Kỹ Thuật Phần Mềm (Software Engineering)

Software Architecture

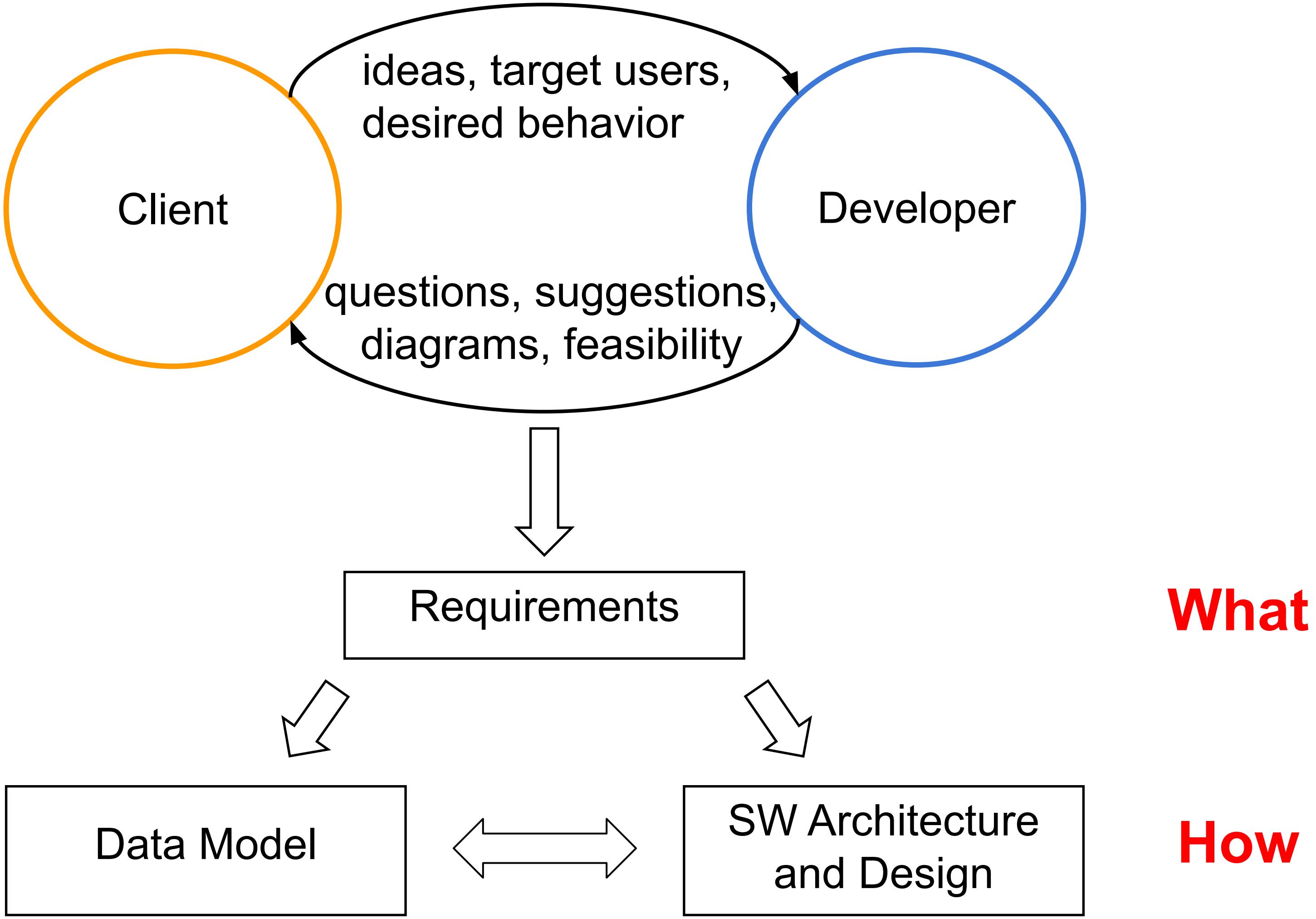
Mai Xuân Tráng, PhD

Khoa Công Nghệ Thông Tin Trường Đại học Phenikaa

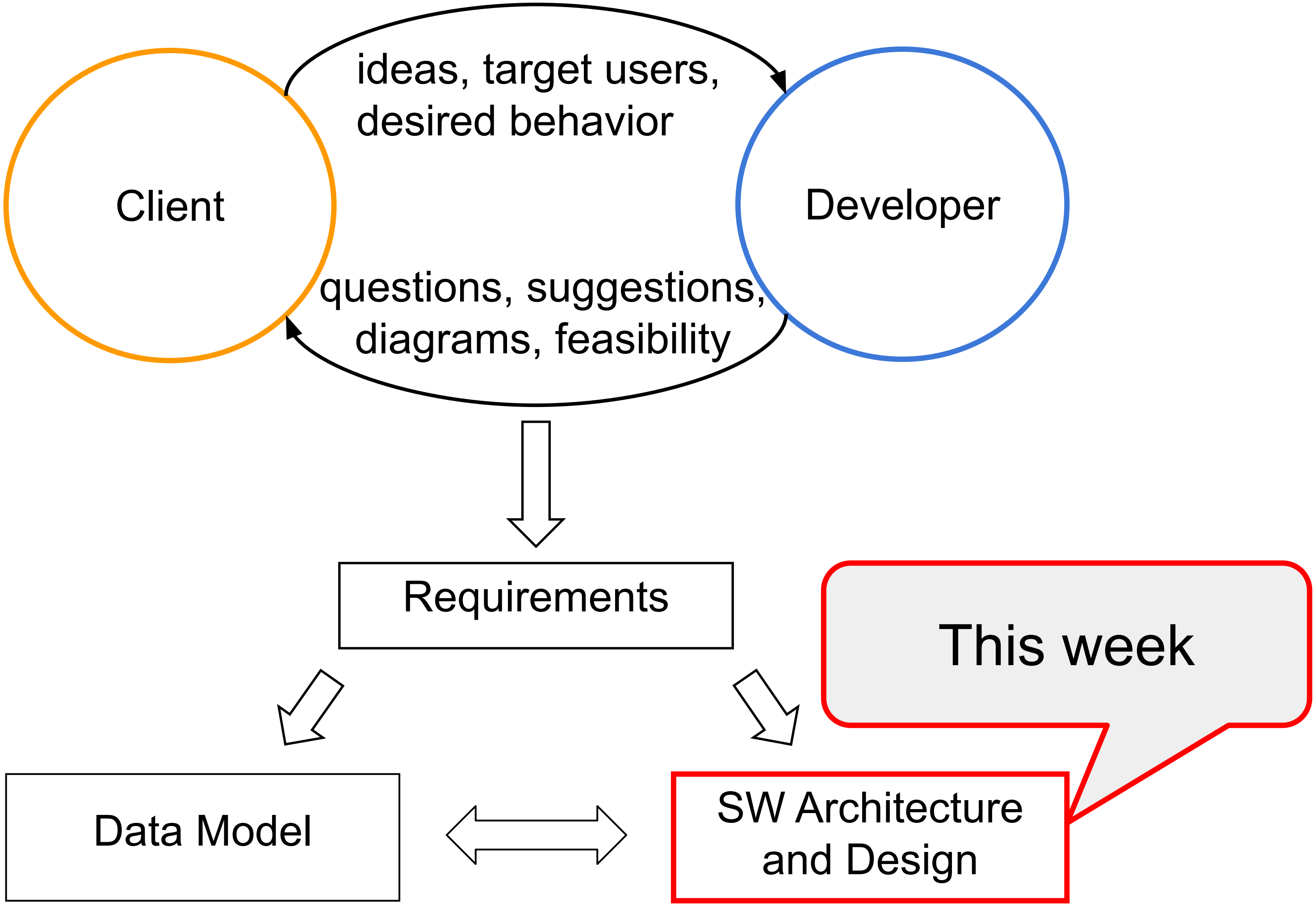
Email: trang.maixuan@phenikaa-uni.edu.vn

SDT: 0965590406

Recap: from Requirements to System Design


ideas, target users, desired behavior
Client
Developer
questions, suggestions, diagrams, feasibility
Requirements
What
Data Model
SW Architecture and Design
How

Recap: from Requirements to System Design



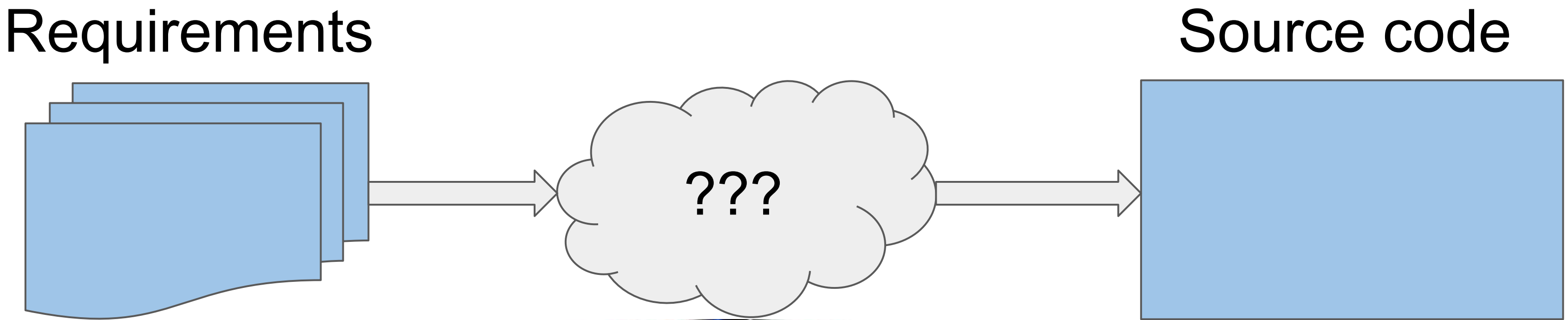
Today

- Software architecture vs. software design
- Common software architecture patterns
- Q&A for requirements and use cases

Software architecture vs. software design

Recall the high-level problem

One solution: “*Here happens a miracle*”



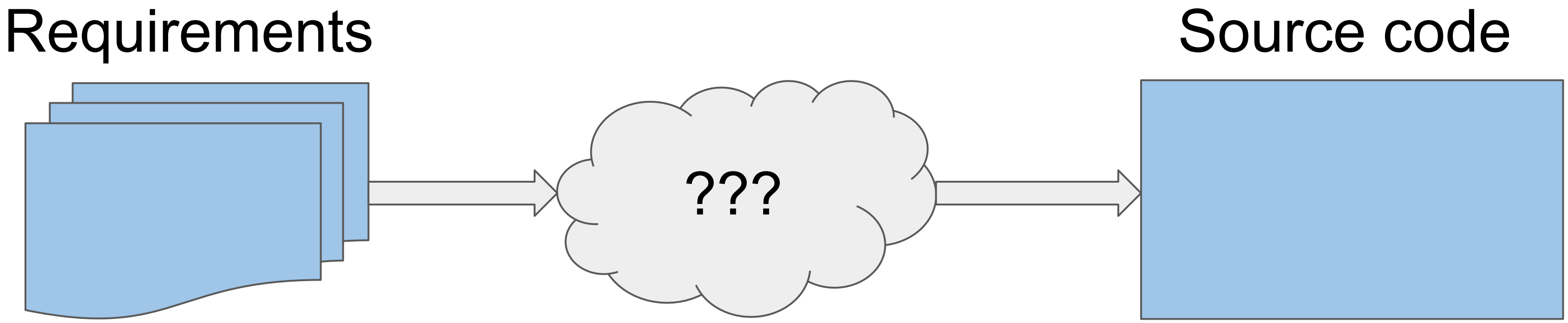
Requirements

???

Source code

Recall the high-level problem

Another solution: Modeling the architecture and design



Requirements

???

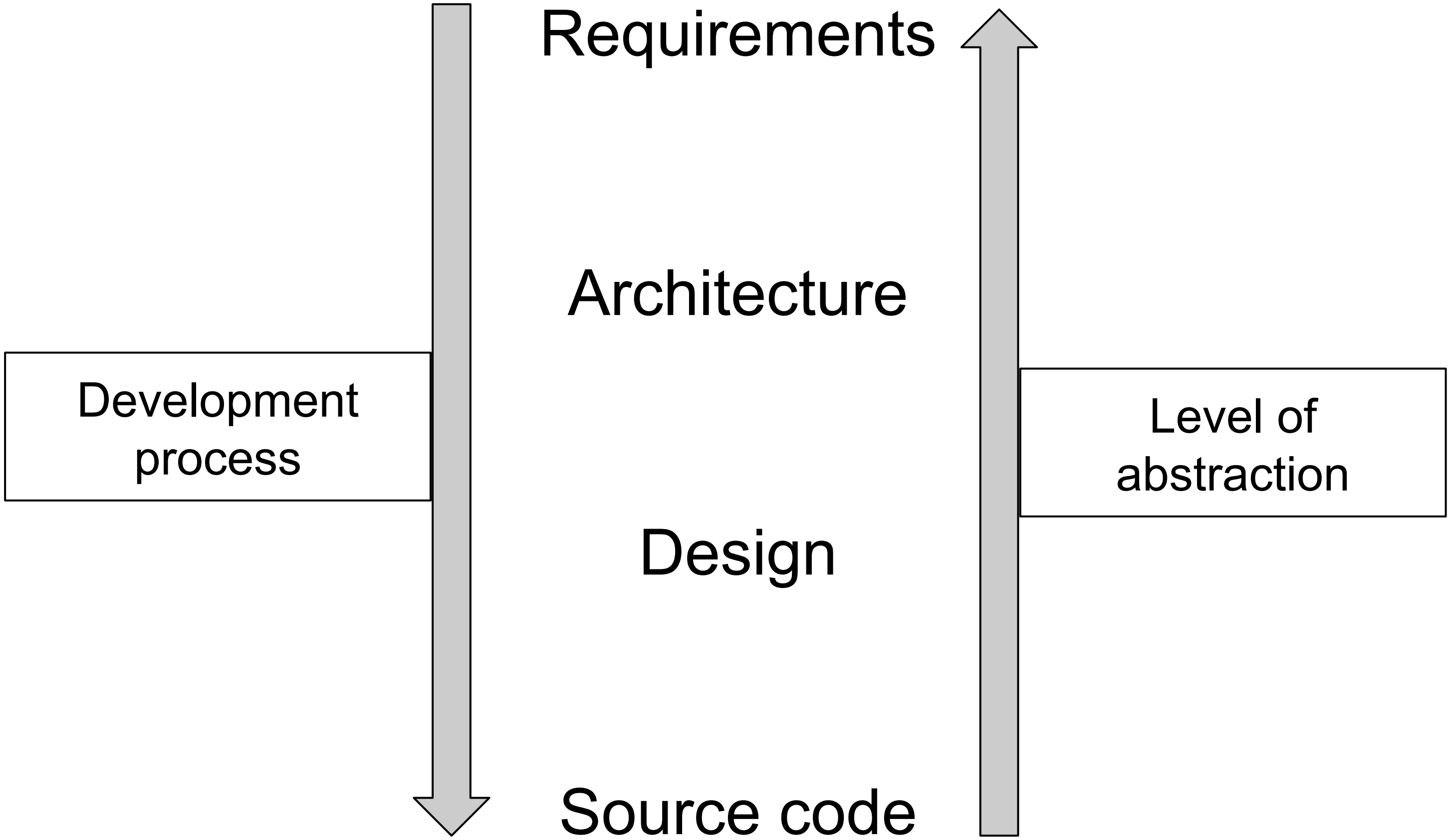
Source code

Why software architecture and design?

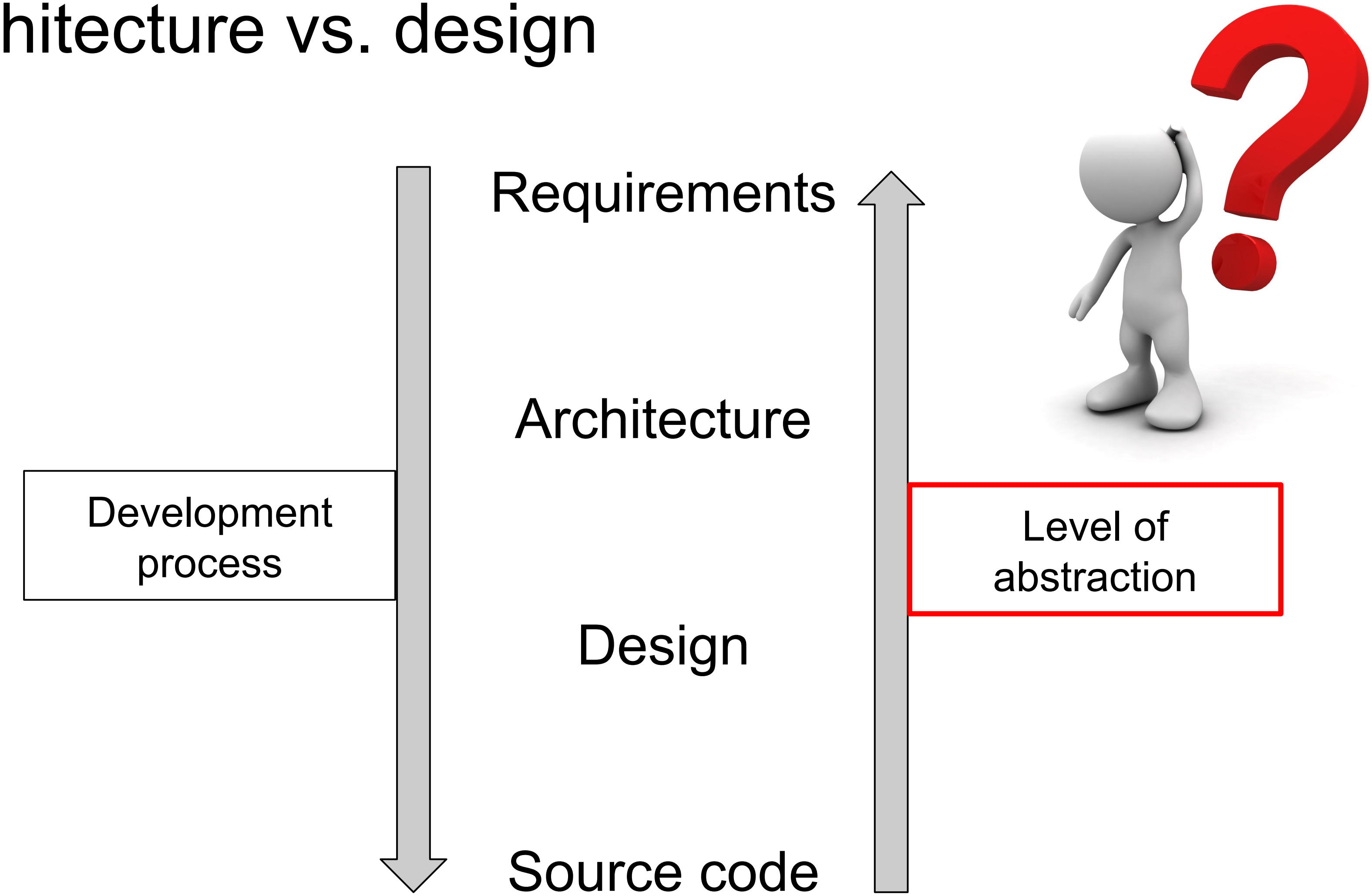
> “There are two ways of constructing a software design:
>
> one way is to make it so simple that there are obviously no deficiencies;
>
> the other is to make it so complicated that there are no obvious deficiencies.” [Tony Hoare]

Goals: separation of concerns and modularity.

Architecture vs. design



Architecture vs. design



Requirements

Architecture

Design

Source code

Development process

Level of abstraction

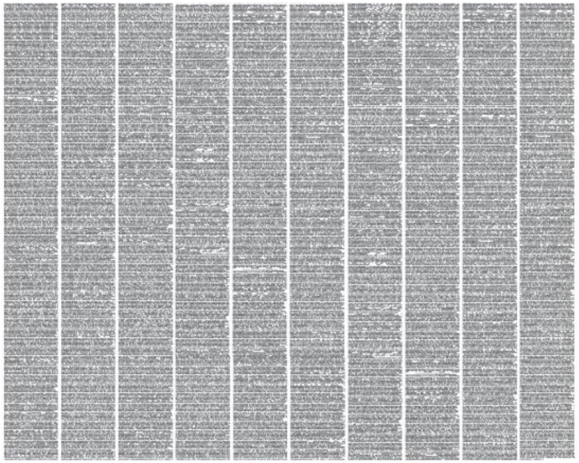
Abstraction

Building an abstract representation of reality

- Ignoring (insignificant) details.
- Focusing on the most important properties.
- Level of abstraction depends on viewpoint and purpose:
  - Communication
  - Component interfaces
  - Verification and validation

Different levels of abstraction

Source code

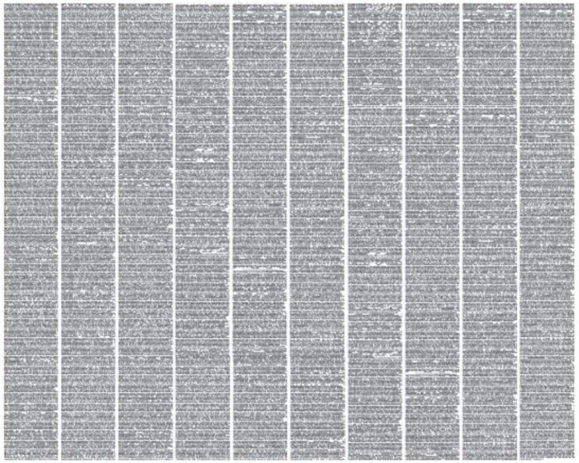


Example: Linux Kernel

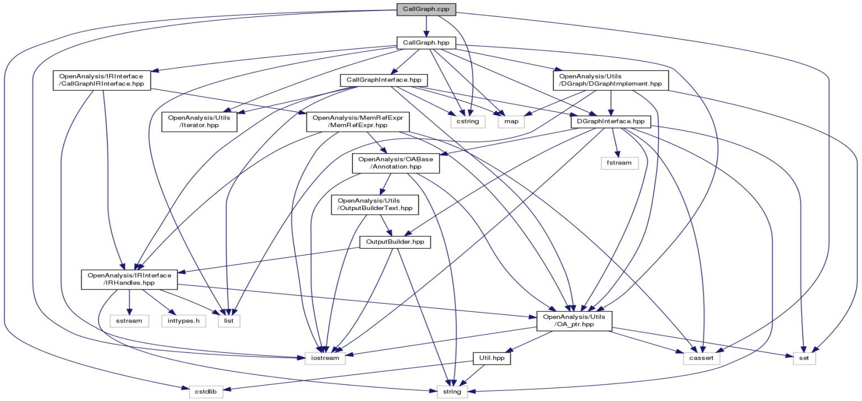
- 16 million Lines of Code!
- What does the code do?
- Are there dependencies?
- Are there different components?

Different levels of abstraction

Source code



Call graph

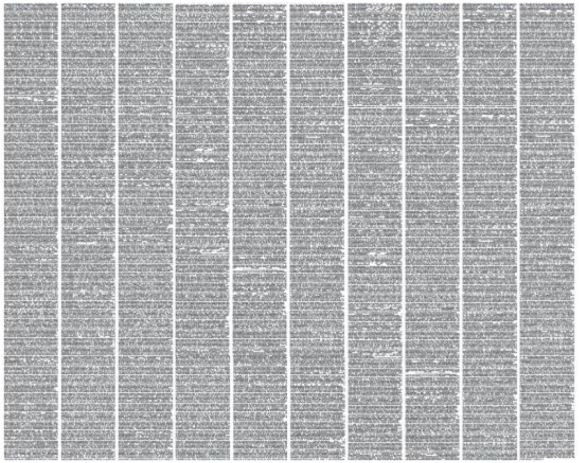

CallGraph.cpp
CallGraph.hpp
OpenAnalysis/IRInterface /CallGraphIRInterface.hpp
CallGraphInterface.hpp
OpenAnalysis/Utils /DGraph/DGraphImplement.hpp
OpenAnalysis/Utils /Iterator.hpp
OpenAnalysis/MemRefExpr /MemRefExpr.hpp
cstring
map
DGraphInterface.hpp
OpenAnalysis/OABase /Annotation.hpp
fstream
OpenAnalysis/Utils /OutputBuilderText.hpp
OutputBuilder.hpp
OpenAnalysis/IRInterface /IRHandles.hpp
sstream
inttypes.h
list
OpenAnalysis/Utils /OA_ptr.hpp
iostream
Util.hpp
cassert
set
cstdlib
string

Example: Linux Kernel

- 16 million Lines of Code!
- What does the code do?
- **Are there dependencies?**
- Are there different components?

