Software Design

CSE 403 Software Engineering

Winter 2025

Project requirements and planning tips

Celebrate your brand – use your product name as a title in your doc

Ensure the **beta release milestone** is in your schedule

Plan for continuous testing (and continuous delivery after beta)

Today’s Outline

1. Quick recap – Architecture vs Design
2. Some practical design considerations
3. Class quiz on coding styles in PollEv ☺

Readings are posted on the Calendar

See also Appendix for a short primer on design material

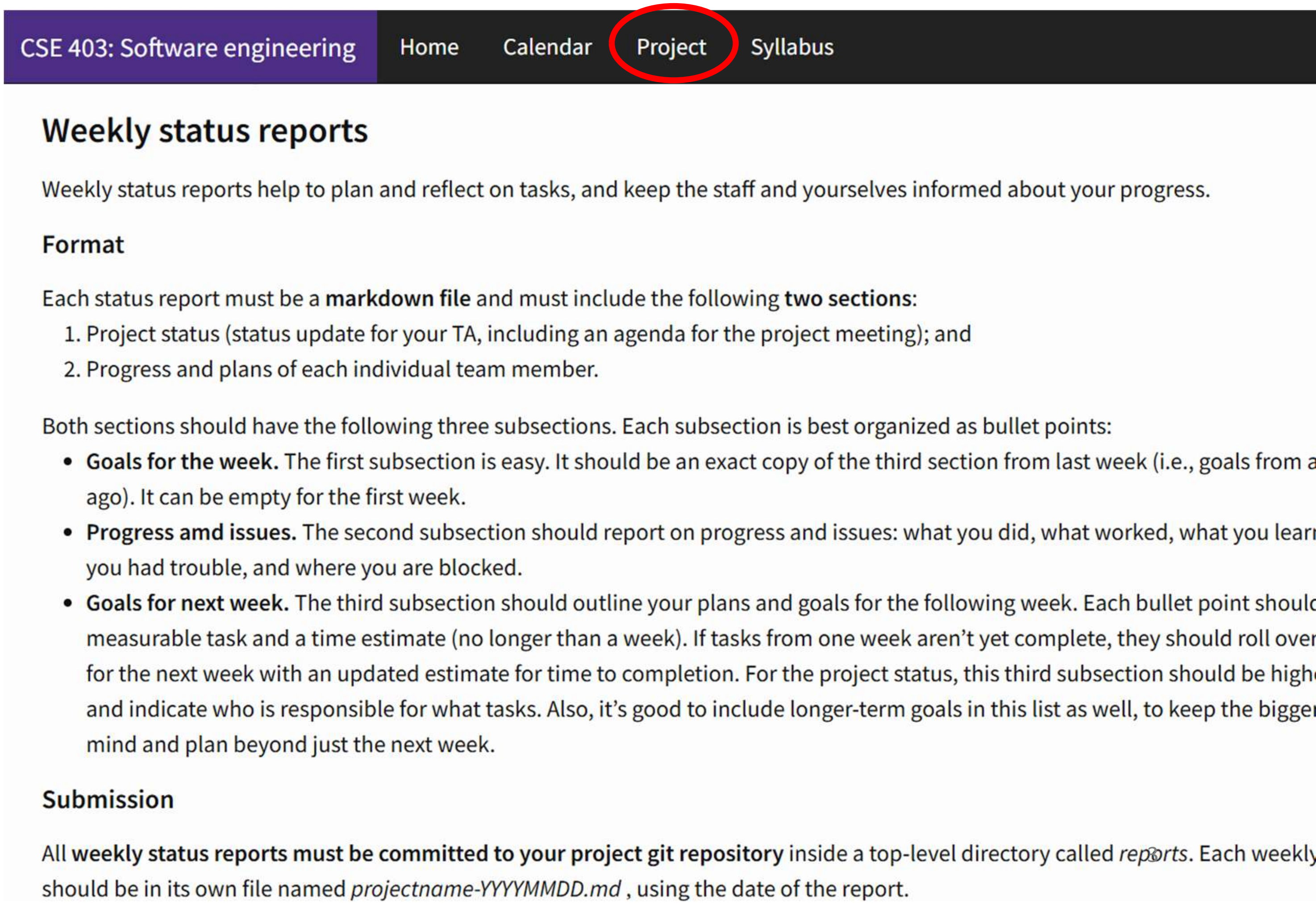
- Visualizing your design with UML (unified modeling language)
- Design principles
- Design patterns

Weekly status reports are underway

Due in github each Wednesday 11:59pm

Include an agenda for Thurs project meeting and any questions for staff

Details on "Project" tab of class website



Weekly status reports

Weekly status reports help to plan and reflect on tasks, and keep the staff and yourselves informed about your progress.

Format

Each status report must be a **markdown file** and must include the following **two sections**:

1. Project status (status update for your TA, including an agenda for the project meeting); and
2. Progress and plans of each individual team member.

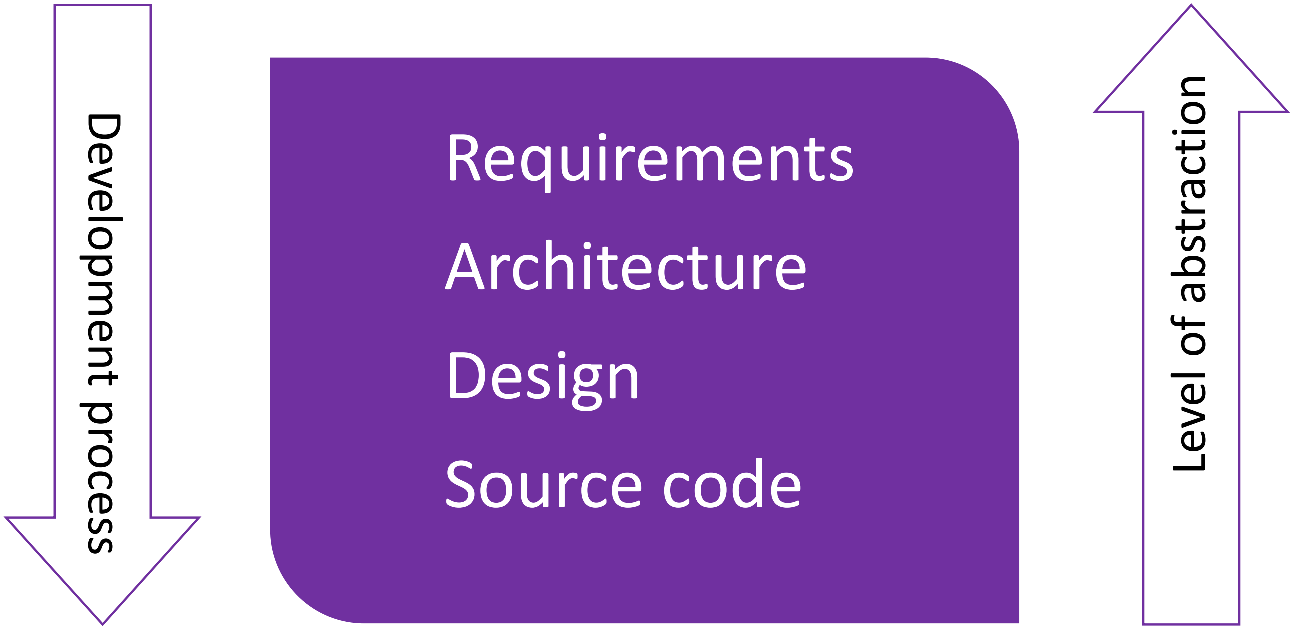
Both sections should have the following three subsections. Each subsection is best organized as bullet points:

- **Goals for the week.** The first subsection is easy. It should be an exact copy of the third section from last week (i.e., goals from a ago). It can be empty for the first week.
- **Progress amd issues.** The second subsection should report on progress and issues: what you did, what worked, what you lear you had trouble, and where you are blocked.
- **Goals for next week.** The third subsection should outline your plans and goals for the following week. Each bullet point shoul measurable task and a time estimate (no longer than a week). If tasks from one week aren't yet complete, they should roll over for the next week with an updated estimate for time to completion. For the project status, this third subsection should be high and indicate who is responsible for what tasks. Also, it's good to include longer-term goals in this list as well, to keep the bigge mind and plan beyond just the next week.

Submission

All **weekly status reports must be committed to your project git repository** inside a top-level directory called *reports*. Each weekly should be in its own file named *projectname-YYYYMMDD.md* , using the date of the report.

High level overview from last class



Development process

Requirements

Architecture

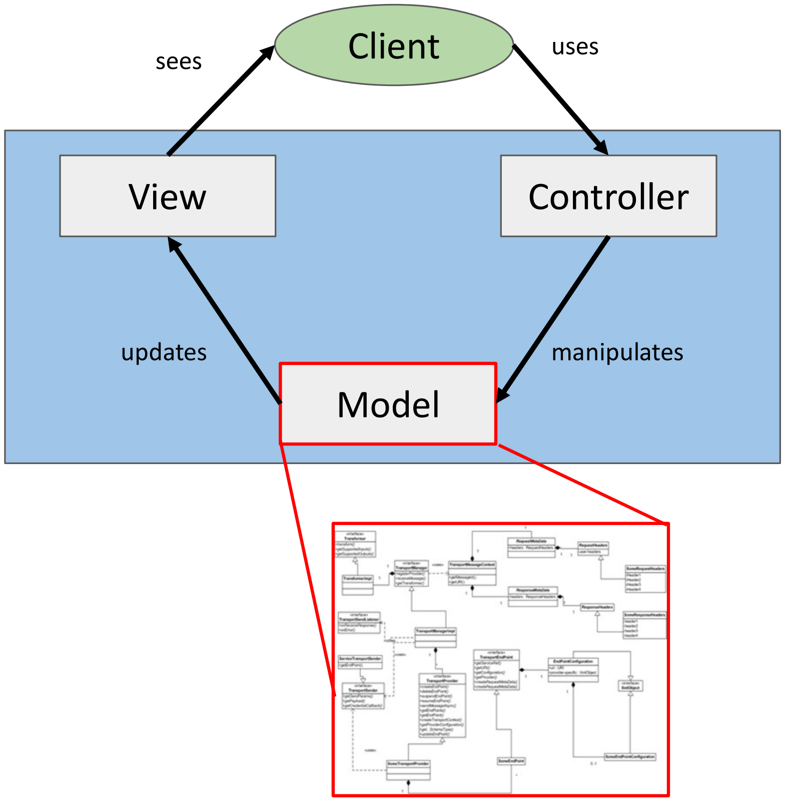
Design

Source code

Level of abstraction

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 themselves are developed



Let’s look at the SOLID design principles

- 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.“
- The Liskov substitution principle: “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 (implement) interfaces that they do not use.“
- The Dependency inversion principle: "Depend upon abstractions, [not] concretes."

Learn more: https://en.wikipedia.org/wiki/SOLID

There are other tried-and-true principles

- KISS principle (keep it simple, stupid)
- YAGNI principle (you ain’t gonna need it)
- DRY principle (don’t repeat yourself)
- Single responsibility (focus on on doing one thing well – high cohesion)
- Open/closed principle (open for extension, closed for modification)
- Liskov substitution principle (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)

Learn more: Geeks for Geeks Design Principles

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 principles and properties …check

How about good patterns to learn and model from

Design patterns

- Tried and true solutions to commonly occurring problems in software design
- Models / blueprints that you can leverage or customize to solve design problems in your code
- Address recurring, common design problems and provide generalizable solutions – models – that you can customize
- Provide a common terminology for developers
- Creational, structural and behavioral patterns

Creational design patterns

- Focus on the process of object creation and problems/complexity related to object creation
- Help in making a system independent of how its objects are created, composed and represented

- Example: Simple Factory pattern
  - Scenario: want to hide all the instantiation logic from the client
  - Simple Factory pattern: provides a clean way to generate an instance for a client without exposing instantiation logic to the client

```
interface Door {
        public function getWidth(): float;
        public function getHeight(): float;
}

class WoodenDoor implements Door {
        protected $width;
        protected $height;

        public function _construct(float $width,
                                  float $height){
        $this->width = $width;
        $this->height = $height;
        }

        public function getWidth(): float {
        return $this->width;
        }

        public function getHeight(): float {
        return $this->height;
}
```

```
class DoorFactory {

   public static function makeDoor($width, $height): Door
   {
        return new WoodenDoor($width, $height);
    }

}
```

```
$door1 = DoorFactory::makeDoor(100, 200);

$door2 = DoorFactory::makeDoor(50, 100);
```

Example from:
https://github.com/kamranahmedse/design-patterns-for-humans

Structural design patterns

- Solve problems related to how classes and objects are composed to form larger structures that are efficient and flexible
- Often use inheritance to compose interfaces or implementations

- Example: Fascade pattern
  - English definition: an outward appearance that is maintained to conceal a less pleasant reality
  - Scenario: provide a simple interface to a complex subsystem
  - Fascade pattern: a facade is an object that provides a simplified interface to a larger body of code

```
class Computer {
            public function getElectricShock() {..}
            public function makeSound() {..}
            public function showLoadingScreen() {..}
            public function bam() {..}
            public function closeEverything() {..}
            public function sooth() {..}
            public function pullCurrent() {..}
}
```

```
$computer = new ComputerFacade (new Computer());
$computer->turnOn();
$computer->turnOff()
```

```
class ComputerFacade {
  protected $computer;

  public function __construct (Computer $computer) {
            $this->computer = $computer;
  }
  public function turnOn() {
            $this->computer->getElectricShock();
            $this-computer->makeSound();
            $this->computer->showLoadingScreen();
            $this->computer->bam();
  }
  public function turnOff() {
            $this->compute->closeEverything();
            $this->computer->pullCurrent();
            $this->computer->sooth():
  }
```

Example from:
https://github.com/kamranahmedse/design-patterns-for-humans

Behavioral design patterns

- Solve problems related to responsibilities and communication between objects
- Describe not just patterns of objects or classes but also the patterns of communication between them
- Identify common communication patterns between objects and realize these patterns

- Example: Mediator pattern
  - Scenario: want to minimize/avoid direct complex dependencies between objects (strive for loose coupling), and/or have centralized coordination

```
interface Airplane {
    void requestTakeoff();
    void requestLanding();
    void notifyAirTrafficControl(String message);
}

class CommercialAirplane implements Airplane {
    private AirTrafficControlTower mediator;

    public CommercialAirplane(AirTrafficControlTower
        mediator) {
        this.mediator = mediator;
    }
    public void requestTakeoff() {
         mediator.requestTakeoff (this);
     }
    ...
}
```

```
interface AirTrafficControlTower {   // Mediator
    void requestTakeoff(Airplane airplane);
    void requestLanding(Airplane airplane);
}

class AirportControlTower implements AirTrafficControlTower {
       public void requestTakeoff(Airplane airplane) {
        //
         // Complex logic for coordinating takeoff
         //
        airplane.notifyAirTrafficControl("Requesting takeoff
                                                                        clearance.");
        }
        ...
}
```

```
AirTrafficControlTower controlTower = new AirportControlTower();
Airplane airplane1 = new CommercialAirplane(controlTower);
Airplane airplane2 = new CommercialAirplane(controlTower);
airplane1.requestTakeoff();
airplane2.requestLanding();
```

Example from:
https://www.geeksforgeeks.org/mediator-design-pattern/

Like most things, design patterns have pros and cons

Pros

- Provide a common language for developers (including interviewing)
- Can improve communication and documentation
- “Toolbox” for devs to leverage known solutions to a known problems (don’t reinvent the wheel)

Cons

- Can get swept into thinking a pattern fits when it does not
- Or using one when there is a better – built in – solution in the language or dev toolkit that you’re using
- Can add complexity when it’s not needed

Some good design patterns references

- **https://github.com/kamranahmedse/design-patterns-for-humans < nice overview with examples**

- https://www.patterns.dev < Java, React, Next.js, Vue.js examples

- https://refactoring.guru/design-patterns/catalog < some motivating examples

- https://www.geeksforgeeks.org/software-design-patterns/ < tutorial like with examples

Let’s look at code
and
assess its style

Many thanks to René Just, UW CSE Prof

Quiz setup

https: //PollEv.com/cse403wi

- Work in small groups of neighboring students
- **Individually register your answer in PollEv**
- 6 code snippets

- Round 1 (PollEv)
  - For each code snippet, decide if it represents good or bad practice
  - Discuss and reach consensus on good or bad practice and why

- Round 2 (Poll results and class discussion)
  - For each code snippet, share opinions on why it is good or bad practice
  - **Goal:** common understanding of good styles and alternatives to bad ones

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?

Design Quiz - Good or bad?

