Metodika - Testovanie

Unit testovanie

Jednotkové testovanie slúži na testovanie malých izolovaných funkcionalít. Jednotkový test má najvyššiu granularitu. Jednotkové testy sú rýchle, spoľahlivé, cielené a malé.

Jednotkový test obsahuje len jeden assert

Treba granulárnejšie písať testy. V teste by sa mal nachádzať iba jeden assertion statement. Ak majú viac assertov, často bývajú testy veľké a náročné na setup a tým pádom sa ťažšie uplatňujú a setupujú. Ak sa v teste nachádza len jeden statement, znamená to že je testovaná tá jediná vec, ktorú chceme.

Jednotkový test by nemal obsahovať if podmienky

Test by nemal obsahovať if statement lebo to indikuje možnosť viacnažného správania na základe situácie. Takéto správanie je neželané. Jednotkový test by mal v každom prípade vykonať to isté ak má byť vyhodnotený ako úspešný. Testy musia byť statické, nie s dynamickým správaním.

Jednotkový test testuje jednu vytvorenú triedu

Jednotkový test by mal byť rozsahom limitovaný na testovanie jednej maximálne jednej metódy alebo triedy. V teste by sa malo vytvorenie triedy, ktorú testujete vyskytovať raz. Vytváranie viacerých tried predstavuje komplikáciu a test sa stáva neprehľadnejším a malé špecifickým. V prípade, že test zlyhá je potrebné vedieť okamžite určiť, v ktorej triede alebo metóde sa nachádza chyba, čo pri kompilácii viacerých tried nie je okamžite jasné.

Závislosti testovacej triedy by sa nemali vytvárať. Nahrádzajú sa použitím tzv. stubs (náhrada, atrapa). Mali by byť imitované pomocou imitovacích knižníc ako Moq alebo NSubstitute.

Taktiež by sa mali imitovať triedy, ktoré priamo neovládame ako napríklad file system. Takáto imitácia sa robí pomocou pomocnej wrapper triedy, ktorá povie ako sa má imitovaná trieda správať.

Príklad:

```
public void
should_find_only_text_files_in_the_specified_directory() {
    File file = mock(File.class);
    when(file.list()).thenReturn(new String[] { "readme.txt",
"foobar" });

    assertThat(store.list(file)).contains("readme.txt");
```

```
}
```

Takáto trieda zabezpečí, že sa nespoliehame na to, čo sa reálne v danom priečinku nachádza.

Jednotkové testy neobsahujú natvrdo napísané hodnoty pokiaľ nie sú relevantné

V jednotkových testoch by nemali figurovať natvrdo napísané hodnoty pre vstupné jednotky. Mali by sa namiesto nich používať takzvané fixtures.

Fixture je nemenný set objektov pre testovanie, ktoré slúžia ako vstupy pre testy. Zmysel toho je že test pracuje v dobre poznanom prostredí vstupných hodnôt aby mohli byť výsledky testu sú opakovateľne dosiahnuté.

Jednotkové testy sú bez poznania stavu aplikácie

Nezáleží na poradí vykonávania testov. Testy nepredpokladajú výsledky iných testov. Testy pracujú iba nad dátami, ktoré boli v rámci predprípravy vykonania testy pridané do testovacej databázy.

Ako písať unit testy?

- Názvy testov by mali vystihovať čo test robí. Mali by začínať slovesom ako should alebo can (it shoud).
- Názvy testov by mali byť písané snake_case.
- Nebáť sa primeranej duplicity kódy. Je lepšie mať duplicitné riadky kódu, lebo to zlepšuje čitateľnosť testu.
- Redukovať before a after dekorátory, pretože môžu narušiť príbeh pri čítaní testov.
- Nebáť sa viac riadkov kódu pri testovacích funkciách. Pre lepšiu čitateľnosť je lepšie mať kód pri sebe ako ho mať skrytý.
- Lokálne premenné - v niektorých prípadoch je lepšie vytvoriť lokálne premenné s cieľom lepšej čitateľnosť (napríklad výpočet) a niekedy je výhodnejšie ponechať natvrdo nastavenú premennú.

```
// not so clear
isAllowedToDrink(currentYear - yearOfBirth);

// this version is easier to understand
int age = currentYear - yearOfBirth;
isAllowedToDrink(age);
```

- Niekedy si vieme uľahčiť prehľadnosť premenných v assertoch.

```
assertThat(createUser("name", "email", "site")).isEqualTo(new
User("name", "email", "site"));

assertThat(toLowerCase("Uppercase Characters")).isEqualTo("uppercase
characters");
```

  “Upppercase Characters” týmto spôsobom prehľadnejšie vystihuje podstatu assertu.

