Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Design for reuse

Introduction to design patterns

Charlie Garrod Bogdan Vasilescu

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- 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

Key concepts from Tuesday

- Behavioral subtyping (continued)
  - Liskov Substitution Principle
  - The `java.lang.Object` contracts
- Design for reuse with delegation (vs. inheritance)

Delegation vs. inheritance

- 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

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();
  …
}
```

Today

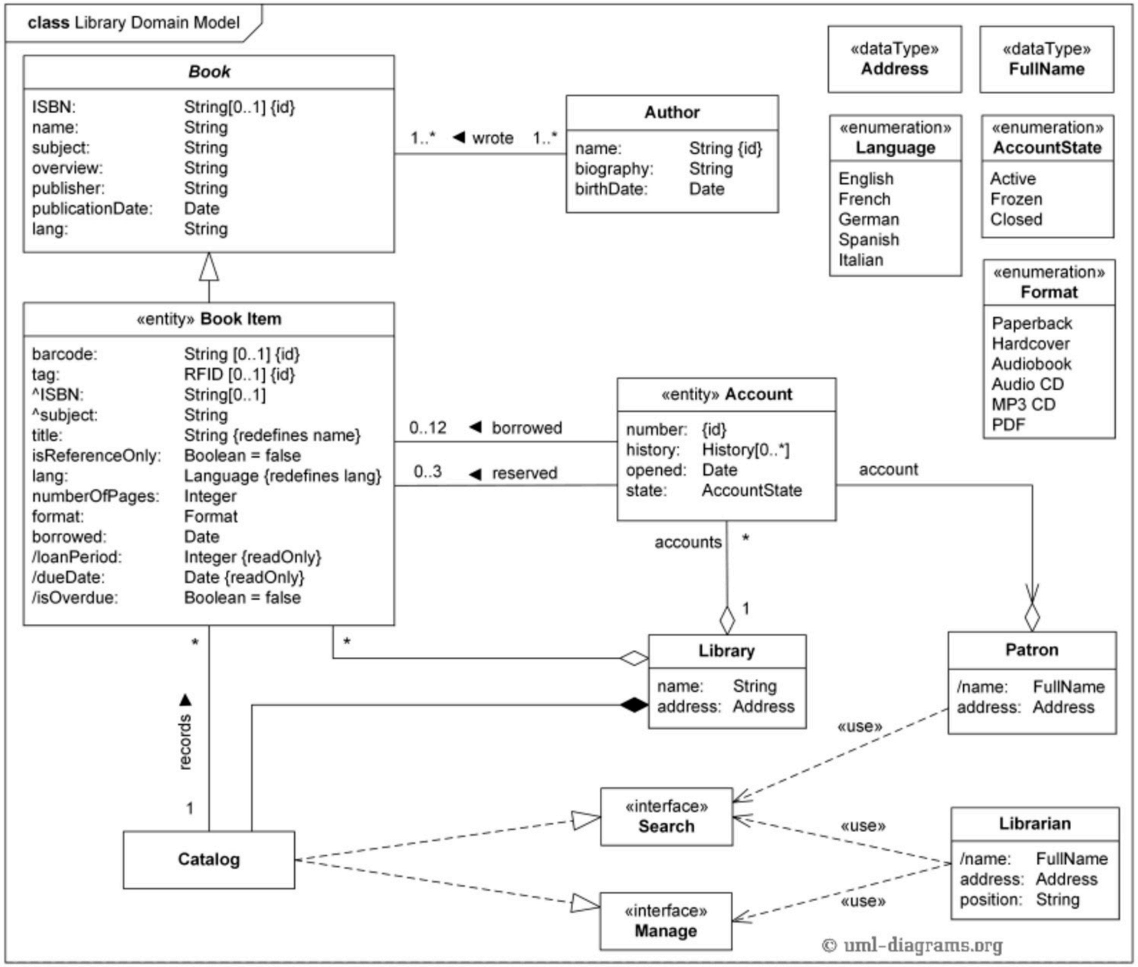
- UML class diagrams
- Introduction to design patterns
  - Strategy pattern
  - Command pattern
- Design patterns for reuse:
  - Template method pattern
  - Iterator pattern
  - Decorator pattern (next week)

