CSE12 - Lecture 25 - C00

PA 6 Late/Resubmit → due tomorrow
PA7/PA8 Late/Resubmit → due Friday
Final Exam → Saturday 8am
Wednesday's Lecture → Extra Credit → take quiz at the beginning

Composition instead of Inheritance

Design Patterns
https://en.wikipedia.org/wiki/Design_Patterns
https://en.wikipedia.org/wiki/Software_design_pattern

Familiar Design Patterns

Iterator - Provide a way to access the elements of an object sequentially without exposing its underlying representation.

Adapter (Wrapper) Pattern - Convert the interface of a class into another interface clients expect.

Queues/ Stacks → Array List

Object Pool - Avoid expensive acquisition and release of resources by recycling objects that are no longer in use.

Factory Method - create objects by calling a factory method rather than by calling a constructor.

Lazy Initialization - Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.

Singleton - Ensure a class has only one instance, and provide a global point of access to it.

Observer or Publish/subscribe - Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
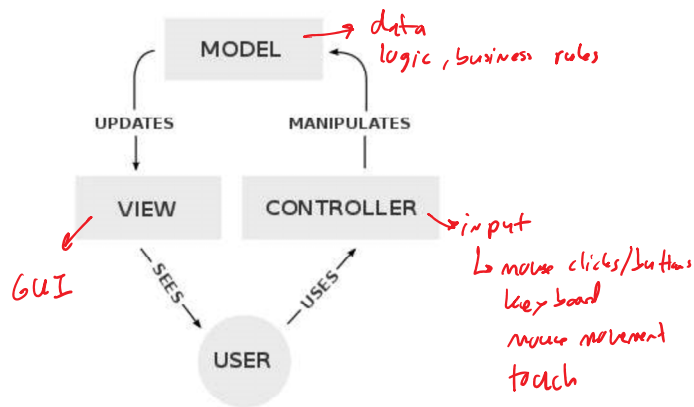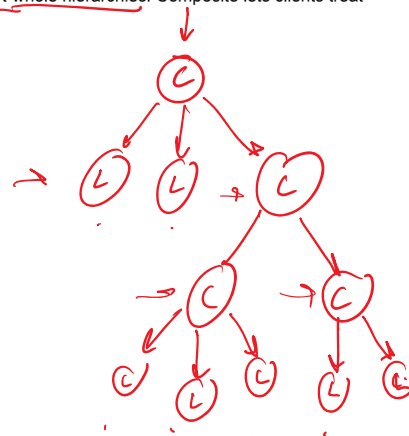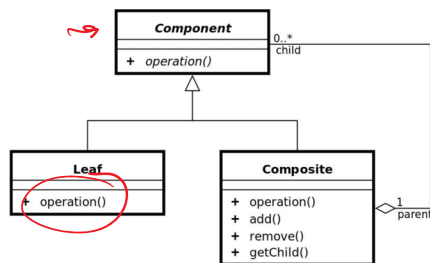
button

Null object	Avoid null references by providing a default object.

Model–view–controller - Commonly used for developing user interfaces that divide the related program logic into three interconnected elements (became popular for designing web applications)
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

MODEL → data
logic, business rules

UPDATES   **MANIPULATES**

VIEW   **CONTROLLER** → input
↳ mouse clicks/buttons
keyboard
mouse movement
touch

GUI

— SEES —   — USES —

USER

Composite - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Component
+ operation()

0..* child

Leaf
+ operation()

Composite
+ operation()
+ add()
+ remove()
+ getChild()

1 parent

```java
class Node<T> {
    T value;
    Node<T> next;
    public Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
```

(handwritten annotations in red/blue):

private ~~public~~ Node(T value, Node<T> next) {

```
private static ArrayList<Node<T>> pool = new ArrayList<>();
public static Node<T> createNode (T value, Node<T> next) {
    if (pool.size() > 0) { return pool.remove(0); }
    return new Node (value, next);
}
public static void removeNode ( Node<T> Node) {
    pool.add(node);
}
```

```
Node<T> Node = pool.remove(0);
Node.value = value;
Node.next = next;
return Node;
```

```java
public class LList<E> implements List<E> {
    Node<E> front;
    int size;

    public LList() {
        this.front = new Node<E>(null, null);
    }
```

Node<E>? Node.createNode (null, null);

```java
    public void prepend(E s) {
        this.front.next = new Node<E>(s, this.front.next);
        this.size += 1;
    }
```

Node.createNode (s, this.front.next);

```java
    public void remove(int index) {
        Node<E> current = this.front;
        for(int i = 0; i < index; i += 1) {
            current = current.next;
        }
        current.next = current.next.next;
        this.size -= 1;
    }
```

Node.removeNode (current.next);

```java
    public void add(E s) {
        Node<E> current = this.front;
        while(current.next != null) {
            current = current.next;
        }
        current.next = new Node<E>(s, null);
        this.size += 1;
    }
}
```

Node.createNode (s, null);

```java
class SingleObject {
    private static SingleObject singleton;

    private
    public SingleObject() {
        //initialization
    }
    public static SingleObject get() {
        if (singleton == Null) {
            singleton = new SingleObject();
        }
        return singleton;
    }
}
```

```java
interface SomeEvent {
    public void fire();
}
```

```java
class SomeEventHandler implements SomeEvent {
    public void fire() {
        System.out.println("SomeEventHandler does some stuff").
    }
}
```

```java
class OtherEventHandler implements SomeEvent {
    public void fire() {
        System.out.println("OtherEventHandler does some stuff").
    }
}
```

```java
class Worker {
    List<SomeEvent> handlers;

    void listen(SomeEvent handler) {
        handlers.add(handler);
    }
    //void unlisten(SomeEvent handler) {}

    void actionHappened() {
        for (SomeEvent handler: handlers) {
            handler.fire();
        }
    }
}
```

```
SingleObject  obj = SingleObject.get();
```

```
SomeEvent evt1 = new SomeEventHandler();
SomeEvent evt2 = new OtherEventHandler();

Worker worker = new Worker();

worker.listen(evt1);
worker.listen(evt2);

worker.run();
```

```
void run() {
    while (   )

    if (  ) {
        actionHappened();
    }
}
```