

INF5120  
"Modellbasert Systemutvikling"  
"Modelbased System development"

Lecture 8: 06.03.2017

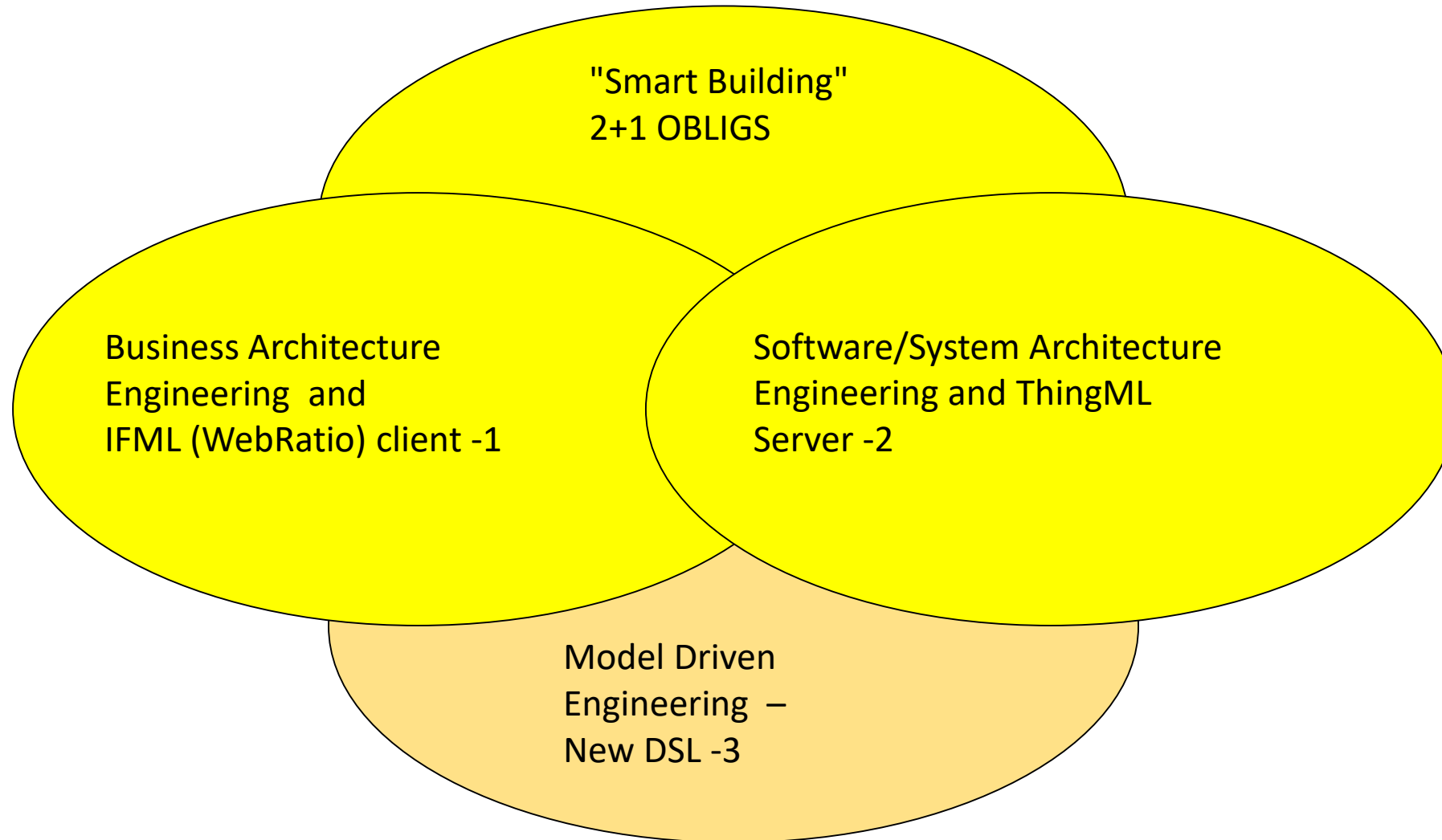
**Arne-Jørgen Berre**

[arneb@ifi.uio.no](mailto:arneb@ifi.uio.no) or [Arne.J.Berre@sintef.no](mailto:Arne.J.Berre@sintef.no)

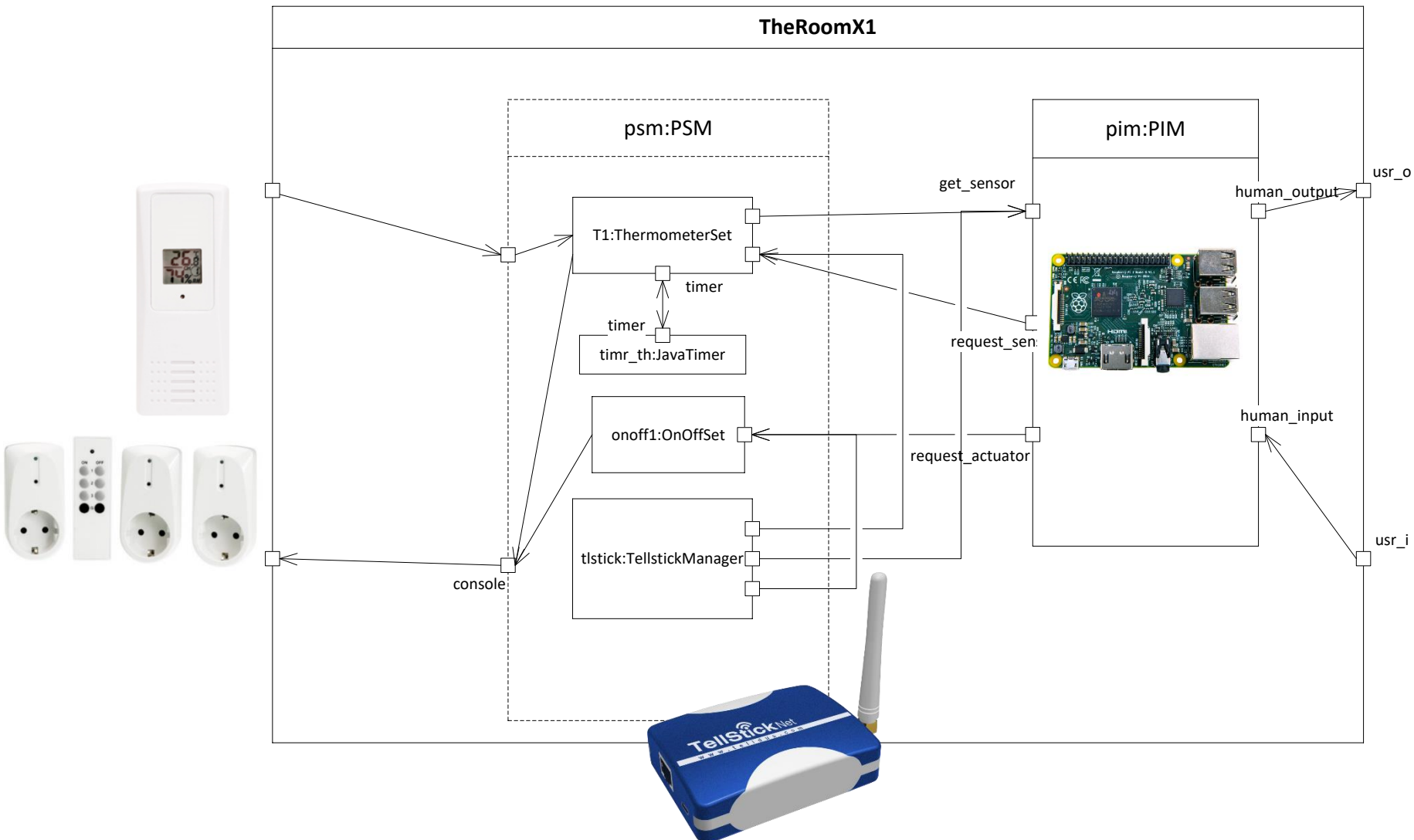
# Course parts (16 lectures) - 2017

- January (1-3) (Introduction to Modeling, Business Architecture and the Smart Building project):
- 1-16/1: Introduction to INF5120
- 2-23/1: Modeling structure and behaviour (UML and UML 2.0 and metamodeling) - (establish Oblig groups)
- 3-30/1: WebRatio for Web Apps/Portals and Mobile Apps – and Entity/Class modeling – (Getting started with WebRatio)
  
- February (4-7) (Modeling of User Interfaces, Flows and Data model diagrams, Apps/Web Portals - IFML/Client-Side):
- 4-6/2: Business Model Canvas, Value Proposition, Lean Canvas and Essence
- 5-13/2: IFML – Interaction Flow Modeling Language, WebRatio advanced – for Web and Apps
- 6-20/2: BPMN process, UML Activ.Diagrams, Workflow and Orchestration modelling value networks
- 7-27/2: Modeling principles – Quality in Models
- 27/2: Oblig 1: Smart Building – Business Architecture and App/Portal with IFML WebRatio UI for Smart Building
  
- March (8-11) (Modeling of IoT/CPS/Cloud, Services and Big Data – UML SM/SD/Collab, ThingML Server-Side):
- 8-6/3: Basis for DSL and ThingML -> UML State Machines and Sequence Diagrams
- 9-13/3: ThingML DSL - UML Composite structures, State Machines and Sequence Diagrams II
- 10-20/3: Guest lecture, "Experience with Modelling", Anton Landmark, SINTEF
- 11-27/3: ThingML and UML Service Modeling, Architectural models, SoaML. Role modeling and UML Collaboration diagrams
  
- April/May (12-14) (MDE – Creating Your own Domain Specific Language):
- 12-3/4: Model driven engineering – Metamodels, DSL, UML Profiles, EMF, Sirius Editors
- 3/4: Oblig 2: Smart Building – Internet of Things control with ThingML – Raspberry Pi, Wireless sensors (temperature, humidity), actuators (power control)
  
- EASTER – 10/4 og 17/4
- 13-24/4: MDE transformations, Non Functional requirements
- 1. Mai – Official holiday
- 14-8/5: SmartBuilding – Integrating App with Server side
- 8/5: Oblig 3 - Your own Domain Specific Language
  
- May (15-17): (Bringing it together)
- 15-15/5: Summary of the course – Final demonstrations
- 16-22/5: Previous exams – group collaborations (No lecture)
- 17-29/5: Conclusions, Preparations for the Exam by old exams
- June (Exam)
- 13/6: Exam (4 hours), June 13<sup>th</sup>, 0900-1300

# Course components



# Smart Building – server side



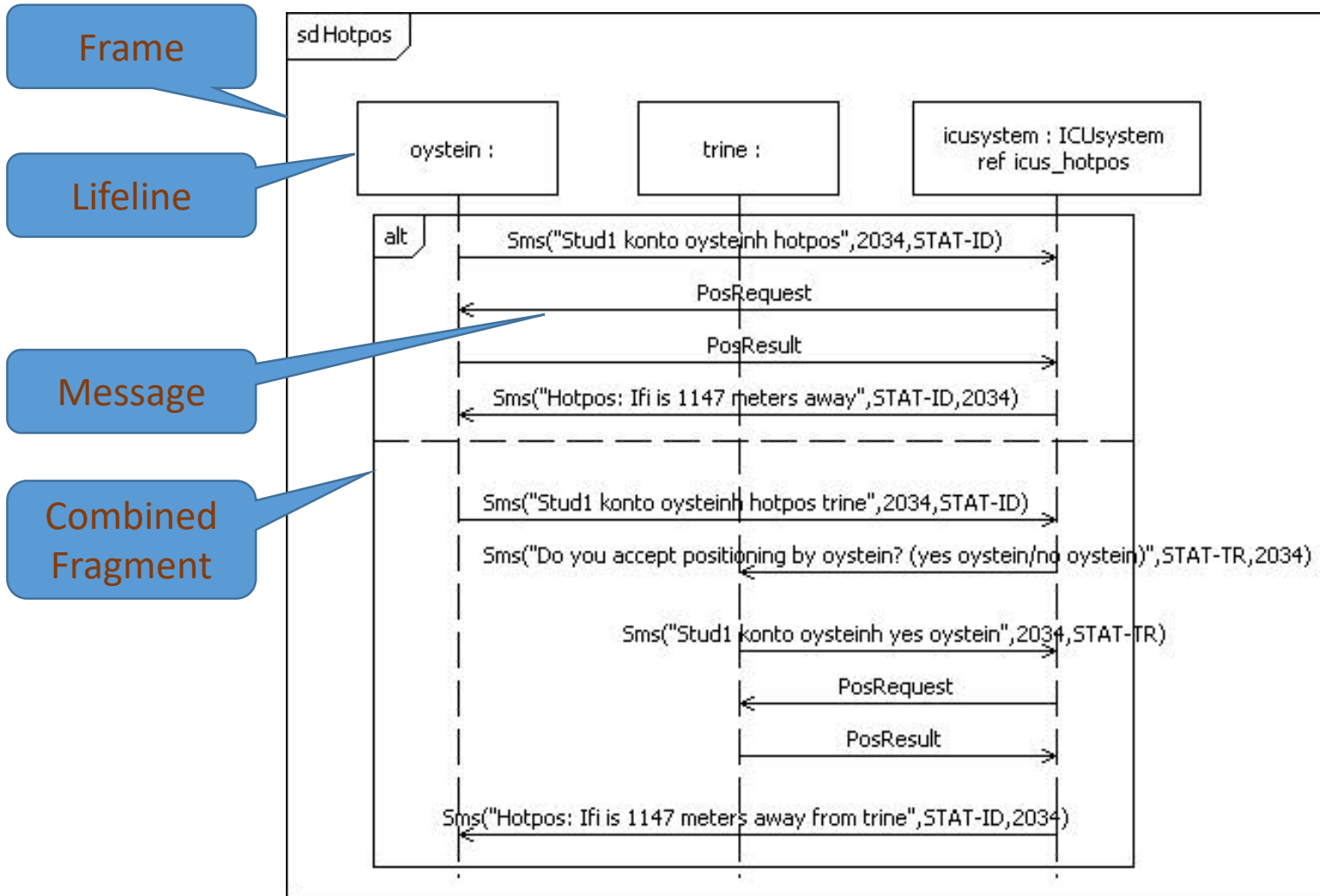
Using ThingML Domain Specific Modeling Language

- Related to UML Sequence Diagrams and State Machines

# Overview of lecture – Sequence Diagram

- Sequence Diagrams
  - What are they intended for?
  - Where in the software engineering process are they used?
- The History Lesson
  - a very short history this time
- Basic sequence diagrams
- Interaction Fragments – structuring mechanisms
- Tooling
  - Sequence Diagrams in Papyrus
  - Interactions or Sequence Diagrams?
  - Experiences and challenges
- Interaction Metamodel

# This is a Sequence Diagram



Frame

Lifeline

Message

Combined  
Fragment

# Sequence Diagrams in a nutshell

- Sequence Diagrams are
  - simple
  - powerful
  - readable
- Emphasizes the interaction between objects when interplay is the most important aspect
  - Often only a small portion of the total variety of behavior is described improve the individual understanding of an interaction problem
- Sequence Diagrams are used to ...
  - document protocol situations,
  - illustrate behavior situations,
  - verify interaction properties relative to a specification,
  - describe test cases,
  - document simulation traces.

# Sequence Diagrams History

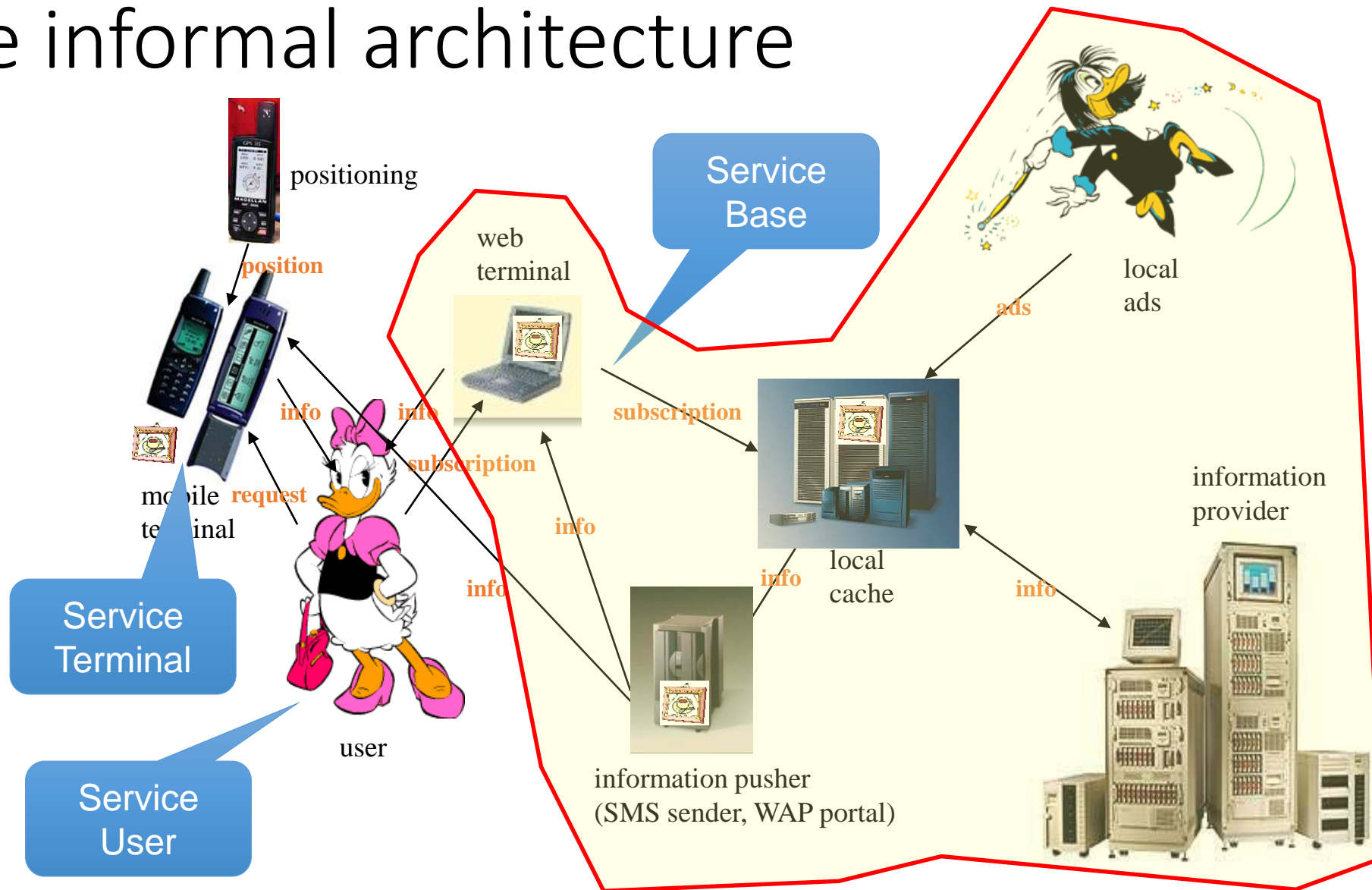
- Used informally for a number of years prior to 1990
- Standardized in 1992 in Z.120
  - Message Sequence Charts – MSC
  - Initiated by Ekkart Rudolph (Siemens) and Jens Grabowski (now Professor in Göttingen)
- Last major revision of MSC is from 1999
  - called MSC-2000 with Ø. Haugen as Rapporteur, representing Ericsson
- Formal semantics of MSC-96 is given in Z.120 Annex B
  - Sjouke Mauw (now Prof at Univ. of Luxembourg) and Michel Reniers (now Assoc. Prof. at Univ. of Eindhoven)
- Included in UML 1 from 1999
  - but in another variant also pioneered by Siemens
- Most of MSC was included in UML 2.0 (2003)
  - Responsible Ø. Haugen (representing Ericsson)



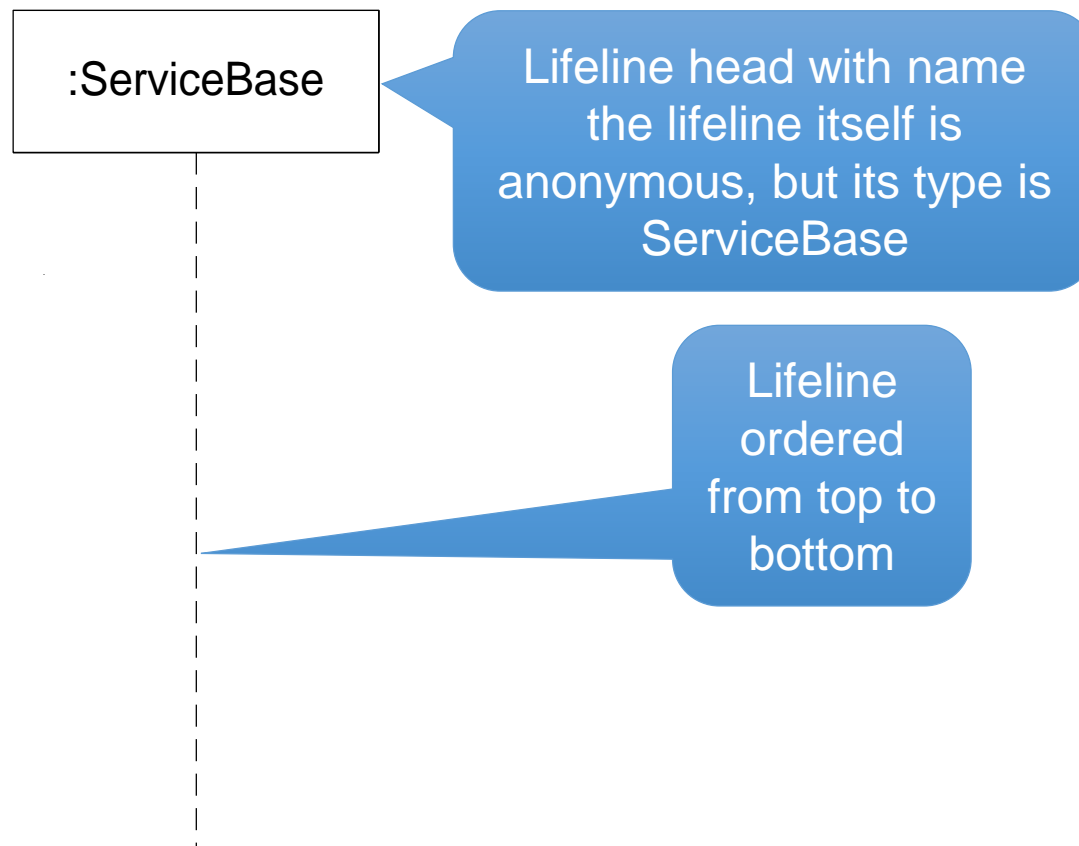
# The example context: Dolly Goes To Town

