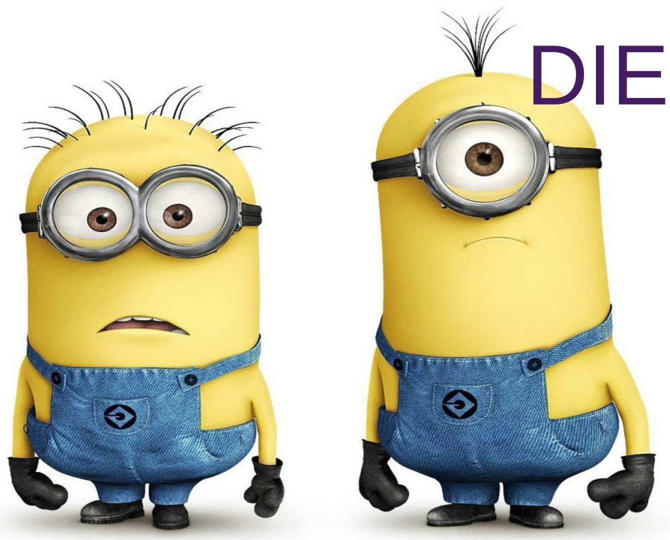
Must know Design Patterns

to know before you

DIE as a Software Engineer !!



Pankaj Gaur
Abhishek Sharma

Speakers

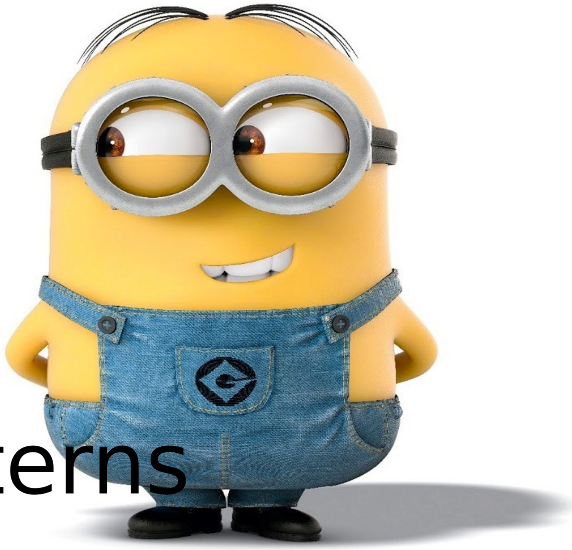
Sr. Principle Consultant Java Consultant (SCWCD, SCJP, OCA)

Agenda

Software Design and Architecture

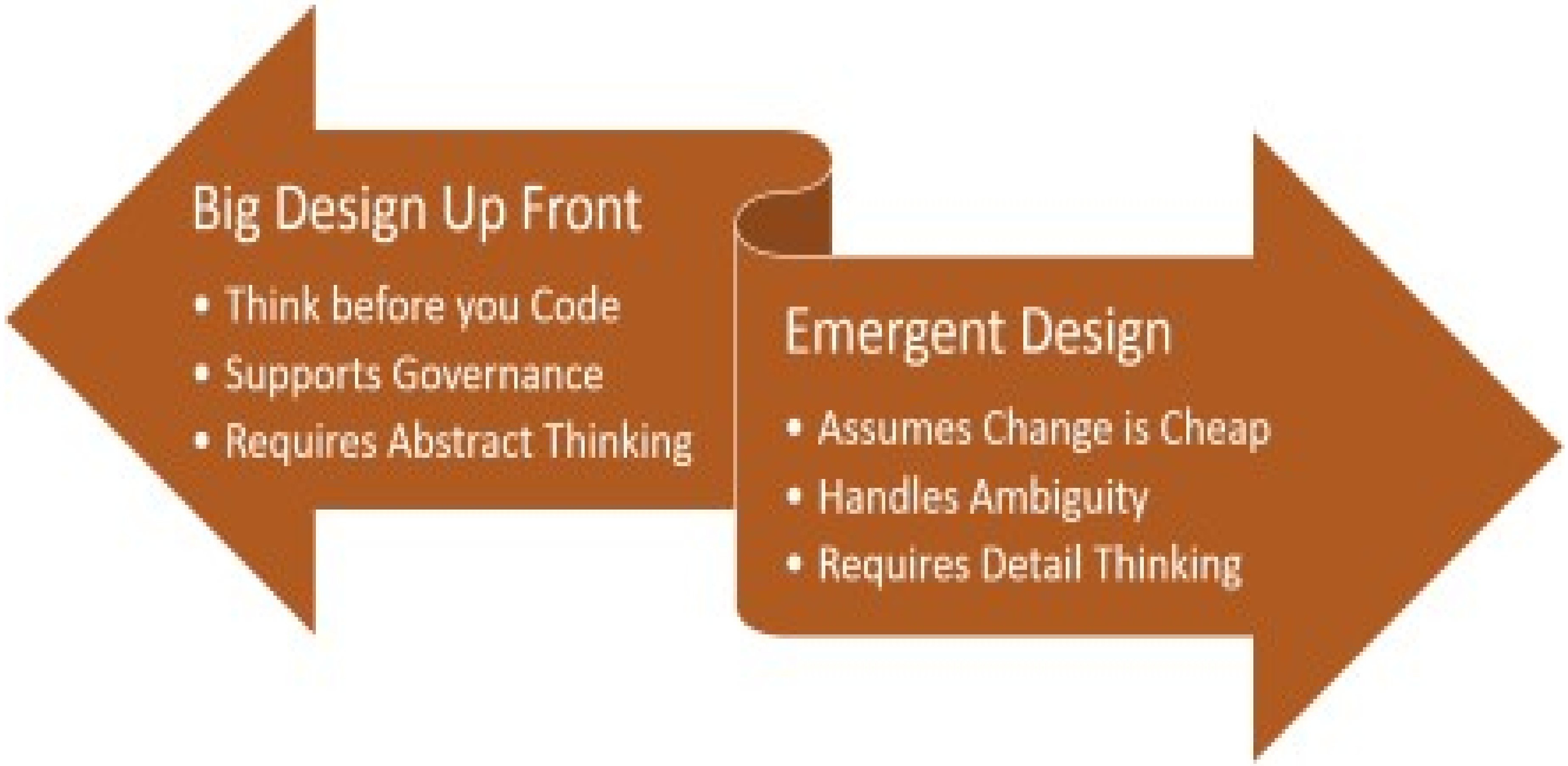
SOLID software !!

