DESIGN PATTERNS

– BY WEBFUSE

AGENDA

- ➔ What is Design Pattern?
- ➔ Gang of Four!
- ➔ Design Pattern classification & usage.
- ➔ Singleton Design Pattern.
- ➔ Questions if any?

What is design pattern?

Template to solve a problem

Obtained by trial and error

Solution to general problems

Best Practices

Common Language among Developers/Architects

Development process is fast because of tested & proven development paradigms

Gang of Four(GOF)!

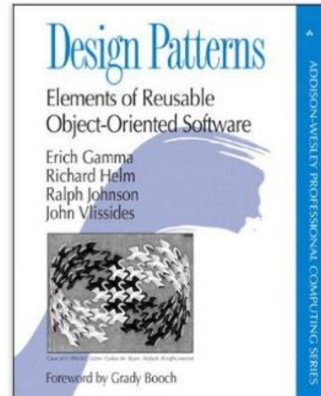
John Vlissides

Erich Gamma

Ralph Johnson

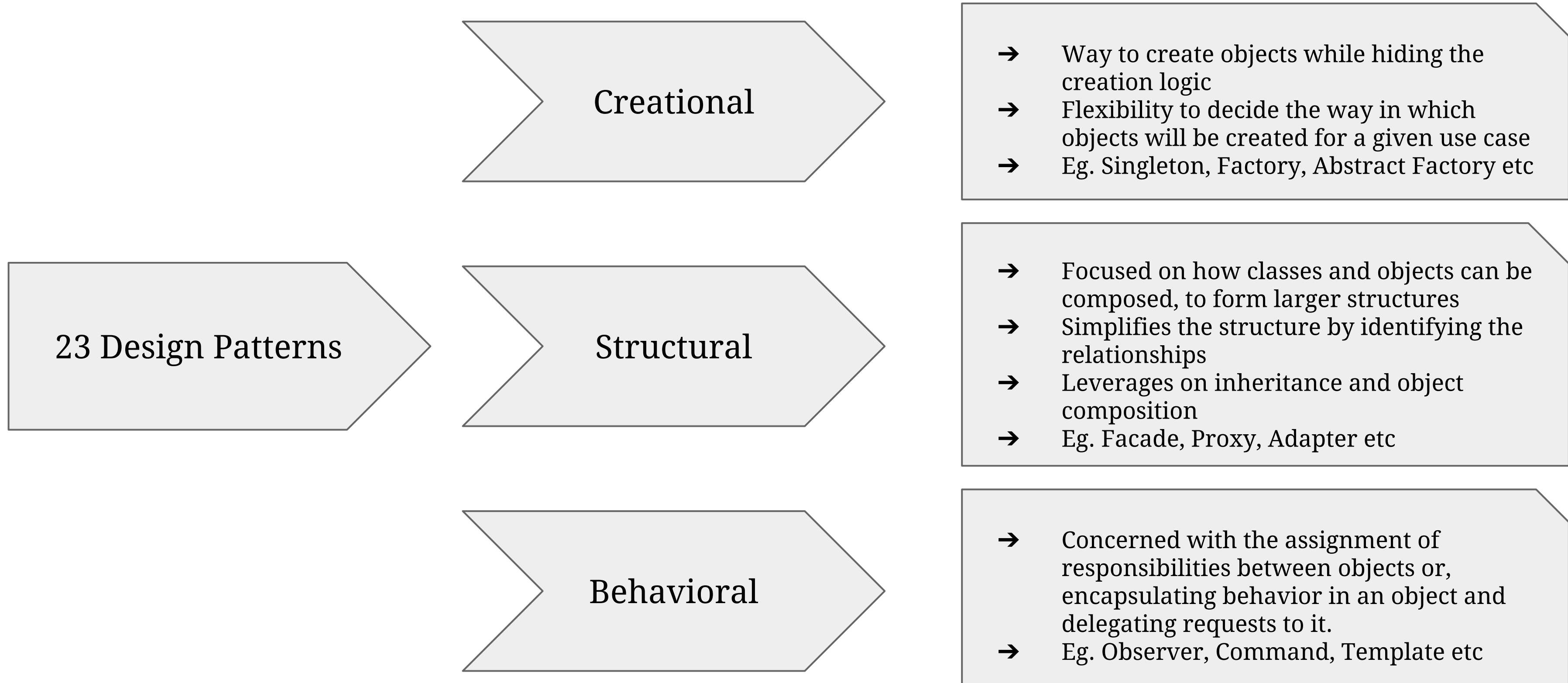
Richard Helm

Program to an interface not an implementation



Favor object composition over inheritance

Design Pattern classification & usage



Singleton Design Pattern

- A Creational Design Pattern category
- Used for resource intensive objects
- Restricts multiple instantiation of a class and ensures that only one instance of the class exists in the JVM
- Provides a global access point to get the instance of the class
- Used for logging, drivers objects, caching, thread pool, database connections etc.

Typical Implementation

Private Constructor → Private Static Variable → Public Static Method (Global access Point)

```
// Eager Initialization
public class SingletonObject {
// Private Static Variable
private static SingletonObject singletonInstance = new
SingletonDemo();

      // Private Constructor
      private SingletonDemo() {/* no instance */ }

      // Global Access Point
      public static SingletonDemo getSingletonInstance(){
            return singletonInstance;
      }
}
```

```
// Lazy initialization
public class SingletonObject {
  // Private Static Variable
  private static SingletonObject singletonInstance = null;
  // Private Constructor
  private SingletonDemo() {/* no instance */ }
  // Global Access Point
  public static SingletonObject getSingletonInstance(){
   if(singletonInstance == null){
      singletonInstance = new SingletonObject();
