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. 27 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. 27 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.

First section

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

Įvadas

Darbo problematika ir aktualumas

Šiuolaikiniame technologijų pasaulyje gatvių vaizdai ir geografinė informacija paprastai pateikiami per grafines sąsajas, tačiau komandinės eilutės (angl. *Command line interface*) aplinka išlieka svarbi daugeliui informacinių technologijų profesionalų ir entuziastų. Šio darbo problematika kyla iš smalsumo ir noro ištirti naujas komandinės eilutės pritaikymo galimybes - ar įmanoma sukurti interaktyvią sąsają gatvės lygio vaizdams, ir jei taip, ar tokia sąsaja gali būti intuityvi ir patogi naudoti. Tai yra ne tik techninio įgyvendinamumo klausimas, bet ir žmogaus-kompiuterio sąveikos tyrimas neįprastoje aplinkoje.

Projekto aktualumas pasireiškia kaip alternatyvių sąsajų tyrinėjimas ir kūrybinis eksperimentas, praplečiantis komandinės eilutės galimybes. Tokia sąsaja galėtų būti įdomi programuotojams, sistemų administratoriams ir kitiems specialistams, kurie daug laiko praleidžia terminalo aplinkoje ir vertina galimybę greitai pasiekti informaciją nepaliekant šios aplinkos. Be to, projektas atskleidžia ASCII stiliaus meno potencialą perteikti sudėtingą vaizdinę informaciją ir kelia klausimus apie tai, kaip skirtingos sąsajos formos veikia mūsų suvokimą ir sąveiką su geografine informacija.

Projektas apima kompiuterių mokslų, žmogaus-kompiuterio sąveikos ir kūrybinių technologijų sritis. Praktinė darbo reikšmė slypi ne tik galimame sukurto įrankio naudojime, bet ir naujų idėjų generavime apie tai, kaip vaizdinis turinys gali būti pristatomas nestandartiniais būdais.

Darbo tikslas ir uždaviniai

Darbo pagrindinis tikslas - sukurti ir ištirti interaktyvią komandinės eilutės sąsają, kuri leistų naudotojams naršyti gatvių vaizdus ASCII formatu, siekiant nustatyti tokios sistemos techninį įgyvendinamumą ir naudojimo patogumą be tradicinės grafinės aplinkos.

Uždaviniai:

1. Išanalizuoti esamas technologijas ir metodus, skirtus vaizdiniam turiniui konvertuoti į ASCII formatą, bei įvertinti jų tinkamumą interaktyviai gatvių vaizdų sistemai.
2. Ištirti „Street View“ programavimo sąsajos (API) galimybes ir apribojimus, siekiant efektyviai gauti ir apdoroti gatvių vaizdų duomenis komandinės eilutės aplinkoje.
3. Sukurti prototipą, demonstruojantį ASCII formatu pateikiamų gatvių vaizdų naršymą, įskaitant judėjimą erdvėje.
4. Parengti ir įgyvendinti intuityvią navigacijos sistemą, pritaikytą specifiniams komandinės eilutės aplinkos apribojimams ir galimybėms.
5. Įgyvendinti žaidybinę programos funkciją, kuri leistų naudotojui lengvai pamatyti esamą funkcionalumą.
6. Atlikti sukurtos sistemos testavimą, vertinant tiek techninį veikimą, tiek naudotojo patirties aspektus skirtingose naudojimo aplinkose.
7. Nustatyti ir dokumentuoti šio tipo sąsajos praktinio taikymo ribas, galimybes ir tobulinimo kryptis.
8. Įvertinti projekto rezultatus ir suformuluoti išvadas apie ASCII komandinės eilutės sąsajų potencialą interaktyvioms geografinėms sistemoms.

Darbo strukrūra

Sistemos apimtis

1 Analizė

1.1 Techninių galimybių analizė

Šiame skyriuje analizuojamos techninės kliūtys ir apribojimai, su kuriais susidurta kuriant komandinės eilutės sąsają „Street View“ tipo platformai, naudojant ASCII meną vaizdams atvaizduoti. Analizė apima tiek išorinius veiksnius (pavyzdžiui, priklausomybes nuo trečiųjų šalių paslaugų), tiek vidinius (pavyzdžiui, pasirinktos technologinės aplinkos apribojimus).

1.1.1 Pagrindinės techninės kliūtys ir sprendimai

1.1.1.1 Prieiga prie „Street View“ duomenų ir API kainodara

Pradinė idėja: Idealus variantas būtų buvęs naudoti plačiausiai paplitusią ir didžiausią aprėptį turinčią „Google Street View“ platformą.

Kliūtis: „Google Maps Platform“ programavimo sąsaja, įskaitant „Street View“ prieigą, neturi nemokamo plano, tinkamo projekto mastui, o mokami planai viršijo projekto finansines galimybes (arba buvo nepraktiški nekomerciniam/eksperimentiniam projektui). Tai tapo esminiu finansiniu ir techniniu barjeru realizuoti pradinę viziją naudojant „Google“ duomenis.

Sprendimas: Siekiant užtikrinti projekto įgyvendinamumą, buvo pasirinkta alternatyvi platforma – „Mapillary“. „Mapillary“ programavimo sąsaja siūlė nemokamą prieigos modelį.

Liekamasis apribojimas: Nors „Mapillary“ leido tęsti projektą, jos duomenų aprėptis tam tikrose geografinėse vietovėse gali būti mažesnė nei „Google Street View“, kas yra techninis apribojimas galutinio produkto naudojimo geografijai. Taip pat, dėl to kad ši sąsaja yra nemokama, ji nėra

ypatingai patikima, pavyzdžiui, ribojamos dėžės (angl. *bounding box*) užklausos dažnai yra atmetamos dėl per didelio bendro užklausų kiekio - tenka laukti, kol „Mapillary“ serveriai bus mažiau naudojami. Šis laukimas yra pagrindinis veiksnys, lemiantis galimą vartotojo sąsajos vėlavimą keičiant vaizdus.

