Singleton

(Creational Pattern)

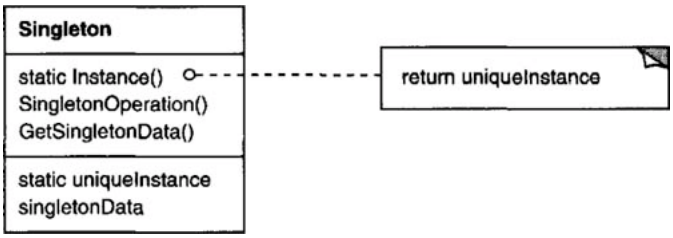
Singleton

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it.
- Motivation
  - One printer spooler, one file system, one window manager
- Problem
  - Ensure there is a single instance
  - Global variable: Accessible, but single instance not guaranteed
- Solution
  - Make the class responsible for its instance

Use Singleton 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.

Structure



Implementation

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```


Participants

- Singleton
  - defines an Instance operation that lets clients access its unique instance.
    - Instance is a class operation (a static member function in C++).
  - may be responsible for creating its own unique instance.
- Clients access a Singleton instance solely through Singleton's Instance operation.

Benefits

- Controlled access to sole instance.
  - Control over how and when clients access it.
- Permits refinement of operations and representation by subclassing.
- Permits a variable number of instances instead of one.
- More flexible than class operations.
  - Alternatively, package a singleton's functionality using class operations.
  - But class operations are not appropriate for change.
  - Subclasses can't override class operations.

Benefits

- Improvement over global variables:
  - Reduced name space.
  - Global instances are always created; with Singleton, there is option to not create an instance at all.
  - Singleton instantiation can be delayed to wait for runtime information.
  - Order of calling constructors of global variables is not defined. Problematic if there are dependences between instances.

Subclassing a Singleton

- Option 1: Use environment variable to decide which class to instantiate.

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ... other possible subclasses

        } else {        // default
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```


Subclassing a Singleton

- Option 2: Use a registry of singletons.

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup

        _instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}
```

Subclassing a Singleton

- Option 2: Use a registry of singletons.
  - Where do Singleton classes register themselves?
  - E.g. In their constructors.

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

  - Define a static instance of MySingleton so that constructor is invoked.

```
static MySingleton theSingleton;
```

  - Singleton class is no longer responsible for creating the singleton.
  - But, all possible Singleton subclasses must be instantiated