- Netreba písať veľké description pri assertoch lebo pri dobrej knižnici, ktorá disponuje veľa metódami na assertovanie, ktoré vedia chybovú správu dobre reprodukovať.

- V unit teste sa nepoužívajú logy, lebo chyby by mali byť jasne indikované assertami a nie je potrebné ich pomocou logov identifikovať. Ak danú chybu nie je možné identifikovať, je potrebné daný test refaktorovať.

- V niektorých prípadoch je potrebné pomocné triedy vytvárať využitím viac ako jedného riadku. Taktiež je niekedy potrebné poslať rôzny počet parametrov, čo sa má odzrkadliť na správaní danej triedy. Takúto triedu je v tom prípade možné simulovať vlastným builderov (triedou ktorá danú triedu vytvorí a vráti) v rámci súboru testu aby sa eliminovala redundancia ale zachovala sa čitateľnosť.

Ako veľmi testovať unit testami

Niekedy sa tím ocitne v situácii, kedy netestuje reálne správanie testovaného softvéru, ale len píše testy, aby splnil požiadavku pokrytia kódu testami. Tieto testy nesprávne testujú a overujú funkcionalitu a nedávajú tímu istotu že ich kód beží správne. Namiesto zamerania sa na kvantitatívne písanie unit testov pre každú metódu triedy je dobré sa zamyslieť nad testovaním správania v unit testoch. Pre každý komponent je vhodné otestovať jeho reálnu funkcionalitu a nie len overovanie návratových hodnôt, ktoré nemusia reprezentovať jeho reálne používanie. Naplnenie pokrytia kódu niekedy vedie k potrebe písania veľmi triviálnych testov, ktoré zvytočne spotrebujú náklady a čas.

Integračné Testovanie

Integračné testovanie zahŕňa spoluprácu viacerých modulov, ktoré sa predtým otestovali samostatne. Pri väčších projektoch je typické, že softvér pozostáva z viacerých modulov, na ktorý pracovali viacerí programátori. Cieľom je odhaliť chyby, ktoré môžu nastať pri vzájomnej interakcii modulov.

Stubs and Drivers

Sú jednoduché programy, ktoré simulujú správanie a komunikáciu modulov, ktoré ešte nie sú implementované.
stub (náhrada) - je volaný modulom počas testovania
driver (ovládač) - volá modul, ktorý má byť testovaný

Typy integračného testovania

Bottom-Up

Je stratégia, pri ktorej sa najskôr otestujú najmenšie moduly a postupne sa prechádza na vyššie úrovne pokým nie sú zapojené všetky moduly softvéru.
Výhody: ľahšie nájdeme chybu
Nevýhody: celú (najdôležitejšiu časť) testujeme na koniec, nevieme testovať prototyp

Top-Down

Najskôr sa testujú najvyššie moduly softvéru. Ak máme modul, ktorý nie je implementovaný používajú sa náhrady (stubs)
Výhody: ľahko nájdeme chybu, možné testovať prototyp, kritické miesta testujeme najskôr
Nevýhody: potrebujeme veľa náhrad (stubs), moduly na nižšej úrovni sú testované nedostatočne

Príklady

Overenie platnosti tokenu - prijatie tokenu z VT, overenie či je token platný, vrátenie odpovede na VT

Feature testovanie

Feature testovanie slúži na testovanie novo pridanej alebo pozemnej funkcionality systému. Jeho cieľom je odhaliť nefungujúce časti a bugy v kóde, pričom sa pomocou tohto testovania vyhodnocuje, či je daná funkcionalita vhodná pre systém.

Ako spraviť feature testovanie:

- je potrebné ovládať danú feature (špecifikáciu, možné správanie, edge-casy) a poznať jej požiadavky.
- je dôležité zamerať sa na slabé články danej funkcionality, ktoré by ju mohli pokaziť a testy zamerať na ich testovanie
- vytvoriť vhodné testovanie scenáre, ktoré zahŕňajú aj pozitívne, negatívne, očakávané a neočakávané výsledky.
- feature test imituje ako reálny používateľ použije danú funkcionalitu

Feature testovanie sa vykonáva až po implementovaní celej funkcionality a otestovaní menšími testami (unit a integration).

Pri našom projekte elektronických volieb si ako integračný test vieme predstaviť overenie funkcionality volebného terminálu (aplikácie), pričom cieľom je od prvotného vloženia tagu sa dostať až po odoslanie hlasu na gateway.

Druhým takým feature testom by mohlo byť zobrazenie výsledkov volieb, kde sa volič dozvie rôzne štatistiky o priebehu a finálnom výsledku.

Regresné testovanie

Regresný test slúži na odhalenie skrytých chýb, ktoré nevzniknú priamo v novo pridanej funkcionalite, ale niekde inde v kóde ako následok jej pridania.

