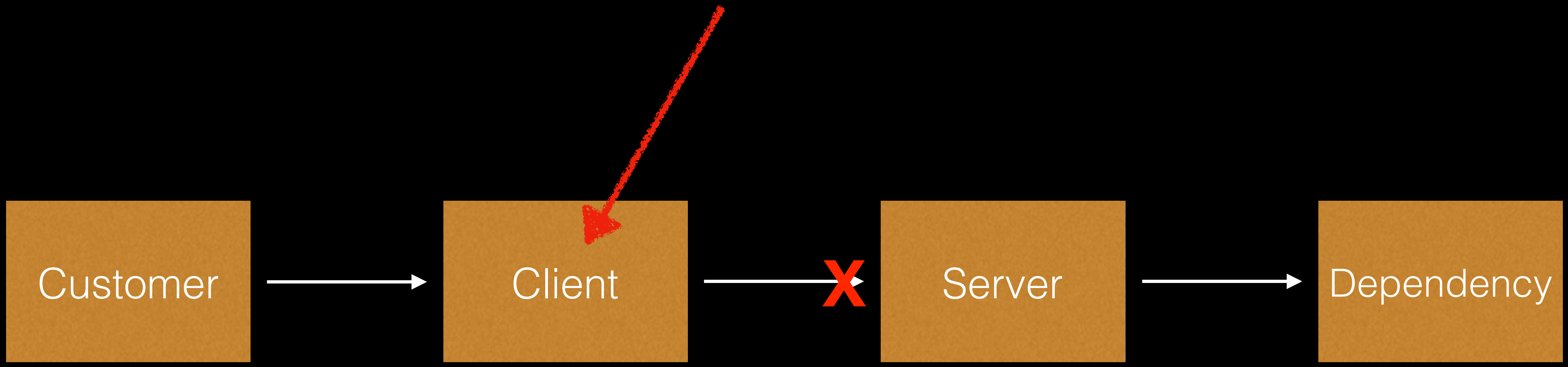
To Retry or Not to Retry

Avery Regier
@avery.regier

Typical Flow



Retry Thrice?

- The server probably hasn’t recovered from whatever made it fail the first time.
- Most of the time the answer will always be the same.
- Delays the customer finding out its failing.
  - HTTP & frameworks do not accommodate multiple responses, or I’m working on it answers.
- You don’t know if the customer has hit reload on the browser.
- End Result: Overwhelmed infrastructure.

The Mob

- Users get unhappier and more impatient as they perceive more delay without feedback.
- User patience is absurdly small.
- When you are taking their money they are full of anxiety.

Why did you do that again?

- Saw some errors in the logs
- Someone filed a defect
- Someone called the CEO
- Usually NOT as a thought through feature

Dependencies are People Too

- Scale
- Cost
- Queueing
- Support Phones
- Families

Delays

- If traffic has overwhelmed the server, the server gets just as much traffic after the next sleep period.
  - Better: Jitter

```
private long jitter() {
    return random.nextInt(1000) - 500;
}
```

- If you delay by the same amount a second time, it is just as likely the situation hasn't changed.
  - Better: Exponential delay

```
private long exponentialDelay(int tries) {
    return ((int)Math.pow(2, tries)) * 1000L;
}
```

Limbo

- Client side timeouts leave you in limbo
  - Is the server still working?
  - If I retry, will I get charged twice?
  - Better: Idempotence

