Structural Design Patterns

Instructor: Mehroze Khan

Structural Design Patterns

- Structural design patterns explain how to **assemble objects and classes** into larger structures, while keeping these structures flexible and efficient.
- Structural class patterns use inheritance to compose interfaces or implementations.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

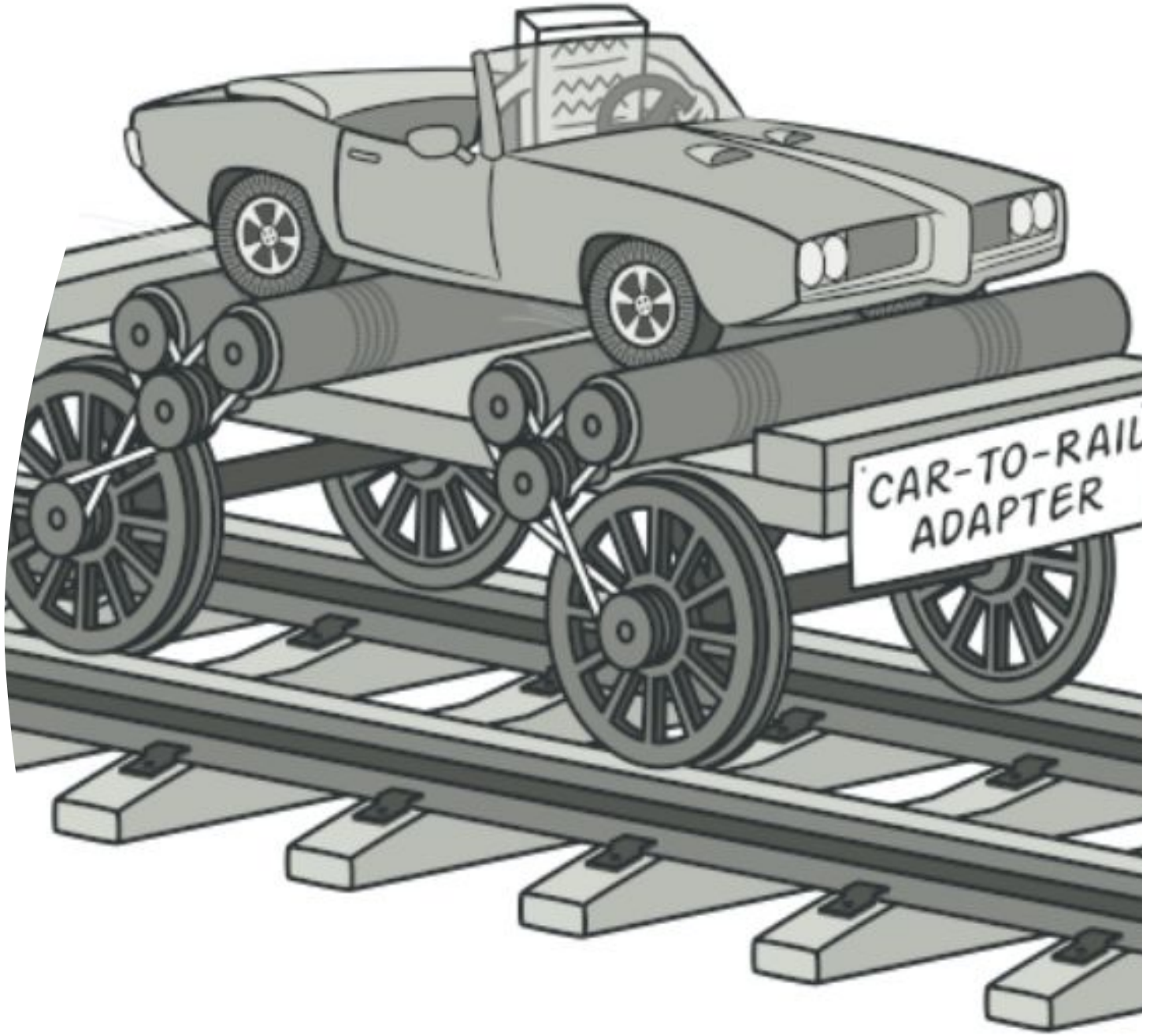
Adapter Pattern

Adapter Design Pattern

Also Known as: **Wrapper**

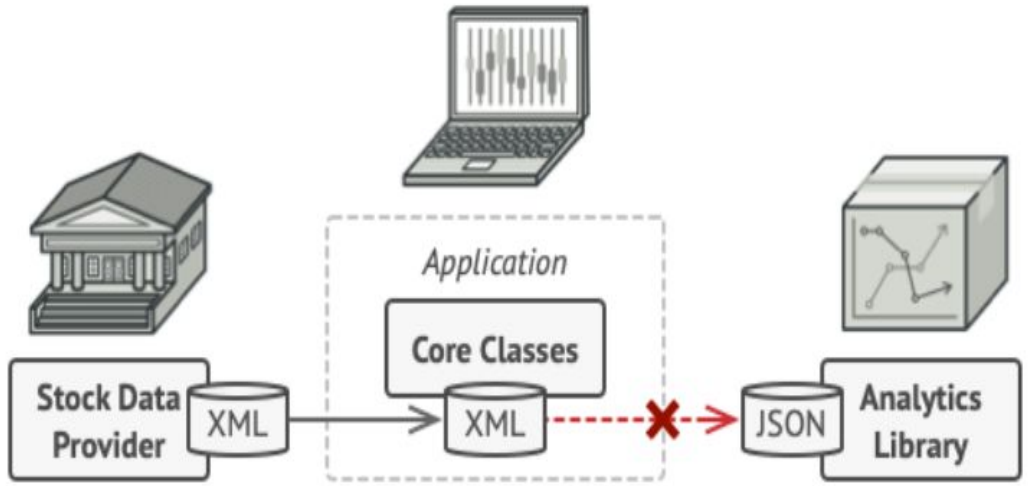
Intent:

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.



