ETH Zürich
Chair of Software Engineering

Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

Lecture 5: Design Patterns

What is a pattern?

- First developed by Christopher Alexander for constructing and designing buildings and urban areas
- "Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution."

What is a pattern?

- First developed by Christopher Alexander for constructing and designing buildings and urban areas
- "Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution."

Example **Web of Shopping** (*C. Alexander, A pattern language*)

Conflict: Shops rarely place themselves where they best serve people's needs and guarantee their own stability.

Resolution: Locate a shop by the following steps:
1) Identify and locate all shops offering the same service.
2) Identify and map the location of potential consumers.
3) Find the biggest gap in the web of similar shops with potential consumers.
4) Within the gap locate your shop next to the largest cluster of other kinds of shops.

What is a pattern?

- First developed by Christopher Alexander for constructing and designing buildings and urban areas
- "Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution."
- Patterns can be applied to many areas, including software development

Patterns in software development

Design pattern:

- A document that describes a general solution to a design problem that recurs in many applications.

Developers adapt the pattern to their specific application.

Why design patterns?

"Designing object-oriented software is hard and designing reusable object-oriented software is even harder." Erich Gamma

- Experienced object-oriented designers make good designs while novices struggle
- Object-oriented systems have recurring patterns of classes and objects
- Patterns solve specific design problems and make OO designs more flexible, elegant, and ultimately reusable

Benefits of design patterns

- Capture the knowledge of experienced developers
- Publicly available repository
- Common pattern language
- Newcomers can learn & apply patterns
- Yield better software structure
- Facilitate discussions: programmers, managers

History of software design patterns

1987: Ward Cunningham and Kent Beck develop a pattern language with five Smalltalk patterns

1991: Erich Gamma and Richard Helm start jotting down catalog of patterns; first presentation at TOOLS

1991: First Patterns Workshop at OOPSLA

1993: Kent Beck and Grady Booch sponsor the first meeting of the Hillside Group

1994: First Pattern Languages of Programs (PLoP) conference

1994: The Gang of Four (GoF: Erich Gamma and Richard Helm, Ralph Johnson, and John Vlissides) publish the *Design Patterns book*

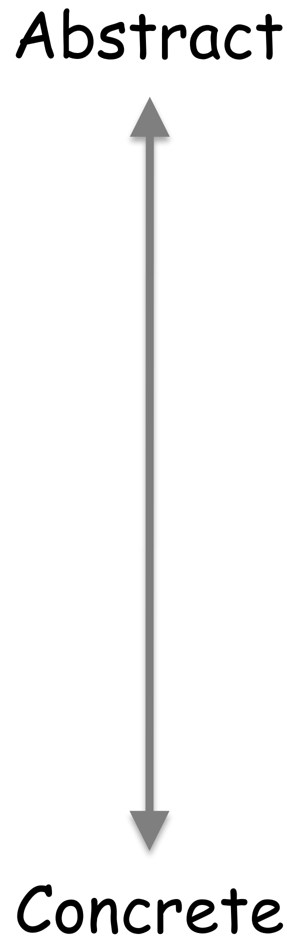
Design patterns

- 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.

Levels of abstraction for design patterns

- Complex design for an entire application or subsystem
- Solution to a general design problem in a particular context
- Simple reusable design class such as a linked list, hash table, etc.



Based on a slide by Bob Tarr, Design Patterns in Java

Gang of Four Design Patterns

- Middle level of abstraction
- "A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design." Gamma et. al.

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Non-GoF patterns

- Model-View-Controller

A pattern is not a reusable solution

Solution to a particular recurring design issue in a particular context:

> "*Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to this problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.*"
>
> Gamma et al.

NOT REUSABLE

A step backwards?

Patterns are not reusable solutions:

- You must implement every pattern every time
- Pedagogical tools, not components

We have done work at ETH to correct this situation:

"*A successful pattern cannot just be a book description: it must be a software component*"

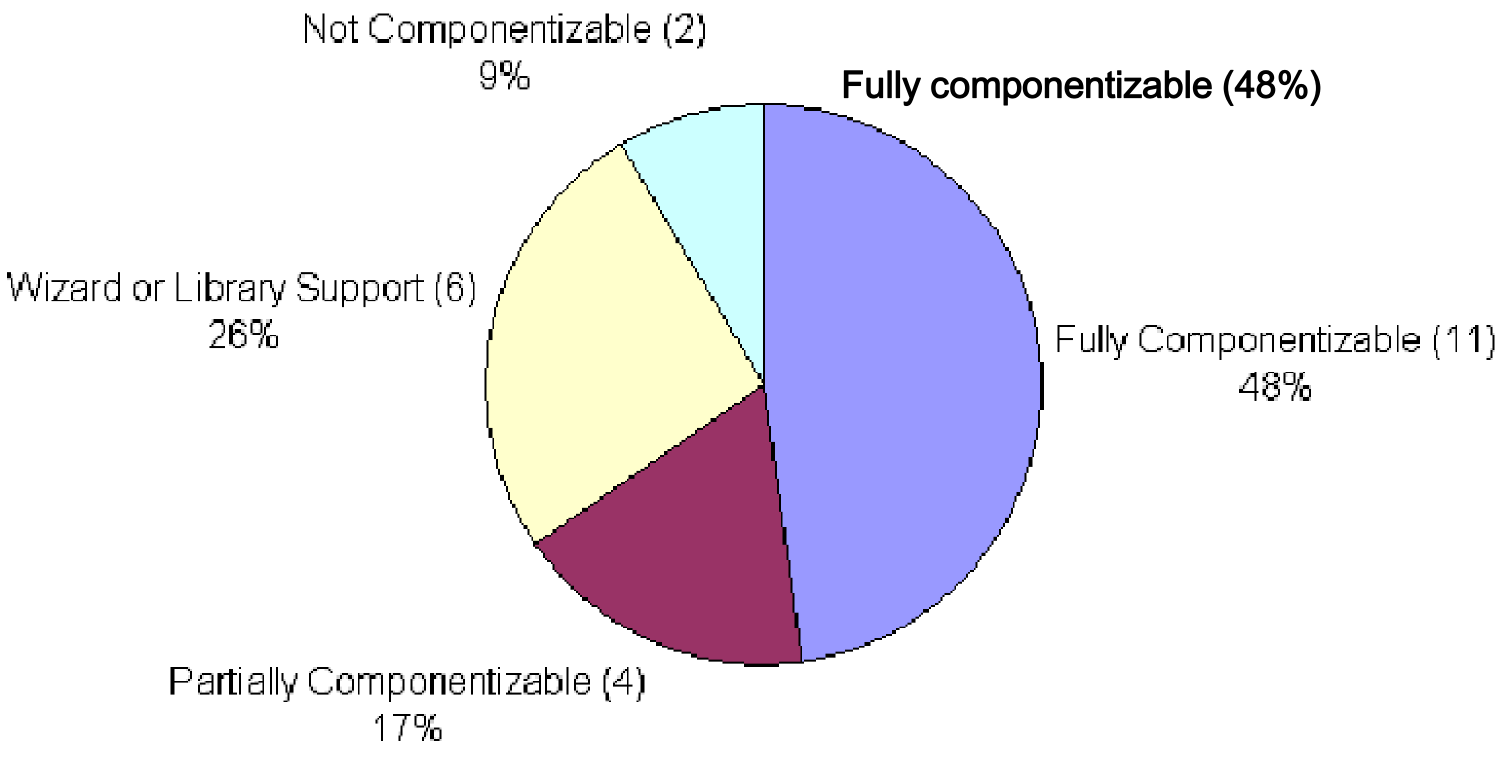
Result: Pattern Library and Pattern Wizard
(see following lectures)

Pattern componentization

Karine Arnout
ETH PhD, 2004

Classification of design patterns:

- Fully componentizable
- Partially componentizable
- Wizard- or library-supported
- Non-componentizable



Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Already covered in Info1

Non-GoF patterns

- Model-View-Controller

Observer pattern and event-driven progr.

Intent: "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

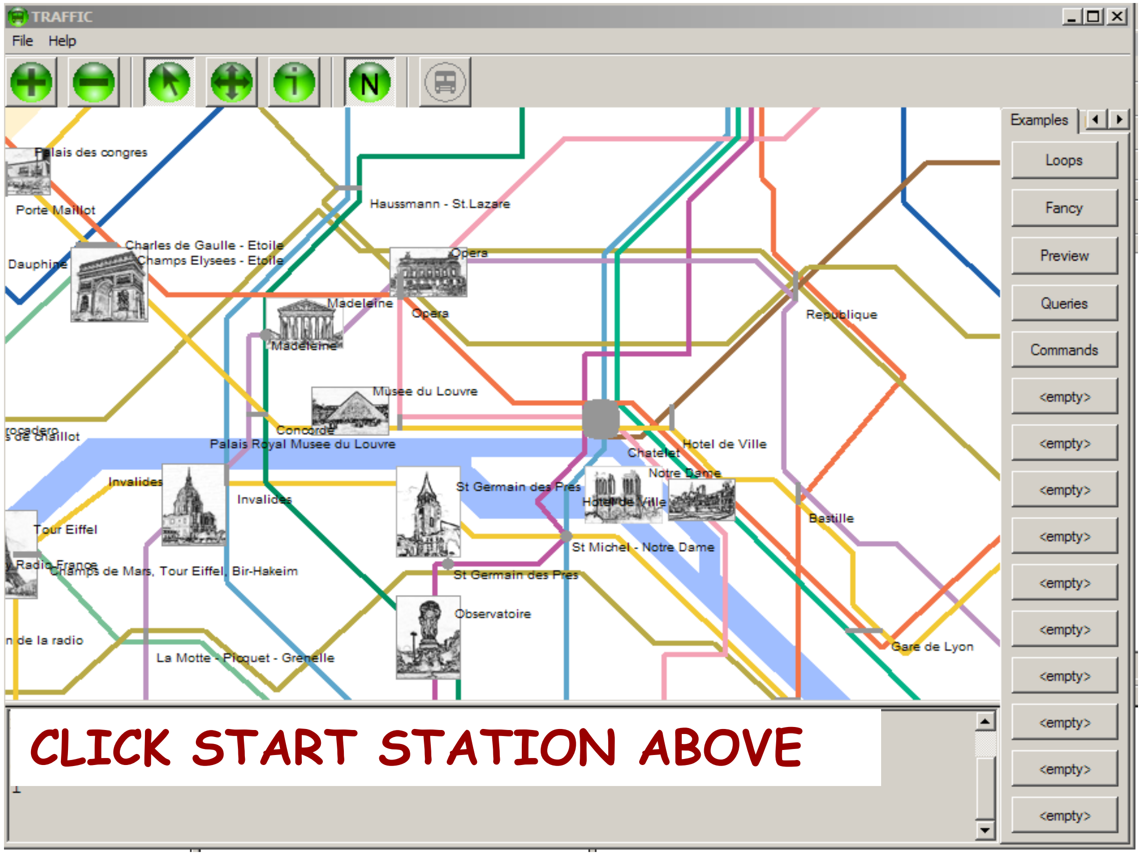
[Gamma et al., p 331]

- Implements publish-subscribe mechanism
- Used in Model-View-Controller patterns, interface toolkits, event
- Reduces tight coupling of classes

Handling input with modern GUIs

User drives program:

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

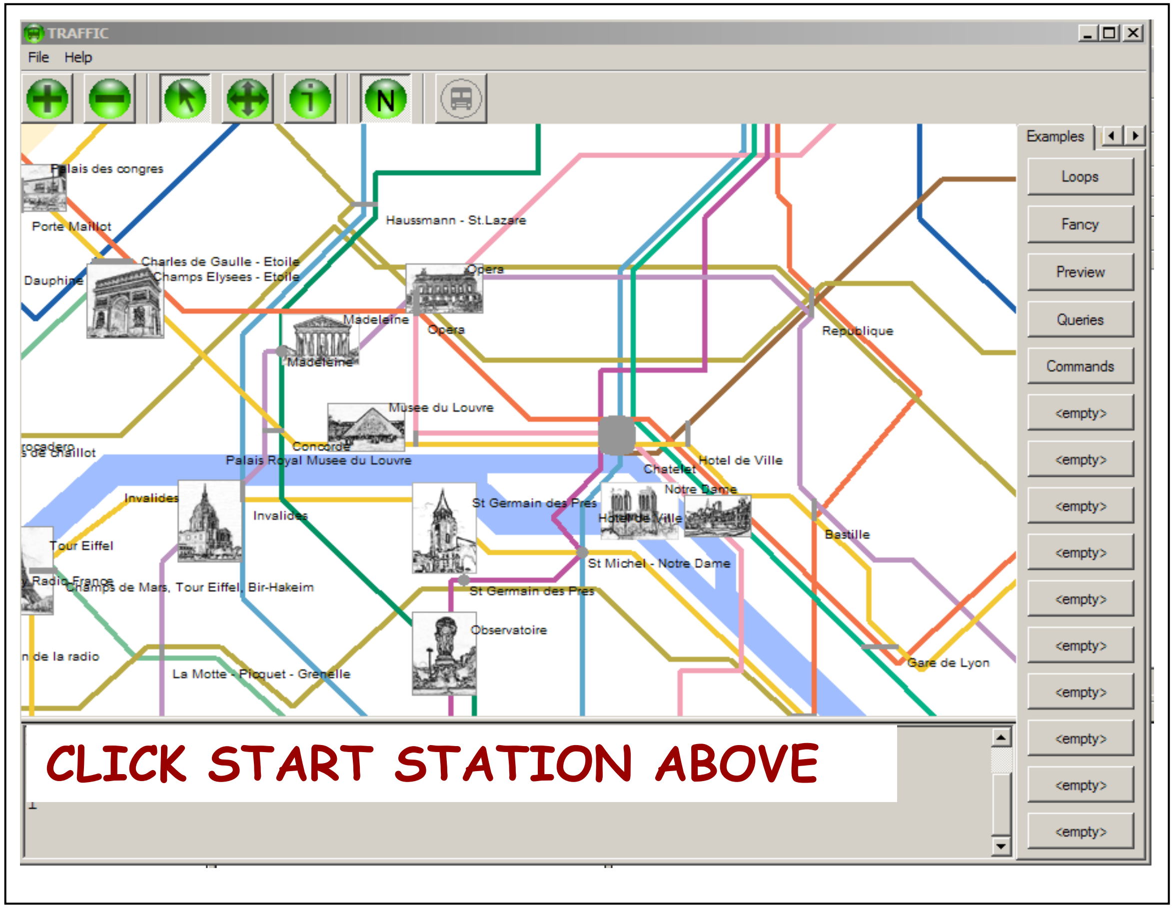


Event-driven programming: an example

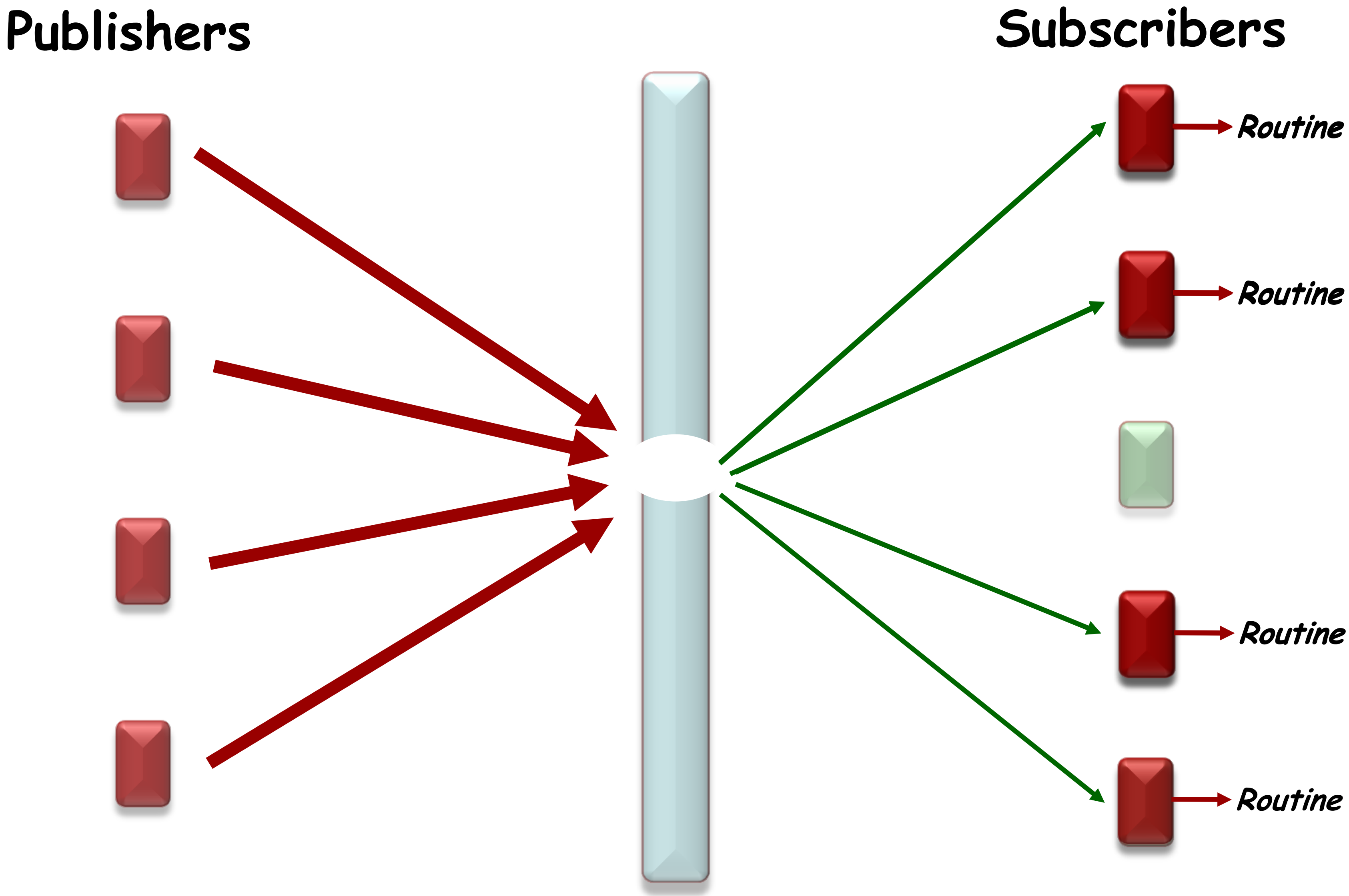
Specify that when a user clicks this button the system must execute

find_station (*x*, *y*)

where *x* and *y* are the mouse coordinates and *find_station* is a specific procedure of your system.



Event-driven programming: a metaphor

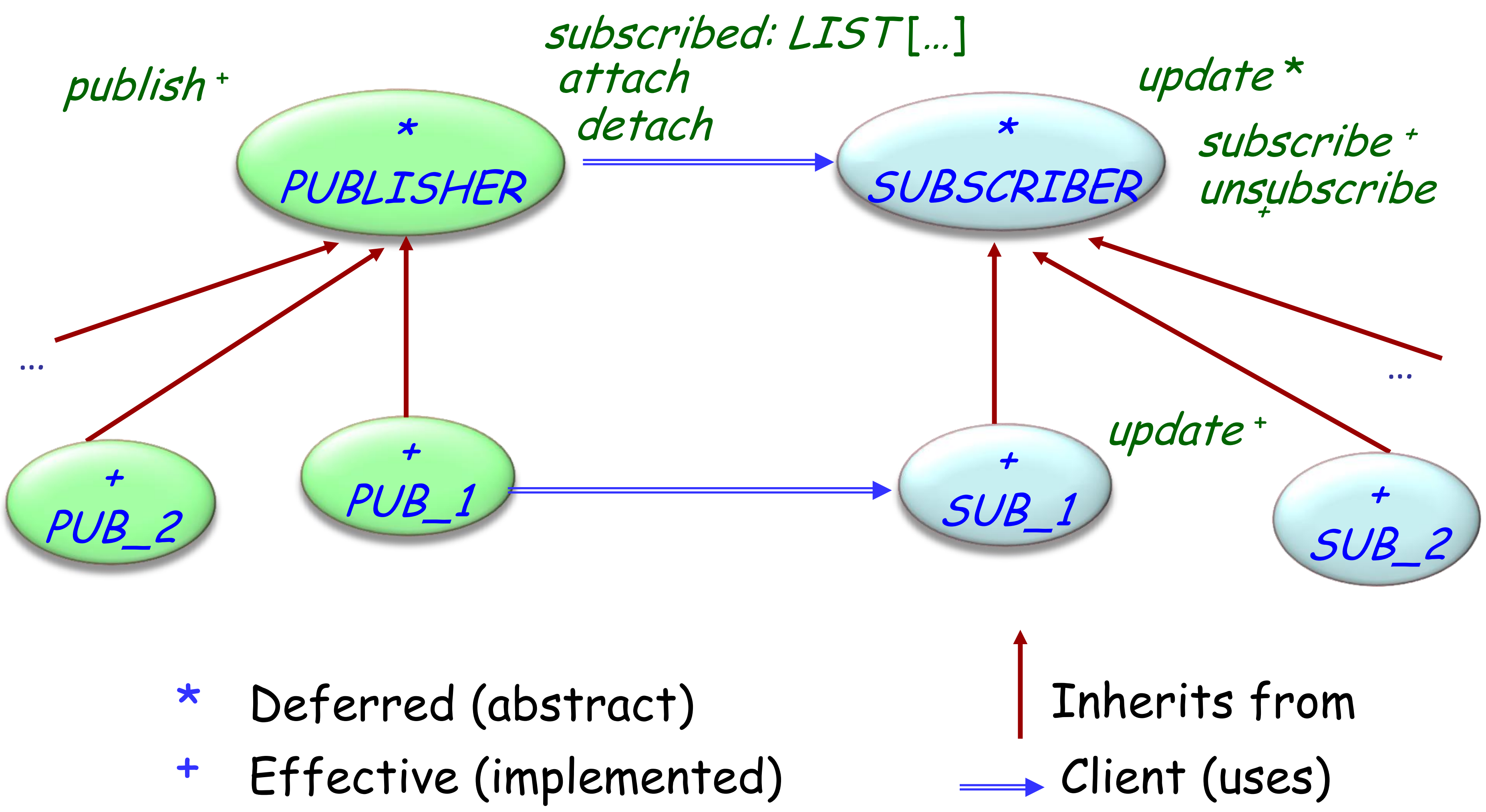


Alternative terminologies

- Observed / Observer
- Subject / Observer
- Publish / Subscribe
- Event-driven design/programming

In this presentation:
Publisher and Subscriber

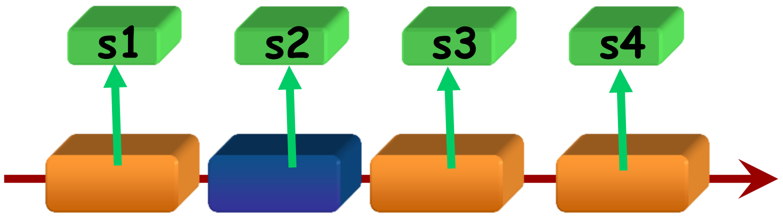
A solution: the Observer Pattern (GoF)



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*:

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

                subscribed.extend (s)
            end
```

Why?

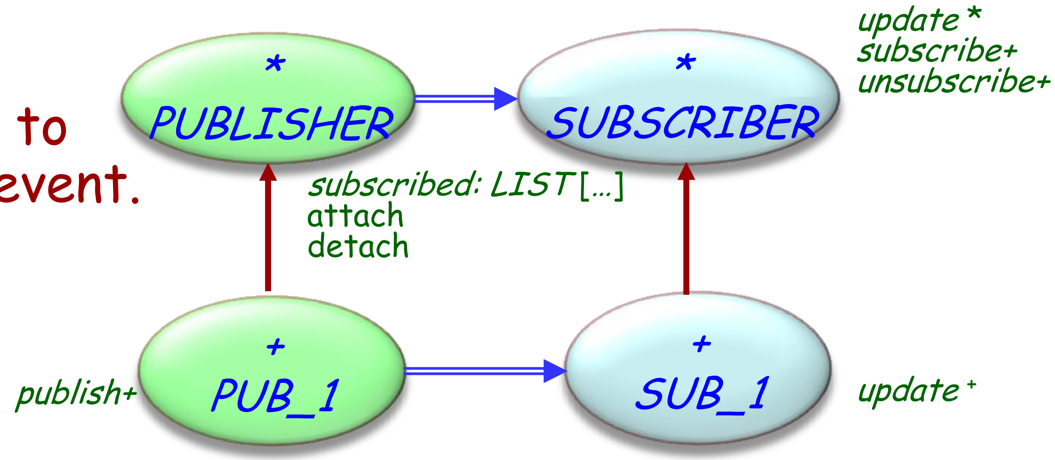
Note that the invariant of *PUBLISHER* includes the clause

subscribed /= Void

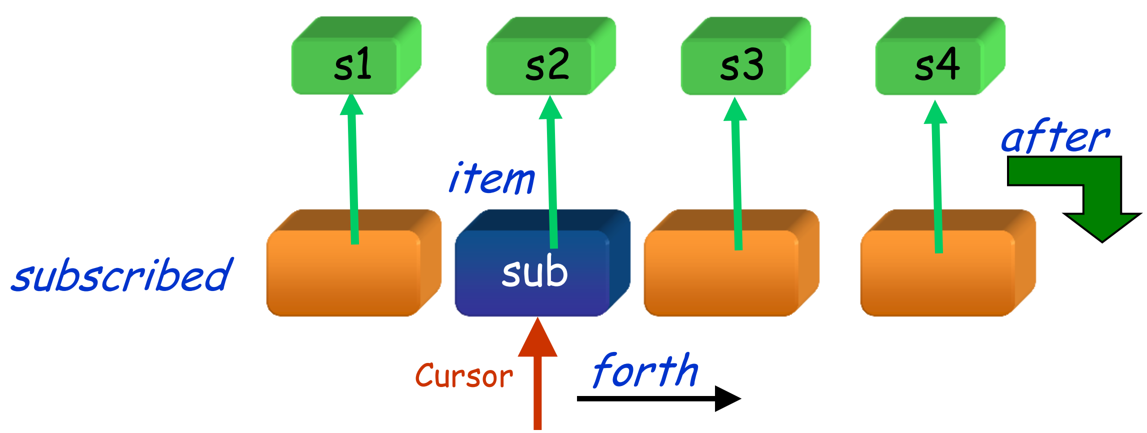
(List *subscribed* is created by creation procedures of *PUBLISHER*)

Triggering an event

```
publish
        -- Ask all observers to
        -- react to current event.
    do
        across
            subscribed
        as
            s
        loop
            s.item.update
        end
    end
```



Dynamic binding



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

Observer - Participants

Publisher

- knows its subscribers. Any number of Subscriber objects may observe a publisher.
- provides an interface for attaching and detaching subscribers.

Subscriber

defines an updating interface for objects that should be notified of changes in a publisher.

Concrete Publisher

- stores state of interest to ConcreteSubscriber objects.
- sends a notification to its subscribers when its state changes.

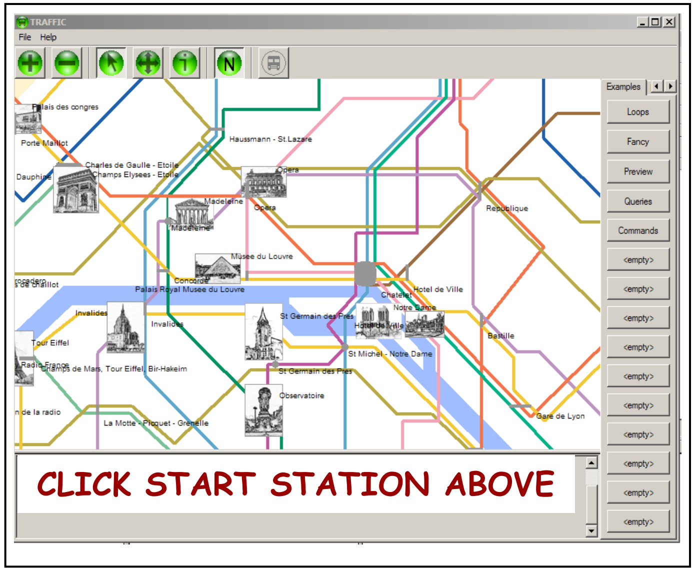
Concrete Subscriber

- maintains a reference to a ConcretePublisher object.
- stores state that should stay consistent with the publisher's.
- implements the Subscriber updating interface to keep its state consistent with the publisher's.

Observer pattern (in basic form)

- Subscriber may subscribe:
  - At most one operation
  - To at most one publisher
- Event arguments are tricky to handle
- Subscriber knows publisher
  (More indirection is desirable)
- Not reusable — must be coded anew for each application

Using agents in EiffelVision



Paris_map.*click*.*subscribe* (**agent** *find_station*)

Mechanisms in other languages

- C and C++: “function pointers”

- C#: delegates (more limited form of agents)

Using agents (Event Library)

Event: each event *type* will be an object

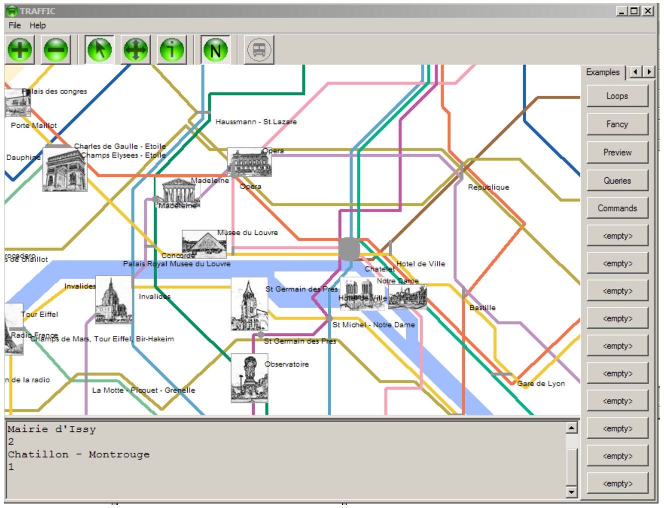
Example: left click

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*):

Event library: a simple implementation

```
class
    EVENT_TYPE [ARGS -> TUPLE]
inherit ANY
    redefine default_create end

feature {NONE} -- Implementation
    subscribers: LINKED_LIST [PROCEDURE [ANY, ARGS]]

feature {NONE} -- Initialization
    default_create
            -- Initialize list.
        do
            create subscribers.make
            subscribers.compare_equal
        end
```

Simplified event library (end)

```
feature -- Basic operations
   subscribe (action: PROCEDURE [ANY, ARGS])
         -- Add action to subscription list.
      require
         exists: action /= Void
      do
         subscribers.extend (action)
      ensure
         subscribed: subscribers.has (action)
      end

   publish (arguments: ARGS)
         -- Call subscribers.
      require
         exist: arguments /= Void
      do
         across subscribers as s loop s.item.call (arguments) end
      end
end
```

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** *find_station*)

Example using the Event library

The subscribers ("observers") subscribe to events:

```
Paris_map.click.subscribe (agent find_station)
```

The publisher ("subject") triggers the event:

```
click.publish ([x_positition, y_position])
```

Someone (generally the publisher) defines the event type :

```
click: EVENT_TYPE [TUPLE [INTEGER, INTEGER]]
		-- Mouse click events
	once
		create Result
	ensure
		exists: Result /= Void
	end
```

Subscriber variants

click.subscribe (**agent** *find_station*)

Paris_map.click.subscribe (**agent** *find_station*)

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

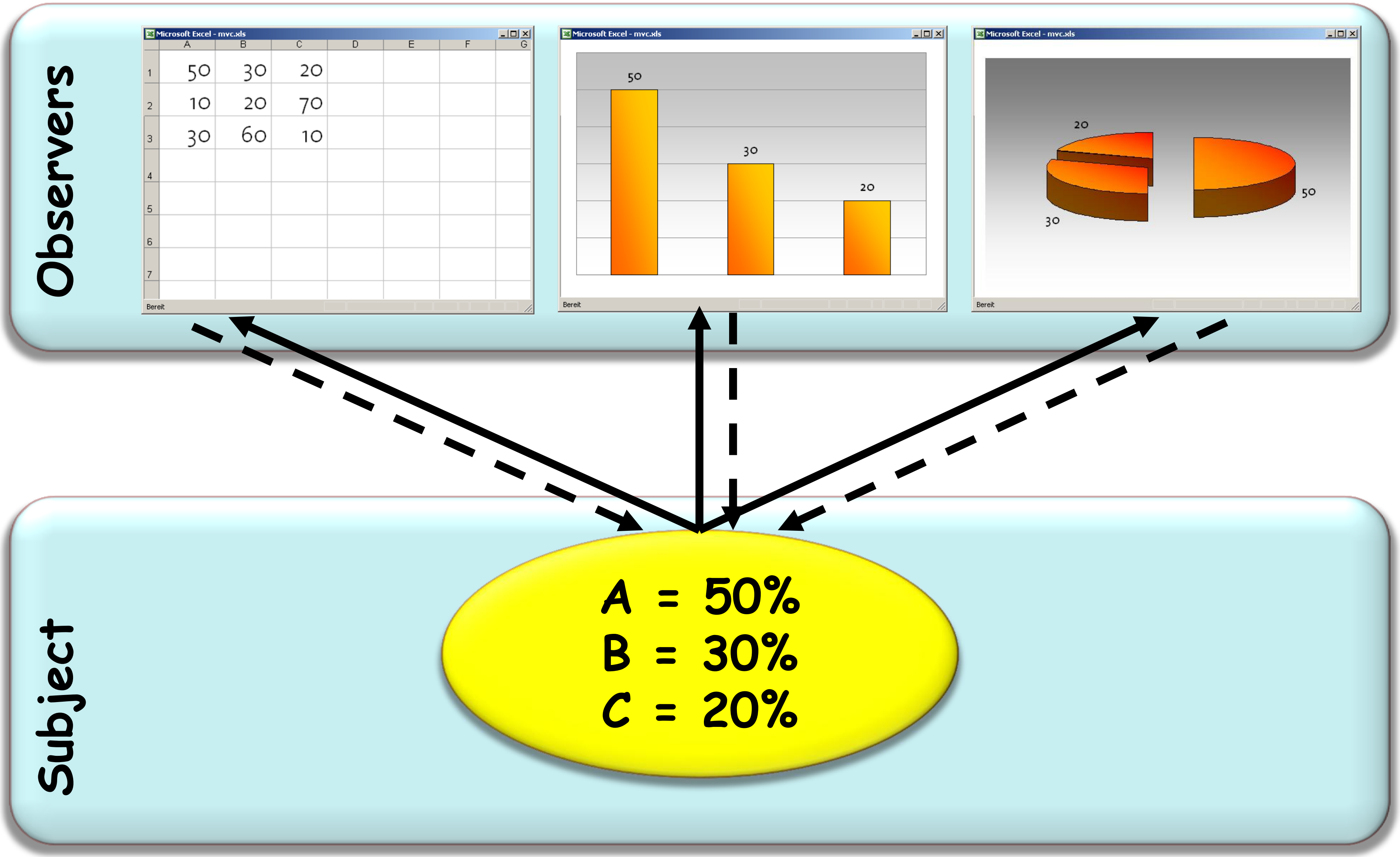
click.subscribe (**agent** *other_object.other_procedure*)

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

Observer and event-driven design



Some issues

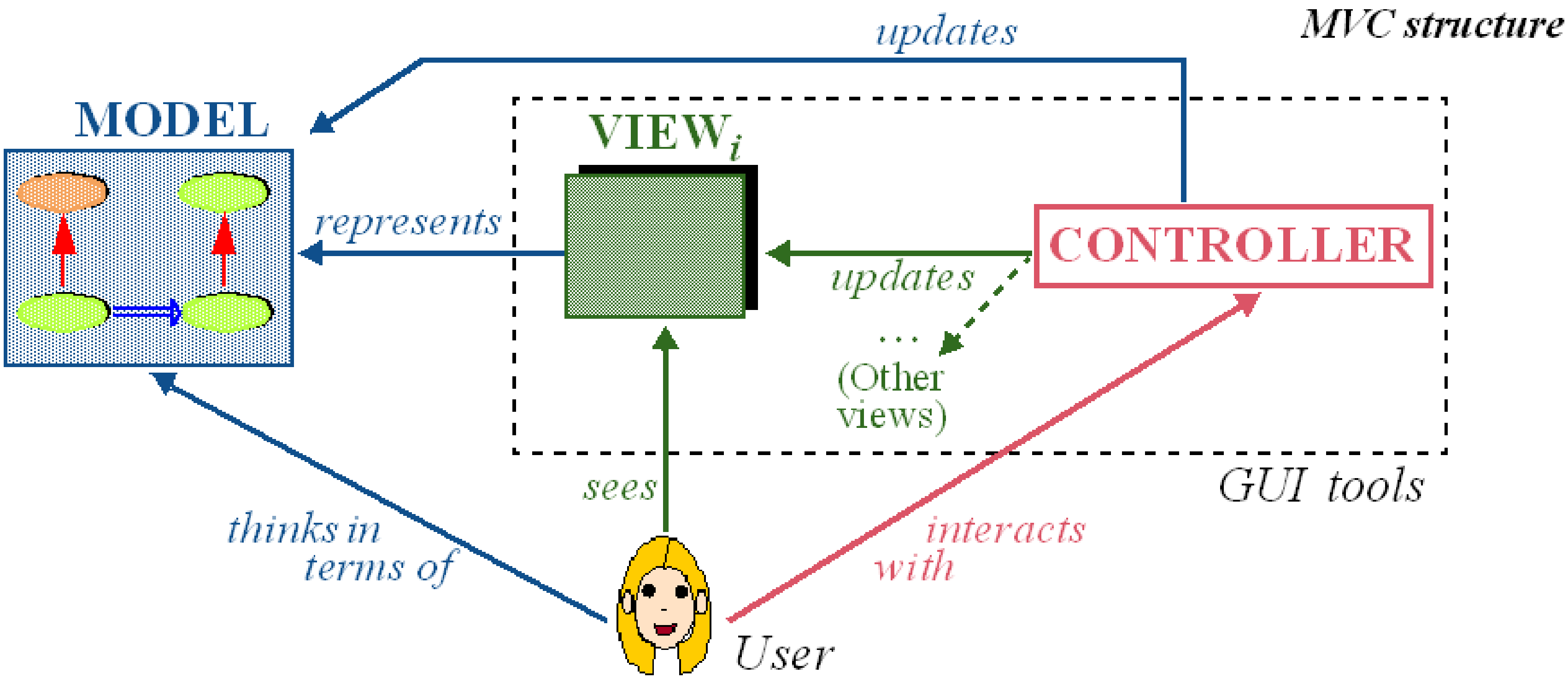
1. Keeping the "business model" and the GUI separate
   - Business model (or just *model*): core functionality of the application
   - GUI: interaction with users

2. Minimizing "glue code" between the two

3. Making sure we keep track of what's going on

Model-View Controller

(Trygve Reenskaug, 1979)



Observer - Consequences

Observer pattern makes the coupling between publishers and subscribers abstract.

Supports broadcast communication since publisher automatically notifies to all subscribers.

Changes to the publisher that trigger a publication may lead to unexpected updates in subscribers.

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

Already covered in Info1

Non-GoF patterns

- ✓ Model-View-Controller

Command pattern

Intent:

"Way to implement an undo-redo mechanism, e.g. in text editors." [OOSC, p 285-290]

"Way to encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations." [Gamma et al., p 233]

Application example

EiffelStudio

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

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

Key step in devising a software architecture

Finding the right abstractions

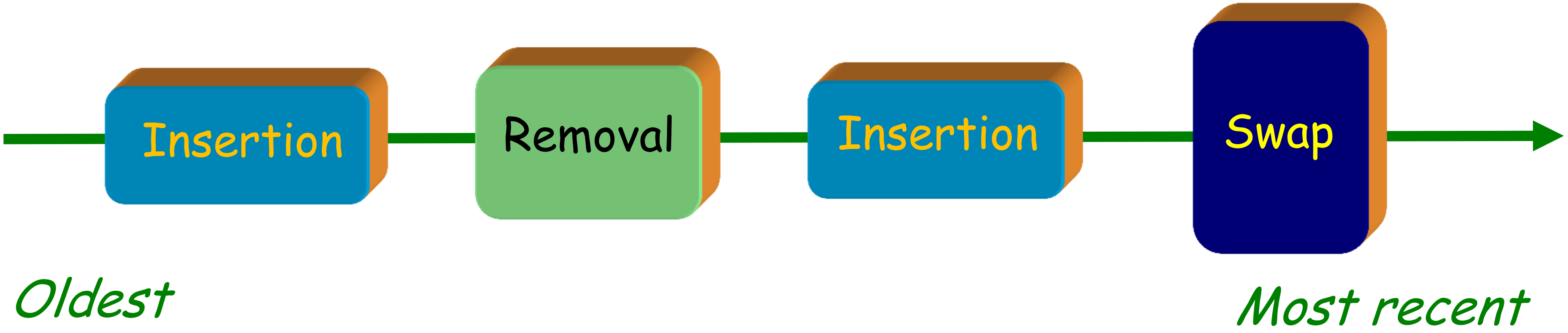
(the interesting object types)

Here:

The notion of "command"

Keeping the history of the session

The history list:



history : *TWO_WAY_LIST* [*COMMAND*]

What's a "command" object?

- 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

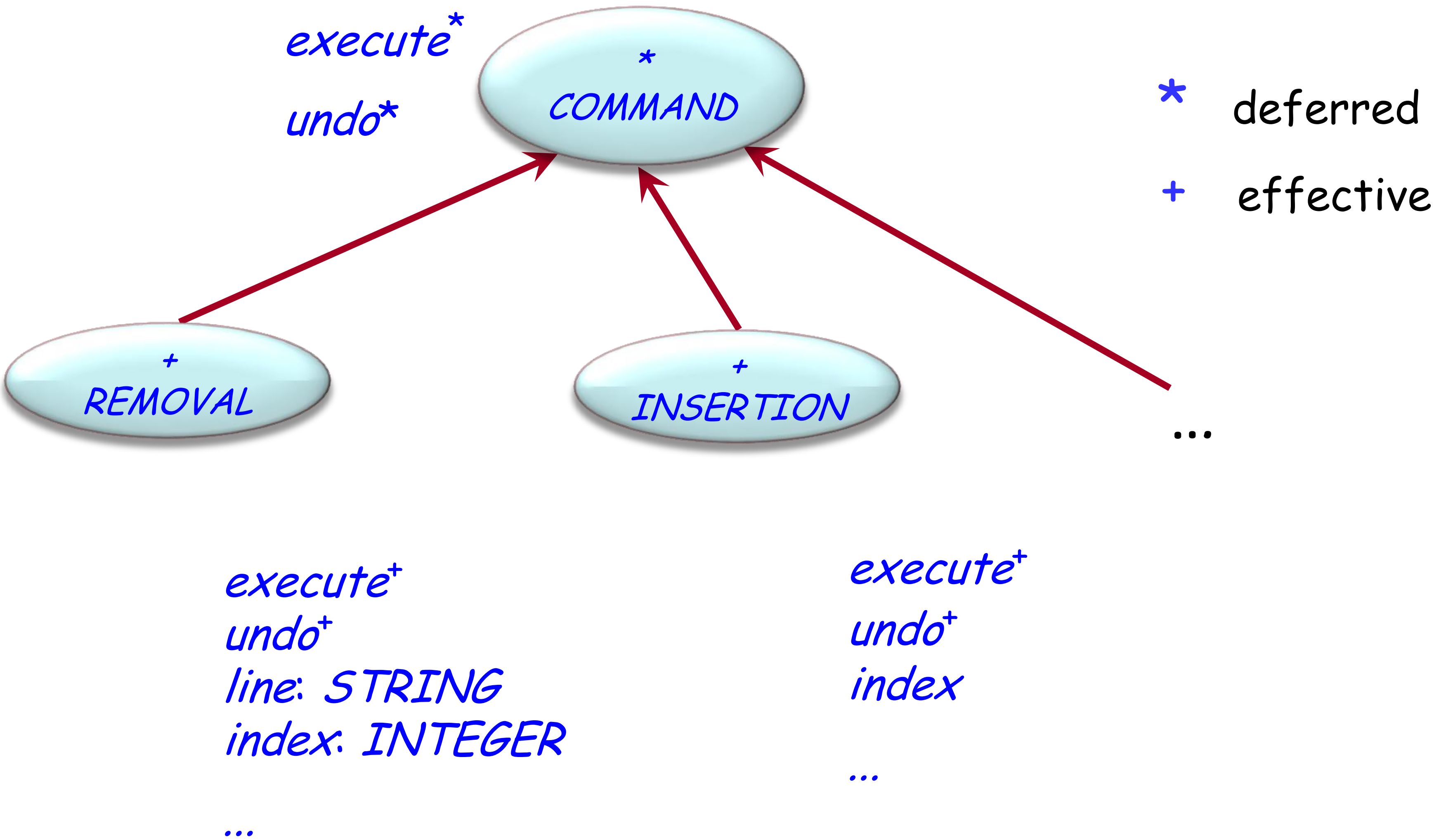
General notion of command

