Testavimas

Programų testavimas yra būtinas programinės įrangos kūrimo proceso etapas, užtikrinantis, kad kuriama sistema atitiktų nustatytus reikalavimus ir veiktų patikimai, be klaidų. Testavimas padeda identifikuoti klaidas, saugumo spragas, našumo trūkumus bei kitas problemas, kurios galėtų paveikti programos kokybę ar vartotojo patirtį.

Tai yra atskira, didelė mokslo ir praktikos sritis, kurios specialistai mokosi visą gyvenimą. Šiame kurse mes prisiliesime tik prie nedidelės testavimo dalies. Išmoksime kelis testavimo metodus, kad galėtume efektyviai tikrinti programas.

Ekvivalentinis sudalijimas

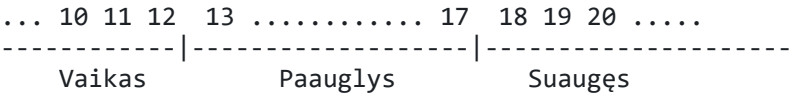
Ekvivalentinis sudalijimas - tai programinės įrangos testavimo metodas, kai įvesties duomenys yra suskirstomi į grupes (vadinamas ekvivalentinėmis klasėmis) pagal tam tikrus kriterijus. Tokiu būdu sudarytos grupės leidžia pasirinkti keletą testavimo atvejų iš kiekvienos grupės ir taip išbandyti didelę dalį funkcionalumo.

Duomenų rinkinys priklauso tai pačiai ekvivalentinei klasei, jei kiekvienas duomenų rinkinio egzempliorius paliečia tą pačią programos kodo dalį. Idėja ta, kad vykdant tas pačias instrukcijas, daugeliu atvejų programa veiks taip pat (teisingai ar neteisingai). Todėl, norint ištestuoti ekvivalentinę klasę - reikia bent vieno duomenų rinkinio egzemplioriaus.

Pavyzdys:

```
if amžius > 12 and amžius < 18:
    print("Asmuo yra paauglys")
elif amžius >= 18:
    print("Asmuo yra suaugęs")
else:
    print("Asmuo yra vaikas")
```

Šiame programos fragmente matome sąlygos sakinį, kuriame yra nusakytas intervalas (tarp 12 ir 18). Vadinasi galime suskirstyti duomenų rinkinius į tokias **3** klases:



Taigi, pagal ekvivalentinio sudalijimo idėją, testavimui būtina paimti **3** *amžius* kintamojo reikšmes iš kiekvienos klasės. Tarkim: **10**, **15** ir **19**.

Ar ekvivalentinei klasei reikia paimti daugiau reikšmių? Jei ekvivalentinės klasės sudarytos pagal programinį kodą, kuris toks paprastas, kaip šiame pavyzdyje, tuomet - ne. Tačiau, dažnai testavimo rinkinius mes sudarome ne iš programinio kodo, o iš programos specifikacijos ir visi programos keliai nėra žinomi. Todėl ekvivalentinei klasei testuoti paimama daugiau rinkinių.

Testavimo pavyzdys

Jūs jau žinote, kad programavime funkcijos (arba metodai objektiniame programavime) yra labai naudingos, nes jos padeda struktūrizuoti programą. Tai yra, ją lengviau skaityti ir tuo pačiu kurti. Pasirodo, funkcijos yra ypač naudingos tuomet kai reikia testuoti arba tikrinti programą. Jos izoliuoja tikrinamą kodo gabaliuką ir taip leidžia patikrinti nedidelę programos dalį.

Taigi, šiame pavyzdyje mes paimsime aukščiau esantį kodą, įdėsime jį į funkciją ir patikrinsime pagal ekvivalentinio sudalijimo taisykles.

```
1. def ar_atsiskaitė(pažymys):
2.     if 0 < pažymys < 5:
3.         print("Neatsiskaitė")
4.     elif pažymys > 5:
5.         print("Atsiskaitė")
6.     else:
7.         print("Pažymys netinkamas")
8.
9. ar_atsiskaitė(5)
```

Įvykdykite šį kodą ir pamatysite, kad jis išspausdina teisingus atsakymus. Mes patikriname rankiniu būdu. Gali kilti klausimas, o ar negalima tikrinti funkcijas automatizuotai? Juk žmogus gali klysti patikrindamas ne tokias paprastas programas! Atsakymas - taip, galima (ir netgi būtina), tačiau tai būtų sudėtingesnio kurso dalis ir šiame kurse automatizuoto testavimo metodų nenagrinėsime. Vis tik, jei labai įdomu ir norisi išmokti efektyviau testuoti programas - pasidomėkite vientų testavimu (angl. *unit testing*).

