Object-Oriented Software Construction

Bertrand Meyer

Lecture 21: Agents and tuples

Agents: the basic idea

Encapsulating routines in objects

agent *r*
agent *r* (*x*, ?, *y*)

Mechanism will first be illustrated through event-driven programming

Handling traditional input

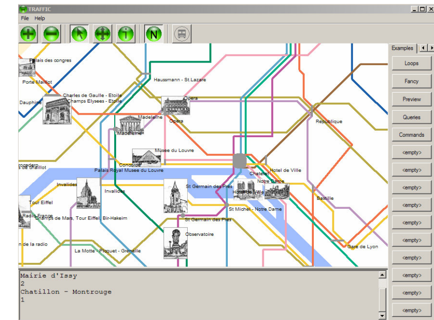
Program drives input:

```
from
    read_next_character
until  last_character = Enter loop
    i := i + 1
    Result.put (last_character, i)
    read_next_character
    end
```

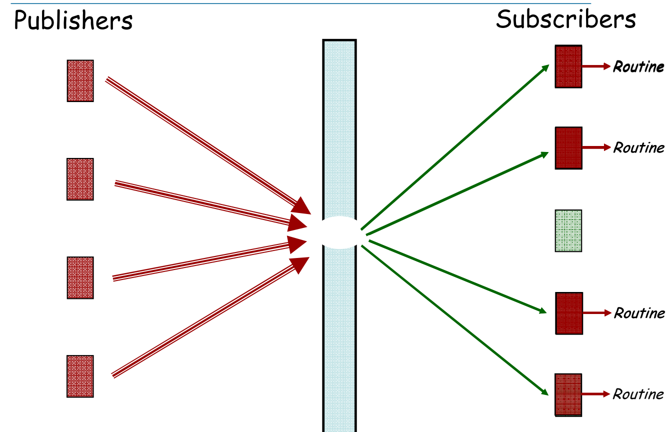

Handling input with modern GUIs

User drives program:

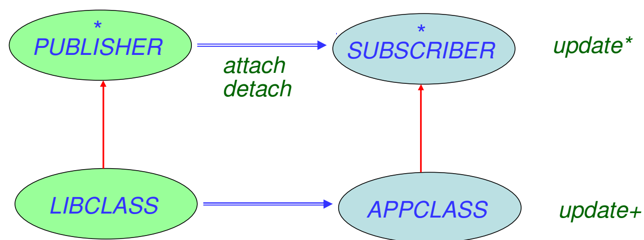
"When a user presses this button, execute that action from my program"



Event-driven programming



A solution: Observer Pattern



* Deferred (abstract)
\+ Effective (implemented)

Inherits from
Client (uses)

Observer pattern

Publisher keeps a list of observers:

subscribed: *LINKED_LIST* [*OBSERVER*]

To register itself, an observer may execute

subscribe (*some_publisher*)

where *subscribe* is defined in *OBSERVER*:

```
subscribe (p: PUBLISHER) is
        -- Make current object observe p.
    require
        publisher_exists: p /= Void
    do
        p.attach (Current)
    end
```


Attaching an observer

In class *PUBLISHER*:

```
attach (s: SUBSCRIBER) is
        -- Register s as subscriber to current publisher.
    require
        subscriber_exists: p /= Void
    do
        subscribed.extend (s)
    end
```

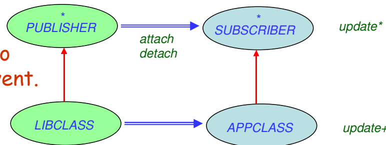
Note that invariant of *PUBLISHER* includes the clause

subscribed /= Void

(List *subscribed* is created by creation procedures of *PUBLISHER*)

Triggering an event

```
trigger is
        -- Ask all observers to
        -- react to current event.
    do
        from
            subscribed.start
        until
            subscribed.after
        loop
            subscribed.item. update
            subscribed.forth
        end
    end
```



Each descendant of *OBSERVER* defines its own version of *update*

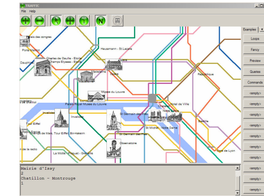
Observer pattern

- Publishers know about subscribers
- Subscriber may subscribe to at most one publisher
- May subscribe at most one operation
- Not reusable — must be coded anew for each application

Another approach: action-event table

Set of triples

[Event, Context, Action]



Event: any occurrence we track
Example: a mouse click

Context: object for which the event is interesting
Example: a particular button

Action: what we want to do when the event occurs in the context
Example: save the file

Action-event table may be implemented as e.g. a hash table.