Problem

- Imagine that you're creating a **stock market monitoring app**. The app downloads the stock data from multiple sources in **XML** format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in **JSON** format.
- You could change the library to work with XML. However, this might break some existing code that relies on the library. And worse, you might not have access to the library's source code in the first place, making this approach impossible.



You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

Solution

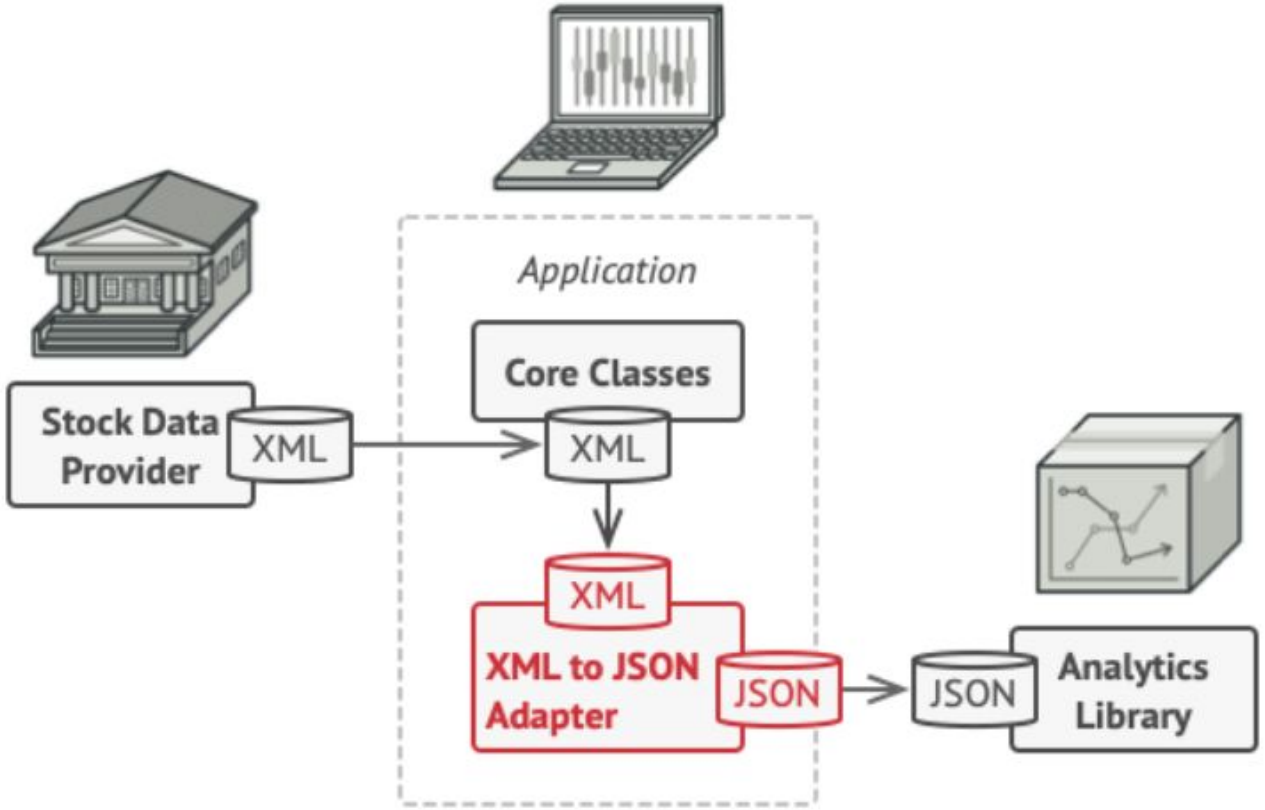
- You can create an ***adapter***. This is a special object that converts the interface of one object so that another object can understand it.
- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes.
- The wrapped object isn't even aware of the adapter.
- For example, you can wrap an object that operates in **meters** and **kilometers** with an adapter that converts all the data to imperial units such as **feet** and **miles**.

Solution

- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate.
- Here's how it works:
  1. The adapter gets an interface, compatible with one of the existing objects.
  2. Using this interface, the existing object can safely call the adapter's methods.
  3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.
- Sometimes it's even possible to create a two-way adapter that can convert the calls in both directions.

Solution

- To solve the dilemma of incompatible formats, you can create **XML-to-JSON adapters** for every class of the analytics library that your code works with directly.
- Then you adjust your code to communicate with the library only via these adapters.
- When an adapter receives a call, it translates the incoming XML data into a JSON structure and passes the call to the appropriate methods of a wrapped analytics object.



