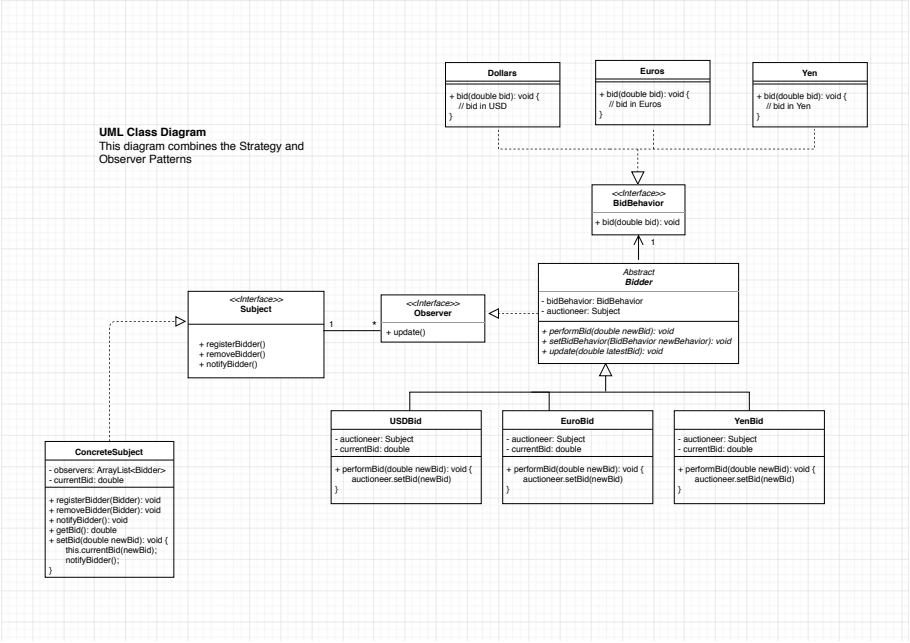
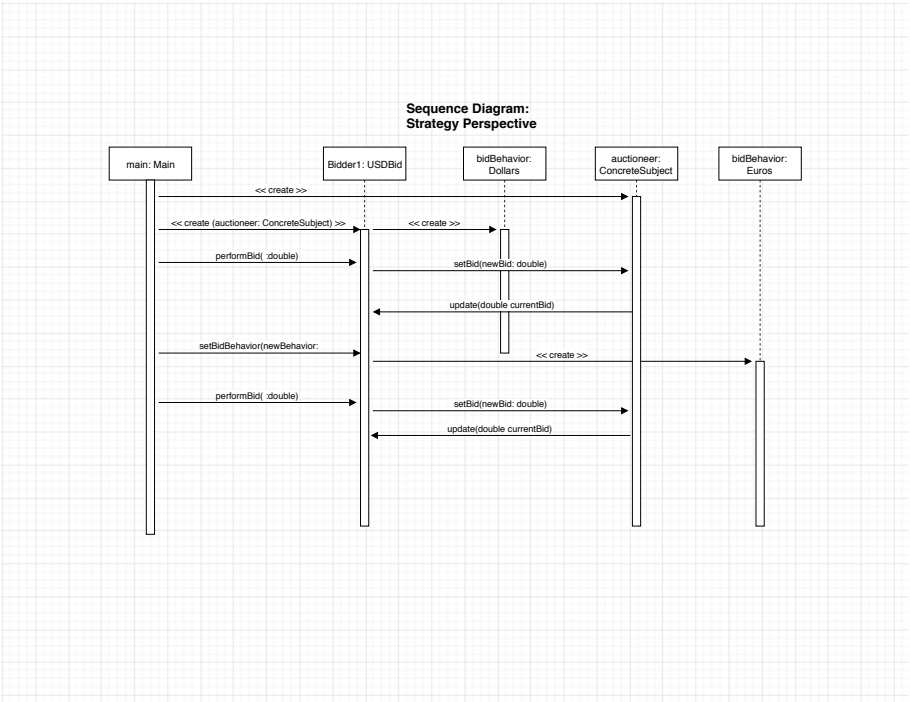
ESOF322 - HW3
Joe Akers, Brad McCoy
Septembery 26, 2019