Different levels of abstraction

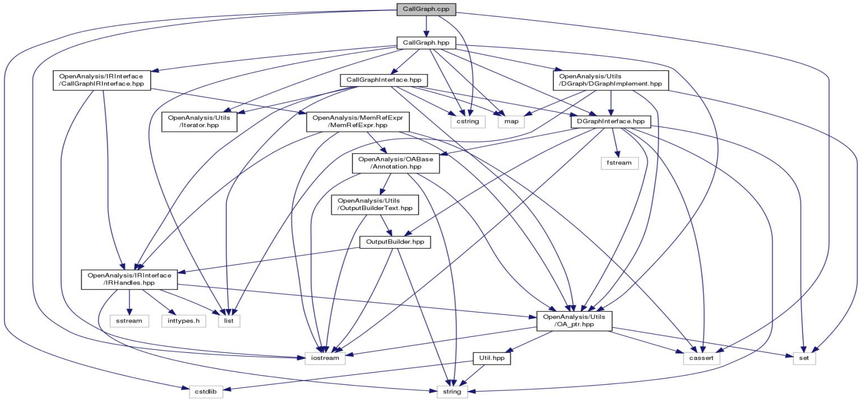
Source code



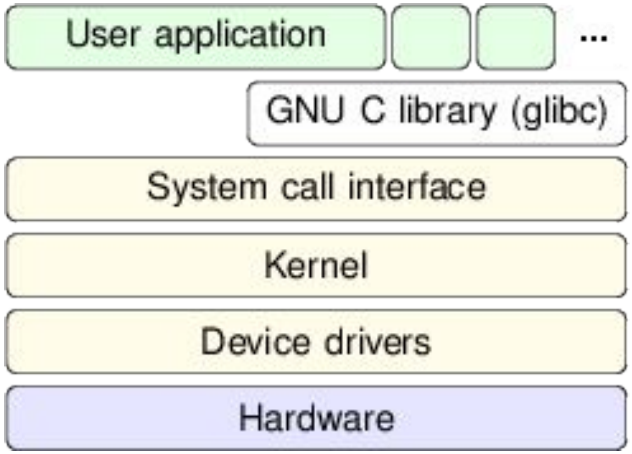
Example: Linux Kernel

- 16 million Lines of Code!
- What does the code do?
- Are there dependencies?
- **Are there different components?**

Call graph


CallGraph.cpp
CallGraph.hpp
OpenAnalysis/IRInterface /CallGraphIRInterface.hpp
CallGraphInterface.hpp
OpenAnalysis/Utils /DGraph/DGraphImplement.hpp
OpenAnalysis/Utils /Iterator.hpp
OpenAnalysis/MemRefExpr /MemRefExpr.hpp
cstring
map
DGraphInterface.hpp
OpenAnalysis/OABase /Annotation.hpp
fstream
OpenAnalysis/Utils /OutputBuilderText.hpp
OutputBuilder.hpp
OpenAnalysis/IRInterface /IRHandles.hpp
sstream
inttypes.h
list
OpenAnalysis/Utils /OA_ptr.hpp
iostream
Util.hpp
cassert
set
cstdlib
string

Layer diagram



User application

...

GNU C library (glibc)

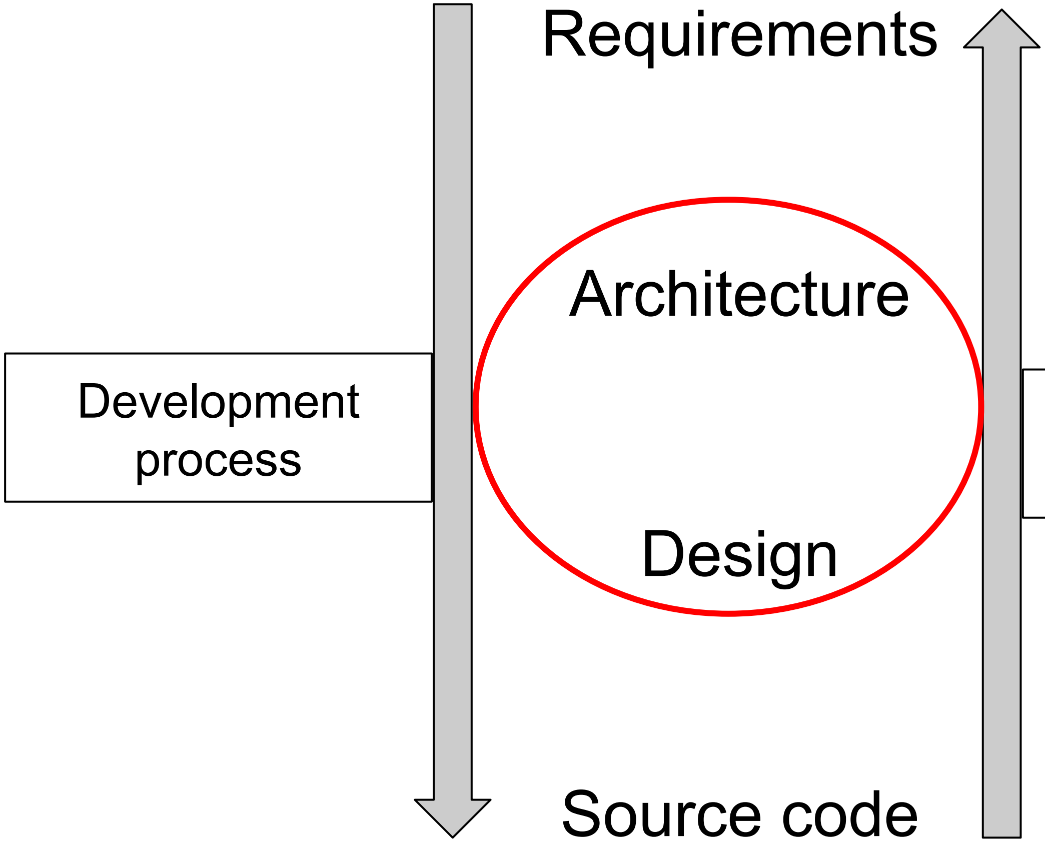
System call interface

Kernel

Device drivers

Hardware

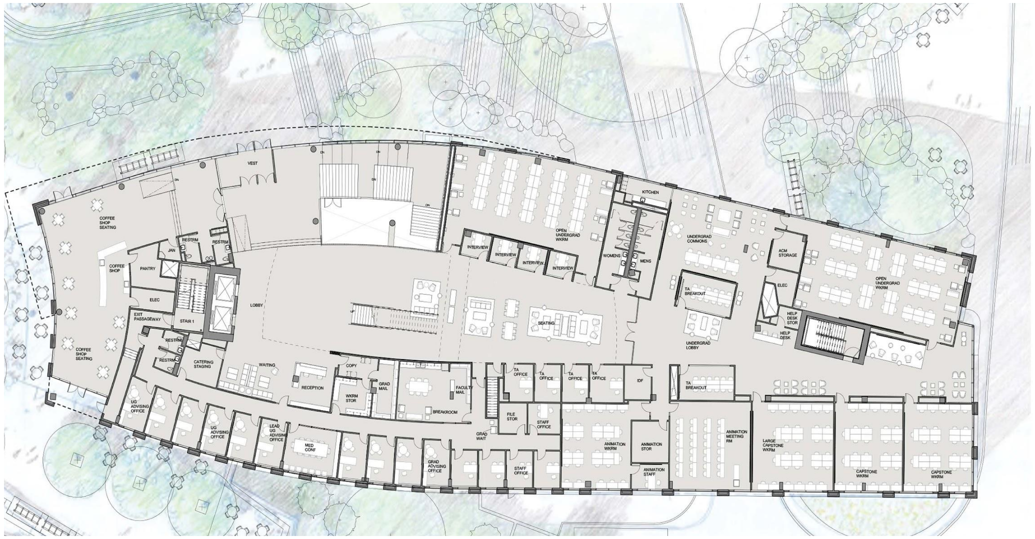
Architecture vs. design



What's the difference?

Architecture vs. design

Architecture



Design

Software architecture vs. design

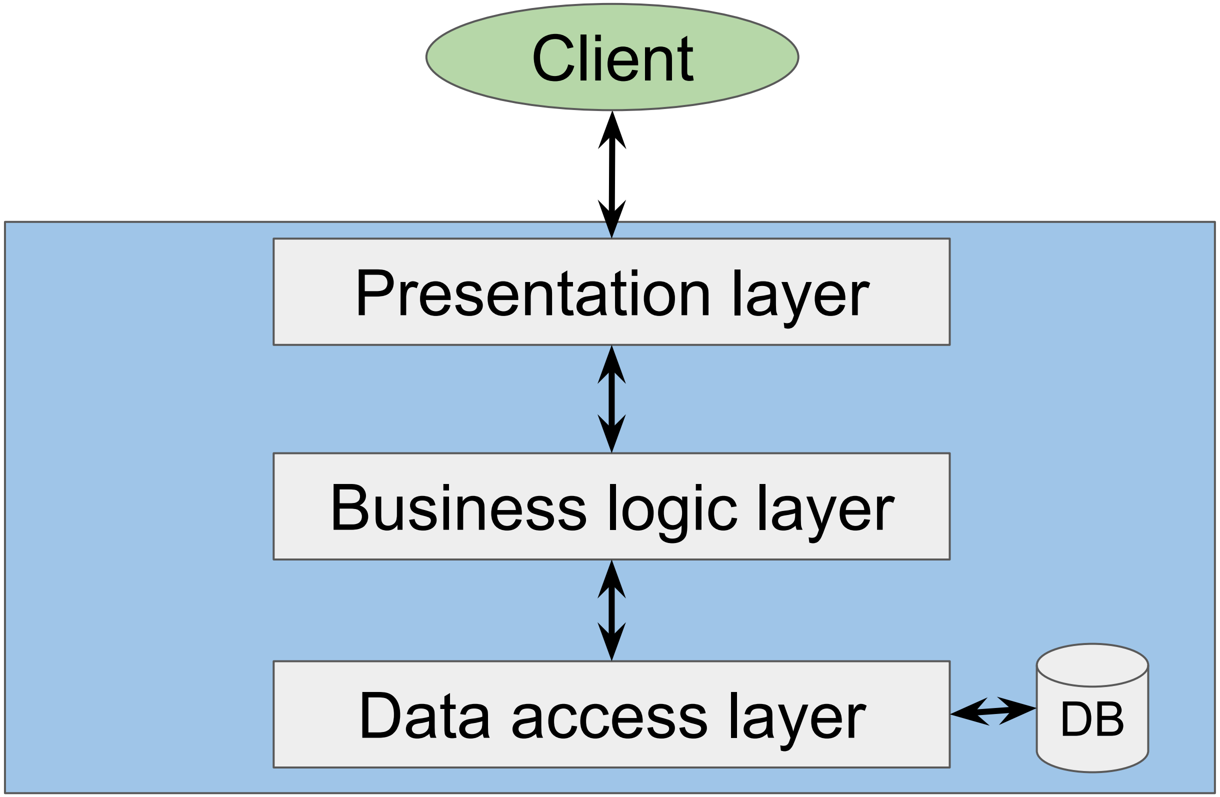
Architecture (what is developed?)

- High-level view of the overall system:
  - What components do exist?
  - What are the protocols between components?
  - What type of storage etc.?

Design (how are the components developed?)

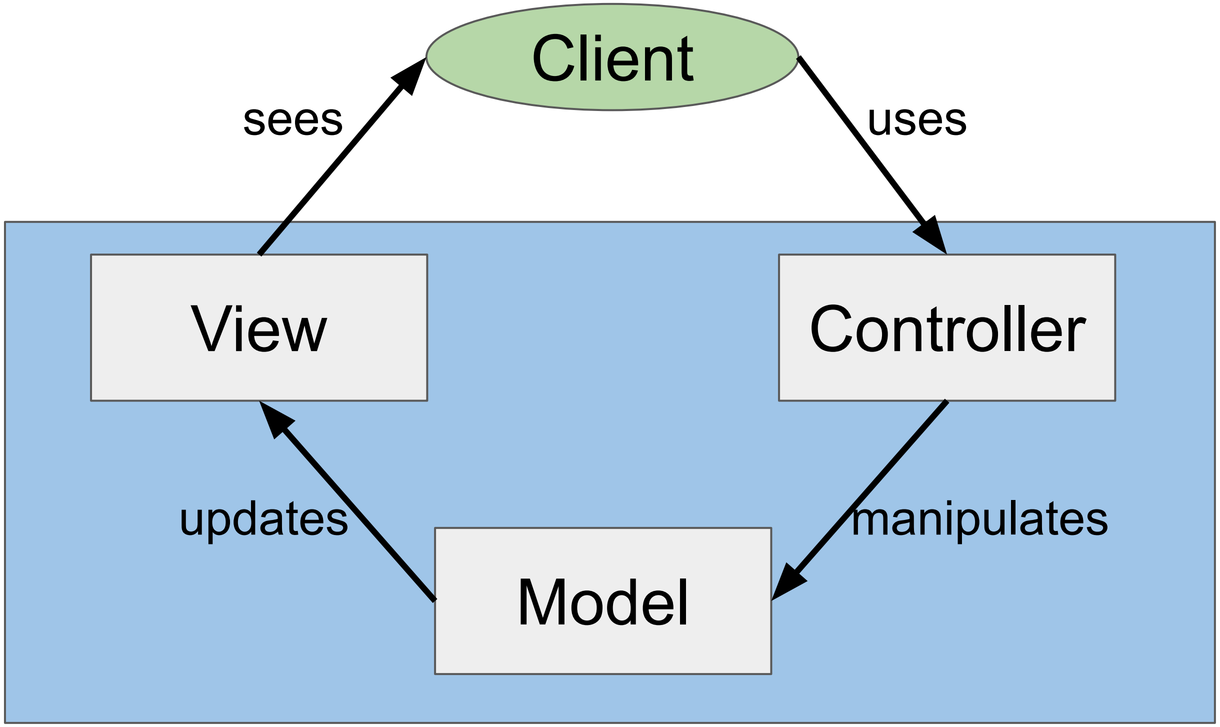
- Considers individual components:
  - Data representation
  - Interfaces, Class hierarchy
  - …

Software architecture: Client-server / n-tier



Simplifies reusability, exchangeability, and distribution.

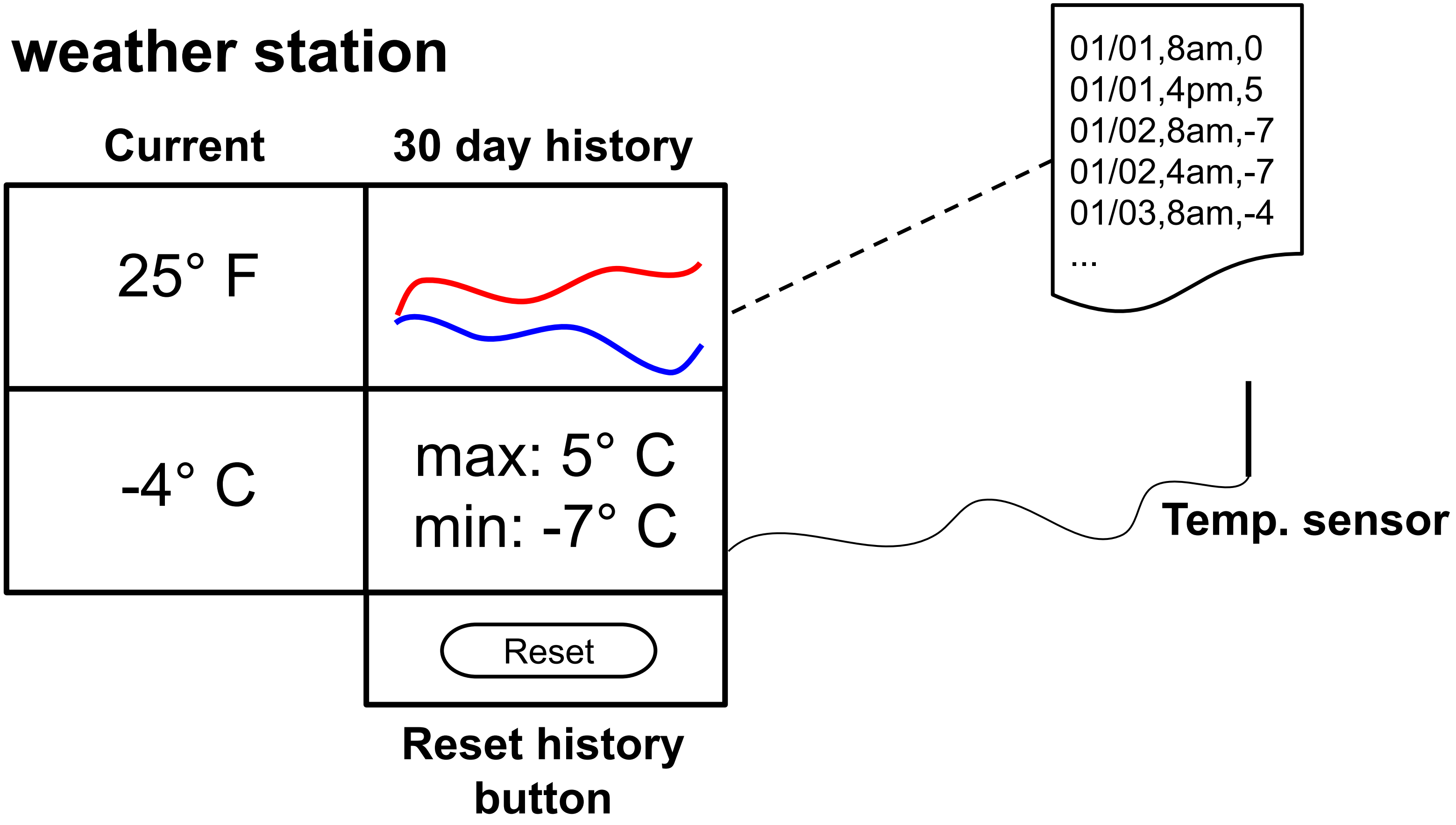
Software architecture: Model View Controller (MVC)



Separates data representation (Model), visualization (View), and client interaction (Controller)

Model View Controller: example

Simple weather station



Current

30 day history

25° F

-4° C

max: 5° C
min: -7° C

Reset

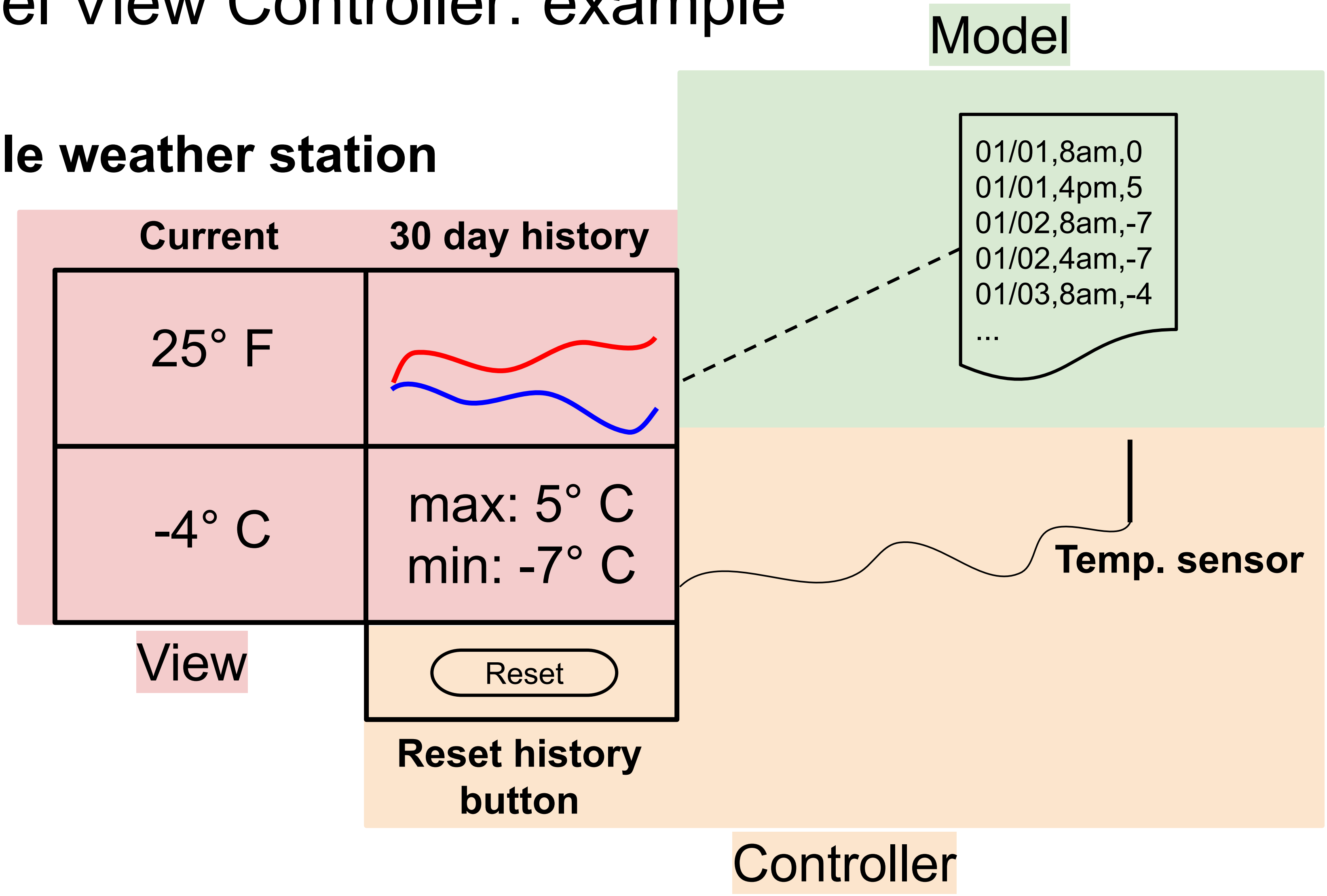
Reset history button

01/01,8am,0
01/01,4pm,5
01/02,8am,-7
01/02,4am,-7
01/03,8am,-4
...

Temp. sensor

Model View Controller: example

Simple weather station



Model

01/01,8am,0
01/01,4pm,5
01/02,8am,-7
01/02,4am,-7
01/03,8am,-4
...

