

Software Ingenieritza

Bomberman Sprint 2

Patroien Txostena

Asier Barrio, Asier Las Hayas, Gaizka Divasson eta Mikel Eguia

2025ko Apirilak 09



Aurkibidea

1	Klase Diagrama	3
2	Patroien informazioa	4
2.1	Strategy	4
2.2	Factory	5
3	Patroien Inplementazioa	6
3.1	BlokeFactory	6
3.2	BombaStrategy	7
3.3	LaberintoFactory	9
3.4	JokalariFactory	10

2 Patroien informazioa

Atal honetan gure proiektuan inplementatutako patroien informazioa aipatuko dugu:

2.1 Strategy

Strategy patroia portaera-patroien kategoriakoa den diseinu-patroi bat da. Bere helburu nagusia algoritmo-familia bat definitzea da, bakoitza bere aldetik kapsulatzea eta exekuzio-denboran trukagarriak izatea ahalbidetzea. Patroi horri esker, testuinguru jakin batean erabiltzen den algoritmoa malgutasunez alda daiteke, hura aipatzen duen kodea aldatu beharrik gabe, eta horrek objektuetara bideratutako programazioaren printzipio irekia/itxia errazten du: hedapenerako irekia, baina aldatzeko itxia.

Strategy patroia ideia nagusia portaera (algoritmoa) eta erabiltzen duen objektua bereiztea da. Horretarako, portaera orokorra irudikatzen duen interfaze komun bat definitzen da (adibidez, Ordenamendua, Mugimendua, Ordainketa, etab.), eta interfaze horretatik abiatuz, estrategia espezifikoak irudikatzen dituzten inplementazio zehatz ugari sortzen dira (adibidez, OrdenamenduaBur-buila, OrdenamenduaQuickSort, etab.). Portaera hori erabiltzen duen klaseak, Testuingurua izenekoak, estrategia-interfazeari erreferentzia egiten dio, eta erabiltzen ari den estrategia zehatzaren esku uzten du portaeraren gauzatzea.

Strategy patroia abantaila argi bat baldintza konplexuak edo if-else edo switch-case bloke anitzak kentzea da. Zer algoritmo erabili erabakitzeak baldintzazko logika idatzi ordez, besterik gabe, estrategia objektua exekuzio denboran alda daiteke. Horren ondorioz, kodea garbiagoa, iraunkorragoa eta hedagarriagoa da.

Adibidez, bideojoko batean, etsai batek mugitzeko modu desberdinak izan ditzake: jokalaria jarraitzea, ausaz mugitzea edo eremu bat patruilatzea. Jokabide horiek guztiak klase berean kodetu beharrean, baldintzekin, mugimendu-estrategiak defini daitezke (MugimenduaJarraipena, AusazkoMugimendua, MugimenduaPatruila), Mugimendua interfaze komun bat inplementatzeko, eta, ondoren, etsaiak interfaze horren instantzia bat besterik ez du erabiltzen, dinamikoki alda daitekeena jokoaren une bakoitzean nahi den portaeraren arabera.

Eredu hori oso erabilgarria da, halaber, algoritmoak maiz aldatzen diren aplikazioetan, hala nola ordainketa-sistemetan (ordainketa txartelarekin, PayPalekin, kriptomonetekin eta abarrekin egin daiteke), irudi-iragazkietan, datu-balidatzaileetan, edo baita pertsonaien portaerak inguruaren arabera aldatzeko adimen artifizialean ere.

Laburbilduz, Strategy ereduak kodea berrerabiltzea sustatzen du, klaseen antolaketa hobetzen du, objektuen arteko akoplamendua murrizten du eta sistemaren portaera zabaltzeko malgutasun handiagoa ahalbidetzen du, dagoen

logika aldatu gabe. Sistema konplexuetan diseinu garbia eta moldagarria mantentzeko tresna indartsua da.

2.2 Factory

Factory patroia, Factory Method izenez ere ezaguna, sorkuntza-diseinuko patroia bat da, eta bere helburu nagusia objektuak sortzeko prozesua abstraitzea da, sorkuntza hori klase edo metodo espezializatu baten esku utziz. Objektuak new operadorearekin zuzenean instantziatu beharrean, eredu honek fabrika-metodo bat erabiltzen du, zer objektu sortu eta instantzia egoki bat itzultzeko. Horri esker, sistemak objektuekin lan egin dezake bere klase zehatza ezagutu gabe. Abstrakzio hori bereziki erabilgarria da kodeak lotutako objektu-familia batekin lan egin behar duenean edo eskatuko diren klaseak parametro edo baldintza jakin batzuen arabera alda daitezkeenean.