0 surveys completed

0 surveys underway

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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;
   }
}
```

And the survey says …

Snippet1: getAllLogs

Good

Bad

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

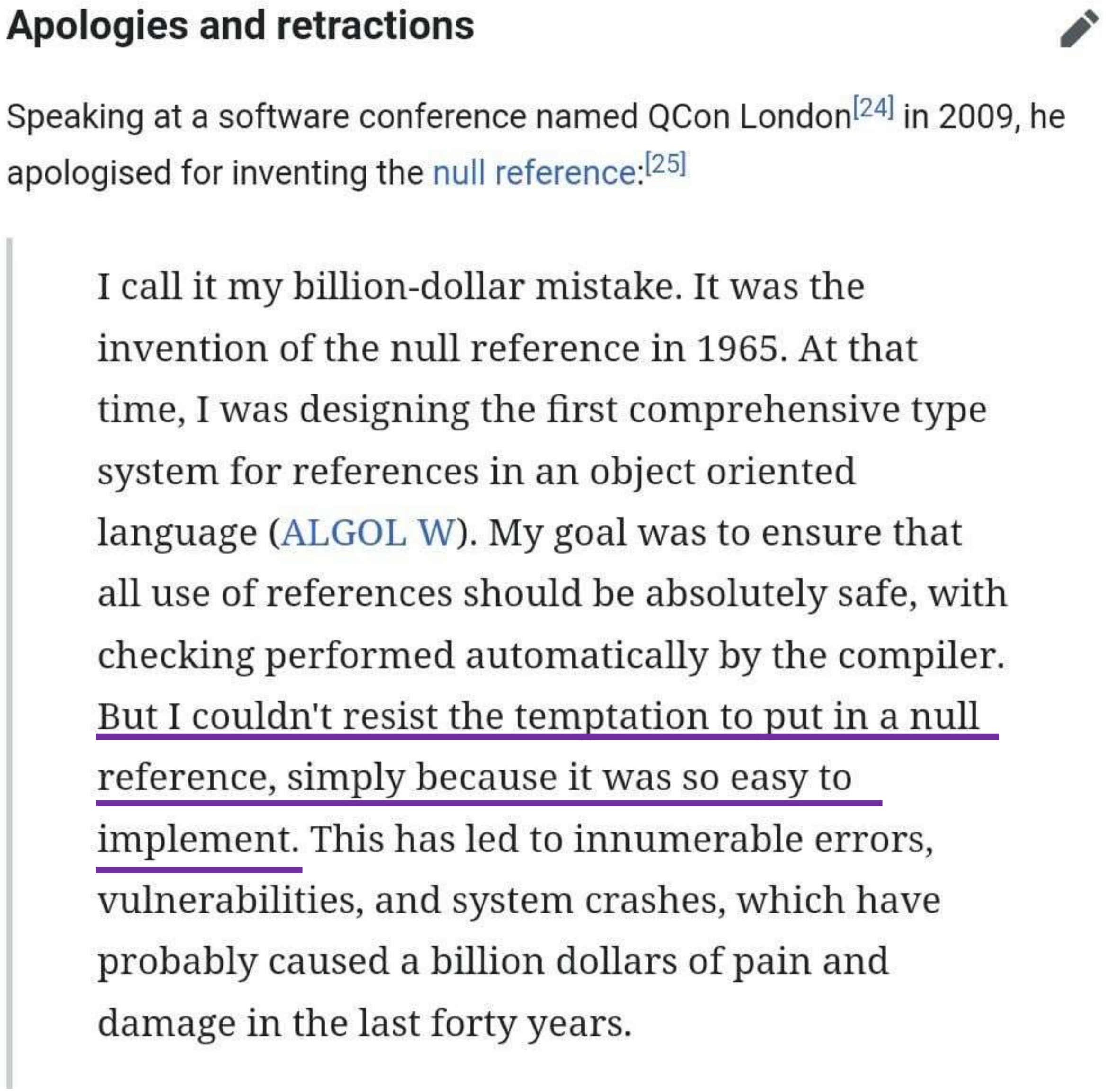
Snippet 1: this is bad! why?

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

Snippet 1: this is bad! why?

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

Null references...the billion dollar mistake.



Apologies and retractions

Speaking at a software conference named QCon London[24] in 2009, he apologised for inventing the null reference:[25]

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Tony Hoare

- Programming languages
- Concurrent programming
- Quicksort

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)
}
```

And the survey says …

Snippet2: addStudent

Good

Bad

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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)
}
```

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");
   }
}
```

And the survey says …

Snippet3: PaymentType

Good

Bad

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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);
}
```

And the survey says …

Snippet4: getAbsMax

Good

Bad

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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);
}
```

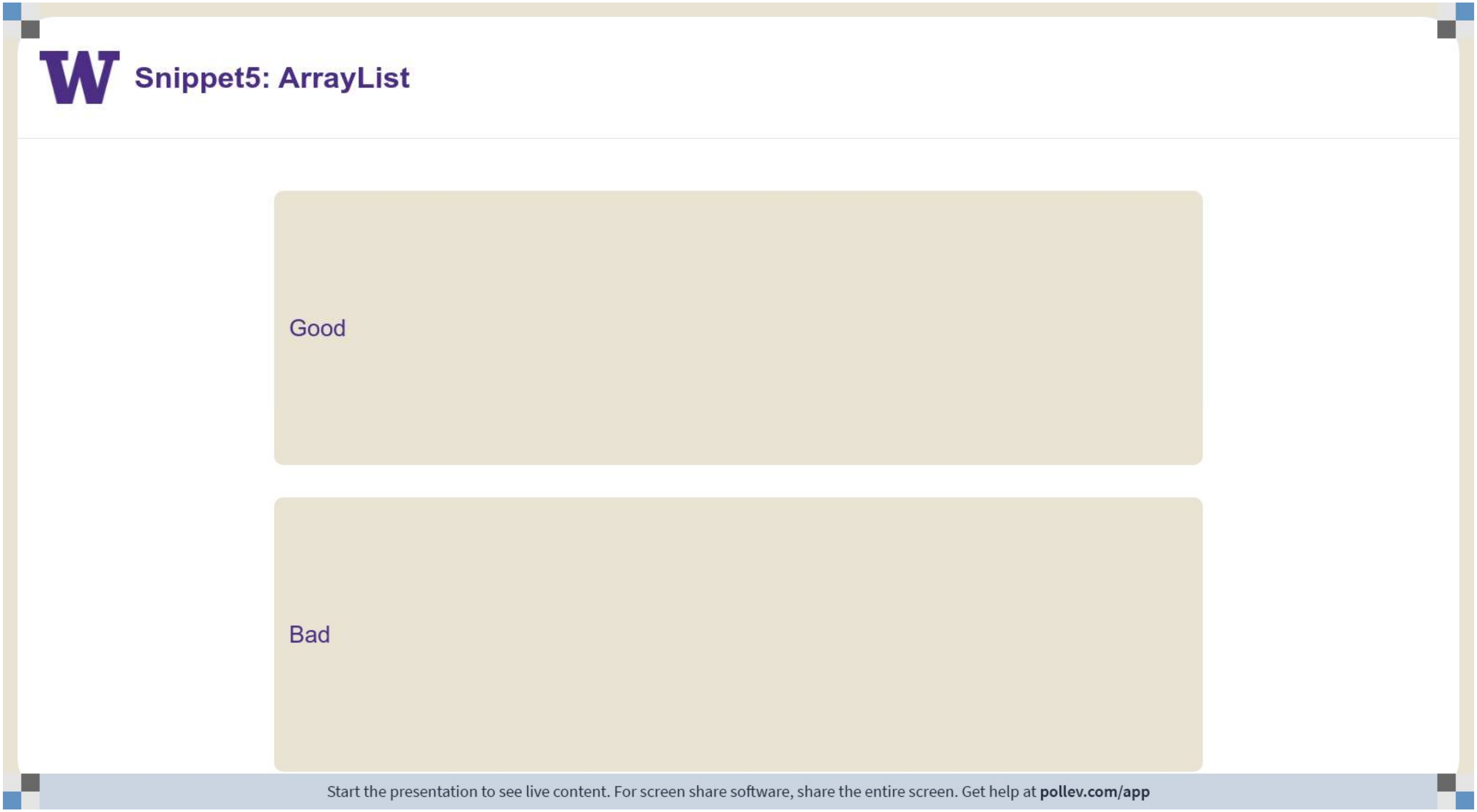
Consider if these are pass by reference…

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) {
      …
   }
   …
}
```

And the survey says …



Snippet5: ArrayList

Good

Bad

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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 overloading with different return values.

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 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;
   }
}
```

And the survey says ...

Snippet6: Point

Good

Bad

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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 light look at **UI design** later in the course – it’s a course in itself, CSE 440 – Intro to HCI
- We may also look at some more **design patterns,** time permitting
- Review the readings on the Calendar and the design primer in the following slides to refresh your knowledge of design considerations for your project and **03 Architecture and design milestone**

Additional Design Material

Provided by René Just, UW CSE Professor

Concepts traditionally covered in CSE 331 – Software design and implementation

UML crash course

UML crash course

The main questions

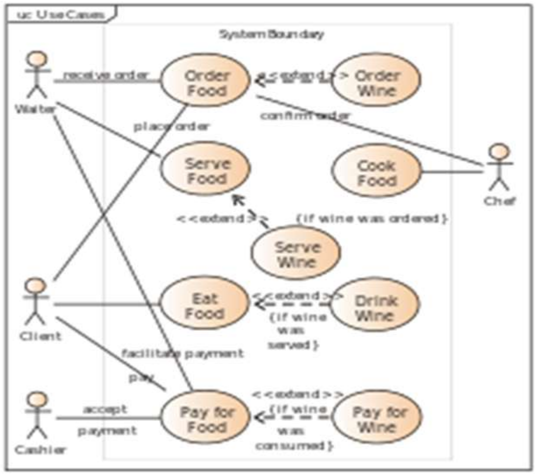
- What is UML?
- Is it useful, why bother?
- When to (not) use UML?

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
  - …

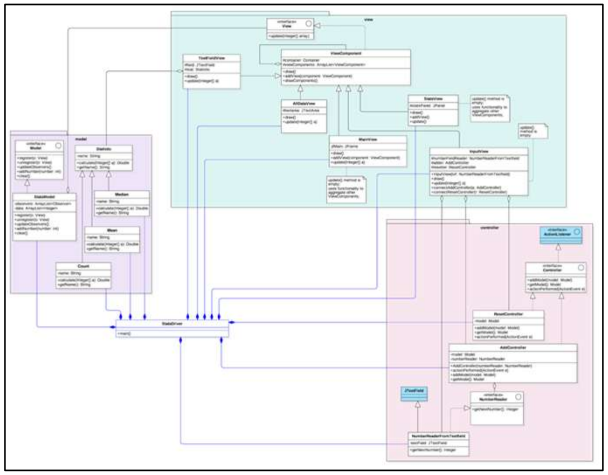
What is UML?

- 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
  - …

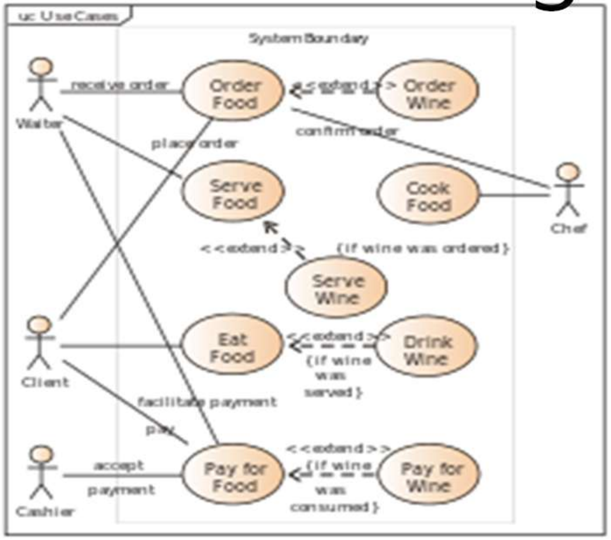


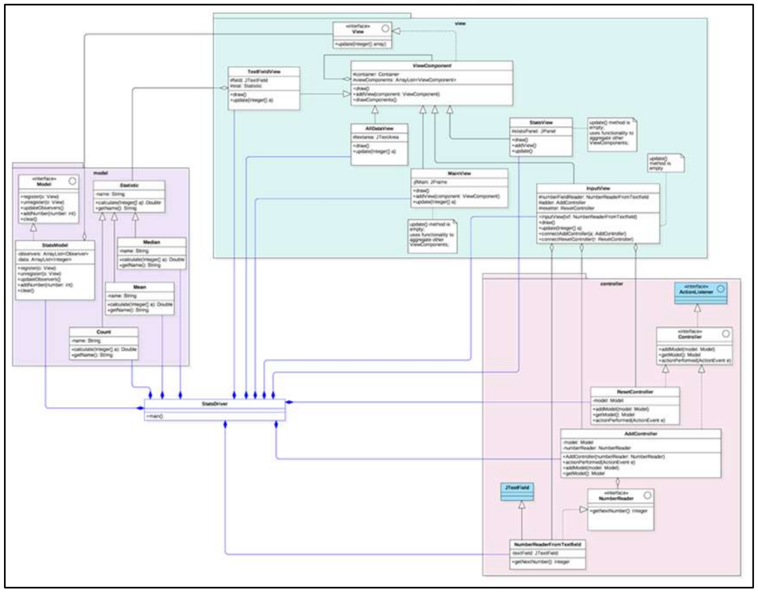
What is UML?

- 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 usefu





Are UML diagrams useful?

Communication

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

Documentation

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

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

Classes vs. objects

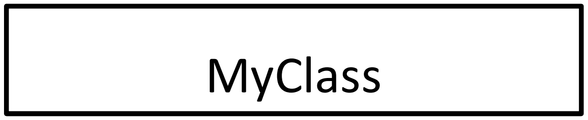
Class

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

Object

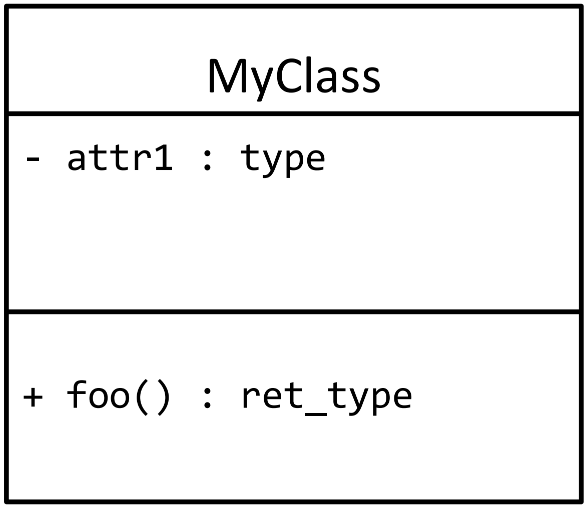
- Entity from the real world.
- Instance of a class
  - Student: Joe (4711), Jane (4712), …
  - Car: Audi A6, Honda Civic, …

UML class diagram: basic notation



MyClass

UML class diagram: basic notation



MyClass

- attr1 : type

+ foo() : ret_type

Name

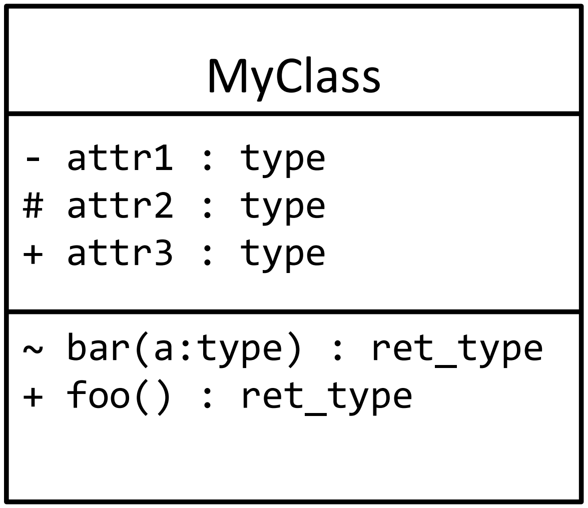
Attributes

<visibility> <name> : <type>

Methods

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

UML class diagram: basic notation



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

Name

Attributes

<visibility> <name> : <type>

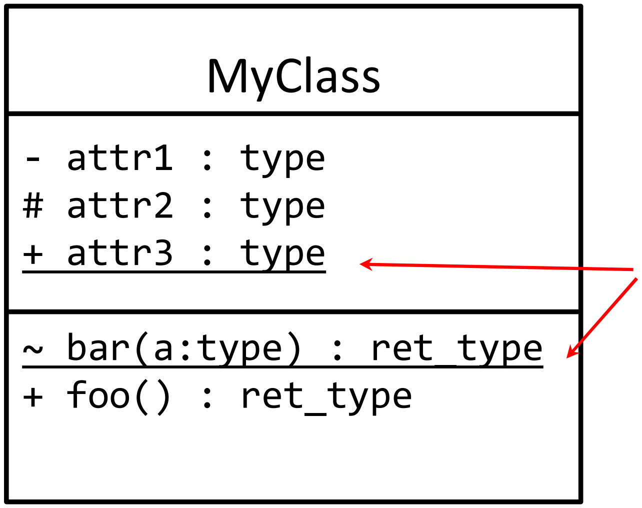
Methods

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

Visibility

- private
~ package-private
# protected
+ public

UML class diagram: basic notation



MyClass

```
- attr1 : type
# attr2 : type
+ attr3 : type
```

```
~ bar(a:type) : ret_type
+ foo() : ret_type
```

