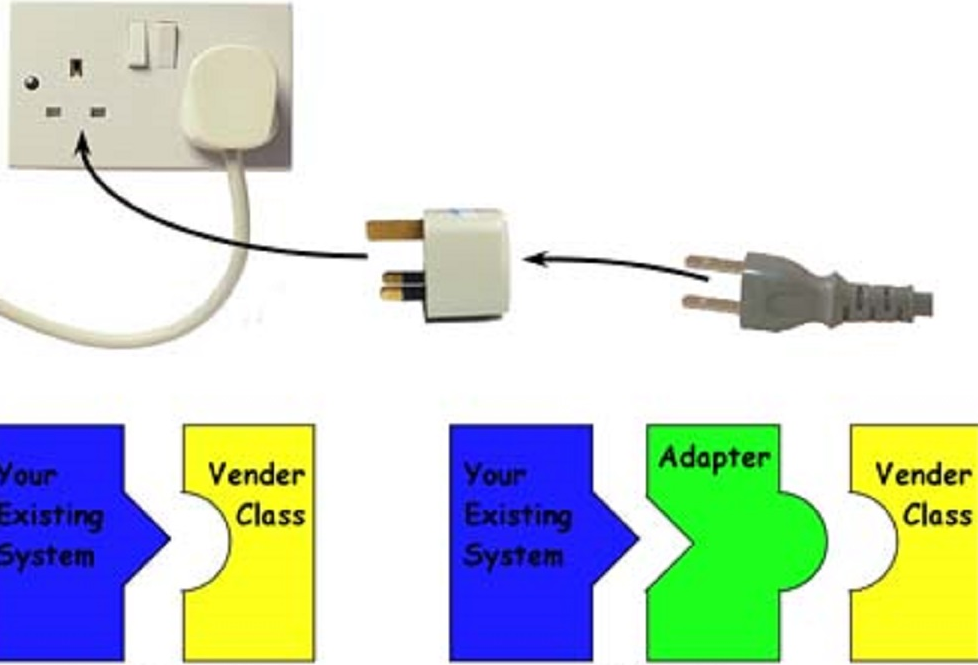
***Adapter Pattern: Match interfaces of different classes***

**Definition:** The Adapter Pattern converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

This pattern is easy to understand as the real world is full of adapters.

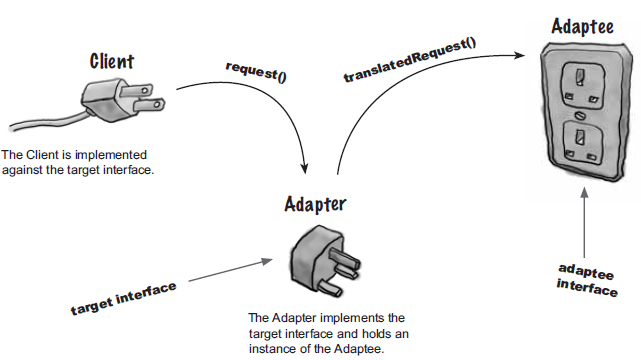
The adapter pattern is adapting between classes and objects. Like any adapter in the real world it is used to be an interface, a bridge between two objects. In real world we have adapters for power supplies, adapters for camera memory cards, and so on. Probably everyone has seen some adapters for memory cards. If you cannot plug in the camera memory in your laptop you can use and adapter. You plug the camera memory in the adapter and the adapter in to laptop slot. That's it, it's really simple.

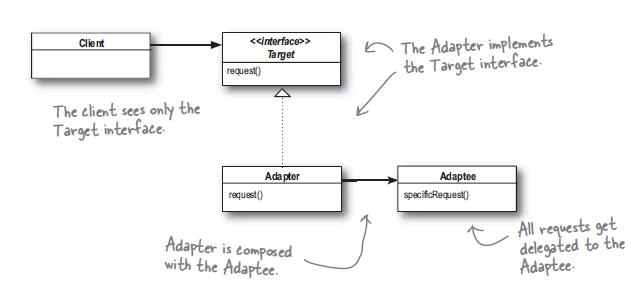


In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adoptee) offering the same features but exposing a different interface.

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receives the results of the call and is unaware of adapter’s presence.





