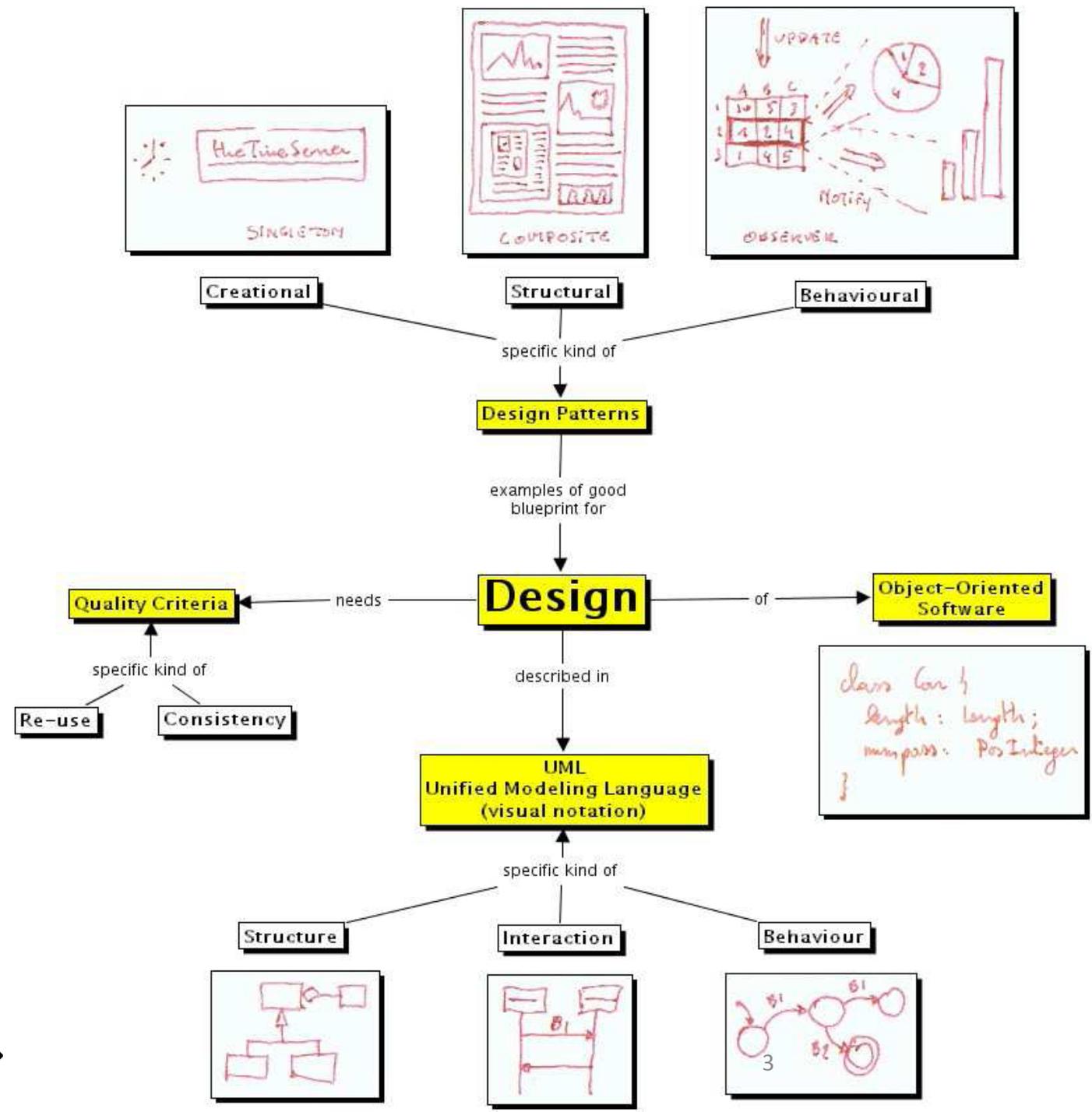
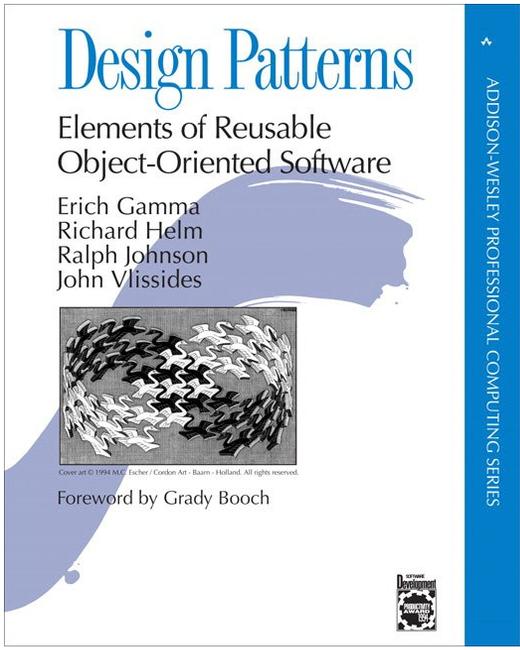


Object-Oriented Software Design with different Unified Modelling Language (UML) notations

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

Bart Meyers

Hans Vangheluwe



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>

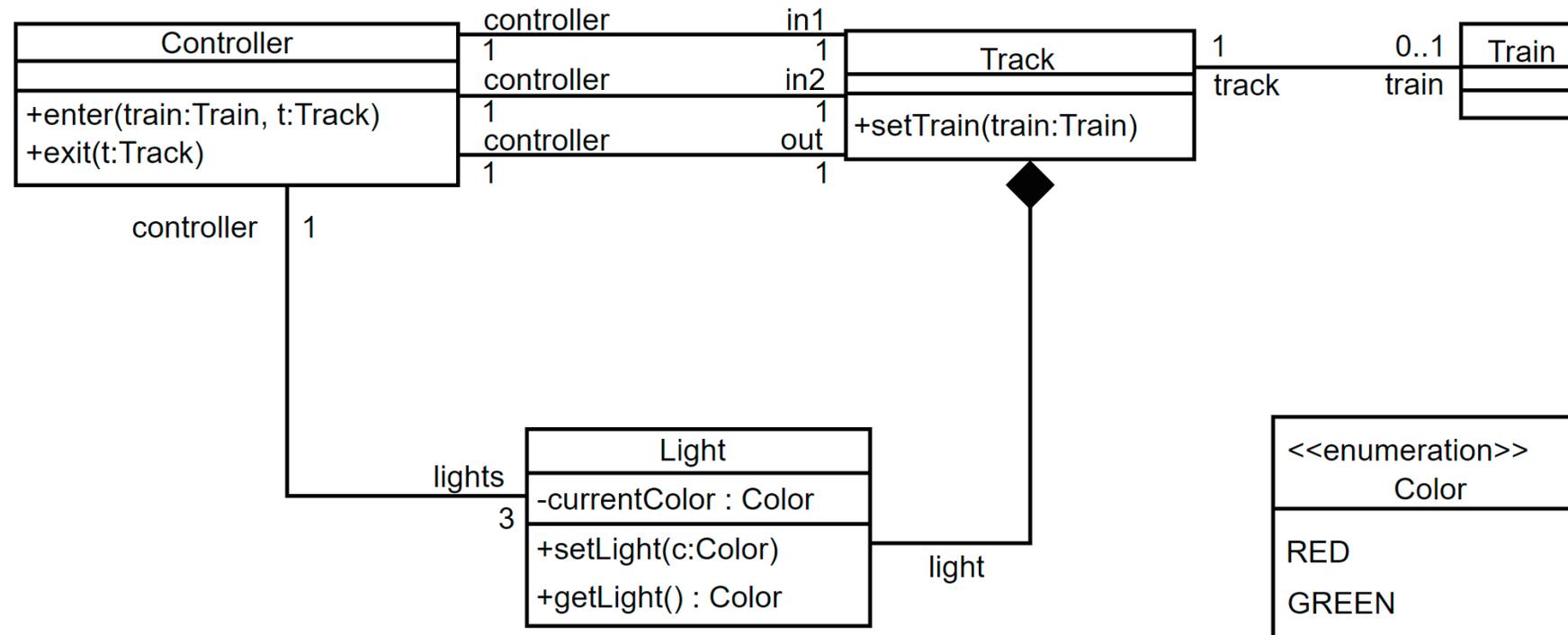
Use Cases

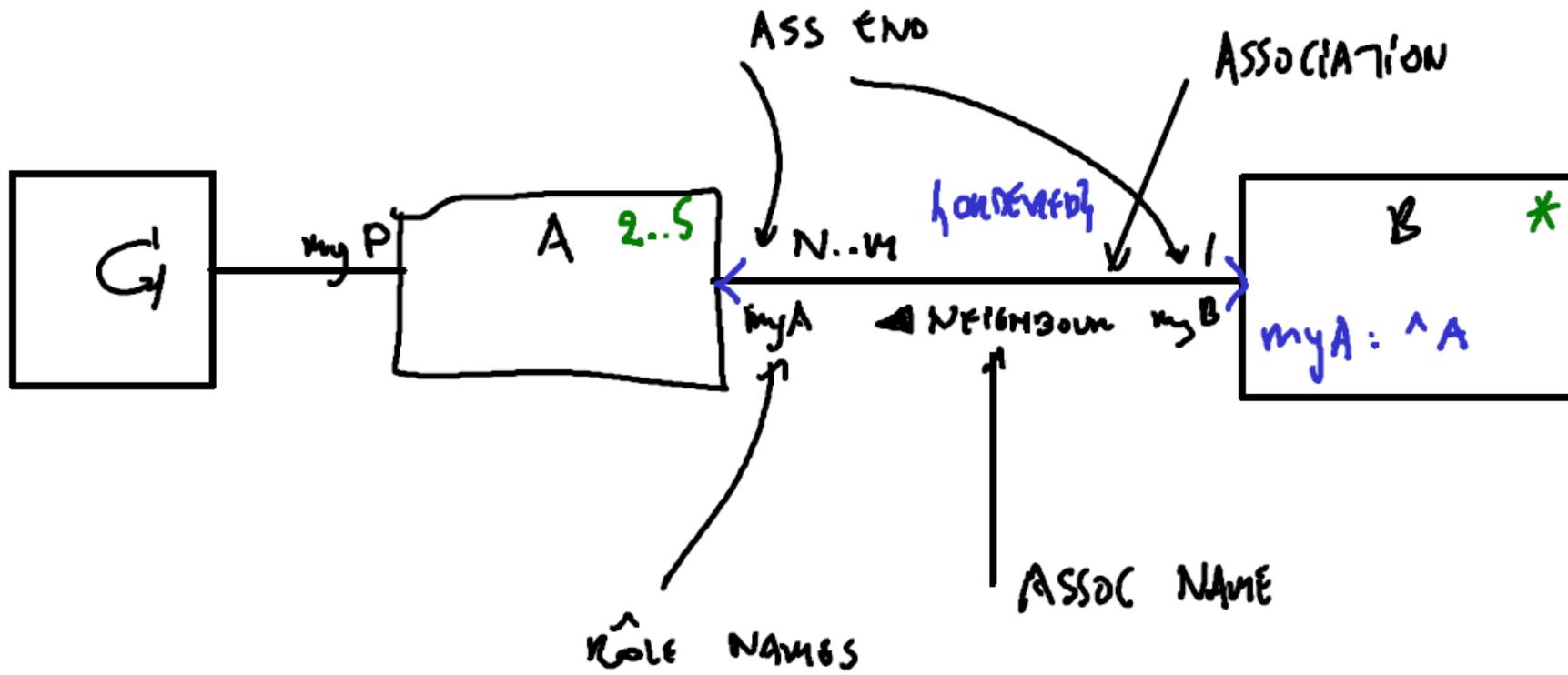
- Document functional requirements of the system
 - interactions between system and **environments** to achieve **user goals**
- Understandable for (non-technical) client
 - semi-formal
 - complete, consistent and verifiable
- Business viewpoint
 - **who** (actors) does **what** (interaction) with **what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

