UML in software development process

(slides based on a previous work by Filippo Pacifici)

Giordano Tamburrelli
tamburrelli@elet.polimi.it

Questions ....

- What is a (software) model ?
- What are software models for ?
- What is UML structure?
- UML for requirements specification?
- UML for architectural description?
- **How can we design software with UML?**

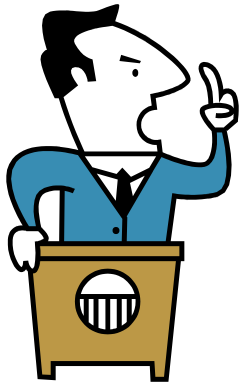
Agenda

- Introduction to models
- UML at a glance
- UML diagrams overview
- UML in software development process

Model: A definition

"A model is a representation in a certain medium of something in the same or another medium.
The model captures the important aspects of the thing being modeled and simplifies or omits the rest"

Grady Booch

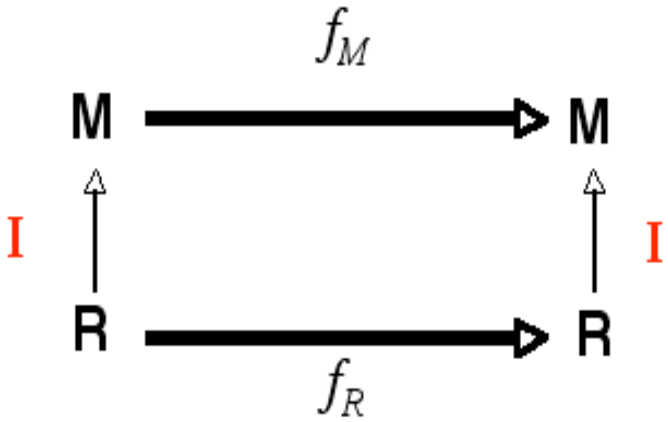


Model: what ?

- Reality R: **Real Things, People, Processes happening during some time, Relationship between things**
- Model M: **Abstractions from (really existing or only thought of) things, people , processes and relationships between these abstractions.**

Model: what ?

- Relationships, which are valid in reality R, are also valid in model M.
  - I : Mapping of real things in reality R to abstractions in the model M (Interpretation)
  - f_M: relationship between abstractions in M
  - f_R: relationship between real things in R
- In a good model the following diagram is commutative:
  - [It preserves relationships among the objects it represents.]



Model: why ?

- We use models:
  - To abstract away from details in the reality, so we can draw complicated conclusions in the reality with simple steps in the model
  - To get insights into the past or present
  - To make predictions about the future

Models of software systems

- A model of a software system is a representation of the system from a specific point of view
  - expressed in a modelling language
  - has semantics and notation
  - easier to use for a specific purpose than the final system
- Modeling language
  - artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules.

What are software models for

- Capture and precisely state requirements and domain knowledge
- Think about the design of a software system
- Generate usable work products
- Give a simplified view of complex systems
- Evaluate and simulate a complex system
- Generate potential configurations of systems
  - all consistent configurations should be possible
  - not always possible to represent all constraints in the model (model is an abstraction !)

What is in a software model

- Semantics
  - captures the meaning of a model
  - used for code generation, evaluation, validity checking....
  - has syntactic structure, well-formedness rules,...
- Notation
  - shows semantic information in a way that can be seen and edited

Modelling issues

- Abstraction vs detail
  - choose what aspects to capture and what aspects to ignore
- Specification vs implementation
  - model can define both “what” the system does and “how” it does
- Description vs instance
  - Models are descriptions, instances used as examples

Modelling issues

- Coherence
  - different views of the system must be coherent
- Variations in interpretation and ambiguity
  - define where different interpretations of the model are acceptable

Agenda

- Introduction to models
- UML at a glance
- UML diagrams overview
- UML in software development process

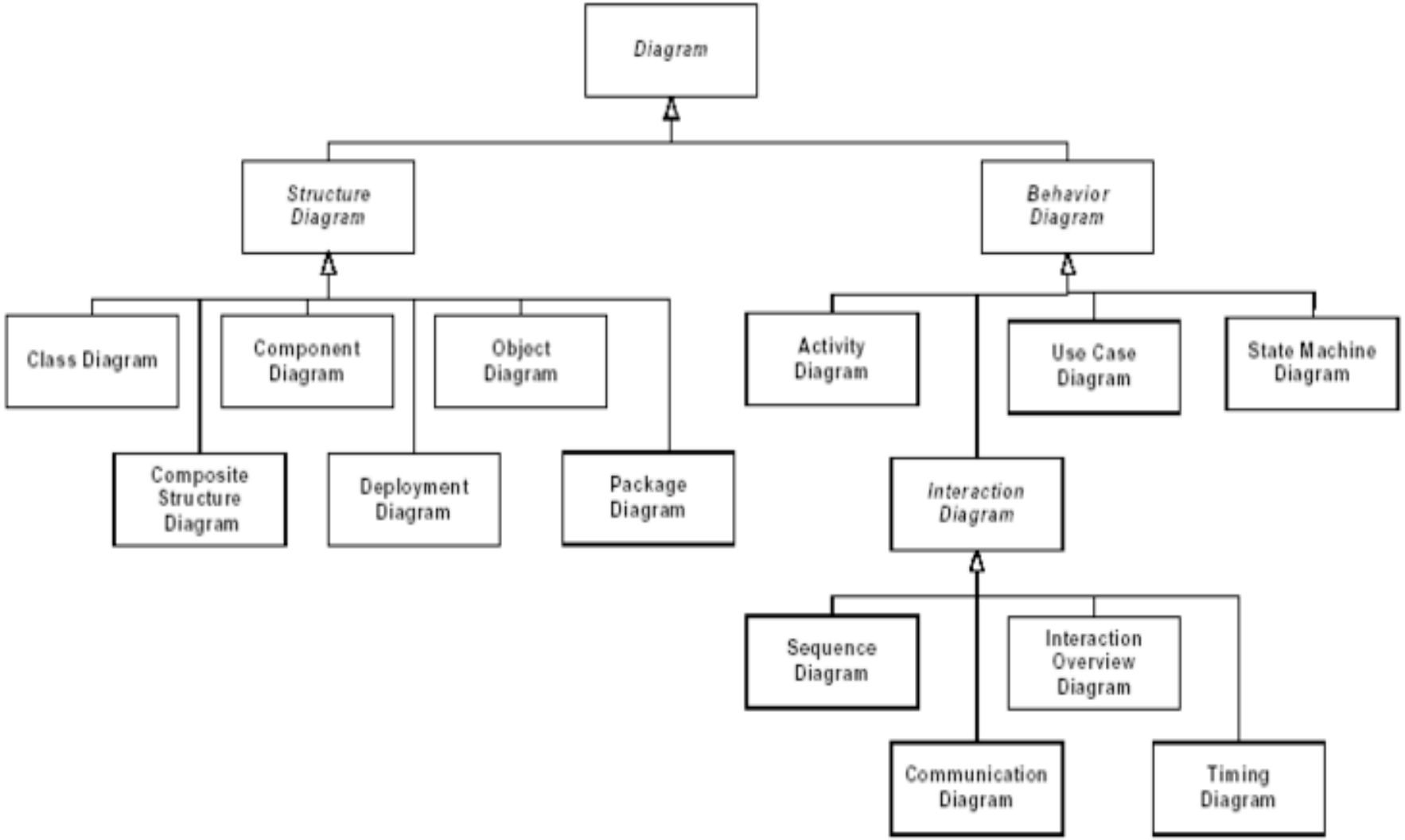
Brief Summary of UML

- Unified Modelling Language
  - general purpose, visual modelling language
  - used for specify, visualize, design, document software systems
  - intended to be supported by CASE tools
  - static structure and dynamic behaviour
  - models discrete systems

Views taxonomy

- 13 types of views in UML
- divided into:
  - Structural views
    - Class diagrams, Object diagrams, Component diagrams,....
  - Behavioral views
    - State diagrams, Activity diagrams, Use case diagrams,...
  - Interaction views
    - Sequence diagrams,...

View taxonomy



Ways of using UML

- UML as Sketch
  - Selectivity is the key issue. UML sketches point out some issues in code you are about to write (the sketch does not describes all the code). The aim is to communicate ideas and alternatives. Sketches sessions are quite informal and dynamic (e.g. whiteboard sessions). Sketches are by far the most common usage of UML.
- UML as Blueprint
  - Completeness is the key issues. The goal is to build a complete description in which all the design decisions are laid out. Blueprints can be used for details or for a particular area of the system.

Ways of using UML

- The distinction between sketches and blueprints is blurry. The former are deliberately incomplete, used to point out important key issues. The latter are comprehensive with the aim of reducing programming to a simple mechanical activity.
- Reading a UML diagram it is important to understand the point of view of the author

  “Sketches are explorative, while blueprints are definitive” *Martin Fowler*

UML perspectives

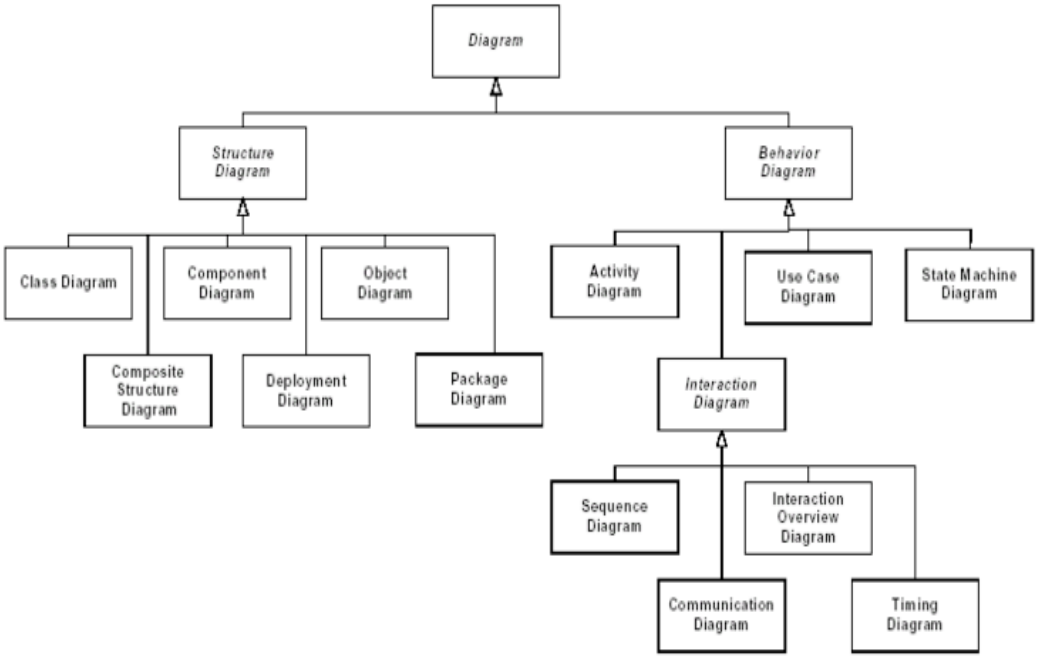
- UML can be adopted to model specific software concepts (Software Perspective) or more generally domain concepts (Conceptual Perspective).
- In Software Perspective the elements of the UML map directly to elements in a software system.
- In Conceptual Perspective, the UML represents a description of concepts of a domain of study.
- Examples:
  - SW Perspective: adopted between SW architects and programmers.
  - CP Perspective: adopted between SW analysts and customers.