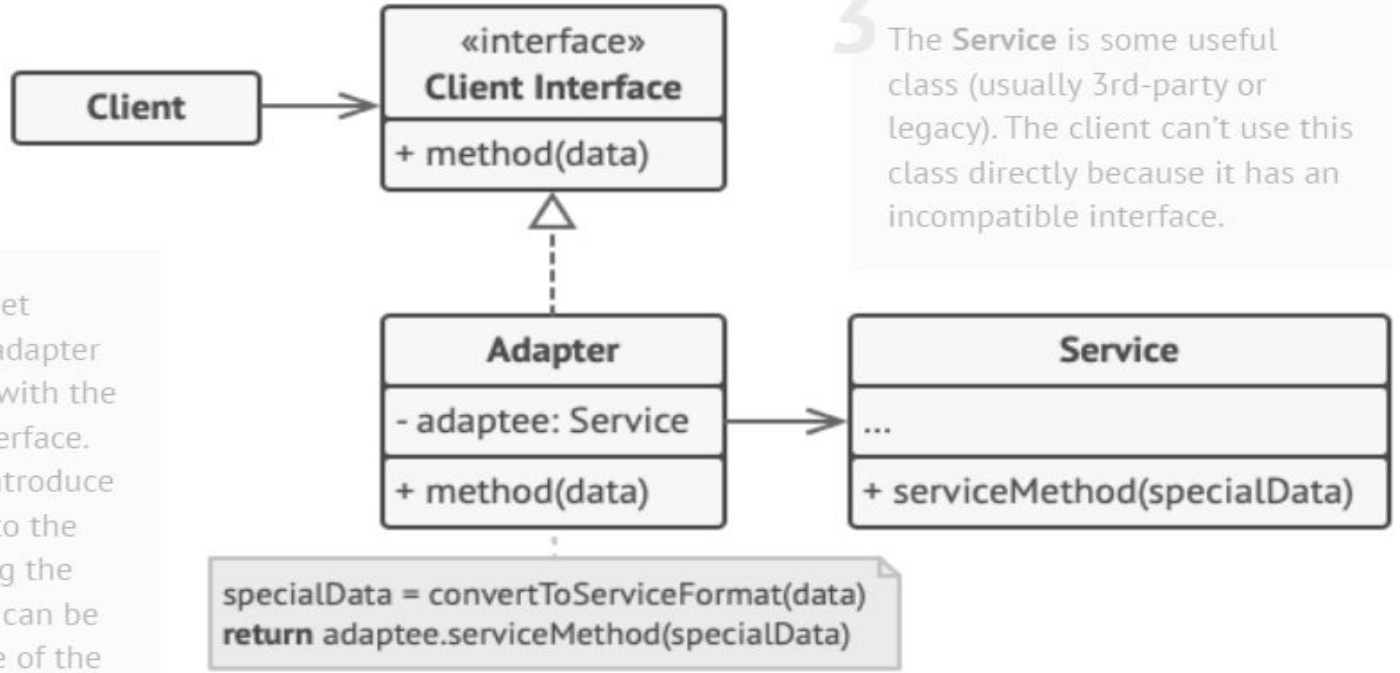
Structure (Object Adapter)

- This implementation uses the **object composition principle**:
  - Adapter implements the interface of one object and wraps the other one. It can be implemented in all popular programming languages.

Structure (Object Adapter)

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

1 The **Client** is a class that contains the existing business logic of the program.



3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

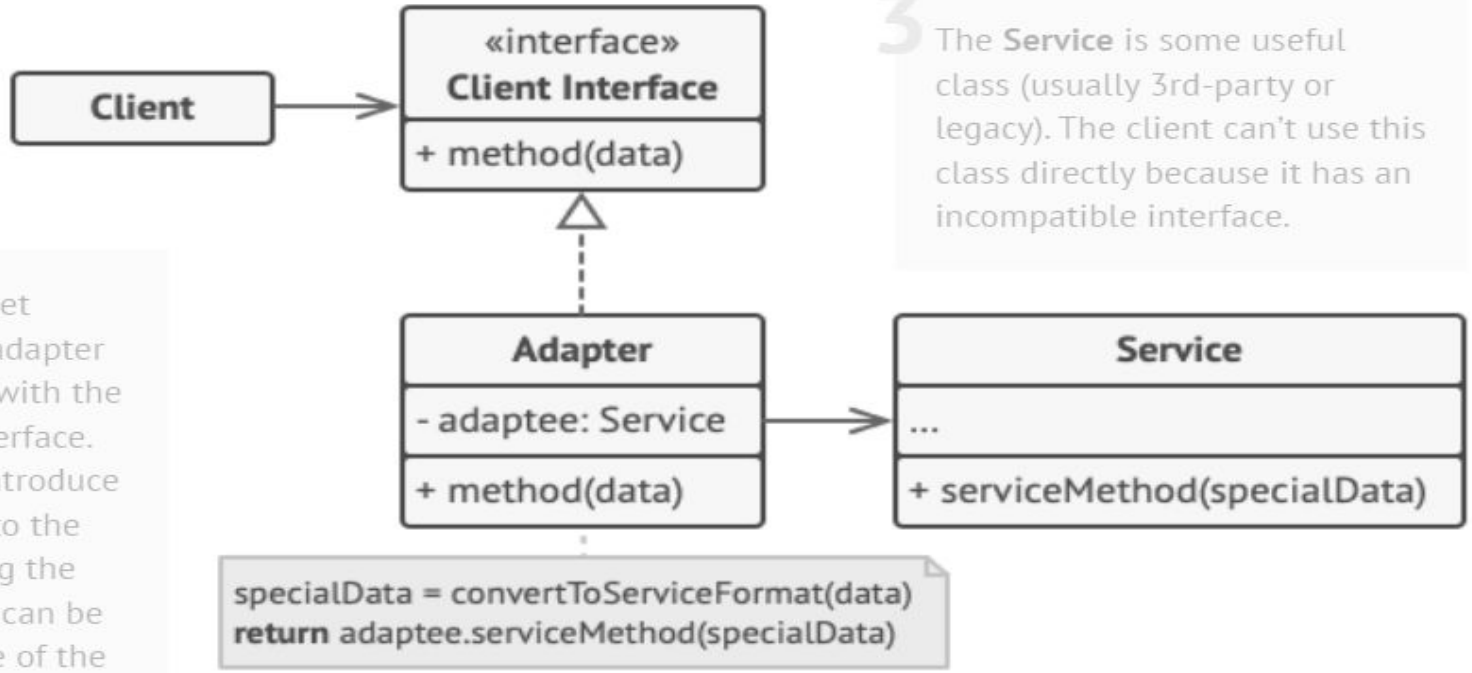
5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

