

# Design patterns

Tapti Palit



# Announcements

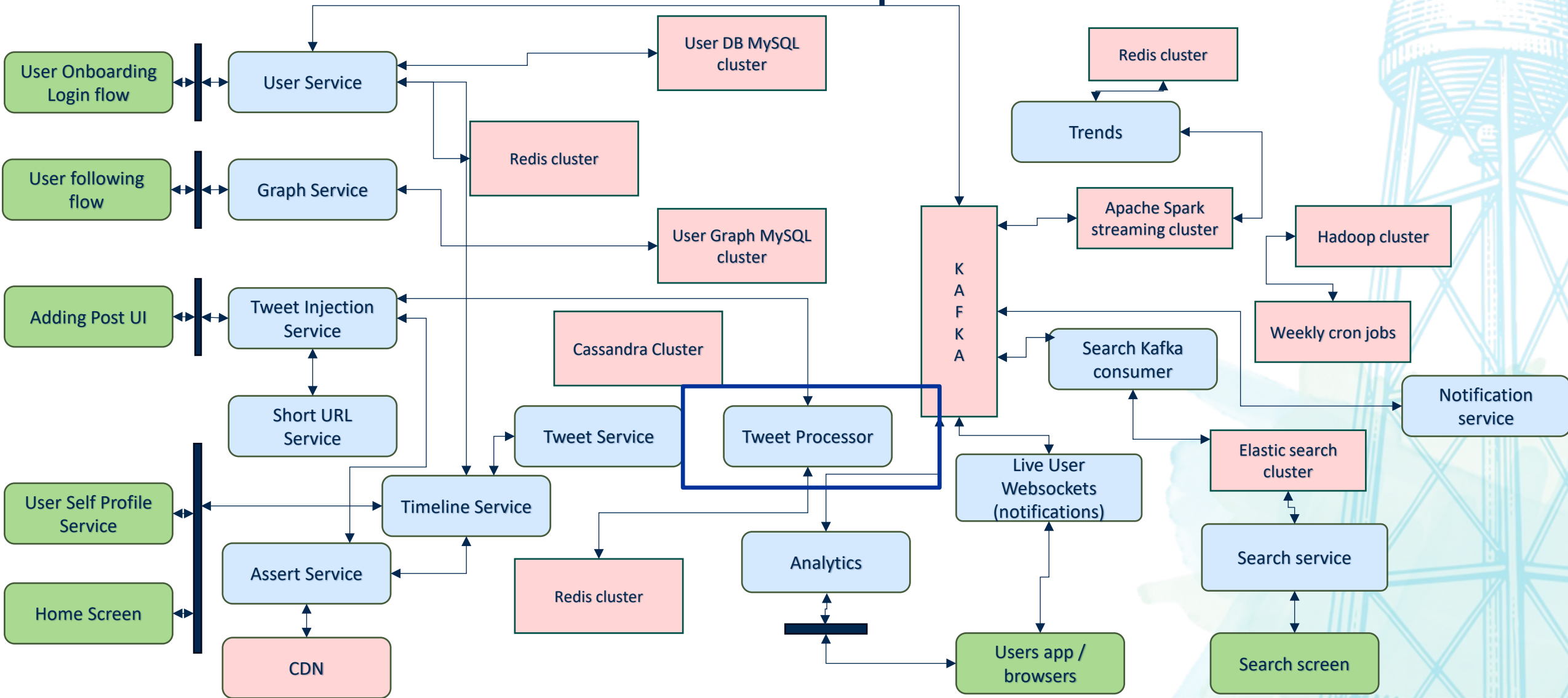
- Quiz 4 was more difficult than other quizzes
  - Feedback: explain each step
  - “If fp<sub>tr</sub> can be leaked...” => How can fp<sub>tr</sub> be leaked?
  - Explain each step using the code provided
- Quiz 5 will be a take-home quiz on Dec 5
- No office hours on 11/26; we will have class though
- HW3 will be released tonight – deadline Dec 12

Quiz	Avg score
Quiz 1	83%
Quiz 2	70%
Quiz 3	95%
Quiz 4	63%

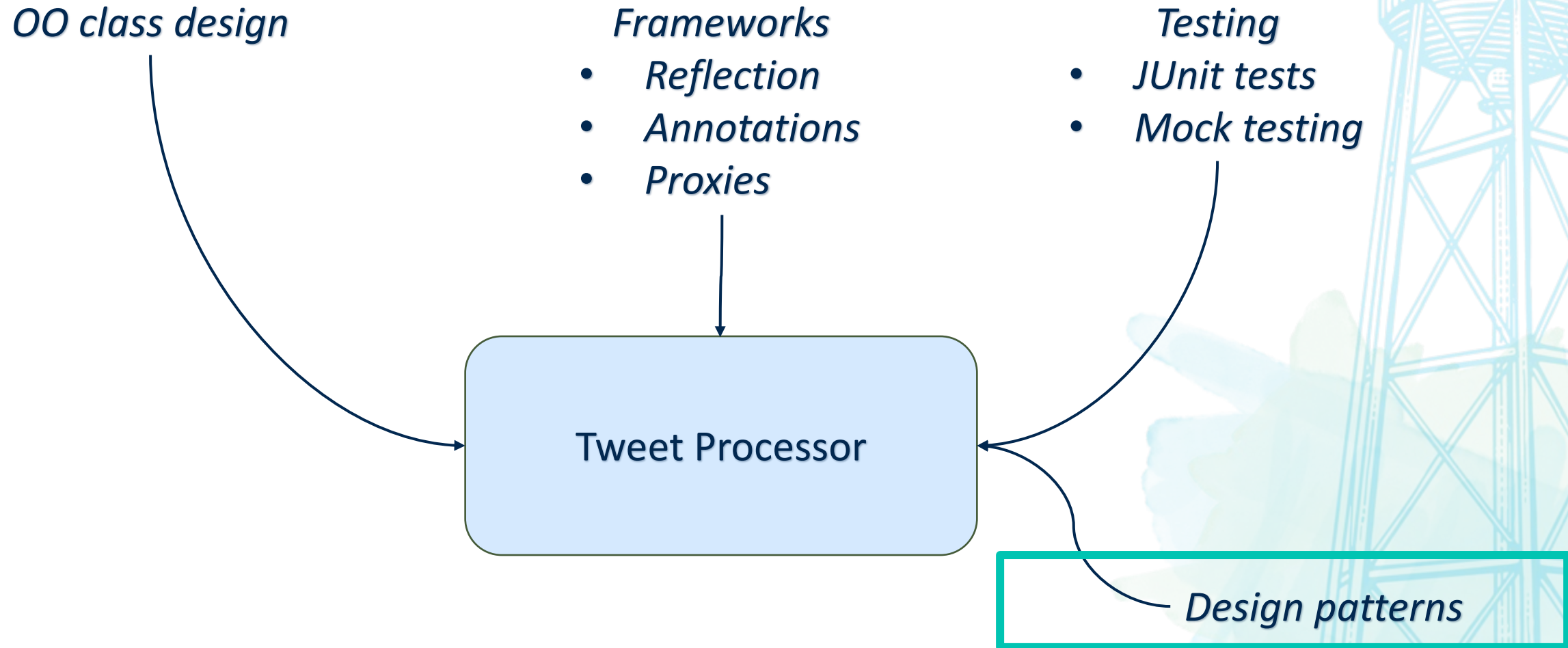
# What are design patterns?

- 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

# Social media platform



# Design patterns – single component focus



# Why do we need 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
  - At any time, only one object of the Logger class must exist
  - Any part of the application can access the Logger object
- How would you design it?

# Why do we need design patterns?

- How would you use this Logger class?
- Create a Logger object and pass it in each method declaration?
  - ... very ugly 😞
- Also, how would you ensure that a future programmer does not accidentally create a new Logger object?

```
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");  
}
```

# Question

- How to ensure a single object of the Logger class is accessible from many parts of the application?



# static field

- Create and store a Logger object as a static field in the Logger class
- The usage site can access the logger object by `Logger.logger`
- No need to pass the logger object as an argument to callee methods

```
class Logger {  
    private FileStream outputFile;  
    public static Logger logger = nullptr;  
  
    public Logger(String outputFileName) {  
        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();  
    Logger.logger = logger;  
  
    anotherFunction();  
}  
public void anotherFunction() {  
    Logger.logger.log("This is a log message");  
}
```

# static field

- 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() {  
        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();  
    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

- Write 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() {
        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();
    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

- Provide a static method `getLogger()` that instantiates the logger
- Turn the constructor private
  - All object creation goes through `getLogger()`
- `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

# Why should we study design patterns?

- Important to quickly recognize common requirements
- Important for reading large codebases
  - Class B extends class A and also contains an object of class A
  - What's **one** possible reason?
  - Could be many reasons, but one reason could be they are implementing the *proxy pattern*

```
class A {  
    // Some fields  
    // Some methods  
}  
  
class B extends A {  
    private A objA;  
  
    // Some fields  
    // Some methods  
}
```

# 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, Composite, 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
  - Very ugly 😞

```
Button button = nullptr;  
if (platform == "win") {  
    button = new WinButton(); // WinButton  
    extends Button  
} else {  
    button = new MacOSButton(); // Extends  
    Button  
}
```

```
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

- Isolates concrete classes
  - Objects are created through the interface / abstract classes
- 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 here?***

# 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, Composite, Decorator, Bridge, Façade

## Behavioral patterns

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

# Structural patterns

- How classes and objects compose to form larger structures
- Structural patterns can be applied at
  - Class level
  - Object level



# Adapter design pattern



# Adapter design pattern

*App*

*libStripe*

**StripeProcessor**

```
void makePayment(int amt);
```

```
interface PaymentProcessor
```

```
void pay(int amt);
```

```
class PaypalProcessor  
implements PaymentProcessor
```

```
void pay(int amt) {// code}
```

```
class MyApp
```

```
main() {  
    PaymentProcessor p = new  
    PaypalProcessor();  
    p.pay(1000);  
}
```

*Use Stripe  
instead of Paypal*

# Adapter design pattern

*App*

*libStripe*

**StripeProcessor**

```
void makePayment(int amt);
```

```
interface PaymentProcessor
```

```
void pay(int amt);
```

```
class PaypalProcessor  
implements PaymentProcessor
```

```
void pay(int amt) {// code}
```

***StripePaymentProcessor  
does not implement PaymentProcessor***

*instead of Paypal*

```
...sor p = new  
StripePaymentProcessor();  
// ... How?  
}
```

# 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
- Also known as *wrapper*

# 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);
}
```