Agenda

- Introduction to models
- UML at a glance
- UML diagrams overview
- UML in software development process

UML diagrams overview

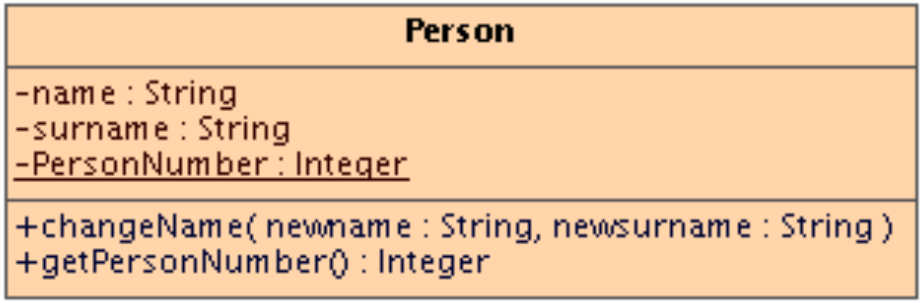
- UML Diagrams
  - Structure Diagrams
    - Class Diagram
    - Object Diagram
    - Component diagram
  - Behaviour Diagrams
    - Use Case
    - Activity Diagram
    - State Machine Diagram
    - Interaction Diagrams
      - Sequence Diagram



Class diagrams / Object diagrams

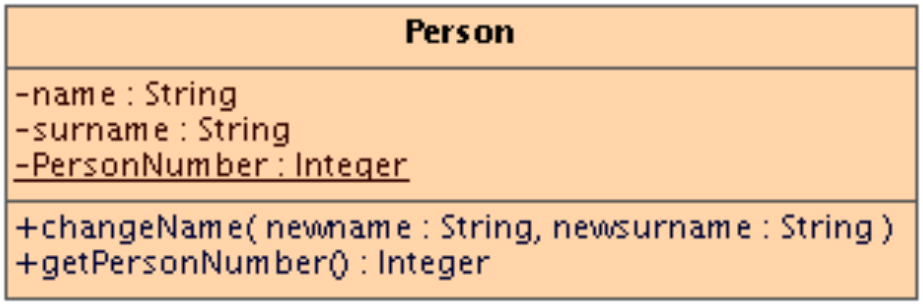
- Base element for object oriented design
  - describe objects in the systems, their classes and their (static) relationships.
  - contain behavioural declarations (Operations) but do not specify their behaviour
- Used also for abstract description (Conceptual Perspective)
- Do not depend on a specific language

Classes



- It is a “thing” in the model
- Defines
  - Name: identifies the class
  - Attributes: describe a property
  - Operations: define actions carried out by the class

Classes: Attributes



visibility name: type multiplicity = default {property-string}

visibility: + (public), - (private), ~(package), #(protected)

name: corresponds to the name of the attribute

type: indicates a restriction on what kind of object can be placed in the attribute

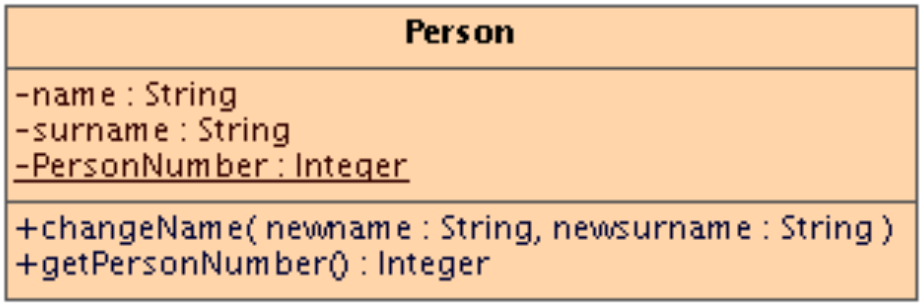
multiplicity: indicates the the cardinality of a property e.g. [0..1], [1] or [*]

default: indicates the default value of the attribute

{property string}: additional properties like {readOnly}

N.B. Underlined attributes indicates static properties

Classes: Operations



visibility name (parameter-list) : return-type {property-string}

visibility: + (public), - (private), ~(package), #(protected)

name: corresponds to the name of the operation

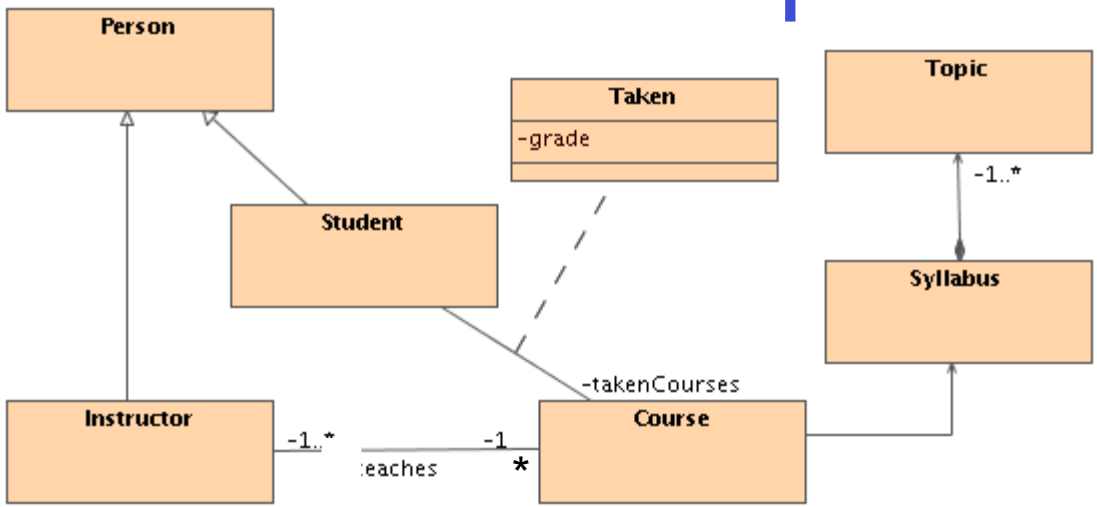
parameter-list: corresponds to the list of parameters for the operation

return-type: type of the returned value (optional)

{property string}: additional properties

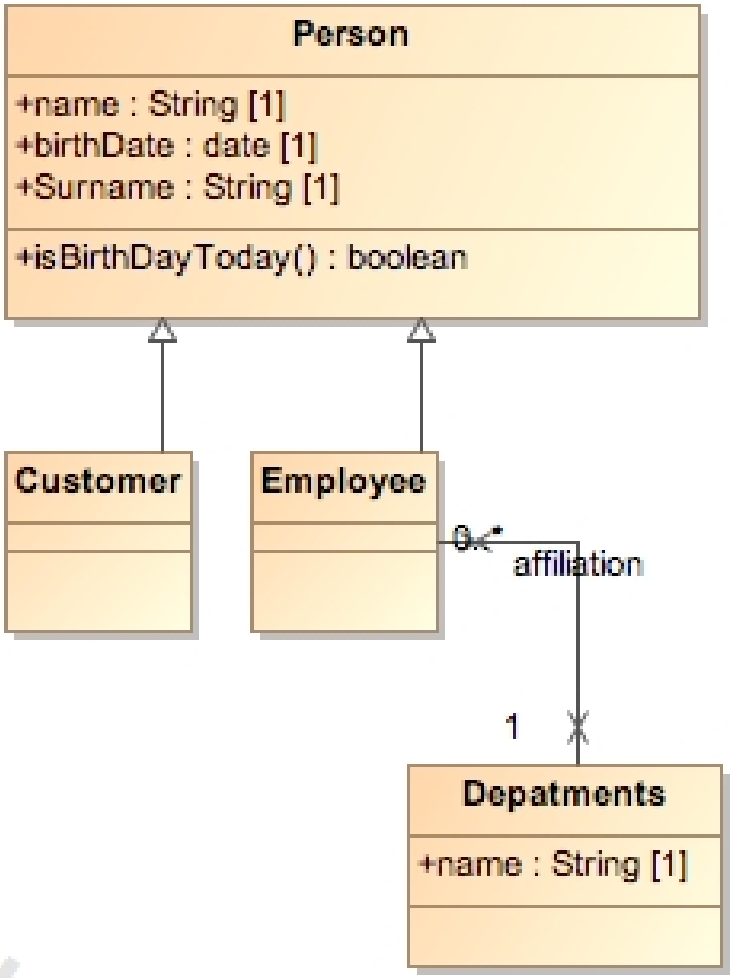
N.B. parameters are specified similarly to properties e.g. newName : String

Relationships



- Generalization:
  - In a software perspective the obvious interpretation is inheritance.
- Association
  - Used to model properties like attributes.
- Other features of class diagrams:
  - Composition/Aggregation
  - Association class

Relationships



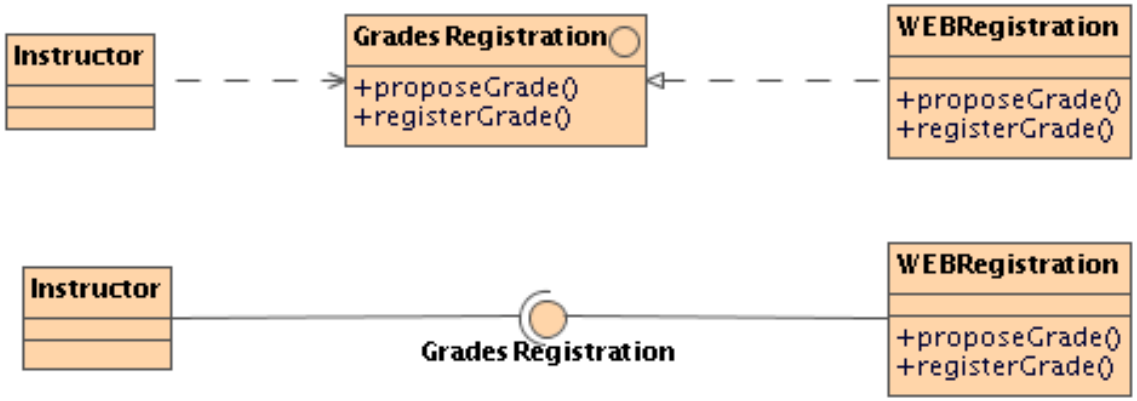
- Example of generalization and association.

Relationships: Composition

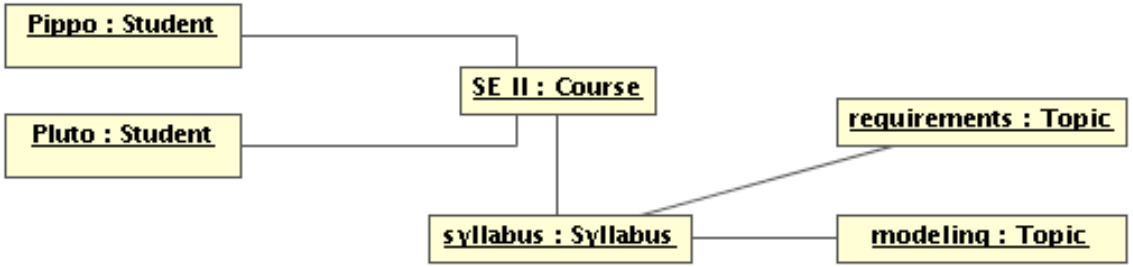
- Sometimes an object is made up of other objects.
- For example, an airplane is made up of a fuselage, wings ….
- You can have recursive composition.
- In UML 2, aggregation would be shown with an open diamond.
- General rule is that, alth ough a class may be a component of many other classes, any instance must be a component of only one owner.
- The class diagram may show multiple classes of potential owners, but any instance has only a single object as its owner
- When the owner of the composition is destroyed, all the composed objects are destroyed as well (for aggregations usually this is not necessarily true).

Interfaces and Objects

- Dashed empty arrows (the one on the left) show dependences between classes while solid arrows show implementation of interfaces.
- Interfaces are defined with a circular symbol inside the class box and the relation between required/provided interfaces is modelled through ball icons (often called “lollipops”).



