CSE 219
COMPUTER SCIENCE III

STRUCTURAL DESIGN PATTERNS

SLIDES COURTESY:

RICHARD MCKENNA, STONY BROOK UNIVERSITY.

Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
		• Iterator
		• State

Textbook: Head First Design Patterns

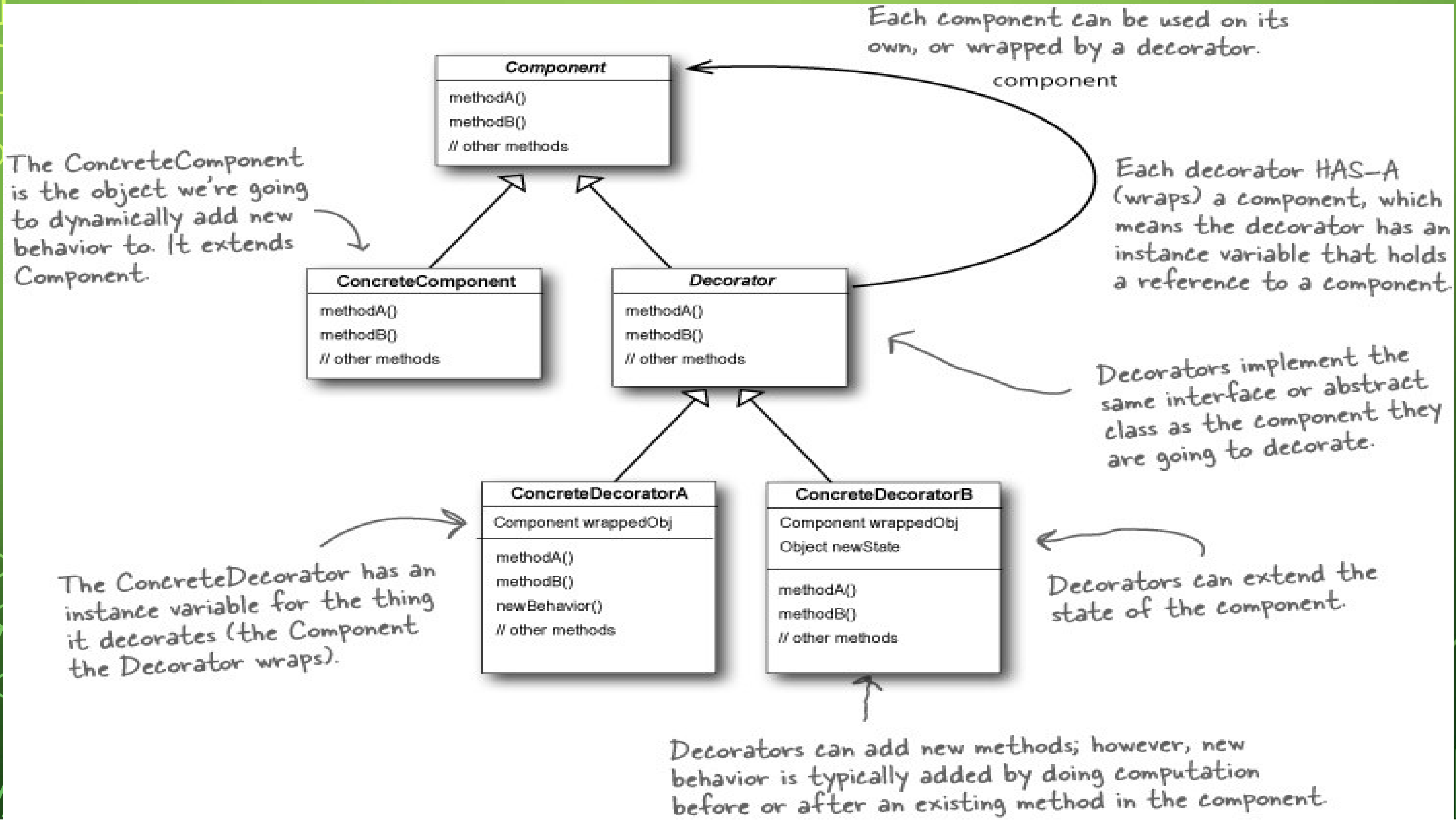
The Decorator Pattern

- Attaches additional responsibilities to an object ***dynamically***.
  - i.e. *decorating* an object
- Decorators provide a flexible alternative to sub-classing for extending functionality
- How?
  - By ***wrapping*** an object
- Works on the principle that classes should be open to extension but closed to modification

