Case Study
Command Do–Undo Interaction

The Domain

- Interactive systems usually have an **undo** operation to be able to back up one or more steps

The Domain – 2

- Interactive systems usually have an **undo** operation to be able to back up one or more steps
- To preserve symmetry need to have a corresponding **redo** operation

The Domain – 3

- Interactive systems usually have an **undo** operation to be able to back up one or more steps
- To preserve symmetry need to have a corresponding **redo** operation
- One keystroke gives undo another gives redo

The Domain – 4

- Interactive systems usually have an **undo** operation to be able to back up one or more steps
- To preserve symmetry need to have a corresponding **redo** operation
- One keystroke gives undo another gives redo
- Not all actions are undo-able

The Domain – 5

- Interactive systems usually have an **undo** operation to be able to back up one or more steps
- To preserve symmetry need to have a corresponding **redo** operation
- One keystroke gives undo another gives redo
- Not all actions are undo-able
  - » **Which ones?**
    What are their properties?

The Domain

- Interactive systems usually have an **undo** operation to be able to back up one or more steps
- To preserve symmetry need to have a corresponding **redo** operation
- One keystroke gives undo another gives redo
- Not all actions are undo-able
  - » **Which ones?**
    What are their properties?
    - > **print, erase, fire missile**
    - > **Have side effects outside of the model**

The Requirements

- Should be applicable to a wide class of interactive applications

The Requirements – 2

- Should be applicable to a wide class of interactive applications
- Should not require redesign for each new command that can be undone

The Requirements – 3

- Should be applicable to a wide class of interactive applications
- Should not require redesign for each new command that can be undone
  - » **Implies that undo and redo are different in nature than the other commands**

The Requirements – 4

- Should be applicable to a wide class of interactive applications
- Should not require redesign for each new command that can be undone
  - » **Implies that undo and redo are different in nature than the other commands**
- Make reasonable use of storage

© Gunnar Gotshalks

The Requirements – 5

- Should be applicable to a wide class of interactive applications
- Should not require redesign for each new command that can be undone
  - » **Implies that undo and redo are different in nature than the other commands**
- Make reasonable use of storage
  - » **Cannot save entire state**
    - > **Incremental saves**

The Requirements – 6

- Should be applicable to a wide class of interactive applications
- Should not require redesign for each new command that can be undone
  - » **Implies that undo and redo are different in nature than the other commands**
- Make reasonable use of storage
  - » **Cannot save entire state**
    - > **Incremental saves**
- Applicable for one-level undo or multi-level undo

Finding the Abstractions

- Undo and redo are properties of particular commands

Finding the Abstractions – 2

- Undo and redo are properties of particular commands
- Redo is actually execution of the command in the current context

Finding the Abstractions – 3

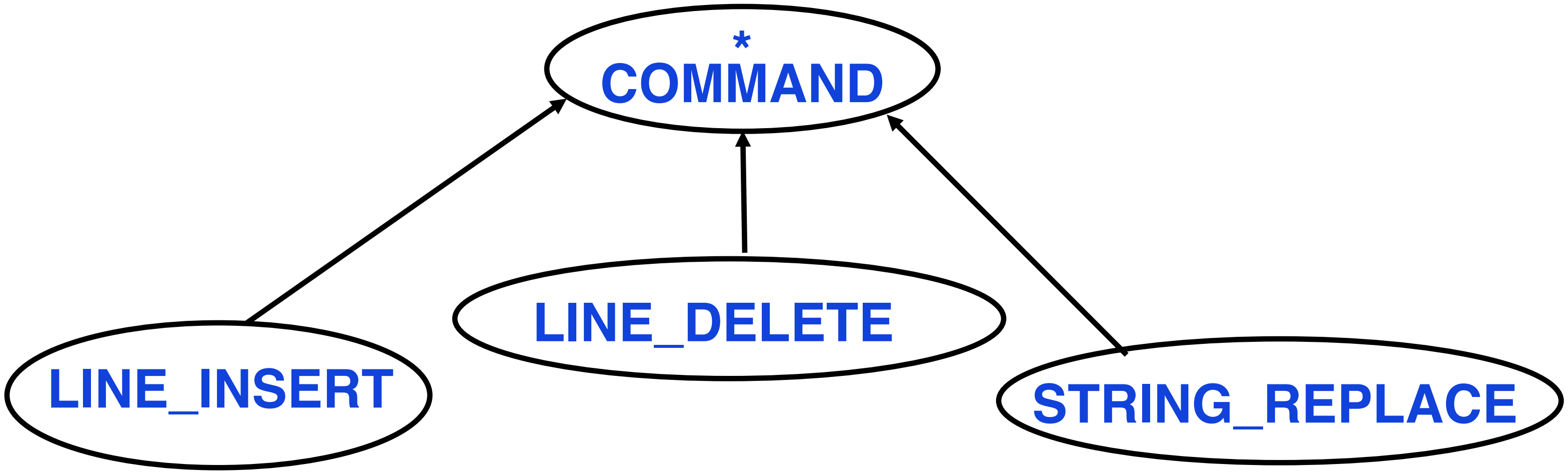
- Undo and redo are properties of particular commands
- Redo is actually execution of the command in the current context
    - **Do not need a separate command**