The EiffelVision style

```
my_button.click.action_list.extend (agent my_procedure)
```

Mechanisms in other languages

C and C++: "function pointers"

C#: delegates (more limited form of agents)

With .NET delegates: publisher (2)

P4. Write new procedure OnClick to wrap handling:

```
protected void OnClick (int x, int y)
    {if (Click != null) {Click (this, x, y)}}
```

P5. For every event occurrence, create new object (instance of ClickArgs), passing arguments to constructor:

```
ClickArgs myClickargs = new Clickargs (h, v)
```

P6. For every event occurrence, trigger event:

```
OnClick (myclickargs)
```


With .NET delegates: publisher (1)

P1. Introduce new class ClickArgs inheriting from EventArgs, repeating arguments types of myProcedure:

```
public class Clickargs {... int x, y; …}
```

P2. Introduce new type ClickDelegate (delegate type) based on that class

```
public void delegate ClickDelegate  (Object sender,  e)
```

P3. Declare new type Click (event type) based on the type ClickDelegate:

```
public event ClickDelegate Click
```


With .NET delegates: subscriber

D1. Declare a delegate myDelegate of type ClickDelegate. (Usually combined with following step.)

D2. Instantiate it with myProcedure as argument:

```
ClickDelegate = new ClickDelegate (myProcedure)
```

D3. Add it to the delegate list for the event:

```
YES_button.Click  += myDelegate
```

Using the Eiffel approach

Event: each event *type* will be an object
Example: mouse clicks

Context: an object, usually representing element of user interface
Example: a particular button

Action: an agent representing a routine
Example: routine to save the file

The EiffelVision style

YES_button.click.action_list.extend (**agent** *my_procedure*)

Event Library style

The basic class is *EVENT_TYPE*

On the publisher side, e.g. GUI library:

- (Once) declare event type:

  click: EVENT_TYPE [*TUPLE* [*INTEGER, INTEGER*]]

- (Once) create event type object:

  create *click*

- To trigger one occurrence of the event:

  click.publish ([*x_coordinate, y_coordinate*])

On the subscriber side, e.g. an application:

click.subscribe (**agent** *my_procedure*)

Subscriber variants

click.subscribe (**agent** *my_procedure)*

my_button. click.subscribe (**agent** *my_procedure)*

click.subscribe (**agent** *your_procedure* (a, ?, ?, b))

click.subscribe (**agent** other_object.other_procedure)

Another example of using agents

$$\int_a^b my_function(x)\, dx$$

$$\int_a^b your_function(x, u, v)\, dx$$

my_integrator.*integral* (**agent** *my_function* , *a*, *b*)

my_integrator.*integral* (**agent** *your_function* (? , *u*, *v*), *a*, *b*)

Applications of agents

- Undo-redo
- Iteration
- High-level contracts
- Numerical programming
- Introspection (finding out properties of the program itself)

Using an iterator

all_positive := *my_integer_list*.*for_all*
(**agent** *is_positive* (?))

Iterators

In class *LINEAR* [*G*], ancestor to all classes for lists, sequences etc., you will find:

for_all
there_exists
do_all
do_if
do_while
do_until

Calling the associated routine

Given an agent, you may call the associated routine through the feature "*call*" :

a.call ([*horizontal_position, vertical_position*])

A tuple

If *a* is associated with a function, *a.item* ([..., ...]) gives the result of applying the function.

Behind agents: Tuples

Tuple types (for any types *A, B, C*, ...):

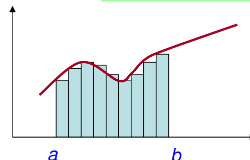
TUPLE
TUPLE [*A*]
TUPLE [*A, B*]
TUPLE [*A, B, C*]
...

A tuple of type *TUPLE* [*A, B, C*] is a sequence of at least three values, first of type *A*, second of type *B*, third of type *C*

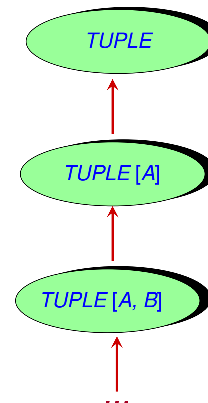
Tuple values: e.g. [*a1, b1, c1*]

The integration function

```
integral (f : FUNCTION [ANY, TUPLE [REAL], REAL];
        low, high: REAL): REAL is
            -- Integral of f over the interval [low, high]
    local
            x: REAL; i: INTEGER
    do
        from x := low until x > high loop
            Result := Result + step * f.item ([x])
            i := i + 1
            x := a + i * step
        end
    end
```



Tuple type inheritance



Accessing and modifying tuple elements

To obtain *i*-th element of a tuple *t*, use

t.item (*i*)

May need assignment attempt:

x ?= *t.item* (*i*)

To change *i*-th element, use *t.put* (*x*, *i*)

Agents and their arguments

An agent can have both "closed" and "open" arguments

Closed arguments set at time of agent definition; open arguments set at time of each call.