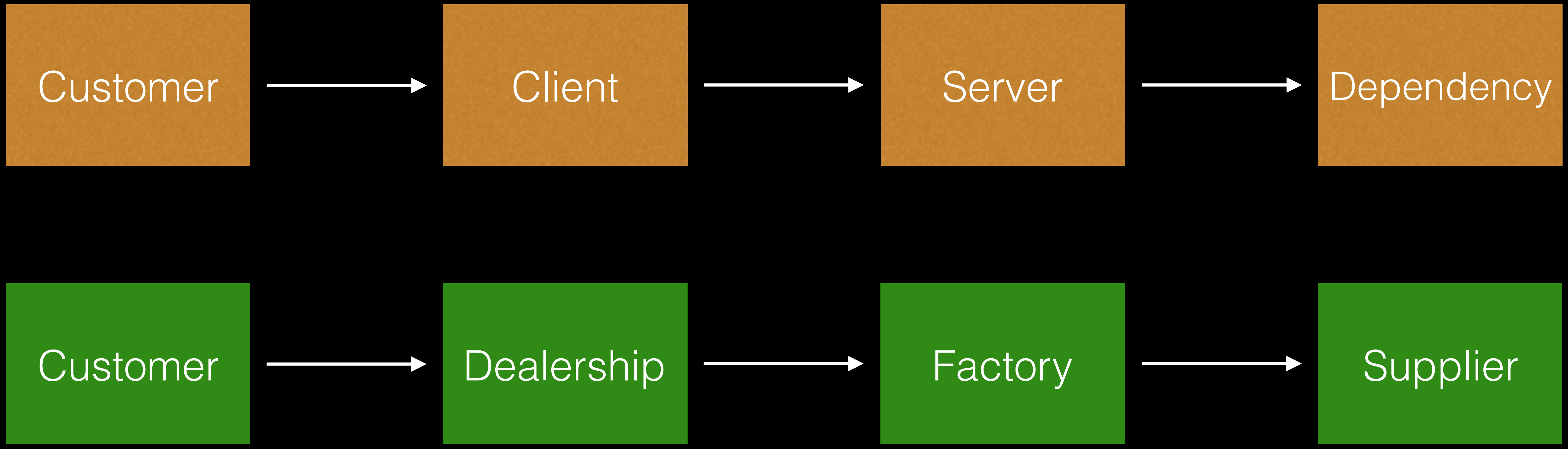
Goal

- Ensure the work gets done for the customer
  - in the shortest time frame
  - and the most efficient way
  - without causing pain to the servers
  - without making others fail too.
  - before they give up…

Avoid Zombies

Scenarios

Typical Flow



Sorry, I can’t sell you that.

Limit retries to potential server issues,
never on known client issues

Packaging vs Assembly

- List of small things
- Wrapping
- Formatting
- Filter for Security

- Chunked file upload/download
- Multi-stage process
- Bill of Material
- Application startup/login to main screen/user context
- Building a map

Pass Through

- When packaging commodities, quality is borne by the supplier.
  - Packaging bolts vs Using bolts
  - Packaging: push the quality process down to supplier
  - Using: double check before relying on them
- Retry decision made by the one assembling the end product.

Large Assembly

- Scenario: bad bolts from supplier
- All These:
  - Now:
    - Assembly testing
    - Avoid using bad batch of bolts
    - Pulling over defective product to fix
    - Factory takes responsibility for a quality end product
  - Later:
    - Ensure consistent quality in supply line

Large Assembly No-No’s

- Ship defective product
- Ask customer to buy another one
- Rebuild tractor from scratch

Learnings

- Retry the failed part of the process, not the whole process.
- Avoid dependencies where there are known failures.
    - Circuit Breakers
    - Failover
    - Fall backs

Shipping

- Interaction:
  - Take an order
  - Give feedback
  - Tracking #
  - Cancellation/Return Policies
  - Back orders

