Software Design and Style

CSE 403 Software Engineering

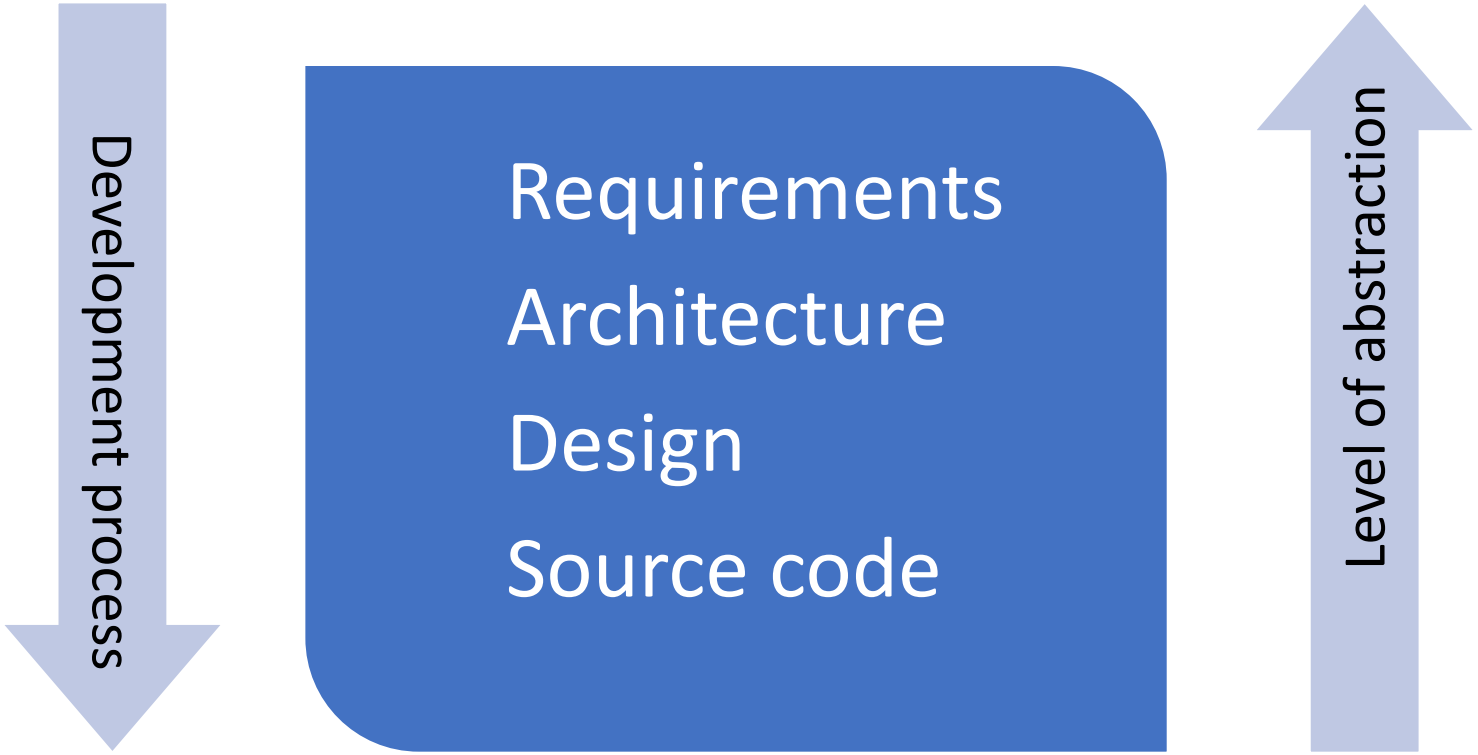
Today's Outline

1. Quick recap – Architecture vs Design
2. Some practical design considerations
3. Class quiz on coding style

See slides at end for a short primer on CSE 331 design material:

- UML (unified modeling language)
- Object oriented design principles
- Design patterns

High level overview from last class

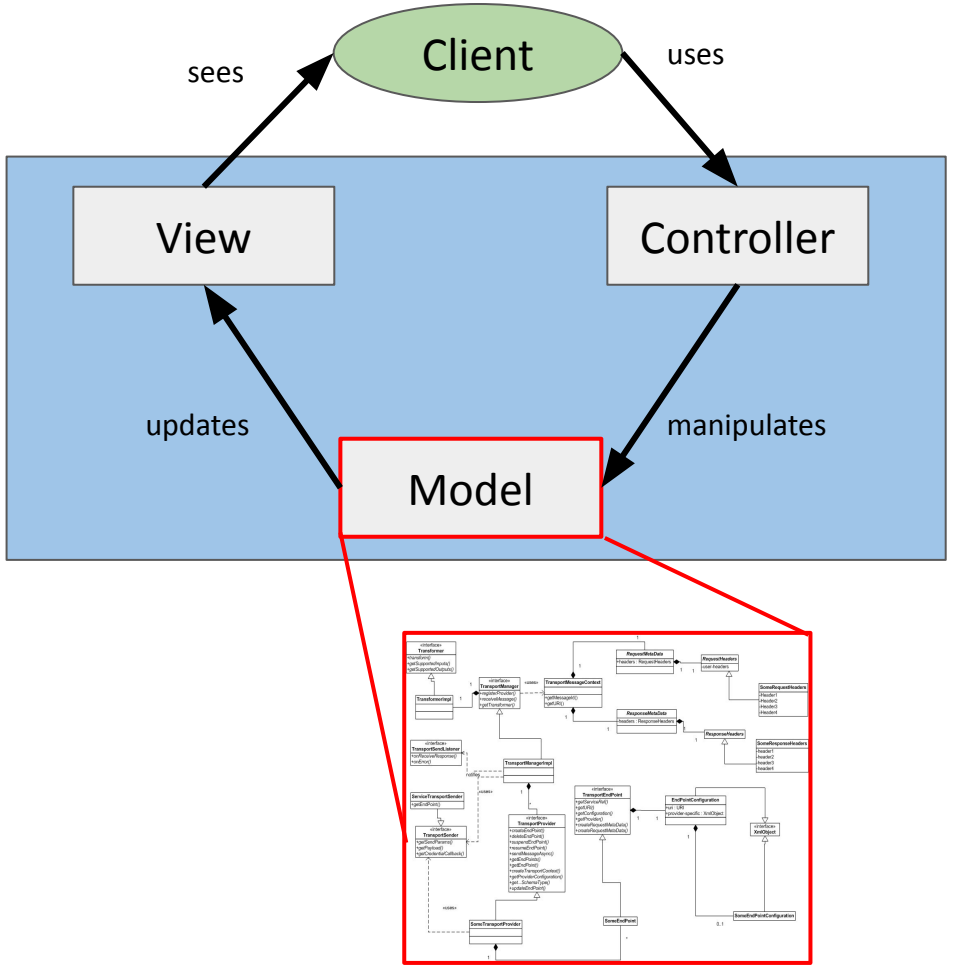


The level of abstraction is key

- With both architecture and design, we're building an abstract representation of reality

- Architecture – what components are needed, and what are their connections

- Design – how the components are developed



Some tried-and-true design principles

- KISS principle (keep it simple, stupid)
- YAGNI principle (you ain't gonna need it)
- DRY principle (don't repeat yourself; use abstractions, inheritance)
- Single responsibility (focus on on doing one thing well – high cohesion)
- Open/closed principle (open for extension, closed for modification)
- Behavioral substitution principle (aka behavioral subtyping, Liskov substitution principle) (stronger specification; user of base class can use instance of derived)
- Interface segregation principle (don't force client to implement an interface if they don't need it)
- High cohension, loose coupling principle (path to design success)

An Introduction to Software Development Design Principles – GeeksforGeeks (3/2023)

SOLID principles

The SOLID ideas are

- The Single-responsibility principle: "There should never be more than one reason for a class to change." In other words, every class should have only one responsibility.
- The Open–closed principle: "Software entities ... should be open for extension, but closed for modification."[7]
- The Liskov substitution principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."
- The Interface segregation principle: "Clients should not be forced to depend upon interfaces that they do not use."
- The Dependency inversion principle: "Depend upon abstractions, [not] concretes."

Source: Wikipedia

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.

Design patterns

What is a design pattern?

Categories of design patterns

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

Let's look at code! (assess its style)

Many thanks to René Just, UW CSE Prof

Quiz setup

- Project groups or small teams of neighboring students
- 6 code snippets

- Round 1
  - For each code snippet, decide if it represents good or bad practice
  - **Goal:** discuss and reach consensus on good or bad practice

- Round 2 (Discussion)
  - For each code snippet, try to understand why it is good or bad practice
  - **Goal:** come up with an explanation or a counterargument

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;
   }
}
```

Round 1: good or bad?

and Round 2: why?

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 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;
   }
}
```

```
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: good or bad?

```
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)
}
```

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 2: short but bad! why?

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

Consider always returning a success/failure value.

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 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: 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 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 (sacred);
use local variables to sanitize inputs.

Snippet 5: good or bad?

```
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) {
        …
    }
    …
}
```

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.

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);
```

Hesitate to use overloading and different return values

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;
   }
}
```

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.

All for now on design

- We'll do a double click on UI design later in the course – it's a course in itself, CSE 440 – Intro to HCI

- Review the design primer in the following slides to refresh your knowledge of design considerations for your project

Additional Design Material

Provided by René Just, UW CSE Professor

Concepts covered in CSE 331 – Software design and implementation

UML crash course

UML crash course

The main questions

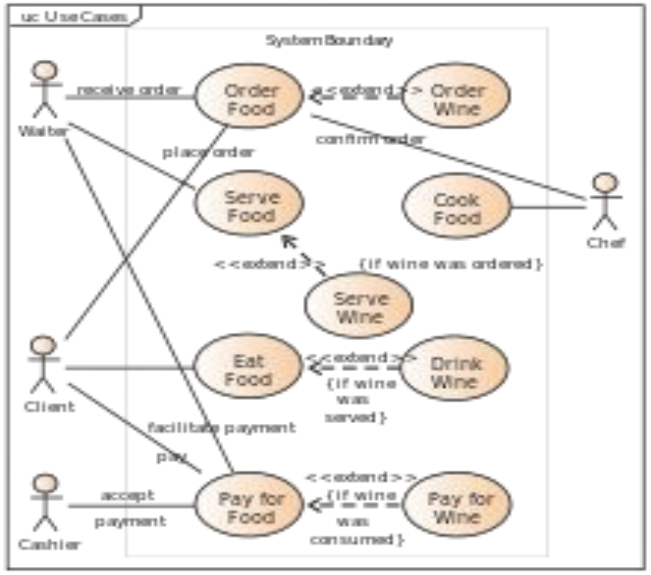
- What is UML?
- Is it useful, why bother?
- When to (not) use UML?

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
  - ...

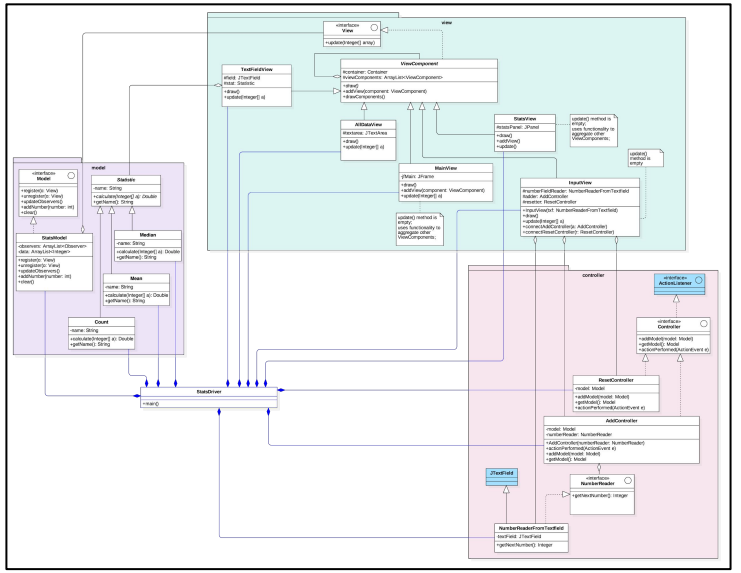
What is UML?

- 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
  - ...

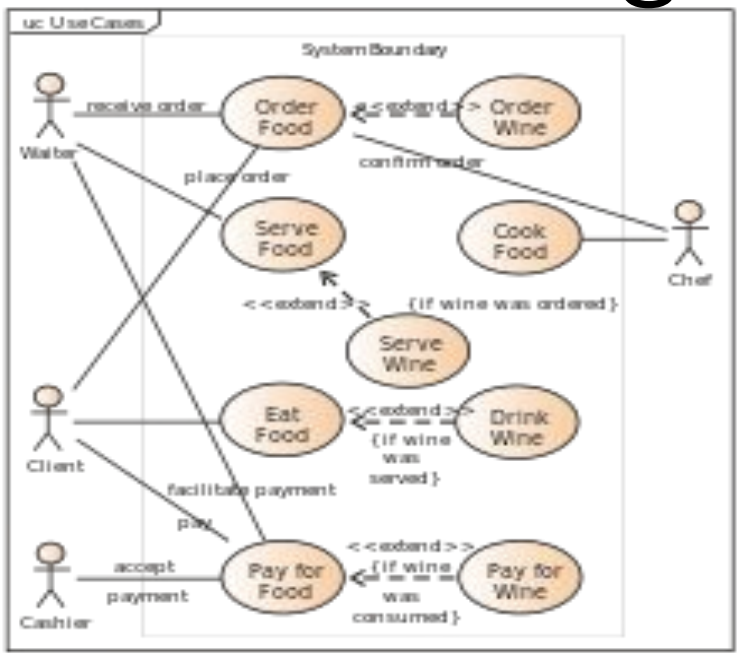


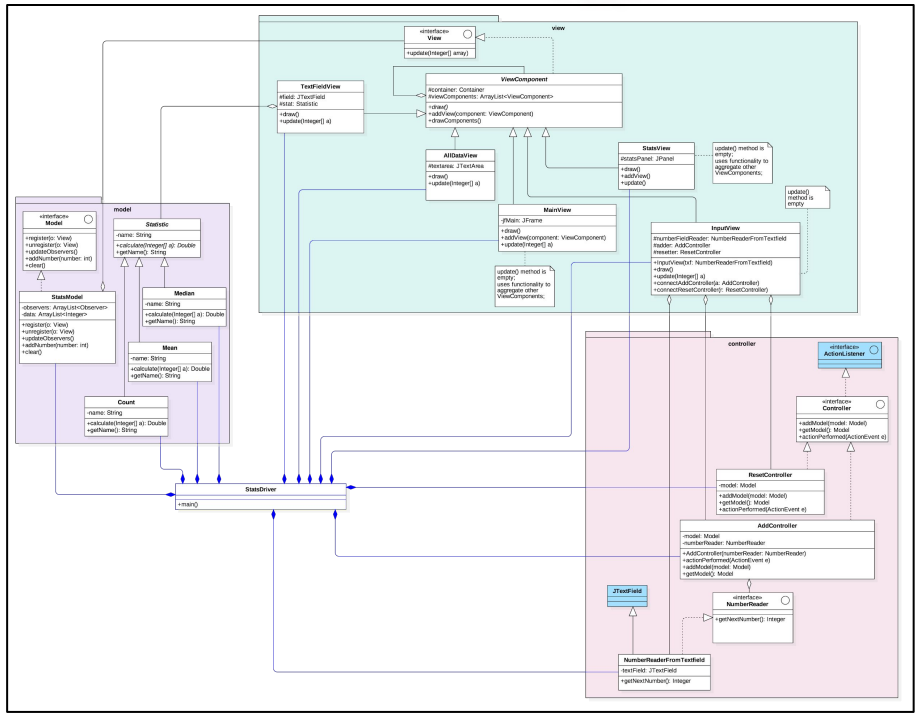
What is UML?

- 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

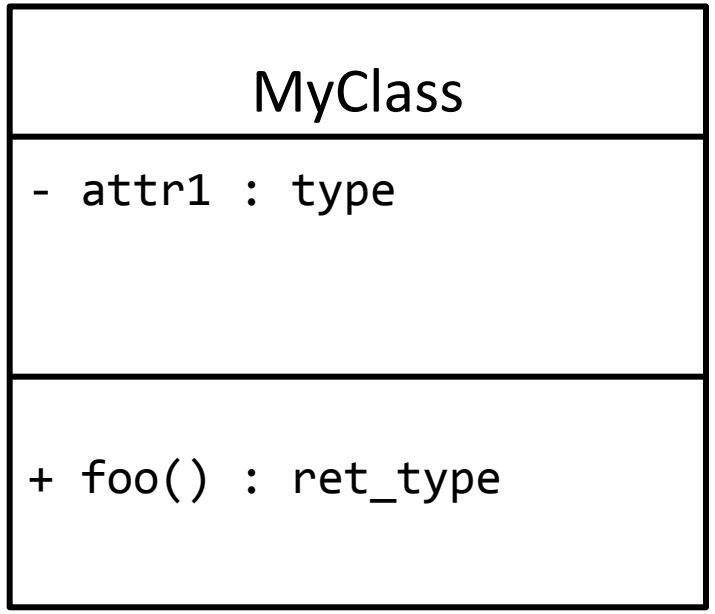
- Grouping of similar objects.
  - Student
  - Car
- Abstraction of common properties and behavior.
  - Student: Name and Student ID
  - Car: Make and Model

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

UML class diagram: basic notation



Name

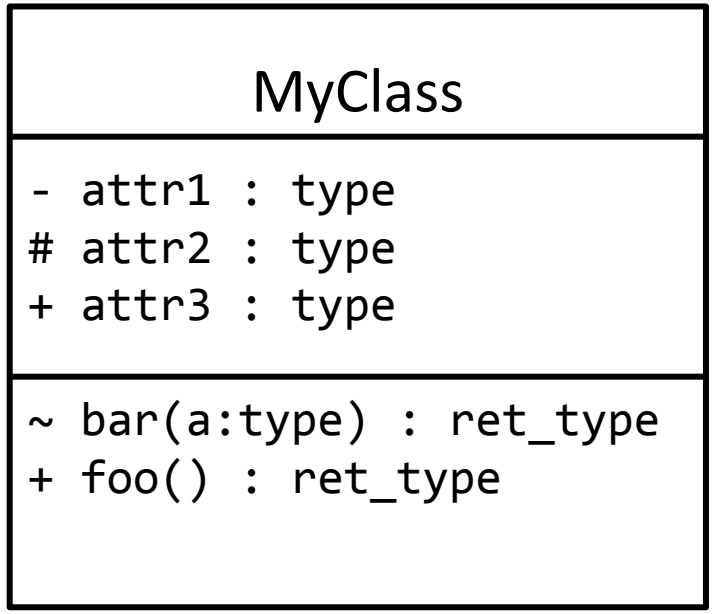
Attributes

<visibility> <name> : <type>

Methods

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

UML class diagram: basic notation



Name

Attributes

<visibility> <name> : <type>

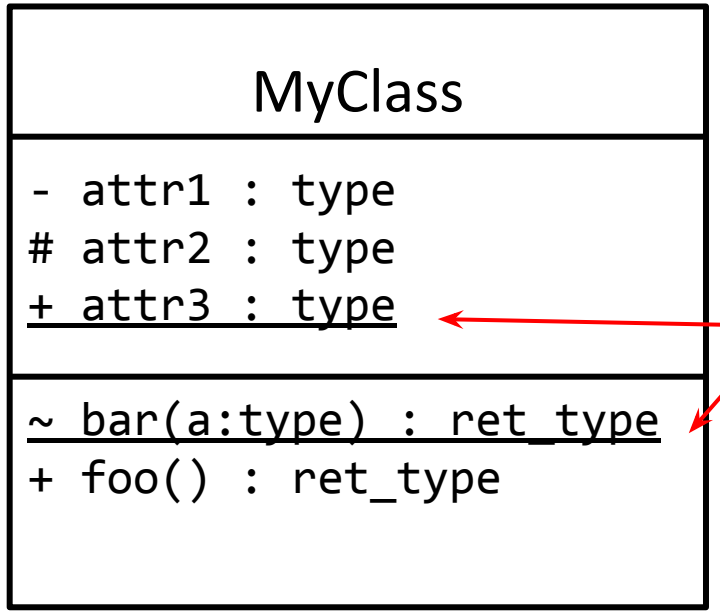
Methods

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

Visibility

- private
~ package-private
 # protected
+ public

UML class diagram: basic notation



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

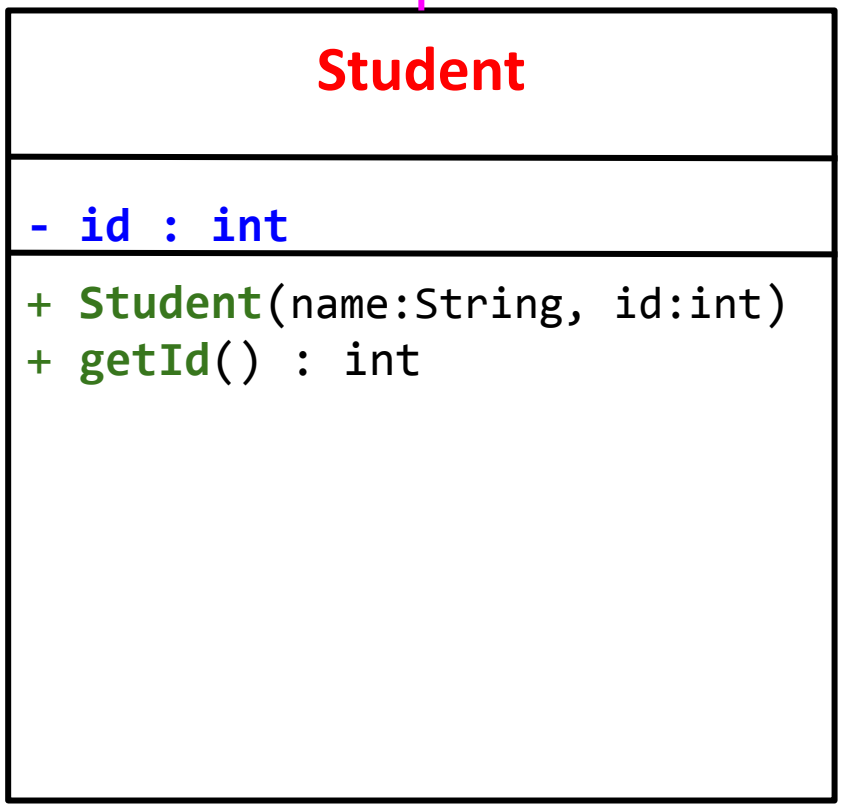
```
public class Person {
  …
}
```


