Last update: 25 May 2007

Software Architecture

Bertrand Meyer

ETH Zurich, March-July 2007

Lecture 10: More patterns: Factory, Builder, Singleton

Creational patterns

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

Some design patterns

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

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

Explicit creation in O-O languages

Eiffel:

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

C++, Java, C#:

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

Some design patterns

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

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

Behavioral
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Abstract factory

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

Abstract Factory: example

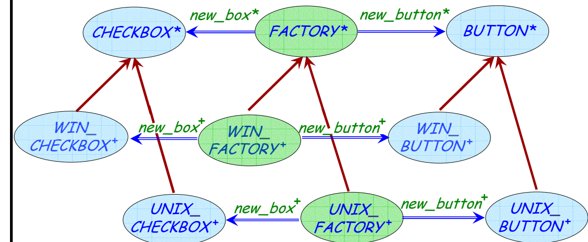
Widget toolkit (EiffelVision, Java Swing)

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

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

→Creation of widget objects should not be distributed

Abstract widget factory example



Abstract Factory usage

Abstract Factory 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

Class *FACTORY*

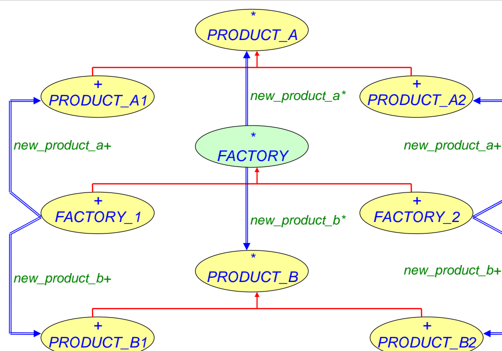
```
deferred class
    FACTORY

feature -- Basic operations

    new_button: BUTTON is
            -- New button
        deferred
        end

    new_checkbox: CHECKBOX is
            -- New checkbox
        deferred
        end
…
end
```

Abstract factory architecture



An example concrete factory: *WIN_FACTORY*

```
class
    WIN_FACTORY
inherit
    FACTORY

feature -- Basic operations
    new_button: BUTTON
            -- New Windows button
        do
            create {WIN_BUTTON} Result
        end

    new_checkbox: CHECKBOX
            -- New Windows checkbox
        do
            create {WIN_CHECKBOX} Result
        end
end…
```

Shared ancestor for factory clients

```
class
    SHARED_FACTORY
...
feature -- Basic operations

    factory: FACTORY
            -- Factory used for widget instantiation
        once
            if is_windows_os then
                create {WIN_FACTORY} Result
            else
                create {UNX_FACTORY} Result
            end
        end

...
end
```

Abstract factory pattern: properties

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

Usage of *FACTORY*

```
class
    WIDGET_APPLICATION
inherit
    SHARED_FACTORY
...
feature -- Basic operations
    some_feature
            -- Generate a new button and use it.
        local
            my_button: BUTTON
        do
            ...
            my_button := factory.new_button
            ...
        end
...
end
```

Abstract notion

Does not name platform

Abstract factory pattern: criticism

Code redundancy:

- The factory classes, e.g. *UNIX_FACTORY* and *WIN_FACTORY* will be similar

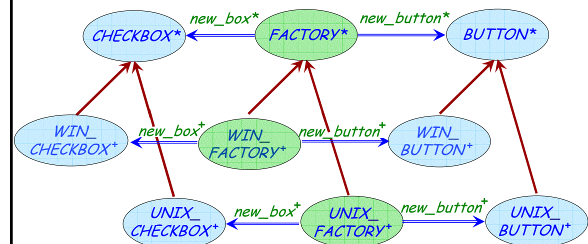
Lack of flexibility:

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

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

Our factory example



Beyond patterns

A pattern is an architectural solution

Each programmer who needs the pattern has to learn it (externals and internals) and reimplement it for every application)

Similar to traditional learning of algorithms and data structures

Can we do better: use components instead?

