Stubs – razbijanje eksternih zavisnosti u kodu i testiranje

1. Uvod

Do sada je bilo priče o testovima koji se bave jednostavnim testiranjem metoda. Drugim rečima, dosadašnji testovi su se bavili testiranjem metoda koje vraćaju neku vrednost i ne koriste neke druge objekte prilikom kalkulacije vrednosti koju vraćaju. Isto tako date metode su mogle da vrše neku jednostavnu promenu stanja objekta kom pripadaju.

Nešto realističniji scenario od ovoga jeste da objekat pod testom (objekat čija se metoda testira) ili sistem pod testom (eng. *System under test – SUT*) zavisi od nekog drugog objekta nad kojim nemamo kontrolu ili čija implementacija još nije gotova. Primer ovog drugog objekta može biti neki veb servis koji se poziva iz objekta pod testom. Kod ovakvog scenaria naš test nema kontrolu nad time šta taj drugi objekat odnosno ta zavisnost vraća našem objektu pod testom, niti kako se on ponaša. U cilju testiranja ovakvih scenaria moguće je koristiti *stubs***.**

Stubs se koriste kako bi se izbegao problem sa testiranjem eksternih zavisnosti i *stub* predstavlja zamenu za neku postojeću zavisnost u sistemu, koju je moguće kontrolisati.

2. Primer

U cilju pojašnjenja ovog koncepta i rešavanja problema testiranja eksternih zavisnosti u nastavku je korak po korak objašnjen primer softvera koji vrši proveru ekstenzije nekog fajla. U datom primeru se neće vršiti isčitavanje konkretnih fajlova već samo provera njihove ekstenzije. Kod koji prati ovaj primer je moguće naći na Git serveru[1] u okviru repozitorijuma *Stubs*.

Solution za dati softver se sastoji od dva projekta:

1. *StubExample* – projekat tipa *Class Library (.Net Framework)*
2. *StubExample.UnitTests* – projekat tipa *Class Library (.Net Framework)*

Prvi projekat sadrži kase koje će biti testirane, dok drugi projekat sadrži *unit* testove za klase iz prvog projekta.

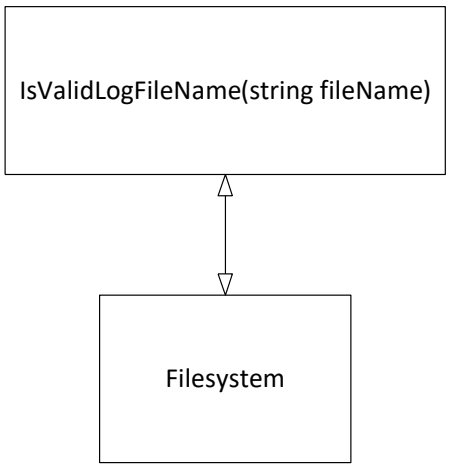
Prvi projekat sadrži klasu *LogAnalyzer* u okviru koje je definisana metoda za proveru ekstenzije nekog fajla na osnovu njegovog punog imena koje se prosleđuje kao parametar datoj metodi. Radi primera predpostavićemo da se sve dozvoljene ekstenzije čuvaju na disku u vidu konfiguracionog fajla. Metoda koja vrši proveru ekstenzije bi mogla da liči na onu koja je prikazana u listingu Listing 1.

[1] Adresa: http://147.91.175.237:8082/Bonobo.Git.Server

Listing 1. Primer metode za proveru validnosti ekstenzije

```
1  public bool IsValidLogFileName(string fileName)
2  {
3     //čitanje konfiguracionog fajla
4     //vraćanje vrednosti true ukoliko je ekstenzija dozvoljena ili false
5     //ukoliko nije
6  }
```