Finding the Abstractions – 4

- Undo and redo are properties of particular commands
- Redo is actually execution of the command in the current context
  - » **Do not need a separate command**

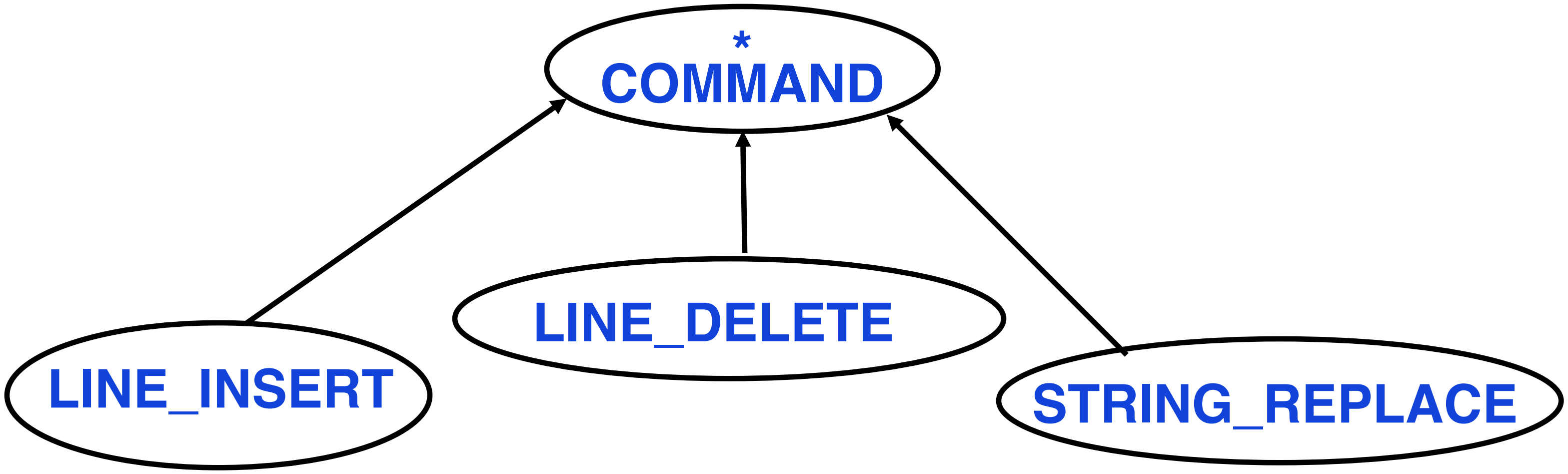
```
deferred class COMMAND
feature
    execute deferred end
    undo deferred end
end
```

