# Principles of Software Construction: Objects, Design, and Concurrency

## Part 1: Designing classes

## Inheritance (continued) and introduction to design patterns

Josh Bloch **Charlie Garrod**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

# Administrivia

- Homework 1 feedback in your GitHub repository
- Homework 2 due tonight 11:59 p.m.
- Homework 3 available tomorrow
- Optional reading due today:  Effective Java Items 18, 19, and 20
  - Required reading due next Tuesday:  UML & Patterns Ch 9 and 10

# Key concepts from Tuesday

# Behavioral subtyping

> Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.
>
> Barbara Liskov

- e.g., Compiler-enforced rules in Java:
  - Subtypes can add, but not remove methods
  - Concrete class must implement all undefined methods
  - Overriding method must return same type or subtype
  - Overriding method must accept the same parameter types
  - Overriding method may not throw additional exceptions
- Also applies to specified behavior. Subtypes must have:
  - Same or stronger invariants
  - Same or stronger postconditions for all methods
  - Same or weaker preconditions for all methods

This is called the *Liskov Substitution Principle*.

# This Square is *not* a behavioral subtype of Rectangle

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

   //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
   //@ requires neww > 0;
   //@ ensures w==neww
              && h==old.h;
    void setWidth(int neww) {
        w=neww;
    }
}
```

```
class Square extends Rectangle {
   //@ invariant h>0 && w>0;
   //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }

   //@ requires neww > 0;
   //@ ensures w==neww
              && h==neww;
   @Override
   void setWidth(int neww) {
      w=neww;
      h=neww;
   }
}
```

# Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
  - e.g. here, the `Sorter` is delegating functionality to some `Order`
- Judicious delegation enables code reuse
  - `Sorter` can be reused with arbitrary sort orders
  - `Orders` can be reused with arbitrary client code that needs to compare integers

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

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

# Today

- Inheritance
  - Design for reuse: delegation vs inheritance
- UML class diagrams
- Introduction to design patterns
  - Strategy pattern
  - Command pattern
- Design patterns for reuse:
  - Template method pattern
  - Iterator pattern (next week)
  - Decorator pattern (next week)

# Consider: types of bank accounts

```
public interface CheckingAccount {
    public long getBalance();
    public void  deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account??? target);
    public long getFee();
}
```

```
public interface SavingsAccount {
    public long getBalance();
    public void  deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account??? target);
    public double getInterestRate();
}
```

# Interface inheritance for an account type hierarchy

```
public interface Account {
    public long getBalance();
    public void  deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account target);
    public void monthlyAdjustment();
}

public interface CheckingAccount extends Account {
    public long getFee();
}

public interface SavingsAccount extends Account {
    public double getInterestRate();
}

public interface InterestCheckingAccount
                        extends CheckingAccount, SavingsAccount {
}
```

# The power of object-oriented interfaces

- Subtype polymorphism
  - Different kinds of objects can be treated uniformly by client code
  - Each object behaves according to its type
    - e.g., if you add new kind of account, client code does not change:

```
If today is the last day of the month:
   For each acct in allAccounts:
      acct.monthlyAdjustment();
```

# Implementation inheritance for code reuse

```
public abstract class AbstractAccount
        implements Account {
   protected long balance = 0;
   public long getBalance() {
        return balance;
   }
   abstract public void monthlyAdjustment();
   // other methods…
}

public class CheckingAccountImpl
        extends AbstractAccount
        implements CheckingAccount {
   public void monthlyAdjustment() {
        balance -= getFee();
   }
   public long getFee() { … }
}
```

# Implementation inheritance for code reuse

```
public abstract class AbstractAccount
        implements Account {
   protected long balance = 0;
   public long getBalance() {
        return balance;
   }
   abstract public void monthlyAdjustment();
   // other methods…
}

public class CheckingAccountImpl
        extends AbstractAccount
        implements CheckingAccount {
   public void monthlyAdjustment() {
        balance -= getFee();
   }
   public long getFee() { … }
}
```

an abstract class is missing the implementation of one or more methods

protected elements are visible in subclasses

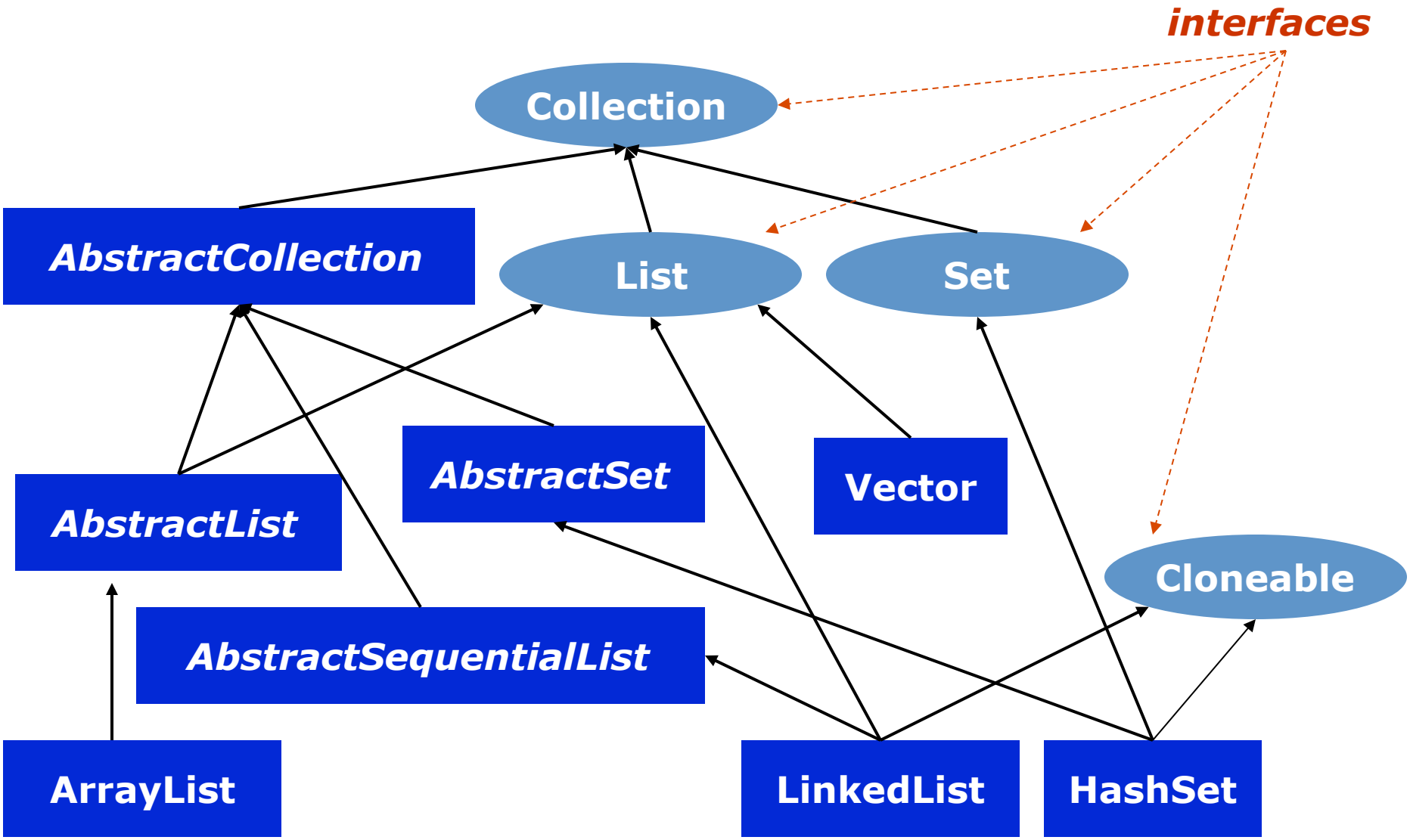
an abstract method is left to be implemented in a subclass

no need to define getBalance() – the code is inherited from AbstractAccount

# Inheritance: a glimpse at the hierarchy

- Examples from Java
  - `java.lang.Object`
  - Collections library

# Java Collections API (excerpt)



# The abstract `java.util.AbstractList<E>`

```
abstract E   get(int i);
abstract int size();
boolean      set(int i, E e);          // pseudo-abstract
boolean      add(E e);                 // pseudo-abstract
boolean      remove(E e);              // pseudo-abstract
boolean      addAll(Collection<? extends E> c);
boolean      removeAll(Collection<?> c);
boolean      retainAll(Collection<?> c);
boolean      contains(E e);
boolean      containsAll(Collection<?> c);
void         clear();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
<T> T[]      toArray(T[] a);
…
```

# Using java.util.AbstractList<E>