To keep an argument open, just replace it by a question mark:

u := **agent** *a0.f* (*a1*, *a2*, *a3*) -- All closed (as before)

w := **agent** *a0.f* (*a1*, *a2*, ?)

x := **agent** *a0.f* (*a1*, ?, *a3*)

y := **agent** *a0.f* (*a1*, ?, ?)

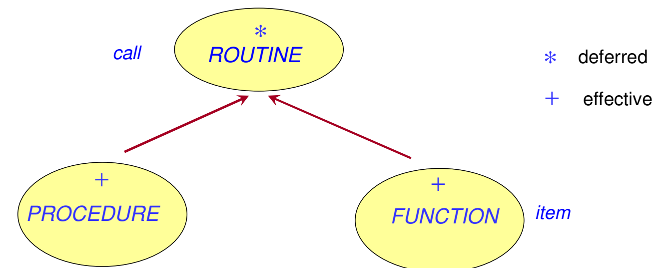
z := **agent** *a0.f* (?, ?, ?)

Calling an agent with arguments

f (*x1*: *T1*; *x2*: *T2*; *x3*: *T3*)
a0: *C*; *a1*: *T1*; *a2*: *T2*; *a3*: *T3*

u := **agent** *a0.f* (*a1*, *a2*, *a3*)	*u.call* ([])
v := **agent** *a0.f* (*a1*, *a2*, ?)	*v.call* ([*a3*])
w := **agent** *a0.f* (*a1*, ?, *a3*)	*w.call* ([*a2*])
x := **agent** *a0.f* (*a1*, ?, ?)	*x.call* ([*a2*, *a3*])
y := **agent** *a0.f* (?, ?, ?)	*y.call* ([*a1*, *a2*, *a3*])

EiffelBase classes representing agents



Agent types

ROUTINE [*BASE*, *ARGS* → *TUPLE*]

PROCEDURE [*BASE*, *ARGS* → *TUPLE*]

FUNCTION [*BASE*, *ARGS* → *TUPLE*, *RESTYPE*]

Declaring an agent

p: *PROCEDURE* [*ANY*, *TUPLE*]
-- Agent representing a procedure,
-- no open arguments

q: *PROCEDURE* [*ANY*, *TUPLE* [*X*, *Y*, *Z*]]
-- Agent representing a procedure,
-- 3 open arguments

f: *FUNCTION* [*ANY*, *TUPLE* [*X*, *Y*, *Z*], *RES*]
-- Agent representing a procedure,
-- 3 open arguments, result of type *RES*

Calling an agent with arguments

f (*x1*: *T1*; *x2*: *T2*; *x3*: *T3*), declared in class *B*
a0: *C*; *a1*: *T1*; *a2*: *T2*; *a3*: *T3*

u := **agent** *a0*.*f* (*a1*, *a2*, *a3*)	PROCEDURE [B, TUPLE [ ]]	u.call ([])
v := **agent** *a0*.*f* (*a1*, *a2*, ?)	PROCEDURE [B, TUPLE [T3]]	v.call ([a3])
w := **agent** *a0*.*f* (*a1*, ?, *a3*)	PROCEDURE [B, TUPLE [T2]]	w.call ([a2])
x := **agent** *a0*.*f* (*a1*, ?, ?)	PROCEDURE [B, TUPLE [T2, T3]]	x.call ([a2, a3])
y := **agent** *a0*.*f* (?, ?, ?)	PROCEDURE [B, TUPLE [T1, T2, T3]]	y.call ([a1, a2, a3])

Making the target open

Open or closed arguments

agent {*TARGET_TYPE*}.*f* (...)

Iterating on the target or the arguments

Procedures in class *COMPANY*:

- *downgrade* -- No argument
- *record_value* (*val: REAL; d: DATE*) -- Two arguments

Then with

companies: *LIST* [*COMPANY*]
values: *LIST* [*REAL*]
some_company: *COMPANY*

you may use both:

companies.do_all (**agent** {*COMPANY*}.*downgrade*)

values.do_all (**agent** *my_company.record_value* (?, *Today*))

End of lecture 21