

Classes

The Static Structure

**Abstract data types equipped with
a possibly partial implementation**

Style Rules

- Read page 180
- Pick a style and stick to it
- Recommend that you use Eiffel style or close approximation

Definitions

- A **class** is a combination of a **type** and a **module**
- A *module* because it has a data part and an operation part
- A *type* because you can declare (and therefore create) instances of a class
- 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 **supplier** class **S** uses **S** by declaring a variable of type **S**.
 - » **S is a supplier of C – C is a client of S**

Stack – Interface

class STACK

feature

-- Enquiry and change

full, empty : BOOLEAN -- functions or attributes ?

push (x : G) -- a procedure

pop -- a procedure

top : G -- function or attribute ?

end -- STACK[G}

- No Specification of how a stack is implemented
- No implementation of features
- Uniform access principle
 - » **client does not know, nor care, if a returned value is stored (an attribute) or computed (a function)**

Person Class – 1

indexing -- For class level documentation

description: "A simple person"

author: "Gunnar Gotshalks"

date:"2000 Jan 9"

class PERSON

create -- list construction features

make

feature

name : STRING

sex : GENDER

age : INTEGER

-- attributes are usually first

**PERSON is a client of
the supplier STRING**

Person Class – 2

```
make( n : STRING ; s : GENDER ; a : INTEGER ) is
```

```
-- Create a complete non default person
```

```
do
```

```
-- Empty body for this example creation procedure
```

```
end
```

```
set_name ( s : STRING ) is
```

```
-- 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 is

-- 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

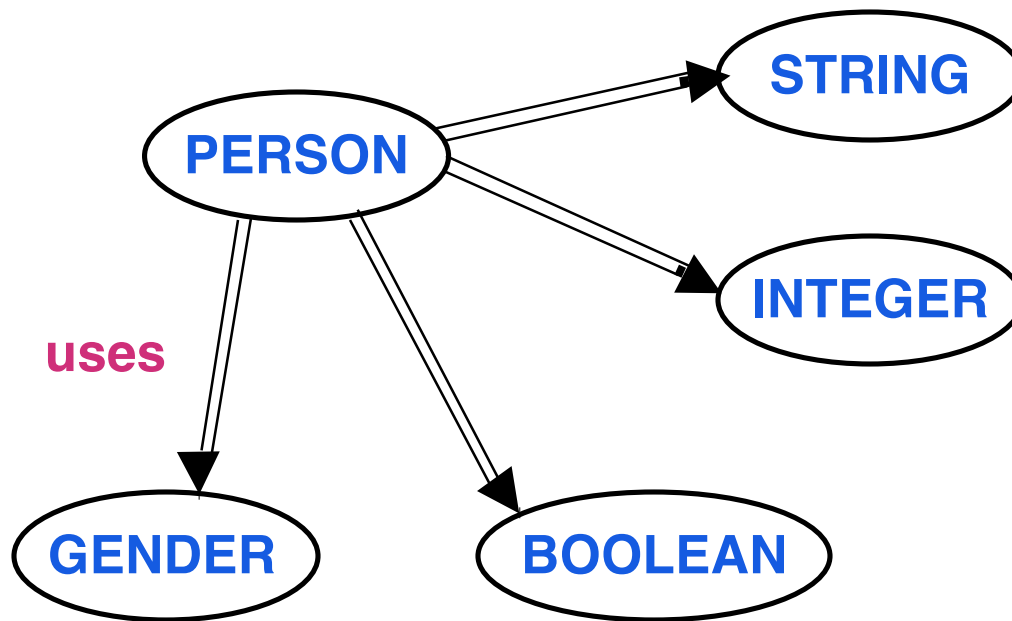
end

end -- PERSON

Client–Supplier BON diagram

- BON stands for

B-business O-bject N-otation

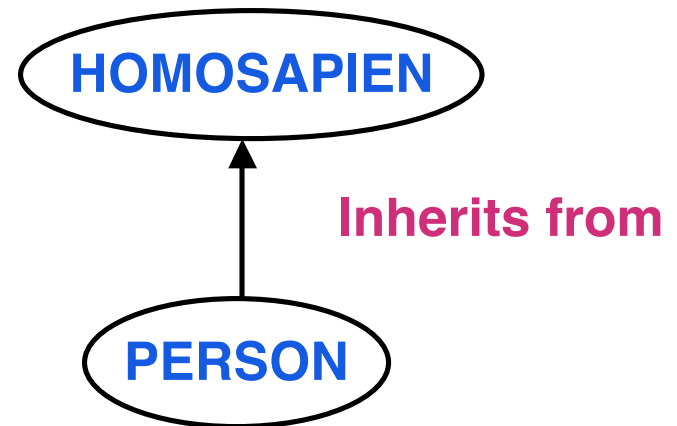


Inheritance

Eiffel text

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

BON diagram



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 to be infix to use standard notation
 - » **Thus**
`plus (other : INTEGER) : INTEGER is do ... end`
 - » **Is used as**
`anInt . plus (otherInt)`
- Eiffel has the infix keyword
 - » **Thus**
`infix "+" (other : INTEGER) : INTEGER is do ... end`
 - » **is used as**
`anInt + otherInt`
- Also have **prefix** for unary operators

Current Instance

- Instance calling the feature is named Current locally

p1 . distance_to (p2) -- example call

distance_to(p : POINT) : REAL is

-- Distance between Current point and p

do

if (p /= Current) then

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

end

End

bound to p2

bound to p1

>> could write as

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

Current Instance – 2

- Partly like

self – in Smalltalk

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 – one cannot assign a value to a function**