```
public class Student
    extends Person {

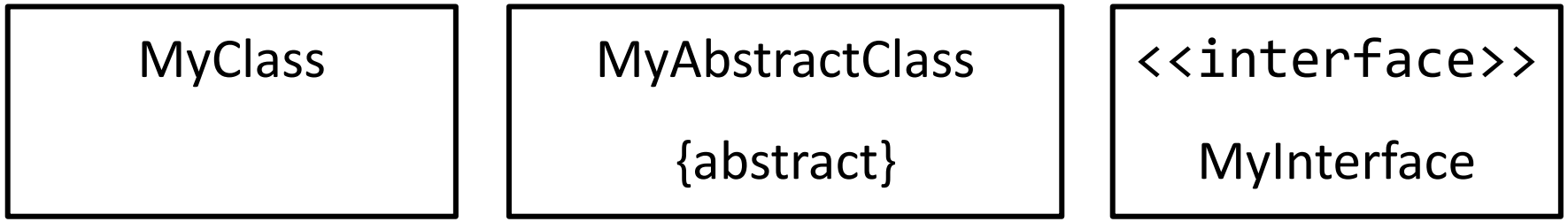
  private int id;

  public Student(String name,
                 int id) {
    ...
  }

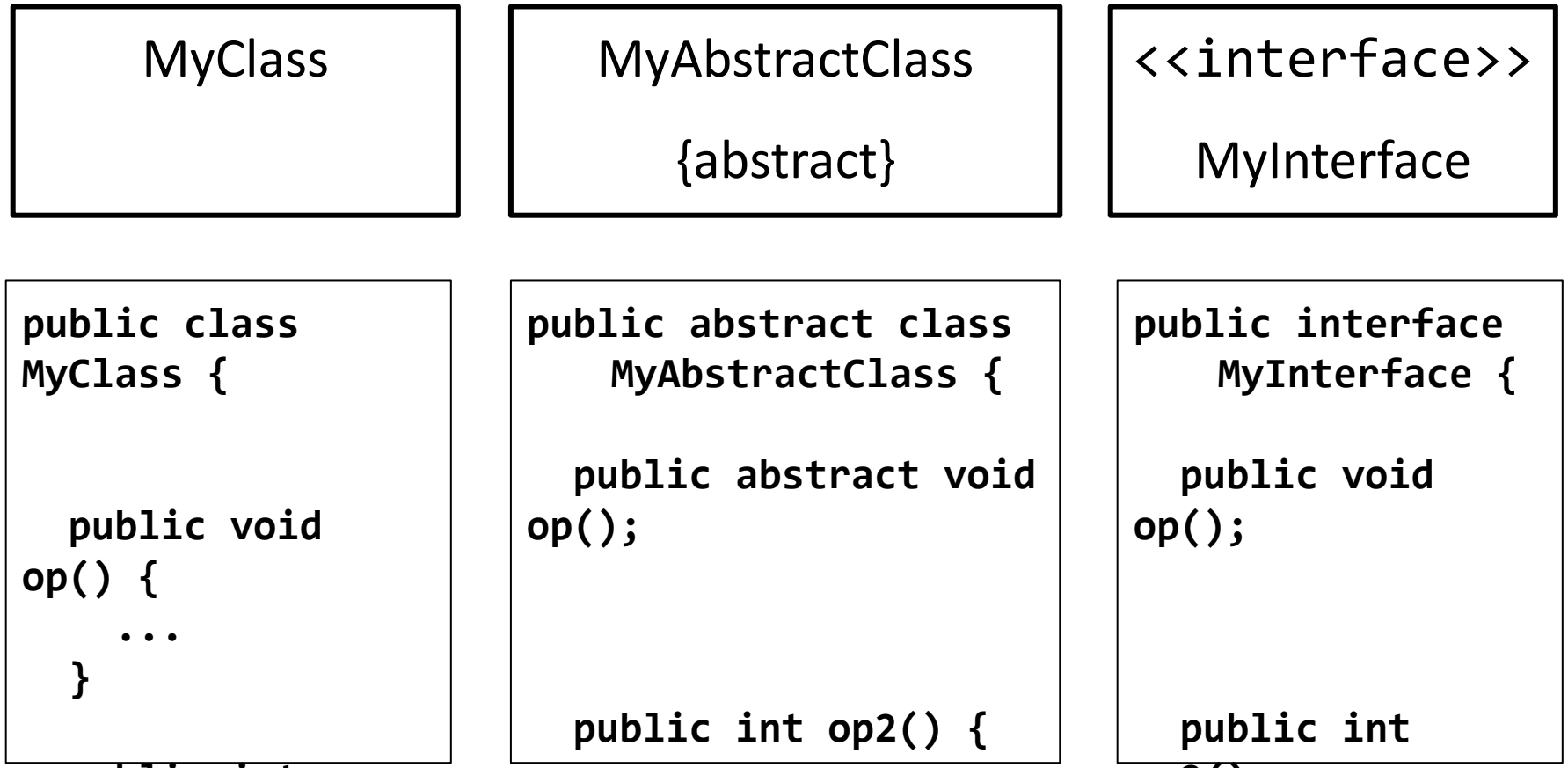
  public int getId() {
    return this.id;
  }
}
```



Classes, abstract classes, and interfaces

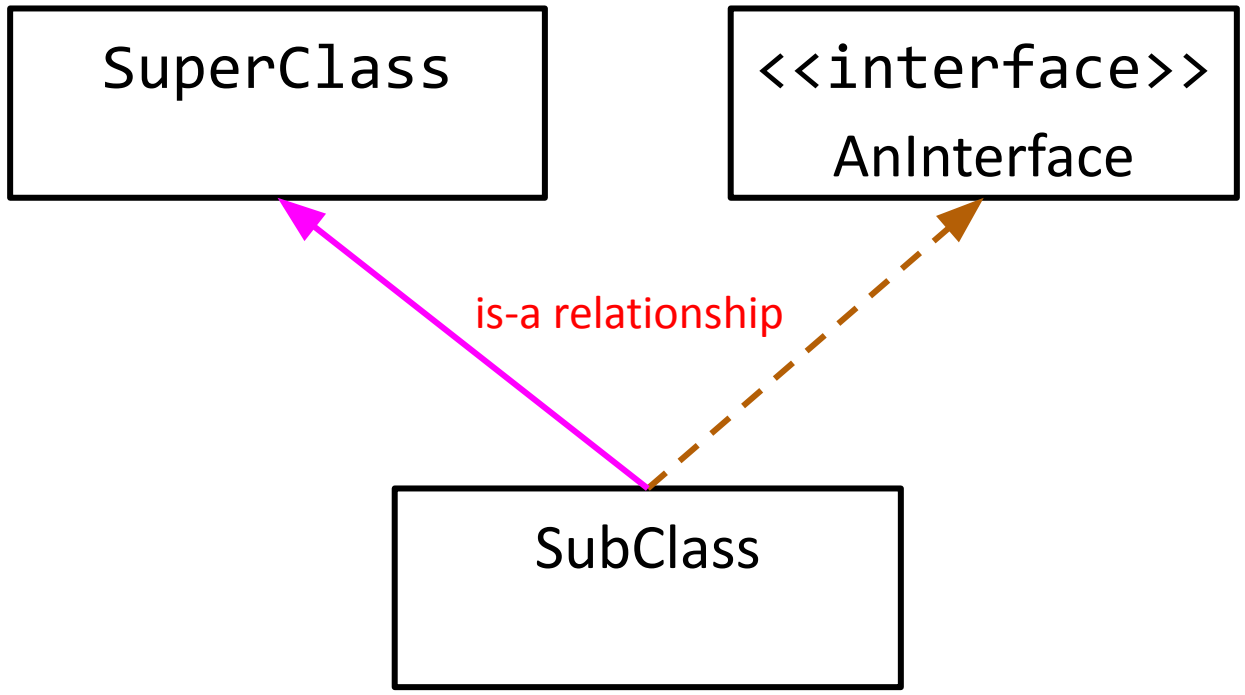


Classes, abstract classes, and interfaces



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

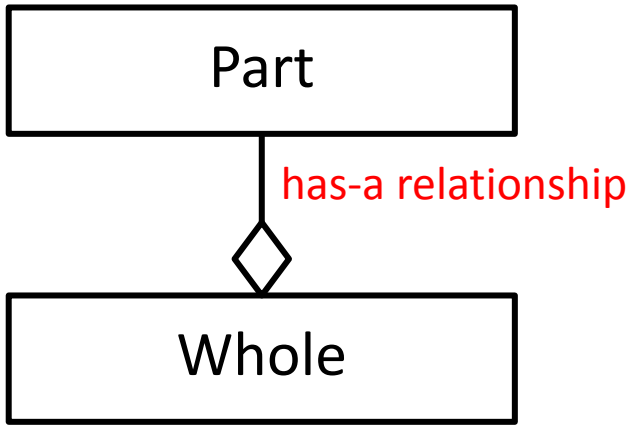
UML class diagram: Inheritance




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

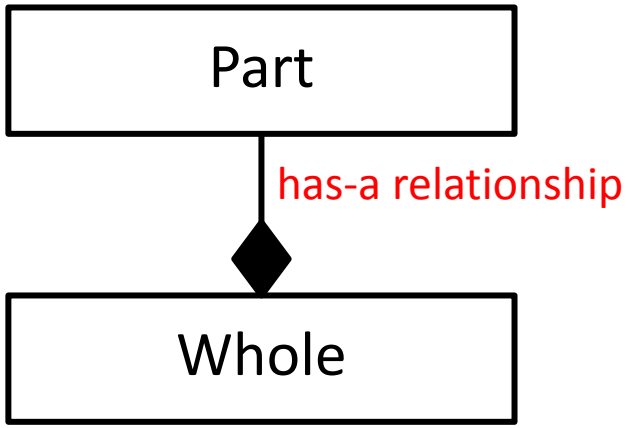
UML class diagram: Aggregation & Composition

Aggregation



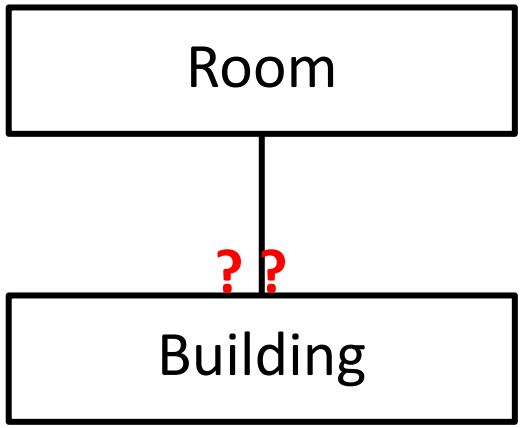
- Existence of Part does not depend on the existence of Whole.
- Lifetime of Part does not depend on Whole.
- No single instance of whole is the unique owner of Part (might be shared with other instances of Whole).

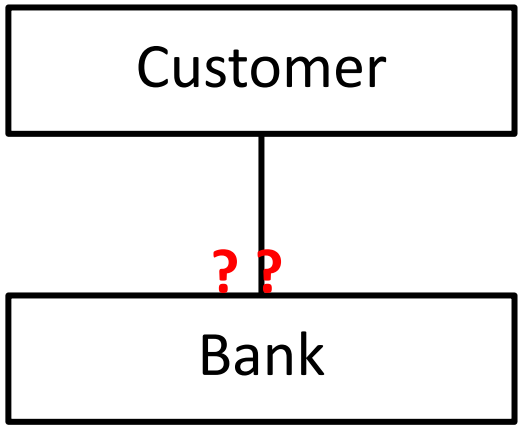
Composition



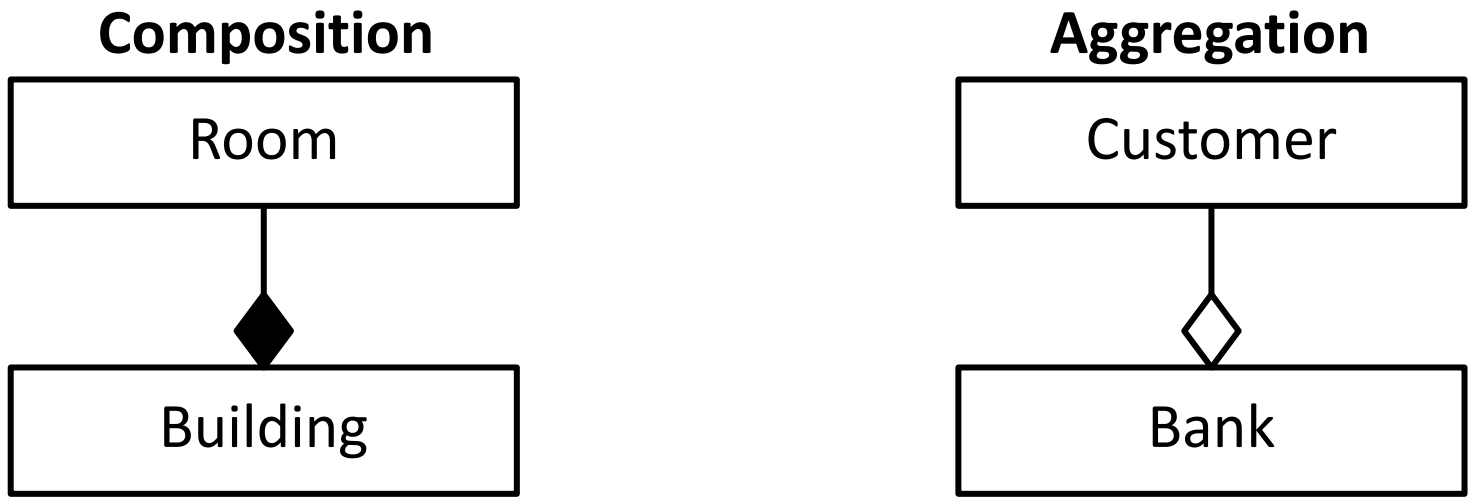
- Part cannot exist without Whole.
- Lifetime of Part depends on Whole.
- One instance of Whole is the single owner of Part.

Aggregation or Composition?



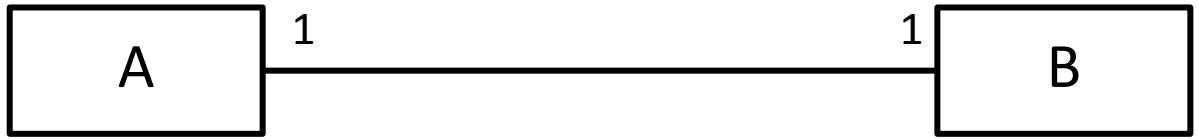


Aggregation or Composition?

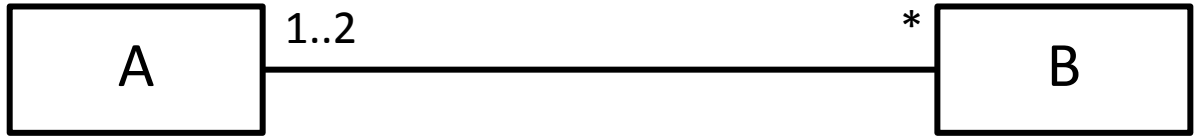


What about class and students or body and body parts?

UML class diagram: multiplicity

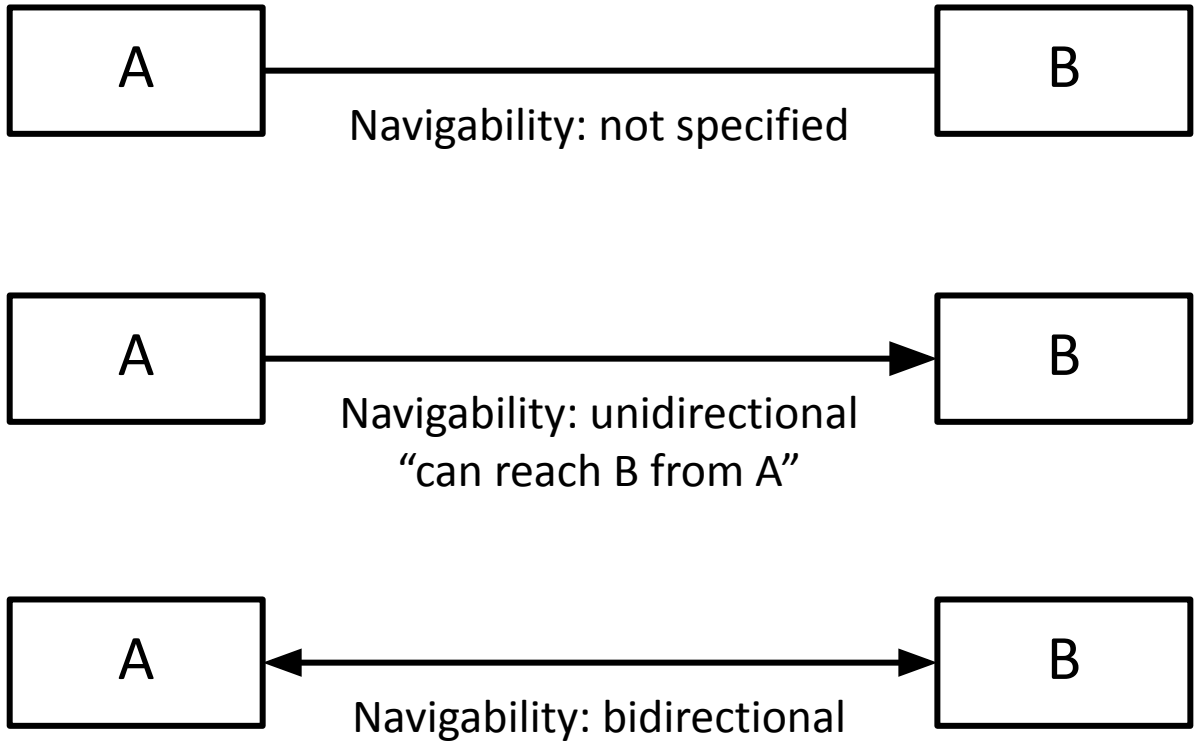


Each A is associated with exactly one B
Each B is associated with exactly one A

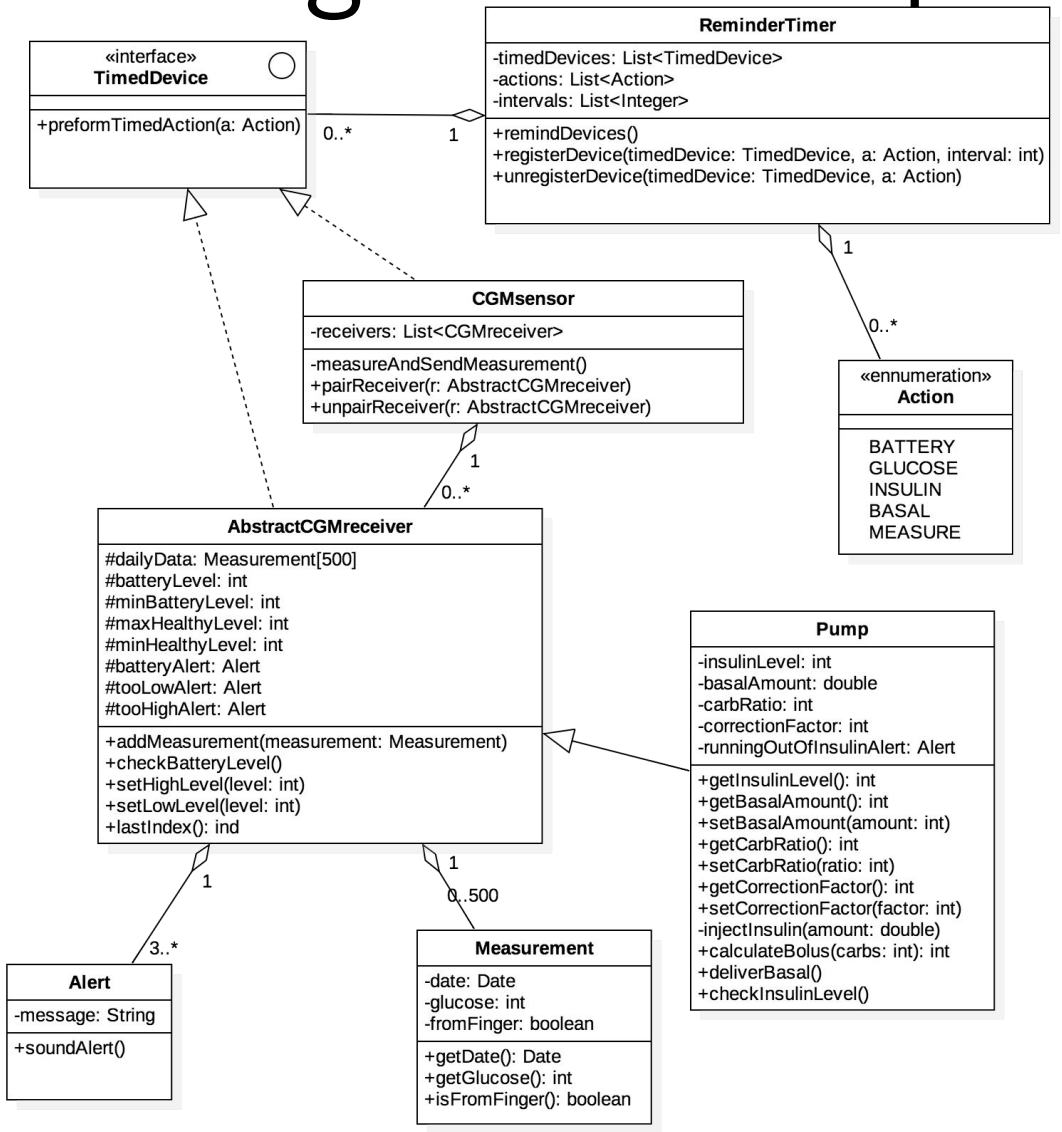


Each A is associated with any number of Bs
Each B is associated with exactly one or two As

UML class diagram: navigability



UML class diagram: example



Summary: UML

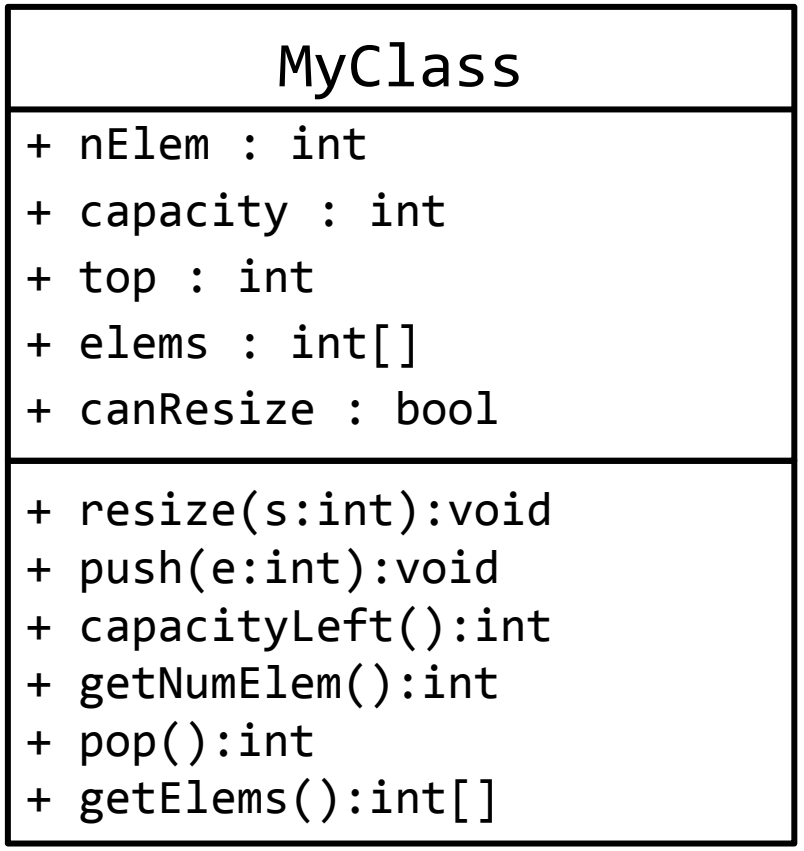
- Unified notation for modeling OO systems.
- Allows different levels of abstraction.
- Suitable for design discussions and documentation.

OO design principles

OO design principles

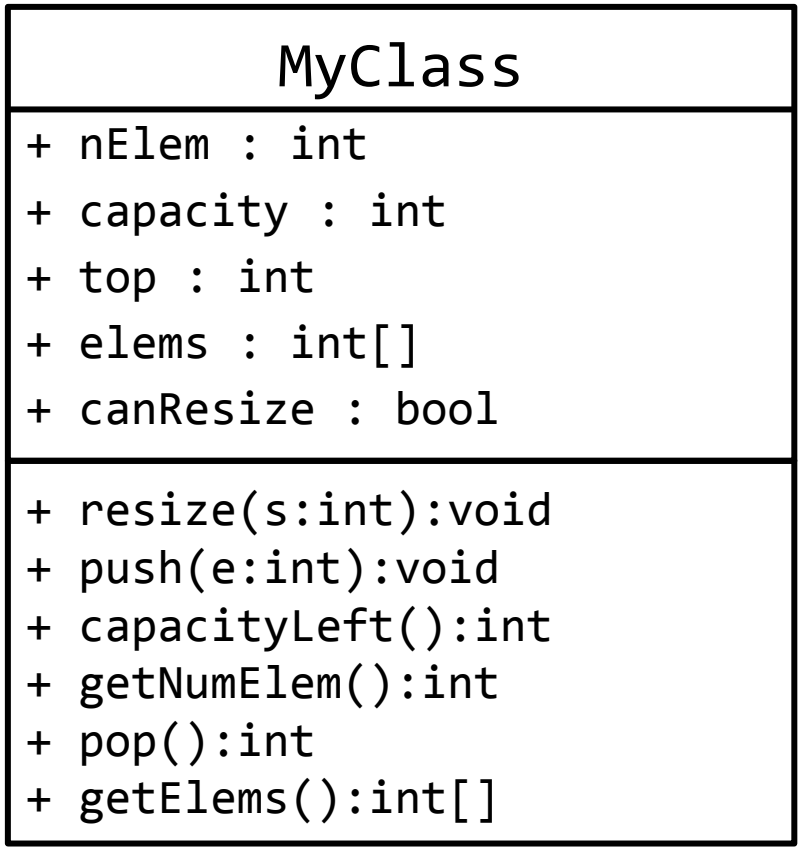
- **Information hiding (and encapsulation)**
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Information hiding




