

CSE12 - Lecture 25

Monday, December 4, 2023 8:00 AM

PA7 Late/Resubmit - due Wednesday @ 8am

PA8 Late/Resubmit due Friday of Week 10 @ 8am

Exam 3 - next Wednesday - Trees, BST, Heaps, Iterators, Improving Lists

- No design patterns

Final Exam - Monday @ 8am - Room change - Peterson 108

Student Evaluation of Teaching (SET)

- Please submit your SETs for the course at

<https://academicaffairs.ucsd.edu/Modules/Evals> by Saturday at 8am

Lecture 25

Design Patterns

https://en.wikipedia.org/wiki/Design_Patterns

https://en.wikipedia.org/wiki/Software_design_pattern

Composition instead of Inheritance
↳ usually interfaces
↳ Dependency inversion

Familiar Design Patterns

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

Iterable, Iterator

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

Stacks/Queues → Array List

delegate the method calls

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

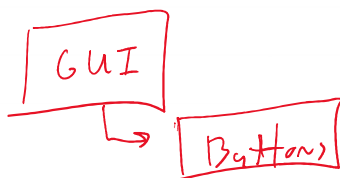
Factory Method - create objects by calling a factory method rather than by calling a constructor.

↳ Abstract Factory / Builder

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
↳ register an event
↳ method called when button is clicked

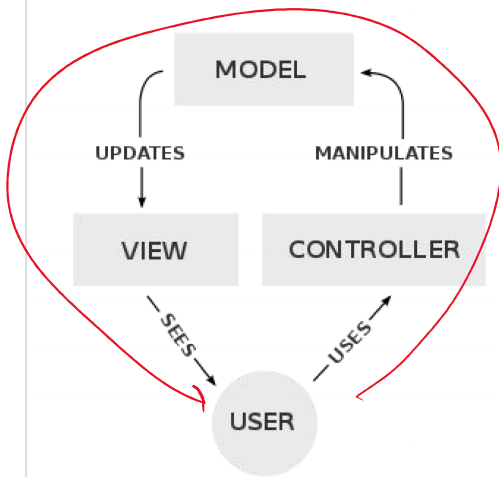
Null object

Avoid null references by providing a default object.

Name: _____ PID: _____ Code: _____

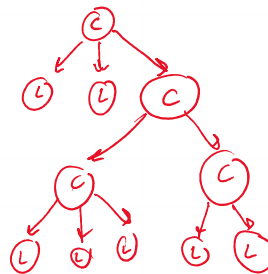
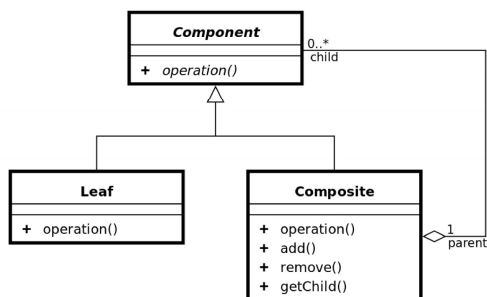
Model-view-controller - Commonly used for developing user interfaces that divide the related program logic into three interconnected elements (became popular for designing web applications)

<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>



Model view presenter
MVP

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



UML

Object Pool, Factory Method

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

private static Node<T> pool = new ArrayList<>();

public static Node<T> createNode(T value, Node<T> next) {
    if (pool.size() > 0) {
        Node<T> node = pool.remove(0);
        node.value = value;
    }
}
```

```

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

public static void removeNode(Node<T> node) {
    pool.add(node);
}

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

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

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

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

    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;
    }
}

```

Singleton / Factory method / Lazy Initialization

```

class SingleObject {
    private static SingleObject singleton;

    private SingleObject() {
        //initialization
    }

    public static SingleObject get() {
        if (singleton == null) {
            singleton = new SingleObject();
        }
        return singleton;
    }
}

```

SingleObject obj = SingleObject.get();

Observer

```

interface SomeEvent {
    public void fire();
}

```

```

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

```

```

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

```

```

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

```

```

class Worker {
    List<SomeEvent> handlers;
    registration/subscribe
    void listen(SomeEvent handler) {
        handlers.add(handler);
    }
    unregister
    //void unlisten(SomeEvent handler) {}

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

```

SomeEvent evt1 = new SomeEventHandler();
SomeEvent evt2 = new OtherEventHandler();
Worker worker = new Worker();
worker.listen(evt1);
worker.listen(evt2);
Workers.run();

```

void run() {
    while (true) {
        if (true) {
            actionHappened();
        }
    }
}

```