50002 - Software Engineering Design - Lecture 13

Oliver Killane

22/11/21

Lecture Recording

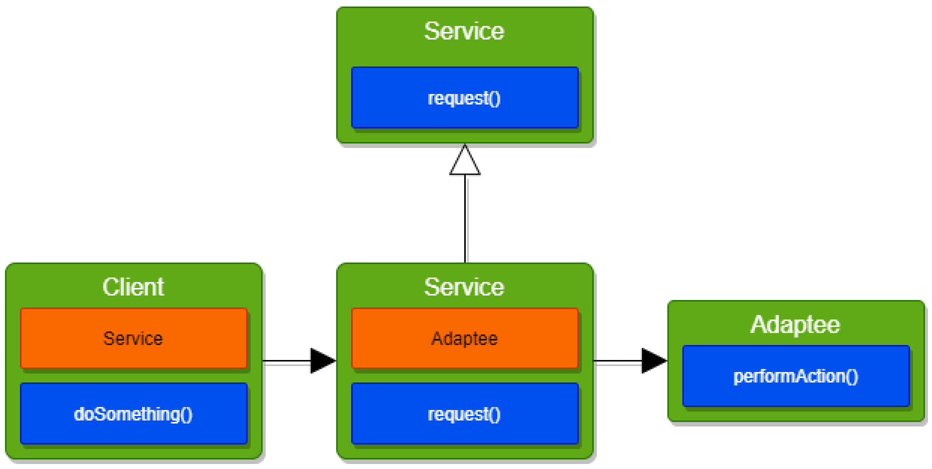
Lecture recording is available here

Adapter

I have X, but I want Y

Converts the interfgace of a class into another interface that the client expects.

Used when an existing component does what we require, but does not have the correct interface to use in our system. We may use our adapter to call multiple methods in the Adaptee, but almost all the behaviour should be in the adaptee.



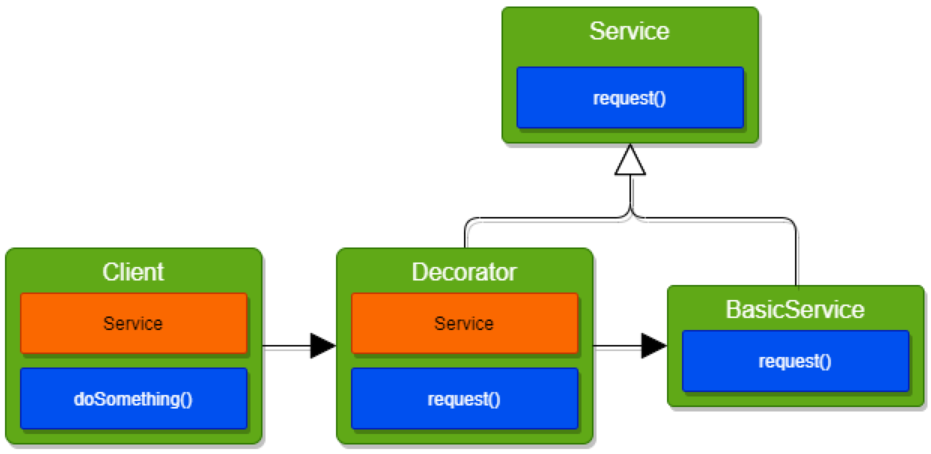
```java
public interface Service {
    void request();
}

public class Adaptee {

    public void performAction() {
        ...
    }
}

public class Adapter implements Service {
    Adaptee adaptee = new Adaptee();

    public void request() {
        adaptee.performAction();
    }
}

public class Client {
    Service adapter = new Adapter();

    private void doSomething() {
        ...
        adapter.request()
        ...
    }
    ...
}
```

Decorator

I have X, but I want a better X

Add more functionality or responsibility to an object dynamically.

We use an existing basic service, and create a new service that uses the basic service.

