Flyweight, Interpreter, Chain of Responsibility

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 29

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

But first...

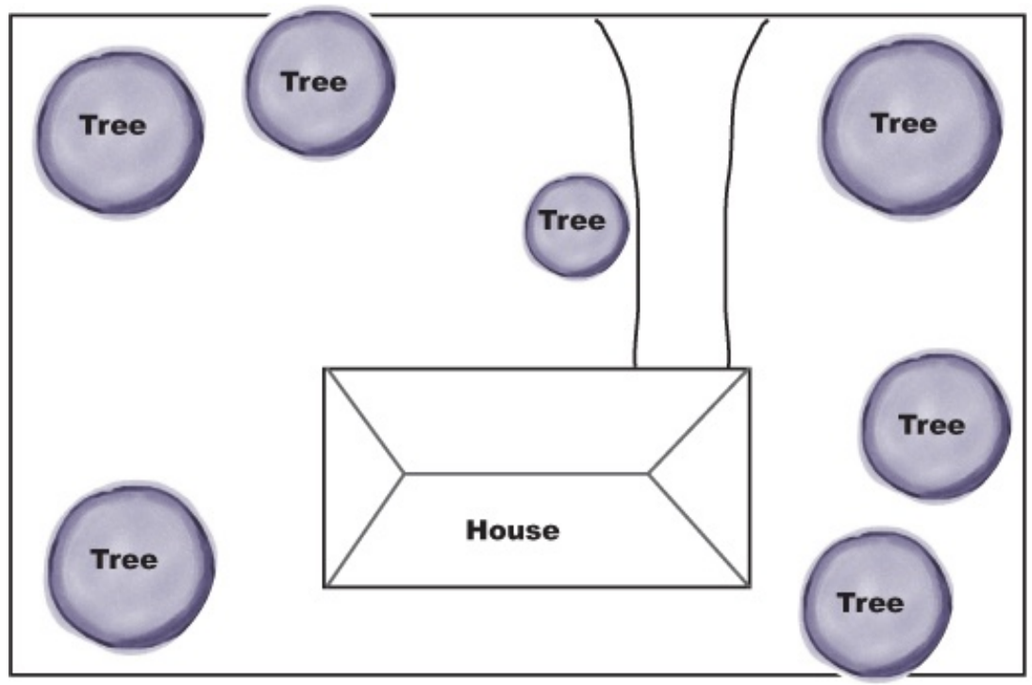
- A Special Bonus Point Exercise!

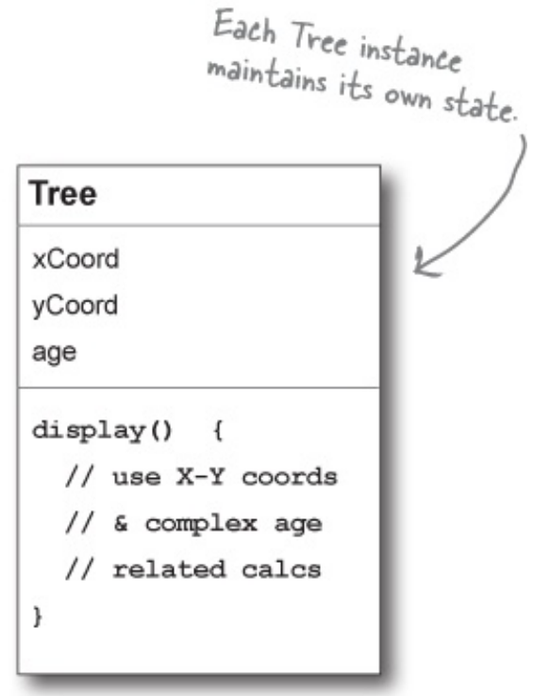
Head First Design Patterns

- Chapter 12 – Better Living with Patterns: Patterns in the Real World
  - Not too much here we haven't covered…
- Chapter 13 – Leftover Patterns
  - Bridge
  - Builder
  - Flyweight
  - Interpreter
  - Chain of Responsibility
  - Mediator
  - Memento
  - Prototype
  - Visitor

Flyweight

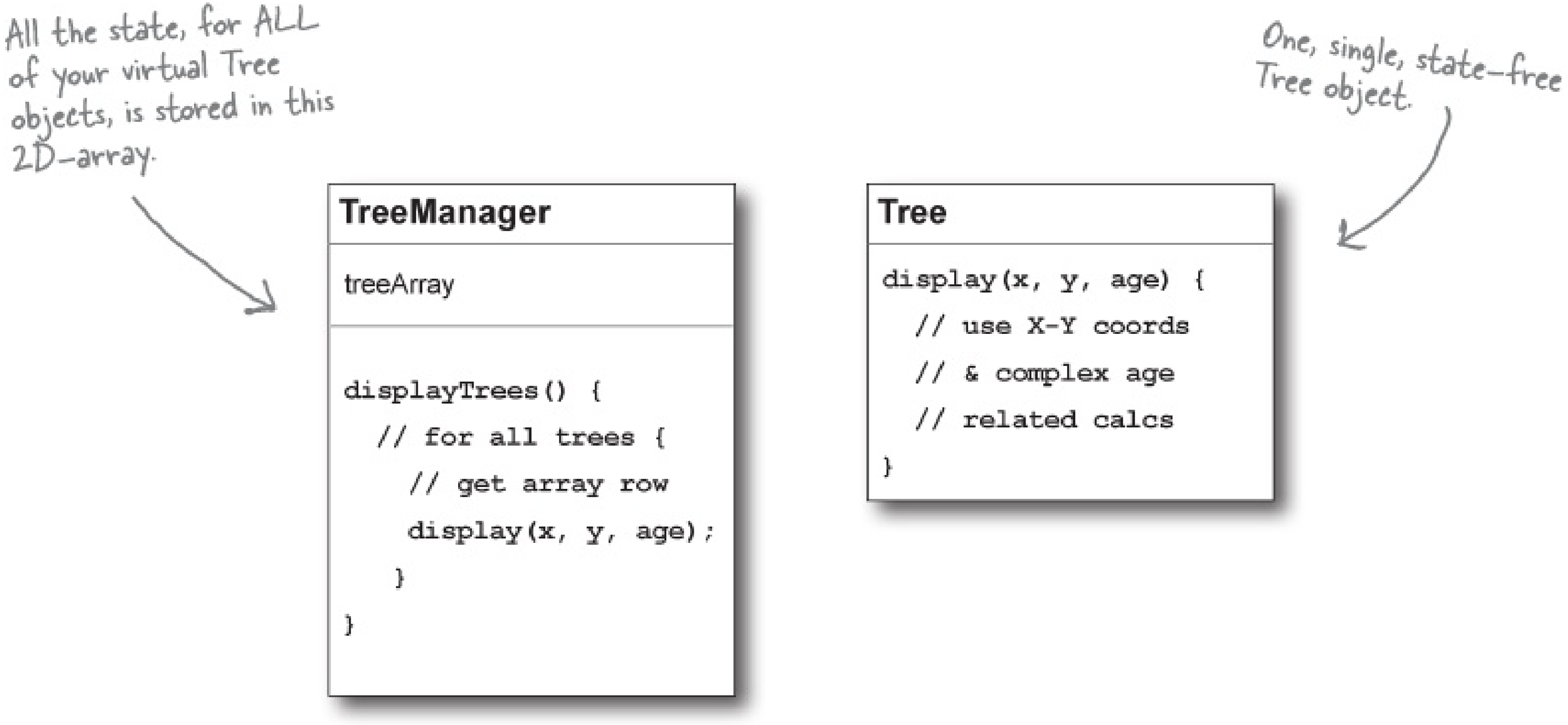
- Use the Flyweight Pattern when one instance of a class can be used to provide many “virtual instances.”
- Flyweight often recycles created objects by storing them after creation
- Head First example: Trees represented in architectural drawings





