DIGITAL ARTS & ENTERTAINMENT

Sound & Services

howest
university of applied sciences

Meltem Ozcelik (Concept: Amandine Gerard) / Daniil Shashenkov (Concept: Tooth Wu) / Joao Desager (Concept: Tan Zhi Hui)

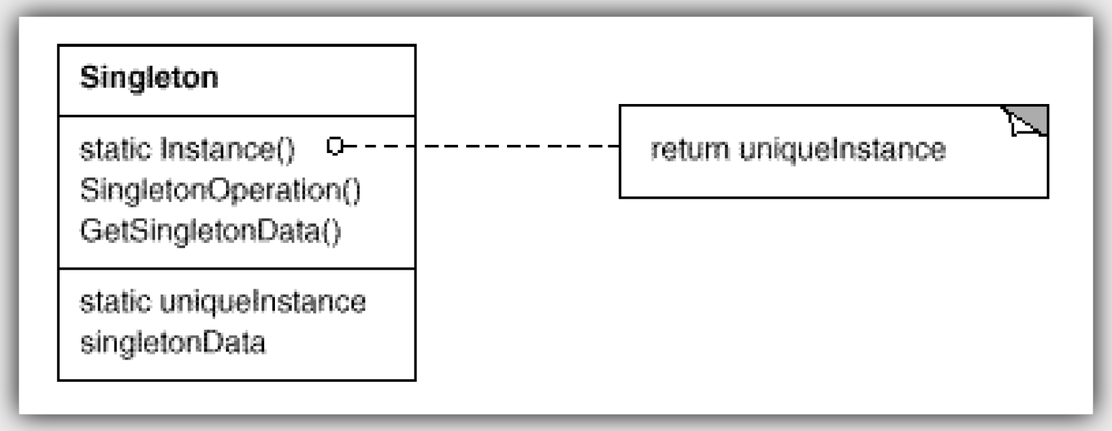
Soundsystem

We’re going to try and create a sound system for our engine.

Main function: playing a sound at a certain volume.

Who are our clients and how do we get them to play a sound? (Remember Event Queue)

Singleton



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

Implementation ?

```
using sound_id = unsigned short;
class sound_system
{
  static sound_system* instance;
public:
  static sound_system* instance()
  {
    if(instance == nullptr)
      instance = new sound_system();
    return instance;
  }

  void play(sound_id id, float volume)
  {
    // ... code that plays sound "id" at given volume ...
  }
};
```

Implementation ?

```
using sound_id = unsigned short;
class sound_system
{
  static sound_system* instance;
  sound_system() = default;
public:
  static sound_system* instance()
  {
    if(instance == nullptr)
      instance = new sound_system();
    return instance;
  }

  void play(sound_id id, float volume)
  {
    // ... code that plays sound "id" at given volume ...
  }
};
```

Implementation ?

```
using sound_id = unsigned short;
class sound_system
{
  static sound_system* instance;
  sound_system() = default;
  sound_system(const sound_system& other) = delete;
  sound_system(sound_system&& other) = delete;
  sound_system& operator=(const sound_system& other) = delete;
  sound_system& operator=(sound_system&& other) = delete;
public:
  static sound_system* instance()
  {
    if(instance == nullptr)
      instance = new sound_system();
    return instance;
  }

  void play(sound_id id, float volume)
  {
    // ... code that plays sound "id" at given volume ...
  }
};
```

Implementation ?

```
using sound_id = unsigned short;
class sound_system
{
  static std::mutex mutex;
  static sound_system* instance;
  sound_system() = default;
  sound_system(const sound_system& other) = delete;
  sound_system(sound_system&& other) = delete;
  sound_system& operator=(const sound_system& other) = delete;
  sound_system& operator=(sound_system&& other) = delete;
public:
  static sound_system* instance()
  {
    std::lock_guard<std::mutex> lock(mutex);
    if(instance == nullptr)
      instance = new sound_system();
    return instance;
  }

  void play(sound_id id, float volume)
  {
    // ... code that plays sound "id" at given volume ...
  }
};
```

