Object-Oriented Software Construction

Bertrand Meyer

Lecture 18:

From design patterns to components

Agenda for today

- Design patterns
- A successful story: the Observer pattern
- From patterns to components

Agenda for today

- Design patterns
- A successful story: the Observer pattern
- From patterns to components

Benefits of design patterns

- Capture the knowledge of experienced developers
- Publicly available “repository”
- Newcomers can learn them and apply them to their design
- Yield a better structure of the software (modularity, extendibility)
- Common pattern language
- Facilitate discussions between programmers and managers

However: 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.*"

Erich Gamma et al., *Design Patterns*, 1995

→ NOT REUSABLE

A step backwards

- A step backwards from reuse:
  - No available “pattern libraries”
  - Programmers need to implement them each time anew
  - A pedagogical tool, not a reuse tool

“A successful pattern cannot just be a book description: it must be a software component”

Bertrand Meyer: *OOSC2*, 1997

Software reuse vs. design reuse

- “*Reuse of architectural and design experience is probably the single most valuable strategy in the basket of reuse ideas*”

  Clemens Szyperski, *Component software*, 1998

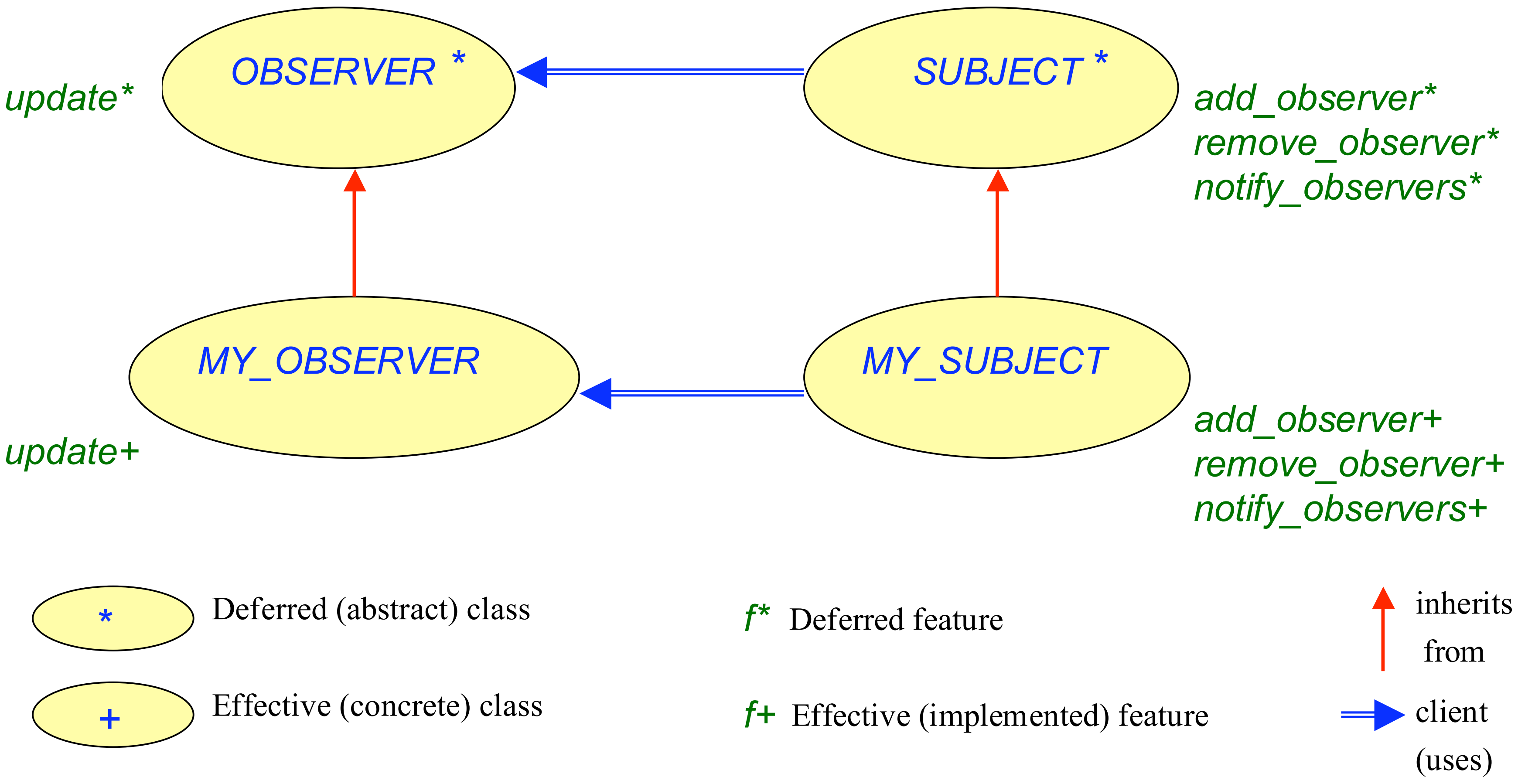
- Software reuse vs. design reuse:
  - Not much different with seamless development