Structure (Object Adapter)

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

1 The **Client** is a class that contains the existing business logic of the program.

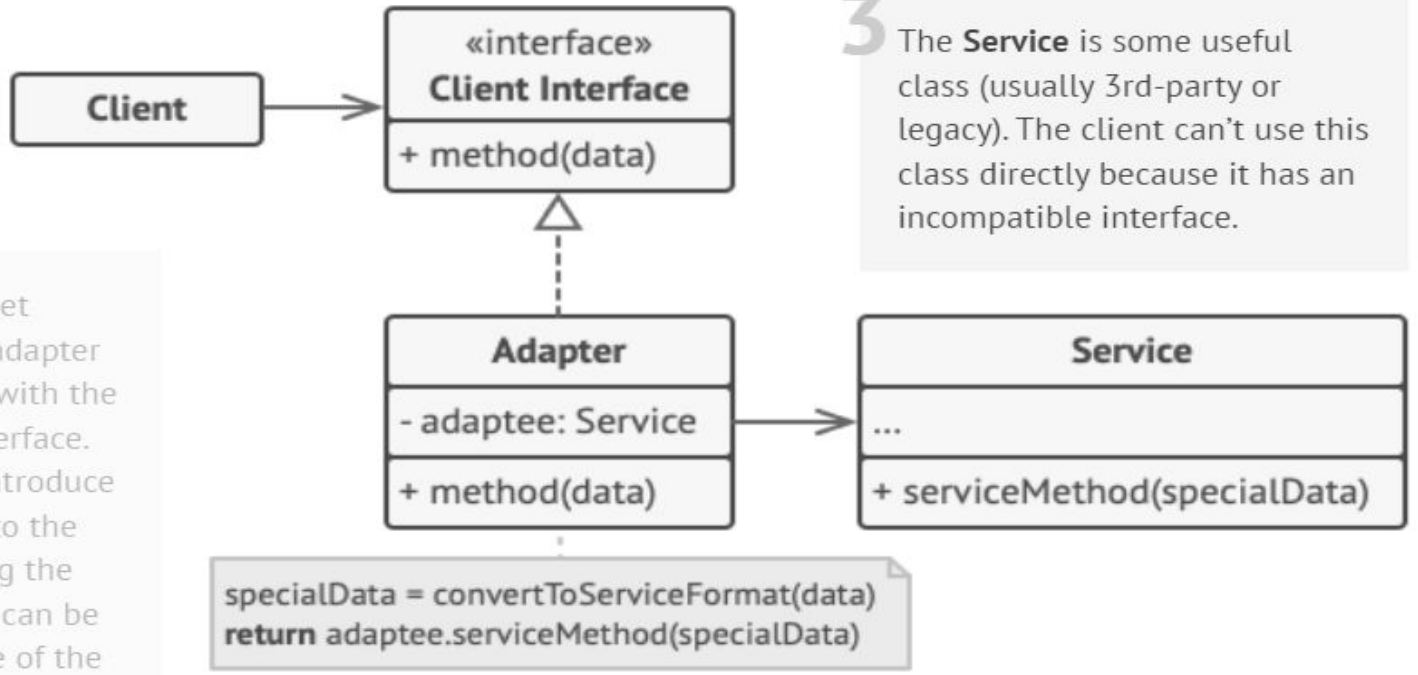


3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

Structure (Object Adapter)



