Introduction to Programming paradigms

different perspectives (to try) to solve problems

17 September 2014,

Introduction to Information Systems – **Practical class**

Giovanni Sileno g.sileno@uva.nl

Leibniz Center for Law

University of Amsterdam

Problem Analysis

Problem/Paradigm association?

- The puzzle of the farmer with goose, fox and beans.
- How much is 2 * 2 + 4 ?
- Prepare a dish of spaghetti.
- Manage your collection of books.
- Given $f(a, b) = a^2 - b^2$, how much is $f(2, 3)$?
- Schedule your weekly physical exercises, considering your personal and professional appointments.
- Find the max of 1, 5, 2, 9, 4, 6, 3, 8, 7.
- Order the same sequence.
- Calculate the taxes you have to pay.

Leibniz Center for Law

Programming

Control flow operators

- The **control flow** basically describes the sequential order in which instructions are evaluated.
- *Control flow operators* modifies such order.

"Dangerous" control flow operators

- Certain operators disrupt the sequence, in the sense that do not allow you to return to the stream you were before.

“Dangerous” control flow operators

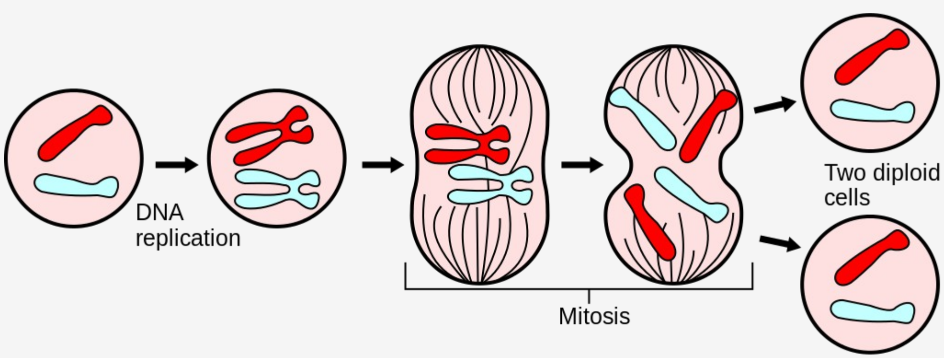
- Certain operators disrupt the sequence, in the sense that do not allow you to return to the stream you were before.
  - Jumps (GOTO)

“Dangerous” control flow operators

- Certain operators disrupt the sequence, in the sense that do not allow you to return to the stream you were before.
  - Jumps (GOTO)
  - Exceptions

“Dangerous” control flow operators

- Certain operators disrupt the sequence, in the sense that do not allow you to return to the stream you were before.
  - Jumps (GOTO)
  - Exceptions
  - Threads



“Dangerous” control flow operators

- As long as you know your code well, this is not necessarily a problem.
- However, when the program is not yours, or it grows in complexity with the development, unstructured control flow operators become difficult to follow.

“Dangerous” control flow operators

- As long as you know your code well, this is not necessarily a problem.
- However, when the program is not yours, or it grows in complexity with the development, unstructured control flow operators become difficult to follow.
- **Worst scenario**
  - complex program
  - modified by many people
  - with a long life cycle

Does anyone enjoy spaghetti code?

Leibniz Center for Law

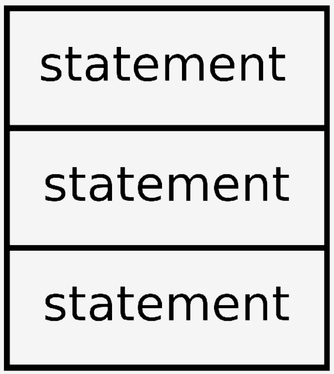
Structured programming

Structured programming

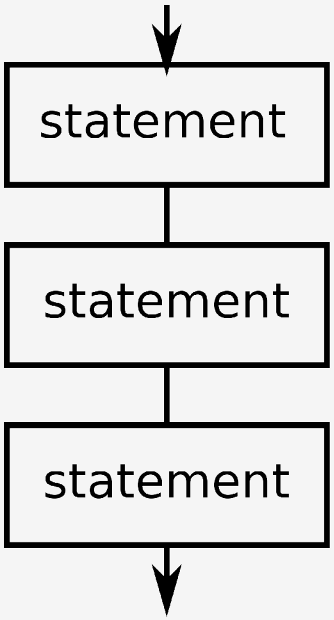
- Structured programming was born to extend imperative programming with control flow operators, while avoiding the use of unconditional branchs (e.g. GOTO).
- It leverages *visual diagramming* techniques as *flow charts* or *Nassi–Shneiderman* diagrams.

Sequential execution

- Normally execution occurs sequentially.



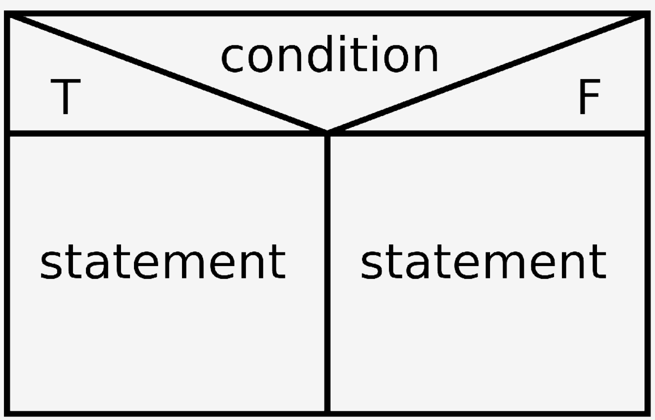
Nassi–Shneiderman (NS) diagrams

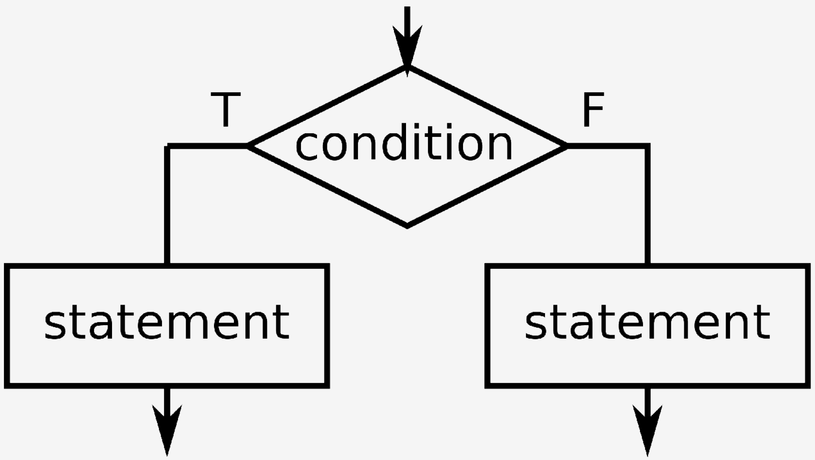


Flow charts

Conditional (IF .. THEN .. ELSE)

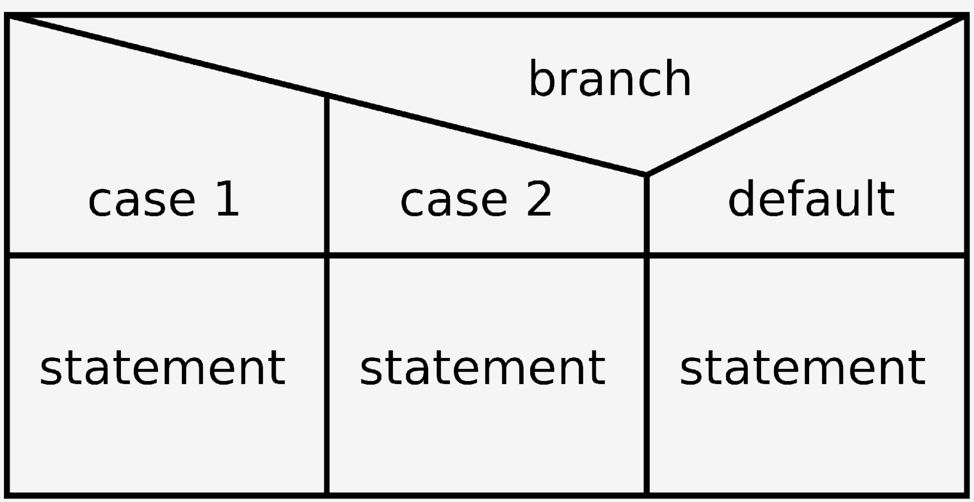
- Used for binary evaluations (true or false).
- IF a certain condition is true THEN perform something, ELSE perform something else