It's better to reuse than do redo.

(if reuse occurs through an API)

The Factory library

```
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
```

Components over patterns

Easier to learn

No need to learn implementation

Professional implementation will be better than manually crafted one

But: do we lose generality?

The Factory library

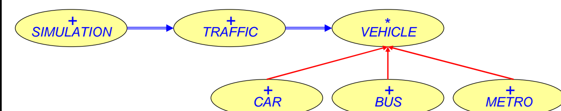
```
feature -- Factory operations
    new: G
            -- New instance of type G
        do
            factory_function.call ([])
            Result := factory_function.last_result
        ensure
            exists: Result /= Void
        end

    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
```

The key to pattern componentization

- Genericity
- Constrained genericity
- Multiple inheritance
- Agents
- Contracts (to make sure we get everything right)

Sample application

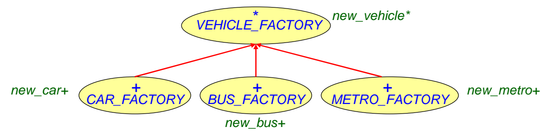


In class *SIMULATION*:

simulated_traffic: *TRAFFIC*

simulated_traffic.add_vehicle (…)

With the Abstract Factory pattern




```
simulated_traffic.add_vehicle (
    car_factory.new_car (p, d, w,h))
```

With:

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

Factory library: create several products

An instance of *FACTORY* describes one kind of product:

```
class
    FACTORY [G]
...
feature -- Factory functions
    new: G ...
        -- New instance of type G
    new_with_args (args: TUPLE): G ...
        -- New instance of type G initialized with args
end
```

Use several factory objects to create several products:

```
class
    LIBRARY
...
feature -- Factories
    fb: FACTORY [CAR]
    fu: FACTORY [TRUCK]
end
```

With the Factory library

```
simulated_traffic.add_vehicle (
        car_factory.new_with_args ([p, d, w, h]))
```

With:

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

Factory pattern vs. library

Benefits:

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

Limitation:

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

With the Factory library

and:

```
new_car (p, d, w, h: INTEGER):CAR
            -- New car with power engine p,
            -- wheel diameter d,
            -- door width w, door height h
        do
            -- Create car engine, wheels, and doors.
            create Result.make (engine, wheels, doors)
        ensure
            exists: Result /= Void
        end
```

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

Explicit creation in O-O languages

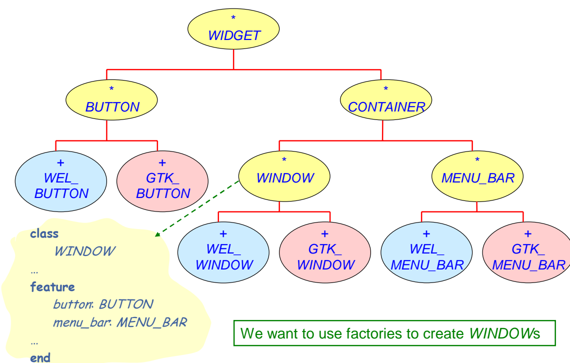
Eiffel:

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

C++, Java, C#:

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

Managing parallel hierarchies with factories



With an Abstract Factory (1/6)

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

With an Abstract Factory (2/6)

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

Factory ensures that all widgets of the window are Unix 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
```

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
```

With the Factory Library (1/3)

The Factory Library can create only one kind of product

```
class
    FACTORY [G]
...
feature -- Factory functions
    new: G is ...
            -- New instance of type G
    new_with_args (args: TUPLE): G is ...
            -- New instance of type G initialized with args
end
```

Use several factory objects to create several products

```
class
    LIBRARY
...
feature -- Factories
    fb: FACTORY [BOOK]
    fu: FACTORY [USER]
end
```

With the Factory Library (2/3)

```
deferred class
    APPLICATION
...
feature -- Initialization
    build_window is
            -- 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
```