Must know Design Patterns

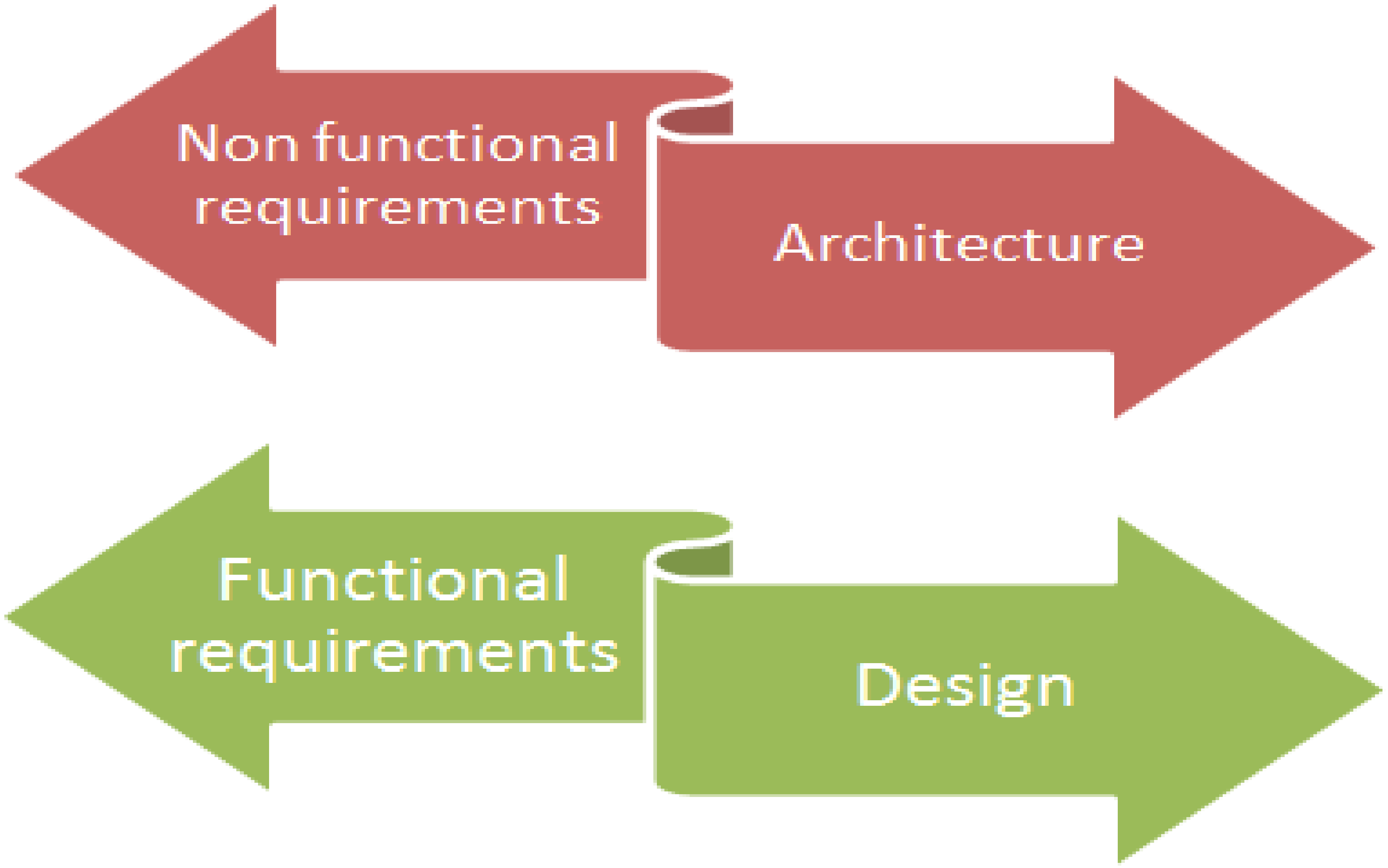


oftware Design and Architecture

Introduction to Software Design and Architecture



Introduction to Software Design and Architecture



Ewww... Bad code

When an application becomes hard to maintain. The signs which indicate a future code rot

Rigidity – small changes causes the entire system to rebuild.