- Combining both worlds:
  - From patterns to Eiffel components…

Agenda for today

- Design patterns
- A successful story: the Observer pattern
- From patterns to components

A successful story: the Observer pattern



OBSERVER *

update*

SUBJECT *

add_observer*
remove_observer*
notify_observers*

MY_OBSERVER

update+

MY_SUBJECT

add_observer+
remove_observer+
notify_observers+

* Deferred (abstract) class

\+ Effective (concrete) class

*f** Deferred feature

f+ Effective (implemented) feature

inherits from

client (uses)

Class SUBJECT

```
deferred class SUBJECT feature -- Observer pattern

	add_observer (an_observer: OBSERVER) is
				-- Add an_observer to the list of observers.
		require
				not_yet_an_observer: not observers.has (an_observer)
		do
				observers.extend (an_observer)
		ensure
				observer_added: observers.has (an_observer)
				one_more: observers.count = old observers.count + 1
		end

	remove_observer (an_observer: OBSERVER) is
				-- Remove an_observer from the list of observers.
		require
				is_an_observer: observers.has (an_observer)
		do
				observers.search (an_observer)
				observers.remove
		ensure
				observer_removed: not observers.has (an_observer)
				one_less: observers.count = old observers.count – 1
		end
```

Class SUBJECT (cont'd)

```
notify_observers is
            -- Notify all observers.
            -- (Call update on each observer.)
    do
            from
                    observers.start
            until
                    observers.after
            loop
                    observers.item.update
                    observers.forth
            end
    end

observers: LINKED_LIST [OBSERVER]
            -- List of observers

invariant

observers_not_void: observers /= Void

end
```

Class OBSERVER

```
deferred class OBSERVER feature -- Observer pattern

   update is
                  -- Update observer according to the state of
                  -- subject data.
         deferred
         end

   data: SUBJECT
         -- Observable data
end
```

A typical OBSERVER

```
class MY_DISPLAY inherit
	OBSERVER
		redefine
			data
		end
create
	make
feature -- Initialization
	make is
			-- Initialize GUI and register an observer of data.
		do
			create add_button.make_with_text_and_action ("Add", agent on_add)
			create remove_button.make_with_text_and_action ("Remove", agent on_remove)
			data.add_observer (Current)
		end
feature -- Access
	add_button: EV_BUTTON
			-- Button with label Add

	remove_button: EV_BUTTON
			-- Button with label Remove
```

A typical OBSERVER (cont'd)

```
    data: MY_DATA
                    -- Data to be observed

feature -- Event handling

    on_add is
                    -- Action performed when add_button is pressed
        do
                    data.add
        end

    on_remove is
                    -- Action performed when remove_button is pressed
        do
                    data.remove
        end

feature -- Observer pattern

    update is
                    -- Update GUI.
        do
                    -- Something here
        end

end
```

