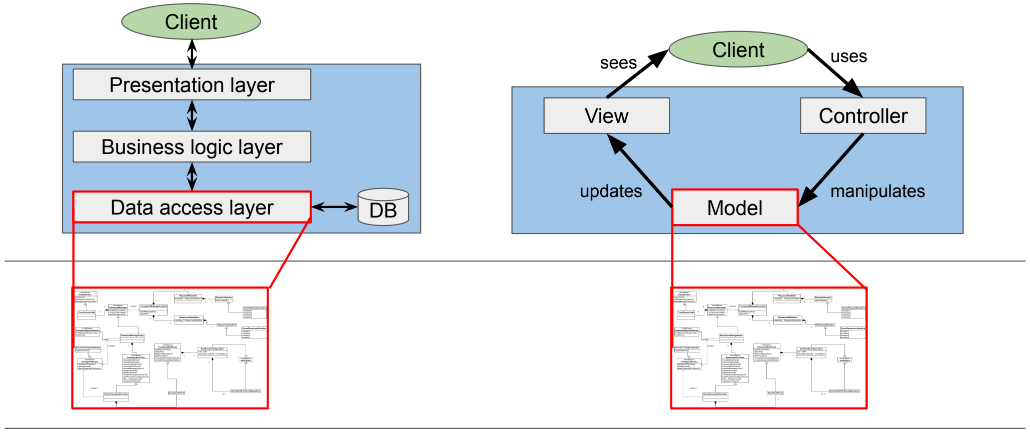
# CSE 403

Software Engineering

Winter 2023

**Software design and best practices**

# Recap: software architecture vs. design



**Architecture and design**

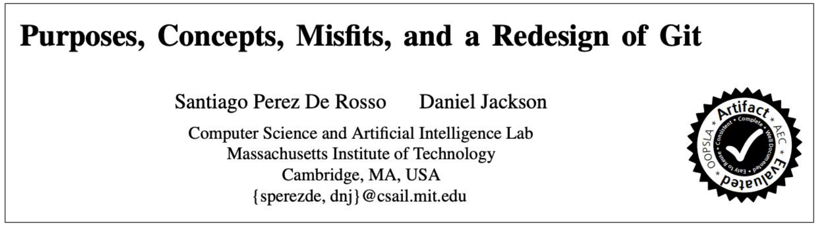
- Components and interfaces: understand, communicate, reuse
- Manage complexity: modularity and separation of concerns
- Process: allow effort estimation and progress monitoring

# Today

- Software design and best practices
- A little quiz on best practices

- Additional material, not covered in class (refresher for 331)
  - UML crash course
  - OO design principles
  - OO design patterns

# SW Design: Purposes, Concepts, and Misfits



**Purposes, Concepts, Misfits, and a Redesign of Git**

Santiago Perez De Rosso  Daniel Jackson

Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA, USA
{sperezde, dnj}@csail.mit.edu

## Concept and motivating purpose

“A **concept** is something you need to understand in order to use an application (and also something a developer needs to understand to work effectively with its code) and is invented to solve a particular problem, which is called the **motivating purpose.**”

Use cases are a good starting point
for defining concepts for motivating purposes.

## Operational principle and misfit

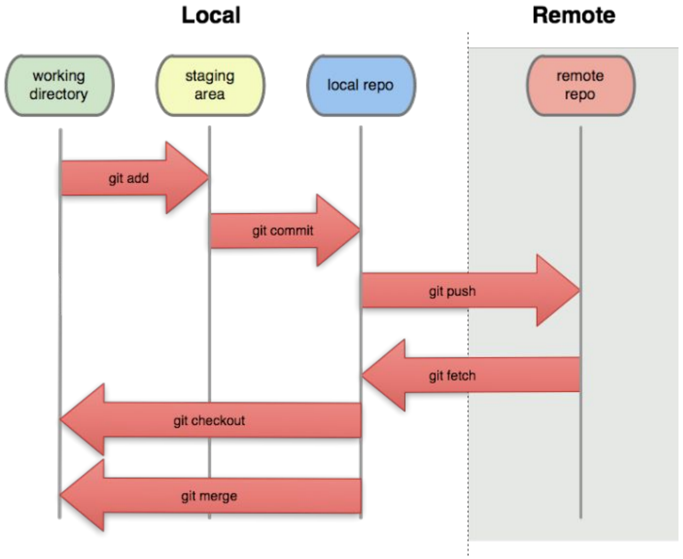
“A concept is defined by an **operational principle**, which is a scenario that illustrates how the concept fulfills its motivating purpose.”

## Operational principle and misfit

“A concept is defined by an **operational principle**, which is a scenario that illustrates how the concept fulfills its motivating purpose.”

“A concept may not be entirely fit for purpose. In that case, one or more **operational misfits** are used to explain why. The operational misfit usually does not contradict the operational principle, but presents a different scenario in which the prescribed behavior does not meet a desired goal.”

## Git: another example for concepts and purposes



## Properties of a good software design

**Motivation**
Each concept should be motivated by at least one purpose.

**Coherence**
Each concept should be motivated by at most one purpose.

**Fulfillment**
Each purpose should motivate at least one concept.

**Non-division**
Each purpose should motivate at most one concept.

**Decoupling**
Concepts should not interfere with one another’s fulfillment of purpose.

## Properties of a good software design

**Motivation**
Each concept should be motivated by at least one purpose.

**Coherence**
Each concept should be motivated by at most one purpose.

**Fulfillment**
Each purpose should motivate at least one concept.

**Non-division**
Each purpose should motivate at most one concept.

**Decoupling**
Concepts should not interfere with one another’s fulfillment of purpose.

## Properties of a good software design

**Motivation**
Each concept should be motivated by at least one purpose.

**Coherence**
Each concept should be motivated by at most one purpose.

**Fulfillment**
Each purpose should motivate at least one concept.

**Non-division**
Each purpose should motivate at most one concept.

**Decoupling**
Concepts should not interfere with one another’s fulfillment of purpose.

## Properties of a good software design

**Motivation**
Each concept should be motivated by at least one purpose.

**Coherence**
Each concept should be motivated by at most one purpose.

**Fulfillment**
Each purpose should motivate at least one concept.

**Non-division**
Each purpose should motivate at most one concept.

**Decoupling**
Concepts should not interfere with one another’s fulfillment of purpose.

## Quiz: setup and goals

- Project groups or small teams
- 6 code snippets
- 2 rounds
  - **First round**
    - For each code snippet, decide whether it represents good or bad practice.
    - **Goal:** discuss and reach consensus on good or bad practice.
  - **Second round** (known “solutions”)
    - For each code snippet, try to understand why it is good or bad practice.
    - **Goal:** come up with an explanation or a counter argument.

## Round 1: good or bad?

## Snippet 1: good or bad?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

# Snippet 2: good or bad?

```
public void addStudent(Student student, String course) {
   if (course.equals("CSE403")) {
      cse403Students.add(student);
   }
   allStudents.add(student)
}
```

# Snippet 3: good or bad?

```
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
  switch (payType) {
    case DEBIT:
       … // process debit card
       break;
    case CREDIT:
       … // process credit card
       break;
    default:
       throw new IllegalArgumentException("Unexpected payment type");
   }
}
```

# Snippet 4: good or bad?

```
public int getAbsMax(int x, int y) {
  if (x<0) {
    x = -x;
  }
  if (y<0) {
    y = -y;
  }
  return Math.max(x, y);
}
```

# Snippet 5: good or bad?

```
public class ArrayList<E> {
   public E remove(int index) {
      …
   }
   public boolean remove(Object o) {
      …
   }
   …
}
```

## Snippet 6: good or bad?

```
public class Point {
   private final int x;
   private final int y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int getX() {
      return this.x;
   }
   public int getY() {
      return this.y;
   }
}
```

## Quiz: setup and goals

- Project groups or small teams
- 6 code snippets
- 2 rounds
  - **First round**
    - For each code snippet, decide whether it represents good or bad practice.
    - **Goal:** discuss and reach consensus on good or bad practice.
  - **Second round** (known “solutions”)
    - For each code snippet, try to understand why it is good or bad practice.
    - **Goal:** come up with an explanation or a counter argument.

https://pollev.com/renejust859

## Round 2: why is it good or bad?

## My take on this

- ✗ Snippet 1: bad
- ✗ Snippet 2: bad
- ✓ Snippet 3: good
- ✗ Snippet 4: bad
- ✗ Snippet 5: bad
- ✓ Snippet 6: good

## Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

## Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

Null references...the billion dollar mistake.

## Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

```
File[] files = getAllLogs();
for (File f : files) {
    …
}
```

Don’t return null; return an empty array instead.

## Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

No diagnostic information.

## Snippet 2: short but bad! why?

```
public void addStudent(Student student, String course) {
   if (course.equals("CSE403")) {
      cse403Students.add(student);
   }
   allStudents.add(student)
}
```

☒

## Snippet 2: short but bad! why?

```
public void addStudent(Student student, String course) {
   if (course.equals("CSE403")) {
      cse403Students.add(student);
   }
   allStudents.add(student)
}
```

☒

Defensive programming: add an assertion (or write the literal first).
Use constants and enums to avoid literal duplication.

## Snippet 3: this is good, but why?

```
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
  switch (payType) {
    case DEBIT:
       … // process debit card
       break;
    case CREDIT:
       … // process credit card
       break;
    default:
       throw new IllegalArgumentException("Unexpected payment type");
  }
}
```

☑

## Snippet 3: this is good, but why?

```
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
  switch (payType) {
    case DEBIT:
       … // process debit card
       break;
    case CREDIT:
       … // process credit card
       break;
    default:
       throw new IllegalArgumentException("Unexpected payment type");
  }
}
```

☑

Type safety using an enum; throws an exception for unexpected cases (e.g., future extensions of PaymentType).

## Snippet 4: also bad! huh?

```
public int getAbsMax(int x, int y) {
  if (x<0) {
    x = -x;
  }
  if (y<0) {
    y = -y;
  }
  return Math.max(x, y);
}
```

## Snippet 4: also bad! huh?

```
public int getAbsMax(int x, int y) {
  if (x<0) {
    x = -x;
  }
  if (y<0) {
    y = -y;
  }
  return Math.max(x, y);
}
```

Method parameters should be final;
use local variables to sanitize inputs.