```
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(){...}
}
```

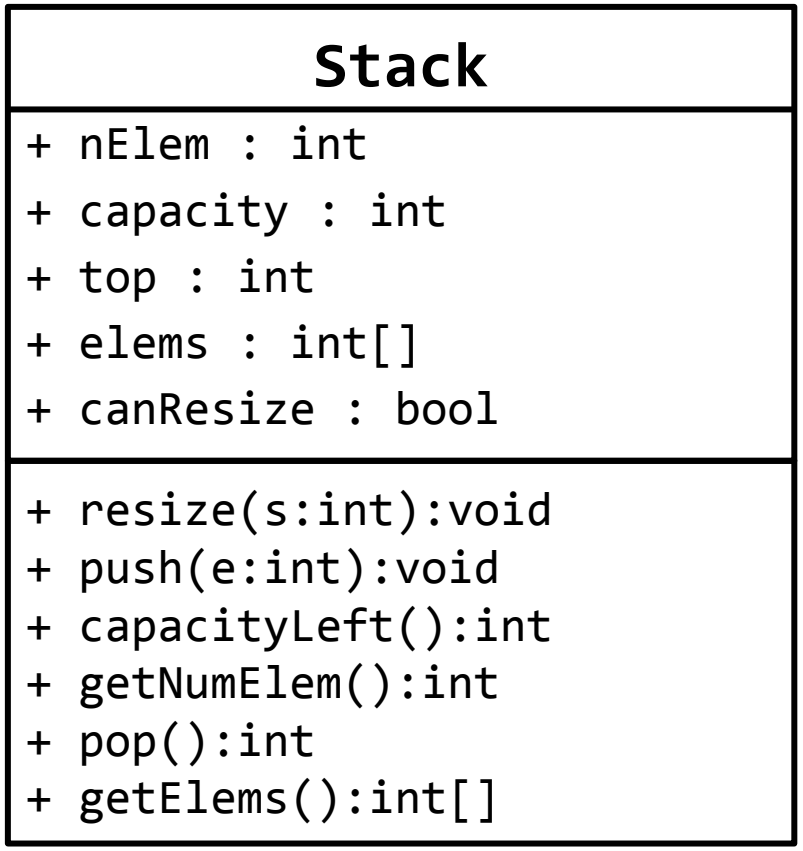
Information hiding




```
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?

