Trusted Components

Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 24: Singleton
Componentizability classification

1. Componentizable
   - 1.1 Built-in
   - 1.2 Library-supported
   - 1.3 Newly componentized
   - 1.4 Possible component

   1.3.1 Fully componentizable but not comprehensive
   - Flyweight
   - Observer
   - Mediator
   - Abstract Factory
   - Factory Method
   - Visitor
   - Command
   - Composite
   - Chain of Responsibility

   1.3.2 Componentizable but not comprehensive
   - Builder
   - Proxy
   - State

   1.3.3 Componentizable but unfaithful
   - Strategy

   1.3.4 Componentizable but useless
   - Memento

2. Non-componentizable
   - 2.1 Skeleton
   - 2.2 Possible skeleton
   - 2.3 Some library support
   - 2.4 Design idea

   2.1.1 Method
   - Decorator
   - Adapter

   2.1.2 No method
   - Template Method
   - Bridge

   2.2.1 Possible skeleton
   - Iterator

   2.3.1 Some library support
   - Facade

   2.4.1 Design idea
   - Interpreter
Agenda for today

- Singleton pattern
- Tentative implementations
- Language extension proposals
- Componentization outcome
Singleton pattern

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

What about cloning?!
Singleton and clone in Eiffel

- Class *ANY* has features *clone, 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

- **Our approach:**
  - Find the best implementation of Singleton with current Eiffel
  - Take for granted that the cloning problem is solved and consider language extension to have a fully correct implementation of Singleton
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`)
Agenda for today

- Singleton pattern
- Tentative implementations
- Language extension proposals
- Componentization outcome
Singleton pattern

- Way to “ensure a class only has one instance, and to provide a global point of access to it.” [GoF, p 127]
The Design Patterns and Contracts approach (1/3)

class
SINGLETON

feature {NONE} -- Implementation

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

invariant
    only_one_instance: Current = the_singleton
end
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
- **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
Singleton skeleton (1/2)

```plaintext
class SHARED_SINGLETON
  feature {NONE} -- Implementation
    singleton: SINGLETON is
      -- Access to a unique instance
      once
        create Result
        ensure
          singleton_not_void: Result /= Void
      end
      is_real_singleton: BOOLEAN is
        -- Do multiple calls to singleton return the same result?
        do
          Result := singleton = singleton
        end
      invariant
        is_real_singleton: is_real_singleton
    end
end
```

Provides a global point of access to the singleton

Same problem as previous solution with the once function
Singleton skeleton (2/2)

```hamlet
deferred class

SINGLETON

feature \{NONE\} -- Access

singleton: SINGLETON is

    -- Effect this as a (frozen) once routine.
    -- It should return Current.

defered
end

invariant

remain_single: Current = singleton
end
```

Class SINGLETON is not sound: Clients have no way to ensure the invariant is satisfied
Class correctness: Definition

- A class is correct with respect to its assertions if and only if:
  - C1: For any valid set of arguments $xp$ to a creation procedure $p$:
    \[ \{ \text{Default c} \text{ and Prep (xp)} \} \text{ Bodyp \{Postp (xp) and INV}} \]
  - C2: For every exported routine $r$ and any set of valid arguments $xr$:
    \[ \{ \text{Prer (xr) and INV} \} \text{ Bodyr \{Postr (xr) and INV} \]

- For class `SINGLETON` to be correct, we should be able to write:
  ```plaintext
  s: SINGLETON
  if is_valid_to_create_a_new_instance then
    create s.make
  else
    -- Either report an error or try to return a
    -- reference to the already created object.
  end
  ```
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.**
class
MY_SHARED_SINGLETON

feature -- Status report
may_create_singleton: BOOLEAN is
    -- May a new singleton be created?
    -- (i.e. is there no already created singleton?)
do
    Result := not singleton_created.item
end

feature -- Access
singleton: MY_SINGLETON is
    -- Access to unique instance
once
    create Result.make (Current)
singleton_created.set_item (True)
ensure
    singleton_not_void: Result /= Void
    may_not_create_singleton: not may_create_singleton
end
feature {NONE} -- Implementation

singleton_created: BOOLEAN_REF is
  -- Has singleton already been created?
  once
  create Result
  ensure
    result_not_void: Result /= Void
  end
end
class MY_SINGLETON
inherit SINGLETON
create make
feature {NONE} -- Initialization
  make (an_accessor: MY_SHARED_SINGLETON) is
    -- Create a singleton from an_accessor.
    require
      an_accessor_not_void: an_accessor /= Void
      may_create: an_accessor.may_create_singleton
    do
    end
feature {NONE} -- Implementation
  singleton: SINGLETON is
    -- Access to unique instance
    once
      Result := Current
    end
end
class MY_TEST
inherit MY_SHARED_SINGLETON
create make
feature {NONE} -- Initialization
  make is -- Create two instances of type MY_SINGLETION.
    local s1, s2: MY_SINGLETION
    do
      if may_create_singleton then
        create s1.make (Current)
      end
      if may_create_singleton then
        create s2.make (Current)
      end
    end
end

