Classes
The Static Structure

Abstract data types equipped with a possibly partial implementation

Style Rules

- Pick a style and stick to it

- Recommend that you use Eiffel style or close approximation
  - » **Look at example program text the case and pattern studies**

Style Rules – 2

- My style is to indent less
  - Rationale: screen real-estate is expensive, so make the best use of it you can
    - Only need two spaces not four
    - No need to indent text between initial class statement and final end statement
      - Does not give any additional structural information

Definitions

- A **class** is a combination of a **type** and a **module**

- A *module* because it is has a data part and an operation part

- A *type* because you can declare (and therefore create) instances of a class

Definitions – 2

- An **object** (a variable) is an instance of a class
  - > **Logically, each object has its own copy of the local attributes and its own copy of the operations in the class**

- A **client** class **C** of a **suppler** class **S** uses **S** by declaring a variable of type **S**.
  - » **S is a supplier of C**
  - » **C is a client of S**

Interface – Stack example

```
class STACK [ G ]

feature                    -- Enquiry and change

    full, empty : BOOLEAN          -- functions or attributes ?

    push ( x : G )                 -- a procedure

    pop                            -- a procedure

    top : G                        -- function or attribute ?

end
```

- No Specification of how a stack is implemented
- No implementation of features

Uniform Access Principle

All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

Uniform Access Principle – 2

- Client should neither know, nor care, if a returned value of a feature without a parameter is …
  - » **Stored as an attribute**
  - » **Computed as function**

```
class STACK [ G ]
feature                        -- Enquiry and change
    full, empty : BOOLEAN          -- functions or attributes ?
    push ( x : G )                 -- a procedure
    pop                            -- a procedure
    top : G                        -- function or attribute ?
end
```

© Gunnar Gotshalks

Person Class – 1

```
note                    -- For class level documentation
  description: "A simple person"
  author: "Gunnar Gotshalks"
  date: 2012 Jan 4"

class PERSON
create make             -- list construction features

feature
     name : STRING
     sex : GENDER
     age : INTEGER
```

↔ PERSON is a client of the suppliers
- STRING
- GENDER
- INTEGER

Person Class – 2

```
make( n : STRING ; s : GENDER ; a : INTEGER )
  -- Create a complete non default person
do
    -- Empty body for this example creation procedure
end

set_name ( s : STRING )
        -- Need to explicitly set attribute values
do
name := s
end
```

Person Class – 3

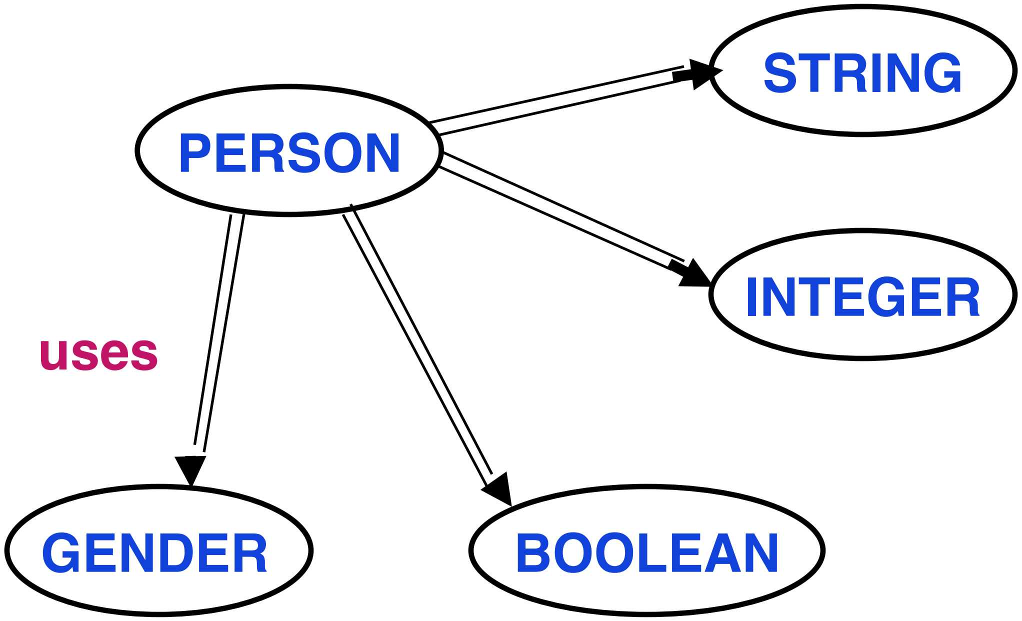
PERSON is a client of the supplier INTEGER

```
      older ( a : INTEGER ) : BOOLEAN
        -- Are you older than me?
      do
          if a > age then
              io_put_string ( "You are older than me. %N")
              Result := true
          else
              io_put_string ( "I am older than you. %N")
              Result := false
          end
      end
end
```