A typical SUBJECT

```
class MY_DATA inherit

    SUBJECT

feature -- Observer pattern

    add is
                -- Add Current to data to be observed.
            do
                -- Do something.
                notify_observers
            end

    remove is
                -- Remove Current from data to be observed.
            do
                -- Do something.
                notify_observers
            end

end
```

Redundancy:

→ Hardly maintainable

→ Not reusable

The Event library

- Basically:
  - One generic class: *EVENT_TYPE*
  - Two features: *publish* and *subscribe*
- For example: A button *my_button* that reacts in a way defined in *my_procedure* when clicked (event *mouse_click*):

Example using the Event library

- The publisher ("subject") creates an event type object:

```
mouse_click: EVENT_TYPE [TUPLE [INTEGER, INTEGER]] is
            -- Mouse click event type
    once
            create Result
    ensure
            mouse_click_not_void: Result /= Void
    end
```

- The publisher triggers the event:

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

- The subscribers ("observers") subscribe to events:

```
my_button.mouse_click.subscribe (agent my_procedure)
```

An encouraging success

- A book idea: the Observer pattern
- A reusable library: the Event library

Let’s go further and explore all design patterns…

Agenda for today

- Design patterns
- A successful story: the Observer pattern
- From patterns to components

Objectives

- A new classification of the design patterns described in Gamma et al.:
  - Artificial design patterns
  - Reusable design patterns
  - Remaining design patterns
- A “pattern library” made of the reusable components obtained from design patterns
- Code templates otherwise

Creational design patterns

Artificial design patterns	Reusable design patterns	Remaining design patterns
▪Prototype	▪Abstract Factory ▪Factory Method	▪Builder ▪Singleton

Creational design patterns

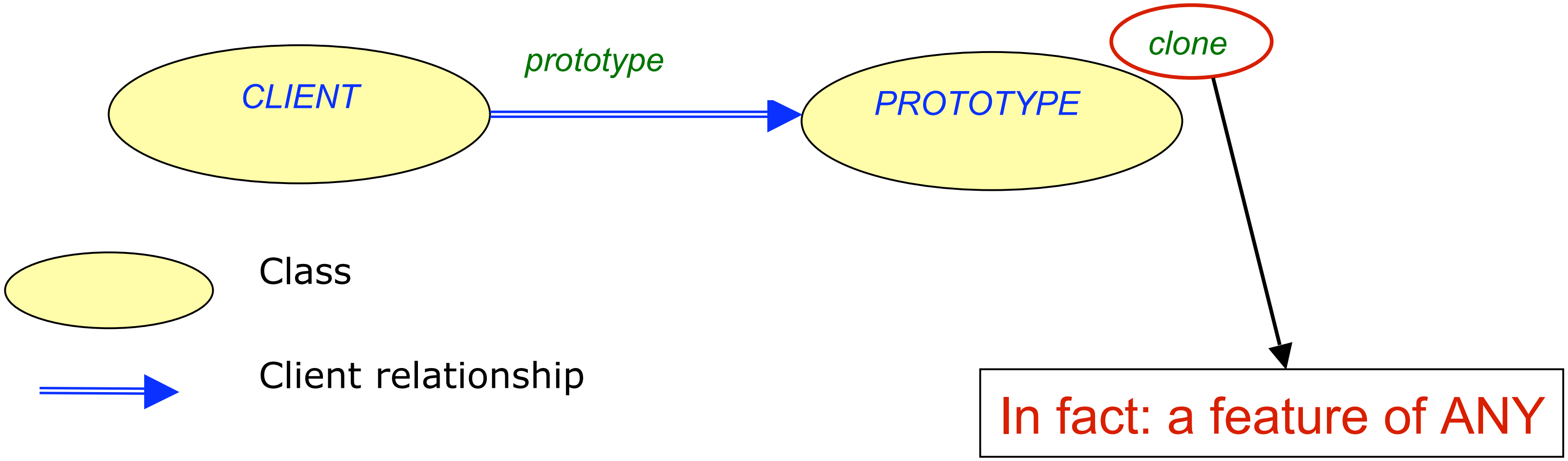
- Prototype
- Abstract Factory
- Factory Method
- Builder
- Singleton

