Mocks – testiranje interakcija objekata

1. Uvod

U prethodnom delu smo videli kako se može vršiti testiranje nekog dela koda koji zavisi od nekog drugog objekta kako bi se pravilno izvršio. Ovo je bilo moguće testirati primenom *stub*-ova. *Stub*-ovi su nam služili da kodu koji se testira podmetnemo sve inpute koji su mu potrebni da bi se pravilno izvršio, a koje bi inače dobio od objekata od kojih je zavistan.

Svi testovi napisani do ovog momenata su bili testovi koji su se bavili testiranjem neke povratne vrednosti ili nekog novog stanja sistema koji se testira (eng. *value-based/state-based testing*). U cilju objašnjenja *mock*-ova potrebno je pojasniti treći vid testiranja, odnosno testiranje interakcija (eng. *interaction testing*). Ovo testiranje se zasniva na testiranju poziva jednog objekta ka drugima. Objekat koji se testira možda ne vraća nikakvu vrednost, niti vrši neko čuvanje novog stanja ali sadrži logiku koja vrši pozive ka nekim objektima nad kojima nemamo kontrolu. Drugim rečima potrebno je testirati da li se određeni objekti pozivaju onako kako bi trebalo iz sistema koji se testira (objekta koji se testira). Principi testiranja koji su primenjivani do sada (*value/state-based*) ovde neće biti primenjivi pošto je potrebno proveriti da li se nešto promenilo u objektu koji se poziva. Interakciono testiranje se primenjuje kada je rezultat nekog posla dela sistema koji se testira poziv ka nekom objektu. Interakciono testiranje unosi dodatne komplikacije za razliku od *value* ili *state-based* testiranja i poželjnije je njihovo korišćenje umesto interakcionog testiranja, ukoliko je to moguće. Naravno, u nekim slučajevima ovo nije moguće i potrebno je testirati neku interakciju. Kako bi testirali da li neki objekat ima korektnu interakciju sa drugim objektom moguće je koristiti *mock* objekte.

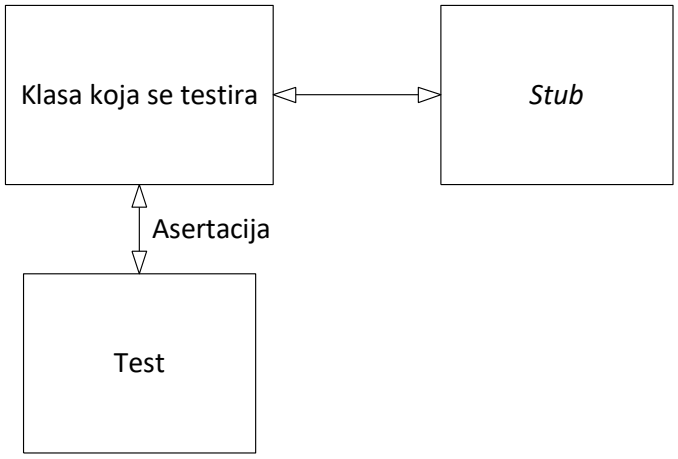
Mock objekat predstavlja lažni objekat u sistemu (lažnu implementaciju) koji "odlučuje" o tome da li je neki *unit* test prošao ili ne. Ovaj objekat ovo radi tako što verifikuje da li je objekat koji se testira izvršio očekivani poziv. Obično se piše jedan *mock* po testu kako se ne bi uvodila dodatna kompleksnost u test.

Sada kad je uveden termin *mock* potrebno je pojasniti koncept lažnih implementacija.

Lažna implementacija predstavlja generički termin koji može da se odnosi ili na *mock* ili na *stub* objekte. Da li je lažna implementacija *mock* ili *stub* zavisi od toga kako se koristi u testu. Ako se koristi u cilju provere neke interakcije (asertacija se vrši na lažnoj implementaciji) u pitanju je *mock* objekat, u suprotnom radi se o *stub*-u.