## Snippet 5: Java API, but still bad! why?

```
public class ArrayList<E> {
   public E remove(int index) {
      …
   }
   public boolean remove(Object o) {
      …
   }
   …
}
```

## Snippet 5: Java API, but still bad! why?

```
public class ArrayList<E> {
   public E remove(int index) {
      …
   }
   public boolean remove(Object o) {
      …
   }
   …
}
```

```
ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add(“Hello”);
l.add(“World”);
l.remove(index);
```

What does the last call return (`l.remove(index)`)?

## Snippet 5: Java API, but still bad! why?

```
public class ArrayList<E> {
   public E remove(int index) {
      …
   }
   public boolean remove(Object o) {
      …
   }
   …
}
```

☒

```
ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add(“Hello”);
l.add(“World”);
l.remove(index);
```

Avoid method overloading, which is statically resolved.
Autoboxing/unboxing adds additional confusion.

## Snippet 6: this is good, but why?

```
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int getX() {
      return this.x;
   }
   public int getY() {
      return this.y;
   }
}
```

✓

## Snippet 6: this is good, but why?

```
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int getX() {
      return this.x;
   }
   public int getY() {
      return this.y;
   }
}
```

✓

Good encapsulation; immutable object.

## Additional material, not discussed in class

# UML crash course

# UML crash course

**The main questions**

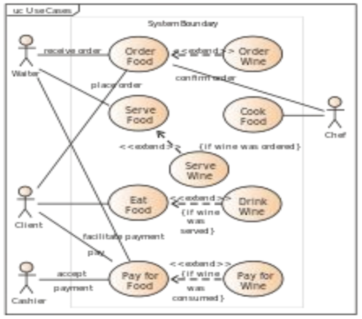
- What is UML?
- Is it useful, why bother?
- When to (not) use UML?

# What is UML?

- Unified Modeling Language.
- Developed in the mid 90’s, improved since.
- Standardized notation for modeling OO systems.
- A collection of diagrams for different viewpoints:
  - Use case diagrams
  - Component diagrams
  - Class and Object diagrams
  - Sequence diagrams
  - Statechart diagrams
  - ...

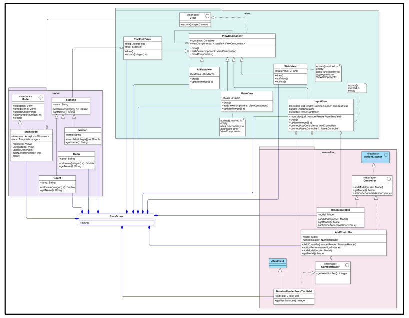
# What is UML?

- Unified Modeling Language.
- Developed in the mid 90’s, improved since.
- Standardized notation for modeling OO systems.
- A collection of diagrams for different viewpoints:
  - **Use case diagrams**
  - Component diagrams
  - Class and Object diagrams
  - Sequence diagrams
  - Statechart diagrams
  - ...

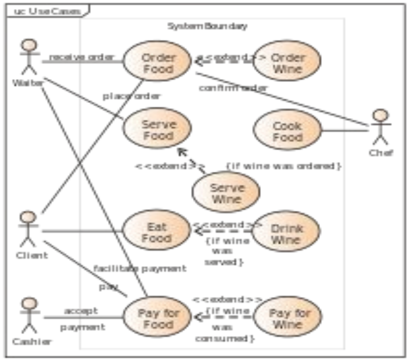


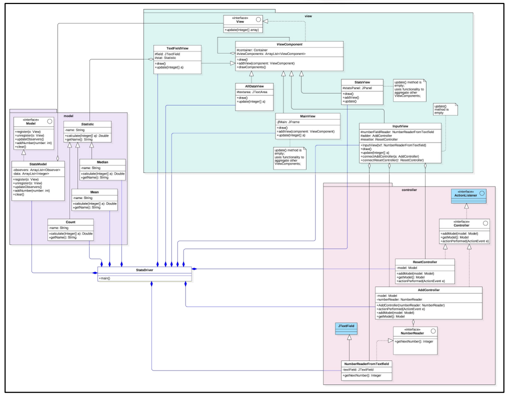
# What is UML?

- Unified Modeling Language.
- Developed in the mid 90’s, improved since.
- Standardized notation for modeling OO systems.
- A collection of diagrams for different viewpoints:
  - Use case diagrams
  - Component diagrams
  - **Class and Object diagrams**
  - Sequence diagrams
  - Statechart diagrams
  - ...



# Are UML diagrams useful?





# Are UML diagrams useful?

**Communication**

- Forward design (before coding)
  - Brainstorm ideas (on whiteboard or paper).
  - Draft and iterate over software design.

**Documentation**

- Backward design (after coding)
  - Obtain diagram from source code.

In this class, we will use UML class diagrams mainly for visualization and discussion purposes.

# Classes vs. objects

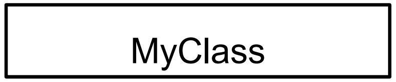
**Class**

- Grouping of similar objects.
  - Student
  - Car
- Abstraction of common properties and behavior.
  - Student: Name and Student ID
  - Car: Make and Model

**Object**

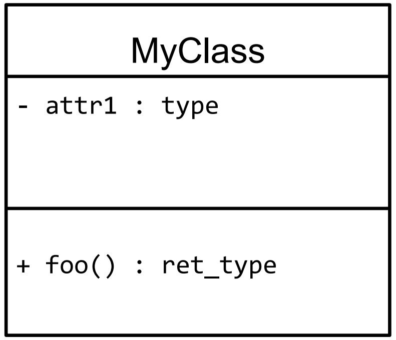
- Entity from the real world.
- Instance of a class
  - Student: Joe (4711), Jane (4712), …
  - Car: Audi A6, Honda Civic, ...

## UML class diagram: basic notation



```
MyClass
```

## UML class diagram: basic notation



```
MyClass
- attr1 : type
+ foo() : ret_type
```

**Name**

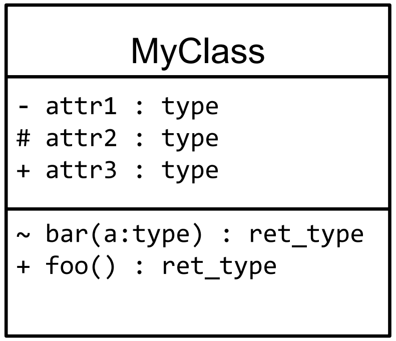
**Attributes**

*<visibility> <name> : <type>*

**Methods**

*<visibility> <name>(<param>*) : <return type>*
*<param> := <name> : <type>*

## UML class diagram: basic notation



```
MyClass
- attr1 : type
# attr2 : type
+ attr3 : type
~ bar(a:type) : ret_type
+ foo() : ret_type
```

**Name**

**Attributes**

*<visibility> <name> : <type>*

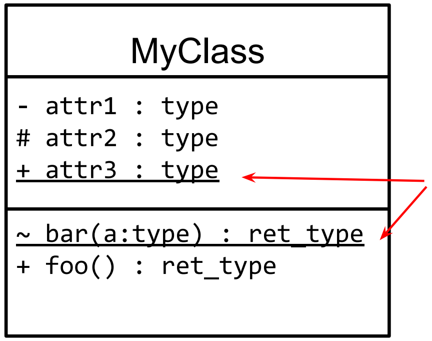
**Methods**

*<visibility> <name>(<param>*) : <return type>*
*<param> := <name> : <type>*

**Visibility**

*- private*
*~ package-private*
*# protected*
*+ public*

## UML class diagram: basic notation



```
MyClass
- attr1 : type
# attr2 : type
+ attr3 : type
~ bar(a:type) : ret_type
+ foo() : ret_type
```

**Name**

**Attributes**

*<visibility> <name> : <type>*

*Static attributes or methods are underlined*

**Methods**

*<visibility> <name>(<param>*) : <return type>*
*<param> := <name> : <type>*

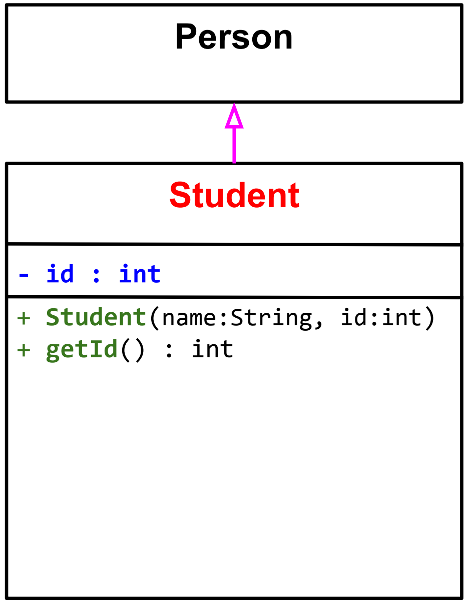
**Visibility**

*- private*
*~ package-private*
*# protected*
*+ public*

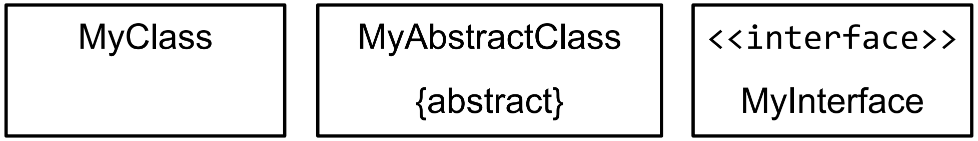
## UML class diagram: concrete example

```
public class Person {
  ...
}
```

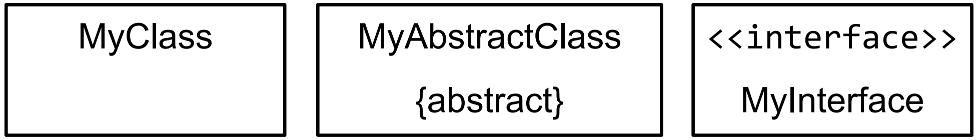
```
public class Student
    extends Person {
  private int id;
  public Student(String name,
                 int id) {
    ...
  }

  public int getId() {
    return this.id;
  }
}
```



## Classes, abstract classes, and interfaces



## Classes, abstract classes, and interfaces



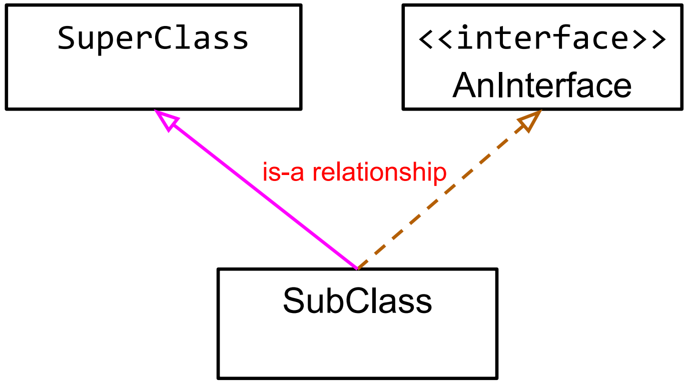
```
public class MyClass {

  public void op() {
    ...
  }

  public int op2() {
    ...
  }
}
```

```
public abstract class
    MyAbstractClass {

  public abstract void op();


  public int op2() {
    ...
  }
}
```

```
public interface
    MyInterface {

  public void op();


  public int op2();
}
```

Level of detail in a given class or interface may vary and depends on context and purpose.