1.1.1.2 Terminalo aplinkos grafiniai apribojimai

Kliūtis: Standartinė komandinės eilutės (terminalo) aplinka turi esminių grafinių galimybių apribojimų, lyginant su grafinėmis vartotojo sąsajomis (angl. *graphical user interface* arba *GUI*). Tai tiesiogiai paveikė galimybes atvaizduoti „Street View“ vaizdus ir kurti vartotojo sąsają:

Spalvų palaikymas: Daugelis standartinių terminalų emuliatorių ir populiarių terminalo vartotojo sąsajos (angl. *text user interface* arba *TUI*) bibliotekų dėl atgalinio suderinamumo arba paprastumo dažnai palaiko ribotą spalvų paletę (pvz., 16 spalvų) arba neleidžia pilnai išnaudoti modernesnių terminalų galimybių (pavyzdžiui, 256 spalvų palaikymo). Tai ženkliai apribotų galimybes tiksliai ir detaliai konvertuoti fotografinius vaizdus į ASCII meną, išlaikant vizualinį aiškumą, jei būtų pasikliauta tik standartiniais įrankiais.

Šrifto dydžio ir stiliaus variacijos: Terminalai natūraliai nepalaiko skirtingų šrifto dydžių ar stilių naudojimo viename ekrano lange, kas apsunkino intuityvios ir vizualiai struktūruotos vartotojo sąsajos elementų (pavyzdžiui, antraščių, mygtukų, informacinių blokų) kūrimą.

Sprendimas/Poveikis:

- Spalvoms: Siekiant įveikti standartinių bibliotekų apribojimus ir pagerinti ASCII meno kokybę, buvo sukurtas nuosavas TUI modulis/komponentas, specialiai pritaikytas išnaudoti platesnes modernių terminalų spalvų galimybes (ypač 256 spalvų režimą). Tai leido pasiekti detalesnį vaizdą, tačiau pareikalavo papildomų programavimo pastangų.
- Šriftams/UI Elementams: UI elementai, kuriems įprastai būtų naudojami skirtingi šrifto dydžiai (pavadinimai, meniu punktai), taip pat buvo realizuoti kaip ASCII menas, leidžiantis vizualiai juos atskirti ir struktūruoti sąsają, tačiau padidinant generuojamo vaizdo sudėtingumą.

1.1.1.3 Vaizdo reprezentacijos tikslumas

Kliūtis: Pats fotografinio vaizdo konvertavimas į ASCII meną yra techniškai ribotas procesas. Nepriklausomai nuo algoritmų, ASCII reprezentacija visada bus ženkliai žemesnės raiškos ir detalumo nei pradinis vaizdas. Tai yra fundamentalus techninis apribojimas, lemiantis, kad galutinis produktas gali perteikti tik apytikslį vaizdą, o ne tikslią fotografinę kopiją. Projekto įgyvendinamumas apsiriboja būtent tokio aproksimuoto vaizdo pateikimu.

Našumo aspektas: Dinaminis ASCII meno generavimas ir atvaizdavimas terminale, ypač naviguojant (t.y., dažnai keičiantis vaizdui), gali atrodyti lėtas. Tačiau pagrindinė vėlavimo priežastis dažniausiai yra ne pats ASCII meno generavimo procesas (kuris yra sąlyginai greitas modernioje technikoje), o laukimas, kol bus gautas atsakymas iš „Mapillary“ API. Senesniuose kompiuteriuose ar lėtesniuose terminaluose pats generavimas taip pat gali prisidėti prie nevisiškai sklandaus veikimo, kas yra techninis naudojimo patirties apribojimas.

Išvada: Nepaisant identifikuotų techninių kliūčių, susijusių su API prieiga ir jos patikimumu, terminalo aplinkos ribotumais ir vaizdo konversijos prigimtimi, projektas buvo techniškai įgyvendinamas pasirinkus alternatyvius sprendimus (pavyzdžiui, „Mapillary“ programavimo sąsaja, nuosavas TUI modulis skirtas geresniam spalvų išnaudojimui) ir pripažįstant neišvengiamus platformos apribojimus (ASCII meno detalumo lygį, priklausomybę nuo „Mapillary“ atsako laiko). Šie sprendimai leido sukurti veikiantį prototipą ar produktą, nors galutinis rezultatas ir skiriasi nuo hipotetinio idealaus varianto, kuris galėtų būti sukurtas neribojant finansų ar technologinių platformų galimybių.

2 Projektas

2.1 Realizacijai keliami reikalavimai

Šiame skyriuje apibrėžiami pagrindiniai nefunkciniai reikalavimai, keliami kuriamai sistemai, apimantys jos naudojimo patogumą, veikimo charakteristikas, aplinkos sąlygas ir kitus svarbius aspektus.

2.1.1 Reikalavimai panaudojamumui

- Intuityvi navigacija: sistema turi leisti vartotojui naršyti (judėti pirmyn/atgal arba į aplinkines lokacijas) naudojant aiškius ir lengvai įsimenamus klaviatūros klavišus, įprastus komandinės eilutės aplinkoje (pvz., rodyklių klavišai, WASD ar panašiai).
- Aiškus atsakas: sąsaja turi aiškiai informuoti vartotoją apie dabartinę būseną (pvz., vaizdo krovimas, klaida gaunant duomenis iš „Mapillary“ sąsajos).
- Mokymosi paprastumas: bazinis sistemos naudojimas (paleidimas, pagrindinė navigacija) turėtų būti lengvai perprantamas tikslinei auditorijai (komandinės eilutės naudotojams), pateikiant trumpą pagalbos informaciją paleidimo metu arba per specialią komandą (pvz., `--help`).
- Klaidų apdorojimas: sistema turi korektiškai apdoroti numatomas klaidas (pvz., „Mapillary“ nepasiekiamumas, neteisingos koordinatės, interneto ryšio nebuvimas) ir pateikti vartotojui suprantamą klaidos pranešimą, neužlūžtant pačiai programai.

2.1.2 Reikalavimai vykdymo charakteristikoms

- Atsako laikas (angl. *response time*): nors bendras atsako laikas priklauso nuo „Mapillary“, pati ASCII vaizdo generavimo ir atvaizdavimo terminale operacija turėtų būti pakankamai sparti, kad nesukeltų reikšmingo papildomo vėlavimo modernioje techninėje įrangoje po atsakymo gavimo.
- Resursų naudojimas (angl. *resource usage*): programa neturėtų nepagrįstai apkrauti sistemos resursų, veikdama kaip tipinė komandinės eilutės aplikacija.

2.1.3 Reikalavimai veikimo sąlygoms

- Terminalo suderinamumas (angl. *terminal compatibility*): sistema turi siekti veikti populiariuose terminalų emuliatoriuose, palaikančiuose bent 256 spalvas (pvz., „GNOME Terminal“, „Konsole“, „iTerm2“, „Windows Terminal“), pagrindinėse operacinėse sistemose („Linux“, „macOS“, „Windows“).
- Priklausomybė nuo tinklo (angl. *network dependency*): veikimui būtinas aktyvus interneto ryšys prieigai prie „Mapillary“ programavimo sąsajos.
- Programinės įrangos priklausomybės (angl. *software dependencies*): reikalingos priklausomybės (pvz., specifinė „JVM“ versija, „Docker“ ir panašiai) turi būti aiškiai dokumentuotos.

