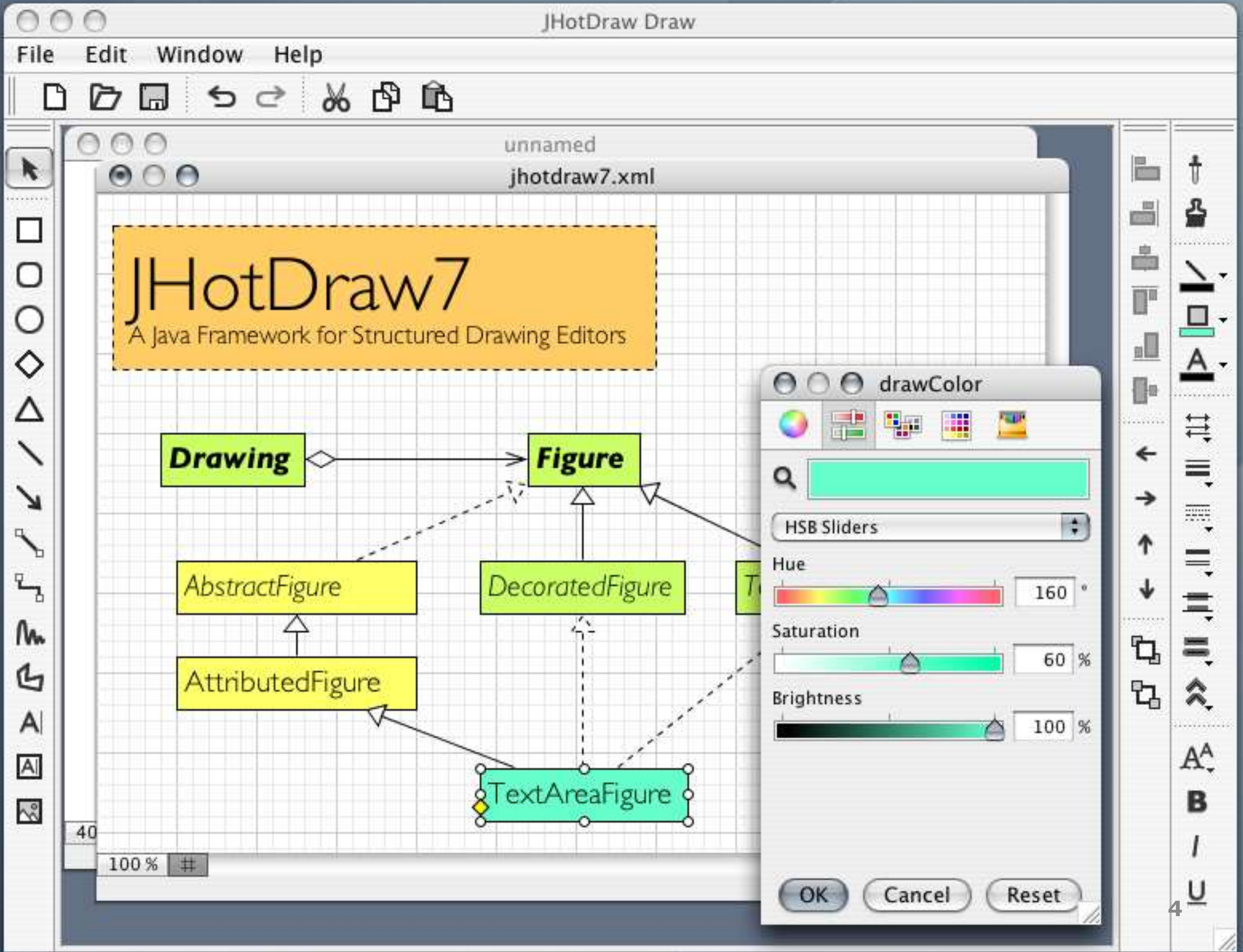
Tradeoffs?

```
void sort(int[] list, String order) {
   …
  boolean mustswap;
  if (order.equals("up")) {
    mustswap = list[i] < list[j];
  } else if (order.equals("down")) {
    mustswap = list[i] > list[j];
  }
  …
}
```

