Behavioural → Strategy

Creational → Prototype

From next class, we are going to start at 9:05 PM sharp

→ OPTIONAL CLASS

Sunday : Remedial on Design Patterns

attendance would count

→ Adapter

→ Decorator =

→ AMA

1.5 hrs

Agenda

① Behavioural → Strategy → >70%

② Creational → Prototype

Strategy : Plan to do something

algorithm to do something

Way to do Something

There can be multiple ways to do that particular Thing.

→ There can be multiple strategies to do that thing.

→ Using Google Maps find the route from Delhi to Noida

Strategy to find route

- Car ⇒
- Bike ⇒
- Walk ⇒
- Public Transport ⇒

Google Maps

findRoute(location start, location end, mode) {

if (mode == public)

elif (mode == car)

}

}

- Every entity should be a class

↳ Every behaviour is a method in the class

Strategy DP

⇒ Create an interface for the thing that you want to do

→ interface RouteFindingStrategy {

findPath(location start, location end);

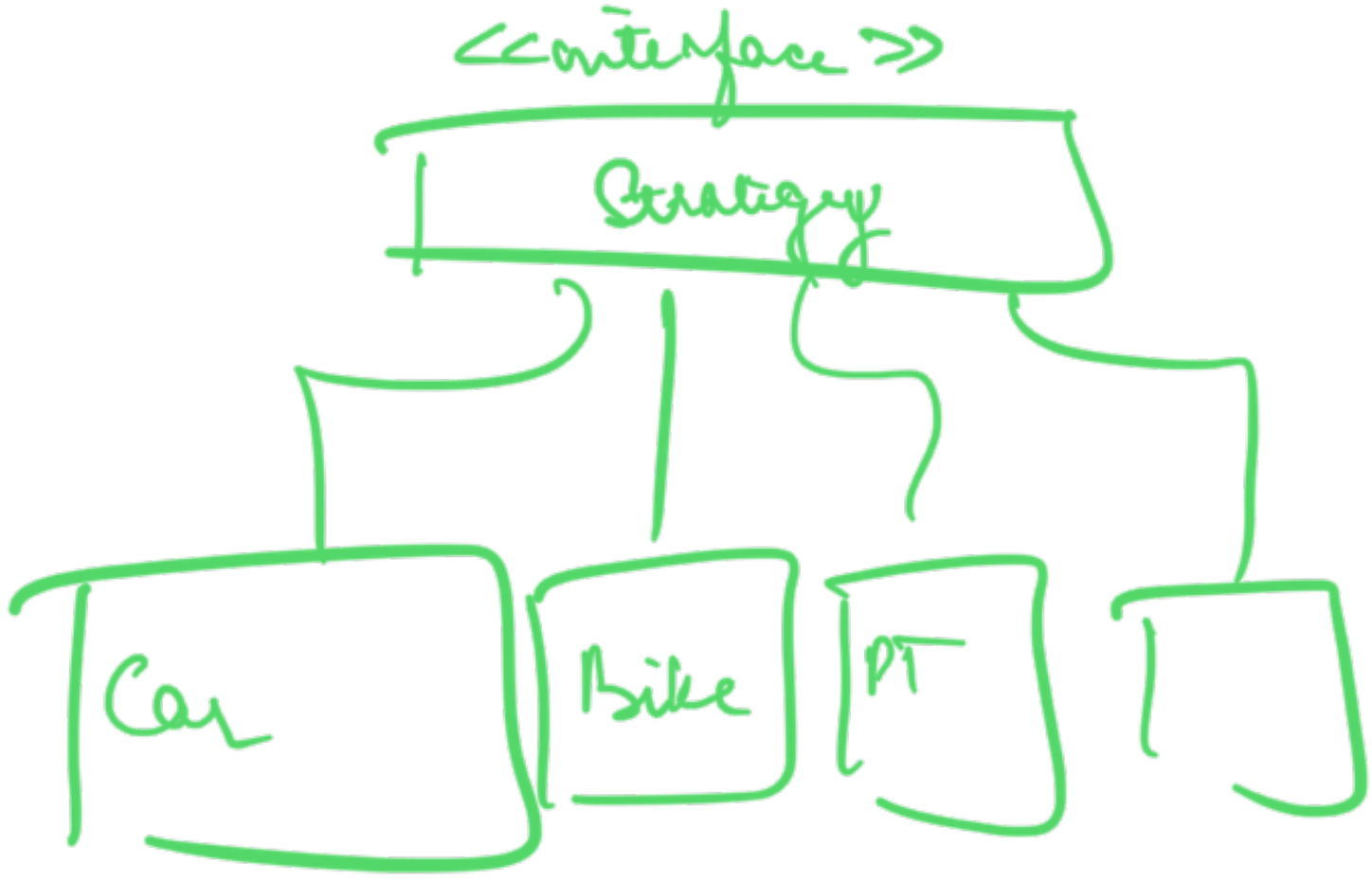
}

→ For every way to perform that behaviour, create a class that implements the Strategy interface.

class Walk Route Finding Strategy implements Route Finding Strat.

class PT Route Finding Strat ——————

class Car Route Finding ——————



Route finding Strategy factory {

getStrategyForMode(mode) {

if (mode == CAR)

return new Car

3

3

App^M Code

↓

Google Maps

findRoute (location A, location B, Mode m)

return RouteFindingStrategyFactory.

