Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Lecture 15: Event-driven programming and Agents
Our goal for this lecture

Extend our control structures with a more flexible mechanism, supporting in particular the needs of interactive, graphical programming (GUI)

The resulting mechanism, agents, has many other exciting applications

Other languages have facilities such as delegates (C#), closures (functional languages)
Handling input through traditional techniques

Program drives user:

from
  i := 0
  read_line
until end_of_file loop
  i := i + 1
  Result [i ] := last_line
  read_line
end
Handling input with modern GUIs

User drives program:

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

CLICK START STATION ABOVE
Event-driven programming: an example

Specify that when a user clicks this button the system must execute

\[ \text{find_station}(x, y) \]

where \( x \) and \( y \) are the mouse coordinates and \text{find_station} is a specific procedure of your system.
Some issues

1. Keeping the “business model” and the GUI separate
   - Business model (or just model): core functionality of the application
   - GUI: interaction with users

2. Minimizing “glue code” between the two

3. Making sure we keep track of what’s going on
Event-driven programming: a metaphor

Publishers

Subscribers

Routine

Routine

Routine
Observing a value

A = 50%
B = 30%
C = 20%
Model-View Controller

(Trygve Reenskaug, 1979)
Our example

Specify that when a user clicks this button the system must execute

```
find_station(x, y)
```

where \( x \) and \( y \) are the mouse coordinates and \texttt{find_station} is a specific procedure of your system.
Events Overview (from .NET documentation)

**Events** have the following properties:

1. The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
2. An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
3. Events that have no subscribers are never called.
4. Events are commonly used to signal user actions such as button clicks or menu selections in graphical user interfaces.
5. When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see [another section].
6. Events can be used to synchronize threads.
7. In the .NET Framework class library, events are based on the `EventHandler` delegate and the `EventArgs` base class.
Alternative terminologies

Observed / Observer

Subject / Observer

Publish / Subscribe

Event-driven design/programming

In this presentation: Publisher and Subscriber
A solution: the Observer Pattern

Deferred (abstract)

Effective (implemented)

Inherits from

Client (uses)
Design patterns

A design pattern is an architectural scheme — a certain organization of classes and features — that provides applications with a standardized solution to a common problem.

Since 1994, various books have catalogued important patterns. Best known is Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley 1994.
A solution: the Observer Pattern

Deferred (abstract)

Effective (implemented)

Inherits from Client (uses)
Observer pattern

Publisher keeps a (secret) list of observers:

subscribed : LINKED_LIST [SUBSCRIBER]

To register itself, an observer executes

subscribe (some_publisher)

where subscribe is defined in SUBSCRIBER :

subscribe (p: PUBLISHER)

   -- Make current object observe p.
   require
      publisher_exists: p /= Void
   do
      p.attach (Current)
   end
Attaching an observer

In class **PUBLISHER**:

```plaintext
feature {SUBSCRIBER}
    attach (s : SUBSCRIBER)
        -- Register s as subscriber to this publisher.
        require
            subscriber_exists : s /= Void
        do
            subscribed.extend (s)
        end
```

Note that the invariant of **PUBLISHER** includes the clause

```
subscribed /= Void
```

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

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

Each descendant of **SUBSCRIBER**
defines its own version of **update**
Observer pattern (in basic form)

- 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: event-context-action table

Set of triples
[Event type, Context, Action]

Event type: any kind of event we track
  Example: left mouse click

Context: object for which these events are interesting
  Example: a particular button

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

Event-context-action table may be implemented as e.g. a hash table
Event-action table

More precisely: Event\_type - Action Table

More precisely: Event\_type - Context - Action Table

<table>
<thead>
<tr>
<th>Event type</th>
<th>Context</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left_click</td>
<td>Save_button</td>
<td>Save_file</td>
</tr>
<tr>
<td>Left_click</td>
<td>Cancel_button</td>
<td>Reset</td>
</tr>
<tr>
<td>Left_click</td>
<td>Map</td>
<td>Find_station</td>
</tr>
<tr>
<td>Left_click</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Right_click</td>
<td>...</td>
<td>Display_Menu</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Action-event table

Set of triples

[Event, Context, Action]

Event: any occurrence we track
Example: a left click

Context: object for which the event is interesting
Example: the map widget

Action: what we want to do when the event occurs in context
Example: find the station closest to coordinates

