LLD - Facade and Observer

- Facade
- Observer
  - event driven
- Strategy

LLD

Design

Machine Coding

- entities
- class Diagram
- schema diagram

UML + schema diagram

Real world

→ Pen - Notebook

Games

→ TicTacToe - Chess

→ Snakes & Ladders

Management Systems

Management

→ Parking lot

→ BMS

→ Email campaigns

Machine coding

- Splitwise

- Spring Boot

- JPA

Facade

↓

Complex

→ present a clean, simple interface to hide internal dragons

E commerce - Flipkart ORder

Inventory service Inv

createOrder(...) {

→ inventory.checkout();

payment.checkout();

recom.checkout();

}

Create order →

- quantity check —
- update inventory —
- generate payment link —
- tracking service —

Inventory Service

Payment Service

- analytics service -
- recommendations -

Testing

- mock all dependencies
- easy

Extensibility

- code duplication


```
Order Manager {

  CreateOrder() {

  }
```

Order V1 Order V2

{ dependency

OrderManagerImpl {

OrderProcessor proc;

createOrder() {

processor.process();

}

}

Testing ✓

Extensibility ✓

SRP

Class Processor {

Invariant inv

Pas inv

process () {

inv.checkout();

analytics.touch();

}

JDBC

Database.create(null)

Order Manager

↓

Business Logic

Order Processor

↓

Utility

Disadvantages —

— God class

— Additional facade

Order

Order Manager

Order Processor

Payment Processor.

Notification Processor.

Notification

notifier.send(_ _)

→ batches

→ history

→ slack | phone | email

Asynchronous

Synchronous

↳ user will wait

request → (1)

(2)

3
response
Sync path
Req
Order Placed
response
On Async Processing
Async
Queues
Pub/Sub

Facade

→ clean interface

hiding complex

Scenarios.

Order Processor

↳ Payment process.

↳ Analytics Engine

Behavioural design patterns

→ how best to implement

common behaviours

→ functionality

Strategy

↑

Bird → flying behaviour

Strategy → switching algo. at runtime

Observer →

Observer

→ continuously goto the store

→ wasteful

→ send everyone an email

→ wasteful

→ subscribe/ register → mailing

list of subscribers → send an email

Observer

↳ notify interested interfaces

← price of bit coin ⤵

- email, tweet

```
BitcoinManagerImpl {
    Bitcoin bitcoin; TweetService
    EmailService email;
    public void setPrice(int) {
        bitcoin.setPrice();
        email.send();
        tweet.send();
    }
}
```

- email
- tweet
- stock
- sms

- OCP
- SRP

(2) Polling


```
if price != previousPrice
    Send Notification
```

```
Bitcoin Poller {
    Bitcoin Manager
        previous
```

Tweet
Email
} notification

poll() {

mongo.getPrice()

if (price != previous){

send notifications

}

}

→ wasteful

→ period — 1min
— 1sec
— 1day

Bitcoin → dependency

→ dependency registers with Observable / Publisher

→ when state changes, publisher calls each observer. — Email
— Tweet

↓
subscriber.

→ Email → subscribe

→ bitcoin price changes

→ Publisher → send a message

to your subscriber.

Decouple dependencies when one action triggers multiple reactions

Observer

→ Ecommerce → Triggin

→ TypeORM

↓

Subscriber

→

event driven

→ sub or

→ select

6:09 = 6:12

10:42

Steps

→ interesting

① Create an observable interface

↓

Subject

publisher.

Bitcoin Manager.

interface ↓

Observable {

registerObj

deregisterObj

}

② Create your observer → interested interface

subscriber

interface Observer { → email, slack, etc.

notify();

abstract class Observable {

List<Observer> observers;

register() {

registe();

deregiste();

notify();

}

③ Concrete observable

```
BitcoinManager extends Observable {
    setPrice() {
        notify()
    }
}
```

④ Concrete observer.

```
class EmailService imp. observer {
```

→ notify() {

// send an email

}

}

Angular - observables

- promise — a next

- Redux store

- React - observer

- in the section - observ.

Factory

→ don't want to use subclasses

→ why → OCP

→ maintenance

→ Simple factory

→ create a static method to accept type

→ Based on type, create instances

→ createBtn (BtnType) {

switch (type) {

P : return P Button();

}

}

→ O/C

→ maintainable

→ readability

Btn btn = SimpleFactory.CreateBtn(ROUND)

↳ Programming to Interfaces

One class → multiple objects of

One class → ~~one~~ instance of one class

Real Product Hierarchy

① Factory interface

BtnFactory → factory method

```
}
```

```
Btn createBtn() {
```

↓

```
RoundBtnFactory implements BtnFa... {
    createBtn() {
        return new RoundBtn();
    }
}
```

→ OCP ✓

→ SRP ✓

Problems

→ class explosion

Abstract factory

→ relate set of products

→ family of product.

Abstract factory → factory of factories

ThemeFactory {

createBtn();

createCheckbox();

}

DarkThemeFactory {
createBtn() {
return DarkBtn
}
LightThemeFactory {
ret urn LightBtn
}