1 The **Client** is a class that contains the existing business logic of the program.

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

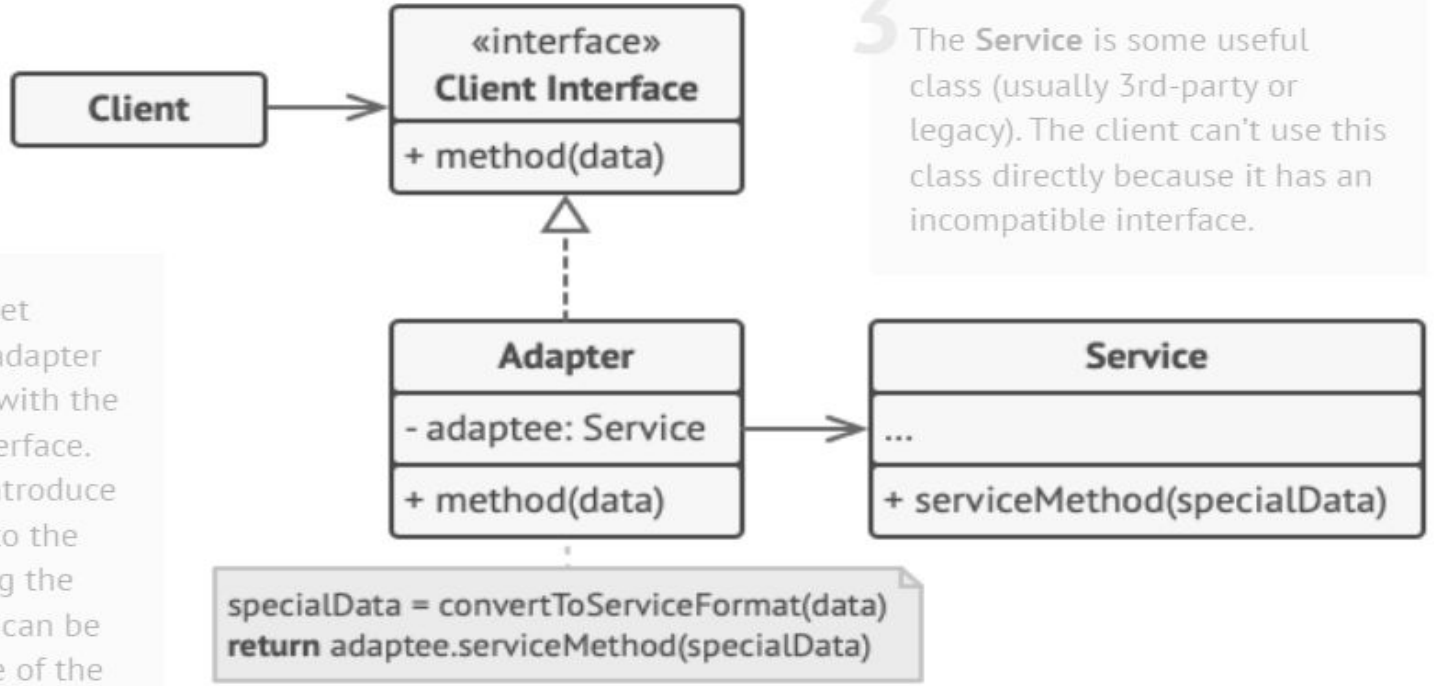
4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

Structure (Object Adapter)

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

1 The **Client** is a class that contains the existing business logic of the program.



3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

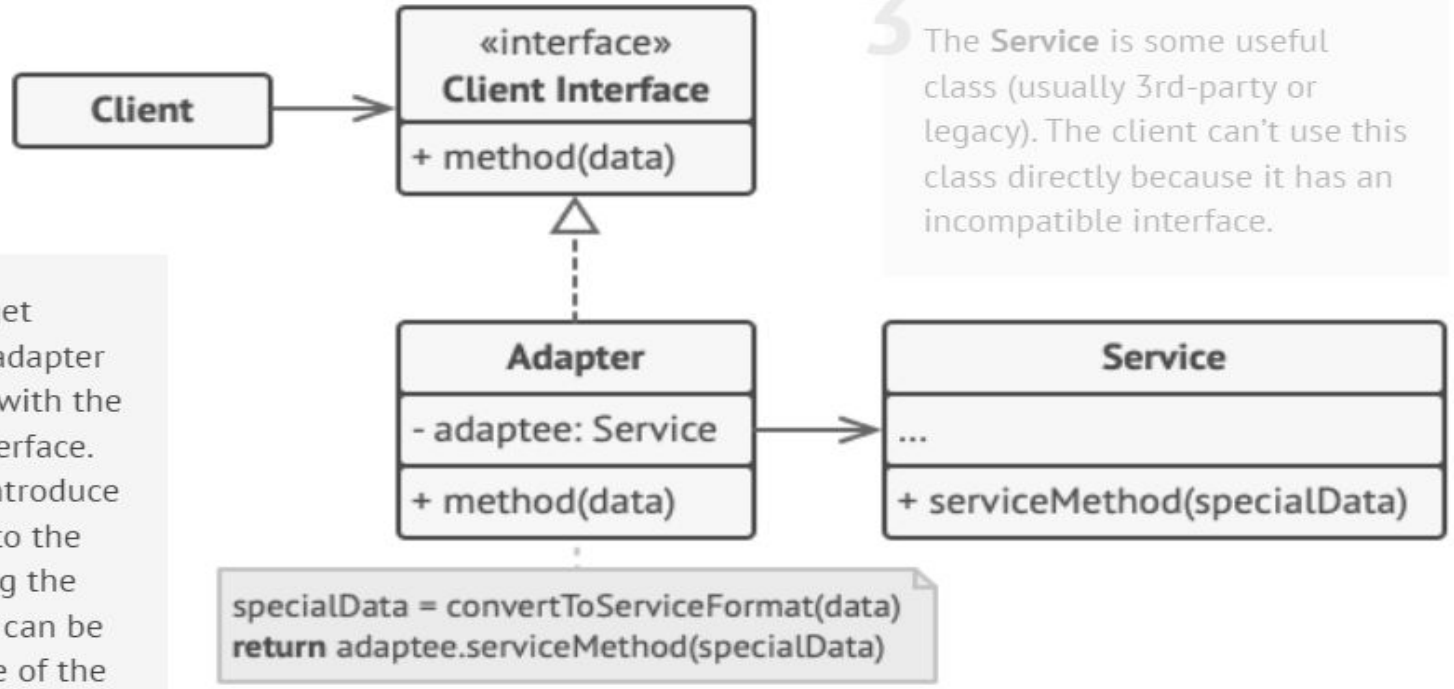
5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

