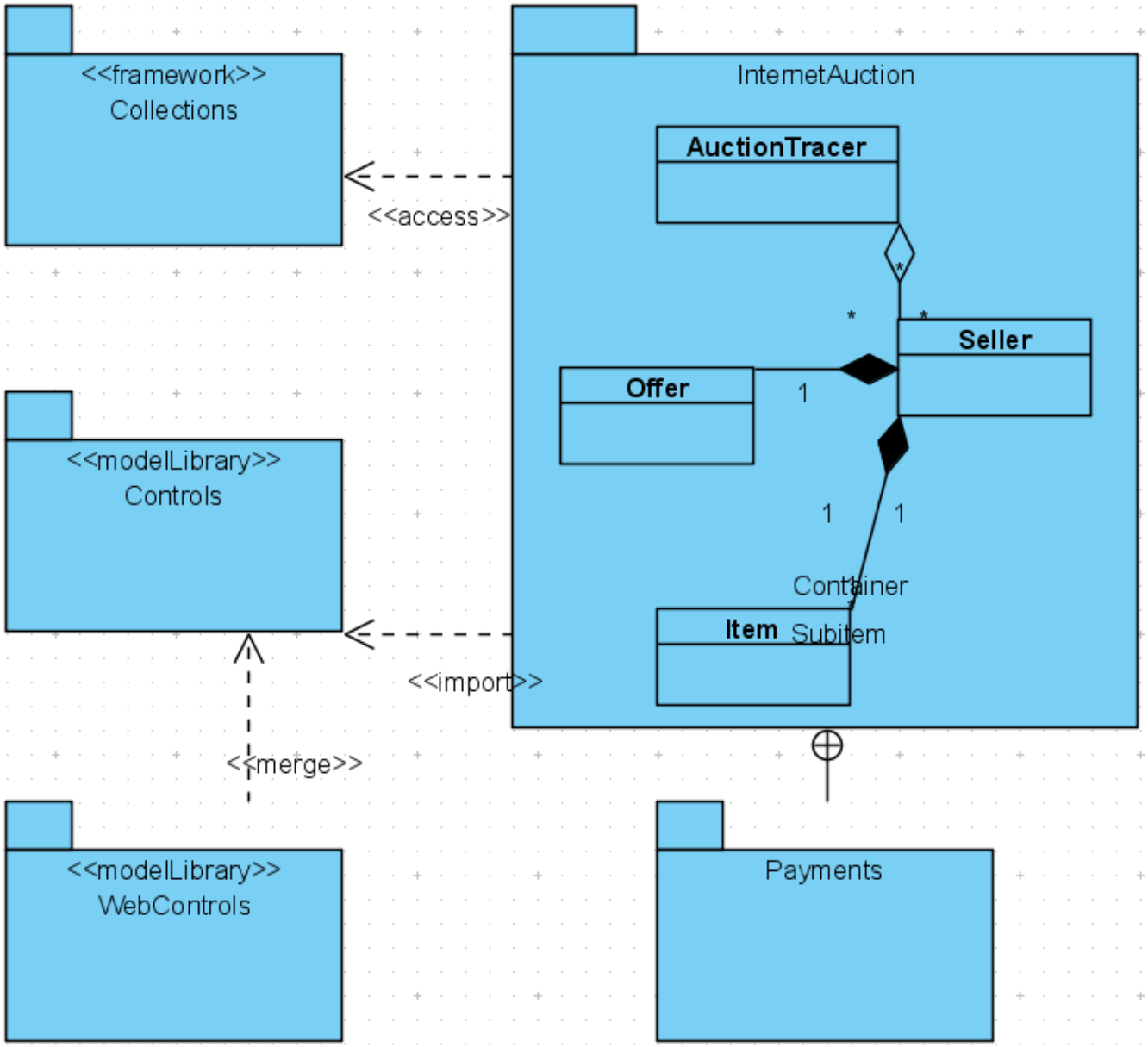
Package diagram

- Packages can contain other packages
- This can be depicted by placing packages inside other packages, or by containment relationship
- The packages can be stereotyped
- Typical stereotypes include <<model>>, <<subsystem>>, <<framework>>
- It is possible to specify visibility of package elements – as with calsses

