Northeastern University

Design Patterns

Maria Zontak

Credits:
CS5004 course built by Dr. Therapon Skotiniotis (Northeastern)
Effective Java
Clean Code
CSE331 Dr. H. Perkins (UW)

Prelude: What is the complexity of the code below?

```
List<Integer> list = new LinkedList<Integer>();
...
for (int i = 0; i < list.size(); i++) {
    int value = list.get(i);
    if (value % 2 == 1) {
        list.remove(i);
    }
}
```

– The complexity is $O(n^2)$
– Can you spot a bug?
– What can we do?

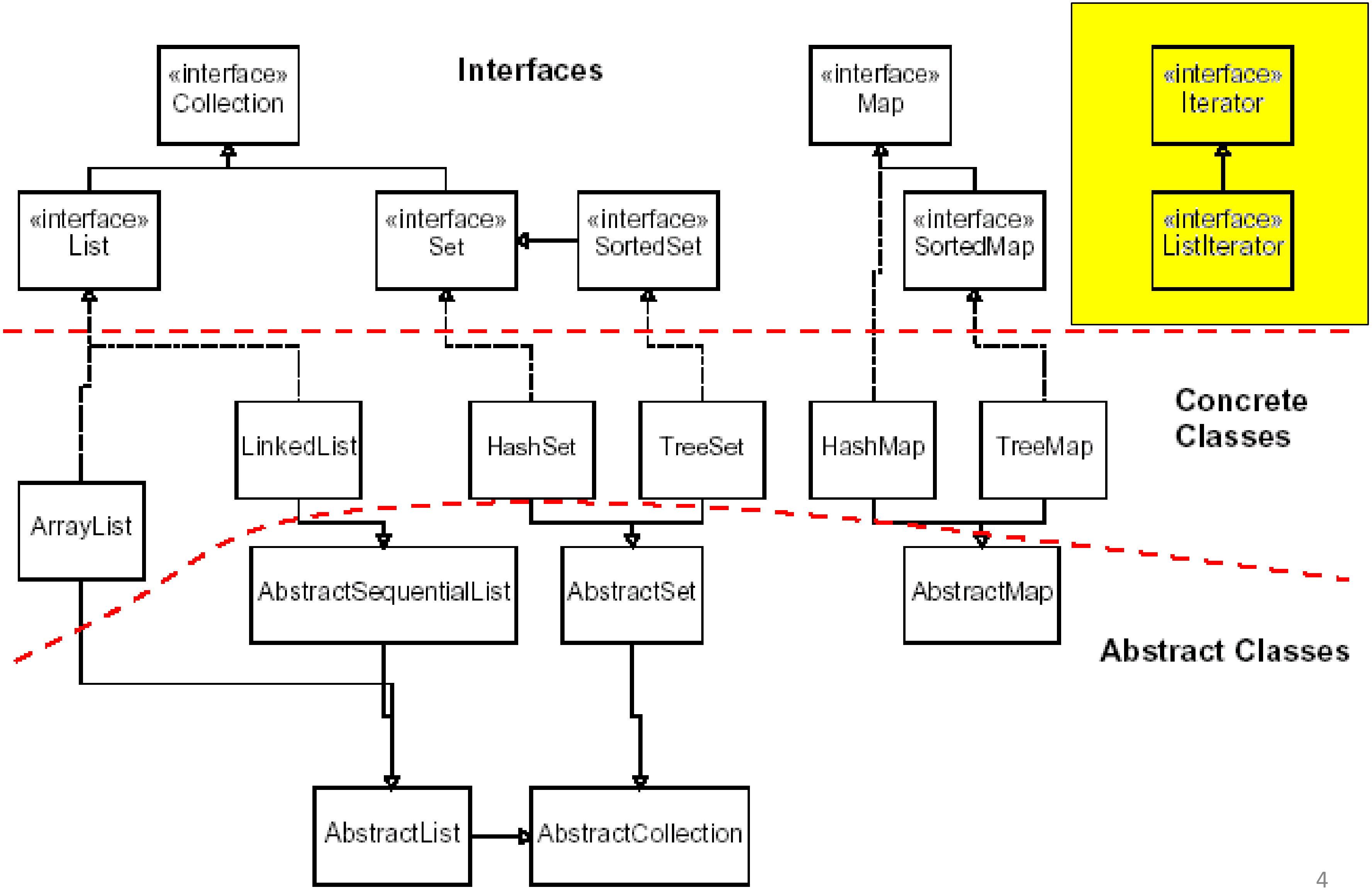
Iterator interface

`hasNext()`	returns `true` if there are more elements to examine
`next()`	returns the next element from the collection (throws a `NoSuchElementException` if there are none left to examine)
`remove()` optional	removes from the collection the last value returned by `next()` (throws `IllegalStateException` if `next()` has NOT been called yet)

Iterator

- Remembers a position within a collection, and allows to:
  - get the element at that position
  - advance to the next position
  - (optionally) remove the element at that position
- Allows to traverse the elements of a collection, regardless of its implementation → promotes abstraction

Java Collections framework



Interface Iterable<E>

```
interface List<E> extends Iterable<E> {
…
}

public abstract class AList<E> implements List<E> {
…
}

public class LinkedList<E> extends AList<E> {
…
}
```

To implement `Iterable` you should override:

`iterator()`	Returns an iterator `Iterator<E>` over a set of elements of type `E`.

GenericListIterator Implementation

```
interface List<E> extends Iterable<E> {
…
}


public abstract class AList<E> implements List<E> {
…

}


public class LinkedList<E> extends AList<E> {

    public Iterator<E> iterator() {
        return new GenericListIterator(this);
    }
…
}
```

Linked list iterator

```
public class LinkedList<E> extends AList<E> {
    ...
  private class GenericListIterator implements Iterator<E> {
      private Node current;    // current position in list

      public GenericListIterator() {
          current = front;
      }
      public boolean hasNext() {
          return current != null;
      }