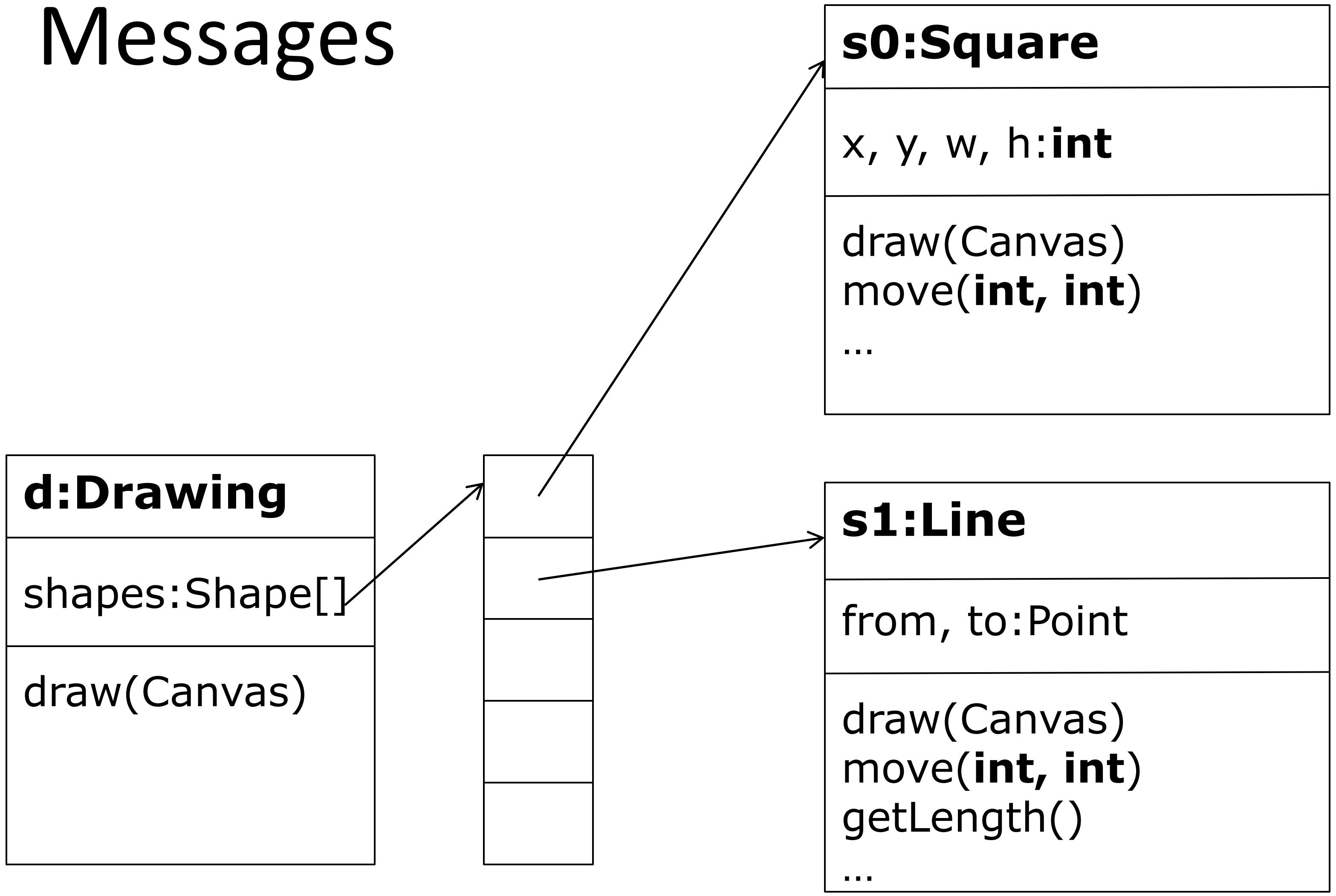
```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int I, int j) { return i<j; }}

class DownComparator implements Comparator {
  boolean compare(int I, int j) { return i>j; }}
```



HSB Sliders
Hue 160 °
Saturation 60 %
Brightness 100 %

Today: How Objects Respond to Messages



Learning Goals

- Explain the need to design for change and design for division of labor
- Understand subtype polymorphism and dynamic dispatch
- Distinguish between static and runtime type
- Understand basic language mechanisms of Java

Design Goals, Principles, and Patterns

- Design Goals
  - Design for Change
  - Design for Division of Labor
- Design Principles
  - Explicit Interfaces (clear boundaries)
  - Information Hiding (hide likely changes)
- Design Patterns
  - Strategy Design Pattern
  - Composite Design Pattern
- Supporting Language Features
  - Subtype Polymorphism
  - Encapuslation