Flyweight - Example

- The book collects Tree flyweights into a TreeManager (aka a Forest)
- Note that the Flyweight objects generally are not performing operations on their own, they are part of larger operations on a collection of objects



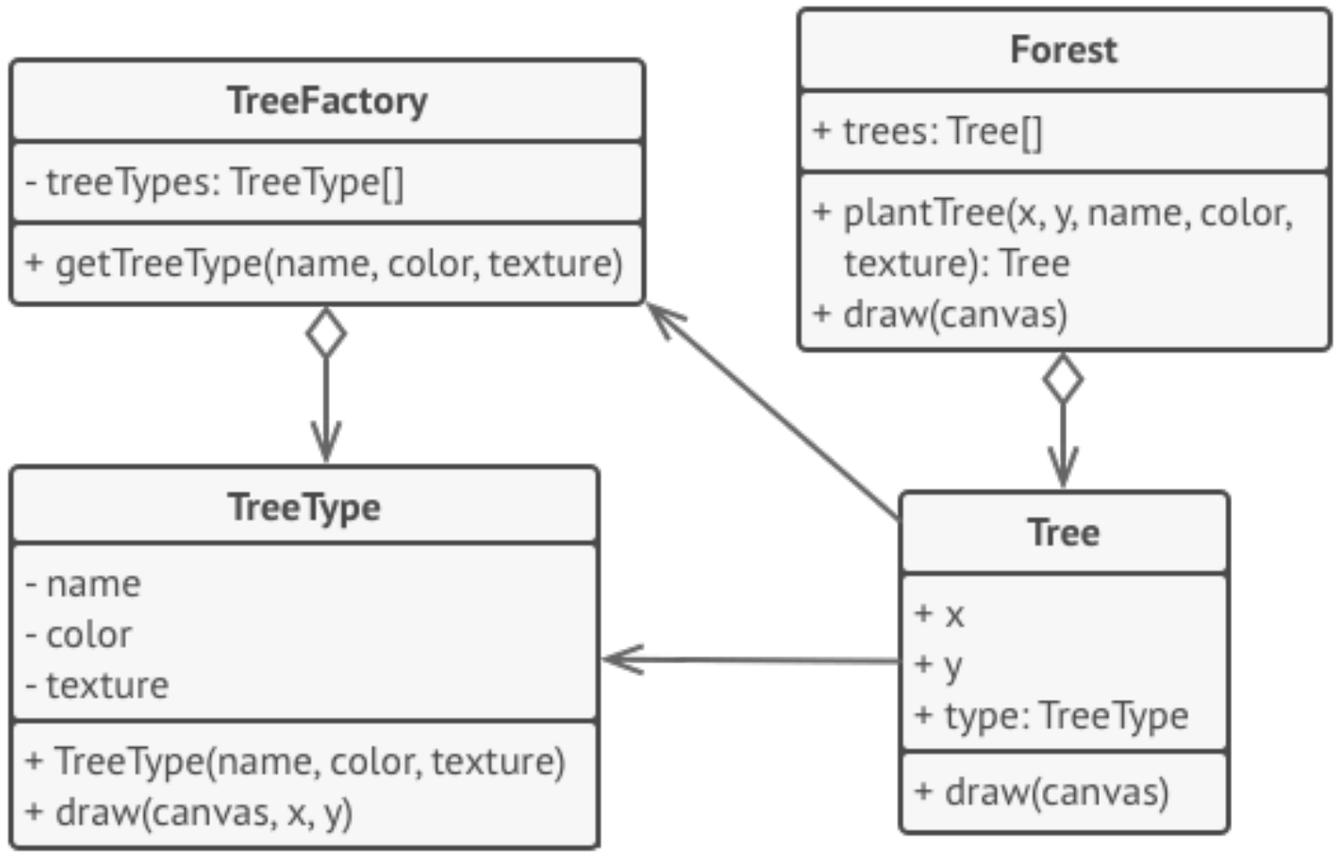
Flyweight – Intrinsic vs. Extrinsic

Flyweight discusses two states for an object: Intrinsic and Extrinsic

Constant data of an object is usually called the *intrinsic state*. It lives within the object; other objects can only read it, not change it

The rest of the object's state, often altered "from the outside" by other objects, is called the *extrinsic state*

One forest of many trees:
Many trees (location and type) = Extrinsic data, set by creators or clients, operations to request and operate on flyweights
A few tree types (and details) = Intrinsic data, often hidden or protected in some way