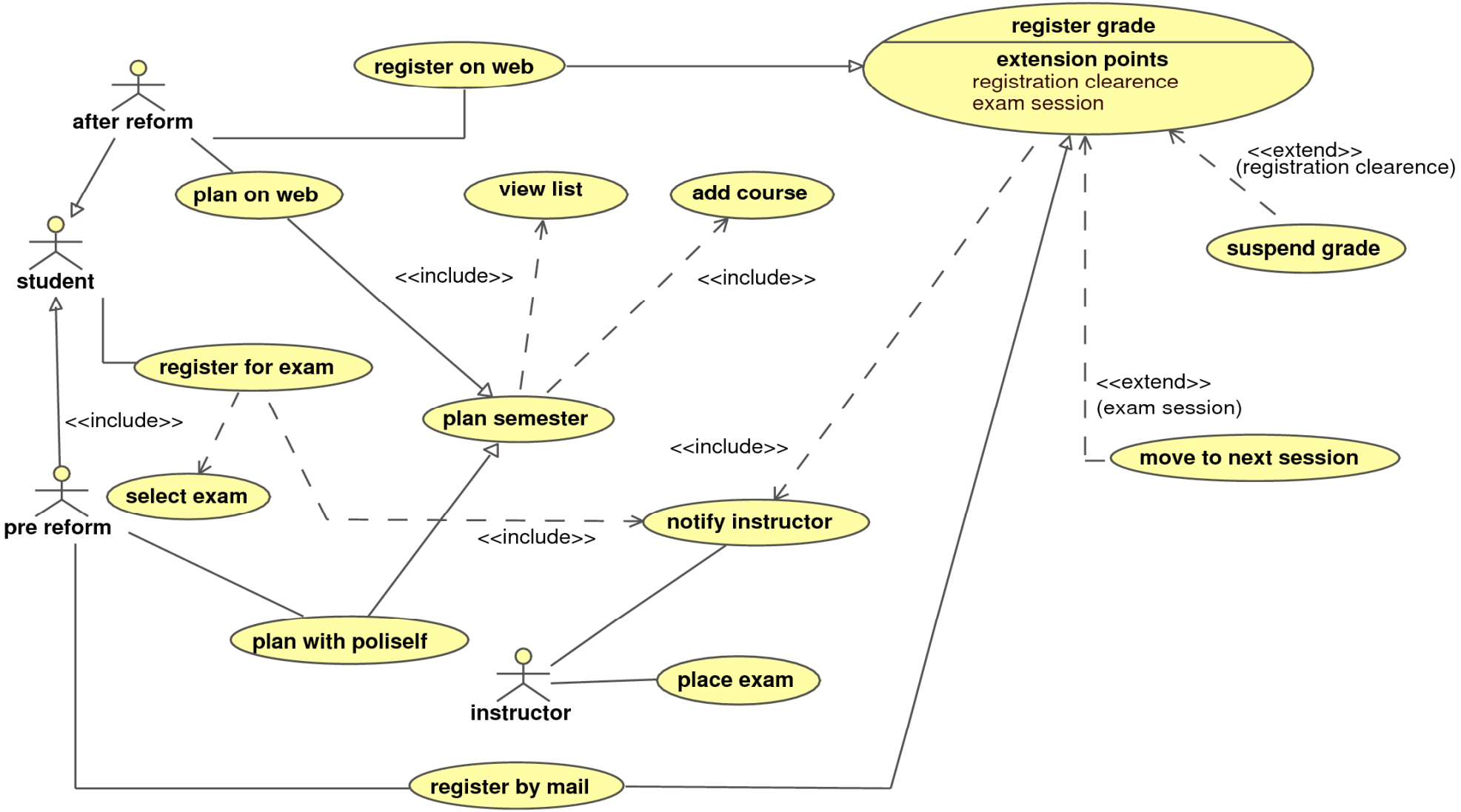
- Object Diagrams: a different notation to model objects, i.e. instances



Use case diagrams

- Capture functional requirements
  - organize requirements into macro-functionalities
  - define relationships between functionalities
  - define role assumed by the user
  - define interaction between user and the system
- Are expressed in terms of the application domain, not in terms of the information system

Use cases notation

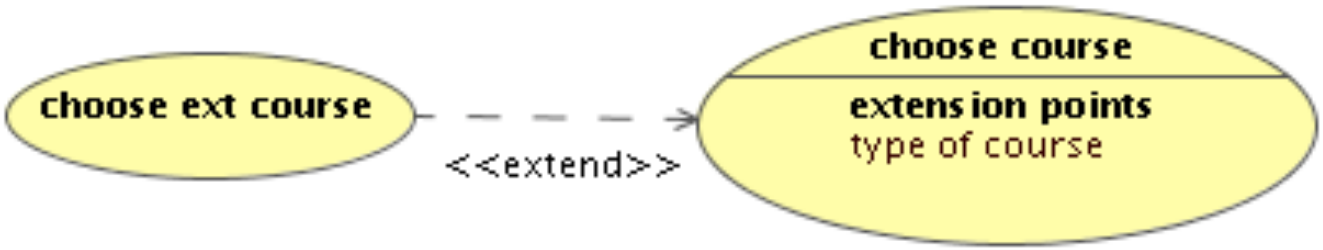


Use case diagrams notation

- Actors
  - represent users of the system
  - Can be related with generalization
  - Can be other systems as well as users
- Use cases
  - represent macro-functionalities of the system
  - extensions, generalization and inclusion
    - generalization: child inherits the behaviour and adds functionalities
    - extension: child adds incremental behaviour to the parent use case

Use case diagrams notation

- Extensions
  - Adds sequences to the base sequence of events in the parent use case
  - Client defines an extension point to be extended
  - Extension use case is not instantiated as a new use case.
  - Use case diagrams must be described by textual use cases!!



Use cases

- Use case diagrams must be described by textual Use Cases!!
- “Use cases, stated simply, allow description of sequences of events that, taken together, lead to a system doing something useful.”
- Use the appropriate degree of detail!!

6.6 Map Series Tool

Use Case Description	
Summary	User generates one or more maps from a series of maps for a given boundary feature (compartment, landscape etc).
Actors	EIMS User
Pre-Conditions	User requires one or more maps sheets from a series, for a boundary feature.
Post-Conditions	Map or series of maps is generated and printed
Priority	Required

Scenario
1) User starts the tool.
System displays a list of map series that the user can select from. Default map series will be ‘Landscape 1:7920’. Can be set at any scale.
2) User selects map series on form.
System then determines if any boundary features are selected. *A. Features Selected:* i. *If features are selected, it asks the user to if they want to generate a map series for the selected feature. Only one feature can used at a time.* *B. No Features Selected:* i. *If no features are selected, or user opts to select the feature manually, the system prompts the user to select the district and compartment of interest from pull downs. It then zooms to that location, generates the map sheet boundaries, draws them with the map sheet names.*
3) User can select individual sheets on screen, or select to print just an index map, or the entire series.
System starts generating and printing maps based on the selected sheets.
4) User collects maps from printer

Notes

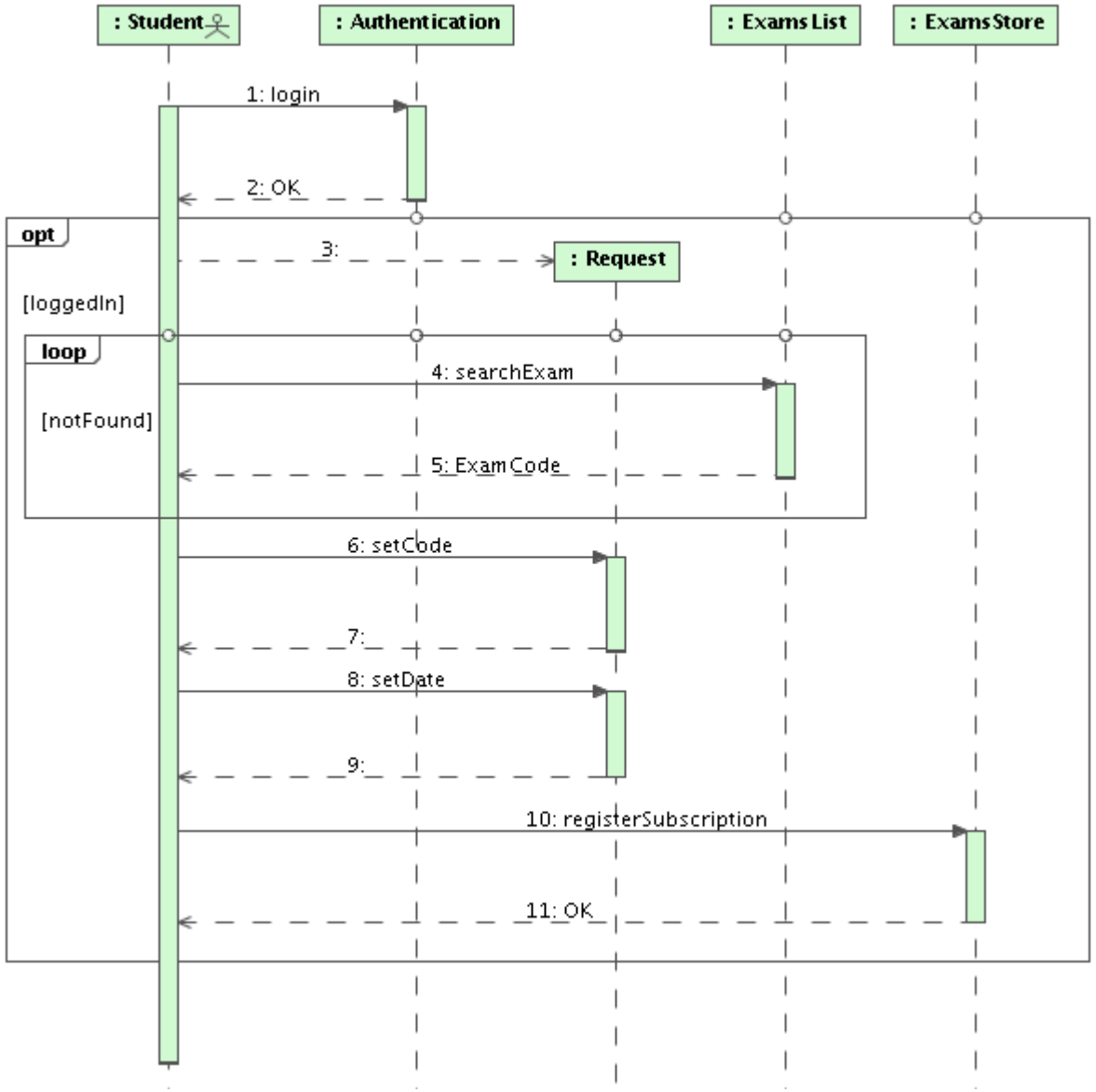
Interaction diagrams

- Describe cooperation between participants from a global perspective
  - Sequence diagrams
- Used for analysis, design, and validation
  - Analysis: used to model event flow in use cases
  - Design: used to identify what participants will be part of the solutions
  - Validations: used to validate specific interactions between chosen participants

Sequence diagrams

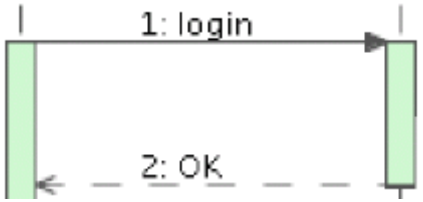
- Bidimensional diagrams:
  - horizontal axis contains participants involved
  - vertical axis represent time
- It describes sequence of messages and not timed intervals
- Every participant has a lifeline
  - messages are exchanged between lifelines
  - participants need not to be objects

Sequence diagram notation



Sequence diagram notation

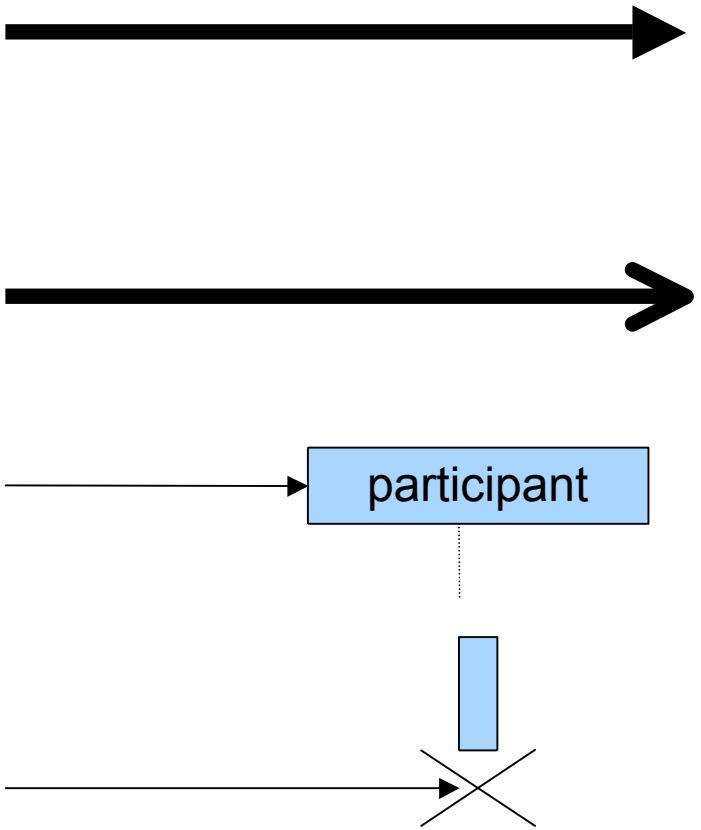
- Participants
  - Can be objects as well as Actors or any classifier in the system



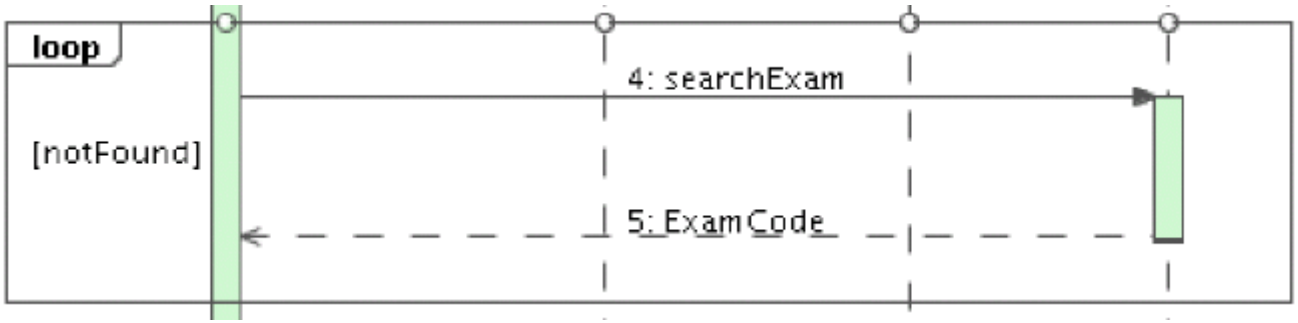
- Synchronous operation call with return value
  - often return message not depicted
  - interaction is depicted with a stronger line on the lifeline

Sequence diagram notation

- Synchronous message
- Asynchronous message
- Creation message
- Destruction message



Sequence diagram notation



- Interaction frames
  - Define areas of the diagram that define control structures.