1 Exercise 1

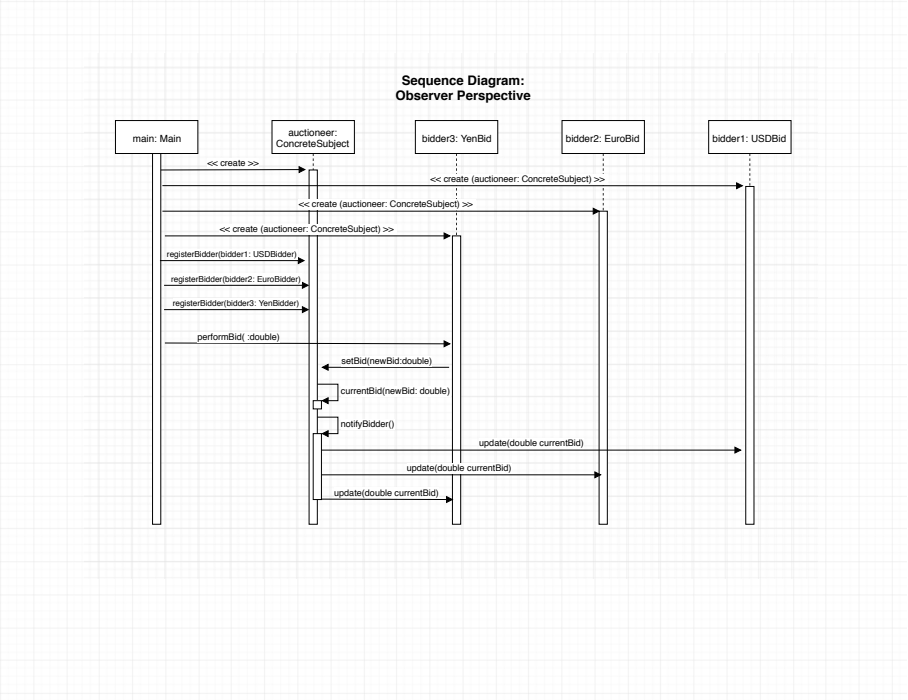
a) Below is a UML diagram for a coupling of a strategy pattern and a observer pattern. The situation modeled by the observer patter is an auction where the auctioneer observers many bidders. The situation modeled by the strategy pattern is that the bidders can swap out what currency they are using to make their bids. The class of bidder is present in both models.



b) The sequence diagram for the strategy pattern.



b) The sequence diagram for the Observer pattern.



2 Exercise 2

a Assume during your team's last sprint, that they completed 32 story points using a 3-person team working in sprints of 3 weeks for a total of 45-mandays. Calculate your team's estimated velocity for the next sprint if we still have 3-week sprints, but you now added 2 engineers to the team, and one of them can only work80% of the time.

Solution: Adding a full time engineer will increase our mandays by 15. Adding an engineer at 80% will add $12 = .8 \cdot 15$ mandays. Now we have a total of 72-mandays. Last week our focus factor was $\frac{32}{45}$. For this sprint we estimate our velocity to be $72 \cdot \frac{32}{45} = 51.2$ rounding down gives an expected velocity of 51 story points.