Fragility – changes to one module causes other unrelated modules to misbehave. Imagine a car system in which changing the radio station affects windows.

Immobility – a module’s internal components cannot be extracted and reused in new environments. For example, if an application’s login module cannot be used in entirely different system then this module is immobile, caused by couplings and dependencies between different modules. The strategy is to decouple central abstractions from low-level details, like a particular database schema or UI implementation (web, desktop) or specific frameworks.

Viscosity – when building and testing are difficult to perform and take a long time to execute. When even a simple change is costly to make, and requires you to make

Good code :)

Code makes an application work for users. Getting tests passing or removing duplication is useful, but the code needs to be shipped so that users can use it.

What kind of code is good?

Code **needs to work**. Assuming that as a prerequisite…

When code is easy to understand, we can change or extend it faster and are less likely to make a mistake in the process. **Readability** counts.

When code is **easy to change**, we can iterate on user feedback faster.

When code is fun to work with, we're happier. Building software is our full time job. If we go home every day feeling wrung dry, something is wrong with our process

"Walking on water and developing software from a specification are easy if both are frozen."

- Edward V. Berard

SOLID software !!

SOLID – by the power of Gray Skull

Xebia

SOLID for a better Software

SOLID are five basic principles which help to create good software architecture. SOLID is an acronym where:-

S stands for SRP (Single responsibility principle)

O stands for OCP (Open closed principle)

L stands for LSP (Liskov substitution principle)

I stands for ISP (Interface segregation principle)

D stands for DIP (Dependency inversion principle)

Single responsibility principle

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

Single responsibility principle

The Single Responsibility Principle (SRP) states that there should never be more than one reason for a class to change. This means that every class, or similar structure, in your code should have only one job to do.

Everything in the class should be related to that single purpose, i.e. be cohesive. It does not mean that your classes should only contain one method or property.

Cohesion and Coupling

Coupling or dependency is the degree to which each program module relies on each one of the other modules.

Cohesion refers to the degree to which the elements of a module belong together. Thus, it is a measure of how strongly related each piece of functionality expressed by the source code of a software module is.

A system with a higher degree of coupling and lower

shaadi ke side/effects
DIRECTED BY SAKET CHAUDHARY
PRODUCED BY SHOBHA KAPOOR EKTA KAPOOR PRODUCED BY PRITISH NANDY RANGITA PRITISH NANDY

Single responsibility principle - exercise

The **Email** example

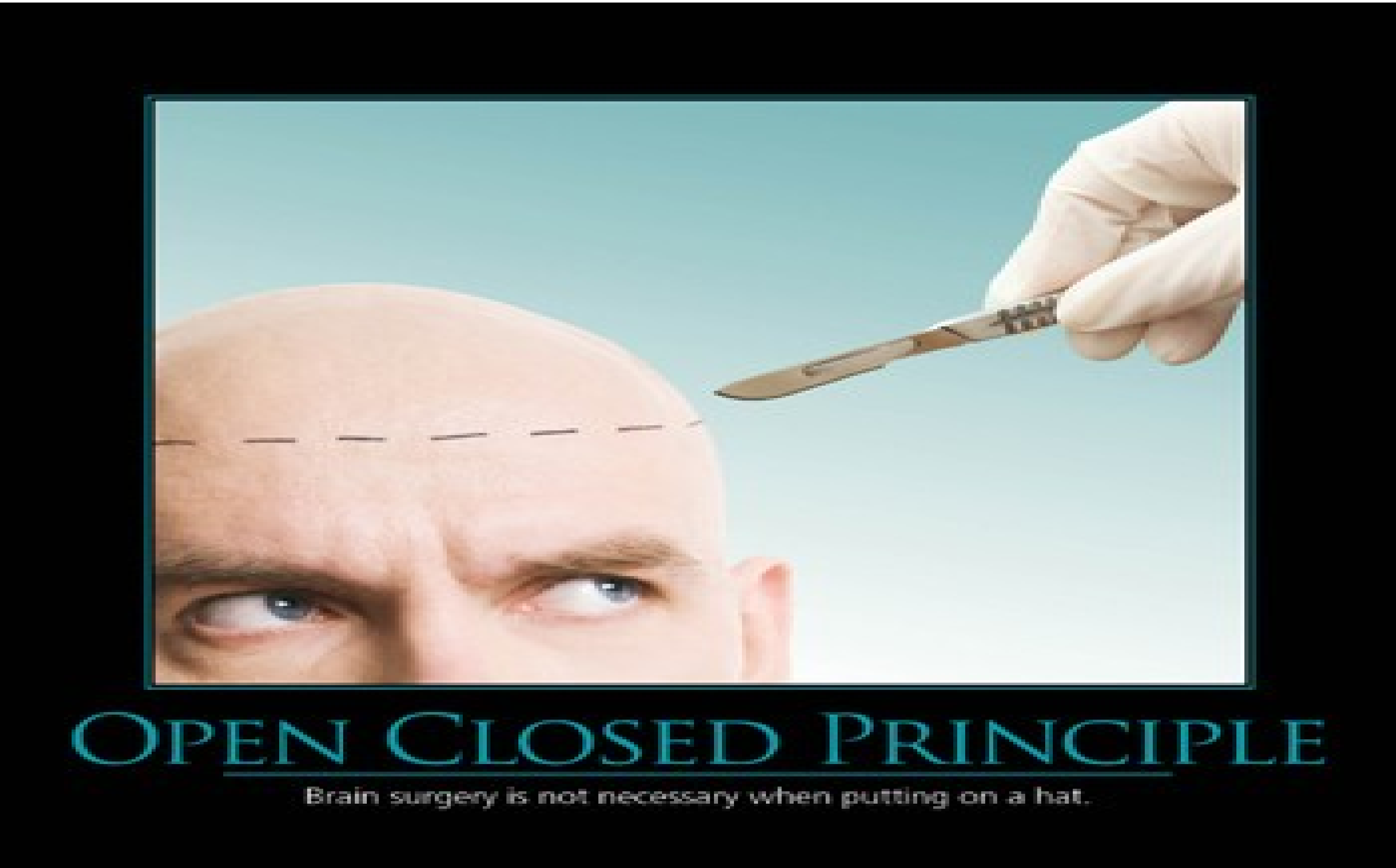
```
interface Email{
    void send(MailSender sender, String content);
}
```

