CS 5004: Lecture 12

Northeastern University, Spring 2021

Credits: thanks to Tamara Bonaci, Tony Mullen & past NEU instructors

At the start of every lecture

1. Pull the latest code from the lecture-code repo
2. Open the Evening_lectures folder
3. Copy this week's folder somewhere else
   - So you can edit it without causing GitHub conflicts
4. Open the code:
   1. Find the build.gradle file in the folder called LectureX
   2. Double click it to open the project

Agenda

Design patterns

- Creational
- Structural
- Behavioral

Design patterns intro

What are design patterns?

- Tried and tested approaches to common problems in software development
- *Can* help to solve inevitable tradeoffs that come with design decisions
- Not for system architecture (like MVC)

  → Problems that occur at specific places in a system e.g. creating a complex object

Common problems, familiar solutions #1

Problem: public fields can be directly manipulated by clients → may break key aspects of an object's state or behavior

Common problems, familiar solutions #1

Problem: public fields can be directly manipulated by clients → may break key aspects of an object's state or behavior

Solution: access modifiers allow us to restrict how client code interacts with our objects (information hiding)

Common problems, familiar solutions #1

Problem: public fields can be directly manipulated by clients → may break key aspects of an object's state or behavior

Solution: access modifiers allow us to restrict how client code interacts with out objects (information hiding)

Tradeoff: may not correctly anticipate client needs for public functionality

Common problems, familiar solutions #2

Problem: repetition in related implementations

→similar abstractions have similar fields, methods

Common problems, familiar solutions #2

Problem: repetition in related implementations

→similar abstractions have similar fields, methods

Solution: inherit shared fields method from a superclass, dynamic dispatch (inheritance)

Common problems, familiar solutions #2

Problem: repetition in related implementations

→similar abstractions have similar fields, methods

Solution: inherit shared fields method from a superclass, dynamic dispatch (inheritance)

Tradeoffs:

- Code for a class is spread out, potentially less understandable
- Run-time dispatch introduces overhead

Common problems, familiar solutions #3

Problem: To access all members of a collection, client must perform a specialized traversal for each data structure

Common problems, familiar solutions #3

Problem: To access all members of a collection, client must perform a specialized traversal for each data structure

Solution: The *implementation* performs the traversals

→ Results are communicated via a standard interface (`Iterator`'s `hasNext()` and `next()`)

Common problems, familiar solutions #3

Problem: To access all members of a collection, client must perform a specialized traversal for each data structure

Solution: The *implementation* performs the traversals

→Results are communicated via a standard interface (`Iterator`'s `hasNext()` and `next()`)

Tradeoff: Iteration order fixed by the implementation, not under the control of the client.

Common problems, familiar solutions #4

Problem: Errors happening in one part of the code should be handled elsewhere

- Return values should not be preempted by error codes.

Common problems, familiar solutions #4

Problem: Errors happening in one part of the code should be handled elsewhere

- Return values should not be preempted by error codes.

Solution: Standard structures for throwing and catching exceptions

Common problems, familiar solutions #4

Problem: Errors happening in one part of the code should be handled elsewhere

- Return values should not be preempted by error codes.

Solution: Standard structures for throwing and catching exceptions

Tradeoffs:

- Code may still be cluttered
- Important exceptions may be handled silently

Common problems, familiar solutions #5

Problem: Data structures (Lists, Sets etc) hold only one type of object.

Common problems, familiar solutions #5

Problem: Data structures (Lists, Sets etc) hold only one type of object.

Solution: Generic types allows flexibility and type checking.

Common problems, familiar solutions #5

Problem: Data structures (Lists, Sets etc) hold only one type of object.

Solution: Generic types allows flexibility and type checking.

Tradeoffs:

- More verbose code.
- Limitations to the flexibility

Why more design patterns?

Because there are more problems ☺

- Not enough to know everything in a language
- Common design problems still occur

Why more design patterns?

Because there are more problems ☺

- Not enough to know everything in a language
- Common design problems still occur

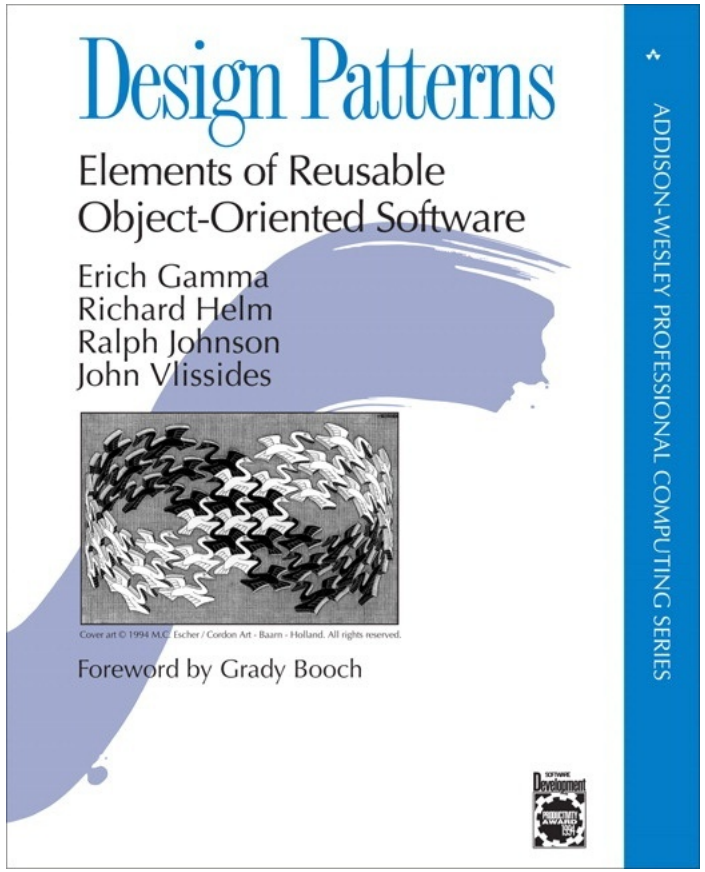
Design patterns capture common, successful solutions for high-level design problems

- Slightly different from the previous examples

History of design patterns

Classic textbook from 1994

- Authors known as “Gang of Four”
- Describes 23 design patterns in 3 categories:
  - **Creational** – for creating objects
  - **Structural** – for object composition
  - **Behavioral** – for communication between objects
- Examples in C++ and SmallTalk



A preview of some of the patterns

There should really only ever be one instance of some class

e.g. GameState in the tic-tac-toe example

```
public static void main(String[] args) {
    GameState state = new GameState();
    Player x = new Player( character: "X", state);
    Player o = new Player( character: "O", state);
    Scanner scanner = new Scanner(System.in);
```

What if client doesn't realize players should have a reference to the *same* `GameState`

GoF solution: Singleton pattern (creational)

A preview of some of the patterns

Lots of concrete implementations of a parent class, client doesn't really need to know about construction

e.g. the Cookies in last week's revised cookiecounter

```
switch (args[i]) {
    case SAMOA:
        this.items.put(new SamoaCookie(), numCookies);
        break;
    case DOSIDOS:
        this.items.put(new DosidoCookie(), numCookies);
        break;
    case THIN_MINTS:
        this.items.put(new ThinMintCookie(), numCookies);
        break;
    default:
        throw new InvalidArgumentsException("Unknown cookie: " + args[i]);
}
```

GoF solution: Factory pattern (creational)

A preview of some of the patterns

There are too many optional parameters in an class's constructor

e.g. the Triangle from lecture 5

```
public Triangle(Point2D reference, int sideA, int sideB, int sideC, int angleA, int angleB, int angleC) {
    super(reference);
    this.sideA = sideA;
    this.sideB = sideB;
    this.sideC = sideC;
    this.angleA = angleA;
    this.angleB = angleB;
    this.angleC = angleC;
}
```

GoF solution: Builder pattern (creational)

The GoF design patterns

Creational

Builder

Prototype

Factory method

Abstract Factory

Singleton

Structural

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

Behavioral

Chain of responsibility

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template method

Visitor

The GoF design patterns

Creational

Builder

Prototype

Factory method

Abstract Factory

Singleton

Structural

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

Behavioral

Chain of responsibility

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template method

Visitor

A tip for today's lecture

We're going through A LOT of examples → give yourself time to digest!