- Dolly is going to town and
  - wants to subscribe for bus schedules back home
  - given her current position
  - and the time of day.
  - The service should not come in effect until a given time in the evening

# The informal architecture

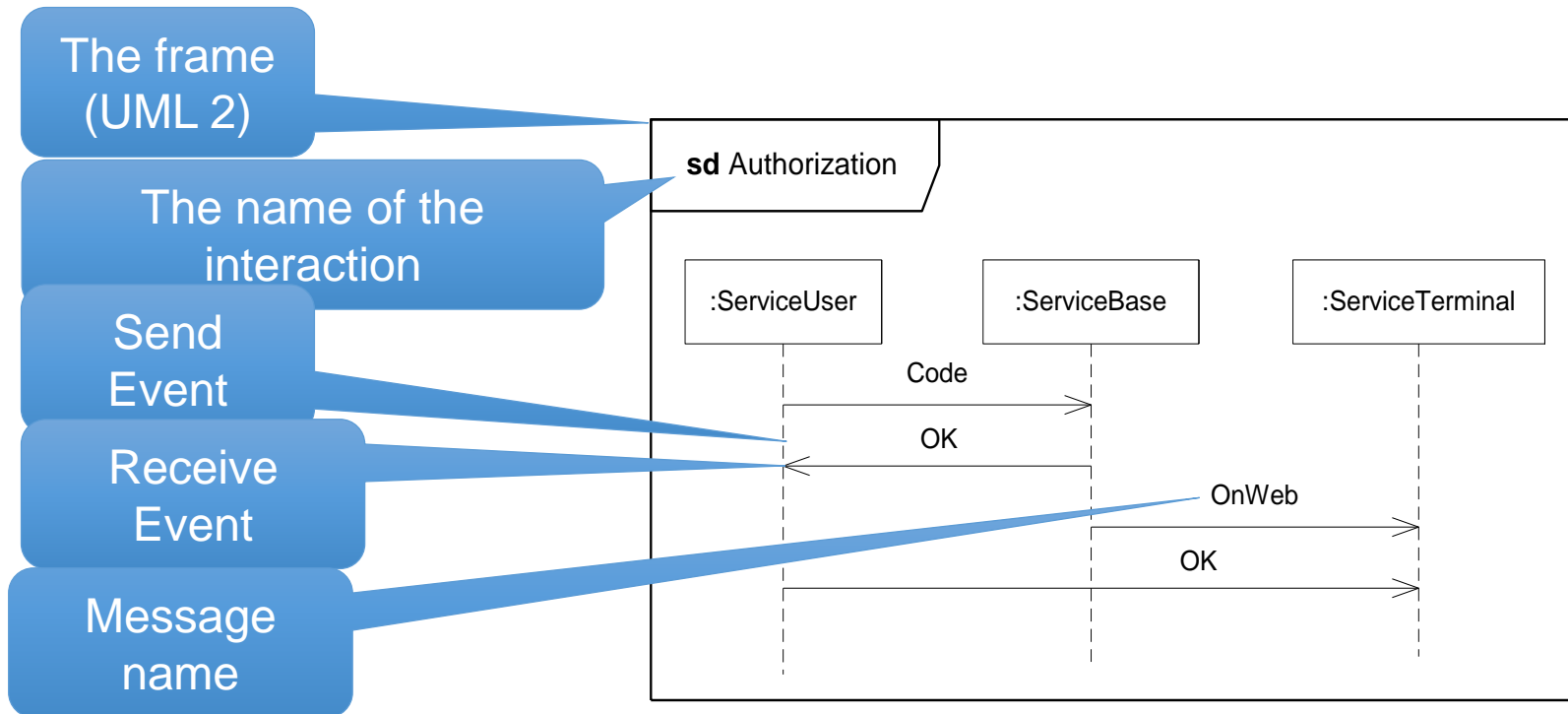


# Lifeline – the “doers”



# (Simple) Sequence Diagram

- Messages have one send event, and one receive event.
  - The send event must occur before the receive event.
- Events are strictly ordered along a lifeline from top to bottom

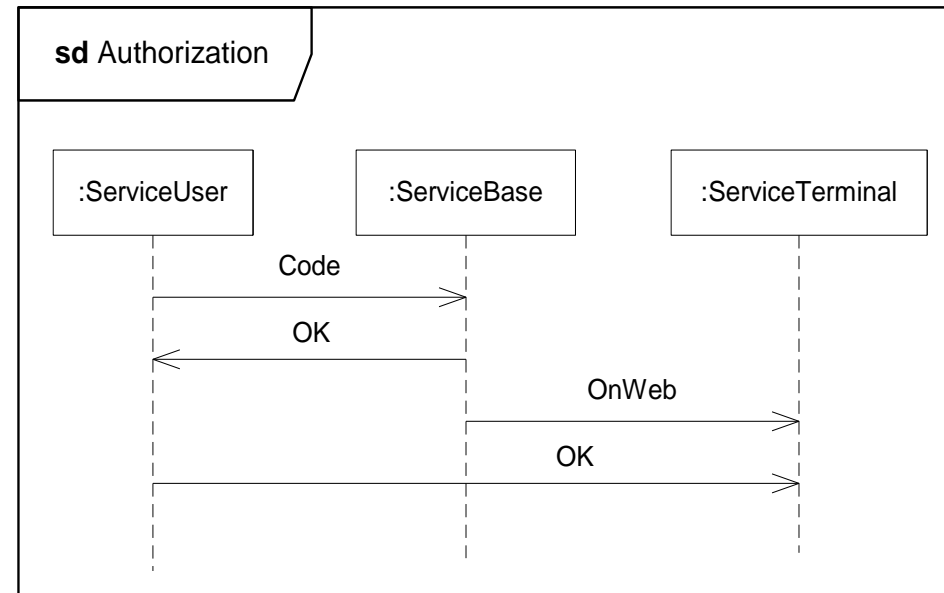


# How many global traces are there in this diagram?

- The only invariants:
  - Messages have one send event, and one receive event. The send event must occur before the receive event.
  - Events are strictly ordered along lifeline

How many?

- 1, 2, 3, 4, 5, 6,..?



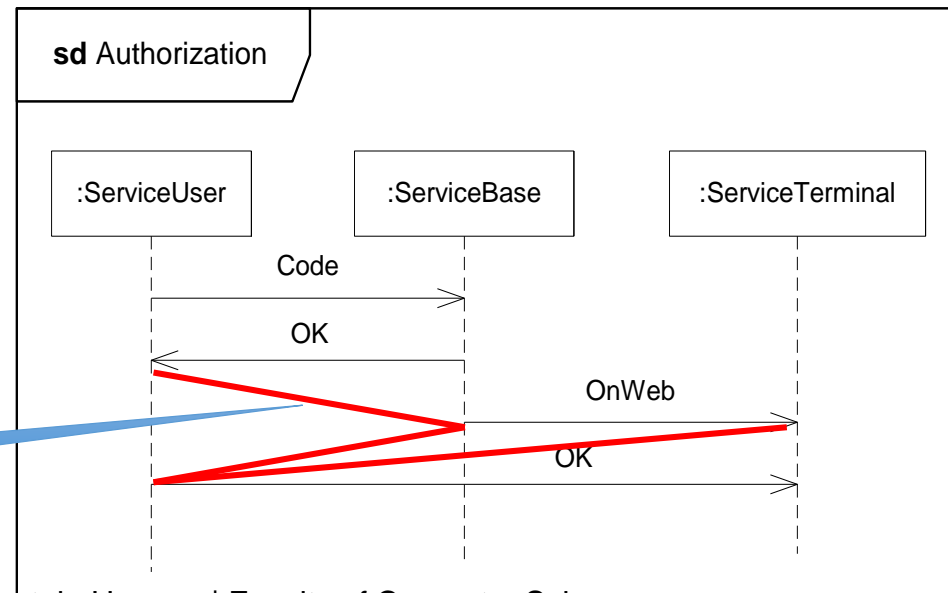
# How many global traces are there in this diagram?

- The only invariants:
  - Messages have one send event, and one receive event. The send event must occur before the receive event.
  - Events are strictly ordered along lifeline

How many?

- 1, 2, 3, 4, 5, 6,..?

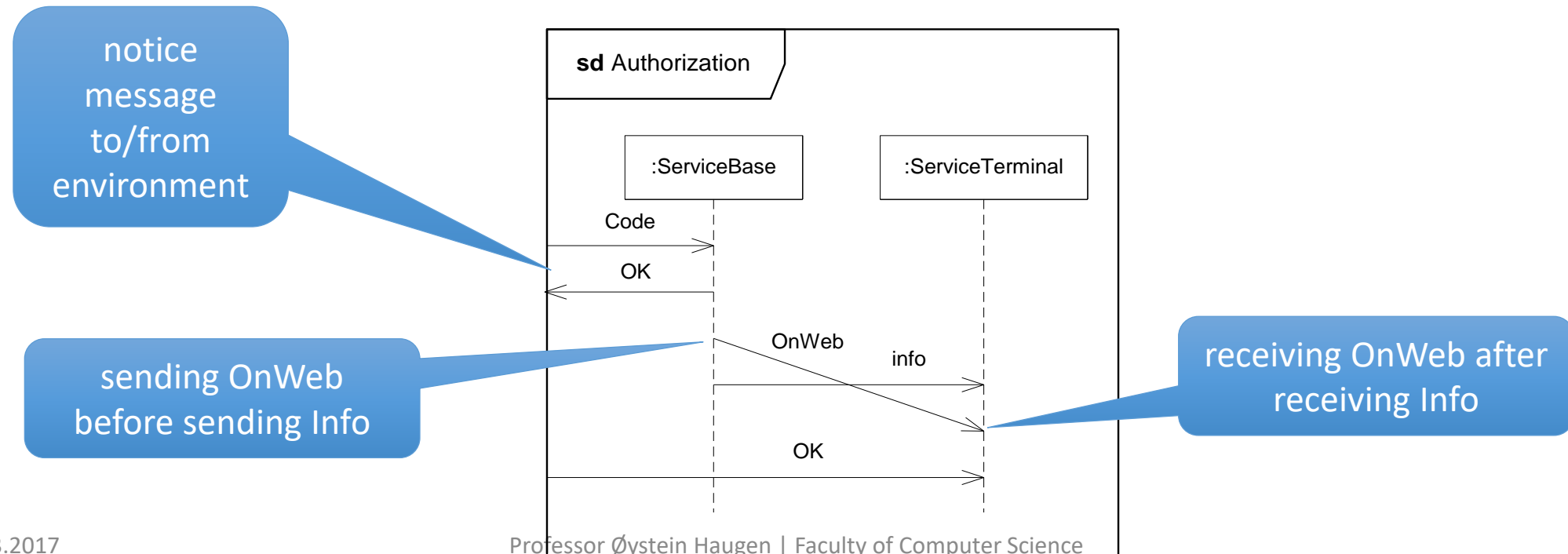
independent!





# Asynchronous messages: Message Overtaking

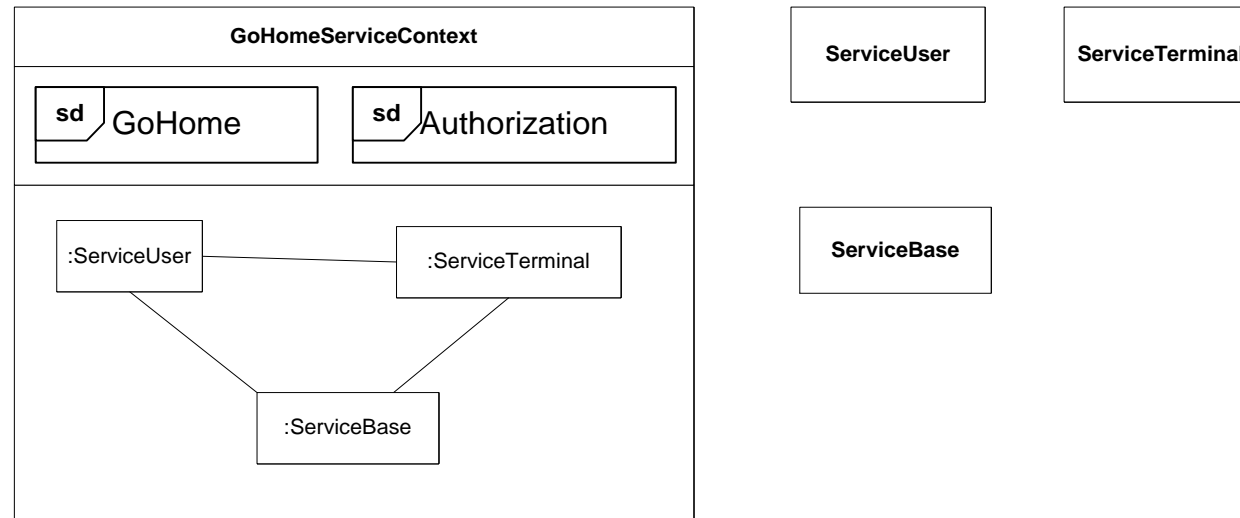
- asynchronous communication = when the sender does not wait for the reply of the message sent
- Reception is normally interpreted as consumption of the message.
- When messages are asynchronous, it is important to be able to describe message overtaking.



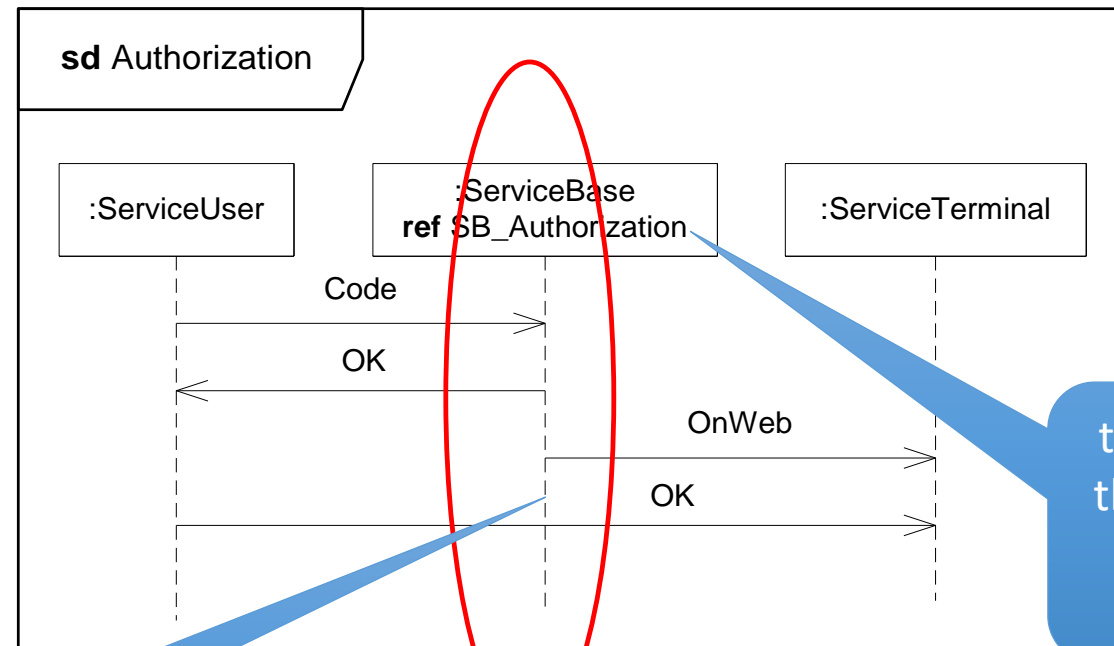


# The context of a Sequence Diagram

- The context is a Classifier with Composite Structure (of properties)
  - Properties (parts) are represented by Lifelines
- The concept of a context with internal structure leads to an aggregate hierarchy of entities (parts)
  - We exploit this through the concept of Decomposition



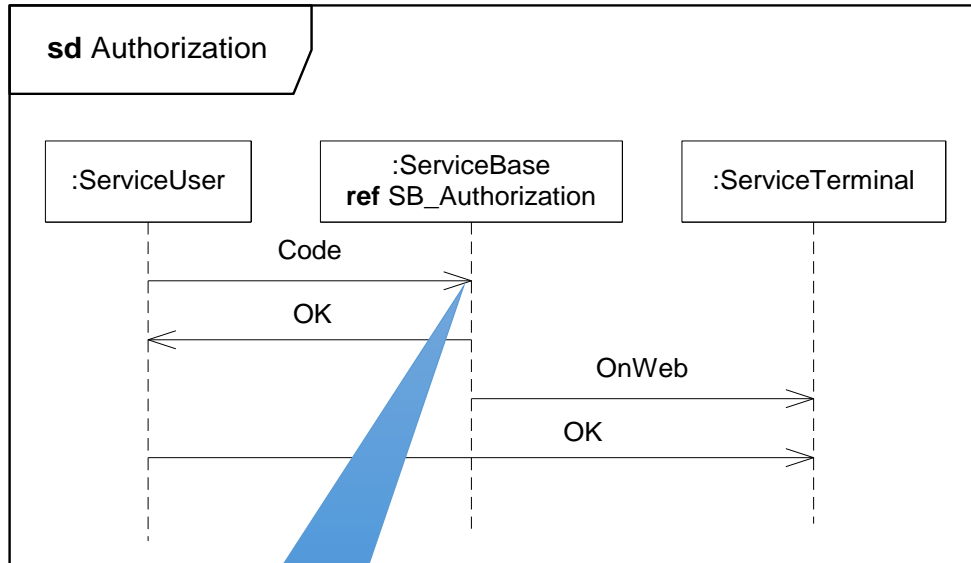
# Decomposing a Lifeline relative to an Interaction



we want to look into this lifeline

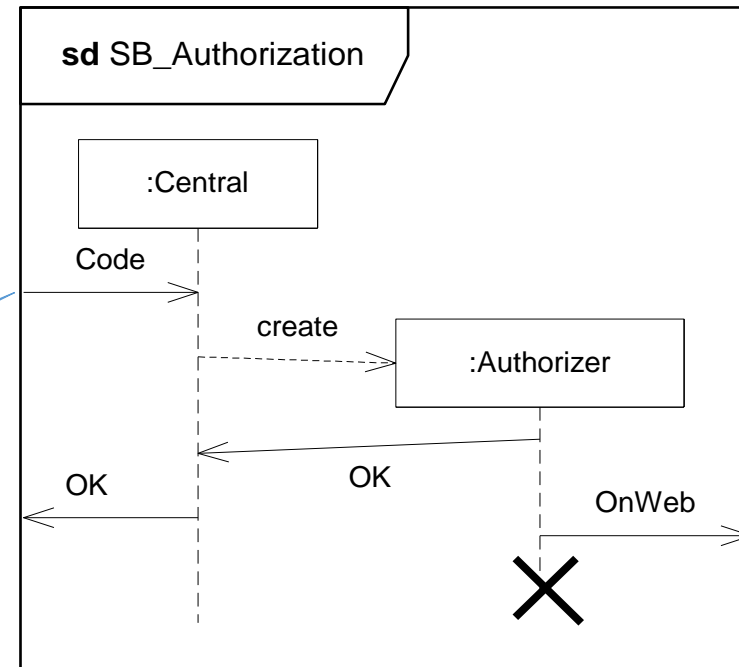
this is the name of the diagram where we find the decomposition

# The Decomposition



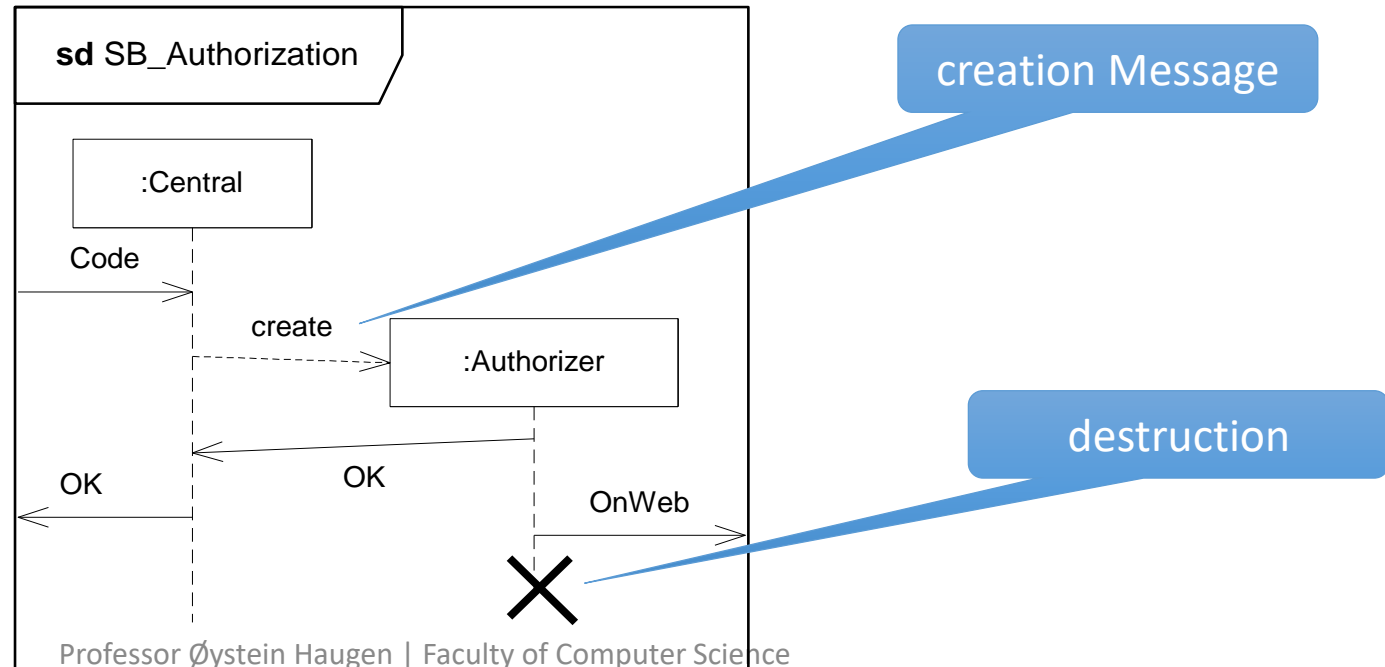
notice the *event* correspondence!

notice the *gate* correspondence!