.getStrategyFor Mode(m)

- findRoute(A, B);

}

---

① Multiple ways to do Something (fly)

interface FlyingStrategy {

    fly()

}

class SlowFlyingStrategy implements FlyingStrategy {

fly() {

}

}

class FastFlyingStrategy {

fly() {

}

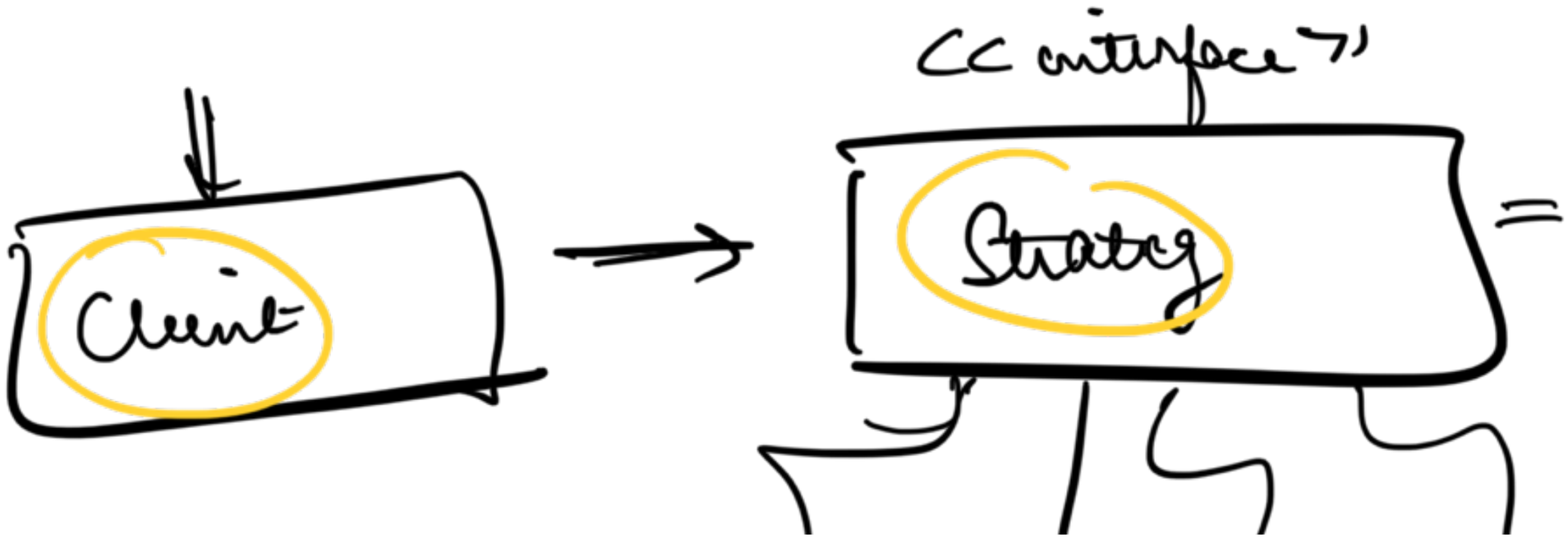
}

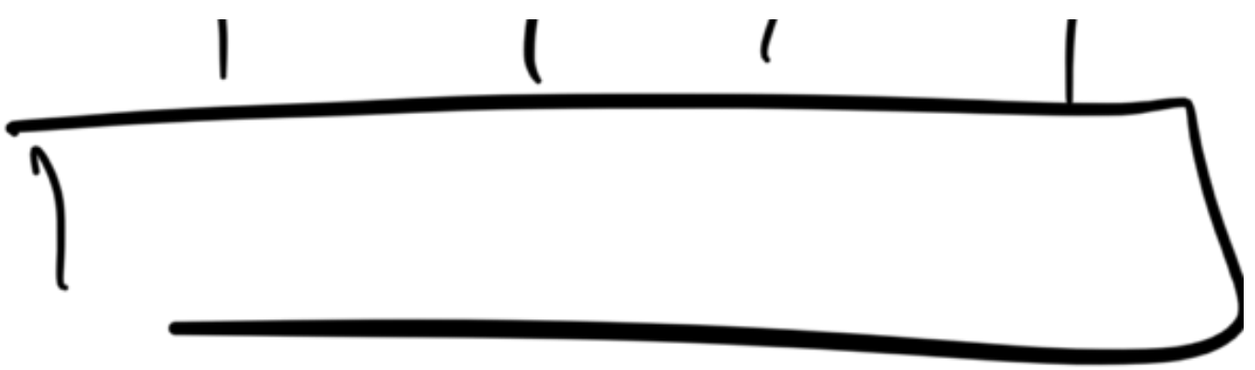
Client {

→ ① Factory

get Strategy for sum( )

}





Use Case: ① Our Google maps turn are multiple ways to calculate path

② But at one time google Maps will only use one of those ways for everyone

③ Way that is used is given via a config

Google Maps

RouteFinding Strategy Strategy;

Google Maps (Route finding Strategy ⟵)

this - Strategy = Strategy.

}

Main {

Read via Config

Main() {

GoogleMaps maps = new GoogleMap( )

}

}

@Autowired

UserService {

We encrypt the password

i.e. HASHING + SALTING

~~login~~ ( )

→ Password Hashing Strategy (PasswordHasher)

Sign Up (username, password, email) {

}

User Service (Stra)

}

→ Whenever there can be multiple ways to do a behaviour, abstract the behaviour out of

the entity class into a separate behaviour

class.

Bird

fly()