      public E next() {
          if (!hasNext()) throw new NoSuchElementException("…");
          E result = current.data;
          current = current.next;
          return result;
      }

      public void remove() {       // not implemented for now
          throw new UnsupportedOperationException("Not Now");
      }
  }
}
```

```
//inner class
private class Node {
    public E data;
    public Node next;
}
```

Improving complexity and fixing the bug!

Before:

```
List<Integer> list = new LinkedList<Integer>();
...
for (int i = 0; i < list.size(); i++) {
    int value = list.get(i);
    if (value % 2 == 1) {
        list.remove(i);
    }
}
```

After:

```
List<Integer> list = new LinkedList<Integer>();
...
Iterator<Integer> itr;
for (itr = list.iterator(); itr.hasNext(); ) {
    int value = itr.next();
    if (value % 2 == 1) {
        itr.remove(); //implemented in Java
    }
}
```

Complexity now is O(n)

Copy Constructor – Simplified version

```
public ArrayList(Collection<? extends E> c) {
    this((int) (c.size() * 1.1f));//use existing
                                    constructor
    addAll(c); //Add each element in the supplied
               Collection to this List, in order that is
               specified by collection's Iterator.
}
public void addAll(Collection<? extends E> c) {
    Iterator<? extends E> itr = c.iterator();
    int csize = c.size();
    ensureCapacity(csize);
    int index = this.size;
    while(itr.hasNext()) {
        elementData[index++] = itr.next(); //get element
                                             and copy it
    }
    this.size += csize; //do not forget to update the size
}
```

Any type which is a subclass of E

`ArrayList` is implemented using bound array, with possibility to reallocate to a bigger array upon reaching maximum capacity

Some limitations…

- We can iterate only in one direction (unless you use `ListIterator`)
- Iteration can be done only once, till the end of the series

→ to iterate again, get a new Iterator

- Iterator returned by iterator() is **fail-fast**: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove methods, the iterator will throw a `ConcurrentModificationException`.

