Java + OO overview

Java is an object-oriented language

- Java is an object-oriented programming language
  - Everything is an object
- A class is a blueprint for creating objects
  - Defines properties (fields) and behaviors (methods)
- An object is an instance of a class

```
public class Car {
    private String model;
    private int year;
    private int mileage;

    public void start() { …}
    public void stop() { … }
    public void accelerate() { … }
    public void brake() { … }
}
public class MyApplication {
    public static void main(String[] args) {
        Car car = new Car();
    }
}
```

Reference ***Object***

Encapsulation

- Internal details of a class are hidden from outside world
- Access is controlled using public, private, protected access modifiers
  - Direct access to private fields prevented
  - Needs getter and setter methods

```
public class Car {
    private String model;
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
    // … other fields and methods
}
public class MyApplication {
    public static void main(String[] args) {
        Car car = new Car();
        car.model = “Toyota Hybrid”; // COMPILER ERROR
    }
}
```

Abstraction

- Abstraction means hiding details
  - Enables user to use component without knowing details
- Encapsulation ***enforces*** abstraction
- **Critical** to ensuring users of your code use it in the way you intend!

```
// Breaking abstractions
class ClassDetails {
  public List<Student> students;
  public void dropStudent(Student s) {
    students.remove(s);
    if (s.isInternational()
        && !s.hasMinCredits()) {
        s.notifyEmail();
    }
  }
}

Student s = …;
ClassDetails details = …;
details.students.remove(s); // no check!
```

Inheritance

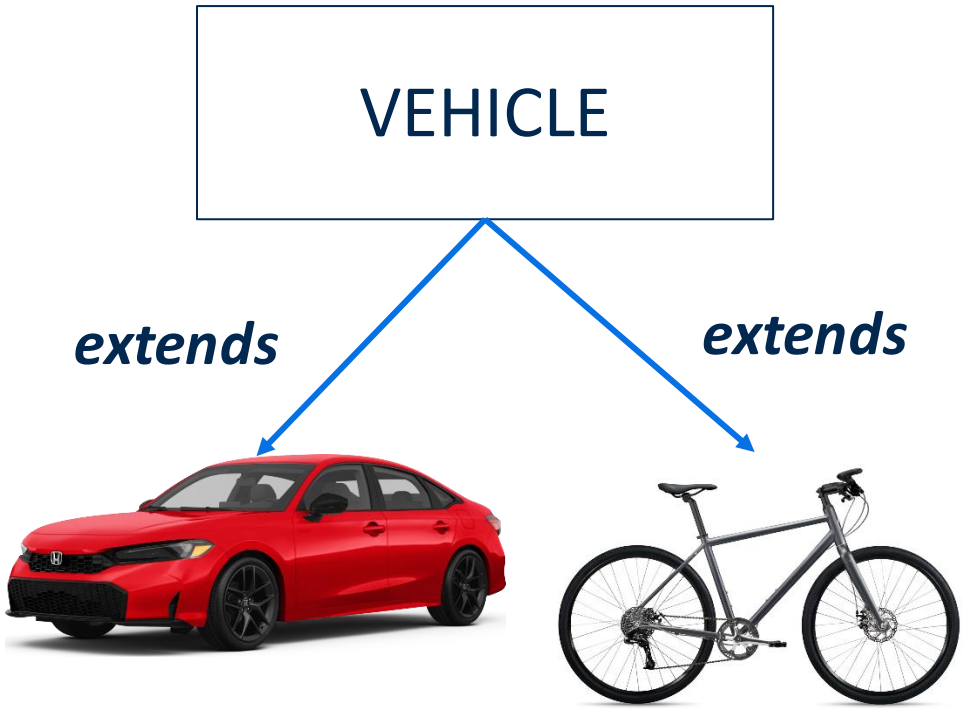
Inheritance allows code reuse

```
public class Car {
    private String model;
    private int year;
    private int mileage;
    public void start() { …}
    public void stop() { … }
    public void accelerate() { … }
    public void brake() { … }
}
```

Code duplication!

```
public class Bicycle {
    private String model;
    private int year;
    public void start() { …}
    public void stop() { … }
    public void accelerate() { … }
    public void brake() { … }
    public void ringBell() { … }
}
```

Inheritance




```
public class Vehicle { // shared properties
                       // and functions
    private String model;
    private int year;
    public void start() { …}
    public void stop() { … }
    public void accelerate() { … }
    public void brake() { … }
}

public class Car extends Vehicle {
    private int mileage;
    // getter and setter for mileage
}
public class Bicycle extends Vehicle {
    public void ringBell() { … }
}

public static void main(String[] args) {
    Bicycle bicycle = new Bicycle();
    bicycle.start();
    bicycle.ringBell();
}
```

What does it mean to instantiate an object of type Vehicle?

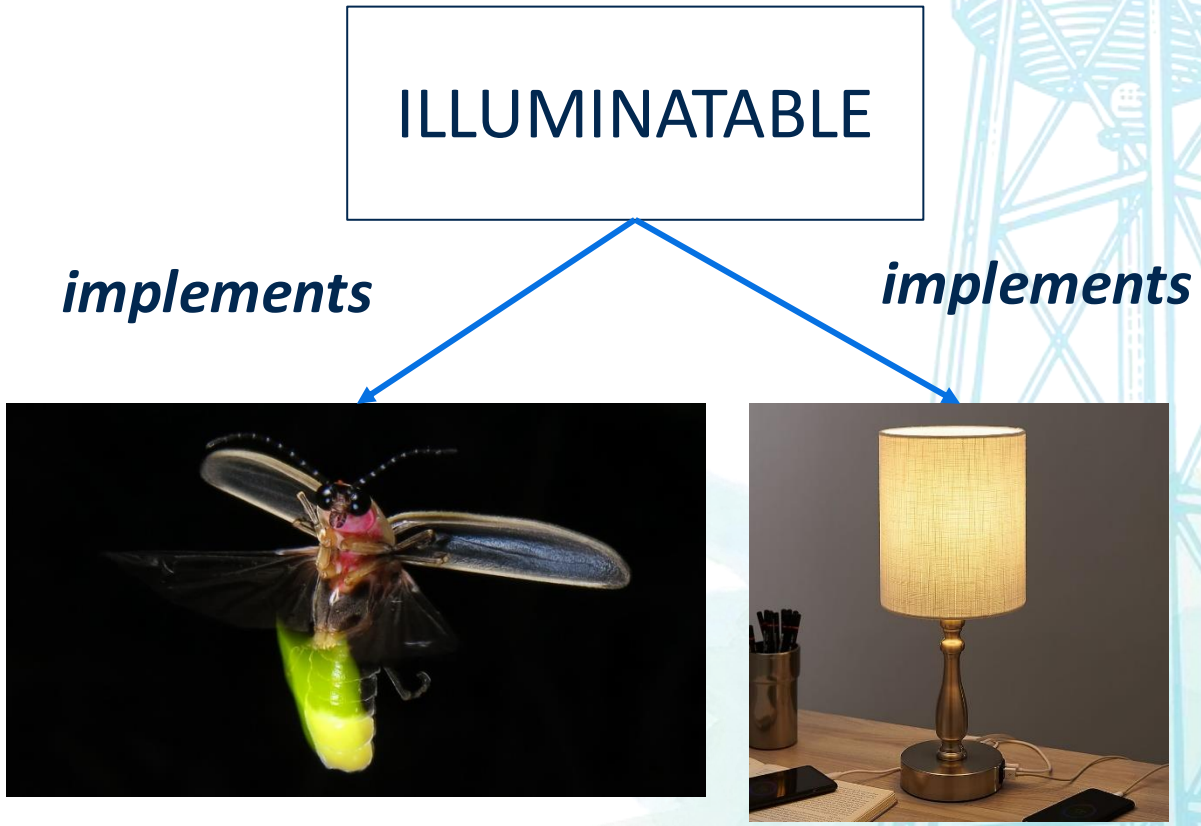
Abstraction via abstract classes

- Use abstract class when you want to prevent a class to be initiated
- An abstract class cannot be instantiated
- Must be extended
- Subclass must implement all unimplemented methods

```
public abstract class Vehicle {
    private String model;
    private int year;
    public void start() { …}
    public void stop() { … }
    public void accelerate() { … }
    public void brake() { … }
}

public class Car extends Vehicle {
    private int mileage;
    // getter and setter for mileage
}
public class Bicycle extends Vehicle {
    public void ringBell() { … }
}

public static void main(String[] args) {
    Vehicle vehicle = new Vehicle(); // Compiler error!!
}
```

Abstraction via interfaces

- Defines a contract for what a class must do but now how
- Supports multiple inheritance
- All methods in interface are abstract by default
- Cannot instantiate interfaces



Abstraction via interfaces

- Defines a contract for what a class must do but now how
- Supports multiple inheritance
- All methods in interface are abstract by default
- Cannot instantiate interfaces

```
public interface Illuminatable {
    void emitLight();
}

public class Lamp implements Illuminatable {
    public void emitLight() {
         System.out.println(“Lamp emits light!”);
    }
}

public class Firefly implements Illuminatable {
    public void emitLight() {
    System.out.println(“Firefly emits light”);
    }
}
```

Runtime polymorphism

- Real power behind inheritance
- A parent type reference can point to a child type object
- The method to be invoked is determined at runtime, depending on the type of object
- Key feature supporting design patterns, frameworks, and many more

```
public class Bicycle {
    private int speed = 8;
    public void accelerate() {
      this.speed+=5;
    }
}

public class ElectricBicycle extends Bicycle{
    public void accelerate() {
      this.speed+=10;
    }
}

public static void main(String[] args) {
    Bicycle b = new Bicycle();
    accelerate(b); // call's Bike
    ElectricBike e = new ElectricBike();
    accelerate(e); // call's ElectricBike
}

public static void accelerate(Bicycle b) {
    b.accelerate();
}
```

Static class members

- Declaring a field static ensures exactly a single copy of that field is created and shared among all instances of that class

```
public class Car {
    private String name;
    private String engine;

    public static int numberOfCars = 0;

    public Car(String name, String engine) {
this.name = name;
        this.engine = engine;
        numberOfCars++;
    }

    // getters and setters
}

public static void main(..) {
    Car c1 = new Car("Jaguar, "V8");
    Car c2 = new Car("Bugatti", "W16");
    System.out.println("Total number of cars: " +
Car.numberOfCars);
     // Prints 2
}
```

Static class methods

- A static class method does not need an object to invoke it

```
public class Car {
    private String name;
    private String engine;

    private static int numberOfCars = 0;

    // .. Other code
    public static int getNumberOfCars() {
        return numberOfCars;
    }
}

public static void main(..) {
    Car c1 = new Car("Jaguar, "V8");
    Car c2 = new Car("Bugatti", "W16");
    System.out.println("Total number of cars: " +
Car.getNumberOfCars());
     // Prints 2
}
```

Design patterns

Tapti Palit

What are design patterns?

- Classes and objects abstract object behavior
  - Objects of type Class Car abstract the behavior of a car
  - Focus on reusing object behavior
- Design patterns abstract object instantiation, composition, and behavior
  - Focus on abstracting the class design itself
  - Allows software engineers to *reuse* previous class designs and architectures

Design patterns are language-specific

- We will focus on object-oriented design patterns
  - Particularly, Java design patterns, but are applicable to C++, C#, etc., too
- Design patterns for functional programming languages and Rust will look very different

Need for design patterns

- Example: design a `Logger` class for an application consisting of hundreds of classes
  - The logger creates a log file with name as the current date-time, say – `log_11_21_2025_12_00_01.txt`, and logs events in it
- To ensure uniformity
  - Only one object of the `Logger` class must exist
    - Should be created by the first log message
  - Any part of the application can access the `Logger` object

Need for design patterns

- Naïve design
- Create a `Logger` object and pass it in each method declaration?
  - Requires extensive method signature modification

```
class Logger {
    private FileStream outputFile;

    public Logger() {
        outputFile = FileStream("log"+
LocalDateTime.now().toString()+".log");
    }

    public log(String msg) {
        outputFile.write(log);
    }

    protected finalize() throws Throwable {
        // close file
    }
}

public static void main(String[] args) {
    Logger logger = new Logger();
    anotherFunction(logger);
}

public void anotherFunction(Logger logger) {
    logger.log("This is a log message");
}
```

Need for design patterns

- Naïve design
- Create a `Logger` object and pass it in each method declaration?
  - Requires extensive method signature modification
  - Doesn't prevent accidental creation of new `Logger` objects

```
class Logger {
    private FileStream outputFile;

    public Logger() {
        outputFile = FileStream("log"+
LocalDateTime.now().toString()+".log");
    }

    public log(String msg) {
        outputFile.write(log);
    }

    protected finalize() throws Throwable {
        // close file
    }
}

public static void main(String[] args) {
    Logger logger = new Logger();
    anotherFunction(logger);
}

public void anotherFunction(Logger logger) {
    Logger logger2 = new Logger();
    logger2.log("This is a log message");
}
```

Need for design patterns

- Goal: Any part of the application can access the `Logger` object

`static` field design

- Create and store a `Logger` object as a static field in the `Logger` class
- We can access the logger object by `Logger.logger`
- No need to change method signature

```
class Logger {
    private FileStream outputFile;
    public static Logger logger = nullptr;

    public Logger(String outputFileName) {
        // ... code
    }

    public log(String msg) {
        // ... code
    }

    // ... code
}
public static void main(String[] args) {
    Logger logger = new Logger();
    Logger.logger = logger;

    anotherFunction();
}
public void anotherFunction() {
    Logger.logger.log("This is a log message");
}
```

`static` field design

- Requirements
  - At any time, only one object of the `Logger` class must exist
  - Any part of the application can access the `Logger` object

```
class Logger {
    private FileStream outputFile;
    public static Logger logger = nullptr;

    public Logger(String outputFileName) {
        // ... code
    }

    public log(String msg) {
        // ... code
    }

    // ... code
}
public static void main(String[] args) {
    Logger logger = new Logger();
    Logger.logger = logger;
     // log some things
    anotherFunction();
}
public void anotherFunction() {
    Logger logger = new Logger();
    Logger.logger = logger;
    Logger.logger.log("This is a log message");
}
```

Inconsistent logging!!!

Preventing misuse

- Writing code that cannot be misused
- If someone uses your class according to the class's public API it should just work!
- No "hidden" requirements

```
class Logger {
    private FileStream outputFile;
    public static Logger logger = nullptr;

    public Logger(String outputFileName) {
        // ... code
    }

    public log(String msg) {
        // ... code
    }

    // ... code
}
public static void main(String[] args) {
    Logger logger = new Logger();
    Logger.logger = logger;
     // log some things
    anotherFunction();
}
public void anotherFunction() {
    Logger logger = new Logger();
    Logger.logger = logger;
    Logger.logger.log("This is a log message");
}
```

Solution: singleton design pattern

- Turn the constructor `private`
- Provide a static method `getLogger()` that instantiates the logger
  - Must use `getLogger()` to get the logger
- `getLogger()` reuses existing logger if already created, if not, creates a new logger

```
class Logger {
     private FileStream outputFile;
     private static Logger logger = nullptr;

     private Logger() {
          outputFile = FileStream("log"+
LocalDateTime.now().toString()+".log");
     }

     public static Logger getLogger() {
          if (logger == nullptr) {
             logger = new Logger();
          }
          return logger;
     }

     public log(String msg) {
          outputFile.write(log);
     }

     // … more stuff
}
public static void main(String[] args) {
     anotherFunction();
}
public void anotherFunction() {
     Logger logger = Logger.getLogger();
     logger.log("This is a log message");
}
```

Solution: singleton design pattern

- Requirements
  - At any time, only one object of the `Logger` class must exist 
  - Any part of the application can access the `Logger` object 

```
class Logger {
     private FileStream outputFile;
     private static Logger logger = nullptr;

     private Logger() {
          outputFile = FileStream(“log”+
LocalDateTime.now().toString()+”.log”);
     }

     public static Logger getLogger() {
          if (logger == nullptr) {
             logger = new Logger();
          }
          return logger;
     }

     public log(String msg) {
          outputFile.write(log);
     }

     // … more stuff
}
public static void main(String[] args) {
     anotherFunction();
}
public void anotherFunction() {
     Logger logger = Logger.getLogger();
     logger.log(“This is a log message”);
}
```

Why do we need design patterns?

- The design of the Logger class captures a *design pattern*
  - Applicable in many other contexts
    - `Configuration` class, `DatabaseConnector` class, and so on
- Understanding this design pattern allows software engineers to apply the same solution to these other contexts without having to re-engineer it

Design pattern classification

Creational patterns

- Control how objects are created
- E.g, Factory, Singleton

Structural patterns

- Control how objects and classes are composed
- Deal with object relationships
- E.g., Adapter, Proxy, Decorator, Bridge, Facade

Behavioral patterns

- Control how objects distribute responsibilities
- Template method, State, Observer, Visitor

Creational patterns

- Abstract the instantiation process
- Provides flexibility in
  - What gets created
  - Who creates it
  - How it is created and when
- Singleton pattern (seen earlier), abstract factory

Abstract factory design pattern

Case study: UI toolkit

- Example: design a GUI app that can support both Windows and MacOS UI components
- Option 1 – add if-else statements for each UI component creation
  - Error-prone!

```
public class Button { // button stuff }
public class WinButton extends Button { // … }
public class MacOSButton extends Button { // … }

Button button = nullptr;
if (platform == “win”) {
     button = new WinButton();
} else {
     button = new MacOSButton();
}

Textbox textbox = nullptr;
if (platform == “win”) {
     textbox = new WinTextBox();
} else {
     button = new MacOSTextBox();
}

DropdownBox dropdownbox = nullptr;
if (platform == “win”) {
      dropdownbox = new WinDropdownBox();
} else {
      dropdownbox = new MacOSDropdownBox();
}
```

Abstract factory pattern for UI toolkit

- Create an abstract `UIFactory` class that provides abstract methods for UI component creation
- Create concrete factory classes for both Windows and MacOS

```
public class Button { // button stuff }
public class WinButton extends Button { // … }
public class MacOSButton extends Button { // … }
```

```
abstract Class UIFactory {
    public void createButton();
}
```

```
public class WinFactory extends UIFactory {
    @Override
    public void createButton() {
        return new WinButton();
    }
}
```

```
public class MacOSFactory extends UIFactory {
    @Override
    public void createButton() {
        return new MacOSButton();
    }
}
```

Abstract factory pattern for UI toolkit

- Application contains a field of type UIFactory
  - Initialized depending on the platform
  - ONLY place where the platform check is performed

```
public class Button { // button stuff }
public class WinButton extends Button { // … }
public class MacOSButton extends Button { // … }

abstract Class UIFactory { … }

public class WinFactory extends UIFactory { … }

public class MacOSFactory extends UIFactory { …}

public class Application {
    private UIFactory uiFactory;
    public Application() {
        String platform = detectPlatform();
        if (platform == “win”) {
            uiFactory = new WinFactory();
        } else if (platform == “macos”) {
            uiFactory = new MacOSFactory();
        }
    }

    public drawGUI() {
        Button button = uiFactory.createButton();
    }
}
```

Abstract factory

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- An abstract factory class provides an abstract interface for object creation
- Concreate factory sub-classes implement that abstract interface

Characteristics

- Promotes consistency among products
  - All UI families must support same functionalities
- Supporting new kind of UI family is easier

Characteristics

- Promotes consistency among products
- Supporting new kind of UI family is easier

```
Button button = nullptr;
if (platform == “win”) {
    button = new WinButton();
} else if (platform == “macos”) {
    button = new MacOSButton();
} else {
    button = new GnomeButton(); // linux
}
```

```
Textbox textbox = nullptr;
if (platform == “win”) {
    textbox = new WinTextBox();
} else if (platform == “macos” {
    button = new MacOSTextBox();
} // Linux not handled
```

What happens?

