Design Patterns & Software Architecture
Singleton

dr. Joost Schalken-Pinkster

Windesheim University of Applied Science

The Netherlands

The contents of these course slides is (in great part) based on:
Chris Loftus, *Course on Design Patterns & Software Architecture for NEU*. Aberystwyth University, 2013.
Jeroen Weber & Christian Köppe, *Course on Patterns and Frameworks*. Hogeschool Utrecht, 2013.
Leo Pruijt, *Course on Software Architecture*. Hogeschool Utrecht, 2010-2013.

Session overview

- The Singleton

Singleton design pattern

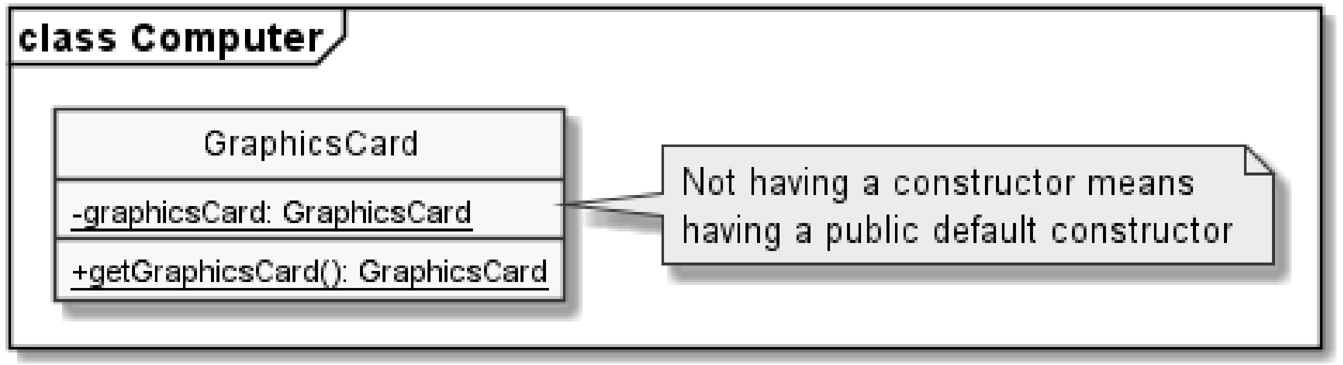
Let’s find a design pattern

Will now present, on the board, a problem that we wish to solve and that will lead to the application of a design pattern.

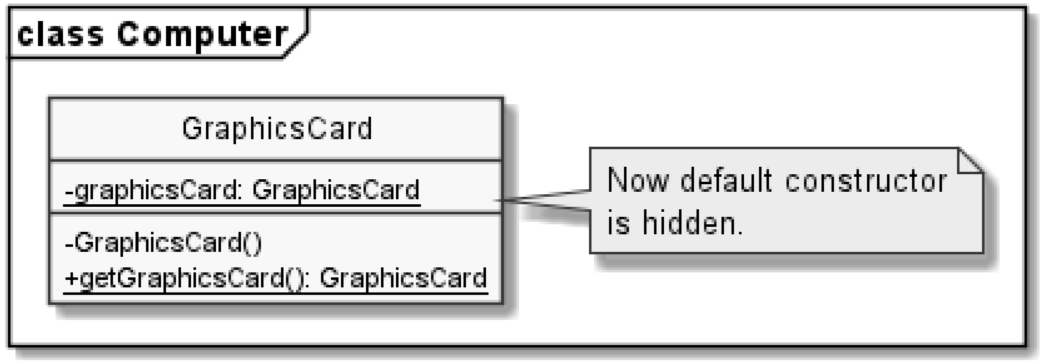
Case: Graphics adapter Requirements

Please create a design for a GraphicsAdapter class, in such a way that we can only have one GraphicsAdapter object.

Case: Graphics adapter
Design 1: Mistake default contructor


class Computer
GraphicsCard
-graphicsCard: GraphicsCard
+getGraphicsCard(): GraphicsCard
Not having a constructor means having a public default constructor

Case: Graphics adapter Design 2


class Computer
GraphicsCard
-graphicsCard: GraphicsCard
-GraphicsCard()
+getGraphicsCard(): GraphicsCard
Now default constructor is hidden.