The "for each" loop – requires Iterator

```
for (type name : collection) {
    statements;
}
```

→ A clean syntax for looping over the elements of a `Set`, `List`, array, or **other collection that implements `Iterable` interface**

```
List<Integer> grades = new ArrayList<>(14);
…
for (int grade : grades) {
    System.out.println("Student's grade: " + grade);
}
```

Item 46[EJ]: Prefer for-each loops to traditional `for` loops

Why for-each?

```
// Can you spot the bug?  (From EJ, Item 46)
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN,
      EIGHT,NINE, TEN, JACK, QUEEN, KING } ...

Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());
List<Card> deck = new ArrayList<Card>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
   for (Iterator<Rank> j = ranks.iterator(); j.hasNext();)
      deck.add(new Card(i.next(), j.next()));
```

`next()` is called too many times

→ `NoSuchElementException` will be thrown

One more…

```
// Same bug, different symptom! (from EJ)
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX } ...
Collection<Face> faces = Arrays.asList(Face.values());
    for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
        for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
            System.out.println(i.next() + " " + j.next());
```

What will happen now?

The program will print 6 times from “ONE ONE” to “SIX SIX”

for - each for rescue

```
// Preferred idiom for nested iteration on
// collections and arrays (from EJ)
for (Suit suit : suits)
   for (Rank rank : ranks)
      deck.add(new Card(suit, rank));
```

However, comes with 3 limitations (things you CANNOT do):

1. **Filtering**—traversing and removing selected elements (use an explicit iterator instead)
2. **Transforming**—traversing and replacing some/all values
3. **Parallel iteration**— traversing multiple collections in parallel

List interface

```
// Represents a list of values.
public interface List<E> extends Iterable<E> {
    public void add(E value);
    public void add(int index, E value);
    public void addAll(Collection<? extends E> c)
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public Iterator<E> iterator();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}
```

Adding Static Factory to `List` interface

```
// Represents a list of values.
public interface List<E> extends Iterable<E> {

    //static factory method
    public static <E> List<E> createLinkedList(){
      return new LinkedList<E>();
    };

    public void add(E value);
    public void add(int index, E value);
    public void addAll(Collection<? extends E> c)
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public Iterator<E> iterator();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}
```

We do NOT know concrete type

And what EJ thinks about Static Factory methods?

Item 1: Consider (using) static factory methods instead of constructors

Advantages of static factory methods:

- Have names.
- NOT required to create a new object each time they are invoked
  - → This allows immutable classes to use preconstructed instances, or to cache instances, and dispense them repeatedly to avoid duplicate objects (better performance if creating an instance is expensive)
  - → Can be used to create Singletons
  - → Instance – control classes (Class has control over created instances).
  - → You might want to consider NON-Public constructor (Item 4) or even private to enforce non- insatantiability
- Can return an object of ANY subtype of their return type

But there are some disdvantages

- Classes without public or protected constructors cannot be sub-classed.

→ "Blessing in disguise" – encourages usage of composition over inheritance

- They are not salient compared to other static methods.

KNOW PROS and CONS of your DESIGN

Open-Closed Principle

Software entities should be:

- ***Open for Extension***

But

- **Closed for Modification**

→ To add NEW features to your system:

- Add new classes or reuse existing ones in new ways
- If possible, do NOT make changes by modifying existing ones. Why?
  - Existing code works and changing it can introduce bugs and errors.

Find Problems with the method below?

```
//Payroll.java From CleanCode Listing 3-4
// What OO Design principles are violated here?
public Money calculatePay(Employee e) {
    switch (r.getType()) {
        case COMMISSIONED: return calculateCommissionedPay(e);
        case HOURLY: return calculateHourlyPay(e);
        case SALARIED: return calculateSalariedPay(e);
        default: throw new InvalidEmployeeType(e.type);
    }
}
```

`switch` will ALWAYS do more than one thing

- It is large and when new employee types are added → it will grow

→ Violates the Open Closed Principle (OCP) because must change whenever new types are added

- Problem might repeat in other `Employee` methods `isPayday`, and `deliverPay`.

