

Dr. Michael Eichberg

Software Engineering

Department of Computer Science

Technische Universität Darmstadt

Software Engineering

Enforcing Singularity

For details see Gamma et al. in "Design Patterns"



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Enforcing Singularity

Example / Motivation

- In some cases a mechanism is required **to enforce singularity of objects**; i.e. it is necessary to enforce that there exists at most one instance of a class at runtime.

For example, ...

- in a system there should be only one printer spooler,
- there should be only one class to handle interactions with the database.
- *Two patterns for enforcing Singularity:*
 - *Singleton*
 - *Monostate*

Enforcing Singularity

Example / Motivation

- In some cases a mechanism is required **to enforce singularity of objects**; i.e. it is necessary to enforce that there exists at most one instance of a class at runtime.

For example:

- in a system
 - there is a database
 - *Two patterns*
 - *Singleton*
 - *Monostate*
- th the

Beware: Often it is best to just create one instance of an object (using the constructor) at program initialization time and to use this object.

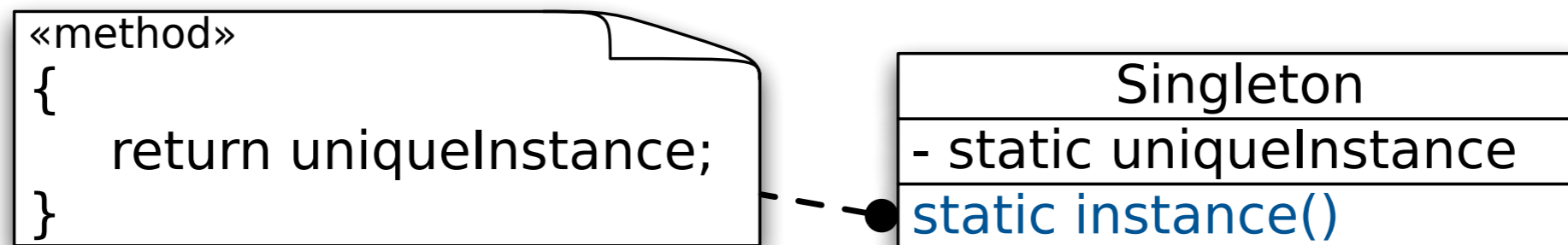
The Singleton Design Pattern

Intent and Structure

Intent

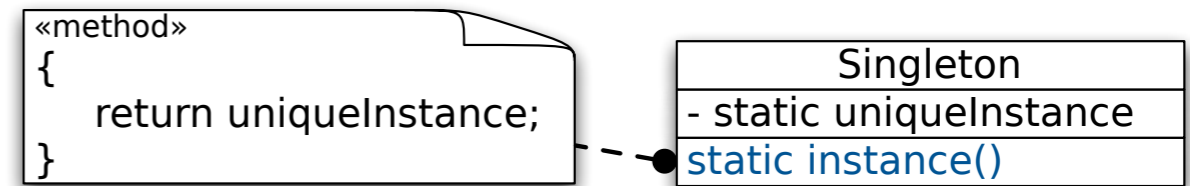
Ensure a class only has one instance, and provide a global point of access to it.

Structure



The Singleton Design Pattern

Implementation



```
public class Singleton {
```

```
    private static Singleton theInstance = null;
```

```
    private Singleton();
```

The constructor "should" be private or protected.

```
    public static Singleton instance() {
```

```
        if (theInstance == null)
```

```
            theInstance = new Singleton();
```

```
        return theInstance;
```

```
    }
```

```
}
```

The implementation is not thread safe.
(Recall the discussion of the double-checked locking idiom.)

The Singleton Design Pattern

Benefits

- **Cross platform**

Using appropriate middleware, Singleton can be extended to work across many JVMs.

- **Applicable to any class**

- Can be created through derivation
Given a class, you can create a subclass that is a Singleton.

Costs

- **Lazy creation**

(Controlled access to sole instance.)

If the singleton is never used, it is never created.