- Don't expect to walk away from class "knowing" design patterns
- Revisit the code on your own after class

Creational design patterns

Ways to create or control the creation of objects

Builder pattern

Builder pattern

A class that encapsulates a complex object creation process

- complex = lots of fields, which may be optional / have defaults

> From GoF:
>
> Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Builder pattern

A class that encapsulates a complex object creation process

- complex = lots of fields, which may be optional / have defaults

When to use:

- You have a complex object ☺
- Constructor overloading is too complex, or you just have a lot of potential constructor variations

Example use case

There are too many optional parameters in a class's constructor

e.g. the Triangle from lecture 5

```
public Triangle(Point2D reference, int sideA, int sideB, int sideC, int angleA, int angleB, int angleC) {
    super(reference);
    this.sideA = sideA;
    this.sideB = sideB;
    this.sideC = sideC;
    this.angleA = angleA;
    this.angleB = angleB;
    this.angleC = angleC;
}
```

Builder pattern: Java implementation

```
public class ComplexObject {
  // instance variables
  public ComplexObject(Integer field1, ..., Integer field10) {
    this.field1 = field1;
    ...
    this.field10 = field10;
  }
}
```

Builder pattern: Java implementation

```
public class ComplexObject {
  // instance variables
  public ComplexObject(Integer field1, ..., Integer field10) {
    this.field1 = field1;
    ...
    this.field10 = field10;
  }
}
```

Goal: replace with something more readable & better suited for flexible construction

Builder pattern: Java implementation

```
public class ComplexObject {
  // instance variables
  private ComplexObject(Builder builder) {
    this.field1 = builder.field1;
    ...
    this.field10 = builder.field10;
  }
  public static class Builder {
    // Does the building
  }
}
```

Builder pattern: Java implementation

```
public class ComplexObject {
  // instance variables
  private ComplexObject(Builder builder) {
    this.field1 = builder.field1;
    ...
    this.field10 = builder.field10;
  }
  public static class Builder {
    // Does the building
  }
}
```

Public nested “builder” class

- Will have public methods to handle parent construction

Builder pattern: Java implementation

```java
public class ComplexObject {
  // instance variables
  private ComplexObject(Builder builder) {
    this.field1 = builder.field1;
    ...
    this.field10 = builder.field10;
  }
  public static class Builder {
    // Does the building
  }
}
```

Public nested “builder” class

- Will have public methods to handle parent construction
- Replaces parent constructor parameters

Builder pattern: Java implementation

```
public class ComplexObject {
  // instance variables
  private ComplexObject(Builder builder) {
    this.field1 = builder.field1;
    ...
    this.field10 = builder.field10;
  }
  public static class Builder {
    // Does the building
  }
}
```

Parent constructor should be private

- Client code has to go through the builder

Builder pattern: Builder class implementation

```
public static class Builder {
  // Same instance variables as parent
  public Builder(Integer field1) {
    this.field1 = field1;
  }
  public Builder addField2(Integer field2) {
    this.field2 = field2;
    return this;
  }
  public ComplexObject build() {
    return new ComplexObject(this);
  }
}
```

Builder pattern: Builder class implementation

```
public static class Builder {
  // Same instance variables as parent
  public Builder(Integer field1) {
    this.field1 = field1;
  }
  public Builder addField2(Integer field2) {
    this.field2 = field2;
    return this;
  }
  public ComplexObject build() {
    return new ComplexObject(this);
  }
}
```

All instance variables that would have been passed to parent constructor should be in the builder class

- ...as well as the parent
- Keep private

Builder pattern: Builder class implementation

```
public static class Builder {
  // Same instance variables as parent
  public Builder(Integer field1) {
    this.field1 = field1;
  }
  public Builder addField2(Integer field2) {
    this.field2 = field2;
    return this;
  }
  public ComplexObject build() {
    return new ComplexObject(this);
  }
}
```

Include only required parameters

- Set default values as needed
- Do not include optional parameters

Builder pattern: Builder class implementation

```
public static class Builder {
  // Same instance variables as parent
  public Builder(Integer field1) {
    this.field1 = field1;
  }
  public Builder addField2(Integer field2) {
    this.field2 = field2;
    return this;
  }
  public ComplexObject build() {
    return new ComplexObject(this);
  }
}
```

Separate methods to set optional parameters

- Return type = builder class
- **`return this`**
  - The builder in it's current state, with all fields populated so far
  - Allows methods to be called in any order

Builder pattern: Builder class implementation

```
public static class Builder {
  // Same instance variables as parent
  public Builder(Integer field1) {
    this.field1 = field1;
  }
  public Builder addField2(Integer field2) {
    this.field2 = field2;
    return this;
  }
  public ComplexObject build() {
    return new ComplexObject(this);
  }
}
```

The final step in the build process

- Calls the parent's private constructor
- Passes the "complete" builder instance

Using a builder: the client perspective

```
ComplexObject mostBasic
     = new ComplexObject.Builder(1).build();

ComplexObject aFewParams
     = new ComplexObject.Builder(1)
                                .addField2(1000)
                                .addField10(-5)
                                .build();
```

Using a builder: the client perspective

```
ComplexObject mostBasic
     = new ComplexObject.Builder(1).build();

ComplexObject aFewParams
     = new ComplexObject.Builder(1)
                         .addField2(1000)
                         .addField10(-5)
                         .build();
```

“method chaining”

Using a builder: the client perspective

```
ComplexObject mostBasic
     = new ComplexObject.Builder(1).build();

ComplexObject aFewParams
     = new ComplexObject.Builder(1)
                         .addField2(1000)
                         .addField10(-5)
                         .build();
```

Must start with Builder constructor....