```
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
```

A command class (sketch, no contracts)

```
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
end
```

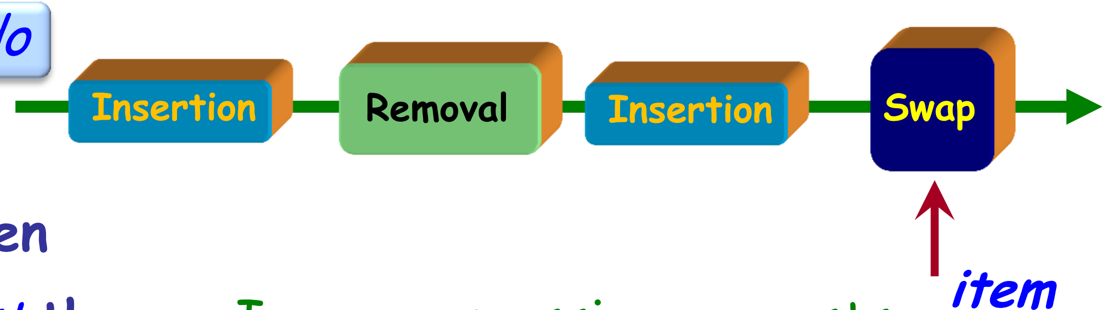
Command class hierarchy



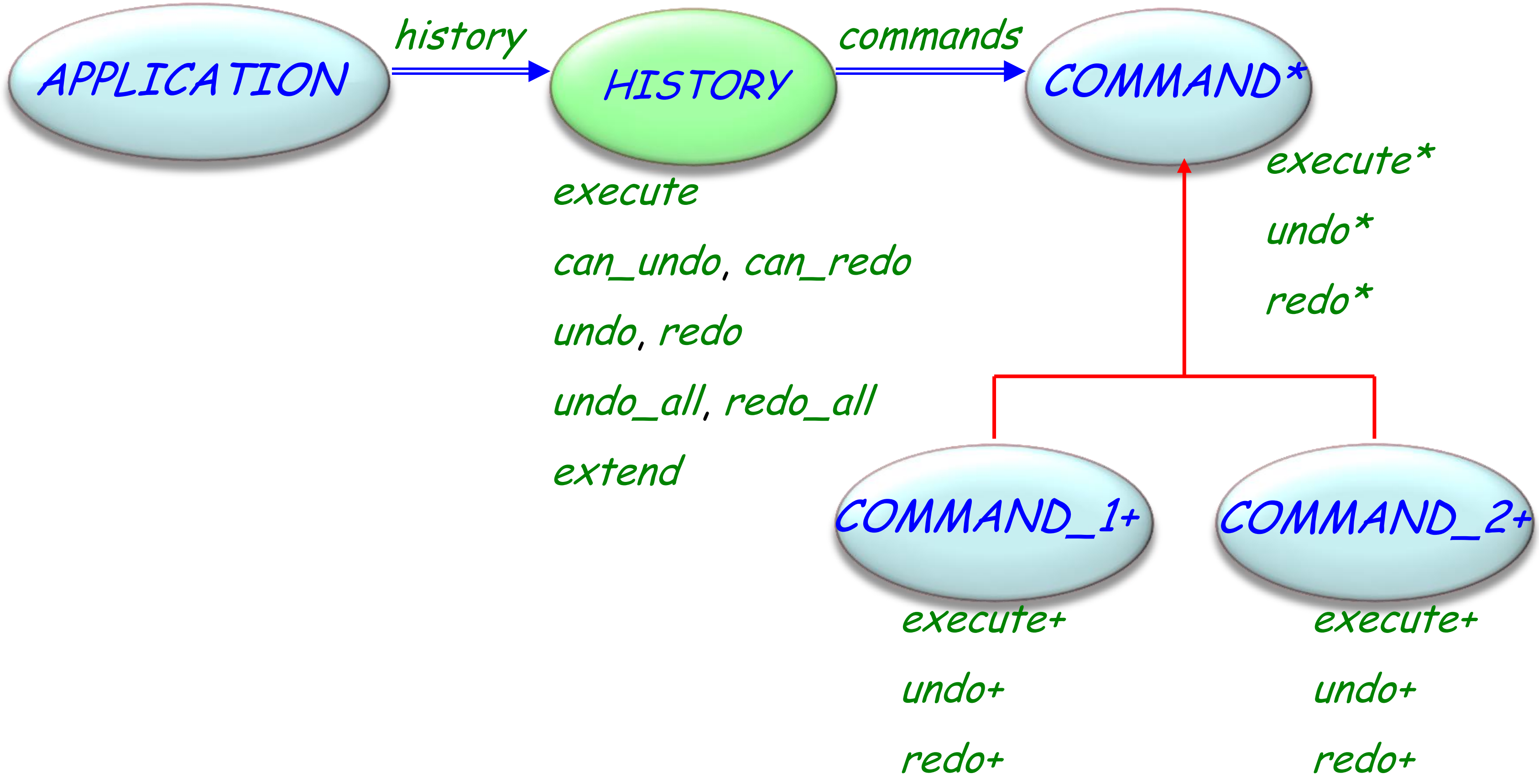
Executing a user command

```
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      -- Ignore excessive requests
            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
```

Pseudocode, see implementation next



Command pattern: original architecture (GoF)



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

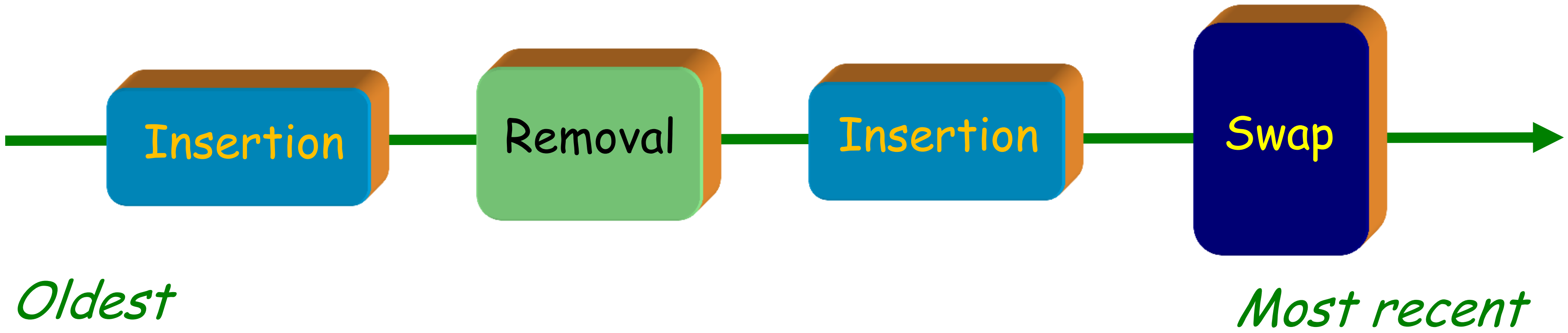
Using agents

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*]

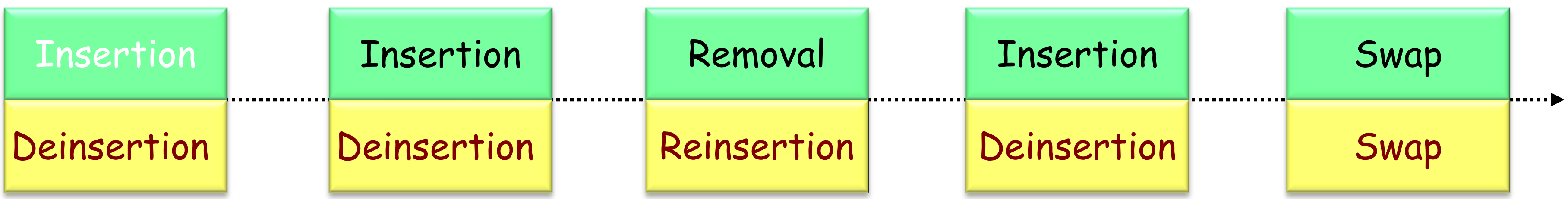


The history list using agents

The history list simply becomes a list of agents pairs:

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

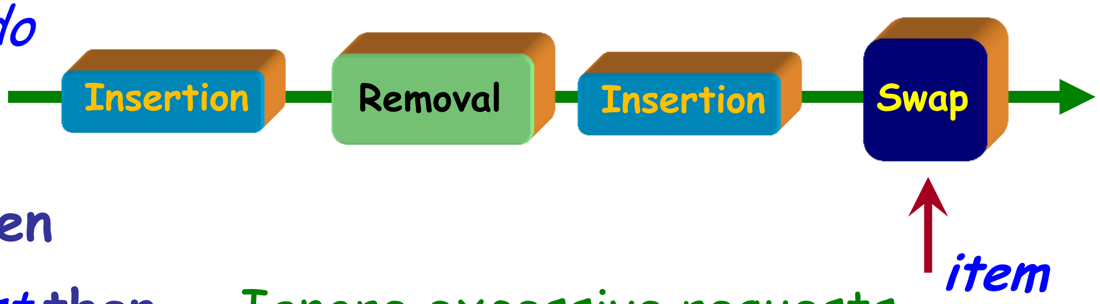
Named tuple



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

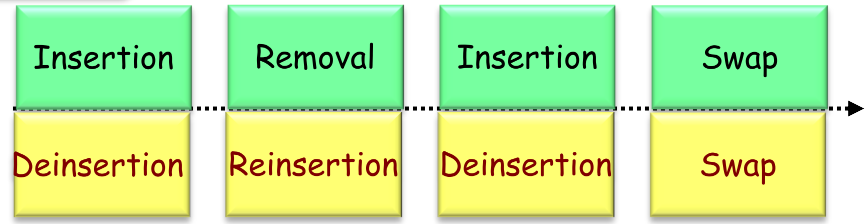
Executing a user command (before)

```
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        -- Ignore excessive requests
            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
```



Executing a user command (now)

```
"Decode user_request giving two agents do_it and undo_it"
if "Request is normal command" then
        history.extend ([do_it, undo_it ])
        do_it.call ([])
elseif "Request is UNDO" then
        if not history.before then
                history.item.undoer .call ([])
                history.back
        end
elseif "Request is REDO" then
        if not history.is_last then
                history.forth
                history.item.doer .call ([])
        end
end
```



Command - Consequences

Command decouples the object that invokes the operation from the one that knows how to perform it.

Commands are first-class objects. They can be manipulated and extended like any other object.

You can assemble commands into a composite command.

It is easy to add new Commands, because you do not have to change existing classes.

Command - Participants

Command

declares an interface for executing an operation.

Concrete command

- defines a binding between a Receiver object and an action.
- implements Execute by invoking the corresponding operation(s) on Receiver.

Client

creates a ConcreteCommand object and sets its receiver.

Invoker

asks the command to carry out the request.

Receiver

knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Design patterns – Pattern categories

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Bridge pattern

Intent:

"Decouple[s] an abstraction from its implementation so that the two can vary."

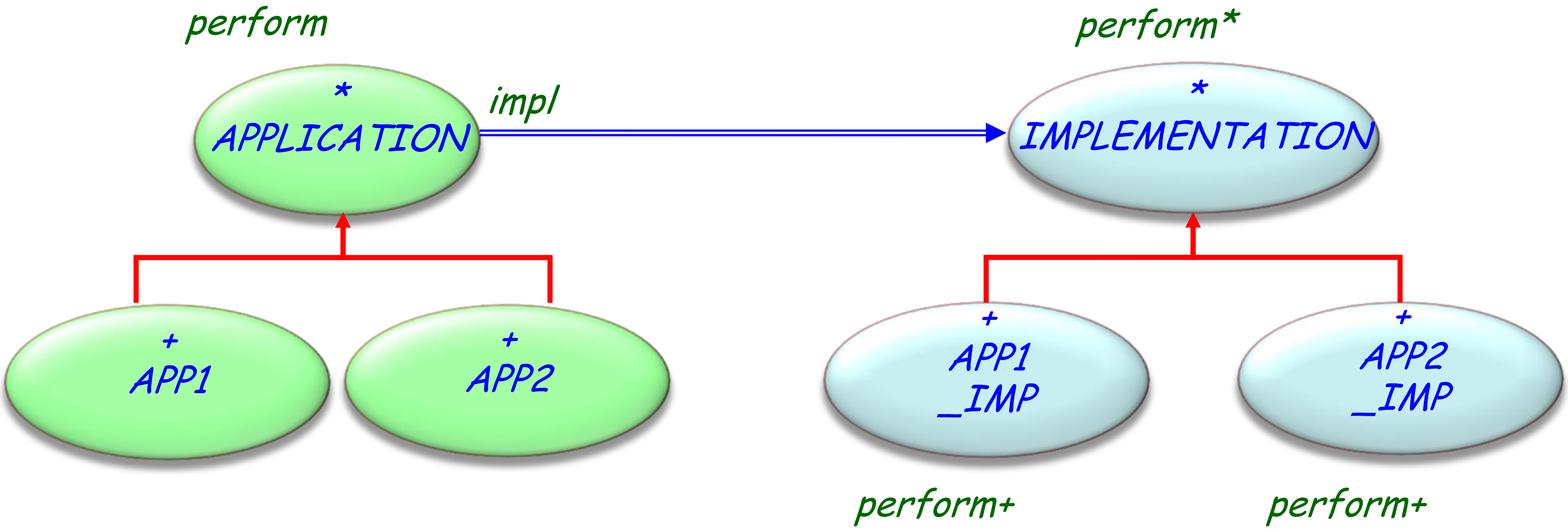
In other words:

It separates the class interface (visible to the clients) from the implementation (that may change later)

Bridge: an example

- EiffelVision 2 library: multi-platform GUI library
- Supports wide range of interaction "widgets" (or "controls")
- Must run under various environments, including Windows and Unix/Linux/VMS (X Windows system)
- Must conform to local look-and-feel of every platform

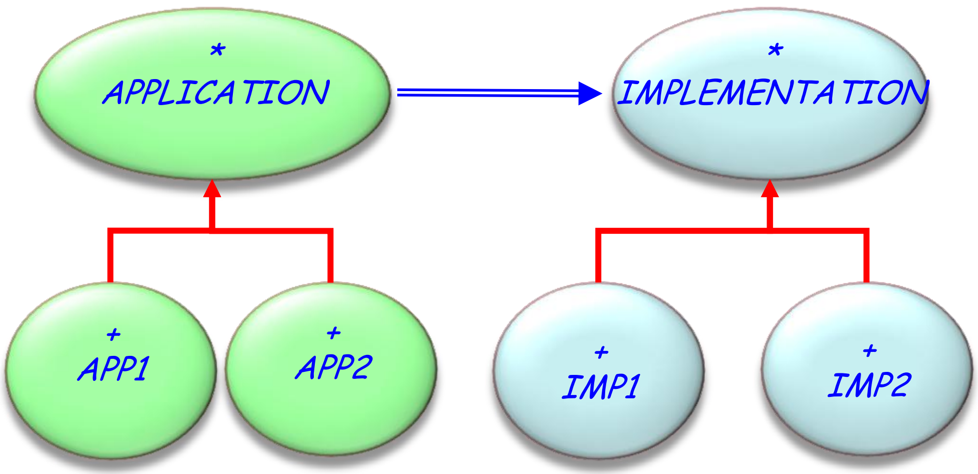
Bridge: Original pattern



Bridge: Classes

```
deferred class
    APPLICATION
feature {NONE} -- Initialization
    make (i: like impl)
            -- Set i as
implementation.
        do impl := i end

feature {NONE} -- Implementation
    impl: IMPLEMENTATION
        -- Implementation

feature -- Basic operations
    perform
    -- Perform desired operation.
        do impl.perform end
end
```

```
deferred class IMPLEMENTATION

feature -- Basic operations
    perform
            -- Perform basic operation.
        deferred end
end
```



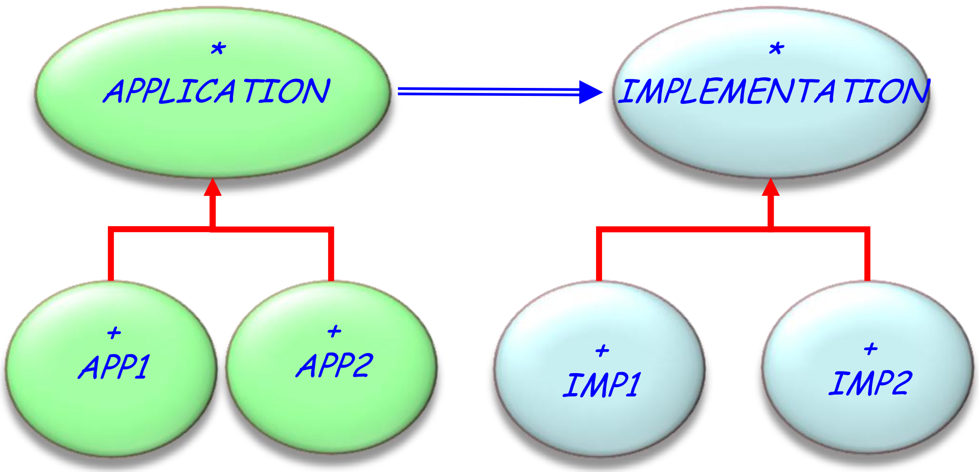
Bridge: Classes

class *APP1* **inherit** *APPLICATION* **create**

make

...

end



class *IMP1* **inherit** *IMPLEMENTATION* **feature**

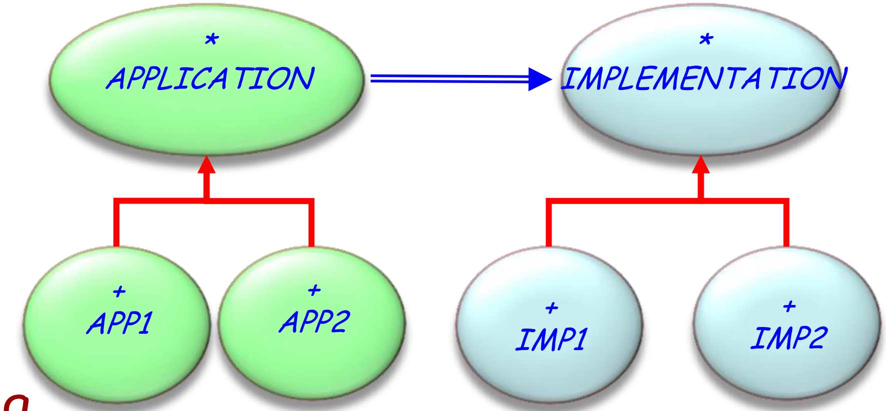
perform

-- Perform desired operation.

do ... **end**

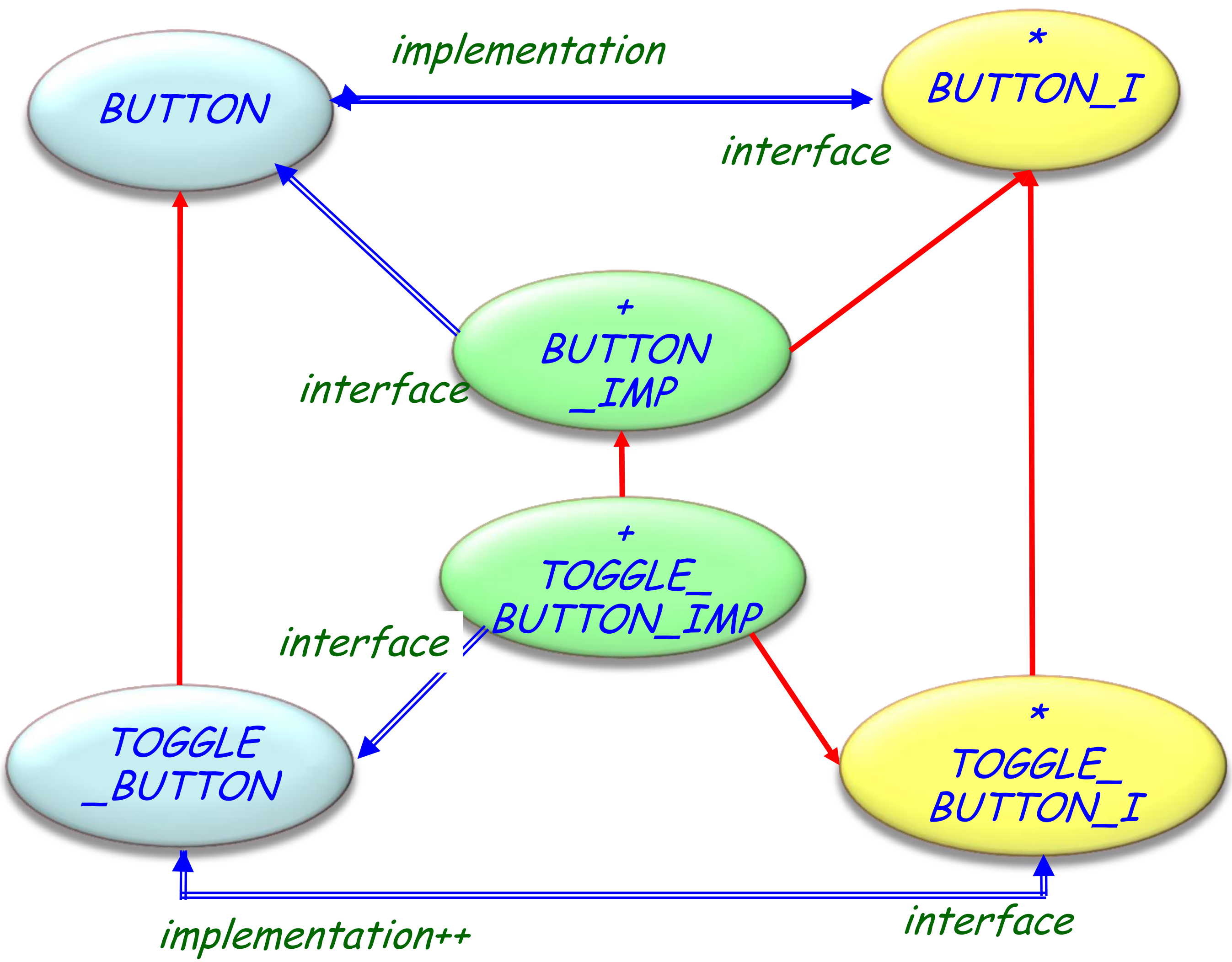
end

Bridge: Client view




```
class CLIENT create
        make

feature -- Basic operations
        make
                        -- Do something.
                local
                        app1: APP1
                        app2: APP2
                do
                        create app1.make (create {IMP1})
                        app1.perform
                        create app2.make (create {IMP2})
                        app2.perform
                end
end
```

Bridge: A variation used in EiffelVision 2



Bridge: EiffelVision 2 example

```
class
        BUTTON

feature {ANY, ANY_I} -- Implementation

        implementation: BUTTON_I -- Implementation

feature {NONE} -- Implementation

        create_implementation
                        -- Create corresponding button implementation.
            do
                create {BUTTON_IMP} implementation.make (Current)
            end

...
end
```

Bridge: Advantages (or when to use it)

- No permanent binding between abstraction and implementation
- Abstraction and implementation extendible by subclassing
- Implementation changes have no impact on clients
- Implementation of an abstraction completely hidden from clients
- Implementation share with several objects, hidden from clients

Bridge: Componentization

- Non-componentizable (no library support)

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- **Composite**
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

Non-GoF patterns

- ✓ Model-View-Controller

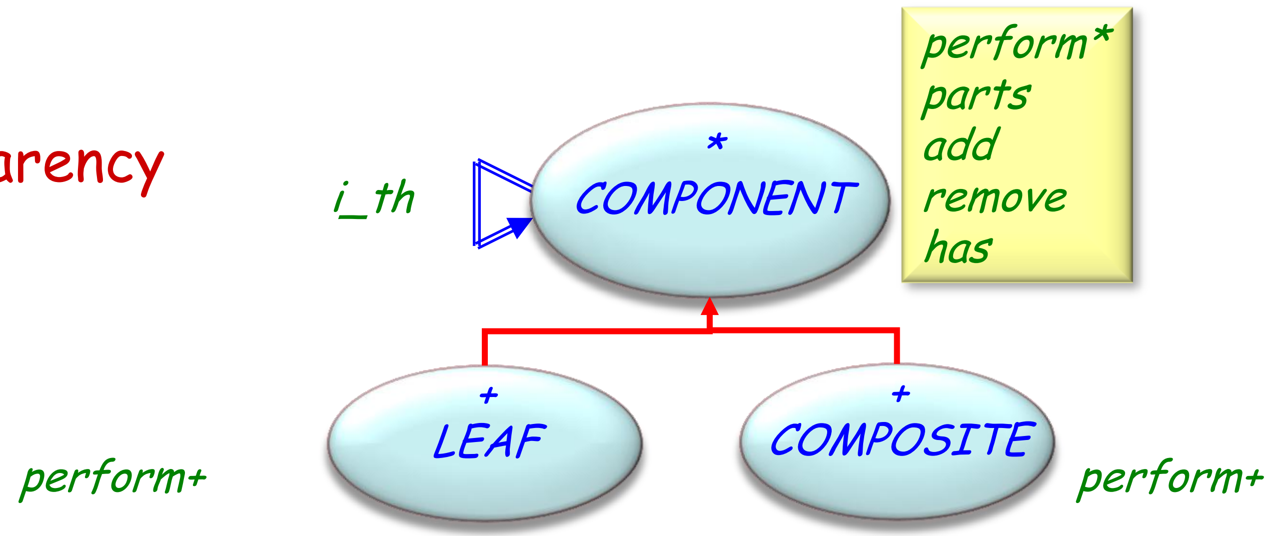
Composite pattern

Intent:

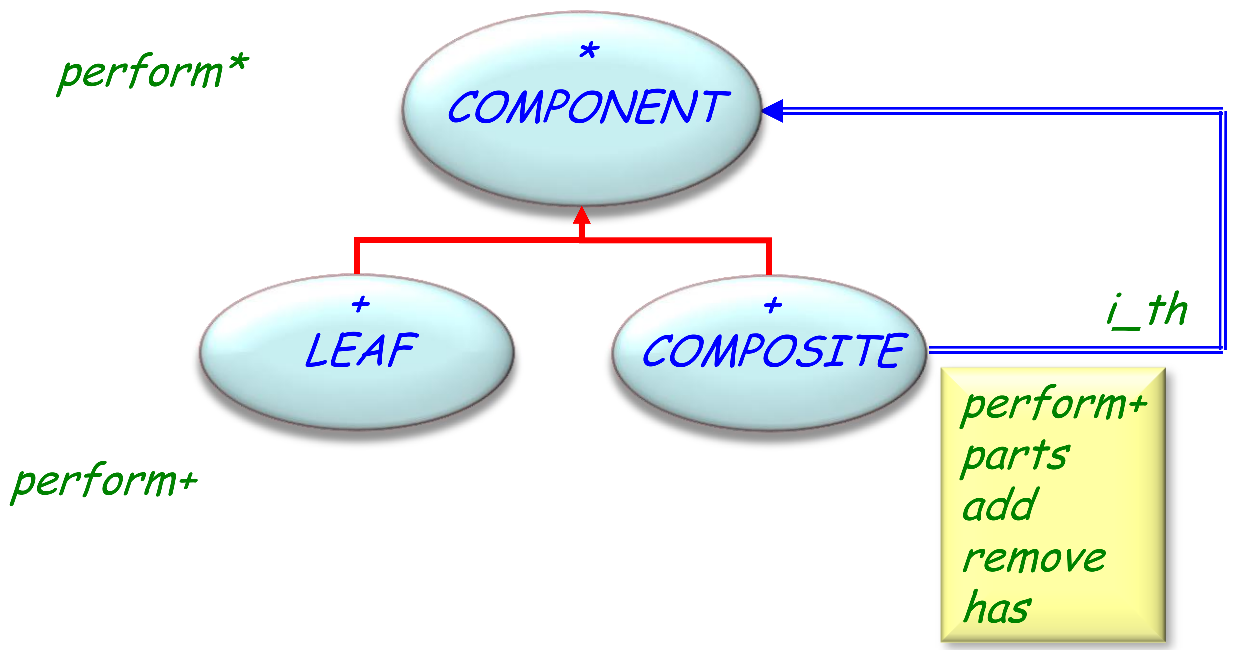
"Way to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."

Composite: Original pattern

Transparency version



Safety version



Composite pattern, safety version (1/5)

```
deferred class
        COMPONENT

feature -- Basic operation
        perform
                    -- Do something.
                deferred
                end

feature -- Status report
        is_composite: BOOLEAN
                    -- Is component a composite?
                do
                    Result := False
                end
end
```

Composite pattern, safety version (2/5)

```
class
        COMPOSITE
inherit
        COMPONENT
                redefine
                        is_composite
                end
create
        make,
        make_from_components

feature {NONE} -- Initialization
        make
                        -- Initialize component parts.
                do
                        create parts.make
                end
```

Composite pattern, safety version (3/5)

```
make_from_components (part_list: like parts)
            -- Initialize from part_list.
        require
            parts_not_void: part_list /= Void
            no_void_component: not some_components.has (Void)
        do
            parts := part_list
        ensure
            parts_set: parts = part_list
        end

feature -- Status report
    is_composite: BOOLEAN
            -- Is component a composite?
        do
            Result := True
        end
```

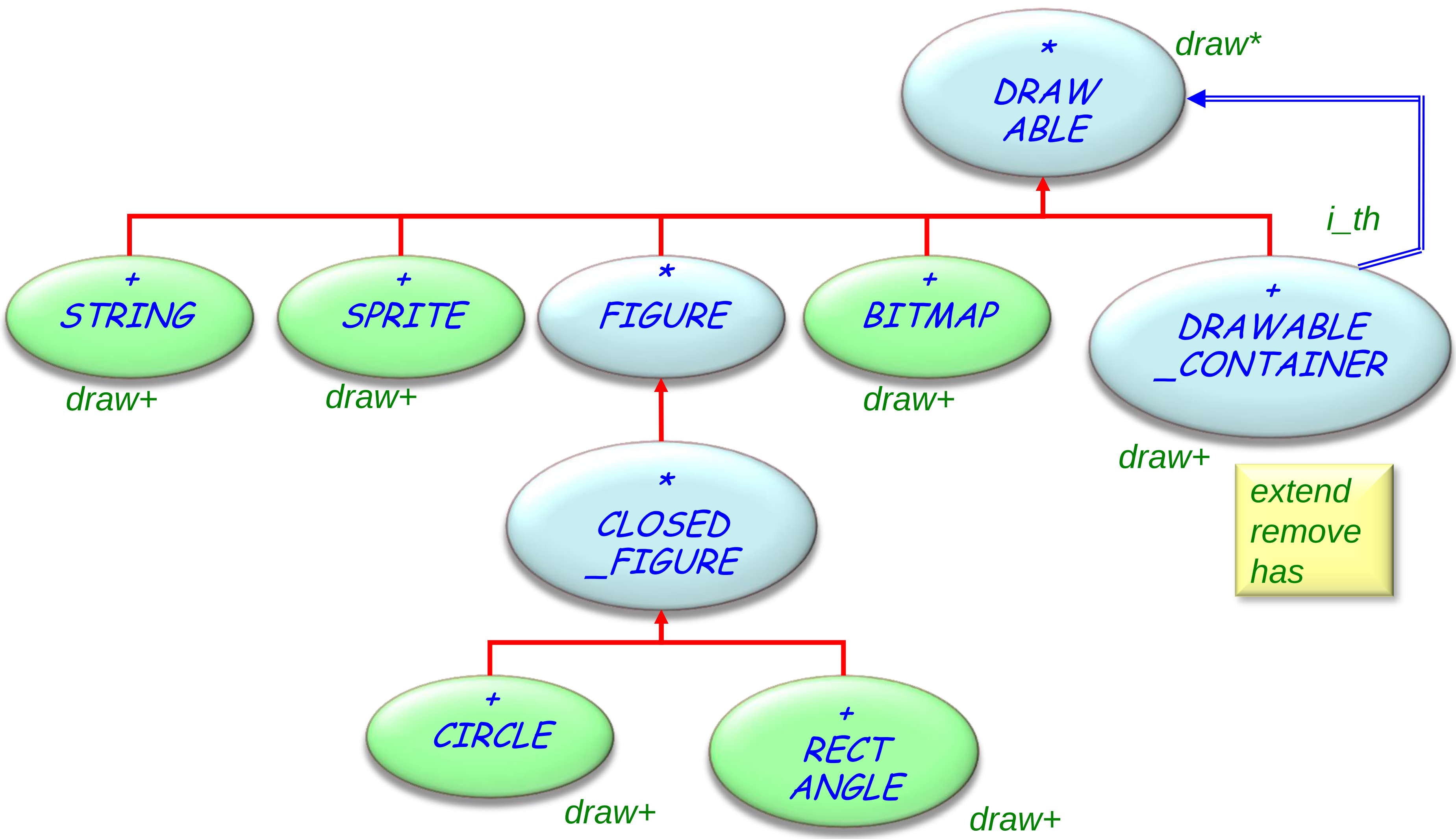
Composite pattern, safety version (4/5)

```
feature -- Basic operation
        perform
                        -- Performed desired operation on all components.
                do
                        from parts.start until parts.after loop
                                parts.item.perform
                                parts.forth
                        end
                end
feature -- Access
        item: COMPONENT
                        -- Current part of composite
                do
                        Result := parts.item
                ensure
                        definition: Result = parts.item
                        component_not_void: Result /= Void
                end
```

Composite pattern, safety version (5/5)