2.1.4 Reikalavimai sistemos išvaizdai

- Vizualinis aiškumas (angl. *visual clarity*): ASCII menas, nors ir riboto detalumo, turėtų būti generuojamas taip, kad pagrindiniai objektai ir erdvės kryptis būtų bent apytiksliai atpažįstami. Spalvų naudojimas (kai palaikoma) turėtų didinti aiškumą.
- Sąsajos konsistencija (angl. *interface consistency*): tekstiniai vartotojo sąsajos elementai (pranešimai, meniu, pagalba) turėtų naudoti nuoseklų formatavimą ir stilių visoje aplikacijoje.

2.1.5 Reikalavimai sistemos priežiūrai

- Kodo struktūra ir skaitymas (angl. *code structure and readability*): kodas turi būti logiškai struktūrizuotas (pvz., pagal modulius ar klases) ir parašytas laikantis bendrų programavimo gerosios praktikos principų (pvz., prasmingi pavadinimai, komentarai sudėtingesnėse vietose), kad būtų lengviau jį suprasti ir modifikuoti ateityje, kas ypač svarbu akademiniam darbui.

2.1.6 Reikalavimai saugumui

- Išorinės sąsajos raktų apsauga (angl. *API key protection*): jei naudojamas „Mapillary“ ar kitokios sąsajos raktas, jis neturėtų būti tiesiogiai įkoduotas viešai prieinamame kode. Rekomenduojama naudoti konfigūracijos failą ar aplinkos kintamąjį.
- Duomenų privatumas (angl. *data privacy*): sistema neturėtų rinkti, saugoti ar perduoti jokių vartotojo asmeninių duomenų, išskyrus tuos, kurie būtini išorinės sąsajos užklausoms (pvz., geografinės koordinatės).

2.1.7 Teisiniai reikalavimai

- Išorinės programavimo sąsajos naudojimo sąlygos (angl. *API Terms of Service*): sistemos naudojimas turi nepažeisti „Mapillary“ naudojimo sąlygų ir politikos.
- Bibliotekų licencijos (angl. *library licensing*): Naudojamos trečiųjų šalių bibliotekos turi turėti su projekto tikslais (pvz., akademinis, galimai atviras kodas) suderinamas licencijas, ir turi būti laikomasi tų licencijų reikalavimų.

2.2 Projektavimo metodai

2.2.1 Kodėl „Scala“?

2.2.1.1 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.

2.2.1.2 Istorinė programavimo kalbų raida

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

2.2.1.2.1 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 paties žemiausio lygio prieigą prie procesoriaus instrukcijų. Tai leidžia patyrusiam programuotojui pasiekti didesnę greitaveiką, tikslesnį veikimą bei mažesnį galutinio failo dydį. Visa tai yra labai naudinga specifinėse situacijose, kai resursai yra ypatingai riboti. Tačiau net tokiu atveju, dauguma profesionalų rinktūsi įrankį, suteikiantį šiek tiek daugiau abstrakcijos.

2.2.1.2.2 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.

2.2.1.2.3 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.

2.2.1.2.4 „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“.

2.2.1.2.5 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 alternativos sistemų programavimo standartui „C“. Turint tiek daug pasirinkimo laisvės, renkantis programavimo kalbą galima daugiau galvoti apie jos stilių bei abstrakcijos lygį.

2.2.1.3 Kalbos rinkimasis

2.2.1.3.1 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.

2.2.1.3.2 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 kompiliuojamą programavimo kalbą.

2.2.1.3.3 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 būs sąlyginai didelė, todėl mes pasirinkome nudoti kalbą su statiniais tipais.

2.2.1.3.4 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.

2.2.1.3.5 Programavimo kalba

Po šios nuoseklios analizės mes turime bendrą idėją, ko tikimės iš pasirinktos programavimo kalbos:

- sąlyginai aukšto abstrakcijos lygio;
- galimybės kodą kompiliuoti;
- griežtų statinių tipų;
- funkcinio programavimo stiliaus;

Yra daugybė pasirinkimų, atitinkančių šiuos kriterijus, kaip „Haskell“, „Clojure“, „Scala“, „F#“, „OCaml“ bei daugybė kitų. Visos šios kalbos yra plačiai naudojamos didelėse įmonėse ir yra puikiai tinkamos spręsti įvairiausioms problemoms, taip pat ir mūsų projektui:

- „Facebook“, socialinės medijos platforma, naudoja „Haskell“ programavimo kalbą siekiant kovoti su šlamštu savo platformoje (citata https://engineering.fb.com/2015/06/26/security/fighting-spam-with-haskell/)
- „Walmart“, JAV mažmeninės prekybos centras, naudoja „Clojure“ savo duomenų valdymo sistemai (citata https://clojure.org/community/success_stories)
- „X“ (anksčiau buvusi „Twitter“ socialinės medijos platforma), plačiai naudoja „Scala“.
- „Microsoft“, JAV programinės ir techninės įrangos gamintojas, sukūrė ir naudoja „F#“ įvairioms paslaugoms (citata https://learn.microsoft.com/en-us/dotnet/fsharp/).
- „Jane Street“, JAV patentuota prekybos įmonė, naudoja „OCaml“ prekybos sistemoms ir finansinei analizei (citata https://blog.janestreet.com/why-ocaml/).

Žinodami, jog beveik visas programavimo kalbas galima vienaip ar kitaip panaudoti, sprendžiant pačias įvairiausias problemas, galutiniame kalbos pasirinkime labiausiai vadovavomės esama pažintimi su kalba. Pasirinkę kalbą, kurios pagrindus jau žinome, galime sutelkti daugiau dėmesio pačiam programos veikimui. Šitaip mąstant, prieš akis iškyla pagrindinis favoritas - „Scala“.

