Design patterns. Creational software design pattern

Singleton pattern

1. Design pattern description

The singleton design pattern is one of the twenty-three well-known "Gang of Four" design patterns.

The singleton design pattern solves problems like:

- How can it be ensured that a class has only one instance?
- How can the sole instance of a class be accessed easily?
- How can a class control its instantiation?
- How can the number of instances of a class be restricted?

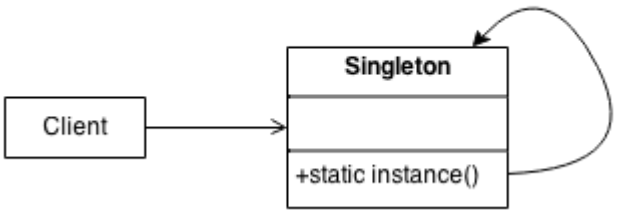
The singleton design pattern describes how to solve such problems:

- Hide the constructor of the class.
- Define a public static operation (getInstance()) that returns the sole instance of the class.

The key idea in this pattern is to make the class itself responsible for controlling its instantiation(that it is instantiated only once). The hidden constructor (declared *private*) ensures that the class can never be instantiated from outside the class. The public static operation can be accessed easily by using the class name and operation name (Singleton.getInstance()).

Singleton should be considered only if all three of the following criteria are satisfied:

- Ownership of the single instance cannot be reasonably assigned
- Lazy initialization is desirable
- Global access is not otherwise provided for



Common uses

- The abstract factory, builder, and prototype patterns can use singletons in their implementation.
- Facade objects are often singletons because only one facade object is required.
- State objects (https://en.wikipedia.org/wiki/State_pattern) are often singletons.

- Singletons are often preferred to global variables because:
    - They do not pollute the global namespace (or, in languages with namespaces, their containing namespace) with unnecessary variables.
    - They permit lazy allocation and initialization, whereas global variables in many languages will always consume resources.

2. Design pattern example

OUTPUT

```
BookBorrower1 asked to borrow the book
BookBorrower1 Author and Title:
Design Patterns by Gamma, Helm, Johnson, and Vlissides


BookBorrower2 asked to borrow the book
BookBorrower2 Author and Title:
I don't have the book


BookBorrower1 returned the book

BookBorrower2 asked to borrow the book
BookBorrower2 Author and Title:
Design Patterns by Gamma, Helm, Johnson, and Vlissides
```

CODE

```
class BookSingleton {
    private $author = 'Gamma, Helm, Johnson, and Vlissides';
    private $title  = 'Design Patterns';
    private static $book = NULL;
    private static $isLoanedOut = FALSE;

    private function __construct() {
    }

    static function borrowBook() {
      if (FALSE == self::$isLoanedOut) {
        if (NULL == self::$book) {
           self::$book = new BookSingleton();
        }
        self::$isLoanedOut = TRUE;
        return self::$book;
      } else {
        return NULL;
      }
    }

    function returnBook(BookSingleton $bookReturned) {
        self::$isLoanedOut = FALSE;
    }
```

```
    function getAuthor() {return $this->author;}

    function getTitle() {return $this->title;}

    function getAuthorAndTitle() {
      return $this->getTitle() . ' by ' . $this->getAuthor();
    }
  }

class BookBorrower {
    private $borrowedBook;
    private $haveBook = FALSE;

    function __construct() {
    }

    function getAuthorAndTitle() {
      if (TRUE == $this->haveBook) {
        return $this->borrowedBook->getAuthorAndTitle();
      } else {
        return "I don't have the book";
      }
    }

    function borrowBook() {
      $this->borrowedBook = BookSingleton::borrowBook();
      if ($this->borrowedBook == NULL) {
        $this->haveBook = FALSE;
      } else {
        $this->haveBook = TRUE;
      }
    }

    function returnBook() {
      $this->borrowedBook->returnBook($this->borrowedBook);
    }
  }

/**
 * Initialization
 */

  writeln('BEGIN TESTING SINGLETON PATTERN');
  writeln('');

  $bookBorrower1 = new BookBorrower();
  $bookBorrower2 = new BookBorrower();

  $bookBorrower1->borrowBook();
  writeln('BookBorrower1 asked to borrow the book');
  writeln('BookBorrower1 Author and Title: ');
  writeln($bookBorrower1->getAuthorAndTitle());
  writeln('');

  $bookBorrower2->borrowBook();
  writeln('BookBorrower2 asked to borrow the book');
  writeln('BookBorrower2 Author and Title: ');
  writeln($bookBorrower2->getAuthorAndTitle());
  writeln('');
```

```
$bookBorrower1->returnBook();
writeln('BookBorrower1 returned the book');
writeln('');

$bookBorrower2->borrowBook();
writeln('BookBorrower2 Author and Title: ');
writeln($bookBorrower1->getAuthorAndTitle());
writeln('');

writeln('END TESTING SINGLETON PATTERN');
```

3. Existing pattern in FEM-MAT-OO?

Not implemented.

4. Design proposal in FEM-MAT-OO

Singleton implementation for FEM-MAT-OO factories:

```
classdef IntegratorFactoryTest < Singleton
    methods (Access = public, Static)
        function integrator = create(mesh)
            switch mesh.meshType
                case 'INTERIOR'
                    integrator = Integrator_Interior;
                case 'BOUNDARY'
                    integrator = Integrator_Boundary;
                otherwise
                    error('Invalid Mesh type')
            end
        end
      % Concrete implementation.  See Singleton superclass.
        function obj = instance()
            persistent uniqueInstance
            if isempty(uniqueInstance)
                obj = IntegratorFactoryTest();
                uniqueInstance = obj;
            else
                obj = uniqueInstance;
            end
        end
    end
end
```