Using a builder: the client perspective

```
ComplexObject mostBasic
     = new ComplexObject.Builder(1).build();

ComplexObject aFewParams
     = new ComplexObject.Builder(1)
                         .addField2(1000)
                         .addField10(-5)
                         .build();
```

...and end with `build()`

Using a builder: the client perspective

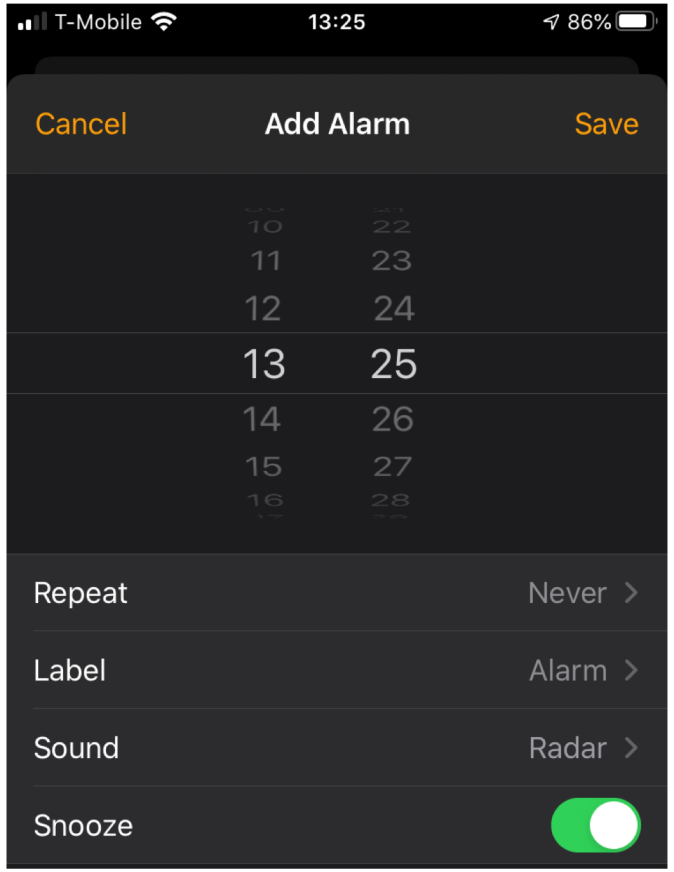
```
ComplexObject mostBasic
     = new ComplexObject.Builder(1).build();

ComplexObject aFewParams
     = new ComplexObject.Builder(1)
                    .addField2(1000)
                    .addField10(-5)
                    .build();
```

Chain other methods in between as needed

Walkthrough: Alarm example

Lecture12 > alarm package

- BasicAlarm – standard, no builder
- Alarm – uses a builder
- Main – alarm construction for several scenarios



Factories

Creating objects without (client) calling constructors

Factory motivation: changing implementations

Supertypes support multiple implementations e.g.

```
class Cookie extends AbstractFoodItem {...}
class SamoaCookie extends Cookie {...}
class DosidoCookie extends Cookie {...}
```

Factory motivation: changing implementations

Supertypes support multiple implementations e.g.

```
class Cookie extends AbstractFoodItem {...}
class SamoaCookie extends Cookie {...}
class DosidoCookie extends Cookie {...}
```

Clients use the supertype (Cookie):

```
Map<Cookie, Integer> cookieStash = new HashMap<>();
```

Factory motivation: changing implementations

Supertypes support multiple implementations e.g.

```
class Cookie extends AbstractFoodItem {...}
class SamoaCookie extends Cookie {...}
class DosidoCookie extends Cookie {...}
```

Clients use the supertype (Cookie):

```
Map<Cookie, Integer> cookieStash = new HashMap<>();
```

Still need to use subtype constructors:

```
cookieStash.put(new SamoaCookie(), 10);
```

Factory motivation: changing implementations

Supertypes support multiple implementations e.g.

```
class Cookie extends AbstractFoodItem {...}
class SamoaCookie extends Cookie {...}
class DosidoCookie extends Cookie {...}
```

Clients use the supertype (Cookie):

```
Map<Cookie, Integer> cookieStash = new HashMap<>();
```

Still need to use subtype constructors:

```
cookieStash.put(new SamoaCookie(), 10);
```

Factories put the constructor behind an abstraction

Factory method

Client code calls a method to get an instance of a class (instead of a constructor).

```
class CookieFactory {
  public static Cookie bestCookie() {
    return new SamoaCookie();
  }
}
```

Advantages:

- To switch implementation, change only one place
- Method can do computations to decide which subclass to return

DateFormat Factory Methods

DateFormat class encapsulates how to format dates & times as text

- Just date? Just time? Date + time? Where in the world?
- Instead of passing all options to constructor, use factories
- Subtype created by factory call need not be specified

```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL,
                                            Locale.FRANCE);

Date today = new Date();
df1.format(today); // "Jan 1, 2020"
df2.format(today); // "10:15:00 AM"
df3.format(today); // "mercredi 1 janvier 2020"
```

Abstract Factory pattern

Similar idea to the method but with an additional abstraction layer:

- Factory class
- Use a switch/if statement to create appropriate instances of some supertype based on some input
  - switch is acceptable if:
    - only appears once
    - used to create polymorphic objects
    - hidden behind an inheritance relationship so the rest of the system doesn't see it

Abstract factory example

```
class EarbudFactory {
  Earbuds chooseEarbuds(int usageHours, Jack connector) {
    if (usageHours < BluetoothEarbuds.BATTERY_LIFE)
      return new BluetoothEarbuds();
    else if (connector == Jack.LIGHTNING)
      return new WiredEarbudsWithDongle();
    else return new WiredEarbuds();
  }
}
```

Walkthrough: CookieFactory

lecture12 > cookiefactory package:

- CookieFactory class → returns Cookie subclass based on argument
- Used in CookieStash → doesn't need to know anything about the concrete classes

Singleton pattern

Only allow a single instance of a class to exist

Possible reasons for a singleton

- Whenever there should be only one instance of an object
  - e.g. representations of some global state
  - e.g. GameState in the tic-tac-toe example

```
public static void main(String[] args) {
    GameState state = new GameState();
    Player x = new Player( character: "X", state);
    Player o = new Player( character: "O", state);
    Scanner scanner = new Scanner(System.in);
```

- Players should have a reference to the *same* `GameState`
- Doesn't make sense to allow more than one GameState

This pattern has critics! Mostly due to overuse (global data generally considered bad).

Singleton implementation: option 1

```
public class OnlyOne {
  private static final OnlyOne instance = new OnlyOne();

  private OnlyOne() {...}

  public static OnlyOne getInstance() {
    return instance;
  }
}
```

Singleton implementation: option 1

```
public class OnlyOne {
  private static final OnlyOne instance = new OnlyOne();

  private OnlyOne() {...}

  public static OnlyOne getInstance() {
    return instance;
  }
}
```

`static final`

- Once instance created, can't be re-created
- Instance created ahead of time (eager allocation)

Singleton implementation: option 1

```
public class OnlyOne {
  private static final OnlyOne instance = new OnlyOne();

  private OnlyOne() {...}

  public static OnlyOne getInstance() {
    return instance;
  }
}
```

private
Prevents instantiation outside class

Singleton implementation: option 1

```
public class OnlyOne {
  private static final OnlyOne instance = new OnlyOne();

  private OnlyOne() {...}

  public static OnlyOne getInstance() {
    return instance;
  }
}
```

Static method provides global access to instance

Singleton implementation: option 2

```
public class OnlyOne {
  private static OnlyOne instance;

  private OnlyOne() {...}

  public static OnlyOne getInstance() {
    if (instance == null)
      instance = new OnlyOne();
    return instance;
  }
}
```

Singleton implementation: option 2

```
public class OnlyOne {
  private static OnlyOne instance;

  private OnlyOne() {...}

  public static OnlyOne getInstance() {
    if (instance == null)
      instance = new OnlyOne();
    return instance;
  }
}
```

static (not final)