Ribinių reikšmių analizė

Prisiminkime nagrinėtą asmens brandos nustatymo programą. Ji sukurta pagal tokias taisykles:

1. asmuo iki 12 metų imtinai yra vaikas;
2. asmuo nuo 13 metų iki 17 metų imtinai yra paauglys;
3. asmuo nuo 18 metų yra suaugęs.

Programa buvo teisinga. Mes šiek tiek paišdykaukime ir parašykime ne tokią teisingą programą:

```
 1. def nustatyti_brandą(amžius):
 2.     if amžius >= 12 and amžius < 18:
 3.         print("Asmuo yra paauglys")
 4.     elif amžius >= 18:
 5.         print("Asmuo yra suaugęs")
 6.     else:
 7.         print("Asmuo yra vaikas")
 8.
 9. nustatyti_brandą(10)
10. nustatyti_brandą(15)
11. nustatyti_brandą(20)
```

Ar pastebite klaidą? Programoje pažeidžiama pirma taisyklė! Programa net ir dvylikos metų vaiką laikys paaugliu. Įvykdykime programą ir pamatysime, kad klaida su tokiais testavimo duomenimis yra neaptinkama. Taigi, šiuo atveju ekvivalentinio sudalijimo metodas neveikia :(Taigi, susipažinkime su metodu, kuris leis aptikti šią klaidą!

Ribinių reikšmių analizė - tai programinės įrangos testavimo metodas, kai testavimo atvejai yra sudaromi remiantis ribinėmis reikšmėmis tam tikrame intervale. Pagrindinė šio metodo idėja yra tokia: tikėtina, kad klaidos pasirodys būtent ties ribinėmis vertėmis, nes sistemos reakcija į kraštutines reikšmes gali skirtis nuo reakcijos į vidutines reikšmes.

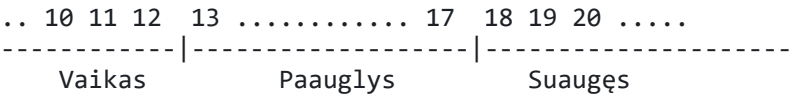
Prisiminkime ekvivalentinį sudalijimą. Įvesties duomenys gali būti suskirstyti į klases, kuriose kiekvienos klasės vertės sistemoje elgiasi panašiai. Tarp šių klasių atsiranda ribos, kur reikšmės keičia savo klasę - tai vadinama ribine verte.

Ribinės reikšmės turi būti parenkamos taip, kad mažiausias pasikeitimas įtakotų klasės (arba intervalo) pasikeitimą. Pavyzdžiui, jei turime integer tipo kintamąjį, tuomet mažiausias pasikeitimas - vienetu. Jei tai skaičius su kableliu, tuomet - tai mažiausia įmanoma reikšmė.

Panagrinėkime pavyzdį:

```
if amžius > 12 and amžius < 18:
    print("Asmuo yra paauglys")
elif amžius >= 18:
    print("Asmuo yra suaugęs")
else:
    print("Asmuo yra vaikas")
```

Šiame programos fragmente matome sąlygos sakinį, kuriame yra nusakytas intervalas (tarp 12 ir 18). Vadinasi, klasės pasikeitimas vyksta intervalo ribom:



Šiuo atveju reikia testuoti su šiomis kintamojo amžius reikšmėm: **12**, **13**, **17**, **18**. Jeigu tai skaičius su kableliu, kuriam skiriami du skaitmenys po kablelio, tuomet reiktų testuoti su reikšmėm: **12.99**, **13.00**, **17.99**, **18.00**.