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

Class Diagrams

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

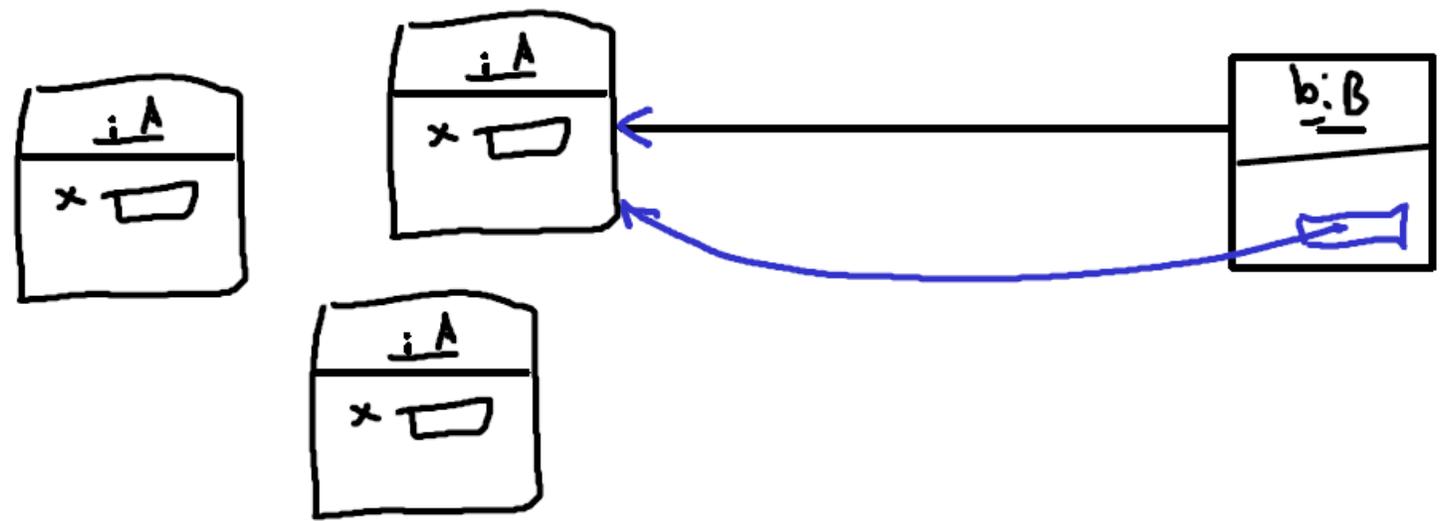




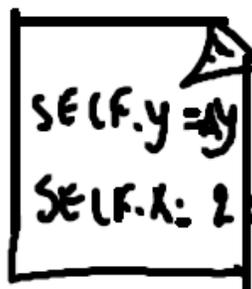
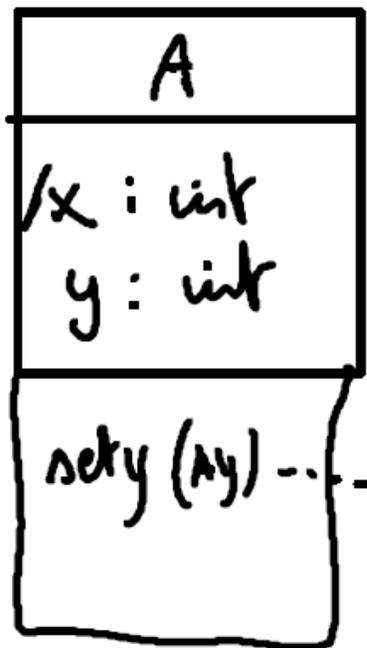
1..1 1
 0..* *
 N..M
 ∅ *
 1

cardinality is "the number of elements in a particular set or other grouping"

multiplicity as "A specification of the range of allowable cardinality values – the size that a set may assume"