Creational design patterns

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

Prototype: an artificial DP

- Intent:
  - *"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype."* [Gamma 1995, p 117]



CLIENT

prototype

PROTOTYPE

clone

Class

Client relationship

In fact: a feature of ANY

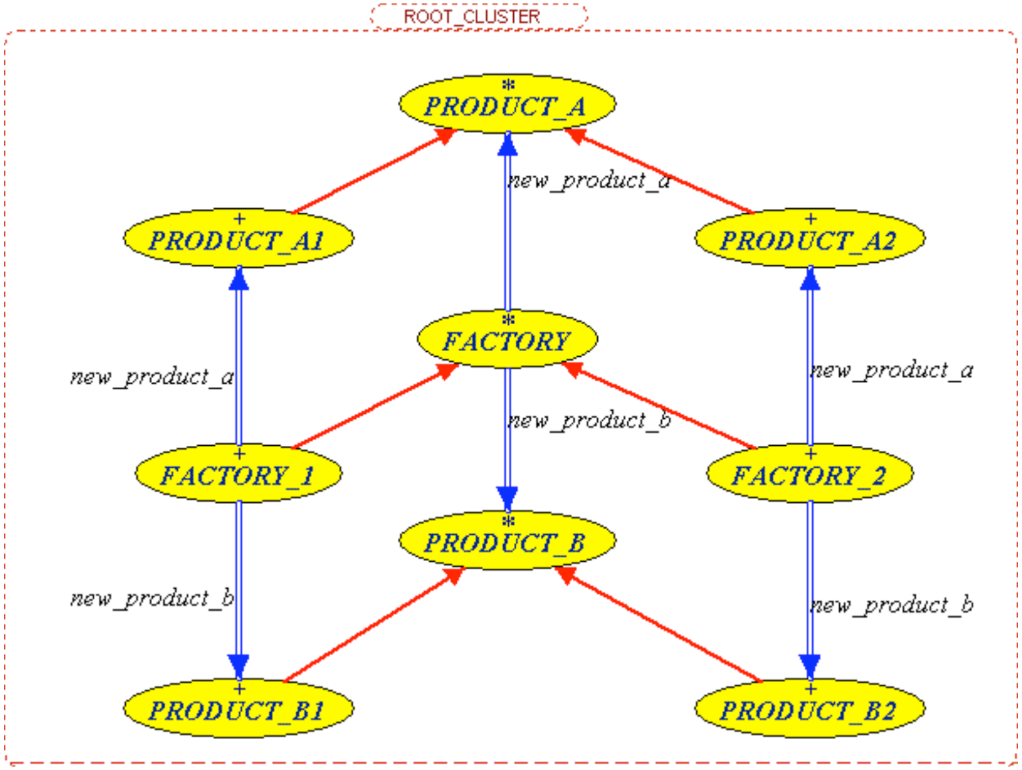
In Eiffel, every object is a prototype!

Creational design patterns

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

Abstract Factory: a reusable DP

- Intent:
  - *“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”* [Gamma 1995, p 87]



Class FACTORY

```
deferred class FACTORY feature -- Factory methods

    new_product_a: PRODUCT_A is
                    -- New product of type PRODUCT_A
        deferred
        ensure
                    product_a_not_void: Result /= Void
        end

    new_product_b: PRODUCT_B is
                    -- New product of type PRODUCT_B
        deferred
        ensure
                    product_b_not_void: Result /= Void
        end

end
```

Class FACTORY_1

```
class FACTORY_1 inherit

    FACTORY

feature -- Factory methods

    new_product_a: PRODUCT_A1 is
                    -- New product of type PRODUCT_A1
        do
                    create Result
        end

    new_product_b: PRODUCT_B1 is
                    -- New product of type PRODUCT_B1
        do
                    create Result
        end

end
```