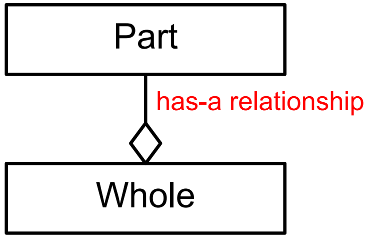
## UML class diagram: Inheritance



```
public class SubClass extends SuperClass implements AnInterface
```

# UML class diagram: Aggregation and Composition

**Aggregation**

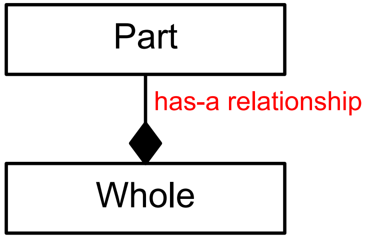

Part

has-a relationship

Whole

- Existence of Part does not depend on the existence of Whole.
- Lifetime of Part does not depend on Whole.
- No single instance of whole is the unique owner of Part (might be shared with other instances of Whole).

**Composition**

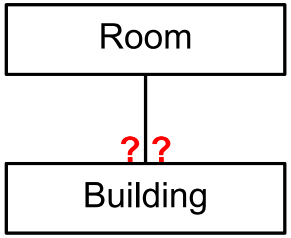

Part

has-a relationship

Whole

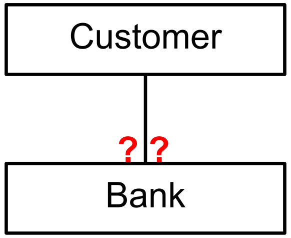
- Part cannot exist without Whole.
- Lifetime of Part depends on Whole.
- One instance of Whole is the single owner of Part.

# Aggregation or Composition?


Room

??

Building

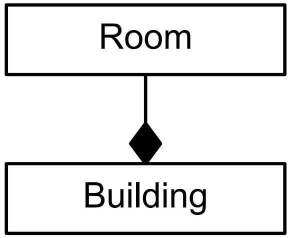

Customer

??

Bank

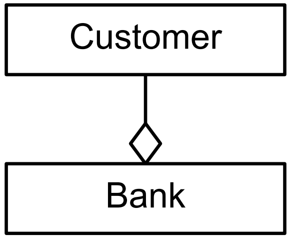
# Aggregation or Composition?

**Composition**


Room

Building

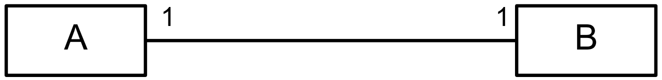
**Aggregation**


Customer

Bank

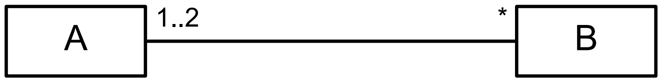
What about class and students or body and body parts?

# UML class diagram: multiplicity


A

B

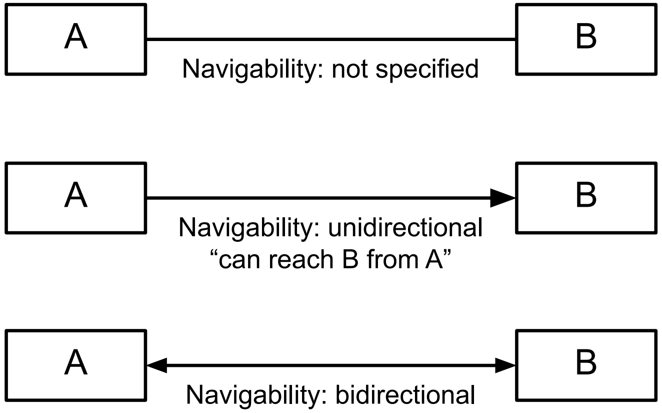
Each A is associated with exactly one B
Each B is associated with exactly one A


A

B

Each A is associated with any number of Bs
Each B is associated with exactly one or two As

# UML class diagram: navigability



A

B

Navigability: not specified

A

B

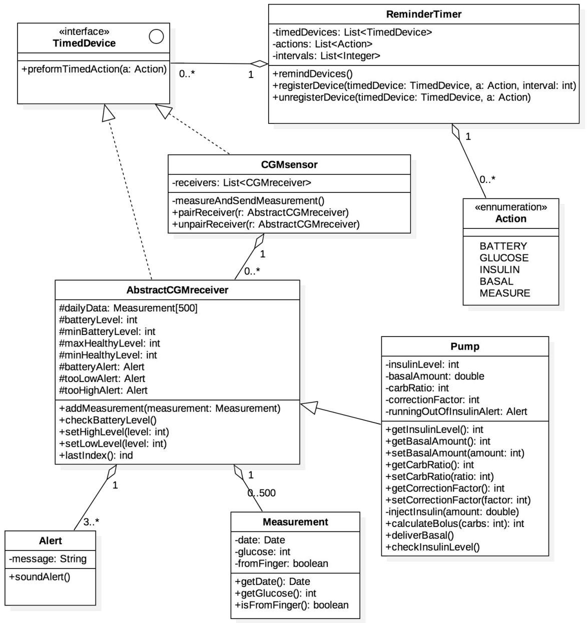
Navigability: unidirectional
“can reach B from A”

A

B

Navigability: bidirectional

# UML class diagram: example



«interface»
**TimedDevice**

+preformTimedAction(a: Action)

**ReminderTimer**

-timedDevices: List<TimedDevice>
-actions: List<Action>
-intervals: List<Integer>

+remindDevices()
+registerDevice(timedDevice: TimedDevice, a: Action, interval: int)
+unregisterDevice(timedDevice: TimedDevice, a: Action)

**CGMsensor**

-receivers: List<CGMreceiver>

-measureAndSendMeasurement()
+pairReceiver(r: AbstractCGMreceiver)
+unpairReceiver(r: AbstractCGMreceiver)

«ennumeration»
**Action**

BATTERY
GLUCOSE
INSULIN
BASAL
MEASURE

**AbstractCGMreceiver**

#dailyData: Measurement[500]
#batteryLevel: int
#minBatteryLevel: int
#maxHealthyLevel: int
#minHealthyLevel: int
#batteryAlert: Alert
#tooLowAlert: Alert
#tooHighAlert: Alert

+addMeasurement(measurement: Measurement)
+checkBatteryLevel()
+setHighLevel(level: int)
+setLowLevel(level: int)
+lastIndex(): ind

**Pump**

-insulinLevel: int
-basalAmount: double
-carbRatio: int
-correctionFactor: int
-runningOutOfInsulinAlert: Alert

+getInsulinLevel(): int
+getBasalAmount(): int
+setBasalAmount(amount: int)
+getCarbRatio(): int
+setCarbRatio(ratio: int)
+getCorrectionFactor(): int
+setCorrectionFactor(factor: int)
-injectInsulin(amount: double)
+calculateBolus(carbs: int): int
+deliverBasal()
+checkInsulinLevel()

**Alert**

-message: String

+soundAlert()

**Measurement**

-date: Date
-glucose: int
-fromFinger: boolean

+getDate(): Date
+getGlucose(): int
+isFromFinger(): boolean

# Summary: UML

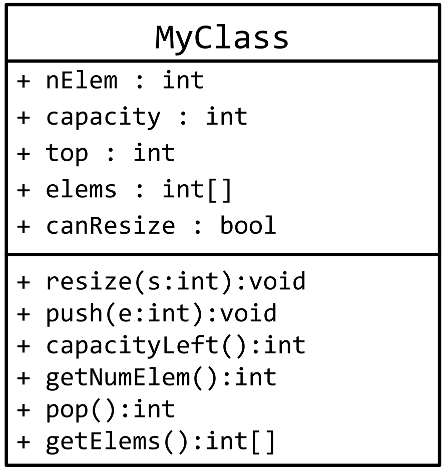
- Unified notation for modeling OO systems.
- Allows different levels of abstraction.
- Suitable for design discussions and documentation.

## **OO design principles**

# OO design principles

- **Information hiding (and encapsulation)**
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

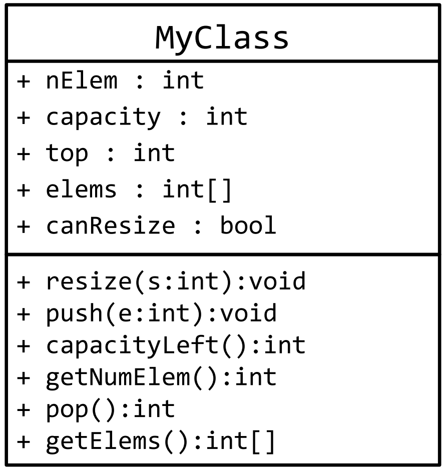
# Information hiding



```
MyClass
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

```
public class MyClass {
  public int nElem;
  public int capacity;
  public int top;
  public int[] elems;
  public boolean canResize;
  ...
  public void resize(int s){...}
  public void push(int e){...}
  public int capacityLeft(){...}
  public int getNumElem(){...}
  public int pop(){...}
  public int[] getElems(){...}
}
```

# Information hiding

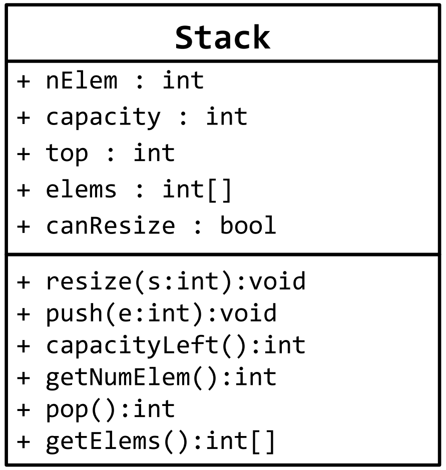


```
MyClass
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

```
public class MyClass {
  public int nElem;
  public int capacity;
  public int top;
  public int[] elems;
  public boolean canResize;
  ...
  public void resize(int s){...}
  public void push(int e){...}
  public int capacityLeft(){...}
  public int getNumElem(){...}
  public int pop(){...}
  public int[] getElems(){...}
}
```

What does MyClass do?