Religious debates…

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

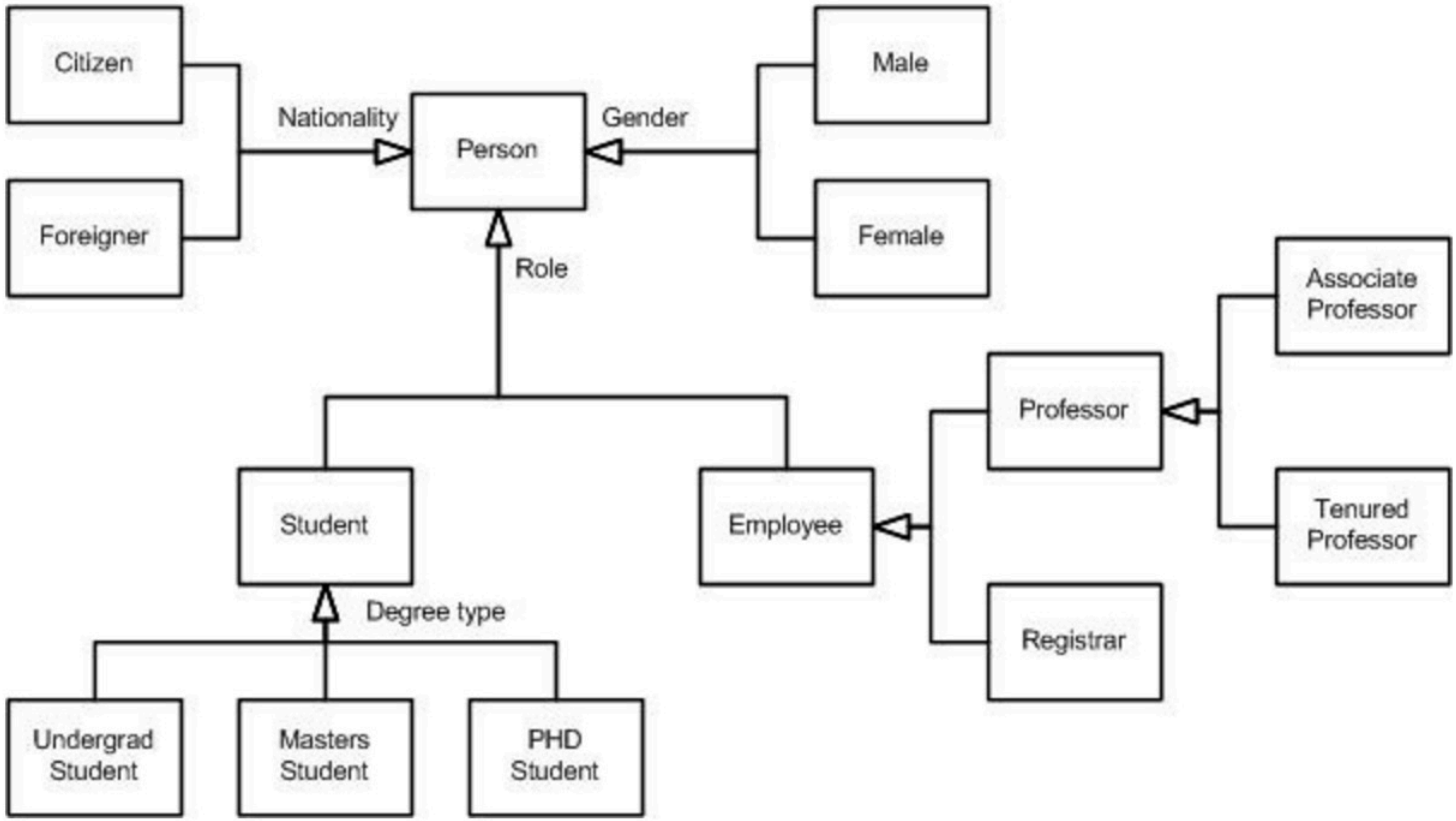
-- (allegedly) Winston Churchill

UML: Unified Modeling Language

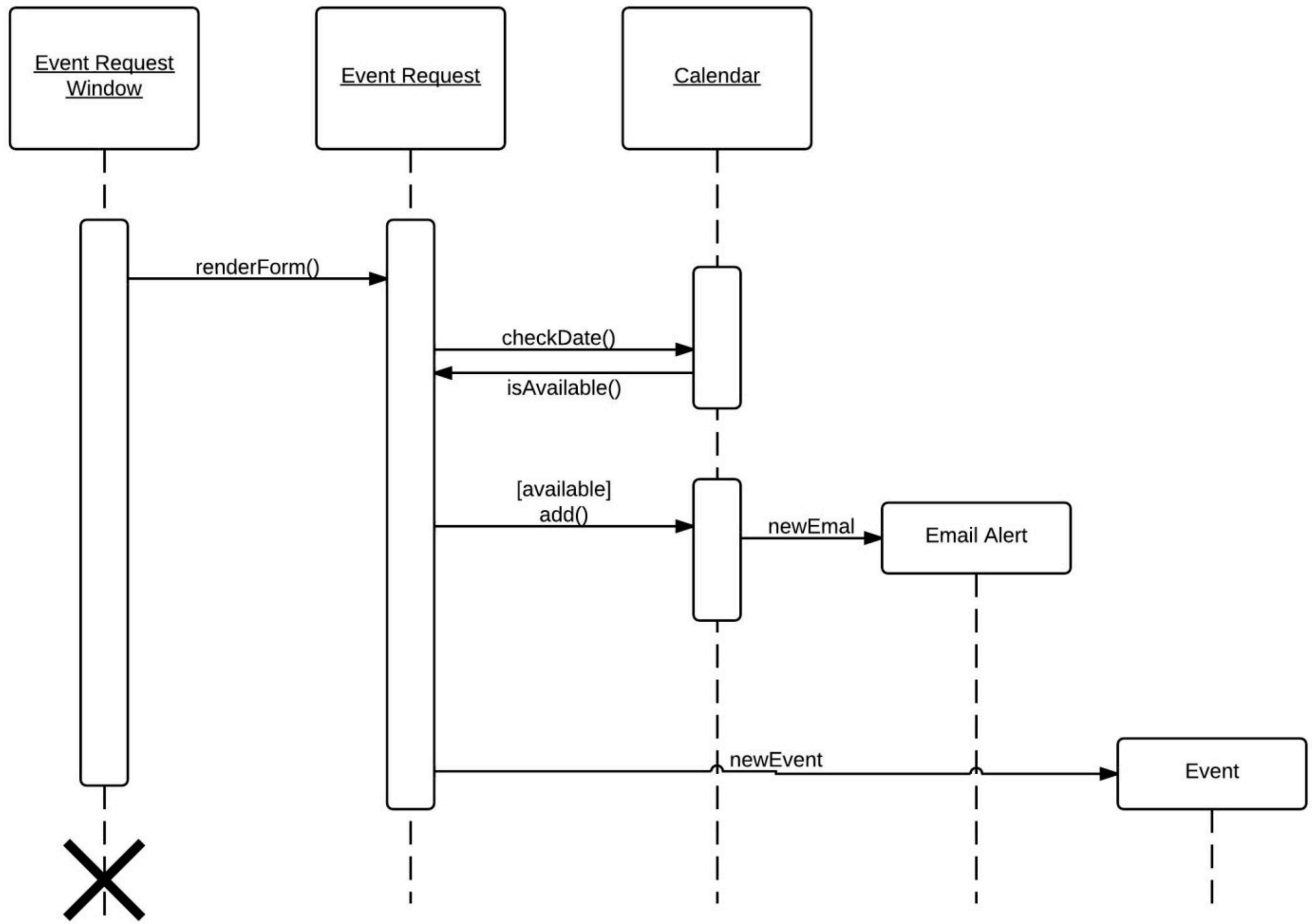


institute for SOFTWARE RESEARCH

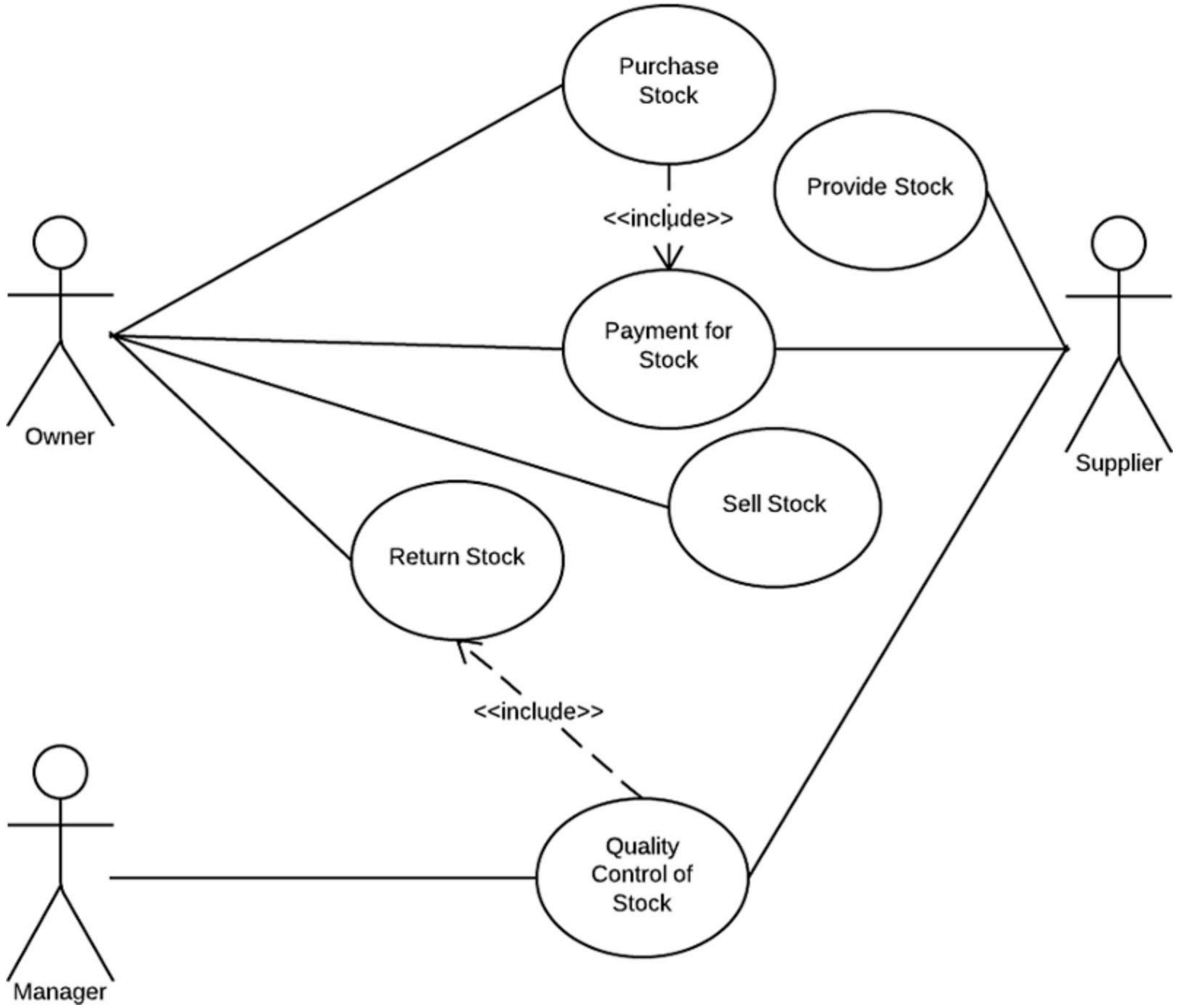
UML: Unified Modeling Language



UML: Unified Modeling Language



UML: Unified Modeling Language



UML in this course

- UML class diagrams
- UML interaction diagrams
  - Sequence diagrams
  - Communication 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)

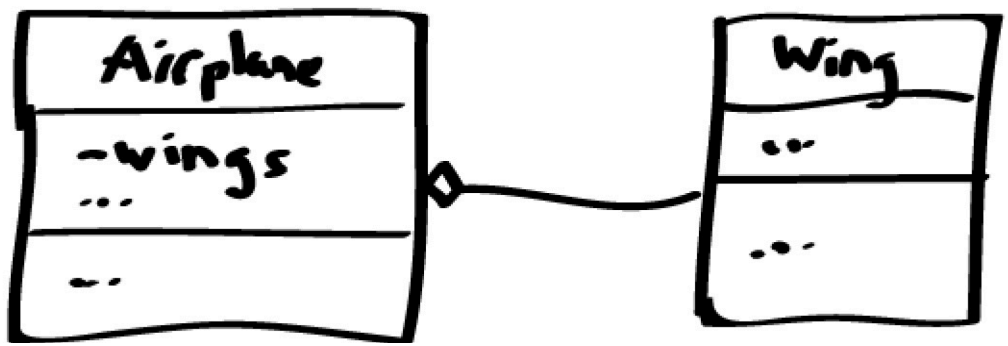
```
public abstract class AbstractAccount
        implements Account {
   protected long balance = 0;
   public long getBalance() {
        return balance;
   }
   abstract public void monthlyAdjustment();
   // other methods…
}