Current

30 day history

25° F

-4° C

max: 5° C
min: -7° C

View

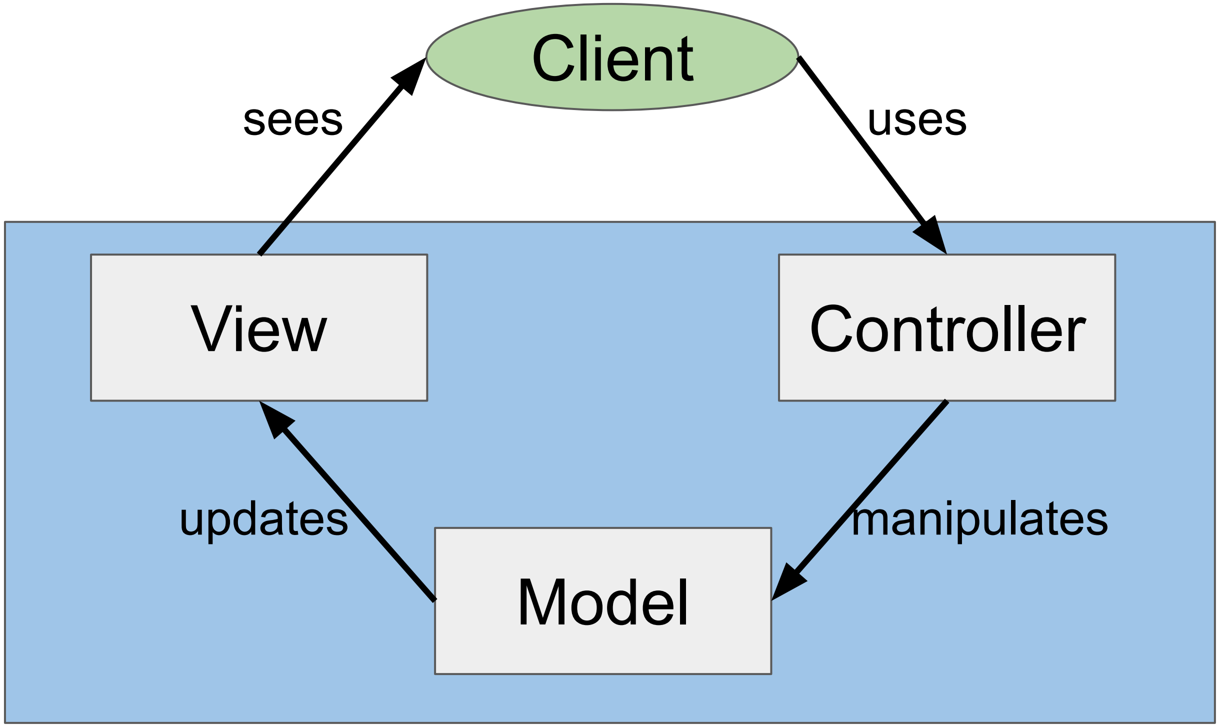
Reset

Reset history button

Temp. sensor

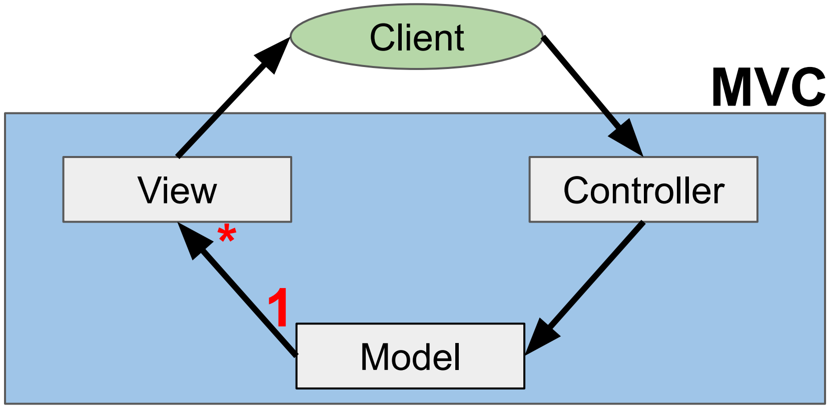
Controller

Software architecture: Model View Controller (MVC)



Separates data representation (Model),
visualization (View), and client interaction (Controller)

MVC vs. MVP vs. MVVM


MVC

Client

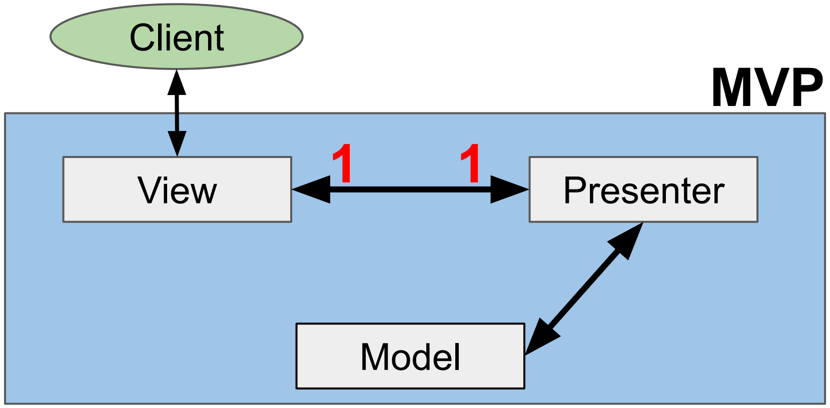
View

Controller

Model

*

1


MVP

Client

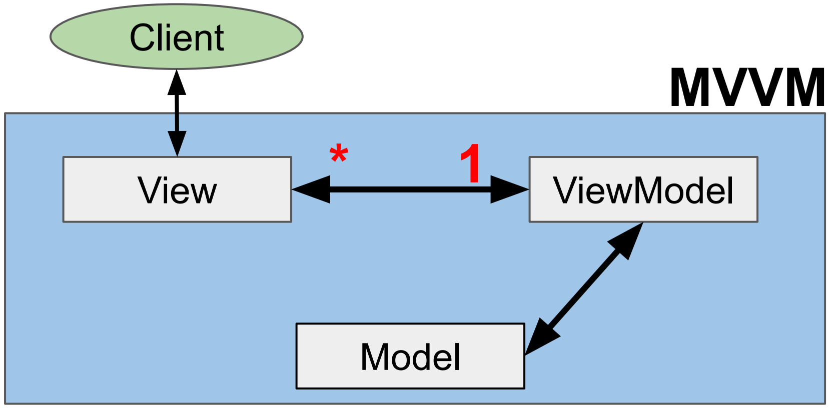
View

Presenter

Model

1

1


MVVM

Client

View

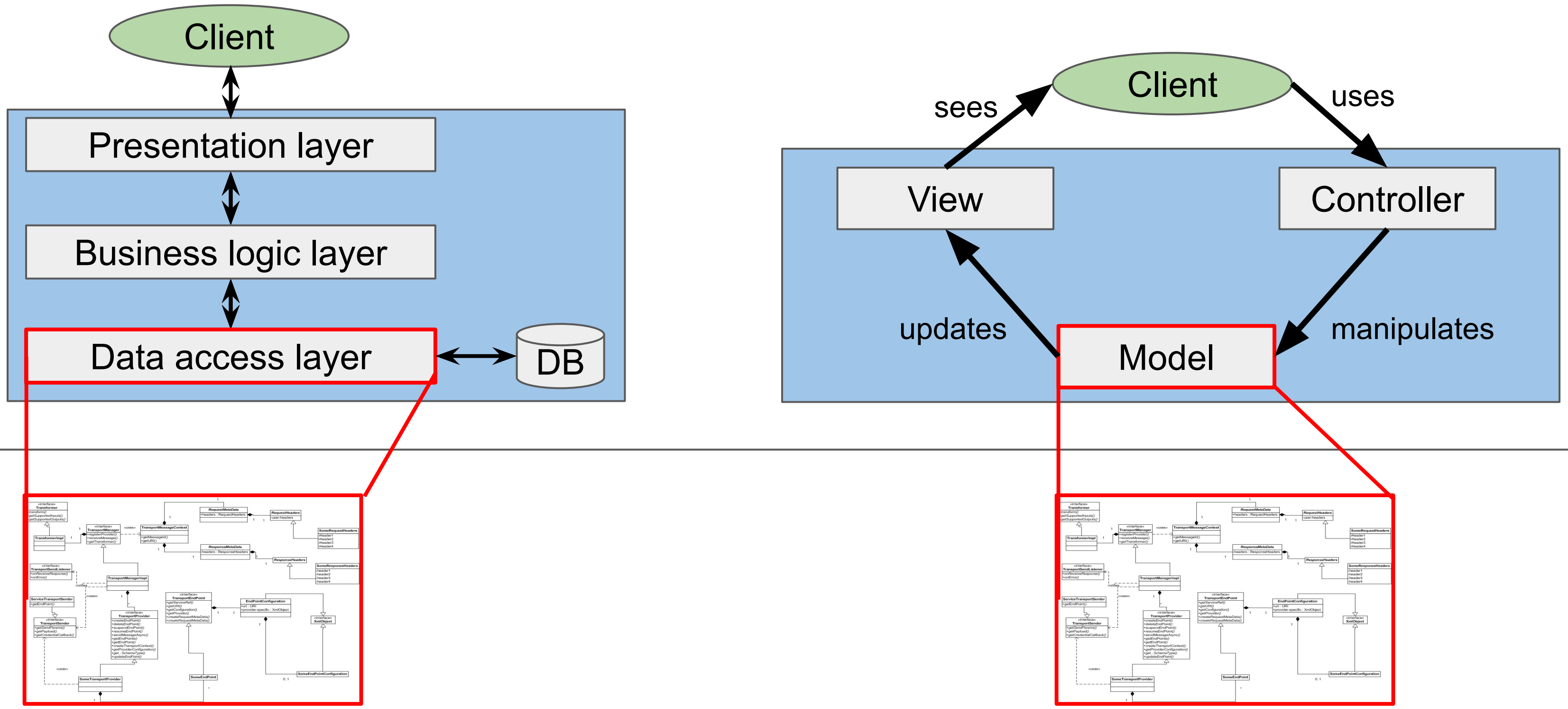
ViewModel

Model

*

1

Software architecture vs. design: summary



Architecture and design

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

Q & A

UML crash course

UML crash course

The main questions

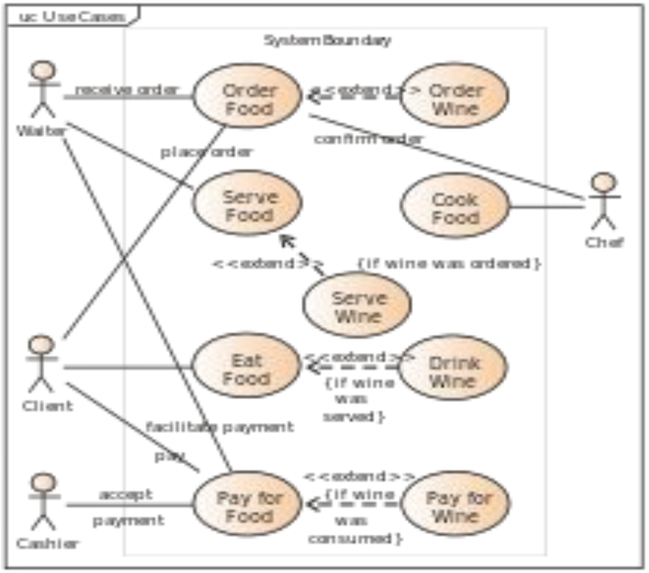
- What is UML?
- Is it useful, why bother?
- When to (not) use UML?

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
  - ...

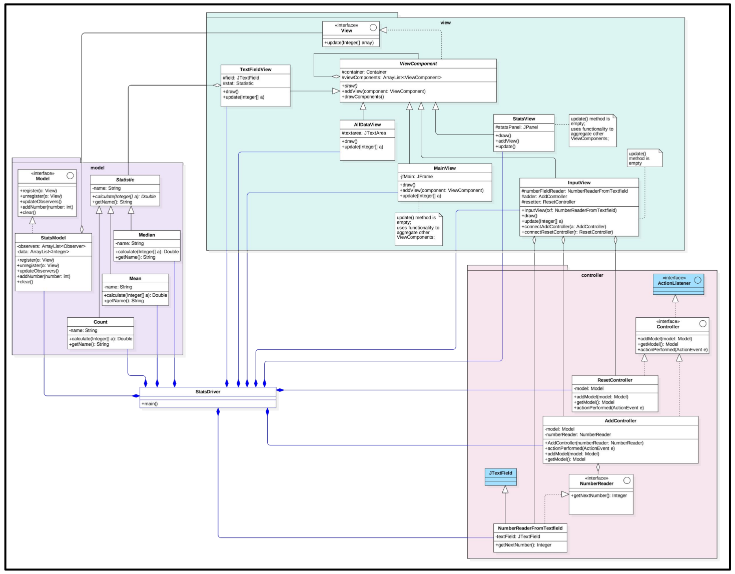
What is UML?

- 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
  - ...

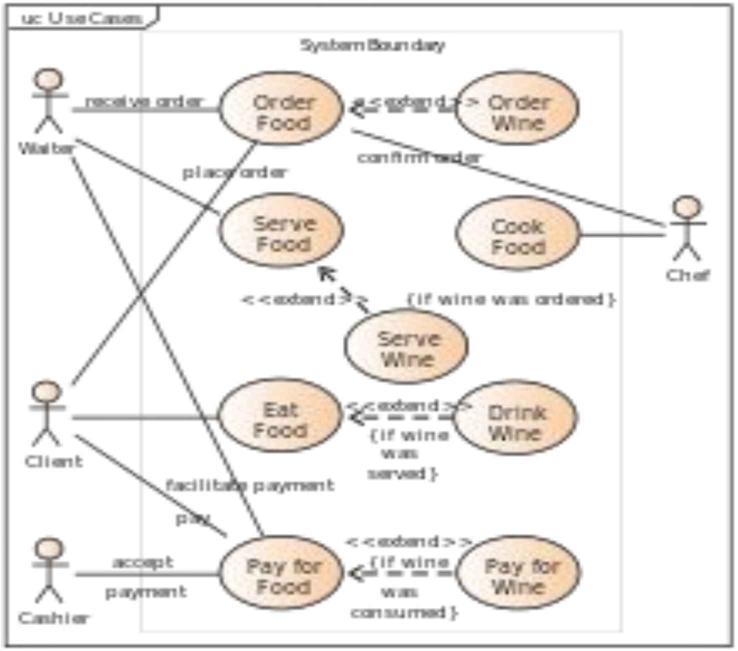


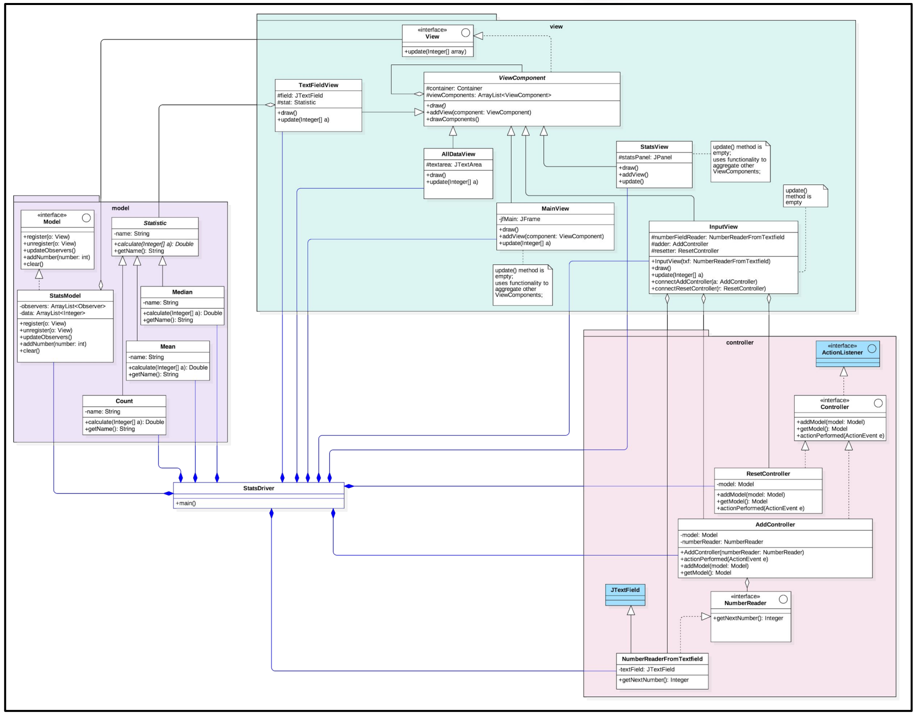
What is UML?

- 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

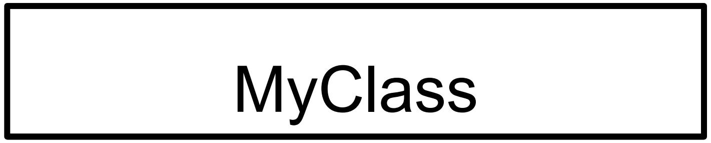
Class

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

Object

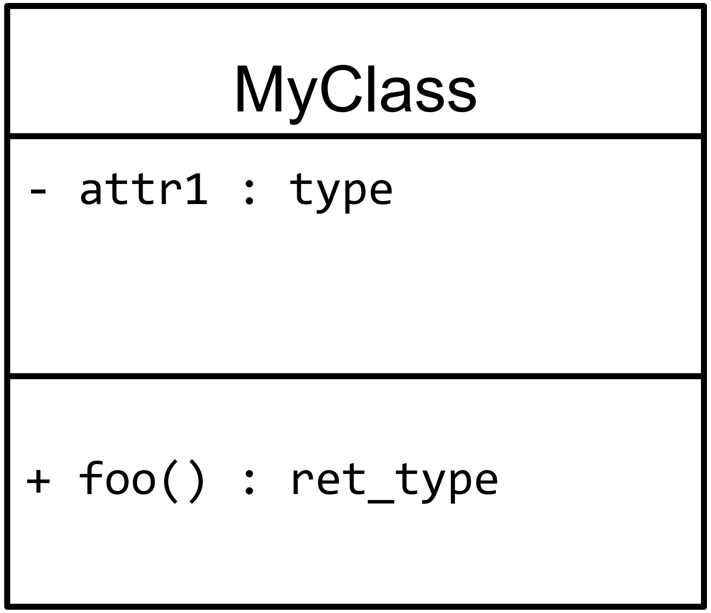
- Entity from the real world.
- Instance of a class
  - Student: Joe (4711), Jane (4712), …
  - Car: Audi A6, Honda Civic, ...

UML class diagram: basic notation



MyClass

UML class diagram: basic notation



MyClass

- attr1 : type

+ foo() : ret_type

Name

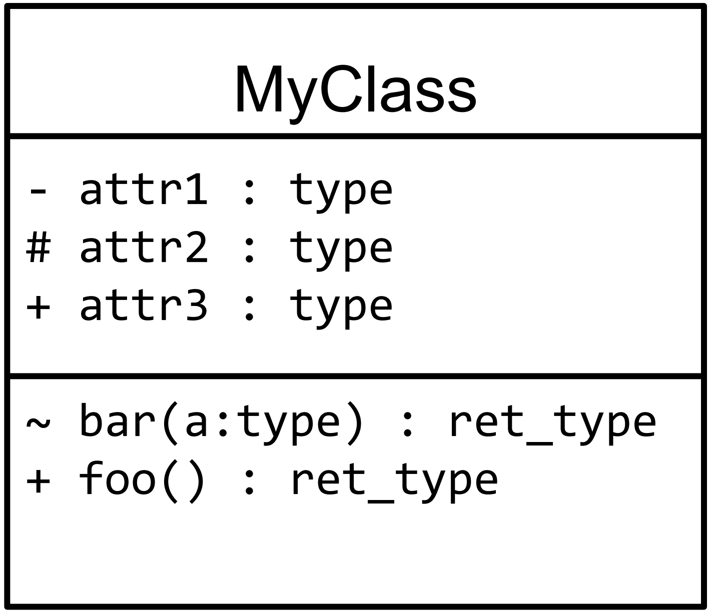
Attributes

<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>

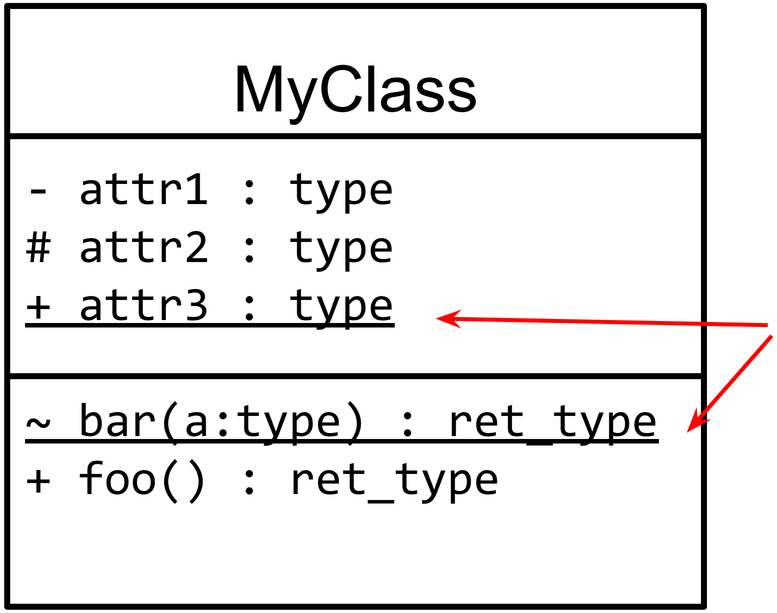
Methods

<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>

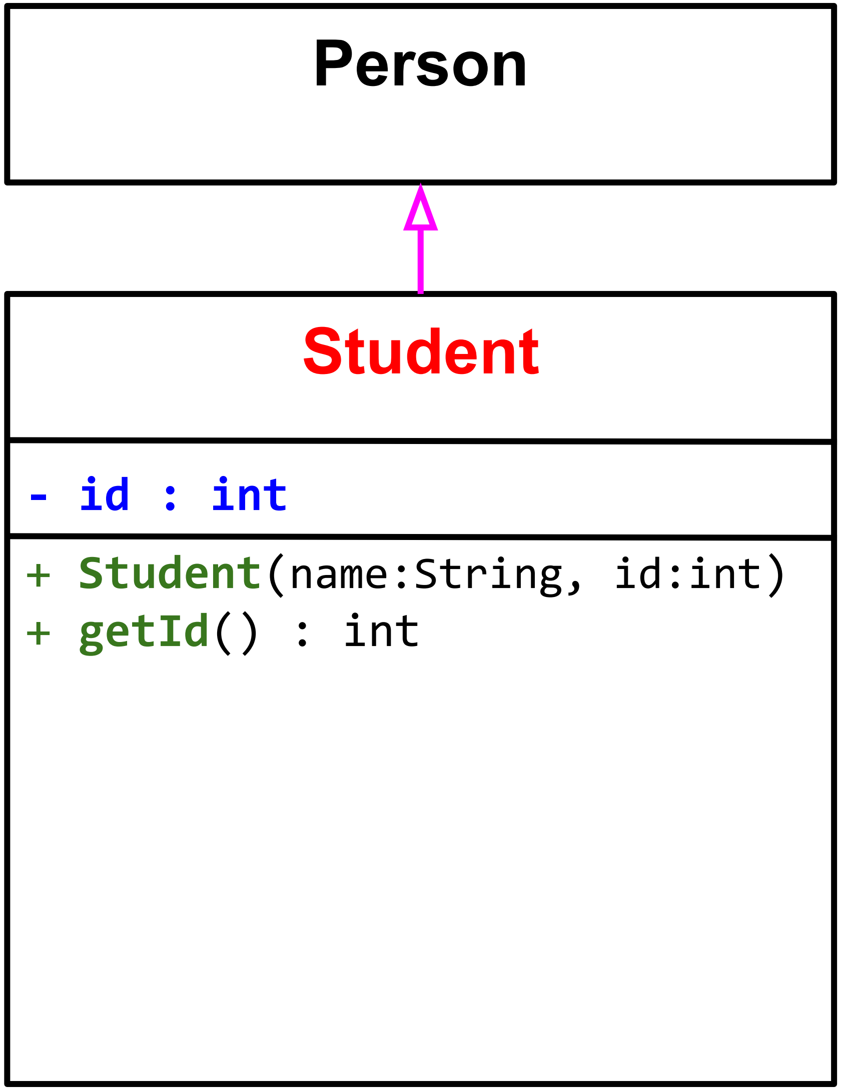
Visibility

- private
~ package-private
# protected
+ public

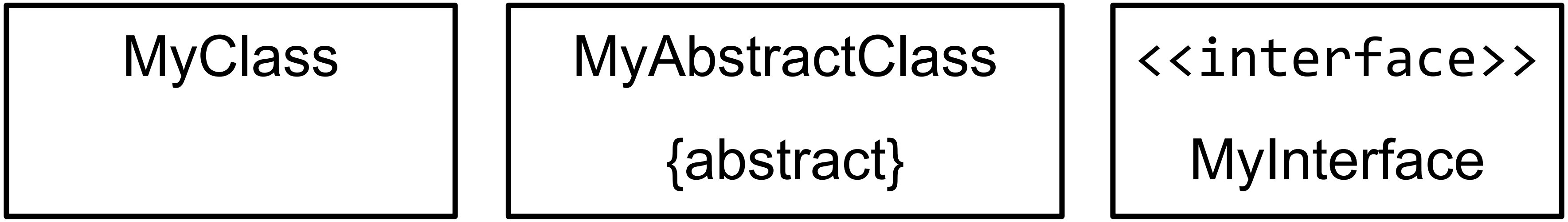
UML class diagram: concrete example

```java
public class Person {
  …
}
```

```java
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



MyClass

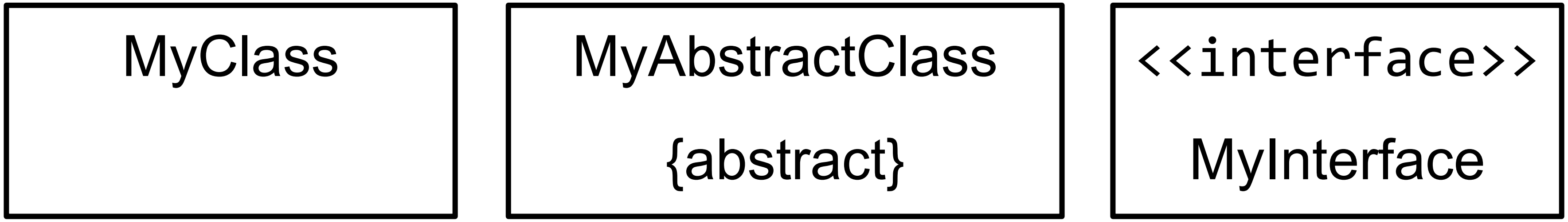
MyAbstractClass

{abstract}

<<interface>>

MyInterface

Classes, abstract classes, and interfaces



MyClass

MyAbstractClass

{abstract}

<<interface>>

MyInterface

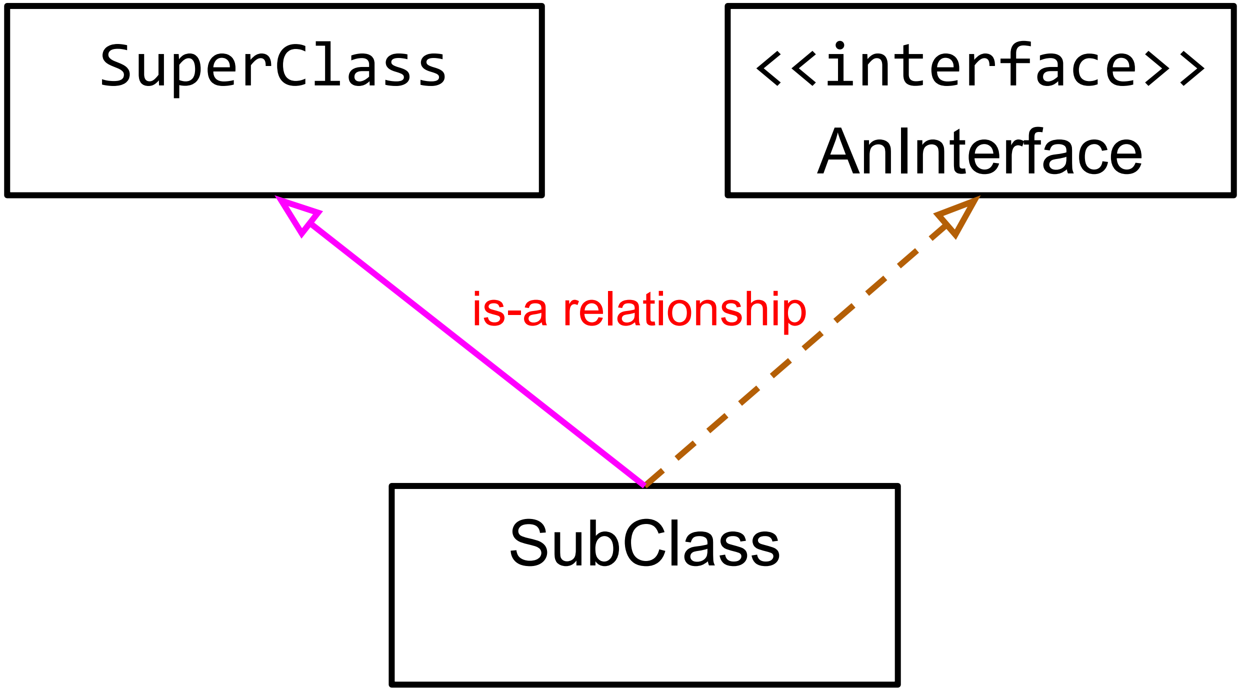
```
public class MyClass {

  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.