Abstract Factory [GOF]

- The factory will use the `switch` statement to create appropriate instances of the derivatives of `Employee`
- The various methods, such as `calculatePay`, `isPayday`, and `deliverPay`, will be dispatched **<u>polymorphically</u>** through the `Employee` interface.
- `switch` statements can be tolerated if
  - they appear only once
  - are used to create polymorphic objects
  - are hidden behind an inheritance relationship so that the rest of the system can NOT see them
- This rule might be violated

Abstract Factory [GOF]

```
//Employee and Factory - Clean code Listing 3-5
public interface Employee {
   boolean isPayday();
   Money calculatePay();
   void deliverPay(Money pay);
}

Public interface EmployeeFactory {
   Employee makeEmployee(EmployeeRecord r);
}

public class ConcreteEmployeeFactory implements EmployeeFactory {
   Employee makeEmployee(EmployeeRecord r){
      switch (r.getType()) {
         case COMMISSIONED:return new CommissionedEmployee(r);
         case HOURLY: return new HourlyEmployee(r);
         case SALARIED: return new SalariedEmploye(r);
         default: throw new InvalidEmployeeType(r.getType());
      }
   }
}
```

Abstract Factory vs. Factory method

Factory Method pattern:

- A single method
- Uses inheritance and relies on a subclass to handle the desired object instantiation.

Abstract Factory pattern:

- Encapsulates many factory methods
- Has a single responsibility of creating a FAMILY of objects.
- Another class delegates the responsibility of object instantiation to the Factory class

- The usage of `Employee` is decoupled from constructing `Employee`
- Promotes Single Responsibility Principle

Abstract Factory Example:

```
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();
    …
  }
  …
}
```

Problem:
We are reimplementing the constructor in every **`Race`** subclass in order to use a different subclass of **`Bicycle`**

Abstract Factory Example:

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

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

HOW does this help?

```
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 Example:

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
- Separation of concerns

```
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 Example:

```
class Race {
  BicycleFactory bfactory;
  public Race(BicycleFactory f) {
    bfactory = f;
    Bicycle bike1 = bfactory.createBicycle();
    Bicycle bike2 = bfactory.createBicycle();
    …
  }
  public Race() { this(new BicycleFactory()); }
  …
}
```

Promotes composition over inheritance

```
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 FreeRace (can have ANY type of bicycle)?