Name

Attributes

<visibility> <name> : <type>

Static attributes or methods are underlined

Methods

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

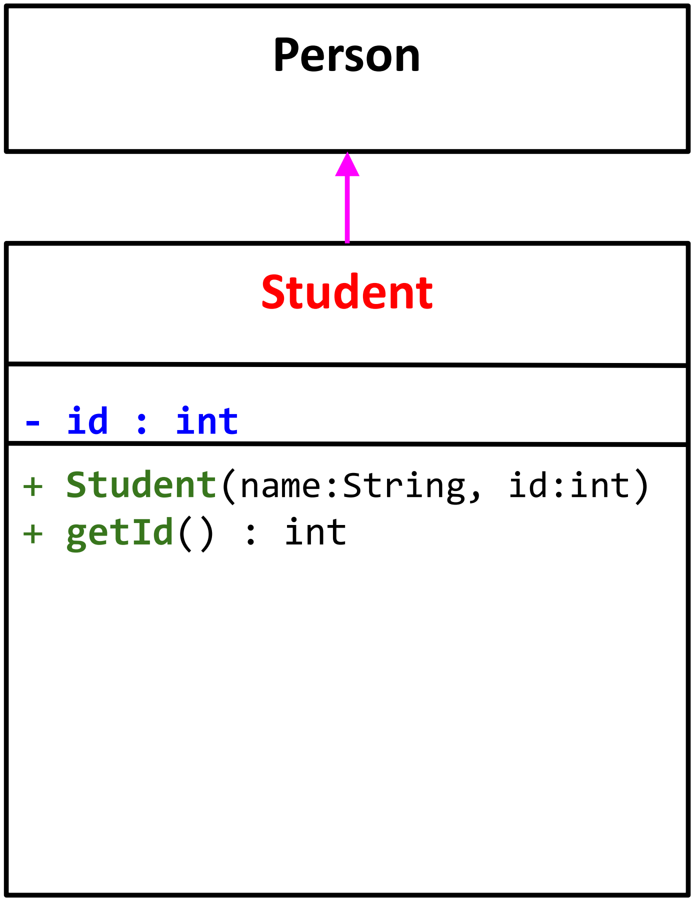
Visibility

- private
~ package-private
# protected
+ public

UML class diagram: concrete example

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

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

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



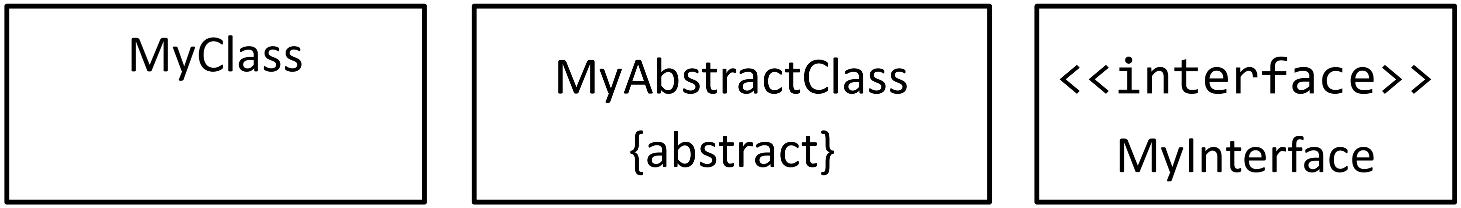
Person

Student

- id : int

+ Student(name:String, id:int)
+ getId() : int

Classes, abstract classes, and interfaces

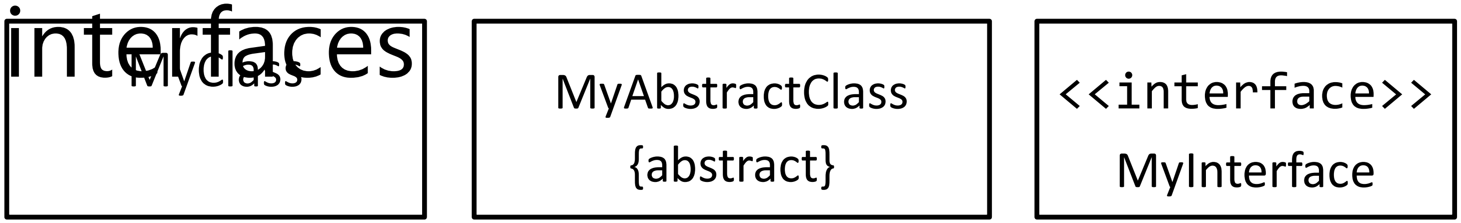


MyClass

MyAbstractClass
{abstract}

`<<interface>>`
MyInterface

Classes, abstract classes, and interfaces



MyClass

MyAbstractClass
{abstract}

<<interface>>
MyInterface

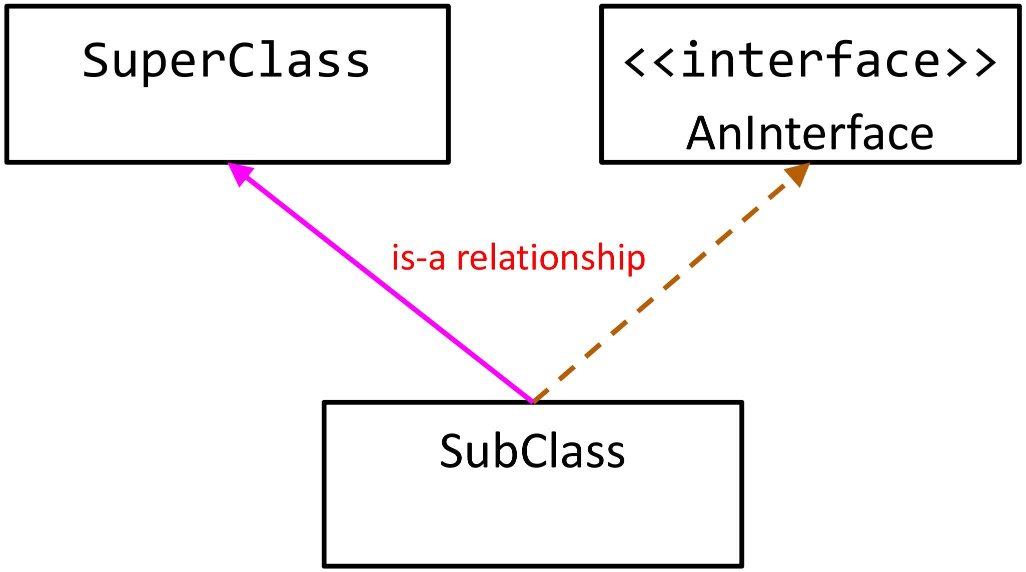
```
public class
MyClass {


  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.

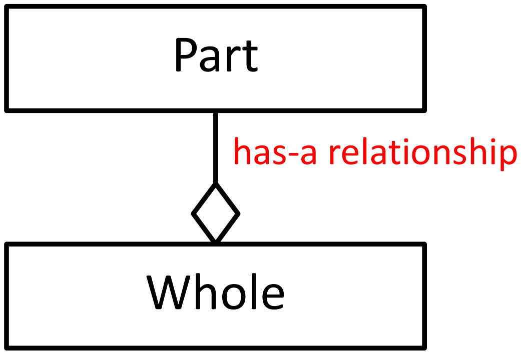
UML class diagram: Inheritance



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

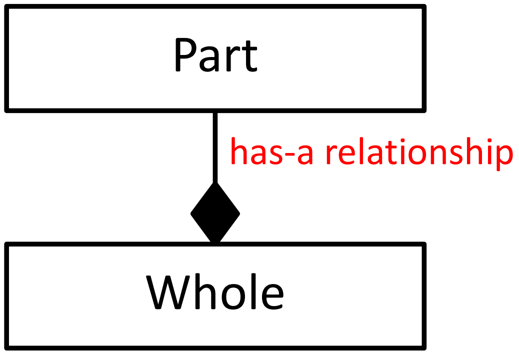
UML class diagram: Aggregation andComposition

Aggregation



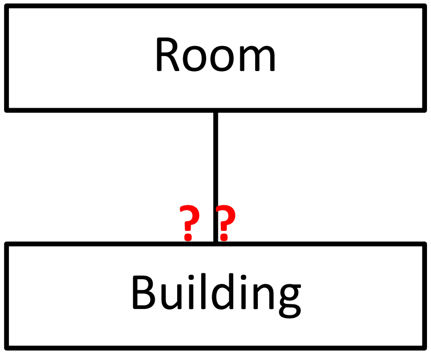
- Existence of Part does not depend on the existence of Whole.
- Lifetime of Part does not depend on Whole.
- No single instance of whole is the unique owner of Part (might be shared with other instances of Whole).

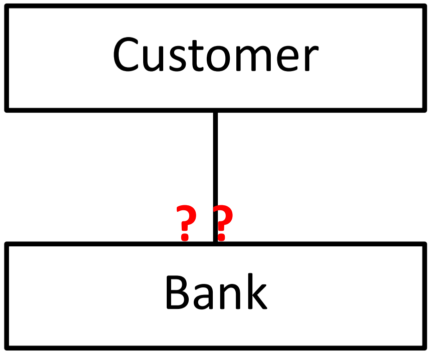
Composition



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

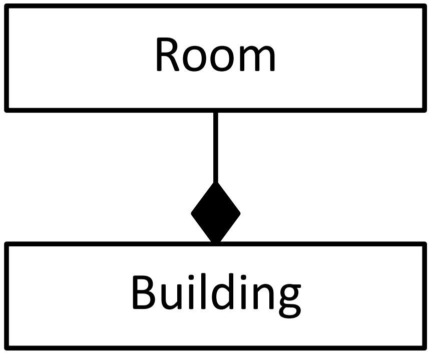
Aggregation or Composition?



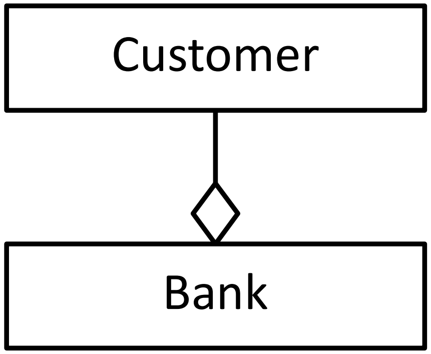


Aggregation or Composition?

Composition

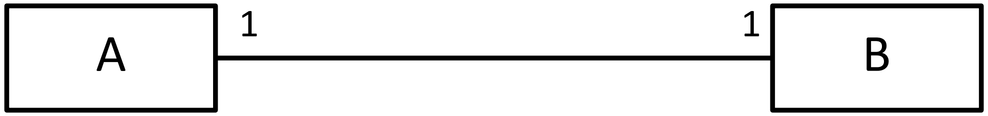


Aggregation

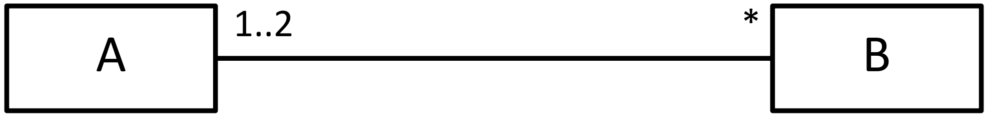


What about class and students or body and body parts?

UML class diagram: multiplicity

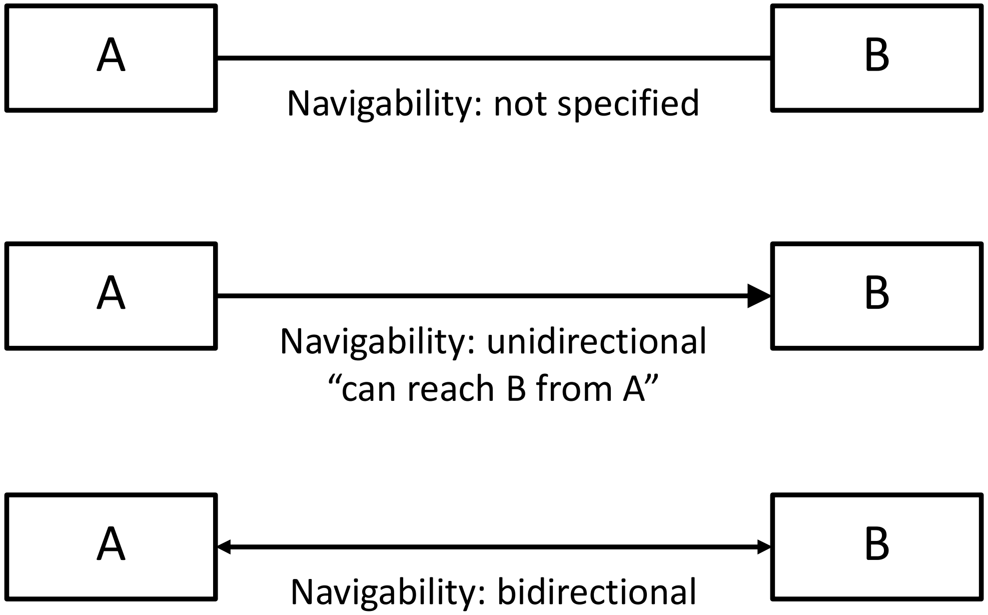


Each A is associated with exactly one B
Each B is associated with exactly one A



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

UML class diagram: navigability

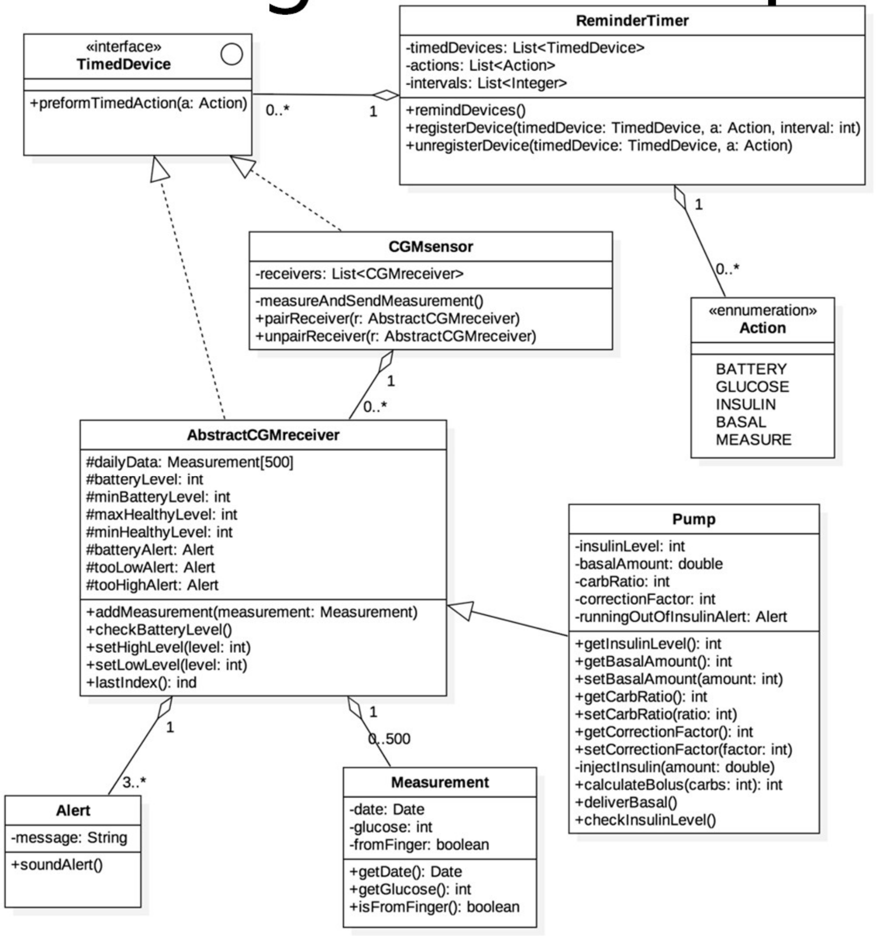


Navigability: not specified

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

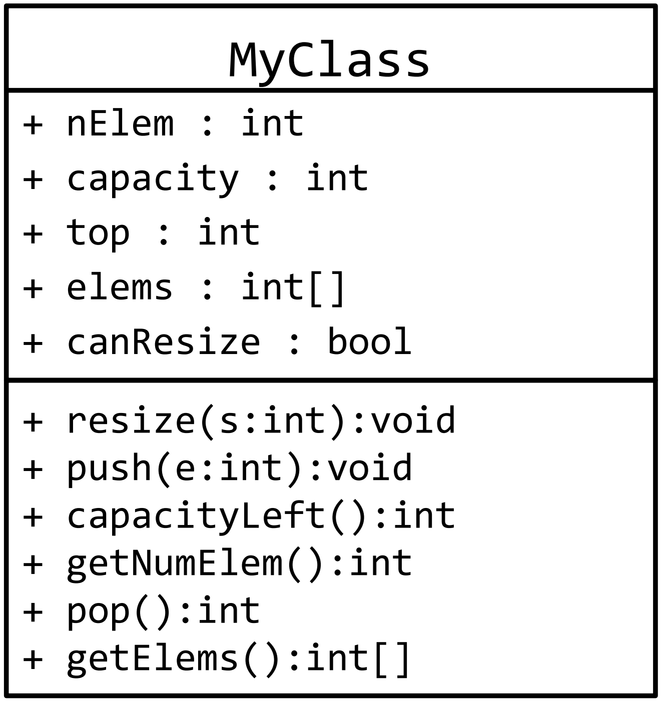
- Unified notation for modeling OO systems.
- Allows different levels of abstraction.
- Suitable for design discussions and documentation.

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

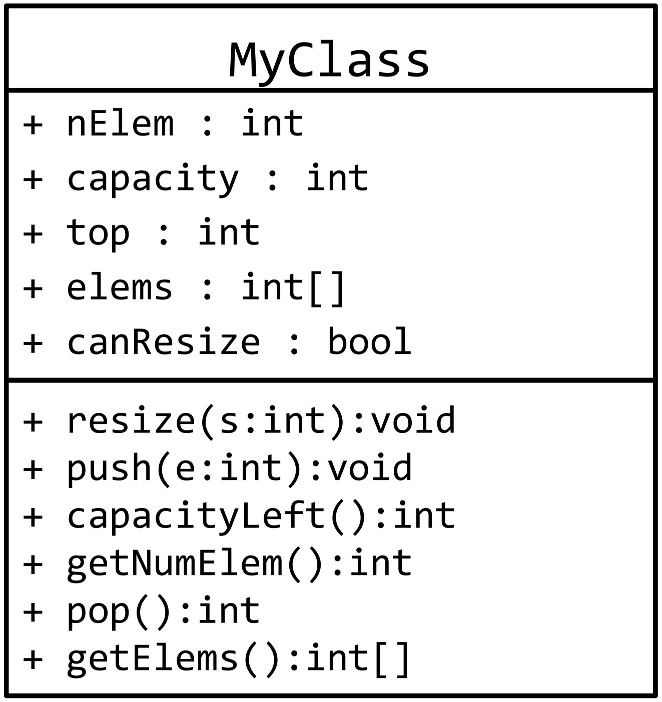
Information hiding



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

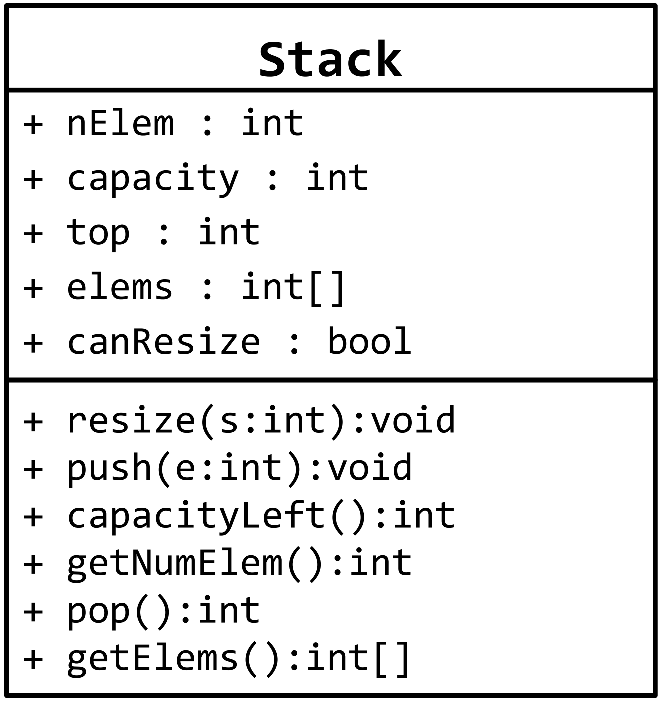
Information hiding