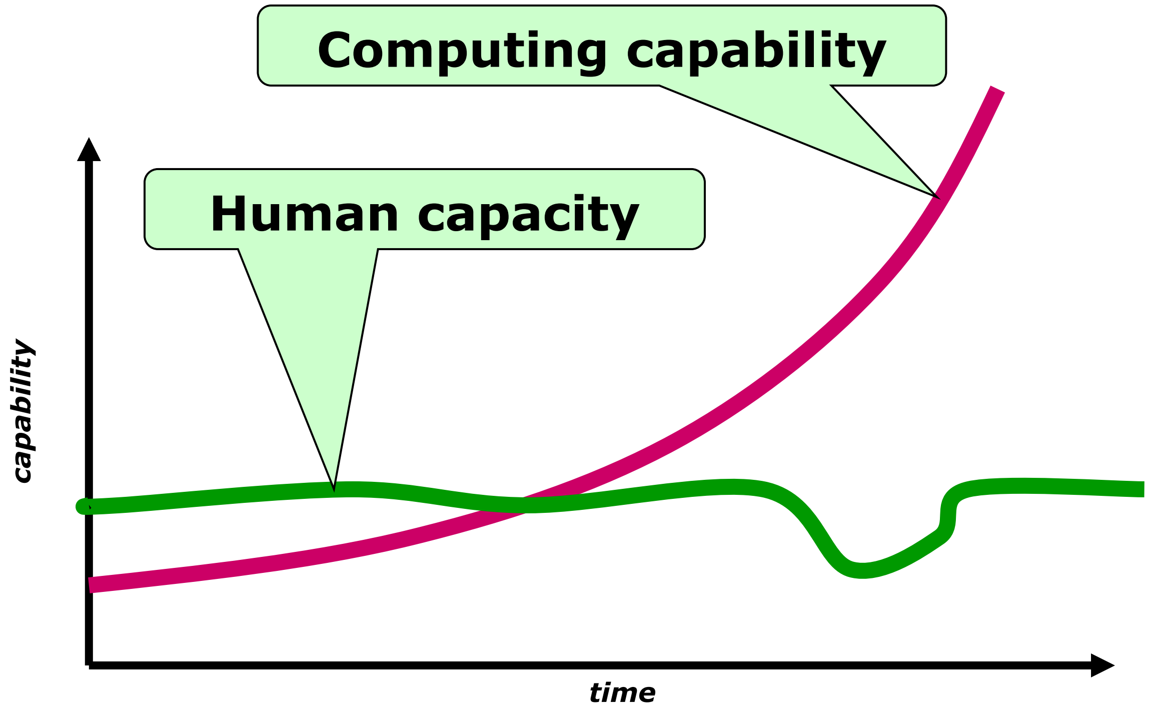
Software Change

- …accept the fact of change as a way of life, rather than an untoward and annoying exception.
  —Brooks, 1974

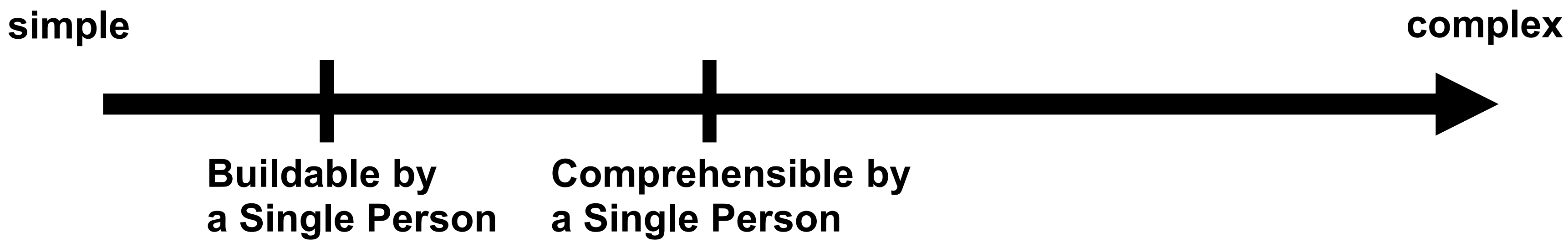
- Software that does not change becomes useless over time.
  —Belady and Lehman

- For successful software projects, most of the cost is spent evolving the system, not in initial development
  – Therefore, reducing the cost of change is one of the most important principles of software design

The limits of exponentials



Building Complex Systems


simple
complex
Buildable by a Single Person
Comprehensible by a Single Person

- Division of Labor
- Division of Knowledge and Design Effort
- Reuse of Existing Implementations

Design Goals for Today and Next Week

- **Design for Change** (flexibility, extensibility, modifiability)

also

- Design for Division of Labor
- Design for Understandability

SUBTYPE POLYMORPHISM / DYNAMIC DISPATCH (OBJECT-ORIENTED LANGUAGE FEATURE ENABLING FLEXIBILITY)

Objects

- A package of state (data) and behavior (actions)
- Can interact with objects by sending messages
  - perform an action (e.g., move)
  - request some information (e.g., getSize)

```
Point p = …
int x = p.getX();
```

```
IntSet a = …; IntSet b = …
boolean s = a.isSubsetOf(b);
```

- Possible messages described through an interface

```
interface Point {
    int getX();
    int getY();
    void moveUp(int y);
    Point copy();
}
```

```
interface IntSet {
    boolean contains(int element);
    boolean isSubsetOf(
             IntSet otherSet);
}
```

Subtype Polymorphism

- There may be multiple implementations of an interface
- Multiple implementations coexist in the same program
- May not even be distinguishable

- Every object has its own data and behavior

Creating Objects

```
interface Point {
   int getX();
   int getY();
}
Point p = new Point() {
   int getX() { return 3; }
   int getY() { return -10; }
}
```

Classes as Object Templates

```
interface Point {
    int getX();
    int getY();
}
class CartesianPoint implements Point {
    int x,y;
    Point(int x, int y) {this.x=x; this.y=y;}
    int getX() { return this.x; }
    int getY() { return this.y; }
}
Point p = new CartesianPoint(3, -10);
```

More Classes

```
interface Point {
    int getX();
    int getY();
}
class SkewedPoint implements Point {
    int x,y;
    SkewedPoint(int x, int y) {this.x=x + 10; this.y=y * 2;}
    int getX() { return this.x -  10; }
    int getY() { return this.y / 2; }
}
Point p = new SkewedPoint(3, -10);
```

Polar Points

```
interface Point {
    int getX();
    int getY();
}
class PolarPoint implements Point {
    double len, angle;
    PolarPoint(double len, double angle)
          {this.len=len; this.angle=angle;}
    int getX() { return this.len * cos(this.angle);}
    int getY() { return this.len * sin(this.angle); }
  double getAngle() {…}
}
Point p = new PolarPoint(5, .245);
```