Characteristics

- Promotes consistency among products
- Supporting new kind of UI family is easier
- Catching errors at compile time is ***much better*** than during execution
  - ***Why…?***

```
public class Button { // button stuff }
public class WinButton extends Button { // … }
public class MacOSButton extends Button { // … }

abstract Class UIFactory {
     public void createButton();
     public void createTextbox();
}

public class WinFactory extends UIFactory {
     public void createButton() {
          return new WinButton();
     }
     public void createTextbox() {
          return new WinTextbox();
     }
}

public class LinuxFactory extends UIFactory {
     public void createButton() {
          return new GnomeButton();
     }
     // Missing createTextbox
}
```

What happens?

Design pattern classification

Creational patterns

- Control how objects are created
- E.g, Factory, Singleton

Structural patterns

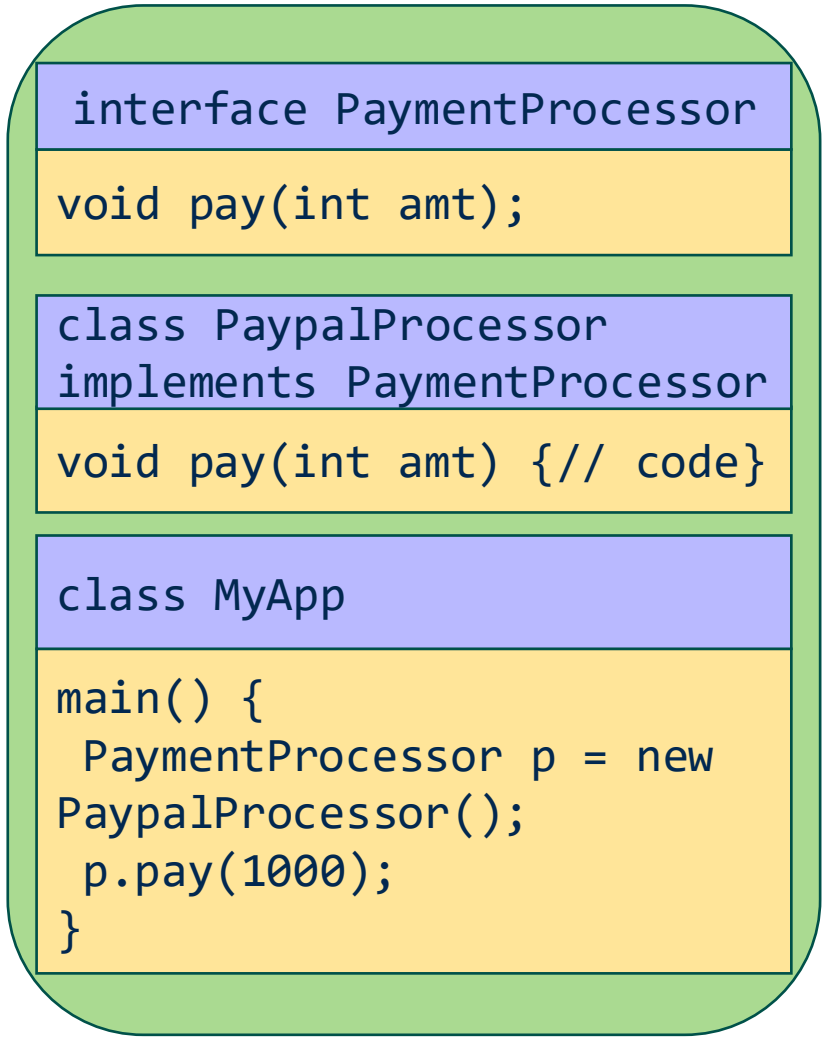
- Control how objects and classes are composed
- Deal with object relationships
- E.g., Adapter, Proxy, Decorator, Bridge, Façade

Behavioral patterns

- Control how objects distribute responsibilities
- Template method, State, Observer, Visitor

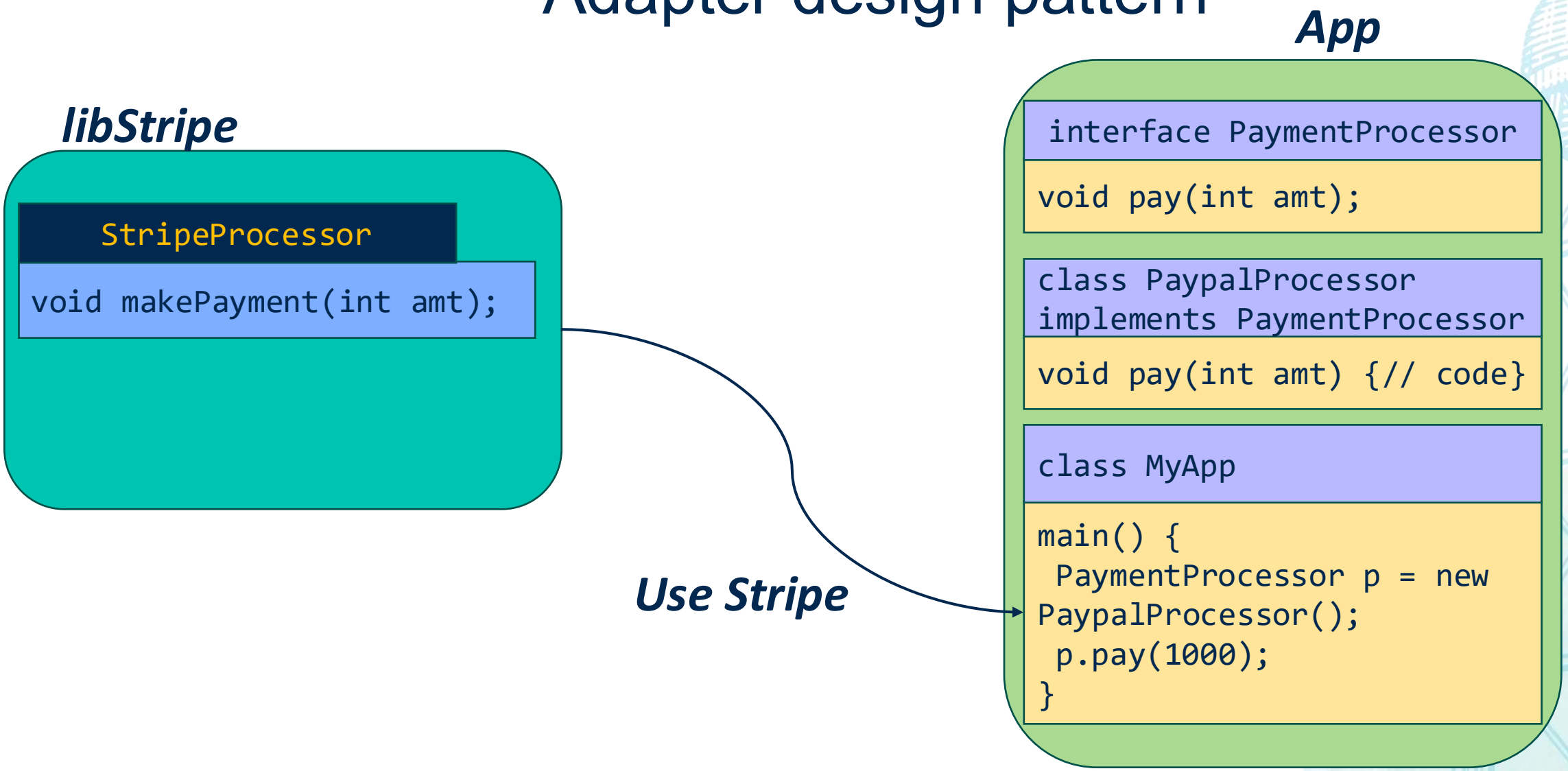
Adapter design pattern

Use case: payment processor

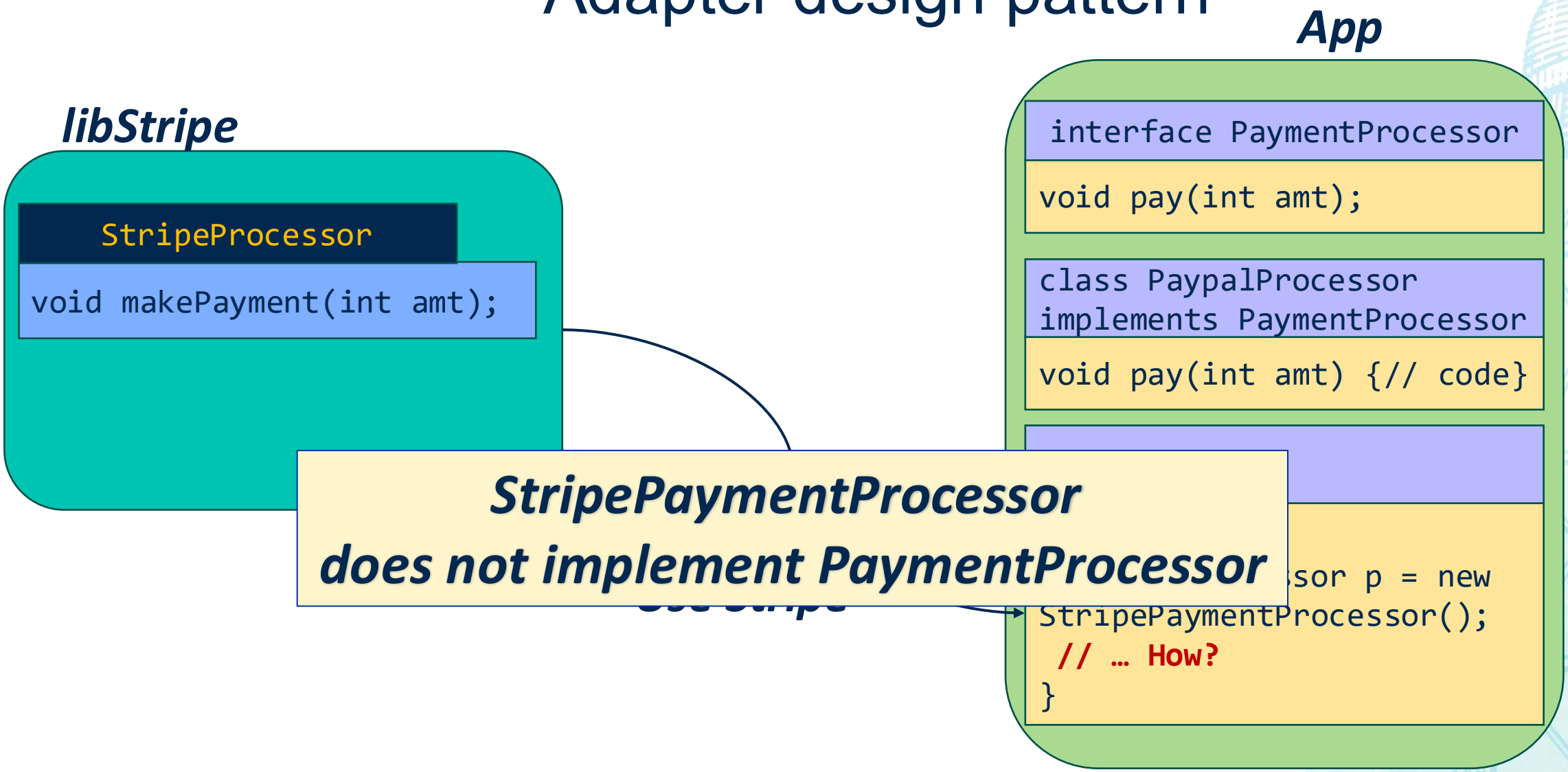


App

Adapter design pattern



Adapter design pattern



Payment interface adapter

- Example: an e-commerce website uses a unified payment interface (`PaymentProcessor`) to handle all payment requests

- Want to integrate a new third-party payment gateway (`ThirdPartyPayment`) with a different interface

```
interface PaymentProcessor
{
    void pay(int amount);
}

class PaypalProcessor implements PaymentProcessor {
    void pay (int amount) {
        // paypal functionality
    }
}

public static void main(…) {
    PaymentProcessor p = new PaypalProcessor();
    handlePurchase(p);
}
public void handlePurchase(PaymentProcessor p) {
    p.pay(2000);
}
```

Third party payment gateway

```
class StripeProcessor {
    void makePayment(double amount) {
        // stripe functionality
    }
}
```

Adapter design pattern

- Convert the interface of a class into another interface clients expect
- Allows classes to work together that couldn't otherwise because of incompatible interfaces
  - … just like a HDMI to USB-C adapter

Payment interface adapter

- Wrap the `StripeProcessor` object (adaptee) in an Adapter
- The `StripeAdapter` implements the `PaymentProcessor` interface
  - And internally invokes the adaptee's `makePayment()` method
- Can use `StripeAdapter` wherever `PaymentProcessor` is used

```
class StripeAdapter implements PaymentProcessor{
    private StripeProcessor stripeProcessor;

    public StripeAdapter(StripeProcessor p) {
        this.stripeProcessor = p;
    }

    public void pay(double amount) {
        stripeProcessor.makePayment(amount);
    }
}

public static void main(…) {
    StripeProcessor stripe = new StripeProcessor();
    PaymentProcessor p = new StripeAdapter(stripe);
    handlePurchase(p);
}
public void handlePurchase(PaymentProcessor p) {
    p.pay(200.0);
}
```

What did we achieve?

- Ability to use the same interface PaymentProcessor for Stripe payments
- Once you initialize it, that's enough
- No need to create special if-checks where the PaymentProcessor is used

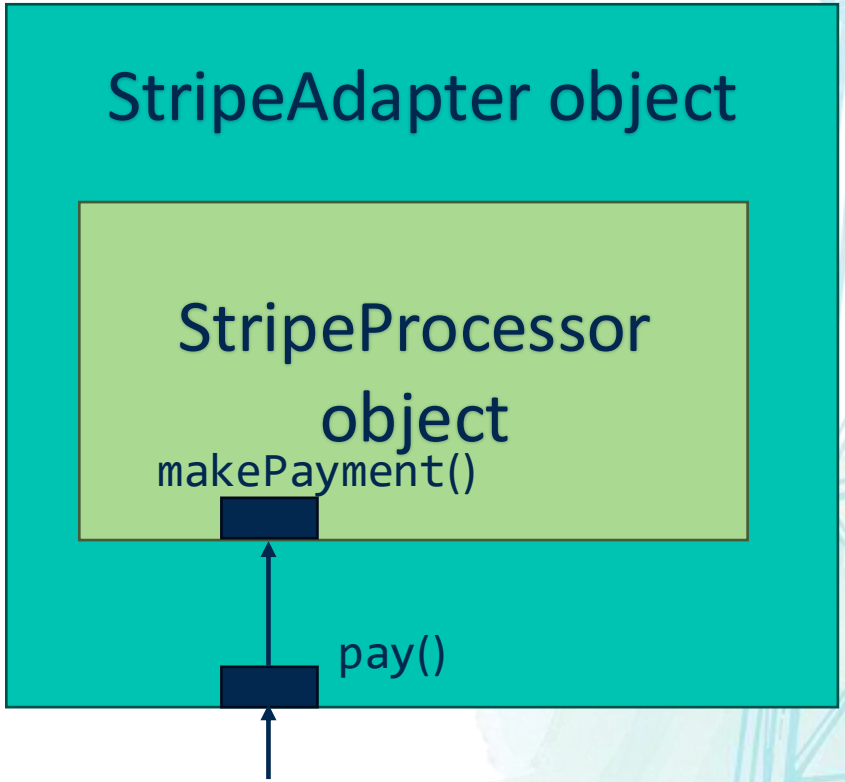
```
interface PaymentProcessor
{
    void pay(int amount);
}

class PaypalProcessor implements PaymentProcessor {
    void pay (int amount) {
        // paypal functionality
    }
}
```

```
public static void main(…) {
    PaymentProcessor p = new PaypalProcessor();
    StripeProcessor sp = new StripeProcessor();
    handlePurchase(p, sp, true);
}
public void handlePurchase(PaymentProcessor p,
StripeProcessor sp, bool isStripe) {
    if (isStripe) {
        sp.makePayment(2000);
    } else {
        p.pay(2000);
    }
}
```

Bad design:
very error-prone

Payment interface adapter

- Wrap the `StripeProcessor` object (adaptee) in an Adapter
- The `StripeAdapter` implements the `PaymentProcessor` interface
  - And internally invokes the adaptee's `makePayment()` method
- Can use `StripeAdapter` wherever `PaymentProcessor` is used



Proxy design pattern

Use case: add logging functionality

- Goal: invoke `log()` method before every method invocation on a Person object, without changing other method signatures

```
class Person {
    public String walk() { … }
    public void talk() { … }
}

public static void main() {
    Person p = new Person();
    animate (p);
}

static void animate(Person p) {
    p.walk(); p.talk();
}
```

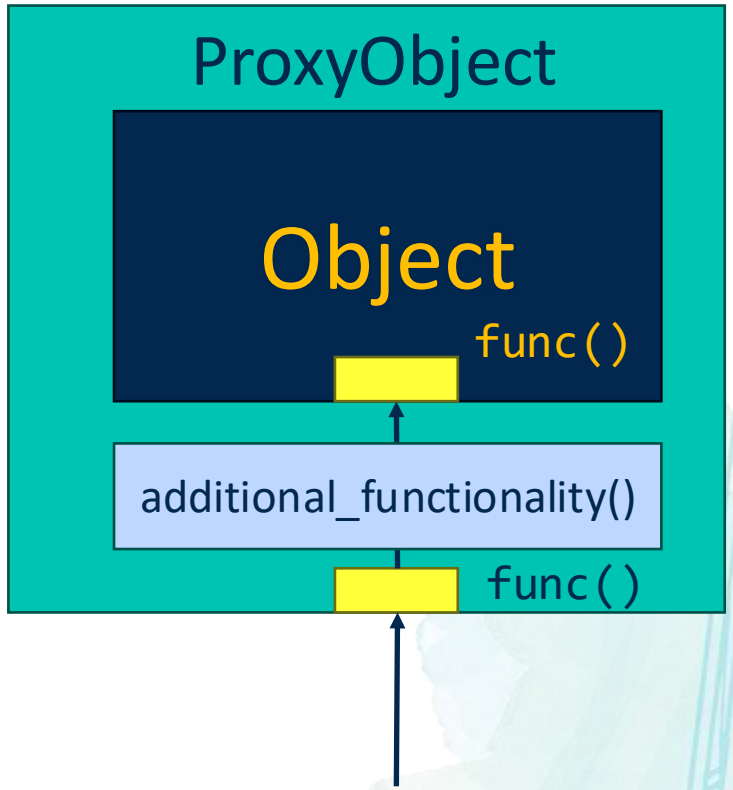
Use case: add logging functionality

- Goal: invoke `log()` method before every method invocation on a Person object , without changing other method signatures
- But can't modify the Person class
  - A third-party library provided the class
  - All person objects are not loggable

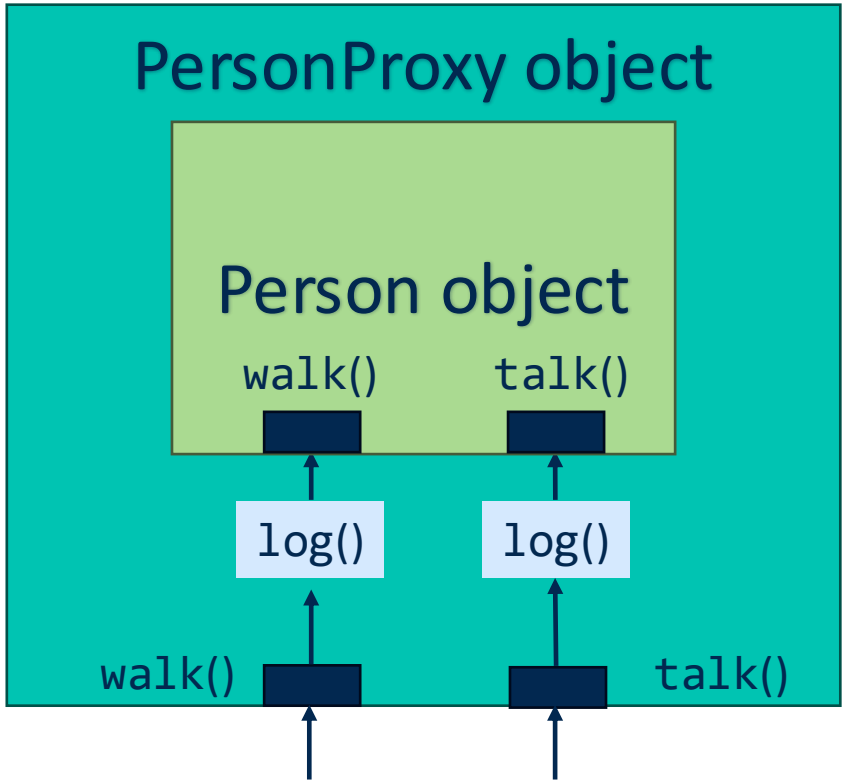
```
class Person {
    public String walk() { log(); // … }
    public void talk() { log(); // … }
}

public static void main() {
    Person p = new Person();
    animate (p);
}

static void animate(Person p) {
    p.walk(); p.talk();
}
```

Proxy design pattern - what is a proxy object?