   }
   return singletonInstance;
  }
}
```

Challenges of Singleton

- Break By Reflection
- Serialization & Deserialization
- Clone Problem
- Multithreaded Environment Problem
- Static Holder Pattern
- Singleton Using Enum
- Multiple Class Loader
- Garbage Collection

Break by reflection

```
public class BreakByReflection {
	@SuppressWarnings({ "rawtypes", "unchecked" })
	public static void main(String[] args) throws Exception {
		SingletonObject singleton1 = SingletonObject.getSingletonInstance();
		SingletonObject singleton2 = SingletonObject.getSingletonInstance();

		print("singleton1", singleton1);
		print("singleton2", singleton2);

		// Break by Reflection
		Class clazz = Class.forName("com.singleton.SingletonObject");
		Constructor<SingletonObject> constructor = clazz.getDeclaredConstructor();
		constructor.setAccessible(true);

		SingletonObject singleton3 = constructor.newInstance();

		print("singleton3", singleton3);

	}

	static void print(String name, SingletonObject demoObject){
		System.out.println(String.format("Object: %s, HashCode: %d", name, demoObject.hashCode()));
	}
}
```

Break by reflection (fix)

```
public class SingletonObject {
	private static SingletonObject singletonInstance = null;

	// Private Constructor
	private SingletonObject() {
		System.out.println("Creating instance...");
		// Avoid Break By Reflection
		if (singletonInstance != null) {
			throw new RuntimeException("Can not create! Please use getInstance().");
		}
	}

	// Global Access Point
	public static SingletonObject getSingletonInstance() {
		if (singletonInstance == null) {
			singletonInstance = new SingletonObject();
		}
		return singletonInstance;
	}
}
```

Challenges of Singleton

- ➔ Break By Reflection
- ➔ Serialization & Deserialization
- ➔ Clone Problem
- ➔ Multithreaded Environment Problem
- ➔ Static Holder Pattern
- ➔ Singleton Using Enum
- ➔ Multiple Class Loader
- ➔ Garbage Collection