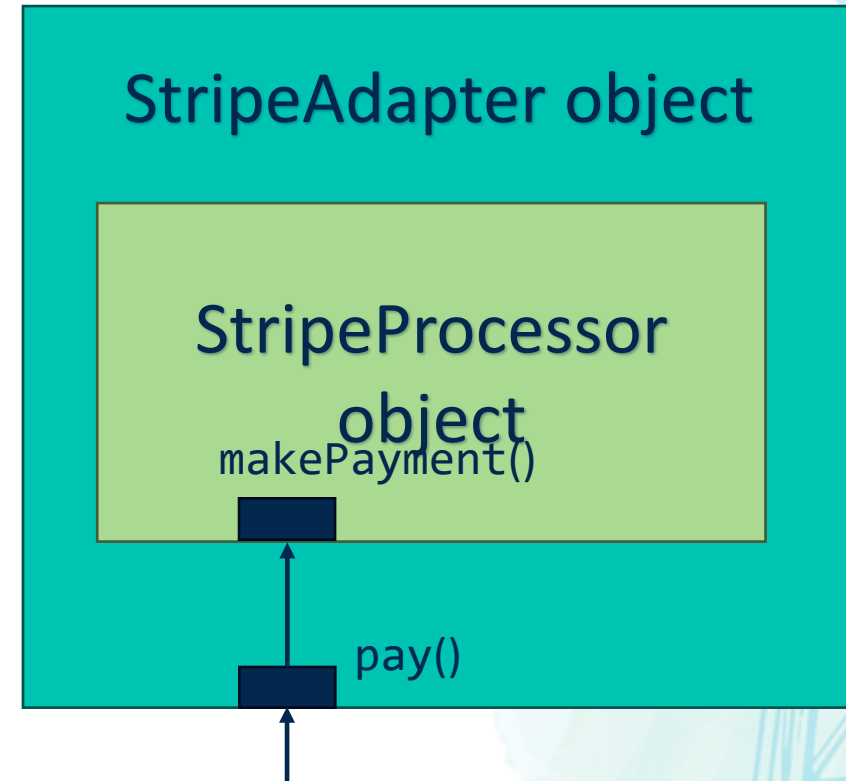
```
class PaypalProcessor implements PaymentProcessor {
    void pay (int amount) {
        // paypal functionality
    }
}
```

```
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);
    }
}
```

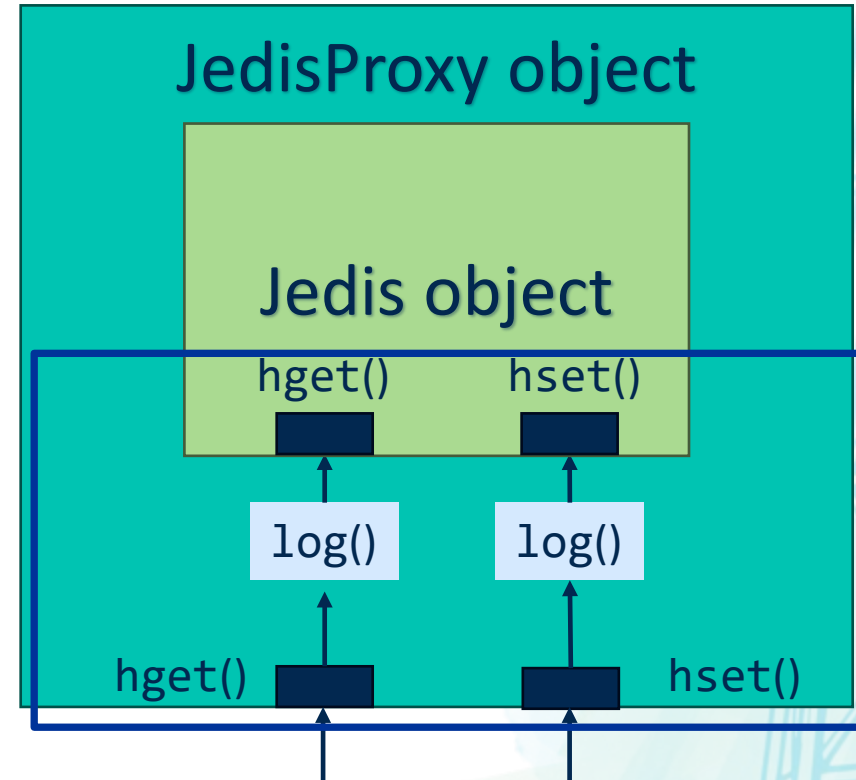
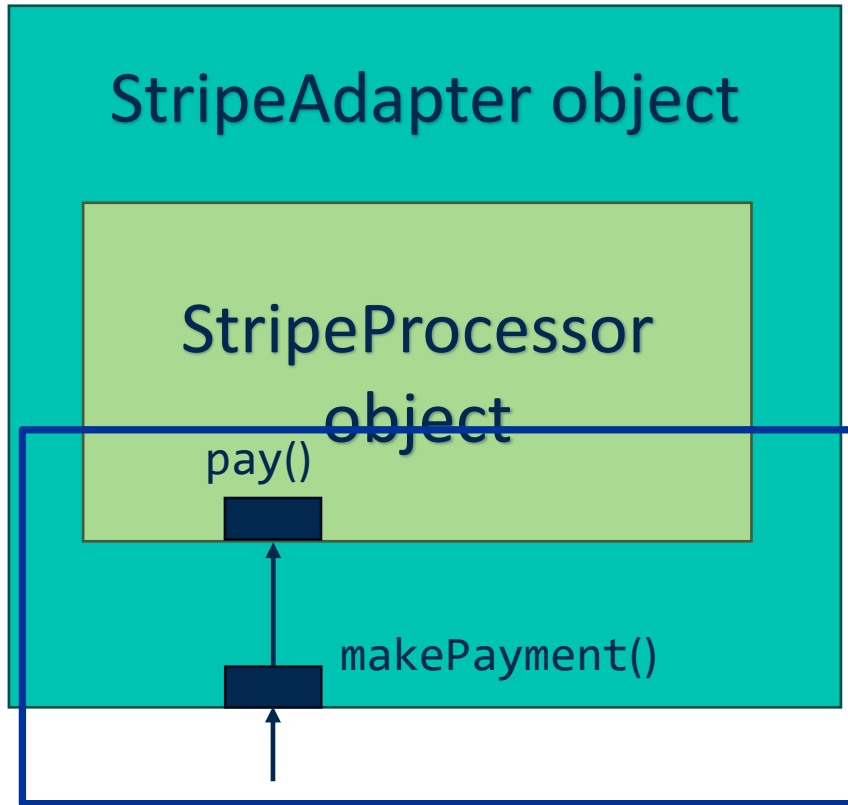
# Payment interface adapter

- 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



# Adapter vs. proxy design pattern

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



# Characteristics

- A single adapter can work with many adaptees – that is, the Adaptee itself and all of its subclasses (any subclass of `StripeProcessor` in the previous example)
- Makes it harder to override Adaptee behavior
  - Must subclass the Adaptee and make Adapter refer to the subclass object

# Design patterns



# Announcements

- Late day count pauses tonight at 11:59 PM and resumes on 12/1 at 12:00 AM



# Agenda

- Sample questions
- Composite pattern
- Bridge pattern
- Façade pattern
- Decorator pattern



# Sample question 1

- Design patterns: singleton pattern, abstract factory pattern, adapter pattern, proxy pattern
- 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 question 2

- I have a `GitHubRepo` class with fields for – `author`, `url`, `creationDate`, and `lastTenCommitList`. The `lastTenCommitList` contains a list of the last ten commits – the code diffs, the file names – everything.
- The details of the `GitHubRepo` class is fetched using API calls
  - `https://api.github.com/repo?name=<repo_name>`
  - `https://api.github.com/repo_commits?name=<repo_name>&limit=<num_commits>`
- I want the `lastTenCommitList` field, which is a `List` object, to be lazy-loaded. Only when the user performs a call to `getLastTenCommitlist()` should the commits be fetched. How should I design this?

## Sample question 2

- I want to design a framework that supports lazy loading from an API. The framework should support a class-level annotation `@Remote` and a method-level annotation `@RemoteField(url = “/api/endpoint”)`. When the getter of a field annotated with `@RemoteField` is invoked, the url endpoint should be invoked, which would return a JSON array object. This JSON array object should be deserialized into a Java `List` and stored to the field annotated with the `@RemoteField` annotation.
- How should I design this framework? Assume that fields annotated with `@RemoteField` are only accessed by their getters and that a getter named `get<Field>` is always present for that field. Also, assume that only fields of type `List` are annotated with the `RemoteField` annotation

# 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, **Composite**, Bridge, Façade, Decorator

## Behavioral patterns

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

# Composite design pattern



# Case study: social media posts

- A Post is a leaf social media entry
- A Thread is a collection of posts and/or threads
- Both Post and Thread should have similar operations such as show, hide, report, etc.
- ... *bring them under same type hierarchy*

**Thread**

jjconnor77 • 5h ago •  
Bought from whom?  
351 upvotes, 0 replies, 0 awards, 0 shares

garyzxcv • 5h ago •  
Located within Grand Teton National Park, the Kelly Parcel stretches 640 acres through the Greater Yellowstone Ecosystem. It has been owned by the state of Wyoming since the state's establishment, but it's only been a part of the national park since 1950.  
495 upvotes, 0 replies, 0 awards, 0 shares

8trackthrowback • 5h ago •  
So the park was not protected, despite being a National Park? Because somehow Wyoming owns a National Park which makes no sense too  
But the Federal Government now buys it and it is protected now? This is so strange  
I thought FDR created the National Park System which protects land forever once it's classified as such  
343 upvotes, 0 replies, 0 awards, 0 shares

way2lazy2care • 5h ago •  
The park is 310,000 acres, and this parcel is 600 of it. So this parcel was more or less on loan to the government. Most of the park has been protected for a long time.  
249 upvotes, 0 replies, 0 awards, 0 shares

LostWoodsInTheField • 3h ago •  
And if people look at national park lands often there is private tracts inside of them, or state plots. This isn't unusual at all. Sometimes they will even trade land with private owners to get better spots for national resource preservation, or the owners wanting land for development (where their current land isn't useful for that) and will trade larger acres for smaller acres.  
I think Utah went through something like that this last year. Not sure what the outcome was.  
83 upvotes, 0 replies, 0 awards, 0 shares

