Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing Classes
Design for Change

Josh Bloch Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due Tuesday 1/19, 11:59 p.m. EST
- Homework 2 due Tuesday 1/26, 11:59 p.m. EST

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

Outline

Metrics of software quality

Source: Braude, Bernstein, Software Engineering. Wiley 2011

- Sufficiency / functional correctness
  - Fails to implement the specifications … Satisfies all of the specifications
- Robustness
  - Will crash on any anomalous even … Recovers from all anomalous events
- Flexibility
  - Will have to be replaced entirely if specification changes … Easily adaptable to reasonable changes
- Reusability
  - Cannot be used in another application … Usable in all reasonably related apps without modification
- Efficiency
  - Fails to satisfy speed or data storage requirement … satisfies requirement with reasonable margin
- Scalability
  - Cannot be used as the basis of a larger version … is an outstanding basis…
- Security
  - Security not accounted for at all … No manner of breaching security is known

Design challenges/goals

Source: BC Forestry website

A design case study: Simulating an invasive species

Lodgepole Pine

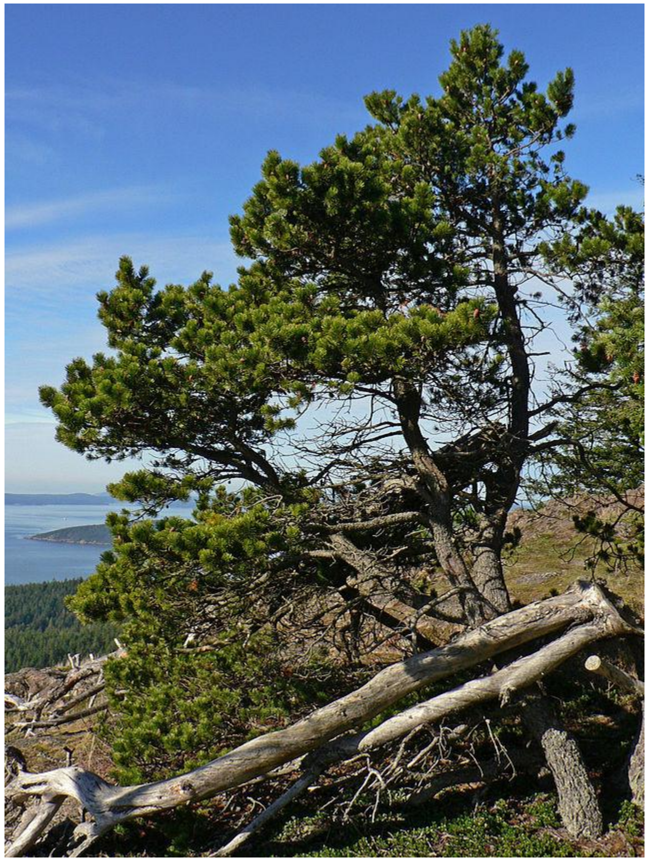


Photo by Walter Siegmund

Mountain Pine Beetle

Galleries carved in inner bark

Widespread tree death

Further reading: Liliana Péreza and Suzana Dragićević. Exploring Forest Management Practices Using an Agent-Based Model of Forest Insect Infestations. International Congress on Environmental Modelling and Software Modelling for Environment's Sake, 2010.

Preview: Design goals, principles, and patterns

- ***Design goals*** enable evaluation of designs
  - e.g. maintainability, reusability, scalability
- ***Design principles*** are heuristics that describe best practices
  - e.g. high correspondence to real-world concepts
- ***Design patterns*** codify repeated experiences, common solutions
  - e.g. template method pattern

Preview: The design process

- Object-oriented analysis
  - Understand the problem
- Object-oriented design
  - Cheaply create and evaluate plausible alternatives
- Implementation
  - Convert design to code

Outline

I. Key concepts from Tuesday
II. Object-Oriented programming basics in Java
III. Information hiding
IV. Design patterns – the Strategy pattern

Objects

- An **object** is a bundle of state and behavior
- State – the data contained in the object
  - In Java, these are the **fields** of the object
- Behavior – the actions supported by the object
  - In Java, these are called **methods**
  - Method is just OO-speak for function
  - invoke a method = call a function

Classes

- Every object has a class
  - A class defines methods and fields
  - Methods and fields are collectively known as **members**
- Class defines both type and implementation
  - type ≈ where the object can be used
  - implementation ≈ how the object does things
- Loosely speaking, the methods of a class are its **Application Programming Interface (API)**
  - Defines how users interact with instances of the class

