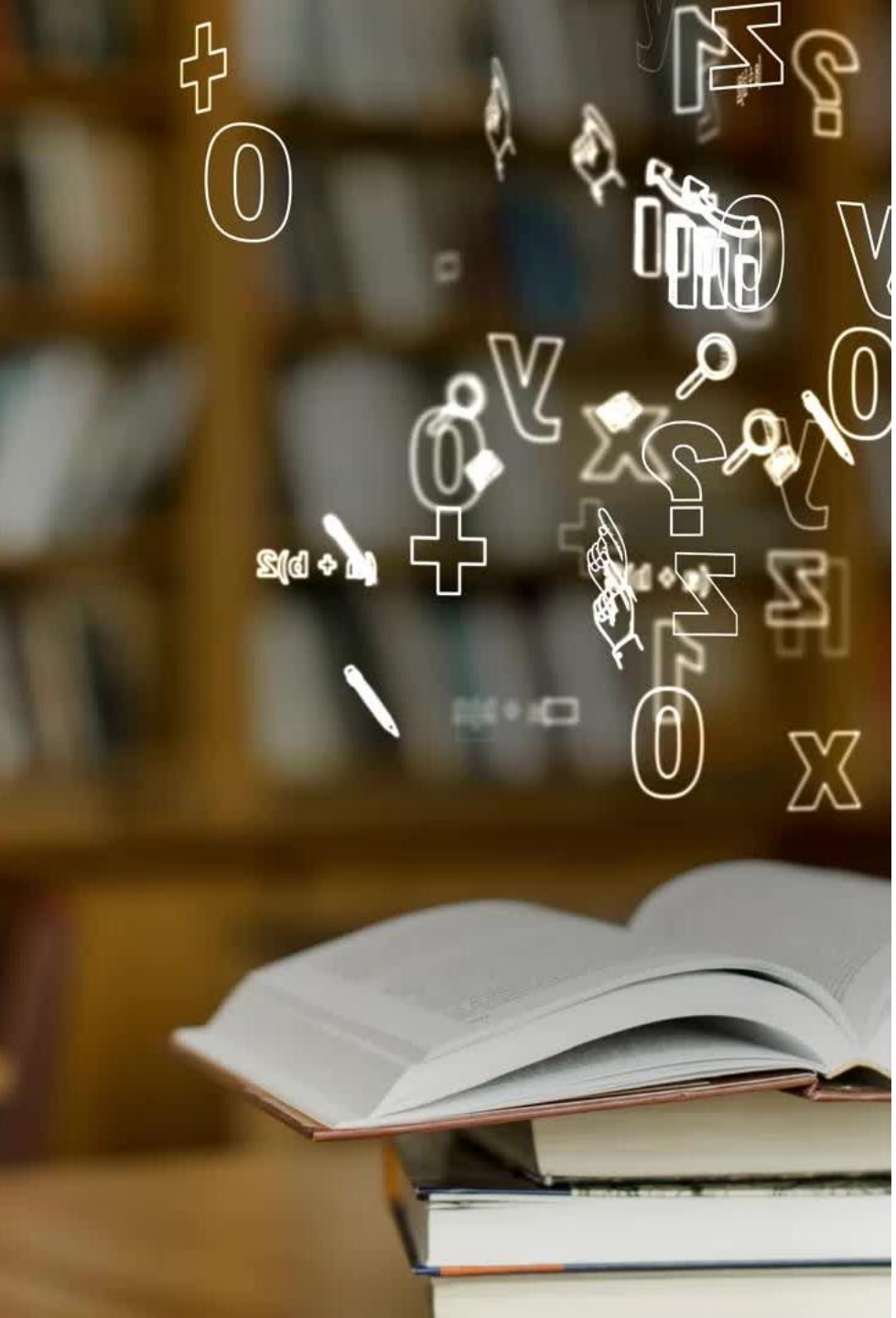
Design Patterns

CS 5010 Fall 2022

Credits: portions of these materials adapted from Maria Zontak's CS 5010 course for Northeastern University



Administrative info

- Codewalk #2
  - due tonight, Wednesday, October 12 @ 11:59pm
- Homework #3
  - Coming out shortly
  - due Monday, October 24th @ 11:59pm.
- Lab next week
- [Optional] Homework #1 Refactor
- Start thinking of who you will partner with for hw 4-6
  - Groups of 2
  - Survey will come out in a few days
  - New “Group” repo

Homework #1 Refactor

- Optional opportunity to increase your score for HW #1
  - Improve your designs (if needed)
  - Fix magic numbers, tests, javadocs, UMLs, etc
- Deadline: Friday, October 14th @ 11:59pm.
  - Any submissions after deadline will not be accepted
- Don't break your existing solutions
  - If you have good tests, you have less danger of this!
- See details in Piazza post @102

Factories

Abstract Factory vs Factory method

- Factory Method pattern:
  - A single method
  - Uses inheritance and relies on subclass to handle desired object instantiation
- Abstract Factory pattern:
  - Encapsulates many factory methods
  - Single responsibility for creating *family* of objects
  - Another class delegates responsibility of object instantiation to Factory class

Factory method

```
class Race {
    public Race() {
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();
        ...
    }
    ...
}
```

```
class TourDeFrance extends Race {
    public TourDeFrance() {
        Bicycle bike1 = new RoadBicycle();
        Bicycle bike2 = new RoadBicycle();
        ...
    }
    ...
}
```

```
class Cyclocross extends Race {
    public Cyclocross() {
        Bicycle bike1 = new MountainBicycle();
        Bicycle bike2 = new MountainBicycle();
        ...
    }
    ...
}
```

Factory method

Problem: re-implementing constructor in every Race subclass to use a different subclass of Bicycle

Factory method

```
class Race {
    Bicycle createBicycle() { return new Bicycle(); }
    public Race() {
        Bicycle bike1 = createBicycle();
        Bicycle bike2 = createBicycle();
        ...
    }
    ...
}
```

Use a Factory method to avoid dependency on specific new kind of Bicycle in constructor

Factory method

```
class TourDeFrance extends Race {
    Bicycle createBicycle() { return new RoadBicycle(); }
    public TourDeFrance() { super() }
    ...
}

class Cyclocross extends Race {
    Bicycle createBicycle() { return new MountainBicycle(); }
    public Cyclocross() { super() }
    ...
}
```

Subclasses can override Factory method and return any subtype of Bicycle

Abstract Factory

Encapsulation: move the factory method into a separate class - a factory object

Advantages:

- Can pass factories around as objects for flexibility
  - Choose a factory at runtime
  - Use different factories in different objects (e.g. races)
- Promotes composition over inheritance
- Supports separation of concerns

Abstract Factory examples

```
class BicycleFactory {
    Bicycle createBicycle() {
        return new Bicycle();
    }
}

class RoadBicycleFactory extends BicycleFactory {
    Bicycle createBicycle() {
        return new RoadBicycle();
    }
}

class MountainBicycleFactory extends BicycleFactory {
    Bicycle createBicycle() {
        return new MountainBicycle();
    }
}
```

Abstract Factory examples

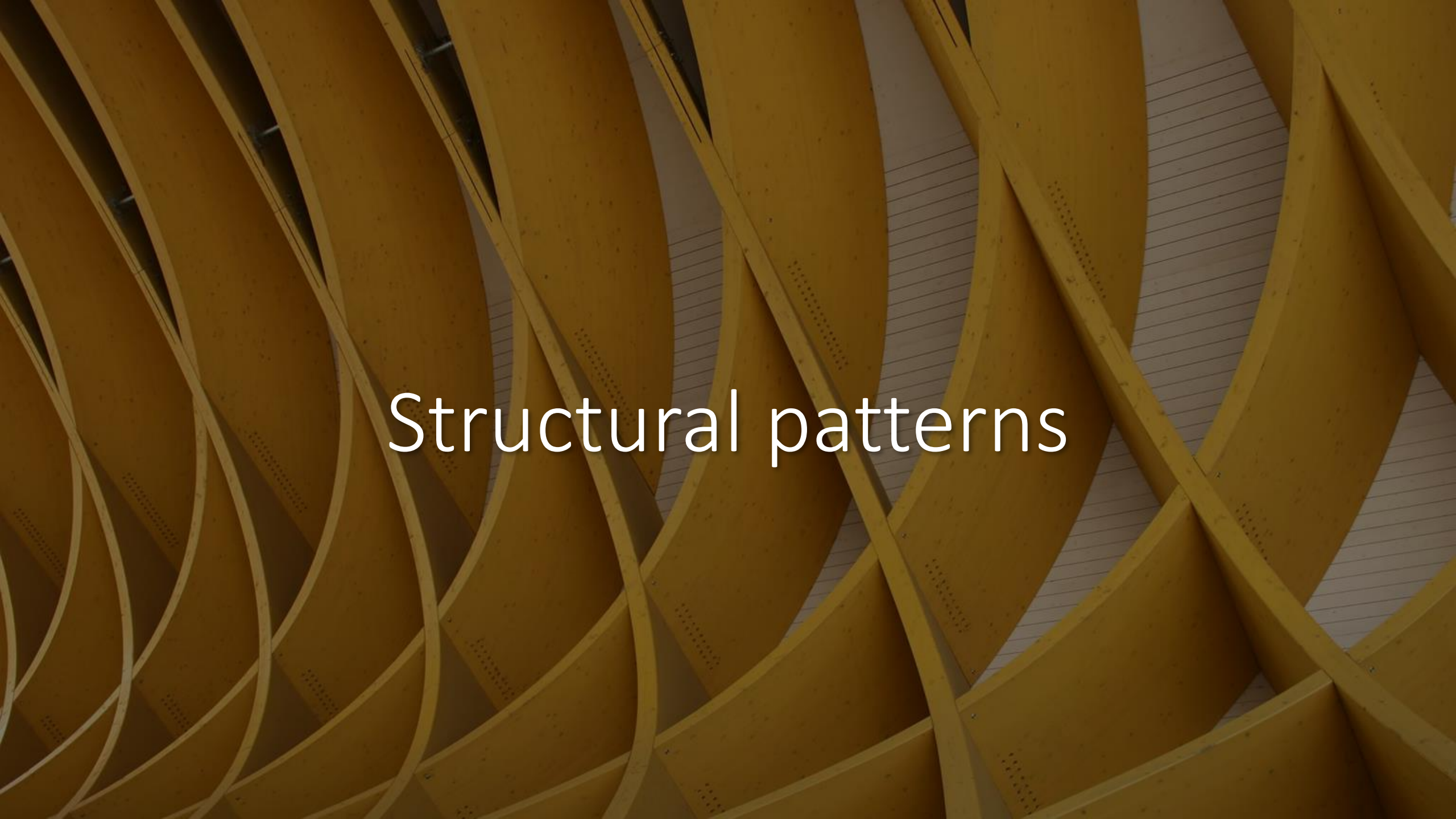
```
class Race {
    BicycleFactory factory;
    public Race(BicycleFactory f) {
        factory = f;
        Bicycle bike1 = factory.createBicycle();
        Bicycle bike2 = factory.createBicycle();
    }
}
```

```
class TourDeFrance extends Race {
    public TourDeFrance() {
        super(new RoadBicycleFactory());
    }
}

class Cyclocross extends Race {
    public Cyclocross() {
        super(new MountainBicycleFactory());
    }
}
```

Abstract Factory: separation of concerns

Separate control over Bicycles and Races:

- Can swap different Factories for different Races
- What about a free race? We can overload a race's constructor with a BicycleFactory argument.