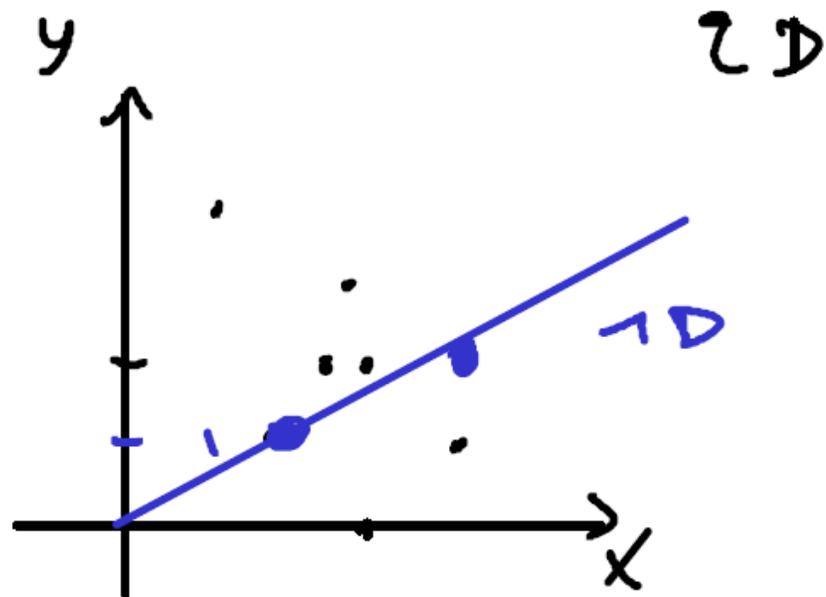


b. myA, x

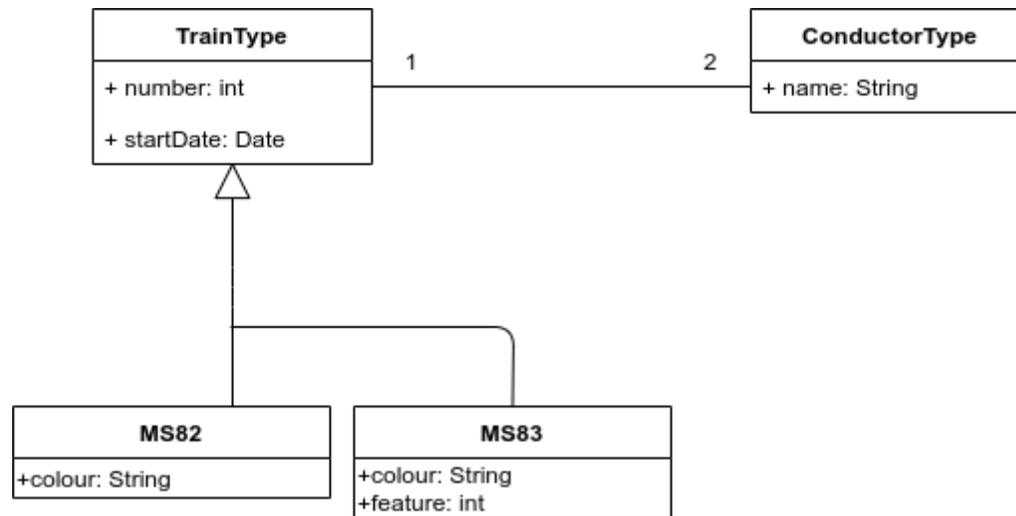


/ + PUBLIC
 - PRIVATE
 # PROTECTED

/ DERIVED



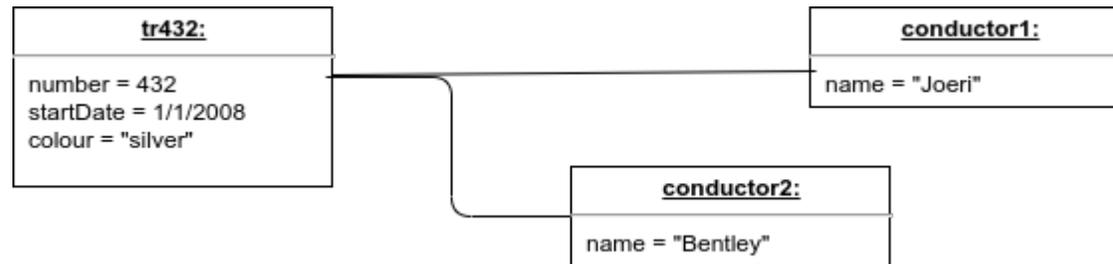
Class Diagram



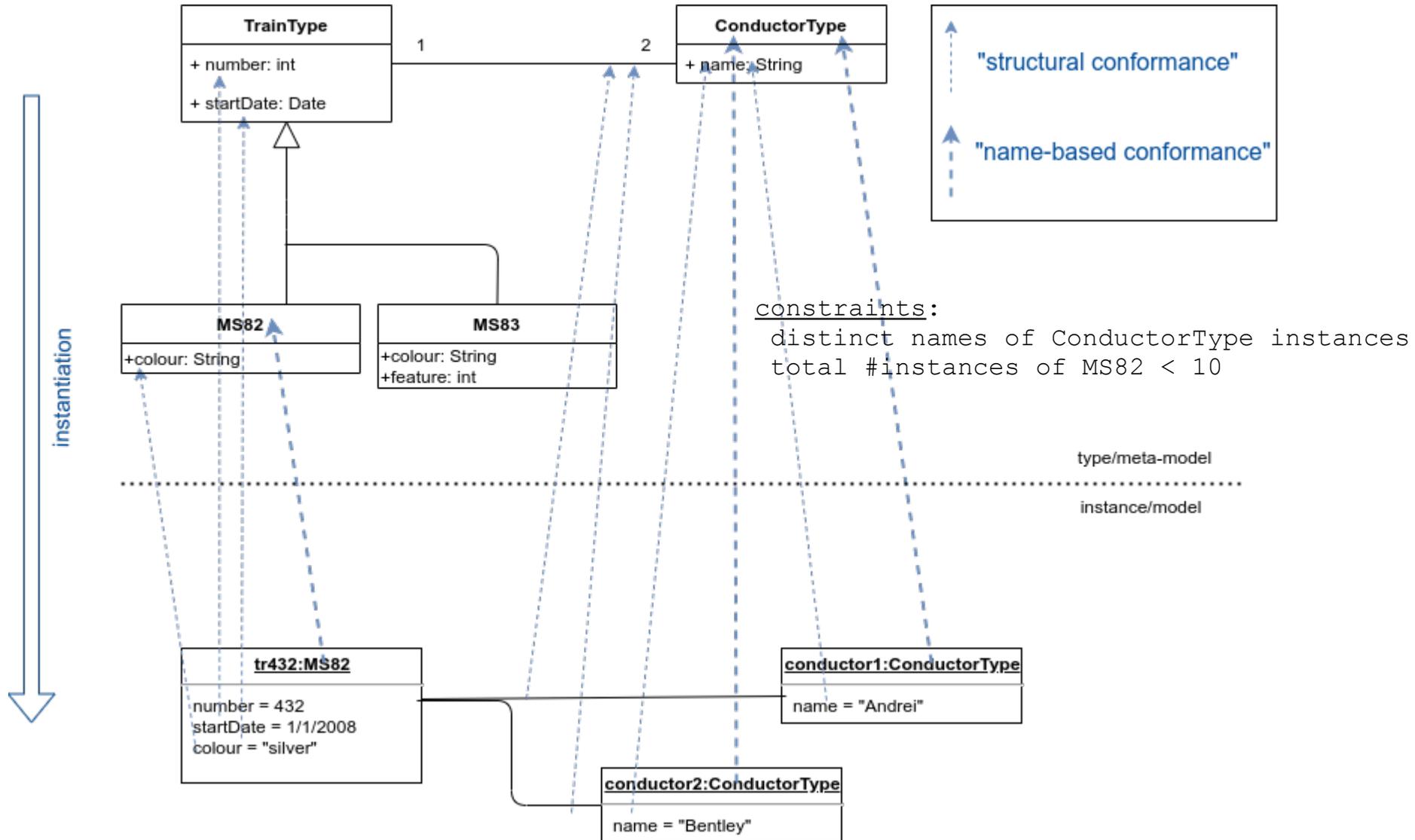
Object Diagram

Convention:

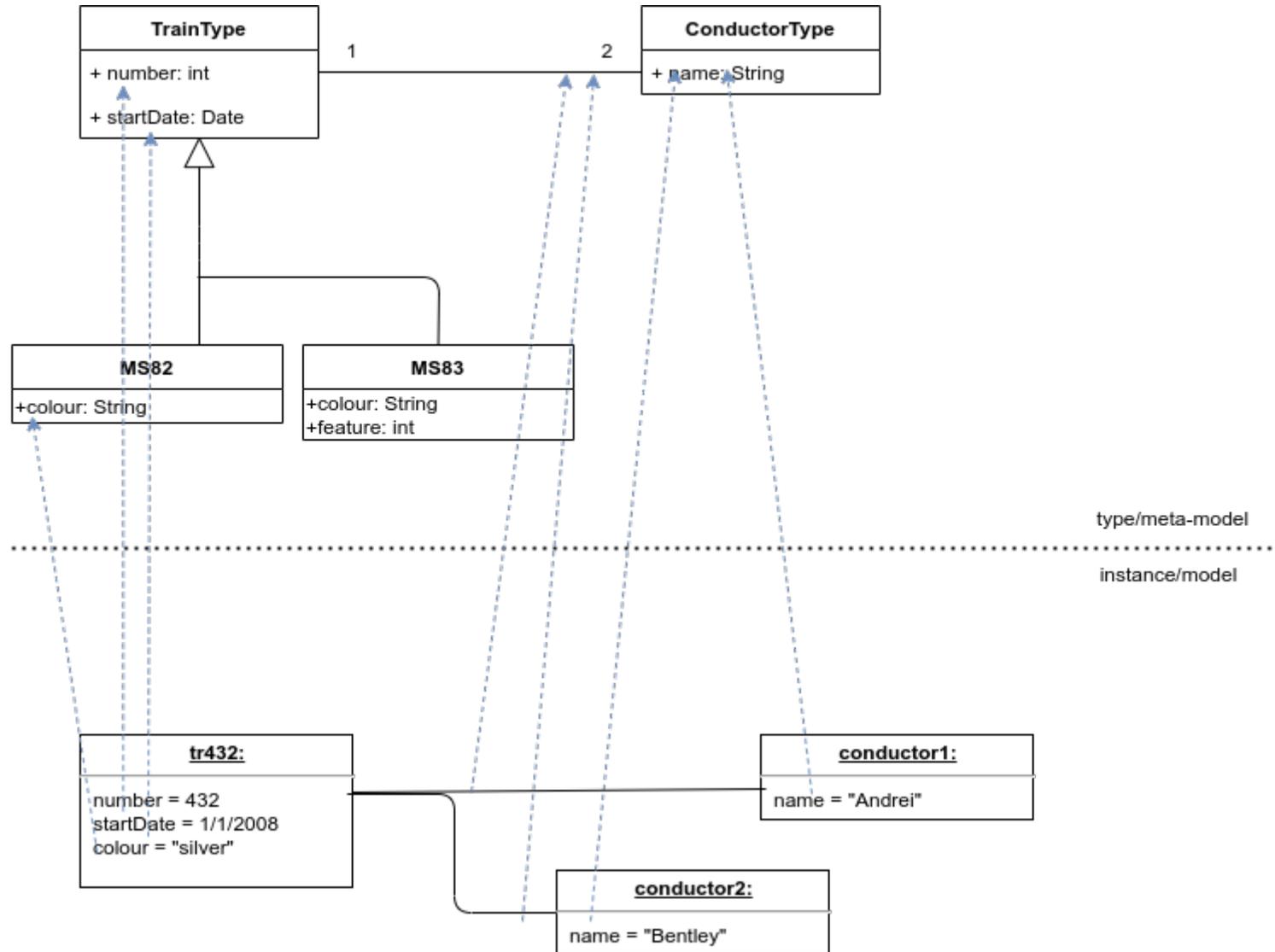
objectName:ClassName



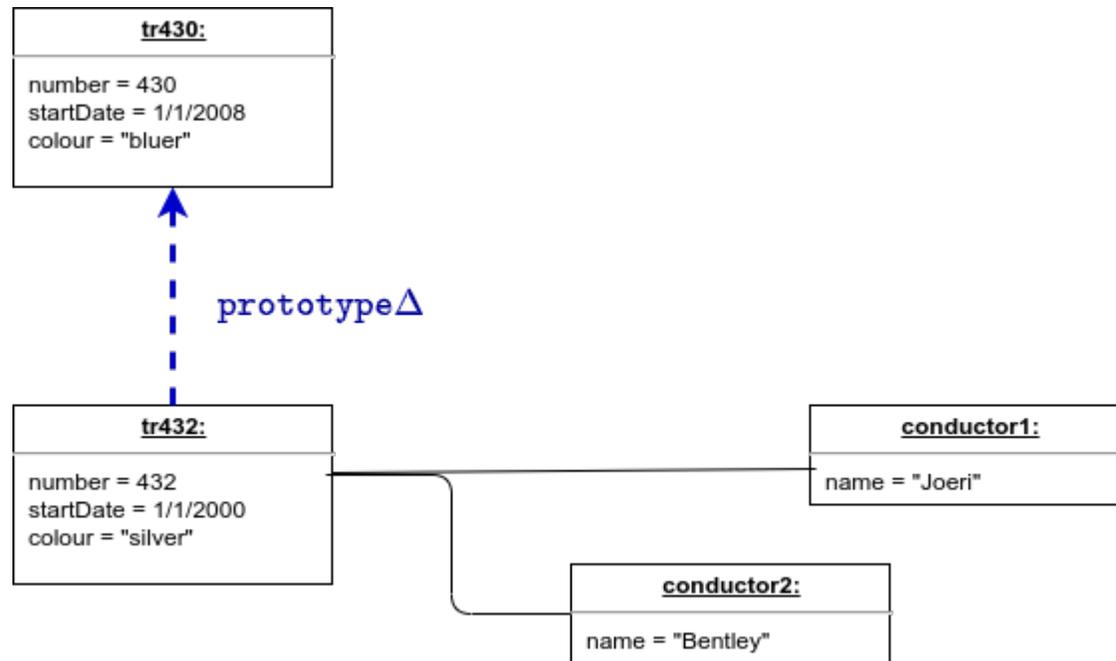
Class vs. Instance



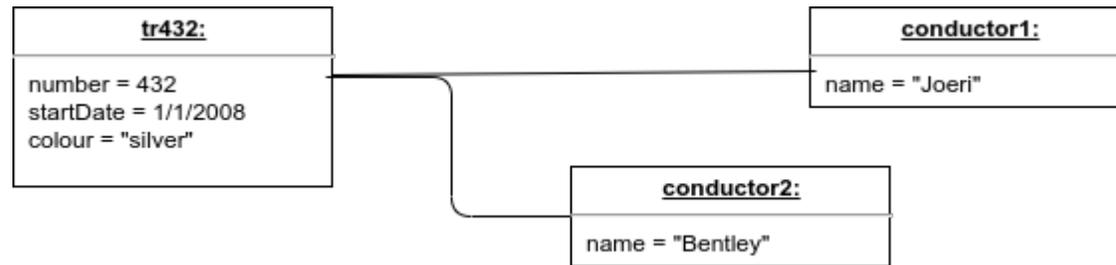
“duck typing”



delegation, object-based

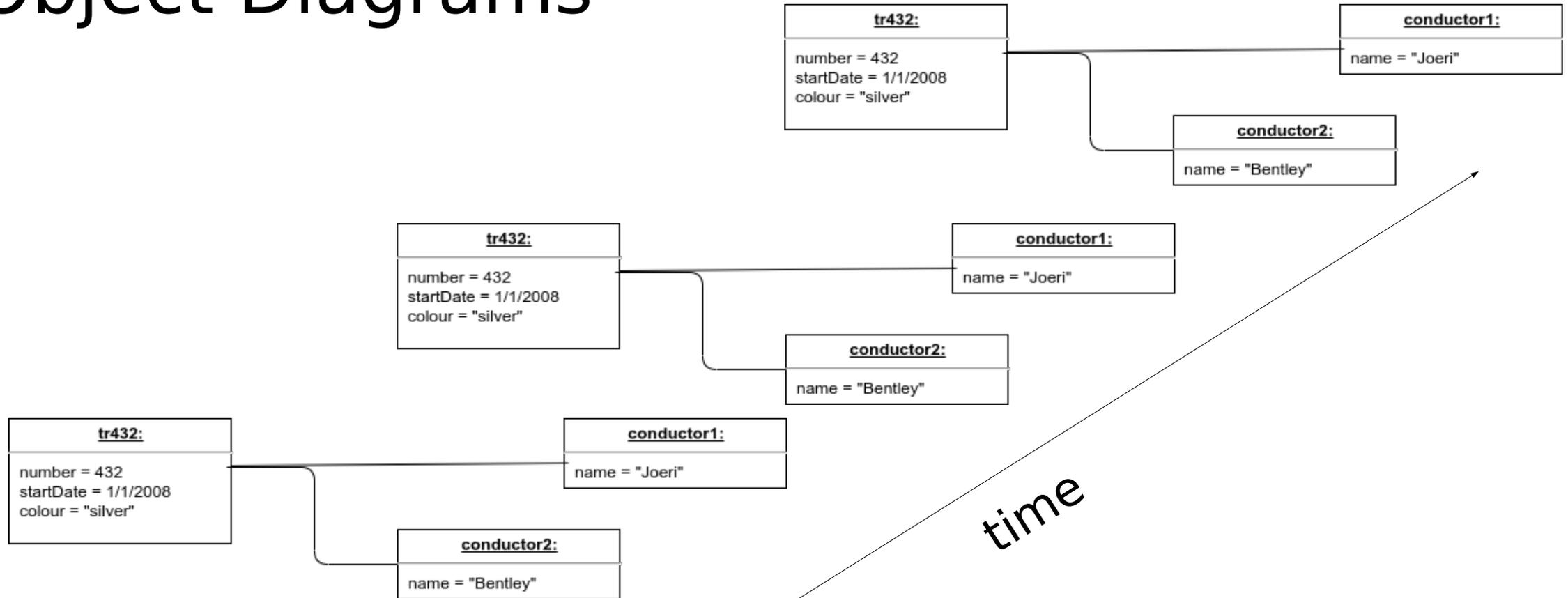


Object Diagram



“(state) snapshot” during program execution

Object Diagrams



“meaning” / “(trace) semantics” of a program

= “behaviour trace”