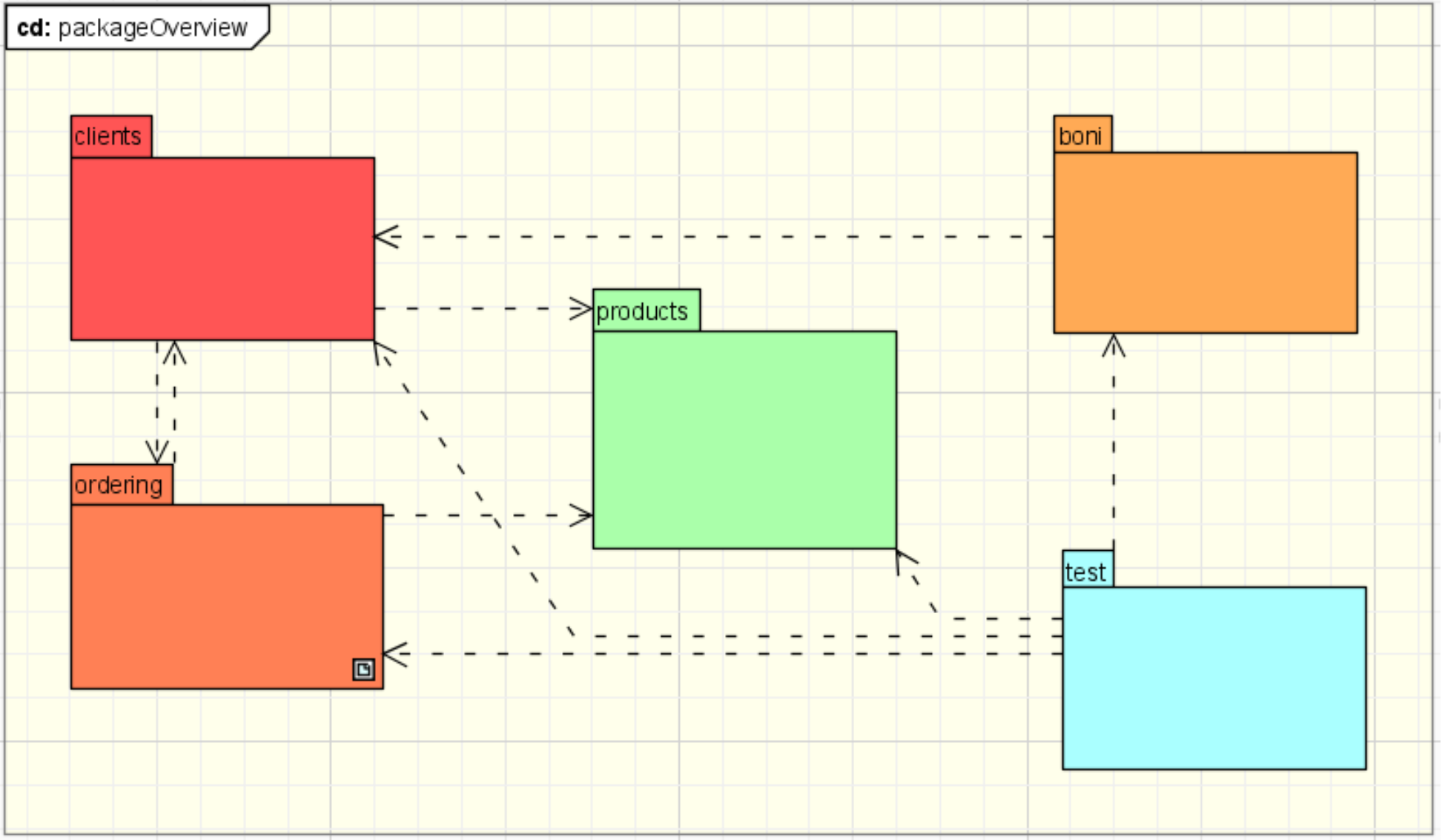
Package diagram



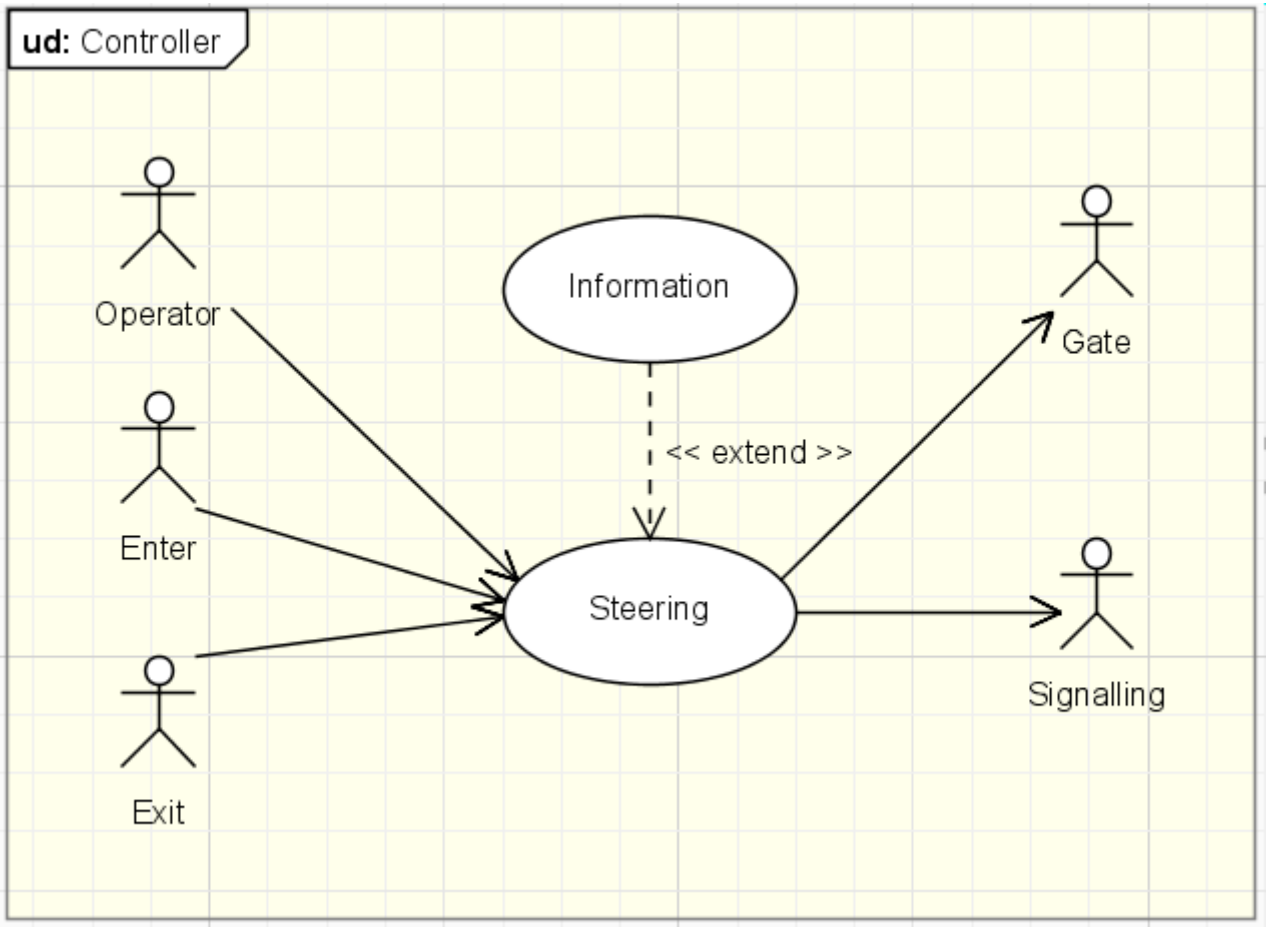
Package diagram



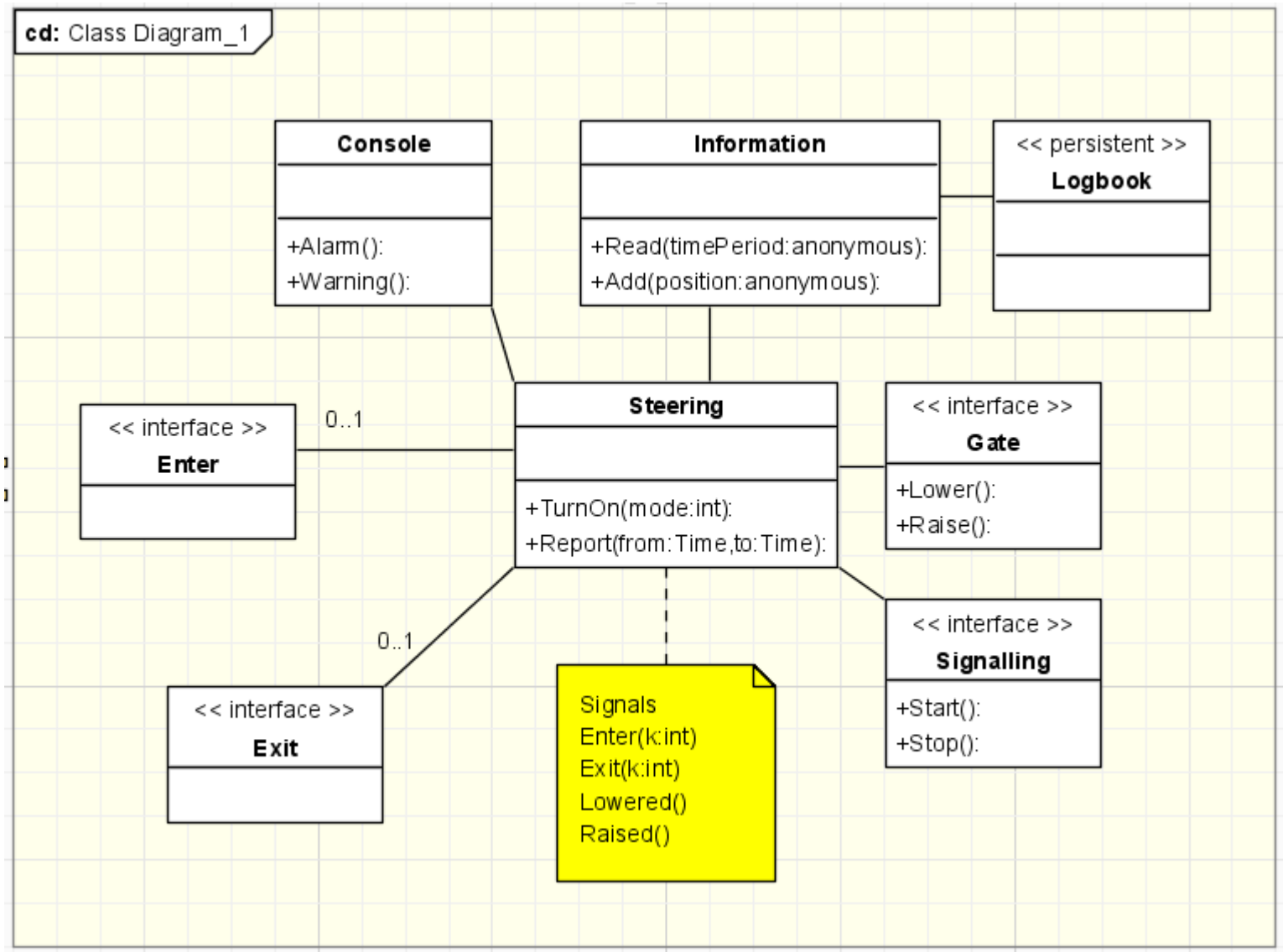
Sample package diagram



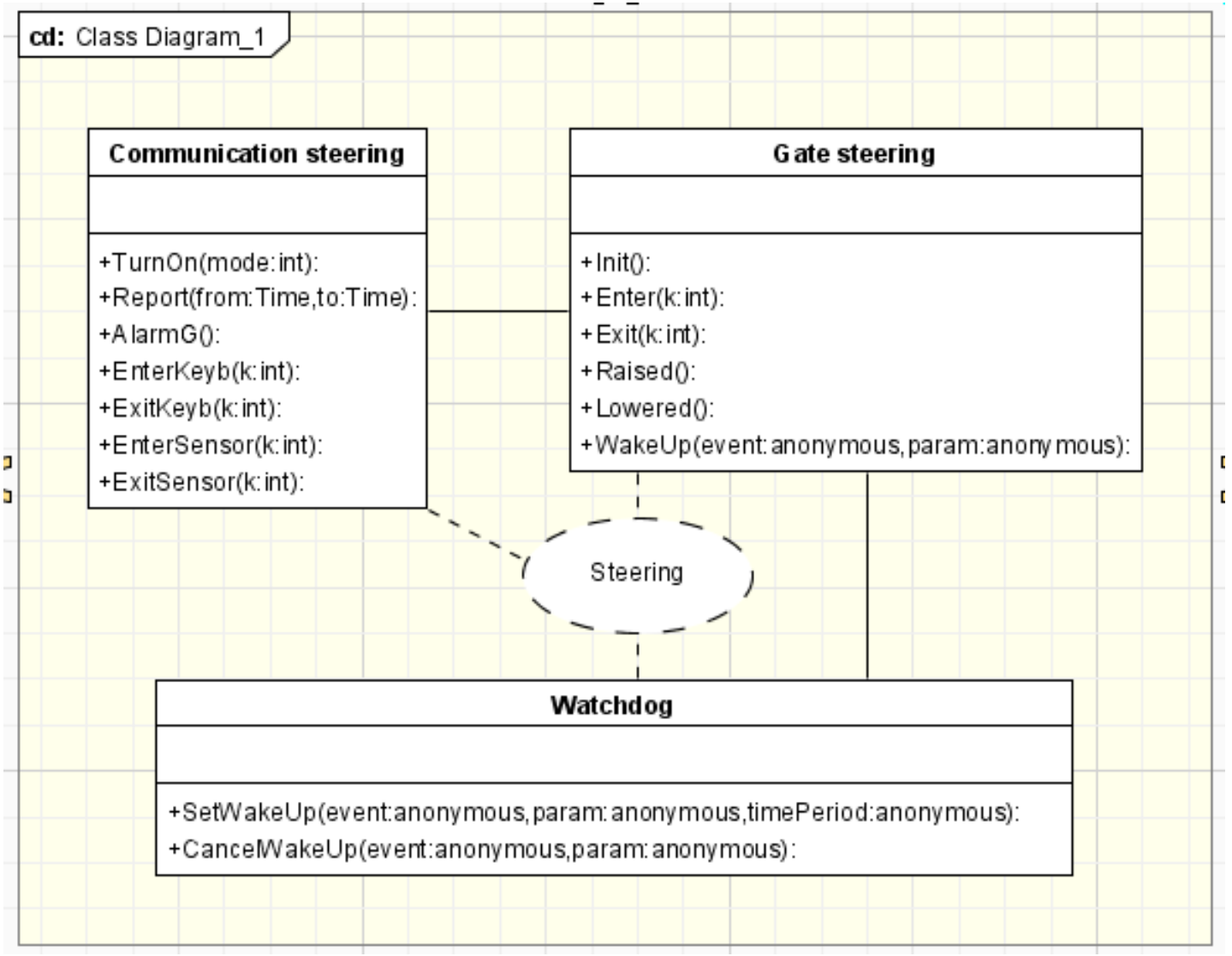
Railway crossing case



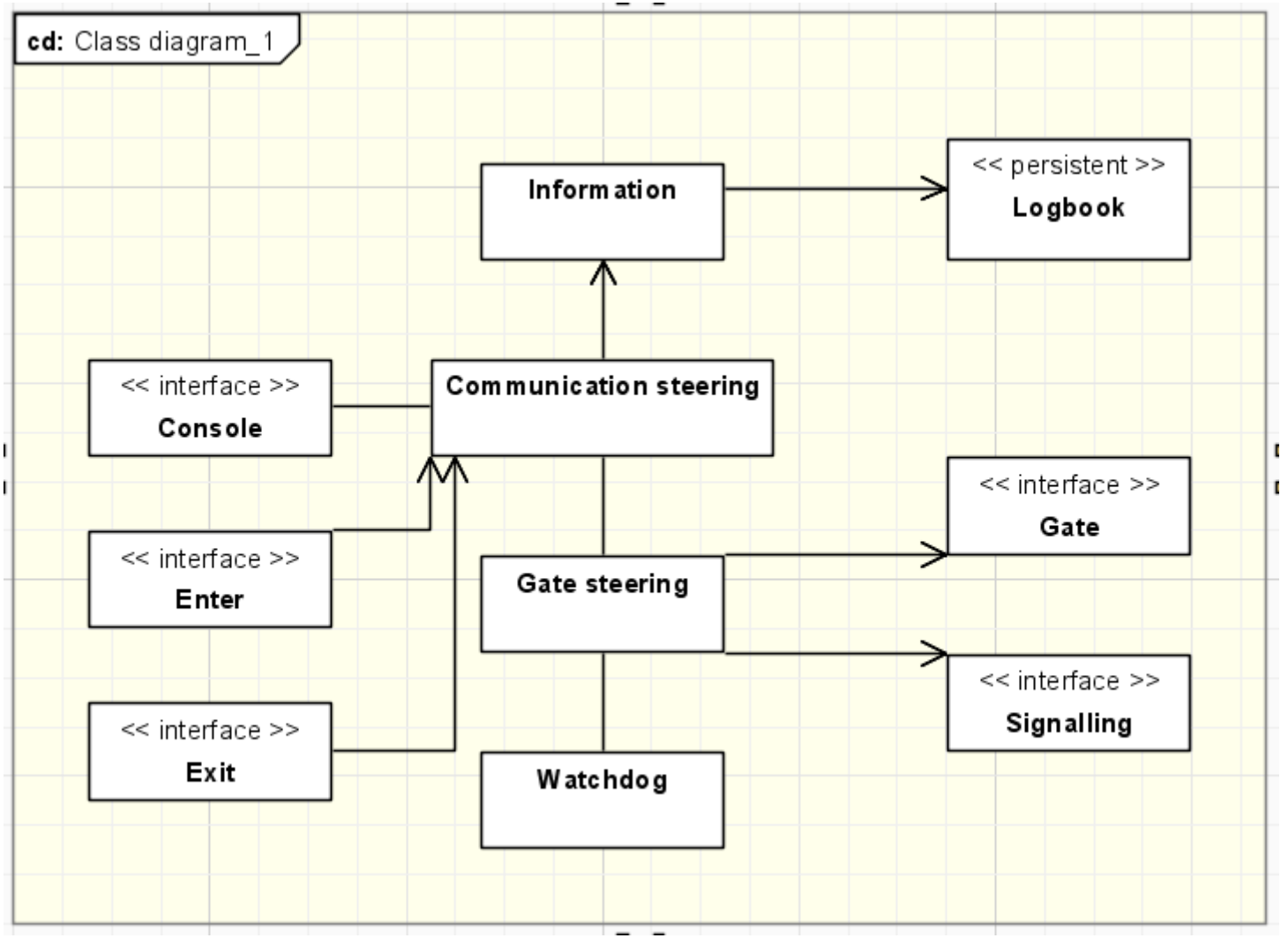
Railway crossing case



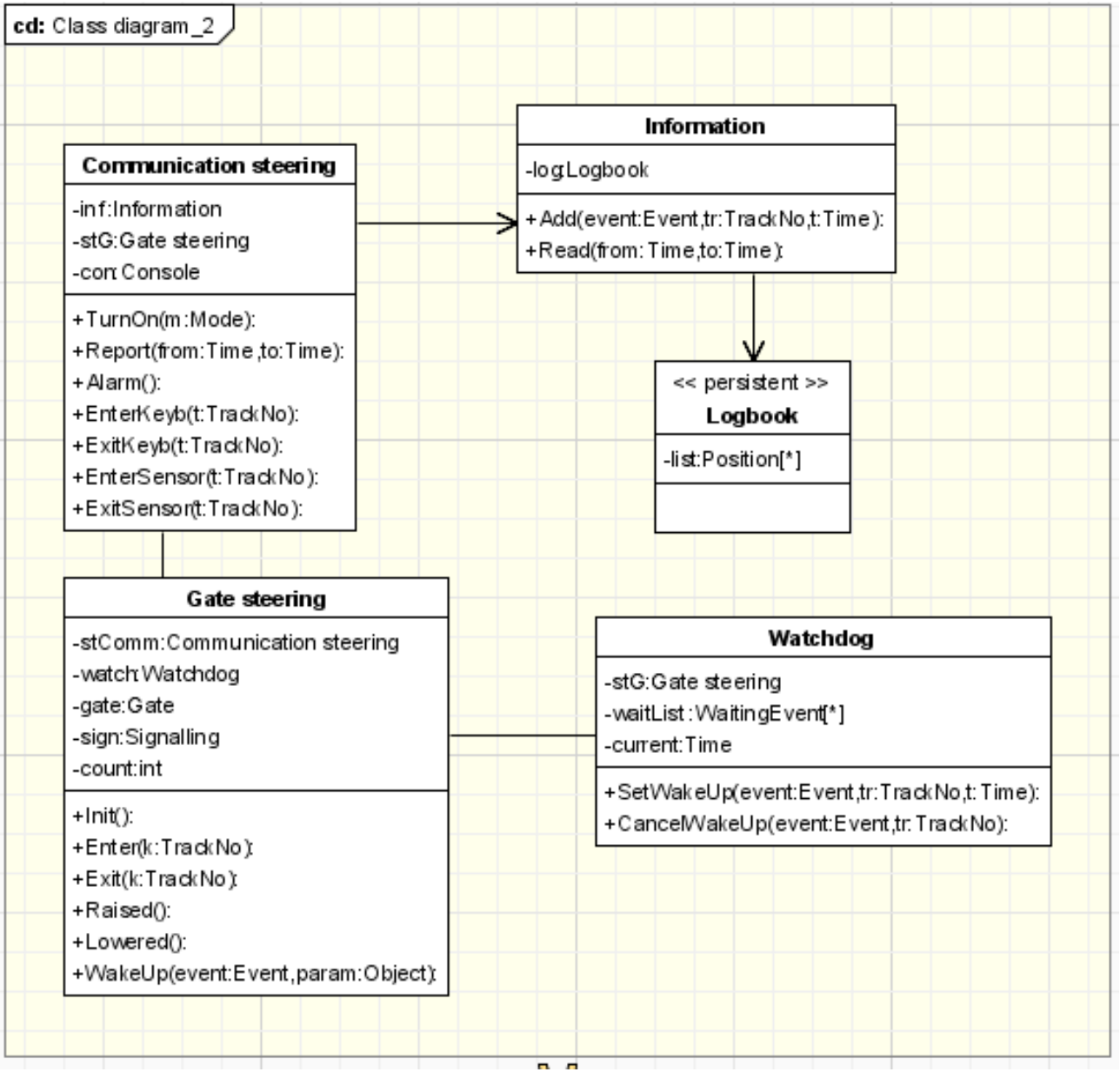
Railway crossing case



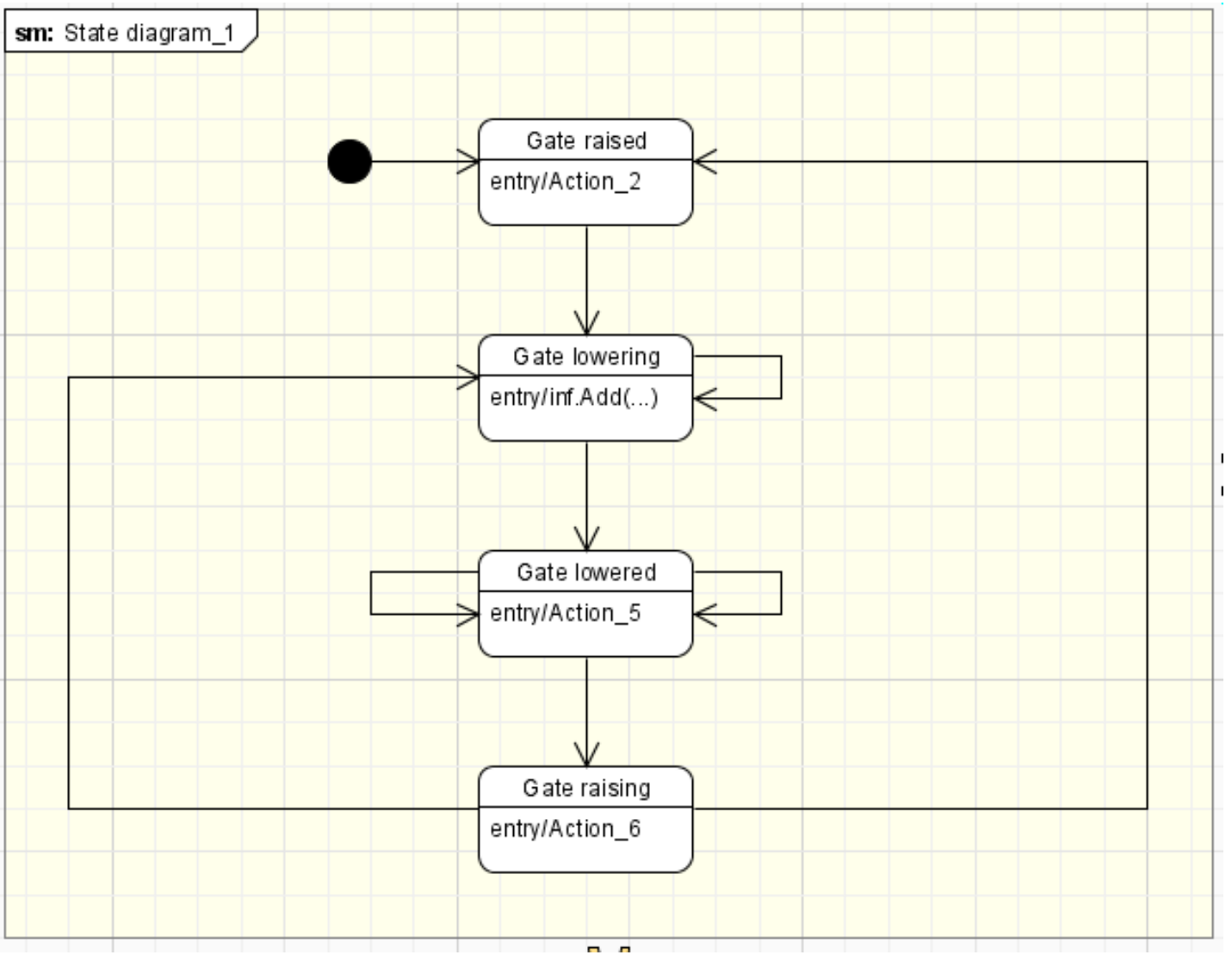