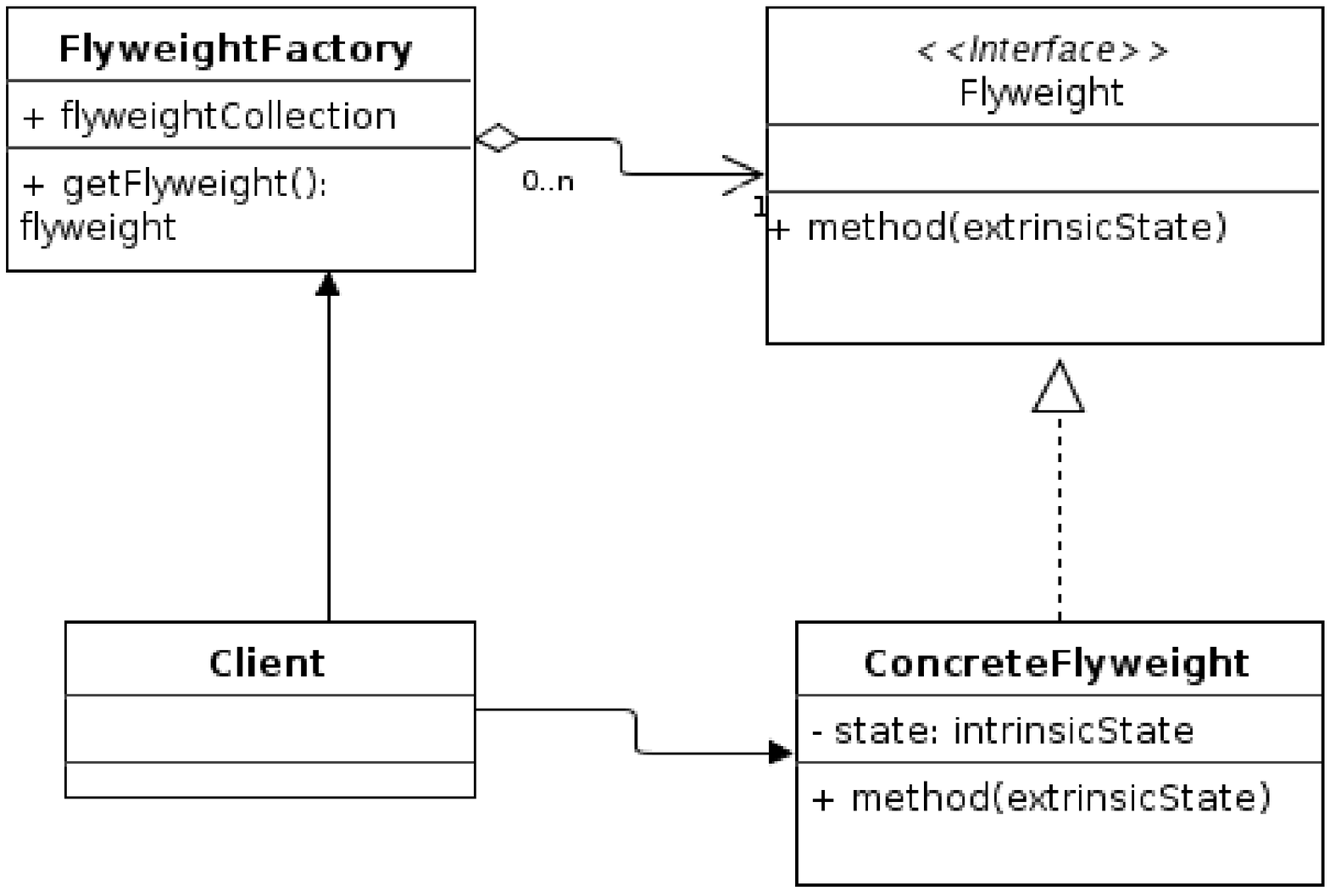


UML for Flyweight Class

In the UML for Flyweight, Flyweight itself is Abstract or an Interface

Note that the UML here does not show the idea of separating common intrinsic state data out to another object (From Tree to TreeType), but it is private data

Usually, the factory method getFlyweight will also not create a new flyweight object if one exists that can be referenced (like Object Pool)



https://javapapers.com/design-patterns/flyweight-design-pattern/

Flyweight Psuedo-code

Java example at: https://www.baeldung.com/java-flyweight
Python example at: https://refactoring.guru/design-patterns/flyweight/python/example

```
// Intrinsic state: texture, color and other repeating data in a
// separate object which lots of individual tree objects can
// reference.
class TreeType is
    field name
    field color
    field texture

    constructor TreeType(name, color, texture) { ... }

    method draw(canvas, x, y) is
        // 1. Create a bitmap of a given type, color & texture.
        // 2. Draw the bitmap on the canvas at X and Y coords.


// Flyweight factory decides whether to re-use existing
// flyweight or to create a new object.
class TreeFactory is
    static field treeTypes: collection of tree types
    static method getTreeType(name, color, texture) is
        type = treeTypes.find(name, color, texture)
        if (type == null)
            type = new TreeType(name, color, texture)
            treeTypes.add(type)
        return type
```

```
// The contextual object contains the extrinsic part of the tree
// state. An application can create billions of these since they
// are pretty small: just two integer coordinates and one reference field.
class Tree is
    field x,y
    field type: TreeType
    constructor Tree(x, y, type) { ... }
    method draw(canvas) is
        type.draw(canvas, this.x, this.y)

// The Tree and the Forest classes are the flyweight's clients.
// You can merge them if you don't plan to develop the Tree  class any
further.

class Forest is
    field trees: collection of Trees
    method plantTree(x, y, name, color, texture) is
        type = TreeFactory.getTreeType(name, color, texture)
        tree = new Tree(x, y, type)
        trees.add(tree)

    method draw(canvas) is
        foreach (tree in trees) do
            tree.draw(canvas)
```

Flyweight and other Patterns

- Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem
- Flyweight is often combined with Composite to implement shared leaf nodes
- Flyweight can combine with Object Pool to control the number of Flyweight objects if needed
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight (we'll see that shortly)
- Flyweight could also be used to model shared State objects

- https://sourcemaking.com/design_patterns/flyweight

Flyweight – Key Points

- Reduces the number of object instances at runtime, saving memory
- Centralizes state for many “virtual” objects into a single location
- The Flyweight is used when a class has many instances, and they can all be controlled identically
- A drawback of the Flyweight pattern is that once you’ve implemented it, single, logical instances of the class will not be able to behave independently from the other instances

The Interpreter Pattern

- The Interpreter pattern is used to build an interpreter for a language
- Interpreter Pattern is based on formal grammars
- Consider a language to control… a duck… in a simulator.

```
right;
while (daylight) fly;
quack;
```

Turn the duck right.

Fly all day...

...and then quack.

The formal grammar for Duck control

```
expression ::=  <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= right | quack | fly
repetition ::= while '(' <variable> ')'<expression>
variable ::= [A-Z,a-z]+
```

A program is an expression consisting of sequences of commands and repetitions ("while" statements).