Hen

flyes

Pigeon

flyes

Penguin

Crow

flyes

Peacock

flyes

fly in the exact same name

fly in the exact same name

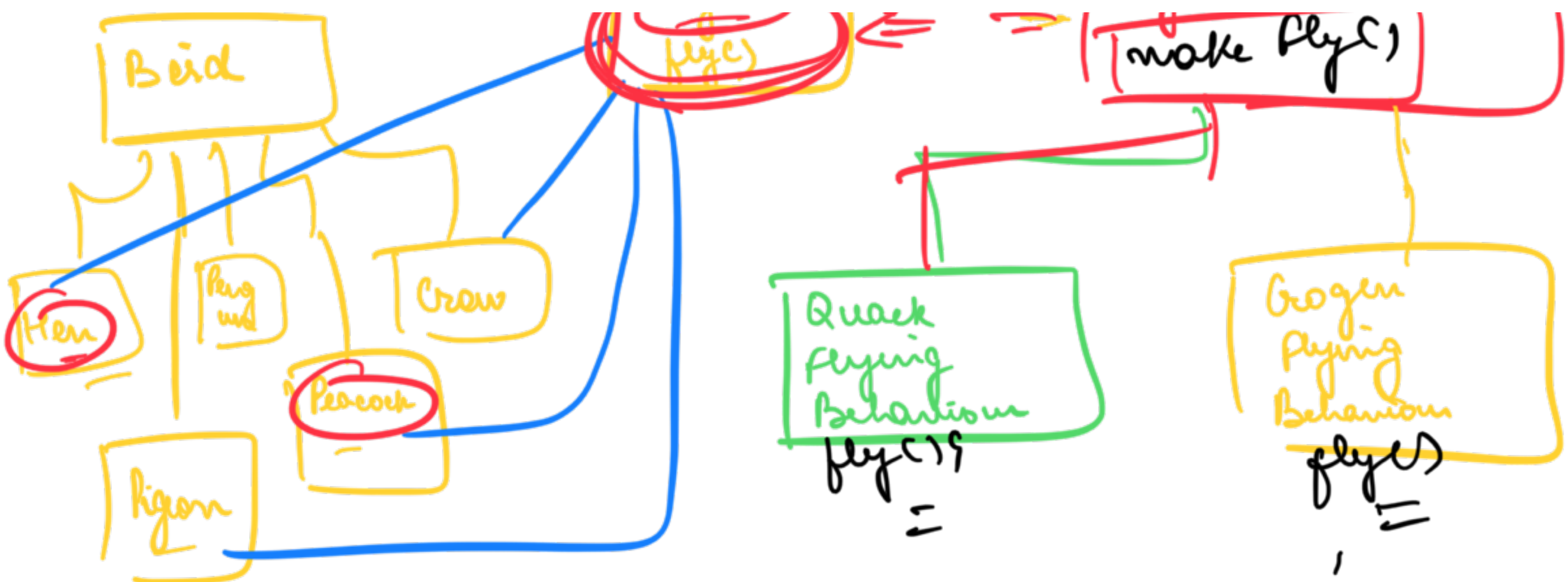
depict birds that can fly

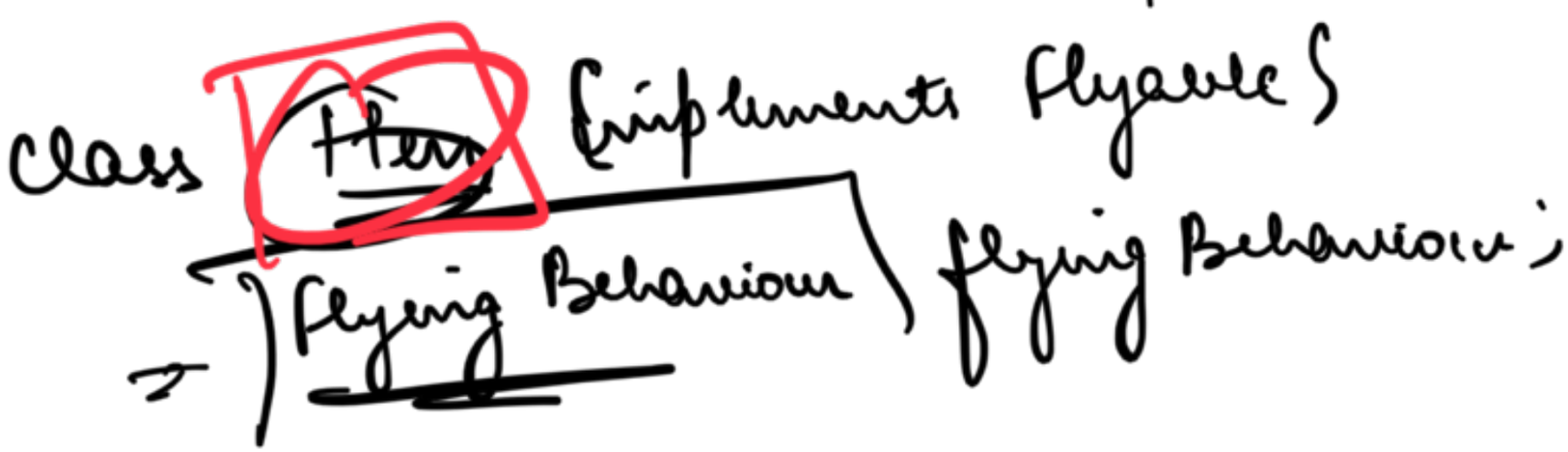
depict how birds can't

Interface

Flyable

Flyes Behaviour





Hen (FlyingBehaviour fb)

this.

}

fly() {

this.flyingBehaviour.

makeFly()