The diagram shown above represents the generic class diagram of the Adapter Pattern. We start with the Client object. It is directly associated with the Target interface but expects the behavior of the **Adaptee** object. The problem is our **Adaptee** object does not implement the Target interface the Client has access to. Enter the Adapter class. It will implement the Target interface and delegate the behavior using composition to the **Adaptee** object. So in this diagram, when the Client calls **Request**() via the Target interface, the Adapter makes the **SpecificRequest**() to the **Adaptee**. This is considered an object adapter.

Check list

1. Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).
2. Identify the interface that the client requires.
3. Design a "wrapper" class that can "impedance match" the adaptee to the client.
4. The adapter/wrapper class "has a" instance of the adaptee class.
5. The adapter/wrapper class "maps" the client interface to the adaptee interface.
6. The client uses (is coupled to) the new interface

### Adapter Pattern Example

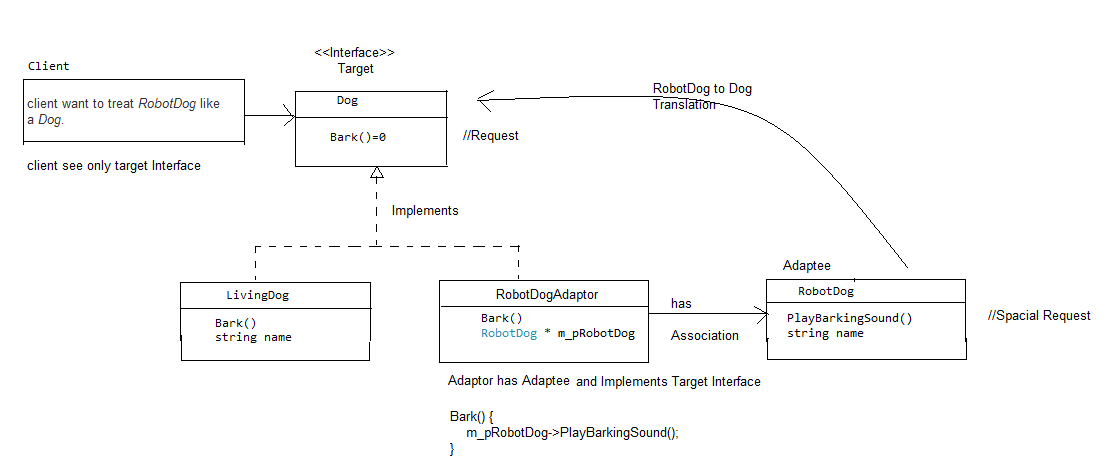
Consider a **Dog** interface that has an abstract method **Bark**(). Say, for the moment, there is only one subclass, **LivingDog**. It implements **Bark**().

Additionally, we have a class called **RobotDog** that implements a **PlayBarkingSound**() method. What if we have some new work that requires a client to treat **RobotDog** like a Dog?

We could have **RobotDog** inherit from **Dog** but the existing interfaces don’t quite line up and we would rather not modify the existing **RobotDog** class **(open/closed principle).**

Here is where the Adapter pattern comes in. Our Dog interface would be considered our Target and the **RobotDog** class would be our **Adaptee**. So we would need an **Adapter** class to implement the Dog interface and delegate the behavior we want to the **RobotDog** class.

Let’s call our Adapter class the **RobotDogAdapter**. We decide to implement the behavior of **Bark**() to just call **RobotDog’s** **PlayBarkingSound**() method. Now since we are delegating the behavior through composition this is an example of an object adapter.



class Dog

{

public:

virtual ~Dog() {}

virtual void Bark() = 0;

};

class LivingDog : public Dog {

public:

LivingDog(std::string name): m\_name(name){}

virtual ~LivingDog();

virtual void Bark(){

std::cout << m\_name << " is barking!\n";

}

private:

std::string m\_name;

};

class RobotDog {

public:

RobotDog(std::string name):m\_name(name){}

virtual ~RobotDog();

void PlayBarkingSound(){

std::cout << m\_name << " is playing barking sound\n";

}

private:

std::string m\_name;

};

class RobotDogAdapter : public Dog {

public:

RobotDogAdapter(RobotDog \* robotDog) : m\_pRobotDog(robotDog){}

virtual ~RobotDogAdapter();

virtual void Bark(){

m\_pRobotDog->PlayBarkingSound();