Structural patterns

Structural patterns: Adapter

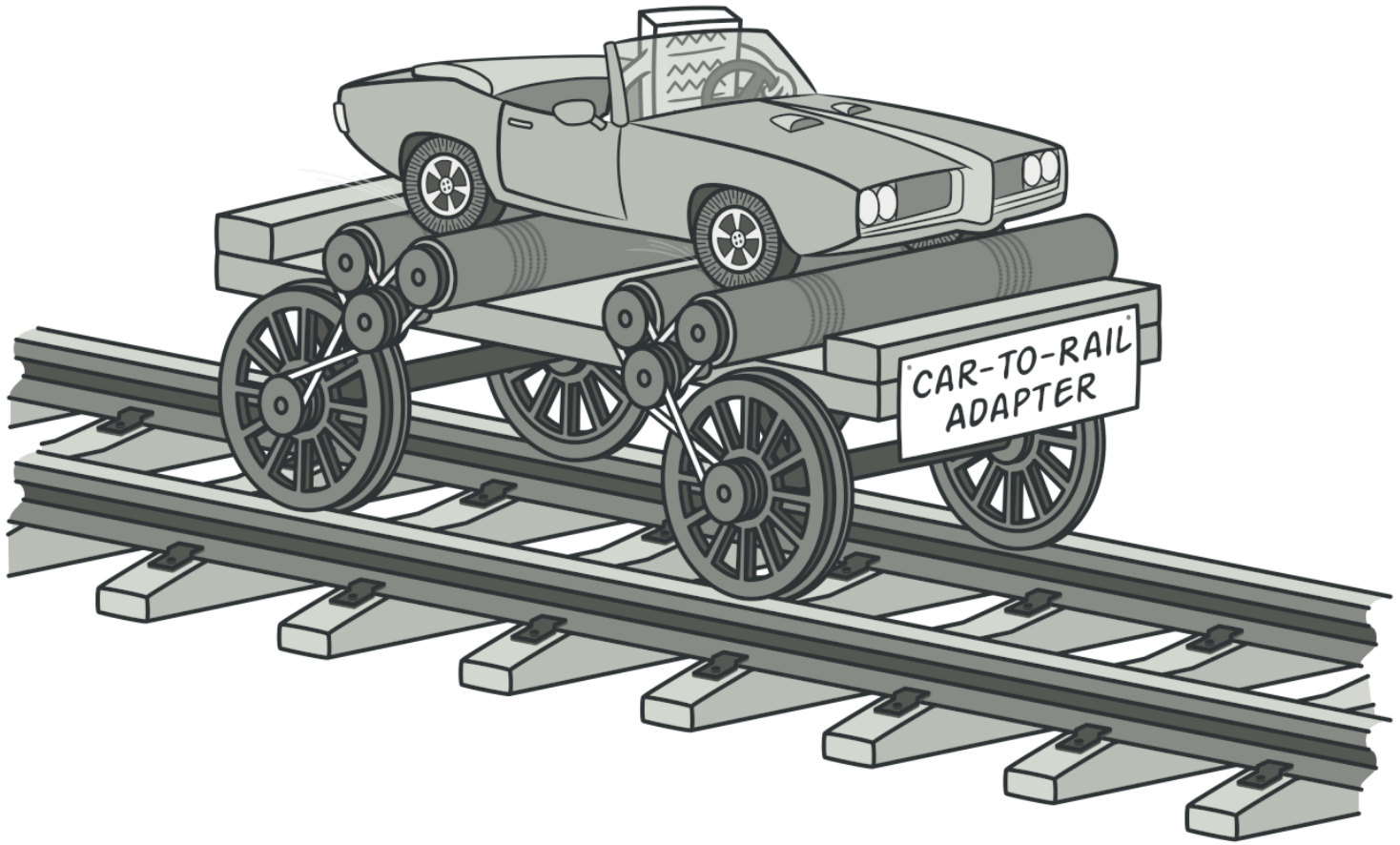


Image credit: https://refactoring.guru/design-patterns/adapter

Structural patterns: Adapter

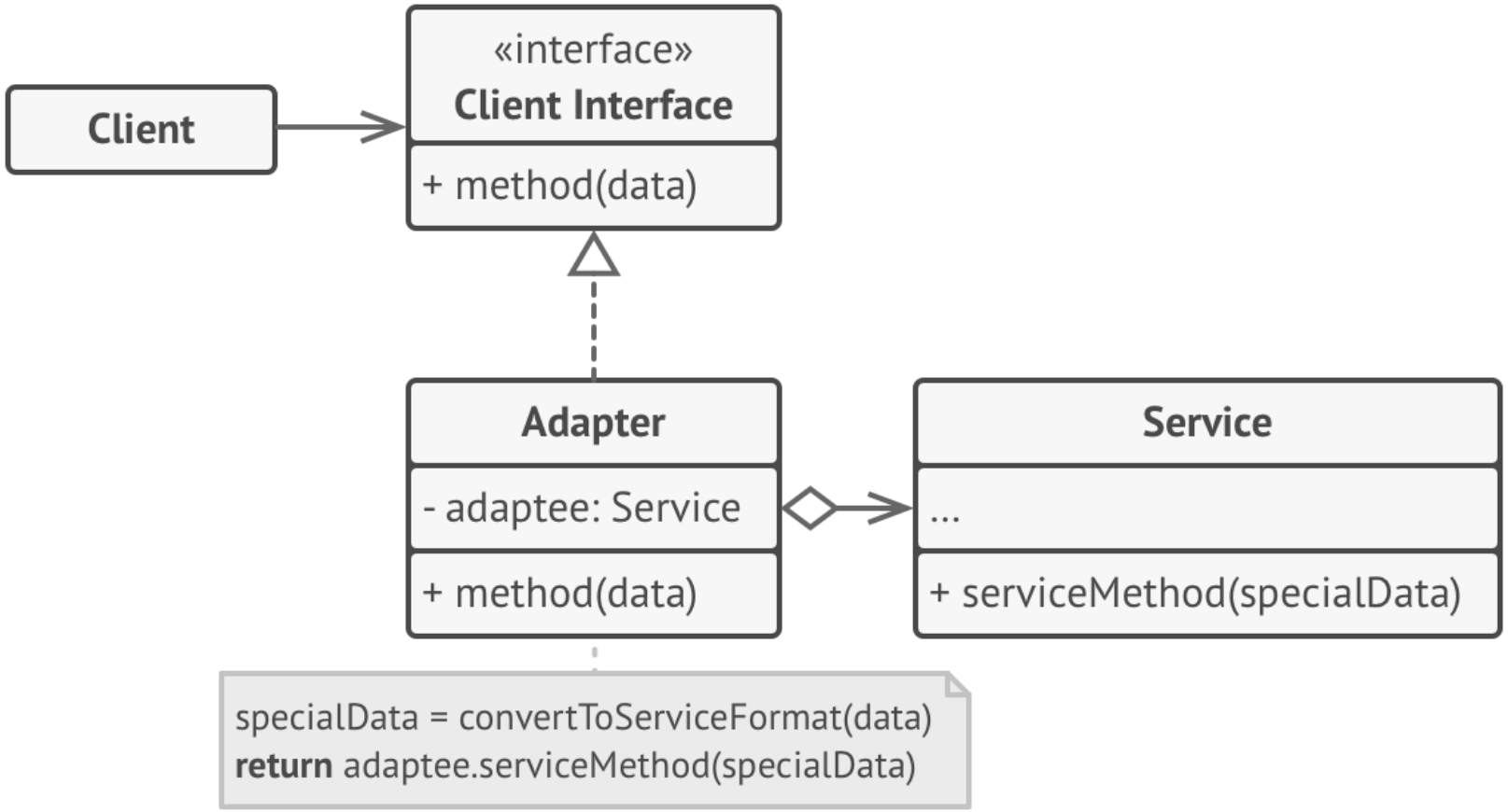


Image credit: https://refactoring.guru/design-patterns/adapter

Adapter / Decorator / Facade

- Adapter
  - When you need to convert one interface to another
  - Ex: Bringing legacy code forward
- Decorator
  - When you need to add functionality to an interface
- Façade
  - Simplify an interface

Structural patterns: Bridge

- Separates an abstraction from its implementation by putting them in separate class hierarchies
- The abstraction delegates to the implementor object.
- Enables the ability configure the abstracted type with an implementor at run-time

Structural patterns: Bridge

Bridge pattern:

- Separate abstraction from implementation
- Create an interface for implementors
- Implementor class provides implementations used by instances

