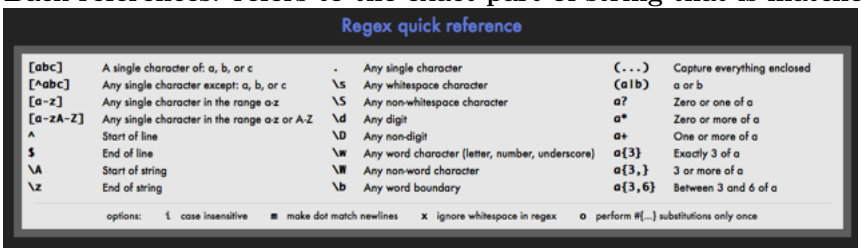
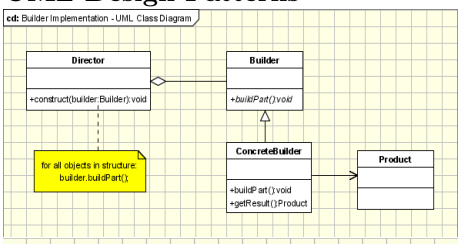
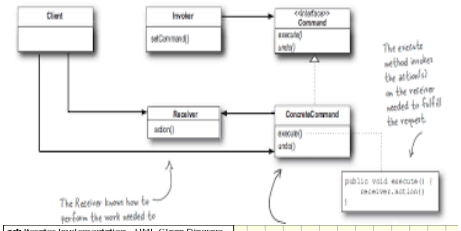
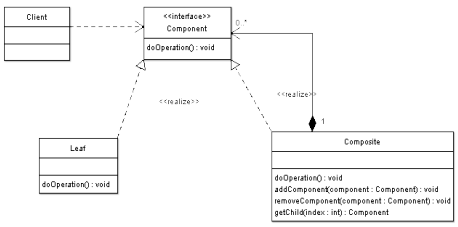
Back references: refers to the exact part of string that is matched to the group number not matched to the pattern

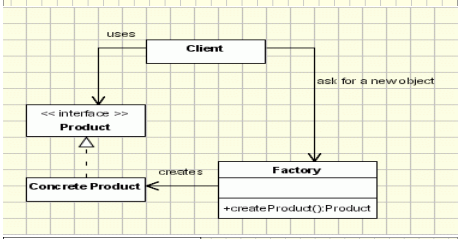


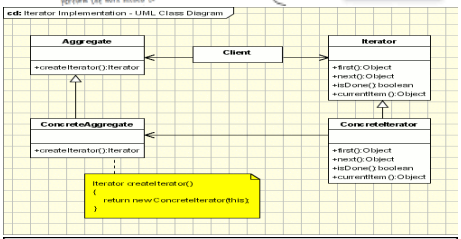
UML Design Patterns











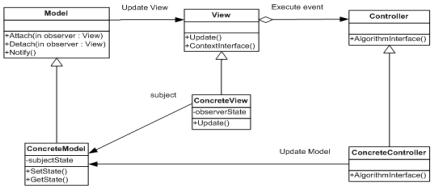
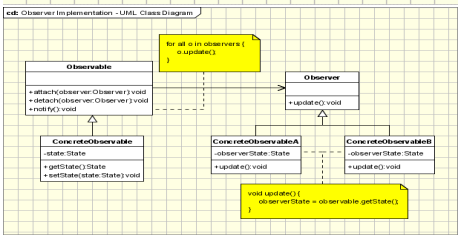
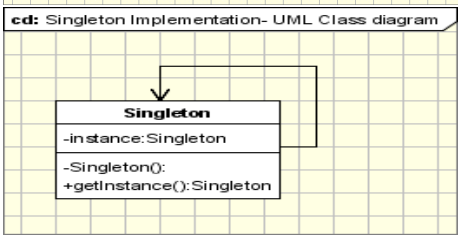
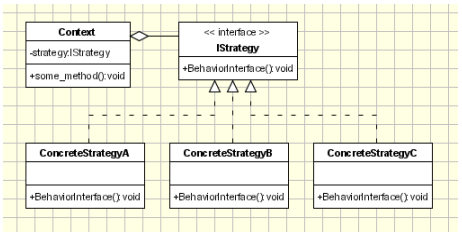


Figure 3: MVC







Explanation of UML diagrams

Git Basics git pull [location name]
git add [file name] or . (adds all)
git commit -m (message)
git push [file name]
git checkout [branch name] – switches branches
git branch [branch name] – creates new branch or checks which branches exist.
git log – shows log of past push history
git clone [location]
git merge [file name]
git branch -D [branch] Force delete the specified branch
git branch -d [branch] Delete the specified branch. prevents you from deleting the branch if it has unmerged changes.
git branch -m [branchName] rename branch to branchName

