# 31.2.3 Testavimas ir taisymas.

## Ką reikia mokėti

* Diskutuojama apie testavimo strategijas
* įprasti ir ekstremalūs duomenų rinkiniai
* Konstruojami pradiniai duomenų rinkiniai, mokomasi naudojantis šiais rinkiniais ieškoti klaidų ir jas taisyti

## Testavimo strategija

Programų inžinerijoje, **testavimo strategija** - tai planas, apibūdinantis, kaip bus testuojama programinė įranga jos kūrimo cikle. Testavimo strategijos tikslas - išsamiai aprašyti veiksmus, kurie padės pasiekti kokybės užtikrinimo tikslus. Testavimo strategija atskiriems projektams gali skirtis. Testavimo strategiją kuria ir aptaria visi projekto dalyviai, kad būtų išvengta nesusipratimų ir kad visi žinotų savo vaidmenis ir atsakomybes. Laikui bėgant strategija reguliariai peržiūrima ir atnaujinama, atsižvelgiant į organizacijos ir produkto pokyčius. Strategija padeda suvienyti skirtingus kokybės užtikrinimo srities dalyvius: suvienodina naudojamą terminologiją, nustato testavimo lygius ir planuoja išteklius.

Testavimo strategija **mokyklos lygyje** suprantama kaip apibrėžtas būdas, nusakantis kaip testuoti sukurtą programą, paprastai turinčią **įvestį** (duomenis) ir **išvestį** (rezultatus), neįtraukiant našumo, naudotojų sąsajos, naudotojų patirties ir kitų aspektų.

Kadangi mokyklos kurse neįtraukiamas automatizuotas testavimas, strategija iš esmės susiveda į duomenų rinkinių konstravimą, kuriuos naudojant bus ieškoma klaidų. Dažniausiai naudojami yra šie būdai: **ekvivalentinio sudalijimo**, **ribinių reikšmių** ir **ekstremalių reikšmių**.

Šie būdai ir bus pagrinde nagrinėjami šioje medžiagoje.

## Ekvivalentinis sudalijimas (angl. *Equivalence partitioning*)

**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 (iš uždavinių):

if temperatūra > 10 and temperatūra < 20:
 print("Vidutinė temperatūra")
elif temperatūra >= 20:
 print("Šilta diena")
else:
 print("Šalta diena")

Šiame programos fragmente matome sąlygos sakinį, kuriame yra nusakytas interavalas (10, 20). Vadinasi galime suskirstyti duomenų rinkinius į tokias **3** klases:

 ... 8 9 10 11 ............. 19 20 21 22 .....
 --------------|-------------------|---------------------
 šalta diena vidutinė temp. šalta diena

Taigi, pagal ekvivalentinio sudalijimo idėją, testavimui būtina paimti 3 *temperatūra* kintamojo reikšmes iš kiekvienos klasės. Tarkim: **8**, **15** ir **22**.

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.

*Šiuose uždaviniuose laikysime, kad ekvivalentinei klasei reikia parinkti bent vieną duomenų rinkinio egzempliorių, tačiau nebus klaida, nei parenkama daugiau.*

## Ribinių reikšmių analizė

**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 temperatūra > 10 and temperatūra < 20:
 print("Vidutinė temperatūra")
elif temperatūra >= 20:
 print("Šilta diena")
else:
 print("Šalta diena")

Šiame programos fragmente matome sąlygos sakinį, kuriame yra nusakytas interavalas (10, 20). Vadinasi, klasės pasikeitimas vyksta intervalo ribom:

 ... 8 9 10 11 ............. 19 20 21 22 .....
 --------------|-------------------|---------------------
 šalta diena vidutinė temp. šalta diena

Šiuo atveju reikia testuoti su šiomis kintamojo *temperatūra* reikšmėm: **10**, **11**, **19**, **20**. Jeigu tai skaičius su kableliu, kuriam skiriami du skaitmenys po kablelio, tuomet reiktų testuoti su reikšmėm: **10.99**, **10.00**, **19.99**, **20,00**.

## Ekstremalių reikšmių analizė

Šis metodas skirtas 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.