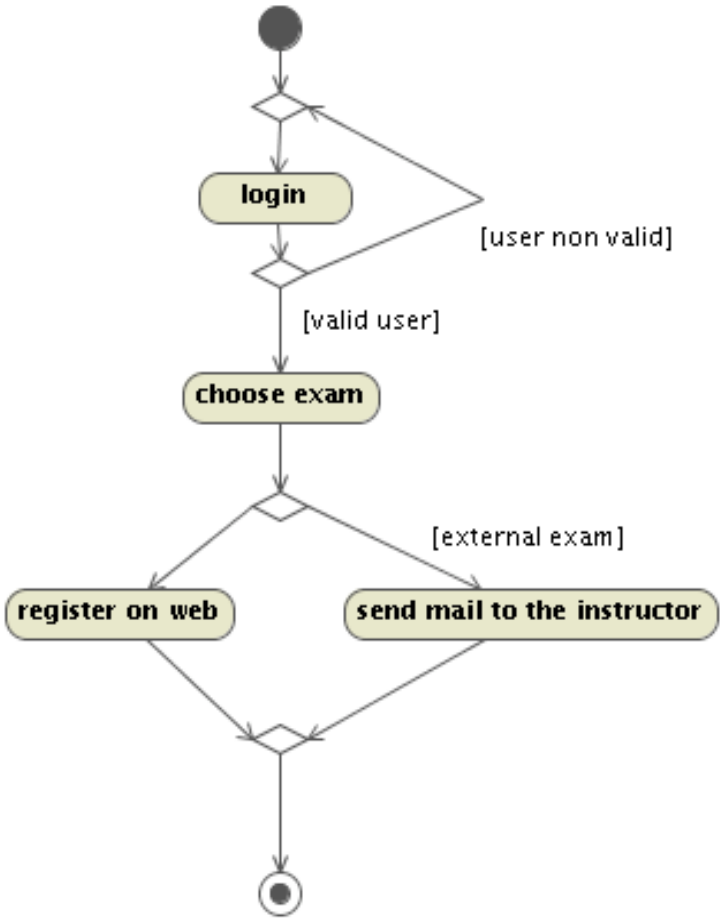
Sequence diagrams notation

- Interaction frames:
  - loop: executed until the condition is true
  - opt: executed if the condition is true
  - alt: executes one frame if the condition is true, and the other if the condition is false
  - ref: includes another sequence diagram
  - par: executes two frames in parallel
  - ....

Activity Diagrams

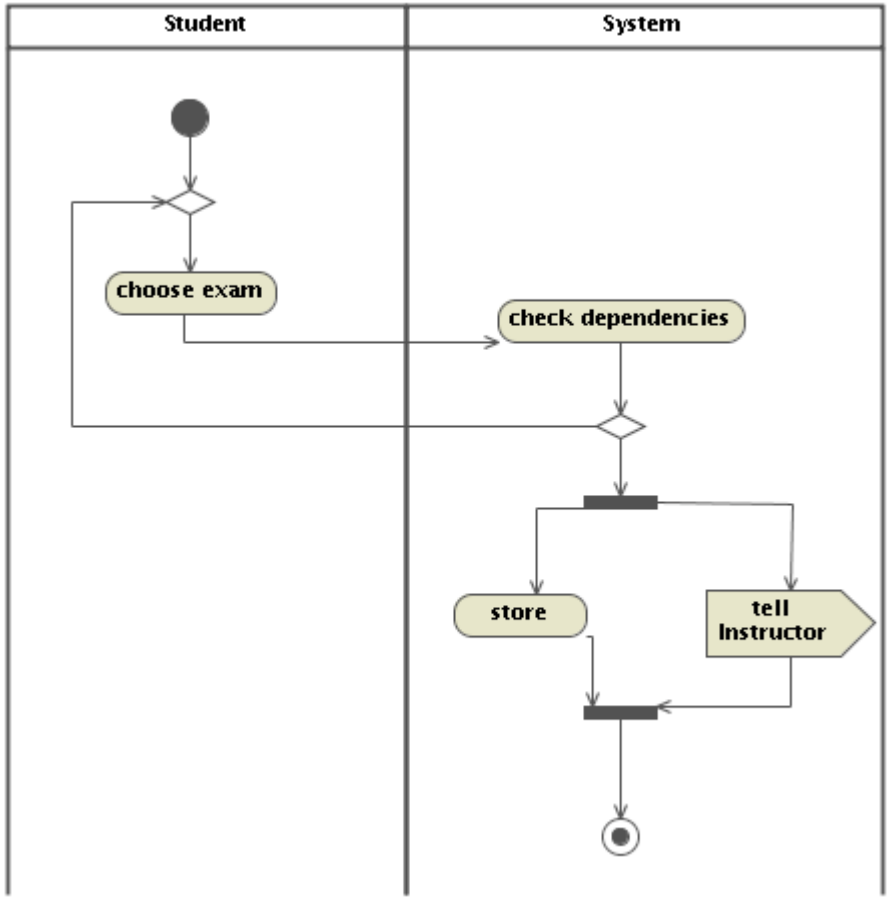
- Show the flow of control of computational activities involved in performing a calculation or a workflow.
  - Can describe both internal flow of control and communication between elements
  - Can describe parallel tasks
  - Can describe data flow between activities and objects

Activity diagram notation



- Actions (boxes) define specific activities
  - several predefined actions divided into categories
- Transitions (arrows), braches and merges (diamonds) define control flow
- transitions can be associated to conditions (in square brackets).

Activity diagrams notation

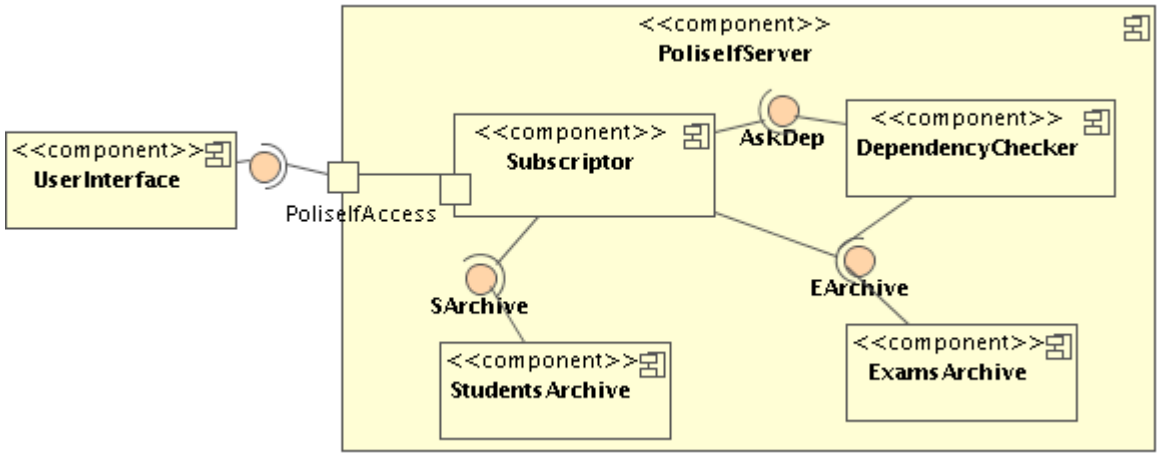


- Vertical divisions (Swim lanes) identify responsibilities in the workflow
- fork and join (black boxes) allow to define parallel compositions
  - join policy can be specified
- Signals for interaction can be modelled (arrow shaped boxes)
  - send signal: SignalSendState1
  - accept signal: SignalAcceptState1

Components

- A modular part of a system that hides its implementation behind a set of external interfaces
  - Components that satisfy the same interfaces can be substituted
  - Distinction between components and classes is blurred
  - A component can disappear in the implementation (implemented by a series of classes)
  - Hierarchically decomposed in other components
  - Viewed at different levels of abstraction

Components diagrams

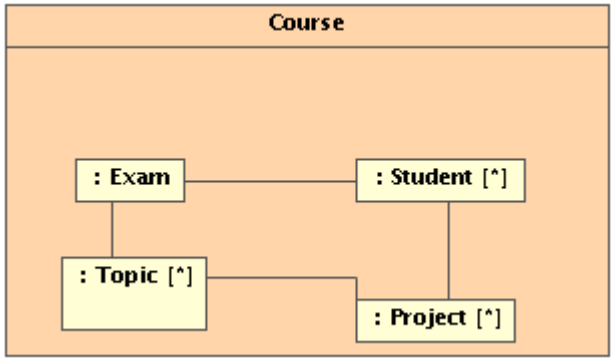


- Every component logically organizes interfaces in ports.
- Every component specifies needed and provided interfaces
- Interfaces and delegation wire the components together.

Composite structure

- Structured class: class defined not only in term of operations and state but with its decomposition into parts
  - defines ports
  - defines internal parts: role that an instance plays inside an instance of a structured class
  - defines connectors
  - defines parts multiplicity

Composite structure



- Defines for every part a multiplicity and the relationships with roles assumed by other parts