Client–Supplier BON diagram

- BON stands for

B-usiness O-bject N-otation



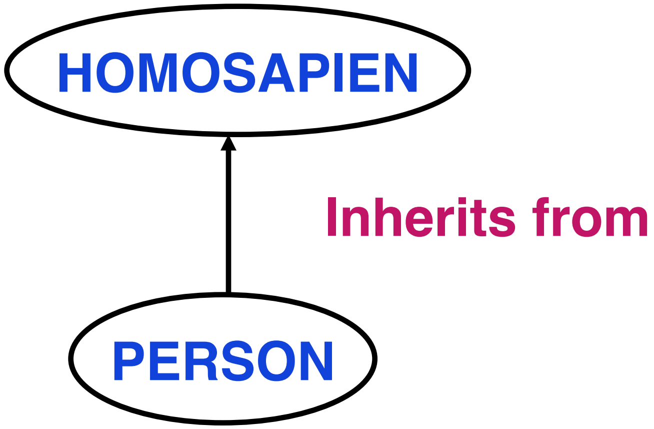
© Gunnar Gotshalks

Inheritance

Eiffel text

```
class PERSON inherit
   HOMOSAPIEN
feature
....
end
```

BON diagram



HOMOSAPIEN

Inherits from

PERSON

Feature Call

object . function (arguments)

- Evaluate the arguments to the **function**
- Then apply the function to the **object**
- In non OO languages this is equivalent to

function (object , arguments)

  - **where object = Current = self = this**

Infix Feature Call

- Can define operators that have one argument to be infix to use standard notation

  » **Thus**

  ```
  infix '<' ( other : TROLL) : BOOLEAN
            -- Compare me to another Troll
      do ... End
  ```

  » **Or used as**

  ```
  troll_1 <  troll_2
  ```

Prefix Feature Call

- Can define operators that have no arguments to be prefix to use standard notation
  - » Thus

    ```
    prefix '–' : INTEGER
            -- Unary minus
        do ... End
    ```

  - » Or used as

    ```
    – an_integer
    ```

Current Instance

- Instance calling the feature is locally named **Current**

```
p1 . distance_to ( p2 )     -- example call

                                                    bound to p2
     distance_to( p : POINT ) : REAL
       -- Distance between Current point and p

     do                                         bound to p1
        if ( p /= Current ) then
           Result := sqrt( ( x - p.x )^2 + ( y - p.y )^2 )
        end
     End
```

» could write as follows but that is considered poor style

```
     Result := sqrt( ( current.x - p.x )^2
                   + ( current.y - p.y )^2 )
```

Current Instance – 2

- Partly like
  - » **self – in Smalltalk and Objective-C**
    this – in C++ and Java
- But uniform access principle has attributes as parameter-less functions
  - » **Thus the following is illegal as Current.x could be a function call**
    - > **You cannot assign a value to a function**

```
x : INTEGER
 t ( y : INTEGER )
do
  Current.x := y
end
```