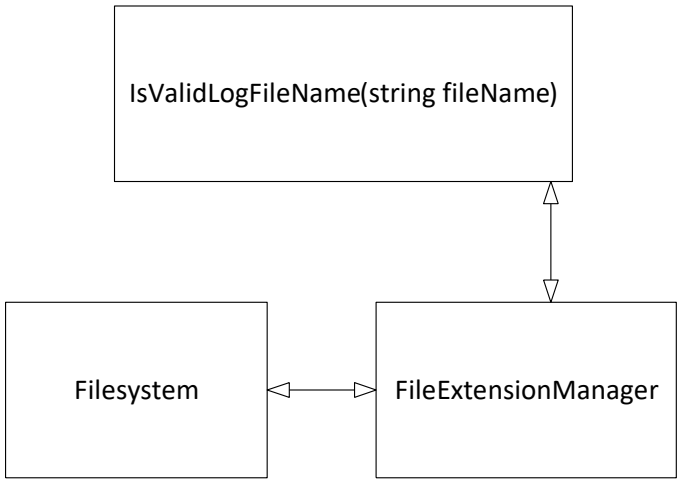
Problem koji leži ovde jeste u zavisnosti date metode od fajlsistema (Slika 1). Da bi testirali ovu metodu praktično treba da sprovedemo integracioni test, koji je dosta zahtevniji od *unit* testa. Ovakav dizajn nije testabilan s obzirom na postojanje eksterne zavisnosti koja može dovesti do greške uprkos tome da naš kod funkcioniše.



Slika 1. Ilustracija zavisnosti metode od fajlsistema

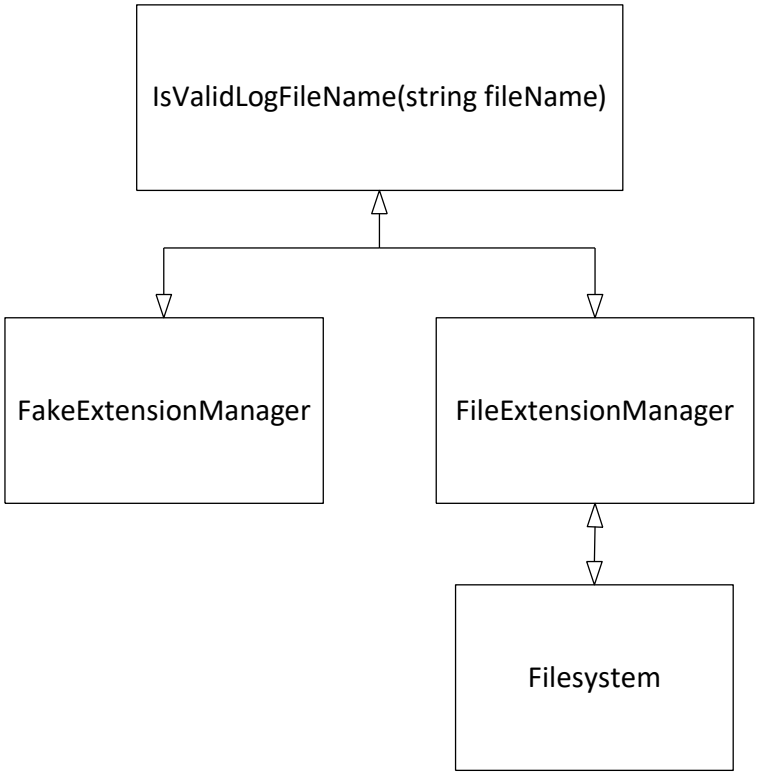
Kako bi rešili dati problem i učinili dizajn više testabilnim potrebno je da uvedemo novi sloj koji će praktično da agregira sve eksterne zavisnosti. U tu svrhu potrebno je identifikovati zavisnost, izdvojiti tu zavisnost u poseban sloj, zameniti implementaciju sa nekom logikom nad kojom imamo kontrolu i tu logiku podmetnuti testu umesto konkretne zavisnosti. Logika koju podmećemo testu zapravo neće imati nikakvog kontakta sa fajlsistemom, već će samo simulirati odgovore koje možemo dobiti čitanjem pomenutog konfiguracionog fajla iz fajlsistema.

Izmeštanjem zavisnosti u poseban sloj dobijamo sledeću strukturu (Slika 2).