Statecharts

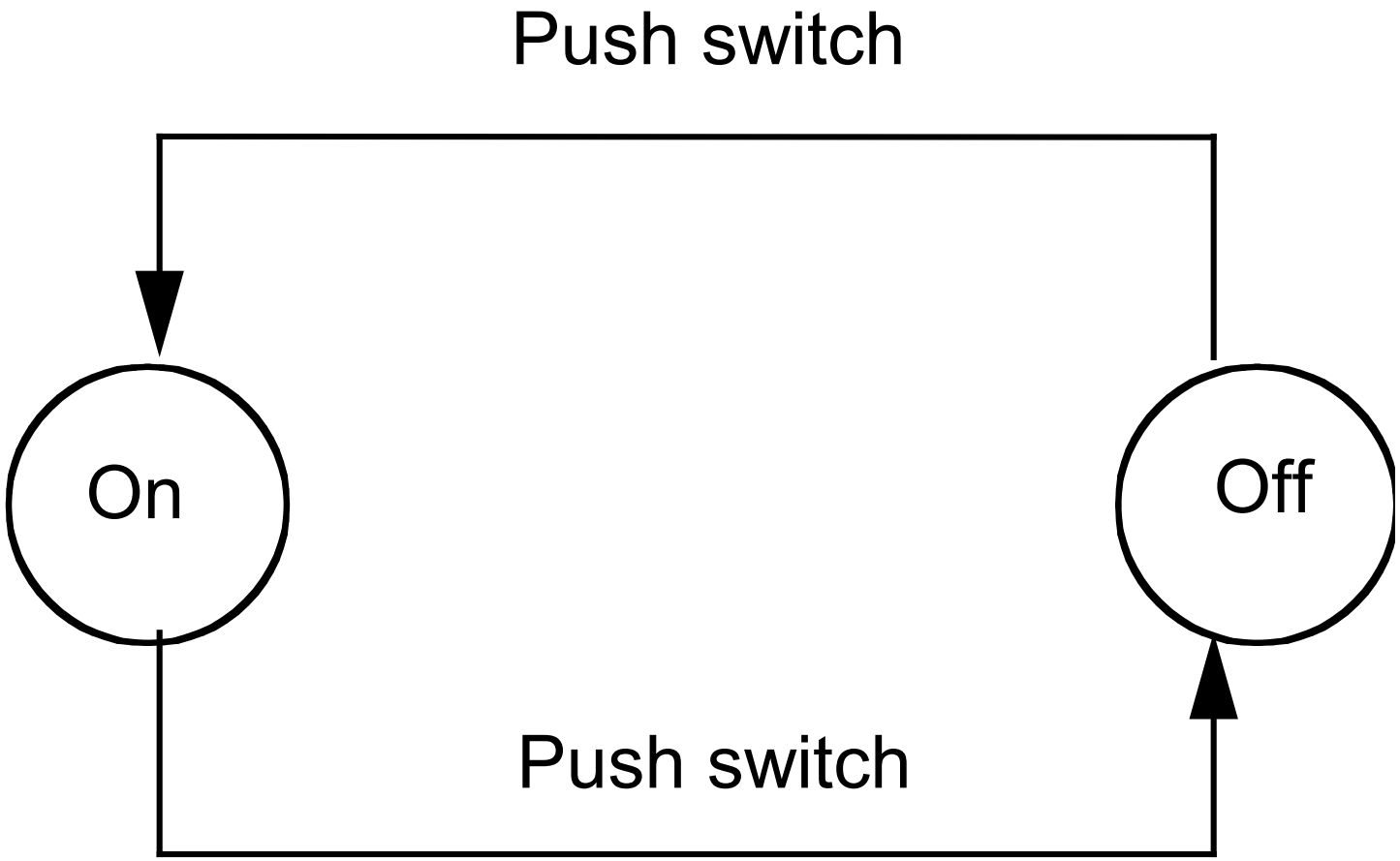
- Variation of Harel's statecharts
- Description of the behavior of the system
  - state based desc. for reactive systems
  - describes elements in isolation with respect to interaction diagrams
  - A statechart is defined for every class of object (providing the behaviour of that class is interesting)
  - Many semantics have been defined
- Can describe dynamic behaviour of classes, use cases, collaborations,....

Basics on finite state machines

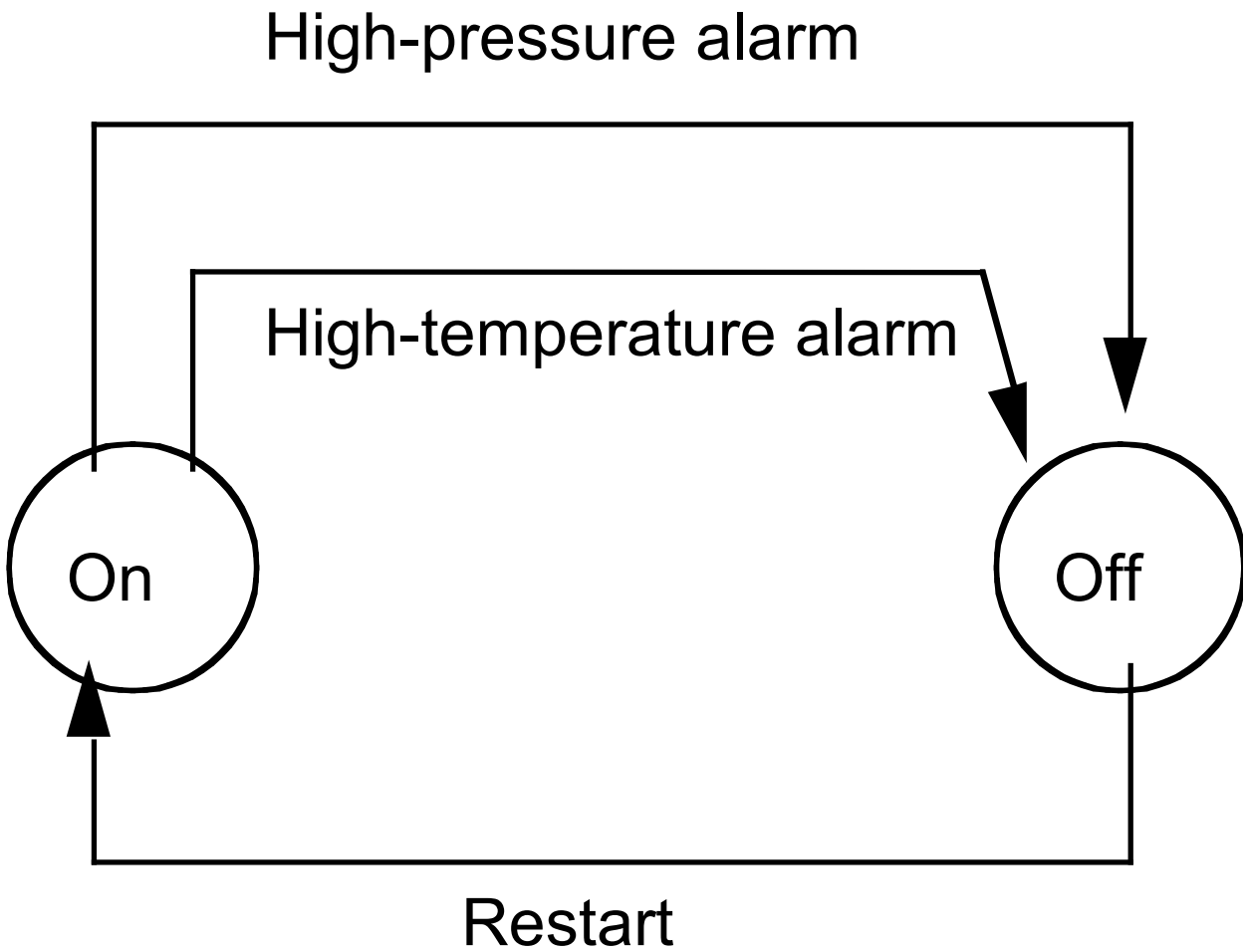
- They are the typical example of an operational specification notation
  - S a finite set of states
  - I a finite set of inputs
  - δ a state transition function
- Appealing graphical representation
- "Animation" straightforward

An example

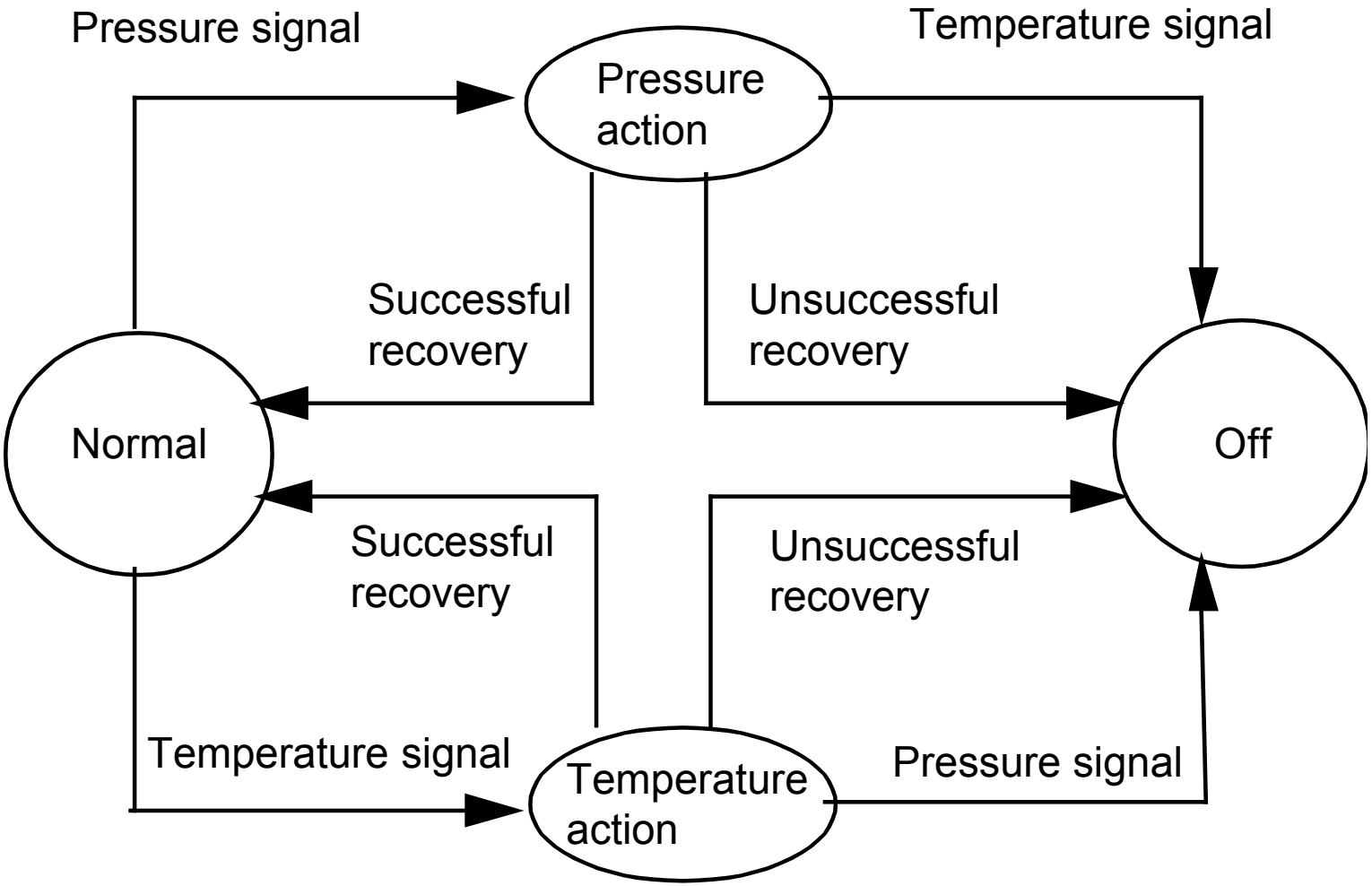
- Behavior of a lamp



Another example: a plant control system



A refinement



Classes of FSMs

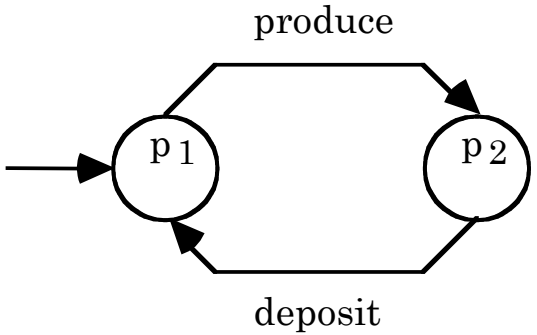
- Deterministic/nondeterministic
- FSM as recognizers
- FSM as transducers
- . . .