Polar Points

```
interface Point {
     int getX();
     int getY();
}
```

```
interface PolarPoint {
   double getAngle() ;
   double getLength();
}
```

```
class PolarPointImpl implements Point, PolarPoint {
     double len, angle;
     PolarPoint(double len, double angle)
          {this.len=len; this.angle=angle;}
     int getX() { return this.len * cos(this.angle);}
     int getY() { return this.len * sin(this.angle); }
     double getAngle() {…}
     double getLength() {… }
}
PolarPoint p = new PolarPointImpl(5, .245);
Point q = new PolarPointImpl(5, .245);
```

Middle Points

```
interface Point {
    int getX();
    int getY();
}
class MiddlePoint implements Point {
    Point a, b;
    MiddlePoint(Point a, Point b) {this.a = a; this.b = b; }
    int getX() { return (this.a.getX() + this.b.getX()) / 2;}
    int getY() { return (this.a.getY() + this.b.getY()) / 2; }
}
Point p = new MiddlePoint(new PolarPoint(5, .245),
                          new CartesianPoint(3, 3));
```

Example: Points and Rectangles

Subtype Polymorphism

```
interface Point {
    int getX();
    int getY();
}
… = new Rectangle() {
    Point origin;
    int width, height;
    Point getOrigin() { return this.origin; }
    int getWidth() { return this.width; }
    void draw() {
        this.drawLine(this.origin.getX(), this.origin.getY(),      // first line
            this.origin.getX()+this.width, this.origin.getY());
        … // more lines here
    }
};
```

Points and Rectangles: Interface

```
interface Point {
    int getX();
    int getY();
}
interface Rectangle {
    Point getOrigin();
    int getWidth();
    int getHeight();
    void draw();
}
```

What are possible implementations of the IRectangle interface?

Discussion Subtype Polymorphism

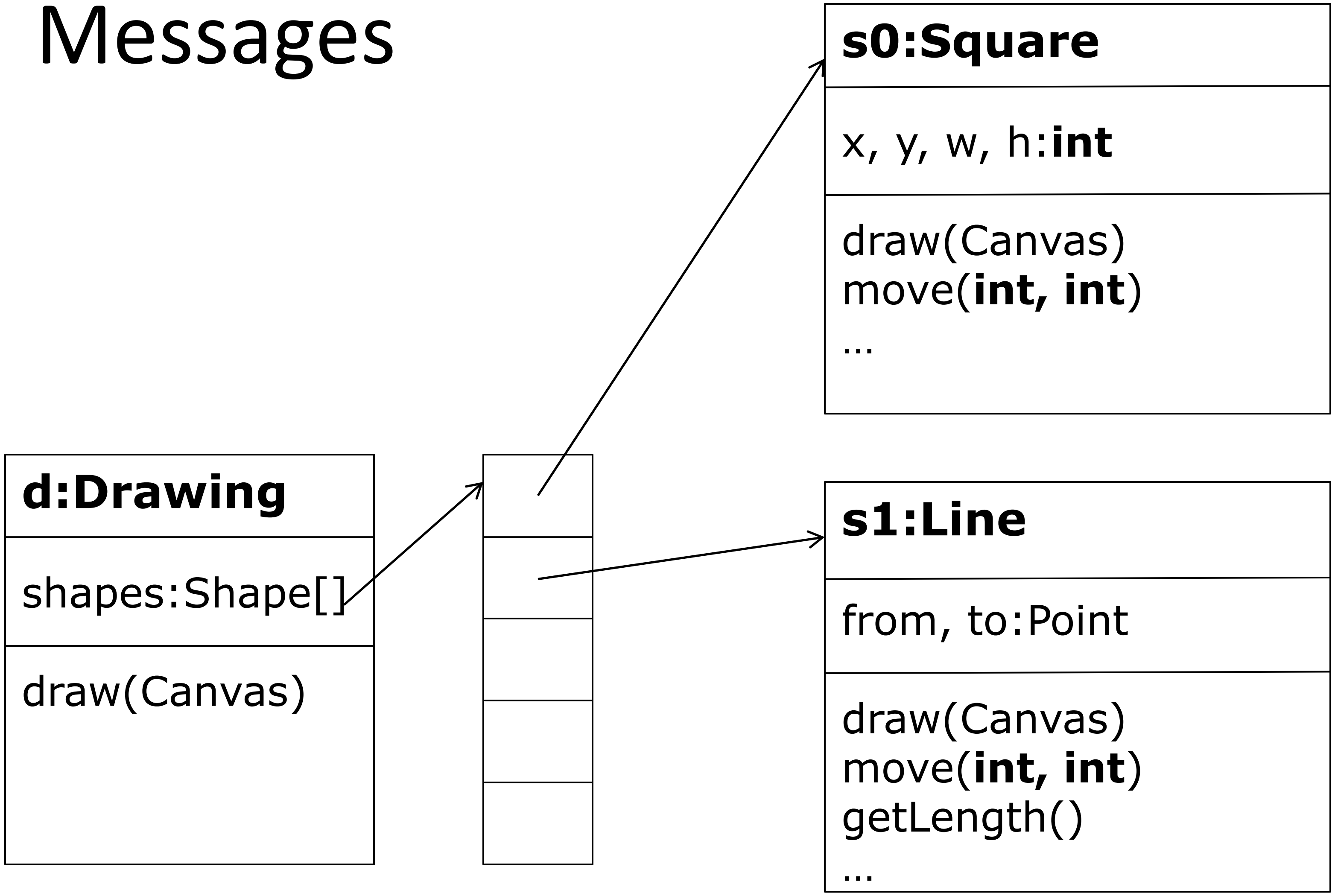
- A user of an object does not need to know the object's implementation, only its interface
- All objects implementing the interface can be used interchangeably
- Allows flexible **change** (modifications, extensions, reuse) later without changing the client implementation, even in unanticipated contexts