```
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

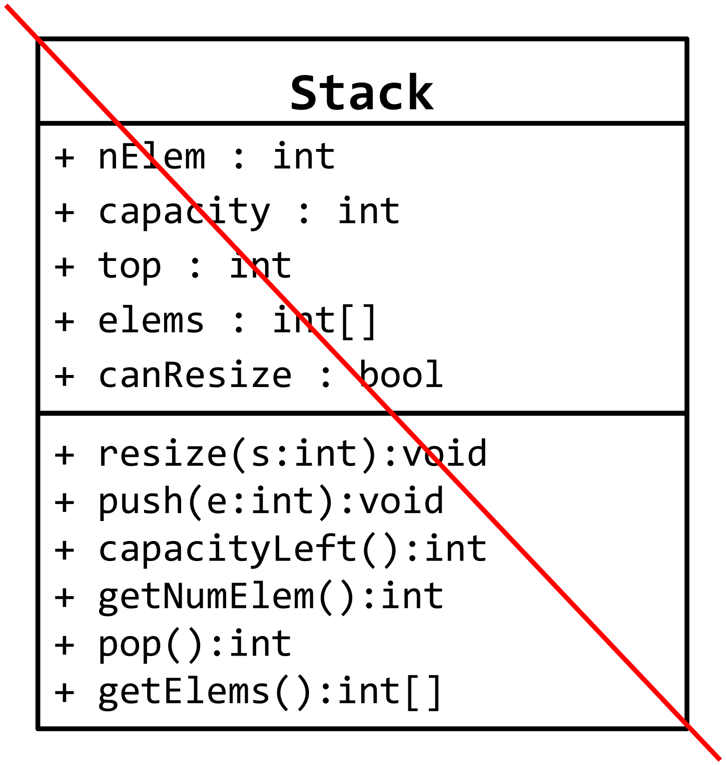


```
Stack
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]
```

```
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?

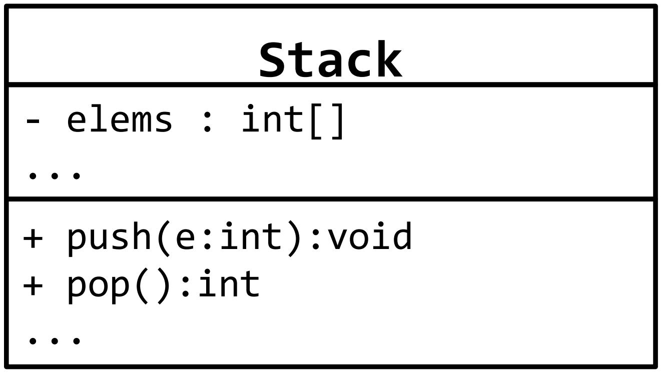
Information hiding



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

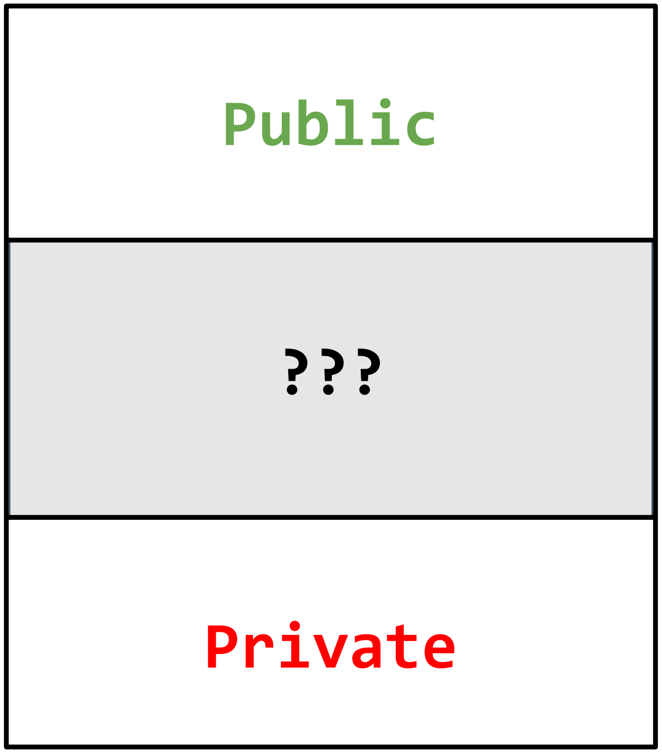
```
- elems : int[]
...
```

```
+ 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

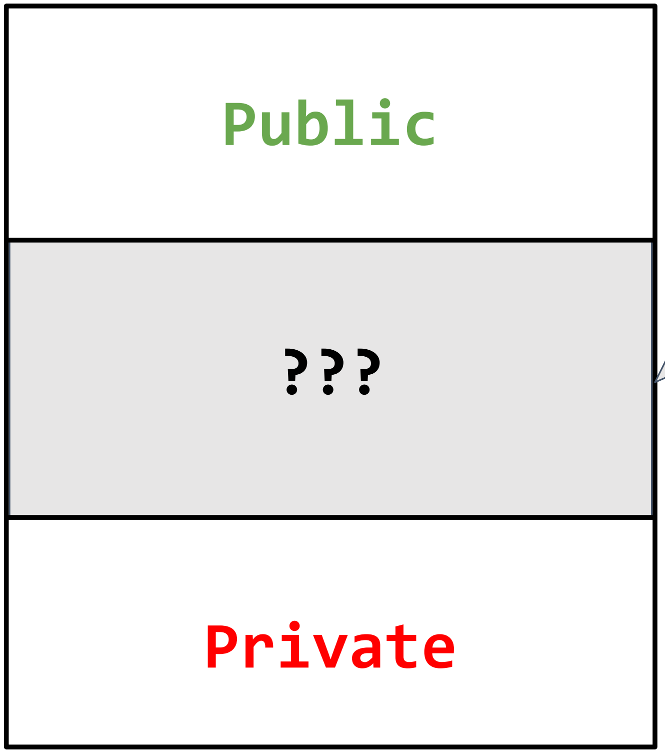


Public

???

Private

Information hiding vs. visibility



Public

???

Private

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

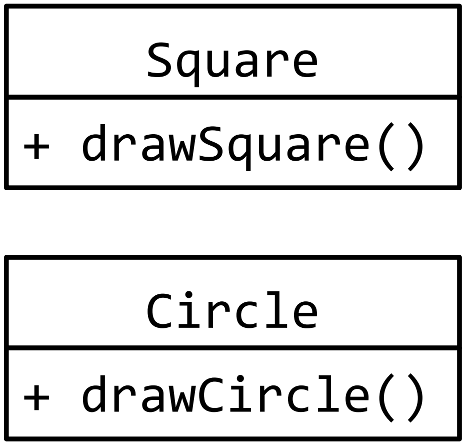
Open/closed principle

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?



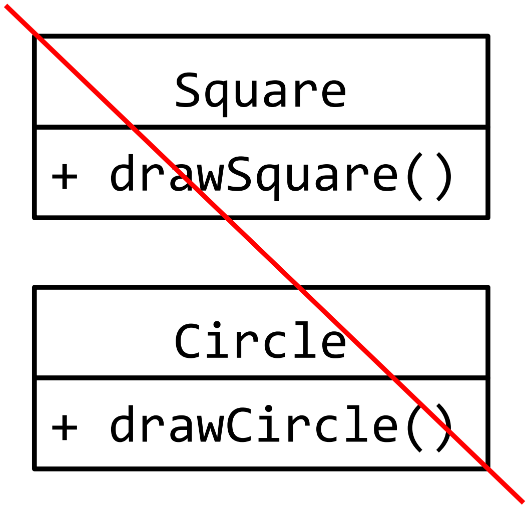
Open/closed principle

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!