Limitations

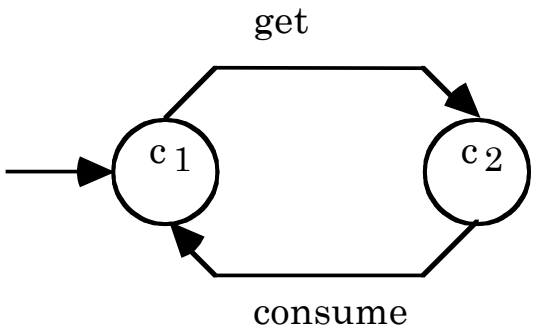
- Finite memory
- State explosion
  - Given a number of FSMs with k_1, k_2, … k_n states, their composition is a FSM with $k_1 * k_2 * \ldots * k_n$. This growth is exponential with the number of FSMs, not linear (we would like it to be $k_1 + k_2 + \ldots + k_n$)

State explosion: an example

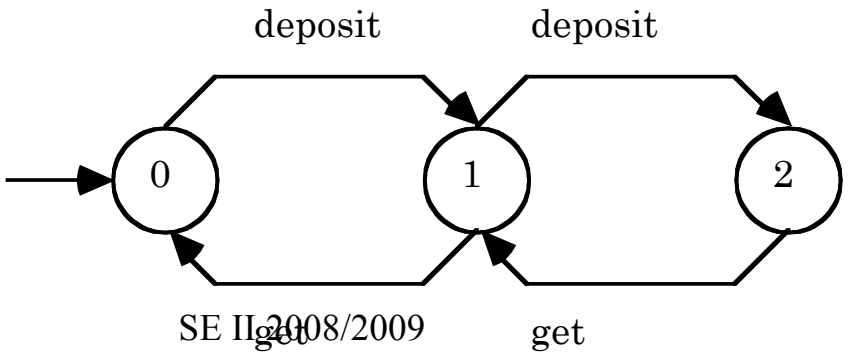
Producer



Consumer

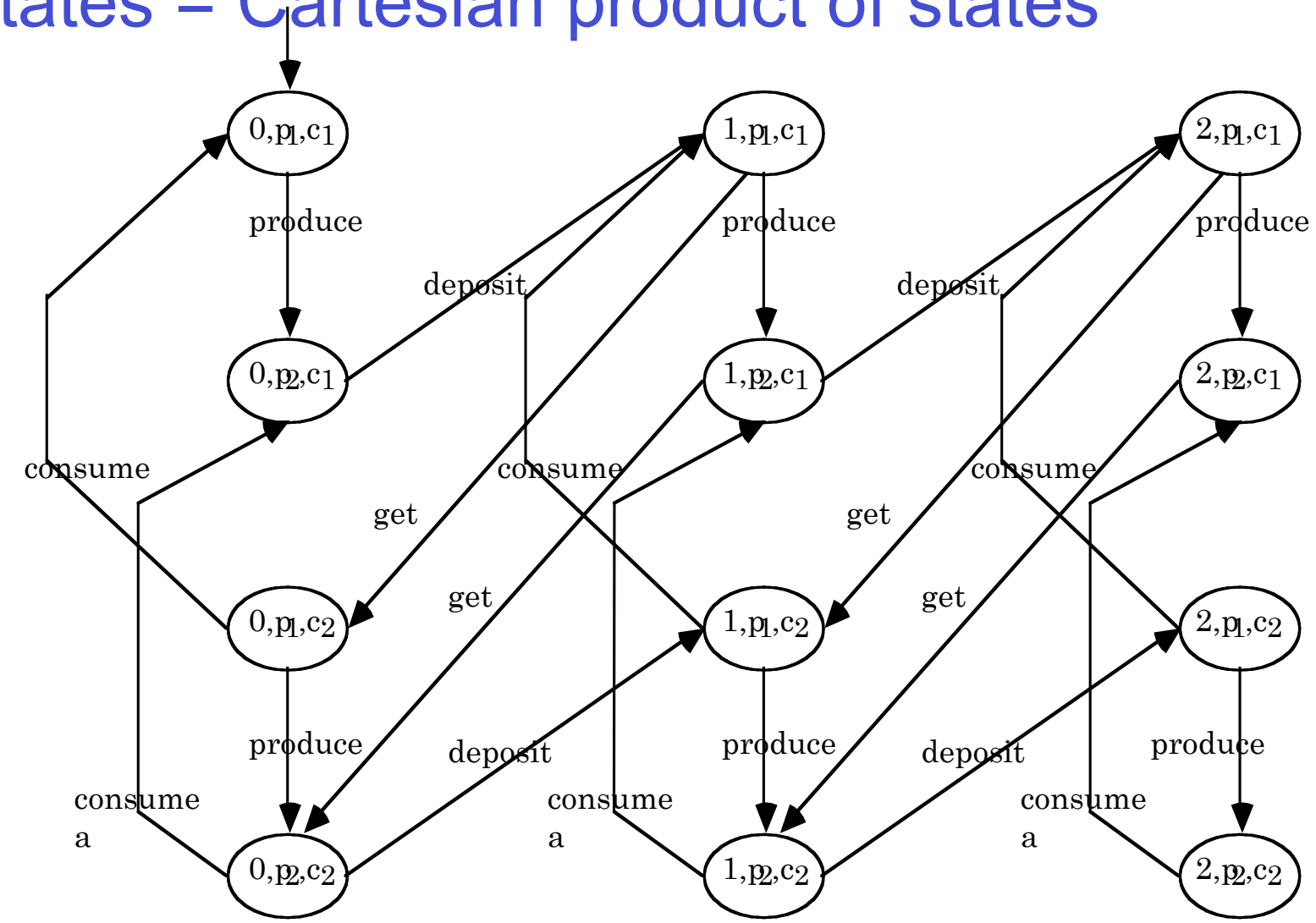


Storage



How to combine into an FSM?

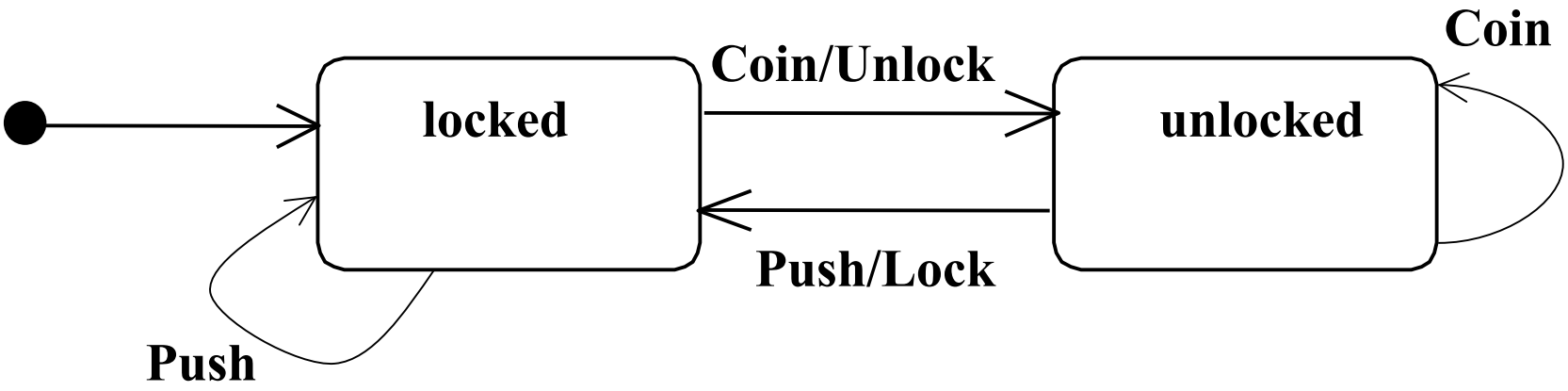
States = Cartesian product of states



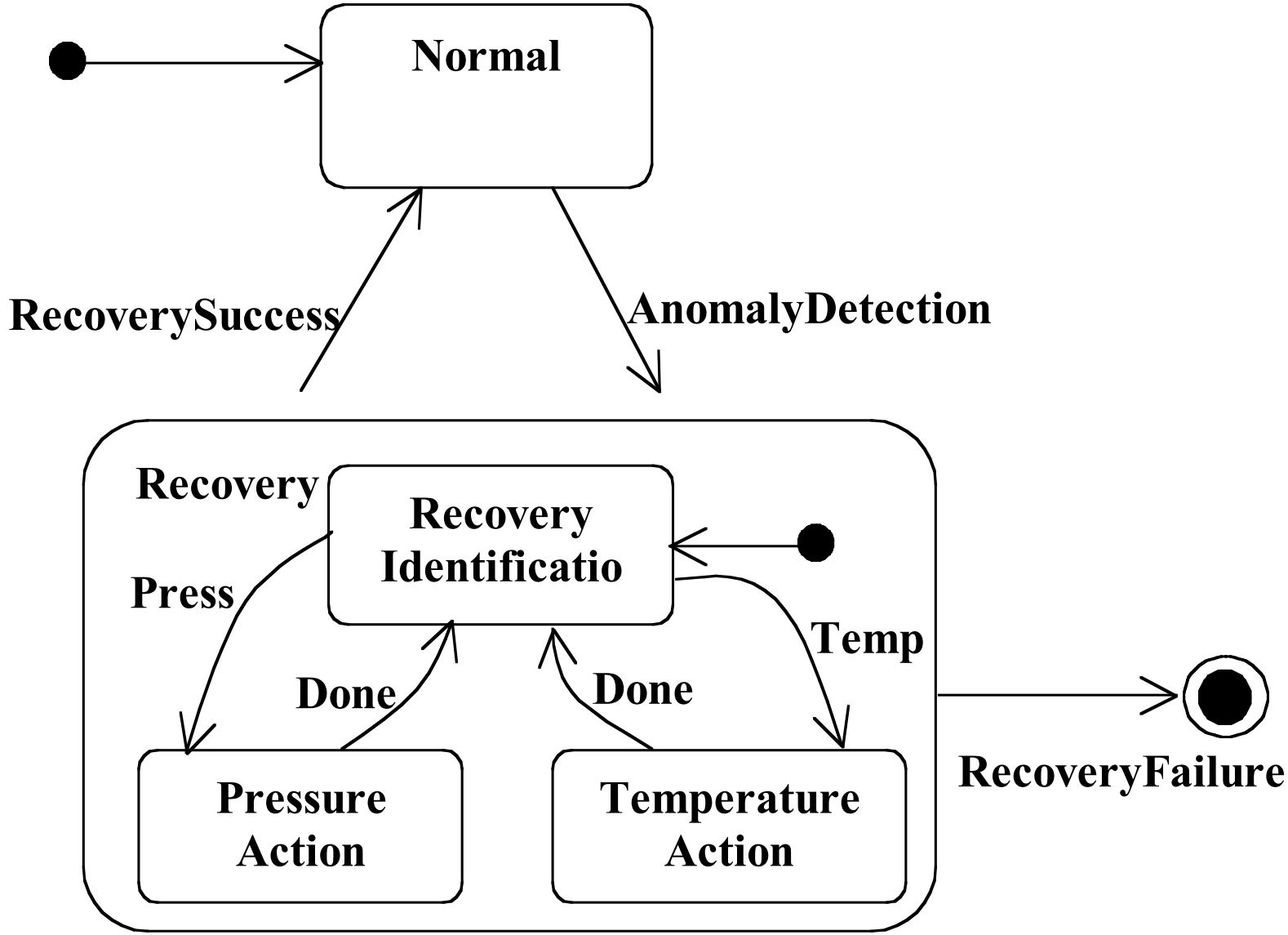
How to solve these problems?

- Statecharts: A modular hierarchical notation for automata
  - cooperating finite state machines
  - used in UML as state diagram
- Petri nets

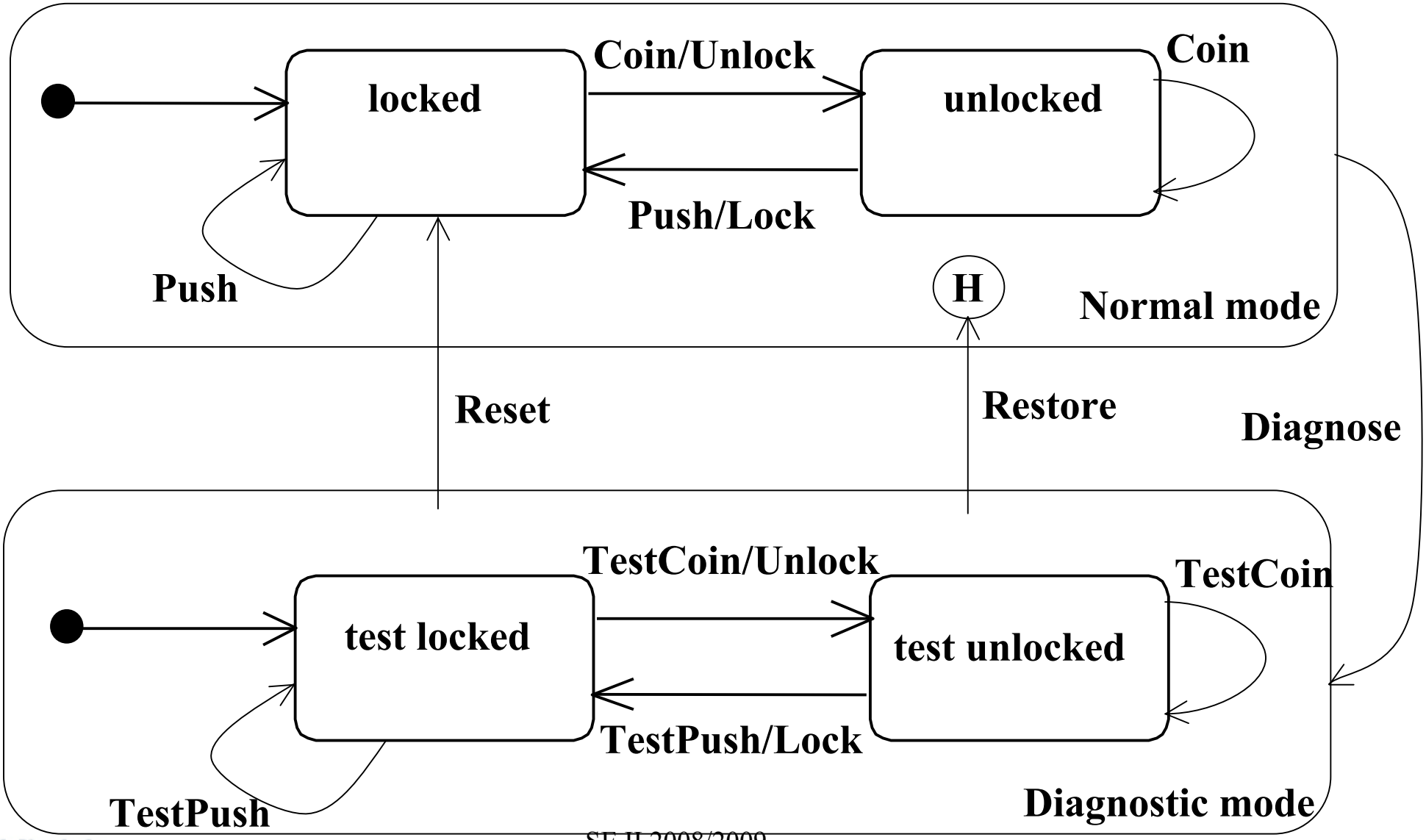
Statecharts: events and actions (turnstile example)