Railway crossing case



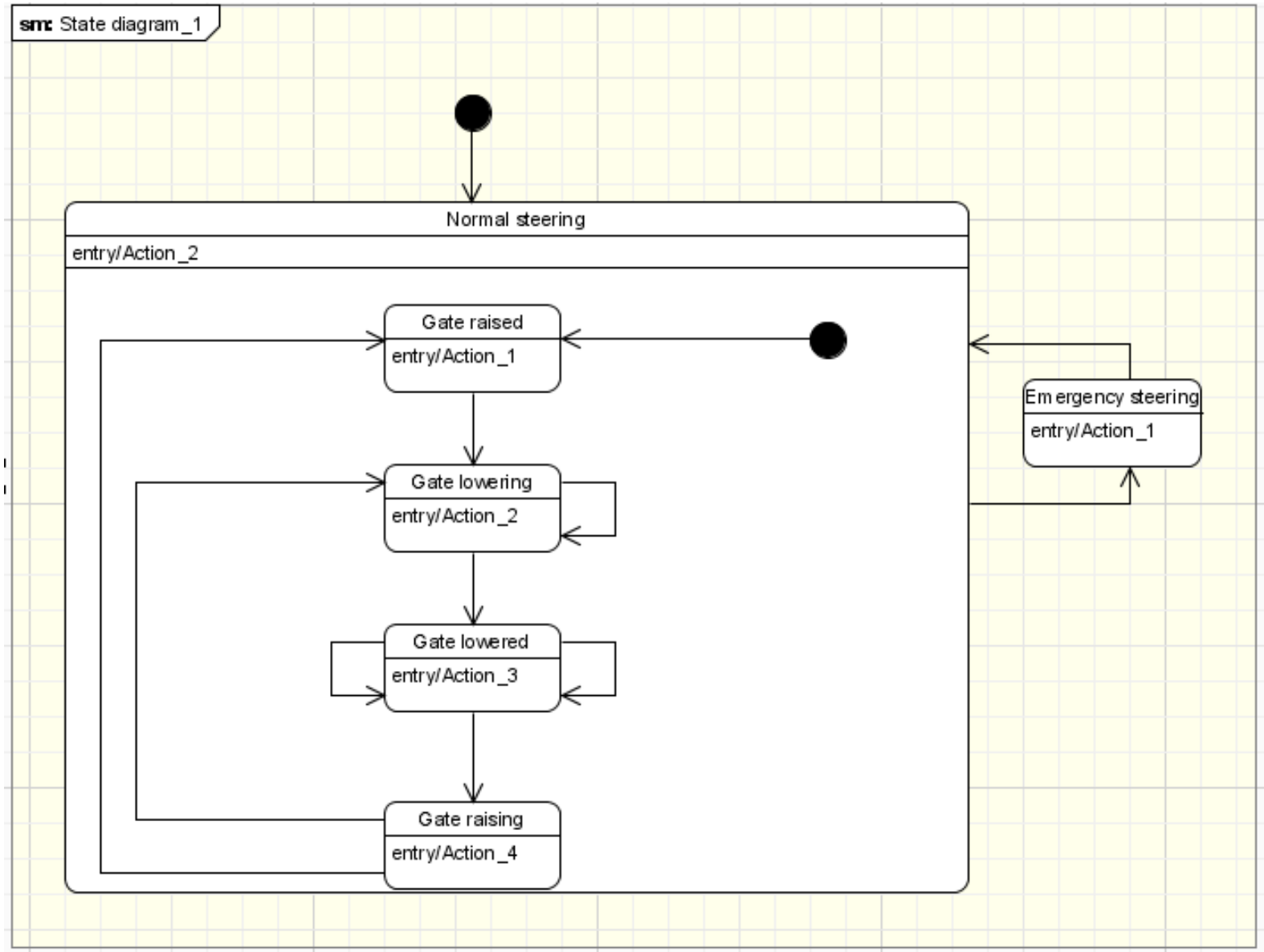
Railway crossing case



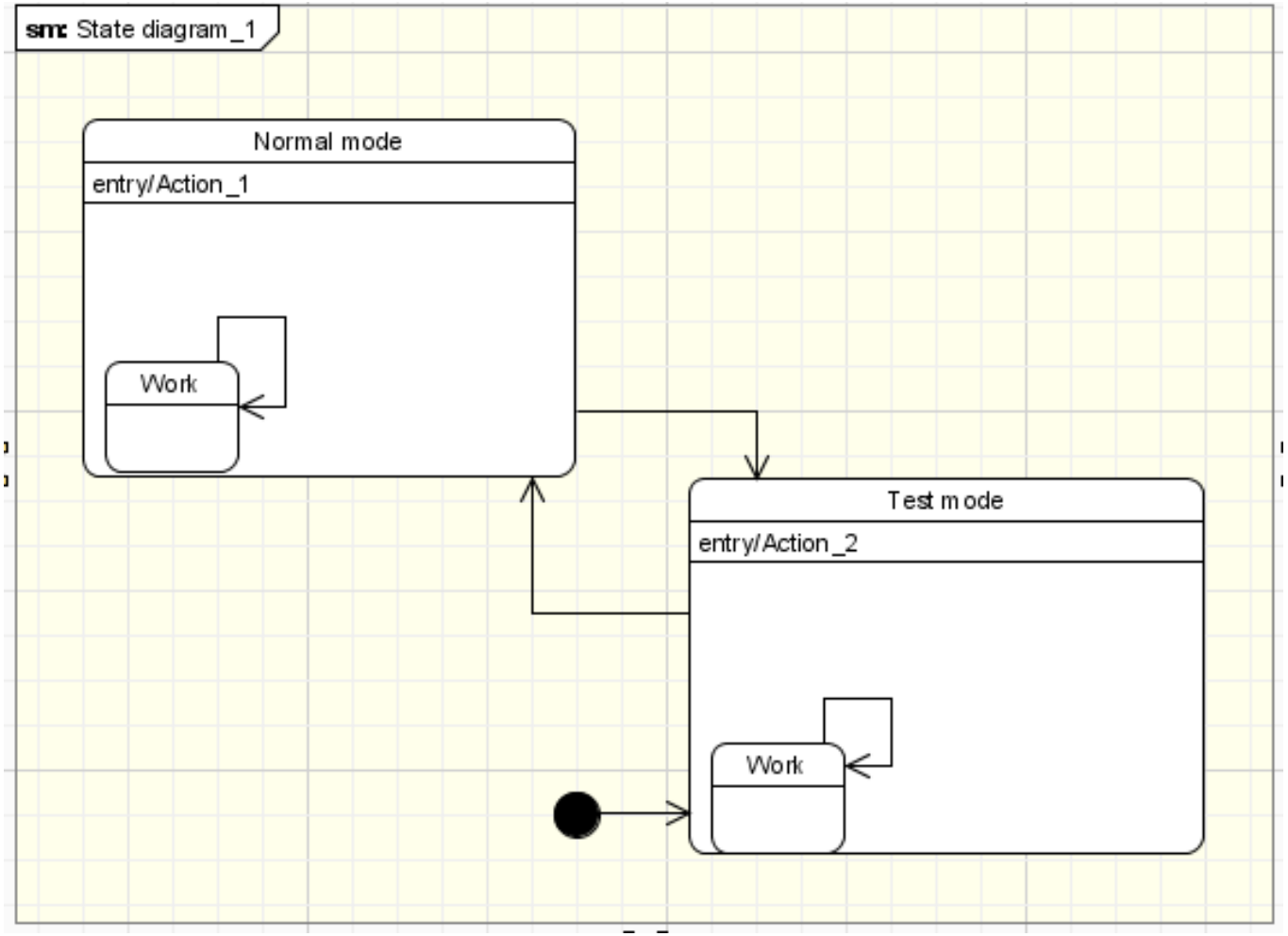
Railway crossing case



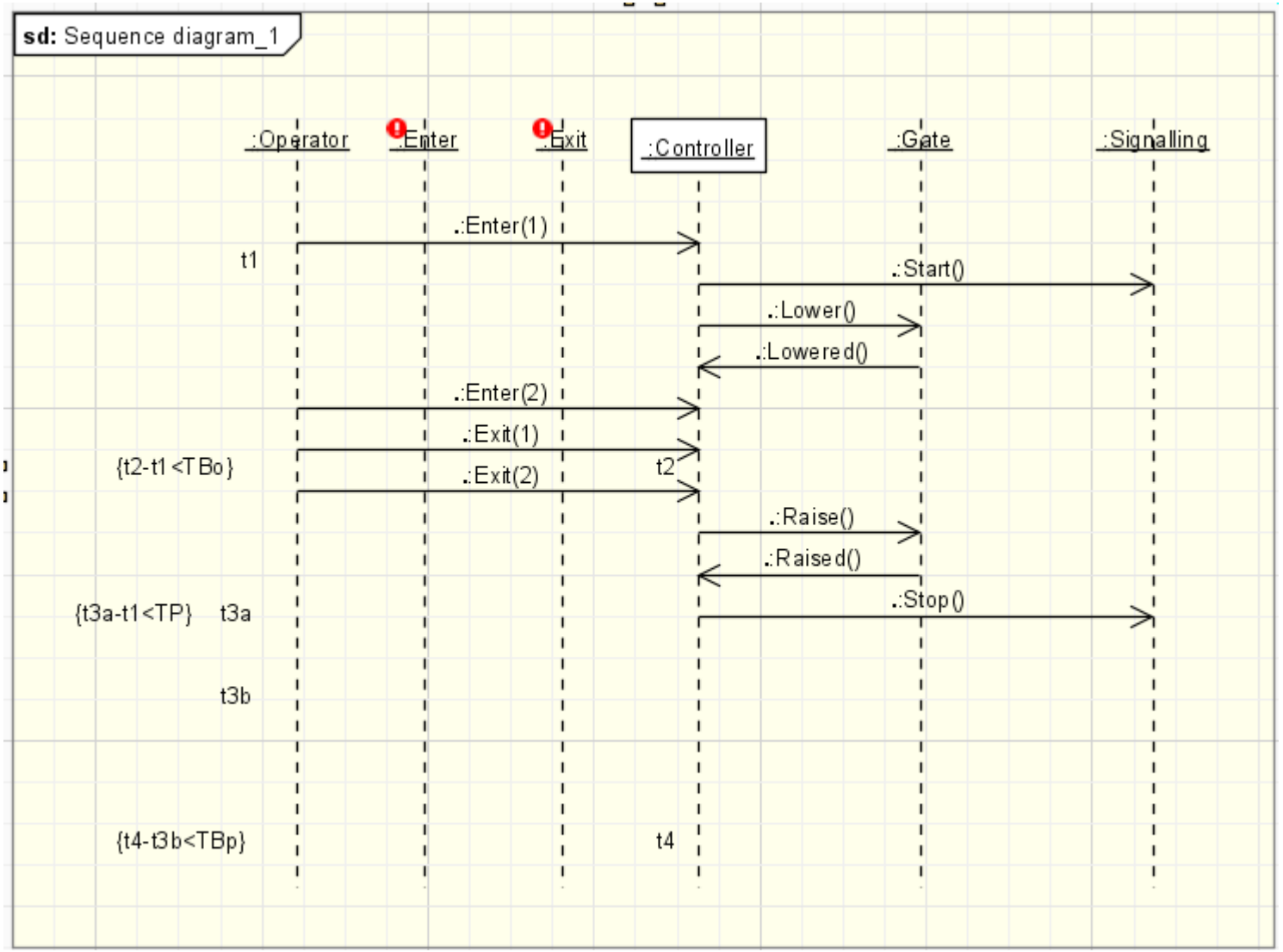
Railway crossing case



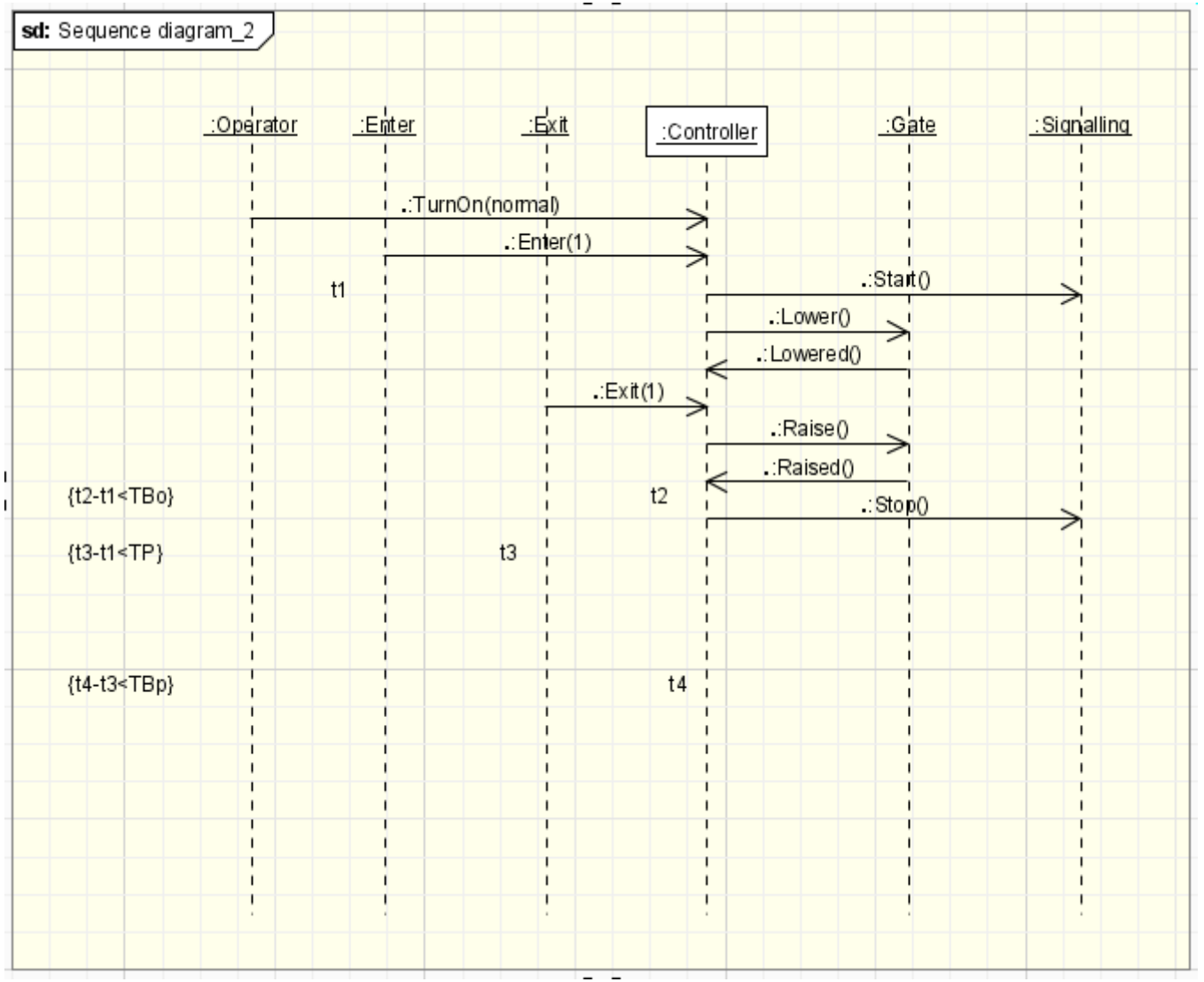
Railway crossing case



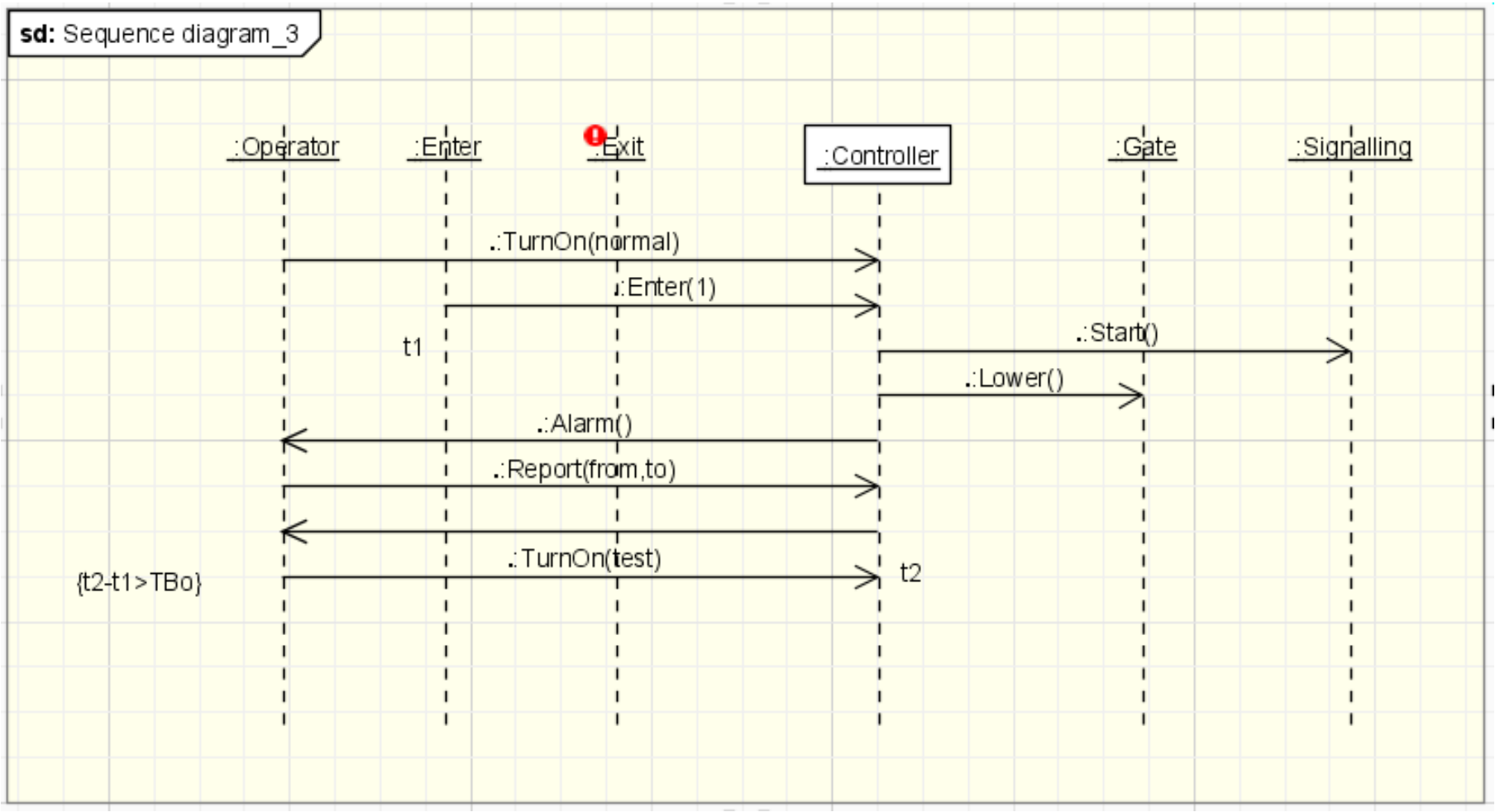
Railway crossing case



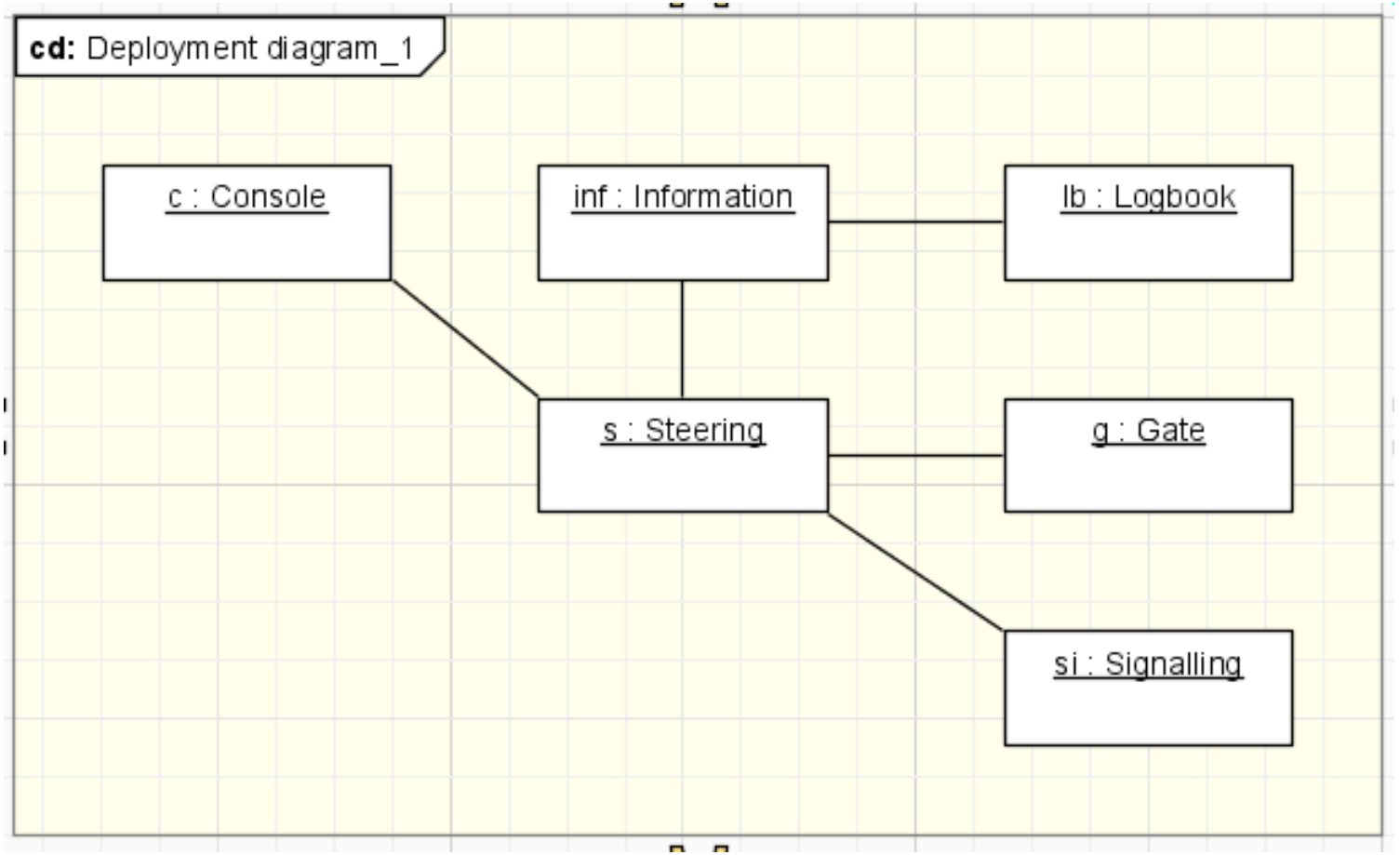
Railway crossing case



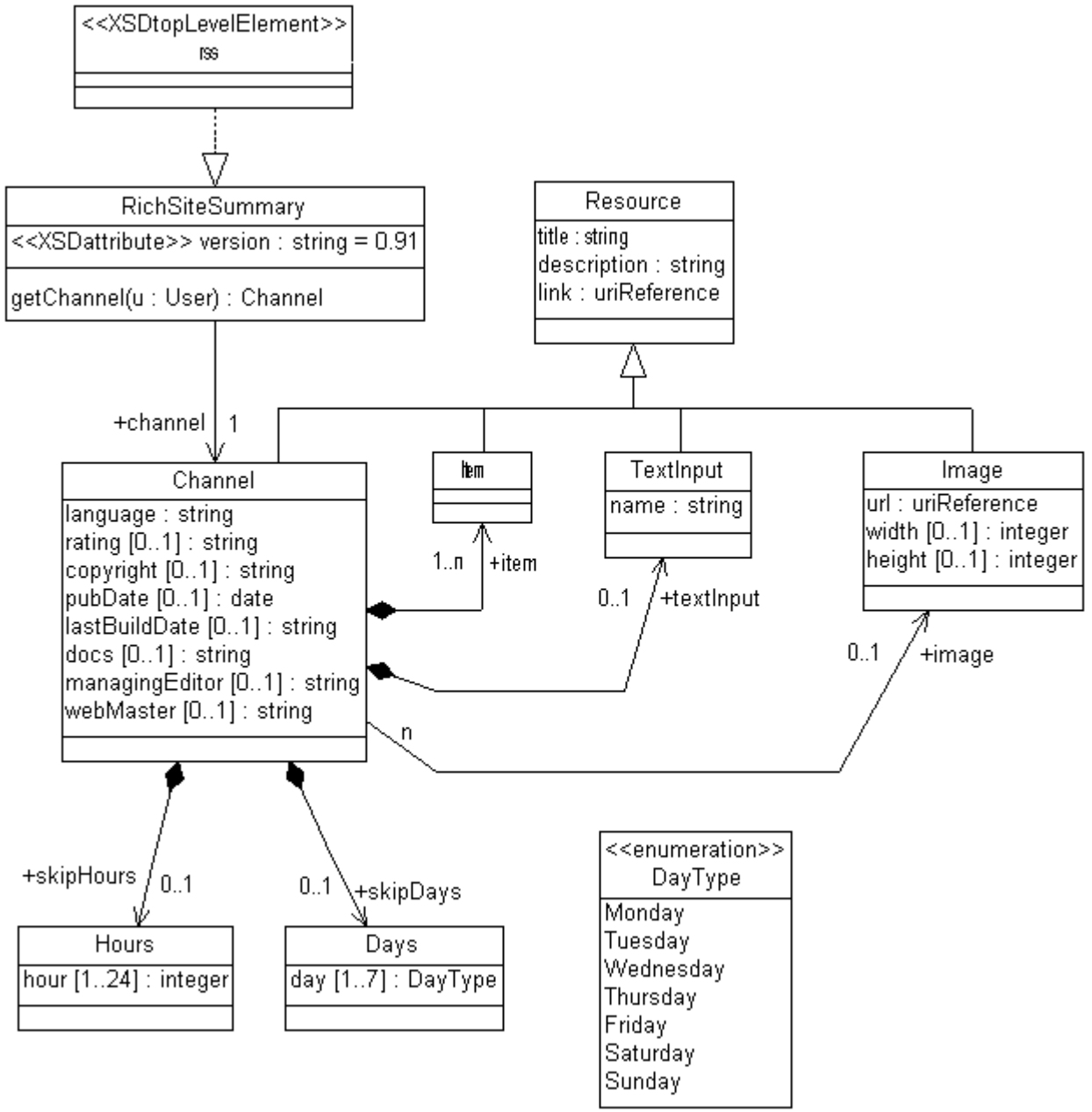
