Kauno technologijos universitetas
Informatikos fakultetas

Projekto pavadinimas

Baigiamasis bakalauro studijų projektas

Vardenis Pavardenis
Projekto autorius

prof. Vardas Pavardė
Vadovas

Kaunas, 2024

Kauno technologijos universitetas
Informatikos fakultetas

Projekto pavadinimas

Baigiamasis bakalauro studijų projektas
Programų sistemos (6121BX012)

Vardenis Pavardenis
Projekto autorius

prof. Vardas Pavardė
Vadovas

prof. Vardenė Pavardenė
Recenzentė

Kaunas, 2024

Kauno technologijos universitetas
Informatikos fakultetas
Vardenis Pavardenis

Projekto pavadinimas

Akademinio sąžiningumo deklaracija

Patvirtinu, kad:

1. baigiamąjį projektą parengiau savarankiškai ir sąžiningai, nepažeisdama(s) kitų asmenų autoriaus ar kitų teisių, laikydamasi(s) Lietuvos Respublikos autorių teisių ir gretutinių teisių įstatymo nuostatų, Kauno technologijos universiteto (toliau – Universitetas) intelektinės nuosavybės valdymo ir perdavimo nuostatų bei Universiteto akademinės etikos kodekse nustatytų etikos reikalavimų;
2. baigiamajame projekte visi pateikti duomenys ir tyrimų rezultatai yra teisingi ir gauti teisėtai, nei viena šio projekto dalis nėra plagijuota nuo jokių spausdintinių ar elektroninių šaltinių, visos baigiamojo projekto tekste pateiktos citatos ir nuorodos yra nurodytos literatūros sąraše;
3. įstatymų nenumatytų piniginių sumų už baigiamąjį projektą ar jo dalis niekam nesu mokėjęs (-usi);
4. suprantu, kad išaiškėjus nesąžiningumo ar kitų asmenų teisių pažeidimo faktui, man bus taikomos akademinės nuobaudos pagal Universitete galiojančią tvarką ir būsiu pašalinta(s) iš Universiteto, o baigiamasis projektas gali būti pateiktas Akademinės etikos ir procedūrų kontrolieriaus tarnybai nagrinėjant galimą akademinės etikos pažeidimą.

Vardenis Pavardenis
Patvirtinta elektroniniu būdu

Vardenis Pavardenis. Projekto pavadinimas. Baigiamasis bakalauro studijų projektas. Vadovas prof. Vardas Pavardė. Informatikos fakultetas, Kauno technologijos universitetas.

Studijų kryptis ir sritis: Informatikos mokslai, Programų sistemos.

Reikšminiai žodžiai: Raktažodis1, Raktažodis2 Raktažodis3.

Kaunas, 2024. 14 p.

Santrauka

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere.

Vardenis Pavardenis. Project Title. Bachelor's Final Degree Project. Supervisor prof. Vardas Pavardė. Faculty of Informatics, Kaunas University of Technology.

Study field and area: Computer Sciences, Software Systems.

Keywords: Keyword1, Keuword2, Keyword3, etc.

Kaunas, 2024. 14 pages.

Summary

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere.

Turinys

Lentelių sąrašas

Paveikslų sąrašas

Santrumpų ir terminų sąrašas

Santrumpos:
Doc. – docentas;
Lekt. – lektorius;
Prof. – profesorius.

Terminai:
Saityno analitika – lorem ipsum dolor sit amet, eam ex decore persequeris, sit at illud lobortis atomorum. Sed dolorem quaerendum ne, prompta instructior ne pri. Et mel partiendo suscipiantur, docendi abhorreant ea sit. Recteque imperdiet eum te.
Tinklaraštis – lorem ipsum dolor sit amet, eam ex decore persequeris, sit at illud lobortis atomorum. Sed dolorem quaerendum ne, prompta instructior ne pri. Et mel partiendo suscipiantur, docendi abhorreant ea sit. Recteque imperdiet eum te.