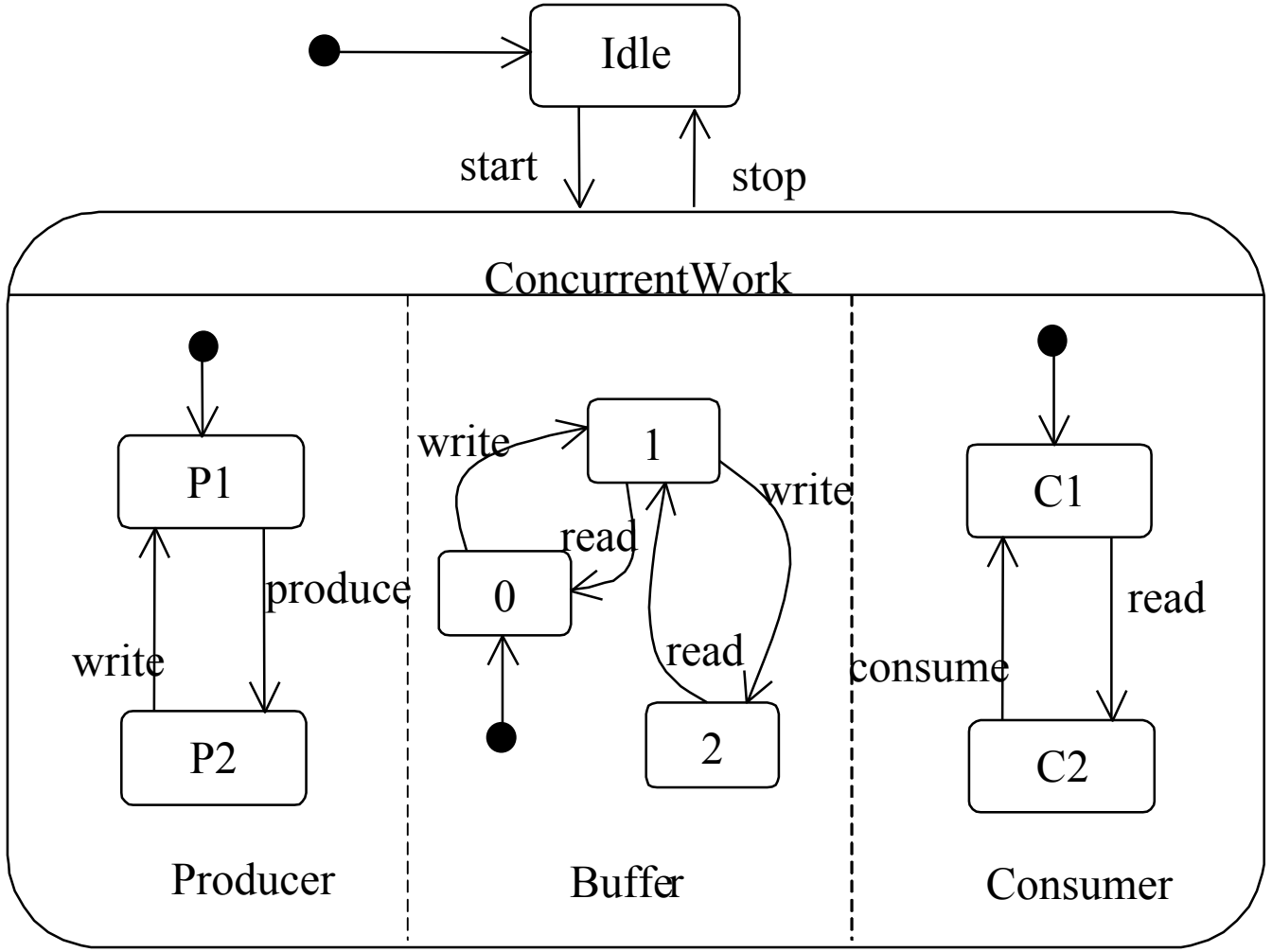
Plant control system: substates



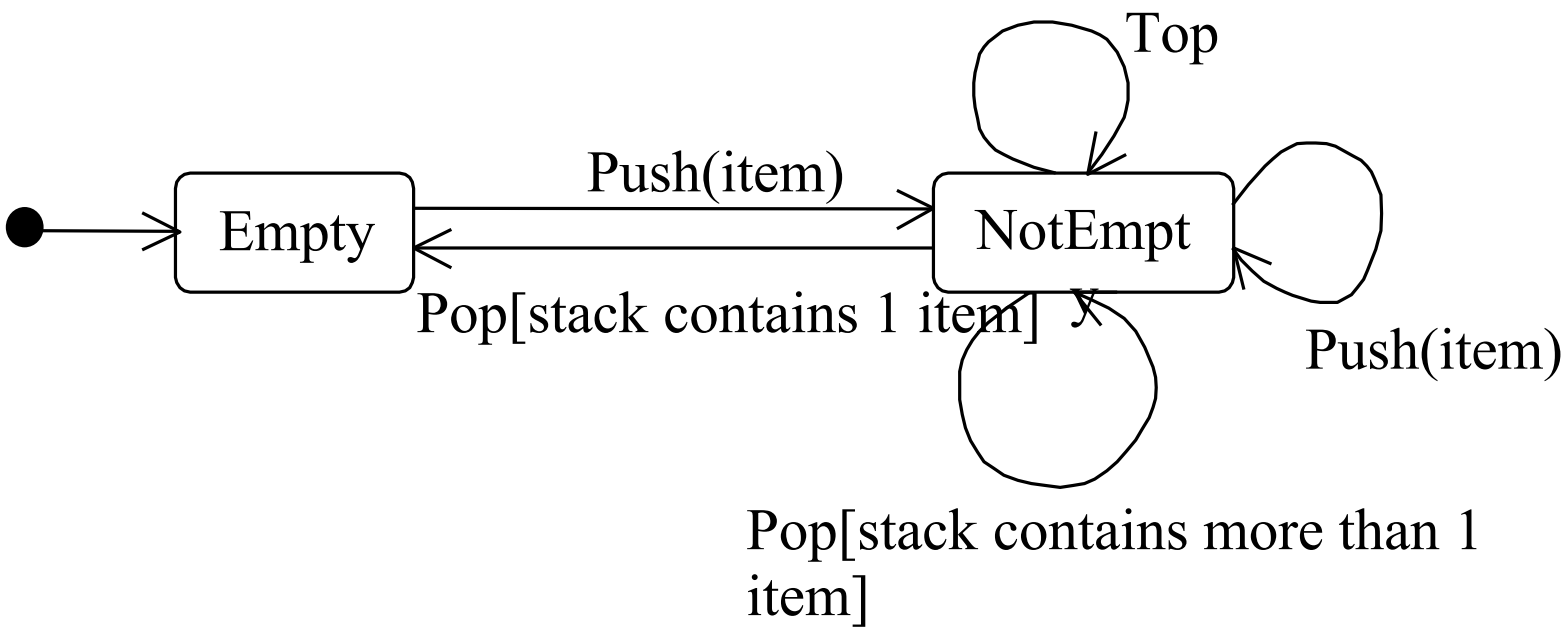
Turnstile II ver: history



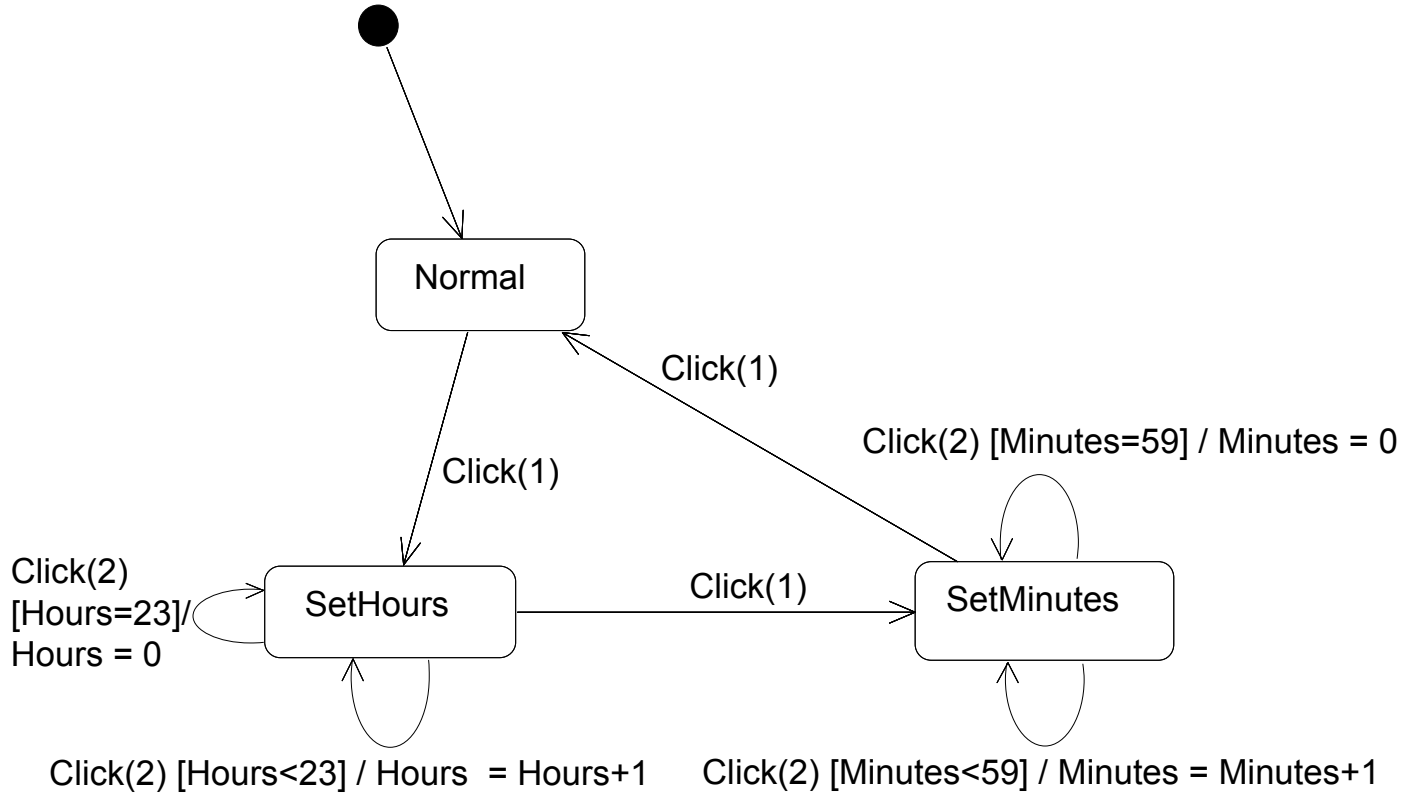
Concurrent substates



State specification of abstract object: conditions



Digital clock: events, conditions, and actions



Event conditions and actions

`Event [Condition] / Action`

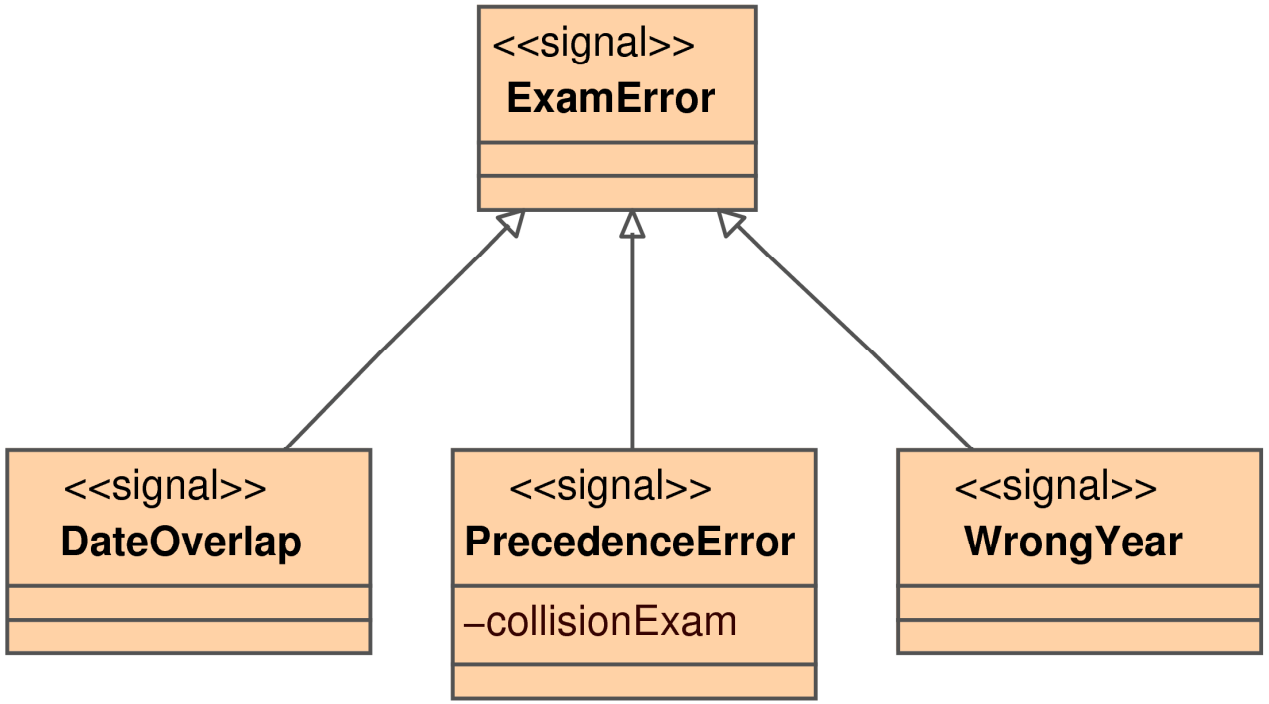
- None of the parts is mandatory
- Event:
  - occurs at a point in time (has no duration)
  - may have parameters
  - Taxonomy:
    - call event: receipt of a synchronous request by an object
    - change event: change in the boolean value of an expr.
    - signal event: receipt of an asynchronous communication
    - time event: arrival of an absolute time

Event conditions and actions

`Event [Condition] / Action`

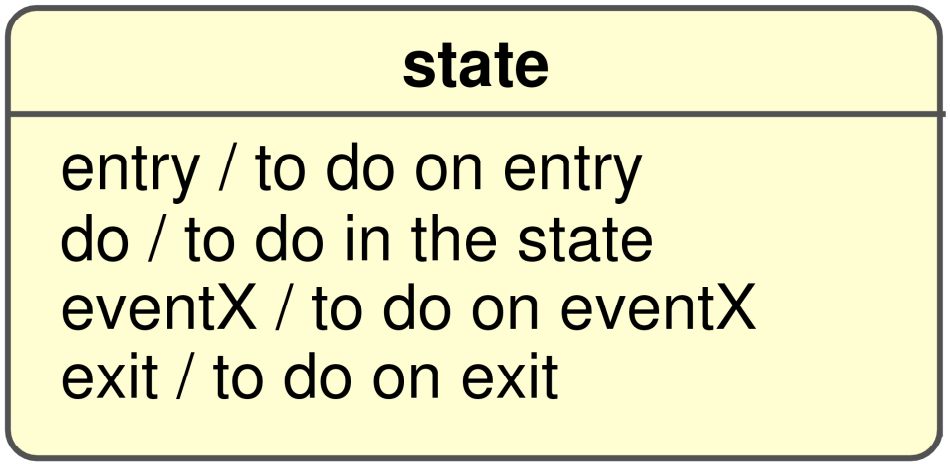
- Action: activity executed during the change of the state.
  - different semantics in different notations
  - execution of a simple action
  - execution of a complex action modeled with other behavioral notations
  - sending of a broadcast asynchronous signal (with SEND prefix)
  - sometimes described informally

Signals



