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

## Design Patterns, Part II

Charlie Garrod **Chris Timperley**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

# Administrivia

- Reading due today:  UML and Patterns Chapters 9 and 10
- Optional reading for Thursday:
  - UML and Patterns Chapter 17
  - Effective Java items 49, 54, and 69
- Homework 3 due Thursday at 11:59 p.m.
- Midterm exam next Thursday (September 26th)
  - Review session, practice exam info coming soon

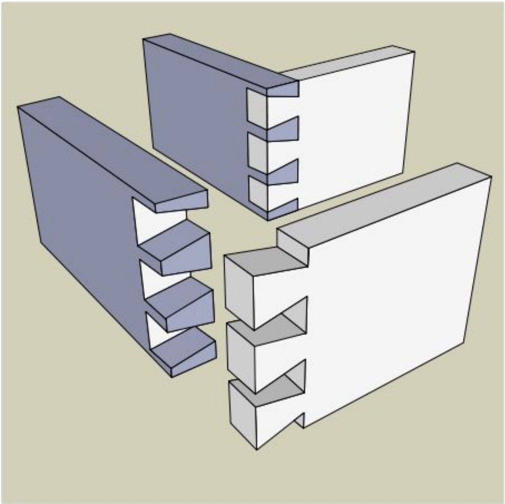
# Key concepts from last Thursday

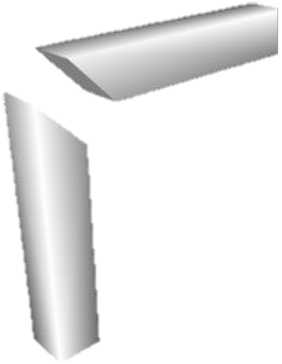
# UML you should know

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

# 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?"





# 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

# 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

# Today

- More design patterns for reuse
  - Template method pattern
  - Iterator pattern
  - Decorator pattern
- Design goals and design principles

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

# 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

- More design patterns for reuse
  - Template method pattern
  - Iterator pattern
  - Decorator pattern
- Design goals and design principles

# 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.: (Prefer the for-each loop!)

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

# Getting an Iterator

```
public interface Collection<E> extends Iterable<E> {
  boolean     add(E e);
  boolean     addAll(Collection<? extends E> c);
  boolean     remove(Object e);
  boolean     removeAll(Collection<?> c);
  boolean     retainAll(Collection<?> c);
  boolean     contains(Object e);
  boolean     containsAll(Collection<?> c);
  void        clear();
  int         size();
  boolean     isEmpty();
  Iterator<E> iterator();
  Object[]    toArray()
  <T> T[]     toArray(T[] a);
  …
}
```

*Defines an interface for creating an Iterator, but allows Collection implementation to decide which Iterator to create.*

# An Iterator implementation for Pairs

```
public class Pair<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second = s; }









}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

# An Iterator implementation for Pairs

```
public class Pair<E> implements Iterable<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second = s; }
  public Iterator<E> iterator() {
    return new PairIterator();
  }
  private class PairIterator implements Iterator<E> {
    private boolean seenFirst = false, seenSecond = false;
    public  boolean hasNext() { return !seenSecond; }
    public  E next() {
      if (!seenFirst)  { seenFirst  = true; return first;  }
      if (!seenSecond) { seenSecond = true; return second; }
      throw new NoSuchElementException();
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

# Iterator design pattern

- Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
  - Order is unspecified, but access every element once
- Solution: A strategy pattern for iteration
- Consequences:
  - Hides internal implementation of underlying container
  - Easy to change container type
  - Facilitates communication between parts of the program

# Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable…
- …but their `Iterator` implementations assume the collection does not change while the `Iterator` is being used
  - You will get a `ConcurrentModificationException`

# Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable…
- …but their `Iterator` implementations assume the collection does not change while the `Iterator` is being used
  - You will get a `ConcurrentModificationException`
  - If you simply want to remove an item:

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
      it.hasNext();  ) {
   String s = it.next();
   if (s.equals("Chris"))
     arguments.remove("Chris"); // runtime error
}
```

# Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable…
- …but their `Iterator` implementations assume the collection does not change while the `Iterator` is being used
  - You will get a `ConcurrentModificationException`
  - If you simply want to remove an item:

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
     it.hasNext();  ) {
  String s = it.next();
  if (s.equals("Chris"))
    it.remove();
}
```

# Today

- More design patterns for reuse
  - Template method pattern
  - Iterator pattern
  - Decorator pattern
- Design goals and design principles

# Limitations of inheritance

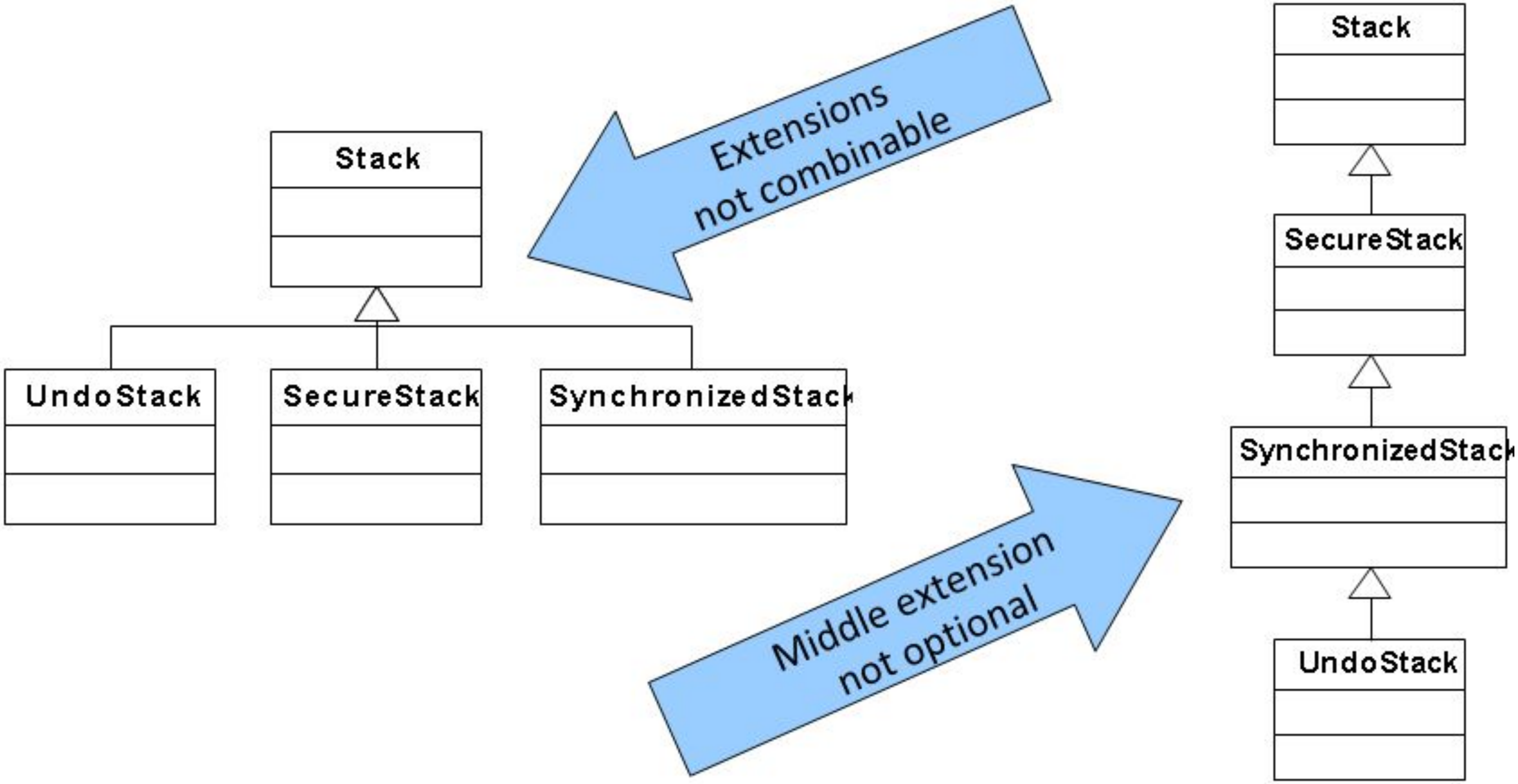
- Suppose you want various extensions of a `Stack` data structure…
  - `UndoStack`: A stack that lets you undo previous push or pop operations
  - `SecureStack`: A stack that requires a password
  - `SynchronizedStack`: A stack that serializes concurrent accesses

# Limitations of inheritance

- Suppose you want various extensions of a `Stack` data structure…
  - `UndoStack`: A stack that lets you undo previous push or pop operations
  - `SecureStack`: A stack that requires a password
  - `SynchronizedStack`: A stack that serializes concurrent accesses
  - `SecureUndoStack`: A stack that requires a password, and also lets you undo previous operations
  - `SynchronizedUndoStack`: A stack that serializes concurrent accesses, and also lets you undo previous operations
  - `SecureSynchronizedStack`: …
  - `SecureSynchronizedUndoStack`: …

**Goal: arbitrarily composable extensions**

# Limitations of inheritance



# Workarounds?

- Combining inheritance hierarchies?
- Multiple inheritance?