- A proxy is *also* wrapper around the original/target object
- ***Mediates access*** to the inner object and is used ***in place of*** inner object
- The user accesses the proxy object instead of the original target object
- Proxy performs some additional logic, then forwards the request to the target object



Proxy object wraps and extends target class

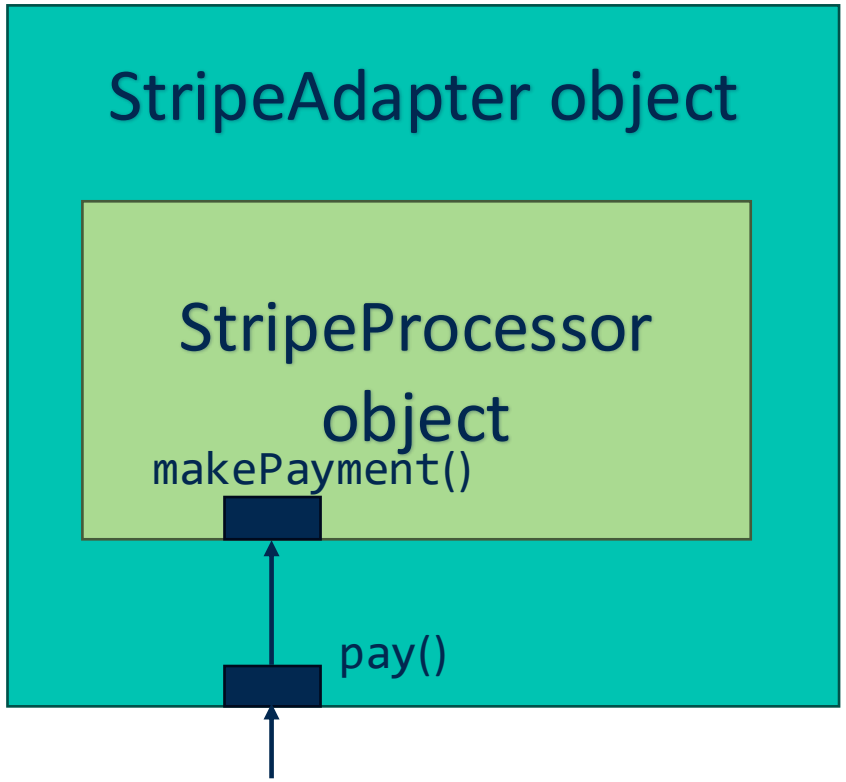



```
class Person {
     public String walk() { … }
     public void talk() { … }
}

class PersonProxy extends Person{
     private Person person; // Wrap person obj

     public log() { S.o.p(…); }

     public void walk() {
          log();
          person.walk();
     }
     public void talk() {
          log();
          person.talk();
     }
}
public static void main() {
     Person p = new PersonProxy();
     animate (p);
}

static void animate(Person p) {
     p.walk(); p.talk();
}
```

Proxies intercept the method invocations

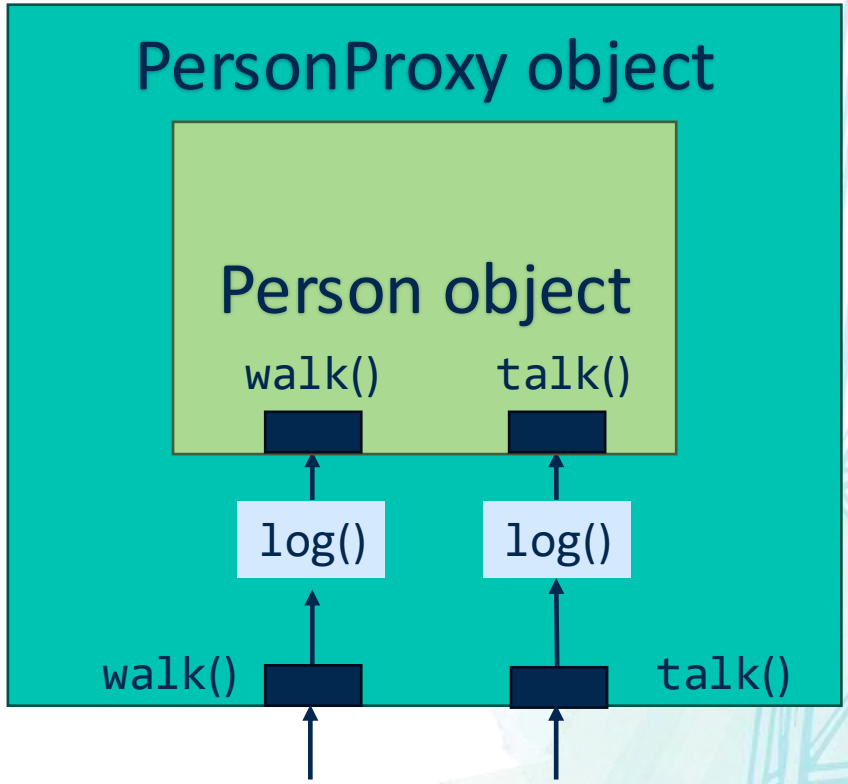
Adapter vs. proxy design pattern

Implements/extends ***target*** interface/class



Adapts the adaptee API to target's

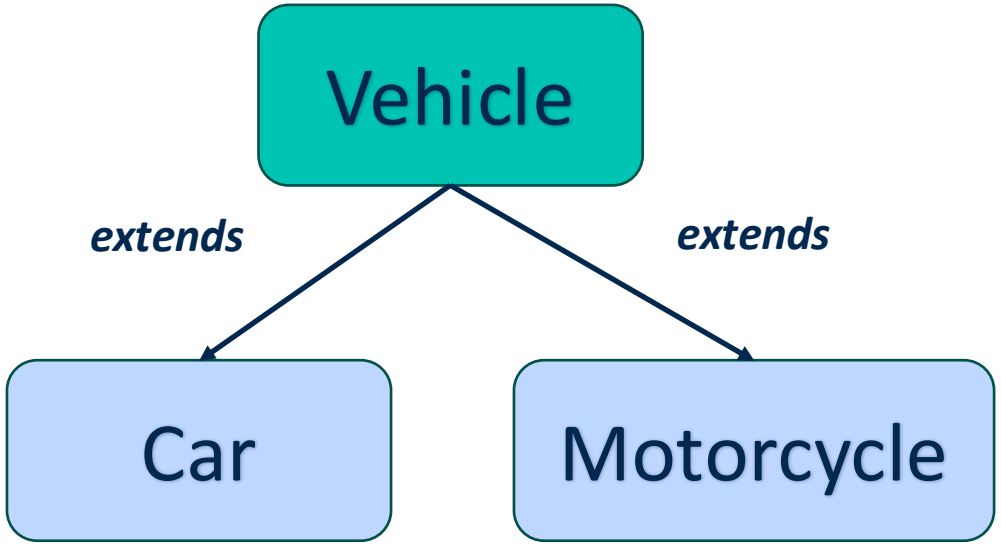
Implements/extends interface/class of ***inner*** obj



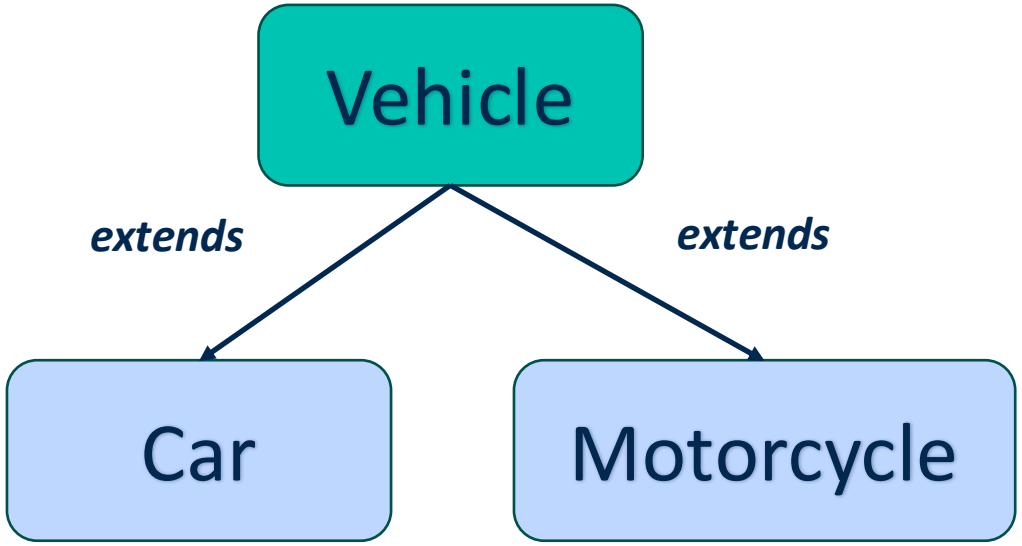
Has the same API as inner object class

Bridge design pattern

Case study: vehicle class hierarchy

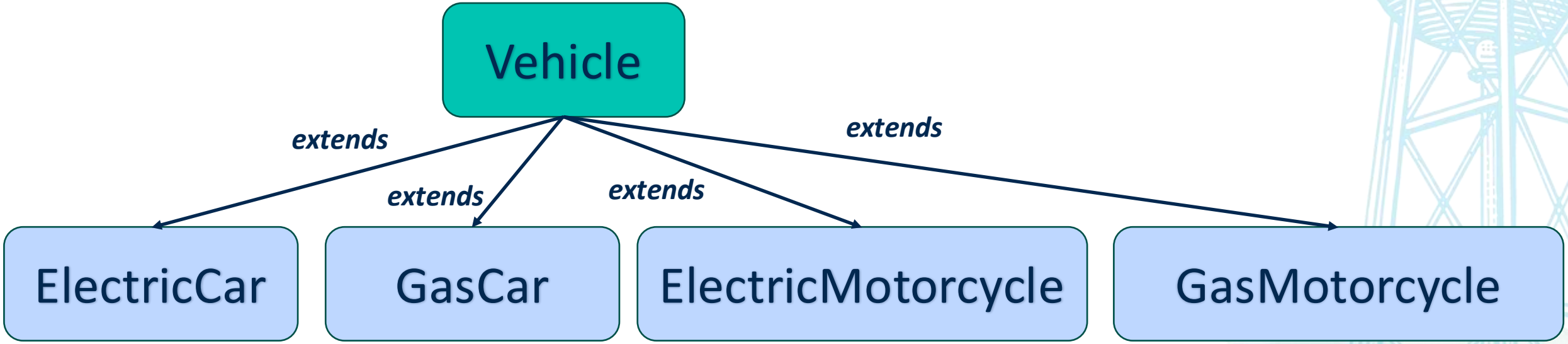


Case study: vehicle class hierarchy



A car can be either gas or electric. A motorcycle can also be either gas or electric

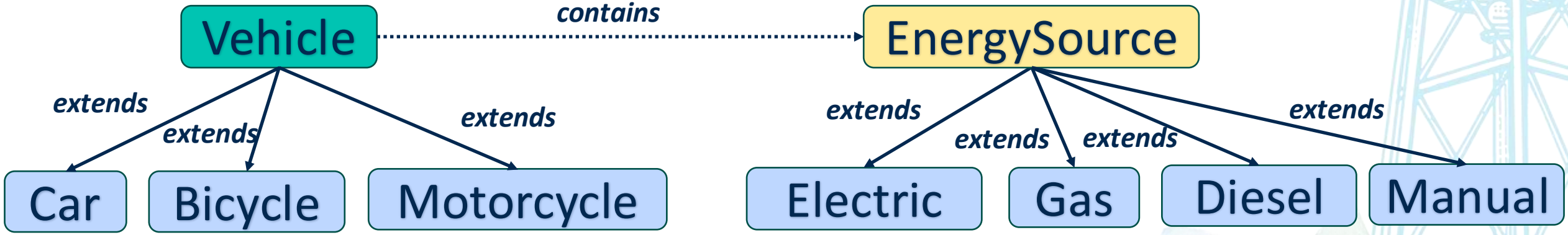
Case study: vehicle class hierarchy



What if we want to add diesel vehicles?

What about other vehicles like Bicycle which are manually operated?

Case study: vehicle class hierarchy



Abstraction*

Implementation

*Not an abstract class; just means that it abstracts the implementation details

Case study: vehicle class hierarchy

- Supports adding both new vehicles and new energy sources
- No class explosion

```
class Vehicle {
    private EnergySource energySource;
}

class Car extends Vehicle { }

class Bicycle extends Vehicle { }

class EnergySource {}

class Electric extends EnergySource {}

class Manual extends EnergySource { }

class Diesel extends EnergySource { }
```

Bridge design pattern

- Split a large class or group of classes into two separate hierarchies – abstraction and implementation – which can be developed independently of each other
- When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance
- Inheritance binds an implementation to an abstraction permanently
  - Need more flexibility in many cases

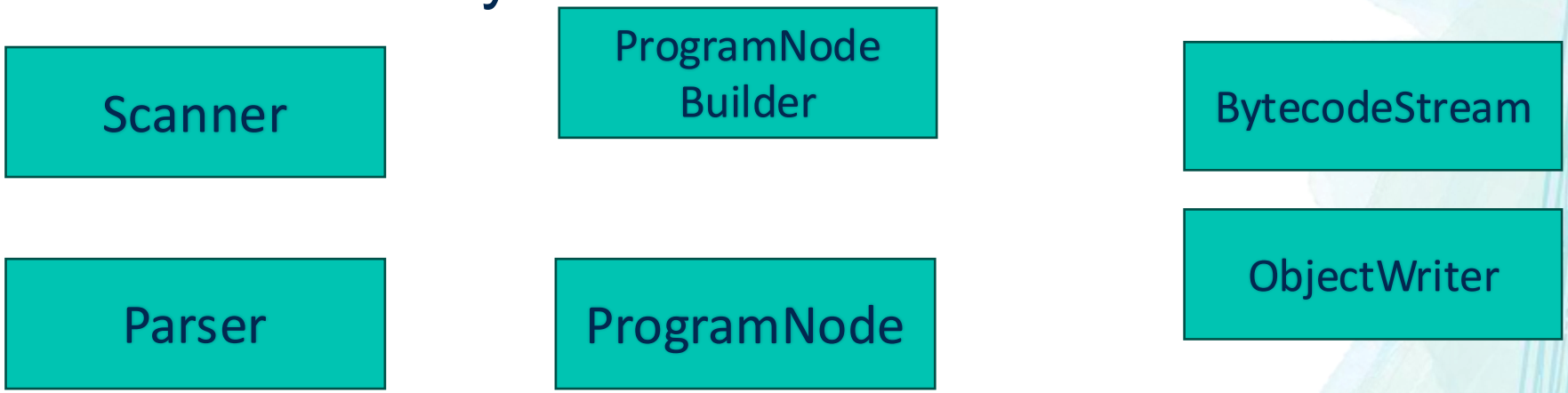
Characteristics

- Decouples interface (abstraction) and implementation
- The “implementation” is not bound to the abstraction
  - For example, you can turn a manual bike to an electric
- Improved extensibility

Facade design pattern

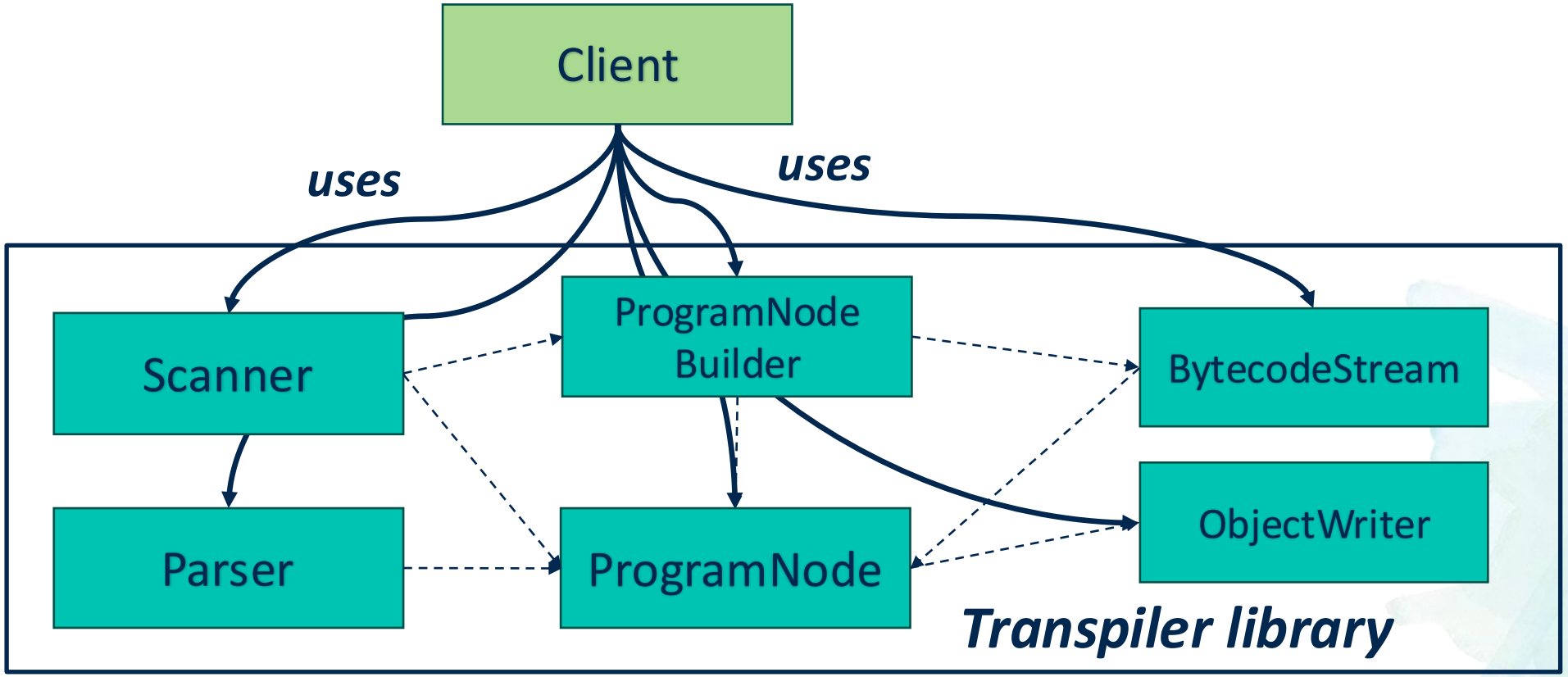
Case study – transpilation library

- Façade pattern useful for accessing ***really*** complex software systems with many sub-systems
- C-to-Rust transpilation ***library***
  - Allows clients to use it and add compilation capabilities to their software
- Consists of different subsystems and classes