```
feature -- Others
        -- Access: i_th, first, last
        -- Status report: has, is_empty, off, after, before
        -- Measurement: count
        -- Element change: add
        -- Removal: remove
        -- Cursor movement: start, forth, finish, back

feature {NONE} – Implementation
        parts: LINKED_LIST [like item]
                        -- Component parts
                        -- (which are themselves components)
invariant
        is_composite: is_composite
        parts_not_void: parts /= Void
        no_void_part: not parts.has (Void)
end
```

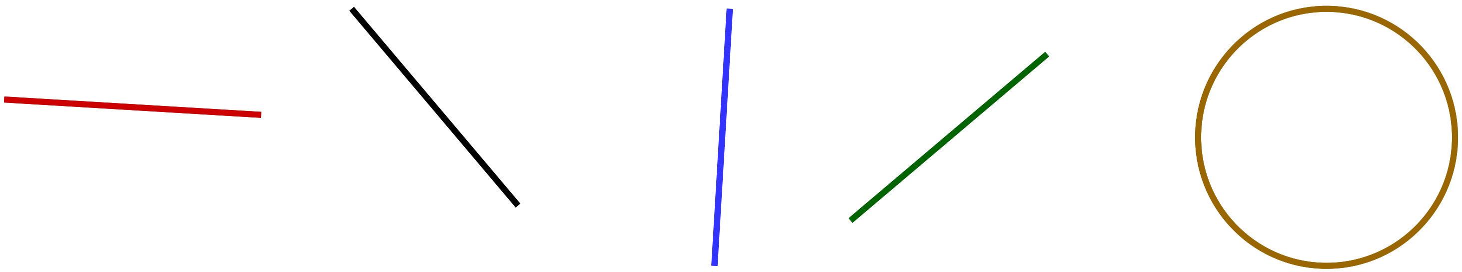
Composite: Variation used in EiffelMedia



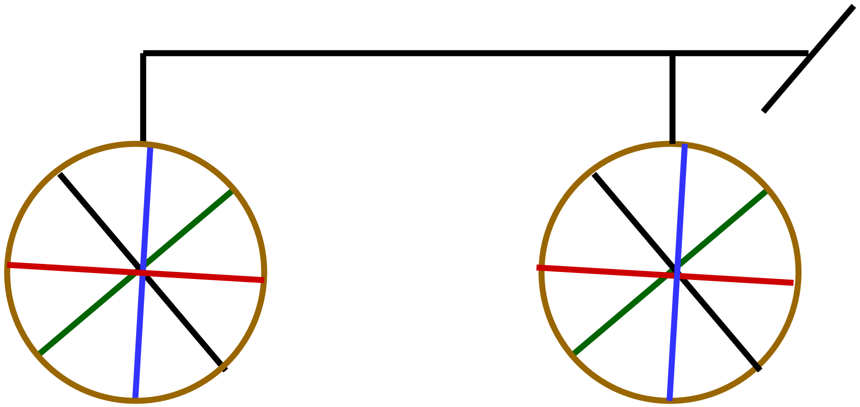
Composite: Advantages (or when to use it)

- Represent part-whole hierarchies
- Clients treat compositions and individual objects uniformly

Figures

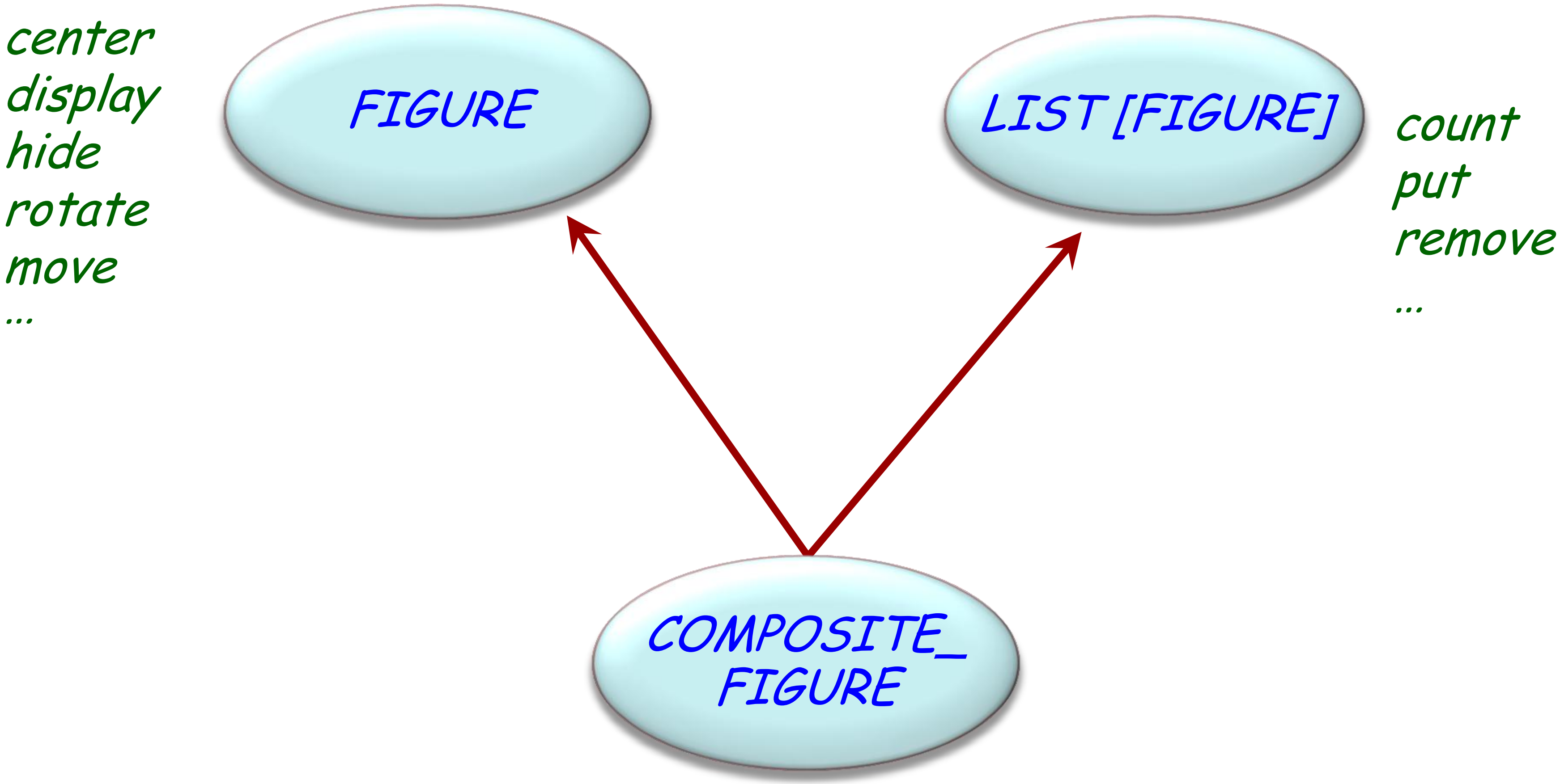


Simple figures

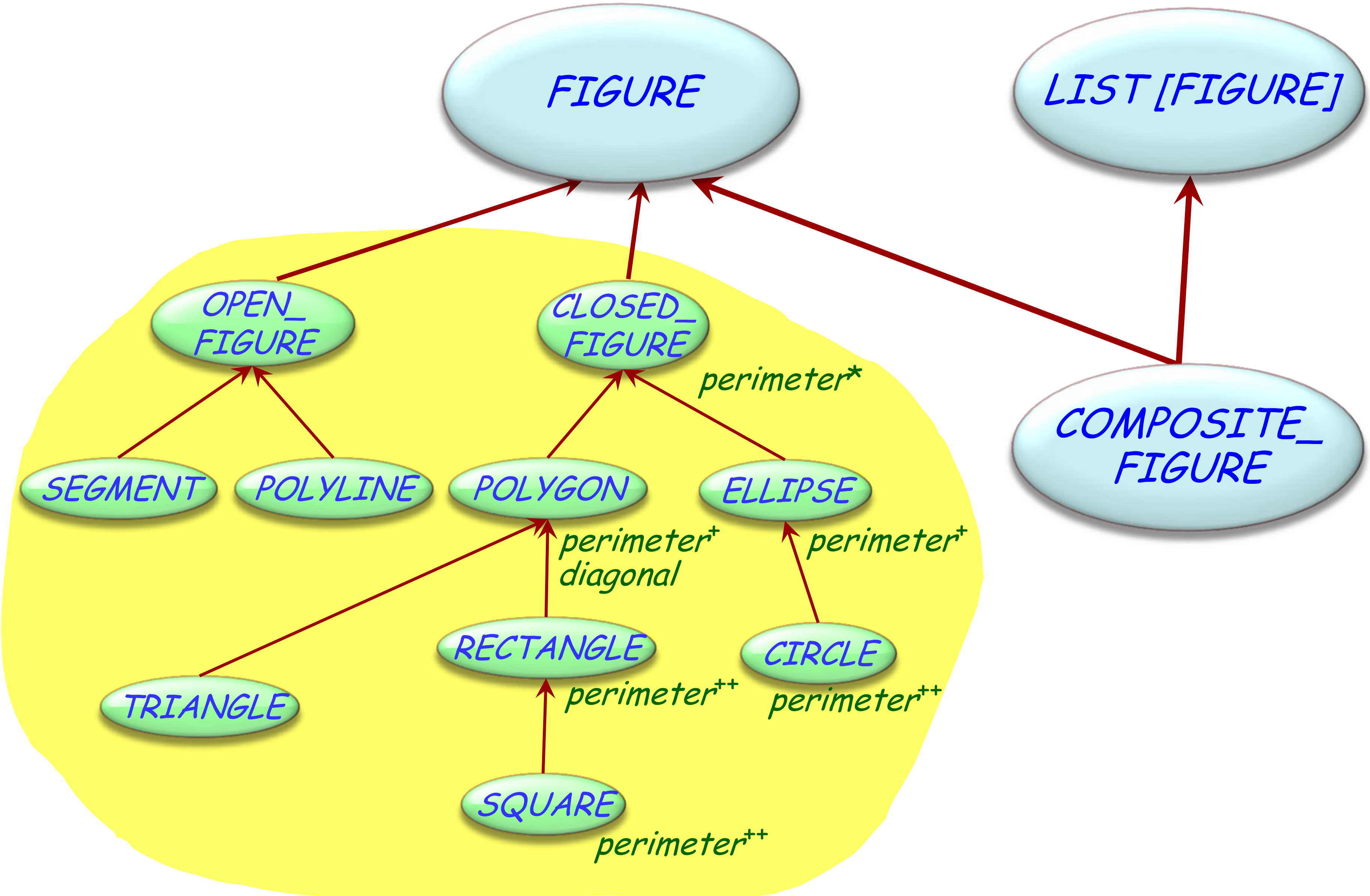


A composite figure

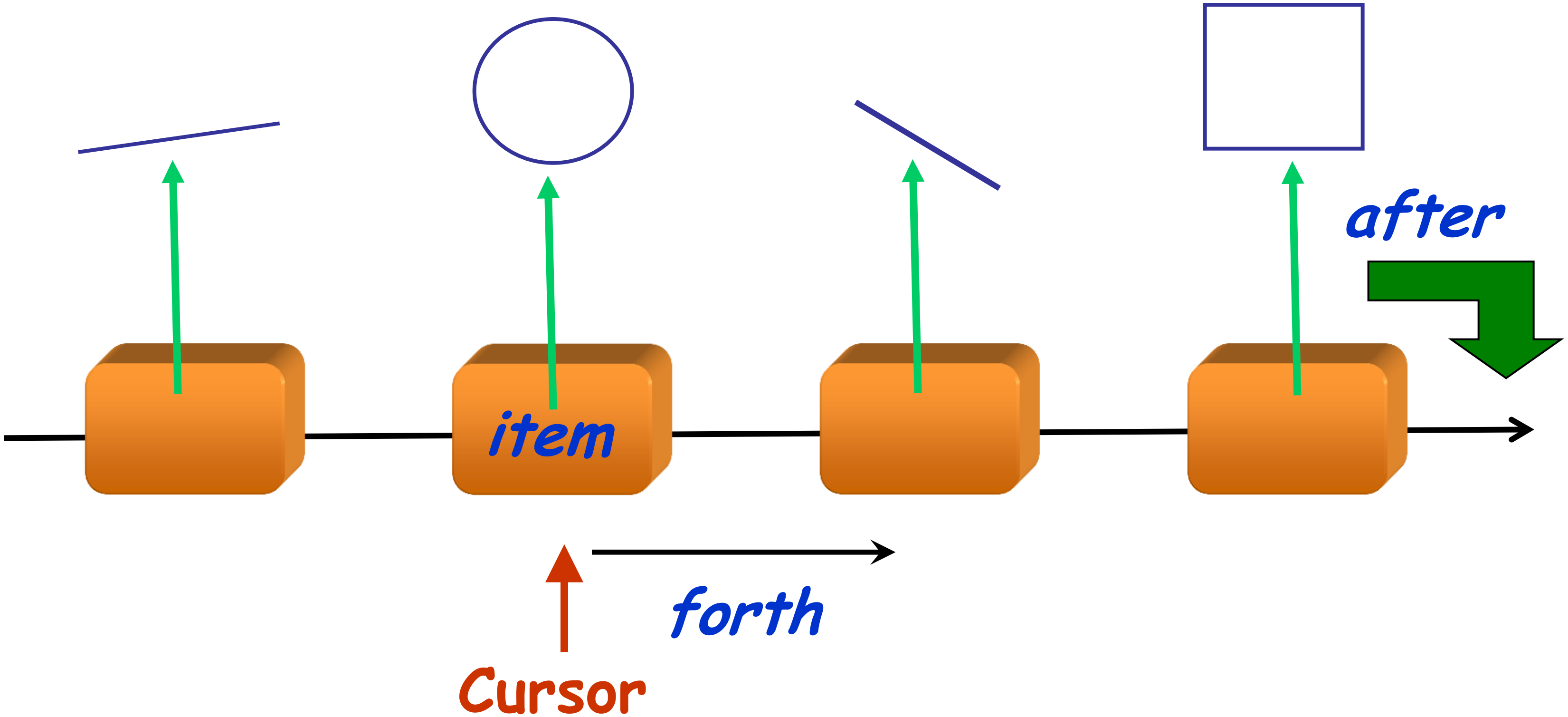
Defining the notion of composite figure



In the overall structure



A composite figure as a list



Composite figures

```
class COMPOSITE_FIGURE inherit
        FIGURE

        LIST [FIGURE]
feature
        display
                -- Display each constituent figure in turn.
        do
                from start until after loop

                        item.display

                        forth
                end
        end
        ... Similarly for move, rotate etc. ...
end
```

Requires dynamic binding

Composite: Componentization

- Fully componentizable
- Library support
- Main idea: Use genericity

- But: the library version lacks flexibility and makes the structure difficult to understand

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- **Decorator**
- Façade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

Non-GoF patterns

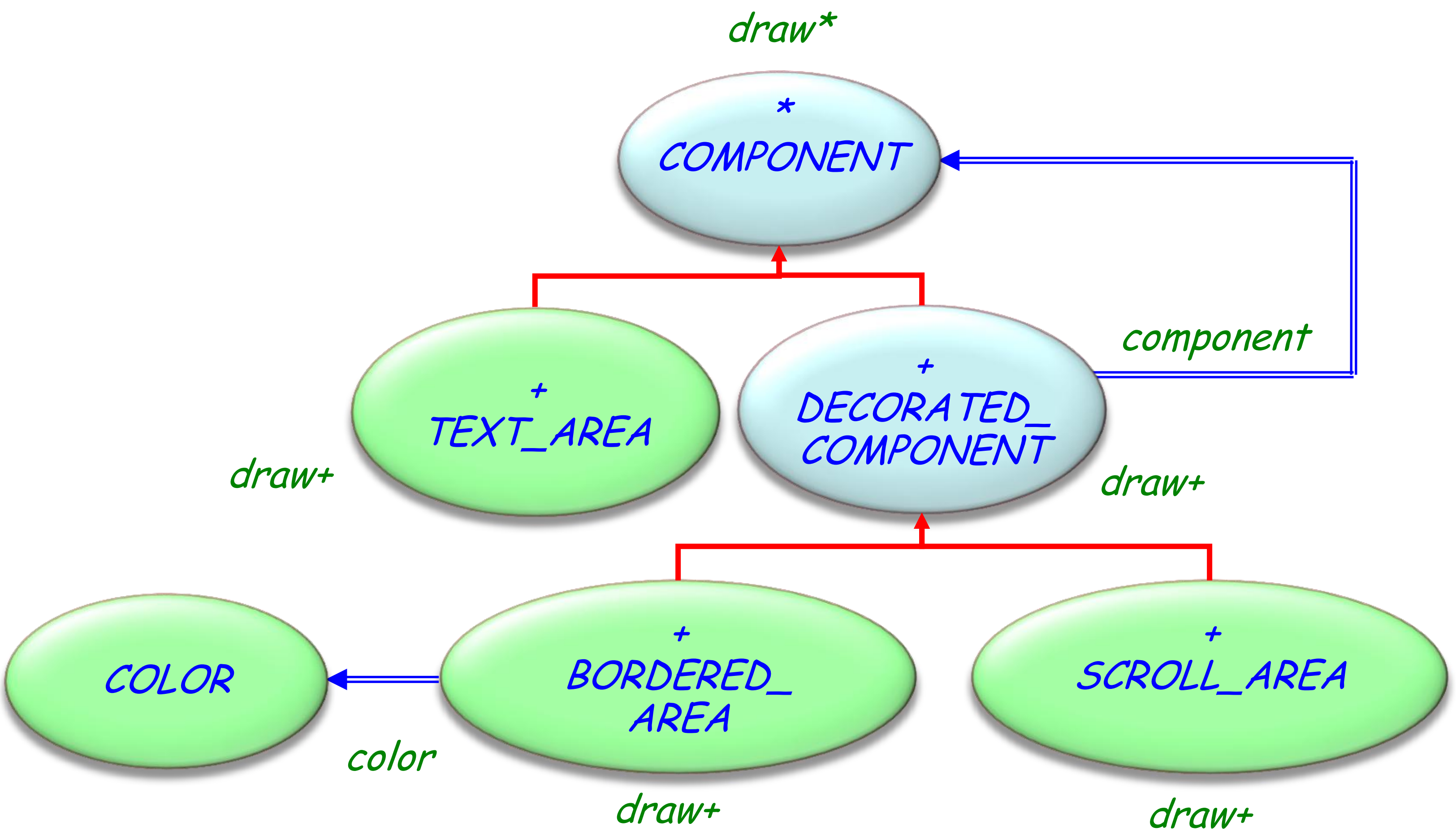
- ✓ Model-View-Controller

Decorator pattern

Intent:

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

Decorator: Example



Decorator: example

Display an area with a border of a certain color

```
class
    BORDERED_AREA
inherit
    DECORATED_COMPONENT
...
feature
    color: COLOR
    set_color (c: like color) ...

    draw
        do
            draw_border (color)
            component.draw
        end
end
```

Decorator: Exporting additional features?

Newly introduced features do not need to be visible to clients, but they may.

e.g. Display an area with a border of a certain color

```
class
    BORDERED_AREA
inherit
    DECORATED_COMPONENT
...
feature
    color: COLOR
    set_color (c: like color) ...

    draw
            do
                    draw_border (color)
                    component.draw
            end
end
```

Client can change the *color* by calling *set_color* if it has direct access to the *BORDERED_AREA*

Decorator: Advantages (or when to use it)

- Add responsibilities to individual objects dynamically and transparently
- Responsibilities can be withdrawn
- Avoid explosion of subclasses to support combinations of responsibilities

Decorator: Componentization

- Non-componentizable
- Skeleton classes can be generated

Decorator skeleton, attribute (1/2)

```
note
        description: "Skeleton of a component decorated with additional attributes"
class
        DECORATED_COMPONENT -- You may want to change the class name.
inherit
        COMPONENT -- You may need to change the class name
                redefine
                        -- List all features of COMPONENT that are not deferred.
                end
create
        make
        -- You may want to add creation procedures to initialize the additional attributes.
feature {NONE} -- Initialization
        make (a_component : like component)
                        -- Set component to a_component.
                require
                        a_component_not_void: a_component /= Void
                do
                        component := a_component
                ensure
                        component_set: component = a_component
                end
-- List additional creation procedures taking into account additional attributes.
```

Decorator skeleton, attribute (2/2)

```
feature -- Access
        -- List additional attributes.

feature -- To be completed
      -- List all features from COMPONENT and implement them by
      -- delegating calls to component as follows:
      -- do
      --        component.feature_from_component
      -- end

feature {NONE} -- Implementation
      component: COMPONENT
                  -- Component that will be used decorated
invariant
      component_not_void: component /= Void
end
```

Decorator skeleton, behavior (1/2)

```
note
        description: "Skeleton of a component decorated with additional behavior"
class
        DECORATED_COMPONENT -- You may want to change the class name.
inherit
        COMPONENT -- You may need to change the class name
                redefine
                        -- List all features of COMPONENT that are not deferred.
                end
create
        make
feature {NONE} -- Initialization
        make (a_component: like component)
                        -- Set component to a_component.
                require
                        a_component_not_void: a_component /= Void
                do
                        component := a_component
                ensure
                        component_set: component = a_component
                end
```

Decorator skeleton, behavior (2/2)

```
feature -- To be completed
        -- List all features from COMPONENT and implement them by
        -- delegating calls to component as follows:
        -- do
        --         component.feature_from_component
        -- end

        -- For some of these features, you may want to do something more:
        -- do
        --         component.feature_from_component
        --         perform_more
        -- end

feature {NONE} -- Implementation
        component: COMPONENT
                        -- Component that will be used for the "decoration"
invariant
        component_not_void: component /= Void
end
```

Decorator skeleton: Limitations

```
feature -- To be completed
        -- List all features from COMPONENT and implement them by
        -- delegating calls to component as follows:
        -- do
        --      component.feature_from_component
        -- end
```

Does not work if *feature_from_component* is:

- an attribute: cannot redefine an attribute into a function (Discussed at ECMA)
- a frozen feature (rare): cannot be redefined, but typically:
  - Feature whose behavior does not need to be redefined (e.g. *standard_equal*, … from *ANY*)
  - Feature defined in terms of another feature, which can be redefined (e.g. *clone* defined in terms of *copy*)

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

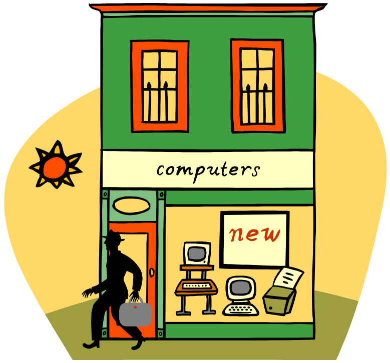
Non-GoF patterns

- ✓ Model-View-Controller

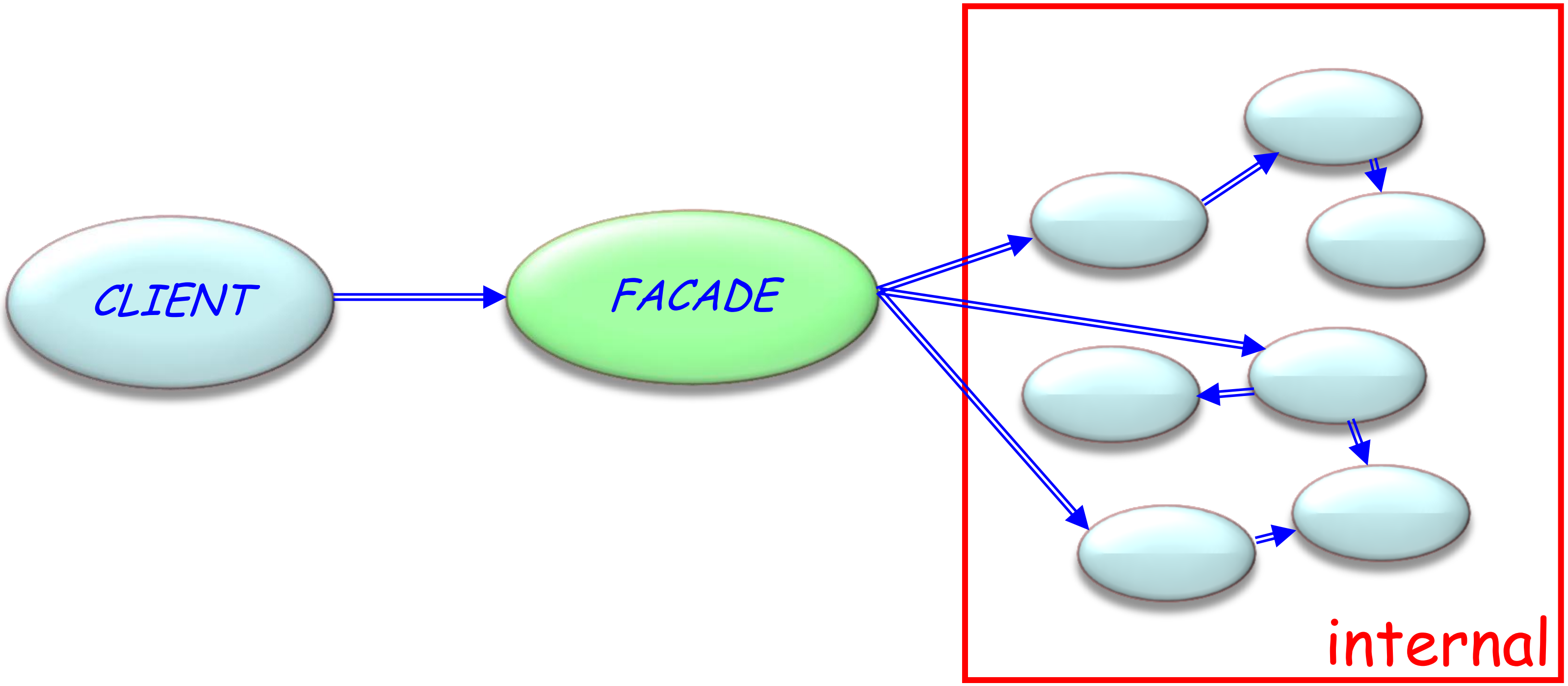
Façade

Intent:

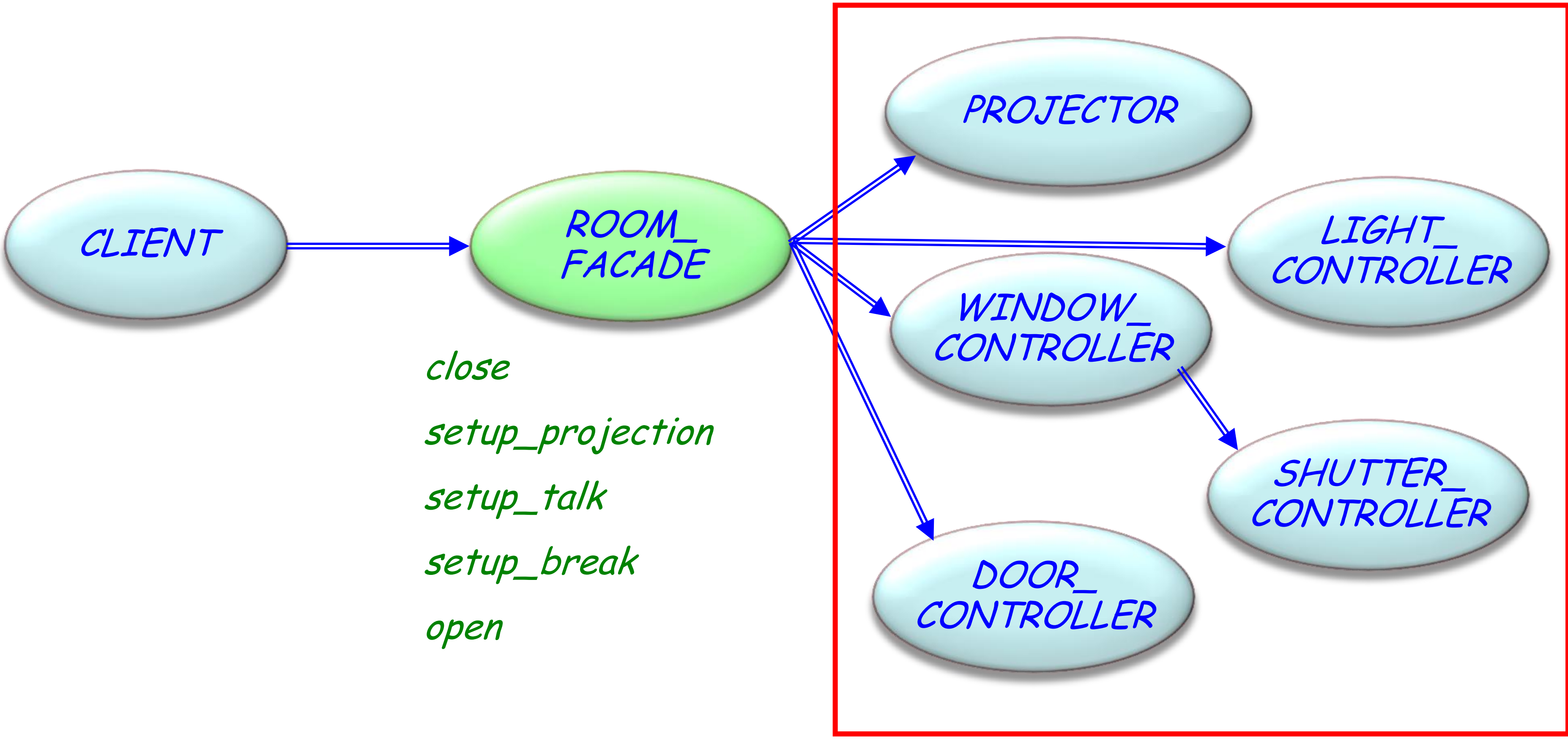
"*Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.*" [GoF, p 185]



Façade: Original pattern



Façade: Example



Other example: Compiler, where clients should not need to know about all internally used classes.

Façade: Advantages (or when to use it)

- Provides a simple interface to a complex subsystem
- Decouples clients from the subsystem and fosters portability
- Can be used to layer subsystems by using façades to define entry points for each subsystem level

Façade: Componentization

- Non-componentizable

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- **Flyweight**
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Flyweight pattern

Intent:

"*Use sharing to support large numbers of fine-grained objects efficiently.*"

Without the Flyweight pattern (1/2)

```
class
        CLIENT
...
feature -- Basic operation
        draw_lines
                -- Draw some lines in color.
            local
                line1, line2: LINE
                red: INTEGER
            do
                ...
                create line1.make (red, 100, 200)
                line1.draw
                create line2.make (red, 100, 400)
                line2.draw
                ...
            end
...
end
```

Creates one *LINE* object for each line to draw

Without the Flyweight pattern (2/2)

```
class interface
        LINE
create
        make
feature -- Initialization
        make (c, x, y: INTEGER)
                        -- Set color to c, x as x_position, and y as y_position.
                ensure
                        color_set: color = c
                        x_set: x_position = x
                        y_set: y_position = y
feature -- Access
        color: INTEGER
                        -- Line color
        x_position, y_position: INTEGER
                        -- Line position
feature -- Basic operation
        draw
                        -- Draw line at position (x_position, y_position) with color.
end
```

With the Flyweight pattern (1/3)

```
class
        CLIENT
feature -- Basic operation
        draw_lines
                        -- Draw some lines in color.
                local
                        line_factory: LINE_FACTORY
                        red_line: LINE
                        red: INTEGER
                do
                        ...
                        red_line := line_factory.new_line (red)
                        red_line.draw (100, 200)
                        red_line.draw (100, 400)
                        ...
                end
...
end
```

Creates only one *LINE* object per color

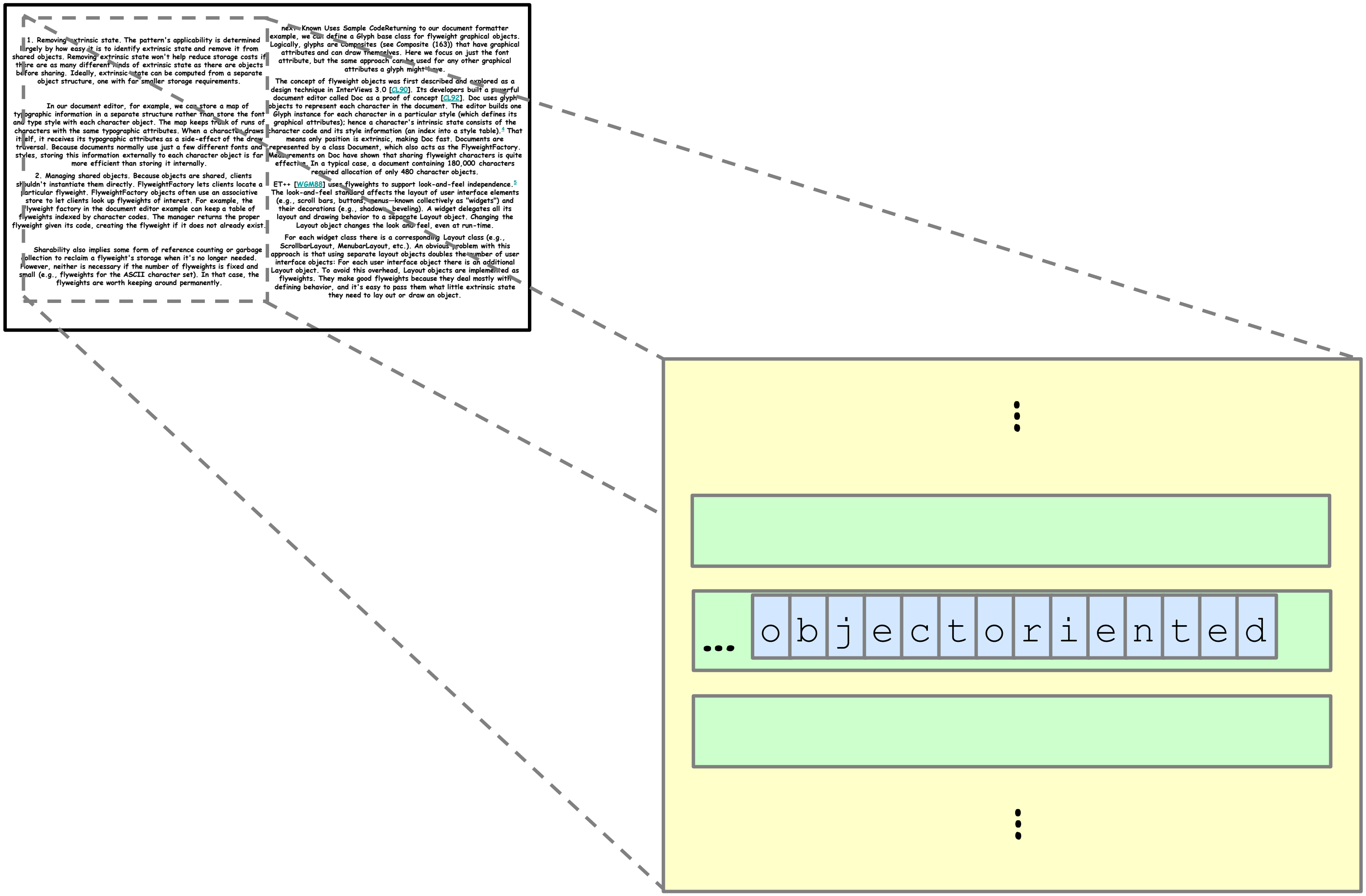
With the Flyweight pattern (2/3)

```
class interface
        LINE_FACTORY

feature -- Initialization
        new_line (c: INTEGER): LINE
                        -- New line with color c
                ensure
                        new_line_not_void: Result /= Void
...
end
```

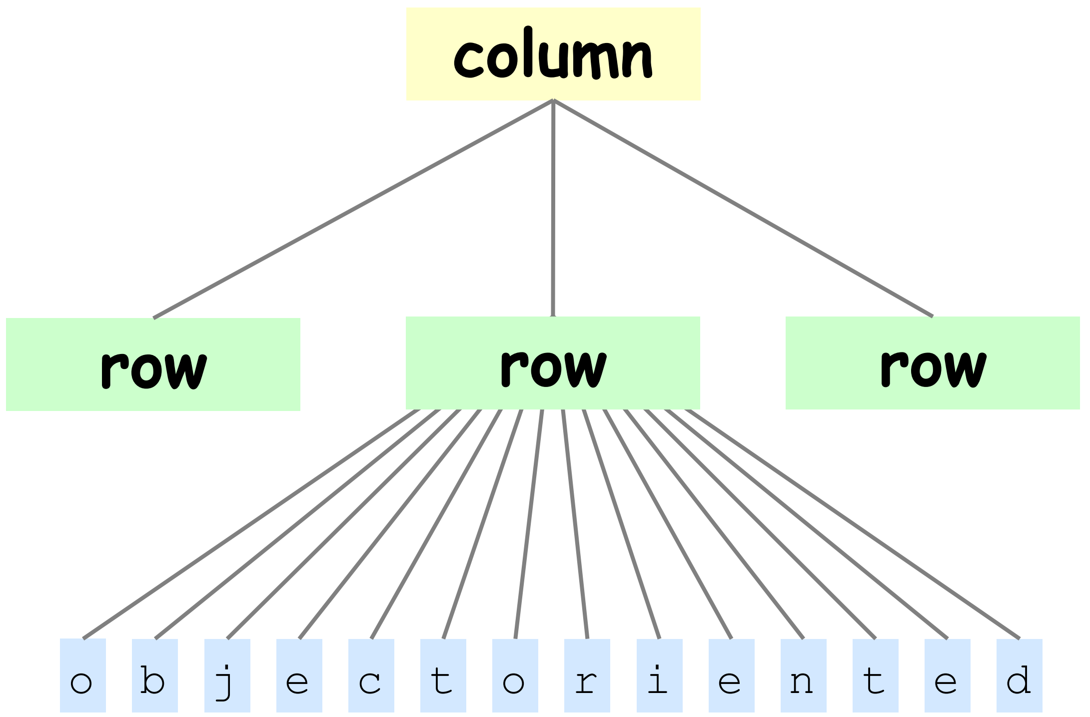
With the Flyweight pattern (3/3)