Add these functionality:

- –save as draft in database,
- –print to the nearest printer,
- –display on browser/screen,
- –send HTML,
- –Send XML data,
- –Send attachments

“A violation”

Open closed principle



Open closed principle

Software entities (classes, modules, functions, etc.) should be

open for extension,

but

closed for modification.

Liskov substitution principle

If it looks like an apple but does not taste like one

You have probably the wrong abstraction!

Liskov substitution principle - excercise

Design a OO class hierarchy for the Flying Birds

Sample Impl classes:

Parrot

Sparrow

Kingfisher

Ostrich

Liskov substitution principle

The Liskov substitution principle is the L in SOLID principle and it states that subtypes must be substitutable for their base types. An LSP can be achieved by :

- –Child classes must not remove the behavior of base classes
- –Child class must not violate base class invariants

Interface segregation principle

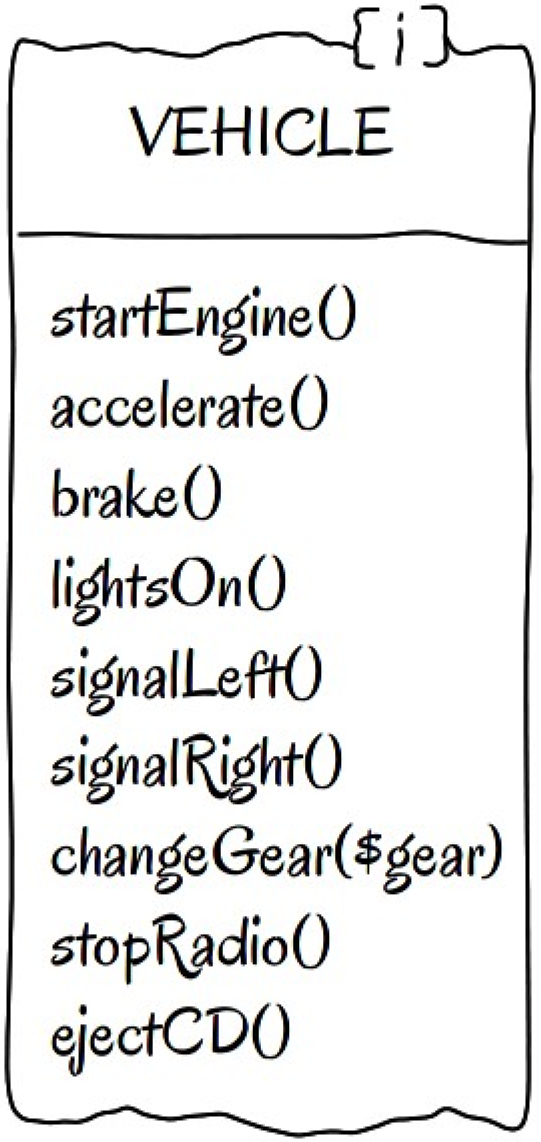
Interface Segregation Principle
If IRequireFood, I want to Eat(Food food) not,
LightCandelabra() or LayoutCutlery(CutleryLayout preferredLayout)

Interface segregation principle

The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.

