CMPE 202

Gang of Four Design Patterns

Singleton

Motivation

- Want to have one-and-only-one instance of a class in a system
- Examples:
  - A Print Spooler
  - An Database Key Generator

Intent

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

Applicability

Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Participants

- **Singleton**

    - defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).

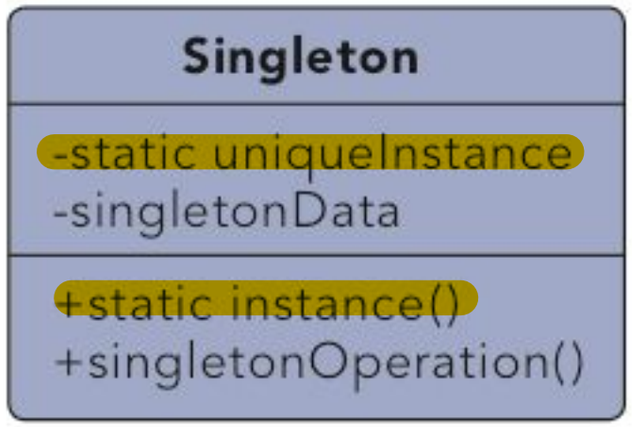
    - may be responsible for creating its own unique instance.

Collaborations

- Clients access a Singleton instance solely through Singleton's Instance operation.

SINGLETON

Object Creational

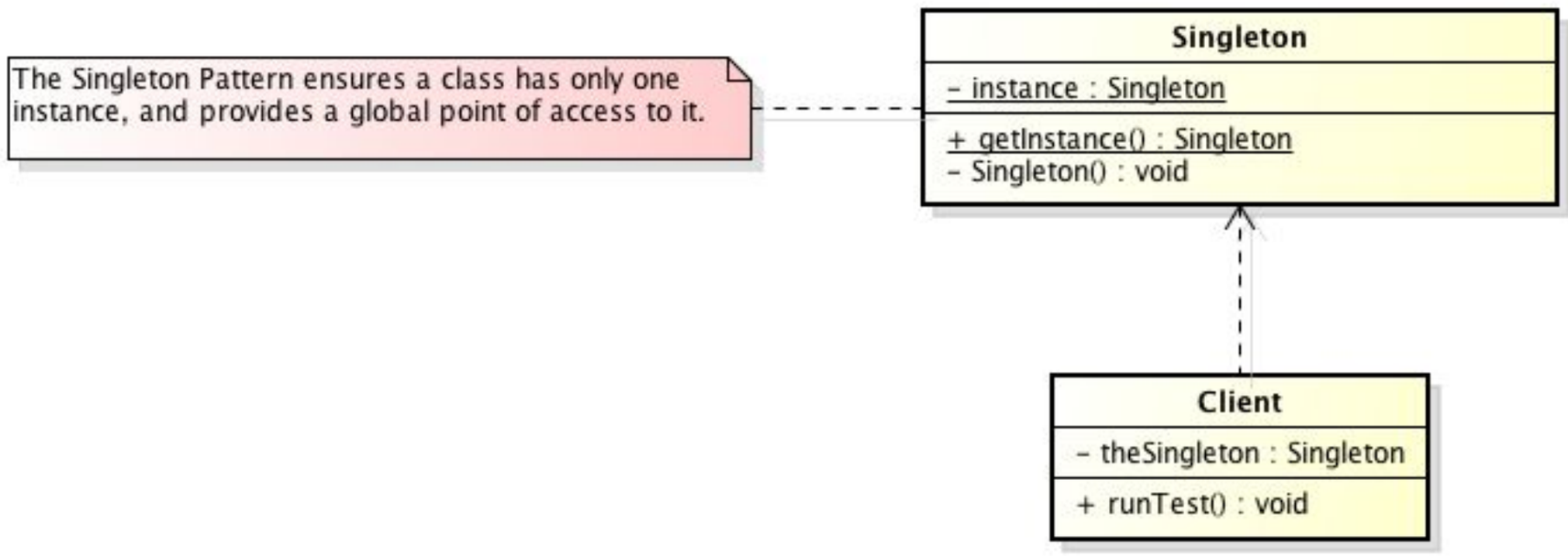


Purpose

Ensures that only one instance of a class is allowed within a system.

Use When

- Exactly one instance of a class is required.
- Controlled access to a single object is necessary.




```java
/* classic version */

public class Singleton {

    private static Singleton instance;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

```java
public class Client {

    private Singleton theSingleton;

    public void runTest() {

        // error - can not instantiate directly
        //theSingleton = new Singleton() ;

        // access the singleton instance
        theSingleton = Singleton.getInstance() ;

    }

}
```

```java
/* classic version */

public class Singleton {

    private static Singleton instance;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

```java
/* thread safe version */

public class SingletonThreadSafe {

    private static SingletonThreadSafe uniqueInstance;

    private SingletonThreadSafe() {
    }

    public static synchronized SingletonThreadSafe getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new SingletonThreadSafe();
        }
        return uniqueInstance;
    }

}
```

```java
/* double check lock version -- not guaranteed to work prior to Java 5  */

public class SingletonDoubleCheckLock {

    private volatile static SingletonDoubleCheckLock uniqueInstance;

    private SingletonDoubleCheckLock() {
    }

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