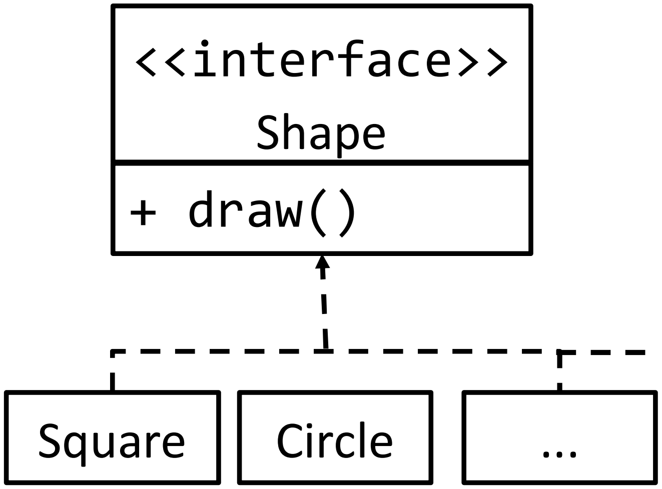
Open/closed principle

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 {
     …
  }
}
```

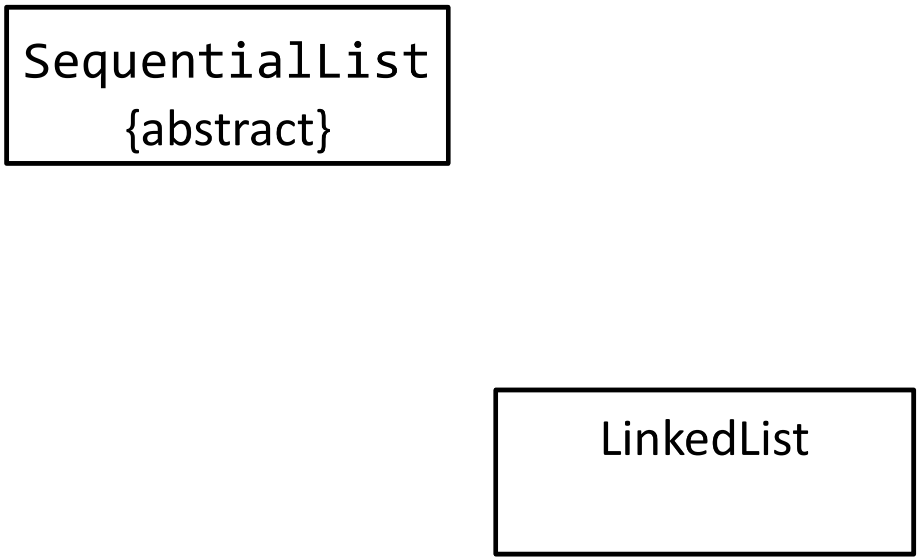
```
public static void draw(Shape s) {
  s.draw();
}
```



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

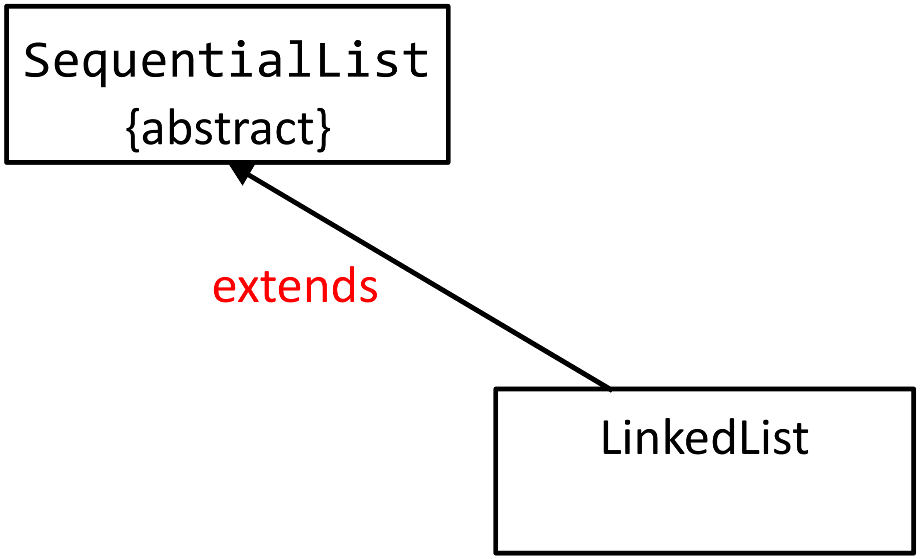


SequentialList
{abstract}

LinkedList

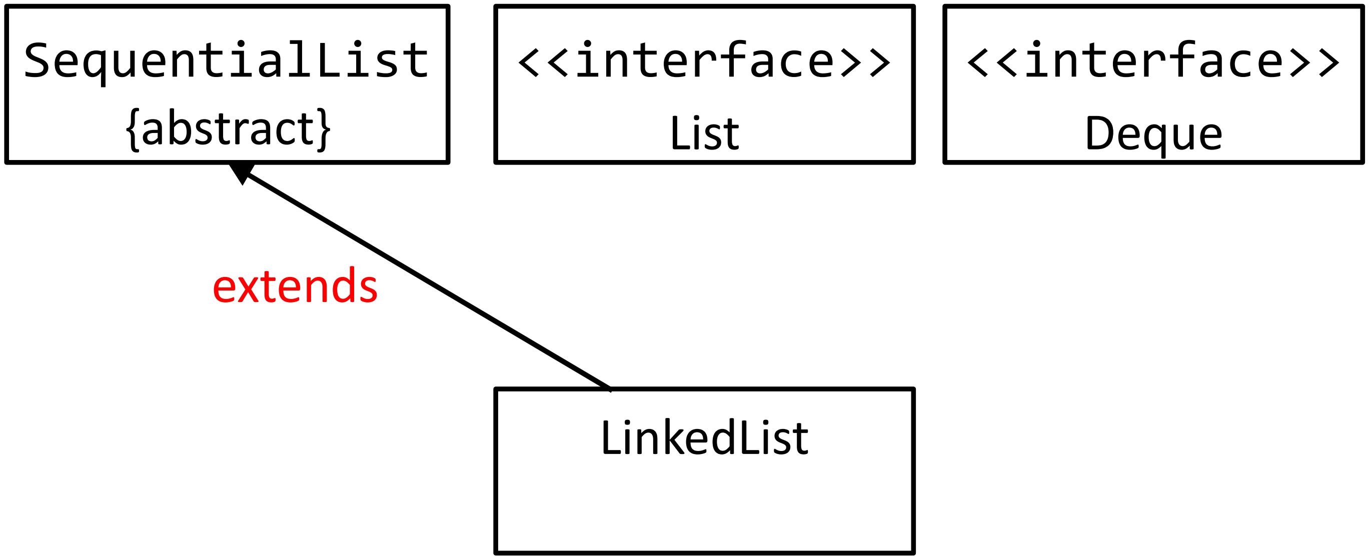
Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList



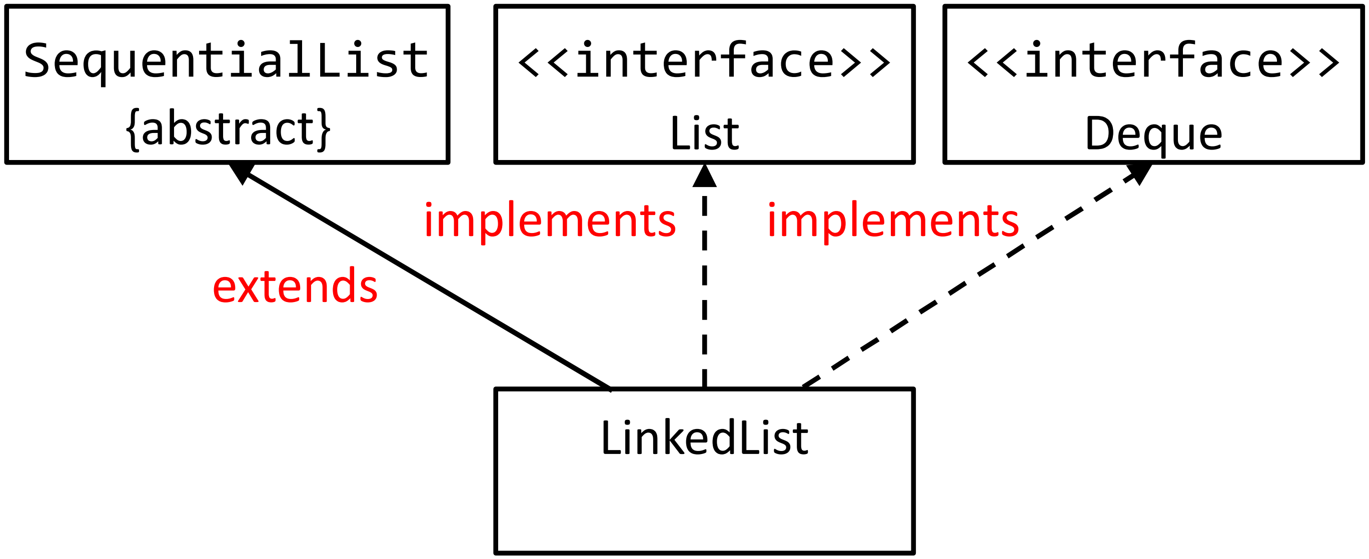
Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList

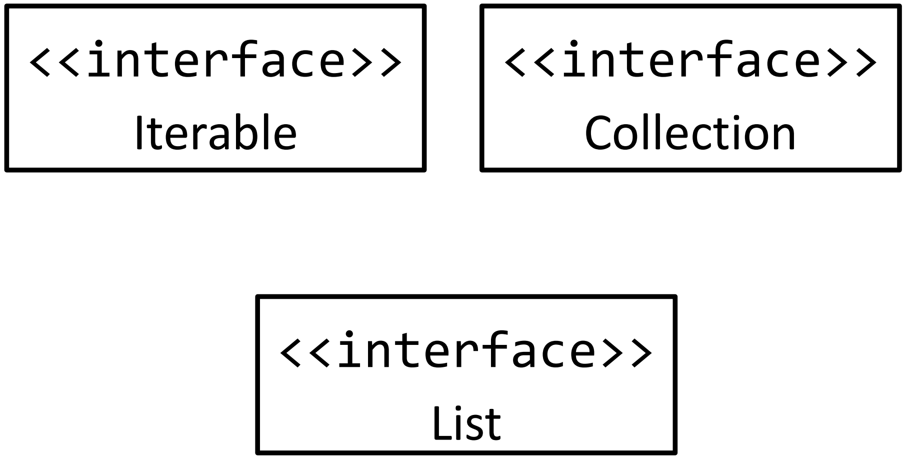


Inheritance: (abstract) classes and interfaces

LinkedList extends SequentialList implements List, Deque



Inheritance: (abstract) classes and interfaces

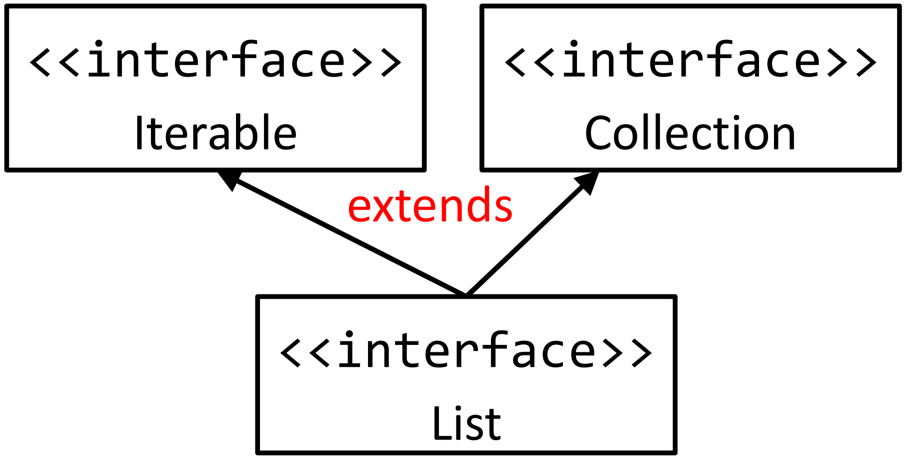


<<interface>>
Iterable

<<interface>>
Collection

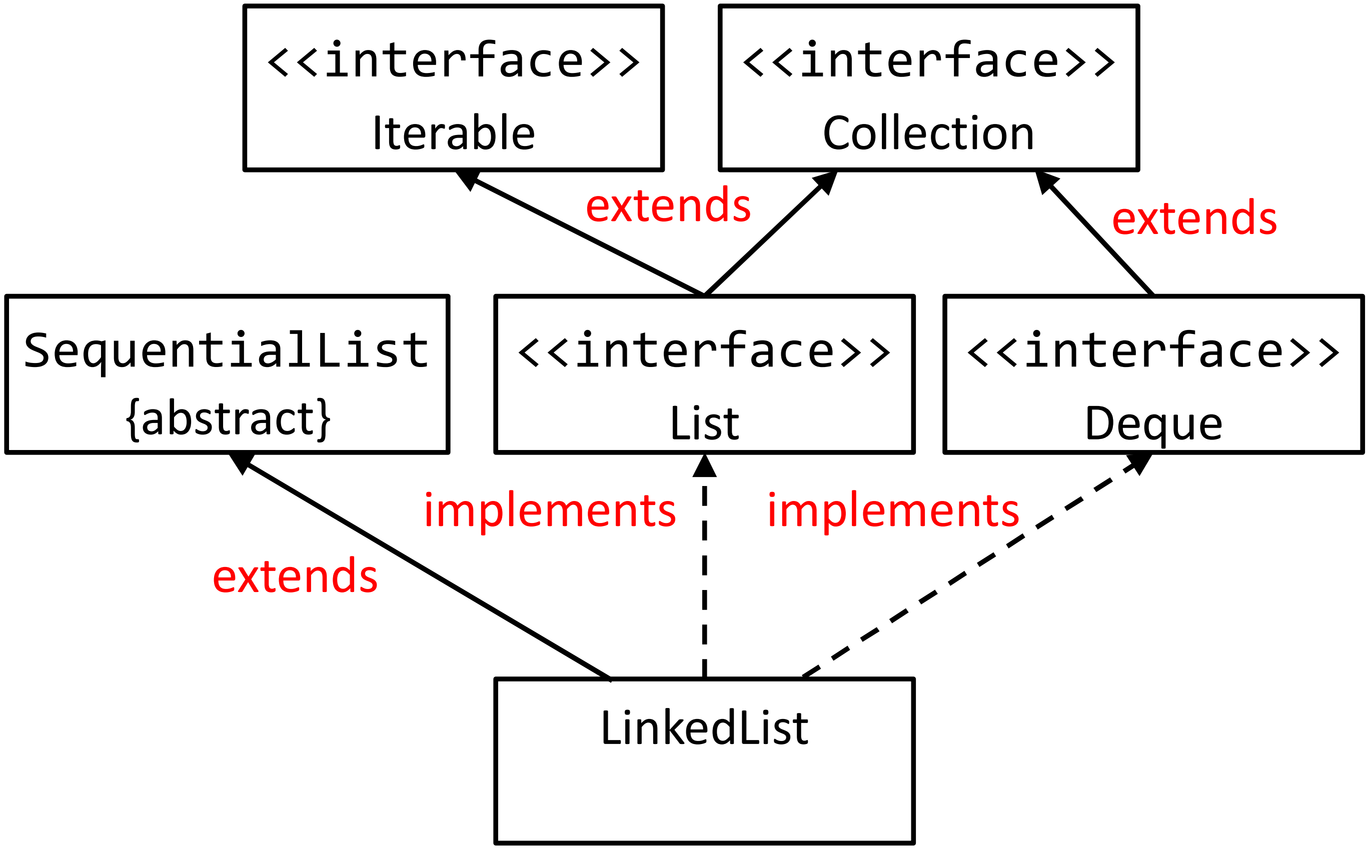
<<interface>>
List

Inheritance: (abstract) classes and interfaces



List extends Iterable, Collection

Inheritance: (abstract) classes and interfaces

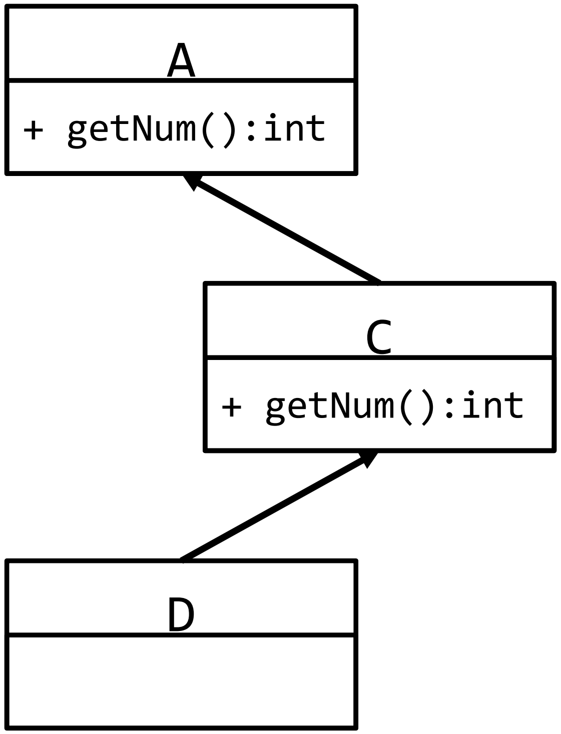


OO design principles

- 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();
...
```



A

+ getNum():int

C

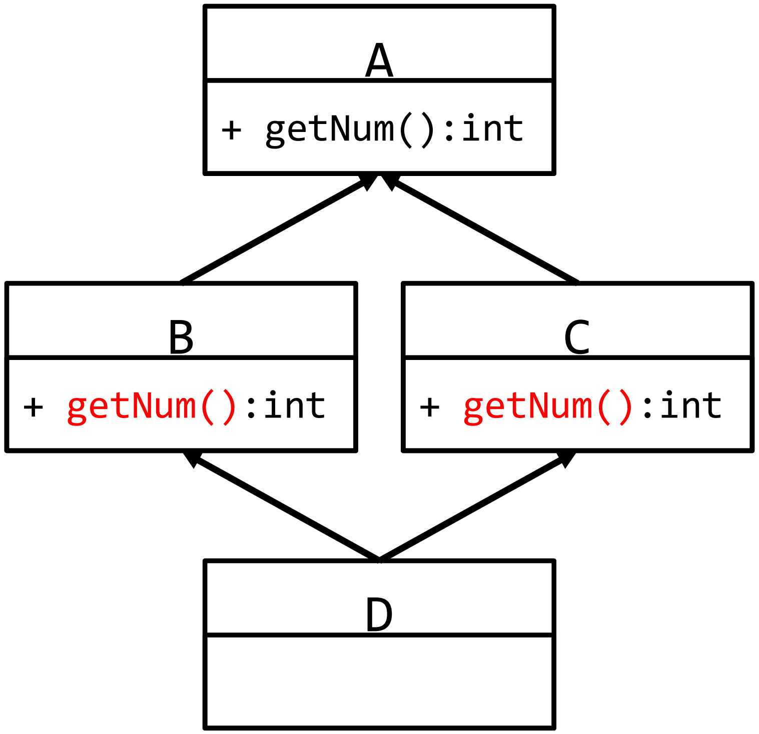
+ getNum():int

D

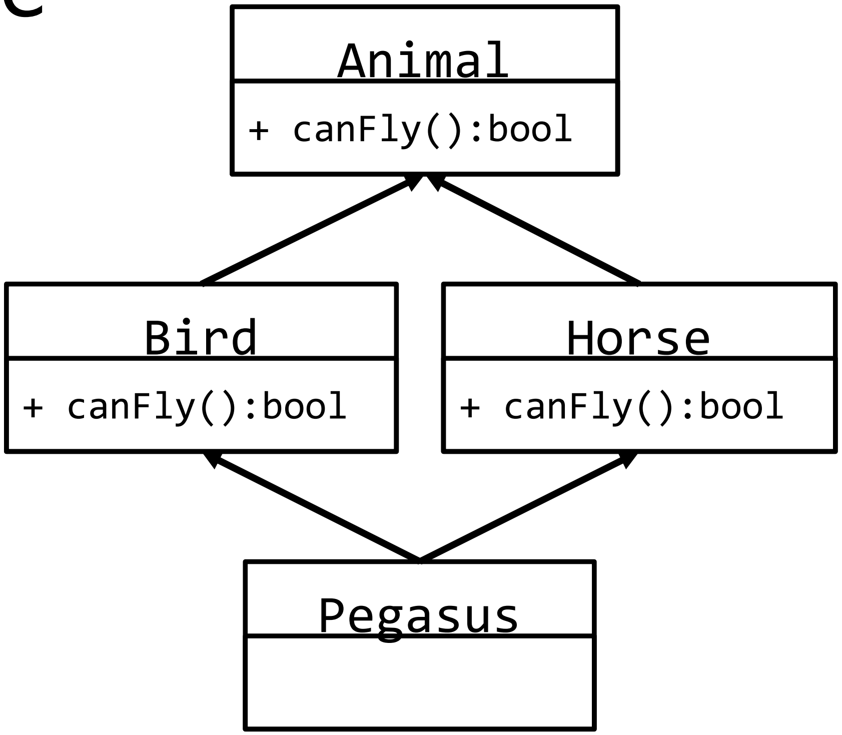
The “diamond of death”: the problem

```
...
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.

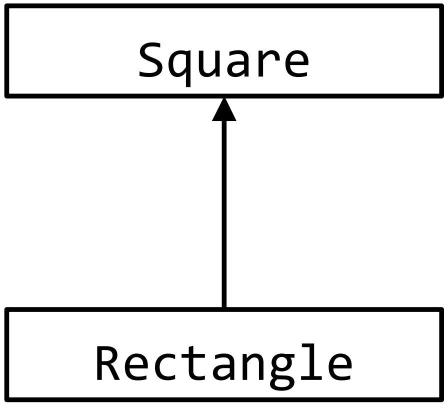
OO design principles

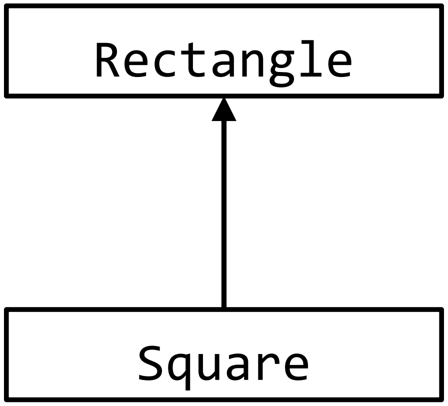
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- **Liskov substitution principle**
- Composition/aggregation over inheritance

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?

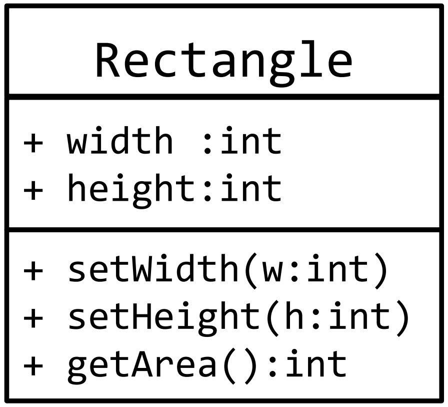


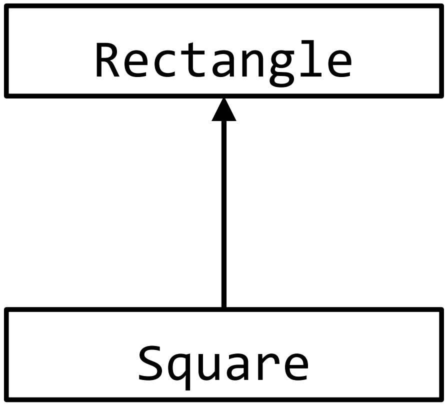


Design principles: Liskov substitution principle

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.



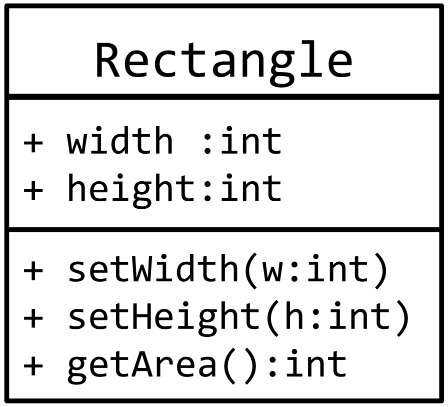


Is the subtype requirement fulfilled?

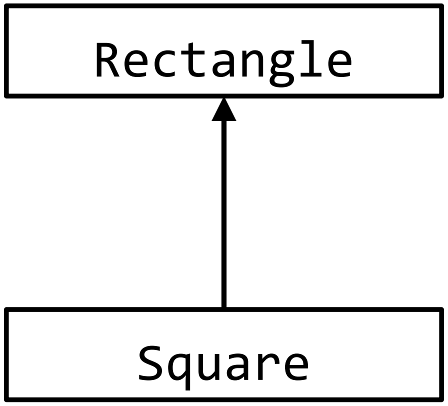
Design principles: Liskov substitution principle

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.



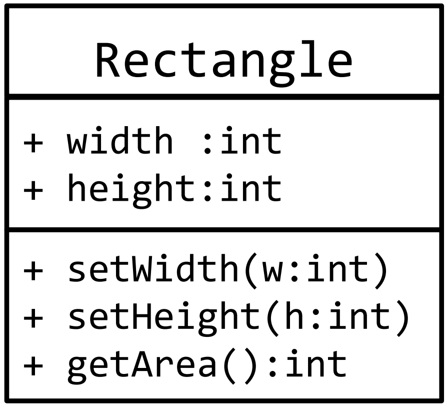
```
Rectangle r =
  new Rectangle(2,2);


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

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



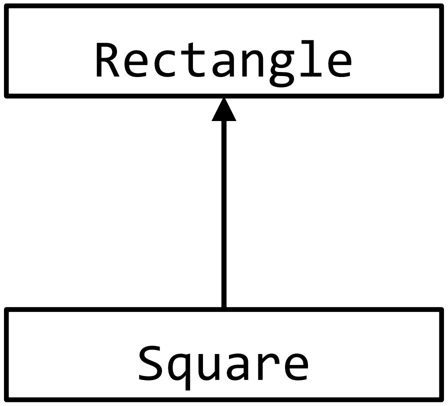
Design principles: Liskov substitution principle

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.