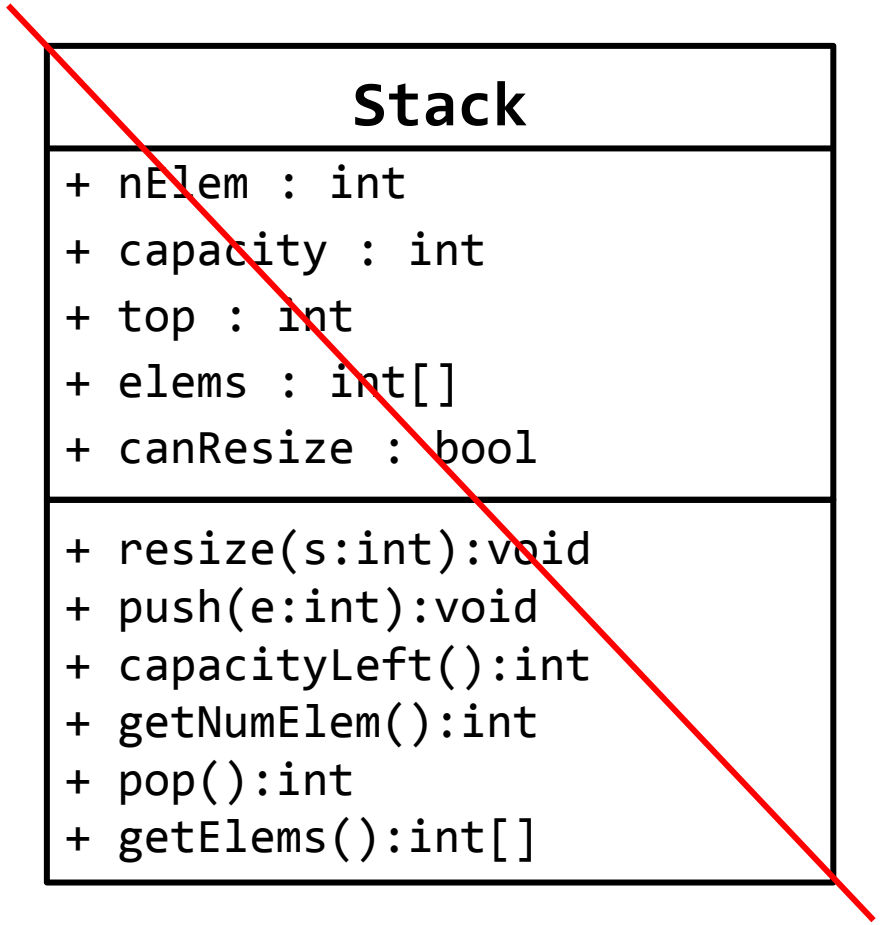
Information hiding

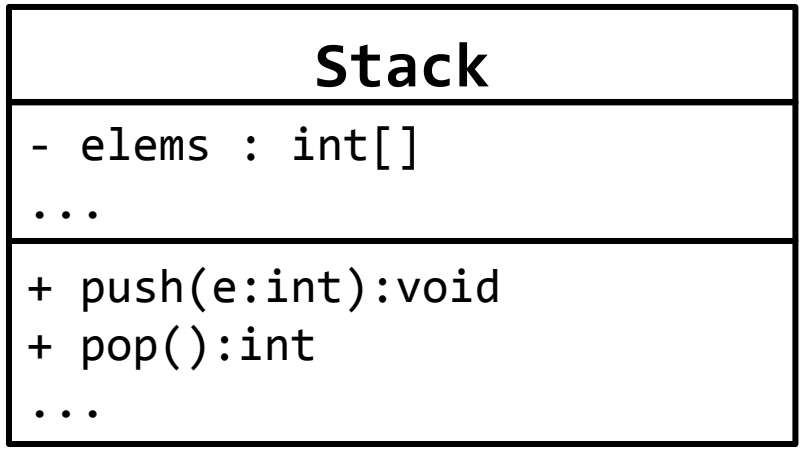



```
public class Stack {
  public int nElem;
  public int capacity;
  public int top;
  public int[] elems;
  public boolean canResize;
  ...
  public void resize(int s){...}
  public void push(int e){...}
  public int capacityLeft(){...}
  public int getNumElem(){...}
  public int pop(){...}
  public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

Anything that could be improved in this implementation?

Information hiding

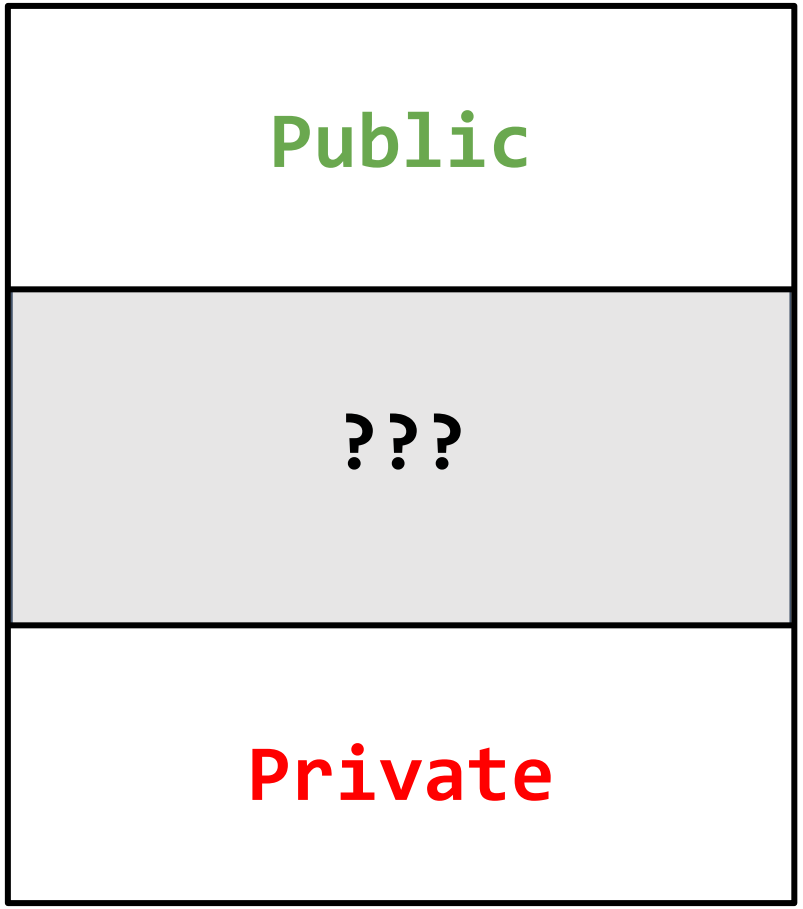




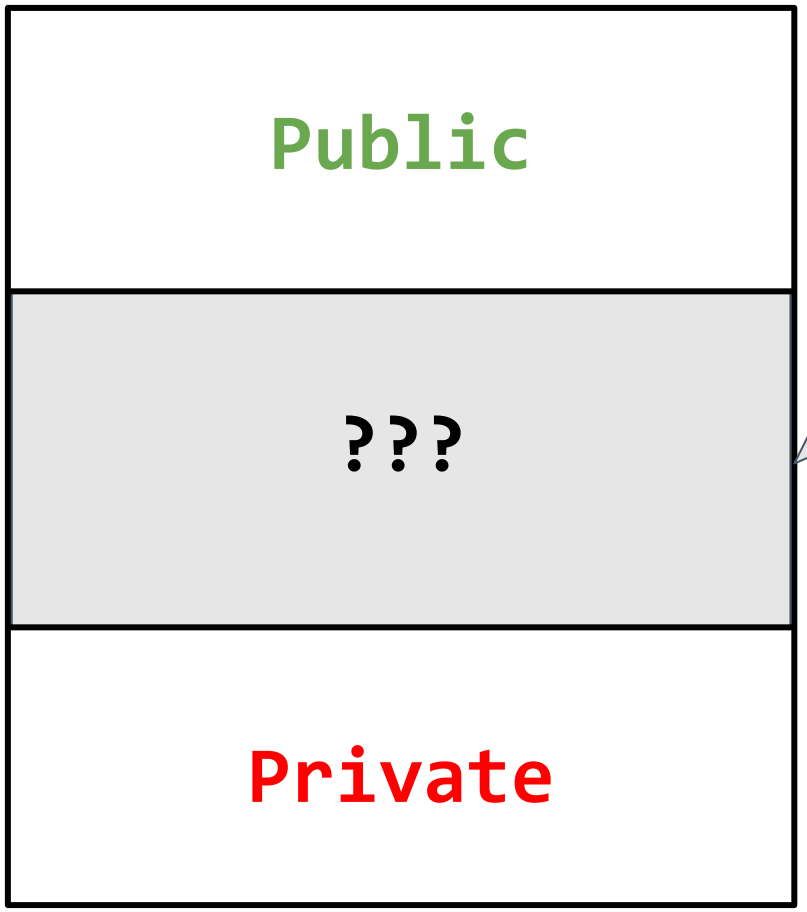
Information hiding:

- 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 = ...;` ⇒ toString() can be overridden in subclasses
    `obj.toString();` 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 = ...;` ⇒ toString() can be overridden in subclasses
    `obj.toString();` 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

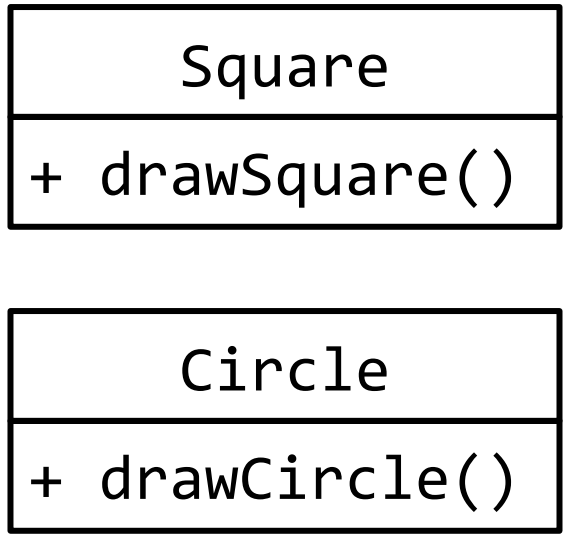
Open/closed principle

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?



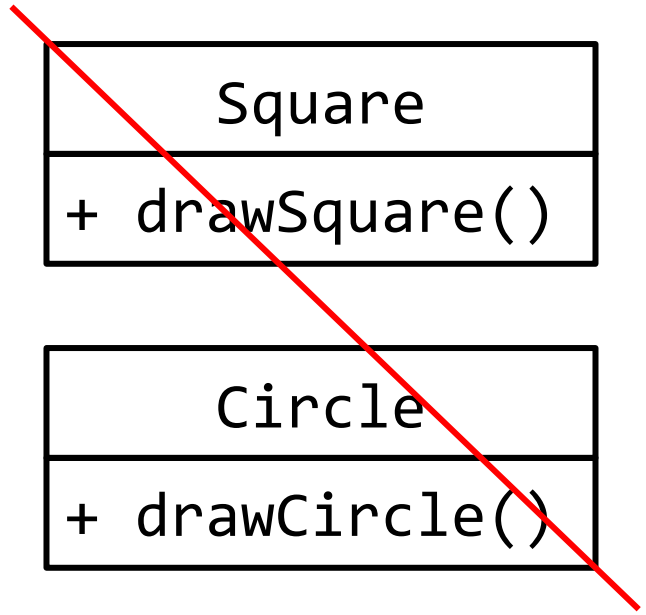
Open/closed principle

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!



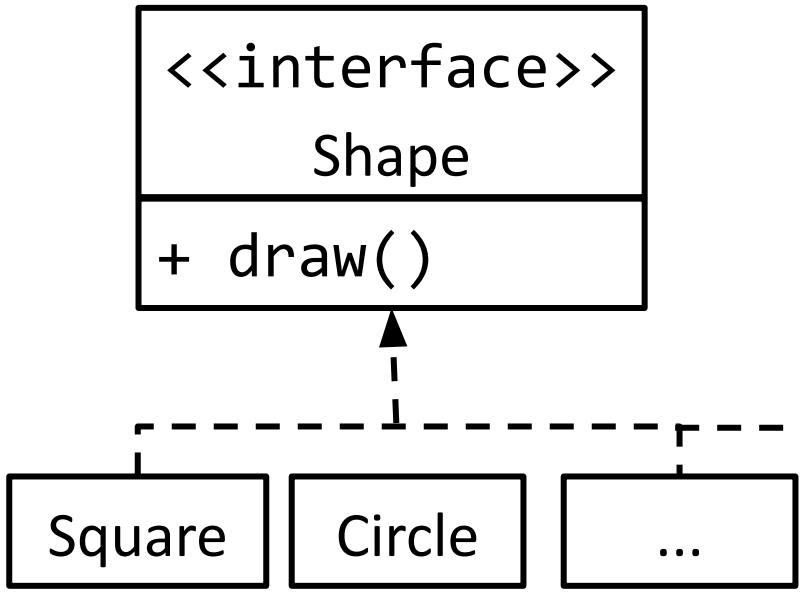
Open/closed principle

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 {
     …
  }
}
```

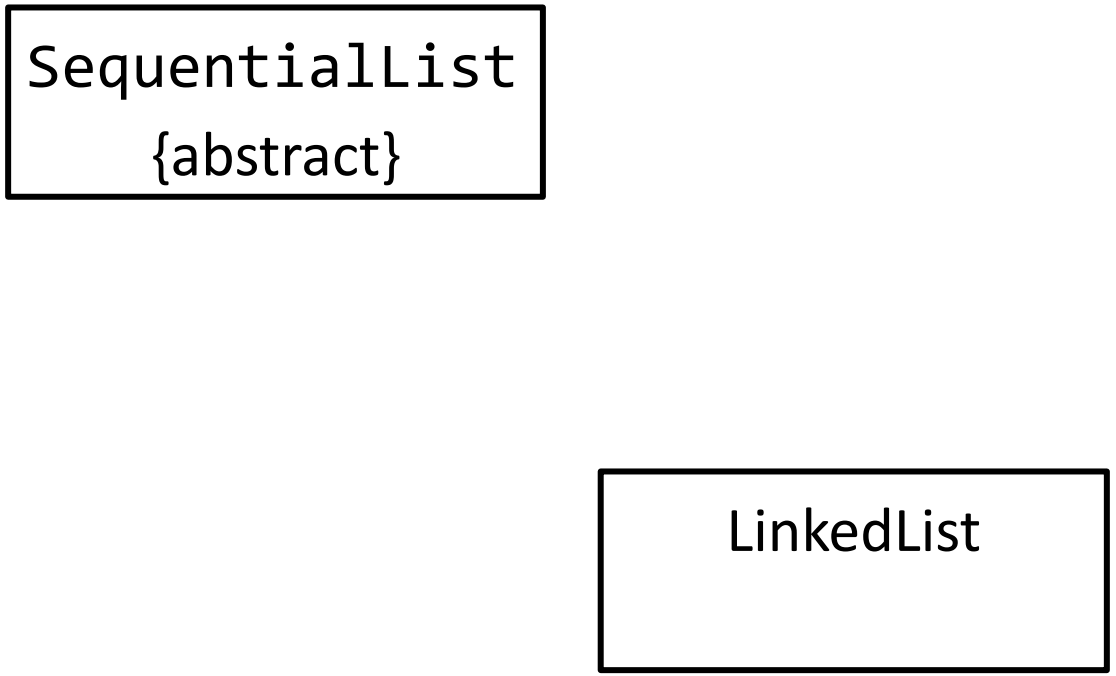
```
public static void draw(Shape s) {
  s.draw();
}
```



OO design principles

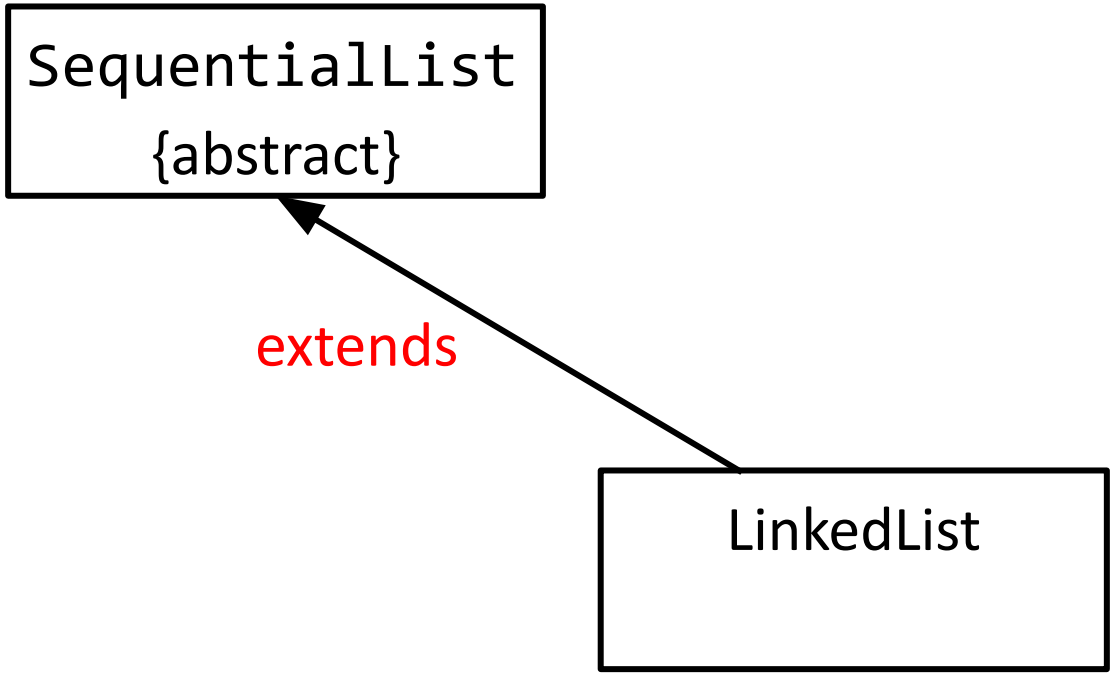
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- **Inheritance in Java**
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Inheritance: (abstract) classes and interfaces



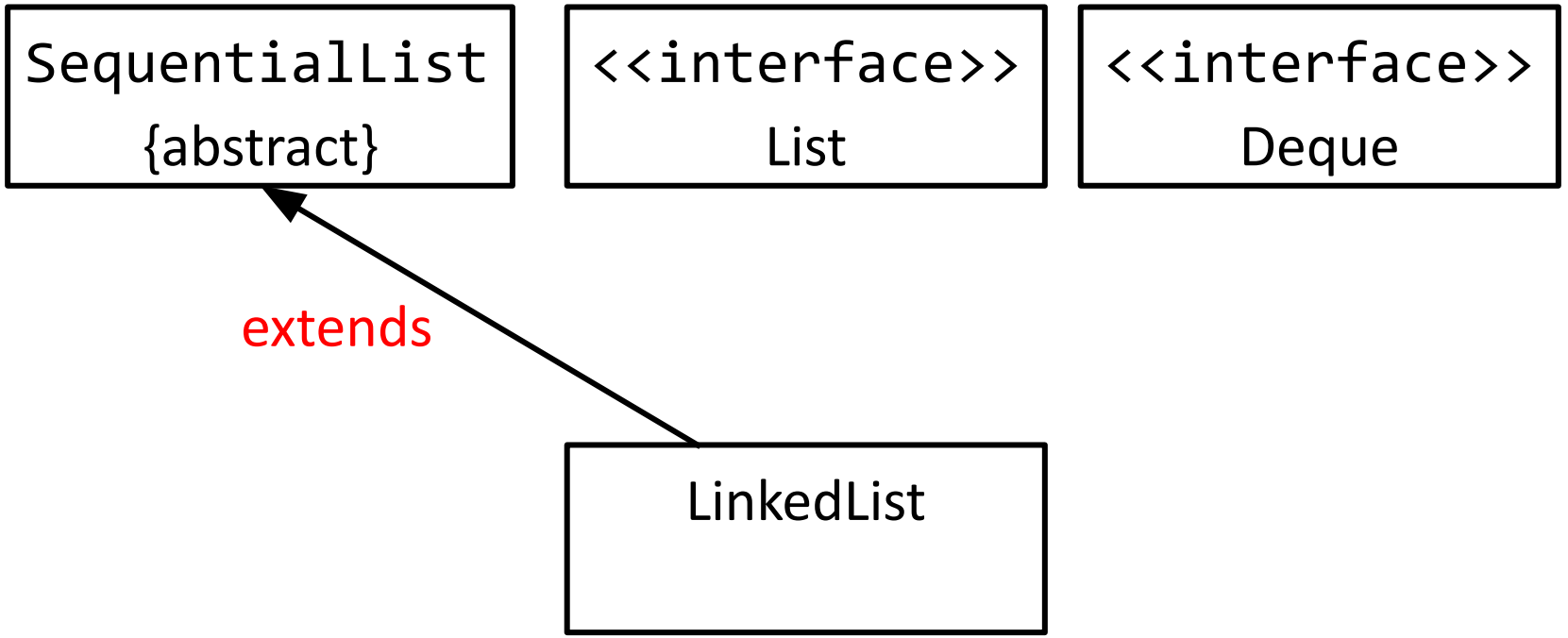
Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList



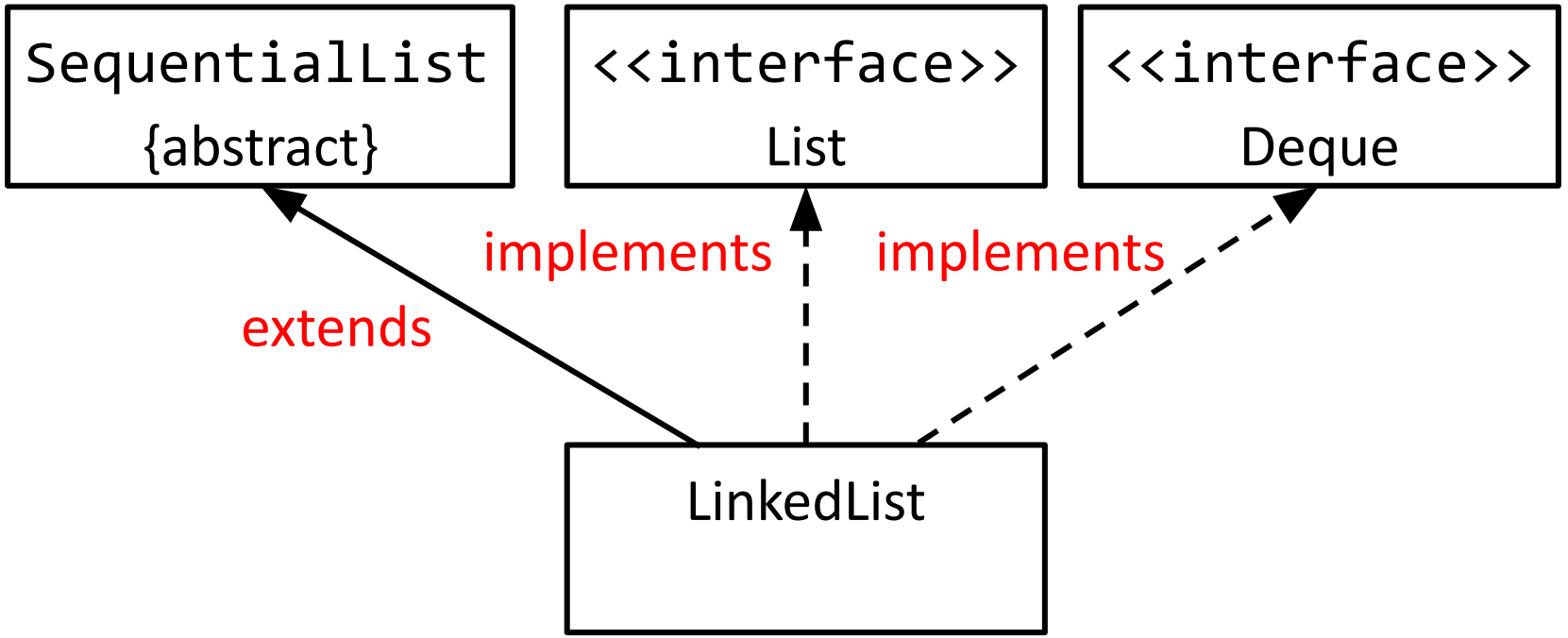
Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList

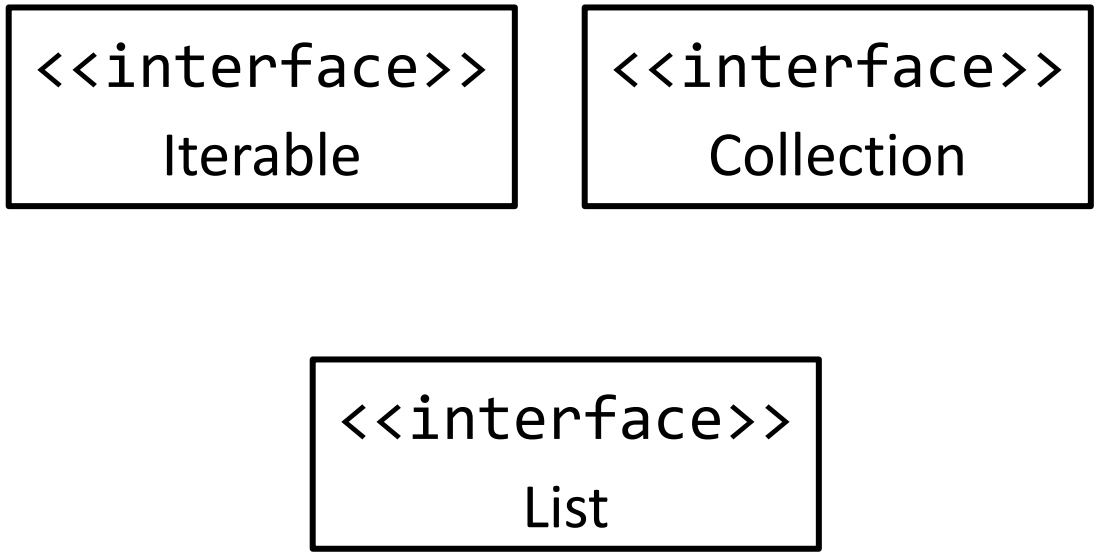


Inheritance: (abstract) classes and interfaces

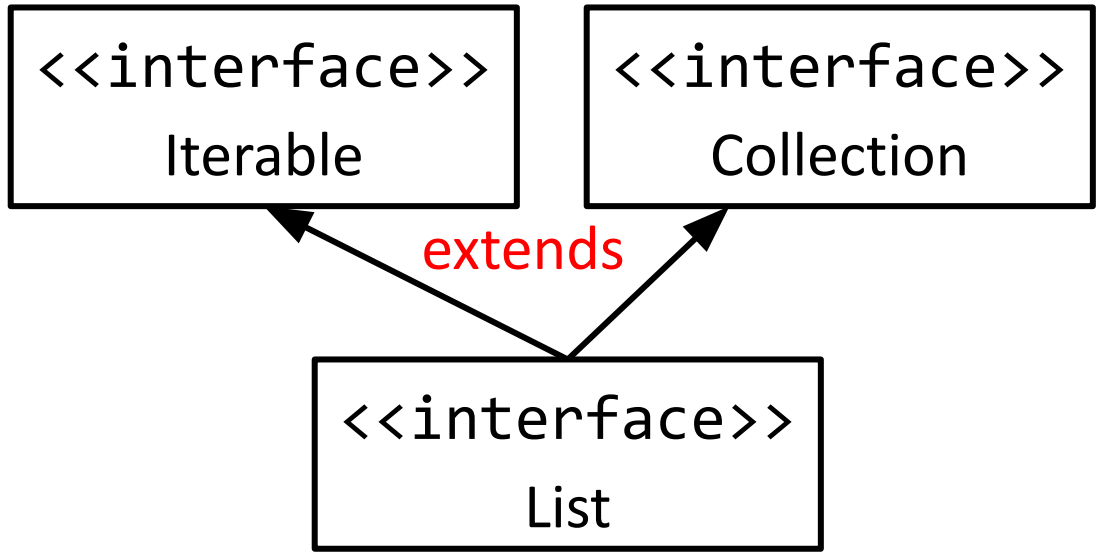
LinkedList extends SequentialList implements List, Deque



Inheritance: (abstract) classes and interfaces

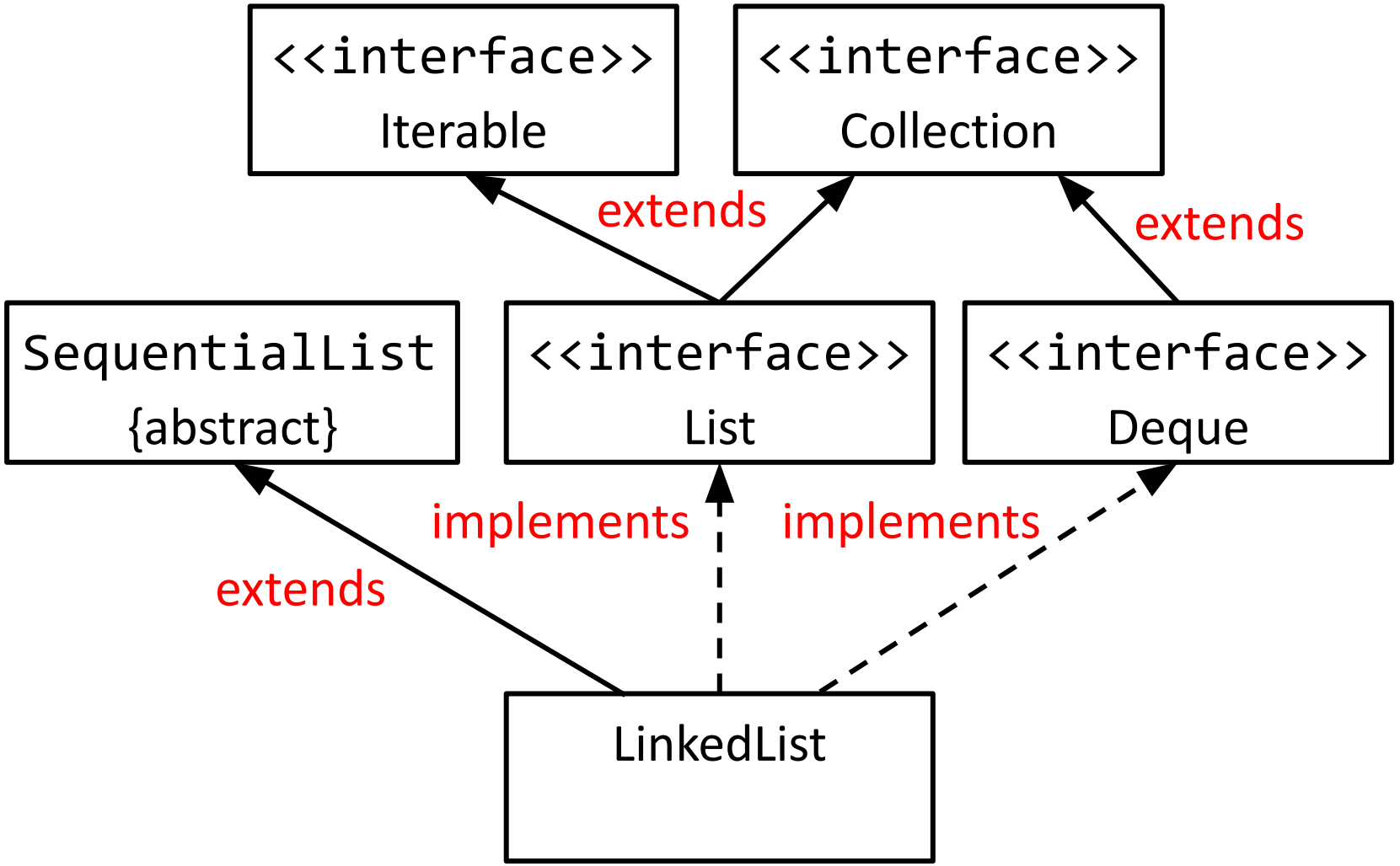


Inheritance: (abstract) classes and interfaces



List extends Iterable, Collection

Inheritance: (abstract) classes and interfaces

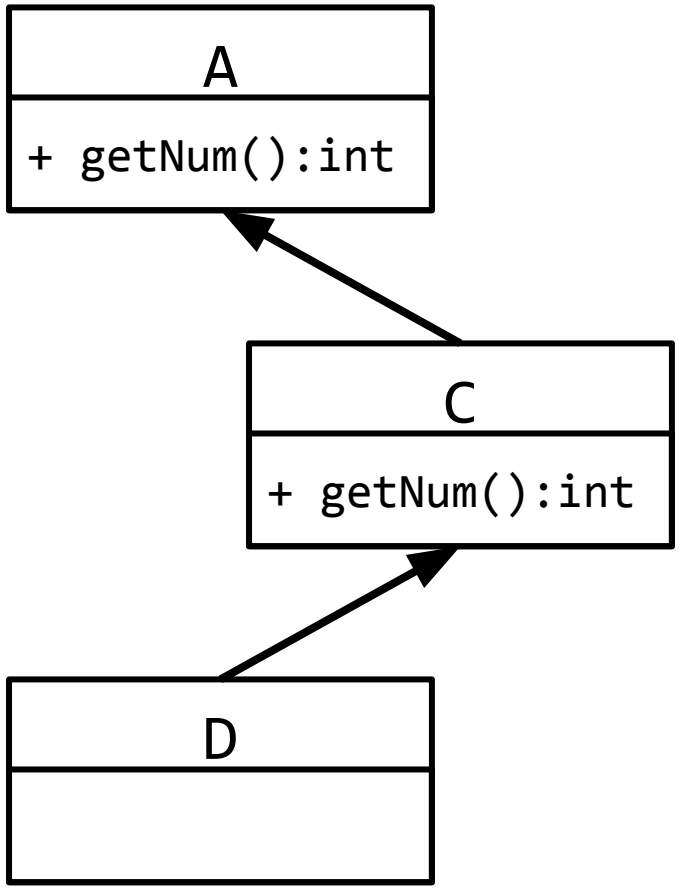


OO design principles

- 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

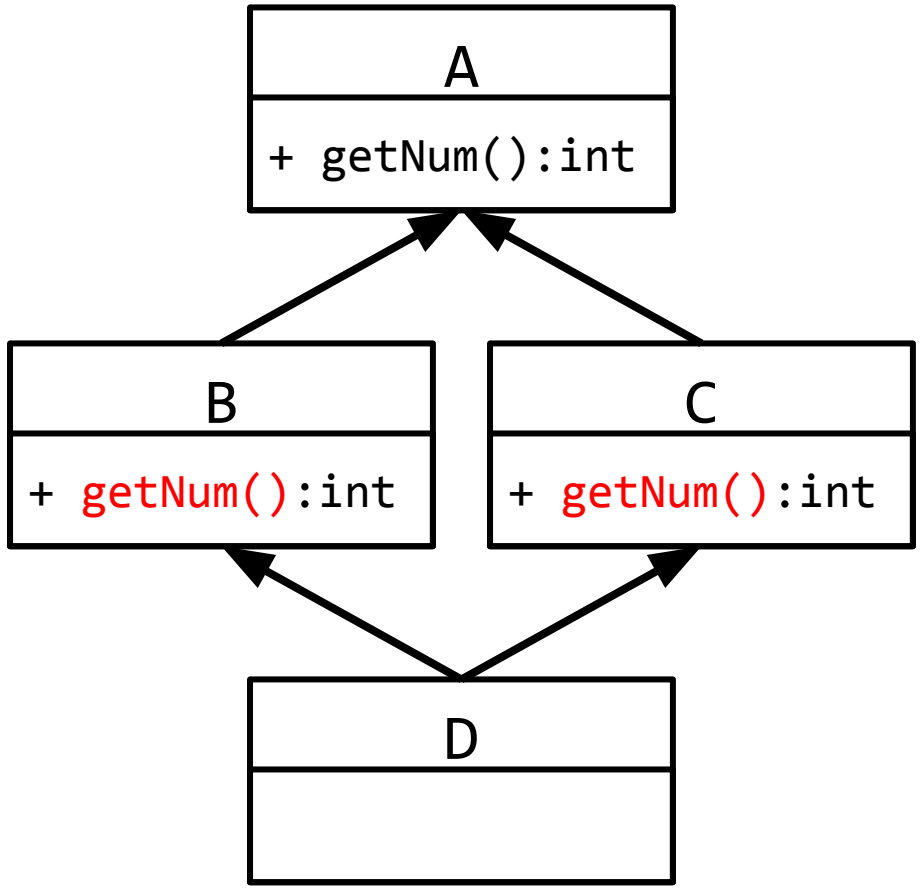
```
...
A a = new D();
int num = a.getNum();
...
```



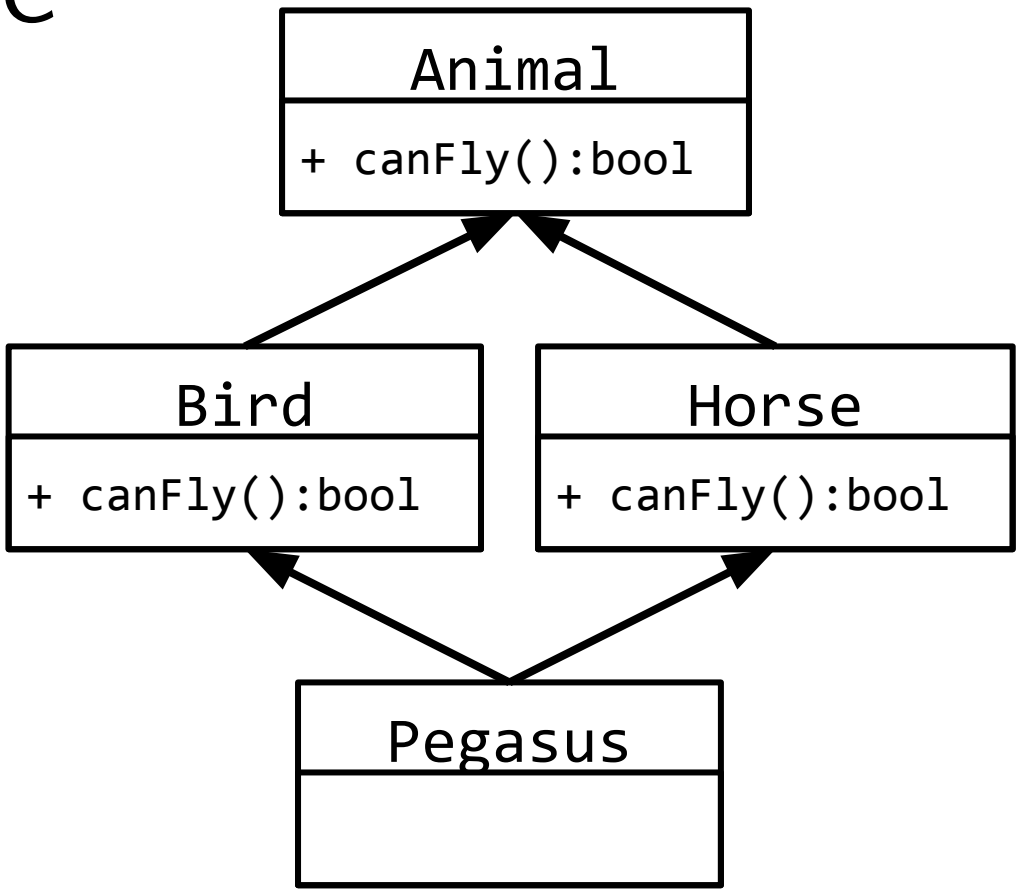
The “diamond of death”: the problem

```
...
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.

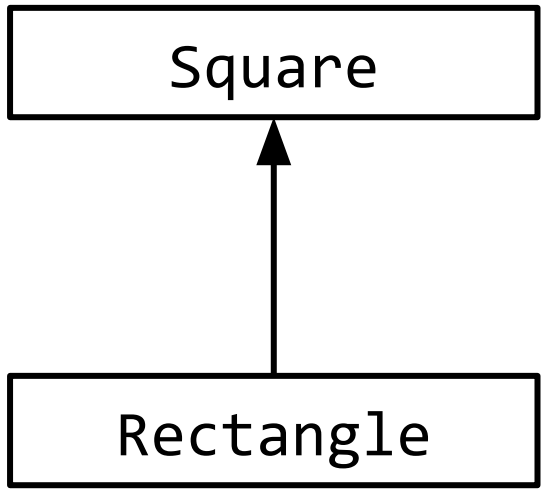
OO design principles

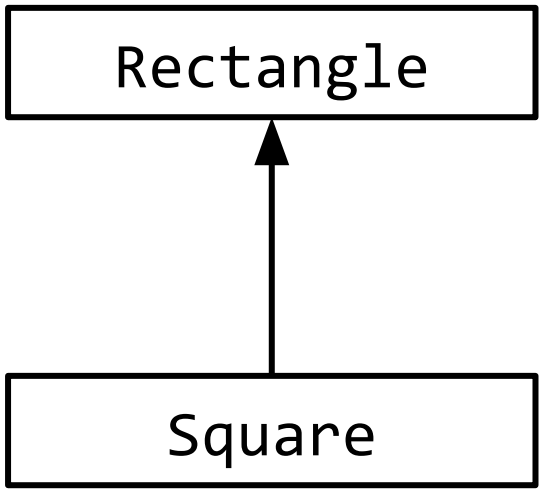
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- **Liskov substitution principle**
- Composition/aggregation over inheritance

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?

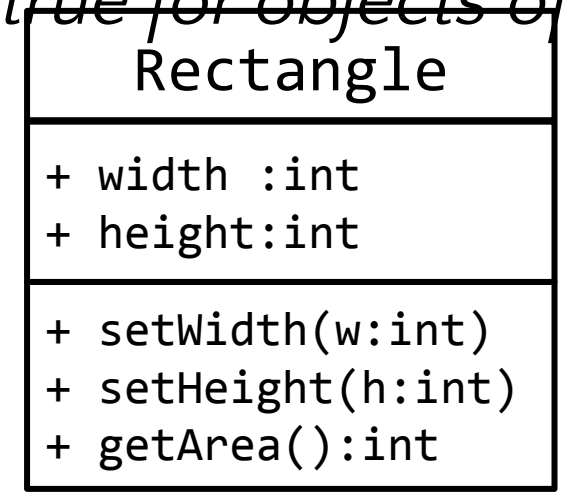


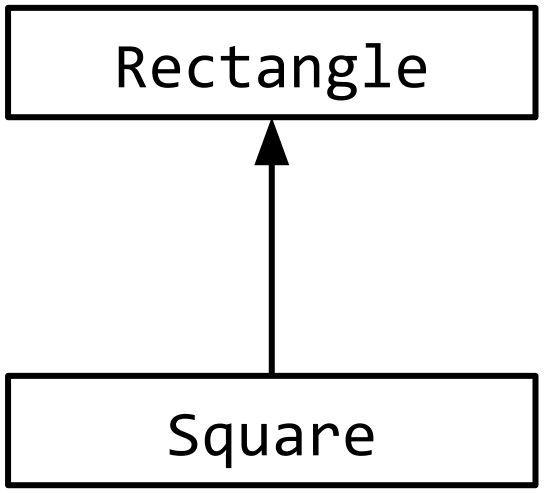


Design principles: Liskov substitution principle

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.



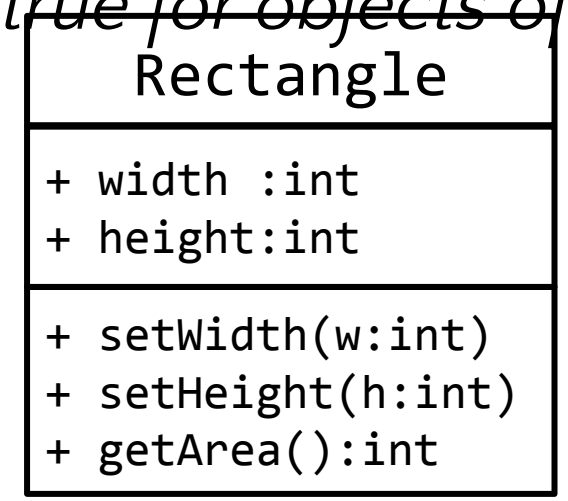


Is the subtype requirement fulfilled?

Design principles: Liskov substitution principle

Subtype requirement

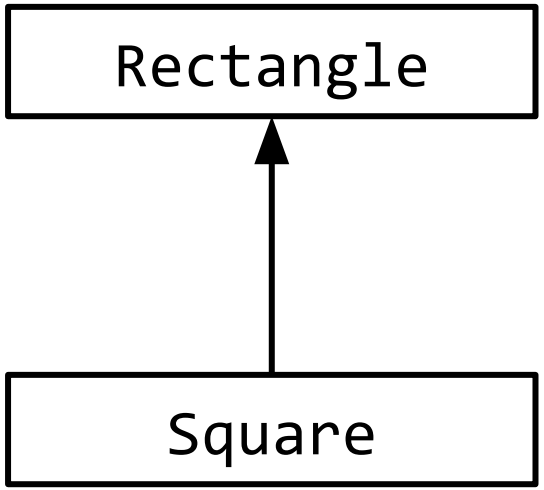
Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.




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


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

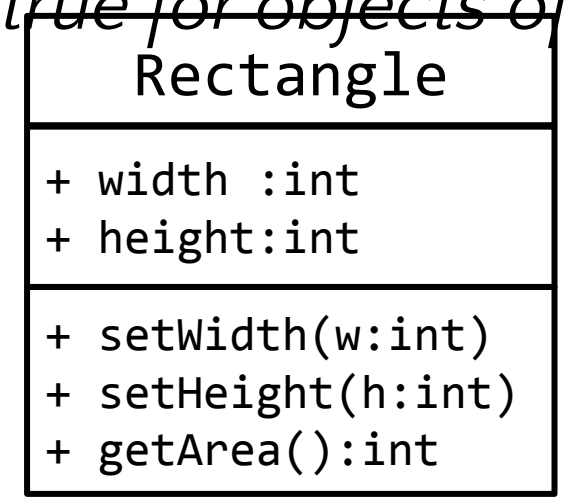
assertEquals(A * 2,
    r.getArea());
```



Design principles: Liskov substitution principle

Subtype requirement

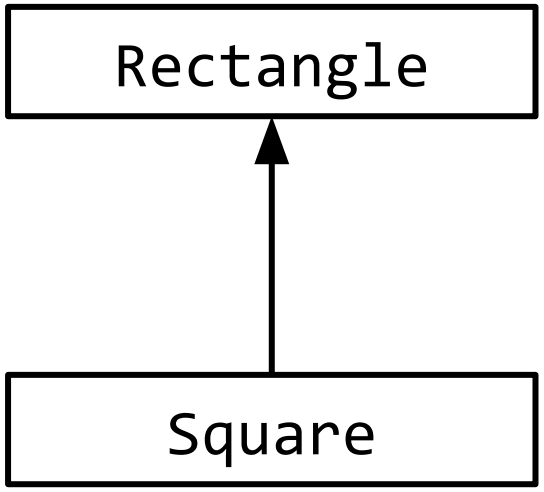
Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.