*Kadangi nėra tikslaus mato, kokia turi būti ekstremali reikšmė, šiuose uždaviniuose laikysime, kad ekstremali reikšmė turi bent viena eile būti didesnė/mažesnė už teisingų reikšmių intervalą arba kitų testinių duomenų reikšmes.*

Panagrinėkime pavyzdį, kuris neapsaugotas nuo klaidingo veikimo:

diskriminantas = b\*\*2 - 4\*a\*c
if diskriminantas > 0:
 root1 = (-b + math.sqrt(diskriminantas)) / (2\*a)
 root2 = (-b - math.sqrt(diskriminantas)) / (2\*a)
 print(f"Šaknys: x1 = {root1}, x2 = {root2}")

Šiame programos fragmente matome kvadratinės lygties diskriminanto skaičiavimą. Kintamojo *a* ekstremalių reikšmių pavyzdys būtų: **-1000000**, **-0.0000001**, **0**, **0.0000001**, **1000000**. Kaip matote, tarp ekstremalių reikšmių įtraukėme ir tas, kurios artimos **0**. Šios reikšmės leistų aptikti, kad kintamasis *a* negali būti lygus nuliui ir dėl to reiktų pridėti atatinkamą patikrinimą.

## Testavimo duomenys ir testavimo atvejai

**Testavimo duomenys** - tai specifiniai duomenys (vertės ar įvestys), naudojami programinės įrangos veikimui patikrinti. Testavimo duomenys gali būti skirtingų tipų, priklausomai nuo programos, pvz., skaičiai, tekstai, failai. Kai turime **tik** testavimo duomenis, programos veikimą turime tikrinti patys.

**Testavimo atvejai** - tai testavimo duomenys kartu su tikėtinais rezultatais, kurie turėtų būti gaunami įvykdžius programą su testavimo duomenimis. Testavimo atvejai leidžia automatizuoti programos patikrinimą.

## Testavimo metodologijos pagal žinias apie sistemos struktūrą

### Juodosios dėžės testavimas

Testavimo metodas, kai testuotojas neturi informacijos apie sistemos vidinį veikimą ir analizuoja tik jos išorinius veikimo aspektus. Tikrinama, ar sistema tinkamai reaguoja į įvairias įvestis ir ar pateikia teisingus rezultatus. Šis metodas dažnai naudojamas funkciniam testavimui. Tinka visos aptartos strategijos: ribinių ir ekstremalių reikšmių analizė, ekvivalentinio sudalijimo metodas.

### Baltosios dėžės testavimas

Testavimo metodas, kai testuotojas turi prieigą prie sistemos vidinio kodo ir struktūros. Tikrinama, kaip kiekviena kodo dalis veikia, ir ieškoma klaidų kodo lygmenyje. Šis metodas dažnai naudojamas vienetų testavimui. Atlikus kodo analizę gali būti taikomos aptartos strategijos.

### Pilkosios dėžės testavimas

Tai mišrus testavimo metodas, kai testuotojas turi ribotą prieigą prie sistemos vidinio veikimo (pvz., pagrindinių struktūrų ar duomenų bazės). Naudojamas funkciniams testams atlikti kartu su pagrindinėmis techninėmis žiniomis apie sistemą.

## Testavimo lygiai

Testavimo lygiai yra skirtingi programinės įrangos testavimo būdai, atliekami skirtinguose programos kūrimo žingsniuose, siekiant užtikrinti kokybę kiekviename etape. Kiekvienas lygis skirtas tam tikram tikslui ir padeda testuotojams patikrinti skirtingus sistemos aspektus.

## Vienetų testavimas (angl. *Unit testing*)

Testavimo lygis, kuriame atskiri programos komponentai arba „vienetai“ (pvz., funkcijos, metodai ar moduliai) yra testuojami atskirai. Kai kurdami programą iškviečiam vieną funkciją su tam tikrais parametrais ir pasitikrinam jos rezultatus - tai vienetų testavimas. Paprastai jis atliekamas automatizuotai.

## Integracinis testavimas (angl. *Integration testing*)

Testavimo lygis, kuriame testuojama, kaip skirtingi programinės įrangos komponentai veikia kartu. Tai padeda nustatyti, ar nėra klaidų komponentų sąveikoje. Kai mes tikriname programą, kuri nuskaito duomenis iš failo ir atlieka skaičiavimus - tai jau integracinis testavimas.

## Sistemos testavimas (angl. *System testing*)

Testavimo lygis, kuriame mes tikriname visos programos veikimą, įtraukiant ir naudotojo sąsają. Šie testai gali būti automatizuoti, tačiau dažnai atliekami rankiniu būdu, nes kai kuriose taikomosiose srityse automatizuoti visos sitemos patikrinimą yra sudėtinga.

## Programos funkcijų patikra

Čia kalbama apie patikrą, kuri atliekama prieš priduodant (arba - priduodant) progamą.

### Funkcinis testavimas

**Funkcinis testavimas** yra skirtas patikrinti ar programa teisingai veikia pagal iškeltus funkcinius reikalavimus. Į funkcinį testavimą įeina vienetų, integraciniai, sistemo testai ir kiti testavimo būdai, kurie tikrina sistemos veikimą pagal jos specifikaciją.