2.2.2 Funkcinis programavimas su „Scala“

2.2.2.1 Apie „Scala“

„Scala“ programavimo kalba, taip pat kaip ir „Java“, yra skirta dirbti su „JVM“ (*Java Virtual Machine*) platforma - tai reiškia, jog programinis kodas yra kompiliuojamas ne tiesiai į dvejetainį kodą, o į specialų bitų kodą (angl. *bytecode*), kuris gali veikti bet kokioje operacinėje sistemoje. Taip pat tai reiškia, jog „Scala“ kode galima naudotis ne tik „Scala“ įskiepiais, viskuo, ką mums suteikia „Java“ programavimo kalba, net galime tiesiogiai kreiptis į „Java“ kodą. Tai yra ypač svarbus privalumas, žinant, jog „Scala“ yra sąlyginai nepopuliari kalba, kurioje gali trūkti norimų įskiepių.

Ši programavimo kalba išsiskiria nuo kitų funkcinių kalbų tuo, jog ji nėra vien tik funkcinė. Joje, priešingai nei kalboje kaip „Haskell“, galime kurti kintamas reikšmes, nors tai nėra rekomenduojama. Taip pat „Scala“ turi klases, bruožus (angl. *trait*) ir daugybę kitų kodo projektavimo modelių, kuriuos dažnai matome objektinio stiliaus kalbose - tai stipriai palengvina darbą žmogui, pirmą kartą bandančiam funkcinį programavimą.

„Scala“ buvo sukurta 2004 metais Martino Odersky (citata https://www.oreilly.com/library/view/scala-and-spark/9781785280849/005ee526-d74c-4d1e-a1b3-34f6213d5ece.xhtml), siekiant sukurti modernią programavimo kalbą, kuri apjungtų objektinio ir funkcinio programavimo privalumus. Vienas didžiausių „Scala“ privalumų yra jos išraiškingumas – dažnai galima parašyti daugiau funkcionalumo su mažiau kodo eilučių, palyginus su „Java“ ar kitomis tradicinėmis kalbomis. Štai pavyzdys, kaip paprasčiau gali atrodyti funkcinis kodas. Šių kodo tikslas - išfiltruoti iš skaičių sąrašo tik tuos skaičius, kurie dalinasi iš dviejų, ir gauti jų kvadratus.

„Java“ pavyzdys:

```
List<Integer> result = new ArrayList<>();
for (Integer number : numbers) {
    if (number % 2 == 0) {
        result.add(number * number);
    }
}
```

„Scala“ pavydzys:

```
val result = numbers.filter(_ % 2 == 0).map(x => x * x)
```

Toks programavimo stilius, bent mūsų nuomone, yra daug lengviau skaitomas žmogui. Dideliame projekte tai žymiai palengina svetimo žmogaus kodo skaitymą, o pati projekto apimtis būna žymiai mažesnė, lyginant su tradicinėmis programavimo kalbomis. Taip pat, kadangi nėra naudojamos jokios kintamos reikšmės, tokį kodą yra saugu naudoti lygiagrečioje aplinkoje, nereikia papildomai galvoti apie lenktynių sąlygas.

Statinis tipizavimas yra dar vienas svarbus „Scala“ bruožas. Nors programuotojui nebūtina nurodyti kintamųjų tipus (dėl automatinio tipo išvedimo mechanizmo), kompiliatorius vis tiek aptinka tipo nesuderinamumo klaidas kompiliavimo metu, o ne vykdymo metu. Tai padeda išvengti daugelio klaidų dar prieš paleidžiant programą.

„Scala“ turi turtingą sintaksę, kuri leidžia kurti aiškias, glaustas ir elegantiškas duomenų struktūras bei algoritmus. Šabloninis atitikimas (angl. *pattern matching*), aukštesnės eilės funkcijos, tingi (angl *lazy*) inicializacija ir nemutuojamų duomenų struktūrų palaikymas – tai tik keletas funkcinio programavimo bruožų, kuriuos egzistuoja „Scala“ kalboje.

Pramonėje „Scala“ dažnai naudojama didelės apimties duomenų apdorojimo sistemose. „Apache Spark“ (citata https://www.chaosgenius.io/blog/apache-spark-with-scala/), vienas populiariausių didelių duomenų apdorojimo karkasų, yra parašytas būtent „Scala“ kalba. Tokios įmonės kaip „X“, „LinkedIn“ ir „Netflix“ (citata https://sysgears.com/articles/how-tech-giants-use-scala/) naudoja „Scala“ savo pagrindinėse sistemose dėl jos gebėjimo efektyviai valdyti lygiagrečias užduotis ir didelius duomenų srautus.

„Scala“ ekosistema taip pat siūlo keletą galingų įrankių ir įskiepių, tokių kaip „Akka“ (citata https://akka.io/) (aktorių modeliu pagrįsta lygiagretumo sistema), „Play Framework“ (citata https://www.playframework.com/) (tinklalapių kūrimo karkasas) ir „Cats“ (citata https://typelevel.org/cats/) (funkcinio programavimo abstrakcijos). Šios bibliotekos padeda programuotojams kurti tvarų, testuojamą ir lengvai prižiūrimą kodą.

Nors „Scala“ mokymosi kreivė gali būti šiek tiek statesnė nei kai kurių kitų programavimo kalbų, jos teikiami privalumai – ypač kuriant sudėtingas, didelio masto sistemas – dažnai atperka pradinį mokymosi laiką. Tai yra puikus pasirinkimas programuotojams, norintiems išplėsti savo įgūdžius ir įsisavinti funkcinio programavimo koncepcijas, išlaikant pažįstamą objektinio programavimo aplinką.

2.2.2.2 „Cats-Effect“ karkasas

Kaip minėjome anksčiau, „Scala“ nėra idealiai funkcinė kalba. Vienas pagrindinis funkcionalumas, kurio nėra šioje programavimo kalboje, kuris dažnai randamas kitose funkcinėse programavimo kalbose - efektų valdymas.

Prieš aiškinantis kaip reikia valdyti šalutinius efektus, reikia suprasti, kas tiksliai yra funkcinis programavimas. Funkcinis programavimas yra pagrįstas matematinėmis funkcijomis, taigi jomis ir galime pasinaudoti apibūdinant funkcinio programavimo paradigmą. Štai pažiūrėkime į šią funkciją:

$f(x) = 3x$

Tokia funkcija yra tarytum sujungimas tarp dviejų skaičių sarašų. Pavyzdžiui, sąrašas (1, 2, 3) patampa sąrašu (3, 6, 9). Kieviena įvestis turi vieną ir tik vieną išvestį. Nesvarbu kokia yra išvestis, jai visada bus išvestis (nėra jokių išimčių). Funkcijos rezultatas yra tiesiogiai išvedamas iš įvesties ir iš nieko daugiau (neskaitant žinoma kitokių konstantų, kaip 3). Funkcija tik apskaičiuoja išvestį ir nieko daugiau - ji nekeičia kažkokių kitų reikšmių, nesiunčia laiško, neperka obuolių - ji tik įvestį paverčia išvestimi. Tai ir yra visa esmė funkcinio programavimo.