Partial Inheritance Hierarchy



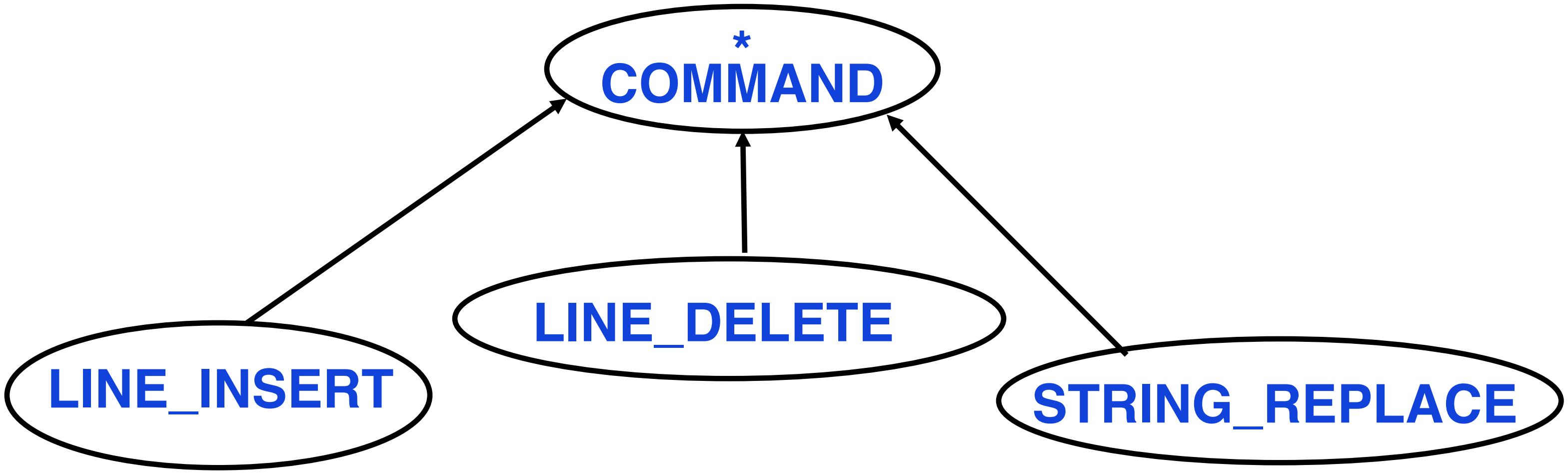
- Each class provides attributes sufficient to support local variants of execute and undo

Partial Inheritance Hierarchy – 2



- Each class provides attributes sufficient to support local variants of execute and undo
- Undo/redo spread through the system

Partial Inheritance Hierarchy – 3



- Each class provides attributes sufficient to support local variants of execute and undo
- Undo/redo spread through the system
  - » **Operations distributed over data**

Class LINE_DELETE

```
class LINE_DELETE inherit  COMMAND
feature
   deleted_line_index : INTEGER
   deleted_line : STRING

   set_deleted_line_index ( n : INTEGER )
      do deleted_line_index := n end

   execute do
      -- delete line
   end

   undo do
      -- restore the last line
   end
end
```

45	deleted_line_index
"text line"	deleted_line

INTERPRETER Class – Run feature

- The root for execution

```
class INTERPRETER create run feature
 ...
  run do
    from
      start
    until
      quit_confirmed
    loop
      interactive_step
    end
  end
 ...
end
```

Interactive Step – 1 level Undo – template

```
interactive_step  do
   -- get latest user request and decode it
   if normal_command then
         -- execute the command
   elseif request is undo then -- toggle undo/redo
      if there is a command to undo then
         -- undo last command
      elseif there is a command to redo then
         -- redo the command
      end
   else report erroneous request
   end
end
```

Interactive Step – One Level Undo

```
requested : COMMAND      -- remember only 1cmd

 interactive_step
local cmd_type : INTEGER
do
   cmd_type :=   get_and_decode_user_request

         -- create object and attach it to requested
   create_command (cmd_type) -- sets requested

   -- Do the command

end
```

Interactive Step – Do the Command

```
if normal_command then
  requested.execute ; undoing := False
elseif request is undo and requested /= void then
  if undoing then --  2'nd undo in a row is a redo !
        requested.execute ; undoing := False
  else requested.undo ; undoing := True
  end
else report erroneous request
end
```

Technicalities

- Do not store the full state, just the difference

Technicalities – 2

- Do not store the full state, just the difference
- Key to solution
  - » **dynamic binding & polymorphism**
    - > **requested.execute & requested.undo**

Technicalities – 3

- Do not store the full state, just the difference
- Key to solution
  - » **dynamic binding & polymorphism**
    - > **requested.execute & requested.undo**
- Nothing application specific
  - » **Add specific subclasses of COMMAND**