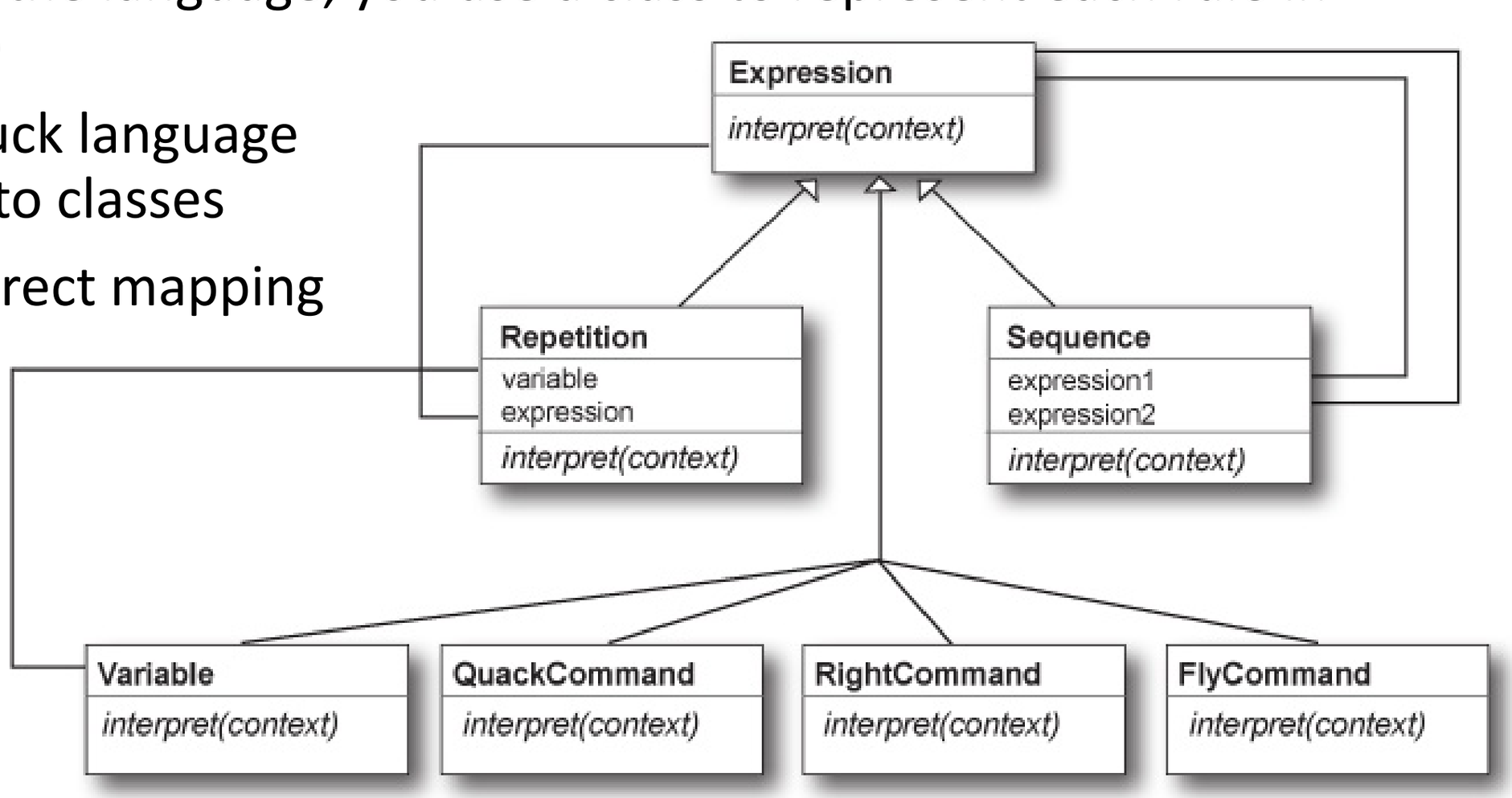
A sequence is a set of expressions separated by semicolons.

We have three commands: right, quack, and fly.

A while statement is just a conditional variable and an expression.

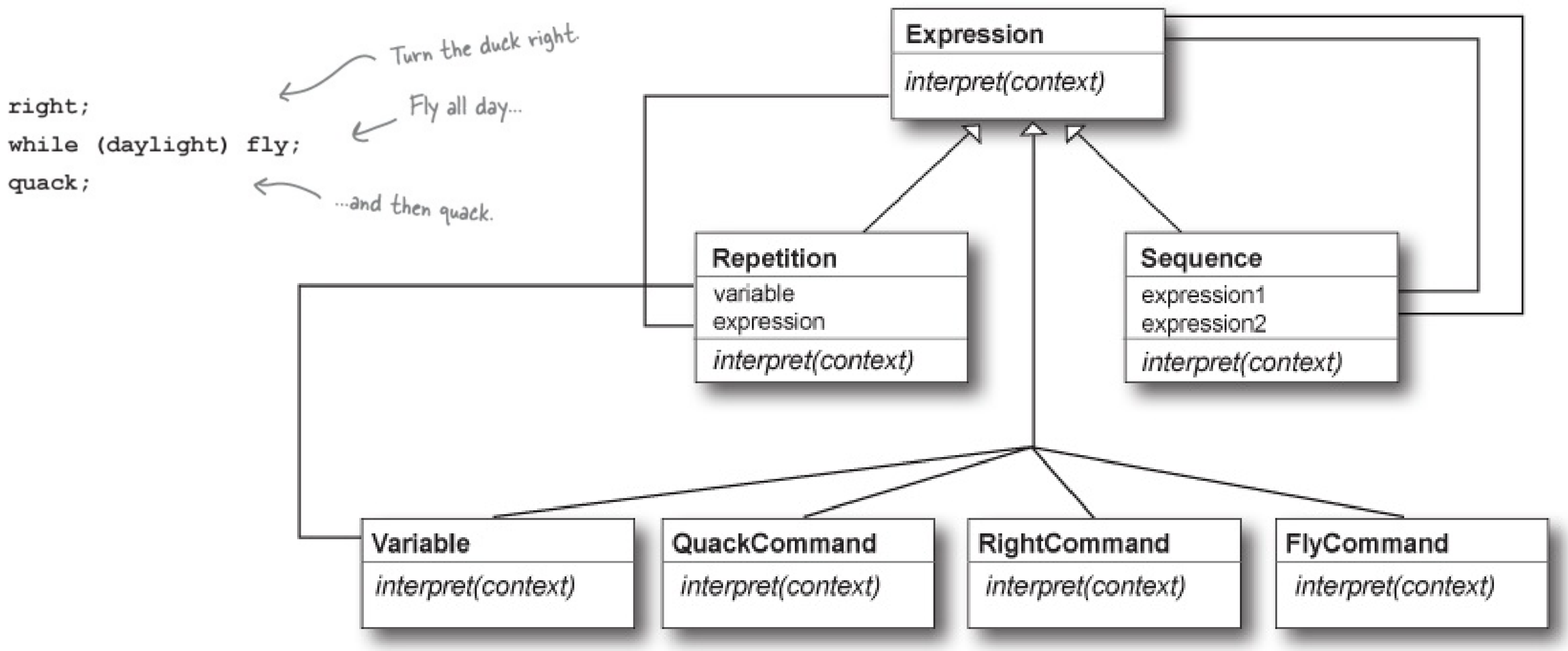
Implementing a Duck Interpreter

- When you need to implement a simple language, the Interpreter Pattern defines a class-based representation for its grammar along with an interpreter to interpret its sentences
- To represent the language, you use a class to represent each rule in the language
- Here's the duck language translated into classes
- Notice the direct mapping to the grammar.



