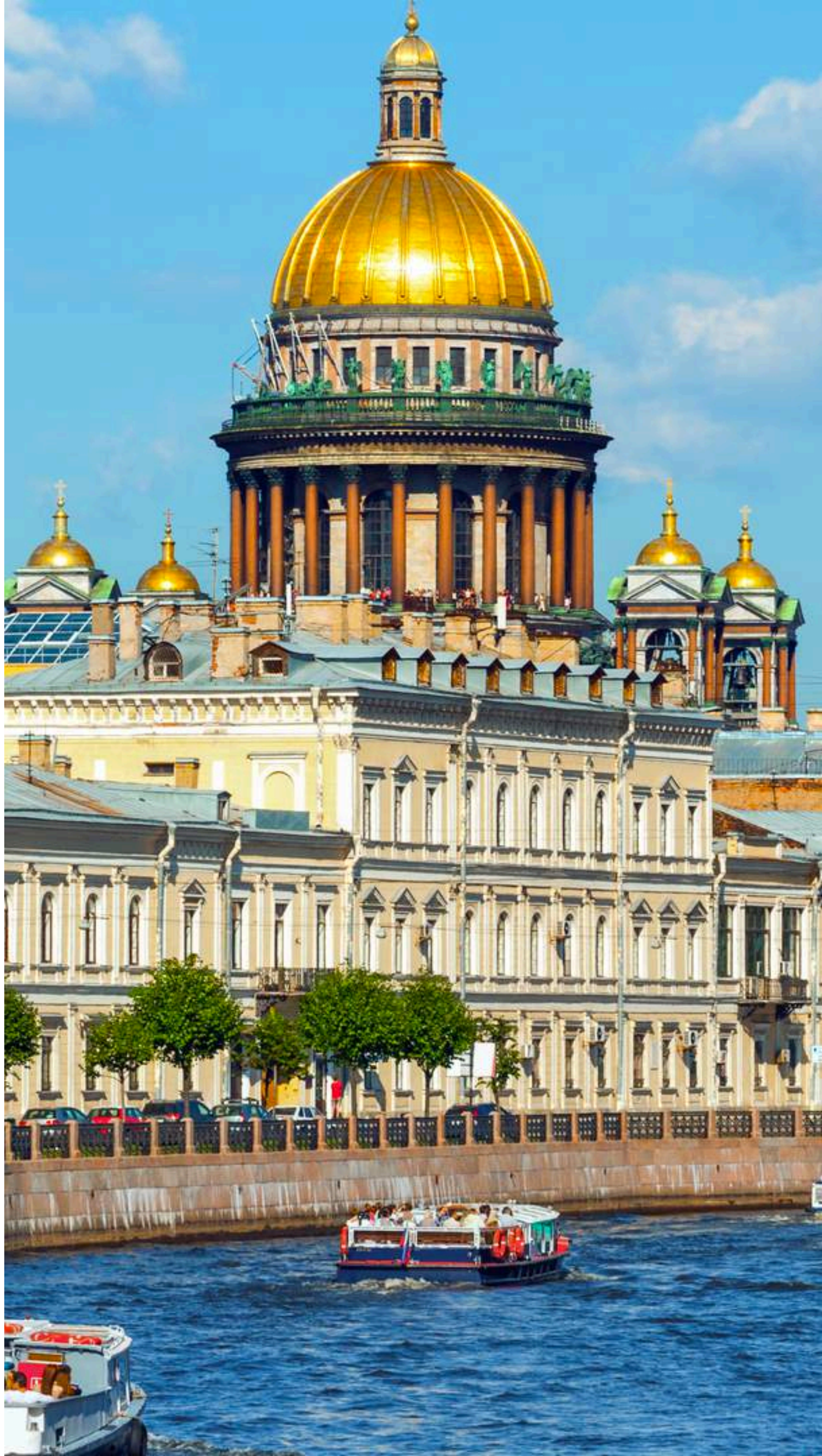
PROGRAMMING IN ROLE ORIENTED CONCURRENT CONTEXTS WITH ROCOCO

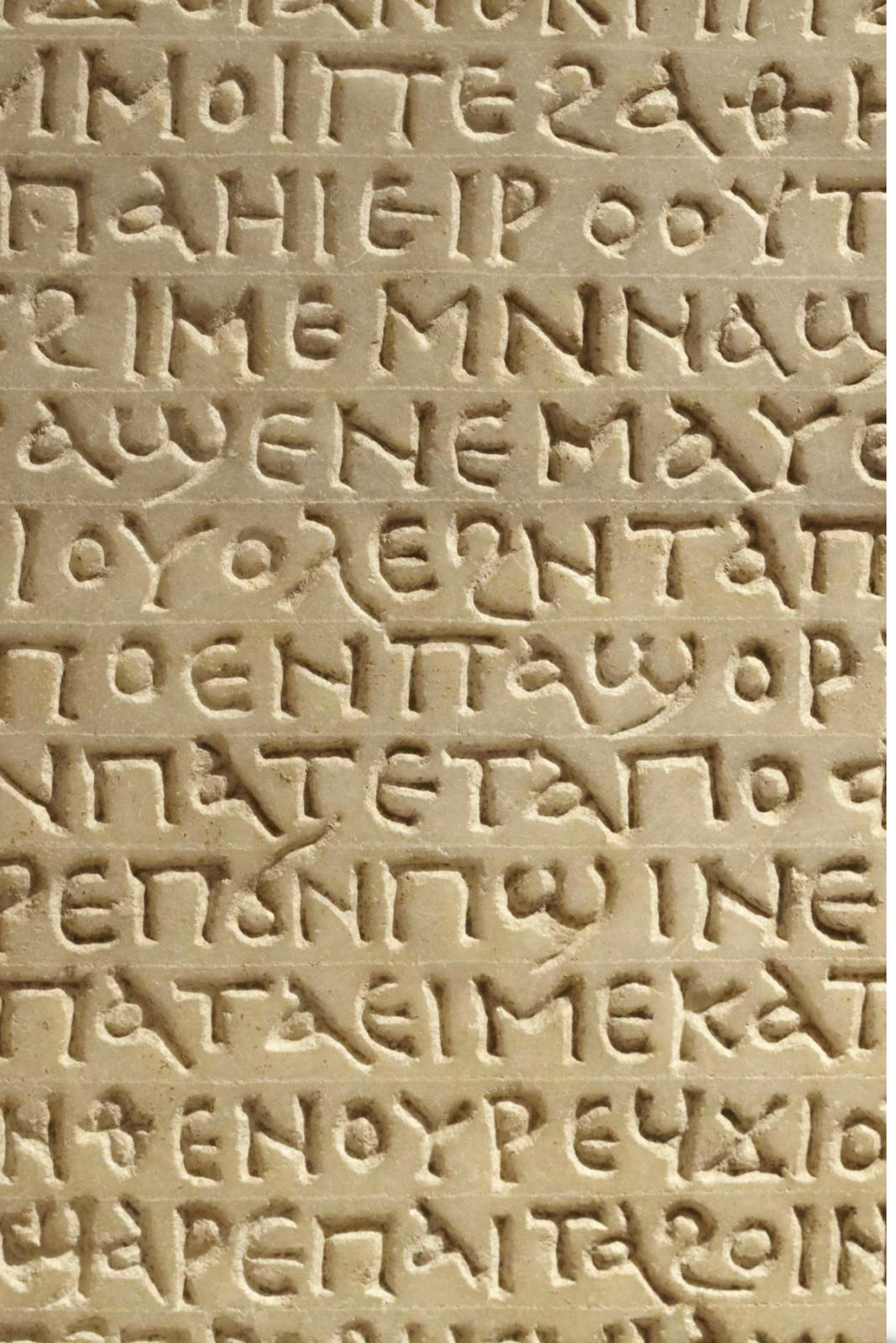
Cevat Balek
Nadia Erdoğan
İstanbul Technical University, Faculty of Computer and Informatics Engineering

SECR

SOFTWARE ENGINEERING CONFERENCE RUSSIA

ST. PETERSBURG 2019





TECHNOLOGY

τέχνη

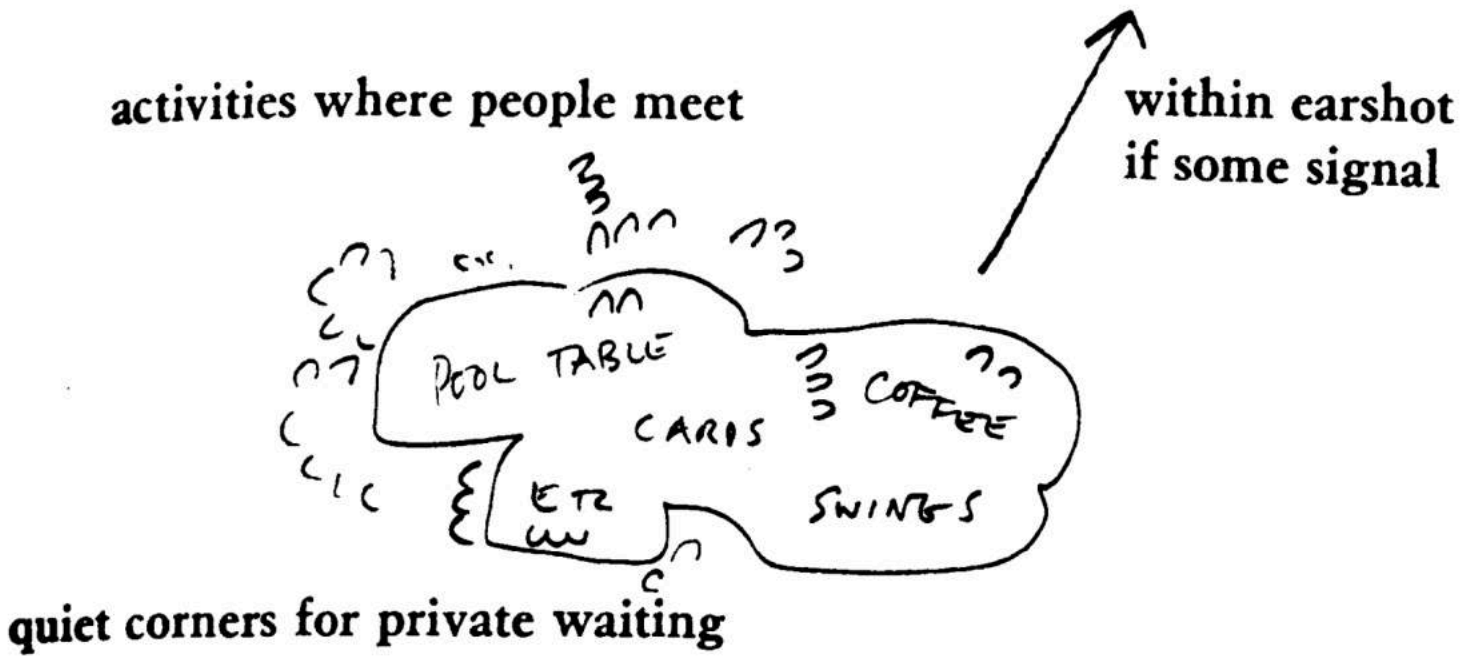
- téchnē means craftsmanship, craft or art in ancient Greeks
- and emphasizes both skill and beauty

VOID

FIRST, THERE WAS VOID

- In engineering or art disciplines, still except software development, architecture has always been about
  - to capture VOID
  - by defining FORM
  - to let both the stability and creativity happening ALIVE within the generated space
- Technical details and even measurements come later
- In software development, however, we ignore the very essense of form relying only on metrics such as coupling and cohesion, # of lines