### Priėmimo testavimas

**Priėmimo testavimas** yra skirtas patikrinti ar sukurta teisinga programa. Ar programa daro tai ko reikia naudotojui. Gali kilti klausimas - jei sistema korektiškai veikia pagal funkcinius reikalavimus, ar ji gali būti neteisinga. Taip, vienas iš variantų - klaidos padarytos specifikuojant (aprašant) būsimą programą. Priėmimo testavimas dažniausiai atliekamas rankiniu būdu, kartu su programos savininku.

## Kitos testavimo metodologijos

### Regresinis testavimas

**Regresinis testavimas** - tai testavimas, kuris atliekamas po to, kai programinė įranga buvo patobulinta, pakeista arba ištaisyta. Pagrindinis regresinio testavimo tikslas yra patikrinti, ar nauji pakeitimai nesukėlė neigiamų pasekmių ir nesugadino anksčiau veikusių sistemos dalių. Todėl testavimo duomenys yra svarbūs ne tik programos kūrimo metu.

### Našumo testavimas

**Našumo testavimas** - tai testavimas, skirtas įvertinti, kaip sistema veikia esant skirtingoms apkrovoms, ir nustatyti jos galimybes, kai naudojama realiomis sąlygomis. Tai yra, kai ją naudoja daug vartotojų, kai atliekamos sudėtingos užklausos arba atliekami sudėtingi skaičiavimai. Našumas apima net tik programos geritį, bet ir reikalavimus atminčiai, tinklui ir kt. Su našumo testavimu susijęs **stresinis testavimas**, kurio tikslas - nustatyti programos našumo ribas ir atsparumą.

### Scenarijų testavimas

**Scenarijų testavimas** - tai testavimo metodas, kurio metu kuriami realistiniai scenarijai (veiksmų sekos), atspindintys, kaip vartotojas naudotų programinę įrangą. Šio metodo tikslas - patikrinti, kaip sistema veikia imituojant darbo su programa procesus. Dažnai naudojamas priėmimo testavimo metu.

## Testais paremtas programų kūrimas (angl. *Test driven development*)

Tai programavimo metodas, kai programos kūrimas prasideda nuo testų rašymo (testinių duomenų rinkinių kūrimo), o tik tada pereinama prie funkcionalumo programavimo. Pagrindinis tikslas - užtikrinti, kad kiekviena parašyta kodo dalis atitinka reikalavimus ir veikia teisingai.

Tai įdomus programos kūrimo būdas, kurį galima išmėginti su moksleiviais. Kaip pavyzdys, galėtų būti tokie būtų programos kūrimo žingsniai:

* Sukurkite duomenų failą
* Įgyvendinkite duomenų nuskaitymą
* Sukurkite minimalią skaičiavimų funkciją, kad pradiniai testai (duomenys) praeitų
* Patobulinkite kodą (kodas optimizuojamas, kad būtų kuo paprastesnis, efektyvesnis ir patikimesnis).
* Iteruokite (pridėkite daugiau testinių duomenų, kurie aptiktų paliktas klaidas; tobulinkite programą, kad testai praeitų).

## Kodo, šakų padengimas

Kodo padengimas (angl. *Code coverage*) - tai metrika, kuri matuoja, kokia dalis programinio kodo buvo patikrinta testavimo metu. Kodo padengimo analizė padeda nustatyti, kurios kodo dalys buvo išbandytos, ir nustatyti tas, kurios dar neturi testų.

Kodo padengimo tipai:

* Eilučių padengimas (angl. *Line coverage*): matuoja, kiek kodo eilučių buvo vykdyta testavimo metu. Jei kodo eilutė nebuvo vykdyta nė su vienais testiniais duomenimis, ji laikoma nepadengta.
* Šakų padengimas (angl. *Branch coverage*): matuoja, ar visos loginės šakos (pvz., sąlygos, kaip *if*, *else*, *elif/else if*, *switch*) buvo vykdytos. Kitaip sakant, tikrinama, ar įvykdomas kiekvienas sąlygos blokas.
* Sąlygų padengimas (angl. *Condition coverage*): kiekviena loginė sąlyga tikrinama atskirai, siekiant užtikrinti, kad buvo patikrinti visi sąlygos rezultatai (pvz., tikrinama tiek true, tiek false rezultatai). Sąlygų padengimas paprastai įtraukia daugiau testų nei šakų padengimas, nes *if* sakinys gali įtraukti kelias sąlygas.

*Testuose įtraukti šakų padengimo ir sąlygų padengimo uždaviniai.*