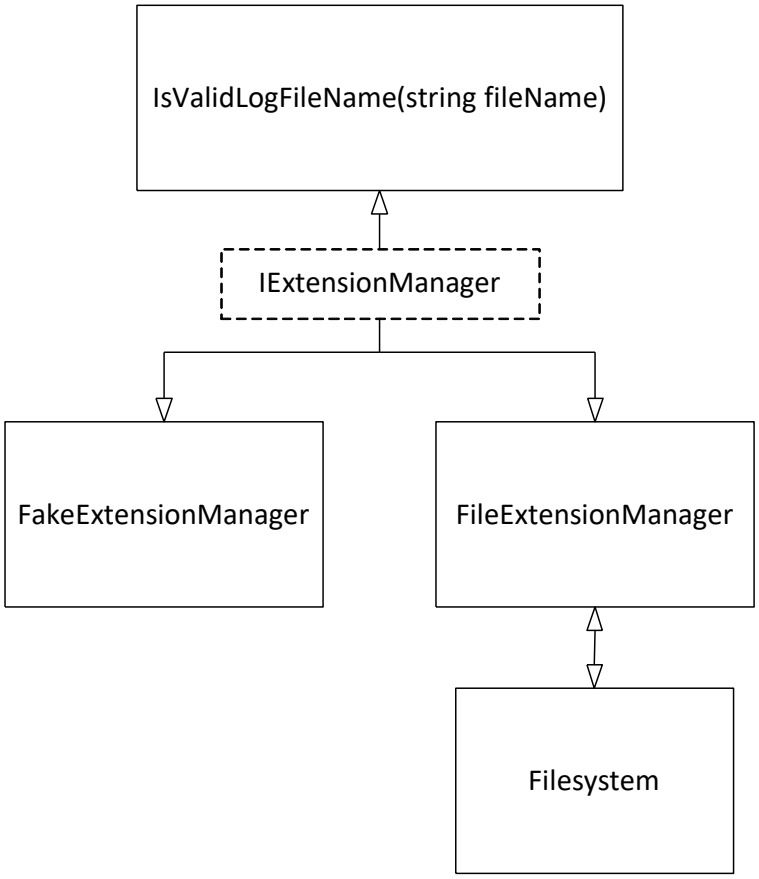
Slika 2. Struktura nakon izmeštanja zavisnosti u poseban sloj

Nakon kreiranja novog sloja potrebno je dodati lažne implementacije date zavisnosti, čime se dolazi do strukture koja je prikazana na slici Slika 3.



Slika 3. Struktura nakon dodavanja lažne zavisnosti

Takođe, moguće je otići i korak dalje i definisati interfejs za lažne implementacije. Ova struktura je ilustrovana na slici Slika 4.



Slika 4. Struktura nakon dodavanja interfejsa *IextensionManager*

U cilju razbijanja zavisnosti koda u odnosu na fajlsistem potrebno je izvršiti malo refaktorisanje koda. Moguće je definisati dva osnovna tipa refaktorisanja:

1. Abstrakcija konkretnih objekata u interfejse
2. Refaktorisanje u cilju ubacivanja lažnih implementacija (zavisnosti) datih interfejsa

Ubacivanje (eng. *injeciton*) lažne implementacije je moguće raditi na sledeće načine[2]:

1. Ubacivanje lažne implementacije (*stub-a*) u konstruktor
2. Ubacivanje lažne implementacije u vidu *property*-ja klase (*get* i *set*)

U nastavku će biti pojašnjeni ovi načini, ali prvo će biti pojašnjena ekstrakcija interfejsa kako bi se omogućila jednostavna zamena logike (uvođenje lažne implementacije).

[2] Postoje i drugi načini, ali će u okviru ovog kursa biti obrađeni samo ovi.

3. Ekstrakcija interfejsa radi zamene potrebne logike

Kao što je napomenuto prvi korak predstavlja izdvajanje koda koji je zavistan u poseban sloj. Klasa *`LogAnalyzer`* sadrži dati kod u sklopu metode *`IsValidLogFileName(string fileName)`* i ovaj kod se izdvaja u posebnu klasu *`FileExtensionManager`* koja se nalazi u folderu *Managers*. Listing 2 prikazuje poziv premeštenoj logici iz klase *`LogAnalyzer`*.