Examples: The complexity of buying a TV

The Flying Bird example revisited



Xebia

Dependency inversion principle

Resolving **dependencies**

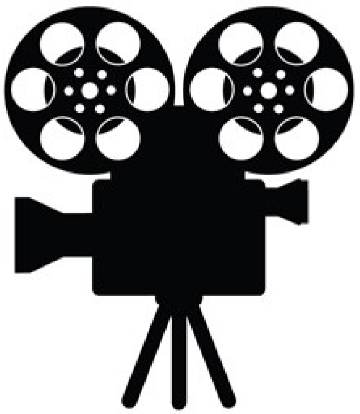
and

Hollywood principle

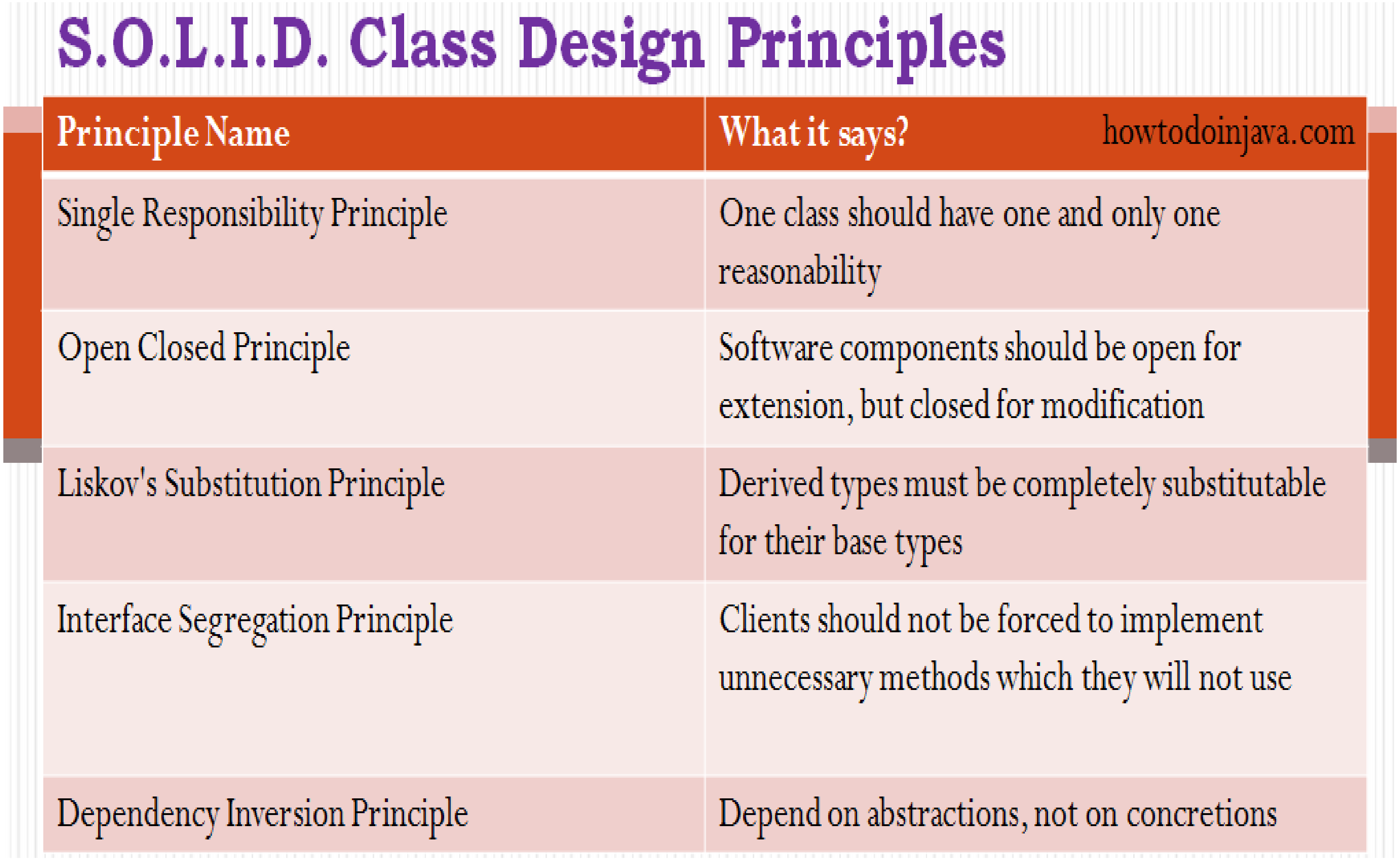
00011011 - A binary example

Famous **DI** s

IOC DI + IOC flow control + IOC program structure



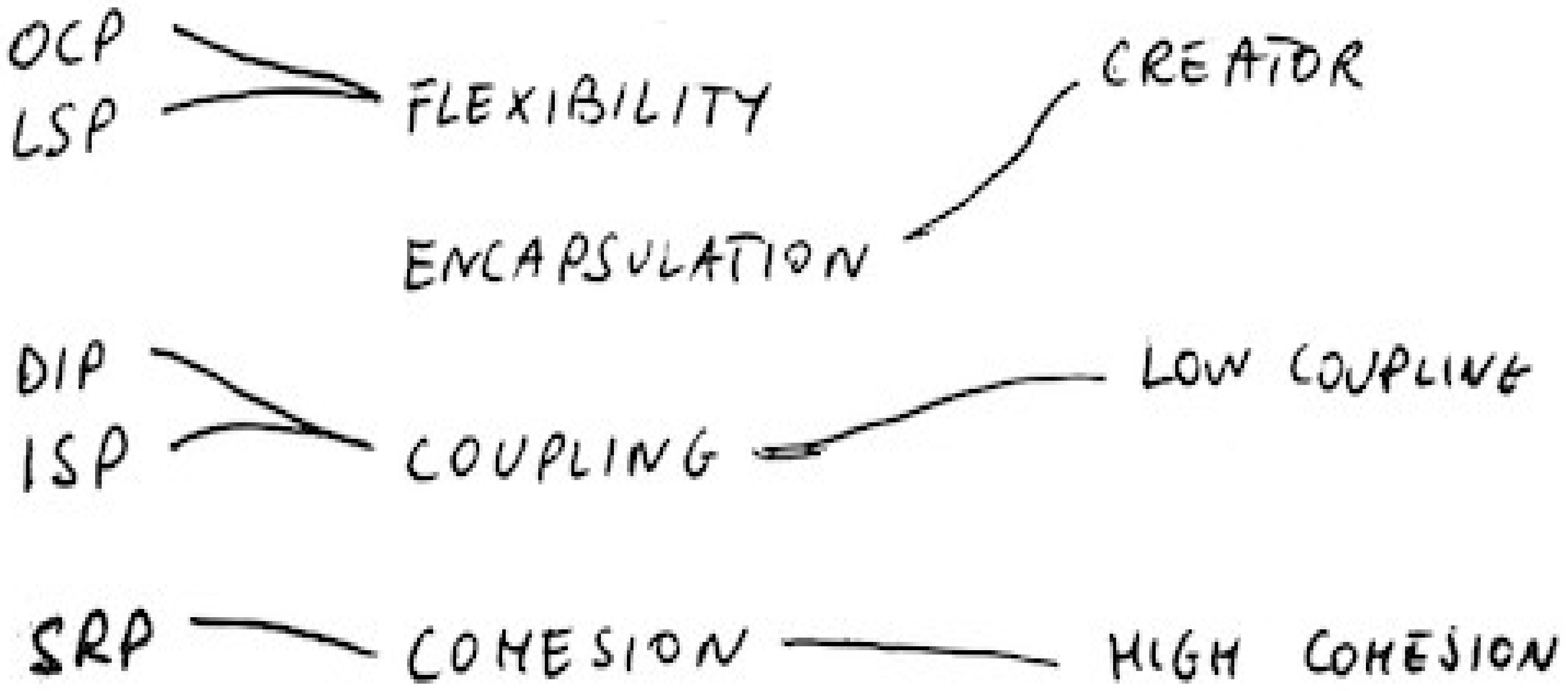
Revisiting SOLID principles



S.O.L.I.D. Class Design Principles

Principle Name	What it says? howtodoinjava.com
Single Responsibility Principle	One class should have one and only one reasonability
Open Closed Principle	Software components should be open for extension, but closed for modification
Liskov's Substitution Principle	Derived types must be completely substitutable for their base types
Interface Segregation Principle	Clients should not be forced to implement unnecessary methods which they will not use
Dependency Inversion Principle	Depend on abstractions, not on concretions

Revisiting SOLID principles



Must Know Design Patte

Singleton pattern

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects.

"The Singleton pattern is deceptively simple, even and especially for Java developers."

~ JavaWorld.com

Real examples being:

- We need just one entity that defines the configuration of the system.
- We get a single Runtime reference.
- We do not open two search panels for "Ctrl + F" pressed two times on a view.

Famous Singleton Implementations

Singleton with early initialization

```
Class Singleton{
Private static Singleton _instance = new Singleton();

Public static Singleton getInstance(){
    Return _instance;
}

// Other functionality of this Object
}
```

Famous Singleton Implementations