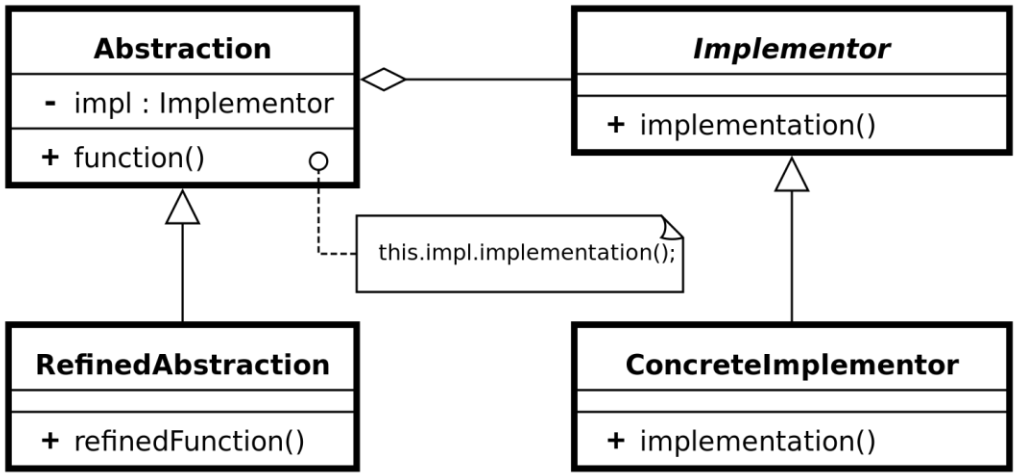


Image credit: https://en.wikipedia.org/wiki/Bridge_pattern

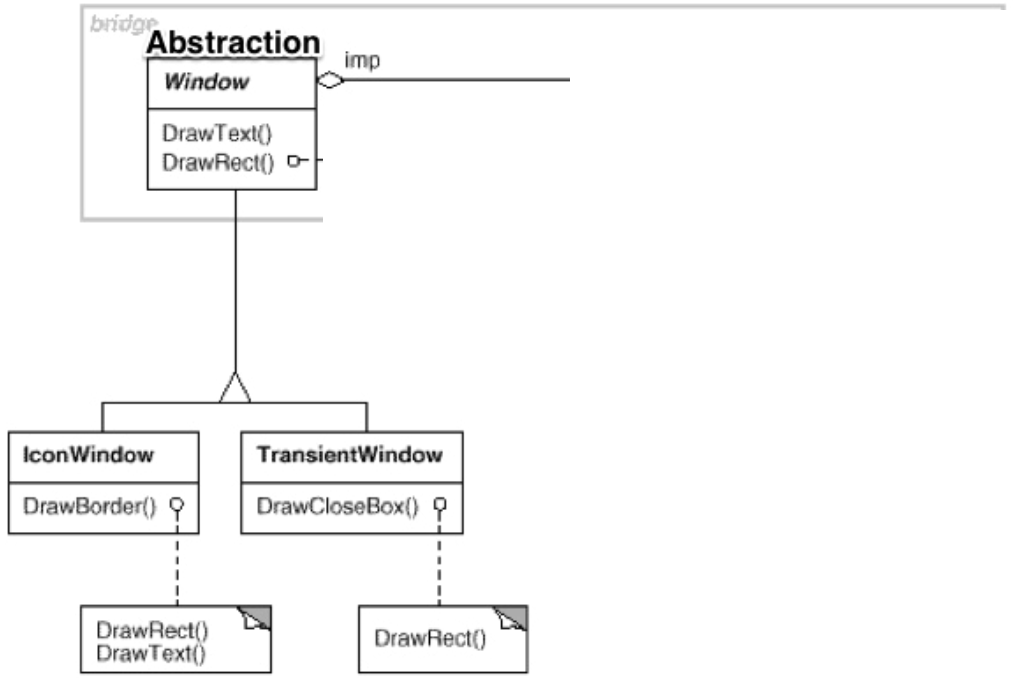
Structural patterns: Bridge

- Consider a UI library that offers Window creation and management functionality.
- Windows as an abstraction have their own logic. Various types of windows have various behaviors.
- Implementation significant: different platforms/operating systems require entirely separate implementations for all types of windows.
- Bridge pattern separates (creates a "bridge" between) *abstraction* and *implementation*.
- Use the Bridge pattern when you want to avoid permanent binding between abstraction and implementation.

Structural patterns: Bridge

Bridge pattern window example:

- Window, IconWindow, TransientWindow are window abstractions
- Implementor subclasses for various platforms
- Window abstraction employs appropriate implementor as imp. Low-level functionality is handled by this object



Structural patterns: Composite

- Treat a group of elements the way you treat a single element
- Structure complex components as a tree-like structure of other composites or primitive components
- Based on "has-a" relationships

Structural patterns: Composite

Composite pattern:

1. Interface describes behavior common to all components
2. "Leaf" element has no sub-components, doesn't delegate
3. Composite elements have sub-elements; interact with them only via the component interface
4. Client interacts with all components via component interface.

Image credit: https://refactoring.guru/design-patterns/composite

Structural patterns: Composite

Favor composition over inheritance.

When and why?

Inheritance is safe when

- Inheritance within a single package is generally safe (same programmers writing subclass and superclass code)
- Classes have been explicitly designed and implemented *to be extended*

However, in general **inheritance violates encapsulation**. One entity's behavior (the subclass) depends on another entity's implementation (the superclass).

Broken example of inheritance

```
public class InstrumentedHashSet<E>
    extends HashSet<E> {
        // The number of attempted insertions
        private int addCount = 0;

        public InstrumentedHashSet() {
        }

        public InstrumentedHashSet(int initCap, float loadFactor) {
            super(initCap, loadFactor);
        }

        @Override public boolean add(E e) {
            addCount++;
            return super.add(e);
        }

       }
```

```
@override public boolean addAll(Collection<? extends E> c) {
            addCount += c.size();
            return super.addAll(c);
        }

        public int getAddCount() {
            return addCount;
        }
```

Composition vs Inheritance

What would we expect `getAddCount` to return?

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(List.of("Snap", "Crackle", "Pop"));
```

Probably our design calls for it to return 3, but it doesn't! It returns 6.

Composition vs Inheritance

- In `HashSet`, `addAll` is implemented by calling `add` on each member of a collection. But our class has overridden `add` to increment the count. So we increment the count by three in our `addAll`, then the inherited `addAll` calls `add` three more times.
- So the subclass logic is broken due to details of the implementation of the superclass. Boo! Encapsulation violation!
- We could fix this behavior in a few ways, still using inheritance, but the point is that we need to know the implementation of the superclass in able to correctly extend it. This is a problem.
- Avoiding "self use" may lead to re-implementation of functionality, which defeats much of the purpose of inheritance in the first place.

A compositional alternative

- Rather than extending the class, we can give our class a private field that references an instance of the class whose functionality we want to use.
- Methods on our class *forward* their requests on to corresponding methods on the component class.
- We can implement `InstrumentedSet` using composition and forwarding.
- Forwarding behavior can be generalized to work for other composites of sets.
- We create `ForwardingSet` to handle composition and forwarding for sets generally.

A compositional alternative

```
public class InstrumentedHashSet<E>  extends FowardingSet<E> {
        private int addCount = 0;

        public InstrumentedSet(Set<E> s) {
            super(s);
        }

        @override public boolean add(E e) {
            addCount++;
            return super.add(e)
        }

        @Override public boolean addAll(Collection<? extends E> c) {
            addCount += c.size();
            return super.addAll(c);
        }

        public int getAddCount() {
            return addCount;
        }
}
```

Our new implementation is similar to the previous subclass, but extends `ForwardingSet` instead.

Composition and forwarding happen in `ForwardingSet` itself

```
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;
    public ForwardingSet(Set<E> s) { this.s = s; }

    public void clear()               { s.clear();            }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()          { return s.isEmpty();   }
    public int size()                 { return s.size();      }
    public Iterator<E> iterator()     { return s.iterator();  }
    public boolean add(E e)           { return s.add(e);      }
    public boolean remove(Object o)   { return s.remove(o)    }
    public boolean containsAll(Collection<?> c) {
        return s.containsAll(c);
    }
    public boolean addAll(Collection<? extends E> c) {
        return s.addAll(c);
    }
```