Railway crossing case



Railway crossing case

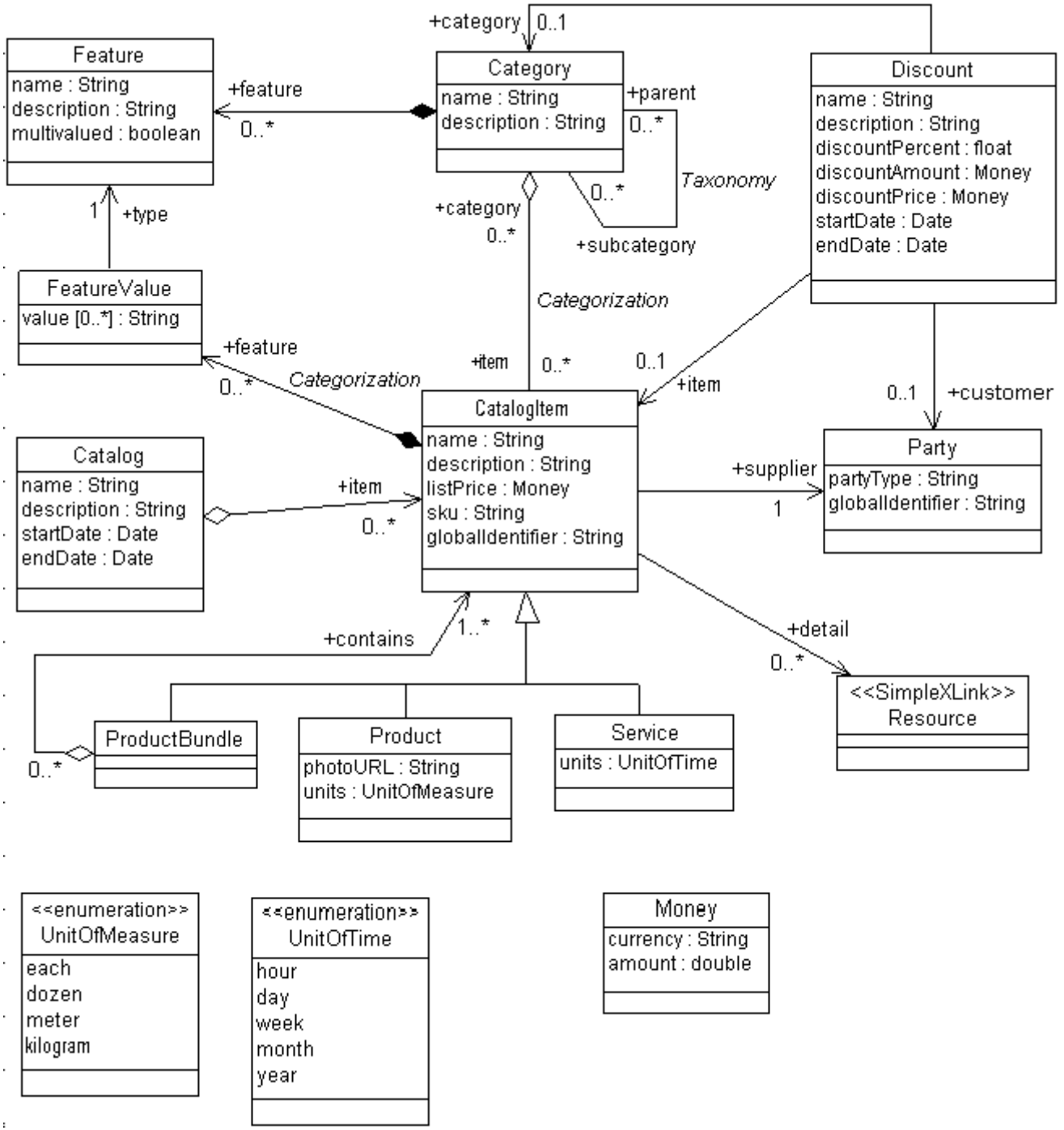


RSS case



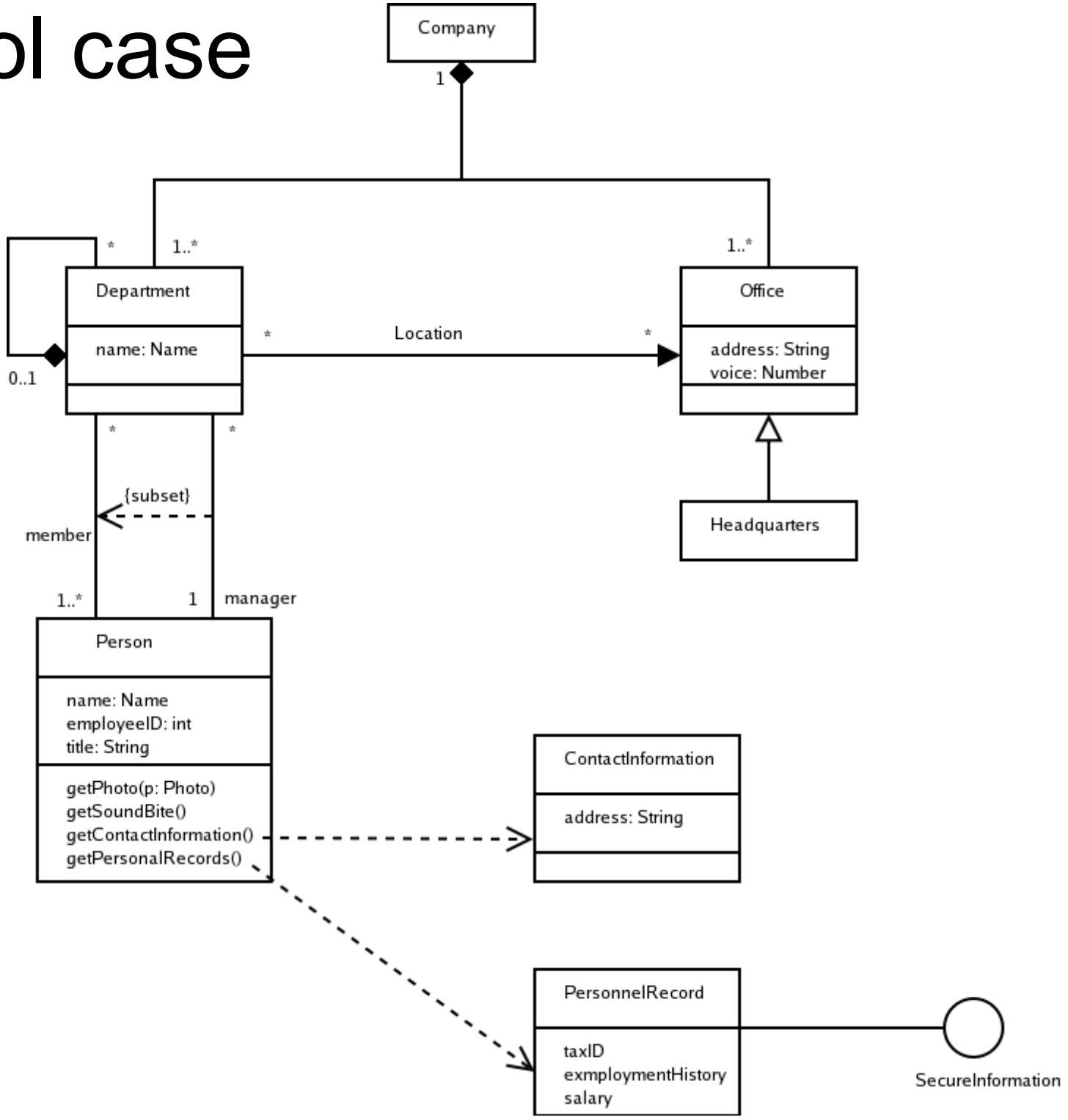
Modeling XML Applications with UML, by David Carlson, Ph.D. Published by: Addison-Wesley 2001, ISBN 0-201-70915-5

XML case



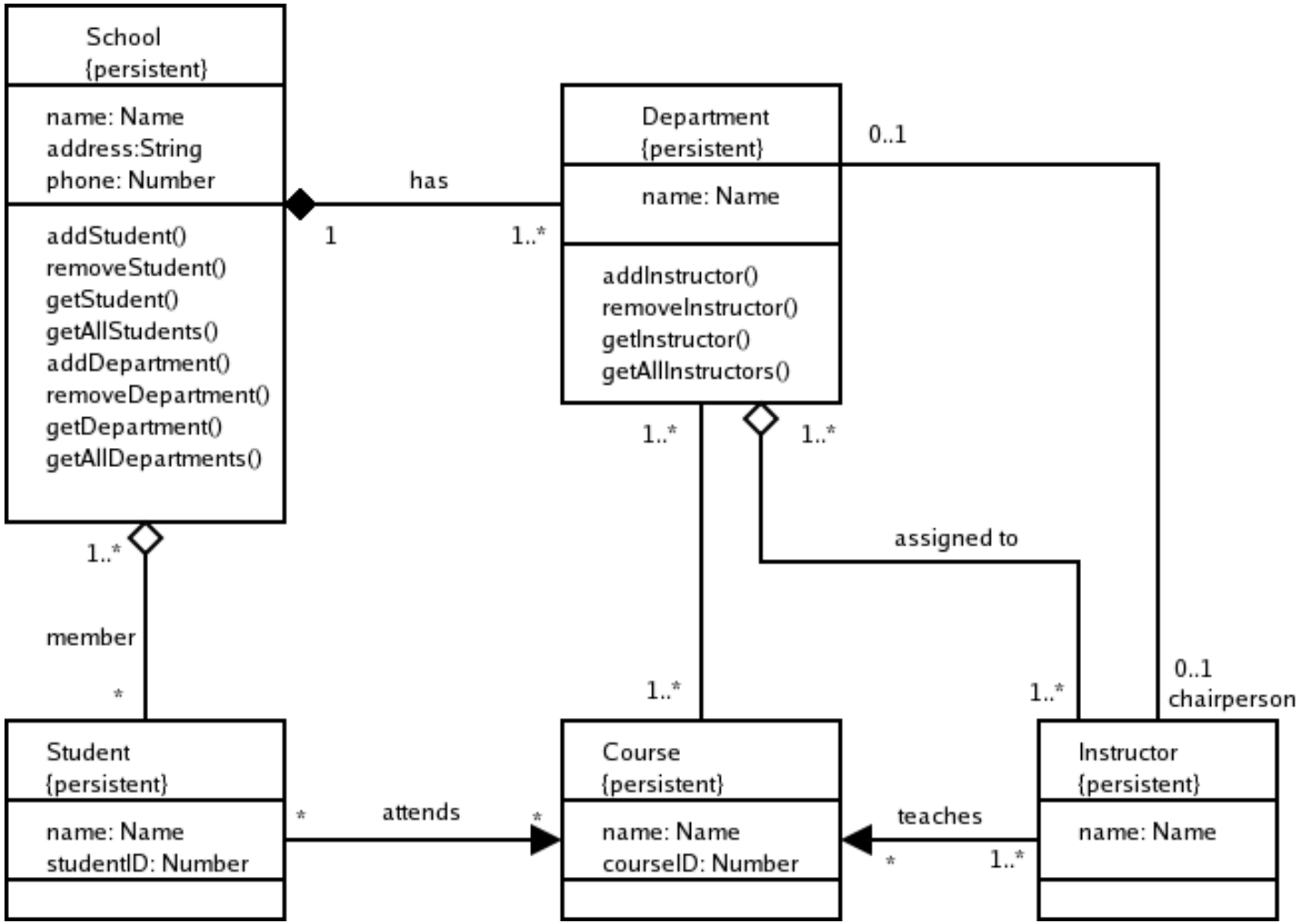
Modeling XML Applications with UML, by David Carlson, Ph.D. Published by: Addison-Wesley 2001, ISBN 0-201-70915-5