}

List<Flyable>

In Strategy dp

behaviour is extracted out from entity class

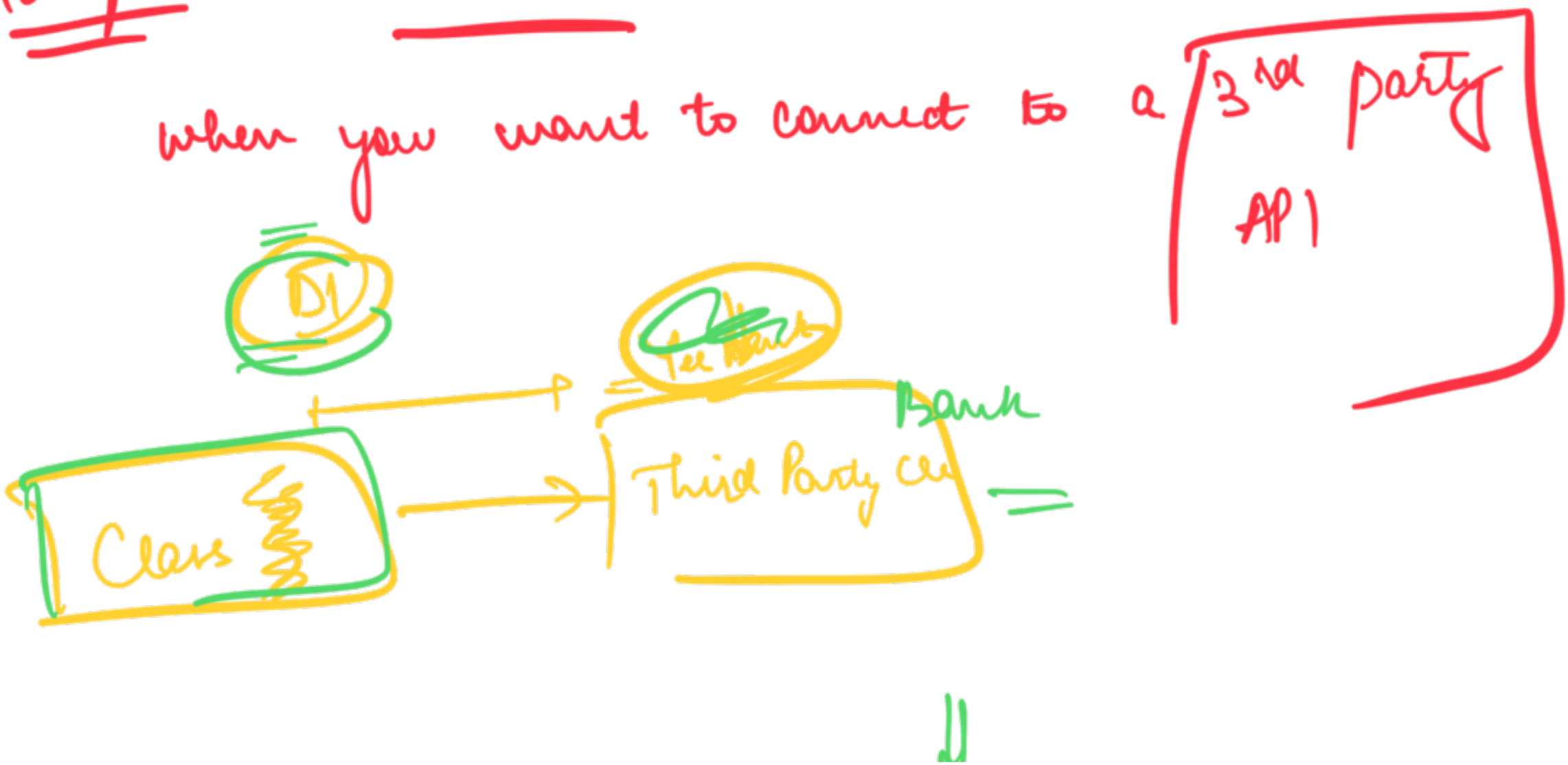
to a separate class:

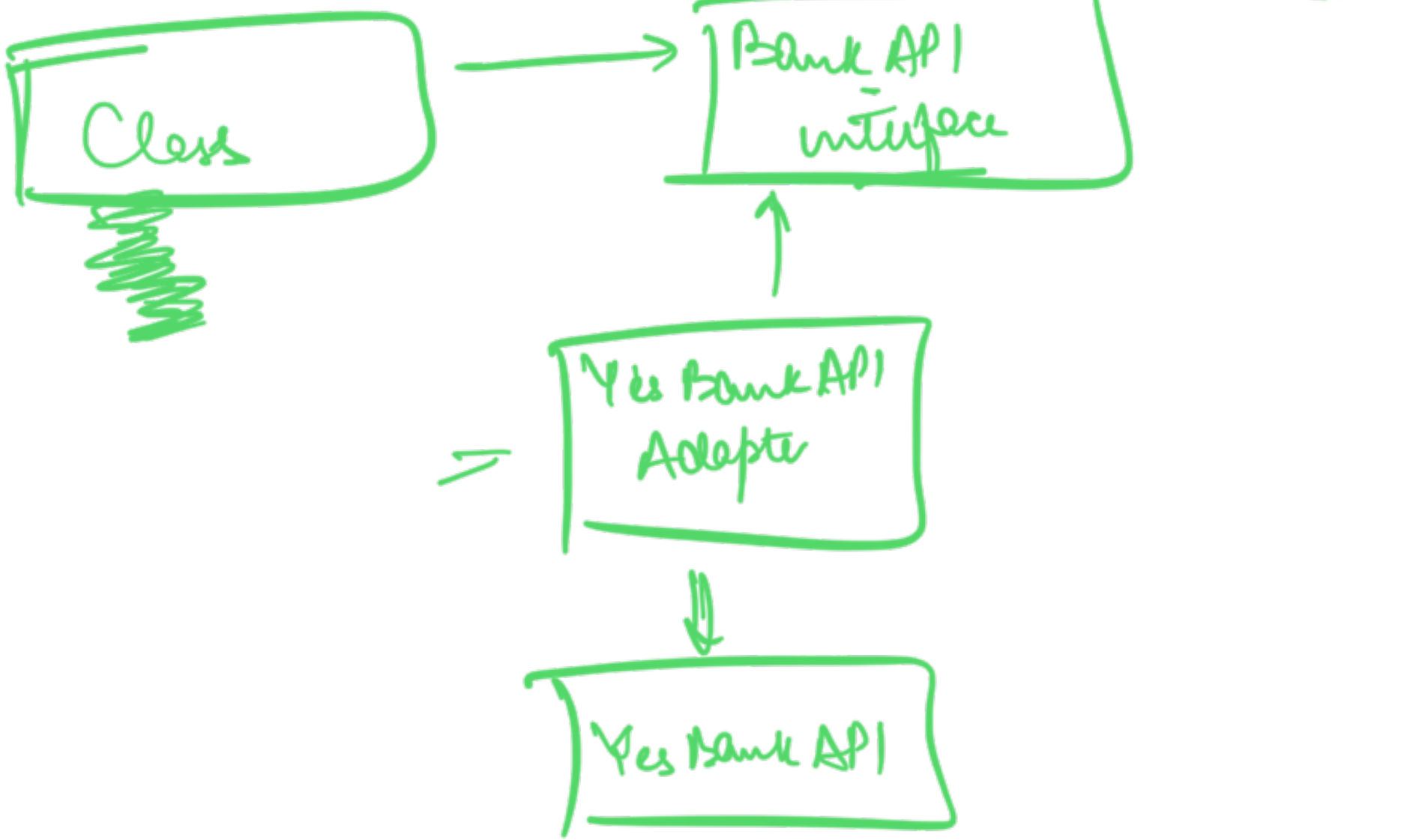
Entity class just delegates to behaviour class