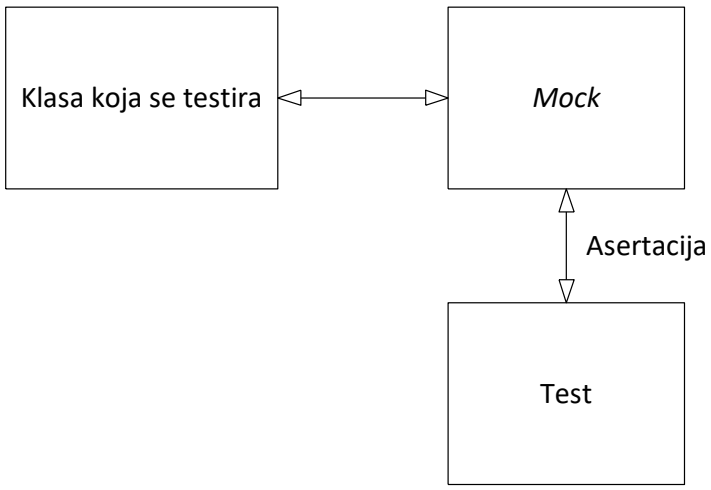
2. *Mocks VS Stubs*

Stubs vrše zamenu nekog objekta kako bi se moglo vršiti testiranje nekog drugog objekta bez smetnji. Slika 1 ilustruje način na koji koristimo jedan *stub* prilikom testiranja. Sa slike je moguće primetiti da se asertacija vrši nad klasom koja se testira, dok *stub* omogućava da se test uspešno izvrši time što eliminiše neku eksternu zavisnost time što lažira neki odgovor od te zavisnosti.



Slika 1. Upotreba stub-*a prilikom testiranja nekog dela sistema*

Na prvi pogled *mock* i *stub* objekti deluju veoma slično, ali postoji jedna osnovna razlika. *Stubs* ne mogu da dovedu do toga da neki test ne prođe. Asertacija koja vrši testiranje se izvršava nad nekom klasom koja koristi neki *stub*, a za koji je uvek poznato šta vraća odnosno kako se ponaša. Drugim rečima *stub* predstavlja zavisnost koju je u svakom momentu moguće kontrolisati. Sa druge strane *mock* objekti mogu da prouzrokuju stanje koje dovodi do toga da neki test ne prođe. Ovo je moguće pošto testovi koji koriste *mock* objekte asertacije vrše nad njima kako bi proverili neko stanje koje ne mora da bude onakvo kakvo se očekuje u testu. Prilikom testiranja uz pomoć *mock* objekata komunikacija koja se testira se memoriše u samom *mock* objektu.



Slika 2. Upotreba mock-a prilikom testiranja neke interakcije u sistemu

3. Primer

U cilju pojašnjenja upotrebe *mock* objekata u nastavku je korak po korak objašnjen primer njihovog kreiranja i upotrebe kroz nadogradnju primera koji je dat u prethodnoj skripti *Stubs – razbijanje eksternih zavisnosti u kodu i testiranje* (*Stubs.pdf*)[1]. S obzirom da primer koji je dat u nastavku predstavlja nadogradnju primera iz pomenute skripte potrebno je napomenuti da postupak identifikacije zavisnosti i njihove izolacije u poseban slog takođe važi i za primenu *mock* objekata. Drugim rečima, ukoliko se koriste samo *mock* objekti opet je potrebno ponoviti taj postupak izolacije zavisnosti u kodu u okviru posebnog sloja. Kod za ovaj primer je moguće naći na Git serveru u okviru repozitorijuma *Mocks*[2].

Na osnovu do sada rečenog moguće je dodati da je kreiranje *mock* objekata veoma slično kreiranju *stub*-ova, samo što će kreirani *mock* objekti obavljati nešto više posla. Za razliku od *stub*-ova oni će sačuvati istoriju komunikacije koja će kasnije biti verifikovana kao očekivana komunikacija od strane testa. Dakle, *mock* je lažna implementacija nekog objekta koji se poziva od strane nekog objekta koji se testira, ta komunikacija (poziv) se u nekoj formi pamti u samom *mock*-u i testira iz nekog *unit* testa.

Radi kreiranja potrebe za *mock* objektom na dosadašnji LogAnalyzer će biti dodat jedan novi zahtev. Potrebno je uvesti interakciju sa eksternim veb servisom kojem će biti poslata poruka o grešci svaki put kada LogAnalyzer naiđe na naziv fajla koji je previše kratak (npr. < 10 karaktera). Zamislićemo da servis koji treba da bude iskorišćen još uvek nije u potpunosti implementiran, ali svakako naš kod mora da bude pokriven adekvatnim testovima koji bi testirali slučaj da se vrši komunikacija sa datim veb servisom. U ovu svrhu je potrebno izvršiti određene modifikacije u kodu.