School case



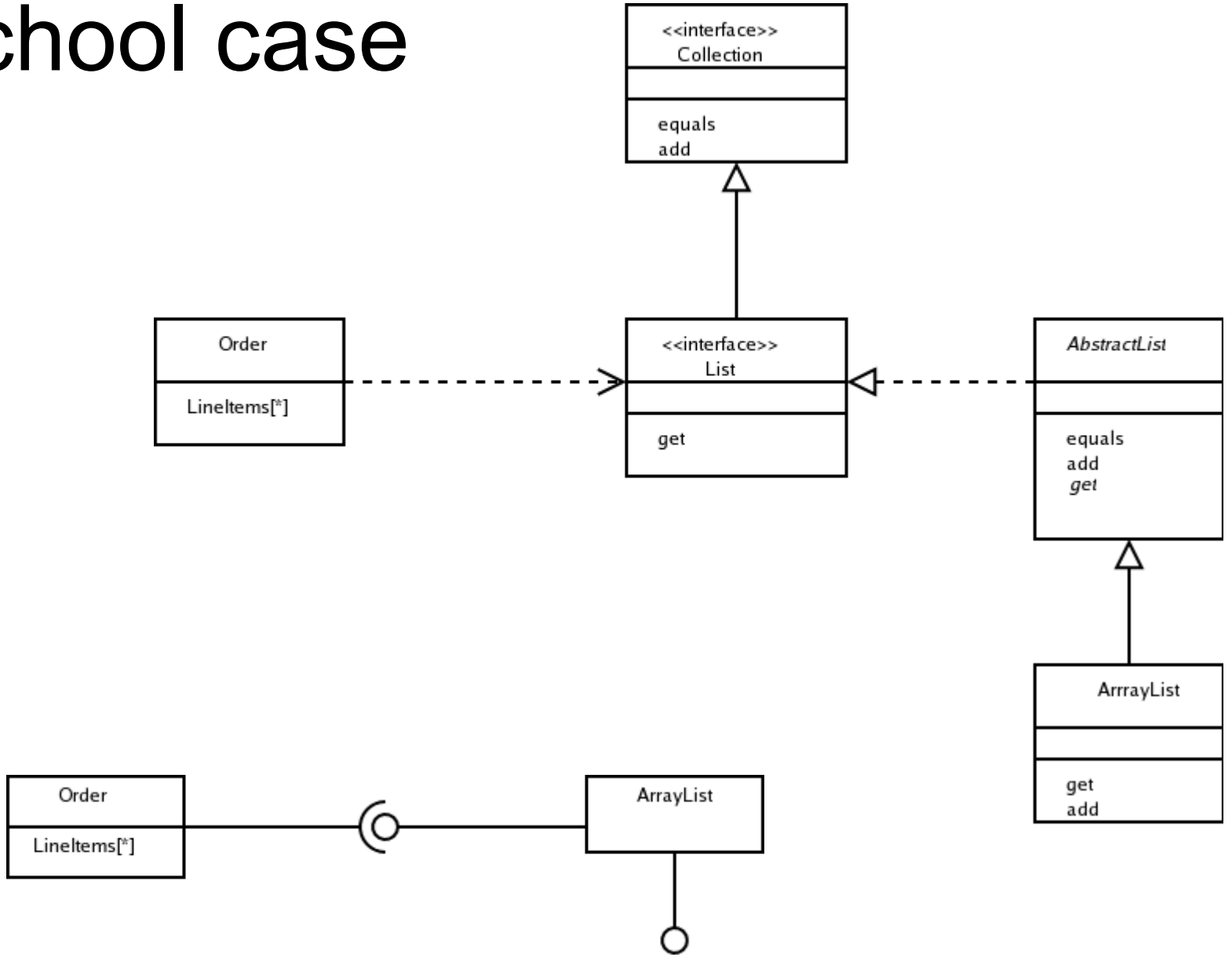
Booch, Rumbaugh and Jacobson 1999

School case



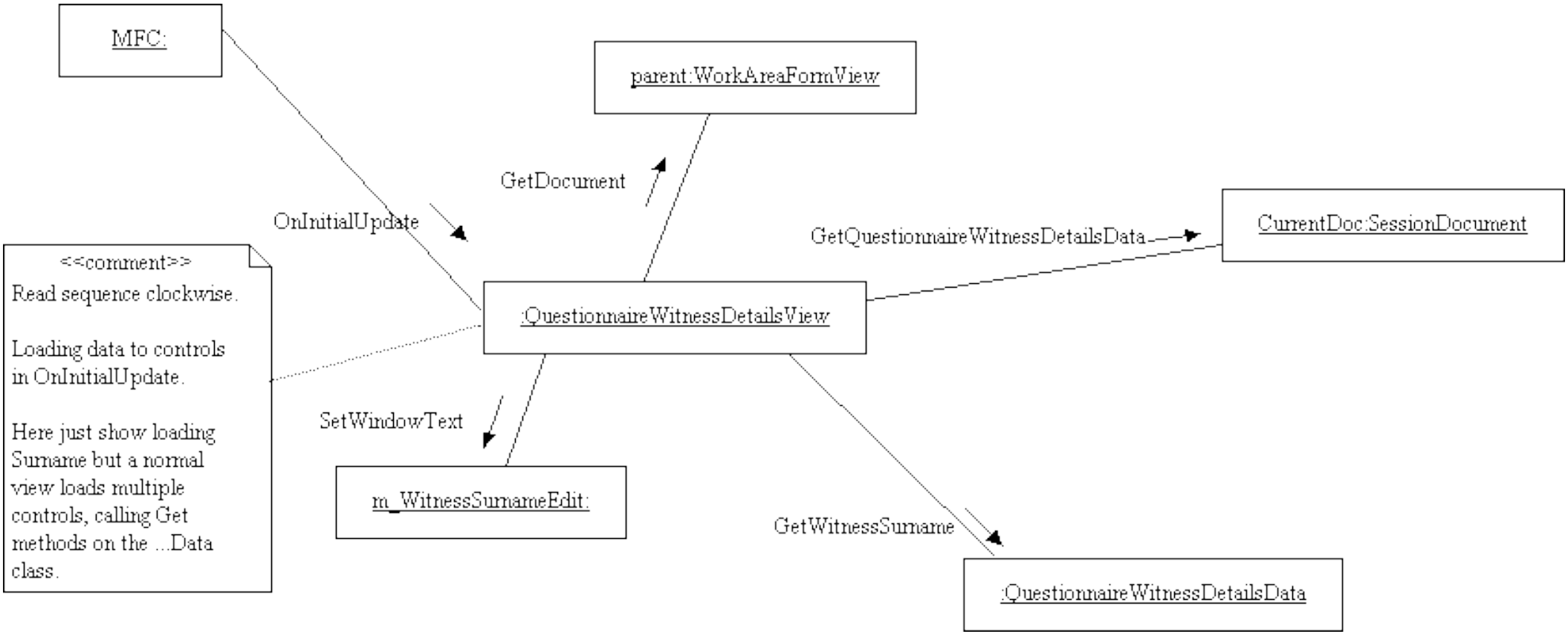
Fowler 2004

School case



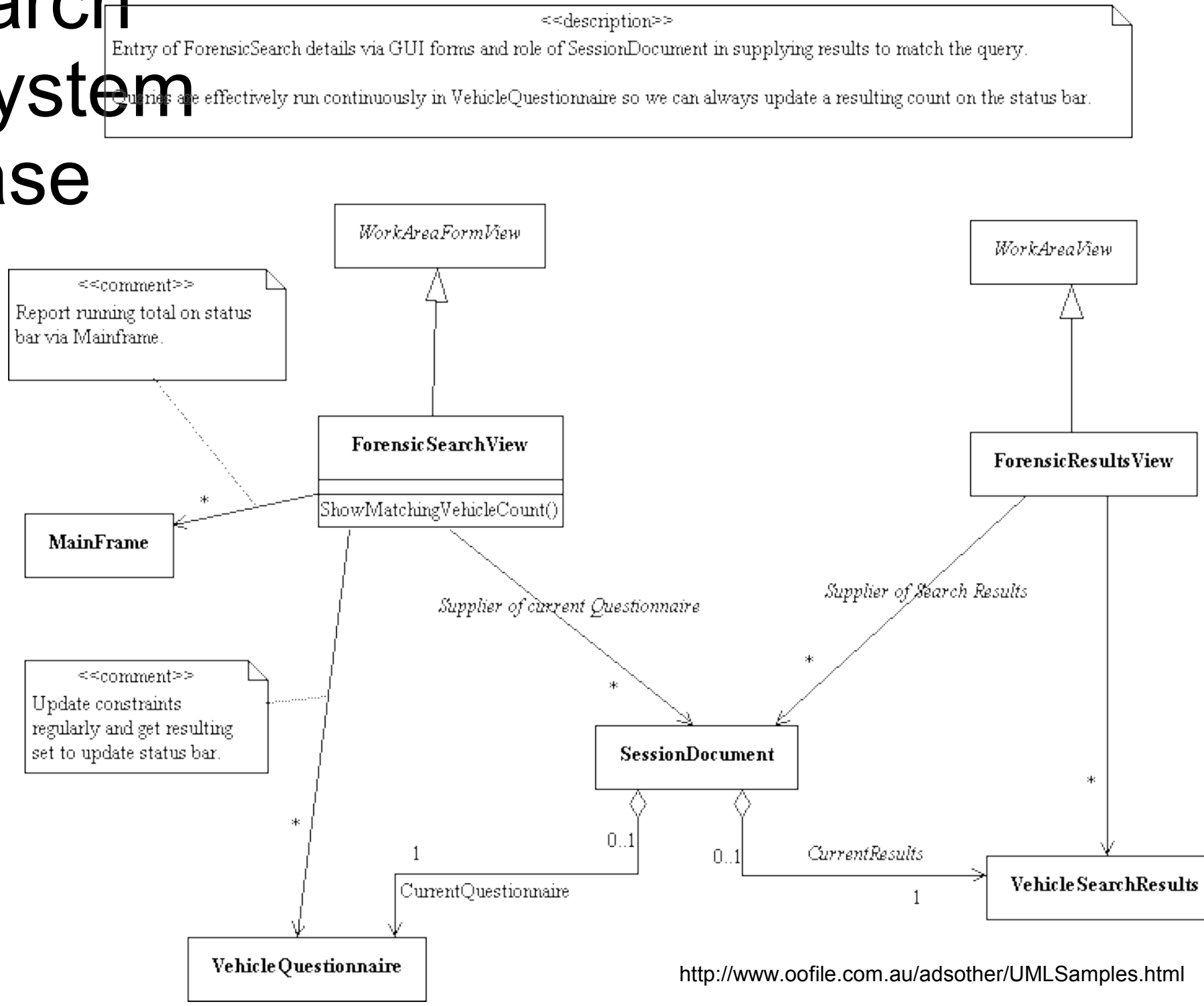
Fowler 2004

Field loading case



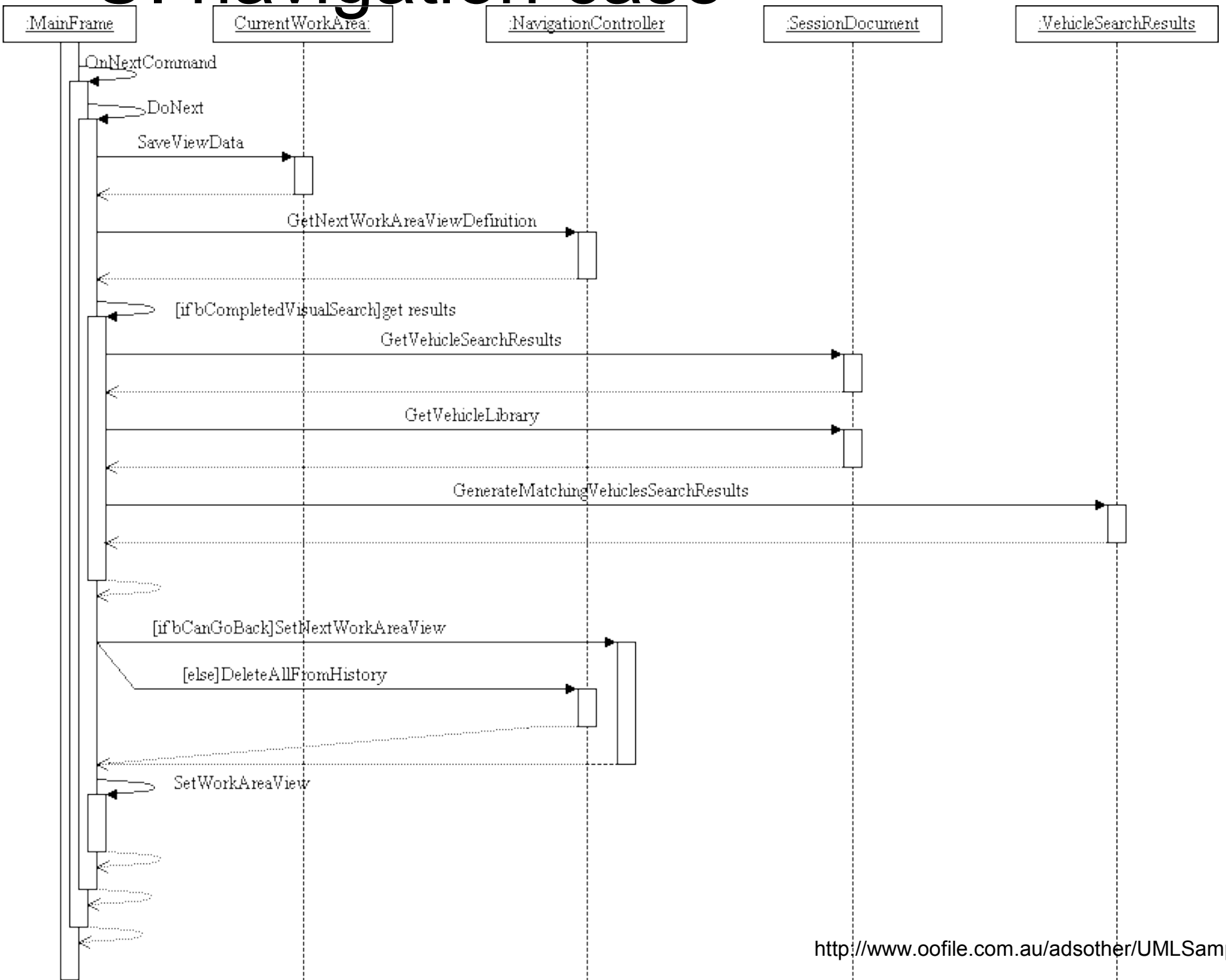
http://www.oofile.com.au/adsother/UMLSamples.html

Search subsystem case



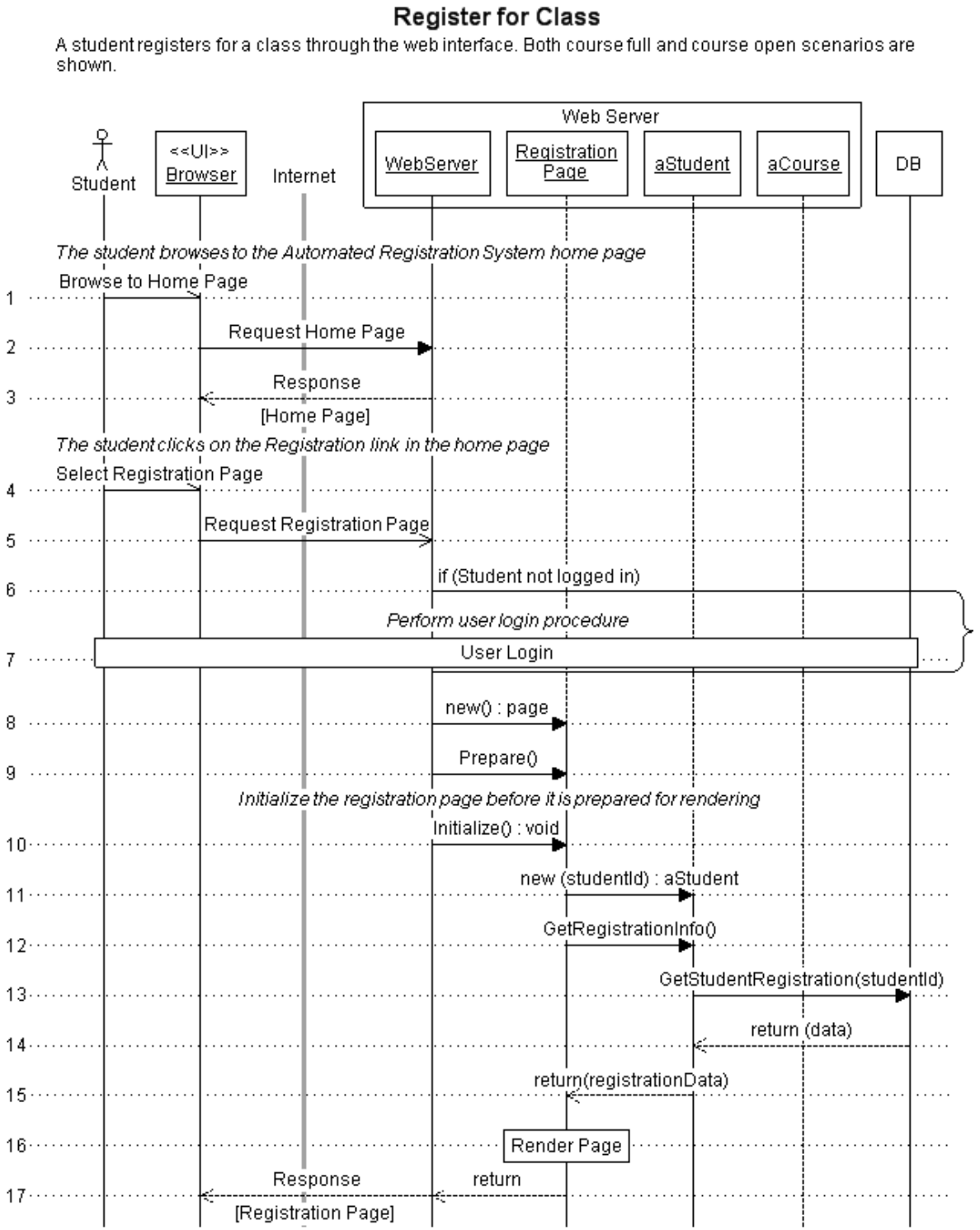
http://www.oofile.com.au/adsother/UMLSamples.html

