How to Approach

Object oriented design questions are very important, as they demonstrate the quality of a candidate's code. A poor performance on this type of question raises serious red flags.

Handling Ambiguity in an Interview

OOD questions are often intentionally vague to test if you'll make assumptions, or if you'll ask clarifying questions. How do you design a class if the constraints are vague? Ask questions to eliminate ambiguity, then design the classes to handle any remaining ambiguity.

Object Oriented Design for Software

Imagine we're designing the objects for a deck of cards. Consider the following approach:

1. What are you trying to do with the deck of cards? Ask your interviewer. Let's assume we want a general purpose deck of cards to implement many different types of card games.
2. What are the core objects—and what "sub types" are there? For example, the core items might be: Card, Deck, Number, Suit, PointValue
3. Have you missed anything? Think about how you'll use that deck of cards to implement different types of games, changing the class design as necessary.
4. Now, get a little deeper: how will the methods work? If you have a method like Card Deck:.getCard(Suit s, Number n), think about how it will retrieve the card.

Object Oriented Design for Real World Object

Real world objects are handled very similarly to software object oriented design. Suppose you are designing an object oriented design for a parking lot:

1. What are your goals? For example: figure out if a parking spot is taken, figure out how many cars of each type are in the parking lot, look up handicapped spots, etc.
2. Now, think about the core objects (Car, ParkingSpot, ParkingLot, ParkingMeter, etc—Car has different subclasses, and ParkingSpot is also subclassed for handicapped spot).
3. Have we missed anything? How will we represent parking restrictions based on time or payment? Perhaps, we'll add a class called Permission which handles different payment systems. Permission will be sub-classed into classes PaidPermission (fee to park) and FreeParking (open parking). ParkingLot will have a method called GetPermission which will return the current Permission object based on the time.
4. How will we know whether or not a car is in a spot? Think about how to represent the data so that the methods are most efficient.

7.1 Design the data structures for a generic deck of cards. Explain how you would subclass it to implement particular card games.

pg 151

7.2 Imagine you have a call center with three levels of employees: fresher, technical lead (TL), product manager (PM). There can be multiple employees, but only one TL or PM. An incoming telephone call must be allocated to a fresher who is free. If a fresher can't handle the call, he or she must escalate the call to technical lead. If the TL is not free or not able to handle it, then the call should be escalated to PM. Design the classes and data structures for this problem. Implement a method getCallHandler().

pg 152

7.3 Design a musical juke box using object oriented principles.

pg 154

7.4 Design a chess game using object oriented principles.

pg 156

7.5 Design the data structures for an online book reader system.

pg 157

7.6 Implement a jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle.

pg 159

7.7 Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?

pg 161

7.8 Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or both the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves, and the win is assigned to the person with the most pieces. Implement the object oriented design for Othello.

pg 163

7.9 Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

pg 166

7.10 Describe the data structures and algorithms that you would use to implement a garbage collector in C++.

pg 167

7.1 Design the data structures for a generic deck of cards. Explain how you would subclass it to implement particular card games.

pg 62

SOLUTION

```
1   public class Card {
2       public enum Suit {
3           CLUBS (1), SPADES (2), HEARTS (3), DIAMONDS (4);
4           int value;
5           private Suit(int v) { value = v; }
6       };
7
8       private int card;
9       private Suit suit;
10
11      public Card(int r, Suit s) {
12          card = r;
13          suit = s;
14      }
15
16      public int value() { return card; }
17      public Suit suit() { return suit; }
18  }
```

Assume that we're building a blackjack game, so we need to know the value of the cards. Face cards are ten and an ace is 11 (most of the time, but that's the job of the Hand class, not the following class).

```
1   public class BlackJackCard extends Card {
2       public BlackJackCard(int r, Suit s) { super(r, s); }
3
4       public int value() {
5           int r = super.value();
6           if (r == 1) return 11; // aces are 11
7           if (r < 10) return r;
8           return 10;
9       }
10
11      boolean isAce() {
12          return super.value() == 1;
13      }
14  }
```

7.2 Imagine you have a call center with three levels of employees: fresher, technical lead (TL), product manager (PM). There can be multiple employees, but only one TL or PM. An incoming telephone call must be allocated to a fresher who is free. If a fresher can't handle the call, he or she must escalate the call to technical lead. If the TL is not free or not able to handle it, then the call should be escalated to PM. Design the classes and data structures for this problem. Implement a method getCallHandler().

pg 62

SOLUTION

All three ranks of employees have different work to be done, so those specific functions are profile specific. We should keep these specific things within their respective class.

