Robotics Corner

Robotics Corner

Design Patterns

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design

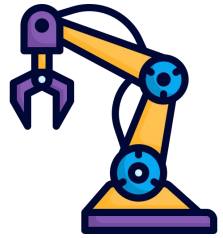
01 Proxy

02 Singleton

03 Adaptor

ROBOTICS CORNER

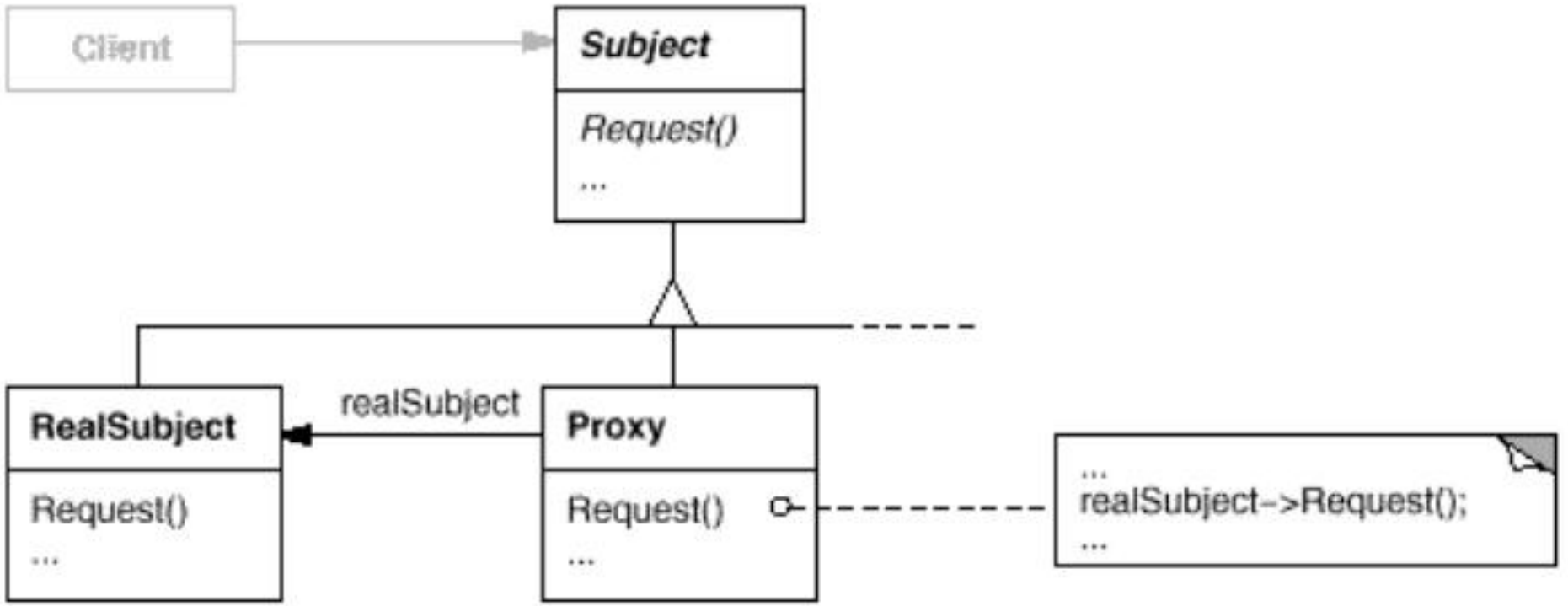
01 Proxy



A Proxy provides an interface that forwards function calls to another interface of the same form. A Proxy pattern is useful to modify the behavior of the RealSubject class while still preserving its interface. This is particularly useful if the RealSubject class is in third-party library and hence not easily modifiable directly.

ROBOTICS CORNER

UML for Proxy Pattern



Use case 1

- Implement lazy instantiation of the RealSubject object
- In this case, the RealSubject object is not actually instantiated until a method call is invoked. This can be useful if instantiating the RealSubject object is a heavyweight operation that we wish to defer until absolutely necessary.

Use case 2

- Implement access control to the RealSubject object
- We may want to insert a permissions layer between the Proxy and the RealSubject objects to ensure that users can only call certain methods on the RealSubject object if they have appropriate permission.