Interpreting a language

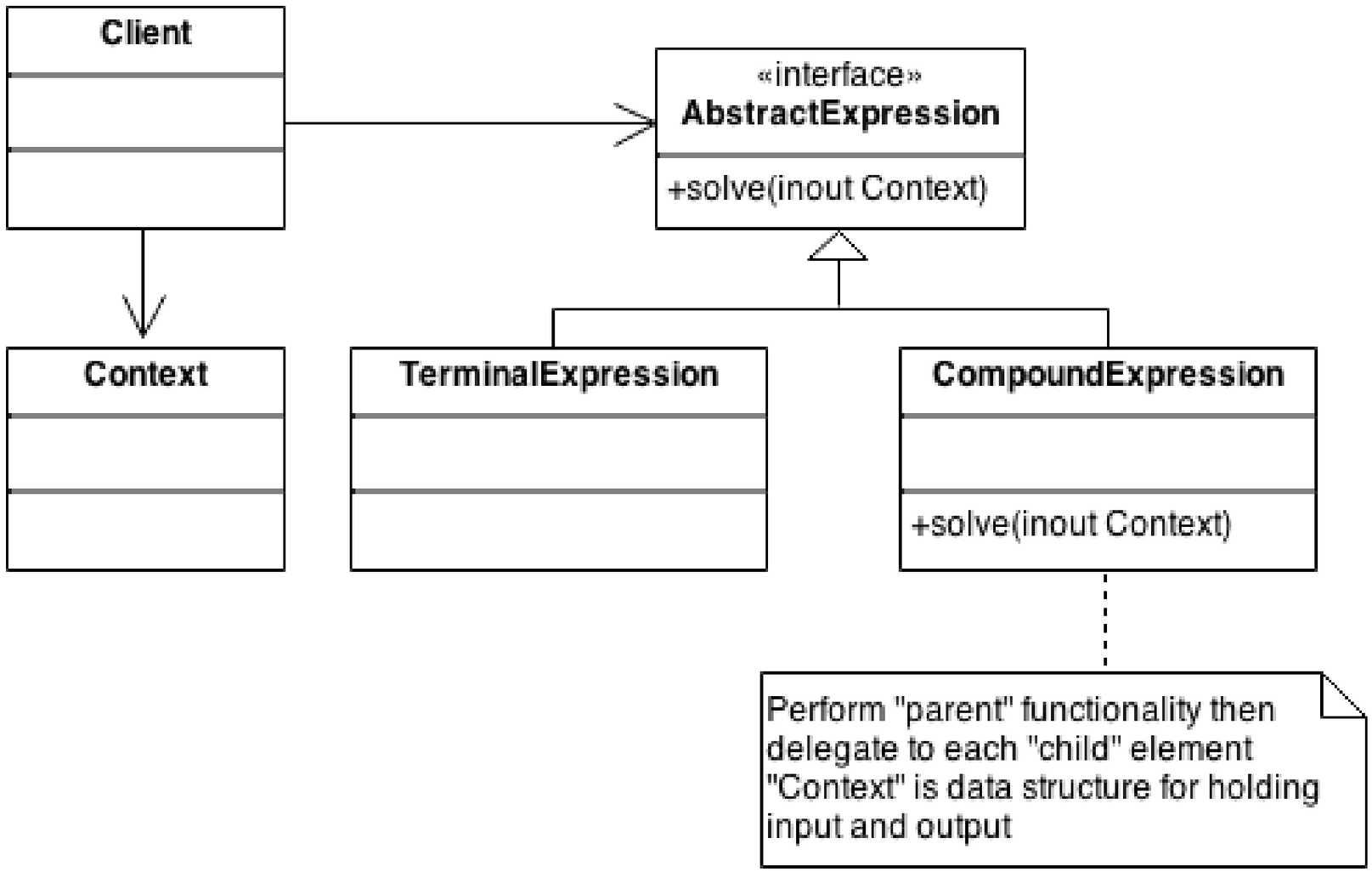
- To interpret the language, call the interpret() method on each expression type
- This method is passed a context – which contains the input stream of the program we're parsing – evaluates it – and stores the output



Interpreter UML Class Diagram

- Define a grammar for the language
- Map each production in the grammar to a class
- Organize the suite of classes into the structure of the Composite pattern
- Define an interpret(Context) method in the Composite hierarchy
- The Context object encapsulates the current state of the input and output as the former is parsed and the latter is accumulated; it is manipulated by each grammar class as the "interpreting" process transforms the input into the output