Creating a COMMAND Object

- Do after decoding a request

Creating a COMMAND Object – 2

- Do after decoding a request
- All commands created are descendants of COMMAND

Creating a COMMAND Object – 3

- Do after decoding a request
- All commands created are descendants of COMMAND

```
create_command (cmd_type : INTEGER) do
    if cmd_type is Line_Insert then
        create {LINE_INSERT} requested.make(...)
    elseif cmd_type is Line_Delete then
        create {LINE_DELETE} requested.make(...)
    elseif....
end
```

Creating a COMMAND Object – 4

- Do after decoding a request
- All commands created are descendants of COMMAND
- **What about commands with no undo?**

```
create_command (cmd_type : INTEGER) do
    if cmd_type is Line_Insert then
        create {LINE_INSERT} requested.make(...)
    elseif cmd_type is Line_Delete then
        create {LINE_DELETE} requested.make(...)
    elseif....
end
```

Multi-Level Undo

- Need to maintain a history of previous commands

Multi-Level Undo – 2

- Need to maintain a history of previous commands
  - » **Actually keep only the commands in the path from start to last command**

Multi-Level Undo – 3

- Need to maintain a history of previous commands
  - » **Actually keep only the commands in the path from start to last command**
    - > **or as far back as we are able to remember**

Multi-Level Undo – 4

- Need to maintain a history of previous commands
  - » **Actually keep only the commands in the path from start to last command**
    - > **or as far back as we are able to remember**
  - » **Why do we only keep a path?**

Multi-Level Undo – 5

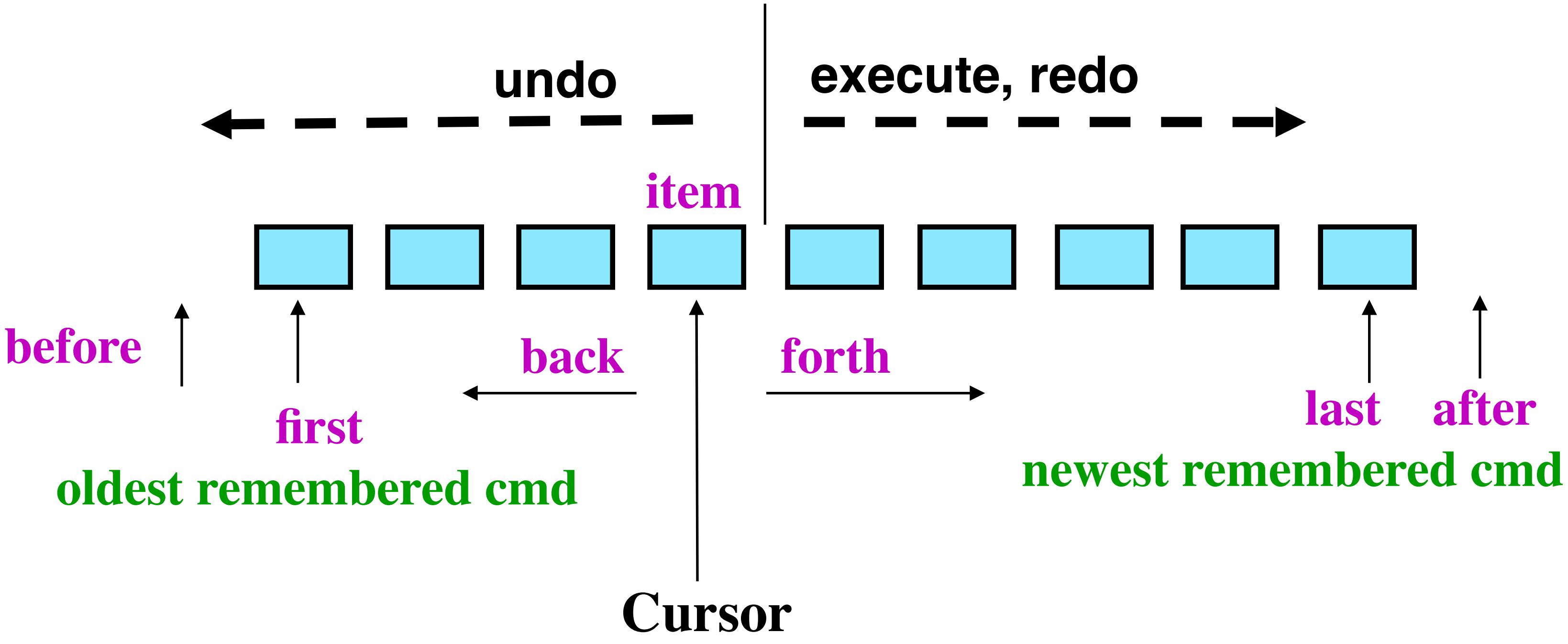
- Need to maintain a history of previous commands
  - » **Actually keep only the commands in the path from start to last command**
    - > **or as far back as we are able to remember**
  - » **Why do we only keep a path?**
    - > **Cognitive constraint**
      - – **Other structures too complex to use**