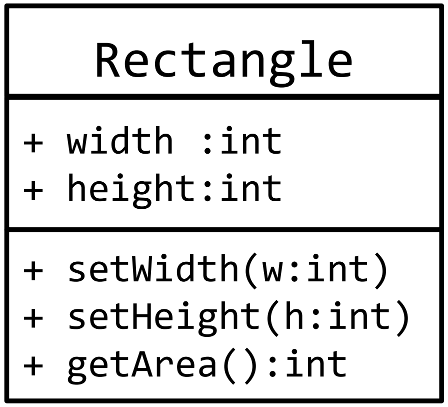
```
Rectangle r =
  ~~new Rectangle(2,2);~~
  new Square(2);

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

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



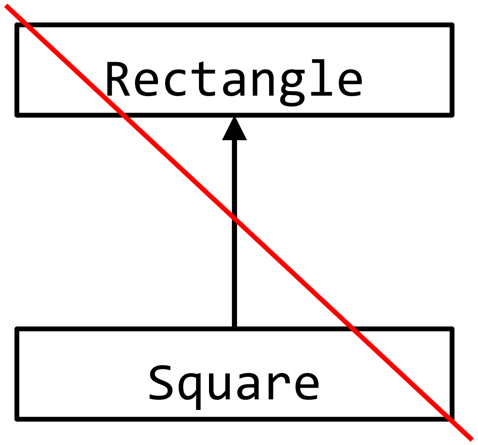
Design principles: Liskov substitution principle

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());
```

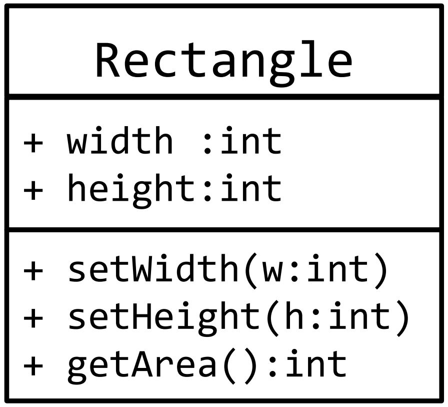


Violates the Liskov substitution principle!

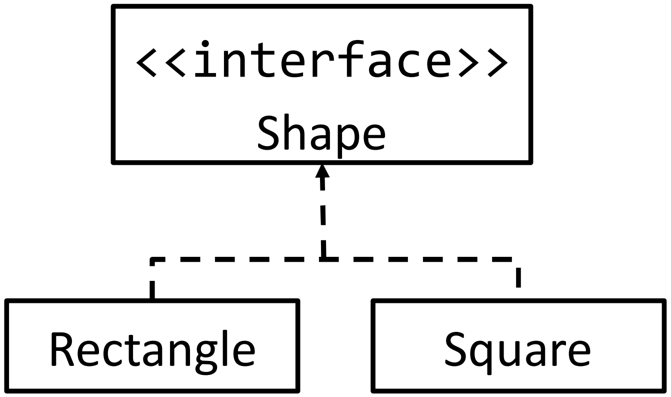
Design principles: Liskov substitution principle

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.



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



<<interface>>

Shape

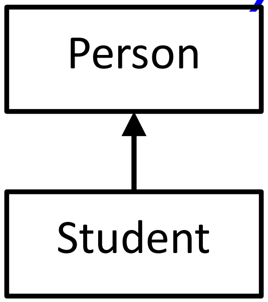
Rectangle

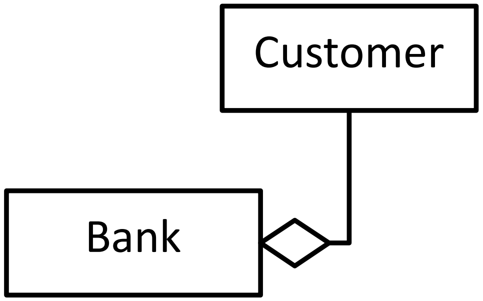
Square

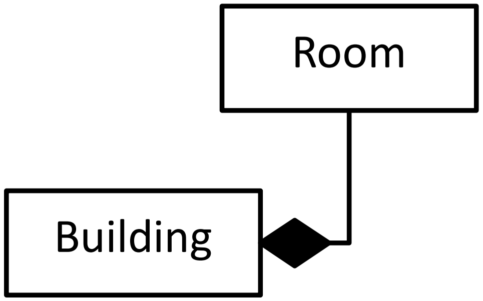
OO design principles

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

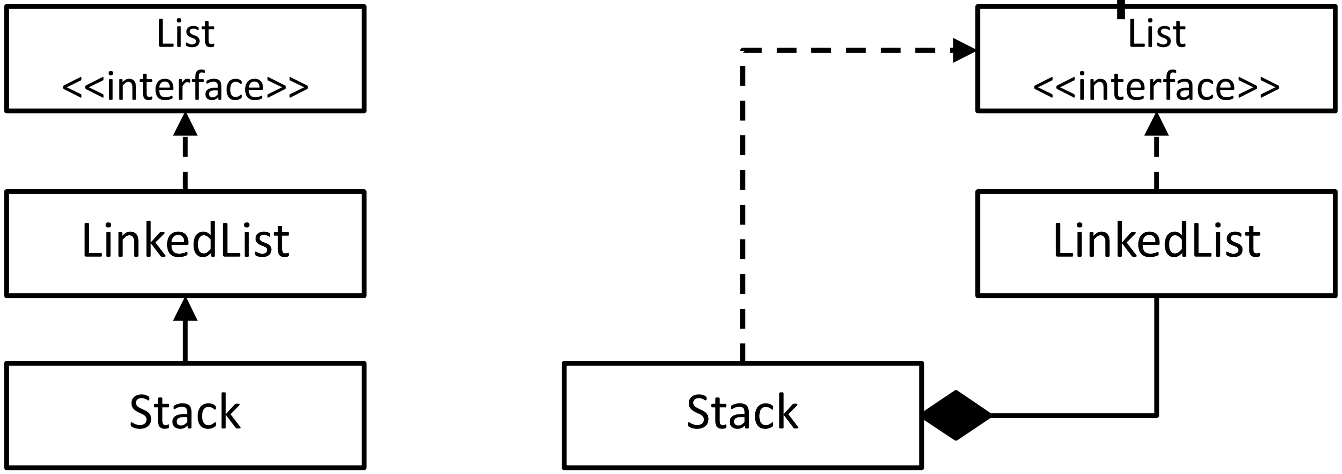
is-a relationship

```
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?

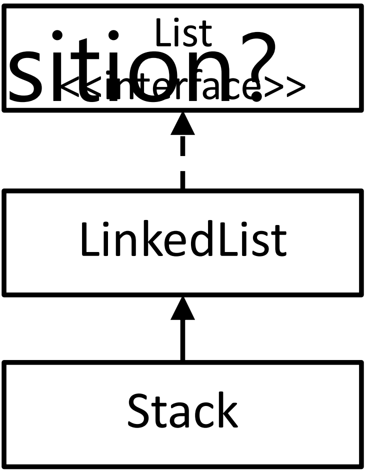


```
public class Stack<E>
    extends
LinkedList<E> {
  ...
}
```

```
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?

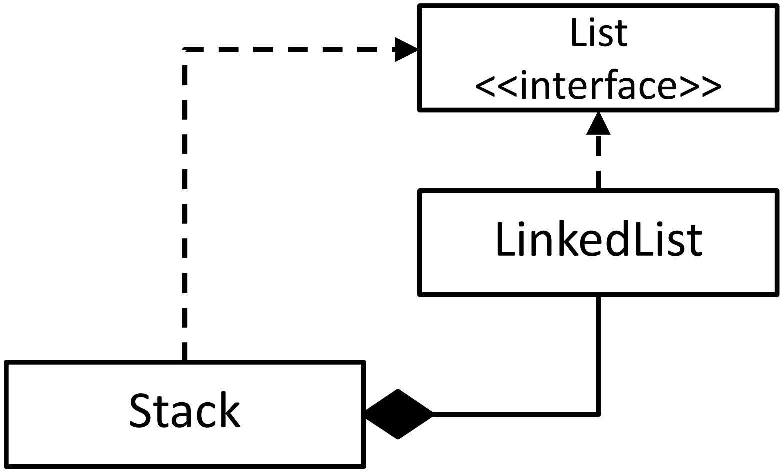


Pros

- 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.

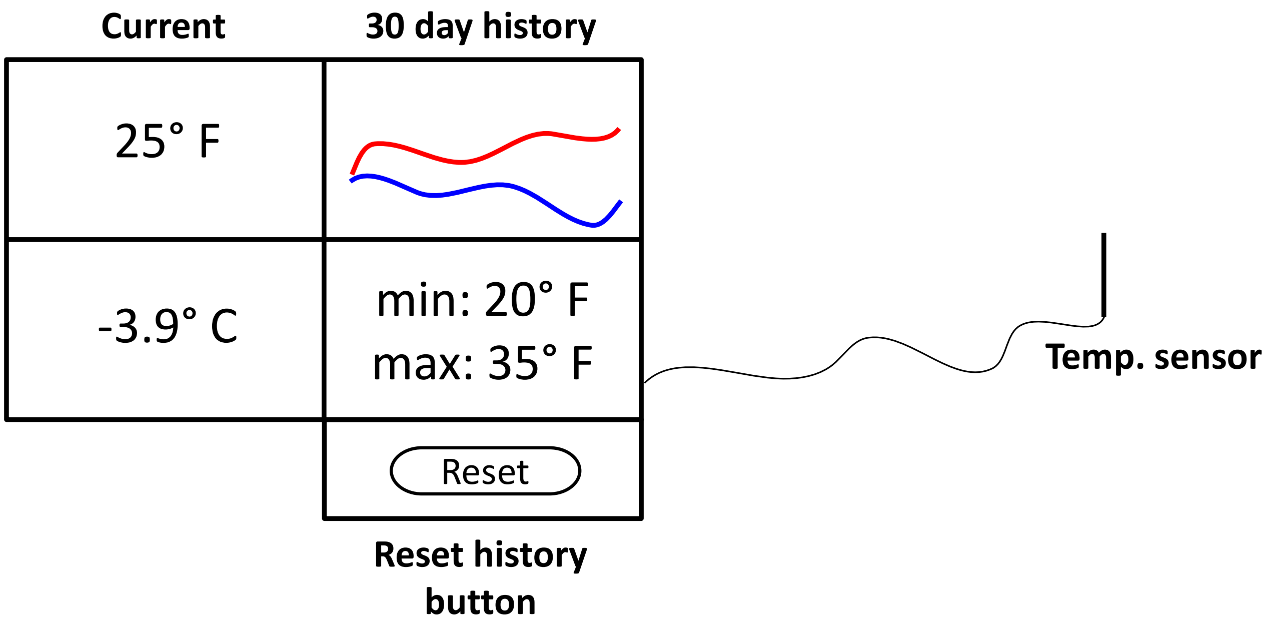
OO design principles: summary

- 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

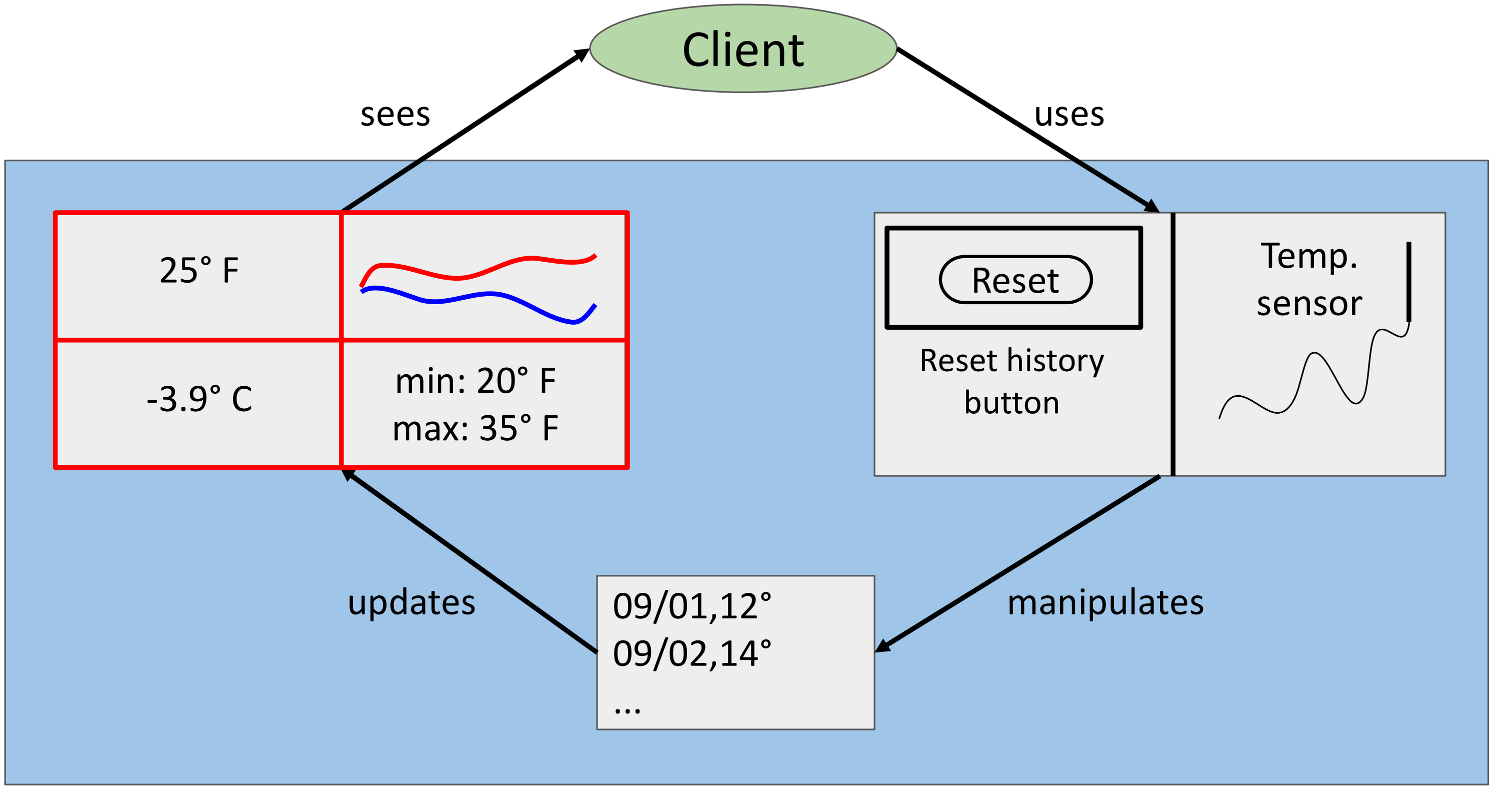
max: 35° F

Reset

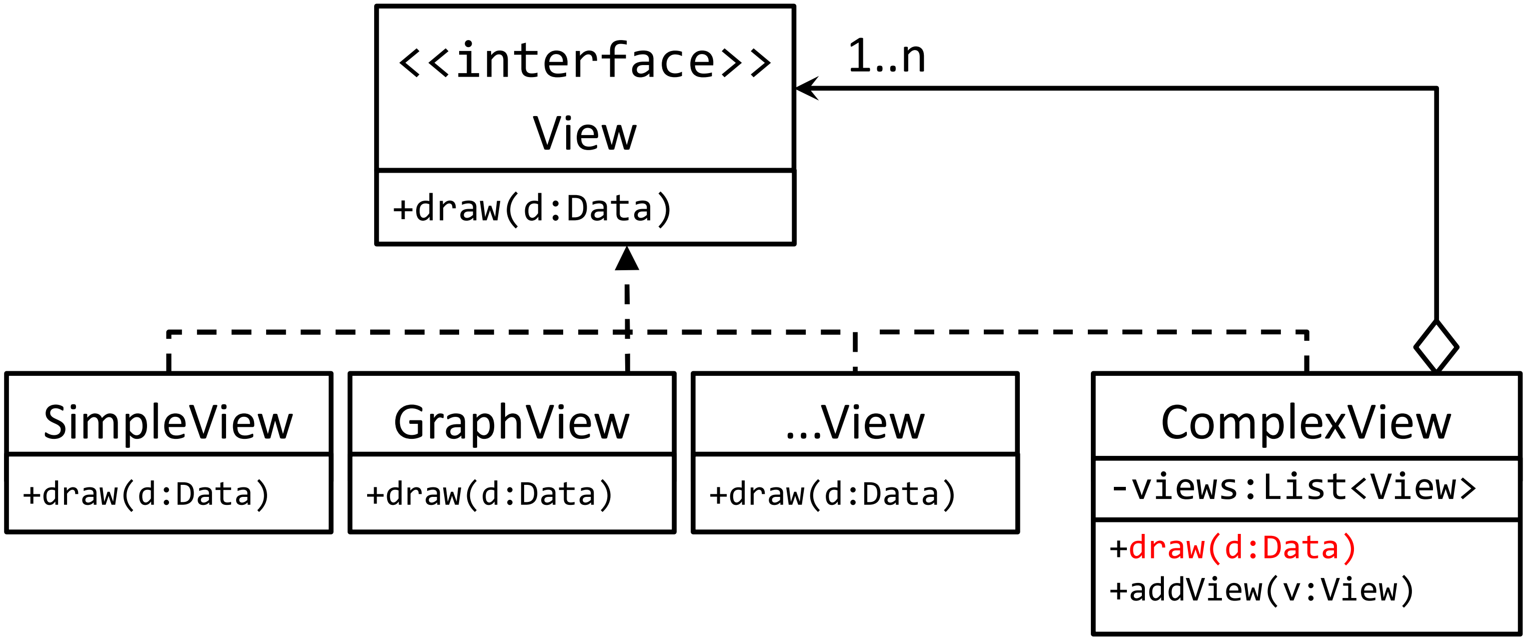
Reset history button

Temp. sensor

What’s a good design for the view component?