Class example – complex numbers

```
class Complex {
  double re;  // Real Part
  double im;  // Imaginary Part

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Class usage example

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new Complex(-1, 0);
        Complex d = new Complex(0, 1);

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it prints

```
-1.0 + 1.0i
-0.0 + -1.0i
```

Interfaces and implementations

- Multiple implementations of API can coexist
  - Multiple classes can implement the same API
  - They can differ in performance and behavior
- In Java, an API is specified by *interface* or *class*
  - Interface provides only an API
  - Class provides an API and an implementation
  - A Class can implement multiple interfaces

An interface to go with our class

```
public interface Complex {
    // No constructors, fields, or implementations!
    double realPart();
    double imaginaryPart();
    double r();
    double theta();

    Complex plus(Complex c);
    Complex minus(Complex c);
    Complex times(Complex c);
    Complex dividedBy(Complex c);
}
```

An interface defines but does not implement an API

Modifying class to use interface

```
class OrdinaryComplex implements Complex {
  double re;  // Real Part
  double im;  // Imaginary Part

  public OrdinaryComplex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Modifying client to use interface

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new OrdinaryComplex(-1, 0);
        Complex d = new OrdinaryComplex(0, 1);

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it still prints

```
-1.0 + 1.0i
-0.0 + -1.0i
```

Interface permits multiple implementations

```
class PolarComplex implements Complex {
  double r;
  double theta;

  public PolarComplex(double r, double theta) {
    this.r = r;
    this.theta = theta;
  }

  public double realPart()      { return r * Math.cos(theta) ; }
  public double imaginaryPart() { return r * Math.sin(theta) ; }
  public double r()             { return r; }
  public double theta()         { return theta; }

  public Complex plus(Complex c)      { ... } // Completely different impls
  public Complex minus(Complex c)     { ... }
  public Complex times(Complex c)     { ... }
  public Complex dividedBy(Complex c) { ... }
}
```

Interface decouples client from implementation

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new PolarComplex(Math.PI,   1);  // -1 in polar form
        Complex d = new PolarComplex(Math.PI/2, 1);  //  i in polar form

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                                   + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                                   + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it STILL prints

```
-1.0 + 1.0i
-0.0 + -1.0i
```

Why multiple implementations?

- Different performance
  - Choose the implemenation that works best for your use
- Different behavior
  - Choose the 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`

Fancy name: *subtype polymorphism*

- Object's user doesn't need to know implementation, only interface
- Objects implementing interface can be used interchangeably
  - Behavior may vary as indicated in implementation specs
  - But only within limits: implementation specs can *refine* interface specs
- Using interfaces as types makes it easy to substitute new implementation for improved behavior or performance
- In other words, interfaces facilitate change!

Classes as types

- Classes *are* usable as types…
  - Public methods in classes usable like methods in interfaces
  - Public fields directly accessible from other classes
- But prefer programming to interfaces
  - Use interface types, not class types, for variables and parameters
  - Unless you *know* that only one implementation will ever make sense
  - Supports change of implementation
  - Prevents client from depending on implementation details

```
Complex c = new OrdinaryComplex(42, 0);          // Do this…
OrdinaryComplex c = new OrdinaryComplex(42, 0);  // Not this
```

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() { moo(); }
    public void moo() {System.out.println("Moo!"); }
}
```

What Happens?

1. `Animal a = new Animal();`
   `a.makeSound();`
2. `Dog d = new Dog();`
   `d.makeSound();`
3. `Animal b = new Cow();`
   `b.makeSound();`
4. `b.moo();`

Historical note: Simulation and the origins of objects

Dahl and Nygaard at the time of Simula's development

- Simula 67 was the first object-oriented programming language
- Developed by Kristin Nygaard and Ole-Johan Dahl at the Norwegian Computing Center
- Developed to support discrete-event simulations
  - Much like our tree beetle simulation
  - Application: operations research, e.g. for traffic analysis
  - Extensibility was a key quality attribute for them
  - Code reuse was another

Outline

I. Key concepts from Tuesday
II. Object-Oriented programming basics in Java
III. Information hiding
IV. Design patterns – the Strategy pattern

Information hiding

- **The single most important factor that distinguishes a well-designed module from a bad one is the degree to which it hides its internal data and other implementation details from other modules**
- Well-designed modules hide *all* implementation details
  - Cleanly separates API from implementation.
  - Modules communicate *only* through APIs
  - The are oblivious to each others' inner workings.
- Concept known as *information hiding* or *encapsulation*
- Fundamental tenet of software design [David Parnas, '72]

Benefits of information hiding

- **Decouples** the classes that comprise a system
  - Allows them to be developed, tested, optimized, used, understood, and modified in isolation
- **Speeds up system development**
  - Classes can be developed in parallel
- **Eases burden of maintenance**
  - Classes can be understood more quickly an debugged with little fear of harming other modules.
- **Enables effective performance tuning**
  - "Hot" classes can be optimized in isolation
- **Increases software reuse**
  - Loosely-coupled classes often prove useful in other contexts

Information hiding with interfaces

- Define an interface
- Client uses only interface methods
- Fields not accessible from client code
- But this only takes us so far
  - Client can access non-interface class members directly
  - In essence, it's voluntary information hiding

Java provides *visibility modifiers*

- **private**—Member accessible *only* from declaring class
- **package-private**—Member accessible from any class in package where it is declared
  - Technically known as default access
  - You get this if no access modifier is specified
- **protected**—Member accessible from subclasses of declaring class and from any class in its package
- **public**—Member is accessible from anywhere

Hiding interior state in OrdinaryComplex