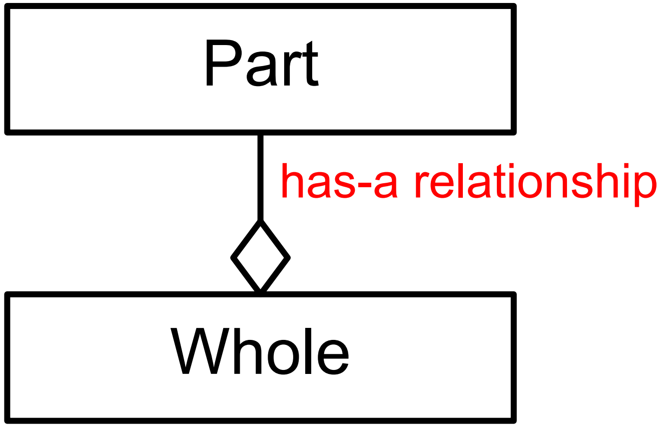
UML class diagram: Inheritance



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

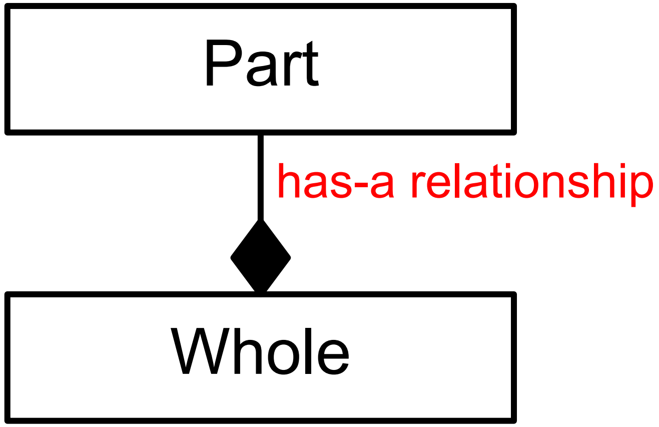
UML class diagram: Aggregation and Composition

Aggregation



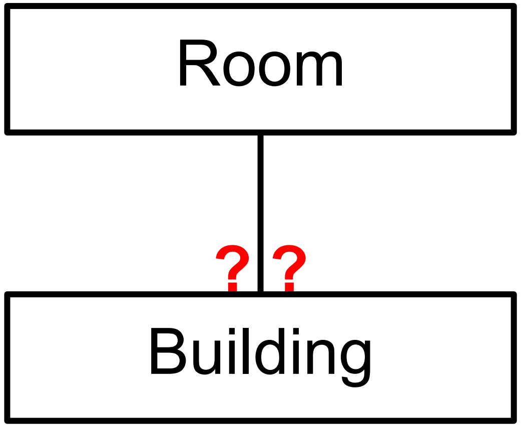
- Existence of Part does not depend on the existence of Whole.
- Lifetime of Part does not depend on Whole.
- No single instance of whole is the unique owner of Part (might be shared with other instances of Whole).

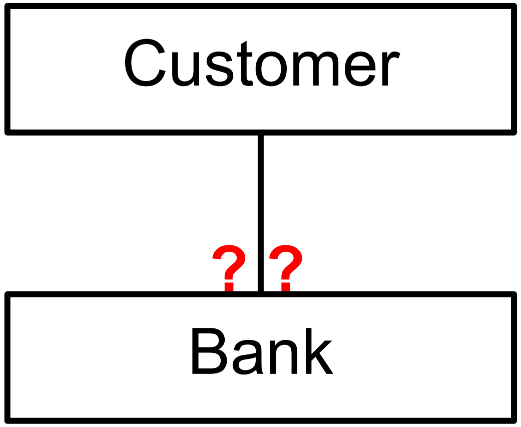
Composition



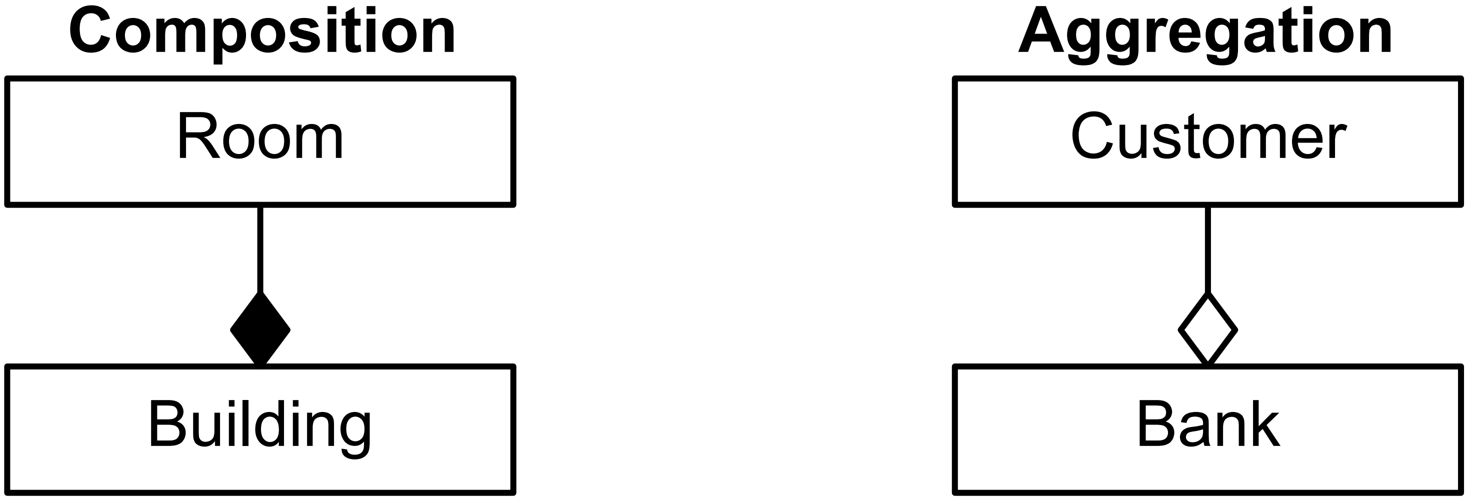
- Part cannot exist without Whole.
- Lifetime of Part depends on Whole.
- One instance of Whole is the single owner of Part.

Aggregation or Composition?



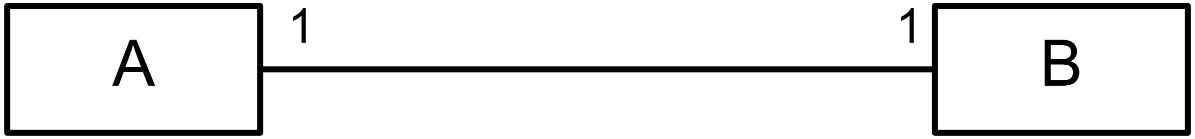


Aggregation or Composition?

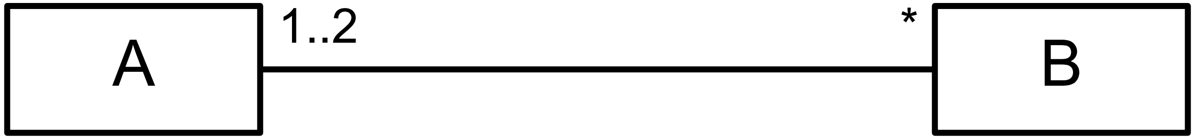


What about class and students or body and body parts?

UML class diagram: multiplicity

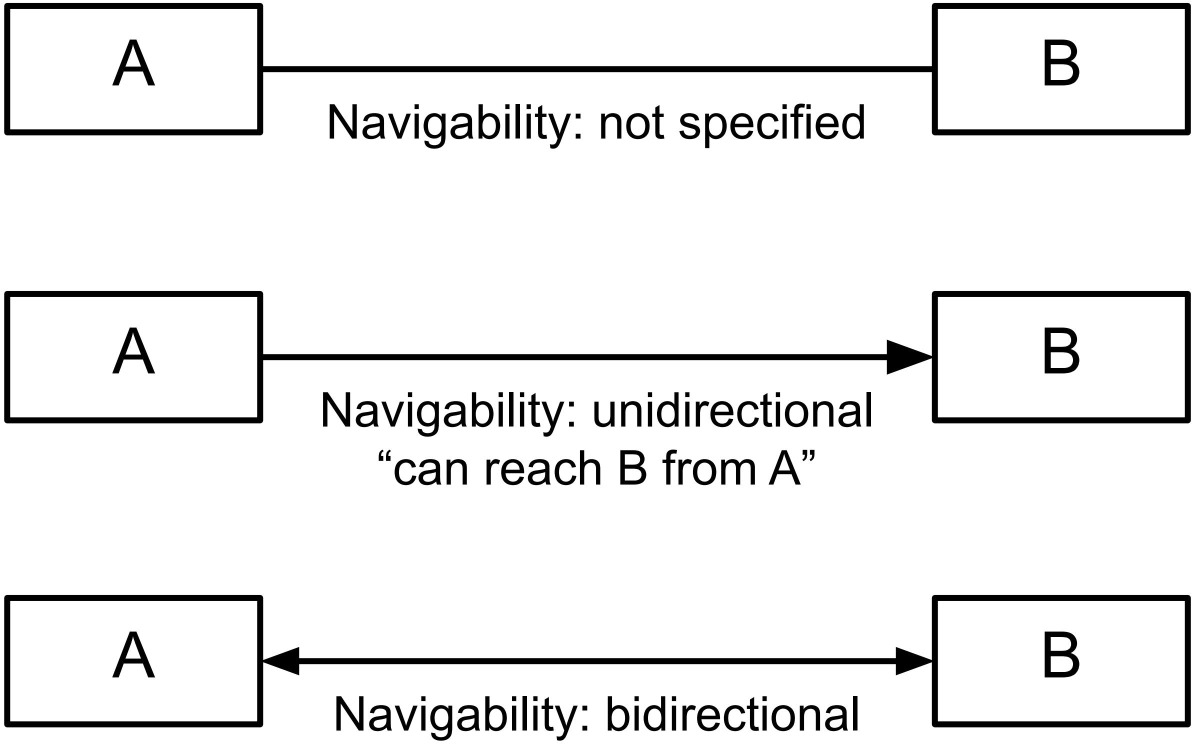


Each A is associated with exactly one B
Each B is associated with exactly one A



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

UML class diagram: navigability

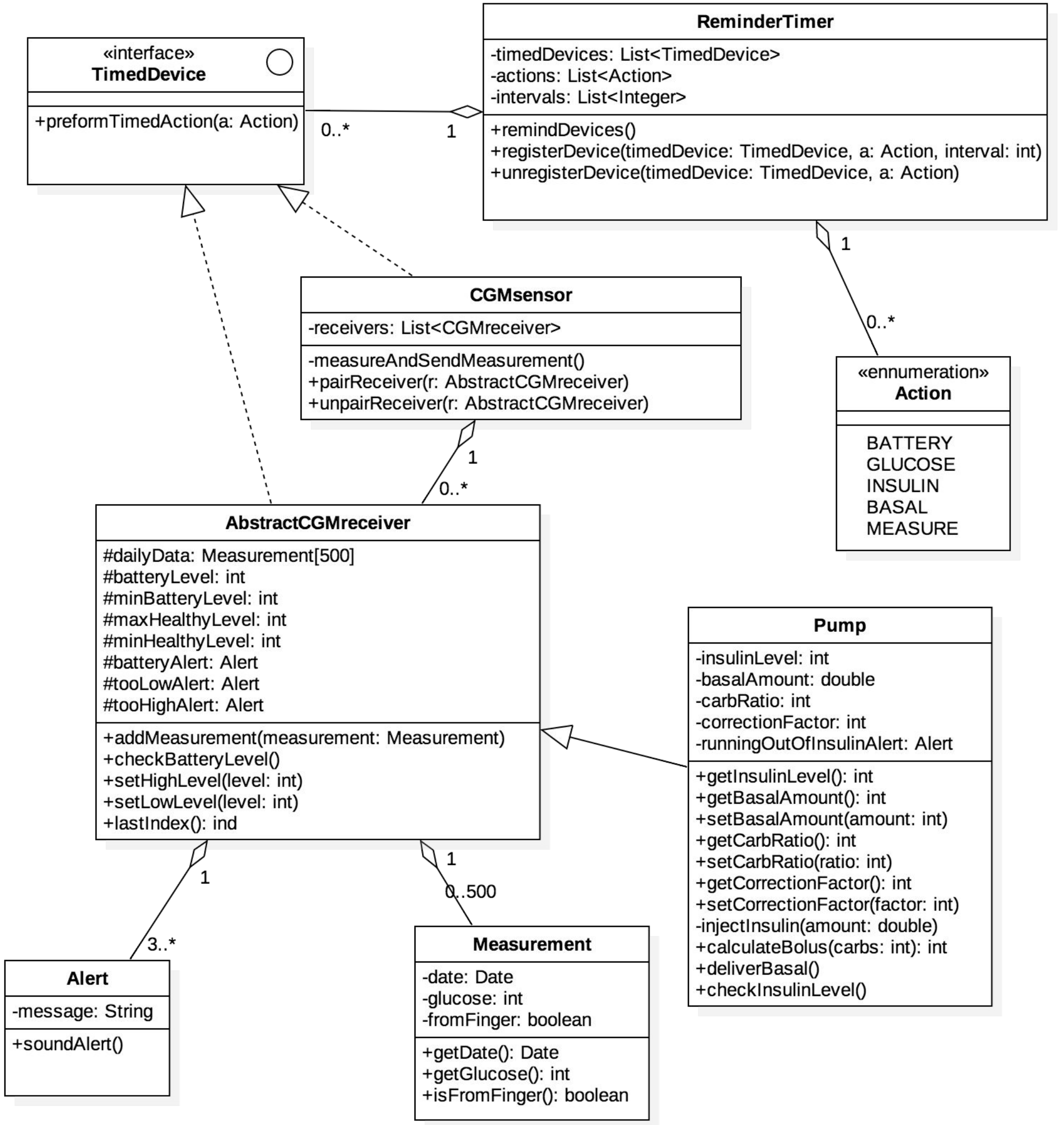


Navigability: not specified

Navigability: unidirectional
“can reach B from A”

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

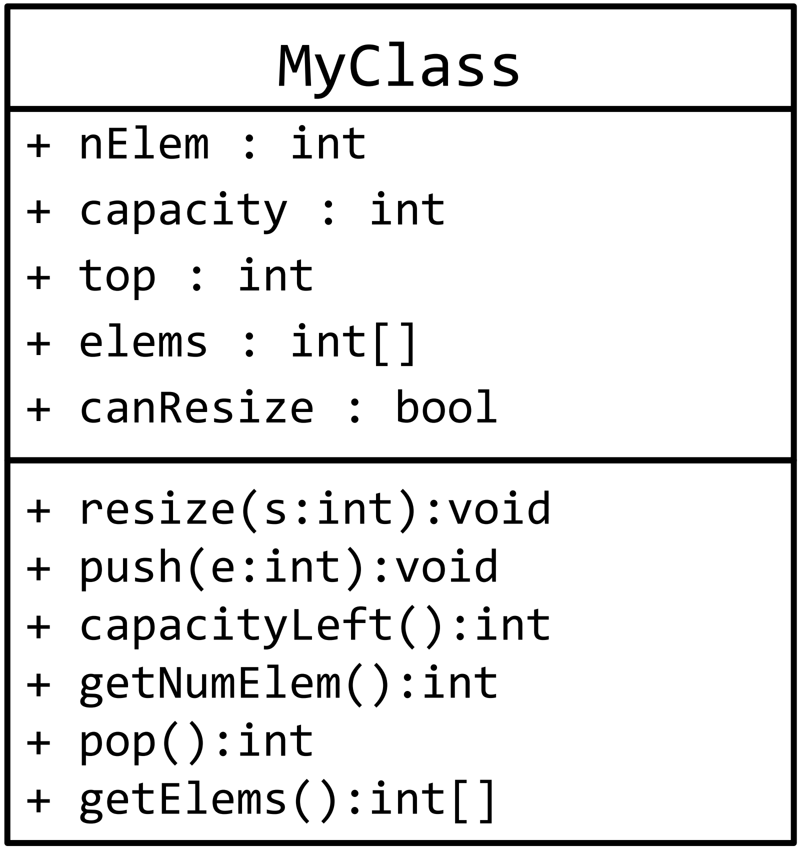
- Unified notation for modeling OO systems.
- Allows different levels of abstraction.
- Suitable for design discussions and documentation.

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



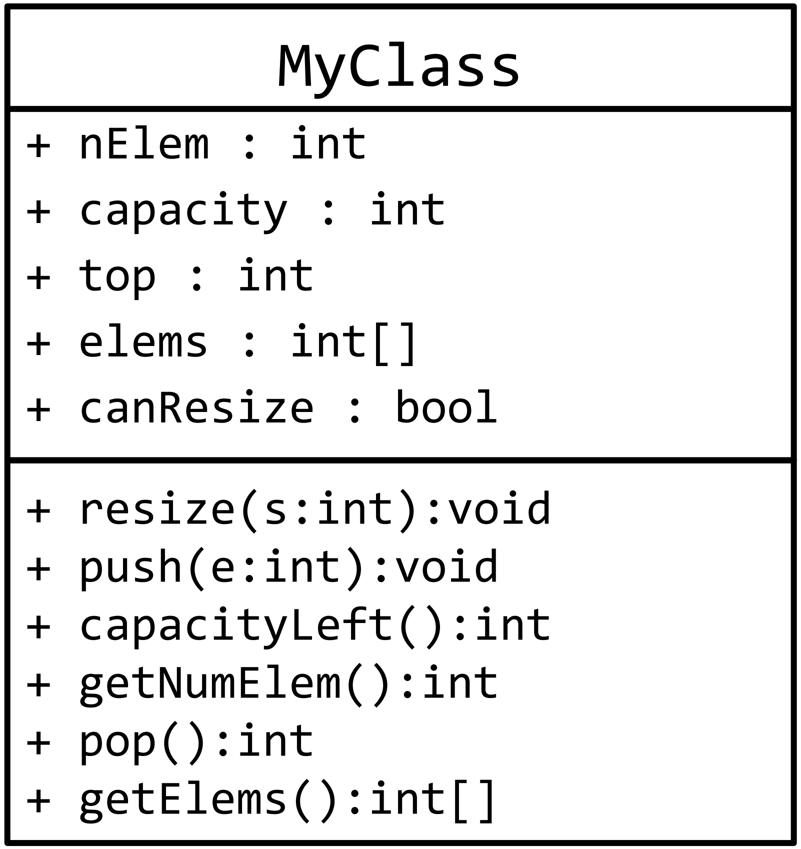
MyClass

+ 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

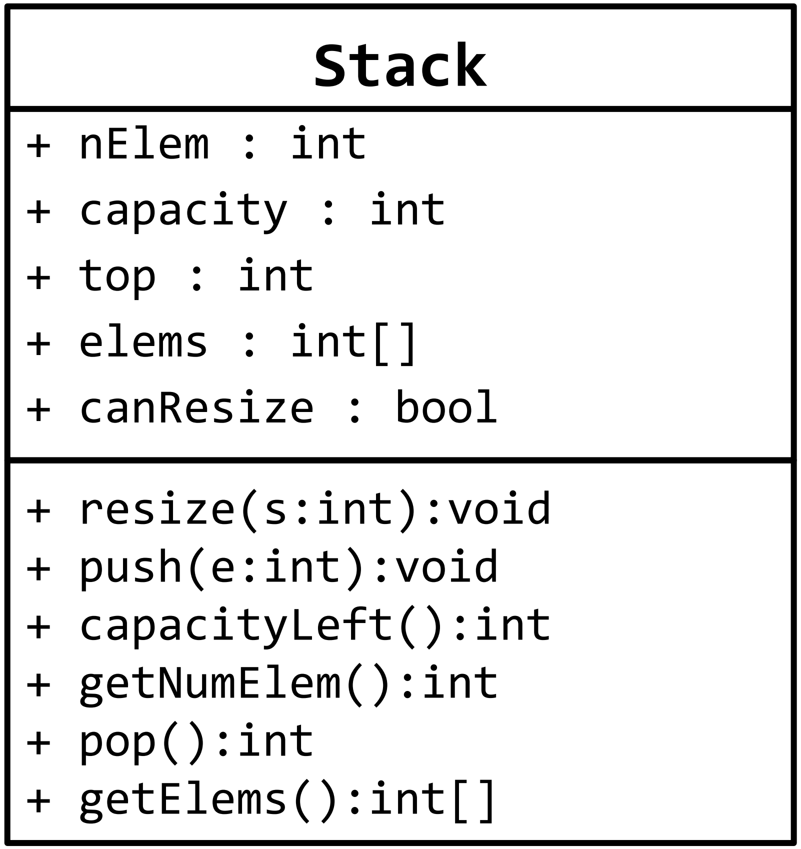


```
MyClass
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

```java
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

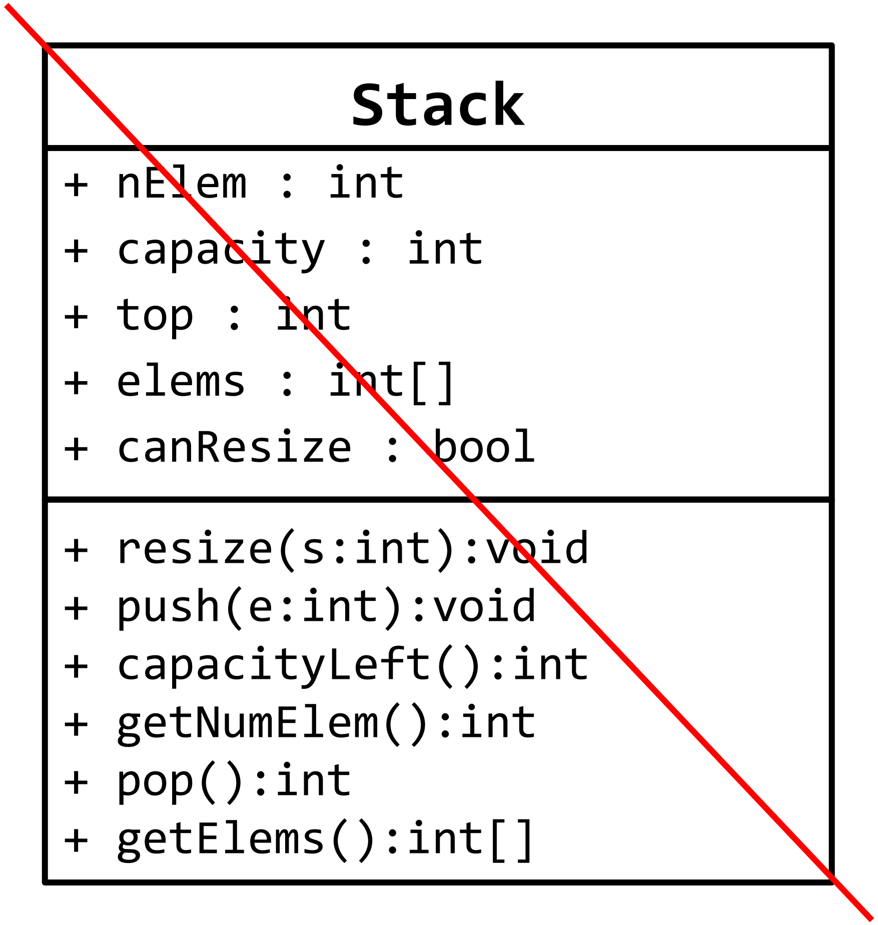
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool

+ 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?

