LLD II - Prototype and Factory design patterns

* Prototype
    * basic
    * Prototype Registry
* Factory
    * Simple Factory
    * Factory method
    * Abstract factory — next

Prototype

Real world

Prototype

↳ copy

↳ modify

Research

make a copy

modify

Prototype

→ it is hard to create a new instance

→ Games

→ Testing

(x, y, width, length)

⇓

pixels

Computationally

Tree → prototype

→ make a copy of it

→ x, y

Testing

API - user

↳ user objects

↳ random.me

↓

user name

user → copy

Prototype

• clone

copy control..

Step 1

User user = new User();

User newUser = user.clone();
newUser.setIp(2);

Prototype

↳ when creating an object takes time

↳ User user = new Student();

Subclasses

User user = prototype.clone();

Client

User user =

new Student()

Library

User

Backend

incompatible

① You are tied to the subclass

② You need to know the subclasses

prototype

prototype.clone

Registry

Student {psd}

→ Instructor {jobExp}

prototype

prototypes → type

Registry {

add Prototype (User u)

get Proto type (UserType type)

}

Student {

psp: 100

name: Tantra

}. clone ()

Instructor {

jobExp:

}

The real Prototype

Instance Prototype.

---

class UserRegistry {

Map<UserType, ...> get(UserType) }

add(User)

}

Game Object

Background Foreground Game

① Create clonable interface

Graphical Object {

clone();

}

② Create sub classes

↳ bg Object
↳ fg Object
→ Graphical Object

③ Implement clone

④ Create a registry

```
Map<Object> reg {
    add();
    get();
}
```

(5) Create prototypes and add to registry

6:23 - 6:28

10:58

Have I gotten rid of all subclass usage?

Prototype

↳ Subclasses

Factory

↳ Frontend

↳ Configuration for different env

2 → dev

→ prod

↳ Database

JDBC

Database db = Database.create(
url
)

mysql

Mongo

url

"mysql://host:port"

"mongo://host:port

→ On the basis of a parameter, (url)

Create objects of different type

→ decoupling our subclasses

Form 2

```
Button primary = new Primary();
Button round = new Round();
```

tightly couples my code

Factory pattern

Factory → Simple Factory

→ Factory Method

→ Abstract Factory

} next

Simple Factory

↳ Not a OOP

* We want to remove subclass from client code
* Construction logic is complicated.

```
if (Room.isNew)
    color = blue

if (room is half filled)
    color = red
```

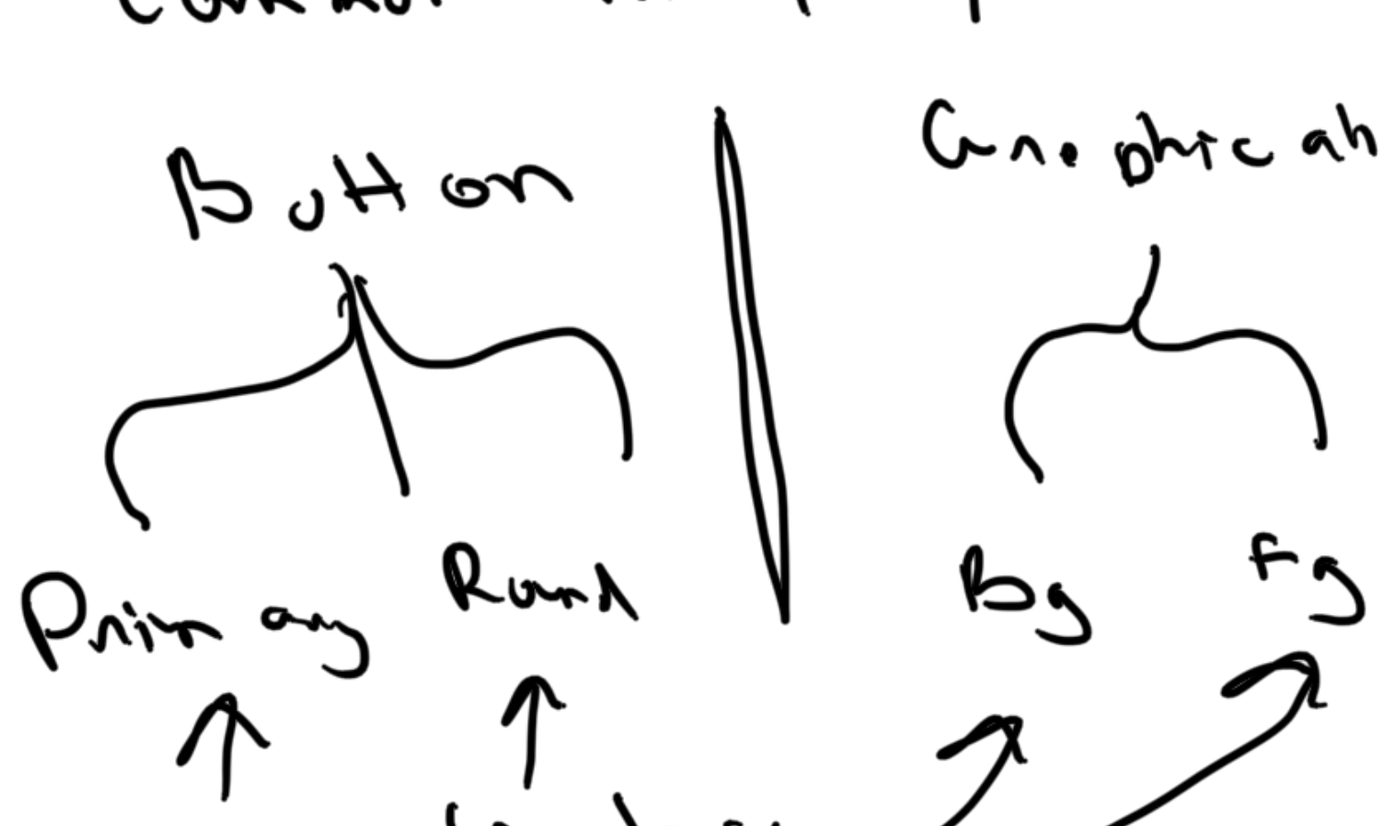
* Common pleu to create ny complex objeAs

```
Button button = new Primary()
Button button = ButtonFactory.creat(type)
```

Create whatever you want

Implement Simple factory

① Create common interface / ac



② Create

(2)

(3) Create factory class

→ Static method
to create instance

→ conditionally
↳ type
create objects.