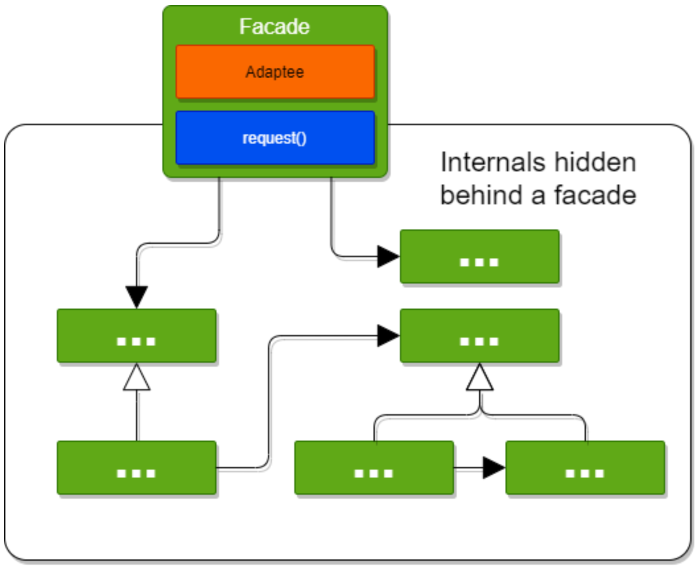


```
public interface Service {
    void request();
}

public class BasicService implements Service {

    public void request() {
        ...
    }
}

public class Decorator implements Service {
    Service basic = new BasicService();

    public void request() {
        ...
        basic.request();
        ...
    }
}

public class Client {
    Service decor = new Decorator();

    private void doSomething() {
        ...
        decor.request()
        ...
    }
    ...
}
```

Facade

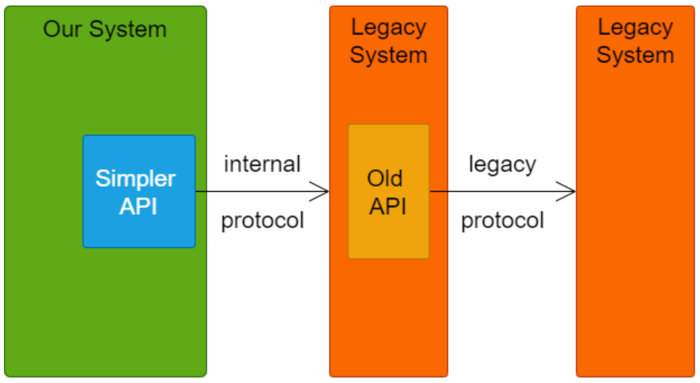
I have X, but I want a simpler X

The facade hides the internal complexity, hading the use of many objects & interfaces behind one simple interface.



Simplicator

A common use case for this is the **Simplicator** for interacting with legacy networking systems.



This can make it easier to test as we can mock our Simpler API for testing the system, and use our simpler api alone to create tests for checking our legacy system.

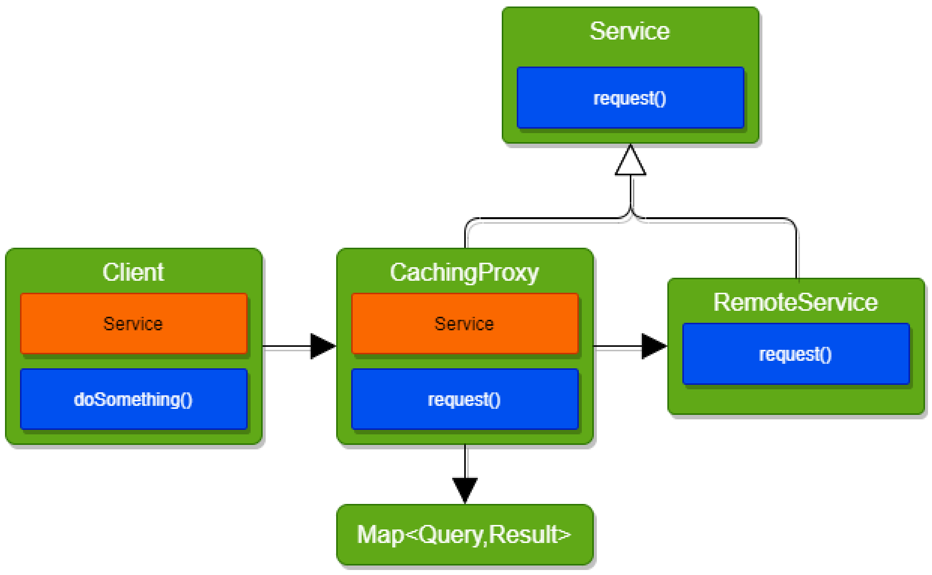
Proxy

I have X, but its too slow

Control access to an object by providing a placeholder or surrogate object. Is put in front of an expensive resource to manage it (e.g prioritisation, load balancing, caching), delegates to that resource.