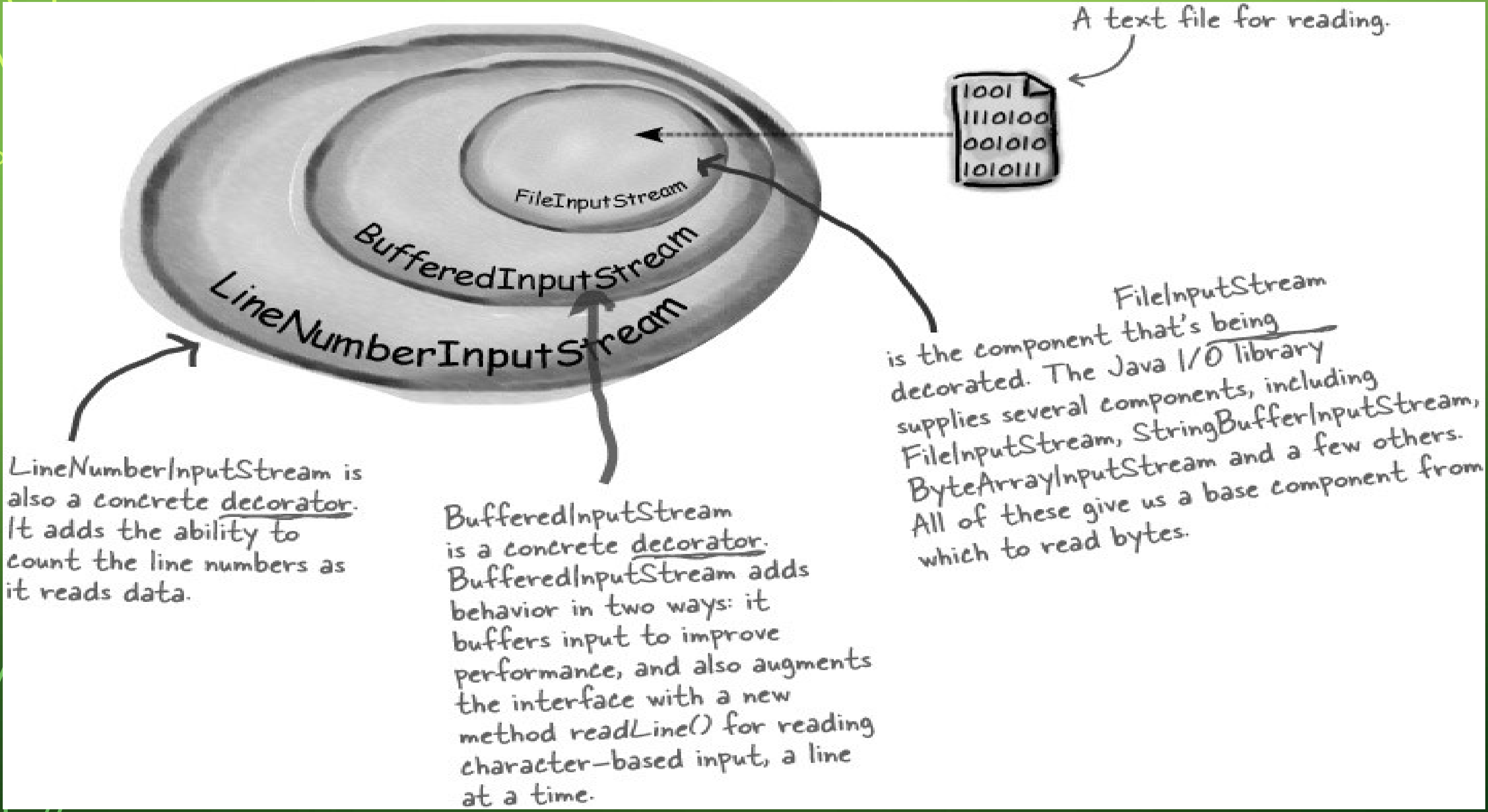
Decorator Goal

- Allow classes to be easily extended to incorporate new behavior without modifying existing code
- What do we get if we accomplish this?
  - Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

Decorators Override Functionality


Each component can be used on its own, or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcreteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcreteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorates (the Component the Decorator wraps).
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

