The Singleton Pattern

Other Considerations

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Singleton* pattern can be applied to centralize access to global resources.
- Understand the structure & functionality of the *Singleton* pattern.
- Know how to implement the *Singleton* pattern in C++.
- Be aware of other considerations when applying the *Singleton* pattern.

Singleton GoF Object Creational

Consequences

+ Helps "declutter" class & method interfaces

Singleton GoF Object Creational

Consequences

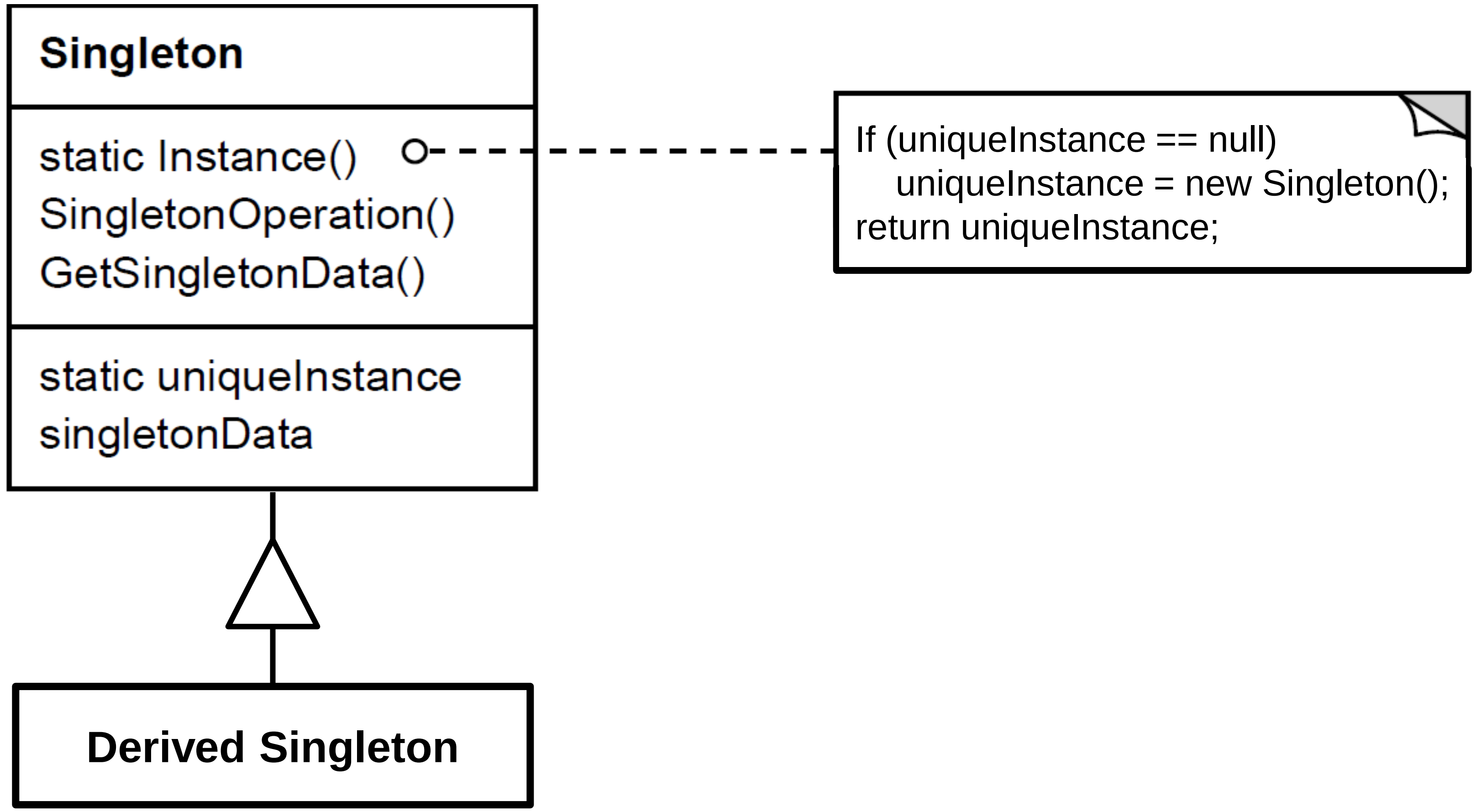
+ Reduces namespace pollution & centralizes access to global resources

