Univerzitet u Beogradu

Matematički fakultet

Neke metode za automatizovano testiranje softvera

Master rad

Kruna Matijević

Beograd 2013.

Matematički fakultet

Univerzitet u Beogradu

Master rad

Autor: Kruna Matijević

Naslov: Neke metode za automatizovano testiranje softvera

Mentor: Profesor Dr. Dušan Tošić, Matematički fakultet

Članovi komisije: Docent Dr. Filip Marić, Matematički fakultet

Asistent Mr. Milena Vujošević-Janičić, Matematički fakultet

SADRŽAJ

1 Uvod

Testiranje je važan deo procesa razvoja softvera. Neke procene govore da se za testiranje i debagovanje utroši 50% ukupnog vremena [1].

Da bismo bili sigurni da softver radi ono što treba, i da to radi dobro, testiranje je neophodno. Kao zamena za testiranje nekad se nude metode formalnog dokazivanja. Iako one garantuju ispravnost, moguće ih je sprovesti samo na sisteme ograničene veličine i njihova potpuna automatizacija, za sada, nije moguća. Testiranje sistema se može automatizovati i ono je, sa strane isplativosti, jeftinije. S druge strane ono ne može da garantuje ispravnost, samo da poveća stepen pouzdanosti sistema.

Softverski sistemi postaju sve važniji za sve vrste organizacija, ali i za individue. Samim tim raste i značaj njihovog kvaliteta. Ozbiljnost grešaka, koje se javljaju u softveru, može biti bilo gde na skali od malih vizualnih problema do pada sistema. Neke od njih mogu da dovedu do opasnih situacija za ljude. Ako za primer uzmemo sisteme koji se koriste pri lansiranju raketa u svemir, jasno je da i male greške u sistemu vode do ozbiljnih posledica. Iz svega ovoga proizilazi i važnost testiranja softvera. Na žalost, postoji mnogo problema u vezi sa testiranjem. Ono je komplikovano i oduzima dosta vremena. Uopšteno, komplikovane zadatke obavljaju ljudi koji su skloni greškama. Ovo se može popraviti automatizacijom. Da bi se uvela automatizacija testiranja, često je potrebno izmeniti sâm process razvoja softvera. Takođe, nemoguće je automatizovati sve moguće testove, pa je i donošenje odluke o tome šta treba automatizovati teška. Lista problema koji su vezani za automatizaciju je dugačka. Dosta istraživanja se bavi ovim problemima i, kao i uvek, postoji neka vrsta razdora između onoga što je teoretski dobro i onoga što se primenjuje u praksi.

Automatizacija se može i treba primeniti na različite načine i u različitim kontekstima i, još uvek, ne postoji jedna strategija koja će uvek dati podjednako dobre rezultate.

2 O testiranju softvera

Uprkos velikom broju veoma korisnih alata, okruženja i biblioteka, razvoj softvera je i dalje u najvećoj meri manuelni proces. Samim tim, greške se javljaju. Testiranje softvera predstavlja proces interakcije sa sistemom ili delom softvera, koji za cilj ima otkrivanja grešaka. Vrsta sistema i način njegove implementacije u velikoj meri utiču na to kako će da se sprovede testiranje. Na primer, grafički korisnički interfejs zahteva drugačije načine testiranja od programske biblioteke ili konzolnog softvera.

Proces testiranja softvera uključuje validaciju i verifikaciju. Validacija služi za utvrđivanja da li se pravi softver u skladu sa specifikacijom koja je utvrđena na početku procesa izrade softvera, kao i da li ta specifikacija odgovara korisničkim zahtevima. Najveći deo validacije se obavlja pri analizi korisnički zahteva i pri dizajniranju sistema. Najčešće je cilj validacije da odgovori na pitanje da li

nam je određeni deo sistema potreban. Verifikacija opisuje proces utvrđivanja da li se softver pravi ispravno, npr. da li je ispravan u odnosu na svoj dizajn. Obavlja se pri razvoju sistema. Njen cilj je da pokaže da svi delovi sistema, pojedinačno i kao celina, rade ispravno. Uzmimo za primer razvoj mobilne aplikacije koja, prema korisničkim zahtevima, treba da ima funkciju kalendara u koji možemo da unosimo zakazane obaveze. U procesu validacije treba da ustanovimo šta ova aplikacija treba da sadrži, na primer mogućnost unosa nove obaveze, dnevni, nedeljni i mesečni pregled obaveza, notifikaciju koja obaveštava o predstojećoj obavezi. Istovremeno, validacija treba da nas spreči da u istoj aplikaciji napravimo i budilnik, jer to proizilazi izvan zahteva. Nakon implementacije aplikacije, u procesu verifikacije, pokušaćemo da, između ostalog, kreiramo jednu obavezu u aplikaciji i da je vidimo na pregledu kalendara.

Kako testiranje skoro nikada ne može da bude kompletno, ono ne može da pokaže nepostojanje grešaka. Odatle se nameće zaključak da je cilj testiranja detektovanje što većeg broja grešaka.

Neke greške su kompleksnije od drugih i njihov efekat može biti ozbiljniji. Testiranje obično ne mora da se fokusira na neki određenu vrstu grešaka zbog postojanja takozvanog grupisanja defekata. DeMillo kaže da su kompleksne greške povezane sa jednostavnijim [2]. Posledično, dovoljno je testirati sa cijem otkrivanja jednostavnih grešaka da bi se otkrile kompleksne greške. Kao što postoji više različitih tipova grešaka, postoji i više različitih tipova testiranja. Neke od klasifikacija testiranja su opisane u narednim poglavljima, ali to svakako nije kompletna kategorizacija.

2.1 Dinamičko i statičko testiranje

Na visokom nivou, testiranje softvera se može podeliti na dinamičko i statičko. Ova podela se vrši na osnovu toga da li se u toku testiranja kôd izvršava ili ne. Statičko testiranje je testiranje bez izvršavanja koda. Ono se obično vrši kroz razne vrste pregledanja i revizija. Predmet tih pregleda i revizija može biti dokumentacija ili kôd. Druga vrsta statičkog testiranja je statička analiza koda, kao što je provera sintaksne ispravnosti koda ili analiza složenosti koda. Sa ovakvim testiranjem greške se mogu uočiti u ranim fazama razvoja softvera.

Dinamičko testiranje podrazumeva izvršavanje samog koda, odnosno izvršavanje delova ili celog sistema. Osnovna podela dinamičkog testiranja je podela na funkcionalno i nefunkcionalno testiranje [3].

2.2 Funkcionalno i nefunkcionalno testiranje

Cilj funkcionalnog testiranja je da se ustanovi da softver odgovara svojim zahtevima. Funkcionalno testiranje se fokusira na unošenje određenih ulaznih parametara u sistem i verifikaciji izlaznih podataka i stanja. Koncept funkcionalnog testiranja je prilično sličan za sve sisteme, iako se ulazi i izlazi razlikuju od sistema do sistema.

Nefunkcionalno testiranje znači testiranje kvalitativnih aspekata softvera. Primeri nefunkcionalnog testiranja su:

- Testiranje performansi, koje ima za cilj da ustanovi da li softver radi ispravno u odnosu na zahteve vezane za brzinu i vreme izvršavanja određenih zahteva.
- Stres testiranje, koje evaluira ponašanje sistema pod velikim opterećenjem.
- Test pouzdanosti, koji ispituje da li se sistem ponaša korektno kroz duži vremenski period.
- Testiranje sigurnosti, koje se fokusira na bezbednost implementiranih procesa i podataka.

2.3 Metodologije crne i bele kutije

Pri testiranju, sistem za testiranje (*eng. System Under Test, SUT*) može da se posmatra kao crna kutija ili kao bela kutija. Kad se koristi strategija bele kutije, unutrašnja struktura sistema za testiranje je poznata. Cilj testiranja sa ovakvim pristupom je verifikacija korektnog ponašanja unutrašnjih strukturnih elemenata. Ovo se može postići, na primer, izvršavanjem različitih grana kôda. Strategija bele kutije je testiranje koje je veoma vremenski zahtevno i obično se primenjuje na manje delove sistema. Naročito je korisno za nalaženje grešaka u dizajnu, logičke greške i greške u toku podataka [3].

Kod testiranja zasnovanog na strategiji crne kutije, sistem se posmatra kao crna kutija i ne zna se ništa o unutrašnjoj strukturi sistema ili kôdu. Jedino znanje koje je potrebno je kako sistem funkcioniše. Sprovodi se tako što se u sistem uvode određeni ulazni podaci i onda se proverava da li je izlaz u skladu sa specifikacijom. Često se zove i funkcionalno testiranje jer uzima u obzir samo ponašanje, tj. funkcionalnost softvera.

2.4 Nivoi testiranja

Testiranje može da se sprovodi na različitim nivoima. Obično se testiranje deli na jedinično testiranje, integraciono testiranje, sistemsko testiranje i testiranje prihvatljivosti. Ove podele predstavljene su detaljnije u [4], [5] i [3]. Cilj ovih različitih tipova testiranja je da se sistem sagleda iz različitih perspektiva i da se pronađu različiti tipovi grešaka. Ako se ovi nivoi sagledavaju iz perspektive automatizacije, nivoi mogu da budu jedinično testiranje, testiranje komponenti i sistemsko testiranje.

2.4.1 Jedinično testiranje

Najmanji deo softvera je jedinica. Praktično posmatrano, to je funkcija ili procedura u programskom jeziku. Kod objektno orjentisanih sistema metode i klase se posmatraju kao jedinice. Osnovni cilj jediničnog testiranja je da otkrije funkcionalne i strukturne defekte u odgovarajućoj jedinici. U praksi, najčešće programer koji je napisao jedinicu sistema, piše i odgovarajuće jedinične testove. Greške otkrivene ovim testovima su obično jednostavne za lociranje i ispravljanje, imajući u vidu da se testira jedna jedinica. Samim tim, najjeftinije je pronalaženje i ispravljanje grešaka na nivou jediničnih testova.

2.4.2 Integraciono testiranje

Kada se osnovne jedinice sistema grupišu, dobijaju se klasteri ili podsistemi. Cilj integracionog testiranja je da verifikuje da interfejsi komponenata (klasa) ispravno rade. Dodatno, integracioni testovi verifikuju i korektnost toka operacija i toka podataka između komponenti.

2.4.3 Sistemsko testiranje

Kad se spremni i testirani podsistemi sjedine u finalni sistem, moguće je izvršavati sistemske testove. Oni uključuju i funkcionalno ponašanje sistema i nefunkcionalne osobine sistema. Cilj je utvrditi da li se sistem ponaša korektno kada se sve komponente koriste zajedno. Nakon ovog testiranja i sprovedenih ispravki, sistem je spreman za korisnički test prihvatljivosti.

2.4.4 Testiranje prihvatljivosti

Kada se softver pravi za određenu namenu i određenog klijenta, klijent želi da proveri da li softver zadovoljava specifikaciju. Ova verifikacija se obavlja u fazi testiranja prihvatljivosti. Ove testove kreira klijent zajedno sa test timom i izvršavaju se posle sistemskog testiranja. Cilj je evaluacija softvera u smislu klijentskih očekivanja i ciljeva. Kada se ova faza uspešno obavi, softver može da uđe u fazu produkcije. Ukoliko je u pitanju proizvod namenjen široj upotrebi koji se razvija bez klijenta, često nije moguće obaviti ovakvo testiranje prihvatljivosti. U tom slučaju, test prihvatljivosti se sprovodi kroz alfa i beta testiranje. U alfa testiranju, potencijalni korisnici sistema i članovi organizacije koja vrši razvoj sistema, testiraju proizvod pod pretpostavkama te organizacije. Posle uklanjanja defekata otkrivenih u alfa testiranju, sprovodi se beta testiranje. Proizvod se šalje korisnicima koji ga koriste u realnom okruženju i izveštavaju o pronađenim greškama [3].

2.5 Regresiono testiranje

Regresiono testiranje ima za cilj da utvrdi da li su stare karakteristike sistema i dalje zadovoljene nakon nekih izmena u sistemu i da verifikuje da promene nisu proizvele nove defekte. Regresiono testiranje nije nivo testiranja i može se primeniti na svim nivoima. Važnost regresionog testiranja je veća kod sistema koji se implementiraju dugo vremena, na čijem razvoju radi više programera i koji se objavljuju u više verzija. Funkcionalnosti implementirane u prethodnim verzijama i dalje treba da rade zajedno sa novim funkcionalnostima i verifikacija toga oduzima mnogo vremena. Za ovakve zadatke se naročito preporučuje automatizacija testiranja softvera.

3 Automatizovano testiranje softvera

U testiranju softvera, automatizovano testiranje je korišćenje specijalnog softvera, najčešće nezavisnog od softvera koji je predmet testiranja, za kontrolu izvršavanja testova, poređenje

dobijenih i očekivanih rezultata, dovođenje sistema u stanja neophodna za izvršavanje testova i druge kontrolne aktivnosti kao i aktivnosti vezane za izveštavanje [6].

3.1 Metodologije za automatizaciju testiranja softvera

Automatizacija testiranja softvera se može sprovesti na različite načine, primenom neke od postojećih metodologija. Svaka metodologija ima svoje prednosti i nedostatke tako da je često dobro rešenje kombinacija nekoliko različitih metodologija. Na odabir metodologija utiču mnogi faktori, neki od njih su:

- Softver koji se testira,
- Vreme raspoloživo za automatizaciju,
- Resursi raspoloživi za automatizaciju,
- Cilj automatizacije (smanjenje vremena potrebnog za izvršavanje testova, pokrivenost koda, pouzdanost testiranja).