Design for Change!

Why multiple implementations?

- Different performance
  - Choose implementation that works best for your use
- Different behavior
  - Choose implementation that does what you want
  - Behavior *must* comply with interface spec (“contract”)
- Often performance and behavior *both* vary
  - Provides a functionality – performance tradeoff
  - Example: `HashSet`, `TreeSet`

Today: How Objects Respond to Messages



Check your Understanding

```
  interface Animal {
      void makeSound();
  }
  class Dog implements Animal {
      public void makeSound() { System.out.println("bark!"); }
  }
  class Cow implements Animal {
      public void makeSound() { mew(); }
      public void mew() {System.out.println("Mew!"); }
  }
0 Animal x = new Animal() {
          public void makeSound() { System.out.println("chirp!"); }}
   x.makeSound():
1 Animal a = new Animal();
2 a.makeSound();
3 Dog d = new Dog();
4 d.makeSound();
5 Animal b = new Cow();
6 b.makeSound();
7 b.mew();
```

- What happens?

Historical note: simulation and the origins of OO programming

- Simula 67 was the first object-oriented language
- Developed by Kristin Nygaard and Ole-Johan Dahl at the Norwegian Computing Center

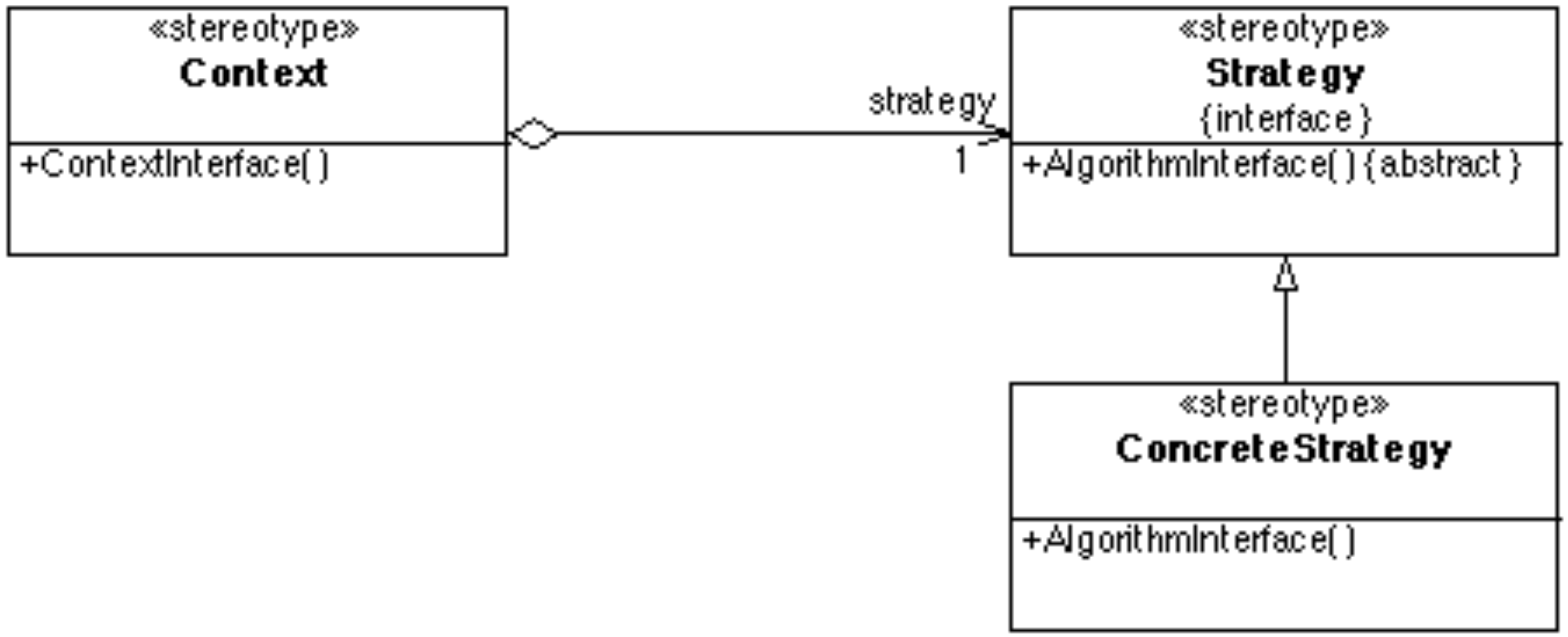
Dahl and Nygaard at the time of Simula's development