See wiki.c2.com/?GlobalVariablesConsideredHarmful

Singleton GoF Object Creational

Consequences

+ Allows extension by subclassing



```
If (uniqueInstance == null)
    uniqueInstance = new Singleton();
return uniqueInstance;
```

Singleton GoF Object Creational

Consequences

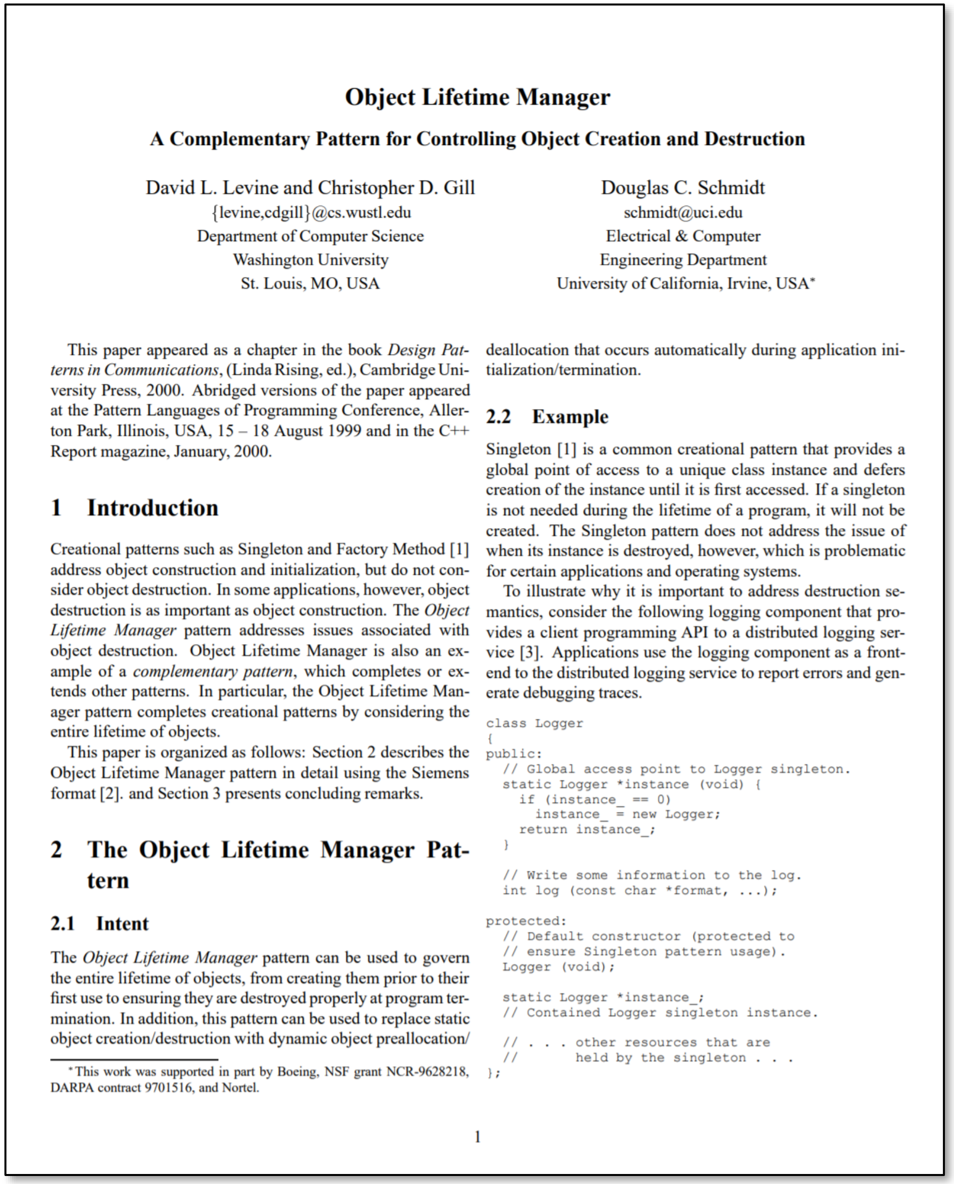
+ Only allocates resources for objects actually accessed at least once