U ovom poglavlju su opisane neke od metodologija i strategija za automatizaciju testiranja softvera.

3.1.1 Snimi i reprodukuj

Još od ranih 90tih godina prošlog veka postoje alati za automatizaciju testiranja softvera koji softver testiraju isključivo interakcijom sa njegovim korisničkim interfejsom. To su takozvana *out-of-the-box* (pravo iz kutije) rešenja i kao takva oduvek su imala brojne nedostatke.

Svi ovi alati zahtevaju da test inženjer najpre izvrši testni slučaj ručno, pri čemu alat snima korake koji se izvršavaju nad sistemom za testiranje. Snimak se čuva u vidu nekog skripta čijim se izvršavanjem zapravo izvršava testni slučaj. Snimak, odnosno skripta se sastoji iz niza akcija. Snimljene akcije su zapisane u obliku niza funkcija koje vrše odgovarajuće operacije nad elementima korisničkog interfejsa. Pod tim operacijama se podrazumeva, na primer, pomeranje kursora do određene pozicije i klik na dugme koje se na toj poziciji nalazi.

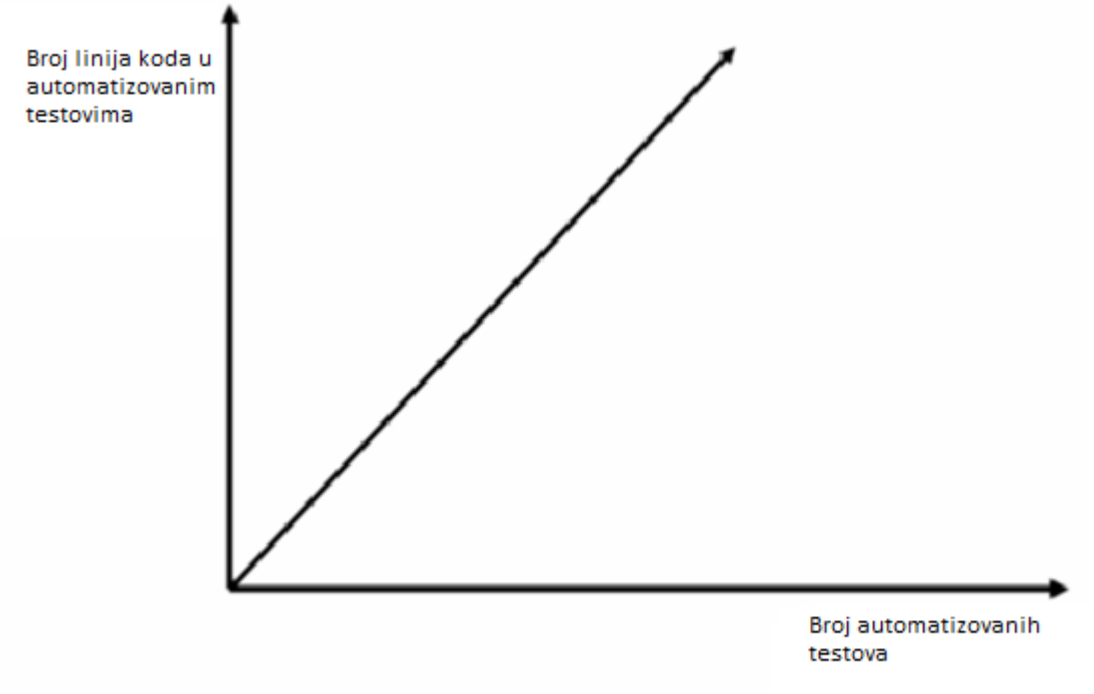
Danas je, generalno, prihvaćeno da je automatizacija testiranja softvera prostim snimanjem nekim od alata loša praksa [7, 8] .

Precizno definisanje manjkavosti *snimi i reprodukuj* strategije, nije bilo baš jednostavno. Osnovni problemi su:

- Snimanje proizvodi skriptove koji se teško održavaju jer se sastoje iz dugačkih lista akcija nad objektima i teško ih je dešifrovati.
- Snimak se neće uvek uspešno reprodukovati zbog sinhronizacionih problema. Sinhronizacija je važno pitanje u automatizaciji testiranja interfejsa. Skript nekada mora da sačeka neko vreme da se prozor pojavi. Ukoliko se za predviđeno vreme očekivani prozor ne pojavi, skript će pokušati da nastavi sa izvršavanjem akcija nad pogrešnim prozorom i naravno, neće uspeti.
- Sa *snimi i reprodukuj* alatima dobija se test program sa tvrdo kodiranim podacima. Ovo samo po sebi nije dobro po osnovnim principima softverskog inženjerstva.

- *Snimi i reprodukuj* skriptovi nisu pogodni za ponovnu upotrebu.

Razlog zbog kojeg je teško da se tačno odredi fundamentalni problem sa ovim alatima je taj što oni na malom broju slučajeva mogu da rade, tj. kada se radi o sistemu male veličine. Problem je u veličini. Ukupni broj linija koda u skriptovima je proporcionalan broju skriptova koji su kreirani. Što vise testova imamo, to imamo više linija koda za održavanje. Svaki test ima sopstveni kôd, ne postoji mogućnost za ponovno korišćenje već napisanog test skripta, ili nekog njegovog dela. Ovi brojevi imaju tendenciju da rastu sve dok cena održavanja ne postane prevelika za svaki realan bužet.
Slika 3-1 prikazuje grafički odnos broja linija koda automatizacije u odnosu na broj skriptova.



Slika 3-1 Broj linija koda automatizacije u odnosu na broj skriptova

Kako svaki test ima sopstveni skript, skoro je nemoguće dobiti testirani kôd takvom strukturom. Greška u jednom skriptu nam ne daje nikakve informacije o mogućim greškama u drugim skriptovima. Ne postoje očigledni načini za testiranje snimljenih skriptova, jer se ovi testovi često neuspešno izvršavaju iz razloga koji nisu ni greška u sistemu za testiranje, ni greška u skriptu, već, na primer, promena rezolucije ekrana koji se koristi ili promena u brzini rada sistema za testiranje. Nemoguće je dovesti ove testove u stanje visoke pouzdanosti.
Dakle, osnovni nedostatak *snimi i reprodukuj* strategije je cena održavanja. Da, jeste moguće održavati ove testove, ali cena toliko raste s porastom broja testova da se gubi na praktičnosti. Ova metodologija se može koristiti za male sisteme, za mali pažljivo odabrani broj testova.

Primer test skripta koji nastaje *snimi i reprodukuj* metodologijom je na slici 3-2. Ovaj test pokreće kalkulator aplikaciju i proverava da li se ona pojavila na ekranima. Zatim proverava sabiranje 2 broja. Za identifikovanje elementa na koje treba da klikne, test koristi slike (.png).

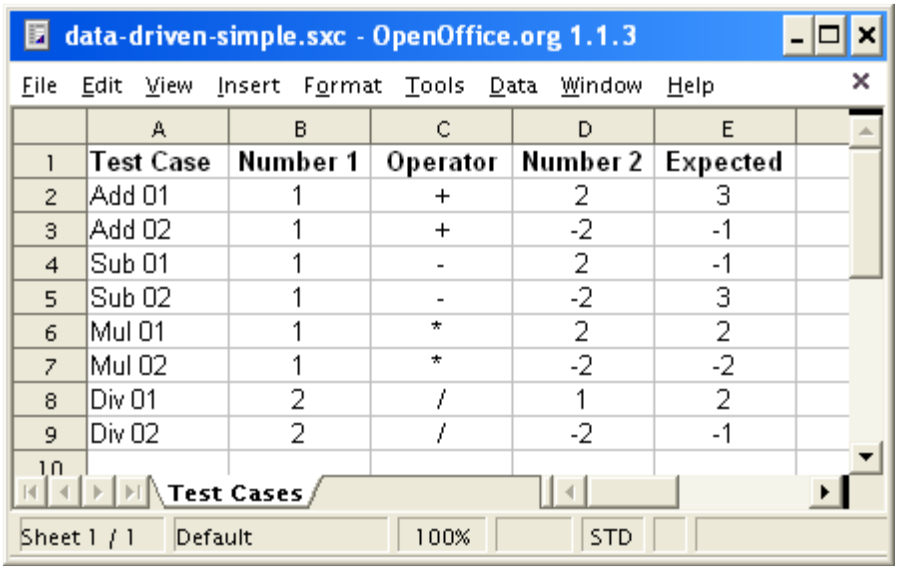
```
#Start program
calcApp = App("Calculator")
if not calcApp.window():
   App.open("calc.exe"); wait(2)
   calcApp.focus(); wait(1)
#Verify that the window appeared
if exists("CalcApp.png"):
   print("PASS: Calculator window appeared")
else:
   print("FAIL: No calculator window")
click("2.png"); click("plus.png"); click("3.png")
#Verify the addition result
find("5.png")
```

Slika 3-2 Skript nastao alatom za snimi i reprodukuj testiranje

Očigledno, za pokretanje ovog testa, potrebno je najpre napraviti slike svih elemenata koje on koristi. U ovom slučaju, to su slika izgleda kalkulatora na ekranu, slike tastera sa brojevima 2 i 3, slika tastera koji označava operaciju sabiranja i slika rezultata.

3.1.2 Testiranje vođeno podacima

Malo zrelija arhitektura je poznata pod nazivom automatizacija vođena podacima i nastala je vrlo rano u razvoju test alata. Kod *snimi i reprodukuj* testova, podaci koji se koriste kao ulazni podaci za sistem za testiranje, se tvrdo kodiraju u sam skript. Ovo negativno utiče na održavanje testova. Kada se javi potreba za promenom ovih podataka, moraju da se menjaju sami skriptovi, i to svi oni koji koriste zastarele podatke. Takođe, postoje i promene u aplikaciji koje mogu da zahtevaju izmenu svih skriptova [9]. Umesto ovog tvrdog kodiranja podataka, kod testiranja vođenog podacima, ti podaci se skladište u posebnim fajlovima koje skriptovi čitaju i koriste kao ulazne podatke za sistem za testiranje. Primer ovakvog fajla je dat na slici 3-3.



	A	B	C	D	E
1	Test Case	Number 1	Operator	Number 2	Expected
2	Add 01	1	+	2	3
3	Add 02	1	+	-2	-1
4	Sub 01	1	-	2	-1
5	Sub 02	1	-	-2	3
6	Mul 01	1	*	2	2
7	Mul 02	1	*	-2	-2
8	Div 01	2	/	1	2
9	Div 02	2	/	-2	-1
10					

Slika 3-3 Primer fajla za čuvanje podataka kod testiranja vođenog podacima

Bitna odlika ovih fajlova je da budu u formatu koji se lako čita i menja. Često se koristi tabelarni oblik, kao što je excel datoteka, baza podataka, XML datoteka ili tekstualni fajl koji ima CSV format. [9-11].

U skriptu, koji koristi ovaj fajl, postojaće funkcija koja čita te podatke i unosi ih u odgovarajući ekran sistema.

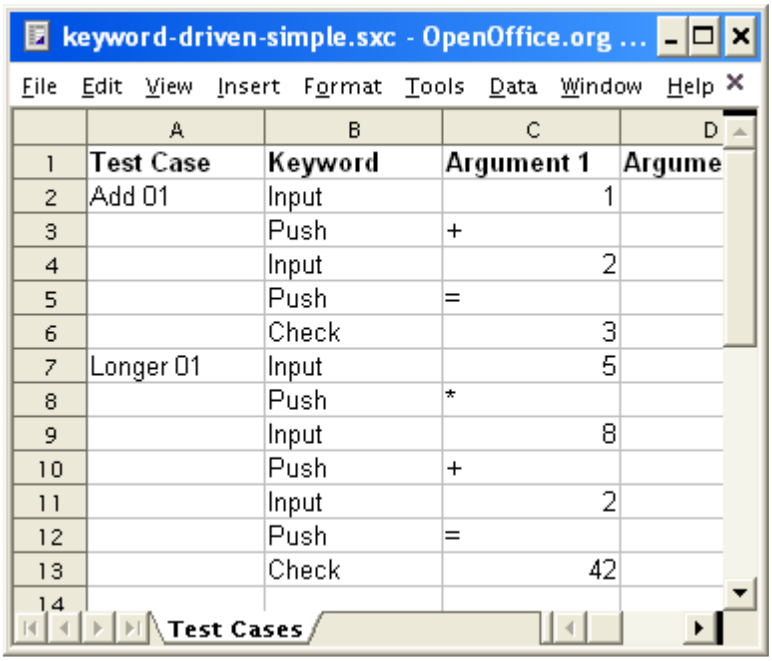
Glavna prednost testiranja vođenog podacima je mogućnost kreiranja i izvršavanja velikog broja različitih varijanti testova, na dosta jednostavan način. Više o prednostima ovakvog testiranja može se pročitati u [11].

Druga prednost je to što se testni podaci mogu kreirati rano u razvoju sistema, čak i pre same implementacije, i mogu se koristiti u manualnom testiranju. U odnosu na *snimi i reprodukuj*, kod ove strategije i održavanje je lakše. Kada se izmeni sistem, postojeći testovi se mogu prilagoditi ili izmenom samo podataka, ili izmenom u skriptovima. Najveće ograničenje testiranja vođenog podacima je u tome što su svi testovi, kreirani jednom grupom metoda, međusobno dosta slični. Da bi se kreirali novi testovi, potrebno je kreirati nove skriptove koji razumeju druge podatke i druge poslovne akcije sistema za testiranje. Ako uzmemo primer podataka sa slike 3-3, vidimo da su dizajnirani da testiraju funkcionalnosti izračunavanja koja uključuju samo 2 broja, a za bilo šta složenije, bila bi potrebna temeljna prerada i podataka i samih skriptova za čitanje i obradu.