```
    public boolean removeAll(Collection<?> c) {
        return s.removeAll(c);
    }
    public boolean retainAll(Collection<?> c) {
        return s.retainAll(c);
    }
    public Object[] toArray()          { return s.toArray();     }
    public <T> T[] toArray(T[] a)   {  return s.toArray(a);    }
    @Override public boolean equals(Object o) {
        return s.equals(o);
    }
    @Override public int hashCode()    { return s.hashCode();    }
    @Override public String toString() { return s.toString();    }
}
```

Structural patterns: Decorator

- Also sometimes known as a “wrapper”
- Extends functionality of an object at runtime, without altering the object’s type

Structural patterns: Decorator

- Consider the case of a notification library
- Application has a notifier and uses it to send alerts on some events

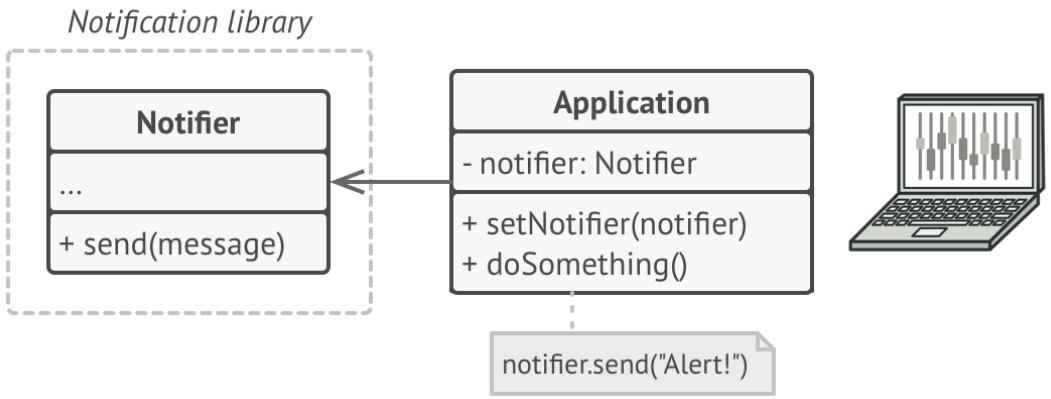


Image credit: https://refactoring.guru/design-patterns/decorator

Structural patterns: Decorator

- We might want a variety of notifier types
- We could subclass with inheritance, but this is limiting
- What if we want arbitrary combinations of behaviors?

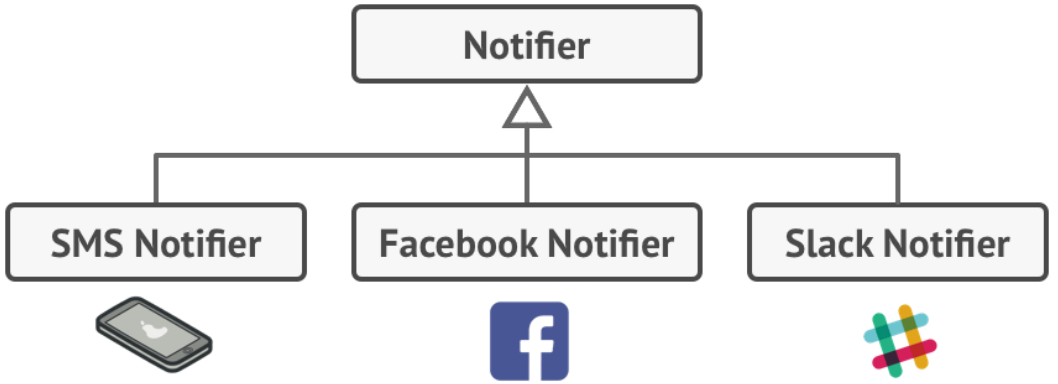


Image credit: https://refactoring.guru/design-patterns/decorator

Relying on inheritance could get complicated

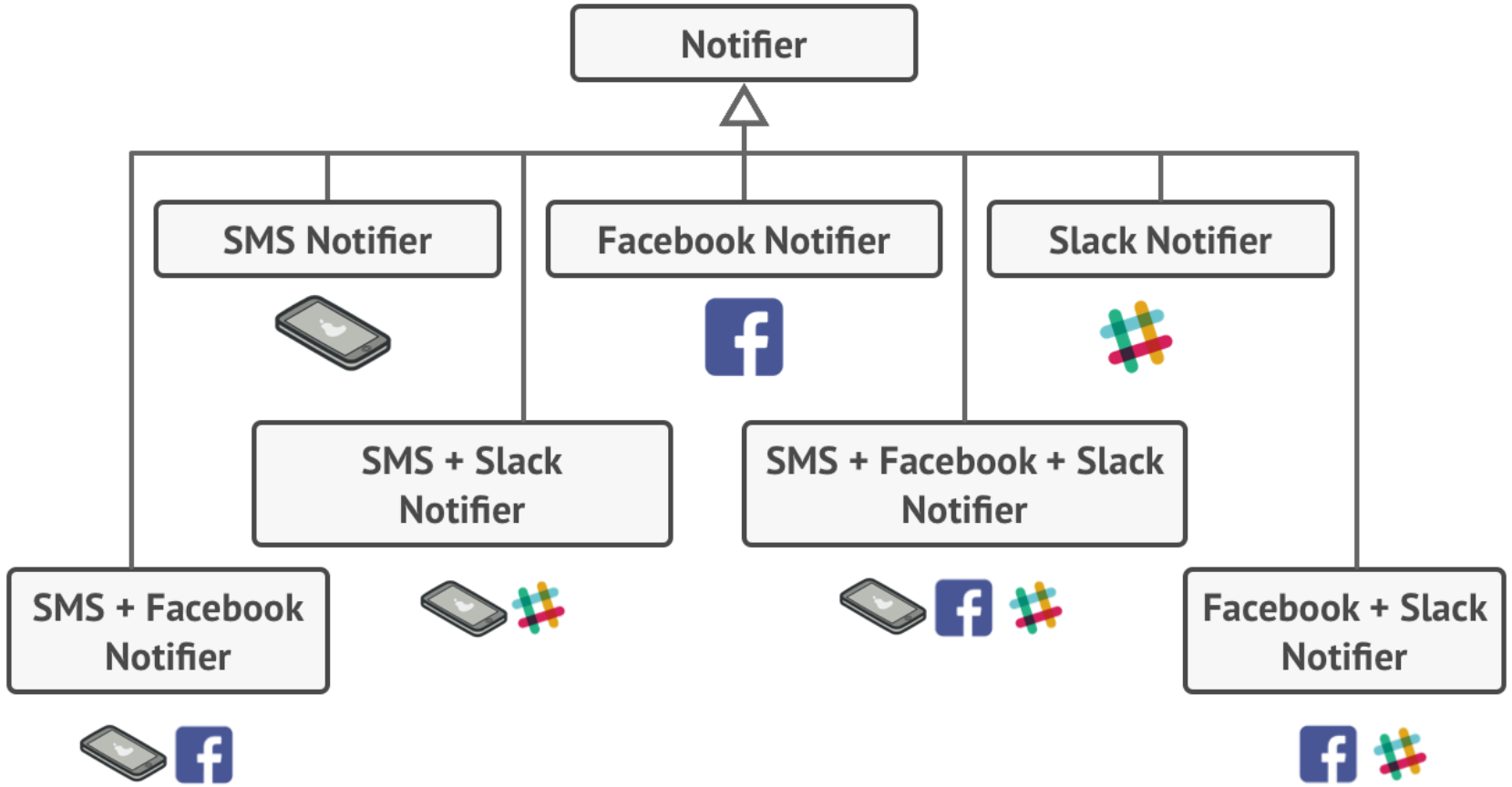


Image credit: https://refactoring.guru/design-patterns/decorator

Decorator Pattern

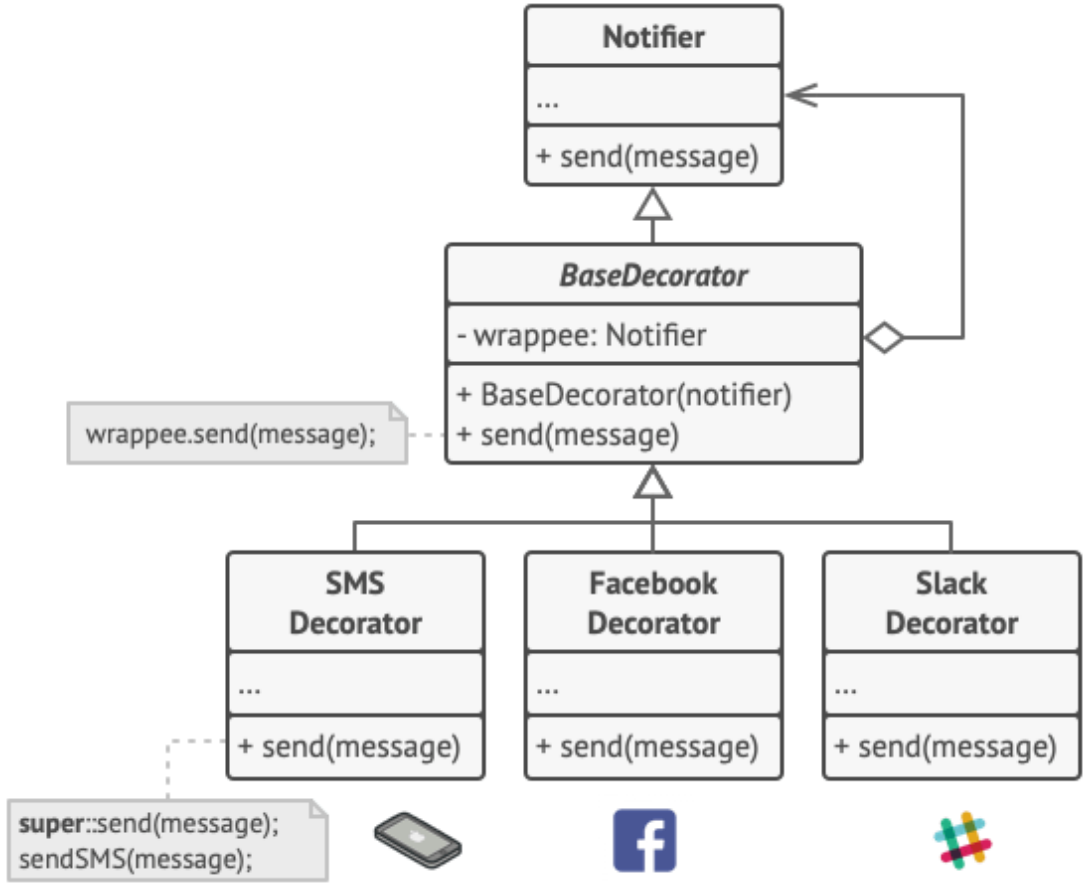


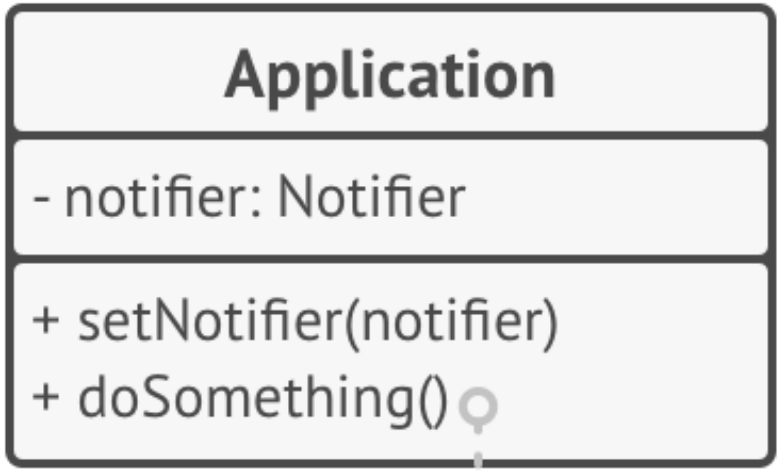
Image credit: https://refactoring.guru/design-patterns/decorator

Structural patterns: Decorator

Each decorator adds its functionality to the component it contains.

The decorator implements the same interface as the original component