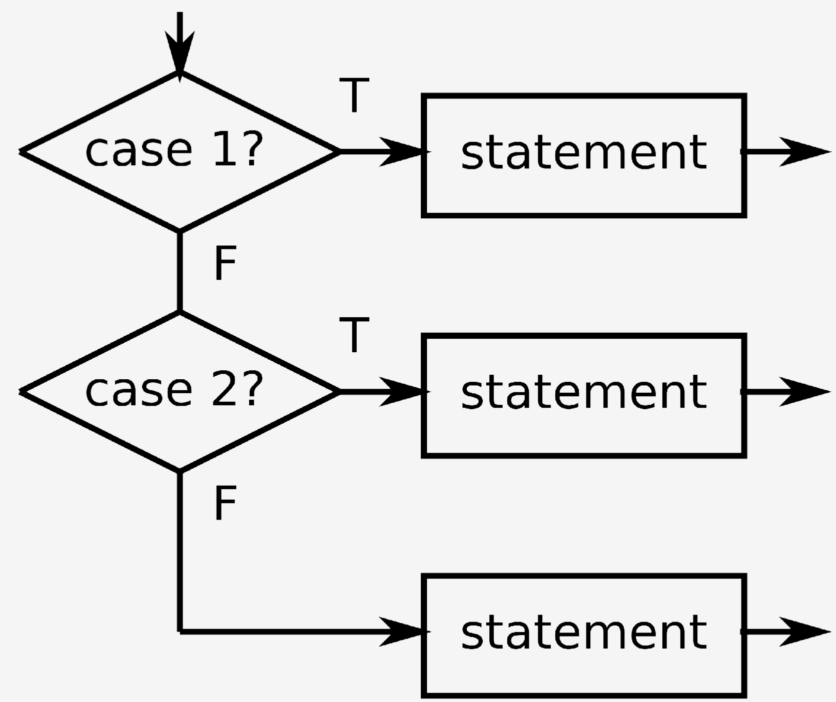




Conditional (SWITCH/CHOICE)

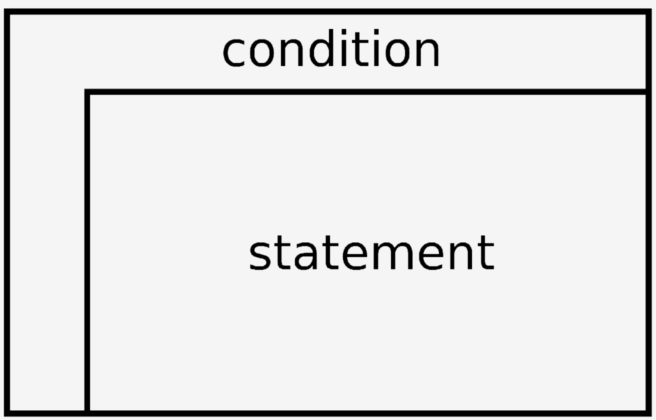
- This conditional is used for multiple choices. *Default* is the "other", not explicitly defined case.

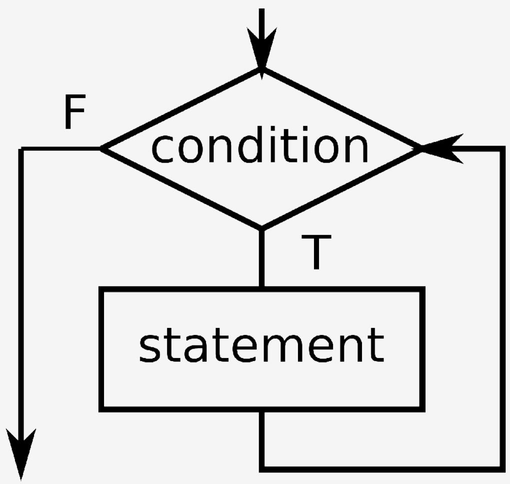




Loop (WHILE .. DO ..)

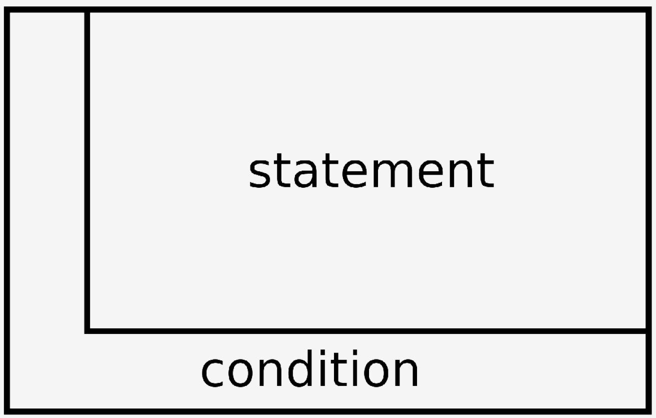
- This loop repeats its code as much as the condition is true.

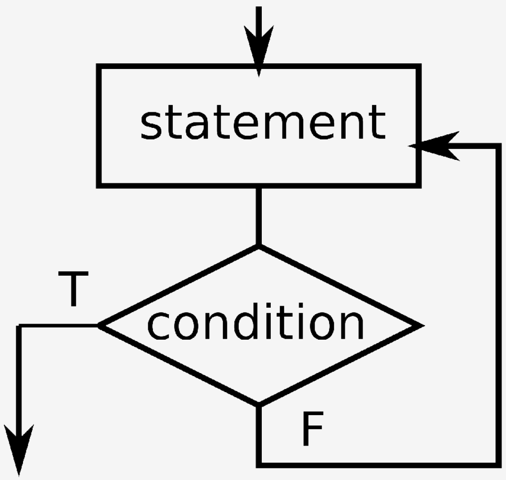




Loop (DO .. UNTIL ..)

- This loop repeats its code until the condition becomes true.



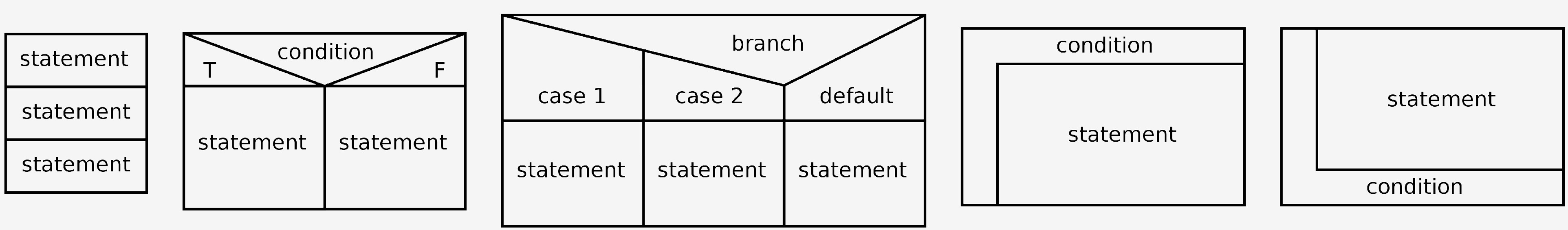


Draw a *Nassi–Shneiderman* diagram of the cooking of a dish of *spaghetti*.

- If there are sieved tomatoes, you cook the tomato sauce, with salt and basilic. Add a bit of sugar if the tomatoes are acid.

- Otherwise you do a *carbonara*. You fry sliced bacon in the pan, and when the pasta is ready, break the eggs adding parmisan and a bit of pepper in the pasta pot.
- The pasta is cooked letting the water to boil in a pot, adding salt, and then the pasta. Wait the suggested cooking time.

