Creational Design Pattern

Hung Tran

Fpt software

September 4, 2021

Outline

Creational Pattern Overview

Construction process of an object.

- **Singleton**: Ensure only one instance.
- **Factory Method**: Create instance without depending on its concrete type.
- **Object pool**: Reuse existing instances.
- **Abstract factory**: Create instances from a specific family.
- **Prototype**: Clone existing objects from a prototype.
- **Builder**: Construct a complex object step by step.

Game Example

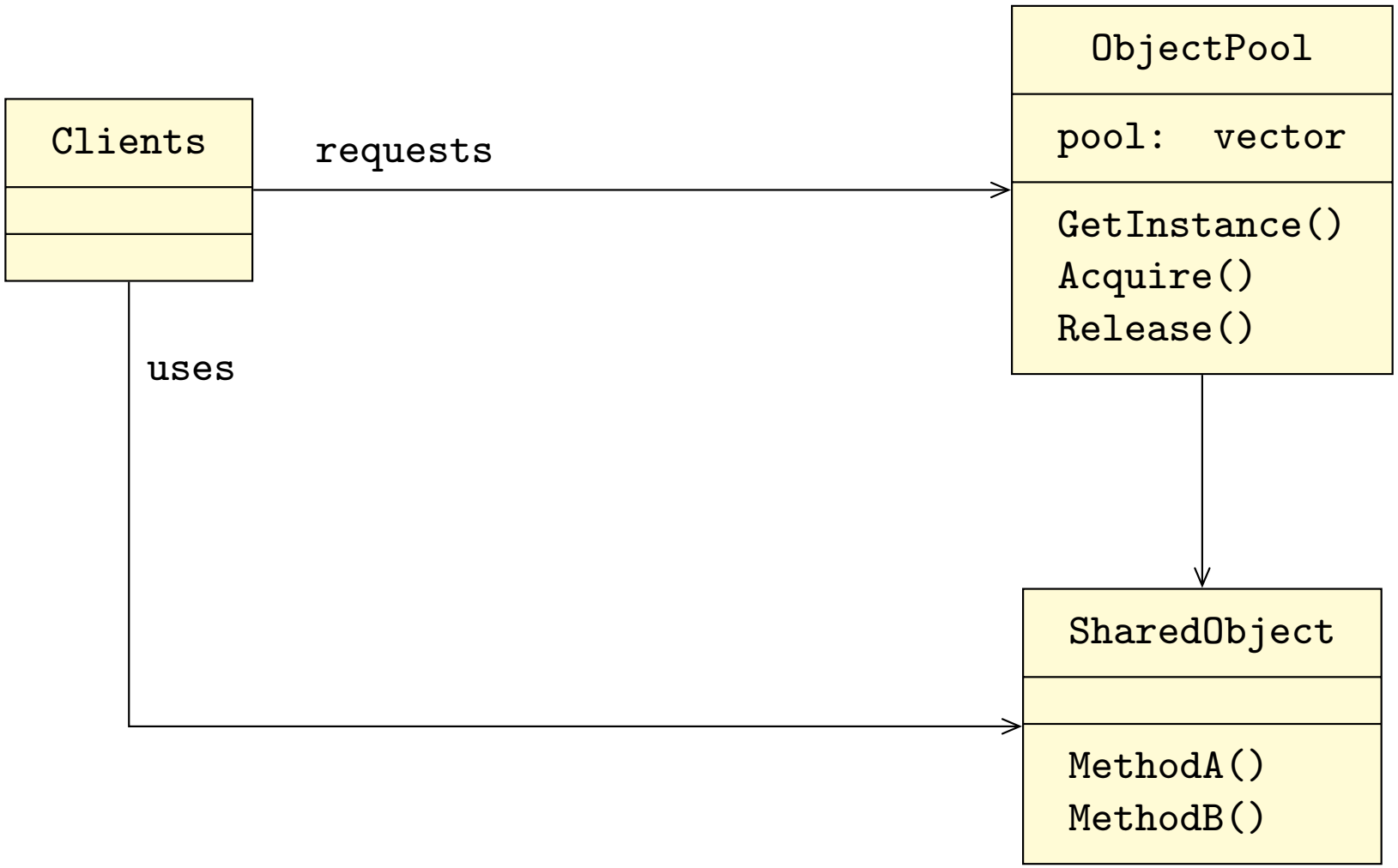
Factory Pattern

- Always create new instance

The Intent of Object Pool Design Pattern

Improve performance and memory use by reusing objects from fixed pool instead of allocating and freeing them repetitively.