Java's IO Library



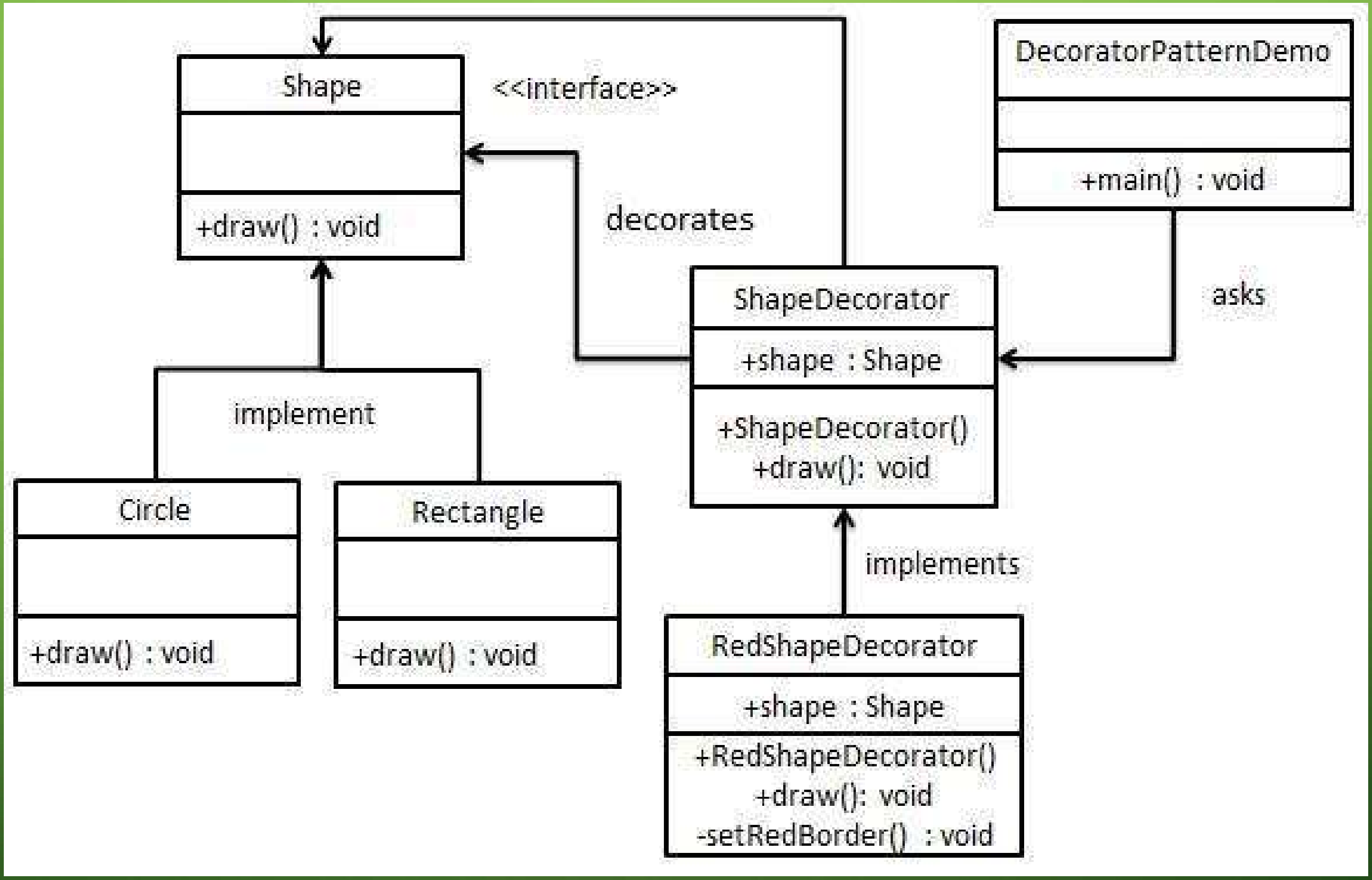
A text file for reading.

FileInputStream is the component that's being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.