UI navigation case

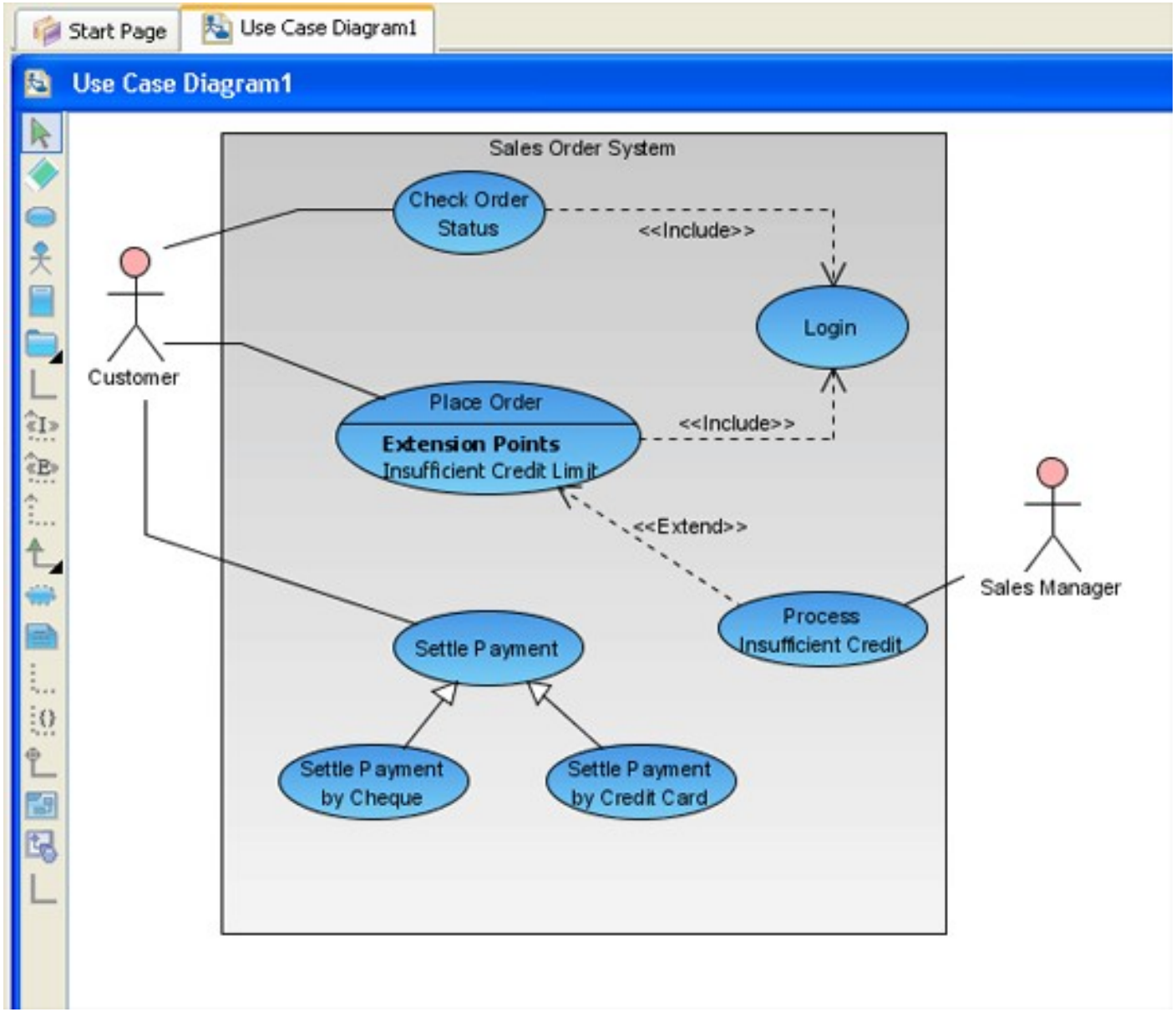


http://www.oofile.com.au/adsother/UMLSamples.html

Class registration case

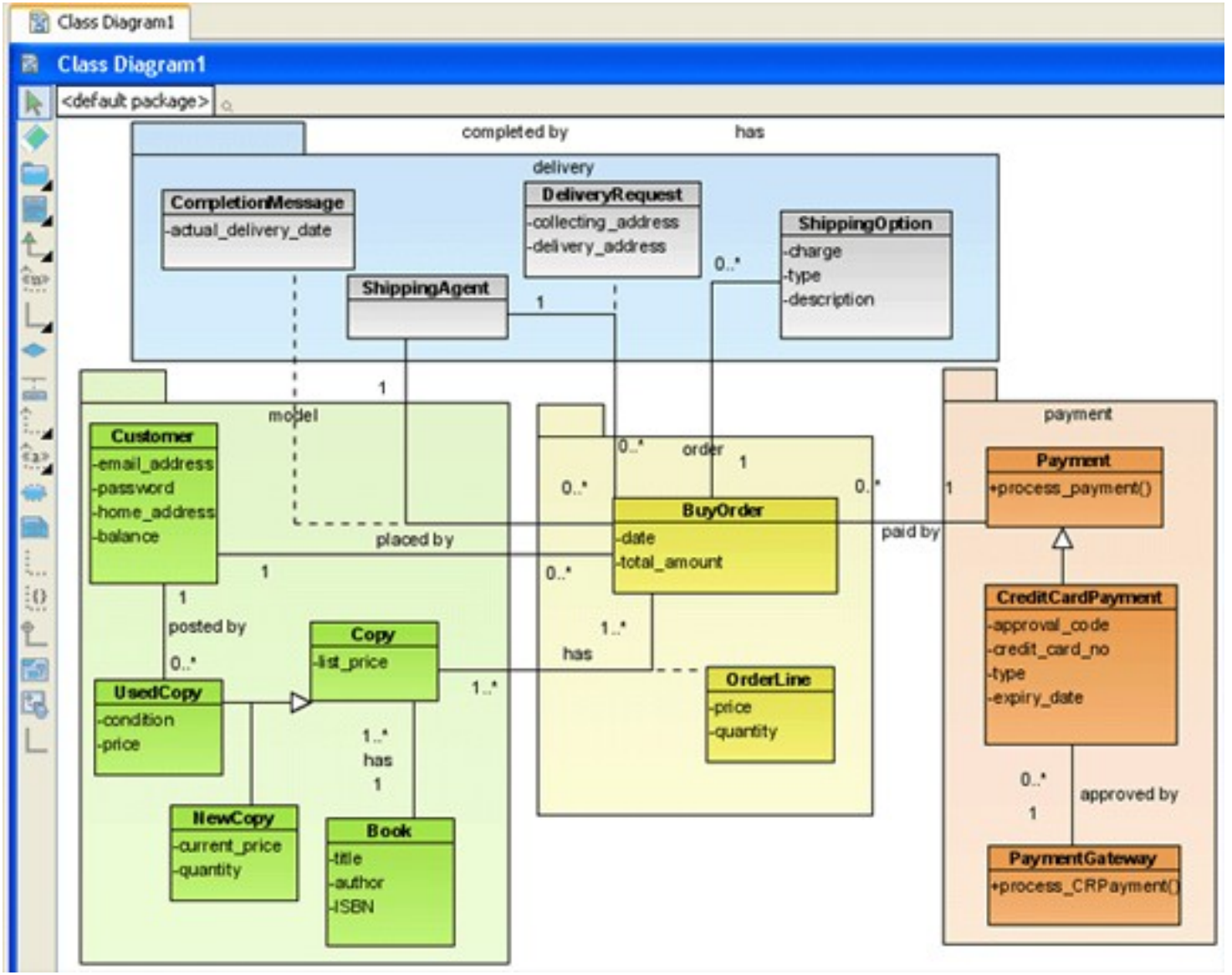


Sale case



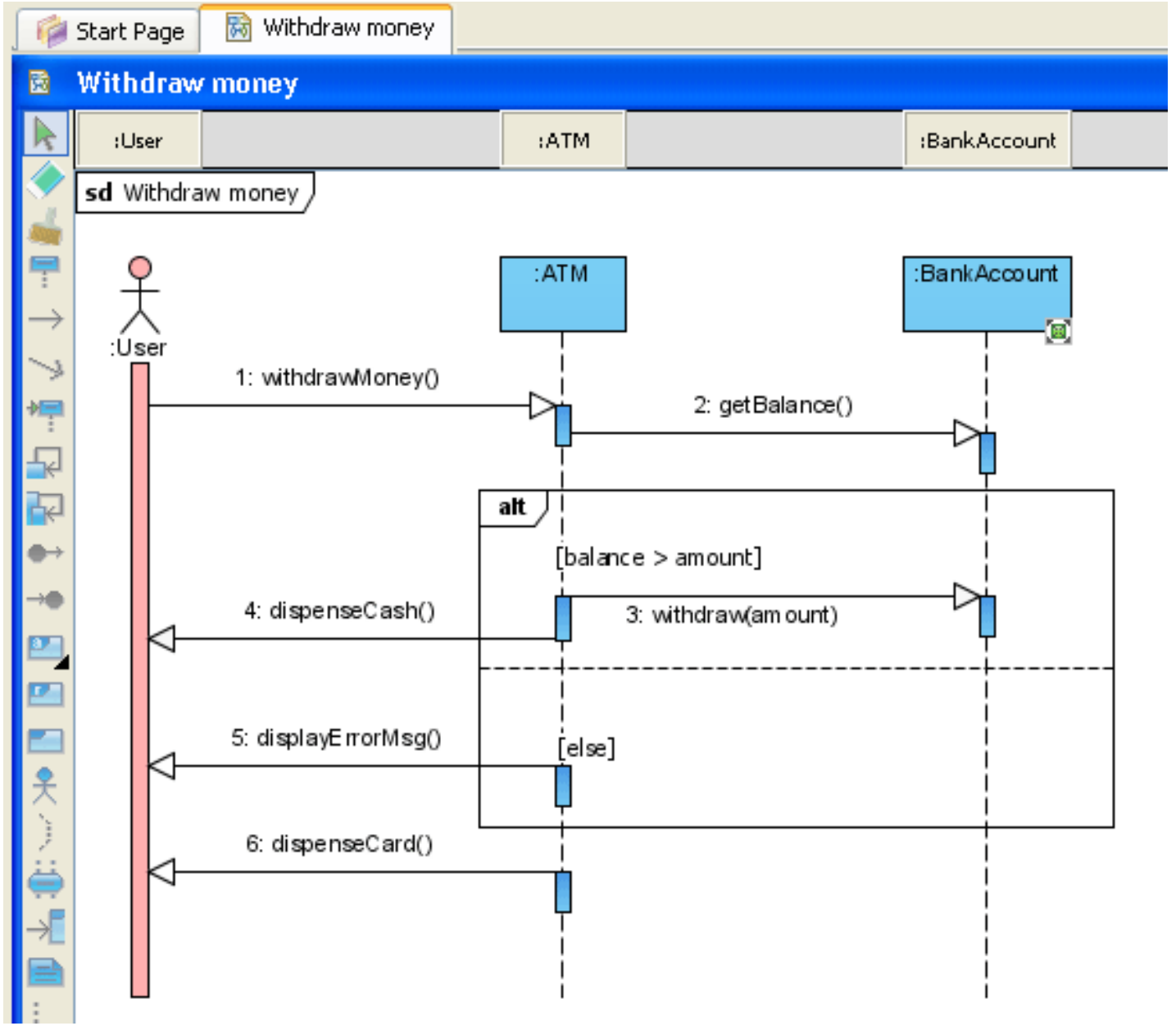
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Sale case



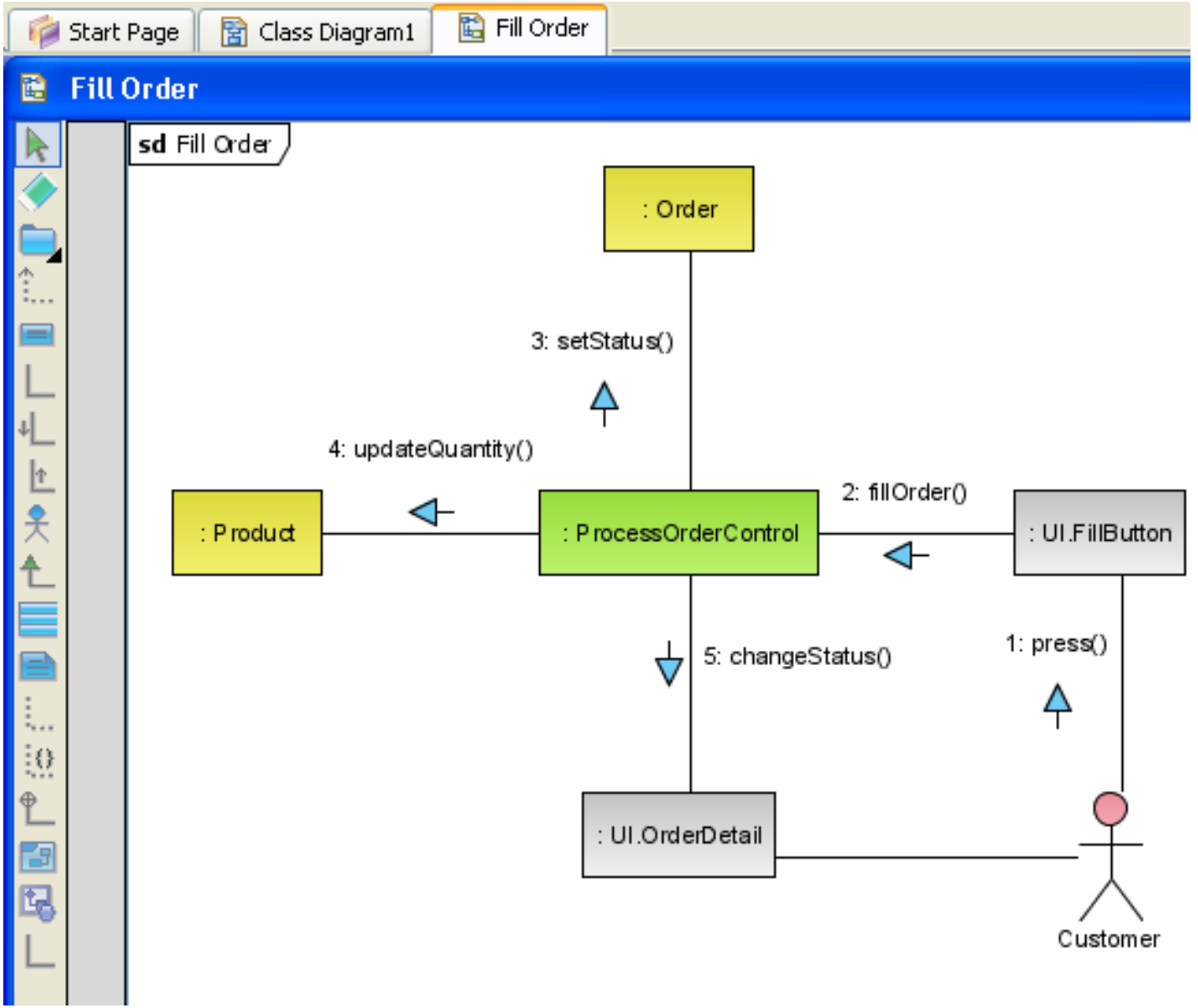
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

ATM case



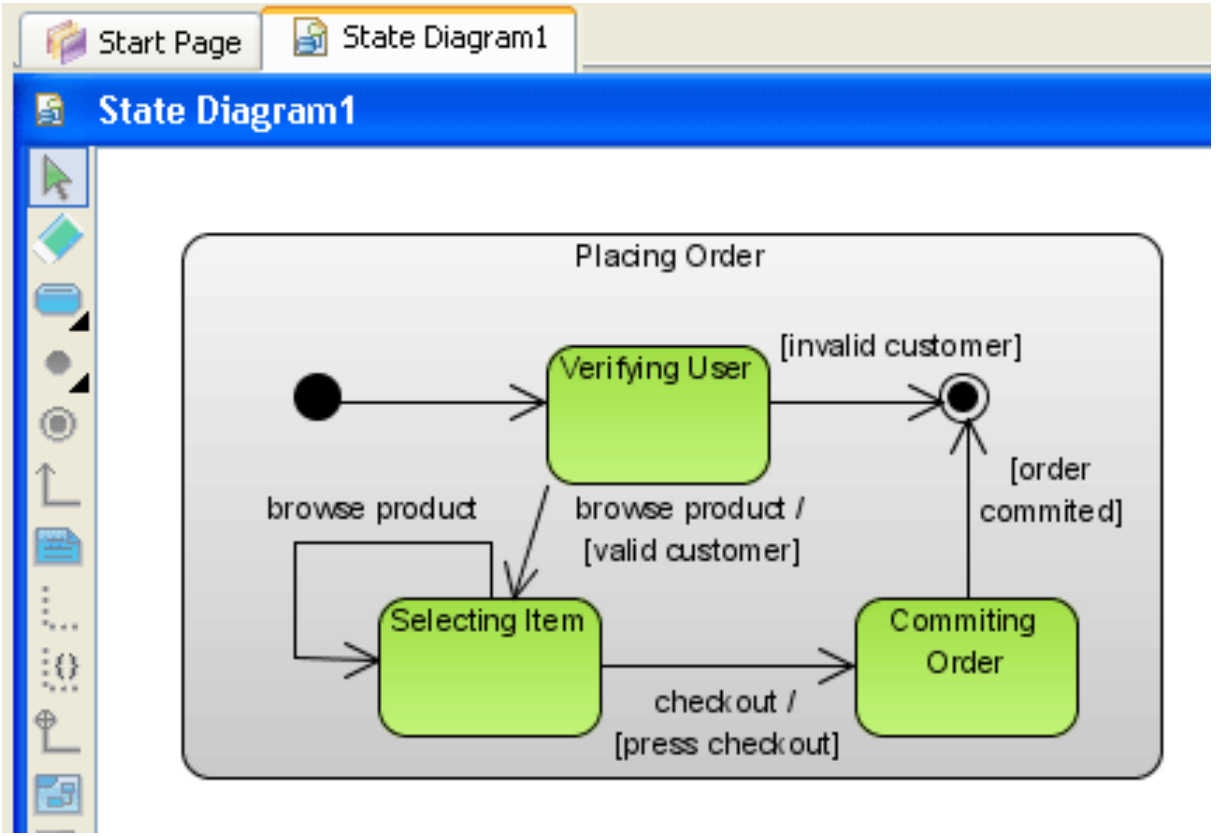
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Order case



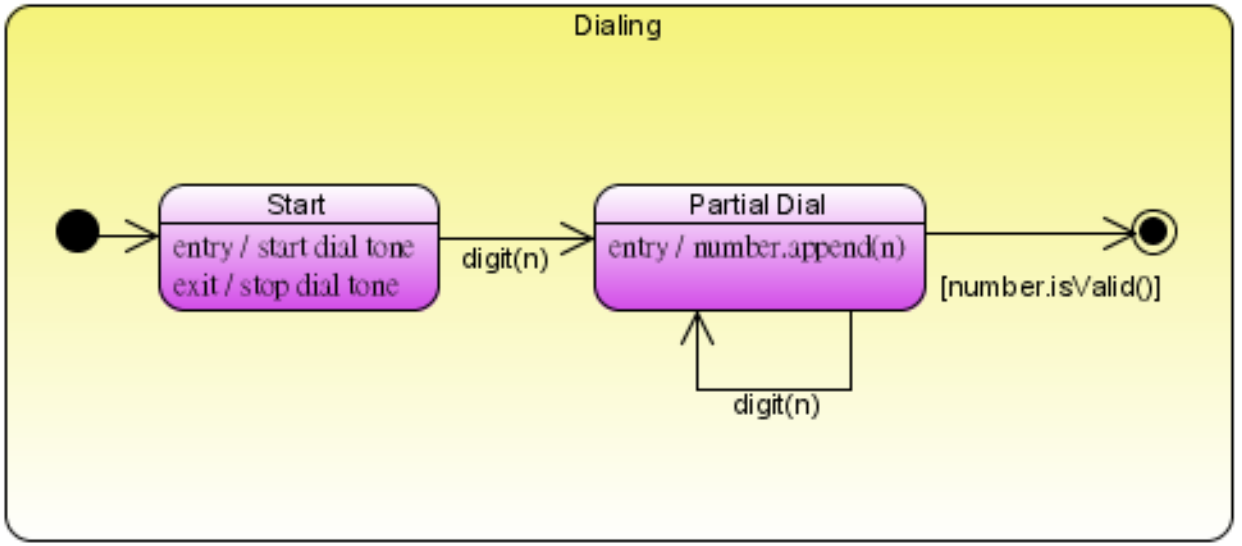
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Order case



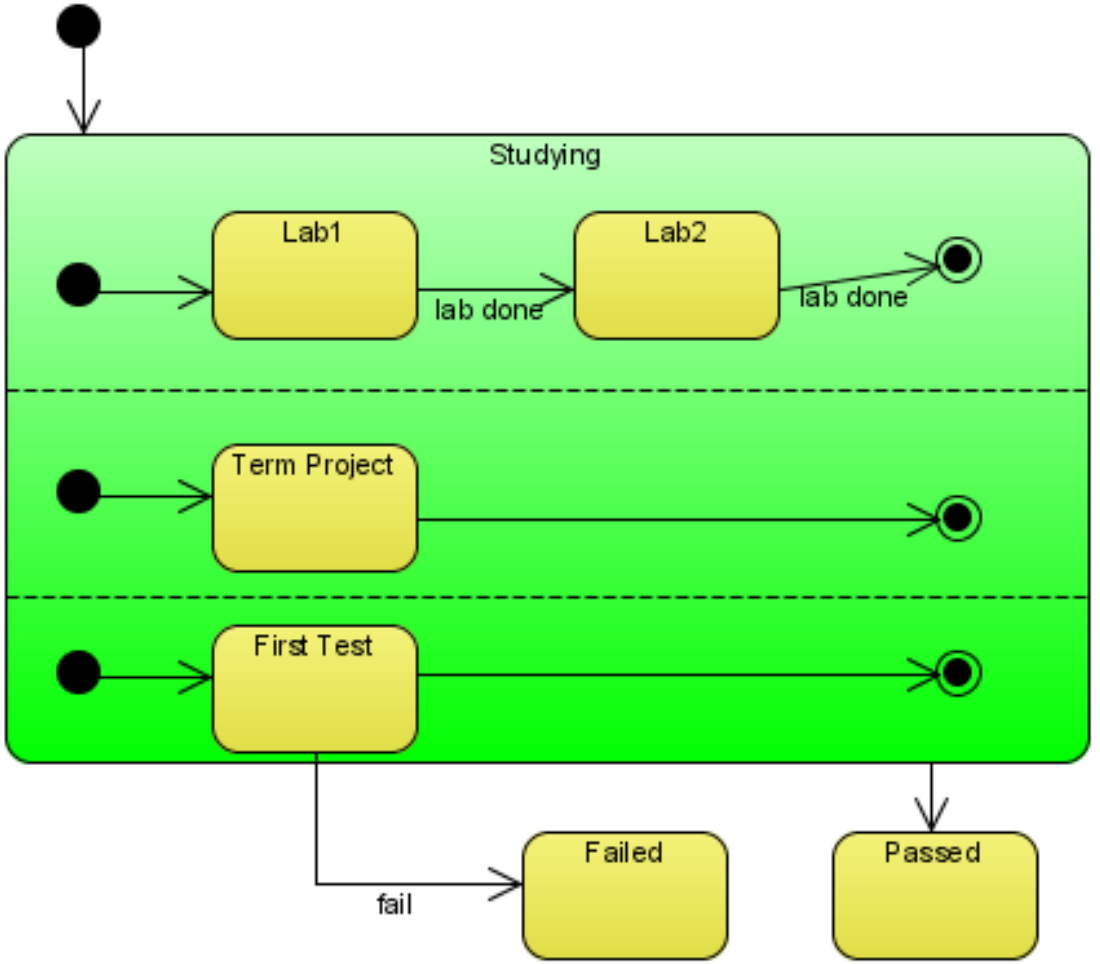
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Dialing case