- Instance will be created on demand (lazy allocation)

Singleton implementation: option 2

```
public class OnlyOne {
  private static OnlyOne instance;

  private OnlyOne() {...}

  public static OnlyOne getInstance() {
    if (instance == null)
      instance = new OnlyOne();
    return instance;
  }
}
```

private
Prevents instantiation outside class

Singleton implementation: option 2

```
public class OnlyOne {
  private static OnlyOne instance;

  private OnlyOne() {...}

  public static OnlyOne getInstance() {
    if (instance == null)
      instance = new OnlyOne();
    return instance;
  }
}
```

Static method provides global access to instance

- Creates the instance if and only if it has not already been created

Walkthrough: Singleton tic-tac-toe

Lecture12 > tictactoe package

- model > GameState is now a singleton

Structural design patterns

Focus on object composition, relations between objects (e.g. inheritance), and relations between objects and the system as a whole

Structural patterns

A *wrapper* translates between "incompatible" interfaces:

- Modify the interface
- Extend behavior
- Restrict access

Some of these patterns line up with already familiar Java capabilities

Adapter pattern

Change an interface without changing functionality

“Change an interface without changing functionality”

“interface” != a Java Interface

- Remember, design patterns are not Java-specific
- “interface” = the public behavior a client will use

“without changing functionality”

- Without modifying existing classes: open-closed principle

Drew Breunig
@dbreunig

Follow

Apple's fastest growing product category.

How to make an adapter

- Subclassing → make a new class that inherits the original class and adds / changes functionality
- Delegation → use the existing class as a member of a new class that has the new interface

A familiar example: subclassing

```java
public class Point2D {
  private int x;
  private int y;
  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int x() {
    return this.x;
  }
  public int y() {
    return this.y;
  }
}
```

```java
public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y,
                 int z) {
    super(x, y);
    this.z = z;
  }
  public int z() {
    return this.z;
  }
}
```

Delegation version

```
public class Point3D {
  private Point2D xy;
  private int z;
  public Point3D(int x, int y, int z) {
    this.xy = new Point2D(x, y);
    this.z = z;
  }
  public int x() { return this.xy.x(); }
  public int y() { return this.xy.y(); }
  public int z() { return this.z; }
}
```

Adapter by subclassing vs. delegation

Subclassing

- automatically gives access to all methods of superclass
  - even those you might not want

Delegation

- permits removal of methods
  - don’t get polymorphic capability

Decorator pattern

Attach additional behavior or responsibilities to an object dynamically. An alternative to sub-classing for extending functionality.

Decorator pattern

- Add functionality without changing the interface
- Add to existing methods to do something additional
  - (while still preserving the previous specification)
- Subclassing can be an example of the decorator pattern
  - But it’s not the only option

Decorator example: notification system

Application has a notifier and uses it to send alerts on particular events

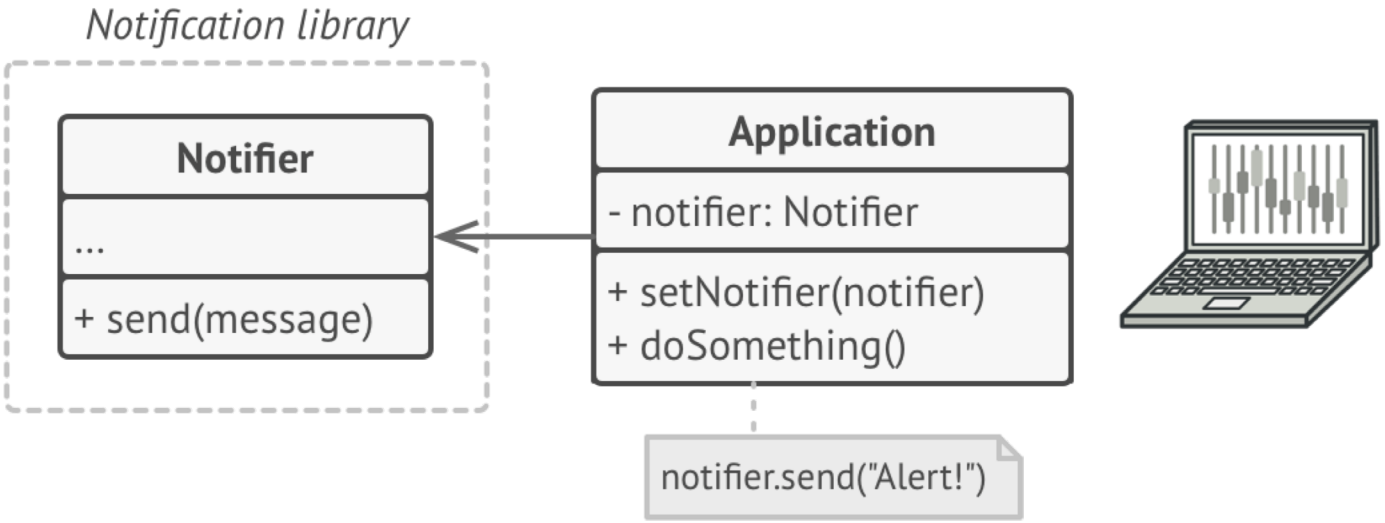
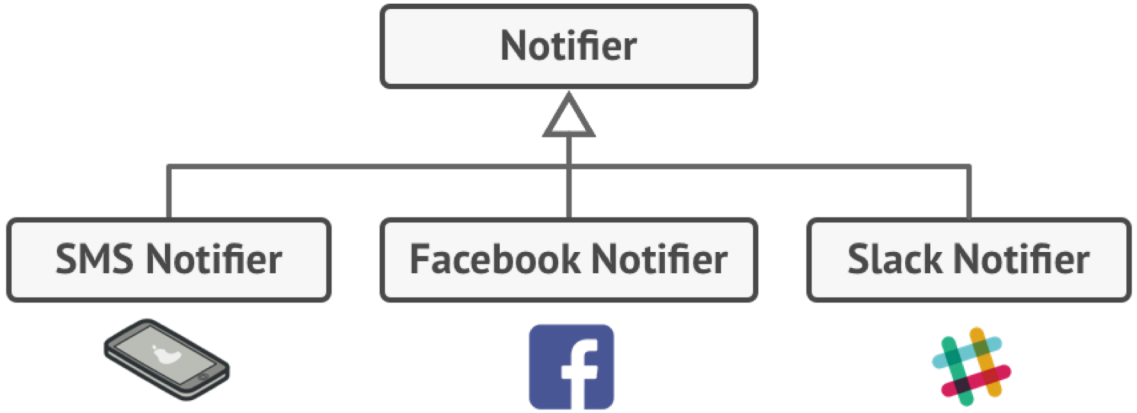


Image credit: https://refactoring.guru/design-patterns/decorator

Decorator example: notification system

System wants to support a variety of platforms for notifications



- Could use subclassing, but...
- What if we want to combine notifiers in arbitrary ways?
  - e.g. Facebook + Slack or SMS only