Pamėginkime patikrinti mūsų programą, kuri yra su klaida:

```
 1. def nustatyti_brandą(amžius):
 2.     if amžius >= 12 and amžius < 18:
 3.         print("Asmuo yra paauglys")
 4.     elif amžius >= 18:
 5.         print("Asmuo yra suaugęs")
 6.     else:
 7.         print("Asmuo yra vaikas")
 8.
 9. nustatyti_brandą(12)
10. nustatyti_brandą(13)
11. nustatyti_brandą(17)
12. nustatyti_brandą(18)
```

Tikėtini (tie kurių tikimės) programos rezultatai:

```
Asmuo yra vaikas
Asmuo yra paauglys
Asmuo yra paauglys
Asmuo yra suaugęs
```

Gavome:

```
Asmuo yra paauglys
Asmuo yra paauglys
Asmuo yra paauglys
Asmuo yra suaugęs
```

Ką gi - aptikome klaidą! Reiškia šiuo atveju ribinių reikšmių analizės metodas veikia. Tik, jokiu būdu neskubėkite nurašyti ekvivalentinio sudalijimo metodo! Dabar ribinių reikšmių analizė veikia. Kitu atveju - gali ir nesuveikti.

Ekstremalių reikšmių analizė

Ekstremalių reikšmių analizė skirta patikrinti, kaip sistema veikia esant labai didelėms arba labai mažoms (ekstremalioms) įvesties reikšmėms, kurios yra už normalaus (leistino) diapazono ribų. Ekstremalioms reikšmėms taip pat kartais priskiriamos tokios reikšmės, kurios yra jautrios taikomąjai sričiai. Pavyzdžiui, skaičius 0 irgi gali būti laikoma ekstremalia reikšme ten kur naudojama dalyba ir gali pasireikšti dalybos iš nulio operacija.

Ekstremalių reikšmių testavimas padeda nustatyti sistemos elgesį neįprastose situacijose, siekiant užtikrinti, kad sistema nesugriūtų ir tinkamai praneštų apie klaidas, kai įvedamos reikšmės, kurios yra už įprasto veikimo ribų.

Kokios tos ekstremalios reikšmės - priklauso nuo taikomosios srities. Ekstremalias reikšmes siekama parinkti tokias, kurios perpildytų kintamojo tipų galimų reikšmių aibę arba masyvą.

Panagrinėkime kvadratinės lygties šaknų skaičiavimo programą, kuri neapsaugota nuo klaidingo veikimo:

```python
import math

def skaičiuoti_šaknis(a, b, c):
    diskriminantas = b**2 - 4*a*c
    root1 = (-b + math.sqrt(diskriminantas)) / (2*a)
    root2 = (-b - math.sqrt(diskriminantas)) / (2*a)
    print(f"Šaknys: x1 = {root1}, x2 = {root2}")

# Kai diskriminantas > 0 (skaičiuoja teisingai)
skaičiuoti_šaknis(1, 3, 2)
# Kai diskriminantas = 0, turėtų skaičiuoti 1 šaknį
skaičiuoti_šaknis(1, 2, 1)
# Kai diskriminantas < 0, turėtų rašyti, kad lygtis realių šaknų neturi
skaičiuoti_šaknis(3, 1, 4)

# Tikrinam ekstremalias a reikšmes
skaičiuoti_šaknis(-10000000000000, 2, 1)
skaičiuoti_šaknis(-0.000000000001, 2, 1)
# Kai a = 0, turėtų rašyti kad tai nėra kvadratinė lygtis
skaičiuoti_šaknis(0, 2, 1)
skaičiuoti_šaknis(0.000000000001, 2, 1)
# Turėtų rašyti, kad lygtis realių šaknų neturi
skaičiuoti_šaknis(10000000000000, 2, 1)
```

Šioje programoje tiek daug klaidų, kad ji net negali suveikti kelis kartus nenulūždama. Tai yra mūsų, kaip testuotojų, **sėkmė!** :) Testas laikomas sėkmingu, jei jis aptiko klaidas (o ne atvirkščiai!). Jei kalbėtume apie kintamojo a ekstremalias reikšmes, tai jos šiame pavyzdyje yra: **-10000000000000**, **-0.000000000001**, **0**, **0.000000000001**, **10000000000000**. Kaip matote, šios reikšmės leidžia aptikti, kad kintamasis **a** negali būti lygus nuliui ir dėl to reiktų pridėti atitinkamą patikrinimą. Kitas rimtas klausimas - kodėl pasirinkau 10000000000000? Ogi todėl, kad šis skaičius nepatenka į daugelio programavimo kalbų (C, C++, JAVA, C#...) sveiko skaičiaus apibrėžimo sritį ir jei nesuprogramuotas patikrinimas, programa nulužtų arba skaičiuotų neteisingai. Pitonas nuo viso šito apsaugo, tačiau tai nereiškia, jog šių atvejų nereikia tikrinti jei programuojame Pitonu.