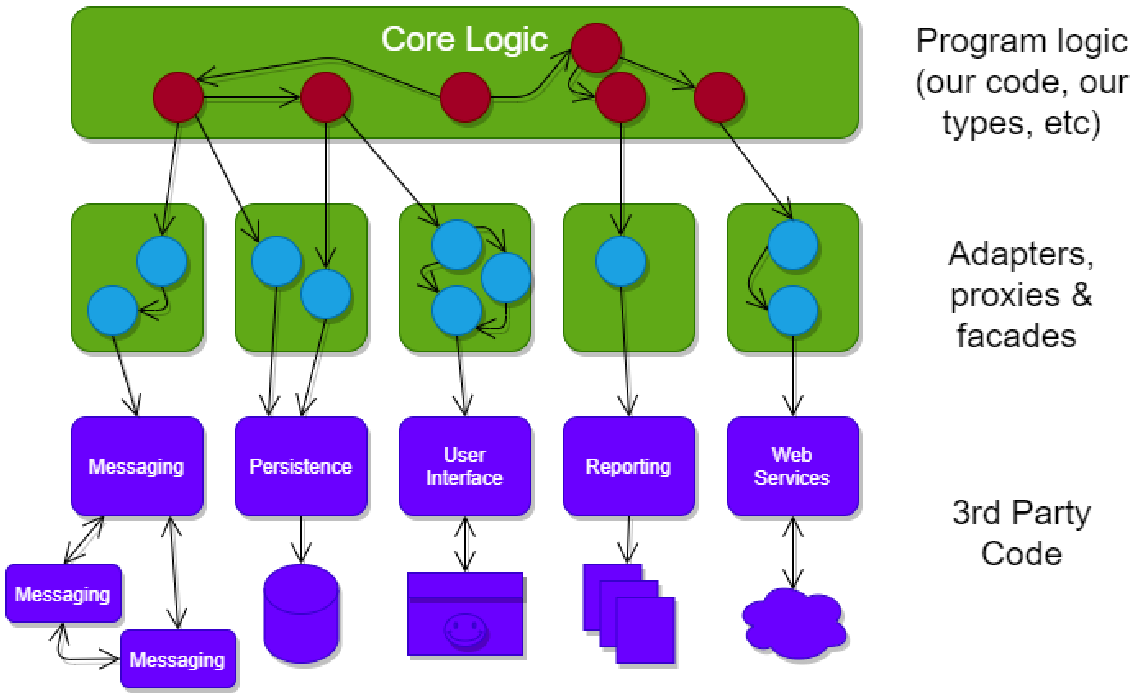
Proxy may (or may not) decide to push a request on to the service, or may hold onto the result (caching).

Caching



Caching can be very effective when combined with a simplificator. We can use the simplificator to integrate off-the-shelf caching software and have it work with our legacy system.

Hexagonal Architecture



By using adapters to bridge our code for the program logic, and 3rd party services we can:

- **Unit Test the Logic**
  As we can mock the adapters, we can easily test the program logic without having to spin up any 3rd party services.
- **Perform Integration Tests**
  Test the adapters in isolation from the program logic.
- **Perform System Tests**
  Can test all components together, though this is more complex, slow and unlikely to cover all behaviour.
- **Decouple 3rd party services and program logic**
  We can alter program logic easily without dealing with types & conventions from our services. Furthermore we can easily change service by only changing the adapter.

Test Proportions

A common ratio for *unit* : *integration* : *system* tests is 70% : 20% : 10% though this varies by speed of tests & how focused/wide ranging the tests are.