Object-Oriented Programming

Design Pattern Singleton

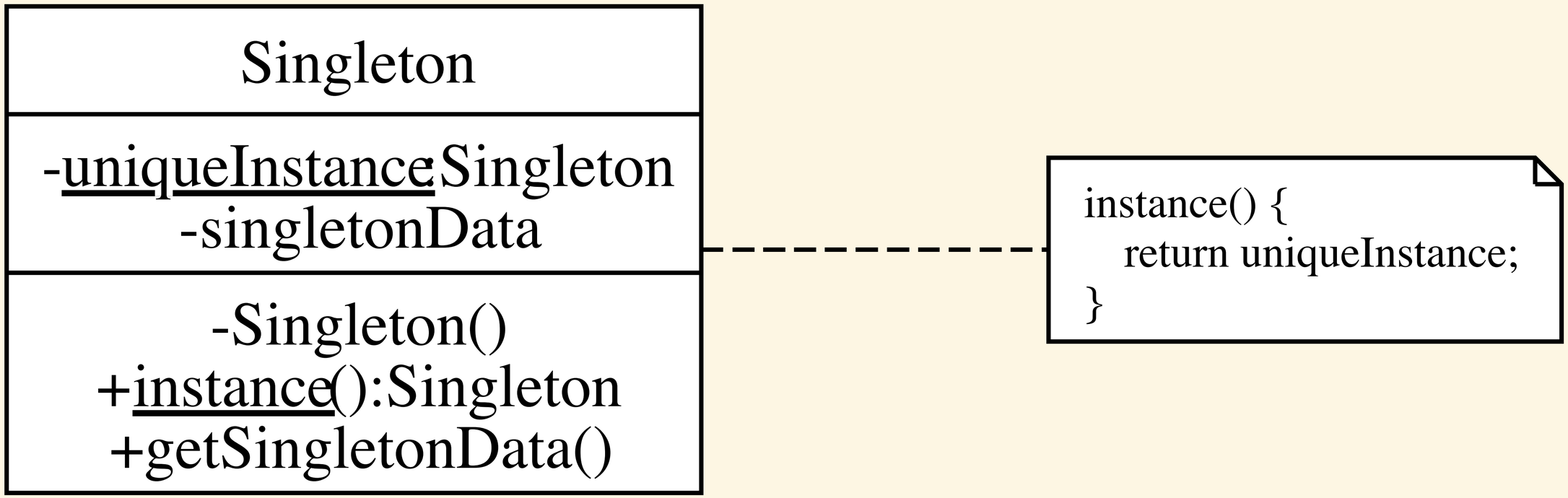
Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

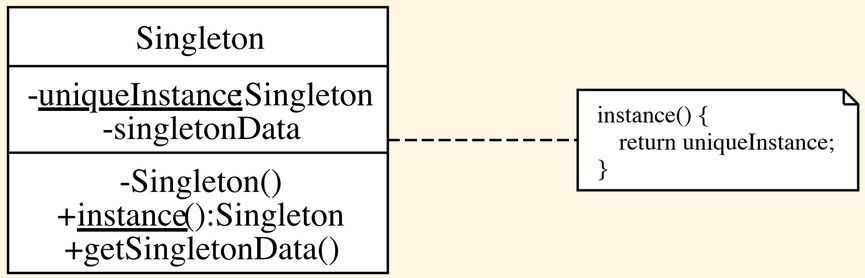
Singleton

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

- Structural Pattern


Singleton
-uniqueInstance:Singleton
-singletonData
-Singleton()
+instance():Singleton
+getSingletonData()
instance() {
return uniqueInstance;
}

Singleton: Motivation

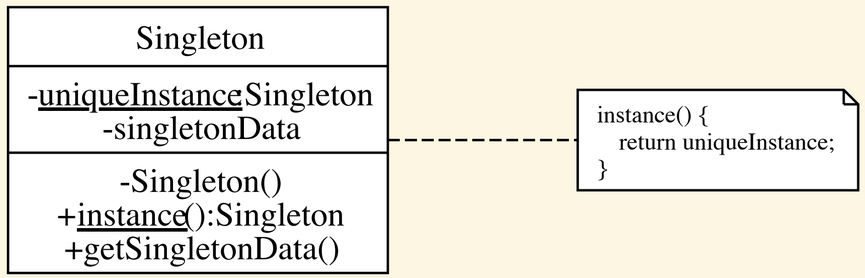


- Sometimes, a class must have exactly one instance, e.g., a class that maps to a physical resource

- How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible but doesn't keep you from instantiating multiple objects.