}

private:

RobotDog \* m\_pRobotDog;

};

int main()

{

Dog \* livingDog = new LivingDog(std::string("Chester"));

livingDog->Bark();

RobotDog \* robotDog = new RobotDog(std::string("Sony's Aibo"));

Dog \* robotDogAdapter = new RobotDogAdapter(robotDog);

robotDogAdapter->Bark();

delete livingDog;

delete robotDogAdapter;

delete robotDog;

return 0;

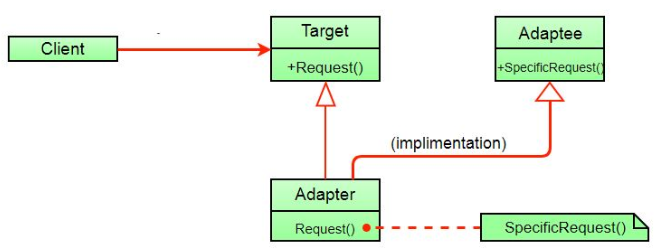
}

### Class Adaptor

There is another version of the Adapter pattern called a class adapter where the Adapter class inherits from the Adaptee class.

The object adapter is the preferred adapter due to it supporting a common design principle to prefer composition over inheritance. Additionally, some languages don’t allow multiple inheritance (i.e. Java) so the object adapter is the only option.

The alternative would be a class adapter pattern that would have the RobotDogAdapter privately inherit from RobotDog. The object adapter pattern is preferred by most so we will just stick with it as our example. Below is the class diagram of our object adapter



//Adaptor. C++ allows Multiple inheritance.

class RobotDogAdapter : public Dog,public RobotDog {

public:

RobotDogAdapter(RobotDog \* robotDog) : m\_pRobotDog(robotDog) {}

virtual ~RobotDogAdapter(){}

virtual void Bark() {

m\_pRobotDog->PlayBarkingSound();

}

private:

RobotDog \* m\_pRobotDog;

};

//Client, Rest Implementation is same as Class Adaptor. Hear for the simplicity, I have //removed LivingDog class, if we want we can add it.

int main() {

RobotDog \* robotDog = new RobotDog(std::string("Sony's Aibo"));

Dog \* robotDogAdapter = new RobotDogAdapter(robotDog);

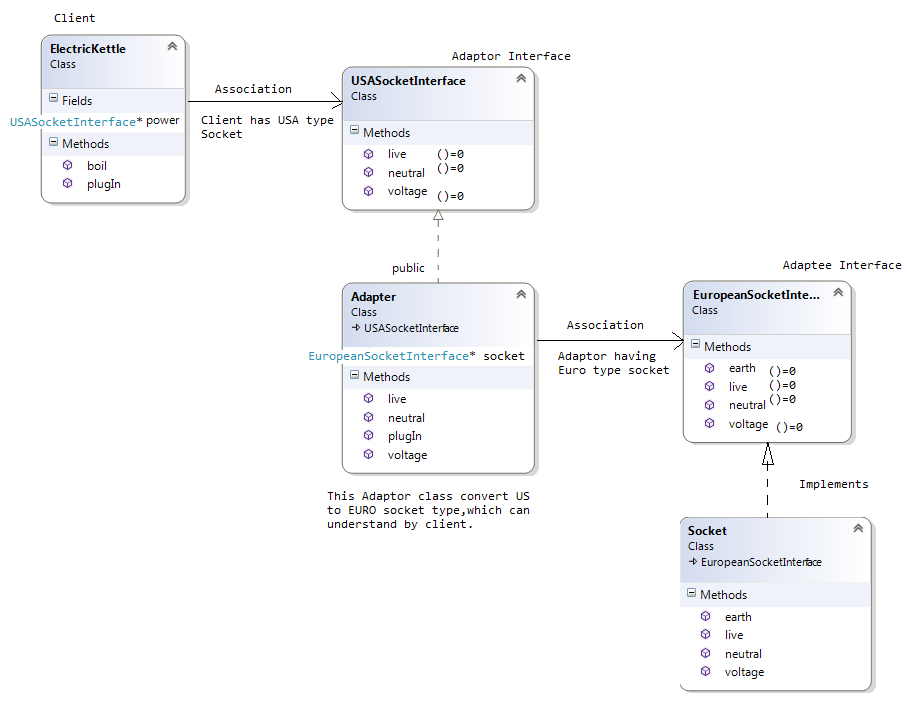
robotDogAdapter->Bark();

delete robotDogAdapter;

delete robotDog;

