CS 520
Theory and Practice of Software Engineering
Fall 2022

OO Design Patterns

September 29, 2022
Today

- Recap: OO design principles
- Design patterns:
  - What is a design pattern?
  - Categories of design patterns
  - Structural design patterns
- Final project
Recap: OO design principles

SOLID:

- **Goal: Single responsibility**
  **Solution:** Supported by modularity

- **Goal: Open/closed principle**
  **Solution:** Supported by information hiding (and encapsulation)

- **Goal: Liskov substitution principle**
  **Solution:** Supported by composition/aggregation instead of inheritance
Inheritance and polymorphism

Example:
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Stack.html

1) For the Java inheritance, what are the is-a relationships for the Stack class?

2) What kinds of polymorphism are being used by this class?
   - Ad-hoc
   - Subtype
   - Parametric
Open/Closed principle
(using information hiding)

3) What is a Stack field that is open only for extensions?

4) What is a Stack method that is open only for extensions?
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.

- Supertype T1 is Vector // It is a List.
- Subtype T2 is Stack
- One key property of a List is that the List is represented as an ordered sequence of elements.
- Is this principle: satisfied or violated?
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 $T_2$.

- Supertype $T_1$ is Vector // It is a List.
- Subtype $T_2$ is Stack
- One key property of a List is that the List is represented as an ordered sequence of elements.
- Is this principle: **satisfied** or **violated**?
Composition/Aggregation over inheritance

Another example: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html

- From this class name, the two parts of the functionality are providing Map access and Linked access.

- How is the class implementing the Map access?

- How is the class implementing the Linked access?
Design patterns

- What is a design pattern?
- Categories of design patterns
- Structural 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
Model View Controller: example

Simple weather station

<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</td>
</tr>
<tr>
<td></td>
<td>max: 35° F</td>
</tr>
</tbody>
</table>

Reset

Reset history button

Model

01/01 -> 0
01/02 -> -5
01/03 -> -10
01/04 -> -4
...

Temp. sensor

Controller
What’s a good design for the view?

Client sees

25° F
-3.9° C

min: 20° F
max: 35° F

Reset
Reset history button
Temp. sensor

09/01, 12°
09/02, 14°
...

Client uses

Client manipulates

Client updates
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)

How do we need to implement draw(d:Data)?
public void draw(Data d) {
    for (View v : views) {
        v.draw(d);
    }
}
Design pattern: Composite

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

1..n

CompA
+operation()

CompB
+operation()

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

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

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

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

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

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

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

1. Structural
   ○ Composite
   ○ Decorator
   ○ ...

2. Behavioral
   ○ Template method
   ○ Visitor
   ○ ...

3. Creational
   ○ Singleton
   ○ Factory (method)
   ○ ...
Another design problem: I/O streams

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

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

```java
<<interface>>
InputStream

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

```java
FileInputStream

+read():int
+read(buf:byte[]):int
```
Another design problem: I/O streams

InputStream is = new FileInputStream(...);

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

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

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[])`. 
Design pattern: Decorator

<<interface>>
Component
+operation()

CompA
+operation()

CompB
+operation()

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

<<interface>>
Component

1..n
- comps: Collection<Component>
+ operation()

1

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

CompA
+ operation()

Decorator
- decorated: Component
+ Decorator(d: Component)
+ operation()
Find the median in an array of doubles

Examples:
- median([1, 2, 3, 4, 5]) = ???
- median([1, 2, 3, 4]) = ???
Find the median in an array of doubles

Examples:
- median([1, 2, 3, 4, 5]) = 3
- median([1, 2, 3, 4]) = 2.5

Algorithm
Input: array of length $n$      Output: median
Find the median in an array of doubles

Examples:
- median([1, 2, 3, 4, 5]) = 3
- median([1, 2, 3, 4]) = 2.5

Algorithm

**Input:** array of length \( n \)  
**Output:** median

1. Sort array
2. if \( n \) is odd return \(((n+1)/2)\)th element
   otherwise return arithmetic mean of \((n/2)\)th element and \(((n/2)+1)\)th element
Median computation: naive solution

```java
public static void main(String ... args) {
    System.out.println(median(1,2,3,4,5));
}

public static double median(double ... numbers) {
    int n = numbers.length;
    boolean swapped = true;
    while(swapped) {
        swapped = false;
        for (int i = 1; i<n; ++i) {
            if (numbers[i-1] > numbers[i]) {
                ...
                swapped = true;
            }
        }
    }
    if (n%2 == 0) {
        return (numbers[(n/2) - 1] + numbers[n/2]) / 2;
    } else {
        return numbers[n/2];
    }
}
```

What's wrong with this design? How can we improve it?
Ways to improve
One possible solution: **template method pattern**

```
AbstractMedian
{abstract}
+ median(a:double[]):double
  # sort(a:double[])

SimpleMedian
# sort(a:double[])
```

Italics indicate an abstract method.
One possible solution: template method pattern

AbstractMedian

{abstract}

+ median(a:double[]):double

# sort(a:double[])

- The template method (median) implements the algorithm but leaves the sorting of the array undefined.

- The concrete subclass only needs to implement the actual sorting.
One possible solution: template method pattern

AbstractMedian

   {abstract}

   + median(a:double[]):double
   # sort(a:double[])

SimpleMedian

   # sort(a:double[])

Should the median method be final?

- The template method (median) implements the algorithm but leaves the sorting of the array undefined.

- The concrete subclass only needs to implement the actual sorting.
Another solution: *strategy pattern*