Structure (Object Adapter)

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

1 The **Client** is a class that contains the existing business logic of the program.

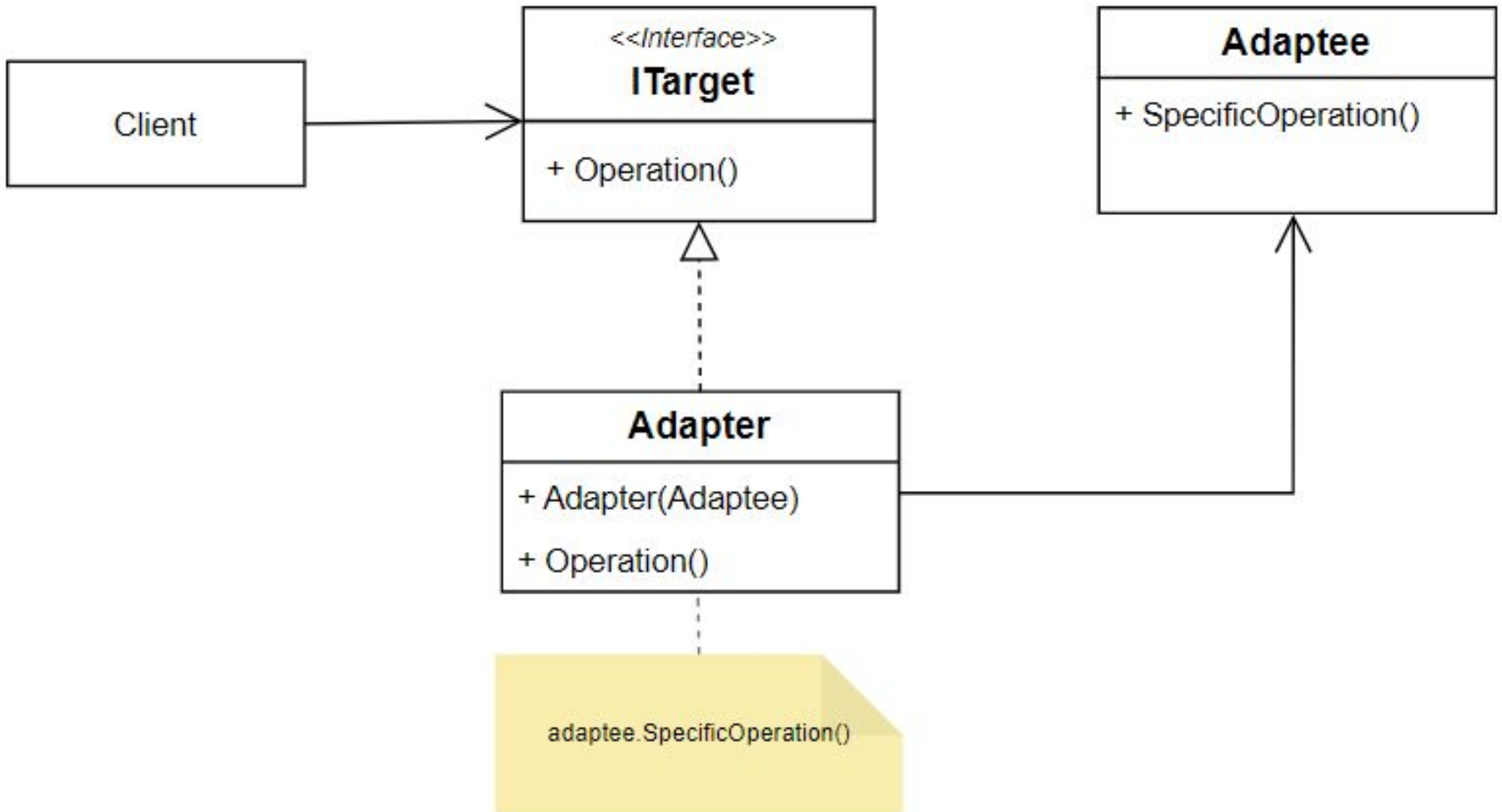


3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

Example (Object Adapter)



Example (Object Adapter)

```
// Target: Desired interface our client expects
class Target {
public:
    virtual void request() = 0;
};
// Adapter: Adapts the Adaptee's interface to the Target interface
class Adapter : public Target {
private:
    Adaptee* adaptee;
public:
    Adapter(Adaptee* a) {
        adaptee = a;
    }
    void request() override {
        // Using Adaptee's specificRequest within the Target's request
        adaptee->specificRequest();
    }
};
```

```
// Adaptee: Existing class with an incompatible interface
class Adaptee {
public:
    void specificRequest() {
        cout << "Adaptee's specificRequest called" << endl;
    }
};
// Client code
int main() {
    Adaptee* adaptee = new Adaptee();
    Target* adapter = new Adapter(adaptee);

    // Client uses the Target interface to call the Adaptee's method
    adapter->request();
}
```