Current Instance – 3

- **Current** can be used in the following contexts
  - **Passing instance as a parameter**

    a.f (Current)

  - **Comparing with another reference**

    x = Current

  - **Use as an anchor in anchored declarations**

    object : like Current

    - **Will see this again in inheritance**

Unique names features & parameters

- The following is **illegal**

```
a_var : INTEGER
…
a_procedure ( a_var : INTEGER )
do
  io.put_string ( a_var )
end
```

Single Name Rule

Two different items within a class may not have the same final name

a_var cannot be both a feature and a parameter of a feature

Selective Exports

- Need to restrict access by clients
- In Java have public, protected and private
- In Eiffel can be more selective

```
class S feature
    -- all features exported -- public

feature { A , B }
    -- export only to A and B -- protected

feature { NONE }
    -- export to no one -- private, secret
    -- NOT EVEN TO S – include self if needed !

end
```

System Execution

- Create a certain object
  - » **called the root object for the execution**

- Apply a certain procedure to that object
  - » **called the creation procedure**

    This is the BIG BANG!

- Not the same as a system top
  - » **NOT the top of the architecture**
  - » **Just the start of execution**

Class Definition

Class A class is an abstract data type equipped with a possibly partial implementation.

Deferred / Effective Class

A class which is fully implemented is said to be effective.
A class which is implemented partially, or not at all, is said to be deferred. Any class is either deferred or effective.

In Java a deferred class is called an abstract class
In Java an interface is a class with all methods deferred and no objects

Role of Deferred Classes

- Design and analysis
- Pure description
  - **No implementation details required**
- Concentrate on architectural properties
- Provide for variations in implementation while preserving a particular type
- Provide for evolutionary development and its history

OO Software Construction (OOSC)

Object oriented software construction technical definition

The building of software systems as structured collections of possibly partial abstract data type implementations

OOSC and ADTs

- Basis is ADT

- Need ADT implementations

- Can have partial implementations

OOSC and ADTs – 2

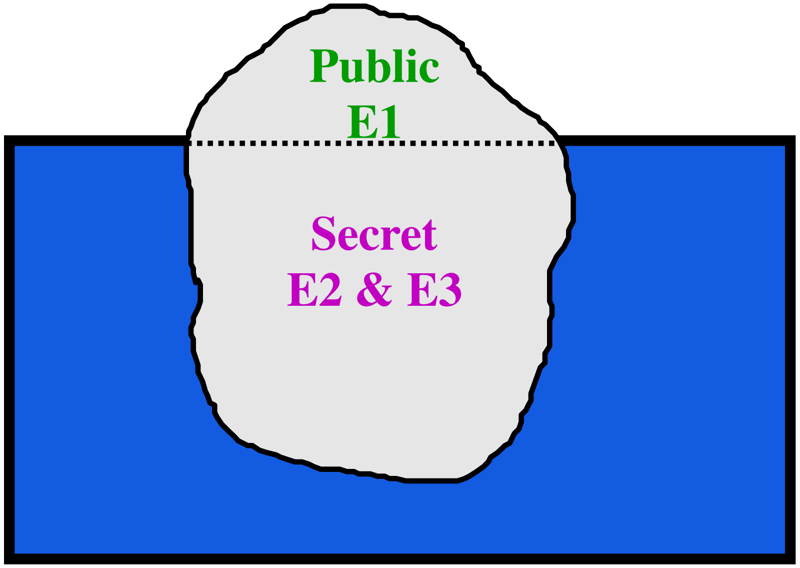
- System is a collection of classes
  - » No one class particularly in charge
    - > No top or main program
    - > Although execution requires a starting location
      - – Could change
        - In principle and practice

OOSC and ADTs – 2

- The collection is structured by two inter-class relations
  - » **Client – supplier**
    - > **Client has_a supplier**
  - » **Inheritance**
    - > **Class is_a …**

