CSE12 - Lecture 25

Monday, June 5, 2023 8:00 AM

PA6 Late/Resubmit → due tomorrow
PA7/PA8 Late/Resubmit → due Friday (no slip day)

Final → Fri 6/16

Cape survey

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.

Stack/Queues → Array List delegate to method calls

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

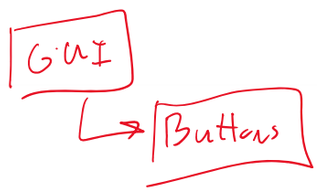
Factory Method - create objects by calling a factory method rather than by calling a constructor.

↳ 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.

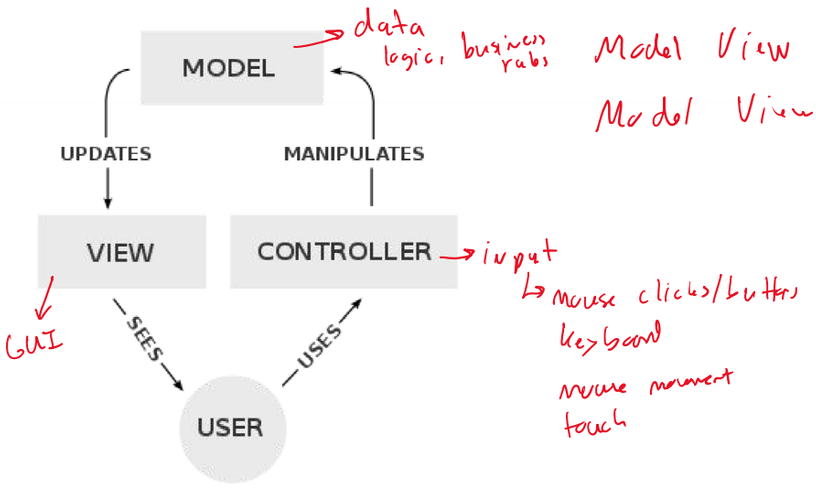


register method

Null object	Avoid null references by providing a default object.

Name: _____ PID: _____ Code: 8003

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



data logic, business rules

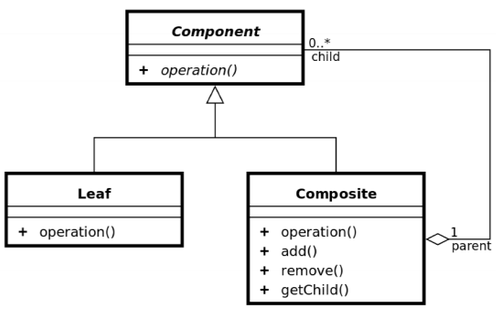
Model View Presenter

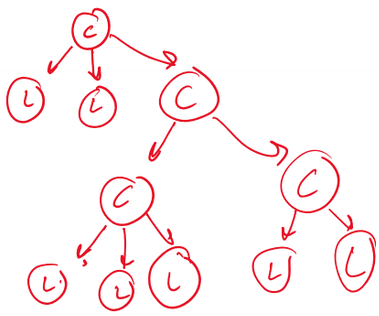
Model View View Model

input
↳ mouse clicks/buttons
keyboard
mouse movement
touch

GUI

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





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

```
public 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<T>(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;
}
```

```
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);
```

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

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

```
  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;
  }
}
```

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

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;
  }
```

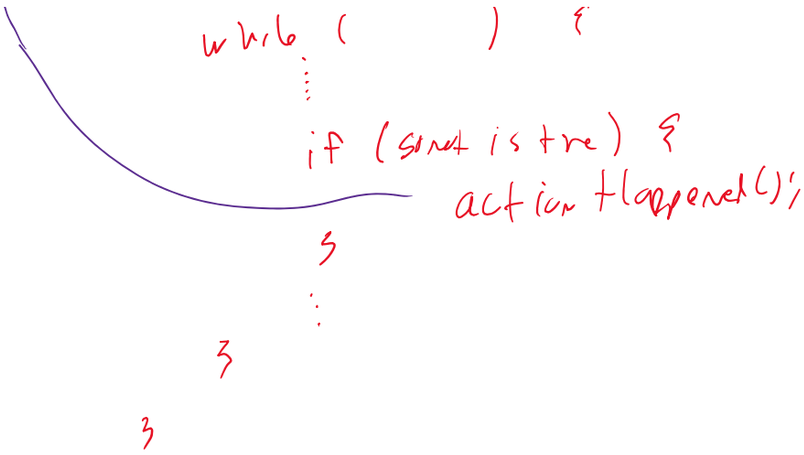
```
SingleObject obj = SingleObject.get();
```

Observer

```
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; = new ...
      register
  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);

worker.run();
```

```
void run() {
    while (          ) {
        ⋮
```


while ( ) {
if (sent is true) {
actionHappened();
}
}
}