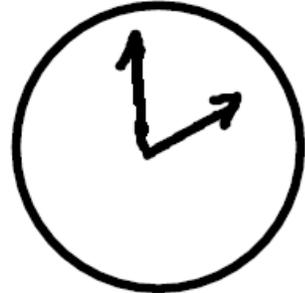
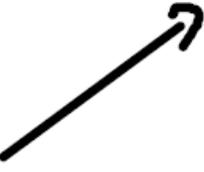
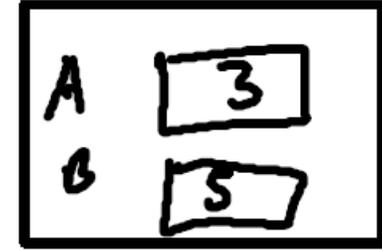
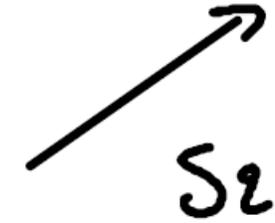
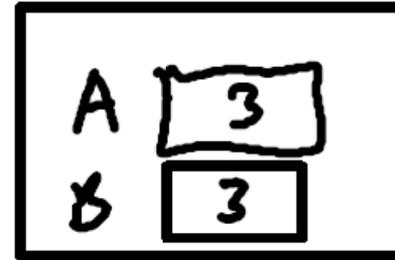
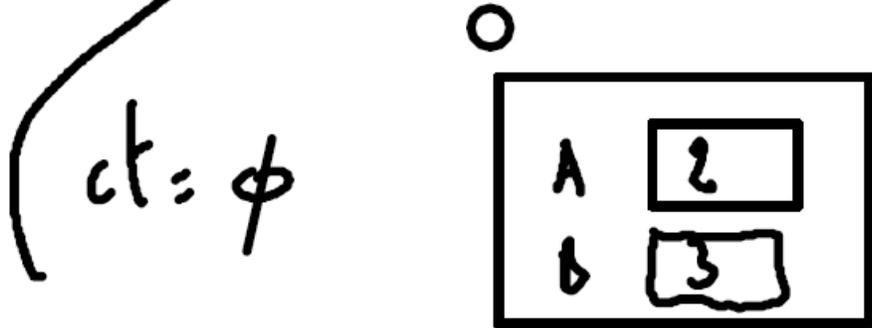
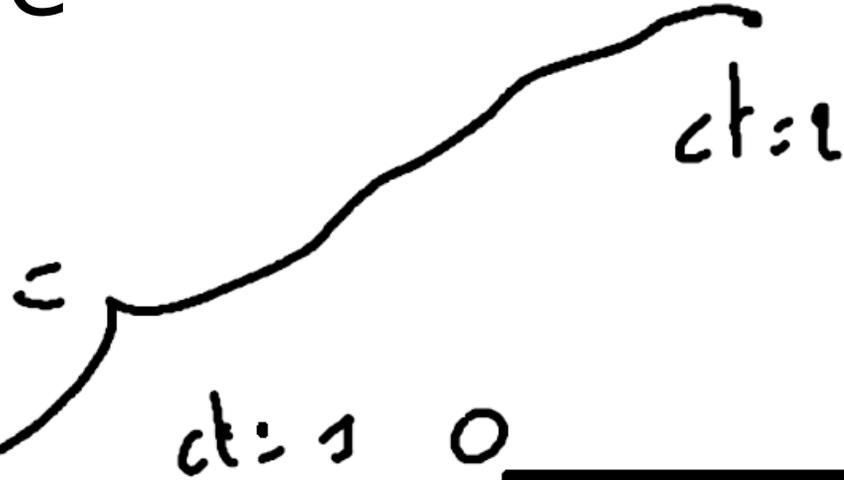
= sequence of state “snapshots”
(produced by code/method execution)

Static Structure

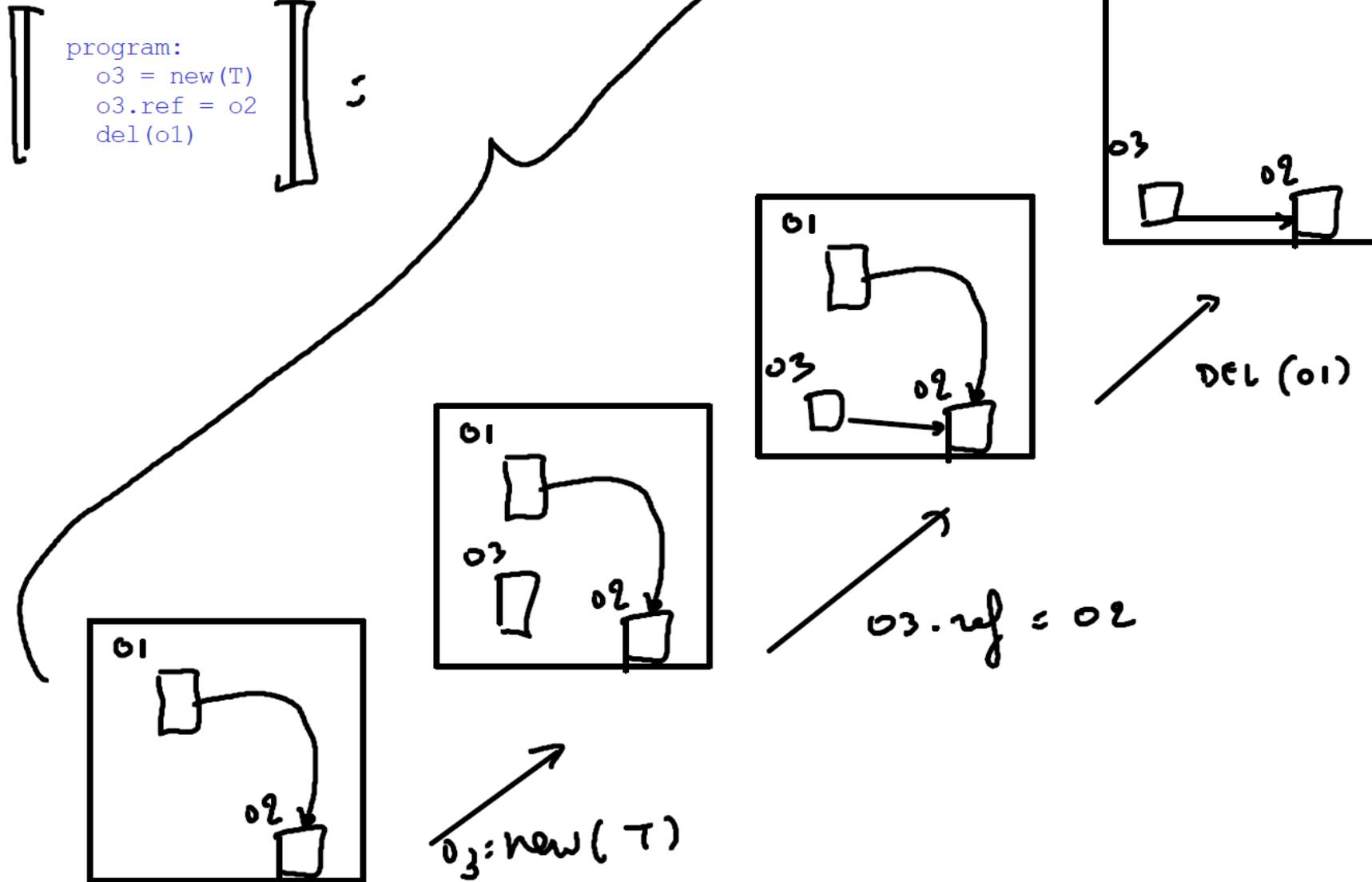
s_1
 s_2
 s_3

program:

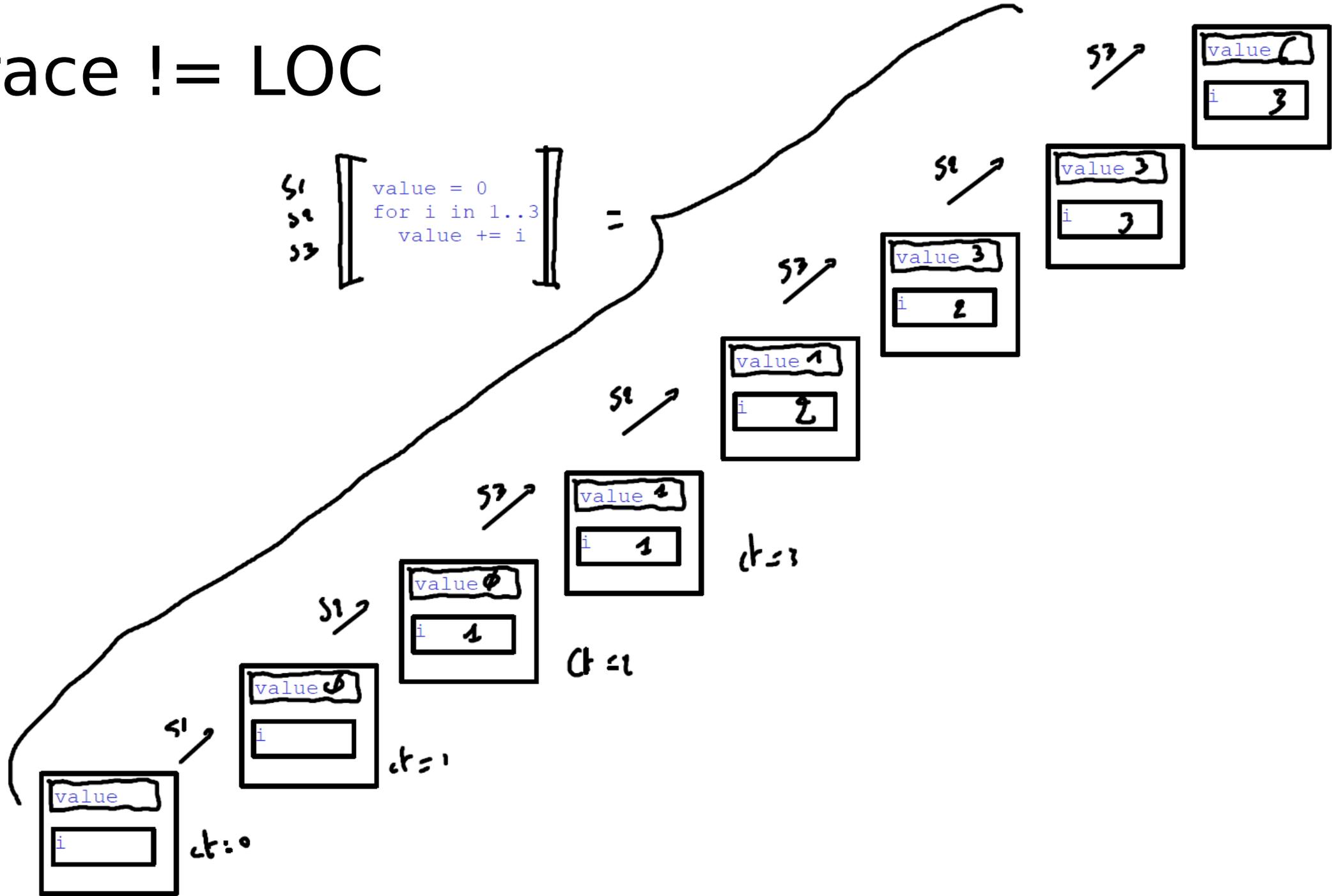
```
o.A = 3;  
o.B = 5;  
o.A = o.B + 3;  
...
```



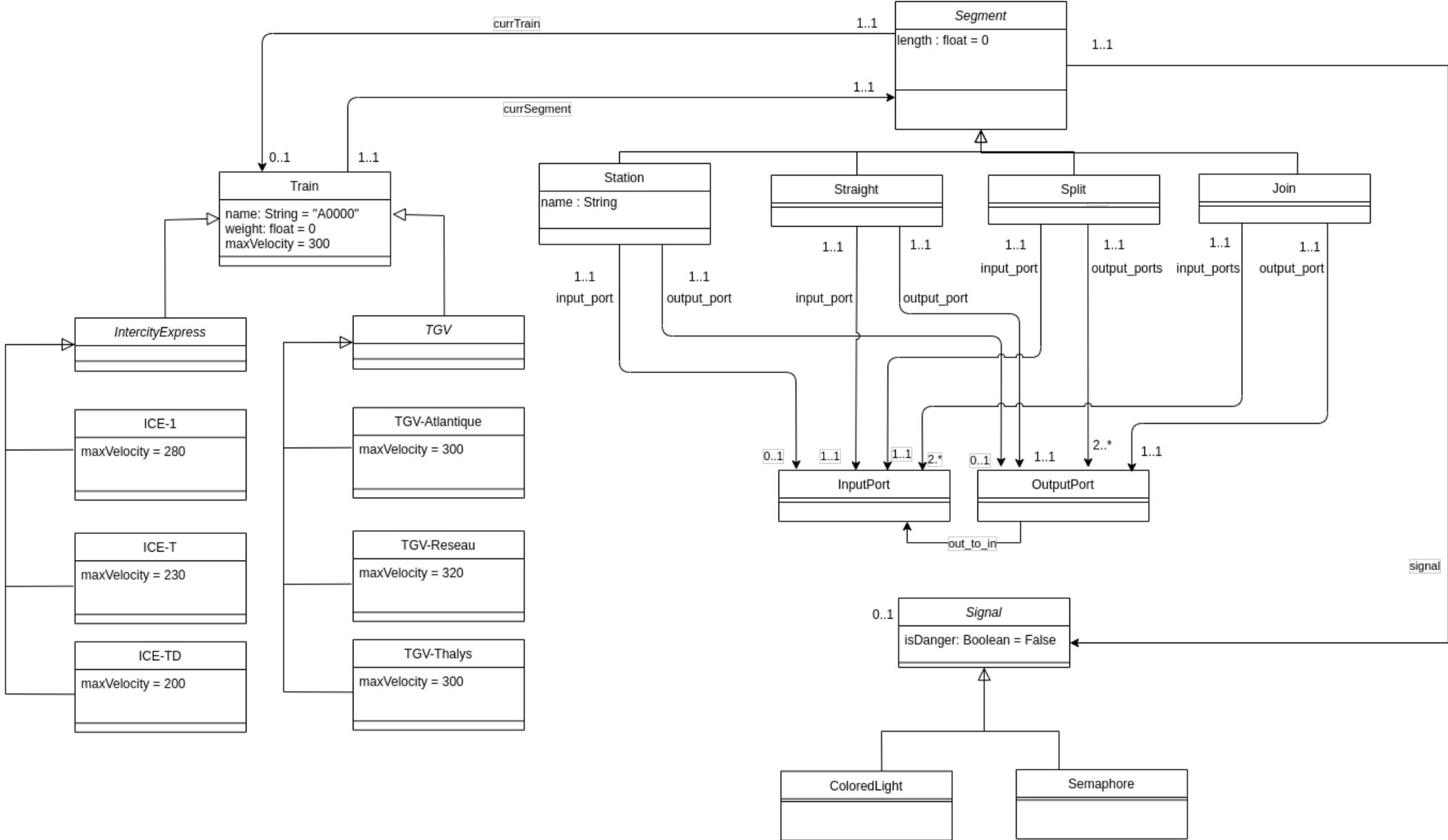
Dynamic Structure



Trace != LOC



Railway Network Class Diagram

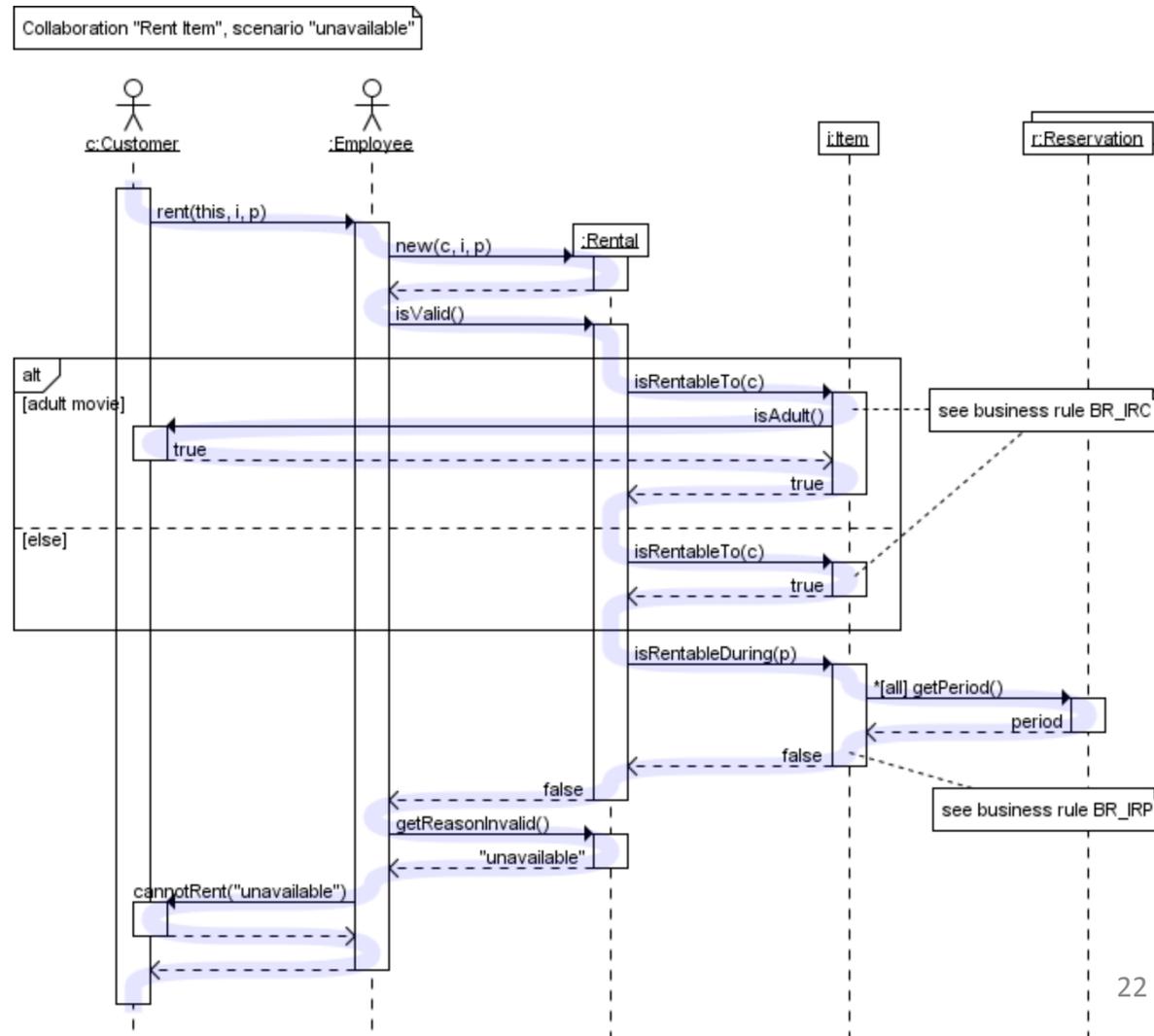


Sequence Diagrams

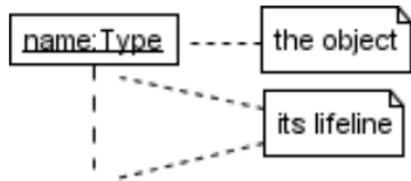
- Behaviour of the system
 - (Object) **Interaction** diagram (not internal)
- Complementary to Class Diagrams (structure vs. behaviour)
- Complementary to Use Cases (“**what?**” vs. “how?”)