Kaip galvojate - ar jau pakanka testavimo duomenų? Tikrai ne - juk nepatikrinome, kaip elgtųsi programa su ekstremaliomis **b** ir **c** koeficientų reikšmėmis. Matote - testavimo kodas tampa didesnis už pačios programos kodą :) Taigi, kol neišmokote pažangesnių testavimo būdų - pasikliaukite intuicija tikrindami šiuo metodu silpniausias programos vietas.

Pabaigai - *ištaisykite duotą programą*, kad ji veiktų teisingai su duotais testiniais variantais!

Testavimu paremtas programavimas

Tai nėra tema, kurią būtina žinoti šiame kurse. Tiesiog norėjome jus supažindinti su programavimo metodika, kuri orientuota į programų kūrimą jas nuolat tikrinant. Naudokite savo malonumui! :)

Taigi, suprogramuokime kvadratinės lygties sprendimo programą.

1 žingsnis - padarome pirmąjį testą ir minimalią funkciją, kuri neveikia!

```
 1. import math
 2.
 3. # Gražina x1 ir x2 šaknis, jei nėra - None
 4. def skaičiuoti_šaknis(a, b, c):
 5.     return None, None
 6.
 7. def testuok():
 8.     # Kai diskriminantas > 0
 9.     x1, x2 = skaičiuoti_šaknis(1, 3, 2)
10.     print("1:", x1, x2)
11.
12. testuok()
```

Įvykdykite šią programą! Kaip matote - ji negali veikti, nes tik gražina reikšmes None bet kuriuo atveju, bet mes jau turime testą, kuris tai gali nustatyti :)

2 žingsnis - padarome tiek programos, kad ji veiktų su duotu testu!

```
 1. import math
 2.
 3. # Gražina x1 ir x2 šaknis, jei nėra - None
 4. def skaičiuoti_šaknis(a, b, c):
 5.     diskriminantas = b**2 - 4*a*c
 6.     x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
 7.     x2 = (-b - math.sqrt(diskriminantas)) / (2*a)
 8.     return x1, x2
 9.
10. def testuok():
11.     # Kai diskriminantas > 0
12.     x1, x2 = skaičiuoti_šaknis(1, 3, 2)
13.     print("1:", x1, x2)  # Tikėtinas rezultatas: 1: -1.0 -2.0
14.
15. testuok()
```

Įvykdykite. Matote - programa veikia! Blogai - mūsų kaip testuotojų ar hakerių tikslas - padaryti, kad ši programa neveiktų!

3 žingsnis - Pridedam naują testą:

```
import math

# Gražina x1 ir x2 šaknis, jei nėra - None
def skaičiuoti_šaknis(a, b, c):
    diskriminantas = b**2 - 4*a*c
    x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
    x2 = (-b - math.sqrt(diskriminantas)) / (2*a)
    return x1, x2

def testuok():
    # Kai diskriminantas > 0
    x1, x2 = skaičiuoti_šaknis(1, 3, 2)
    print("1:", x1, x2)  # Tikėtinas rezultatas: 1: -1.0 -2.0
    x1, x2 = skaičiuoti_šaknis(1, 2, 1)
    print("1:", x1, x2)  # Tikėtinas rezultatas: 2: -1.0 None

testuok()
```

Aha - mums pasisekė! Vietoj to, kad programa gražintų `-1.0 None`, gražina `-1.0 -1.0`.