Multi-Level Undo – 6

- Need to maintain a history of previous commands
  - » **Actually keep only the commands in the path from start to last command**
    - > **or as far back as we are able to remember**
- Also have a cursor to move back and forth through that single path

History List

history : LIST [COMMAND]



Feature names are in magenta

Undo

```
history : LIST [ COMMAND ]

if not history.empty and not history.before then
    history.item.undo
    history.back
else
    message ("Nothing to undo")
end
```

Redo

```
history : LIST [ COMMAND ]

if not history.is_last then
    history.forth
    history.item.execute
else
    message ("Nothing to redo")
end
```

Execute Normal Command

```
history : LIST [ COMMAND ]

if not history.is_last then
    history.remove_all_right
end
    history.put ( requested )
    requested.execute
```

Issue: Command Arguments

- Some commands will need arguments
    - > **LINE_INSERT need lines of text**

Issue: Command Arguments – 2

- Some commands will need arguments
  - LINE_INSERT need lines of text
- Solution
  - Add to COMAND an attribute and a procedure to set the argument

```
argument : ANY

set_argument (a : like argument )
do  argument := a  end
```

Issue: Command Arguments – 4

- Some commands will need arguments
  - > LINE_INSERT need lines of text
- Solution
  - > Add to COMAND an attribute and a procedure to set the argument

Many arguments?

```
argument : ANY
set_argument (a : like argument )
do  argument := a  end
```

Issue: Command Arguments – 5

- Some commands will need arguments
  - > LINE_INSERT need lines of text
- Solution
  - > Add to COMAND an attribute and a procedure to set the argument

```
argument : ANY
set_argument (a : like argument )
do  argument := a  end
```

- Alternate is to pass the argument through execute

```
execute ( argument : ANY ) do ... end
```

© Gunnar Gotshalks

Issue: create_command Structure

- We can do better than the **if ... then ... elseif ...** structure of **create_command**

Issue: create_command Structure – 2

- We can do better than the **if … then … elseif …** structure of **create_command**
- Pre-compute an instance of every command
  - » **polymorphic instance set**

Issue: create_command Structure – 3

- We can do better than the **if ... then ... elseif ...** structure of **create_command**

- Pre-compute an instance of every command
  - » **polymorphic instance set**

```
commands : ARRAY [ COMMAND ]

create commands.make ( 1, command_count )
create {LINE_INSERT} requested .make
   commands[1] := requested
create {LINE_DELETE} requested .make
   commands[2] := requested
...
```

Issue: create_command Structure – 4

- We can do better than the **if ... then ... elseif ...** structure of **create_command**
- Pre-compute an instance of every command
  - » **polymorphic instance set**

Example use of Prototype pattern

```
commands : ARRAY [ COMMAND ]

create commands.make ( 1, command_count )
create {LINE_INSERT} requested .make
   commands[1] := requested
create {LINE_DELETE} requested .make
   commands[2] := requested
...
```

Issue: create_command Structure – 5

- Replace the feature **create_command** with ...

```
requested :=  commands [ cmd_type ] . twin
```

Issue: create_command Structure – 5

- Replace the feature **create_command** with ...