```
class TourDeFrance extends Race {
  public TourDeFrance() {
    super(new RoadBicycleFactory()); // or this(…)
  }
  public TourDeFrance(BicycleFactory f) {
    super(f);
  }
  …
}
```

Mid-way summary

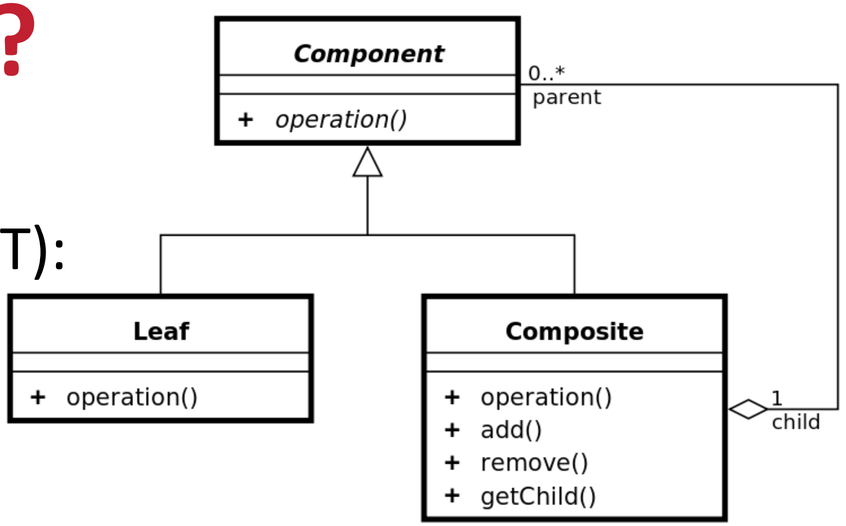
✓ Creational patterns (constructing objects)

- Behavioral patterns (affecting object semantics)
  - Already seen: Observer
  - Now Interpreter vs. Visitor

Traversing composites

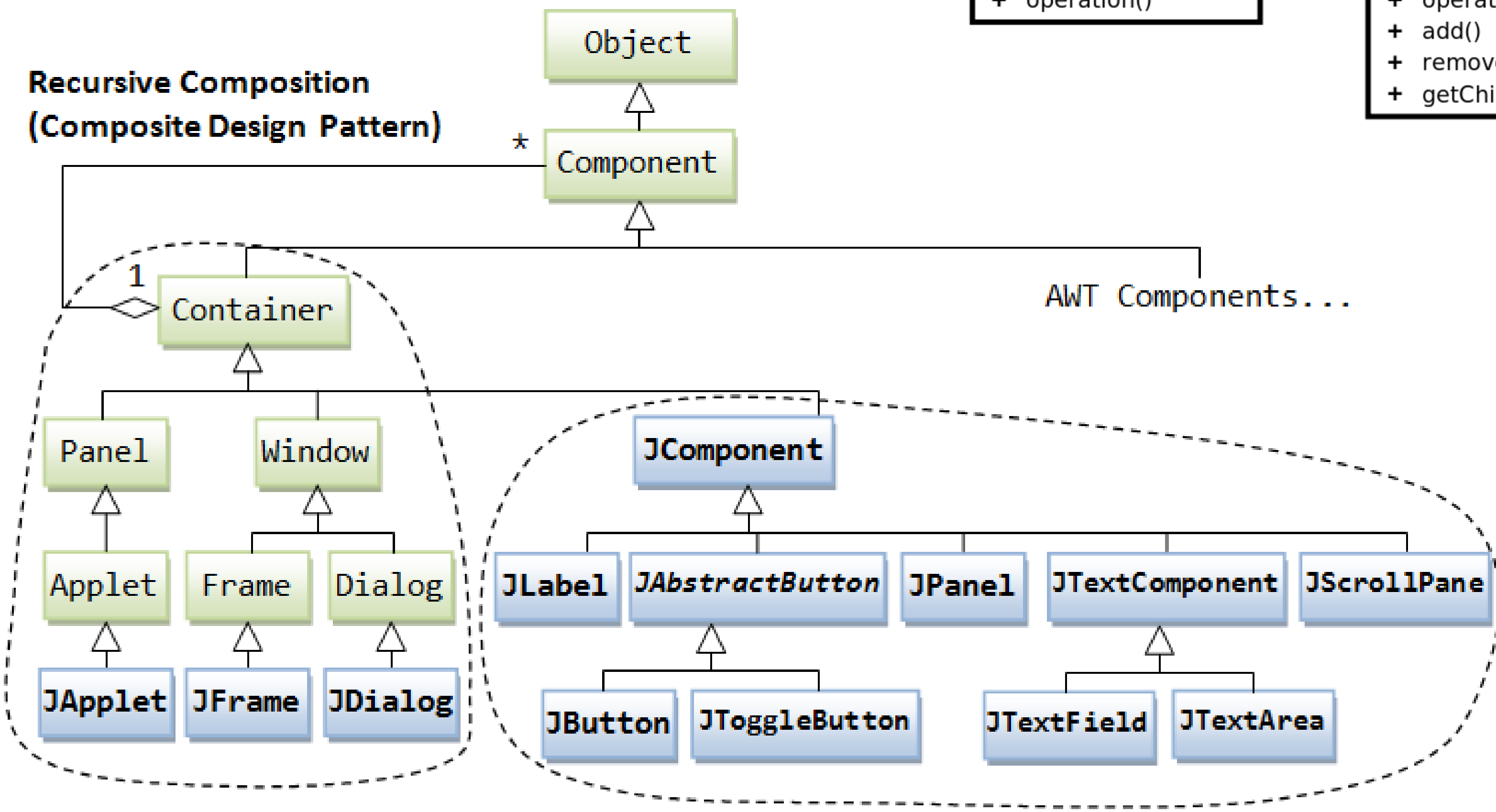
- Goal: perform operations on all parts of a composite
- Idea: generalize the notion of an iterator – process the components of a composite in an order appropriate for the application
