Homework 3 Recap

CSE331 Spring 2011
Section 5

Design Questions

Party, Table, Servers, Restaurant, Utility

Who knows about whom?

- Which module logically contains the other?
- Why is this important?

Where should policies be enforced?
Who is responsible for a particular feature?

- In what modules should we place specific behaviors?

Before we begin...

These examples are from your code without comments.

Learning is an iterative process. Through this you can make better decisions from both good and bad examples.

We aren't discussing grading here. Code shown doesn't gain or lose points necessarily for these discussions.

Parties

Parties: Keeping reference to a Table

```
public void seatAtATable(Table table, int serverNo){
    if(table.getSeats() > 0){
        table.seatParty(this);
        this.table = table;
        this.server = serverNo;
    }
}
```

Should Party keep track of a Table and a server?
Where should the mapping(s) between Party and Table go?
Who is responsible for seating a party?

Parties: Static collection of info

```
public class Party {
    private static Set<String> names = new HashSet<String>();
    ...
    public Party(String name, int count) {
        ...
        names.add(name);
    }
    public static void removeParty(Party p) {
        names.remove(p.getName());
    }
    public static Set<String> getNames() {
        return names;
    }
}
```

In what part of the program should the unique Party name policy be enforced?

What if there were more than a single Restaurant?

Parties: Managing Tables

```
private boolean canFitAt(Table t) {
    return size <= t.getSize();
}
public Table minTableAvailable(Collection<Table> tables) {
    Iterator<Table> i = tables.iterator();
    Table max = null;
    while (i.hasNext()) {
        Table current = i.next();
        if (!current.isOccupied() && canFitAt(current) && (max == null || current.getCap() < max.getCap())) {
            max = current;
    }   }
    return max;
}
```

Probably best to let Table or Restaurant manage instances of Table.

Parties: Computing taxes

```
public final class Party {
    ...
    public static double getTotal(double subTotal) {
        Utility.isValid(subTotal >= 0);
        return 1.1 * subTotal;
    }
```

What if the tax rate changes?
Avoid magic numbers!
Should the Party really compute this?

Parties: Comparable

```
public final class Party implements Comparable<Party>{
    public static int newArrival = 1;
    private int partyOrder;
    ...
    public Party(String partyName, int partySize, int tableNumber, int serverNumber) {
        this.partyOrder = newArrival;
        newArrival++;
    }

    public int compareTo(Party other) {
        return this.partyOrder - other.partyOrder;
    }
```

Is the order of arrival *the* "natural ordering" for Parties?

Tables

Tables: Fields and Constructor

```
public class Table {
    private boolean occupied;
    private Servers server;
    private int seats;
    private int tableNum;

    public Table(int seats, boolean occ, int tableNum)
{
        this.seats = seats;
        this.occupied = occ;
        this.tableNum = tableNum;
     }
```

Does the table ever start out occupied?

Table or Tables?

```
public class Table{
    private Set<singleTable> emptyTables;
    private Set<singleTable> servingTables;
    private int currentNum;
    private final int maxSize;
    public Table(LinkedList<Integer> tableInfo) {
        emptyTables = new TreeSet<singleTable>();
        int maxSize = 0;
        while(!tableInfo.isEmpty()) {
            int size = tableInfo.pop();
            maxSize = Math.max(maxSize, size);
            emptyTables.add(new singleTable(size));
        }
        this.maxSize = maxSize;
        servingTables = new TreeSet<singleTable>();
        currentNum = 0;
    }
```

A table class should contain a table, not tables.

Tables: Immutability

```
public class Table {
    private final int size;
    ...
    public Table clear() {
        return new Table(this.id, this.size, -1, null);
    }
    public Table seat(Party p, int serverId) {
        return new Table(this.id, this.size, serverId, p);
    }
    public Table setServer(int serverId) {
         return new Table(this.id, this.size, serverId, this.party);
    }
```

Immutable tables are great!

Tables: Cloning

```
public void assign(Party p, int server) {
    this.p = p.clone();
}
```

Cloning should happen before p is passed in.