- https://sourcemaking.com/design_patterns/interpreter



Composite, Interpreter, State, Flyweight

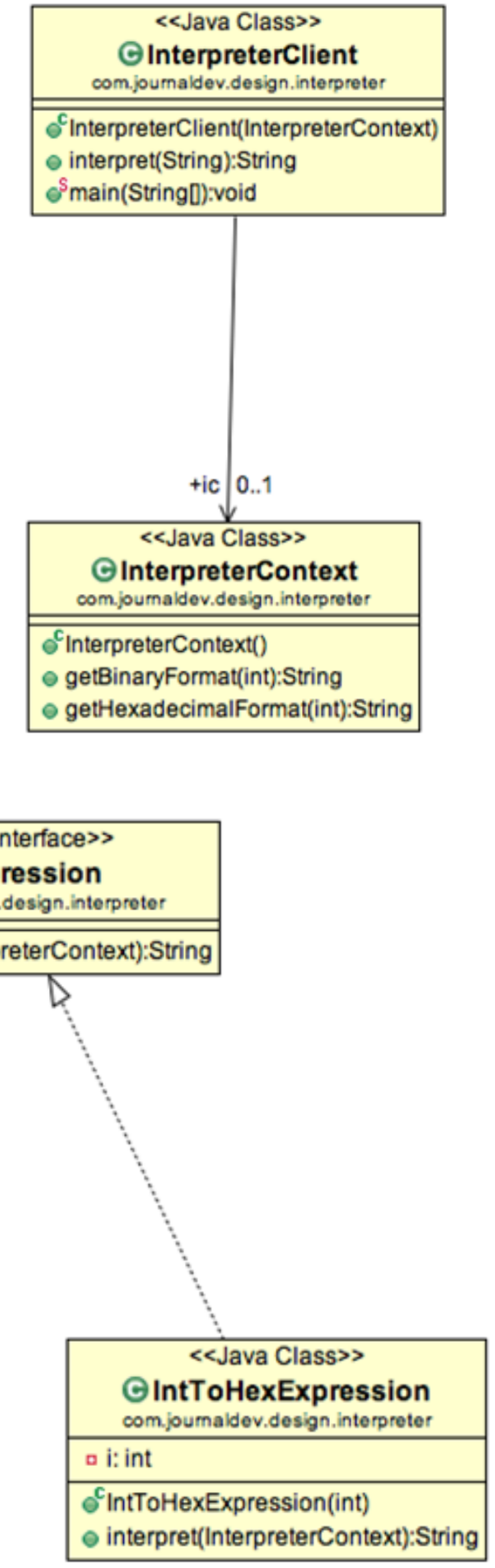
- Considered in its most general form (i.e. an operation distributed over a class hierarchy based on the Composite pattern), nearly every use of the Composite pattern will also contain the Interpreter pattern
- But the Interpreter pattern should be reserved for those cases in which you want to think of this class hierarchy as defining a language
- Interpreter can use State to define parsing contexts
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight

- https://sourcemaking.com/design_patterns/interpreter

Java Example for Integer Conversion

```
public class InterpreterContext {
        public String getBinaryFormat(int i){
                return Integer.toBinaryString(i);
        }
        public String getHexadecimalFormat(int i){
                return Integer.toHexString(i);
        }
}
```

```
public interface Expression {
        String interpret(InterpreterContext ic);
}
public class IntToBinaryExpression implements Expressi
        private int i;
        public IntToBinaryExpression(int c){
                this.i=c;
        }
        @Override
        public String interpret(InterpreterContext ic)
                return ic.getBinaryFormat(this.i);
        }
}
```



Java Example for Integer Conversion

Python example at:
https://sourcemaking.com/design_patterns/interpreter/python/1

```
public class InterpreterClient {
        public InterpreterContext ic;
        public InterpreterClient(InterpreterContext i){
                this.ic=i;
        }
        public String interpret(String str){
                Expression exp = null;
                //create rules for expressions
                if(str.contains("Hexadecimal")){
                        exp=new IntToHexExpression(Integer.parseInt(str.substring(0,str.indexOf(" "))));
                }else if(str.contains("Binary")){
                        exp=new IntToBinaryExpression(Integer.parseInt(str.substring(0,str.indexOf(" "))));
                }else return str;

                return exp.interpret(ic);
        }
        public static void main(String args[]){
                String str1 = "28 in Binary";
                String str2 = "28 in Hexadecimal";
                InterpreterClient ec = new InterpreterClient(new InterpreterContext());
                System.out.println(str1+"= "+ec.interpret(str1));
                System.out.println(str2+"= "+ec.interpret(str2));
        }
}
```

Output:
28 in Binary= 11100
28 in Hexadecimal= 1c

https://www.journaldev.com/1635/interpreter-design-pattern-java

Interpreter – Key Points

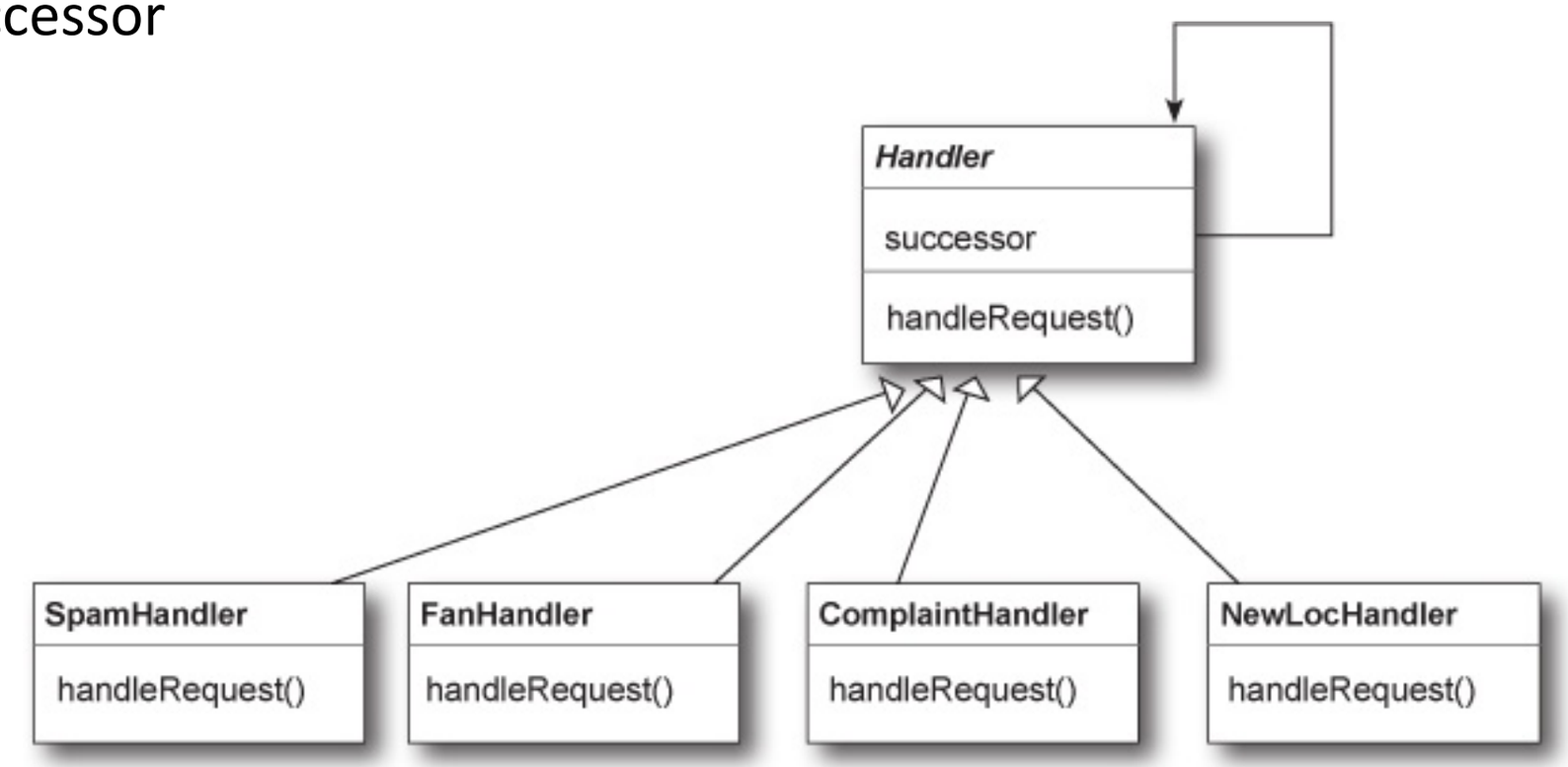
- Representing each grammar rule in a class makes the language easy to implement
- Because the grammar is represented by classes, you can easily change or extend the language
- By adding methods to the class structure, you can add new behaviors beyond interpretation, like pretty printing and more sophisticated program validation
- Use interpreter when you need to implement a simple language
- Appropriate when you have a simple grammar and simplicity is more important than efficiency
- Used for scripting and programming languages
- This pattern can become cumbersome when the number of grammar rules is large; In these cases a parser/compiler generator may be more appropriate