- Separate Processing from Traversing

What is a composite?



Composite Pattern (structural pattern):

Very useful in GUI Libraries (Swing, AWT):



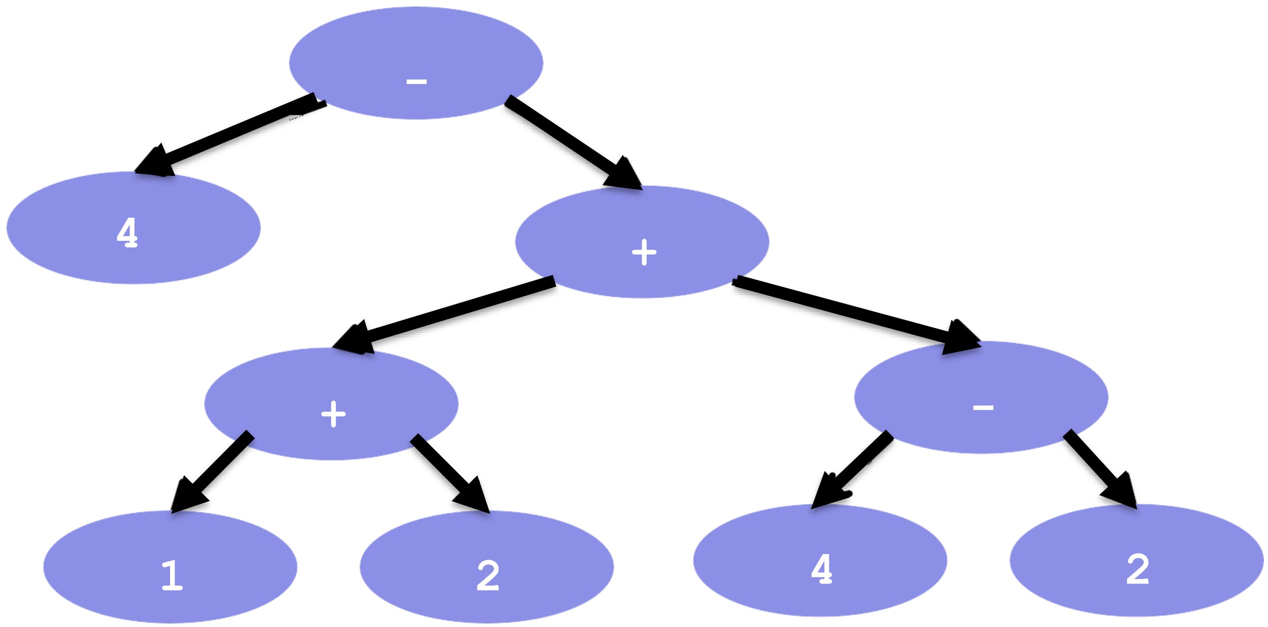
Simple Arithmetic Calculator

Our calculator deals only with integers and supports the following operations:

- **addition** given two sub-expressions perform mathematical addition
- **subtractions** given two sub-expressions perform mathematical subtraction
- **unary minus** given one sub-expression return it's negative value

Simple Arithmetic Calculator

