DEO I

Primer valuta

U Delu I, razvijaćemo tipičan model koda potpuno vođen testovima (osim kada pogrešimo, isključivo u obrazovne svrhe). Moj cilj je da vidite ritam razvoja vođenog testiranjem (Test Driven Development, TDD), koji se može sumirati na sledeći način.

1. Brzo dodajte test.
2. Pokrenite sve testove i vidite kako novi zakazuje.
3. Napravite malu promenu.
4. Pokrenite sve testove i vidite kako svi uspevaju.
5. Refaktorišite da biste uklonili dupliranje.

Iznenađenja će verovatno uključivati

- Kako svaki test može pokriti mali inkrement funkcionalnosti
- Koliko male i ružne promene mogu biti da bi se novi testovi pokrenuli
- Koliko često se testovi pokreću
- Koliko sitnih koraka čine refaktorisanja

Poglavlje 1

Viševalutni novac

Počećemo sa objektom koji je Ward kreirao u WyCashu, viševalutnim novcem (pogledajte Uvod). Pretpostavimo da imamo ovakav izveštaj:

Instrument	Akcije	Cena	Ukupno
IBM	1000	25	25000
GE	400	100	40000
		Ukupno	65000

Da bismo napravili viševalutni izveštaj, moramo dodati valute:

Instrument	Akcije	Cena	Ukupno
IBM	1000	25 USD	25000 USD
Novartis	400	150 CHF	60000 CHF
		Ukupno	65000 USD

Takođe moramo navesti kurseve:

Iz	U	Kurs
CHF	USD	1.5

$5 + 10 CHF = $10 if rate is 2:1
$5 * 2 = $10

Koje ponašanje će nam biti potrebno da bismo proizveli revidirani izveštaj? Drugim rečima, koji skup testova, kada budu prošli, će demonstrirati prisustvo koda za koji smo sigurni da će tačno izračunati izveštaj?

- Moramo biti u stanju da sabiramo iznose u dve različite valute i konvertujemo rezultat s obzirom na skup kurseva.
- Moramo biti u stanju da pomnožimo iznos (cenu po akciji) sa brojem (brojem akcija) i dobijemo iznos.

Napravićemo listu zadataka da nas podseti šta treba da uradimo, da nas drži fokusiranima i da nam kaže kada smo završili. Kada počnemo da radimo na stavki, učinićemo je podebljanom, **kao ovo**. Kada završimo stavku, precrtaćemo je, ~~kao ovo~~. Kada pomislimo na još jedan test za pisanje, dodaćemo ga na listu.

Kao što možete videti iz liste zadataka na prethodnoj stranici, prvo ćemo raditi na množenju. Dakle, koji objekat nam je prvo potreban? Zagonetno pitanje. Ne počinjemo sa objektima, počinjemo sa testovima. (Stalno moram da se podsećam na ovo, pa ću se pretvarati da ste jednako spori kao ja.)

Pokušaj ponovo. Koji test nam je prvo potreban? Gledajući listu, taj prvi test izgleda komplikovano. Počnite od malih stvari, ne počinjite nešto drugo. Množenje, da li bi to moglo biti teško? Prvo ćemo raditi na tome.

Kada pišemo test, zamišljamo savršen interfejs za našu operaciju. Pričamo sebi priču o tome kako će operacija izgledati spolja. Naša priča se neće uvek ostvariti, ali bolje je početi od najboljeg mogućeg API-ja (Application Program Interface) i raditi unazad, nego stvari učiniti komplikovanim, ružnim i "realističnim" od samog početka.

Evo jednostavnog primera množenja:

```
public void testMultiplication() {
   Dollar five= new Dollar(5);
   five.times(2);
   assertEquals(10, five.amount);
}
```

(Znam, znam, javna polja, sporedni efekti, celi brojevi za novčane iznose i sve to. Mali koraci. Zabeležićemo sve to i ići dalje. Imamo test koji pada, i želimo da se traka što pre pozeleni.)