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

UML you should know

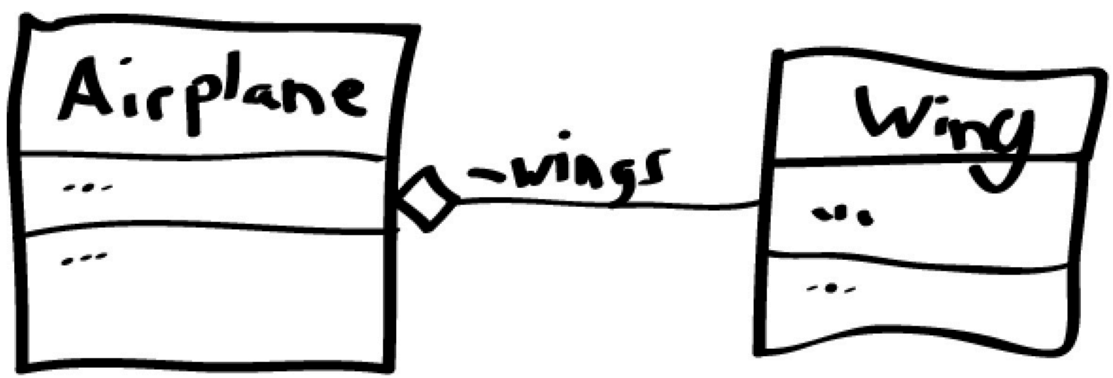
- Interfaces vs. classes
- Fields vs. methods
- Relationships:
  - "extends" (inheritance)
  - "implements" (realization)
  - "has a" (aggregation)
  - non-specific association
- Visibility: + (public) - (private) # (protected)
- Basic best practices…

UML advice

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



good:



Today

- UML class diagrams
- Introduction to design patterns
  - Strategy pattern
  - Command pattern
- Design patterns for reuse:
  - Template method pattern
  - Iterator pattern
  - 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

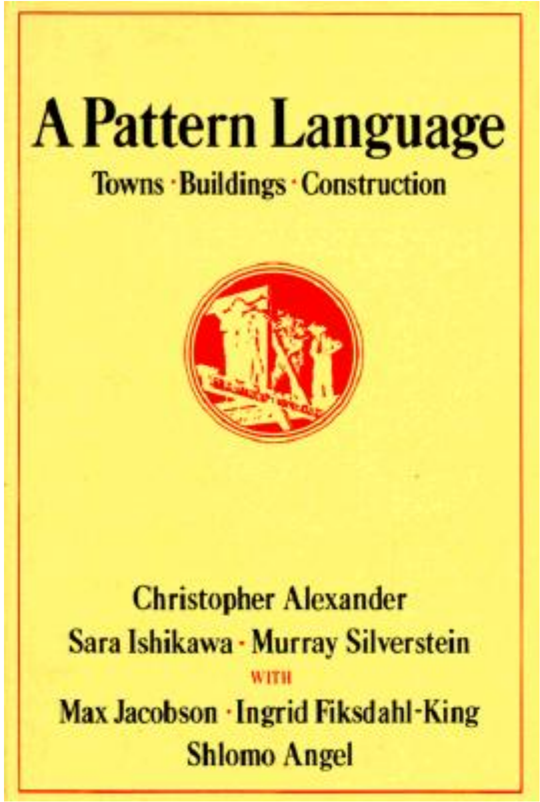
- Suppose we need to sort a list in different orders…

```
interface Comparator {
  boolean compare(int i, int j);
}

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

static void sort(int[] list, Comparator cmp) {
  …
  boolean mustSwap =
    cmp.compare(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)

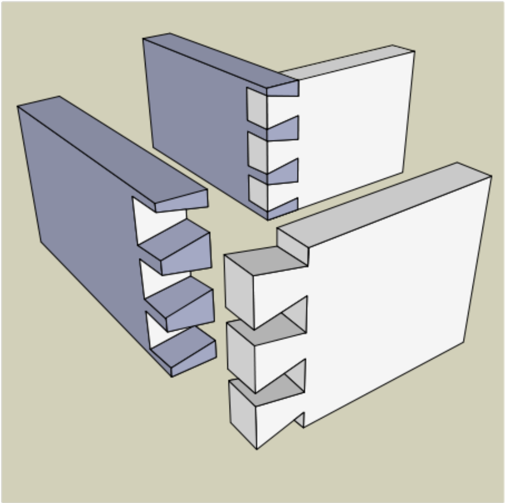


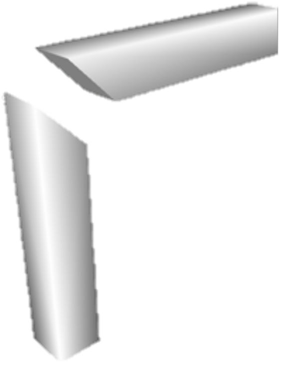
How not to discuss design (from Shalloway and Trott)

- 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…
- 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…

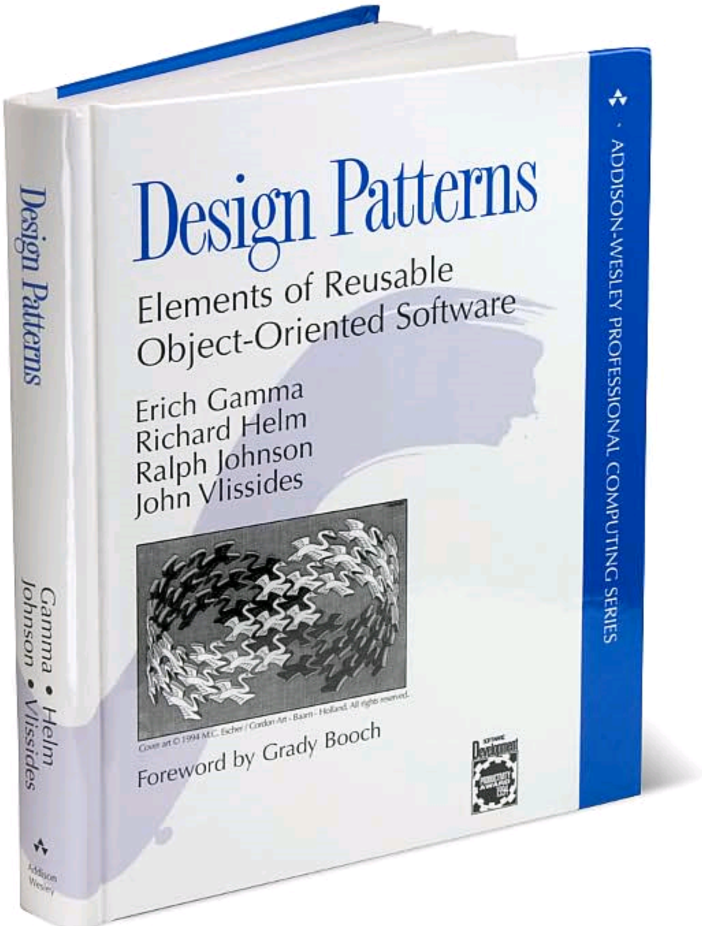
Discussion with design patterns

- 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.

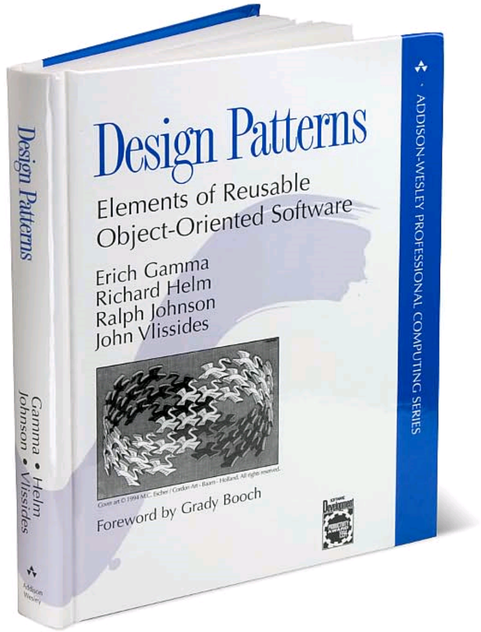
Some patterns have the same structure

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



Today

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

One design scenario

- A GUI-based document editor works with multiple document formats. Some parts of the algorithm to load a document (e.g., reading a file, rendering to the screen) are the same for all document formats, and other parts of the algorithm vary from format-to-format (e.g. parsing the file input).

Another design scenario

- Several versions of a domain-specific machine learning algorithm are being implemented to use data stored in several different database systems. The basic algorithm for all versions is the same; just the interactions with the database are different from version to version.

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