PATTERN LANGUAGES


activities where people meet
within earshot if some signal
POOL TABLE
CARDS
COFFEE
ETC
SWINGS
quiet corners for private waiting

Gertrud & Cope

- Christopher Alexander introduced pattern languages in 1970s.
- A PATTERN — PLACE TO WAIT
  - In places where people end up waiting, create a situation which makes the waiting positive.
  - Fuse the waiting with some other activity—newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simply waiting.
  - And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence

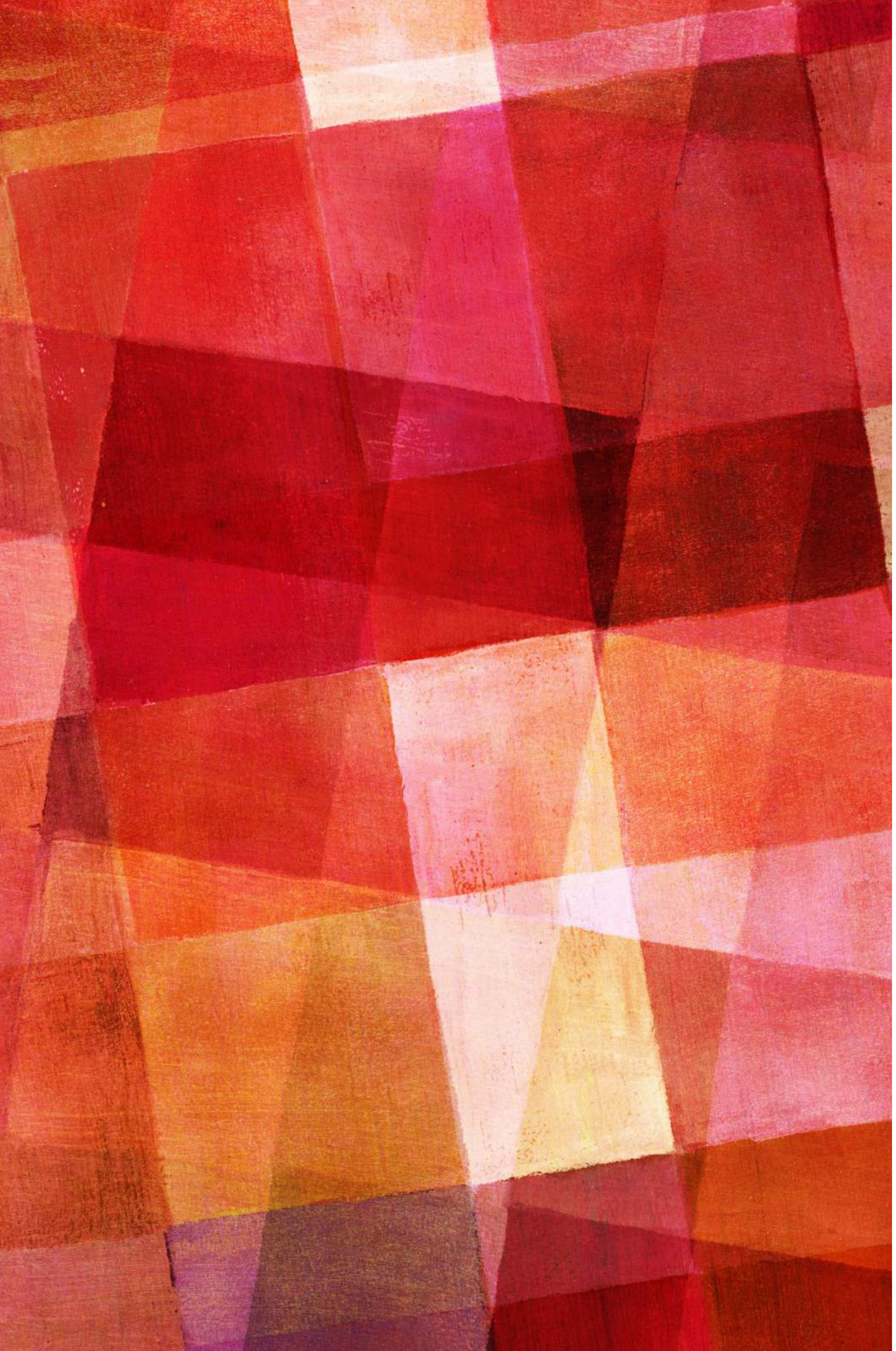
STRUCTURE EVOLVES IN TIME

- TEST OF TIME: Thinking too ahead in time and hardening the structure too early based on immature decisions without considering time will destroy the ALIVENESS of the architecture
- Timeless thinking is completely different and the very basic idea is the awareness of radically different rates of change of different parts of a solution
- Only then, we start to begin understanding the effects and importance of FORM & VOID

… several acts of building, each one done to repair and magnify the product of the previous acts, will slowly generate a larger and more complex whole than any single act can generate

-Christopher Alexander

OCCASVS
7
8
9
10
11
12



PARADIGM SHIFT

- As with other Newtonian and engineering methods, software or even computing itself has born from male principle in charge
  - crisp determination
  - command and control, etc.
- It is not so wrong to state that even the very first idea about the possibility of computers originated in military researches in 1930s
- Female principle, which is ignored for centuries is gaining importance now disguised as systems thinking
  - awareness of environment
  - feelings > intellect, etc.

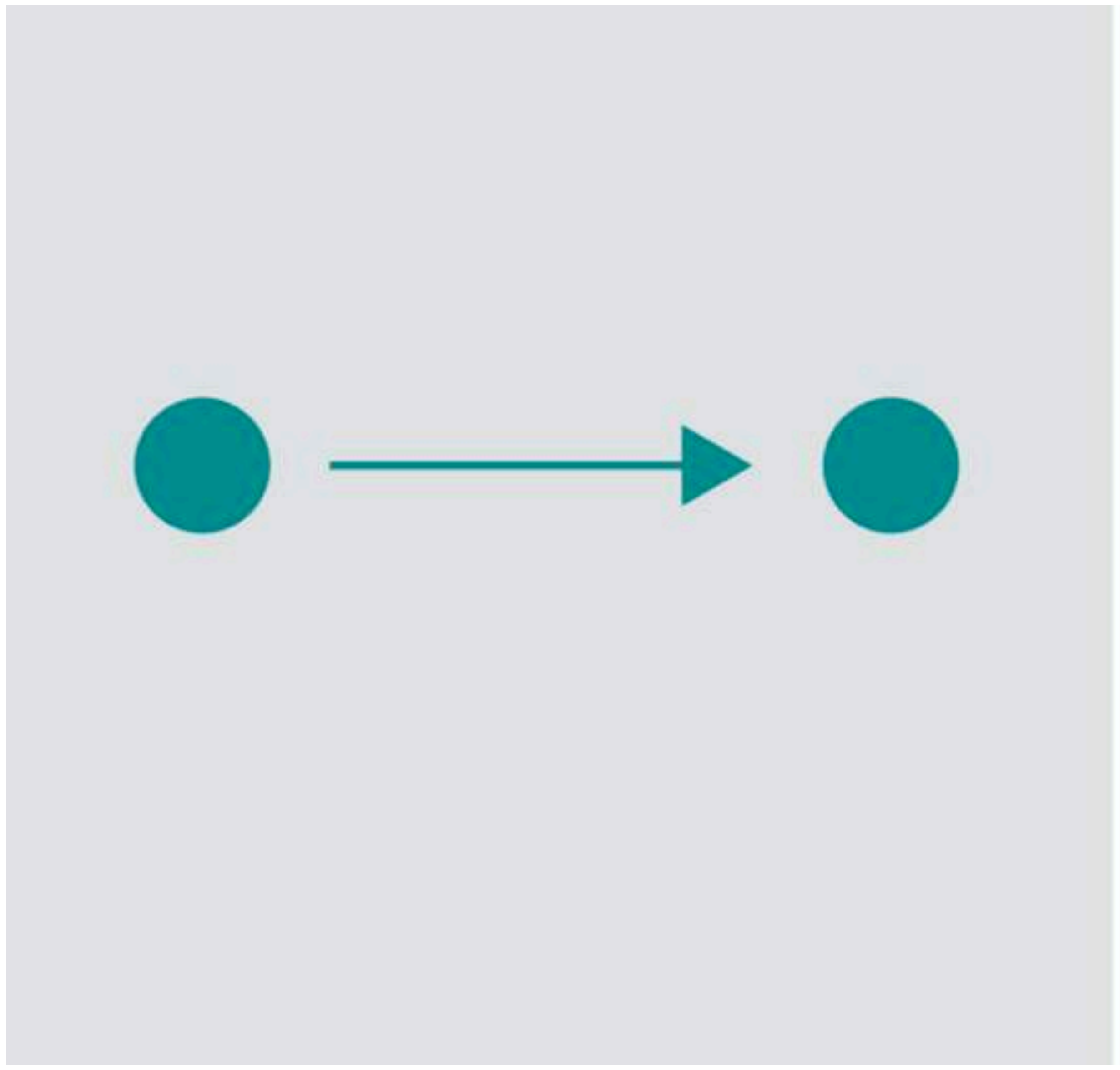
LOOK AT THE KEYS IN YOUR KEYBOARD: AREN'T THEY MILITARY?

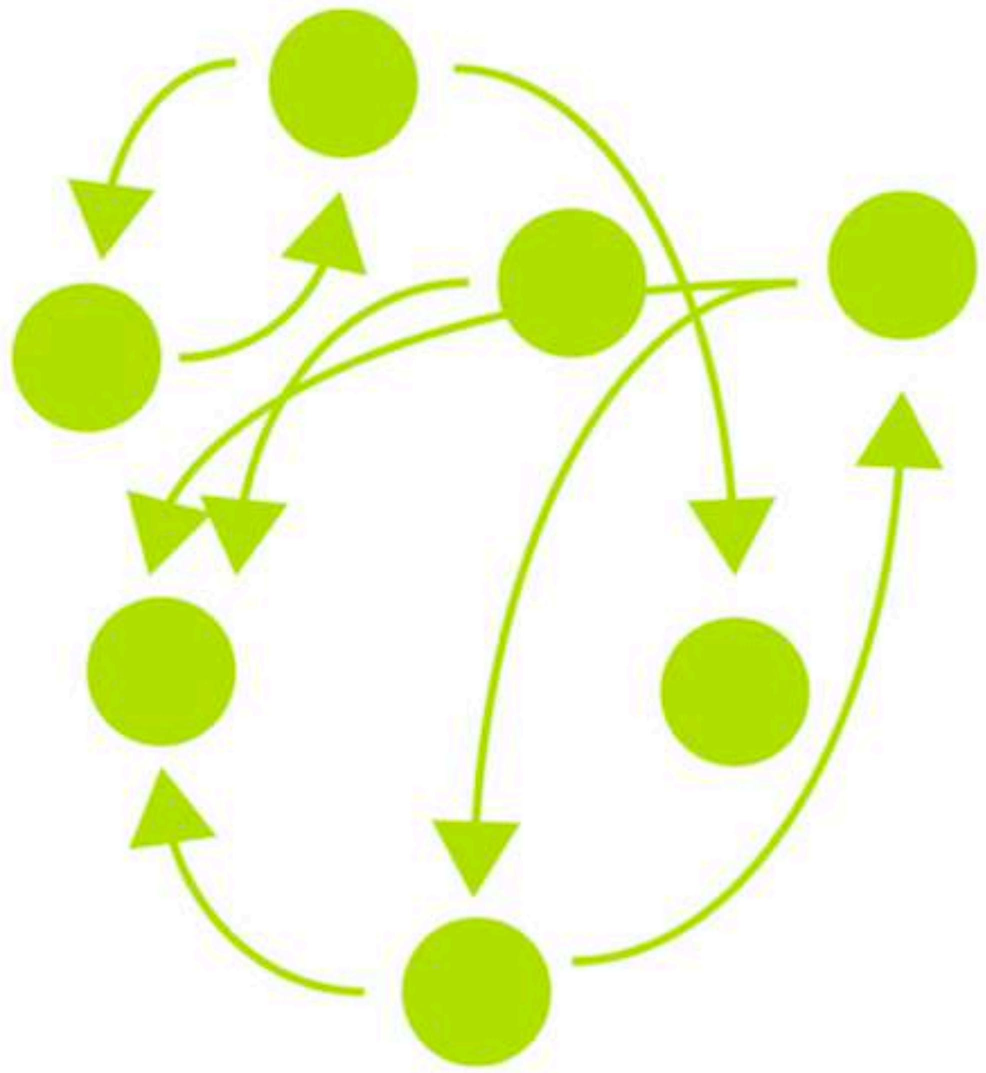
- command
  - enter
  - return (to base)
- control
- escape
- shift? (yes, even that)