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

Decorator example: notification system

Relying on inheritance could get complicated

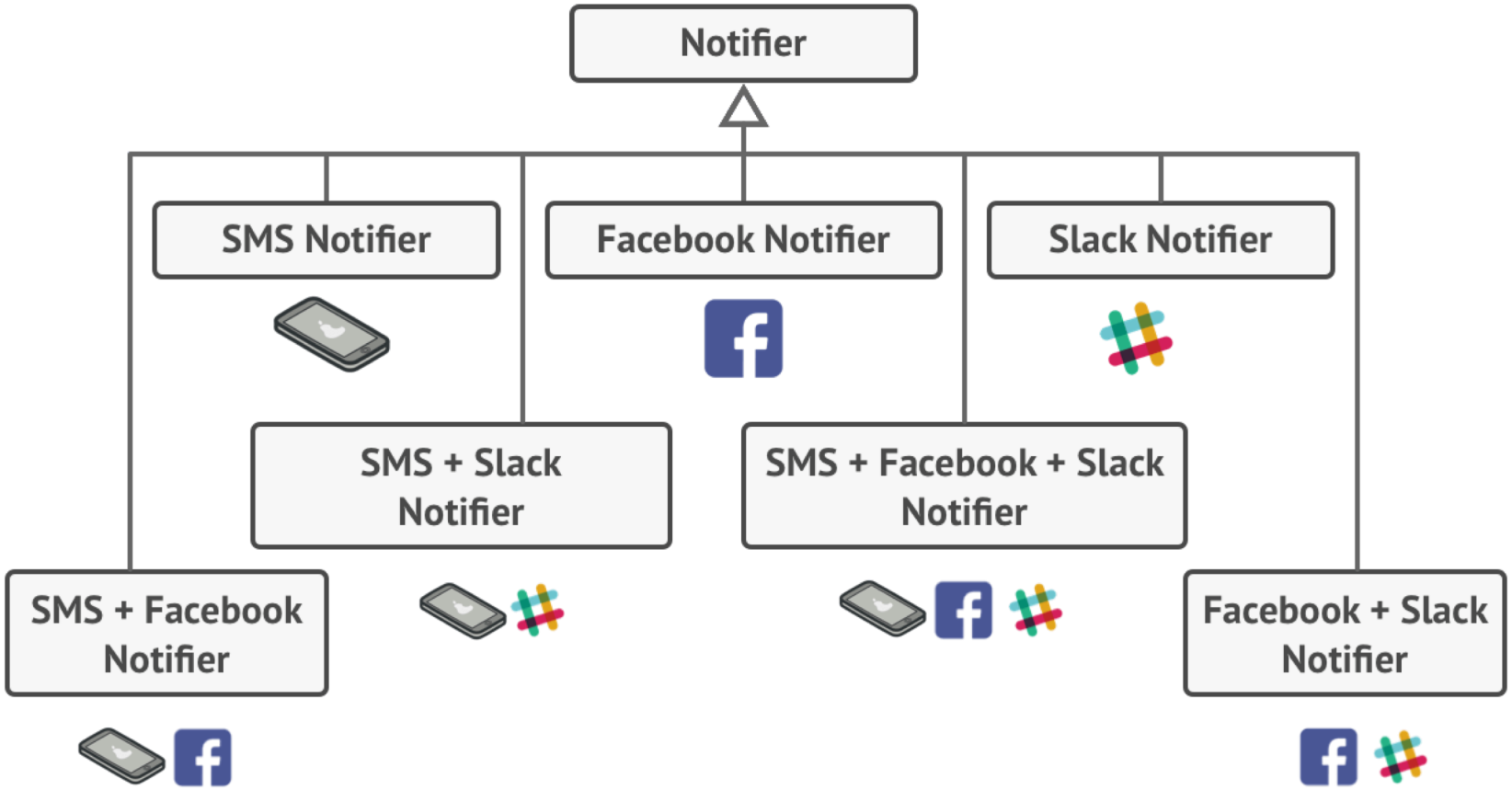


Image credit: https://refactoring.guru/design-patterns/decorator

Decorator example: notification system

Using “decorators”