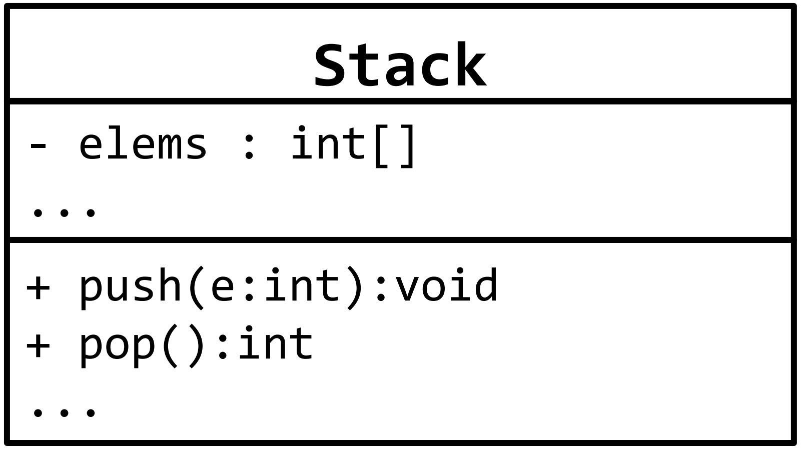
Information hiding



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

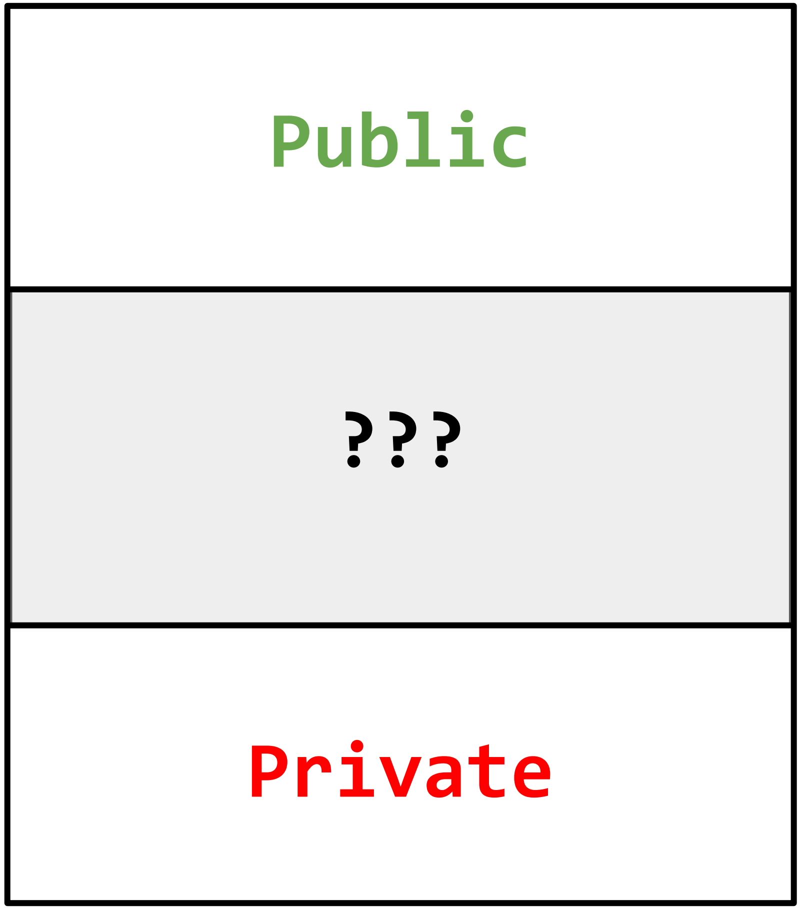
- elems : int[]
...

+ 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

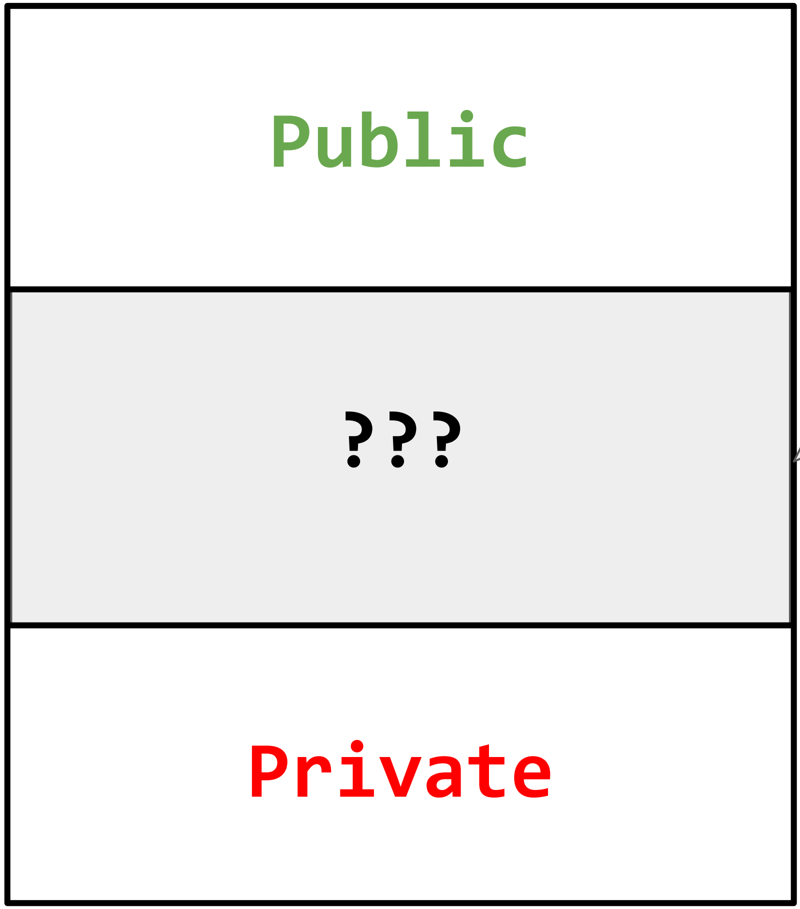


Public

???

Private

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

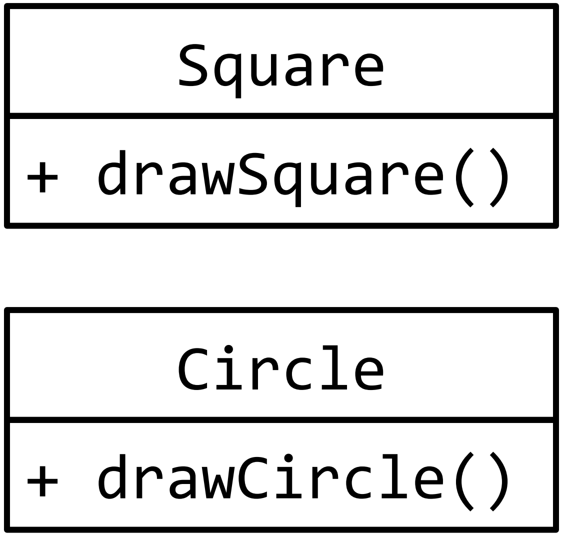
Open/closed principle

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?



Square

+ drawSquare()

Circle

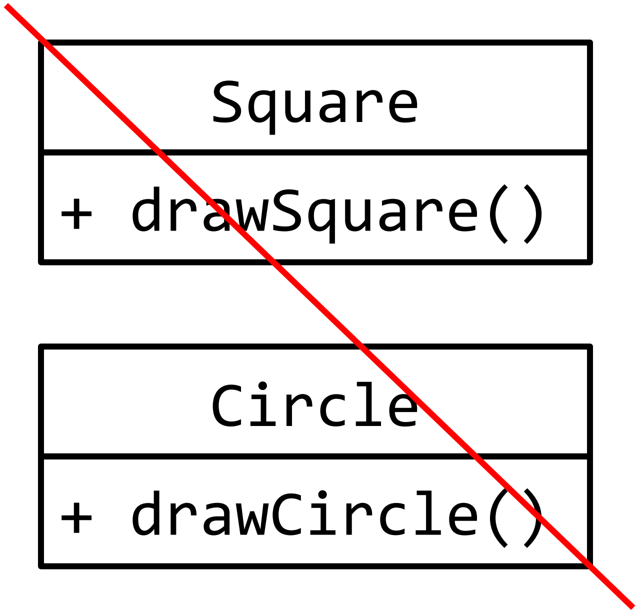
+ drawCircle()

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 {
    ...
  }
}
```



Square

+ drawSquare()

Circle

+ drawCircle()

Violates the open/closed principle!

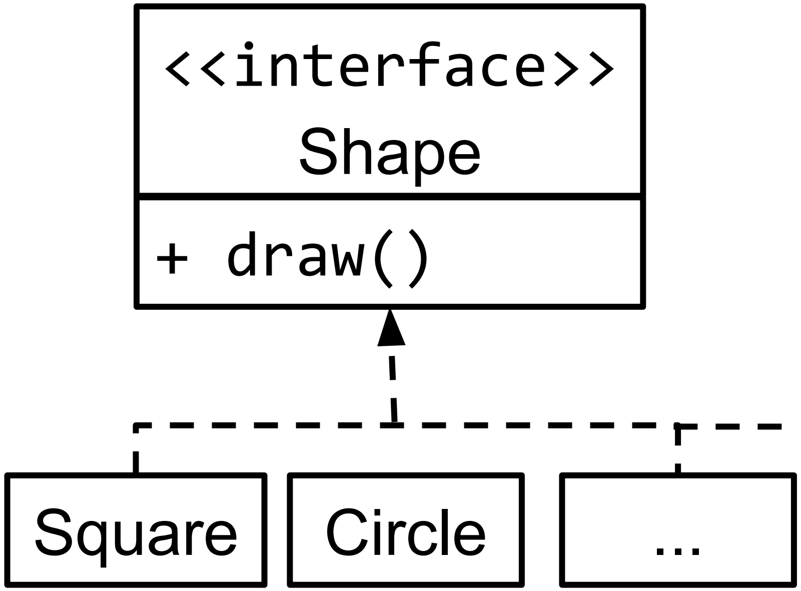
Open/closed principle

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 {
     …
  }
}
```

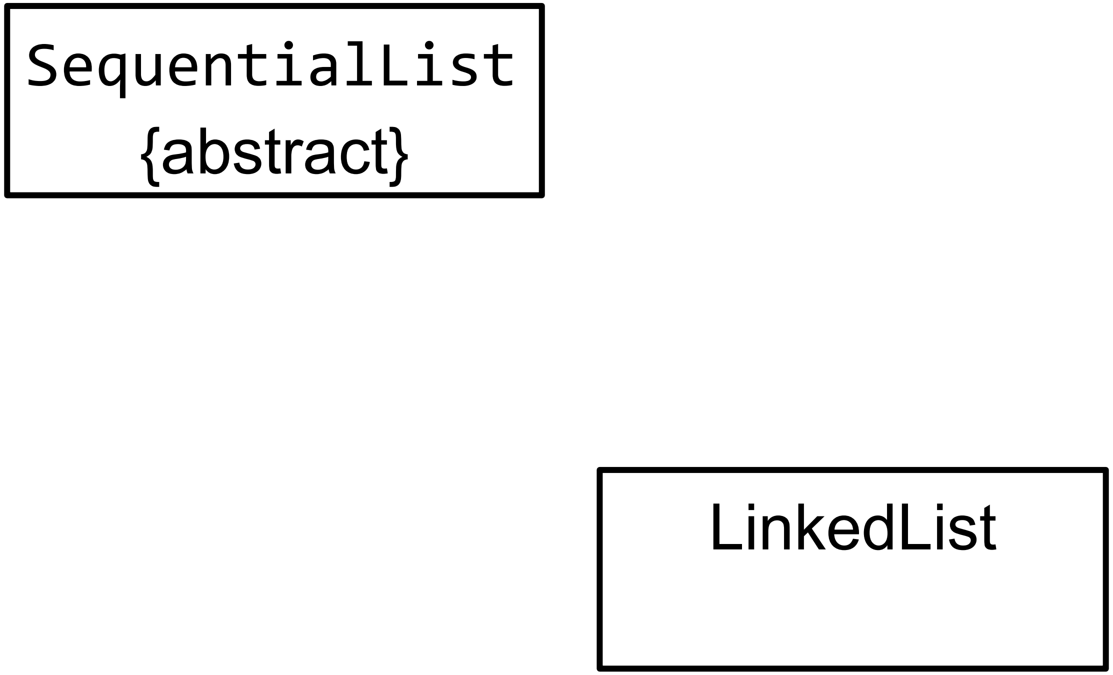
```
public static void draw(Shape s) {
  s.draw();
}
```



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



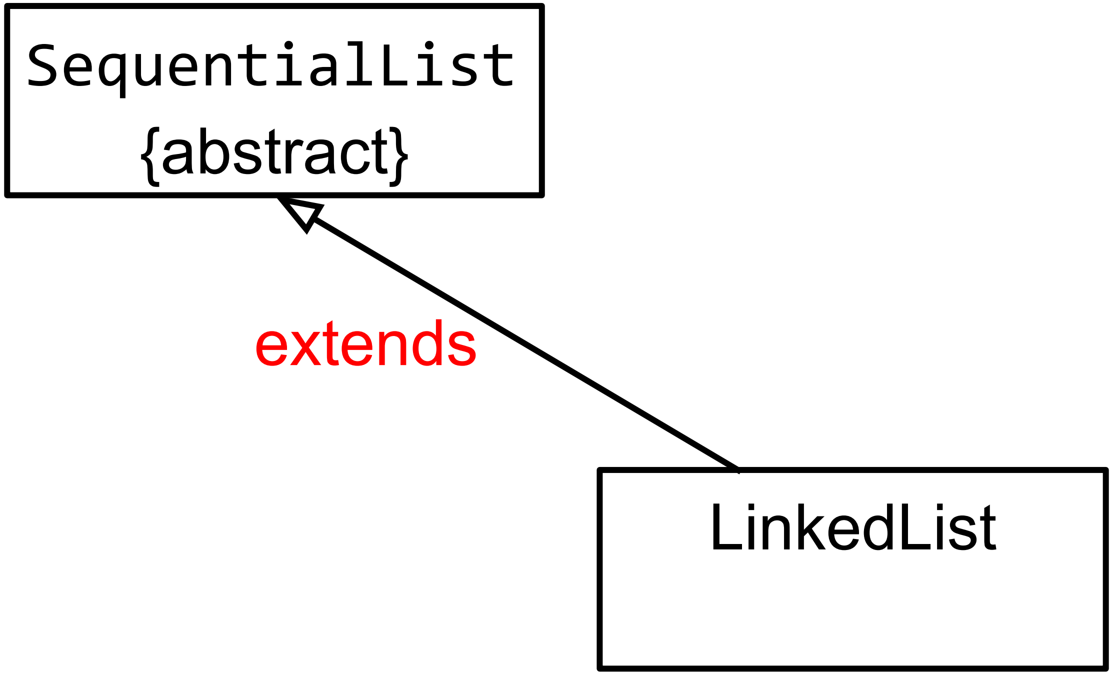
`SequentialList`

{abstract}

LinkedList

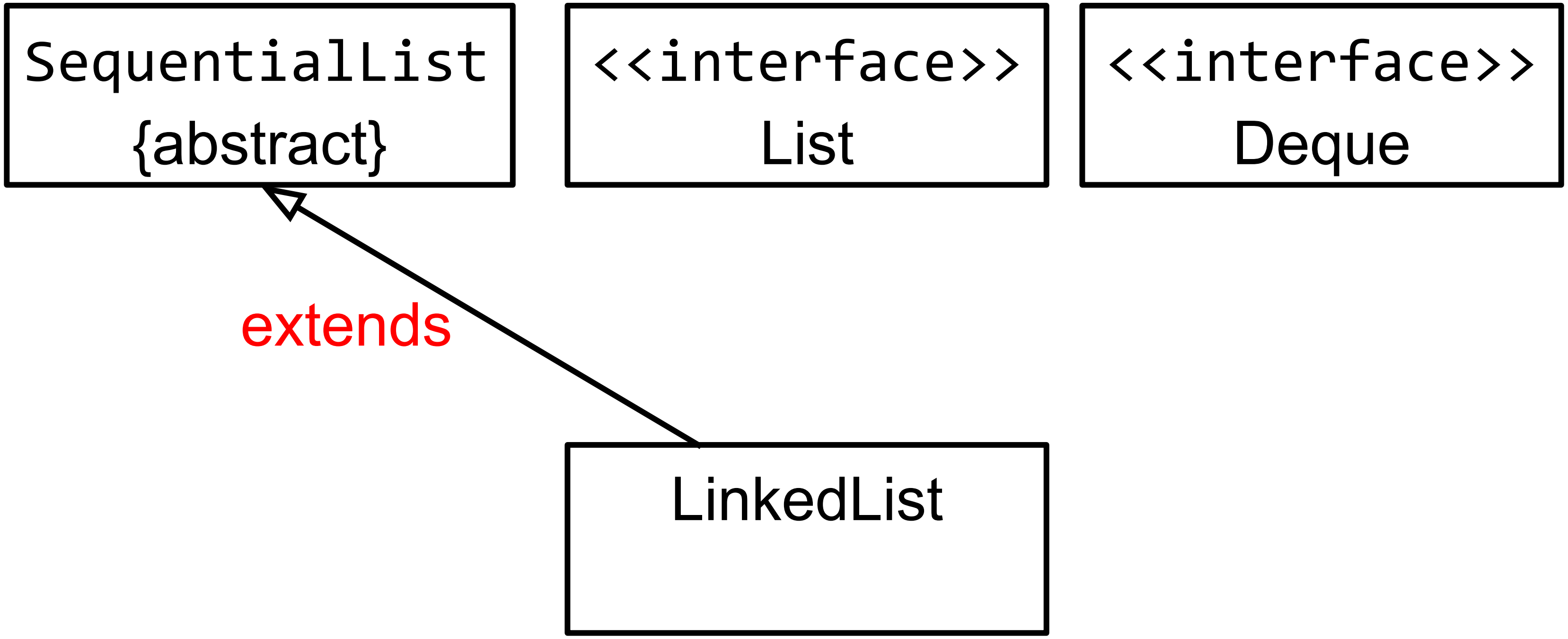
Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList



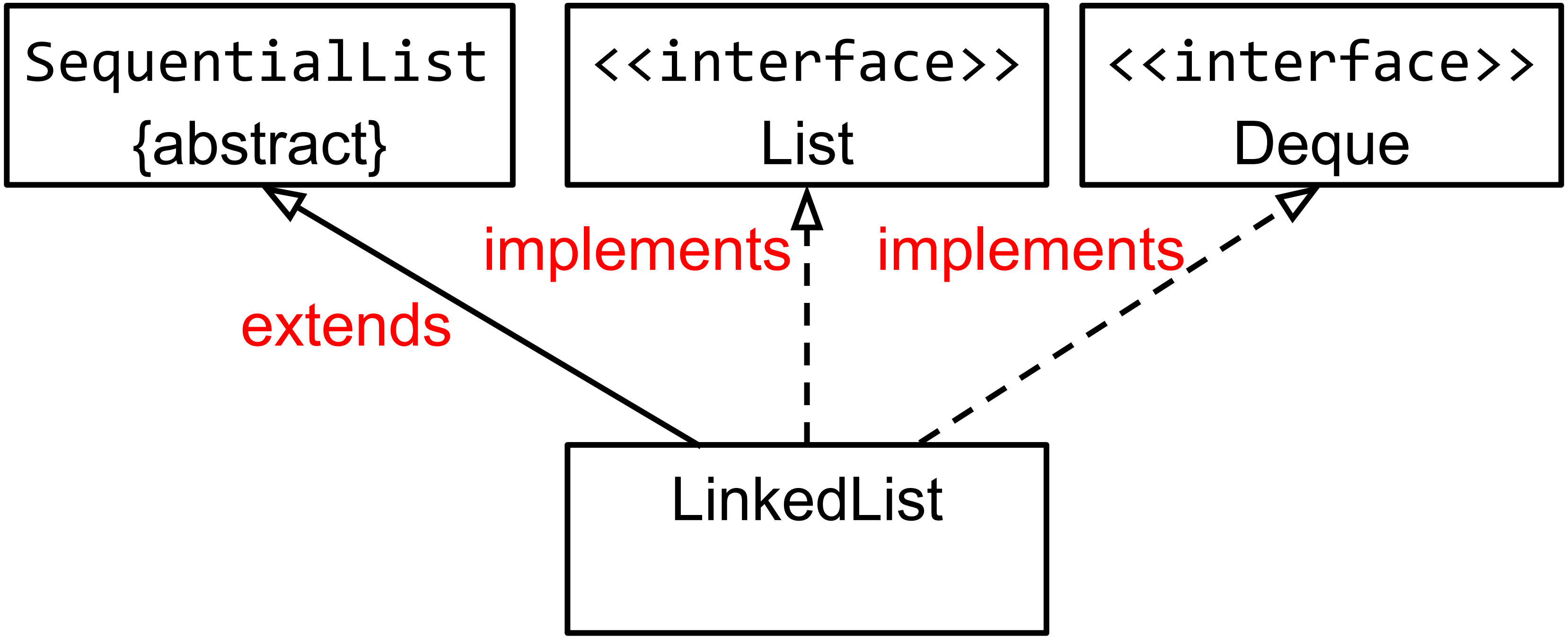
Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList

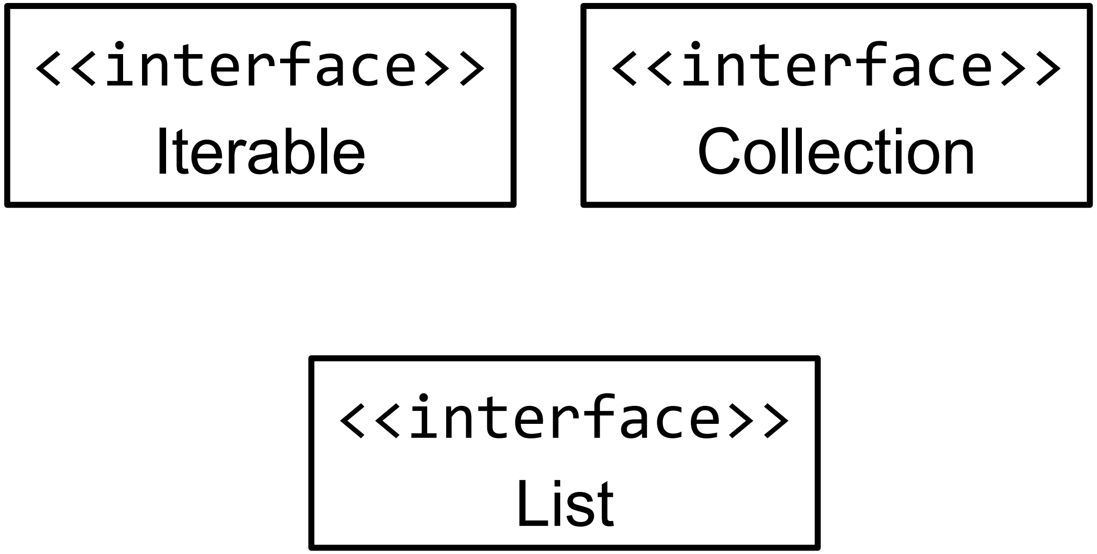


Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList implements List, Deque



Inheritance: (abstract) classes and interfaces

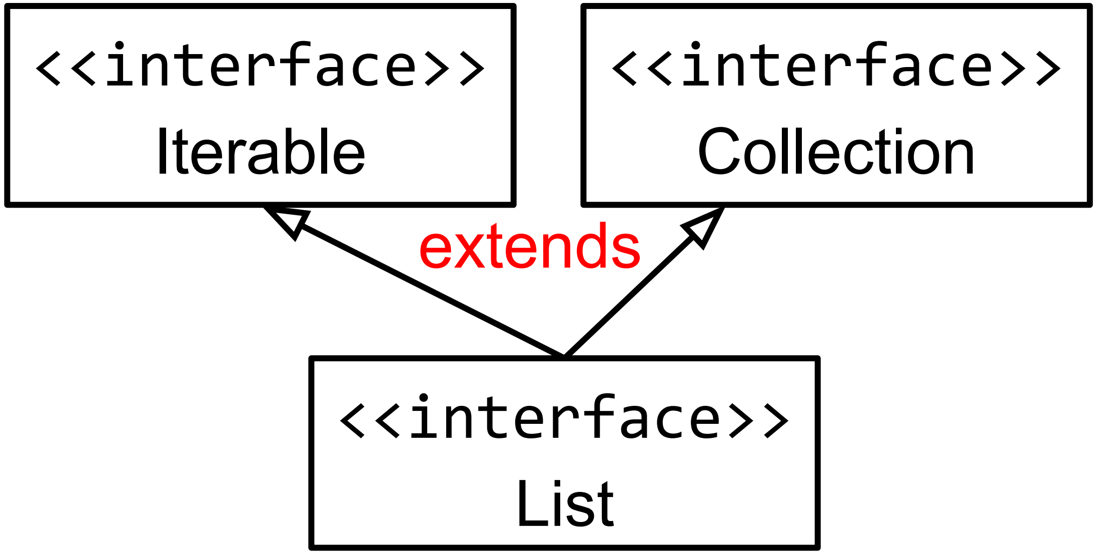


<<interface>>
Iterable

<<interface>>
Collection

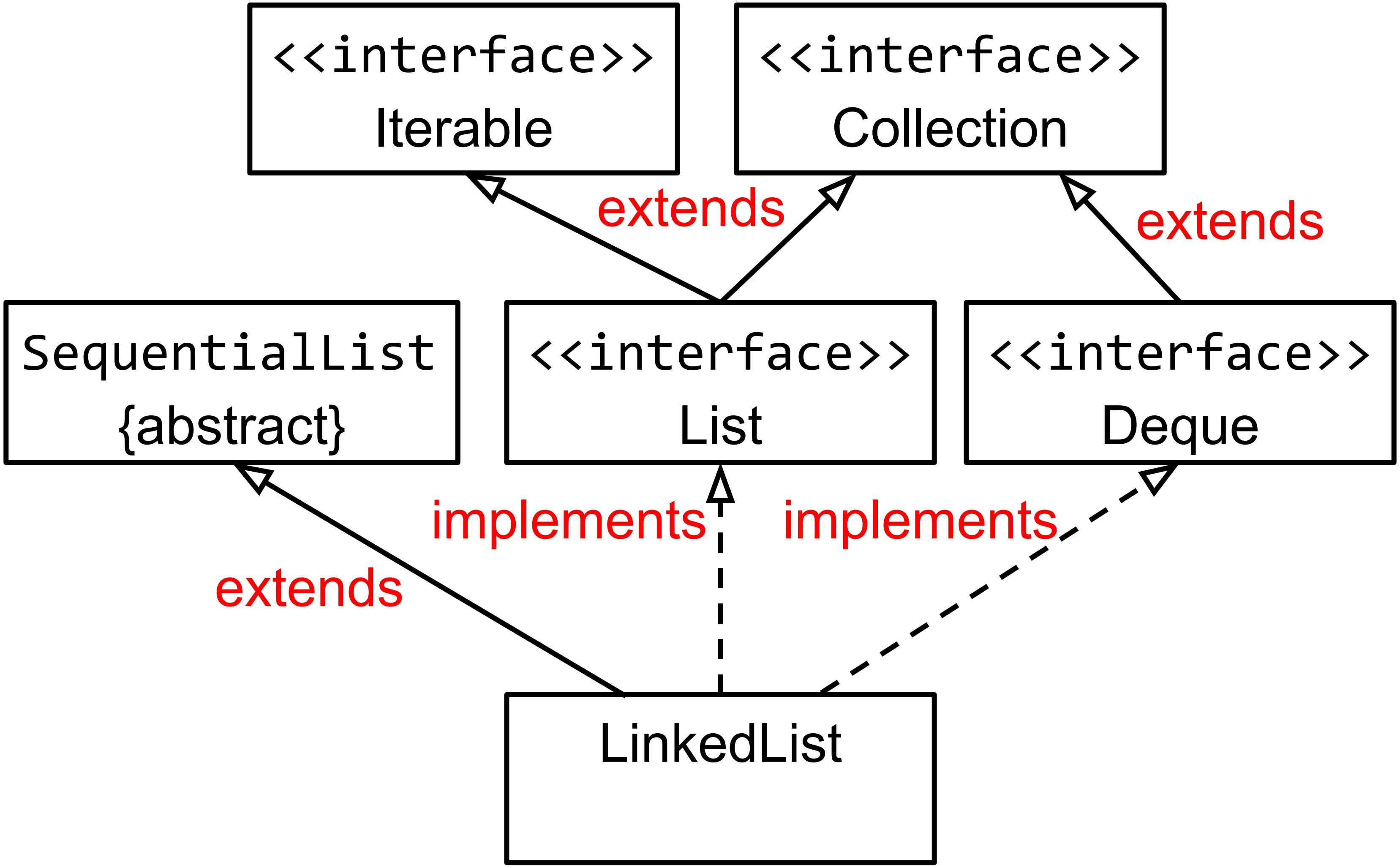
<<interface>>
List

Inheritance: (abstract) classes and interfaces



List extends Iterable, Collection

Inheritance: (abstract) classes and interfaces

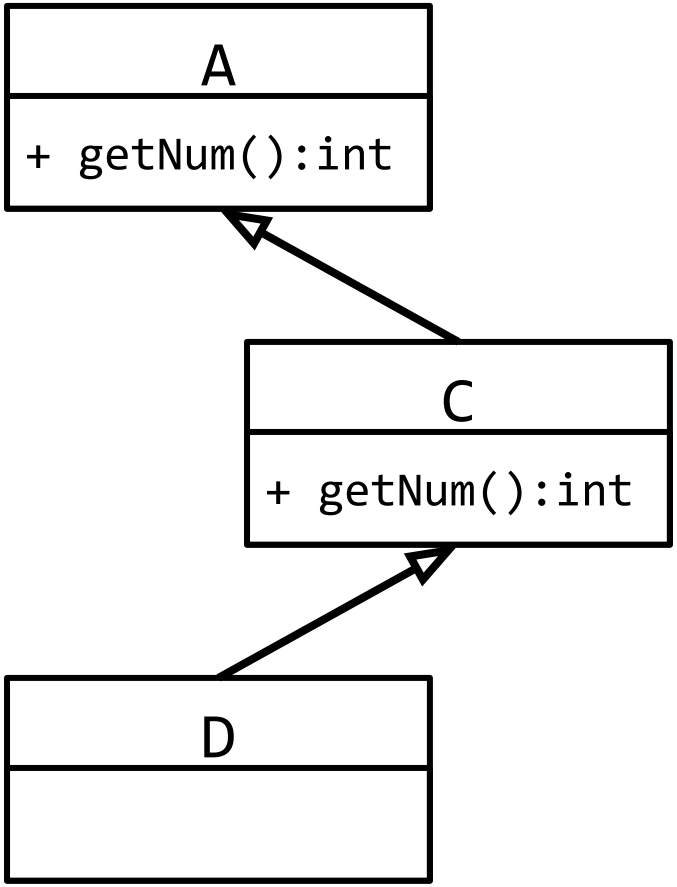


OO design principles

- 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

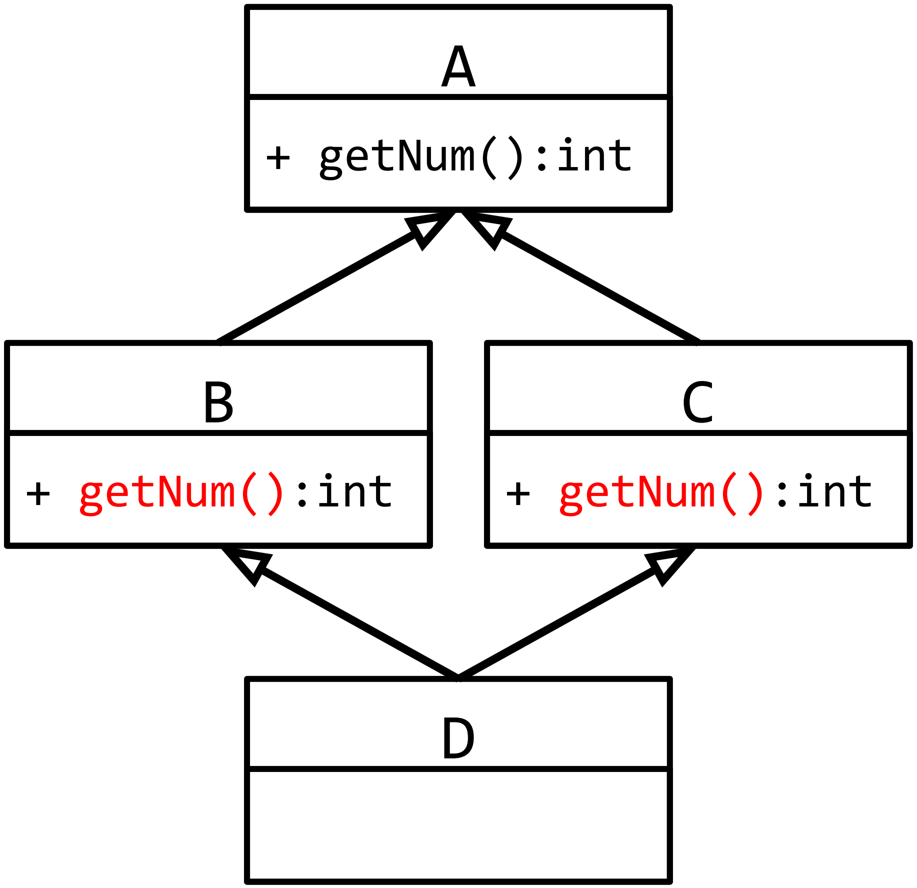
```
...
A a = new D();
int num = a.getNum();
...
```



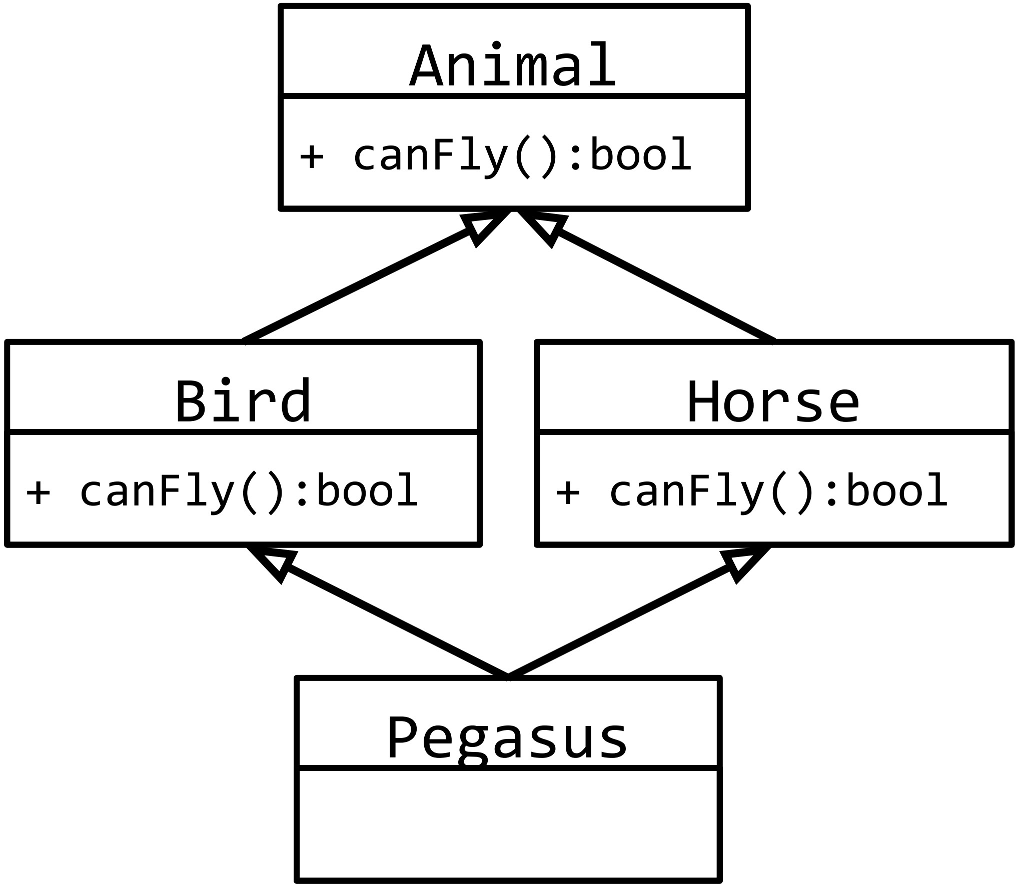
The “diamond of death”: the problem

```
...
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.

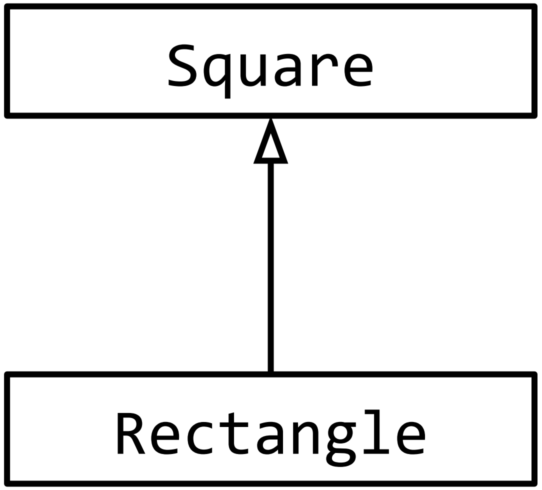
OO design principles

- 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

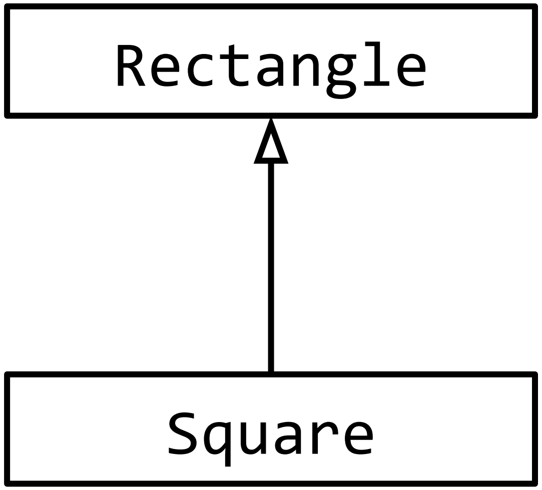
Motivating example

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



Square

Rectangle



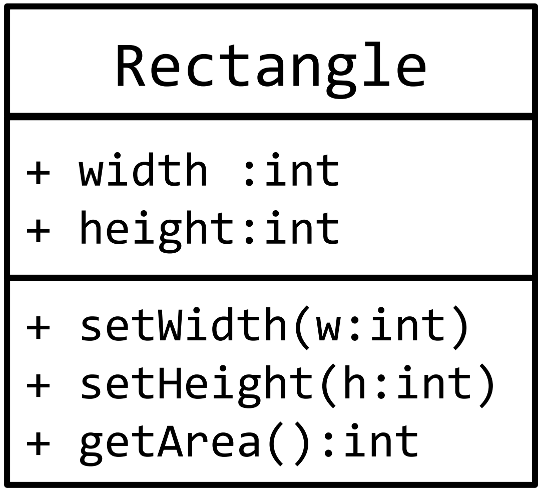
Rectangle

Square

Design principles: Liskov substitution principle

Subtype requirement

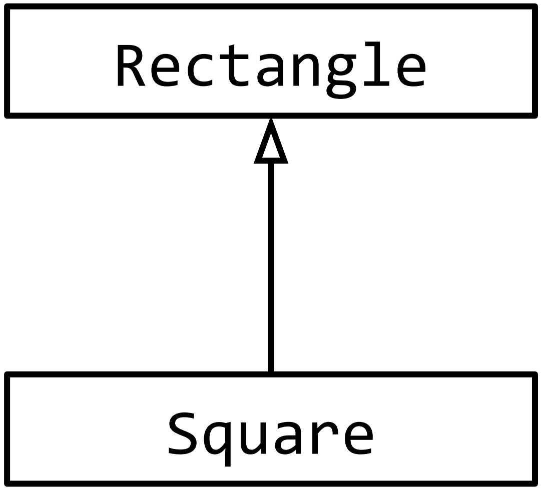
Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.



Rectangle

+ width :int
+ height:int

+ setWidth(w:int)
+ setHeight(h:int)
+ getArea():int



Rectangle

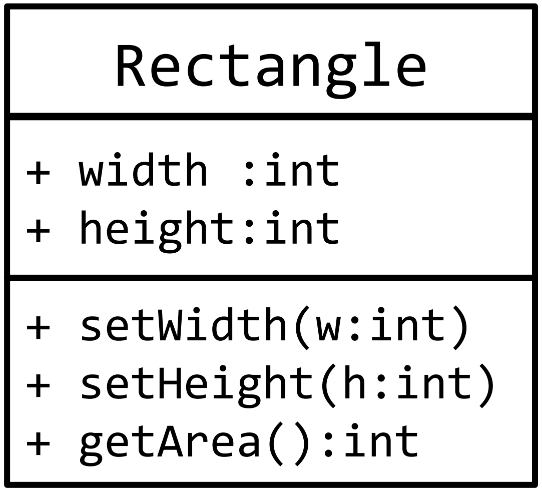
Square

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.

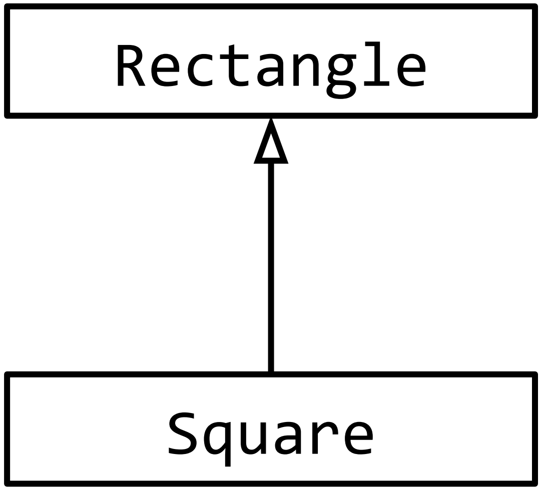


Rectangle

+ 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());
```



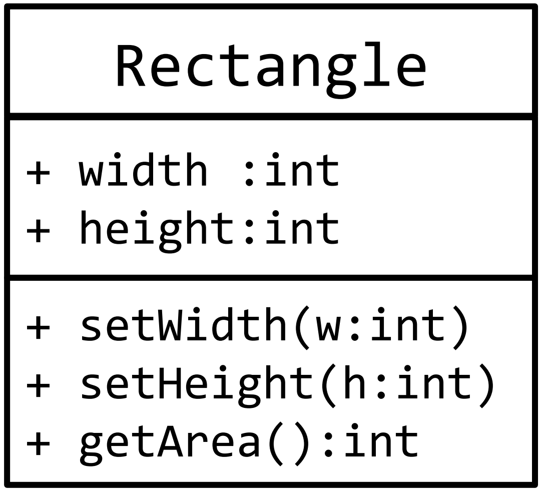
Rectangle

Square

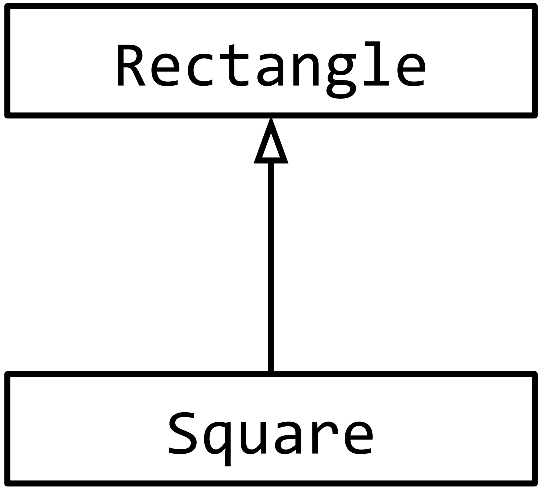
Design principles: Liskov substitution principle

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.



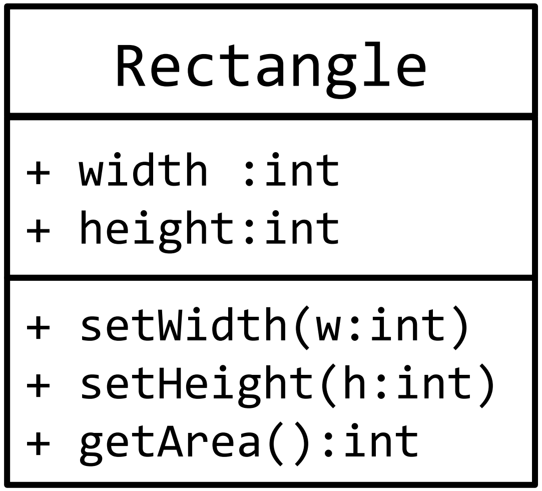
```
Rectangle r =
  ~~new Rectangle(2,2);~~
  new Square(2);

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

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



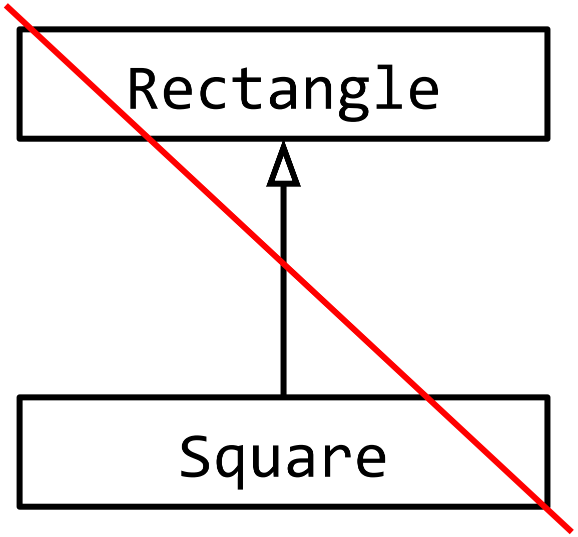
Design principles: Liskov substitution principle

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());
```

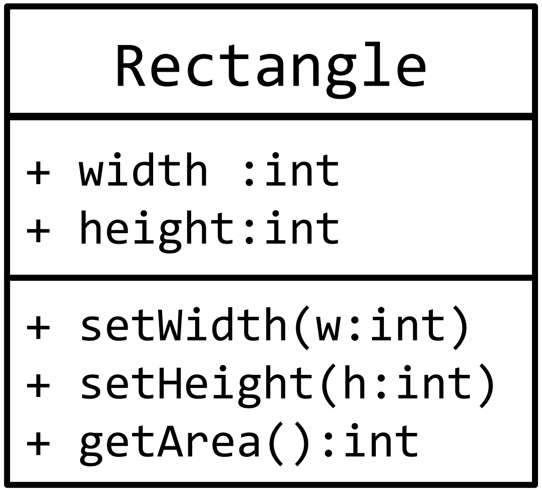


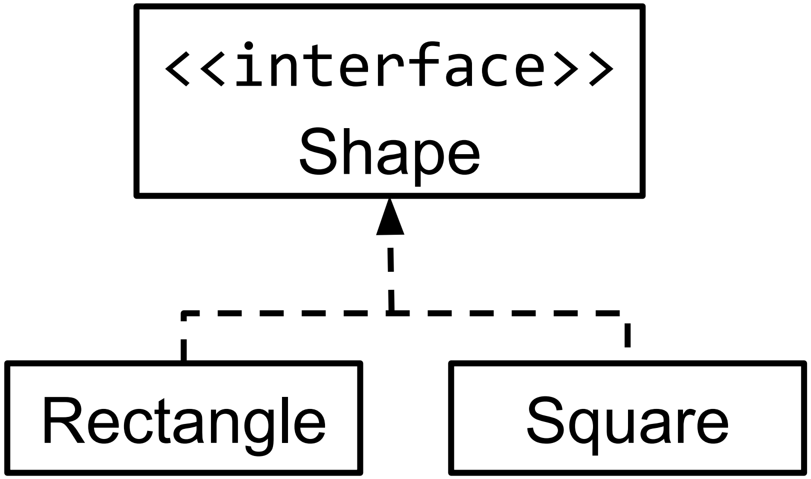
Violates the Liskov substitution principle!

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.

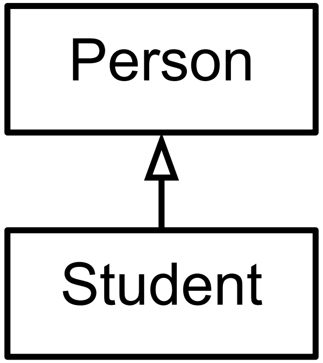




OO design principles

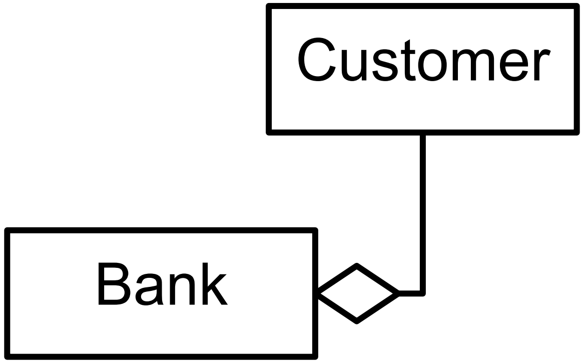
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- **Composition/aggregation over inheritance**

Inheritance vs. (Aggregation vs. *Composition*)

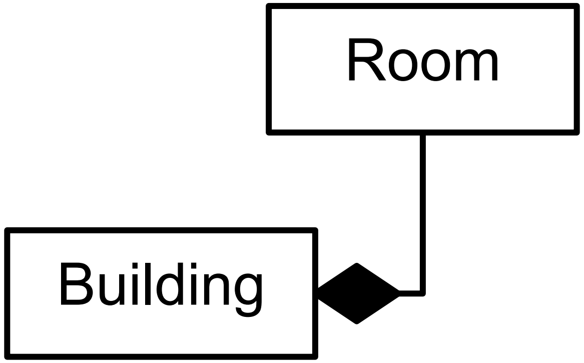


```
public class Student
    extends Person{

 public Student(){
 }


 ...
}
```

is-a relationship



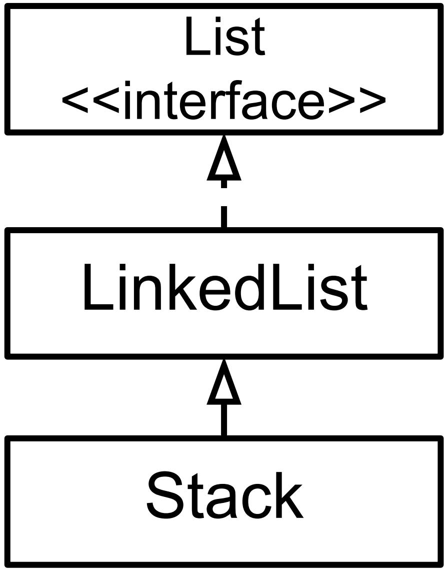
```
public class Bank {
 Customer c;

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

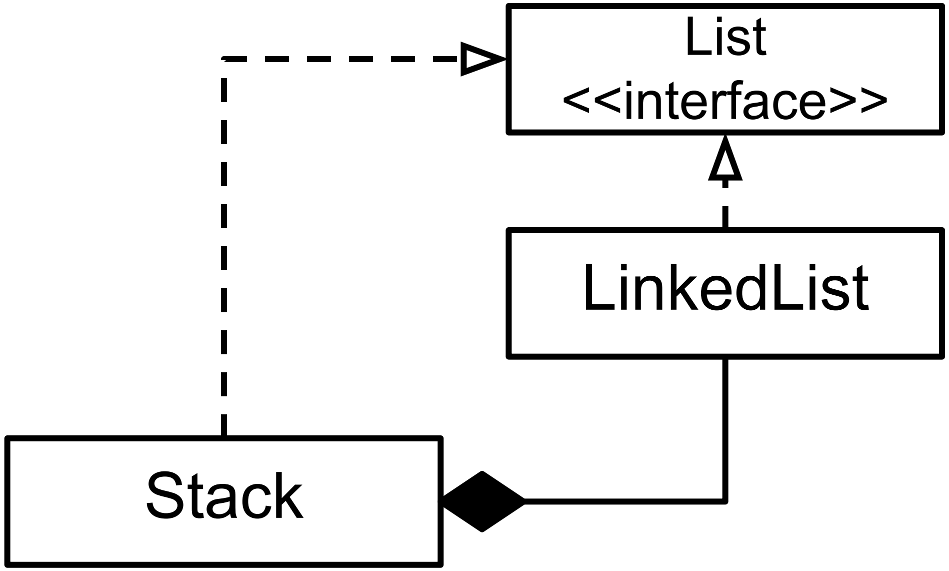


```
public class Building {
 Room r;

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

has-a relationship

Design choice: inheritance or composition?