Use case 3

- Support debug or dry-run modes
- This lets us insert debugging statement into the Proxy methods to log all call to the RealSubject object or we can stop the forwarding to certain RealSubject method with a flag to let us call the Proxy in a dry-run mode, such as to turn off writing the object's state to disk

Use Case 4

- Make the RealSubject class to be thread safe
- This can be done by adding mutex locking to the methods that are not thread safe. While this may not be the most efficient way to make the underlying class thread safe, it is a useful if we cannot modify the RealSubject.

Use case 5

- Share resources
- We could have multiple Proxy objects share the same underlying RealSubject class. This could be used to implement reference counting, for instance. This is, actually, another design pattern called the Flyweight pattern, where multiple objects share the same underlying data to minimize memory.

Use case 6

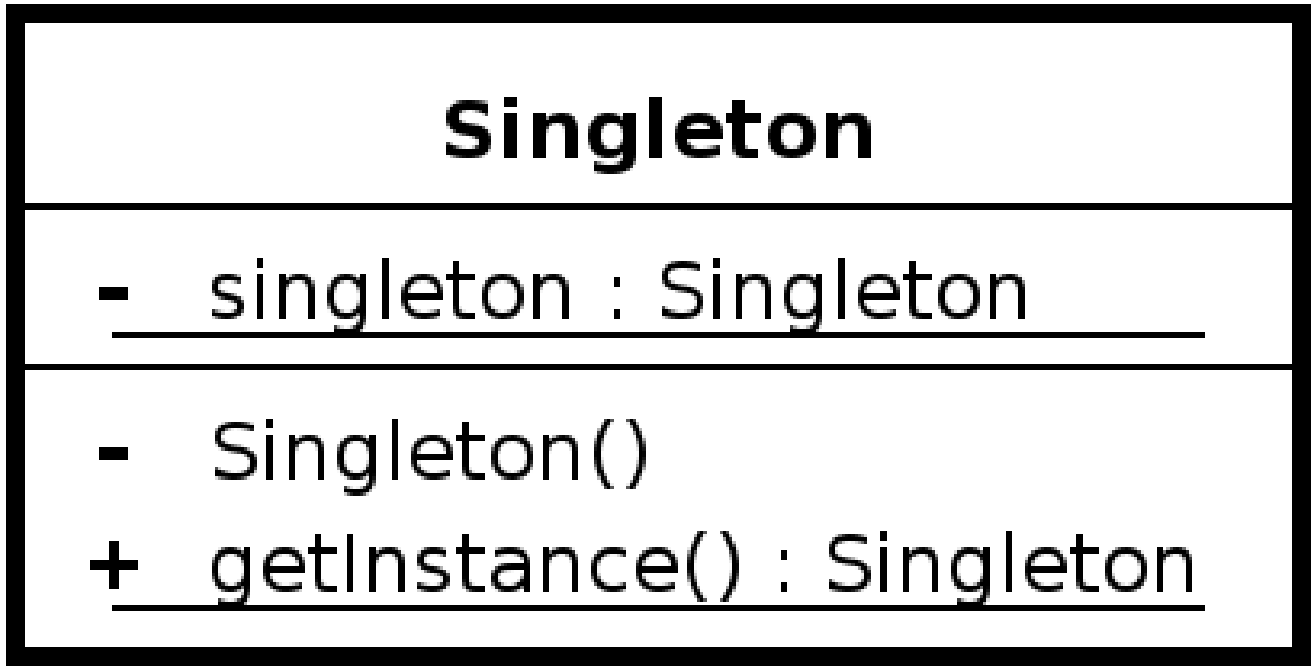
- Protect against future changes in the RealSubject class
- We anticipate that a dependent library will change in the future so we create a proxy wrapper around that API that directly mimics the current behavior. So, when the library changes later, we can preserve the old interface via our proxy object and simply change its underlying implementation to use the new library methods.

02 Singleton

- The Singleton pattern ensures a class has only one instance, and provides a global point of access to it.

