Predgovor

Čist kod koji radi, prema jezgrovitoj frazi Rona Jeffriesa, cilj je razvoja vođenog testovima (Test-Driven Development, TDD). Čist, ili jasan, kod koji radi je vredan cilj iz mnogo razloga.

- To je predvidljiv način razvoja. Znate kada ste završili, bez brige o dugom tragu grešaka.
- Daje vam priliku da naučite sve lekcije koje kod ima da vas nauči. Ako samo sklepate prvu stvar koja vam padne na pamet, onda nikada nemate vremena da smislite drugu, bolju stvar.
- Poboljšava živote korisnika vašeg softvera.
- Omogućava vašim saradnicima da računaju na vas, a vi na njih.
- Dobar je osećaj pisati ga.

Ali kako dolazimo do čistog koda koji radi? Mnoge sile nas udaljavaju od čistog koda, pa čak i od koda koji radi. Bez previše savetovanja sa našim strahovima, evo šta radimo: vodimo razvoj automatizovanim testovima, stilom razvoja koji se naziva razvoj vođen testovima. U razvoju vođenom testovima mi

- Pišemo novi kod samo ako automatizovani test ne uspe
- Eliminišemo dupliranje

Ovo su dva jednostavna pravila, ali ona generišu složeno individualno i grupno ponašanje sa tehničkim implikacijama kao što su sledeća.

- Moramo dizajnirati organski, sa pokrenutim kodom koji pruža povratne informacije između odluka.
- Moramo sami pisati testove, jer ne možemo čekati 20 puta dnevno da neko drugi napiše test.
- Naše razvojno okruženje mora pružati brzu reakciju na male promene.

- Naši dizajni moraju se sastojati od mnogo visoko kohezivnih, labavo povezanih komponenti, samo da bi testiranje bilo lako.

Dva pravila impliciraju redosled programskih zadataka.

1. Crveno – Napišite mali test koji pada, a možda se u početku čak ni ne kompajlira.
2. Zeleno – Učinite da test prođe, čineći sve neophodne "grehe" u tom procesu.
3. Refaktorišite – Uklonite svo dupliranje stvoreno samim pokretanjem testa.

Crveno/zeleno/refaktoriši – TDD mantra.

Pod pretpostavkom da je takav stil programiranja moguć, nadalje bi bilo moguće dramatično smanjiti gustinu defekata koda i učiniti predmet rada kristalno jasnim svima uključenima. Ako je tako, onda pisanje samo onog koda koji je zahtevan neuspešnim testovima takođe ima društvene implikacije.

- Ako se gustina defekata može dovoljno smanjiti, onda kontrola kvaliteta (QA) može preći sa reaktivnog na proaktivni rad.
- Ako se broj neprijatnih iznenađenja može dovoljno smanjiti, onda projektni menadžeri mogu sarađivati iz minuta u minut, umesto na dnevnom ili nedeljnom nivou.
- Ako se teme tehničkih razgovora mogu dovoljno razjasniti, onda softverski inženjeri mogu sarađivati iz minuta u minut, umesto na dnevnom ili nedeljnom nivou.
- Opet, ako se gustina defekata može dovoljno smanjiti, onda možemo imati isporučiv softver sa novom funkcionalnošću svakog dana, što vodi novim poslovnim odnosima sa korisnicima.

Dakle, koncept je jednostavan, ali koja je moja motivacija? Zašto bi softverski inženjer preuzeo dodatni posao pisanja automatizovanih testova? Zašto bi softverski inženjer radio u sitnim, malim koracima kada je njegov ili njen um sposoban za velike uzlete dizajna? U pitanju je hrabrost.

Hrabrost

Razvoj vođen testovima je način upravljanja strahom tokom programiranja. Ne mislim na strah na loš način – *jadni programer* – već na strah u legitimnom smislu, ovo je težak problem i ne vidim kraj od početka. Ako je bol prirodna poruka za "Stani!", onda je strah prirodna poruka za "Budi oprezan." Biti oprezan je dobro, ali strah ima niz drugih efekata.

- Strah vas čini neodlučnim.
- Strah vas tera da manje komunicirate.

- Strah vas tera da izbegavate povratne informacije.
- Strah vas čini mrzovoljnim.

Nijedan od ovih efekata nije koristan prilikom programiranja, posebno kada se programira nešto teško. Stoga se postavlja pitanje kako se suočiti s teškom situacijom i,

- Umesto da budemo neodlučni, počnemo konkretno da učimo što je brže moguće.
- Umesto da ćutimo, komuniciramo jasnije.
- Umesto da izbegavamo povratne informacije, tražimo korisne, konkretne povratne informacije.
- (Na mrzovoljnosti ćete morati sami da radite.)