3.1.3 Testiranje vođeno ključnim rečima

U odeljku 3.1.2 je, kao najveće ograničenje testiranja vođenog podacima, navedeno to da su svi testovi slični i da je uvođenje novih zahtevno i podrazumeva programerski trud. Rešenje ovog problema, koje su ponudili, između ostalih i Fewster i Graham [10] [11], je pristup vođen ključnim rečima. Osnova ovog pristupa je u tome što se, osim testnih podataka, u fajlove smeštaju i direktive koje govore šta sa tim podacima treba raditi. Te direktive se zovu ključne reči. Osnovna ideja ostaje

ista kao i kod testiranja vođenog podacima – pročitati podatke iz eksternih fajlova i izvršiti testove zasnovane na njima. Testiranje zasnovano na ključnim rečima je logična nadgradnja na testiranje vođeno podacima. Primer podataka za ovo testiranje dat je na slici 3-4.



	A	B	C	D
1	**Test Case**	**Keyword**	**Argument 1**	**Argume**
2	Add 01	Input	1	
3		Push	+	
4		Input	2	
5		Push	=	
6		Check	3	
7	Longer 01	Input	5	
8		Push	*	
9		Input	8	
10		Push	+	
11		Input	2	
12		Push	=	
13		Check	42	
14				

Slika 3-4 Podaci za testiranje vođeno ključnim rečima

Ne postoji suštinska razlika između baratanja podacima za testiranje vođeno podacima i onog vođenog ključnim rečima. U oba slučaja se koriste tabelarni podaci u nekom poznatom formatu ili čak centralizovana baza podataka.
Jedna od važnih odluka vezanih za testiranje vođeno ključnim rečima je nivo ključnih reči koji treba koristiti. Na slici 3-4 taj nivo je prilično nizak, samim tim one su prilagođene za detaljno testiranje na nivou interfejsa. Kada se testiraju funkcionalnosti višeg nivoa, kao što je poslovna logika, ključne reči niskog nivoa dovode do jako dugačkih testnih slučajeva i one višeg nivoa su mnogo adekvatnije. Najčešće je potrebno da postoje ključne reči oba nivoa i da se naprave odgovarajuće funkcije koje njima rukuju u skladu sa tim. Često je dobra praksa da se ključne reči višeg nivoa konstruišu od onih niskog nivoa.

Slika 3-4 prikazuje jednu od mogućnosti za konstrukciju novih ključnih reči.

Sve prednosti koje ima pristup vođen podacima, ima i pristup vođen ključnim rečima. Osnovna razlika je, kao što je već navedeno, to što nije potrebno programersko znanje za kreiranje novih vrsta testova. Poređenje slike 3-3 i slike 3-4 ovo dobro ilustruje. Zaključak je da je testiranje vođeno ključnim rečima, iako logičan naslednik onog vođenog podacima, ipak veliki iskorak napred.
Problem koji se javlja kod ovog pristupa je taj što testovi postaju sve duži i kompleksniji u odnosu na one koji nastaju korišćenjem pristupa vođenog podacima. Na primer, test add01 je dugačak

samo jednu liniju koda u prvom pristupu, a 5 linija u drugom. Ovaj problem izazvan je većom fleksibilnošću i može se prevazići korišćenjem ključnih reči višeg nivoa. Takođe, ako poredimo ova dva pristupa, očigledno je da je za testiranje vođeno ključnim rečima potrebna dosta složenija biblioteka za testiranje, a samim tim i veći programerksi trud.

3.2 Testiranje zasnovano na modelima

Testiranje zasnovano na modelima je najčešće funkcionalno testiranje za koje je specifikacija zadata test modelom. Test model se izvodi iz zahteva koje sistem treba da ispuni. Kod ovakvog testiranja, testni slučajevi i klase automatizovanih testova se (polu)automatiski izvode iz test modela. Unutrašnja struktura samog sistema za testiranje ne mora da bude poznata. Testiranje zasnovano na modelima se može primeniti na svim nivoima testiranja – od jediničnih testova do sistemskih testova.

Testiranje zasnovano na modelima ima više prednosti: model je obično mali, jednostavan, lak za održavanje. Korišćenje modela omogućava testiranje nakon razvoja sistema kao i u metodologijama koje koriste testiranje pre i tokom razvoja. Neka iskustva pokazuju da rano kreiranje formalnih test modela omogućava pronalaženje grešaka i nekonzistentnosti u samim zahtevima i specifikacijama [12]. Takođe, test-model može da se koristi za automatsko kreiranje klasa testova koje zadovoljavaju određeni kriterijum, najčešći kriterijum je pokrivenosti koda.

Postoji više pristupa testiranju zasnovanom na modelima i u narednim odeljcima su predstavljeni neki od njih.

3.2.1 Algoritmi za pretragu grafova

UML dijagrami stanja i automati, kao i mnogi drugi modeli ponašanja, su vrste grafova. Opisano ponašanje je suma svih mogućih putanja kroz te grafove. Algoritmi za pretragu grafova mogu da se upotrebe za pronalaženje putanja sa određenim osobinama. Chow [13] kreira testove iz konačnih automata koristeći grafovske algoritme, a u radu [14] se identifikuju elementi koje treba pokriti u UML dijagramima stanja i iz njih se kreiraju testovi korišćenjem grafovskih algoritama. Drugi algoritmi koriste i informacije o toku podataka za pretragu putanja kroz graf [15].

3.2.2 Slučajno testiranje

Mnogi pristupi generisanju testova iziskuju puno napora u generisanje testnih slučajeva iz modela na pametan način. Da li je taj trud uvek opravdan je podložno diskusiji. Naročito u slučaju testiranja crnom kutijom, kada je mnogo toga nepoznato i izvođenje testnih informacija je skup i težak posao. Statistički pristupi testiranju, kao što je slučajno testiranje, su uspešni u mnogim oblastima primene. U slučajnom testiranju, obično se kreira jako veliki broj testova bez razmatranja kvaliteta pojedinačnog testa. Ovakav pristup se zasniva na pretpostavci da su greške slučajno raspoređene kroz ceo sistem. U takvom sučaju, slučajno testiranje ima prednosti nad bilo kojim vođenim načinom generisanja testova. Pretpostavka da su greške obično raspoređene blizu granica klasa ekvivalencije bi ovo promenila. Ipak, prednost slučajnog testiranja nad drugim tehnikama je to što je većina tih drugih tehnika nedovoljno razvijena ili koristi specifikacije koje često i same sadrže greške.

3.2.3 Rešavanje problema zadovoljenja ograničenja (Constraint Solving)

Problem zadovoljenja ograničenja je zasnovan na skupu objekata koji moraju da zadovolje određeni set ograničenja. Proces pronalaženja tih objekata je poznat kao rešavanje problema ograničenja. Postoji nekoliko pristupa rešavanju ovih problema u zavisnosti od veličine domena u kom se problem rešava. U zavisnosti od karakteristika sistema za testiranje, bira se odgovarajući rešavač koji se primenjuje na problem generisanja testova.

3.2.4 Testiranje particionisanjem

Testiranje particionisanjem se sastoji iz kreiranja particija vrednosti ulaznih parametara i biranjem njihovih predstavnika. Ova selekcija je bitna za smanjenje troškova testiranja. Particionisanje kategorijama je metod koji je fokusiran na generisanje particija od prostora ulaznih vrednosti za testove. Napredniji pristup je particionisanje klasifikacionim drvetom koje omogućava test inženjerima da definišu proizvoljne particije i izaberu njihove predstavnike. Nekada se koriste i težinske strategije u kojima se kreiraju klase testnih slučajeva koje imaju različite prioritete.

3.2.5 Sečenje

Sečenje je tehnika za eliminisanje delova programa ili modela da bi se uklonili nepotrebni delovi i time uprostilo generisanje testova. Idea je da su odsečeni delovi lakši za razumevanje i samim tim iz njih se lakše generišu testovi, nego iz kompletnog modela.

3.3 Fazi testiranje

Fazi (eng. *Fuzzy*) testiranje softvera se zasniva na generisanju nevalidnih, slučajnih ili neočekivanih ulaznih podataka u sistem i proveravanjem kako se sistem ponaša kada mu se daju ovakvi podaci. Ovo testiranje je korisno za testiranje rukovanja greškama i izuzecima kao i upravljanja memorijom. Fazi testiranje može da koristi i metodologiju crne i metodologiju bele kutije, od čega zavisi način generisanja ulaznih podataka.

Kada se testiranje vrši metodologijom crne kutije, ulazni podaci se generišu ili mutacijom nekog seta ulaznih parametara, ili korišćenjem nekog šablona (gramatike), koji opisuje set ulaznih podataka [22]. Prvi način je češće korišćen jer omogućava da se za kratko vreme kreira mnogo ulaznih podataka koji često uspevaju da otkriju veliki broj grešaka.

Pri korišćenju metodologije bele kutije, generisanje podataka koristi unutrašnju strukturu i dizajn sistema za generisanje parametara. Pri izvršavanju sistema sa prvim (zadatim) setom ulaznih parametara, prate se stanja u sistemu i generišu se ograničenja za parametre. Ova ograničenja se prave ispitivanjem svih uslovnih grananja u programu (npr. *if/else* grane). Najčešće, prikupljanje ograničenja se postiže simboličkim izvršavanjem sistema za testiranje. Simboličko izvršavanje sistema podrazumeva da se, pri izvršavanju tog sistema, koriste simboličke vrednosti za sve parametre, umesto stvarnih vrednosti. Pri tom se generišu tabele sa ciljem da se u njih smeste sva ograničenja svih parametara. Generator podataka na ova ograničenja primenjuje metode za rešavanje problema zadovoljenja ograničenja da bi proizveo podatke koji, kada se sprovedu u sistem, izvršavaju što više putanja kroz sistem, ili neki njegov deo [22]. Teoretski, posle dovoljno izvršenih testova, trebalo bi da se postigne potpuna pokrivenost koda sistema, odnosno da testovi

izvrše celokupan kod sistema. U praksi je ovo često nemoguće zbog veličine sistema i broja parametara, sporosti simboličkog izvršavanja, kao i sporosti samih metoda za rešavanje problema zadovoljenja ograničenja.

3.4 Alati za automatizaciju testiranja softvera

Postoji mnogo besplatnih i komercijalnih alata koji omogućavaju automatizaciju testiranja softvera. Mogu se podeliti u grupu alata za jednično testiranje i grupu alata za testiranje kroz korisnički interfejs.

3.4.1 Alati za jedinično testiranje

Za većinu programskih jezika postoji biblioteka koja podržava jedinično testiranje, a najčešće te biblioteke podržavaju i kreiranje kodiranih testova višeg nivoa.
Ono što je zajedničko za alate ovog tipa je da svi omogućavaju:

- Kreiranje testova, tj. test metoda i njihovu diferencijaciju od ostalih metoda,
- Kreiranje metoda koje izvršavaju operacije postavljanja (eng. *Setup*) i čišćenja (eng. *Cleanup*), a koje se izvršavaju pre, odnosno posle, jednog testa ili grupe testova. Metode postavljanja se koriste za inicijalizaciju svega što je potrebno za izvršavanje nekog testa. To mogu biti, na primer, globalne promenljive, neki slogovi u bazi, ili objekti koji će se koristiti u više testova koji imaju zajedničku metodu postavljanja. Metoda čišćenja vrši uklanjanje svega što su testovi napravili. Ovde spada, na primer, oslobađanje memorije i brisanje testnih podataka iz baze.
- Metode za poređenje dobijenog i očekivanog rezultata metode koja se testira. Od rezultata te metode zavisi da li će test biti uspešno izvršen.

Za programski jezik Java, najpoznatija biblioteka je JUnit, koja je otvorenog koda.
Za C++ se najčešće koristi CppUnit, a za programski jezik Ruby se koristi RSpec. Za C# i druge jezike koji se oslanjaju na .Net biblioteku, postoji MSTest biblioteka, koja je ugrađena u MS Visual Studio okruženje.

Za testiranje Java koda, korišćenjem JUnit biblioteke, potrebno je, najpre, napraviti klasu koja proširuje TestMethod klasu iz pomenute biblioteke. U toj klasi se, dalje, pišu metode koje su označene atributom @Test. Taj atribut označava da je u pitanju test metoda. Unutar test metoda se koriste poredbene metode.

Primer testa napisanog korišćnjem JUnit biblioteke je ispod.

```
public class TestClass extends TestMethod {
   String str1;
   String str2:

   @Before
   public void before() {
      str1 = new String ("abc");
      str2 = new String ("abc");
}
   @Test
   public void testStrings() {
      assertEquals(str1, str2);
   }
}
```

Iz primera se vidi da se metoda koja se izvršava pre testa označava atributom @Before. U ovom primeru korišćna je i metoda assertEquals(), koja poredi jednakost prosleđenih objekata.
Isti test napisan korišćenjem MSTest alata izgleda ovako:

```
[TestClass]
public class AuthenticatorTests
{
    string str1 = null;
    string str2 = null;

    [TestInitialize()]
    public static void Setup()
    {
        str1 = "abc";
        str2 = "abc";
    }

    [TestMethod]
    public void CreateUserValidParameters()
    {
        Assert.AreEqual(str1, str2);
    }
}
```

Očigledno je da je način na koji se koriste JUnit i MSTest biblioteke jako sličan. Isto važi i za ostale biblioteke za jedinično testiranje. Osnovne razlike između različitih biblioteka su sintaksne.