$5 + 10 CHF = $10 if rate is 2:1
$5 * 2 = $10
Neka "amount" bude privatno
Sporedni efekti dolara?
Zaokruživanje novca?

Test koji smo upravo napisali čak se ni ne kompajlira. (Objasniću gde i kako ga pišemo kasnije, kada budemo više govorili o testnom okviru, JUnitu.) To je lako za popraviti. Šta je najmanje što možemo da uradimo da se kompajlira, čak i ako ne radi? Imamo četiri greške pri kompajliranju:

- Nema klase `Dollar`
- Nema konstruktora
- Nema metode `times(int)`
- Nema polja `amount`

Hajde da ih uzmemo jednu po jednu. (Uvek tražim neku numeričku meru napretka.) Možemo se rešiti jedne greške definisanjem klase `Dollar`:

Dollar
```
class Dollar
```

Jedna greška manje, tri greške preostale. Sada nam je potreban konstruktor, ali ne mora ništa da radi samo da bi se test kompajlirao:

Dollar
```
Dollar(int amount) {
}
```

Dve greške preostale. Potrebna nam je "stub" implementacija `times()`. Opet ćemo uraditi najmanje moguće posla samo da bi se test kompajlirao:

Dollar
```
void times(int multiplier) {
}
```

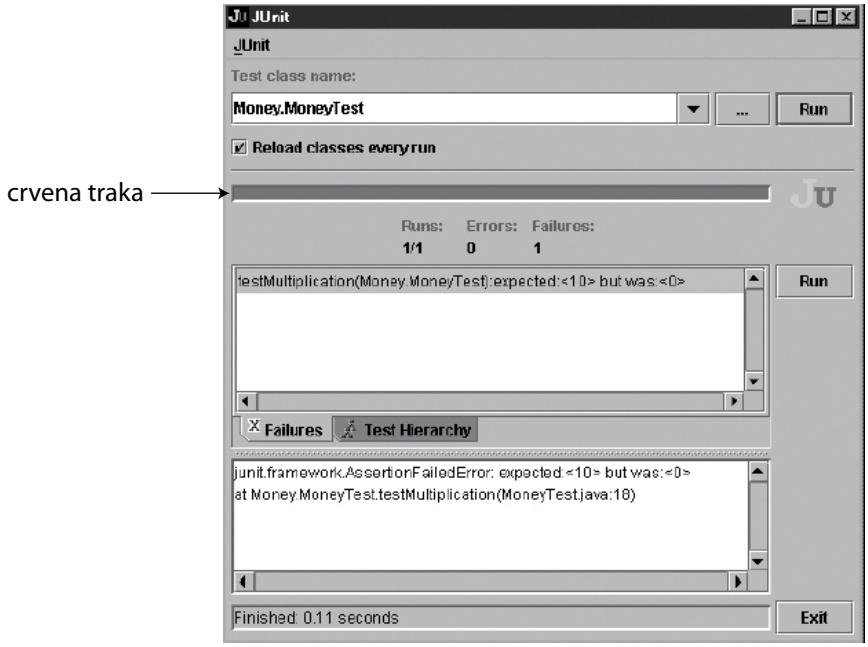
Svedeno na jednu grešku. Konačno, potrebno nam je polje `amount`:

Dollar
```
int amount;
```

To je to! Sada možemo pokrenuti test i gledati kako zakazuje, kao što je prikazano na Slici 1.1.

Vidite ozloglašenu crvenu traku. Naš testni okvir (JUnit, u ovom slučaju) je pokrenuo mali isečak koda sa kojim smo počeli i primetio da smo, iako smo očekivali "10" kao rezultat, umesto toga videli "0". Grr.

Ne, ne. Neuspeh je napredak. Sada imamo konkretnu meru neuspeha. To je bolje nego samo nejasno znati da ne uspevamo. Naš programski problem je transformisan iz "dajte mi više valuta" u "učinite da ovaj test radi, a zatim učinite da i ostali testovi rade." Mnogo jednostavnije. Mnogo manji opseg za strah. Možemo učiniti da ovaj test radi.