IF .. THEN .. SWITCH/CASE .. WHILE .. DO .. DO .. UNTIL..



Decomposition

Decomposition (or factoring)

- Decomposition is a strategy for organizing a program as a number of parts.
- The objective of decomposition is to increase modularity of the program and its maintainability.

Decomposition (or factoring)

- Decomposition is a strategy for organizing a program as a number of parts.
- The objective of decomposition is to increase modularity of the program and its maintainability.
- We can decompose both **data** (the logic) and **procedures/functions** (the control).

Divide et impera (divide and conquer)

- Decomposition allows to take a strategic algorithmic approach
- Rather than facing the complete problem, we tackle it down to smaller (and simpler) independent components.

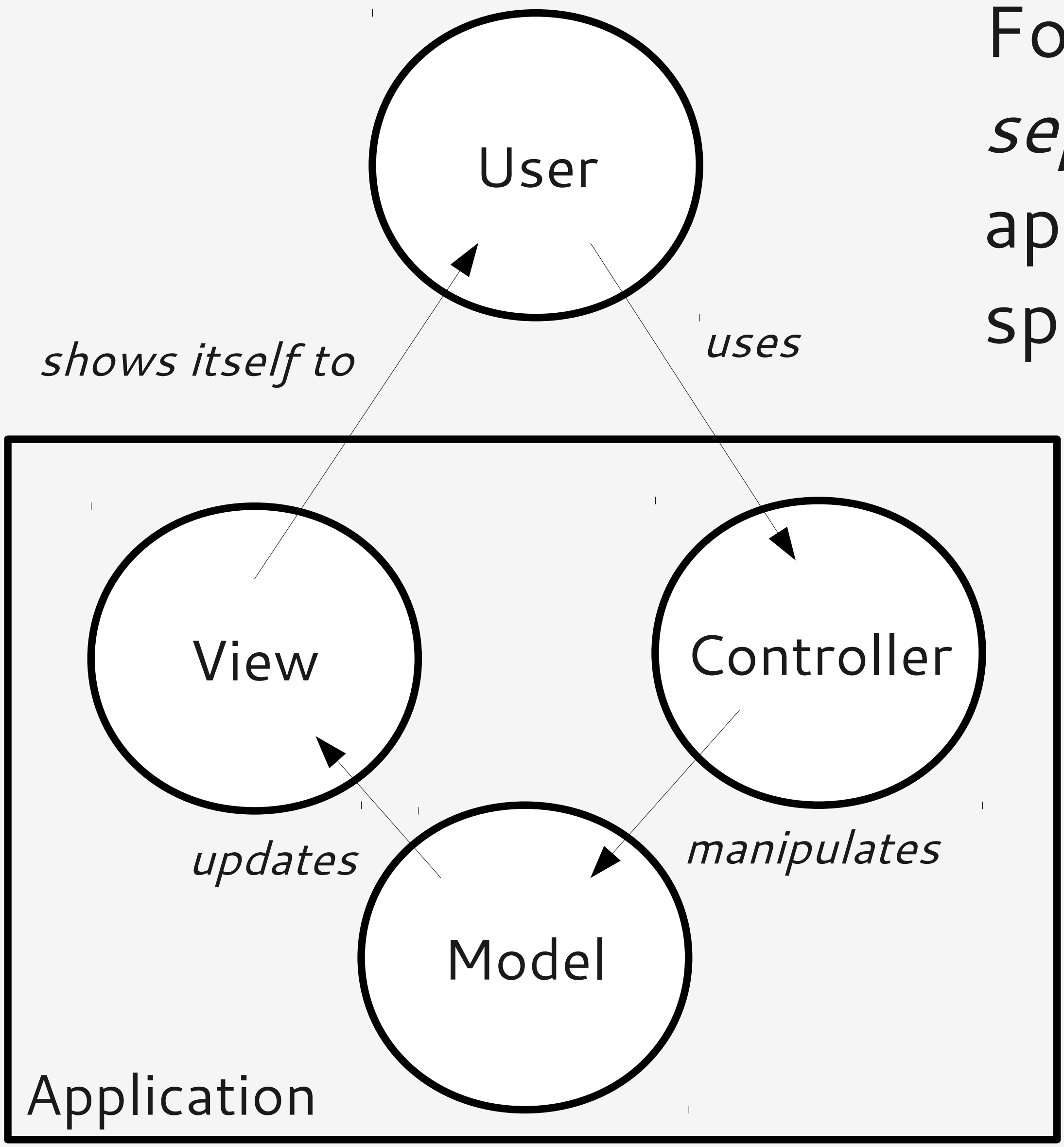
  → Different teams may work on different sub-problems.

Leibniz Center for Law

Decomposition (or factoring)

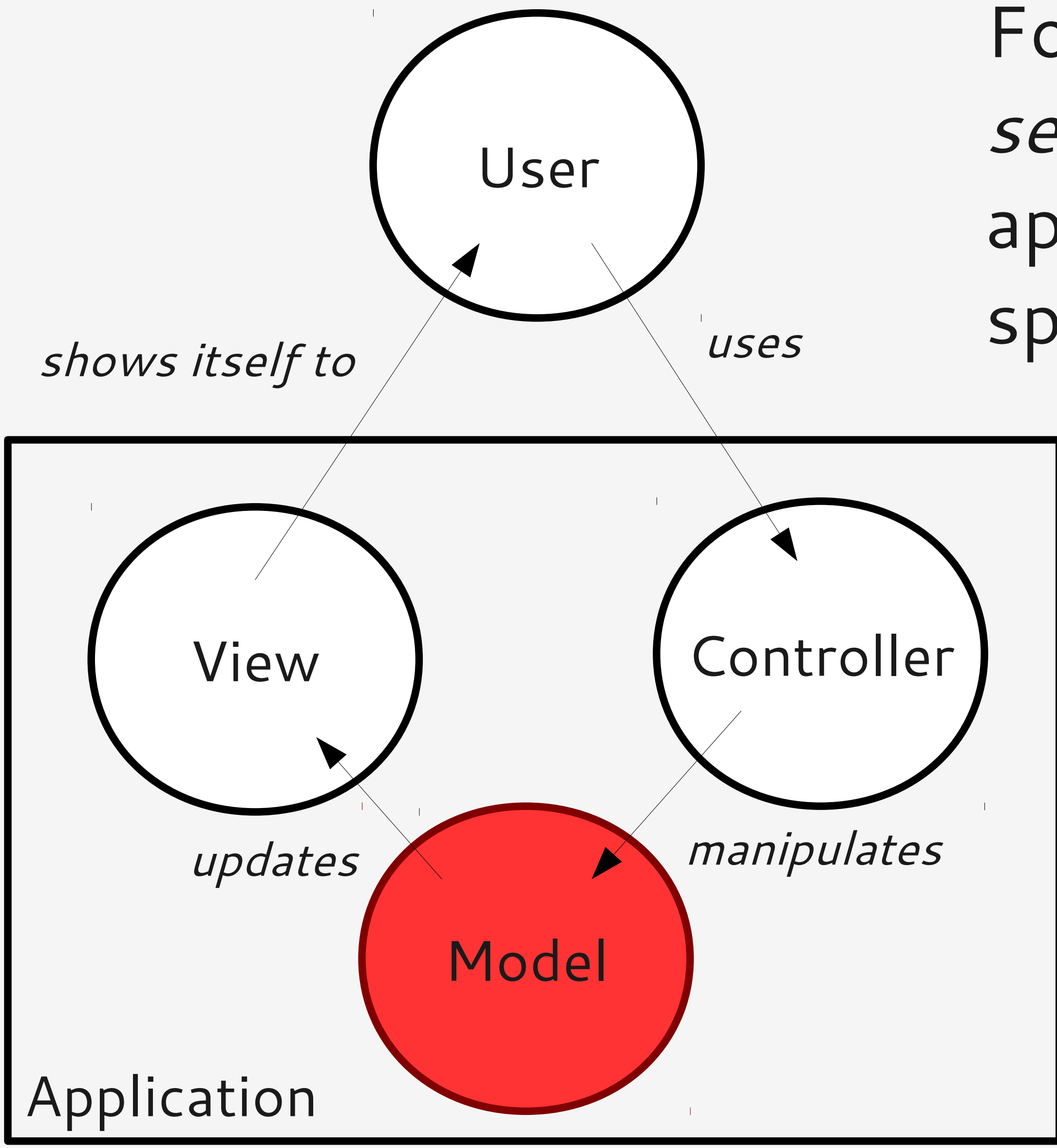
- Intuitively the breaking down should be made in order to:
  - minimize the static dependencies among the parts → **low coupling** *between modules*
  - maximise the cohesion (how much the elements belong together) within each part. → *modular* **high cohesion**