# Information hiding

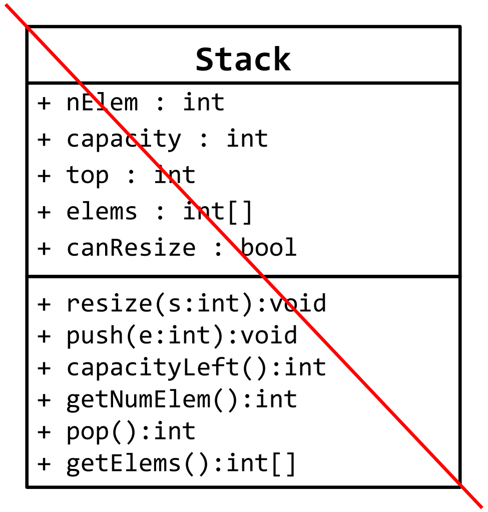


```
Stack
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

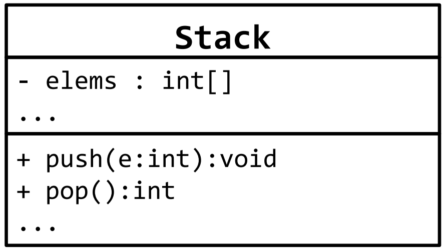
```
public class Stack {
  public int nElem;
  public int capacity;
  public int top;
  public int[] elems;
  public boolean canResize;
  ...
  public void resize(int s){...}
  public void push(int e){...}
  public int capacityLeft(){...}
  public int getNumElem(){...}
  public int pop(){...}
  public int[] getElems(){...}
}
```

Anything that could be improved in this implementation?

## Information hiding



```
Stack
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

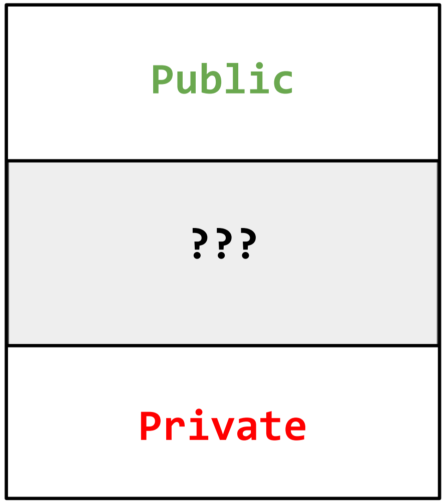


```
Stack
- elems : int[]
...
+ push(e:int):void
+ pop():int
...
```

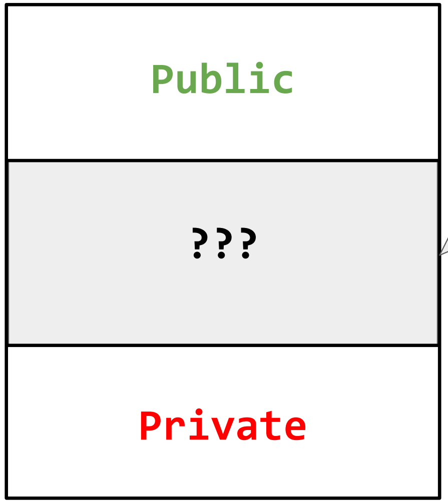
**Information hiding:**

- Reveal as little information about internals as possible.
- Segregate public interface and implementation details.
- Reduces complexity.

## Information hiding vs. visibility



## Information hiding vs. visibility



- Protected, package-private, or friend-accessible (C++).
- Not part of the public API.
- Implementation detail that a subclass/friend may rely on.

## OO design principles

- Information hiding (and encapsulation)
- **Polymorphism**
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

## A little refresher: what is Polymorphism?

## A little refresher: what is Polymorphism?

An object’s ability to provide different behaviors.

**Types of polymorphism**

- Ad-hoc polymorphism (e.g., operator overloading)
  - `a + b` ⇒ String vs. int, double, etc.
- Subtype polymorphism (e.g., method overriding)

  -
    ```
    Object obj = ...;
    obj.toString();
    ```
    ⇒ toString() can be overridden in subclasses and therefore provide a different behavior.
- Parametric polymorphism (e.g., Java generics)

  -
    ```
    class LinkedList<E> {
      void add(E) {...}
      E get(int index) {...}
    ```
    ⇒ A LinkedList can store elements regardless of their type but still provide full type safety.

## A little refresher: what is Polymorphism?

An object’s ability to provide different behaviors.

**Types of polymorphism**

- Subtype polymorphism (e.g., method overriding)

  -
    ```
    Object obj = ...;
    obj.toString();
    ```
    ⇒ toString() can be overridden in subclasses and therefore provide a different behavior.

Subtype polymorphism is essential to many OO design principles.

## OO design principles

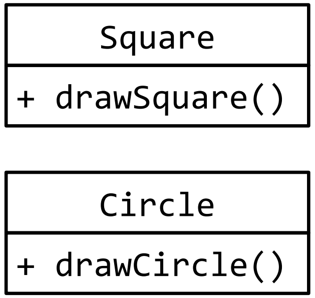
- Information hiding (and encapsulation)
- Polymorphism
- **Open/closed principle**
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

# Open/closed principle

**Software entities** (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

```
public static void draw(Object o) {
  if (o instanceof Square) {
    drawSquare((Square) o)
  } else if (o instanceof Circle) {
    drawCircle((Circle) o);
  } else {
    ...
  }
}
```



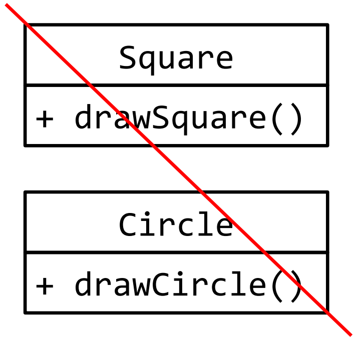
Good or bad design?

# Open/closed principle

**Software entities** (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

```
public static void draw(Object o) {
  if (o instanceof Square) {
    drawSquare((Square) o)
  } else if (o instanceof Circle) {
    drawCircle((Circle) o);
  } else {
    ...
  }
}
```



Violates the open/closed principle!

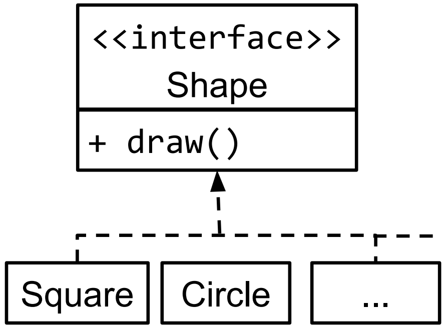
# Open/closed principle

**Software entities** (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