4 žingsnis - taigi, taisome mūsų programą:

```
import math

# Gražina x1 ir x2 šaknis, jei nėra - None
def skaičiuoti_šaknis(a, b, c):
    diskriminantas = b**2 - 4*a*c
    if diskriminantas > 0:
        x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
        x2 = (-b - math.sqrt(diskriminantas)) / (2*a)
    elif diskriminantas == 0:
        x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
        x2 = None
    return x1, x2

def testuok():
    # Kai diskriminantas > 0
    x1, x2 = skaičiuoti_šaknis(1, 3, 2)
    print("1:", x1, x2)  # Tikėtinas rezultatas: 1: -1.0 -2.0
    # Kai diskriminantas == 0
    x1, x2 = skaičiuoti_šaknis(1, 2, 1)
    print("2:", x1, x2)  # Tikėtinas rezultatas: 2: -1.0 None

testuok()
```

Mūsų programa jau veikia!

5 žingsnis - bandom iš naujo ją įveikti!

```
 1. import math
 2.
 3. # Gražina x1 ir x2 šaknis, jei nėra - None
 4. def skaičiuoti_šaknis(a, b, c):
 5.     diskriminantas = b**2 - 4*a*c
 6.     if diskriminantas > 0:
 7.         x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
 8.         x2 = (-b - math.sqrt(diskriminantas)) / (2*a)
 9.     elif diskriminantas == 0:
10.         x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
11.         x2 = None
12.     return x1, x2
13.
14. def testuok():
15.     # Kai diskriminantas > 0
16.     x1, x2 = skaičiuoti_šaknis(1, 3, 2)
17.     print("1:", x1, x2)  # Tikėtinas rezultatas: 1: -1.0 -2.0
18.     # Kai diskriminantas == 0
19.     x1, x2 = skaičiuoti_šaknis(1, 2, 1)
20.     print("2:", x1, x2)  # Tikėtinas rezultatas: 2: -1.0 None
21.     # Kai diskriminantas < 0
22.     x1, x2 = skaičiuoti_šaknis(3, 1, 4)
23.     print("3:", x1, x2)  # Tikėtinas rezultatas: 3: None None
24.
25. testuok()
```

Na va, mūsų programa net nesuveikia. Nulūžta! Jėga! Naujas iššūkis mums kaip programuotojams.

6 žingsnis - Taisome programą.

```
 1. import math
 2.
 3. # Gražina x1 ir x2 šaknis, jei nėra - None
 4. def skaičiuoti_šaknis(a, b, c):
 5.     diskriminantas = b**2 - 4*a*c
 6.     if diskriminantas > 0:
 7.         x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
 8.         x2 = (-b - math.sqrt(diskriminantas)) / (2*a)
 9.     elif diskriminantas == 0:
10.         x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
11.         x2 = None
12.     else:
13.         x1 = None
14.         x2 = None
15.     return x1, x2
16.
17. def testuok():
18.     # Kai diskriminantas > 0
19.     x1, x2 = skaičiuoti_šaknis(1, 3, 2)
20.     print("1:", x1, x2)  # Tikėtinas rezultatas: 1: -1.0 -2.0
21.     # Kai diskriminantas == 0
22.     x1, x2 = skaičiuoti_šaknis(1, 2, 1)
23.     print("2:", x1, x2)  # Tikėtinas rezultatas: 1: -1.0 None
24.     # Kai diskriminantas < 0
25.     x1, x2 = skaičiuoti_šaknis(3, 1, 4)
26.     print("3:", x1, x2)  # Tikėtinas rezultatas: 1: None None
27.
28. testuok()
```

Programa pataisyta ir veikia kaip numatyta! Negi tai viskas? Ne, tikrai ne! Pamėginam nulaužti programą, matydami, kad galima dalyba iš 0.