Slika 1.1 *Napredak! Test zakazuje*

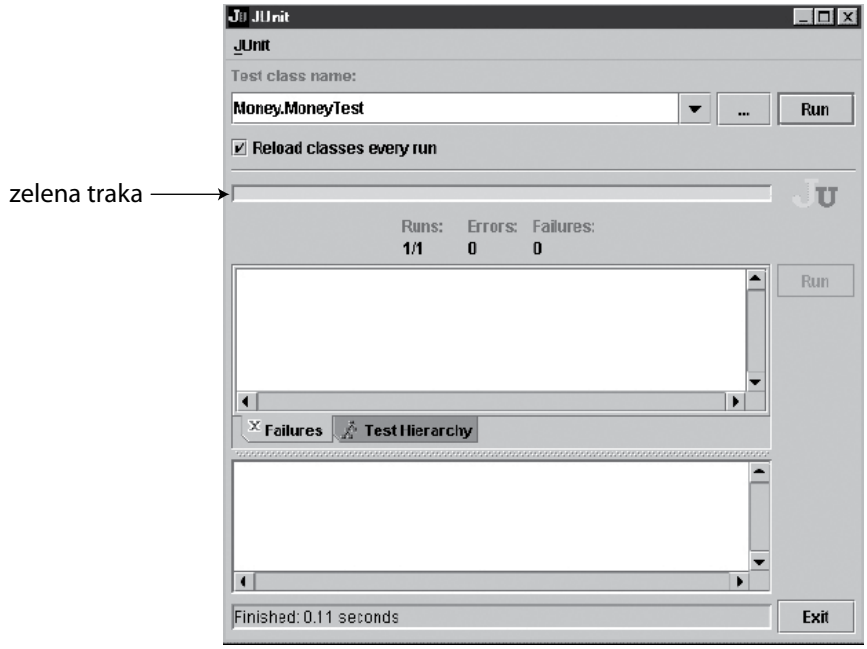
Verovatno vam se neće svideti rešenje, ali cilj sada nije da se dobije savršen odgovor, već da se prođe test. Žrtvovaćemo se na oltaru istine i lepote kasnije.

Evo najmanje promene koju sam mogao da zamislim koja bi prouzrokovala da naš test prođe:

Dollar

```
int amount= 10;
```

Slika 1.2 prikazuje rezultat kada se test ponovo pokrene. Sada dobijamo zelenu traku, opevanu u pesmama i pričama



Slika 1.2 *Test se pokreće*

Sreća, sreća, radost! Ne tako brzo, hakeru (ili hakerko). Ciklus nije završen. Vrlo je malo unosa u stvarnom svetu koji će prouzrokovati da tako ograničena, tako naivna implementacija prođe. Moramo generalizovati pre nego što krenemo dalje. Zapamtite, ciklus je sledeći.

1. Dodajte mali test.
2. Pokrenite sve testove i pad.
3. Napravite malu promenu.
4. Pokrenite testove i uspite.
5. Refaktorišite da biste uklonili dupliranje.

Zavisnost i dupliranje

Steve Freeman je istakao da problem sa testom i kodom nije dupliranje(na koje vam još nisam ukazao, ali obećavam da hoću čim ova digresija bude gotova). Problem je zavisnost između koda i testa – ne možete promeniti jedno, a da ne promenite drugo. Naš cilj je da budemo u mogućnosti da napišemo još jedan test koji nam "ima smisla", bez potrebe da menjamo kod, nešto što nije moguće sa trenutnom implementacijom.

Zavisnost je ključni problem u razvoju softvera na svim skalama. Ako imate detalje implementacije SQL-a jednog dobavljača razbacane po kodu i odlučite da pređete na drugog dobavljača, onda ćete otkriti da je vaš kod zavisan od dobavljača baze podataka. Ne možete promeniti bazu podataka, a da ne promenite kod.