```
class interface
        LINE
create
        make
feature -- Initialization
        make (c: INTEGER)
                        -- Set color to c.
                ensure
                        color_set: color = c
feature -- Access
        color: INTEGER
                        -- Line color
feature -- Basic operation
        draw (x, y: INTEGER)
                        -- Draw line at position (x, y) with color.
end
```

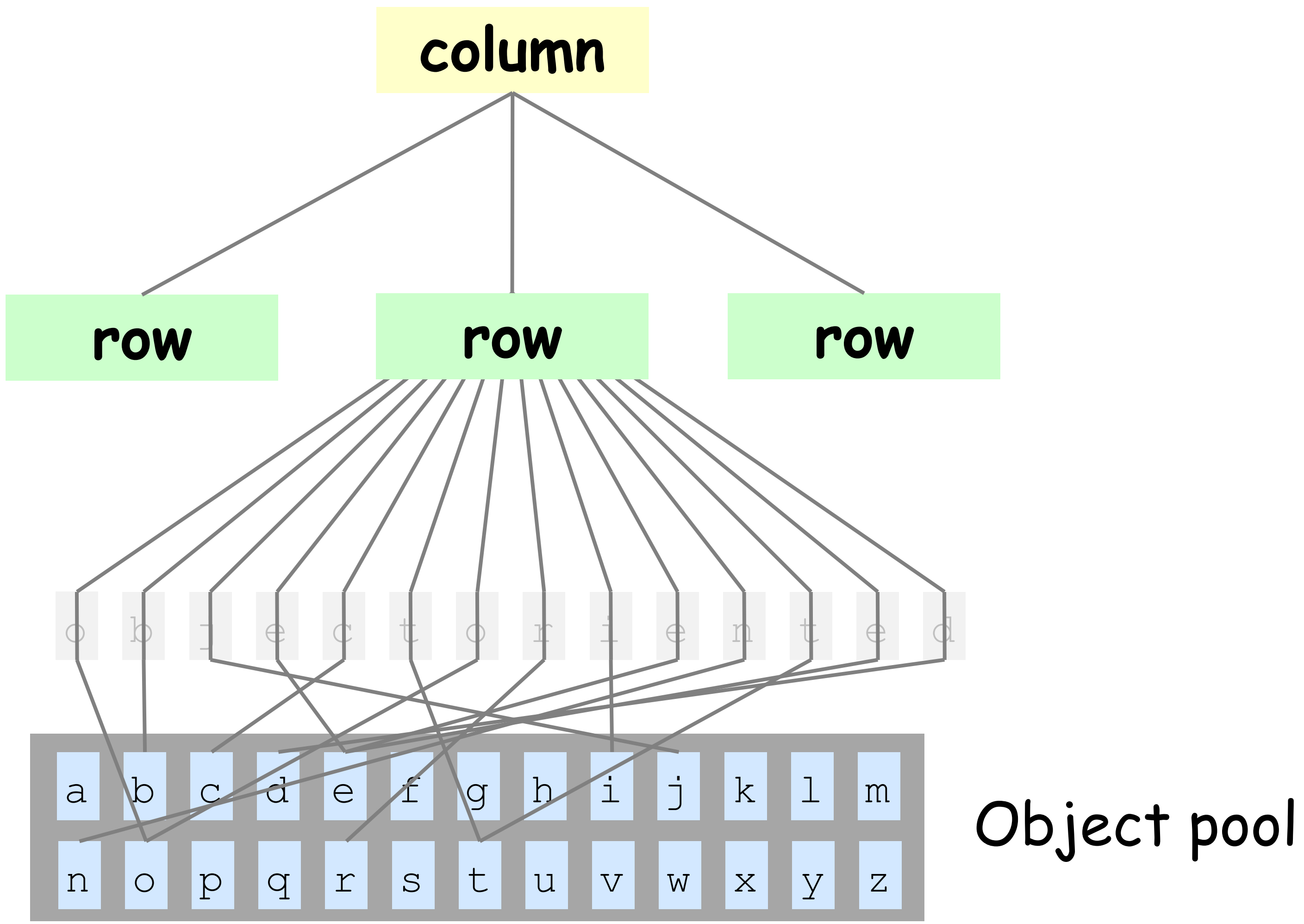
Another example: Document processing



Object structure without flyweight



Object structure with flyweight



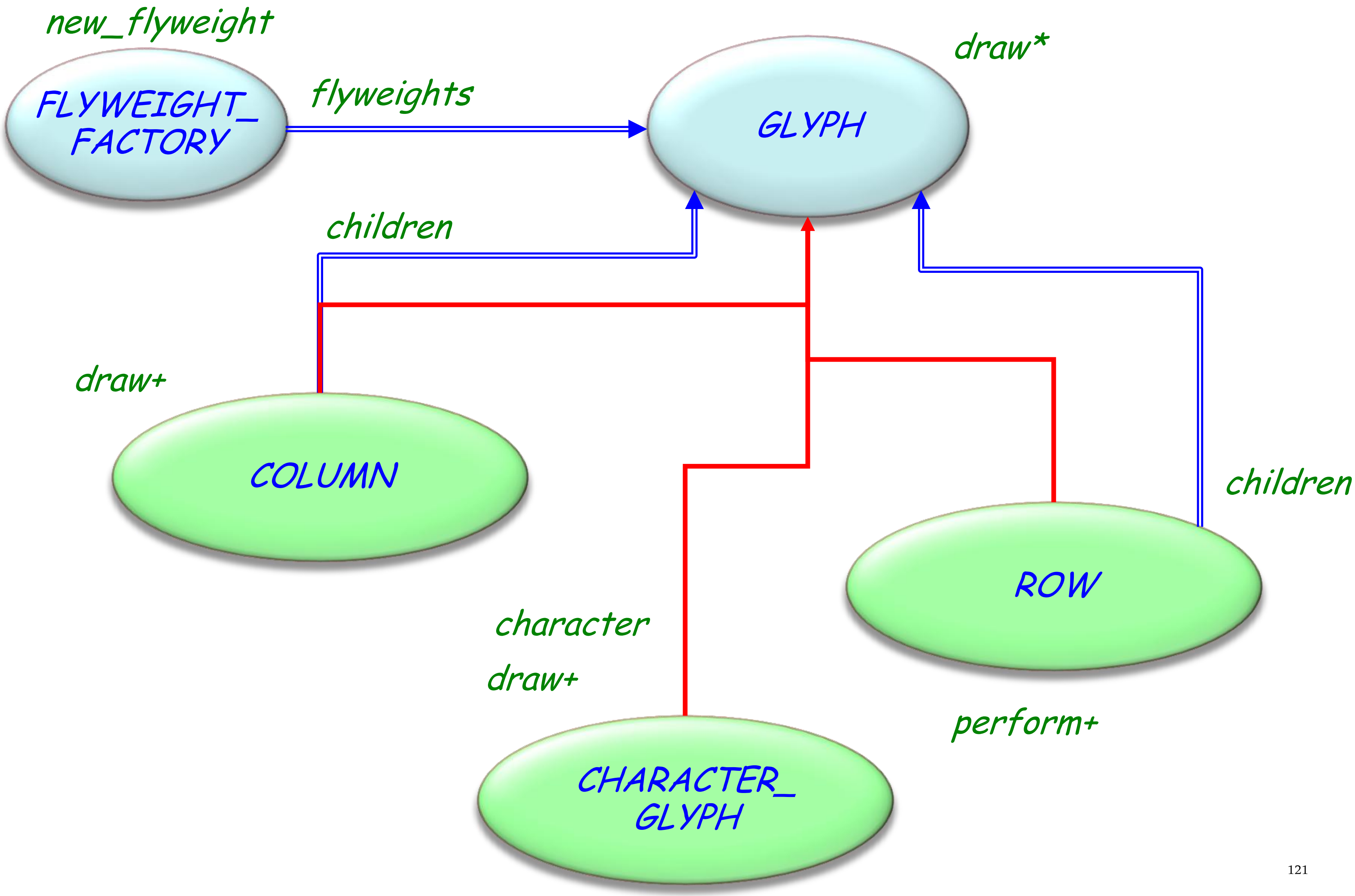
Text processing

- In document processing system: one flyweight per character code

- Other properties, such as font, position in document etc. are stored in client.

- Basic distinction:
  - **Intrinsic** properties of state: stored in flyweight
  - "**Extrinsic**" properties: stored in "context" for each use.

Text processing class hierarchy



Shared/unshared and (non-)composite objects

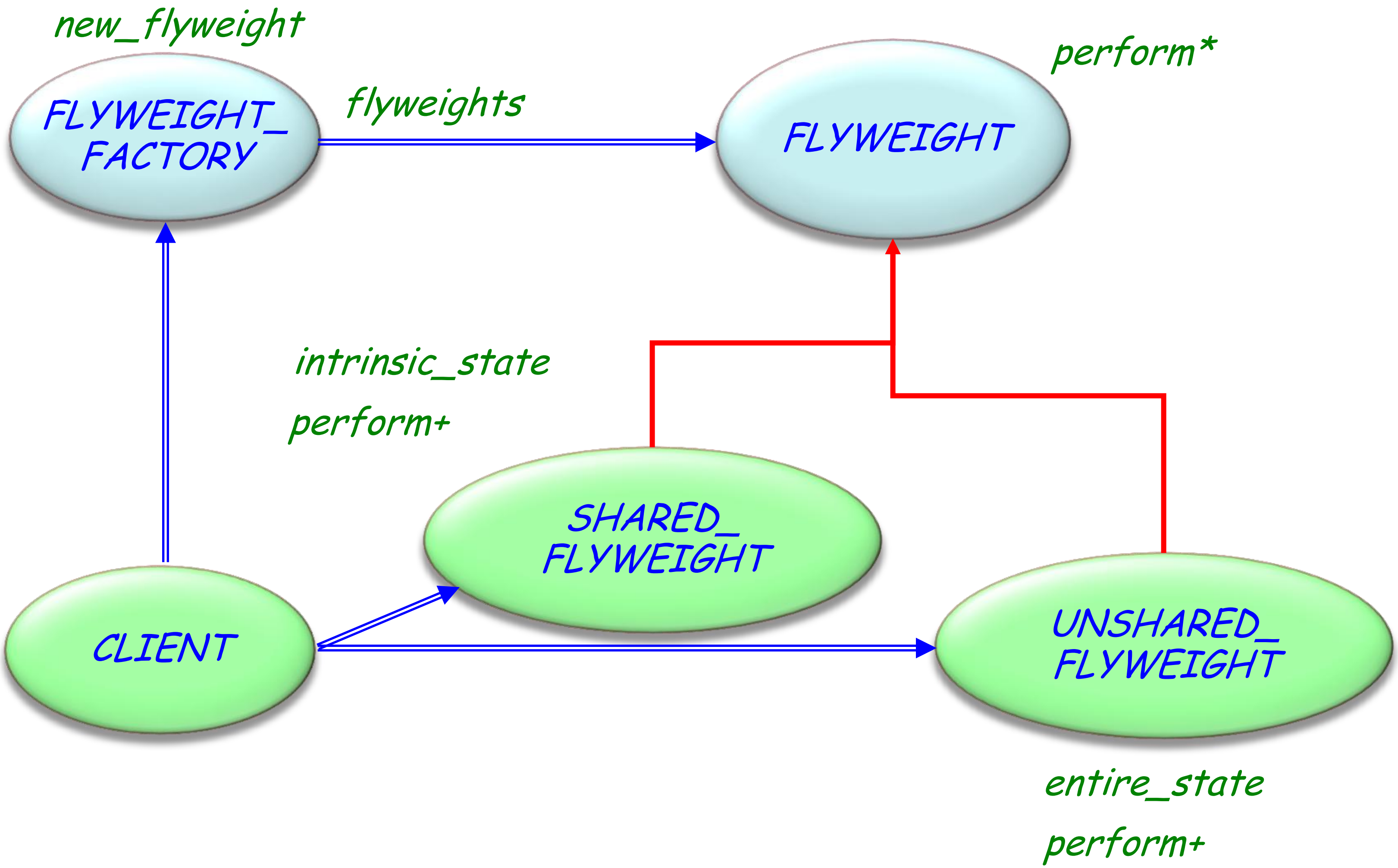
The color of the *LINE*

Two kinds of property:

- Intrinsic characteristics stored in the flyweight
- Extrinsic characteristics moved to the client (typically a "flyweight context")

The coordinates of the *LINE*

Flyweight: Original pattern



Flyweight pattern: Description

Intent: "Use sharing to support large numbers of fine-grained objects efficiently."

Participants:

- *FLYWEIGHT*: Offers a service *perform* to which the extrinsic characteristic will be passed
- *SHARED_FLYWEIGHT*: Adds storage for intrinsic characteristic
- *UNSHARED_FLYWEIGHT*: Not all flyweights need to be shared
- *FACTORY*: Creates and manages the flyweight objects
- *CLIENT*: Maintains a reference to flyweight, and computes or stores the extrinsic characteristics of flyweight

Shared/unshared and (non-)composite objects

Two kinds of flyweights:

- Composites (shared or unshared)
- Non-composites (shared)

Flyweight: Advantages (or when to use it)

- If a large number of objects are used, can reduce storage use:
  - By reducing the number of objects by using shared objects
  - By reducing the replication of intrinsic state
  - By computing (rather than storing) extrinsic state

Flyweight: Componentization

- Fully componentizable

- Mechanisms enabling componentization:
  - Constrained genericity, agents
  - Uses Factory Library and Composite Library

- But: Structure is difficult to understand

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Visitor pattern

Intent:

"Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

[Gamma et al., p 331]

- Static class hierarchy
- Need to perform traversal operations on corresponding data structures
- Avoid changing the original class structure

Visitor application examples

Set of classes to deal with an Eiffel or Java program (in EiffelStudio, Eclipse ...)

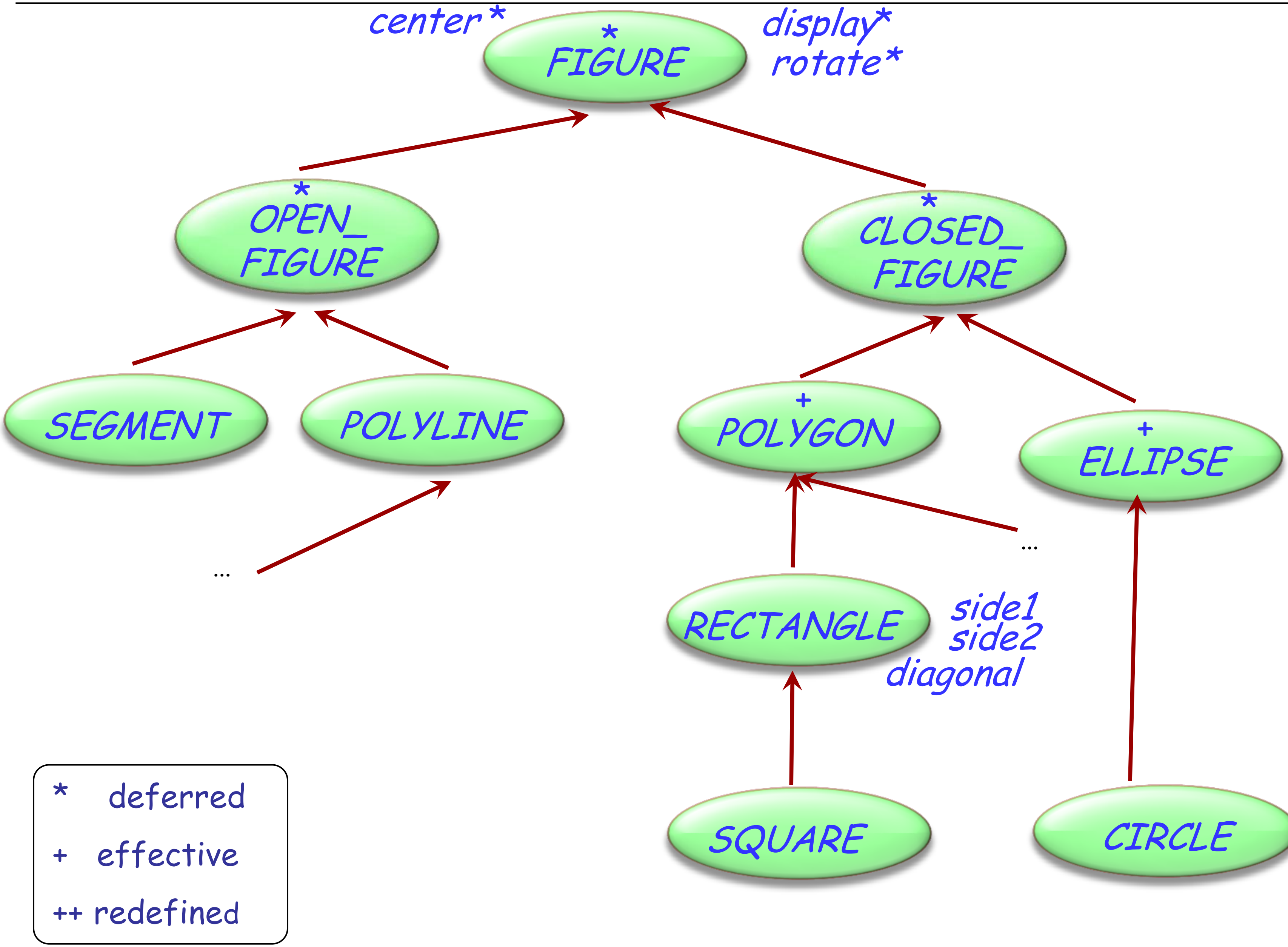
Or: Set of classes to deal with XML documents (*XML_NODE, XML_DOCUMENT, XML_ELEMENT, XML_ATTRIBUTE, XML_CONTENT…)*

One parser (or several: keep comments or not…)

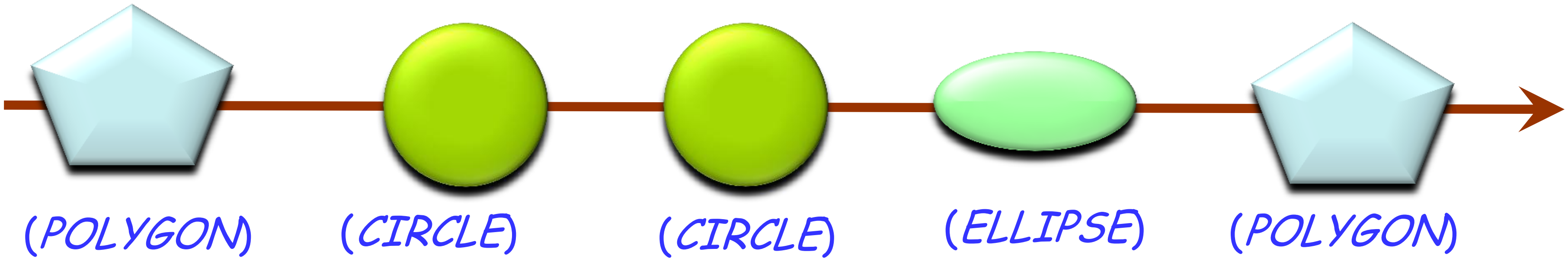
Many formatters:

- Pretty-print
- Compress
- Convert to different encoding
- Generate documentation
- Refactor
- …

Inheritance hierarchy



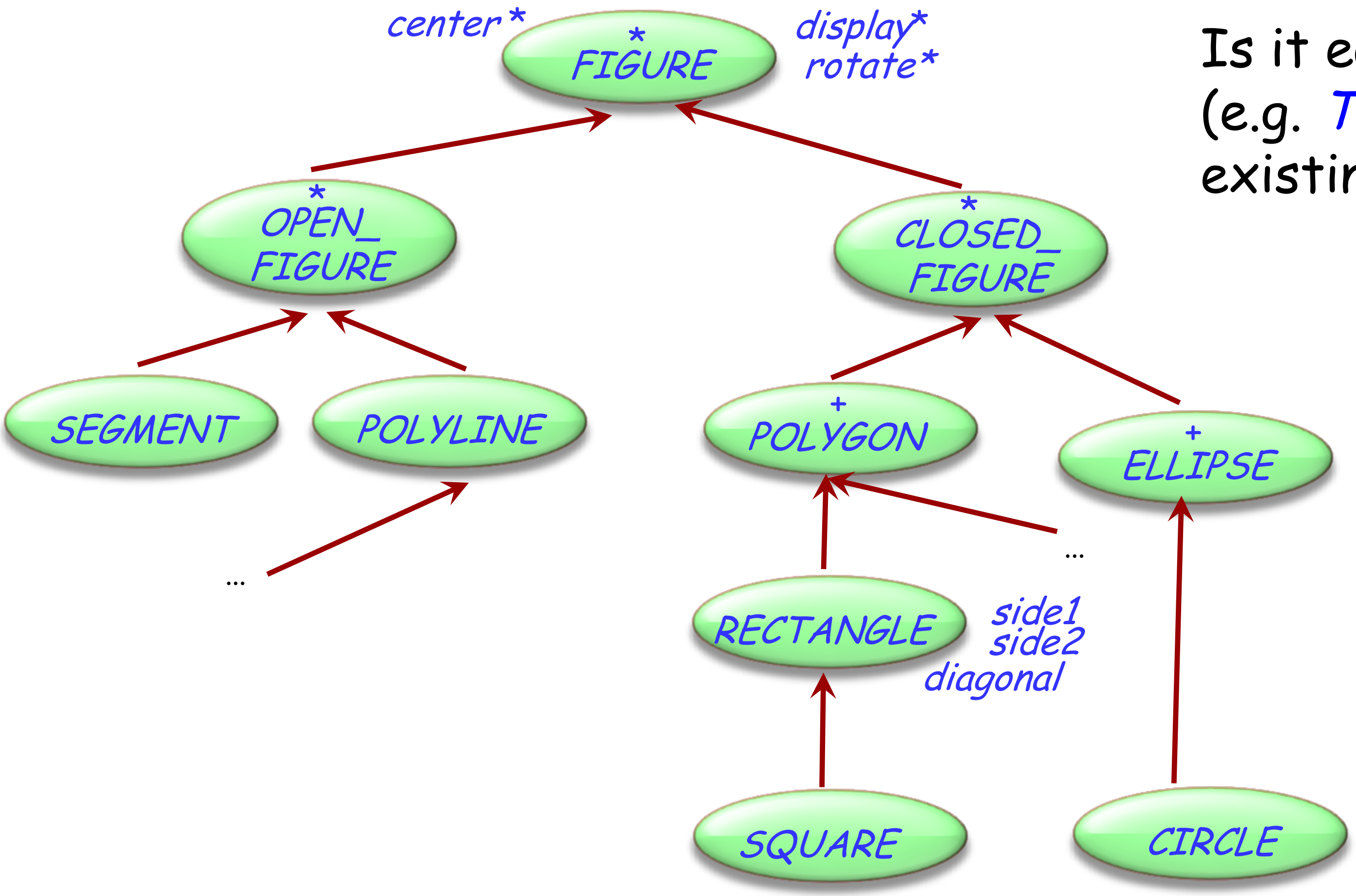
Polymorphic data structures




```
figs : LIST [FIGURE ]
```

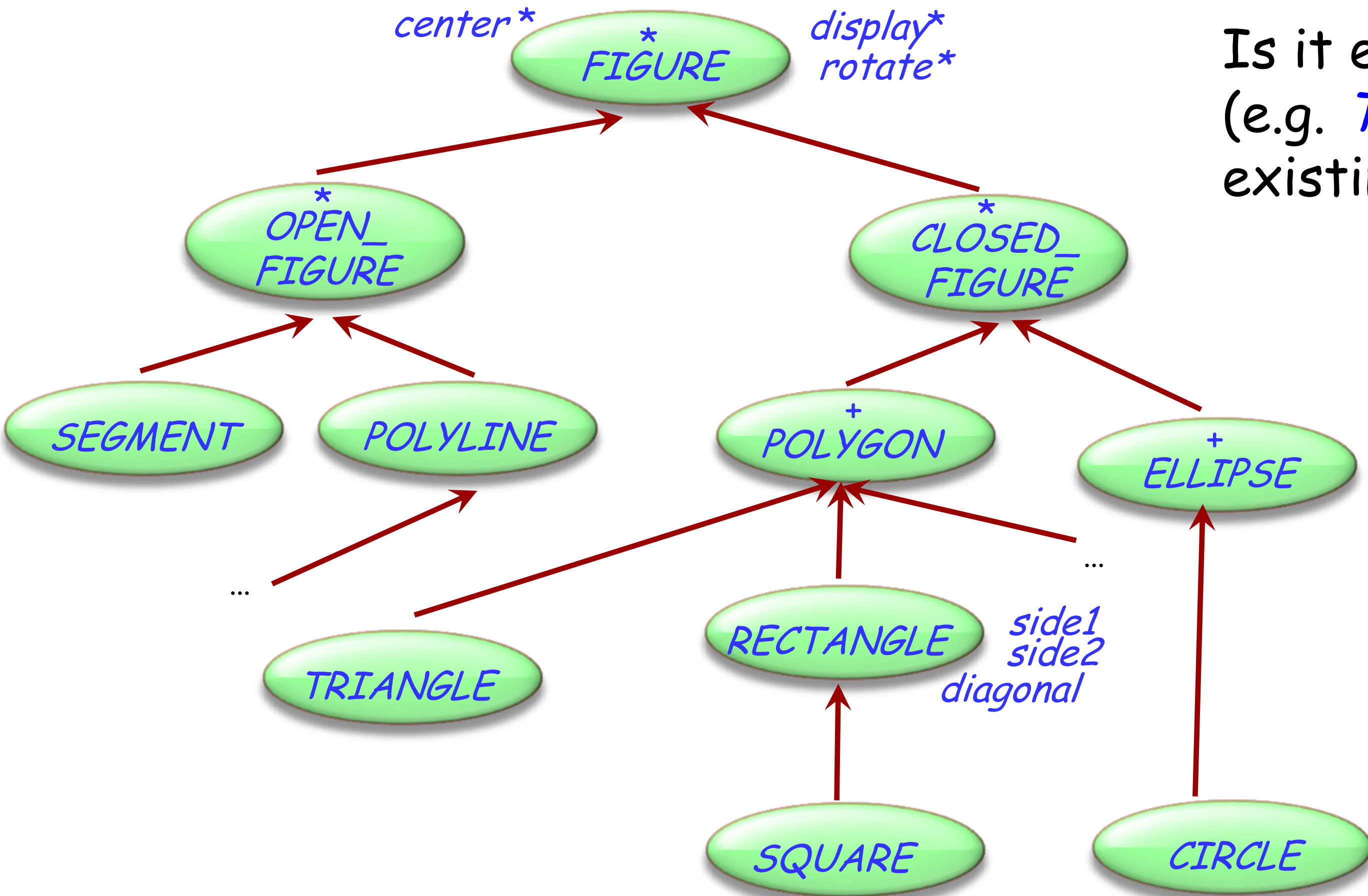
```
from
    figs.start
until
    figs.after
loop
    figs.item .display
    figs.forth
end
```

The dirty secret of O-O architecture



Is it easy to add types (e.g. *TRIANGLE*) to existing operations

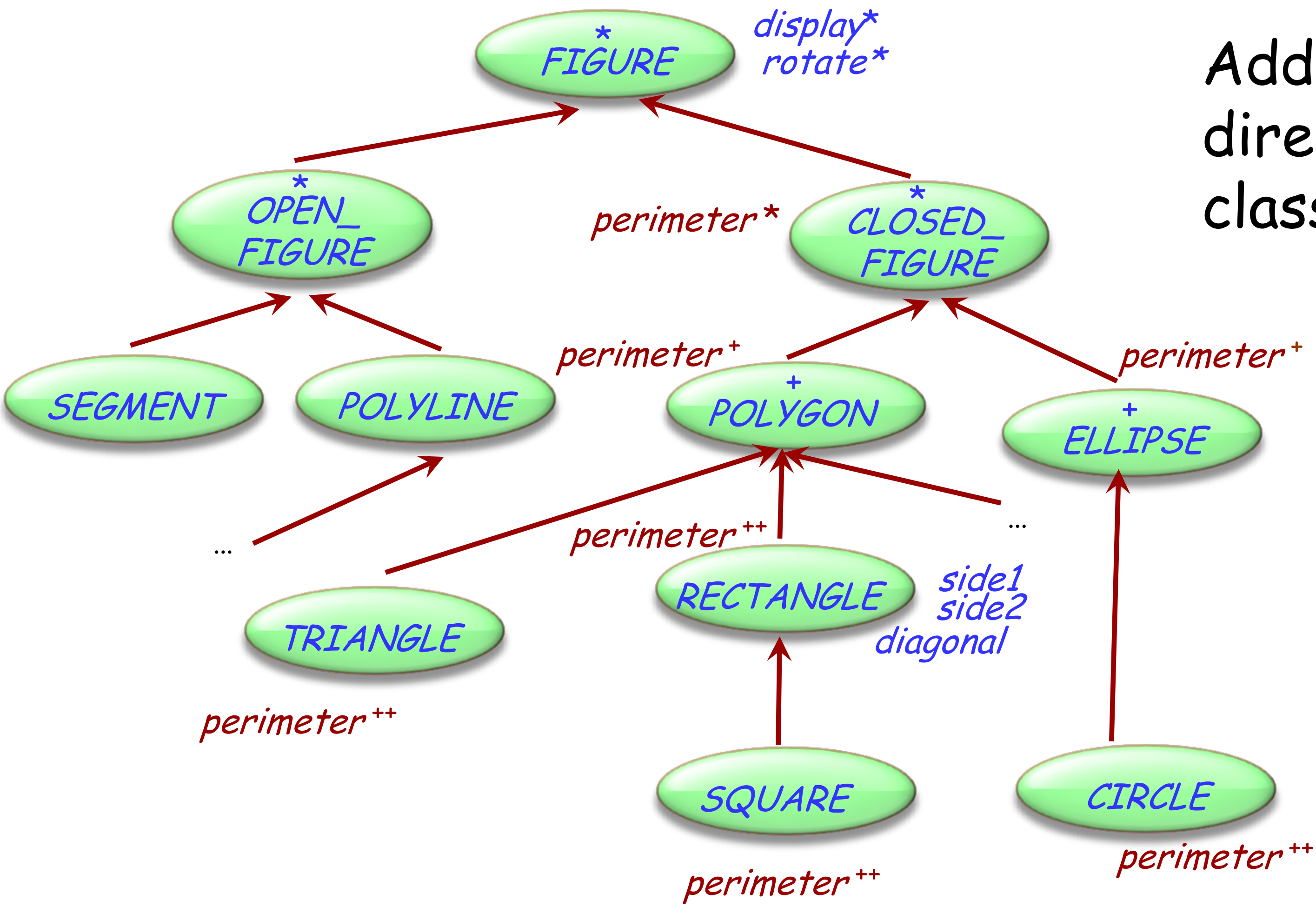
The dirty secret of O-O architecture



Is it easy to add types (e.g. *TRIANGLE*) to existing operations

What about the reverse: adding an operation to existing types?

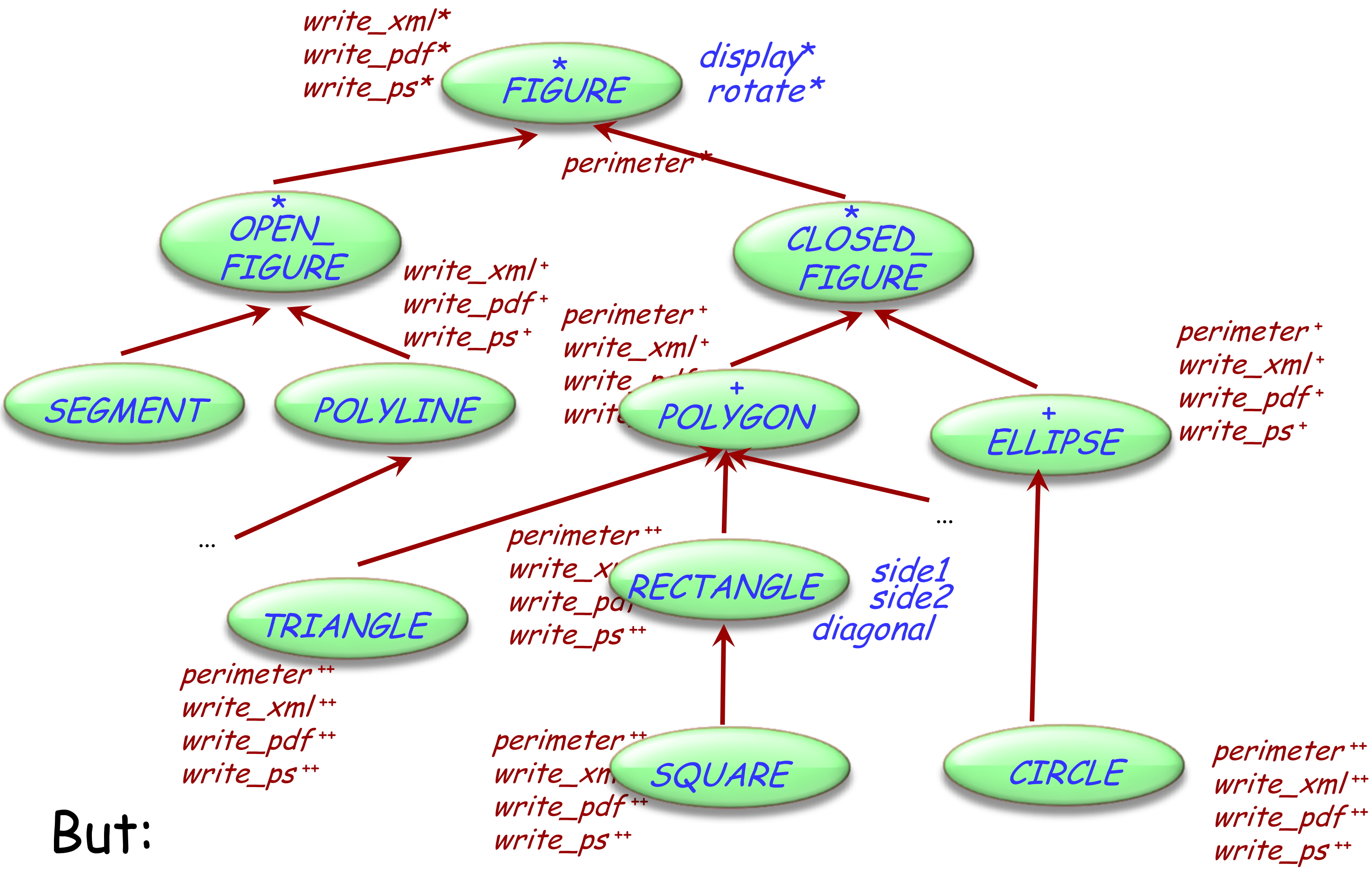
Adding operations – solution 1



Add them directly to the classes

Dynamic binding will take care of finding the right version

Adding operations – solution 1



But:

- operations may clutter the classes
- classes might belong to libraries out of your control

Adding operations – solution 2

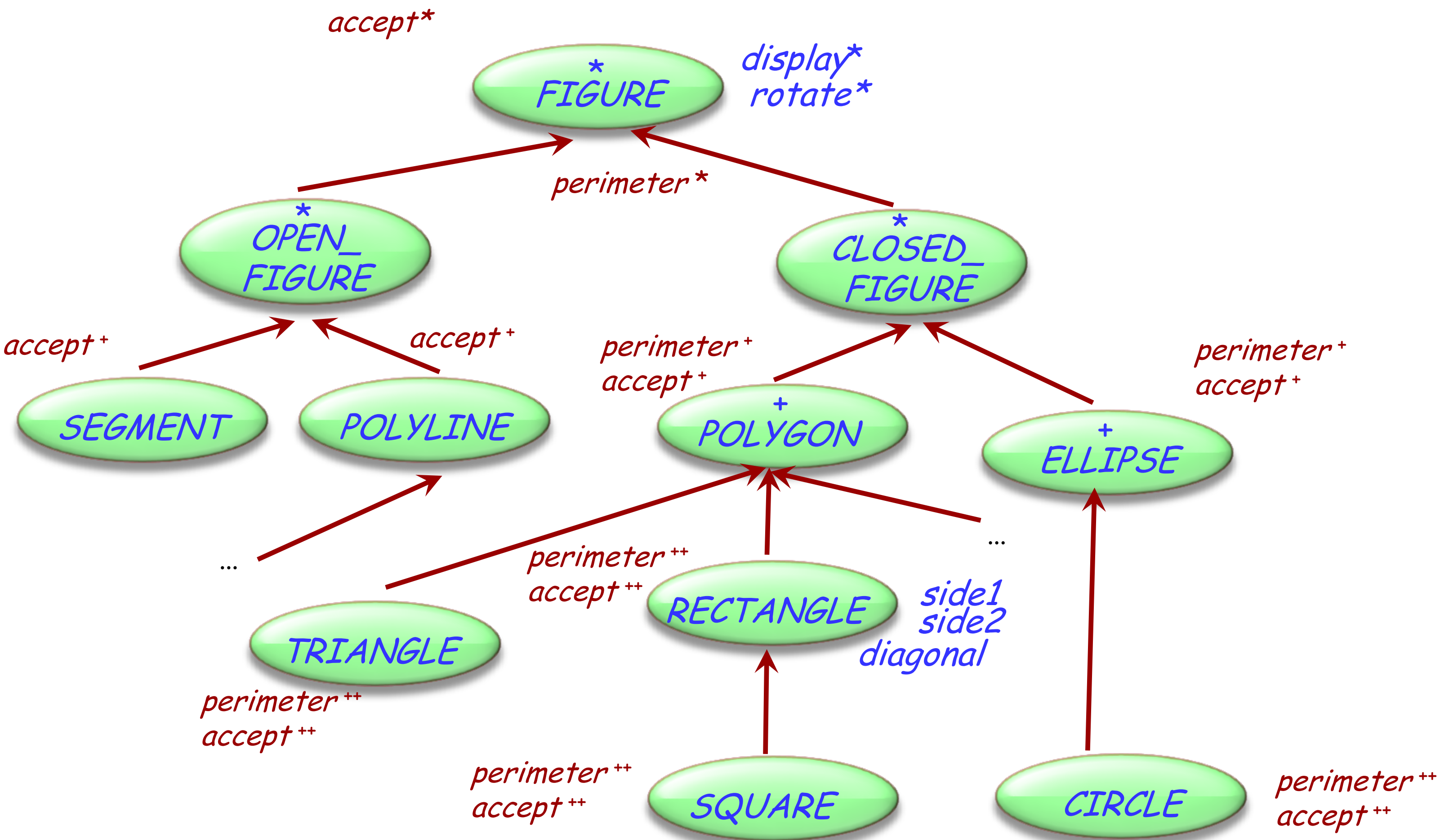
```
write_xml (f : FIGURE)
    -- Write figure to xml.
  require exists: f /= Void
  do
    ...
    if attached {RECT} f as r then
          doc.put_string ("<rect/>")
    end
    if attached {CIRCLE} f as c then
        doc.put_string ("<circle/>")
    end
    ... Other cases ...
  end
end
```

```
write_ps (f : FIGURE)
    -- Write figure to xml.
  require exists: f /= Void
  do
    ...
    if attached {RECT} f as r then
          doc.put_string (r.side_a.out)
    end
    if attached {CIRCLE} f as c then
        doc.put_string (c.diameter)
    end
    ... Other cases ...
  end
end
```

But:

- Loose benefits of dynamic binding
- Many large conditionals

Adding operations – solution 3



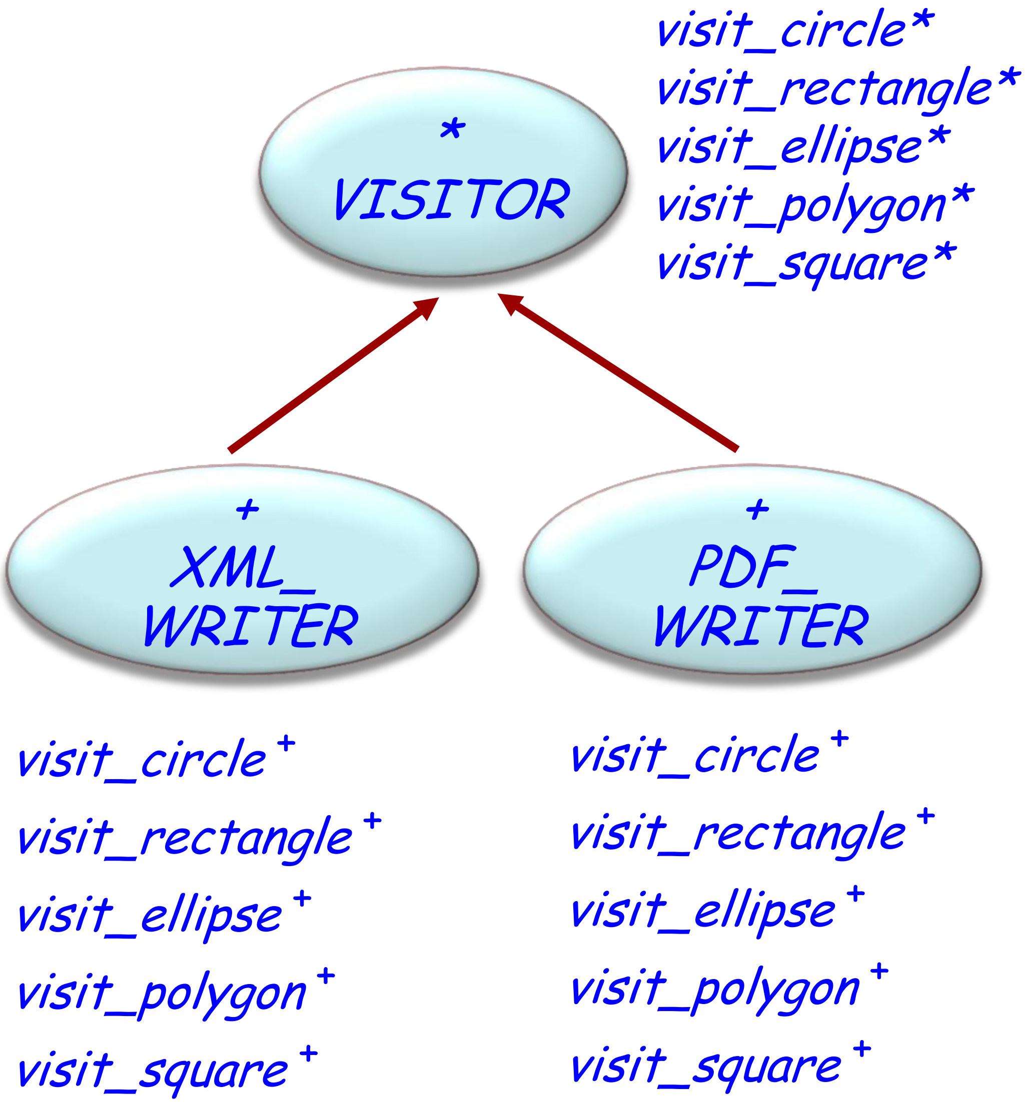
Combine solution 1 & 2:

- Put operations into a separate class
- Add one placeholder operation *accept* (dynamic binding)

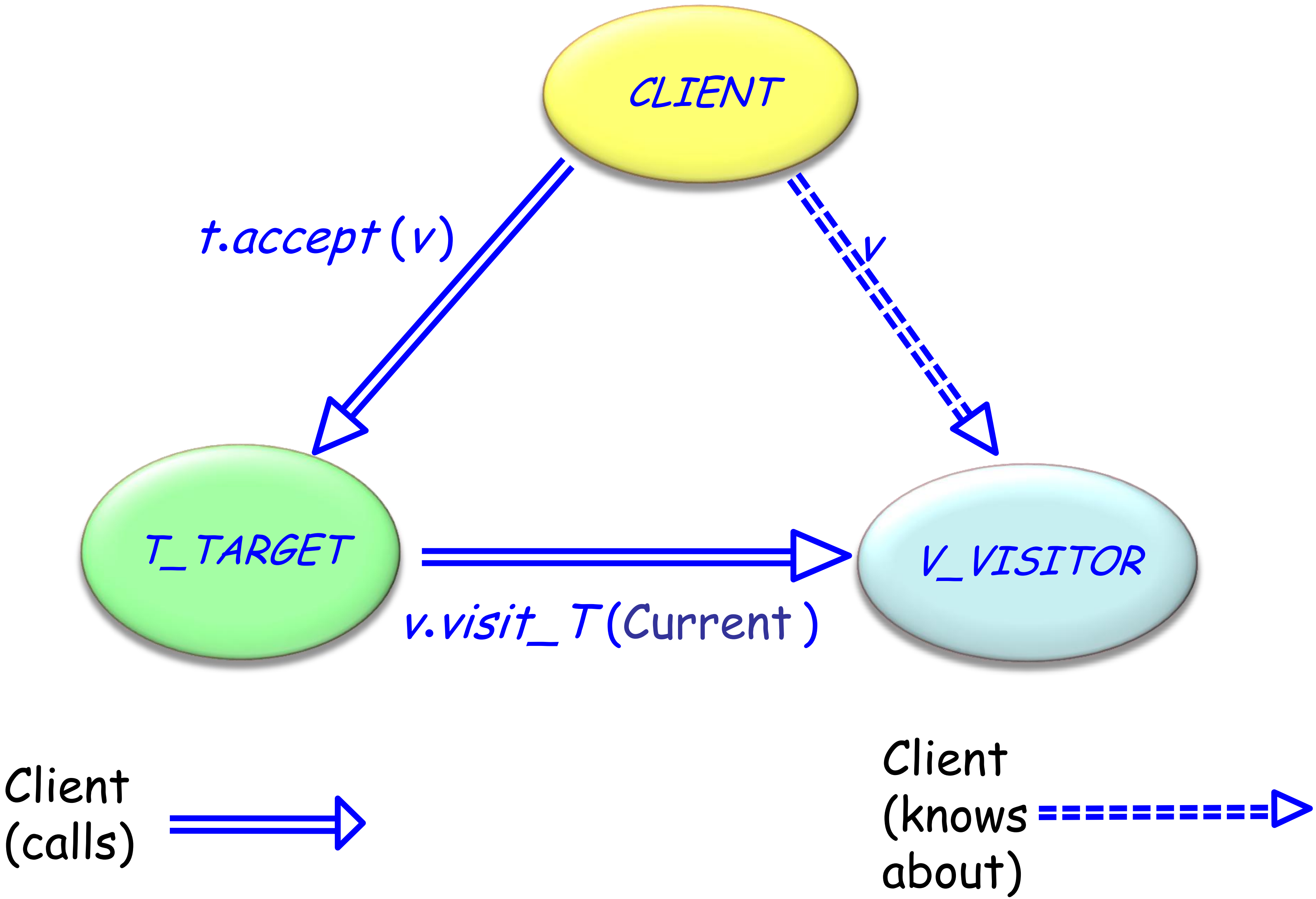
Adding operations – solution 3