Patroiaren oinarritzko egitura fabrika-metodoa deklaratzeko duen oinarritzko klase abstraktu bat da, eta metodo hori inplementatu eta instantzia espezifikoak itzultzen dituzten azpiklase zehatz bat edo batzuk. Bezero batek ere erabiltzeko objektu horiek interfaze komunaren bidez, nola sortu diren jakin beharrik gabe. Horrek bezeroaren eta klase zehatzen arteko akoplamendua murrizten du, sistemaren hedagarritasuna erraztuz. Adibidez, objektu mota berri bat gehitu nahi bada, fabrika-metodoaren inplementazio berri bat sortu behar da bezero-kodea ukitu gabe.

Ohiko erabilera-kasu bat jokalaria mailaren arabera etsai mota desberdinak sortu behar diren bideojoko bat litzateke. Programa osoan baldintza luzeak erabili beharrean, Etsaia interfaze bat eta hainbat inplementazio defini daitezke, hala nola EtsaiaErraza, EtsaiNormala edo EtsaiZaila, eta fabrika-klase bat, egungo zailtasunaren arabera zein itzuli erabakitzen duena. Horrela, aurrerago etsai mota berri bat sartu nahi bada, nahikoa izango da beste mota bat gehitzea eta fabrika aldatzea, sistemaren gainerakoa aldatu gabe.

Beste adibide klasiko bat interfaze grafikoko sistemetan da, non fabrika-metodo batek botoi mota desberdinak itzul ditzakeen (BotoiWindows, BotoiMac, BotoiLinux) aplikazioa exekutatzen ari den sistema eragilearen arabera. Botoiak erabiltzen dituen kodeak ez du zertan mota jakin; fabrika-metodoari deitzen dio eta interfaze komunarekin lan egiten du.

Laburbilduz, Factory ereduak kodearen modularitatea, berrerabilpena eta mantentzea hobetzen du, objektuak sortzeko prozesua erabileratik bereizteko aukera emanez. Egokia da klaseen instantzian malgutasuna behar denean, akoplamendu sendoa saihestu nahi denean edo klase berriak maiz gehituko direla aurreratzen denean; izan ere, sorkuntza-logika zentralizatzen du eta sistema modu ordenatuagoan eta sendoagoan eskalatzea ahalbidetzen du.

3 Patroien Implementazioa

Guk inplementatutako patroien kodea hurrengoa da:

3.1 BlokeFactory

BlokeFactory izeneko Java klase honek Singleton patroia eta Factory patroia inplementatzen ditu Bloke motako objektuak sortzeko. Bere helburua bloke-mota desberdinen sorrera zentralizatzea da (Gogorra, Biguna eta Hutsa) balio oso baten arabera (*pMota*), eta programa osoan fabrika honen instantzia bakarra dagoela ziurtatzea.

```
1 public class BlokeFactory {
2     private static BlokeFactory nireBlokeFactory;
3
4
5     private BlokeFactory() {
6
7     }
8     public static BlokeFactory getNireBlokeFactory() {
9         if (nireBlokeFactory == null) {
10             nireBlokeFactory = new BlokeFactory();
11         }
12         return nireBlokeFactory;
13     }
14
15     public Bloke sortuBlokea(int pMota, int pX, int pY) {
16         Bloke nireBloke = null;
17         if (pMota == 1) {
18             nireBloke = new Gogorra(pX,pY);
19         }
20         else if (pMota == 2){
21             nireBloke = new Biguna(pX, pY);
22         }
23         else {
24             nireBloke = new Hutsa(pX, pY);
25         }
26         return nireBloke;
27     }
28 }
29 }
```

3.2 BombaStrategy

BombaStrategy klasea Javako klase abstraktu bat da, joko batean bonba batek duen portaera orokorra irudikatzen duena, bereziki eztanda-tenporizadoreari dagokionez. Lehertzean portaera espezifikoa inplementatzen dituzten azpik-laseek erabil dezaketen bonba-estrategia bat definitzen du. Haren diseinuari esker, bonba bakoitzak erregresio-kontu automatiko bat du, lehertu arte, baita posizio bat eta mekanismo bat ere, leherketa gertatzen denean ekintza bat gauzatzeko.

```
1 public abstract class BombaStrategy {
2     //private int radioa;
3     protected int X;
4     protected int Y;
5     protected boolean eztanda;
6     protected int kont;
7     protected int PERIODO = 3;
8     protected Timer timer = null;
9
10    /*protected BombaStrategy(*int pRadioa, int pX, int pY){
11        //this.radioa = pRadioa;
12        this.X = pX;
13        this.Y = pY;
14        eztanda = true;
15        kont = PERIODO;
16        TimerTask timerTask = new TimerTask() {
17            @Override
18            public void run() {
19                updateKont();
20            }
21        };
22        timer = new Timer();
23        timer.scheduleAtFixedRate(timerTask, 0, 1000);
24    } */
25    public void timerHasi() {
26        TimerTask timerTask = new TimerTask() {
27            @Override
28            public void run() {
29                updateKont();
30            }
31        };
32        timer = new Timer();
33        timer.scheduleAtFixedRate(timerTask, 0, 1000);
34    }
35
36
37
38    public int getX() {
39        return this.X;
40    }
```

```
41
42     public int getY() {
43         return this.Y;
44     }
45
46     public void setX(int pX) {
47         X =pX;
48     }
49
50     public void setY(int pY) {
51         Y = pY;
52     }
53
54     protected void updateKont() {
55         kont--;
56         if (kont == 0) {
57             kont = PERIODO;
58             eztanda = true;
59             apurtu();
60             timer.cancel();
61         }
62     }
63
64     protected void apurtu() {
65     }
66
67 }
```