# The decorator design pattern

- Problem: You need arbitrary or dynamically composable extensions to individual objects.
- Solution: Implement a common interface as the object you are extending, add functionality, but delegate primary responsibility to an underlying object.
- Consequences:
  - More flexible than static inheritance
  - Customizable, cohesive extensions
  - Breaks object identity, self-references

# Decorators use both subtyping and delegation

```
public class LoggingList<E> implements List<E> {
  private final List<E> list;
  public LoggingList<E>(List<E> list) { this.list = list; }
  public boolean add(E e) {
      System.out.println("Adding " + e);
      return list.add(e);
  }
  public E remove(int index) {
      System.out.println("Removing at " + index);
      return list.remove(index);
  }
  …
```

# The AbstractStackDecorator forwarding class

```
public abstract class AbstractStackDecorator
           implements Stack {
  private final Stack stack;
  public AbstractStackDecorator(Stack stack) {
      this.stack = stack;
  }
  public void push(Item e) {
   stack.push(e);
  }
  public Item pop() {
   return stack.pop();
  }
  …
}
```

# The concrete decorator classes

```
public class UndoStack extends AbstractStackDecorator
    implements Stack {
  private final UndoLog log = new UndoLog();
  public UndoStack(Stack stack) { super(stack); }
  public void push(Item e) {
    log.append(UndoLog.PUSH, e);
    super.push(e);
  }
  …
}
```

# Using the decorator classes

- To construct a plain stack:

```
Stack stack = new ArrayStack();
```

- To construct an undo stack:

# Using the decorator classes

- To construct a plain stack:

```
Stack stack = new ArrayStack();
```

- To construct an undo stack:

```
UndoStack stack = new UndoStack(new ArrayStack());
```

# Using the decorator classes

- To construct a plain stack:

```
Stack stack = new ArrayStack();
```

- To construct an undo stack:

```
UndoStack stack = new UndoStack(new ArrayStack());
```

- To construct a secure synchronized undo stack:

# Using the decorator classes

- To construct a plain stack:

```
Stack s = new ArrayStack();
```

- To construct an undo stack:

```
UndoStack s = new UndoStack(new ArrayStack());
```

- To construct a secure synchronized undo stack:

```
SecureStack s = new SecureStack(new SynchronizedStack(
                new UndoStack(new ArrayStack())));
```

# Decorators from `java.util.Collections`

- Turn a mutable collection into an immutable collection:
```
static List<T>  unmodifiableList(List<T> lst);
static Set<T>   unmodifiableSet(Set<T>   set);
static Map<K,V> unmodifiableMap(Map<K,V> map);
```
- Similar for synchronization:
```
static List<T>  synchronizedList(List<T> lst);
static Set<T>   synchronizedSet(Set<T>   set);
static Map<K,V> synchronizedMap(Map<K,V> map);
```

# The UnmodifiableCollection (simplified excerpt)

```
public static <T> Collection<T> unmodifiableCollection(Collection<T> c) {
    return new UnmodifiableCollection<>(c);
}
class UnmodifiableCollection<E> implements Collection<E>, Serializable {
    final Collection<E> c;
    UnmodifiableCollection(Collection<> c) { this.c = c; }
    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}
    public boolean add(E e) {throw new UnsupportedOperationException();   }
    public boolean remove(Object o) { throw new UnsupportedOperationException();   }
    public boolean containsAll(Collection<?> coll) {  return c.containsAll(coll);   }
    public boolean addAll(Collection<? extends E> coll) {
        throw new UnsupportedOperationException();
    }
    public boolean removeAll(Collection<?> coll) {
        throw new UnsupportedOperationException();
    }
    public boolean retainAll(Collection<?> coll) {
        throw new UnsupportedOperationException();
    }
    public void clear() {  throw new UnsupportedOperationException();   }
}
```

# The decorator pattern vs. inheritance

- Decorator composes features at run time
  - Inheritance composes features at compile time
- Decorator consists of multiple collaborating objects
  - Inheritance produces a single, clearly-typed object
- Can mix and match multiple decorations
  - Multiple inheritance is conceptually difficult

# Today

- More design patterns for reuse
  - Template method pattern
  - Iterator pattern
  - Decorator pattern
- Design goals and design principles

# Metrics of software quality, i.e., *design goals*

| | |
|---|---|
| Functional correctness | Adherence of implementation to the specifications |
| Robustness | Ability to handle anomalous events |
| Flexibility | Ability to accommodate changes in specifications |
| Reusability | Ability to be reused in another application |
| Efficiency | Satisfaction of speed and storage requirements |
| Scalability | Ability to serve as the basis of a larger version of the application |
| Security | Level of consideration of application security |

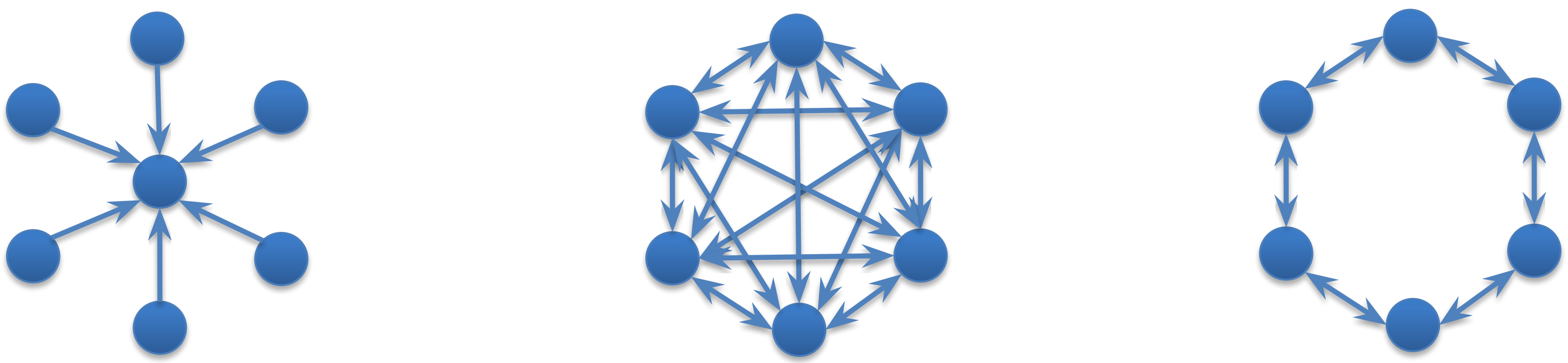
**Source: Braude, Bernstein, Software Engineering. Wiley 2011**

# Design principles: heuristics to achieve design goals

- Low coupling
- Low representational gap
- High cohesion

# A design principle for reuse: *low coupling*

- Each component should depend on as few other components as possible



- Benefits of low coupling:
  - Enhances understandability
  - Reduces cost of change
  - Eases reuse

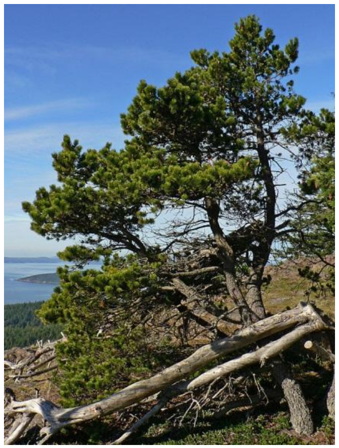
# Law of Demeter (Principle of Least Knowledge)

- "Only talk to your immediate friends"

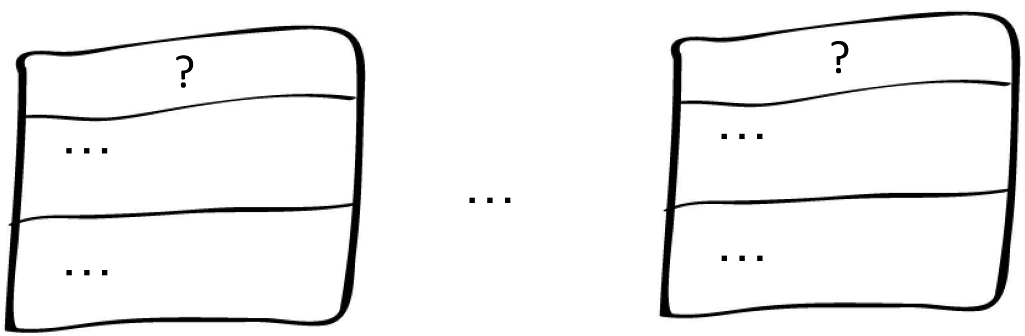
~~foo.bar().baz().quz(42)~~

# Representational gap

- Real-world concepts:

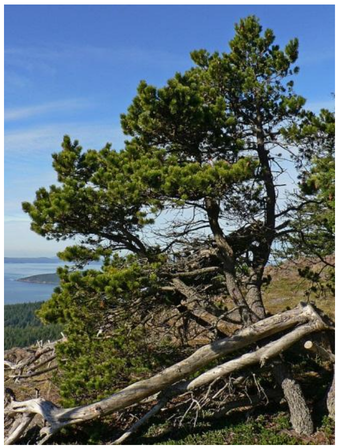


- Software concepts:

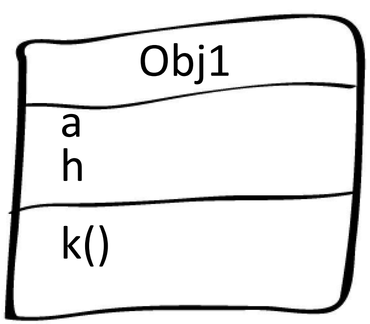


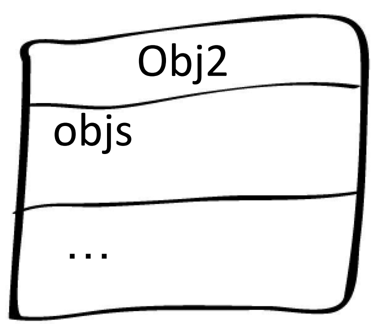
# Representational gap

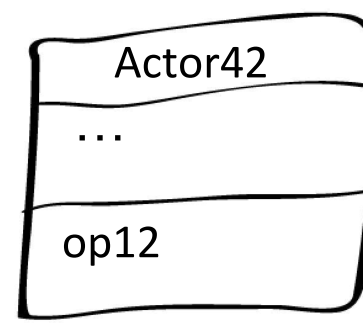
- Real-world concepts:



- Software concepts:

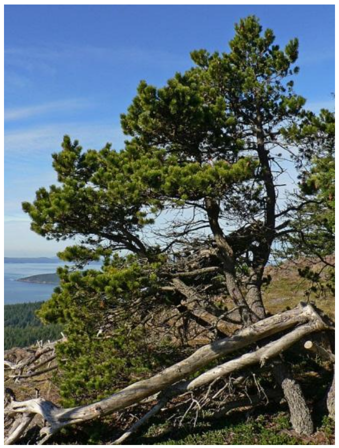




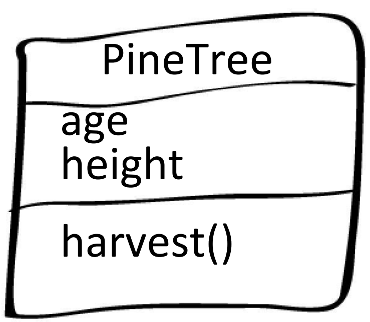


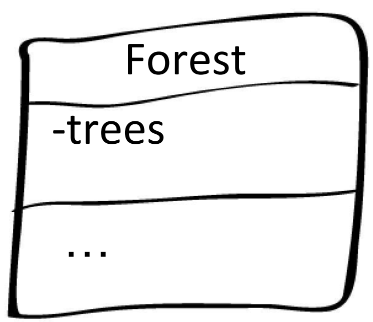
# Representational gap

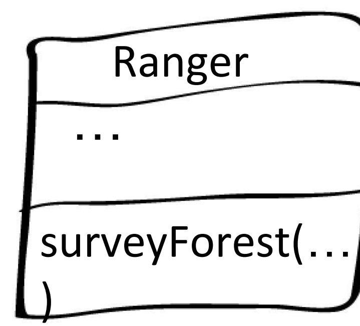
- Real-world concepts:



- Software concepts:



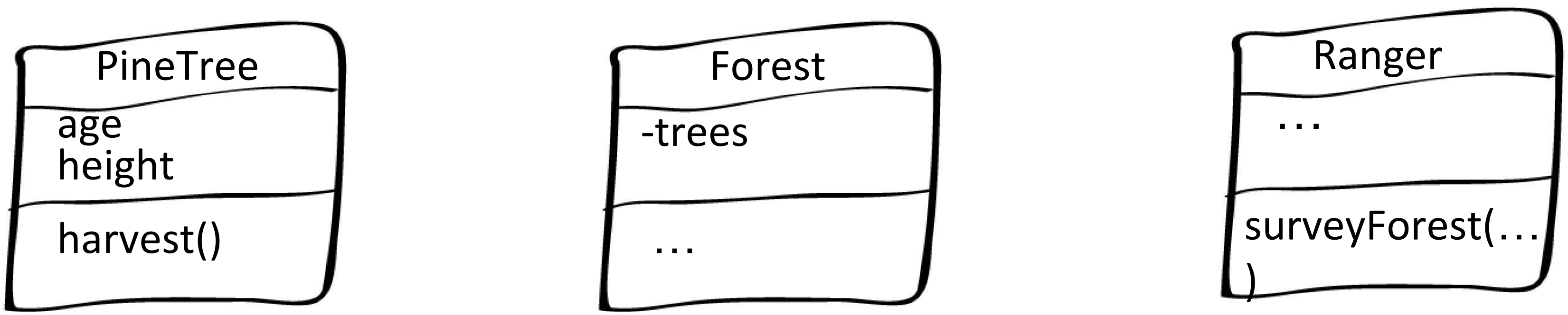



# Benefits of low representational gap

- Facilitates understanding of design and implementation
- Facilitates traceability from problem to solution
- Facilitates evolution

# A related design principle: high cohesion

- Each component should have a small set of closely-related responsibilities
- Benefits:
  - Facilitates understandability
  - Facilitates reuse
  - Eases maintenance