- A better solution is to make the class itself responsible for keeping track of its sole instance, i.e., a Singleton

Singleton: Applicability

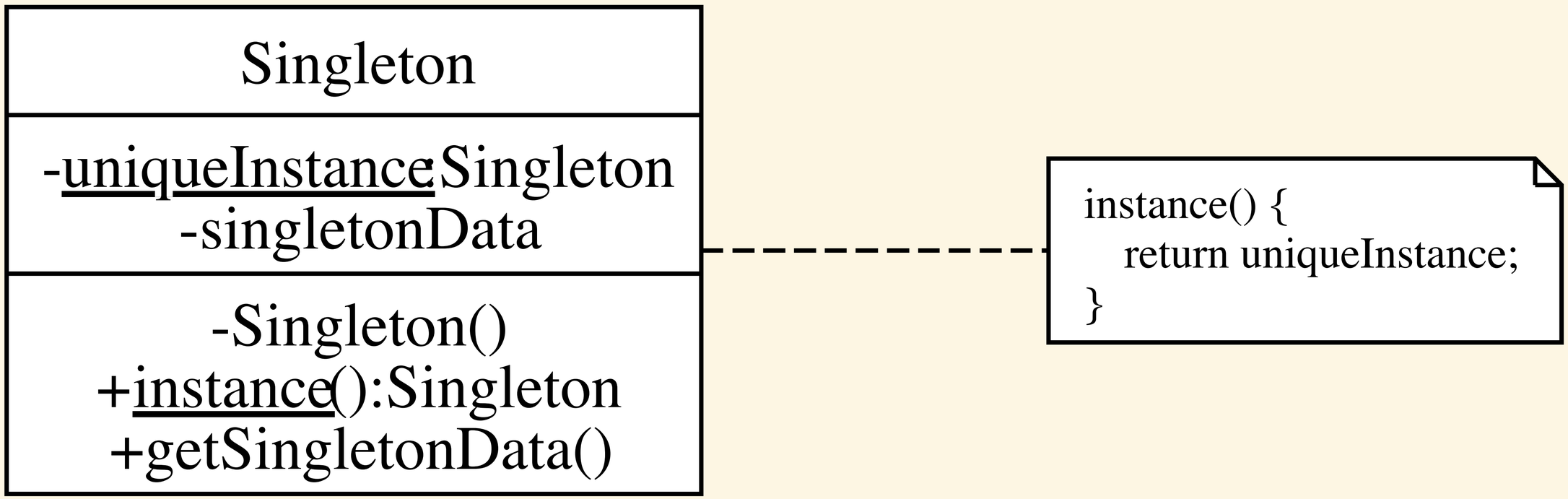


- Use the Singleton pattern when:

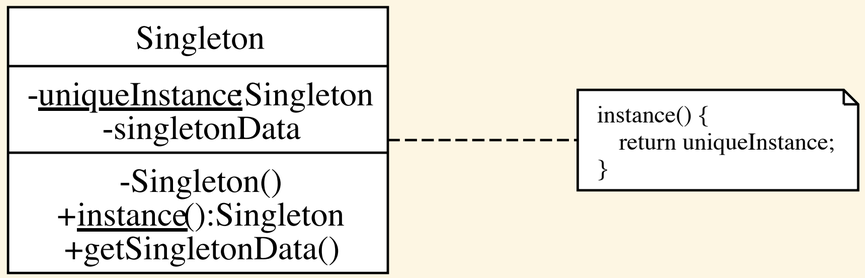
There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point

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

Singleton: Structure



Singleton: Participants



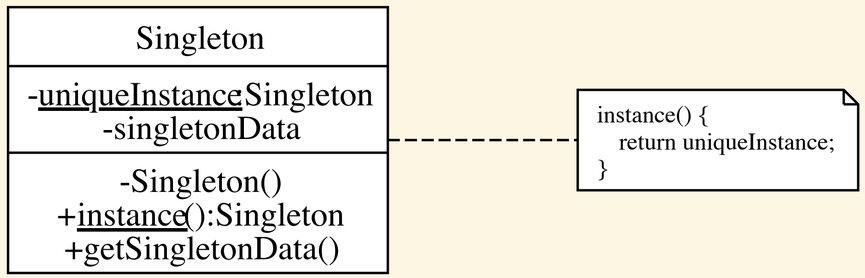
- *Singleton*

  Defines an instance operation that lets clients access a unique instance

  The instance (in C++) is a static member function

  Class may be responsible for creating a unique instance (typical)

Singleton: Consequences



- Controls access to the sole instance
- In place of global variables, and avoids polluting the global namespace
- The Singleton class can be subclassed
- Once in place, you can change your mind and allow more than one instance
- More flexible than static member functions, which are not polymorphic (i.e., cannot make them virtual)

Implementation

```
class MazeFactory {
public:
    // existing interface ...

    static MazeFactory* instance();
protected:
    MazeFactory();
private:
    static MazeFactory* pinstance;
};
```

```
MazeFactory* MazeFactory::pinstance{ nullptr };

// @NOTE Not thread safe
MazeFactory* MazeFactory::instance() {

    if (pinstance == nullptr) {
        pinstance = new MazeFactory;
    }

    return pinstance;
}
```