- Can we blame the military for dominating the rest of the industry?
- However, this is not enough to explain what's really going on here
- Root cause of the problem is the way we think, our thought process
- We ignored female principle everywhere: in education, at work, …

SYSTEMS THINKING: SCIENCE FOR LIVING SYSTEMS

- We don't scare to look at really complex problems any more
- We are aware of them and develop methods to deal with them
- This is just a little bit off the road on our old way of thinking



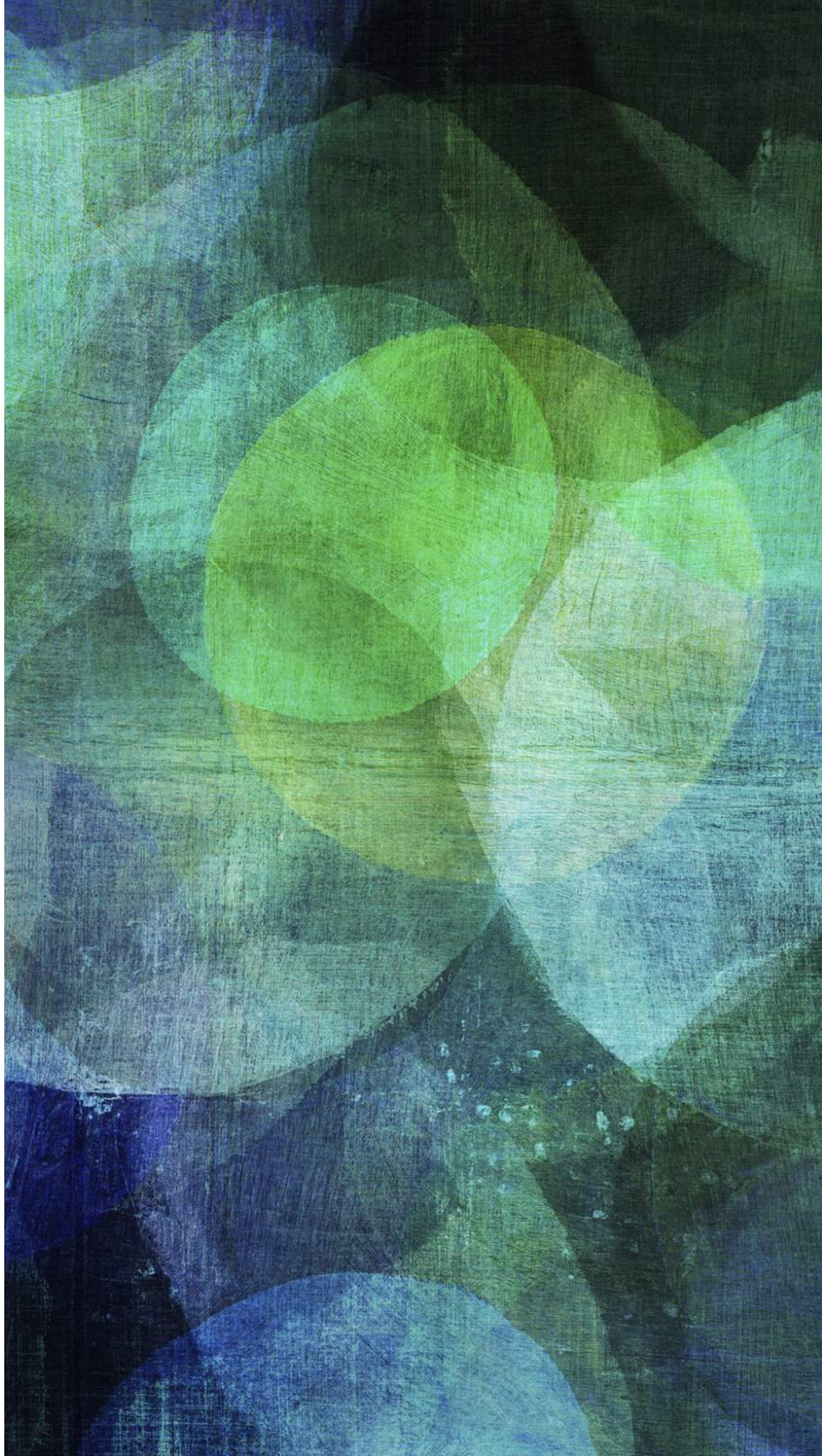


SO, WHAT IS

THE KEY:

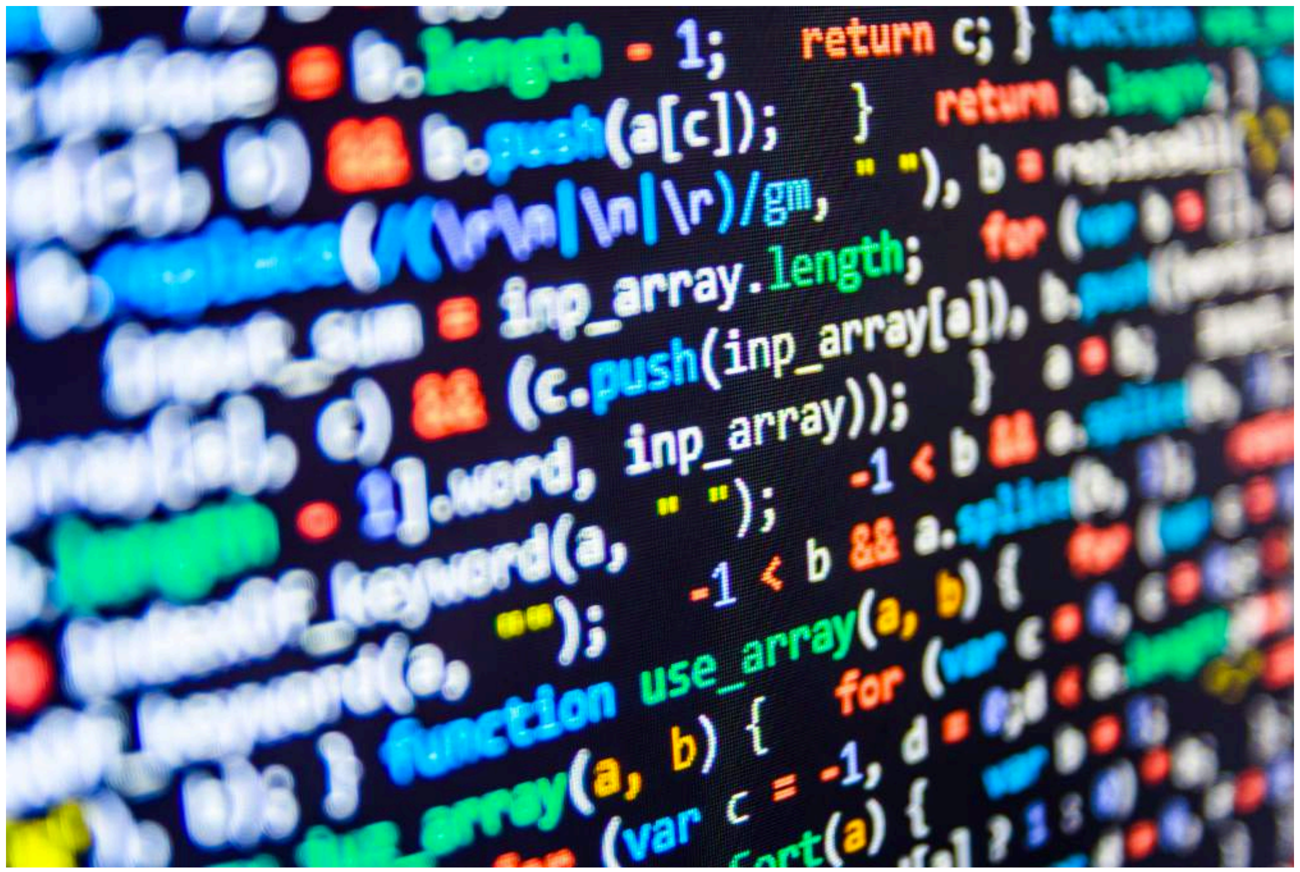
CO-EXISTANCE

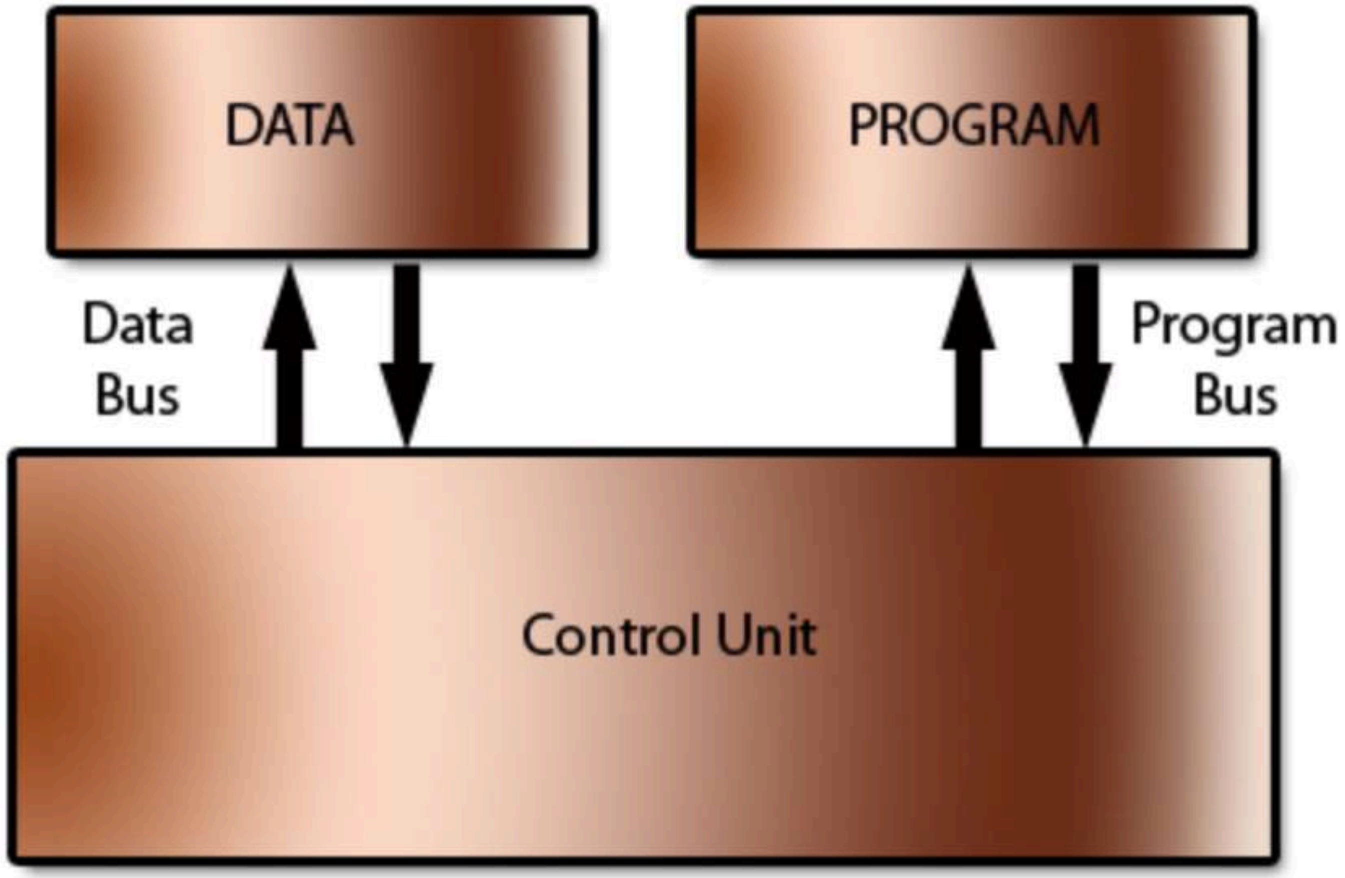
Every living system co-exits in a complex web of relations