3.4.2 Alati za testiranje kroz korisnički interfejs

Za testiranje aplikacija kroz korisnički interfejs postoji veliki broj alata koji omogućavaju neku vrstu *snimi i reprodukuj* testiranja. Ono što je zajedničko za većinu njih je da ne zahtevaju nikakvo programersko znanje. Osnovni princip njihovog funkcionisanja se može podeliti u 5 koraka:

- Pokrenuti snimanje testa
- Izvršiti operacije potrebne za test nad interfejsom sistema za testiranje
- Definisati željeni ishod (na primer, očekujemo da se u određenom broju nalazi određeni broj)

- Prekinuti snimanje testa
- Pokrenuti snimljeni test

Jedan od najpoznatijih alata za ovakvo testiranje je Selenium. Postoji mnogo verzija ovog alata, svaka dolazi sa grafičkim okruženjem za snimanje i izvršavanje testova. Jedna od verzija Selenium alata podržava i testiranje web aplikacija i može se instalirati kao dodatak za Firefox pretraživač, besplatno. Za razliku od većine drugih alata, Selenium ima mogućnost kreiranja test skriptova u više različitih programskih jezika, tako da programeri mogu da ih izmene i dopune po potrebi.
Za Windows aplikacije, jedan od često korišćenih alata je WinRunner. WinRunner podržava kreiranje mape grafičkog korisničkog interfejsa, koja omogućava identifikovanje elemenata interfejsa po zadatim kriterijumima i praćenje akcija koje se nad objektima u mapi izvršavaju. Takođe, moguće je snimiti i testove koji prate kretanje kursora po ekranu, a koji su pogodni za testiranje aplikacija koje imaju funkcionalnosti crtanja.
TestComplete je još jedan od alata koji omogućavaju testiranje aplikacija kroz njihog grafički korisnički interfejs. On podržava i višestruke validacije u toku jednog testa, što znači da možemo da, u toku izvršavanja nekog kompleksnog scenarija, više puta proverimo stanje sistema, u različitim trenucima. Ova mogućnost znatno olakšava pronalaženje greške u sistemu kada se neki od složenih testova neuspešno izvrši.

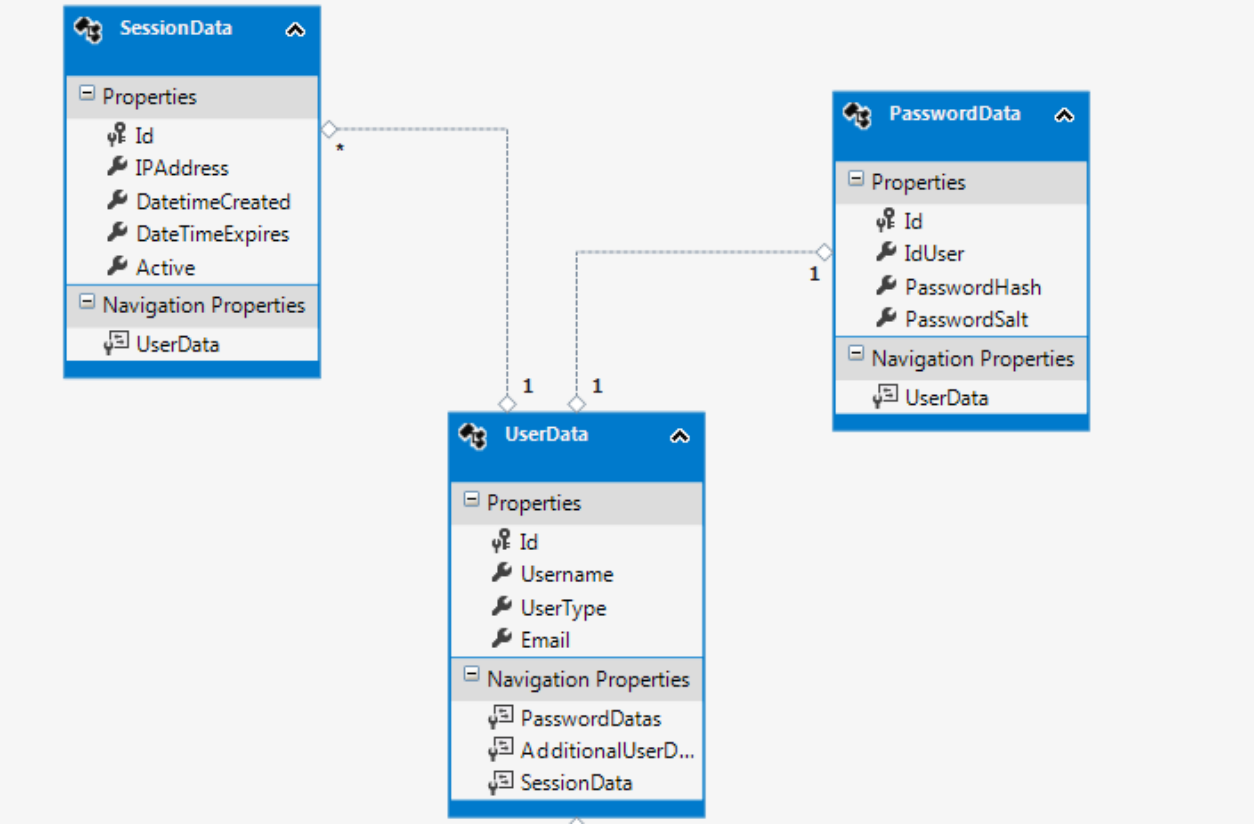
4 Sistem za testiranje

Sistem koji ćemo koristiti za ilustraciju nekih od metoda automatizovanog testiranja je sistem za registrovanje korisnika i proveru korisničkih podataka prilikom prijavljivanja korisnika na sistem. Pri prijavljivanju korisnika na sistem kreira se sesija. Sesija je, kada je web programiranje u pitanju, polu-trajna interaktivna razmena informacija između 2 računara. Po pravilu, sesija ima rok trajanja, odnosno prekida se po isteku određenog vremenskog intervala. U slučaju ovog sistema, sesija se kreira u trenutku kada se korisnik uspešno autentifikuje i ističe zadati broj minuta kasnije (podrazumevano vreme trajanja sesije je 30 minuta). Ona nam omogućava da klijent, to jest autentifikovani korisnik, može da, za vreme trajanja sesije, komunicira sa serverom bez da se njegovo korisničko ime i lozinka proveravaju pri svakoj poruci. Svi podaci vezani za sesiju se čuvaju u bazi podataka. Taj softver je deo većeg sistema, kojeg je razvio autor, a o kojem neće biti reči u ovom radu.
Celokupan sistem se sastoji iz sledećih delova:

- Baza podataka u kojoj se skladište podaci o korisnicima i aktivnim sesijama (korišćena je MS SQL baza)
- Servisa koji pruža usluge kreiranja korisnika, provere korisničkih podataka i kreiranja odgovarajućih sesija (korišćen je WCF alat)
- Web aplikacije koja, preko WCF servisa, krajnjem korisniku omogućava da kreira novog korisnika u sistemu, proveri korisničko ime i lozinku i prijavi korisnika na sistem.

Sistem je razvijen u programskom jeziku C#, u okruženju VISUAL STUDIO 2012, uz korišćenje *.Net Framework*-a verzije 4.0

Softver je sačinjen od nekoliko slojeva. Najniži sloj je baza podataka u kojoj se skladište informacije o registrovanim korisnicima i njihovim sesijama. Šema baze je prikazana na slici 4-1.



Slika 4-1 Šema baze podataka u kojoj se čuvaju informacije o korisnicima

Sloj koji upravlja bazom podataka čine klase UserManipulator i SessionManipulator. One u sebi sadrže metode za kreiranje novog korisnika u bazi podataka, proveru korisničkih podataka (korisničkog imena i lozinke) i za kreiranje i brisanje sesija. U tim metodama se direktno pristupa bazi. Funkcionalnosti koje ove klase treba da implementiraju su:

- Kreiranje novog korisnika u bazi, koje se sastoji iz kreiranja novih slogova u tabelama UserData i PasswordData, na osnovu prosleđenih parametara
- Validacija postojanja korisnika koji odgovara zadatom korisničkom imenu i lozinci
- Kreiranje novog sloga u tabeli SessionData za korisnika sa zadatim identifikatorom
- Brisanje sloga iz tabele SessionData za korisnika sa zadatim identifikatorom
- Validacija postojanja sloga u tabeli SessionData za korisnika sa zadatim identifikatorom
- Dobavljanje informacija o sesiji za korisnika sa zadatim identifikatorom

Korisničke lozinke se u sistemu čuvaju kao heš vrednosti lozinki koje su korisnici odabrali, zajedno sa niskom karaktera koja služi za soljenje pri heširanju. Klasa Hasher sadrži implementaciju algoritma za heširanje. Heširanje je proces preslikavanja skupa ulaznih podataka, koji su promenljive dužine, na skup podataka koji su fiksne dužine. Algoritam implementiran u klasi

Hasher na ulaznu nisku karaktera dodaje slučajnu nisku karaktera, tj. so, i nad tako dobijenom niskom obavlja niz bitovnih operacija, koje proizvode nisku veličine tačno 256 karaktera. Ovaj proces se, u okviru jednog heširanja, obavlja više puta. Soljenje se koristi kao dodatni vid zaštite od različitih napada na sistem.

Servis je aplikacija koja implementira servisno orijentisanu arhitekturu. Ova arhitektura se zasniva na kreiranju razdvojenih delova softvera koji, drugim aplikacijama, pružaju određene usluge, odnosno funkcionišu kao servisi. Servisi usluge mogu da pružaju i drugim servisima. Primalac usluga se zove klijent. WCF je alat za pravljenje servisno orjentisanih aplikacija koje podržavaju distribuiranost, u smislu da servis mogu da koriste i udaljeni klijenti. Kod servisa napravljenih korišćenjem WCF-a, komunikacija sa klijentom se obavlja kroz poruke koje su u XML formatu, što znači da korišćenje servisa ne zavisi od platforme klijenta. Više informacija o servisno orijentisanoh arhitekturi i WCF alatu se može naći u [21].

WCF servis implementiran u sistemu koji se testira u ovom radu je implementiran nad UserManipulator, SessionManipulator i Hasher klasama. Servis koristi ove klase i njihove metode kako bi omogućio sledeće servisne operacije:

- Kreiranje novog korisnika
- Autentifikacija postojećeg korisnika
- Prijavljivanje autentifikovanog korisnika na sistem kreiranjem odgovarajuće sesije

U narednim poglavljima biće opisano testiranje sistema na više različitih nivoa, primenom različitih metodologija i tehnika. Takođe, biće opisane i neke smernice, čijim praćenjem se dobija testabilan kôd.

Web aplikacija je razvijena primenom MVC projektnog uzorka, u ASP.NET tehnologiji. MVC projektni uzorak podrazumeva raslojavanje arhitekture sistema na tri dela: model (eng. *Model*), pogled (eng. *View*) i kontroler (eng. *Controller*). Centralna komponenta je model, on upravlja podacima i sadrži svu poslovnu logiku. Pogled je izlazna reprezentacija podataka, odnosno ono što klijent vidi. U pogledu je definisan sam izgled korisničkog interfejsa. Kontroler ima ulugu da zahteve koji dolaze iz pogleda (najčešće od korisnika) prosledi odgovarajućem delu modela, kao i da podatke koji dolaze od modela prosledi odgovarajućem pogledu.

U slučaju ovog sistema, web aplikacija u svom modelu koristi usluge servisa, a pogled predstavlja grafički korisnički interfejs koji omogućava autentifikaciju i prijavljivanje korisnika na sistem.

4.1 Testabilnost

Kod automatizovanog testiranja koje ne koristi isključivo korisnički interfejs, već direktno pokreće kôd aplikacije, često se javlja problem testabilnosti koda sistema za testiranje. Ako se ne razmišlja o automatizaciji testiranja u toku razvoja samog sistema, često je, bez refaktorisanja, nemoguće napisati zadovoljavajuću automatizaciju. Najčešći problemi su vezani za apstrahovanje izvora podataka i izolaciju pojedinačnih klasa. Neki načini za rešavanje ovih problema u toku razvoja sistema za testiranje su predstavljeni u poglavljima 4.1.1 i 4.1.2.

4.1.1 Lažni kontekst podataka

U sistemu za autentifikaciju, baza podataka je apstrahovana i predstavljena kroz DataContext klasu. Ova klasa je sastavni deo .Net biblioteke i predstavlja preslikavanje baze podataka na entitete, odnosno klase koje sadrže odgovarajuća polja. Svaka tabela iz definisane baze podataka dobiće u kontekstu podataka jednu, automatski generisanu, klasu koja je predstavlja. Za automatizovano testiranje je potrebno da možemo da, bez promene produkcionog kôda, koristimo ili lažni ili testni kontekst podataka. Testni kontekst podataka je DataContext klasa koja je preslikana na bazu podataka sa podacima koji se koriste za testiranje. Ovakva baza nam omogućava da testiranje vršimo relevantnim podacima i i da pri tom ne menjamo stanje produkcione baze podataka. Lažni kontekst podataka je slika baze podataka koja postoji u memoriji. Kod lažnog konteksta podataka koriste se samo klase koje predstavljaju strukturu baze, a koje su kreirane u kontekstu podataka. Kako su te klase kolekcije objekata koje predstavljaju odgovarajuće redove u bazi, možemo da ih kreiramo kroz kôd, tako da one postoje u memoriji samo dok se program, ili u ovom slučaju test, izvršava. Lažni kontekst podataka je idealan za jedinično testiranje, kao i svako drugo testiranje koje nema za cilj da proverava konekciju ka bazi podataka i samu bazu. Korišćenje lažnog konteksta takođe pozitivno utiče na brzinu izvršavanja testova, tako što test ne uključuje kreiranje i oslobađanje konekcije na bazu podataka.