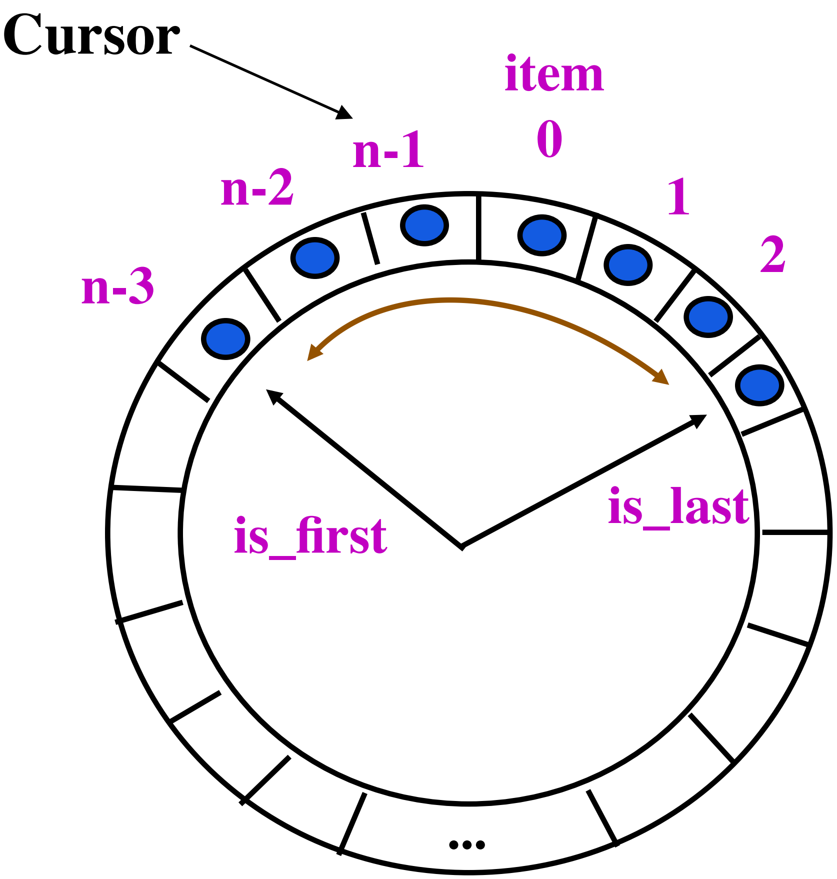
  ```
  requested :=  commands [ cmd_type ] . twin
  ```

- If the argument is passed through execute, then only one instance of each command is needed. Do not need to clone.

  ```
  requested := commands [ cmd_type ]
  ```

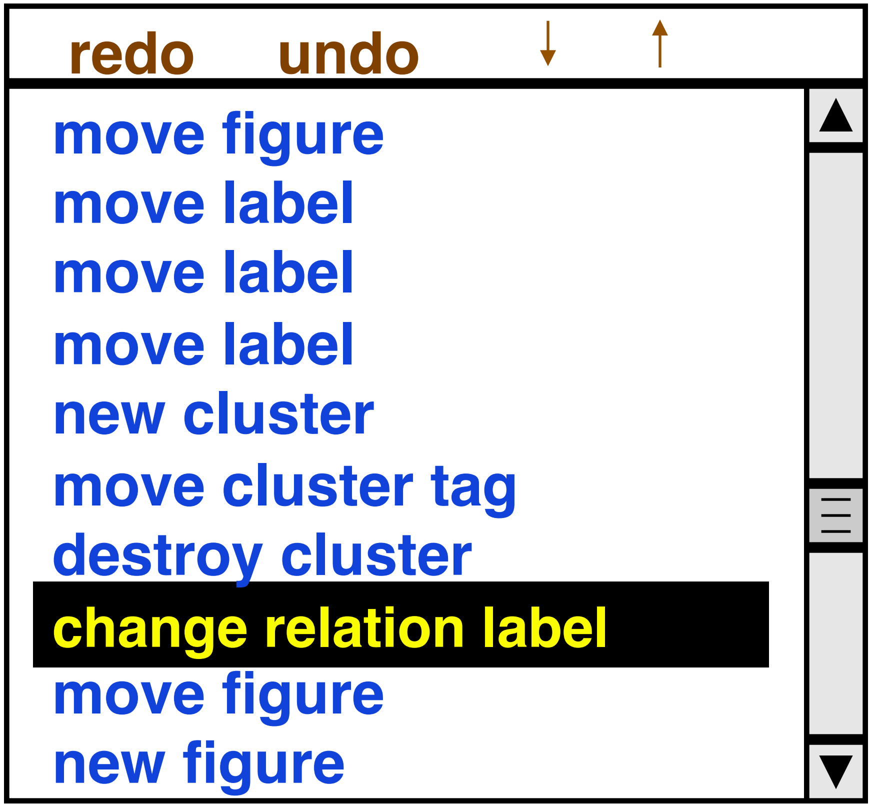
History List Implementation

- Circular Array if bounded capacity is suitable



User Interface

- Correspondence with implementation
  - » **Could have derived either from the other**



move figure
move label
move label
move label
new cluster
move cluster tag
destroy cluster
change relation label
move figure
new figure