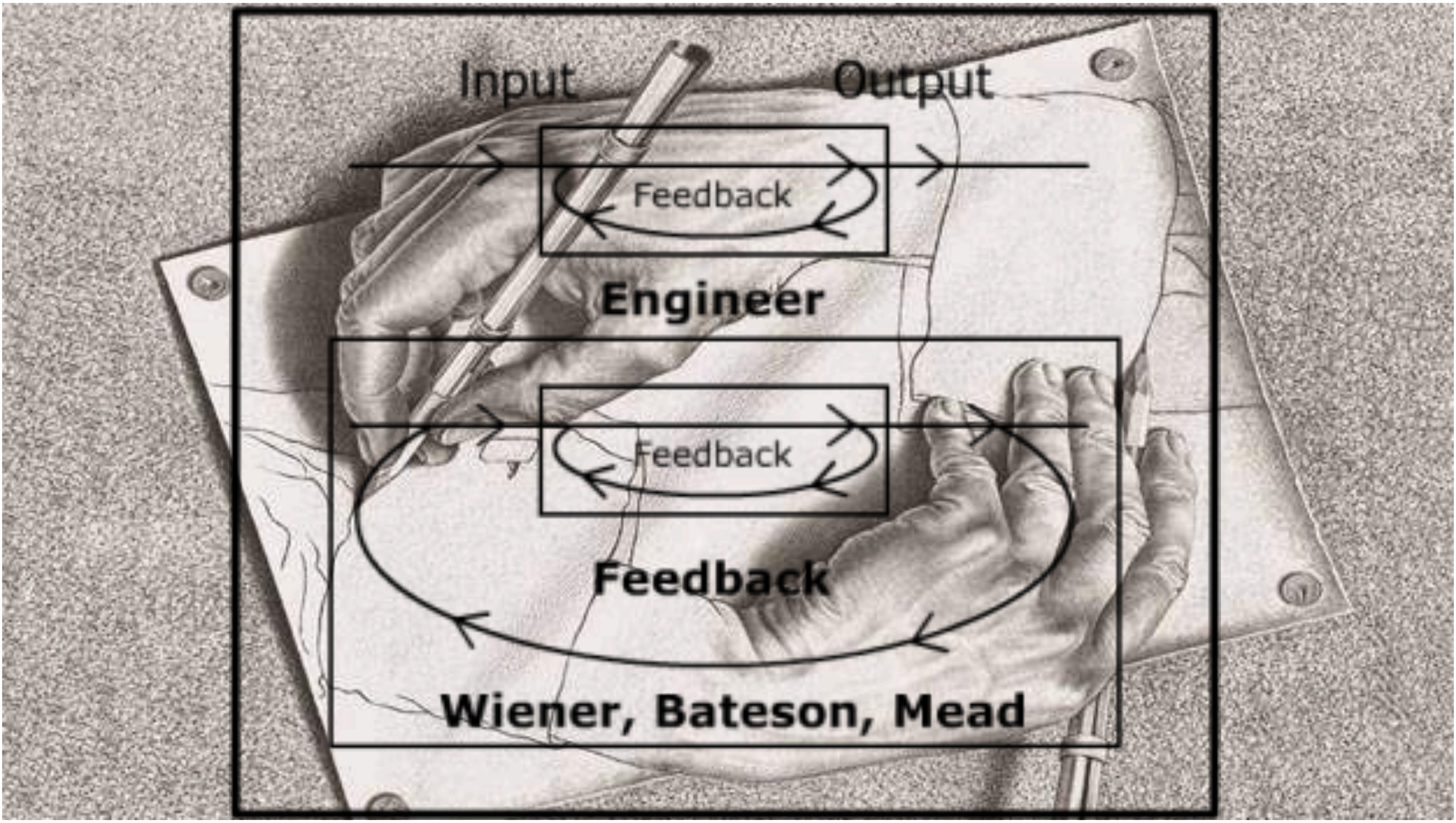


CAN YOU SAY SOFTWARE IS NOT A LIVING SYSTEM?

- Do you think Von-Neumann architecture will survive?
  - Well, it is the foundation of all computing so far
  - It won’t be completely forgotten but it may fade away
- WERE THERE AN ALTERNATIVE?



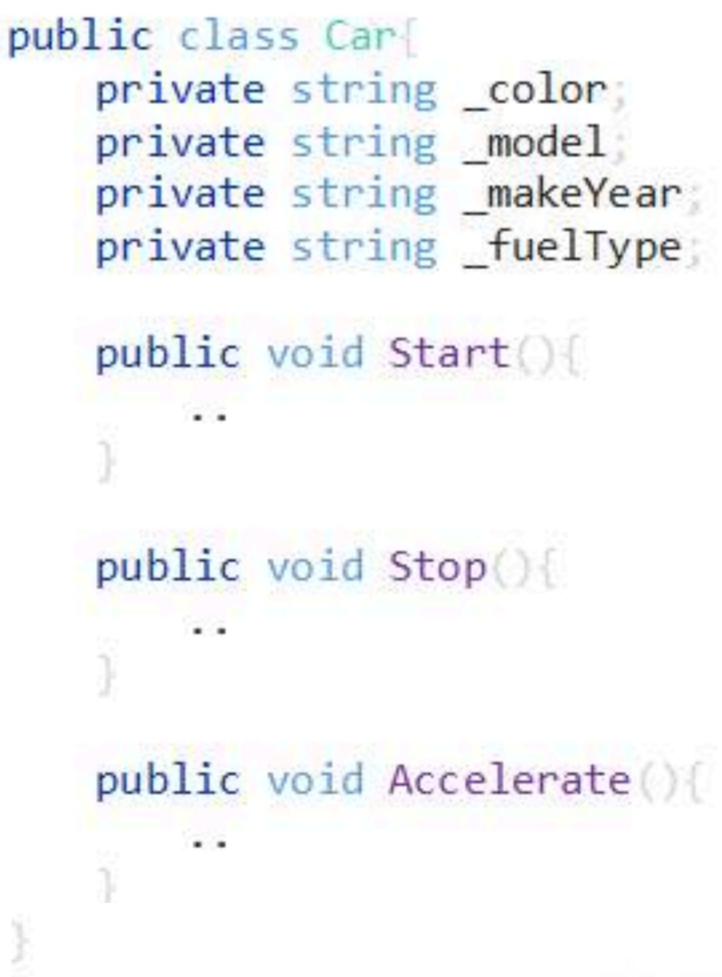



Input
Output
Feedback
Engineer
Feedback
Feedback
Wiener, Bateson, Mead

“

In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing “computer stuff” into things each less strong than the whole -like data structures, procedures and functions which are the usual paraphernalia of programming languages- each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network.

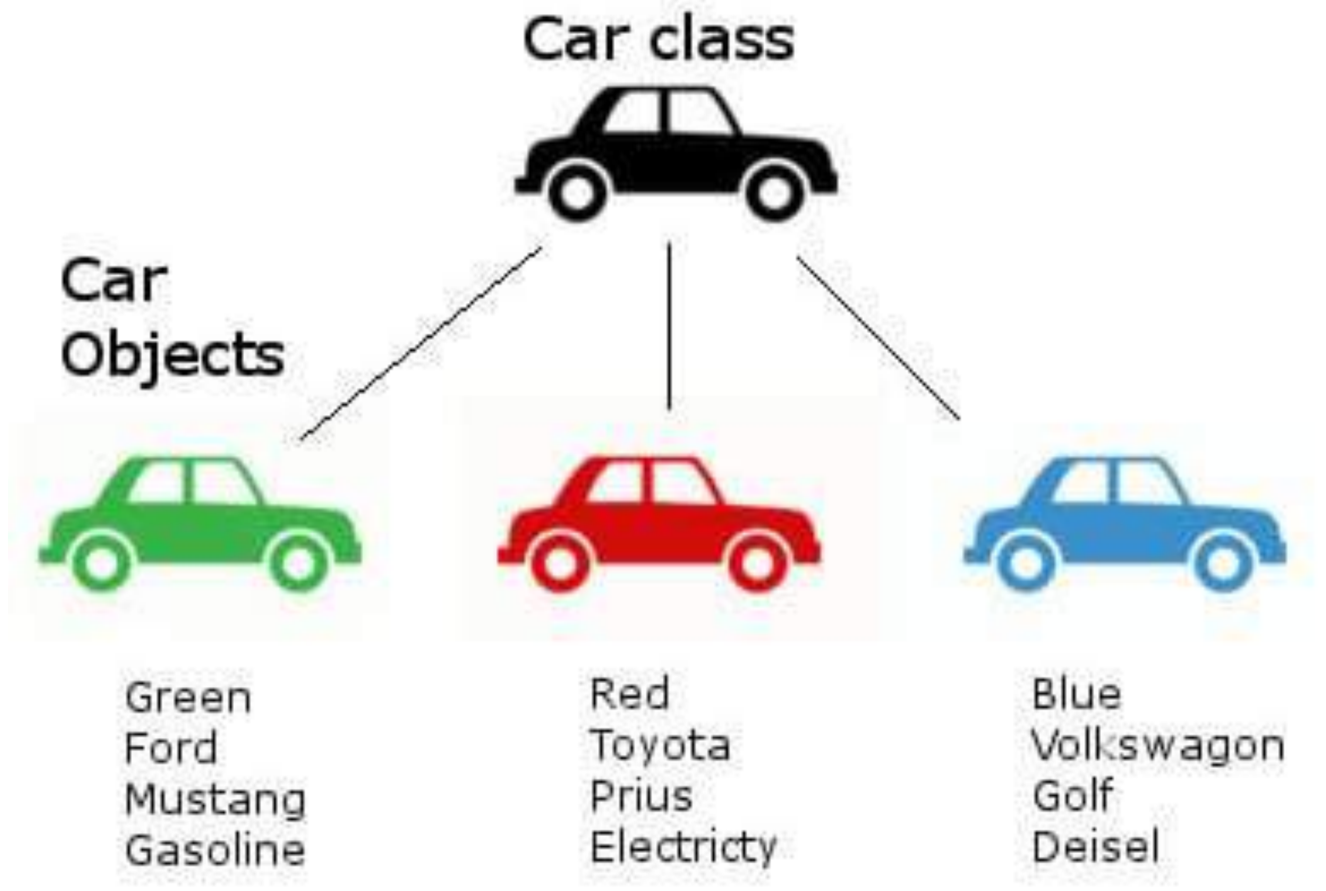
-Alan Kay



```
public class Car{
    private string _color;
    private string _model;
    private string _makeYear;
    private string _fuelType;

    public void Start(){
        ..
    }

    public void Stop(){
        ..
    }

    public void Accelerate(){
        ..
    }
}
```