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

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

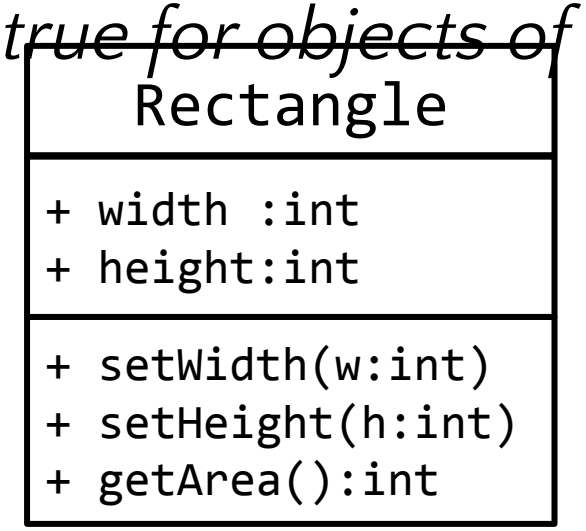
assertEquals(A * 2,
    r.getArea());
```



Design principles: Liskov substitution principle

Subtype requirement

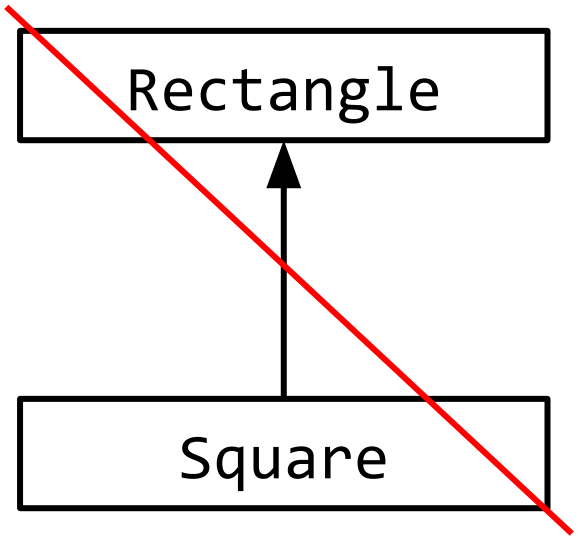
Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.




```
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());
```

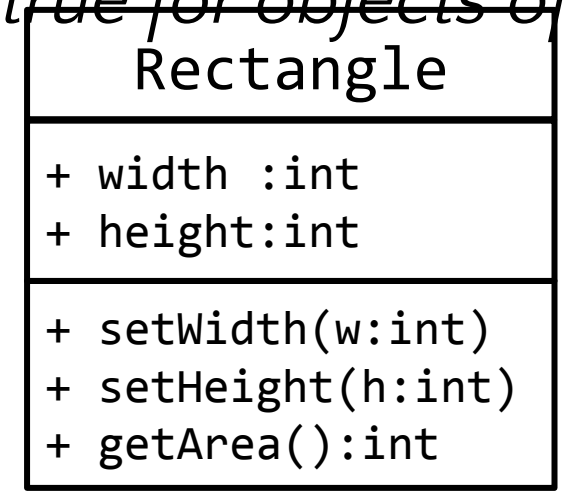


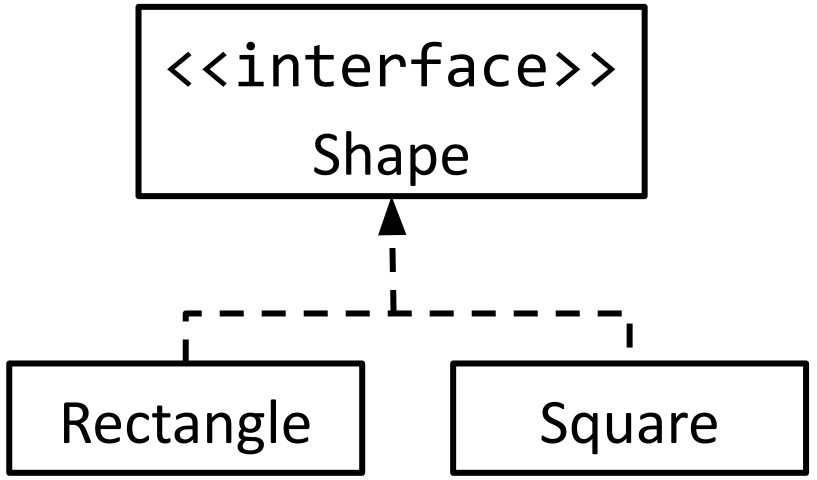
Violates the Liskov substitution principle!

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.

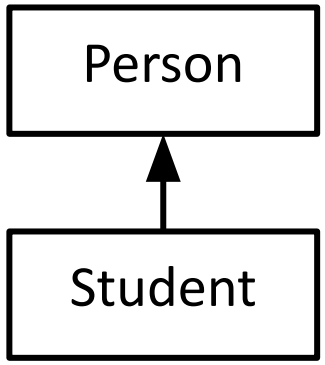




OO design principles

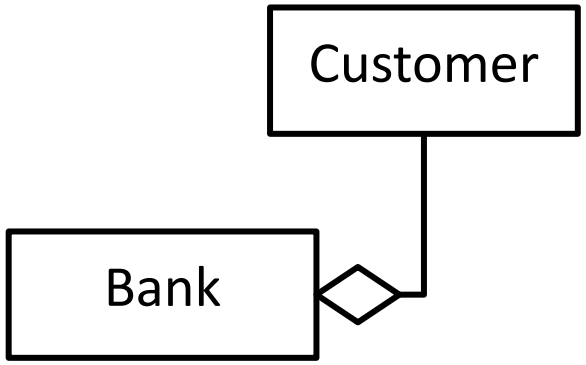
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- **Composition/aggregation over inheritance**

Inheritance vs. (Aggregation vs. *Composition*)

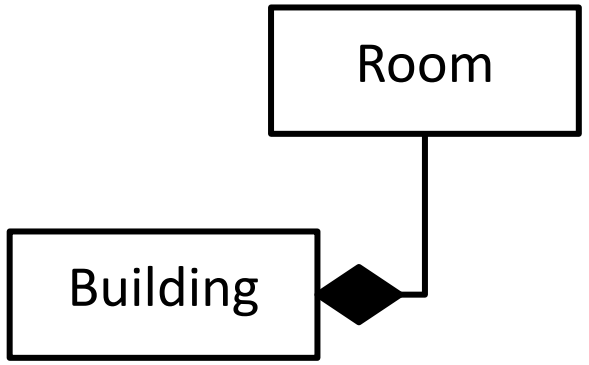



```
public class
Student
    extends
Person{
 public Student(){
 }
  ...
}
```

is-a relationship



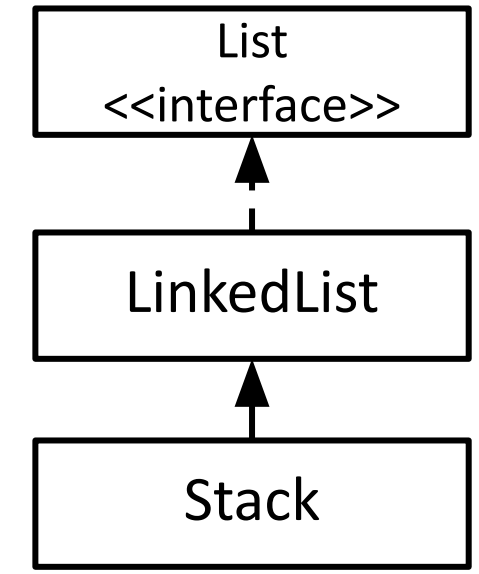

```
public class Bank {
 Customer c;
 public Bank(Customer
c){
  this.c = c;
 }
 ...
}
```

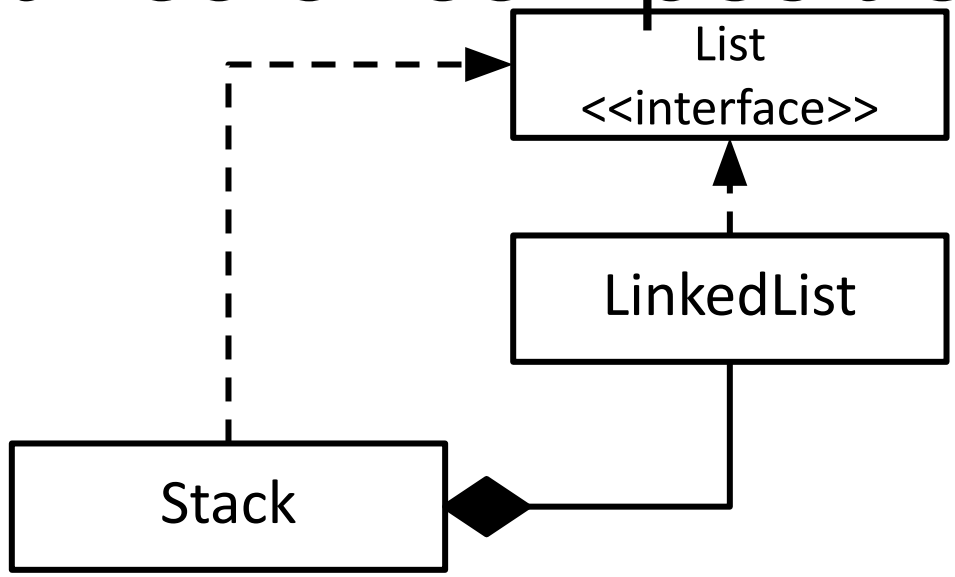



```
public class Building
{
 Room r;
 public Building(){
  this.r = new Room();
 }
 ...
}
```

has-a relationship

Design choice: inheritance or composition?



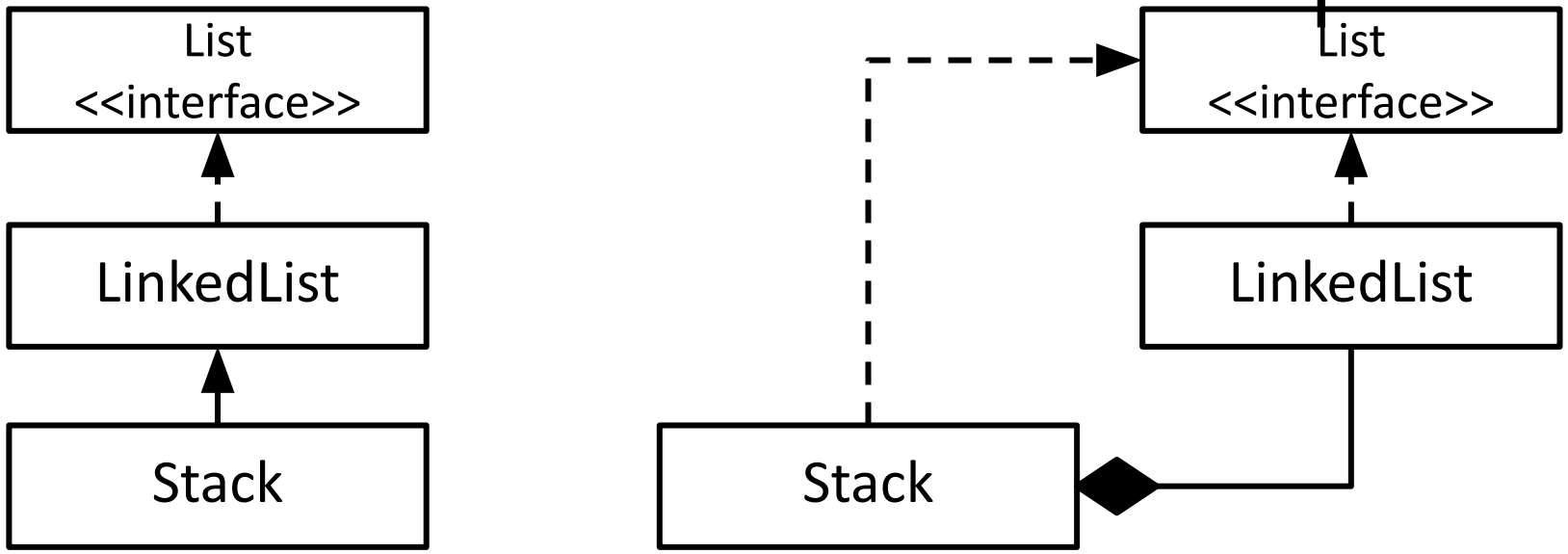



```
public class Stack<E>
    extends
LinkedList<E> {
  ...
}
```

```
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.