Action-event table may have various implementations, e.g. hash table.
In EiffelVision

Paris_map.click.action_list.extend (agent find_station)
Mechanisms in other languages

C and C++: “function pointers”

C#: delegates (more limited form of agents)
In non-O-O languages, e.g. C and Matlab, there is no notion of agent, but you can pass a routine as argument to another routine, as in

```integral (& f, a, b)```

where \( f \) is the function to integrate. \& \( f \) (C notation, one among many possible ones) is a way to refer to the function \( f \). (We need some such syntax because just `\texttt{f}` could be a function call.)

Agents (or delegates in C#) provide a higher-level, more abstract and safer technique by wrapping the routine into an object with all the associated properties.
With .NET delegates: publisher (1)

P1. Introduce **new class** `ClickArgs` inheriting from `EventArgs`, repeating arguments types of `yourProcedure`:

```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, ClickArgs e);
```

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

```csharp
public event ClickDelegate Click;
```
P4. Write new procedure `OnClick` to wrap handling:

```csharp
protected void OnClick (Clickargs c)
{
    if (Click != null) { Click (this, c); }
}
```

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

```csharp
ClickArgs yourClickargs = new Clickargs (h, v);
```

P6. For every event occurrence, trigger event:

```csharp
OnClick (yourClickargs);
```
With .NET delegates: subscriber

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

D2. Instantiate it with `yourProcedure` as argument:

```
myDelegate = new ClickDelegate (yourProcedure);
```

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

```
YES_button.Click += myDelegate;
```
Using the Eiffel approach (Event Library)

Event: each event type will be an object
Example: left click

Context: an object, usually
representing a user interface element
Example: the map

Action: an agent representing a routine
Example: find_station
The Event library

Basically:

- One generic class: EVENT_TYPE
- Two features: publish and subscribe

For example: A map widget Paris_map that reacts in a way defined in find_station when clicked (event left_click):
The basic class is `EVENT_TYPE`

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

- (Once) declare event type:
  ```java
  click : EVENT_TYPE [TUPLE [INTEGER, INTEGER]]
  ```
- (Once) create event type object:
  ```java
  create click
  ```
- To trigger one occurrence of the event:
  ```java
  click.publish ([x_coordinate, y_coordinate])
  ```

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

```java
click.subscribe (agent find_station)
```
Example using the Event library

The subscribers (“observers”) subscribe to events:

\[ \text{Paris\_map.left\_click.subscribe(\text{agent find\_station})} \]

The publisher (“subject”) triggers the event:

\[ \text{left\_click.publish([x\_position, y\_position])} \]

Someone (generally the publisher) defines the event type:

\[ \text{left\_click : EVENT\_TYPE [TUPLE [INTEGER, INTEGER]]} \]
\[ \quad -- \text{Left mouse click events} \]
\[ \quad \text{once} \]
\[ \quad \text{create Result} \]
\[ \quad \text{ensure} \]
\[ \quad \exists: \text{Result} \neq \text{Void} \]
\[ \text{end} \]
Observer pattern vs. Event Library

In case of an existing class **MY_CLASS**:

- **With the Observer pattern:**
  - Need to write a descendant of **SUBSCRIBER** and **MY_CLASS**
  - Useless multiplication of classes

- **With the Event Library:**
  - Can reuse the existing routines directly as agents
Subscriber variants

click.subscribe (agent find_station)

Paris_map.click.subscribe (agent find_station)

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

click.subscribe (agent other_object other_procedure)
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 $C$.

Tuple values: e.g.

[a1, b1, c1, d1]
Labeled tuple types

TUPLE [author : STRING ; year : INTEGER ; title : STRING]

Restricted form of class
A labeled tuple type denotes the same type as unlabeled form, here

TUPLE [STRING , INTEGER , STRING]

but facilitates access to individual elements
To denote a particular tuple (labeled or not):

[“Tolstoi “, 1865, “War and Peace”]

To access tuple elements: use e.g. t.year
What you can do with an agent \( a \)

Call the associated routine through the feature \texttt{call} ,
whose argument is a single tuple:

\[
\texttt{a\_call ([horizontal\_position, vertical\_position])}
\]

If \( a \) is associated with a function, \( \texttt{a\_item ([ ..., ...])} \) gives
the result of applying the function.
Tuples: Procedures vs. Functions

Features applicable to an agent \( a \):

- If \( a \) represents a procedure, \( a.\text{call} ([\text{argument_tuple}]) \)
calls the procedure

- If \( a \) represents a function, \( a.\text{item} ([\text{argument_tuple}]) \)
calls the function and returns its result
Tuple type inheritance

TUPLE

TUPLE [A ]

TUPLE [A, B ]
Using the Eiffel Event Library

The basic class is `TRAFFIC_EVENT_CHANNEL`

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

- (Once) declare event type:
  
  ```plaintext
click: TRAFFIC_EVENT_CHANNEL [TUPLE [INTEGER, INTEGER]]
  ```

- (Once) create event type object:
  
  ```plaintext:create click
  ```

- To trigger one occurrence of the event:
  
  ```plaintext:click.publish ([x_coordinate, y_coordinate])
  ```

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