ADT to Class

- Basic steps in getting a class from an abstract data type
  - » **E1 – Create an ADT**
  - » **E2 – Chose a representation**
  - » **E3 – Create a mapping of the operations in E1 to the representation in E2**



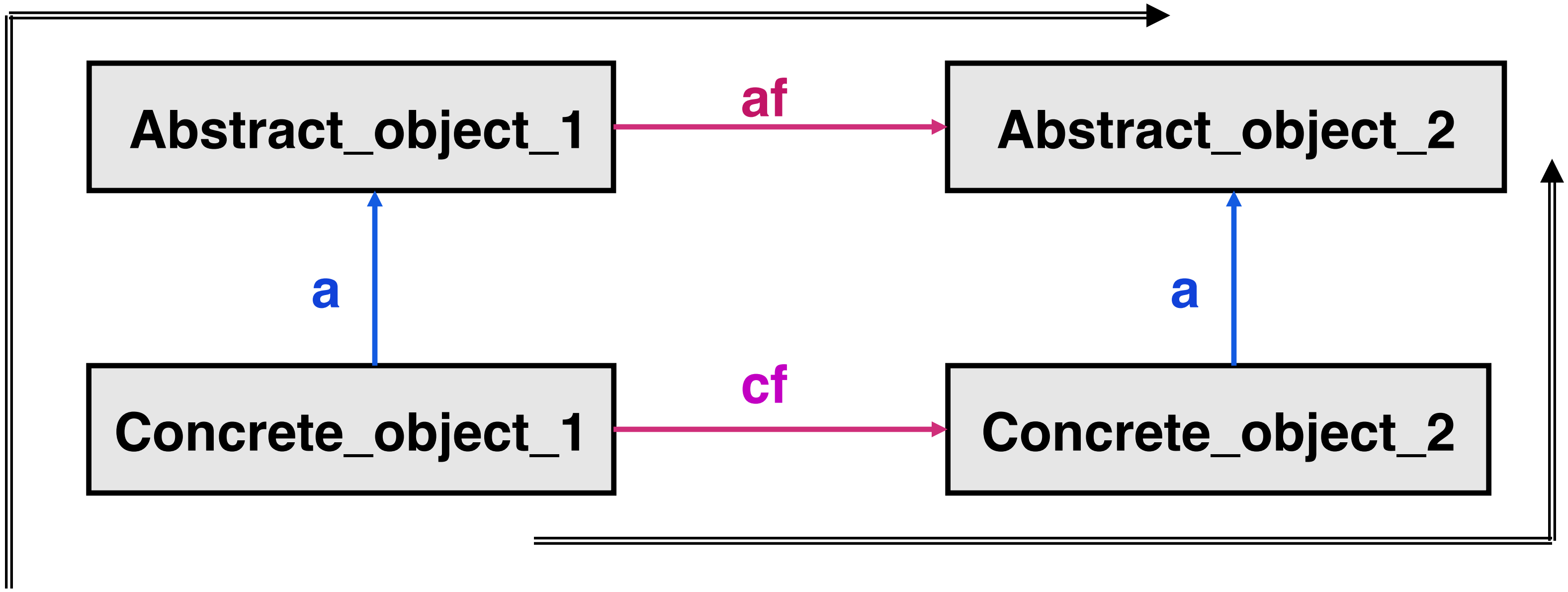
© Gunnar Gotshalks

Class–ADT Relationship

$$\begin{tikzcd}
\text{Abstract\_object\_1} \arrow[r, "af"] & \text{Abstract\_object\_2} \\
\text{Concrete\_object\_1} \arrow[u, "a"] \arrow[r, "cf"] & \text{Concrete\_object\_2} \arrow[u, "a"]
\end{tikzcd}$$

- **a – maps a concrete object into an abstract object**
- **af – function that maps abstract object 1 into abstract object 2**
- **cf – function that maps concrete object 1 into concrete object 2**

Class–ADT Consistency Property



$$a \; ; \; af \quad \equiv \quad cf \; ; \; a$$