**Post**

# Composite design pattern for social media post

- Create abstract class `SocialMediaContent` that supports the display, hide, report functionalities
- Post and Thread are subclasses
  - Post is the “leaf”
  - Thread is the “composite” composed of a List of `SocialMediaContent`
- What is the benefit?

```
public abstract class SocialMediaComponent {  
    public abstract void show();  
    public abstract void hide();  
    public abstract void report();  
}
```

```
public class Post extends SocialMediaComponent {  
    private String content;  
    public void show() {  
        // display logic  
    }  
    // other methods  
}
```

```
public class Thread extends SocialMediaComponent {  
    private List<SocialMediaComponent> components =  
    ...;  
    public void show() {  
        for (SocialMediaComponent c: components) {  
            c.show();  
        }  
    }  
}
```

# Benefits

- Reduces code duplication in Thread class
- No need to have distinct code for posts and threads

```
// Without composite design pattern
public class Post {
    private String content;
    public void show() {
        // display logic
    }
    // other methods
}
```

***Code duplication***

```
public class Thread {
    private List<Post> leafPosts = ...;
    private List<Thread> subThreads = ...;
    public void show() {
        for (Post p: leafPosts) {
            p.show();
        }
        for (Thread t: subThreads) {
            t.show();
        }
    }
}
```

# Benefits

- Reduces code duplication in Thread class
- No need to have distinct code for posts and threads

```
// With composite design pattern
public abstract class SocialMediaComponent {
    public void show();
    // other methods
}

public class Post {
    private String content;
    public void show() {
        // display logic
    }
    // other methods
}

public class Thread {
    private List<SocialMediaContent> children;
    public void show() {
        for (SocialMediaContent child: children) {
            child.show();
        }
    }
}
```

# Benefits

- Prevents code duplication in **users** of Thread and Post class
- E.g., UI components for post and threads can have unified handling

## *Code duplication*

```
// Without composite design pattern
class PostShowButton {
    Post post;
    void getString() {
        post.getString();
    }
}

class ThreadShowButton {
    Thread thread;
    void getString() {
        thread.getString();
    }
}
```

## *No code duplication*

```
// With composite design pattern
class ShowButton {
    SocialMediaComponent content;
    void getString () { content.getString(); }
}
```

# Benefits

- Easy to add another `SocialMediaComponent`

```
public abstract class SocialMediaComponent {  
    public abstract void show();  
    public abstract void hide();  
    public abstract void report();  
}  
  
public class Post extends SocialMediaComponent {  
}  
  
public class Thread extends SocialMediaComponent {}  
  
public class Spaces extends SocialMediaComponent {  
    public void show() { // show livestream }  
    public void hide() { // hide livestream }  
    public void report() { // report livestream }  
}  
  
class ShowButton {  
    SocialMediaComponent content;  
    void onClick () { content.show(); }  
}
```

# Composite design pattern

- Ideal for representing hierarchical structures where an object can contain other objects from the same class hierarchy
- Composite pattern allows clients to treat individual objects and compositions of objects uniformly

# Characteristics

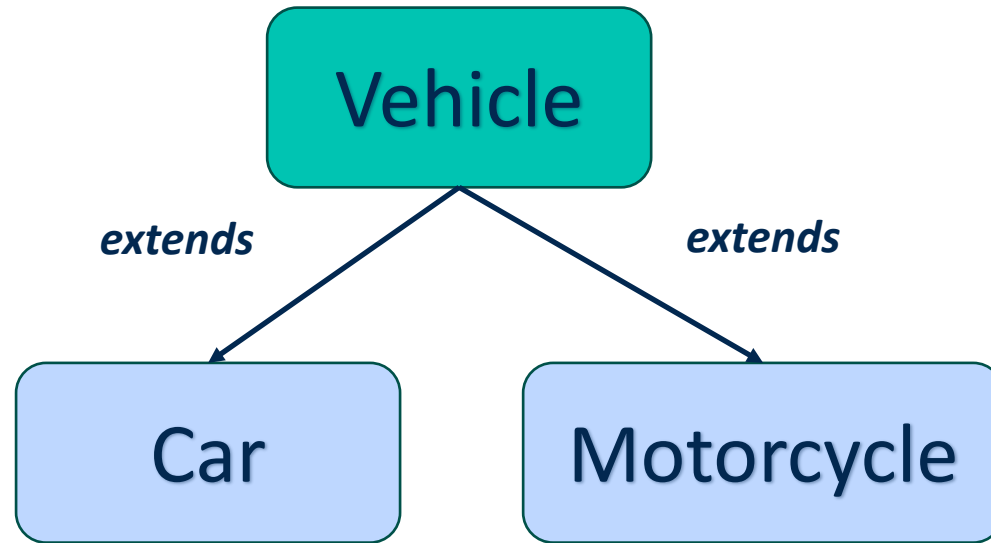
- Simplifies the client
- Clients can treat leaf and composite objects uniformly
- Makes it easier to add new kinds of components