There are a few things which are common to them, like address, name, job title, age, etc. These things can be kept in one class and can be extended / inherited by others.

Finally, there should be one CallHandler class which would route the calls to the concerned person.

NOTE: On any object oriented design question, there are many ways to design the objects. Discuss the trade-offs of different solutions with your interviewer. You should usually design for long term code flexibility and maintenance.

```
1   public class CallHandler {
2       static final int LEVELS = 3; // we have 3 levels of employees
3       static final int NUM_FRESHERS = 5; // we have 5 freshers
4       ArrayList<Employee>[] employeeLevels = new ArrayList[LEVELS];
5       // queues for each call's rank
6       Queue<Call>[] callQueues = new LinkedList[LEVELS];
7
8       public CallHandler() { ... }
9
10      Employee getCallHandler(Call call) {
11          for (int level = call.rank; level < LEVELS - 1; level++) {
12              ArrayList<Employee> employeeLevel = employeeLevels[level];
13              for (Employee emp : employeeLevel) {
14                  if (emp.free) {
15                      return emp;
16                  }
17              }
18          }
19          return null;
20      }
21
22      // routes the call to an available employee, or adds to a queue
```

```
23      void dispatchCall(Call call) {
24          // try to route the call to an employee with minimal rank
25          Employee emp = getCallHandler(call);
26          if (emp != null) {
27              emp.ReceiveCall(call);
28          } else {
29              // place the call into queue according to its rank
30              callQueues[call.rank].add(call);
31          }
32      }
33      void getNextCall(Employee e) {...} // look for call for e's rank
34  }
35
36  class Call {
37      int rank = 0; // minimal rank of employee who can handle this call
38      public void reply(String message) { ... }
39      public void disconnect() { ... }
40  }
41
42  class Employee {
43      CallHandler callHandler;
44      int rank; // 0- fresher, 1 - technical lead, 2 - product manager
45      boolean free;
46      Employee(int rank) { this.rank = rank; }
47      void ReceiveCall(Call call) { ... }
48      void CallHandled(Call call) { ... } // call is complete
49      void CannotHandle(Call call) { // escalate call
50          call.rank = rank + 1;
51          callHandler.dispatchCall(call);
52          free = true;
53          callHandler.getNextCall(this); // look for waiting call
54      }
55  }
56
57  class Fresher extends Employee {
58      public Fresher() { super(0); }
59  }
60  class TechLead extends Employee {
61      public TechLead() { super(1); }
62  }
63  class ProductManager extends Employee {
64      public ProductManager() { super(2); }
65  }
```

7.3 Design a musical juke box using object oriented principles.

pg 62

SOLUTION

Let's first understand the basic system components:

» CD player

» CD

» Display () (displays length of song, remaining time and playlist)

Now, let's break this down further:

» Playlist creation (includes add, delete, shuffle etc sub functionalities)

» CD selector

» Track selector

» Queueing up a song

» Get next song from playlist

A user also can be introduced:

» Adding

» Deleting

» Credit information

How do we group this functionality based on Objects (data + functions which go together)?