Flaws of the approach

- Code redundancy:
  - *FACTORY_1* and *FACTORY_2* will be similar

- Lack of flexibility:
  - *FACTORY* fixes the set of factory functions *new_product_a* and *new_product_b*

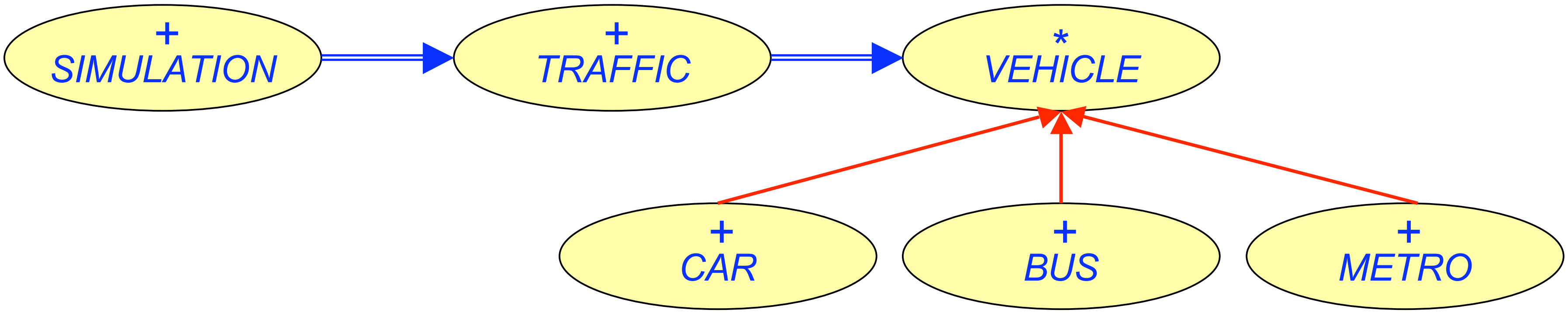
The Factory library

```
class FACTORY [G] create

    make

feature -- Initialization

    make (a_function: like factory_function) is
                    -- Set factory_function to a_function.
            require
                    a_function_not_void: a_function /= Void
            do
                    factory_function := a_function
            ensure
                    factory_function_set: factory_function = a_function
            end

feature -- Access

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

The Factory library (cont’d)

```
feature – Factory methods

    new: G is
                    -- New instance of type G
        do
                    factory_function.call ([])
                    Result := factory_function.last_result
        ensure
                    new_not_void: Result /= Void
        end

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

invariant

    factory_function_not_void: factory_function /= Void

end
```

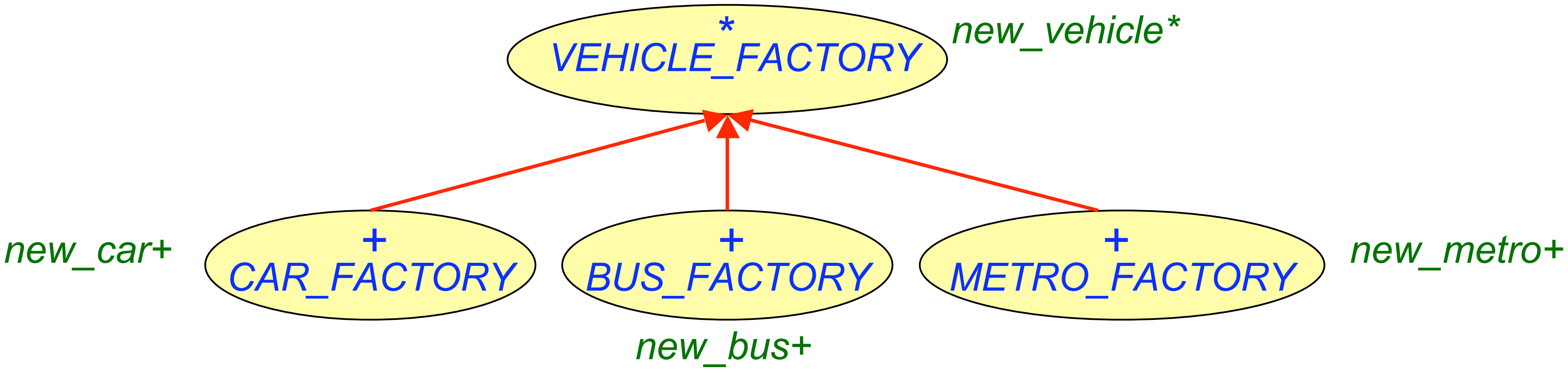
Sample application



simulated_traffic: *TRAFFIC*

simulated_traffic.*add_vehicle* (...)

With the Abstract Factory DP



