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

Publishers

Subscribers

Routine

Routine

Routine

Routine
A solution: Observer Pattern

- * Deferred (abstract)
- + Effective (implemented)

Inherits from Client (uses)
Observer pattern

Publisher keeps a list of observers:

\[ \textit{subscribed}: \text{LINKED\_LIST}[\text{OBSERVER}] \]

To register itself, an observer may execute

\[ \textit{subscribe} (\text{some\_publisher}) \]

where \textit{subscribe} is defined in \text{OBSERVER}:

\[ \textit{subscribe} (p: \text{PUBLISHER}) \text{ is} \]

\[ \quad \text{-- Make current object observe } p. \]

\[ \quad \text{require} \]

\[ \quad \quad \textit{publisher\_exists}: p \neq \text{Void} \]

\[ \quad \text{do} \]

\[ \quad \quad p.\text{attach} (\text{Current}) \]

\[ \quad \text{end} \]
Attaching an observer

In class \texttt{PUBLISHER}:
\begin{verbatim}
attach (s: SUBSCRIBER) is
    -- Register \texttt{s} as subscriber to current publisher.
    require
        subscriber_exists: p /= Void
    do
        subscribed.extend (s)
    end
\end{verbatim}

Note that invariant of \texttt{PUBLISHER} includes the clause

$$\text{subscribed} /= \text{Void}$$

(List \texttt{subscribed} is created by creation procedures of \texttt{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 (1)

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

```csharp
public class ClickArgs {... int x, y; ...}
```

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

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

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

```csharp
public event ClickDelegate Click
**With .NET delegates: publisher (2)**

**P4.** Write new procedure `OnClick` to wrap handling:

```csharp
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:

```csharp
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
The EiffelVision style

YES_button.click.action_list.extend(agent my_procedure)
Event Library style

The basic class is \texttt{EVENT\_TYPE}.

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

- \texttt{(Once) declare event type:}
  
  \texttt{click: EVENT\_TYPE [TUPLE [INTEGER, INTEGER]]}

- \texttt{(Once) create event type object:}
  
  \texttt{create click}

- \texttt{To trigger one occurrence of the event:}
  
  \texttt{click.publish ([x\_coordinate, y\_coordinate])}

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

\texttt{click.subscribe (agent my\_procedure)}
Subscriber variants

\texttt{click.subscribe(\texttt{agent my\_procedure})}

\texttt{my\_button. click.subscribe(\texttt{agent my\_procedure})}

\texttt{click.subscribe(\texttt{agent your\_procedure(a, ?, ?, b)})}

\texttt{click.subscribe(\texttt{agent other\_object. other\_procedure})}
Another example of using agents

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

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

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

\[
my\_integrator\_integral\left(agent\ your\_function(?, u, v), a, b\right)
\]
Applications of agents

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

\[
all_{\text{positive}} := \text{my\_integer\_list\_for\_all}
\]

\[
(\text{agent is\_positive (?)})
\]
Iterators

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

\begin{verbatim}
  for_all
  there_exists
  do_all
  do_if
  do_while
  do_until
\end{verbatim}
Calling the associated routine

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

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

A tuple

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

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

\[
\text{-- Integral of } f \text{ over the interval } [\text{low, high}]
\]

\[
\text{local} \]

\[
x : \text{REAL}; \ i : \text{INTEGER}
\]

\[
do \]

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

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

\[
i := i + 1
\]

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

\[
\text{end}
\]

\[
\text{end}
\]
Behind agents: Tuples

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

\[
\begin{align*}
\text{TUPLE} \\
\text{TUPLE}[A] \\
\text{TUPLE}[A,B] \\
\text{TUPLE}[A,B,C] \\
\ldots
\end{align*}
\]

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. \([a1, b1, c1]\)
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:

$x MethodInvocation t.item(i)$

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:

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

\[ w \ := \ \text{agent} \ a0.f(a1, a2, ?) \]
\[ x \ := \ \text{agent} \ a0.f(a1, ?, a3) \]
\[ y \ := \ \text{agent} \ a0.f(a1, ?, ?) \]
\[ z \ := \ \text{agent} \ a0.f(?, ?, ?) \]
Calling an agent with arguments

\[ f(x_1: T_1; x_2: T_2; x_3: T_3) \]
\[ a_0: C, a_1: T_1; a_2: T_2; a_3: T_3 \]

\[ u := \text{agent } a_0.f(a_1, a_2, a_3) \]
\[ v := \text{agent } a_0.f(a_1, a_2, ?) \]
\[ w := \text{agent } a_0.f(a_1, ?, a_3) \]
\[ x := \text{agent } a_0.f(a_1, ?, ?) \]
\[ y := \text{agent } a_0.f(? , ?, ?) \]

\[ u\.call([]) \]
\[ v\.call([a_3]) \]
\[ w\.call([a_2]) \]
\[ x\.call([a_2, a_3]) \]
\[ y\.call([a_1, a_2, a_3]) \]
EiffelBase classes representing agents

- **ROUTINE**
  - call
  - deferred
  - effective
- **PROCEDURE**
- **FUNCTION**
  - item
Agent types

ROUTINE [BASE, ARGS → TUPLE]

PROCEDURE [BASE, ARGS → TUPLE]

FUNCTION [BASE, ARGS → TUPLE, RESTYPE]
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 \text{RES} \]
Calling an agent with arguments

\[ f(x_1: T_1; x_2: T_2; x_3: T_3), \text{ declared in class } B \]
\[ a_0: C, a_1: T_1; a_2: T_2; a_3: T_3 \]

\[ u := \text{agent } a_0.f(a_1, a_2, a_3) \]

\[ \text{PROCEDURE } [B, \text{TUPLE } [ ]] \]
\[ u.call([]) \]

\[ v := \text{agent } a_0.f(a_1, a_2, ?) \]

\[ \text{PROCEDURE } [B, \text{TUPLE } [T_3]] \]
\[ v.call([a_3]) \]

\[ w := \text{agent } a_0.f(a_1, ?, a_3) \]

\[ \text{PROCEDURE } [B, \text{TUPLE } [T_2]] \]
\[ w.call([a_2]) \]

\[ x := \text{agent } a_0.f(a_1, ?, ?) \]

\[ \text{PROCEDURE } [B, \text{TUPLE } [T_2, T_3]] \]
\[ x.call([a_2, a_3]) \]

\[ y := \text{agent } a_0.f(?, ?, ?) \]

\[ \text{PROCEDURE } [B, \text{TUPLE } [T_1, T_2, T_3]] \]
\[ y.call([a_1, a_2, a_3]) \]
Making the target open

agent \{ \text{TARGET\_TYPE} \}.f(...)
Iterating on the target or the arguments

Procedures in class COMPANY:
- **downgrade**
- **record_value**(val: REAL; d: DATE)

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