MVC design pattern



Following the principle of *separation of concerns*, an application can be specified distinguishing:

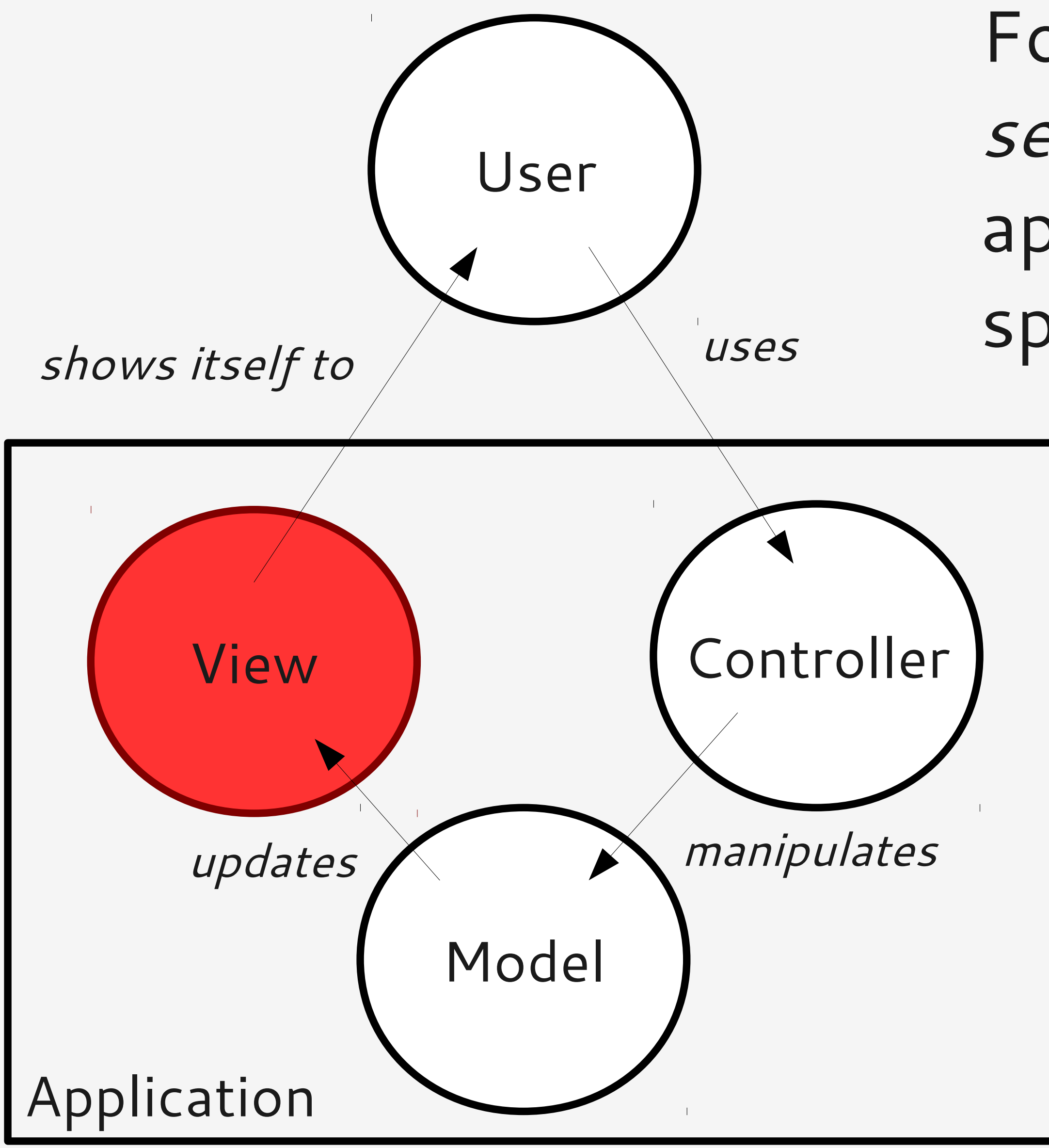
MVC design pattern



Following the principle of *separation of concerns*, an application can be specified distinguishing:

MODEL: the knowledge, i.e. data structures as e.g. objects, or more often structures of them (e.g. *databases*)

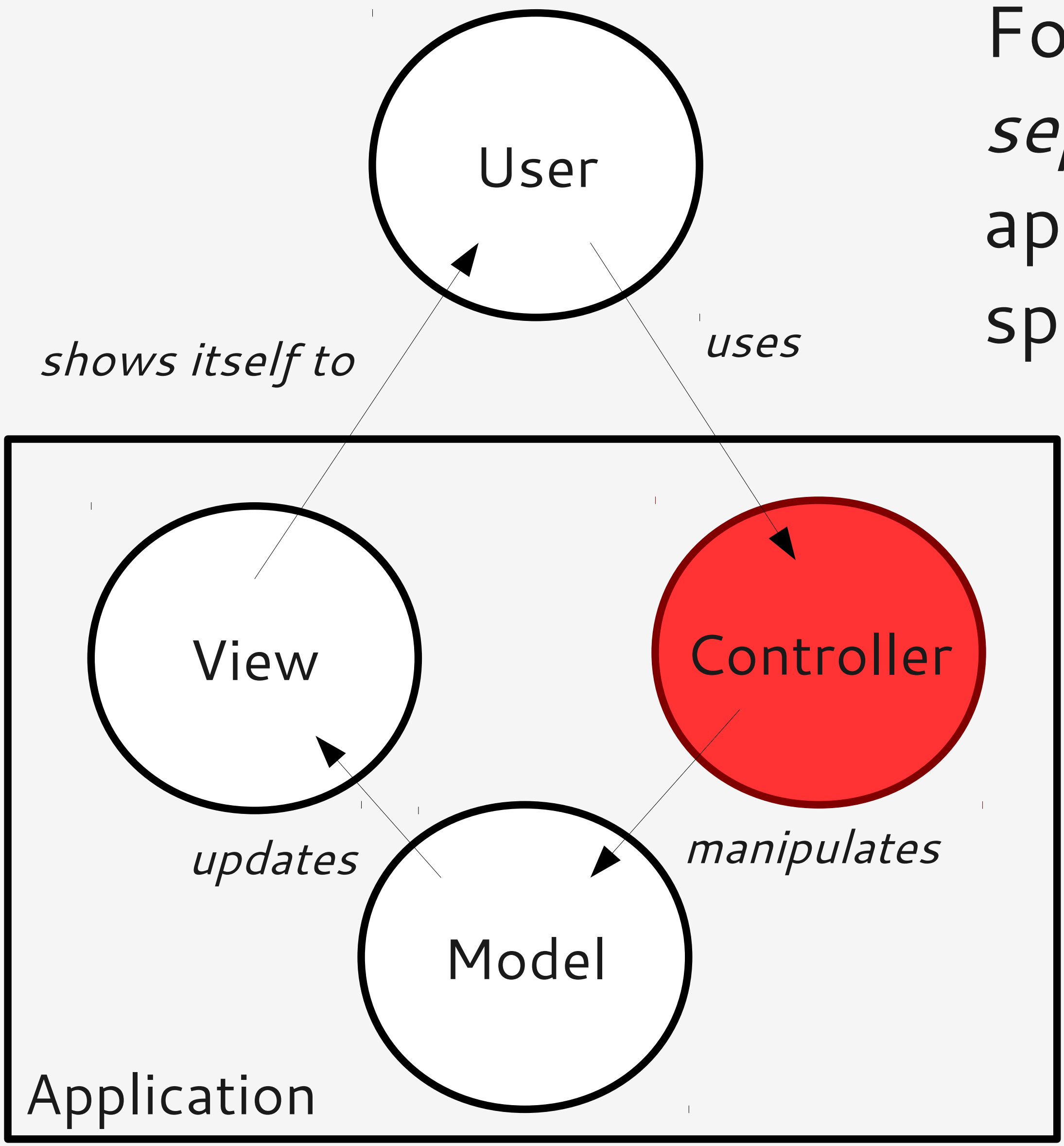
MVC design pattern



Following the principle of *separation of concerns*, an application can be specified distinguishing:

- **VIEW**: a visual representation of the model (there may be multiple views!)