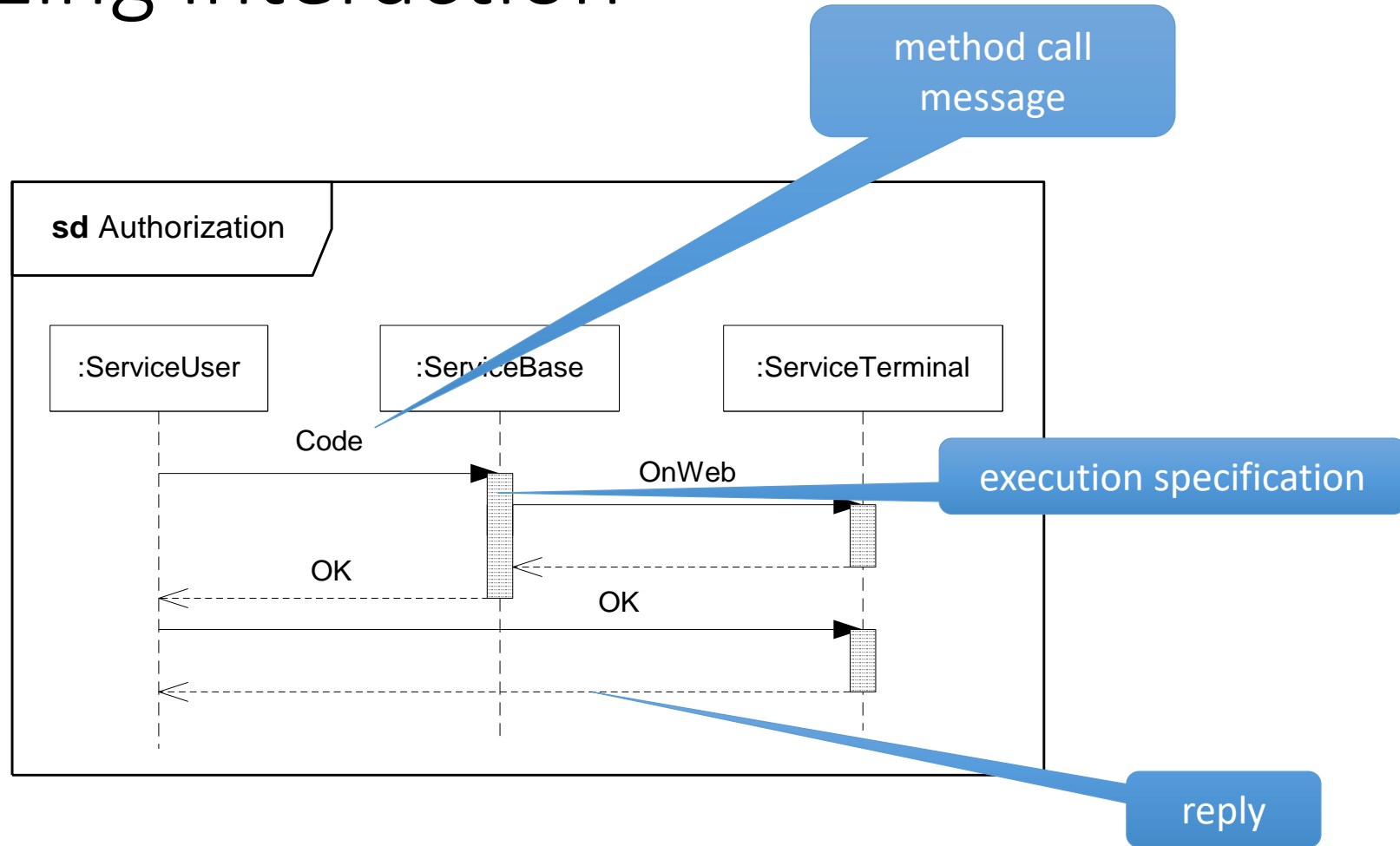


# Lifeline creation and destruction

- We would like to describe Lifeline creation and destruction
- The idea here (though rather far fetched) is that the ServiceBase needs to create a new process in the big mainframe computer to perform the task of authorizing the received Code. We see a situation where several Authorizers work in parallel



# Synchronizing interaction



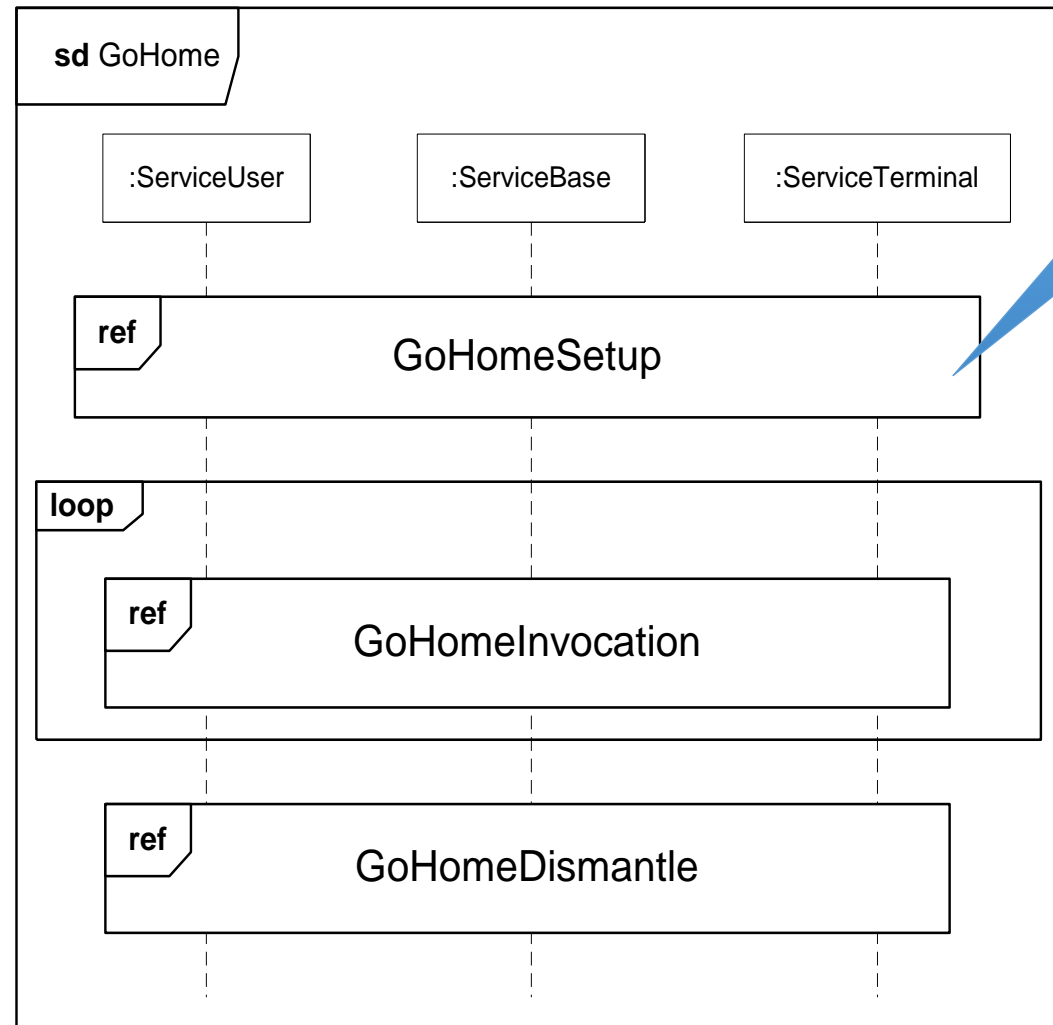
# Basic Sequence Diagrams Summary

- We consider mostly messages that are **asynchronous**, the sending of one message must come before the corresponding reception
- UML has traditionally described **synchronizing** method calls rather than asynchronous communication
- The events on a lifeline are strictly **ordered**
- The **distance** between events is not significant.
- The **context** of Interactions are classifiers
- A lifeline (within an interaction) may be detailed in a **decomposition**
- Dynamic **creation** and **destruction** of lifelines

# More structure (UML 2.0 from MSC-96)

- **interaction uses** – such that Interactions may be referenced within other Interactions
- **combined fragments** – combining Interaction fragments to express alternatives, parallel merge and loops
- **better overview** of combinations – High level Interactions where Lifelines and individual Messages are hidden
  - Not so useful since no tools support this
- **gates** – flexible connection points between references/expressions and their surroundings
  - we have looked at this in the context of decomposition, but gates are also on InteractionUse and CombinedFragments

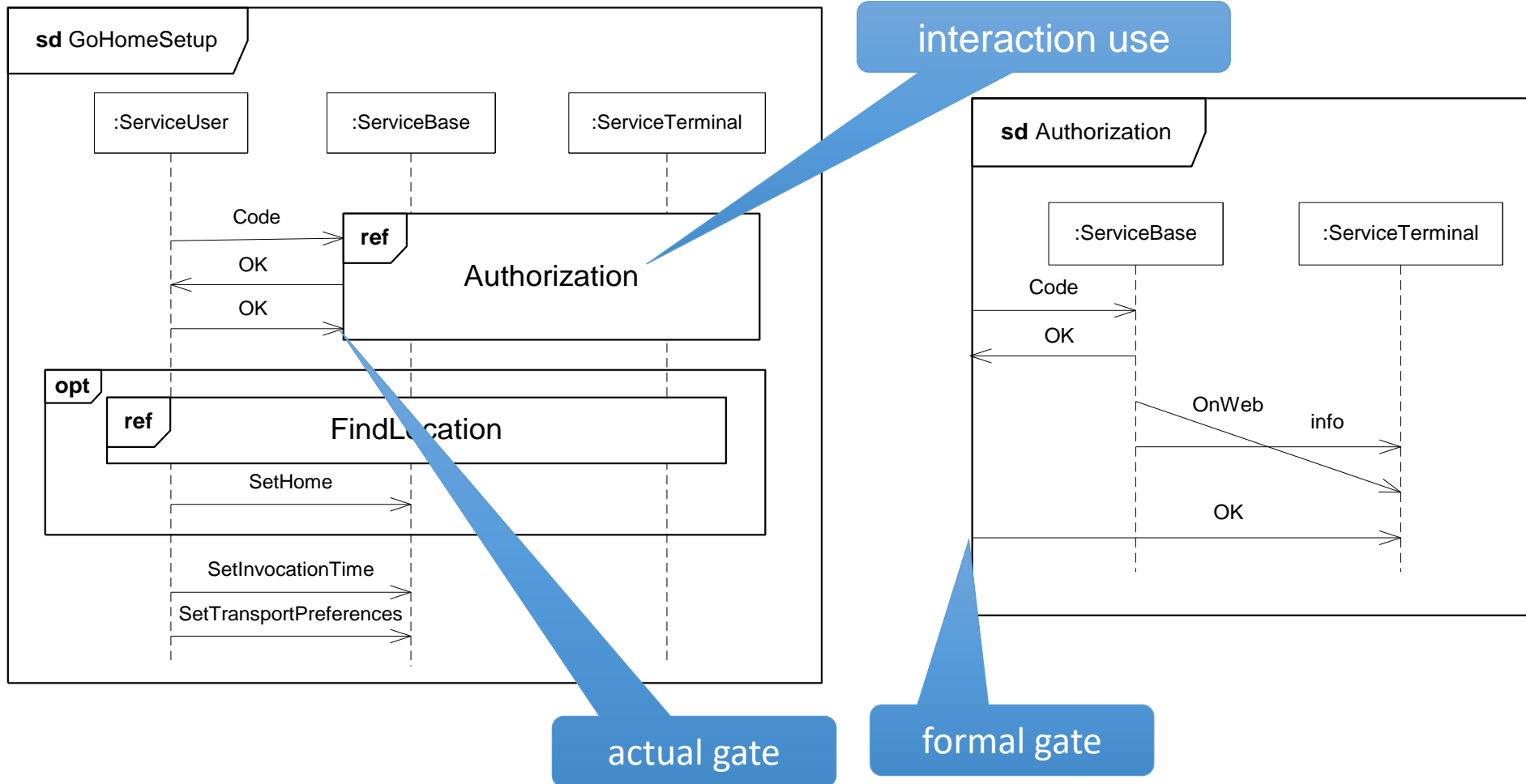
# References



interaction use



# Gates





# Data in Guards

guard on global or static values

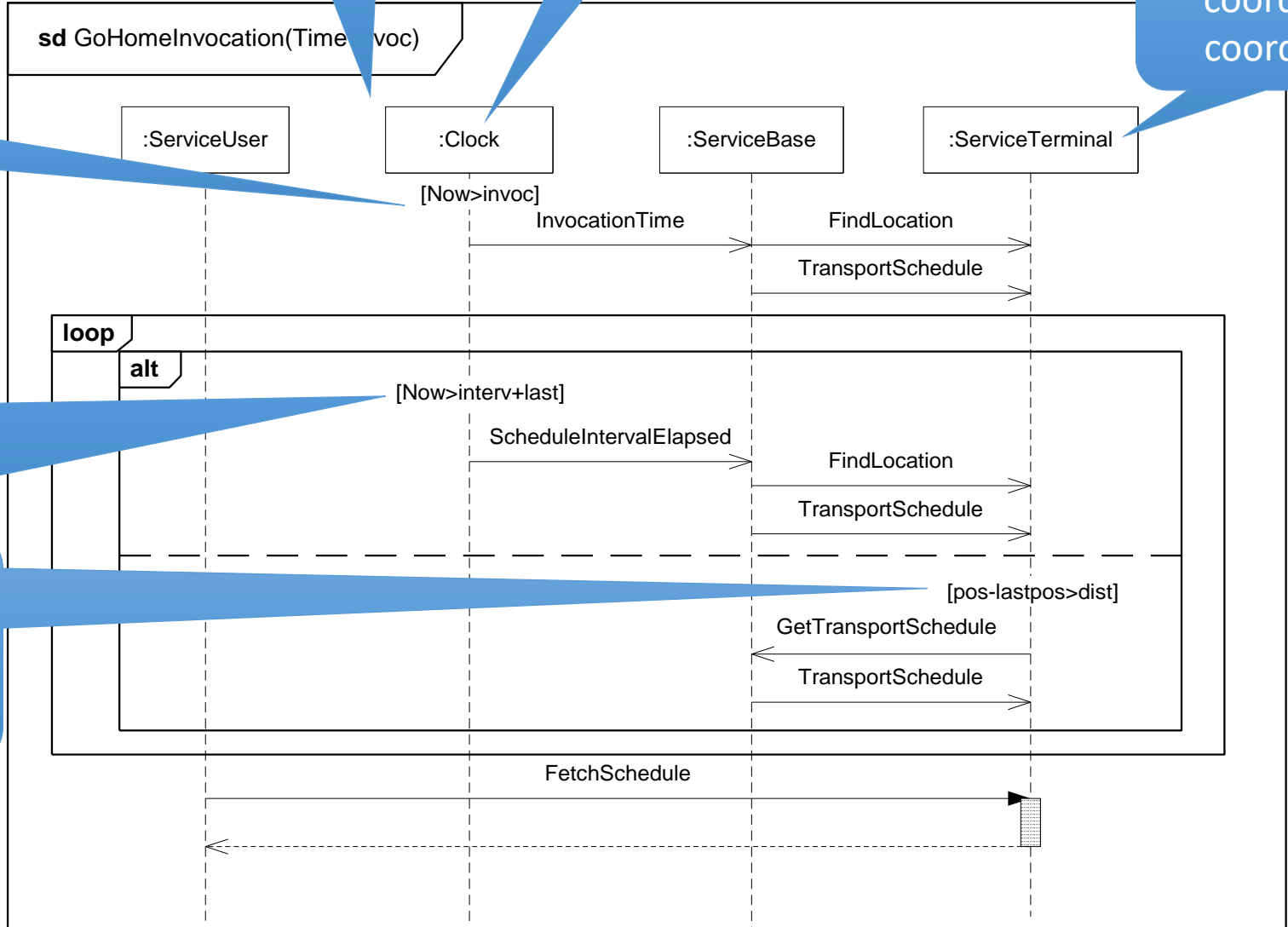
guard on dynamic and local values, local to the object of the first event of operand

note that there is no single object deciding the choice, but each guard is limited

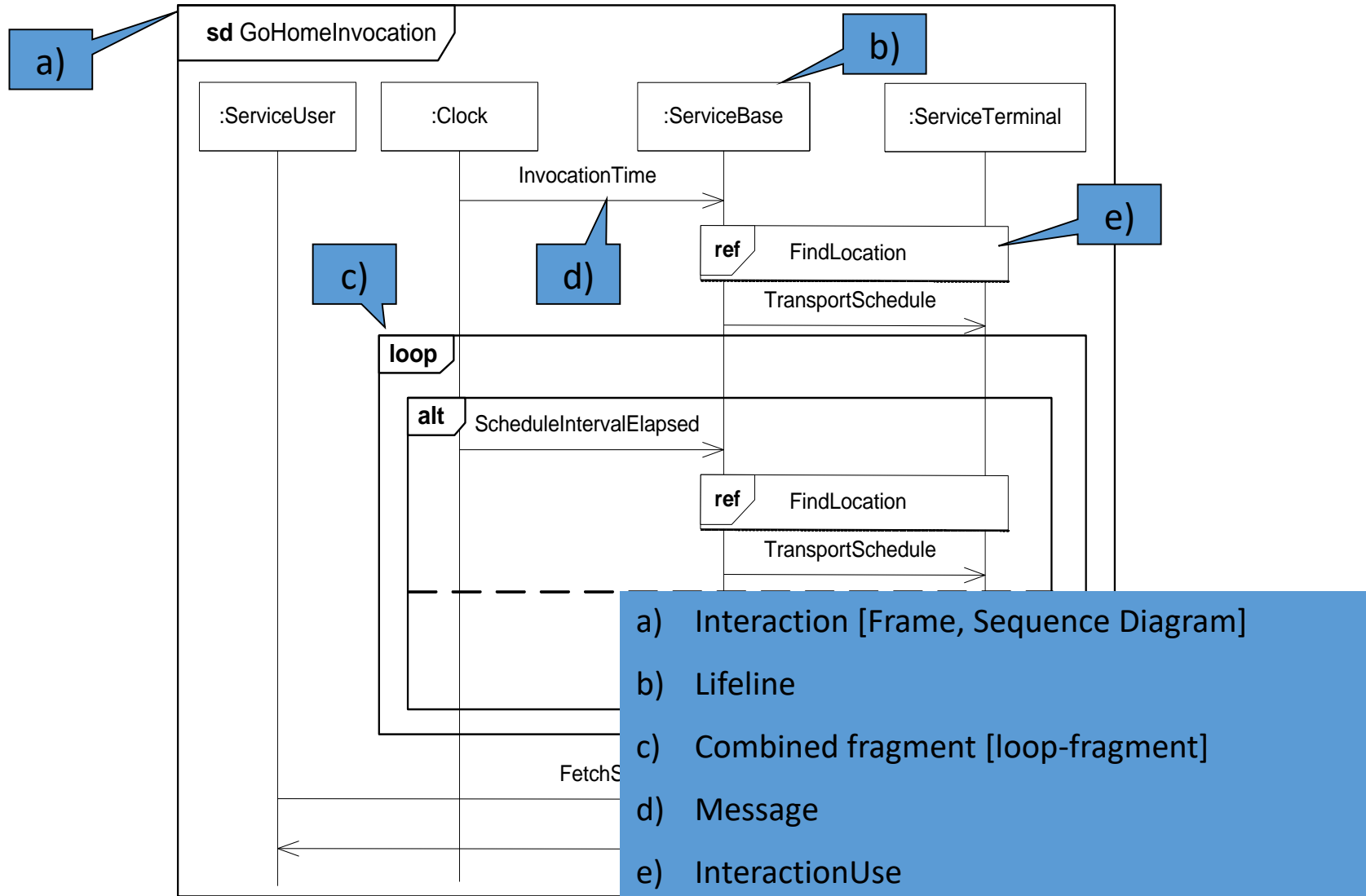
static parameter

time intv;  
time last;