Structure of Object Pool Design Pattern



How to Implement of Object Pool Design Pattern?

- *ObjectPool* class maintains an array or a list of *SharedObject* instances
- *SharedObject* instances are not created by Clients; instead they use the *ObjectPool* class
- Objects are constructed when: The program starts, The pool is empty, An existing object is not available.
- For the last case, the pool can be *grow dynamically*.
- ObjectPool should have *only one instance* (Singleton or Monostate)

How to Implement of Object Pool Design Pattern?

- The clients acquire a *SharedObject* instance by invoking a factory method in the pool.
- When the clients gets a *SharedObject* instance, it is either removed from the *ObjectPool* or marked as used.
- The clients may manually return a *SharedObject* to the *ObjectPool* or t may be done automatically.
- The instance can be used again.

How to Implement of Object Pool Design Pattern?

- The Pool object instance can be reset: before giving it to the client or after it is returned to the pool.
- The ObjectPool is responsible for deleting the pooled instances
- These instances are usually deleted at the end of the program.
- To avoid tight coupling with concreate pooled objects, ObjectPool can use a factory to instantiate them.

Basic Implementation

```
#ifndef SHARED_OBJECT_H
#define SHARED_OBJECT_H

class SharedObject {
  bool m_IsUsed{true};
public:
  bool IsUsed() const {
    return m_IsUsed;}
  void SetUsedState(bool
    used) {m_IsUsed = used
    ;}
  void Reset();
  void MethodA();
  void MethodB();
  ~SharedObject();
};
#endif
```

```
#include "SharedObject.h"
#include <iostream>

void SharedObject::Reset()
    {
  std::cout << "Resetting
    the state\n";
}

void SharedObject::MethodA
    () {
  std::cout << "MethodA\n";
}

void SharedObject::MethodB
    () {
  std::cout << "MethodB\n";
}

```

Basic Implementation

```
#ifndef OBJECT_POOL_H
#define OBJECT_POOL_H
#include <vector>
#include <memory>
class SharedObject;

class ObjectPool {
  ObjectPool() = default;
  inline static std::vector
    <std::shared_ptr<
    SharedObject>>
    m_PooledObjects{};
public:
  static std::shared_ptr<
    SharedObject>
    AcquireObject();
  static void ReleaseObject
    (std::shared_ptr<
    SharedObject> pSO);
```

```
#include "ObjectPool.h"
#include <iostream>
#include "SharedObject.h"

std::shared_ptr<
    SharedObject>
    ObjectPool::
    AcquireObject() {
  for (auto &so :
    m_PooledObjects) {
    if(!so->IsUsed()) {
      std::cout << "[POOL]
    returning an existing
    object\n";
      so->SetUsedState(true
    );
      so->Reset();
      return so;
    }
```

Basic Implementation

```
#include "ObjectPool.h"
#include "SharedObject.h"

int main() {
  auto s1 = ObjectPool::AcquireObject();
  s1->MethodA();
  s1->MethodB();

  auto s2 = ObjectPool::AcquireObject();
  s2->MethodA();
  s2->MethodB();

  ObjectPool::ReleaseObject(s1);
  auto s3 = ObjectPool::AcquireObject();
  s3->MethodA();
  s3->MethodB();
}
```

Basic Implementation

- ObjectPool class is monostate (or singleton)
- Put a flag in SharedObject to know used or not
- In AcquireObject(): check if an existing object is available in the pool and used or not?
- In ReleaseObject(): compare object with the pool and change the flag

Game Example

Game Example with ObjectPool

Multiple Objects in ObjectPool

ObjectPool with Factory Method

Generic ObjectPool

Pros and Cons

Pros

- Reduce coupling with concrete classes
- Behave like operator new, but more flexible
- Caching existing instances improves performance of the app
- Reduce the overhead of heap allocation and deallocation
- Reduce heap fragmentation

Cons

- Memory may be wasted on unused pooled objects
- Pooled objects may remain in memory until the end of program
- Objects that are acquired from the pool must be reset prior to their use
- Clients have to ensure that an unused object is returned to the pool
- ObjectPool class may get

Where to use?

- Frequently create and destroy objects
- Allocating heap objects is slow
- Objects are expensive to create