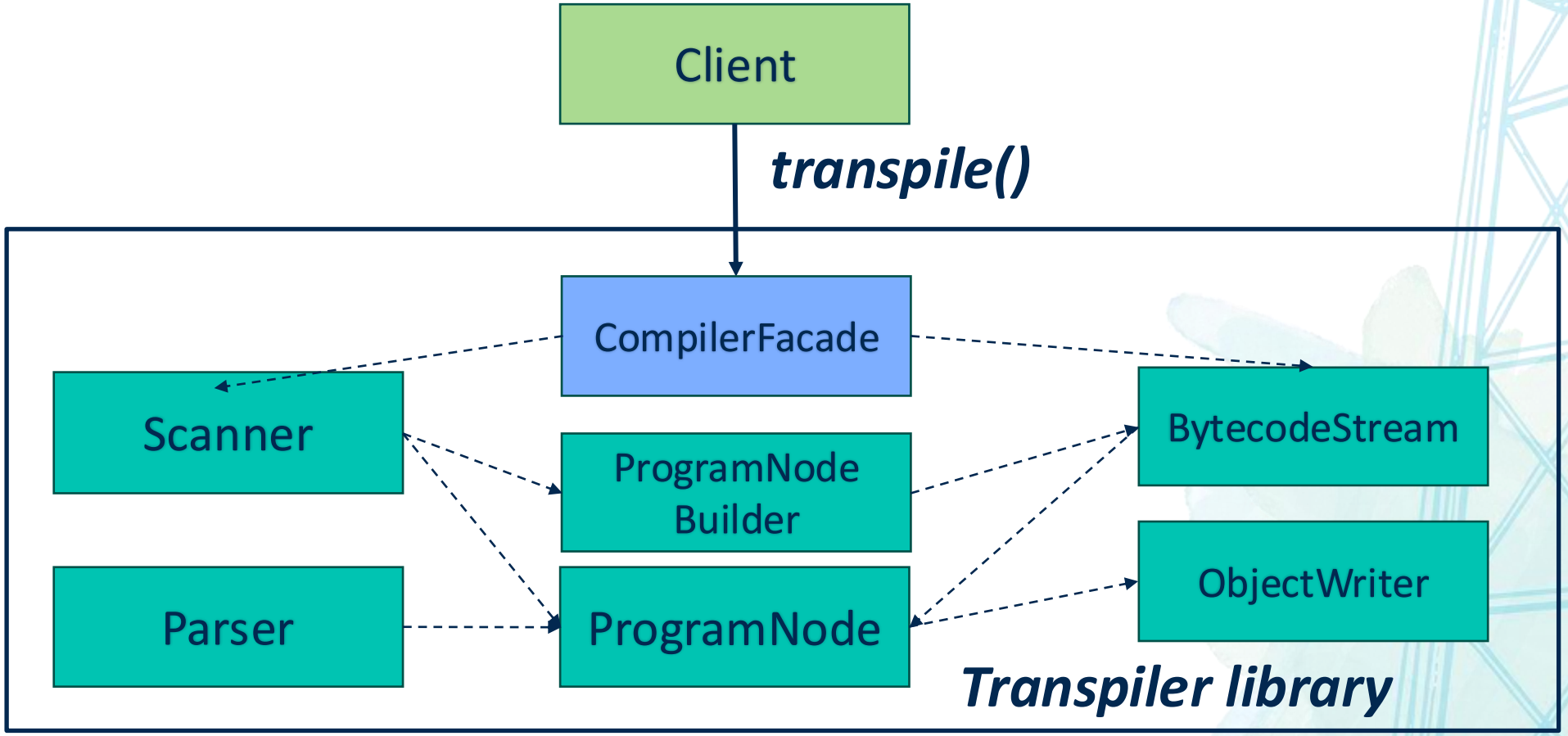
Case study – transpilation library

- The client can access each of these subcomponents individually



Case study – transpilation library

- A transpiler library can expose a “facade” class



Characteristics

- Provide a unified interface to a set of interfaces in a subsystem
- It shields clients from subsystem components
  - Reduces the number of objects that clients deal with
  - Makes the subsystem easier to use
- Weak coupling between the subsystem and the client
  - Subsystems can change without the client having to worry about it

Decorator design pattern

Case study – fraud detecting payment processor

- Check for fraud before paying
- Don't want to change the `complete_transaction` method
- Don't want to or **can't** modify `PaypalProcessor`

```
interface PaymentProcessor
{
    void pay(double amount);
}
```

```
class PaypalProcessor implements PaymentProcessor
{
    void pay (double amount) {
        // paypal functionality
    }
}
```

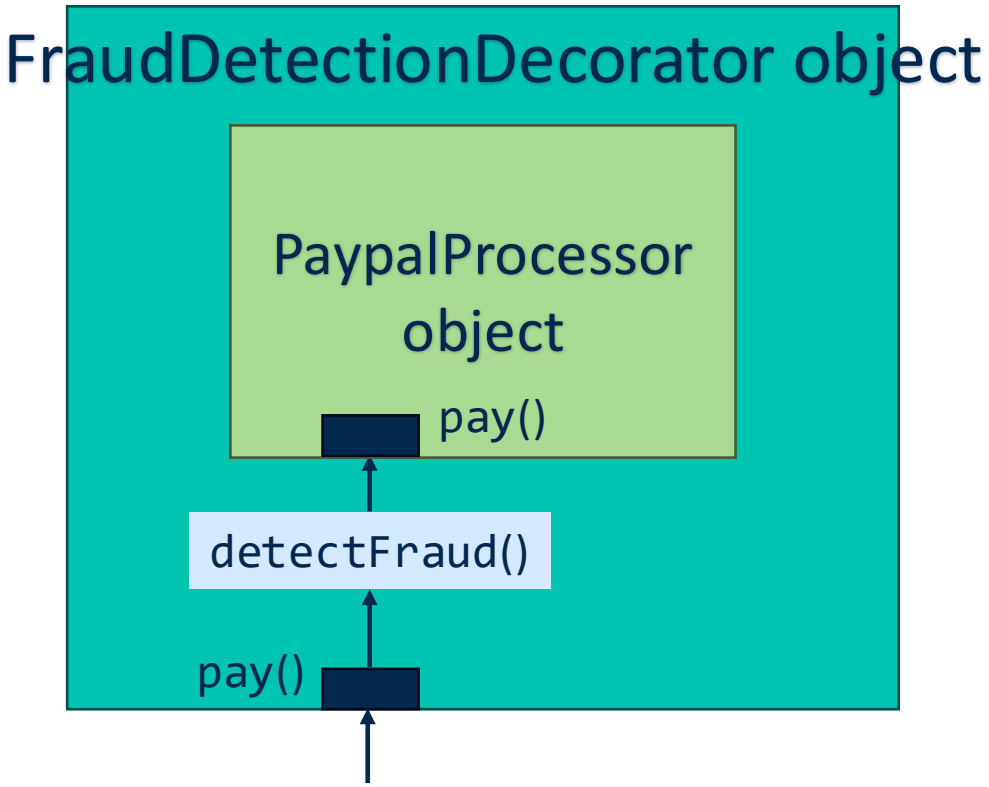
Detect fraud

```
public static void main(String[] args) {
    PaymentProcessor paypalProc = new
PaypalProcessor();
    complete_transaction(paypalProc);
}

static void complete_transaction(PaymentProcessor
p) { /// 100 uses of p }
```

Decorator design pattern

- Can attach additional functionalities to an object dynamically
- Basic idea: enclose the object (decoratee) in another object (decorator) that adds the additional functionality

Decorators visually



How to use FraudDetectionDecorator object in place of PaymentProcessor?

Fraud detection decorator for payment processor

- Create an abstract class for the decorator that wraps the inner payment processor
  - And implements/extends PaymentProcessor

```
interface PaymentProcessor
{
    void pay(double amount);
}

class PaypalProcessor implements PaymentProcessor {
    void pay (double amount) {
        // paypal functionality
    }
}

abstract class PaymentProcessorDecorator implements
PaymentProcessor {
    protected PaymentProcessor wrappedProcessor;
    public PaymentProcessorDecorator(PaymentProcessor
processor) {
        this.wrappedProcessor = processor;
    }
    @Override
    public void pay(double amount) {
    wrappedProcessor.pay(amount);
    }
}
```

Fraud detection decorator for payment processor

- Extend the abstract decorator as the `FraudDetectionDecorator`

```
abstract class PaymentProcessorDecorator implements
PaymentProcessor {
    protected PaymentProcessor wrappedProcessor;
    public PaymentProcessorDecorator(PaymentProcessor
processor) {
        this.wrappedProcessor = processor;
    }
    @Override
    public void pay(double amount) {
    wrappedProcessor.pay(amount);
    }
}
```

```
class FraudDetectionDecorator extends
PaymentProcessorDecorator {
    @Override
    public void pay(double amount) {
        detectFraud();
        wrappedProcessor.pay(amount);
    }
     private void detectFraud() { //… }
}
```

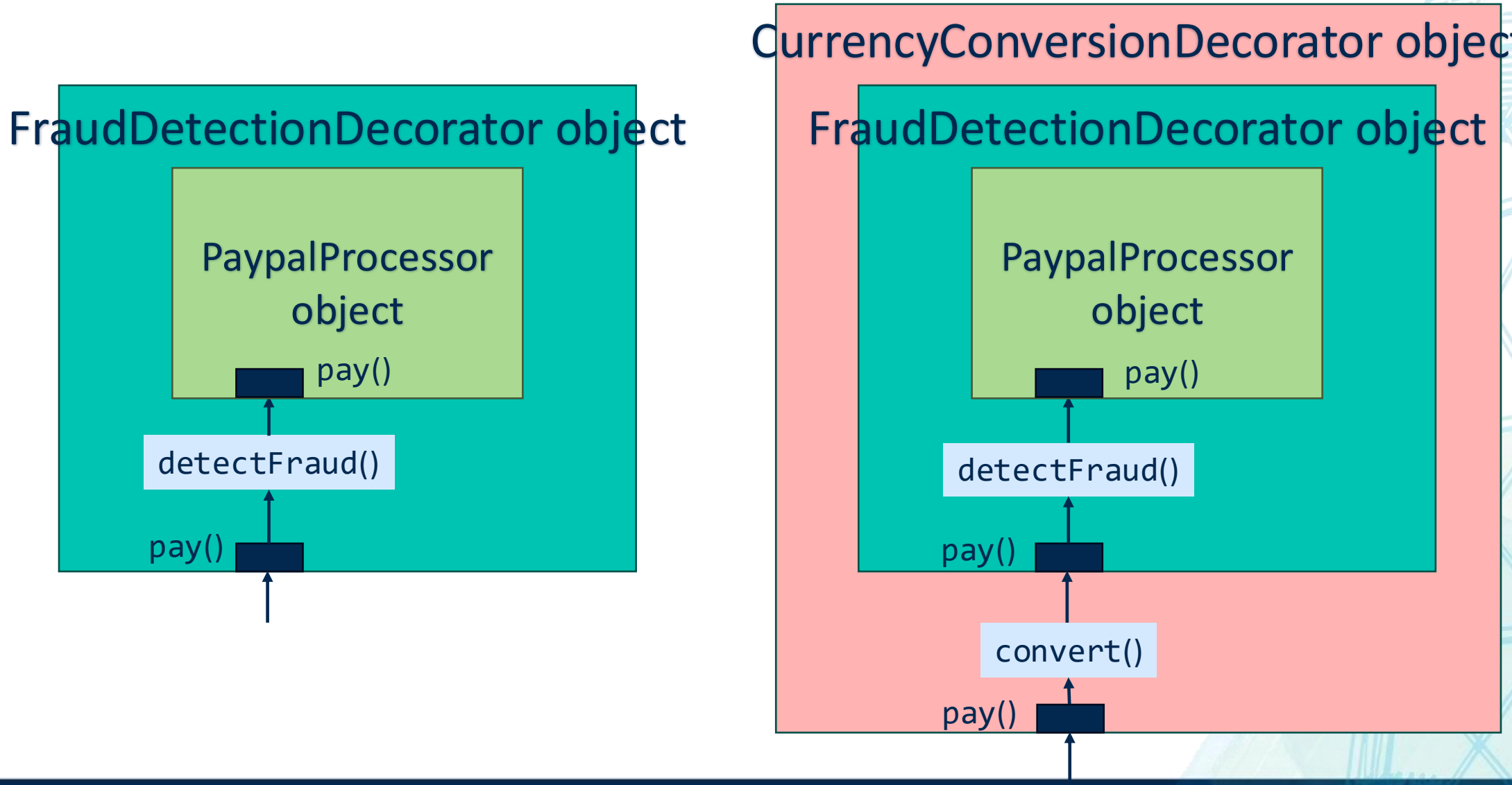
Fraud detection decorator for payment processor

- Wrap `PaymentProcessor` objects in `FraudDetectionDecorator` objects
- Use the `FraudDetectionDecorator` objects in any place where a `PaymentProcessor` is needed

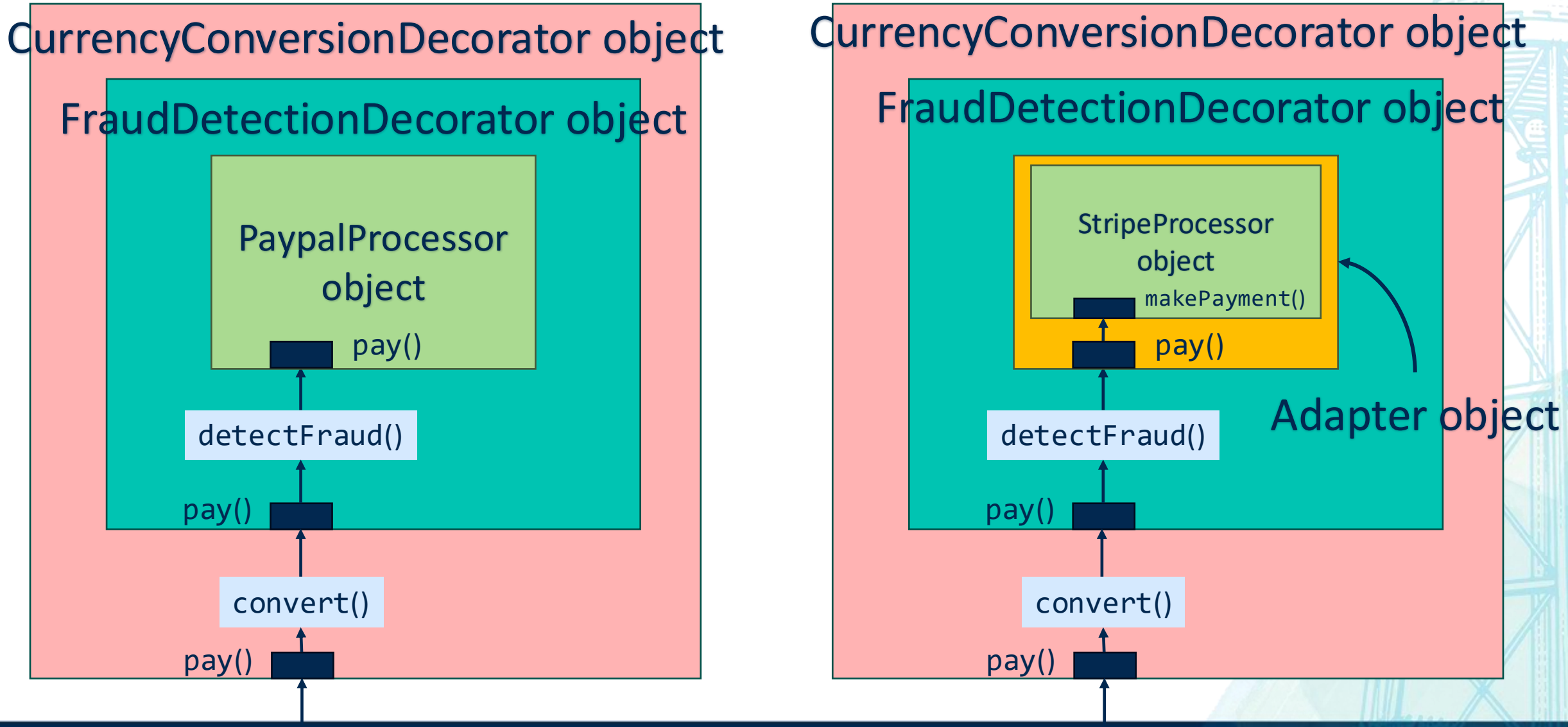
```
abstract class PaymentProcessorDecorator implements
PaymentProcessor {
    // ... snip
}

class FraudDetectionDecorator extends
PaymentProcessorDecorator {
    // ... snip
}
```

```
public static void main(String[] args) {
    PaypalProcessor paypalProc = new
PaypalProcessor();
    FraudDetectionDecorator fraudProc = new
                FraudDetectionDecorator(paypalProc);
    complete_transaction(fraudProc);
}

static void complete_transaction(PaymentProcessor p)
{ /// 100 uses of p }
```

Decorators visually



Combining adapter and decorator patters



Isn’t this a proxy?? [Revisit next module]

https://www.cs.unc.edu/~stotts/GOF/hires/pat4gfso.htm

- Although decorators can have similar implementations as proxies, decorators have a different purpose. **A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.**
- **Proxies vary in the degree to which they are implemented like a decorator.** A protection proxy might be implemented exactly like a decorator. On the other hand**, a remote proxy will not contain a direct reference to its real subject but only an indirect reference**, such as "host ID and local address on host." A virtual proxy will start off with an indirect reference such as a file name but will eventually obtain and use a direct reference.

Non-ideal alternate implementations: subclass?

