VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Reaktyvus programavimas įvykių kaupimo sistemose

Reactive Programming in Eventsourcing Systems

Magistro baigiamasis darbas

Atliko:	Žilvinas Kučinskas	(parašas)
Darbo vadovas:	Viačeslav Pozdniakov	(parašas)
Recenzentas:	prof. Rimantas Vaicekauskas	(parašas)

Vilnius – 2017

Santrauka

Įvykių kaupimo principas yra dabartinės būsenos saugojimas kaip eilė įvykių nutikusių sistemoje. Įvykiai saugo visą informaciją, reikalingą dabartinės būsenos atkūrimui. Šis būdas leidžia pasiekti aukštą operacijų kiekį bei įgalina efektyvias replikacijas. Tuo tarpu reaktyvus programavimas leidžia realizuoti reaktyviąsias programas komponavimo būdu, tai yra deklaratyviu stiliumi. Tokiu stiliumi kuriamos programos leidžia atskirti pagrindinę logiką į smulkesnes, lengviau suprantamas dalis. Darbe siekiama sukurti reaktyviojo programavimo programos sąsają, kuri leistų šiuos principus sujungti. Pritaikius reaktyvų programavimą įvykių kaupimo principu paremtose sistemose galima modeliuoti ne tik momentinius įvykius, tačiau turėti ir jų istoriją. Paslėpus įvykių žurnalą arba duomenų saugyklą, galima orientuotis į pačią sprendžiamos srities problemą, nekreipiant dėmesio į žemesnio lygio realizacijos detales. Darbe naudojama "Ruby" programavimo kalbos sintaksė, parodomas aukšto lygmens architektūros modelis, apjungiantis reaktyvaus programavimo, įvykių kaupimo bei komandų-užklausų atsakomybių atskyrimo principus bei aprašomos įvykių kaupimo sistemų kūrimo gairės. Panaudojant reaktyvius operatorius yra leidžiama kurti skaitymo modelį paslepiant visas operacijų su duomenų baze realizacijos detales.

Raktiniai žodžiai: reaktyvus programavimas, įvykių kaupimas, komandų-užklausų atsakomybių atskyrimas, "Reaktyvumo manifestas"

Summary

Eventsourcing describes current state as series of events that occurred in a system. Events hold all information that is needed to recreate current state. This method allows to achieve high volume of transactions, and enables efficient replication. Whereas reactive programming lets implement reactive systems in declarative style, decomposing logic into smaller, easier to understand components. Thesis aims to create reactive programming program interface, incorporating both principles. Applying reactive programming in eventsourcing systems enables modelling not only instantaneous events, but also have their history. Furthermore, it enables focus on the solvable problem, regardless of low level realisation details. In the following thesis, Ruby programming language is being used, high-level architecture, incorporating reactive programming, event sourcing and command-query responsibility segregation principles, is being shown and eventsourcing system creation guidelines are being described. Reactive operators enable read model creation without exposing realisation details of operations with data storage.

Keywords: reactive programming, event sourcing, command-query responsibility segregation, Reactive Manifesto

TURINYS

1. Įvadas

1.1. Tyrimo objektas

Tyrimo objektas yra reaktyvaus programavimo bei įvykių kaupimo principai. Reaktyvus programavimas yra susijęs su reaktyviomis sistemomis, o kalbant apie įvykių kaupimo sistemas, įvykių kaupimo principas yra dažniausiai neatsiejamas nuo komandų-užklausų atsakomybių atskyrimo principo. Dėl to reikia apžvelgti ir pačią sistemą, įkomponuojančią šiuos principus, jog būtų galima susidaryti sistemos architektūros (aukštesnės abstrakcijos) vaizdą, analizuoti panaudojimo atvejus bei aprašyti tokių sistemų kūrimo gaires.

1.2. Darbo tikslas

Darbo tikslas yra pritaikyti reaktyvaus programavimo principus įvykių kaupimo sistemose taip, jog skaitymo modelis būtų kuriamas tik komponavimo būdu, neturėtų būsenos, tai yra visos operacijos su duomenų baze būtų paslėptos, o programinis kodas - griežtai tipizuotas.

1.3. Darbo uždaviniai

Siekiant tikslo, turi būti išspręsti šie uždaviniai:

- išnagrinėti egzistuojančią (-ias) reaktyvaus programavimo biblioteką (-as).
- išnagrinėti egzistuojančią (-ias) įvykių kaupimo biblioteką (-as) ar programavimo karkasą (-us).
- sukurti konceptualų architektūros modelį, apjungiantį reaktyvaus programavimo bei įvykių kaupimo principus, aprašyti praktines įvykių kaupimo sistemos kūrimo gaires.
- sukurti arba praplėsti esamą konkretizuotą kalbą (angl. domain specific language), apjungiančią reaktyvaus programavimo bei įvykių kaupimo principus.
- aprašyti konkretizuotuos kalbos kūrimo metodiką, apibrėžti gautų rezultatų apribojimus, suformuluoti iškilusias problemas bei paaiškinti jų priežastis.

1.4. Darbo nauda įgyvendinus tikslą

Akademinė:

- sukurtas inovatyvus arba alternatyvus būdas kurti skaitymo modelį įvykių kaupimo sistemose deklaratyviai, paslėpiant veiksmus su duomenų saugykla.
- medžiagą būtų galima naudoti dėstant apie įvykių sistemas, turėti tokių reaktyvių sistemų kūrimo gaires, pritaikomas praktinių užsiėmimų metu.

Verslo:

- Sukurtas pavyzdinis projektas, paruoštas naudoti gamybos aplinkoje (angl. production environment) - sutaupytų daug laiko, nes nereikėtų konfigūruoti aplinkos nuo nulio.

Kadangi reaktyvus programavimas leidžia kurti sistemas komponavimo būdu, deklaratyviai, pasiekus tikslą būtų galima supaprastinti įvykių kaupimu pagrįstų sistemų kūrimą, struktūrizuoti kodą į smulkesnius, lengviau suprantamus ir palaikomus komponentus.

1.5. Tyrimo aktualumas

Reaktyvusis programavimas (RP) yra programavimo kalbos paradigma, konkrečiai taikoma reaktyviosioms programoms kurti. Per pastaruosius keletą metų vis daugiau įmonių ir programų kūrėjų pradėjo naudoti RP žiniatinklio programoms, vartotojo sąsajoms ir asinchroninei įvykių valdomai programinei įrangai kurti. RP tapo ypač populiarus „JavaScript“ bendruomenėje, kur tokios bibliotekos kaip „Angular.js“ ir „Bacon.js“ priėmė reaktyviąją paradigmą, palaikančią automatinį keitimų platinimą.

Įrodyta, kad RP leidžia programų kūrėjui realizuoti reaktyviąsias programas komponavimo būdu, paverčiant abstrakcija tokias detales kaip duomenų priklausomybės aptikimas, dėl ko reaktyviąją programinę įrangą tampa lengviau suprasti [CK06; MGBCGBK09; SHM14].

Reaktyvusis programavimas – tai programavimo paradigma, pritaikyta reaktyviosioms programoms kurti. Reaktyviojo programavimo kalbų yra įvairių, tačiau jų pagrindinė mintis, kad programų kūrėjai deklaratyviai pateikia duomenų srauto aprašą programoje. Kalbos vykdyklė apdoroja platinamus keitimus ir priklausomų reikšmių perskaičiavimą, kai to reikia.

Reaktyvusis programavimas yra plačiai naudojamas programavimo modelis. Reaktyvųjį programavimą naudoja ir „Microsoft“ bei „Netflix“. Iš kūrimo perspektyvos, reaktyviojo programavimo tikslas yra sumažinti reaktyviųjų programų sudėtingumą. Be to, reaktyvusis programavimas padaro jas geriau prižiūrimas ir sumažina klaidų skaičių.

Įvykių kaupimo principo esmė – objektas yra atvaizduojamas kaip įvykių seka. Kaip pavyzdį tai galima parodyti remiantis banko sąskaita. Tarkime vartotojas, banko klientas, turi 100 eurų sąskaitos balansą. Sakykime vartotojas nusipirko prekę už 20 eurų, tada įnešė į savo sąskaitą 15 eurų ir galiausiai nusipirko tam tikrą paslaugą už 30 eurų. Akivaizdu, jog turint šią įvykių seką, galima atvaizduoti dabartinę objekto būseną - tai yra 65 eurai vartotojo sąskaitoje. Įvykių kaupimo principas užtikrina, jog visi būsenos pasikeitimai yra saugomi įvykių žurnale kaip įvykių seka [Ver12]. Įvykių kaupimo principui yra būdinga, jog įvykių negalima ištrinti bei atnaujinti, duomenys yra nekeičiami, dėl to įvykių žurnalas yra sistemos gyvavimo istorija (tiesos šaltinis). Tačiau toks modelis turi ir trūkumų. Jis nėra pritaikytas patogiam užklausų rašymui. Iš įvykių srautų yra kuriamos projekcijos, skirtos konkretiems sistemoms poreikiams, pavyzdžiui: paieškai, klasifikacijai ar ataskaitų ruošimui.

Pritaikius reaktyvų programavimą įvykių kaupimo principu paremtose sistemose būtų galima modeliuoti ne tik momentinius įvykius, tačiau turėti ir jų istoriją. Yra poreikis sukurti konkretizuotą kalbą (angl. domain specific language), kuri įgalintų paslėpti įvykių žurnalą (arba duomenų saugyklą). Pastarosios naudotojas galėtų orientuotis į pačią sprendžiamos srities problemą, nekreipdamas dėmesio į žemesnio lygio realizacijos detales. Šiuo atveju būtų galima deklaratyviai

(ką kažkuri programos dalis turi daryti) apsirašyti elgseną, nutikus įvykiui, kartu su imperatyviomis (instrukcijos, kurios aprašo, kaip programos dalys atlieka savo užduotis) struktūromis.