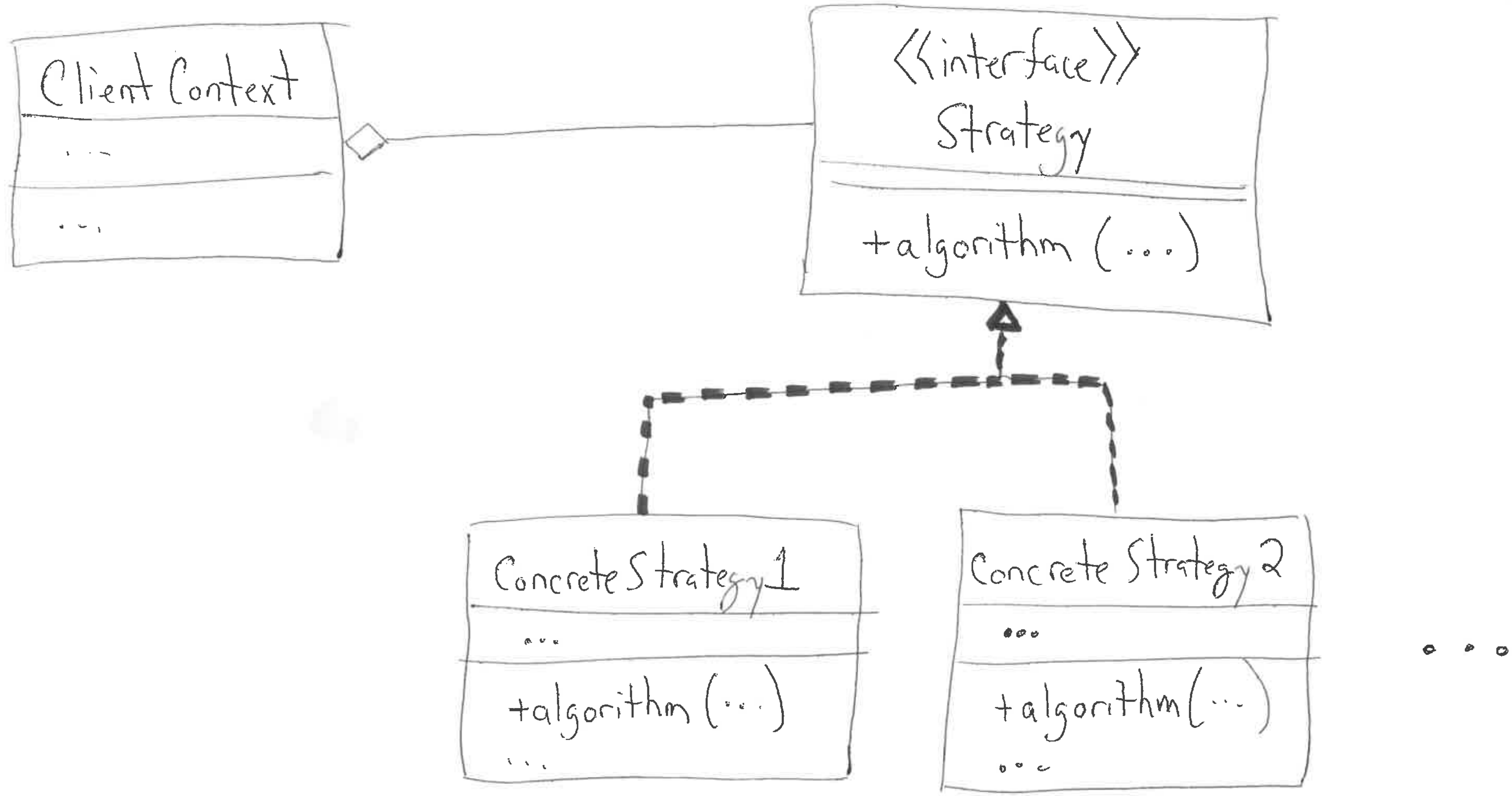


# Coupling vs. cohesion

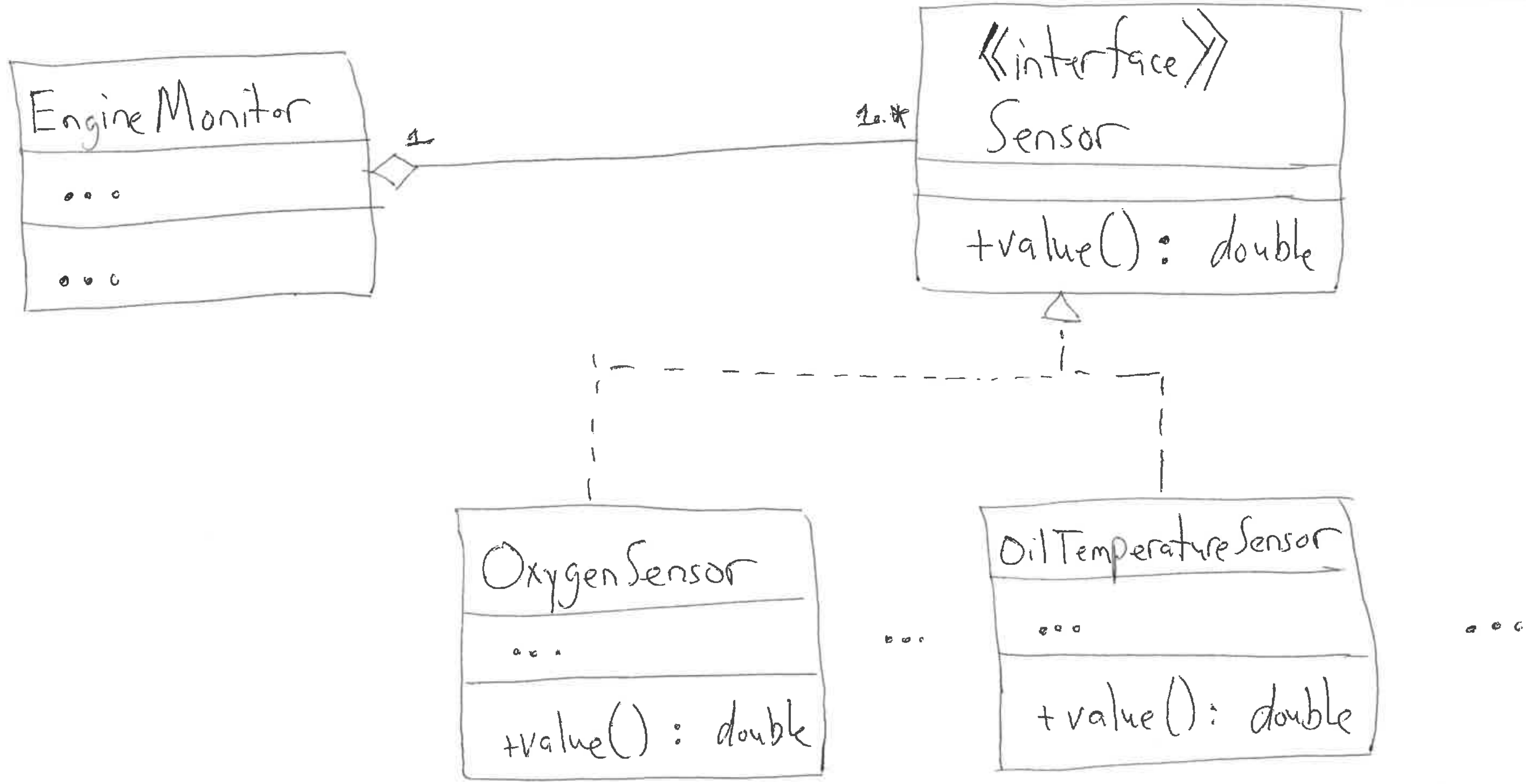
- All code in one component?
  - Low cohesion, low coupling
- Every statement / method in a separate component?
  - High cohesion, high coupling

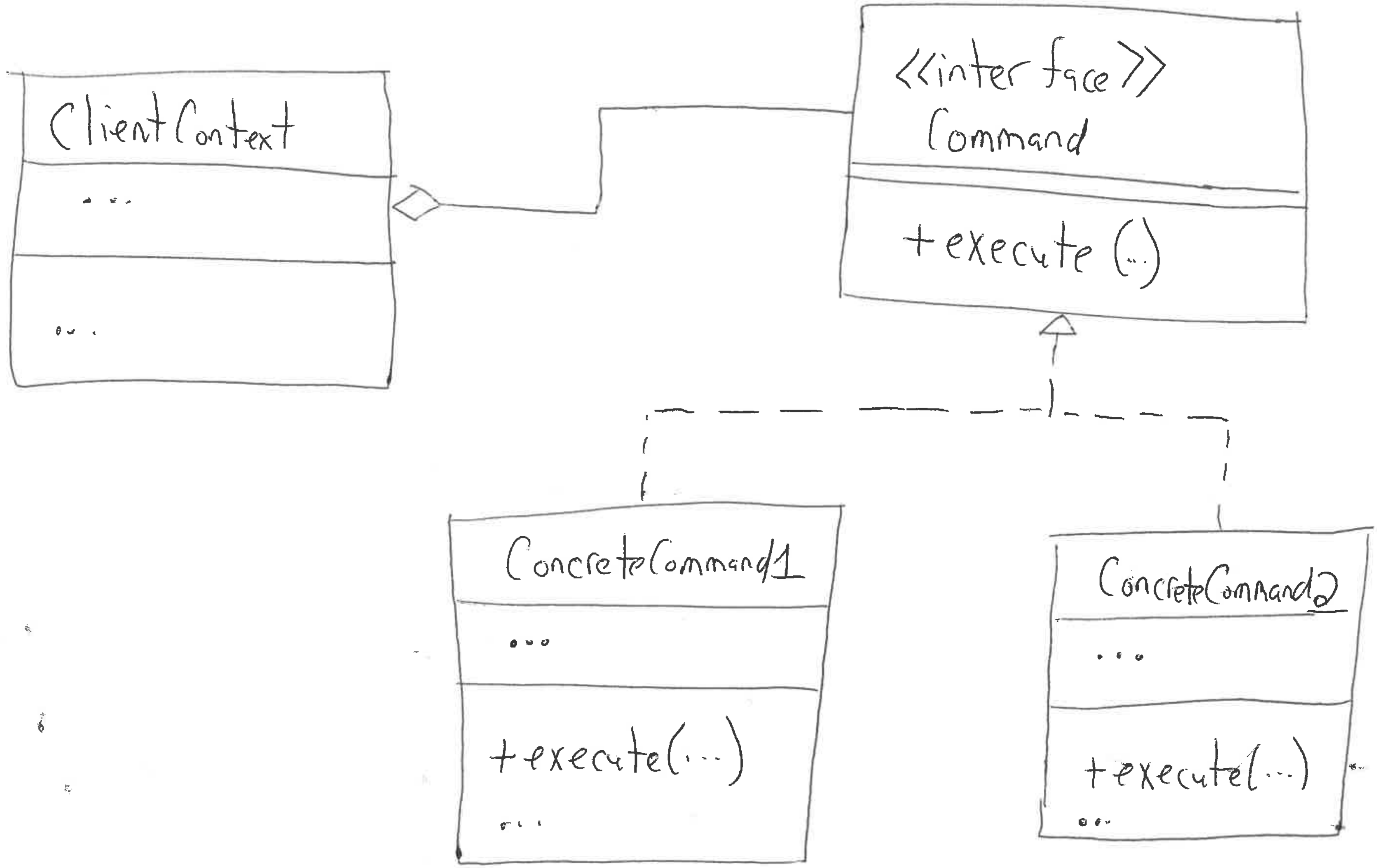
# Summary

- Five design patterns to facilitate reuse…
- Design principles are useful heuristics
  - Reduce coupling to increase understandability, reuse
  - Lower representational gap to increase understandability, maintainability
  - Increase cohesion to increase understandability

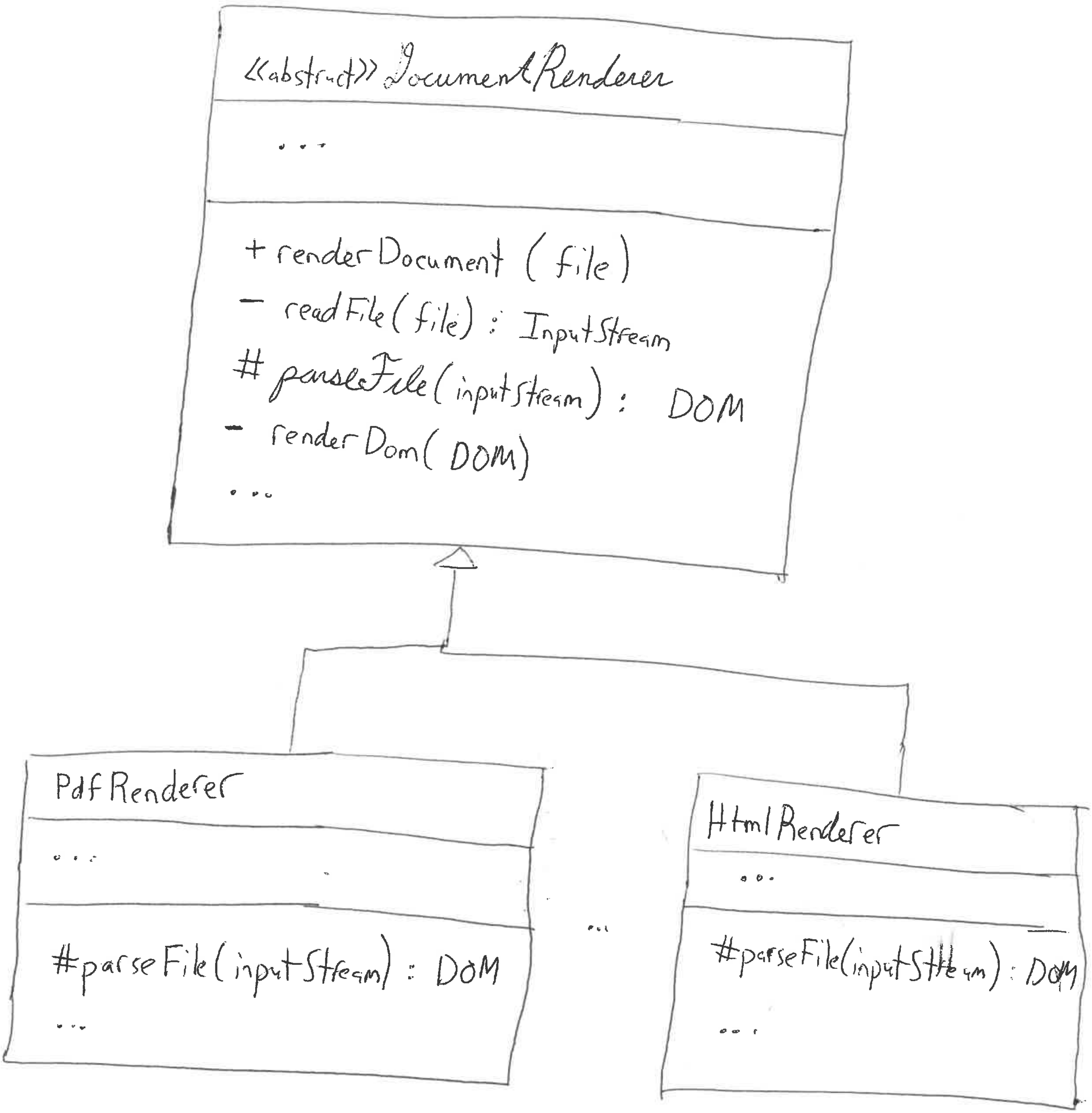


Client Context
...
...
<<interface>>
Strategy
+algorithm (...)
Concrete Strategy 1
...
+algorithm (...)
...
Concrete Strategy 2
...
+algorithm(...)
...
...