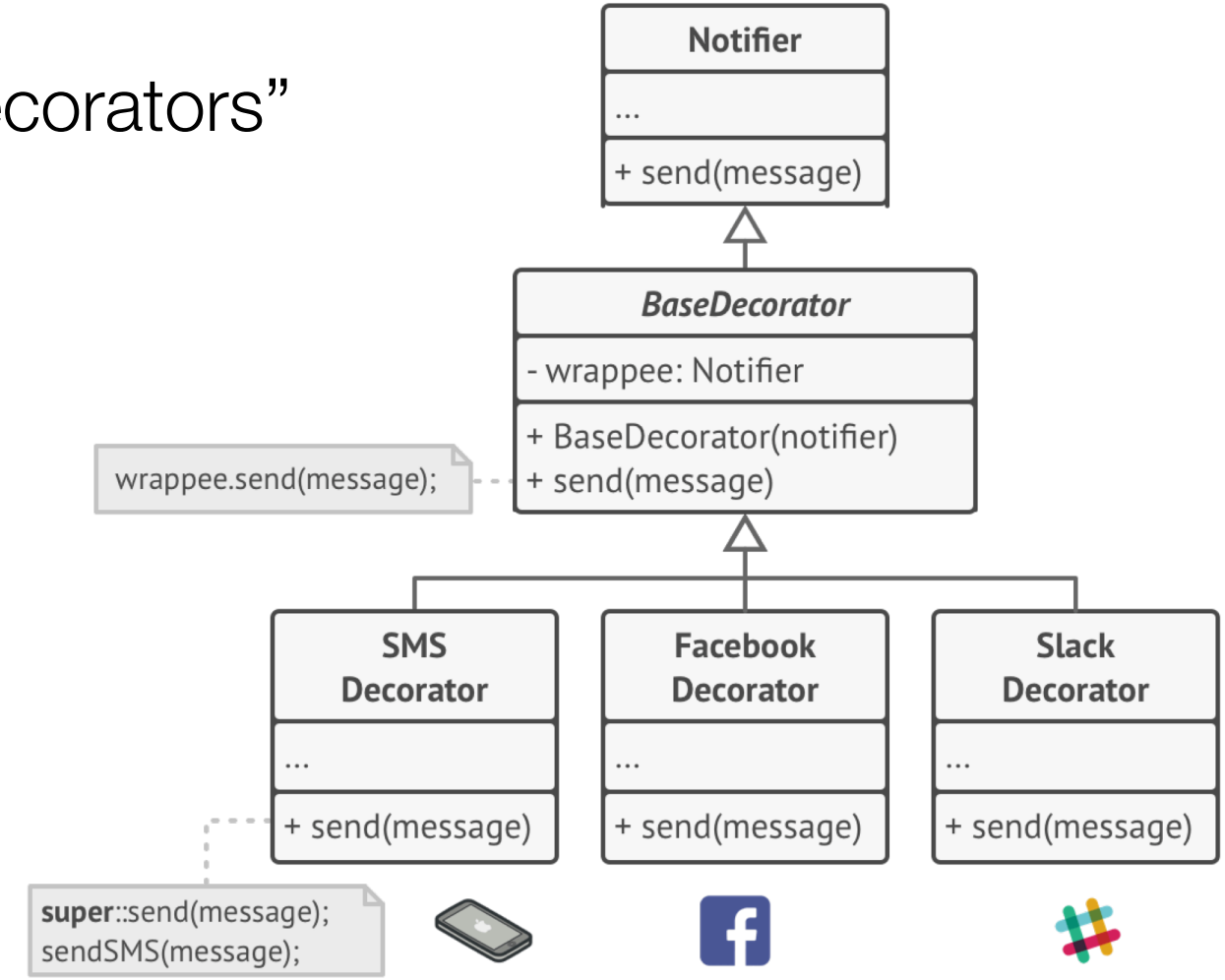


Image credit: https://refactoring.guru/design-patterns/decorator

Decorator example: notification system

Each decorator adds its functionality and returns an object of the original type

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

app.setNotifier(stack)
```

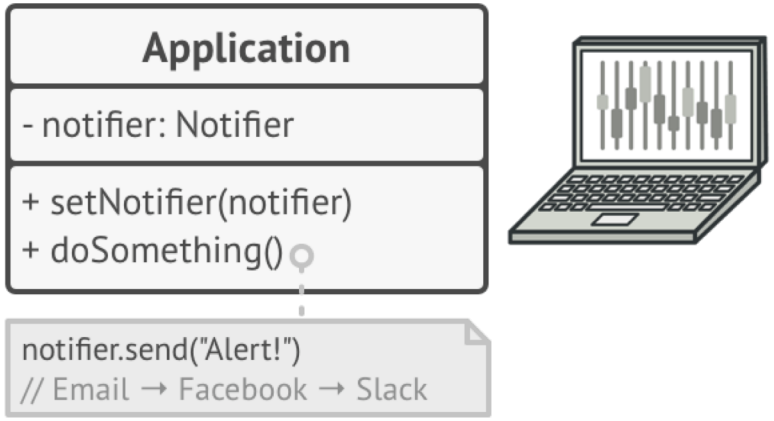


Image credit: https://refactoring.guru/design-patterns/decorator

Break (10 mins)

Walkthrough: notifications

Lecture12 > notifications package

- Decorators for each platform
- NotificationSettings tracks which platforms are enabled
- Decorators allow SocialApp to customize notifications

Behavioral patterns

Improving or streamlining communication between objects

Iterator

Separates the structure of a collection from the process of iterating over it

Iterator

Separates the structure of a collection from the process of iterating over it **by creating a class that encapsulates a particular approach to traversing the collection**.

In Java: `Iterable<T>` and `Iterator<T>`

`ListADT<T> implements Iterable<T>`

```java
/**
 * Returns an iterator over elements of type {@code T}.
 *
 * @return an Iterator.
 */
@Override
public Iterator<T> iterator() {
    return new ListIterator<>(this);
}
```

Allows for-each iteration over the list's contents

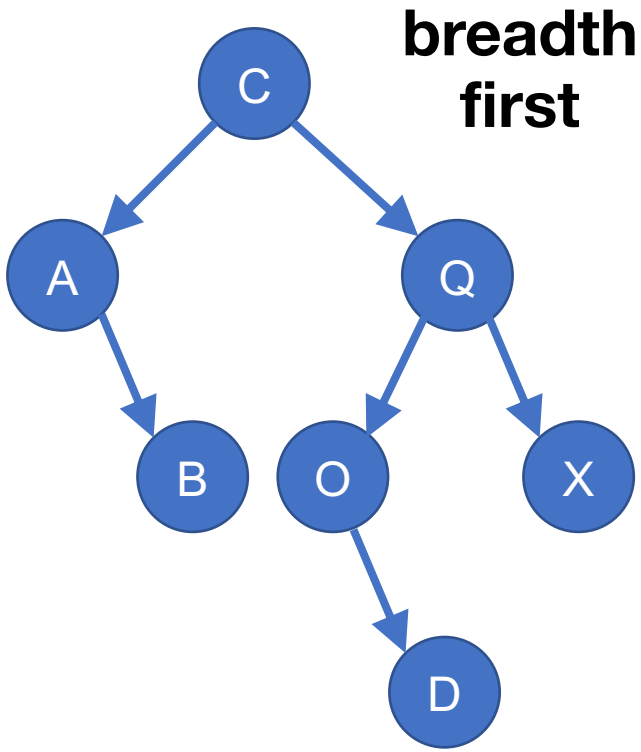
`ListIterator<T>`

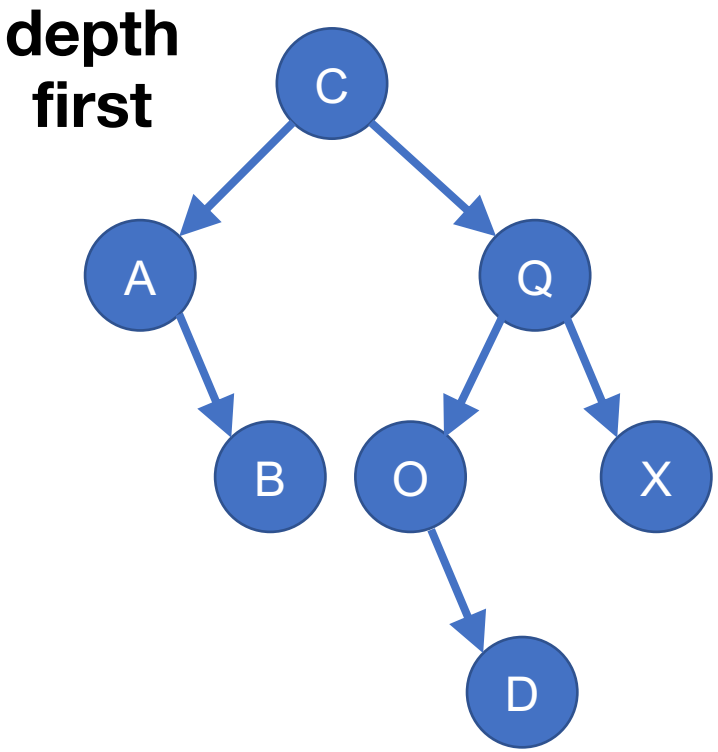
```java
public class ListIterator<T> implements Iterator<T> {
    private IListAdt<T> list;
    private int current;

    public ListIterator(IListAdt<T> list) {
        this.list = list;
        this.current = 0;
    }

    @Override
    public boolean hasNext() {
        return this.current < this.list.size();
    }