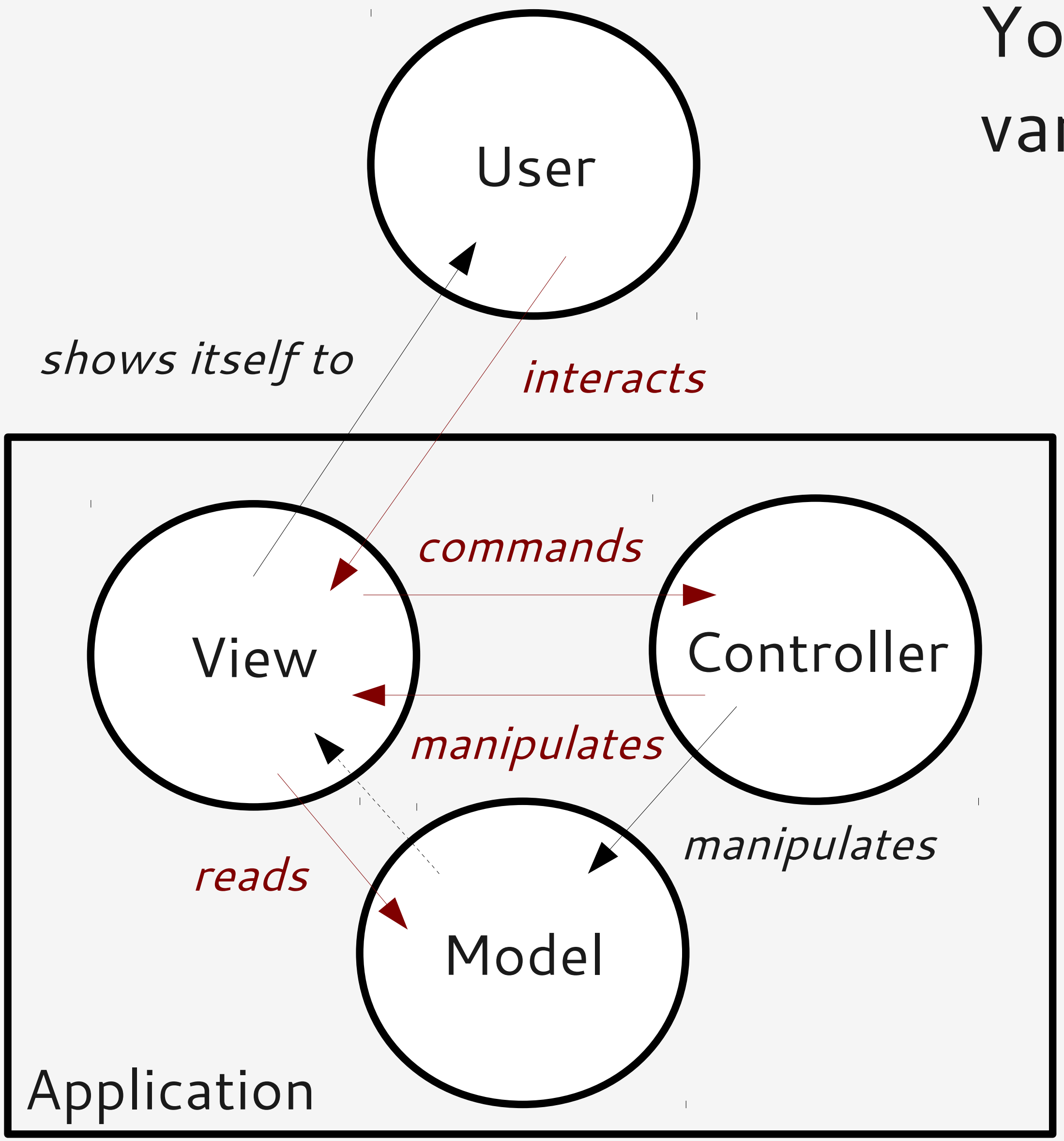
MVC design pattern



Following the principle of *separation of concerns*, an application can be specified distinguishing:

- **CONTROLLER**: the operational logic of the application, serves as a interface between the user and the model

MVC design pattern (variations)

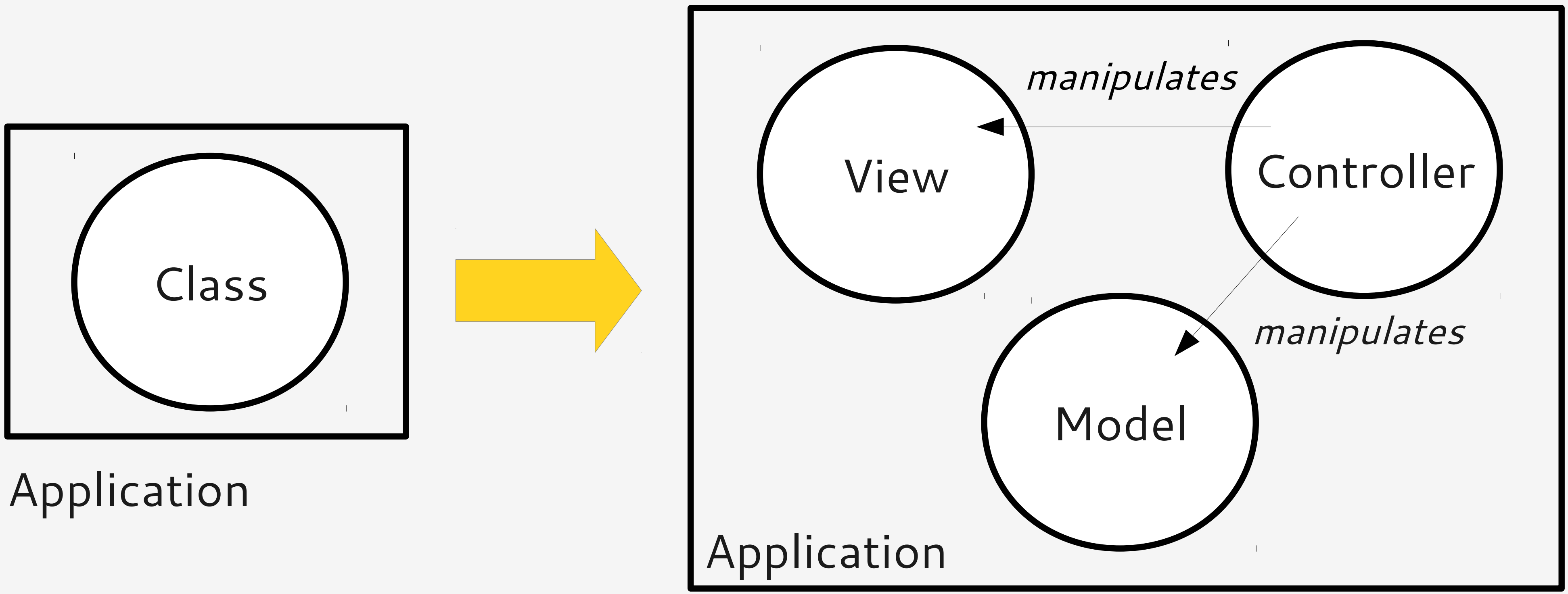


You can encounter some variations of the pattern:

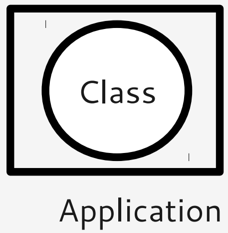
- The user interacts with the view to command the controller (e.g. buttons)
- The controller modifies the view
- The view actively reads the model

Exercise

Transform the given code following this pattern:



Exercise Code

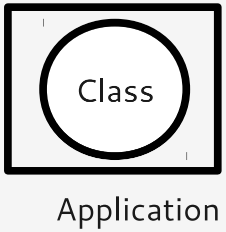



```
class Student {
   String number
   String name

   void show() {
      println("-- Student --")
      println("Name:  " + name)
      println("Number:  " + number)
   }

   void setName(String studentName){
      name = studentName
   }

   String getName() {
      return name
   }

   void setNumber(String studentNumber){
      number = studentNumber
   }

   String getNumber() {
      return number
   }
}
```

```
student = new Student()
student.setName("Jahn")
student.setNumber("143AB")

// print the student information
student.show()

// correct the name
student.setName("John")

// print the student information
student.show()
```

To download the code: http://justinian.leibnizcenter.org/noMVC.groovy
To run it: https://groovyconsole.appspot.com/

Exercise Code




```
class Student {
   String number
   String name

   void show() {
      println("-- Student --")
      println("Name:  " + name)
      println("Number:  " + number)
   }

   void setName(String studentName){
      name = studentName
   }

   String getName() {
      return name
   }

   void setNumber(String studentNumber){
      number = studentNumber
   }

   String getNumber() {
      return number
   }
}
```

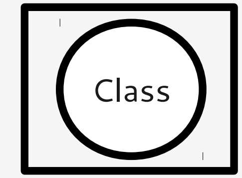
```
student = new Student()
student.setName("Jahn")
student.setNumber("143AB")

// print the student information
student.show()

// correct the name
student.setName("John")

// print the student information
student.show()
```

output

?

To download the code: http://justinian.leibnizcenter.org/noMVC.groovy
To run it: https://groovyconsole.appspot.com/

Exercise Code



Application

```
class Student {
   String number
   String name

   void show() {
      println("-- Student --")
      println("Name:  " + name)
      println("Number:  " + number)
   }

   void setName(String studentName){
      name = studentName
   }

   String getName() {
      return name
   }

   void setNumber(String studentNumber){
      number = studentNumber
   }

   String getNumber() {
      return number
   }
}
```

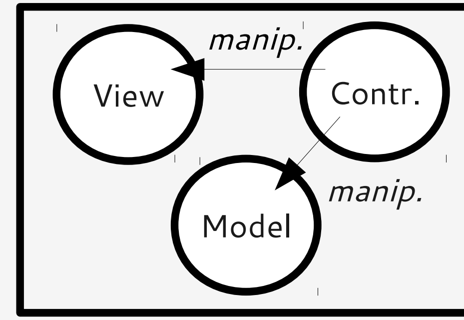
```
student = new Student()
student.setName("Jahn")
student.setNumber("143AB")

// print the student information
student.show()

// correct the name
student.setName("John")

// print the student information
student.show()
```

output

```
-- Student --
Name: Jahn
Number: 143AB
-- Student --
Name: John
Number: 143AB
```

To download the code: http://justinian.leibnizcenter.org/noMVC.groovy
To run it: https://groovyconsole.appspot.com/

Exercise Code



Application

```
class Student {
   String number
   String name

   void show() {
      println("-- Student --")
      println("Name: " + name)
      println("Number: " + number)
   }

   void setName(String studentName){
      name = studentName
   }

   String getName() {
      return name
   }

   void setNumber(String studentNumber){
      number = studentNumber
   }

   String getNumber() {
      return number
   }
}
```

```
student = new Student()
student.setName("Jahn")
student.setNumber("143AB")

// print the student information
student.show()

// correct the name
student.setName("John")

// print the student information
student.show()
```

?

```
student = new Student()
student.setName("Jahn")
student.setNumber("143AB")

view = new StudentView()
controller
     = new StudentController(model, view)

controller.updateView()
controller.setStudentName("John")
controller.updateView()
```

To download the code: http://justinian.leibnizcenter.org/noMVC.groovy
To run it: https://groovyconsole.appspot.com/

“Composition”

Top-down and bottom-up programming

- **Top-down** is a programming style in which design begins by specifying complex pieces and then dividing them into successively smaller pieces (generally associated to procedural languages).
- **Bottom-up** approach starts from detailing the smaller pieces and work by composing them to create the intended program (sometimes with Object-Oriented languages).

Bottom-up programming

- It is usual in LISP to start by defining new primitive operators. The resulting program is usually shorter.
- Sometimes in Object Oriented languages you start by defining the "smaller" element, i.e. the one with no dependencies, and then you build up the others.
  - e.g. rather then starting from the document and filling it with words, you start from the word, and define the document as filled with words.

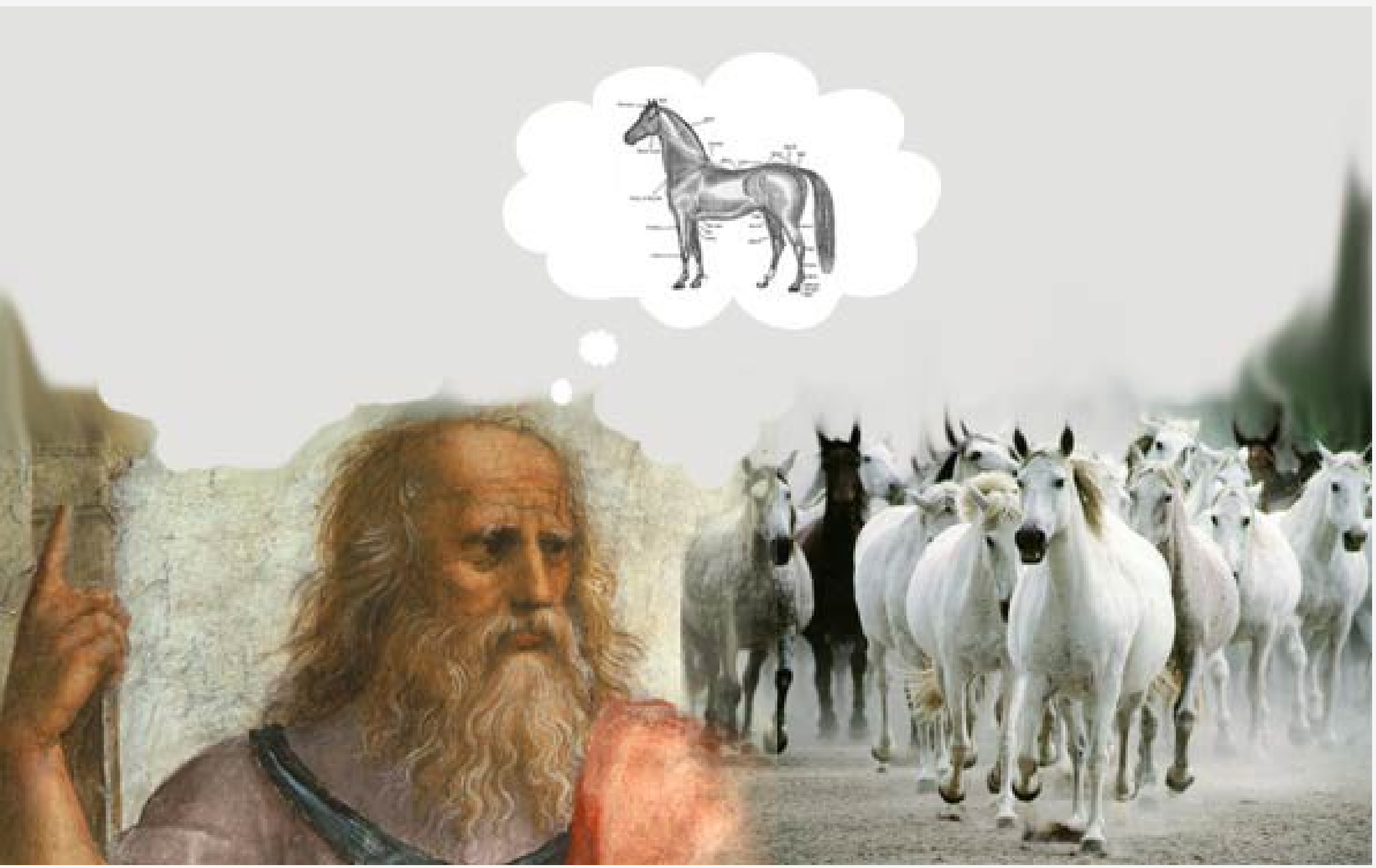
Bottom-up programming

- It is usual in LISP to start by defining new primitive operators. The resulting program is usually shorter.
- Sometimes in Object Oriented languages you start by defining the "smaller" element, i.e. the one with no dependencies, and then you build up the others.
  - e.g. rather then starting from the document and filling it with words, you start from the word, and define the document as filled with words.
- *Programmers usually work with both techniques.*

Hierarchy and individuals

Individuals (IS-INSTANCE-OF)

- The concept of class intuitively refers to some entity that belongs to that class.