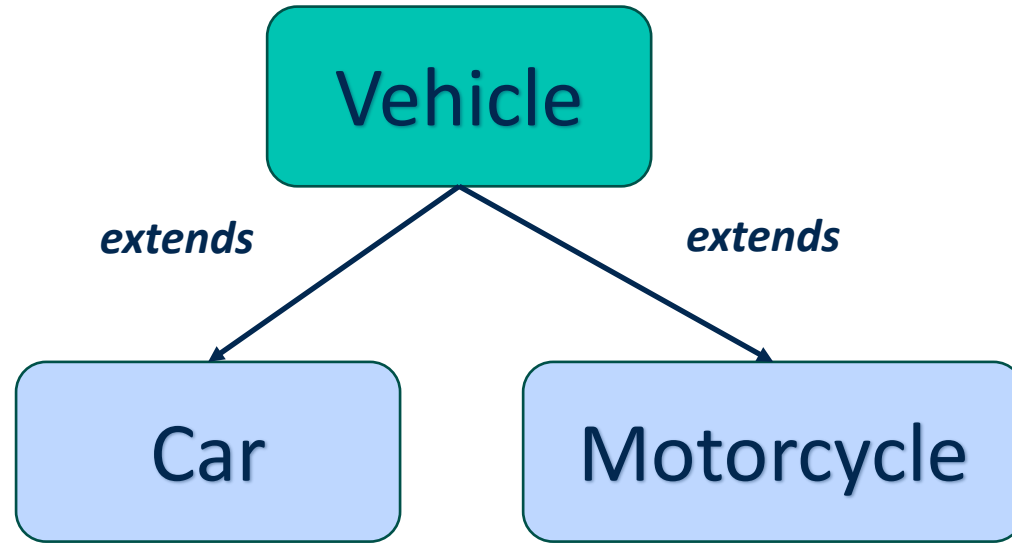
# Bridge design pattern



# Case study: vehicle class hierarchy

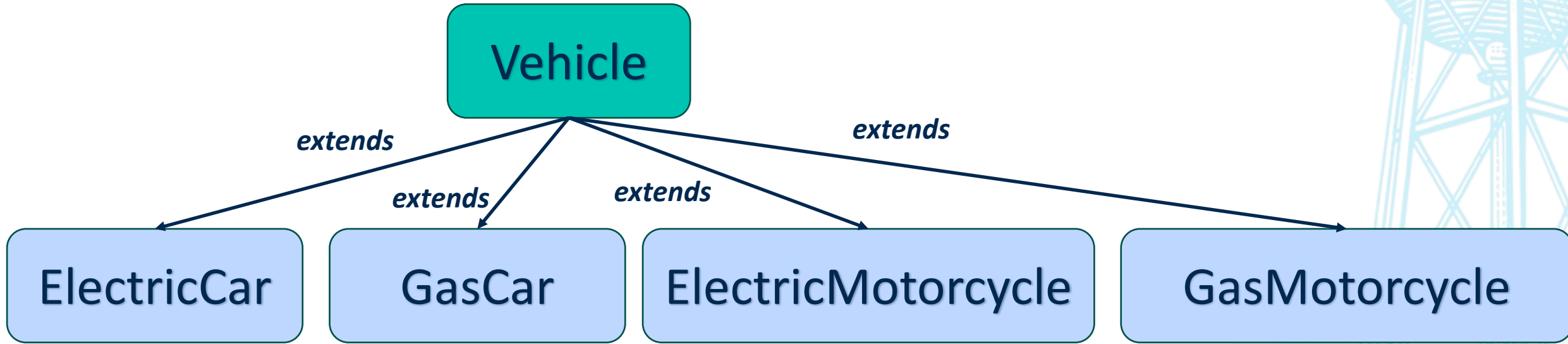


# Case study: vehicle class hierarchy



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

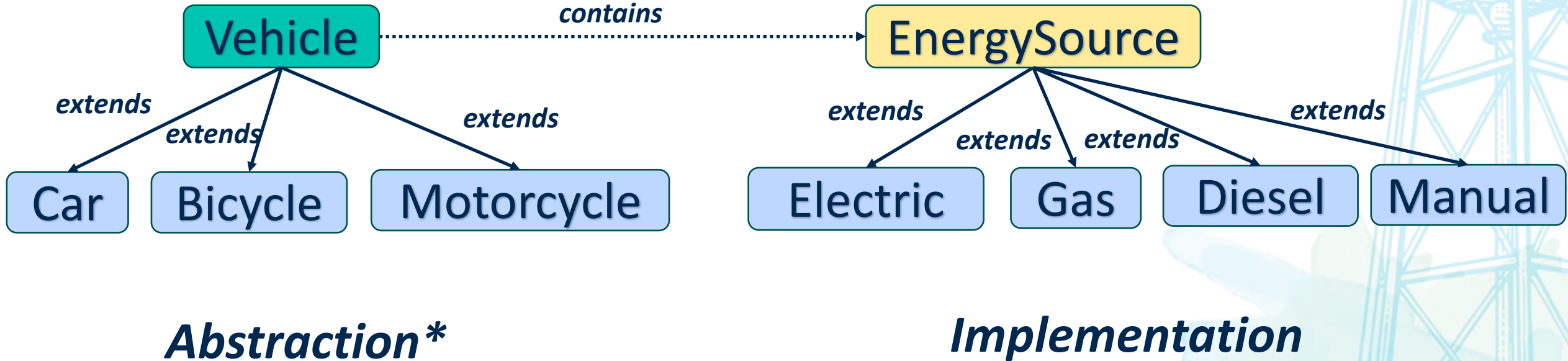
# Case study: vehicle class hierarchy



*What if we want to add diesel vehicles?*

*What about other vehicles like Bicycle which are manually operated?*

# Case study: vehicle class hierarchy



\*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
- SRP maintained

```
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



# 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, Composite, Bridge, Façade, Decorator

## Behavioral patterns

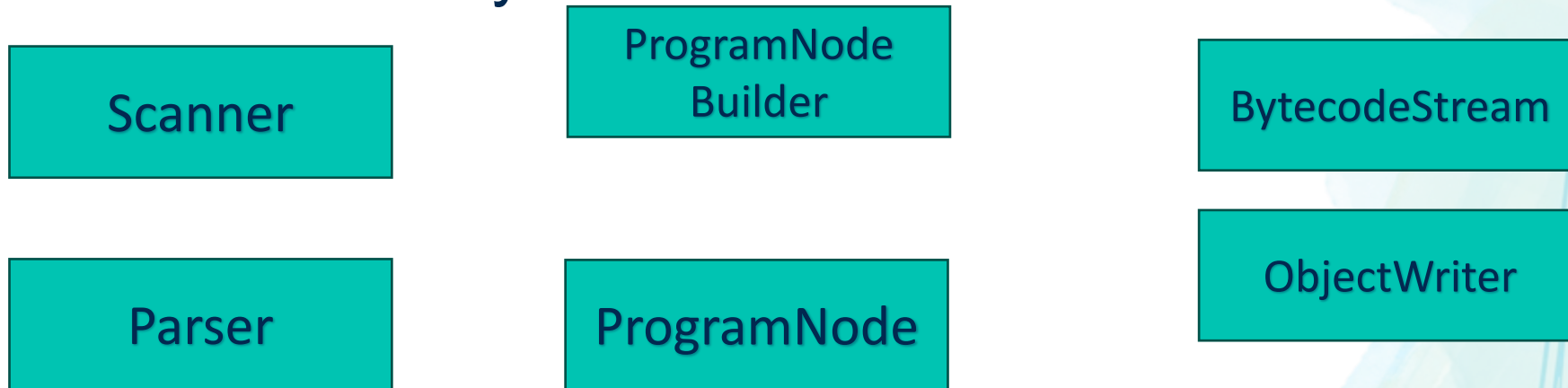
- Control how objects distribute responsibilities
- Template method, State, Observer, Visitor

# Facade design pattern



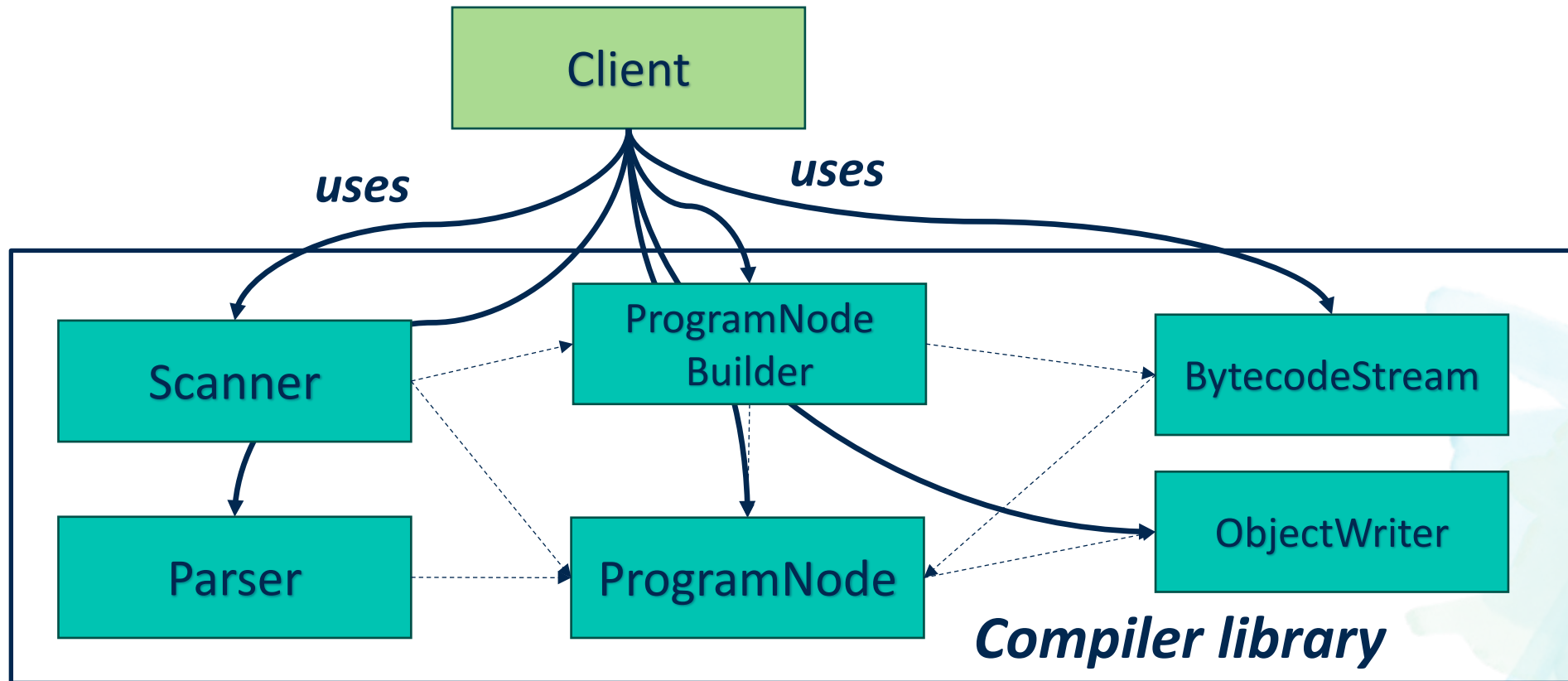
# Case study – compiler library

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



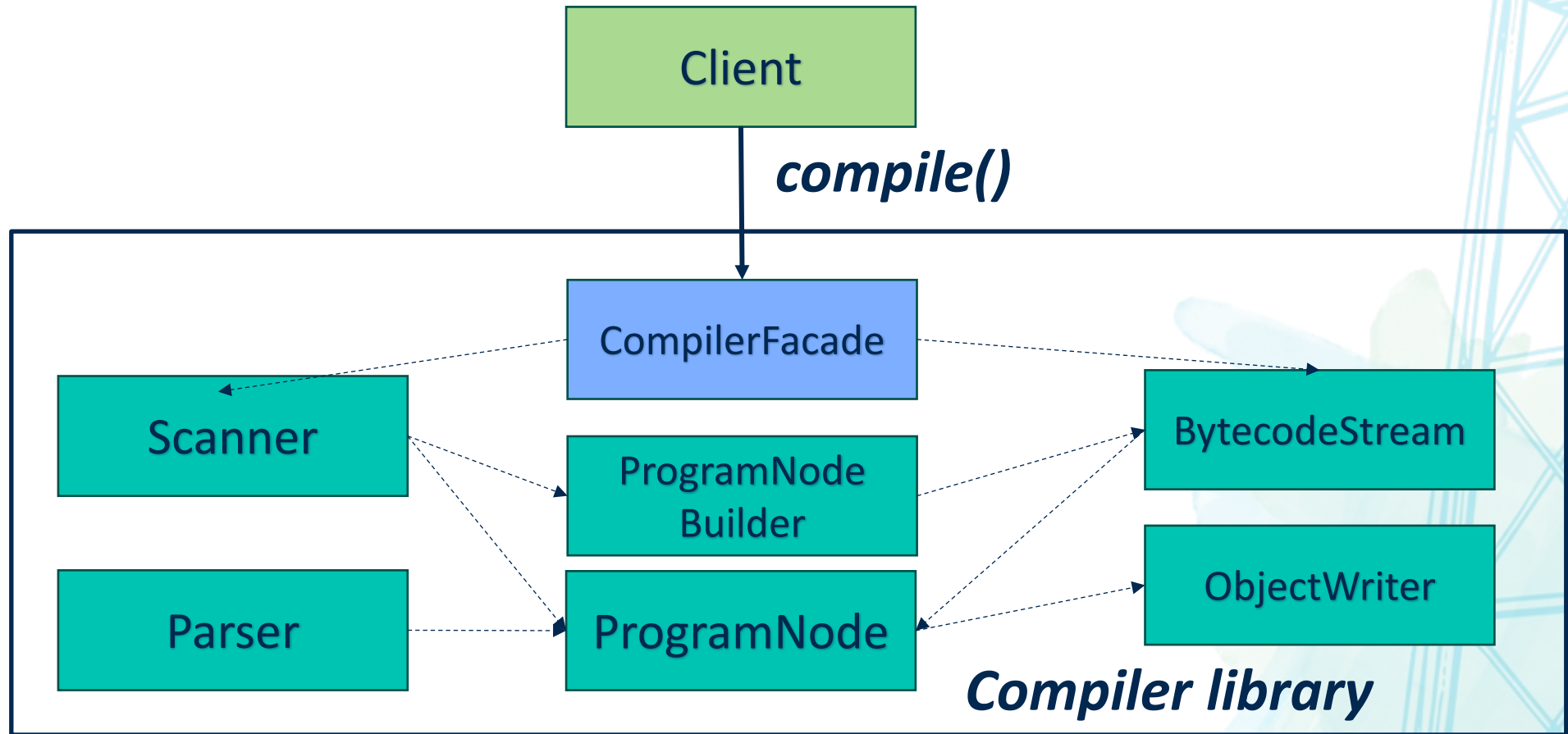
# Case study – compiler library

- The client can access each of these subcomponents individually



# Case study – compiler library

- A compiler 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

# Design patterns



# Announcements

- HW2 grades should be out by next Sunday
- Extra Zoom office hours 12/4 from 11 AM – 12:30 PM
- Next class: sample software security and design pattern questions
- Final lecture: revision

# Agenda

- Structural design patterns
  - Decorator pattern
  - Flyweight pattern
- Behavioral design pattern
  - Observer pattern
  - State pattern
  - Template method pattern
- Sample design pattern questions
- Behavioral design pattern – Visitor pattern



# 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, Composite, Bridge, Façade, **Decorator**, **Flyweight**

## Behavioral patterns

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

# Decorator design pattern



# Decorator design pattern

- Can attach additional functionalities to an object dynamically
- Basic idea: enclose the object in another object (the decorator) that adds the additional functionality
- Implementation can look the same as a proxy
  - Intent is somewhat different

# Case study – fraud detecting payment processor

- Check for fraud before paying

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

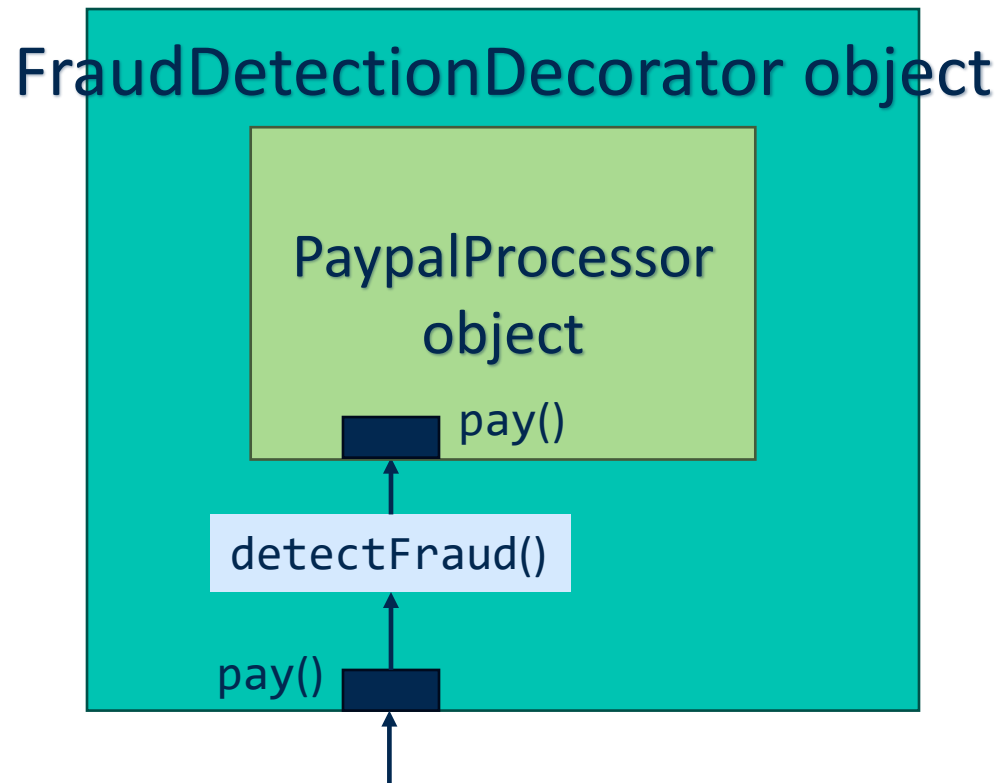
class PaypalProcessor implements PaymentProcessor
{
    void pay (double amount) {
        // paypal functionality
    }
}
```