```
class OrdinaryComplex implements Complex {
  private double re;  // Real Part
  private double im;  // Imaginary Part

  public OrdinaryComplex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Discussion

- You know the benefits of private fields
- What are the benefits of private methods?

Best practices for information hiding

- Carefully design your public API
- Provide *only* the functionality required by clients
  - *All* other members should be private
- You can always make a private member public later without breaking clients
  - But not vice-versa!

Micro-scale information hiding

- How could we better hide information here?

```
class Utilities {
    private int total;
    public int sum(int[] data) {
        total = 0;
        for (int i = 0; i < data.length; ++i) {
            total += data[i];
        }
        return total
    }
    // other methods…
}
```

- Should be a local variable of the sum method
- This would hide part of the implementation of sum from the rest of the class!

Micro-scale information hiding

- Better (but still not terribly good)

```
class Utilities {
    public int sum(int[] data) {
        int total = 0;
        for (int i = 0; i < data.length; ++i) {
            total += data[i];
        }
        return total;
    }
    // other methods…
}
```

A Piazza question

- **Should I add a getter/setter for every private field?**
  - What do you think?

A Piazza question

- **Should I add a getter/setter for every private field?**
  - What do you think?

- Information hiding suggests
  - A getter *only* when clients need the information
  - A setter only when clients need to mutate the data
    - Avoid where possible!
    - We'll discuss this in another lecture.
  - Methods with signatures at the right level of abstraction

An infamous design problem

```
// Represents a Java class
public class Class {
       //  Entities that have digitally signed this class
       //  Use the class only if you trust a signer
       private Object[] signers;

       // what getters/setters should we provide?



}
```

An actual* security bug

```
// Represents a Java class
public class Class {
        // Entities that have digitally signed this class
        // so use the only class if you trust a signer
        private Object[] signers;

        // Get the signers of this class
        // VULNERABILITY: clients can change
        // the signers of a class
        public Object[] getSigners() { return signers; }
}
```

*simplified slightly for presentation, but a real Java bug (now fixed)

Outline

I. Key concepts from Tuesday
II. Object-Oriented programming basics in Java
III. Information hiding
IV. Design patterns – the Strategy pattern

Remember this slide from Tuesday?

```
void sort(int[] list, boolean ascending) {
   …
   boolean mustSwap;
   if (ascending) {
      mustSwap = list[i] < list[j];
   } else {
      mustSwap = list[i] > list[j];
   }
   …
}
```

```
interface Comparator {
  boolean compare(int i, int j);
}
final Comparator ASCENDING =  (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

void sort(int[] list, Comparator cmp) {
  …
  boolean mustSwap =
    cmp.compare(list[i], list[j]);
  …
}
```

It uses the Strategy design pattern

- Pass in a “behavior” object to parameterize the behavior of another object
- The behavior object is know as the strategy
  - The Comparator is a strategy for sorting
- General technique for writing flexible objects
- Enables code reuse

