Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

Lecture 6:

Patterns
(with material by other members of the team)

Note about these slides

For a more extensive version (from the “Software Architecture” course), see

http://se.inf.ethz.ch/courses/2011a_spring/soft_arch/lectures/04_softarch_patterns.pdf

The present material is a subset covering the patterns of direct relevance to the Robotics Programming Laboratory

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.

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.

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

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.

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.

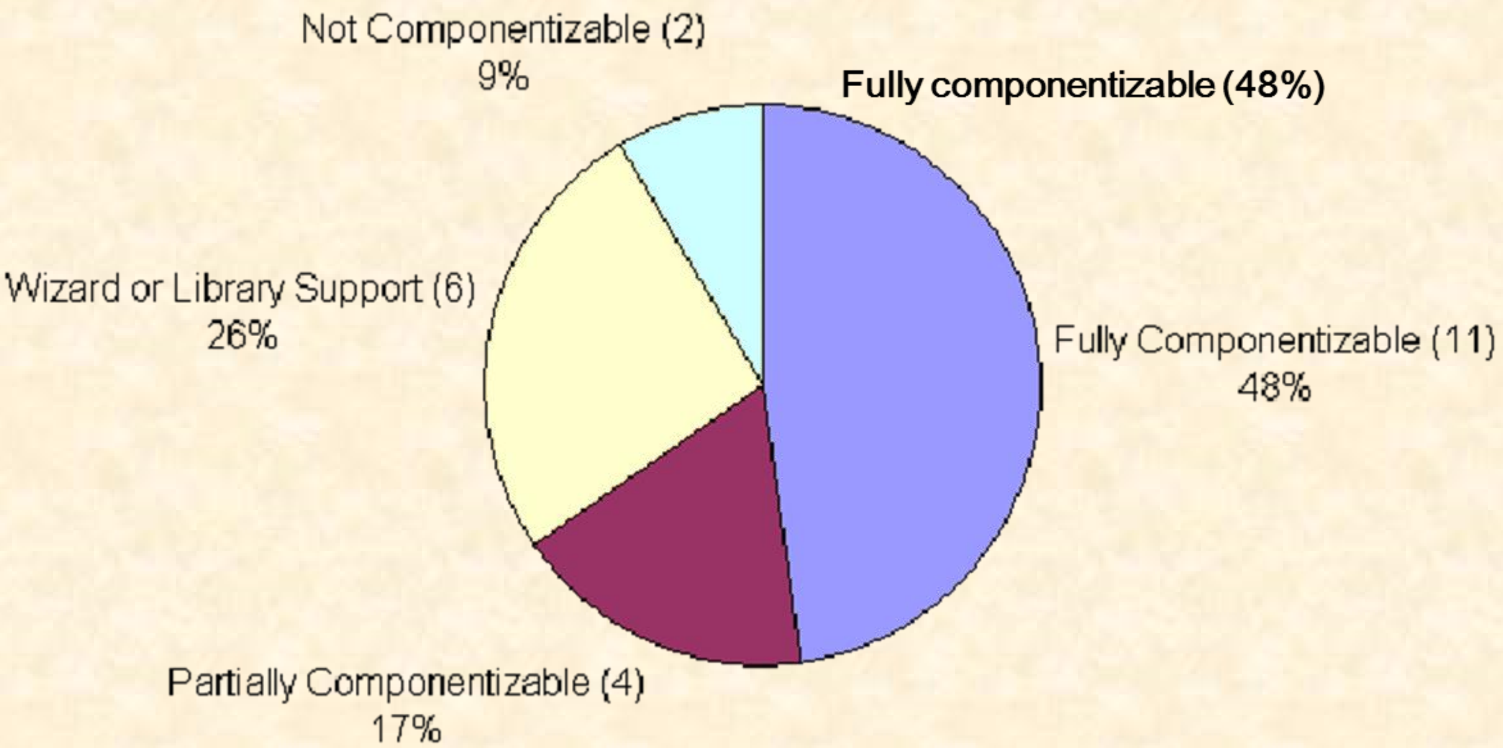
NOT REUSABLE

Pattern componentization

Karine Arnout
ETH PhD, 2004

Classification of design patterns:

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



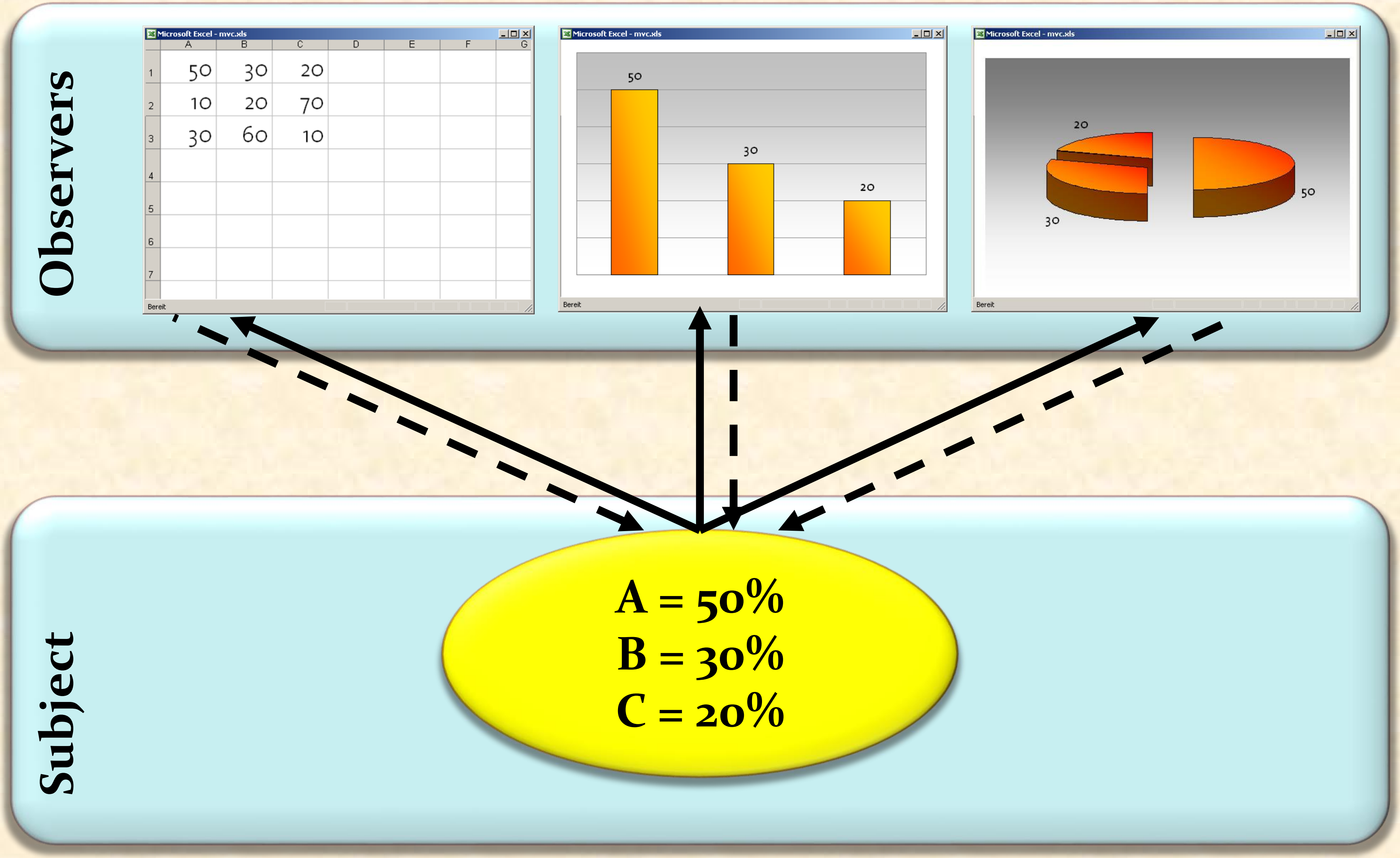
Observer pattern and event-driven progr.

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

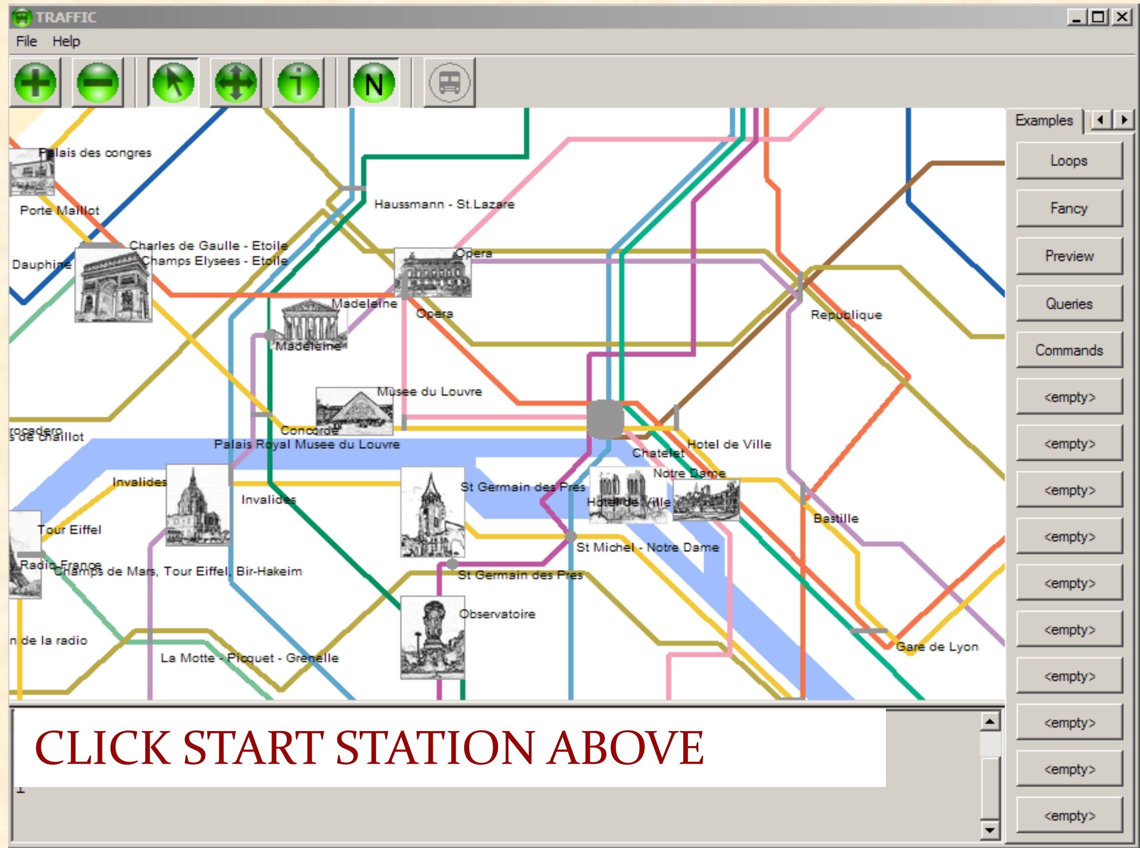
Observer and event-driven design



Handling input with modern GUIs

User drives program:

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

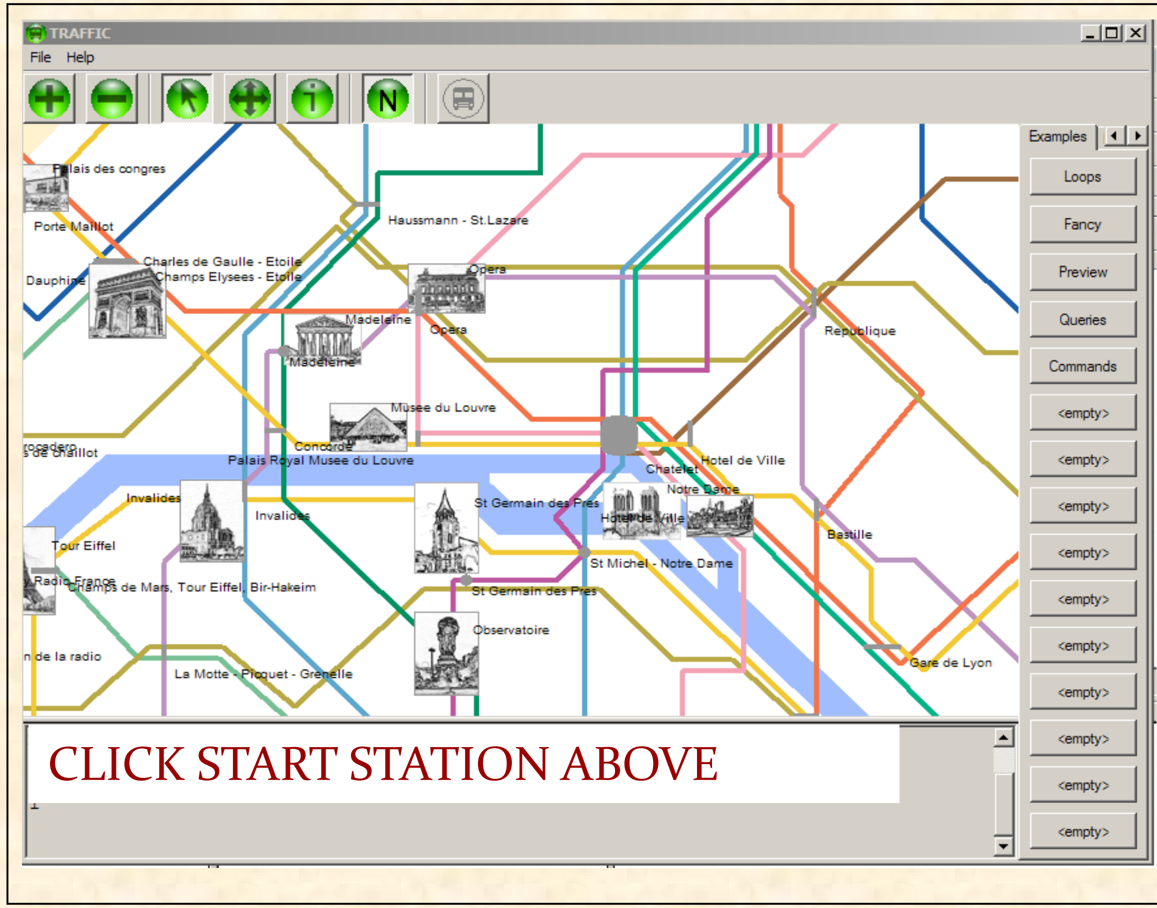


Event-driven programming: an example

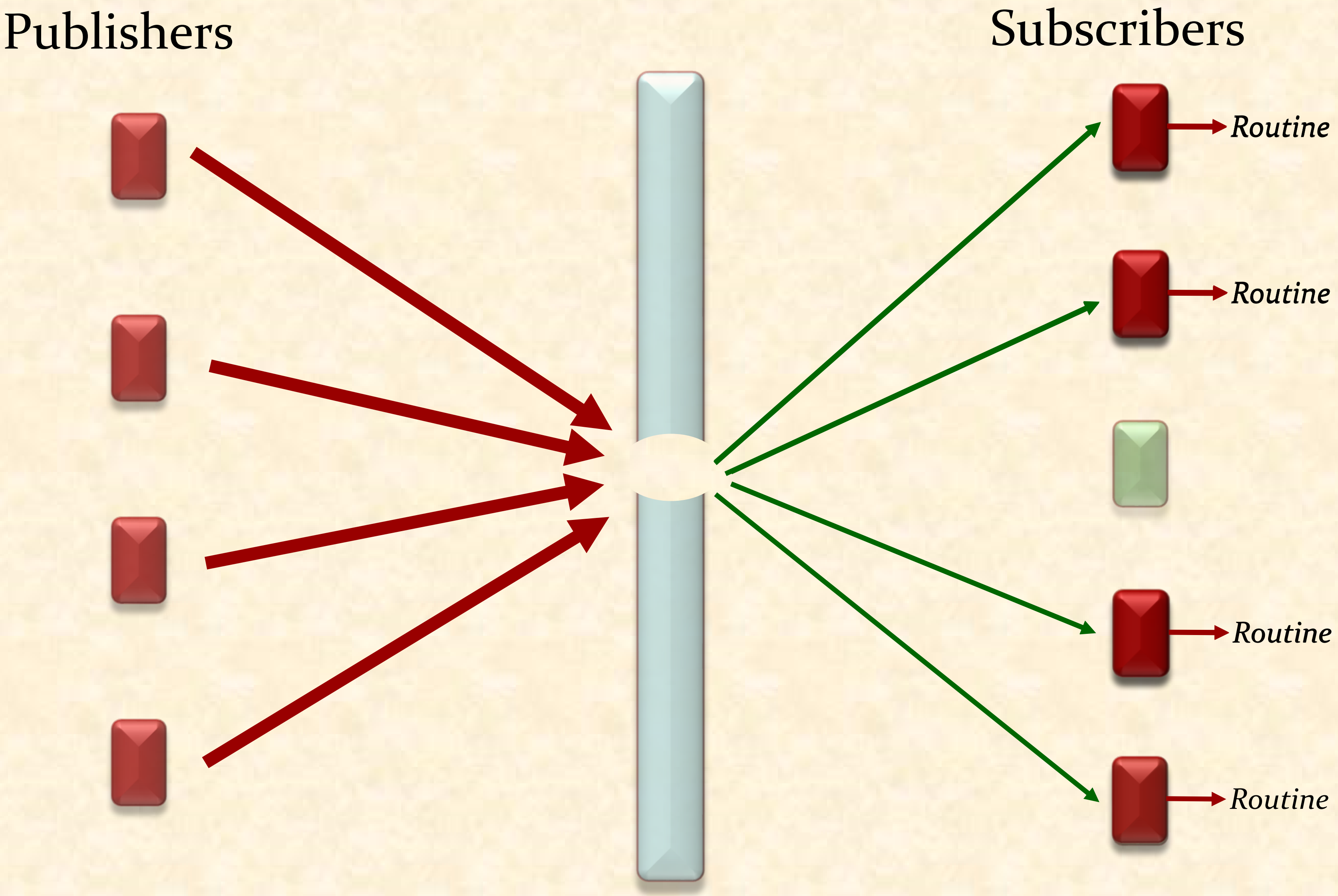
Specify that when a user clicks this button the system must execute

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

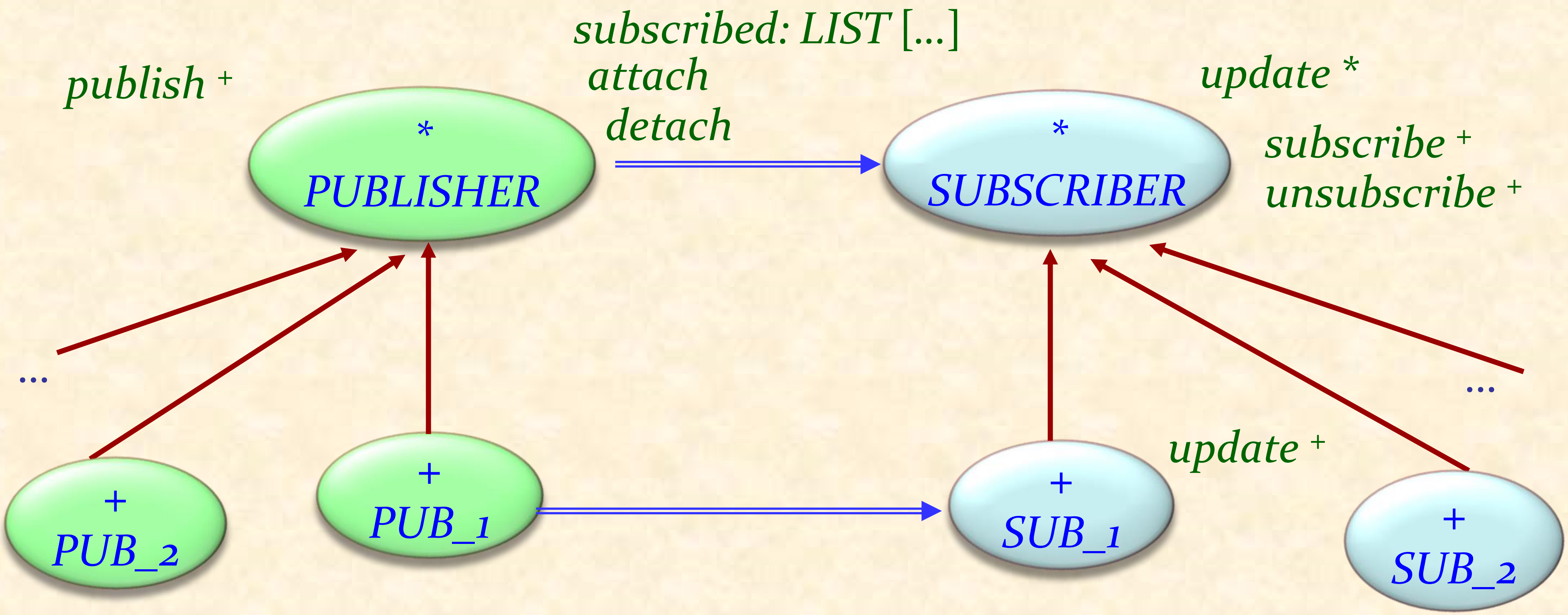
Subject / Observer

Publish / Subscribe

Event-driven design/programming

In this presentation:
Publisher and Subscriber

A solution: the Observer Pattern (GoF)



* Deferred (abstract)

\+ Effective (implemented)

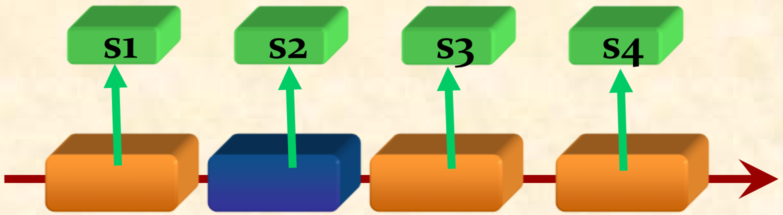
Inherits from

Client (uses)

Observer pattern

Publisher keeps a (secret) list of observers:

subscribed : *LINKED_LIST* [*SUBSCRIBER*]



To register itself, an observer executes

subscribe (*some_publisher*)

where *subscribe* is defined in *SUBSCRIBER* :

```
subscribe (p: PUBLISHER)
        -- Make current object observe p.
    require
        publisher_exists: p /= Void
    do
        p.attach (Current)
    end
```

Attaching an observer

In class *PUBLISHER* :

Why?