1 First section

1.1 Subsection

S	A	T	O	R
A	R	E	P	O
T	E	N	E	T
O	P	E	R	A
R	O	T	A	S

Table 1: Dummy table

S	A	T	O	R
A	R	E	P	O
T	E	N	E	T
O	P	E	R	A
R	O	T	A	S

Table 2: Long caption: Dummy table Dummy table Dummy table Dummy table Dummy table Dummy table Dummy table Dummy table Dummy table

Figure 1: KTU logo

2 Kodėl „Scala“?

2.1 Įvadas

Programavimo kalbos yra pagrindinis tarpininkas tarp žmogiškos logikos ir mašininio kodo - jos leidžia programuotojams paversti abstrakčias idėjas ir problemų sprendimo metodus į instrukcijas, kurias kompiuteriai gali vykdyti. Tinkamos programavimo kalbos pasirinkimas projektui yra kritinis sprendimas, kuris daro įtaką kūrimo efektyvumui, sistemos veikimui, priežiūrai ir galiausiai projekto sėkmei. Šiame skyriuje pateikiamas kontekstas „Scala“ pasirinkimui kaip šios disertacijos įgyvendinimo kalbai, nagrinėjant platesnį programavimo kalbų kraštovaizdį, jų evoliuciją ir įvairias paradigmas, kurias jos atstovauja.

2.2 Istorinė programavimo kalbų raida

Manome, jog galima išskirti kelis tipus programavimo kalbų, priklausomai nuo jų sukūrimo laiko bei paskirties.

2.2.1 Mašininis kodas

Ankstyviausi kompiuteriai reikalavo programavimo dvejetainiu mašininiu kodu – 1 ir 0 sekomis, tiesiogiai atitinkančiomis procesoriaus instrukcijas. Šis metodas, nors ir tiesiogiai vykdomas aparatinės įrangos, žmogui programuotojui buvo labai varginantis ir linkęs į klaidas.

Taigi šis programavimo metodas yra sudėtingas, juo parašytas programinis kodas yra praktiškai kalbant neįskaitomas, bei jį naudojant yra labai lengva padaryti klaidų. Ar yra bent vienas tikslas

programuoti mašininiu kodu? Teoriškai, taip - mašininis kodas suteikia programuotojui patiesžemiausio lygio prieigą prie procesoriaus instrukcijų. Tai leidžia patyrusiam programuotojui pasiektididesnę greitaveiką, tikslesnį veikimą bei mažesnį galutinio failo dydį. Visa tai yra labai naudingaspecifinėse situacijose, kai resursai yra ypatingai riboti. Tačiau net tokiu atveju, dauguma profesionalų rinktūsi įrankį, suteikiantį šiek tiek daugiau abstrakcijos.

2.2.2 Asemblerio kalbos

Asemblerio kalbos pristatė simbolinius mašininių instrukcijų atvaizdavimus, leidžiančius programuotojams naudoti žmogui suprantamą kodą vietoje dvejetainių procesoriaus instruktijų. Nors vis dar glaudžiai susietos su aparatinės įrangos architektūra, asemblerio kalbos buvo pirmoji abstrakcija nuo mašininio kodo.

Šiais laikais kompiuteriai yra taip stipriai pažengę, jog programinio kodo rašymas Asemblerio kalbomis dažniausiai yra visiškai nepraktiškas sprendimas. Kai egzistuoja tiek daug aukštesnio lygio kalbų, Asemblerio kalbos naudojamos tik esant labai griežtiems resursų ir greičio reikalavimams – panašiai kaip mašininis kodas. Vienas pavyzdys tokio panaudojimo – „Apollo-11“ orientacinio kompiuterio programinis kodas (citata https://github.com/chrislgarry/Apollo-11). Šis kompiuteris turėjo labai ribotą atminties kiekį - 2048 žodžius atsitiktinės prieigos atminties bei 36 864 žodžius pagrindinės atminties (citata https://en.wikipedia.org/wiki/Apollo_Guidance_Computer). Taip pat, žinoma, kompiuterio procesorius lyginant su šių dienų standartais buvo ypatingai silpnas, dėl ko reikėjo parašyti patį optimaliausią kodą.

Žinoma, „Apollo-11“ skrydžio laikais nebuvo daug alternatyvų Asemblerio kalboms, tačiau ir šiomis dienomis jos vis dar yra naudojamos operacinių sistemų branduliuose, realaus laiko programose, įrenginių tvarkyklėse ir kitose programose, kur greitis ir resursų valdymas yra kritinis taškas.