Tutorial



https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

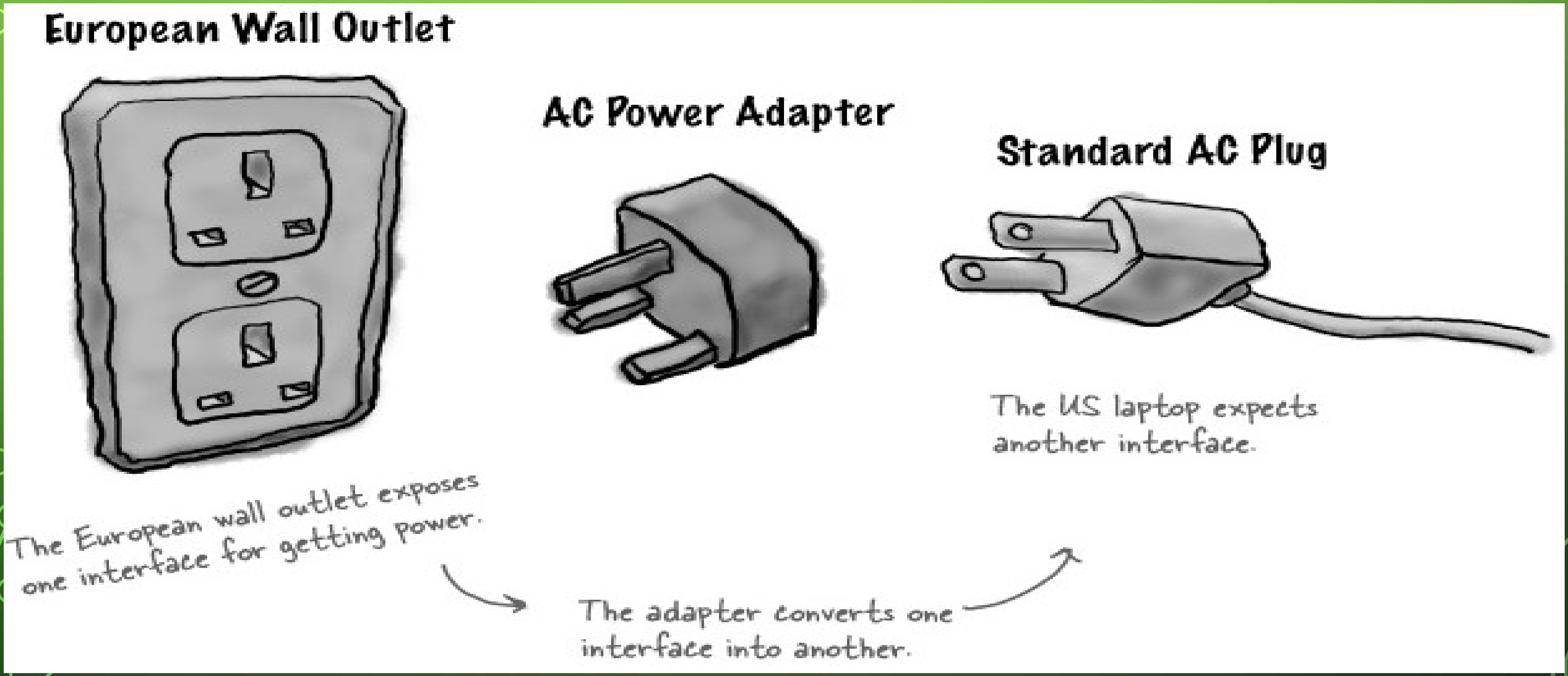
Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
		• Iterator
		• State

Textbook: Head First Design Patterns

Ever been to Europe?

- This is an abstraction of the Adapter Pattern


European Wall Outlet
AC Power Adapter
Standard AC Plug
The European wall outlet exposes one interface for getting power.
The adapter converts one interface into another.
The US laptop expects another interface.