1.6. Pritaikymo pavyzdys

Tarkime turime domeno sritį - elektroninė komercija. Norint turėti greitą paieškos algoritmą dažnai naudojama kokia nors NoSQL duomenų bazė, pavyzdžiui ElasticSearch. Įprastas būdas perduoti produktų pakeitimus į šią duomenų bazę yra naudojant atgalinius iškvietimus:

```
class Product < ActiveRecord::Base
  after_commit :reindex_product
  after_commit :reindex_user
  # ...
end
```

Kiekvieną kartą kai `Product` yra sukuriamas, atnaujinamas arba ištrinamas, modelio informacija yra perduodama į kita duomenų saugyklą, skirtą paieškai. Dabar įsivaizduokime, programuotojas prideda stulpelį `pageviews` šiai lentelei. Jeigu programuotojas nenaudos specialių metodų atnaujinant produkto peržiūrų, skirtų išsaugoti įrašą, bet praleidžiant atgalinius iškvietimus, atsiras be galo daug atnaujinimų paieškos duomenų saugyklai, ko pasekoje visa sistema gali neatlaikyti apkrovos. Tokių atvejų praktikoje pasitaiko neretai. Dažna to priežastis yra sudėtingas sistemos suvokimas, kadangi atgaliniai iškvietimai yra išsisklaidę visoje sistemoje ir sunku pasakyti kas po ko seka. Kartais praktikoje reikia praleisti nemažai laiko derinant ir aiškinantis sistemos veikimą.

Naudojant įvykių kaupimo sistemą ir panaudojant reaktyvų programavimą, atgalinius iškvietimus būtų galima projektuoti daug konkrečiau ir vienoje vietoje. Tai galėtų atrodyti:

```
Stream.new(ProductImageUploaded, ProductInformationChanged)
    .as_persistent_type(Product)
    .each( -> (state, event) { state.reindex })
```

Čia `Stream` yra duomenų srautas. Mes nežinome kada bus gauta reikšmė, tačiau jau galime aprašyti logiką, kuri bus pritaikyta ateityje, įvykus tam tikram įvykiui. Tokiu būdu pati domeno srities logika būtų vienoje vietoje ir būtų daug lengviau suprantama. Šis apibrėžimas nurodytų kaip nutikus vienokiam ar kitokiam įvykiui, jis yra apdorojamas.

Sujungus reaktyvaus programavimo principus bei įvykių kaupimą būtų galima deklaratyviai apsirašyti skaitymo modelio kūrimo aprašą pritaikant reaktyvius operatorius. Pažvelkime į vartotojo sąskaitos `Account` skaitymo modelio kūrimo atvejį bankininkystės sistemoje:

```
Stream.new(MoneyDeposited, MoneyWithdrawn).
  as_persistent_type(Account).
  init( -> (state) { state.balance = 0} ).
  when(MoneyDeposited, -> (state, event) { state.balance += event.data[:amount]
     }).
  when(MoneyWithdrawn, -> (state, event) { state.balance -= event.data[:amount]
```

```
    })
```

Verta pastebėti, jog lokali duomenų saugykla nebuvo paminėta arba apibrėžta. Pastaroji gali būti sugeneruota bei valdoma automatiškai. Kiekvieną kartą kai sistemoje įvyksta įvykis atsinaujina sąskaitos skaitymo modelio tipas `Account(account_id: string, balance: decimal)`, o užklausos vykdomos pasinaudojant aktyvaus įrašo projektavimo šablonu, pavyzdžiui: `Account.find_by(account_id: 'LT121000011101001000')`

1.7. Tyrimo metodika

Darbo tikslui pasiekti tiriamojoje dalyje bus pasirinkta Ruby programavimo kalba bei aprašoma kūrimo metodika. Ruby leidžia programuoti tiek objektiškai, tiek funkciškai (palaiko aukštesnės eilės funkcijas).

2. Reaktyvus programavimas

Šiame skyriuje trumpai aprašomas deklaratyvus programavimas, pateikiami faktai, kodėl toliau nebus kalbama apie funkcinį reaktyvų programavimą. Toliau seka įvadas į reaktyvų programavimą, parodantis paprastą reaktyvaus programavimo pavyzdį, aprašomos išskirtinės reaktyviojo programavimo kalbos ypatybės, tada seka platesnis reaktyvaus programavimo aprašymas, minintis būdingus reaktyvaus programavimo bibliotekų bruožus. Skaitytojas supažindinamas su įvykių ir pranešimų valdomais programavimo stiliais, kuriuos aprašo reaktyvumo manifestas, ko pasekoje aprašomos dokumente apibūdinamos reaktyvios sistemos, jų bruožai ir sąvybės, reaktyvaus programavimo sąsaja su reaktyviomis sistemomis. Galiausiai skyrius konkretizuoja reaktyvaus programavimo pranašumus bei parodo jo svarbą šio metu.

2.1. Deklaratyvus programavimas

Deklaratyvus programavimas yra programavimo paradigma, išreiškianti skaičiavimo logiką, neapibūdinant jos kontrolės srauto. Kitais žodžiais, šis kompiuterinių programų kūrimo stilius aprašo kokias reikšmes reikia apdoroti, o ne tai, kaip jas reikėtų apdoroti. Laikoma, kad deklaratyvusis stilius yra iš esmės pranašesnis už lygiavertį imperatyvųjį kodą [Llo94].

2.2. Funkcinis reaktyvus programavimas

Funkcinis reaktyvus programavimas, neretai vadinamas tiesiog FRP, dažnai yra nesuprastas. Prieš 20 metų Conal Elliott labai aiškiai apibrėžė FRP [EH97]. Daugiau šio mokslų daktaro darbų susijusių su FRP galima rasti jo asmeninėje svetainėje[1]. “Lambda Jam” konferencijoje 2015 metais Conal Eliott teigia, jog FRP terminas buvo neteisingai panaudotas aprašant technologijas, tokias kaip Elm, Bacon.js, reaktyvius plėtinius (RxJava, Rx.NET, RxJS, RxRuby) bei kitas [Ell15]. Dauguma bibliotekų, kurios teigia, jog palaiko FRP, beveik vienareikšmiškai aprašo reaktyvų programavimą. Dėl šios priežasties, šiame darbe toliau bus kalbama tik apie reaktyvųjį programavimą.

2.3. Įvadas į reaktyvųjį programavimą

Reaktyvųjį programavimą galima laikyti tam tikra deklaratyviojo programavimo atmaina. Pagrindinė jo idėja – reaktyviųjų reikšmių samprata. Tokios reikšmės priklauso viena nuo kitos, jos dinamiškai keičiasi ir yra skaidriai atnaujinamos. RP tikslas – apibrėžti statinius arba dinaminius reikšmių ryšius, kai jų nereikia atnaujinti ar sinchronizuoti išreikštiniu būdu. Kodas tiesiogiai apibrėžia duomenų srautą, o vykdomoji aplinka netiesiogiai apdoroja platinamus pakeitimus. 1 kodo pavyzdyje pavaizduota reaktyviojo programavimo idėja imperatyvioje nuoseklių komandų aplinkoje. Reikšmė `sum` nurodo skaičiavimą `a + b`. Kai kuri nors iš šių reikšmių pasikeičia, `sum` reikšmė išlieka teisinga.

```
a <- 0
```

[1] http://conal.net/papers/frp.html

```
b <- 5
sum <- a + b
a <- 40
print(sum) # Prints 45
```

Kodo pavyzdys 1: Reaktyvaus programavimo pseudokodas

Dinaminių kalbų atžvilgiu, RP galima apibrėžti kaip asinchroninį, kadangi visos reikšmės yra uždariniai visoje aplinkoje. Pavyzdžiui, programavimo kalboje „Ruby“ anksčiau pateiktą elgseną galima būtų sumodeliuoti naudojant `lambda` funkciją be argumentų: `sum = -> { a + b }`. Pilnas pseudokodo realizacijos Ruby kalboje iliustracija pateikta 2 kodo pavyzdyje.

```
a = 0
b = 5
sum = -> { a + b }
a = 40
p sum.call # Prints 45
```

Kodo pavyzdys 2: Reaktyvaus programavimo pavyzdys Ruby kalboje

Reaktyvumo koncepcija leidžia programuotojui reikšmes apibrėžti tik kartą ir nereikia iš naujo įvertinti išraiškos, kai pasikeičia dešinė aprašo pusė. Kai yra kompleksinių reikšmių, programuotojams nereikia tiesiogiai paskelbti funkcijos jai apskaičiuoti, taip pat nereikia perduoti teisingų argumentų iškvietimo metu.

2.4. Išskirtinės reaktyviojo programavimo kalbos ypatybės

Reaktyviojo programavimo kalbos suteikia (reaktyviąsias) reikšmes ir (reaktyviuosius) operatorius.

2.4.1. Reaktyviosios reikšmės

Reaktyviosios reikšmės yra įvykiai ir elgsenos [BCCMM13].

- Įvykiai – reikšmė, kuri teikia begalinį keitimų srautą. Reikšmė pateikia keitimų įvykius, kurie gali turėti reikšmę kaskart, kai įvyksta keitimas. Kitaip nei elgsena (paaiškinta toliau), įvykių srautas neturi reikšmės.
- Elgsena – tai reikšmė, kuri keičiasi laike [EH97]. Apskritai ji reiškia funkcinę priklausomybę (elgsenos išraiška) nuo kitų elgsenų. Jos reikšmė gaunama vertinant elgsenos išraišką. Kai kuriose reaktyviojo programavimo kalbose elgsena dar vadinama signalu.

Daugelyje reaktyviojo programavimo kalbų galima naudoti įvykius ir elgsenas. Kai kuriose kalbose galima naudoti tik elgsenas, nes jas galima realizuoti kaip įvykių apibendrinimą [CC13; WTH02]. Šiose kalbose kūrėjas elgseną gali laikyti reikšmę turinčiu įvykiu.