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

            subscribed.extend (s )
        end
```

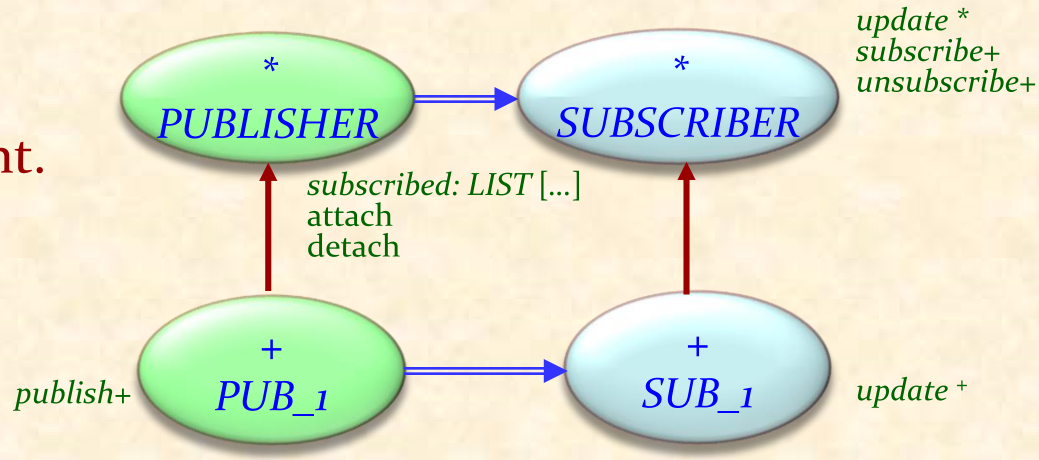
Note that the invariant of *PUBLISHER* includes the clause

subscribed /= Void

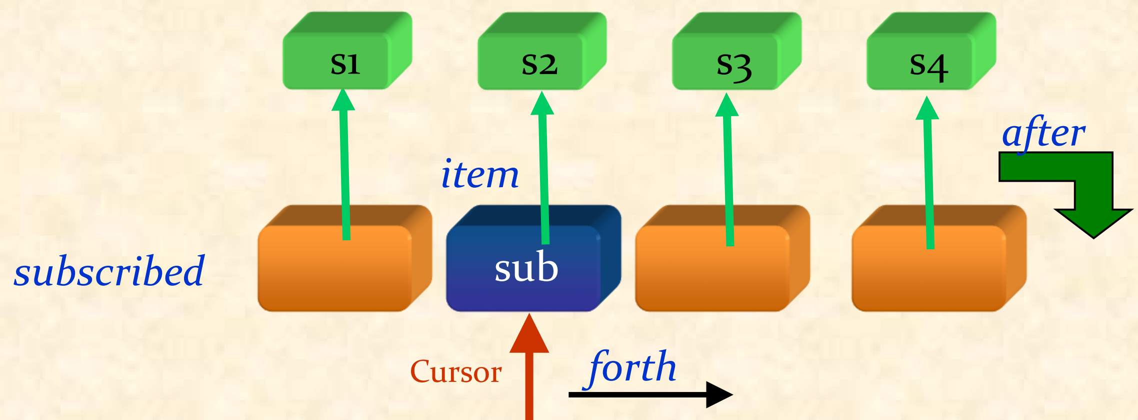
(List *subscribed* is created by creation procedures of *PUBLISHER*)

Triggering an event

```
publish
        -- Ask all observers to
        -- react to current event.
    do
        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

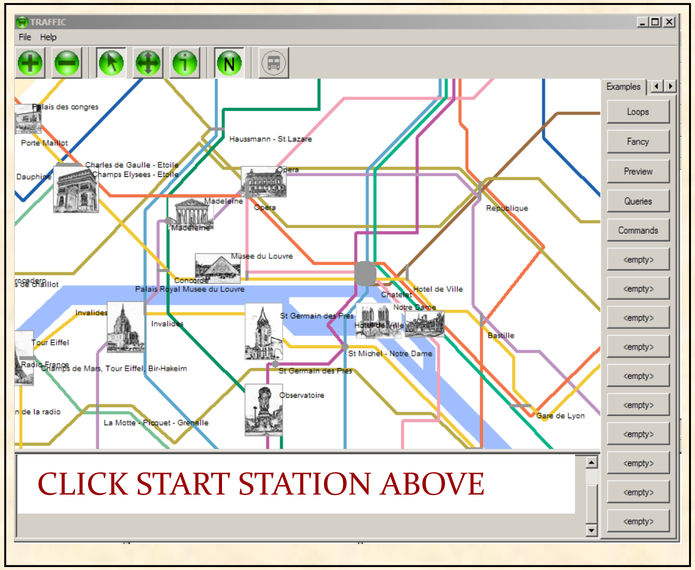
Observer - Consequences

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.

Using agents in EiffelVision



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

Mechanisms in other languages

- C and C++: “function pointers”

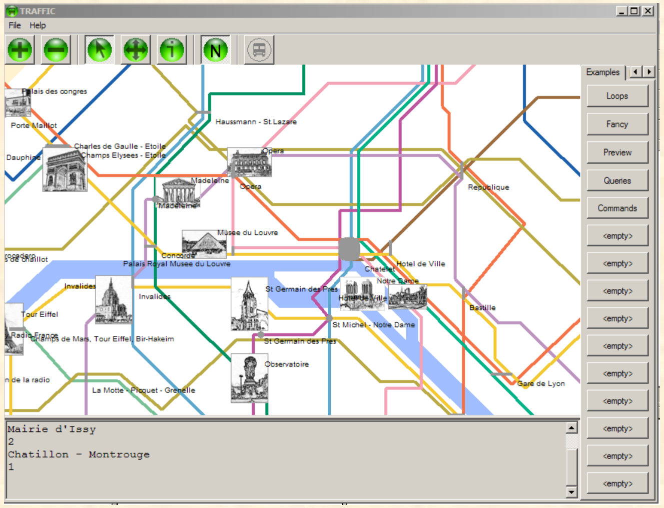
- C#: delegates (more limited form of agents)

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

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 ...)

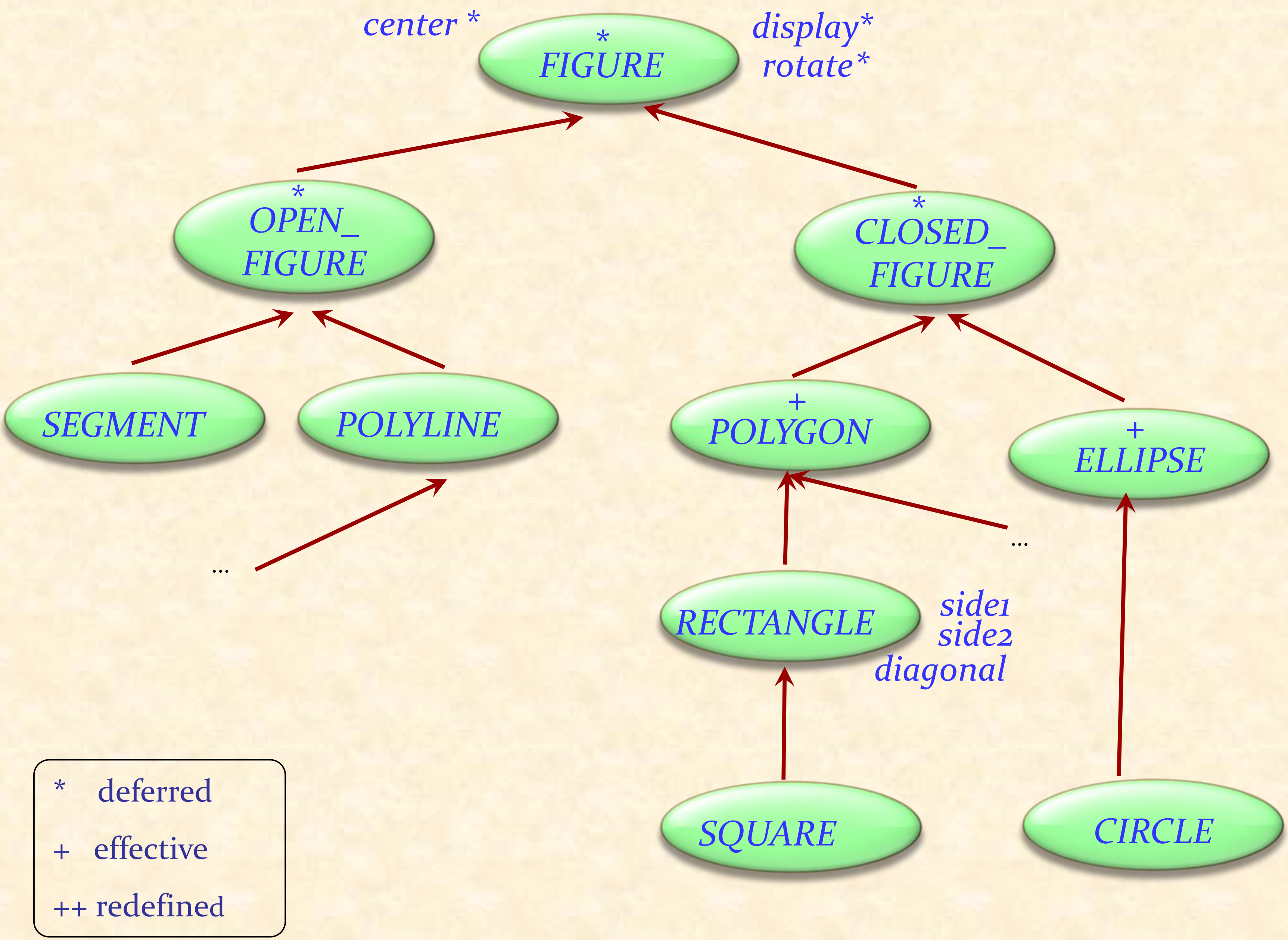
Or: Set of classes to deal with XML documents (*XML_NODE, XML_DOCUMENT, XML_ELEMENT, XML_ATTRIBUTE, XML_CONTENT…)*

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

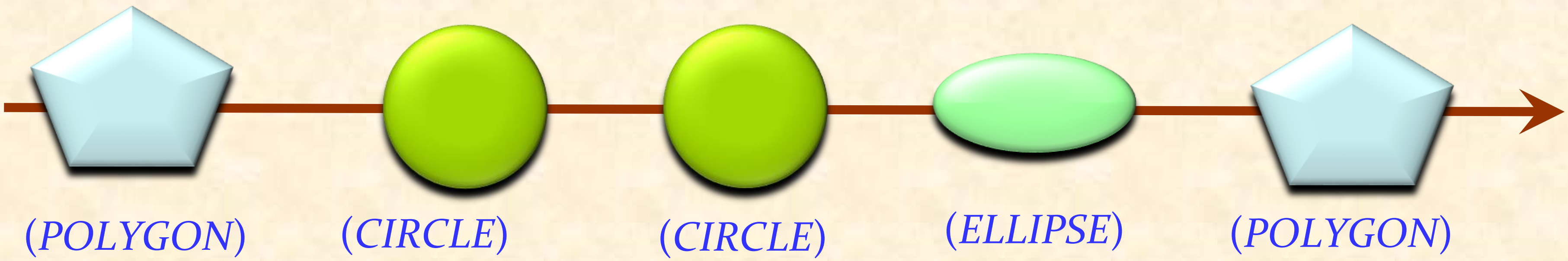
Many formatters:

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

Inheritance hierarchy



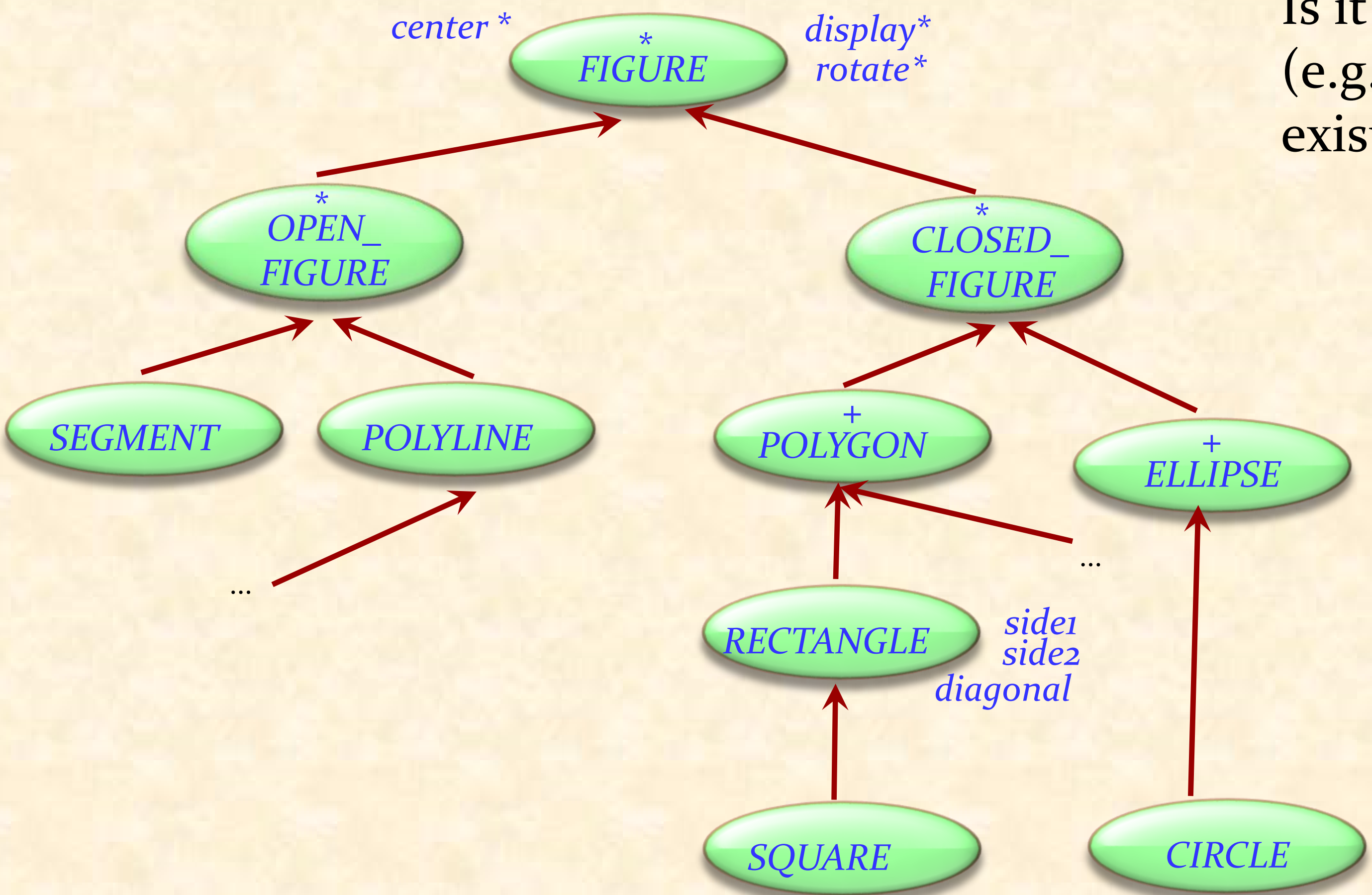
Polymorphic data structures



figs : LIST [FIGURE]

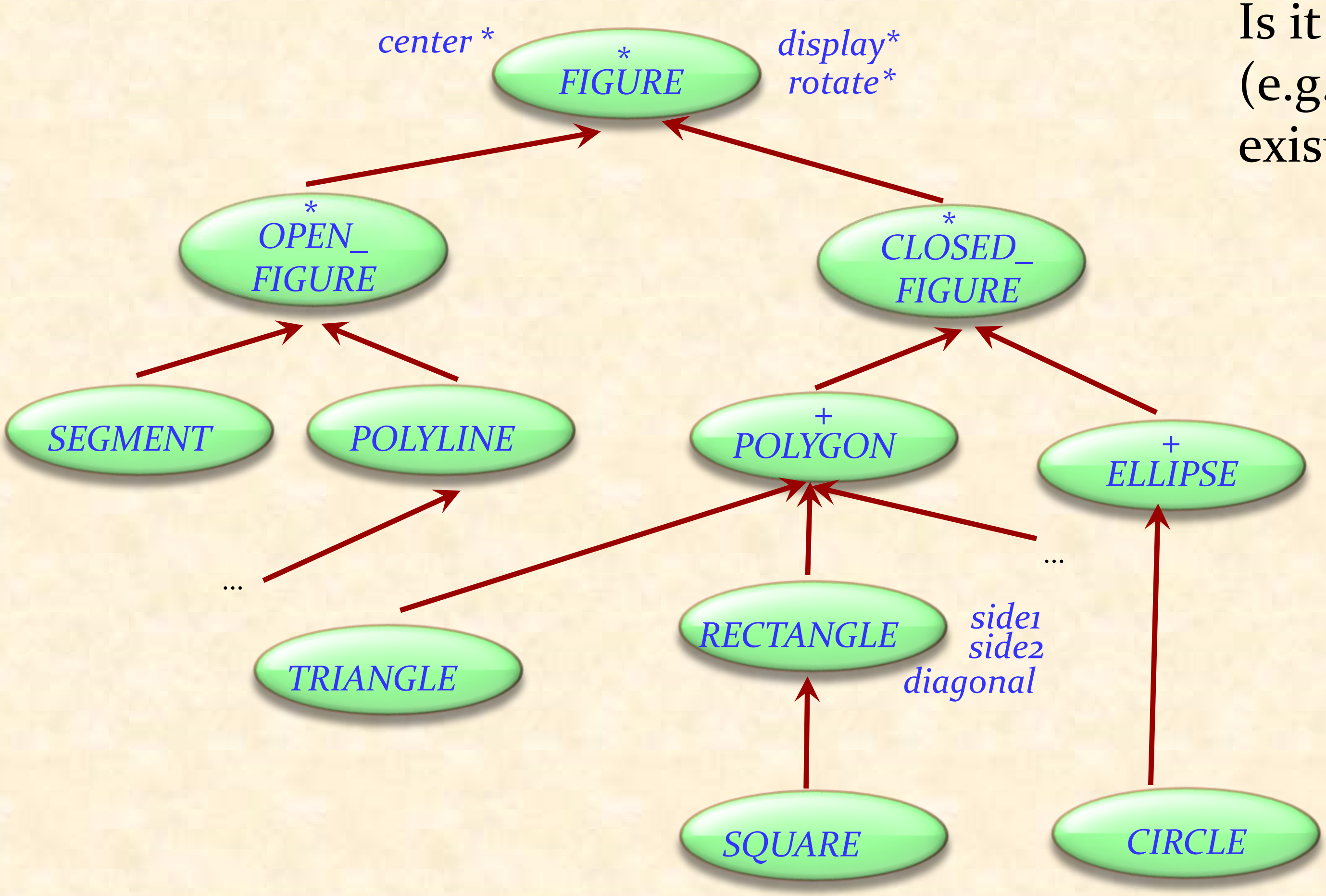
```
from
    figs.start
until
    figs.after
loop
    figs.item .display
    figs.forth
end
```