Case: Graphics adapter Implementation (not thread safe)

```java
public class GraphicsCard {
    private static GraphicsCard graphicsCard;

    private GraphicsCard() {
    }

    public static GraphicsCard getGraphicsCard() {
        if (graphicsCard == null) {
            graphicsCard = new GraphicsCard();
        }
        return graphicsCard;
    }
}
```

Case: Graphics adapter Implementation (thread safe)

```java
public class GraphicsCard {
    private volatile static GraphicsCard graphicsCard;

    private GraphicsCard() {
    }

    public static GraphicsCard getGraphicsCard() {
        if (graphicsCard == null) {
            synchronized (GraphicsCard.class) {
                if (graphicsCard == null) {
                    graphicsCard = new GraphicsCard();
                }
            }
        }
        return graphicsCard;
    }
}
```

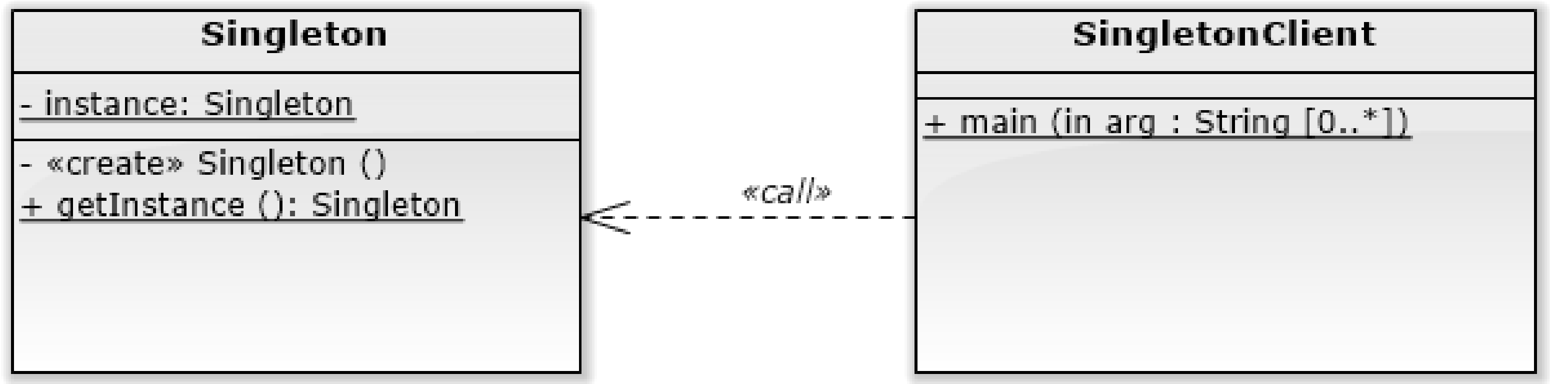
Example: The Singleton Pattern

- **Intent**: Ensure a class only ever has one object instance and provides a global point of access

- **Motivation**: In some situations there should only be one instance of a class to ensure that all objects access the same object. For example, for playing audio clips you want at maximum one audio clip manager object. Therefore, make the class responsible for creating the instance and keep track of its sole instance. 'User' code must then access class for instance.

- **Applicability**: Applicable where exactly one instance required that is accessible from across the application.

- **Structure:**



Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 145

- **Participants**: Singleton class
  - Defines a getInstance operation that lets clients access its unique instance. getInstance is a class operation.
- **Collaborations**: Clients access a Singleton instance solely through getInstance operation

- **Consequences:**
  - Benefits:
    - Controlled access to sole instance…
    - Reduced name space…
    - Sub-classing is possible with some refinement…
    - Permits a variable number of instances…

- **Consequences in detail**
  1. *Controlled access to sole instance.* Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
  2. *Reduced name space.* The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
  3. *Permits refinement of operations and representation.* The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.
  4. *Permits a variable number of instances.* The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.

Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 145-146

- **Implementation**:
  Need to ensure a unique instance. What are the issues?
  - Need a method that obtains that instance for Client object
  - Need to restrict access to creating instances; e.g. make the constructor private, package private or protected
  - Might need to make access to instance thread safe

Sample code (thread safe)

```
public class MethodStats {
    private static volatile MethodStats instance;
    private java.util.HashMap<String, Statistic> stats = new HashMap<String, Statistic>();

    private MethodStats(){}

    public static MethodStats getInstance(){
        if (null == instance) {
            synchronized(MethodStats.class) {
                if (null == instance) {
                    instance = new MethodStats();
                }
            }
        }
        return instance;
    }
}
```

Singleton Pattern (alternative implementations)

- There are alternative ways of implementing a Singleton in Java
- For example:
  - Using an enum…
  - Making the instance final and creating it immediately…
- As a paper exercise, do this now, that is convert the example on the previous slide

Additional information (not required) Inheritance and Singleton

- The sub-class can share state with the base class. Now you have two objects that share state.
- "Gang of Four" recommends usages of a registry of Singletons. The base class reads an environment variable to determine which Singleton to use.
- Bottom Line: Singletons and Inheritance do not mix well.