DataContext klasa ne implementira ni jedan interfejs, što znači da za kreiranje lažnog konteksta treba da napravimo omotač oko same DataContext klase.

```
public interface IDataContextWrapper : IDisposable
    {
        List<T> Table<T>() where T : class;
        void DeleteAllOnSubmit<T>(IEnumerable<T> entities) where T : class;
        void DeleteOnSubmit<T>(T entity) where T : class;
        void InsertOnSubmit<T>(T entity) where T : class;
        void SubmitChanges();
    }
```

Dalje, napravićemo klasu koja ga implementira:

```
public class DataContextWrapper<T> : IDataContextWrapper where T : DataContext, new()
```

Ova generička klasa može da bude instancirana sa bilo kojim *DataContext* objektom, ili objektom koji nasleđuje DataContext klasu. Koristićemo je kao stvarnu implementaciju koja radi direktno sa bazom podataka. Druga implementacija IDataContextWrapper interfejsa će biti MockDataContextWrapper klasa koja će raditi sa lažnom bazom podataka.

Lažna baza podataka je klasa koja implementira jednostavan interfejs koji sadrži samo deklaraciju kolekcije tabela:

```
public interface IMockDatabase
    {
```

```
        Dictionary<Type, IList> Tables { get; set; }
    }
```

Ova apstrakcija konteksta podataka ne spada u testove, ali spada u infrastrukturu potrebnu za efikasno automatizovano testiranje.

Apstrahovanjem klasa za rad sa podacima, kao i kreiranjem pomoćnih interfejsa i klasa smo postigli to da sada, umesto tvrdo-kodiranog konteksta podataka, možemo da koristimo omotač. Za kompletnu implementaciju lažnog konteksta podataka potrebno je i da kôd implementira ubrizgavanje zavisnosti (eng. *Dependency injection*).

4.1.2 Ubrizgavanje zavisnosti

Najčešći način korišćenja DataContext objekata je njihovo instanciranje u samoj klasi koja ga koristi. Takav kôd čini da kontekst podataka i klasa koja ga koristi budu čvrsto povezani (eng. *tightly coupled*). Za ukidanje te čvrste veze se koristi projektni uzorak koji se zove ubrizgavanje zavisnosti. On se implementira tako što, umesto da se neka klasa instancira unutar same klase koja ga koristi, ona se prosledi njoj kao parametar u konstruktoru. Na ovaj način, možemo da, na primer, produkcionom kôdu prosleđujemo kontekst podataka koji odgovara produkcionoj bazi, a kôdu koji je testni, prosleđujemo testni kontekst podataka. Sama klasa i njene metode su na ovaj način nezavisne od konteksta podataka koji koriste. Ako ovaj obrazac povežemo sa lažnim kontekstom podataka kreiranim u poglavlju 4.1.1, to znači da klasi koja koristi kontekst podataka treba da, kroz konstruktor, prosledimo omotač oko konteksta podataka. Ta klasa će onda da radi sa omotačem umesto direktno sa kontekstom i biće fleksibilna u odnosu na konkretni konktekst podataka koji koristi. Konkretno, klasa *Authenicator* je, pre uvođenja lažnog konteksta podataka i ubrizgavanja zavisnosti, imala konstruktor koji instancira DataContext klasu:

```
public UserManipulator(string userName, string password)
        {
            _userName = userName;
            _password = password;
            _dataContext = new CredentialsEntities();
        }
```

Nakon implementacije dva opisana principa, *UserManipulator* klasa se instancira dodatnim parametrom:

```
public UserManipulator( IDataContextWrapper dataContextWrapper ,
                                string userName, string password)
        {
            _userName = userName;
            _password = password;
            _dataContext = dataContextWrapper;
        }
```

Pri instanciranju ove klase, prosledićemo neku od kreiranih implementacija *IDataContextWrapper* interfejsa.

4.2 Jedinični testovi klasa koje manipulišu podacima

Za testiranje koda korišćena je konvenciju da se za svaku metodu implementiranih klasa, koja je javna za ceo projekat ili za jedan izvršni fajl (metodu sa atributom vidljivosti *public* ili *internal*), napišu po bar 2 jedinična testa. Za svaku metodu potrebo je da postoji barem jedan pozitivan i barem jedan negativan test. Pod pozitivnim testovima, podrazumevaju se oni koji vode ka uspešnom izvršavanju metode, tj. testovi sa ulaznim podacima koji izazivaju regularan tok metode. Negativni testovi su oni koji izazivaju alternativno ponašanje metode. Pod alternativnim ponašanjem se misli na odgovarajući izuzetak (eng. *Exception*) ili na poruku o grešci (eng. *Error Message*).

Jedinični testovi, po svojoj definiciji, ne treba da se oslanjaju na druge delove sistema. Apstrahovanje ostatka sistema od dela koji se testira postižemo korišćenjem lažnog konteksta podataka i ubrizgavanja zavisnosti.

4.2.1 Jedinični testovi klase UserManipulator

Prva klasa za koju ćemo napisati jedinične testove je UserManipulator klasa. Fokus je na metodama te klase koje su dostupne iz drugih delova sistema:

1. bool ValidateUser()
2. bool CreateUser().

Obe se pozivaju nad instancom UserManipulator klase, kreirane konstruktorom koji je opisan u prethodnom poglavlju. Očekivano ponašanje metode, koja kreira novog korisnika, uključuje proveru validnosti podataka i odgovarajuću poruku o grešci ako neki od njih nije validan. U slučaju validnih parametara, u bazi se kreira korisnik čiji svi podaci odgovaraju unetim, a korisnička šifra se čuva u vidu svog heša kreiranog pomoću soljenja. Pored šifre čuva se i niska karaktera korišćena za soljenje. 3 jedinična testa kreirana za ovu metodu pokrivaju uspešno kreiranje korisnika, poruku o grešci kada korisničko ime nije uneto i poruku o grešci kada lozinka nije uneta. Korišćenjem metode particionisanja, dolazimo do zaključka da svaki od ulaznih parametara može biti:

- Validan string (string koji nije prazan i vrednost mu nije *null*)
- Nevalidan prazan string

- Nevalidan string vrednosti *null*
- Izlazni parameter može biti:
- *true*, kada je korisnik uspešno kreiran
- *false*, kada korisnik nije uspešno kreiran

Pošto su u pitanju jedinični testovi, koristimo prednost testiranja metodologijom bele kutije. U kodu se validnost ulaznog parametra proverava metodom `string.IsNullOrEmpty(_userName)`. Dakle, nije neophodno da proveravamo obe opcije nevalidnog ulaznog parametra. Kombinujući ulazne i izlazne parametre, to jest klase njihovih vrednosti, napravljeni su sledeći testovi:

- CreateUserValidParameters
- CreateUserInvalidUsername
- CreateUserInvalidPassword.

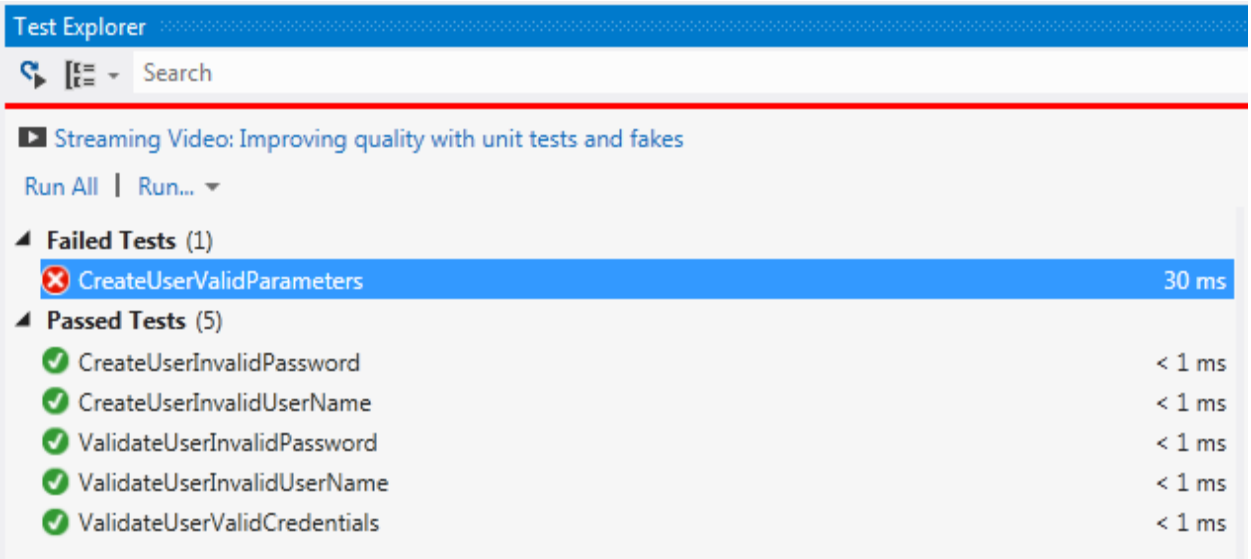
Istom metodologijom kreirane su i tri test metode za metod ValidateUser():

- ValidateUserValidCredentials
- ValidateUserInvalidUsername
- ValidateUserInvalidPassword.

Svi testovi se nalaze u istoj klasi testova, jer se odnose na istu klasu iz sistema koji se testira. Da bi se smanjilo vreme izvršavanja testova, kao i ponavljanje koda, u klasi testova je napisana jedna metoda koja obavlja sve funkcije inicijalizacije, u ovom slučaju to je instanciranje lažne baze podataka i lažnog konteksta podataka. Metoda je okićena atributom ClassInitialize, koji omogućava da se ona izvrši jednom za sve testove iz iste klase testova, i to pre izvršavanja samih testova. Svi testovi su koristili lažni kontekst podataka i ubrizgavanje zavisnosti; inicijalizacija lažnog konteksta podataka i lažne baze podataka je izvršena u metodi inicijalizacije, kôd je prikazan ispod.

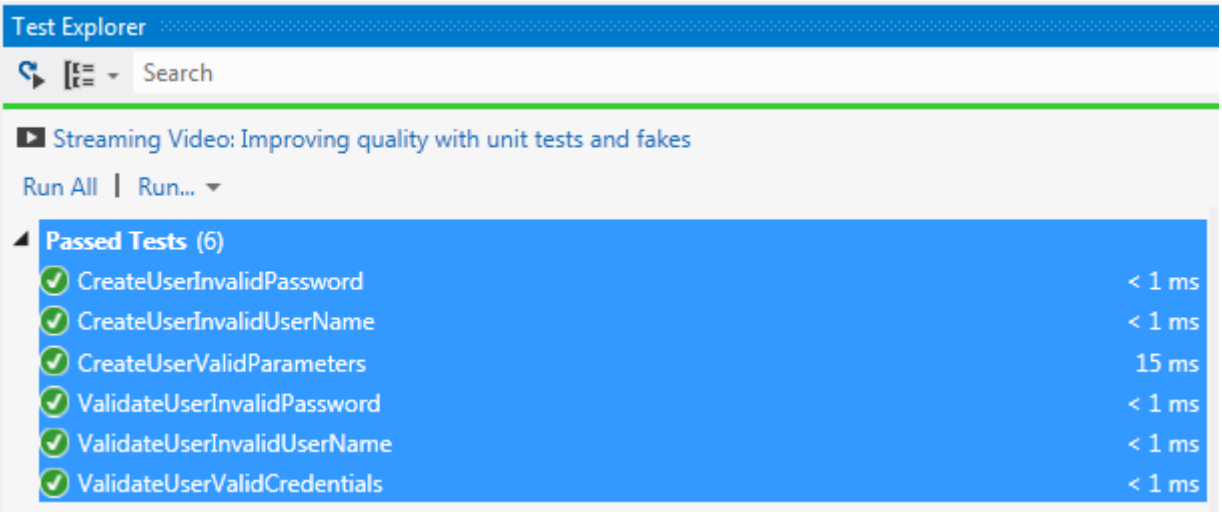
```
[ClassInitialize()]
 public static void AuthenticatorTestSetup(TestContext tc)
 {
     MockDatabase _db = new AuthenticationDBMock();
     IDataContextWrapper _wrapper = new MockDataContextWrapper(_db);
 }
```

U prvom izvršavanju, test koji provarava validaciju korisnika sa ispravnim korisničkim imenom i lozinkom je neuspešno izvršen, ostali su uspešno izvršeni. Slika 4-2 prikazuje rezultate.



Slika 4-2 Rezultati prvog izvršavanja testova - 1 neuspešan test

Analizom koda koji se izvršava pri validaciji korisnika, ustanovljeno je da se, pri kreiranju heš vrednosti korisničke lozinke, ne koristi niska karaktera za soljenje kreirana pri kreiranju korisnika, koja se nalazi u bazi podataka, već se kreira nova. Nakon ispravljanja greške i ponovnog pokretanja jediničnih testova, svi su uspešno izvršeni. Slika 4-3 ilustruje rezultat drugog izvršavanja jediničnih testova. Bitna osobina ovih jediničnih testova je to da su se izvršili za manje od jedne sekunde, što znači da nam za proveru ispravnosti koda ne treba mnogo vremena.