```
simulated_traffic.add_vehicle (
    car_factory.new_car (a_power,
                         a_wheel_diameter,
                         a_door_width,
                         a_door_height)
                         )
```

With:

```
car_factory: CAR_FACTORY is
        -- Factory of cars
    once
        create Result
    ensure
        car_factory_not_void: Result /= Void
    end
```

With the Factory library

```
simulated_traffic.add_vehicle (
        car_factory.new_with_args ([a_power,
                                     a_wheel_diameter,
                                     a_door_width,
                                     a_door_height]
                                    )
                              )
```

With:

```
car_factory: FACTORY [CAR] is
        -- Factory of cars
    once
        create Result.make (agent new_car)
    ensure
        car_factory_not_void: Result /= Void
    end
```

With the Factory library (cont'd)

and:

```
new_car (a_power,a_diameter,a_width,a_height: INTEGER):CAR is
                    -- New car with power engine a_power,
                    -- wheel diameter a_diameter,
                    -- door width a_width, door height a_height
          do
                    -- Create car engine, wheels, and doors.
                    create Result.make (engine, wheels, doors)
          ensure
                    car_not_void: Result /= Void
          end
```

Factory pattern vs. library

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

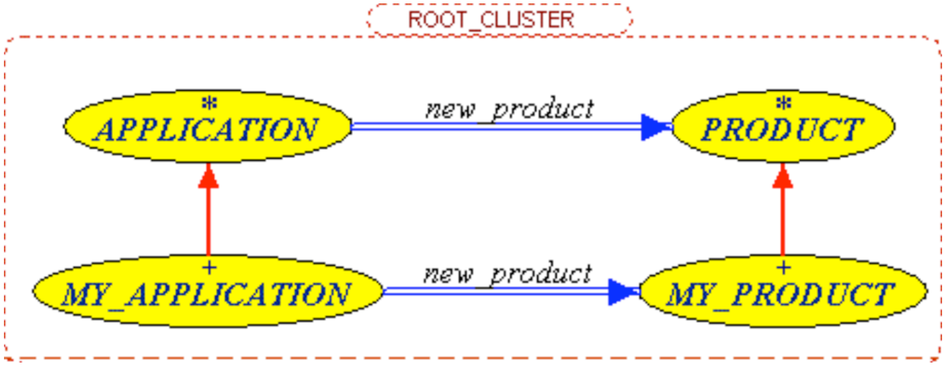
- One caveat though:
  - Likely to yield a bigger client class (because similarities cannot be factorized through inheritance)

Creational design patterns

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

Factory Method: a reusable DP

- Intent:
  - *"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."* [Gamma 1995, p 107]



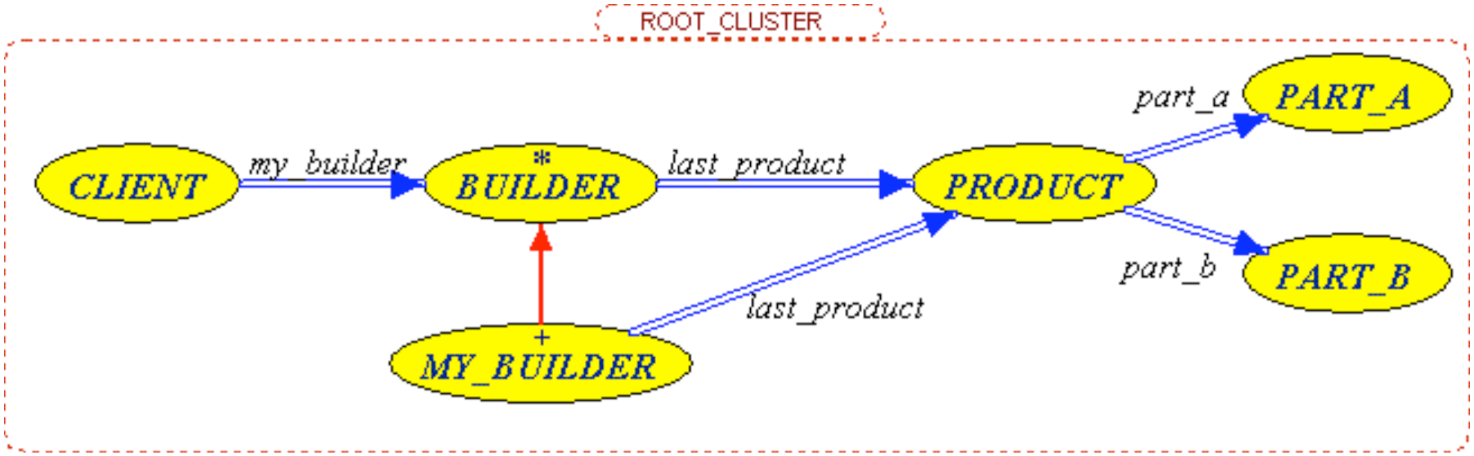
➡ A special case of the Abstract Factory

Creational design patterns

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

Builder: a remaining DP

- Intent:
  - *"Separate the construction of a complex object from its representation so that the same construction process can create different representations."* [Gamma 1995, p 97]



Class BUILDER