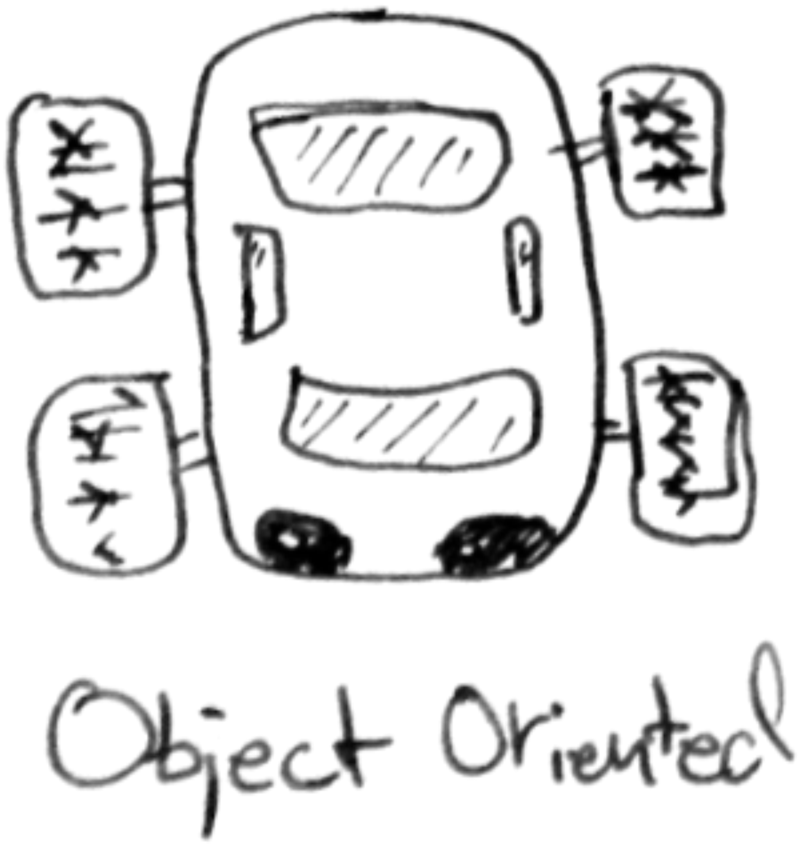
OO AS WE KNOW IT (NOW)

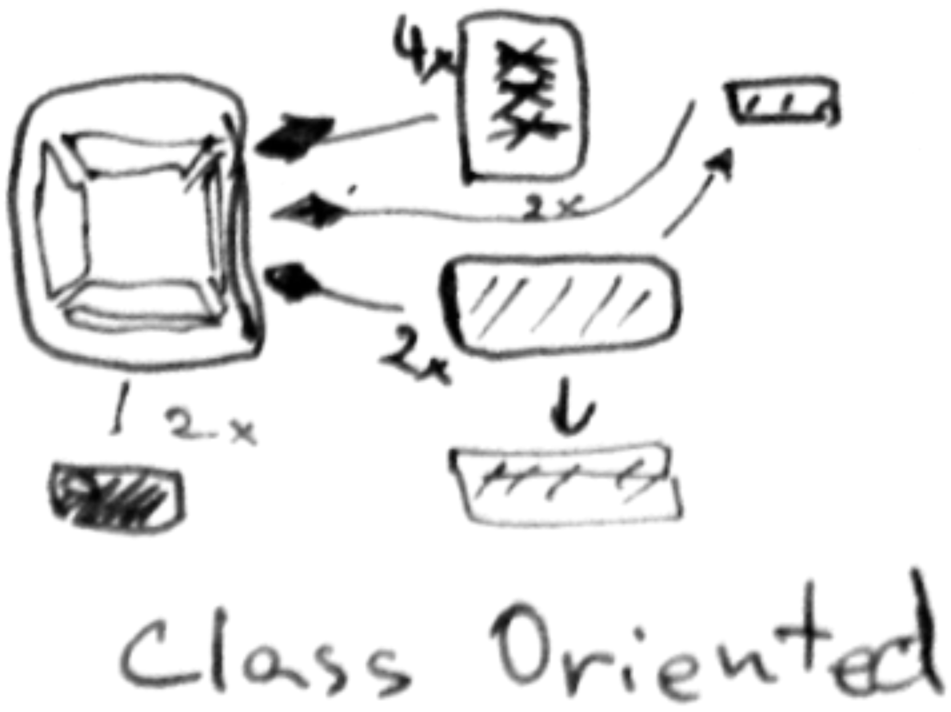
- Although there are methods (procedures) like Start, Stop and Accelerate, car is depicted from an angle of view which highlights data perspective
- This view is too narcissist
- Interaction with environment (such as road) is not designed
- There is no where to save the collaborative code in the source
  - when a human sits on the driver seat (as the driver) how the car collaborates?



CLASS ORIENTED IS NOT OBJECT ORIENTED (NOT EVEN CLOSE)

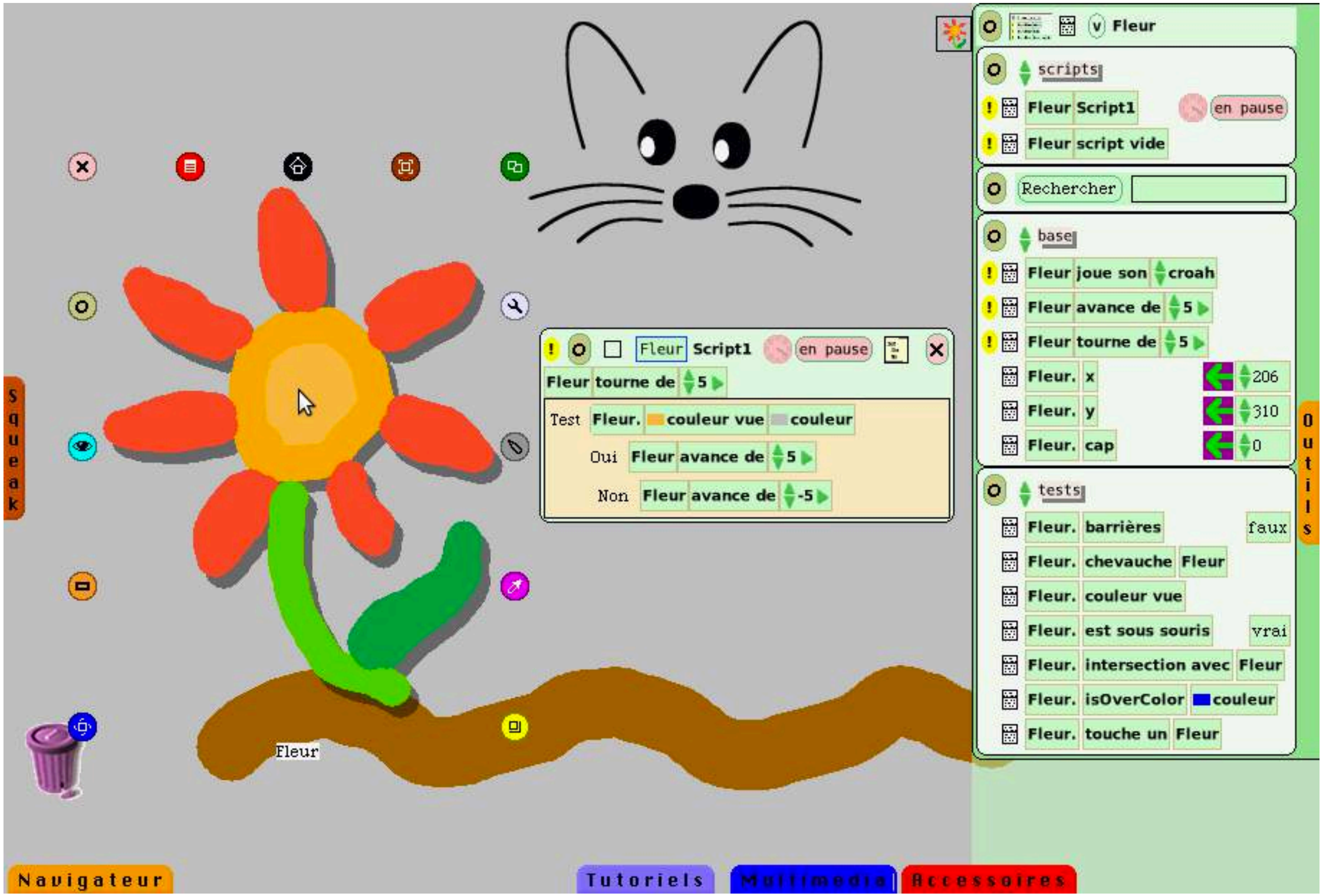
- Systems thinker says that a system is more than the sum of its parts
  - Is it enough to state that a car has 1 steering wheel and 4 tyres, etc. ?