- Create subclass that first detects fraud and then pays
- ***Problems?***

```
interface PaymentProcessor
{
    void pay(double amount);
}

class PaypalProcessor implements PaymentProcessor
{
    void pay (double amount) {
        // paypal functionality
    }
}

class FraudDetectingPaypalProcessor extends
PaypalProcessor {
    void detectFraud() { … }
    void pay (double amount) {
        detectFraud();
        super.pay(amount);
    }
}
```

Problem 1: breaks SRP

- Single Responsibility Principle – FraudDetectingPaypalProcessor does two things
  - Detects frauds
  - Does Paypal payments

```
interface PaymentProcessor
{
    void pay(double amount);
}

class PaypalProcessor implements PaymentProcessor
{
    void pay (double amount) {
        // paypal functionality
    }
}
```

```
class FraudDetectingPaypalProcessor extends
PaypalProcessor {
    void detectFraud() { … }
    void pay (double amount) {
        detectFraud();
        super.pay(amount);
    }
}
```

Problem 2: class explosion

- What if we had BitcoinPaymentProcessor?
- And then had FraudDetectingBitcoinPaymentProcessor
- For each payment processor needs one fraud detecting class
- What if you want to add another functionality in addition to fraud detection?
  - Results in class explosion

```
interface PaymentProcessor { }

class PaypalProcessor implements PaymentProcessor
{}

class FraudDetectingPaypalProcessor extends
PaypalProcessor {}

class BitcoinProcessor implements
PaymentProcessor {}

class FraudDetectingBitcoinProcessor extends
BitcoinProcessor {}

class AnotherProcessor implements
PaymentProcessor() {}

class FraudDetectingAnotherProcessor extends
AnotherProcessor { }
```

Characteristics

- Decorators also make it easy to add a functionality twice
  - How would you perform fraud detection twice?
- Avoids feature-laden classes high up in the hierarchy
- Provides a pay-as-you-go approach
- Disadvantage – can be hard to learn and debug a system which uses many decorators

Design pattern classification

Creational patterns

- Control how objects are created
- E.g, Factory, Singleton

Structural patterns

- Control how objects and classes are composed
- Deal with object relationships
- E.g., Adapter, Proxy Decorator, Bridge, Façade

Behavioral patterns

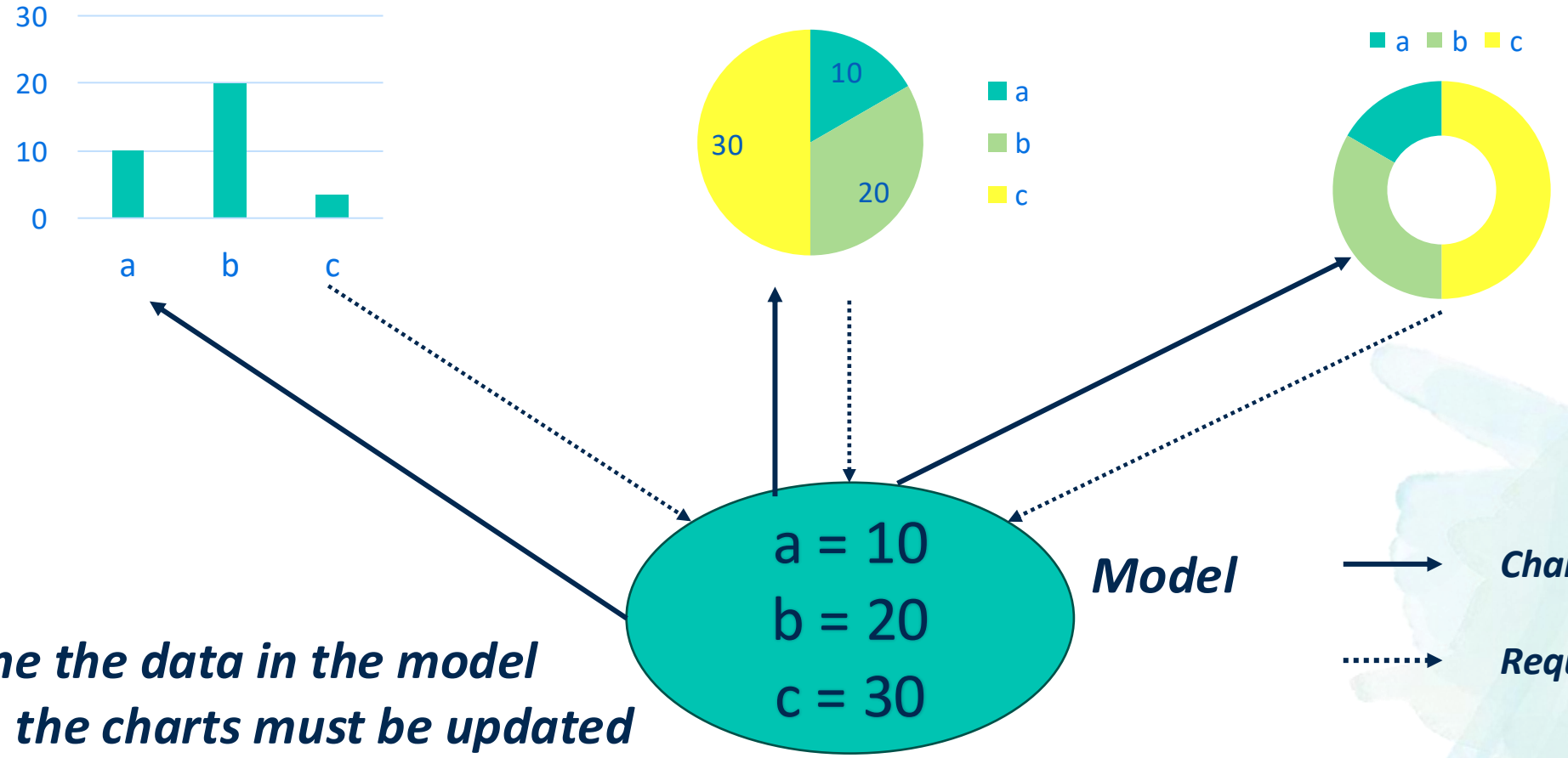
- Control how objects distribute responsibilities
- Template method, State, Observer, Visitor

Behavioral pattern

- Concerned with algorithms and the assignment of responsibilities between objects
- Behavioral class patterns use inheritance to distribute behavior between classes
- Behavioral object patterns use object composition

Observer design pattern

Case study - Graphical views for application data

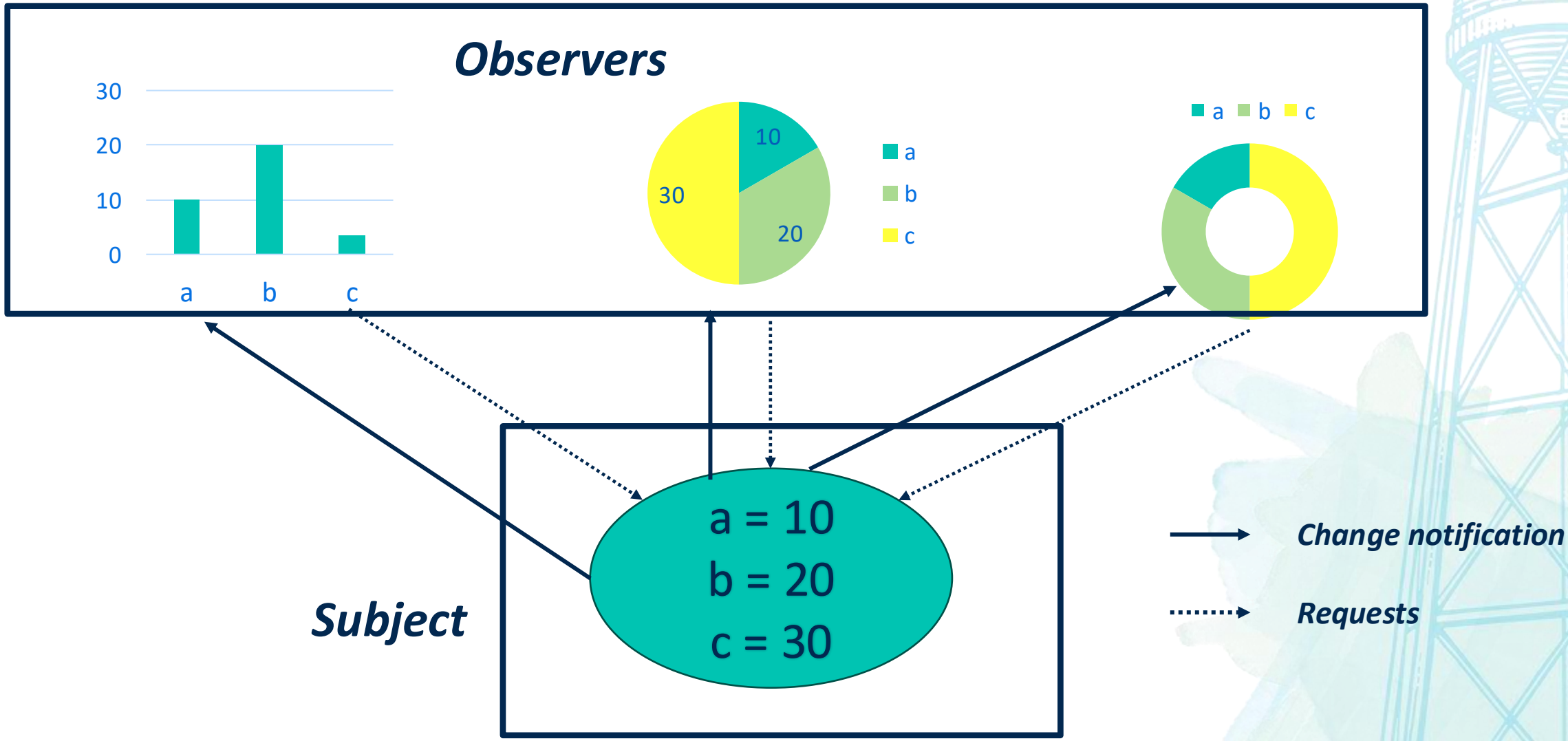


Every time the data in the model changes, the charts must be updated

Observer designer pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- The object being observed is called “subject”
- The object doing the observing is called “observer”

Observer design pattern



Graphical views for application data

- The subject contains a list of observers and provides an `add()` and `remove()` API for observers to register and deregister
- Also provides a `notify()` API to notify the observers
  - The `notify()` API is invoked when the data is updated

```
class DataModel {
    private int a, b, c;
    private List<Observer> observers = new ArrayList<>();

    // Add observer
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    // Remove observer
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

     // Notify observers
    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(a, b, c);
        }
    }
    // Set data and notify observers
    public void setData(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
        notifyObservers();
    }
}
```

Graphical views for application data

- Observers extend the `Observer` interface

```
// Observer Interface
interface Observer {
    void update(int a, int b, int c);
}
```

```
// Concrete Observer
class BarChart implements Observer {
    @Override
    public void update(int a, int b, int c) {
        // Render the bar chart here
    }
}

class PieChart implements Observer {
    @Override
    public void update(int a, int b, int c) {
        // Render the pie chart here
    }
}
```

Graphical views for application data

- The observers register with the subject
- When the subject's data changes, the observers are notified

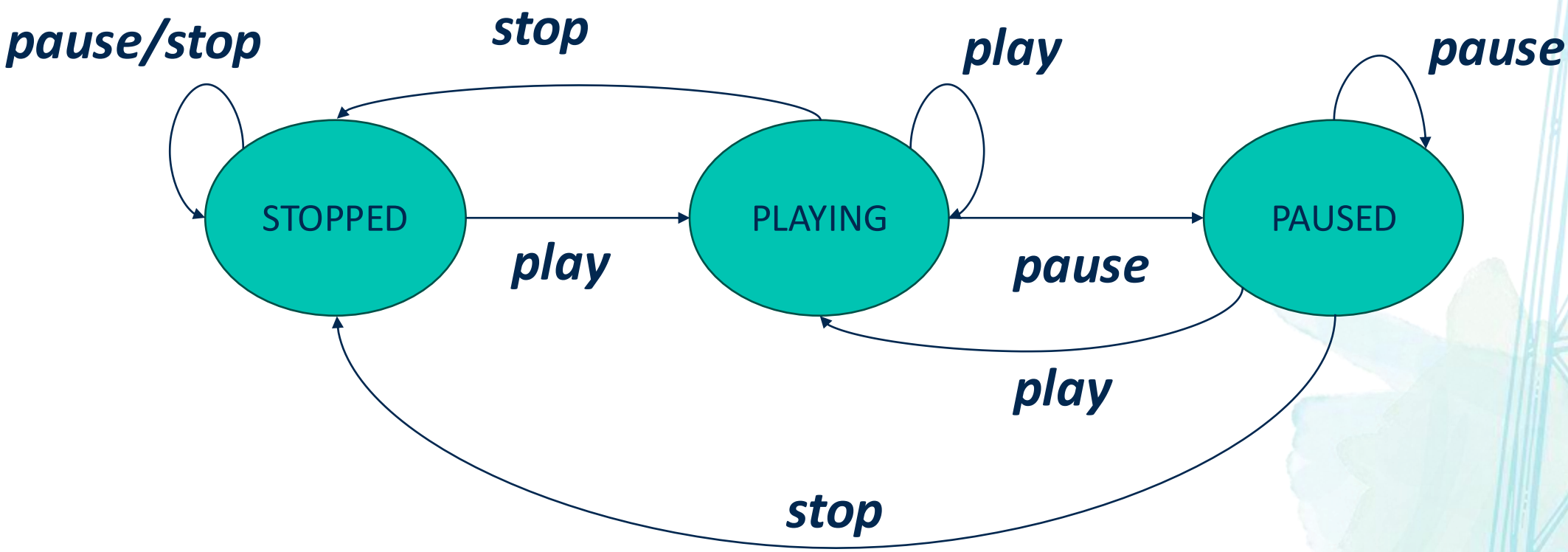
```
public class ObserverPatternExample {
    public static void main(String[] args) {
        // Create the subject
        DataModel dataModel = new DataModel();

        // Create observers
        BarChart barChart = new BarChart();
        PieChart pieChart = new PieChart();

        // Attach observers to the subject
        dataModel.addObserver(barChart);
        dataModel.addObserver(pieChart);

        // Update data
        dataModel.setData(10, 20, 30);
        dataModel.setData(15, 25, 35);
    }
}
```

Characteristics

- Abstract coupling between the subject and observers
- Support for broadcast communication
- Possible disadvantage
  - Can cause cascading updates without the subject's knowledge
  - Data model updates data can trigger many other observers to update

State design pattern

Case study: media player state diagram



Media player states

- A media player can be in three states
  - Playing
  - Paused
  - Stopped
- Behavior of actions such as `play()`, `stop()`, and `pause()` depends on the state
- How would we design this without a design pattern?

Without design pattern

- MediaPlayer class
  - `state` field
  - Methods for `start()`, `stop()`, and `pause()`
    - Behavior depends on the state
    - Multiple `if-else` branches

What if you miss one state?

```
class MediaPlayer {
    private String state; // STOP, PLAY, PAUSE

    public MediaPlayer() { state = “STOP”; }

    public void start() {
        if (state == “STOP”) {
            System.out.println(“Starting”);
            state = “PLAY”;
        } else if (state == “PLAY”) {
            System.out.println(“Already playing.”);
        } else if (state == “PAUSE”) {
            S.o.p(“Resuming from pause.”);
            state = “PLAY”;
        }
    }

    public void pause() {
        if (state == “STOP”) {
            S.o.p(“Player stopped, can’t pause.”);
        } else if (state == “PLAY”) {
            S.o.p(“Player paused”);
            state = “PAUSE”;
        } else if (state == “PAUSE”) {
            S.o.p(“Player already paused. Can’t pause
again.”);
        }
    }
}
```

With state design pattern

- Interface `MediaPlayerState`
  - Concrete subclasses for `PlayState`, `PauseState`, and `StopState`

Now what if you miss one state?

```
interface MediaPlayerState {
    void play(MediaPlayer context);
    void pause(MediaPlayer context);
    void stop(MediaPlayer context);
}
```

```
class PlayState implements MediaPlayerState {
    @Override
    public void play(MediaPlayer context) {
        System.out.println("Media is already
playing.");
    }

    @Override
    public void pause(MediaPlayer context) {
        System.out.println("Pausing media...");
        context.setState(new PausedState());
    }

    @Override
    public void stop(MediaPlayer context) {
        System.out.println("Stopping media...");
        context.setState(new StoppedState());
    }
}
// Similarly, for PauseState and StopState
```

With state design pattern

- `MediaPlayer` class maintains a reference to the current state
- Delegates behavior to the current state
- The state transition logic for each state is encapsulated in the state object
- No messy `if-else` statements

```
class MediaPlayer {
    private MediaPlayerState currentState;

    public MediaPlayer() {
        // Default state is "Stopped"
        this.currentState = new StoppedState();
    }

    public void setState(MediaPlayerState state) {
        this.currentState = state;
    }

    public void play() {
        currentState.play(this);
    }

    public void pause() {
        currentState.pause(this);
    }

    public void stop() {
        currentState.stop(this);
    }
}
```

State design pattern

- Allow an object to alter its behavior when its internal state changes
- The object will *appear* to have changed its class

Characteristics

- Localizes state-specific behavior and partitions behavior for different states
- Makes state transitions explicit
  - When state is represented by internal values, it's state representation has no explicit representation
  - Introducing state objects makes the state explicit

Template method design pattern

Case study: find all paths from source to destination

- Given a graph represented as an adjacency list in a file, find all paths from a source node to a destination node, and print them
- Solution structure
  - Initialize the graph
  - Traverse the graph using BFS or DFS
  - Print the results

Case study: find all paths from source to destination

- Abstract class GraphPathFinder
  - Concrete method `initializeGraph()`
  - Concrete method `collectResults()`
  - Abstract method `traverse()`

```
abstract class GraphPathFinder {
    protected Map<Integer, List<Integer>> graph = new HashMap<>();
    protected List<List<Integer>> allPaths = new ArrayList<>();
    protected int source, destination;

    // Template method
    public final void findPaths(int source, int destination) {
        this.source = source;
        this.destination = destination;
        initialize();
        traverseGraph();
        collectResults();
    }

    // Initialize the graph
    protected void initialize() {
        System.out.println("Initializing graph...");
         graph = readFile();
        // more initialization

    }

    // Abstract method for traversal (BFS or DFS)
    protected abstract void traverseGraph();

    // Collect and display the results
    protected void collectResults() {
      for (List<Integer> path : allPaths) {
            System.out.println(path);
        }
    }
}
```

Case study: find all paths from source to destination

- Concrete class `BFSPathFinder` and `DFSPathFinder` implement `traverseGraph()`

```
class BFSPathFinder extends GraphPathFinder {
    @Override
    protected void traverseGraph() {
        System.out.println("Using BFS for traversal...");
        Queue<List<Integer>> queue = new LinkedList<>();
        queue.add(Arrays.asList(source));

        while (!queue.isEmpty()) {
            List<Integer> path = queue.poll();
            int currentNode = path.get(path.size() - 1);

            if (currentNode == destination) {
                allPaths.add(new ArrayList<>(path));
            } else {
                for (int neighbor :
graph.getOrDefault(currentNode, new ArrayList<>())) {
                    List<Integer> newPath = new
ArrayList<>(path);
                    newPath.add(neighbor);
                    queue.add(newPath);
                }
            }
        }
    }
}
```

Template method design pattern

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- Allows subclasses to redefine certain steps of an algorithm without changing the algorithm's structure
- Typically used to design variants of the same algorithm

Characteristics

- Fundamental technique for code reuse
- Leads to an “inverted” control structure
  - Parent class calls the operations of the child class and not the other way around
- Template methods can be
  - *Hooks:* the base class provides empty implementation and subclasses can ***optionally*** implement them
  - *Abstract operations:* the base class provides abstract methods and the subclasses ***must*** implement them
- Commonly used for implementing algorithms that share a lot of same steps and differ only in a few steps

Visitor design pattern

Method overloading

- Methods can be overloaded
  - Methods with same name, but different argument count/type are treated distinctly by the compiler
- Method overloading is resolved at compile time
- Which version of f is invoked is decided at compile time

```
class A {
    public void f() { S.o.p("Hello"); }
    public void f(int a) { S.o.p("Hi" + a); }
    public void f(bool b) { if (b) { S.o.p("Bye"); }
}


public static void main(String args[]) {
    A objA = new A();
    objA.f();  // prints "Hello"
    objA.f(10); // prints "Hi 10"
    objA.f(false); // prints nothing
}
```