The Chain of Responsibility Pattern

- Use the Chain of Responsibility Pattern when you want to give more than one object a chance to handle a request…
- The scenario from the book… Gumballs…
- Mighty Gumball has been getting more email than they can handle since the release of the Java-powered Gumball Machine
- From their own analysis they get four kinds of email
  - fan mail from customers that love the new 1-in-10 game
  - complaints from parents whose kids are addicted to the game
  - requests to put machines in new locations
  - Spam
- All fan mail should go straight to the CEO, all complaints should go to the legal department and all requests for new machines should go to business development; Spam should be deleted

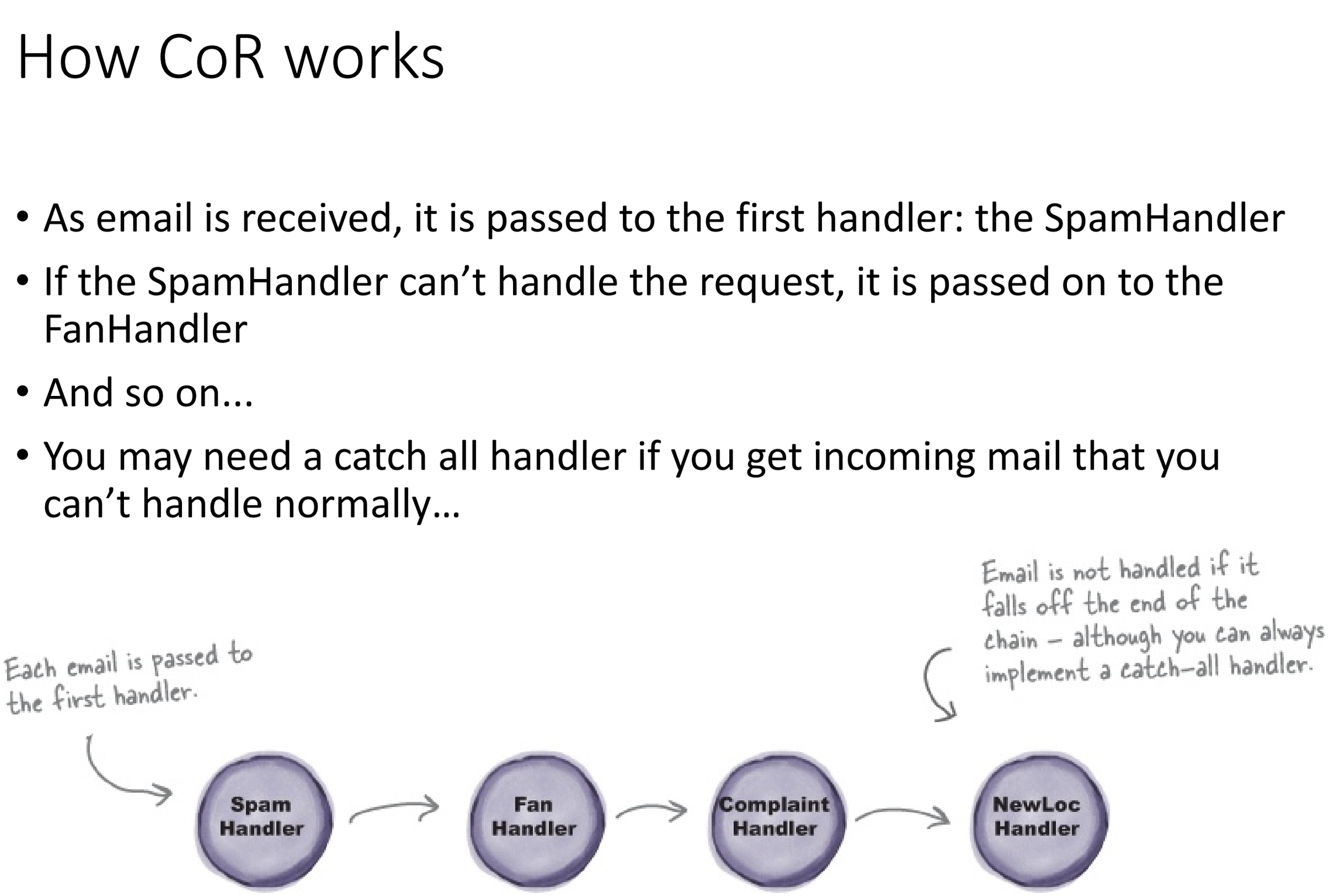
How CoR works

- With the Chain of Responsibility Pattern, you create a chain of objects to examine requests
- Each object in turn examines a request and either handles it, or passes it on to the next object in the chain
- Each object in the chain acts as a handler and has a successor object
- If it can handle the request, it does; otherwise, it forwards the request to its successor