The dirty secret of O-O architecture



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

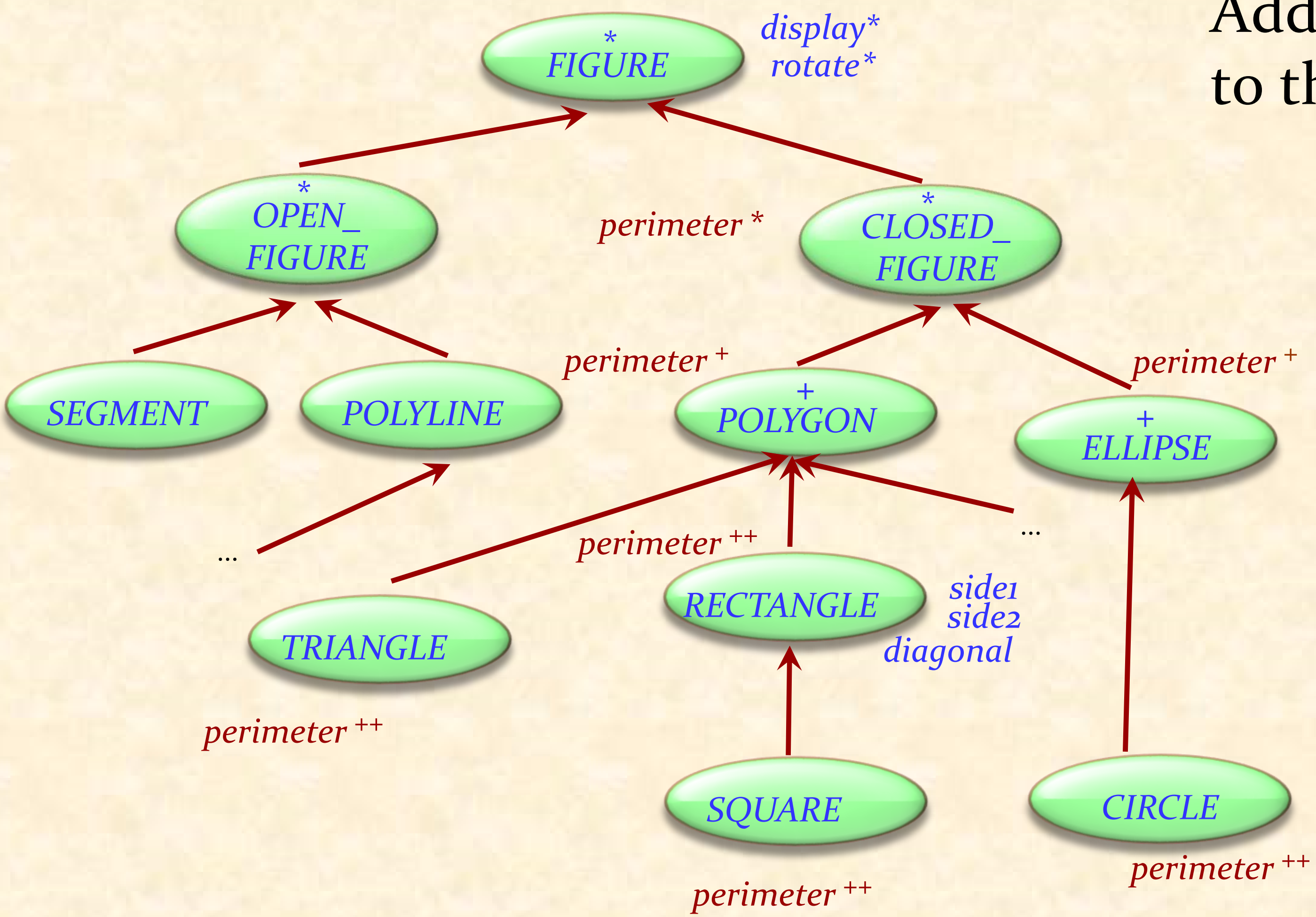
The dirty secret of O-O architecture



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

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

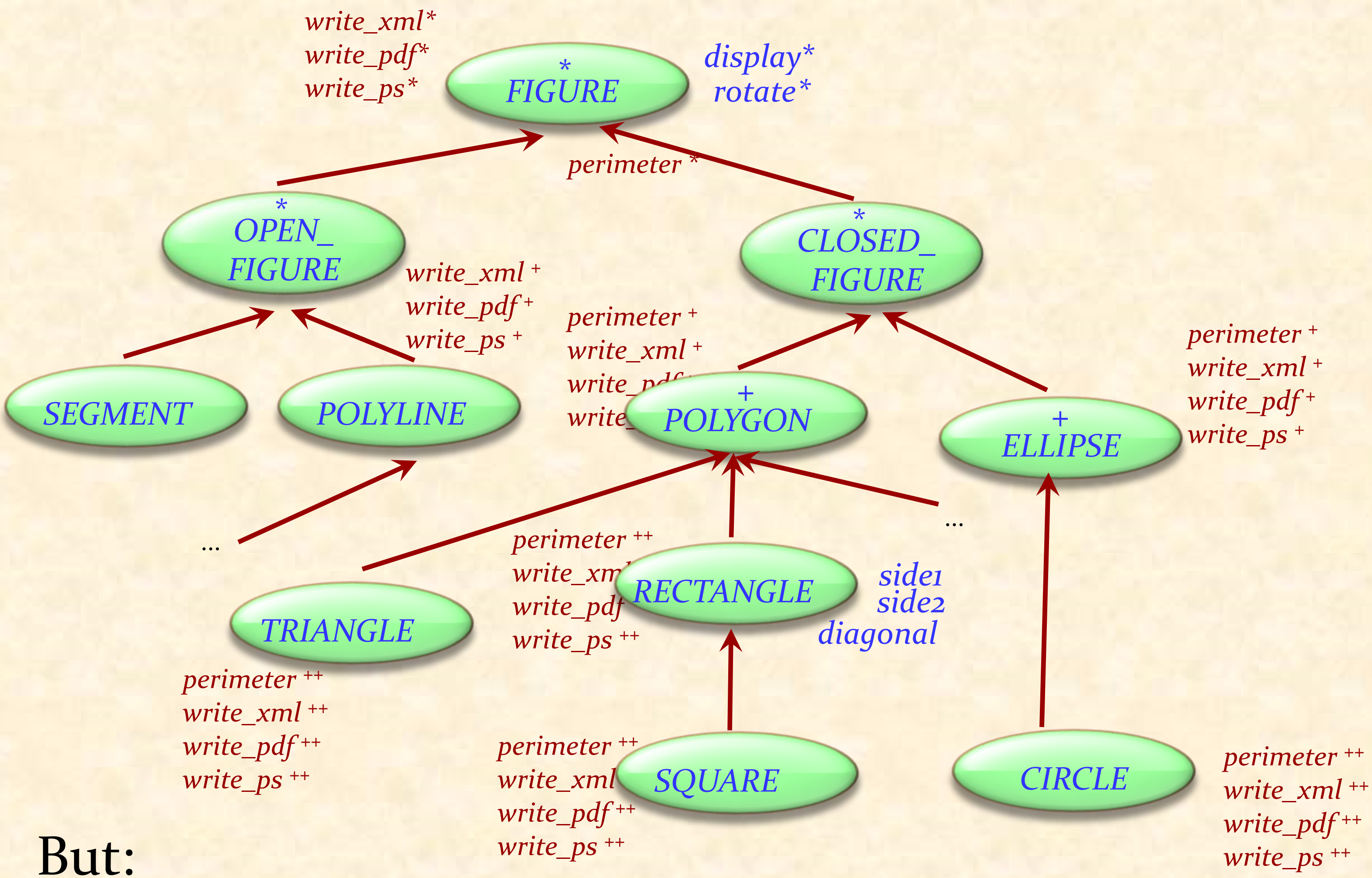
Adding operations – solution 1



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

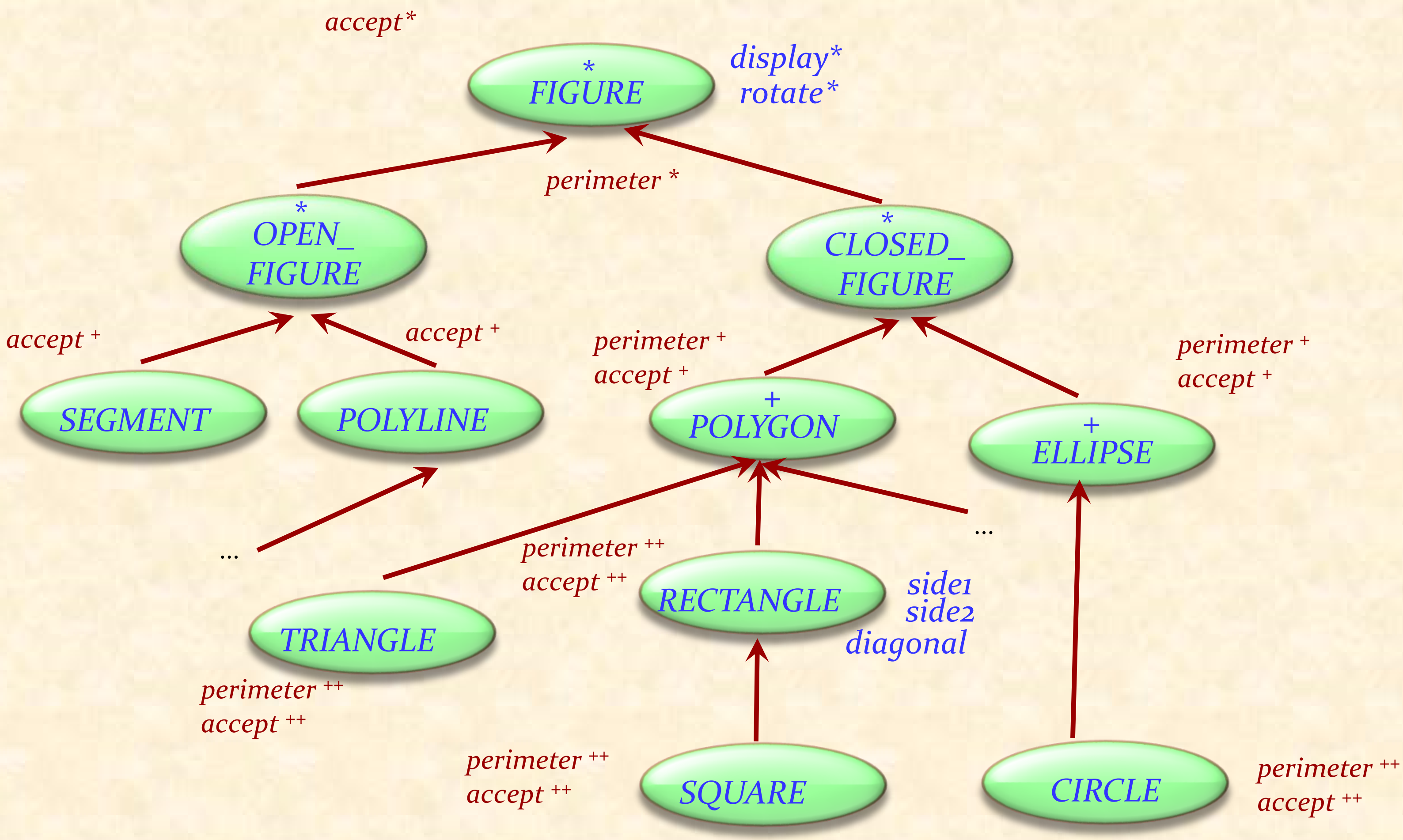
```
write_xml (f: FIGURE)
    -- Write figure to xml.
  require exists: f/= Void
  do
    ...
    if attached {RECT} f as r then
          doc.put_string ("<rect/>")
    end
    if attached {CIRCLE} f as c then
          doc.put_string ("<circle/>")
    end
    ... Other cases ...
  end
end
```

```
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:

- Lose benefits of dynamic binding
- Many large conditionals

Adding operations – solution 3



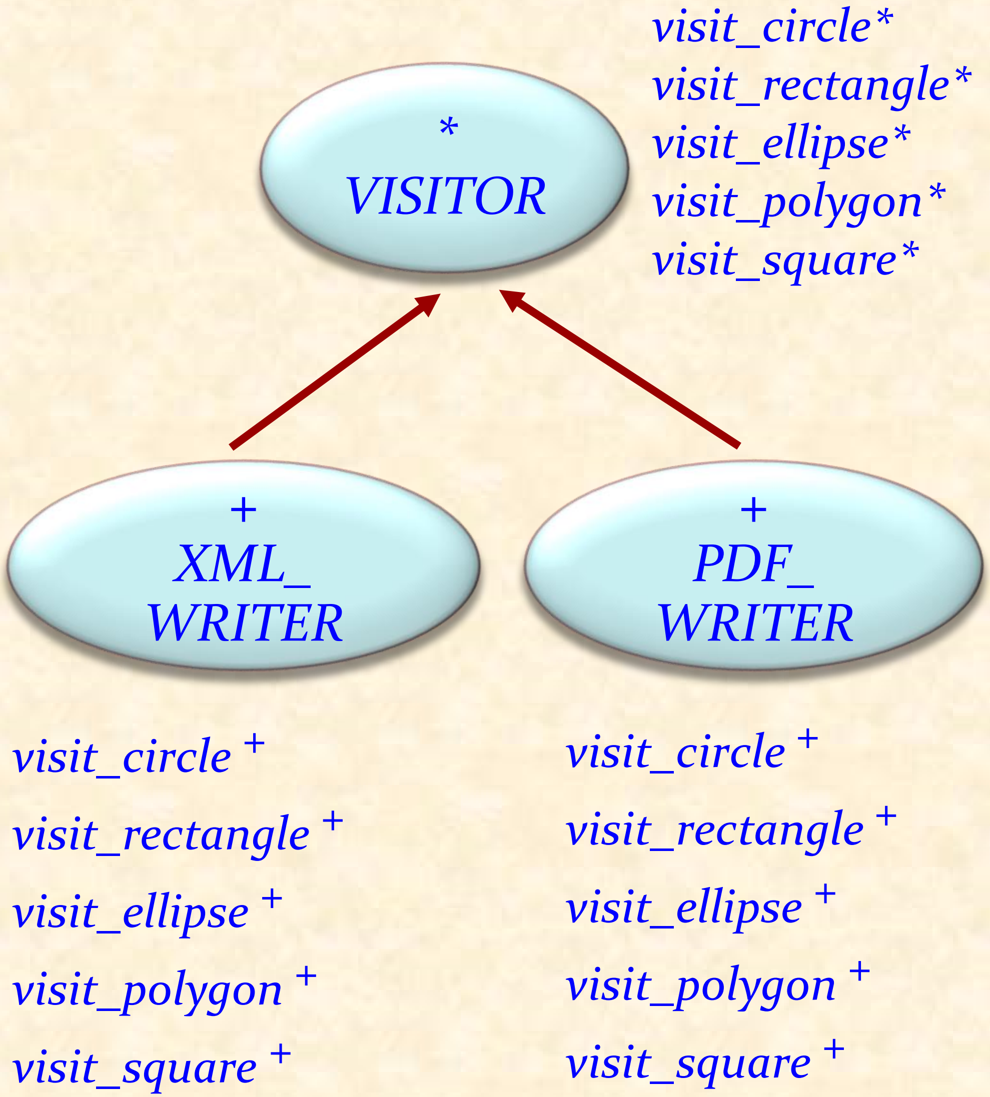
Combine solution 1 & 2:

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

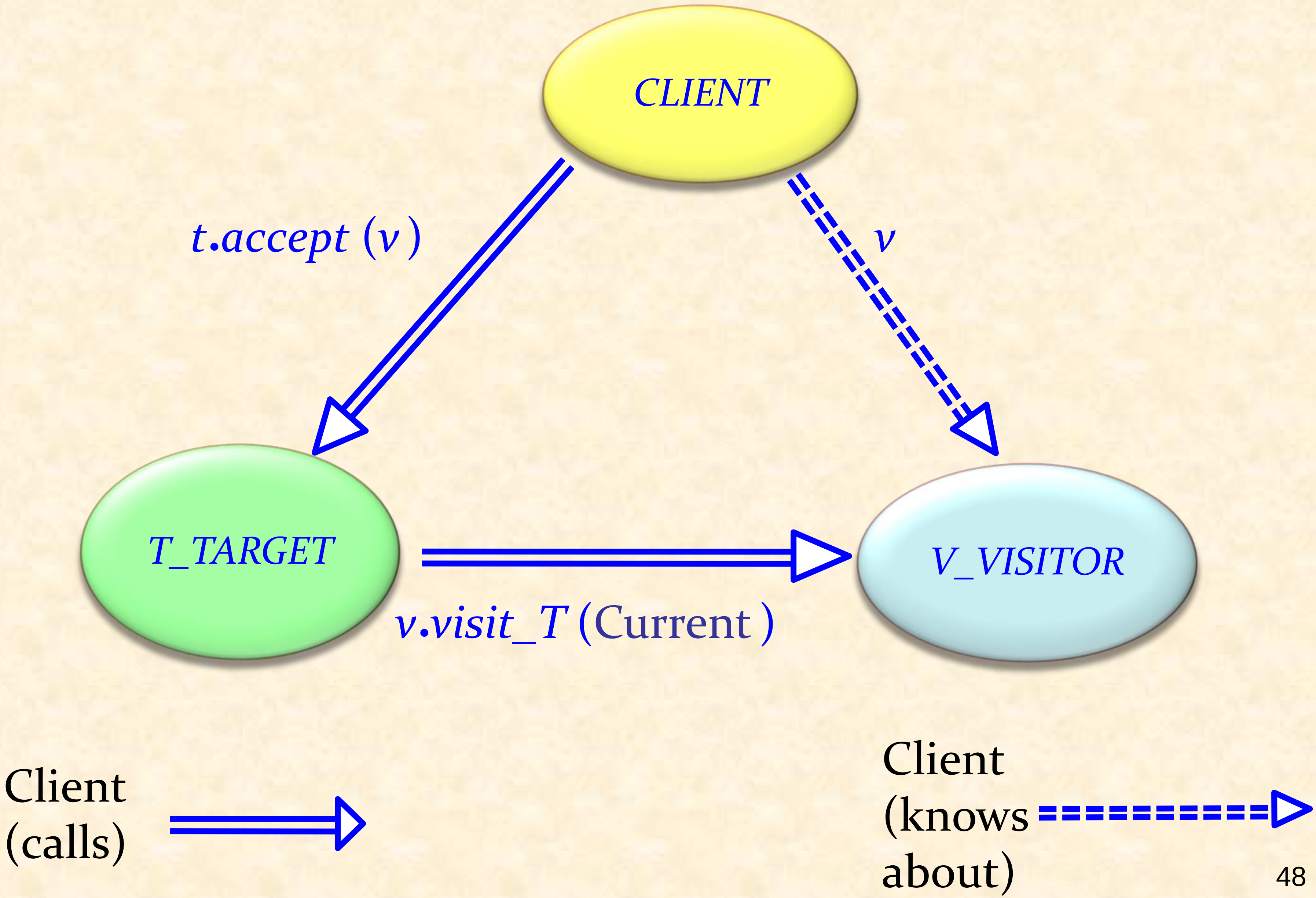
Adding operations – solution 3

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

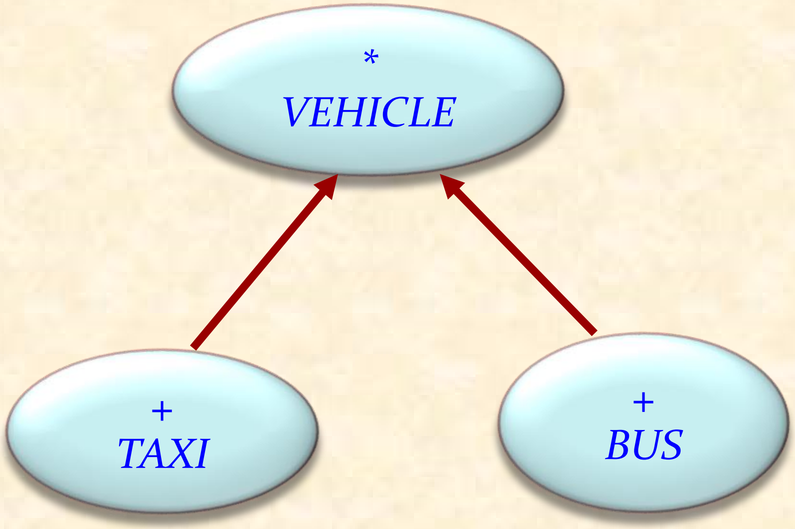
```
class CIRCLE
feature
accept (v : VISITOR)
      --Call procedure of visitor.
   do
      v.visit_circle (Current)
   end
      ... Other features ...
end
```



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

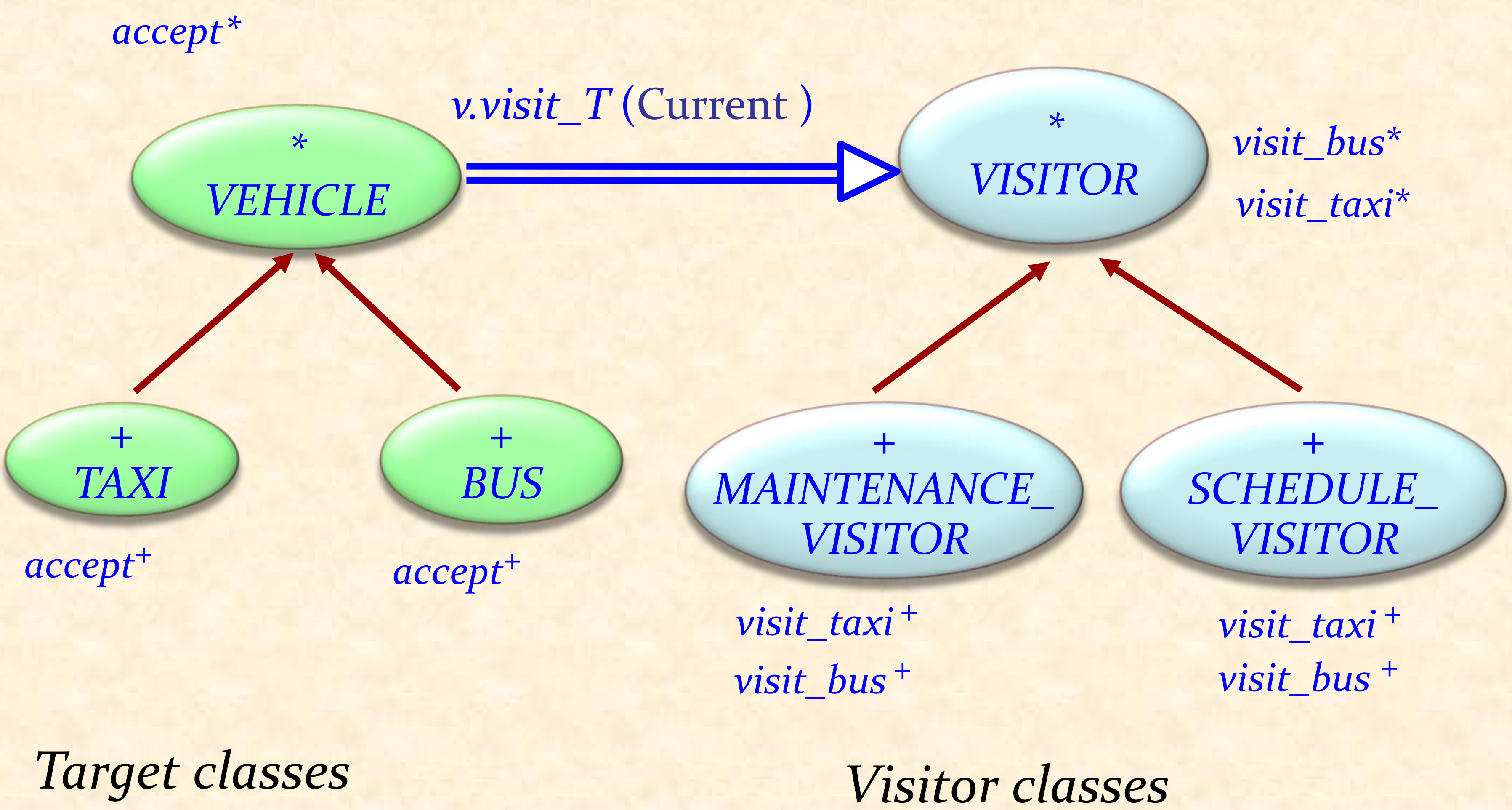
Target

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
```