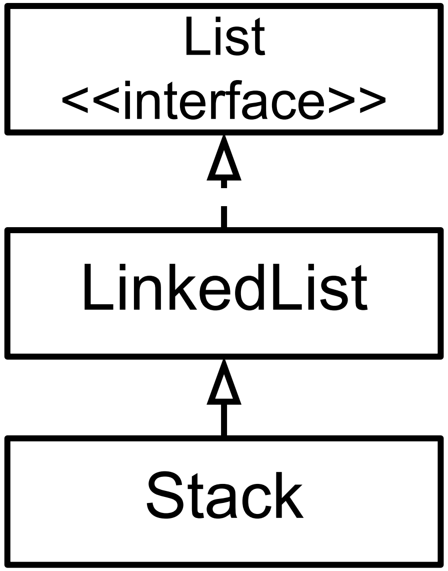


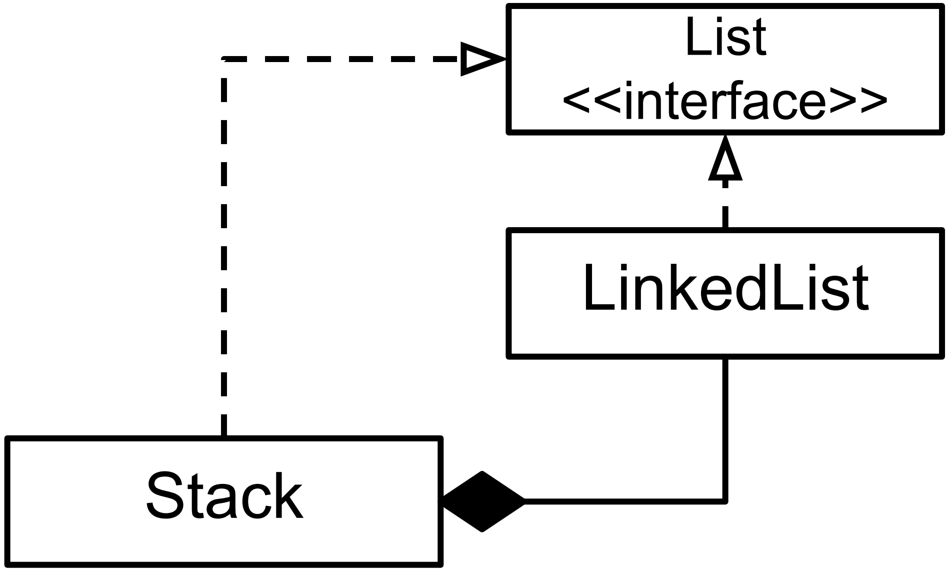
```
public class Stack<E>
     extends LinkedList<E> {
  ...
}
```

```
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.

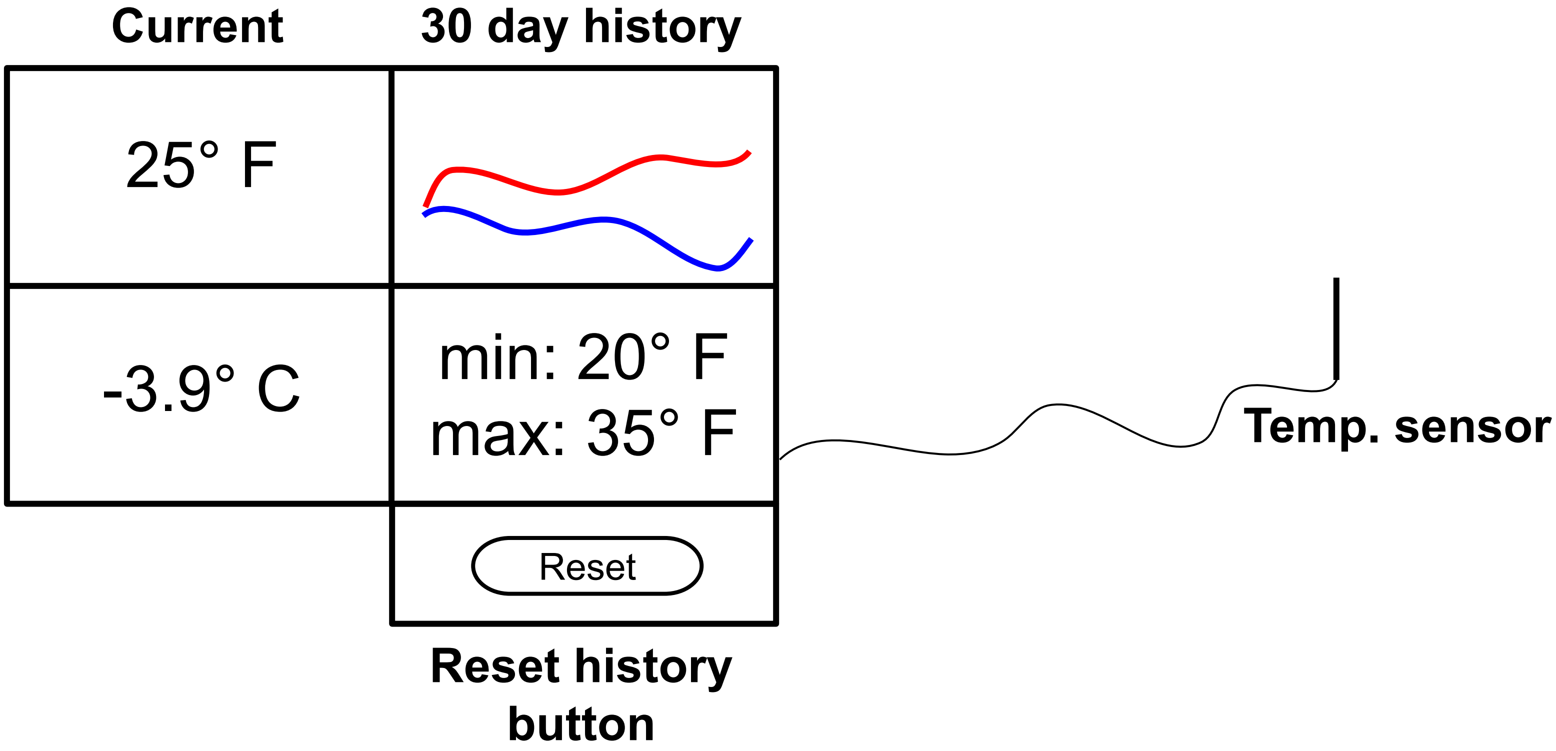
OO design principles: summary

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

OO design patterns

A first design problem

Weather station revisited



Current

30 day history

25° F

-3.9° C

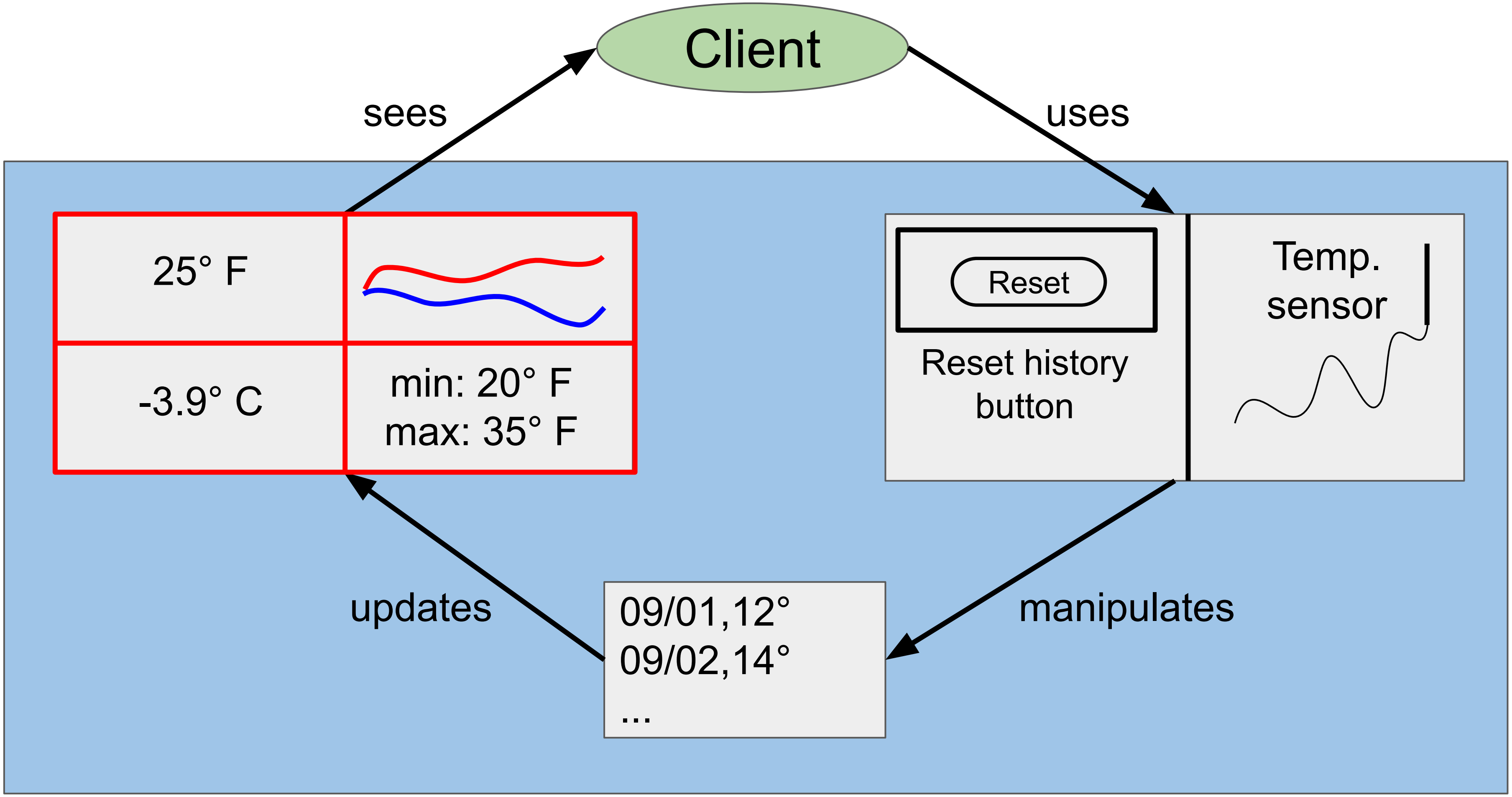
min: 20° F
max: 35° F

Reset

Reset history button

Temp. sensor

What’s a good design for the view component?



Client

sees

uses

25° F

-3.9° C

min: 20° F
max: 35° F

Reset

Reset history button

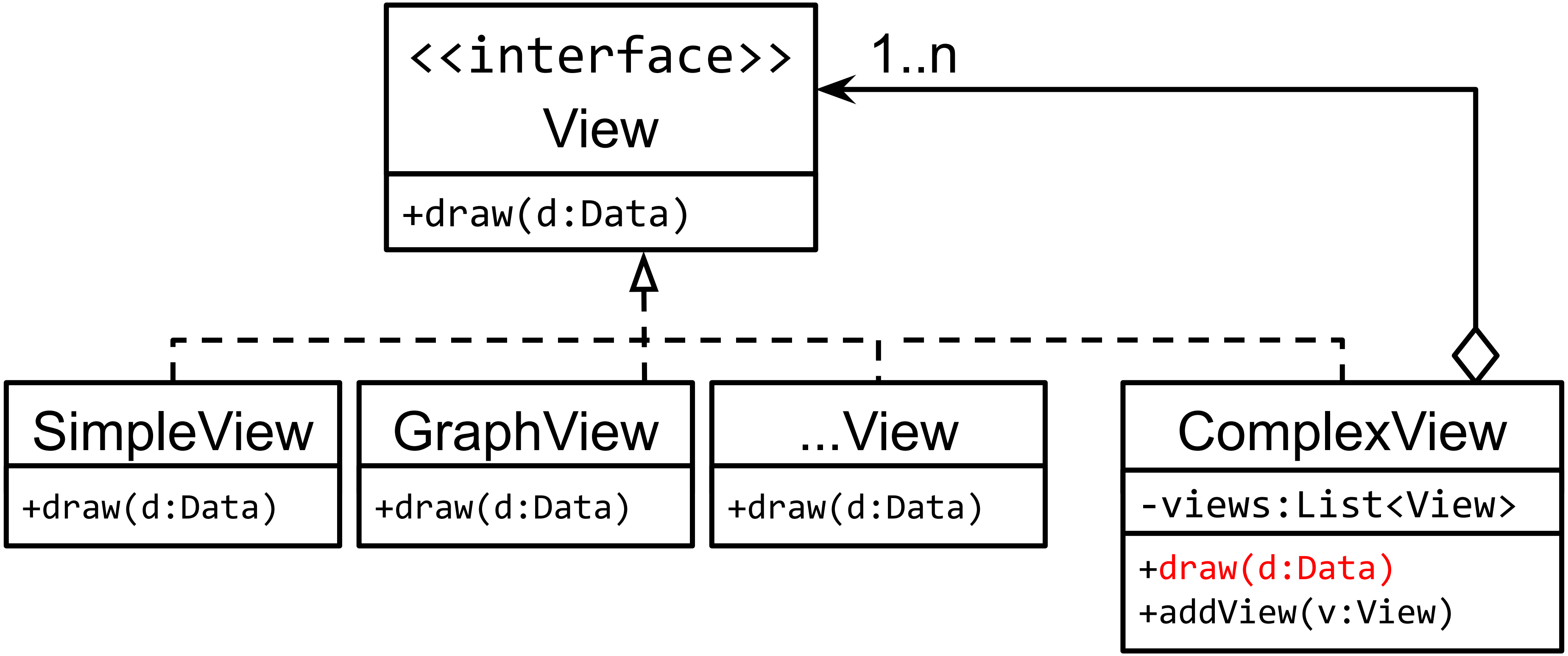
Temp. sensor

updates

manipulates

09/01,12°
09/02,14°
...

Weather station: view



<<interface>>
View

+draw(d:Data)

1..n

SimpleView

+draw(d:Data)

GraphView

+draw(d:Data)

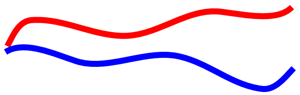
...View

+draw(d:Data)

ComplexView

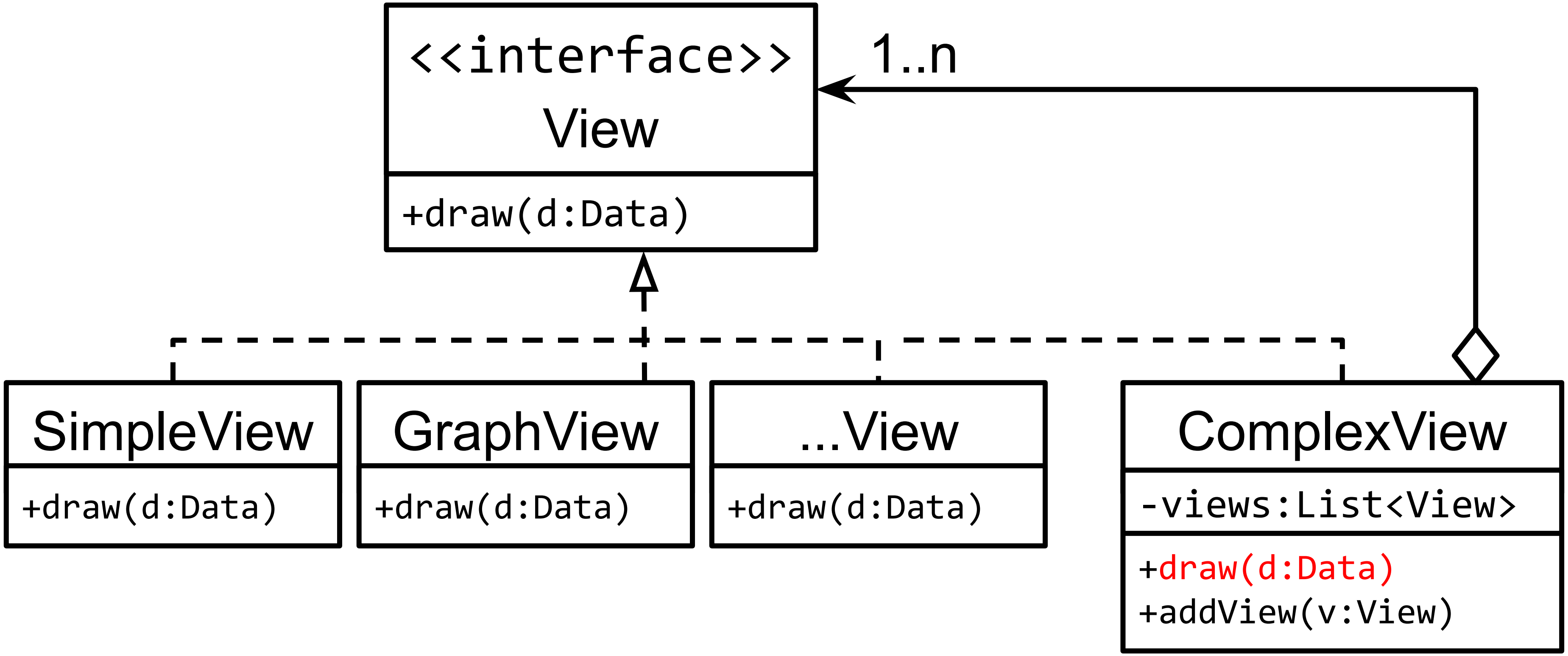
-views:List<View>

+draw(d:Data)
+addView(v:View)

25° F	
-3.9° C	min: 20° F max: 35° F

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

Weather station: view



<<interface>>
View
+draw(d:Data)

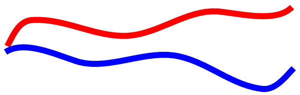
1..n

SimpleView
+draw(d:Data)

GraphView
+draw(d:Data)

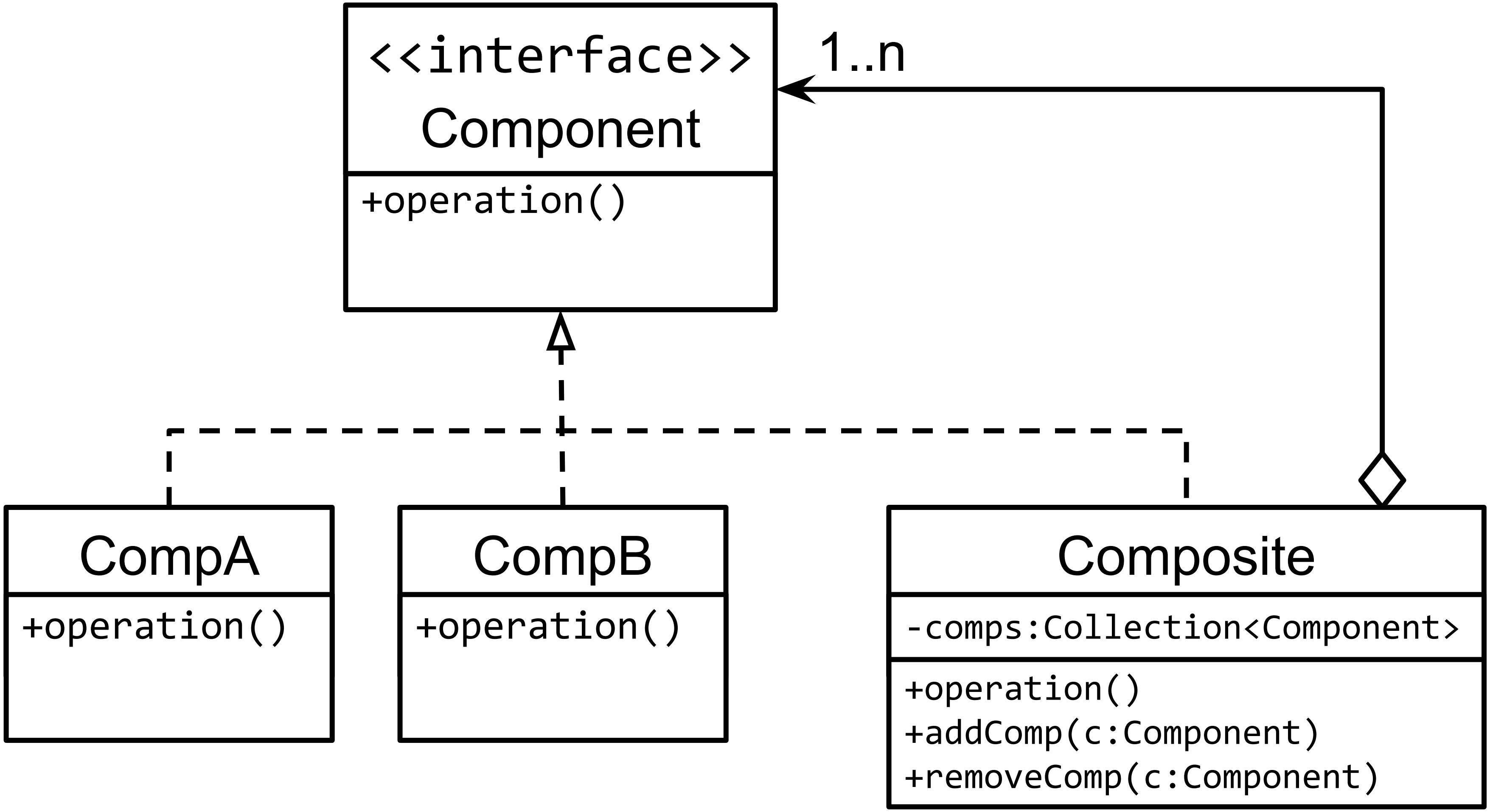
...View
+draw(d:Data)

ComplexView
-views:List<View>
+draw(d:Data)
+addView(v:View)

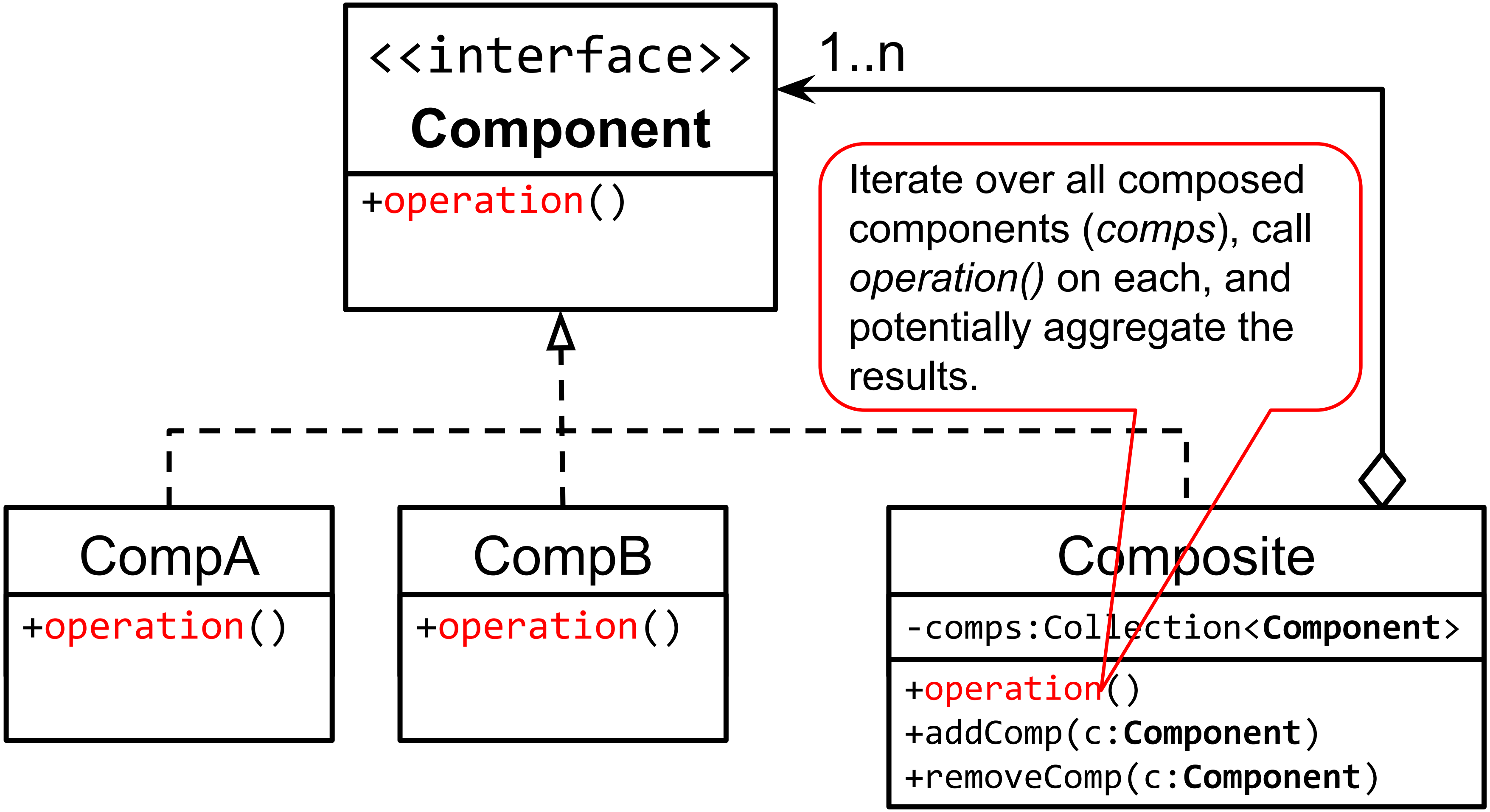
25° F	
-3.9° C	min: 20° F max: 35° F

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

The general solution: Composite pattern



The general solution: Composite pattern



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

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.

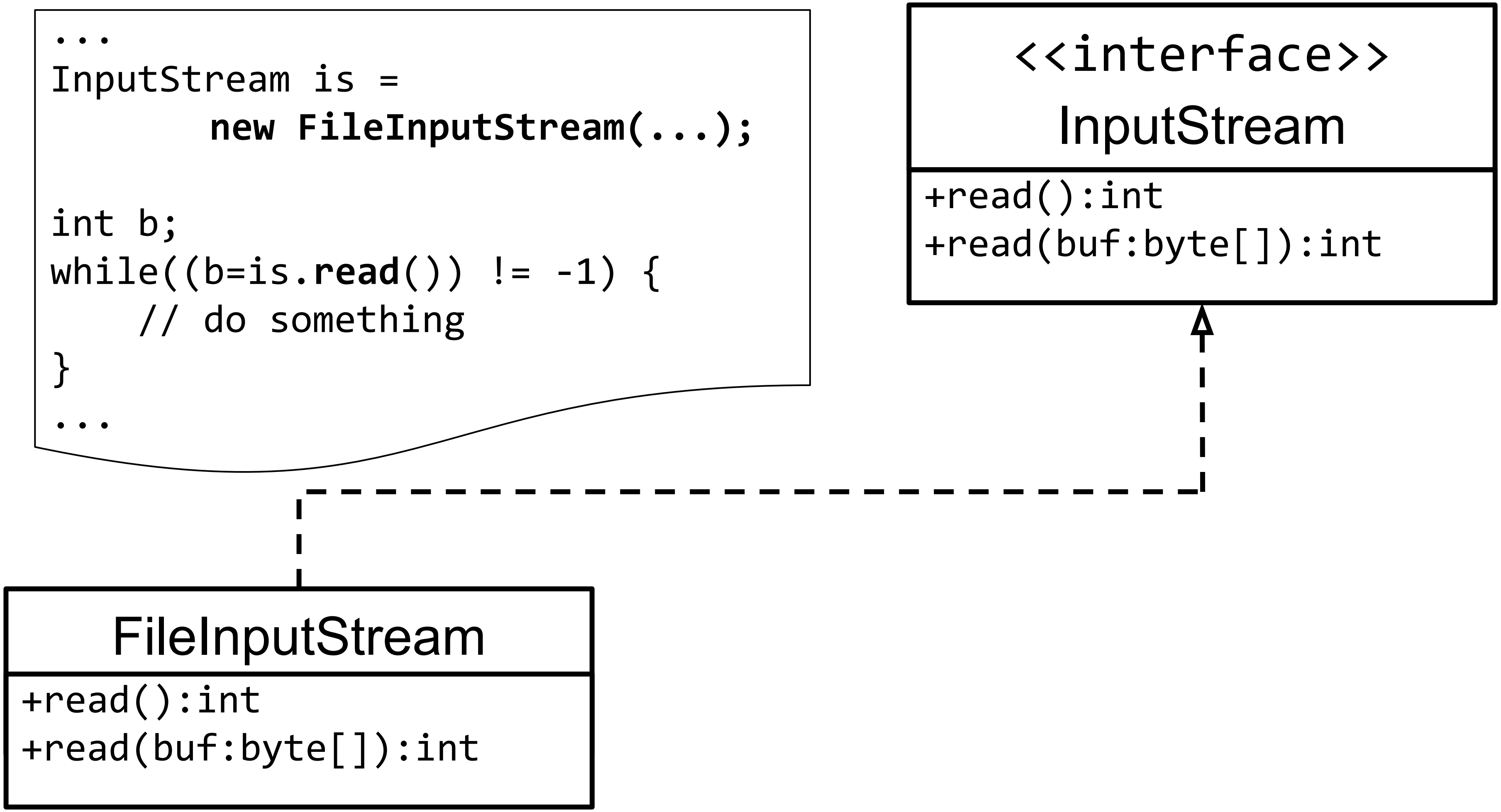
Design patterns: categories

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
}
...
```