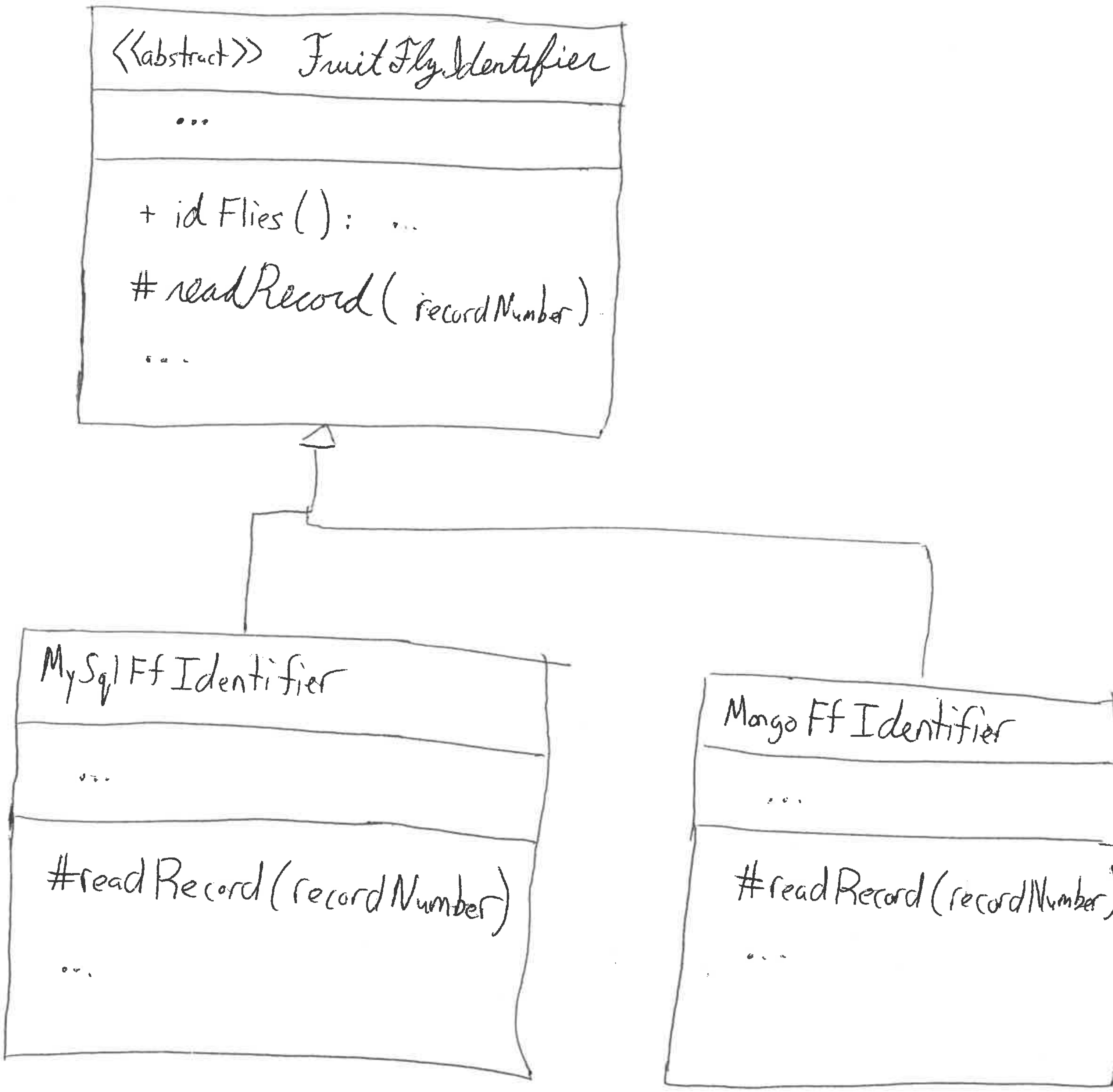

EngineMonitor
...
...
1
1..*
«interface»
Sensor
+value() : double
OxygenSensor
...
+value() : double
...
OilTemperatureSensor
...
+value() : double
...