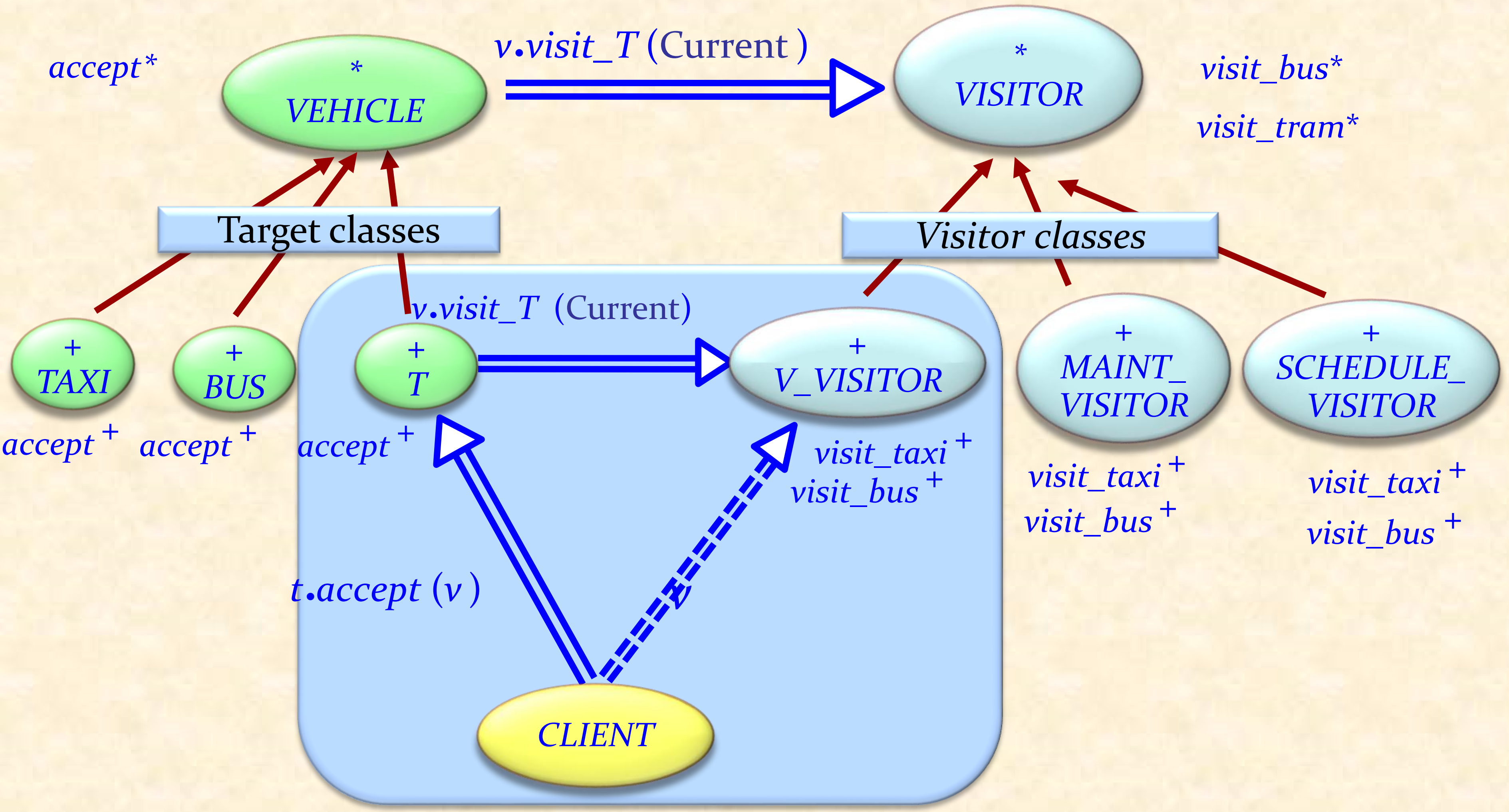
Changes to the target classes

```
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*)

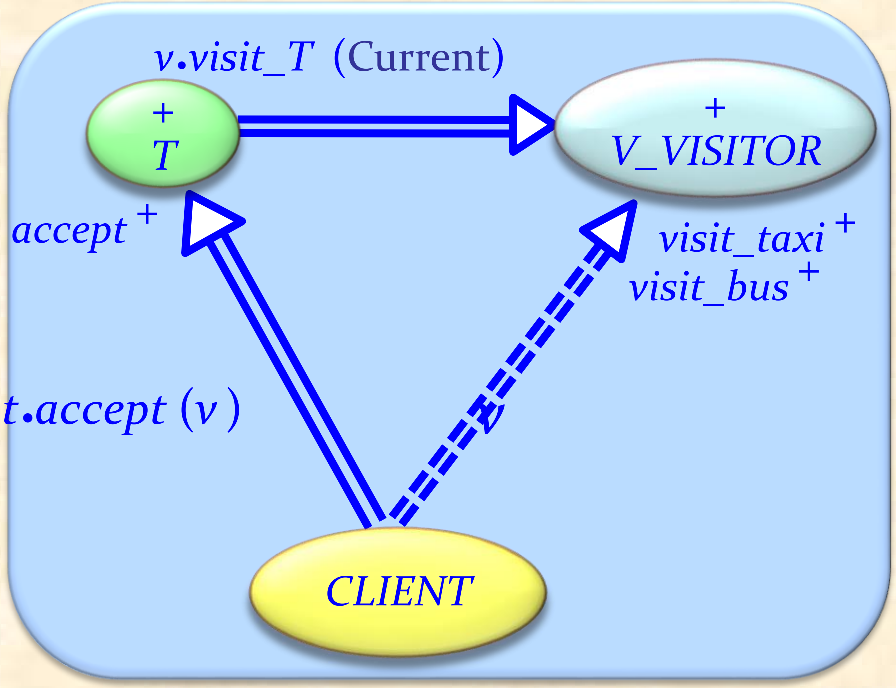
Target class (in *accept*):

v.*visit_T* (*t*)

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

v.*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
```