See en.wikipedia.org/wiki/Lazy_initialization

Singleton GoF Object Creational

Consequences

+ Alleviates problems with global variables in certain programming languages



Object Lifetime Manager

A Complementary Pattern for Controlling Object Creation and Destruction

David L. Levine and Christopher D. Gill
{levine,cdgill}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO, USA

Douglas C. Schmidt
schmidt@uci.edu
Electrical & Computer
Engineering Department
University of California, Irvine, USA*

This paper appeared as a chapter in the book *Design Patterns in Communications*, (Linda Rising, ed.), Cambridge University Press, 2000. Abridged versions of the paper appeared at the Pattern Languages of Programming Conference, Allerton Park, Illinois, USA, 15 – 18 August 1999 and in the C++ Report magazine, January, 2000.

1 Introduction

Creational patterns such as Singleton and Factory Method [1] address object construction and initialization, but do not consider object destruction. In some applications, however, object destruction is as important as object construction. The *Object Lifetime Manager* pattern addresses issues associated with object destruction. Object Lifetime Manager is also an example of a *complementary pattern*, which completes or extends other patterns. In particular, the Object Lifetime Manager pattern completes creational patterns by considering the entire lifetime of objects.

This paper is organized as follows: Section 2 describes the Object Lifetime Manager pattern in detail using the Siemens format [2]. and Section 3 presents concluding remarks.

2 The Object Lifetime Manager Pattern

2.1 Intent

The *Object Lifetime Manager* pattern can be used to govern the entire lifetime of objects, from creating them prior to their first use to ensuring they are destroyed properly at program termination. In addition, this pattern can be used to replace static object creation/destruction with dynamic object preallocation/deallocation that occurs automatically during application initialization/termination.

2.2 Example

Singleton [1] is a common creational pattern that provides a global point of access to a unique class instance and defers creation of the instance until it is first accessed. If a singleton is not needed during the lifetime of a program, it will not be created. The Singleton pattern does not address the issue of when its instance is destroyed, however, which is problematic for certain applications and operating systems.

To illustrate why it is important to address destruction semantics, consider the following logging component that provides a client programming API to a distributed logging service [3]. Applications use the logging component as a front-end to the distributed logging service to report errors and generate debugging traces.

```
class Logger
{
public:
  // Global access point to Logger singleton.
  static Logger *instance (void) {
    if (instance_ == 0)
      instance_ = new Logger;
    return instance_;
  }

  // Write some information to the log.
  int log (const char *format, ...);

protected:
  // Default constructor (protected to
  // ensure Singleton pattern usage).
  Logger (void);

  static Logger *instance_;
  // Contained Logger singleton instance.

  // . . . other resources that are
  //       held by the singleton . . .
};
```

*This work was supported in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, and Nortel.

1

www.dre.vanderbilt.edu/~schmidt/PDF/ObjMan.pdf: Singleton management

Singleton GoF Object Creational

Consequences

– Does not address all the liabilities with global variables
  - In particular, increased implicit dependencies & reduced program clarity

Singleton GoF Object Creational

Consequences

– Implementation may be less efficient than a global variable
  - Due to additional indirection & synchronization overhead

Singleton GoF Object Creational

Consequences

– Subtle concurrency & dynamic loading traps & pitfalls

Consequences

– c2.com/cgi/wiki?SingletonsAreEvil summarizes *Singleton* drawbacks

Singleton GoF Object Creational

Implementation considerations

- Determine if you really must use Singleton!
  - www.ibm.com/developerworks/webservices/library/co-single has good tips:
    - Will every application use this class exactly the same way? (*exactly* is the key word)
    - Will every application ever need only one instance of this class? (*ever* & *one* are the key words)
    - Should the clients of this class be unaware of the application they are part of?

It's often possible (& desirable) to avoid using *Singleton* in your programs.