Tables: Static

```
public class Table implements Comparable<Table> {
    private static int largestTable = 0;
    public Table(int max) {
        if (max > largestTable) {
            largestTable = max;
        }
    }

    public static int getBiggestTable() {
        return largestTable;
    }
```

A restaurant has a largest table, not a table.
What if we have two restaurants?

Servers

Servers: Overly Complex Types

```
public class Servers {
    private static Map<Integer,
                       List<Map<Party,
                                Double>>> serverInfo;
    private static int lastAssignedID;
    private static List(Integer> serverOrder;
```

When your types get this complex, it's time to look at how to better structure your design.

Servers: Linked List?

```
public class Servers {
    private Set(Server> allServers;
    private Server front;
    private Server back;
    private static Server pointer;
.......
class Server implements Comparable<Server> {
    private static int cardinality = 0;
    private double tip;
    private int idNumber;
    private Server next;
```

You shouldn't need to make your class a data structure. Existing collections exist and give you one less thing to keep track of.

Servers: Inner Class for Server

```
public class Servers {
    private class Server {
        public double tips;
        public int id;
        public Set<Table> tables;
```

Who used these? What fields did you put in Server? This can be a clean way to structure parts of Servers.

Servers: Parallel Data Structures

```
final public class Servers {
    private int serverCount;
    private Map<Integer, Double> tips;
    private Queue<Integer> roundRobin;
    private Map<Integer, LinkedList<Table>>
            serverTables;
```

These can be an alternative to an inner class approach, but you have to keep all of these consistent (remove/add to all).

Servers: Limit Coupling

```
public void serveParty(Table t, Party p) {
        Server s = servers.remove();
        t.seatParty();
        p.assignServer(s); <--- server assigned to party
        s.addParty(t, p);  <--- party assigned to server
        servers.add(s);
```

Server knows about Party and Party knows about Server. This only needs to happen in one direction.

Restaurant

Restaurant: Printing Strings

```
public void printServers() {
    for (Server temp : waiters.getServers()) {
        System.out.println(temp.toString());
    }
}
public void printWaitList() {
    for (Party temp : waitlist.keySet()) {
        System.out.println(temp.toString());
    }
}
```

Core data classes should not print.
It locks them into the text representation of one particular client.

Restaurant: Returning Strings to Print

```
public String incMoney(Table t, double subtotal) {
    cashRegister += subtotal;
    t.unassign();
    return "Gave total of " + nf.format(subtotal) ...;
}
public String getWaitListString(){
    String ans = "";
    for(Party p : waitList)
        ans += p + "\n";
    return ans;
}
```

Returning strings to print is basically println'ing.
Locks the restaurant into a single client's output format.

Restaurant: Fixed-Size Array

```
public class Restaurant {
    private Table[] tables;
    private Party[] parties;

    public Restaurant(Table[] tables, int max) {
        this.tables = tables;
        parties = new Party[tables.length + 1];
    }
```

We can never have more Parties than we have Tables. So is it okay to declare an array of Parties of the size shown?

Restaurant: Searching for Tables

```
public Table check(Party p, double total, double tip) {
    for (Table t : tables) {
        if (t.isOccupied() && t.getParty().matches(p)) {
            profit += total;
            return seatWaitedParty(t);
        }
    }
    return null;
}
```

Shouldn't have to search all tables (O(N)) for this party.
Should keep a Map from parties (or names) to Tables.

Restaurant: You want another one?

```
public Restaurant clone(){
    try{
        Restaurant clone = (Restaurant) super.clone();
        clone.tables = new ArrayList<Table>(tables);
        clone.serversOnDuty = new Servers();
        clone.waitList = new ArrayList<Party>();
        ...
    }
}
```

You shouldn't add methods like clone or equals if it isn't natural to compare this kind of object in this way.

Restaurant: Utility object

```
public class Restaurant {
    private Utility u;

    public Restaurant() {
        u = new Utility();
        ...
    }
```

Utility has no state. It should be static, or a singleton.