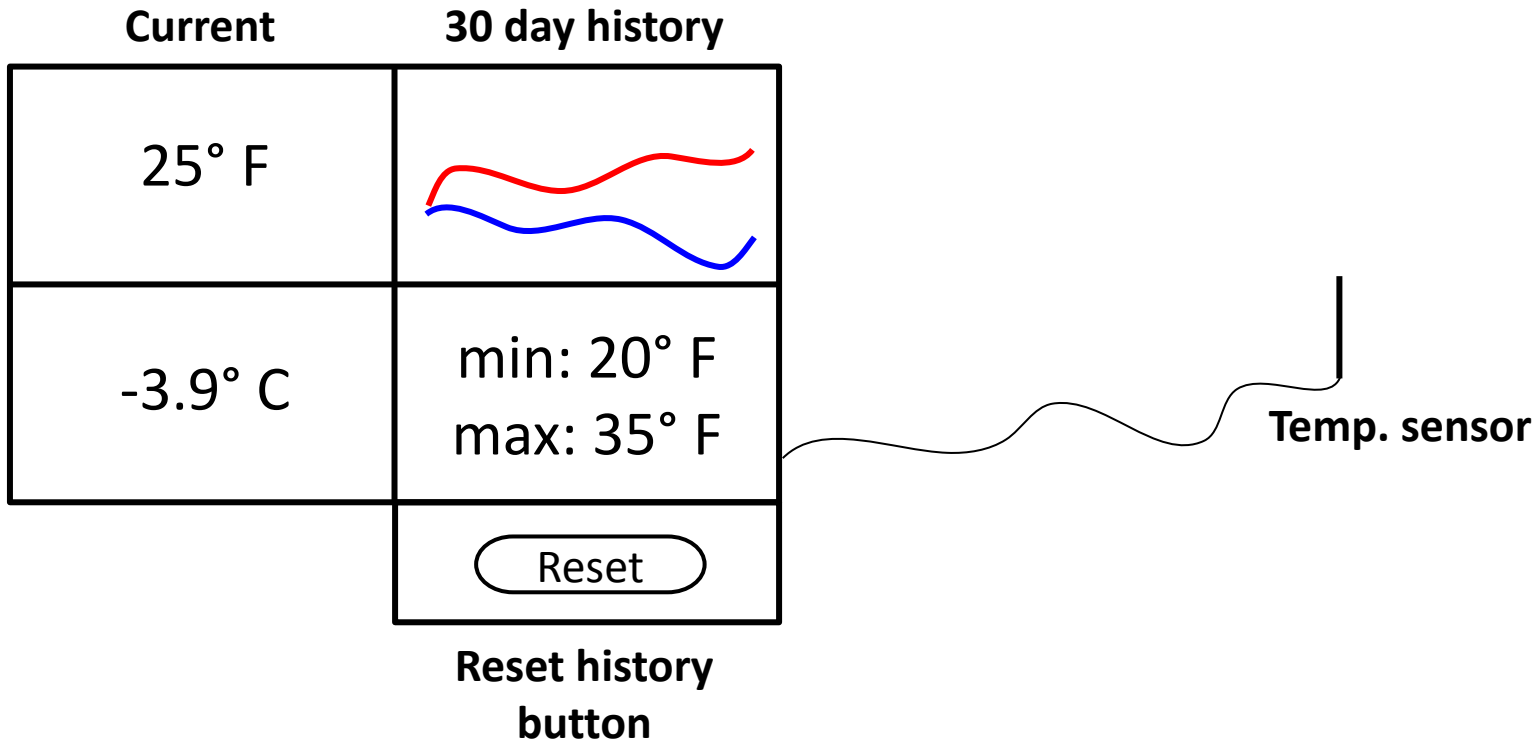
OO design principles: summary

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

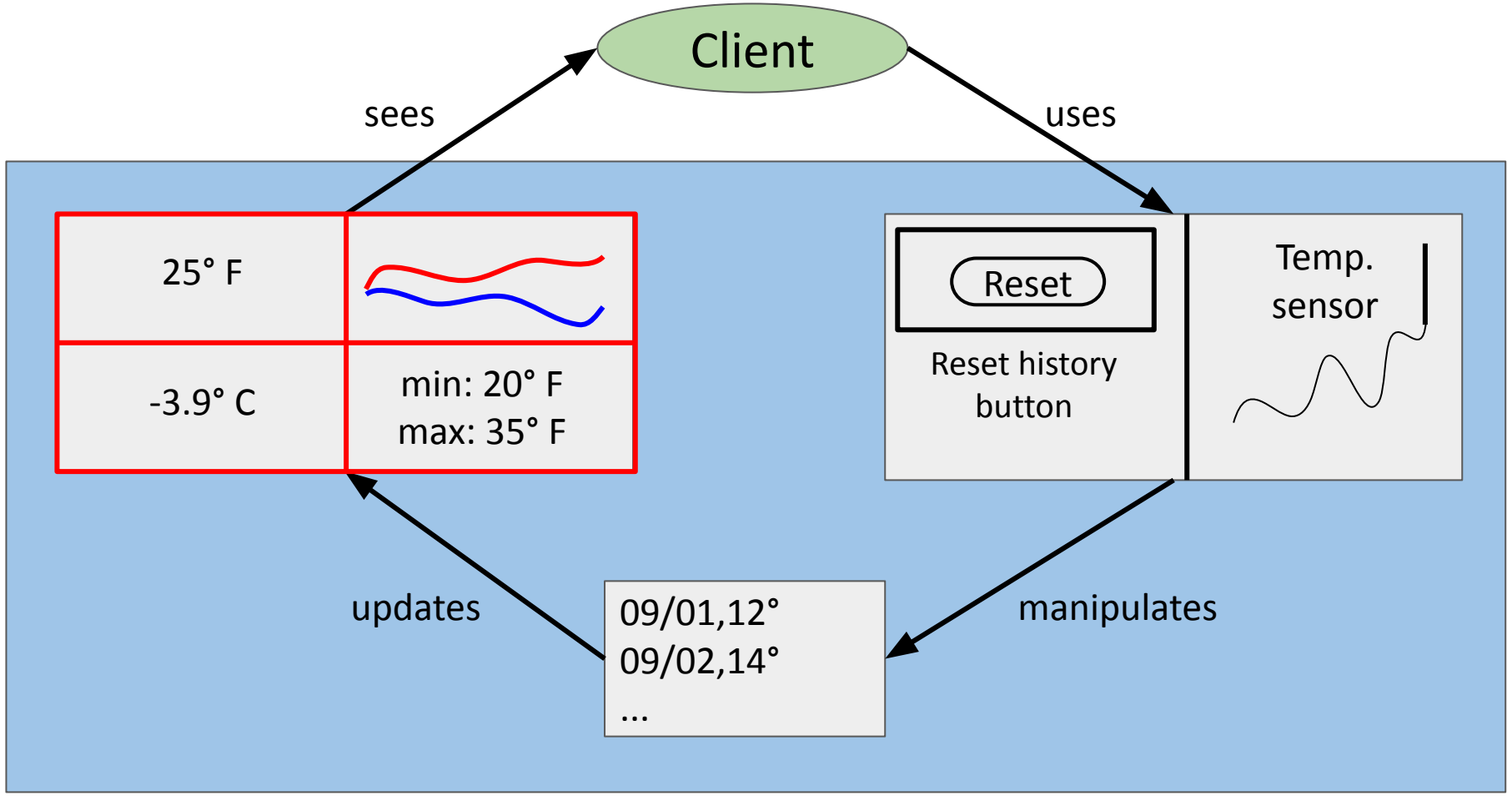
OO design patterns

A first design problem

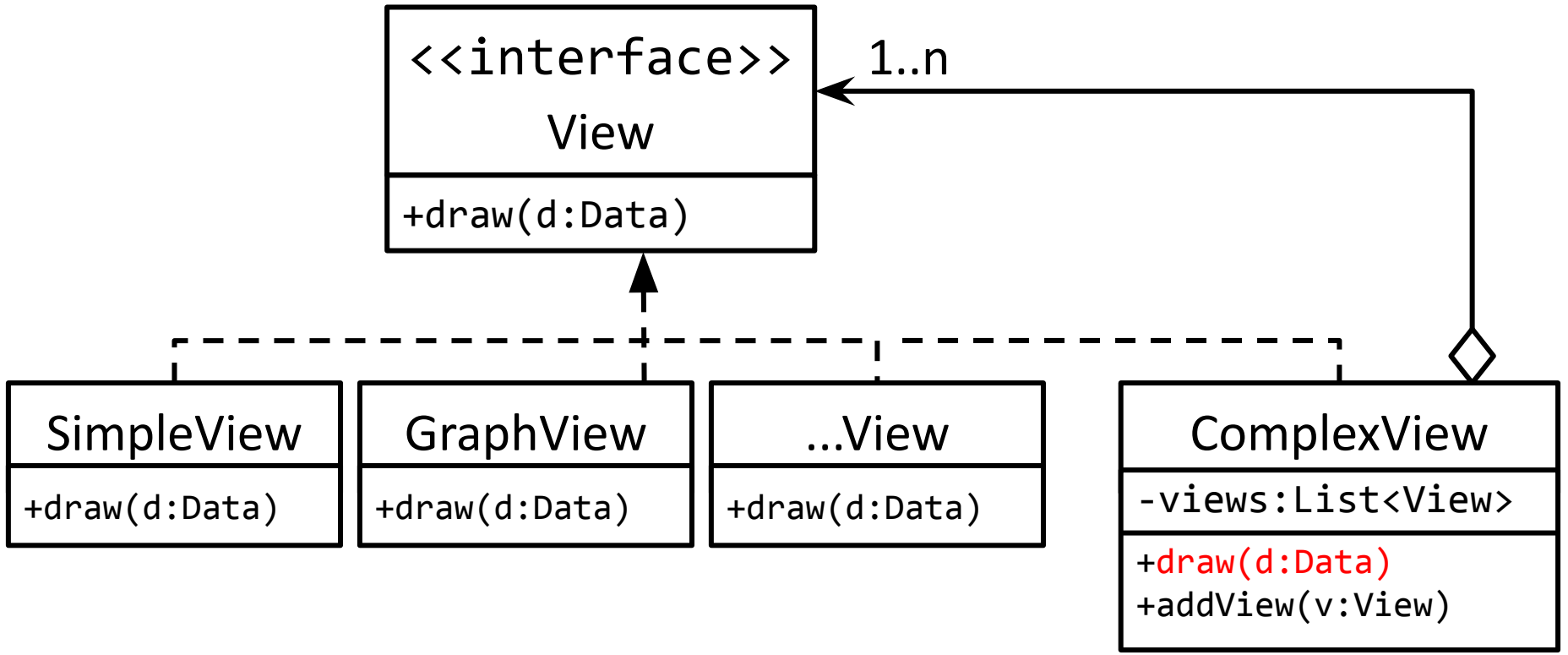
Weather station revisited

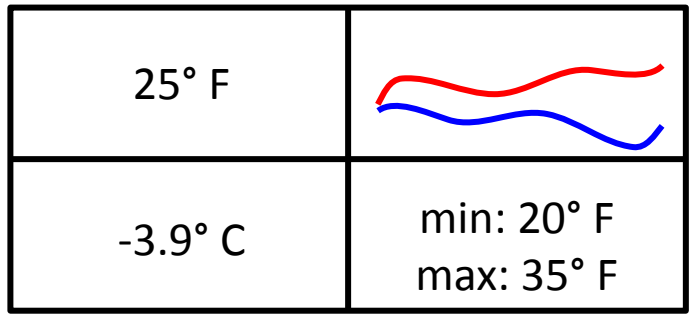


What's a good design for the view component?



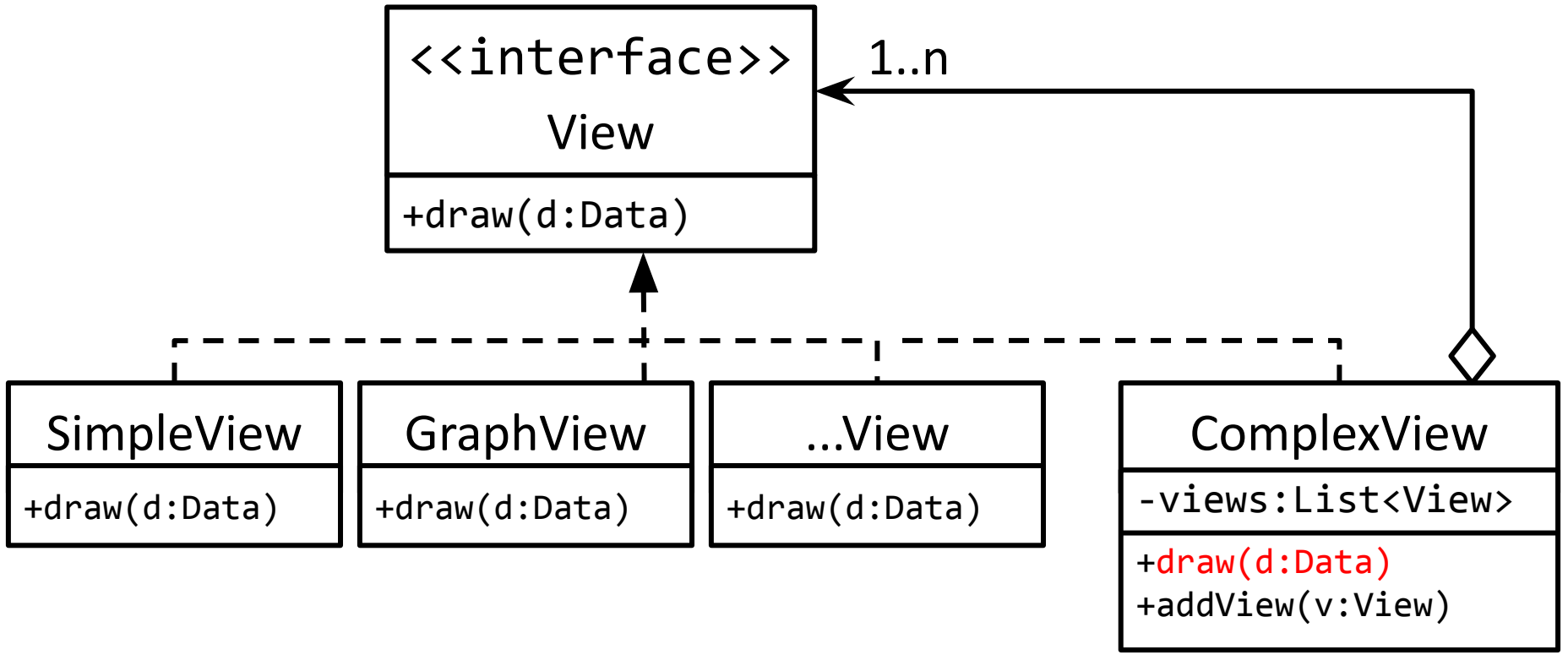
Weather station: view

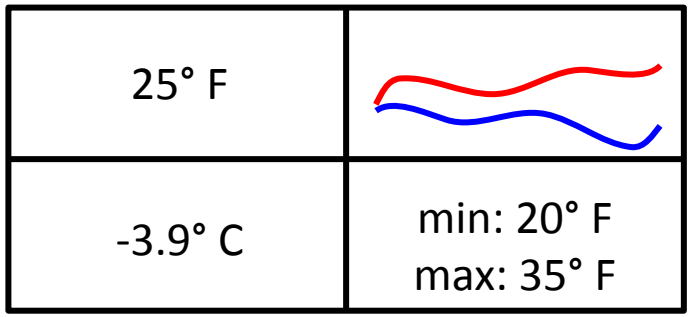




How do we need to implement draw(d:Data)?

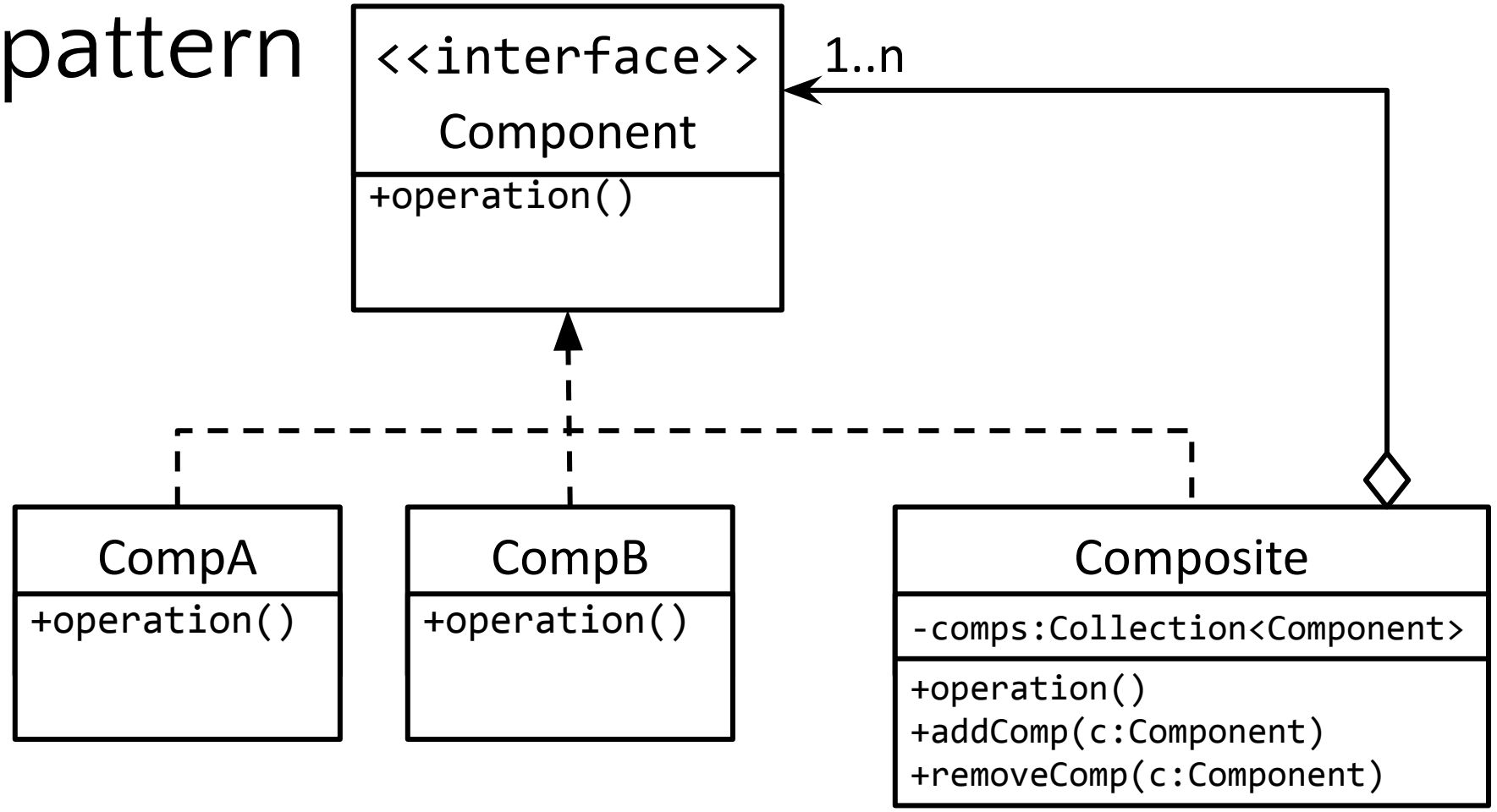
Weather station: view



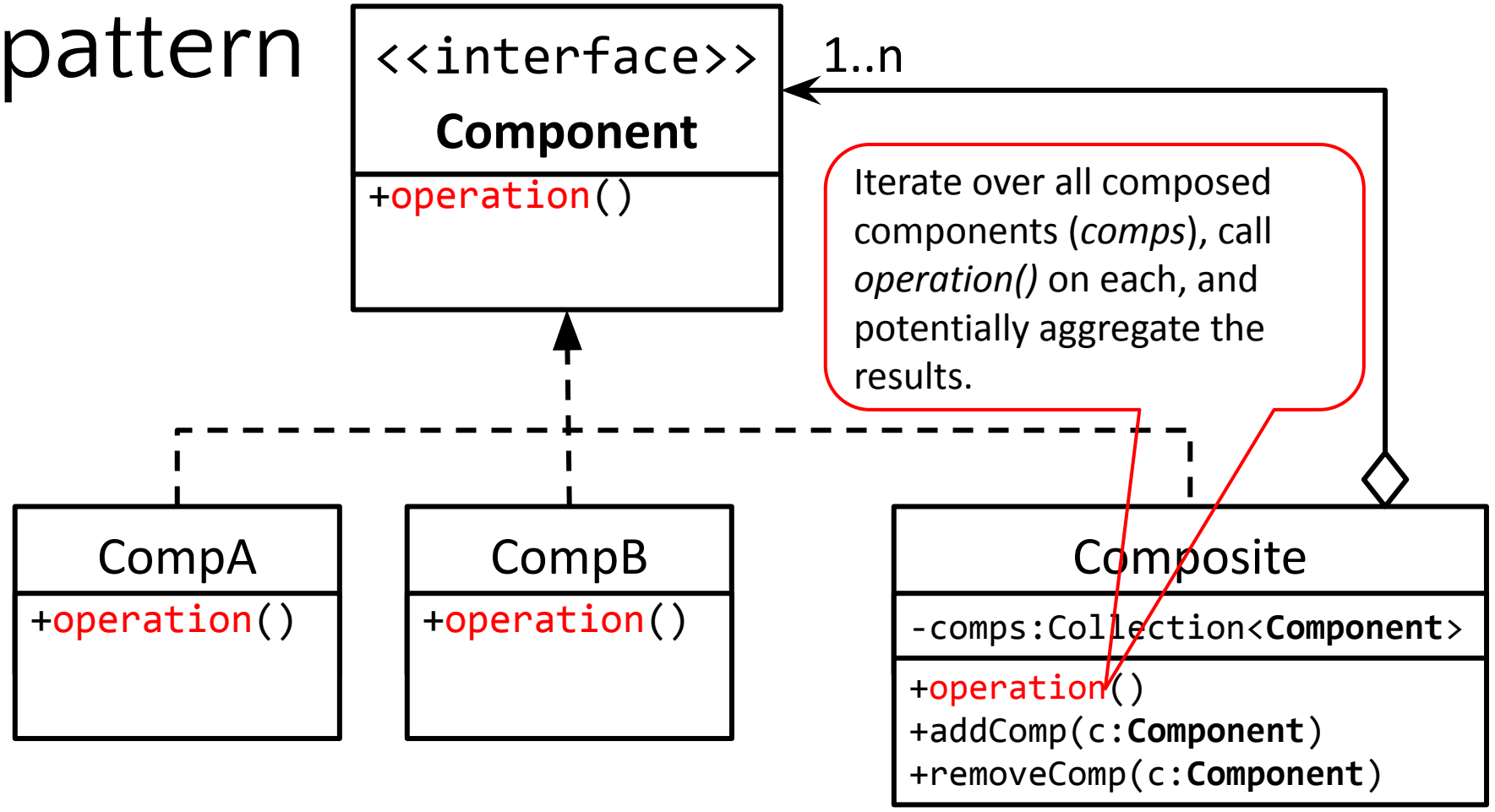



```
public void draw(Data d) {
  for (View v : views) {
    v.draw(d);
  }
}
```

The general solution: Composite pattern



The general solution: Composite pattern



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
    - ...
1. Behavioral
    - Template method
    - Visitor
    - ...
1. Creational
    - Singleton
    - Factory (method)
    - ...