```
public static void draw(Object s) {
  if (s instanceof Shape) {
     s.draw();
  } else {
     …
  }
}
```

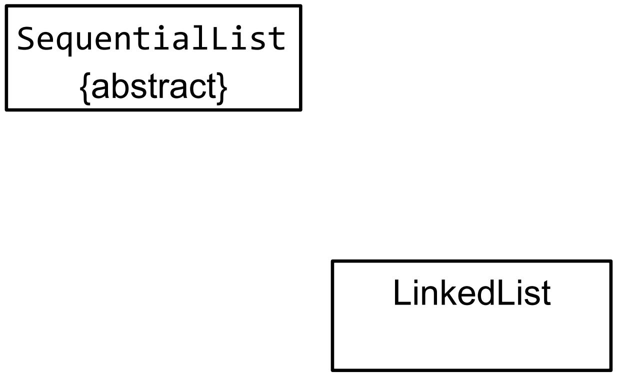
```
public static void draw(Shape s) {
  s.draw();
}
```



# OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- **Inheritance in Java**
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

## Inheritance: (abstract) classes and interfaces

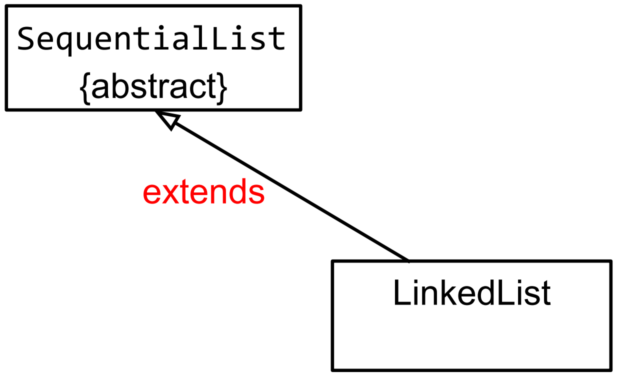


SequentialList {abstract}

LinkedList

## Inheritance: (abstract) classes and interfaces

**LinkedList extends SequentialList**



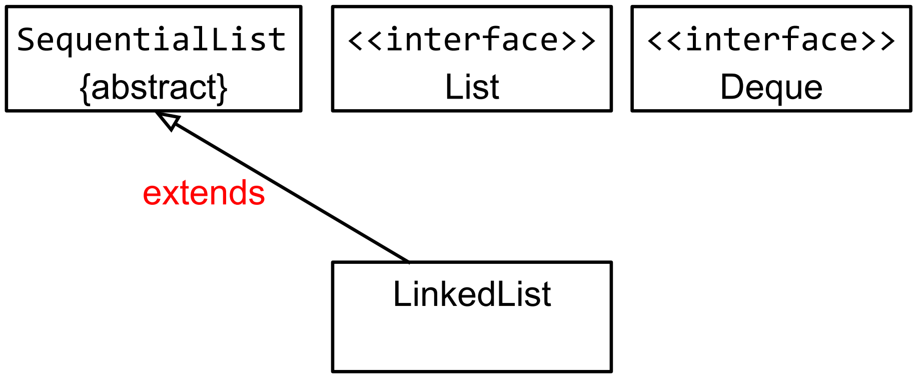
SequentialList {abstract}

extends

LinkedList

## Inheritance: (abstract) classes and interfaces

**LinkedList extends SequentialList**



SequentialList {abstract}

<<interface>> List

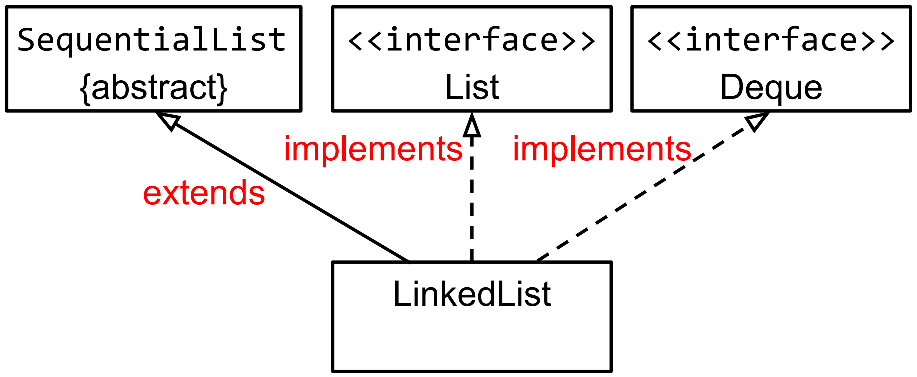
<<interface>> Deque

extends

LinkedList

## Inheritance: (abstract) classes and interfaces

**LinkedList extends SequentialList implements List, Deque**



SequentialList {abstract}

<<interface>> List

<<interface>> Deque

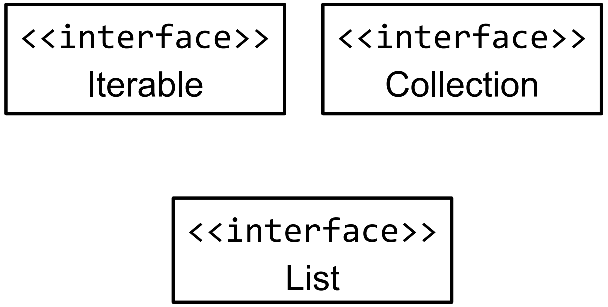
implements

implements

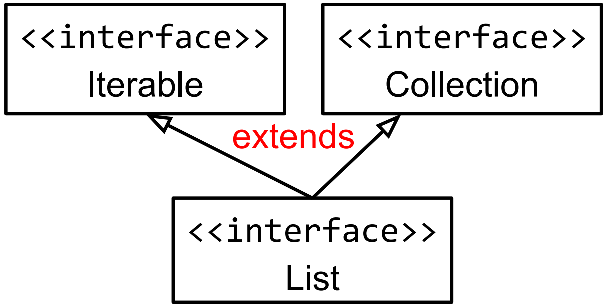
extends

LinkedList

# Inheritance: (abstract) classes and interfaces

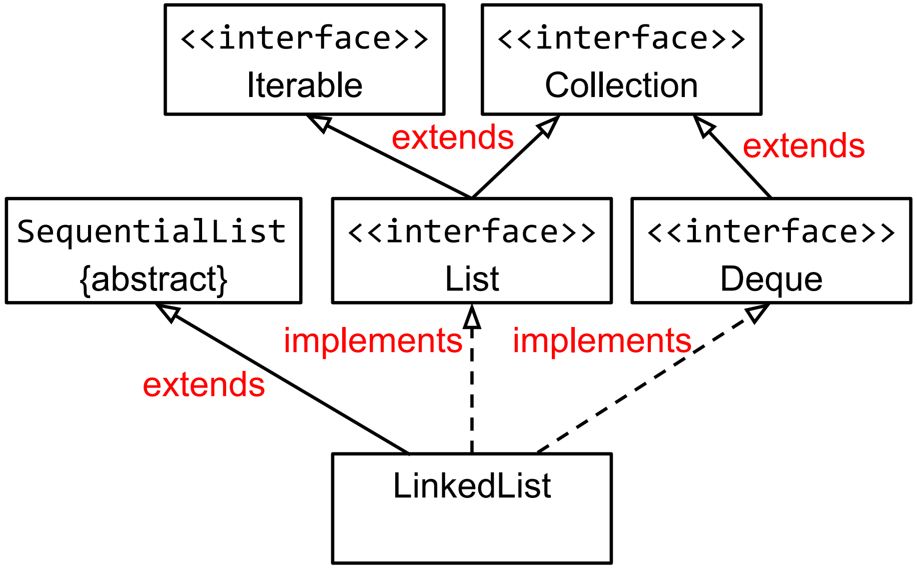


# Inheritance: (abstract) classes and interfaces



**List extends Iterable, Collection**

# Inheritance: (abstract) classes and interfaces

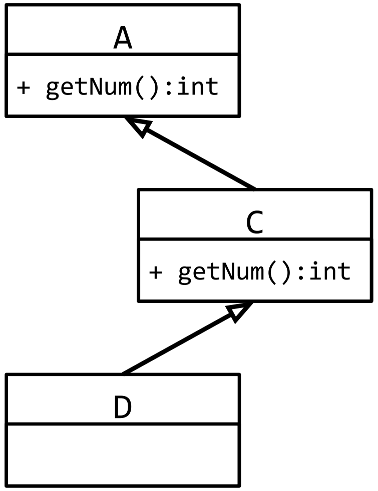


# OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- **The diamond of death**
- Liskov substitution principle
- Composition/aggregation over inheritance

## The “diamond of death”: the problem

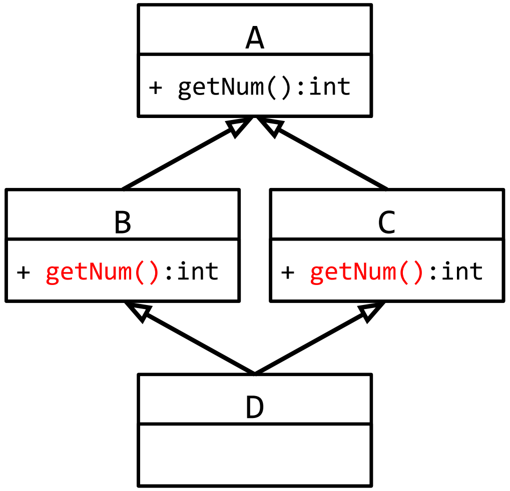
```
...
A a = new D();
int num = a.getNum();
...
```



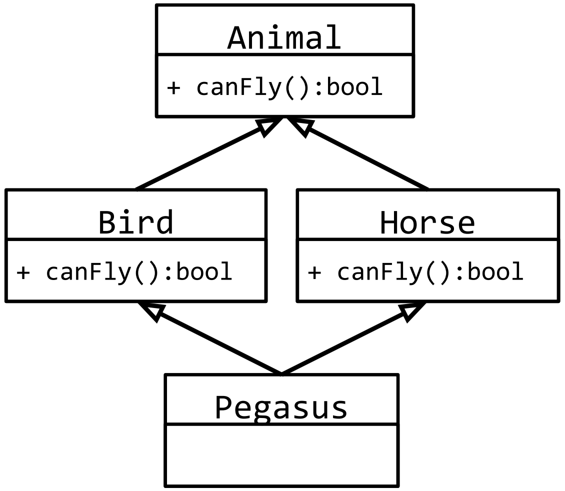
## The “diamond of death”: the problem

```
...
A a = new D();
int num = a.getNum();
...
```

Which getNum() method should be called?



## The “diamond of death”: concrete example



Can this happen in Java? Yes, with default methods in Java 8.

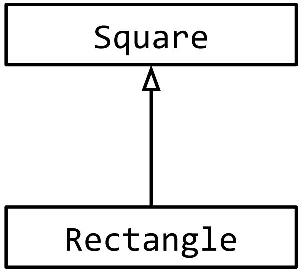
## OO design principles

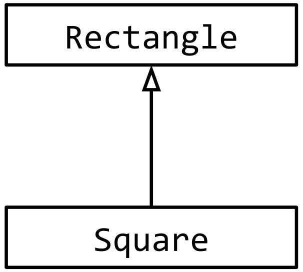
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- **Liskov substitution principle**
- Composition/aggregation over inheritance

# Design principles: Liskov substitution principle

**Motivating example**

*We know that a square is a special kind of a rectangle. So, which of the following OO designs makes sense?*

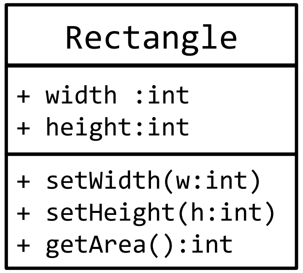




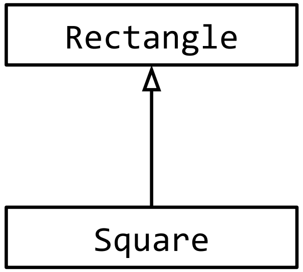
# Design principles: Liskov substitution principle

**Subtype requirement**

*Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.*



```
Rectangle
+ width :int
+ height:int
+ setWidth(w:int)
+ setHeight(h:int)
+ getArea():int
```

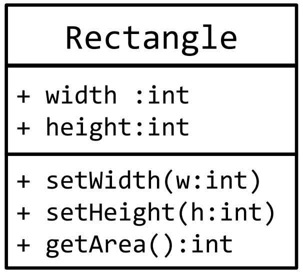


Is the subtype requirement fulfilled?

# Design principles: Liskov substitution principle

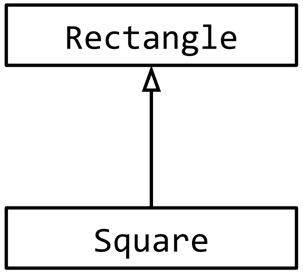
**Subtype requirement**

*Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.*