```
abstract T   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();
abstract Iterator<E>  iterator();
Object[]     toArray()
<T> T[]      toArray(T[] a);
…
```


institute for SOFTWARE RESEARCH

Template method pattern

- Problem: An algorithm consists of customizable parts and invariant parts
- Solution: Implement the invariant parts of the algorithm in an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations
- Consequences
  - Code reuse for the invariant parts of algorithm
  - Customization is restricted to the primitive operations
  - Inverted (Hollywood-style) control for customization

Template method vs. the strategy pattern

- Template method uses inheritance to vary part of an algorithm
  - Template method implemented in supertype, primitive operations implemented in subtypes
- Strategy pattern uses delegation to vary the entire algorithm
  - Strategy objects are reusable across multiple classes
  - Multiple strategy objects are possible per class

Today

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

Traversing a collection

- Since Java 1.0:

```
List<String> arguments = …;
for (int i = 0; i < arguments.size(); ++i) {
  System.out.println(arguments.get(i));
}
```

- Java 1.5: for-each loop

```
List<String> arguments = …;
for (String s : arguments) {
  System.out.println(s);
}
```

- For-each loop works for every implementation of `Iterable`

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

The `Iterator` interface

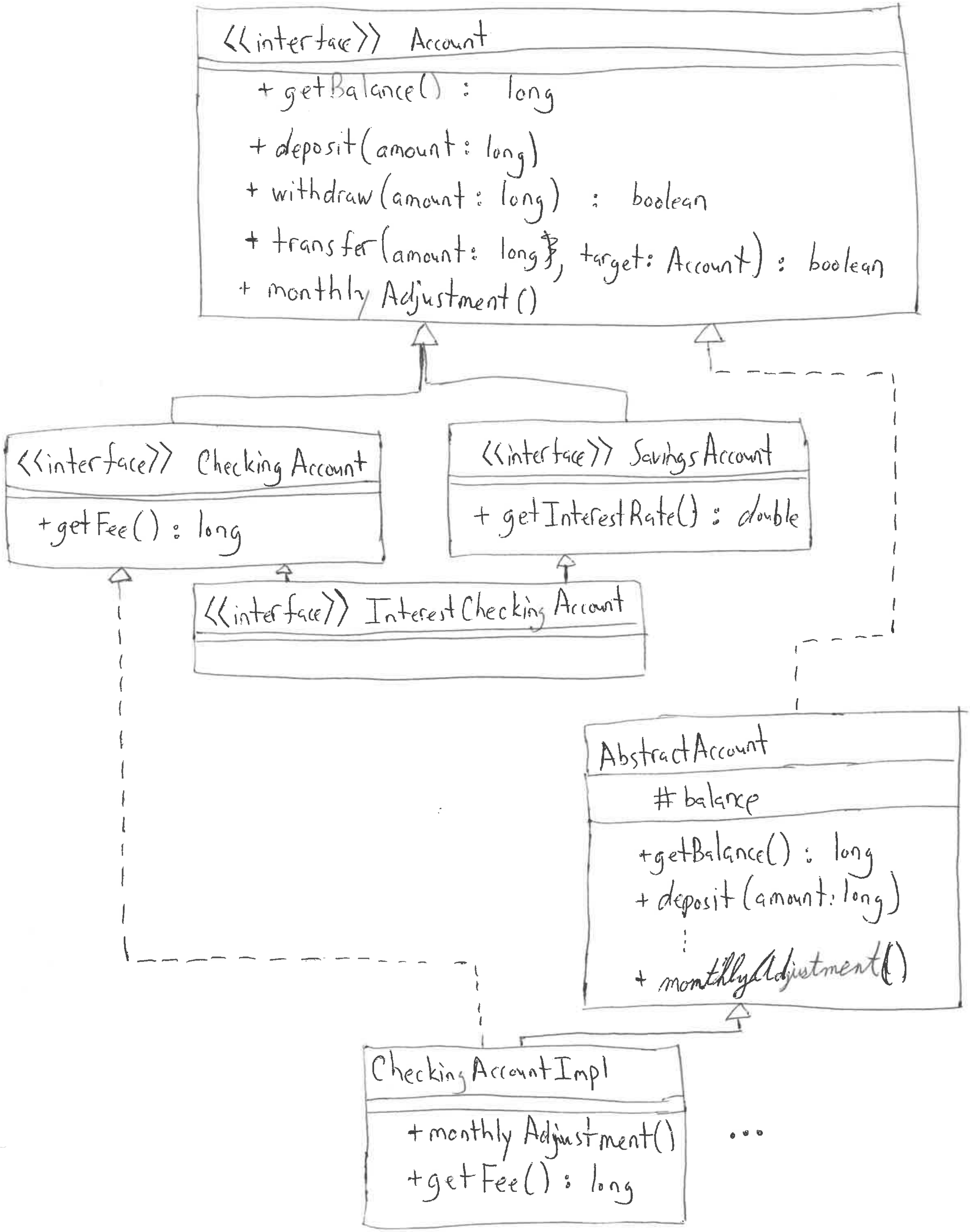
```
public interface java.util.Iterator<E> {
  boolean hasNext();
  E next();
  void remove();  // removes previous returned item
}                 // from the underlying collection
```

- To use explicitly, e.g.:

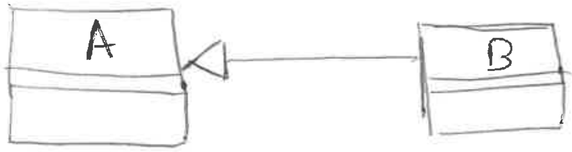