Singleton with lazy initialization for Single Threaded application

```
Class Singleton{
Private static Singleton _instance;

Public static Singleton getInstance(){
        if(_instance == null){
                _instance = new Singleton();
        }
Return _instance;
}
// Other functionality of this Object
}
```

Famous Singleton Implementations

Singleton with synchronized lazy initialization for Multi Threaded application

```
Class Singleton{
Private static Singleton _instance;
Public synchronized static Singleton getInstance(){
        if(_instance == null){
                _instance = new Singleton();
        }
Return _instance;
}
// Other functionality of this Object
}
```

Famous Singleton Implementations

Singleton with optimistic lazy initialization for Multi Threaded application

```
Class Singleton{
Private static Singleton _instance;
Public static Singleton getInstance(){
        if(_instance == null){
                synchronized(this){
                _instance = new Singleton();
        }}
Return _instance;
}
// Other functionality of this Object
}
```

Famous Singleton Implementations

Double locking threads and high performance double locked singleton object

```
Class Singleton{
Private static Singleton _instance;
Public static Singleton getInstance(){
        if(_instance == null){
                synchronized(Singleton.class){
                if(_instance == null){
                _instance = new Singleton();
        }}}
Return _instance;
}// Other functionality of this Object
}
```

Breaching Singletons

Subclassing

The Singleton Design Pattern is meant to give you control over access to the Singleton class. While I have mostly discussed the control of instantiation, other code can access your class another way: by subclassing it.