- Developed to support discrete-event simulation
  - Application: operations research, e.g. traffic analysis
  - Extensibility was a key quality attribute for them
  - Code reuse was another

also see UML and Patterns textbook 26.7

STRATEGY DESIGN PATTERN (EXPLOITING POLYMORPHISM FOR FLEXIBILITY)

Behavioral: Strategy



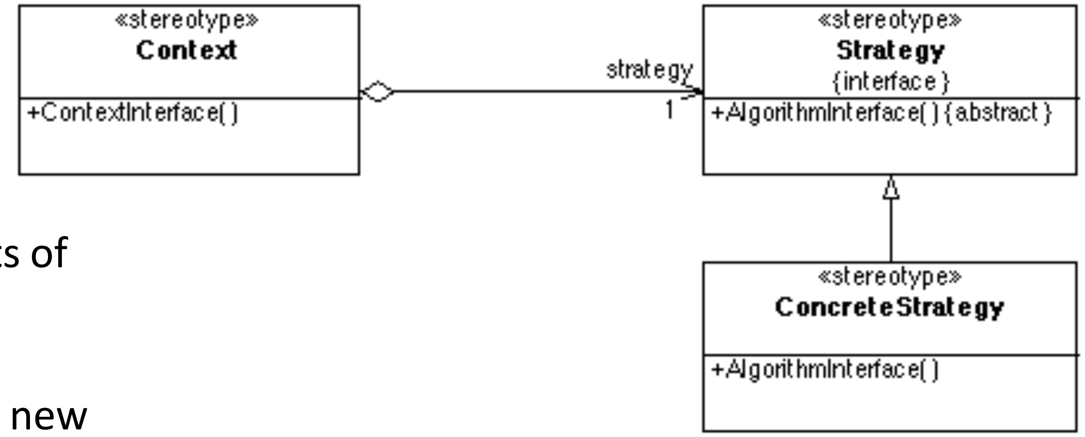
Tradeoffs

```
void sort(int[] list, String order) {
   …
  boolean mustswap;
  if (order.equals("up")) {
    mustswap = list[i] < list[j];
  } else if (order.equals("down")) {
    mustswap = list[i] > list[j];
  }
  …
}
```

```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int I, int j) { return i<j; }}

class DownComparator implements Comparator {
  boolean compare(int I, int j) { return i>j; }}
```

Behavioral: Strategy



- Applicability
  - Many classes differ in only their behavior
  - Client needs different variants of an algorithm
- Consequences
  - Code is more extensible with new strategies
    - compare to conditionals
  - Separates algorithm from context
    - each can vary independently
    - design for change and reuse; reduce coupling
  - Adds objects and dynamism
    - code harder to understand
  - Common strategy interface
    - may not be needed for all Strategy implementations – may be extra overhead

- Design for change
  - Find what varies and encapsulate it
  - Allows changing/adding alternative variations later
  - Class *Context* closed for modification, but open for extension
- Equivalent in functional progr. languages: Higher-order functions

Design Patterns

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”
  – Christopher Alexander

- Every Strategy interface has its own domain-specific interface
  - But they share a common problem and solution

Examples

- Change the sorting criteria in a list
- Change the aggregation method for computations over a list (e.g., fold)
- Compute the tax on a sale
- Compute a discount on a sale
- Change the layout of a form

Benefits of Patterns

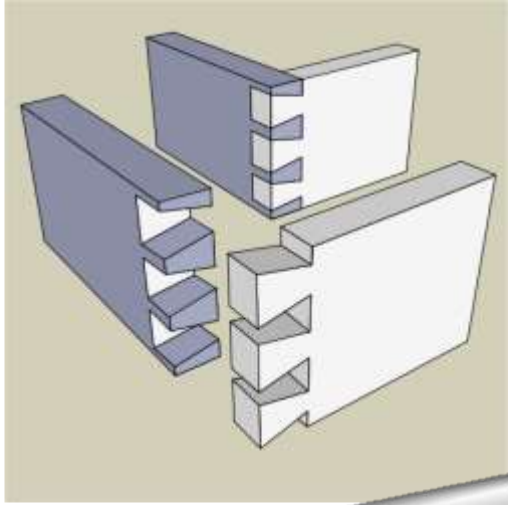
- Shared language of design
  - Increases communication bandwidth
  - Decreases misunderstandings

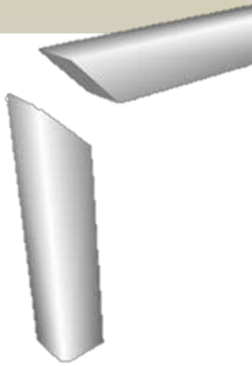
- Learn from experience
  - Becoming a good designer is hard
    - Understanding good designs is a first step
  - Tested solutions to common problems
    - Where is the solution applicable?
    - What are the tradeoffs?