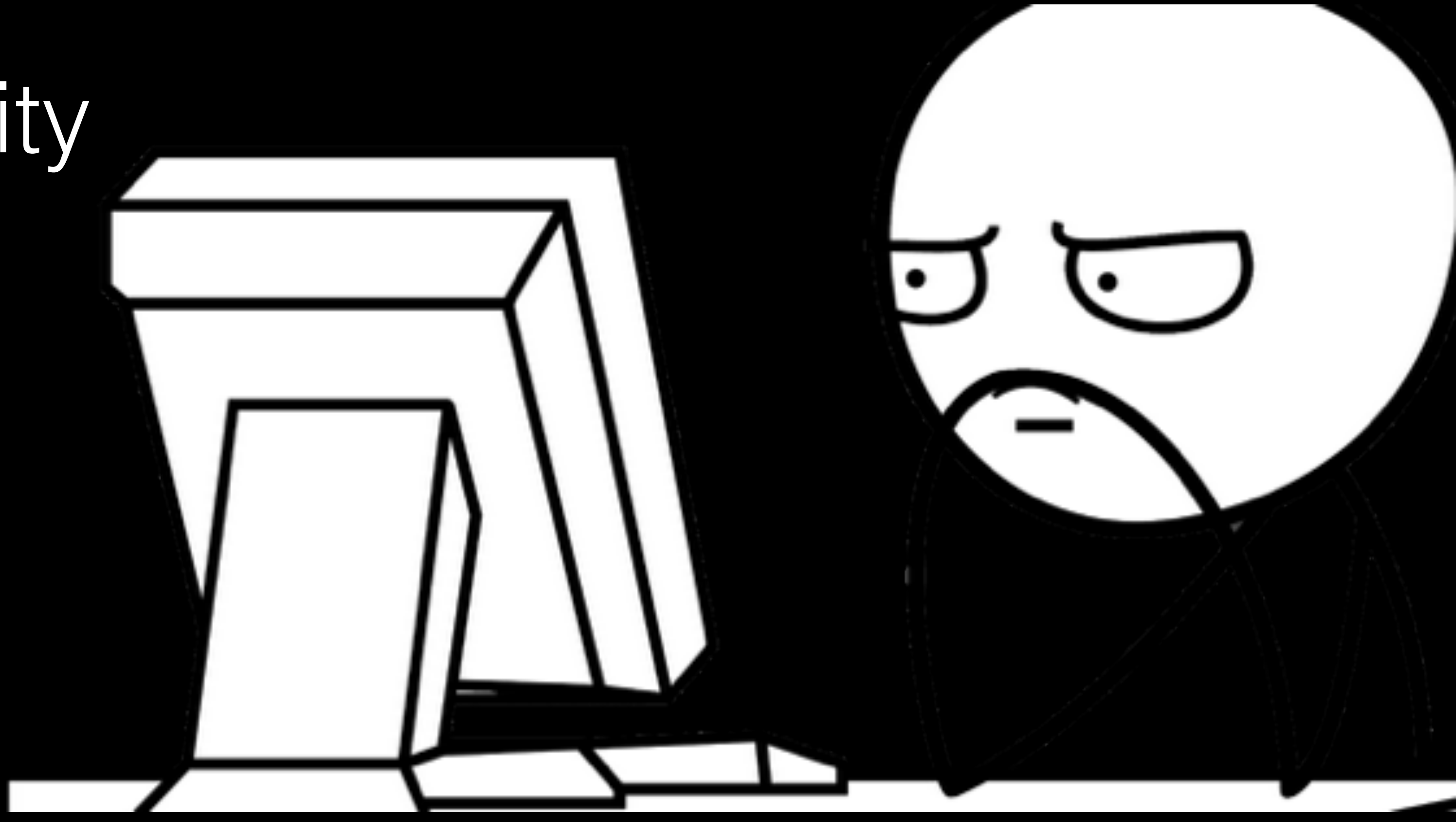
- Design so that the responsibility for completing the process is not left to the end user.
  - Asynchronous Delivery
  - Notifications
    - When complete
    - Problems
  - Know customer's
    - needs
    - willingness to wait
  - Persistence

Retry Principles

- Smart
    - Limit to potential server issues, never on known client issues
- Delayed
    - Follow Retry-After
    - If no Retry-After, start with 1 second
        - Exponential backoff
            - If the previous retry fails, the server is likely to need longer to heal
- Limited
    - By time that the **customer is willing to wait** ⬅ **THIS**
        - Not a number of attempts
        - When unknown, defer retries to the client who does.
        - Large assemblies usually will know.

