Programfejlesztés Scala nyelven: Gépi tanulás magasabbrendű függvényekkel

Rigó Ádám

2013. április 22.

Tartalomjegyzék

1. Bevezető

TODO

2. A Scala nyelvről

TODO

3. Implementált algoritmusok

TODO

4. Tervezési kérdések

Ez a fejezet azokról a problémákról és lehetséges megoldásakiról szól amik a gépi tanulással kapcsolatos algoritmusok implementálása közben felmerültek. Először a problémát mindig általánosan mutatom be, majd ezt követően egy konkrét algoritmus(család) Scala nyelven történő implementációján keresztül.

4.1. Metóduskombinációk problémája

4.1.1. Motiváció

Egy interfész implementálása során előfordulhat, hogy több metódusnak is több lehetséges implementációja van és emiatt a különböző metódusok implementációinak nagyon nagy számú értelmes kombinációja lehetséges. Amennyiben a lehetséges implementációkat a hagyományos öröklődés segítségével szeretnénk modellezni, nagyon nagy számú olyan osztályra lehet szükségünk amiknek interfésze megegyezik, csupán az egyes konkrét kombinációk kedvéért vezettük be őket az osztálychierarchiánkba. (Például ha 3 metódusunk van és mindegyiknek 5 lehetséges implementációja, továbbá ezen implementációk minden kombinációja értelmes, máris 5*5*5=125 különböző osztályra lehet szükségünk.) Különösen pazarló lehet ez, például akkor ha a lehetséges kombinációknak egy részét fogjuk csak a programunkban használni, viszont fordítási időben még nem tudjuk, hogy pontosan melyeket. Ezt a pazarlást el szeretnénk kerülni, mivel egyrészt eleve macerás, hosszú és gépies munka a rengeteg osztály definiálása, másrészt (még ha előbbit esetleg sikerül is automatizálnunk), nagymértékben rontja a kód olvashatóságát és karbantarthatóságát. (Például ha egy olyan új metódust szeretnénk hozzávenni az osztályainkhoz, amelynek szintén több implementációja van, az azt eredményezi, hogy az összes eddigi osztály kódját módosítanunk kell és további osztályok definiálásra kényszeríthet minket, megtöbbszörözve ezzel eddig sem kicsiny osztályhierarchiánkban szereplő osztályaink számát.)

4.1.2. Objektumorientált megközelítés

Megoldás dinamikus öröklődés segíytségével. Ha nem szeretnénk az összes lehetséges értelmes metóduskombinációnak egy külön osztályt létrehozni és mindig az adott helyzetben szükségeset példányosítani, úgy kell definiálnunk egy osztályt, hogy példányainak viselkedését futásidőben a számunkra megfelelően alakíthassuk. Akár az Állapot (State), akár a Straté-

gia (Strategy) tervezési mintát alkalmazzuk a probléma megoldására, ezek használata nagy számú metódus esetén nagyon kényelmetlen (és más egyéb problémákat is felvet).

Helyettük tekintsük a következő interfészt, ami olyan metódusok deklarációit tartalmazza, amelyek közül mindnek többféle implementációja lehetséges.

```
public interface DynamicInterface {
        public void DynamicMethod1();
        public void DynamicMethod2();
        public void DynamicMethod3();
}
```

Az interfészt implementálva létrehozunk egy olyan osztályt, amely lehetőséget biztosít arra, hogy más osztályok viselkedését vegye át futásidőben.

```
public class DynamicMethods implements DynamicInterface{

        DynamicInterface innerLayer = null;

        DynamicMethods(DynamicInterface innerLayer)
        {
                this.innerLayer = innerLayer;
        }

        DynamicMethods()
        {
        }

        public void DynamicMethod1()
        {
                if(innerLayer != null)
                        innerLayer.DynamicMethod1();
                else
                        System.out.println("Nincs definialva a
                            DynamicMethod1");
        }
        .
        .
        .
}
```

Amikor létrehozzuk az osztályt, annak konstruktor paraméterként megadunk

egy másik osztályt, ami hasonlóképpen a DynamicInterface interfészt implementálja, tehát rendelkezik azoknak a metódusoknak valamilyen implementációjával, amiket futásidőben szeretnénk megadni. A fenti osztályunkban egy alapértelmezett viselkedést definálunk, amelyet a leszármazott osztályokban később felüldefinálhatunk.

```
public class DynamicMethods12 extends DynamicMethods {

        DynamicMethods12(DynamicInterface innerLayer)
        {
                this.innerLayer = innerLayer;
        }

        DynamicMethods12()
        {
        }

        public void DynamicMethod1()
        {
            //This is the implementation of DynamicMethod1
                System.out.println("DynamicMethod1");
        }

        public void DynamicMethod2()
        {
            //This is the implementation of DynamicMethod2
                System.out.println("DynamicMethod2");
        }
}
```

Az alapértelmezett viselekdést két metódus esetén definiáltunk felül és a belőle létrejövő objektumokat hasonlóan más osztályoknak is adhatjuk konstruktorparaméterül. Az összes nem felüldefiniált metódus alapértelmezett módon fog viselkedni, tehát ha van paraméterül kapott objektum, akkor rekurzívan annak hívódik meg a megfelelő metódusa, mindaddig amíg nem talál egy az alapértelmezettől eltérő implementációt, vagy egy olyan objektumhoz el nem jut, amelynek nem volt a DynamicInterface-t implementáló konstruktorparamétere és így nincs lehetősége a hívást továbbhárítani.