Illustration [Shalloway and Trott]

- Carpenter 1: How do you think we should build these drawers?
- Carpenter 2: Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating…
- SE example: “I wrote this if statement to handle … followed by a while loop … with a break statement so that…”

A Better Way



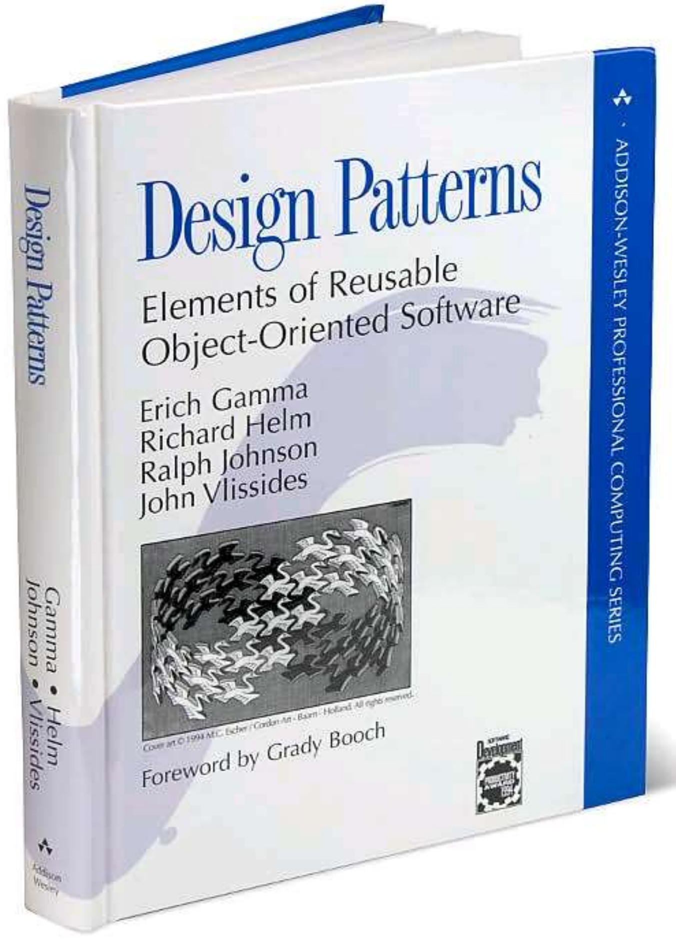


- Carpenter 1: Should we use a dovetail joint or a miter joint?
- Subtext:
  - miter joint: cheap, invisible, breaks easily
  - dovetail joint: expensive, beautiful, durable
- Shared terminology and knowledge of consequences raises level of abstraction
  - CS: Should we use a Strategy?
  - Subtext
    - Is there a varying part in a stable context?
    - Might there be advantages in limiting the number of possible implementations?

Elements of a Pattern

- Name
  - Important because it becomes part of a design vocabulary
  - Raises level of communication
- Problem
  - When the pattern is applicable
- Solution
  - Design elements and their relationships
  - Abstract: must be specialized
- Consequences
  - Tradeoffs of applying the pattern
    - Each pattern has costs as well as benefits
    - Issues include flexibility, extensibility, etc.
    - There may be variations in the pattern with different consequences