```
deferred class BUILDER feature -- Access

    last_product: PRODUCT is
                -- Product under construction
        deferred
        end

feature -- Basic operations

    build is
                -- Create and build last_product.
        do
                build_product
                build_part_a
                build_part_b
        ensure
                last_product_not_void: last_product /= Void
        end
    ...
end
```

A reusable builder?

- Issue:
  - How to know how many parts the product has?

  Not reusable

- Handle some usual cases, e.g. a “two part builder” by reusing the Factory library:

```
class TWO_PART_BUILDER [F -> BUILDABLE, G, H]
        -- Build a product of type F
        -- composed of two parts:
        -- the first part of type G,
        -- the second part of type H.
```

Class BUILDABLE

```
deferred class BUILDABLE feature -- Access

    g: ANY
                    -- First part of the product to be created

    h: ANY
                    -- Second part of the product to be created

feature {TWO_PART_BUILDER} -- Status setting

    -- set_g
    -- set_h

end
```

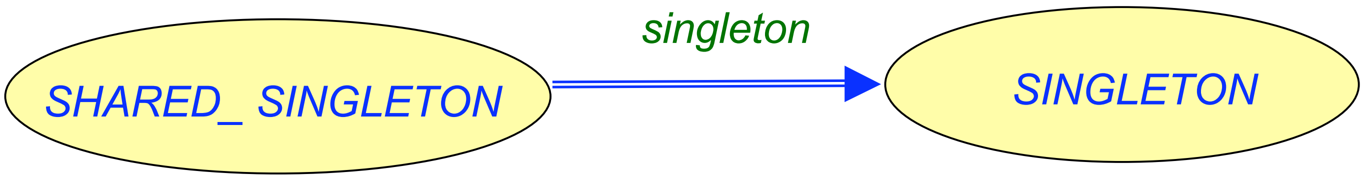
Creational design patterns

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

Singleton: a remaining DP

- Intent:
  - *“Ensure a class only has one instance, and provide a global point of access to it.”*

    [Gamma 1995, p 127]



SHARED_ SINGLETON

singleton

SINGLETON

Harder than it looks…

A wrong approach