Creates two singletons!
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
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
  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
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
Limitations:
- Cloning
- Persistence

Inheritance from \texttt{MY\_SHARED\_SINGLETON} by “cheating” by putting back Void to the cell after the singleton has already been created

Soundness problem: Descendants of class \texttt{MY\_SINGLETON} may not have their own direct instances without violating the class invariant \texttt{Current = singleton}
Registry of singletons (1/2)

class \texttt{SINGLETON}

feature \{NONE\} -- Initialization
  \textbf{frozen register\_in\_system is}
  -- Register an instance of this singleton. (Must be called by every
  -- creation procedure of every descendants of \texttt{SINGLETON} to fulfill the
  -- class invariant \textit{is\_singleton}.)
  \textbf{require}
  no\_singleton\_in\_system: not singletons\_in\_system\_has (generating\_type)
  \textbf{do}
  singletons\_in\_system\_put (\texttt{Current}, generating\_type)
  \textbf{ensure}
  count\_increased: singletons\_in\_system\_count =
      old singletons\_in\_system\_count + 1
  singleton\_registered: singletons\_in\_system\_has (generating\_type)
end

feature \{NONE\} -- Implementation
  \textbf{frozen singletons\_in\_system: HASH\_TABLE [\texttt{SINGLETON, STRING}] is}
  -- All singletons in system stored by name of generating type
  \textbf{once}
  create \texttt{Result} make (1)
  \textbf{ensure}
  singletons\_in\_system\_not\_void: Result /= Void
end
Clients cannot test for the precondition because:

- Do not have access to `singletons_in_system`
- Do not know the value of `generating_type` because the object has not been created yet.
Agenda for today

- Singleton pattern
- Tentative implementations
- Language extension proposals
- Componentization outcome
Once creation procedures

- **Semantics of a creation procedure:**
  1. Allocate memory
  2. Initialize the fields with their default values
  3. Call the creation procedure (to ensure the invariant)
  4. Attach the resulting object to the creation entity (e.g. \( x \) in `create x.make`)

- **New semantics for once creation procedures:**
  - If not called yet, create an object following steps 1 to 4
  - Otherwise attach to the creation target entity the object created by the first call to the once creation procedure for this type

**Different once statuses** (i.e. is it the first call or a subsequent one?) according to whether it is called as a creation procedure or as a regular procedure
Another attempt with once procedures

- Wouldn’t it be possible to have a procedure `default_create` that calls a once procedure (that would create the single object)? (*)

- Once creation procedures forbidden in Eiffel (ETL2) because only the first created object would be initialized.

  (⇒ New semantics for the proposal about once creation procedures)

- (*) only shifts the problem:
  - It would allow creating any number of objects but only one would be initialized.
  - `default_create` is a procedure: does not return any object ⇒ Does not help!
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`
frozen class
  \textit{SHARED\_SINGLETON}
feature -- Access
  \texttt{singleton: SINGLETON is}
    -- Global access point to singleton
    once
    create \textit{Result}
    ensure
      singleton\_not\_void: \textit{Result} /\neq \textit{Void}
end
end

class
  \texttt{SINGLETON}
create \{\textit{SHARED\_SINGLETON}\}
  \texttt{default\_create}
end
Use of singleton with frozen classes

class  
MY_SHARED_SINGLETON

feature -- Access

singleton: SINGLETON is  
  -- Unique instance
  once
      Result :=
          (create {SHARED_SINGLETON}).singleton
  end
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`).
Agenda for today

- Singleton pattern
- Tentative implementations
- Language extension proposals
- Componentization outcome
Singleton: Componentization outcome (1/2)

- **Inheritance:** Once function `singleton` would be inherited by all descendants and would keep the value of the first created instance; all these descendants would share the same value. (See *The Design Patterns and Contracts* approach.)

- **Genericity:** Would mean having two classes `SINGLETON [G]` and `SHARED_SINGLETION [G]` where `G` corresponds to the type to turn into a singleton
  \[\Rightarrow\] once function `singleton` depending on a generic parameter; forbidden in Eiffel
Singleton: Componentization outcome (2/2)

- **Conversion**: Internally relies on creation ⇒ problem shift (it does not provide a way to restrict the creation of instances of certain types and the access to these instances).

- **Aspects**: Add behavior at a certain point of a routine execution; it does not provide a way to restrict the creation of instances of certain types and the access to these instances.

- **Agents**: Need to duplicate the once function singleton to support multiple singleton objects in a system (whether singleton is a function or a call to an agent does not change).
  - The Pattern Wizard generates skeleton classes ⇒ Removes the need to copy/paste those functions.
Complementary material (1/3)

- From Patterns to Components:
  - Chapter 18: Singleton

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


Further reading:

  [Link](http://groups.google.com/groups?q=Once+creation+procedures&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).
End of lecture 24