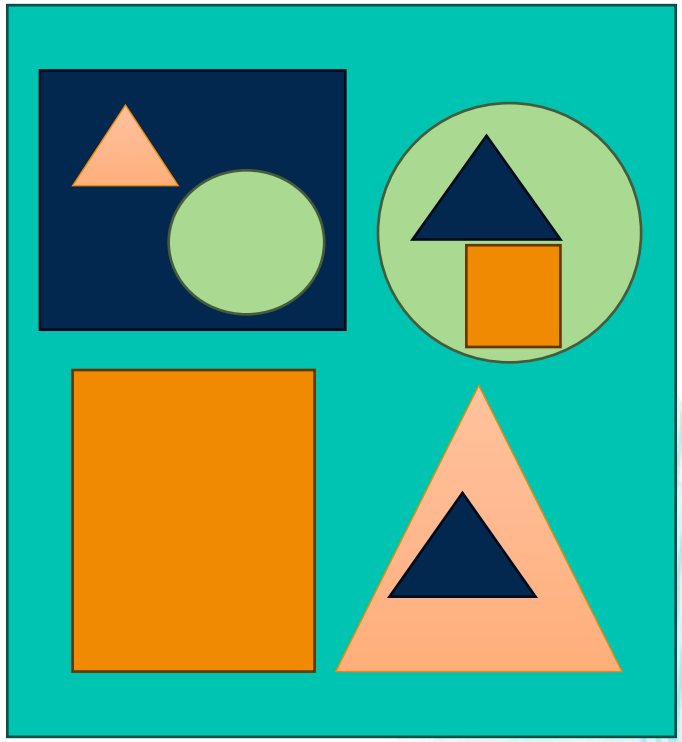
Method overloading

- Which version of `f` is invoked is decided at compile time
- For objects, the overloaded method invoked depends on the reference type
- Which `f` method of `objA` is invoked?
- Which `print` method is invoked?

```
class X {
    public void print() { S.o.p("1"); }
}

class Y extends X {
    public void print() { S.o.p("2"); }
}

class A {
   public void f(X x) { S.o.p("Hello"); }
   public void f(Y y) { S.o.p("Hi"); }
}


public static void main(String args[]) {
   X objX = new Y();
   A objA = new A();
   objA.f(objX);
   objX.print();
}
```

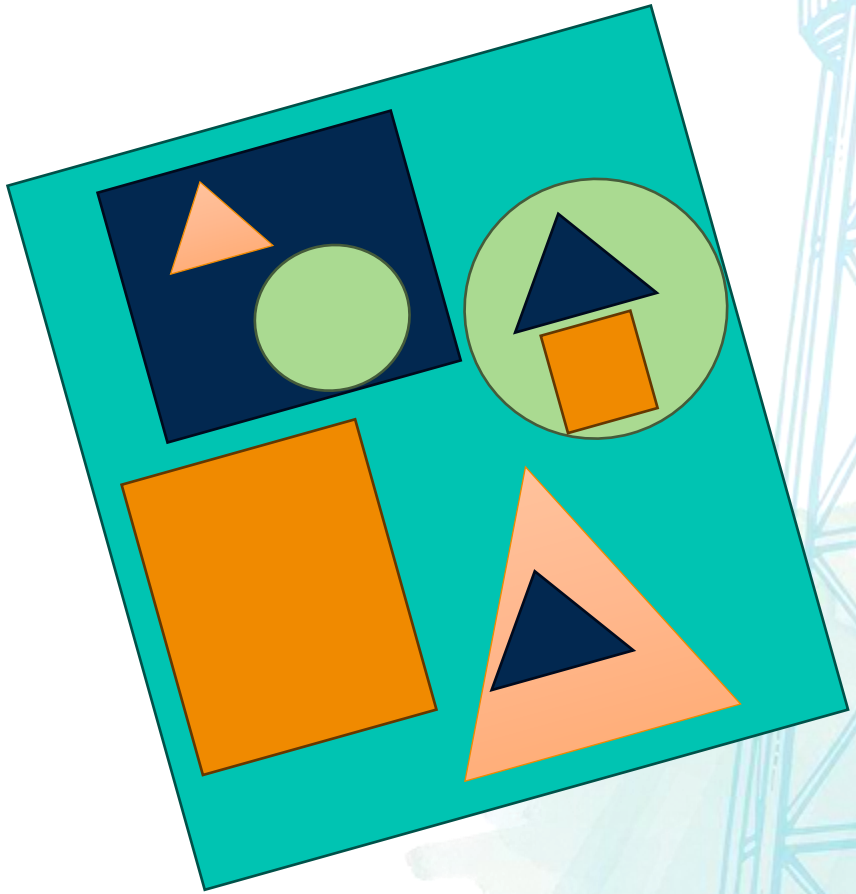
Case study – transform nested shapes

- A shape can be
  - `CompoundShape`
  - `Rectangle`
  - `Triangle`
  - `Circle`



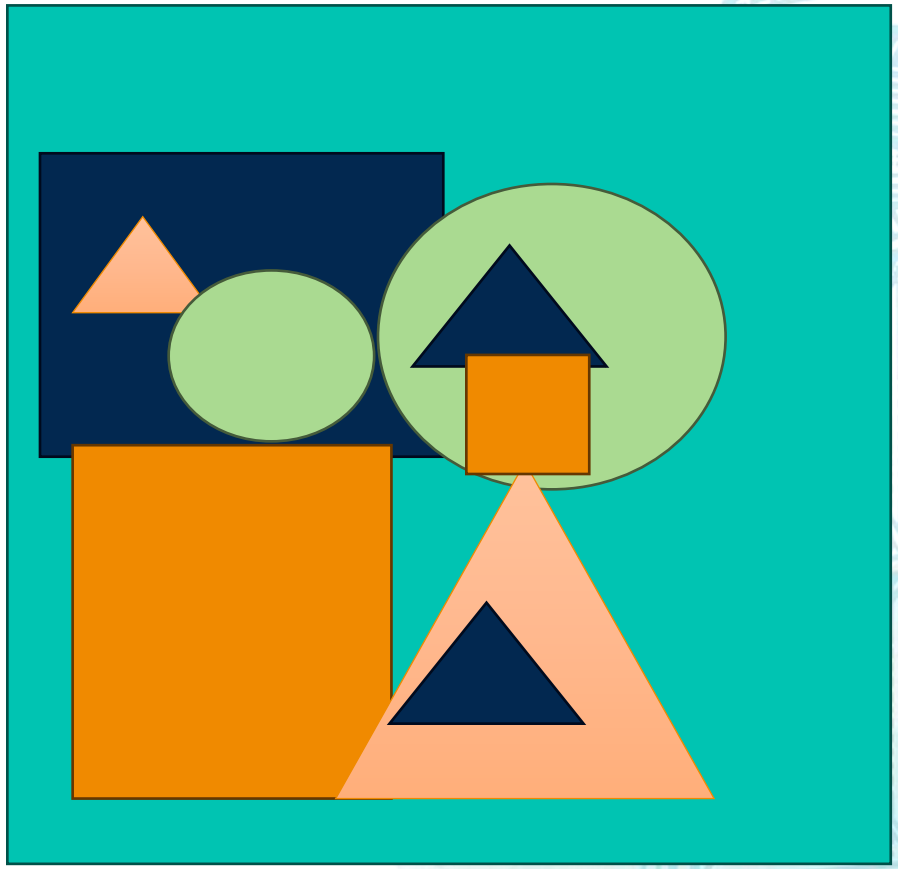
Case study – transform nested shapes

- Shape operations
  - `Rotate`



Case study – transform nested shapes

- Shape operations
  - `Scale`
  - Translate
  - … any other operation



Requirements

- Visit every shape in the shape object hierarchy and apply a custom operation on each shape visited
- Keep the operation logic **in a single class**
  - Operation logic should not be spread across many classes
  - Translation logic for all shapes should be in a single class
  - Rotation logic in a single class
  - And so on…

Visitor design pattern

- Encapsulate the operational logic in a “Visitor” class
- Abstract visitor class provides concrete implementation for the `visit` method for `CompoundShape`
- Concrete subclasses will override `visit` method with translation, scaling, rotation logic
- `accept` method is the glue that holds it all together

```
abstract class Visitor {
    abstract void visit(Circle circle);
    abstract void visit(Rectangle rectangle);

    void visit(CompoundShape c) {
        for (Shape shape: c.getShapeList()) {
            shape.accept(this);
        }
    }

}
```

Visitor design pattern so far…

```
abstract class Visitor
visit(Circle c);
visit(Rectangle r);
visit(Compound c) {
  for (Shape s:
   c.getShapeList() {
    s.accept(this);
  }
}
```

Visitor design pattern

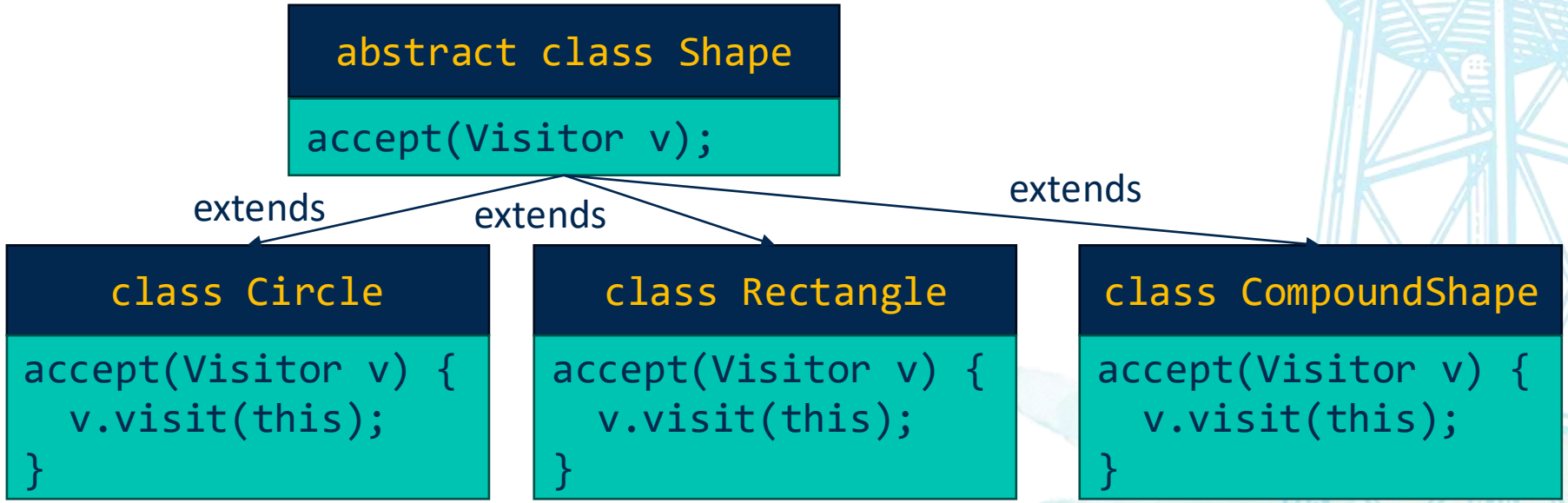
- Create abstract `accept` method in parent `Shape` class and create concrete implementations in subclasses
- The `accept` method invokes the visitor passed as method argument
  - And passes the `this` reference to it

```
abstract class Shape {
    abstract void accept(Visitor visitor);
}
```

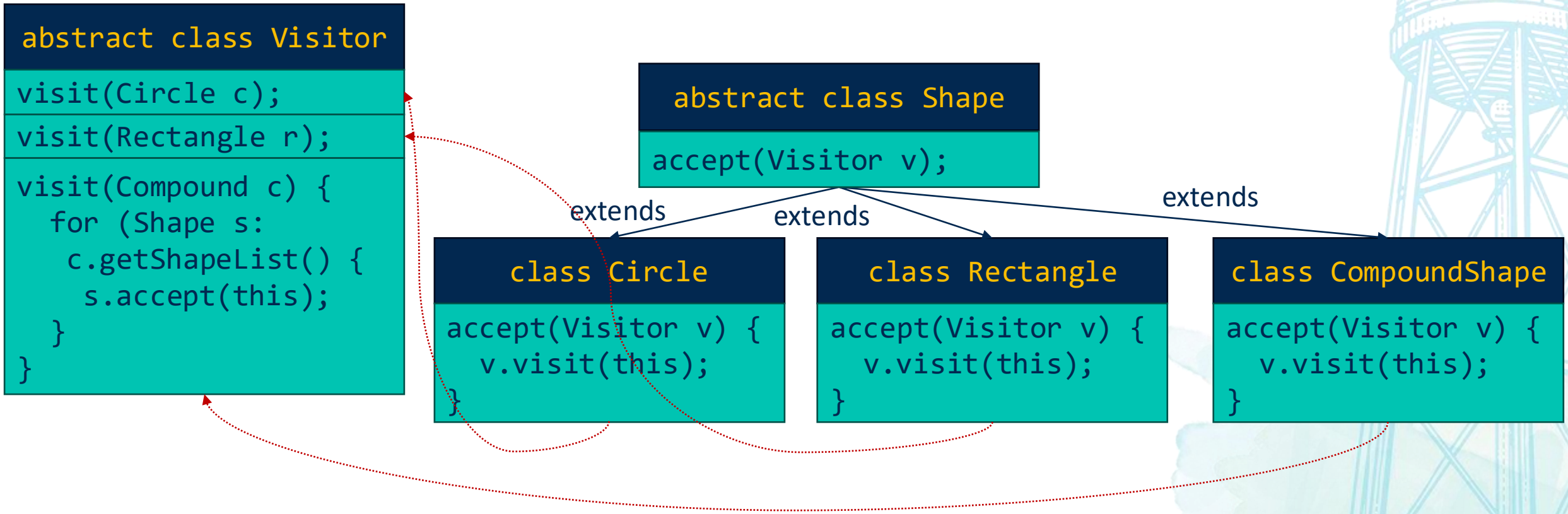
```
class Circle extends Shape {
    /// … fields and getters/setters
    void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Rectangle extends Shape {
    /// …
    void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

Visitor design pattern so far…