Pri regresnom teste je dôležité zvážiť aká veľká časť kódu sa pri regresnom teste testuje. Testovanie všetkých častí a spustenie všetkých menších testov (unit, integration a feature) býva pri veľkých projektoch nákladné a v takom prípade je potrebné určiť časti kódy, ktoré sú relevantné (a môžu byť ovplyvnené novo pridanou funkcionalitou)

Ako vybrať správne testy a časti kódu do regresného testovania:

- scenáre ktoré sú často chybné
- scenáre, ktoré sú pre používateľa viditeľnejšie
- hlavné časti systému
- časti, ktoré prešli najväčšími alebo nedávnymi zmenami
- všetky integračné testy
- všetky komplexné testy (feature)
- testovanie edgecasov
- testovanie úspešnej a neúspešnej funkcionality
- prioritizácia testov podľa dôležitosti

Akceptačné testovanie

Akceptačné testovanie je vykonávané konečným používateľom alebo klientom v neskorých fázach projektu s cieľom validovať naplnenie požiadaviek na systém a business využitie.

Akceptačné testovanie sa môže robiť až keď jednotkové, integračné a regresné boli úspešné.Výsledkom akceptačného testovania je rozhodnutie klienta/koncového používateľa je akceptovanie alebo neakceptovanie daného produktu. Takéto akceptačné testovanie je potrebné hlavne v prípade, ak bol produkt vyvinutý podľa špecifikácie a nebol predmetom pravidelnej komunikácie a validácie produktu s klientom.

Predpoklady akceptačného testovania:

- jasná definícia biznis požiadaviek
- dokončený aplikačný kód
- vykonané unit, integration a feature testy
- iba drobné chyby sú akceptovateľné pred akceptačným testovaním
- neboli nájdené veľké chyby počas regresného testovania
- všetky testami odhalené nedostatky sú opravené

Používateľské testovanie

Úlohou používateľského testovania je odhaliť možné chyby v použiteľnosti aplikácií. Malo by sa vykonávať s ľuďmi z rôznych - napr. vekových skupín, úrovne vzdelania a technickej zdatnosti. Pri testovaní pozorujeme chovanie ľudí čím môžeme odhaliť chyby, ktoré môžu byť počas vývoja skryté.

Najlepšie je testovanie vykonať na skupine 5 až 7 ľudí, ktorí sú dostatočnou vzorkou na overenie chýb v užívateľskom rozhraní aplikácie. Testovanie by malobyť dôkladne pripravené a malo by postupovať podľa striktného scenára, ktorý zabezpečí rovnaké testovacie scenáre pre účastníkov testovania. Pri testovaní by mal byť koordinátor, ktorý začne testovanie s participantom, číta mu otázka, odpovedá na prípadné nejasnosti a v kritickej situácii vie respondentovi pomôcť aby sa nenarušil priebeh testovania. Okrem koordinátora by mal testovanie sledovať zapisovateľ, ktorý zapisu respondentove odpovede, reakcie a správanie, ktoré budú po teste vyhodnocované a spracovávané. Pri testovaní sa odporúča uchovávať audiovizuálny záznam aby sa nestratili žiadne informácie.

Po testovaní sa vykoná evaluácia v ktorej sa identifikujú problémy zistené počas testovania. Chyby sa kategorizujú podľa dôležitosti a dopadu a posunú sa klientovi na zapracovanie.

Testovanie Django aplikácie

Praktiky:

- ak sa može kód pokaziť, treba ho testovať
- každý test by mal testovať len jednu funkciu
- testy by nemali byt zložité ani rozsiahle
- testy sa spúšťajú pri pull a push z repozitára a pri nasadení do stagingu

Knižnice:

- django-webtest
- coverage
- django-discover-runner
- factory_boy. model_mommy, mock (mokovacie knižnice a fixtures)

```
from django.test import TestCase
from whatever.models import Whatever
from django.utils import timezone
from django.core.urlresolvers import reverse
from whatever.forms import WhateverForm

# models test
class WhateverTest(TestCase):

    def create_whatever(self, title="only a test", body="yes, this is only a test"):
        return Whatever.objects.create(title=title, body=body, created_at=timezone.no

    def test_whatever_creation(self):
        w = self.create_whatever()
        self.assertTrue(isinstance(w, Whatever))
        self.assertEqual(w.__unicode__(), w.title)
```

Na hore uvedenom príklade vidíme test vytvorenie modelu a overenia správneho vytvorenia nadpisu v danom objekte. Je to príklad unit testu.

Testovanie frontend Views

Na testovanie view je v niektorých prípadoch potrebné použiť knižnicu Selenium na testovanie requestov a ich návratových kódov a taktiež automatizovane klikať na elementy.