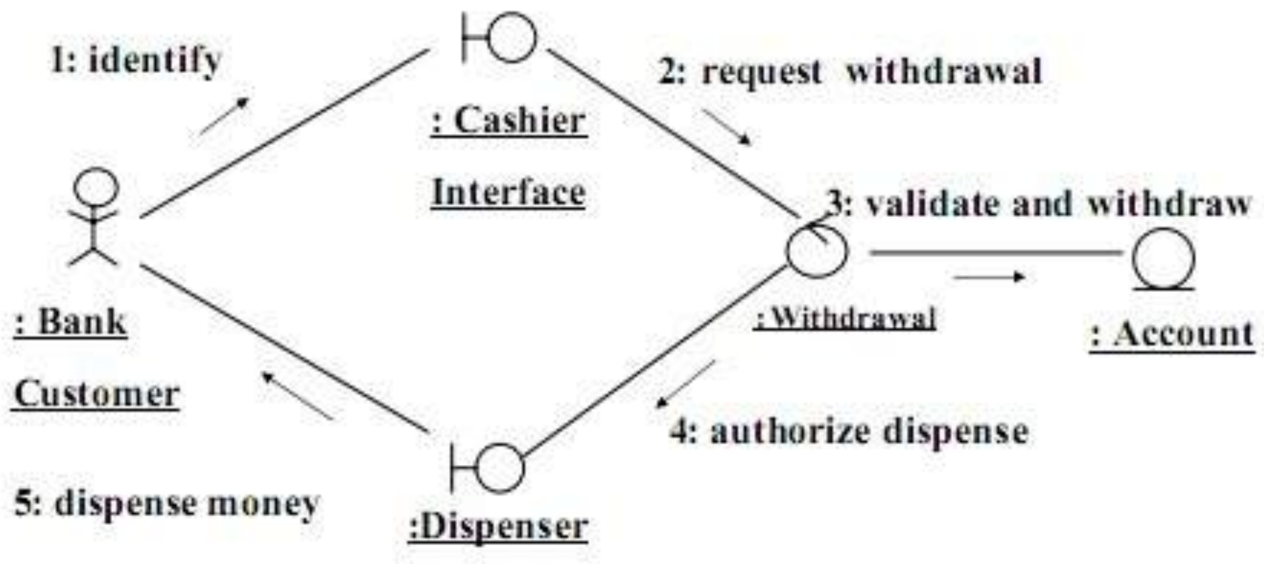
- Defining only the structure is not enough to design a living system
- Interactions should also be defined in bounding contexts (recursively)

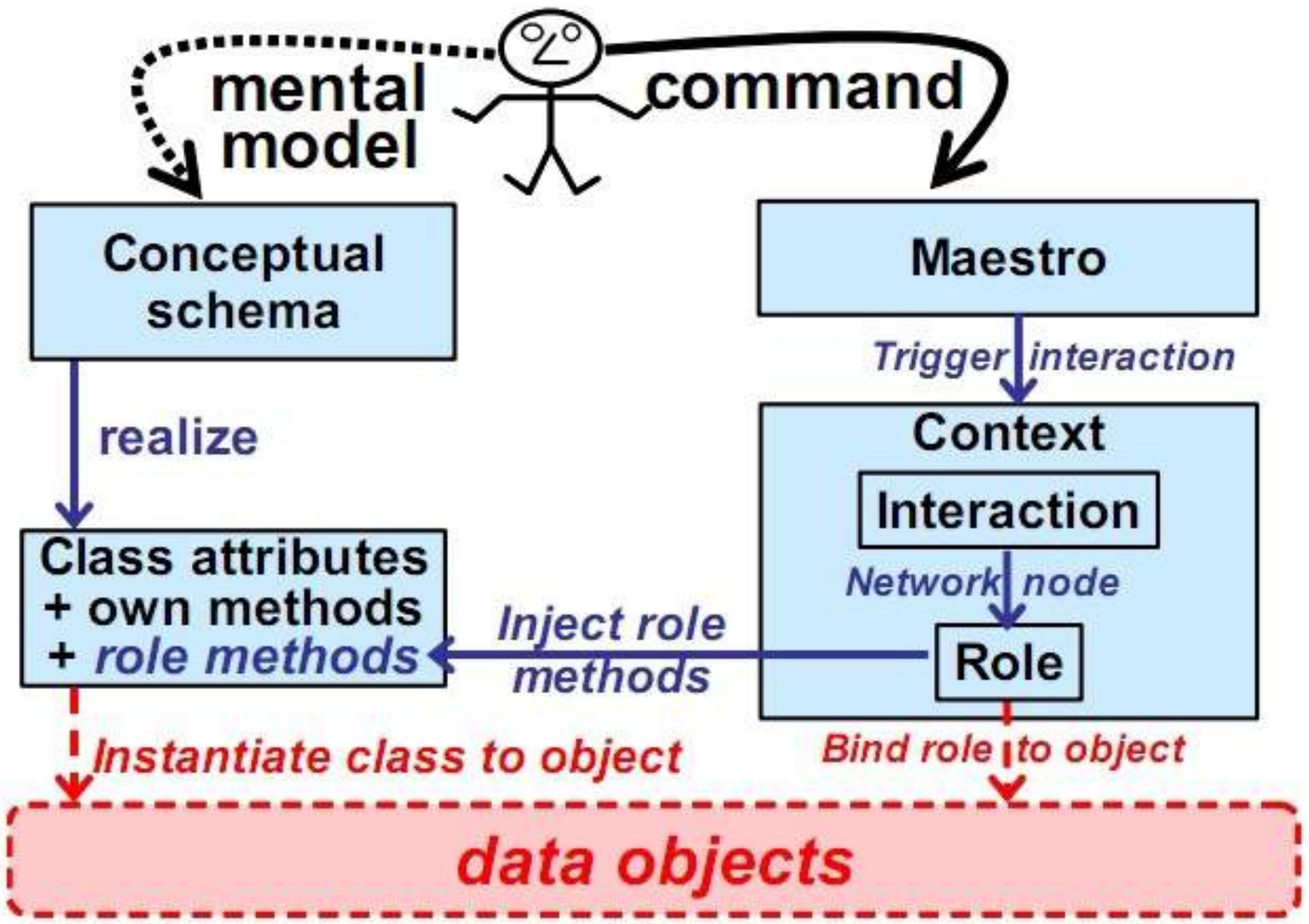
THIS IS THE VISION: OBJECTS AND SCRIPTS ARE HAND IN HAND



```
Fleur Script1   en pause
Fleur tourne de 5
Test  Fleur. couleur vue  couleur
   Oui  Fleur avance de 5
   Non  Fleur avance de -5
```

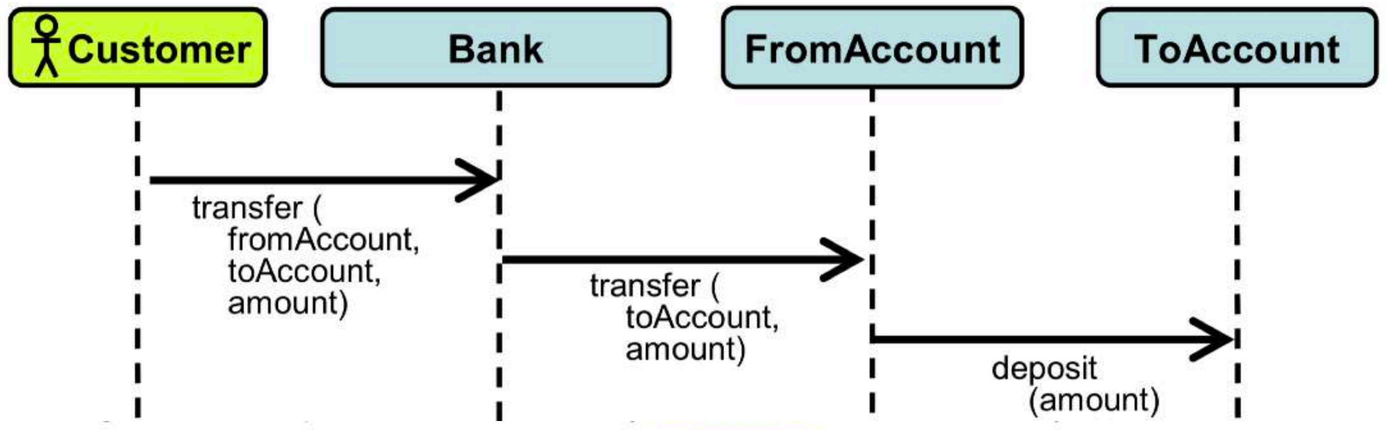
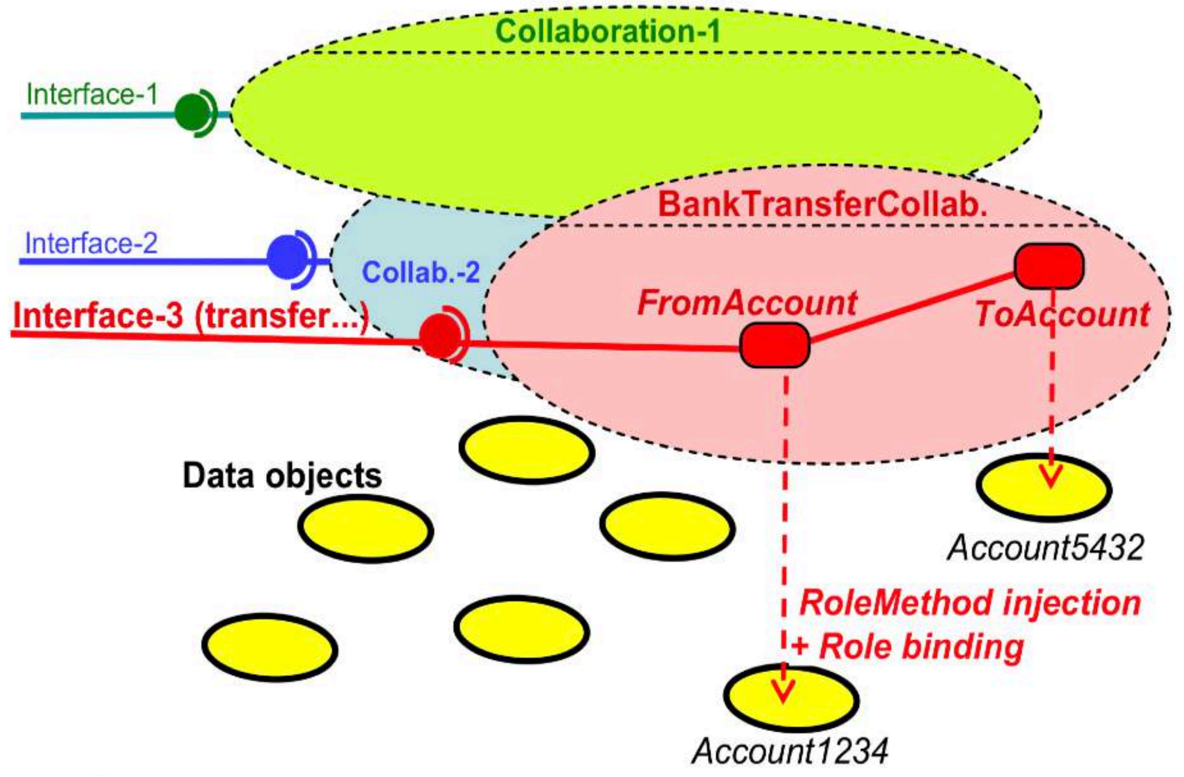
```
Fleur
scripts
  Fleur Script1        en pause
  Fleur script vide
Rechercher
base
  Fleur joue son croah
  Fleur avance de 5
  Fleur tourne de 5
  Fleur. x             206
  Fleur. y             310
  Fleur. cap           0
tests
  Fleur. barrières             faux
  Fleur. chevauche Fleur
  Fleur. couleur vue
  Fleur. est sous souris       vrai
  Fleur. intersection avec Fleur
  Fleur. isOverColor couleur
  Fleur. touche un Fleur
```