Sequence Diagrams

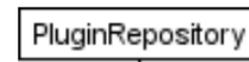
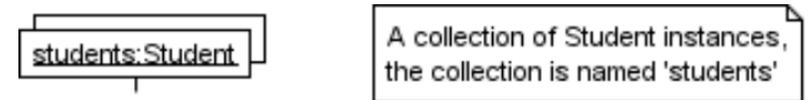
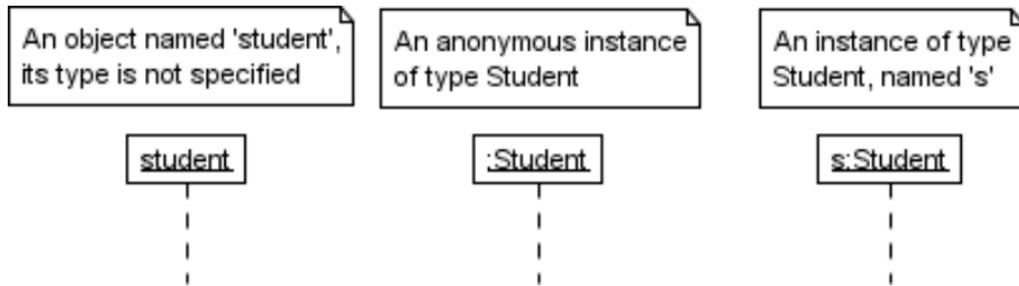
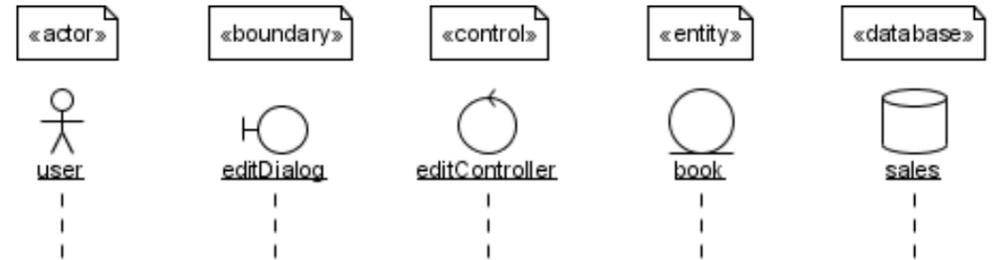
Here: single thread of interaction
(though multiple interacting objects)



Objects and Lifelines



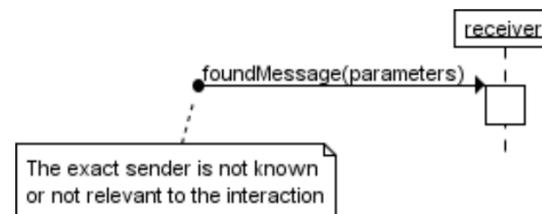
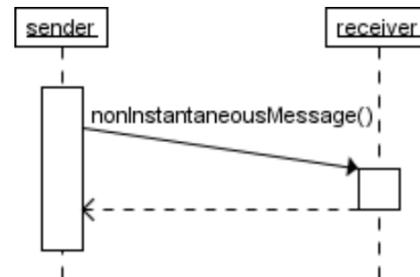
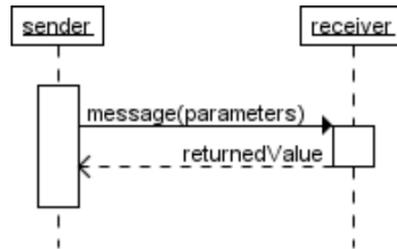
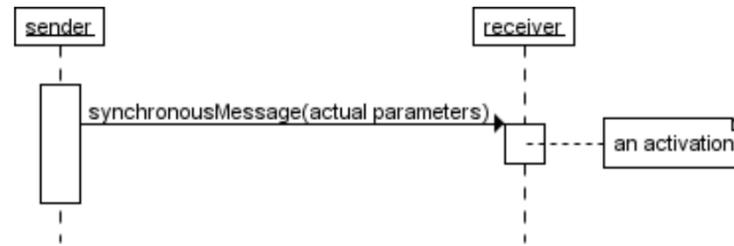
types of objects:



class object
(only use class messages)

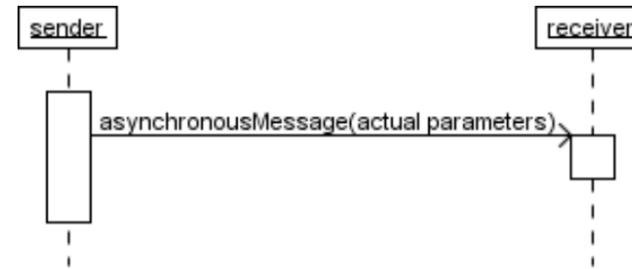
Messages (1)

- Synchronous:
- Returned value:
- Not instantantaneous:
- Found message:

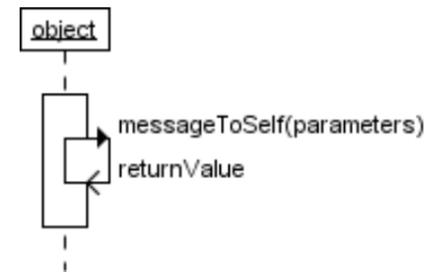


Messages (2)

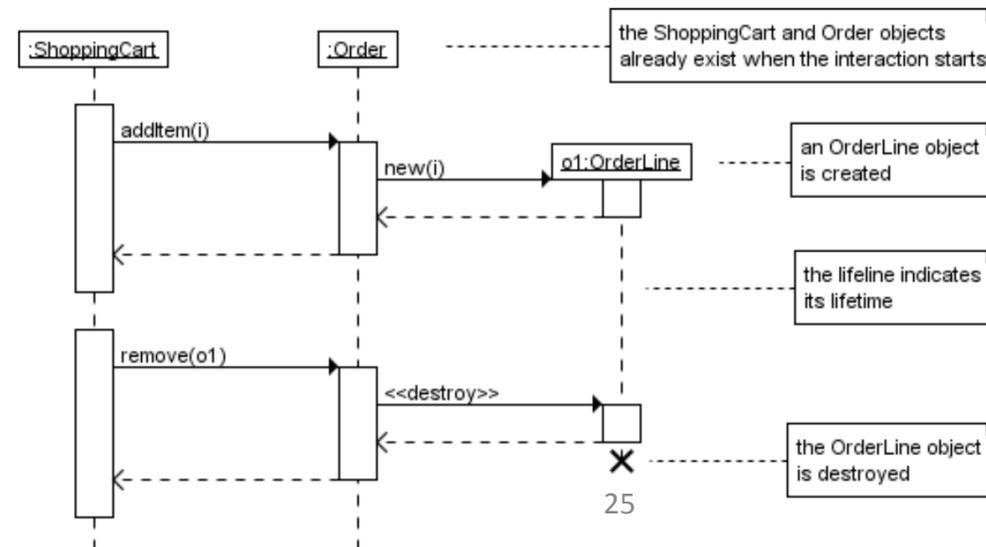
- Asynchronous:



- Message to self:

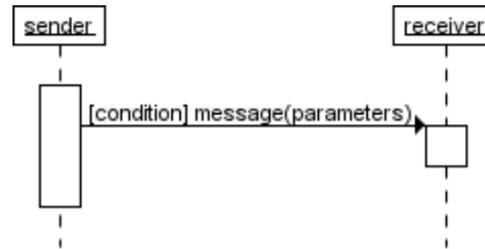


- Object creation/destruction:

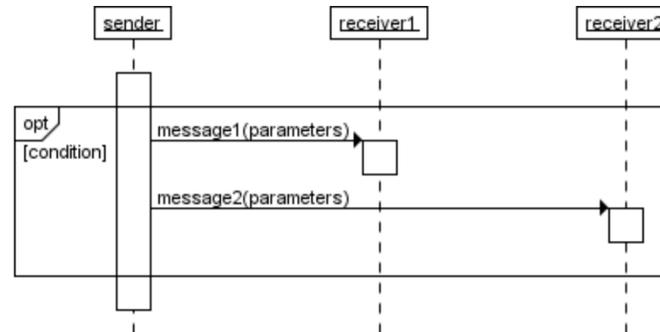


Conditional Interaction

- Message with guard:

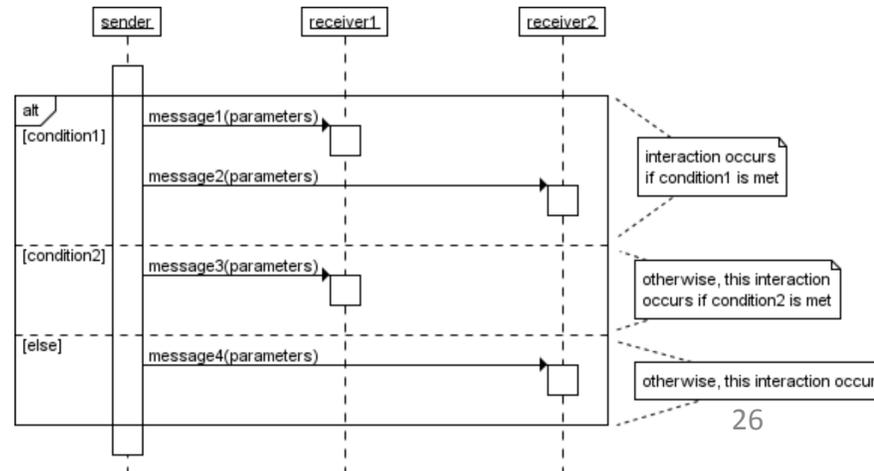


- Multiple messages:



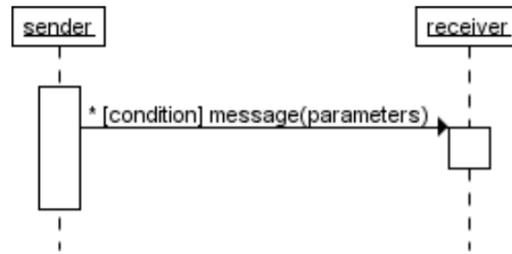
“combined fragment”

- Alternative interactions:



Repeated Interaction (1)

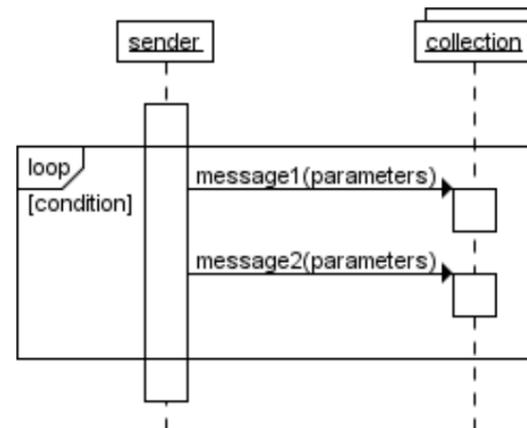
- Repeated message:



- Elements in a collection:

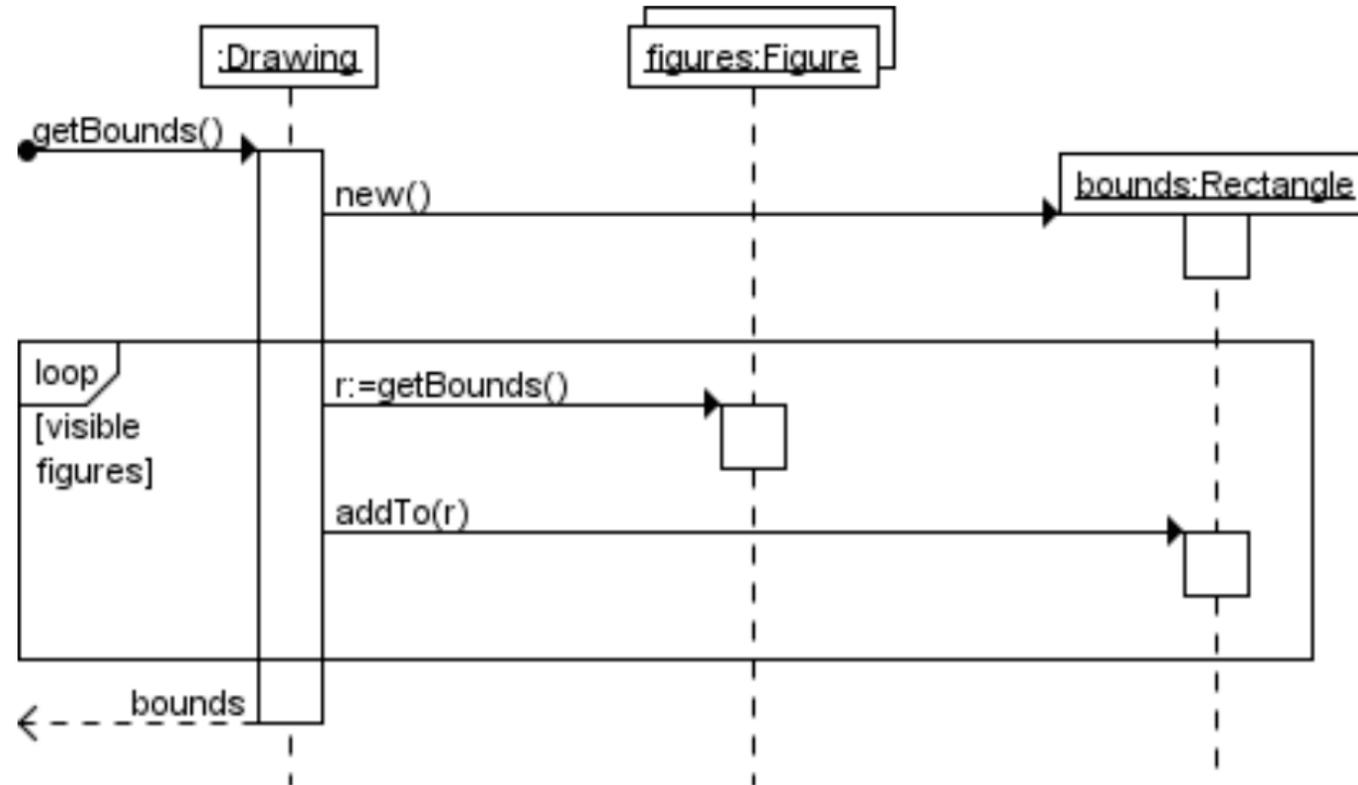


- Combined fragment:



Repeated Interaction (2)

- Example:



Regular Expressions

$$\llbracket \text{"ABC"} \rrbracket_{RE} = \{ \text{"ABC"} \}$$

$$\llbracket \text{"[ABC]"} \rrbracket_{RE} = \{ \text{"A"}, \text{"B"}, \text{"C"} \}$$

↑ ↑
"META" CHAR

$$\llbracket \text{"[A-Z]"} \rrbracket_{RE} = \{ \text{"A"}, \text{"B"}, \dots, \text{"Z"} \}$$

$$\llbracket \text{"[ABC][^A]"} \rrbracket_{RE} = \{ \text{"AB"}, \text{"AC"}, \dots, \text{"AZ"}, \text{"BB"}, \dots \}$$

SIZE (↑ ↑)
 3 * (26-1)

$$\llbracket \text{"A*"} \rrbracket_{RE} = \{ \text{""}, \text{"A"}, \text{"AA"}, \dots \}$$

$$\llbracket \text{"A?" } \rrbracket_{RE} = \{ \text{""}, \text{"A"} \}$$

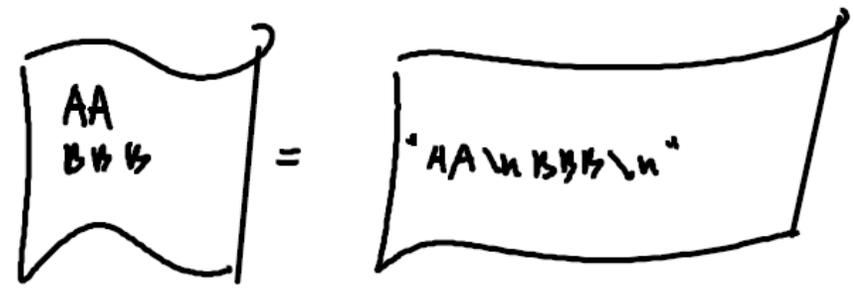
$$\llbracket \text{"A+" } \rrbracket_{RE} = \{ \text{"A"}, \text{"AA"}, \text{"AAA"}, \dots \}$$

$$\llbracket \text{"^[AB]+"} \rrbracket_{RE} = \{ \text{"A"}, \text{"B"}, \text{"AA"}, \dots \}$$

↑
BEGINNING OF STRING

$\llbracket " ^X \dots A $" \rrbracket_{RE}$

$\llbracket " A + \backslash n B * \backslash n " \rrbracket_{RE}$



$\llbracket " A (AB)^+ \rrbracket_{RE} = \{ " A AB (" , " A ABAB (" , \dots \}$
 $\neq [AB]^+$

$\backslash n$ NEWLINE
 $\backslash d$ DECIMAL NUMBER

$[0-9]$

$[0-9]^+$

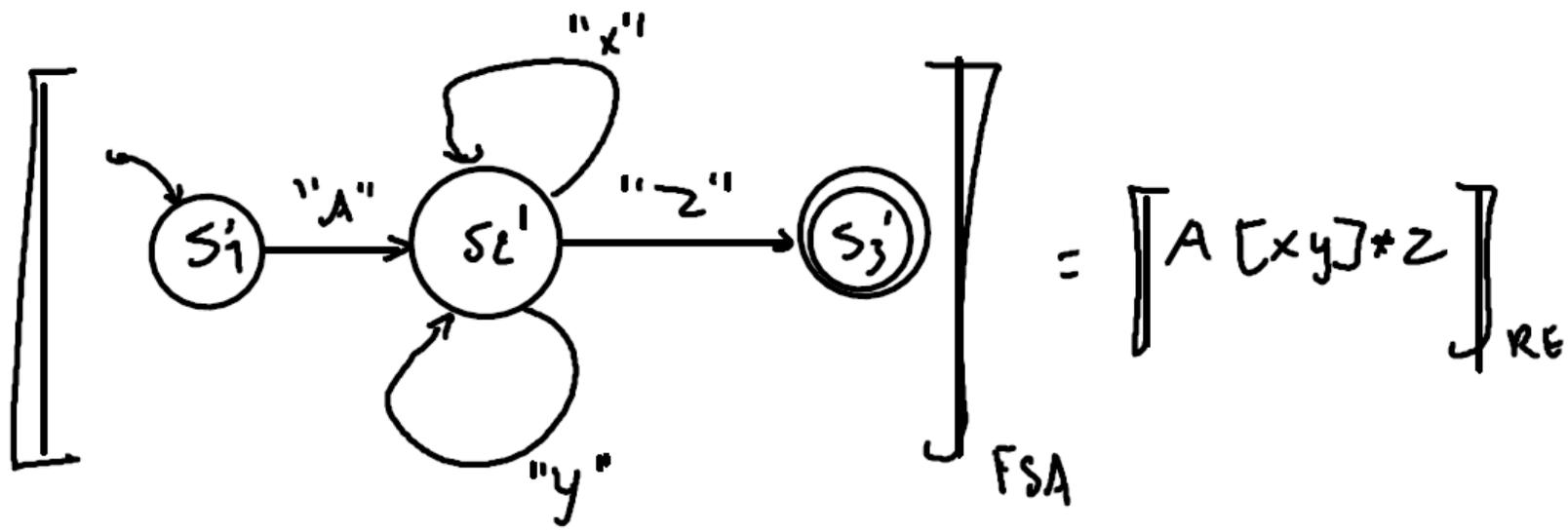
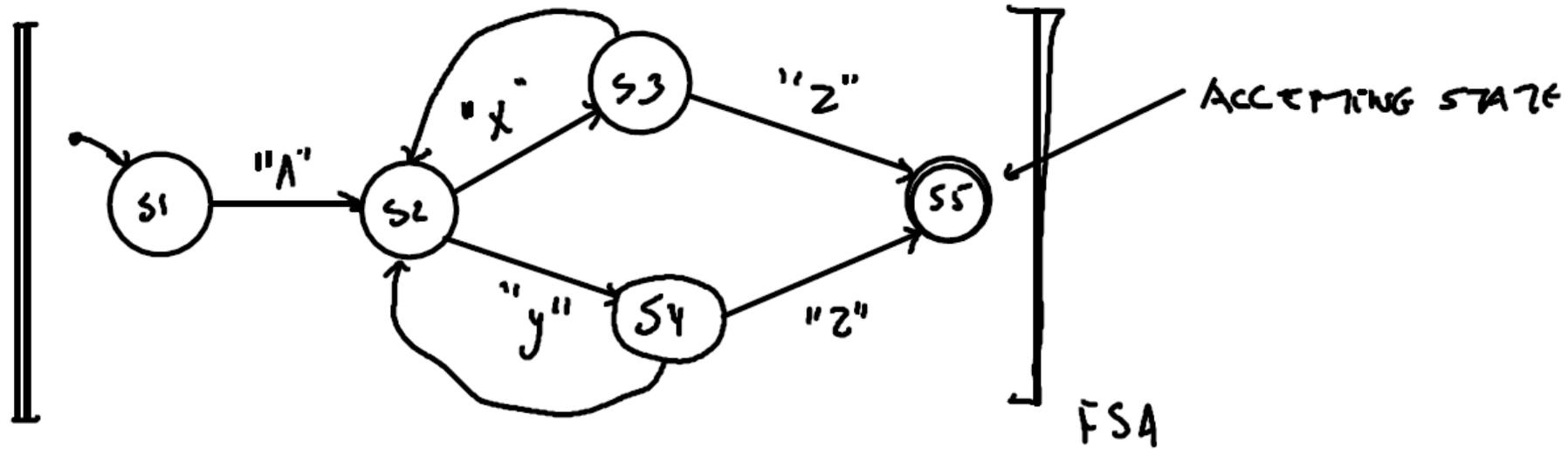
$[1-9][0-9]^*$

$\neq 005$

$\llbracket x | y \rrbracket_{RE}$

$$\left\{ "A [xy]^* Z" \right\}_{RE} = \left\{ "Axz", "Ayz", \dots \right\}$$

"Axyxyz"