Slika 4-3 Uspešno izvršeni jedinični testovi

4.2.2 Jedinični testovi klase SessionManipulator

Kao što je ranije opisano, klasa SessionManipulator je zadužena za manipulaciju podacima u tabeli SessionData. Njene osnovne metode su:

- Int CreateSession(int userId, string ipAddress, DateTime currentTIme, TimeSpan sessionDuration)
- bool ValidateSession(int userId, string ipAddress)
- bool ValidateSession(int sessionId)
- bool DeleteSession(int sessionId)
- bool DeleteSession(int userId, string ipAddress)

Za testiranje ovih metoda je korišćen lažni kontekst podataka. U njemu su kreirani podaci koji odgovaraju sledećim scenarijima:

1. Validni korisnik sa aktivnom sesijom
2. Validni korisnik sa sesijom koja je istekla
3. Validni korisnik bez sesije.

Sve testne metode koriste ove testne podatke pri izvršavanju.

Od metode za kreiranje sesije očekuje se da za prosleđene parametre ispravno unese slog u tabelu SessionData. Ispravni unos podataka znači da identifikator korisnika, IP adresa, kao i datum i vreme budu identični onima koji su prosleđeni i da vreme isteka sesije bude ispravno izračunato. Ova metoda pozivaocu vraća identifikator kreiranje sesije. U slučaju nekog nevalidnog podatka, metoda treba da baci izuzetak tipa InvalidArgumentException. Potpisi testnih metoda izgledaju ovako:

```
[TestMethod]
public void CreateSessionValidParameters();

[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void CreateSessionInvalidUserId();
```

Izvršavanjem testova je otkriveno da, pri prosleđivanju nepostojećeg korisničkog identifikatora (parametar *userId*), izuzetak koji se baca ne odgovara očekivanom.

Od metoda za validacuju sesije se ošekuje da vrate vrednost *true* kada sesija koja odgovara parametrima postoji i aktivna je (nije istekla), odnosno *false* u suprotnom.

Kreirani su sledeći testovī:

```
[TestMethod]
public void ValidateSessionByIdValidSessionIdValidSession();
[TestMethod]
public void ValidateSessionByIdValidSessionIdExpiredSession();
[TestMethod]
public void ValidateSessionByIdInvalidSessionId();
[TestMethod]
public void ValidateSessionByUserValidUserDataExpiredSession();
[TestMethod]
public void ValidateSessionByUserValidUserDataValidSession();
[TestMethod]
public void ValidateSessionByUserInvalidUserData();
```

Prva tri testa se odnose na metodu za validaciju sesije koja se pronalazi na osnovu identifikatora sesije, druga tri testa se odnose na metodu koja validira sesiju koja se pronalazi na osnovu identifikatora korisnika i korisničke IP adrese. Pri prvom izvršavanju testova, neuspešno su izvršeni sledeći testovi:

```
public void ValidateSessionByIdInvalidSessionId();
public void ValidateSessionByUserInvalidUserData();
```

Analizom samih metoda koje su testirane, ustanovljeno je da ne postoji provera postojanja podataka u bazi pre njihove validacije. Nakon impementacije dodatne provere, svi testovi su uspešno izvršeni.

Metoda za brisanje sesija treba da vrati vrednost *true* kada je brisanje uspešno izvršeno, odnosno *false* ako je brisanje bilo neuspešno. Jedinični testovi su napisani korišćenjem iste metodologije kao i za prethodne metode: po jedan test za validne ulazne parametre, i po jedan za nevalidne. Svi testovi koji su se odnosili na ovu metodu su uspešno izvršeni, dakle nije pronađena ni jedna greška.

4.3 Jedinični testovi klase *Hasher*

Klasa *Hasher* ima jednu javno dostupnu metodu, deklarisanu kao:

```
public string HashData(string input, string salt)
```

Algoritam implementiran u okviru te metode neće biti prikazan u ovom radu zbog zaštite autorskih prava.

Prvi test odnosi se na determinisanost funkcije. Očekujemo da, za iste ulazne parametre, svaki put dobijemo istu izlaznu vrednost. Ovde je bitno da pokušamo sa mnogo različitih ulaznih vrednosti. Koristićemo pristup testiranja vođenog podacima, i napisaću 2 automatizovana testa. Prvi test služi za ispitivanje da li iste niske karaktera sa istom niskom karaktera za soljenje proizvode istu vrednost heša, a druga da različite niske karaktera, uz istu nisku karaktera za soljenje proizvode različitu. Niske karaktera, koje se koriste kao ulazni parametar *input*, treba da mogu da sadrže karaktere iz engleskog alfabeta, karaktere iz srpskog alfabeta, specijalne karaktere i cifre. Maksimalna dužina ulazne niske je 256 karaktera. Na osnovu ovih podataka, možemo napraviti nekoliko klasa ekvivalencije za ulazni parametar *input,* koje smatramo bitnim*:*

- Niska dužine 1, element je karakter
- Niska dužine 1, element je specijalni karakter
- Niske dužine između 2 i 255, elementi su karakteri engleskog alfabeta
- Niske dužine između 2 i 255, elementi su karakteri srpskog alfabeta
- Niske dužine između 2 i 255, sadrži elemente iz srpskog alfabeta, specijalne karaktere i cifre

- Niske dužine 256 karaktera, sadrži elemente iz srpskog alfabeta, specijalne karaktere i cifre

Za svaku klasu ekvivalencije, kreirano je po 10 predstavnika i oni čine podatke za test vođen podacima. Test za zadati parametar kreira 2 instance klase Hasher. Nad obe instance poziva metodu *HashData* i prosleđuje iste vrednosti za parametar *input*, koji dobija iz excel datoteke, i isti parametar za soljenje. Kôd automatizovanog testa je prikazan ispod.

```
[TestMethod()]
[DeploymentItem("App.config")]
[DataSource("PasswordData")]
public void HashingDataValidateEquality()
{
    string constantSalt = "temporaryExampleSalt";
    string password = TestContext.DataRow["Password"].ToString();
    Hasher hasher1 = new Hasher();
    Hasher hasher2 = new Hasher();
    string hash1 = hasher1.HashData(password, constantSalt);
    string hash2 = hasher2.HashData(password, constantSalt);
    Assert.AreEqual(hash1, hash2);
}
```

Atribut DataSource nam govori gde se nalazi izvor podataka, a DeploymentItem atribut nam govori da je potrebno uključiti i konfiguracioni fajl pri kompajliranju.

Test metoda se izvršila ukupno 60 puta, za svaki podatak po jednom. Svih 60 puta se izvršila neuspešno. Nakon naknadnog ispitivanja, moglo se uočiti da metoda ne uzima u obzir prosleđenu nisku za soljenje, već pri svakom pozivu kreira novu slučajnu nisku za soljenje. Pošto je ova greška u sistemu ispravljena, prelazimo na drugu test metodu koja treba da pokaže da se za različite niske karaktera i isto soljenje dobija različita vrednost heša. Ono što je bitno je da se za slične niske karaktera dobijaju različite vrednosti heša. Pod sličnim niskama karaktera se podrazumevaju:

- Niske koje se razlikuju u prvom karakteru
- Niske koje se razlikuju u poslednjem karakteru
- Niske koje se razlikuju u jednom karakteru koji nije ni prvi ni poslednji
- Niske koje se razlikuju po tome što jedna sadrži karaktere srpskog alfabeta a druga analogni karakter iz engleskog, svi ostali karakteri su jednaki (u jednoj nisci se nalazi, na primer č, dok se u drugoj na mestu tog karaktera nalazi c)

Ovaj opis sličnih karaktera je ujedno odabran i kao relevantne klase ekvivalencije. Za svaku od klasa ekvivalencije smo, u excel datoteci, napisali po 2 niske karaktera. Kôd samog testa je sličan onom napisanom za validaciju ponašanja metode kada su u pitanju identične niske karaktera, s tim što se testira nejednakost povratnih vrednosti metode. Za ukupno 8 parova ulaznih parametara, test se izvršio 4 puta neuspešno. Neuspešna izvršavanja su bila ona koja su testirala niske koje se razlikuju u poslednjem karakteru, kao i ona koja su testirala parove kod kojih je razlika u zameni slova č slovom c. Greške su bile u odsecanju poslednjeg karaktera i enkodovanju ulaznog parametra kao ascii niske umesto kao UTF8.

4.4 Automatizovani testovi za WCF servis

WCF servis za autentifikaciju korisnika je spoljašnim aplikacijama i komponentama predstavljen preko *UserOperations* interfejsa. On pruža operacije potrebne za kreiranje korisnika i prijavljivanje korisnika na sistem. Za sve te operacije on koristi *UserManipulator* i *SessionManipulator* klase za komunikaciju sa bazom podataka. Relevantni deo interfejsa je prikazan ispod:

```
[ServiceContract]
    public interface IAuthentication
    {
        [OperationContract]
        OperationResult CreateNewUser(UserDataLight userDataLight, IDataContextWrapper
dataContextWrapper);

        [OperationContract]
        OperationResult LoginUser(string username, string password, IDataContextWrapper
dataContextWrapper);
    }
```

Za svaku od prikazanih servisnih operacija ćemo napraviti set automatizovanih testova. To će biti funkcionalni testovi, jer će proveravati da li su zadovoljeni funkcionalni zahtevi operacije. S obzirom na to da će se njihovim izvršavanjem testirati i funkcionalnosti UserManipulator i Hasher klase, kao i baze podataka, ti testovi će biti integracionog tipa. Testovi će koristiti testnu i lažnu bazu podataka, u koje možemo da unesemo podatke koji nam odgovaraju.

4.4.1 Testiranje operacije za kreiranje novog korisnika

Operacija *CreateNewUser()* ima povratnu vrednost tipa *OperationResult.* Klasa izgleda ovako:

```
    [DataContract]
    public class OperationResult
    {
        [DataMember]
        public bool success;

        [DataMember]
        public string message;
    }
```

Atributi DataContract i DataMember se koriste zbog serijalizacije. Oni obaveštavaju WCF servis o tome kako treba povratna poruka da se serijalizuje. Polje *success* govori da li je operacija uspešno izvršena. Polje *message* nosi odgovarajuću poruku. U ovom radu poruka će biti relevantna samo u

slučaju kada se operacija kreiranja korisnika nije uspešno izvršila. U tabeli prikazanoj na slici 4-4 su očekivane vrednosti za ova dva parametra prikazane u kolonama *success* i *message*.

Ovim testovima želimo da testiramo što veći broj različitih mogućih scenarija za kreiranje novog korisnika. Pored validnih i nevalidnih ulaznih parametara, s obzirom da su u pitanju integracioni testovi koji u svom izvršavanju uključuju i komunikaciju sa bazom, uključićemo i testove koji u obzir uzimaju i stanje u bazi. Koristićemo metodologiju testiranja vođenog podacima. U excel datoteci se nalaze podaci za testiranje. Prikaz datoteke je na slici 4-4.

Username	email	success	message
TestUsername1	testemail1@microsoft.com	TRUE	
TestUsernameDuplicate	testemail6@microsoft.com	FALSE	User name is already registered.
TestUsername	testemailDuplicate@microsoft.com	FALSE	User with email provided is already registered.
user name with white spaces	testemail3@microsoft.com	FALSE	User name cannot contain spaces.
Testusernamewhichcontainsfiftycharactersprecisely	testemail5@microsoft.com	TRUE	
Testusernamewhichcontainsfifty1charactersprecisely	testemail7@microsoft.com	FALSE	User name cannot have more than 50 characters.

Slika 4-4 Testni podaci za servisnu operaciju kreiranja korisnika

U bazi podataka se već nalaze redovi koji odgovaraju korisničkom imenu TestUserNameDuplicate, kao i adresi testemailDuplicate@microsoft.com. Zbog jednostavnosti, korisnička lozinka će svaki put biti ista i njena vrednost će biti validna.