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```



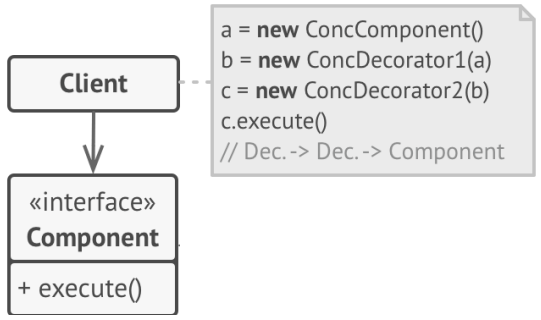
```
notifier.send("Alert!")
// Email → Facebook → Slack
```

Image credit: https://refactoring.guru/design-patterns/decorator

Structural patterns: Decorator

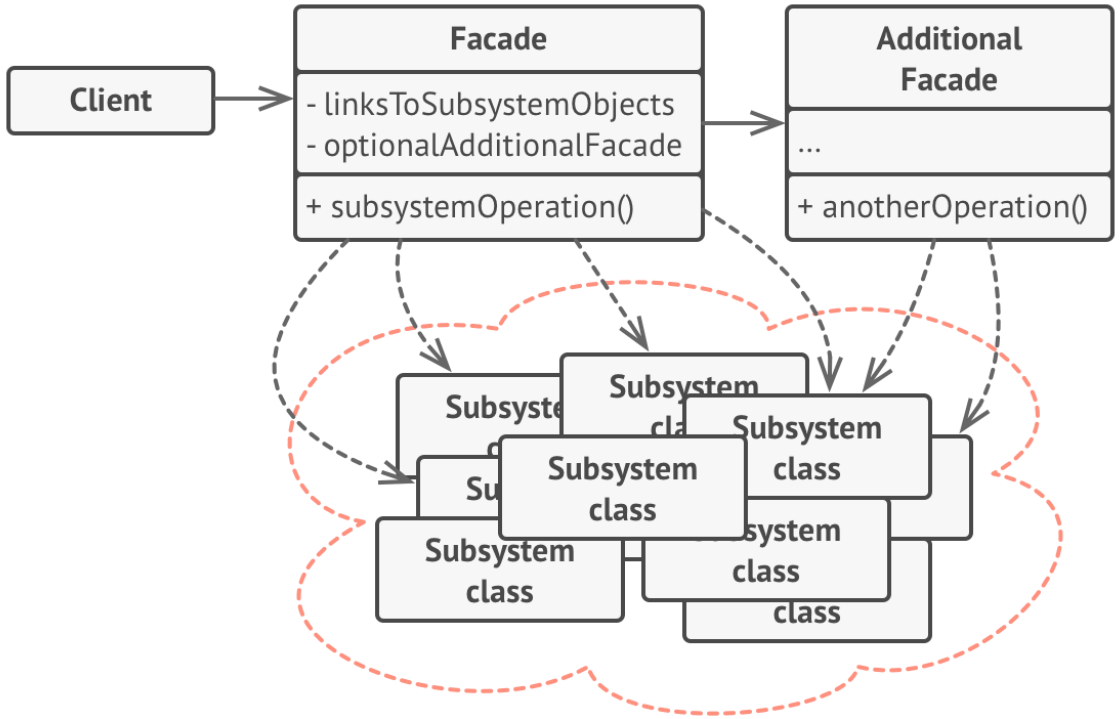
Decorator structure:

- Base decorator implements the same interface as the concrete component to be decorated.
- Base decorator delegates all behaviors to concrete component.
- Concrete decorators add functionality while maintaining interface.



Structural patterns: Facade

- Provide a limited simple interface to a complex area of functionality
- Use composition to access related areas of functionality within A facade class.
- May use additional facades to avoid polluting a single facade




Behavioral patterns

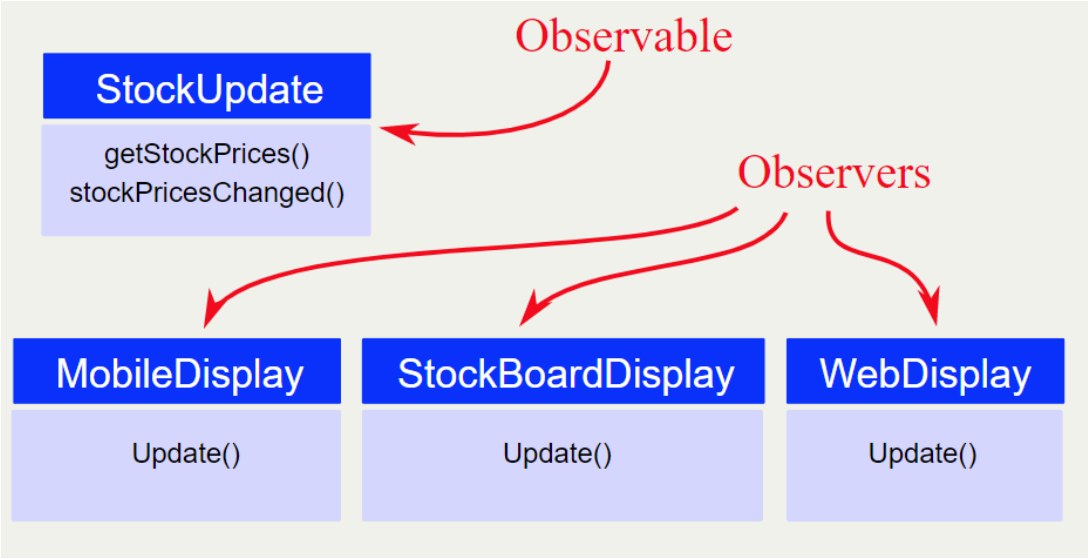
Behavioral pattern example: Observer pattern

The Observer pattern

- Object that might change ("observable" or "subject") keeps a list of interested "observers" and notifies them when something happens
- Observers can react however they like
- Used in many areas
  - Events and listeners for button clicks
  - Java RMI (remote method invocation): observable acts as server providing remote access for an observer to subscribe to it

Observer pattern

- Consider monitoring and displaying updates on stock prices.
- Observable maintains state that holds stock prices.
- Observers await notifications about changes, and display the news.



Observer pattern weakens the coupling

What should `StockUpdate` class know about viewers?

- Observer pattern: call an update() method with changed data

Observer pattern weakens the coupling