Zamislite programiranje kao okretanje ručice da biste izvukli kantu vode iz bunara. Kada je kanta mala, okretanje ručice je lako. Kada je kanta velika i puna vode, umorićete se pre nego što kanta bude potpuno izvučena. Potreban vam je mehanizam sa zaustavnom točkom (ratchet) koji vam omogućava da se odmorite između perioda okretanja. Što je kanta teža, zubi na zaustavnom točku moraju biti bliže. Testovi u razvoju vođenom testovima su zubi zaustavnog točka. Kada jedan test proradi, znamo da radi, sada i zauvek. Jedan smo korak bliže da sve radi nego što smo bili kada je test zakazivao. Sada ćemo pokrenuti sledeći, pa sledeći, pa sledeći. Po analogiji, što je programski problem teži, to bi svaki test trebalo da pokrije manje područje. Čitaoci moje knjige *Extreme Programming Explained* primetiće razliku u tonu između Ekstremnog programiranja (XP) i TDD-a. TDD nije apsolut kao XP. XP kaže: "Evo stvari koje morate biti u stanju da uradite da biste bili spremni da se dalje razvijate." TDD je malo maglovitiji. TDD je svest o jazu između odluke i povratne informacije tokom programiranja, i tehnike za kontrolu tog jaza. "Šta ako radim dizajn na papiru nedelju dana, pa zatim testiram kod? Da li je to TDD?" Naravno, to je TDD. Bili ste svesni jaza između odluke i povratne informacije, i namerno ste kontrolisali taj jaz.

Ipak, većina ljudi koji uče TDD otkrivaju da se njihova praksa programiranja zauvek promenila. *Test Infected* je fraza koju je skovao Erich Gamma da opiše ovu promenu. Možda ćete se naći u situaciji kako pišete više testova ranije, i radite u manjim koracima nego što ste ikada sanjali da bi bilo razumno. S druge strane, neki softverski inženjeri uče TDD, a zatim se vraćaju na svoje ranije prakse, čuvajući TDD za posebne prilike kada obično programiranje ne napreduje.

Svakako postoje programski zadaci koji se ne mogu voditi isključivo testovima (ili barem, još ne). Sigurnosni softver i konkurentnost, na primer, su dve teme gde je TDD nedovoljan da mehanički demonstrira da su ciljevi softvera ispunjeni. Iako je istina da sigurnost zavisi od esencijalno koda bez grešaka, ona se takođe oslanja na ljudsku procenu metoda korišćenih za osiguravanje softvera. Suptilni problemi konkurentnosti ne mogu se pouzdano duplirati pokretanjem koda.

Kada završite sa čitanjem ove knjige, trebalo bi da budete spremni da

- Započnete jednostavno
- Pišete automatizovane testove
- Refaktorišete da biste dodavali dizajnerske odluke jednu po jednu

Ova knjiga je organizovana u tri dela.

- Deo I, Primer novca – Primer tipičnog modela koda napisanog korišćenjem TDD-a. Primer mi je pre mnogo godina dao Ward Cunningham i od tada sam ga mnogo puta koristio: viševalutna aritmetika. Ovaj primer će vam omogućiti da naučite da pišete testove pre koda i organski razvijate dizajn.
- Deo II, xUnit primer – Primer testiranja složenije logike, uključujući refleksiju i izuzetke, razvijanjem okvira za automatizovano testiranje. Ovaj primer će vas upoznati sa xUnit arhitekturom koja je u srži mnogih programerskih alata za testiranje. U drugom primeru, naučićete da radite u još manjim koracima nego u prvom primeru, uključujući i vrstu samoreferentne "hoo-ha" koju vole kompjuterski naučnici.
- Deo III, Obrasci za razvoj vođen testovima – Uključeni su obrasci za odlučivanje koje testove pisati, kako pisati testove korišćenjem xUnit-a, i izbor "najvećih hitova" dizajnerskih obrazaca i refaktorisanja korišćenih u primerima.

Primere sam pisao zamišljajući sesiju parnog programiranja. Ako volite da pogledate mapu pre nego što lutate, onda možda želite da idete pravo na obrasce u Delu III i koristite primere kao ilustracije. Ako više volite da samo lutate i zatim gledate mapu da vidite gde ste bili, onda pokušajte da čitate primere, pozivajući se na obrasce kada želite više detalja o nekoj tehnici, i koristeći obrasce kao referencu. Nekoliko recenzenata ove knjige komentarisalo je da su najviše izvukli iz primera kada su pokrenuli programsko okruženje, uneli kod i pokrenuli testove dok su čitali.

Napomena o primerima. Oba primera, izračunavanje u više valuta i okvir za testiranje, izgledaju jednostavno. Postoje (i video sam) komplikovani, ružni, neuredni načini rešavanja istih problema. Mogao sam da izaberem jedno od tih komplikovanih, ružnih, neurednih rešenja, da bih knjizi dao "dašak realnosti". Međutim, moj cilj, i nadam se vaš cilj, je da pišem čist kod koji radi. Pre nego što prebrzo procenite primere kao previše jednostavne, provedite 15 sekundi zamišljajući programerski svet u kojem je sav kod ovako jasan i direktan, gde nije bilo komplikovanih

rešenja, samo naizgled komplikovanih problema koji vape za pažljivim razmišljanjem. TDD vam može pomoći da se dovedete do upravo tog pažljivog razmišljanja.