```
4 - 1 + 2 + 4 - 2;
```



Simple Arithmetic Calculator Herarchy

```
public interface Expression {
        int evaluate();
        String asString();
}

class Value implements Expression {
        private int value;
}
class UnaryOp implements Expression {
        private Expression singleExp;
}
abstract class BinaryOp implements Expression {
        protected Expression leftExp;
        protected Expression rightExp;
}
class PlusOp extends BinaryOp {
}
class MinusOp extends BinaryOp {
}
…
```

Operations on AST

Need to write code for each entry in this table

	Types of Objects	
Operations	UnaryOp	PlusOp
evaluate		
asString		

- What code should we group together?
  - the code for a particular operation, or
  - the code for a particular expression
  - → Do we group the code into rows or columns?
- Given an operation and an expression, how do we “find” the proper piece of code?

Procedural Design vs. Object Oriented

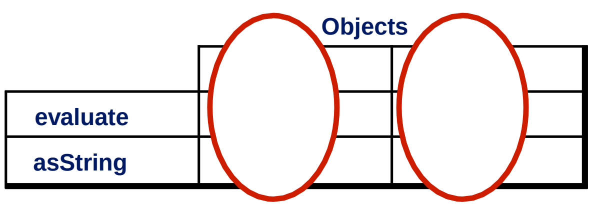
Procedural code makes it easy to add new methods without changing the existing data structures.

OO code makes it easy to add new classes without changing existing methods.

Interpreter and Visitor patterns

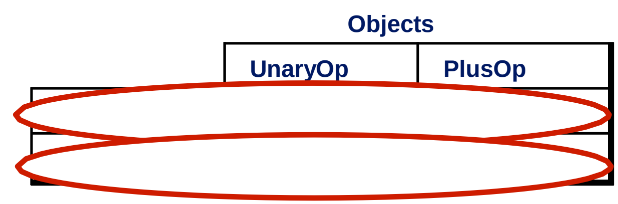
Interpreter:
collects code for similar objects, spreads apart code for similar operations

- Easy to add types of objects, hard to add operations



Visitor:
collects code for similar operations, spreads apart code for similar objects

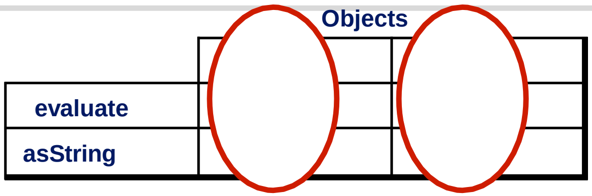
- Easy to add operations, hard to add types of objects



Selecting between interpreter and procedural:

- Are the algorithms central, or are the objects?
- What aspects of the system are most likely to change?

Interpreter pattern



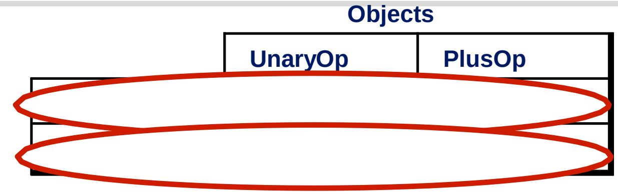
Add a method to each class for each supported operation

```
class UnaryOp implements Expression {
...
       int evaluate() { ... }
       String asString() { ... }
}
abstract class BinaryOp implements Expression {
…
}
class PlusOp extends BinaryOp {
       int evaluate() { ... }
       String asString() { ... }

…
}

…
```

Dynamic dispatch chooses the right implementation, for **`someExpr.evaluate()`**

Overall type-checker spreads across classes

Procedural pattern



Create a class per operation, with a method per operand type:

```
class Evaluator {
   int evaluatePlusOp(PlusOp op) {
            ...
   }
   int evaluateMinusOp(MinusOp op) {
            ...
   }

   int evaluateUniaryOp(UnaryOp op) {
            ...
   }
   int evaluateValue(Value val) {
            ...
   }
}
```

How to invoke the right method for an expression `someExpr`?

Procedural pattern

Objects

CondExpr	EqualOp

```
class Evaluator {
...
 int evaluateExpression(Expression expr) {
   if (e instanceof PlusOp) return evaluatePlusOp((PlusOp)expr);
   else if (e instanceof MinusOp) return evaluateMinusOp((MinusOp)expr);
   else if (e instanceof UnaryOp) return evaluateUnaryOp((UnaryOp)expr);
   else if (e instanceof Value) return evaluateValue((Value)expr);
   else ...
  ...
}
}
```

- Maintaining this code is tedious and error-prone.
- The cascaded if tests are likely to run slowly.
- This code must be repeated in `asString` and

every other operation class (remember switch problem)

Visitor Pattern

- Visitor encodes a traversal of a hierarchical data structure
- Nodes (objects in the hierarchy - expressions) accept visitors
- Visitors visit nodes (objects)

```
class someExpression implements Expression {
   void accept(Visitor v) {
       for each child of this node {
              child.accept(v);
        }
       v.visit(this);
   }
}
class someVisitor implements Visitor{
   void visit(someExpression exp) {
       perform work on exp
   }

}
```

someExpr.accept(v) traverses the structure rooted at **exp**, performing **v**'s operation on each element of the structure

Example: accepting visitors