2.2.3 Ankstyvosios aukšto lygio kalbos

Šešto dešimtmečio pabaigoje ir septinto dešimtmečio pradžioje įvyko proveržis programavimo kalbų srityje – buvo sukurtos pirmosios tuo metu vadinamos aukšto lygio kalbos: „FORTRAN“, „COBOL“, „LISP“ ir „ALGOL“. Šios kalbos pristatė revoliucinį pokytį programavimo procesuose, nes jos leido programuotojams:

- Rašyti kodą, kuris buvo nepriklausomas nuo konkretaus kompiuterio architektūros
- Naudoti abstrakčias matematines išraiškas vietoj procesoriaus instrukcijų
- Struktūrizuoti programas į funkcijas ir procedūras
- Kurti programas, kurios buvo žymiai lengviau skaitomos ir suprantamos žmonėms

„FORTRAN“ (angl. *Formula Translation*) buvo sukurta moksliniams skaičiavimams ir tapo pirmąja plačiai naudojama aukšto lygio kalba. Ji leido mokslininkams ir inžinieriams rašyti programas matematinėmis formulėmis, o ne mašininėmis instrukcijomis. „COBOL“ (angl. *Common Business-Oriented Language*) buvo sukurta verslo aplikacijoms ir pasižymėjo itin skaitoma angliška sintakse. Ji buvo specialiai sukurta taip, kad netechninio išsilavinimo žmonės galėtų skaityti ir suprasti programinį kodą. Nepaisant savo amžiaus, „COBOL“ vis dar naudojama kai kuriose finansų ir vyriausybinėse sistemose. „LISP“ (angl. *List Processing*) buvo sukurta dirbtinio intelekto tyrimams ir įvedė tokias koncepcijas kaip rekursija, dinaminis tipizavimas ir automatinis atminties valdymas. Ji buvo pirmoji funkcinė programavimo kalba ir turėjo didžiulę įtaką vėlesnėms programavimo kalboms. „ALGOL“ (angl. *Algorithmic Language*) buvo sukurta kaip universali algoritmų aprašymo kalba. Ji įvedė blokų struktūrą, lokalius kintamuosius ir procedūras su parametrais. „ALGOL“ tapo daugelio vėlesnių kalbų, tokių kaip „Pascal“, „C“ ir „Java“ protėviu.

2.2.4 „C“ kalba ir sisteminės kalbos

„C“ kalba, sukurta „Bell“ laboratorijose 1972 metais (https://www.geeksforgeeks.org/c-language-introduction/), tapo viena įtakingiausių programavimo kalbų istorijoje. Ji užėmė unikalią nišą tarp žemo lygio asemblerio kalbų ir aukšto lygio kalbų, siūlydama išskirtinį balansą tarp efektyvumo ir abstrakcijos. „C“ buvo sukurta „UNIX“ operacinei sistemai kurti ir greitai tapo standartu sisteminiam programavimui. Ji suteikė programuotojams galimybę tiesiogiai manipuliuoti kompiuterio atmintimi naudojant rodykles, bet tuo pačiu siūlė struktūrinę sintaksę ir modulinę struktūrą. C kalba pasižymėjo perkeliamumu – programos, parašytos viename kompiuteryje, galėjo būti nesunkiai adaptuotos kitam, kas buvo revoliucinis pokytis to meto kontekste. Daugelis šiuolaikinių operacinių sistemų, įskaitant „Linux“ ir „Windows“, yra parašytos „C“ kalba, o jos įtaka matoma beveik visose vėlesnėse programavimo kalbose, įskaitant „C++“, „Java“, „C#“ ir net „Python“.

2.2.5 Modernios kalbos

Šiais laikais programavimo kalbų pasirinkimas yra beveik begalinis. Yra įvairiausių kalbų visokioms problemoms spręsti. Interpretuojamos kalbos kaip „Python“ idealiai tinka lengvai suprantamiems, greitai parašomiems scenarijams. „Java“, „C#“ ir kitos panašios aukšto lygio objektinės kalbos pasižymi savo tipų saugumu ir skalabilumu didelės apimpties programose. „Rust“ ir „Zig“ yra puikios modernios alternatyvos sistemų programavimo standartui „C“. Turint tiek daug pasirinkimo laisvės, renkantis programavimo kalbą galima daugiau galvoti apie jos stilių bei abstrakcijos lygį.