Output:

```
Adaptee's specificRequest called
```

Another Example (Object Adapter)

```
// Target: Represents a universal voltage interface
class Device220V {
public:
   virtual void operateAt220V() = 0;
};
// Adapter: Adapts Device110V to operate at 220V
class VoltageAdapter : public Device220V {
private:
   Device110V* device;
public:
   VoltageAdapter(Device110V* d) {
         device = d;
   }
   void operateAt220V() override {
// Using the Device110V's method within the Device220V's interface
      cout << "Adapter converting 220V to 110V." << endl;
      device->operateAt110V();
   }
```

```
// Adaptee: Represents a device designed for 110V
class Device110V {
public:
   void operateAt110V() {
      cout << "Operating at 110V" << endl;
   }
};
// Client code
int main() {
   Device110V* device110V = new Device110V();
   Device220V* adapter = new VoltageAdapter(device110V);
   adapter->operateAt220V();
}
```

Output:

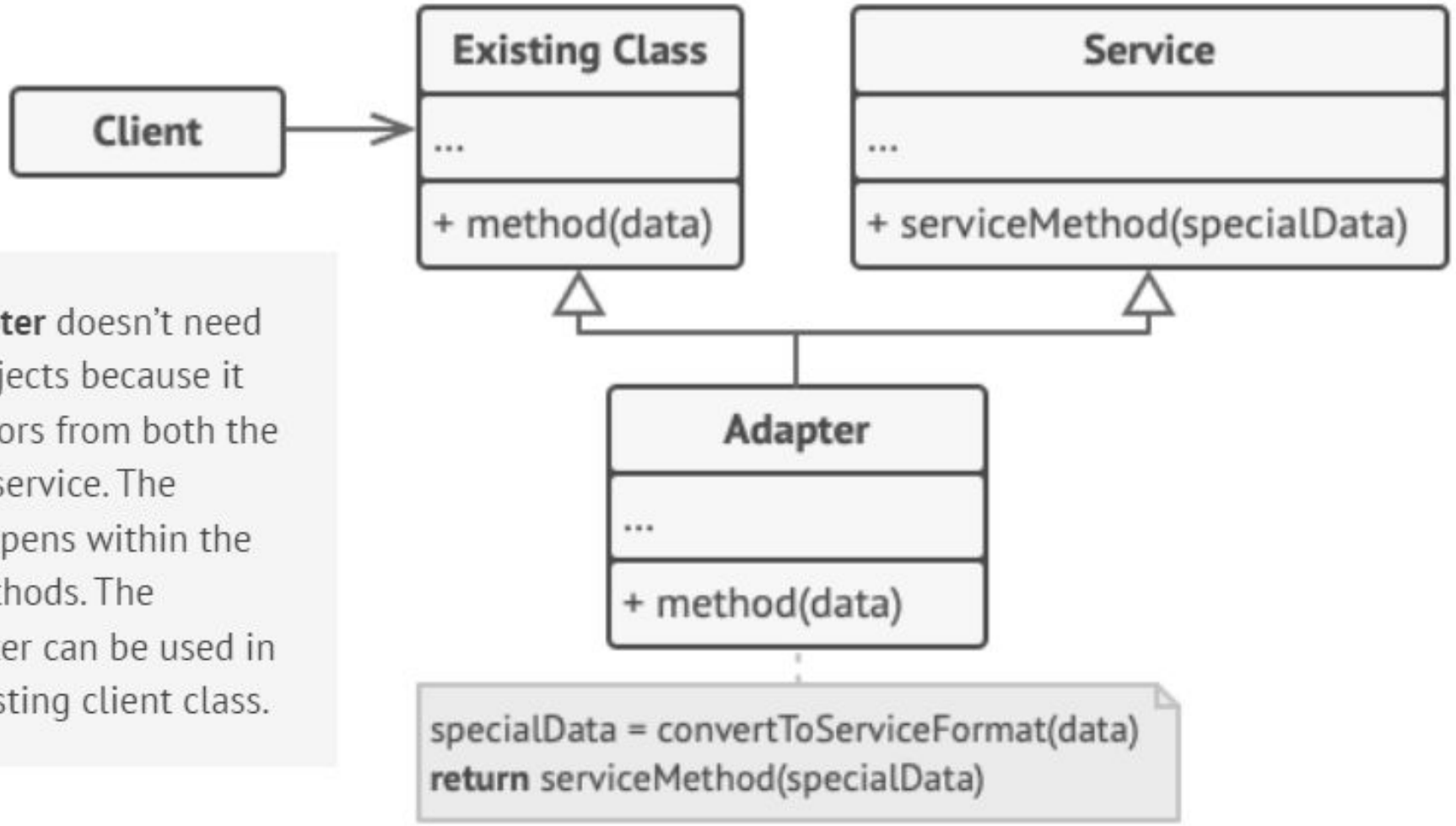
Adapter converting 220V to 110V.