```
class FIGURE
feature
accept (v : VISITOR)
    --Call procedure of visitor.
  deferred
  end
    ... Other features …
end
```

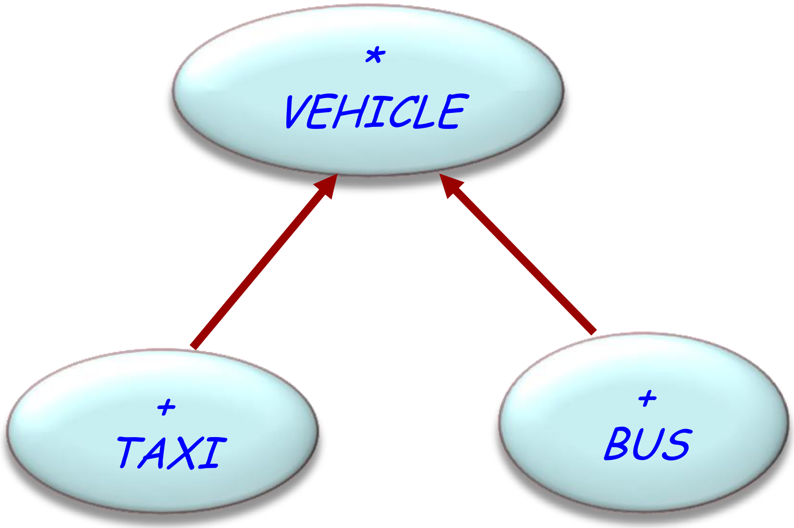
```
class CIRCLE
feature
accept (v : VISITOR)
    --Call procedure of visitor.
  do
    v.visit_circle (Current)
  end
    ... Other features …
end
```



The visitor ballet



Vehicle example



We want to add external functionality, for example:

- Maintenance
- Schedule a vehicle for a particular day

Visitor participants

Target classes

Example: *BUS*, *TAXI*

Client classes

Application classes that need to perform operations on target objects

Visitor classes

Written only to smooth out the collaboration between the other two

Visitor participants

Visitor

General notion of visitor

Concrete visitor

Specific visit operation, applicable to all target elements

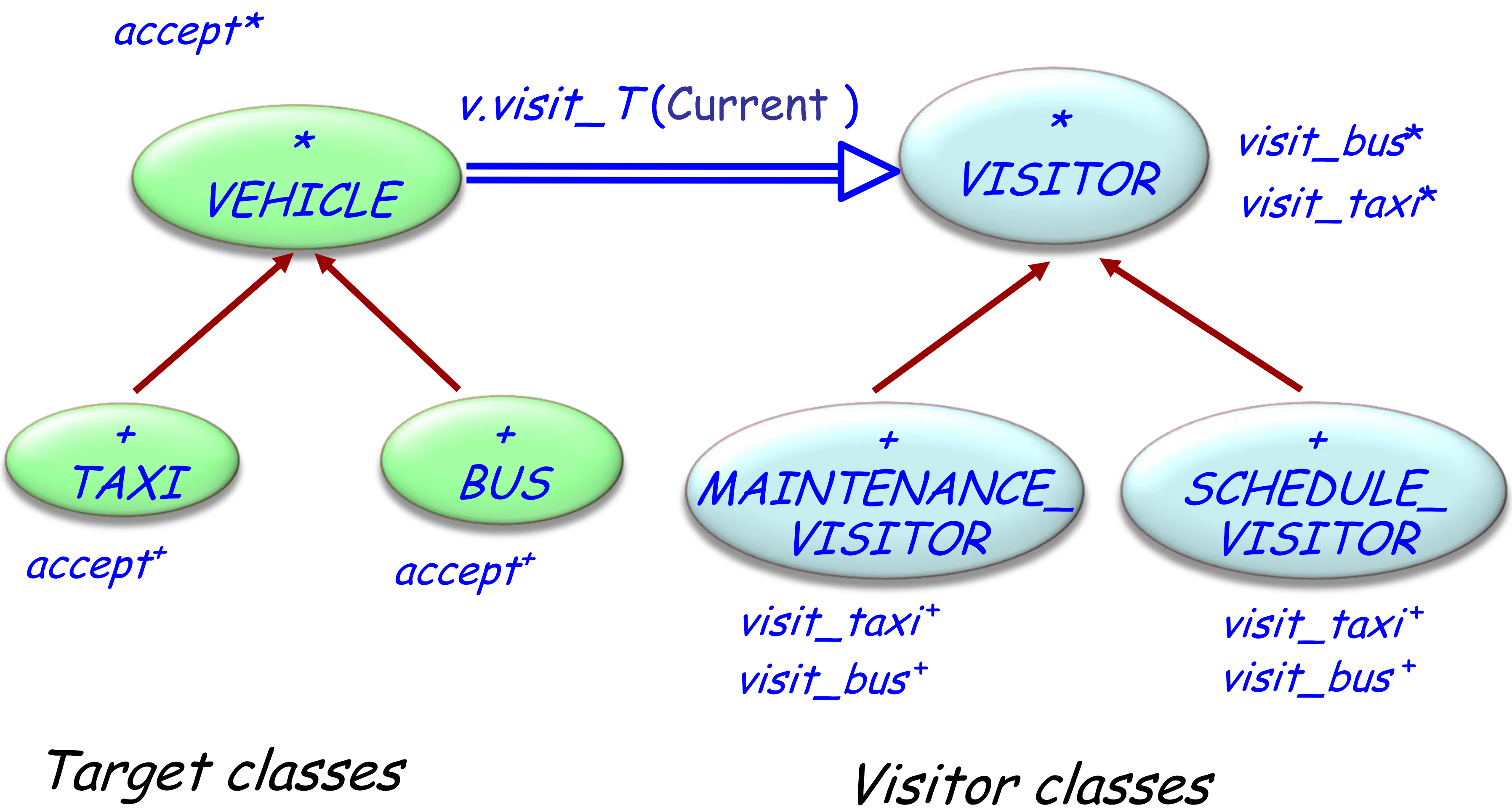
Target

General notion of visitable element

Concrete target

Specific visitable element

Visitor class hierarchies



The maintenance visitor

```
class MAINTENANCE_VISITOR inherit
    VISITOR
feature -- Basic operations
    visit_taxi (t: TAXI)
                -- Perform maintenance operations on t.
        do
                t.send_to_garage (Next_monday)
        end

    visit_bus (b: BUS)
                -- Perform maintenance operations on b.
        do
                b.send_to_depot
        end
end
```

The scheduling visitor

```
class MAINTENANCE_VISITOR inherit
    VISITOR
feature -- Basic operations
    visit_taxi (t: TAXI)
                -- Perform scheduling operations on t.
        do
                ...
        end

    visit_bus (b: BUS)
                -- Perform scheduling operations on b.
        do
                ...
        end
end
```

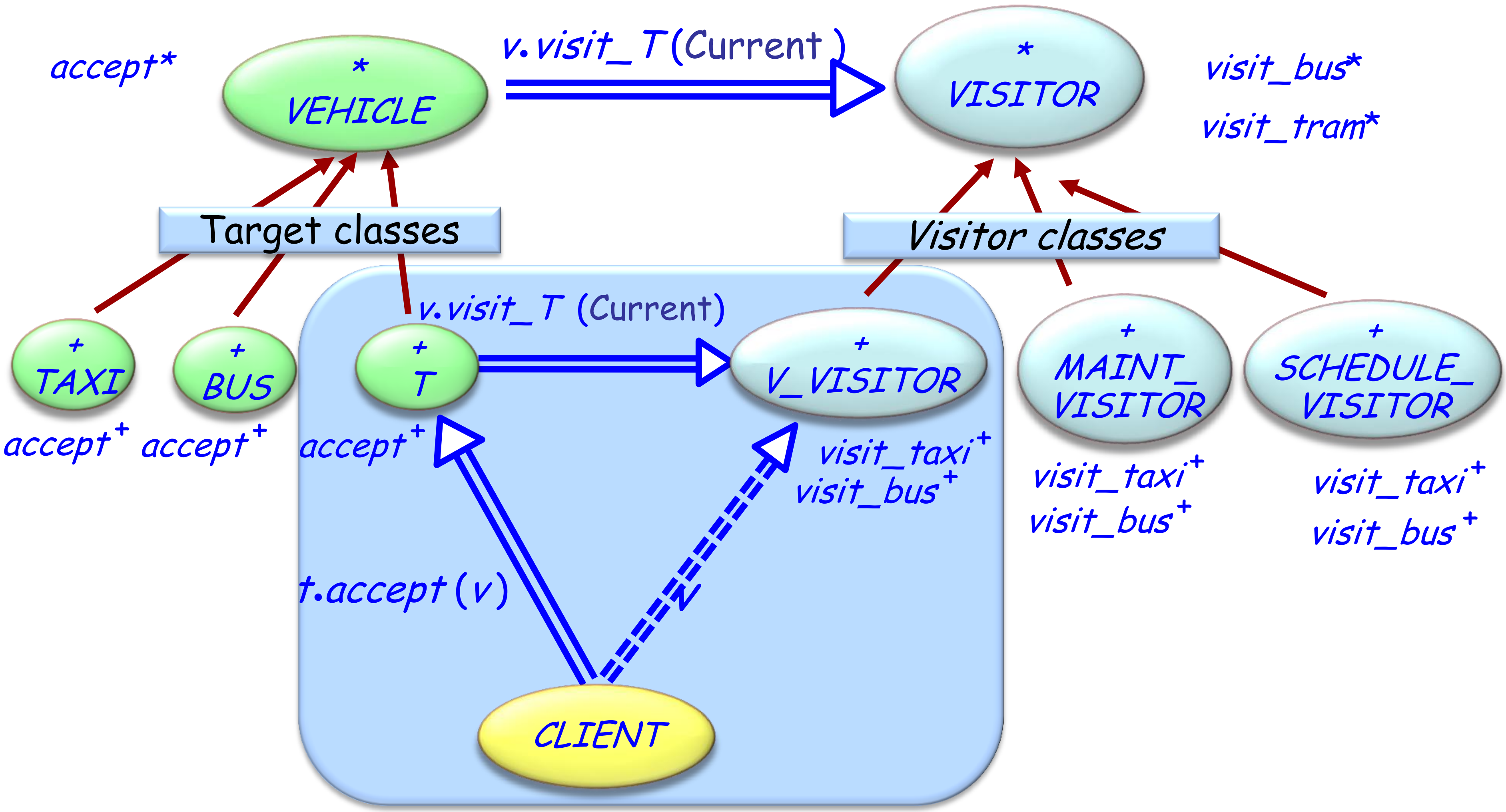
Changes to the target classes

```
deferred class
  VEHICLE
feature

  ... Normal VEHICLE
  features ...

  accept (v: VISITOR)
      -- Apply vehicle visit to v.
    deferred
    end

end
```

```
class BUS inherit
  VEHICLE
feature
  accept (v: VISITOR)
        -- Apply bus visit to v.
    do
      v.visit_bus (Current)
    end
end
```

```
class TAXI inherit
  VEHICLE
feature
  accept (v: VISITOR)
        -- Apply taxi visit to v.
    do
      v.visit_taxi (Current)
    end
end
```

The visitor pattern



Example client calls:

bus21.accept (maint_visitor)

fleet.item.accept (maint_visitor)

Visitor provides double dispatch

Client:

t.accept (v)

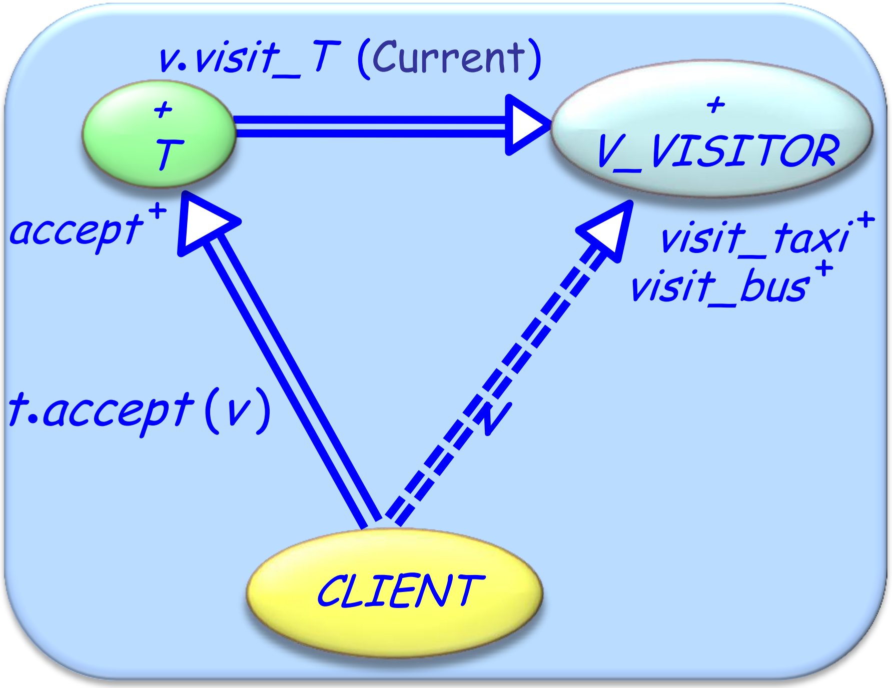
Target class (in *accept*):

v.visit_T (t)

Visitor class *V_VISITOR* (in *visit_T*):

visit_T (t)

-- For the right *V* and *T*!



Visitor - Consequences

Makes adding new operations easy

Gathers related operations, separates unrelated ones

Avoids assignment attempts

- Better type checking

Adding new concrete element is hard

Visitor vs dynamic binding

Dynamic binding:

- Easy to add types
- Hard to add operations

Visitor:

- Easy to add operations
- Hard to add types

Visitor – Componentization

Fully componentizable

One generic class *VISITOR* [*G*]

e.g. *maintenance_visitor*: *VISITOR* [*VEHICLE*]

Actions represented as agents

actions: *LIST* [*PROCEDURE* [*ANY*, *TUPLE* [*G*]]]

No need for *accept* features

visit determines the action applicable to the given element

For efficiency

Topological sort of actions (by conformance)

Cache (to avoid useless linear traversals)

Visitor Library interface (1/2)

```
class
    VISITOR [G]
create
    make
feature {NONE} -- Initialization
    make
            -- Initialize actions.
feature -- Visitor
    visit (e : G)
                -- Select action applicable to e .
            require
                e_exists: e /= Void
feature -- Access
    actions: LIST [PROCEDURE [ANY, TUPLE [G]]]
                -- Actions to be performed depending on the element
```

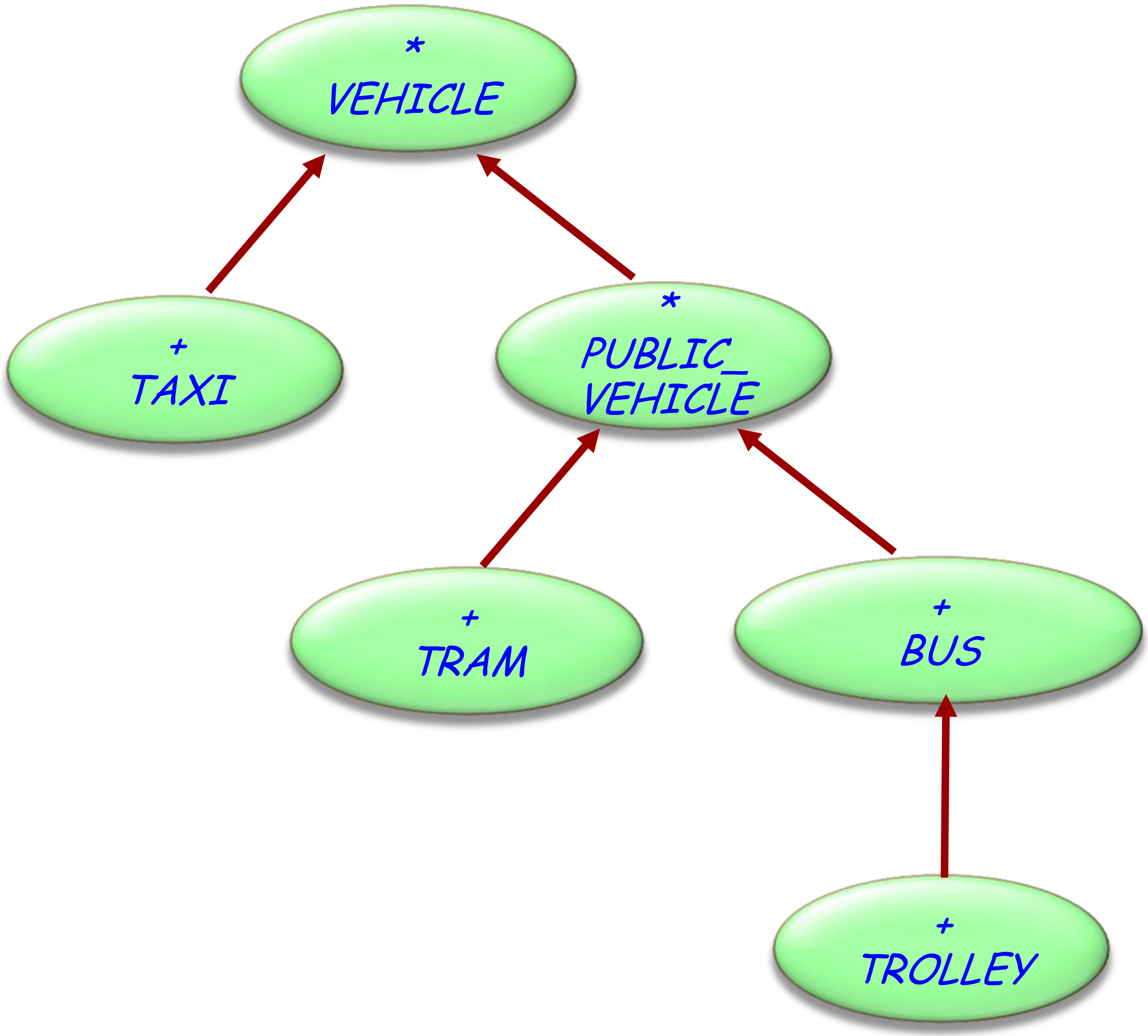
Visitor Library interface (2/2)

```
feature -- Element change
    extend (action: PROCEDURE [ANY, TUPLE [G]])
            -- Add action to list.
        require
            action_exists: action /= Void
        ensure
            one_more: actions.count = old actions.count + 1
            inserted: actions.last = action

    append (some_actions: ARRAY [PROCEDURE [ANY, TUPLE [G]]])
            -- Append actions in some_actions
            -- to the end of the actions list.
        require
            actions_exit: some_actions /= Void
            no_void_action: not some_actions.has (Void)
invariant
    actions_exist: actions /= Void
    no_void_action: not actions.has (Void)
end
```

Using the Visitor Library

```
maintenance_visitor: VISITOR [VEHLICLE]

create maintenance_visitor.make
maintenance_visitor.append ([
                    agent maintain_taxi,
                    agent maintain_trolley,
                    agent maintain_tram
               ])

maintain_taxi (a_taxi: TAXI) ...
maintain_trolley (a_trolley: TROLLEY) ...
maintain_tram (a_tram: TRAM) ...
```

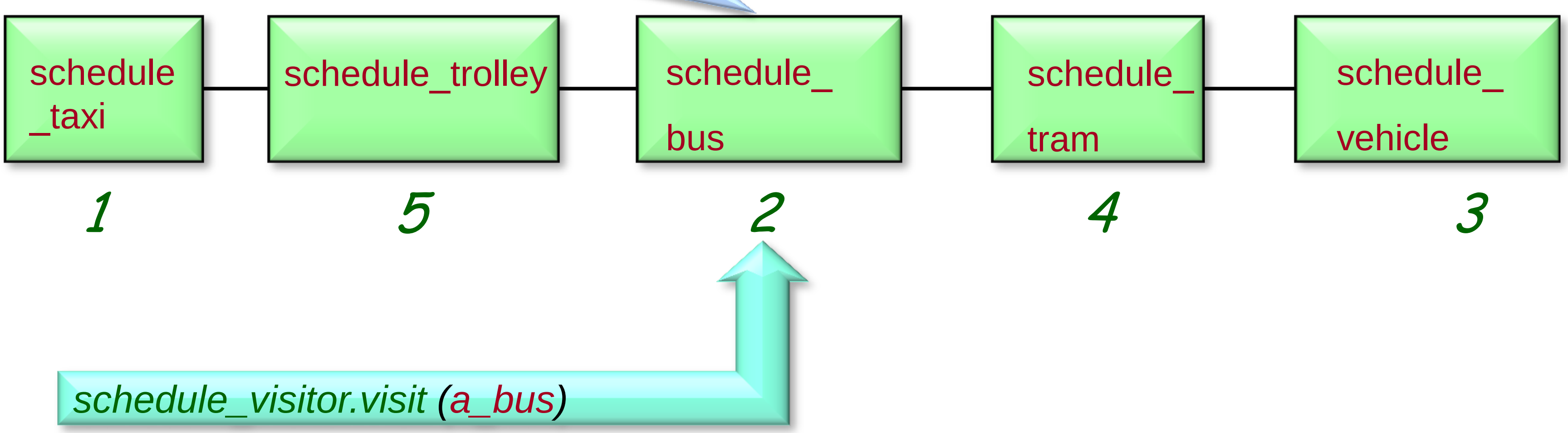
Topological sorting of agents (1/2)



Topological sorting of agents (2/2)

schedule_visitor.extend (agent *schedule_taxi*)
schedule_visitor.extend (agent *schedule_bus*)
schedule_visitor.extend (agent *schedule_vehicle*)
schedule_visitor.extend (agent *schedule_tram*)
schedule_visitor.extend (agent *schedule_trolley*)

For agent *schedule_a* (*a*: A) and *schedule_b* (*b*: B), if A conforms to B, then position of *schedule_a* is before position of *schedule_b* in the agent list



Visitor library vs. visitor pattern

Visitor library:

- Removes the need to change existing classes
- More flexibility (may provide a procedure for an intermediate class, may provide no procedure)
- More prone to errors – does not use dynamic binding to detect correct procedure, no type checking

Visitor pattern

- Need to change existing classes
- Dynamic binding governs the use of the correct procedure (type checking that all procedures are available)
- Less flexibility (need to implement all procedures always)

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- Strategy
- Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Strategy

Intent:

"*Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it*".

[Gamma et al., p 315]

Example application

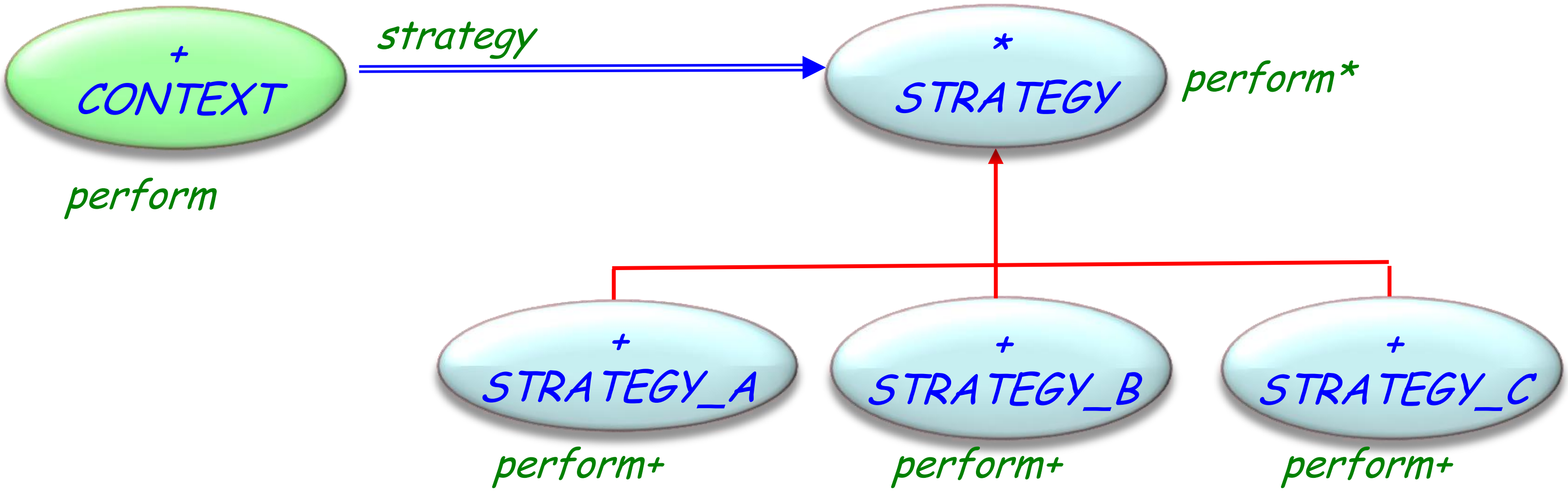
selecting a sorting algorithm on-the-fly

Life without strategy: a sorting example

```
feature -- Sorting
    sort (il: LIST [INTEGER ]; st: INTEGER)
            -- Sort il using algorithm indicated by st.
        require
            is_valid_strategy (st)
        do
            inspect
                st
            when binary then …
            when quick then …
            when bubble then …
            else …
            end
        ensure
            list_sorted: …
        end
```

What if a new algorithm is needed ?

Strategy pattern: overall architecture



Class STRATEGY

```
deferred class
    STRATEGY

feature -- Basic operation

    perform
            -- Perform algorithm according to chosen strategy.
        deferred
        end

end
```

Using a strategy

```
class
    CONTEXT

create
    make

feature -- Initialization

    make (s: like strategy)
            -- Make s the new strategy.
            -- (Serves both as creation procedure and to reset strategy.)
        do
            strategy := s
        ensure
            strategy_set: strategy = s
        end
```

Using a strategy

```
feature -- Basic operations

    perform
            -- Perform algorithm according to chosen strategy.
        do
            strategy.perform
        end

feature {NONE} -- Implementation

    strategy: STRATEGY
        -- Strategy to be used

end
```

Using the strategy pattern

```
sorter_context: SORTER_CONTEXT
bubble_strategy: BUBBLE_STRATEGY
quick_strategy: QUICK_STRATEGY
hash_strategy: HASH_STRATEGY

create sorter_context.make (bubble_strategy)
sorter_context.sort (a_list)
sorter_context.make (quick_strategy)
sorter_context.sort (a_list)
sorter_context.make (hash_strategy)
sorter_context.sort (a_list)
```

Now, what if a new algorithm is needed ?

Application classes can also inherit from *CONTEXT* (rather than use it as clients)

Strategy - Consequences

- Pattern covers classes of related algorithms
- Provides alternative implementations without conditional instructions

- Clients must be aware of different strategies
- Communication overhead between Strategy and Context
- Increased number of objects

Strategy - Participants

Strategy

declares an interface common to all supported algorithms.

Concrete strategy

implements the algorithm using the Strategy interface.

Context

- is configured with a concrete strategy object.
- maintains a reference to a strategy object.

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- ✓ Strategy
- Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Chain of responsibility - Intent

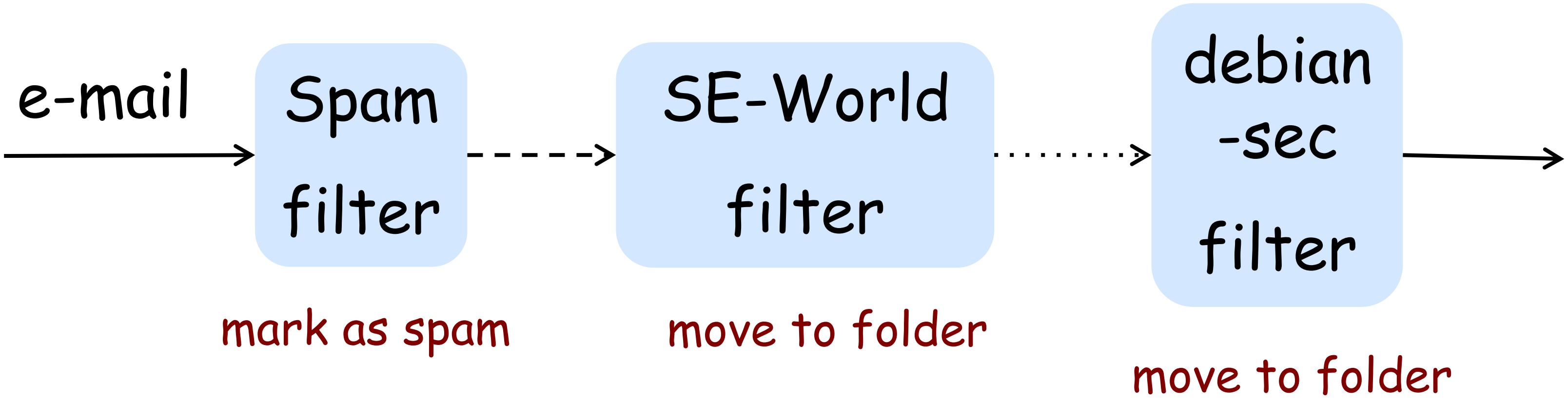
Intent:

"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."

Example application

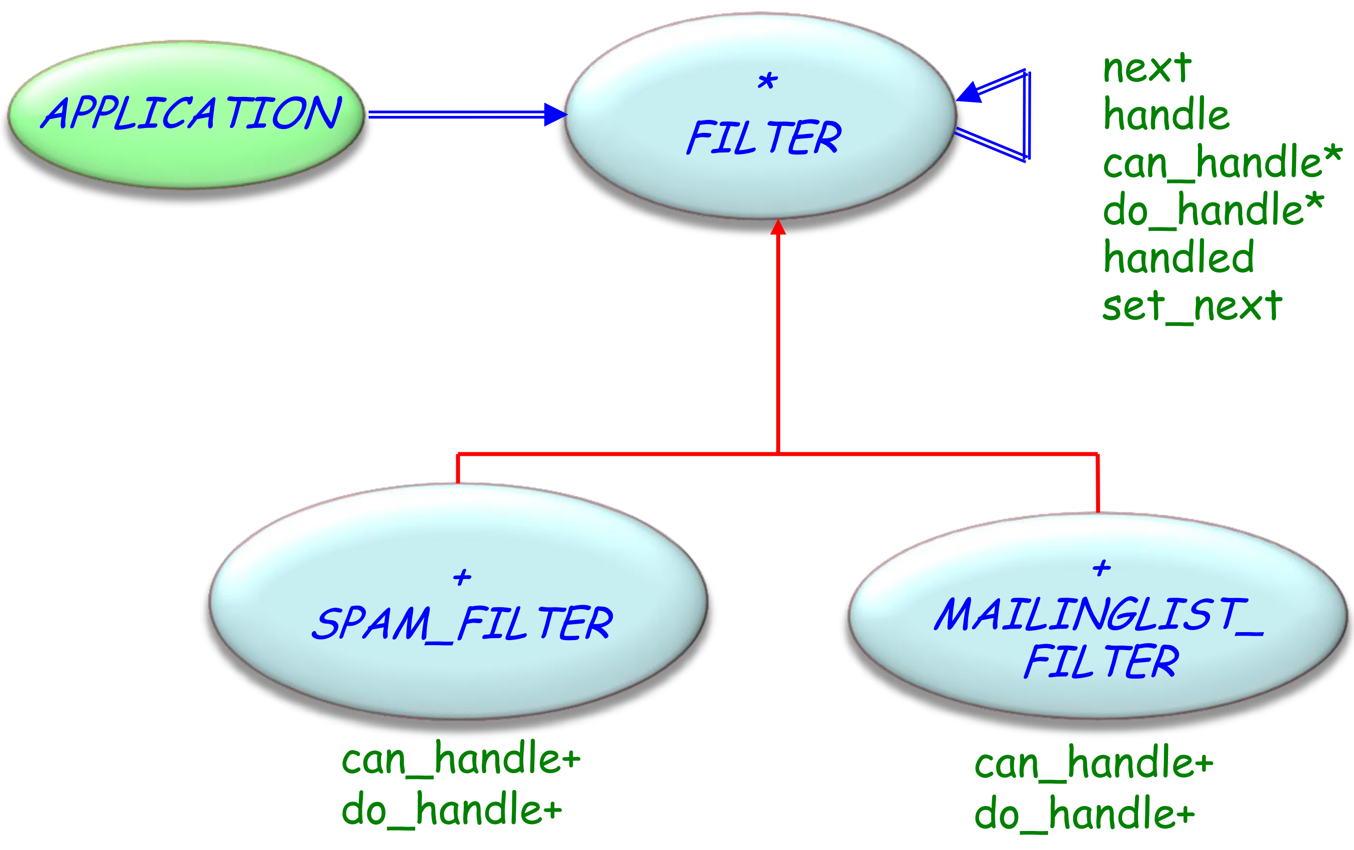
A GUI event is passed from level to level (such as from button to dialog and then to application)

Example: e-mail filtering



If a filter can handle the request (e-mail) it will. Otherwise it will pass it on to the next filter, until it drops out of the chain of responsibility.

Example implementation



Filter

```
deferred class FILTER
feature – Access
    next : FILTER          -- Successor in the chain of responsibility

feature -- Element change
    set_next (n : like next)
            -- Set next to n.
        do
            next := n
        ensure
            next_set: next = n
        end

feature -- Status report
    can_handle (r : E_MAIL): BOOLEAN  deferred end
                           -- Can this handler handle r?
    handled: BOOLEAN       -- Has request been handled?
```

Filter

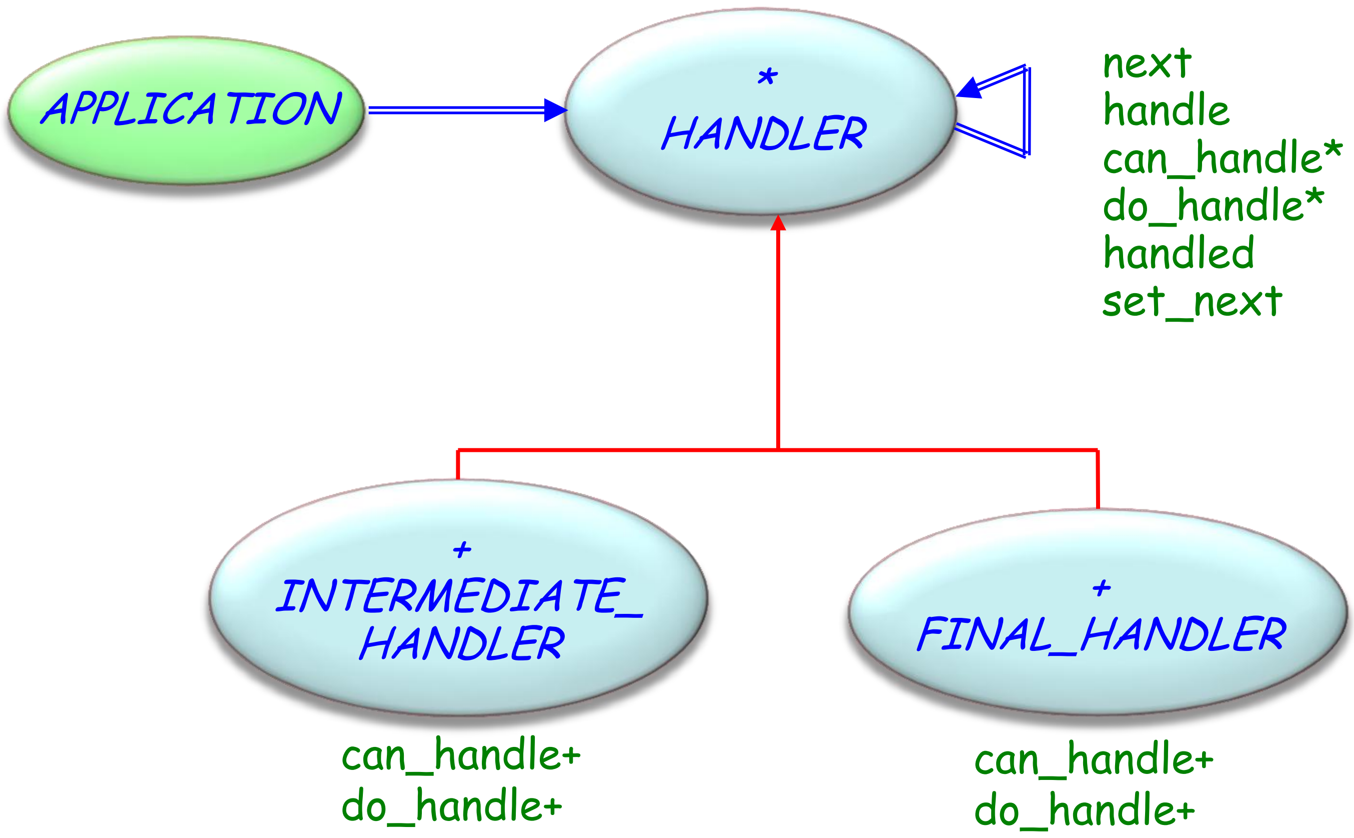
```
feature {NONE} -- Implementation

    do_handle (r : G)
            -- Handle r.
        require
            can_handle: can_handle (r)
        deferred
        end

feature -- Basic operations
    handle (r : E_MAIL)
            -- Handle r if can_handle otherwise forwardto next.
            -- If no next, set handled to False.
        do
            if can_handle (r ) then do_handle (r ) ; handled := True
            else
                if next /= Void then next.handle (r ) ; handled := next.handled
                else handled := False end
            end
        ensure
            can_handle (r ) implies handled
            (not can_handle (r ) and next /= Void) implies handled = next.handled
            (not can_handle (r ) and next = Void) implies not handled
        end
end
```

Concrete filters

```
class SPAM_FILTER inherit FILTER
create set_next, default_create
feature -- Status report
    can_handle (r : E_MAIL)
            -- Can this handler handle r?
        do
            -- Find out whether it
            -- classifies as spam.
        end

feature {NONE} – Implementation

do_handle (r : G)
            -- Handle r.
        do
            -- Mark e-mail as spam.
        end
end
```

```
class MAILINGLIST_FILTER
inherit FILTER
create set_next, default_create
feature -- Status report
    can_handle (r : E_MAIL)
            -- Can this handler handle r?
        do
            -- Is it an e-mail sent to a
            -- mailinglist?
        end

feature {NONE} -- Implementation
do_handle (r : G)
            -- Handle r.
        do
            -- Move to correct folder.
        end

folder : FOLDER -- Folder to move mail

... -- Implementation of set_folder
end
```

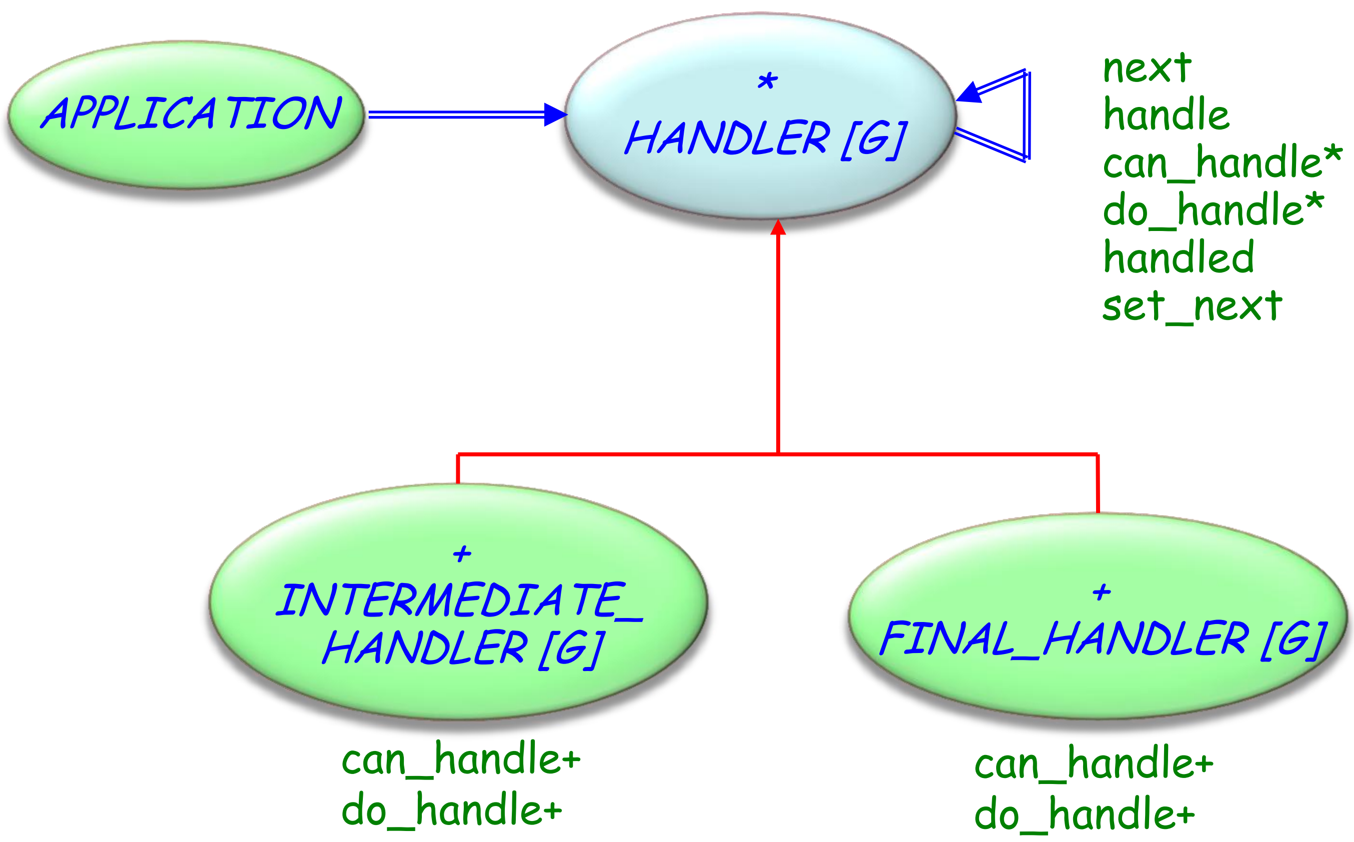
Chain of responsibility: overall architecture



Chain of responsibility: Componentization

Fully componentizable

Chain of responsibility: library



Handlers