```
Rectangle
+ width :int
+ height:int
+ setWidth(w:int)
+ setHeight(h:int)
+ getArea():int
```

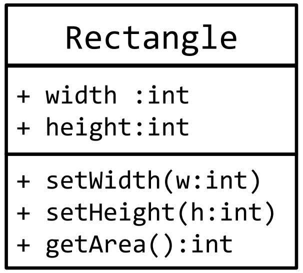
```
Rectangle r =
  new Rectangle(2,2);


int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);

assertEquals(A * 2,
    r.getArea());
```



# Design principles: Liskov substitution principle

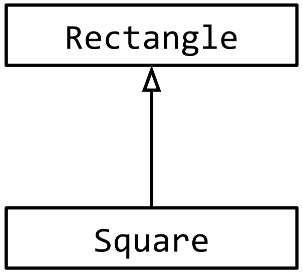
**Subtype requirement**

*Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.*



```
Rectangle
+ width :int
+ height:int
+ setWidth(w:int)
+ setHeight(h:int)
+ getArea():int
```

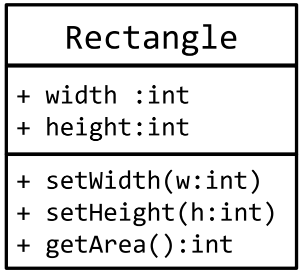
```
Rectangle r =
  ~~new Rectangle(2,2);~~
  new Square(2);

int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);

assertEquals(A * 2,
    r.getArea());
```



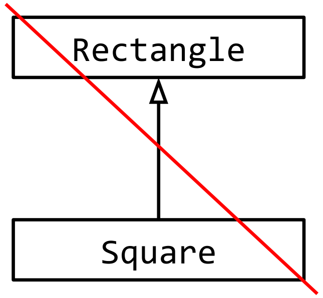
## Design principles: Liskov substitution principle

**Subtype requirement**

*Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.*



```
Rectangle r =
  ~~new Rectangle(2,2);~~
  new Square(2);

int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);

assertEquals(A * 2,
    r.getArea());
```

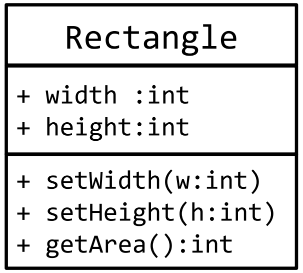


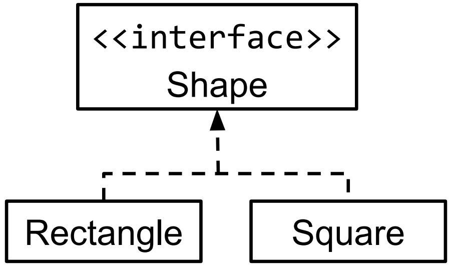
Violates the Liskov substitution principle!

## Design principles: Liskov substitution principle

**Subtype requirement**

*Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.*

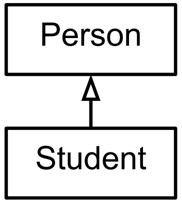


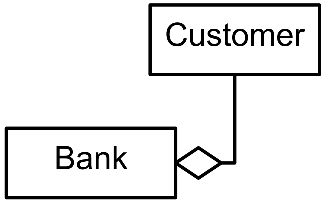


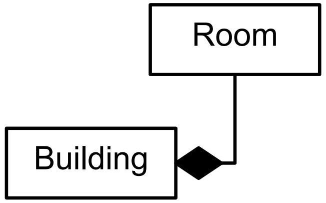
## OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- **Composition/aggregation over inheritance**

## Inheritance vs. (Aggregation vs. *Composition*)







```
public class Student
    extends Person{

 public Student(){
 }

 ...
}
```

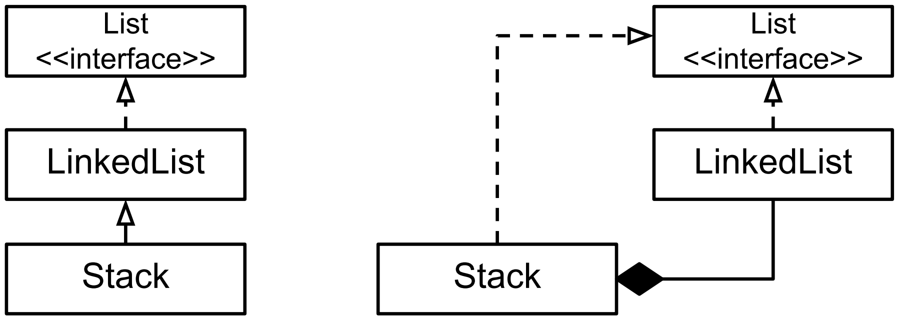
```
public class Bank {
 Customer c;

 public Bank(Customer c){
  this.c = c;
 }
 ...
}
```

```
public class Building {
 Room r;

 public Building(){
  this.r = new Room();
 }
 ...
}
```

is-a relationship

has-a relationship

## Design choice: inheritance or composition?

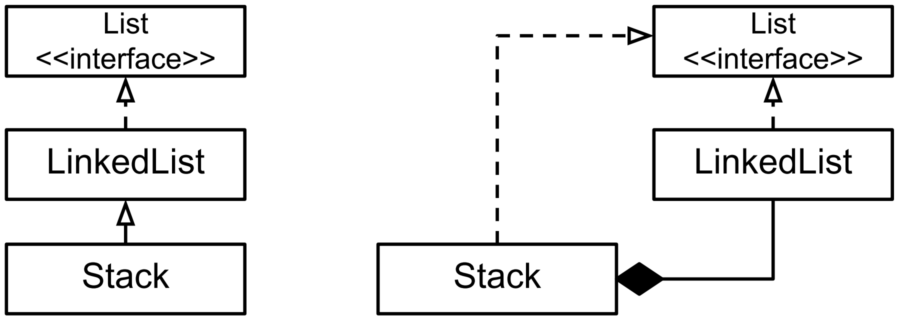