***Detect fraud***

# Fraud detection decorator for payment processor

- Create an abstract class for the decorator that wraps the inner payment processor
- Extend the abstract decorator as the FraudDetectionProcessor
- Wrap PaymentProcessor objects in FraudDetectionProcessor objects
- Use the FraudDetectionProcessor objects in any place where a PaymentProcessor is needed

# Decorators visually



# Fraud detection decorator for payment processor

- Create an abstract class for the decorator that wraps the inner payment processor

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

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

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

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

# Fraud detection decorator for payment processor

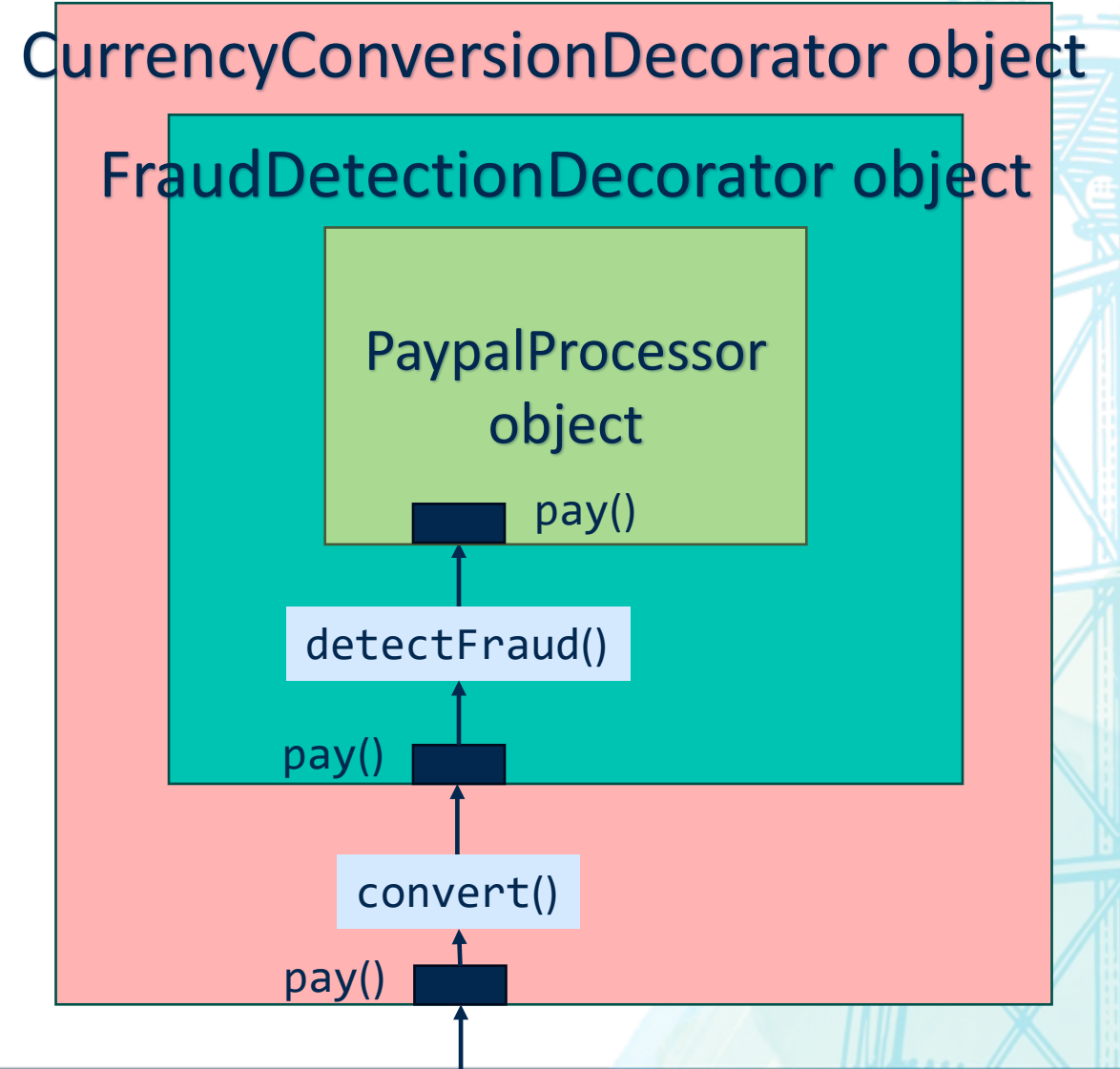
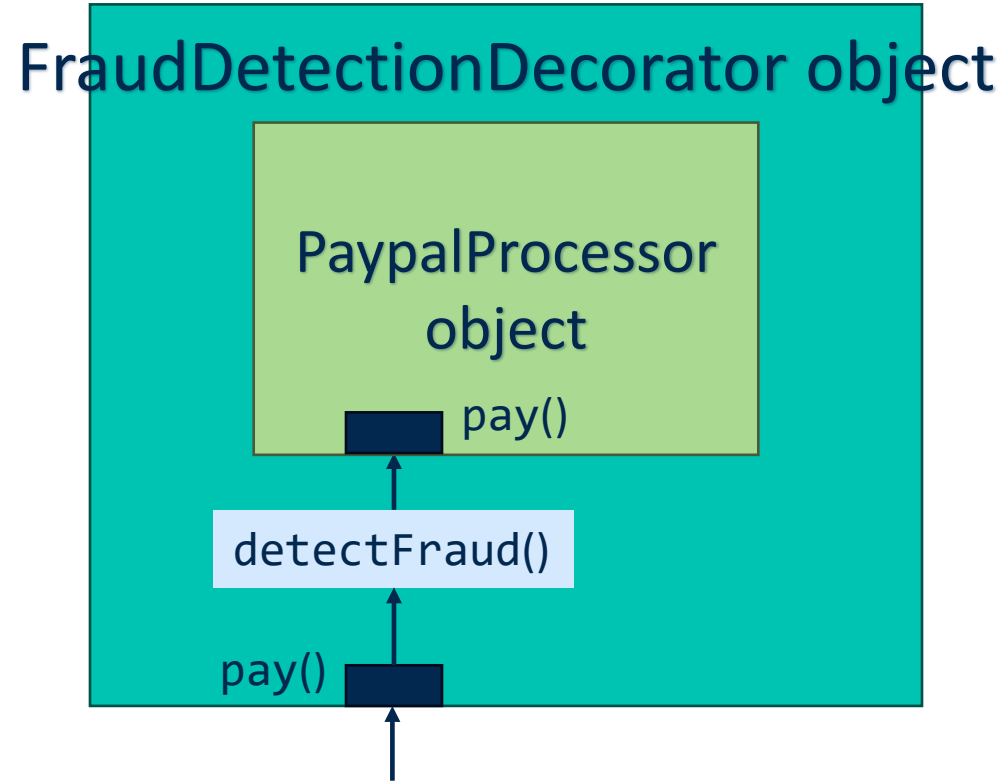
- Wrap PaymentProcessor objects in FraudDetectionDecorator objects
- Use the FraudDetectionDecorator objects in any place where a PaymentProcessor is needed

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

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

public static void main(String[] args) {
    PaypalProcessor paypalProc = new PaypapProcessor();
    FraudDetectionDecorator fraudProc = new
        FraudDetectionDecorator(paypalProc);
    fraudProc.processPayment(200.0);
}
```

# Decorators visually



# Combining adapter and decorator patterns

CurrencyConversionDecorator object

FraudDetectionDecorator object

PaypalProcessor  
object

pay()

detectFraud()

pay()

convert()

pay()

CurrencyConversionDecorator object

FraudDetectionDecorator object

StripeProcessor  
object

makePayment()

pay()

detectFraud()

pay()

convert()

pay()

# Isn't this a proxy??

<https://www.cs.unc.edu/~stotts/GOF/hires/pat4gfs0.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?

