**Object-Oriented Software Construction**

Bertrand Meyer

**Lecture 21: Agents and tuples**

---

**Handling traditional input**

Program drives input:

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

---

**Agents: the basic idea**

Encapsulating routines in objects

- `agent r`
- `agent r(x, ?, y)`

Mechanism will first be illustrated through event-driven programming

---

**Handling input with modern GUIs**

User drives program:

"When a user presses this button, execute that action from my program"
Event-driven programming

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

A solution: Observer Pattern

Observer pattern

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

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

### Another approach: action-event table

**Set of triples**

\[[\text{Event}, \text{Context}, \text{Action}]\]

**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.

### 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

### 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 (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: 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: 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

Event Library style

The basic class is \texttt{EVENT\_TYPE}

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

\begin{itemize}
  \item (Once) declare event type:
    \begin{verbatim}
    click: EVENT\_TYPE[TUPLE[INTEGER, INTEGER]]
    \end{verbatim}
  \item (Once) create event type object:
    \begin{verbatim}
    create click
    \end{verbatim}
  \item To trigger one occurrence of the event:
    \begin{verbatim}
    click.publish([x\_coordinate, y\_coordinate])
    \end{verbatim}
\end{itemize}

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

\begin{verbatim}
    click.subscribe(agent my\_procedure)
\end{verbatim}

The EiffelVision style

\begin{verbatim}
YES\_button.click.action\_list.extend(agent my\_procedure)
\end{verbatim}

Subscriber variants

\begin{verbatim}
    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)
\end{verbatim}
Another example of using agents

\[
\int_{a}^{b} my\_function (x) \, dx
\]

\[
\int_{a}^{b} your\_function (x, u, v) \, dx
\]

\[
my\_integrator\_integral (agent my\_function, a, b)
\]

\[
my\_integrator\_integral (agent your\_function (?, u, v), a, b)
\]

Using an iterator

\[
all\_positive := my\_integer\_list, for\_all (agent is\_positive(?))
\]

Applications of agents

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

Iterators

In class \texttt{LINEAR[?],} ancestor to all classes for lists, sequences etc., you will find:

\[
\begin{align*}
&for\_all \\
&there\_exists \\
&do\_all \\
&do\_if \\
&do\_while \\
&do\_until
\end{align*}
\]
**Calling the associated routine**

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

\[ a.\text{call}([\text{horizontal\_position, vertical\_position}]) \]

A tuple

If \( a \) is associated with a function, \( a.\text{item}([\ldots, \ldots]) \) gives the result of applying the function.

---

**Behind agents: Tuples**

Tuple types (for any types \( A, B, C, \ldots \)):

\[
\text{TUPLE} \\
\text{TUPLE}[A] \\
\text{TUPLE}[A, B] \\
\text{TUPLE}[A, B, C] \\
\ldots
\]

A tuple of type \( \text{TUPLE}[A, B, C] \) 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**

\[
\text{integral}( f: \text{FUNCTION}[\text{ANY}, \text{TUPLE}[\text{REAL}], \text{REAL}]; \text{low, high: REAL}) \text{ REAL is}
\]

local \( x: \text{REAL}; i: \text{INTEGER} \)

\[
\text{do from } x := \text{low until } x > \text{high loop}
\]

Result := Result + step * \( f.\text{item}([x]) \)

\[
i := i + 1 \\
x := a + i \times \text{step}
\]

end

---

**Tuple type inheritance**

```
TUPLE
```

```
TUPLE[A]
```

```
TUPLE[A, B]
```

```
...
```
### 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 \( t.put(x, i) \)

### Calling an agent with arguments

\[
f(x1: T1; x2: T2; x3: T3)
\]

\[
a0: C; a1: T1; a2: T2; a3: T3
\]

\[
u := \text{agent } a0.f(a1, a2, a3)
\]

\[
u.call(())
\]

\[
v := \text{agent } a0.f(a1, a2, ?)
\]

\[
v.call([a3])
\]

\[
w := \text{agent } a0.f(a1, ?, a3)
\]

\[
w.call([a2])
\]

\[
x := \text{agent } a0.f(a1, ?, ?)
\]

\[
x.call([a2, a3])
\]

\[
y := \text{agent } a0.f(? , ?, ?)
\]

\[
y.call([a1, a2, a3])
\]

### 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:

\[
u := \text{agent } a0.f(a1, a2, a3) -- \text{All closed (as before)}
\]

\[
w := \text{agent } a0.f(a1, a2, ?)
\]

\[
x := \text{agent } a0.f(a1, ?, a3)
\]

\[
y := \text{agent } a0.f(a1, ?, ?)
\]

\[
z := \text{agent } a0.f(? , ?, ?)
\]

### EiffelBase classes representing agents

- **Routine**: `call` + `*` deferred + `*` effective
- **Procedure**: `+` `*` item
- **Function**: `+` `*` item
**Agent types**

ROUTINE[BASE, ARGS \rightarrow TUPLE]

PROCEDURE[BASE, ARGS \rightarrow TUPLE]

FUNCTION[BASE, ARGS \rightarrow TUPLE, RESTYPE]

**Calling an agent with arguments**

\[ f(x1: T1; x2: T2; x3: T3), \text{declared in class } B \]
\[ a0: C; a1: T1; a2: T2; a3: T3 \]

\[ u := \text{agent } a0.f(a1, a2, a3) \]

\[ \text{PROCEDURE } [B, \text{TUPLE } \{1\}] \text{ } u:=\text{call } [a1] \]

\[ v := \text{agent } a0.f(a1, a2, ?) \]

\[ \text{PROCEDURE } [B, \text{TUPLE } \{1, 2\}] \text{ } v:=\text{call } [a2] \]

\[ w := \text{agent } a0.f(a1, ?, a3) \]

\[ \text{PROCEDURE } [B, \text{TUPLE } \{1, 2\}] \text{ } w:=\text{call } [a1, a3] \]

\[ x := \text{agent } a0.f(a1, ?, ?) \]

\[ \text{PROCEDURE } [B, \text{TUPLE } \{1, 2, 3\}] \text{ } x:=\text{call } [a1, a2, a3] \]

**Declaring an agent**

\[ p: \text{PROCEDURE } [\text{ANY, TUPLE}] \]
-- Agent representing a procedure, no open arguments

\[ q: \text{PROCEDURE } [\text{ANY, TUPLE } \{X, Y, Z\}] \]
-- Agent representing a procedure, 3 open arguments

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

**Making the target open**

\[ \text{agent } (\text{TARGET\_TYPE} \text{), } f(...) \]

Open or closed arguments
Iterating on the target or the arguments

Procedures in class \texttt{COMPANY}:
- \texttt{downgrade} \quad \text{-- No argument}
- \texttt{record\_value (val: REAL; d: DATE)} \quad \text{-- Two arguments}

Then with

\begin{verbatim}
companies: LIST[COMPANY]
values: LIST[REAL]
some\_company: COMPANY
\end{verbatim}

you may use both:

\begin{verbatim}
companies\_do\_all (agent \texttt{COMPANY}, downgrade)
values\_do\_all (agent my\_company, record\_value (\$T, Today))
\end{verbatim}

End of lecture 21