Svarbi tokių grynų funkcijų savybė yra referencinis skaidrumas (angl. *referential transparency*). Tai reiškia, kad bet kurį funkcijos iškvietimą su konkrečiomis įvesties reikšmėmis galima mintyse (ar net kodo pertvarkymo metu) pakeisti jos rezultatu, nepakeičiant programos elgsenos visumos. Pavyzdžiui, jei žinome, kad mūsų funkcija f(2) visada grąžina 6, mes galime visur programoje, kur matome f(2), įsivaizduoti tiesiog reikšmę 6. Tai daro kodą daug lengviau suprantamą, testuojamą ir nuspėjamą, nes funkcijos rezultatas nepriklauso nuo jokių paslėptų faktorių ar ankstesnių įvykių – tik nuo jos argumentų.

Panagrinėkime kelis pavyzdžius.

```
def doSomething(value: Int) = value * 3
```

Štai čia matome funkciniame programavime vadinamą gryną (angl. *pure*) funkciją - ji įvestį paverčiame išvestimi ir nieko daugiau. Ji yra referenciškai skaidri. Pasižiūrėkime, kokie pavyzdžiai nebūtų grynos funkcijos ir kaip galėtume jas paversti grynomis funkcijomis.

```
def doSomething(value: Int) = 5 / value
```

Ši funkcija dalina iš įvesties - tai reiškia, jog ne kiekvienai reikšmei yra išvestis. T.y. reikšmei 0 išvesties nėra - programoje įvyks dalybos iš nulio klaida. Tai galima išspręsti pridėję papildomą sąlygą, kuri patikrintų įvestį:

```
def doSomething(value: Int) =
    if (value == 0) 0
    else 5 / value
```

Dabar ši funkcija yra gryna. Galima ir kitaip sugadinti funkcijos grynumą:

```
def doSomething(value: Int) = {
    x++ // Šalutinis kintamos reikšmės padidinimas
    println("Šalutinis spausdinimas") // Šalutinis spausdinimas
    value * 3
}
```

Ši funkcija nebėra gryna, nes ji daro daugiau, nei reikia norint gauti išvestį. Ji pažeidžia referencinį skaidrumą, nes jos iškvietimas ne tik grąžina reikšmę, bet ir turi šalutinį poveikį (pakeičia x reikšmę, išspausdina tekstą), todėl negalime jos tiesiog pakeisti rezultatu, neprarasdami šių poveikių. Kitaip tariant, turėtų būti aišku ką daro funkcija vien iš jos įvesties ir išvesties tipų, net neskaitant pačios

funkcijos implementacijos. Tokie šalutiniai efektai žymiai apsunkina programos klaidų ieškojimą ir kodo supratimą.

Tačiau kai kurios funkcijos negali būti idealiai grynos. Pavyzdžiui, spausdinimas į ekraną ar HTTP užklausa - abi šios funkcijos priklauso nuo išorinės aplinkos. Jei programa neturi kur spausdinti, ji neveiks. Jei serveris į kurį siunčiame užklausą neegzistuoja ar neveikia, mūsų programa taip pat neveiks. Šiai problemai spręsti funkcinėse programavimo kalbose paprastai yra kažkokia forma efektų valdymo.

Efektų valdymas funkciniame programavime yra būdas tvarkyti šalutinius efektus (angl. *side effects*) – procesus, kurie keičia programos būseną už funkcijos aprėpties ribų, pavyzdžiui, duomenų nuskaitymas ar įrašymas, tinklo operacijos, atsitiktinių skaičių generavimas ir panašios operacijos. Tradicinėse funkcinėse kalbose šalutiniai efektai yra aiškiai apibrėžiami ir izoliuojami, kas leidžia programuotojams tiksliai žinoti, kokius poveikius gali turėti jų funkcijos. Tai suteikia geresnes galimybes testuoti kodą, lengviau suprasti programos veikimą, išvengti netikėtų šalutinių pasekmių bei nesunkiai valdyti programos klaidas. „Cats-Effect“ (citata https://typelevel.org/cats-effect) karkasas „Scala“ programavimo kalbai įveda šią koncepcija per IO monadą ir kitus abstrakcijos mechanizmus, kurie leidžia programuotojams apibrėžti ir komponuoti efektus deklaratyviu būdu, kartu išlaikant griežtą tipų saugumą.

Toliau panagrinėsime koks tikslas yra naudoti efektų valdymo karkasą kaip „Cats-Effect“ bei kokias problemas jis padeda išspręsti.

Esminė šio karkaso abstrakcija yra pluoštai (angl. *Fibers*) (citata https://typelevel.org/cats-effect/docs/concepts#fibers). Tai yra „Cats-Effect“ paralelizmo pagrindas. Pluoštai yra lengvos gijos, skirtos reprezentuoti seką veiksmų, kurie programos veikimo metu galiausiai bus realizuoti. Pluoštai yra ypatingai lengvi - vienas pluoštas užima vos 150 baitų atminties. Tai reiškia, jog mes galima sukurti dešimtis milijonų pluoštų be jokių problemų. Per daug nelendant į technines detales, galima jų naudą apibendrinti taip - pluoštai leidžia mums lengvai, be papildomo vargo, valdyti paralelizmą bei suteikia mums galimybę bet kurį skaičiavimo procesą sustabdyti ar atšaukti, net jei jis jau yra vykdomas.

Šio karkaso konteksta efektas (angl. *effect*) (citata https://typelevel.org/cats-effect/docs/concepts#effects) yra veiksmo (ar veiksmų) apibrėžimas, kuris bus įvykdytas, kai vyks kodo vertinimas (angl. *evaluation*). Pagrindinis toks efektas yra IO.

```
val spausdintuvas: IO[Unit] = IO.println("Labas, pasauli!")
```

Šiame kodo fragmente reikšmė *spausdintuvas* yra aprašymas veiksmo, kuris atspausdina tekstą į komandinę eilutę. Nesvarbu, kiek kartų mes iškviesime šią reikšmę, spausdinimas nebus įvykdytas nė karto, pavyzdžiui:

```
printer
printer
printer
```

Šis kodas neišspausdins teksto nė karto, nes mes dar nenurodėme, jog efektą reikia įvykdyti. Jei nurodytume, jog efektas turi būti įvykdytas, tekstas būtų išspausdintas kiekvieną kartą. Tai mums leidžia dirbti su bet kokiomis reikšmėmis, net tokiomis kaip *Unit* (kitose kalbose dažniau naudojamas terminas yra *void*) taip pat, kaip dirbtume su paprastomis reikšmėmis, kaip *Int*, *String* ar kitomis - jas galime naudoti, perpanaudoti, grąžinti naują reikšmę ir panašiai. Tai yra galima todėl, nes mes programiniame kode dirbame ne su pačia šalutine reikšme, o su jos apibūdinimu.