```
abstract class Visitor
visit(Circle c);
visit(Rectangle r);
visit(Compound c) {
  for (Shape s:
   c.getShapeList() {
    s.accept(this);
  }
}
```

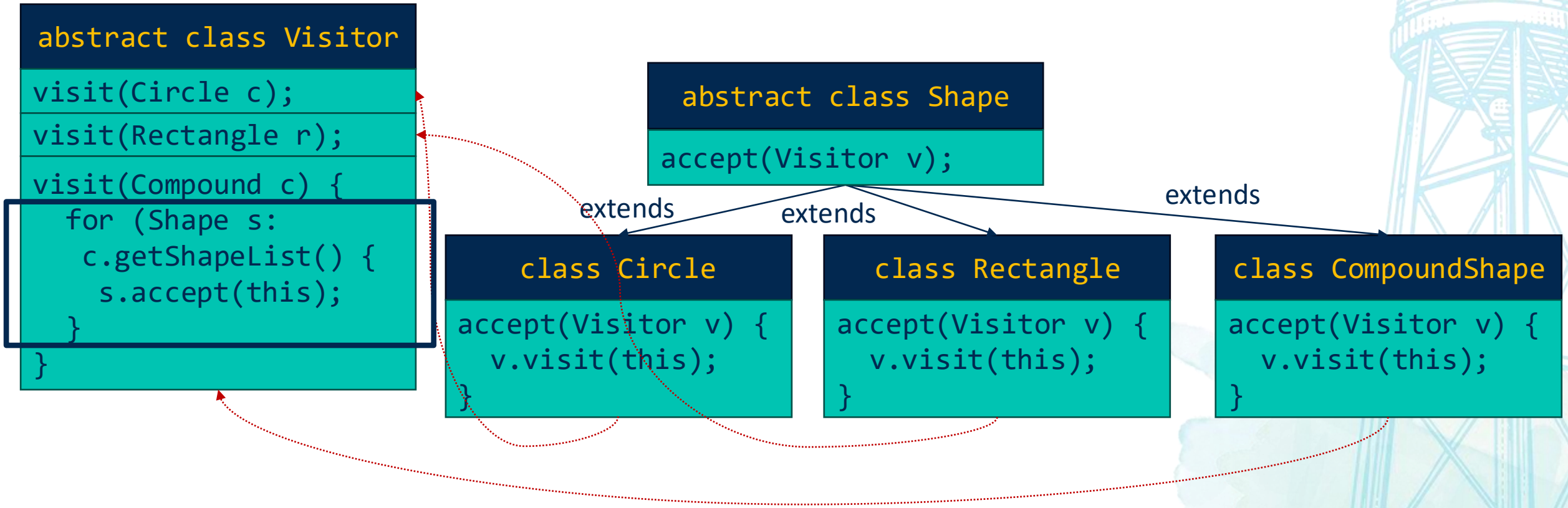


Visitor design pattern so far…



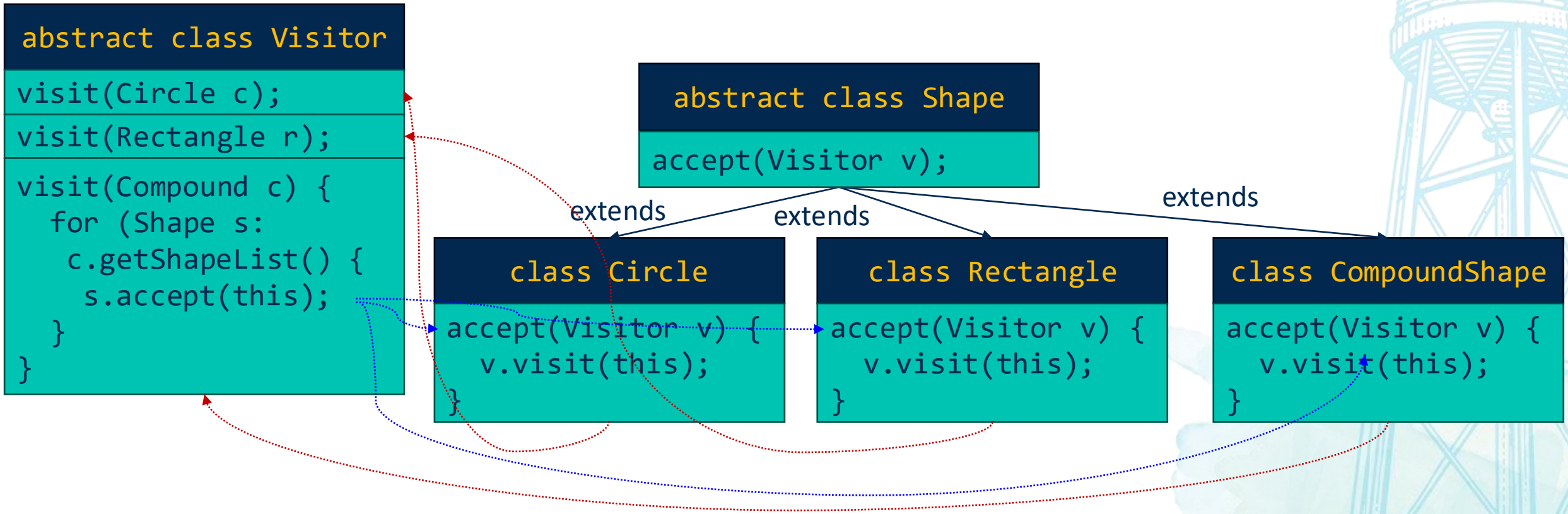
Method overloading ensures the correct `visit` ***method is invoked by the*** `accept` ***method of each*** Shape ***sub-class***

Visitor design pattern so far…



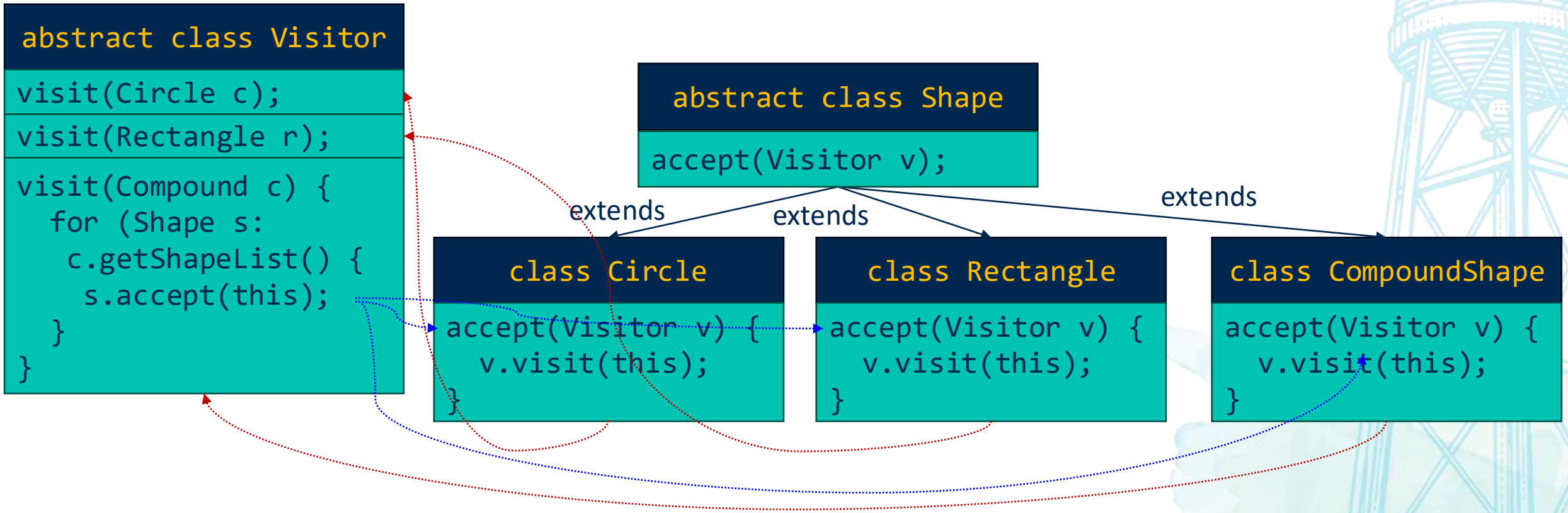
Method overloading ensures the correct `visit` ***method is invoked by the*** `accept` ***method of each*** Shape ***sub-class***

Visitor design pattern so far…



Runtime polymorphism ensures the correct accept method is invoked for each Compound shape

Visitor design pattern so far…



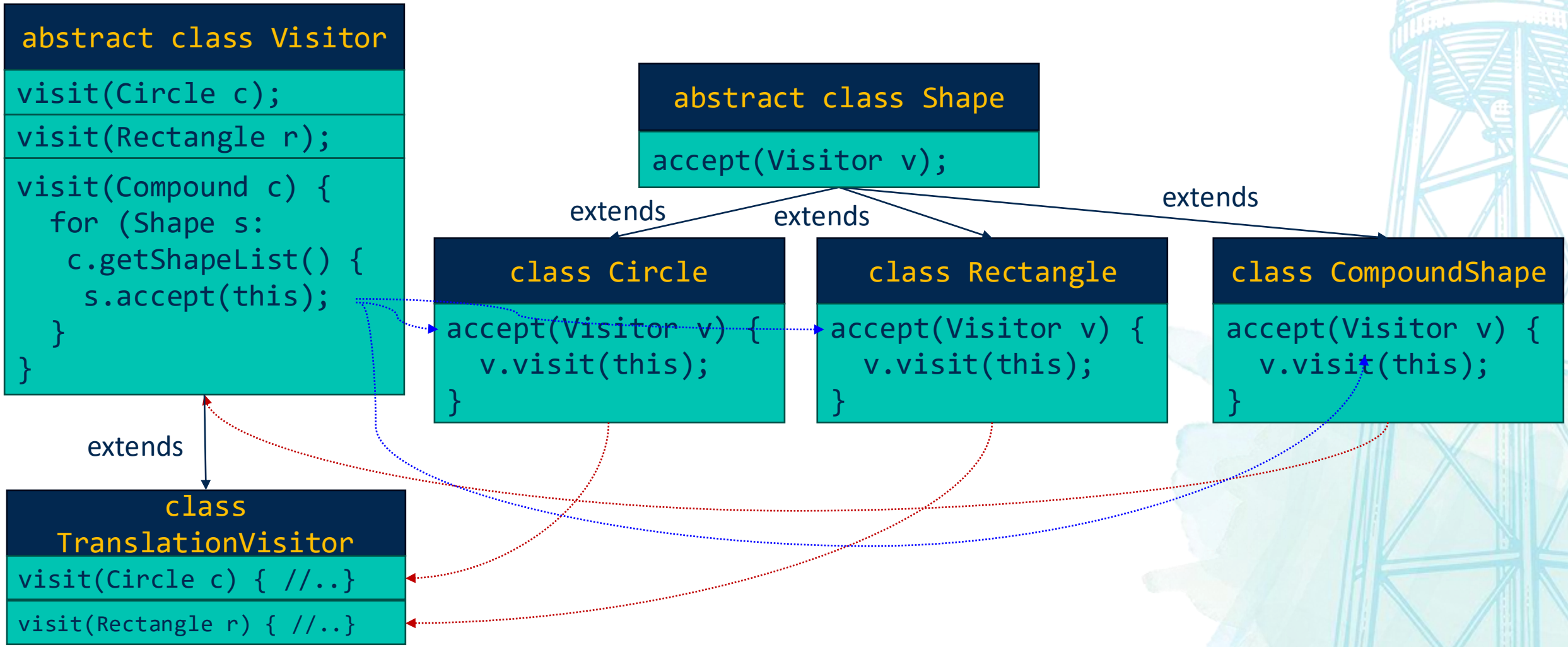
What would happen if there were nested CompoundShapes?

Visitor design pattern

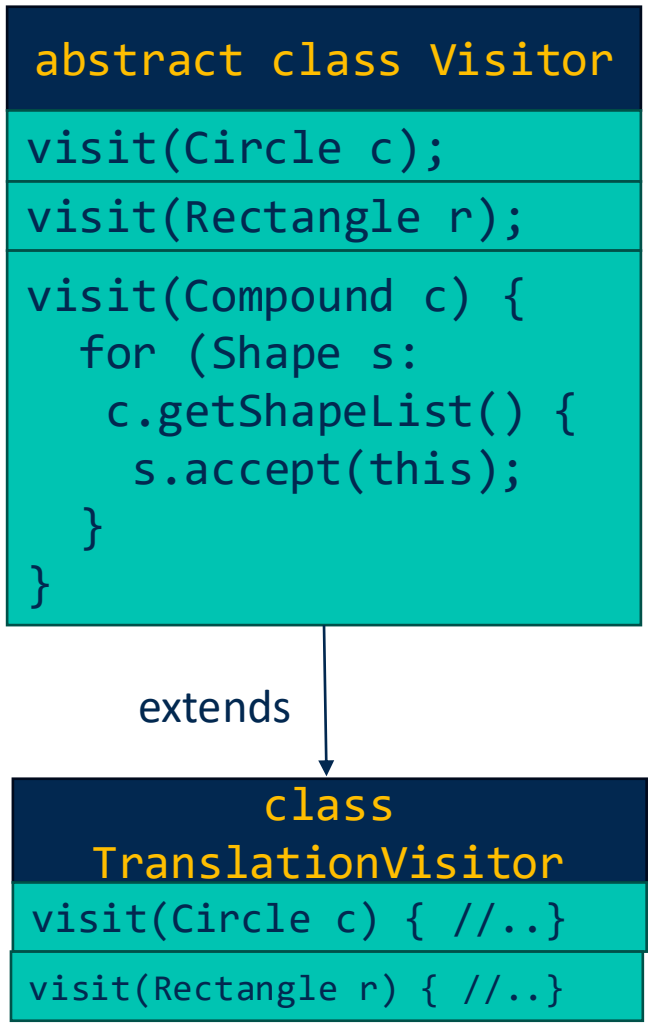
- Concrete visitor subclasses implement the particular visitor logic
- The operation logic is encapsulated in its own visitor class

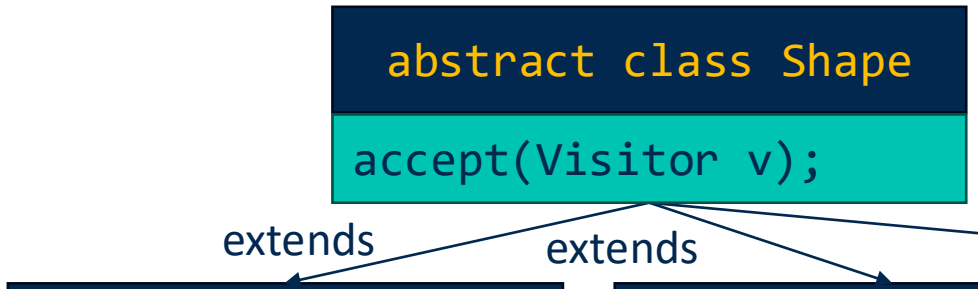
```
class TranslationVisitor extends Visitor {

    int x_off; int y_off;
    TranslationVisitor(int x, int y) {
        this.x_off = x; this.y_off = y;
    }
    @Override
    public void visit(Circle c) {
        c.setRadius(c.getRadius().getX() + x_off,
                c.getRadius().getY() + y_off);
    }

    @Override
    public void visit(Rectangle rectangle) {
        // translate 4 points
    }

    // No need to override visit for
CompoundShape
}
```

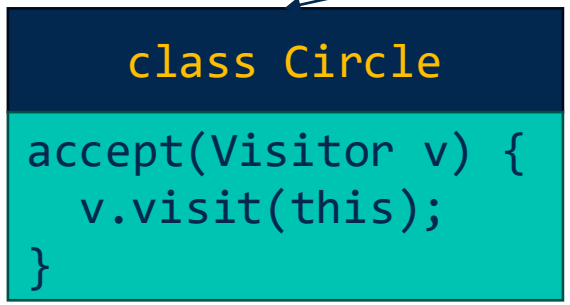
Visitor design pattern so far…

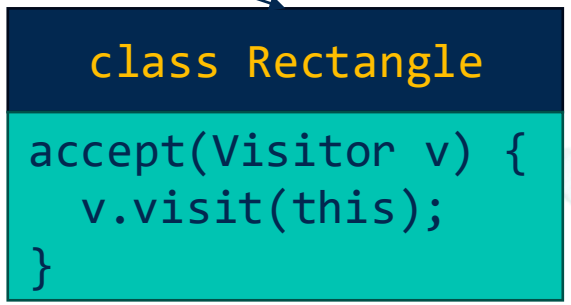


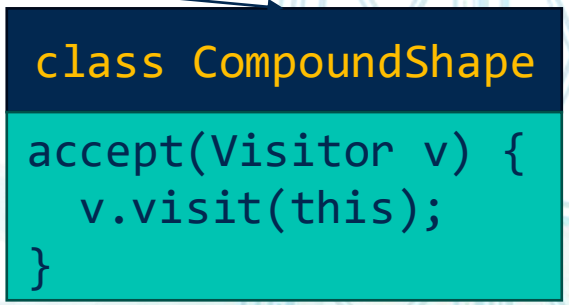
Example







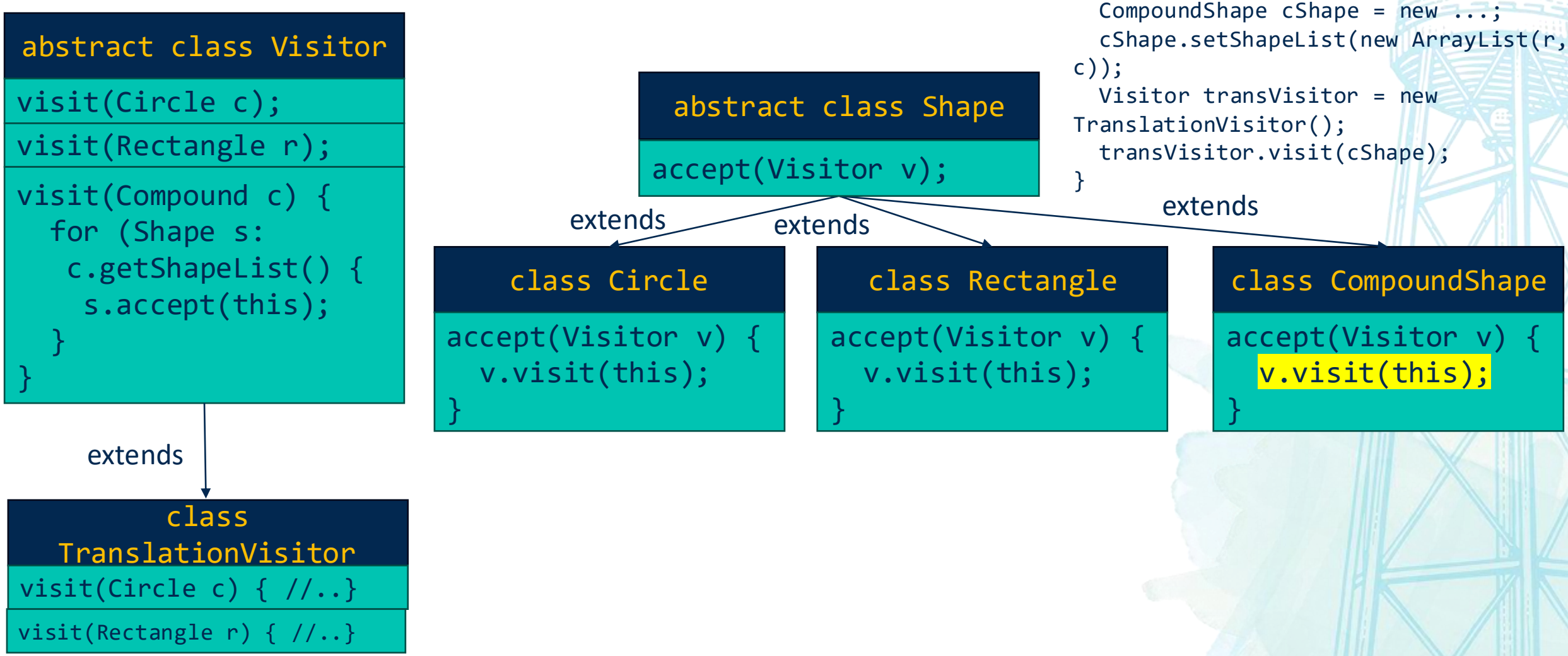




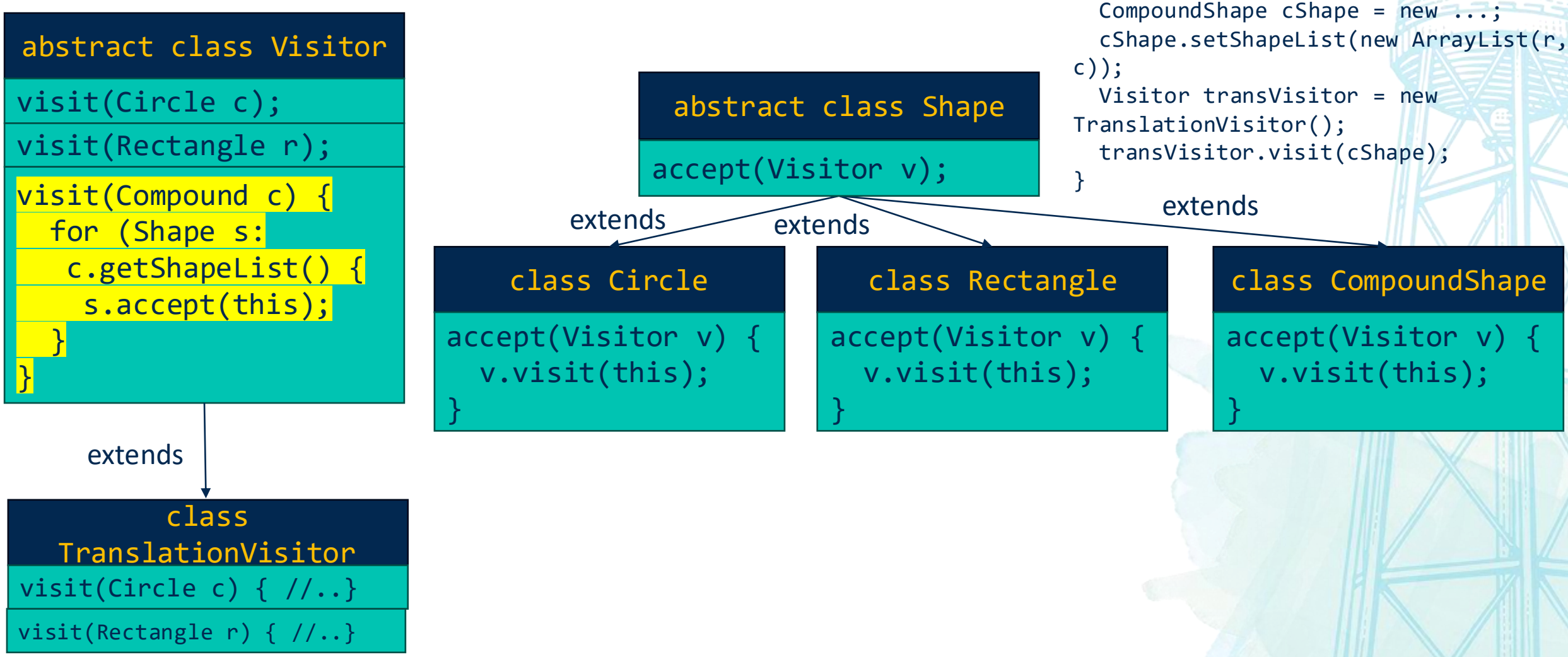

```
main() {
  Rectangle r = new …;
  Circle c = new …;
  CompoundShape cShape = new ...;
  cShape.setShapeList(new ArrayList(r,
c));
  Visitor transVisitor = new
TranslationVisitor();
  transVisitor.visit(cShape);
}
```

Example

```
main() {
  Rectangle r = new …;
  Circle c = new …;
  CompoundShape cShape = new ...;
  cShape.setShapeList(new ArrayList(r,
c));
  Visitor transVisitor = new
TranslationVisitor();
  transVisitor.visit(cShape);
}
```



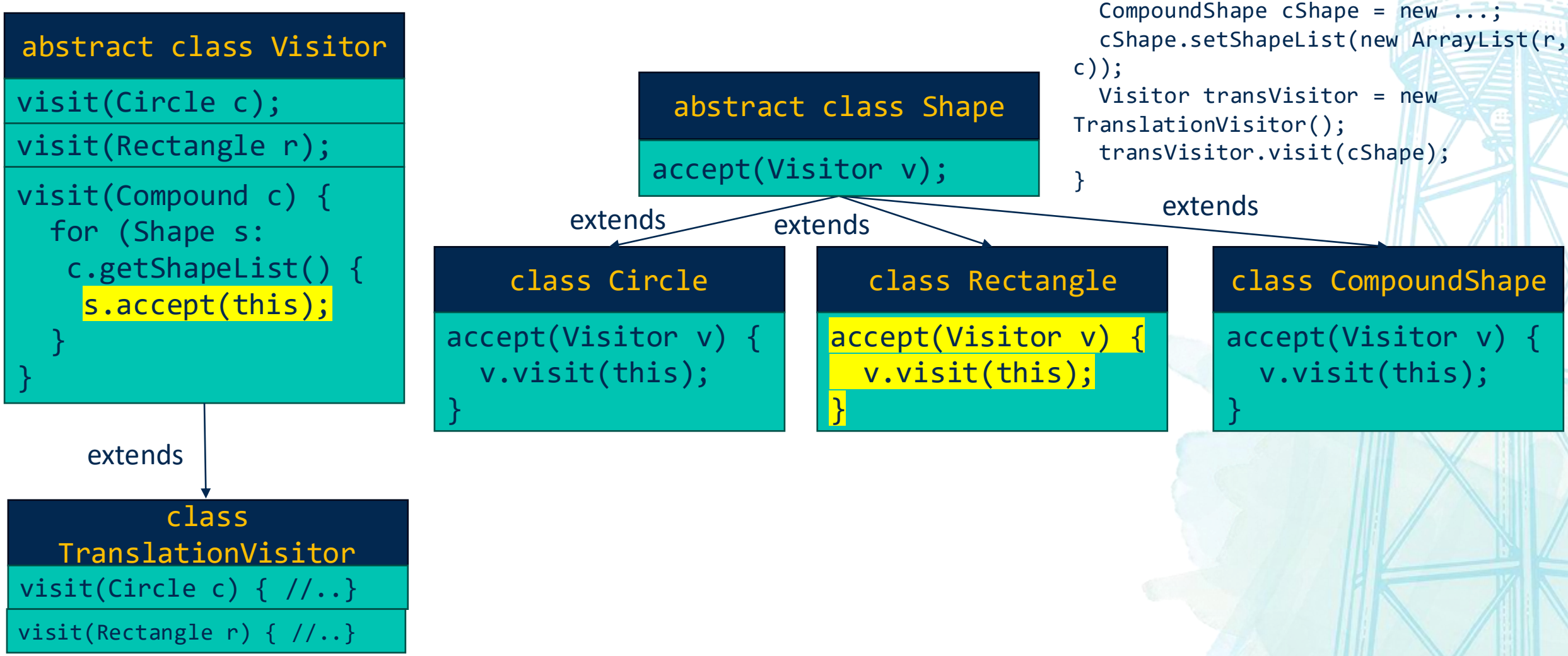
Example