```
deferred class
        HANDLER [G]
create default_create, make

feature {NONE} -- Initialization
        make (n : like next)
                        -- Set next  to n.
                do
                        next := n
                ensure
                        next_set: next = n
                end

feature -- Access
    next : HANDLER [G]
                -- Successor in the chain of responsibility
feature -- Status report
        can_handle (r : G): BOOLEAN  deferred end
                        -- Can this handler handle r?

        handled: BOOLEAN
                        -- Has request been handled?
```

Handlers

```
feature -- Basic operations
    handle (r : G)
            -- Handle r if can_handle  otherwise forward it to next.
            -- If no next, set handled  to False.
      do
            if  can_handle (r ) then
                  do_handle (r ) ; handled := True
            else
                  if  next /= Void  then
                        next.handle (r )  ; handled := next.handled
                  else
                        handled := False
                  end
            end
      ensure
            can_handle (r ) implies  handled
            (not  can_handle (r ) and  next /= Void) implies  handled = next.handled
            (not  can_handle (r ) and  next = Void)  implies not  handled
      end
```

Class *HANDLER* [*G*] (3/3)

```
feature -- Element change
    set_next (n : like next)
            -- Set next to n.
        do
            next := n
        ensure
            next_set: next = n
        end

feature {NONE} – Implementation

    do_handle (r : G)
            -- Handle r.
        require
            can_handle: can_handle (r)
        deferred
        end

end
```

Chain of responsibility - Consequences

Reduced coupling

An object only has to know that a request will be handled "appropriately". Both the receiver and the sender have no explicit knowledge of each other

Added flexibility in assigning responsibilities to objects

Ability to add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time

Receipt is not guaranteed

the request can fall off the end of the chain without ever being handled

Chain of responsibility - Participants

Handler

- defines an interface for handling requests.
- (optional) implements the successor link.

Concrete handler

- handles requests it is responsible for.
- can access its successor.
- if the Concrete handler can handle the request, it does so; otherwise it forwards the request to its successor.

Application (Client)

initiates the request to a Concrete handler object on the chain.

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- State
- ✓ Strategy
- Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

State pattern

Intent:

"*Allows an object to alter its behavior when its internal state changes. The object will appear to change its class*".

Application example:

- Add attributes without changing class.
- Simulate the (impossible) case of an object changing its type during execution.
- State machine simulation.

Example application: Drawing tool

Mouse actions have different behavior

- Pen tool
  - Mouse down: Start point of line
  - Mouse move: Continue draw of line
  - Mouse up: End draw line, change back to selection mode
- Selection tool
  - Mouse down: Start point selection rectangle
  - Mouse move: Update size of selection rectangle
  - Mouse up: Select everything inside selection rectangle
- Rectangle tool
  - Mouse down: Start point of rectangle
  - Mouse move: Draw rectangle with current size
  - Mouse up: End draw rectangle, change back to selection mode
- ...

Tool state

```
deferred class TOOL_STATE  feature
    process_mouse_down (pos: POSITION)
            -- Perform operation in response to mouse down.
        deferred end

    process_mouse_up (pos: POSITION)
            -- Perform operation in response to mouse up.
        deferred end

    process_mouse_move (pos: POSITION)
            -- Perform operation in response to mouse move.
        deferred end


-- Continued on next slide
```

Tool states know their context (in this solution)

```
feature -- Element change
   set_context (c: CONTEXT)
            -- Attach current state to c.
         do
            context := c
         end

feature {NONE} – Implementation

   context: CONTEXT
         -- The client context using this state.

end
```

A particular state

```
class RECTANGLE_STATE inherit TOOL_STATE
feature -- Access
    start_position: POSITION

feature -- Basic operations
    process_mouse_down (pos: POSITION)
            -- Perform operation in response to mouse down.
        do start_position := pos end

    process_mouse_up (pos: POSITION)
            -- Perform operation in response to mouse up.
        do context.set_state (context.selection_tool) end

    process_mouse_move (pos: POSITION)
            -- Perform edit operation in response to mouse move.
        do context.draw_rectangle (start_position, pos) end

end
```

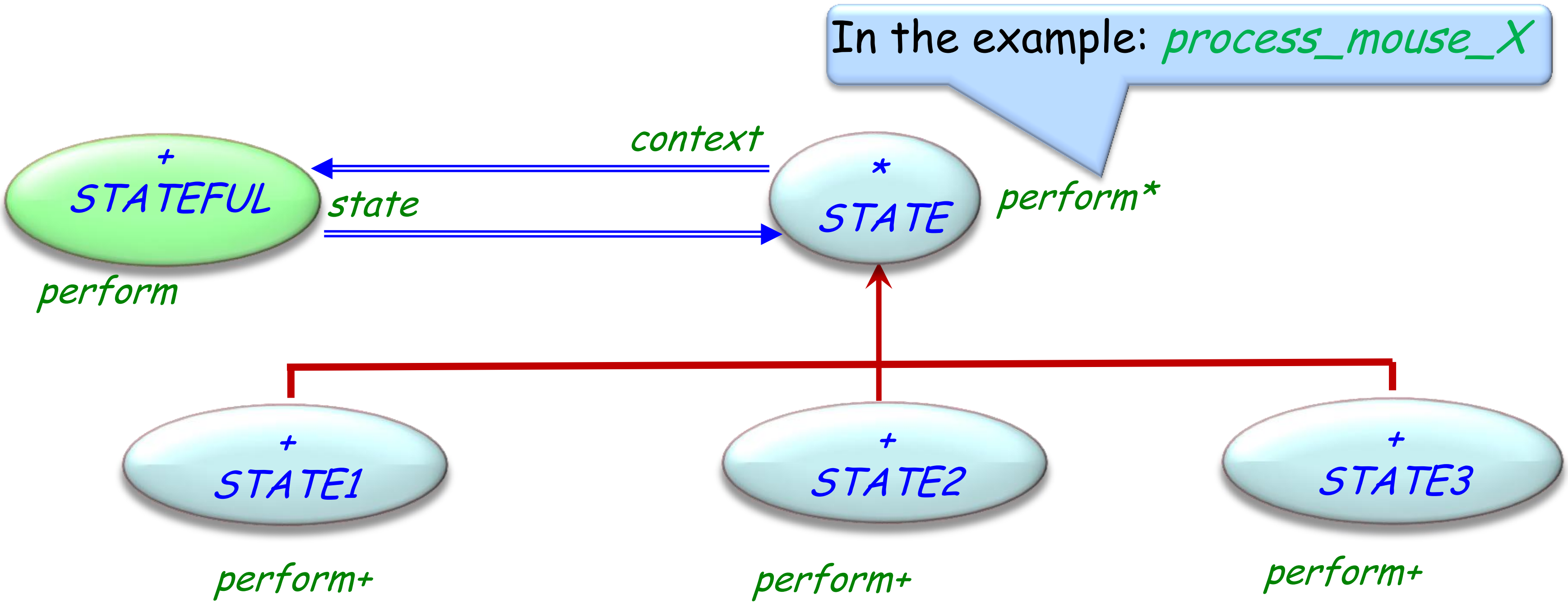
A stateful environment client

```
class CONTEXT feature -- Basic operations
    process_mouse_down (pos: POSITION)
            -- Perform operation in response to mouse down.
        do
            state.process_mouse_down (pos)
        end

    process_mouse_up (pos: POSITION)
            -- Perform operation in response to mouse up.
        do
            state.process_mouse_up (pos)
        end

    process_mouse_move (pos: POSITION)
            -- Perform operation in response to mouse move.
        do
            state.process_mouse_move (pos)
        end
```

Stateful client: status and element change

```
feature -- Access

        pen_tool, selection_tool, rectangle_tool: like state
                        -- Available (next) states.

        state : TOOL_STATE

feature -- Element change

        set_state (s: STATE)
                        -- Make s the next state.
                do
                        state := s
                end

    ... -- Initialization of different state attributes

end
```

State pattern: overall architecture



State pattern - componentization

Componentizable, but not comprehensive

State - Consequences

The pattern localizes state-specific behavior and partitions behavior for different states

It makes state transitions explicit

State objects can be shared

State - Participants

Stateful

- defines the interface of interest to clients.
- maintains an instance of a Concrete state subclass that defines the current state.

State

defines an interface for encapsulating the behavior associated with a particular state of the Context.

Concrete state

each subclass implements a behavior associated with a state of the Context

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Creational patterns

- Hide the creation process of objects
- Hide the concrete type of these objects
- Allow dynamic and static configuration of the system

Explicit creation in O-O languages

Eiffel:

```
create x.make (a, b, c)
```

C++, Java, C#:

```
x = new T (a, b, c)
```

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- **Factory Method**
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Factory Method pattern

Intent:

"*Define*[*s*] *an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*" [Gamma et al.]

C++, Java, C#: emulates constructors with different names

Factory method

In client, instead of

```
create {T} x.make
```

use

```
x := new_t
```

with *new_t* defined as

```
new_t (args: G): T
        -- New instance of T
    do
        create {S} Result.make (args)
            -- S conforms to T
    end
```

Benefits of factory method

Factory method is not just the syntactic replacement of

create { *T* } *x.make* (1)

by

x := *factory.new_t* (2)

because:

T could be a deferred class

then (1) would not be possible

factory can take advantage of polymorphism

Design patterns (GoF)

Creational

- Abstract Factory
- Singleton
- ✓ Factory Method
- Builder
- Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Abstract factory pattern

Intent:

"*Provide*[s] *an interface for creating families of related or dependent objects without specifying their concrete classes.*" [Gamma et al.]

Abstract Factory: example

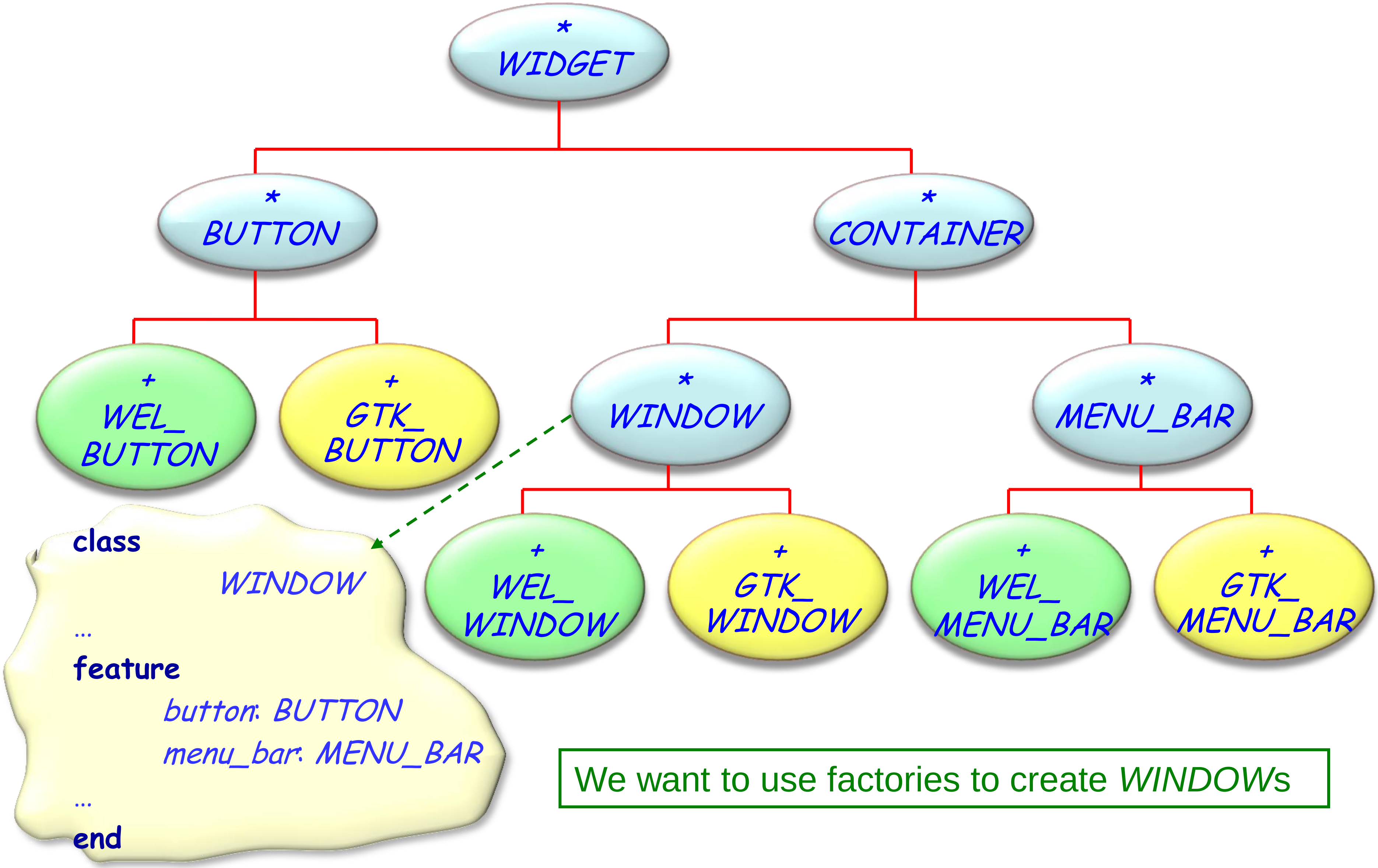
Widget toolkit (EiffelVision, Java Swing)

- Different look and feel, e.g. for Unix & Windows
- Family of widgets: Scroll bars, buttons, dialogs…
- Want to allow change of look & feel

→ Most parts of the system need not know which look & feel is used

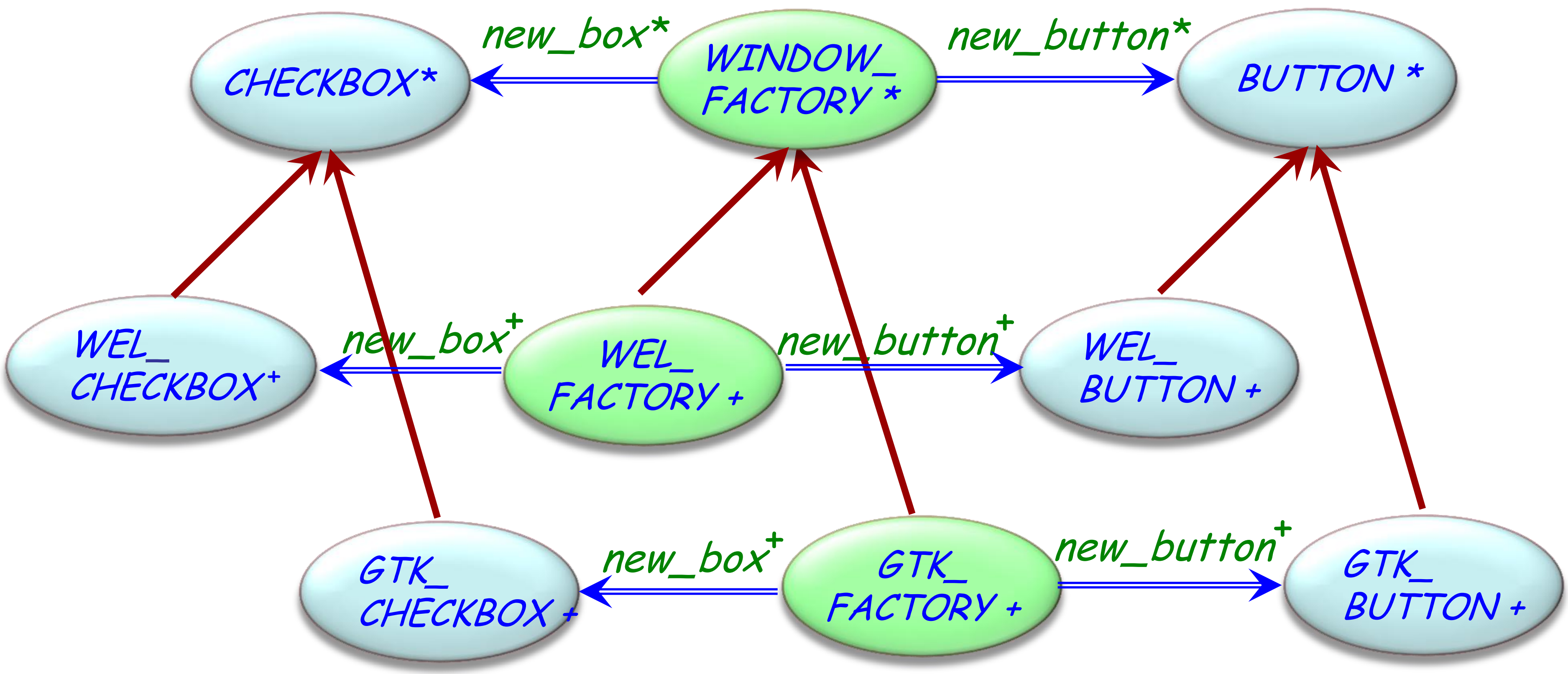
→ Creation of widget objects should not be distributed

Managing parallel hierarchies with factories



We want to use factories to create *WINDOW*s

Abstract widget factory example



With an Abstract Factory (1/6)

```
deferred class
    WINDOW_FACTORY
feature -- Factory functions
    new_window: WINDOW deferred end
    new_button: BUTTON deferred end
    new_menu_bar: MENU_BAR deferred end
...
end
```

With an Abstract Factory (2/6)

```
class
        WEL_WINDOW_FACTORY
inherit
        WINDOW_FACTORY
create
        make
feature {NONE} -- Initialization
        make (...) do ...
feature -- Factory functions
        new_window: WEL_WINDOW do ...
        new_button: WEL_BUTTON do ...
        new_menu_bar: WEL_MENU_BAR do ...
...
end
```

Factory ensures that all widgets of the window are Windows widgets

With an Abstract Factory (3/6)

```
class
    GTK_WINDOW_FACTORY
inherit
    WINDOW_FACTORY
create
    make
feature {NONE} -- Initialization
    make (...) do ...
feature -- Factory functions
    new_window: GTK_WINDOW do ...
    new_button: GTK_BUTTON do ...
    new_menu_bar: GTK_MENU_BAR do ...
...
end
```

Factory ensures that all widgets of the window are Gtk widgets

With an Abstract Factory (4/6)

```
deferred class
        APPLICATION
...
feature -- Initialization
        build_window is
                        -- Build window.
                local
                        window: WINDOW
                do
                        window := window_factory.new_window
                        ...
                end
feature {NONE} -- Implementation
        window_factory: WINDOW_FACTORY
                        -- Factory of windows
invariant
        window_factory_not_void: window_factory /= Void
end
```

Abstract notion

Does not name platform

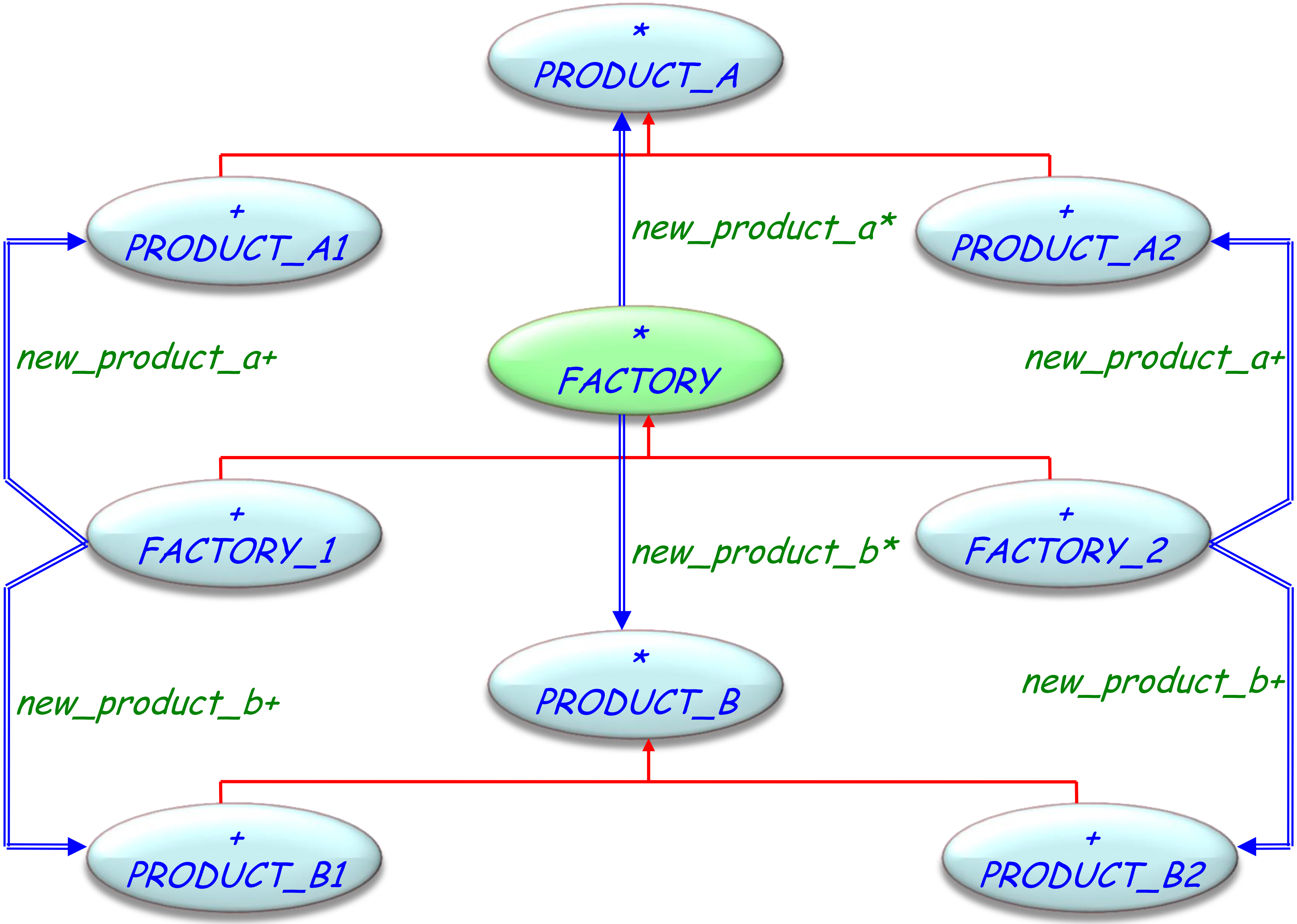
With an Abtract Factory (5/6)

```
class
        WEL_APPLICATION
inherit
        APPLICATION
create
        make
feature {NONE} -- Initialization
      make is
                -- Create window_factory.
            do
                create {WEL_WINDOW_FACTORY}
                window_factory.make(...)
            end
...
end
```

With an Abtract Factory (6/6)

```
class
        GTK_APPLICATION
inherit
        APPLICATION
create
        make
feature {NONE} -- Initialization
      make is
                -- Create window_factory.
            do
                create {GTK_WINDOW_FACTORY}
                window_factory.make(...)
            end
...
end
```

Abstract factory: overall architecture



Reasons for using an abstract factory

- Most parts of a system should be independent of how its objects are created, are represented and collaborate
- The system needs to be configured with one of multiple families
- A family of objects is to be designed and only used together
- You want to support a whole palette of products, but only show the public interface

Abstract factory pattern: properties

- Isolates concrete classes
- Makes exchanging product families easy
- Promotes consistency among products
- Supporting new kinds of products is difficult

Abstract factory pattern: criticism

Code redundancy:

- The factory classes, e.g. *GTK_FACTORY* and *WEL_FACTORY* will be similar

Lack of flexibility:

- *FACTORY* fixes the set of factory functions *new_button* and *new_box*

Abstract factory – Componentization

Fully componentizable

Abstract factory library (1/2)

```
class
    FACTORY [G]
create
        make
feature -- Initialization
        make (f : like factory_function)
                        -- Initialize with factory_function set to f.
                require
                        exists: f /= Void
                do
                        factory_function := f
                end
feature -- Access

        factory_function : FUNCTION [ANY, TUPLE [], G ]
                -- Factory function creating new instances of type G
```

Abstract factory library (2/2)

The Factory Library can create only one kind of product

```
feature -- Factory operations
        new: G
                        -- New instance of type G
                do
                        factory_function.call ([])
                        Result := factory_function.last_result
                ensure
                        exists: Result /= Void
                end

        new_with_args (args: TUPLE): G
                        -- New instance of type G initialized with args
                do
                        factory_function.call (args)
                        Result := factory_function.last_result
                ensure
                        exists: Result /= Void
                end
invariant
        exists: factory_function /= Void
end
```

With the Factory Library (1/2)

```
deferred class
        APPLICATION
...
feature -- Initialization
        build_window
                -- Build window.
            local
                window: WINDOW
            do
                window := window_factory.new
                ...
            end
feature {NONE} -- Implementation
        window_factory: FACTORY [WINDOW]
        button_factory: FACTORY [BUTTON]
        menu_bar_factory: FACTORY [MENU_BAR]
...
end
```

Use several factory objects to create several products

With the Factory Library (2/2)

```
class
        WEL_APPLICATION
inherit
        APPLICATION
create
        make
feature
        make
                        -- Create factories.
            do
                create {FACTORY [WEL_WINDOW]} window_factory.make (…)
                create {FACTORY [WEL_BUTTON]} button_factory.make (…)
                create {FACTORY [WEL_MENU_BAR]} menu_bar_factory.make (…)
            end
…
end
```

- Client must make sure that all factories are configured to create Windows widgets
- More error-prone with several factories

However, the problem already existed in the Abstract Factory pattern; it is concentrated in class *WINDOW_FACTORY*

Factory library vs. factory pattern

Advantages of the library:

- Get rid of some code duplication
- Fewer classes
- Reusability

Limitations of the library:

- Likely to yield a bigger client class (because similarities cannot be factorized through inheritance)

Factory method vs. abstract factory

Factory method:

- Creates one object
- Works at routine level
- Helps a class perform an operation, which requires creating an object

Abstract factory:

- Creates families of object
- Works at class level
- Uses factory methods (e.g. features *new* and *new_with_args* of the Factory Library are factory methods)

Design patterns (GoF)

Creational
- ✓ Abstract Factory
- • Singleton
- ✓ Factory Method
- • Builder
- • Prototype

Structural
- • Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- • Proxy

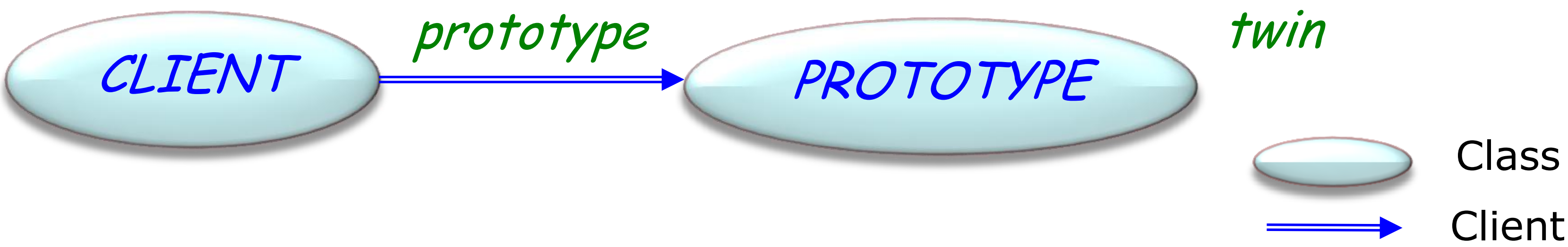
Behavioral
- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- • Interpreter
- • Iterator
- • Mediator
- • Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- • Template Method
- ✓ Visitor

Non-GoF patterns
- ✓ Model-View-Controller

Prototype pattern

Intent:

> "*Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.*" [Gamma 1995]



No need for this in Eiffel: just use function *twin* from class *ANY*.

y := *x*.*twin*

In Eiffel, every object is a prototype

Cloning in Java, C#, and Eiffel

Java

Class must implement the interface Cloneable defining clone (to have the right to call clone defined in Object)

C#

Class must implement the interface ICloneable defining Clone (to have the right to call MemberwiseClone defined in Object)

Next version of Eiffel

Class must broaden the export status of *clone*, *deep_clone* inherited from *ANY* (not exported in *ANY*)

Design patterns (GoF)

Creational

- ✓ Abstract Factory
- • Singleton
- ✓ Factory Method
- • Builder
- ✓ Prototype

Structural

- • Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- • Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- • Interpreter
- • Iterator
- • Mediator
- • Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- • Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Builder pattern

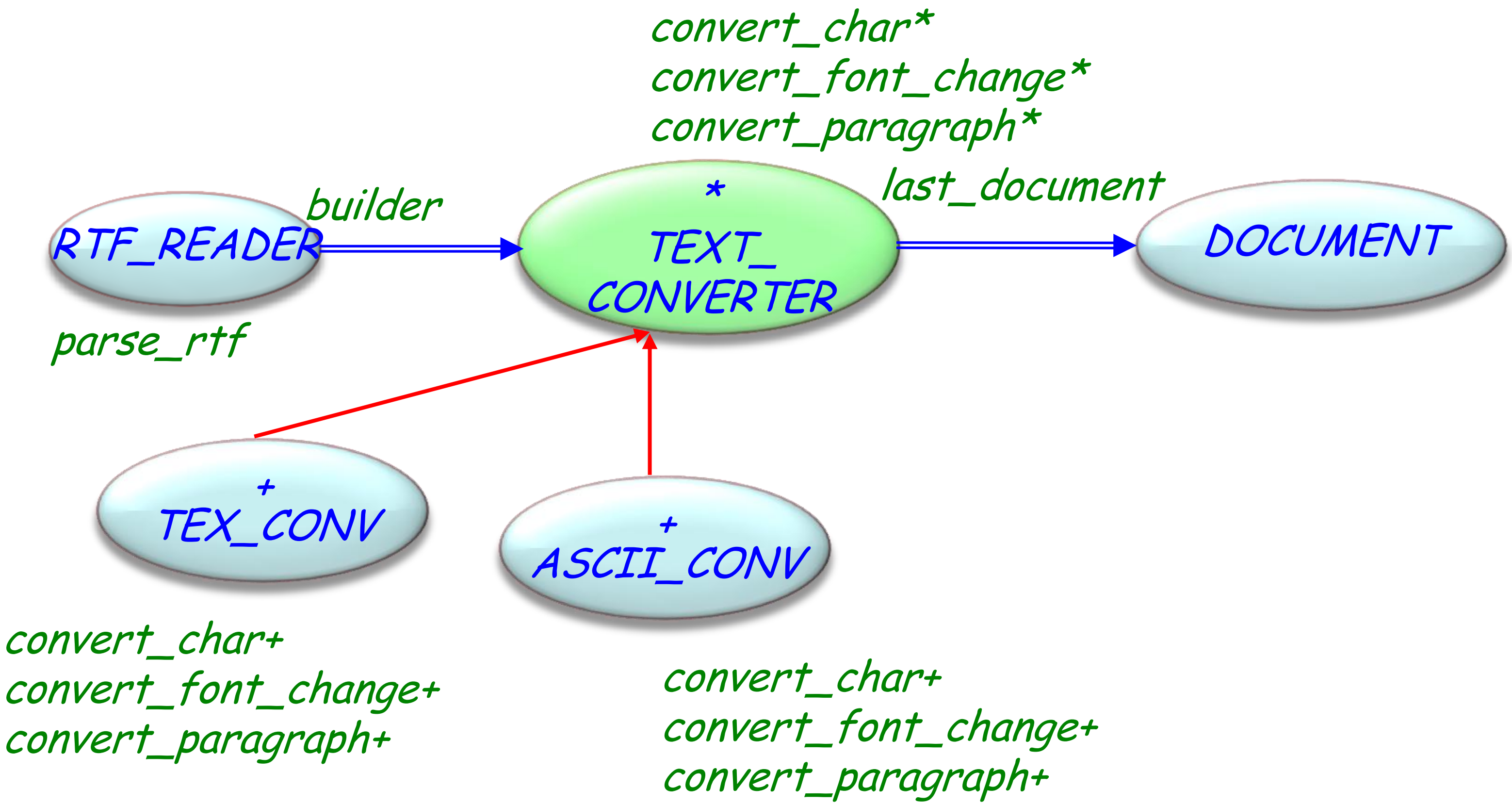
Intent:

"*Separate the construction of a complex object from its representation so that the same construction process can create different representations*"

(Gamma et al.)

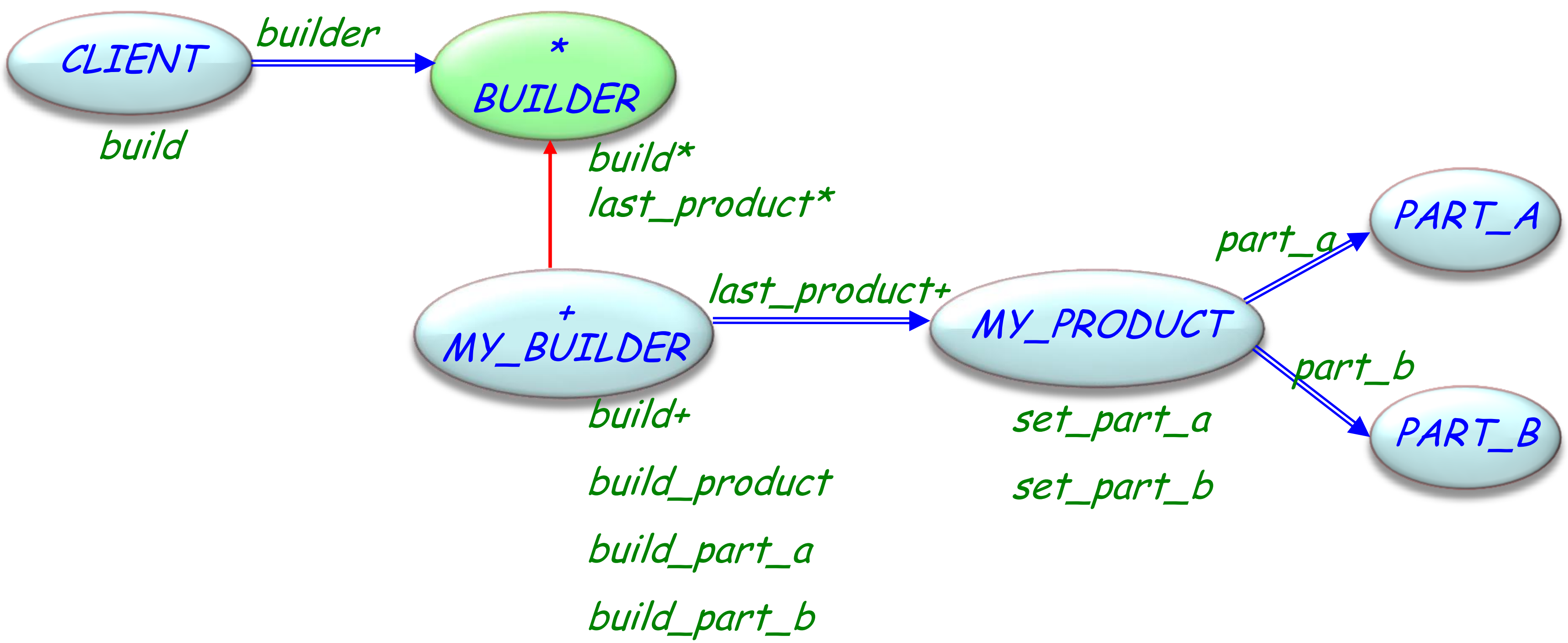
Example use: build a document out of components (table of contents, chapters, index…) which may have some variants.

RTF example



convert_char*
convert_font_change*
convert_paragraph*
builder
RTF_READER
*
TEXT_
CONVERTER
last_document
DOCUMENT
parse_rtf
+
TEX_CONV
+
ASCII_CONV
convert_char+
convert_font_change+
convert_paragraph+
convert_char+
convert_font_change+
convert_paragraph+

Builder pattern



Builder Library

```
deferred class
    BUILDER [G]
feature -- Access
    last_product : G
            -- Product under construction
        deferred
        end
feature -- Status report
    is_ready : BOOLEAN
            -- Ready to build last_product ?
        deferred
        end
feature -- Basic operations
    build
            -- Build last_product.
        require
            is_ready: is_ready
        deferred
        ensure
            last_product_exists: last_product /= Void
        end
end
```

Mechanisms enabling componentization: unconstrained genericity, agents

+ Factory Library

Two-part builder

class

TWO_PART_BUILDER [*F* -> *BUILDABLE*, *G*, *H*]

-- *F*: type of product to build
-- *G*: type of first part of the product
-- *H*: type of second part of the product

The builder knows the type of product to build and number of parts

In the original Builder pattern:

Deferred builder does not know the type of product to build
Concrete builders know the type of product to build

TWO_PART_BUILDER is a concrete builder

$\Rightarrow$ compatible with the pattern

Example using a two-part builder

```
class
        APPLICATION
create
        make
feature {NONE} -- Initialization
        make is
                        -- Build a new two-part product with a two-part builder.
                local
                        my_builder: TWO_PART_BUILDER [TWO_PART_PRODUCT,
                                                        PART_A, PART_B]
                        my_product: TWO_PART_PRODUCT
                do
                        create my_builder.make (agent new_product, agent new_part_a,
                                                agent new_part_b)
                        my_builder.build_with_args (["Two-part product"],["Part A"],["Part B"])
                        my_product := my_builder.last_product
                end
feature -- Factory functions
        new_product (a_name: STRING): TWO_PART_PRODUCT do ...
        new_part_a (a_name: STRING): PART_A do ...
        new_part_b (a_name: STRING): PART_B do ...
end
```

Two-part builder (1/4)

```
class interface
        TWO_PART_BUILDER [F -> BUILDABLE, G, H]
inherit
        BUILDER [F]
create
        make
feature {NONE} -- Initialization
        make (f: like factory_function_f; g: like factory_function_g;
                h: like factory_function_h)
                        -- Set factory_function_f to f. Set factory_function_g to g.
                        -- Set factory_function_h to h.
                require
                        f_not_void: f /= Void
                        g_not_void: g /= Void
                        h_not_void: h /= Void
                ensure
                        factory_function_f_set: factory_function_f = f
                        factory_function_g_set: factory_function_g = g
                        factory_function_h_set: factory_function_h = h
feature -- Access
        last_product: F
                -- Product under construction
```

Two-part builder (2/4)

```
feature -- Status report
	is_ready: BOOLEAN
			-- Is builder ready to build last_product?
	valid_args (args_f, args_g, args_h: TUPLE): BOOLEAN
			-- Are args_f, args_g and args_h valid arguments to
			-- build last_product?

feature -- Basic operations
	build
			-- Build last_product. (Successively call build_g and
			-- build_h to build product parts.)
		do
			last_product := f_factory.new
			build_g ([])
			build_h ([])
		ensure then
			g_not_void: last_product.g /= Void
			h_not_void: last_product.h /= Void
		end
```

Two-part builder (3/4)

```
        build_with_args (args_f, args_g, args_h: TUPLE)
                        -- Build last_product with args_f. (Successively
                        -- call build_g with args_g and build_h with
                        -- args_h to build product parts.)
                require
                        valid_args: valid_args (args_f, args_g, args_h)
                ensure
                        g_not_void: last_product.g /= Void
                        h_not_void: last_product.h /= Void

feature -- Factory functions
        factory_function_f: FUNCTION [ANY, TUPLE, F]
                        -- Factory function creating new instances of type F
        factory_function_g: FUNCTION [ANY, TUPLE, G]
                        -- Factory function creating new instances of type G
        factory_function_h: FUNCTION [ANY, TUPLE, H]
                        -- Factory function creating new instances of type H
```

Two-part builder (4/4)

```
feature {NONE} -- Basic operations
        build_g (args_g: TUPLE) do ...
        build_h (args_h: TUPLE) do ...

feature {NONE} -- Factories
        f_factory: FACTORY [F]
                        -- Factory of objects of type F
        g_factory: FACTORY [G]
                        -- Factory of objects of type G
        h_factory: FACTORY [H]
                        -- Factory of objects of type H
invariant
        factory_function_f_not_void: factory_function_f /= Void
        factory_function_g_not_void: factory_function_g /= Void
        factory_function_h_not_void: factory_function_h /= Void
        f_factory_not_void: f_factory /= Void
        g_factory_not_void: g_factory /= Void
        h_factory_not_void: h_factory /= Void
end
```

Builder Library using factories?

> Very flexible because one can pass any agent as long as it has a matching signature and creates the product parts