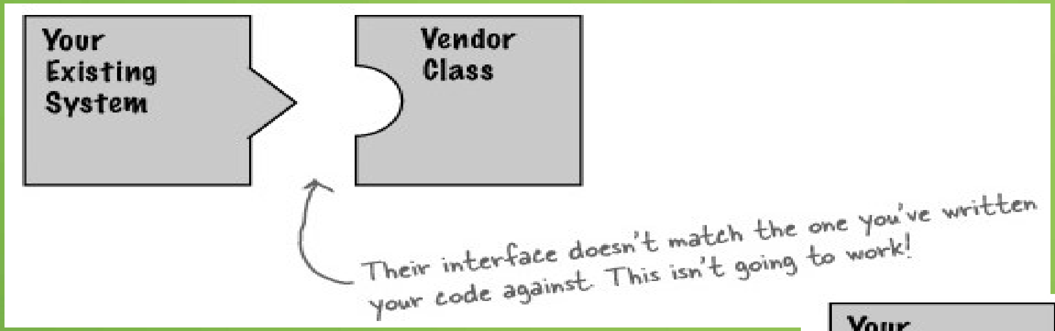
The Adapter Pattern

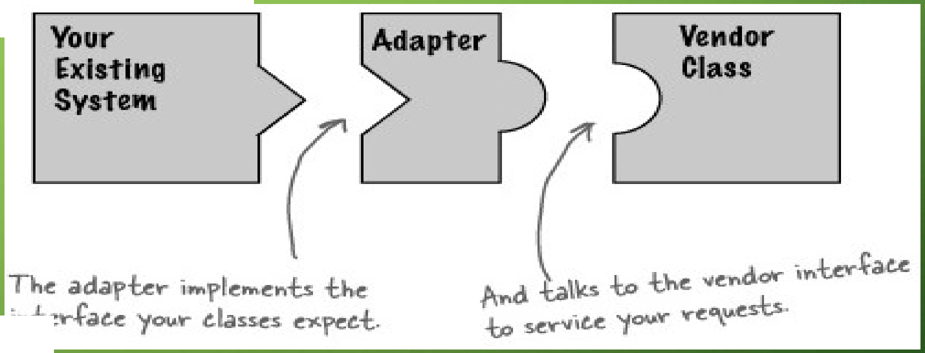
- Converts the interface of a class into another interface a client expects
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- Interfaces?
  - Do you know what a driver is?

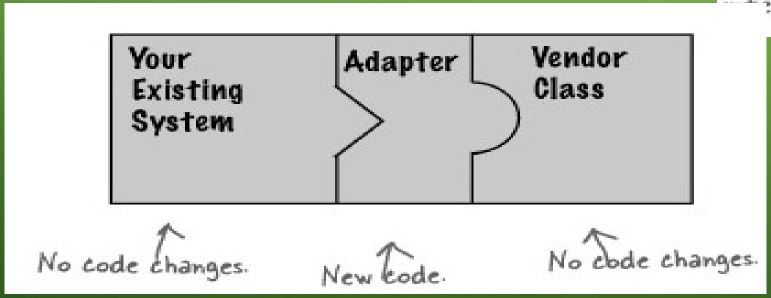
Adapter Scenario

- You have an existing system
- You need to work a vendor library into the system
- The new vendor interface is different from the last vendor
- You really don’t want to change your existing system
- Solution?
  - Make a class that adapts the new vendor interface into what the system uses

Adapter Visualized


Your Existing System
Vendor Class
Their interface doesn't match the one you've written your code against. This isn't going to work!


Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.



How do we do it?

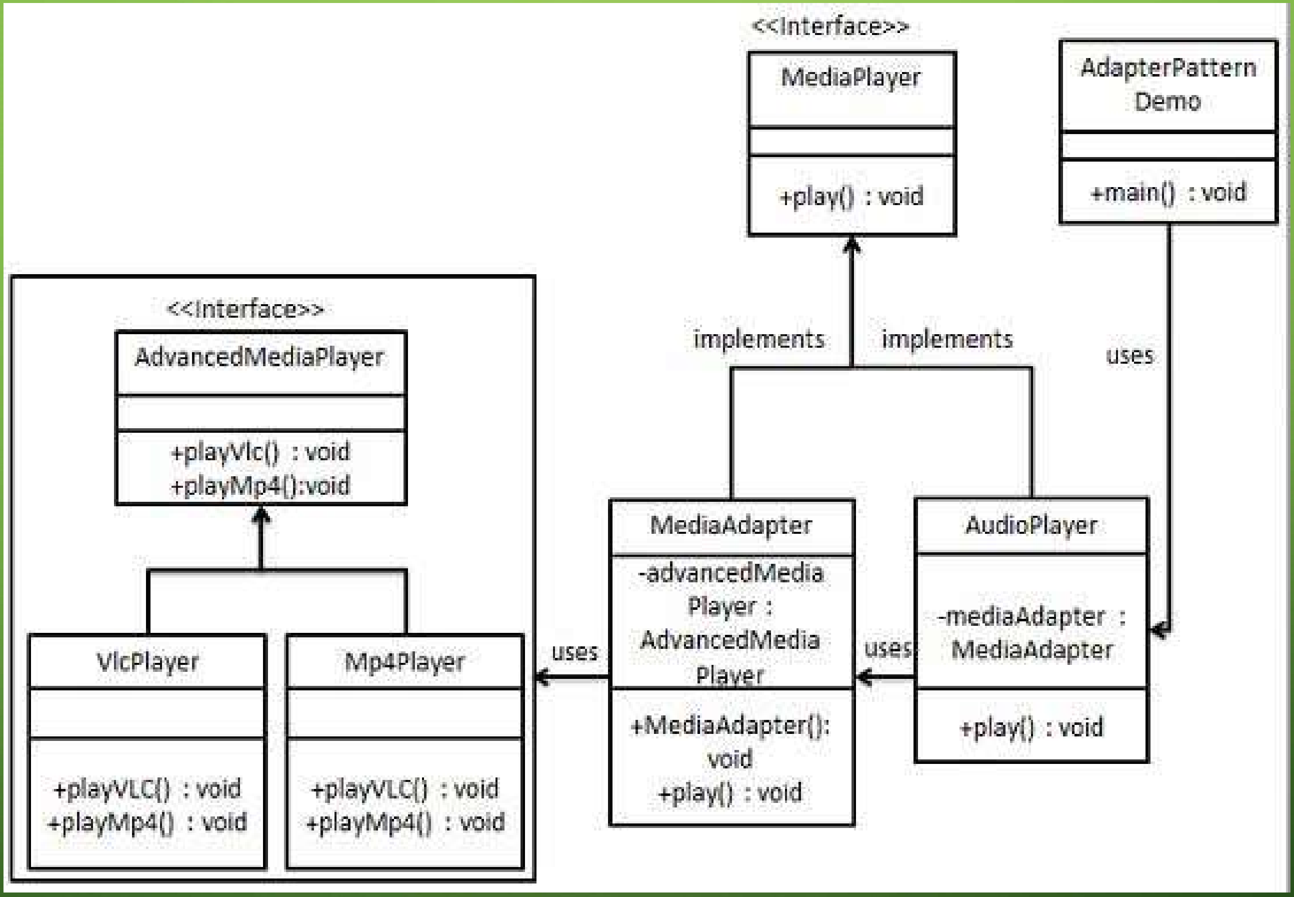
- Ex: Driver
  - Existing system uses a driver via an interface
  - New hardware uses a different interface
  - Adapter can adapt differences