Singleton

Implementation considerations

- Defining static instance method & data

```
class Singleton {
  private static Singleton sInst = null;
  public static Singleton instance() {
    Singleton result = sInst;
    if (result == null) {
      sInst = result = new Singleton();
    }
    return result;
  }
  ...
```

Singleton GoF Object Creational

Implementation considerations

- Avoiding concurrency hazards

```
class Singleton {
  private static Singleton sInst = null;
  public static Singleton instance() {
    Singleton result = sInst;
    if (result == null) {
      sInst = result = new Singleton();
    }
    return result;
  }
  ...
```

Too little synchronization

Singleton GoF Object Creational

Implementation considerations

- Avoiding concurrency hazards

Too much synchronization

```
class Singleton {
  private static Singleton sInst = null;
  public static Singleton instance() {
    synchronized(Singleton.class) {
      Singleton result = sInst;
        if (result == null) {
          sInst = result =
            new Singleton();
      }
      return result;
    }
  }
  ...
```

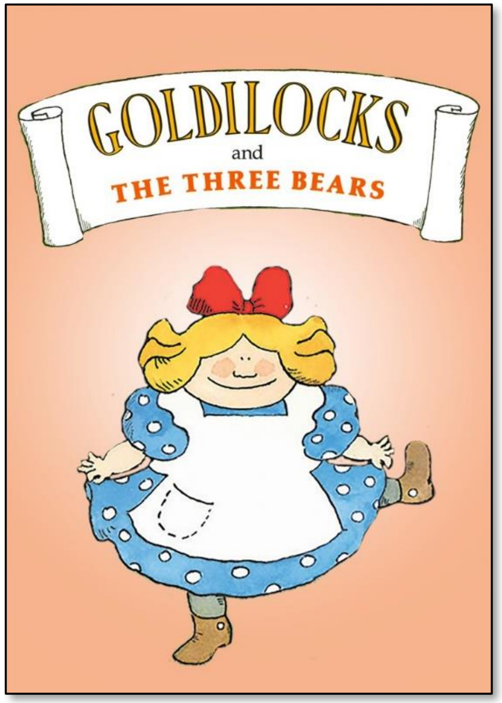
Singleton GoF Object Creational

Implementation considerations

- Avoiding concurrency hazards

```
class Singleton {
  private static volatile Singleton
                              sInst = null;
  public static Singleton instance() {
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null) {
          sInst = result =
            new Singleton();
        }
      }
    }
    return result;
  }
  ...
```

Just the right amount of synchronization



Implementation considerations

- Avoiding concurrency hazards

```
class Singleton {
  private static volatile Singleton
                                   sInst = null;
  public static Singleton instance() {
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null) {
          sInst = result =
            new Singleton();
        }
      }
    }
    return result;
  }
  ...
```

Only synchronize when sInst is null

Implementation considerations

- Avoiding concurrency hazards

```
class Singleton {
  private static volatile Singleton
                                 sInst = null;
  public static Singleton instance() {
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null) {
          sInst = result =
            new Singleton();
        }
      }
    }
    return result;
  }
  ...
```

No synchronization after sInst is created

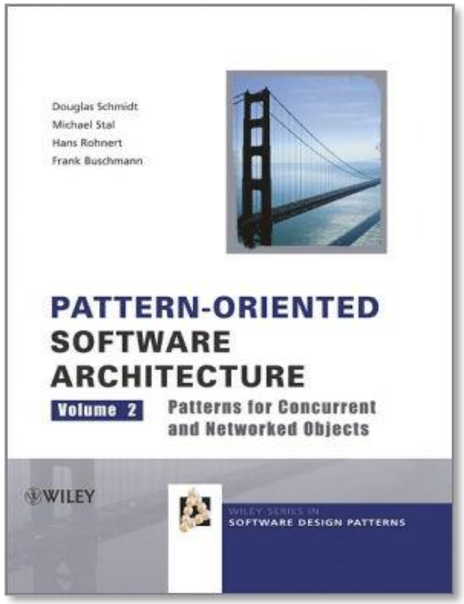
Singleton GoF Object Creational

Implementation considerations

- Avoiding concurrency hazards

This solution only works in JDK5 & above.

```
class Singleton {
  private static volatile Singleton
                                                 sInst = null;
  public static Singleton instance() {
    Singleton result = sInst;
    if (result == null) {
      synchronized(Singleton.class) {
        result = sInst;
        if (result == null) {
          sInst = result =
            new Singleton();
        }
      }
    }
    return result;
  }
  ...
```


Douglas Schmidt
Michael Stal
Hans Rohnert
Frank Buschmann
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
Volume 2
Patterns for Concurrent and Networked Objects
WILEY
WILEY SERIES IN SOFTWARE DESIGN PATTERNS

See en.wikipedia.org/wiki/Double-checked_locking for more information.