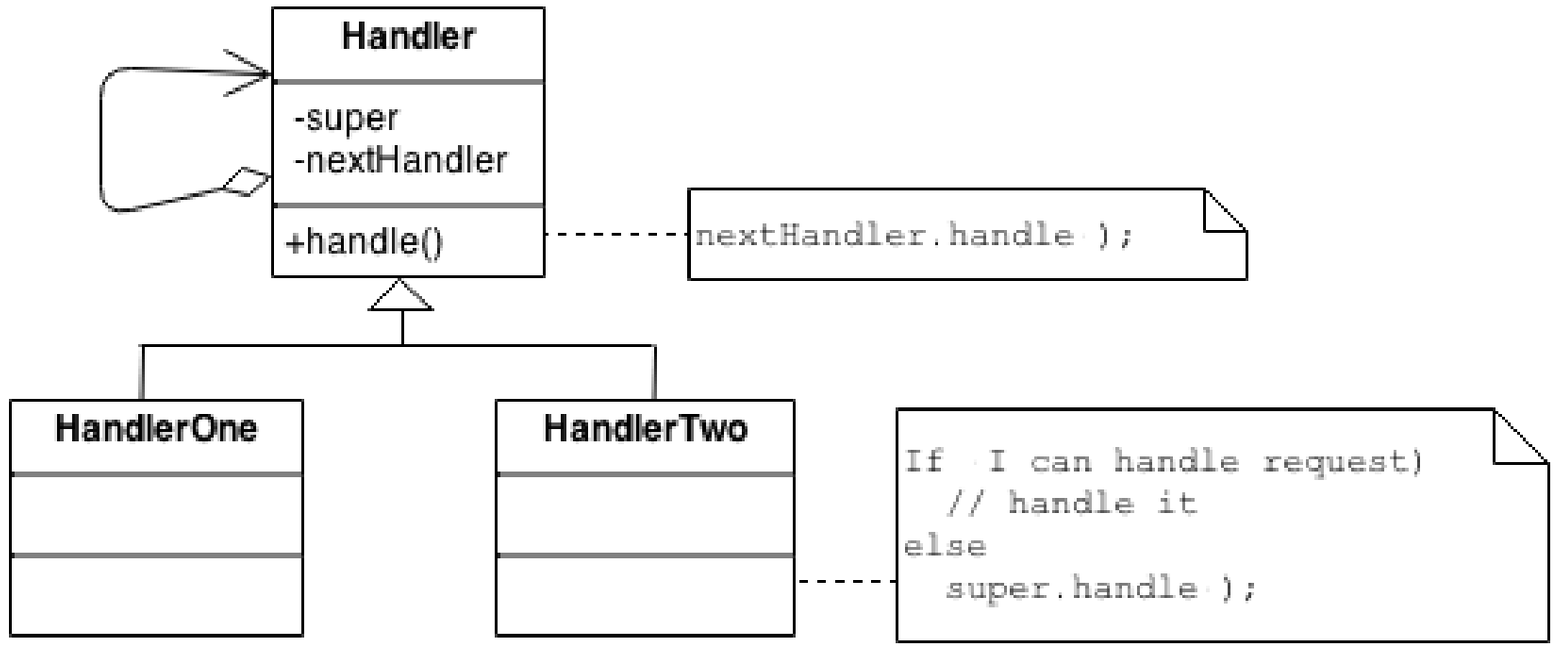
How CoR works

- As email is received, it is passed to the first handler: the SpamHandler
- If the SpamHandler can't handle the request, it is passed on to the FanHandler
- And so on...
- You may need a catch all handler if you get incoming mail that you can't handle normally…



Chain of Responsibility UML Class Diagram

- The derived classes know how to satisfy Client requests
- If the "current" object is not available or sufficient, then it delegates to the base class, which delegates to the "next" object, and the circle of life continues



- Multiple handlers could contribute to the handling of each request; the request can be passed down the entire length of the chain, with the last link being careful not to delegate to a "null next“

- https://sourcemaking.com/design_patterns/chain_of_responsibility

CoR and other Patterns

- Chain of Responsibility, Command, Mediator (tbd), and Observer address how you can decouple senders and receivers, but with different trade-offs
- Chain of Responsibility passes a sender request along a chain of potential receivers
- Chain of Responsibility can use Command to represent requests as objects
- Chain of Responsibility is often applied in conjunction with Composite, where a component's parent can act as its successor

- https://sourcemaking.com/design_patterns/chain_of_responsibility

Partial Java Example – an ATM

Python example at: https://refactoring.guru/design-patterns/chain-of-responsibility/python/example


```
import java.util.Scanner;
public class ATMDispenseChain {
        private DispenseChain c1;
        public ATMDispenseChain() {
                // initialize the chain
                this.c1 = new Dollar50Dispenser();
                DispenseChain c2 = new Dollar20Dispenser();
                DispenseChain c3 = new Dollar10Dispenser();
                // set the chain of responsibility
                c1.setNextChain(c2);
                c2.setNextChain(c3);
        }
        public static void main(String[] args) {
                ATMDispenseChain atmDispenser = new ATMDispenseChain();
                while (true) {
                        int amount = 0;
                        System.out.println("Enter amount to dispense");
                        Scanner input = new Scanner(System.in);
                        amount = input.nextInt();
                        if (amount % 10 != 0) {
                                System.out.println("Amount should be in multiple of 10s.");
                                return;
                        }
                        // process the request
                        atmDispenser.c1.dispense(new Currency(amount));
                }
        }
}
```

Full example at https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java

Chain of Responsibility – Key Points

- Decouples the sender of the request and its receivers
- Simplifies your object because it doesn’t have to know the chain’s structure and keep direct references to its members
- Allows you to add or remove responsibilities dynamically by changing the members or order of the chain
- Commonly used in windows systems to handle events like mouse clicks and keyboard events
- Execution of the request isn’t guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage)
- Can be hard to observe and debug at runtime

How I’m grading Participation…

- Mostly, it’ll be based on your Piazza posts for the discussion topics
- I’ll look for any postings you’ve made during the semester on discussion topics (tagged “participation”)
  - This includes your direct responses to discussion topics and your comments on other people’s posts
- I’ll count them up
- Then I’ll rank those counts into tiers and assign grades (up to 100 points)
- Last day to participate in the Piazza posts will be Thursday 12/9 at midnight, last day of classes

Next Steps

- Project 5 design package is due 11/3
- Graduate peer review is due 11/3
- Details on assignments in Canvas Files/Class Files
- There is a new Quiz up, due today
- New Piazza topic this week for your comments for Participation Grade
- Coming soon…
  - Mediator, Memento (App. A)
  - Prototype, Visitor (App. A)
  - Design techniques
  - ORMs/Databases
  - Refactoring
  - Dependency Injection
  - Reflection
  - Architecture
  - APIs
  - Anti-/Other Patterns
- Piazza article posts now available for extra bonus points (if you're not getting points in class)
- Office hours and GitHub user IDs for class staff are on Piazza and Canvas
- If you'd like to see Dwight, Max, or Roshan at times other than their office hours, ping them on Piazza or email
- If you'd like to see Bruce outside of office hours, use my appointment app: https://brucem.appointlet.com