```
interface PriceObserver {
    void update(PriceInfo priceInfo);
}
```

Observer pattern weakens the coupling

```
class StockUpdate {
    private PriceInfo priceInfo;
    private List<PriceObserver> observers;

    ... // constructor comes here

    void addObserver(PriceObserver newObs) {
        observers.add(newObs);
    }
```

```
 void stockPricesChanged(PriceInfo newPriceInfo) {
        this.priceInfor = newPriceInfo;
        notifyObservers();
 }

 void notifyObservers() {
        for (PriceObserver obs : observers) {
            obs.update(priceInfo);
        }
 }

}
```

The observer pattern

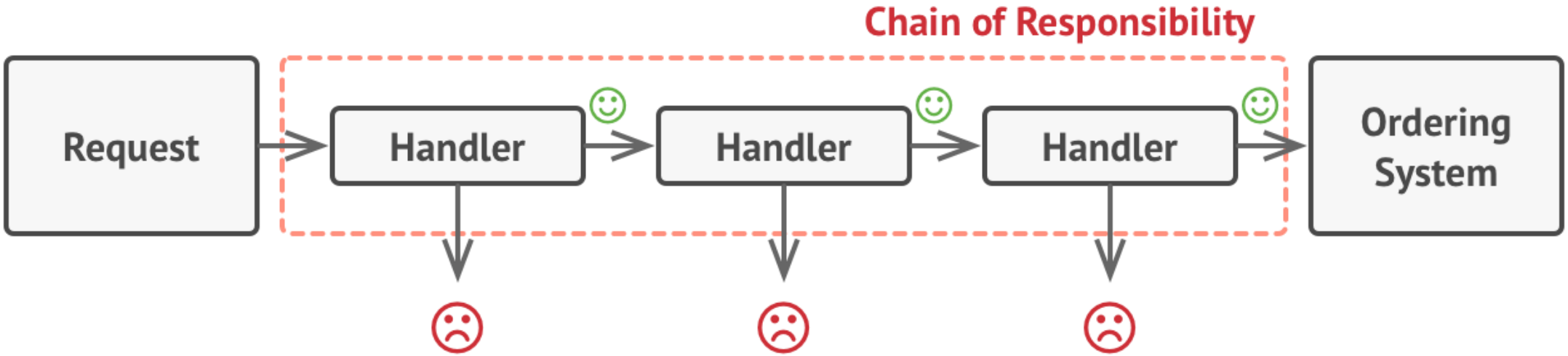
- `StockUpdate` is not responsible for viewer creation
- Application (e.g. `Main`) passes viewers to `StockUpdate` as observers
- `StockUpdate` keeps list of `PriceObservers` and notifies them of changes via callback
- Note: `update`() must pass specific information to (unknown) viewers (that otherwise have no idea about stock prices)

Note on “Push” vs “Pull”

- Observer pattern implements *push* models
- *Pull* model: give all viewers access to `StockUpdate`
- Let them extract whatever data they need
- More flexible, but more tightly coupled


Break time

Behavioral patterns: Chain of Responsibility



- Each stage in processing depends on the next.
- Intermediate processes could terminate sequence.

Image credit: https://refactoring.guru/design-patterns/chain-of-responsibility

Behavioral patterns: Chain of Responsibility

- **Handler** is an interface common to all concrete handlers.
- **BaseHandler** is an optional class for common handling code
- **ConcreteHandlers** actually handle or pass along.
  - Can call parent's handle(request) to move to next in the chain

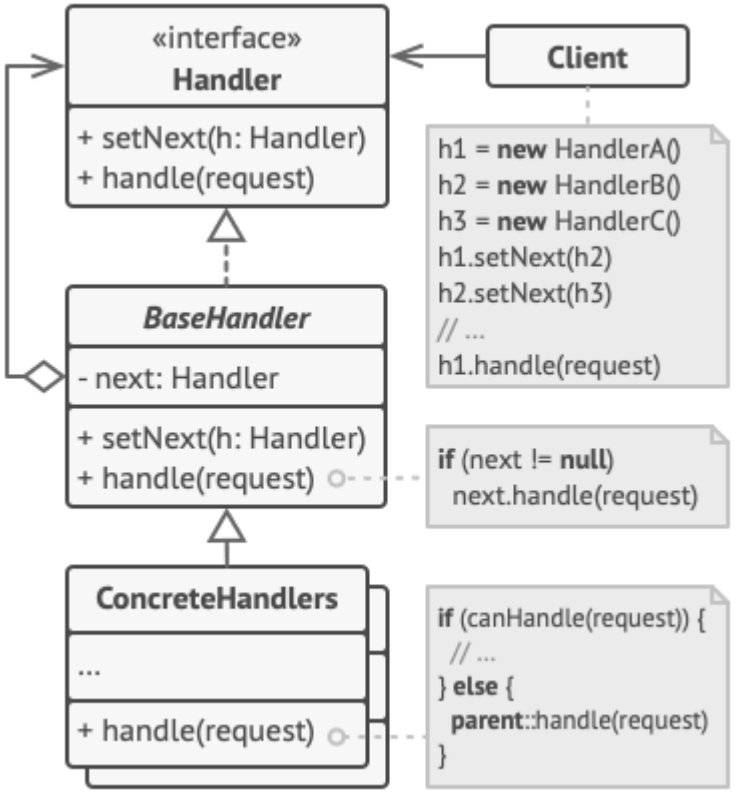


Image credit: https://refactoring.guru/design-patterns/chain-of-responsibility

Behavioral patterns: Command

Make a request into an object so that its specific behavior can be parametrized at runtime and it can support functionality related to its primary functionality (logging, un-doing, etc.)

Consider, buttons and menu items in a UI library

- We don't know what the specifics are of the request it will have to make (this depends on the application)
- Likely there will be some known functionality: OK Button, Cancel Button, Print Button, Send Button
- Potentially a lot of subclasses if we use inheritance (plus we don't know what exactly we need)
- Furthermore, similar functionality is shared across, e.g. buttons and menu items

Behavioral patterns: Command

We could parametrize the request and send the same parameters from different interface events (button, menu item, hotkey shortcut)

This is control coupling, because the UI needs too much information about business logic. (The meaning of parameter '2' shouldn't be something that the GUI needs to know.)

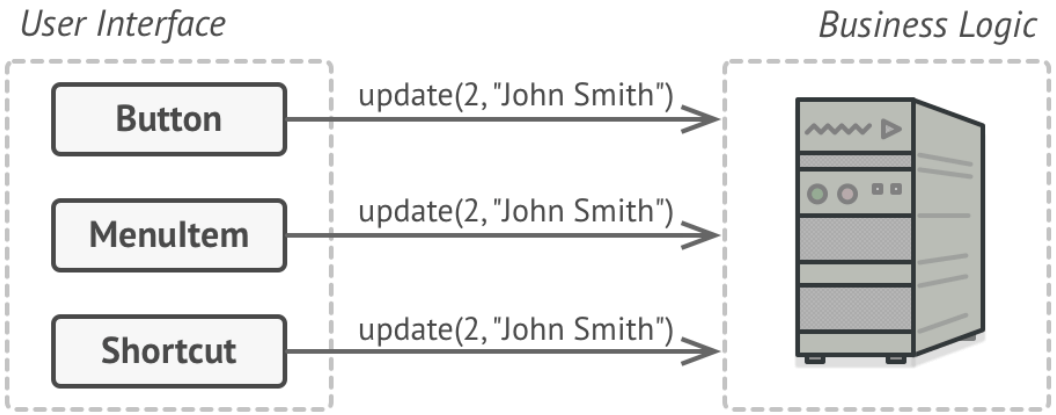


Image credit: https://refactoring.guru/design-patterns/command

Behavioral patterns: Command

- Command pattern suggests a solution: expose the Save command to the UI and have that object handle the communication with business logic.

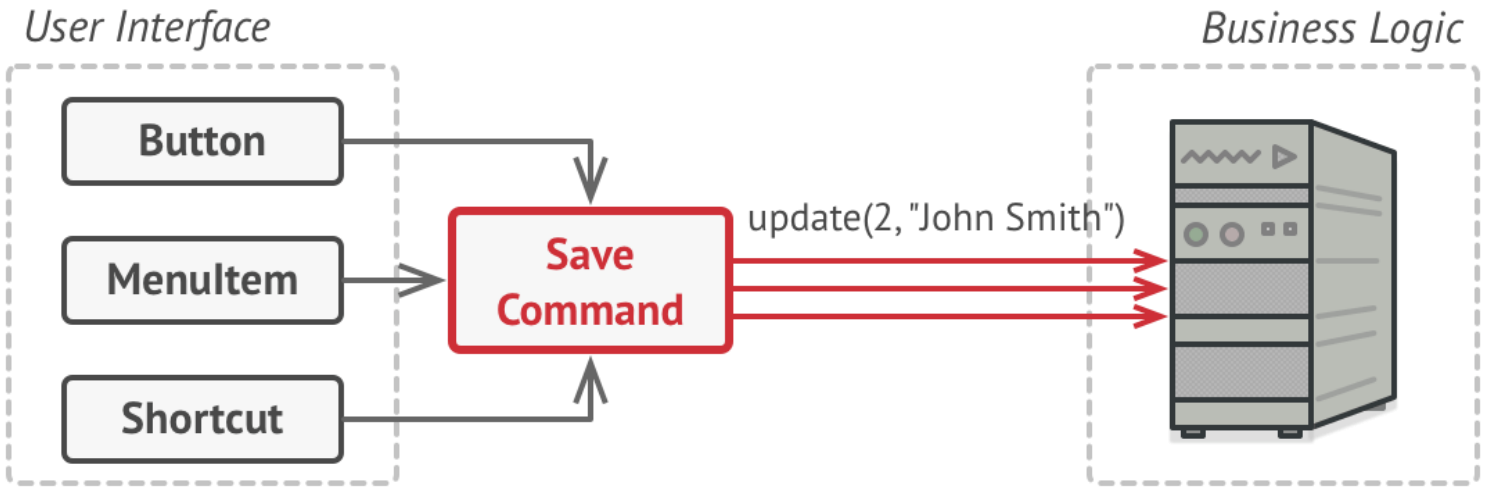
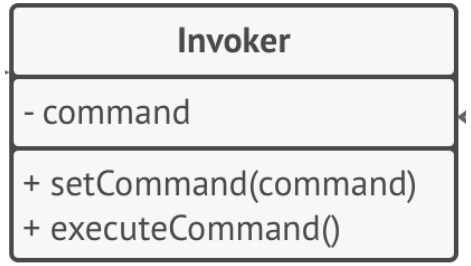


Image credit: https://refactoring.guru/design-patterns/command

Behavioral patterns: Command



Command structure:

- Invoker (sender) holds reference to a command
- Invokers request triggers command's execute method
- Command holds references to parameters and receiver (business logic) which it gets upon creation by the client
- Same Command can be invoked by multiple client components.

Regular Expressions

CS 5010 Fall 2020

Credits: materials from Maria Zontak's and Tony Mullen CS 5010 course for Northeastern University

Regular Expressions

Regular expressions (regexes, regexps) are a simple but powerful language for matching (finding, replacing) substrings. Found in (among others):

- Web
  - Google search
- Unix/Linux
  - Commmand line utilities: grep, less, sed, awk
  - Editors: Emacs, ed, vi
- Scripting/Programming
  - Perl, Python, Matlab, tcl, PHP, JS
  - C++ libraries, .NET, Java, C#

Regular Expressions

Regular expressions (regexes, regexps) are a simple but powerful language for matching (finding, replacing) substrings. Found in (among others):

- Editors/IDEs
  - Atom, VSC, Sublime, etc...
  - IntelliJ, Eclipse...
  - VSCode

Regular Expressions

What are they?

- A pattern of characters used to match strings in a search

How do they work?

- Metacharacters have special meaning
  - . * ? +
  - Special characters using \, e.g. \s (whitespace), \w (alphanumeric)
  - \ escapes the special meaning of the character that follows \
- Non-metacharacters match themselves:
  - A-Z a-z 0-9

Metacharacters

.	Any character, except the new line
[a-z]	Any one (single) character in the set (lowercase letters)
[^a-z]	Any one (single) character not in the set
*	Zero or more of the preceding character
+	One or more of the preceding character
| or (|)	or

Metacharacters - Examples

(Mr|Mrs) X	"Mr X" or "Mrs X"
[0-9A-Fa-f]	Hex digit
0x[0-9A-Fa-f]+	Hex number
\[[^\]]*\]	Text in square brackets

Anchors

^	Beginning of line
$	End of line
\b	word boundary

Examples

^[\s]*$	Blank line
\bA[A-Za-z]*\b	Word that starts with capital 'A'
^//.*	Comment line (in C/Java)
[0-9]+\.[0-9]*0	Float number with at least one trailing zero

Capturing & greediness

Pattern matches surrounded by parentheses are stored in special $ variables.

```
String str = "ABCDE12345abcdefgLMNOP";
String outstring = str.replaceAll(
   ".*([0-9]+)([a-z]+).*", "Matched $1 first, then $2"
);
```

What will outstring be?

```
Matched 5 first, then abcdefg
```

Is this what you expected?

Capturing & greediness

matching is *greedy* by default. It matches as much as it can. The opposite is *lazy* matching, which matches as little as it can. Lazy matching uses `*?`

```
String str = "ABCDE12345abcdefgLMNOP";
String outstring = str.replaceAll(
   ".*?([0-9]+)([a-z]+).*", "Matched $1 first, then $2"
);
```

This time, outstring is:

```
Matched 12345 first, then abcdefg
```

Try it

- Go to https://regexr.com/4a0jg. In the body of the sample text, you'll find the following strings representing legitimate and illegitimate emails.

Legitimate email strings	Not an email string
Alice.J.Smith+spamCatcher@gmail.com	@steve
doug@speedypizza.com	www.amazon.com
alice1998@supernet.co.uk	
J_smith@betterworld.org	

A) Write a regex that matches all of the strings in the left column in the text and lists them in the "list" window at the bottom of the screen.

B) Modify the reg ex and use capture variables (e.g. $1, $2 represent the captured content of the first two capture groups) to replace the emails in the text with identical emails enclosed by angle brackets, with any "plus" portion stripped out.

Expression JavaScript ▾ Flags ▾

/[redacted]/g

Text **4 matches** (0.0ms)

Many people use email. Gmail enables you to put a + symbol into your address to filter out spam or help to organize incoming mail. Something like this: Alice.J.Smith+spamCatcher@gmail.com. Plenty of other kinds of domains for email exist as well. Here are a few: doug@speedypizza.com, alice1998@supernet.co.uk, which has a country code top-level domain, and j_smith@betterworld.org. A pattern like @steve may be a Twitter handle. A URL like www.amazon.com is also not an email address.

In the Replace area, all the emails above have been placed into angle brackets.

Tools Replace List Details Explain

<[redacted]>

Many people use email. Gmail enables you to put a + symbol into your address to filter out spam or help to organize incoming mail. Something like this: <Alice.J.Smith@gmail.com>. Plenty of other kinds of domains for email exist as well. Here are a few: <doug@speedypizza.com>, <alice1998@supernet.co.uk>, which has a country code top-level domain, and <j_smith@betterworld.org>. A pattern like @steve may be a Twitter handle. A URL like www.amazon.com is also not an email address.

In the Replace area, all the emails above have been placed into angle brackets.

Expression JavaScript ▾ Flags ▾

```
/((\w*\.*)*\w+)(\+\w*)?(\@\w*(\.\w+)+)/g
```

Text **4 matches** (0.0ms)

Many people use email. Gmail enables you to put a + symbol into your address to filter out spam or help to organize incoming mail. Something like this: Alice.J.Smith+spamCatcher@gmail.com. Plenty of other kinds of domains for email exist as well. Here are a few: doug@speedypizza.com, alice1998@supernet.co.uk, which has a country code top-level domain, and j_smith@betterworld.org. A pattern like @steve may be a Twitter handle. A URL like www.amazon.com is also not an email address.

In the Replace area, all the emails above have been placed into angle brackets.

Tools Replace | List | Details | Explain