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

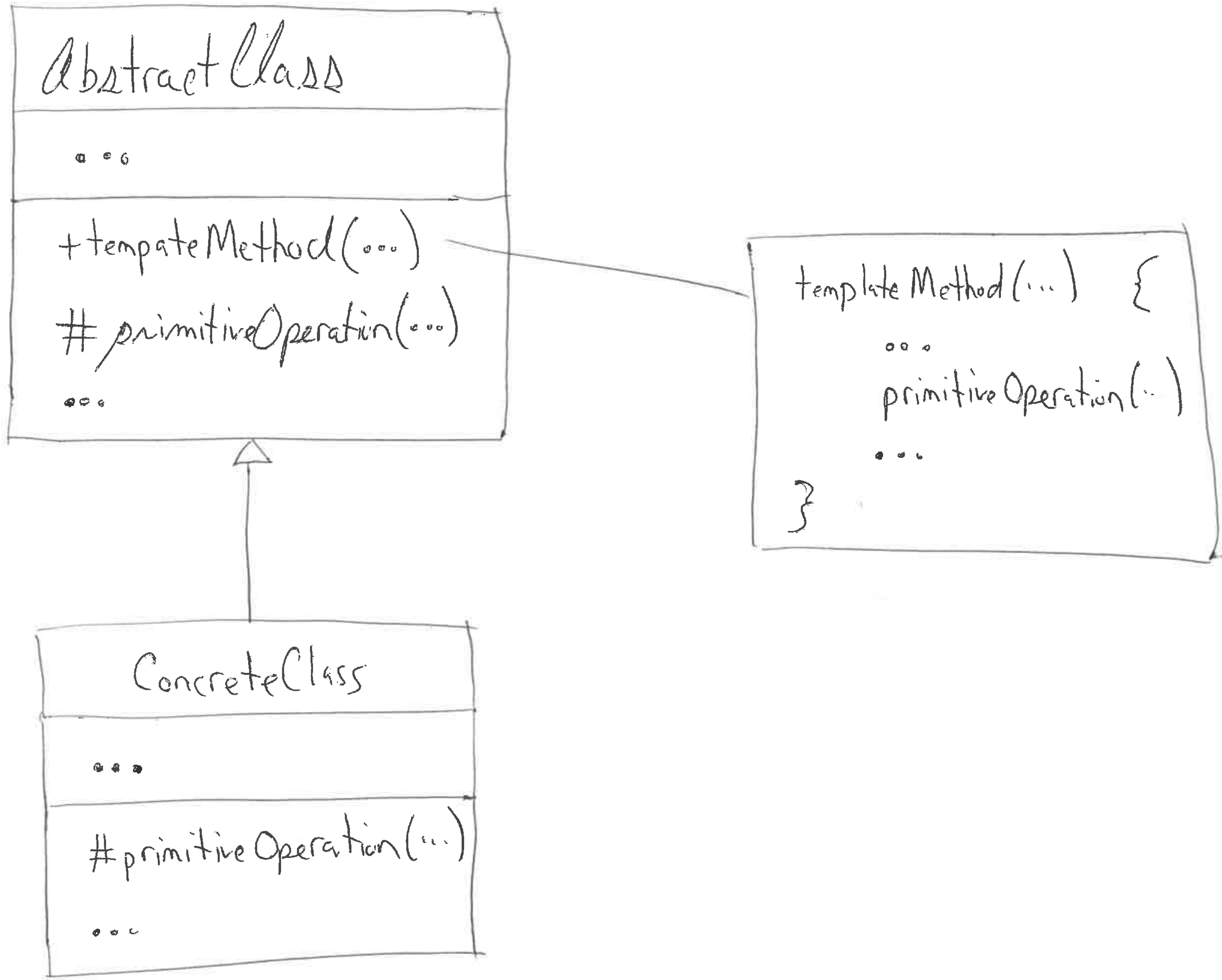


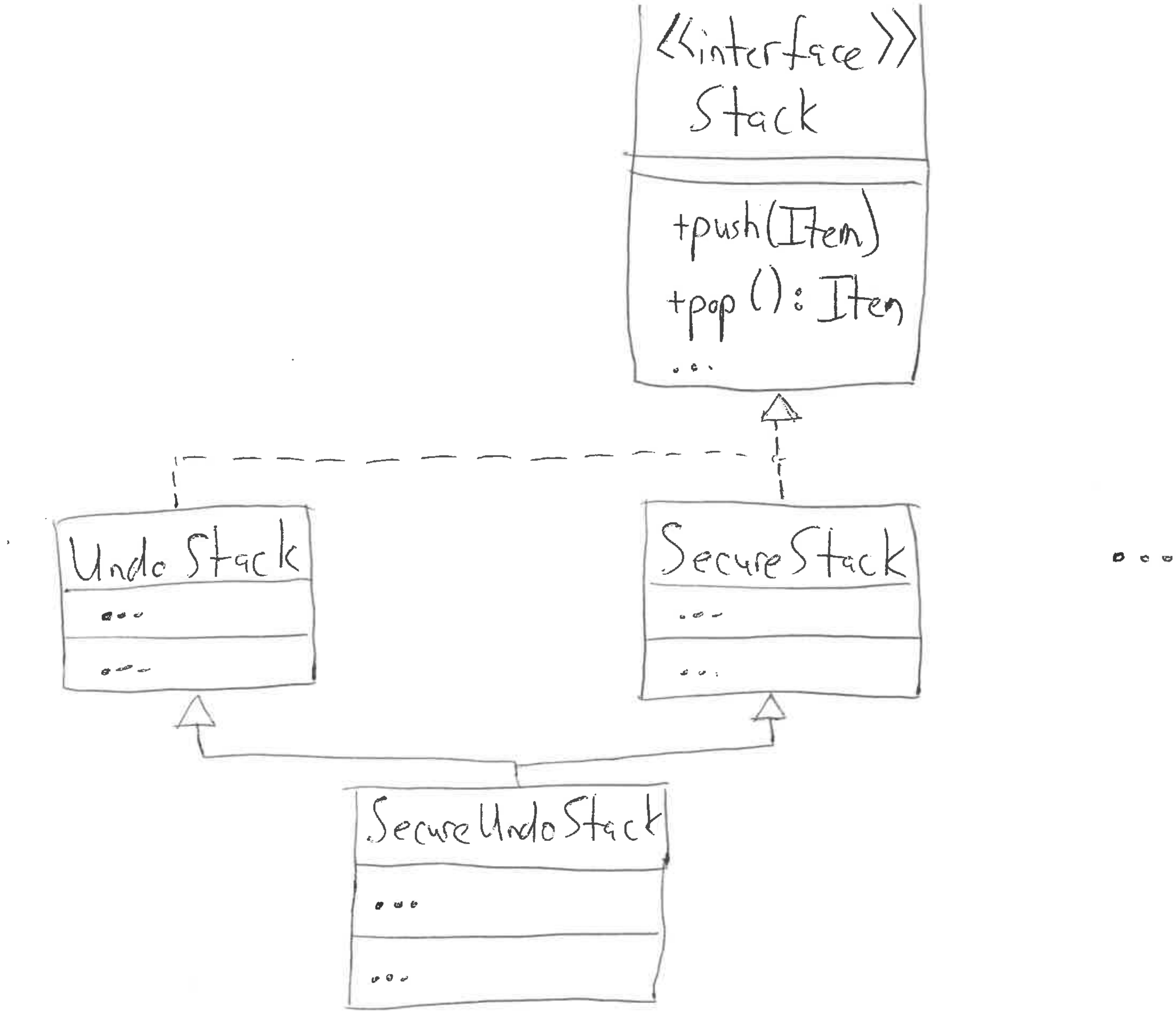
«abstract» DocumentRenderer
...
+ renderDocument ( file )
- readFile( file) : InputStream
# parseFile( inputStream) : DOM
- renderDom( DOM)
...
PdfRenderer
...
#parseFile(inputStream) : DOM
...
...
HtmlRenderer
...
#parseFile(inputStream) : DOM
...

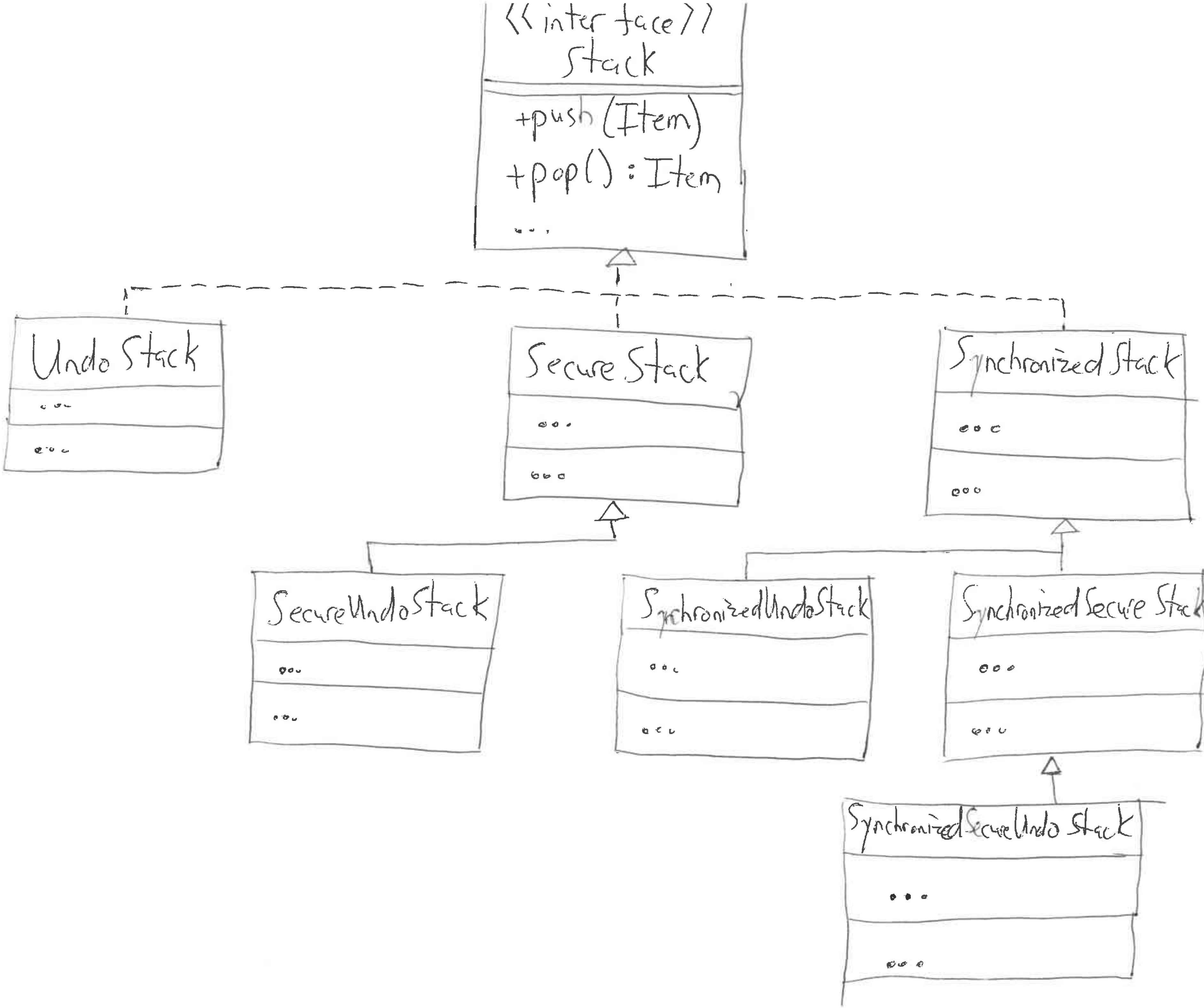


<<abstract>> FruitFlyIdentifier
...
+ idFlies(): ...
# readRecord(recordNumber)
...
MySqlFfIdentifier
...
#readRecord(recordNumber)
...
MongoFfIdentifier
...
#readRecord(recordNumber)
...

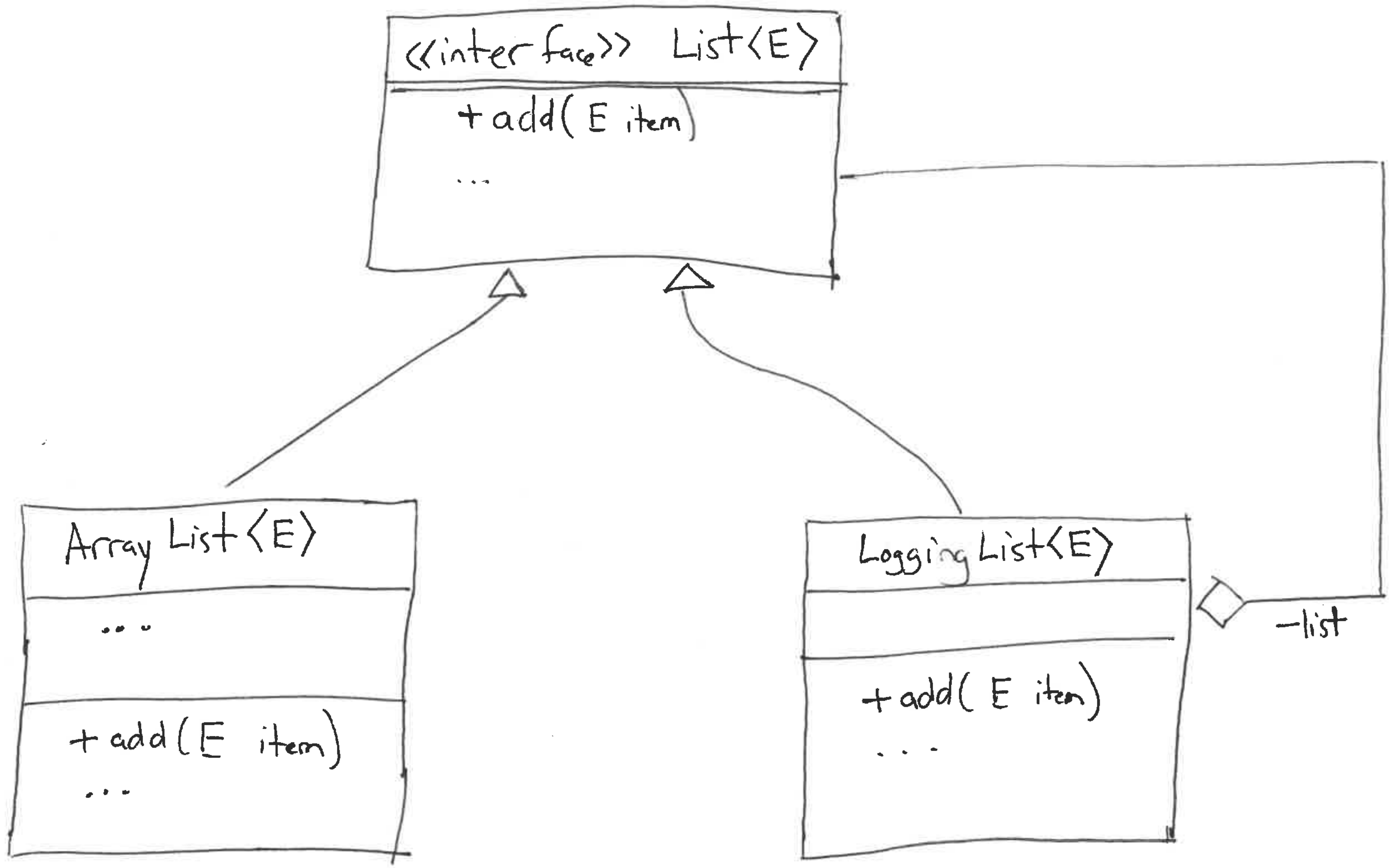
```java
public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c) {
        if (add(e)) {
            modified = true;
        }
    }
    return modified;
}
```