3.3 LaberintoFactory

LaberintoFactory klaseak Singleton diseinu-patroia inplementatzen du, programa exekutatzen den bitartean fabrikako instantzia bakarra dagoela ziurtatuz. Hori lortzeko, *nLF* klaseko instantzia pribatu eta estatiko bat deklaratzeko da, eta eraikitzaile pribatu bat definitzeko da, klasetik kanpo objektuak zuzenean sor ez daitezkeen.

getLF() metodoa publikoa eta estatikoa da, eta *LaberintoFactory* instantzia bakarrera sarbidea ematen du. Instantzia hori oraindik sortu ez bada (*nLF* = null), metodoak hasieratu egingo du. Bestela, lehendik dagoen instantzia itzuli besterik ez du egiten.

Bestalde, *SortuLabirinto(int pMota)* metodoak labirinto motako objektuak sortzea ahalbidetzen du, jasotako parametroaren arabera. Switch egitura bat erabiltzen du sortu beharreko labirinto mota zehazteko: balioa 1 bada, *LaberintoClassic* bat sortzen da; 2 bada, *LaberintoSoft* bat; eta 3 bada, *LaberintoEmpty* bat. Metodo honek labirinto-fabrika baten moduan jotzen du, eta emandako parametroaren arabera hainbat labirinto mota dinamikoki sortzeko aukera ematen du.

```

1  public class LaberintoFactory {
2      private static LaberintoFactory nLF = null;
3
4      private LaberintoFactory(){}
5
6      public static LaberintoFactory getLF() {
7          if(nLF == null)
8              nLF = new LaberintoFactory();
9          return nLF;
10     }
11
12     public Laberinto sortuLaberinto(int pMota) {
13         Laberinto nLE = null;
14         switch(pMota) {
15             case 1: nLE = new LaberintoClassic(); break;
16             case 2: nLE = new LaberintoSoft(); break;
17             case 3: nLE = new LaberintoEmpty(); break;
18         }
19         return nLE;
20     }
21 }
22
23 }

```

3.4 JokalariFactory

JokalariFactory klaseak Singleton diseinu-patroiari jarraitzen dio, eta horrek esan nahi du ikasgelako instantzia bakarra dagoela programa exekutatzen den bitartean. Horretarako, *nireJokalariFactory* deituriko atributu estatiko bat eta klasetik kanpo objektuak sortzea eragozten duen eraikitzaile pribatu bat definitzen dira. *GetNireJokalariFactory()* metodoa arduratzen da instantzia bakar hori itzultzeaz: oraindik sortu ez bada, hasieratu egiten da; bestela, lehendik dagoena itzultzen da. Bestalde, *sortuJokalari(int i, int pX, int pY)* metodoak Jokalari-Eredu motako objektu bat sortzea eta itzultzea ahalbidetzen du, *i* parametroaren balioaren arabera. *i* 1 bada, *BombbermanTxuria* motako jokalaria bat sortzen da *pX* eta *pY* koordinatuekin; 2 bada, *BombermanBeltza* bat sortzen da koordinatu berberekin. Horrela, klase honek hainbat motatako jokalarien sorrera zentralizatu eta kontrolatzen du, eta horien kudeaketa modu bateratu eta koherentean errazten du.

```

1  public class JokalariFactory {
2      private static JokalariFactory nireJokalariFactory;
3
4
5      private JokalariFactory() {
6
7      }
8
9      public JokalariFactory getNireJokalariFactory() {
10         if (nireJokalariFactory == null) {
11             nireJokalariFactory = new JokalariFactory();
12         }
13         return nireJokalariFactory;
14     }
15
16     public JokalariEredu sortuJokalari(int i, int pX, int pY) {
17         JokalariEredu bomberman = null;
18         if(i == 1) {          //bomberman txuria sortu
19             bomberman = new BombermanTxuria(pX,pY);
20         }
21         else if (i == 2) {    //bomberman beltza sortu
22             bomberman = new BombermanBeltza(pX, pY);
23         }
24         return bomberman;
25     }
26
27 }

```