Számos hátránya van azonban ennek a megközelítésnek is. Előfordulhat, hogy egy metódusnak csak az alapértelmezett implementációja marad, tehát nem kapunk semmilyen figyelmeztetést arról, ha valamelyik tagfüggény-

hez elfelejtettünk értelmes implementációt rendeleni, márpedig ez jelentősen megnehezítheti programjainkban a hibák megtalálását. Probléma lehet az is, ha véletlenül több implementációt is hozzárendelünk egy metódushoz. Ekkor abban az objektumban lévő implementáció fog végrehajtódni, amelyik valamilyen módon az összes többi a metódust másképp definiáló objektumot tartalmazza. Sajnos ez sok metódus esetén nem biztos, hogy mindig az, aminek meghívására számítunk (mivel egyszerűen túl bonyolult a helyzet ahhoz, hogy átlássuk), másrészt további problémákhoz vezethet, például akkor ha egyik objektumban lévő implementáció szeretne felhasználni egy őt tartalmazó objetkumban lévő implementációt.

Megoldás delegáltak segítségével. A C# nyelvben a delegáltak teszik lehetővé, hogy egy függvényt paraméterként adjunk át. Deklarálásukkor meg kell adnunk az átadni kívánt függvény szignatúráját és visszatérési értékét. Amikor megpróbálunk átadni egy függvényt, a fordítóprogram ellenőrzni, hogy a függvény szignatúrája és visszatérési értéke megfelel-e az előzetesen vártnak, így biztosítva az átadott függvények típushelyességét.

```
    public delegate void DynamicMethod1();
    public delegate void DynamicMethod2();
    public delegate void DynamicMethod3();
```

Tekintsük a korábbi interfészt.

```
   public interface IDynamicInterface
    {
        void DynamicMethod1();
        void DynamicMethod2();
        void DynamicMethod3();
    }
}
```

Az interfészt implementáló osztály konstruktor paraméterként várja a delegáltak és a metódusait a delegáltak segítségével implementálja. Ebben az implementációban a delegáltak segítségével nyílik lehetőség arra, hogy az osztály viselkedését futási időben adhassuk meg.

```
public class DynamicMethods : IDynamicInterface
    {
        private DynamicMethod1 Dm1;
        private DynamicMethod2 Dm2;
```

```
        private DynamicMethod3 Dm3;

        public DynamicMethods(DynamicMethod1 dm1, DynamicMethod2
            dm2, DynamicMethod3 dm3)
        {
            Dm1 = dm1;
            Dm2 = dm2;
            Dm3 = dm3;
        }

        public void DynamicMethod1()
        {
            Dm1();
        }
                .
                .
                .
```

A fenti szerkezet használatához definiálnunk kell az átadni kívánt függvényt (amely megfelel a delegált deklaráláskor megadott szignatúrának és visszatérési értéknek).

```
        static void f1()
        {
            Console.WriteLine("f1 method");
        }
```

Majd a függvénydefiníciót követően azokat delegáltként átadva állíthatjuk be dinamikusan az objektum megfelelő metódusainak implementációjaként.

```
DynamicMethods d = new DynamicMethods(
                    new DynamicMethod1(f1),
                    new DynamicMethod2(f2),
                    new DynamicMethod3(f3)
                    );
            d.DynamicMethod1(); //"f1 method"
```

A megoldás legnagyobb hátránya, hogy a függvények noha egymást képesek meghívni, az objektum rejtett adattagjait és függvényeit implementációjakor nem képes felhasználni. (TODO: hátrányokat részletesebben kifejteni)

4.1.3. Funkcionális megközelítés

Megoldás magasabbrendű függvények segítségével. Noha a fenti megoldás implementálható Scalaban is, annak hátrnyaitól szeretnénk megszabadulni, miközben az egyes metódusokat mindenképpen futásidőben szeretnénk kombinálni.