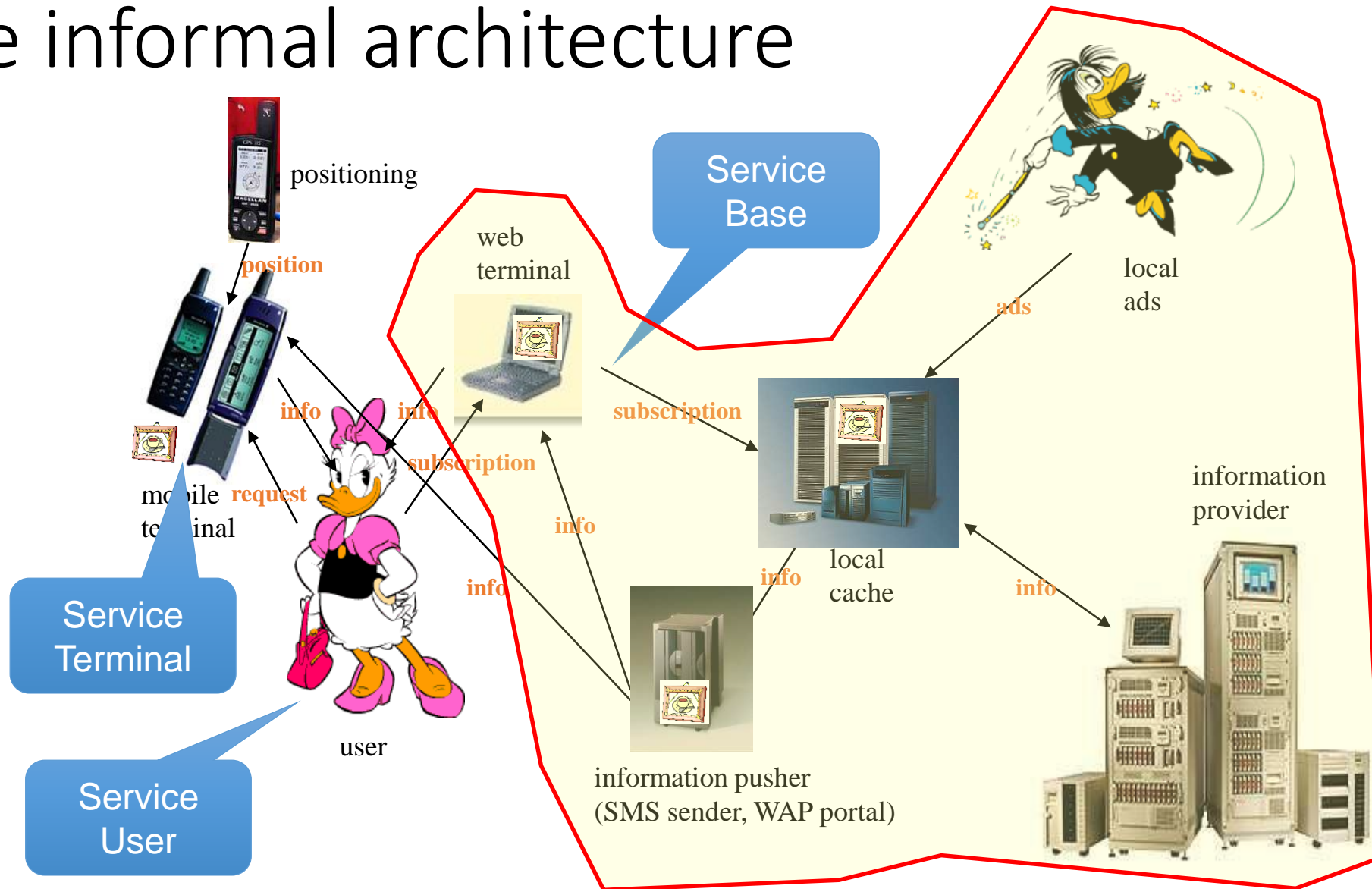
coord lastpos;  
coord dist;  
coord pos;



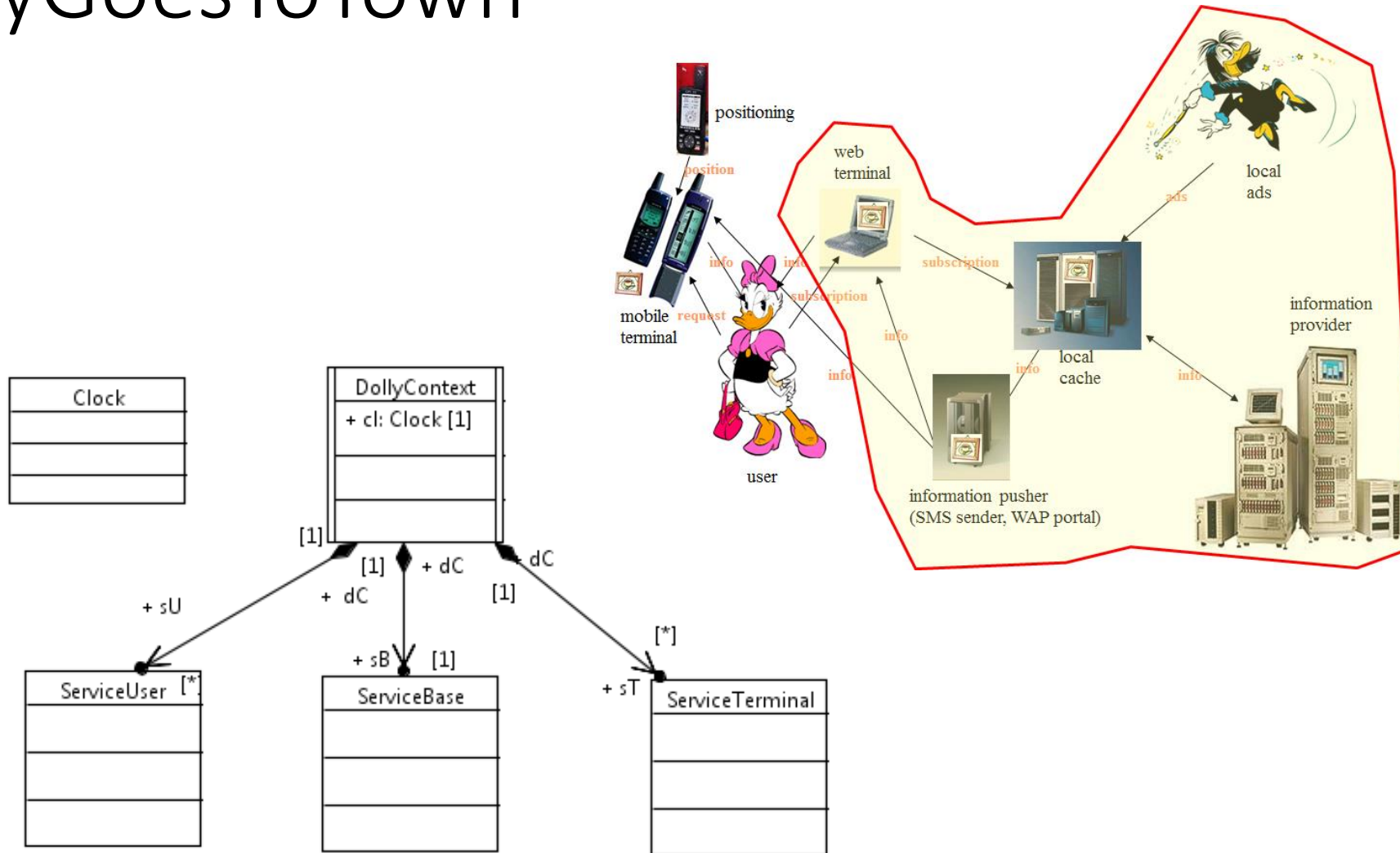
# And now chiefly yourselves !!!



# The informal architecture



# The UML architecture of the DollyGoesToTown

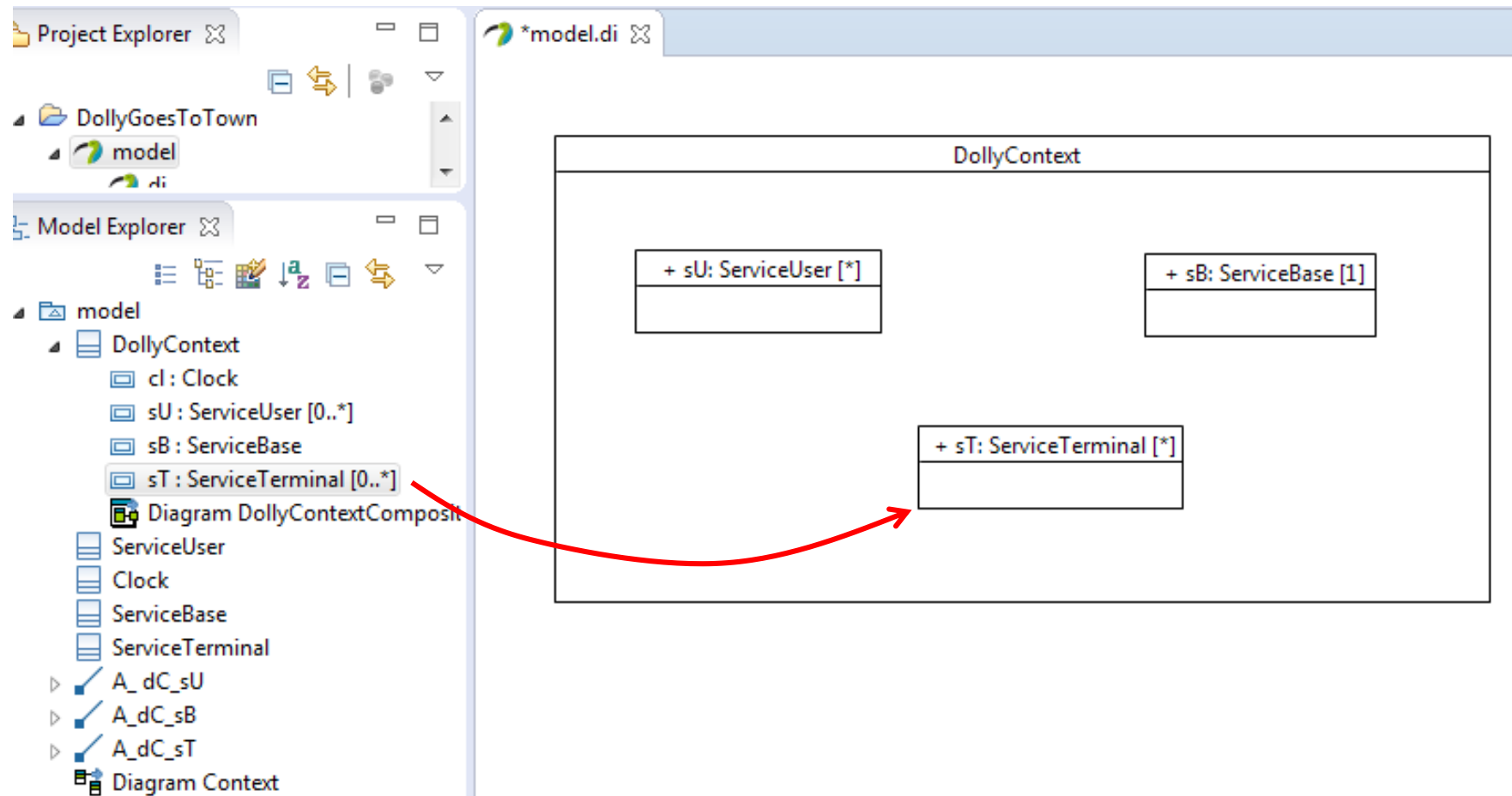


# Creating a Composite Structure

The screenshot shows the Eclipse IDE interface with the following components:

- Project Explorer:** Shows the project structure for 'DollyGoesToTown' with a sub-package 'model' containing 'di', 'notation', and 'uml'.
- Model Explorer:** Shows the 'DollyContext' class and its associated elements: 'cl: Clock', 'sU: ServiceUser', 'sB: ServiceBase', 'sT: ServiceTerminal', and 'Clock'.
- Diagram:** A UML Class Diagram for 'DollyContext' is displayed. It has an attribute 'cl: Clock [1]'. It is associated with 'ServiceUser' (multiplicity 1, role 'sU'), 'ServiceBase' (multiplicity 1, role 'sB'), and 'ServiceTerminal' (multiplicity 1, role 'sT'). There are also dependencies on 'ServiceUser' (multiplicity 1, role 'dC') and 'ServiceTerminal' (multiplicity 1, role 'dC').
- Context Menu:** A context menu is open over the 'DollyContext' class, with 'Create a new UML Composite Structure Diagram' selected.
- Palette:** The 'Nodes' section is expanded, showing various UML diagram elements like Class, Component, and Association.
- Properties View:** Shows the properties for the 'DollyContext' class, including 'Qualified name: model::DollyContext', 'Is abstract: false', 'Is leaf: false', 'Visibility: public', and 'Owned attribute: cl: Clock, sU: ServiceUser [0..\*], sB: ServiceBase, sT: ServiceTerminal'.

# Dragging Properties into it





# Sometimes there is just too much text

The screenshot displays the Papyrus UML editor interface. The main diagram shows a context named 'DollyContext' containing several elements: a class 'ServiceUser' with multiplicity '[\*]', a class 'ServiceBase' with multiplicity '[1]', and a class 'ServiceTerminal' with multiplicity '[\*]'. There are two associations: one from 'ServiceUser' to 'ServiceBase' with role 'to\_sB' and multiplicity '[\*]', and another from 'ServiceBase' to 'ServiceUser' with role 'from\_sU' and multiplicity '[1]'. A third association is shown as a dashed line from 'ServiceBase' to 'ServiceTerminal' with role 'to\_sT' and multiplicity '[1]'. This association is circled in red. The 'Properties' view at the bottom is also circled in red, showing the 'Appearance' tab for the 'to\_sT' association. The 'Label customization' section has several checkboxes checked: 'Multiplicity', 'Visibility', 'Type', and 'Is Derived'. The 'Font' section shows 'Segoe UI' font and '9' font height. The 'Style' section shows a bold 'B' icon. The 'Position' is set to 'Top Left'. The 'Stereotype display' is set to 'Text' and 'Text alignment' is 'Horizontal'. The 'Applied stereotypes' section is empty.

# Other means to change appearance

The screenshot shows the Eclipse IDE interface with a UML diagram open. A context menu is displayed over a connector, with the 'No Connector Labels' option selected. A blue callout bubble points to this menu item with the text 'Filter out labels'. Another blue callout bubble points to the connector with the text 'Rightclick on connector'. The Properties view for 'Connector3' is visible at the bottom, showing various appearance settings like line width, color, and routing styles.

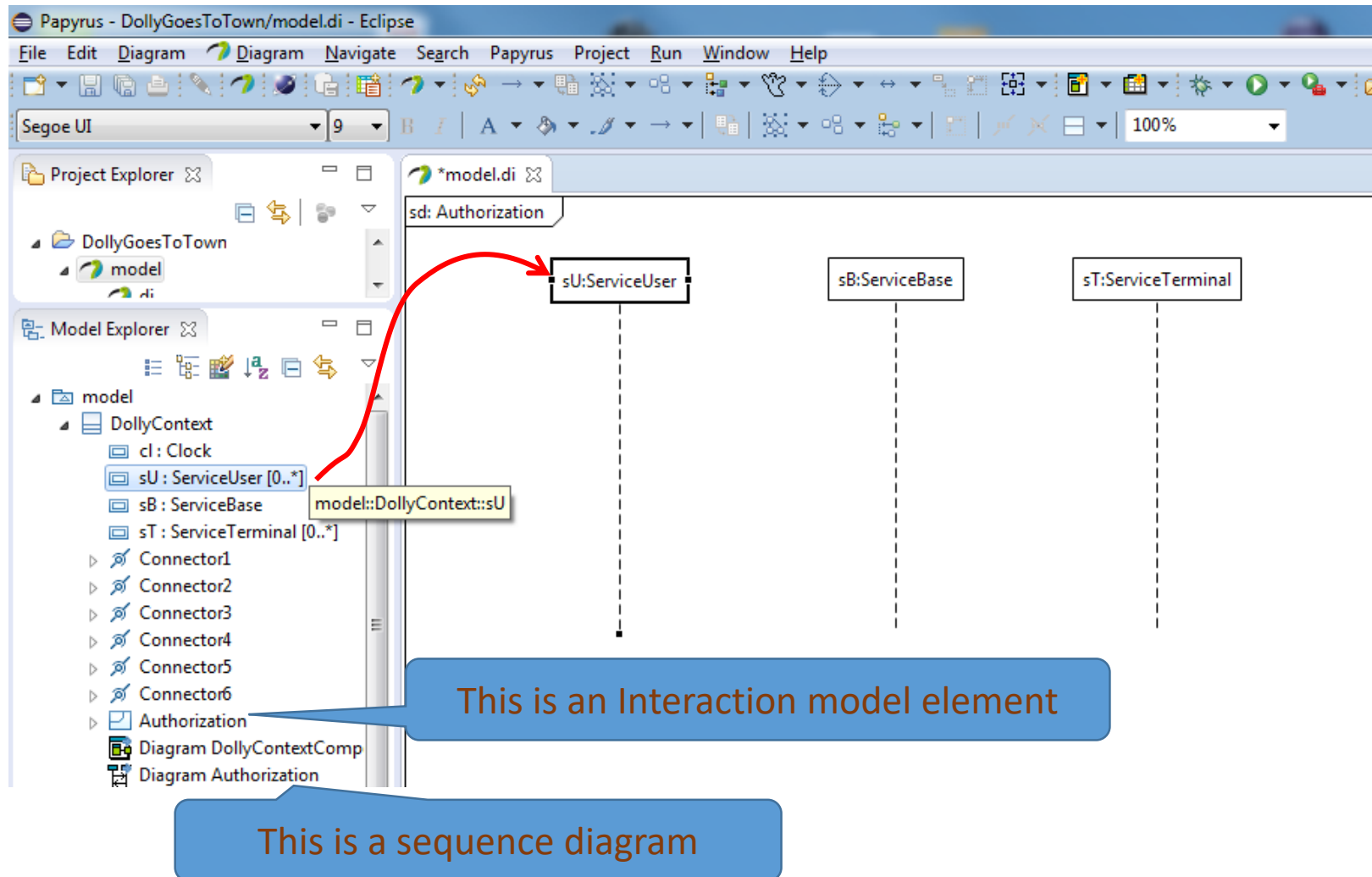
# Adding Sequence Diagrams

The screenshot shows the Eclipse IDE interface with a UML model named 'DollyContext'. The 'DollyContext' class is selected in the Package Explorer, and a context menu is open over it. The menu options are:

- New SysML Child
- New Child
- New Diagram
  - Create a new UML Activity Diagram
  - Create a new UML Communication Diagram
  - Create a new UML Composite Structure Diagram
  - Create a new UML Inner Class Diagram
  - Create a new UML Interaction Overview Diagram
  - Create a new UML Sequence Diagram
  - Create a new UML StateMachine Diagram
- Delete
- Rename
- Undo
- Redo
- Cut
- Copy
- Paste
- Import
- Validation
- Create submodel unit
- Enable write

The 'Create a new UML Sequence Diagram' option is highlighted. A callout bubble points to the 'DollyContext' class in the Package Explorer with the text 'Rightclick on Context (owner)'. Another callout bubble points to the 'Create a new UML Sequence Diagram' menu item with the text 'Create seq diagram'. The Properties view at the bottom shows the details for the 'DollyContext' class, including its qualified name, visibility, and owned attributes.