    @Override
    public T next() {
        T item = this.list.get(this.current);
        this.current++;
        return item;
    }
}
```

Iterator: not just for for-each

Iterator's can be used independently, similar to comparators

e.g. iterating a tree using breadth first or depth first traversal

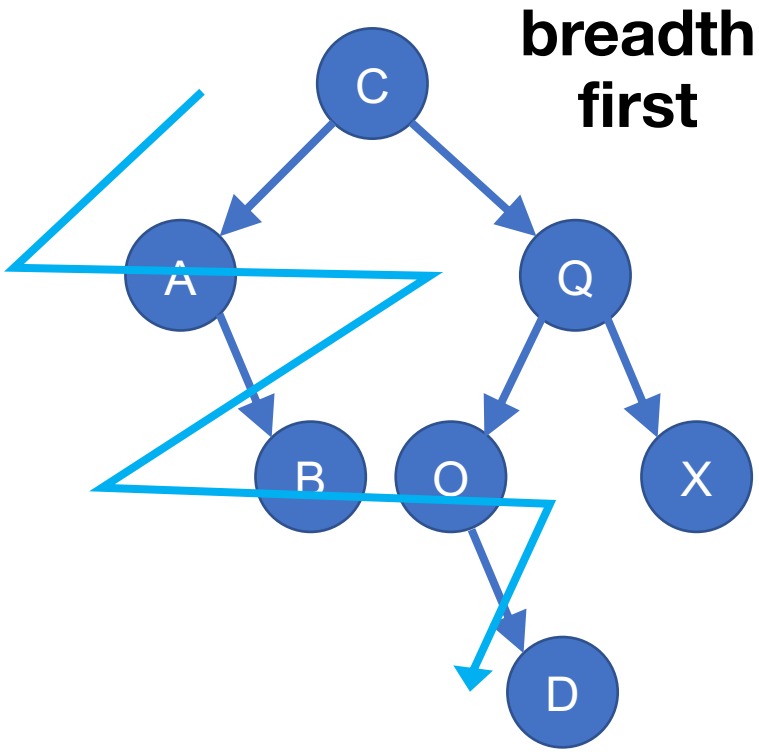


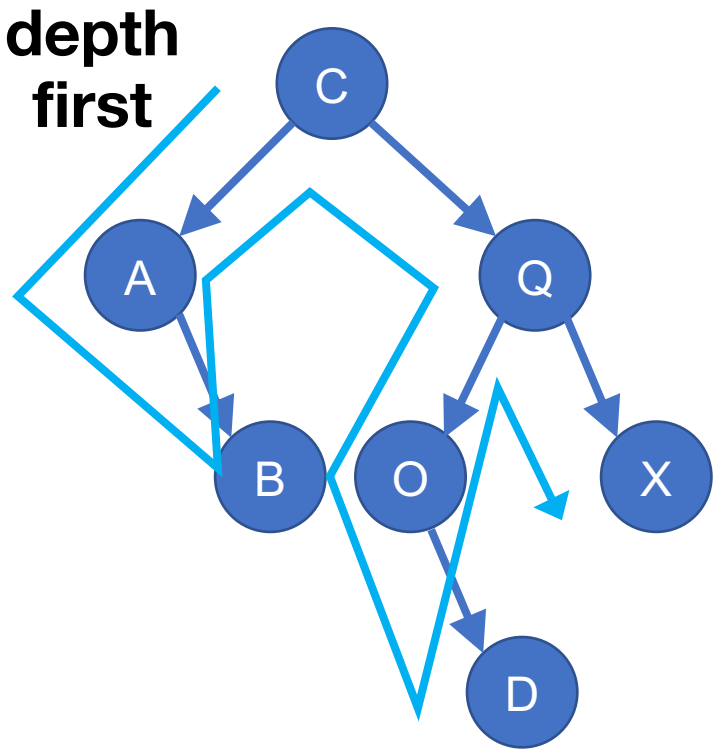


Iterator: not just for for-each

Iterator's can be used independently, similar to comparators

e.g. iterating a tree using breadth first or depth first traversal





Using an `Iterator` on its own

```
SomeIterator<T> iterator = new
SomeIterator<>(...params...);

while (iterator.hasNext())
  T next = iterator.next();
```

Walkthrough: using iterators

Lecture12 > treeiterators

- Two iterators: depth first, breadth first
- Main – uses each iterator
- (the iterator implementations only work for the non-nested version)

Observer pattern

Enable multiple objects to respond to events or state changes on another object

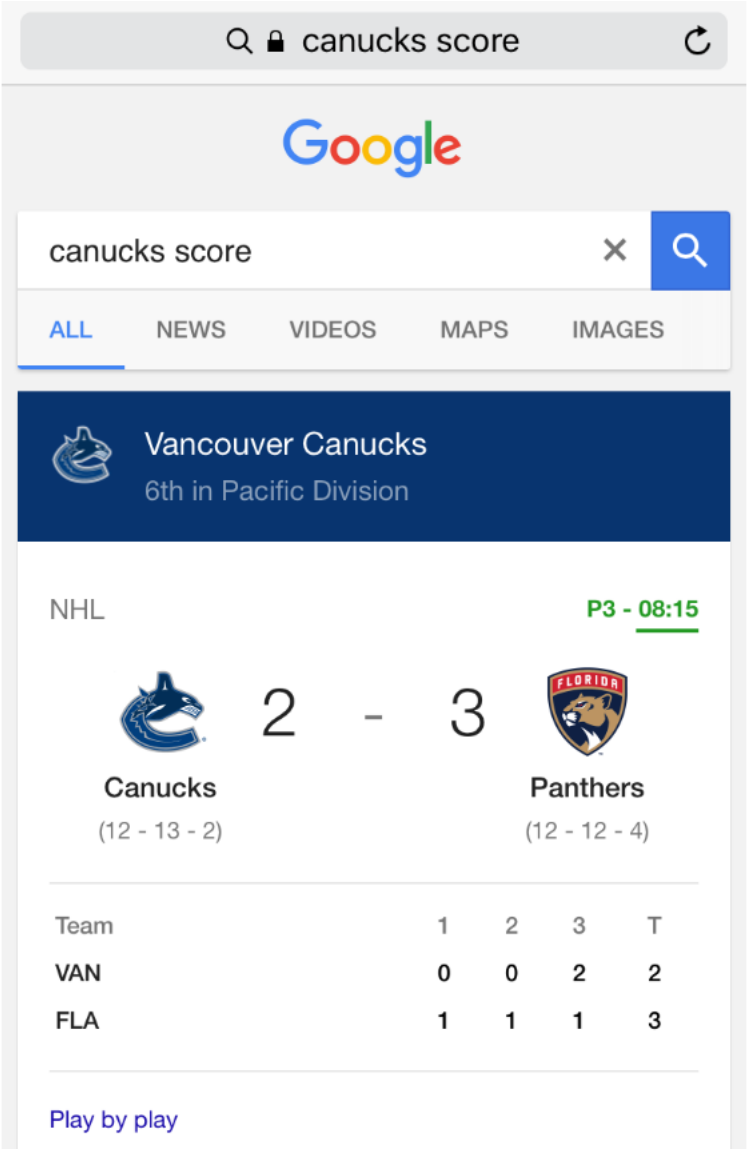
Observer pattern

- An object that might change (“observable” or “subject”) keeps a list of interested “observers” → notifies them when something happens
- Observers react however they like
- Used in many areas
  - Events and listeners for UI interactions e.g. button clicks
- Java *used* to have support for it (deprecated since version 9)
  - **`java.util.Observer`**
  - **`java.util.Observable`**

Observer pattern

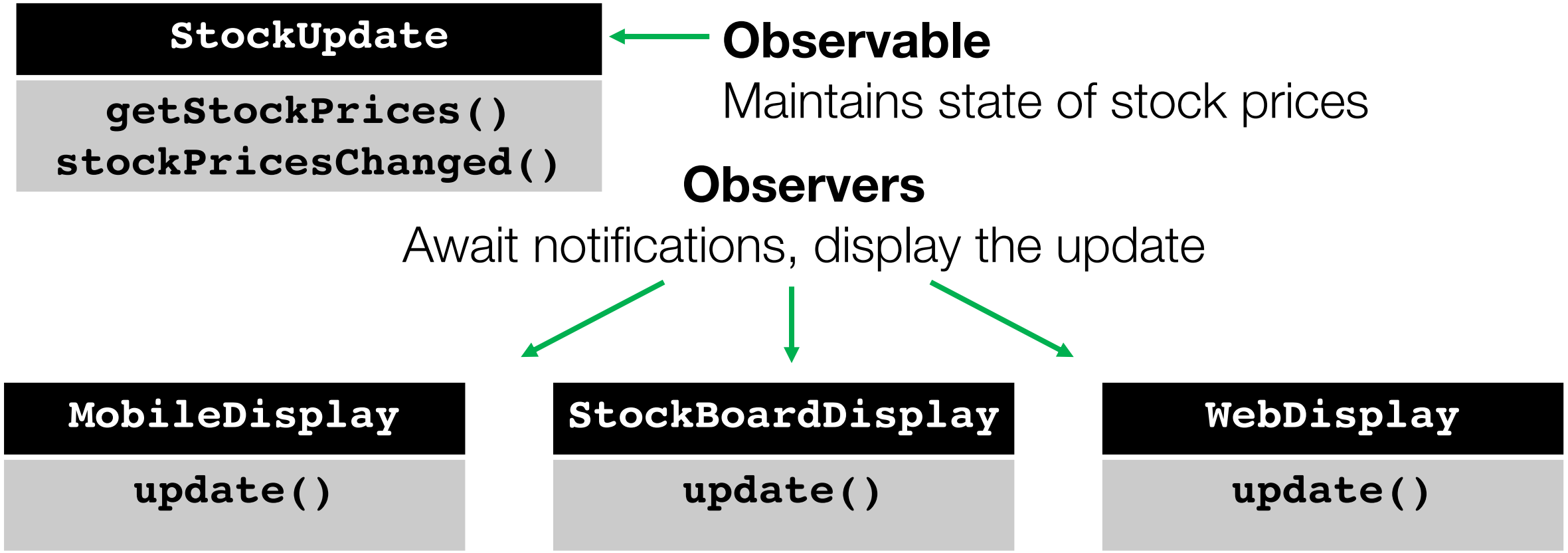
Web page observes game state

→ updates whenever there is a change



Observer pattern

Monitoring and displaying updates on stock prices



Observer pattern

What should **`StockUpdate`** class know about it's observers?

- Only how to contact them...

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