Object oriented design suggests wrapping up data with their operating functions in a single entity class.

```
1   public class CD { }
2   public class CDPlayer {
3       private Playlist p;
4       private CD c;
5       public Playlist getPlaylist() { return p; }
6       public void setPlaylist(Playlist p) { this.p = p; }
7       public CD getCD() { return c; }
8       public void setCD(CD c) { this.c = c; }
9       public CDPlayer(Playlist p) { this.p = p; }
10      public CDPlayer(CD c, Playlist p) { ... }
11      public CDPlayer(CD c) { this.c = c; }
12      public void playTrack(Song s) { ... }
13  }
14
15  public class JukeBox {
```

```
16      private CDPlayer cdPlayer;
17      private User user;
18      private Set<CD> cdCollection;
19      private TrackSelector ts;
20
21      public JukeBox(CDPlayer cdPlayer, User user, Set<CD> cdCollection,
22                     TrackSelector ts) { ... }
23      public Song getCurrentTrack() { return ts.getCurrentSong();    }
24      public void processOneUser(User u) { this.user = u;    }
25  }
26
27  public class Playlist {
28      private Song track;
29      private Queue<Song> queue;
30      public Playlist(Song track, Queue<Song> queue) { ... }
31      public Song getNextTrackToPlay(){ return queue.peek(); }
32      public void queueUpTrack(Song s){ queue.add(s); }
33  }
34
35  public class Song {
36      private String songName;
37  }
38
39  public class TrackSelector {
40      private Song currentSong;
41      public TrackSelector(Song s) { currentSong=s; }
42      public void setTrack(Song s) { currentSong = s;}
43      public Song getCurrentSong() { return currentSong; }
44  }
45
46  public class User {
47      private String name;
48      public String getName() { return name; }
49      public void setName(String name) { this.name = name; }
50      public long getID() { return ID; }
51      public void setID(long iD) { ID = iD; }
52      private long ID;
53      public User(String name, long iD) { ... }
54      public User getUser() { return this; }
55      public static User addUser(String name, long iD) { ... }
56  }
```

7.4 Design a chess game using object oriented principles.

pg 62

SOLUTION

```
1  public class ChessPieceTurn { };
2  public class GameManager {
3      void processTurn(PlayerBase player) { };
4      boolean acceptTurn(ChessPieceTurn turn) { return true; };
5      Position currentPosition;
6  }
7
8  public abstract class PlayerBase {
9      public abstract ChessPieceTurn getTurn(Position p);
10 }
11 class ComputerPlayer extends PlayerBase {
12     public ChessPieceTurn getTurn(Position p) { return null; }
13     public void setDifficulty() { };
14     public PositionEstimator estimater;
15     public PositionBackTracker backtracter;
16 }
17 public class HumanPlayer extends PlayerBase {
18     public ChessPieceTurn getTurn(Position p) { return null; }
19 }
20
21 public abstract class ChessPieceBase {
22     abstract boolean canBeChecked();
23     abstract boolean isSupportCastle();
24 }
25 public class King extends ChessPieceBase { ... }
26 public class Queen extends ChessPieceBase { ... }
27
28 public class Position { // represents chess positions in compact form
29     ArrayList<ChessPieceBase> black;
30     ArrayList<ChessPieceBase> white;
31 }
32
33 public class PositionBackTracker {
34     public static Position getNext(Position p) { return null; }
35 }
36 public class PositionEstimator {
37     public static PositionPotentialValue estimate(Position p) { ... }
38 }
39 public abstract class PositionPotentialValue {
40     abstract boolean lessThan(PositionPotentialValue pv);
41 }
```

7.5 Design the data structures for an online book reader system.

pg 62

SOLUTION

Since the problem doesn't describe much about the functionality, let's assume we want to design a basic online reading system which provides the following functionality:

» User membership creation and extension.

» Searching the database of books

» Reading the books

To implement these we may require many other functions, like get, set, update, etc. Objects required would likely include User, Book, and Library.

The following code / object oriented design describes this functionality:

```
1   public class Book {
2       private long ID;
3       private String details;
4       private static Set<Book> books;
5
6       public Book(long iD, String details) { ... }
7       public static void addBook(long iD, String details){
8           books.add(new Book(iD, details));
9       }
10
11      public void update() { }
12      public static void delete(Book b) { books.remove(b); }
13      public static Book find(long id){
14          for (Book b : books)
15              if(b.getID() == id) return b;
16          return null;
17      }
18  }
19
20  public class User {
21      private long ID;
22      private String details;
23      private int accountType;
24      private static Set<User> users;
25
26      public Book searchLibrary(long id) { return Book.find(id); }
27      public void renewMembership() { ... }
28
29      public static User find(long ID) {
```

```
30          for (User u : users) {
31              if (u.getID() == ID) return u;
32          }
33          return null;
34      }
35
36      public static void addUser(long ID, String details,
37                                 int accountType) {
38          users.add(new User(ID, details, accountType));
39      }
40
41      public User(long iD, String details, int accountType) { ... }
42  }
43
44  public class OnlineReaderSystem {
45      private Book b;
46      private User u;
47      public OnlineReaderSystem(Book b, User u) { ... }
48      public void listenRequest() { }
49      public Book searchBook(long ID) { return Book.find(ID); }
50      public User searchUser(long ID){ return User.find(ID); }
51      public void display() { }
52  }
```