you can't control its interface

Adapter ⇒ Structural

when you want to connect to a 3rd party API

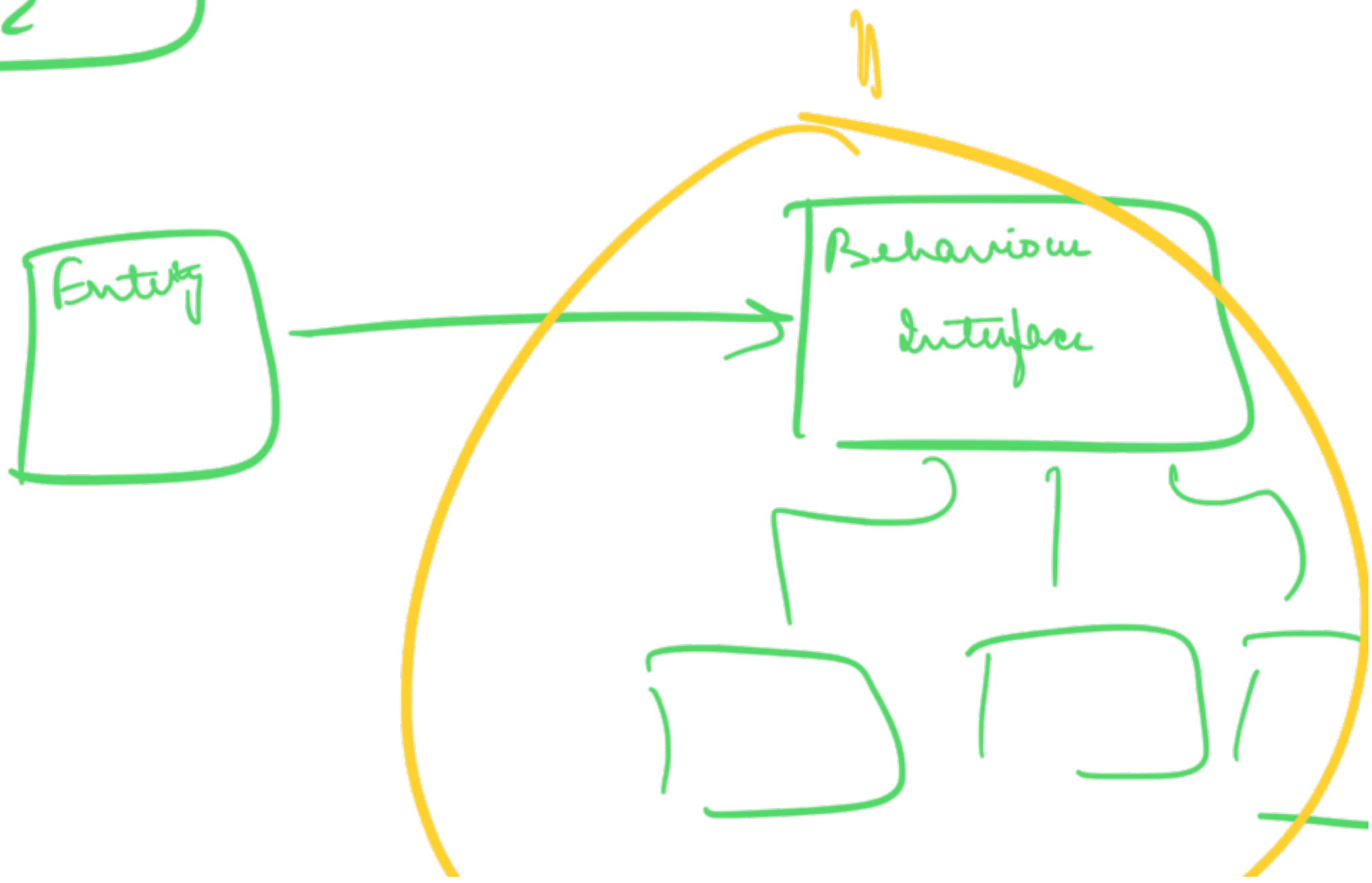




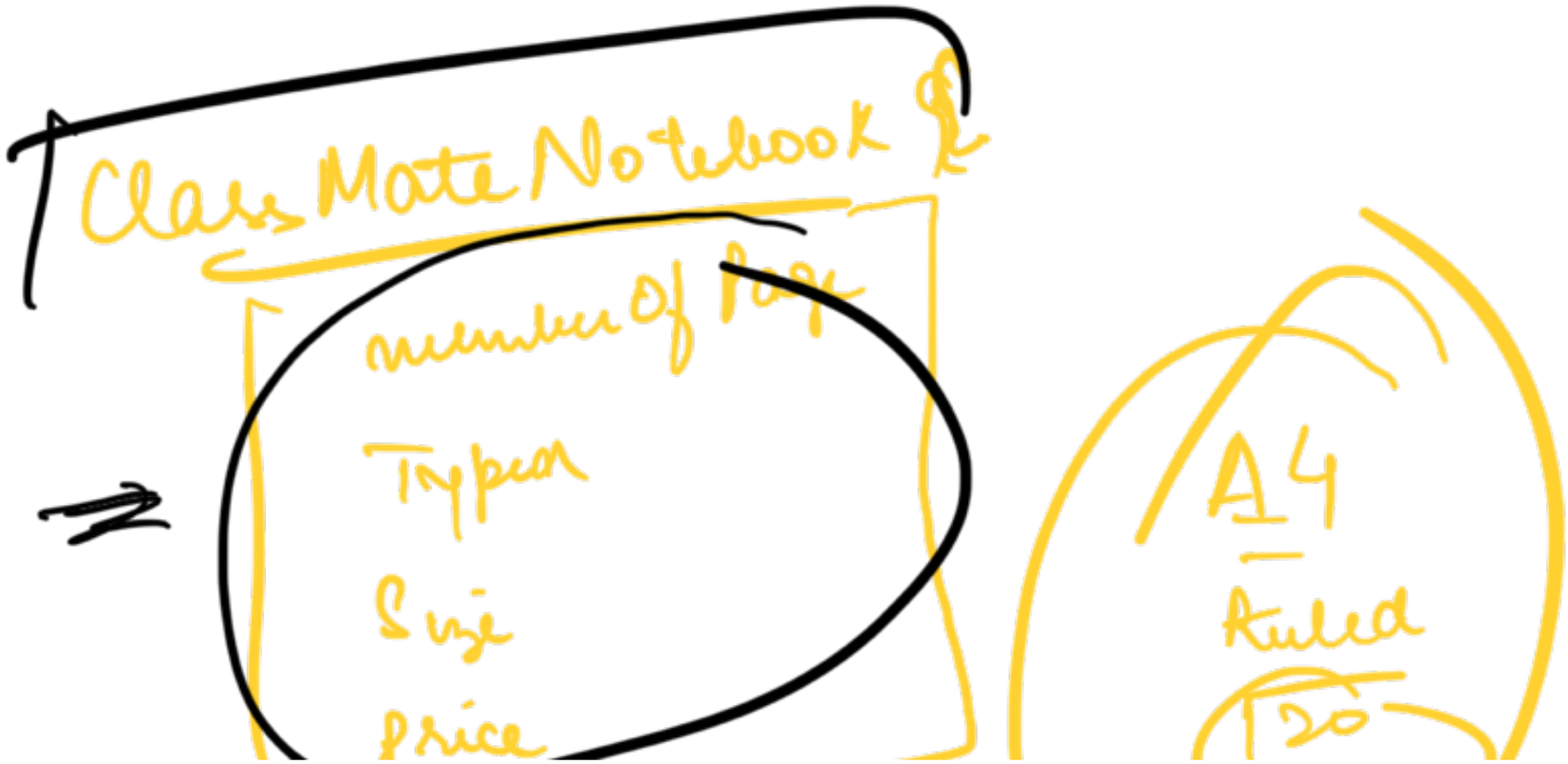
Strategy ⇒ Behavioural

⇒ multiple ways to do something


Entity


Entity
Behaviour
Interface

Prototype Design Pattern



cover image

fun facts

10000

for every notebook we have to create a new MathematicsNotebook object.

```
for (i=0; i<10000; ++i){

    CN cn = new CN();
    cn.numberOfPage = 1000;
```

> cn. ~~[illegible]~~ type = A4 —

Because constructing object is expensive or repetitive, why not store a previous object and just make a copy of it

↓ ↓

DB obj1 = Database(url) ⟸