Listing 2. Poziv izmeštenoj logici iz stare metode

```
1  public class LogAnalyzer
2  {
3
4    public bool IsValidLogFileName(string fileName)
5    {
6       FileExtensionManager mngr = new FileExtensionManager();
7       return mngr.IsValid(fileName);
8    }
9
10 }
11
12 public class FileExtensionManager
13 {
14
15   public bool IsValid(string fileName)
16   {
17       //pristupanje fajlsistemu i čitanje konfiguracionog fajla
18       //vraćanje vrednosti true/false u zavisnosti od toga da li se
19       //u fajlu nalazi ekstenzija koja je prosleđena u punom imenu fajla
20       //u parametru fileName
21   }
22
23 }
```

Nakon ovoga moguće je reći klasi koja se testira (objektu klase *LogAnalyzer*) da će umesto konkretne klase *`FileExtensionManager`* koristiti neku vrstu *ExtensionManager*-a, bez toga da poznaje konkretnu implementaciju te menadžer klase. Ovo je moguće uvođenjem već pomenutog interfejsa (Slika 4) *`IExtensionManager`*. Listing 3 ilustruje uvođenje ovog interfejsa.

Listing 3. Uvođenje interfejsa IExtensionManager

```
1  public interface IExtensionManager
2  {
3      bool IsValid(string filename);
4  }
5
6
7
8  public class FileExtensionManager : IExtensionManager
9  {
10     public bool IsValid(string fileName)
11     {
12        //pristupanje fajlsistemu i čitanje konfiguracionog fajla
13        //vraćanje vrednosti true/false u zavisnosti od toga da li se
14        //u fajlu nalazi ekstenzija koja je prosleđena u punom imenu fajla
15        //u parametru fileName
16     }
17 }
18
19
20 public class LogAnalyzer
21 {
22
23   public bool IsValidLogFileName(string fileName)
24   {
25      IExtensionManager mngr = new FileExtensionManager();
26      return mngr.IsValid(fileName);
27   }
28
29 }
```

Nakon uvođenja interfejsa napravljen je prostor za jednostavno uvođenje lažne implementacije *FileExtensionManager* klase. Lažna implementacija biće uvedena u vidu nove klase *AlwaysValidFakeExtensionManager* koja implementira interfejs *IExtensionManager.* Listing 4 prikazuje uvođenje lažne implementacije u vidu nove klase. Data klasa je takođe kreirana u Managers folderu.

Listing 4. Uvođenje lažne implementacije koja zamenjuje eksternu zavisnost

```
1  public class AlwaysValidFakeExtensionManager : IExtensionManager
2  {
3      public bool IsValid(string filename)
4      {
5          return true;
6      }
7  }
```

Kao što je moguće videti iz listinga Listing 4 sama logika ove klase je vrlo jednostavna. Takođe, bitno je obratiti pažnju na naziv klase. U ovom slučaju klasa se zove *AlwaysValidFakeExtensionManager* gde je od posebnog značaja prvi deo *AlwaysValidFake.* Na osnovu ovoga naziva vrlo je jednostavno zaključiti da se radi o lažnoj implementaciji koja služi za simulaciju isključivo validnih slučajeva. Isto tako je moguće kreirati klasu za slučajeve koji nisu validni (*AlwaysInvalidFakeExtensionManager*).

Nakon dodavanje ove lažne implementacije potrebno je na neki način javiti metodi koja se testira koju implementaciju da koristi, konkretnu ili lažnu, tj. *AlwaysValidFakeExtensionManager* ili *FileExtensionManager*. Ovo je moguće uraditi na već pomenuta dva načina, odnosno ubacivanjem zavisnosti na nivou konstruktora ili u vidu *property*-ja klase koja se testira.

4. Ubacivanje zavisnosti na nivou konstruktora

Na ovaj način vrši modifikacija konstruktora (ili dodavanje novog) klase koja sadrži metodu koja se testira tako da on sada prihvata parametar koji predstavlja objekat klase koju je potrebno kotistiti prilikom testa. Listing 5 ilustruje postupak dodavanja ovog konstruktora i izmenu metode `IsValidLogFileName`.