Dažnas IO monados apibūdinimas skamba taip: IO aprašo transformaciją iš vienos pasaulio būsenos į kitą. Kiekvienas veiksmas IO viduje yra ne pats veiksmas, o receptas naujai pasaulio būsenai, kuri

gautųsi įvykdžius tą veiksmą. Kaip matome, šitoks apibūdinimas nepažeidžia funkcinio programavimo taisyklių - nebuvo jokių kintamų reikšmių ar tiesioginių šalutinių efektų pačiame aprašyme, tik dvi atskiros, nekintamos koncepcijos - pasaulis prieš ir po veiksmo aprašymo.

Tuo tarpu „Scala“ paralelizmo monada „Future“ to negali.

```
val spausdintuvas: Future[Unit] = Future(println("Labas, pasauli!"))
```

Kad ir kiek kviestume šia reikšmę, ji išspausdins rezultatą vieną ir tiek vieną kartą, vykdydama efektą iš karto ją sukūrus. Tai nėra intuityvu, neleidžia mums perpanaudoti reikšmės ateityje ir pažeidžia referencinį skaidrumą (angl. *referential transparency*) – pagrindinį funkcinio programavimo principą, kurio IO laikosi dėl savo tingumo (angl. laziness).

Anksčiau minėjome klaidų valdymą. „Cats-Effect“ karkasas mums taip pat suteikia paprastas ir intuityvias sąsajas valdyti klaidoms, įvykusioms IO monados veiksmų metu. Mes galime saugiai dirbti su galimai klaidą sukeliančiais efektais naudodami metodus kaip *attempt* (kuris paverčia rezultatą kurio galime negauti dėl klaidos į *Either* tipą, kuris saugo arba rezultatą, arba įvykusią klaidą) arba *handleErrorWith* (kuris leidžia aprašyti, kaip elgtis klaidos atveju).

```
val galimaiKlaidingas: IO[Int] = IO(5 / 0) // Efektas, kuris mes klaidą

val apdorotaKlaida: IO[Int] = galimaiKlaidingas.handleErrorWith { klaida =>
  // Jei įvyko klaida, atspausdiname pranešimą ir grąžiname numatytąją reikšmę
  IO.println(s"Įvyko klaida: ${klaida.getMessage}") *> IO.pure(-1)
}
```

Dar vienas ypatingai patogus dalykas, kurį suteikia šis karkasas, yra resursų valdymas. Daugelis šalutinių efektų apima darbą su resursais, kuriuos reikia ne tik atidaryti ar įsigyti, bet ir saugiai uždaryti ar paleisti, nepriklausomai nuo to, ar operacijos su jais pavyko, ar įvyko klaida (pavyzdžiui, failų skaitytuvai, duomenų bazių prisijungimai, tinklo lizdai). Rankiniu būdu tai užtikrinti sudėtinga ir linkę į klaidas (resursų nutekėjimą). „Cats-Effect“ siūlo elegantišką sprendimą – *Resource* duomenų tipą. Jis aprašo, kaip įsigyti (angl. *acquire*) resursą ir kaip jį paleisti (angl. *release*).

```
import cats.effect._
import java.io._

// Aprašome, kaip saugiai gauti ir uždaryti failo skaitytuvą
def failoSkaitytuvas(kelias: String): Resource[IO, BufferedReader] =
  Resource.make {
    IO(new BufferedReader(new FileReader(kelias))) // Kaip įsigyti
  } { skaitytuvas =>
    IO(skaitytuvas.close()).handleErrorWith(_ => IO.unit) // Kaip paleisti
(užtikrintai)
  }

// Naudojame resursą saugiai: .use garantuoja, kad release bus iškviestas
val saugusSkaitymas: IO[String] = failoSkaitytuvas("manoFailas.txt").use
{ skaitytuvas =>
  IO(skaitytuvas.readLine()) // Darbas su resursu
}
```

Tai užtikrina, jog resursai bus paleisti net jei programoje įvyks klaida, ar ji bus nutraukta rankiniu būdu.

Visos šitos abstrakcijos leidžia mums rašyti lengviau suprantamą, pertvarkomą ir patikimesnį programinį kodą. Žinant, jog šio projekto dydis bus sąlyginai didelis, o jame daug pašalinių efektų dirbant su komandinės eilutės spausdinimu, konfigūracinių failų nuskaitymu, išorinių sąsajų

bendravimu bei daugybe baitų ir kitokių tipų transformacijų, šios abstrakcijos mums labai padėjo parašyti patikimai veikiančią programą.

2.2.3 Projektavimo valdymas ir eiga

Atsižvelgiant į projekto tiriamąjį ir eksperimentinį pobūdį bei pradinį neapibrėžtumą dėl galutinio sprendimo techninio įgyvendinamumo, nebuvo taikomas griežtas, iš anksto suplanuotas programinės įrangos kūrimo modelis, pavyzdžiui, krioklio (angl. *waterfall*). Vietoj to, buvo pasirinktas lankstus, iteracinis ir inkrementinis (angl. *iterative and incremental*) kūrimo procesas, turintis prototipavimo (angl. *prototyping*) ir eksperimentinio kūrimo (angl. *exploratory development*) bruožų.