The uniqueness of the class cannot be imposed as a compile-time constraint on the subclass unless you use a private constructor. If you want to allow subclassing, for example, you might make the constructor protected. A subclass could then expose a public constructor, allowing anyone to make instances. Since an instance of a subclass is an instance of your superclass, you could find multiple instances of the Singleton.

Breaching Singletons

De – Serialization

If the Singleton class implements the java.io.Serializable interface, when a singleton is serialized and then deserialized more than once, there will be multiple instances of Singleton created. In order to avoid this the readResolve method should be implemented. See Serializable () and readResolve Method () in javadocs.

Breaching Singletons

ReflectionAPI

ReflectionAPI allows introspection of class files and would allow to powerfully change and control the access of constructors and methods.

```
Class singletonClass =
    Class.forName(_singletonClassName);
Constructor ctor =
    singletonClass.getDeclaredConstructor();
ctor.setAccessible(true);
ctor.newInstance();
```

Breaching Singletons

Javassist and JRebel

Some library play around with bytecode modification – and can easily breach your singleton implementation.

```
ClassPool pool = ClassPool.getDefault();
CtClass subClass = pool.makeClass("Subclass");
CtClass superClass = pool.get(_singletonClassName);
subClass.setSuperclass(superClass);
Class subclassClass = subClass.toClass();
Object obj = subclassClass.newInstance();
```

Breaching Singletons

Context of a singleton object

Example - Tomcat

NOT so famous Singleton Implementations

Static class as a singleton

A static class can also be used as a singleton object for your system. Since Java creates the static objects upon reference and maintains them in a separate memory space

Example **Math.java**

Benefits: Concentrate on the functionality – that is what a singleton is all about and exploit the power of statics in Java

Negatives: A static cannot subclass – is eagerly created and maintained upon reference.

NOT so famous Singleton Implementations

Enums as a singleton

An Enum can also be used as a singleton object for your system. In the second edition of his book Effective Java, Joshua Bloch claims that "a single-element enum type is the best way to implement a singleton".

```
public enum Singleton {
            INSTANCE;
    public void execute (String arg) {
            // Perform operation here
    }}
```

This approach implements the singleton by taking advantage of Java's guarantee that any enum value is instantiated only once in a Java program. Since Java enum values are globally accessible, so is the singleton, initialized lazily by the class loader. The drawback is that the enum type is somewhat inflexible