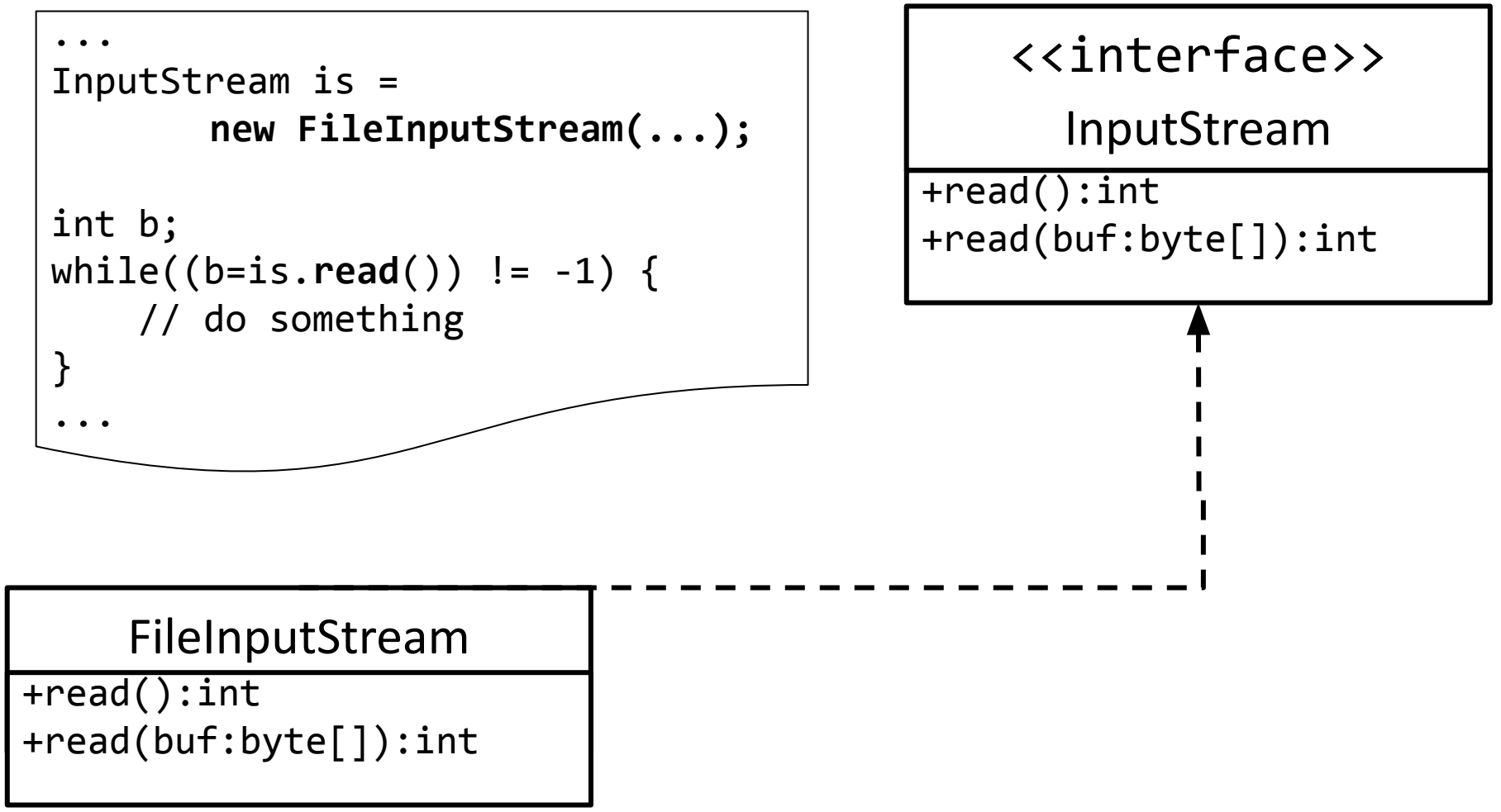
Design patterns: categories

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

Another design problem: I/O streams

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

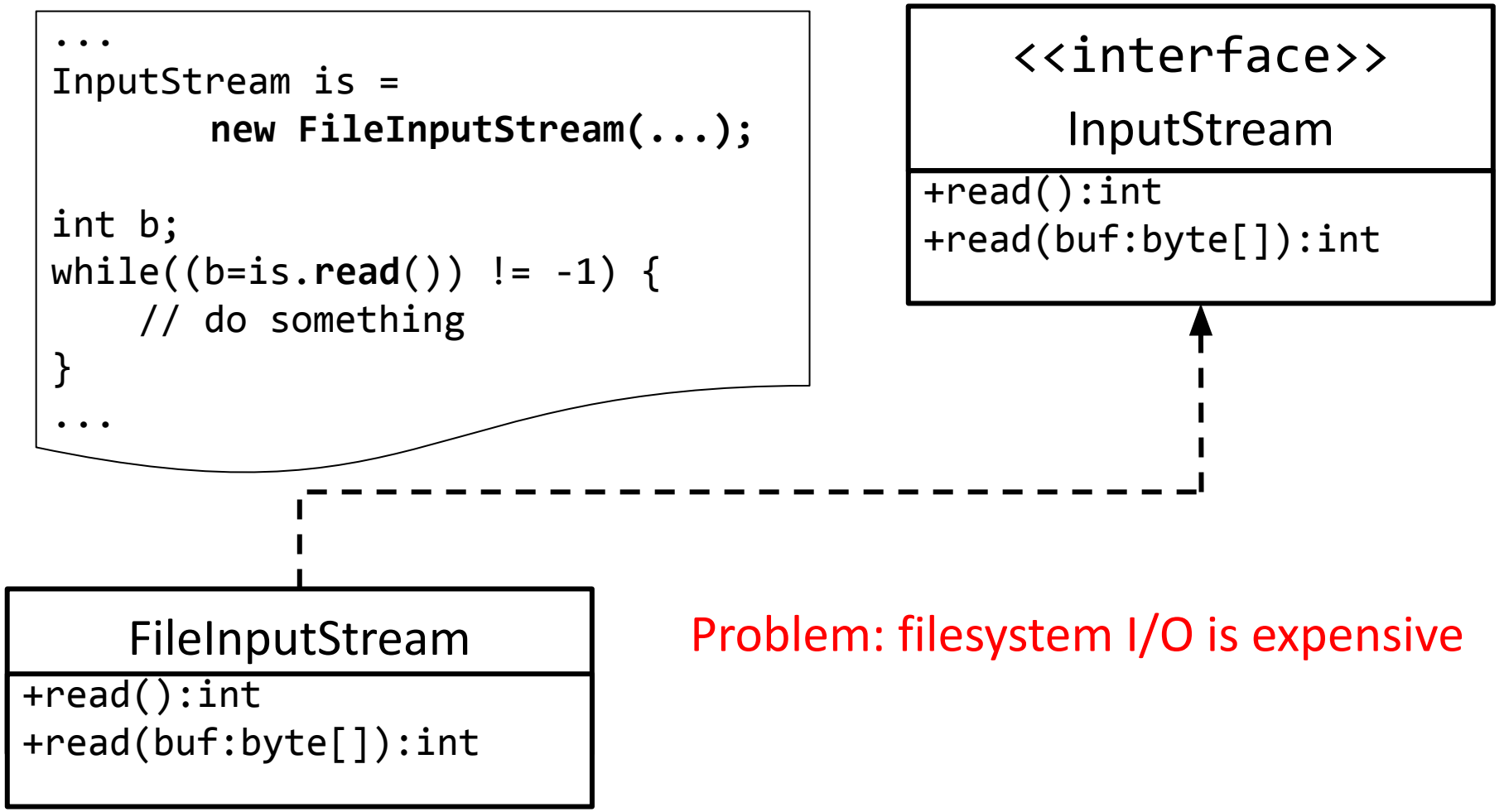
int b;
while((b=is.read()) != -1) {
    // do something
}
...
```



Another design problem: I/O streams

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

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

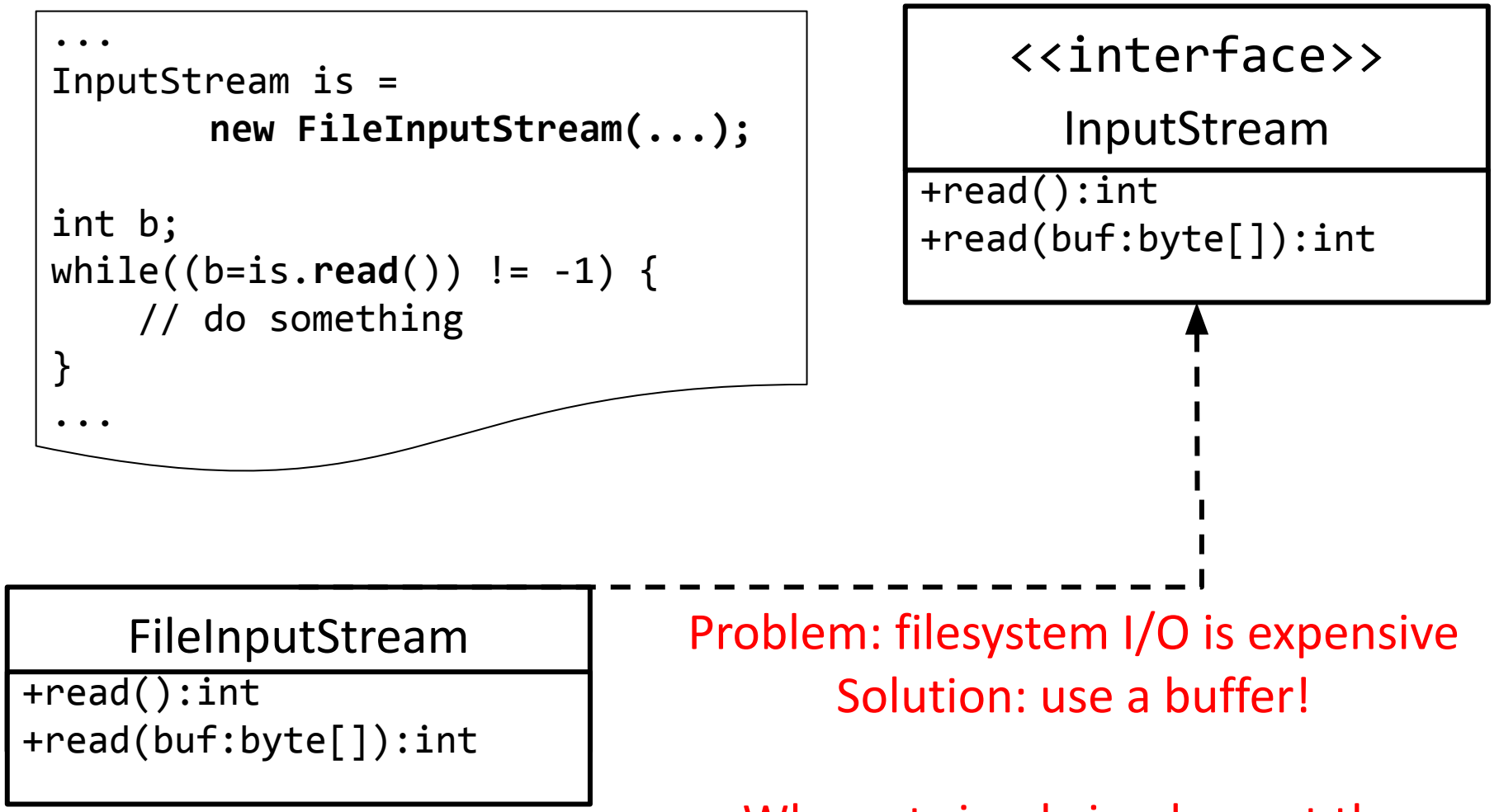


Problem: filesystem I/O is expensive

Another design problem: I/O streams

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

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

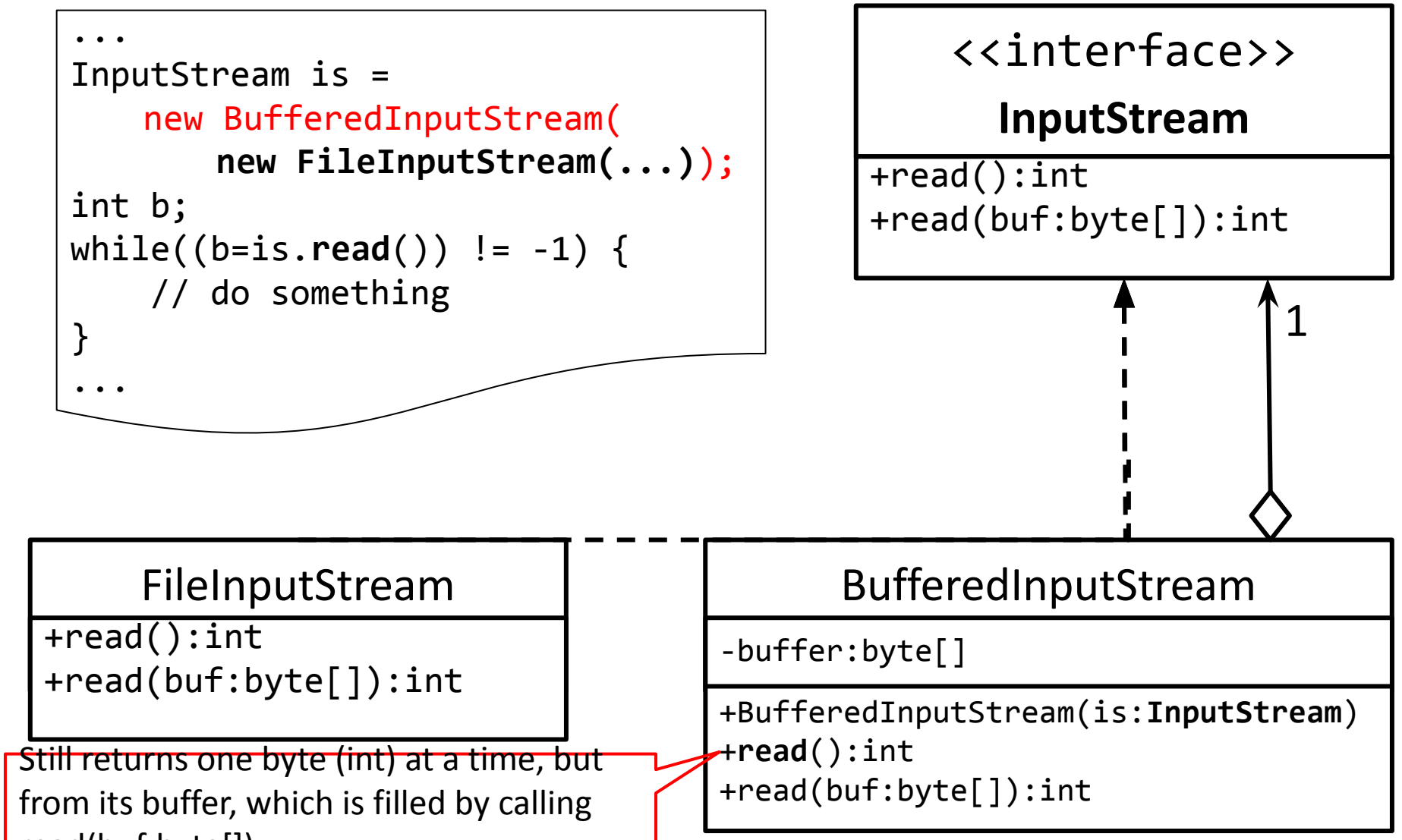


Problem: filesystem I/O is expensive
Solution: use a buffer!

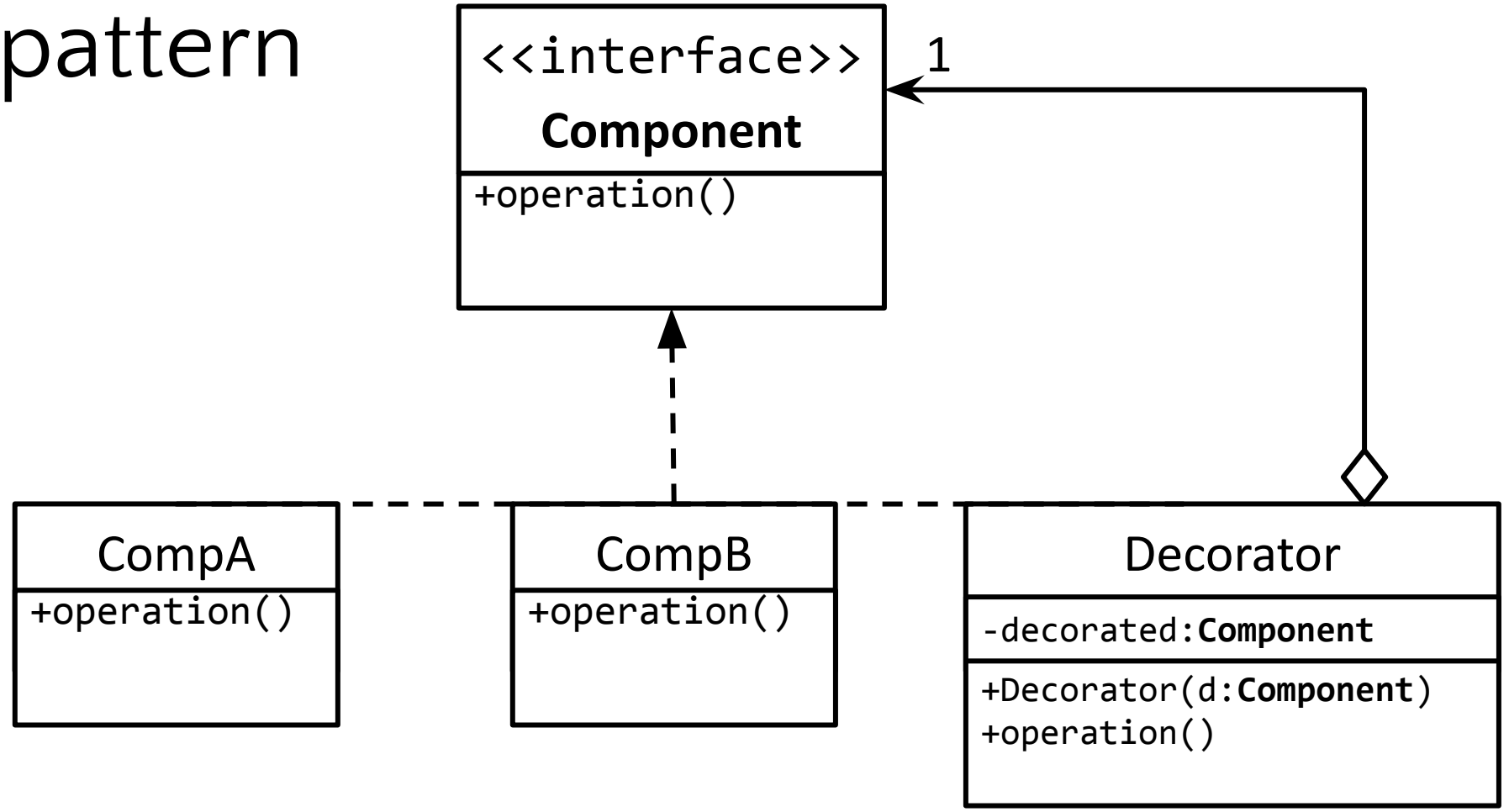
Why not simply implement the buffering in the client or subclass?

Another design problem: I/O streams

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



The general solution: Decorator pattern



Composite vs. Decorator