Projekto eiga buvo valdoma dinamiškai, reaguojant į kylančius iššūkius ir atradimus:

1. Pradinis tyrimas ir analizė: pirmiausia buvo atlikta esamų technologijų analizė (ASCII generavimo metodai, „Street View“ tipo sąsajų galimybės ir apribojimai), siekiant įvertinti bendrą idėjos įgyvendinamumą.
2. Komponentų identifikavimas: pagrindinės sistemos dalys (pvz., prieiga prie „Mapillary“, vaizdo konvertavimas į ASCII, terminalo vartotojo sąsajos (TUI) modulis, navigacijos logika) buvo identifikuotos kaip atskiri funkciniai blokai.
3. Iteracinis kūrimas ir integravimas:
   - Buvo kuriamos ir testuojamos nedidelės, atskiros funkcionalumo dalys (pvz., pirminis užklausų siuntimas, bazinis ASCII generavimas).
   - Veikiantys komponentai buvo palaipsniui integruojami tarpusavyje.
   - Kiekvienos iteracijos pabaigoje buvo vertinamas rezultatas, sprendžiami iškilę techniniai sunkumai (pvz., „Mapillary“ patikimumo problemos, terminalo spalvų palaikymo iššūkiai).
4. Adaptacija ir krypties koregavimas: remiantis iteracijų rezultatais ir techninių galimybių analize, buvo priimami sprendimai dėl tolimesnės eigos. Pavyzdžiui, paaiškėjus standartinių TUI bibliotekų apribojimams, buvo nuspręsta kurti nuosavą TUI komponentą. Susidūrus su „Google Street View“ „API“ kainodaros kliūtimis, buvo pereita prie „Mapillary“.
5. Funkcionalumo plėtra: įsitikinus pagrindinių dalių veikimu, buvo pridedamos papildomos funkcijos (pvz., navigacijos patobulinimai, žaidybinis elementas).

Šis lankstus požiūris leido nuolat tikrinti technines hipotezes ir prisitaikyti prie realių apribojimų, kas buvo būtina tokio pobūdžio eksperimentiniam projektui. Darbai nebuvo skirstomi pagal griežtą grafiką, o prioritetai buvo nustatomi pagal einamųjų iteracijų poreikius ir techninę būtinybę.

2.2.4 Projektavimo technologija

Dėl anksčiau minėto iteracinio ir eksperimentinio projekto pobūdžio, nebuvo naudojamos specifinės formalios projektavimo technologijos ar griežti grafiniai žymėjimo standartai (notacijos), tokie kaip UML (angl. *Unified Modeling Language*) diagramos, visai sistemai aprašyti iš anksto. Sistemos projektas ir architektūra formavosi palaipsniui, vykstant kūrimo procesui.

Pagrindiniai projektavimo sprendimai ir sistemos struktūra buvo įtvirtinti ir dokumentuoti šiais būdais:

- Kodas kaip dokumentacija: pati programos kodo struktūra (moduliai, klasės, funkcijos), parinkti pavadinimai ir vidiniai komentarai tarnavo kaip pagrindinis techninio projekto artefaktas. Buvo stengiamasi laikytis bendrų programavimo gerosios praktikos principų, kad kodas būtų kuo aiškesnis ir lengviau suprantamas.
- Tekstinė dokumentacija: esminiai projektavimo sprendimai, ypač susiję su techniniais apribojimais ir pasirinktomis alternatyvomis (pvz., „Street View“ sąsajos pasirinkimas, TUI realizacija), yra aprašyti šiame baigiamajame darbe.

- Prototipavimas: funkcionalumo dalys buvo greitai prototipuojamos ir testuojamos tiesiogiai terminalo aplinkoje, kas leido empiriškai patikrinti projektavimo idėjas.

Nors formalūs projektavimo įrankiai nebuvo naudojami, kūrimo procese pasitelkti šie standartiniai programinės įrangos kūrimo įrankiai:
- Programavimo kalba: „Scala“ (su „Java“ virtualia mašina).
- Kūrimo aplinka (angl. *Integrated development environment* arba *IDE*): „IntelliJ IDEA“ su „Scala“ įskiepiu.
- Versijų kontrolės sistema: „Git“, kodui saugoti ir versijuoti, tikėtina, naudojant platformą „GitHub“.
- Bibliotekų valdymas ir projekto kompiliavimas: „sbt“ („Scala Build Tool“) – standartinis įrankis „Scala“ projektams.
- Tikslinė aplinka: įvairūs terminalų emuliatoriai („Linux“, „macOS“, „Windows“ sistemose), palaikantys 256 spalvas.

Apibendrinant, projektavimo technologija šiame darbe buvo labiau orientuota į praktinį įgyvendinimą ir laipsnišką sprendimo formavimą, o ne į išankstinį formalų modeliavimą.

3 Nuotraukų konvertavimas į ASCII

3.1 ASCII

Ascii (angl. *American Standard Code for Information interchange*) yra vienas iš populiariausių teksto simbolių kodavimo formatų, naudojamas atvaizduoti tekstą kompiuterinėse sistemose ir internete (https://www.techtarget.com/whatis/definition/ASCII-American-Standard-Code-for-Information-Interchange). Šis kodavimo standartas buvo sukurtas 1963 metais siekiant, jog skritingų gamintojų kompiuterių sistemos galėtų dalintis ir apdoroti informaciją. ASCII simboliai skirstomi į dvi grupes: spausdinamuosius ir nespausdinamuosius. Spausdinamieji simboliai apima raides, skaičius, skirybos ženklus bei specialius simbolius, tuo tarpu nespausdinamųjų aibė yra sudaryta iš eilučių pabaigos ženklų, tabuliacijos simbolių ir t.t. Šiame bakalauriniame darbe daugiausiai dėmesio skirsime spausdinamiesiems simboliams, kadangi tik iš jų gali būti atvaizduojami įvairūs vaizdai. ASCII standartas pasižymi paprastu ir kompaktišku simbolių kodavimu, kadangi vienam simboliui reprezentuoti užtenka vos 7 arba 8 bitų, priklausomai ar naudojama išplėstinių ASCII simbolių aibė. Šis paprastumas ir yra vienas iš didžiausių šio formato minusų, nes palaikomi yra tik 255 unikalūs simboliai. Tai lėmė, jog 2003 metais standartų organizacija IETF (angl. *Internet Engineering Task Force*) įvedė naująjį „Unicode“ simbolių kodavimo standartą. Šis standartas pakeitė ASCII, tačiau naujasis formatas pilnai palaiko ASCII atgalinio suderinamumo pagalba. Nors šiomis dienomis naudojame „Unicode“ standartą, 255 simbolių rinkinys, anksčiau priklausęs ASCII formatui, vis dar vadinamas ASCII.

3.2 ASCII menas

ASCII menas tai grafinio dizaino technika, kuria vaizdai atvaizduojami pasitelkiant teksto simbolius. Šios meno formos pirmieji egzemplioriai užfiksuoti dar prieš ASCII standarto sukūrimą (Figure 2).

Figure 2: Spausdinimo mašinėles menas, kūrėjas Julius Nelson 1939m.