- Defined as classes stereotyped with “signal”
  - Generalizable
  - Any transition that is fired by a signal event is also fired by all its subclasses

More on states

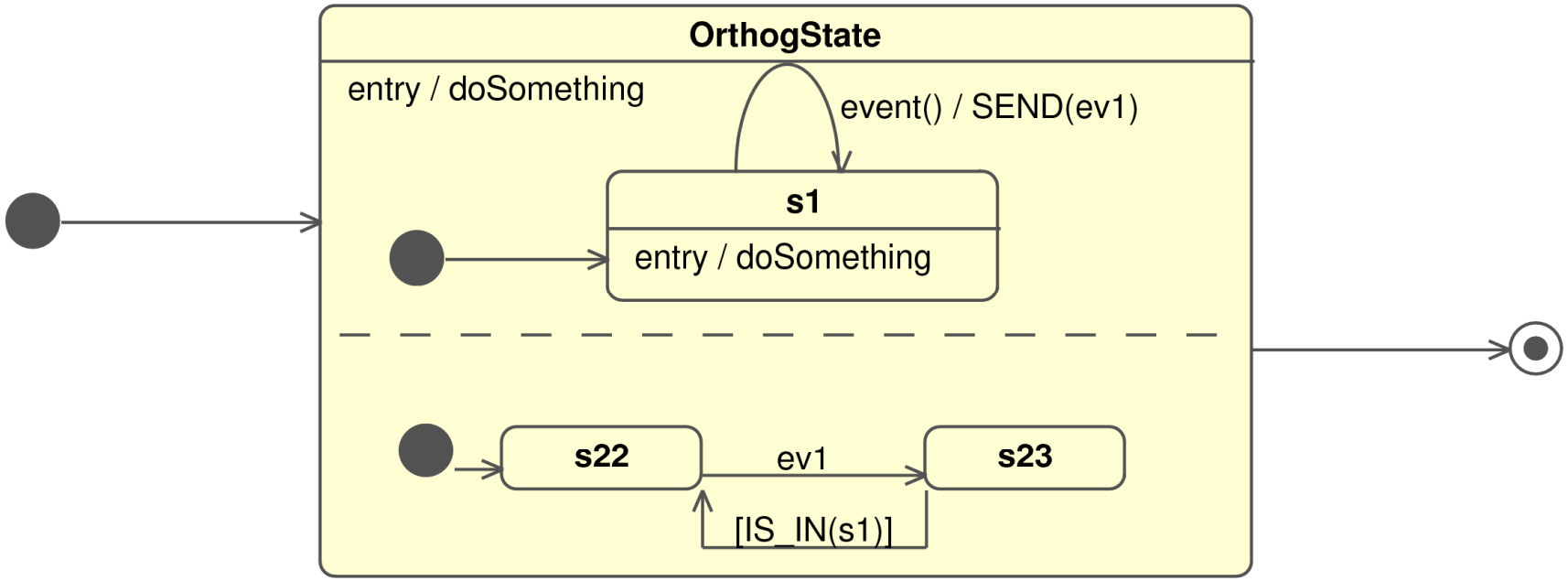


- Internal transitions:
  - transitions that do not change the state of the systems:
  - on entry, exit, specific events.
  - Can define the main behaviour of the state (do/)
  - event, condition and action work exactly as for ordinary transitions

More on states

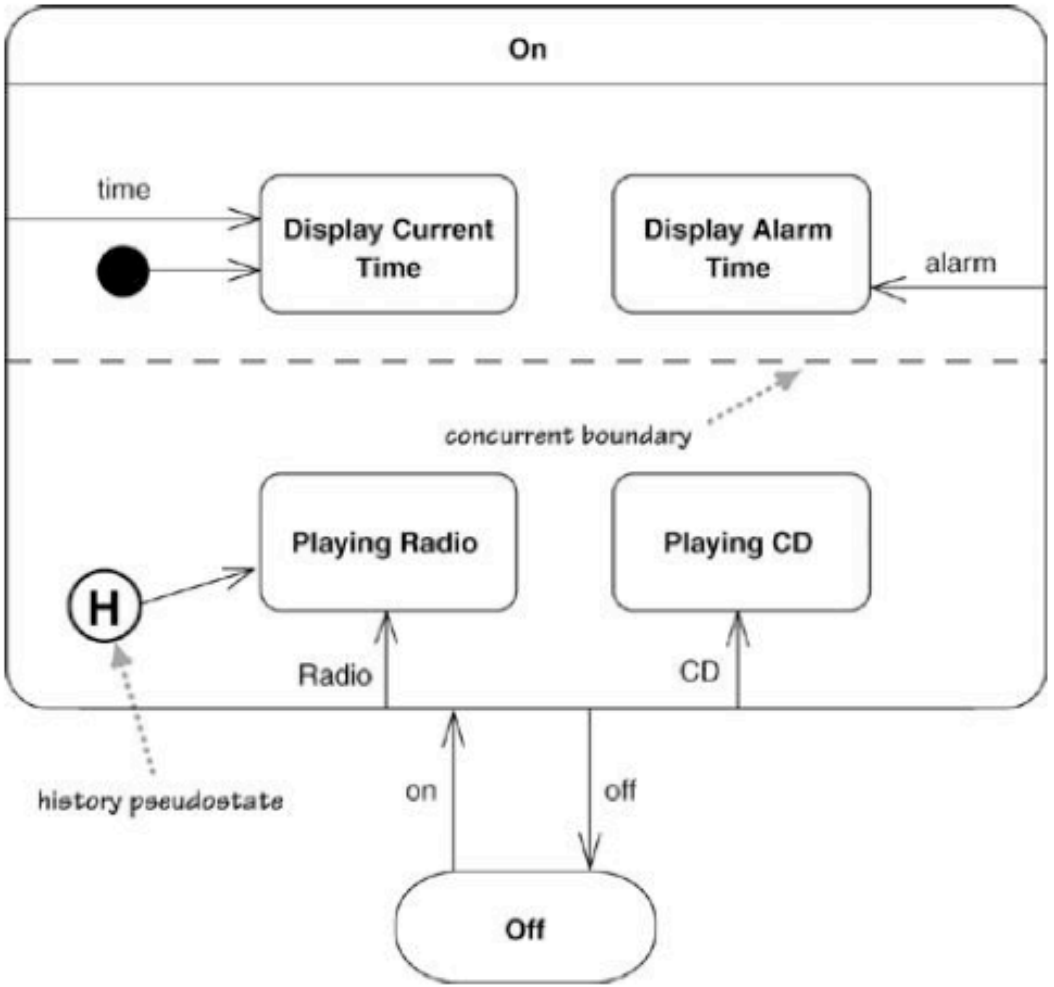
- Internal transitions
  - actions executed in a predefined order:
    - exit actions of the source states(s)
    - transition actions
    - entry actions of the target state(s)
- Different semantics of transitions:
  - concurrent transitions

Concurrency and synchronization



- Event broadcast
- IS_IN

Radio CD example



References

- *Boch, Jacobson, Rumbaugh, The Unified Modeling Language User Guide, 2nd Edition. Addison Wesley*
- *Baresi, Lavazza, Pianciamore, Dall'idea al codice con UML2, Pearson*
- *Bruegge, Dutoit, Object-Oriented Software Engineering, Pearson*
- *Fowler,UML Distilled 2nd Edition, Addison Wesley*
- *Boch, Jacobson, Rumbaugh, The Unified Modeling Language Reference, 2nd Edition. Addison Wesley*