- Existing system HAS-A OldInterface
- Adapter implements OldInterface and HAS-A NewInterface
- Existing system calls OldInterface methods on adapter, adapter forwards them to NewInterface implementations

What’s good about this?

- Decouple the client from the implemented interface
- If we expect the interface to change over time, the adapter encapsulates that change so that the client doesn’t have to be modified each time it needs to operate against a different interface.

Tutorial



https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

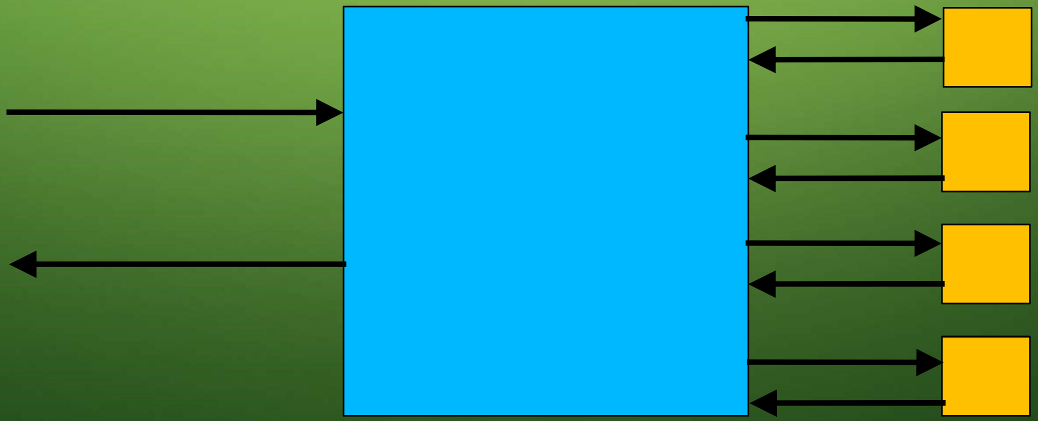
Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
		• Iterator
		• State

Textbook: Head First Design Patterns

The Facade Pattern

- Provides a unified interface to a set of interfaces in a subsystem.
- The facade defines a higher-level interface that makes the subsystem easier to use
- Employs the principle of least knowledge
- A facade is a class or a group of classes hiding internal implementation/services from the user.
- The factory pattern is used when you want to hide the details on constructing instances.