Singleton GoF Object Creational

Implementation considerations

- Avoiding concurrency hazards

```
class Singleton {
   /** Private constructor */
   private Singleton() { }

   /**
    * SingletonHolder's loaded on first
    * execution of Singleton.instance()
    * or first access to SingletonHolder.
    * INSTANCE, not before
    */
    private static class SingletonHolder {
      public static final Singleton
        instance_ = new Singleton();
    }

    /** Returns single instance */
    public static Singleton instance()
    { return SingletonHolder.instance_; }
}
```

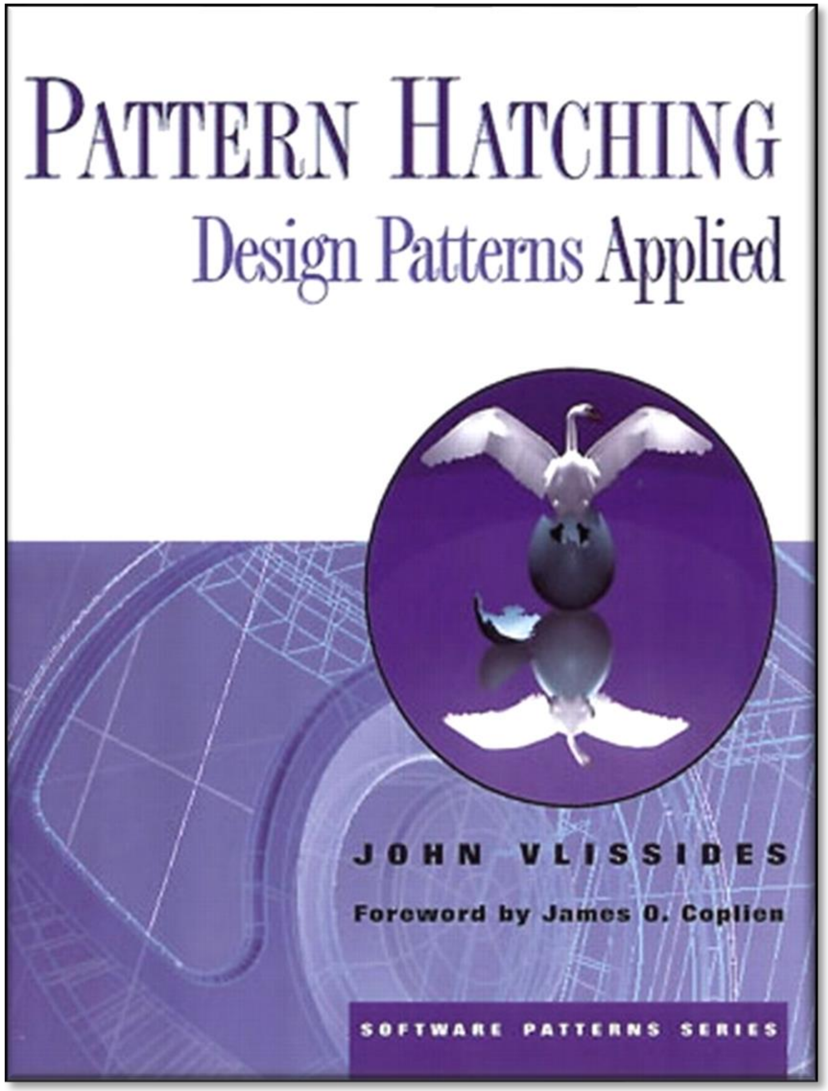
This solution works in all JDKs!

en.wikipedia.org/wiki/Singleton_pattern#The_solution_of_Bill_Pugh has more.