2.3 Kalbos rinkimasis

2.3.1 Abstrakcijos lygmuo

Renkantis programavimo kalbą svarbu nuspręsti, kiek žemo lygio kontrolės reikės mūsų kuriamam projektui. Pavyzdžiui, jei pasirinksime tai, ką šiais laikais vadintume žemo lygio kalbomis, kaip „C“ ar „Rust“, galėtume daug atidžiau kontroliuoti visus programos veikimo niuansus, bet tai reikalautų daug daugiau laiko bei didesnio programinio kodo kiekio, taip pat didintų kodo sudėtingumą. Aukšto lygio kalba kaip „Java“ paspartintų programos kūrimą, nes aukšto lygio kalbose paprastai nereikia pačiam programuotojui valdyti atminties, jose būna daug įskiepių, kurie gali padėti išspręsti įvairias problemas, bei kodo sudėtingumas dažniausiai būna žymiai maženis.

Mūsų projektas šiuo atveju yra pakankamai lankstus – komandinės eilutės programą tikrai galima rašyti ir aukšto, ir žemo lygio kalbomis. Šiam projektui nėra skirta jokių griežtų greičio ar apimties apribojimų, todėl pasirinkome naudoti aukštesnio lygio kalbą, kad programinio kodo rašymo metu būtų galima daugiau dėmėsio telkti programos funkcionalumui.

2.3.2 Kompiliuojama ar interpretuojama kalba?

Programavimo kalbos paprastai yra skirstomos į 2 pagrindinius tipus priklausomai nuo to, kaip jų kodas yra paleidžiamas:

- Kompiliuojamos kalbos - programinis kodas yra paverčiamas mašininiu (arba kokiu nors

tarpiniu kodu, kuris po to verčiamas mašininiu, kaip „Java Virtual Machine“). To rezultatas - ilgesnis programos paleidimas programuojant, bet greitesnis veikimas, nes kompiliatorius gali optimizuoti mašininį kodą prieš jo įvykdymą. Taip pat dauguma sintaksės ar kitokių klaidų aptinkama prieš programos paleidimą, kompiliavimo metu.
- Interpretuojamos kalbos - programinis kodas yra vykdomas eilutė po eilutės, iš eilės,

nėra jokio tarpinio žingsnio tarp kodo parašymo ir paleidimo. Tai puikiai tinka įvairiems scenarijams (angl. *scripts*), tačiau stipriai nukenčia programos greitąveika.

Siekdami neprarasti per daug programos veikimo spartumo, nusprendėme pasirinkti kompiliuojąmą programavimo kalbą.

2.3.3 Statiniai ar dinaminiai tipai?

Programavimo kalbos yra skirstomos į 2 pagrindines grupes pagal tai, kaip jos kontroliuoja kintamųjų tipus:

- Statiniai tipai - kiekviena reikšmė ar kintamasis programiniame kode turi savo tipą (*int*, *char* ir t.t.), tas tipas negali keistis programos eigoje. Tai suteikia savotinio saugumo, neleidžia programuotojui daryti žmogiškų klaidų. Taip pat turint statinę tipų sistemą, galima kurti savo tipus, taip pridedant dar daugiau saugumo, pavyzdžiui:

  ```
  def doSomething(name: String, surname: String) = ()
  doSometing("pavardenis", "vardenis")
  ```

  Matome, kad galime iškviesti funkciją *doSomething* įvedę vardą ir pavardę apkeistus vietomis. Tačiau, jei sukurtume savo tipus vardui ir pavardei, to būtų galima išvengti:

  ```
  case class Name(value: String)
  case class Surname(value: String)
  def doSomething(name: Name, surname: Surname) = ()
  doSometing(Surname("pavardenis"), Name("vardenis"))
  ```

  Šiuo atveju kompiliavimo metu matytume klaidą, kuri išgelbėtų mus nuo atsitiktinio funkcijos argumentų sumaišymo.