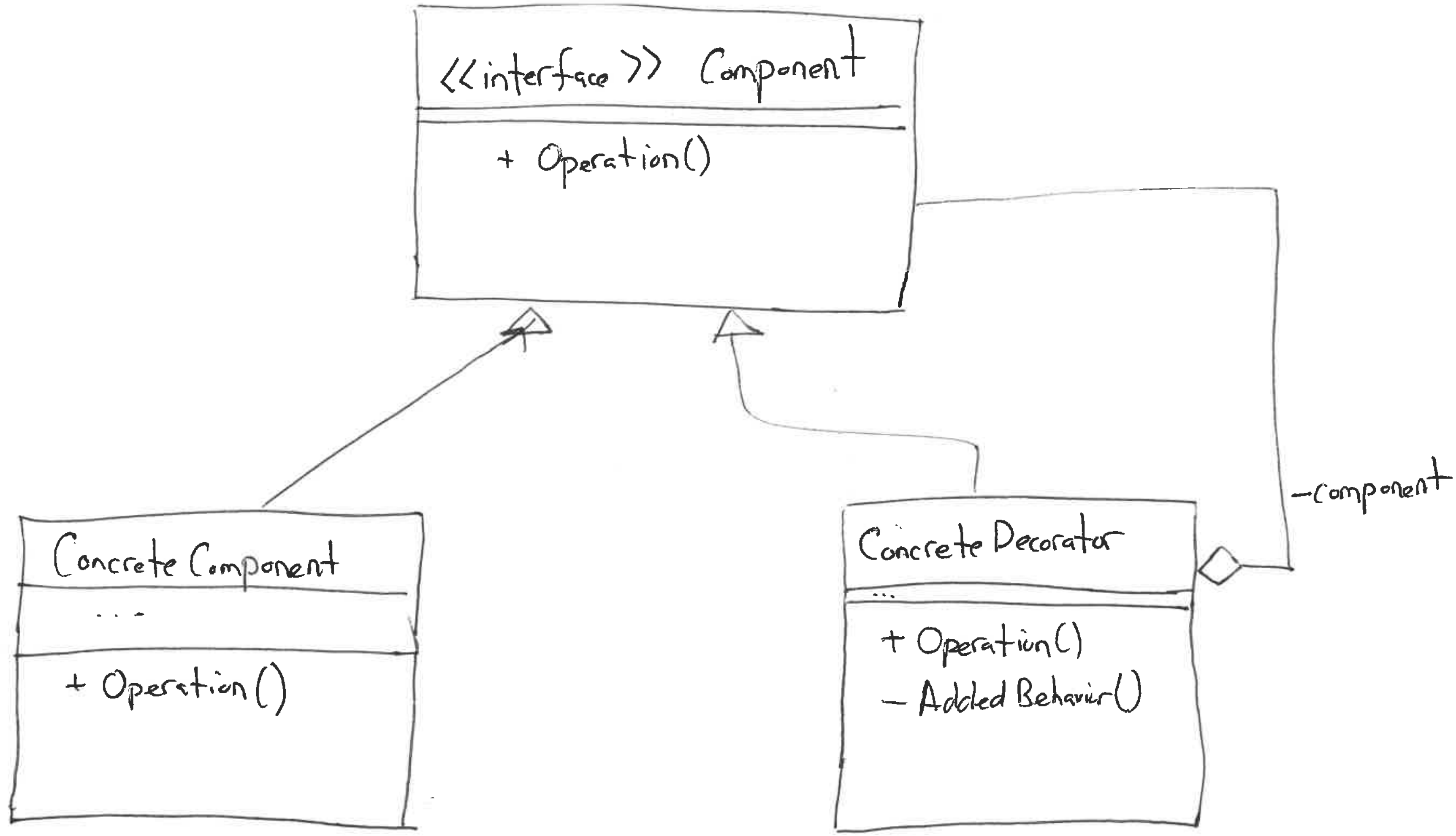

AbstractClass
...
+tempateMethod(...)
#primitiveOperation(...)
...
templateMethod(...) {
...
primitiveOperation(..)
...
}
ConcreteClass
...
#primitiveOperation(...)
...


«interface»
Stack
+push(Item)
+pop(): Item
...
UndoStack
...
...
SecureStack
...
...
SecureUndoStack
...
...
...

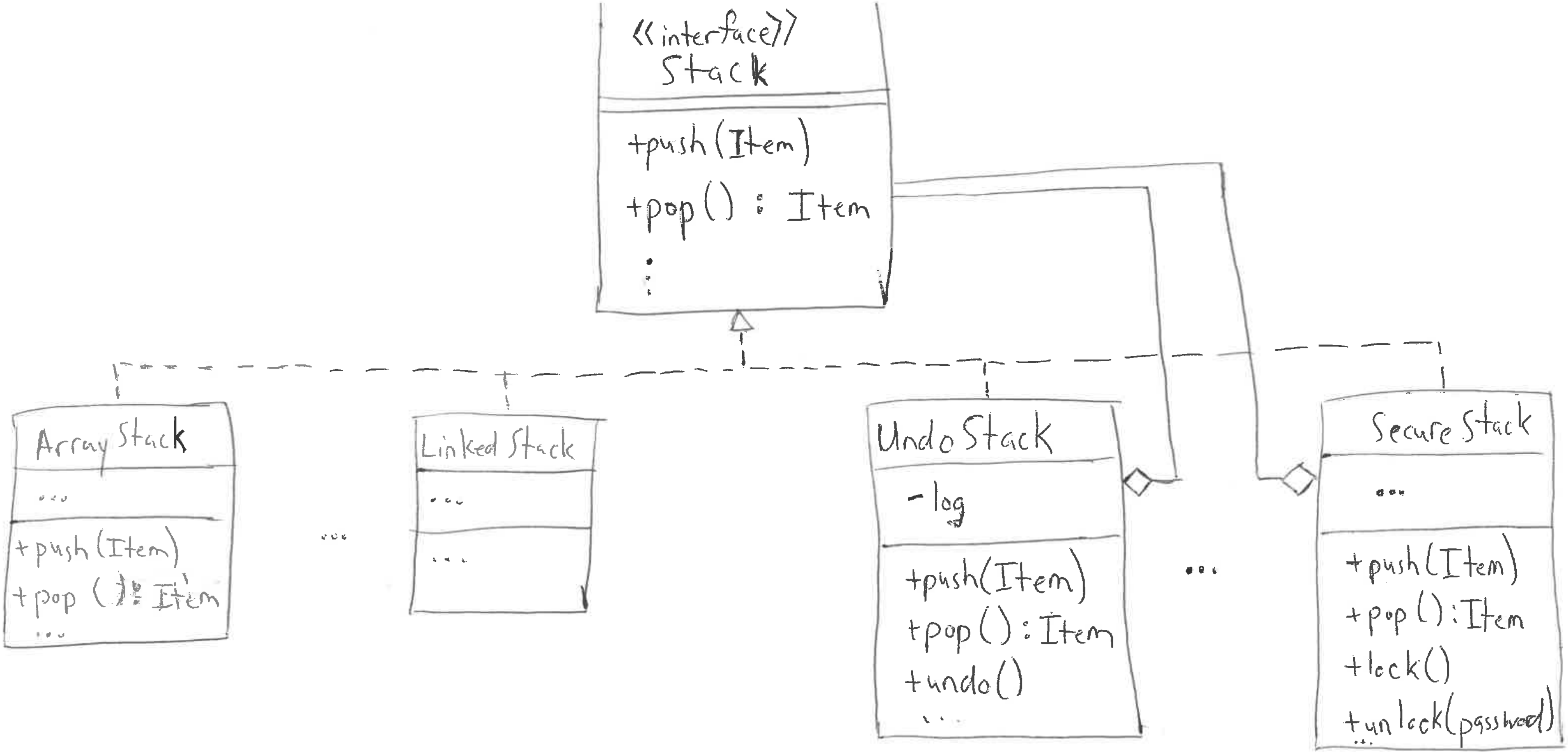


<<interface>>
Stack
+push(Item)
+pop() : Item
...
UndoStack
...
...
SecureStack
...
...
SynchronizedStack
...
...
SecureUndoStack
...
...
SynchronizedUndoStack
...
...
SynchronizedSecureStack
...
...
SynchronizedSecureUndoStack
...
...