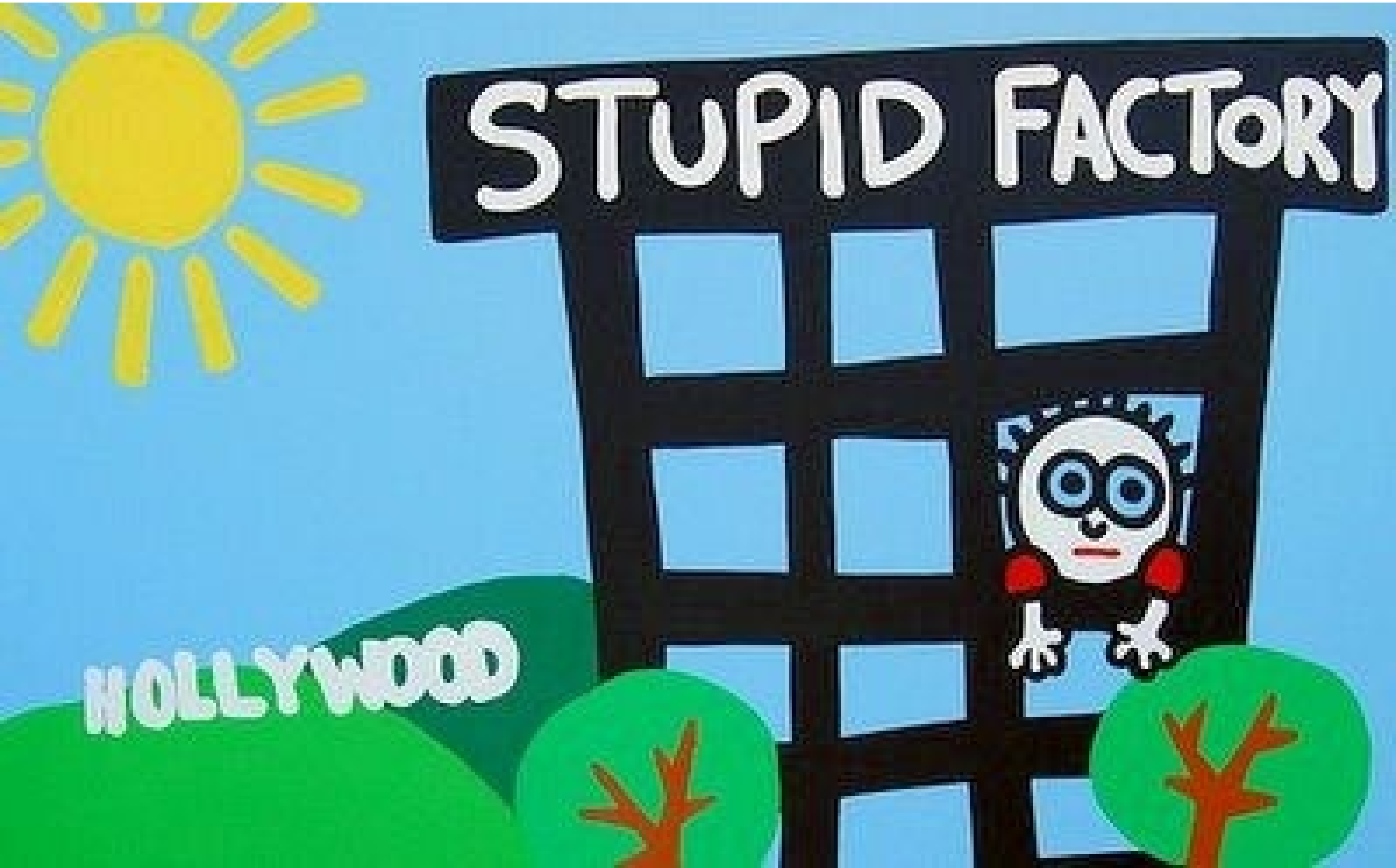
NOT so famous Singleton Implementations

Multiton

A key based registry of singleton reference. The multiton pattern expands on the singleton concept to manage a map of named instances as key-value pairs.

```
public class FooMultiton {
private static final Map<Object, FooMultiton> instances = new
    HashMap<Object, FooMultiton>();
   private FooMultiton() {    }
   public static synchronized FooMultiton getInstance(Object key) {
       FooMultiton instance = instances.get(key);
       if (instance == null) {
          instance = new FooMultiton();
          instances.put(key, instance);
       }
   return instance;
   }   // other fields and methods ...
}
```

Factory



Introduction

Factory Pattern is one of the creational pattern where an interface or a class is used to encapsulate creation of objects. It is more of umbrella term that covers more than one implementation strategies.

Following are major variants that are used :-

1)Simple Factory.

2)Factory Method.

3)Product Type registration.

Simple Factory or Parameterized Factory

-Most Commonly Used.

-Takes input some identifier of product to return its concrete implementation.

-Appropriate for simple hierarchies/groups and violate design principle.

```
public class RecordFactory{
          public Product createRecord(String type){
                    if (type.equals("Org"))
                              return new Organization();
                    if (type.equals("Person"))
                              return new Person();
           return null; // no match scenario
    }  }
```

Factory Method Pattern

-Relatively less used. Primarily where product hierarchies are involved.

-Uses inheritance mechanism thereby preserving certain principles to create objects.

-Used in conjunction with Abstract Factory.

Abstract Factory

Introduction

AbstractFactory pattern provides an interface to create different type of however related and/or dependent products without specifying any concrete implementation.

1) It typically uses Factory or Prototype pattern to create actual products.

Prototype pattern

The prototype pattern is a creational design pattern in software development. It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

Cloning

Prototype Manager

The PrototypeManager class is just a manager class that is used to add and retrieve prototypes by an index number, it has the following variable and methods:

prototypeList variable is the collection that stores all the prototype

AddPrototype method allows you to add a prototype to the collection and assigning it an index number

GetPrototype method allows you to retrieve a prototype from the collection using an index number

Builder Pattern

Builder Pattern separate the construction of a complex object from its representation so that the same construction process can create different representations.

Object pool pattern

The object pool pattern is a software creational design pattern that uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and destroying them on demand. A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object to the pool rather than destroying it; this can be done manually or automatically.

Object pools are primarily used for performance: in some circumstances, object pools significantly improve performance. Object pools complicate object lifetime, as objects obtained from and return to a pool are not actually created or destroyed at this time, and thus require care in implementation.

Immutability

5x-2=13
then
x = 3
But professor, you said yesterday that x was equal to 2!

Immutability

Advantages of immutability

- It helps to reduce memory usage(Allowing identical values to be combined together and referenced from various locations in your codes)
- It simplifies multi-threaded programming(Once you initialized, the value cannot be changed)
- Safe to work(There is no problem with alternatives)
- It simplifies the design and implementation(Re-usability of a String)

Disadvantages of immutability

- Less efficient(It creates a heavy garbage collection)

Object pool - examples

String pool

DB connection pool

Integer, long pool ??

LOONEY TUNES
"That's all Folks!"

Teaser for the next !!

Must know Design Patterns if you really want to be called a Software Engineer !!

1. **Adap**ter
2. De**cor**ator
3. Facade
4. **Pr**oxy
5. Chain **of** Responsibilities
6. Com**mand**
7. **State**
8. **Strategy**
9. Template
10. **Observer**

Anti-patterns