- This entity or *object* is said to an **instance of** that class.

Individuals (IS-INSTANCE-OF)

```
class Person {
  String name

  void setName(String newName) {
    name = newName
  }
}

p = new Person()
p.setName("Plato")
```

Individuals (IS-INSTANCE-OF)

```
class Person {
  String name

  void setName(String newName) {
    name = newName
  }
}

p = new Person()
p.setName("Plato")
```

describe properties of the system

Individuals (IS-INSTANCE-OF)

```
class Person {
  String name

  void setName(String newName) {
    name = newName
  }
}

p = new Person()
p.setName("Plato")
```

actually allocate (memory) space for the object

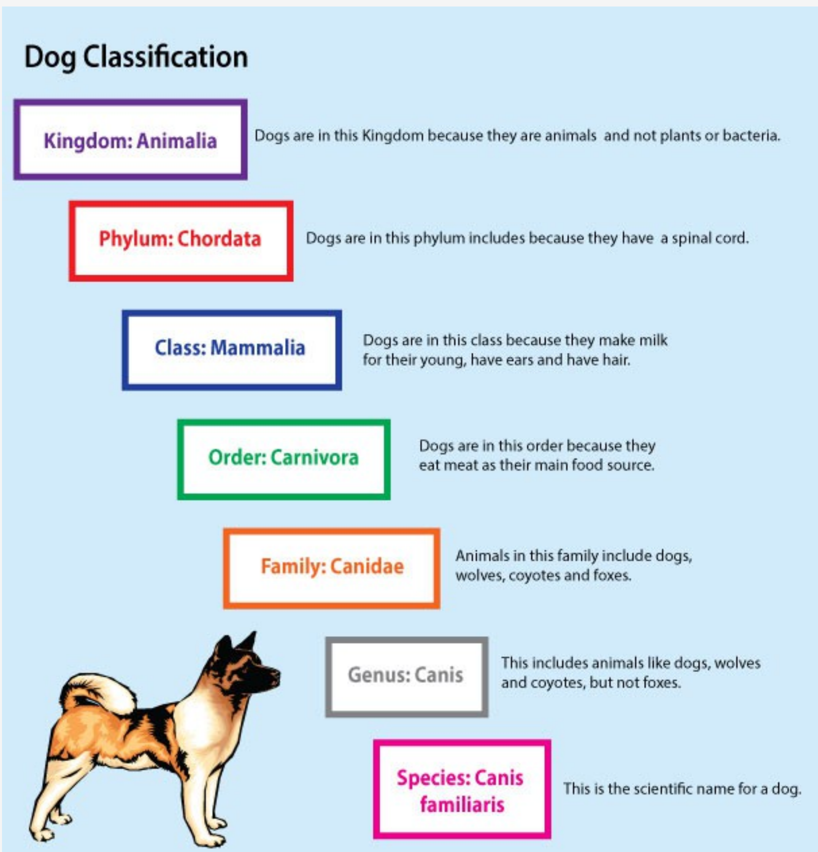
Individuals (IS-INSTANCE-OF)

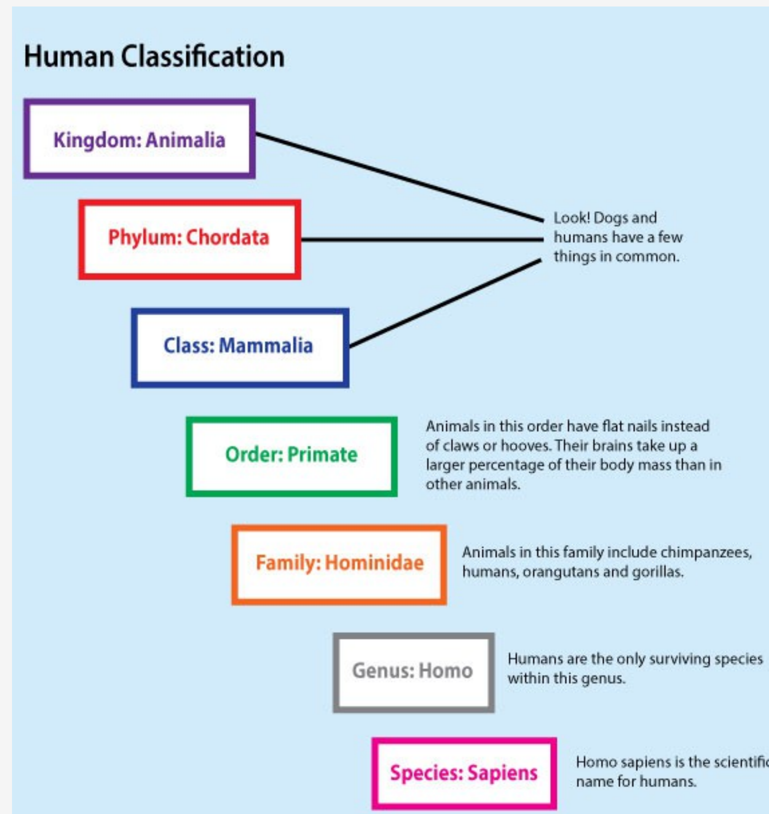
```
class Person {
  String name

  void setName(String newName) {
    name = newName
  }
}

p = new Person()
p.setName("Plato")
```

apply the required method to the object

Hierarchy as Taxonomy (IS-A)

- Things and concepts are usually hierarchically classified both in common and expert knowledge.





Leibniz Center for Law

Hierarchy as Taxonomy (IS-A)

- Things and concepts are usually hierarchically classified both in common and expert knowledge.
- Given a certan class, a subclass or **derived class** inherits certain properties (as attributes and methods) from the first.
- From the perspective of the second class, the first is called *superclass*.
- Some derivation may be *overridden.*

Hierarchy as Taxonomy (IS-A)

```
class A {
  String salutation = "Ciao"
  void show() {
    print(salutation + "! My type is A.")
  }
}
```

output

?

```
a = new A()
a.show()
```

Hierarchy as Taxonomy (IS-A)

```
class A {
  String salutation = "Ciao"
  void show() {
    print(salutation + "! My type is A.")
  }
}
```

output

Ciao! My type is A.

```
a = new A()
a.show()
```

Hierarchy as Taxonomy (IS-A)

```
class A {
  String salutation = "Ciao"
  void show() {
    print(salutation + "! My type is A")
  }
}

class B extends A {}
```

```
b = new B()
b.show()
```

output

?

Hierarchy as Taxonomy (IS-A)

```
class A {
  String salutation = "Ciao"
  void show() {
    print(salutation + "! My type is A")
  }
}

class B extends A {}
```

```
b = new B()
b.show()
```

output

Ciao! My type is A.

Hierarchy as Taxonomy (IS-A)

```
class A {
  String salutation = "Ciao"
  void show() {
    print(salutation + "! My type is A")
  }
}

class B extends A {
  @Override
  void show() {
    print(salutation + "! My type is B")
  }
}

b = new B()
b.show()
```

output

?