Listing 5. Dodavanje konstruktora koji prihvata parametar tipa klase koju je potrebno koristiti za komunikaciju sa fajlsistemom

```
1   public class LogAnalyzer
2   {
3      private IExtensionManager manager;
4
5      public LogAnalyzer(IExtensionManager mngr)
6      {
7         manager = mngr;
8      }
9
10     public bool IsValidLogFileName(string fileName)
11     {
12        return manager.IsValid(fileName);
13     }
14
15  }
```

Nakon refaktorisanja konstruktora moguće je napisati testove koji će proslediti objekat odgovarajuće klase (lažna implementacija) i na taj način će biti izvršeno zaobilaženje eksterne zavisnosti. Testovi se nalaze u posebnom projektu (*StubExample.UnitTests*), a testove za klasu `LogAnalyzer` je moguće napisati u klasi `LogAnalyzerTests`.

Listing 6. Pisanje testova koji vrše ubacivanje lažne implementacije na nivou konstruktora

```
1  [TestFixture]
2  public class LogAnalyzerTests
3  {
4      [Test]
5      public void IsValidFileName_GoodExtension_ReturnsTrue()
6      {
7          AlwaysValidFakeExtensionManager fakeManager =
8                              new AlwaysValidFakeExtensionManager();
9
10         //slanje stub-a, odnosno ubacivanje lažne implementacije
11         LogAnalyzer log = new LogAnalyzer(fakeManager);
12
13         bool result = log.IsValidLogFileName("test.ots");
14         Assert.True(result);
15     }
16
17     [Test]
18     public void IsValidFileName_BadExtension_ReturnsFalse()
19     {
20         AlwaysInvalidFakeExtensionManager fakeManager =
21                              new AlwaysInvalidFakeExtensionManager ();
22
23         //slanje stub-a, odnosno ubacivanje lažne implementacije
24         LogAnalyzer log = new LogAnalyzer(fakeManager);
25
26         bool result = log.IsValidLogFileName("test.ost");
27         Assert.False(result);
28     }
29
30 }
```

Listing 6 ilustruje primer dva testa. Prvi test vrši testiranje validnog scenarija odnosno prosleđivanje naziva fajla metodi koja vrši proveru, gde se instancira objekat klase `AlwaysValidFakeExtensionManager` koji vrši simulaciju odgovora konkretne klase `FileExtensionManager` i to u slučaju da ona vraća *true*. Analogno ovom scenariju drugi test vrši testiranje scenaria u kome se vraća vrednost *false*.

Kod iz listinga Listing 6 je moguće modifikovati tako da se dodatno pojednostavi proces testiranja. Klase koje predstavljaju lažne implementacije klase `FileExtensionManager` je moguće izbaciti iz foldera Managers (gde su prvobitno definisane) i kreirati jednu internu klasu paralelno sa klasom koja vrši testiranje klase `LogAnalyzer`. U okviru te jedne klase moguće je definisati *boolean* polje koje će ukazivati na scenario koji se testira, odnosno validan ili ne. Listing 7 ilustruje ovu implementaciju.

Listing 7. Implementacija interne klase koja predstavlja lažnu implementaciju

```
1  [TestFixture]
2  public class LogAnalyzerTests
3  {
4      [Test]
5      public void IsValidFileName_GoodExtension_ReturnsTrue()
6      {
7          FakeExtensionManager fakeManager = new FakeExtensionManager();
8
9          fakeManager.Valid = true;
10
11         //slanje stub-a, odnosno ubacivanje lažne implementacije
12         LogAnalyzer log = new LogAnalyzer(fakeManager);
13
14         bool result = log.IsValidLogFileName("test.ots");
15         Assert.True(result);
16     }
17
18     [Test]
19     public void IsValidFileName_BadExtension_ReturnsFalse()
20     {
21         FakeExtensionManager fakeManager = new FakeExtensionManager ();
22
23         //slanje stub-a, odnosno ubacivanje lažne implementacije
24         LogAnalyzer log = new LogAnalyzer(fakeManager);
25
26         bool result = log.IsValidLogFileName("test.ost");
27         Assert.False(result);
28     }
29
30 }
31
32 internal class FakeExtensionManager : IExtensionManager
33 {
34     public bool Valid = false;
35
36     public bool IsValid(string filename)
37     {
38         return Valid;
39     }
40 }
```

Na osnovu listinga Listing 7 moguće je videti da prethodne klase koje su predstavljale lažne impl. `AlwaysValidFakeExtensionManager` i `AlwaysInvalidFakeExtensionManager` više nisu neophodne i da je sada moguće koristiti novu internu klasu `FakeExtensionManager` kako bi se izvršilo ubacivanje odgovarajuće lažne implementacije u konstruktor klase `LogAnalyzer`.

