

# LLD Interview Guide — 50 Minute Coding Plan

---

Each system follows the same template. Memorize the **structure**, not the code.

---

## Universal 50-Min Template

Time	Phase	What to Do
0-5 min	Clarify	Ask requirements, scope, constraints. Write enums + exceptions
5-15 min	Domain	Write domain classes (2-4 classes, fields + constructor only)
15-30 min	Service	Write the main service class with core operations (synchronized)
30-40 min	Tests	Write main() with 3-4 basic tests + 1 concurrency test
40-50 min	Polish	Run, fix, add edge cases, explain design decisions

---

## Common Patterns Across ALL LLDs

1. ENUMS → Status, Type (2-3 enums, one-liners)
2. EXCEPTIONS → 1-2 custom exceptions
3. DOMAIN → 2-4 simple classes (UUID ids, fields, constructor)
4. SERVICE → 1 main class with synchronized methods
5. MAIN + TESTS → Setup → Basic test → Edge case → Concurrency test

---

## 1. Amazon Locker (15 min core)

**Classes:** `Compartment`, `Parcel`, `Assignment`, `Locker`

**Key Methods:** `addParcel()`, `pickupParcel()`

**Pattern:** synchronized on Locker, fallback to larger size

**Edge Cases:** code expired, code already used, no compartment

```
Start with: Compartment(Size, Status) → Parcel(Size) → Assignment(code,
parcelId, compartmentId)
Core:      Locker.addParcel(parcel) → find free → assign → return code
          Locker.pickupParcel(code) → validate → free compartment
```

---

## 2. Movie Ticket Booking (20 min core)

**Classes:** `ShowSeat`, `Movie`, `Screen`, `Show`, `Booking`, `Theater`

**Key Methods:** `bookSeats()`, `cancelBooking()`

**Pattern:** synchronized(show) — per-show locking, not global

**Edge Cases:** double booking, seat not found, cancel + rebook

```
Start with: ShowSeat(row, col, type, price) → Screen(name, rows, cols) →
Movie(title)
Key class: Show has Map<seatId, ShowSeatStatus> — per-show seat state
Core:      BookingService.bookSeats(showId, userId, seatIds)
           1. Validate all AVAILABLE 2. Lock 3. Calculate price 4.
Mark BOOKED
           BookingService.cancelBooking(bookingId)
           1. Find booking 2. Free seats 3. Mark CANCELLED
```

---

### 3. Facebook Comment Section (20 min core)

**Classes:** User, Post, Comment

**Key Methods:** addComment(), editComment(), deleteComment(), reactToComment()

**Pattern:** synchronized(post) — tree structure with parentCommentId + replyIds

**Edge Cases:** unauthorized edit, reply to deleted comment, reaction toggle

```
Start with: User(name) → Post(authorId, content)
Key class: Comment has parentCommentId (null=top-level), replyIds[],
reactions Map<userId, ReactionType>
Core:      CommentService.addComment(postId, userId, parentId, content)
           CommentService.reactToComment(commentId, userId,
ReactionType)
           reactions.put(userId, type) — replaces previous (no
duplicate)
           Soft delete: status=DELETED, content="[deleted]"
```

---

### 4. Rate Limiter (20 min core)

**Interface:** RateLimiter { allowRequest(clientId) }

**Implementations:** TokenBucketRateLimiter, FixedWindowRateLimiter,  
SlidingWindowLogRateLimiter

**Key Methods:** processRequest(), batchRequests()

**Pattern:** Strategy per endpoint, per-client isolation

**Edge Cases:** refill over time, window reset, concurrent burst

Pick 2 of 3 algorithms in interview:

```
TokenBucket: double[] bucket = [tokens, lastRefillTime]
              refill = min(cap, tokens + elapsed * rate)
              if tokens >= 1 → allow, tokens--
```

```

FixedWindow:    long[] window = [windowStart, count]
                 if now - start >= windowMs → reset
                 if count < max → allow, count++

SlidingWindowLog: Deque<Long> timestamps
                  evict expired → if size < max → allow, add now

Service:        Map<endpoint, RateLimiter> + default limiter
                 processRequest(endpoint, clientId) → get limiter →
allowRequest

```

## 5. Load Balancer (15 min core)

**Interface:** `LoadBalancingStrategy { selectServer(servers) }`

**Implementations:** `RoundRobinStrategy, LeastConnectionsStrategy, RandomStrategy`

**Key Methods:** `routeRequest(), completeRequest(), markUnhealthy()`

**Pattern:** Strategy pattern, health filtering, connection tracking

**Edge Cases:** all servers down, dynamic strategy switch, server recovery

```

Start with: Server(id, host, port, status, activeConnections)
Core:      LoadBalancer.routeRequest(requestId)
           1. Filter healthy  2. Select via strategy  3. Increment
connections
           LoadBalancer.completeRequest(server) → decrement connections

Strategies: RoundRobin → AtomicInteger index % servers.size()
             LeastConn  → min(server.activeConnections)
             Random      → random.nextInt(servers.size())

```

## 6. Notification Service (20 min core)

**Interface:** `NotificationSender { send(userId, message) }`

**Implementations:** `EmailSender, SmsSender, PushSender`

**Key Methods:** `sendNotification(), broadcastToUser(), bulkSend()`

**Pattern:** Strategy + user channel preferences + retry logic

**Edge Cases:** channel disabled, all retries fail, bulk with mixed prefs

```

Start with: Notification(userId, message, channel, priority, status,
retryCount)
           UserPreference(userId, enabledChannels)
Core:      NotificationService.sendNotification(userId, msg, channel,
priority)
           1. Check preference  2. Get sender  3. Try send with
retries

```

```
4. Mark SENT or FAILED
broadcastToUser → iterate all enabled channels
bulkSend → iterate all userIds
```

---

## 7. Twitter / Social Feed (20 min core)

**Classes:** `User`, `Tweet`, `TwitterService`

**Interface:** `FeedGenerationStrategy { getFeed(userId, tweets, limit) }`

**Key Methods:** `addTweet()`, `follow()`, `unfollow()`, `getFeed()`, `likeTweet()`

**Pattern:** Strategy for feed ranking, follow/follower as Sets

**Edge Cases:** self-follow, duplicate user, retweet

```
Start with: User(userId, name, followers Set, following Set)
            Tweet(tweetId, text, userId, timestamp, likeCount,
retweetCount)
Core:      follow(userId, targetId) → user.addFollowing +
target.addFollower
            addTweet(userId, text) → create Tweet, add to tweetsByUser
map
            getFeed(userId, limit) → apply strategy (Chronological or
Engagement)
Strategies: Chronological → sort by timestamp desc
            Engagement    → sort by (likes + retweets) desc, then
timestamp
```

---

## Interview Tips

1. **Start by writing enums + exceptions** (shows structure, buys thinking time)
2. Use `synchronized(specificObject)` not `synchronized(this)` — explain why
3. **UUID for IDs** — `UUID.randomUUID().toString().substring(0,6)`
4. **Always write 1 concurrency test** — interviewers love seeing thread safety awareness
5. **Name your threads** — `new Thread(() -> {}, "BookThread-" + i)` helps debugging
6. **Print thread name** — `Thread.currentThread().getName()` in every operation
7. Use `Collections.synchronizedList()` for thread-safe result collection