```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
     it.hasNext();  ) {
  String s = it.next();
  System.out.println(s);
}
```

Summary

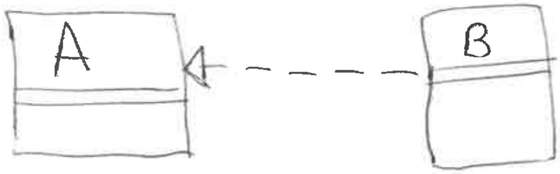
- Iterator will be continued, and more design patterns for reuse next week
- Use UML class diagrams to simplify communication
- Design patterns…
  - Convey shared experience, general solutions
  - Facilitate communication
- Specific design patterns for reuse:
  - Strategy
  - Template method
  - Iterator



<<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
...



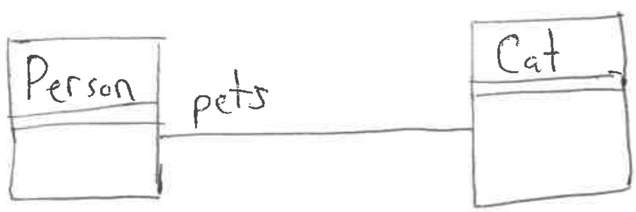
B extends A

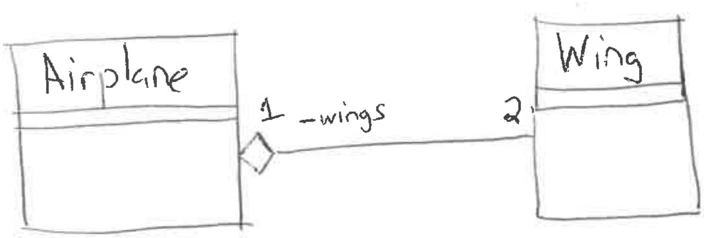


B implements A

B has a A

B is associated with A

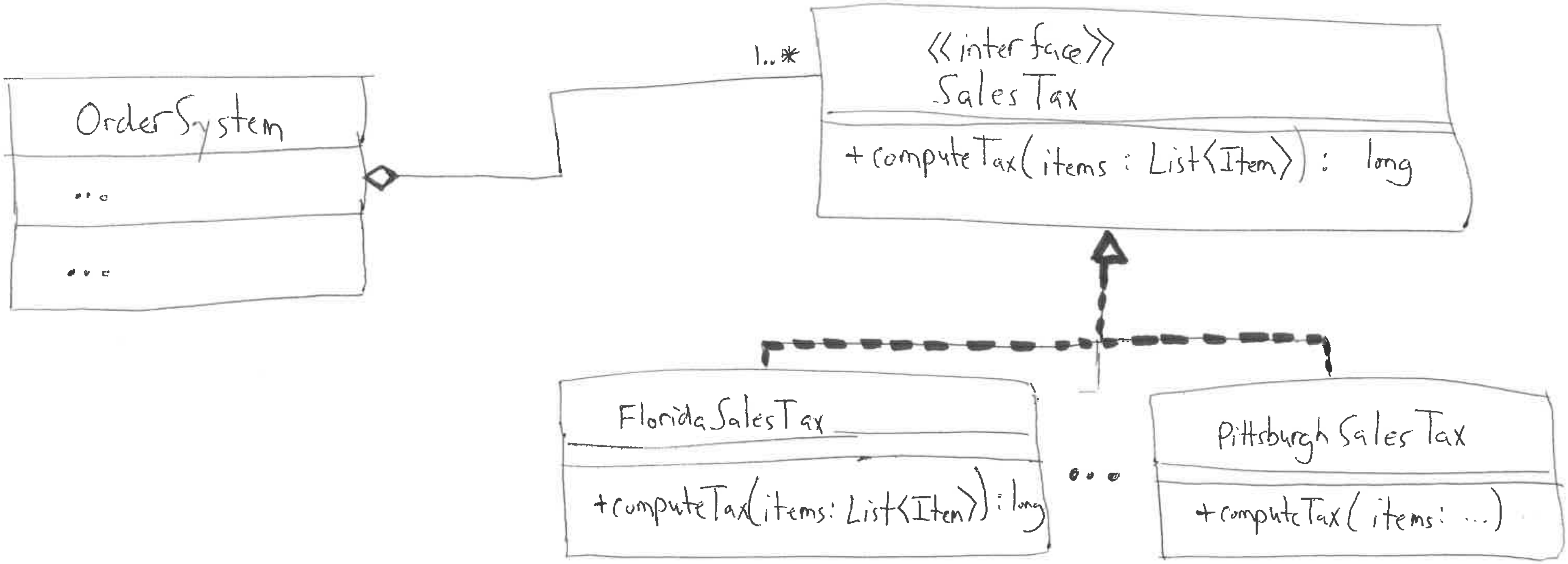




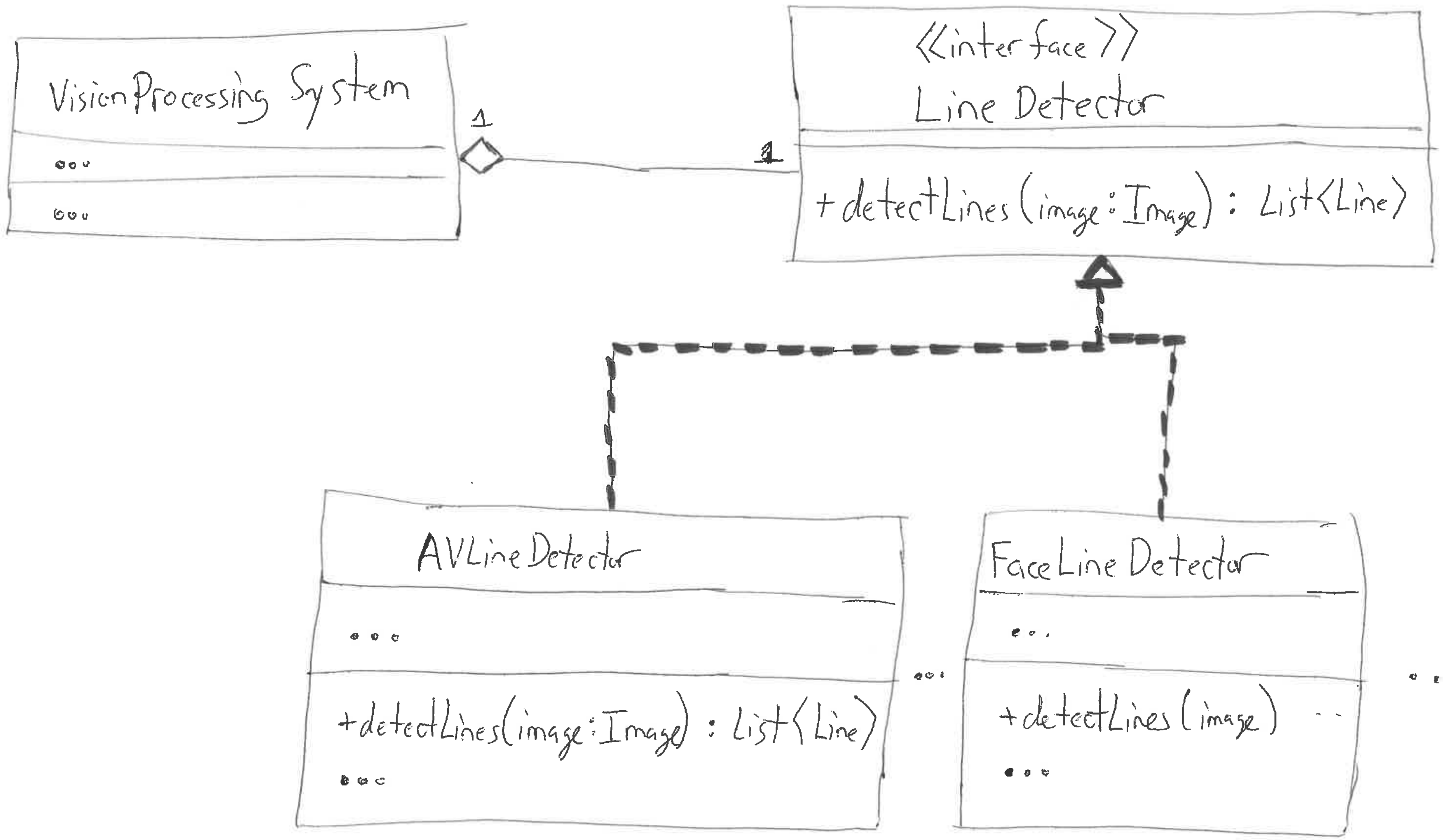
An airplane has exactly two wings.

A wing is in/on exactly one airplane.

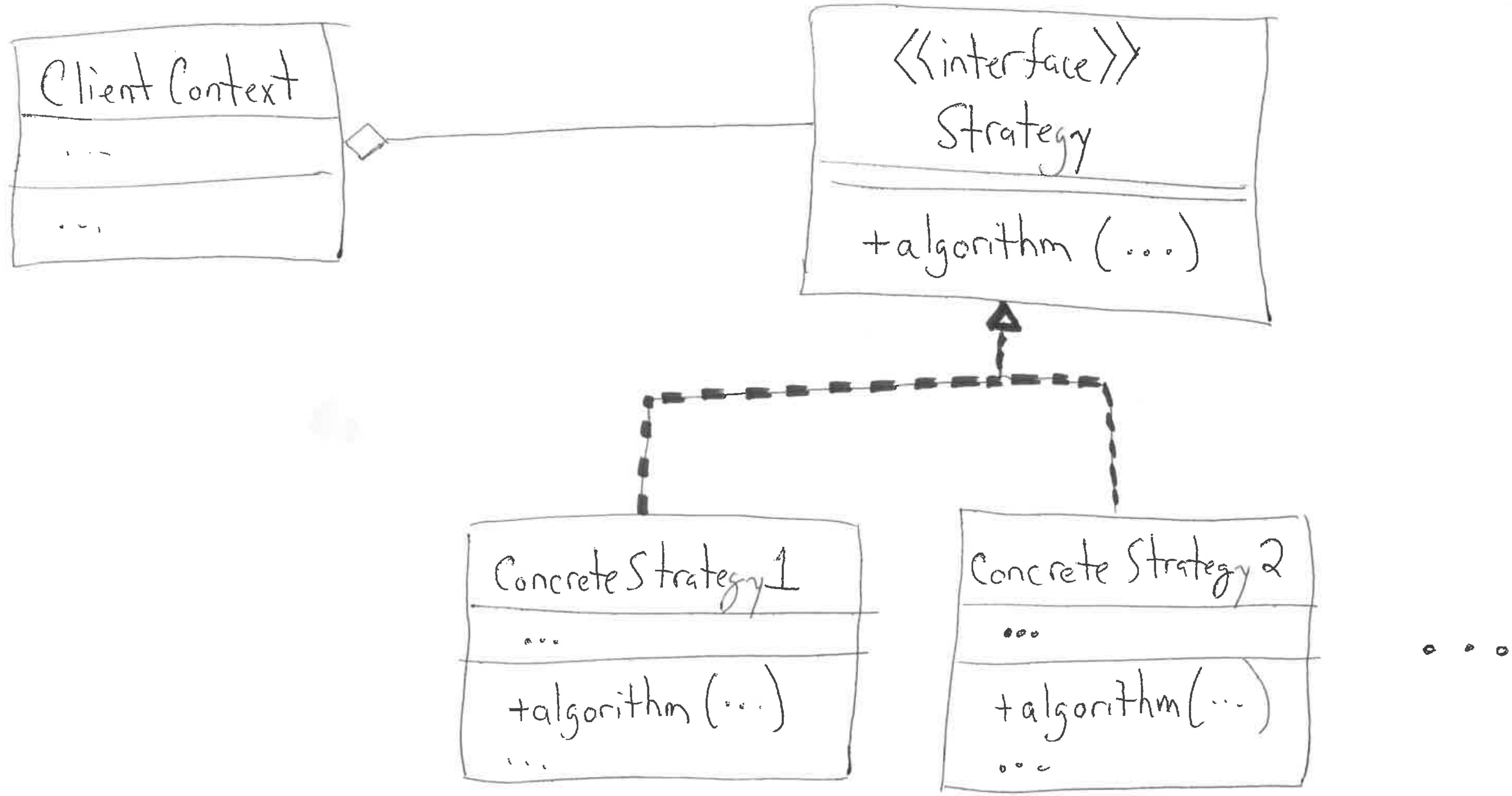