```
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

# Flyweight design pattern



# Case study – game system

- Two types of terrain tiles
- Each tile contains a sprite
- Each sprite consists of two images of 4 KB
- ***What happens if there are thousands of tiles***

```
class TerrainTile {
    private int xLoc;
    private int yLoc;
    private Sprite terrainSprite;
    public TerrainTile(String spriteLocation) {
        this.terrainSprite =
        Sprite.loadSprite(spriteLocation);
    }
}

class IndoorTerrainTile extends TerrainTile {
    public IndoorTerrainTile() {
        super("sprites/indoor/");
    }
}

class OutdoorTerrainTile extends TerrainTile {
    public IndoorTerrainTile() {
        super("sprites/outdoor/");
    }
}

class Sprite {
    private Image spriteImage; // 4 KB
    private Image closeUpSpriteImage; // 4 KB
    // getters and setters
    static Sprite loadSprite(String path) {
        // Load both images
    }
}
```

# Flyweight design pattern

- Aims to reduce memory consumption
- Separate ***shared*** and ***unique*** state
- Shared state goes into a Flyweight object and is shared by all objects of the same type
- Constructor that creates the terrain sprites from the images

***Unique state***

*// Without flyweight design pattern -> memory wastage*

```
class TerrainTile {
```

***Shared state***

```
    private int xLoc,
```

```
    private int yLoc;
```

```
    private Sprite terrainSprite;
```

```
    public TerrainTile(String spriteLocation) {
```

```
        this.terrainSprite =
```

```
        Sprite.loadSprite(spriteLocation);
```

```
    }
```

```
}
```

```
class IndoorTerrainTile extends Terrain {}
```

```
class OutdoorTerrainTile extends Terrain {}
```

*// With flyweight design pattern*

```
class Flyweight {
```

```
    private Sprite terrainSprite;
```

```
    public FlyWeight(String key) {
```

```
        if (key == "indoor") {
```

```
            terrainSprite = Sprite.load("sprites/indoor");
```

```
        } else { terrainSprite =
```

```
        Sprite.load("sprites/outdoor"); }
```

```
        public void render(TerrainTile tile) { ..}
```

```
        // getters and setters
```

# Flyweight design pattern

- How does the original object access the shared state?
- FlyweightFactory class creates flyweight objects only if it already isn't created
- Maintains a map of terrain type vs. flyweight object

```
// With flyweight design pattern
class Flyweight {
    private Sprite terrainSprite;
    public FlyWeight(String key) { // create sprite
        images
    }
    public void render(TerrainTile tile) { ..}
}

class TerrainTile {
    private int xLoc;
    private int yLoc;
    // getters and setters
}

class FlyweightFactory {
    private static Map<String, Flyweight>
    flyweights = new HashMap<>();

    public static Flyweight getFlyweight(String
    key) {
        if (!flyweights.containsKey(key)) {
            flyweights.put(key, new Flyweight(key));
        }
        return flyweights.get(key);
    }
}
```

# Putting it all together

```
class Flyweight {
    private Sprite terrainSprite;
    public FlyWeight(String key) { // create sprite
images
        // getters and setters
    }

    class TerrainTile {
        private int xLoc;
        private int yLoc;
        // getters and setters
    }
    class IndoorTerrainTile extends TerrainTile {}
    class OutdoorTerrainTile extends TerrainTile {}

    class FlyweightFactory {
        private static Map<String, Flyweight>
flyweights = new HashMap<>();

        public static Flyweight getFlyweight(String
key) {
            if (!flyweights.containsKey(key)) {
                flyweights.put(key, new Flyweight(key));
            }
            return flyweights.get(key);
        }
    }
}
```

```
int main(void) {
    // Place an indoor tile
    IndoorTerrainTile indoorTile = new
IndoorTerrainTile();
    indoorTile.setX(10);
    indoorTile.setY(10);

    Flyweight indoorTileIcon =
FlyweightFactory.getFlyweight("indoor");

    indoorTileIcon.render();

    // Place outdoor tile
    OutdoorTerrainTile outdoorTile = new
OutdoorTerrainTile();
    // set X and Y
    Flyweight outdoorTileIcon =
FlyweightFactory.getFlyweight("outdoor");
    outdoorTileIcon.render(outdoorTile);
}
```

# Characteristics

- Prevent duplication of state
- Shared state ***must*** be immutable
- May introduce runtime costs from indirection



# Sample design pattern question

- You are designing an email system for your company. The email system automatically must insert the company header and footer images to each email that is being sent. The company header and footer images are 16 KB each. You expect the email system to store tens of thousands of emails. Moreover, some users can enable optional features such as spell-check annotations and keyword highlighting that stores additional metadata with the email and influences how the email is rendered on the email screen (spell-check feature highlights misspelled words in red, for example). You want future developers to be able to easily add new optional features as well. How would you design this system?

# 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, Composite, Decorator, Bridge, Façade, Flyweight

## Behavioral patterns

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

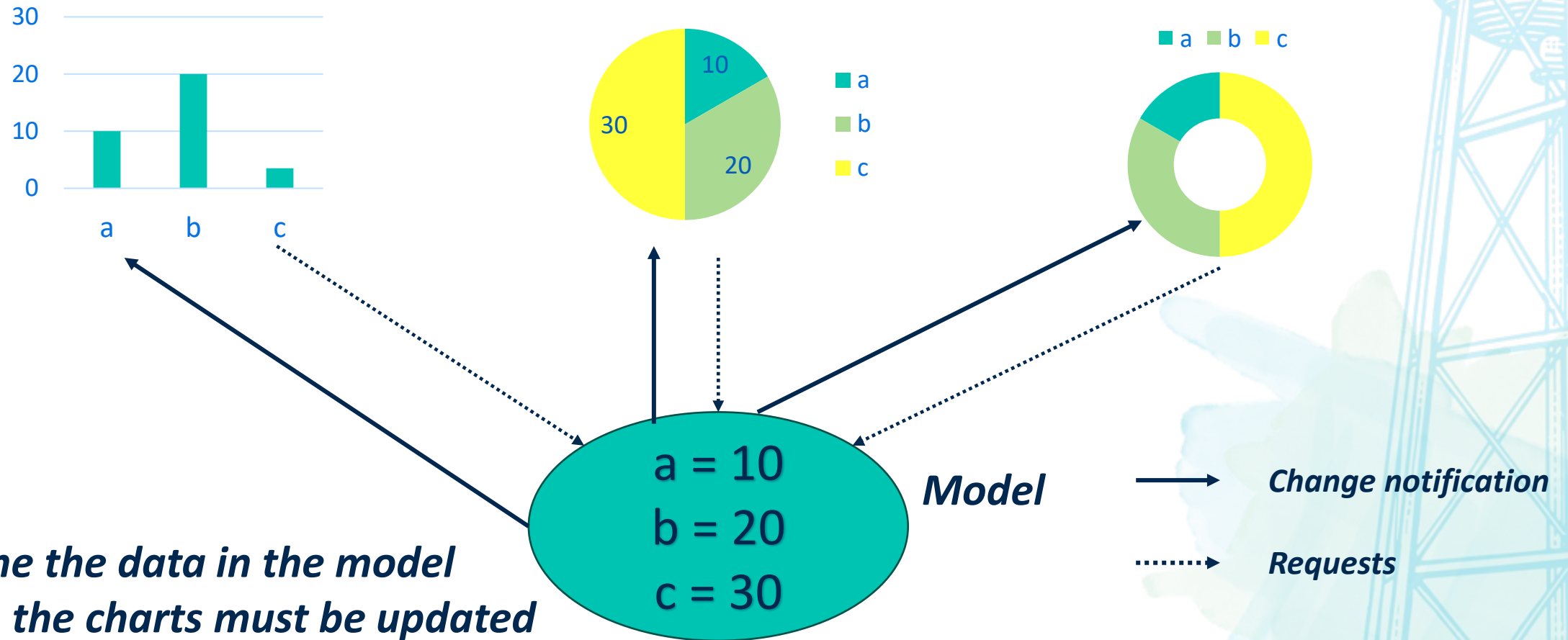
# Behavioral pattern

- 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



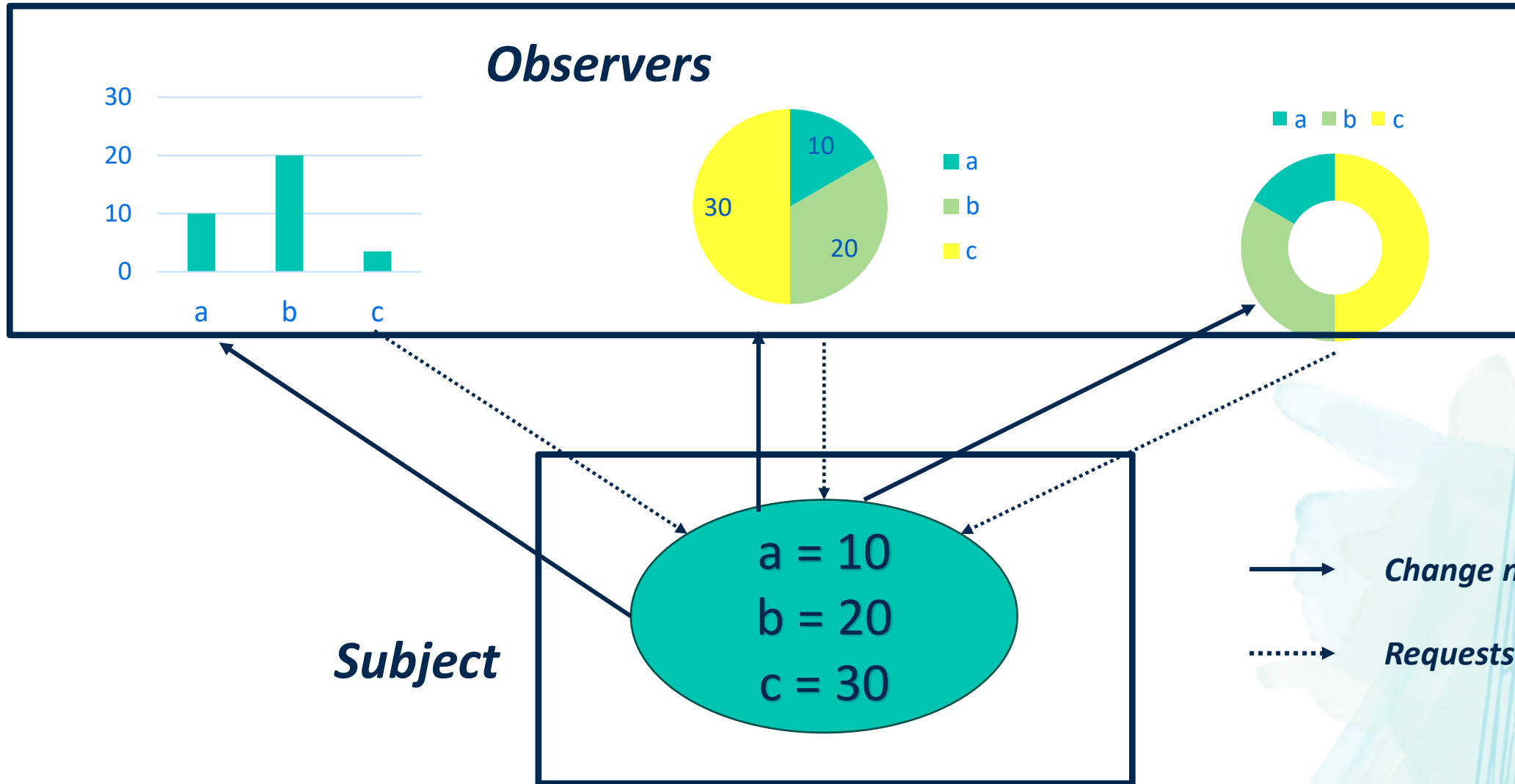
# Case study - Graphical views for application data