Aprašant reaktyvųjį programavimą, daugelis elgsenos ir įvykių koncepcijų veikia vienodai. Be to, kalbos, kurios palaiko abi šias koncepcijas, leidžia operatoriams keisti įvykį į elgseną ir atvirkščiai. Todėl šiame darbe terminą „elgsena“ naudosime elgsenoms ir įvykiams aprašyti.

2.4.2. Reaktyvieji operatoriai

M.Viering išskiria reaktyvius operatorius kaip išskirtinę reaktyviojo programavimo ypatybę [Vie15]. Reaktyviojo programavimo kalbos operatoriai veikia su įvykiais ir elgsenomis. Šie operatoriai gali transformuoti arba jungti elgsenas į naują elgseną. Reaktyviojo programavimo kalbos teikiami operatoriai skirtingose kalbose yra skirtingi. Tačiau paprastai yra operatorius, skirtas transformuoti elgseną (map), operatorius, skirtas sujungti elgsenas į vieną elgseną (merge), ir operatorius, kuris veikia esant būsenai (fold). Kadangi šie reaktyviojo programavimo kalbų operatoriai yra dažnai sutinkami, toliau jie yra plačiau paaiškinami.

- **map** - „map“ operatorius taiko funkciją `f` elgsenai `b`. Šio proceso metu sukuriama nauja elgsena ir šios naujos elgsenos reikšmė yra funkcija `f`, taikoma elgsenos `b` reikšmei.
- **merge** - „merge“ operatorius sujungia dvi elgsenas į naują elgseną. Naujos elgsenos reikšmė yra naujausios pakeistos elgsenos reikšmė.
- **fold** - „fold“ operatorius elgsenoje `b` apima funkciją `f` ir reikšmę `v`. Jis sukuria naują elgseną `b_new`. Pradinė `b_new` reikšmė yra `v`. Kai `b` reikšmė pasikeičia, `b_new` reikšmė pasikeičia į `f` (sena `b_new` reikšmė, `b` reikšmė). Aprašant laisvai, `b_new` reikšmė `f` taikoma senai `b_new` reikšmei ir `b` reikšmei.

2.5. Reaktyvus programavimas

Reaktyvusis programavimas, kurio nereikėtų painioti su funkciniu reaktyviuoju programavimu, tai asinchroninio programavimo porūšis ir paradigma, kai logiką valdo atsiradusi nauja informacija, o ne kontrolės srautą valdo vykdymo gija.

Jis palaiko problemos suskaidymą į keletą atskirų veiksmų, kurių kiekvieną galima įvykdyti asinchroniniu ir neblokuojančiu būdu ir tada iš jų sudaryti darbo eigą, kurios įvestys ir išvestys gali būti begalinės.

„Oxford Dictionary“ žodį „asynchronous“[2] apibrėžia kaip „nesantį ar nevykstantį tuo pačiu metu“, o tai šiame kontekste reiškia, kad pranešimo ar įvykio apdorojimas vyksta tam tikru atsitiktiniu laiku, tikėtina, ateityje. Tai labai svarbus reaktyviojo programavimo metodas, kadangi jis įgalina neblokuojantį vykdymą, kai vykdymo gijoms, konkuruojančioms dėl bendrai naudojamo ištekliaus, nereikia laukti dėl užblokavimo (dėl ko vykdymo gija negali atlikti kito darbo, kol neatliktas dabartinis) ir jos gali atlikti kitą naudingą darbą, kol išteklius yra užimtas. Amdalo dėsnis teigia [Rod85], kad konkurencija yra didžiausias išplečiamumo priešas, taigi reaktyvioji programa turėtų retai blokuoti arba neblokuoti visai.

[2] http://www.oxfordlearnersdictionaries.com/definition/english/asynchronous?q=asynchronous

Reaktyvųjį programavimą iš esmės valdo įvykiai, skirtingai nuo reaktyviųjų sistemų, kurias valdo pranešimai. Skirtumas tarp įvykių valdomo ir pranešimų valdomo programavimo bus paaiškintas tolesnėje šio darbo dalyje.

Reaktyviojo programavimo programų sąsajos (API) bibliotekos paprastai yra:

- Atgalinių iškvietimų valdomos – anoniminiai pašalinio veikimo atgaliniai iškvietimai yra susieti su įvykių šaltiniais ir iškviečiami, kai įvykiai praeina pro duomenų srauto grandinę.
- Deklaratyvios – naudojamas funkcinis komponavimas ir paprastai nusistovėję kombinatoriai, pvz., perkoduoti, filtruoti, perlenkti ir kt.

Šį programavimo metodą palaikančių programavimo abstrakcijų pavyzdžiai:

- Ateitis / pažadai (angl.: Futures / Promises) – atskiros reikšmės konteineriai, semantika „daugelis skaito / vienas rašo“, kai galima įtraukti asinchronines reikšmės transformacijas, net jei ji dar neprieinama. [JC15]
- Srautai – reaktyvieji srautai: begaliniai duomenų apdorojimo srautai, įgalinantys asinchronines, neblokuojančias, kompensuojančias apkrovą transformacijos komandų grandines tarp daugybės šaltinių ir paskirties vietų.
- Duomenų srauto kintamieji – atskirai priskiriami kintamieji (atminties elementai), kurie gali būti priklausomi nuo įvesties, procedūrų ir kitų elementų, kad jie būtų automatiškai atnaujinami po pakeitimų. Praktinis to pavyzdys yra skaičiuoklės, kuriose pasikeitus langelio reikšmei, tai paveikia visas priklausomas funkcijas ir dėl to sukuriamos naujos reikšmės.

Populiarios bibliotekos, JVM palaikančios reaktyviojo programavimo metodus, apima, bet neapsiriboja, „Akka Streams“[3], „Ratpack“[4], „Reactor“[5], „RxJava“[6] ir „Vert.x“[7]. Šios bibliotekos realizuoja reaktyviųjų srautų specifikaciją, kuri yra reaktyviojo programavimo bibliotekų JVM funkcinio suderinamumo standartas ir pagal aprašymą yra „...iniciatyva pateikti asinchroninio srauto apdorojimo su neblokuojančiu apkrovos kompensavimu standartą.“

2.6. Įvykių valdomas ir pranešimų valdomas programavimas

Kaip jau minėta, reaktyvusis programavimas (pagrįstas skaičiavimu naudojant efemeriškas duomenų srauto grandines) paprastai valdomas įvykių, o reaktyviosios sistemos (pagrįstos paskirstytųjų sistemų komunikacijos ir koordinavimo atsparumu ir lankstumu) yra valdomos pranešimų (dar vadinama duomenų siuntomis).

Pranešimų valdomos sistemos su ilgalaikiais adresuojamaisiais komponentais ir įvykių bei duomenų srautų valdomo modelio pagrindinis skirtumas yra tas, kad pranešimai yra iš principo

[3] http://doc.akka.io/docs/akka/2.4.3/scala/stream/index.html
[4] https://ratpack.io/
[5] https://projectreactor.io/
[6] https://github.com/ReactiveX/RxJava
[7] http://vertx.io/

nukreipti, o įvykiai – ne. Pranešimai turi aiškią (vieną) paskirties vietą, o įvykiai yra faktai, kuriuos kiti turi stebėti. Be to, pageidautina, kad pranešimų siuntimas būtų asinchroninis, kai siuntimas ir gavimas yra atitinkamai atskirtas nuo siuntėjo ir gavėjo.

Reaktyvumo manifestas šių sąvokų skirtumą apibrėžia taip [BFKT14]:

„Pranešimas – tai duomenų elementas, siunčiamas į konkrečią paskirties vietą. Įvykis – tai signalas, komponento skleidžiamas pasiekus tam tikrą būseną. Pranešimų valdomoje sistemoje adresuojami gavėjai laukia pranešimų ir reaguoja į juos, o kitu atveju būna neveiklūs. Įvykių valdomoje sistemoje pranešimų klausytojai yra susieti su įvykių šaltiniais ir yra iškviečiami, kai paskleidžiamas įvykis. Tai reiškia, kad įvykių valdoma sistema yra pagrįsta adresuojamais įvykių šaltiniais, o pranešimų valdoma sistema koncentruojasi į adresuojamus gavėjus.“

Pranešimai reikalingi komunikacijai tinkle ir sudaro komunikacijos paskirstytosiose sistemose pagrindą, o įvykiai skleidžiami lokaliai. Pažvelgus giliau, paprastai pranešimų siuntimas naudojamas norint įveikti įvykių valdomos sistemos ribotumą tinkle, siunčiant įvykius pranešimų viduje. Tai leidžia išlaikyti santykinį įvykių valdomo programavimo modelio paskirstytajame kontekste paprastumą ir gali puikiai tikti specializuotiems ir tinkamos aprėpties naudojimo atvejams (pvz., „AWS Lambda“, paskirstytojo srauto apdorojimo produktai kaip „Spark Streaming“, „Flink“, „Kafka“ ir Akka Streams“ naudojant „Gearpump“, ir paskirstytieji publikavimo-prenumeravimo produktai kaip „Kafka“ ir „Kinesis“).

Tačiau reikalingas kompromisas: gaunamas abstrahavimas ir programavimo modelio paprastumas, tačiau prarandama kontrolė. Pranešimų siuntimas verčia mus susidurti su paskirstytųjų sistemų realybe ir apribojimais – tokiais dalykais kaip dalinės triktys, trikčių aptikimas, pamesti / besidubliuojantys / pertvarkyti pranešimai, galutinis nuoseklumas, kelių vienalaikių realybių valdymas ir kt. Pavyzdžiui, prieš pradėdamas kurti “Akka” produktą, Jonas Boner susidūrė su minėtomis išplečiamumo bei atsparumo problemomis[8] šiose technologijose: “EJB”, “RPC”, “CORBA” ir “XA”.