```
public class Stack<E>
     extends LinkedList<E> {
  ...
}
```

```
public class Stack<E> implements List<E> {
  private List<E> l = new LinkedList<>();
  ...
}
```

Hmm, both designs seem valid -- what are pros and cons?

## Design choice: inheritance or composition?



**Pros**

- No delegation methods required.
- Reuse of common state and behavior.

**Cons**

- Exposure of all inherited methods (a client might rely on this particular superclass -> can't change it later).
- Changes in superclass are likely to break subclasses.

**Pros**

- Highly flexible and configurable: no additional subclasses required for different compositions.

**Cons**

- All interface methods need to be implemented -> delegation methods required, even for code reuse.

Composition/aggregation over inheritance allows more flexibility.

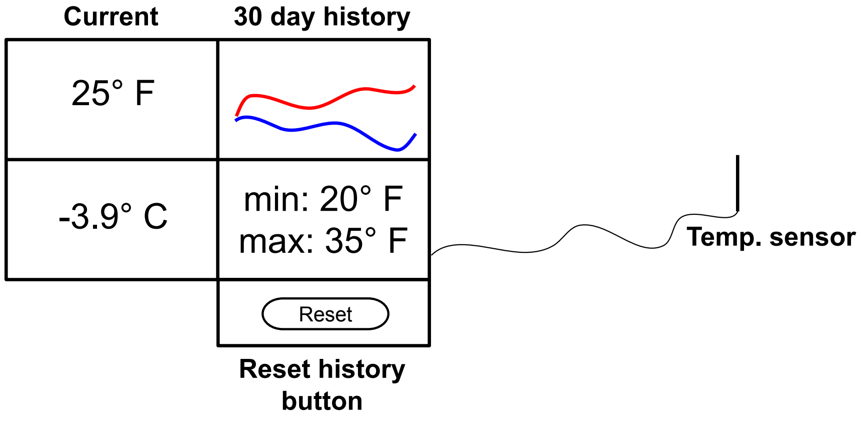
## OO design principles: summary

- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle
- Composition/aggregation over inheritance

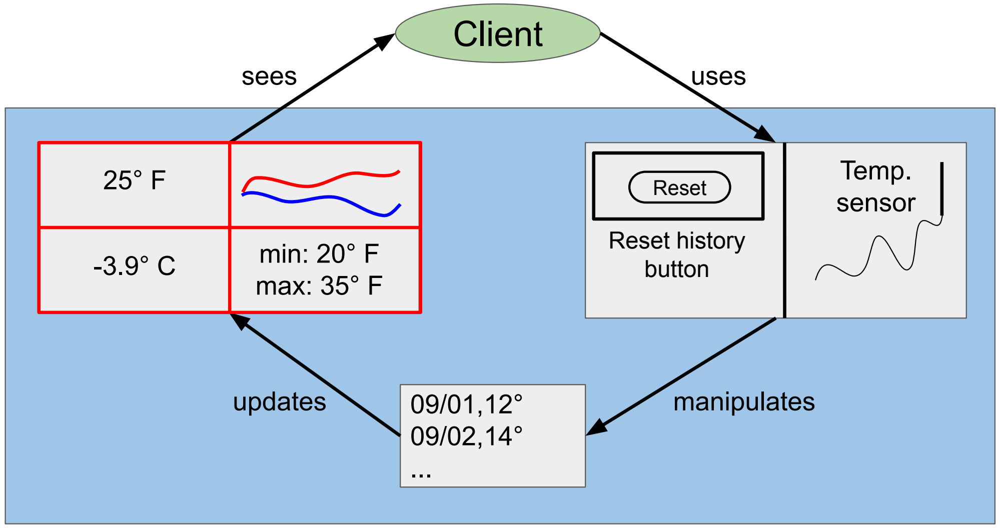
## **OO design patterns**

# A first design problem

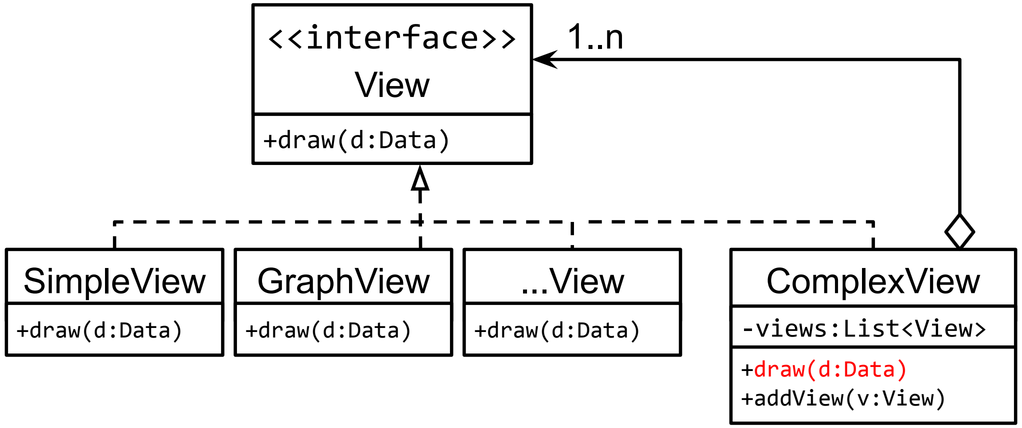
## Weather station revisited

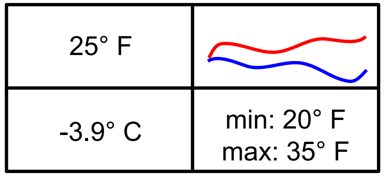


# What's a good design for the view component?



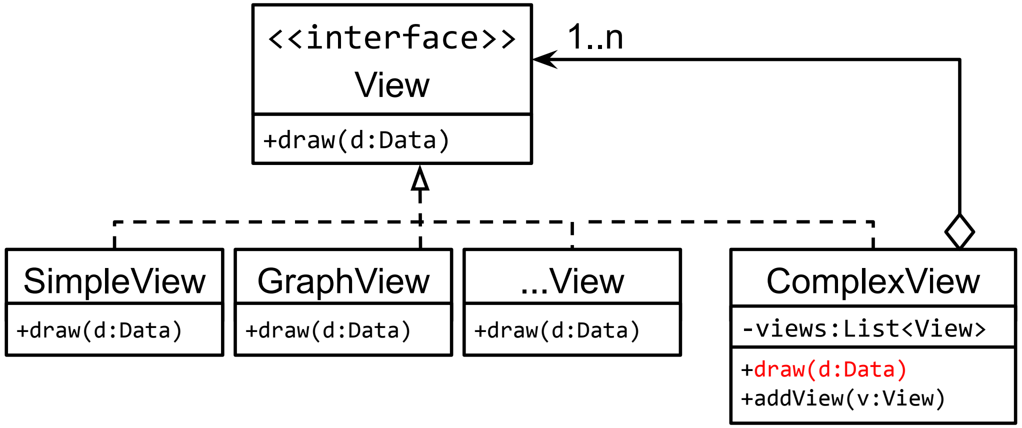
# Weather station: view

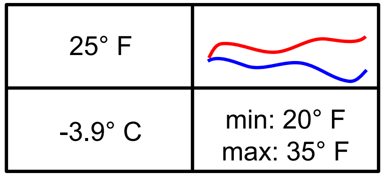




How do we need to implement draw(d:Data)?

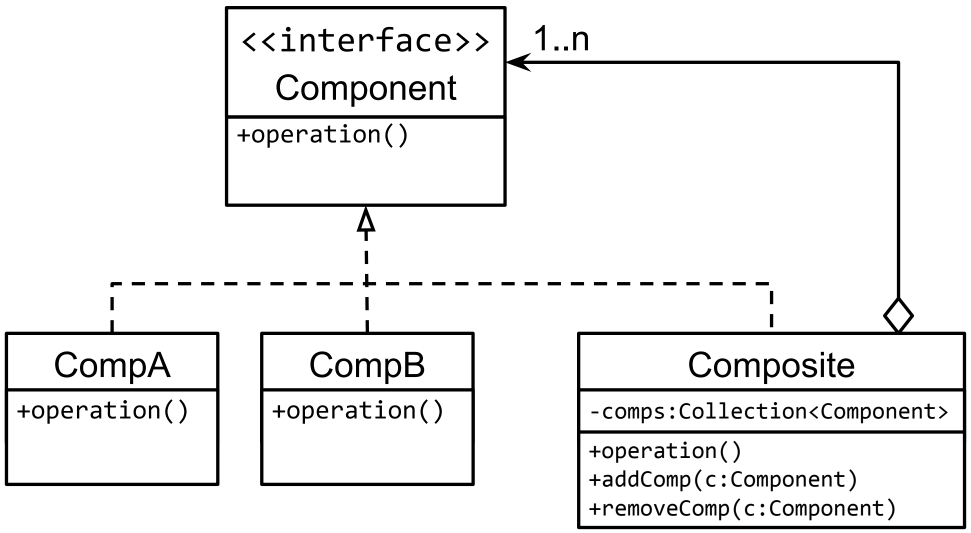
# Weather station: view





```
public void draw(Data d) {
  for (View v : views) {
    v.draw(d);
  }
}
```

## The general solution: Composite pattern



<<interface>>
Component
+operation()

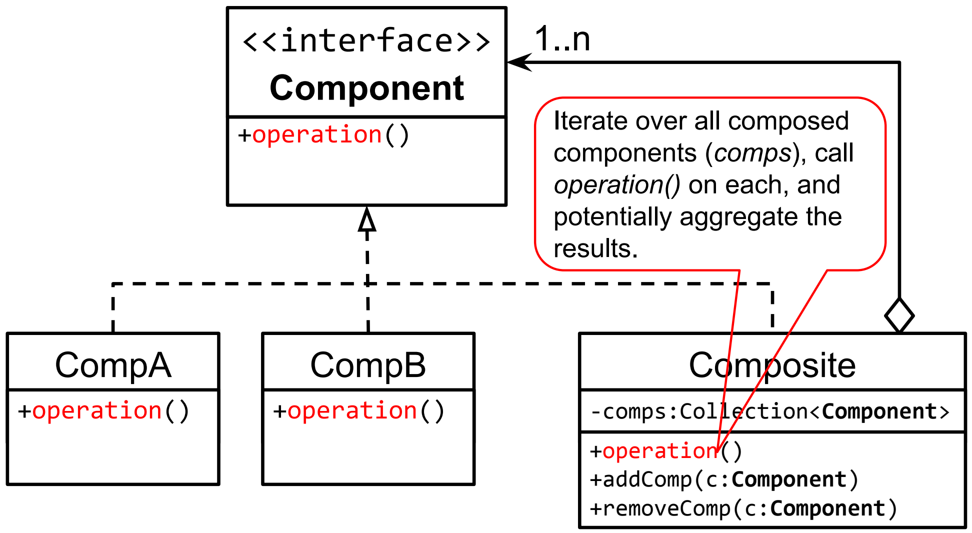
1..n

CompA
+operation()

CompB
+operation()

Composite
-comps:Collection<Component>
+operation()
+addComp(c:Component)
+removeComp(c:Component)

## The general solution: Composite pattern



<<interface>>
**Component**
+operation()

1..n

Iterate over all composed components (*comps*), call *operation()* on each, and potentially aggregate the results.

CompA
+operation()

CompB
+operation()

Composite
-comps:Collection<**Component**>
+operation()
+addComp(c:**Component**)
+removeComp(c:**Component**)

## What is a design pattern?

- Addresses a recurring, common design problem.
- Provides a generalizable solution.
- Provides a common terminology.

## What is a design pattern?

- Addresses a recurring, common design problem.
- Provides a generalizable solution.
- Provides a common terminology.

**Pros**

- Improves communication and documentation.
- “Toolbox” for novice developers.

**Cons**

- Risk of over-engineering.
- Potential impact on system performance.

More than just a name for common sense and best practices.

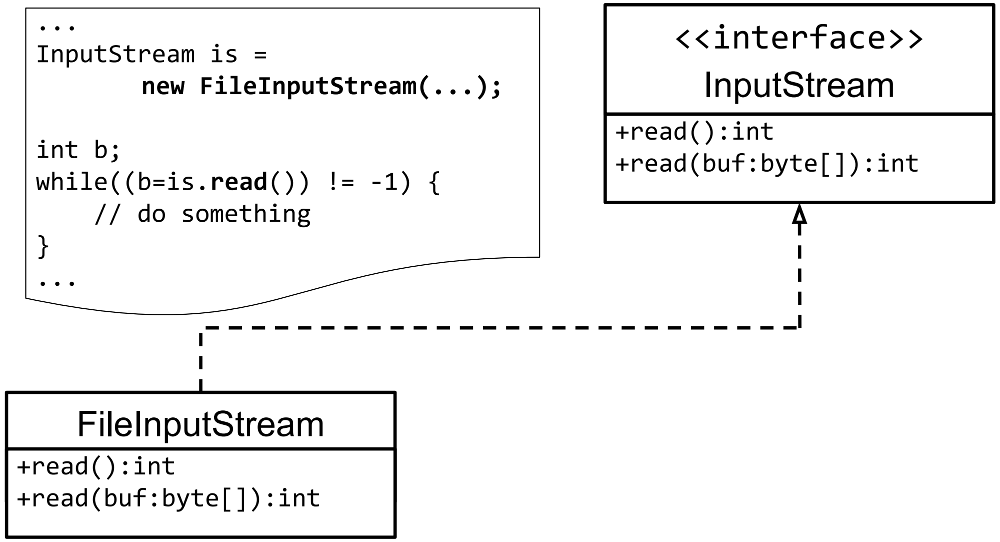
## Design patterns: categories

1. Structural
   - Composite
   - Decorator
   - ...
2. Behavioral
   - Template method
   - Visitor
   - ...
3. Creational
   - Singleton
   - Factory (method)
   - ...

## Design patterns: categories

1. Structural
   - Composite
   - Decorator
   - ...
2. Behavioral
   - Template method
   - Visitor
   - ...
3. Creational
   - Singleton
   - Factory (method)
   - ...

## Another design problem: I/O streams

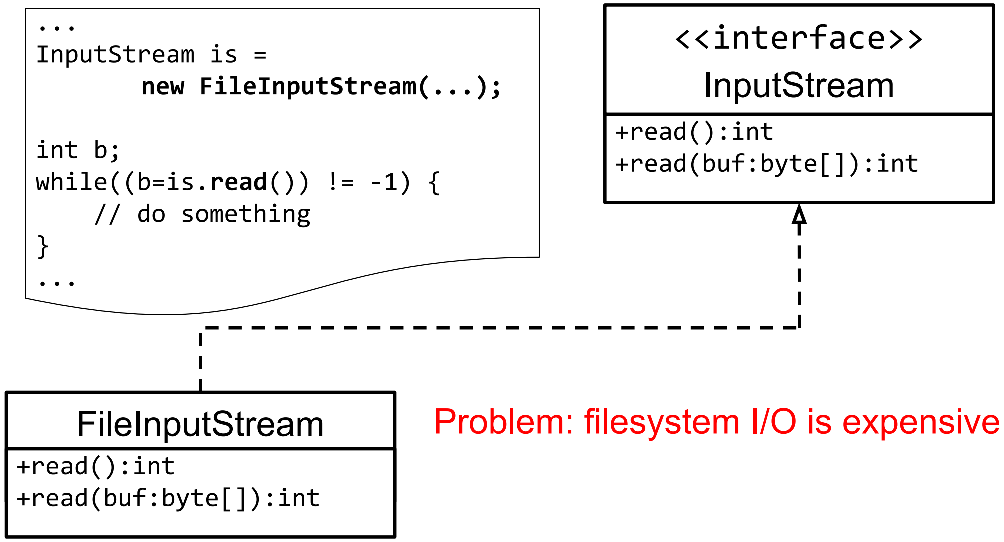


