The Adapter Pattern

Interface with anything!

Adapter in a Nutshell

- An adapter **takes an object with one interface,**
  and **changes the interface** to make it look like something it's not.

- Allows two objects to work together, even though they were not designed to.

- Doesn't necessarily add any new functionality (in the strictest sense). Just performs a conversion

What is a example of an adapter that we have all used at some point?

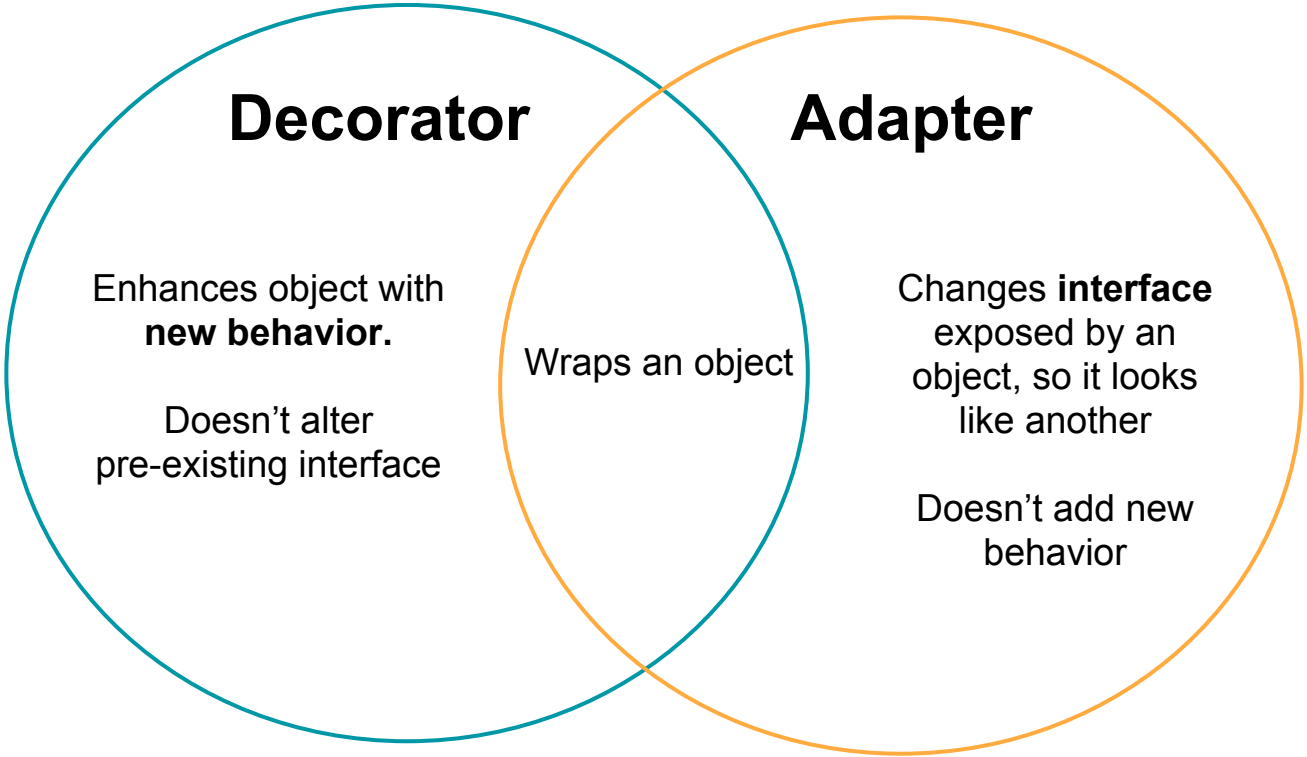
AC power **adapter** lets US plugs work with Euro outlets

Some adapters can have complex internals

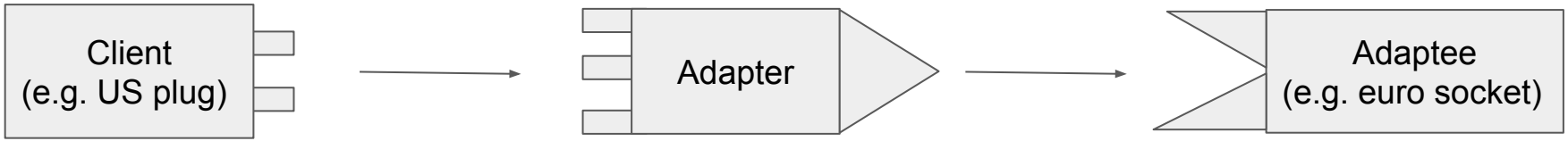
- E.g to increase/decrease voltage when appropriate

No need to modify either the client (the plug) or the adaptee (the socket).

Sounds like decorator? Close, but not quite...



What are the steps for using adapter?



1. Client makes a request to the adapter by calling a method on it using the target interface
2. The adapter translates the request into one or more calls on the adaptee using the adaptee interface
3. The client receives the results of the call and doesn't need to know anything about the adaptee to do so.

Designing an adapter

1) **Consider your target interface**
   a) This is the interface that your adapter will be implementing
   b) Ask: What is the interface that your clients want to use?

2) **Consider your adaptee**
   a) This is what your adapter will be composing (converting)
   b) Perform a mapping of target interface methods to adaptee methods

3) **What about methods on target that don't map to adaptee?**
   a) Throw an `UnsupportedOperationException`
   b) This is the best we can do, adapter can't solve this problem :)

Talk is cheap...

When would we use adapter?

- Third party library integrations
  - Ideally, 3rd parties will provide the adapter!
- When you have working code, and do not want to modify its implementation
- When changes to an interface happen frequently (decouples client)
- Migrations from one interface to another (two-way adapter)
  - Adapter would implement both the old and new interfaces!
- New interface is released, but not supported by legacy classes
  - I.e. the enumeration interface vs the iterator interface

The adapter pattern **encapsulates** necessary interface changes into one place

Bonus pattern: Facade

Simplify your Subsystems

Home theater system

- Has many different interacting components
- To watch a movie requires that the client (you) **interface directly with many portions of the subsystem**
- You need to remember the order for both subsystems and

Home theater Facade

- Contains a **simplified interface** for clients to interact with

- **Does not encapsulate** the subsystem… can still call components directly

- **Decouples** clients from subsystem components. Can “upgrade” components without affecting client!

The Principle of Least Knowledge

- Be careful of the number of classes your objects interact with

- More dependencies can lead to tightly coupled designs

- Following this strictly can lead to more wrapper classes in your code, which can increase complexity and decrease runtime performance.

More of a **guideline** instead of a **rule**

Guidelines

From any method of an object, the principle suggests that we should only invoke methods that belong to

- The object itself
- Objects passed in as a parameter to the method
- Any object the method creates or instantiates
- Any component of the object (i.e. composed objects in the class)
- **Do Not** call methods on objects **returned** from other method calls.

Summary

Adapter

- Wraps Object to **convert interface** from one to another
- Does not add new behavior

Facade

- Provides new interface to **simplify** complex subsystems
- Provides template methods for performing common tasks

Decorator

- Wraps object to provide **new behavior**
- Does not modify existing interface otherwise

Questions?