With the Factory Library (3/3)

```
class
    WEL_APPLICATION
inherit
    APPLICATION
create
    make
feature
    make is
            -- 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*

With an Abstract Factory

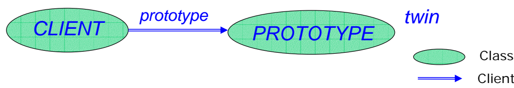
```
class
    WEL_WINDOW_FACTORY
inherit
    WINDOW_FACTORY
create
    make
feature {NONE} -- Initialization
    make (...) is ...
feature -- Factory functions
    new_window: WEL_WINDOW is ...
    new_button: WEL_BUTTON is ...
    new_menu_bar: WEL_MENU_BAR is ...
...
end
```

Factory ensures that all widgets of the window are Windows widgets

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*)

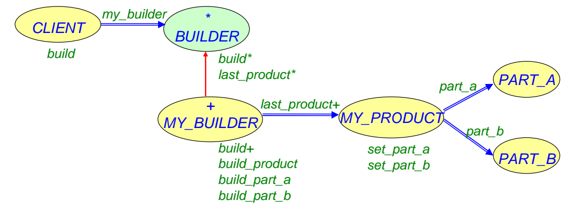
Builder pattern

Purpose

- "*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.

Builder pattern



Builder Library

Mechanisms enabling componentization: unconstrained genericity, agents

+ Factory 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
```

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

⇒ 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 is ...
    new_part_a (a_name: STRING): PART_A is ...
    new_part_b (a_name: STRING): PART_B is ...
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) is ...
    build_h (args_h: TUPLE) is ...
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

⇒ Incomplete support of the Builder pattern ("Componentizable but non-comprehensive")

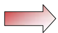
Basic Eiffel singleton mechanism

Once routines

But: does not prevent cloning

Singleton pattern

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



Once routines

If instead of

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

you write

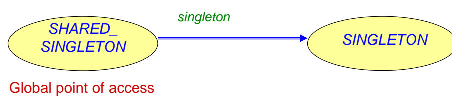
```
r is
    once
        ... Instructions ...
    end
```

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.

Singleton pattern

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



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
```

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

Singleton and cloning in Eiffel

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 ⇒ possible to have singletons

Singleton pattern use

```
deferred class
    SHARED_SINGLETON
feature {NONE} -- Implementation
    singleton: SINGLETON is
                    -- Access to a unique instance. (Should be redefined
                    -- as once function in concrete descendants.)
        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
```

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*)

Singleton pattern issue

Problem: Allows only one singleton per system

the_singleton: once function inherited by all descendants of *SINGLETON*

- ⇒ would keep the same value
- ⇒ would violate the invariant of *SINGLETON* in all its descendants, except the one for which the singleton was created first

Basic singleton implementation*

```
class
    SINGLETON
feature {NONE} -- Implementation
    frozen the_singleton: SINGLETON
                -- Unique instance of this class
        once
                Result := Current
        end
invariant
    only_one_instance: Current = the_singleton
end
```

*Jézéquel, Train, Mingins, *Design Patterns and Contracts*, Addison-Wesley 1999

Singleton and system correctness

The Singleton property

"There exists only one object of this class"

is a global invariant of the system.

However, Eiffel assertions are only at a class-level, not at the system-level.

For a guaranteed singleton property in current Eiffel: see Karine Arnout's thesis

Frozen classes

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 without frozen classes

Frozen classes require the ability to restrict the exportation of creation procedures (constructors)
⇒ Not applicable in Java and C++

Java and C++ use static features to implement the Singleton pattern:

- A class Singleton with a protected constructor and a static function Instance() that creates the singleton if it was not yet created, otherwise returns it (use of a private field _instance).

Complementary material (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.
- Paul Cohen. "Re: Working singleton implementation". *comp.lang.eiffel*. http://groups.google.com/groups?dq=&hl=en&lr=&ie=UTF-8&selm=3AD984B6.CCEC91AA%40enea.se&rnum=8.

Complementary material (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 (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.