```
...
InputStream is =
        new FileInputStream(...);

int b;
while((b=is.read()) != -1) {
    // do something
}
...
```

<<interface>>
InputStream
+read():int
+read(buf:byte[]):int

FileInputStream
+read():int
+read(buf:byte[]):int

## Another design problem: I/O streams



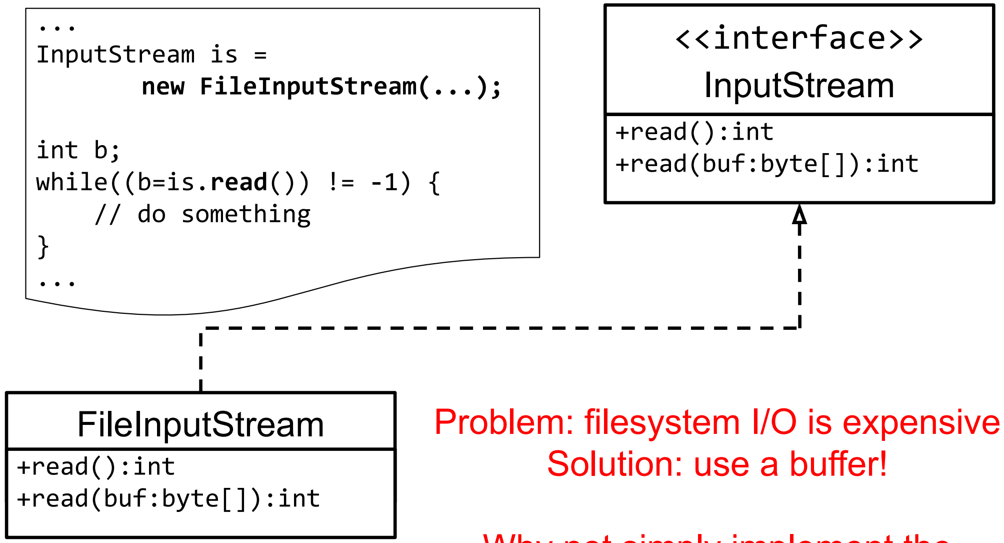
```
...
InputStream is =
        new FileInputStream(...);

int b;
while((b=is.read()) != -1) {
    // do something
}
...
```

<<interface>>
InputStream
+read():int
+read(buf:byte[]):int

FileInputStream
+read():int
+read(buf:byte[]):int

Problem: filesystem I/O is expensive

# Another design problem: I/O streams

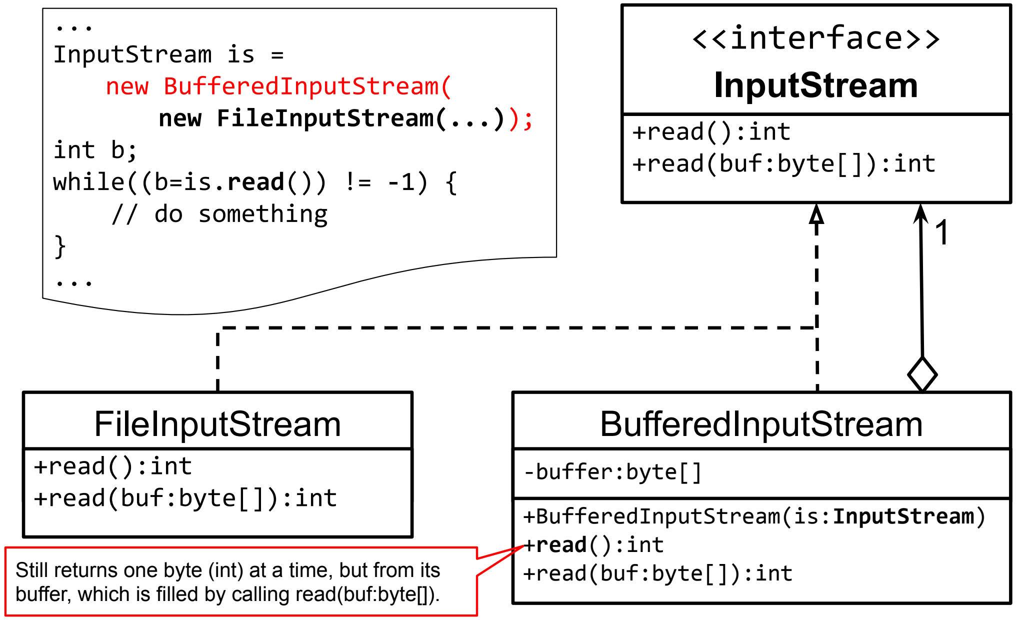


```
...
InputStream is =
        new FileInputStream(...);

int b;
while((b=is.read()) != -1) {
    // do something
}
...
```

Problem: filesystem I/O is expensive
Solution: use a buffer!

Why not simply implement the
buffering in the client or subclass?

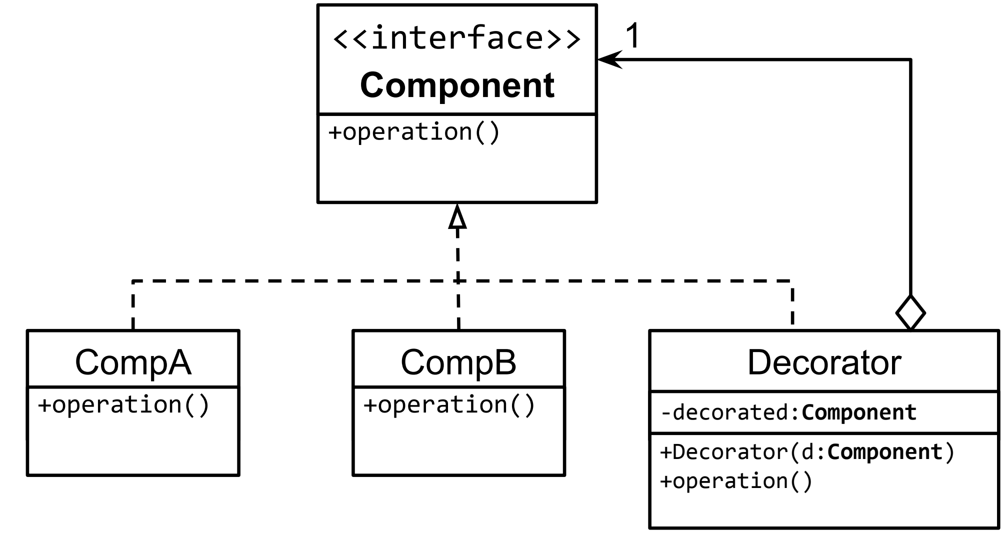
# Another design problem: I/O streams



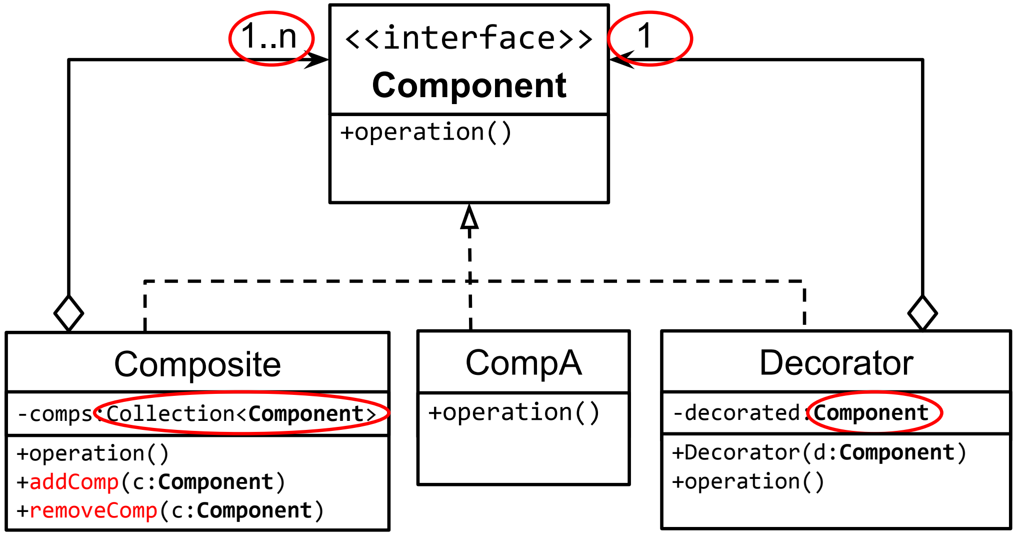
```
...
InputStream is =
    new BufferedInputStream(
        new FileInputStream(...));
int b;
while((b=is.read()) != -1) {
    // do something
}
...
```

Still returns one byte (int) at a time, but from its buffer, which is filled by calling read(buf:byte[]).

# The general solution: Decorator pattern



# Composite vs. Decorator



1..n
<<interface>>
Component
+operation()
1
Composite
-comps:Collection<Component>
+operation()
+addComp(c:Component)
+removeComp(c:Component)
CompA
+operation()
Decorator
-decorated:Component
+Decorator(d:Component)
+operation()