Scenario: We need a dialog

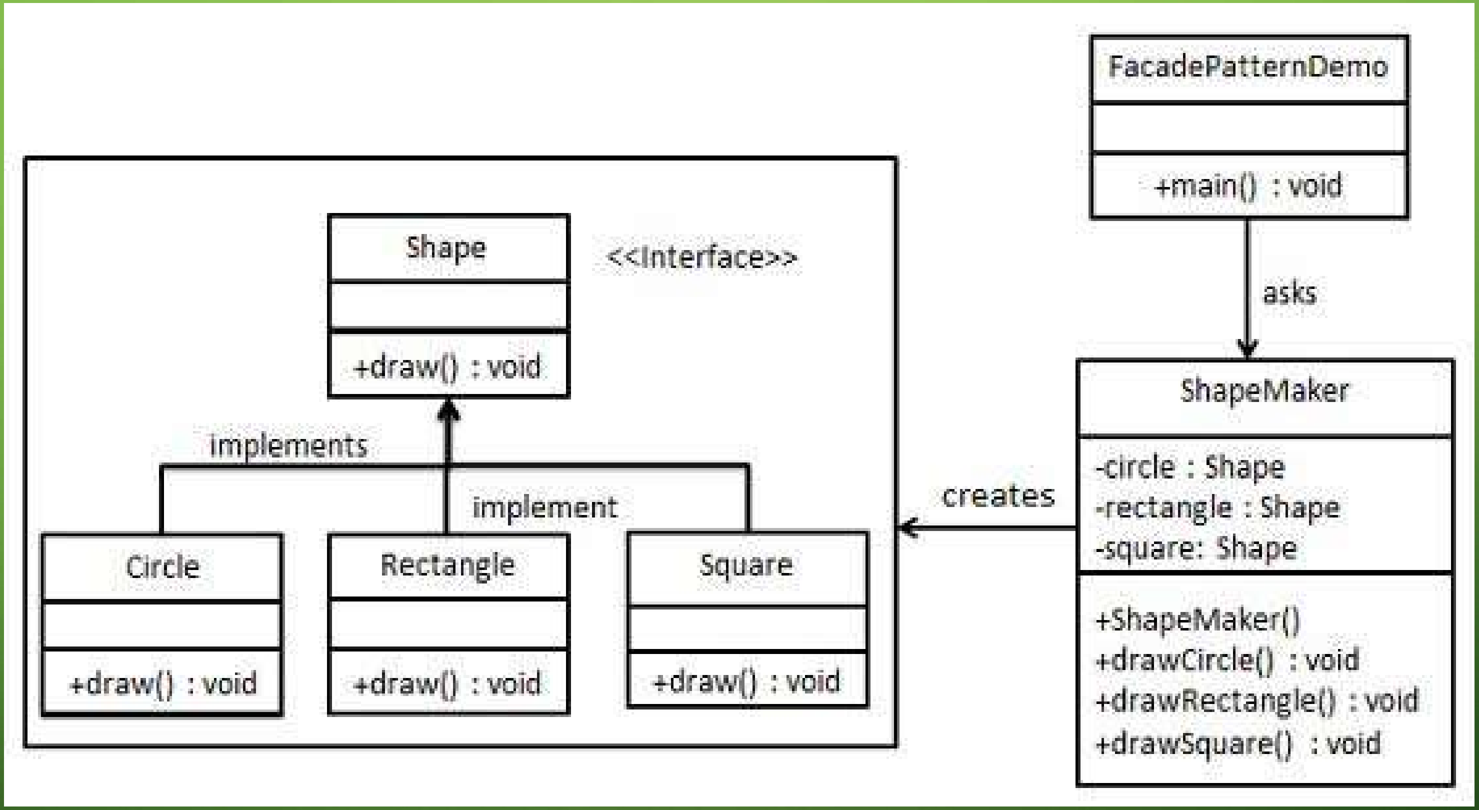
- Making a dialog can be a pain
  - setting up controls
  - providing layout
  - many common simple dialogs needed
  - applications like common presentation settings

- Solution?
  - **`AppDialogsFacade`**

AppDialogsFacade

```
public class AppDialogsFacade {
    public static void showAboutDialog(
    public static void showExportDialog(
    public static void showHelpDialog(
    public static void showLanguageDialog(
    public static void showMessageDialog(
    public static File showOpenDialog(
    public static File showSaveDialog(
    public static void showStackTraceDialog(
    public static String showTextInputDialog(
    public static String showWelcomeDialog(
    public static ButtonType showYesNoCancelDialog(
}
```

Tutorial



https://www.tutorialspoint.com/design_pattern/facade_pattern.htm

Which is which?