Singleton GoF Object Creational

Implementation considerations

- Deleting singletons



PATTERN HATCHING

Design Patterns Applied

JOHN VLISSIDES

Foreword by James O. Coplien

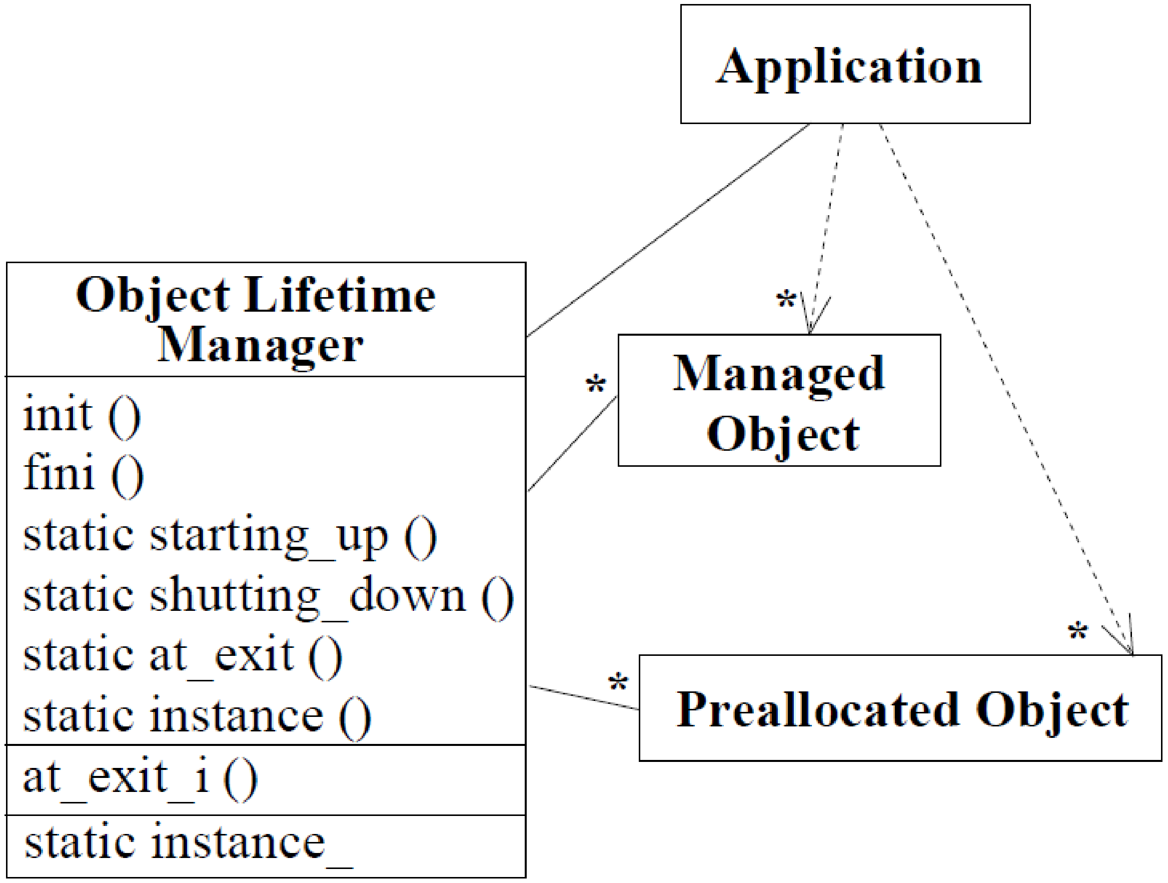
SOFTWARE PATTERNS SERIES

"To Kill a Singleton" sourcemaking.com/design_patterns/to_kill_a_singleton

Singleton GoF Object Creational

Implementation considerations

- Registering the singleton instance with manager



www.dre.vanderbilt.edu/~schmidt/PDF/ObjMan.pdf: Singleton management

Singleton GoF Object Creational

Known uses

- Unidraw's Unidraw object
- Smalltalk-80 ChangeSet, the set of changes to code
- InterViews Session object
- ACE Singleton