Ako je zavisnost problem, dupliranje je simptom. Dupliranje najčešće poprima oblik duplirane logike – isti izraz se pojavljuje na više mesta u kodu. Objekti su izvrsni za apstrahovanje dupliranja logike.

Za razliku od većine problema u životu, gde eliminisanje simptoma samo prouzrokuje da se problem pojavi negde drugde u gorem obliku, eliminisanje dupliranja u programima eliminiše zavisnost. Zato se drugo pravilo pojavljuje u TDD-u. Eliminacijom dupliranja pre nego što pređemo na sledeći test, maksimiziramo šansu da sledeći test proradi sa jednom i samo jednom promenom.

Prošli smo stavke od 1 do 4. Sada smo spremni da uklonimo dupliranje. Ali gde je dupliranje? Obično vidite dupliranje između dva dela koda, ali ovde je dupliranje između podataka u testu i podataka u kodu. Ne vidite? Šta ako napišemo sledeće:

Dollar
```
int amount= 5 * 2;
```

Ta `10` je moralo odnekud doći. Množenje smo uradili u glavi tako brzo da nismo ni primetili. `5` i `2` su sada na dva mesta, i moramo nemilosrdno eliminisati dupliranje pre nego što nastavimo. Pravila tako kažu.

Ne postoji nijedan korak koji će eliminisati `5` i `2`. Ali šta ako prebacimo postavljanje iznosa iz inicijalizacije objekta u metodu `times()`?

Dollar
```
int amount;

void times(int multiplier) {
   amount= 5 * 2;
}
```

Test i dalje prolazi, traka ostaje zelena. Sreća je i dalje na našoj strani.

Čine li vam se ovi koraci premali? Zapamtite, TDD se ne odnosi na preduzimanje sićušnih koraka, već na *mogućnost* preduzimanja sićušnih koraka. Da li bih svakodnevno kodirao sa ovako malim koracima? Ne. Ali kada stvari postanu iole uvrnute, drago mi je što mogu. Pokušajte sa sićušnim koracima na primeru po vašem izboru. Ako možete napraviti korake preterano male, sigurno možete napraviti i

korake prave veličine. Ako preduzimate samo veće korake, nikada nećete znati da li su manji koraci odgovarajući.

Odbranu na stranu, gde smo stali? Rešavali smo se dupliranja između test koda i radnog koda. Odakle možemo dobiti 5? To je bila vrednost prosleđena konstruktoru, pa ako je sačuvamo u varijabli amount,

Dollar
```
Dollar(int amount) {
   this.amount= amount;
}
```

onda je možemo koristiti u times():

Dollar
```
void times(int multiplier) {
   amount= amount * 2;
}
```

Vrednost parametra "multiplier" je 2, tako da možemo zameniti konstantu sa parametrom:

Dollar
```
void times(int multiplier) {
   amount= amount * multiplier;
}
```

Da bismo demonstrirali naše temeljno poznavanje Java sintakse, koristićemo operator *= (koji, mora se reći, smanjuje dupliranje):

Dollar
```
void times(int multiplier) {
   amount *= multiplier;
}
```

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Neka "amount" bude privatno
Sporedni efekti dolara?
Zaokruživanje valute?

Sada možemo označiti prvi test kao završen. Zatim ćemo se pozabaviti tim čudnim sporednim efektima. Ali prvo, hajde da ponovimo. Uradili smo sledeće:

- Napravili smo listu testova za koje smo znali da treba da rade
- Ispričali smo priču sa isečkom koda o tome kako smo želeli da vidimo jednu operaciju
- Ignorisali smo detalje JUnit-a za sada

- Učinili smo da se test kompajlira sa “stubovima”
- Učinili smo da test radi čineći užasne greške
- Postepeno smo generalizovali radni kod, zamenjujući konstante promenljivama
- Dodavali smo stavke na našu listu zadataka umesto da ih sve rešavamo odjednom