Software Design
CSE 403 Software Engineering
Autumn 2023
Today’s Outline

1. Quick recap – Architecture vs Design
2. Some practical design considerations
3. Class quiz on some design/coding best practices 😊

See Appendix for a short primer on CSE 331 design material:
• UML (unified modeling language)
• Object oriented design principles
• Design patterns
Reminder – Weekly status reports start now

Due each Wednesday 11:59pm
Submit to your github – 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:

- Team report (status update for your TA, including an agenda for the project meeting); and
- Contributions of individual team members.

Both sections should have the following three subsections -- each about a paragraph or organized as bullet points.

- The first subsection is easy. It should be an exact copy of the third section from last week (i.e., goals from a week ago). It can be empty for the first week.
- The second subsection should report on progress and issues: what you did, what worked, what you learned, where you had trouble, and where you are blocked.
- The third subsection should outline your plans and goals for the following week. Bullet points are fine. If tasks from one week aren’t yet complete, they should roll over into tasks for the next week, with an updated estimate for time to completion. For the team report, this subsection should be higher-level 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 bigger picture in 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.
High level overview from last class

Requirements
Architecture
Design
Source code

Development process
Level of abstraction

Pointing Arrows

Diagram with categorized sections
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)
- Single responsibility (focus 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 cohesion, loose coupling principle (path to design success)
Let’s shake things up and look at code!

Many thanks to René Just, UW CSE Prof
Quiz setup

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

- **Round 1 (PollEverywhere - [https://pollev.com/cse403au](https://pollev.com/cse403au))**
  - 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 counter argument
Round 1: good or bad?

https://pollev.com/cse403au
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?

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

```java
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) {
        ...  
    }
    ...
}
```
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;
    }
}

Design Quiz - Good or bad?
Round 1: good or bad?
and Round 2: why?
Spoiler alert - staff opinions on this 😊

- Snippet 1: bad
- Snippet 2: bad
- Snippet 3: good
- Snippet 4: bad
- Snippet 5: bad
- Snippet 6: good
Snippet 1: good or bad?

```java
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

Total Results: 0
Snippet 1: this is bad! why?

```java
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?

```java
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.
Snippet 1: this is bad! why?

```java
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?

```java
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?

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

And the survey says ...
When poll is active, respond at pollev.com/cse403au

Snippet2: addStudent

Good
Bad

Total Results: 0
Snippet 2: short but bad! why?

```java
public void addStudent(Student student, String course) {
    if (course.equals("CSE403")) {
        cse403Students.add(student);
    }
    allStudents.add(student)
}
```
Snippet 2: short but bad! why?

```java
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?

```java
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 ...
Snippet 3: this is good, but why?

```java
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?

```java
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?

```java
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

When poll is active, respond at poll-ev.com/cse403au

Total Results: 0
Snippet 4: also bad! huh?

```java
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?

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

*Assuming these are pass by reference...*

Method parameters should be final (sacred); use local variables to sanitize inputs.
Snippet 5: good or bad?

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

And the survey says ...
When poll is active, respond at pollev.com/cse403au

Snippet5: ArrayList

Good

Bad

Total Results: 0
Snippet 5: Java API, but still bad! why?

```java
public class ArrayList<E> {
    public E remove(int index) {
        ...
    }
    public boolean remove(Object o) {
        ...
    }
    ...
}
```
Snippet 5: Java API, but still bad! why?

```java
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?

```java
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?

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

```java
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?

```java
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 ...
When poll is active, respond at pollev.com/cse403au

Snippet6: Point

Good
Bad

Total Results: 0
Snippet 6: this is good, but why?

