CSE12 - Lecture 25

Monday, June 3, 2024 10:00 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
- https://ucsd-cse12-sp24.github.io/lectures/exam3.html

Final Exam - Monday @ 8am - Same room
- https://ucsd-cse12-sp24.github.io/lectures/final-exam.html

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.

Iterator, Iterable

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

Stacks/Queues → ArrayList

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.

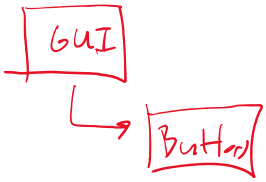
↳ Abstract Factory / Builder

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.

event handlers

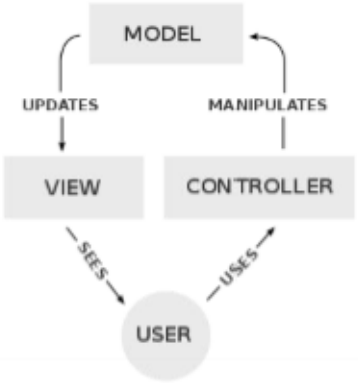


Button
↳ register an event
↳ method called when button is clicked

Name: _____ PID: _____ Code: 4550

Null object	Avoid null references by providing a default object.

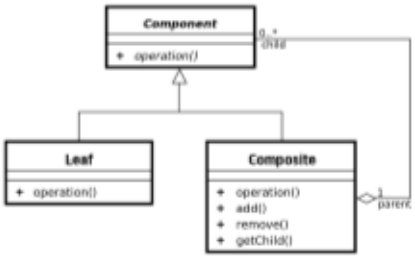
Model–view–controller - Commonly used for developing user interfaces that divide the related program logic into three interconnected elements (became popular for designing web applications)
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller



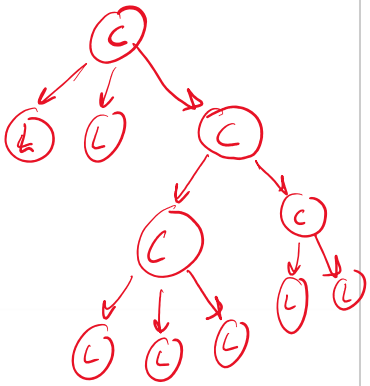
MVC

MVP → model view presenter

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, Singleton

```
class Node<T> {
  T value;
  Node<T> next;
  public Node(T value, Node<T> next) {
    this.value = value;
    this.next = next;
  }
}
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) {
    this.front.next = new Node<E>(s, this.front.next);
    this.size += 1;
  }

  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;
  }

  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;
  }
}
```

```
private
public static removeNode(Node<T> node) {
    pool.add(node);
}
private static ArrayList<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);   // size-1
        node.value = value;
        node.next = next;
        return node;
    }
    else {
        return new Node<>(value, next);
    }
}
```

Node.createNode(null, null);

Node.createNode(s, this.front);

Node.removeNode(current.next);

Node.createNode(s, null);

Singleton / Factory Method / Lazy Initialization

```
class SingleObject {
   private static SingleObject singleton;

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

SingleObject obj = SingleObject.get();

Observer pattern

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

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;

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

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

SomeEvent ev1 = new SomeEventHandler();
SomeEvent ev2 = new OtherEventHandler();

Worker worker = new Worker();
worker.listen(ev1);
worker.listen(ev2);

worker.run();

↳ void run() {
  while (---) {
    :
    if (true) {
      actionHappened();
    }
  }