- A Singleton is an elegant way of maintaining global state, but we should always question whether we need global state

Class Diagram



- Singleton pattern offers several advantages over global variables because it does the following:

- Enforces that only one instance of the class can be instantiated.
- Allows control over the allocation and destruction of the object.
- Provides thread-safe access to the object's global state.
- Prevents the global namespace from being polluting.

Singleton in Multi-threading

- The revised version is not thread safe because there could be a race condition during the initialization of the static Singleton, Singleton& Singleton::getInstance().

Thread safe Singleton

```
Singleton& Singleton::getInstance()
{
        Mutex mutex;
        ScopedLock(&mutex;);  // to unlock mutex on exit
         static Singleton instance;
         return instance;
}
```

Singleton : pattern vs Anti-pattern

- In 2009, the authors of the original design patterns said the only pattern they would consider removing from the original list is Singleton. This is because it is essentially a way to store global data and tends to be an indicator of poor design.

Alternatives

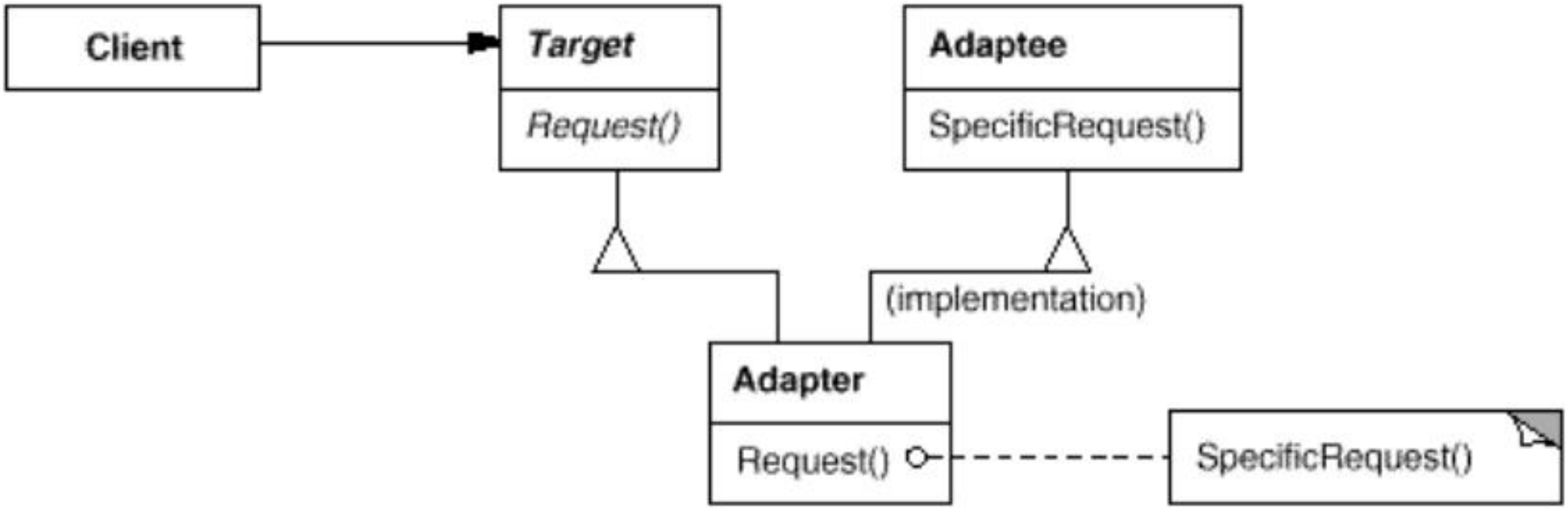
- There are several alternatives to the Singleton pattern:

- dependency injection
- Monostate pattern
- session context

03 Adaptor

- Convert the interface of a class into another interface clients expect. Adapter (or Wrapper) lets classes work together that couldn't otherwise because of incompatible interfaces. Adapter pattern's motivation is that we can reuse existing software if we can modify the interface.