Šie semantikos ir pritaikomumo skirtumai turi didelę įtaką programų kūrimui, įskaitant tokius dalykus kaip atsparumas, lankstumas, mobilumas, vietos skaidrumas ir kompleksiškas paskirstytųjų sistemų valdymas.

Reaktyviojoje sistemoje, ypač tokioje, kurioje naudojamas reaktyvusis programavimas, bus naudojami ir įvykiai, ir pranešimai, kadangi vienas yra puikus komunikacijos (pranešimai) įrankis, o kitas – puikus būdas išreikšti faktus (įvykiai).

2.7. Reaktyviosios sistemos ir architektūra

Kaip apibrėžiama Reaktyvumo manifeste, reaktyviosios sistemos – tai architektūrinių kūrimo principų rinkinys, skirtas modernioms sistemoms, galinčioms patenkinti augančius šiuolaikinių sistemų poreikius, kurti.

Reaktyviųjų sistemų principai toli gražu nėra nauji ir juos galima aptikti jau 8-ajame ir 9-ajame dešimtmetyje, reikšmingame Jim Gray darbe apie „Tandem“ sistemą [Gra86] ir Joe Armstrong darbuose apie „Erlang“ [Arm97]. Vis dėlto šie žmonės pralenkė laiką, kadangi tik per pasta-

[8] http://readwrite.com/2014/07/10/akka-jonas-boner-concurrency-distributed-computing-internet-of-things/

ruosius 5–10 metų technologijų pramonė buvo priversta peržiūrėti esamas įmonių sistemų kūrimo geriausias praktikas ir išmoko reaktyvumo principų žinias pritaikyti šiuolaikiniame pasaulyje, kurį apibrėžia daug branduolių, debesų kompiuterija ir internetu sąveikaujantys įrenginiai (angl. „Internet of Things“).

Reaktyviosios sistemos pagrindą sudaro pranešimų perdavimas, sukuriantis laikiną ribą tarp komponentų, leidžiančią juos atskirti laike (tai įgalina vienalaikiškumą) ir erdvėje (tai įgalina paskirstymą ir mobilumą). Toks atskyrimas būtinas norint visiškai izoliuoti komponentus ir užtikrina atsparumą bei lankstumą.

Programos aprašytos Reaktyvumo manifeste turėtų:

- Reaguoti į įvykius: įvykiais paremta prigimtis įgalina sekančias sąvybes.
- Reaguoti į apkrovą: koncentruotis į išplečiamumą, o ne našumą vienam vartotojui.
- Reaguoti į trikdžius: kurti atsparias sistemas, turinčias galimybę atkurti būseną bet kuriuo metu.
- Reaguoti į vartotojus: apjungti paminėtus bruožus siekiant užtikrinti interaktyvią vartotojo patirtį.

2.8. Reaktyviojo programavimo pranašumai

Pagrindiniai reaktyviojo programavimo pranašumai: didesnis skaičiavimo išteklių naudojimas kelių branduolių ir kelių CPU aparatūroje ir didesnis našumas sumažinant nuoseklinimo taškus pagal Amdalo dėsnį [Rod85] ir, žiūrint plačiau, pagal Giunterio universaliojo išplečiamumo dėsnį [Gun98].

Papildomas pranašumas yra programų kūrėjo produktyvumas, kadangi tradicinėms paradigmoms sunkiai sekėsi pateikti paprastą ir įgyvendinamą metodą asinchroniniam ir neblokuojančiam skaičiavimui ir I/O apdoroti. Reaktyvusis programavimas susitvarko su daugeliu iš šių iššūkių, kadangi jis paprastai pašalina tiesioginio koordinavimo tarp aktyvių komponentų poreikį.

Reaktyvusis programavimas ypač sėkmingai taikomas komponentų kūrimo ir darbo eigų komponavimo srityse. Norint išnaudoti visas asinchroninio vykdymo galimybes ir išvengti perteklinio naudojimo arba begalinio išteklių naudojimo, būtina įtraukti apkrovos kompensavimą.

Nors reaktyvusis programavimas yra labai naudingas kuriant modernią programinę įrangą, galvojant apie sistemą aukštesniu lygmeniu, reikia naudoti kitą įrankį – reaktyviąją architektūrą – reaktyviųjų sistemų kūrimo procesą. Taip pat svarbu prisiminti, kad yra daugybė programavimo paradigmų ir reaktyvusis programavimas yra tik viena iš jų. Kaip ir bet koks kitas įrankis, jis nėra skirtas visiems galimiems atvejams.

2.9. Reaktyviojo programavimo svarba šiais laikais

Paskelbus Reaktyvumo manifestą, „Scala“ kūrėjas Martin Odersky, reaktyviųjų plėtinių kūrėjas Erik Meijer ir „Akka“ techninis vadovas Roland Kuhn „Coursera“ paskelbė nemokamą kursą

„Principles of Reactive Programming“[9] (Reaktyviojo programavimo principai):

„Antrojo kurso tikslas – išmokti reaktyviojo programavimo principus. Reaktyvusis programavimas yra naujai atsirandantis mokymo dalykas, apimantis vienalaikiškumą ir įvykiais pagrįstas bei asinchronines sistemas. Jis būtinas rašant bet kokio tipo žiniatinklio paslaugą arba paskirstytąją sistemą ir taip pat sudaro daugelio didelio našumo vienalaikių sistemų pagrindą. Reaktyvųjį programavimą galima vertinti kaip aukštesnio lygmens funkcinio programavimo vienalaikių sistemų natūralų plėtinį, kuris tvarko paskirstytąją būseną, koordinuodamas ir organizuodamas asinchroninių duomenų srautus, kuriais keičiasi veikėjai.“

RP plačiai naudoja „Netflix“ [BC13], ji netgi skyrė „Rx“ „Java“ prievadą:

„Reaktyvusis programavimas su „RxJava“ leido „Netflix“ kūrėjams panaudoti serverio vienalaikiškumą, nepatiriant įprastų su gijomis susijusių ir sinchronizavimo problemų. API paslaugos lygmens realizavimas kontroliuoja vienalaikiškumo primityvus, kurie leidžia mums gerinti sistemos našumą, nesibaiminant sugadinti kliento kodo. „RxJava“ efektyviai veikia mums serveryje ir kuo daugiau naudojame, tuo giliau įsiskverbia į mūsų kodą.“

2013 metais „Facebook“ taip pat išleido „React JavaScript“ biblioteką, skirtą kitos kartos vartotojo sąsajoms kurti. „Facebook“ inžinierius Stoyan Stefanov aprašo pagrindinę „React“ koncepciją [Ste13]:

„React“ leidžia kurti programą naudojant komponentus, kurie gali atvaizduoti kai kuriuos duomenis. Kai duomenys pasikeičia, komponentai labai efektyviai ir tik tuomet, kai reikia, automatiškai atnaujinami. O visomis įvykių doroklių prijungimo ir atjungimo užduotimis pasirūpinama už jus. Taip pat efektyvu naudoti ir perdavimą.“

Verta pastebėti, kad RP jau plačiai naudoja vartojo sąsajos kūrėjų bendruomenė, kurie maždaug 2009 m. pradėjo naudoti originalų „Flapjax“ dokumentą [MGBCGBK09] ir toliau naudojo kelias bibliotekas, kuriose buvo realizuoti RP principai, pvz., „Bacon.js“[10], „Knockout“[11], „Meteor“[12], „React.js“[13], „Reactive.coffee“[14] ir „RxJS“[15].

[9] https://www.lightbend.com/blog/principle-of-reactive-programming-coursera
[10] https://baconjs.github.io/
[11] http://knockoutjs.com/
[12] https://www.meteor.com/
[13] https://facebook.github.io/react/
[14] http://yang.github.io/reactive-coffee/
[15] http://reactivex.io/

3. Įvykių kaupimas

Šis skyrius aprašo įvykių kaupimą, susijusią terminologiją. privalumus bei trūkumus, įvykių srautus bei panaudojimo atvejus. Taip pat aprašomas architektūrinis stilius - komandų-užklausų atsakomybių atskyrimas (CQRS), kuris yra naudojamas įvykių kaupimu pagrįstose sistemose.

3.1. Aktyvus įrašas

Dauguma programų sistemų reikalauja tam tikro duomenų valdymo. Įprastas būdas išlaikyti dabartinę programos būseną yra naudojant tam tikrą duomenų saugyklą (pvz.: duomenų bazę) ir gražinti saugomų objektų būseną panaudojant užklausą. Aktyvus įrašas (angl. Active Record) yra programinės įrangos kūrimo šablonas darbui su dabartine programos būsena pritaikant sukūrimo, skaitymo, atnaujinimo ir ištrynimo operacijas (CRUD) objektams reliacinėje duomenų bazėje [Fow02] (1 a) pav.). Taikant šį programinės įrangos kūrimo šabloną, tik dabartinė objekto būsena yra palaikoma ir manipuliuojama, buvusios objekto reikšmės yra perrašomas ir neišsaugoti pakeitimai prarandami. Šis kelias puikiai tinka daugumai programų, bet turi trūkumų kalbant apie veiksmų atsekamumą bei operacijas darbui su programos būsenų istorija.

3.2. Įvykių kaupimo apibrėžimas

Akademinė literatūra šia tema yra ganėtinai reta. Daugiausia informacijos galima rasti internetiniuose dienoraščiuose, prezentacijose bei programinės įrangos dokumentacijose. Šia tema nėra standartizuoto žodyno, terminologija ir apibrėžimai skiriasi priklausomai nuo autoriaus.