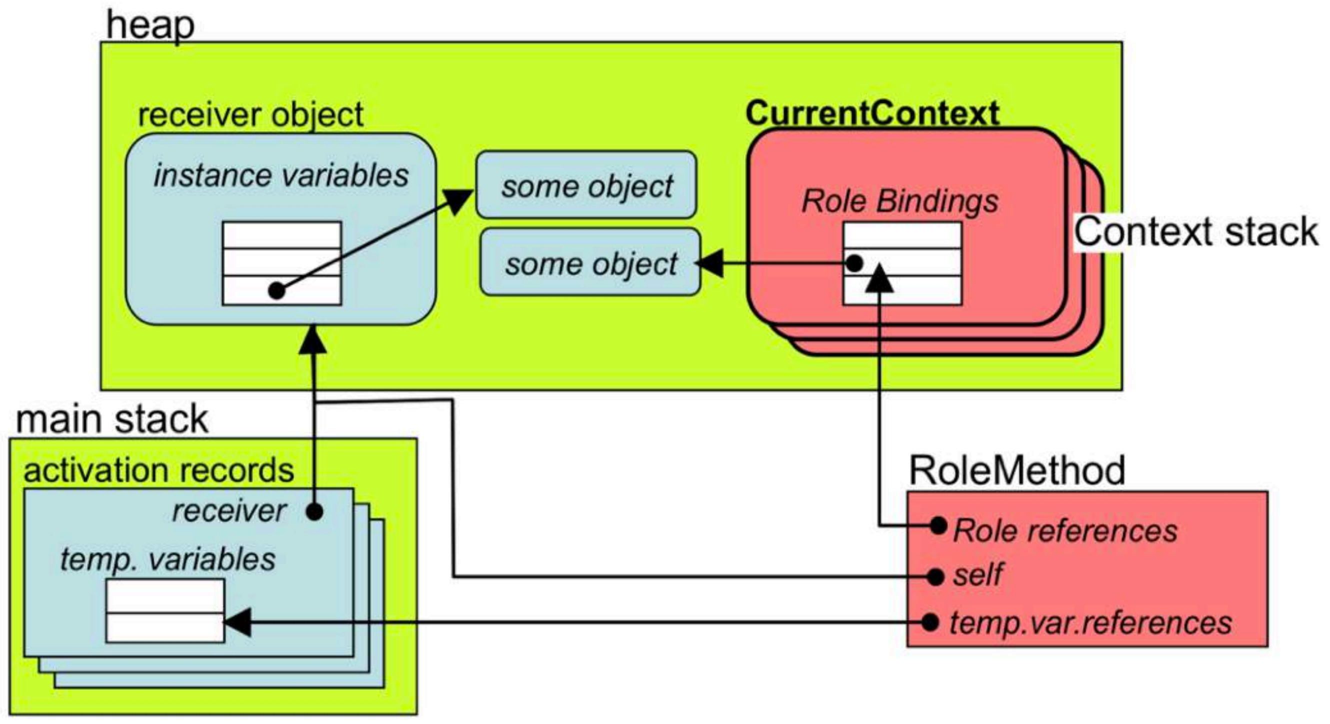




DATA-CONTEXT-INTERACTION

- DCI decouples inheritance mechanism from specifying the behavior in code and restricts the use of classes to only where they are needed and effective most: DATA
- DCI is a natural extension to object orientation to include use cases and the like directly in the code which are thought so far only as analysis and even test artifacts
-


Customer
Bank
FromAccount
ToAccount
transfer (fromAccount, toAccount, amount)
transfer (toAccount, amount)
deposit (amount)




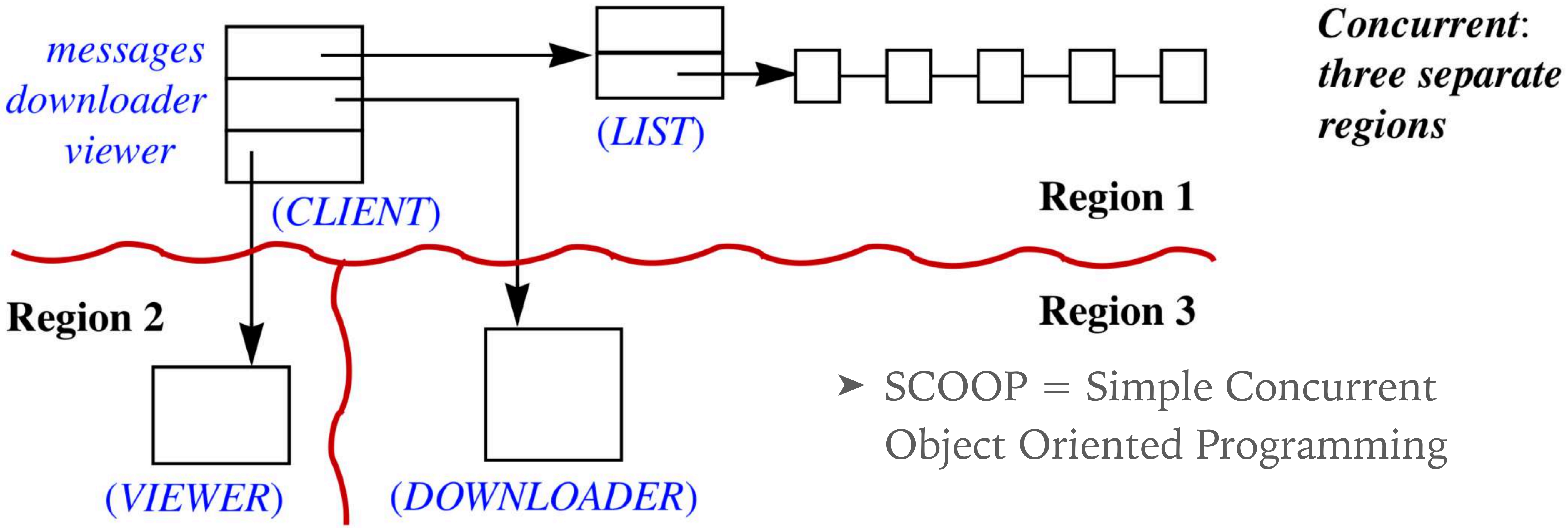
DCI EXECUTION MODEL

- DCI aligns well with end user
- DCI frees the data objects
- DCI allows contextual codes where the objects collaborate
- All the code to orchestrate the objects in a context is in the contextual code, they do not need to be inside the objects
- DCI execution model is single-threaded, is NOT concurrent
- ROCOCO addresses this issue by applying SCOOP technique

SCOOP

- Originated in Eiffel language

```
class CLIENT feature
    messages: LIST [STRING]                  -- Email messages received
    downloader: separate DOWNLOADER          -- Downloading engine
    viewer: separate VIEWER                  -- Message viewing engine
end
```



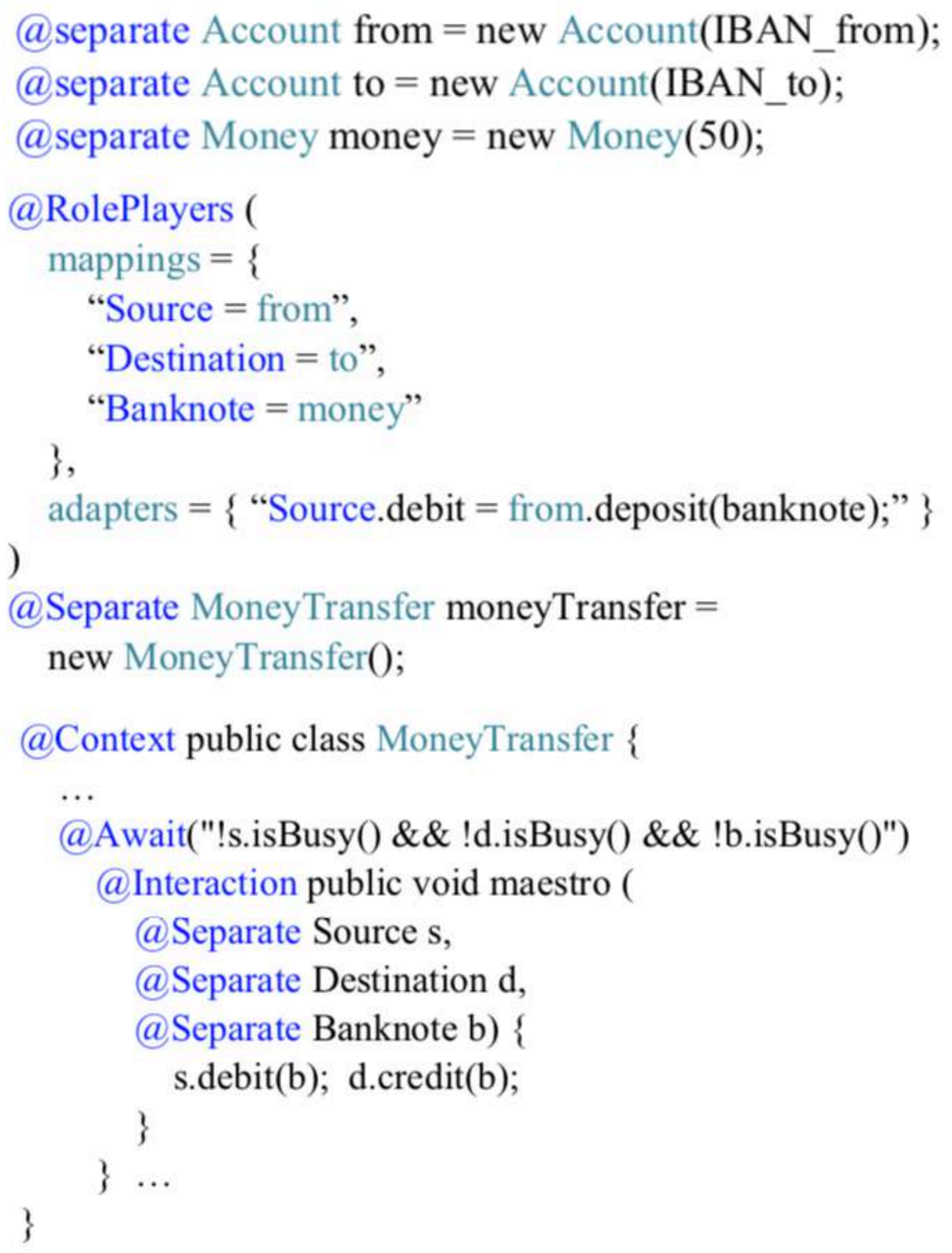
- SCOOP = Simple Concurrent Object Oriented Programming

ROCOCO = DCI + SCOOP

ROCOCO applies SCOOP to enable concurrency in a DCI way of role orientation

ROCOCO is named after a late Baroque ornamental and theatrical style to create surprise and the illusion of motion and drama, and technically means Role Oriented Concurrent Contexts