Implementation ?

```
using sound_id = unsigned short;
class sound_system
{
  sound_system() = default;
  sound_system(const sound_system& other) = delete;
  sound_system(sound_system&& other) = delete;
  sound_system& operator=(const sound_system& other) = delete;
  sound_system& operator=(sound_system&& other) = delete;
public:
  static sound_system& instance()
  {
    static sound_system instance{};
    return instance;
  }

  void play(sound_id id, float volume)
  {
    // ... code that plays sound "id" at given volume ...
  }
};
```

Implementation - CRTP

```
template <typename T>
class singleton
{
public:
  static T& instance()
  {
    static T instance{};
    return instance;
  }

  virtual ~singleton() = default;
  singleton(const singleton& other) = delete;
  singleton(singleton&& other) = delete;
  singleton& operator=(const singleton& other) = delete;
  singleton& operator=(singleton&& other) = delete;

protected:
  singleton() = default;
};
```

```
using sound_id = unsigned short;
class sound_system final :
  public singleton<sound_system>
{
  friend class singleton<sound_system>;
  sound_system() = default;
public:
  void play(sound_id id, float volume)
  {
    // ... code that plays sound "id" at given volum
  }
};
```

Singleton

But Singletons have a bad rep. It’s sometimes deemed an “Anti-Pattern”

- They’re still global variables
- These couple code tightly.
- And invisibly.
- Tons of singles
- Difficult to unit test

```
class SomeClass {
public:
  void SomeMethod() {
    Actor a = ActorManager::instance().GetActor(someId);
    // do someting with the actor
  }
};
```

An alternative is **dependency injection**.

Dependency injection

“Dependency injection” is a fancy word for giving an object or method its instance variables or parameters needed to do the job.

```
class SomeClass {
public:
  void SomeMethod(ActorManager& actorManager) {
    Actor a = actorManager.GetActor(someId);
    // do someting with the actor
  }
};
```

It solves two problems: the dependency is now visible and you can unit test.

- It’s still tightly coupled though.
- You don’t want to pass the logging system to every method…
- Time is also an example that can be cumbersome.

Service locator

Using a singleton couples the calling code to the concrete implementation of the class.

“That’s like giving 10000 strangers your phone number so they can call you” - That makes a service locator the telephone book.

The service locater itself is a singleton, or completely static.

- Services register for duty with the locator.
- Clients must keep in mind that the service might not be found!

Implementation

The service:

```
using sound_id = unsigned short;
class sound_system
{
public:
  virtual ~sound_system() = default;
  virtual void play(const sound_id id, const float volume) = 0;

  // ...and other relevant methods of course...
};
```

Is an ***interface***!

Implementation

```
using sound_id = unsigned short;
class sound_system
{
public:
  virtual ~sound_system() = default;
  virtual void play(const sound_id id, const float volume) = 0;
};
```

The locator:

```
class servicelocator final
{
  static std::unique_ptr<sound_system> _ss_instance;
public:
  static sound_system& get_sound_system() { return *_ss_instance; }
  static void register_sound_system(std::unique_ptr<sound_system>&& ss) { _ss_instance = std::move(ss)
};
```

Is ***final***!

Implementation

Usage:

```
class sdl_sound_system final : public sound_system :
public:
  void play(const sound_id id, const float volume) override {
    // lots of sdl_mixer code
  }
};
```

```
void main()
{
  // at the start: register a sound system
  servicelocator::register_sound_system(std::make_unique<sdl_sound_system>());

  // ...lots of code...

  // start using the sound system.
  auto& ss = servicelocator::get_sound_system();
  ss.play(10, 100);
}
```

Decorator

```
class logging_sound_system final : public sound_system {
  std::unique_ptr<sound_system> _real_ss;
public:
  logging_sound_system(std::unique_ptr<sound_system>&& ss) : _real_ss(std::move(ss)) {}
  virtual ~logging_sound_system() = default;

  void play(const sound_id id, const float volume) override {
    _real_ss->play(id, volume);
    std::cout << "playing " << id << " at volume " << volume << std::endl;
  }
};
```

```
void main() {
#if _DEBUG
  servicelocator::register_sound_system(
    std::make_unique<logging_sound_system>(std::make_unique<sdl_sound_system>()));
#else
  servicelocator::register_sound_system(std::make_unique<sdl_sound_system>());
#endif
  // ... code ...
  auto& ss = servicelocator::get_sound_system();
  ss.play(10, 100);
}
```