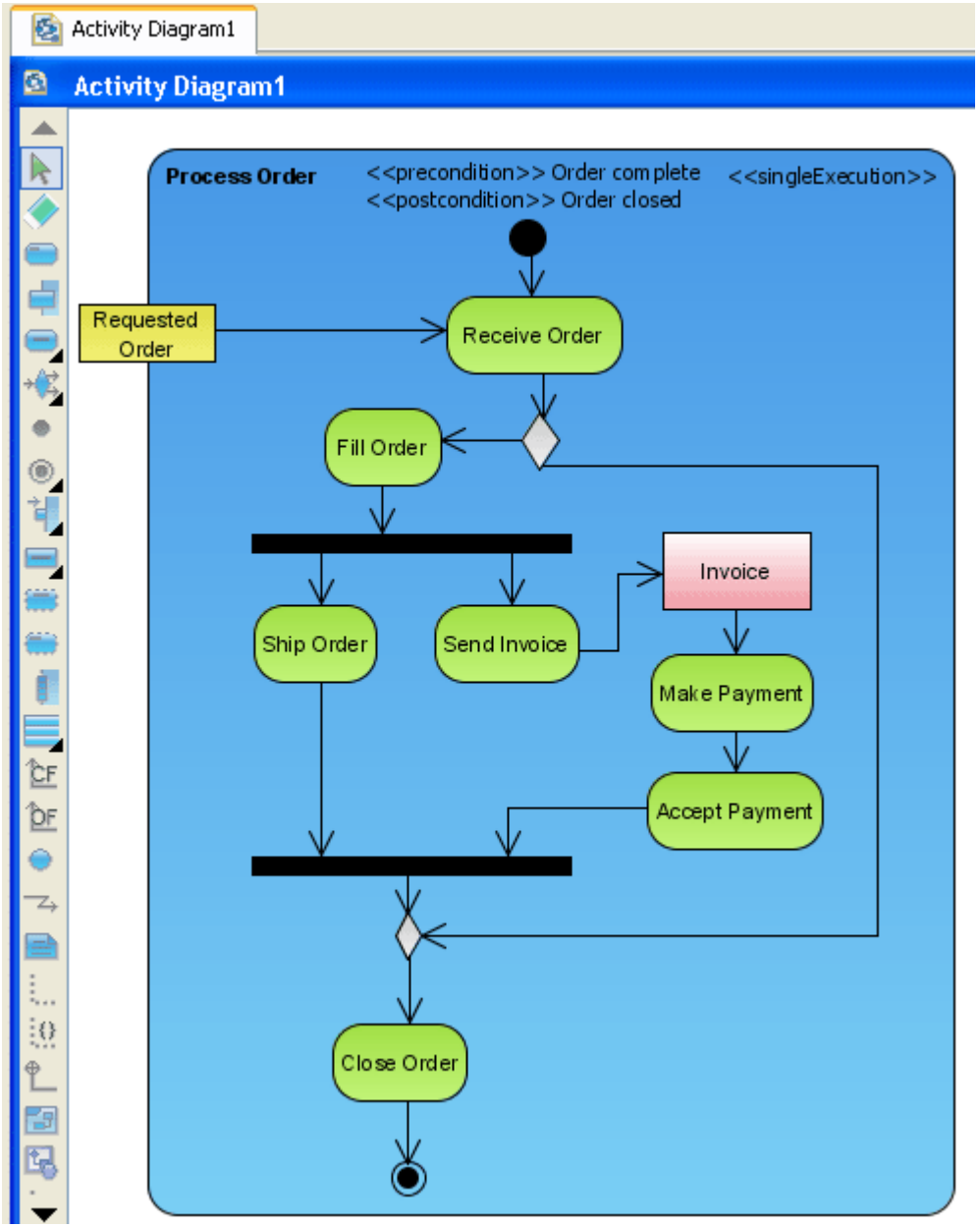
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Study case



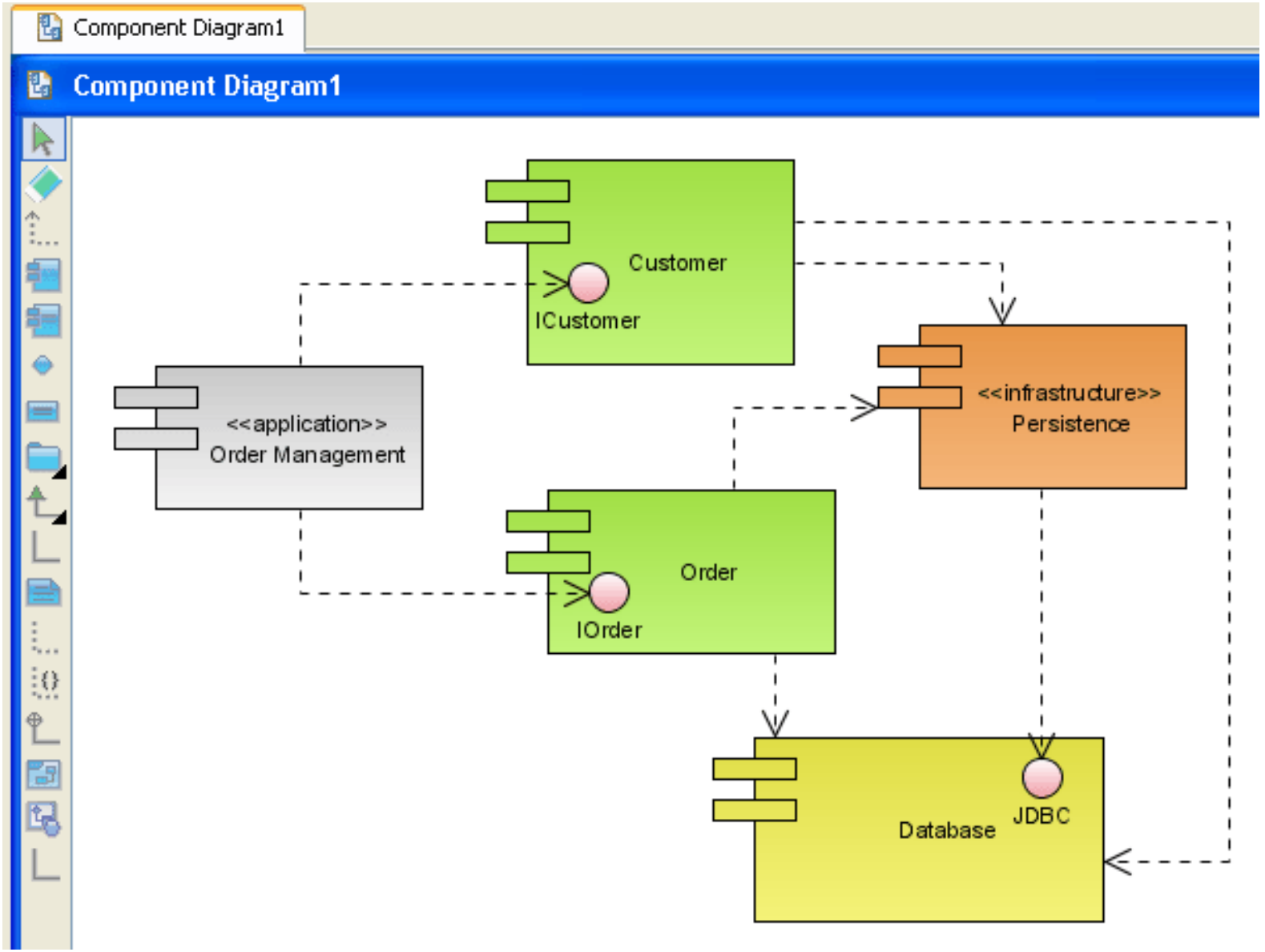
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Order case



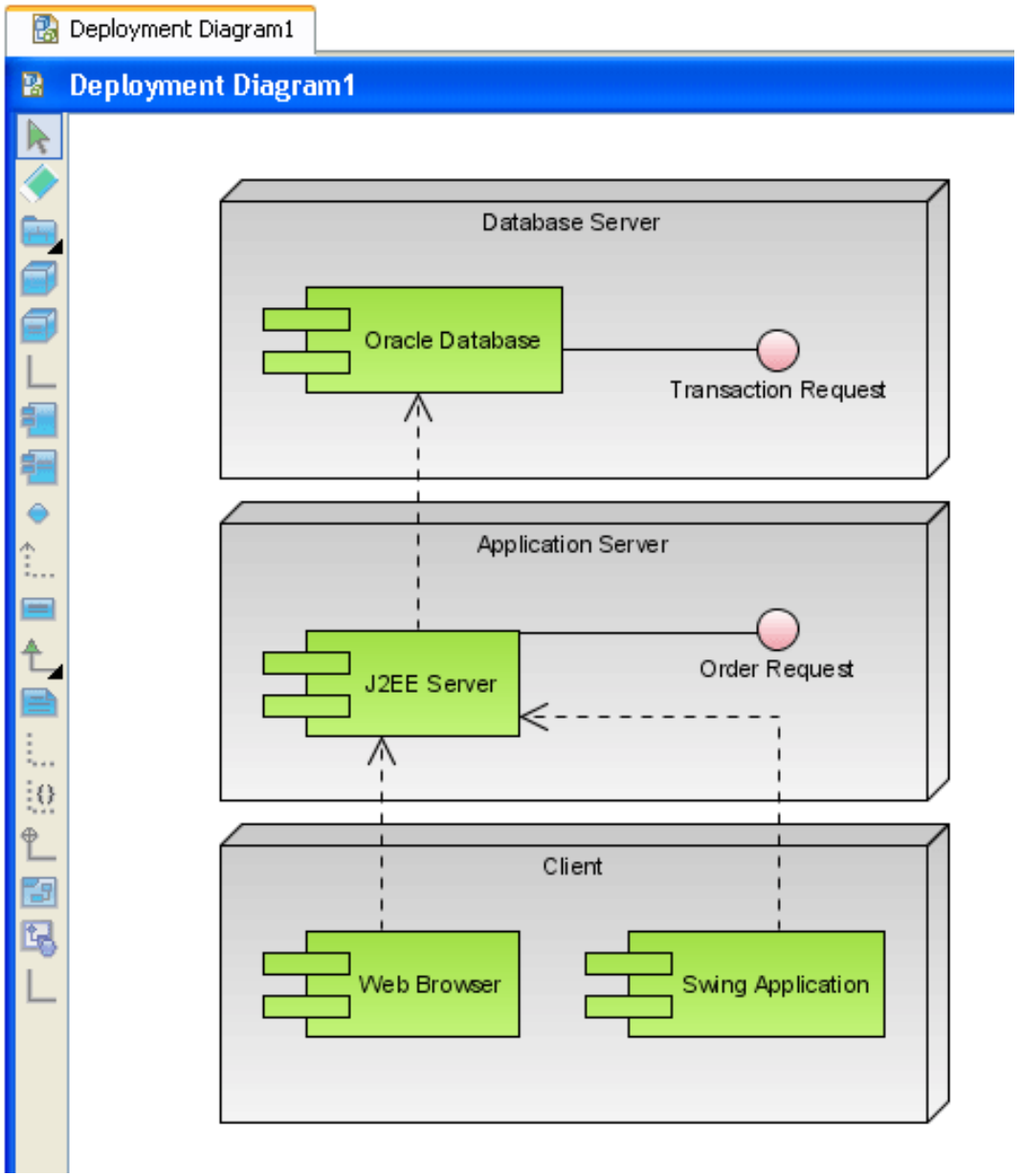
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Order case



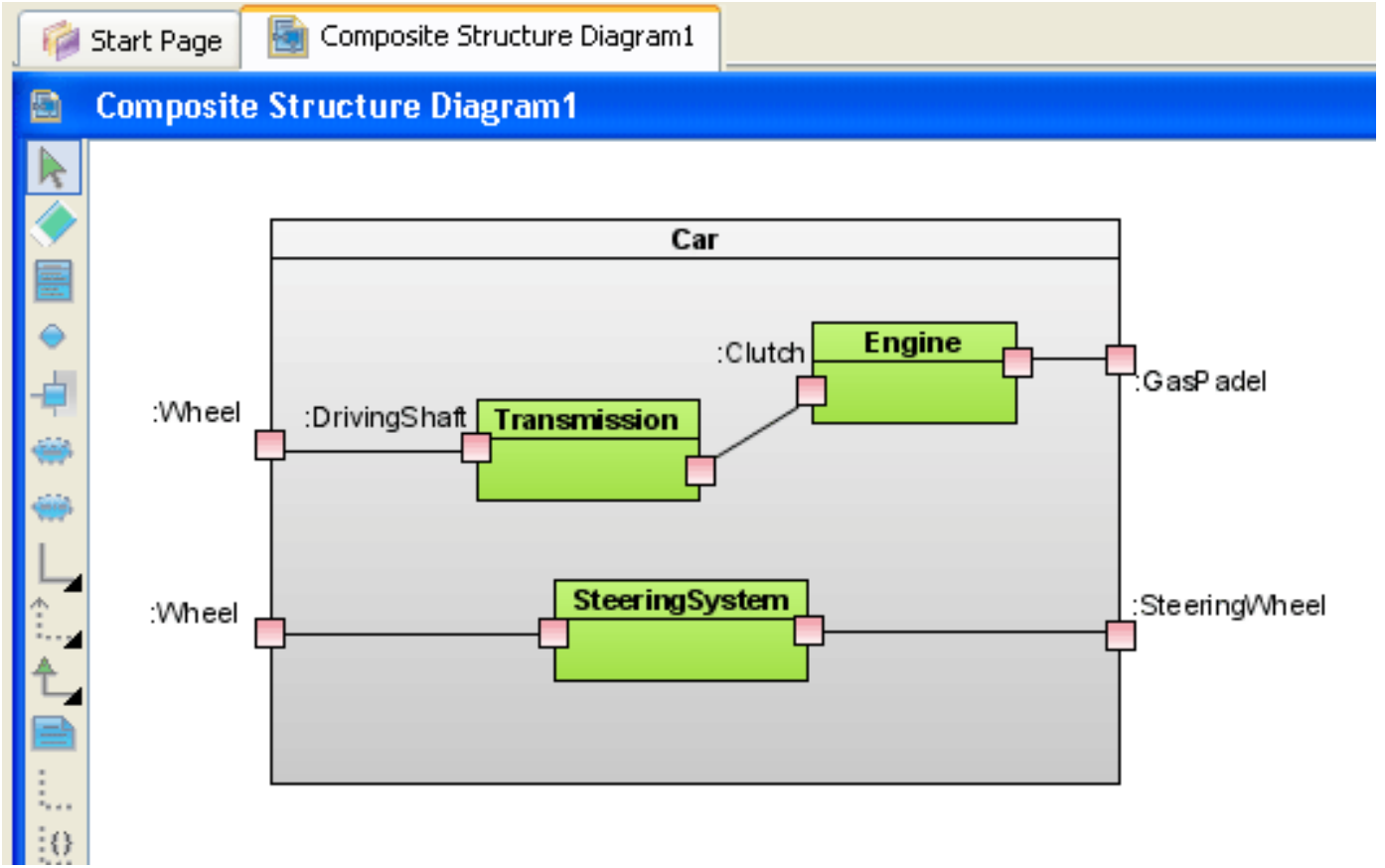
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Order case



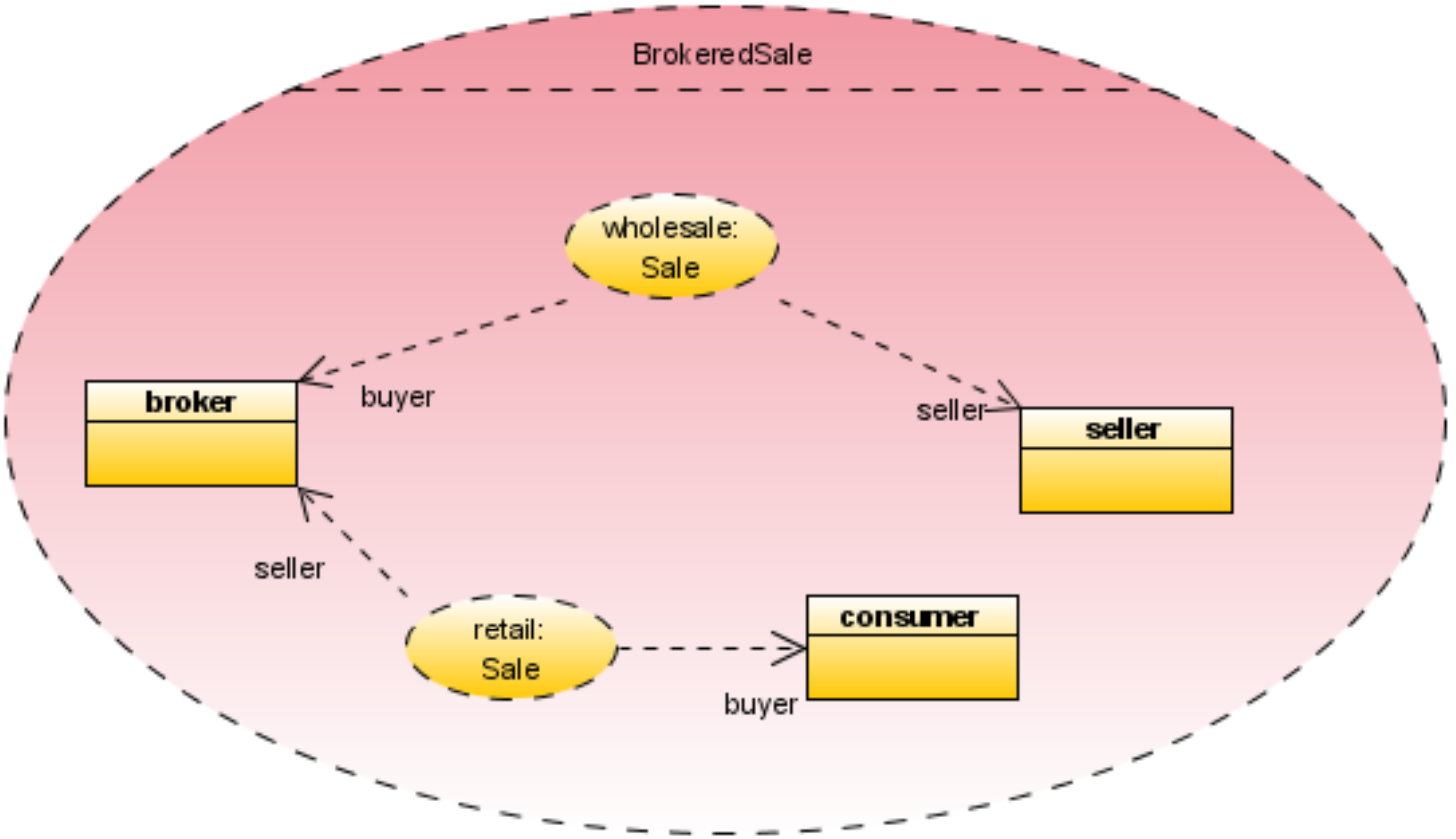
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Car case