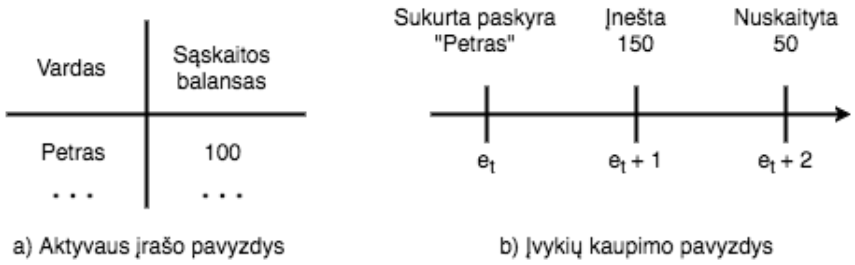
Pirmą kartą įvykių kaupimo terminas 2005 metais paminėjo Martin Fowler savo internetiniame dienoraštyje [Fow15a]. Jis aprašo įvykius kaip eilę programos būsenų pasikeitimų. Šie įvykiai saugo visą informaciją, reikalingą dabartinės būsenos atkūrimui. Įvykiai niekada neištrinami. Vienintelis būdas anuliuoti įvykį yra sukurti grįžtamąjį įvykį (angl. retroactive event) [Fow15b]. Grįžtamasis įvykis grąžina programos būseną į tokia būseną lyg praeitas įvykis nebūtų nutikęs.

Kitas žymus autorius minint įvykių kaupimą yra Greg Young. Jis apibūdina įvykių kaupimą kaip “dabartinės būsenos saugojimą kaip eilę įvykių bei sistemos būsenos atkūrimą pakartojant tą pačią įvykių seką“ [You10b]. Jo požiūriu, įvykių žurnalas skirtas tik įrašymui: “įvykiai yra lyg faktai. Jie įvyko ir negali būti anuliuoti” [You10a]. Tai ką Martin Fowler vadina grįžtamaisiais įvykiais, Greg Young apibūdina kaip grįžtamąsias operacijas (angl. reversal transactions).

Martin Krasser straipsniai bei prezentacijos apie Akka priemonių rinkinį įrašymo moduliui aprašo dar vieną požiūrį į įvykių kaupimą [Kra13; Kra15]. Šiame kontekste, išsiskaidžiusioje sistemoje aktoriai komunikuoja žinutėmis, kurios pakeičia būseną. Įvykių kaupimas naudojamas perduoti pakitimus aktoriui. Būsenos pakitimai yra pridedami kaip nesikeičiantys faktai į įvykių žurnalą. Šis sprendimas yra motyvuojamas tuo, jog “šis būdas leidžia pasiekti labai aukštą operacijų kiekį ir įgalina efektyvias replikacijas”. Aktoriaus būsenos atkūrimas (po perkrovimo ar klaidos) yra pasiekiamas pritaikant jau įrašytus įvykius.

Visiems apibrėžimams yra būdingas vienas bruožas - publikuoti kiekvieną objekto (sistemos ar programos) būsenos pasikeitimą kaip nesikeičiantį įvykį į nemodifikuojamą žurnalą. To pasekoje

skaitant įvykius iš eilės galima atkurti dabartinę būseną (1 b) pav.).



1 pav. Du skirtingi būdai saugoti dabartinę būseną

3.3. Įvykių kaupimo privalumai

Literatūros šaltiniuose dažnai randama rekomendacijų taikyti įvykių kaupimą tik tam tikrose, aiškiai apibrėžtuose sistemos dalyse ir nenaudoti šio principo ten, kur tai nėra svarbu [BDMSS13]. Sudėtinga verslo logika yra dažnas įvykių kaupimu pagrįstų sistemų bruožas. Tai yra priešingybė pavyzdžiui programoms, kurios suteikia galimybę keisti reikšmes vartotojo sąsajoje ir saugoti reliacinėje duomenų bazėje. Kad būtų galima įvertinti taikymo galimybes, reikia aiškiai apibrėžti, kokius privalumus gali suteikti įvykiu kaupimo principu pagrįstos architektūros.

1. **Audito žurnalas**. Reguliuojamose srityse (pvz. finansų industrija), daugelyje šalių valstybės nuostatai reikalauja kompanijų saugoti operacijų istoriją sistemoje. Pavyzdžiui, JAV reikalauja tarpininkų saugoti įrašus neperrašomu ir neištrinamu formatu [SC03]. Įvykių kaupimo principas puikiai tinka šiam reikalavimui įgyvendinti, nes įvykių žurnalas pasižymi tik įrašymo funkcionalumu, o patys įvykiai yra nekintantys.

2. **Derinimas**. Sukaupti įvykiai gali būti panaudoti analizei kaip sistema pasiekė vieną ar kitą būseną ir kurie įvykiai tai įtakojo. Įvykių kaupimo stiprioji pusė yra atsekamumas ir derinimo galymybės - įmanoma atsekti iš kur kilo sistemos klaida.

3. **Išplečiamumas**. Faktas, jog į įvykių žurnalą galima tik įrašyti, yra naudingas išplečiamoms architektūroms. Gana dažnai įvykių kaupimo principu paremtose sistemose turima keletą duomenų modelio kopijų. Norint užtikrinti darną šios kopijos turi būti sinchronizuotos. Tokiose sistemose vienintelis būdas suvienodinti duomenų modelio kopijas yra įrašant įvykius. Manoma, jog toks sprendimas turi mažiau blokavimų ir lengviau išplečiamas skaitymui negu architektūra atnaujinant duomenų modelį [Doc15].

4. **Informacinė nauda**. Visos buvusios sistemos būsenos gali būti tiek atkurtos, tiek gaunamos užklausų pagalba. Tai gali atnešti papildomos naudos sistemoms, kuriose kliento elgesys yra svarbus. Tokiose sistemose dažniausiai nežinoma kokio tyrimo reikės iš verslo pusės. Pavyzdžiui, elektroninės komercijos parduotuvėje gali būti naudinga gauti ir nagrinėti visus produktus, kurie kada nors buvo išimti iš apsipirkimo krepšelio. Įvykių kaupimo sistemose, toks uždavinys gali būti lengvai įgyvendintas.

3.4. Panaudojimo atvejai

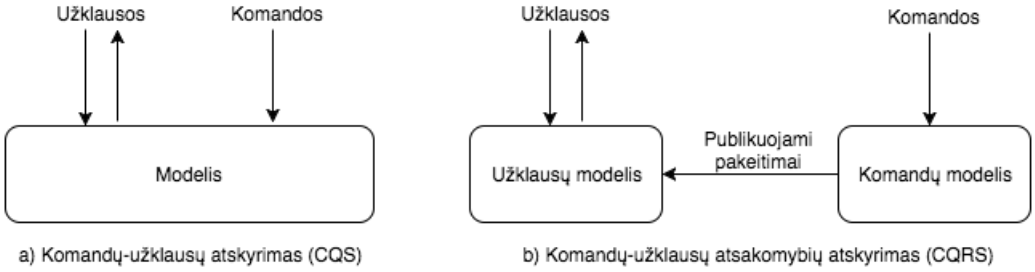
Internete galima rasti straipsnių aprašančių kaip įvykių kaupimo principas buvo panaudotas tikrose sistemose. Galima pažymėti dvi išsamias dokumentacijas: finansinės prekybos platforma LMAX [Fow11] ir Microsoft Windows Azure komandos analizė [BDMSS13]. Taipogi yra sukurti keli pragramavimo karkasai paremti įvykių kaupimu: Akka[16] ir Event Store[17] bei Red Bull Media House kompanijos programavimo priemonių rinkinys Eventuate[18].

3.5. Komandų-užklausų atskyrimo principas (CQS)

Komandų-užklausų atskyrimo (angl. Command Query Separation (CQS)) principą pirmą kartą aprašė Bertrand Meyer [Mey88] norėdamas patobulinti šalutinių efektų apdorojimą kuriant programą ar projektuojant API. Pagrindinė idėja yra atskirti prieigą prie objektų į:

1. **Užklausas**, kurios grąžina informaciją,
2. **Komandas**, kurios keičia būseną (2 a) pav.).

Užklausos neturėtų sukelti pašalinių efektų. Kitais žodžiais tariant - klausimas neturėtų pakeisti atsakymo.



2 pav. CQS ir CQRS palyginimas

3.6. Komandų-užklausų atsakomybių atskyrimas (CQRS)

Komandų-užklausų atsakomybės atskyrimas (angl. Command Query Responsibility Segregation (CQRS)) – šablonas, kurį pirmasis aprašė Greg Young [You10a]. Kai kuriuose ankstesniuose šaltiniuose šis šablonas aprašomas kaip plėtinys arba specialus komandų-užklausų atskyrimo principo atvejis, tačiau šiandien jis laikomas atskiru šablonu, sukurtu CQS principu. Pagal CQRS teigiama, kad užklausoms vykdyti naudojamas skirtingas modelis (tai yra komponentas arba objektas) nei modelis, skirtas komandoms vykdyti (2 b) pav.). CQS padalija atsakomybę pagal kodo lygį, suskirstydamas metodus į užklausas ir komandas; CQRS analizuoja tai dar išsamiau ir netgi skirsto objektus į du tipus: skaitymo arba rašymo. G. Young aprašė šabloną taip: „CQRS yra paprastas dviejų objektų sukūrimas iš anksčiau buvusio vieno objekto“ [You10b].