Hierarchy as Taxonomy (IS-A)

```
class A {
  String salutation = "Ciao"
  void show() {
    print(salutation + "! My type is A")
  }
}

class B extends A {
  @Override
  void show() {
    print(salutation + "! My type is B")
  }
}

b = new B()
b.show()
```

output

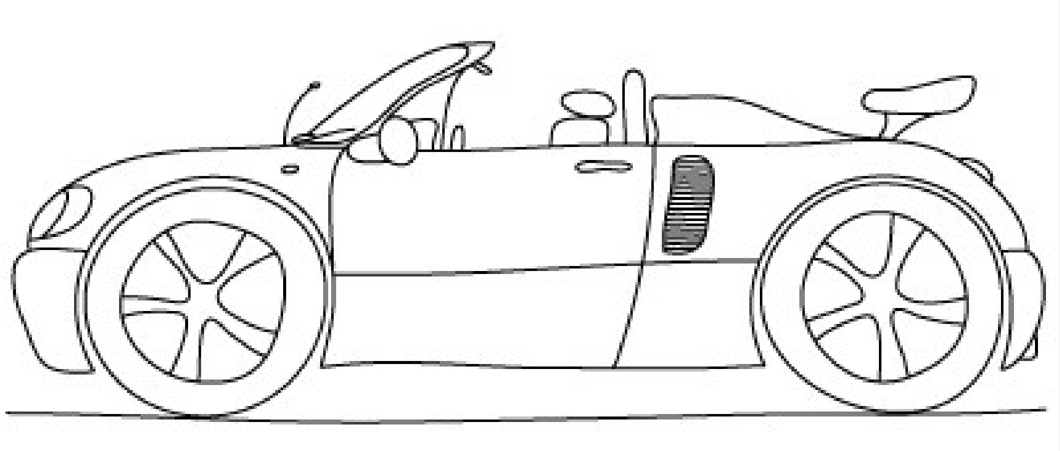
Ciao! My type is B.

Hierarchy as partonomy (HAS-A)

- Given an object of a certain class, if it is composed by other objects, the second ones *belong to* the first.

Hierarchy as partonomy (HAS-A)

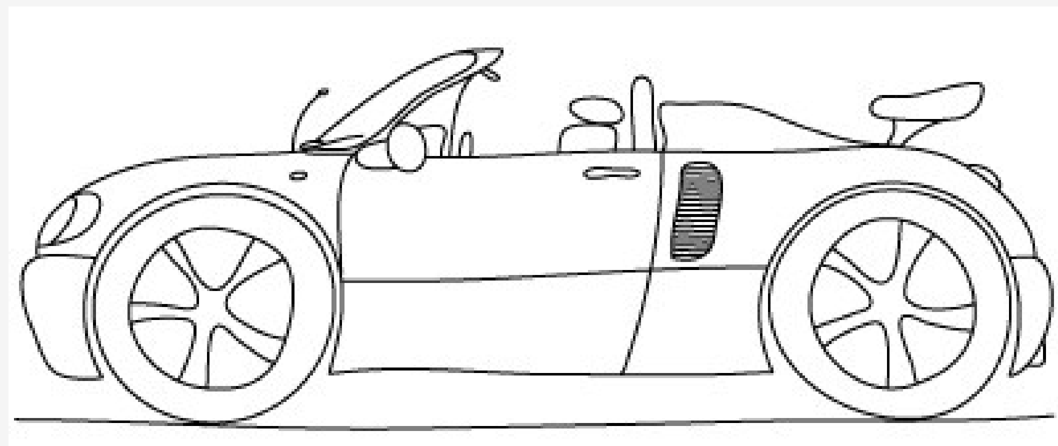
- Given an object of a certain class, if it is composed by other objects, the second ones *belong to* the first.



  - The car *has* four wheels.
  - Those wheels *belongs to* the car.

Hierarchy as partonomy (HAS-A)

- Given an object of a certain class, if it is composed by other objects, the second ones *belong to* the first.



  - The car *has* four wheels.
  - Those wheels *belongs to* the car.

(NB: things are a bit more complicated! *aggregation* vs *composition*)

“Strict” Composition

```
class Car {
  Wheel frontLeftWheel
  Wheel frontRightWheel
  Wheel rearLeftWheel
  Wheel rearRightWheel

  Car {
    frontLeftWheel = new Wheel()
    frontRightWheel = new Wheel()
    rearLeftWheel = new Wheel()
    rearRightWheel = new Wheel()
  }
}

car = new Car()
```

The lifetime of the components **depends on** the composed object.

Leibniz Center for Law

Aggregation

```
class Car {
  Wheel frontLeftWheel
  Wheel frontRightWheel
  Wheel rearLeftWheel
  Wheel rearRightWheel

  Car { }

  void mountWheels(fLW, fRW, rLW, rRW) {
    frontLeftWheel = fLW
    frontRightWheel = fRW
    rearLeftWheel = rLW
    rearLeftWheel = rRW
  }
}

car = new Car()
car.mountWheels(...)
```

The lifetime of the components **can differ** of that of the composed object.

Aggregation

```
class Car {
  Wheel frontLeftWheel
  Wheel frontRightWheel
  Wheel rearLeftWheel
  Wheel rearRightWheel

  Car { }

  void mountWheels(fLW, fRW, rLW, rRW) {
    frontLeftWheel = fLW
    frontRightWheel = fRW
    rearLeftWheel = rLW
    rearLeftWheel = rRW
  }
}

car = new Car()
car.mountWheels(...)
```

The lifetime of the components **can differ** of that of the composed object.

Composition is "strong" aggregation!

Exercise

- Represent
  - people (accounting their name)
  - and students (considering their student ID and their university)
  - the chairs that are in this room
  - people being able to sit on chairs which are in the same place
  - you as a student sitting in this room

Parallel algorithms

parallel algorithms/concurrent tasks

- Divide et impera often leads quite nicely to algorithms which may be computed in parallel, via concurrent processes.

(short) Exercise

- Consider the exercise of structured programming with cooking spaghetti. What we could have done concurrently?

(short) Exercise

- Consider the exercise of structured programming with cooking spaghetti. What we could have done concurrently?

parallel algorithms/concurrent tasks

- A synchronization is necessary when the concurrent components are:
  - communicating (**message-passing concurrency**), at the base for instance of the *Actor* model;
  - referring to the same resource (**shared-state concurrency**); faster, but it requires primitives (e.g. *semaphores*) to avoid *race* conditions
  - ...

Leibniz Center for Law

(short) Exercise

- How concurrency is handled
  - during auctions?
  - when airplanes are landing?
  - with trains?

(short) Exercise

- How concurrency is handled
  - during auctions?
  - when airplanes are landing?
  - with trains?

Summary of the 4 axis

Imperative vs Declarative

- Imperative:
  - programming focused on the **sequence of operations** necessary to solve the problem (which in turn usually stays implicit)
- Declarative
  - programming focused on describing the **problem** (while the sequence of operations to be performed is left implicit)

Procedural vs Object-oriented

- Procedural
  - programming focused on *procedures*: blocks of instructions/portions of code related to specific tasks
- Object-oriented
  - programming focused on the (data) objects which are manipulated during the computation

Sequential, Concurrent, Parallel

- Sequential
  - instructions are executed step by step
- Concurrent
  - execution occurs concurrently, and if it has side-effect over the same components (*race* condition), it produces **non-determinism**
- Parallel
  - determinism is guaranteed if concurrency occurs in separate components (e.g. multicore processors)

Static vs Dynamic

- Static
  - Properties (types of variables, definitions of types, code) are fixed when the program is compiled
- Dynamic
  - The same properties can be changed at *run-time*

Some additional links

- *Understanding Association, Aggregation and Composition*

  http://www.codeproject.com/Articles/330447/Understanding-Association-Aggregation-and-Composit

For your moments of pause:

- *brief-incomplete-and-mostly-wrong story of programming*

  http://james-iry.blogspot.nl/2009/05/brief-incomplete-and-mostly-wrong.html

- *If programming languages were <T>*

  http://lambda-the-ultimate.org/node/3133

Leibniz Center for Law