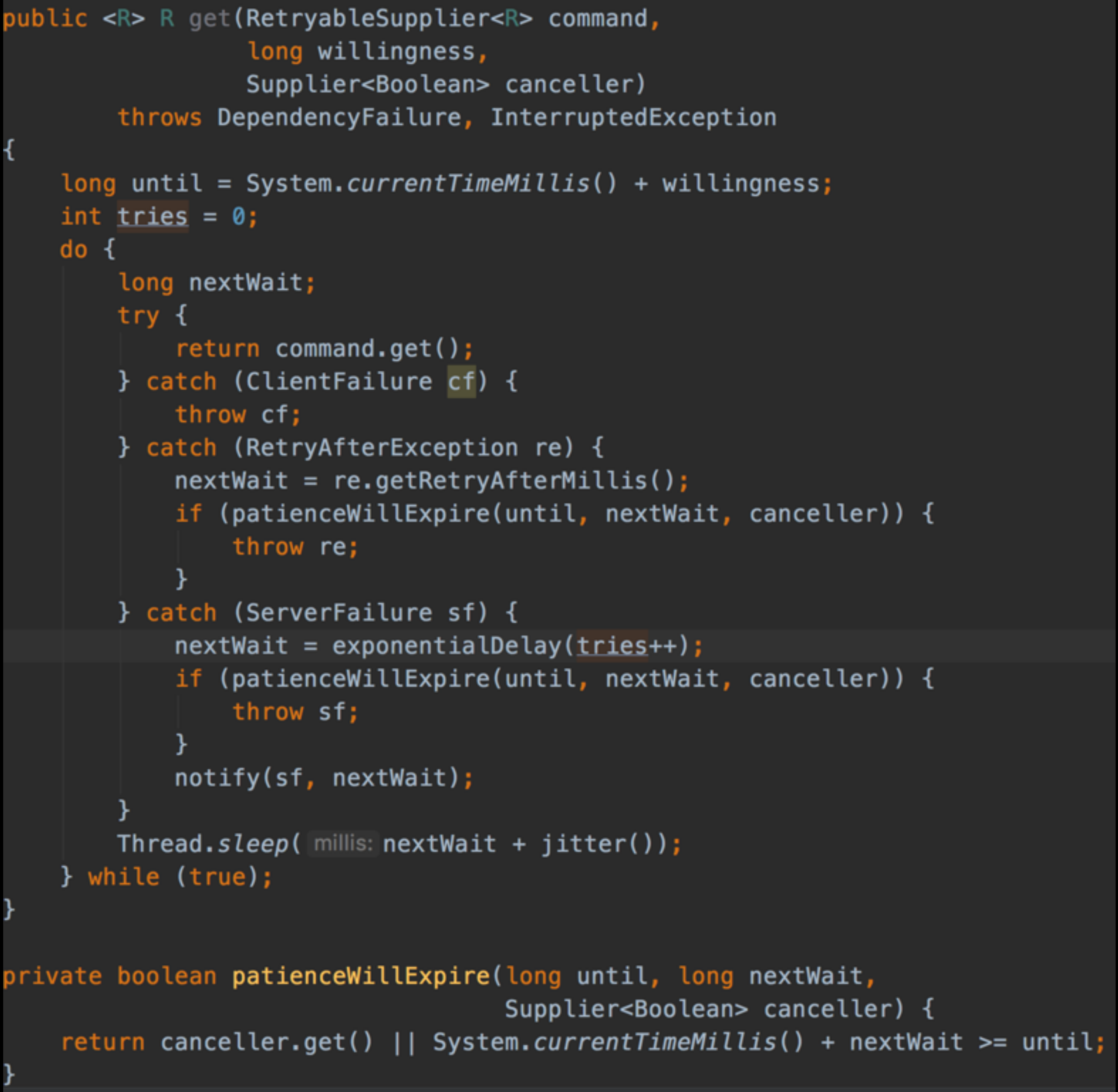
Willingness Hints

- Paid vs Free
- Effort vs Ease
- Custom vs Commodity
- Physical vs Virtual



Ask Them!

- Notifications
- Cancellation
- Customer Knowledge
- UX Testing



```java
public <R> R get(RetryableSupplier<R> command,
                 long willingness,
                 Supplier<Boolean> canceller)
        throws DependencyFailure, InterruptedException
{
    long until = System.currentTimeMillis() + willingness;
    int tries = 0;
    do {
        long nextWait;
        try {
            return command.get();
        } catch (ClientFailure cf) {
            throw cf;
        } catch (RetryAfterException re) {
            nextWait = re.getRetryAfterMillis();
            if (patienceWillExpire(until, nextWait, canceller)) {
                throw re;
            }
        } catch (ServerFailure sf) {
            nextWait = exponentialDelay(tries++);
            if (patienceWillExpire(until, nextWait, canceller)) {
                throw sf;
            }
            notify(sf, nextWait);
        }
        Thread.sleep( millis: nextWait + jitter());
    } while (true);
}

private boolean patienceWillExpire(long until, long nextWait,
                                   Supplier<Boolean> canceller) {
    return canceller.get() || System.currentTimeMillis() + nextWait >= until;
}
```

42