This design is a very simplistic implementation of such a system. We have a class for User to keep all the information regarding the user, and an identifier to identify each user uniquely. We can add functionality like registering the user, charging a membership amount and monthly / daily quota, etc.

Next, we have book class where we will keep all the book's information. We would also implement functions like add / delete / update books.

Finally, we have a manager class for managing the online book reader system which would have a listen function to listen for any incoming requests to log in. It also provides book search functionality and display functionality. Because the end user interacts through this class, search must be implemented here.

7.6 Implement a jigsaw puzzle. Design the data structures and explain an algorithm to solve the puzzle.

pg 62

SOLUTION

```
class Edge {
    enum Type { inner, outer, flat }
    Piece parent;
    Type type;
    bool fitsWith(Edge type) { ... }; // Inners & outer fit together.
}
class Piece {
    Edge left, right, top, bottom;
    Orientation solvedOrientation = ...; // 90, 180, etc
}
class Puzzle {
    Piece[][] pieces; /* Remaining pieces left to put away. */
    Piece[][] solution;
    Edge[] inners, outers, flats;
    /* We're going to solve this by working our way in-wards, starting
     * with the corners. This is a list of the inside edges. */
    Edge[] exposed_edges;

    void sort() {
        /* Iterate through all edges, adding each to inners, outers,
         * etc, as appropriate. Look for the corners—add those to
         * solution. Add each non-flat edge of the corner to
         * exposed_edges. */
    }

    void solve() {
        foreach edge1 in exposed_edges {
            /* Look for a match to edge1 */
            if (edge1.type == Edge.Type.inner) {
                foreach edge2 in outers {
                    if edge1.fitsWith(edge2) {
                        /* We found a match! Remove edge1 from
                         * exposed_edges. Add edge2's piece to
                         * solution. Check which edges of edge2 are
                         * exposed, and add those to exposed_edges. */
                    }
                }
                /* Do the same thing, swapping inner & outer. */
            }
        }
```

```
41          }
42  }
```

Overview:

1. We grouped the edges by their type. Because inners go with outers, and vice versa, this enables us to go straight to the potential matches.

   We keep track of the inner perimeter of the puzzle (exposed_edges) as we work our way inwards. exposed_edges is initialized to be the corner's edges.

7.7 Explain how you would design a chat server. In particular, provide details about the various backend components, classes, and methods. What would be the hardest problems to solve?

pg 62

SOLUTION

What is our chat server?

This is something you should discuss with your interviewer, but let's make a couple of assumptions: imagine we're designing a basic chat server that needs to support a small number of users. People have a contact list, they see who is online vs offline, and they can send text-based messages to them. We will not worry about supporting group chat, voice chat, etc. We will also assume that contact lists are mutual: I can only talk to you if you can talk to me. Let's keep it simple.

What specific actions does it need to support?

» User A signs online

» User A asks for their contact list, with each person's current status.

» Friends of User A now see User A as online

» User A adds User B to contact list

» User A sends text-based message to User B

» User A changes status message and/or status type

» User A removes User B

» User A signs offline

What can we learn about these requirements?
We must have a concept of users, add request status, online status, and messages.

What are the core components?
We'll need a database to store items and an "always online" application as the server. We might recommend using XML for the communication between the chat server and the clients, as it's easy for a person and a machine to read.

What are the key objects and methods?