```plaintext:click.subscribe (agent find_station)
```
What you can do with an agent \texttt{a}

Call the associated routine through the feature \texttt{call}, whose argument is a single tuple:

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

If \texttt{a} is associated with a function, \texttt{a.item([..., ...])} gives the result of applying the function.
Keeping arguments open

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:

\[
\begin{align*}
    u &:= \text{agent } a0.f(a1, a2, a3) \quad \text{-- All closed (as before)} \\
    w &:= \text{agent } a0.f(a1, a2, ?) \\
    x &:= \text{agent } a0.f(a1, ?, a3) \\
    y &:= \text{agent } a0.f(a1, ?, ?) \\
    z &:= \text{agent } a0.f(?, ?, ?)
\end{align*}
\]
Calling the agent

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

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

\[ u \cdot \text{call } ([]) \]
\[ v \cdot \text{call } ([a_3]) \]
\[ w \cdot \text{call } ([a_2]) \]
\[ x \cdot \text{call } ([a_2, a_3]) \]
\[ y \cdot \text{call } ([a_1, a_2, a_3]) \]
Another example of using agents

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

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

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

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

44
The integration function

integral (f : FUNCTION [ANY, TUPLE [REAL], REAL];
        a, b : REAL): REAL

-- Integral of f over interval [a, b]
local x: REAL; i: INTEGER
do
    from x := a until x > b loop
    Result := Result + f.item ([x]) * step
    i := i + 1
    x := a + i * step
end
end
Another application: using an iterator

class C feature
  all_positive, all_married: BOOLEAN

  is_positive (n: INTEGER): BOOLEAN
  -- Is n greater than zero?
  do Result := (n > 0) end

  intlist: LIST [INTEGER]
  emplist: LIST [EMPLOYEE]

  r
  do
    all_positive := intlist.for_all (agent is_positive (?))
    all_married := emplist.for_all (agent {EMPLOYEE} is_married)
  end
end

class EMPLOYEE feature
  is_married: BOOLEAN
  ...
end
Iterators

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

\begin{itemize}
\item \texttt{for\_all}
\item \texttt{there\_exists}
\item \texttt{do\_all}
\item \texttt{do\_if}
\item \texttt{do\_while}
\item \texttt{do\_until}
\end{itemize}
Applications of agents

Patterns: Observer, Visitor, Undo-redo (command)
Iteration
High-level contracts
Numerical programming
Introspection (finding out properties of the program itself)
Kernel library classes representing agents

- ROUTINE
  - PROCEDURE
  - FUNCTION
  - PREDICATE

- call
- last_result
- item
Declaring an agent

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

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

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

\[ f (x_1 : T_1 ; x_2 : T_2 ; x_3 : T_3) \]

\[ a_0 : C ; a_1 : T_1 ; a_2 : T_2 ; a_3 : T_3 \]

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

\[ u . \text{call } ([]) \]
\[ v . \text{call } ([a_3]) \]
\[ w . \text{call } ([a_2]) \]
\[ x . \text{call } ([a_2, a_3]) \]
\[ y . \text{call } ([a_1, a_2, a_3]) \]
Another application: undoing-redoing

Previous solution: using inheritance and a polymorphic list (summarized in the next few slides)

References:

- Chapter 21 of my *Object-Oriented Software Construction*, Prentice Hall, 1997

- Erich Gamma et al., *Design Patterns*, Addison-Wesley, 1995: “Command pattern”
The problem