```
<<interface>>
Median
+median(a:double[]):double

<<interface>>
Sorter
+sort(array:double[])
```

```
StrategyMedian
-sortStrategy: Sorter
+median(a:double[]):double
+setSorter(s:Sorter)
```

```
HeapSort
+sort(...)
```

```
QuickSort
+sort(...)
```

"median" delegates the sorting of the array to a "sortStrategy"
Template method pattern vs. strategy pattern

Two solutions to the same problem

What are the differences, pros, and cons?
Template method pattern vs. strategy pattern

Two solutions to the same problem

Template method
- Behavior selected at compile time.
- Template method is usually final.

Strategy
- Behavior selected at runtime.
- Composition/aggregation over inheritance.
Model-View-Controller revisited

Design patterns in a MVC architecture

Client sees View

Client uses Controller

View updates Model

Controller manipulates Model
Model-View-Controller revisited

Design patterns in a MVC architecture

Composite

View

Controller

Model

Client sees

uses

updates

manipulates

Strategy/Template method
Model-View-Controller revisited

Design patterns in a MVC architecture

Client sees

View

uses

Controller

updates

Model

???
Observer pattern

Observer pattern
From Wikipedia, the free encyclopedia

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

- Problem solved:
  - A one-to-many dependency between objects should be defined without making the objects tightly coupled.
  - When one object changes state, an open-ended number of dependent objects are updated automatically.
  - One object can notify an open-ended number of other objects.
Observer pattern

Observable

{abstract}

# observers: Set<Observer>
+ register(o: Observer)
+ unregister(o: Observer)
+ stateChanged()

MyObservable

- state: State
+ getState(): State
+ setState(state: State)

Observer

<<interface>>

+ update()

MyObserver

+ update()

public void stateChanged() {
    for (Observer o : observers) {
        o.update();
    }
}

// For the setState method, use the stateChanged method
Variations of the Observer update method

- `update(state:State)`
  - Alternatively, could decompose the State into pieces

- `update(observable:Observable)`
  - Use the Observable `getState` method(s)
Example: Observer pattern for MVC

1) Which is the Observable? Model or View

2) Which is the Observer? Model or View

3) Which class should use the getState method
   Model, View, or Controller

4) Which class should use the setState method?
   Model, View, or Controller
Model-View-Controller revisited

Design patterns in a MVC architecture

Client sees View

Client uses Controller

View updates Model

Controller manipulates Model

Composite

Observer

Strategy/Template method
Final project selection

- Form team of 2, 3, or 4 students

- Select one of the following 3 topics:
  1. Research (e.g., MSR mining challenge)
  2. Experimental evaluation (e.g., ML development toolkit comparison)
  3. Development (e.g., EleNa: Elevation-based navigation)

  NOTE) Could proposed your own project for one of the above topics

- Due: Thursday October 6, 2022 11:59 PM
Final project: Selected topic

1. Read some background materials (e.g., papers, user manuals, code)
2. Start to develop
3. Create and give a mid-point presentation
4. Continue to develop
5. Create and give a final presentation or demonstration
6. Put together final deliverables

NOTE) All groups need to create a version control repository for their final project and share it with the course support (instructor, TA, graders).
Final project:
MSR mining challenge objectives

- Read 8-10 papers
- Select one or more research questions
- Propose an approach to investigate the research question(s)
- Develop experiments to evaluate the proposed approach by applying to the provided dataset
- Write up the approach and experimental results

https://2021.msrconf.org/track/msr-2021-mining-challenge

https://conf.researchr.org
/track/msr-2022/msr-2022-mining-challenge
Final project:
Replication study objectives

● Read 4-5 papers
● Select a tool that applies SE topics (e.g., SOSRepair)
  ○ Learn about the benchmark (i.e. initial data) for the selected tool
● Replicate the experiments to evaluate the selected tool by applying to the initial data
  ○ Extend the experiments to further evaluate the selected tool on additional data
● Write up the replication study
  ○ Including a manual review of the tool(s)
Final project: Tool comparison study

- Read any necessary documentation
- Select one or more tools that apply SE topics (e.g., ML development toolkits such as wandb.ai)
  - Also select one or more data sets
- Develop experiments to evaluate the selected tools by applying to the selected data set(s)
- Write up the experimental results
  - Including a code review of the tool(s) along with possibly a UI design review

https://wandb.ai/site
Final project:
Development project objectives - Elevation-based Navigation (EleNa)

- **Goal:** Develop a software system that determines, given a start and an end location, a route that maximizes or minimizes elevation gain, while limiting the total distance between the two locations to x% of the shortest path

- **Components:**
  - Data model that represents the geodata
  - A component that populates the data model, querying, e.g., [https://www.openstreetmap.org](https://www.openstreetmap.org)
  - The actual routing algorithm that performs the multi-objective optimization
  - Another component that outputs or renders the computed route
Final project: Development

- Read any necessary technical documents
- Design main components
- Implement the designed components
- Build a test plan for the implemented components and carry out that test plan
- Could additionally perform other evaluation of the system (e.g., accessibility, usability)
- Demo at the final presentation
- Briefly write up a design document (e.g., requirements, architecture, design) as well as the evaluation (e.g., testing results, usability survey results)