```
public class ReversedList<E> extends java.util.AbstractList<E>
             implements java.util.List<E> {
    private final List<E> list;

    public ReversedList(List<E> list) {
        this.list = list;
    }

    @Override
    public int size() {
        return list.size();
    }

    @Override
    public E get(int index) {
        return list.get(size() - index - 1);
    }
}
```

# Benefits of inheritance

- Reuse of code
- Modeling flexibility

# Inheritance and subtyping

- Subtyping is for polymorphism
  - Accessing objects the same way, but getting different behavior
  - Subtype is substitutable for supertype

```
class A implements B
class A extends B
```

- Inheritance is for polymorphism and code reuse
  - Write code once and only once
  - Superclass features implicitly available in subclass

```
class A extends B
```

# Typical roles for interfaces and classes

- An interface defines expectations / commitments for clients
- A class fulfills the expectations of an interface
  - An abstract class is a convenient hybrid
  - A subclass specializes a class's implementation

# Java details: extended reuse with super

```
public abstract class AbstractAccount implements Account {
   protected long balance = 0;
   public boolean withdraw(long amount) {
       // withdraws money from account (code not shown)
   }
}

public class ExpensiveCheckingAccountImpl
      extends AbstractAccount implements CheckingAccount {
   public boolean withdraw(long amount) {
      balance -= HUGE_ATM_FEE;
      boolean success = super.withdraw(amount)
      if (!success)
         balance += HUGE_ATM_FEE;
      return success;
   }
}
```

Overrides `withdraw` but also uses the superclass `withdraw` method

# Java details: constructors with `this` and `super`

```
public class CheckingAccountImpl
      extends AbstractAccount implements CheckingAccount {

   private long fee;

   public CheckingAccountImpl(long initialBalance, long fee) {
      super(initialBalance);
      this.fee = fee;
   }

   public CheckingAccountImpl(long initialBalance) {
      this(initialBalance, 500);
   }
   /* other methods… */ }
```

Invokes a constructor of the superclass. Must be the first statement of the constructor.

Invokes another constructor in this same class

# Java details: `final`

- A final field: prevents reassignment to the field after initialization
- A final method: prevents overriding the method
- A final class: prevents extending the class
  - e.g., `public final class CheckingAccountImpl { …`

# Note: type-casting in Java

- Sometimes you want a different type than you have
  - e.g.,
```
double pi = 3.14;
int indianaPi = (int) pi;
```
- Useful if you know you have a more specific subtype:
  - e.g.,
```
Account acct = …;
CheckingAccount checkingAcct =
                        (CheckingAccount) acct;
long fee = checkingAcct.getFee();
```
  - Will get a `ClassCastException` if types are incompatible
- Advice: avoid downcasting types
  - Never(?) downcast within superclass to a subclass

# An aside: `instanceof`

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {
    long adj = 0;
    if (acct instanceof CheckingAccount) {
        checkingAcct = (CheckingAccount) acct;
         adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        savingsAcct = (SavingsAccount) acct;
         adj = savingsAcct.getInterest();
    }
    …
}
```

**Do not do this. This code is bad.**

- Advice: avoid `instanceof` if possible
  - Never(?) use `instanceof` in a superclass to check type against subclass

# An aside: `instanceof`

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {
    long adj = 0;
    if (acct instanceof CheckingAccount) {
        checkingAcct = (CheckingAccount) acct;
         adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        savingsAcct = (SavingsAccount) acct;
         adj = savingsAcct.getInterest();
    } else if (acct instanceof InterestCheckingAccount) {
        icAccount = (InterestCheckingAccount) acct;
        adj = icAccount.getInterest();
        adj -= icAccount.getFee();
    }
    …
}
```

**Do not do this. This code is bad.**

# Java details: Dynamic method dispatch

1. (Compile time) Determine which class to look in
2. (Compile time) Determine method signature to be executed
    1. Find all accessible, applicable methods
    2. Select most specific matching method

# Java details: Dynamic method dispatch

1. (Compile time) Determine which class to look in
2. (Compile time) Determine method signature to be executed
    1. Find all accessible, applicable methods
    2. Select most specific matching method
3. (Run time) Determine dynamic class of the receiver
4. (Run time) From dynamic class, determine method to invoke
    1. Execute method with the same signature found in step 2 (from dynamic class or one of its supertypes)

# Use polymorphism to avoid `instanceof`

```
public interface Account {
    …
    public long getMonthlyAdjustment();
}

public class CheckingAccount implements Account {
    …
    public long getMonthlyAdjustment() {
        return getFee();
    }
}

public class SavingsAccount implements Account {
    …
    public long getMonthlyAdjustment() {
        return getInterest();
    }
}
```

# Use polymorphism to avoid `instanceof`

```
public void doSomething(Account acct) {
  long adj = 0;
  if (acct instanceof CheckingAccount) {
    checkingAcct = (CheckingAccount) acct;
    adj = checkingAcct.getFee();
  } else if (acct instanceof SavingsAccount) {
    savingsAcct = (SavingsAccount) acct;
    adj = savingsAcct.getInterest();
  }
  …
}
```

Instead:

```
public void doSomething(Account acct) {
  long adj = acct.getMonthlyAdjustment();
  …
}
```

# Delegation vs. inheritance summary

- Inheritance can improve modeling flexibility
- Usually, favor composition/delegation over inheritance
  - Inheritance violates information hiding
  - Delegation supports information hiding
- Design and document for inheritance, or prohibit it
  - Document requirements for overriding any method

# Today

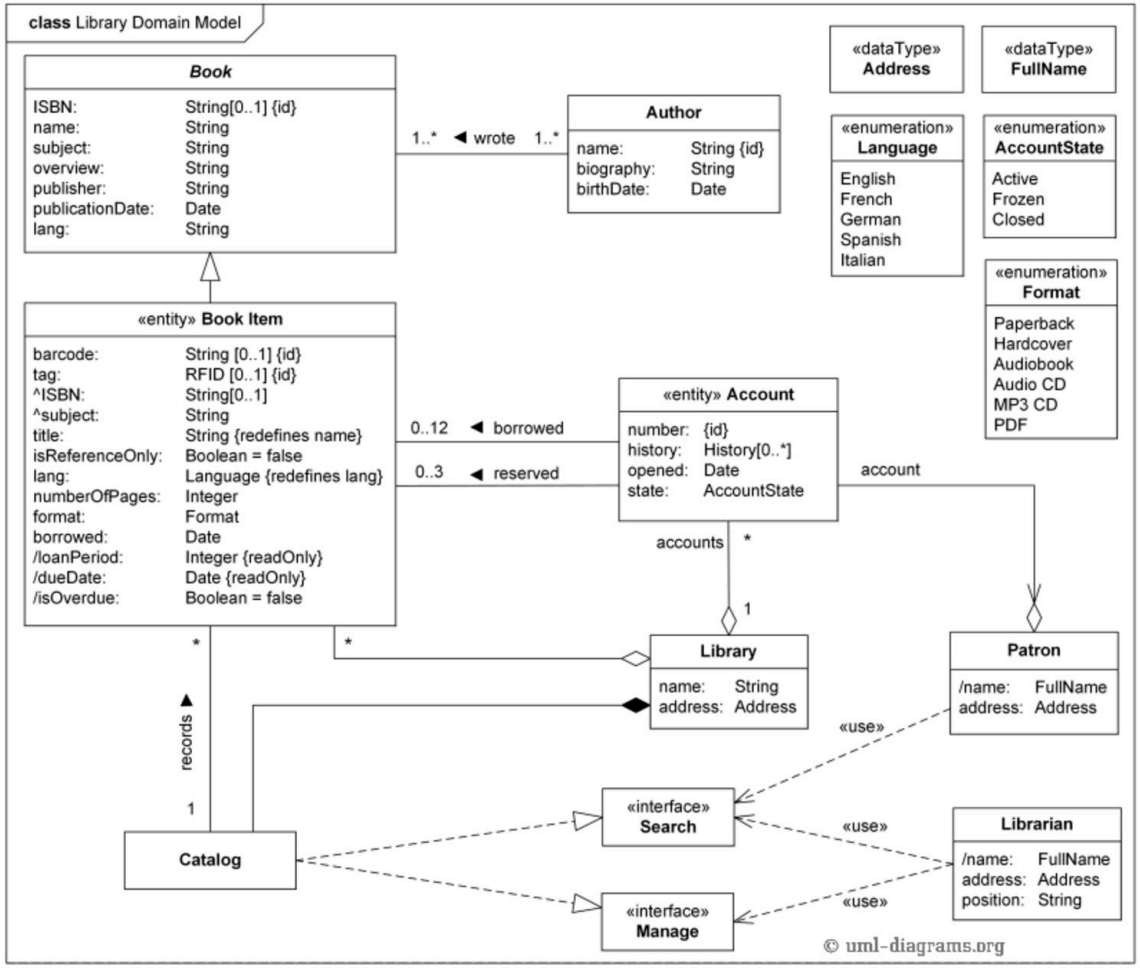
- Inheritance
  - Design for reuse: delegation vs inheritance
- UML class diagrams
- Introduction to design patterns
  - Strategy pattern
  - Command pattern
- Design patterns for reuse:
  - Template method pattern
  - Iterator pattern (next week)
  - Decorator pattern (next week)

# Religious debates…

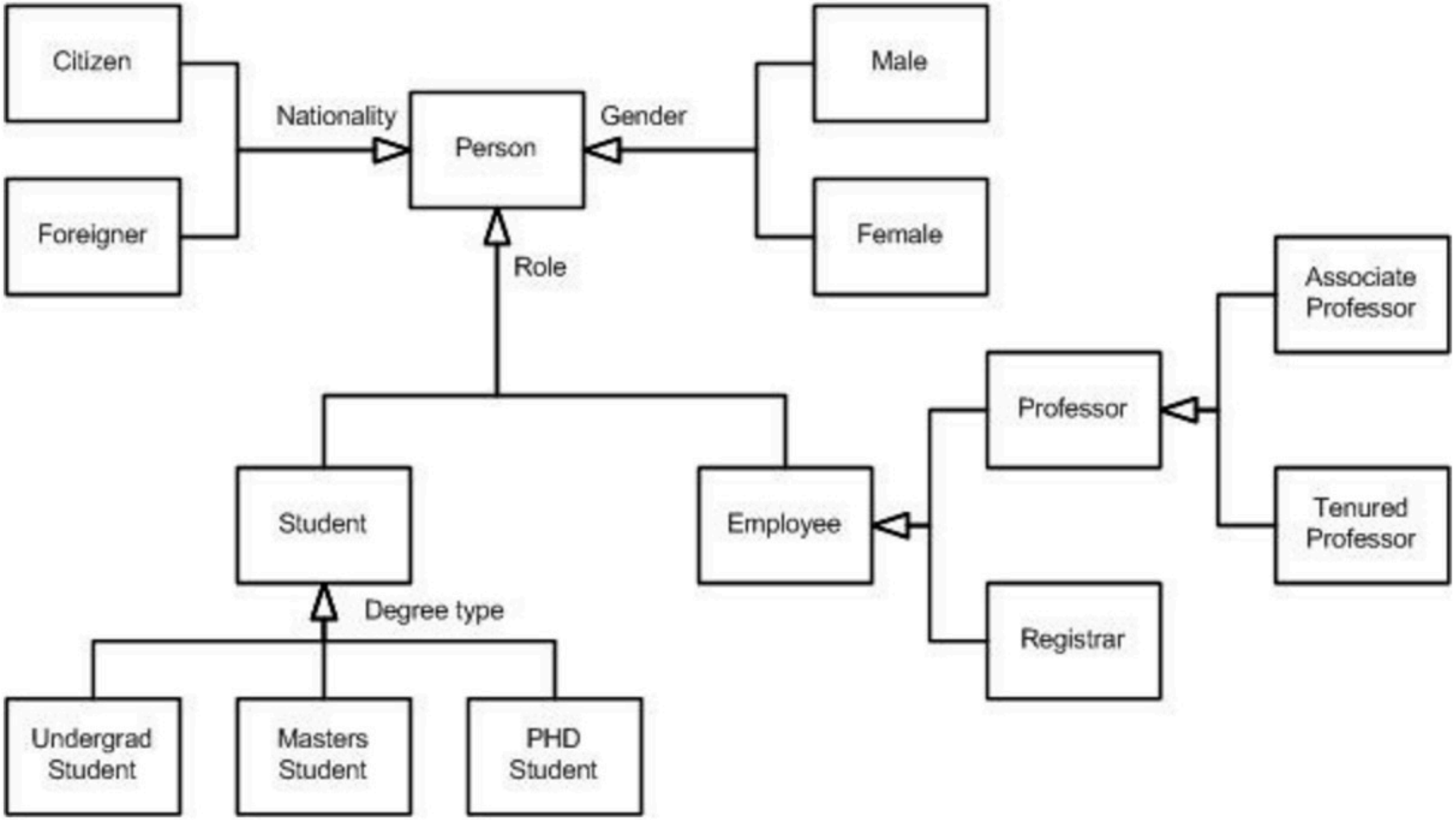
"Democracy is the worst form of government, except for all the others…"

-- (allegedly) Winston Churchill

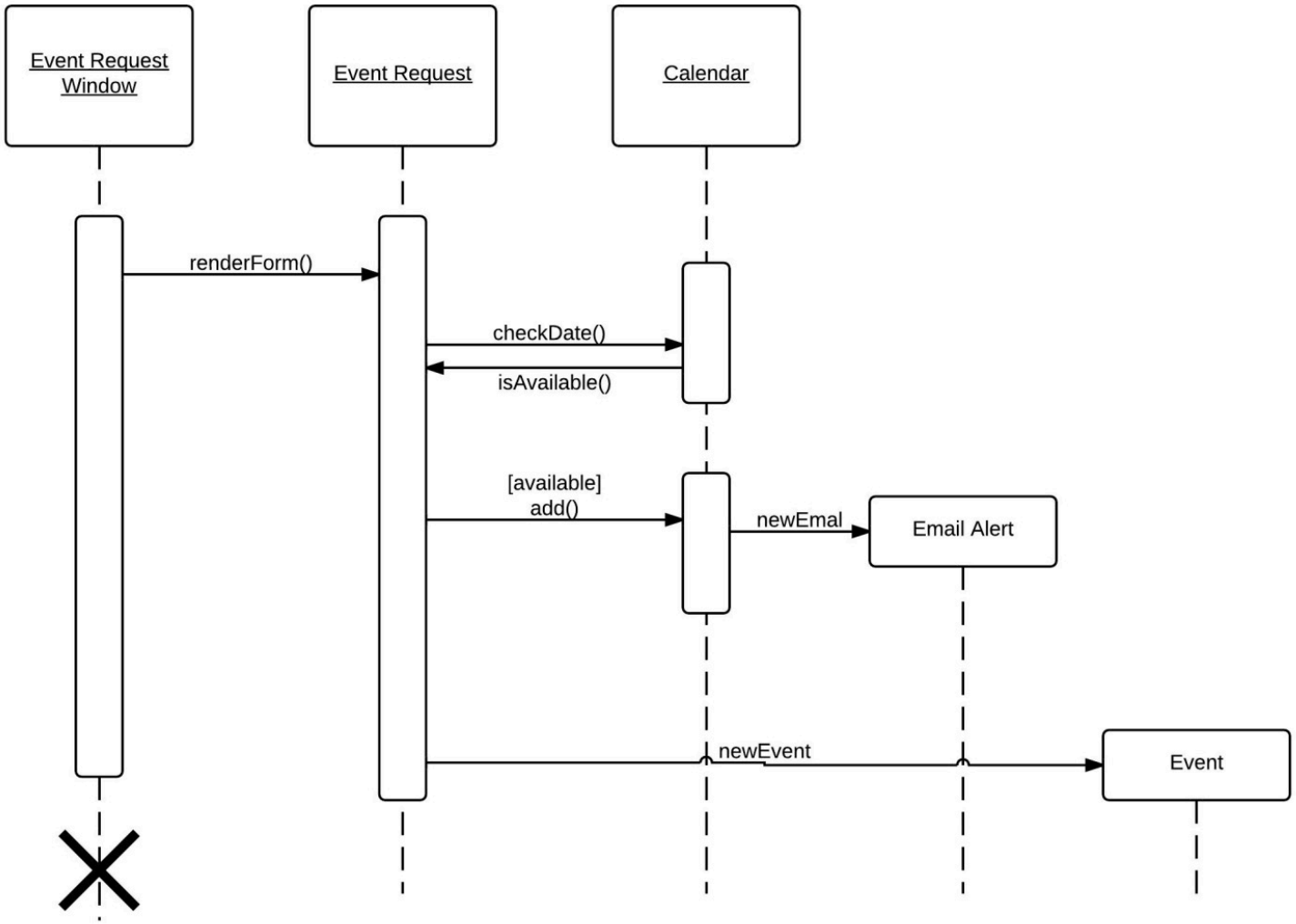
# UML: Unified Modeling Language



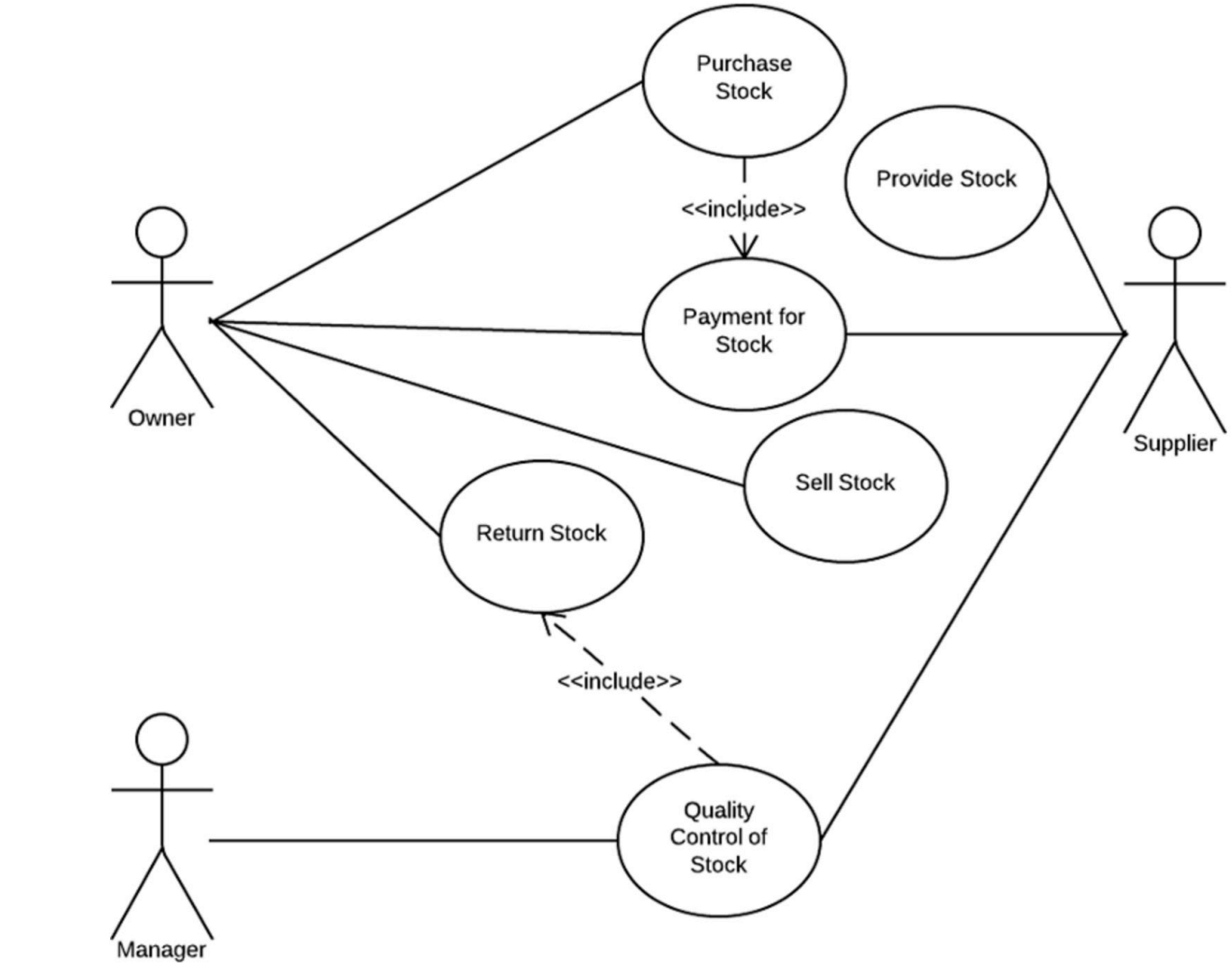
# UML: Unified Modeling Language



# UML: Unified Modeling Language



# UML: Unified Modeling Language



# UML in this course

- UML class diagrams
- UML interaction diagrams
  - Sequence diagrams

# UML class diagrams (interfaces and inheritance)