Thread-Safe Implementation

```cpp
#include <atomic>
#include <mutex>

class MazeFactory {
public:
    // existing interface ...

    static MazeFactory* instance();
protected:
    MazeFactory();
private:
    static std::atomic<MazeFactory*> pinstance;
    static std::mutex m;
};
```

```cpp
std::atomic<MazeFactory*> MazeFactory::pinstance{ nullptr };
std::mutex MazeFactory::m;

MazeFactory* MazeFactory::instance() {

    {
        std::lock_guard<std::mutex> lock(m);
        if (pinstance == nullptr) {
            pinstance = new MazeFactory();
        }
    }

    return pinstance;
}
```

DCL (Double-Checked Locking) Pattern

```h
#include <atomic>
#include <mutex>

class MazeFactory {
public:
    // existing interface ...

    static MazeFactory* instance();
protected:
    MazeFactory();
private:
    static std::atomic<MazeFactory*> pinstance;
    static std::mutex m;
};
```

```cpp
std::atomic<MazeFactory*> MazeFactory::pinstance{ nullptr };
std::mutex MazeFactory::m;

MazeFactory* MazeFactory::instance() {

    if (pinstance == nullptr) {
        std::lock_guard<std::mutex> lock(m);
        if (pinstance == nullptr) {
            pinstance = new MazeFactory();
        }
    }

    return pinstance;
}
```

DCL Comparison

```
std::atomic<MazeFactory*> MazeFactory::pinstance{ nullptr };
std::mutex MazeFactory::m;

MazeFactory* MazeFactory::instance() {

    {
        std::lock_guard<std::mutex> lock(m);
        if (pinstance == nullptr) {
            pinstance = new MazeFactory();
        }
    }

    return pinstance;
}
```

```
std::atomic<MazeFactory*> MazeFactory::pinstance{ nullptr };
std::mutex MazeFactory::m;

MazeFactory* MazeFactory::instance() {

    if (pinstance == nullptr) {
        std::lock_guard<std::mutex> lock(m);
        if (pinstance == nullptr) {
            pinstance = new MazeFactory();
        }
    }

    return pinstance;
}
```

std::call_once()

```
#include <atomic>
#include <mutex>

class MazeFactory {
public:
    // existing interface

    static MazeFactory* instance();
protected:
    MazeFactory();
private:
    static std::atomic<MazeFactory*> pinstance;
    static std::once_flag flag;
};
```

```
std::atomic<MazeFactory*> MazeFactory::pinstance{ nullptr };
std::once_flag MazeFactory::flag;

MazeFactory* MazeFactory::instance() {

    std::call_once(flag, [](){
        pinstance = new MazeFactory();
    });

    return pinstance;
}
```

std::call_once() Comparison

```cpp
std::atomic<MazeFactory*> MazeFactory::pinstance{ nullptr };
std::mutex MazeFactory::m;

MazeFactory* MazeFactory::instance() {

    if (pinstance == nullptr) {
        std::lock_guard<std::mutex> lock(m);
        if (pinstance == nullptr) {
            pinstance = new MazeFactory();
        }
    }

    return pinstance;
}
```

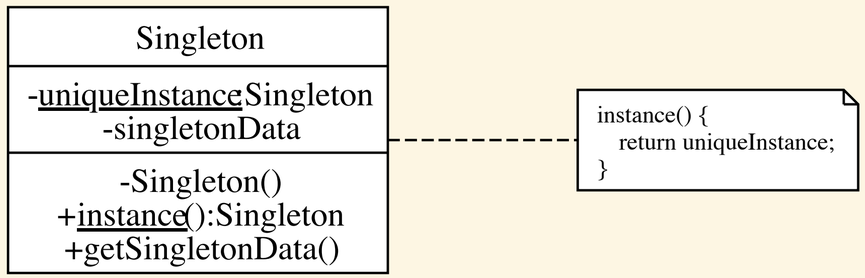
```cpp
std::atomic<MazeFactory*> MazeFactory::pinstance{ nullptr };
std::once_flag MazeFactory::flag;

MazeFactory* MazeFactory::instance() {

    std::call_once(flag, [](){
        pinstance = new MazeFactory();
    });

    return pinstance;
}
```

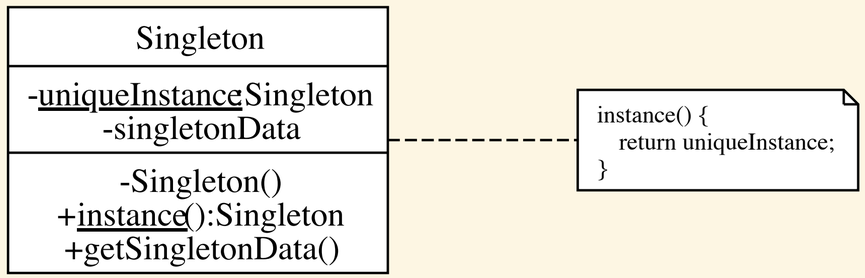
Related Patterns



- *Abstract Factory*, *Builder*, *Prototype*

Can be implemented using the Singleton pattern

Discussion



- Some strong opinions against the necessity of using the Singleton pattern

- Singleton may be used to get around a poor design

- Why? A singleton is basically a global variable (object)

- Note that the implementations of many standard data structures are not entirely thread-safe