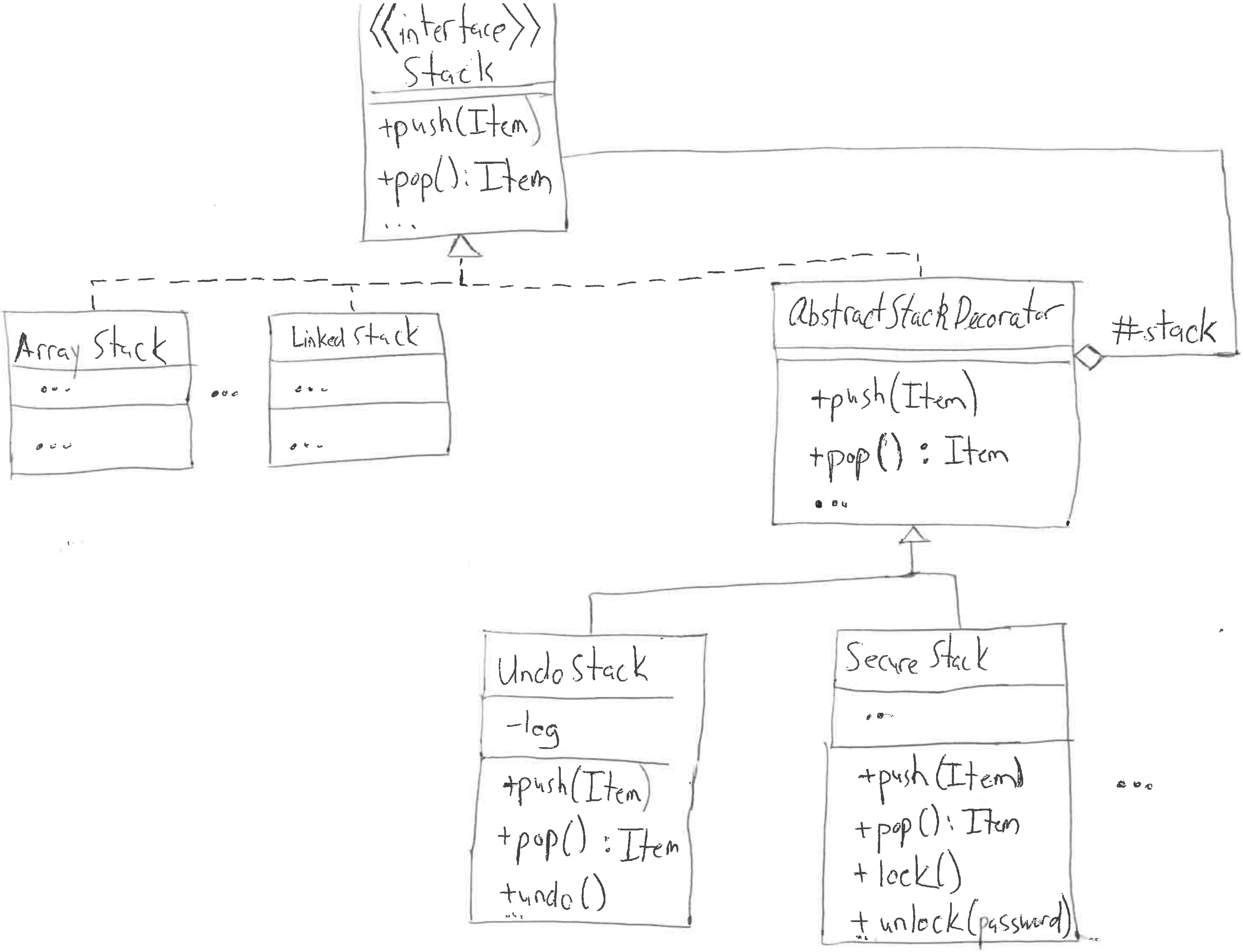

<<interface>> List<E>
+add(E item)
...
ArrayList<E>
...
+add(E item)
...
LoggingList<E>
+add(E item)
...
-list



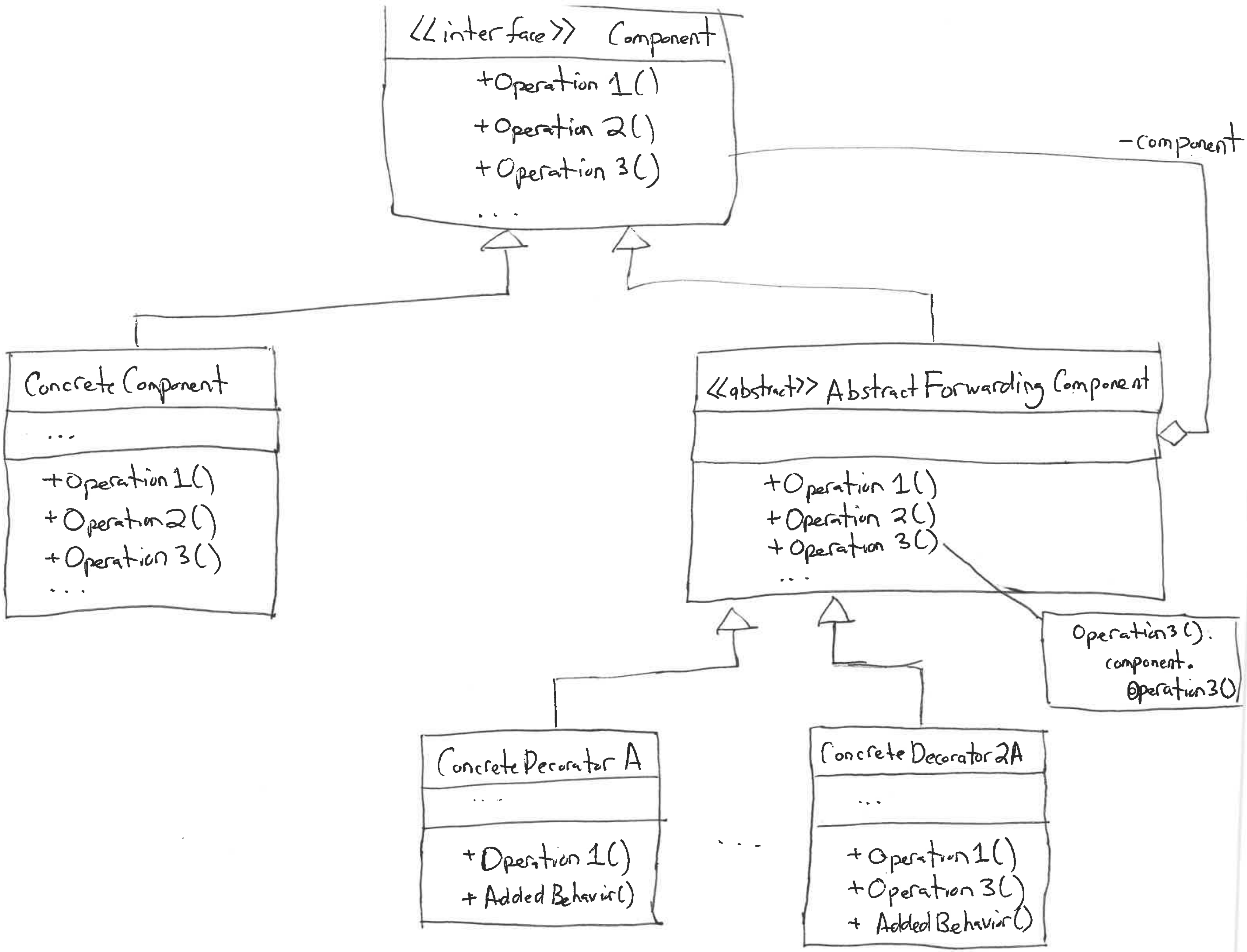
<<interface>> Component
+ Operation()
Concrete Component
...
+ Operation()
Concrete Decorator
...
+ Operation()
- Added Behavior()
-component



<<interface>>
Stack
+push(Item)
+pop() : Item
⋮
ArrayStack
...
+push(Item)
+pop() : Item
...
...
LinkedStack
...
...
UndoStack
-log
+push(Item)
+pop() : Item
+undo()
...
...
SecureStack
...
+push(Item)
+pop() : Item
+lock()
+unlock(password)
...



<<interface>>
Stack
+push(Item)
+pop(): Item
...
Array Stack
...
...
...
Linked Stack
...
...
AbstractStackDecorator
+push(Item)
+pop() : Item
...
#stack
Undo Stack
-log
+push(Item)
+pop() : Item
+undo()
...
Secure Stack
...
+push(Item)
+pop(): Item
+lock()
+unlock(password)
...
...


<<interface>> Component
+Operation 1()
+Operation 2()
+Operation 3()
...
-component
Concrete Component
...
+Operation 1()
+Operation 2()
+Operation 3()
...
<<abstract>> Abstract Forwarding Component
+Operation 1()
+Operation 2()
+Operation 3()
...
Operation3(): component.Operation3()
Concrete Decorator A
...
+Operation 1()
+Added Behavior()
...
Concrete Decorator 2A
...
+Operation 1()
+Operation 3()
+ Added Behavior()