```
main() {
  Rectangle r = new …;
  Circle c = new …;
  CompoundShape cShape = new ...;
  cShape.setShapeList(new ArrayList(r,
c));
  Visitor transVisitor = new
TranslationVisitor();
  transVisitor.visit(cShape);
}
```

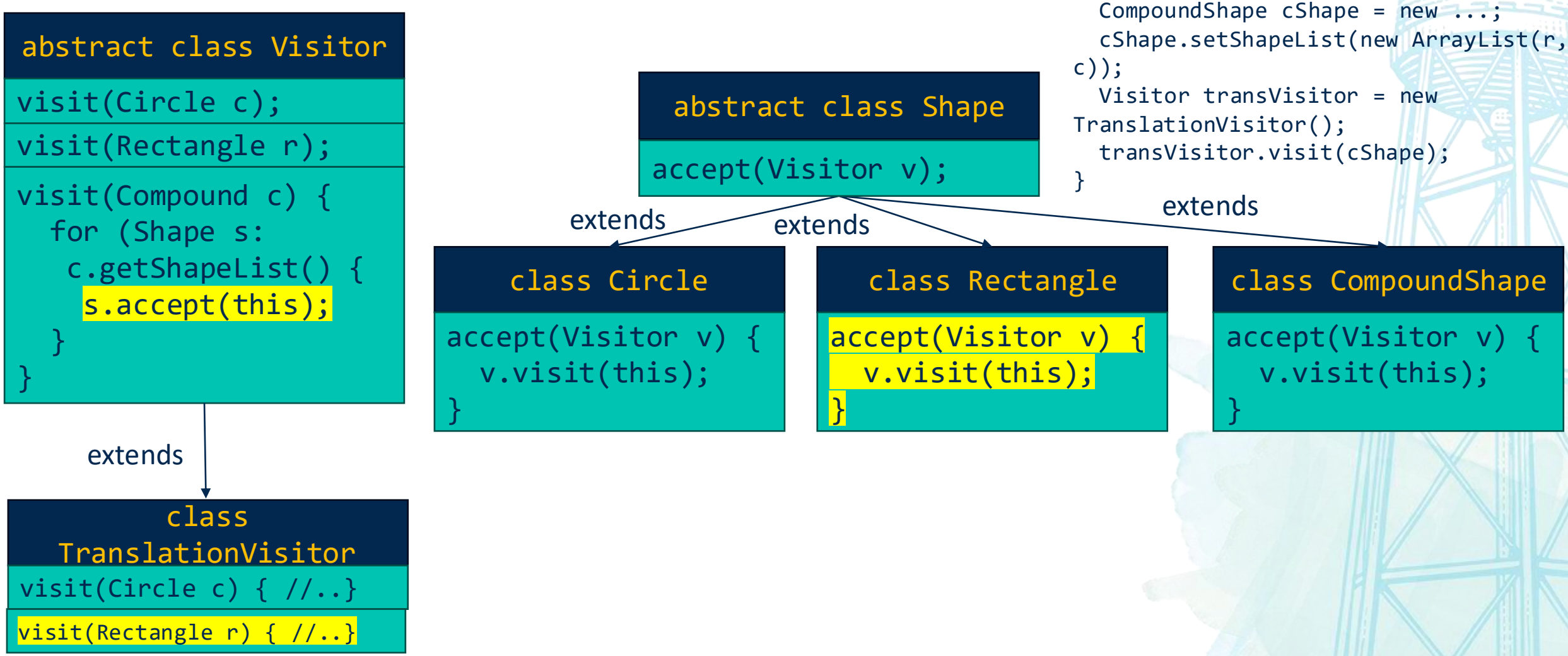
Example

```
main() {
  Rectangle r = new …;
  Circle c = new …;
  CompoundShape cShape = new ...;
  cShape.setShapeList(new ArrayList(r,
c));
  Visitor transVisitor = new
TranslationVisitor();
  transVisitor.visit(cShape);
}
```



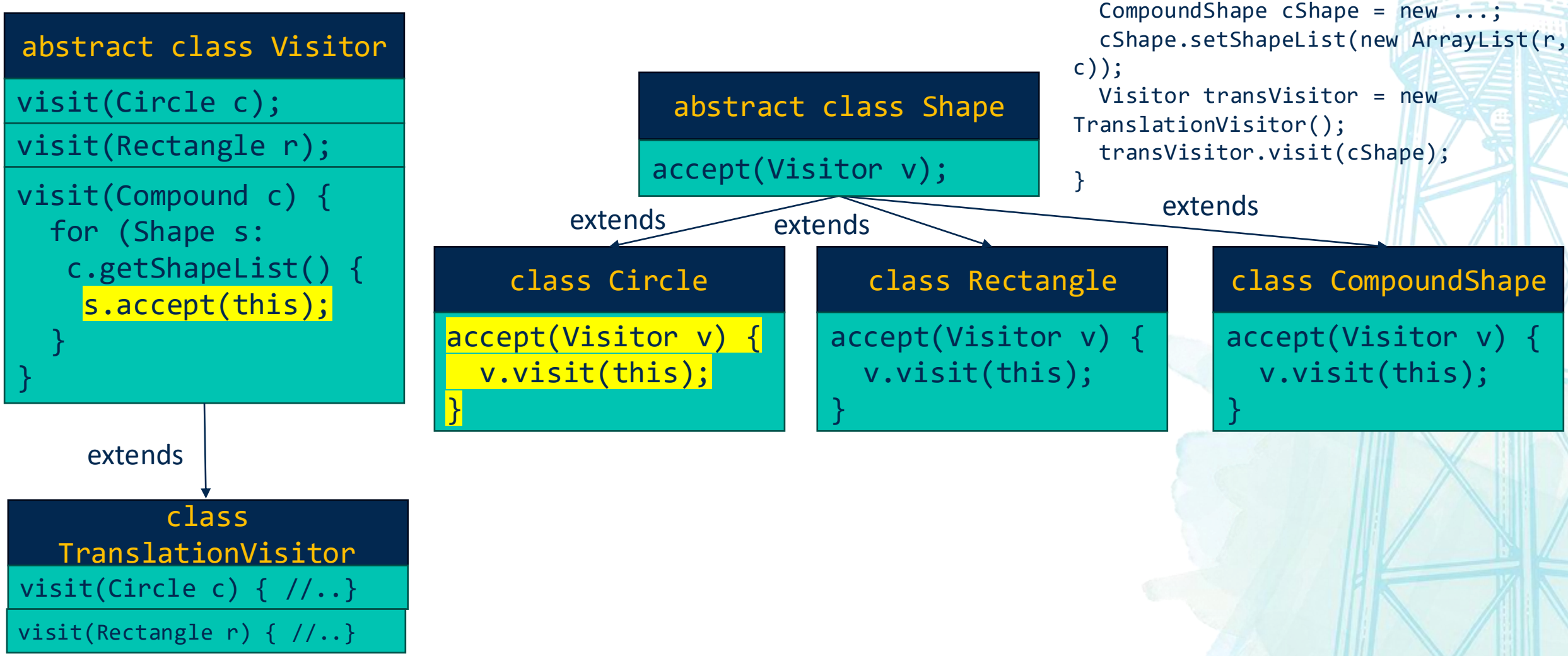
Example

```
main() {
  Rectangle r = new …;
  Circle c = new …;
  CompoundShape cShape = new ...;
  cShape.setShapeList(new ArrayList(r,
c));
  Visitor transVisitor = new
TranslationVisitor();
  transVisitor.visit(cShape);
}
```



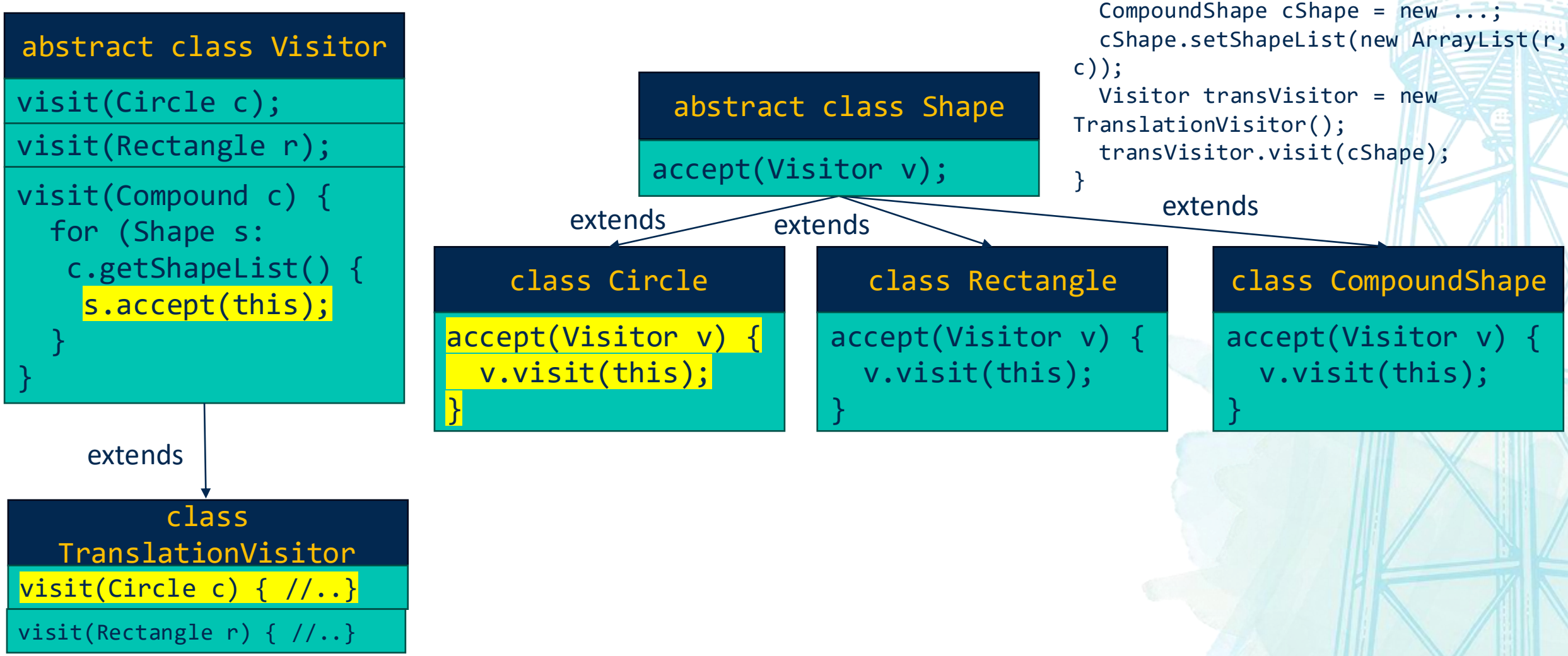
Example

```
main() {
  Rectangle r = new …;
  Circle c = new …;
  CompoundShape cShape = new ...;
  cShape.setShapeList(new ArrayList(r, c));
  Visitor transVisitor = new TranslationVisitor();
  transVisitor.visit(cShape);
}
```



Example

```
main() {
  Rectangle r = new …;
  Circle c = new …;
  CompoundShape cShape = new ...;
  cShape.setShapeList(new ArrayList(r,
c));
  Visitor transVisitor = new
TranslationVisitor();
  transVisitor.visit(cShape);
}
```

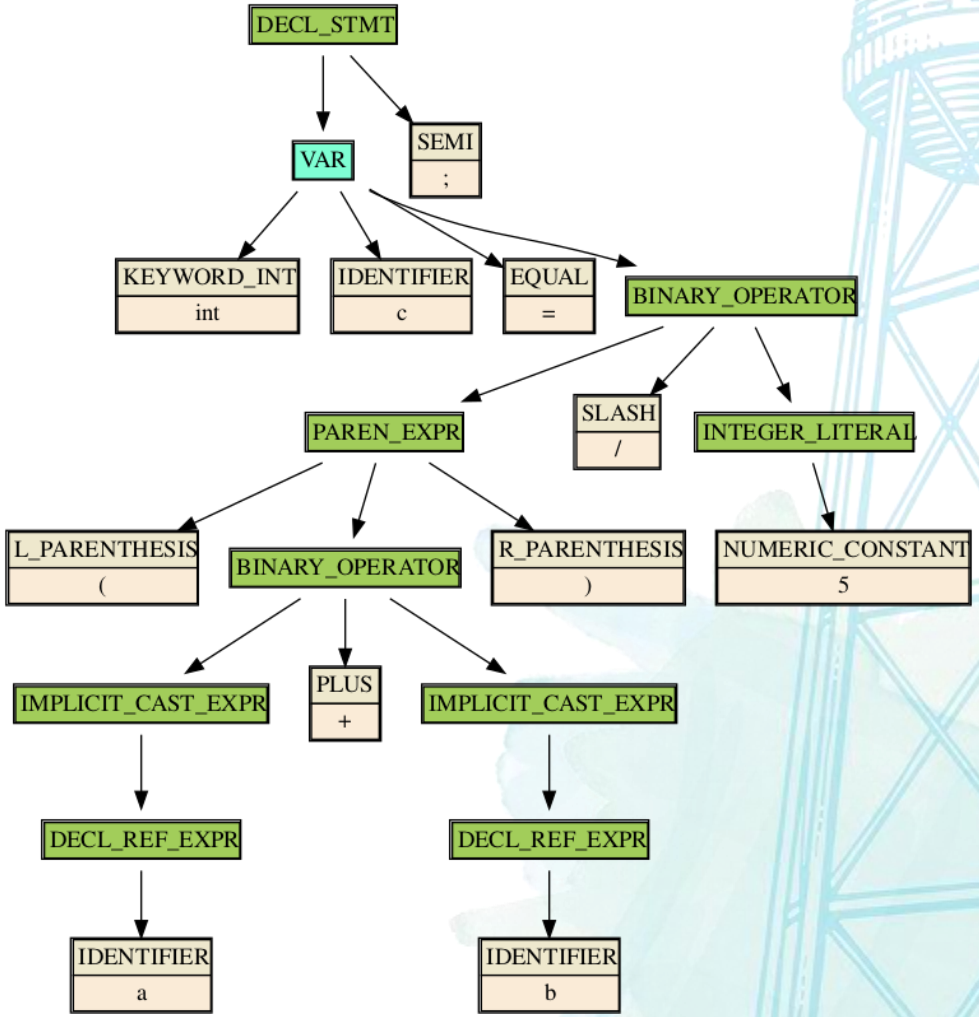


Visitor design pattern

- Separates operations (behavior) from the object structure (elements) they act upon
  - Can easily handle deeply nested compound objects uniformly
  - Abstract visitor can encapsulate visiting logic
  - `visit(Type obj)` only contains code relevant to visiting that particular object and not any sub-types
- Widely used in
  - Compilers/interpreters
  - Serialization/deserialization (JSON parsing)
  - Static analysis/code auditing

High-level case study – compiler code generation

- Abstract Syntax Tree (AST) used in compilers to represent source code
- You want to perform many different types of operations on each node
  - E.g., CountConstants, EvaluateExprs, GenerateCode



Summary

- Design patterns abstract object instantiation, composition, and behavior
- Three types – creational, structural, behavioral
- Creational – deals with object creation
  - Singleton, Factory/Abstract Factory
- Structural – deals with how objects are composed
  - Adapter, Proxy Decorator, Bridge, Facade
- Behavioral – how objects distribute responsibilities
  - Observer, state, template method, visitor
- Design patterns can be combined to solve complex tasks

Sample design pattern questions

Question 3

- Which design pattern should be applicable here? I want to implement a WebServer class. There should be exactly 3 webservers in a system at the most (let's call them larry, moe, and curly); there should never be more than 3, and it shouldn't be possible for programmers to accidentally create more than 3. These servers will be created only when needed. i.e., if no requests arrive, no webservers will be created; all 3 servers will exist only after the first 3 requests have arrived. Explain how you would design this system.

Sample design pattern question

You are developing software to support the automated feeding and watering of expensive flowering/fruiting Chocolate Truffle Persimmon plants at a computer-controlled greenhouse. These plants (depending on the season, and time of day) are in different biological conditions: ***dormant, growing, flowering, fruiting,*** **and** ***seeded***. Depending on the condition, (don't worry about how you know this, just assume you know) they will need to be watered differently, and fed different things. Do the wrong thing at the wrong time, they're dead. A separate timing mechanism (not your responsibility) periodically (say twice a day) issues requests to feed, and water, the plants. Your system should do the right thing, depending on the biological conditions of the plants. Which pattern would you use ?

You are designing a web server. The web server has a Connection Manager. All connections to this web server are handled by this ConnectionManager class.

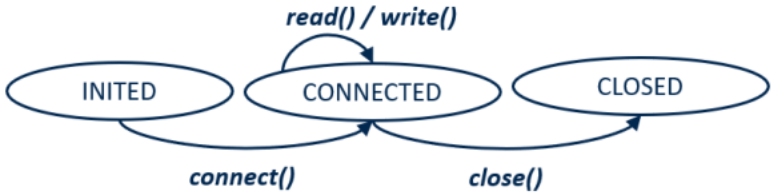
The ConnectionManager class creates and manages objects of type Connection and must support the following API -

1) Connection connect(); // returns a Connection object
2) String read(Connection c); // Tries to read the Connection object and return a String
3) void write(Connection c, String msg); // Tries to write msg to the Connection c
4) void close(Connection c); // Closes the Connection c

Similarly, the Connection class provides the following API -
1) void connect();
2) String read();
3) void write(String msg);
4) void close();

The connection can be in one of the following states -
1) INITED
2) CONNECTED
3) CLOSED

Not all operations are valid for all states. The valid operations and state transitions are provided in the following state transition graph. All other operations should print an error message on the terminal.



The ConnectionManager must also ensure that no more than 1000 active (non-closed) connections exist at any time.

Design the ConnectionManager and Connection classes, keeping in mind that you might have to modify the Connection class to support more states later. Note that you might have to tweak one or more design patterns to achieve what you want. **First, clearly specify which design pattern/s you are using in 1-2 sentences.** Then, provide the pseudo-code for the system.