A korábbi interface definíciónak megfelelő trait Scalaban:

```
trait DynamicInterface{
   def DynamicMethod1()
   def DynamicMethod2()
   def DynamicMethod3()
}
```

A fenti traitből származtathatunk egy osztályt, amely a traitben deklarált metódusokhoz egy implementációt rendel. A magasabbrendű függvényeknek köszönhetően van lehetőségünk ezeket az implementációkat konstruktor paraméterként átadni, így biztosítva annak lehetőségét, hogy elég legyen az objektum létrehozásakor eldönteni az egyes metódusokhoz rendelt konkrét implementációt.

```
class DynamicMethotds[T1, T2, T3](
        val f1: () => T1,
        val f2: () => T2,
        val f3: () => T3) extends DynamicInterface{

   def DynamicMethod1() = f1()
   def DynamicMethod2() = f2()
   def DynamicMethod3() = f3()

    if(f1 == null){
          println("Not implemented f1")
       }
       .
       .
       .
}
```

Ennek a megoldásnak több előnye is van a tisztán objektumorientált, dinamikus öröklődést használó implementációval szemben.

Ha véletlenül elfelejtünk átadni egy implementált függvényt konstruktorpara-

méternek, akkor az így még fordítási időben kiderül (míg a dinamikus öröklődés esetén - különösen ha sok függvényünk van és az új objektumunkat sok másik objektum kombinálásával hozzuk létre - könnyen megfeledkezhetünk egy függvény implementációjának megadásáról, ráadásul ez a hiba futásidőben is csak akkor derül ki, amikor a függvényt amiről megfeledkeztünk meghívjuk). Persze a fordító nem véd meg minket attól, hogy esetleg véletlenül egy null paraméterértéket adjuk át megfelelő konstruktorparaméterek helyett, így amennyiben egy a konstruktorban a null paraméterértéknek megfeleltetett tagfüggvényt hívnánk meg, rögtön olyan futásidejű hibával találnánk magunkat szemben, aminek oka még az objektum létrehozásának idejére tehető. Mivel az objektum a program futása során akár jóval korábban is létrejöhetett, így a hiba megtalálása akár igen időigényes is lehet. Szerencsére ha ezt az implementációt választjuk, akkor még az objektum létrehozásakor leellenőrizhető, hogy az átadott paraméter null-e és amennyiben igen, még az előtt lehetőségünk van a hiba kezelésére - vagy amennyiben nem sikerült kezelni a hibát a felhasználó értesítésére - hogy az a megfelelő függvényhívás miatt valahol máshol a programban futásidejű hibát okozna. Ily módon elősegítjük, hogy a példányosításnál előforduló esetleges hibák még keletkezésük helyén derüljenek ki, könnyebbé téve ezzel kezelésüket.

További előnye ennek a megoldásnak, hogy a lehetséges tagfüggvények implementációjuk során felhasználhatják a többi lehetséges tagfüggvény implementációját, így csökken a kódismétlés.

Ezekért az előnyökért cserébe azonban több kellemetlenséggel kell fizetnünk. Egyrészt kényelmetlen lehet a tagfüggvények nagy száma esetén az összeset átadni konstruktorparaméterként, másrészt a konstruktoparaméterként átadott metódusok definíciója óhatatlanul az osztályon kívül helyezkedik el, így nem fér hozzá az osztály rejtett részéhez, ami gondot okoz, ha elvárnánk a metódustól, hogy módosítsa az objektum állapotát vagy használja annak a külvilág számára nem látható metódusait.

Utóbbi problémára megoldás lehetne, ha az osztályon kívüli függvény definíciójának helyéről nem látható adattagokat vagy metódusokat átadnánk a függvénynek paraméterül. Ennek a megoldásnak sajnos nagyon komoly hátrányai vannak. Egyrészt a függvény deklarálásakor ismernünk kell az osztálynak azt a részét, amit a függvény használni kíván, de nem látható számára. Másrészt ha úgy határozunk, hogy további adattagokat vagy metódusokat szeretnénk hozzávenni az osztályhoz, módosíthatnunk kellhet az osztályon kívül definiált függvény(ek) szignatúráját, amennyiben azok implementációja támaszkodna osztályunk frissen definiált részeire. Ez a probléma nem jelentkezik, ha az osztály adattagjai és metódusai láthatóak a külső függvé-

nyek definíciójának helyéről. Szerencsére a Scala nyelvben nagyon finoman lehet szabályozni, hogy az osztály mely részei mely csomagokból legyenek láthatóak, így nincs más dolgunk, mint a külvilág számára elrejteni kívánt adattagokat olyan módosítóval ellátni, amivel még a külső függvényeink számára láthatóak maradnak, de mások számára továbbra is rejtve lesznek.

TODO: csharp delegált(Ok), Java dinamikus öröklődés(ok), ugyanezek Scalaban, OptiML, javítani a Scala résznél az objektum belső állapotára vonatkozó okoskodást (az adattagokat nem kell átadni paraméterül, csak private[vmi object] kell, és a tagfüggvények lehetséges implementációi a valami objectben lesznek együtt.

4.1.4. Konkrét példa a problémára