Points to Ponder

- Design may involve many relatively small classes
  - » **one for each type of command**

Points to Ponder – 2

- Design may involve many relatively small classes
  - » **one for each type of command**
- Simple inheritance structure, so efficiency is not a concern

Points to Ponder – 3

- Design may involve many relatively small classes
  - » **one for each type of command**
- Simple inheritance structure, so efficiency is not a concern
- Efficiency concerns often arise when you introduce classes to represent actions

Points to Ponder – 4

- Design may involve many relatively small classes
  - » **one for each type of command**
- Simple inheritance structure, so efficiency is not a concern
- Efficiency concerns often arise when you introduce classes to represent actions
  - » **Does this abstraction deserve to be a class?**

Points to Ponder – 5

- Design may involve many relatively small classes
  - » **one for each type of command**
- Simple inheritance structure, so efficiency is not a concern
- Efficiency concerns often arise when you introduce classes to represent actions
  - » **Does this abstraction deserve to be a class?**
    - > **Individual sort algorithms**
    - > **Can pass the algorithm to use in other routines**
    - > **Example sort routine**

InsertSort as Object – Java

```
public class InsertSort implements ArraySort {

  public void sort ( final Object[] array,
                     final BinaryPredicate bp ) {

     execute ( array , bp );
  }

   public static void execute ... // see next slide
     // can also use without an instance in Java
    //        InsertSort.execute (.... )
}

// Notice that BinaryPredicate is also an executable
// object
```

InsertSort – 2

```
public static void execute ( final Object [] array,
                             final BinaryPredicate bp) {
    Object tmp;
    for (int i = 1 ; i < array.length ; i++) {
      for ( int j = i
           ; j > 0  && bp.execute (array [ j ] , array [ j – 1 ] )
           ; j-- ) {
          tmp = array [ j ];
          array[j] = array [ j – 1 ];
          array [ j – 1 ] = tmp;
      }
    }
  }
// BinaryPredicate is an executable object defined in a
// similar way to InsertSort
```

Points to Ponder – 6

- Alternate is to pass functions as arguments

Points to Ponder – 7

- Alternate is to pass functions as arguments
- Example function passing
  - » **Numerical integration that needs the function f to use for integration**

Points to Ponder – 8

- Alternate is to pass functions as arguments
- Example function passing
  - » **Numerical integration that needs the function f to use for integration**
    - > **C approach pass f to the integration routine**
    - > **OO approach pass f as an object**

Points to Ponder – 9

- Alternate is to pass functions as arguments
- Example function passing
  - » **Numerical integration that needs the function f to use for integration**
    - > **C approach pass f to the integration routine**
    - > **OO approach pass f as an object**
      - – **Use data abstraction to make it a class**
      - – **With the desired function as a feature**
      - – **Pass the object to the integration method**

Points to Ponder – 10

- Not all function passing is poor practice

Points to Ponder – 11

- Not all function passing is poor practice
  - > **Different paradigm**

© Gunnar Gotshalks

Points to Ponder – 12

- Not all function passing is poor practice
    - > **Different paradigm**
  - » **Agents in Eiffel**

© Gunnar Gotshalks

Points to Ponder – 13

- Not all function passing is poor practice
    - > Different paradigm
    - » Agents in Eiffel
    - » Functional programming

© Gunnar Gotshalks

Points to Ponder – 14

- Not all function passing is poor practice
    - > Different paradigm
  - » Agents in Eiffel
  - » Functional programming
    - > Pass functions as input

Points to Ponder – 15

- Not all function passing is poor practice
    - > Different paradigm
  - » Agents in Eiffel
  - » Functional programming
    - > Pass functions as input
    - > Return functions as output

Points to Ponder – 10

- Not all function passing is poor practice
    - > **Different paradigm**
  - » **Agents in Eiffel**
  - » **Functional programming**
    - > **Pass functions as input**
    - > **Return functions as output**
      - – **Functions compute functions to use later !**