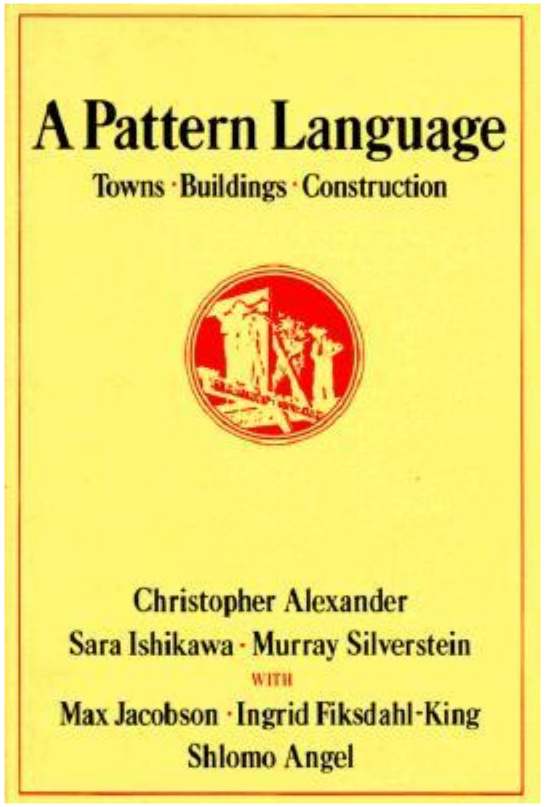
Strategy interfaces

- Every Strategy interface
  - Has its own domain-specific interface
  - Shares a common problem and solution
- Examples
  - Change the sorting criteria in a list
  - Compute the tax on a sale
  - Compute a discount on a sale
  - C’s printf statement(!)

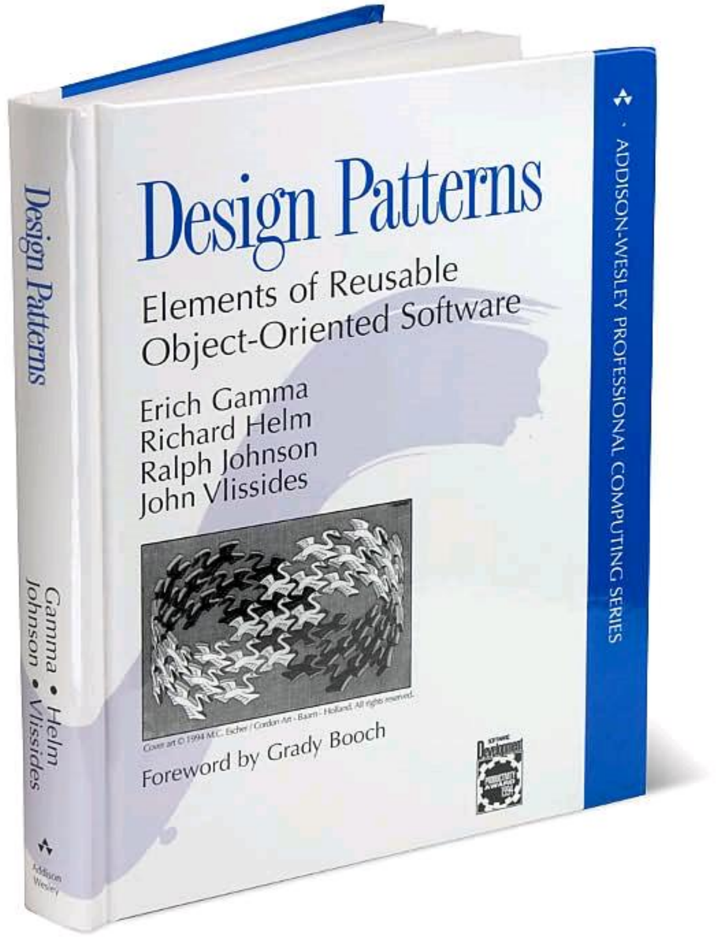
What is a design pattern?

"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, Architect (1977)

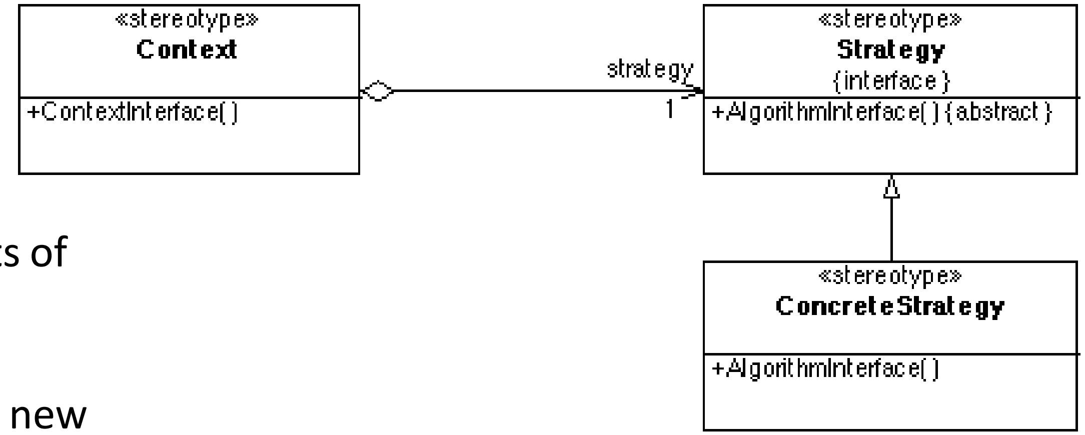

A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

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
- Published 1994

Design pattern: 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
  - But a Strategy interface may include more than one function