SERIALIZATION/DESERIALIZATION ISSUE

```
public class BreakBySerialization {
      @SuppressWarnings({ "resource" })
      public static void main(String[] args) throws Exception {
         SingletonObject singleton1 = SingletonObject.getSingletonInstance();
         SingletonObject singleton2 = SingletonObject.getSingletonInstance();

         print("singleton1", singleton1);
         print("singleton2", singleton2);

         // Break by Serialization
         ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("D:/tmp/singleton2.dat"));
         oos.writeObject(singleton2);

         ObjectInputStream ois = new ObjectInputStream(new FileInputStream("D:/tmp/singleton2.dat"));
         SingletonObject singleton3 = (SingletonObject) ois.readObject();

         print("singleton3", singleton3);
      }

      static void print(String name, SingletonObject demoObject) {
            System.out.println(String.format("Object: %s, HashCode: %d", name, demoObject.hashCode()));
      }
}
```

SERIALIZATION/DESERIALIZATION ISSUE (FIX)

```
public class SingletonObject implements Serializable {
	private static SingletonObject singletonInstance = null;

	// Private Constructor
	private SingletonObject() {
		System.out.println("Creating instance...");
	}

	// Global Access Point
	public static SingletonObject getSingletonInstance() {
		if (singletonInstance == null) {
			singletonInstance = new SingletonObject();
		}
		return singletonInstance;
	}

	// Avoid Break By Serialization
	private Object readResolve() throws ObjectStreamException {
		System.out.println("inside read resolve method...");
		return singletonInstance;
	}
}
```

Challenges of Singleton

- ➔ Break By Reflection
- ➔ Serialization & Deserialization
- ➔ Clone Problem
- ➔ Multithreaded Environment Problem
- ➔ Static Holder Pattern
- ➔ Singleton Using Enum
- ➔ Multiple Class Loader
- ➔ Garbage Collection

Clone problem

```
public class CloneProblem {
      public static void main(String[] args) throws Exception {
            SingletonObject singleton1 = SingletonObject.getSingletonInstance();
            SingletonObject singleton2 = SingletonObject.getSingletonInstance();

            print("singleton1", singleton1);
            print("singleton2", singleton2);

            SingletonObject singleton3 = (SingletonObject) singleton2.clone();

            print("singleton3", singleton3);

      }

      static void print(String name, SingletonObject demoObject) {
            System.out.println(String.format("Object: %s, HashCode: %d", name, demoObject.hashCode()));
      }
}
```

Clone problem (Fix)

```
public class SingletonObject implements Cloneable {
	private static SingletonObject singletonInstance = null;

	// Private Constructor
	private SingletonObject() {
		System.out.println("Creating instance...");
	}

	// Global Access Point
	public static SingletonObject getSingletonInstance() {
		if (singletonInstance == null) {
			singletonInstance = new SingletonObject();
		}
		return singletonInstance;
	}

	// Avoid Cloning
	@Override
	protected Object clone() throws CloneNotSupportedException {
		throw new CloneNotSupportedException("Clone is not supported in this class");
		// return singletonInstance;
	}
}
```

Challenges of Singleton

- ➔ Break By Reflection
- ➔ Serialization & Deserialization
- ➔ Clone Problem
- ➔ Multithreaded Environment Problem
- ➔ Static Holder Pattern
- ➔ Singleton Using Enum
- ➔ Multiple Class Loader
- ➔ Garbage Collection

Multithreaded environment problem

```
public class MultithreadedEnvironment {
	public static void main(String[] args) throws Exception {
		ExecutorService executorService = Executors.newFixedThreadPool(2);
		executorService.submit(MultithreadedEnvironment::useSingleton);
		executorService.submit(MultithreadedEnvironment::useSingleton);

		executorService.shutdown();
	}

	static void useSingleton() {
		SingletonMT singletonObj = SingletonMT.getSingletonInstance();
		print("SingletonObject", singletonObj);
	}

	static void print(String name, SingletonMT demoObject) {
		System.out.println(String.format("Object: %s, HashCode: %d", name, demoObject.hashCode()));
	}
}
```

Multithreaded environment (Fix - method level)

```
public class SingletonMT {
      private static SingletonMT singletonInstance = null;

      // Private Constructor
      private SingletonMT() {
            System.out.println("creating.....");
      }

      // Global Access Point
      public static synchronized SingletonMT getSingletonInstance() {// Method level
            if (singletonInstance == null) {
                  singletonInstance = new SingletonMT();
            }
            return singletonInstance;
      }
}
```