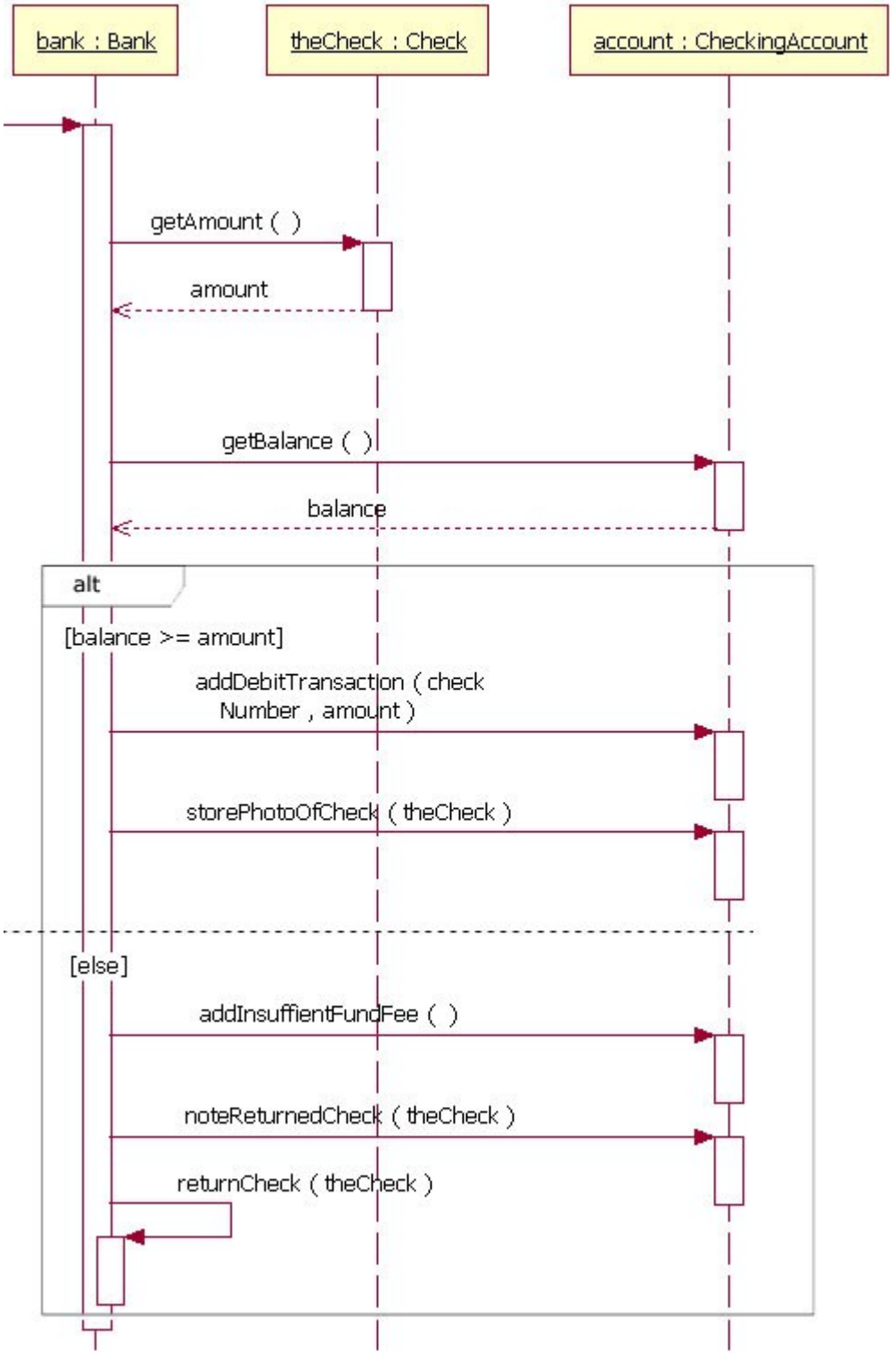
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Sale case



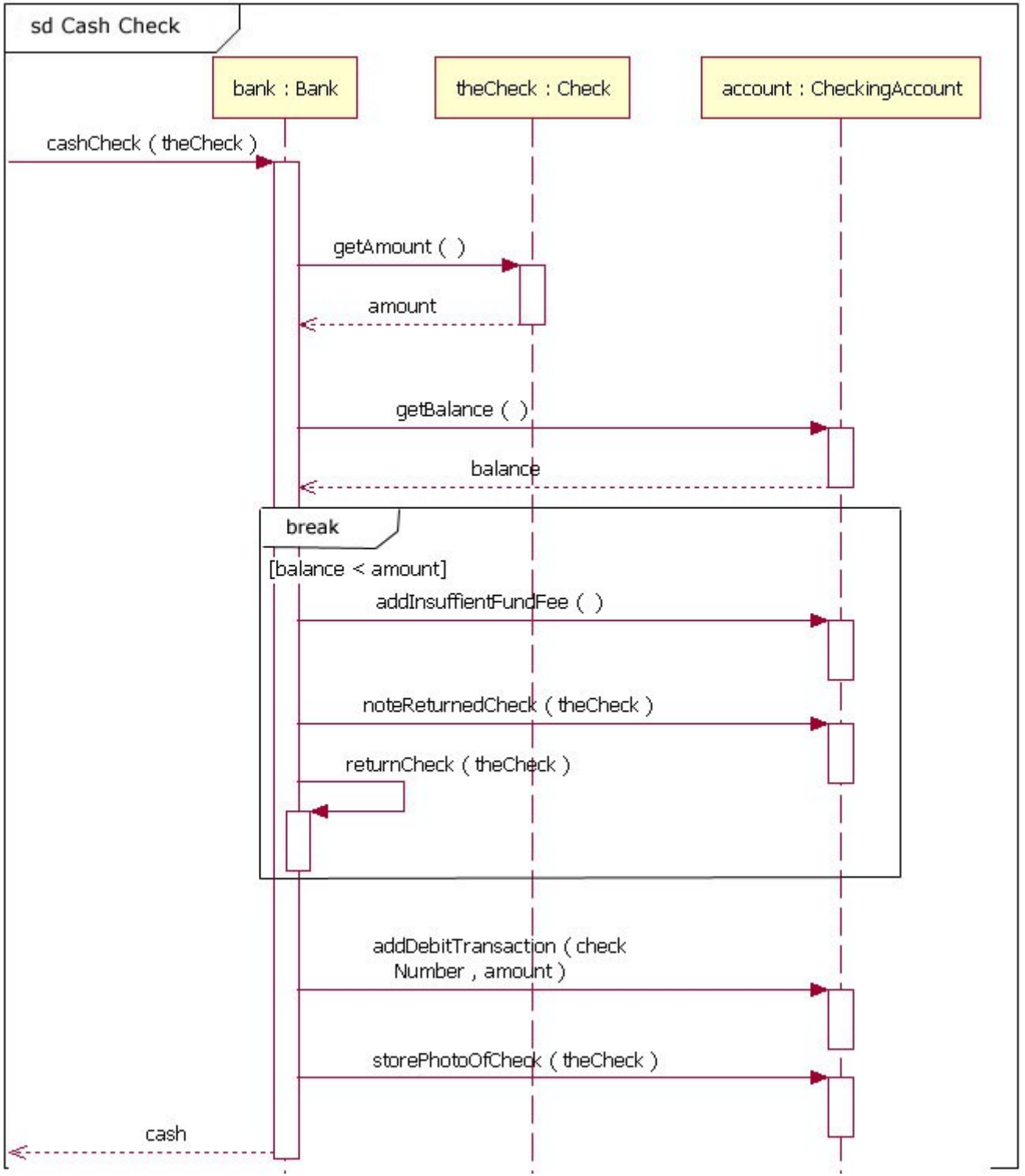
http://www.visual-paradigm.com/VPGallery/diagrams/index.html

Check case



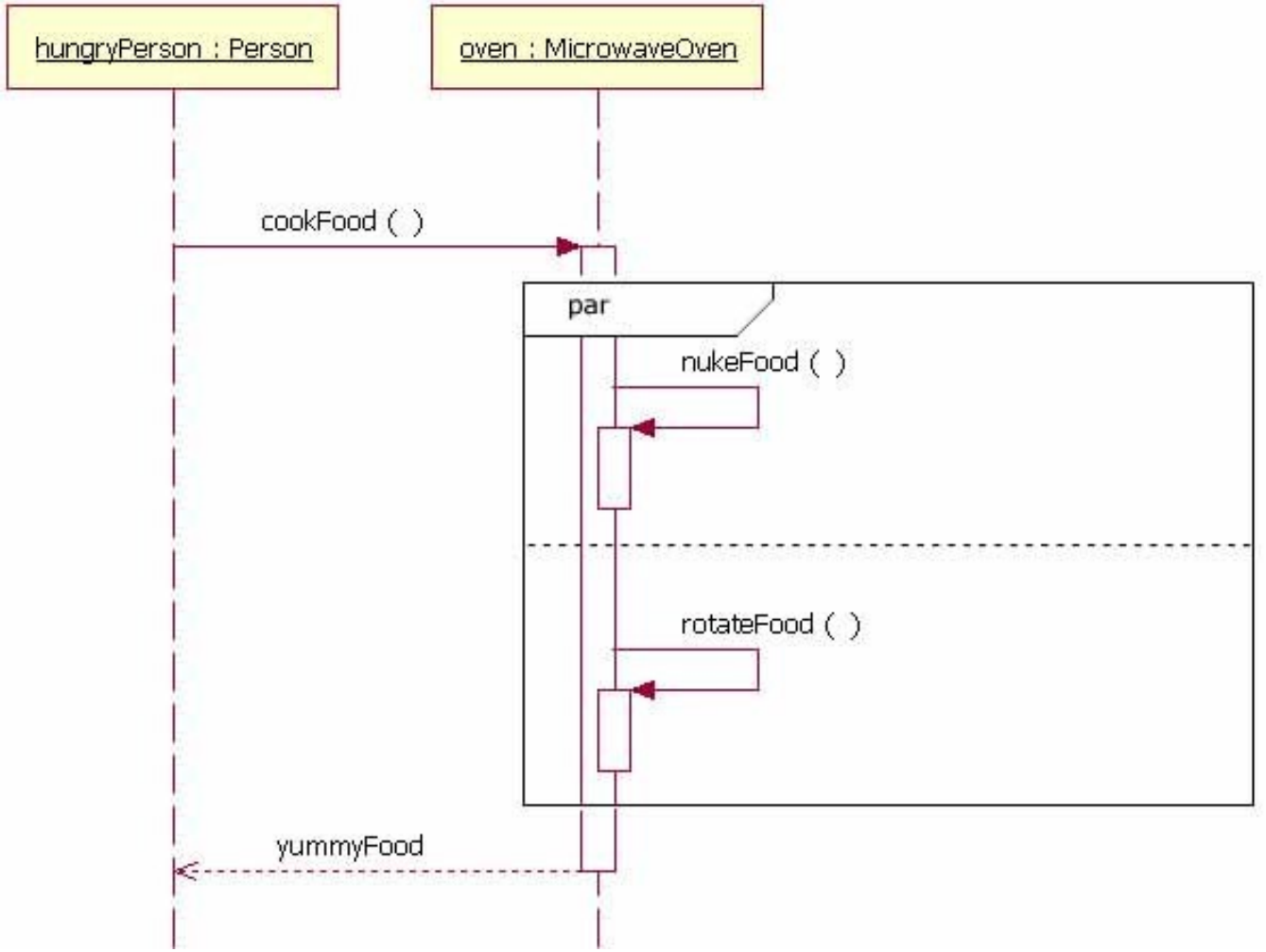
http://www-128.ibm.com/developerworks/rational/library/3101.html

Check case



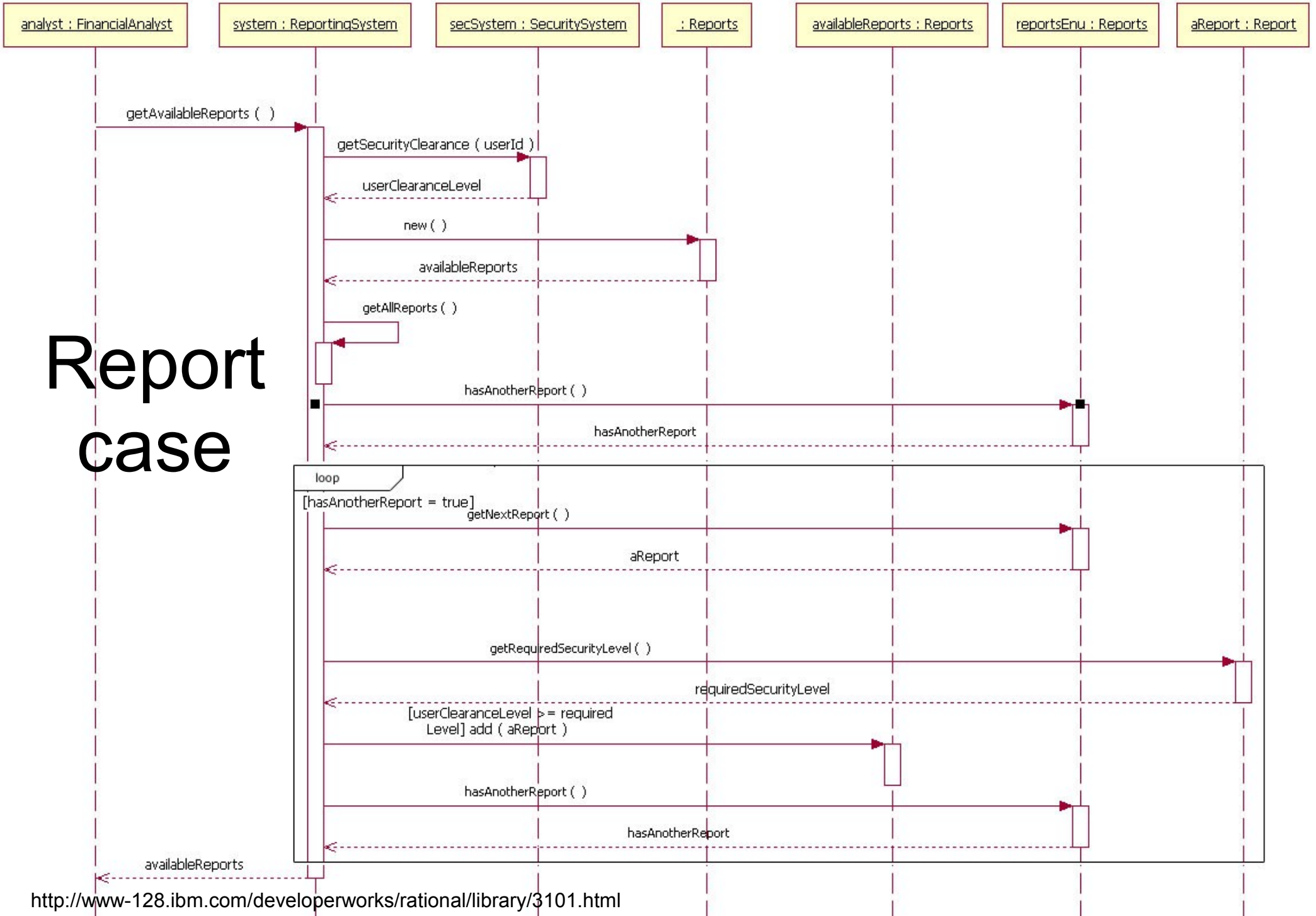
http://www-128.ibm.com/developerworks/rational/library/3101.html

Microwave case



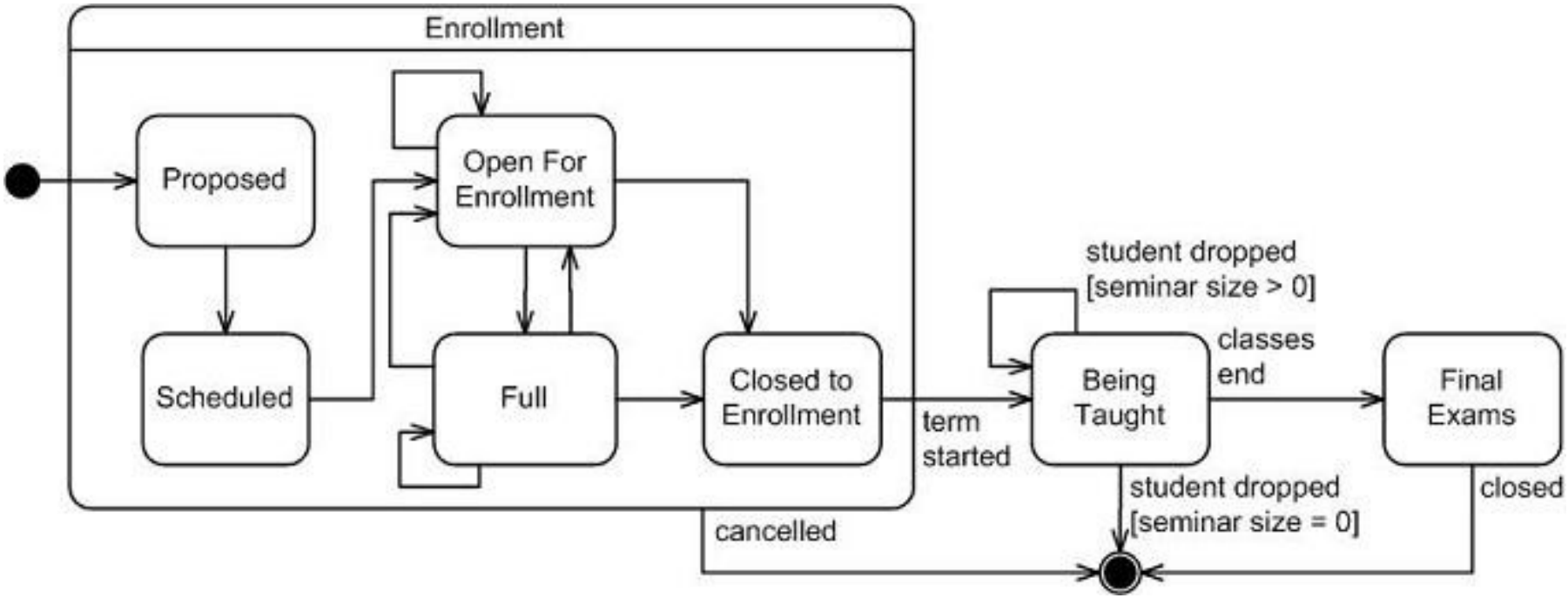
http://www-128.ibm.com/developerworks/rational/library/3101.html

Report case



http://www-128.ibm.com/developerworks/rational/library/3101.html

Seminar case



http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm