Object-Oriented Software Design with different Unified Modelling Language (UML) notations

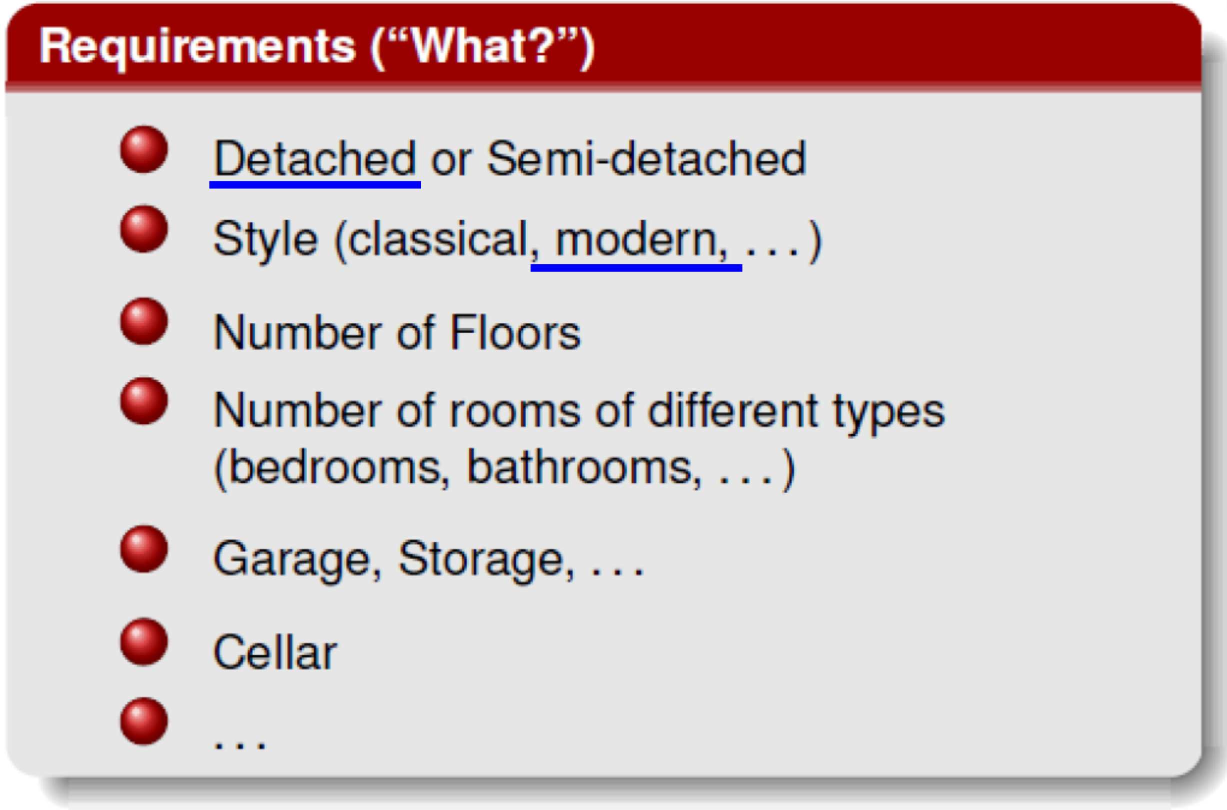
Use Case Notation, Class Diagrams, Object Diagrams, Sequence Diagrams, Regular Expressions and State Automata

Bart Meyers

Hans Vangheluwe

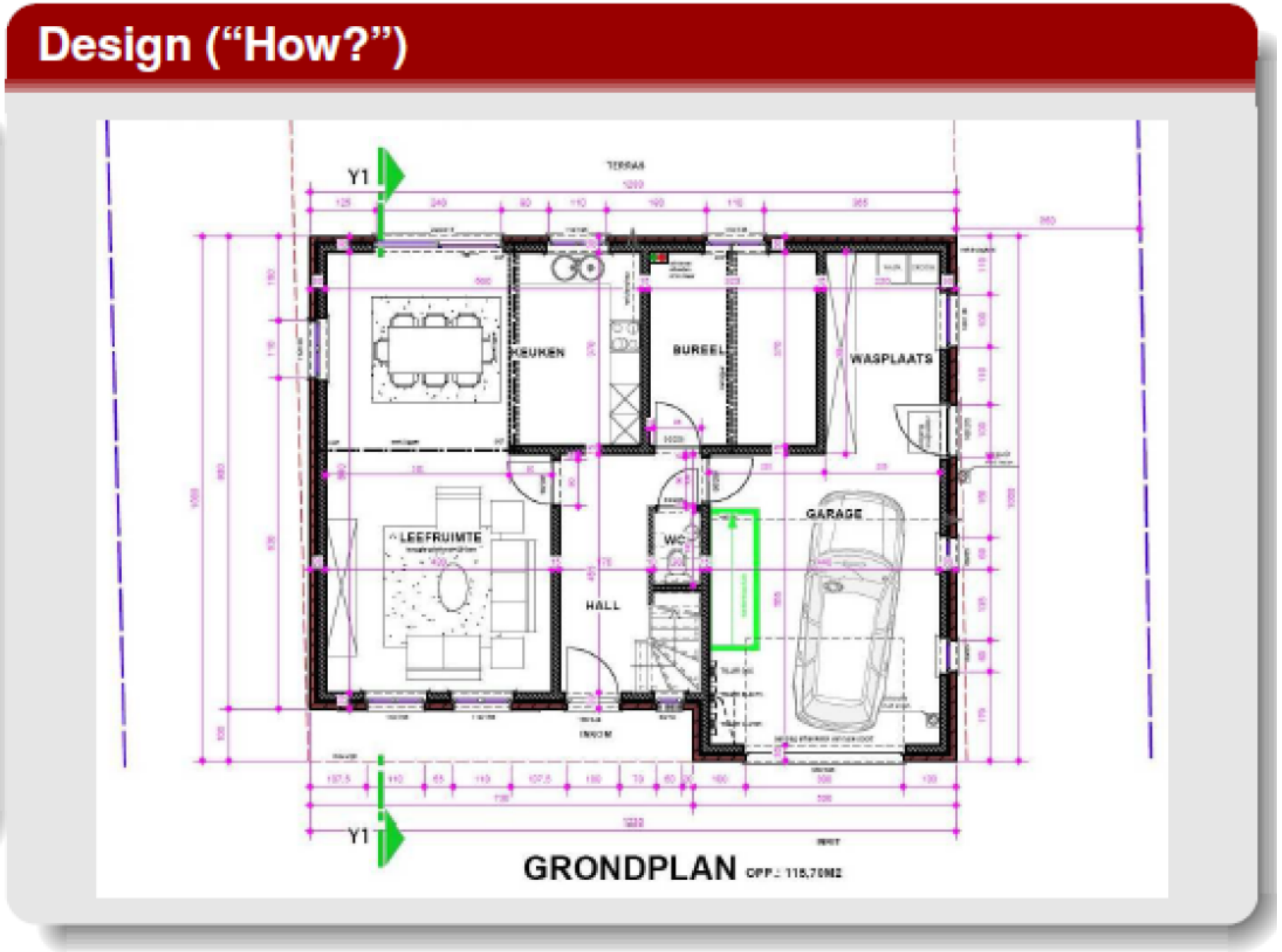
Universiteit Antwerpen
Ansymo
Antwerp Systems & Software Modelling
University of Antwerp
Nexor
Cyber-Physical Systems
University of Antwerp
FLANDERS MAKE
MANUFACTURING INNOVATION NETWORK
McGill
MPM4CPS
cost

What vs. How



Requirements ("What?")

- Detached or Semi-detached
- Style (classical, modern, . . .)
- Number of Floors
- Number of rooms of different types (bedrooms, bathrooms, . . .)
- Garage, Storage, . . .
- Cellar
- . . .

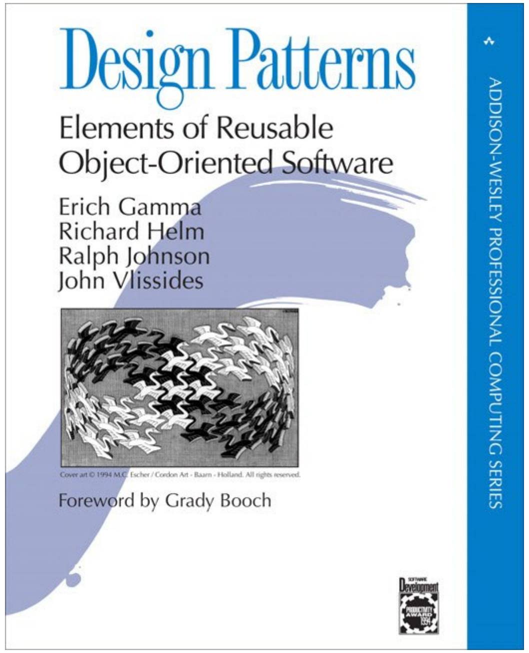


Design ("How?")

requirements
(i.e., a set of properties)

satisfied by → design

(may in turn serve as requirements ...)



Design Patterns

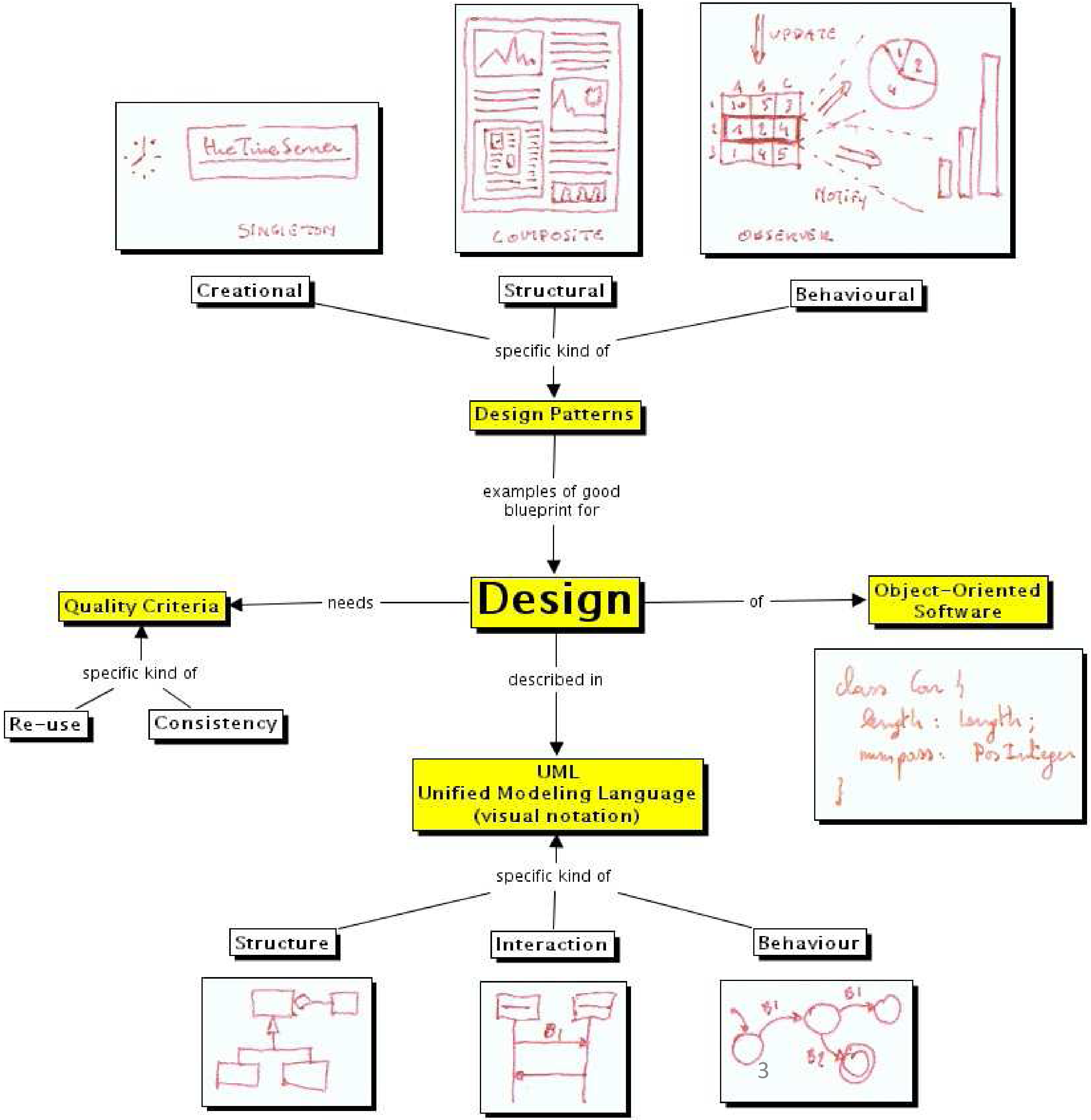
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

UNIFIED MODELING LANGUAGE™ UML®



Creational
Structural
Behavioural
specific kind of
Design Patterns
examples of good blueprint for
Design
needs
Quality Criteria
specific kind of
Re-use
Consistency
of
Object-Oriented Software
described in
UML
Unified Modeling Language
(visual notation)
specific kind of
Structure
Interaction
Behaviour

OO Design Notations (UML) →

Sources/Background Material

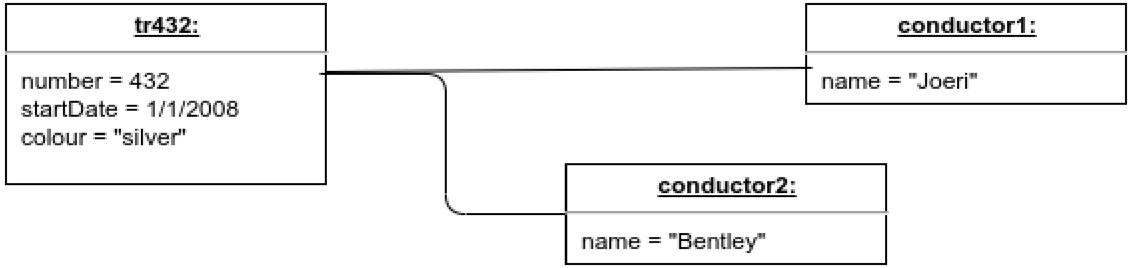
- Use cases: `http://www.cs.mcgill.ca/~joerg/SEL/COMP-533_Handouts_files/COMP-533%204%20Use%20Cases.pdf`
- UML diagram editing: `http://diagrams.net (drawio)`
- Simple UML rendering tool: `http://plantuml.com/`
- Class diagrams: `http://www.uml-diagrams.org/class-diagrams-overview.html`
- Sequence diagrams: `http://www.uml-diagrams.org/sequence-diagrams.html`
- Regular expressions: `http://www.zytrax.com/tech/web/regex.htm`
- Test Regular Expressions online: `https://regex101.com/`
- FSA: `http://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html`

Use Cases

- Document functional requirements of the system
    - interactions between system and **environments** to achieve **user goals**
- Understandable for (non-technical) client
    - semi-formal
    - complete, consistent and verifiable
- Business viewpoint
    - **who** (actors) does **what** (interaction) with **what purpose** (goal)?
    - no implementation details: **black box**
- See Joerg Kienzle and Shane Sendall's presentation
    - (slide 8 -> …)

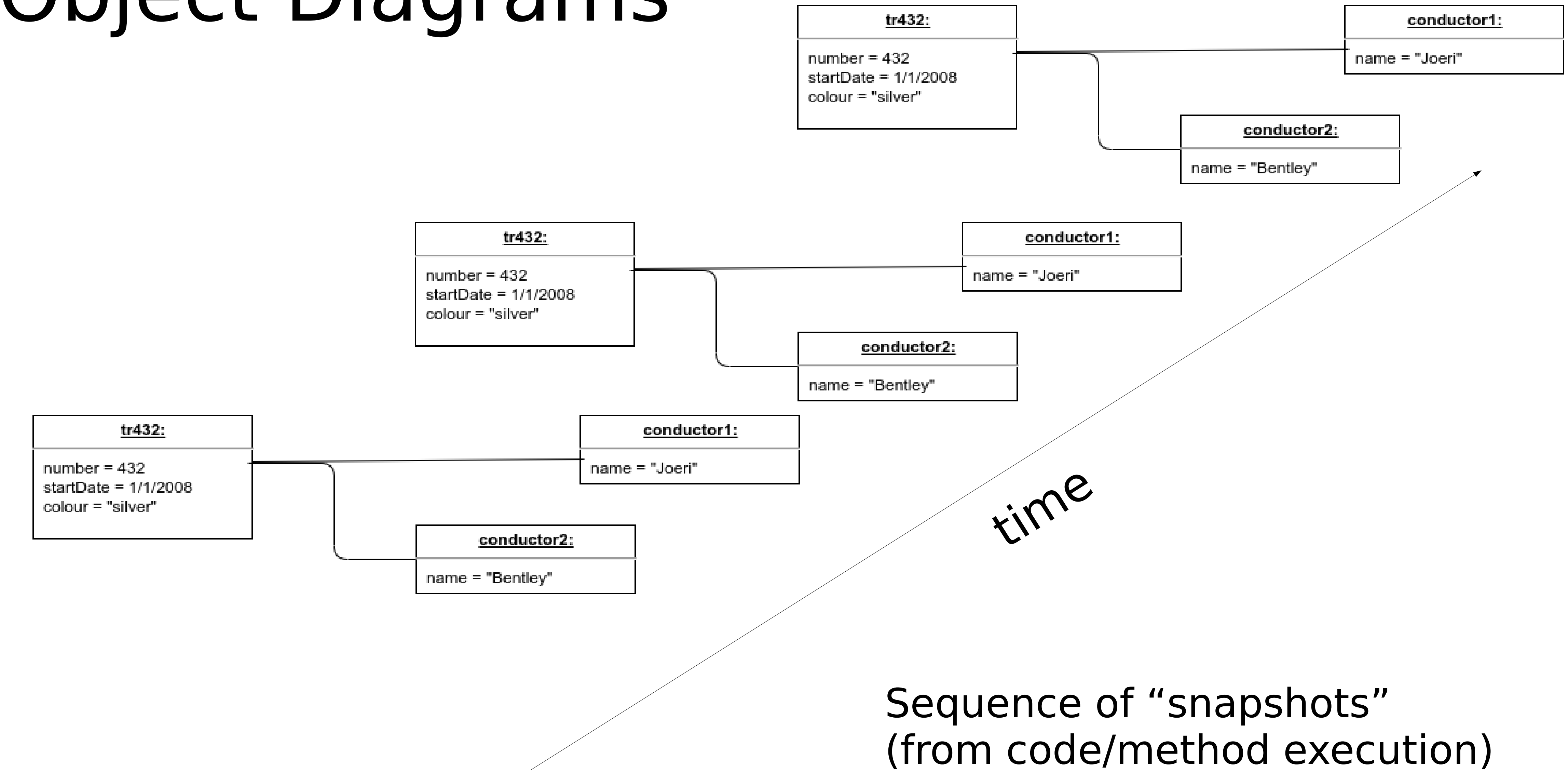
`http://www.cs.mcgill.ca/~joerg/SEL/COMP-533_Handouts_files/COMP-533%204%20Use%20Cases.pdf`

Object Diagrams



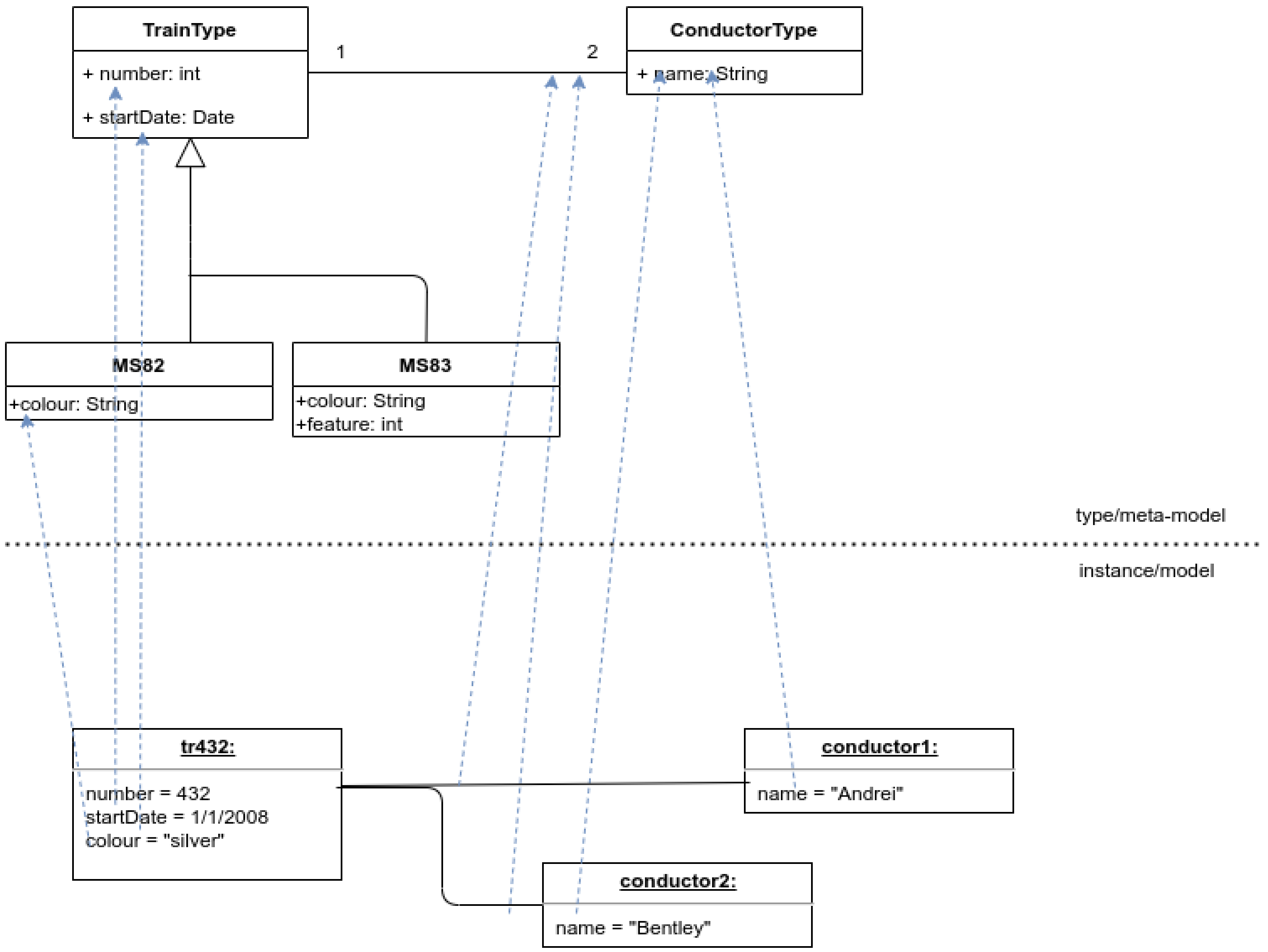
“snapshot”

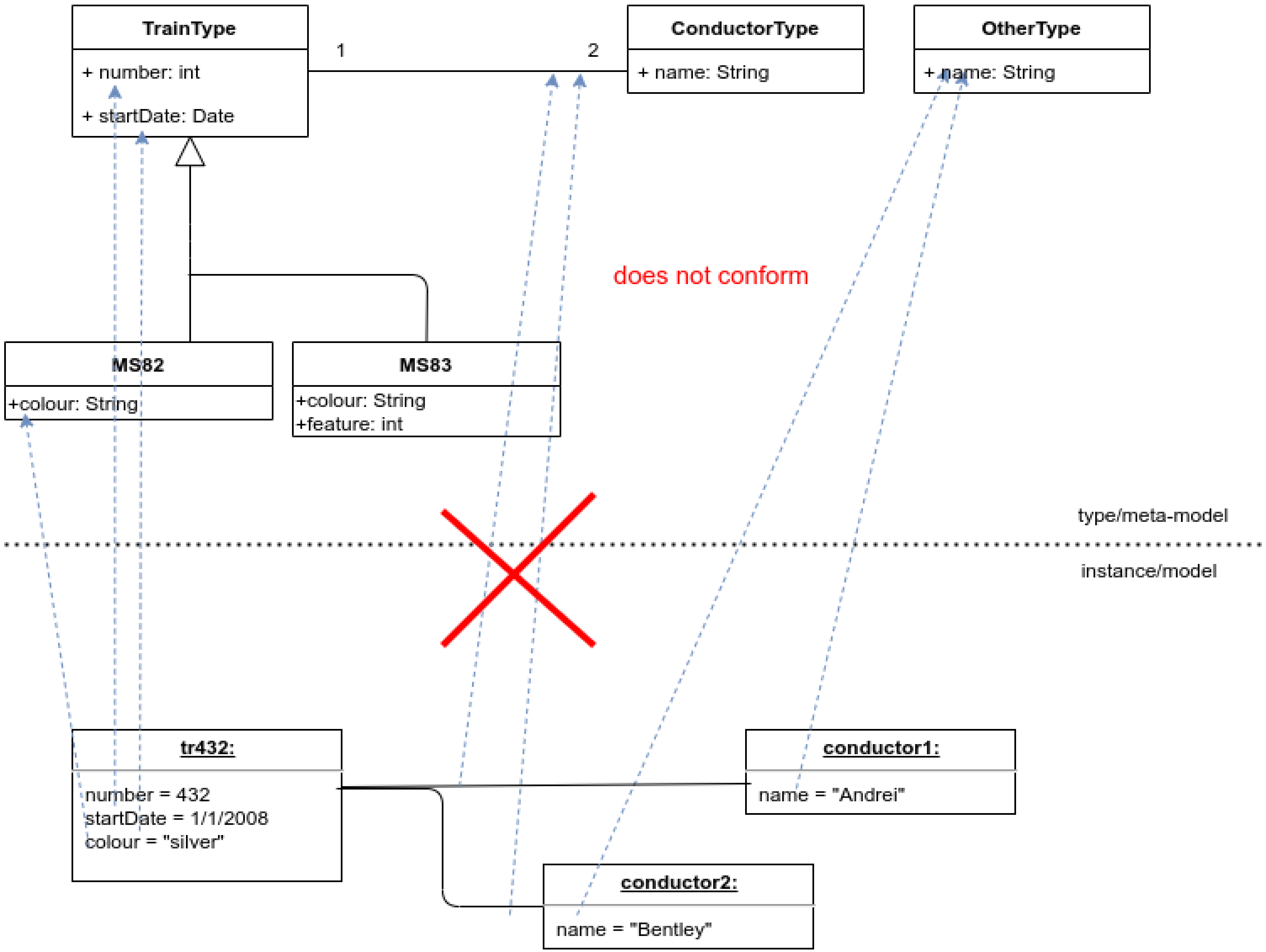
Object Diagrams

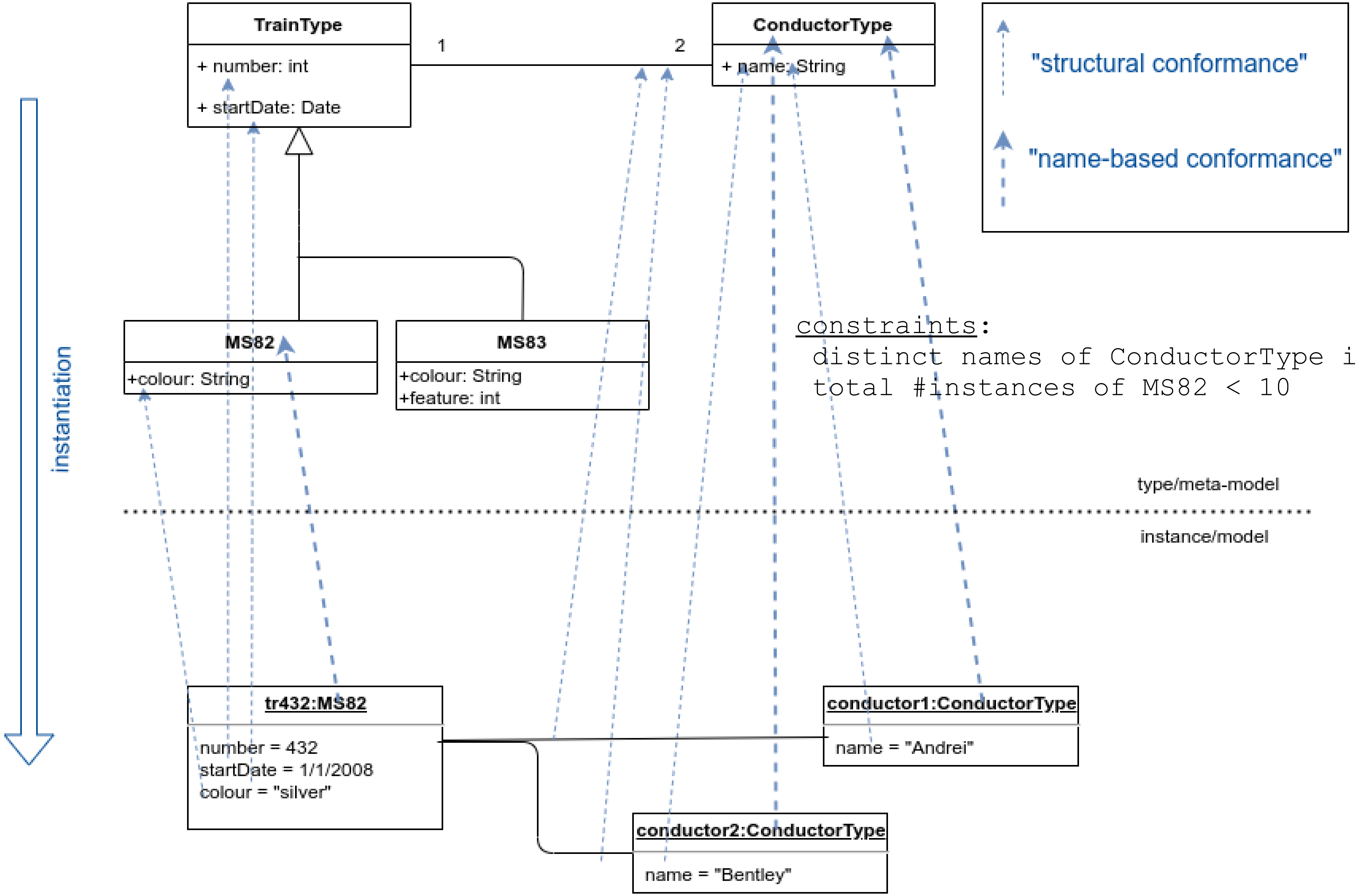


Sequence of “snapshots”
(from code/method execution)

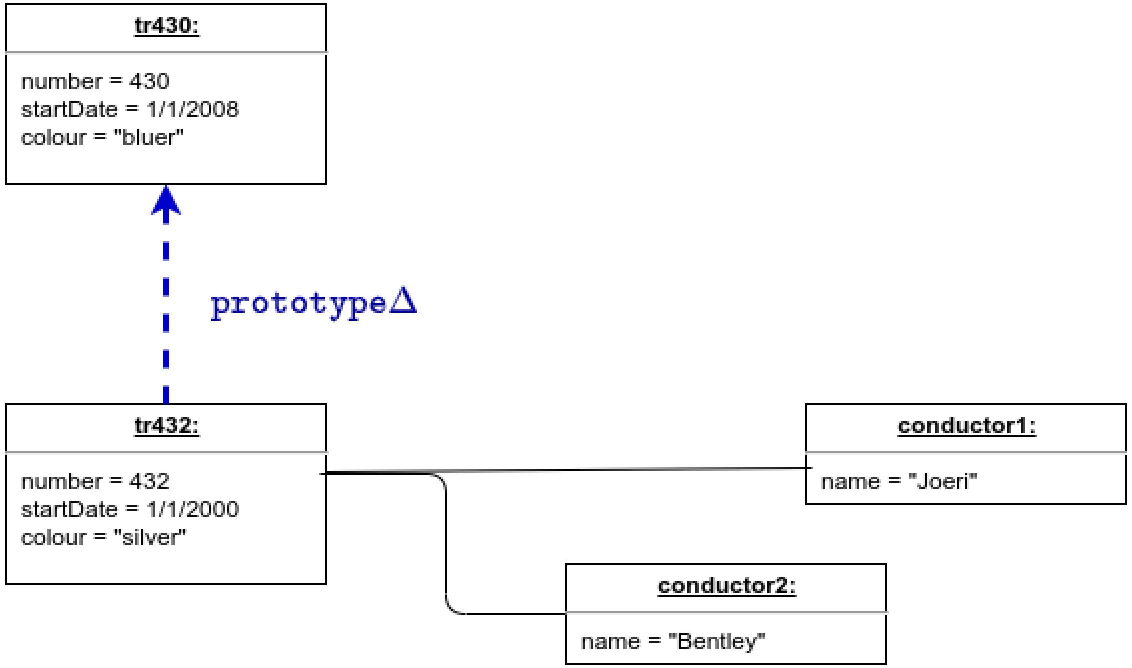
Class Diagrams




TrainType
+ number: int
+ startDate: Date
1
2
ConductorType
+ name: String
OtherType
+ name: String
does not conform
MS82
+colour: String
MS83
+colour: String
+feature: int
type/meta-model
instance/model
tr432:
number = 432
startDate = 1/1/2008
colour = "silver"
conductor1:
name = "Andrei"
conductor2:
name = "Bentley"



constraints:
distinct names of ConductorType instances
total #instances of MS82 < 10



tr430:

number = 430
startDate = 1/1/2008
colour = "bluer"

`prototypeΔ`

tr432:

number = 432
startDate = 1/1/2000
colour = "silver"

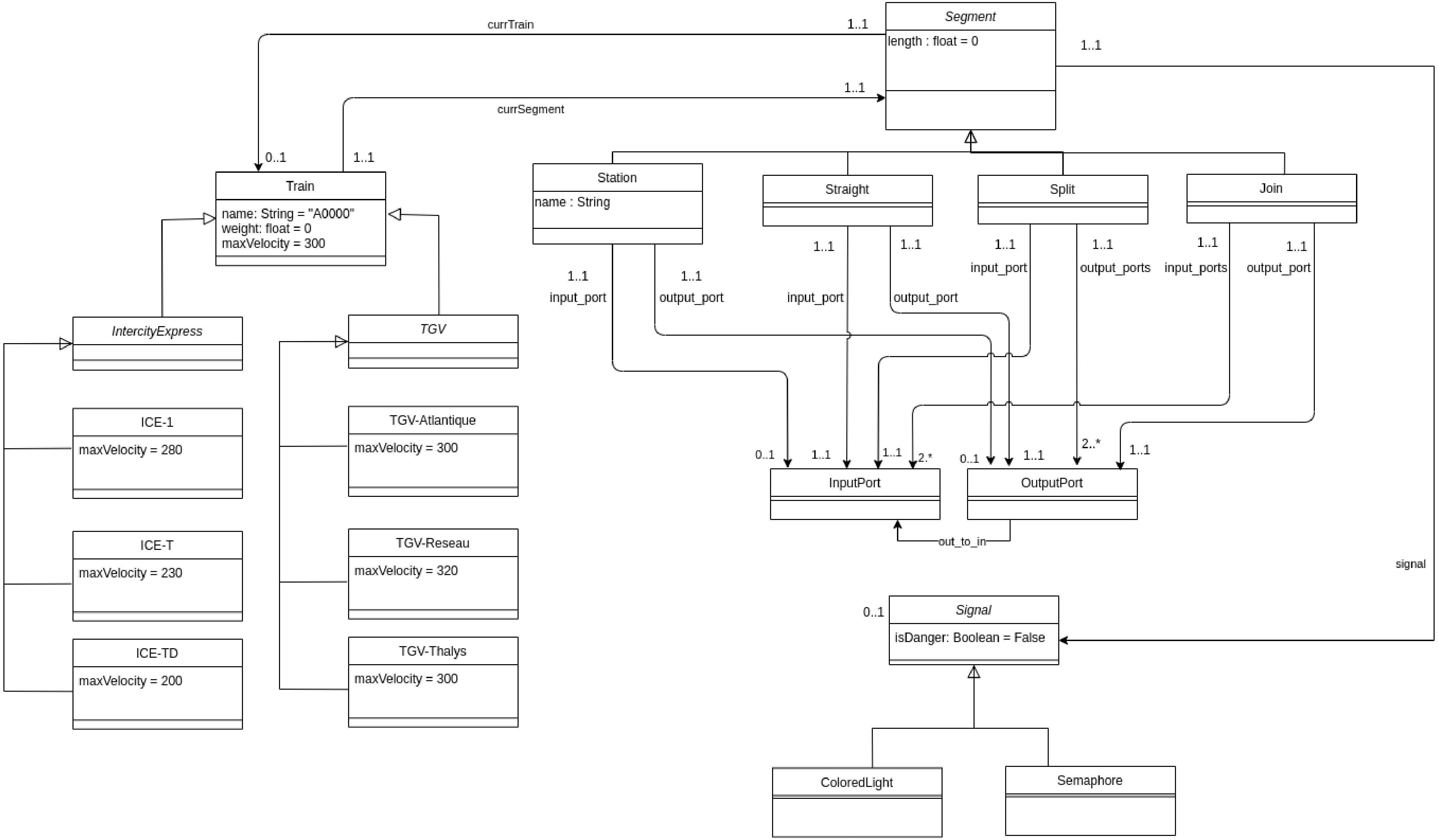
conductor1:

name = "Joeri"

conductor2:

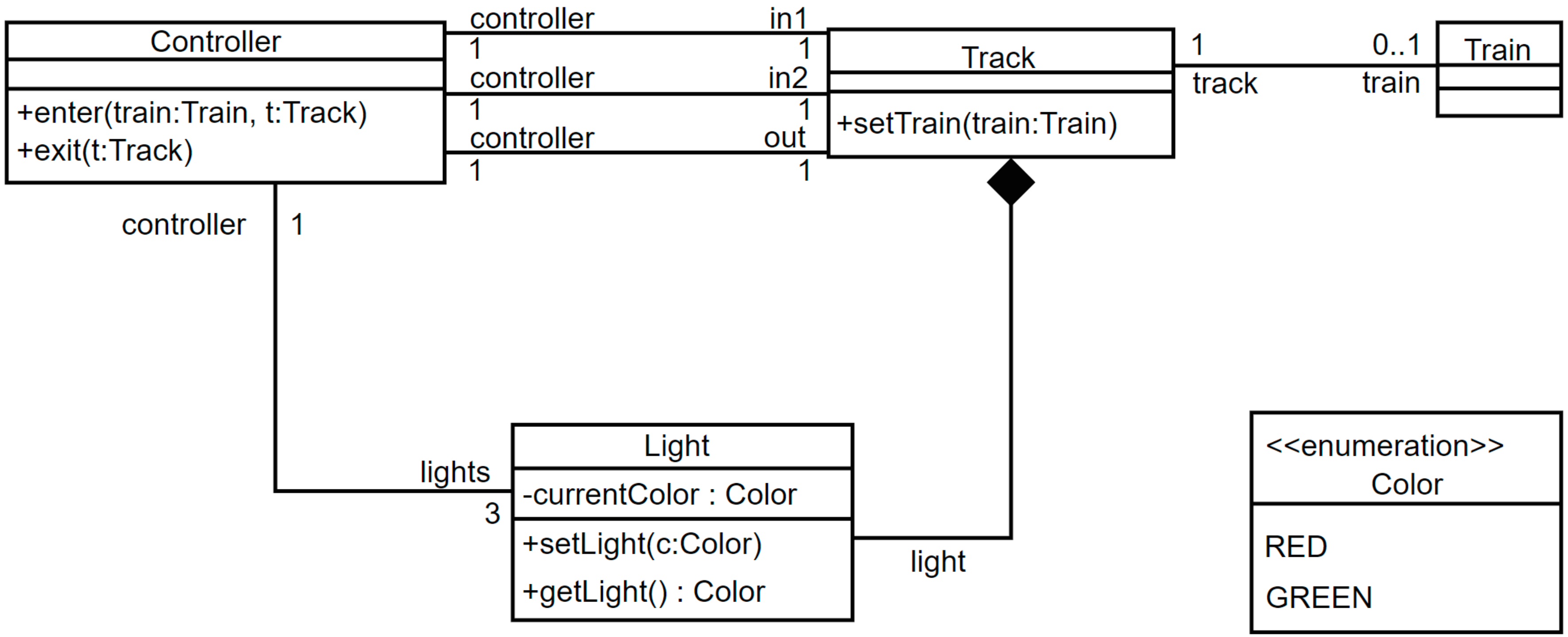
name = "Bentley"

Railway Network Class Diagram



Class Diagrams

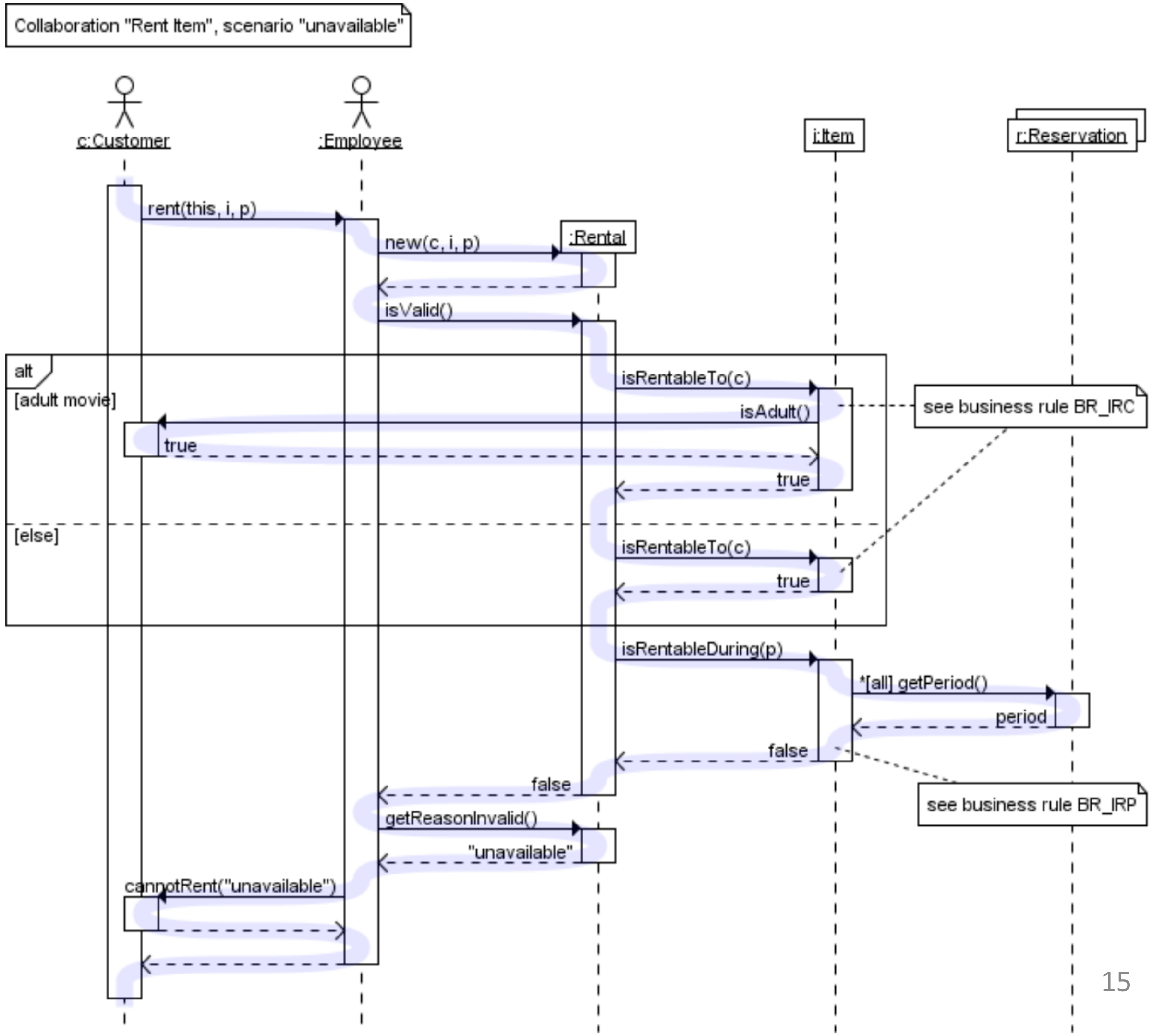
- Structure of the system + behaviour (Object-Oriented)
  - instantiation: Object Diagrams



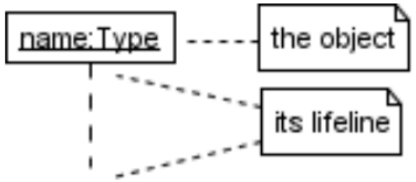
Sequence Diagrams

- Behaviour of the system
    - Interaction diagram
- Complementary to Class Diagrams (structure vs. behaviour)
- Complementary to Use Cases (“what?” vs. “how?”)

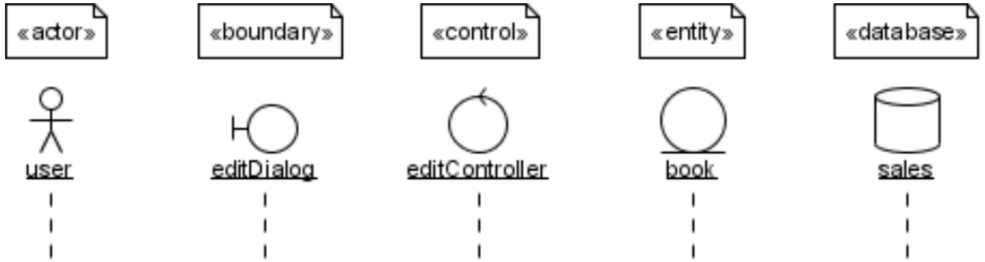
Sequence Diagrams

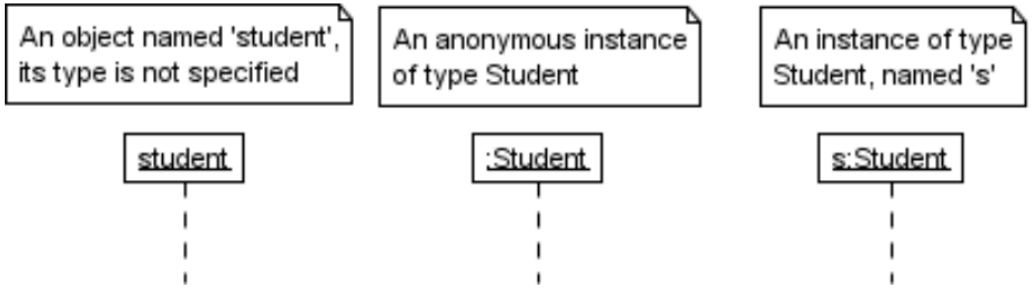


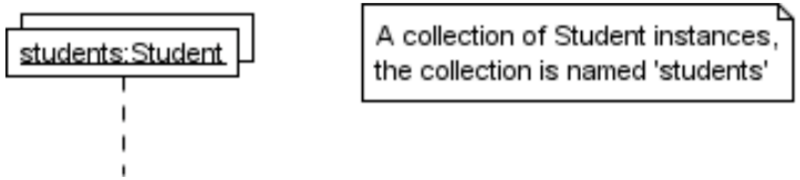
Objects and Lifelines

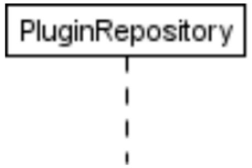


types of objects:





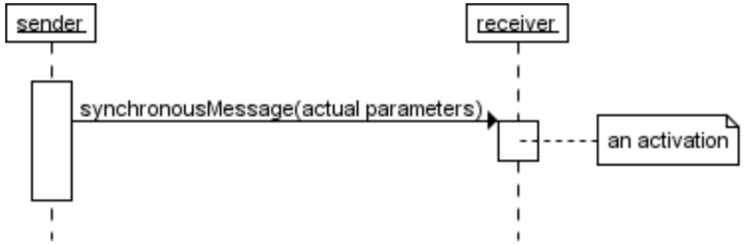




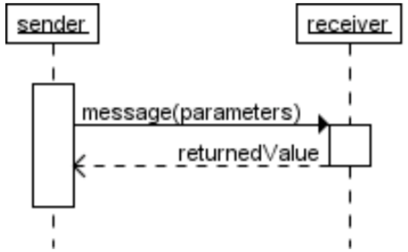
class object
(only use class messages)

Messages (1)

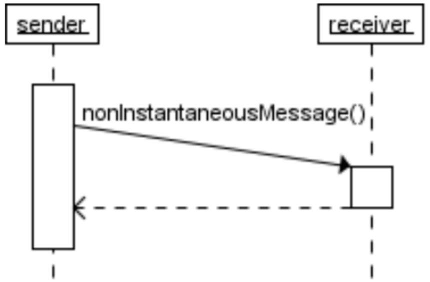
- Synchronous:



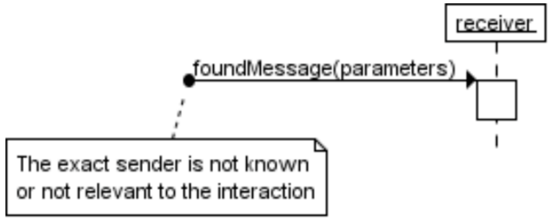
- Returned value:



- Not instantantaneous:

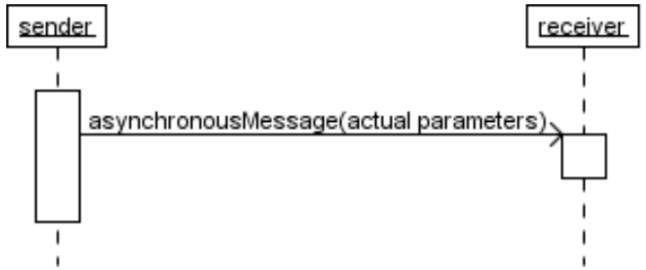


- Found message:

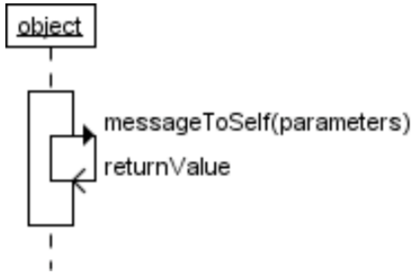


Messages (2)

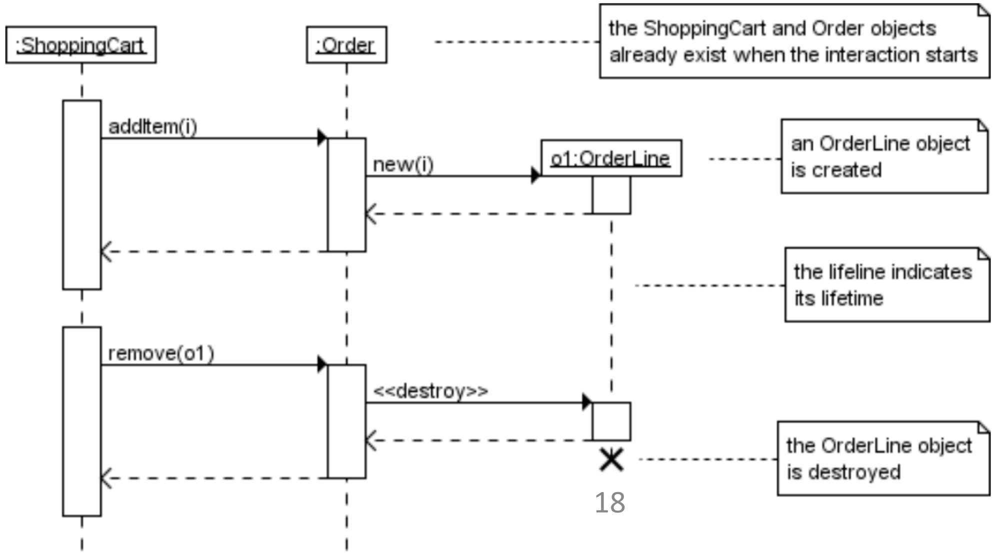
- Asynchronous:



- Message to self:



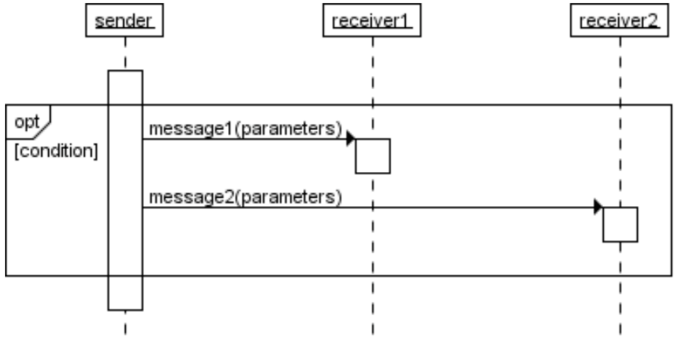
- Object creation/destruction:



Conditional Interaction

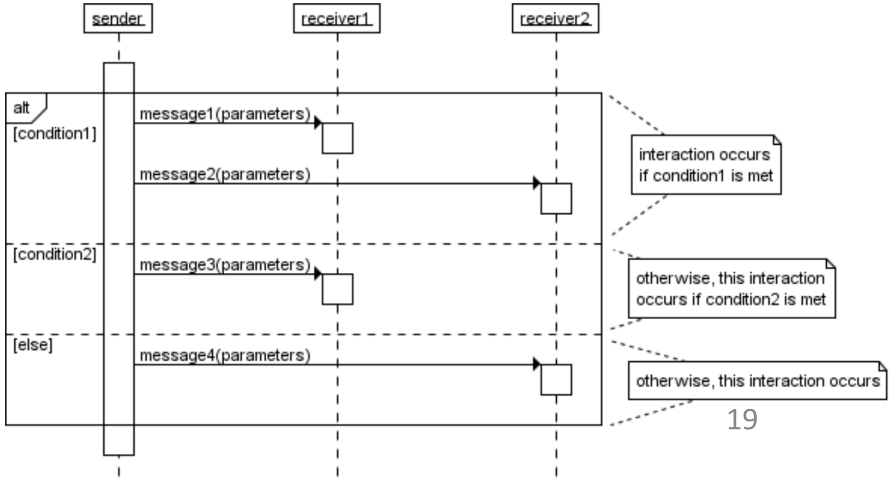
- Message with guard:

- Multiple messages:



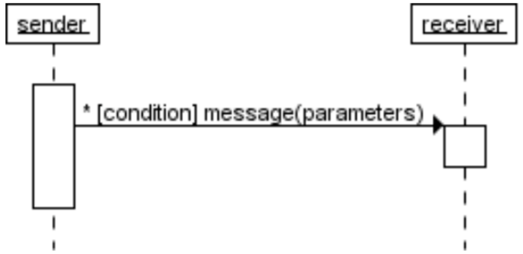
"combined fragment"

- Alternative interactions:



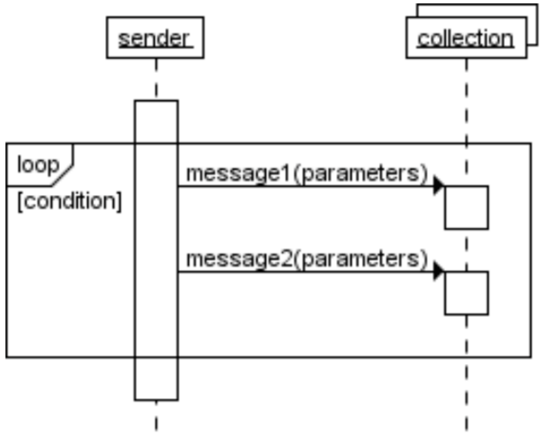
Repeated Interaction (1)

- Repeated message:



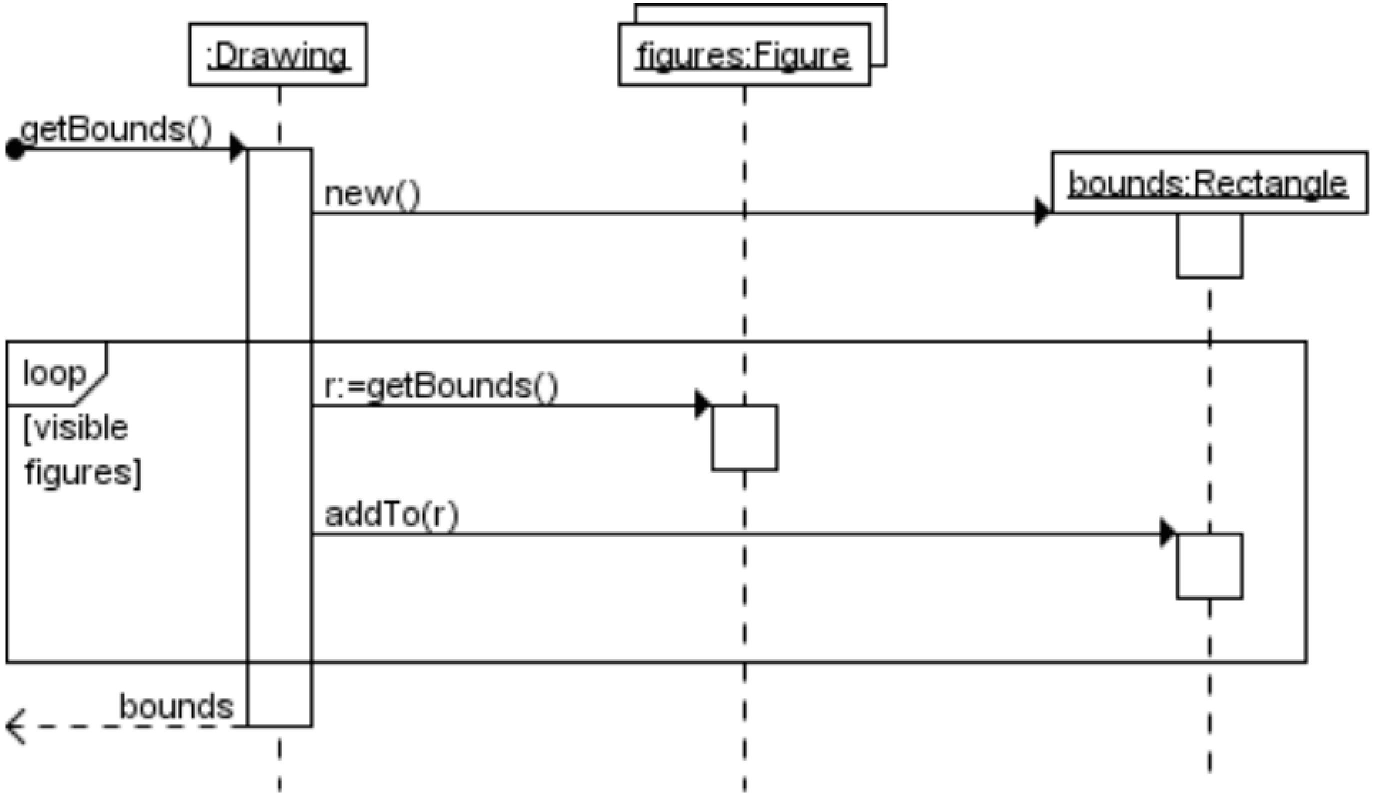
- Elements in a collection:

- Combined fragment:



Repeated Interaction (2)

- Example:



Regular Expressions

Search pattern for finding occurrences in a string

- [eE] stands for e or E.
- [a-z] stands for one of the characters in the range a to z.
- ^ means "match at the beginning of a line/string".
- $ means "match at the end of the line/string".
- X|Y means "match either X or Y", with X and Y both sub-expressions.
- [^x] means not x, so [^E].*\n matches every line except those that start with the E character
- . matches any single character.
- X? matches 0 or 1 repetitions of X.
- X* matches 0 or more repetitions of X.
- X+ matches 1 or more repetitions of X.
- \ is used to escape meta-characters such as (. If you want to match the character (, you need the pattern \(.
- The (and) meta-characters are used to memorize a match for later use. They can be used around arbitrarily complex patterns. For example ([0-9]+) matches any non-empty sequence of digits. The matched pattern is memorized and can be referred to later by using \1. Following matched bracketed patterns are referred to by \2, \3, etc. Note that you will need to encode powerful features such as this one by adding appropriate actions (side-effects) to your automaton encoding the regular expression. This can easily be done by storing a matched pattern in a variable and later referring to it again.

Example: Railway Junction Controller Trace

Write regular expressions (refer to the format of the given output trace) for verifying the above use cases. We use abbreviations to shorten the messages that you need to recognize in your RegExp/FSA. Here are the mappings:

E := A train **E**nters the specified segment (En with n the segment number)

R := A **R**ed signal is sent to the specified segment

G := A **G**reen signal is sent to the specified segment

X := A train leaves the specified segment

Beyond that, each segment has a simple encoding:

1 := left incoming railway segment

2 := right incoming railway segment

3 := outgoing railway segment

Example: Regular Expression

If a train wants to enter the junction, it will eventually get a green light.

Regular expression pattern (for segment 1):
^((([^E].*)|(E [23]))\n)*(**E 1**\n(.*\n)* **G 1**\n((([^E].*)|(E [23]))\n)*)*$

For segment 2:
^((([^E].*)|(E [13]))\n)*(**E 2**\n(.*\n)* **G 2**\n((([^E].*)|(E [13]))\n)*)*$

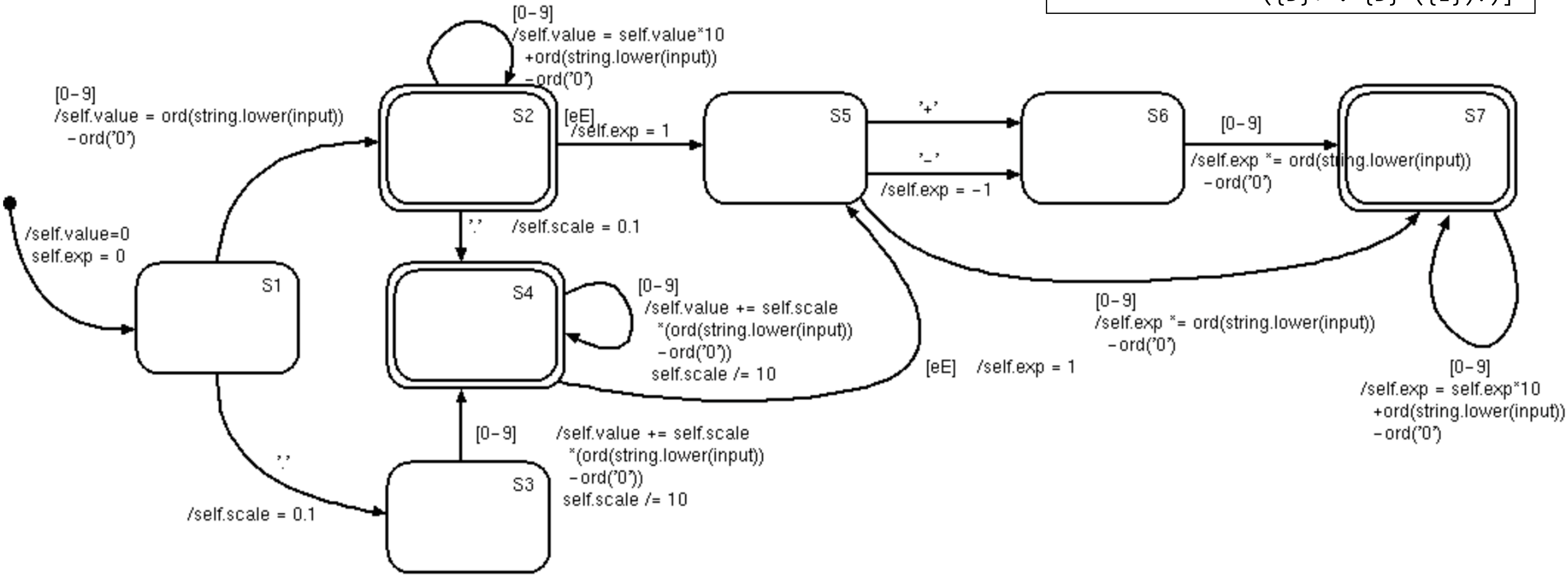
Anything except for E 1:
(([^E].*)|(E [23]))\n

Finite State Automata

- Discrete states + transitions
- Change **state** in response to **external inputs**: **transition**
- Can be used to encode regular expressions

Finite State Automata Example

```
D               [0-9]
E               [eE][+-]?({D})+
Number          [({D}+{E}?)
                 ({D}*'.'{D}+({E})?)
                 ({D}+'.'{D}*({E})?)]
```



Finite State Automata Implementation (semantics)

```
class Scanner:
 """
  A simple Finite State Automaton simulator.
  Used for scanning an input stream.
 """
 def __init__(self, stream):
  self.set_stream(stream)
  self.current_state=None
  self.accepting_states=[]

 def set_stream(self, stream):
  self.stream = stream

 def scan(self):
  ...
```

```
def scan(self):

 self.current_state=self.transition(self.current_state, None)

 if __trace__:
   print "\ndefault transition --> "+self.current_state

 while 1:
   # look ahead at the next character in the input stream
   next_char = self.stream.showNextChar()

   # stop if this is the end of the input stream
   if next_char == None: break

   if __trace__:
     print str(self.stream)
     if self.current_state != None:
       print "transition "+self.current_state+" -| "+next_char,

   # perform transition and its action to the appropriate new state
   next_state = self.transition(self.current_state, next_char)

   if __trace__:
     if next_state == None:
       print
     else:
       print "|-> "+next_state

   # stop if a transition was not possible
   if next_state == None:
     break
   else:
     self.current_state = next_state
     # perform the new state's entry action (if any)
     self.entry(self.current_state, next_char)

   # now, actually consume the next character in the input stream
   next_char = self.stream.getNextChar()

 if __trace__:
   print str(self.stream)+"\n"

 # now check whether to accept consumed characters
 success = self.current_state in self.accepting_states
 if success:
  self.stream.commit()
 else:
  self.stream.rollback()
 return success
```

Finite State Automata: encoding a specific FSA

```
class NumberScanner(Scanner):

 def __init__(self, stream):

   # superclass constructor
   Scanner.__init__(self, stream)

   # define accepting states
   self.accepting_states=["S2","S4","S7"]

 def __str__(self):

   return str(self.value)+"E"+str(self.exp)

 def entry(self, state, input):

   pass
```

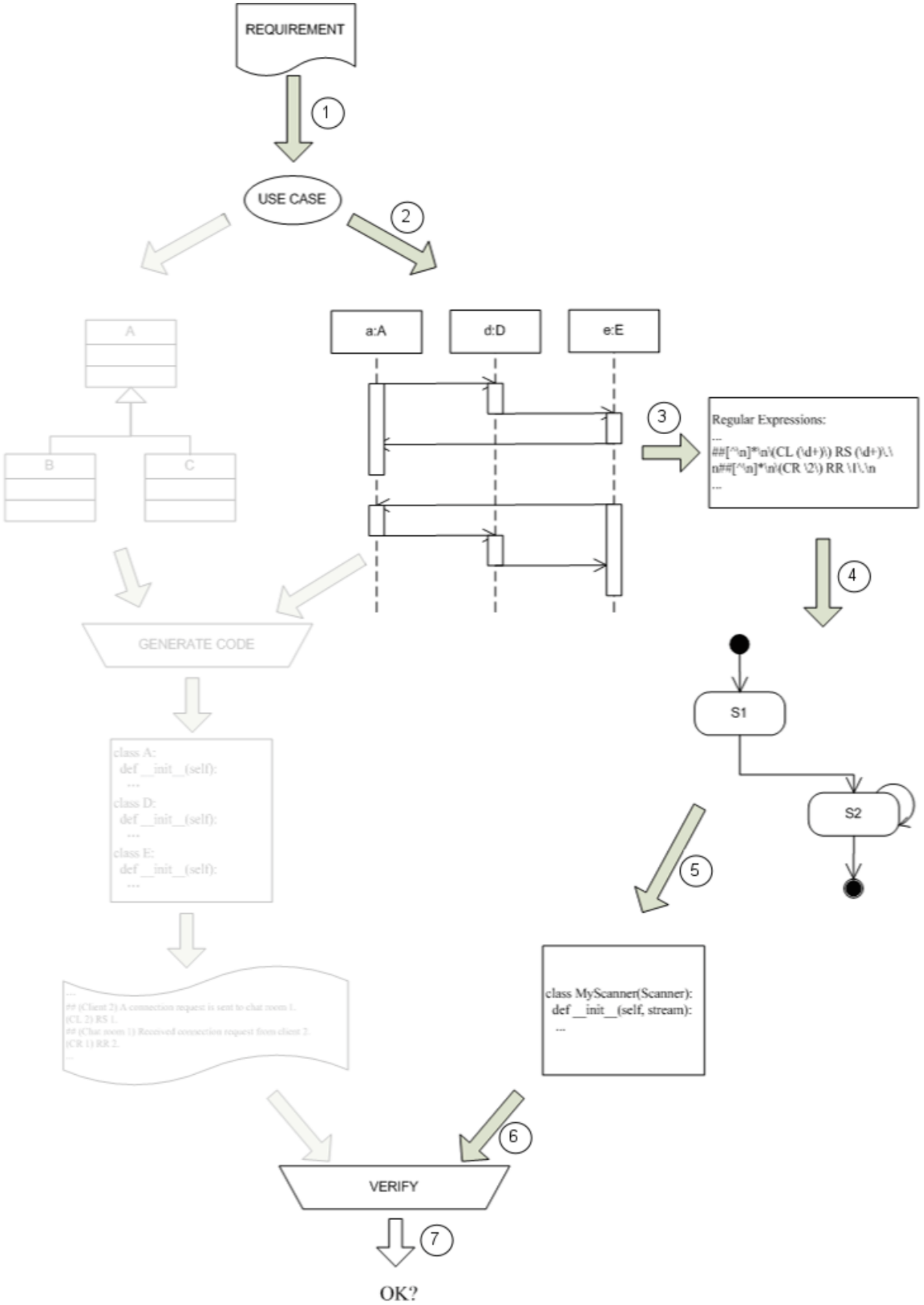
```
def transition(self, state, input):
   """
   Encodes transitions and actions
   """

   if state == None:
      # action
      # initialize variables
      self.value = 0
      self.exp = 0
      # new state
      return "S1"

   elif state == "S1":
    if input  == '.':
      # action
      self.scale = 0.1
      # new state
      return "S3"
    elif '0' <= input <= '9':
      # action
      self.value = ord(string.lower(input))-ord('0')
      # new state
      return "S2"
    else:
      return None

   elif state == "S2":
    if input  == '.':
      # action
      self.scale = 0.1
      # new state
      return "S4"
    elif '0' <= input <= '9':
      # action
      self.value = self.value*10+ord(string.lower(input))-ord('0')
      # new state
      return "S2"
    elif ...
```

Workflow



```
Regular Expressions:
...
##[^\n]*\n\(CL (\d+)\) RS (\d+)\.\
n##[^\n]*\n\(CR \2\) RR \1\.\n
...
```

```
class A:
  def __init__(self):
    ...
class D:
  def __init__(self):
    ...
class E:
  def __init__(self):
    ...
```

```
...
## (Client 2) A connection request is sent to chat room 1.
(CL 2) RS 1.
## (Chat room 1) Received connection request from client 2.
(CR 1) RR 2.
...
```

```
class MyScanner(Scanner):
  def __init__(self, stream):
    ...
```