[16] http://akka.io/
[17] http://docs.geteventstore.com/
[18] https://rbmhtechnology.github.io/eventuate/

Kuriant architektūrą pagal CQRS šabloną, ilgainiui sistemoje sukuriamas nuoseklus veikimas. Užklausos ir komandos modelis yra atskiri komponentai, todėl, atsižvelgiant į jų sujungimą, jie nebūtinai turi būti sinchroniški. Jei jie sujungiami laisvai, užklausos modelyje nebūtinai kaskart bus tie patys duomenys kaip komandos modelyje. Ši charakteristika, kai užklausos modelyje gali būti laikinai nenuosekli (tai yra pasenusi) būsena, kuri kažkuriame ateities taške ilgainiui taps nuosekli, yra vadinama galutiniu nuoseklumu. Šis galutinai nuoseklus veikimas tampa itin svarbus paskirstytosioms sistemoms, kai užklausos modelis (-iai) yra fiziškai atskiriamas (-i) nuo komandos modelio (-ių). Brewer CAP teorema [FB99] teigia, kad paskirstytoji sistema turi neišvengiamai nuspręsti, kaip apdoroti nuoseklumą, nes, įvykus tinklo klaidai, ji gali užtikrinti tik dvi iš šių trijų garantijų:

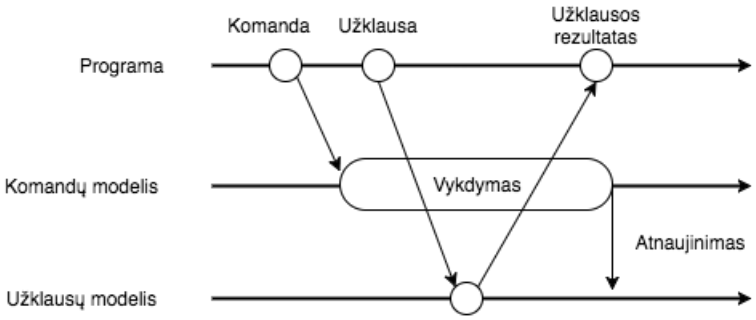
- Nuoseklumą: visi dalyviai peržiūri tuos pačius duomenis.
- Pasiekiamumą: atsakoma į kiekvieną užklausą.
- Atskyrimo toleravimą: sistema ir toliau veikia net tada, jei tarp sistemos savavališkų dalyvių prarandami pranešimai. Tai turi poveikį nuoseklumui, nes dalyviai turi sutvarkyti pranešimų praradimą. Be to, tai veikia pasiekiamumą, nes kiekvienas dalyvis turi vykdyti užklausas.

Taigi, jei paskirstytoji sistema kuriama kaip architektūros stilių naudojant CQRS, o vietoj nuoseklumo pasirenkamas atskyrimo toleravimas ir pasiekiamumas, sistema gali generuoti tik galutinį nuoseklumą. Nuslėpus faktą, kad sistema veikia galutinai nuosekliai (tai yra numatant optimistines prielaidas), gali būti daroma pavojinga klaida, nes programuotojai ir programinės įrangos komponentai tuomet veiktų pagal klaidingas prielaidas. Progresyvus nuoseklumo problemų sprendimas gali būti laikomas paskirstytosios ES+CQRS sistemos stipriąja puse, nes jis verčia programą kuriančius programuotojus atsižvelgti į tokį nuoseklų veikimą, kuris yra integruota bendros architektūros dalis. Užklausos gali visą laiką pateikti pasenusį rezultatą ir nebeaišku, kada vykdomos komandos. Pastaraisiais metais šis progresyvus požiūris į didelių paskirstytųjų sistemų problemas sukūrė tam tikrą požiūrio kampą ir sugeneravo kelis projektus. Reaktyvumo manifestas [BFKT14] yra svarbus 2013 m. pasirodęs dokumentas. Jame apžvelgiamos pagrindinės ypatybės, kuriomis pasižymi paskirstytosios sistemos projektavimas, atitinkantis įvykių kaupimą ir CQRS.

Komandos ir užklausos atskyrimas kelia vien nepatogumą, nes jis lemia sudėtingesnį sistemos kūrimą ir tampa sunku sukurti priežastiniu būdu susijusias komandų ir įvykių serijas. Komandos veikia asinchroniškai ir nepateikia reikšmės, todėl programa neturi priemonių sužinoti, kada matomas komandos rezultatas. Taigi, po komandos einanti užklausa gali pateikti pasenusį rezultatą. Tačiau tolimesnės užklausos galiausiai kažkuriuo metu pateiks nuoseklią būseną. 3 paveikslėlyje pateikiama tipiška komandų ir užklausų seka. Tokį veikimą galima apeiti, pavyzdžiui, pavėlinus užklausų vykdymą, kol užklausos modelis bus atnaujintas į tam tikrą versiją (kaip panaudota sąlyginėse užklausose platformoje „Eventuate“[19]). Tačiau reiktų pastebėti, kad šiuos apėjimo metodus galima naudoti tik retais konkrečiais atvejais, nes naudojant šiuos metodus sumažėja efektyvumas ir

[19]https://rbmhtechnology.github.io/eventuate/user-guide.html#conditional-requests

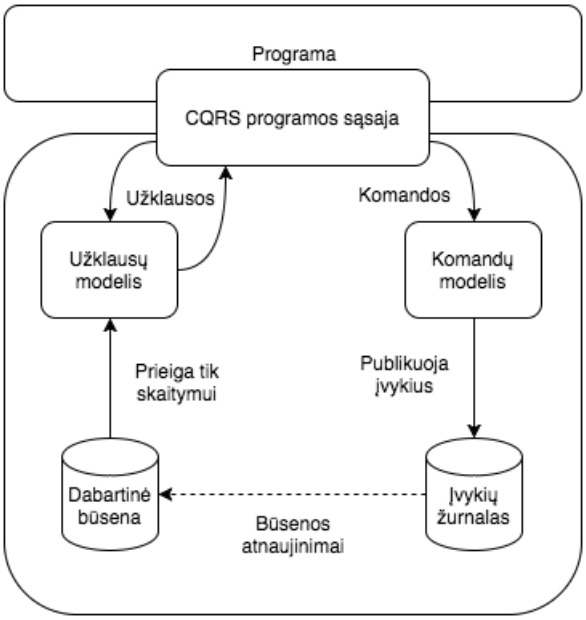
reikia daugiau dirbti su šių sistemų charakteristikomis. Geriau būtų, jei programa natūraliai galėtų apdoroti galutinai nuoseklų veikimą.



3 pav. CQRS architektūriniame modelyje, komanda ir po jos sekanti užklausa nebūtinai užtikrina duomenų darną. Komandos yra asinchroninės ir neaišku kada jos vykdomos ir baigiamos vykdyti. Iliustruojamas galutinio nuoseklumo atvejis kai komanda atnaujina užklausų modelio būseną, o jos vykdymo metu įsiterpia užklausa. Užklausa gražina dar neatnaujintus duomenis.

3.7. Įvykių kaupimo principas ir CQRS

CQRS – šablonas, suformuojantis simbiotinį ryšį su įvykių kaupimu ir dažnai yra naudojamas įvykių kaupimu pagrįstoje architektūroje. Operacijų klasifikacija būseną keičiančiose komandose ir tik skaityti skirtose užklausose puikiai dera su koncepcijomis, randamomis įvykių kaupimu pagrįstose sistemose. CQRS šablone komandos nurodo tikslus, kurie siunčiami į komandų rengyklę. Čia jos apdorojamos ir pateikiami būsenos pakeitimai – įvykiai. Tada įvykiai įtraukiami į įvykių žurnalą ir publikuojami užklausų modeliuose, kur jų būsena taip pat atnaujinama. Užklausų modeliai vykdo tik skaityti skirtas užklausas ir gali pateikti tam tikrą duomenų rodinį (pvz., sąskaitų lentelę ir jų likučius). Įvykiai naudojami kaip modelių sinchronizavimo priemonė. Taigi komandos aiškiai atskiriamos nuo kitų operacijų, kurios nepateikia papildomos operacijos, jos pridedamos įvykių žurnale. 4 paveikslėlyje pateikiama tipiška ES+CQRS architektūra: verslo logika glūdi aukštesniame taikomosios programos lygmenyje; komandos ir užklausos slepia diegimo informaciją apie tai, kaip jos vykdomos.



4 pav. CQRS ir įvykių kaupimo principais paremtos sistemos architektūra. Iliustracija paimta iš [EK14].

Komandų-užklausų atskyrimas sugeneruoja charakteristikas, kurios tinkamai veikia su įvykių kaupimu. Įvykiai nekeičiami ir pridedami tik po to, kai įvykdoma komanda, todėl įvykiai yra priemonė, leidžianti išlaikyti duomenų modelių sinchronizavimą. Užklausų modeliams tereikia gauti naujų įvykių, kad jie išliktų suderinti su komandos modeliu. Padalijus problemas objekto lygiu, palaikomas kiekvieno modelio atskiras optimizavimas. Tai itin svarbi savybė tų sistemų, kurioms taikomas reikalavimas keisti dydį, nes skaitymo ir rašymo optimizavimo technikos skiriasi. Atskyrus komandos ir užklausos modelius, tampa įmanoma jų dydį keisti atskirai. Vienas iš būdų optimizuoti skaitymo operacijas priskirtojoje sistemoje – duomenų bazės dubliavimas. Tada šie duomenų bazės dublikatai turi būti sinchronizuojami, tačiau kiekvienas iš jų gali siųsti atsakymus į užklausas. Antra vertus, optimizuoti rašymą reiškia sumažinti perteklių (ir rašymą) duomenų bazėje. To galima pasiekti atitinkamai sutvarkius vieną duomenų bazę naudojant, pvz., normalizavimo taisykles.