```
class Value implements Expression {
  …
  void accept(Visitor v) {
    v.visit(this);
  }
}
class UnaryOp implements Expression {
  …
  void accept(Visitor v) {
    singleExp.accept(v);
    v.visit(this);
  }

class PlusOp extends BinaryOp {
  …
  void accept(Visitor v) {
    leftExp.accept(v);
    rightExp.accept(v);
    v.visit(this);
  }
}
class MinusOp extends BinaryOp{
  …
  //same accept as in PlusOp
```

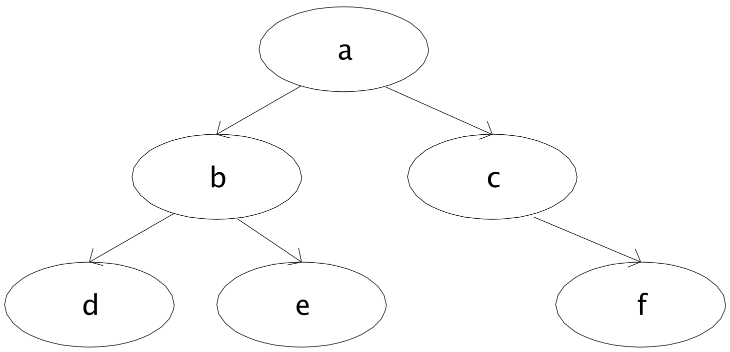
Traversing:
All children (components) should accept the visitor

Algorithm:
Let visitor do the job

- The visitor has a `visit` method for each kind of expression, thus picking the right code for this kind of expression
- Overloading makes this look more magical than it is…
- Clients can provide unexpected visitors

Sequence of calls to accept and visit

```
a.accept(v)
  b.accept(v)
    d.accept(v)
v.visit(d)
    e.accept(v)
      v.visit(e)
    v.visit(b)
  c.accept(v)
    f.accept(v)
      v.visit(f)
    v.visit(c)
  v.visit(a)
```



Sequence of calls to visit: d, e, b, f, c, a

Example: Implementing visitors

Overloading for ALL possible CONCRETE operations:

```
class EvaluatorVisitor implements Visitor {
    void visit(Value op) { … }
    void visit(UnaryOp op) { … }
    void visit(PlusOp op) { … }
    void visit(MinusOp op) { … }
}
class AsStringVisitor implements Visitor {
    void visit(Value op) { … }
    void visit(UnaryOp op) { … }
    void visit(PlusOp op) { … }
    void visit(MinusOp op) { … }
}
```

Why not to abstract out?

```
class PlusOp extends BinaryOp {
…
    void accept(Visitor v) {
        leftExp.accept(v);
        rightExp.accept(v);
        v.visit(this);
    }
}
```

```
class MinusOp extends BinaryOp {
…
        void accept(Visitor v) {
            leftExp.accept(v);
            rightExp.accept(v);
            v.visit(this);
        }
}
```

The `accept` in both classes is identical

→Why NOT to remove duplicate code and move it into abstract class?

```
abstract class BinaryOp implements Expression {
    …
    void accept(Visitor v) {
        leftExp.accept(v);
        rightExp.accept(v);
        v.visit(this);
    }
}`
```

You CANNOT abstract this code, because overloading in java is done using **STATIC binding (static polymorphism)** Static binding happens at compile time and uses **class to decide the type**

How will we get the result?

- Have a private field that accumulates the results and return it with getter
- Create generic Visitor interface with type parameter for return type visit:

```
public interface GenericVisitor<T>{
    T visit(Value op);
    T visit(Unary op);
…
}
class EvaluatorVisitor implements GenericVisitor<Integer> {
    Integer visit(Value op) {
        return op.getValue();
    }
    Integer visit(UnaryOp op) { … }
    Integer visit(PlusOp op) { … }
    Integer visit(MinusOp op) { … }
}
```

Finally

```
class Value implements Expression {
…
   int evaluate () {
       return accept(new EvaluatorVisitor());
    }
   <T> T accept(GenericVisitor<T> v) {
       return v.visit(this);
    }
}
```

Similar to static methods, you can define specific generic type for instance methods

Alternative Visitor Pattern

- Sometimes traversal is delegated to visitor:

```
class someExpression implements Expression {
   void accept(Visitor v) {
       v.visit(this);
   }
}
class someVisitor implements Visitor{
   void visit(someExpression exp) {
       for each child of this node (exp) {
              perform work on this child
       }
   }
}
```