Additional information (not required) SingletonRegistry Class

Describes a SingletonRegistry whose purpose is to store Singletons!

```
public static Singleton getInstance() {
        return (Singleton)SingletonRegistry.REGISTRY
                        .getInstance(classname);
}
```

Source

- http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html?page=6

Issues with Singletons: (not required) Dependencies: Case Searcher

```
public class Searcher {
  //Singleton Setup code Here

  public Collection FindCustomers(String criteria) {
    String conStr = GetConnectionFromRegistry();
    OracleConnection conn = new OracleConnection(conStr);

    //Query database using criteria

   return (Collection of customers);
  }
}
```

Issues with Singletons: (not required) Dependencies: Case Searcher

To search the database, the client code would need to do the following:

```
Searcher.getInstance().FindCustomers("some criteria")
```

- What happens if we need to change the database to SQL Server?
- Or change how we get the connection string?

Issues with Singletons: (not required) Dependencies: Case Searcher

Conclusions

- The Singleton is tightly coupled to Oracle. If the database changed in the future, this object would need to be changed. It is also tightly coupled in how it retrieves the connection string.
- The Singleton hides object dependencies (Oracle and registry). Anyone using the Singleton would need to inspect the internals to find out what is really going on.
- Possibly memory leak since the Singleton may hold onto the resource for an infinite amount of time.

Issues with Singletons: (not required) Unit Testing

There are many problems with unit testing a Singleton.

- Problem: If you are running multiple unit tests that accesses the same Singleton, the Singleton will retain state between unit tests. This may lead to undesired results!
- Solution: Use Reflection to get access to the private static instance variable. Then set it to null. This should be done at the end of each unit test.

```
private static Singleton uniqueInstance;
```

Issues with Singletons: (not required)
Unit Testing

Other problems:

- You want to unit test a method of a Singleton, but the method refers to other external resources. It is hard to inject a mock in this case.
- You are unit testing a method that does a lot of business logic. One of the method calls is a Singleton that accesses an external resource. How can you replace the Singleton with a mock call?

Issues with Singletons: (not required)
Scott Densmore "Why Singletons are Evil"

- "…the dependencies in your design are hidden inside the code, and not visible by examining the interfaces of your classes and methods. You have to inspect the code to understand exactly what other objects your class uses."

- "Singletons allow you to limit creation of your objects... you are mixing two different responsibilities into the same class."

- "Singletons promote tight coupling between classes."

- "Singletons carry state with them that last as long as the program lasts…Persistent state is the enemy of unit testing."

Scott Densmore (2004). *Why Singletons are Evil*. Microsoft 2004.
http://blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx

Issues with Singletons: (not required) Jeremy Miller "Chill out on the Singleton…"

- "Using a stateful singleton opens yourself up to all kinds of threading issues. When you screw up threading safety you can create really wacky bugs that are devilishly hard to reproduce."

- "My best advice is to not use caching via static members until you absolutely have to for performance or resource limitation reasons."

Jeremy D. Miller (2005), Chill out on the Singleton…, CodeBetter, 2005. http://codebetter.com/blogs/jeremy.miller/archive/2005/08/04/130302.aspx

Issues with Singletons: (not required) How to Decontaminate a Singleton

- "Define a Spring bean for each Singleton you use. For new DI-like implementations use the Spring bean as a wrapper that allows to access the functionality of the Singleton."

Rainer Eschen (2006), How to Decontaminate a Singleton. http://blog.rainer.eschen.name/2006/12/15/how-to-decontaminate-a-singleton/

Issues with Singletons: (not required) Comments from Others

- “Sub-classing or creating mocks for Singletons is impossible (unless it implements an interface for its type)”
- “It really should only be used in cases of classes for Constants for example.”
- “Access to a singleton in an application must be serialized, which complicates multi-threading issues in applications hence introducing possible issues of thread locking.”

Jon Simon (2010), Design Patterns: *Week 1: Introduction and Singleton*.
www.slideshare.net/jonsimon2/ & jonathan_simon@yahoo.com

Issues with Singletons: (not required) Comments from Others

- "I had cases where I wanted to create a subclass, and the singleton kept referring to the superclass."
- "..writing a good unit test was impossible because the class invoked a singleton, and there was no way to inject a fake."
- "Unit testing a singleton is not so much a problem as unit testing the other classes that use a singleton. They are bound so tightly that there is often no way to inject a fake (or mock) version of the singleton class. In some cases you can live with that, but in other cases, such as when the singleton accesses an external resource like a database, it's intolerable for a unit test."

Jon Simon (2010), Design Patterns: *Week 1: Introduction and Singleton*.
www.slideshare.net/jonsimon2/ & jonathan_simon@yahoo.com

Other useful articles *(not required)*

Singleton Explained

- http://c2.com/cgi/wiki?SingletonPattern
- http://www.oodesign.com/singleton-pattern.html

Unit Testing

- http://imistaken.blogspot.com/2009/11/unit-testing-singletons.html
- http://weblogs.asp.net/okloeten/archive/2004/08/19/217182.aspx

Reading

For this lesson please read:

- Chapter 5 (One of a kind Objects) of Head First Design Patterns