```
class StockUpdate {
  private PriceInfo priceInfo;
  private List<PriceObserver> observers;
  // constructor etc
  void addObserver(PriceObserver newObs) {
    observers.add(newObs);
  }
  void stockPricesChanged(PriceInfo newPrice) {
    this.priceInfo = newPrice;
    this.notifyObservers();
  }
  void notifyObservers() {
    for (PriceObserver obs: this.observers) {
      obs.update(priceInfo);
    }
  }
}
```

Observer pattern: key details

- The observable (StockUpdate) is not responsible for observer creation
- Application (e.g. Main) passes observers to the observable
- The observable keeps a list of observers and notifies them of changes via a callback
  - Callback = some method to call when something has happened

Walkthrough: Lottery observer

Lecture12 > lotteryobserver

- Observable: LotteryDraw
- Observer: LotteryTicket

Visitor pattern

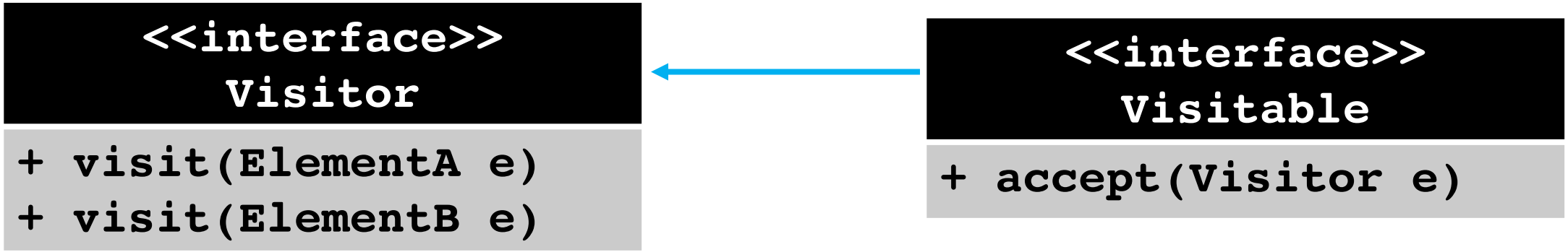
Implement functionality in a separate class that can “visit” objects of dissimilar classes and lend them consistent functionality

Visitor pattern

- Sometimes we need unrelated, varied objects to provide consistent and appropriate behavior for a specific purpose
- It may not be appropriate to include the behavior in all the classes' definitions
- Visitor pattern enables use to "visit" objects with personalized functionality as needed

Visitor pattern

- Visitable classes accept a “visitor” → via an “accept” method
- A visitor works directly with a specific concrete class
  - Via a “visit” method, overloaded for each visitable class



- Concrete visitor implementations decide how to handle implementation for each visitable type.

Visitor pattern - example

Consider an online store inventory system for a brick & mortar shop

Where would we put shipping related method, `calculatePostage()` ?

Example credit: https://dzone.com/articles/design-patterns-visitor

Visitor pattern - example

Consider an online store inventory system for a brick & mortar shop

Where would we put shipping related method, `calculatePostage()` ?

- Doesn't really belong in the item classes
- We could use the visitor pattern instead
  - Not the only option here!

Visitor pattern - example

Create a Visitable interface with an “accept” method & a Visitor class with a “visit” method for each inventory item

```
public interface Visitable {
  void accept(Visitor v);
}
```

```
public interface Visitor {
  void visit(Book b);
  void visit(Stationery s);
  void visit(Mug m);
}
```

Visitor pattern - example

Create a Visitable interface with an “accept” method & a Visitor class with a “visit” method for each inventory item

```
public interface Visitable {
  void accept(Visitor v);
}
```

```
public interface Visitor {
  void visit(Book b);
  void visit(Stationery s);
  void visit(Mug m);
}
```

implement Visitable

Book

Stationary

Mug

```
public void accept(Visitor v) {
  v.visit(this)
}
```

Visitor pattern - example

Create a Visitable interface with an “accept” method & a Visitor class with a “visit” method for each inventory item

```
public interface Visitable {
  void accept(Visitor v);
}
```

```
public interface Visitor {
  void visit(Book b);
  void visit(Stationery s);
  void visit(Mug m);
}
```

implement Visitable

Book

Stationary

Mug

```
public void accept(Visitor v) {
  v.visit(this)
}
```

Concrete Visitors determine how to implement the new functionality

Visitor pattern - example

Client class uses concrete visitor to apply the “visit” functionality:

```
public class ShoppingCart {
  // Other instance variables
  List<Visitable> items;
  PostageVisitor visitor = new PostageVisitor();
  // Constructor etc
  public double calculatePostage() {
    for (Visitable item: items) {
      item.accept(visitor);
    }
    return visitor.getTotalPostage();
  }
```

Benefits to design patterns

Shared language of design

- Decrease misunderstandings between developers

Tested solutions to common problems

- Some questions to ask:
  - Where is the solution applicable?
  - What are the tradeoffs?

Today’s exercise

A course registration system

- A course, which has a max capacity
- A course roster, which contains students
- A student, who is enrolled in courses

How to manage

- registration i.e. a student enrolls in a course
- waitlists i.e. when a student wants to enroll in a full course
- drops – a student drops a course... what happens with the waitlist