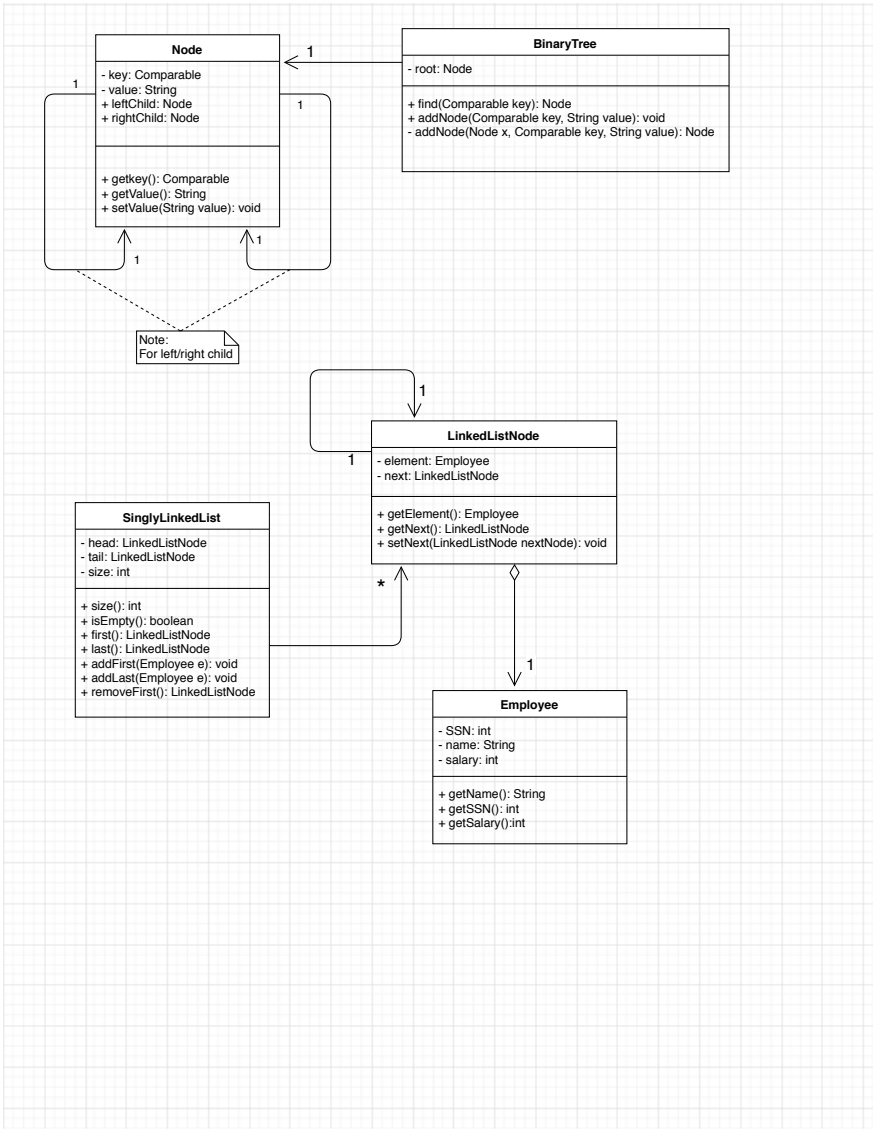
b How would you estimate a focus factor for a brand-new team?

Solution: As mentioned in class 70% is a standard focus factor. Since the team is settling into their roles it is reasonable to expect some distractions. To be conservative I would estimate a focus factor of 60% in case changes need to be made. Then we can adjust for the next sprint.

c We looked at using poker using semi-Fibonacci sequences to estimate story points. Think of another way to estimate story points and explain it. Is it better or worse than poker?

Solution: During the meeting everyone will anonymously enter how many story points a task will take. The distribution of the answers will be projected. Any outliers will have to explain their reasoning. Use the mean if the outliers do not convince you otherwise. This is similar to the semi-Fibonacci in that everyone is voting. Anonymity will allow people to be more honest.

d Draw a UML class diagram of a binary tree. Each node contains an integer.