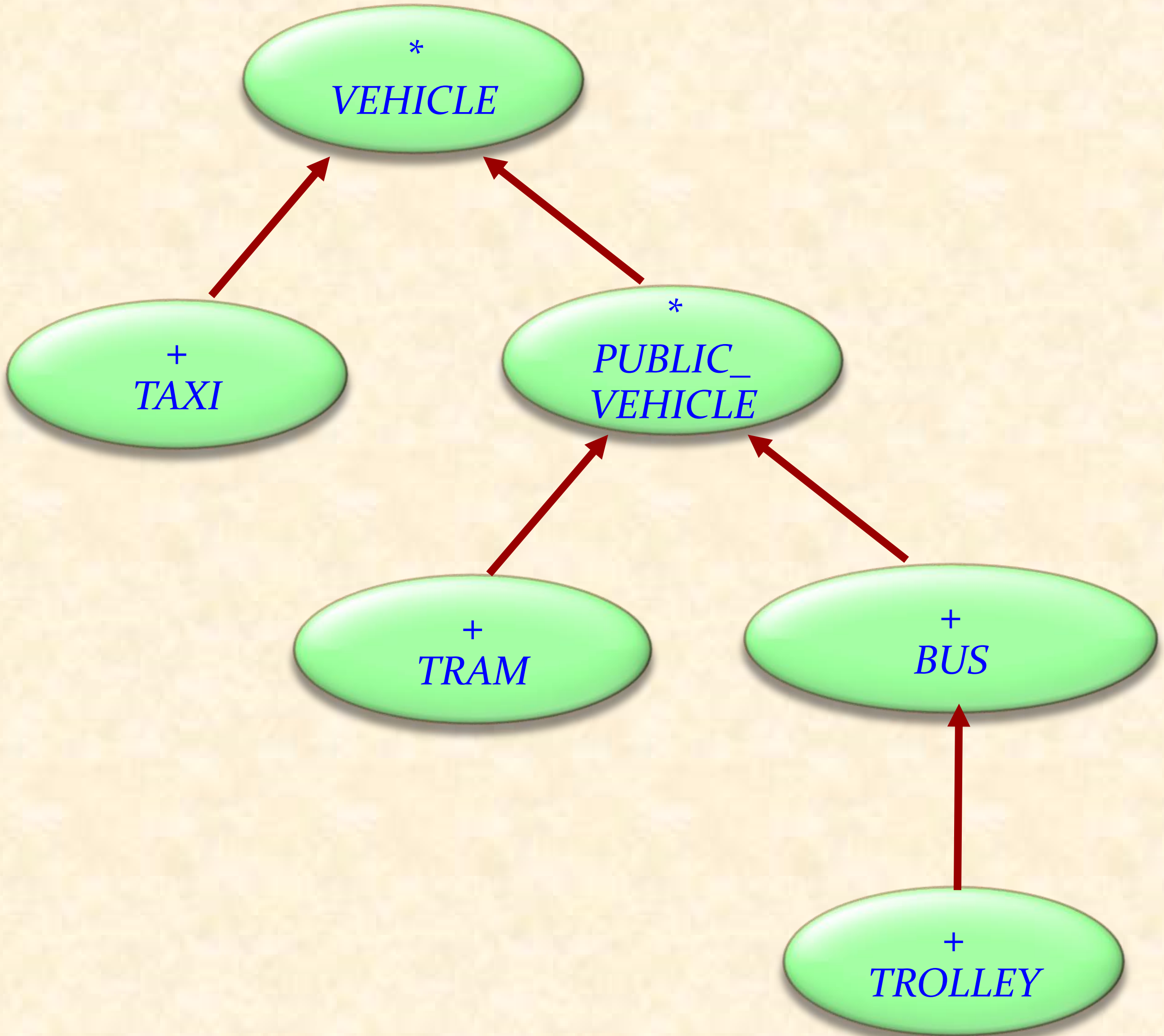
Visitor Library interface (2/2)

```
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*)

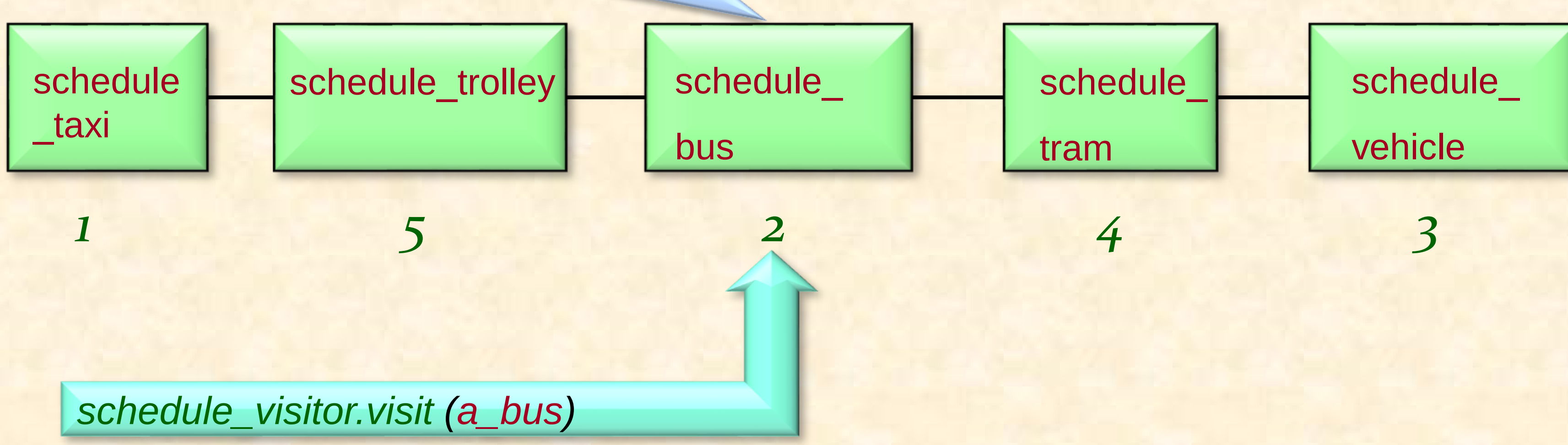
schedule_visitor.extend (agent *schedule_bus*)

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

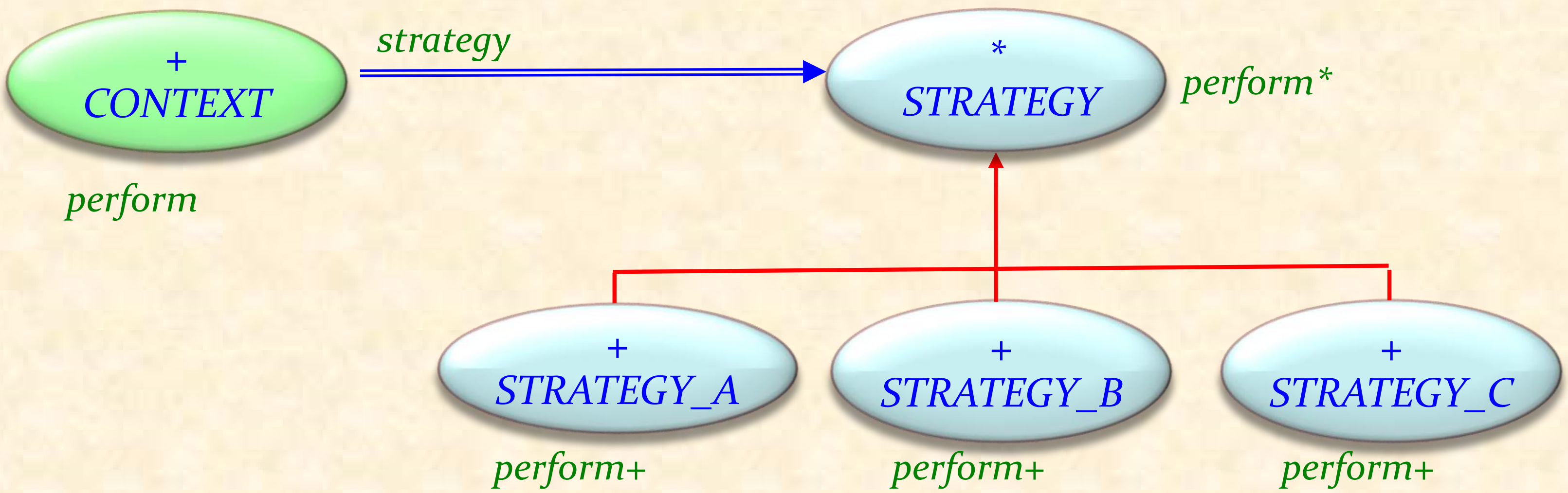
selecting a sorting algorithm on-the-fly

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*

Now, what if a new algorithm is needed ?

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*)

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

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
```