DB dbb2 = new Database()

dbb2.url = dbb1.url

dbb2.a = dbb1.a

dbb2.b = dbb1.b;

Prototype

→ There can be cases where we might need multiple objects with same

attribute values.

⇒ But creation of those objects from scratch might be inefficient

⇒ Making a copy of the object is better.

⇒ The class itself should in that particular case support creating a copy

new Database(url)

The original object acts like a prototype

Out of

of itself.

class CloneNotebook {

pages

type

facts

Desi

CN clone() {

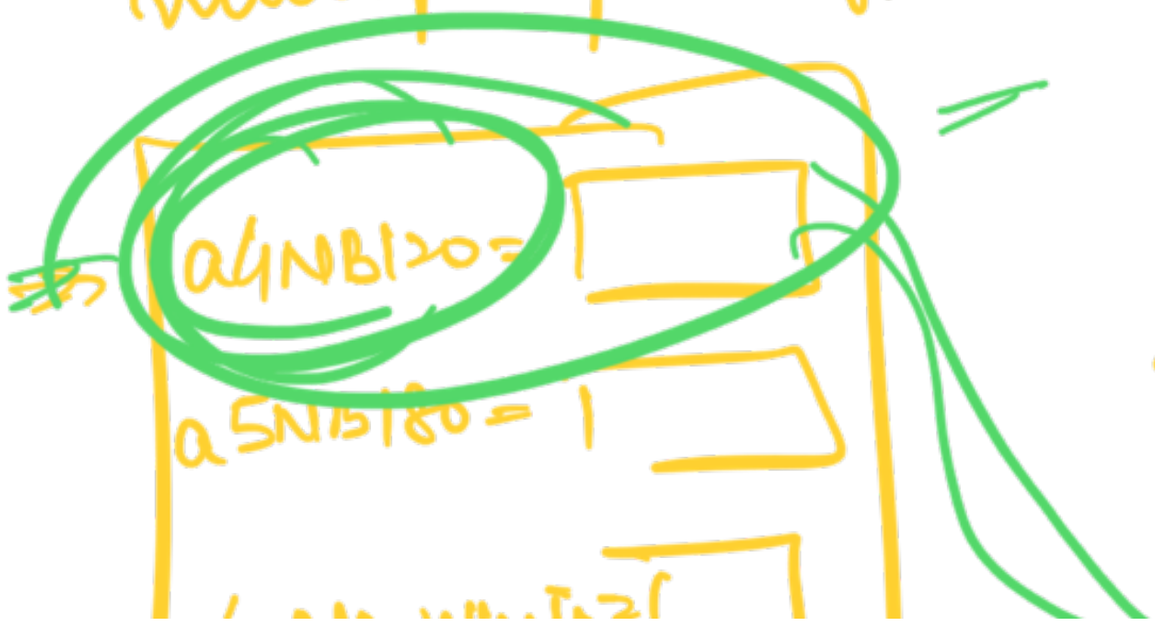
CN cn2 = new CN();

cn2.pages = this.pages

cn2.type = this.type

return 0;

When we need to construct copies from multiple prototypes.