```
class SINGLETON feature {NONE}

   frozen the_singleton: SINGLETON is
               -- The unique instance of this class
         once
               Result := Current
         end

invariant

   only_one_instance: Current = the_singleton

end
```

A wrong approach (cont’d)

```
deferred class SHARED_SINGLETON feature {NONE}

    singleton: SINGLETON is
                -- Access to unique instance
        deferred
        end

    is_real_singleton: BOOLEAN is
                -- Do multiple calls to singleton return the same result?
        do
                Result := singleton = singleton
        end

invariant

    singleton_is_real_singleton: is_real_singleton

end
```

What’s wrong?

- If one inherits from *SINGLETON* several times:
    - The inherited feature *the_singleton* keeps the value of the first created instance.
    - Violates the invariant of class *SINGLETON* in all descendant classes except the one for which the singleton was created first.

There can only be one singleton per system

A correct Singleton example

```
class MY_SHARED_SINGLETON feature -- Access
    singleton: MY_SINGLETON is
                        -- Singleton object
            do
                        Result := singleton_cell.item
                        if Result = Void then
                                create Result.make
                        end
            ensure
                        singleton_created: singleton_created
                        singleton_not_void: Result /= Void
            end
feature -- Status report
    singleton_created: BOOLEAN is
                        -- Has singleton already been created?
            do
                        Result := singleton_cell.item /= Void
            end
feature {NONE} -- Implementation
    singleton_cell: CELL [MY_SINGLETON] is
                        -- Cell containing the singleton if already created
            once
                        create Result.put (Void)
            ensure
                        cell_not_void: Result /= Void
            end
end
```

A correct Singleton example (cont'd)

```
class MY_SINGLETON inherit
    MY_SHARED_SINGLETON
create
    make
feature {NONE} -- Initialization
    make is
                    -- Create a singleton object.
            require
                    singleton_not_created: not singleton_created
            do
                    singleton_cell.put (Current)
            end
invariant
    singleton_created: singleton_created
    singleton_pattern: Current = singleton
end
```

In fact, one can still break it by:

- Cloning a singleton.
- Using persistence.
- Inheriting from *MY_SHARED_SINGLETON* and putting back Void to the cell after the singleton has been created.

A Singleton in Eiffel: impossible?

- Having frozen classes (from which one cannot inherit) would enable writing singletons in Eiffel

- But it would still not be a reusable solution

Structural design patterns

Artificial design patterns	Reusable design patterns	Remaining design patterns
	▪Composite ▪Flyweight	▪Proxy
		▪Decorator ▪Adapter ▪Bridge ▪Facade

Behavioral design patterns

- Not done yet
- But can expect DP like the Visitor and the Strategy to be reusable through the Eiffel agent mechanism

References: Design patterns

- Gamma et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

- J_z_quel et al.: *Design Patterns and Contracts*, Addison-Wesley, 1999.

References: From patterns to components

- Karine Arnout. Contracts and tests. Ph.D. research plan, December 2002. Available from http://se.inf.ethz.ch/people/arnout/phd_research_plan.pdf
- Karine Arnout, and Bertrand Meyer. “From Design Patterns to Reusable Components: The Factory Library”. Available from http://se.inf.ethz.ch/people/arnout/arnout_meyer_factory.pdf
- Karine Arnout, and Éric Bezault. “How to get a Singleton in Eiffel?”. Available from http://se.inf.ethz.ch/people/arnout/arnout_bezault_singleton.pdf
- Volkan Arslan. Event library (sources). Available from http://se.inf.ethz.ch/people/arslan/data/software/Event.zip
- Volkan Arslan, Piotr Nienaltowski, and Karine Arnout. “Event library: an object-oriented library for event-driven design”. JMLC 2003. Available from http://se.inf.ethz.ch/people/arslan/data/scoop/conferences/Event_Library_JMLC_2003_Arslan.pdf
- Bertrand Meyer. “The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design”. Available from http://www.inf.ethz.ch/~meyer/ongoing/events.pdf

End of lecture 18