- Converts one interface to another
- Makes an interface simpler
- Doesn’t alter the interface, but adds responsibility

A)Decorator

B)Adapter

C)Facade

Common Design Patterns

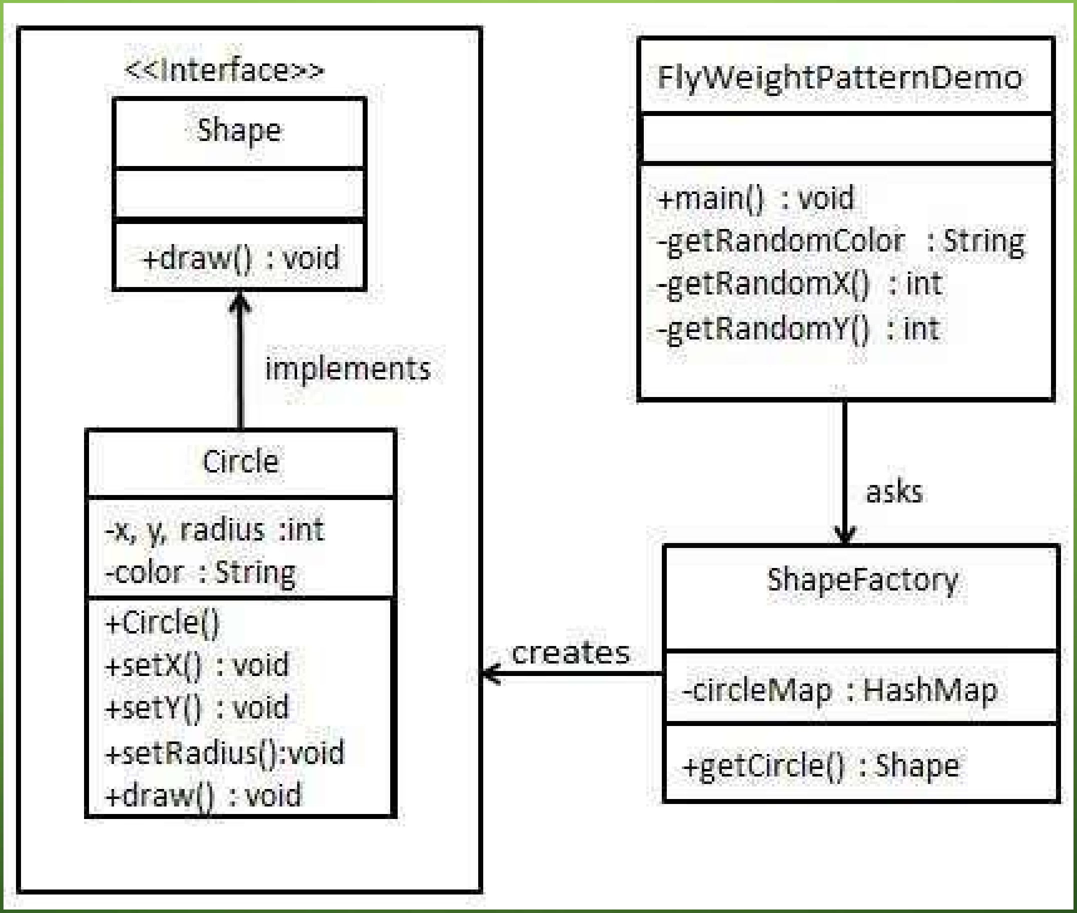
Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
		• Iterator
		• State

Textbook: Head First Design Patterns

The Flyweight Pattern

- A “neat hack”
- Allows one object to be used to represent many identical instances
  - Flyweights must be immutable.
  - Flyweights depend on an associated table
    - maps identical instances to the single object that represents all of them
- Used in processing many large documents
  - search engines
  - a document as an array of immutable Strings
  - repeated Words would share objects
  - just one object for “the” referenced all over the place
    - use static Hashtable to store mappings
    - use javax.naming.Context to provide String to Object binding

Tutorial



https://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm

Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Component Architecture	• Iterator
		• State

Textbook: Head First Design Patterns

A Component Architecture

- System uses a set of pluggable ***components***

- **Each component:**
  - can be plugged in
  - can be updated
  - can be replaced

  independently of the other components

`AppTemplate` uses Components

- With Default behavior:
  - `AppFileModule`
  - `AppFoolproofModule`
  - `AppGUIModule`
  - `AppLanguageModule`
  - `AppRecentWorkModule`
- With Custom behavior:
  - `AppClipboardComponent`
  - `AppDataComponent`
  - `AppFileComponent`
  - `AppWorkspaceComponent`