- Dinaminiai tipai - kiekvienos reikšmės ar kintamojo tipas gali kisti programos vykdymo metu, pavydžiui:

  ```
  some_value = "text"
  some_value = 123
  ```

  Kalba su dinaminiais tipais leistų atlikti tokį reikšmės pakeitimą. Tai gali būti pravartu nišinėse situacijose, tačiau didelės apimties programoje toks programavimo stilius sukelia riziką padaryti daugybę klaidų, kurias vėliau yra labai sunku surasti.

Mūsų programos apimtis bus sąlyginai didelė, todėl mes pasirinkome nudoti kalbą su statiniais tipais.

2.3.4 Programavimo paradigma

Robert Cecil Martin savo knygoje „Clean Architecture“ (citata) išskiria tris pagrindines programavimo paradigmas: struktūrinis, objektinis bei funkcinis programavimas. Pagal autorių, kiekviena paradigma ne suteikia mums kažką, o priešingai -jos atima galimybę iš programuotojų rašyti kodą, kuris lengvai priveda prie klaidų.

- „Pirmoji priimta (bet ne pirmoji išrasta) paradigma buvo struktūrinis programavimas, kurį 1968 m. atrado Edsger Wybe Dijkstra. Dijkstra įrodė, kad nevaržomų šuolių (angl. *goto* teiginių) naudojimas yra žalingas programos struktūrai. (...) šiuos šuolius pakeitė geriau pažįstamomis konstrukcijomis *if/then/else* ir *do/while/until*.

  Struktūrinio programavimo paradigmą galima apibendrinti taip: Struktūrinis programavimas nustato tiesioginio valdymo perdavimo drausmę.“

- „Antroji priimta paradigma iš tikrųjų buvo atrasta dvejais metais anksčiau, t.y. 1966 m. Ole Johano Dahlio ir Kristeno Nygaardo. Šie du programuotojai pastebėjo, kad „AGOL“ kalbos funkcijų iškvietimo dėklo (angl. *stack*) rėmelį galima perkelti į krūvą (angl. *heap*), taip sudarant galimybę funkcijos deklaruotiems vietiniams kintamiesiems egzistuoti ilgą laiką po to, kai funkcijos reikšmė buvo grąžinta. Funkcija tapo klasės konstruktoriumi, o vietiniai kintamieji tapo egzemplioriaus kintamaisiais, o įterptinės funkcijos - metodais. Tai neišvengiamai privedė prie polimorfizmo atradimo disciplinuotai naudojant funkcijų rodykles.

Objektinio programavimo paradigmą galima apibendrinti taip: Objektinis programavimas programavimas įveda drausmę netiesioginiam valdymo perdavimui.“

- „Trečioji paradigma, kuri tik neseniai pradėta taikyti, buvo pirmoji. išrasta. Iš tiesų ji buvo išrasta anksčiau nei pats kompiuterių programavimas. Funkcinis programavimas yra tiesioginis rezultatas Alonzo Čerčo darbo, kuris 1936 m. išrado lambda integralinį ir differencialinį skaičiavimą (angl. *lambda calculus*), spręsdamas tą pačią matematinę problemą, kuri buvo tuo pat metu motyvavo Alaną Tiuringą.“

  Autorius toliau aiškina, jog pagrindinė *lambda calculus* sąvoka yra nekintamumas, t. y. nuostata, kad simbolių reikšmės nesikeičia. Tai reiškia, kad funkcinėje kalboje nėra priskyrimo teiginio. Realybėje kartais yra sunku apsieiti be vertės keitimo, todėl: „Dauguma funkcinių kalbų iš tikrųjų turi tam tikrų priemonių kintamojo vertei keisti, bet tačiau tik labai griežtai laikantis drausmės.“

  Funkcinio programavimo paradigmą galima autorius apibendrina taip: „Funkcinis programavimas nustato priskyrimo discipliną.“

Funkcinis programavimas mums ypač pasirodė įdomus, nes matematinio stiliaus kodas be reikšmių keitimo ne tik padeda išvengti sudėtingo bei klaidingo kodo, bet dažniausiai ir padeda tą pačią problemą išspręsti greičiau ir suprantamiau. Dėl šios priežasties savo programai kurti pasirinkome funkcinio stiliaus kalbą. Detaliau apie funkcinį programavimą ir jo privalumus kalbėsime tolimesniuose skyriuose.