```
class
        TWO_PART_BUILDER [F -> BUILDABLE, G, H]
inherit
        BUILDER [F]
...
feature -- Factory functions
        factory_function_f: FUNCTION [ANY, TUPLE, F]
                        -- Factory function creating new instances of type F
        factory_function_g: FUNCTION [ANY, TUPLE, G]
                        -- Factory function creating new instances of type G
        factory_function_h: FUNCTION [ANY, TUPLE, H]
                        -- Factory function creating new instances of type H
feature {NONE} -- Implementation
        build_g (args_g: TUPLE) is
                        -- Set last_product.g with a new instance of type G created with
                        -- arguments args_g.
                do
                        last_product.set_g (g_factory.new_with_args (args_g))
                        ...
                end
...
end
```

Builder Library: completeness?

Supports builders that need to create two-part or three-part products

Cannot know the number of parts of product to be built in general

$\Rightarrow$ Incomplete support of the Builder pattern ("Componentizable but non-comprehensive")

Design patterns (GoF)

Creational

- ✓ Abstract Factory
- • Singleton
- ✓ Factory Method
- ✓ Builder
- ✓ Prototype

Structural

- • Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- • Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- • Interpreter
- • Iterator
- • Mediator
- • Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- • Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

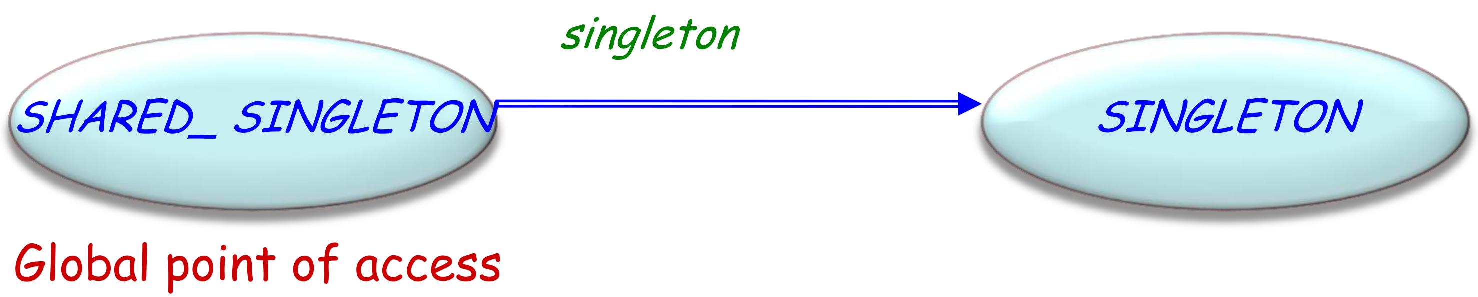
Singleton pattern

Intent:

Way to “ensure a class only has one instance, and to provide a global point of access to it.” [Gamma et al.]

Singleton pattern

Way to "ensure a class only has one instance, and to provide a global point of access to it." [GoF, p 127]



Singletons in Eiffel

Once routines

But: does not prevent cloning

Once routines

If instead of

```
r
        do
                ... Instructions ...
        end
```

you write

```
r
        once
                ... Instructions ...
        end
```

then *Instructions* will be executed only for the first call by any client during execution. Subsequent calls return immediately.

In the case of a function, subsequent calls return the result computed by the first call.

Scheme for shared objects

```
class MARKET_INFO feature
  Christmas: DATE
      once
          create Result.make (...)
      end
   off_days: LIST [DATE]
      once
          create Result.make (...)
          Result.extend (Christmas)
          ...
      end
      ...
end
```

Will always return the same instance for all instances of MARKET_INFO (also descendant instances)
→ Provides global point of access

```
class APPLICATION_CLASS inherit
      MARKET_INFO
feature
    r
           do
                    print (off_days)
                    ...
           end
    ...
end
```

Ensuring the existence of only one instance

Cloning:

Class *ANY* has features *clone* (*twin*), *deep_clone*, …

One can duplicate any Eiffel object, which rules out the Singleton pattern

clone, *deep_clone*, … will be exported to *NONE* in the next version of Eiffel

Ensuring the existence of only one instance

Exporting creation procedure:

Creation procedure of *SINGLETON* should not be exported to any other than the *SHARED_SINGLETON* class:

```
class SINGLETON
create {SHARED_SINGLETON} default_create
end
```

Ensures that no other classes can create instances

But: Descendants of SHARED_SINGLETON may change the export status and clone it!

Ensuring the existence of only one instance

Prohibit classes to inherit from *SHARED_SINGLETON*:

Make SHARED_SINGLETON **frozen**

Frozen means:

- Class that may not have any descendant
- Marked by a keyword **frozen**
- A class cannot be both **frozen** and **deferred**

Advantages:

Straightforward way to implement singletons

No problem of different once statuses

Compilers can optimize code of frozen classes

Weakness:

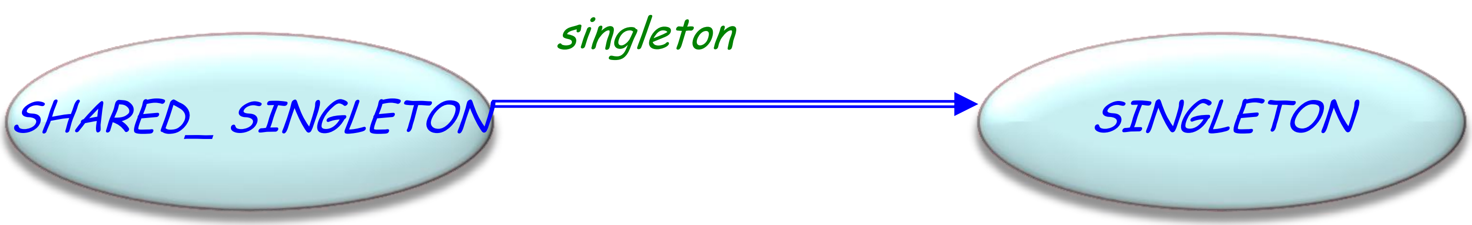
Goes against the *Open-Closed principle*

Singleton with frozen classes

```
frozen class
	SHARED_SINGLETON
feature -- Access
	singleton: SINGLETON is
			-- Global access point to singleton
		once
			create Result
		ensure
			singleton_not_void: Result /= Void
		end
end
```

```
class
	SINGLETON

create {SHARED_SINGLETON}

	default_create

end
```

Singleton in Eiffel – The four ingredients



- **once** feature for creating *SINGLETON*
- **frozen** class (prohibit inheritance)

- allow creation only to *SHARED_SINGLETON* instances
- no *copy*/ *clone* features available to clients

But: currently **once** is once-per-thread (multi-threading will break the guarantee)

Singleton without frozen classes

Frozen classes require the ability to restrict the exportation of creation procedures (constructors)

$\Rightarrow$ Not applicable in *C++*, Java or *C#*

C++, Java and *C#* use static features to implement the singleton pattern

Singletons in C++/Java/C#

Static classes

Making SINGLETON a static class is not enough:

- Multiple declarations of a static object are possible (no global point of access)
- Static classes are initialized at initialization time (which varies according to the details of the language), but the initialization of SINGLETON may require a later initialization at some precise point during the program's execution
- If multiple SINGLETON classes exist, it may be impossible to implement a particular initialization order among them

Singletons in C++/Java/C#

A more flexible solution uses a (non-static) Singleton class with hidden constructor, accessed only through a public static method Instance to retrieve the real singleton

Compared with the class diagram seen before, this solution coalesces SINGLETON and SHARED_SINGLETON

Similar results can be obtained by hiding the declaration of SINGLETON inside SHARED_SINGLETON

Singletons in Java

```
class Singleton {

    public static Singleton Instance() {
        if (_instance == null) { _instance = new Singleton(); }
        return _instance;
    }

    protected Singleton() {
        // ...
    }

    private static Singleton _instance = null;
}
```

Creational patterns - Discussion

- Abstract the creation process
- Make system independent of how objects are created, composed and represented

Creational patterns become important as systems evolve

Two recurring themes:

- encapsulate knowledge about concrete classes used
- hide how instances are created and composed

Freedom: *What* specific instances get created, *who* creates instances, *how* they get created and *when*.

Design patterns (GoF)

Creational

- ✓ Abstract Factory
- ✓ Singleton
- ✓ Factory Method
- ✓ Builder
- ✓ Prototype

Structural

- Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Adapter pattern

Intent: "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

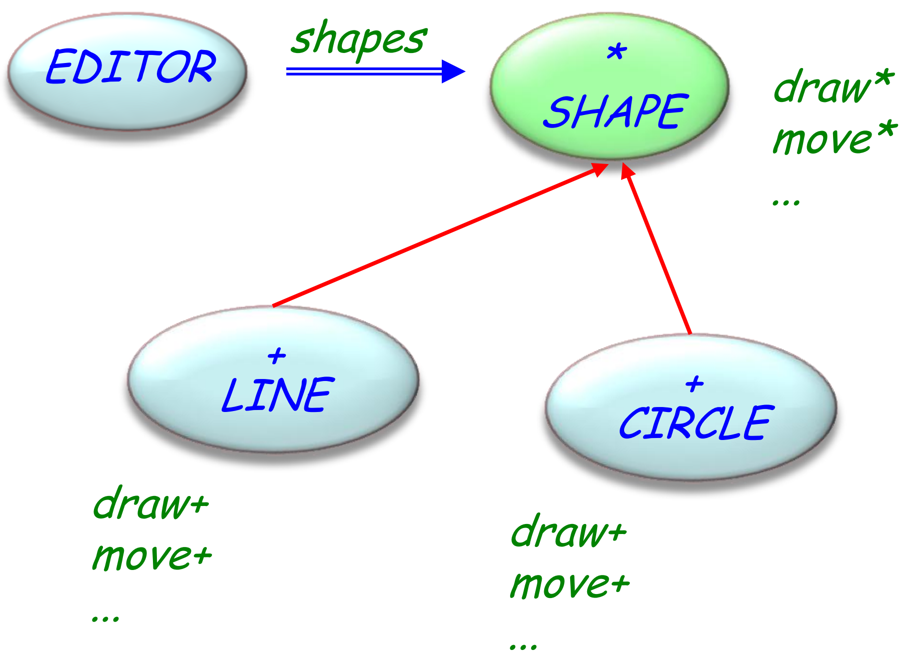
Adapters are also called wrappers.

Motivation: Reuse available components through a different interface.

Example: integrating different components

You want to extend a graphical editor to support the manipulation and visualization of text elements.

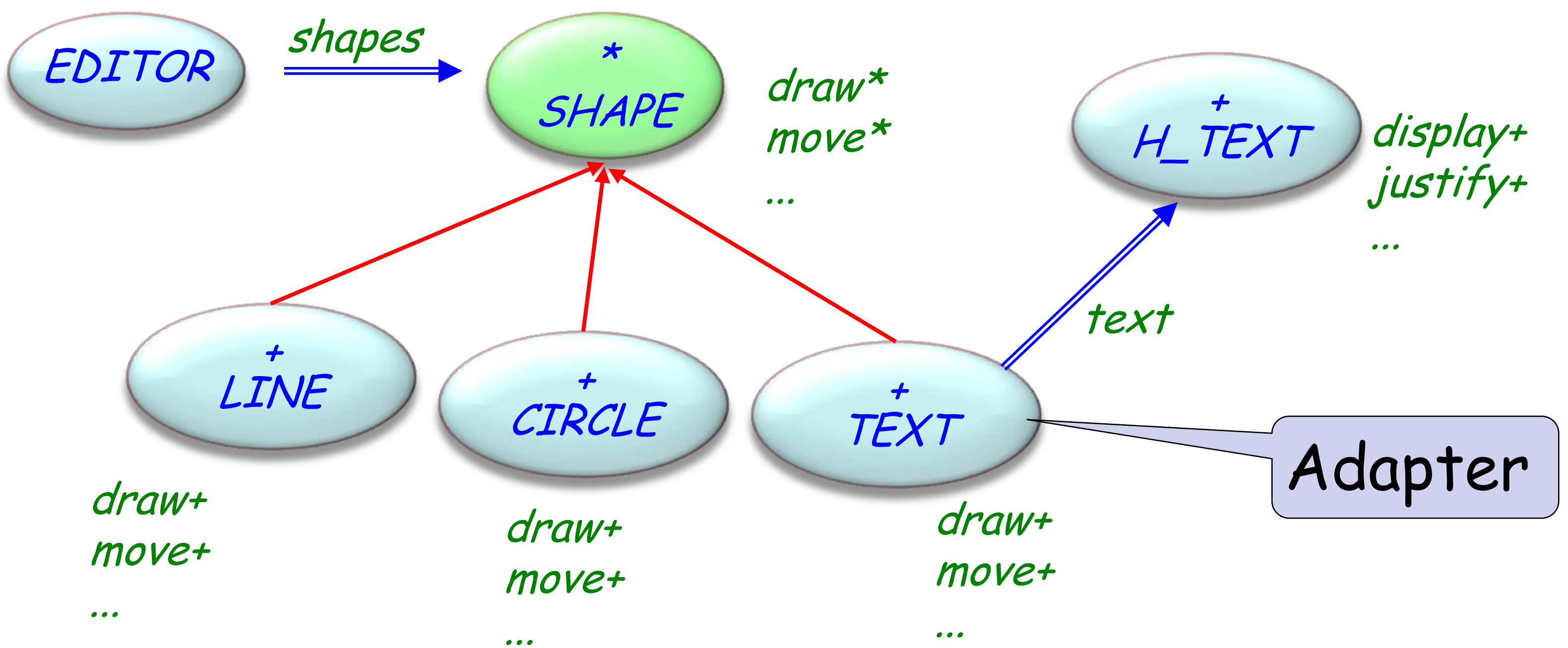
The current implementation relies on a class hierarchy based on the abstraction of shape:



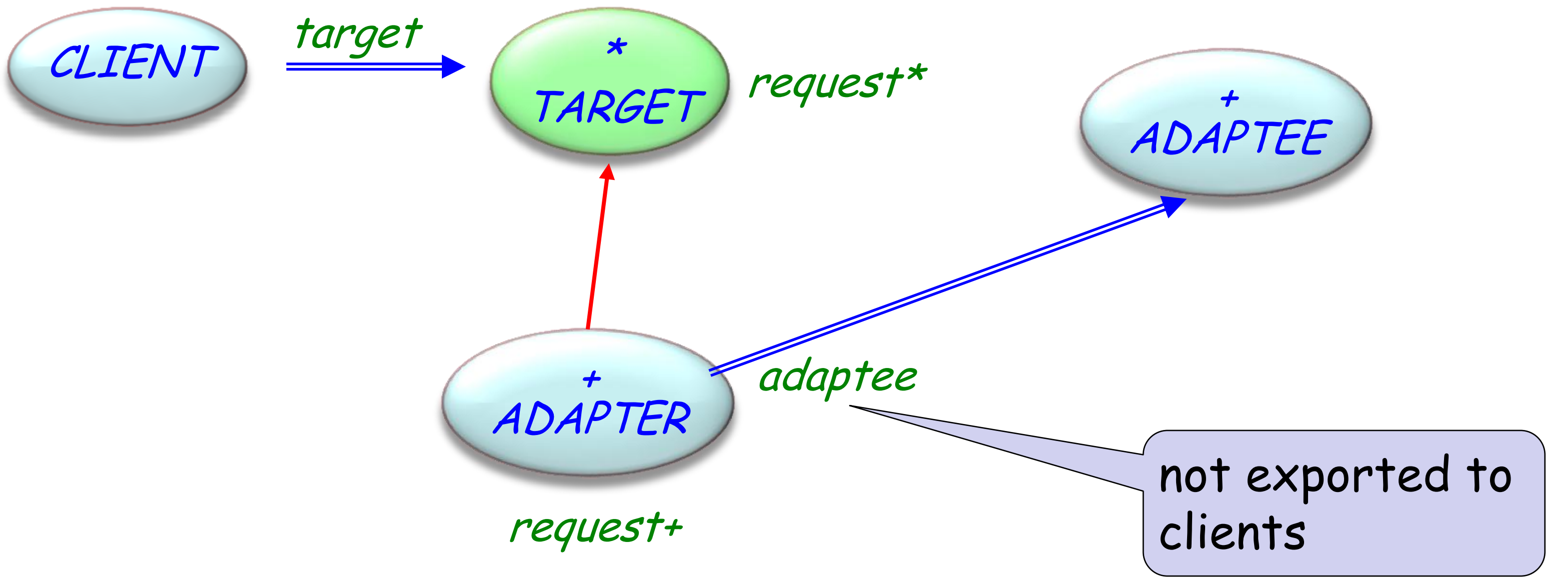
Example: integrating different components

You want to extend a graphical editor to support the manipulation and visualization of text elements.

A class TEXT provides the services by adapting to the SHAPE interface an available implementation H_TEXT

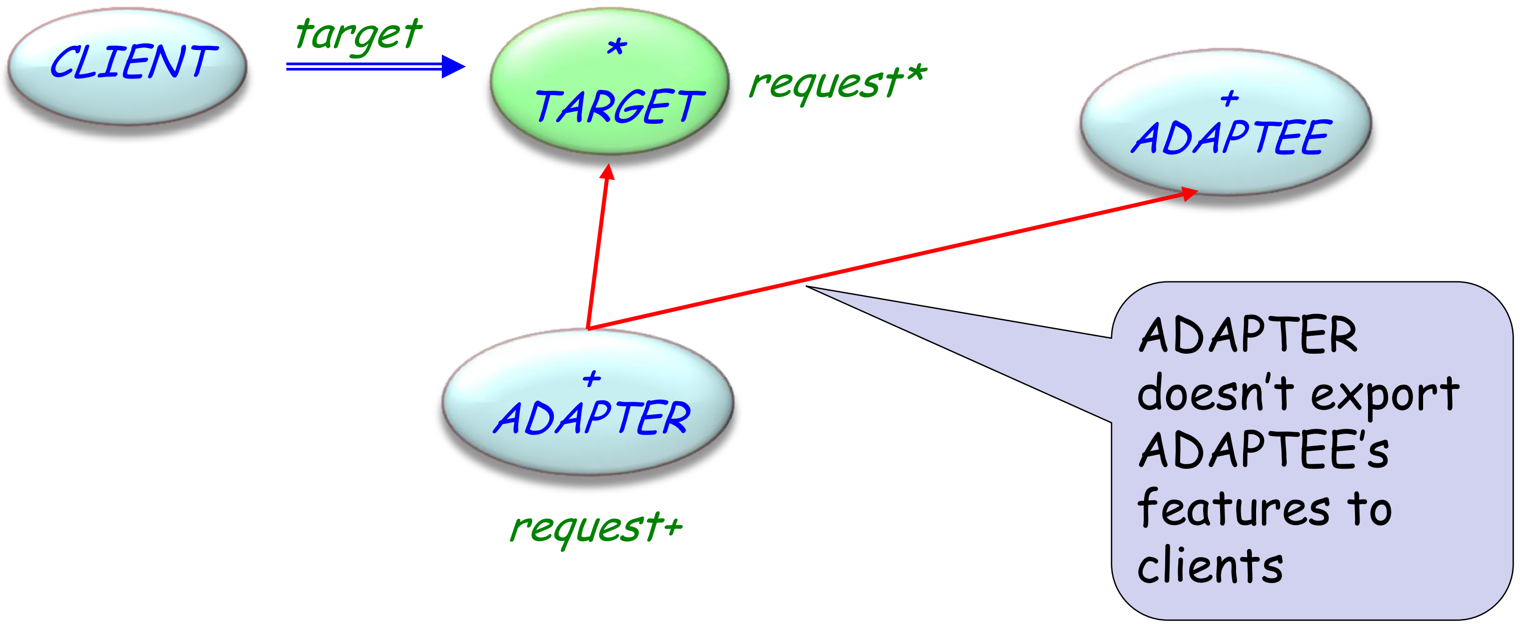


Adapter pattern: object variant



This version of the pattern is called object adapter, because ADAPTER uses an instance of ADAPTEE

Adapter pattern: class variant



This version of the pattern is called class adapter, because ADAPTER inherits from ADAPTEE to adapt its services

Adapter pattern: participants

Target

defines the (specific) interface used by CLIENT

Client

uses objects conforming to
the interface of TARGET

Adaptee

offers services through an existing interface
that needs adapting

Adapter

adapts the ADAPTEE's interface to the TARGET's

Design patterns (GoF)

Creational

- ✓ Abstract Factory
- ✓ Singleton
- ✓ Factory Method
- ✓ Builder
- ✓ Prototype

Structural

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- • Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- • Interpreter
- • Iterator
- • Mediator
- • Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- • Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Proxy patter

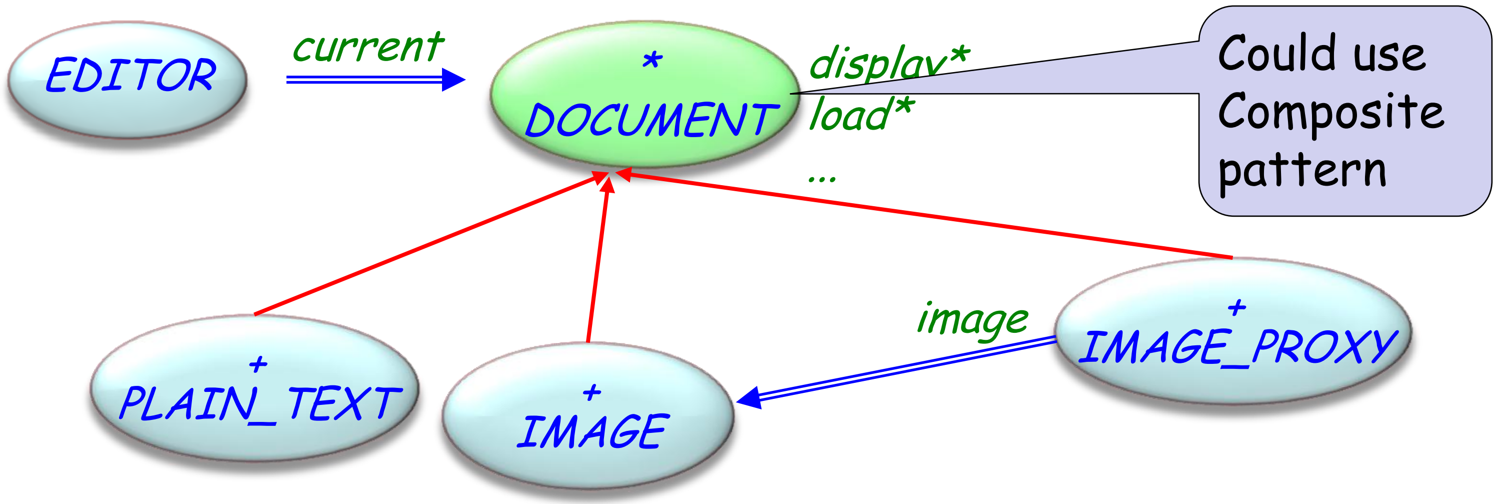
Intent: "Provide a surrogate or placeholder for another object to control access to it."

Motivation: Controlling when the various parts of an object are created – for example to delay creation of the most expensive parts until when they are actually needed.

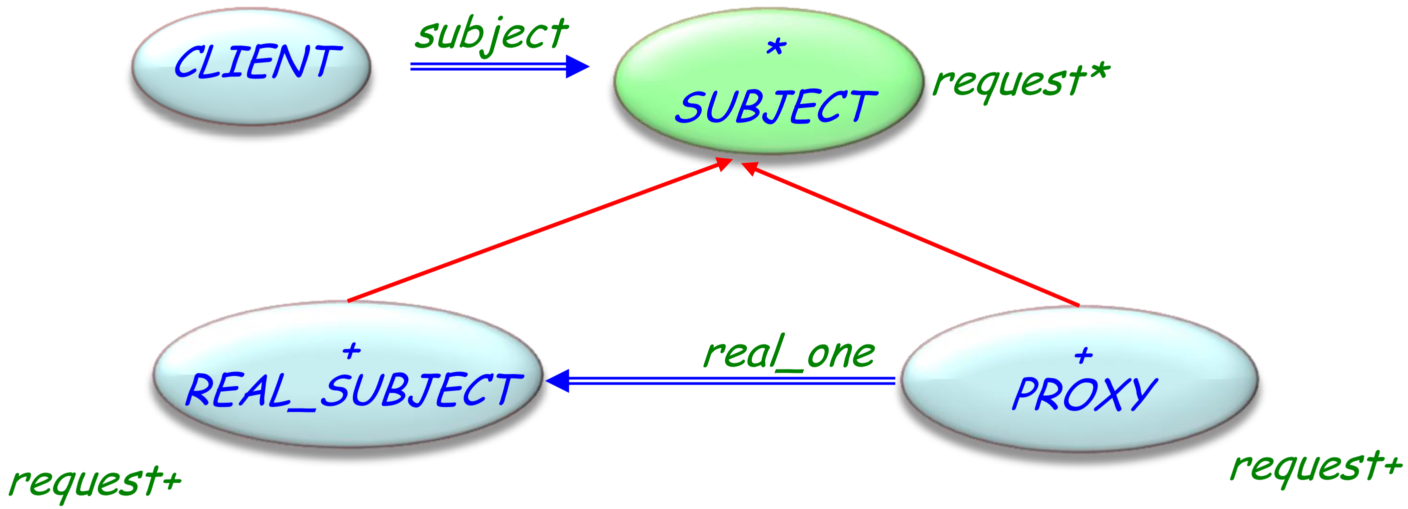
Example: a document editor

A document editor uses a class DOCUMENT that encapsulates all data about an open document.

If a new document includes large bitmap images, opening it takes time unless the creation of the objects for the images is postponed to when it is actually needed (e.g., when the client wants to display a page with images).



Proxy pattern



Proxy class: implementation

```
class PROXY
inherit SUBJECT
feature
        request
                do
                        if not attached real_one then
                                create {REAL_PROXY} real_one
                        end
                        real_one.request
                end
feature {PROXY}
        real_one: SUBJECT
end
```

Proxy patterns: participants

Proxy

- Maintains a reference to access REAL_SUBJECT
- Provides an interface identical to SUBJECT's
- Controls access to REAL_SUBJECT
  (the control policy is application dependent)

Subject

- Defines a common interface for REAL_SUBJECT and PROXY so that a PROXY can replace a REAL_SUBJECT

Real Subject

- Defines the real object that PROXY represents

Types of proxy

Remote proxy

- The real subject is in a different physical or logical location
- The proxy is responsible for sending requests
- Decoupling between client and actual provider

Virtual proxy

- Mediate object creation
- Provide caching and sharing (as in the example)

Protection proxy

- Authorize or reject access to the real object according to the permissions of the client

Design patterns (GoF)

Creational
- ✓ Abstract Factory
- ✓ Singleton
- ✓ Factory Method
- ✓ Builder
- ✓ Prototype

Structural
- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

Behavioral
- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- Template Method
- ✓ Visitor

Non-GoF patterns
- ✓ Model-View-Controller

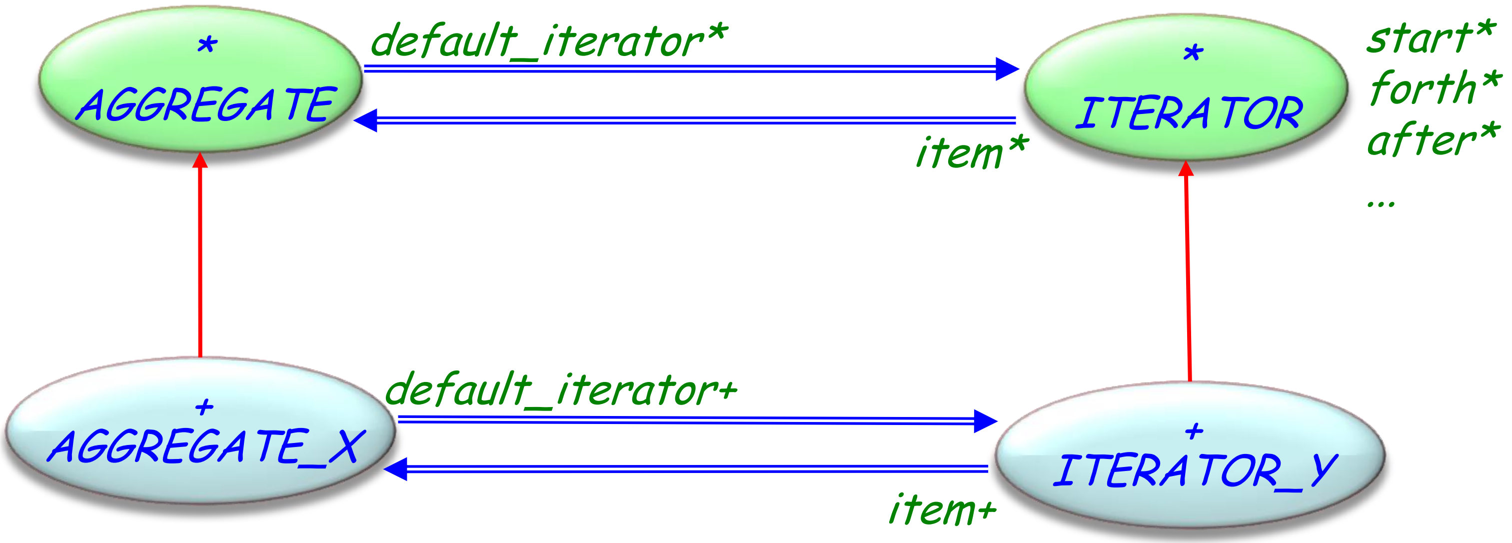
Iterator pattern

Intent: "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."

Motivation: decouple different types of "sequentialization" routines from the interface of the aggregate object.

Example: a tree data structure, with different iterators providing pre-order, post-order, in-order, and breadth-first trasversals.

Iterator pattern



Iterator pattern: participants

Iterator

- Defines an interface for accessing and traversing elements

Concrete iterator

- Implements the actual traversal algorithm

Aggregate

- Provides a default iterator in the interface

Concrete aggregate

- Is linked to a concrete iterator as default
- Makes it possible to implement certain trasversals

Iterator pattern: features

- Different traversals of the same aggregate
  - Adding new traversals does not change the interface of aggregates

  → A cursor is the simplest form of an iterator, which only maintains a reference to the current element. The client defines its own traversal algorithm using the other features of the iterator.

- Several iterators can traverse the same aggregate simultaneously

- The features of a default iterator can be included in the aggregate's interface

  → This is done extensively in EiffelBase

Design patterns (GoF)

Creational

- ✓ Abstract Factory
- ✓ Singleton
- ✓ Factory Method
- ✓ Builder
- ✓ Prototype

Structural

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- • Interpreter
- ✓ Iterator
- • Mediator
- • Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- • Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Template method pattern

Intent: "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."

A template method is similar to pseudo-code, where the deferred operations are refined by effecting (implementation) in subclasses.

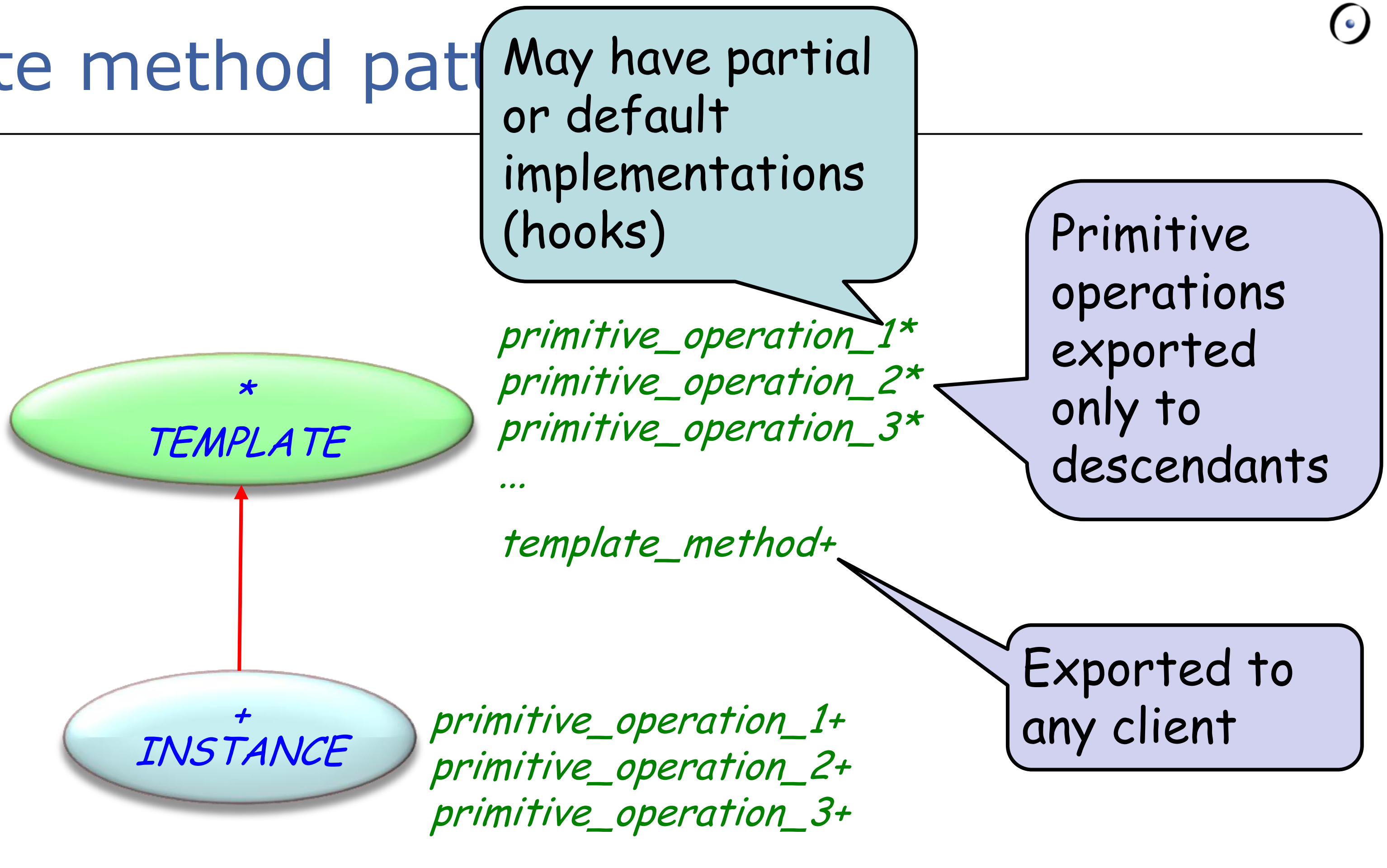
Example: two-player games (1/2)

```
deferred class GAME

feature {GAME} -- Deferred operations
        initialize deferred end -- initialize the game
        play_one deferred end -- player one moves
        play_two deferred end -- player two moves

feature {ANY} -- Status
        done: BOOLEAN
        winner: BOOLEAN -- True iff player one has won
                require game_over: done
                attribute end
```

Example: two-player games (2/2)