Enabling users of an interactive system to cancel the effect of the last command

Often implemented as “Control-Z”

Should support multi-level undo-redo (“Control-Y”), with no limitation other than a possible maximum set by the user
Our working example: a text editor

Notion of “current line”.
Assume commands such as:

- **Remove** current line
- **Replace** current line by specified text
- **Insert** line before current position
- **Swap** current line with next if any
- “Global search and replace” (hereafter **GSR**): replace every occurrence of a specified string by another
- ...

This is a line-oriented view for simplicity, but the discussion applies to more sophisticated views
A straightforward solution

Before performing any operation, save entire state

In the example: text being edited, current position in text

If user issues “Undo” request, restore entire state as last saved

But: huge waste of resources, space in particular

**Intuition:** only save the “diff” between states.
Key step in devising a software architecture

Finding the right abstractions

(the interesting object types)

Here:

The notion of “command”
Keeping the history of the session

The history list:

history : TWO WAY LIST [COMMAND]
A command object includes information about one execution of a command by the user, sufficient to:

- **Execute** the command
- **Cancel** the command if requested later

For example, in a **Removal** command object, we need:

- The position of the line being removed
- The content of that line!
deferred class COMMAND feature

done: BOOLEAN
    -- Has this command been executed?

execute
    -- Carry out one execution of this command.

deferred

ensure
    already: done
end

undo
    -- Cancel an earlier execution of this command.

require
    already: done
deferred
end
end
Command class hierarchy

**COMMAND**
- **execute**
- **undo**

**REMOVAL**
- **execute**
- **undo**
  - line: STRING
  - index: INTEGER

**INSERTION**
- **execute**
- **undo**
  - index

...
Underlying class (from business model)

class EDIT_CONTROLLER feature

  text : TWO_WAY_LIST [STRING]

  remove -- Remove line at current position.
    require not off
do text.remove
  end

  put_right (line : STRING) -- Insert line after current position.
    require not after
do text.put_right (line)
  end

end

... also item, index, go_ith, put_left ...

end
class REMOVAL inherit COMMAND feature
  controller : EDIT_CONTROLLER
    -- Access to business model
  line : STRING
    -- Line being removed
  index : INTEGER
    -- Position of line being removed
execute
  -- Remove current line and remember it.
  do
    line := controller.item ; index := controller.index
    controller.remove ; done := True
  end
undo
  -- Re-insert previously removed line.
  do
    controller.go_i_th (index)
    controller.put_left (line)
  end
The history list

A polymorphic data structure:

history : TWO_WAY_LIST [COMMAND]
Reminder: the list of figures

\[
\text{figs} \text{.extend} (\text{p1}); \text{figs} \text{.extend} (\text{c1}); \text{figs} \text{.extend} (\text{c2}); \\
\text{figs} \text{.extend} (\text{e}); \text{figs} \text{.extend} (\text{p2})
\]

\[
\text{figs : LIST [FIGURE]}
\]
\[
\text{p1, p2 : POLYGON}
\]
\[
\text{c1, c2 : CIRCLE}
\]
\[
\text{e : ELLIPSE}
\]

\[
\text{class LIST [G] feature}
\]
\[
\text{extend} (v : G) \text{ do ... end}
\]
\[
\text{last : G}
\]

...
The history list

A polymorphic data structure:

```
history : TWO_WAY_LIST [COMMAND]
```

---

**Diagram:**

- Insertion
- Removal
- Insertion
- Swap

**Annotations:**

- Oldest
- Most recent
Executing a user command

decode_user_request

if “Request is normal command” then
    “Create command object c corresponding to user request”
    history.extend (c)
    c.execute
elseif “Request is UNDO” then
    if not history.before then
        history.item.undo
        history.back
    end
elseif “Request is REDO” then
    if not history.is_last then
        history.forth
        history.item.execute
    end
end
Conditional creation (1)

```
a1 : A
if condition_1 then
    -- “Create a1 as an instance of B”
elseif condition_2 then
    -- “Create a1 as an instance of C”
... etc.

a1 : A; b1 : B; c1 : C; d1 : D; ...
if condition_1 then
    create b1.make (...)
a1 := b1
elseif condition_2 then
    create c1.make (...)
a1 := c1
... etc.
```
Conditional creation (2)

\[
a1 : A \\
\text{if condition}_1 \text{ then} \\
\quad \text{"Create } a1 \text{ as an instance of } B" \\
\text{elseif condition}_2 \text{ then} \\
\quad \text{"Create } a1 \text{ as an instance of } C" \\
... \text{ etc.}
\]

\[
a1 : A \\
\text{if condition}_1 \text{ then} \\
\quad \text{create } \{B\} \ a1.\text{make}(\ldots) \\
\text{elseif condition}_2 \text{ then} \\
\quad \text{create } \{C\} \ a1.\text{make}(\ldots) \\
... \text{ etc.}
\]
decode_user_request

if “Request is normal command” then
   “Create command object c corresponding to user request”
   history.extend (c)
   c.execute
elseif “Request is UNDO” then
   if not history.before then
      history.item.undo
      history.back
   end
elseif “Request is REDO” then
   if not history.is_last then
      -- Ignore excessive requests
      history.forth
      history.item.execute
   end
end
Creating command objects: first approach

c: COMMAND

...

decode_user_request

if "Request is remove" then
    create {REMOVAL} c
elseif "Request is insert" then
    create {INSERTION} c
else
    etc.
Command class hierarchy

- **COMMAND**
  - execute
  - undo
- **REMOVAL**
  - execute
  - undo
  - line: STRING
  - index: INTEGER
- **INSERTION**
  - execute
  - undo
  - index
  - ...
Creating command objects: better approach

Give each command type a number.

Initially, fill an array `commands` with one instance of every command type.

To get a new command object:

"Determine `command_type`"

c := (commands [command_type]).twin
The undo-redo (or “command”) pattern

Has been extensively used (e.g. in EiffelStudio and other Eiffel tools)
Fairly easy to implement
Details must be handled carefully (e.g. some commands may not be undoable)
Elegant use of O-O techniques
Disadvantage: explosion of small classes
Using agents

For each user command, have two routines:

- The routine to do it
- The routine to undo it!
The history list in the undo-redo pattern

history : TWO_WAY_LIST [COMMAND]

Oldest

Insertion  Removal  Insertion  Swap

Most recent
The history list using agents

The history list simply becomes a list of agents pairs:

```
history : TWO_WAY_LIST [TUPLE
  [doer : PROCEDURE [ANY, TUPLE],
  undoer : PROCEDURE [ANY, TUPLE]]
```

Basic scheme remains the same, but no need for command objects any more; the history list simply contains agents.
decode_user_request

if “Request is normal command” then
   “Create command object c corresponding to user request”
   history.extend (c)
   c.execute

elseif “Request is UNDO” then
   if not history.before then
      history.item.undo
      history.back
   end

elseif “Request is REDO” then
   if not history.is_last then
      history.forth
      history.item.execute
   end
end

-- Ignore excessive requests

Removal
Insertion
Swap
Insertion

item
Executing a user command (now)

“Decode user_request giving two agents do_it and undo_it”
if “Request is normal command” then

\[
\text{history.extend ([do_it, undo_it])}
\]
\[
\text{do_it.call ([])}
\]

elseif “Request is UNDO” then

\[
\text{if not history.before then}
\]
\[
\text{history.item.undoer.call ([])}
\]
\[
\text{history.back}
\]
end
elseif “Request is REDO” then

\[
\text{if not history.is_last then}
\]
\[
\text{history.forth}
\]
\[
\text{history.item.doer.call ([])}
\]
end
Undo-redo

People make mistakes!
Even when they don’t mess up, they want to experiment:
undo-redo supports “trial and error” experimental style

Undo-redo pattern:
- Very useful in practice
- Widely used
- Fairly easy to implement
- Excellent illustration of elegant O-O techniques
- Even better with agents!
What we have seen

1. A powerful pattern: Observer
   Applicable to situations where changes to a value may affect many clients

2. General schemes for interactive applications

3. Encapsulating operations as objects
   Agents, delegates, closures...
   Pass operations around
   Store them in tables

4. Applications of agents, e.g. numerical analysis

5. Auxiliary concepts: tuples, once routines

6. How to undo
   A Through the Command pattern
   B Through agents