Kôd testa je dat ispod:

```
[TestMethod()]
[DeploymentItem("App.config")]
[DataSource("BasicUserData")]
public void RegisterVariousUsersTest()
{
    string username = TestContext.DataRow["Username"].ToString();
    string email = TestContext.DataRow["email"].ToString();
    bool expectedSuccess = Convert.ToBoolean(TestContext.DataRow["success"]);
    string expectedMessage = TestContext.DataRow["message"].ToString();
    UserDataLight udl = new UserDataLight()
    {
        Username = username,
        Password = validPassword,
        Email = email
    };
    OperationResult or = InsertUser(udl);
    Assert.AreEqual(expectedSuccess, or.success);
    Assert.AreEqual(expectedMessage, or.message);
}
```

Metoda *InsertUser* je zadužena za izvršavanje same servisne operacije. Testiranje vođeno podacima u okruženju Microsoft Visual Studio Test Suite nam omogućava da napišemo samo jedan automatizovani test, koji će se izvršiti onoliko puta koliko ima redova u izvoru podataka, svaki put

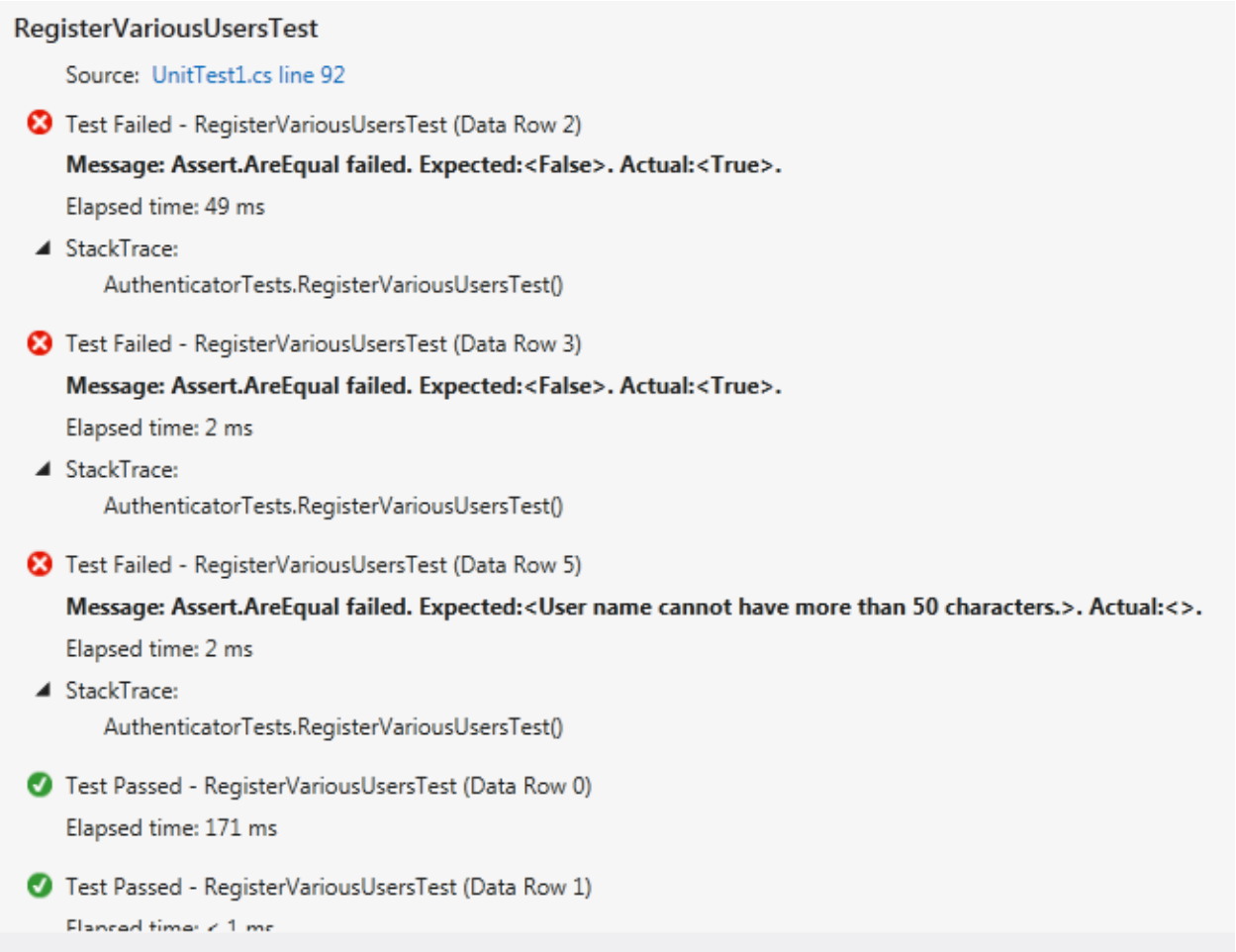
novim setom ulaznih podataka. Ovakav test će biti označen kao uspešno izvršen tek kad se uspešno izvrši za svaki set podataka iz predviđenog izvora podataka.

Pri prvom izvršavanju, test je označen kao neuspešan. Detalji su prikazani na slici 4-5. Analizom rezultata su ustanovljena tri problema sa servisnom operacijom za kreiranje korisnika.

Prvi problem vidljiv iz rezultata je taj što sistem dozvoljava da se više korisnika registruje sa istom e-mail adresom. Kako je baza podataka nepromenjiva u ovom scenariju, promenjen je kôd tako da, pre kreiranja novog reda u tabeli, vrši odgovarajuću proveru.

Drugi problem je nedostatak provere postojanja praznih karaktera u niski karaktera koja je uneta kao korisničko ime. Problem je rešen, kao i u prethodnom slučaju, u kodu aplikacije.

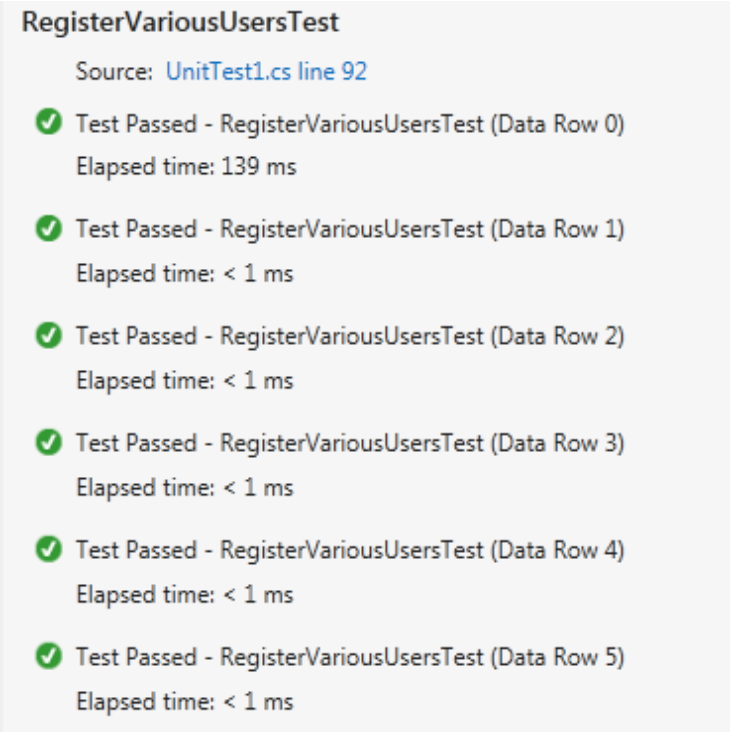
Treći problem drugačijeg karaktera. Pri pokušaju unosa korisničkog imena dužine veće od predviđene, sistem jeste zabranio unos, ali je poruka o grešci bila neodgovarajuća. Analizom koda je utvrđeno da sama aplikacija ne proverava dužinu unetog korisničkog imena. Unos je sprečen jer odgovarajuće ograničenje postoji u bazi podataka. Pošto želimo da servis, u slučaju neuspeha, vrati odgovarajuću poruku, nedostatak je ispravljen uvođenjem dodatne provere.



Slika 4-5 Detalji neuspešno izvršenog testa *RegisterVariousUsersTest*

Nakon ovih ispravki, ponovo je izvršen *RegisterVariousUsersTest* test. Ovog puta, označen je kao uspešan, što znači da se uspešno izvršio za sve ulazne podatke. Na slici 4-6 je prikazan rezultat.

Sa slike 4-6 možemo uočiti i detalje vezane za dužinu izvršavanja testa. Primetno je da se, za prvi set podataka, test izvršava znatno duže nego za ostale setove podataka, 139ms u odnosu na < 1 ms. Razlog ovakvog ponašanja je taj što se pri učitavanju prvog seta podataka otvara konekcija ka excel datoteci koja se koristi kao izvor. Za čitanje svih ostalih redova se koristi već postojeća konekcija.



Slika 4-6 Detalji uspešno izvršenog testa RegisterVariousUsersTest

4.4.2 Testiranje operacije za prijavljivanje korisnika

Operacija za prijavljivanje korisnika kao parametre prima korisničko ime i lozinku korisnika. U koliko su oni ispravni, tj. pripadaju postojećem korisniku, za tog korisnika se kreira sesija u bazi. Podrazumevano vreme trajanje sesije je 30 minuta, a podešava se kroz konfiguracioni fajl. Testovi za ovu metodu će koristiti lažni kontekst podataka, u kom ćemo proveravati postojanje sesije i njenu ispravnost.

Scenarija koja želimo da testiramo su:

- Pri prijavljivanju postojećeg korisnika sa ispravnom lozinkom, za kojeg ne postoji aktivna sesija, kreira se sesija koja mu odgovara, a koja ističe 30 minuta nakon kreiranja
- Pri prijavljivanju nepostojećeg korisnika, ne kreira se sesija i dobija se odgovarajuća poruka o grešci
- Pri prijavljivanju postojećeg korisnika, ali sa pogrešnom lozinkom, ne kreira se sesija i dobija se odgovarajuća poruka o grešci.
- Pri prijavljivanju postojećeg korisnika, za kojeg postoji aktivna sesija, ne kreira se nova sesija i dobija se odgovarajuća poruka o grešci.

Svaki od ovih scenarija validira po dve funkcionalnosti: stanje u kontekstu podataka i odgovarajuću poruku. Dobra praksa je da funkcionalni testovi validiraju po tačno jednu funkcionalnost. Ovakvi testovi su jednostavniji za održavanje i za analizu rezultata. Kada se neki test neuspešno izvrši, ako

on validira tačno jednu funkcionalnost, lakše se lociraju greške. Sledeći ovu praksu, za četiri gore navedena scenarija, napisaćemo osam automatizovanih testova. Oni će biti smešteni u zasebnu klasu testova.
Potpisi testnih metoda su sledeći:

```
[TestMethod]
public void LoginValidParamsNoActiveSession_CheckMessage()
[TestMethod]
public void LoginValidParamsNoActiveSession_CheckSession()
[TestMethod]
public void LoginInvalidParams_CheckMessage()
[TestMethod]
public void LoginInvalidParams_CheckSession()
[TestMethod]
public void LoginInvalidPassword_CheckMessage()
[TestMethod]
public void LoginInvalidPassword_CheckSession()
[TestMethod]
public void LoginValidParamsActiveSession_CheckMessage()
[TestMethod]
public void LoginValidParamsActiveSession_CheckSession()
```

Podaci koji se koriste u testovima, kao i lažni kontekst podataka, definisani su na nivou klase testova. Oni se inicijalizuju na željene vrednosti u metodi podešavanja. Podešavanje se izvršava jednom za celu klasu testova. U metodi čišćenja, koja se izvršava kada se izvrše svi testovi iz klase, uništava se kreirani kontekst podataka. Kôd za testove koji se odnose na prijavljivanje korisnika a ispravnim parametrima, za kojeg ne postoji aktivna sesija, prikazan je ispod. Ostale testne metode su napisane po uzoru na njih.

```
public void LoginValidParamsNoActiveSession_CheckMessage()
{
    OperationResult result = service.LoginUser(existingUserNoSessionUsername,
                                               existingUserNoSessionPassword,
                                               dataContext);

    Assert.AreEqual(true, result.success);
    Assert.AreEqual("User successfully loged in.", result.message);
}

public void LoginValidParamsNoActiveSession_CheckSession()
{
    OperationResult result = service.LoginUser(existingUserNoSessionUsername,
                                               existingUserNoSessionPassword,
                                               dataContext);
    SessionData sd = GetSessionDataForUser(dataContext,
                                    existingUserNoSessionUsername,
                                    existingUserNoSessionPassword);

    DateTime expectedDateExpires = DateTime.Now.AddMinutes(30);
    int expectedHoursExpires = expectedDateExpires.Hour;
    int expectedMinutesExpires = expectedDateExpires.Minute;
```

```
            Assert.AreNotEqual(null, sd);
            Assert.AreEqual(expectedDateExpires.Date, sd.DatetimeExpires.Date);
            Assert.AreEqual(expectedHoursExpires, sd.DatetimeExpires.Hour);
            Assert.AreEqual(expectedMinutesExpires, sd.DatetimeExpires.Minute);
        }
```

Prva test metoda proverava poruku koju vraća servisna operacija. Druga test metoda proverava da li se u kontekstu podataka nalaze očekivane vrednosti za datog korisnika. Pomoćna metoda GetSessionDataForUser se koristi za dobijanje podataka iz SessionData tabele zadatog konteksta podataka, za zadatog korisnika.

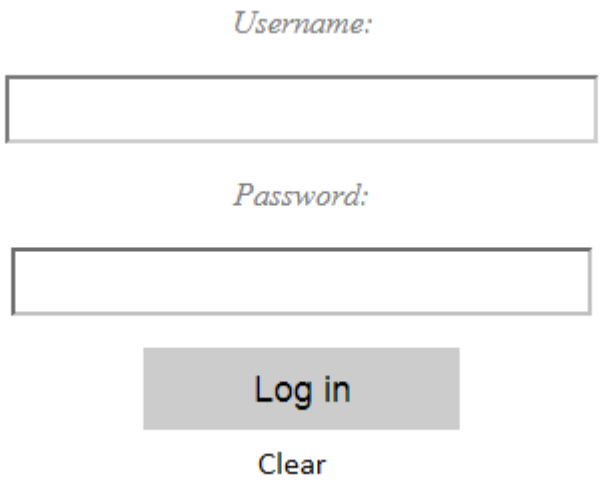
Izvršavanjem svih testova napisanih za servisnu operaciju prijavljivanja korisnika na sistem, otkrivene su sledeće greške:

- Pri prijavljivanju postojećeg korisnika sa pogrešnom lozinkom, kreira se sesija, ali se dobija i poruka o grešci
- Pri prijavljivanju korisnika za kog već postoji sesija, kreira se nova sesija i ne dobija se odgovarajuća poruka o grešci
- Pri prijavljivanju nepostojećeg korisnika, ne dobija se odgovarajuća poruka o grešci.