5. Ubacivanje zavisnosti putem property-ja (get/set)

Drugi način za ubacivanje lažne implementacije jeste putem property-ja klasa. Ovaj način je možda i jednostavniji od prethodnog i sastoji se od deklaracije promenljivih u okviru klase koja će biti pod testom i postavljanjem njihovih vrednosti na odgovarajuće, odnosno na objekte lažne implementacije. Listing 8 prikazuje postupak ubacivanja lažne implementacije putem property-ja klase.

Listing 8. Ubacivanje lažne implementacije putem property-ja i odgovarajući test

```
1  public class LogAnalyzer
2  {
3      private IExtensionManager manager;
4
5      public LogAnalyzer ()
6      {
7          manager = FileExtensionManager;
8      }
9
10     public IExtensionManager ExtensionManager
11     {
12         get { return manager; }
13         set { manager = value; }
14     }
15     public bool IsValidLogFileName(string fileName)
16     {
17         return manager.IsValid(fileName);
18     }
19 }
20
21
22 [TestFixture]
23 public class LogAnalyzerTests
24 {
25     [Test]
26     public void IsValidFileName_GoodExtension_ReturnsTrue()
27     {
28         FakeExtensionManager fakeManager = new FakeExtensionManager();
29
30         fakeManager.Valid = true;
31
32         //slanje stub-a, odnosno ubacivanje lažne implementacije
33         LogAnalyzer log = new LogAnalyzer();
34         log.ExtensionManager = fakeManager;
35
36         bool result = log.IsValidLogFileName("test.ots");
37         Assert.True(result);
38     }
39 }
```

```
40 internal class FakeExtensionManager : IExtensionManager
41 {
42     public bool Valid = false;
43 
44     public bool IsValid(string filename)
45     {
46         return Valid;
47     }
48 }
```

Iz listinga Listing 8 moguće je videti da je u okviru klase `LogAnalyzer` definisam *property* `manager` koji je tipa `IExtensionManager` (linija 3 Listing 8) i moguće ga je podesiti iz neke eksterne klase. Konstruktor date klase prvobitno inicijalizuje dati *property* na sa instancom klase `FileExtensionManager` (linija 7 Listing 8), odnosno sa instancom konkretne klase koja sadrži pravu implementaciju logike koja komunicira sa fajlsistemom (Listing 2). Ovo je moguće zameniti iz testa tako što se pristupa instanci klase `LogAnalyzer` i podešava dati property na objekat koji predstavlja lažnu implementaciju (linija 34 Listing 8).

6. Simulacija izuzetaka (*exceptions*)

Objekat od kog je neki deo sistema zavistan ne mora uvek da vrati neki konkretan odgovor. Jedan takođe realan scenario jeste da je u toku izvršavanja koda, od kog zavisi neki deo sistema pod testom, došlo do nekog izuzetka. Upotrebom *stub*-ova moguće je pokriti i ovakav scenario.

```
1  [TestFixture]
2  public class LogAnalyzerTests
3  {
4      [Test]
5      public void IsValidFileName_BadExtension_ExceptionOccurres()
6      {
7          FakeExtensionManager fakeManager = new FakeExtensionManager();
8 
9          fakeManager.ExceptionWillOccur =
10                             new Exception("An exception has occurred");
11 
12         //proverava da li je nastao izuzetak
13         var ex = Assert.Throws<Exception>(() =>
14                                     myFakeManager.IsValid("valid.ots"));
15         //proverava poruku u izuzetku
16         Assert.That(ex.Message, Is.EqualTo("An exception has occurred");
17 
18     }
19 
20 }
21 
22 
23 
24 
```

```
25 internal class FakeExtensionManager : IExtensionManager
26 {
27     public bool Valid = false;
28     public Exception ExceptionWillOccur = null;
29
30     public bool IsValid(string filename)
31     {
32         if(ExceptionWillOccur != null)
33         {
34             throw ExceptionWillOccur;
35         }
36         return Valid;
37     }
38 }
```