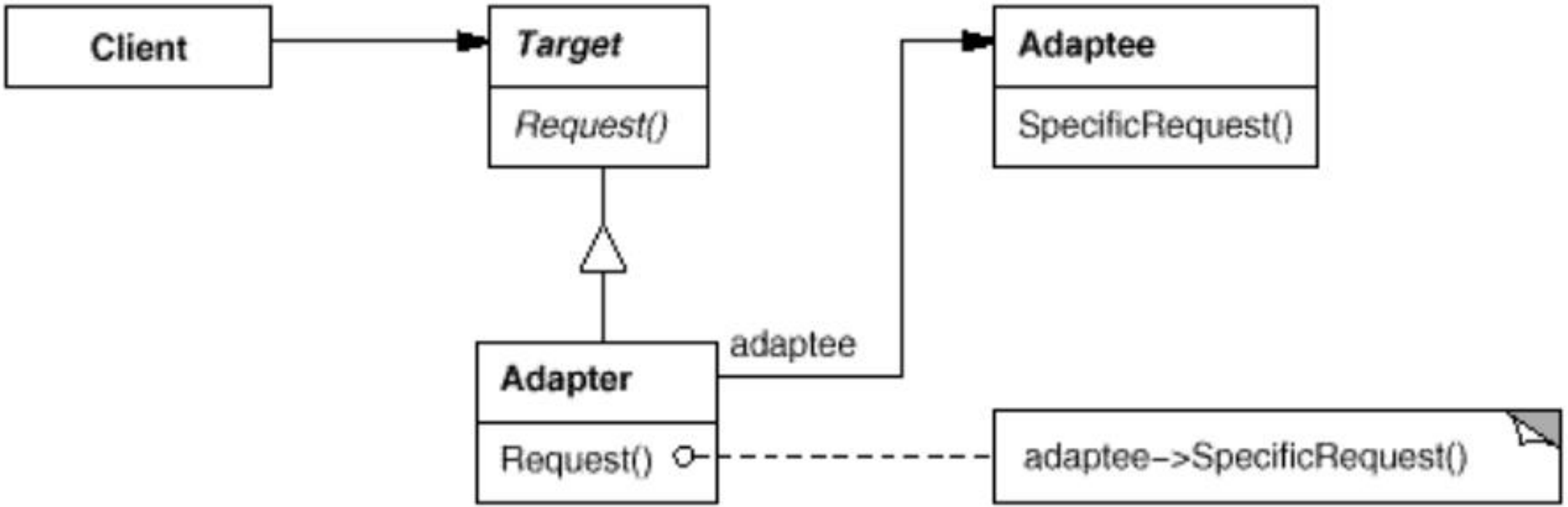
UML for Adaptor



How it works

- Adapter pattern relies on object composition.
- Client calls operation on Adapter object.
- Adapter calls Adaptee to carry out the operation.
- In STL, stack adapted from vector:
- When stack executes push(), underlying vector does vector::push_back().

example


Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()
adaptee->SpecificRequest()

Summary

- The client thinks he is talking to a Rectangle
- The target is the Rectangle class. This is what the client invokes method on.

```
Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw()
```

Code explanation

- Note that the adapter class uses multiple inheritance.
- class RectangleAdapter: public Rectangle, private LegacyRectangle {
- …
- }
- The Adapter RectangleAdapter lets the LegacyRectangle responds to request (draw() on a Rectangle) by inheriting BOTH classes.

Code Exaplanation 2

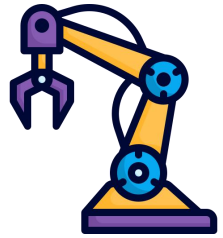
- The LegacyRectangle class does not have the same methods (draw()) as Rectangle,
- but the Adapter(RectangleAdapter) can take the Rectangle method calls and turn around and invoke method on the LegacyRectangle, oldDraw()

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
                    std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
          }

    void draw() {
       std::cout << "RectangleAdapter: draw().\n";
       oldDraw();
    }
```

Conclusion

- Adapter design pattern translates the interface for one class into a compatible but different interface. So, this is similar to the proxy pattern in that it's a single-component wrapper. But the interface for the adapter class and the original class may be different.



Thank You

Do you have any questions?

ROBOTICS CORNER

ROBOTICS CORNER

01211626904

www.roboticscorner.tech

Robotics Corner

Robotics Corner

Robotics Corner

Robotics Corner