Na samom početku potrebno je dodati novi interfejs (`IWebService.cs`) koji će kasnije koristiti za kreiranje *mock* objekata. Ovaj interfejs će posedovati metode koje je potrebno zvati u okviru budućeg veb servisa. Listing 1 ilustruje ovaj interfejs.

Listing 1. Dodavanje novog interfejsa *IWebService*

```
1  public interface IWebService
2  {
3     void LogError(string message);
4  }
```

Ovaj interfejs je moguće iskoristiti i za kreiranje *stub*-ova (kao što je pokazano u prethodnoj skripti).

Nakon ovoga moguće je kreirati i konkretan *mock* objekat. U tu svrhu potrebno je kreirati posebnu klasu koja će predstavljati lažnu implementaciju veb servisa. Na prvi pogled ova klasa će jako ličiti na *stub*-ove ali sa jednom bitnom razlikom – mogućnost da pamti neko stanje. Listing 2

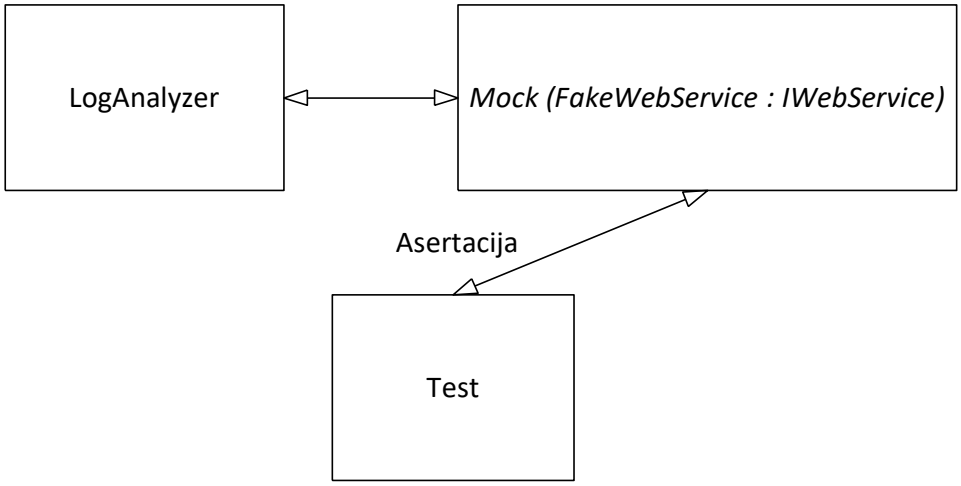
[1] Adresa: http://ellab.ftn.uns.ac.rs/ - kurs Osnove testiranja softvera, sekcija Vežbe
[2] Adresa: http://147.91.175.237:8082/Bonobo.Git.Server

prikazuje novu klasu koja predstavlja lažnu implementaciju veb servisa. Data klasa je kreirana u *Managers* folderu projekta *Mocks*.

Listing 2. Dodavanje lažne implementacije veb servisa

```
1  public class FakeWebService : IWebService
2  {
3     public string LastError;
4
5     public void LogError(string message)
6     {
7         LastError = message;
8     }
9  }
```

Upotrebu ove nove lažne (*mock*) implementacije od strane *LogAnalyzer* klase je moguće i grafiči ilustrovati. Slika 3 upravo ovo i ilustruje. Test će napraviti instancu klase *FakeWebService* koja će pamtiti poruke koje su joj poslate od strane *LogAnalzyer*-a. Ove poruke će nakon toga u testu biti proverene uz pomoć određenih asertacija. Dakle asertacije će se vršiti nad *mock* objektom, odnosno instancom klase *FakeWebService*.



Slika 3. Prikaz upotrebe lažne implementacije od strane klase koja se testira

Naravno, *LogAnalyzer* klasa mora da sadrži metodu koja će vršiti date pozive i ta metoda je *`Analyze(string fileName)`*. Listing 3 ilustruje kreiranje *mock* objekta i testiranje interakcije klase *LogAnalyzer* sa njime.