3.8. Domenu pagrįstas projektavimas (DDD)

Tiek įvykių kaupimas, tiek CQRS šablonas yra glaudžiai susiję su domenu pagrįstu projektavimu. DDD terminus ir koncepcijas 2003 m. aprašė Eric Evans [Eva04]. Iš esmės DDD suteikia mokomųjų principų ir komponentų rinkinį, skirtą programinės įrangos kūrimui. Pagrindinė idėja – skirti daugiausia dėmesio programinės įrangos projektui domene ir jame atliekamoms užduotims. Pagrindinė sistemos projektavimo dalis yra domeno modelio sukūrimas. Be kitų komponentų, šis modelis aprašo pagrindinio domeno užduotis ir įvykius. Šis domeno modelis sukurtas bendradarbiaujant su domenų specialistais. Todėl DDD yra panašus į aktyvius programavimo modelius, kuriuose taip pat skatinamas bendradarbiavimas su domenų specialistais (arba klientais). DDD būtina apibrėžti bendrąja kalba (skvarbiąja kalba), kuri naudojama projektavimo metu. Manoma, kad ši svarbioji kalba sukuria bendrąjį rodinį komandos nariams domeno modelyje. Veiksmažo-

džiai ir daiktavardžiai padeda aiškiai suprasti užduotis, kurias turi atlikti sistema. Be to, manoma, kad bendroji kalba apsaugo nuo nesusipratimų ir leidžia kiekvienam projekto nariui suprantamai kalbėtis su kitu nariu taip suvienydama domeno specialistus ir programuotojus bendram darbui.

DDD taikymas tik šiam susietam kontekstui ir aukšto lygio koncepcijų naudojimas siekiant sukurti abstrakciją iš žemo lygmens informacijos, sukuria sąsajas su įvykių kaupimo sritimi. Įvykių kaupimu pagrįstoje sistemoje paprastai nėra saugomi visi žemo lygio vykstantys įvykiai, čia labiau laikomi tik gana susiję riboto konteksto domeno įvykiai. Manoma, DDD tiktų sistemoms su sudėtingomis verslo taisyklėmis ir aiškiai apibrėžtu bei apribotu domenu. Tačiau paprasta sistema dėl DDD gali tapti sudėtinga ir mažinanti efektyvumą.

Įvykių kaupimo ir CQRS koncepcijos sukurtos iš domeno pagrįsto projektavimo. ES+CQRS sistemoje esančios komandos ir užklausos gali būti modeliuojamos naudojant DDD. Taip jos įgyja aiškią reikšmę domeno modelyje ir yra žinomos programuotojams ir domenų specialistams. Taip pat yra ir su sukurtais įvykiais – kaip domeno įvykiai jie tinka domeno modeliui ir jo logikai. Domeno specialistas aprašo komandas ir užklausas bei jų vidinę logiką. Tada programos kūrėjas per API gali pasiekti šias komandas ir užklausas ir įgyvendinti jas programoje. Komandų ir užklausų abstrakcijos kuriamos iš domenui būdingų sąlygų, todėl darbas su sistema palengvėja. Programos kūrėjas neturi atsižvelgti į tai, kaip komanda tikrinama, arba žinoti, kaip ji įgyvendinama viduje. Sudėtinga verslo logika gaunama pasitelkus supaprastintų komandų naudojimą.

4. Konceptuali sistema

Šioje dalyje bus bandoma parodyti, jog reaktyvų programavimą galima panaudoti įvykių kaupimo sistemose. Pirmiausia nagrinėjami jau esantys reaktyvaus programavimo bei įvykių kaupimo programavimo karkasai/bibliotekos. Kad būtų galima susidaryti bendrą vaizdą ir suvokti įvykių kaupimo sistemas, pirmiausia projektuojamas pavyzdinis įvykių kaupimo sistemos architektūros modelis. Toliau aprašomos šio pavyzdinio architektūros modelio komponentų žemesnio lygmens realizacijos detalės, kūrimo gairės. Atsižvelgiant į DDD, prieš pateikiant konkrečius pavyzdžius, skaitytojas yra supažindinamas su domeno sritimi bei galimais įvykiais. Sekančiame poskyryje aprašomas alternatyvus būdas kurti skaitymo modelį pritaikant reaktyvaus programavimo principus ir paslepiant operacijų su duomenų saugykla realizacijos detales.

4.1. Programavimo kalba

Pasirinkta griežtai tipizuota, dinaminė “Ruby” kalba, palaikanti tiek objektinį, tiek funkcinį programavimą. Kalba yra viena populiariausių atviro kodo talpykloje “Github”, plačiai naudojama informacinių technologijų industrijoje (“Github”, “AirBnb”, “Etsy”, “Shopify”), lengvai skaitoma, turi daugybę kokybiškų bibliotekų bei atitinka darbo tiksle išsikeltą sąlygą.

4.2. Reaktyvus programavimas Ruby kalboje

Žemesniame lygyje paradigma iš esmės yra dviejų kūrimo šablonų konstrukcija, naudojama jau daugiau nei 20 metų. Faktiškai viena iš populiariausių kūrimo šablonų knygų yra „Gang of Four“, kurioje aprašyti šie du kūrimo šablonai: iteratoriaus ir stebėtojo šablonai [GHJV95]. Tai yra du elgsenos kūrimo šablonai, charakterizuojantys objektų ir klasių sąveiką ir atsakomybę.

4.2.1. Iteratorius

Pagrindinė šio šablono (iteratoriaus) idėja – atsakomybė už sąrašo objekto prieigą ir perėjimą. Iteratoriaus klasė apibrėžia prieigos prie sąrašo elementų sąsają. Iteratoriaus objektas atsakingas už esamo elemento stebėjimą; t. y. jis žino, kurie elementai jau buvo pereiti.

„Ruby“ programavimo kalboje `Enumerable` modulis kaip tik tai ir daro: pateikia surašytuvą, kuriame yra uždari duomenys ir perėjimo metodai. Parašykime trumpiau: „Ruby“ iteratorius – tai klasės, į kurią įtrauktas `Enumerable` modulis, egzempliorius, kuris yra beveik bet koks rinkinys.

4.2.2. Stebėtojas

Stebėtojo šablonas yra kūrimo šablonas, kuriame objektas (vadinamas tema) tvarko jo priklausinių (vadinamų stebėtojais) sąrašą ir automatiškai praneša apie bet kokius būsenos pasikeitimus, paprastai iškviesdamas vieną iš jų metodų. Paprastai jis naudojamas paskirstytųjų įvykių tvarkymo sistemoms realizuoti.

Iš esmės temos yra objektai, kurie siunčia pranešimus į objektus, stebinčius tokias temas. Temos priverstinai įkelia pranešimus stebėtojams, todėl tai ir vadinama stebėtojo šablonu.

„Ruby“ programavimo kalba turi `Observable` modulį standartinėje bibliotekoje, kurioje pateikiamas paprastas mechanizmas, skirtas vienam objektui (temai) informuoti prenumeratorių rinkinį (stebėtojus) apie bet kokį būsenos pasikeitimą.

4.2.3. Reaktyvaus programavimo bibliotekos

Darbo autoriui pavyko rasti 2 reaktyvaus programavimo bibliotekas “Ruby” kalboje, todėl jos bus plačiau aptartos.

- “RxRuby”[20] - 595 žvaigždutės bei 52 išsišakojimai, paskutinis atnaujinimas įvyko 2017 metų sausį. (Žiūrėta 2017-01-08)
- “Frappuccino”[21] - 370 žvaigždučių bei 32 išsišakojimai, paskutinis atnaujinimas įvyko 2016 metų vasarį. (Žiūrėta 2017-01-08)

“RxRuby” biblioteka turi labiau apribojančią “Apache” licenziją (2 versija) lyginant su “Frappuccino” MIT licenzija.

4.2.4. RxRuby

Pirmiausia pažiūrėkime kaip veikia “RxRuby” paprasčiausioje formoje:

```
RxRuby::Observable.just(7)
```

`RxRuby::Observable` yra srautas. Srautas yra tema (arba objektas), kurį galima prenumeruoti (arba stebėti). `RxRuby::Observable` modulis pats nedaro nieko, nebent kuris nors jo metodas bus iškviestas. Tęskime pavyzdį:

```
stream = RxRuby::Observable.just(7)
stream.subscribe { |num| puts "Gautas skaicius #{num}" }
```

Objektas, kurį gauname, yra tiesiog skaičius 7, apgaubtas kaip tema. Kadangi tema realizuoja daugumą `Enumerable` modulio metodų, galima su juo daryti praktiškai bet ką. Tačiau, kad išvengtume sudėtingumo šiame pavyzdyje, tiesiog prenumeruokime temą ir perduokime `lambda` konstrukciją. `Lambda` bus iškviesta kiekvieną kartą, kai srautas gaus informaciją (šiuo atveju tik kartą).

Šio pavyzdinio atvejo išvestis bus:

```
Gautas skaicius 7
```

Kadangi turime daug `Enumerable` modulio metodų, kuriuos galime naudoti, galime kažką nuveikti su masyvu. Yra 2 būdai dirbti su masyvais “RxRuby” bibliotekoje: naudojant rėžį ir paprastus masyvus.

[20] https://github.com/ReactiveX/RxRuby
[21] https://github.com/steveklabnik/frappuccino

```
RxRuby::Observable.range(1,10)
  .select { |num| num.even? }
  .sum
  .subscribe { |s| puts "Suma lyginiu skaiciu tarp 1 ir 10 yra: #{s}" }
```

Grąžins:

```
Suma lyginiu skaiciu tarp 1 ir 10 yra: 30
```

Paanalizuokime šį pavyzdį. Pirmiausia sukuriamas temos rėžis, su skaičiais tarp 1 ir 10. Tada iškviečiamas filtravimo operatorius `select` temai, perduodant `lambda` kaip parametrą. `lambda` priima kiekvieną rėžio skaičių kaip parametrą ir filtruoja visus lyginius temos skaičius, grąžindama naują temą. Iškvietus `sum` operatorių, susumuojami visi lyginiai skaičiai.

Galiausiai operatoriaus `sum` grąžinama tema yra prenumeruojama. Prenumeruojant tema (arba duomenų srautas) tiesiog stebima ir `lambda` iškviečiama. Kiekvieną kartą, kai tema gauna naują informaciją, duomenys perduodami lyg per "piltuvėlį" ir paskutinė `lambda` atspausdina rezultatą.