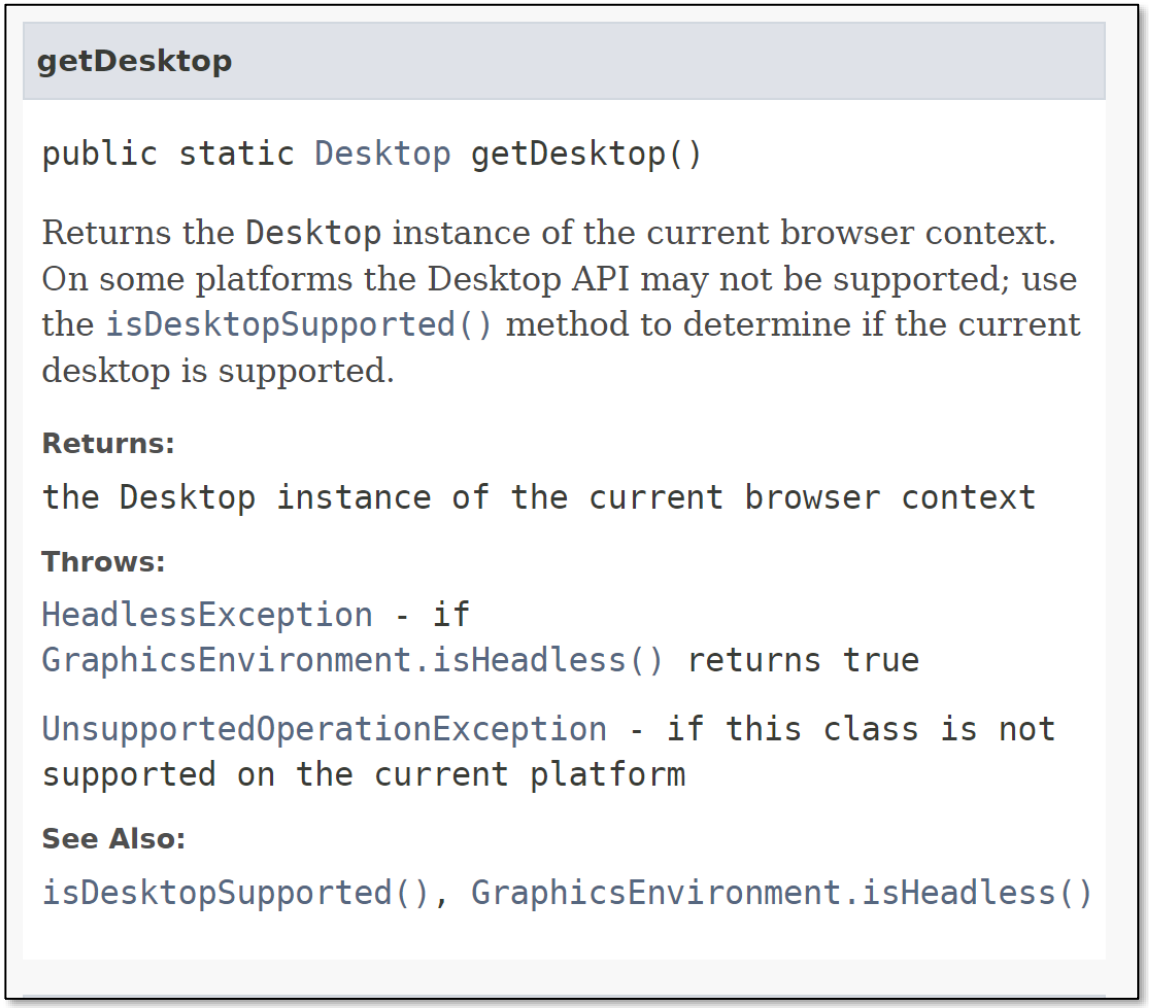
```
template <typename TYPE>
TYPE *ACE_Singleton<TYPE>::instance () {
  TYPE *tmp = instance_;
#if defined (ALPHA_MP)
  // Insert CPU-specific memory barrier
  // instruction to synchronize cache lines.
  asm ("mb");
#endif /* ALPHA_MP */
  // First check
  if (tmp == 0) {
    ACE_Guard<ACE_Thread_Mutex> guard (lock_);
    tmp = instance_; // Reload tmp.
    // Double check.
    if (tmp == 0) {
      tmp = new TYPE;
#if defined (ALPHA_MP)
      // Insert a second CPU-specific memory
      // barrier instruction.
      asm ("mb");
#endif /* ALPHA_MP */
      instance_ = tmp;
    }
  }
  return tmp;
}
```

en.wikipedia.org/wiki/Double-checked_locking has more synchronization information.

Singleton GoF Object Creational

Known uses

- Unidraw's Unidraw object
- Smalltalk-80 ChangeSet, the set of changes to code
- InterViews Session object
- ACE Singleton
- The Java AWT Desktop getDesktop() method



getDesktop

```
public static Desktop getDesktop()
```

Returns the Desktop instance of the current browser context. On some platforms the Desktop API may not be supported; use the isDesktopSupported() method to determine if the current desktop is supported.