We have listed the key objects and methods below. Note that we have hidden many of the details, such as how to actually push the data out to a client.

```
1   enum StatusType {
2       online, offline, away;
3   }
4
```

```
5   class Status {
6       StatusType status_type;
7       String status_message;
8   }
9
10  class User {
11      String username;
12      String display_name;
13      User[] contact_list;
14      AddRequest[] requests;
15      boolean updateStatus(StatusType stype, String message) { … };
16      boolean addUserWithUsername(String name);
17      boolean approveRequest(String username);
18      boolean denyRequest(String username);
19      boolean removeContact(String username);
20      boolean sendMessage(String username, String message);
21  }
22  /* Holds data that from_user would like to add to_user */
23  class AddRequest {
24      User from_user;
25      User to_user;
26  }
27  class Server {
28      User getUserByUsername(String username);
29  }
```

What problems would be the hardest to solve (or the most interesting)?

Q1 *How do we know if someone is online—I mean, really, really know?*

While we would like users to tell us when they sign off, we can't know for sure. A user's connection might have died, for example. To make sure that we know when a user has signed off, we might try regularly pinging the client to make sure it's still there.

Q2 *How do we deal with conflicting information?*

We have some information stored in the computer's memory and some in the database. What happens if they get out of sync? Which one is "right"?

Q3 *How do we make our server scale?*

While we designed out chat server without worrying—too much– about scalability, in real life this would be a concern. We'd need to split our data across many servers, which would increase our concern about out of sync data.

Q4 *How we do prevent denial of service attacks?*

Clients can push data to us—what if they try to DOS us? How do we prevent that?

7.8 Othello is played as follows: Each Othello piece is white on one side and black on the other. When a piece is surrounded by its opponents on both the left and right sides, or both the top and bottom, it is said to be captured and its color is flipped. On your turn, you must capture at least one of your opponent's pieces. The game ends when either user has no more valid moves, and the win is assigned to the person with the most pieces. Implement the object oriented design for Othello.

pg 62

SOLUTION

Othello has these major steps:

2. Game () which would be the main function to manage all the activity in the game:
3. Initialize the game which will be done by constructor
4. Get first user input
5. Validate the input
6. Change board configuration
7. Check if someone has won the game
8. Get second user input
9. Validate the input
10. Change the board configuration
11. Check if someone has won the game...

NOTE: The full code for Othello is contained in the code attachment.

```
1   public class Question {
2       private final int white = 1;
3       private final int black = 2;
4       private int[][] board;
5
6       /* Sets up the board in the standard othello starting positions,
7        * and starts the game */
8       public void start () { ... }
9
10      /* Returns the winner, if any. If there are no winners, returns
11       * 0 */
12      private int won() {
13      if (!canGo (white) && !canGo (black)) {
14          int count = 0;
```

```
15              for (int i = 0; i < 8; i++) {
16                  for (int j = 0; j < 8; j++) {
17                      if (board [i] [j] == white) {
18                          count++;
19                      }
20                      if (board [i] [j] == black) {
21                          count--;
22                      }
23                  }
24              }
25              if (count > 0) return white;
26              if (count < 0) return black;
27              return 3;
28          }
29          return 0;
30      }
31
32      /* Returns whether the player of the specified color has a valid
33       * move in his turn. This will return false when
34       * 1. none of his pieces are present
35       * 2. none of his moves result in him gaining new pieces
36       * 3. the board is filled up
37       */
38      private boolean canGo(int color) { ... }
39
40      /* Returns if a move at coordinate (x,y) is a valid move for the
41       * specified player */
42      private boolean isValid(int color, int x, int y) { ... }
43
44      /* Prompts the player for a move and the coordinates for the move.
45       * Throws an exception if the input is not valid or if the entered
46       * coordinates do not make a valid move. */
47      private void getMove (int color) throws Exception { ... }
48
49      /* Adds the move onto the board, and the pieces gained from that
50       * move. Assumes the move is valid. */
51      private void add (int x, int y, int color) { ... }
52
53      /* The actual game: runs continuously until a player wins */
54      private void game() {
55          printBoard();
56          while (won() == 0) {
57              boolean valid = false;
58              while (!valid) {
59                  try {
60                      getMove(black);
```

```
61                      valid = true;
62                  } catch (Exception e) {
63                      System.out.println ("Enter a valid coordinate!");
64                  }
65              }
66              valid = false;
67              printBoard();
68              while (!valid) {
69                  try {
70                      getMove(white);
71                      valid = true;
72                  } catch (Exception e) {
73                      System.out.println ("Enter a valid coordinate!");
74                  }
75              }
76              printBoard ();
77          }
78
79          if (won()!=3) {
80              System.out.println (won () == 1 ? "white" : "black" +
81                                  " won!");
82          } else {
83              System.out.println("It's a draw!");
84          }
85      }
86  }
```