```
public interface Account {
    public long getBalance();
    public void  deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account target);
    public void monthlyAdjustment();
}

public interface CheckingAccount extends Account {
    public long getFee();
}

public interface SavingsAccount extends Account {
    public double getInterestRate();
}

public interface InterestCheckingAccount
                        extends CheckingAccount, SavingsAccount {
}
```

# UML class diagrams (classes)

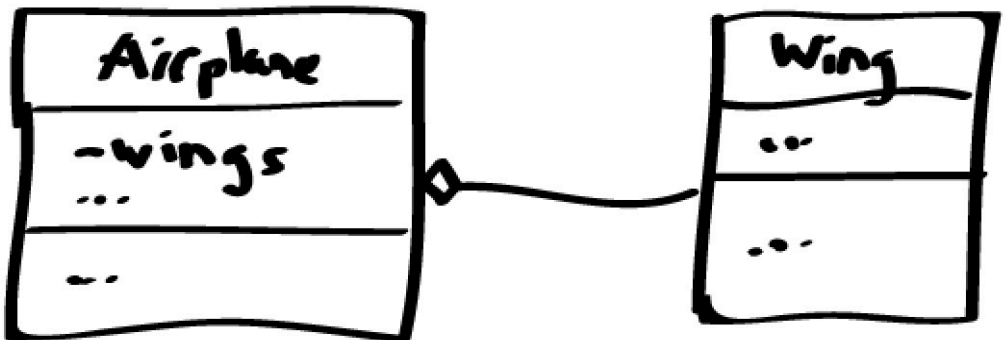
```
public abstract class AbstractAccount
        implements Account {
   protected long balance = 0;
   public long getBalance() {
        return balance;
   }
   abstract public void monthlyAdjustment();
   // other methods…
}

public class CheckingAccountImpl
        extends AbstractAccount
        implements CheckingAccount {
   public void monthlyAdjustment() {
        balance -= getFee();
   }
   public long getFee() { … }
}
```

# UML you should know

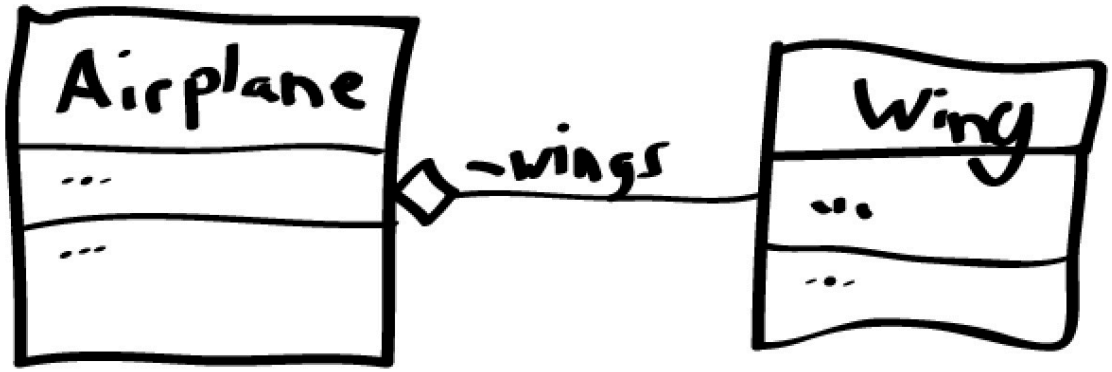
- Interfaces vs. classes
- Fields vs. methods
- Relationships:
  - "extends" (inheritance)
  - "implements" (realization)
  - "has a" (aggregation)
  - non-specific association
- Visibility: + (public) - (private) # (protected)
- Basic best practices…

# UML advice

- Best used to show the big picture
  - Omit unimportant details
    - But show they are there: …
- Avoid redundancy
  - e.g., bad:




good:



# Today

- Inheritance
  - Design for reuse: delegation vs inheritance
- UML class diagrams
- Introduction to design patterns
  - Strategy pattern
  - Command pattern
- Design patterns for reuse:
  - Template method pattern
  - Iterator pattern (next week)
  - Decorator pattern (next week)

# One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

# Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

# A third design scenario

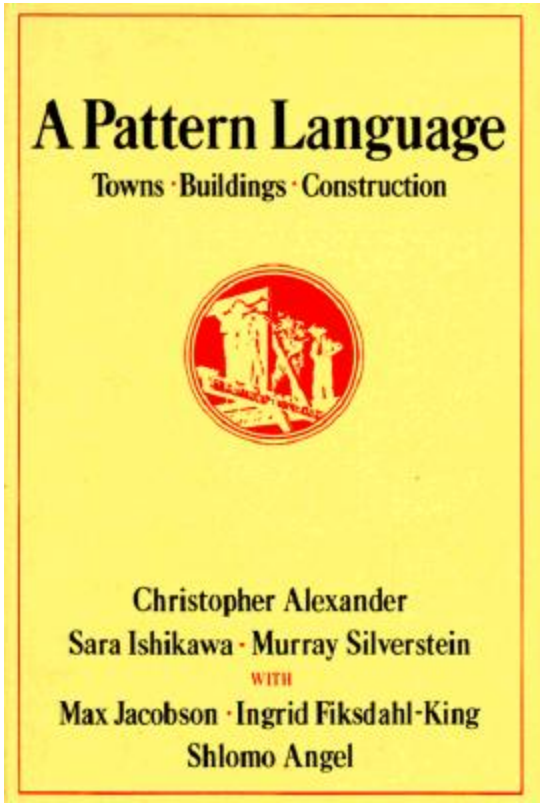
- Suppose we need to sort a list in different orders…