7 žingsnis - Rašome naują testą.

```
 1. import math
 2.
 3. # Gražina x1 ir x2 šaknis, jei nėra - None
 4. def skaičiuoti_šaknis(a, b, c):
 5.     diskriminantas = b**2 - 4*a*c
 6.     if diskriminantas > 0:
 7.         x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
 8.         x2 = (-b - math.sqrt(diskriminantas)) / (2*a)
 9.     elif diskriminantas == 0:
10.         x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
11.         x2 = None
12.     else:
13.         x1 = None
14.         x2 = None
15.     return x1, x2
16.
17. def testuok():
18.     # Kai diskriminantas > 0
19.     x1, x2 = skaičiuoti_šaknis(1, 3, 2)
20.     print("1:", x1, x2)  # Tikėtinas rezultatas: 1: -1.0 -2.0
21.     # Kai diskriminantas == 0
22.     x1, x2 = skaičiuoti_šaknis(1, 2, 1)
23.     print("2:", x1, x2)  # Tikėtinas rezultatas: 2: -1.0 None
24.     # Kai diskriminantas < 0
25.     x1, x2 = skaičiuoti_šaknis(3, 1, 4)
26.     print("3:", x1, x2)  # Tikėtinas rezultatas: 3: None None
27.     # Kai a = 0
28.     x1, x2 = skaičiuoti_šaknis(0, 1, 4)
29.     print("4:", x1, x2)  # Tikėtinas rezultatas: 4: None None
30.
31. testuok()
```

Pataisome programą:

```python
import math

# Gražina x1 ir x2 šaknis, jei nėra - None
def skaičiuoti_šaknis(a, b, c):
    if a == 0:
        return None, None
    diskriminantas = b**2 - 4*a*c
    if diskriminantas > 0:
        x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
        x2 = (-b - math.sqrt(diskriminantas)) / (2*a)
    elif diskriminantas == 0:
        x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
        x2 = None
    else:
        x1 = None
        x2 = None
    return x1, x2

def testuok():
    # Kai diskriminantas > 0
    x1, x2 = skaičiuoti_šaknis(1, 3, 2)
    print("1:", x1, x2)  # Tikėtinas rezultatas: 1: -1.0 -2.0
    # Kai diskriminantas == 0
    x1, x2 = skaičiuoti_šaknis(1, 2, 1)
    print("2:", x1, x2)  # Tikėtinas rezultatas: 2: -1.0 None
    # Kai diskriminantas < 0
    x1, x2 = skaičiuoti_šaknis(3, 1, 4)
    print("3:", x1, x2)  # Tikėtinas rezultatas: 3: None None
    # Kai a = 0
    x1, x2 = skaičiuoti_šaknis(0, 1, 4)
    print("4:", x1, x2)  # Tikėtinas rezultatas: 4: None None

testuok()
```

Labai jau čia įsismaginome. Taisydami ir taisydami prirašėme daug kodo, kurį galime optimizuoti.

8 žingsnis - Tai ir padarome! (optimizuojam programą)

```python
import math

# Gražina x1 ir x2 šaknis, jei nėra - None
def skaičiuoti_šaknis(a, b, c):
    if a == 0:
        return None, None
    x1, x2 = None, None
    diskriminantas = b**2 - 4*a*c
    if diskriminantas > 0:
        x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
        x2 = (-b - math.sqrt(diskriminantas)) / (2*a)
    elif diskriminantas == 0:
        x1 = (-b + math.sqrt(diskriminantas)) / (2*a)
    return x1, x2

def testuok():
    # Kai diskriminantas > 0
    x1, x2 = skaičiuoti_šaknis(1, 3, 2)
    print("1:", x1, x2)  # Tikėtinas rezultatas: 1: -1.0 -2.0
    # Kai diskriminantas == 0
    x1, x2 = skaičiuoti_šaknis(1, 2, 1)
    print("2:", x1, x2)  # Tikėtinas rezultatas: 2: -1.0 None
    # Kai diskriminantas < 0
    x1, x2 = skaičiuoti_šaknis(3, 1, 4)
    print("3:", x1, x2)  # Tikėtinas rezultatas: 3: None None
    # Kai a = 0
    x1, x2 = skaičiuoti_šaknis(0, 1, 4)
    print("4:", x1, x2)  # Tikėtinas rezultatas: 4: None None

testuok()
```

Pabandykite. Atkreipkite dėmesį, kad optimizuodami kodą galime drąsiau jaustis, nes testai žiūri ar mes kur nepaklydome :) Galima būtų tobulinti šį kodą ir toliau, bet čia sustosime. Matote, programos kūrimas vyksta tokiu būdu:

1. parašome testą, kuris nepraeina;
2. pataisome programą, kad testas praeitų;
3. optimizuojame kodą;
4. kartojam iš naujo.

Paprastas ir efektyvu! O kai kas sako, kad ir smagu :) O kaip jums?