Restaurant: Construct from file

```
public class Restaurant {
    public Restaurant(File f)
            throws FileNotFoundException {
        Scanner s = new Scanner(f);
        restaurantName = s.nextLine();
        ...
    }
```

Constructors for core classes should not perform heavy-duty tasks like reading entire files.

Restaurant: Undesired mutators

```
public class Restaurant {
    private List<Table> unoccupied;

    public void setUnoccupied(List<Table> list) {
        unoccupied = list;
        Collections.sort(unoccupied);
        maxTable = ...
    }
```

Now any outside client can set the total list of free tables...

Restaurant: Table not expert

```
private Table findTable(Party newParty){
    Table table = null;
    for (int tableNum: tables.keySet()){
        Table t = tables.get(tableNum);
        if(t.getParty() == null &&
           t.getTableSize() >= newParty.getSize()) {
            if(table == null ||
          table.getTableSize() > t.getTableSize()){
                table = t;
            }
        }
    }
    return table;
}
```

Restaurant has to ask the Table so many questions to figure out if a given party can sit there! Just make the Table the expert.

Restaurant: Server not expert

```
public final class Restaurant {
    private int serverOrdinal;

    public Restaurant() {
        serverOrdinal = 1;  ...
    }

    public void letServerIn() {
        serverQ.add(new Servers(serverOrdinal));
        serverOrdinal ++;
    }
```

The Restaurant shouldn't be counting servers, and it shouldn't be treating a Servers object as a single server.

Restaurant: Well-written checkout

```
public Table checkOut(String partyName, double total, double tip) {
    Table t = findParty(partyName);
    Party p = tables.get(t);
    p.checkOut(tip);
    t.vacate();
    tables.put(t, null);
    register += (total * TAX_MULTIPLIER);
    if (!waitingList.isEmpty()) {
        return seatFromWaitingList(t);
    } else {
        return null;
    }
}
```

Notice how other objects help out a lot, and how it has helper method calls. Very nice.

Restaurant: Wrap the servers?

```
public void addServers(){
    servers.addServer();
}
public int getServersCount(){
    return Servers.getServerCount();
}
public void dismissServer(){
    Collection<Table> tables = new ArrayList<Table>();
    tables = servers.dismissServer();
}
```

Is it good to have methods like this that "wrap" the Servers? Why or why not?

Utility.java

Utility: Use polymorphism

```
public class Utility {
    public static void ensureNameNotNull(String name) {
        if (name == null) throw new IllegalArgumentException("name cannot be null");
    }
    public static void ensureTableNotNull(Table table) {
        if (table == null) throw new IllegalArgumentException("table cannot be null");
    }
    public static void ensurePartyNotNull(Party party) {
        if (party == null) throw new IllegalArgumentException("party cannot be null");
    }
```

Easier to just write one method that takes an Object parameter.

Utility: what should it contain?

```
public class Utility {
    public static final int SALES_TAX = 10;    ...
```

Utility is not necessarily a place to put "miscellaneous" parts.
In general, Utility methods should not be "Restauranty."

Utility: Must consider encapsulation

```
public class Utility {
    public static NumberFormat currencyFormat =
            NumberFormat.getCurrencyInstance();
    ...
```

Missing the 'final' keyword - anyone could modify this NumberFormat.

Utility: Method design

```
public static void checkArgs(Object... objects) {
    for (Object object : objects) {
        if (object == null)
            throw new IllegalArgumentException("method parameters cannot be null");
        if (object instanceof String) {
            String s = (String) object;
            if (s.trim().equals("")) throw new IllegalArgumentException("entirely whitespace");
        } else if (object instanceof Double) {
            double d = (Double) object;
            if (d < 0) throw new IllegalArgumentException("money cannot be negative");
        } else if (object instanceof Integer) {
            int i = (Integer) object;
            if (i < 0) throw new IllegalArgumentException("counts cannot be negative");
        }
    }
}
```

Methods should still do "one thing." Regardless, this is clever.

Review

- Do you have any other thoughts about HW3?
- About class design?
- How would you change your HW3 code now?