```
interface Order {
  boolean lessThan(int i, int j);
}

final Order ASCENDING =  (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

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

# *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, Architect (1977)



# How not to discuss design (from Shalloway and Trott)

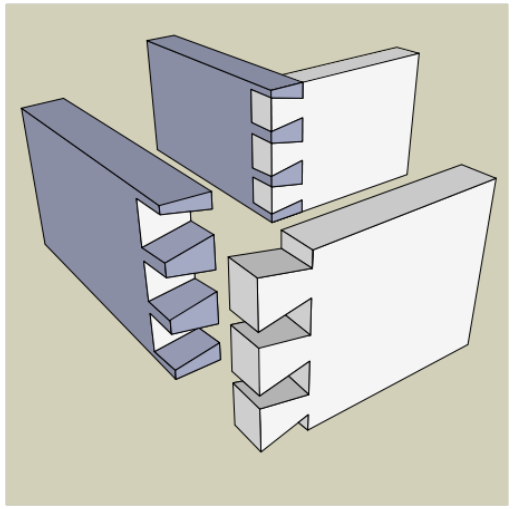
- Carpentry:
  - How do you think we should build these drawers?
  - Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating…

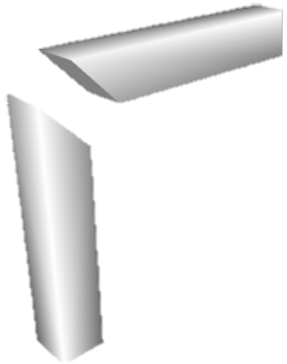
# How not to discuss design (from Shalloway and Trott)

- 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…

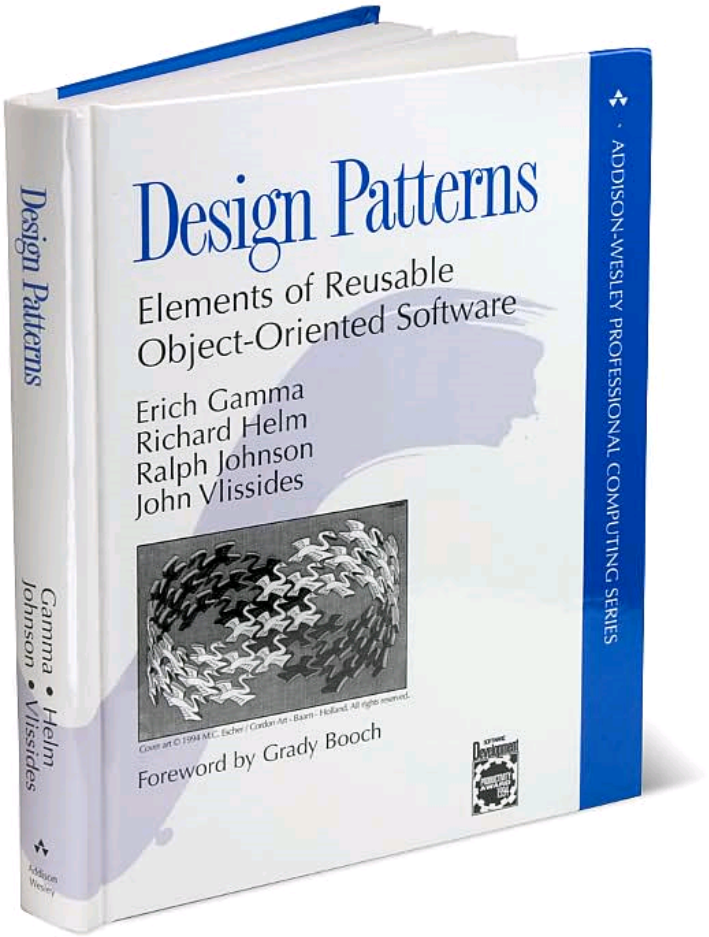
# Discussion with design patterns

- Carpentry:
  - "Is a dovetail joint or a miter joint better here?"
- Software Engineering:
  - "Is a strategy pattern or a template method better here?"





# History: *Design Patterns* (1994)



# Elements of a design pattern

- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences

# Strategy pattern

- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces an extra interface and many classes:
    - Code can be harder to understand
    - Lots of overhead if the strategies are simple

# Patterns are more than just structure

- Consider: A modern car engine is constantly monitored by a software system. The monitoring system must obtain data from many distinct engine sensors, such as an oil temperature sensor, an oxygen sensor, etc. More sensors may be added in the future.

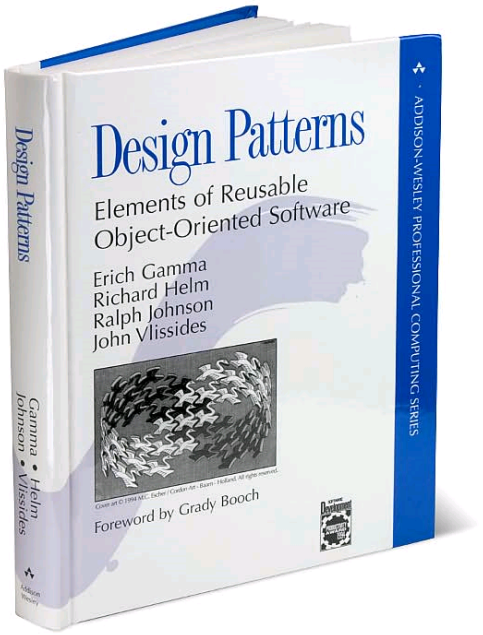
# Different patterns can have the same structure

Command pattern:

- Problem: Clients need to execute some (possibly flexible) operation without knowing the details of the operation
- Solution: Create an interface for the operation, with a class (or classes) that actually executes the operation
- Consequences:
  - Separates operation from client context
  - Can specify, queue, and execute commands at different times
  - Introduces an extra interface and classes:
    - Code can be harder to understand
    - Lots of overhead if the commands are simple