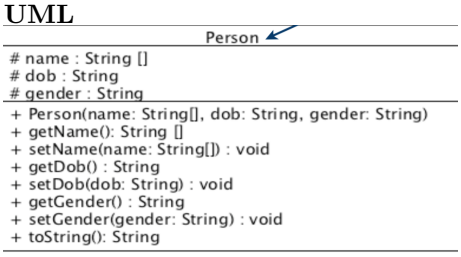
```
public class fileParser(){
    private Pattern pname = Pattern.compile("regex");
    public boolean parse(BufferReader input){
    try{int state 0; Matcher m; String L;
        while((L = input.readLine()) != null) {
            switch(state){
                case 0:
                    m = pname.matcher(L);
                    if(m.matches()){int p1 = Integer.parseInt(m.group(1));state = 1; break;}
                    error("message";
                    return false;
                }#state
            }#while
        }#try
        catch(Exception e){}
        return true;
    }#method
}#class
```

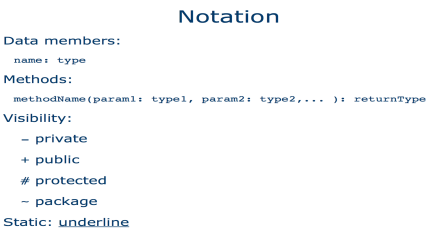
Inheritance

In a subclass:
-use super.attribute to refer to a variable or method in parent class
-use super(attribute) to call a constructor defined in parent class

```
public class LandAnimal extends Animal{
    public LandAnimal(String name){super(name);}}
```

UML





Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default (package private)	Yes	Yes	No	No
private	Yes	No	No	No

Builder Design Pattern Example:

```
public static void main(String[] args){
   Director director = new Director();
   Builder builder = Null;
   Scanner s = new Scanner(System.in);
   String ans = s.nextline();
   if(ans.equals("kid"){builder = new Kidsmealbuilder();}
   else{builder = new Studentmealbuilder();}
   Meal meal = director.createMeal(builder)}

public abstract class MealBuilder {
   protected Meal meal = new Meal();
   public abstract void buildDrink();
   public abstract void buildMain();
   public abstract Meal getMeal();}

public class director{
   #no constructor
   public Meal createMeal(Mealbuilder builder){ builder.buildDrink(); builder.buildFood();
   return builder.getMeal();}

public class KidsMealBuilder extends MealBuilder{
   public void buildDrink(){meal.setDrink("Kid drink: Kool-aid");}
   public void buildMain(){meal.setMain("Chicken nuggets");}
   public Meal getMeal(){return meal;}
```

Singleton Design Pattern

```
public class Client(){
      Singleton S1 = Singleton.getInstance();
      Singleton S2 = Singleton.getInstance(); #S1 and S2 are the same instance of the singleton object}
public class Singleton(){
   private static Singleton instance = new Singleton();
   public static Singleton getInstance() {
        return instance;}}
```

Command Design Pattern

```
public interface Command {public void execute();
}
public class TurnOnCommand implements Command {
Light light;
public TurnOnCommand(Light light) { this.light = light; }

@Override
public void execute() {
this.light.switchOn(); }}

***TURN OFF IS THE SAME THING FAM***

public class RemoteControl {
private Command currentCmd;
public void setCommand(Command cmd) { currentCmd = cmd; }
public void pressButton() { this.currentCmd.execute(); }
}
public class Light {
private boolean on = false;
public void switchOn() {on = true; System.out.println("Turned on"); }
public void switchOff() {on = false;System.out.println("Turned off"); }
}
public class Client {
public static void main(String[] args) { Light light = new Light();
RemoteControl control = new RemoteControl();
Command lightsOn = new TurnOnCommand(light);
Command lightsOff = new TurnOffCommand(light);
// switch on
control.setCommand(lightsOn);
control.pressButton();
// switch off
control.setCommand(lightsOff);
control.pressButton();
}}
```

Strategy Design Pattern:

```
public interface TravelStrategy {
   public void travel(Person p, String location);}
public class CarStrategy implements TravelStrategy {
   public void travel(Person p, String location) {
      p.setLocation(location);
      System.out.println(p.getName() + " has traveled to " + p.getLocation() + " by car.");}}
public class TravelContext {
   private TravelStrategy strategy;
   public void setTravelStrategy(TravelStrategy s){strategy =s;}
```