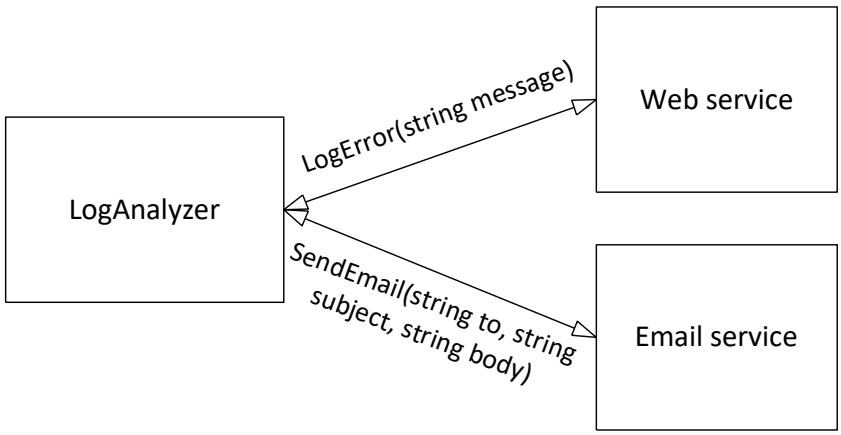
Listing 3. Kreiranje *mock* objekta i testiranje interakcije sa lažnim servisom

```
1  public class LogAnalyzer
2  {
3     private IWebService service;
4
5     public LogAnalyzer(IWebService webService)
6     {
7        service = webService;
8     }
9
10    public void Analyze(string filename)
11    {
12       if(filename.Length < 10)
13       {
14          service.LogError("Filename is too short: " + fileName);
15       }
16    }
17 }
18
19 [TestFixture]
20 public class LogAnalyzerTests
21 {
22     [Test]
23     public void Analyze_TooShortFileName_CallWebService()
24     {
25         FakeWebService mockService = new FakeWebService();
26         LogAnalyzer log = new LogAnalyzer(mockService);
27
28         string tooShortFileName = "test.abc";
29         log.Analyze(tooShortFileName);
30
31         StringAssert.Contains("Filename   is   too   short:   test.abc",
32                               mockService.LastError);
33     }
34 }
```

Na osnovu listinga Listing 3 moguće je videti postupak kreiranja i upotrebe *mock* objekta u okviru jednog *unit* testa. U okviru testa vrši se instanciranje *mock* objekta (Listing 3 linija 25) i prosleđivanje tog objekta klasi koja se testira na neki od načina koji su definisani u prethodnoj skripti koja je obrađivala *stub*-ove. U ovom slučaju koristi se ubacivanje lažne implementacije na nivou konstruktora (Listing 3 linija 26). Linije 28 i 29 vrše kreiranje testnog naziva fajla i pozivanje metode koja se testira, a koja kao rezultat izvršavanja ima poziv servisu ili u ovom slučaju *mock* objektu. S obzirom da *mock* objekat ima sposobnost pamćenja stanja (Listing 2 linija 3) u okviru testa je potrebno proveriti da li je *mock* objekat uspešno pozvan, odnosno da li sadrži poruku koja mu je poslata. Linija 31 u listingu Listing 3 radi upravo ovo tako što vrši asertaciju nad *mock* objektom i ispituje njegovo stanje, tj da li polje `LastError` sadrži poruku koju je slao *LogAnalyzer.*

4. Paralelna upotreba *mock*-ova i *stub*-ova

Usled većeg broja zavisnosti u kodu od nekih eksternih objekata čest scenario koji može da nastane jeste da je u okviru jednog testa potrebno iskoristiti i *mock*-ove i *stub*-ove kako bi se kompletno eliminisala zavisnost nad kojom ne postoji kontrola i uvela ona koju je moguće kontrolisati. U cilju pojašnjenja ovog scenaria prethodni primer će biti malo proširen. Zamislimo da sada *LogAnalyzer* ne samo da treba da kontaktira veb servis u slučaju da je ime nekog fajla previše kratko, već da treba da pošalje mejl obaveštenja na određenu adresu ukoliko nastane neka greška na veb servisu. Slika 4 ilustruje uvođenje nove zavisnosti za klasu *LogAnalyzer*.