```
x : INTEGER
t ( y : INTEGER ) is
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 ) is
```

```
do
```

```
  io.put_string(a_var )
```

```
end
```

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 -- class S

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

Object oriented software construction technical definition

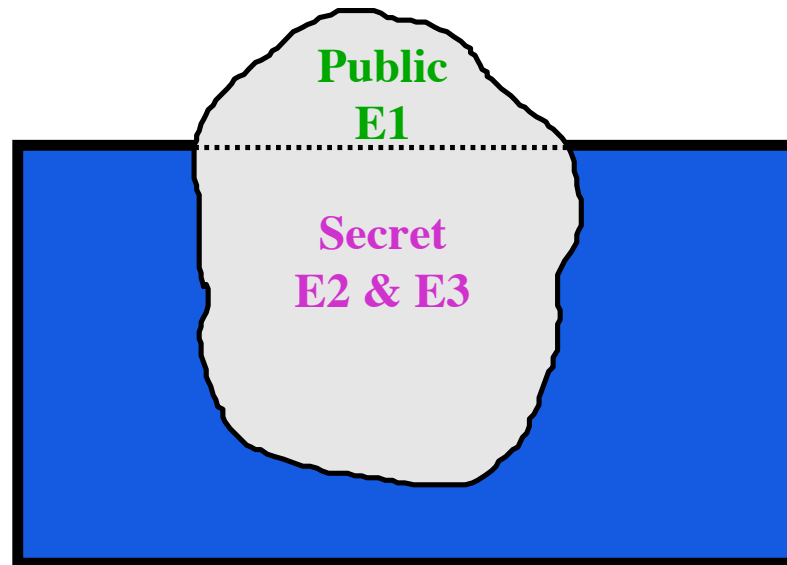
**The building of software systems as structured collections
of possibly partial abstract data type implementations**

OO Software Construction– 2

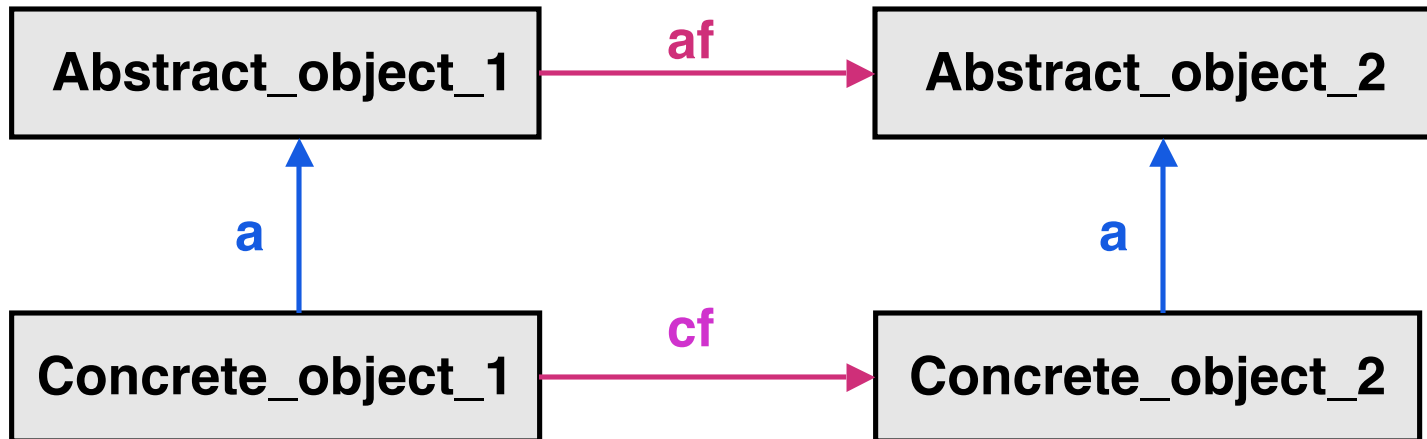
- Basis is ADT
- Need ADT implementations
- Can have partial implementations
- System is a collection of classes
 - » **with no one class particularly in charge – no top or main program**
 - > **Although an execution requires a starting location**
 - > **In principle could change**
- The collection is structured by two inter-class relations
 - » **client – user**
 - » **inheritance.**

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**



Class-ADT Relationship



- » **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