Jednoduchý príklad overenia návratovej hodnoty zo špecifickej url:

```
# views (uses reverse)

    def test_whatever_list_view(self):
        w = self.create_whatever()
        url = reverse("whatever.views.whatever")
        resp = self.client.get(url)

        self.assertEqual(resp.status_code, 200)
        self.assertIn(w.title, resp.content)
```

Na nasledujúcom obrázku môžeme vidieť príklad testovania vyplnenia formuláru na URL /add. Selenium si stiahne obsah stránky a pomocou id nájde potrebné elementy a vyplní ich. Po vyplnení údajov vyvolá kliknutie nad submit tlačidlom a overí či bol presmerovaný na URL /.

```
# views (uses selenium)

import unittest
from selenium import webdriver

class TestSignup(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def test_signup_fire(self):
        self.driver.get("http://localhost:8000/add/")
        self.driver.find_element_by_id('id_title').send_keys("test title")
        self.driver.find_element_by_id('id_body').send_keys("test body")
        self.driver.find_element_by_id('submit').click()
        self.assertIn("http://localhost:8000/", self.driver.current_url)

    def tearDown(self):
        self.driver.quit

if __name__ == '__main__':
    unittest.main()
```

Testovanie formulárov spočíva vo vyplnení údajov formulára a následnom overení či je formulár validný.

```
def test_valid_form(self):
    w = Whatever.objects.create(title='Foo', body='Bar')
    data = {'title': w.title, 'body': w.body,}
    form = WhateverForm(data=data)
    self.assertTrue(form.is_valid())

def test_invalid_form(self):
    w = Whatever.objects.create(title='Foo', body='')
    data = {'title': w.title, 'body': w.body,}
    form = WhateverForm(data=data)
    self.assertFalse(form.is_valid())
```

Testovať sa môže aj API rozhranie, pri ktorom sledujeme či sa vráti správny formát odpovede. Podobný spôsobom môžeme testovať aj či odpoveď obsahuje správne dáta.

```
from tastypie.test import ResourceTestCase

class EntryResourceTest(ResourceTestCase):

    def test_get_api_json(self):
        resp = self.api_client.get('/api/whatever/', format='json')
        self.assertValidJSONResponse(resp)

    def test_get_api_xml(self):
        resp = self.api_client.get('/api/whatever/', format='xml')
        self.assertValidXMLResponse(resp)
```

Testy sa spúštajú príkazom `manage.py test` (ktorý ako ďalší nepovinný parameter akceptuje skupinu alebo názov testu)

Výsledok všetkých testov sa zobrazí v prehľadnej tabuľke so stavom jednotlivých testov a celkovým časom trvania.

```
test_signup_fire (whatever.tests.TestSignup) ... ok
test_invalid_form (whatever.tests.WhateverTest) ... ok
test_valid_form (whatever.tests.WhateverTest) ... ok
test_whatever_creation (whatever.tests.WhateverTest) ... ok
test_whatever_list_view (whatever.tests.WhateverTest) ... ok

----------------------------------------------------------------------
Ran 5 tests in 12.753s

OK
```

Alternatívy knižnice Selenium

Alternatívne k knižnici Selenium je možné použiť na testovanie frontendu testovaciu knižnicu Cypress. Selenium je všeobecný automatizér prehliadača, ktorý umožňuje aj testovanie web aplikácií. Cypress je vytvorený priamo na testovanie web aplikácií.

Priamo pre testovanie Svelte aplikácie existuje aj knižnica testing-library/svelte-testing-library.

Rozloženie Svelte kódu vzhľadom na testovateľnosť aplikačnej logiky

Svelte jazyk odporúča logiku aplikácie rozdeliť na dve časti - vykresľovanie grafického rozhrania (.svelte súbory) a na .ts súbory obsahujúce hlavnú logiku aplikácie, ktoré sú importované do .svelte súborov. Toto rozloženie aplikácie umožní testovať hlavnú logiku aplikácie aj samostatne bez testera cez prostredie webového prehliadača.

Zdroje

https://medium.com/vx-company/the-5-unit-testing-guidelines-f21d39c33e0b
https://github.com/elefevre/elements-of-unit-testing-style
https://www.guru99.com/integration-testing.html
https://www.javatpoint.com/integration-testing
https://medium.com/@sameernyaupane/php-test-driven-development-part-5-integration-testing-51535ca56bf0
https://medium.com/swlh/laravel-end-to-end-testing-with-cypress-e574a73ce222
https://www.guru99.com/user-acceptance-testing.html
https://realpython.com/testing-in-django-part-1-best-practices-and-examples/
https://svelte.dev/faq