```
feature {ANY} -- template method
        play_until_winner
                -- play until somebody wins
                require not_over: not done
                local turn: INTEGER
                do
                        from initialize
                        until done
                        loop
                                if turn.is_even then play_one
                                else play_two end
                                turn := turn + 1
                        end
                        if turn.is_even then winner := False
                        else winner := True end
                ensure game_over: done
                end
```

Template method patt…



Template method pattern: when to use

To implement the invariant parts of an algorithm

To factor out common behavior among subclasses and avoid code duplication

"refactoring to generalize"

To control behavior of subclasses: only primitive operations should be implemented or redefined

→ **frozen** routines in Eiffel

Template method: componentizability

Classes with template methods can be implemented as components

- primitive operations provided as agents
- disadvantage: fewer static checks of complete implementations

Design patterns (GoF)

Creational

- ✓ Abstract Factory
- ✓ Singleton
- ✓ Factory Method
- ✓ Builder
- ✓ Prototype

Structural

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- • Interpreter
- ✓ Iterator
- • Mediator
- • Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template Method
- ✓ Visitor

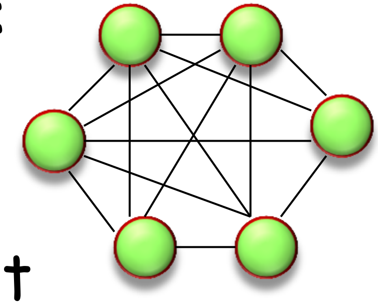
Non-GoF patterns

- ✓ Model-View-Controller

Mediator pattern

Intent: "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."

Motivation: OO design encourages distribution of behavior among objects. Strong distribution:



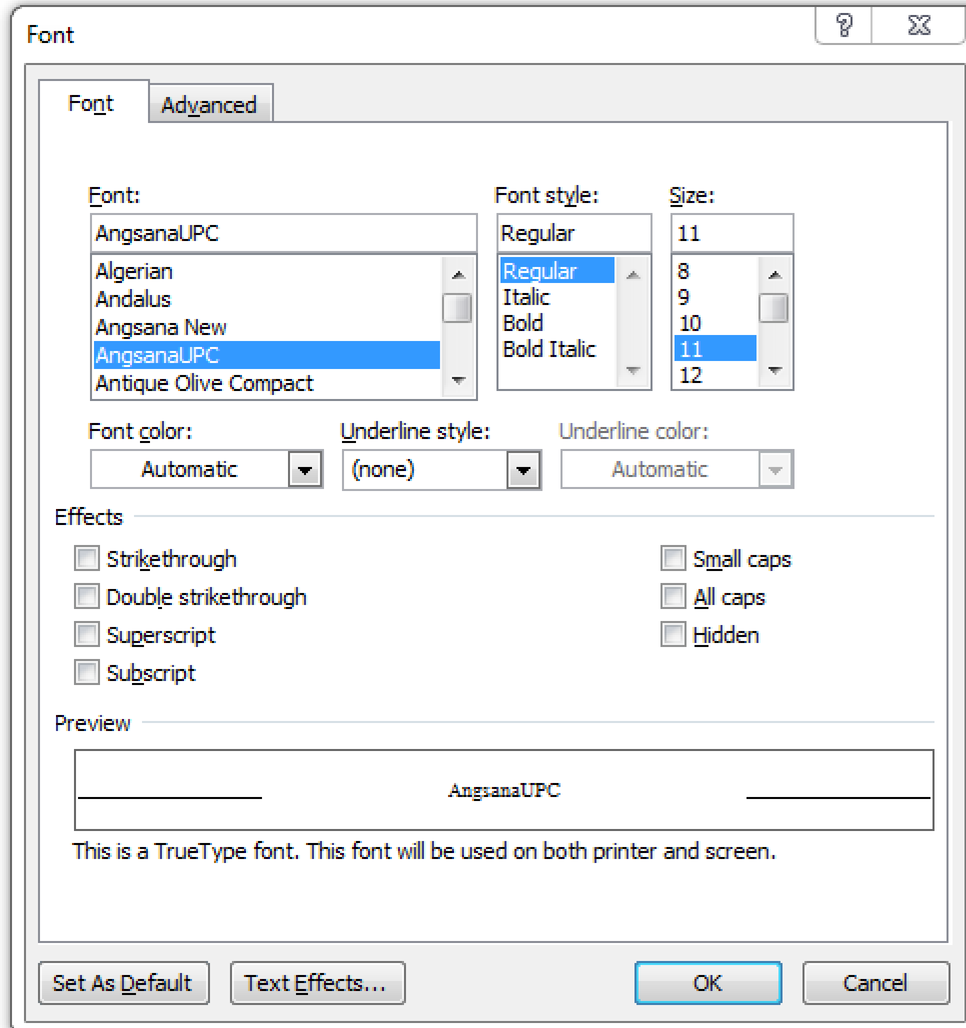
- Can result in structure with many connections between objects
- Objects less likely to work without support of other objects
- More difficult to change system's behavior significantly, since behavior distributed

Mediator pattern: Example

Example:

- Dialog box presents collection of widgets

- Dependencies between widgets (fonts have different styles and sizes; Check boxs are dependent)



Mediator pattern: Example

- Different dialog boxes have different dependencies between widgets
  - Cannot simply reuse stock widget classes
  - Customizing (through subclassing) could be tedious since many classes are involved

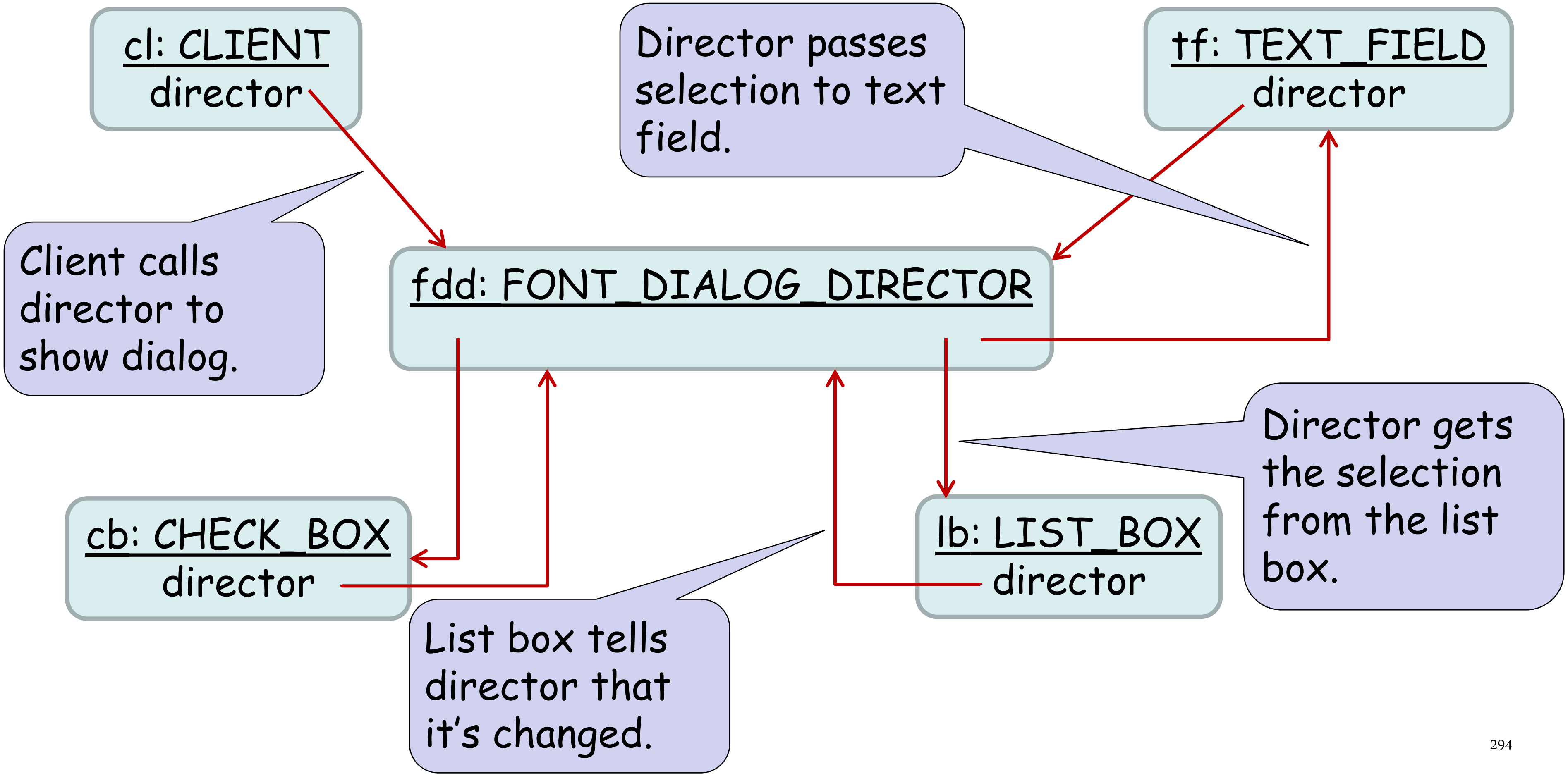
- Avoid these problems by encapsulating collective behavior in a separate **mediator** object

A **mediator** serves as an intermediary that keeps objects in a group from referring to each other explicitly.

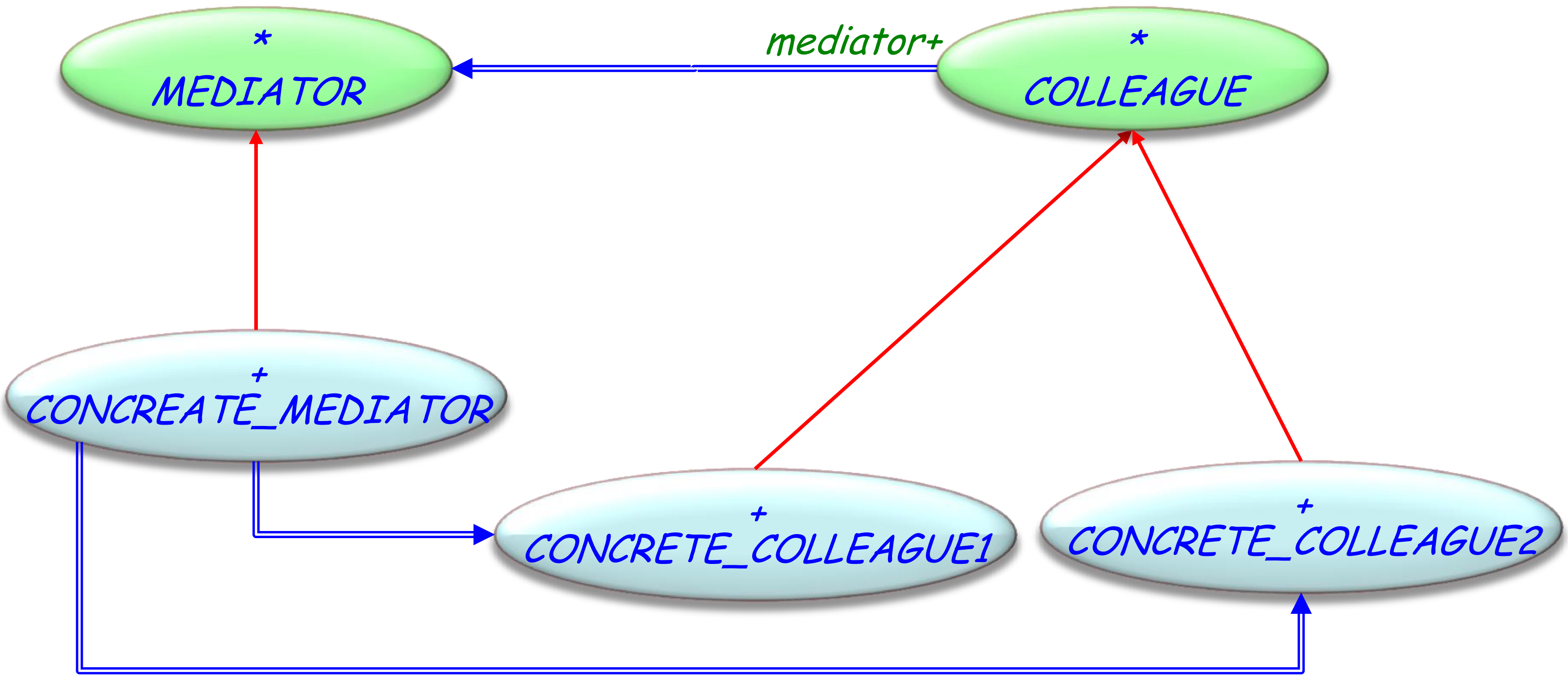
The objects only know the mediator, thereby reducing the number of interconnections.

Mediator pattern: Example

- Mediator acts as a hub of communication for widgets



Mediator pattern: Structure



Mediator pattern: participants

MEDIATOR

- Defines an interface for communicating with COLLEAGUE objects

CONCRETE_MEDIATOR

- Implements cooperative behavior by coordinating COLLEAGUE objects
- Knows and maintains colleagues

COLLEAGUE classes

- Each COLLEAGUE class knows its MEDIATOR object
- Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Mediator pattern: when to use

Use the Mediator pattern when

- Objects communicate in well-defined but complex ways → Resulting dependencies are unstructured and difficult to understand

- Object reuse is difficult because it refers to / communicates with many other objects

- Behavior distributed over several classes should be customizable without a lot of subclassing

Design patterns (GoF)

Creational

- ✓ Abstract Factory
- ✓ Singleton
- ✓ Factory Method
- ✓ Builder
- ✓ Prototype

Structural

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- • Interpreter
- ✓ Iterator
- ✓ Mediator
- • Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Memento pattern

Intent: "Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later."

Motivation: want to record internal state of an object (e.g. as checkpoint or for undo). Objects normally encapsulate some or all of their state; exposing it would violate encapsulation, thus compromising reliability and extensiblity of the application.

Memento pattern: Example

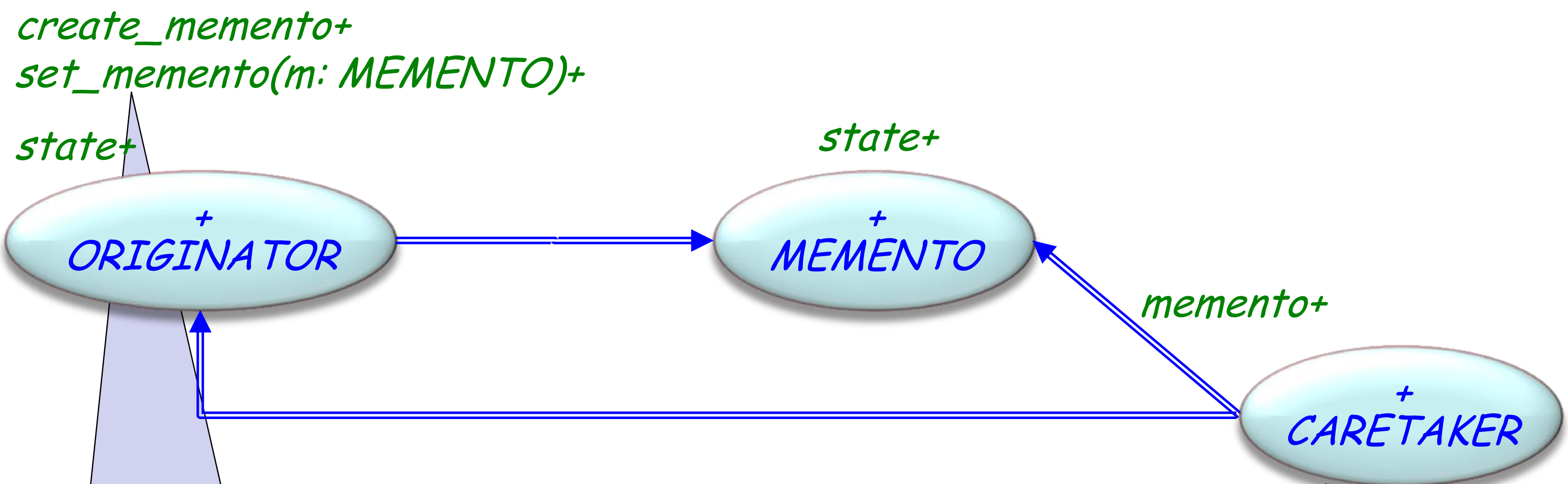
Example

- An object stores form information
- We allow users to make changes to values in the form
- In case of a mistake, users can revert to the previous values in the form.

Instead of exposing all information of the form object, the form object offers a mechanism to store its state
→ it allows for the creation of a **memento** object.

> A **memento** is an object that stores a snapshot of another object – the memento's **originator**.

Memento pattern: Structure



set_memento restores the originator's state based on the information stored in MEMENTO object *m*.

Caretaker calls create_memento before changing originator; stores resulting MEMENTO object.

Memento pattern: participants

MEMENTO

- Stores internal state of the ORIGINATOR object
- Protects against access by objects other than the originator
  - CARETAKER sees *narrow* interface – can only pass the memento to other objects
  - Originators sees *wide* interface – allows access to all data necessary to restore the state

ORIGINATOR

- Creates a memento containing a snapshot of its current internal state
- Uses the memento to restore its internal state

CARETAKER

- Responsible for the memento's safekeeping
- Never operates on or examines the contents of a memento

Memento pattern: when to use

Use the Memento pattern when

- A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later,

and

- A direct interface to obtaining the state would expose implementation details and break the object's encapsulation

Design patterns (GoF)

Creational

- ✓ Abstract Factory
- ✓ Singleton
- ✓ Factory Method
- ✓ Builder
- ✓ Prototype

Structural

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

Behavioral

- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- • Interpreter
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template Method
- ✓ Visitor

Non-GoF patterns

- ✓ Model-View-Controller

Interpreter pattern

Intent: "Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language."

Motivation: if sentences of a simple language occur often enough, it might be worthwhile to build an interpreter for them

Example: check whether a string matches a regular expression

String: *dog dog cat weather*

Reg. expr.: ((` *dog* '`|` *cat* `)* & ` *weather* `

Interpreter pattern: Example

- A grammar for regular expressions:

```
experssion      ::=    literal | alternation | sequence | repetition |
                       `(`expression`)`
alternation     ::=    expression `|` expression
sequence        ::=    expression `&` expression
repetition      ::=    expression `*`
literal         ::=    `a` | `b` | `c` | ... { `a` | `b` | `c` | ... }*
```

Start symbol: `expression`   Terminal symbol: `literal`

- Given inputs
  - regular expression (as an AST)
  - a string

Does not build the AST: it works on it.

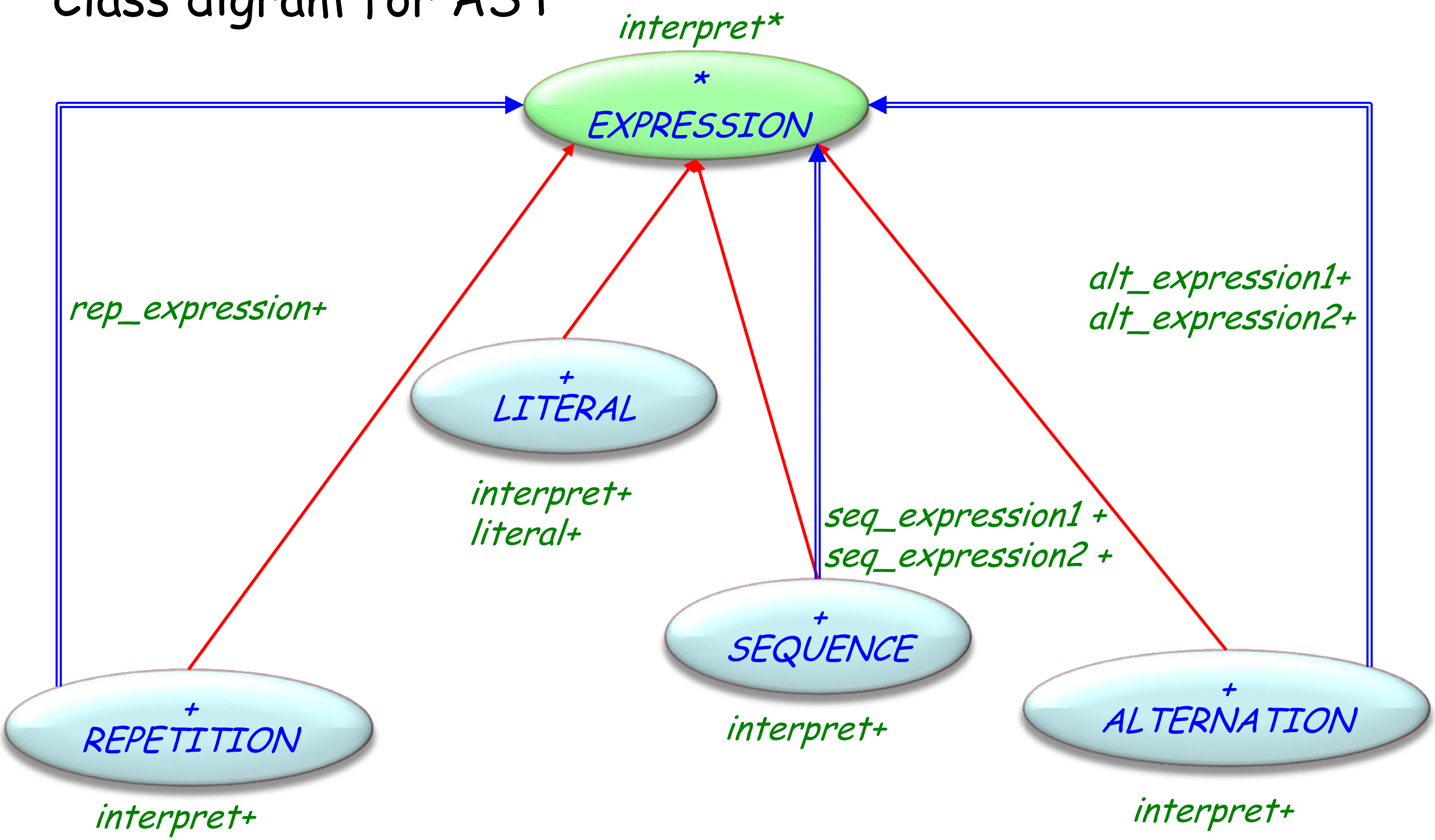
the *Interpreter* implements an interpretation/evaluation of the input (check if string matches reg. Expr)

Interpreter pattern

- Interpreter pattern uses a class to represent each grammar rule
- Each class has an "interpret" procedure
- Symbols on the right-hand side of the rule are attributes of the classes

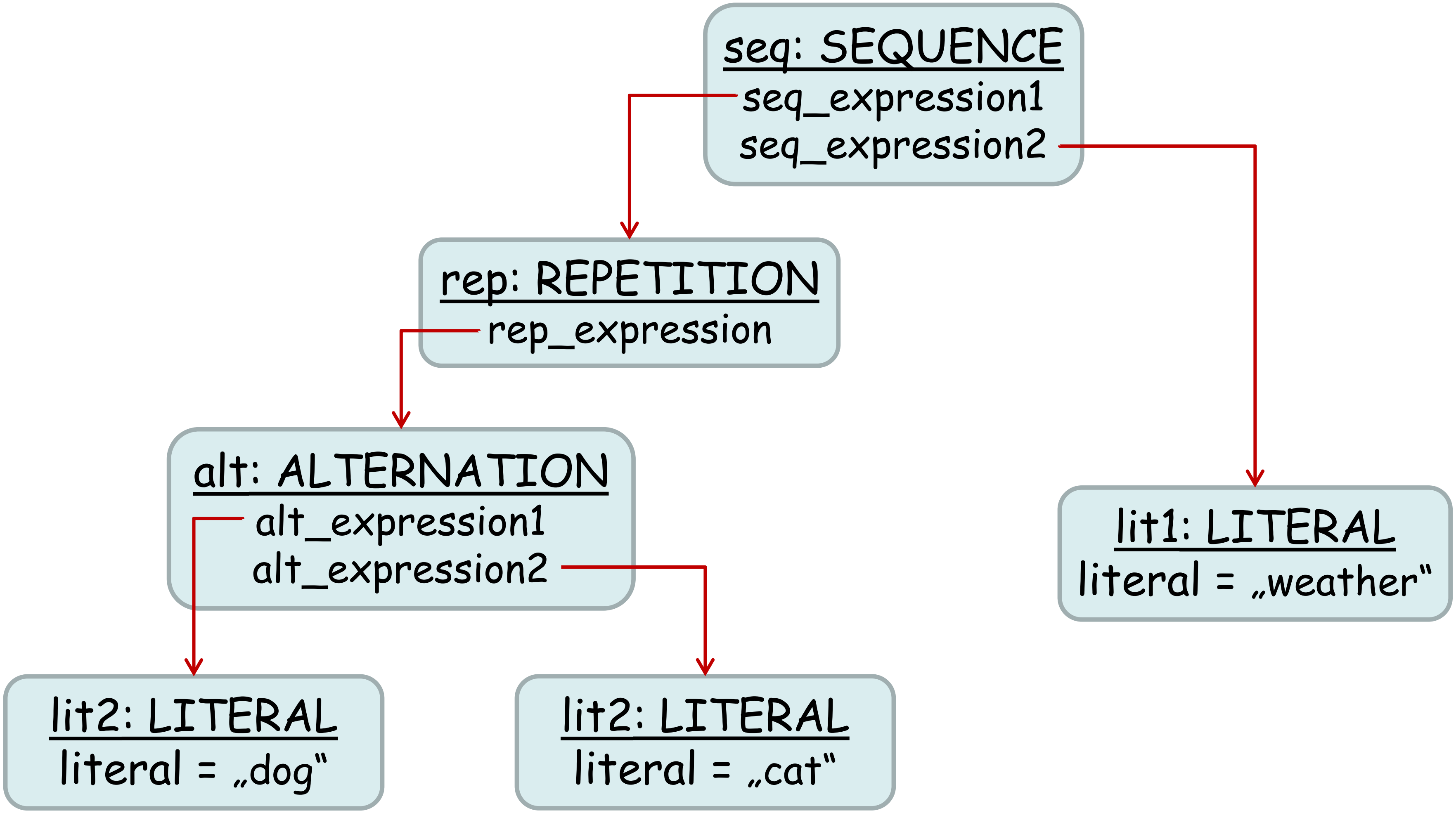
Interpreter pattern: Example

Class digram for AST



Interpreter pattern: Example

- Input AST : *((`dog `/`cat `)* & `weather`*



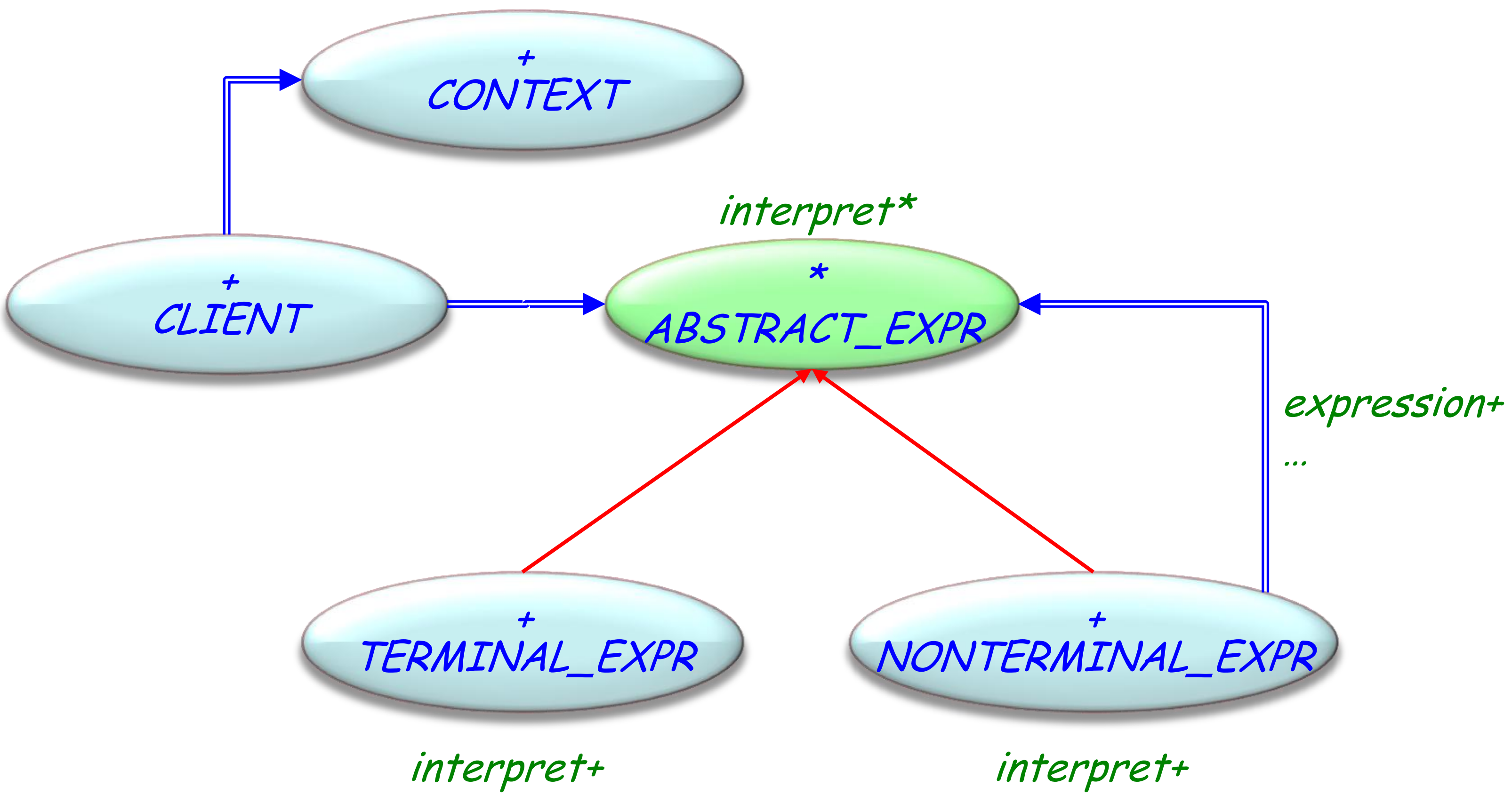
Interpreter pattern: Example

- Create interpreter for regular expression by defining the *interpret* procedure on each subclass of EXPRESSION

- *interpret* takes as argument a **context** in which to interpret the expression; context contains the input string and information on how much of it has been matched so far

- *interpret* for LITERAL: checks if input matches the literal it defines
- *interpret* for ALTERNATION: checks if input machtes any of its alternatvies
- *interpret* for REPETITION: checks if the input has multiple copies of expression it repeats

Interpreter pattern: Structure



Interpreter pattern: participants (1/2)

ABSTRACT_EXPR

- Declares an abstract interpret operation that is common to all nodes in the abstract syntax tree

TERMINAL_EXPR

- Implements and Interpret operation associated with terminal symbols in the grammar
- An instance is required for every terminal symbol in a sentence

NONTERMINAL_EXPR

- One such class is required for every rule in the grammar
- Maintains attributes of type ABSTRACT_EXPR for each rule's subexpressions
- Implements an Interpret procedure for nonterminal symbols in the grammar

Interpreter pattern: participants (2/2)

CONTEXT

- Contains information that is global to the interpreter

CLIENT

- Builds (or is given) an AST representing a particular sentence in the language the grammar defines (AST is assembled from instances of the NONTERMINAL_EXPR and TERMINAL_EXPR classes)

- Invokes the interpret operation

Interpreter pattern: when to use

Use the Interpreter pattern when

- The grammar is simple. For complex grammars, the class hierarchy becomes large and unmanageable. Parser generators are a better alternative then.

- Efficiency is not a critical concern. More efficient interpreters usually don't work on the AST but translate it first into another form (e.g. regular expression are translated into state machines)

Design patterns (GoF): that's all, folks

Creational

- ✓ Abstract Factory
- ✓ Singleton
- ✓ Factory Method
- ✓ Builder
- ✓ Prototype

Structural

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

Behavioral

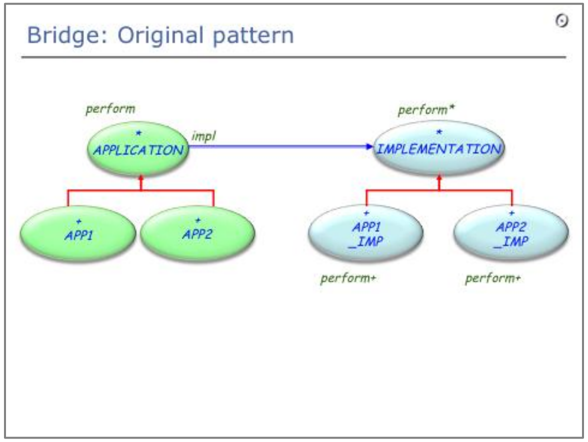
- ✓ Chain of Responsibility
- ✓ Command (undo/redo)
- ✓ Interpreter
- ✓ Iterator
- ✓ Mediator
- ✓ Memento
- ✓ Observer
- ✓ State
- ✓ Strategy
- ✓ Template Method
- ✓ Visitor

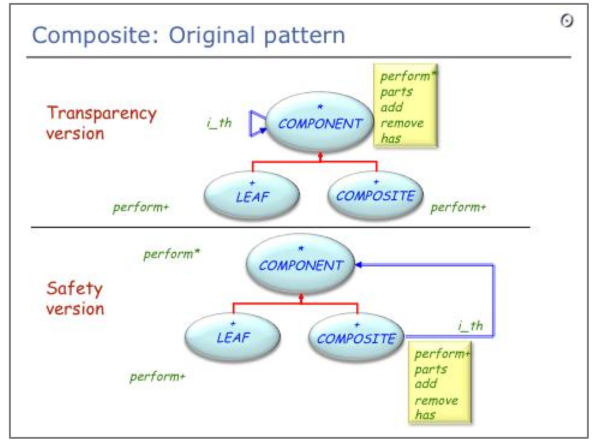
Non-GoF patterns

- ✓ Model-View-Controller

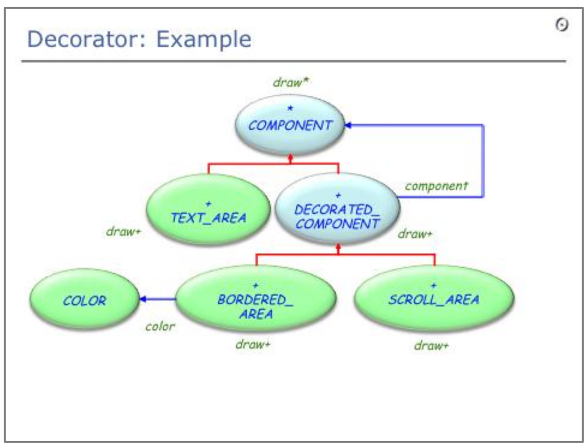
Summary of patterns – Structural patterns

Bridge: Separation of interface from implementation



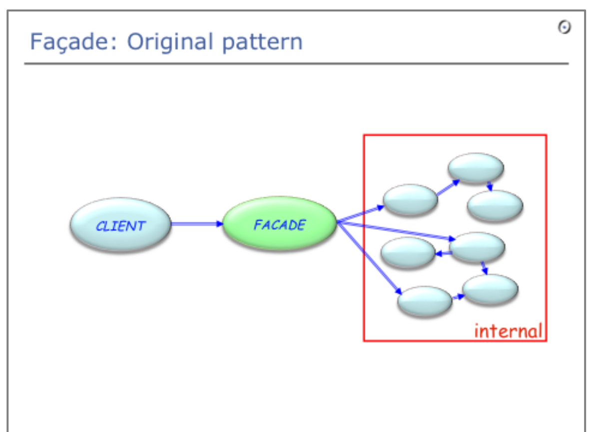


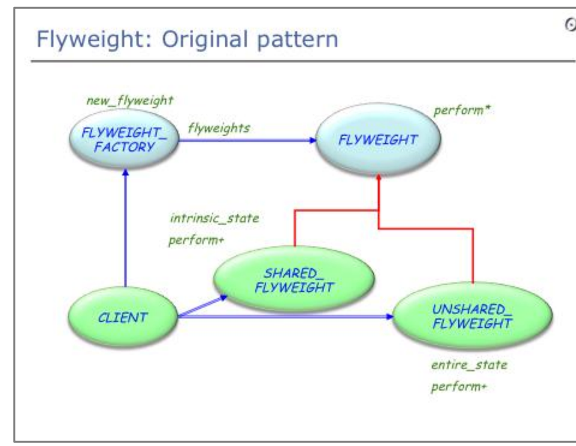
Composite: Uniform handling of compound and individual objects



Decorator: Attaching responsibilities to objects without subclassing

Façade: A unified interface to a subsystem

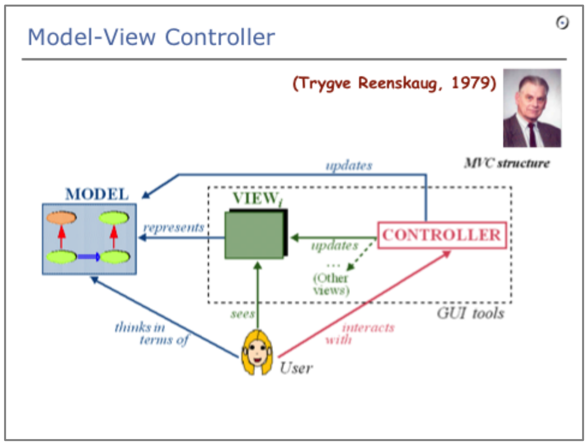




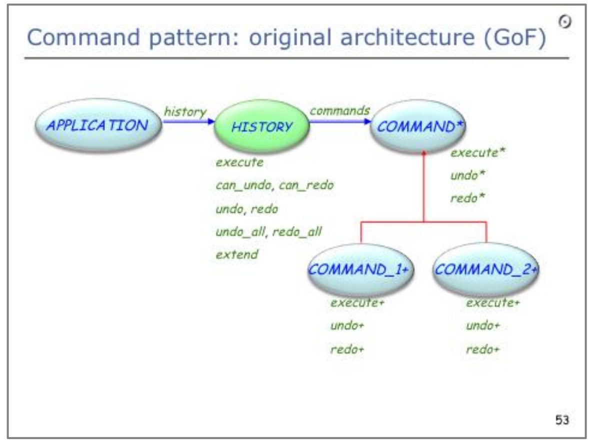
Flyweight: Share objects and externalize state

Summary of patterns – Behavioral patterns

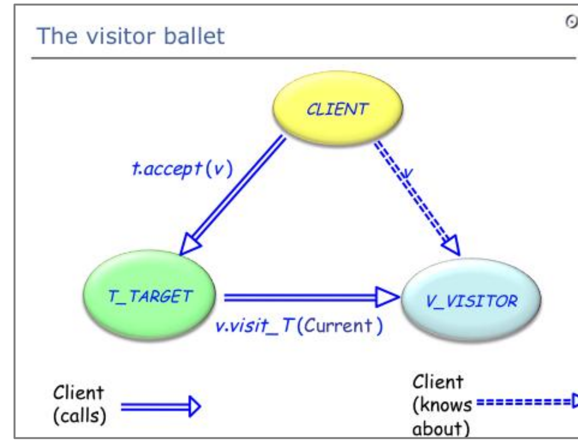
Observer; MVC: Publish-subscribe mechanism (use *EVENT_TYPE* with agents!); Separation of model and view



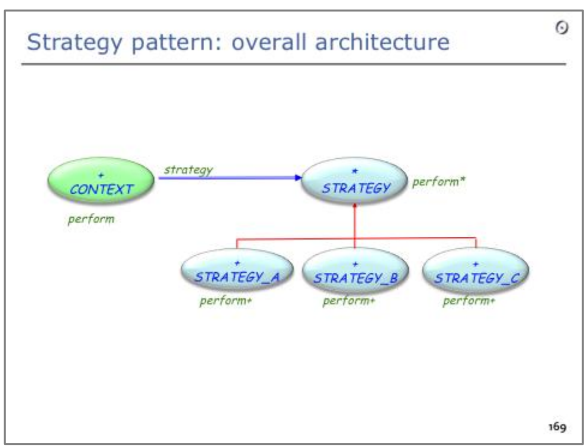
Command: History with undo/redo (use version with agents!)



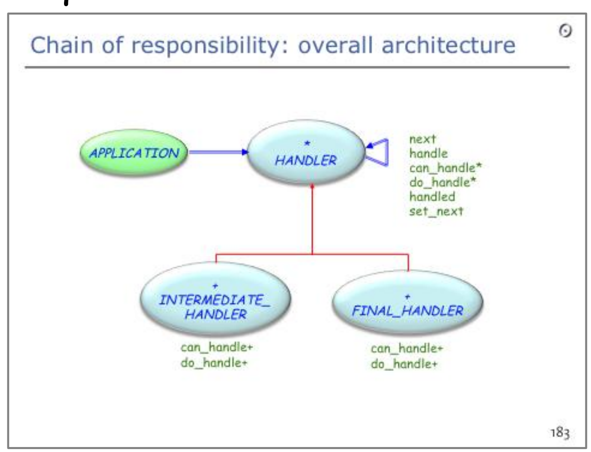
Visitor: Add operations to object hierarchies without changing classes



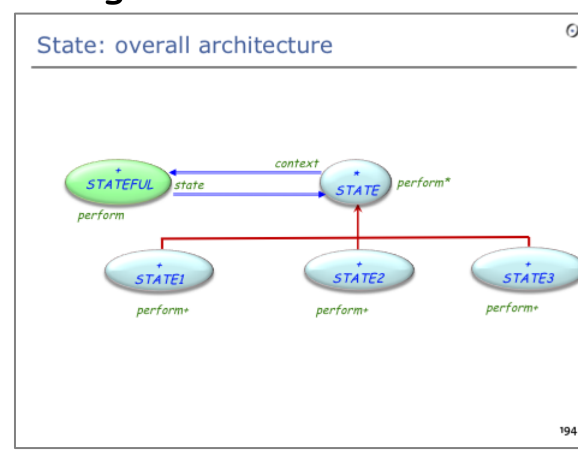
Strategy: Make algorithms interchangeable



Chain of responsibility: Allow multiple objects to handle request



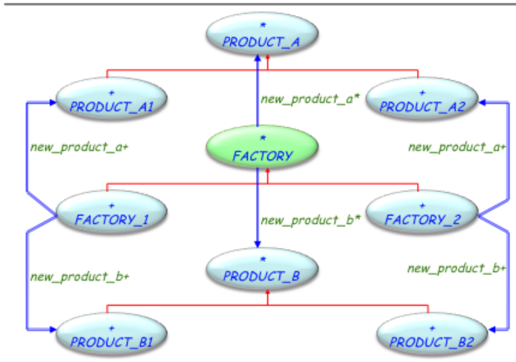
State: Object appears to change behavior if state changes



Summary of patterns – Creational patterns

Abstract factory: Hiding the creation of product families

Abstract factory architecture



Factory Method pattern

Intent:

"*Define[s] an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*" [Gamma et al.]

C++, Java, C#: emulates constructors with names

Factory Method vs. Abstract Factory:

- Creates one object, not families of object.
- Works at the routine level, not class level.
- Helps a class perform an operation, which requires creating an object.
- Features *new* and *new_with_args* of the Factory Library are factory methods

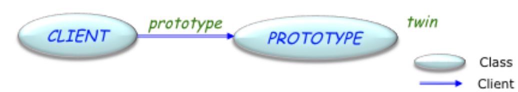
Factory method: Interface for creating an object, but hiding its concrete type (used in abstract factory)

Prototype: Use *twin* or *clone* to duplicate an object

Prototype pattern

Intent:

"*Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.*" [Gamma 1995]



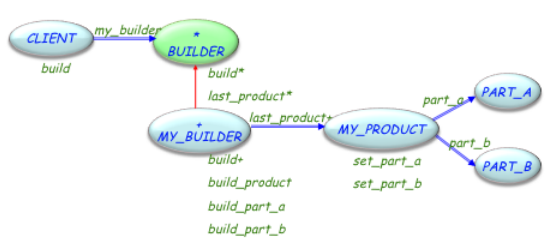
No need for this in Eiffel: just use function *twin* from class *ANY*.

y := x.twin

In Eiffel, every object is a prototype

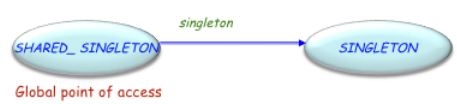
Builder: Encapsulate construction process of a complex object

Builder pattern



Singleton pattern

Way to "ensure a class only has one instance, and to provide a global point of access to it." [GoF, p 127]



Singleton: Restrict a class to globally have only one instance and provide a global access point to it

Complementary material Singleton (1/3)

From Patterns to Components:

Chapter 18: Singleton

Further reading:

- Erich Gamma: *Design Patterns*, 1995.
  (Singleton, p 127-134)

- Karine Arnout and Éric Bezault. "How to get a Singleton in Eiffel", *JOT*, 2004.
  http://www.jot.fm/issues/issue_2004_04/article5.pdf.

Complementary material Singleton (2/3)

Further reading:

- Joshua Fox. "When is a singleton not a singleton?", *JavaWorld*, 2001. http://www.javaworld.com/javaworld/jw-01-2001/jw-0112-singleton.html.
- David Geary. "Simply Singleton", JavaWorld, 2003. http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html.
- Robert C. Martin. "Singleton and Monostate", 2002. http://www.objectmentor.com/resources/articles/SingletonAndMonostate.pdf.

Complementary material Singleton (3/3)

Further reading:

- Miguel Oliveira e Silva. "Once creation procedures". comp.lang.eiffel. http://groups.google.com/groups?dq=&hl=en&lr=&ie=UTF-8&threadm=GJnJzK.9v6%40ecf.utoronto.ca&prev=/groups%3Fdq%3D%26hl%3Den%26lr%3D%26ie%3DUTF-8%26group%3Dcomp.lang.eiffel%26start%3D525.

Design patterns: References

- Erich Gamma, Ralph Johnson, Richard Helms, John Vlissides: *Design Patterns*, Addison-Wesley, 1994

- Jean-Marc Jezequel, Michel Train, Christine Mingins: *Design Patterns and Contracts*, Addison-Wesley, 1999

- Karine Arnout: *From Patterns to Components*, 2004 ETH thesis, http://e-collection.ethbib.ethz.ch/eserv/eth:27168/eth-27168-02.pdf

Pattern componentization: references

- Bertrand Meyer: *The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design*, in *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, Lecture Notes in Computer Science 2635, Springer-Verlag, 2004, pages 236-271
  se.ethz.ch/~meyer/ongoing/events.pdf

- Karine Arnout and Bertrand Meyer: *Pattern Componentization: the Factory Example*, in *Innovations in Systems and Software Technology (a NASA Journal)* (Springer-Verlag), 2006
  se.ethz.ch/~meyer/publications/nasa/factory.pdf

- Bertrand Meyer and Karine Arnout: *Componentization: the Visitor Example*, in *Computer* (IEEE), vol. 39, no. 7, July 2006, pages 23-30
  se.ethz.ch/~meyer/publications/computer/visitor.pdf

- Bertrand Meyer, Touch of Class, *16.14 Reversing the structure: Visitor and agents*, page 606 – 613, 2009
  http://www.springerlink.com/content/n6ww275n43114383/fulltext.pdf