```java
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?

```java
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

MyClass
UML class diagram: basic notation

<table>
<thead>
<tr>
<th>Name</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyClass</td>
<td>- attr1 : type</td>
</tr>
<tr>
<td></td>
<td>+ foo() : ret_type</td>
</tr>
</tbody>
</table>

Attributes
<visibility> <name> : <type>

Methods
<visibility> <name>(<param>*) : <return type>
<param> := <name> : <type>
UML class diagram: basic notation

<table>
<thead>
<tr>
<th>MyClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>- attr1 : type</td>
</tr>
<tr>
<td># attr2 : type</td>
</tr>
<tr>
<td>+ attr3 : type</td>
</tr>
<tr>
<td>~ bar(a:type) : ret_type</td>
</tr>
<tr>
<td>+ foo() : ret_type</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>MyClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>- attr1 : type</td>
</tr>
<tr>
<td># attr2 : type</td>
</tr>
<tr>
<td>+ attr3 : type</td>
</tr>
<tr>
<td>~ bar(a:type) : ret_type</td>
</tr>
<tr>
<td>+ foo() : ret_type</td>
</tr>
</tbody>
</table>

- **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
public class Person {
    // ...
}

public class Student extends Person {
    private int id;
    public Student(String name, int id) {
        // ...
    }
    public int getId() {
        return this.id;
    }
}

UML class diagram: concrete example
Classes, abstract classes, and interfaces

MyClass

MyAbstractClass
{abstract}

<<interface>>
MyInterface
Classes, abstract classes, and interfaces

MyClass

public class MyClass {
    public void op() {
        ...
    }
    public int op2() {
        ...
    }
}

MyAbstractClass {abstract}

public abstract class MyAbstractClass {
    public abstract void op();
    public int op2() {
        ...
    }
}

<<interface>>
MyInterface

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.
public class SubClass extends SuperClass implements AnInterface
UML class diagram: Aggregation and Composition

**Aggregation**
- Part
- Whole
  - has-a relationship
- 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
- Whole
  - has-a relationship
- Part cannot exist without Whole.
- Lifetime of Part depends on Whole.
- One instance of Whole is the single owner of Part.
Aggregation or Composition?

Room

??

Building

Customer

??

Bank
Aggregation or Composition?

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

1. Navigability: not specified
   - A \(\rightarrow\) B

2. Navigability: unidirectional
   - “can reach B from A”
   - A \(\rightarrow\) B

3. Navigability: bidirectional
   - A \(\leftrightarrow\) B
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

### MyClass

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nElem : int</td>
<td></td>
</tr>
<tr>
<td>capacity : int</td>
<td></td>
</tr>
<tr>
<td>top : int</td>
<td></td>
</tr>
<tr>
<td>elems : int[]</td>
<td></td>
</tr>
<tr>
<td>canResize : bool</td>
<td></td>
</tr>
<tr>
<td>resize(s:int):void</td>
<td></td>
</tr>
<tr>
<td>push(e:int):void</td>
<td></td>
</tr>
<tr>
<td>capacityLeft():int</td>
<td></td>
</tr>
<tr>
<td>getNumElem():int</td>
<td></td>
</tr>
<tr>
<td>pop():int</td>
<td></td>
</tr>
<tr>
<td>getElems():int[]</td>
<td></td>
</tr>
</tbody>
</table>

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

| MyClass          | public class MyClass {
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
<td>public int nElem;</td>
</tr>
<tr>
<td>+ capacity : int</td>
<td>public int capacity;</td>
</tr>
<tr>
<td>+ top : int</td>
<td>public int top;</td>
</tr>
<tr>
<td>+ elems : int[]</td>
<td>public int[] elems;</td>
</tr>
<tr>
<td>+ canResize : bool</td>
<td>public boolean canResize;</td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
<td>...</td>
</tr>
<tr>
<td>+ push(e:int):void</td>
<td>public void resize(int s){...}</td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
<td>public void push(int e){...}</td>
</tr>
<tr>
<td>+ getNumElem():int</td>
<td>public int capacityLeft(){...}</td>
</tr>
<tr>
<td>+ pop():int</td>
<td>public int getNumElem(){...}</td>
</tr>
<tr>
<td>+ getElems():int[]</td>
<td>public int pop(){...}</td>
</tr>
<tr>
<td></td>
<td>public int[] getElems(){...}</td>
</tr>
</tbody>
</table>

What does MyClass do?
Information hiding

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

Anything that could be improved in this implementation?
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**
  - Protected, package-private, or friend-accessible (C++).
  - Not part of the public API.
  - Implementation detail that a subclass/friend may rely on.

- **Private**
  - ???
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)
  - $a + b$  \(\Rightarrow\) String vs. int, double, etc.

- Subtype polymorphism (e.g., method overriding)
  - `Object obj = ...;` \(\Rightarrow\) `toString()` can be overridden in subclasses
    - `obj.toString();` and therefore provide a different behavior.

- Parametric polymorphism (e.g., Java generics)
  - `class LinkedList<E> {` \(\Rightarrow\) 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 = ...; \(\Rightarrow\) 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

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

<table>
<thead>
<tr>
<th>Square</th>
<th>+ drawSquare()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>+ drawCircle()</td>
</tr>
</tbody>
</table>

Good or bad design?
Open/closed principle

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

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

```java
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

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

```java
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

\[
\ldots
A \ a = \text{new} \ D(); \\
\text{int} \ \text{num} = \ a.\text{getNum}(); \\
\ldots
\]

---

```
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 \( T_1 \) and object \( y \) be of type \( T_2 \). Further, let \( T_2 \) be a subtype of \( T_1 \) (\( T_2 <: T_1 \)). Any provable property about objects of type \( T_1 \) should be true for objects of type \( T_2 \).