7.9 Explain the data structures and algorithms that you would use to design an in-memory file system. Illustrate with an example in code where possible.

pg 62

SOLUTION

For data block allocation, we can use bitmask vector and linear search (see "Practical File System Design") or B+ trees (see Wikipedia).

```
1   struct DataBlock { char data[DATA_BLOCK_SIZE]; };
2   DataBlock dataBlocks[NUM_DATA_BLOCKS];
3   struct INode { std::vector<int> datablocks; };
4   struct MetaData {
5       int size;
6       Date last_modifed, created;
7       char extra_attributes;
8   };
9   std::vector<bool> dataBlockUsed(NUM_DATA_BLOCKS);
10  std::map<string, INode *> mapFromName;
11  struct FSBase;
12  struct File : public FSBase {
13   private:
14      std::vector<INode> * nodes;
15      MetaData metaData;
16  };
17
18  struct Directory : pubic FSBase { std::vector<FSBase* > content; };
19  struct FileSystem {
20      init();
21      mount(FileSystem*);
22      unmount(FileSystem*);
23      File createFile(cosnt char* name) { ... }
24      Directory createDirectory(const char* name) { ... }
25      // mapFromName to find INode corresponding to file
26      void openFile(File * file, FileMode mode) { ... }
27      void closeFile(File * file) { ... }
28      void writeToFile(File * file, void * data, int num) { ... }
29      void readFromFile(File* file, void* res, int numbutes,
30                        int position) { ... }
31  };
```

7.10 Describe the data structures and algorithms that you would use to implement a garbage collector in C++.

pg 62

SOLUTION

In C++, garbage collection with reference counting is almost always implemented with smart pointers, which perform reference counting. The main reason for using smart pointers over raw ordinary pointers is the conceptual simplicity of implementation and usage.

With smart pointers, everything related to garbage collection is performed behind the scenes - typically in constructors / destructors / assignment operator / explicit object management functions.

There are two types of functions, both of which are very simple:

```
1   RefCountPointer::type1() {
2       /* implementation depends on reference counting organisation.
3        * There can also be no ref. counter at all (see approach #4) */
4       incrementRefCounter(); }
5
6   RefCountPointer::type2() {
7       /* Implementation depends on reference counting organisation.
8        * There can also be no ref. counter at all (see approach #4). */
9       decrementRefCounter();
10      if (referenceCounterIsZero()) {
11          destructObject();
12      }
13  }
```

There are several approaches for reference counting implementation in C++:

1. Simple reference counting.

```
1   struct Object { };
2   struct RefCount {
3       int count;
4   };
5   struct RefCountPtr {
6       Object * pointee;
7       RefCount * refCount;
8   };
```

Advantages: performance.

Disadvantages: memory overhead because of two pointers.

2. Alternative reference counting.

```
1   struct Object { … };
2   struct RefCountPtrImpl {
3       int count;
4       Object * object;
5   };
6   struct RefCountPtr {
7       RefCountPtrImpl * pointee;
8   };
```

Advantages: no memory overhead because of two pointers.

Disadvantages: performance penalty because of extra level of indirection.

3. Intrusive reference counting.

```
1   struct Object { … };
2   struct ObjectIntrusiveReferenceCounting {
3       Object object;
4       int count;
5   };
6   struct RefCountPtr {
7       ObjectIntrusiveReferenceCounting * pointee;
8   };
```

Advantages: no previous disadvantages.

Disadvantages: class for intrusive reference counting should be modified.

4. Ownership list reference counting. It is an alternative for approach 1-3. For 1-3 it is only important to determine that counter is zero—its actual value is not important. This is the main idea of approach # 4.

All Smart-Pointers for given objects are stored in doubly-linked lists. The constructor of a smart pointer adds the new node to a list, and the destructor removes a node from the list and checks if the list is empty or not. If it is empty, the object is deleted.

```
1   struct Object { };
2   struct ListNode {
3       Object * pointee;
4       ListNode * next;
5   }
```