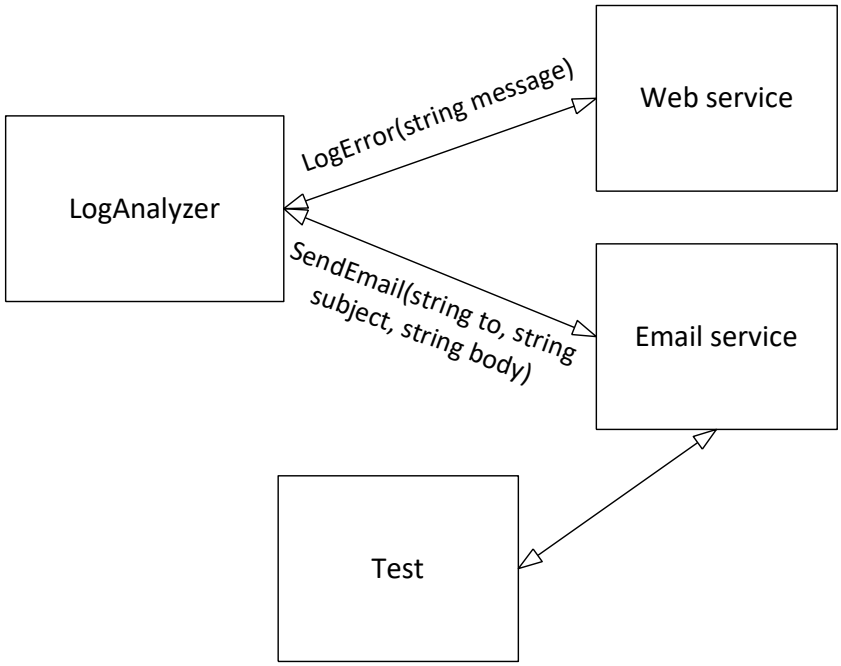


Slika 4. Uvođenje nove zavisnosti za *LogAnalyzer – Email service*

Problematika testiranja koja se postavlja u ovom scenariju jeste kako testirati poziv email servisu iz klase *LogAnalyzer* ukoliko se desi neki izuzetak u veb servisu koji je prvobitno pozvan. Zadatke koje je potrebno obaviti u cilju testiranja ovog scenaria je moguće podeliti na 3 grupe:

- Zamena veb servisa (uvođenje lažne implementacije).
- Simulacija izuzetka u veb servisu.
- Provera da li je email servis pravilno ili uopšte pozvan ukoliko je došlo do simuliranog izuzetka.

Prva dva zadatka se mogu obaviti upotrebom *stub*-ova. Na ovaj način se kreira lažna implementacija koja simulira izuzetak. Ova implementacija se prosleđuje *LogAnalyzer*-u putem konstruktora ili nekog polja uz pomoć *set* metode. Za poslednji zadatak moguće je upotrebiti jedan *mock* objekat koji će da predstavlja email servis i da ima mogućnost pamćenja stanja kako bi bilo moguće proveriti da li je on pravilno pozvan. Na osnovu rečenog moguće je zaključiti da će budući test vršiti asertaciju nad instancom lažne implementacije, odnosno *mock*-om, email servisa (Slika 5).



Slika 5. Test vrši asertaciju nad *mock* objektom

U okviru ovako napisanog testa koristiće se dve lažne implementacije i to jedan *stub* (lažna implementacija veb servisa) i jedan *mock* objekat (lažna implementacija email servisa). Veb servis predstavlja *stub* pošto on neće biti korišćen u cilju verifikacije rezultata testa već samo u cilju da se test izvrši pravilno. Sa druge strane, email servis predstavlja *mock* objekat pošto se asertacija vrši nad njime u cilju provere da li je pravilno pozvan od strane sistema pod testom (metoda *Analyze* klase *LogAnalyzer*).