}

### Adaptor kettle Example:



typedef int Cable; // wire with electrons :-)

class EuropeanSocketInterface { /\* Adaptee (source) interface \*/

public:

virtual int voltage() = 0;

virtual Cable live() = 0;

virtual Cable neutral() = 0;

virtual Cable earth() = 0;

};

/\* Adaptee \*/

class Socket : public EuropeanSocketInterface

{

public:

int voltage() { return 230; }

Cable live() { return 1; }

Cable neutral() { return -1; }

Cable earth() { return 0; }

};

/\* Target interface \*/

class USASocketInterface

{

public:

virtual int voltage() = 0;

virtual Cable live() = 0;

virtual Cable neutral() = 0;

};

/\* The Adapter \*/

class Adapter : public USASocketInterface

{

EuropeanSocketInterface\* socket;

public:

void plugIn(EuropeanSocketInterface\* outlet)

{

socket = outlet;

}

int voltage() { return 110; }

Cable live() { return socket->live(); }

Cable neutral() { return socket->neutral(); }

};

/\* Client \*/

class ElectricKettle

{

USASocketInterface\* power;

public:

void plugIn(USASocketInterface\* supply)

{

power = supply;

}

void boil()

{

if (power->voltage() > 110)

{

std::cout << "Kettle is on fire!" << std::endl;

return;

}

if (power->live() == 1 && power->neutral() == -1)

{

std::cout << "Coffee time!" << std::endl;

}

}

};

int main()

{

Old existing code. Socket(European style) support three cable- live, neutral and

earth. Also support constant voltage supply 220v.

Socket\* socket = new Socket;

/\*

Electric Kettle is our client. It has below requirement -

-It runs on only 110v.

-It doesn’t require earthing cable i.e. it required US style Socket.

Now, this is a problem, we can’t plug-in Electric Kettle into socket directly.

Due to high voltage & addition cabling.

\*/

ElectricKettle\* kettle = new ElectricKettle;

/\*

Move to the solution:

Client need access the existing code, but interface is not compatible.

-It doesn’t require the earthing connection.

-Required the voltage only 110v.

We need something which adjust the voltage & disable/remove the earthing

cable. Adopter has a socket, but before plug-in its responsibility to adjust

the voltage and remove the earthing cable.

In nutshell we can’t say that, this is provide the interface which makes our

Client kettle (new interface) compatible with socket (Old system/existing

code)

\*/

Adapter\* adapter = new Adapter;

/\*

Plugging in Process:

First Adapter must be plug-in into socket

Now adapter provide only 110v & remove earthing cable, which is required by our

Client.

\*/

adapter->plugIn(socket);

/\*

Second, our adaptor is quite familiar with Kettle; we can directly plug-in

kettle into it without any hazard.

\*/

kettle->plugIn(adapter);

/\* And then, have a coffee \*/

kettle->boil();

getchar();

return 0;

}

Benefits of the Adapter Pattern

So now that we have seen the Adapter pattern in action, let’s go over the benefits of using the pattern.

1. In the object adapter pattern, the Adapter class can be used with any subclass of the Adaptee. However, it can be difficult to override Adaptee behavior as you may need to have the Adapter class use subclasses of the Adaptee versus the Adaptee itself
2. The client is decoupled from the incompatible interface and not affected by changes to that interface because the adapter class encapsulates the incompatible interface.
3. Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

Disadvantage

1. All requests are forwarded, so there is a slight increase in the overhead.
2. Sometimes many adaptations are required along an adapter chain to reach the type which is required.

Related Pattern

* Adapter makes things work after they're designed; Bridge makes them work before they are.
* Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.
* Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
* Adapter is meant to change the interface of an existing object. Decorator enhances another object without changing its interface. Decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.
* Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.
* Adapter Pattern and Strategy Pattern - there are many cases when the adapter can play the role of the Strategy Pattern. If we have several modules implementing the same functionality and we wrote adapters for them, the adapters are implementing the same interface. We can simply replace the adapters objects at run time because they implements the same interface.