Vaizdų iš simbolių kūrimo pradžia siejama net ne su kompiuteriais, o su XIX amžiuje plačiai naudojamomis rašymo mašinėlėmis. Vaizdų sudarymas iš simbolių buvo skatinamas rašymo mašinėlių gamintojų rengiamuose turnyruose (https://direct.mit.edu/books/oa-monograph/5649/From-ASCII-Art-to-Comic-SansTypography-and-Popular). Antrasis ASCII meno populiarumo šuolis buvo matomas XX amžiaus viduryje, kai vis daugiau žmonių turėjo prieigą prie pirmųjų kompiuterių. Žinoma, tais laikais kompiuteriai dar neturėjo grafinių sąsajų, todėl vaizdus reprezentuoti buvo galima tik ASCII simboliais. Spausdinti ir masiškai platinti teksto simbolių meną kompiuterio pagalba buvo žymiai paprasčiau, nei naudojantis spausdinimo mašinėle. Tačiau sparčiai populiarėjant grafinėms vartotojo sąsajoms, ASCII menas buvo pakeistas rastrinės grafikos. Šiomis dienomis ASCII menas naudojimas nišiniuose sistemose ir programose dėl savo stilistinių priežasčių ir nostalgijos.

4 Pasiruosimas konvertuoti nuotraukas Į ASCII

4.1 Nuotraukos proporcijų išlaikymas

Siekiant konvertuoti nuotraukos pikselius į ASCII simbolius, susiduriame su proporcijų išlaikymo problema. Kitaip nei rastrinėje grafikoje, kurioje nuotraukos atvaizduojamos vienodo pločio ir aukščio pikseliais, teksto simboliai yra nevienodų dimensijų. Todėl tiesiogiai konvertuojant nuotrauką gausime vaizdą ištemptą vertikaliai. Pavyzdžiui, šrifto stiliaus „Courrier New“ simbolių dimensijos turi santikį 1:0,6, tai yra plotis sudaro 60% aukščio. Žinoma, teigti apie šį santikį galime tik dėl to, nes visi šio, konsolėms pritaikyto šrifto stiliaus simbolių plotis yra vienodas. Dėl paprastumo ir minimaliaus poveikio galutiniam rezultatui buvo laikoma, jog šis santykis yra 1:0,5, kitaip tariant aukštis yra du kartus didesnis už plotį. Siekiant išspręsti šią problemą būtina du kartus sumažinti vertikalią orginalios nuotraukos rezoliuciją, galimi keli sprendimo būdai:

- Vertikalios rezoliucijos sumažinimas pašalinant kas antrą nuotraukos pikselių eilutę. Šis metodas yra pats greičiausias, nereikalaujantis daug kompiuterio resursų. Išlaikomi aiškūs kraštai, tačiau šios kraštinės ne visais atvejais susijungs kaip orginaliame vaizde dėl apdorojimo metų prarandamos informacijos.

- Vertikalios rezoliucijos sumažinimas apskaičiuojant vidurkį tarp gretimų pikselių. Šiuo atveju gretimų pikselių reikšmių vidurkiai yra naudojami sukurti naują pikselio reikšmę neprarandant informacijos. Tačiau pagrindinis šio metodo minusas yra neryškus kraštų atvaizdavimas, kadangi dažnu atveju kelių visiškai skirtingų pikselių reikšmės yra sumaišomos į vieną.

4.2 ASCII simbolių dydžio pasirinkimas

Modernūs fotoaparatai geba sukurti labai aukštos rezoliucijos nuotraukas. Šie vaizdai yra sudaryti iš kelių milijonų pikselių. Konvertuojant kiekvieną nuotraukos pikselį į atskirą ASCII simbolį, gautas rezultatas nesutips į jokį komerciškai prieinamą ekraną. Šios problemos sprendimas yra elementarus - sumažinti šrifto dydį. Šis sprendimas turi daug teigiamų savybių, pavyzdžiui, sumažinus šriftą iki pačio mažiausio leidžiamo dydžio, rezultatas dažnu atveju kokybe neatsiliks nuo orginalaus rastrinio vaizdo. Taip pat, kuo mažesnis yra gaunamas paveiksliukas, tuo lengviau žmogaus smegenys geba atpažinti jo turinį. Mažesnį plotą užimantys objektai dažniausiai suvokiami per jų formą arba figūrą, o didesni objetai suprantami kaip fonas (https://link.springer.com/article/10.3758/BF03207416?utm_source=chatgpt.com). Dėl to suprasti abstraktų paveikslą žiūrint iš toli yra lengviau, tas pats gali būti pritaikyta ir ASCII menui. Žinoma, mažesnis šriftas ne visada yra geriau. Iš teksto simbolių kuriamo vaizdo esmė nėra pati aukščiausia kokybė. ASCII menas yra kuriamas dėl stilistinių tikslų. Taigi sumažinti šrifto dydį galima tik tiek, kol vis dar bus galima įskaityti individualius simbolius. Norint pasiekti optimalų rezultatą būtina suderinti abu anksčiau aptartus reikalavimus.

4.3 Nuotraukos reprezentacija pilkos spalvos tonais

ASCII meną galima skirstyti į 2 grupes: spalvotąjį ir nespalvotąjį. Kadangi visi kadrai gaunami iš gatvės lygio platformų „Google Maps“ ir „Mapillary“ jau bus spalvoti, pasirūpinti reikės tik konvertavimu iš RGB į pilkus atspalvius. Kovertuoti turėsime kiekvieną nuotraukos pikselį, tai atlikti galima pasitelkus viena iš trijų galimų formulių:

- Svertinis vidurkis – remiasi žmogaus akies jautrumu skirtingoms spalvoms. Kadangi žalia spalva žmogaus akiai atrodo šviesiausia, jos koeficientas yra didžiausias. Toliau mažėjimo tvarka seka raudona ir galiausiai mėlyna spalvos.

$$Y=0.299\times R+0.587\times G+0.114\times B$$

- Vidurkis – ši formulė yra pati paprasčiausia. Visos spalvos turi vienodą svorį skaičiuojant pilkos spalvos reikšmę.

$$Y=(R+G+B)/3$$

- Reliatyvus šviesumas - naujesnė svertinio vidurkio formulės atmaina. Kaip ir ankstesnėje formulėje, koeficientai apskaičiuoti remiantis akies jautrumu šviesai. Tačiau šįkart atsižvelgiama į modernių vaizduoklių ir ekranų technologijas bei naujus tyrimus apie akies šviesos suvokimą.

$$Y=0.2126\times R+0.7152G+0.0722B$$

Čia R – raudonos RGB spalvos reikšmė, G - žalios spalvos reikšmė, o B - mėlynos.