---

**Rectangle**

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

**Is the subtype requirement fulfilled?**
**Subtype requirement**

Let object \( x \) be of type \( T_1 \) and object \( y \) be of type \( T_2 \). Further, let \( T_2 \) be a subtype of \( T_1 \) (\( T_2 < : T_1 \)). Any provable property about objects of type \( T_1 \) should be true for objects of type \( T_2 \).

<table>
<thead>
<tr>
<th>Rectangle</th>
<th>Rectangle ( r = ) new Rectangle(2,2);</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ width : int</td>
<td>int ( A = r ).getArea();</td>
</tr>
<tr>
<td>+ height: int</td>
<td>int ( w = r ).getWidth();</td>
</tr>
<tr>
<td>+ setWidth(w:int)</td>
<td>r.setWidth(w * 2);</td>
</tr>
<tr>
<td>+ setHeight(h:int)</td>
<td>assertEquals(A * 2, ( r ).getArea());</td>
</tr>
<tr>
<td>+ getArea():int</td>
<td></td>
</tr>
</tbody>
</table>
**Subtype requirement**

Let object $x$ be of type $T_1$ and object $y$ be of type $T_2$. Further, let $T_2$ be a subtype of $T_1$ ($T_2 <: T_1$). Any provable property about objects of type $T_1$ should be true for objects of type $T_2$.

---

**Rectangle**

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

---

Rectangle $r = \text{new Rectangle}(2,2);$  
$\text{new Square}(2);$  

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

---

**Design principles: Liskov substitution principle**
Design principles: Liskov substitution principle

**Subtype requirement**

Let object $x$ be of type $T_1$ and object $y$ be of type $T_2$. Further, let $T_2$ be a subtype of $T_1$ ($T_2 <: T_1$). Any provable property about objects of type $T_1$ should be true for objects of type $T_2$.

```
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!
**Subtype requirement**

Let object $x$ be of type $T_1$ and object $y$ be of type $T_2$. Further, let $T_2$ be a subtype of $T_1$ ($T_2 <: T_1$). Any provable property about objects of type $T_1$ should be true for objects of type $T_2$. 

```
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)

- **Person**
  - **Student**
    - Public class `Student` extends `Person`{
      - `public Student(){ }
      - is-a relationship
      - ...
    }

- **Customer**
  - Public class `Bank`{
    - `public Bank(Customer c){ this.c = c; }`}
  - Public class `Building`{
    - `public Building(){ this.r = new Room(); }`}

- **Room**

**Relationships**
- **is-a**:
  - Student is-a Person
- **has-a**:
  - Bank has-a Customer
  - Building has-a Room
Design choice: inheritance or composition?

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

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

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

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

<table>
<thead>
<tr>
<th>Current</th>
<th>30 day history</th>
</tr>
</thead>
<tbody>
<tr>
<td>25° F</td>
<td></td>
</tr>
<tr>
<td>-3.9° C</td>
<td>min: 20° F max: 35° F</td>
</tr>
</tbody>
</table>

Temp. sensor

Reset history button

Reset
What’s a good design for the view component?

Client sees

25° F
-3.9° C
min: 20° F
max: 35° F

uses

09/01,12°
09/02,14°
...

Client uses

25° F
-3.9° C
min: 20° F
max: 35° F

updates

Client manipulates

Temp. sensor

Reset
Reset history button
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

```java
public void draw(Data d) {
    for (View v : views) {
        v.draw(d);
    }
}
```
The general solution: Composite pattern

```plaintext
<<interface>>
Component
+operation()

CompA
+operation()

CompB
+operation()

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

1..n
The general solution: Composite pattern

1..n

Iterate over all composed components (comps), call operation() on each, and potentially aggregate the results.
What is a design pattern?

- Addresses a recurring, common design problem.
- Provides a generalizable solution.
- Provides a common terminology.
What is a design pattern?

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

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

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

More than just a name for common sense and best practices.
Design patterns: categories

1. Structural
   • Composite
   • Decorator
   • ...

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

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

```java
... 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

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

CompA
 +operation()

CompB
 +operation()

Decorator
 -decorated: Component
 +Decorator(d: Component)
 +operation()
```
Composite vs. Decorator

**Component**
- operation()

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

**CompA**
- operation()

**Decorator**
- decorated: Component
  + Decorator(d: Component)
  + operation()