```
<$1$4>
```

Many people use email. Gmail enables you to put a + symbol into your address to filter out spam or help to organize incoming mail. Something like this: <Alice.J.Smith@gmail.com>. Plenty of other kinds of domains for email exist as well. Here are a few: <doug@speedypizza.com>, <alice1998@supernet.co.uk>, which has a country code top-level domain, and <j_smith@betterworld.org>. A pattern like @steve may be a Twitter handle. A URL like www.amazon.com is also not an email address.

In the Replace area, all the emails above have been placed into angle brackets.

More Patterns

Behavioral patterns: Mediator

- Promotes loose coupling by keeping objects from referring to each other explicitly
- Behaves as a hub managing interactions of multiple diverse components
- Abstracts over potentially complex interactions
- As software grows via maintenance, the number objects that interact with each other may increase.
  - Increased coupling as time goes on
  - Mediator becomes the interface to interact with those objects.

Behavioral patterns: Mediator

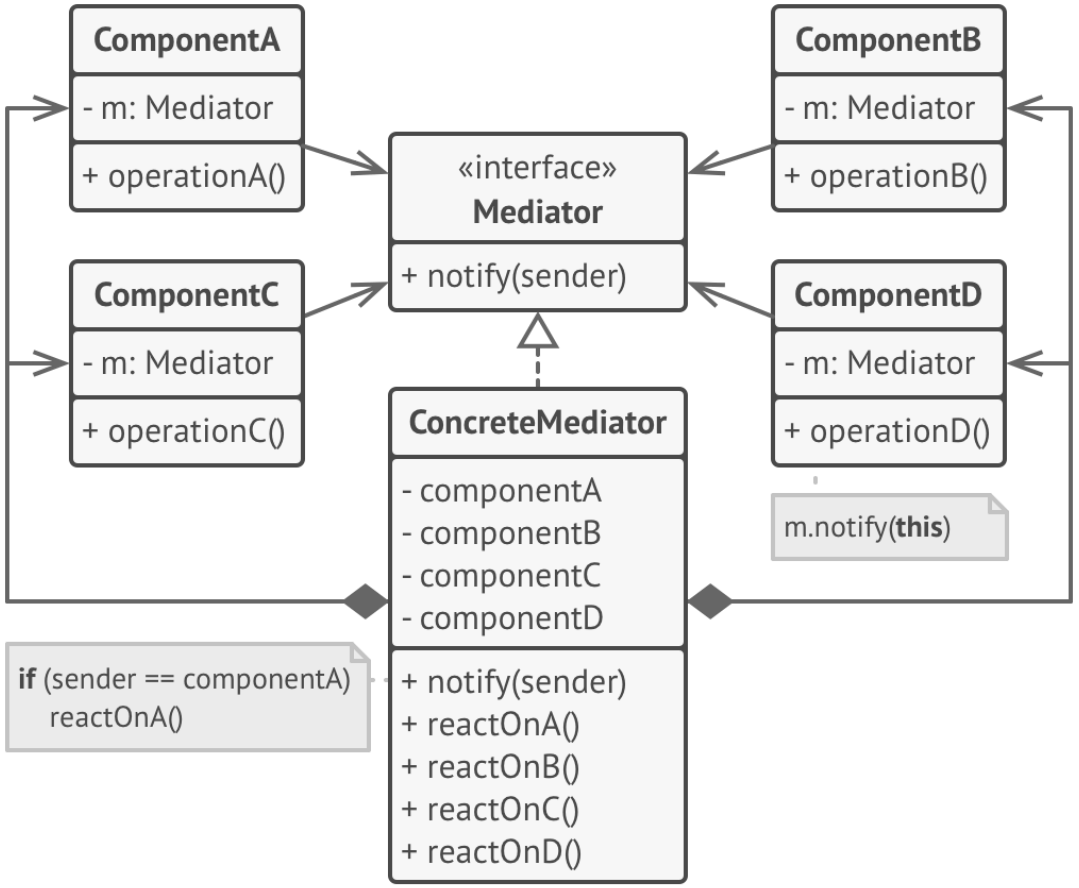


Image credit: https://refactoring.guru/design-patterns/mediator

Behavioral patterns: Memento

Sometimes we want to keep "snapshots" of the state (objects) of our system.

- Enable un-do or operator history
- Retrievability or portability

How to snapshot private state?

- Saving/storage mechanism needs access
- Recreation mechanism needs access

Poses problems for encapsulation

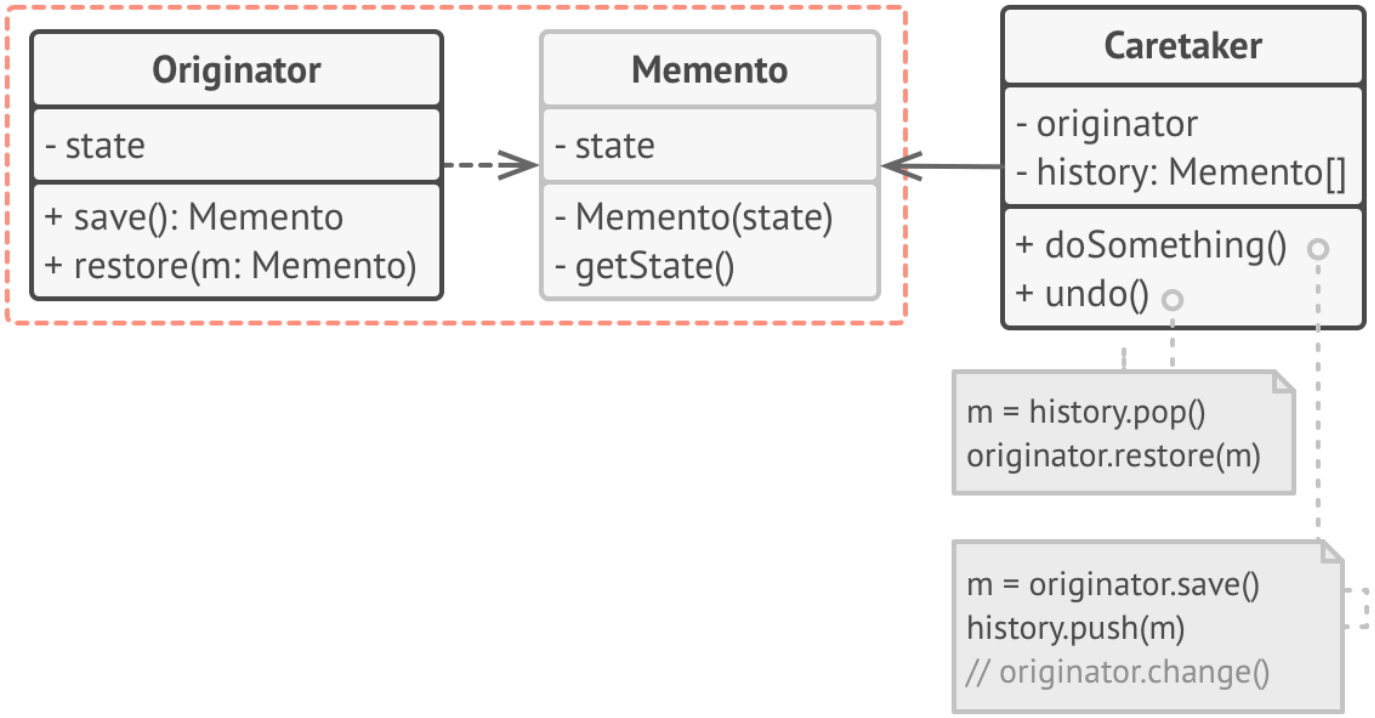
Behavioral patterns: Memento

Let the objects that need to be stored handle their own storage and retrieval logic by providing a *memento*.

Only the object itself needs access to its private data.

Subsequent changes to implementation of the object and backwards compatibility adjustments for previously created mementos all remain in the original class.

Behavioral patterns: Memento

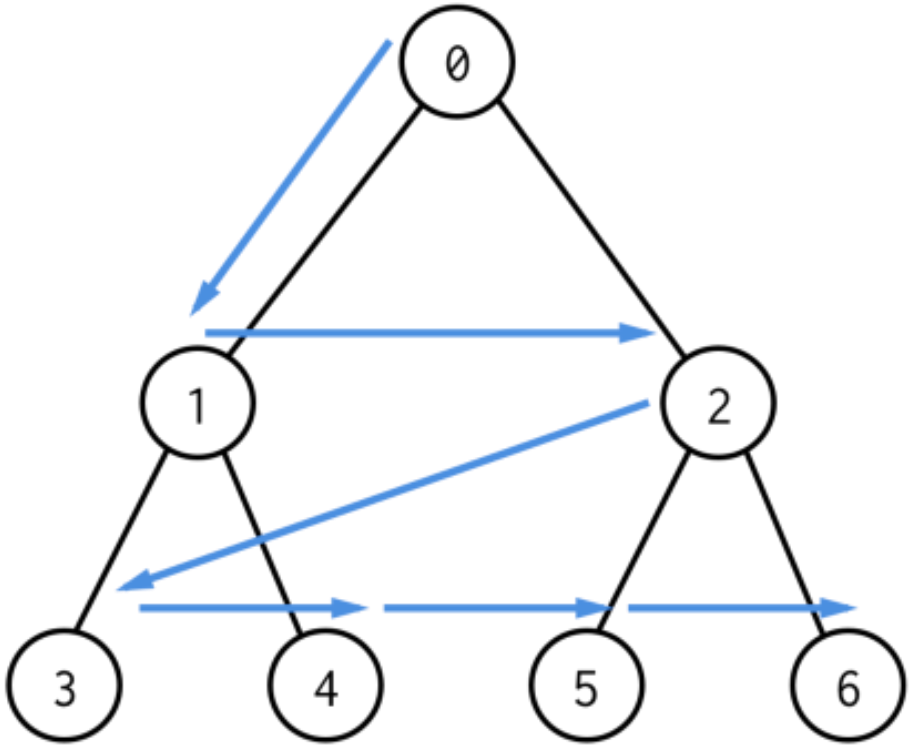


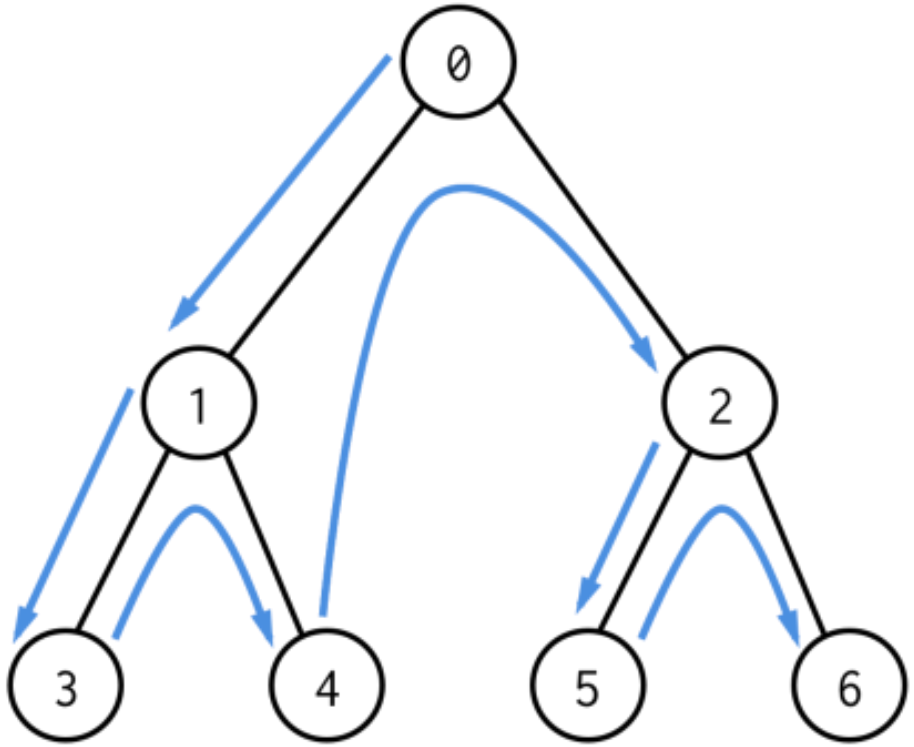
Memento structure:

- May be implemented as a nested class (in Java)
- Originator class both creates Memento object and restores it
- Caretaker can maintain mementos but does not directly access contents
- Memento class exposes necessary data/metadata publicly

Behavioral pattern: Iterator

- We frequently make use of collections or aggregate objects
- An iterator object handles sequential access
- Different iteration algorithms can be handled by different iterators for the same collection



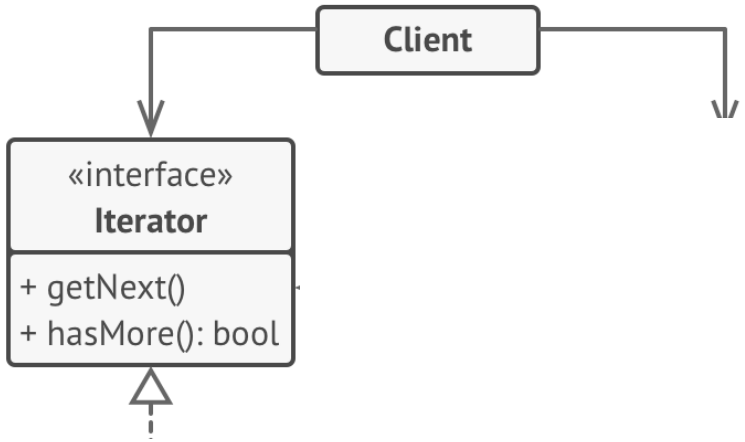


Behavioral pattern: Iterator

- Consider a class with an underlying tree structure
- Using the iterator pattern, we can offer both breadth first or depth first traversal without exposing structure directly

Image credit: http://mishadoff.com/blog/dfs-on-binary-tree-array/

Behavioral patterns: Iterator



Iterator structure:

- Iterator interface declares methods necessary to sequentially access collection elements
- IteratableCollection interface specifies that a collection can create its own iterator
- Specific traversal algorithm defined in concrete iterator

Behavioral pattern: Visitor

- Sometimes polymorphism isn't the answer.
- Sometimes we need unrelated, varied objects to provide consistent and appropriate behavior for a specific purpose.
- The behavior may not be appropriate or sensible to include in those classes definitions or interfaces.
- Good to use when the class structure rarely changes, but you often want to define new operations over that structure.
- Visitor pattern enables us to "visit" objects with personalized functionality as needed.

Behavioral pattern: Visitor

- Consider an existing system made up of various objects and relationships.
- You have been tasked with creating an XML documentation export service for the system.
- You'd *like* to have each object in the system provide a `toXML()` method, but they don't.
- Not ideal to be modifying existing production code to write an exporter. Plus, it's likely you'll need another format (JSON?) soon anyway, and you'll need to do it again.
- Export functionality is not really appropriate behavior for these objects. It's specific to your purposes.

Behavioral pattern: Visitor

Visitor structure:

- Element classes accept a visitor
- A concrete visitor works directly with a specific concrete class (visit method overloaded)
- Accept method can be called recursively, and/or inherited, passing the visitor structure

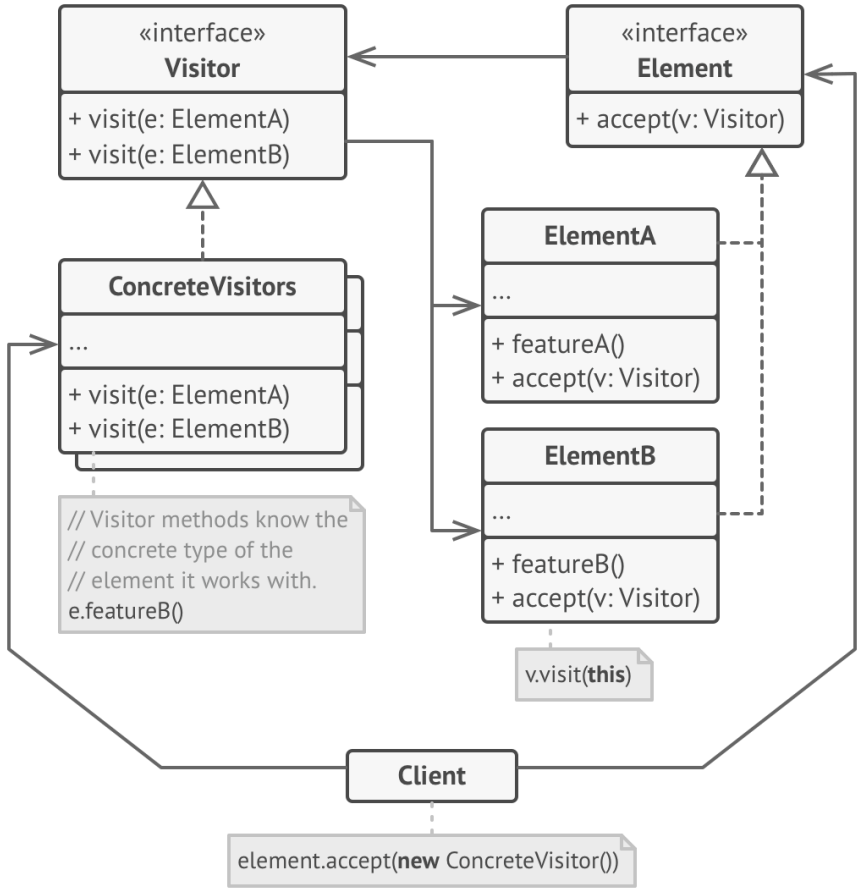


Image credit: https://refactoring.guru/design-patterns/visitor

Visitor Example: Interfaces