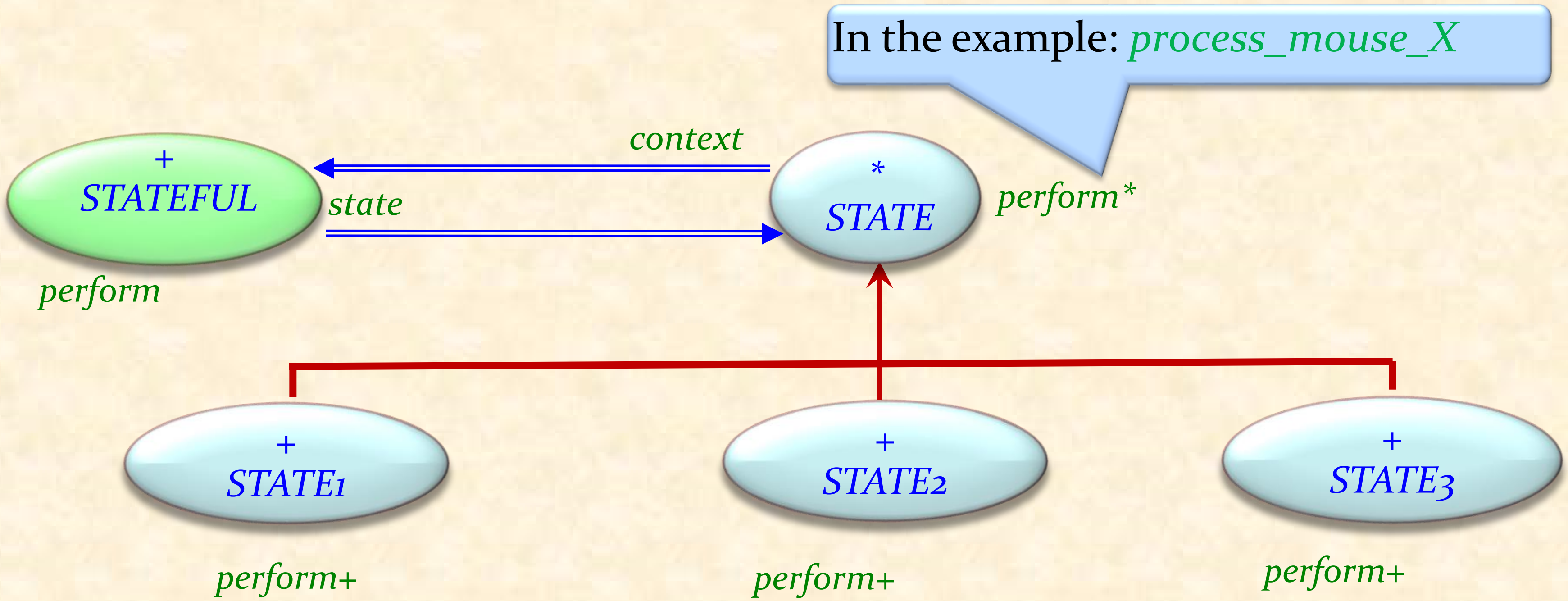
A stateful environment client

```
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

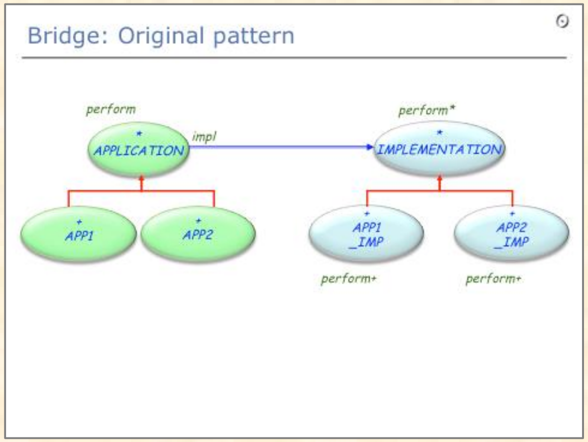
defines an interface for encapsulating the behavior associated with a particular state of the Context.

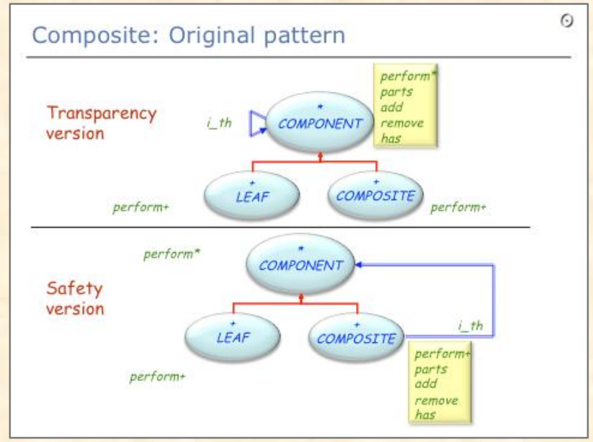
Concrete state

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

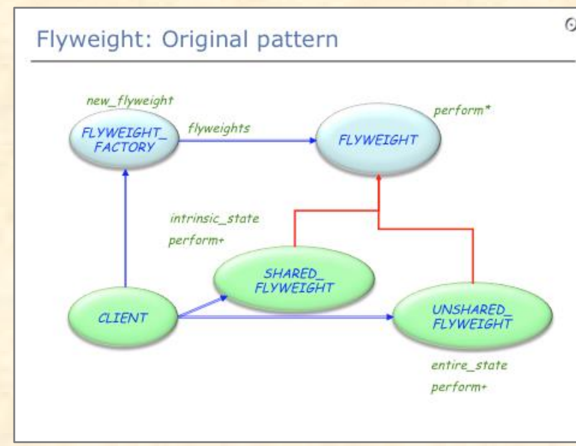
Summary of patterns – Structural patterns

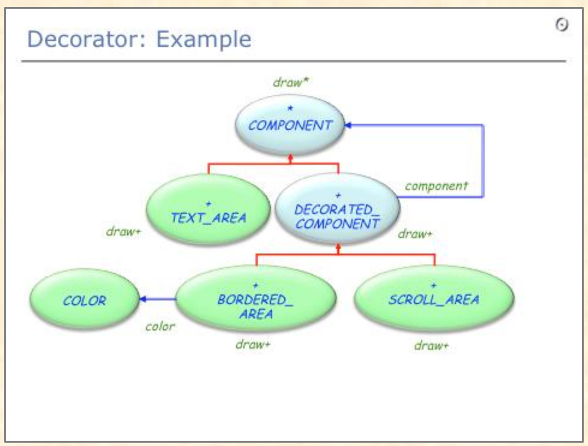
Bridge: Separation of interface from implementation



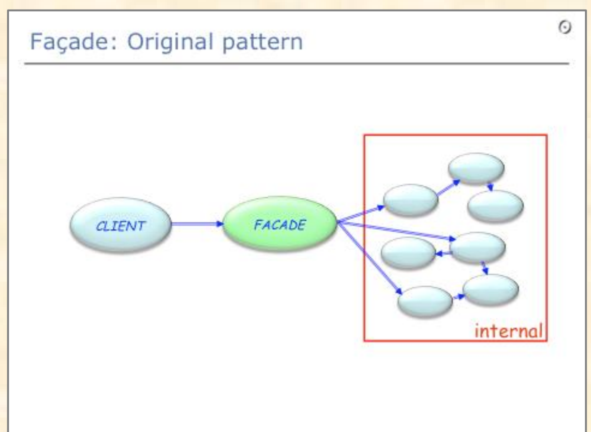


Composite: Uniform handling of compound and individual objects





Façade: A unified interface to a subsystem

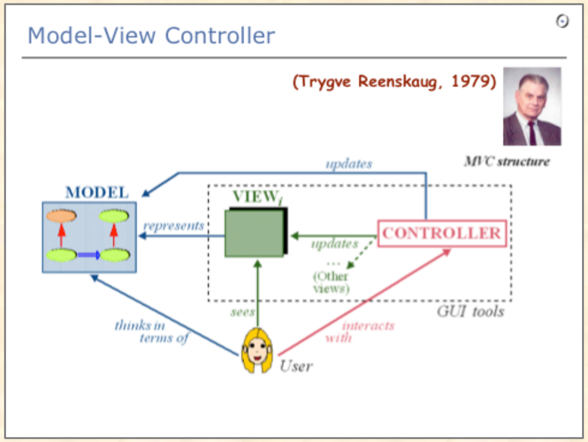


Flyweight: Share objects and externalize state

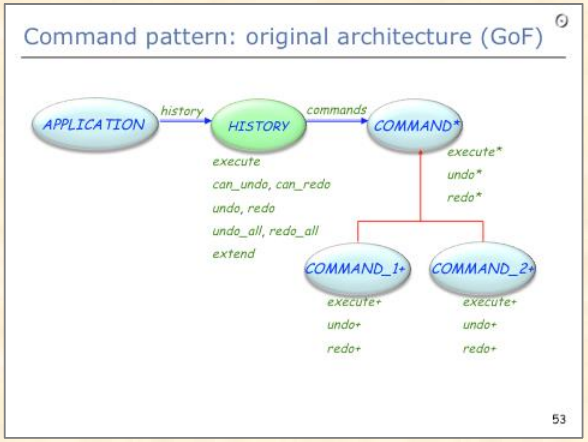
Decorator: Attaching responsibilities to objects without subclassing

Summary of patterns – Behavioral patterns

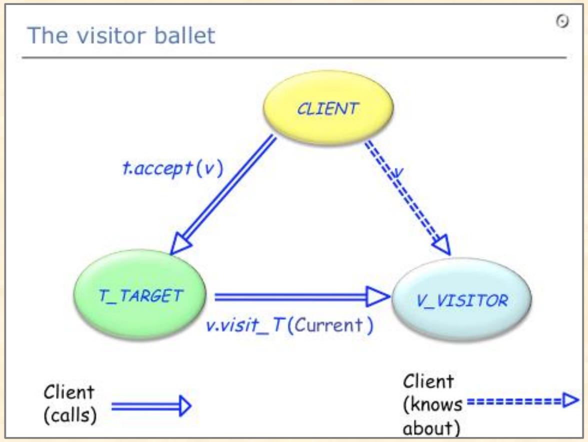
Observer; MVC: Publish-subscribe mechanism (use *EVENT_TYPE* with agents!); Separation of model and view



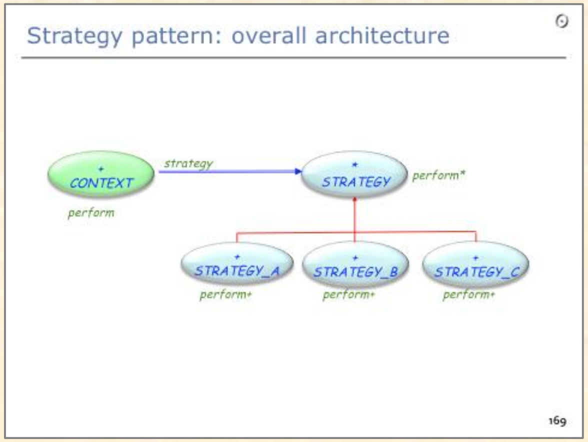
Command: History with undo/redo (use version with agents!)



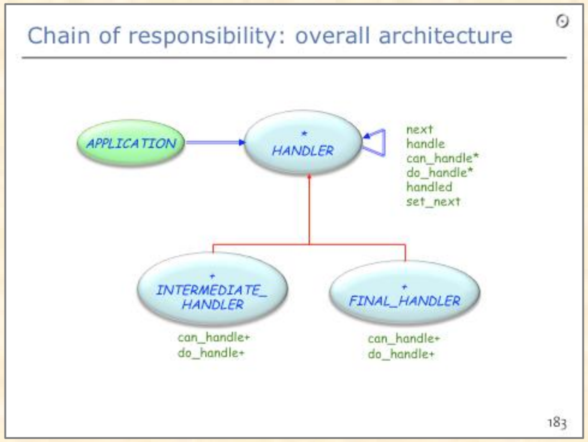
Visitor: Add operations to object hierarchies without changing classes



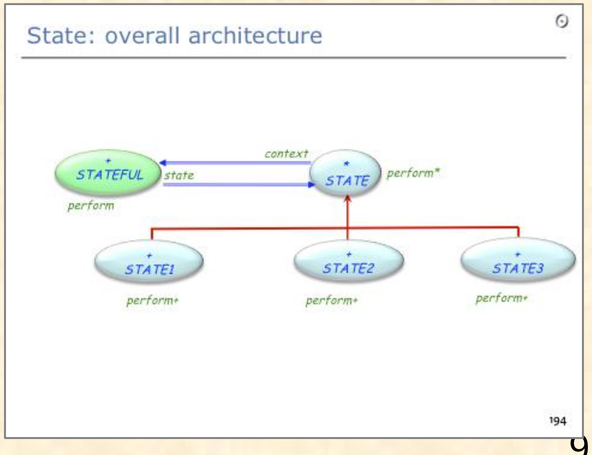
Strategy: Make algorithms interchangeable



Chain of responsibility: Allow multiple objects to handle request



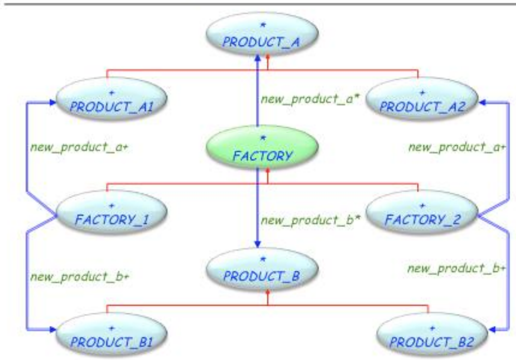
State: Object appears to change behavior if state changes



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:

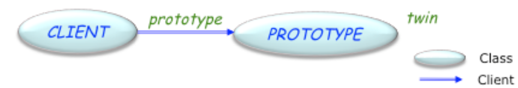
- Creates one object, not families of object.
- Works at the routine level, not class level.
- Helps a class perform an operation, which requires creating an object.
- Features *new* and *new_with_args* of the Factory Library are factory methods

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]



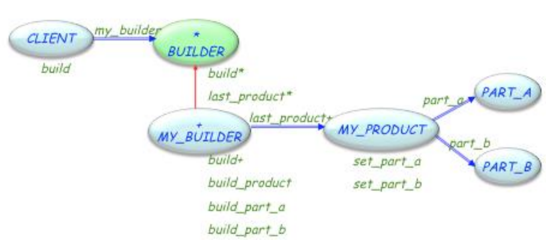
No need for this in Eiffel: just use function *twin* from class *ANY*.

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

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