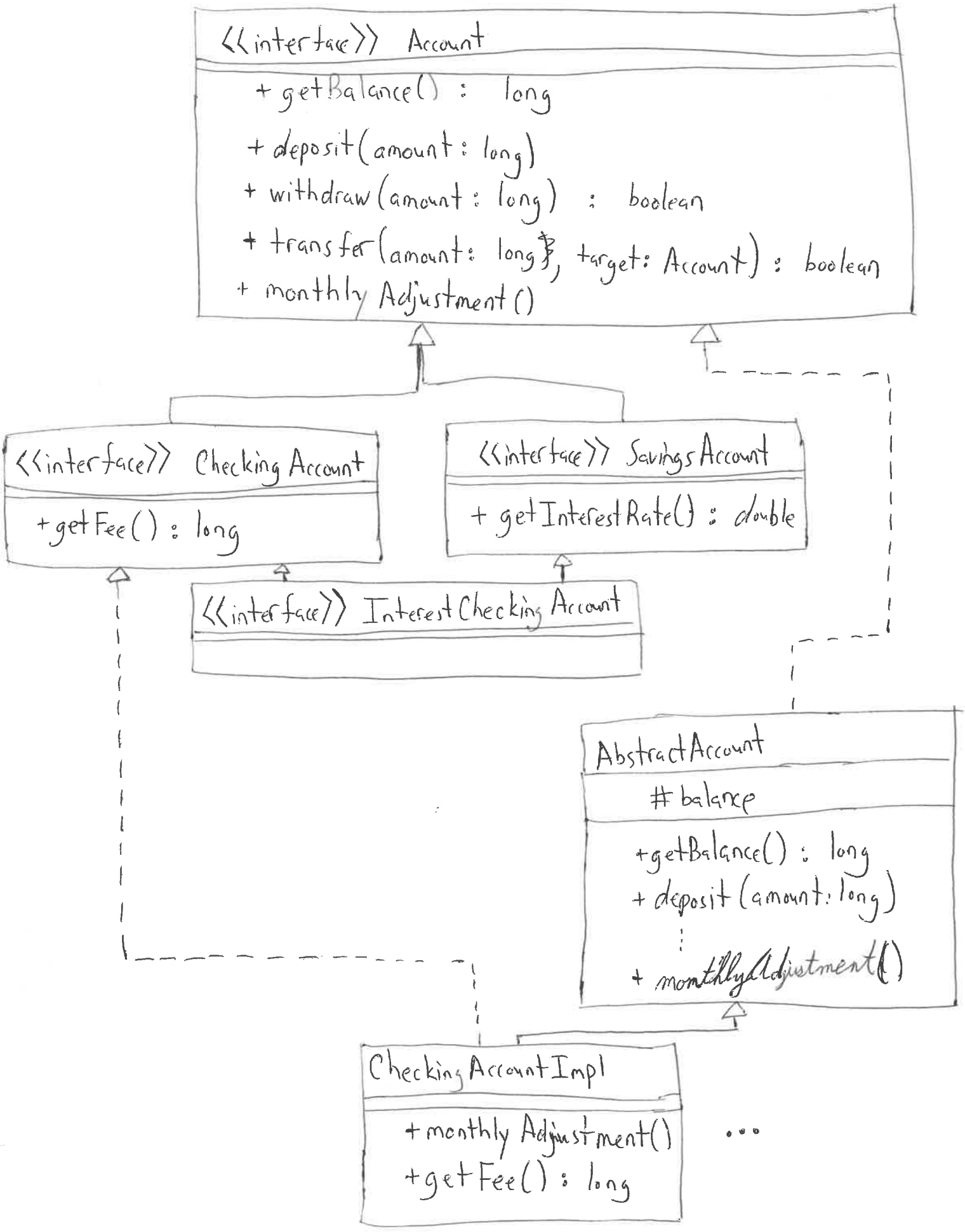
# Design pattern conclusions

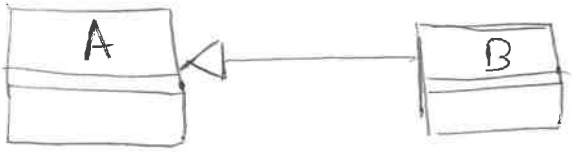
- Provide shared language
- Convey shared experience
- Can be system and language specific



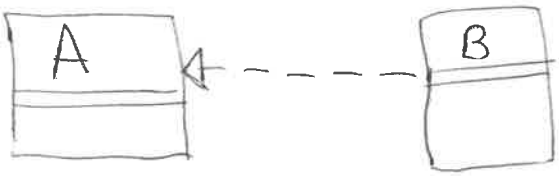
# Summary

- Prefer delegation to inheritance
- Use UML class diagrams to simplify communication
- Design patterns…
  - Convey shared experience, general solutions
  - Facilitate communication
- Specific design patterns for reuse:
  - Strategy
  - Command


<<interface>> Account
+ getBalance() : long
+ deposit(amount: long)
+ withdraw(amount: long) : boolean
+ transfer(amount: long, target: Account) : boolean
+ monthlyAdjustment()
<<interface>> CheckingAccount
+ getFee() : long
<<interface>> SavingsAccount
+ getInterestRate() : double
<<interface>> InterestCheckingAccount
AbstractAccount
# balance
+ getBalance() : long
+ deposit(amount: long)
+ monthlyAdjustment()
CheckingAccountImpl
+ monthlyAdjustment()
+ getFee() : long
...