# 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

# Design patterns



# Announcements

- Design patterns quiz will close on Friday EOD
- Xingming will conduct the revision lecture
- Final will have 2 design/security questions and 2-3 smaller questions

# 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, Composite, Decorator, Bridge, Façade, Flyweight

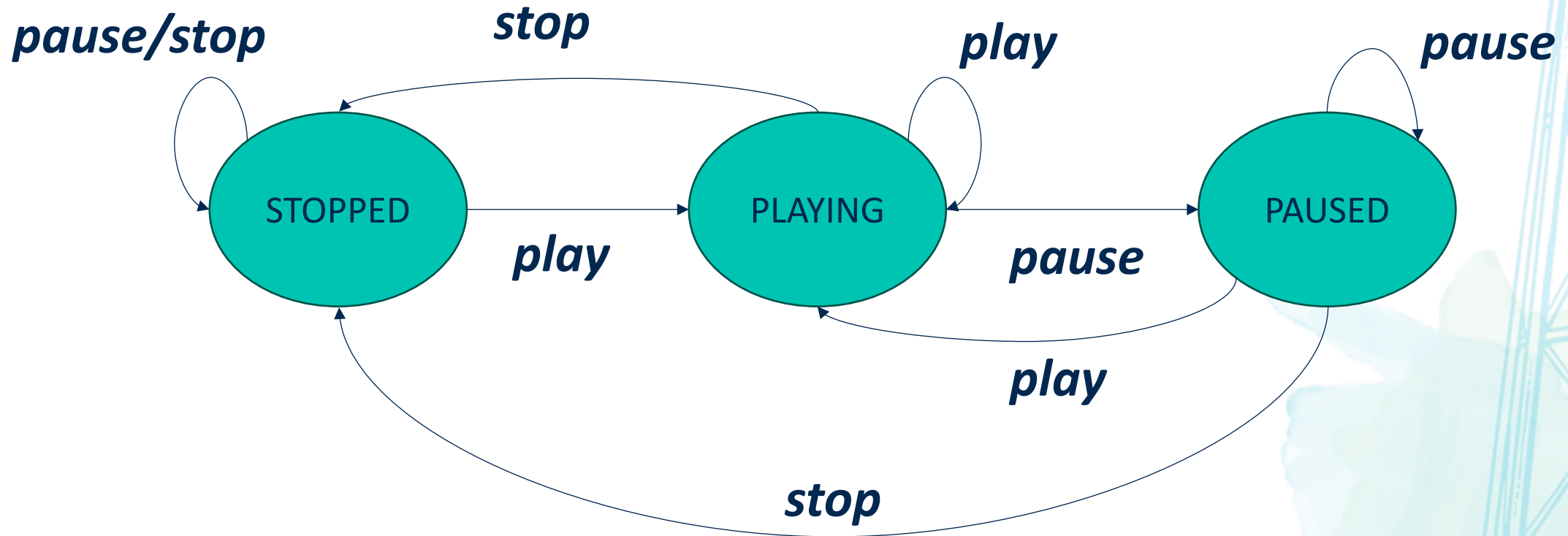
## Behavioral patterns

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

# 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.getDefault(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

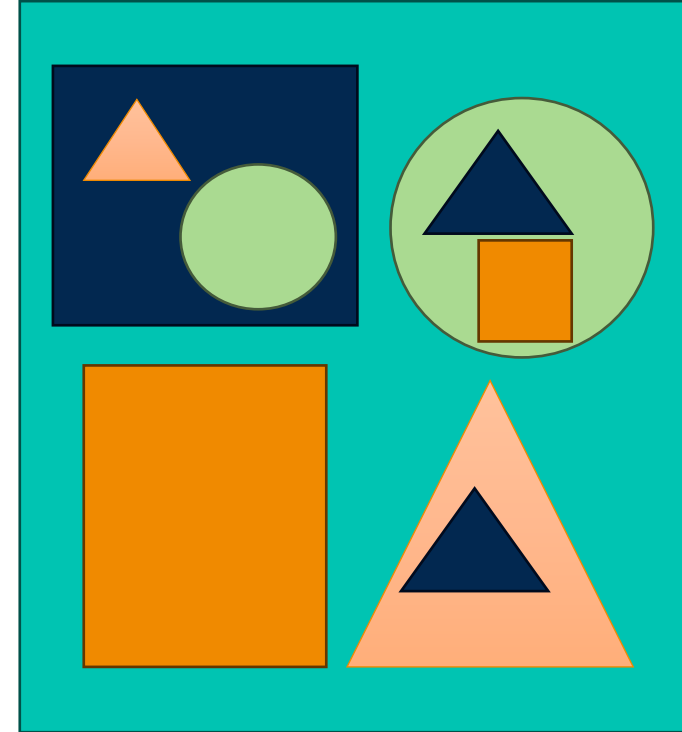
- Fundamental technique for code reuse
- Leads to an “inverted” control structure
  - Parent class calls the operations of the child class and not the other way around
- Template methods can be
  - *Hooks*: the base class provides empty implementation and subclasses can **optionally** implement them
  - *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



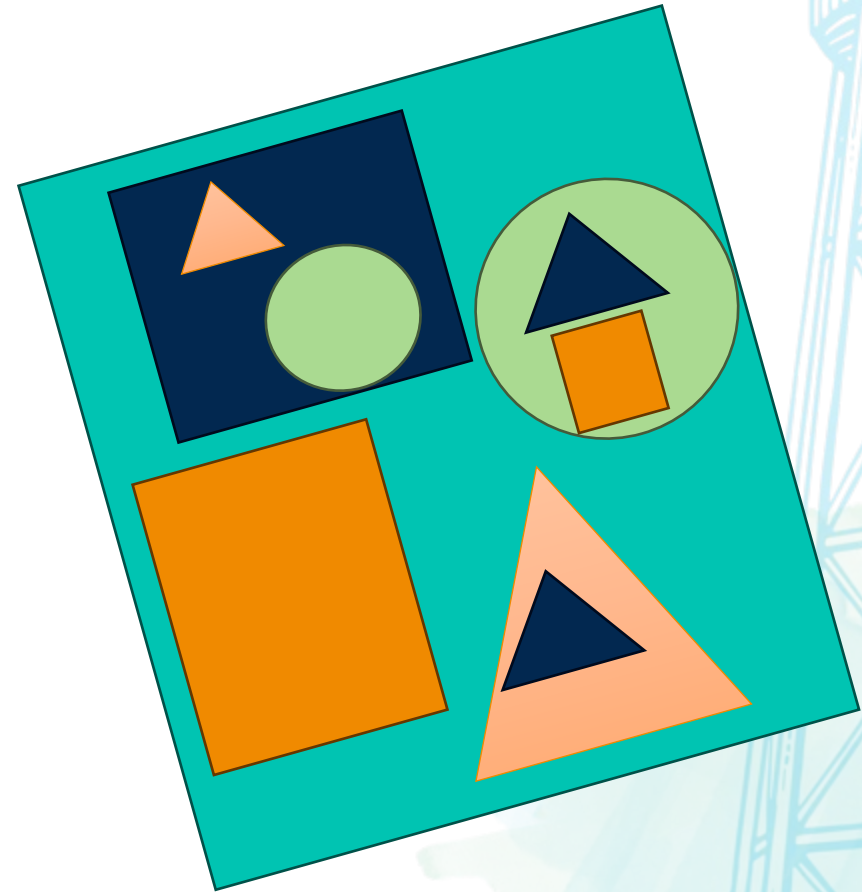
# Case study – transform nested shapes

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



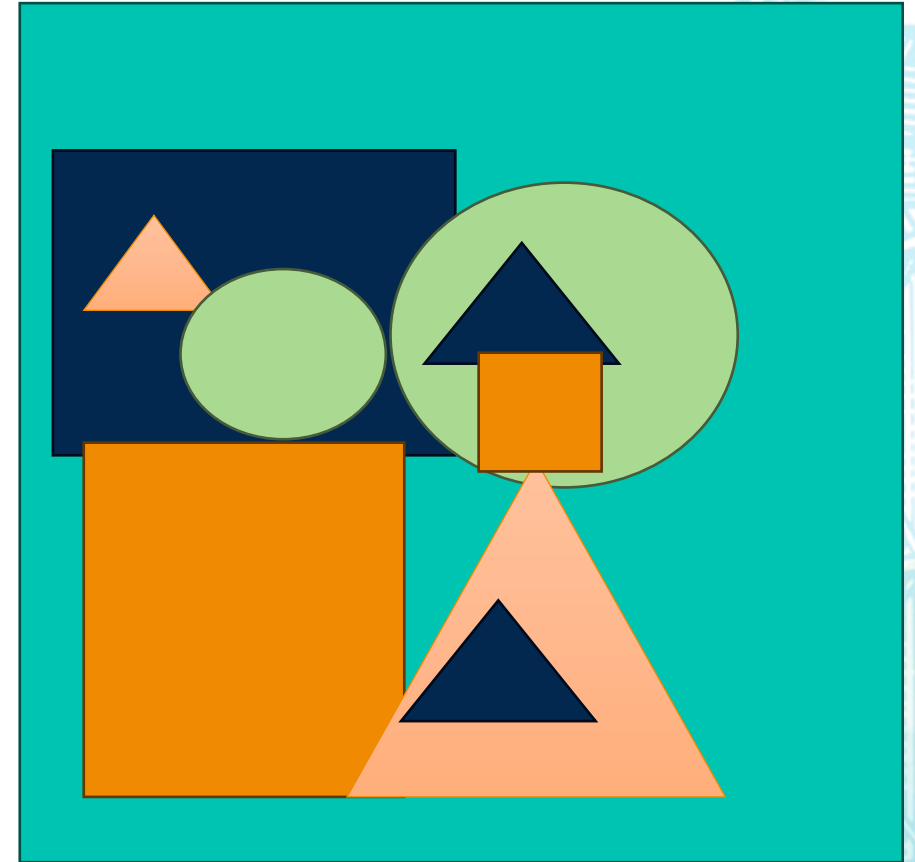
# Case study – transform nested shapes

- Shape operations
  - Rotate



# Case study – transform nested shapes

- Shape operations
  - Scale
  - *Translate*
  - ... *any other operation*



# Design option 1

- Each shape has a translate method

```
class Point {private int x; private int y; }

abstract class Shape {}

class Triangle extends Shape {
    private Point p1;
    private Point p2;
    private Point p3;

    public void translate(int x, int y) {
        p1.setX(p1.getX() + x, p1.getY() + y);
        // for p2 and p3
    }
}

class CompoundShape extends Shape {
    private List shapeList; // getters, setters
    public void translate(int x, int y) {
        for (Shape shape: shapeList) {
            shape.translate(x, y);
        }
    }
}

Shape someShape = ...//
someShape.translate(10, 10);
```