# Drag the Properties to make Lifelines



# Creating messages

**Create a new Message**

Select an existing element

filter out all signals which are not receivable

**Create a new element**

Type: **Signal**

Name: OK

Owner of the created element: <Model> model

No element

OK Cancel

Qualified Name  
Receive Event  
Send Event  
Signature  
Visibility

model:DollyContext::Authorization::OK\_Message  
<Message Occurrence Specification> OK\_MessageRecv  
<Message Occurrence Specification> OK\_MessageSend  
<Signal> OK  
Public

# Creating messages

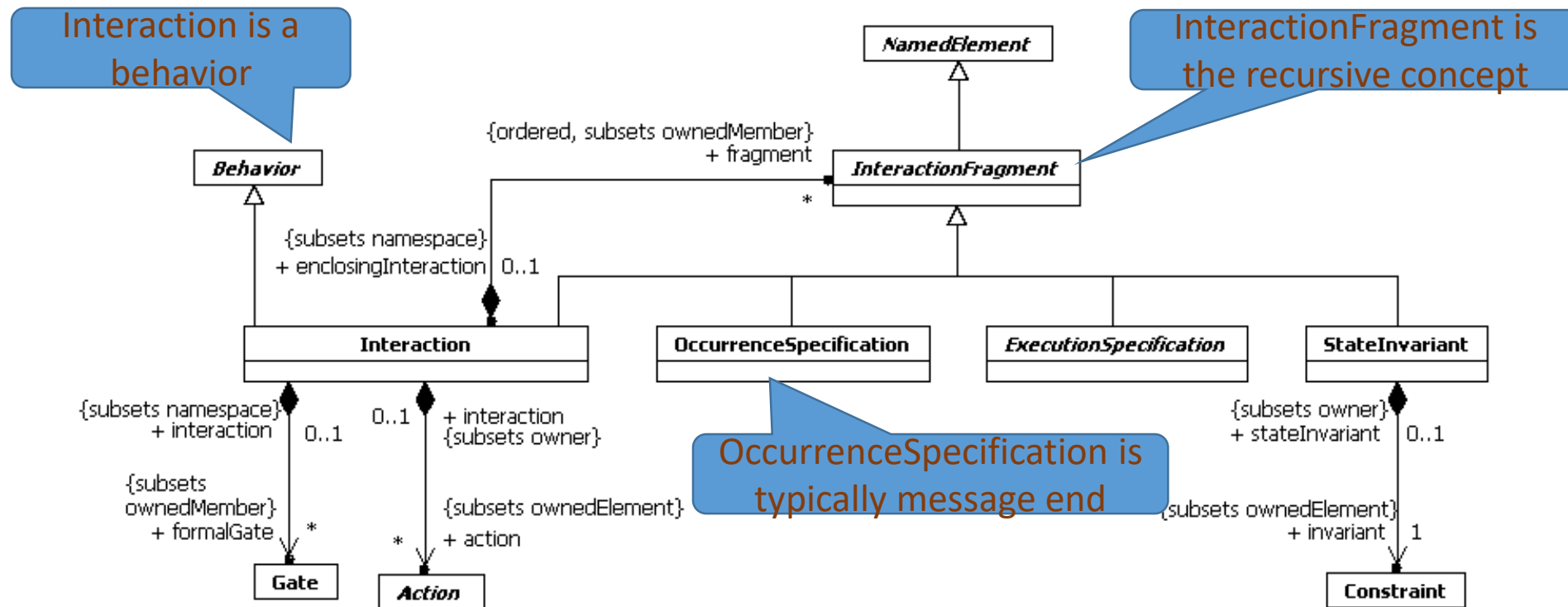
The screenshot displays the Eclipse IDE interface for creating a UML diagram. The central workspace shows a sequence diagram titled 'sd: Authorization' with three lifelines: sU:ServiceUser, sB:ServiceBase, and sT:ServiceTerminal. A message 'Code()' is sent from sU to sB, and a return message 'OK()' is sent from sB to sU. The 'OK\_Message' element is selected in the Model Explorer on the left. The Properties view at the bottom shows the following details for 'OK\_Message':

Property	Value
UML	
Connector	Asynchronous Signal
Message Sort	OK_Message
Name	<Interaction> Authorization
Namespace	model:DollyContext::Authorization::OK_Message
Qualified Name	<Message Occurrence Specification> OK_MessageRecv
Receive Event	<Message Occurrence Specification> OK_MessageSend
Send Event	<Signal> OK
Signature	Public
Visibility	

# Interaction Metamodel

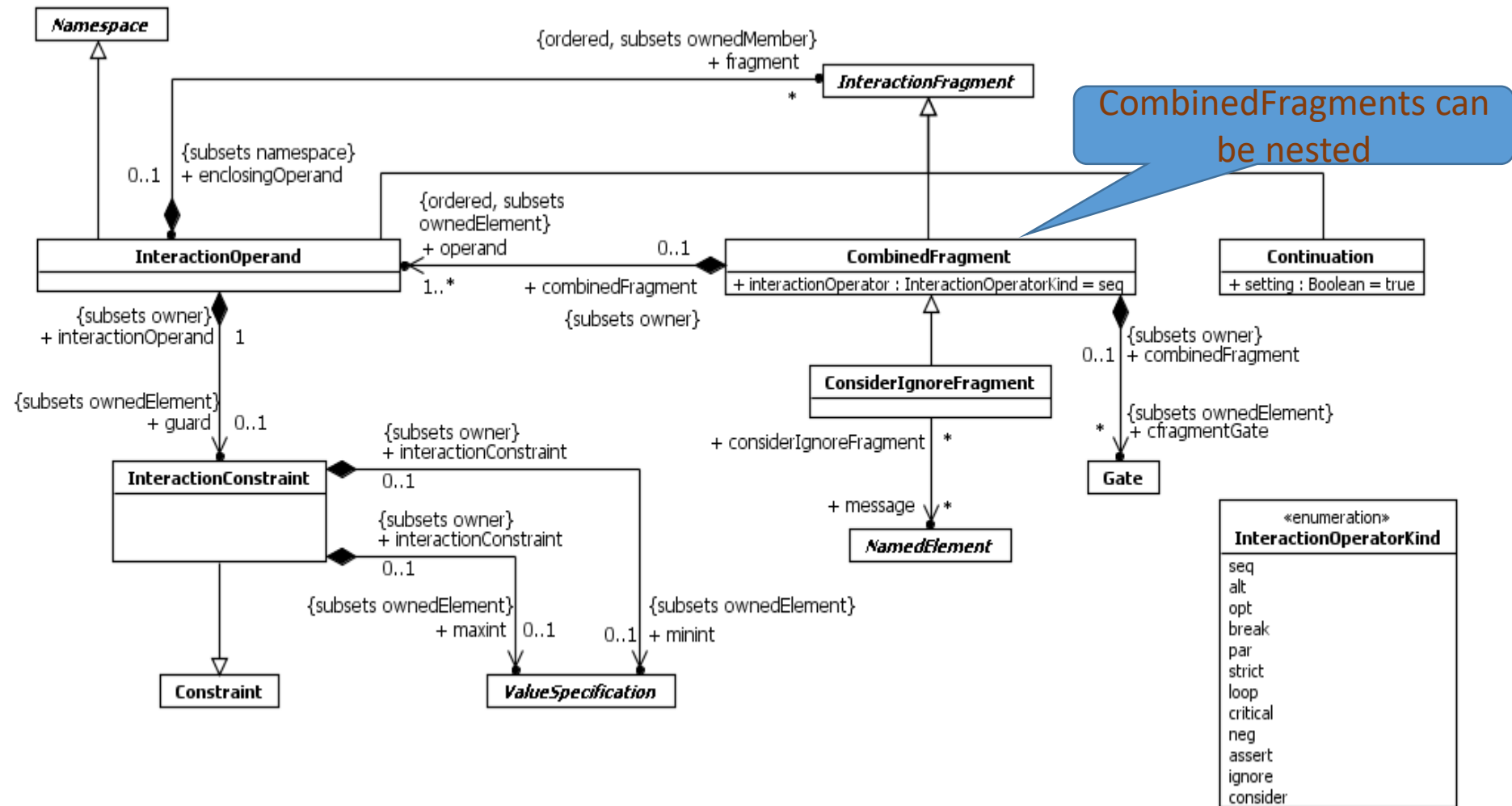


# Parts of ThingML metamodel





# More on ThingML metamodel



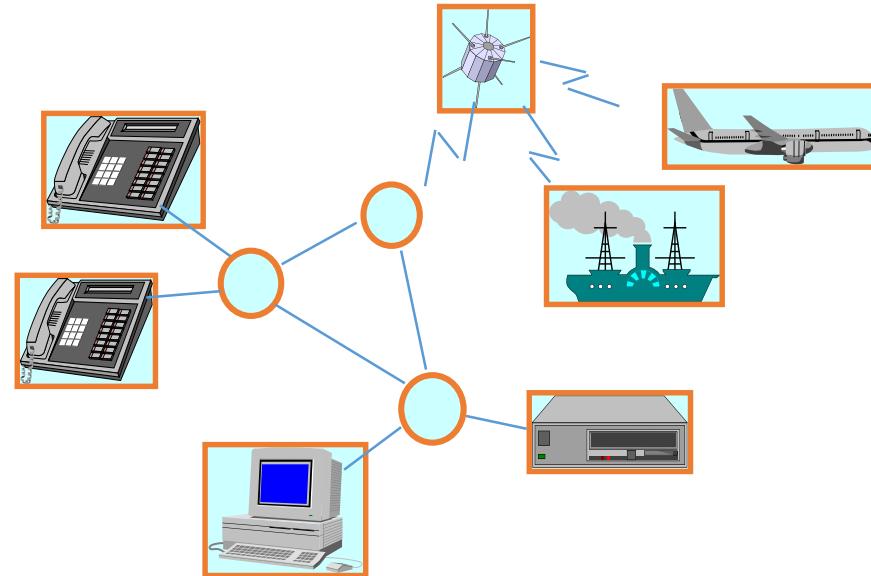
# State Machines and Model Consistency

# Overview of lecture – State Machines and Model Consistency

- State Machines for what kind of systems?
- State Machine – a concept not found in Java
- The History Lesson
- Consistency
  - Design time consistency
  - Runtime consistency
- Tooling
  - Papyrus

# Systems suitable for communicating set of state machines

- reactive
- concurrent
- real-time
- distributed
- heterogeneous
- complex



# Finite State Machines

- Finite
  - a finite number of states
  - [here] a small number of named states
- State
  - a stable situation where the process awaits stimuli
  - a state in a state machine represents the history of the execution
- Machine
  - that only a stimulus (signal, message) triggers behavior
  - the behavior consists of executing transitions
  - may also have local data

# A very brief history of State Machines

- Finite State Machines, or automata, originated in computational theory and mathematical models in support of various fields of bioscience.
- Pioneering efforts of George H. Mealy and Edward F. Moore performed at Bell Labs and IBM (circa 1960s).
  - Mealy and Moore's Finite State Machine concepts proved valuable in language parsing (compilers) and sequential circuit design.
- SDL (ITU recommendation Z.100) from 1980ies
  - Telecom systems were the biggest software of that time
- David Harel published [Statecharts: A Visual Formalism for Complex Systems](#). Harel embellished the Mealy and Moore paradigm with the concept of hierarchical finite state machines (1987).

# An example

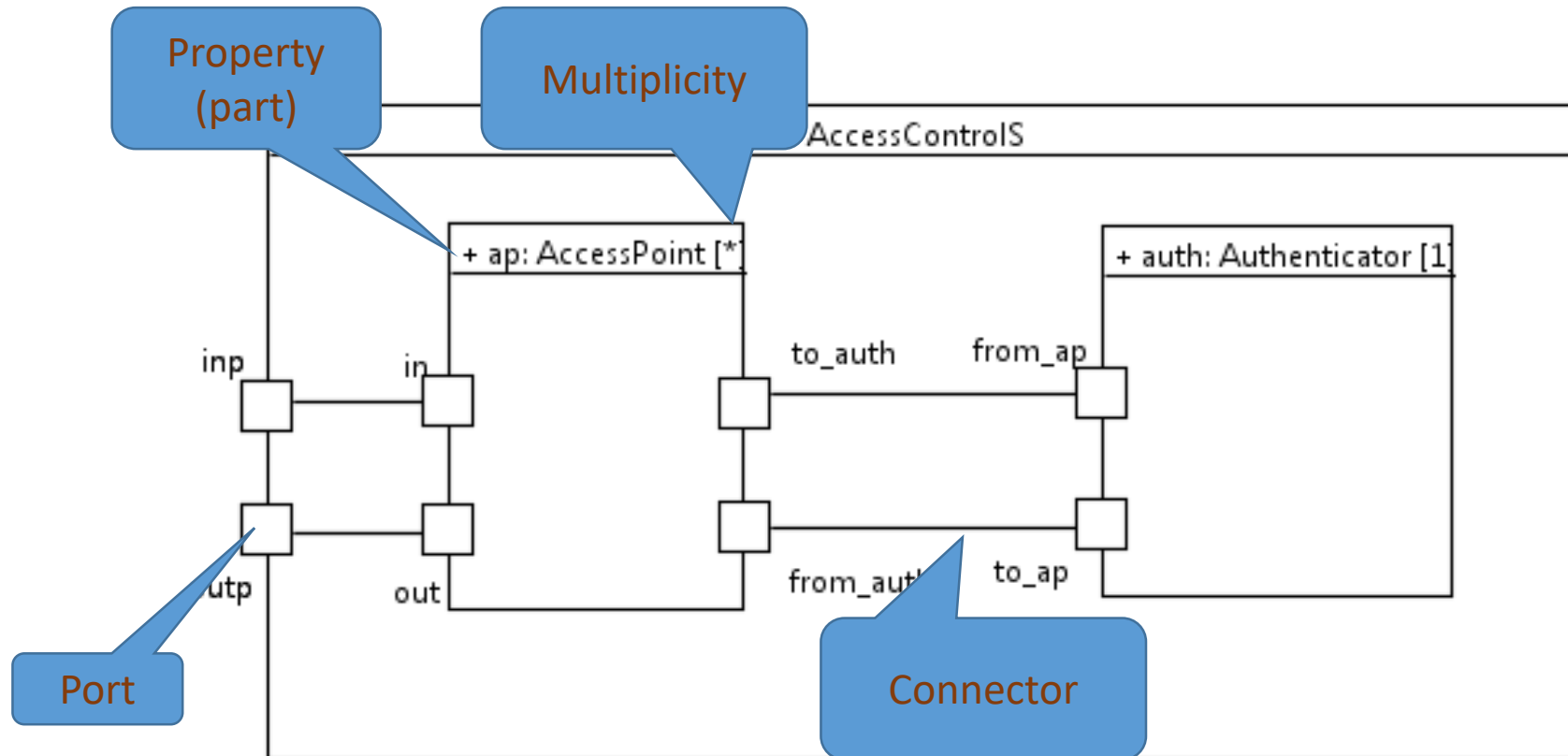


# An Access Control System

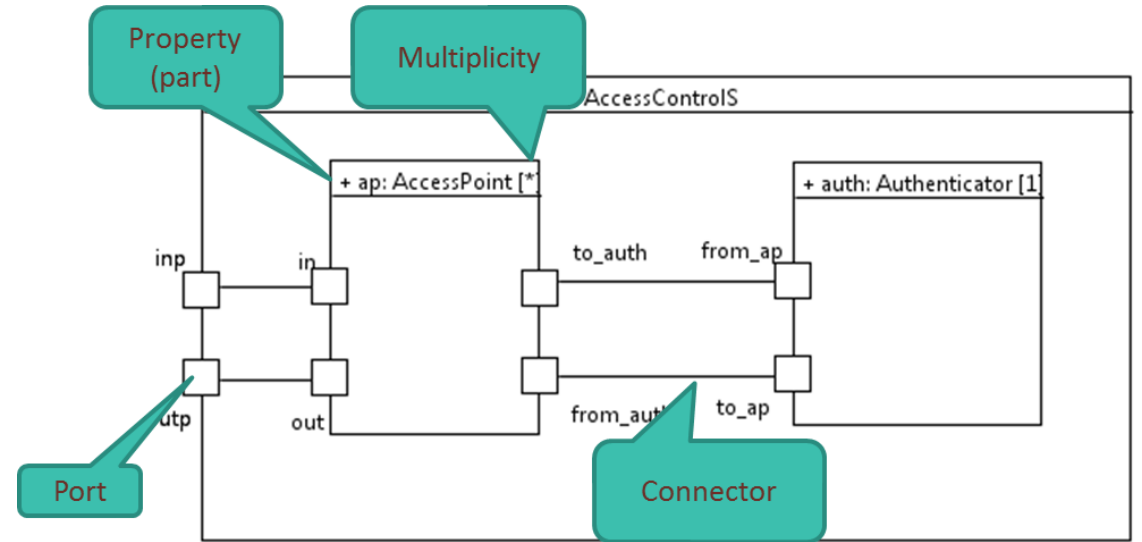
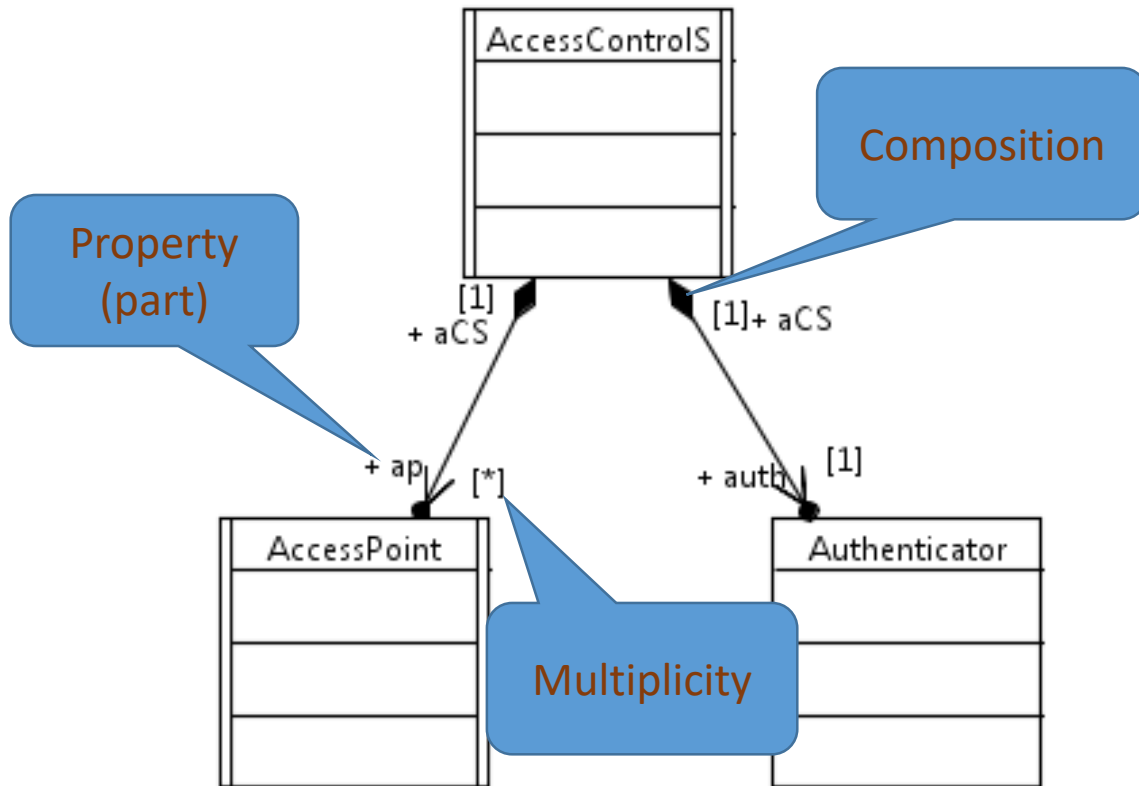
- A set of Access Points are established to control the access to an area
- The Access Points controls the locking of a door
  - in a more abstract sense, access control systems may control bank accounts or any other asset that one wants to protect
- The Access Point access is granted when two pieces of correct identification is presented
  - A card
  - A PIN (Personal Identification Number)
- The access rights are awarded by a central Authentication service



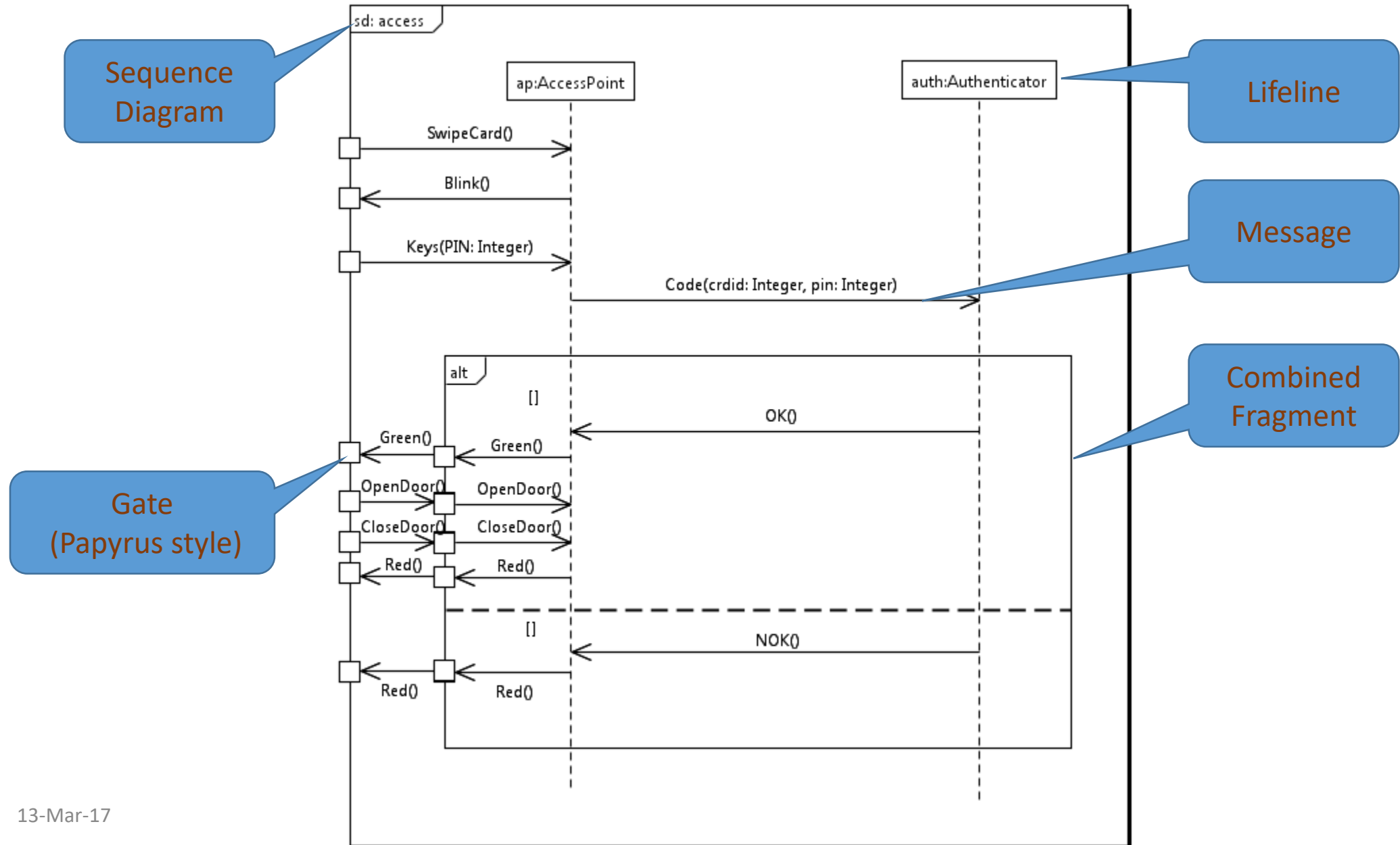
# The architecture in a composite structure