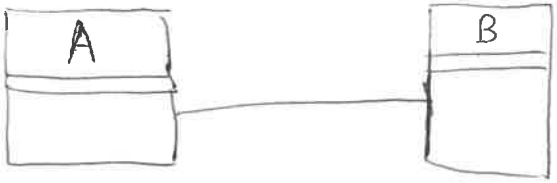



B extends A

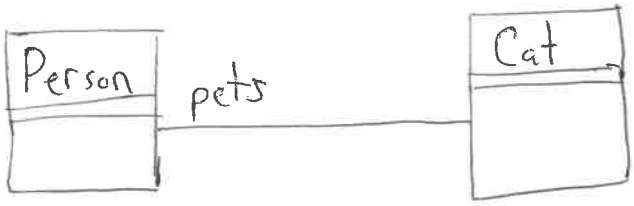


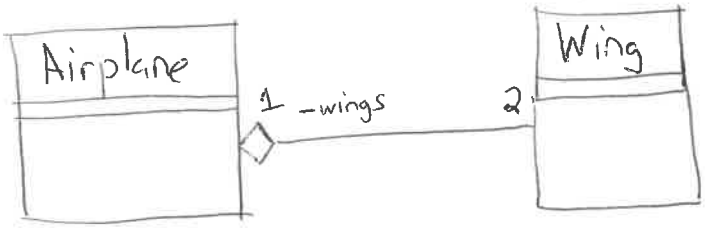

B implements A


B has a A




B is associated with A

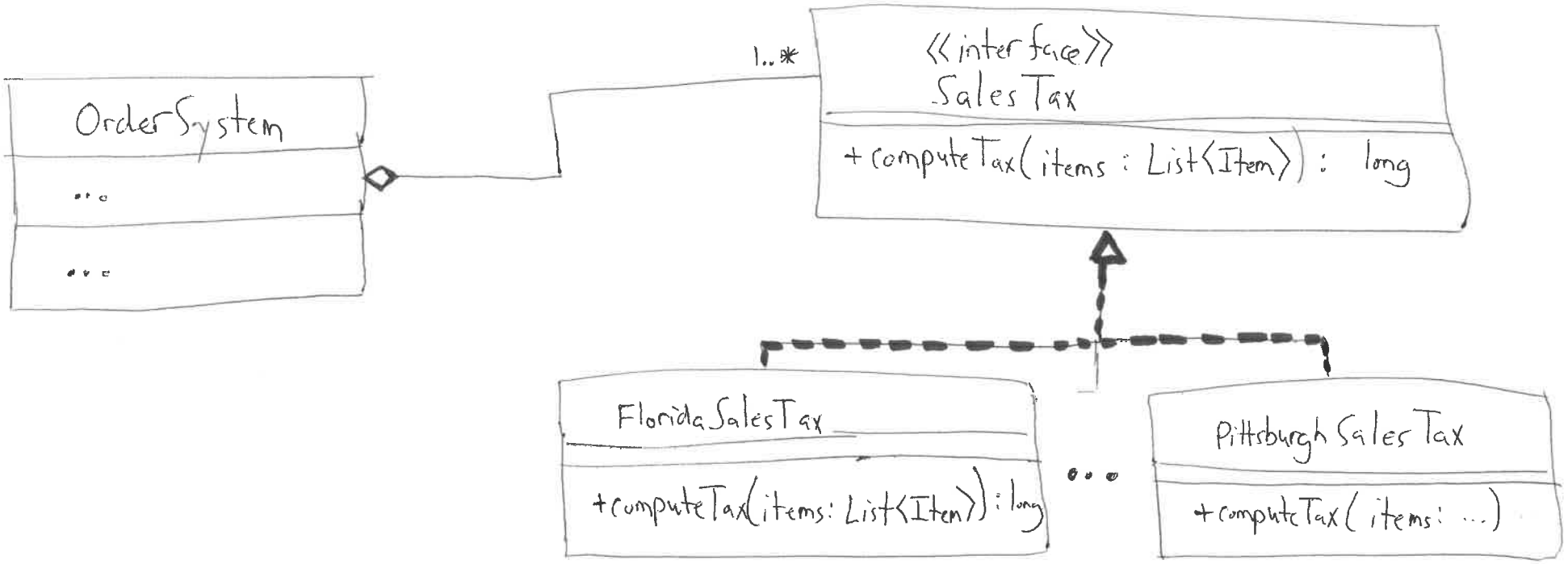


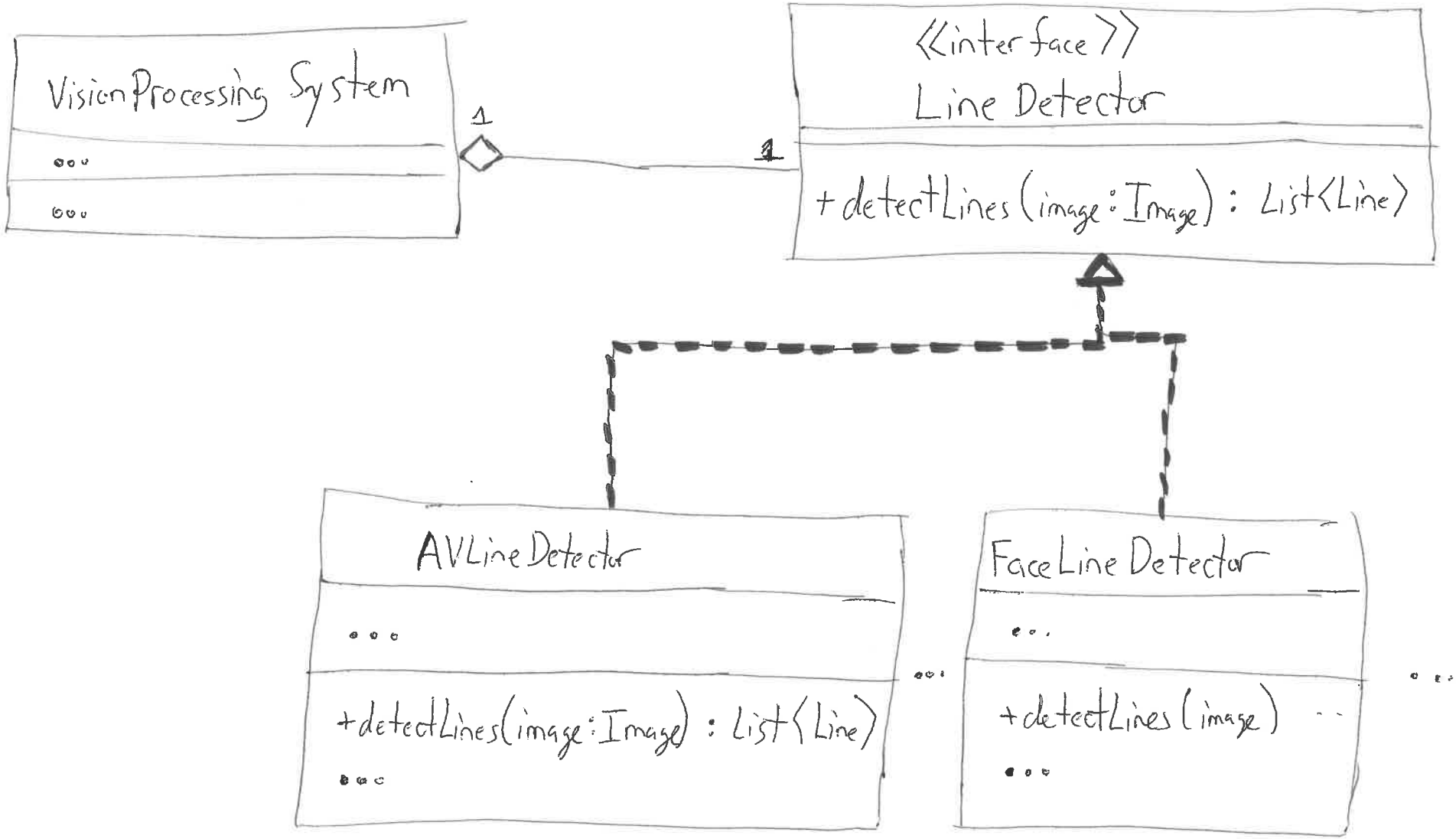



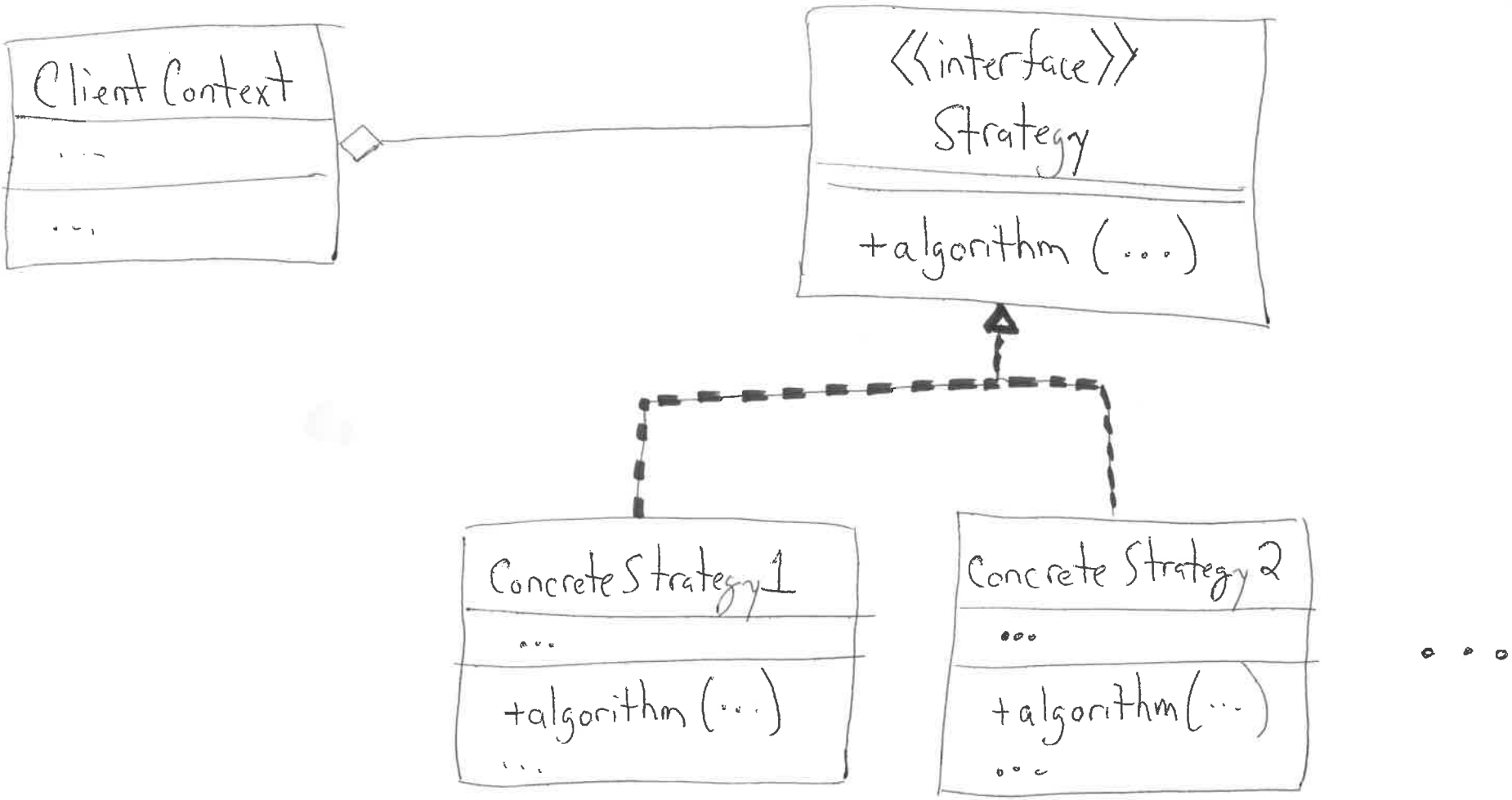

An airplane has exactly two wings.