Multithreaded environment (Double check locking)

```
public class SingletonMT {
	private static SingletonMT singletonInstance = null;

	// Private Constructor
	private SingletonMT() {
		System.out.println("creating.....");
	}

	// Global Access Point
	public static SingletonMT getSingletonInstance() {
		//Double Checked Locking
		if (singletonInstance == null) {// First check
			synchronized(SingletonMT.class){
				if(singletonInstance == null){// Double check
					singletonInstance = new SingletonMT();
				}
			}
		}
		return singletonInstance;
	}
}
```

Multithreaded environment (Half baked instance)

```
public class SingletonMT {
// Introduce volatile to make sure the change to a variable is only published when the change is complete
// Volatile makes sure that a write happens fully before the read on that object
      private static volatile SingletonMT singletonInstance = null;

      // Private Constructor
      private SingletonMT() {
            System.out.println("creating.....");
      }

      // Global Access Point
      public static SingletonMT getSingletonInstance() {
            //Double Checked Locking
            if (singletonInstance == null) {// First check
                  synchronized(SingletonMT.class){
                        if(singletonInstance == null){// Double check
                              singletonInstance = new SingletonMT();
                        }
                  }
            }
            return singletonInstance;
      }
}
```

Challenges of Singleton

- ➔ Break By Reflection
- ➔ Serialization & Deserialization
- ➔ Clone Problem
- ➔ Multithreaded Environment Problem
- ➔ Static Holder Pattern
- ➔ Singleton Using Enum
- ➔ Multiple Class Loader
- ➔ Garbage Collection

Static holder pattern

```
public class SingletonSH {
	// Static holder pattern is also considered as the smartest replace for Double-check-locking
	// Private Constructor
	private SingletonSH() {
		System.out.println("creating.....");
	}

	public static SingletonSH getInstance(){
		return Holder.INSTANCE;
	}

	static class Holder{
		static final SingletonSH INSTANCE = new SingletonSH();
	}
}
```

Challenges of Singleton

- Break By Reflection
- Serialization & Deserialization
- Clone Problem
- Multithreaded Environment Problem
- Static Holder Pattern
- Singleton Using Enum
- Multiple Class Loader
- Garbage Collection

Singleton using enum

```java
public enum SingletonEnum {
      INSTANCE;

      public String sayHelloWorld(){
            return "Hello World!";
      }
}

// Against the intent of enum but still useful and highly recommended

SingletonEnum singletonObj = SingletonEnum.INSTANCE;
```

Challenges of Singleton

- ➔ Break By Reflection
- ➔ Serialization & Deserialization
- ➔ Clone Problem
- ➔ Multithreaded Environment Problem
- ➔ Static Holder Pattern
- ➔ Singleton Using Enum
- ➔ Multiple Class Loader
- ➔ Garbage Collection

Multiple class loader

Because multiple classloaders are commonly used in many situations, including servlet containers, we can end up with multiple singleton instances no matter how carefully we have implemented our singleton classes.

If we want to make sure the same classloader loads your singletons, we must specify our own class loader as below:

```
private static Class getClass(String classname) throws ClassNotFoundException {
      ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
      if(classLoader == null){
         classLoader = Singleton.class.getClassLoader();
      }
      return (classLoader.loadClass(classname));
   }
}
```

Challenges of Singleton

- ➔ Break By Reflection
- ➔ Serialization & Deserialization
- ➔ Clone Problem
- ➔ Multithreaded Environment Problem
- ➔ Static Holder Pattern
- ➔ Singleton Using Enum
- ➔ Multiple Class Loader
- ➔ Garbage Collection

Garbage Collection

A bug prior to Java 1.2 when singleton instance could be garbage collected if there was no global reference to it but that was fixed in Java 1.2 and only way now it can be eligible for garbage collection only if the class loader that loaded this class was garbage collected.

REFERENCES

- https://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom
- http://stackoverflow.com/questions/70689/what-is-an-efficient-way-to-implement-a-singleton-pattern-in-java

Questions and queries?