Danger!

```
void main()
{
  // uh-oh
  auto& ss = servicelocator::get_sound_system();
  ss.play(10, 100);

  // ... code

  servicelocator::register_sound_system(std::make_unique<sdl_sound_system>());

  // ... code

  auto& ss = servicelocator::get_sound_system();
  ss.play(10, 100);
}
```

Default

```
class null_sound_system final : public sound_system
{
  void play(const sound_id, const float) override {}
};

class servicelocator final
{
  static std::unique_ptr<sound_system> _ss_instance;
public:
  static sound_system& get_sound_system() { return *_ss_instance; }
  static void register_sound_system(std::unique_ptr<sound_system>&& ss) {
    _ss_instance = ss == nullptr ? std::make_unique<null_sound_system>() : std::move(ss);
  }
};
```

```
//... somehwere in a cpp:
std::unique_ptr<sound_system> servicelocator::_ss_instance{ std::make_unique<null_sound_system>() };
```

Service locator

You can change a service while running

- For muting the sound
- For applying a different render mode
- To have another input controller
- ...

The service locator does not have to be global, it can also be local to a class.

- The `GetComponent<T>()` method in Unity is exactly that.

Sound

What is it that this “play” function must do?

```
void dae::sdl_sound_system::play(const sound_id id, const float volume)
{
  auto audioclip = audioclips[id];
  if (!audioclip->is_loaded())
    audioclip->load();
  audioclip->set_volume(volume);
  audioclip->play();
}
```

- Load the audioclip if not loaded
- Set volume
- Play

Sound

With a small test sound this takes ~9ms! That’s half a frame!

```
auto start = high_resolution_clock::now();

servicelocator::get_sound_system().play(pacman::pacman_dies, 1.0f);

auto end = high_resolution_clock::now();
auto elapsed = duration_cast<microseconds>(end - start).count();
std::cout << "play took: " << elapsed << "ms\n";
```

What can help?

Exercise

Implement an audio system as a service that you get via a **service locator**.

Create these audio services:

- A regular one, that plays sounds
- A logging service, that ***also*** logs the sounds to std::cout

You can use **SDL_mixer** for this:

- https://github.com/libsdl-org/SDL_mixer
- But other sound systems are allowed too.

Exercise

The audio needs to be loaded and played on **a different thread**

Make use of an **Event Queue** to add “play sound” requests for the other thread
Your thread processes the requests from the queue and while there is nothing in it, **does nothing**.

- Think: how would you implement this? The sound thread needs to be **notified** when there is work to be done.

Also think about **Pimpl**! We do not want to expose the used audio library to the user of our engine.

Submit a version of your engine + game where some sounds can be played

- Preferably via in game actions/events
- Or with a key if you don't have a game yet

The engine must be a **static library** by now.

Singletons in Unity

What I often see

```
public class GameController
{
  public GameController Instance { get; private set; }

  private void Awake()
  {
    if(Instance == null)
      Instance = this;
    else
      Destroy(gameObject);
  }

  // ... more code ...
}
```

What's wrong with this?

Check this

```
public abstract class MonoSingleton<T> : MonoBehaviour where T : MonoSingleton<T>
{
    static T m_Instance;
    static bool hasBeenCreated;
    public static T Instance { get {
      if (m_Instance == null)
      {
          m_Instance = FindAnyObjectByType<T>();
          if (m_Instance == null)
          {
              if (!hasBeenCreated)
                m_Instance = new GameObject("_" + typeof(T), typeof(T)).GetComponent<T>();
          }
          else
          {
            hasBeenCreated = true;
            DontDestroyOnLoad(m_Instance);
            m_Instance.Init();
          }
          return m_Instance;
        }
    }
```

Check this

```
    // If no other monobehaviour request the instance in an awake function
    // executing before this one, no need to search the object.
    private void Awake()
    {
        DontDestroyOnLoad(this);
        if (m_Instance == null)
        {
            m_Instance = this as T;
            hasBeenCreated = true;
            m_Instance.Init();
        }
    }

    // This function is called when the instance is used the first time
    // Put all the initializations you need here, as you would do in Awake
    protected virtual void Init() { }
}
```

Not such a walk in the park ;)