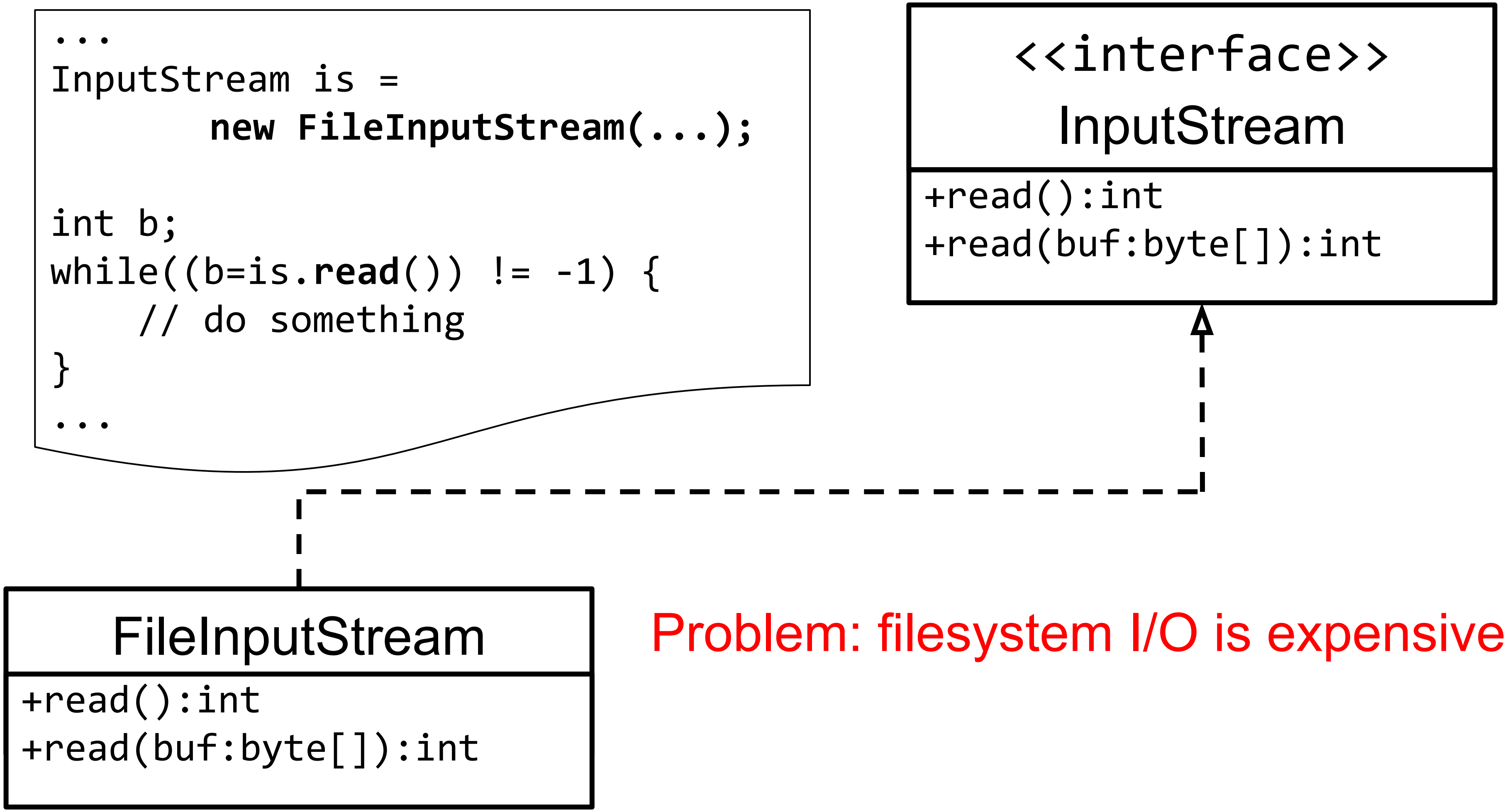
<<interface>>
InputStream

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

FileInputStream

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

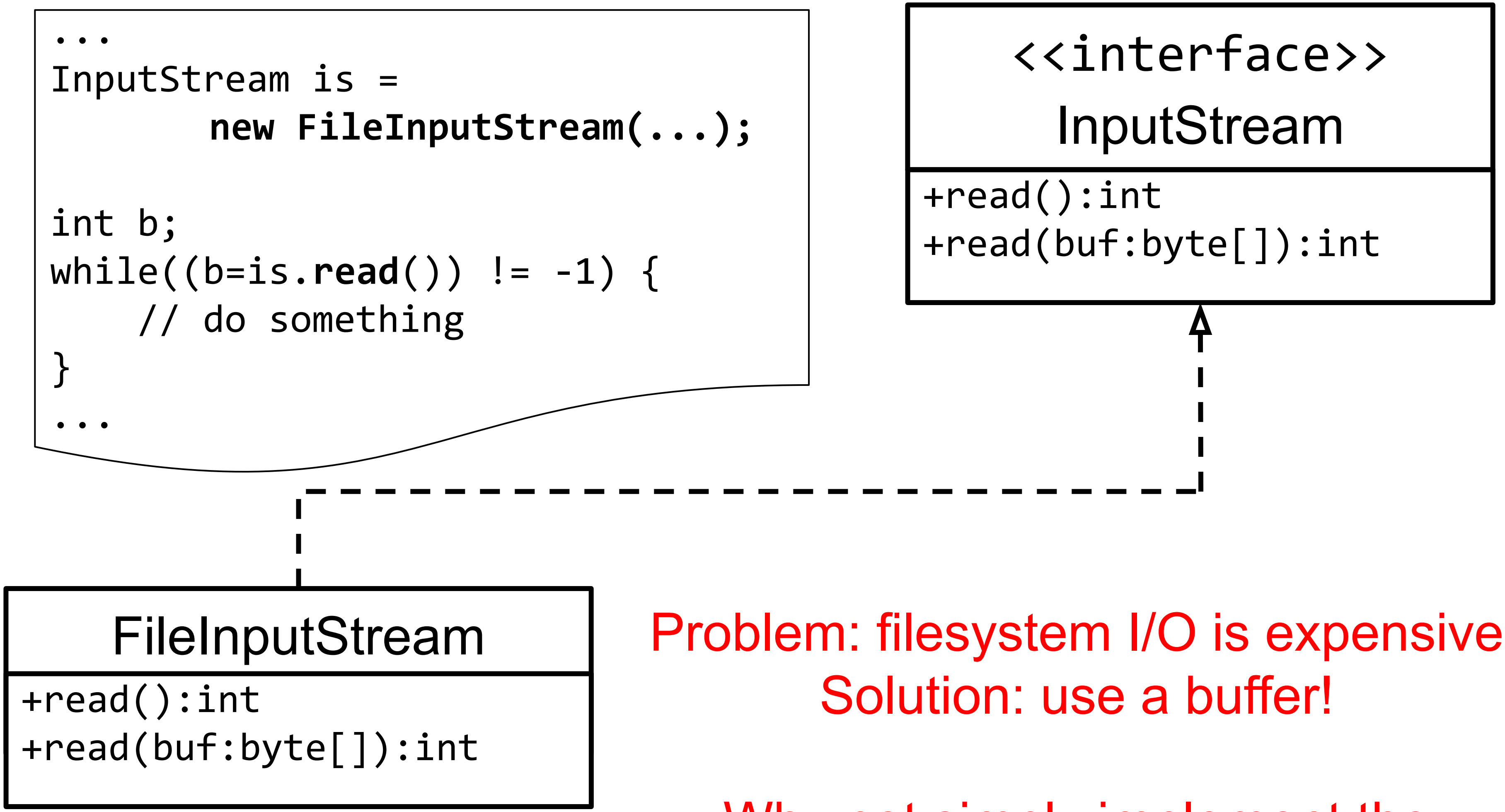
Another design problem: I/O streams



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

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

Problem: filesystem I/O is expensive

Another design problem: I/O streams



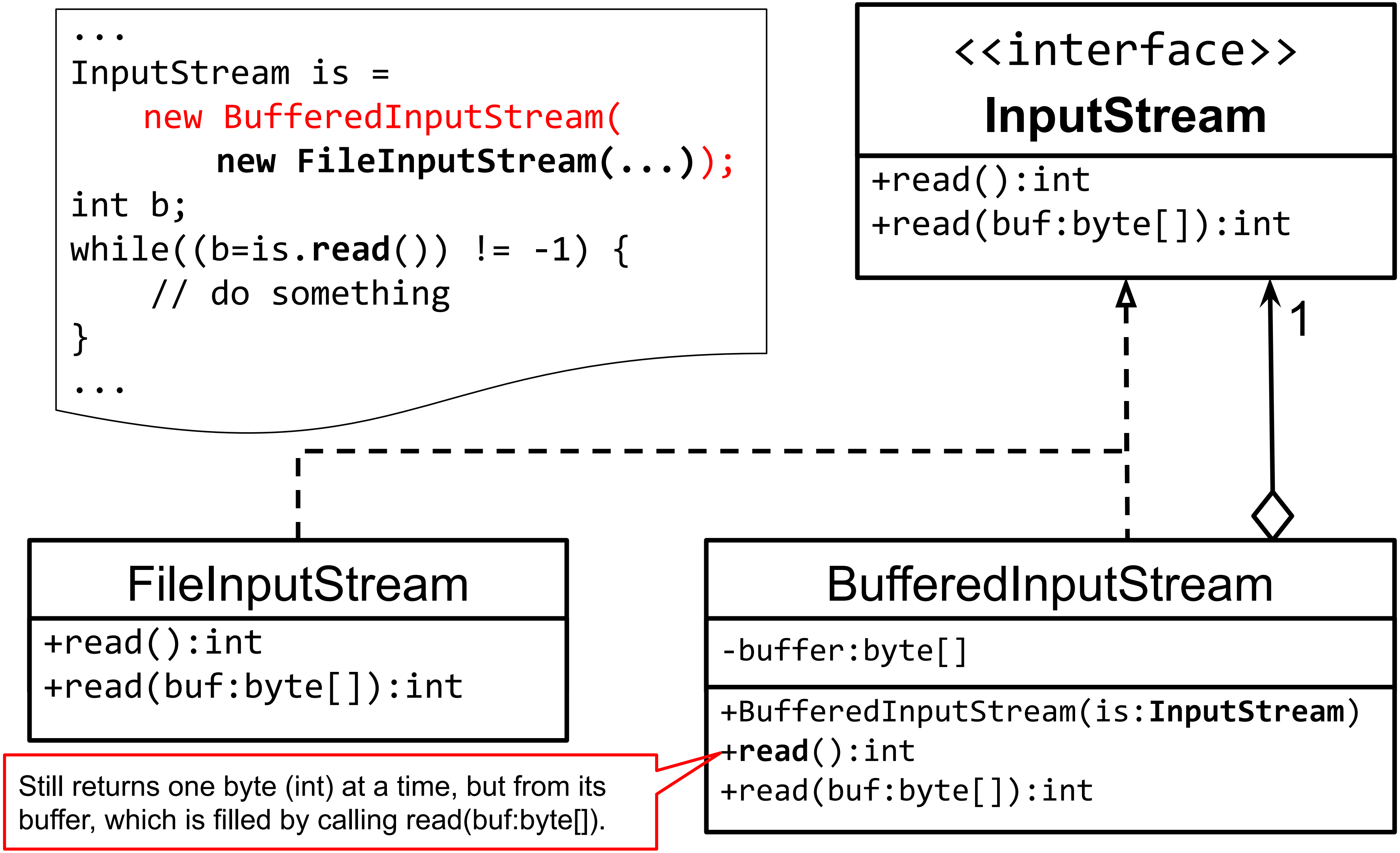
```
...
InputStream is =
       new FileInputStream(...);

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?

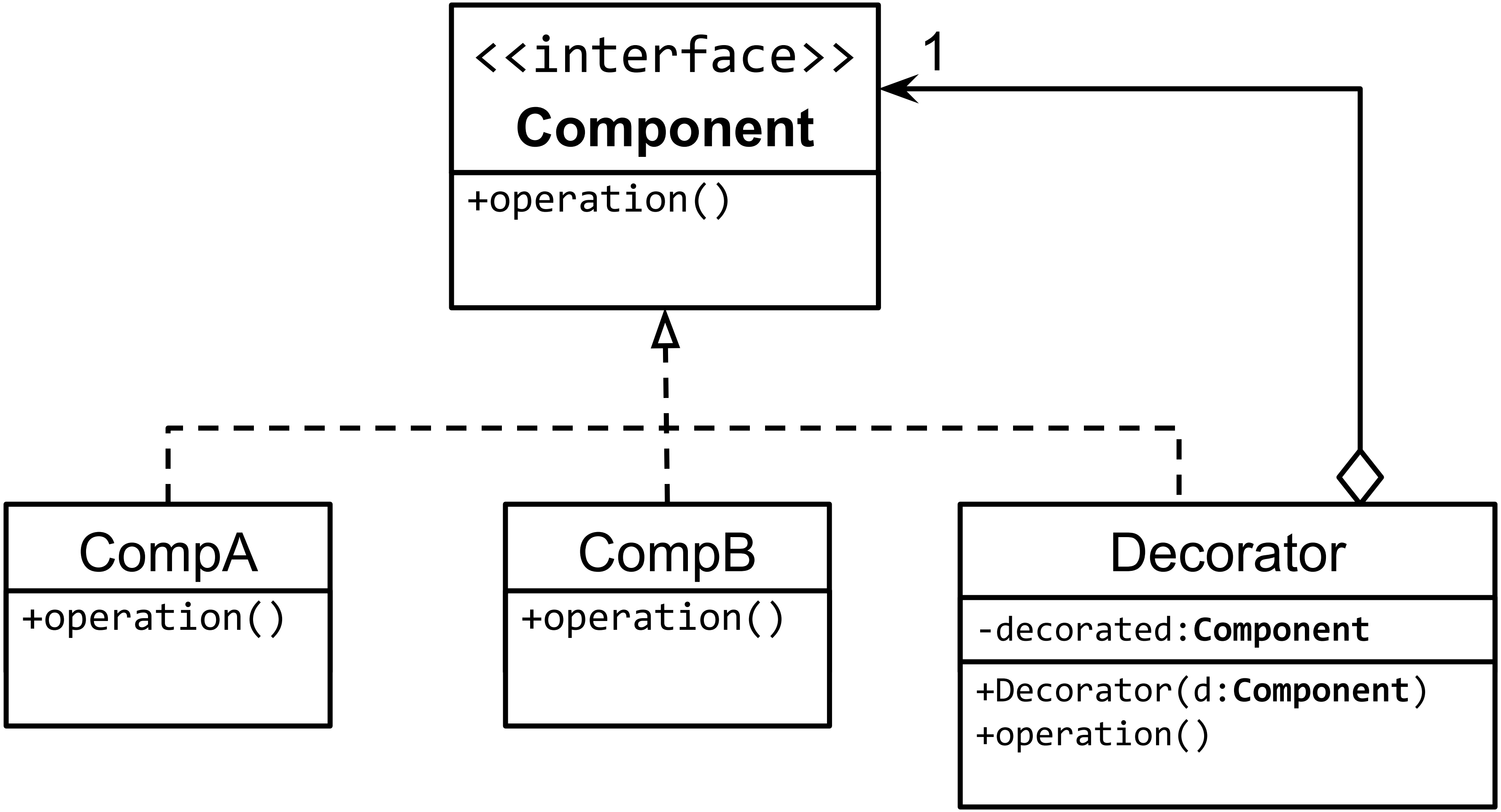
Another design problem: I/O streams

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

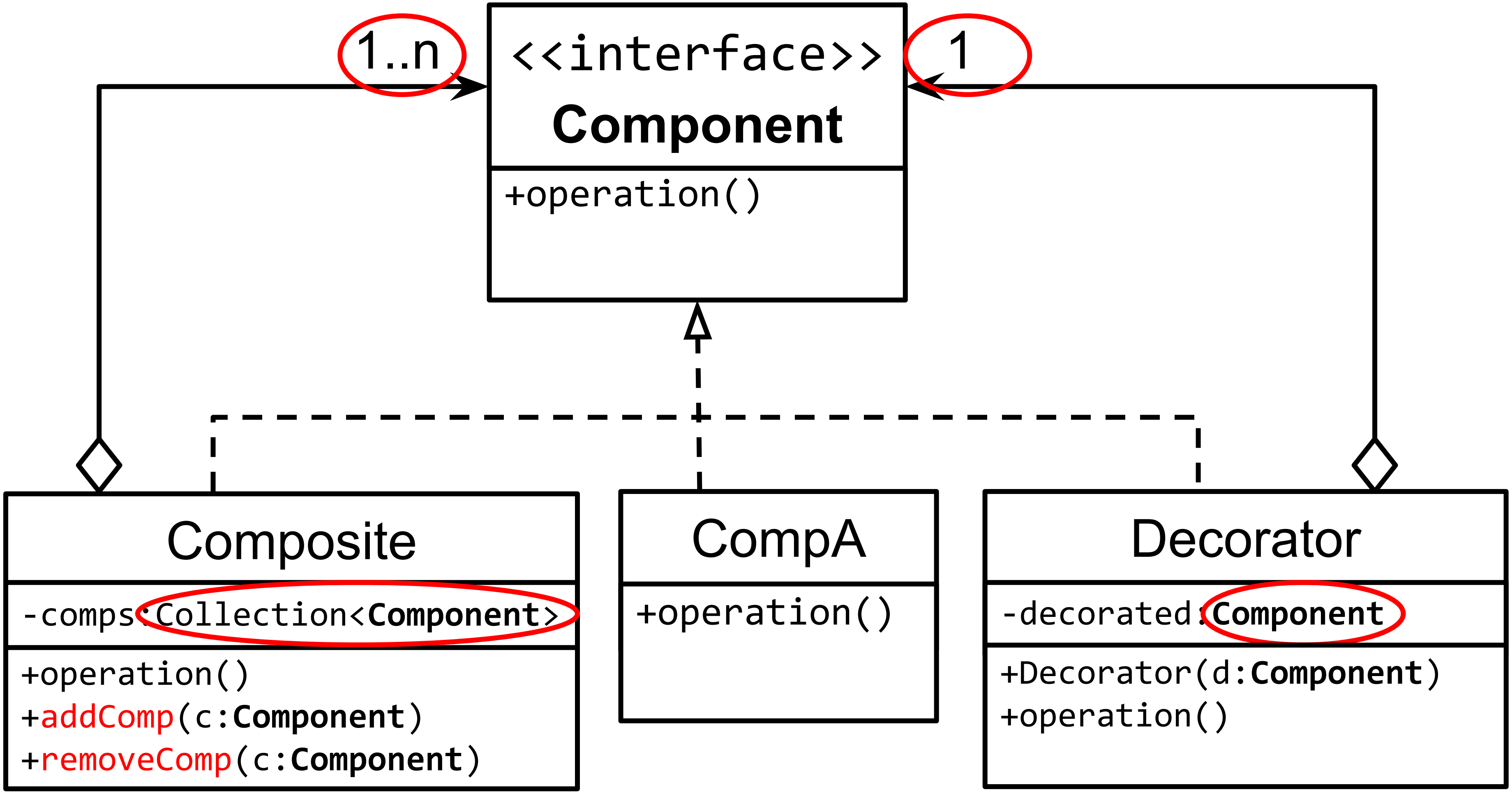


Still returns one byte (int) at a time, but from its buffer, which is filled by calling read(buf:byte[]).

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()