Listing 4. Paralelna upotreba *stub*-a i *mock*-a

```
1  public interface IEmailService
2  {
3     void SendEmail(string to, string subject, string body);
4  }
5
6  public class LogAnalyzer2
7  {
8      private IWebService service;
9      private IEmailService email;
10
11     public LogAnalyzer2(IWebService webService,
12                         IEmailService emailService)
13     {
14         service = webService;
15         email = emailService;
16     }
17
18
```

```
19
20      public IWebService Service
21      {
22          get;
23          set;
24      }
25
26      public IEmailService Email
27      {
28          get;
29          set;
30      }
31
32      public void Analyze(string fileName)
33      {
34          if (fileName.Length < 10)
35          {
36              try
37              {
38                  service.LogError("Filename is too short: " + fileName);
39              }
40              catch (Exception ex)
41              {
42                  email.SendEmail("test@test.com",
43                                  "Can't log test", ex.Message);
44              }
45          }
46      }
47  }
48
49  public class FakeEmailService : IEmailService
50  {
51      public string To { get; set; }
52      public string Subject { get; set; }
53      public string Body { get; set; }
54
55      public void SendEmail(string to, string subject, string body)
56      {
57          To = to;
58          Subject = subject;
59          Body = body;
60      }
61  }
62
63
64
65
66
67
```

```
68 public class FakeWebServiceException : IWebService
69 {
70     public Exception ToThrow;
71 
72     public void LogError(string message)
73     {
74         if (ToThrow != null)
75         {
76             throw ToThrow;
77         }
78     }
79 }
80 
81 [TestFixture]
82 public class LogAnalyzer2Tests
83 {
84     [Test]
85     public void Analyze_WebServiceEncountersException_SendEmail()
86     {
87         FakeWebServiceException stubService =
88                                             new FakeWebServiceException();
89         stubService.ToThrow = new Exception("Some fake exception");
90         FakeEmailService mockEmail = new FakeEmailService();
91         LogAnalyzer2 log = new LogAnalyzer2(stubService, mockEmail);
92         string tooShortFileName = "test.abc";
93         log.Analyze(tooShortFileName);
94 
95         StringAssert.Contains("test@test.com", mockEmail.To);
96         StringAssert.Contains("Can't log test", mockEmail.Subject);
97         StringAssert.Contains("Some fake exception", mockEmail.Body);
98     }
99 }
```

Listing 4 prikazuje način kako je moguće kombinovati *stub* i *mock* u okviru jednog testa kako bi ranije navedena problematika bila rešena. Najpre je potrebno kreirati novi interfejs `IEmailService` koji će biti implementiran od strane klase *FakeEmailService* (Listing 4 linija 49). Za potrebe ovog primera kreirana je klasa *LogAnalyzer2* kako bi kod bio pregledniji (Listing 4 linija 6). Data klasa poseduje metodu *Analyze* koja u slučaju greške na veb servisu šalje mejl obaveštenja na testnu adresu. Klasa takođe ima definisan konstruktor koji omogućava ubacivanje lažnih implementacija veb i email servisa (Listing 4 linija 11). S obzirom da postoji nova klasa kreirana je i nova testna klasa *LogAnalyzer2Tests* koja sadrži testove koji je testiraju (Listing 4 linija 82). Ova klasa sadrži test koji testira slanje mejla obaveštenja u slučaju nastanka izuzetka na veb servisu. Na početku testa je izvršeno instanciranje svih neophodnih lažnih implementacija (Listing 4 linija 77 i 89). S obzirom da veb servis predstavlja *stub* u testu njegovo polje *ToThrow* inicijalizujemo sa izuzetkom (Listing 4 linija 89). Na osnovu novih potreba za testiranjem izuzetaka koji mogu nastati u veb servisu kreirana je nova klasa *FakeWebServiceException* ((Listing 4 linija 68). Ovo će omogućiti simulaciju nastanka izuzetka na veb servisu i njegovo prosleđivanje

LogAnalyzer-u. Ukoliko je izuzetak detektovan vrši se slanje mejla tako što se šalje odgovarajuća poruka *mock* objektu koji je takođe podmetnut klasi *LogAnalyzer*2 (Listing 4 linija 91). Na samom kraju testa vrši se asertacija nad *mock* objektom gde se proverava da li je on uspešno pozvan na osnovu provere njegovih polja koja omogućavaju čuvanje stanja.

5. Saveti za rad sa *mock*-ovima i *stub*-ovima

Dobra praksa jeste da se svaki napisan *unit* test bavi testiranjem samo jedne stvari, odnosno jednog scenaria. Ovo eliminiše nepotrebnu komplekstnost i značajno povećava čitljivost testnog koda. Ukoliko je ovo slučaj ne bi smela da se javi potreba za postojanjem više od jednog *mock* objekta po testu. Postojanje više *mock* objekata može biti indikator da se vrši testiranje više stvari u jednom testu s obzirom da se posmatra interakcija sa više objekata. Ukoliko postoji ovaka potreba bolji scenario jeste razbijanje na više testova.

Prilikom pisanja testova potrebno je pre svega identifikovati šta je krajnji rezultat dela sistema (metode) koji testiramo. Krajnji rezultat može biti neka povratna vrednost, promena stanja objekta ili upućivanje poziva ka nekom drugom objektu. Na osnovu ovoga lakše je odlučiti da li je neophodna upotreba *mock*-ova, da li se posao može obaviti samo uz pomoć *stub*-ova ili je potrebno kombinovati *stub*-ove i *mock*-ove.