# Design option 1

- Each shape has a scale method

```
class Point {private int x; private int y; }  
abstract class Shape {}  
class Triangle extends Shape {  
    // ...  
    public void translate(int x, int y) { // ...}  
    public void scale(int x, int y) { //... }  
}
```

```
class CompoundShape extends Shape {  
    private List shapeList; // getters, setters  
    public void translate(int x, int y) {  
        for (Shape shape: shapeList) {  
            shape.translate(x, y);  
        }  
    }  
    public void scale(int x, int y) {  
        for (Shape shape: shapeList) {  
            shape.scale(x, y);  
        }  
    }  
}
```

```
Shape someShape = ...//  
someShape.translate(10, 10);  
someShape.scale(2, 2);
```

# Disadvantages

- The logic for the `translate`, `scale`, and any other operations are spread across many shape classes
- Changing how translation works needs to update ***all*** shape classes
- What if you want to add a new operation but can't modify the `Shape` class?
  - Shape class must envision all possible operations when being designed

# Visitor design pattern

- Encapsulate the operational logic in a “Visitor” class
- Abstract visitor class provides concrete implementation for the visit method for CompoundShape
- accept method is the “glue” that holds this design pattern together
  - Ensures the Visitor object visits each Shape object

```
abstract class ShapeVisitor {  
    void visit(Circle circle) {}  
    void visit(Rectangle rectangle) {}  
  
    void visit(CompoundShape c) {  
        for (Shape shape: c.getShapeList()) {  
            shape.accept(this);  
        }  
    }  
}
```

*We will discuss this in a couple of slides*

# Visitor design pattern

- Concrete visitor subclasses implement the particular visitor logic
- The visitor logic is encapsulated in its own visitor class
- Now, all we have to do is ensure the Shape objects “accept” a Visitor object

```
class TranslationVisitor extends ShapeVisitor {  
  
    int x_off; int y_off;  
    TranslationVisitor(int x, int y) {  
        this.x_off = x; this.y_off = y;  
    }  
    @Override  
    public void visit(Triangle t) {  
        t.setP1(t.getP1()+x_off...);  
        /// translate 3 points  
    }  
  
    @Override  
    public void visit(Rectangle rectangle) {  
        // translate 4 points  
    }  
  
    // No need to override visit for  
    CompoundShape  
}
```

# Visitor design pattern

- Allow the parent class for all nodes to ***accept*** a Visitor
- Provide concrete implementations of the accept method for each type
  - Just invokes visit on the Visitor
- The accept method invokes the visitor on the shape object
- Note: it accepts an abstract Visitor type object

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

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

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

# Visitor design pattern – putting it all together

```
abstract class ShapeVisitor {  
    void visit(Circle circle) {}  
    void visit(Rectangle rectangle) {}  
    void visit(CompoundShape c) {  
        for (Shape shape: c.getShapeList()) {  
            shape.accept(this);  
        }  
    }  
}
```

**Visitors encapsulate the operation logic**

```
class TranslationVisitor extends ShapeVisitor {  
    TranslationVisitor(int x, int y) { //.. }  
    public void visit(Triangle t) {  
        /// translate 3 points  
    }  
    public void visit(Rectangle rectangle) {  
        // translate 4 points  
    }  
}
```

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

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

**Shapes provide an accept method to accept a visitor and invoke it**

# Visitor design pattern – putting it all together

```
abstract class ShapeVisitor {
    void visit(Circle circle) {}
    void visit(Rectangle rectangle) {}
    void visit(CompoundShape c) {
        for (Shape shape: c.getShapeList()) {
            shape.accept(this);
        }
    }
}

class TranslationVisitor extends ShapeVisitor {
    TranslationVisitor(int x, int y) { //.. }
    public void visit(Triangle t) {
        /// translate 3 points
    }
    public void visit(Rectangle rectangle) {
        // translate 4 points
    }
}
```

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

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

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

main() {
    Rectangle rect = new ...;
    Triangle tria = new ...;
    CompoundShape cShape = new ...;
    cShape.setShapeList(new ArrayList(rect, tria));
    ShapeVisitor translationVisitor = new
    TranslationVisitor();
    translationVisitor.visit(cShape);
}
```

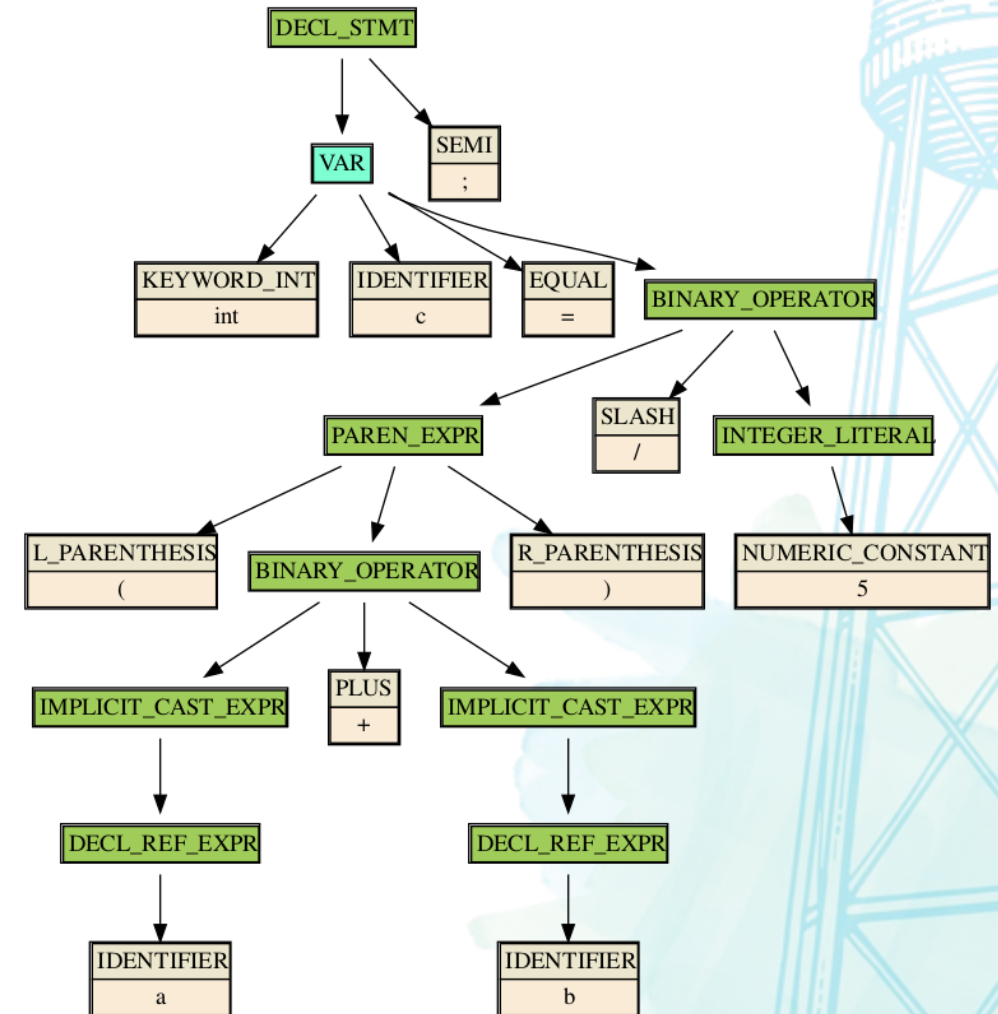


# Visitor design pattern

- Allows adding new operations to a group of related objects without modifying their structure
- Separates operations (behavior) from the object structure (elements) they act upon
- Makes it easy to add operations after class is created
- 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



# ASTVisitor

- Create an abstract class to represent an AST node with an abstract accept method
- Create concrete subclasses with concrete accept methods

```
abstract class ASTNode {  
    void accept(ASTVisitor visitor);  
}  
  
class DeclStmt extends ASTNode {  
    /// ... fields and getters/setters  
    void accept(ASTVisitor visitor) {  
        visitor.visit(this);  
    }  
}  
  
class BinaryOperator extends ASTNode {  
    private ASTNode leftNode;  
    private ASTNode rightNode;  
  
    void accept(ASTVisitor visitor) {  
        visitor.visit(this.leftNode);  
        visitor.visit(this.rightNode);  
    }  
}
```

# ASTVisitor

- Define an abstract class  
ASTVisitor
- Define concrete visitors  
subclasses ASTVisitor

```
abstract class ASTVisitor {
    void visit(DeclStmt stmt);
    void visit(BinaryOperator op);
    void visit(BinaryOperator op) {
        op.getLeft().accept(this);
        op.getRight().accept(this);
    }
}

class CodeGenVisitor extends ASTVisitor {
    private String generatedBinaryCode;

    void visit(DeclStmt declStmt) {
        // Create space on the stack
        generatedBinaryCode.append("sub rsp," +
            declStmt.getSize());
    }

    void visit(BinaryOperator op) {
        super.visit(op);
        switch(op.getKind()) {
            case "+": ///;
            case "-": ///;
        }
    }
}
```

# 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, Composite, Decorator, Bridge, Facade, Flyweight
- Behavioral – how objects distribute responsibilities
  - Observer, state, template method, visitor
- Design patterns can be combined to solve complex tasks

# Sample design pattern questions

# Sample design pattern question

You are developing software to support the automated feeding and watering of expensive flowering/fruitlet 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

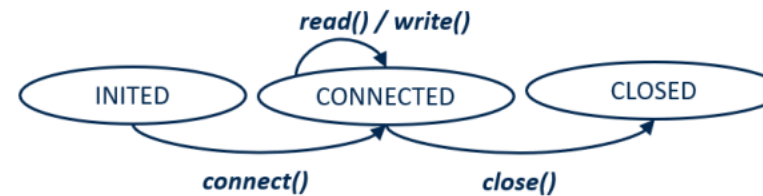
Similarly, the Connection class provides the following API -

- 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.

## 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.

# Summary

- Covered a broad breadth of topics
  - Annotations, reflection, dynamic proxies
  - Microservices
  - Software security
  - Fuzz testing
  - Design patterns
- Homework covered framework design and fuzzing



**Thank you and all the best for the final!**