History: Design Patterns Book

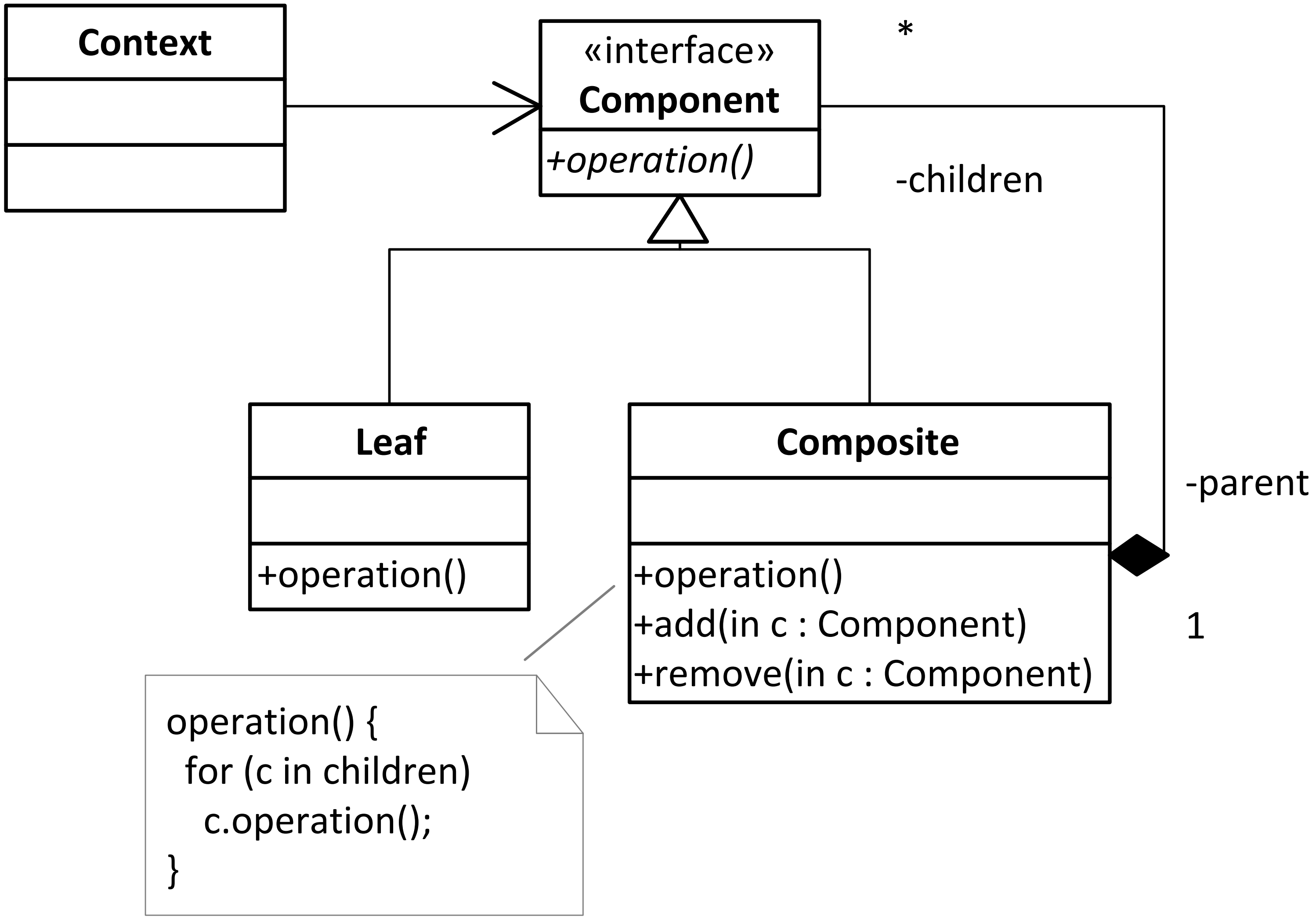


- Brought Design Patterns into the mainstream
- Authors known as the Gang of Four (GoF)
- Focuses on *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*
- Great as a reference text
- Uses C++, Smalltalk

Design Exercise (on paper)

- You are designing software for a shipping company.
- There are several different kinds of items that can be shipped: letters, books, packages, fragile items, etc.
- Two important considerations are the **weight** of an item and its **insurance cost**.
  - Fragile items cost more to insure.
  - All letters are assumed to weigh an ounce
  - We must keep track of the weight of other packages.
- The company sells **boxes** and customers can put several items into them.
  - The software needs to track the contents of a box (e.g. to add up its weight, or compute the total insurance value).
  - However, most of the software should treat a box holding several items just like a single item.
- Think about how to represent packages; what are possible interfaces, classes, and methods? (letter, book, box only)

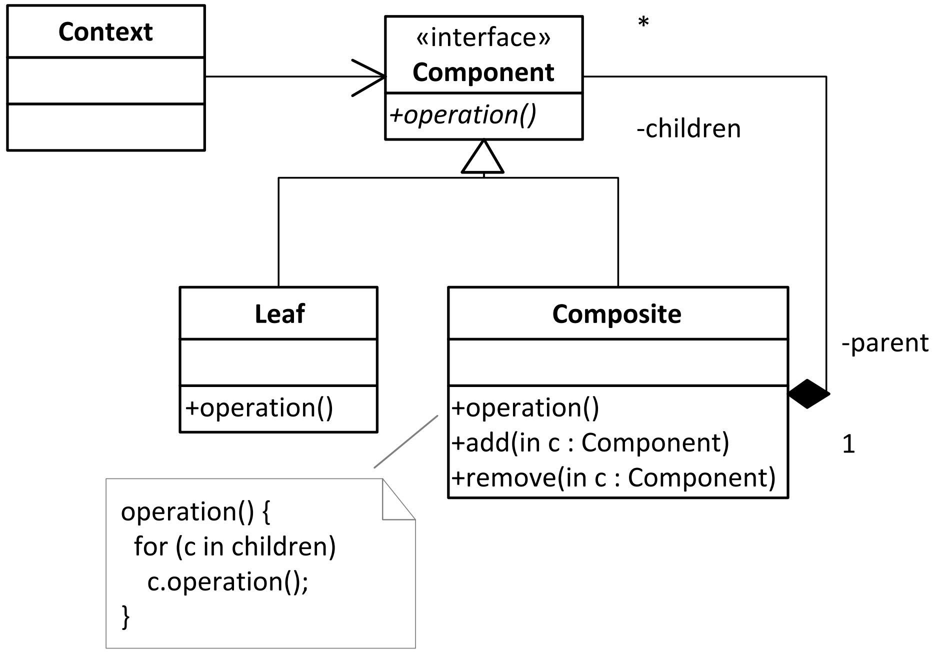
The Composite Design Pattern



```
operation() {
  for (c in children)
    c.operation();
}
```

The Composite Design Pattern

- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components



```
operation() {
  for (c in children)
    c.operation();
}
```

We have see this before

```
interface Point {
        int getX();
        int getY();
}
class MiddlePoint implements Point {
        Point a, b;
        MiddlePoint(Point a, Point b) {this.a = a; this.b = b; }
        int getX() { return (this.a.getX() + this.b.getX()) / 2;}
        int getY() { return (this.a.getY() + this.b.getY()) / 2; }
}
```