for(i=0; i<1000; ++i){

Cloning output

copyNB

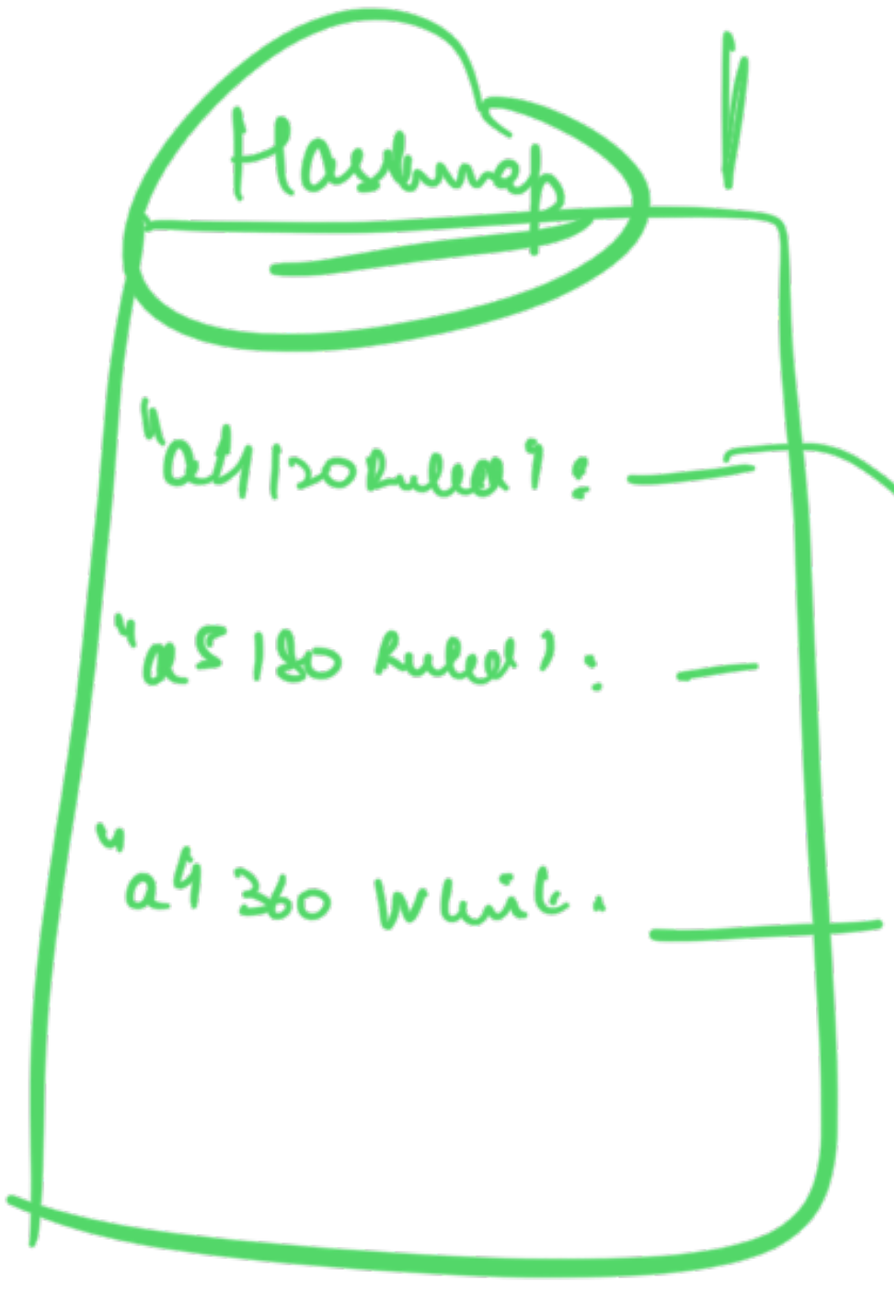
→ copyNB. fact = ——

→ copyNB. area = ——

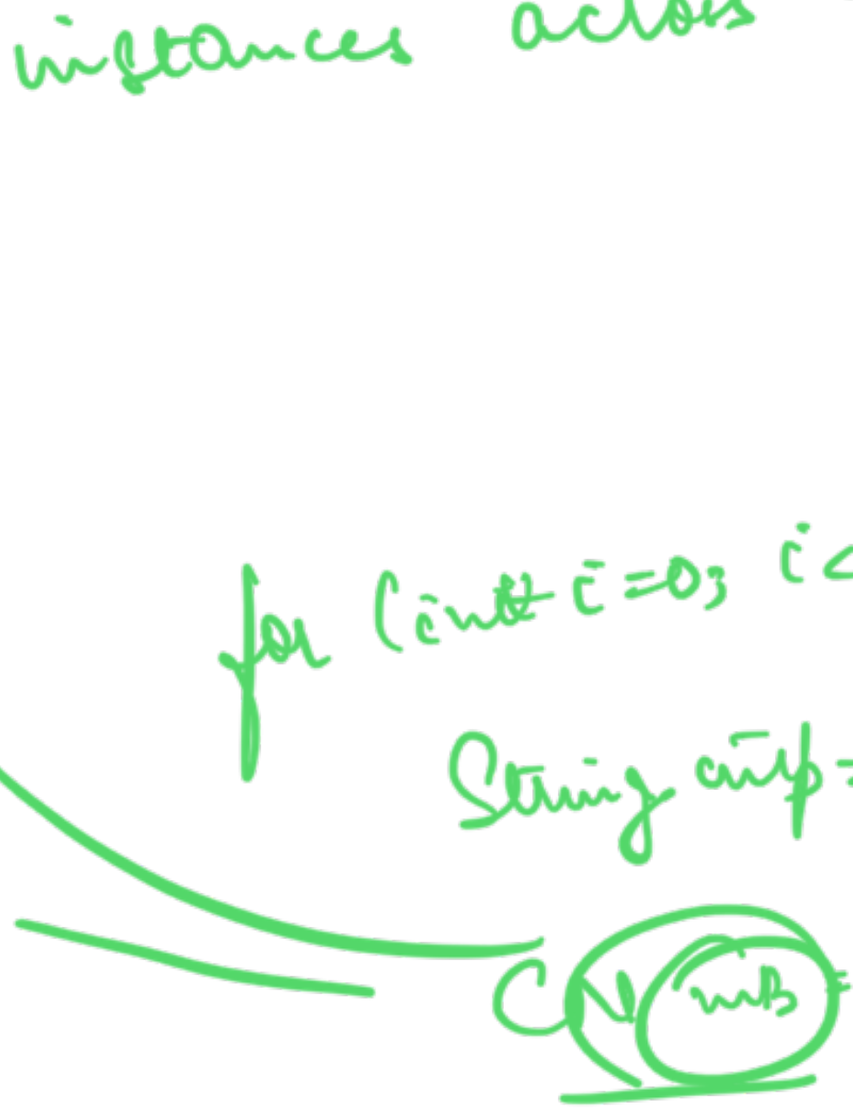
We have 10 different types of object of a class in my system.