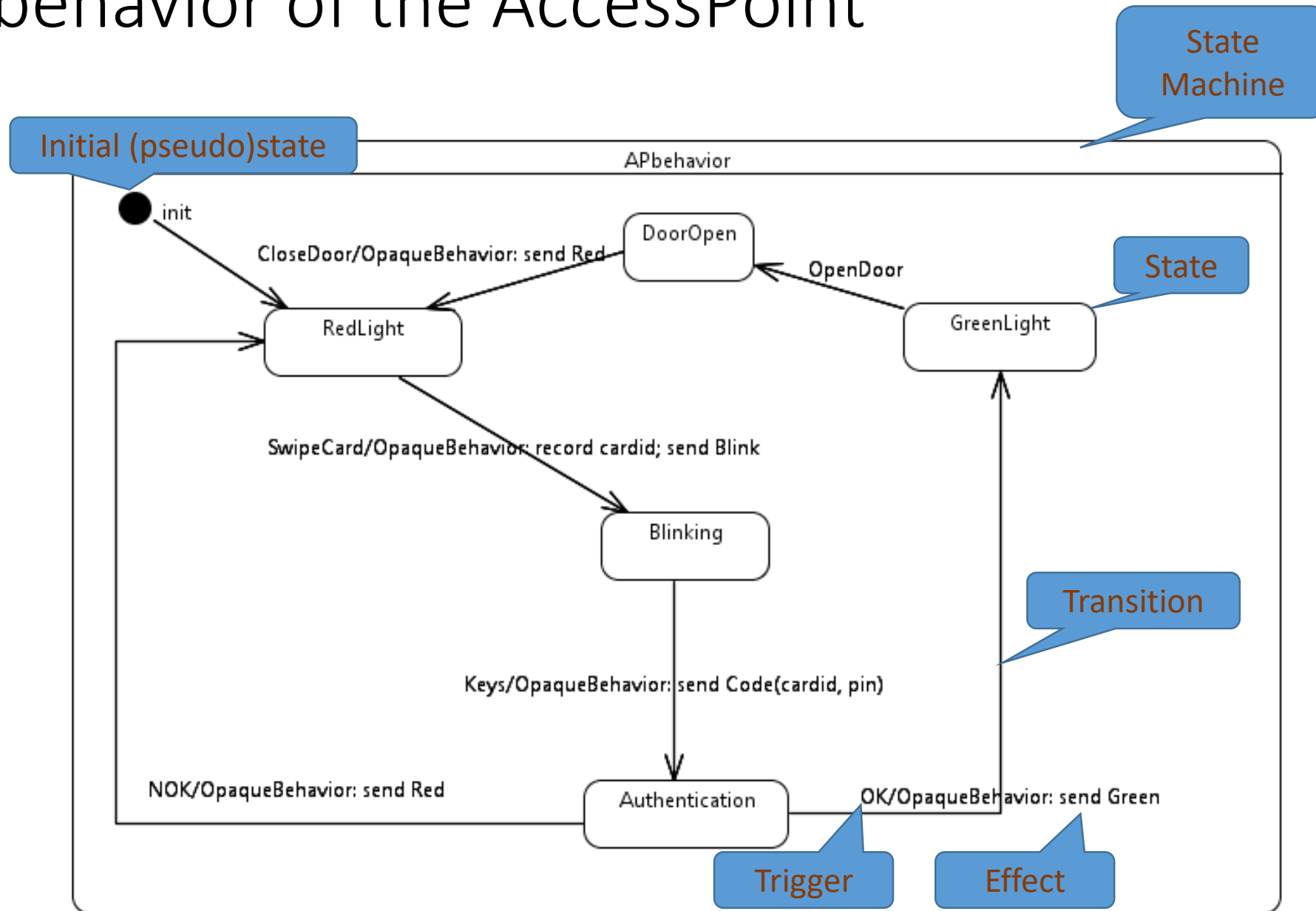
# The concepts in a class diagram



# Happy Day Scenario

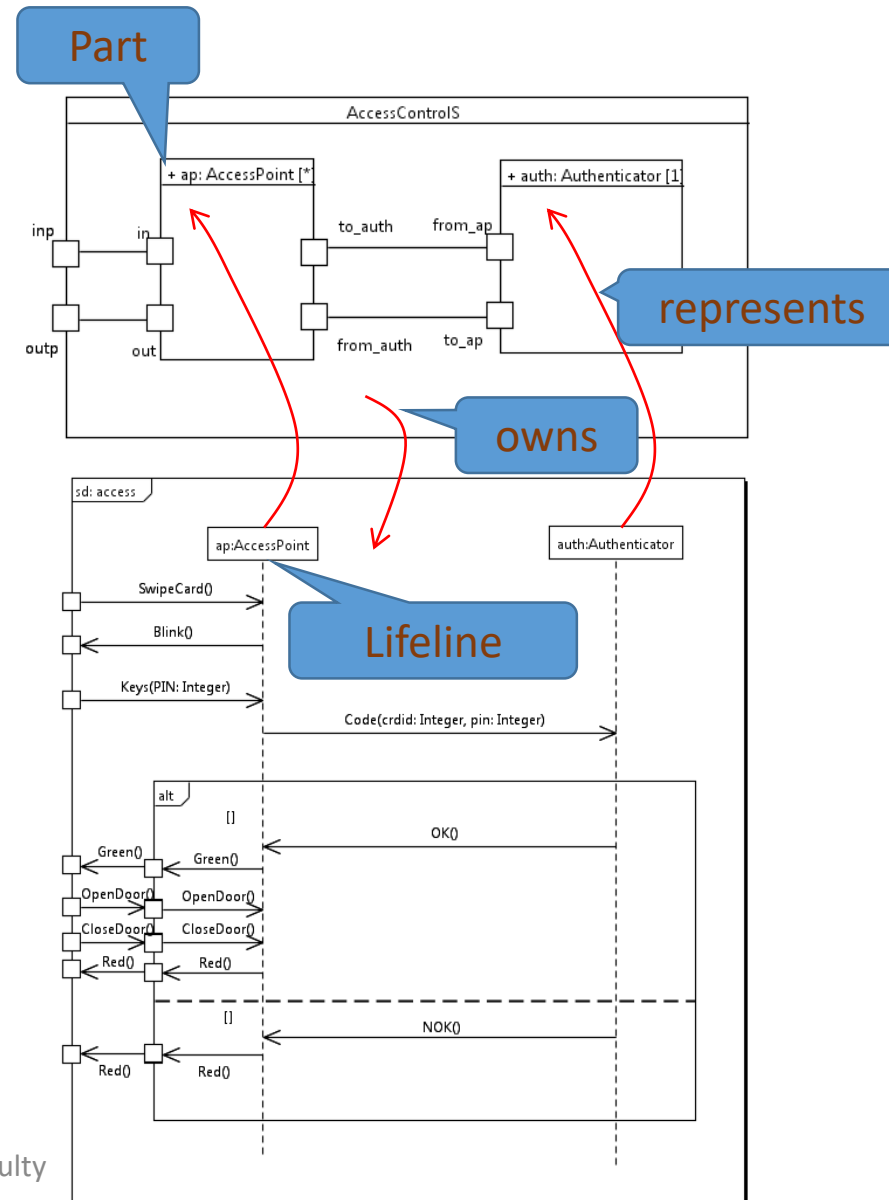


# The behavior of the AccessPoint

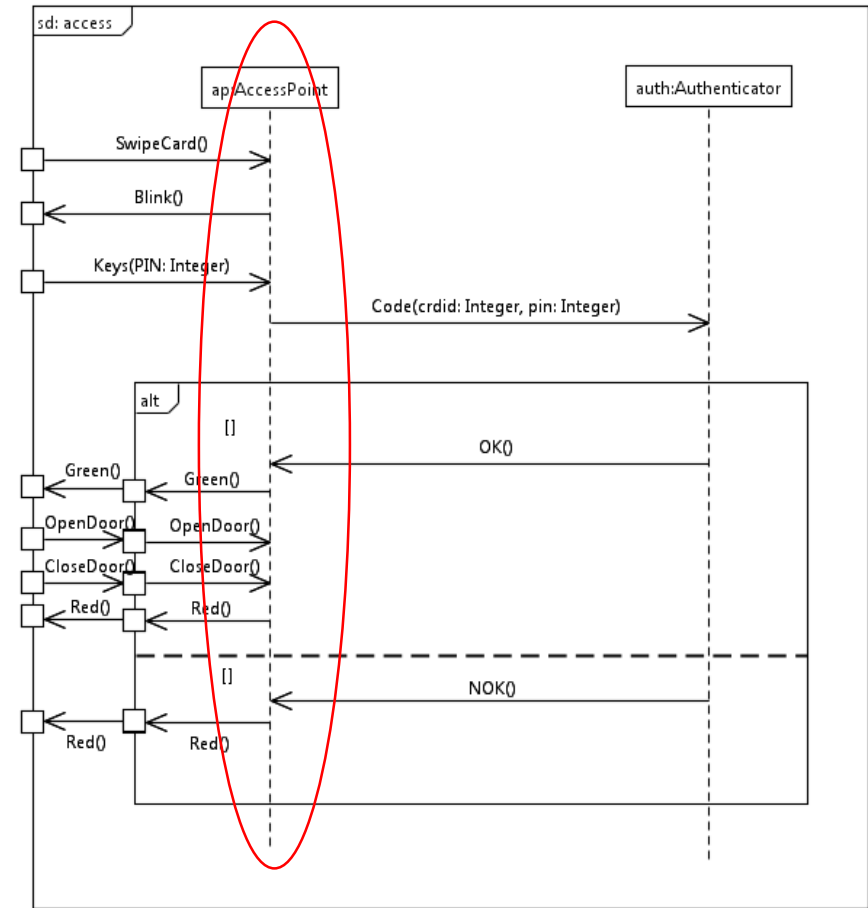
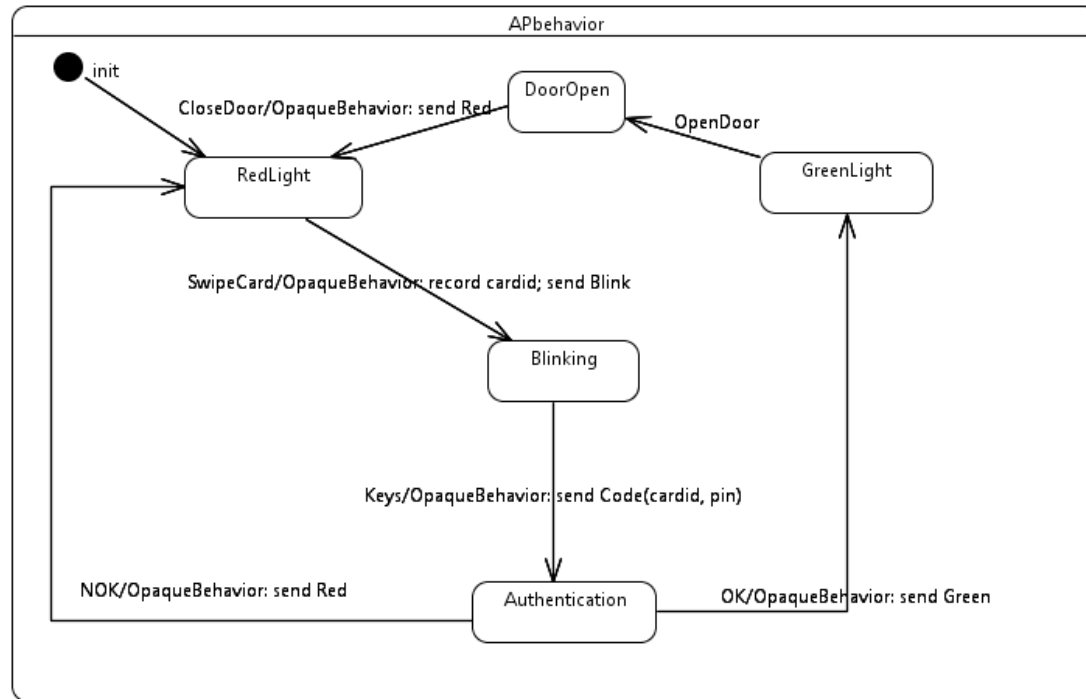


# Design time consistency

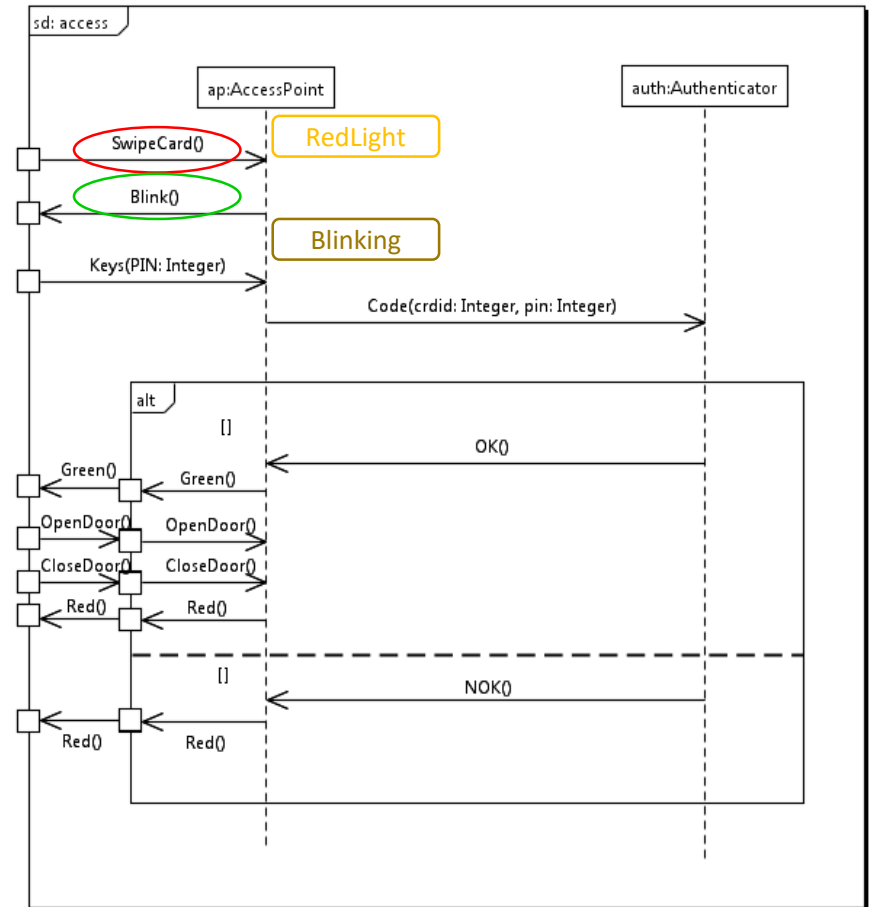
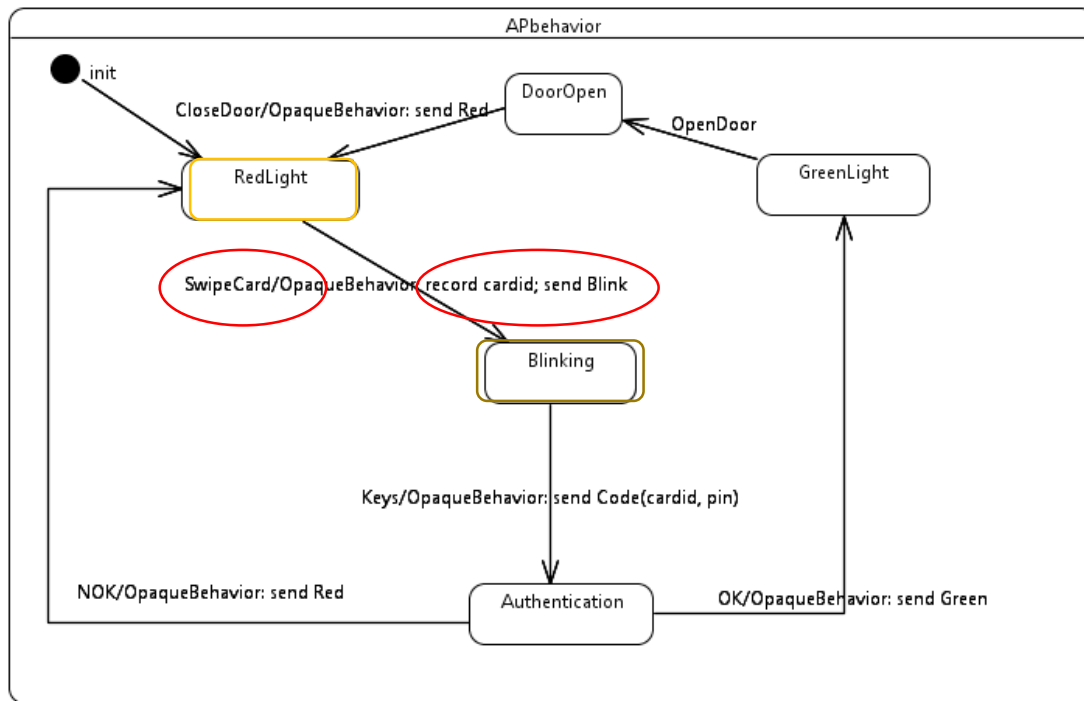
- Can be checked at design time
- Represents structural constraints
- Typically type consistency
  - integer variables can be added, but Boolean variables cannot



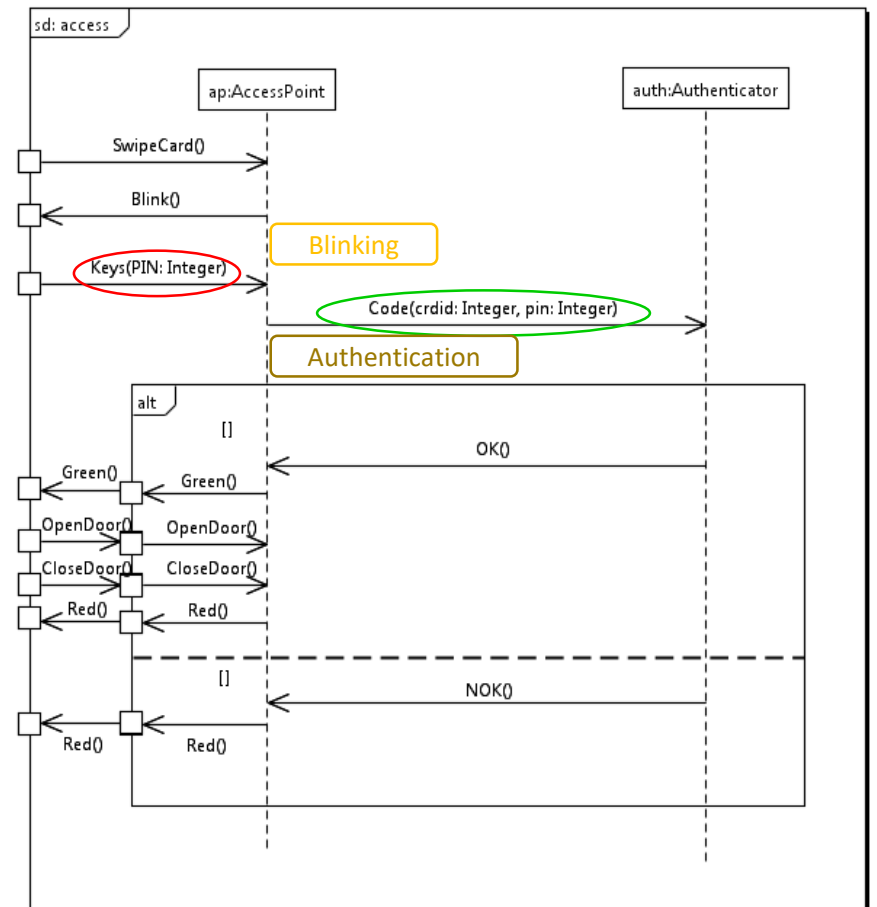
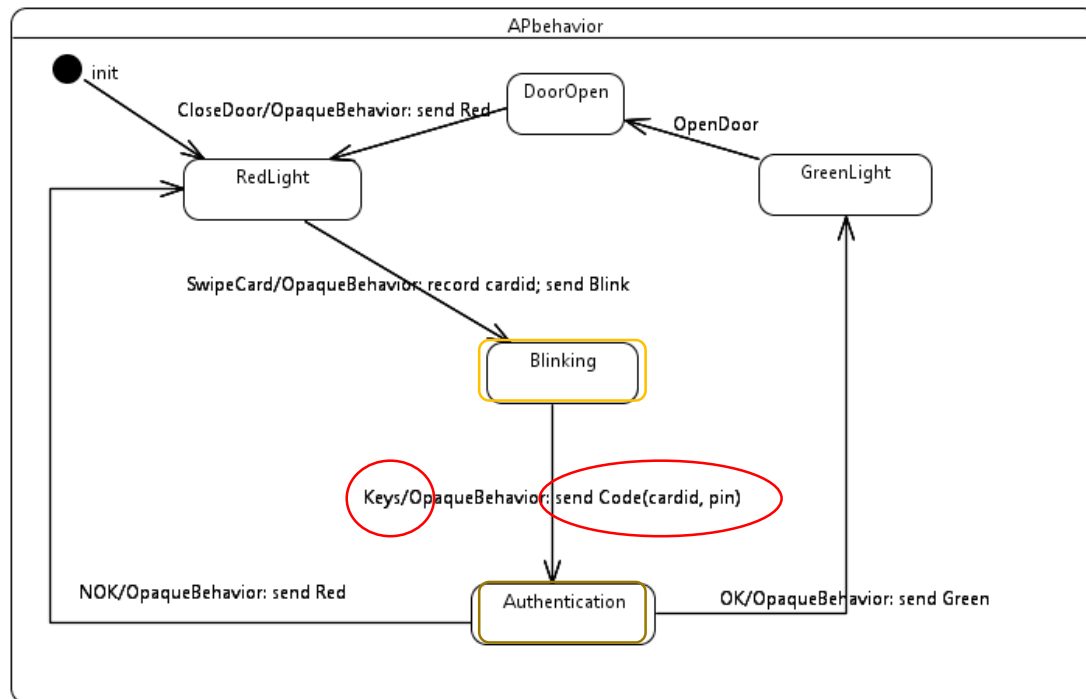
# Runtime consistency – behaviors corresponding