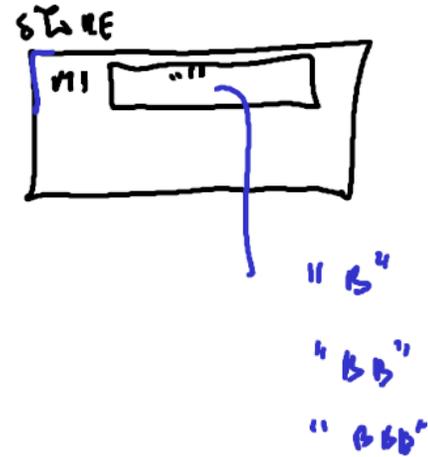
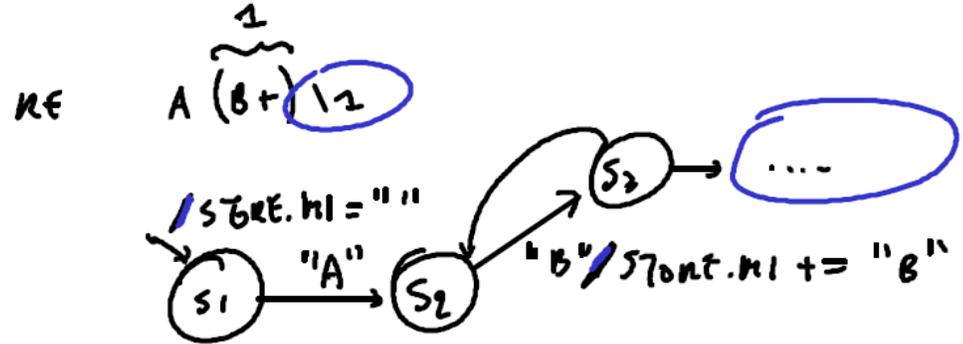
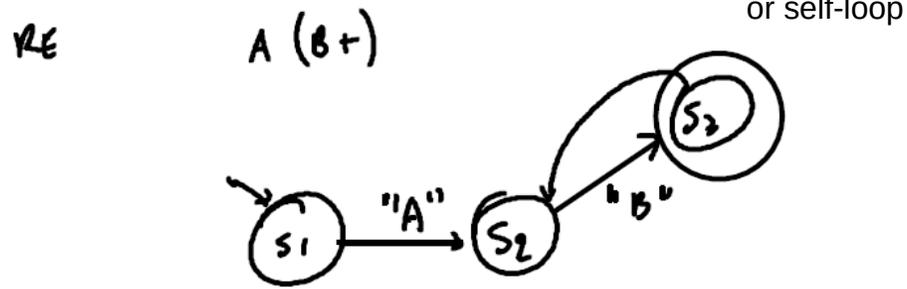


RE "A ([xy] P?) B (Q+) | 1 | 1 R | 2 "

WATCH 1 WATCH 2

A x P y Q Q Q x P x P Q Q Q

n1 n2



RE LIBRARY

{ "A", "B", "C" }

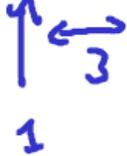
LCHECK = RE.COMPILE("[ABC]")

? = LCHECK.FINDSTR("XABC")



LCHECK = RE.COMPILE("A+")

? = LCHECK.FINDSTR("XAAA")



(1, 3)
↑
pos LENGTH

"A"
"AA"
"AAA"

LONGEST / MAXIMAL

INTENSION EXTENSION
↓ ↓
∈ ["A+"]_{RE} = { "A", "AA", "AAA", ... }

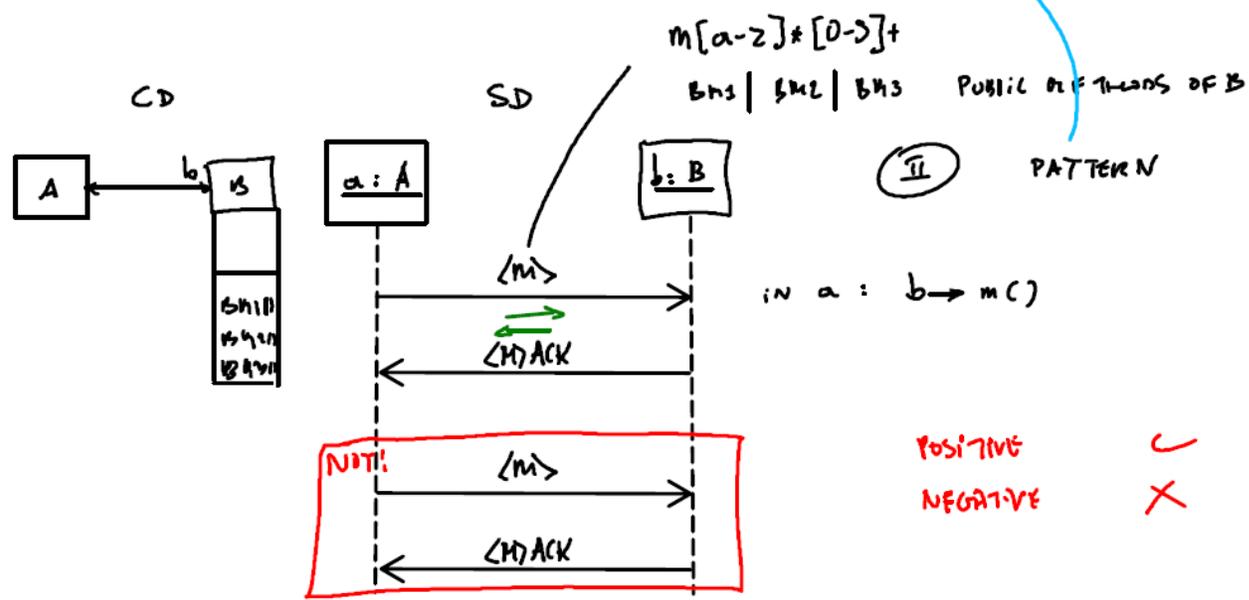
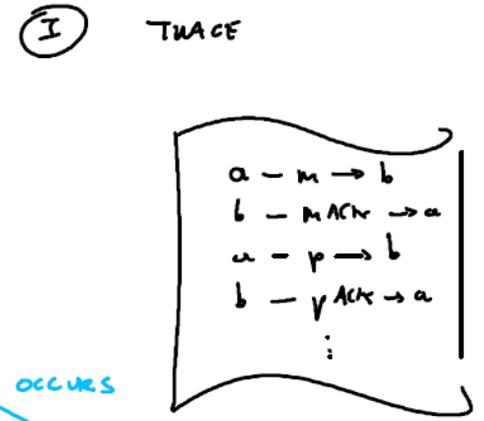
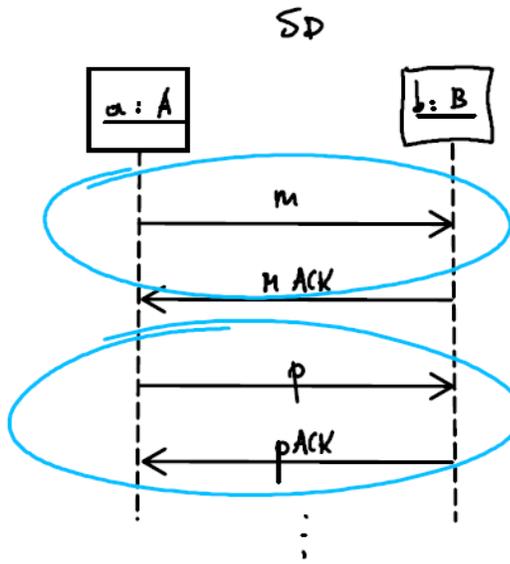
LCHECK = RE.COMPILE("[(A+)(B+)(C+)"])

? = LCHECK.FINDSTR("XAAA Y BB Z CCCC")

GROUPING

[(4, 3), (5, 2), (8, 4)]





RF: $\underbrace{([a-z]^+)}_1 - \underbrace{([a-z]^+)}_2 \rightarrow \underbrace{([a-z]^+)}_3 \setminus n \setminus 3 - \setminus ACK \rightarrow \setminus 2$

#.* <n

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 given 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.

Anything except for “E 1”:

```
(([^E].*)|(E [23]))\n
```

Regular expression pattern (for segment 1):

```
^((([^E].*)|(E [23]))\n)*(E 1\n(.*\n)*G 1\n((([^E].*)|(E [23]))\n)*)*$
```

For segment 2:

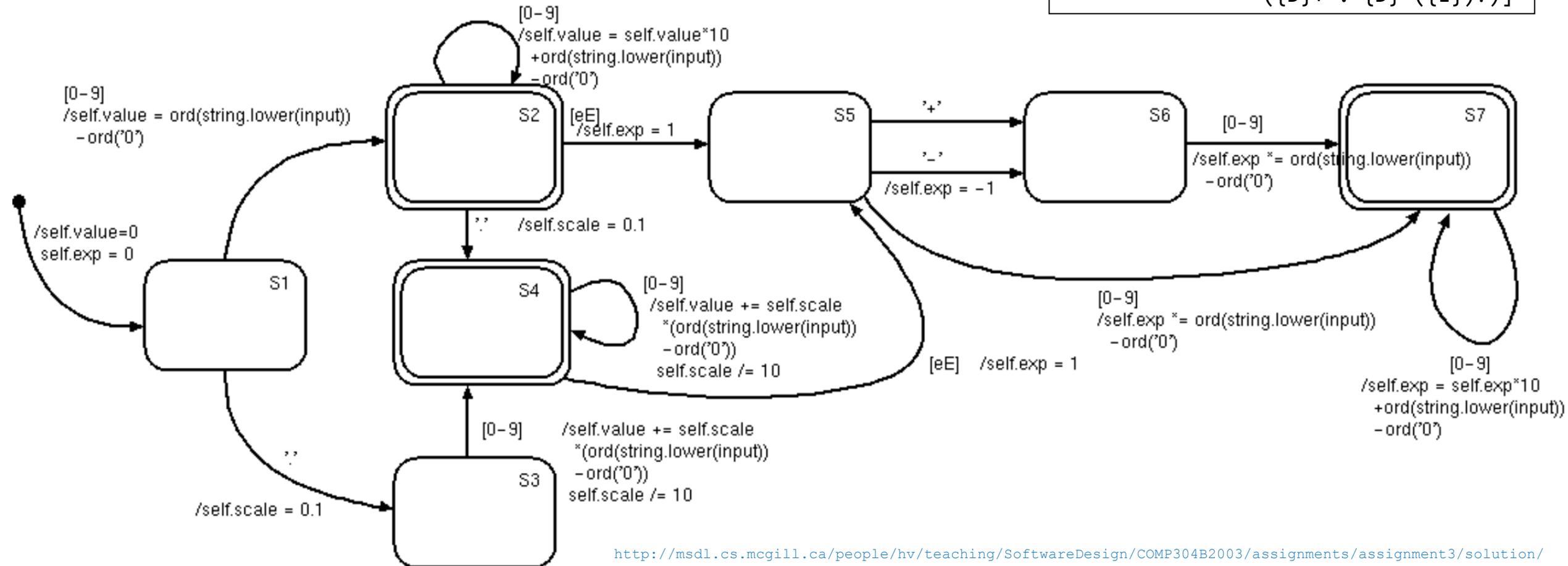
```
^((([^E].*)|(E [13]))\n)*(E 2\n(.*\n)*G 2\n((([^E].*)|(E [13]))\n)*)*$
```

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