The wings are a private variable called wings.

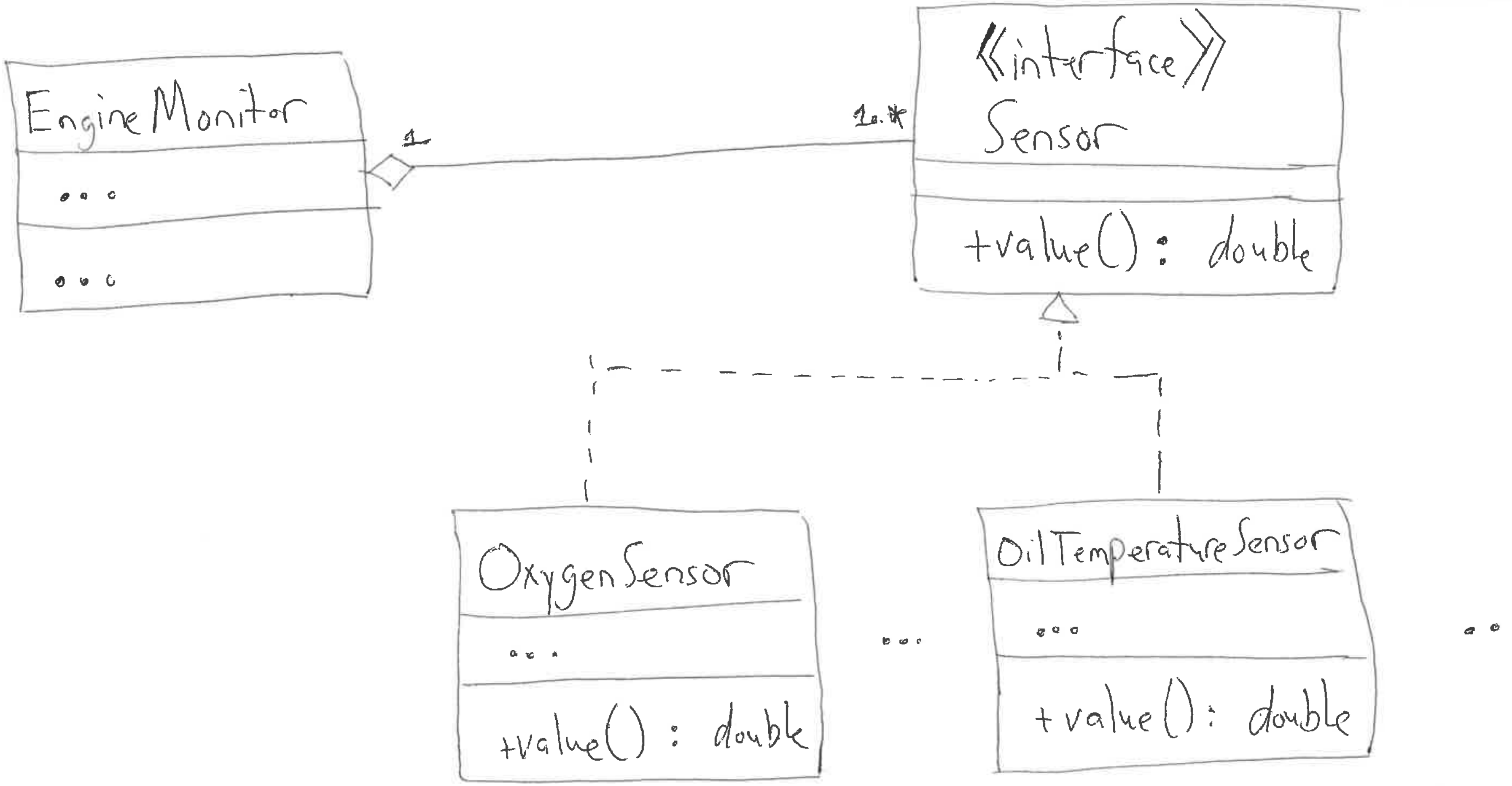


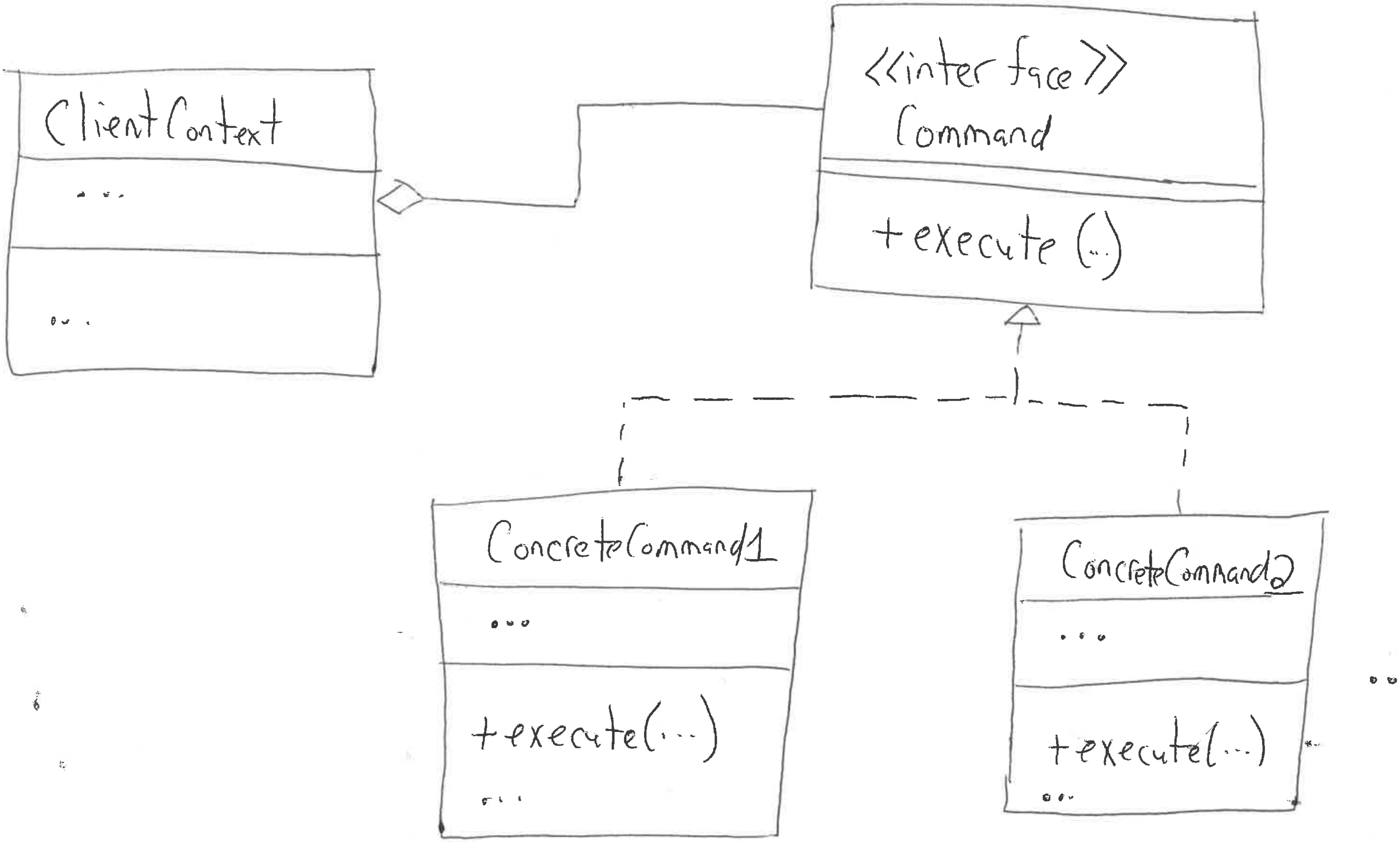
OrderSystem
...
...
1..*
<<interface>>
SalesTax
+computeTax(items : List<Item>) : long
FloridaSalesTax
+computeTax(items: List<Item>) : long
...
PittsburghSalesTax
+computeTax(items: ...)

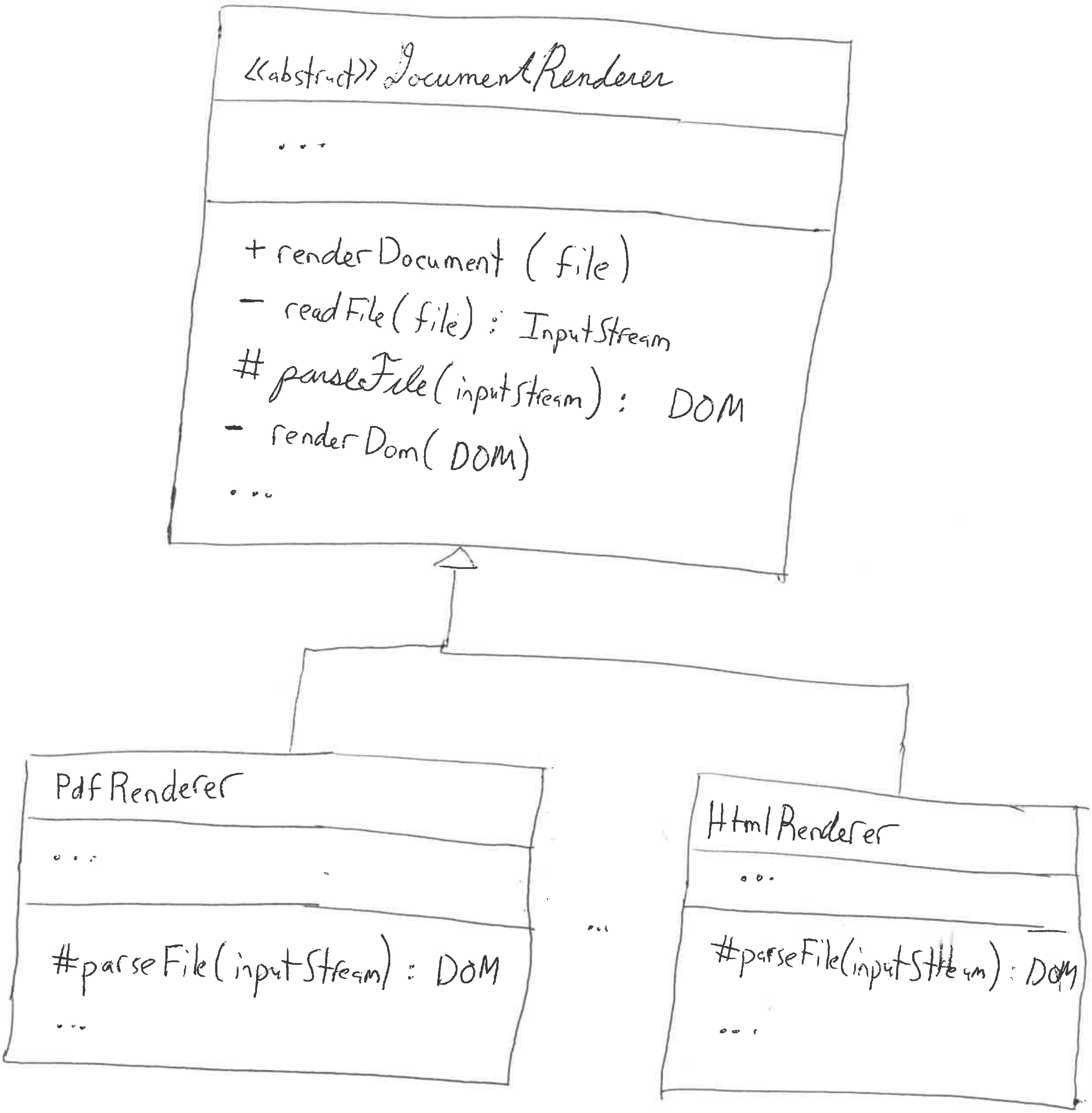


VisionProcessing System
...
...
1
1
<<interface>>
Line Detector
+detectLines(image:Image): List<Line>
AVLineDetector
...
+detectLines(image:Image): List<Line>
...
...
FaceLineDetector
...
+detectLines(image) ..
...
...

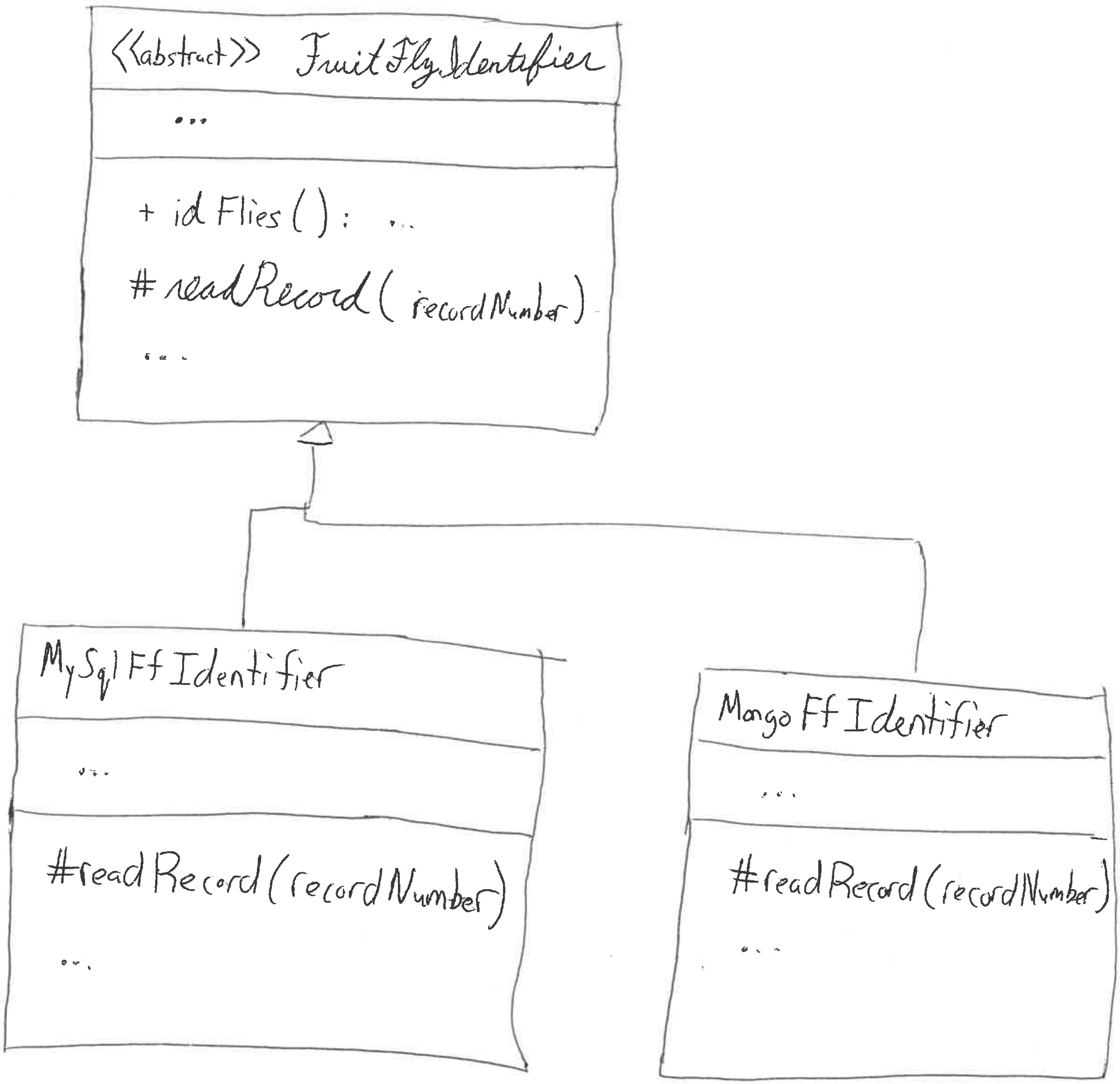

Client Context
...
...
<<interface>>
Strategy
+algorithm (...)
ConcreteStrategy1
...
+algorithm (...)
...
Concrete Strategy 2
...
+algorithm(...)
...
...


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

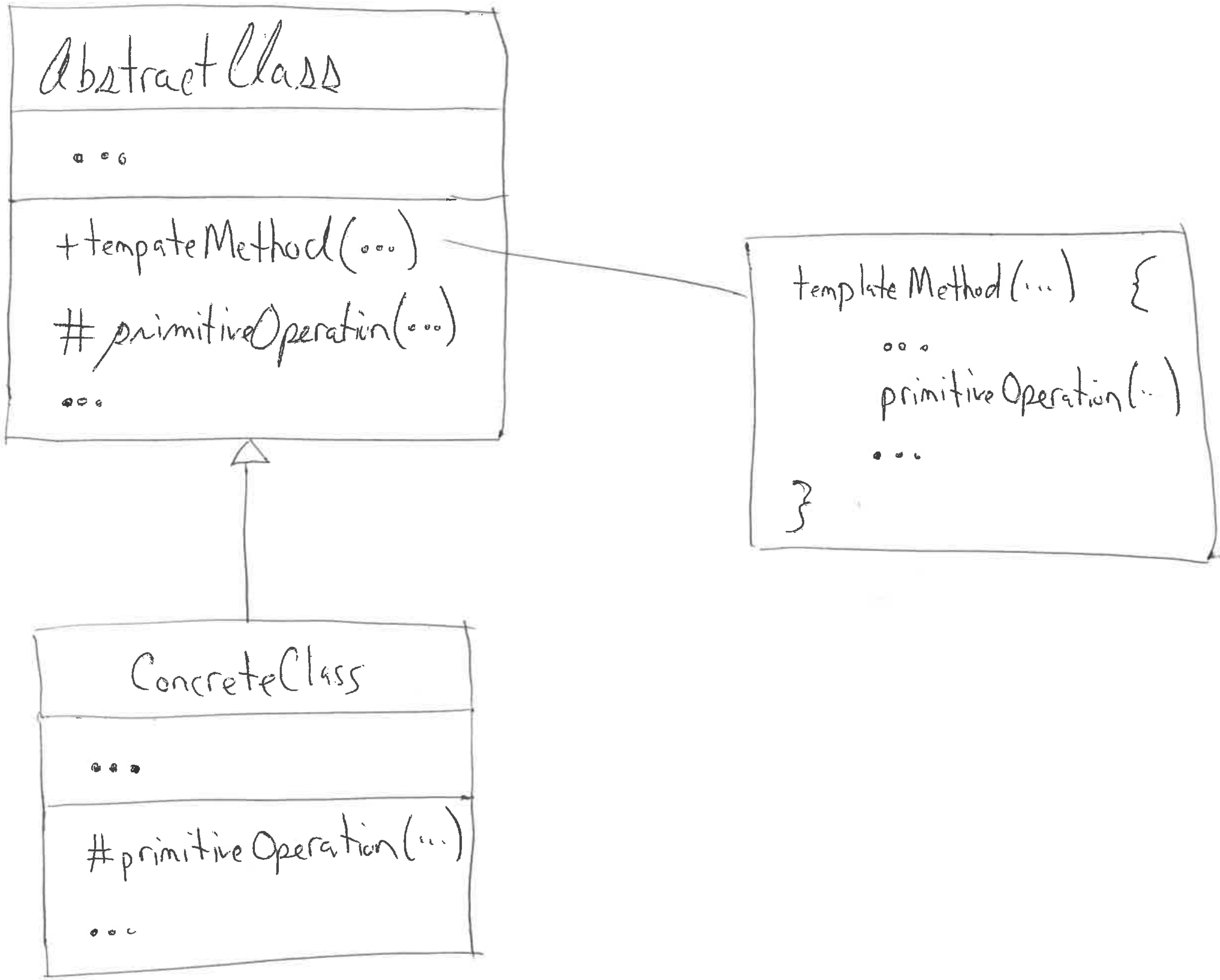

ClientContext
...
...
<<interface>>
Command
+execute(..)
ConcreteCommand1
...
+execute(...)
...
ConcreteCommand2
...
+execute(...)
...



<<abstract>> DocumentRenderer
...
+ renderDocument (file)
- readFile(file) : InputStream
parseFile(inputStream) : DOM
- renderDom(DOM)
...
PdfRenderer
...
#parseFile(inputStream) : DOM
...
...
HtmlRenderer
...
#parseFile(inputStream) : DOM
...


<<abstract>> FruitFlyIdentifier
...
+ idFlies(): ...
#readRecord(recordNumber)
...
MySqlFfIdentifier
...
#readRecord(recordNumber)
...
MongoFfIdentifier
...
#readRecord(recordNumber)
...

```java
public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c) {
        if (add(e)) {
            modified = true;
        }
    }
    return modified;
}
```


AbstractClass
...
+tempateMethod(...)
#primitiveOperation(...)
...
templateMethod(...) {
...
primitiveOperation(..)
...
}
ConcreteClass
...
#primitiveOperation(...)
...