Operating at 110V

Structure (Class Adapter)

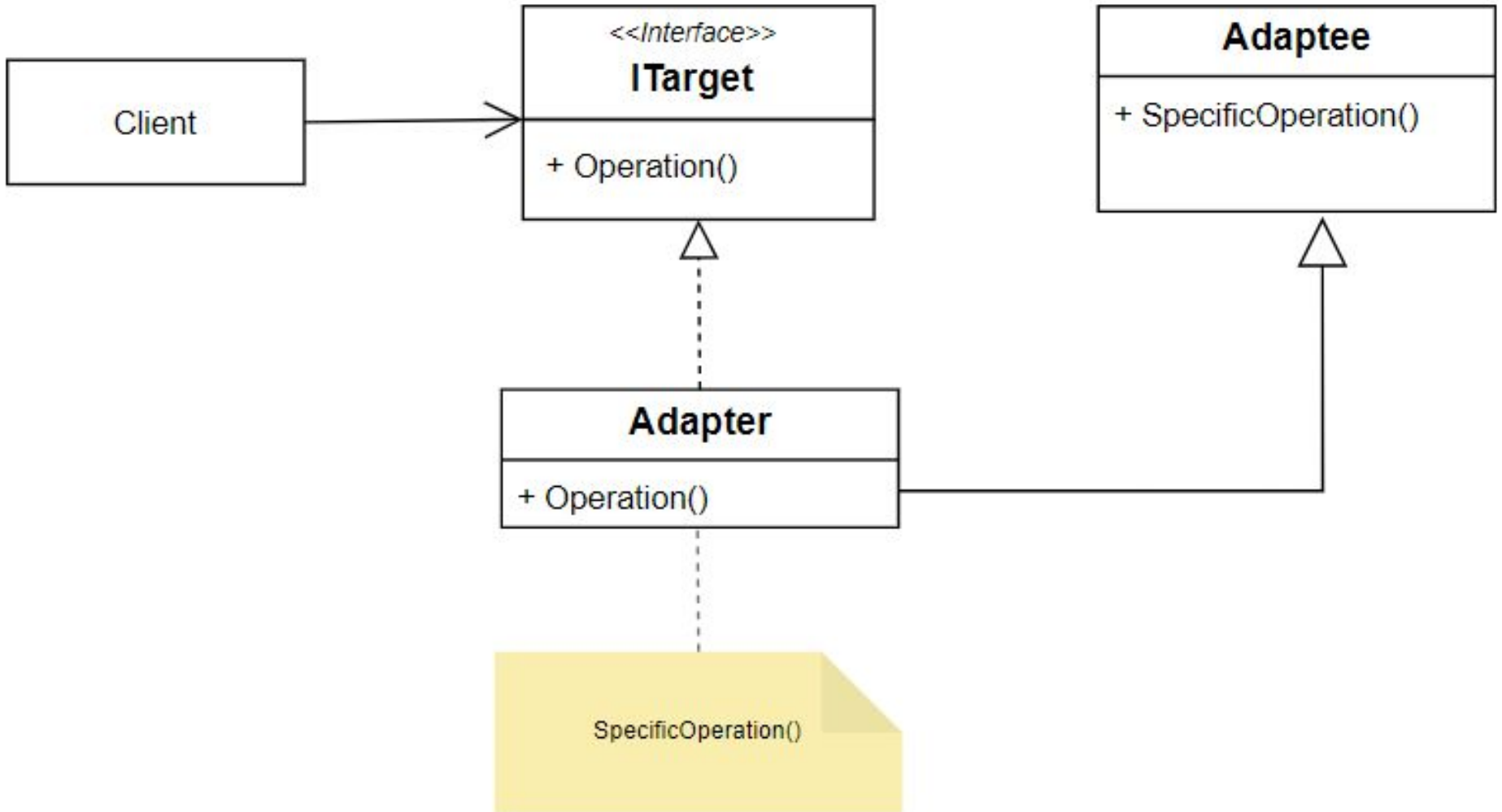
- This implementation uses **inheritance**:
  - Adapter inherits interfaces from both objects at the same time.
- This approach can only be implemented in programming languages that support multiple inheritance, such as C++.

Structure (Class Adapter)



1 The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

Example (Class Adapter)



Example (Class Adapter)

```
// Adaptee: Existing class with an incompatible interface
class Adaptee {
public:
    void specificRequest() {
        cout << "Adaptee's specificRequest called" << endl;
    }
};

// Target: Desired interface our client expects
class Target {
public:
    virtual void request() = 0;
};
```

```
// Adapter: Adapts the Adaptee's interface to the Target interface
class Adapter : public Target, private Adaptee {
public:
    void request() override {
        // Using Adaptee's specificRequest within the Target's request
        specificRequest();
    }
};
// Client code
int main() {
    Target* adapter = new Adapter();
    // Client uses the Target interface to call the Adaptee's method
    adapter->request();
}
```

Output:

Adaptee's specificRequest called

Class Adapter Pattern

- **Implementation:**
  - In the Class Adapter pattern, adaptation is achieved through **class inheritance**.
  - The adapter class extends both the **target interface** (the desired interface the client uses) and the **adaptee class** (the class with an incompatible interface).
- **Pros and Cons:**
  - **Pros:** Simpler structure, as it uses inheritance to adapt the interfaces.
  - **Cons:** It's inflexible when dealing with multiple inheritance limitations (e.g., in languages that do not support multiple inheritance, like Java).

Object Adapter Pattern

- **Implementation:**
  - In the Object Adapter pattern, adaptation is achieved through **composition**.
  - The adapter contains an **instance of the adaptee class**, allowing it to **delegate** the calls from the target interface to the adaptee.
- **Pros and Cons:**
  - **Pros:** More flexible and versatile, as it doesn't rely on inheritance. It can adapt multiple adaptee classes by containing different instances.
  - **Cons:** Can result in more code due to the need for delegation methods and can be more complex to manage multiple instances.

References

- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Pearson, 1995.
- Helping Links:
- https://refactoring.guru/design-patterns/
- https://sourcemaking.com/design_patterns/
- https://www.youtube.com/watch?v=Q1jZ4TI6MF4&t=297s&ab_channel=Telusko