Factory Design Pattern:

```
public abstract class Fruit {
final String type;
public Fruit(String type) { this.type = type; }
public String getType() {return type; }
}
public class Apple extends Fruit {
public Apple() { super("Apple"); }
}
***Orange is the same thing as Apple***

public class FruitFactory {                    //this has no constructor
public Fruit makeFruit(String type) {
Fruit fruit = null;
if (type == "Apple") {fruit = new Apple(); }
else if (type == "Orange") { fruit = new Orange(); }
return fruit; }
}
```

```
public class Main {
public static void main(String[] args) {
Fruit fruit;
FruitFactory fruitFactory = new FruitFactory();
fruit = fruitFactory.makeFruit("Apple");
System.out.println("The fruit is an " + fruit.getType());
fruit = fruitFactory.makeFruit("Orange");
System.out.println("The fruit is an " + fruit.getType()); }
}
```

Product Owner: Responsible for product backlog, represents users, expresses backlog items and orders them by value \\
Development Team: Responsible for delivering potentially shippable increment of working software \\
Scrum Master: Removes obstacles, facilitates scrum events and communication \\
Product Backlog: Source of requirements for any changes to be made to the product. Ordered by value, risk, priority and necessity, estimated by team
Sprint planning meeting: Team selects items from backlog and defines a sprint goal, and the items are converted into tasks and estimated \\
Daily Scrum Meeting: Short meeting for the team to discuss what has been accomplished since last meeting, what will be done before the next meeting, and what obstacles are in the way \\
Sprint Review: Product Owner identifies what has been done, team discusses development process and demos current increment of software, product owner discusses current state of backlog, team decides what to do next \\

IEEE Conversion

$$\underbrace{0}_{+or-}\ \underbrace{01111110}_{8bit}\ \underbrace{01000000000000000000000}_{23bitmantissa}$$

rep exponent + 127

Rounding GRE

round up if mantissa's bit just before G is 1, else round down/do nothing.
101 - round up
110 - round up
111 - round up
Rounding up is done by adding 1 to the mantissa in the mantissa's least significant bit position just before G. G is the 1st element after the 23 mantissa.

Example for float -6.8

6: $2^2 + 2^8 <=> 110$

$$0.8 : 1100... \quad \{ \begin{array}{ll} 0.8 * 2 & = \mathbf{1}.6 \\ 0.6 * 2 & = \mathbf{1}.2 \\ 0.2 * 2 & = \mathbf{0}.4 \\ 0.4 * 2 & = \mathbf{0}.8 \end{array}$$

mantissa = 110.1100 1100 1100 1100 1100 1 (24 since you don't count the first one)
Normalize mantissa to find exponent: 1.10(.)11001100110011001 * 2^2 (Shifted 2 decimal places so exponent is 2)

8 bit exponent: Binary of 127 + exponent value = 127 + 2 = 129 = $2^7 + 2^0 <=>$ 10000001

The IEEE is:

$$\underbrace{1}_{SinceNegative}\ \underbrace{10000001}_{8bit}\ \underbrace{10110011001100110011001}_{23bitmantissa}$$

Round up since GRS is 100 and the element before G is a 1. So its actually since 001 represents 1 in binary so 1+1 = 2 and the binary rep is 010.

Final IEEE: $\underbrace{1}\ \underbrace{10000001}_{8bit}\ \underbrace{10110011001100110011010}_{23bitmantissa}$

Composite Design Pattern

```
public abstract class SheepComponent {public abstract void sheer();
}

public class Sheep extends SheepComponent {
String name;
public Sheep(String name) {this.name = name; }
public String getSheepName() { return name; }
@Override
public void sheer() { System.out.println("Sheering " + getSheepName() + "...\n"); }
}

public class SheepGroup extends SheepComponent {
String groupName;
ArrayList<SheepComponent> sheepComponents;
public SheepGroup(String name) {sheepComponents = new ArrayList<SheepComponent>(); groupName = name; }
public String getGroupName() {return groupName; }
public void add(SheepComponent sheepComponent) { sheepComponents.add(sheepComponent); }
public void remove(SheepComponent sheepComponent) { sheepComponents.remove(sheepComponent); }
public SheepComponent getComponent(int index) { return sheepComponents.get(index); }
@Override
public void sheer() {
int numOfSheep = sheepComponents.size();
System.out.println("Group Name: " + groupName + "\n" + "---" + "\n");
for (int i = 0; i < numOfSheep; i++) { sheepComponents.get(i).sheer(); }
}}
```

```
public class SheepExample {
public static void main(String agrs[]) {
SheepGroup sg1 = new SheepGroup("Sheep Group 1");
Sheep s1 = new Sheep("Sheep 1");
Sheep s2 = new Sheep("Sheep 2");
Sheep s3 = new Sheep("Sheep 3");
sg1.add(s2);
sg1.add(s3);
s1.sheer();
sg1.sheer();
}
}
```