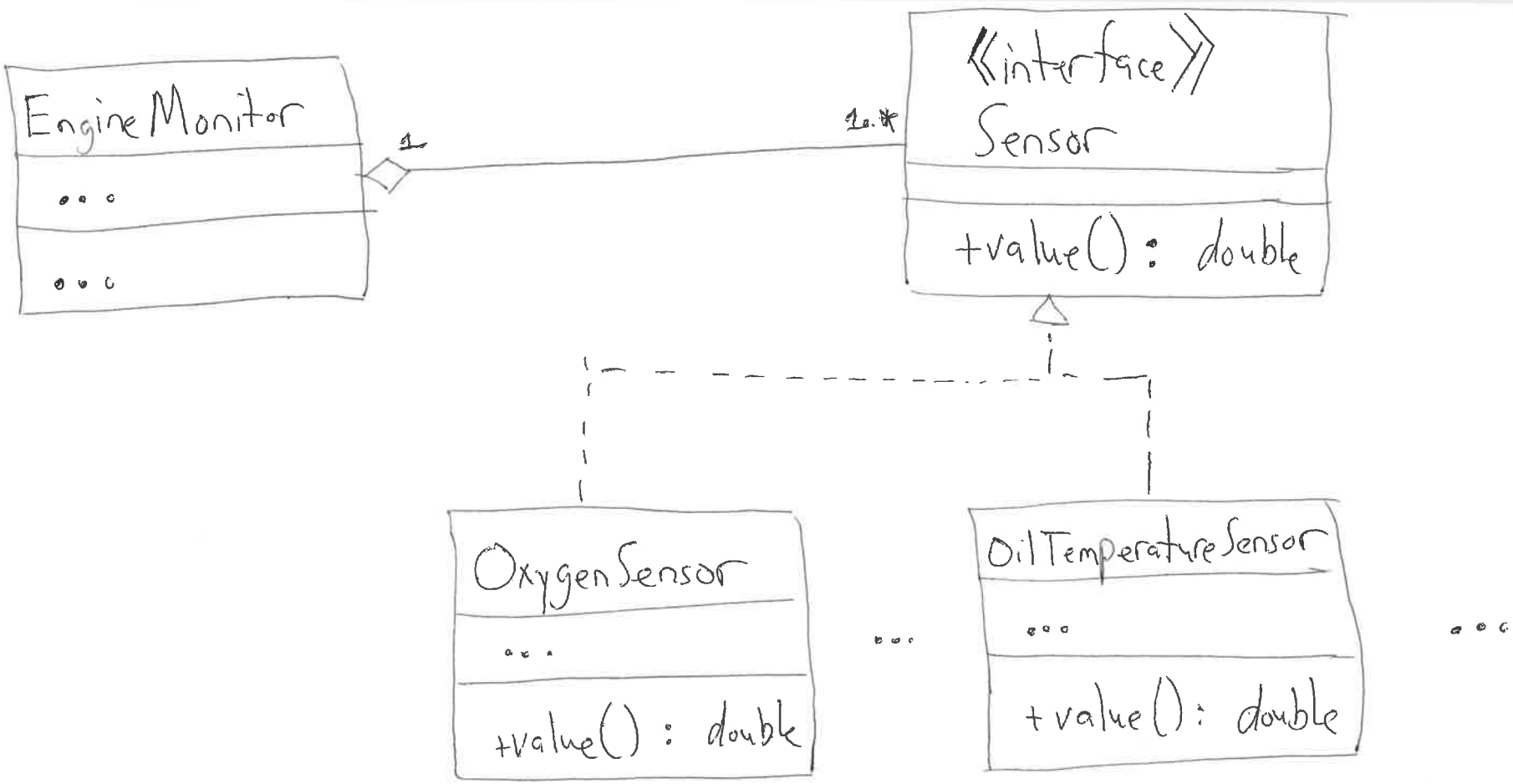
A wing is in/on exactly one airplane.

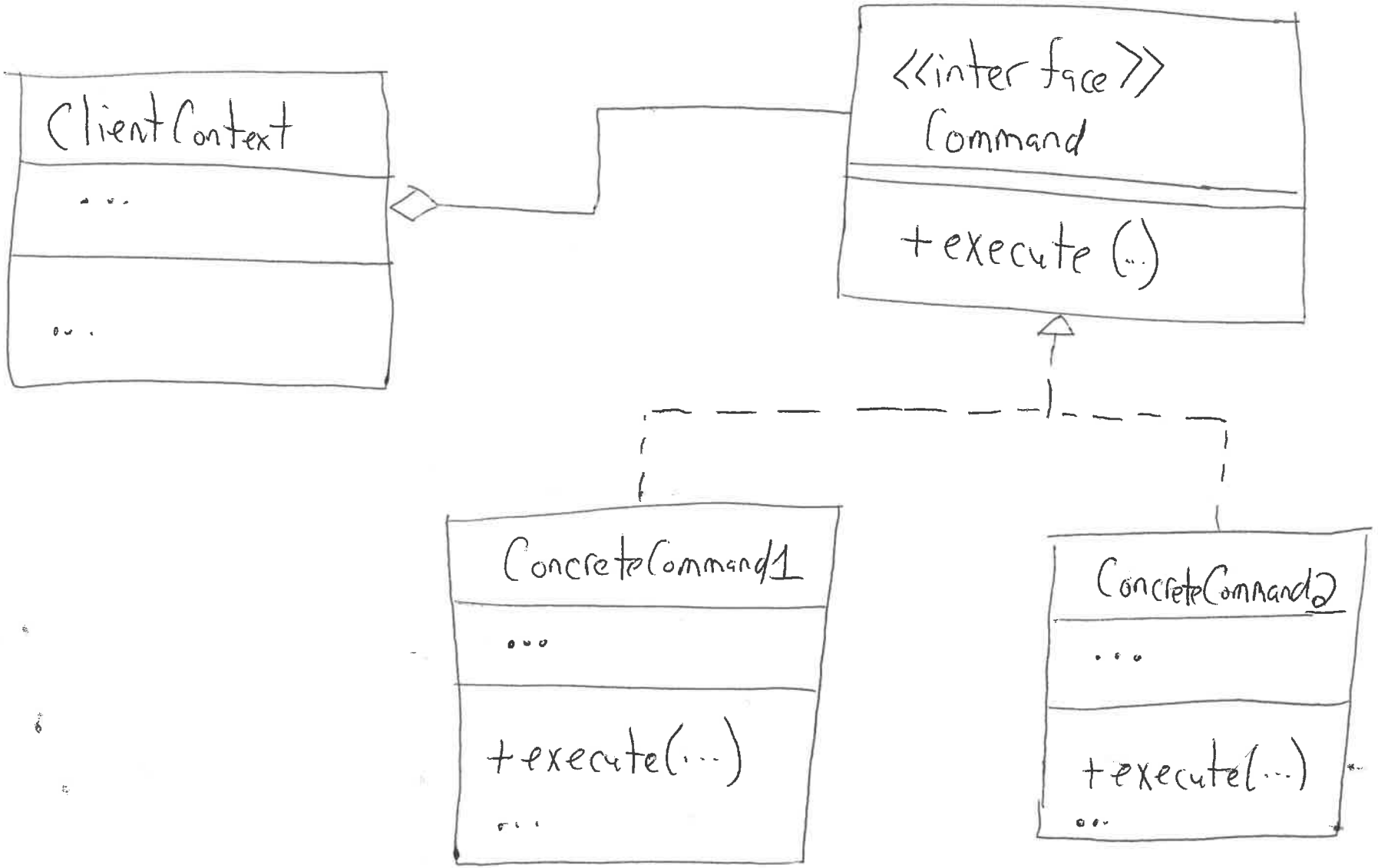
The wings are a private variable called wings.


OrderSystem
...
...
1..*
«interface»
SalesTax
+computeTax(items : List<Item>) : long
FloridaSalesTax
+computeTax(items: List<Item>): long
...
PittsburghSalesTax
+computeTax(items: ...)


VisionProcessing System
...
...
1
1
«interface»
Line Detector
+detectLines(image:Image) : List<Line>
AVLine Detector
...
+detectLines(image:Image) : List<Line>
...
FaceLine Detector
...
+detectLines(image)
...


Client Context
...
...
<<interface>>
Strategy
+algorithm (...)
ConcreteStrategy1
...
+algorithm (...)
...
ConcreteStrategy2
...
+algorithm(...)
...
...


EngineMonitor
...
...
1
1..*
«interface»
Sensor
+value(): double
OxygenSensor
...
+value(): double
...
OilTemperatureSensor
...
+value(): double
...


ClientContext
...
...
<<interface>>
Command
+execute(...)
ConcreteCommand1
...
+execute(...)
...
ConcreteCommand2
...
+execute(...)
...