```
interface CarElement {
    void accept(CarElementVisitor visitor);
}

interface CarElementVisitor {
    void visit(Body body);
    void visit(Car car);
    void visit(Engine engine);
    void visit(Wheel wheel);
}
```

Visitor Example: Elements

```
class Wheel implements CarElement {
  private final String name;

  public Wheel(final String name) {
      this.name = name;
  }

  public String getName() {
      return name;
  }

  @Override
  public void accept(CarElementVisitor visitor) {
      visitor.visit(this);
  }
}
```

```
class Body implements CarElement {
  @Override
  public void accept(CarElementVisitor visitor) {
      visitor.visit(this);
  }
}

class Engine implements CarElement {
  @Override
  public void accept(CarElementVisitor visitor) {
      visitor.visit(this);
  }
}
```

Visitor Example: Elements

```
class Car implements CarElement {
    private final List<CarElement> elements;

    public Car() {
        this.elements = List.of(
            new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left"), new Wheel("back right"),
            new Body(), new Engine()
        );
    }

    @Override
    public void accept(CarElementVisitor visitor) {
        for (CarElement element : elements) {
            element.accept(visitor);
        }
        visitor.visit(this);
    }
}
```

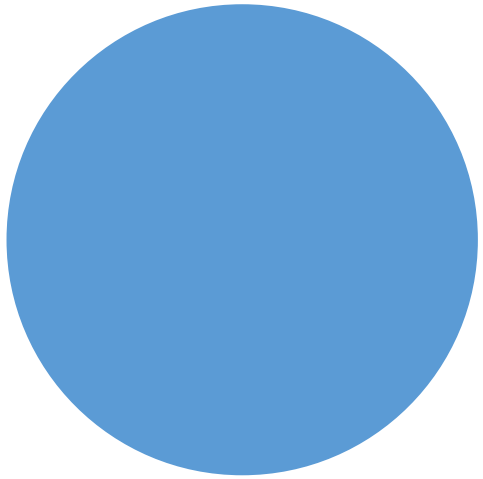
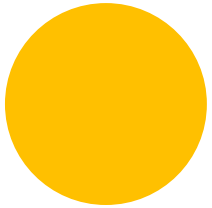
Visitor Example: Visitors

```
class CarElementDoVisitor implements CarElementVisitor {
    @Override
    public void visit(Body body) {
        System.out.println("Moving my body");
    }

    @Override
    public void visit(Car car) {
        System.out.println("Starting my car");
    }

    @Override
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + " wheel");
    }

    @Override
    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }
}
```

```
class CarElementPrintVisitor implements CarElementVisitor {
    @Override
    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    @Override
    public void visit(Car car) {
        System.out.println("Visiting car");
    }

    @Override
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }

    @Override
    public void visit(Wheel wheel) {
        System.out.println("Visiting " + wheel.getName() + " wheel");
    }
}
```

Visitor Example: EntryPoint (main)

```
public class VisitorDemo {
    public static void main(final String[] args) {
        Car car = new Car();

        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

Questions?