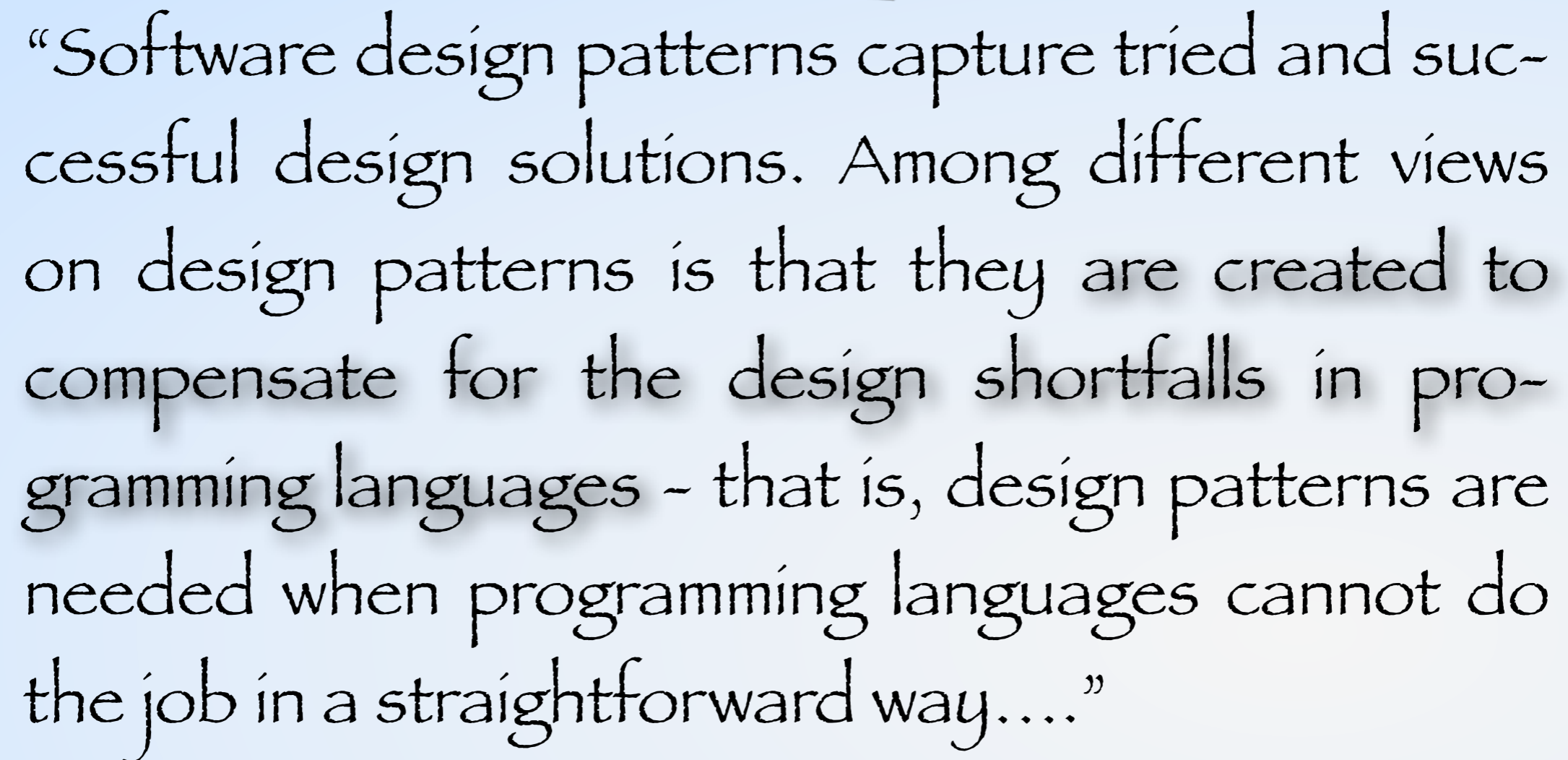
- **Destruction is undefined**

- **Not inherited**

A class derived from a singleton is not a singleton.

- **Nontransparent**

Users of a Singleton know that they are using a Singleton.



“Software design patterns capture tried and successful design solutions. Among different views on design patterns is that they are created to compensate for the design shortfalls in programming languages - that is, design patterns are needed when programming languages cannot do the job in a straightforward way....”

Liping Zhao; **Patterns, Symmetry, and Symmetry Breaking**;
Communications of the ACM, March 2008, Vol. 51, No. 3

Patterns as a motivation for language features.

Singleton objects in the Scala Programming Language

(<http://www.scala-lang.org/>)

“The same program” implemented in two programming languages.

```
1 object Hello {  
2     // methods  
3 }
```

```
1 public class Hello {  
2  
3     private static Hello hello;  
4  
5     private Hello(){ }  
6  
7     public static synchronized Hello instance() {  
8         if (hello == null) {  
9             hello = new Hello();  
10        }  
11        return hello;  
12    }  
13  
14    // methods  
15 }
```



Java

The Monostate Design Pattern

Intent and Implementation

Intent

Make all objects of the same type behave as though they were a single object

Implementation

- ▶ Make all fields static
- ▶ Methods are not static

```
public class X {  
    public X(){...}  
  
    private static boolean x = ...;  
  
    public boolean isX() { return x; }  
}
```

The Monostate Design Pattern

Benefits

- **Transparent**
The user does not need to know that the object is Monostate.
Destruction is well-defined.
- **Derivability**
Derivatives of a Monostate are Monostates; derivatives of a Monostate are part of the same Monostate.
- **Polymorphism**
Since the methods of a Monostate are not static, they can be overridden in a derivative. Derivatives can offer different behavior over the same set of static variables.

Costs

- **No conversion**
A normal class cannot be converted into a Monostate.
- **Efficiency**
A Monostate may go through many creations and destructions because it is a real object.
- **Presence**
The variables of a Monostate take up space, even if the Monostate is never used.
- **Platform local**
A Monostate cannot work across several JVM instances or across several platforms.

Singleton vs. Monostate Design Pattern

Singleton is best used when you have an *existing class that you want to constrain through der-ivation*, and you don't mind that everyone will have to call the `instance()` method to gain ac-cess.

Monostate is best used when you want the singular nature of the class to be transparent to the users, or when you want to *employ polymorphic derivatives of the single object*.

Singleton is about
structure!

Monostate is about
behavior!