Benefits of design patterns

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

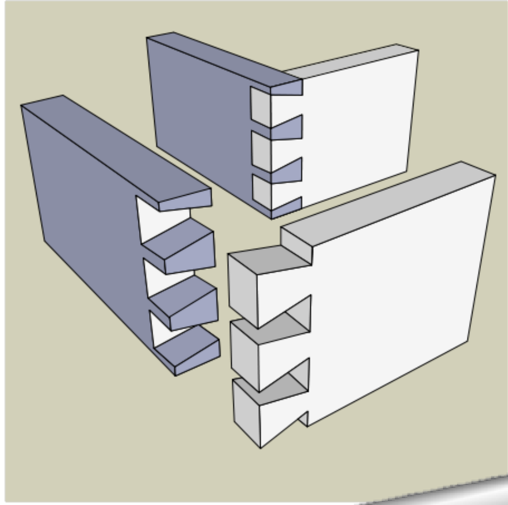
- Learn from experience
  - Becoming a good designer is hard
    - Understanding good designs is a first step
  - Tested solutions to common problems
    - Where is the solution applicable?
    - What are the tradeoffs?

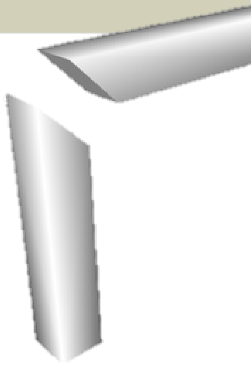
Illustration [Shalloway and Trott]

How not to discuss design

- Carpentry:
  - How do you think we should build these drawers?
  - 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…
- Software Engineering:
  - How do you think we should write this method?
  - “I think we should write this if statement to handle … followed by a while loop … with a break statement so that…”

A better way





- Carpentry
  - Should we use a dovetail joint or a miter joint?
  - Subtext
    - Miter joint: cheap, invisible, breaks easily
    - Dovetail joint: expensive, beautiful, durable

- Software Engineering:
  - 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?

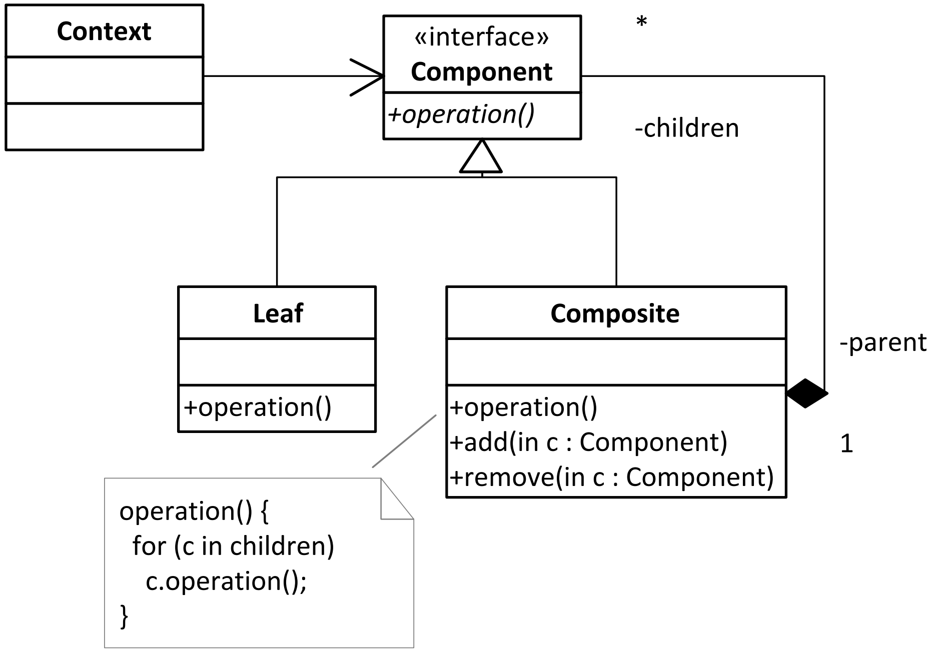
- **Shared terminology and knowledge of consequences raises level of abstraction enables good decision making**

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

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



What we learned

- We must design our software for change
  - It will happen whether or not we design for it
- Java provides classes, interfaces, methods, and fields
- Interfaces-based designs handle change well
- Information hiding is crucial to good design
- Design patterns help us design software