# Let's execute the state machine according to the sequence diagram

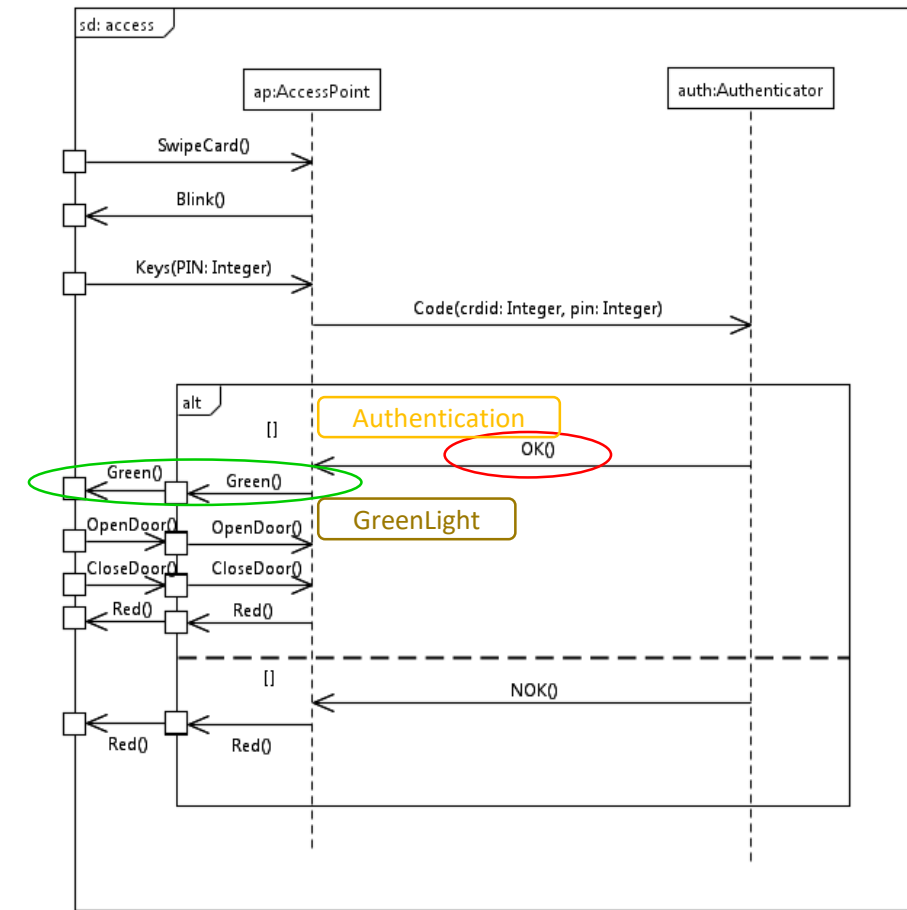
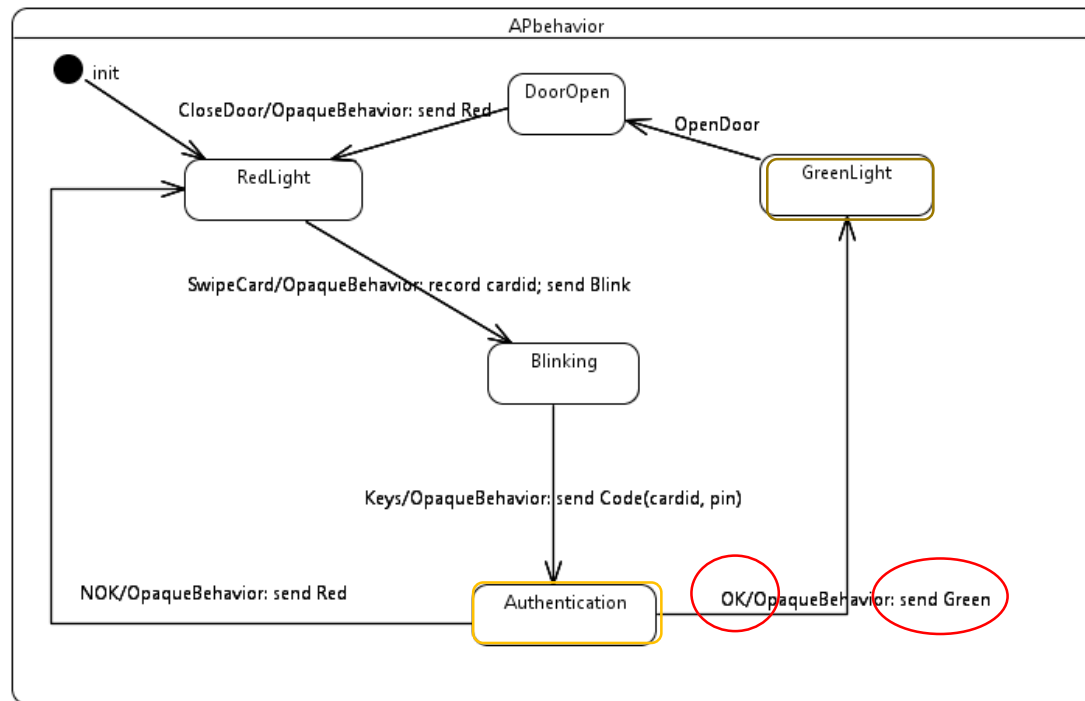


# Play it again, Sam

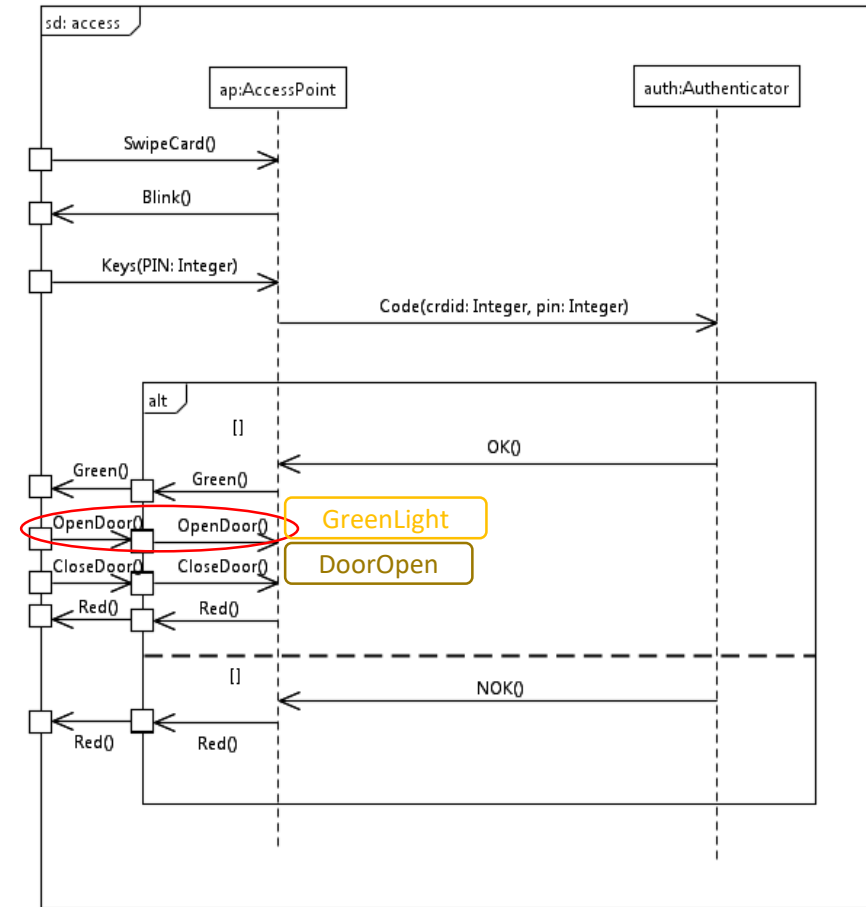
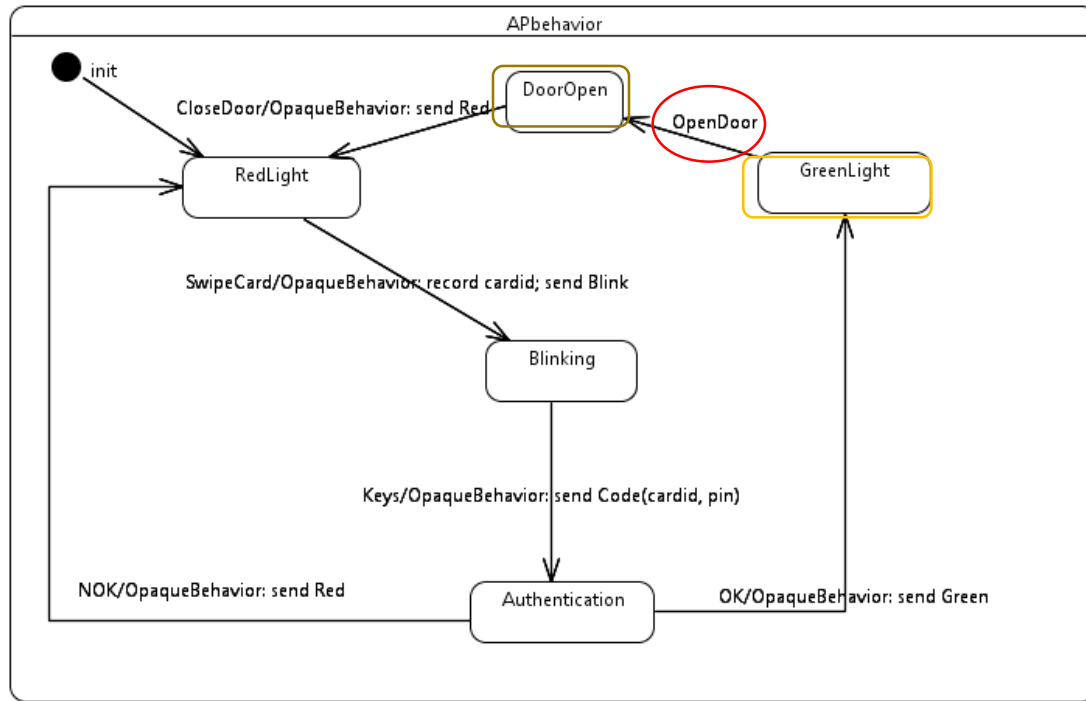




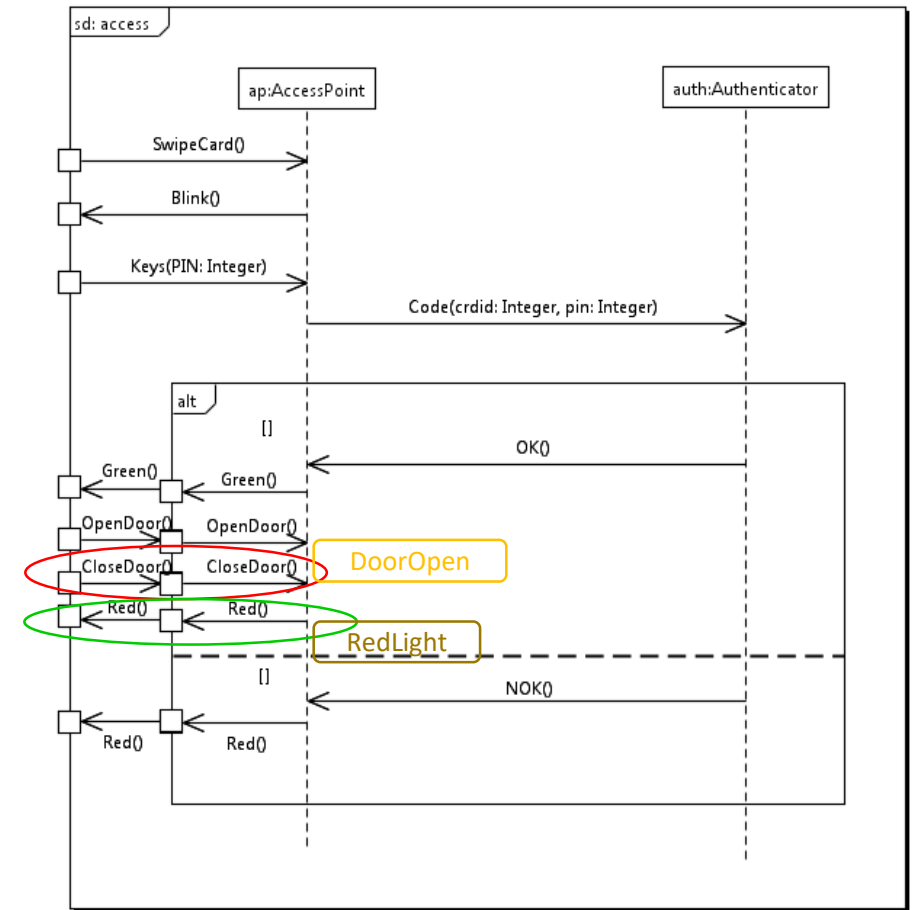
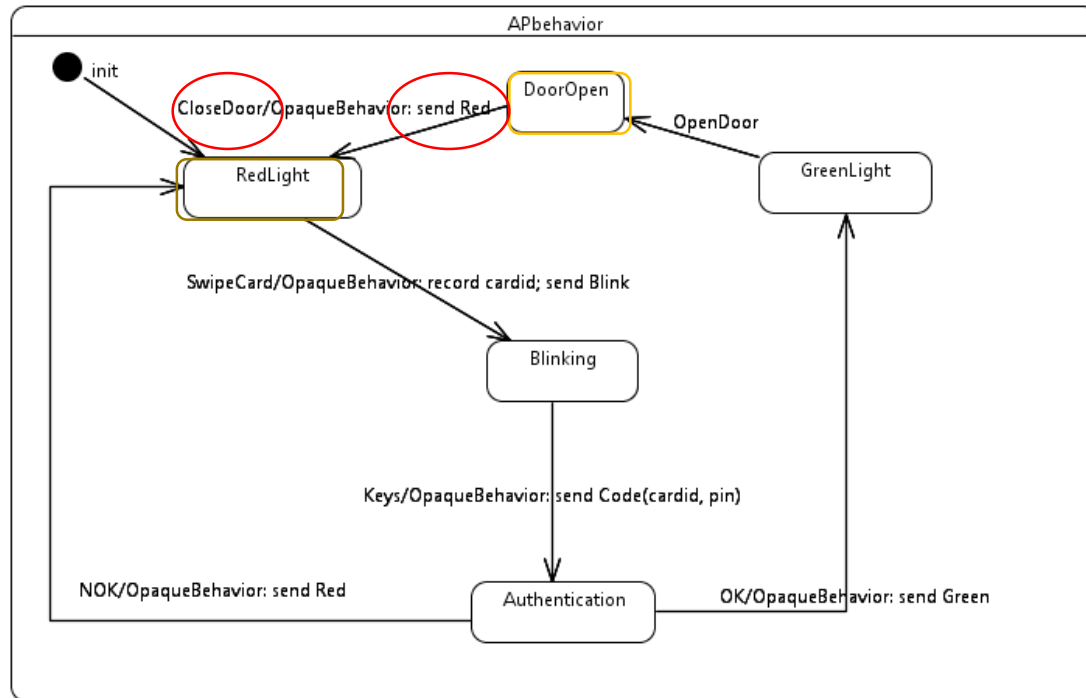
# Access granted (one out of two alternatives)



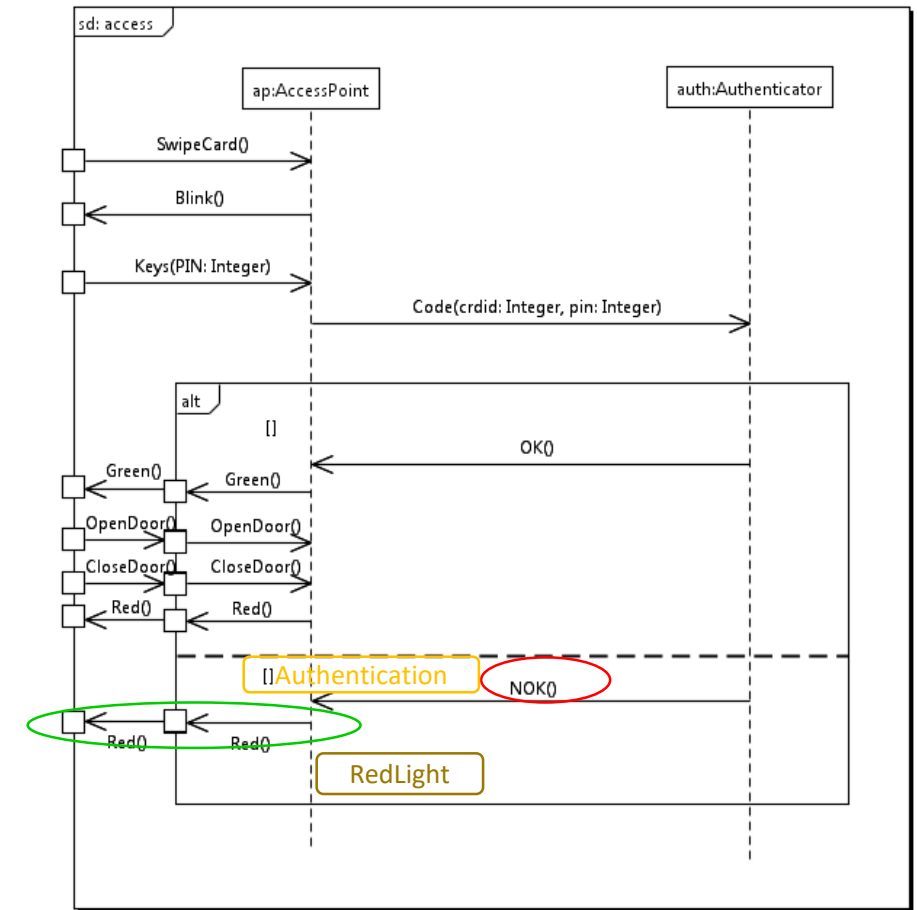
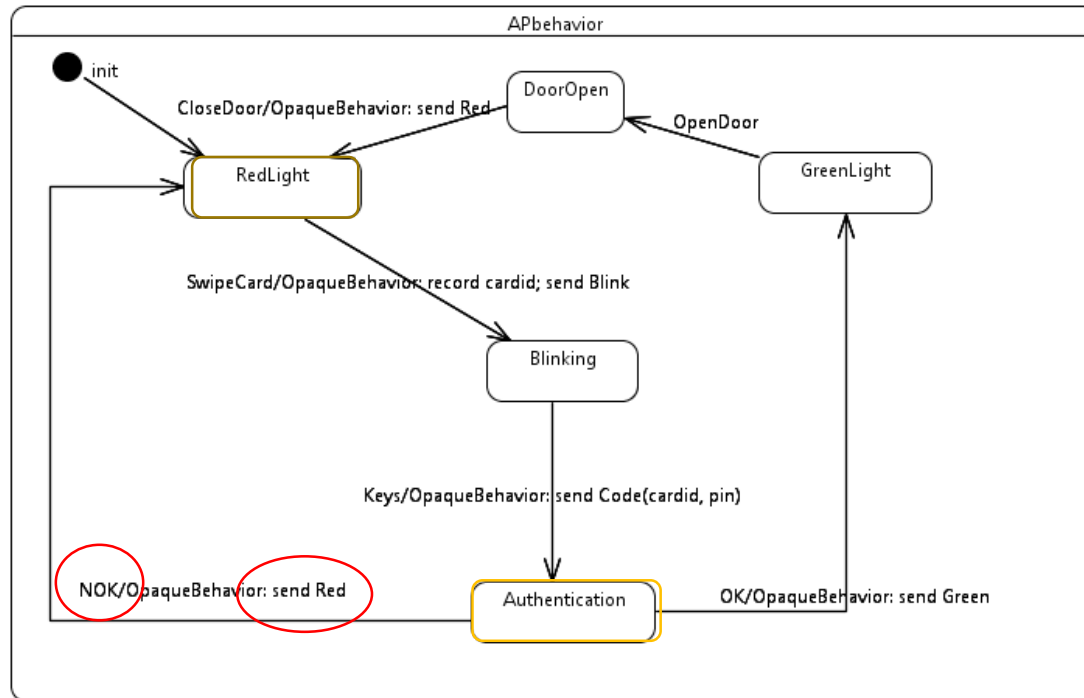
# User opens the door



# User closes the door again



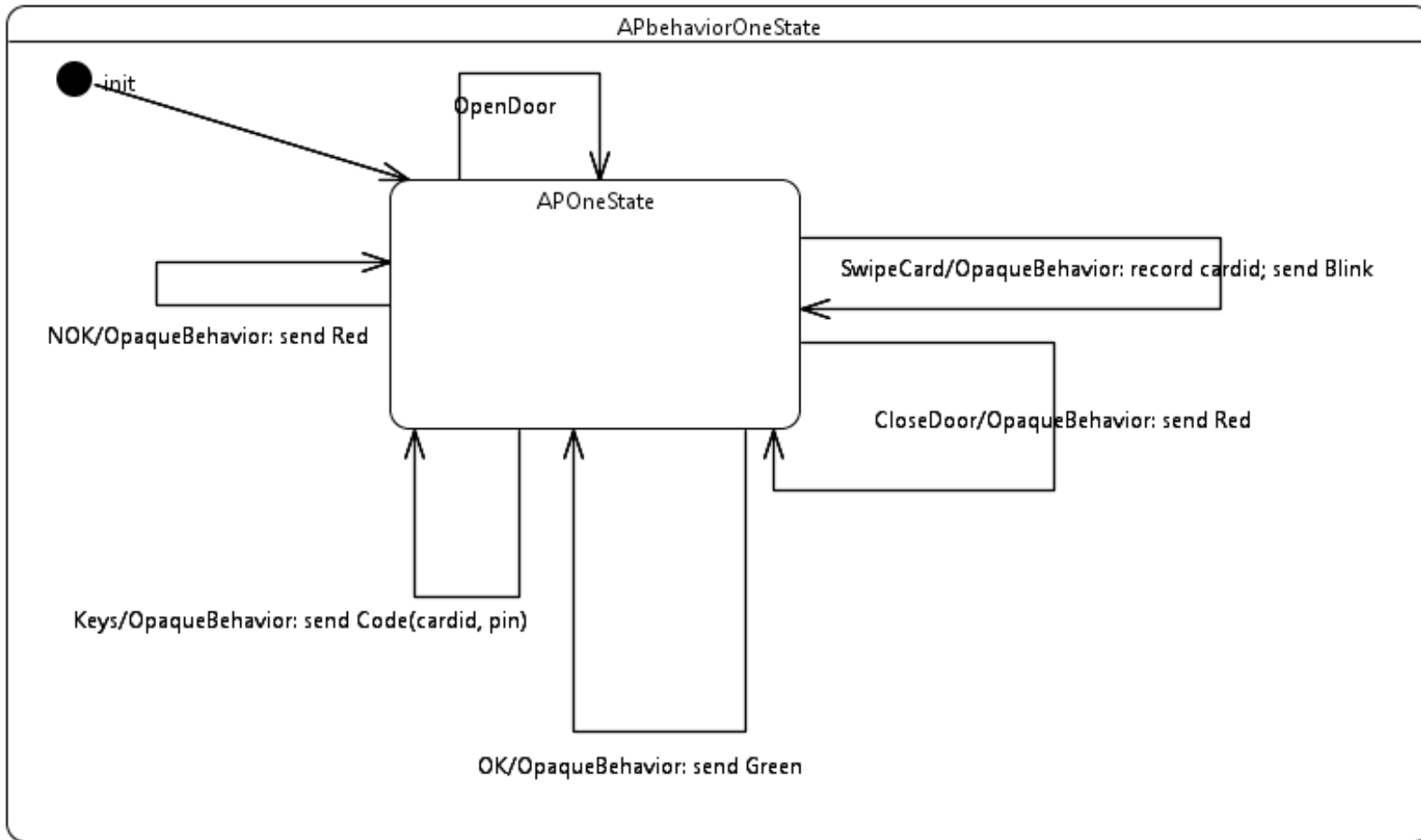
# Access not granted (second of two alternatives)