JAVA Code

Linked List

```
/*  This code adapted from Data Structures and Algorithms in Java (6th ed.)
       Goodrich, Tamassia & Goldwasser
 */

public class SinglyLinkedList{

   // FIELDS
   private LinkedListNode head = null;  // first node (null if empty)
   private LinkedListNode tail = null;  // keep a ref to last node (null if empty)
   private int size = 0;                // track list size

   // CONSTRUCTOR
   public SinglyLinkedList() {
      // Default creates an empty list
   }

   // METHODS
   public int size() {
      return size;
   }

   public boolean isEmpty() {  // Helper method
      return size == 0;
   }

   public LinkedListNode first() {      // returns but doesn't remove the first element
      if (isEmpty()) {
         return null;
      }
      return head;
   }

   public LinkedListNode last() {       // returns last element
      if (isEmpty()) {
         return null;
      }
      return tail;
   }

   public void addFirst(Employee e) {              // adds a new node to front of list
      head = new LinkedListNode(e, head);   // link to a new node
      // this will set the current head to the
      // "next" of the new node and also reasign the "head"
      //  pointer to the new node

      if (size == 0) {
         tail = head;       // if it's the first node, head = tail
      }
      size++;               // increment size
   }

   public void addLast(Employee e) {

      // create a new node to become the tail
      LinkedListNode newestNode = new LinkedListNode(e, null);
```

```
        if (isEmpty()) {     // Check if list is empty
            head = newestNode;
        } else {
            tail.setNext(newestNode); // set the next value of the current tail
        }

        tail = newestNode;        // reset our "tail" pointer to the new node
        size++;                   // increment size
    }


    public LinkedListNode removeFirst(){     // remove and return first node
        if (isEmpty()){
            return null;        // check if list is empty
        }

        LinkedListNode nodeToReturn = head;

        head = head.getNext();  // reset the "head" node
        size--;         // Decriment size

        if (size == 0){     // if this was the last node in the list
            tail = null;    //  the list will now be empty
        }
        return nodeToReturn;

    }

}
```

```
/*  This code adapted from Data Structures and Algorithms in Java (6th ed.)
        Goodrich, Tamassia & Goldwasser
 */

public class LinkedListNode {

    // FIELDS
    private Employee element;               // ref to the element stroed at this node
    private LinkedListNode next;     // reference to the subsequent node in the list

    // CONSTRUCTOR

    public LinkedListNode(Employee element, LinkedListNode nextNode){
        this.element = element;
        this.next = nextNode;
    }

    // METHODS

    public Employee getElement(){
        return element;
    }

    public LinkedListNode getNext(){
        return next;
    }

    public void setNext(LinkedListNode nextNode){
        next = nextNode;
    }
}
```

```
public class Employee {

   // objects to be inserted as attributes in nodes

   private int SSN;
   private String name;
   private int salary;

   public Employee(String name, int ssn, int salary){
      this.name = name;
      this.SSN = ssn;
      this.salary = salary;
   }

   public String getname(){
      return name;
   }

   public int getSSN(){
      return SSN;
   }

   public int getSalary() {
      return salary;
   }
}
```

Binary Tree

```
/* Code Adapted from Algorithms (4th Edition),
   Sedgewick & Wayne
 */

public class BinaryTree {

   // Fields

   private Node root;

   // Constructor

   public BinaryTree(){
      this.root = null;
   }

   public Node find(Comparable key){
      Node x = root;      // Start at our root node

      while (x != null){
         int comparison =  key.compareTo(x.getKey());  // gives a value of -1, 0, 1
         if (comparison < 0) {
            x = x.leftChild;  // reset x and compare again
         } else if (comparison > 0) {
            x = x.rightChild;
         } else {
            return x;
         }
      }
```

```
        return null;
    }

    public void addNode(Comparable key, String value) {

        // if the key is in the tree - reset it's value
        //  otherwise, add a new node

        root = addNode(root, key, value);   // Start at the root, whatever gets returned will get set as root
    }

    private Node addNode(Node x, Comparable key, String value){
        if (x == null){
            return new Node(key, value);    // So if the tree is empty and root is null, the 1st argument will
                                            //  bee null and the a new node is created and set as root
        }

        int compare = key.compareTo(x.getKey());
        if (compare < 0){
            x.leftChild = addNode(x.leftChild, key, value); // if no key match is found, a new node gets created

        } else if (compare > 0){
            x.rightChild = addNode(x.rightChild, key, value);

        } else {
            x.setValue(value);  // if the key already exists, change it's value
        }
        return x;
    }
}
```

```
/* Code Adapted from Algorithms (4th Edition),
   Sedgewick & Wayne
 */

public class Node {
    // Fields
    private Comparable key;
    private String value;
    public Node leftChild;
    public Node rightChild;
    // Constructor
    Node(Comparable key, String value){
        this.key = key;
        this.value = value;
    }

    // get/set methods
    public Comparable getKey(){
        return key;
    }

    public String getValue(){
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```