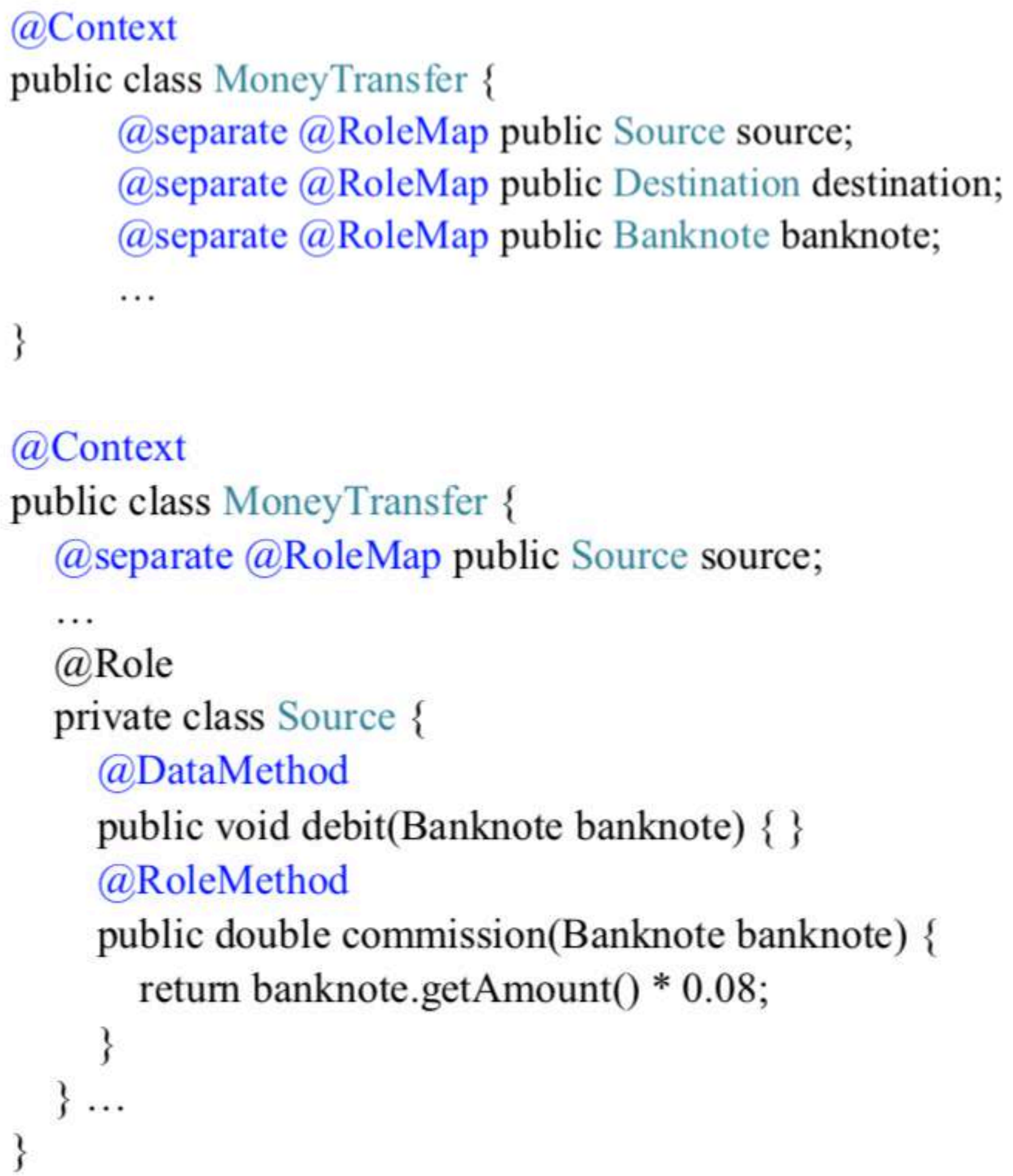
ROCOCO USAGE EXAMPLE



```
@separate Account from = new Account(IBAN_from);
@separate Account to = new Account(IBAN_to);
@separate Money money = new Money(50);

@RolePlayers (
   mappings = {
      “Source = from”,
      “Destination = to”,
      “Banknote = money”
   },
   adapters = { “Source.debit = from.deposit(banknote);” }
)
@Separate MoneyTransfer moneyTransfer =
   new MoneyTransfer();

@Context public class MoneyTransfer {
   …
   @Await("!s.isBusy() && !d.isBusy() && !b.isBusy()")
      @Interaction public void maestro (
         @Separate Source s,
         @Separate Destination d,
         @Separate Banknote b) {
            s.debit(b);  d.credit(b);
         }
      } …
}
```

- from, to and money objects are data objects
- these data objects are being used in MoneyTransfer context which knows all the actions and information about bank transfer, commissions, etc.
- @separate means that these objects and the context may all live in separate threads
- @RolePlayers maps each role to the data object which will play that role in the context

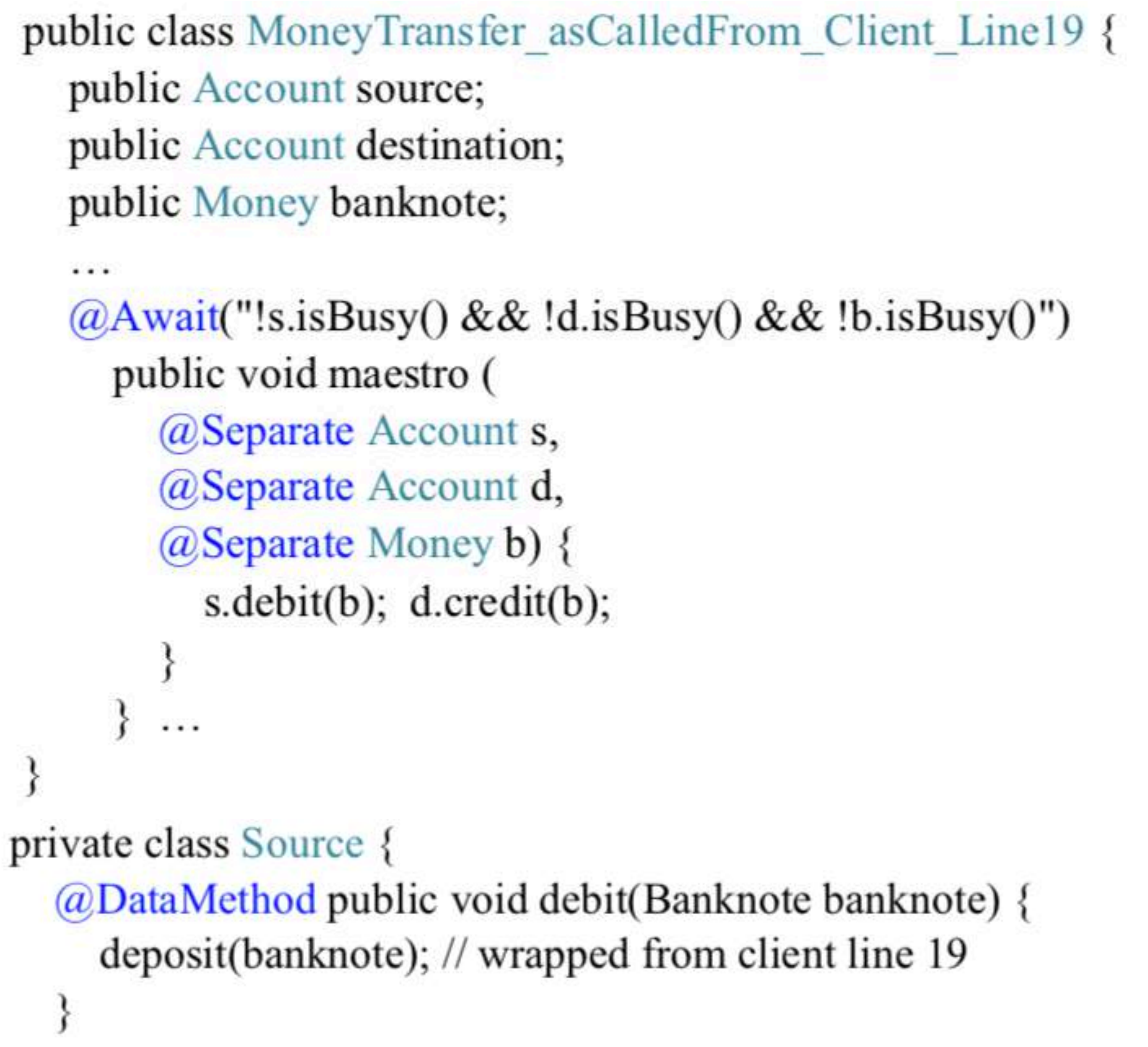
ROCOCO CONTEXT



```
@Context
public class MoneyTransfer {
        @separate @RoleMap public Source source;
        @separate @RoleMap public Destination destination;
        @separate @RoleMap public Banknote banknote;
        …
}

@Context
public class MoneyTransfer {
   @separate @RoleMap public Source source;
   …
   @Role
   private class Source {
      @DataMethod
      public void debit(Banknote banknote) { }
      @RoleMethod
      public double commission(Banknote banknote) {
         return banknote.getAmount() * 0.08;
      }
   } …
}
```

- In ROCOCO, a context is a class annotated with *@Context*
- The main responsibility of a context is to bring data objects together that will interact as role players within the context
- Good contexts are stateless, so a warning can be given if there is any field in the context which is not annotated with *@RoleMap*
- Context is constructed in an atomic call, once established, the role players do not change

ROCOCO TRANSFORMATION



```
public class MoneyTransfer_asCalledFrom_Client_Line19 {
  public Account source;
  public Account destination;
  public Money banknote;
  ...
  @Await("!s.isBusy() && !d.isBusy() && !b.isBusy()")
    public void maestro (
      @Separate Account s,
      @Separate Account d,
      @Separate Money b) {
        s.debit(b);  d.credit(b);
      }
    } ...
}
private class Source {
  @DataMethod public void debit(Banknote banknote) {
    deposit(banknote); // wrapped from client line 19
  }
```

- ROCOCO uses eclipse JAVA Development Toolkit (eclipse JDT) to perform the source code to source code transformation necessary and JSCOOP (an experimental port of SCOOP to JAVA)

- DCI to OOwR (object oriented code with roles) transformation processes only DCI annotations and leaves SCOOP annotations intact.
- DCI code is reduced to OOwR code so that OOwR to ROCOCO transformation is possible through SCOOP which is designed for OO may be applied later.
- After this stage, transformed (expanded) OOwR source code will include all information about role oriented aspects of the computation
- OOwR to ROCOCO transformation processes the SCOOP related annotations left intact by the DCI to OOwR transformation above

ROCOCO BENEFITS

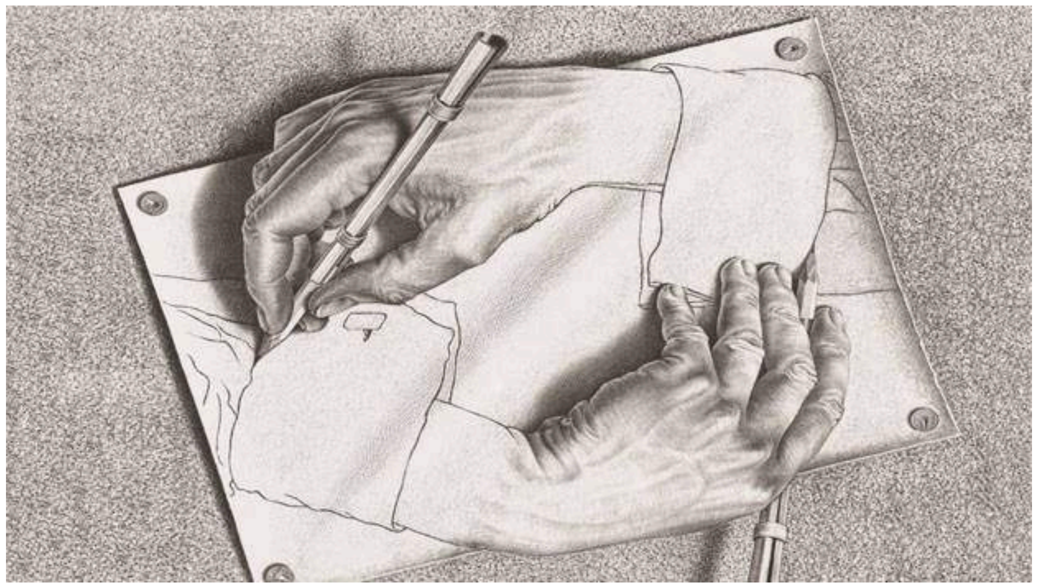
Relaxes the single-threaded constraint in DCI so that collaboration code remains readable

IDE understands the programmer

Correctness proofs become easier

Complete independence of objects from interfaces via data method adapters

paves a way to let the compiler decide optimally whether to run the routine concurrently or not



SECR

İTÜ

Thank you for listening …

Cevat Balek
cevatbalek@gmail.com