# Concluding the runtime consistency check

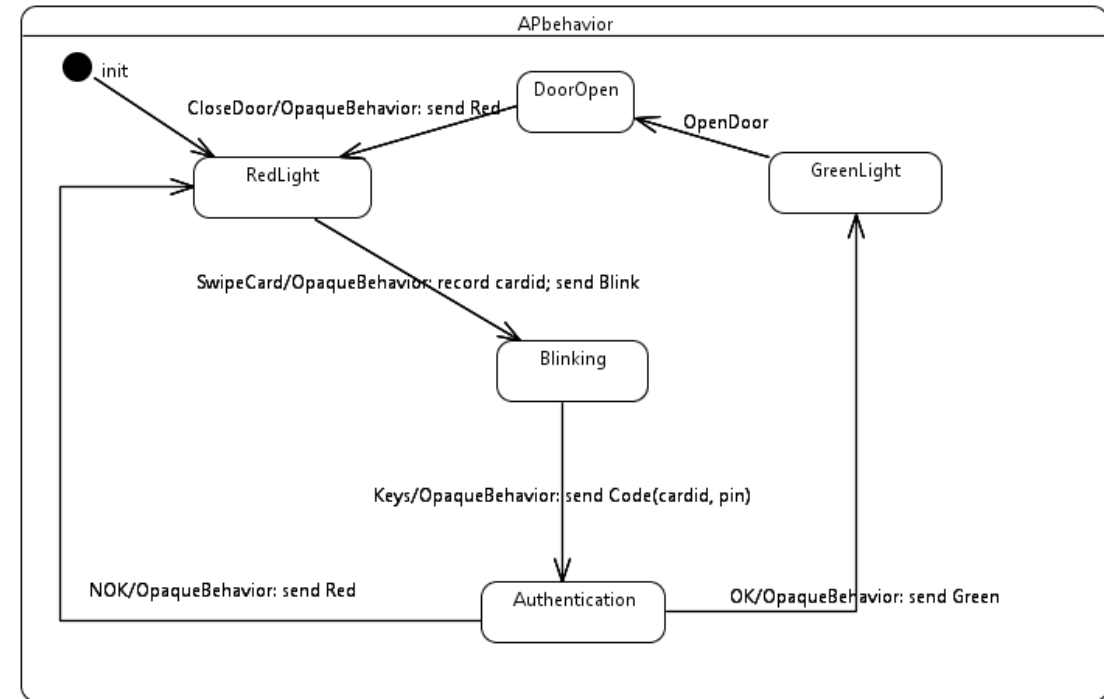
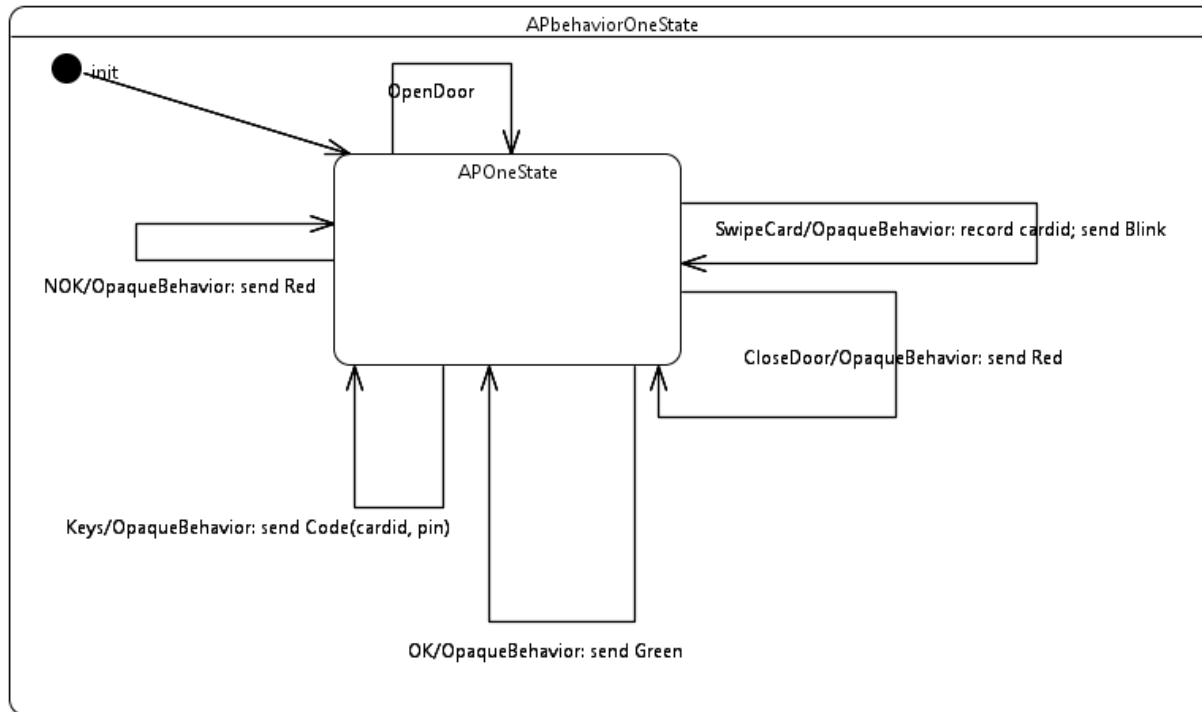
- The APbehavior state machine satisfies all traces of the sequence diagram *access*
- Thus these behaviors are consistent
- Are we then perfectly happy?

# Another attempt to define the state machine





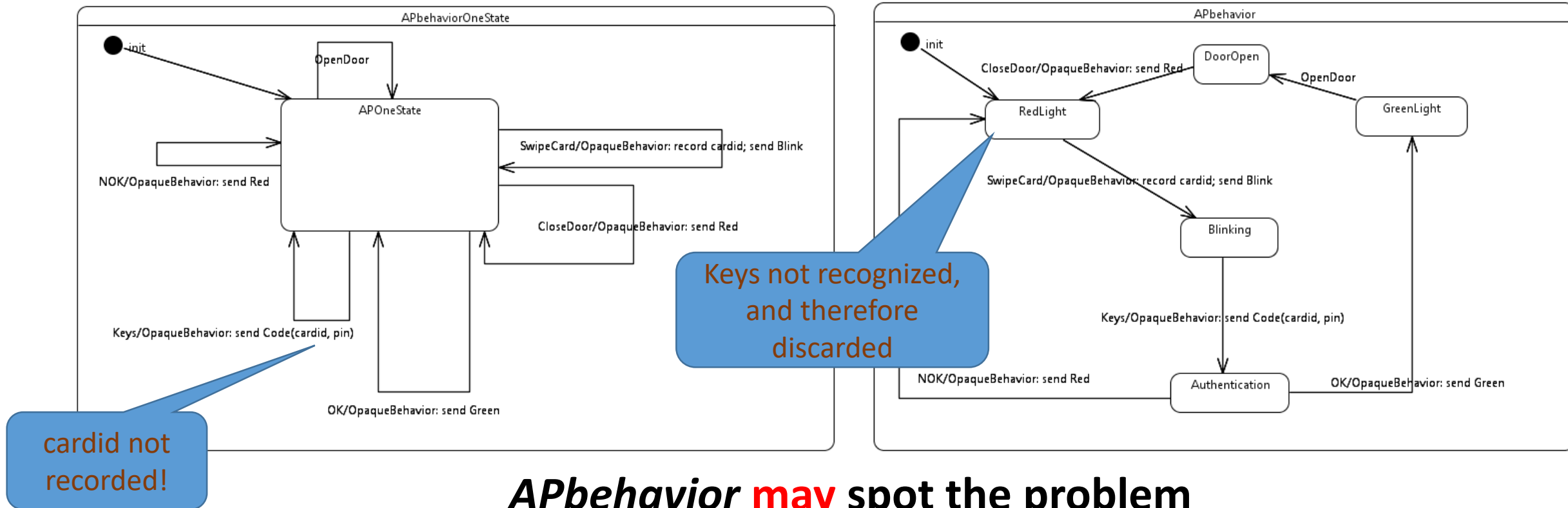
# Which state machine is the better description?



and why?



# What if the user started keying the PIN at once?

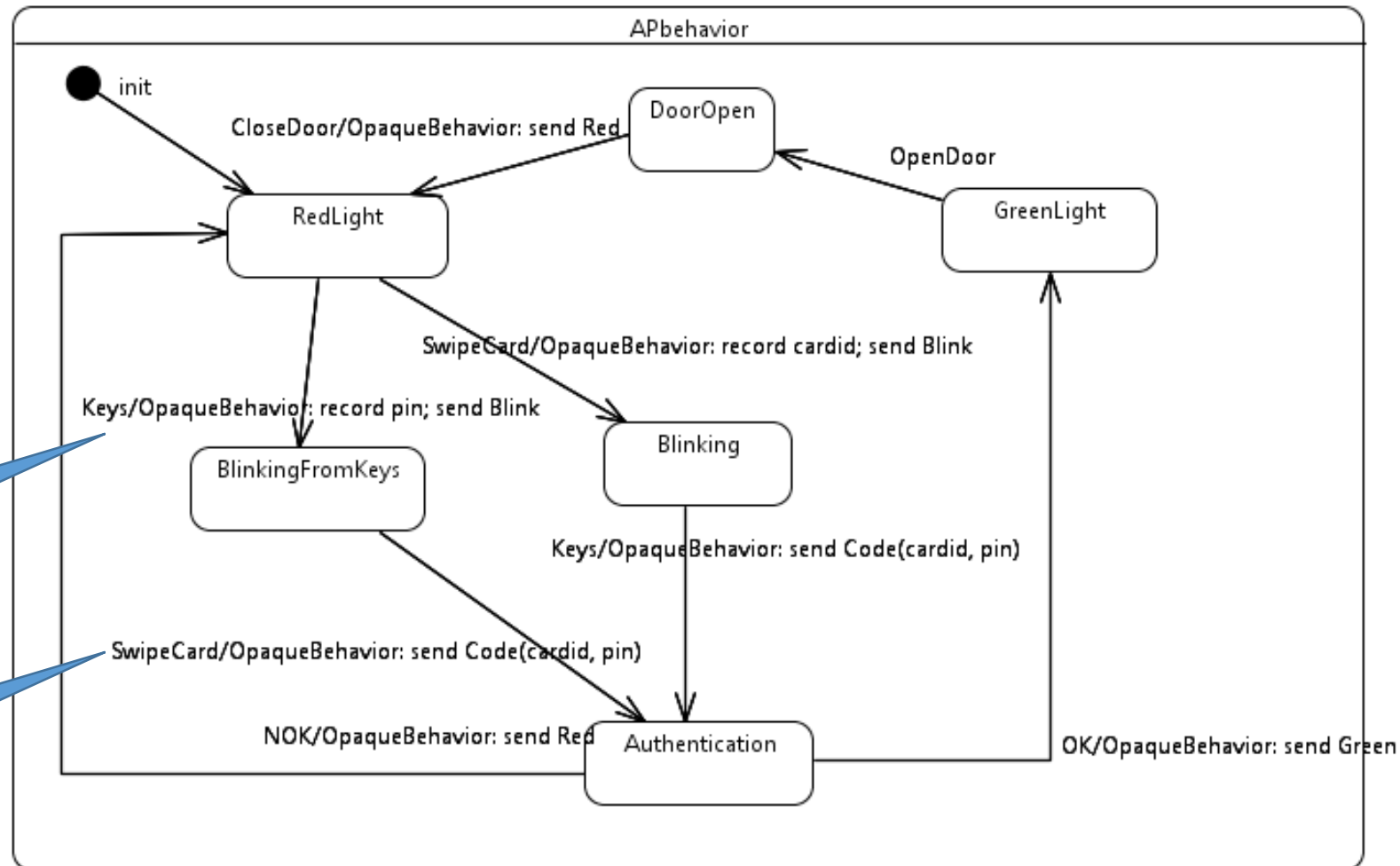


***APbehavior* may spot the problem**  
***APbehaviorOneState* will go on in error**

# Why using different states?

- Several different states distinguishes between different situations
- The same trigger should have different effects in different situations
- A specific state represents in a compact way the whole history of behavior that led to reaching that state

# Slightly more robust and functional AccessPoint



Keys first

Card second

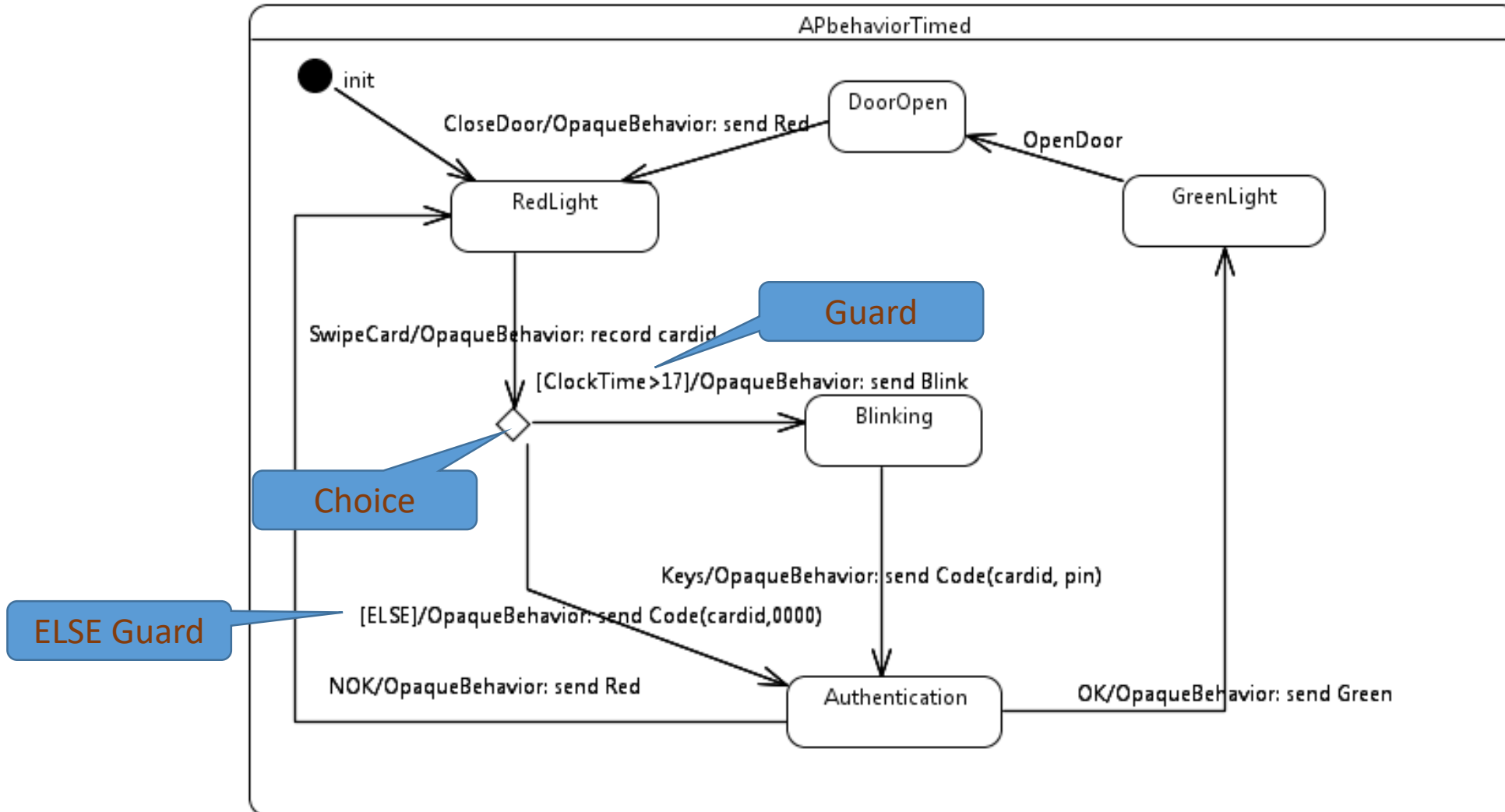
# Guidelines and Reminders

- Even though the state machine was consistent with the sequence diagram, the state machine was flawed
  - The reason was that sequence diagrams are only partial descriptions of the whole, while state machines are complete descriptions of a part of the whole
- Use several states if you can
  - Each state representing a stable, recognizable situation
- We should supplement our state machine with all the possible different transitions
  - This would help us consider and handle most error situations

# What if we need to modify a state machine?

- Our access control system should possibly be acting differently during working hours than at other times
- How well do state machines cope with modifications?

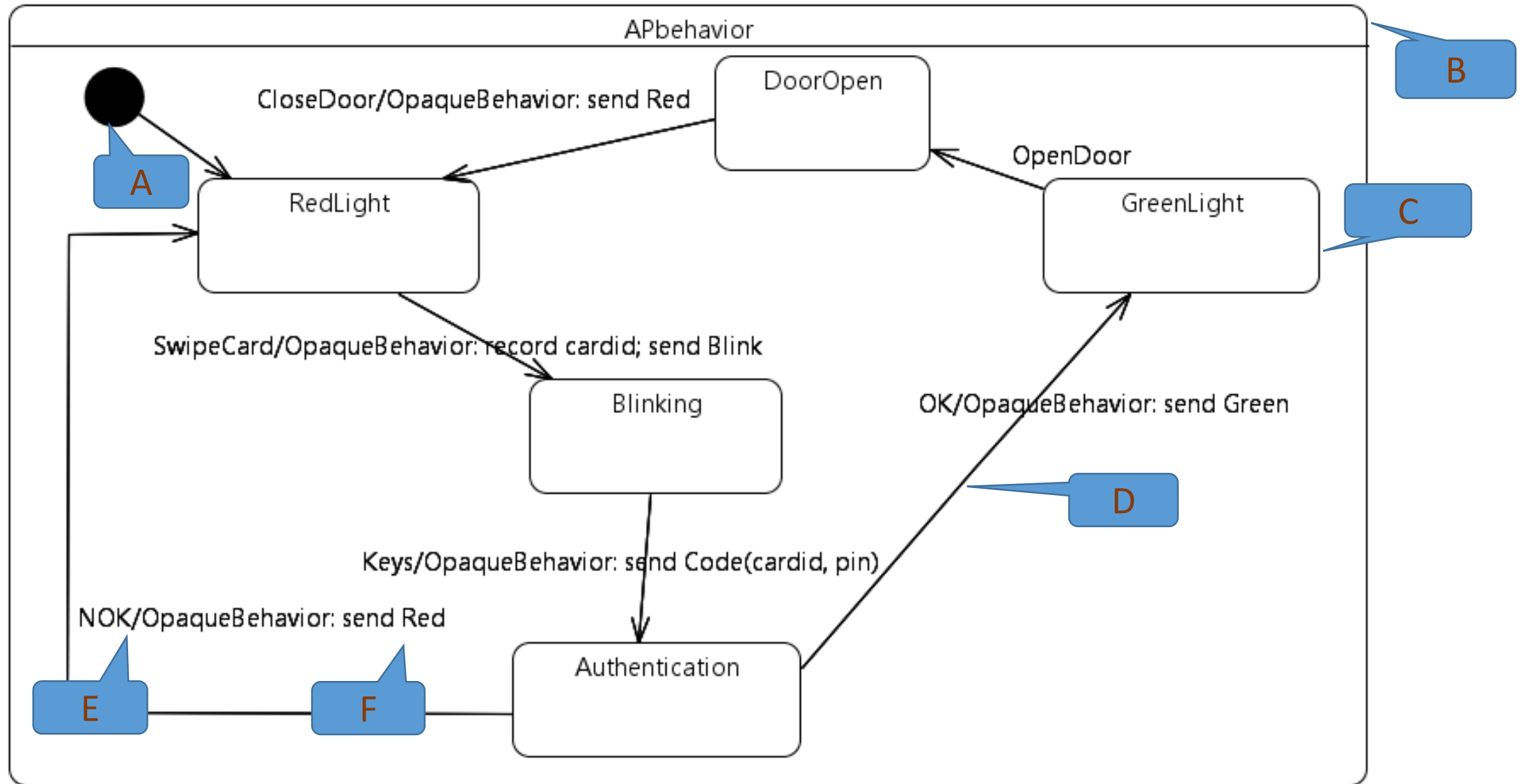
# Enhancing the state machine



# Summarizing

- State machines describe behavior of independently acting components
- Reactive systems are suitable for state machines
- Consistency checks between sequence diagrams and state machines are very useful
  - but not sufficient
- State machines are robust in as much as additional functionality can often be included without ripple effects on other parts of the behavior

# The behavior of the AccessPoint





# The behavior of the AccessPoint