Based upon the input I want to create

10000 obj instances across then 10 obje-




```
for (int i=0; i< 10000; ++i)
    String url= "all180kubet"
```



hashmap.get()
.clone()

Assignment

→ Score

→ ~~Code~~ description

∴ → difficulty

30

HARD

→ input

→ output

→ type

Objective

Subjective

Numerical

HashMap<String, Assignment>

HashMap

Object
Hard

Subjective
easy

Objective
medium

score = 500
descript = "
diff = HARD
input = "
output = "
type = OBJECTIVE

=>

OBJECTIVE
EASY

OBJECTIVE
MEDIUM

OBJECTIVE

SUBJECTIV

HARD

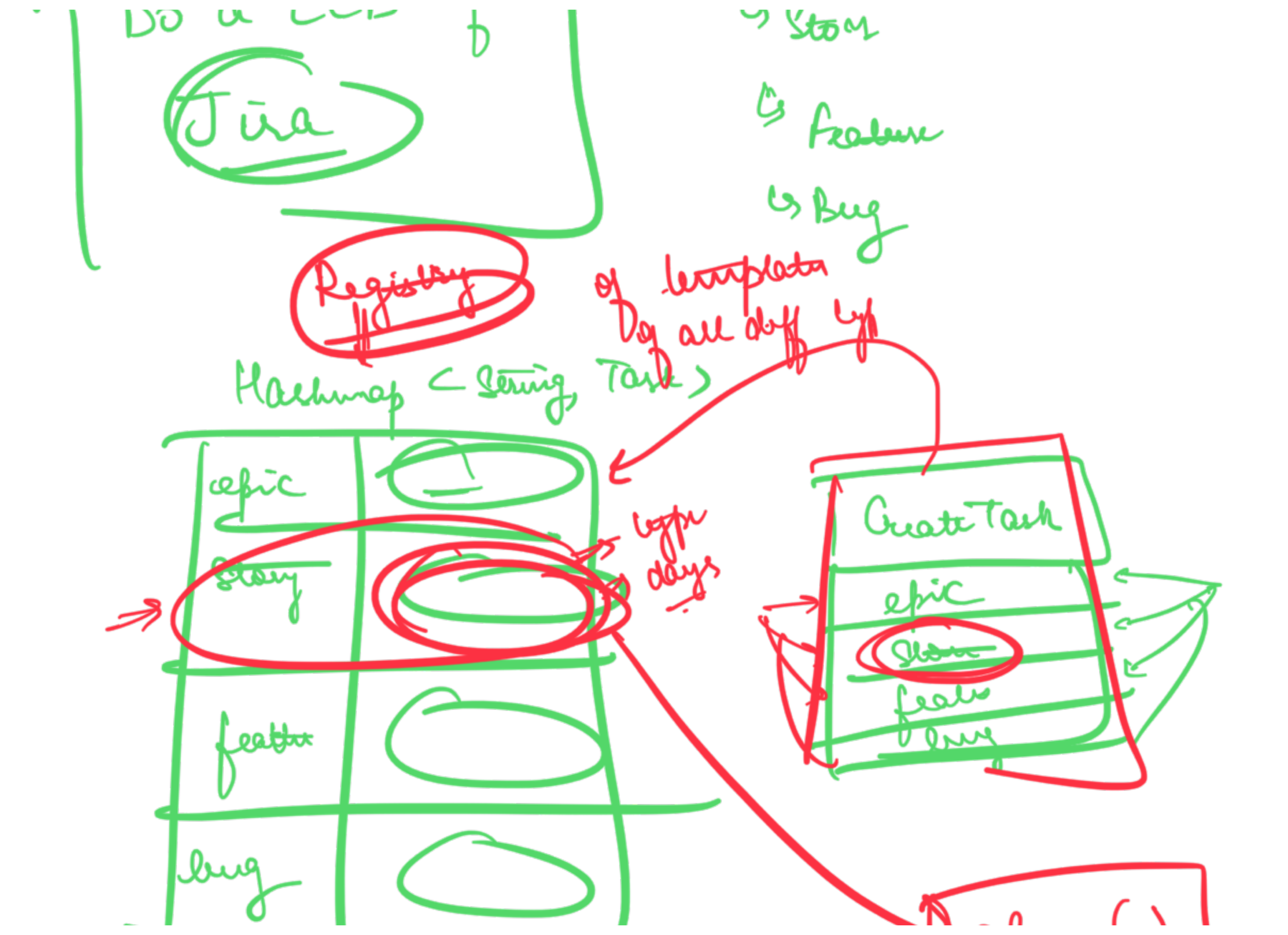
EASY

ASSIGNMEN

HARD

Task

↳ Epic


Jira
Story
Feature
Bug
Registry
of template
Hashmap <String, Task>
epic
story
bug
type
days
Create Task
epic
Story
bug

clone()

Prototype Registry design pattern

Task Registry

HashMap<String, Task>
tasks();

register(task, name)

get(name)

Task

// allow making copy of that obj
clone() {
Task t = new Task
t.type = this.type
t.time = this.time
return t
}

Client

Registry r =

Task epicTemp = new Task)

r . register ("epic", epicTask)

{

CreateTask (String type){

Task t = registry . get (type) . clone ( )