Weather station: view



<<interface>>
View

+draw(d:Data)

1..n

SimpleView

+draw(d:Data)

GraphView

+draw(d:Data)

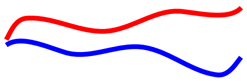
...View

+draw(d:Data)

ComplexView

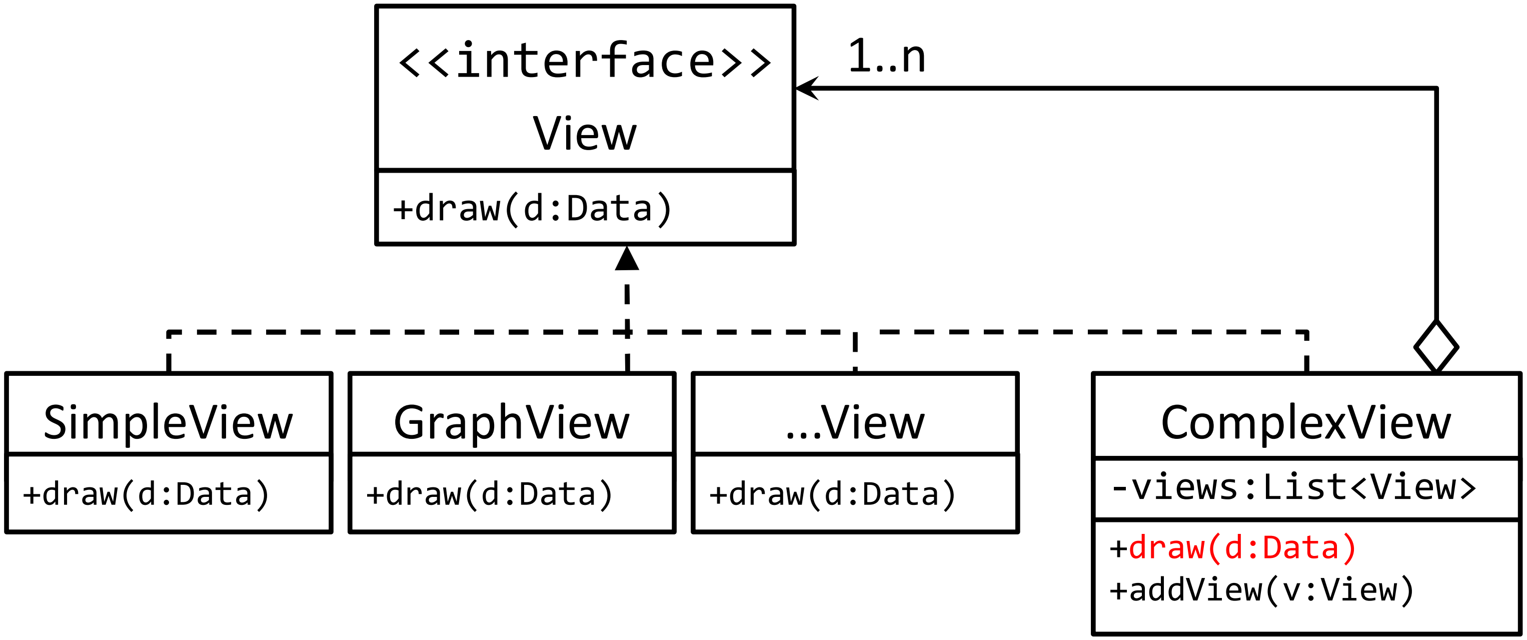
-views:List<View>

+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)

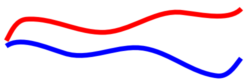
1..n

SimpleView
+draw(d:Data)

GraphView
+draw(d:Data)

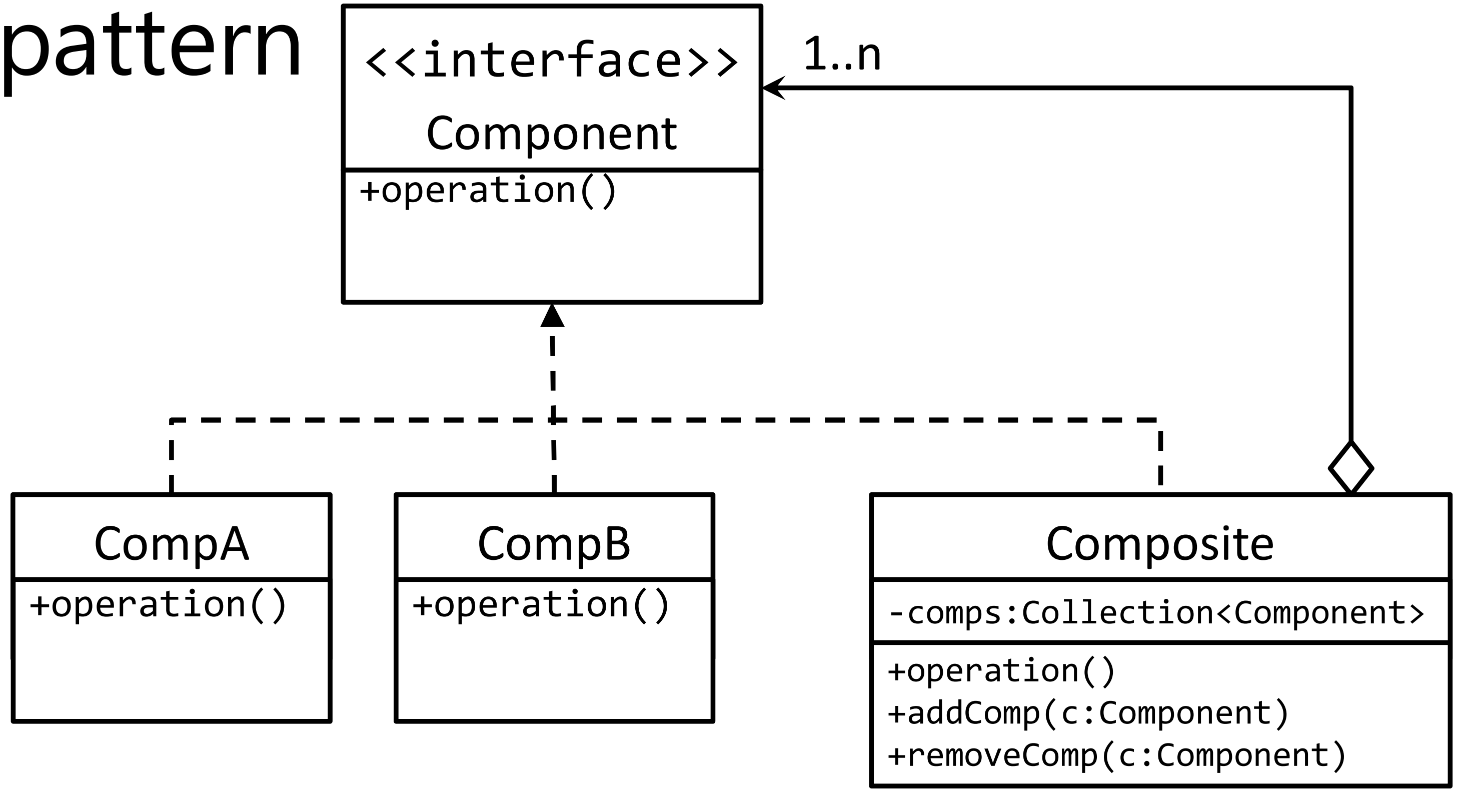
...View
+draw(d:Data)

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

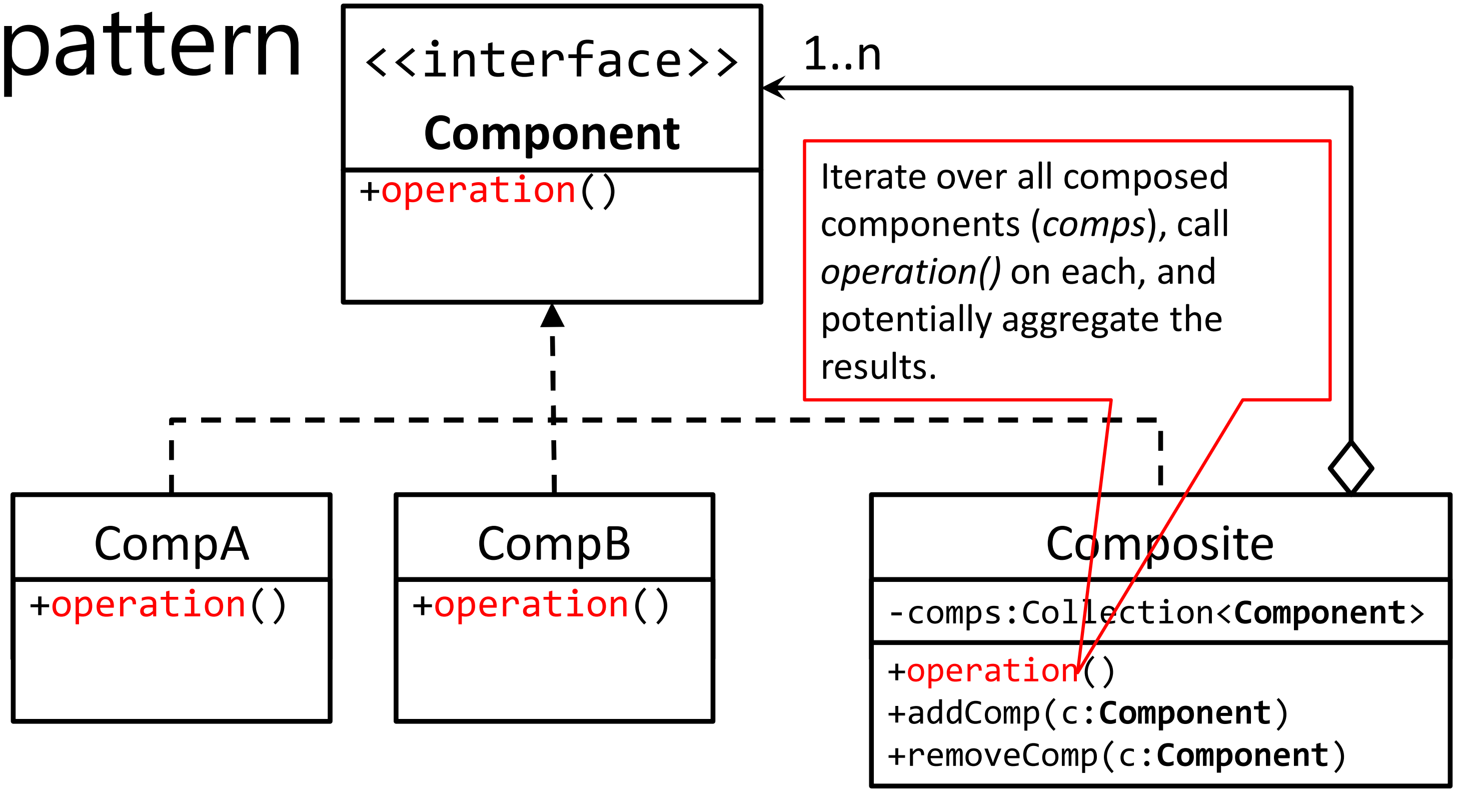
25° F	
-3.9° C	min: 20° F max: 35° F

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

The general solution: Composite pattern

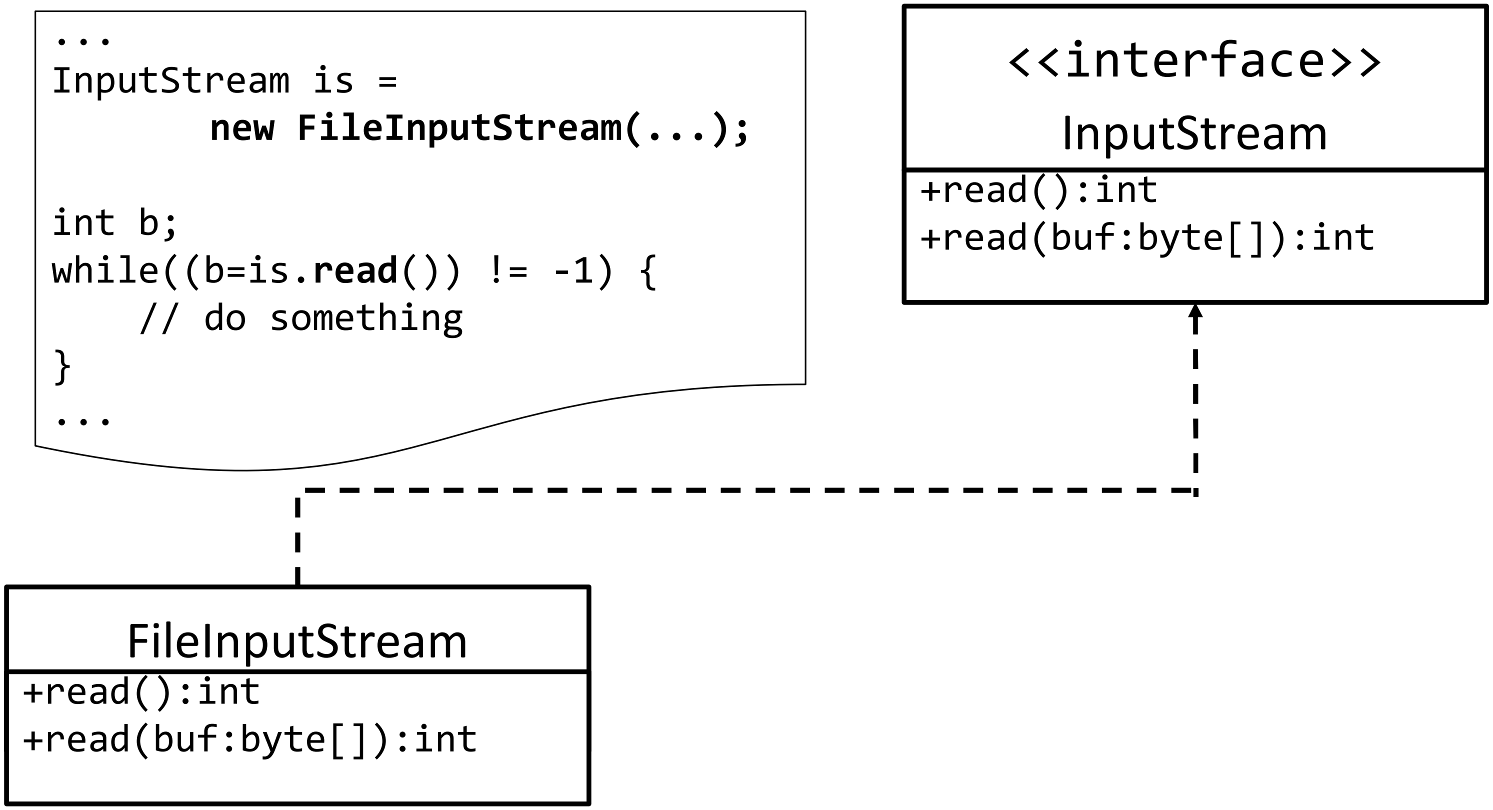


The general solution: Composite pattern



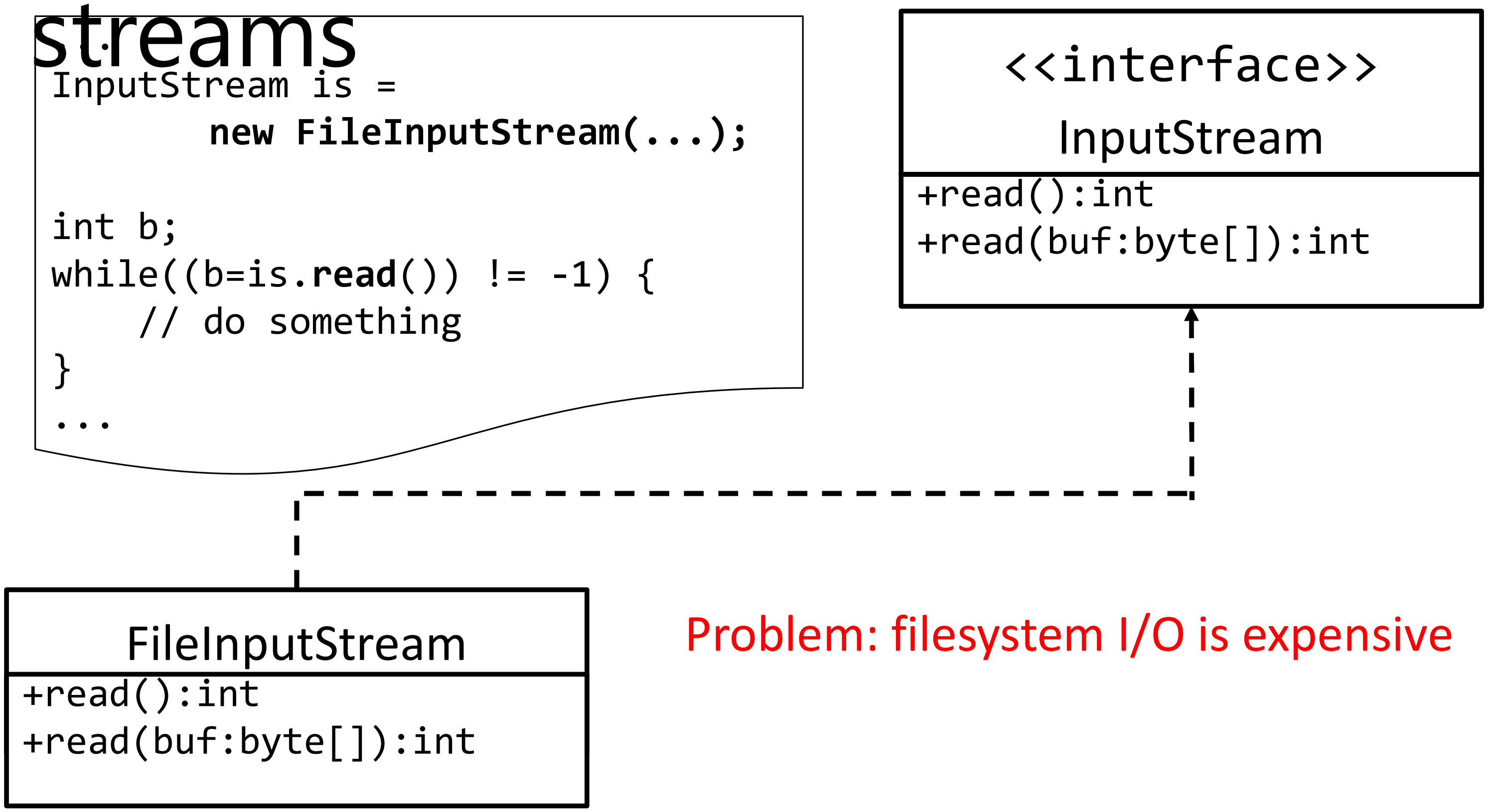
Iterate over all composed components (*comps*), call *operation()* on each, and potentially aggregate the results.