Iterator Design Pattern

```
public class Song {
String name;
String artist;
public Song(String name, String artist) {
this.name = name;
this.artist = artist;
}

public String getName() { return name; }
public String getArtist() { return artist; }
public String toString() { return ("Name: " + this.getName() +" Artist: " +
 this.getArtist());}
}

public class MySongs implements Iterable<Song> {
HashMap<Integer, Song> mySongs;
public MySongs() {
mySongs = new HashMap<Integer, Song>();
mySongs.put(0, new Song("Kingdom Hearts Theme Song", "Utada Hikaru"));
mySongs.put(1, new Song("Sephiroth's Theme Song", "Nobuo Uematsu"));
mySongs.put(2, new Song("Let it go", "Idina Menzel"));
}
@Override
public Iterator<Song> iterator() { return new MySongsIterator(mySongs);}
}

public class MySongsIterator implements Iterator<Song> {
private HashMap<Integer, Song> songs;
private int indexKey;
public MySongsIterator(HashMap<Integer, Song> s) { this.songs = s;
indexKey = 0; }
@Override
public boolean hasNext() {return this.indexKey < this.songs.size(); }
@Override
public Song next() {return this.songs.get(indexKey++); }
}

public class SongsMain {
public static void main(String[] args) {
YourSongs songs1 = new YourSongs();
MySongs songs2 = new MySongs();
for (Song s: songs1) {System.out.println(s); }
for (Song s: songs2) {System.out.println(s); }
// the above is the same as:
Iterator<Song> it = songs1.iterator();
while (it.hasNext()) {System.out.println(it.next());}
}}
```

Observer Design Pattern

```
public class Parcel extends Observable {
    private String trackingNumber;
    private String location;
    public Parcel(String trackingNumber, String location) {
        this.trackingNumber = trackingNumber;
        this.location = location; }
    @Override
    public String toString() {
        return "Parcel has" + trackingNumber + "."; }
    public void updateLocation(String newLocation) {
        location = newLocation;
        this.setChanged();
        this.notifyObservers("Updated location to " + location); }
}
public class Customer implements Observer {
    private String name;
    public Customer(String name) {
        this.name = name; }
    @Override
    public String toString() {return name; }

    @Override
    public void update(Observable o, Object arg) {
        System.out.println("Customer " + this.name + " observed a change in " + o);
        System.out.println("   The notification said: " + arg); }
}

***Company is the same thing as Customer cuz***

public class Main {
    public static void main(String[] args) {
        Customer sadia = new Customer("Sadia");
        Parcel order = new Parcel("ASDF", "Mississauga");
        order.addObserver(sadia);
        order.updateLocation("Toronto"); }
}
```

FAM GOODLUCK!!! YOU GOTs THIS

```
public class AssertTests {
  @Test
  public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    assertArrayEquals("failure - byte arrays not same", expected, actual);}
  @Test
  public void testAssertEquals() {
    assertEquals("failure - strings are not equal", "text", "text"); }

  @Test
  public void testAssertFalse() {
    assertFalse("failure - should be false", false); }

  @Test
  public void testAssertNotNull() {
    assertNotNull("should not be null", new Object()); }

  @Test
  public void testAssertNotSame() {
    assertNotSame("should not be same Object", new Object(), new Object()); }

  @Test
  public void testAssertNull() {
    assertNull("should be null", null); }

  @Test
  public void testAssertSame() {
    Integer aNumber = Integer.valueOf(768);
    assertSame("should be same", aNumber, aNumber); }
```

```
  // JUnit Matchers assertThat
  @Test
  public void testAssertThatBothContainsString() {
    assertThat("albumen", both(containsString("a")).and(containsString("b")));
  }

  @Test
  public void testAssertThatHasItems() {
    assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
  }

  @Test
  public void testAssertThatEveryItemContainsString() {
    assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }),
everyItem(containsString("n")));
  }

  // Core Hamcrest Matchers with assertThat
  @Test
  public void testAssertThatHamcrestCoreMatchers() {
    assertThat("good", allOf(equalTo("good"), startsWith("good")));
    assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
    assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
    assertThat(7, not(CombinableMatcher.<Integer>
either(equalTo(3)).or(equalTo(4))));
    assertThat(new Object(), not(sameInstance(new Object())));
  }

  @Test
  public void testAssertTrue() {
    assertTrue("failure - should be true", true);
  }
}
```