4.5 Sistemski automatizovani testovi

Kada je automatizovan željeni broj testova za pojedinačne delove sistema, red je došao na sistemsko testiranje, odnosno testiranje sistema kao celine. Ovde ćemo se fokusirati na deo sistema koji se odnosi na prijavljivanje korisnika na sistem. Grafički korisnički interfejs, koji omogućava prijavljivanje korisnika na sistem, je prikazan na slici 4-7. Testovi opisani u ovom poglavlju će se razlikovati od opisanih u prethodnim poglavljima jer uključuju interakciju sa grafičkim korisničkim interfejsom.

Deo sistema za prijavljivanje korisnika na sistem se može pojednostavljeno predstaviti modelom prikazanim na slici 4-8. Ovo nije kompletan model jer je svakako moguće izvesti još različitih stanja i operacija, ali je dovoljan. Stanja sistema su predstavljena krugovima, akcije koje omogućavaju prelazak iz jednog u drugo stanje strelicama. Kako bismo bili sigurni da su pokrivene sve putanje kroz model, odnosno sve moguće kombinacije stanja i akcija predstavljene ovim modelom, primenićemo testiranje zasnovano na modelima, opisano u poglavlju 3.2.



Slika 4-7 Grafički korisnički interfejs koji omogućava prijavljivanje na sistem

Za Visual Studio postoji dodatak koji se zove Spec Explorer i koji omogućava implementaciju testiranja zasnovanog na modelima. Ovaj dodatak je, na osnovu modela sa slike 4-8, kreirao sve moguće putanje kroz model, korišćenjem nekog od algoritma za pretragu grafova. Dodato je ograničenje da korisnik ne može više od 10 puta da pokuša da se prijavi na sistem, posle desetog pokušaja nema više mogućnost da se prijavi. Kreirane su prazne metode koje odgovaraju akcijama, kao i prazne metode koje odgovaraju stanjima. Na primer, kreirane su metode:

```
public void tst_validateLoginPageIsShown()
        {

        }

public void action_enterValidCredentials()
        {

        }
```

Metoda tst_validateLoginPageIsShown() odgovara stanju „Strana za prijavljivanje“, i pošto stanja uključuju validaciju, metoda počinje prefiksom tst_.

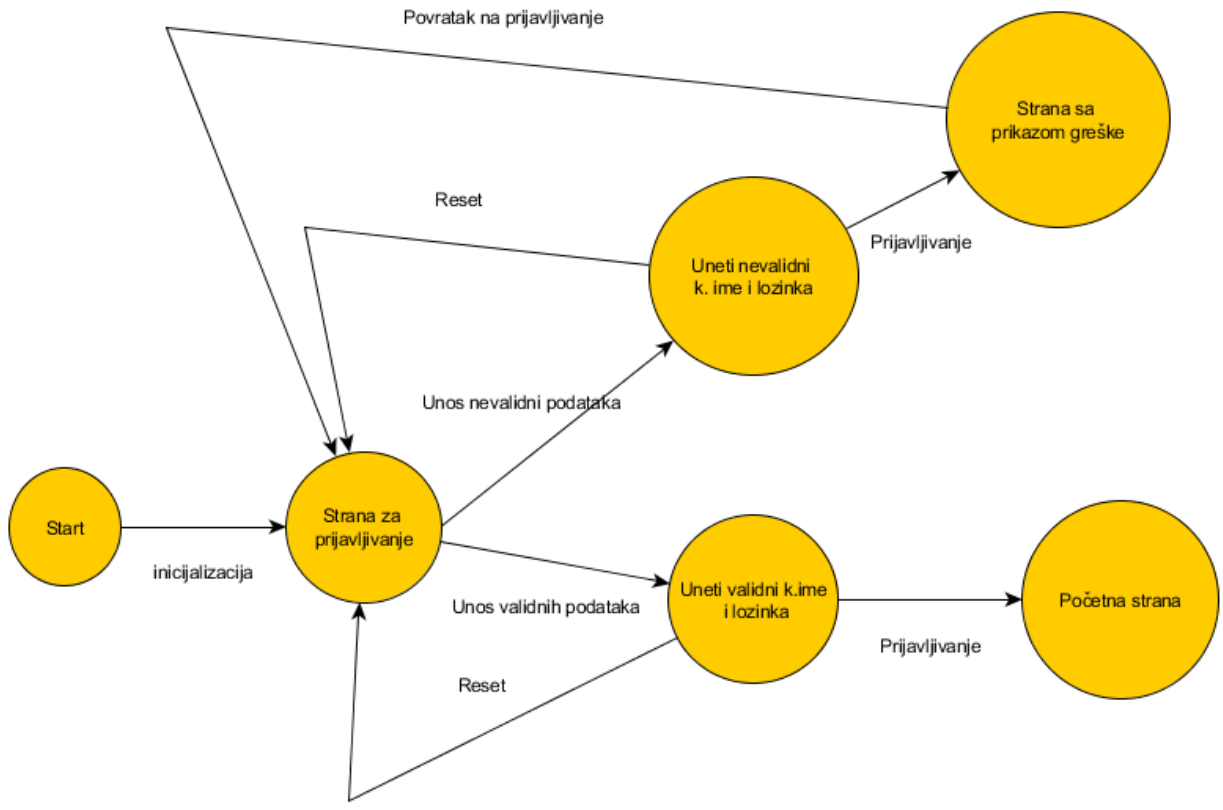
Metoda action_enterValidCredentials() odgovara akciji „Unos validnih podataka“. Kao i sve metode koje odgovaraju akcijama, sistem je kreira sa prefiksom action_.
Nad njima je kreiran skup testova koji se sastoji iz različitih kombinacija ovih metoda. Taj skup testova pokriva sve prolaske kroz model, odnosno sve slučajeve korišćenja sistema zadatog tim modelom.

Jedan od kreiranih testova izgleda ovako:

```
[TestMethod]
public static void TestMethod1()
{
    action_initialize();
    tst_validateHomePageIsShown();
    action_enterInvalidCredentials();
    action_activateResetButton();
    tst_validateHomePageIsShown();
}
```

Ovaj test predstavlja scenario u kojem korisnik otvara stranicu za prijavljivanje na sistem, unosi nevalidne podatke, resetuje formu i ponovo mu se prikazuje prazna forma za prijavljivanje.

Metode koje odgovaraju akcijama i stanjima ćemo sami implementirati, koristeći CodedUI tests dodatak za Visual Studio.



Slika 4-8 Model dela sistema koji predstavlja prijavljivanje korisnika na sistem

4.5.1 Automatizacija testova koja koristi grafički korisnički interfejs

CodedUITests dodatak je alat koji omogućava pisanje naprednih *snimi i reprodukuj* testova. On koristi preslikavanje grafičkog korisničkog interfejsa i identifikovanje njegovih elementa po njihovim identifikatorima, umesto po njihovim fiksnim pozicijama na ekranu. Ovaj alat omogućava da se pojavljivanje elemenata korisničkog interfejsa čeka od isteka zadatog vremenakog intervala, tako da na isvršavanje testova ne utiče brzina sistema za testiranje. Takođe, moguće je snimanje akcija i validacija u zasebne metode, tako da se dobija fleksibilan kôd. Jednom snimljene akcije i validacije mogu da se koriste više puta.

Najpre smo snimili metode koje validiraju stanja u modelu. One validiraju da je prikazana odgovarajuća web stranica, odnosno da prikazana stranica sadrži odgovarajuće elemente. Zatim smo snimili metode koje odgovaraju akcijama. Sve ove metode smo implementirali u praznim metodama generisanim testiranjem zasnovanim na modelu. Najveći deo implementacije je automatski kreiran tokom snimanja. Kôd jedne od implementiranih metoda je ispod.

```
public void ValidateUserNameFields()
        {
            #region Variable Declarations
            HtmlSpan uIUsernamePane = 
this.UIHttplocalhost53088LoWindow.UIHttplocalhost53088LoDocument.UIUsernamePane;
            HtmlEdit uITextBox1Edit = 
this.UIHttplocalhost53088LoWindow.UIHttplocalhost53088LoDocument.UITextBox1Edit;
            #endregion

            // Verify that the 'Id' property of 'Username:' pane equals 'Label1'
            Assert.AreEqual(this.ValidateUserNameFieldsExpectedValues.UIUsernamePaneId, 
uIUsernamePane.Id, "Label1 not present");

            // Verify that the 'InnerText' property of 'Username:' pane equals 'Username:
'

Assert.AreEqual(this.ValidateUserNameFieldsExpectedValues.UIUsernamePaneInnerText, 
uIUsernamePane.InnerText, "Username text not correct");

            // Verify that the 'Name' property of 'TextBox1' text box equals 'TextBox1'
            Assert.AreEqual(this.ValidateUserNameFieldsExpectedValues.UITextBox1EditName, 
uITextBox1Edit.Name, "UserName textBox not present.");

            // Verify that the 'Text' property of 'TextBox1' text box equals ''
            Assert.AreEqual(this.ValidateUserNameFieldsExpectedValues.UITextBox1EditText, 
uITextBox1Edit.Text, "Username textbox not empty");
        }
```

Ova metoda proverava da li se na strani nalaze očekivani elementi i da oni imaju očekivane osobine.

Izvršavanjem svih testova pronađeno je nekoliko manjkavosti grafičkog korisničkog interfejsa. Jedna od njih je ta da polje za unos korisničkog imena nema ograničenje na dužinu unetog teksta od 50 karaktera, kao što bi bilo očekivano s obzirom da je to maksimalna dužina korisničkog imena u sistemu. Ovi testovi su kasnije korišćeni i kao regresioni testovi. Pri naknadnim izmenama sistema, odnosno pri dodavanju novih funkcionalnosti i refaktorisanju postojećeg koda, uspešno

izvršavanje ovih sistemskih testova je garantovalo da nisu nastale nove greške u osnovnim funkcionalnostima sistema.

5 Zaključak

Automatizovano testiranje je, danas, nezaobilazan deo svakog softverskog sistema. Ono nam pruža izvesne garancije da softver, koji pravimo, ispunjava zahteve sa stanovišta ispravnosti, kvaliteta, performansi, sigurnosti. Postoji veliki broj različitih metoda testiranja, nivoa testiranja, kao i alata i biblioteka koje mogu da nam pomognu u testiranju. U zavisnosti od samog sistema, kao i od raspoloživih resursa, bitno je odabrati odgovarajući pravac. U ovom radu su prikazane neke od metodologija automatizovanog testiranja i neki nivoi automatizovanih testova. Na konkretnim primerima je pokazano kakve osobine treba da ima kôd sistema za testiranje, kao i koliko automatizovani testovi smanjuju rizik od takozvanih regresionih grešaka koje se javljaju pri refaktorisanju koda ili pri dodavanju novih funkcionalnosti u postojeći sistem. Iako je, inicijalno, cena uvođenja automatizovanog testiranja velika, praksa pokazuje da je povratak investicije (eng. *Return Of Investment*), kada se automatizacija izvede pravilno, značajan. Najveća ušteda se dobija primenom automatizacije na velike sisteme, naročito kada je u njihov razvoj uključen veliki broj programera. Dugački testni slučajevi, sa puno različitih ulaznih parametara, kada se izvode manualno, zahtevaju mnogo vremena. Kao i svaki manualni rad, podložni su greškama koje spadaju u takozvani „ljudski faktor". Postoje i neki testovi koje je jako teško, nekad čak i nemoguće, izvesti manualno. U ove testove spadaju, na primer, testovi performansi i testovi opterećenja. Automatizovani testovi omogućavaju da se testni slučajevi izvode više puta na identičan način. Vreme uloženo u automatizaciju testova se vraća kroz vreme potrebno za izvršavanje testova, bolji kvalitet sistema za testiranje i merljivost tog kvaliteta.

Slike

Bibliografija

1. Myers, G.J., *The Art of Software Testing.* 1979.
2. DeMillo, R.A. and R.J. Lipton, *A Probabilistic Remark on Algebraic Program Testing.* 1978.
3. Burnstein, I., *Practical Software Testing: A Process-Oriented Approach.* 2003.
4. Dustin, E., *Automated Software Testing.* 1999.
5. Craig, R.D. and S.P. Jaskiel, *Systematic Software Testing.* 2002.
6. Kolawa, A. and D. Huizinga, *Automated Defect Prevention: Best Practices in Software Management.* 2007.
7. Kaner, C., *Improving the Maintainability of Automated Test Suites.* 1997.
8. Kent, J., *Overcoming the Problems of Automated GUI Testing.* 1994.
9. Nagle, C., *Test Automation Frameworks.* 2000.
10. Fewster, M. and D. Graham, *Software Test Automation.* 1999.
11. Kaner, C., J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach.* 2001.
12. Utting, M. and B. Legeard, *Practical Model-Based Testing:A Tools Approach.* 2004.
13. Chow, T.S., *Testing Software Design Modeled by Finite-State Machines.* 1978
14. Offutt, J. and A. Abdurazik, *Generating Tests from UML Specifications* 1999.
15. Briand, L.C., Y. Labiche, and Q. Lin, *Improving Statechart Testing Criteria Using Data Flow Information.* 2005.
16. Clarke, E. and A. Emerson, *Model Checking: Algorithmic Verification and Debugging.* 1981.
17. Dustin, E., T. Garrett, and B. Gauf, *Implementing Automated Software Testing.* 2009.
18. Kan, S.H., *Metrics and Models in Software Quality Engineering.* 2002
19. Hayes, L.G., *Automated Testing Handbook.* 2004
20. Pettichord, B., *Design For Testability*, 2002.
21. Pathak, N., Pro WCF 4: Practical Microsoft SOA Implementation, 2011
22. Godefroid N., Levin M, Molnar D., Automated Whitebox Fuzz Testing, 2008