Returns:

the Desktop instance of the current browser context

Throws:

HeadlessException - if GraphicsEnvironment.isHeadless() returns true

UnsupportedOperationException - if this class is not supported on the current platform

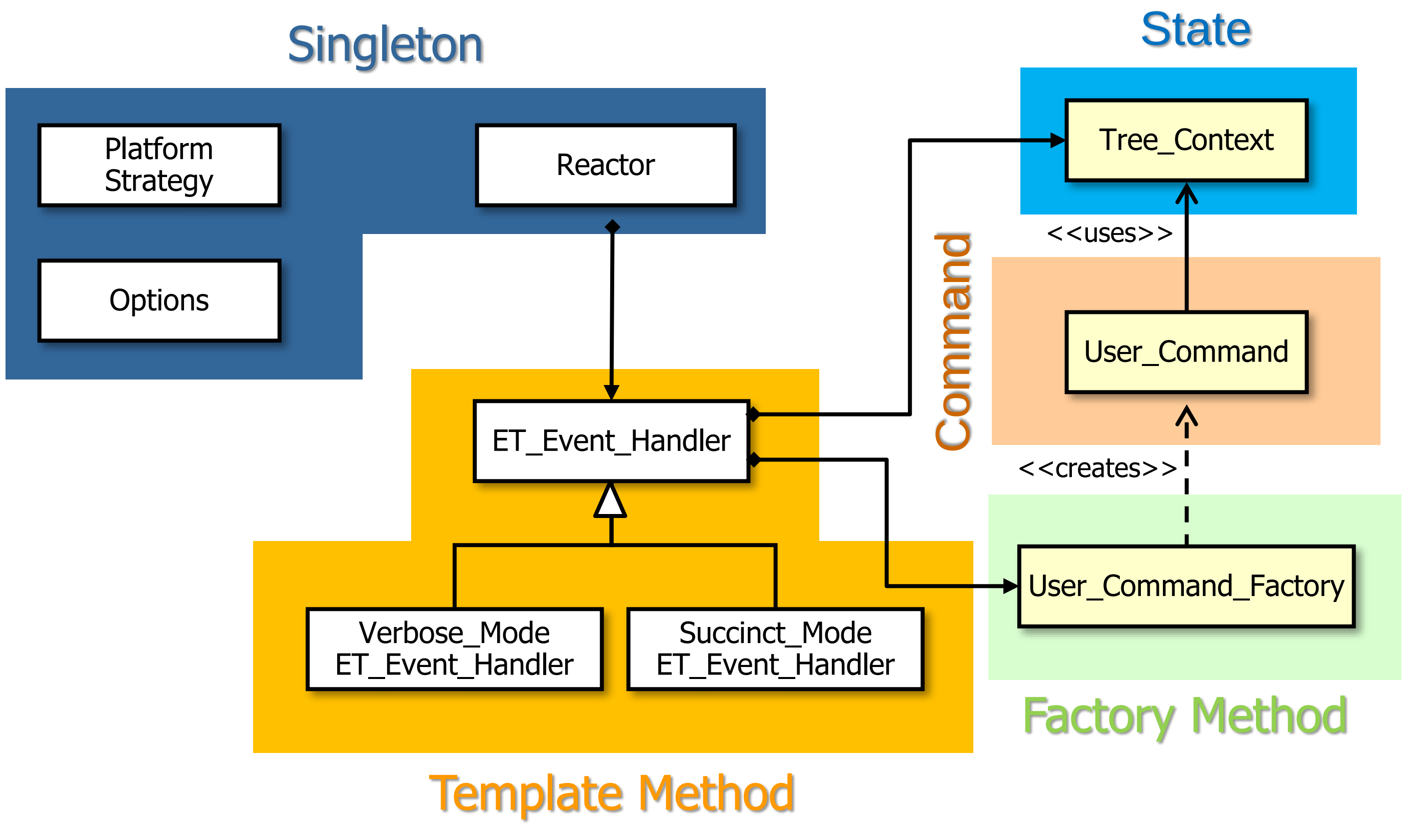
See Also:

isDesktopSupported(), GraphicsEnvironment.isHeadless()

See docs.oracle.com/javase/8/docs/api/java/awt/Desktop.html#getDesktop

Summary of the Singleton Pattern

- *Singleton* simplifies access to global resources in the expression tree processing app.



Singleton is the "go-to" of patterns, so apply it with care.