6. Isolation (mocking) frameworks

Do sada su *stub*-ovi i *mock*-ovi pisani „peške“ odnosno svaka zavisnost u kodu rešavala se putem pisanja klase koja će predstavljati neku lažnu implementaciju. Ovaj postupak je moguće automatizovati putem tkz. izolacionih *framework*-a ili *framework*-a za mokovanje (eng. *Isolation/mocking frameworks*). Uz pomoć ovih *framework*-a moguće je relativno jednostvno kreirati lažne implementacije i podmetnuti ih nekom objektu za upotrebu. Ovako kreirani objekti u trenutku izvršavanja (eng. *runtime*) se još zovu i dinamički objekti.

Dakle, *mocking (isolation) framework* predstavljaju niz API-ova koji znatno pojednostavljuju postupak kreiranja i rada sa lažnim implementacijama. Oni eliminišu repetitivno pisanje koda koji predstavlja neku lažnu implementaciju. Takođe, kod postaje manje podložan promenama jer se eliminiše potreba za konstantnim promenama lažnih implementacija usled nekih drugih promena u kodu.

Postoji veliki broj ovih *framework*-ova, a jedan od popularnijih koji će i ovde biti prikazan jeste *NSubstitute framework*[3]. U nastavku će biti pojašnjena instalacija i upotreba ovog *framework*-a na primeru upućivanja poziva ka veb servisu koji je pominjan ranije (Listing 3).

U cilju ilustracije primera rada sa ovim *framework*-om potrebno je kreirati novi projekat *MocksNSub.UnitTests* koji će biti tipa *Class Library (.Net Framework).* Nakon kreiranja ovog projekta potrebno je instalirati *NUnit* i *NUnit3TestAdapter* pakete pomoću *NuGet* paket menadžera. Pošto su ovi paketi uspešno instalirani potrebno je instalirati i *NSubstitute*

[3] *NSubstitute* dokumentacija: http://nsubstitute.github.io/help/getting-started/

framework. Ovaj *framework* je moguće instalirati pomoću *NuGet* paket menadžera isto kao i prethodne pakete (desni klik na projekat > *NuGet package manager* > *Browse* > *Search "NSubstitute"* > *Install*). *NSubstitute* poseduje klasu *Substitute* koja će se koristiti u cilju generisanja lažnih implementacija u trenutku izvršavanja. Ova klasa ima jednu metodu koja ima svoju generičku i ne generičku varijantu i koristi se za kreiranje lažnih implementacija. Ova metoda se poziva sa onim tipom lažne implementacije koji je potrebno kreirati. Kada se ova metoda pozove sa određenim tipom ona vraća objekat koji se pridržava nekog definisanog interfejsa. Drugim rečima, nema potrebe za kodiranjem tog objekta kao što je to bio slučaj do sada. Nakon kreiranja ova implementacija će se podmetati sistemu pod testom na neki od pominjanih načina (nivo konstruktora ili putem atributa). Listing 5 ilustruje primer pisanja testa za ispitivanje interakcije *LogAnalyzeri*-a sa veb servisom.

Listing 5. Pisanje testa za ispitivanje interakcije sa veb servisom uz pomoć *NSubstitute framework*-a

```
1  [TestFixture]
2  public class LogAnalyzerNSubTests
3  {
4    [Test]
5    public void Analyze_TooShortFileName_CallWebService()
6    {
7       IWebService webService = Substitute.For<IWebService>();
8       LogAnalyzer log = new LogAnalyzer(webService);
9       string tooShortFileName = "test.abc";
10      log.Analyze(tooShortFileName);
11      webService.Received().LogError("Filename is too short: test.abc");
12   }
13 }
```

Na osnovu listinga Listing 5 moguće je primetiti da se simulacija interakcije vrši uz pomoć *mock* objekta koji je kreiran u pozivu tipiziranoj metodi *For* klase *Substitute* (Listing 5 linija 7). Nakon ovoga vrši se standardno podmetanje lažne implementacije sistemu pod testom, kreiranje testnog stringa i pozivanje metode koja se testira (Listing 5 linije 8-10). U okviru linije 11 vrši se provera da li je poruka prihvaćena od strane *mock* objekta, odnosno da li je komunikacija uspešna. Na osnovu ovoga moguće je zaključiti da više nema potrebe za lažnom implementacijom u vidu klasa koje su kreirane u *Managers* folderu projekta (Listing 2).