Another design problem: I/O streams



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

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

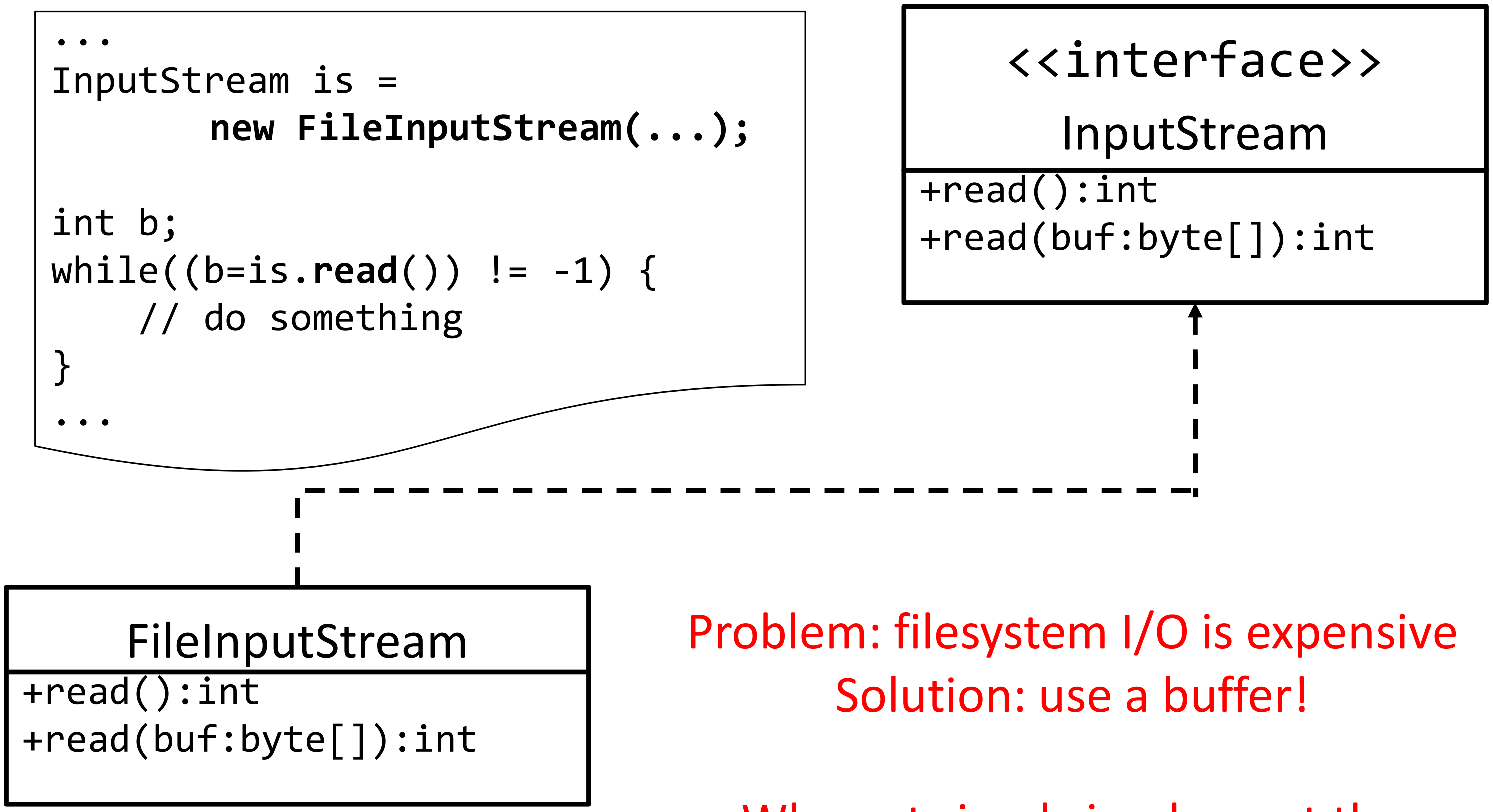
Another design problem: I/O streams



```
...
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



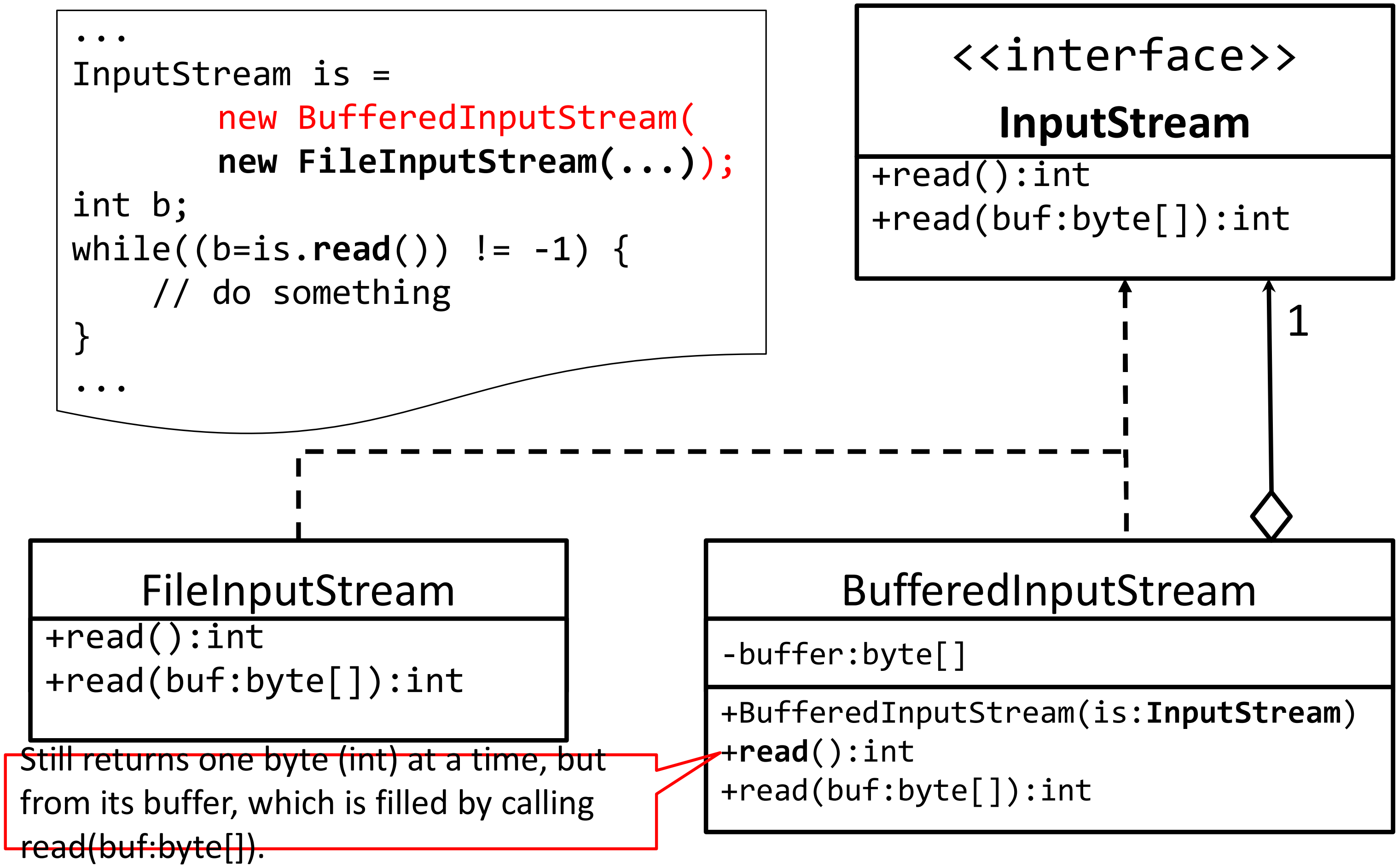
```
...
InputStream is =
        new FileInputStream(...);

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?

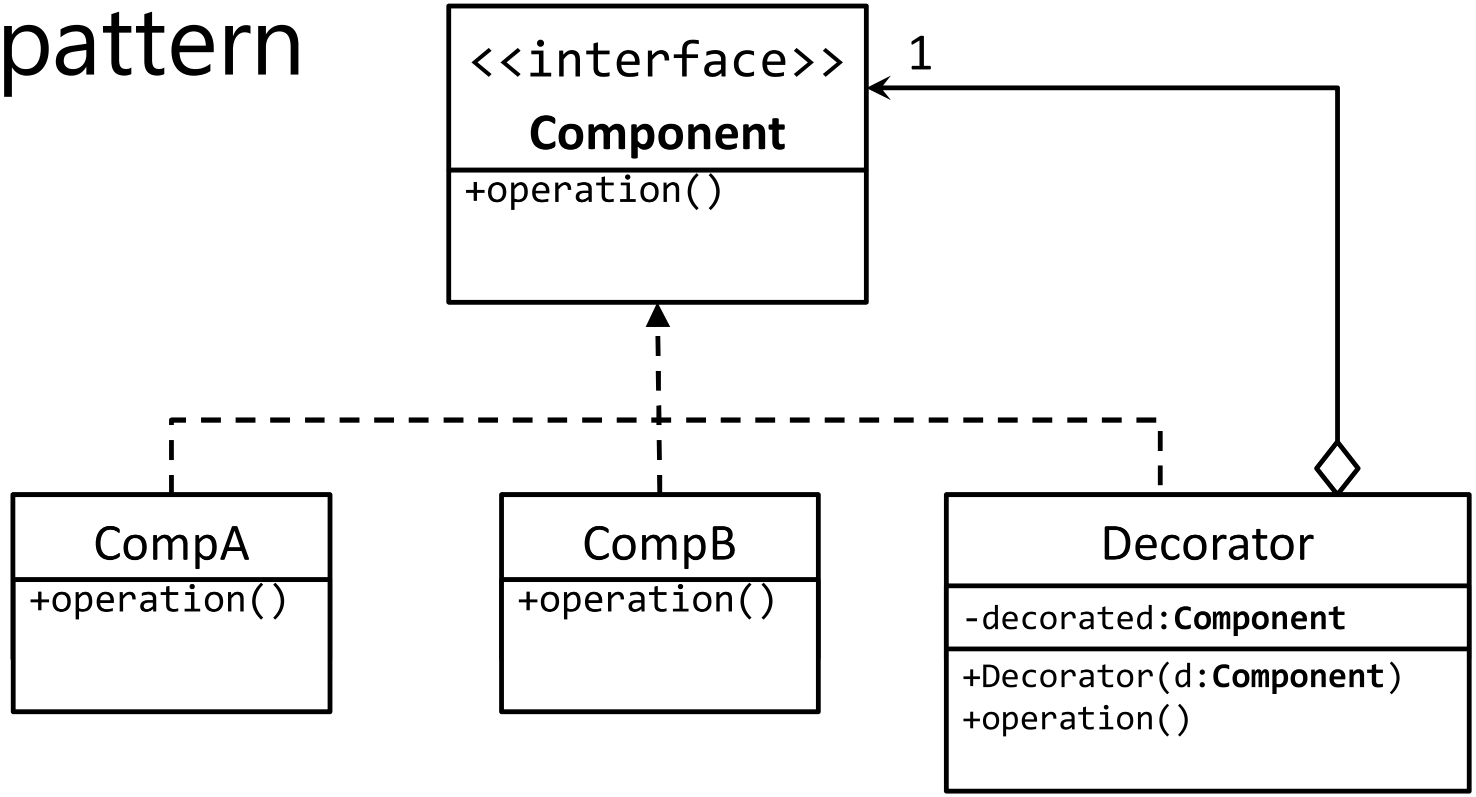
Another design problem: I/O streams

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

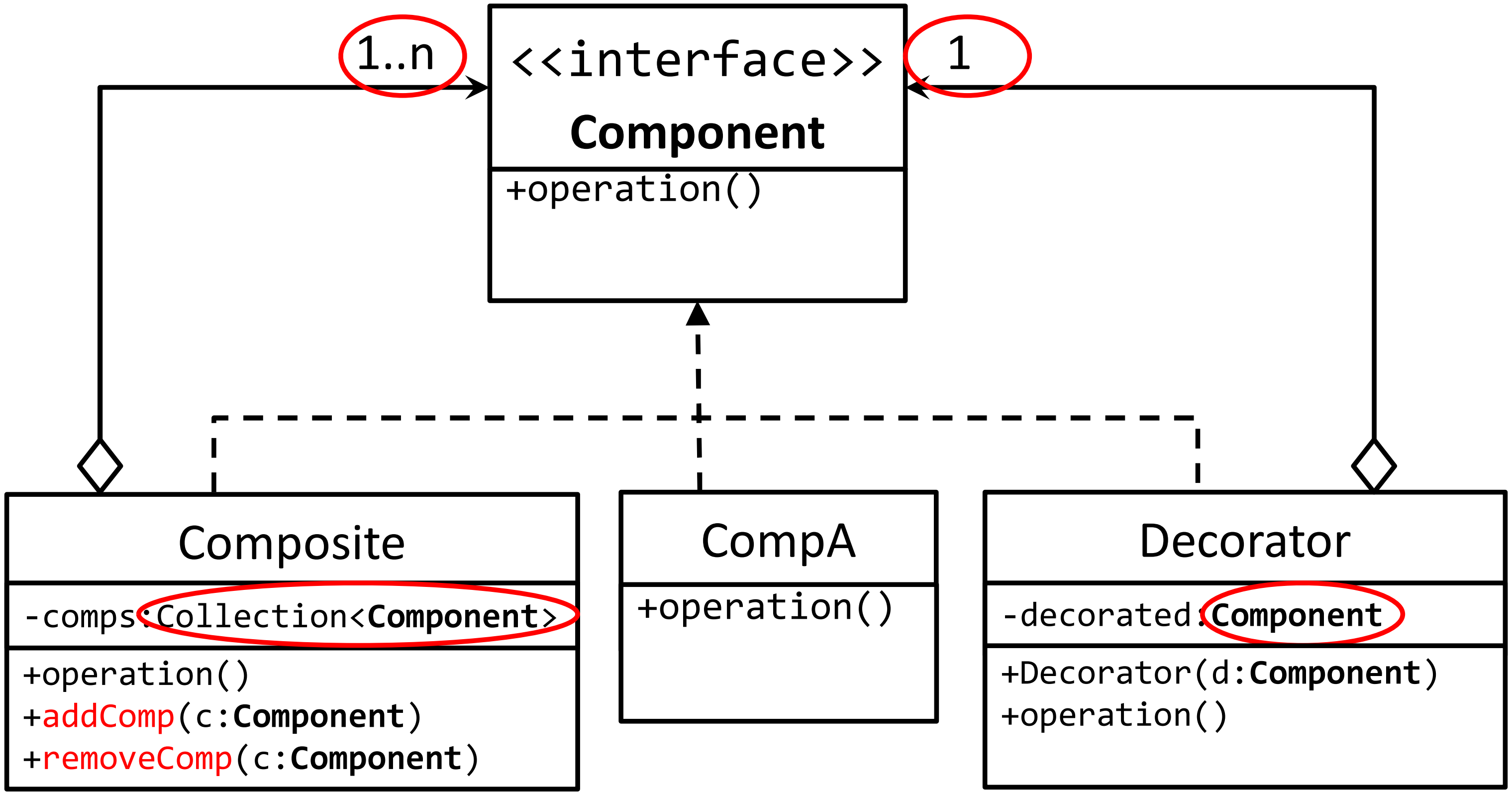


Still returns one byte (int) at a time, but from its buffer, which is filled by calling read(buf:byte[]).

The general solution: Decorator pattern



Composite vs. Decorator



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