4.2.5. Frappuccino

Steve Klabnik, žymus žmogus Ruby bendruomenėje dėl savo įnašo į atviro kodo projektus[22], 2013 metais pristatė reaktyvaus programavimo idėjas Ruby kalboje įvairiose konferencijose, tarp jų Euruko2013 [Kla13a], RubyConf India 2013 [Kla13b] ir kitos.

Panagrinėkime "Frappuccino" bibliotekos panaudojimo pavyzdį, pavaizduotą 3 kodo pavyzdyje (priedas nr. 1). Ši biblioteka, kaip ir RxRuby, naudoja stebėtojo projektavimo šabloną. Sukuriant srautą, klasė `Button` iš tikrųjų dinamiškai išplečia `Frappuccino::Source` klasę, kuri turi `emit` metodą. Taipogi `Button` klasė paverčiama tema. `emit` metodas iškviečia `notify_observers(value)` metodą ir nauja reikšmė perduodama visiems stebėtojams.

Reaktyūs operatoriai (tokie kaip `map`, `merge` ir kiti) yra realizuoti grąžinant naują `Frappuccino::Stream` objektą, kuris taip pat yra stebėtojas.

Šiame pavyzdyje yra sukuriamas mygtukas. Kiekvieną kartą, kai jis paspaudžiamas, visiems temos stebėtojams yra perduodamas pranešimas apie naują reikšmę. `counter` kintamasis parodo, kiek kartų buvo paspaustas mygtukas. Nereikia kaskart iš naujo pritaikyti skaičiavimų, biblioteka leidžia apsirašyti elgseną vieną kartą deklaratyviai. `map` reaktyvus operatorius transformuoja įvykio reikšmę - jeigu gautas simbolis `:pushed` - reikšmė tampa 1, o priešingu atveju 0. Kitas operatorius `inject` priima parametrą - pradinę reikšmę (šiuo atveju 0). Kiekvieną kartą, kai yra gaunama reikšmė, perduodamas kodo blokas (`lambda`) yra iškviečiamas ir rezultatas atnaujinamas.

Rezultatas yra saugomas atmintyje, todėl programai baigus darbą, reikšmės yra prarandamos. Pranešimo metu tarptautinėje konferencijoje "Euruko" Steve Klabnik pasakojo, kad tokią biblioteką būtų galima naudoti vartotojo sąsajos programose. Jis teigia, jog pati idėja yra pasiskolinta iš funkcinių programavimo kalbų, ir įdomu pritaikyti naujus principus Ruby programavimo kalboje.

[22] https://github.com/steveklabnik

Kadangi reaktyvūs operatoriai grąžina naują stebėtoją - galima programuoti labai deklaratyviai:

```
merged_stream = Frappuccino::Stream.merge(one_stream , other_stream)
filtered_stream = merged_stream.select{ |event| event == :pushed }

filtered_stream.on_value do |event|
  # event will only ever be :pushed
end
```

Biblioteka leidžia tiek apjungti, tiek filtruoti, tiek perduoti bloką kodo, kuris bus įvykdomas, kai tik srautas gaus naują informaciją.

4.3. Įvykių kaupimas Ruby kalboje

Įvykių kaupimo principą taikančių bibliotekų Ruby kalboje nėra daug. Galima būtų išskirti 3 žinomiausias:

- "RailsEventStore"[23] - 212 žvaigždučių bei 18 išsišakojimų, paskutinis atnaujinimas įvyko 2016 metų gruodį. (Žiūrėta 2017-01-07)
- "Sandthorn"[24] - 110 žvaigždučių bei 4 išsišakojimai, paskutinis atnaujinimas įvyko 2016 metų balandį. (Žiūrėta 2017-01-07)
- "Event Sourced Record"[25] - 31 žvaigždutė bei 5 išsišakojimai, paskutinis atnaujinimas įvyko 2015 metų balandį. (Žiūrėta 2017-01-07)

Visos bibliotekos turi MIT licenziją. Verta paminėti, jog "RailsEventStore" anksčiau turėjo labiau apribojančią LGPLv3[26] licenziją (iki 2016 metų birželio).

Kadangi "RailsEventStore" yra aktyviai atnaujinima, populiariausia iš trijų, atviro kodo, įvykių kaupimą įgyvendinanti biblioteka, ją ir nagrinėsime.

4.3.1. RailsEventStore

Įvykių kaupimo biblioteka "RailsEventStore" naudoja atskirą biblioteką darbui su įvykiais pavadinimu "rails_event_store_active_record"[27]. Pagal nutylėjimą ji yra numatytoji, tačiau esant reikalui lengvai pakeičiama (pavyzdžiui, norint naudoti Greg Young "GetEventStore"[28] duomenų saugyklą). Taip pat yra galimybė naudoti duomenų saugyklą atmintyje.

[23] https://github.com/arkency/rails_event_store
[24] https://github.com/Sandthorn/sandthorn
[25] https://github.com/fhwang/event_sourced_record
[26] https://github.com/arkency/rails_event_store/commit/212b202f227a98f131dc3e8711e431e1f126b475
[27] https://github.com/arkency/rails_event_store_active_record
[28] https://geteventstore.com/

4.3.2. Naudojimas

Kai į "Ruby on Rails" projekto `Gemfile` failą jau įtrauktas `rails_event_store` modulis, reikia sugeneruoti aktyvaus įrašo migraciją:

```
rails generate rails_event_store:migrate
rake db:migrate
```

Sukurta migracija pavaizduota 4 kodo pavyzdyje (priedas nr. 2. Matome, jog kiekvienas įvykis turi srauto pavadinimą, įvykio tipą, unikalų identifikatorių, tam tikrus meta duomenis (čia gali būti tokia informacija kaip kliento IP, šalis iš kurios buvo atlikta užklausa ir t.t.), duomenų laukas, kuris aktyvaus įrašo pagalba serializuojamas į maišos duomenų struktūrą (angl. hash) bei sukūrimo laiką. Taipogi sukuriami reikalingi indeksai, kurie reikalingi greitesnėms užklausoms filtruojant pagal indeksuojamus lentelės stulpelius.

Norint naudoti bibliotekos funkcionalumą, tereikia sukurti `RailsEventStore::Client` klasės egzempliorių:

```
client = RailsEventStore::Client.new
```

4.3.3. Įvykių kūrimas

Kurti įvykius naudojantis šia biblioteka yra tikrai paprasta. Tereikia apibrėžti įvykio modelį išplečiant `RailsEventStore::Event` klasę:

```
class ProductAdded < RailsEventStore:: Event
end
```

Dabar galima sukurti įvykio egzempliorių ir išsaugoti į duomenų bazę:

```
stream_name = "product_1"
event_data = {data: { name: "Product" }}
event = ProductAdded.new(event_data)
# publishing event for specific stream
client.publish_event(event, stream_name)
# publishing global event with stream_name == 'all'
client.publish_event(event)
```

Biblioteka suteikia galimybę kurti ne tik specifinius įvykius, bet ir globalius. `event_id` yra neprivaloma reikšmė. Jeigu ji neperduodama, biblioteka pati sugeneruoja unikalų identifikatorių.

"RailsEventStore" biblioteka taip pat turi optimistinę lygiagretumo kontrolę. Galima iš anksto apibrėžti laukiamą srauto versiją bekuriant įvykį. Šiuo atveju tai paskutinio įvykio reikšmė:

```
stream_name = "product_1"
  event_data = {
      data: { name: "Product" },
      event_id: "b2e526fd-609d-4ds7-b2df-c624575c8edd"
  }
event = ProductAdded.new(event_data)
  expected_version = "951d352a-a53v-42a1-a5zz-as24chuf5b2l"
client.publish_event(event, stream_name, expected_version)
```

4.3.4. Įvykių skaitymas

Biblioteka pateikia kelis būdus skaityti įvykius iš duomenų bazės. Visais atvejais įvykiai yra išrūšiuoti didėjančia tvarka.

Galime skaityti tam tikrą skaičių įvykių pradedant tam tikru įvykiu:

```
stream_name = "product_1"
start_event = "b2d506fd-409d-4ec7-b02f-c6d2295c7edd"
count = 40
client.read_all_events(stream_name, start_event, count)
```

Galime skaityti visus konkretaus srauto įvykius:

```
stream_name = "product_1"
client.read_all_events(stream_name)
```

Galime skaityti apskritai visus įvykius

```
client.read_all_streams
```

4.3.5. Įvykių trynimas

Įvykių kaupimo principas teigia, jog įvykių žurnalas yra visa sistemos istorija ir įvykiai netrinami. Tačiau gali atsirasti tokia situacija, jog programuotojas dirba lokaliai ir tiesiog testuojasi sistemą ir bando funkcionalumą, nenori užteršti istorijos. Dėl šios priežasties biblioteka palieka galimybę trinti įvykius. Tačiau reiktų gerai pagalvoti prieš naudojant šią komandą.

```
stream_name = "product_1"
client.delete_stream(stream_name)
```

4.3.6. Prenumeratų mechanizmas

Biblioteka leidžia sinchroniškai klausyti specifinių įvykių. Prenumeratorių mechanizmo panaudojimas demonstruojamas 5 kodo pavyzdyje (priedas nr. 2). Vienintelis reikalavimas yra, jog

prenumeratoriaus klasė turi įgyvendinti `call(event)` metodą.

Šis funkcionalumas yra įgyvendintas naudojant reaktoriaus projektavimo šabloną (angl. Reactor pattern). Verta pastebėti, jog įvykių srautas gali būti begalinis, apibrėžiama, kas nutikus įvykiui yra įvykdoma, tai yra atnaujima dabartinė prekių krepšelio reikšmė. Prenumeruojami iškart keli įvykių tipai, tai yra 2 skirtingi srautai. Galime šį veikimą laikyti kaip reaktyvaus operatoriaus `merge` realizaciją.

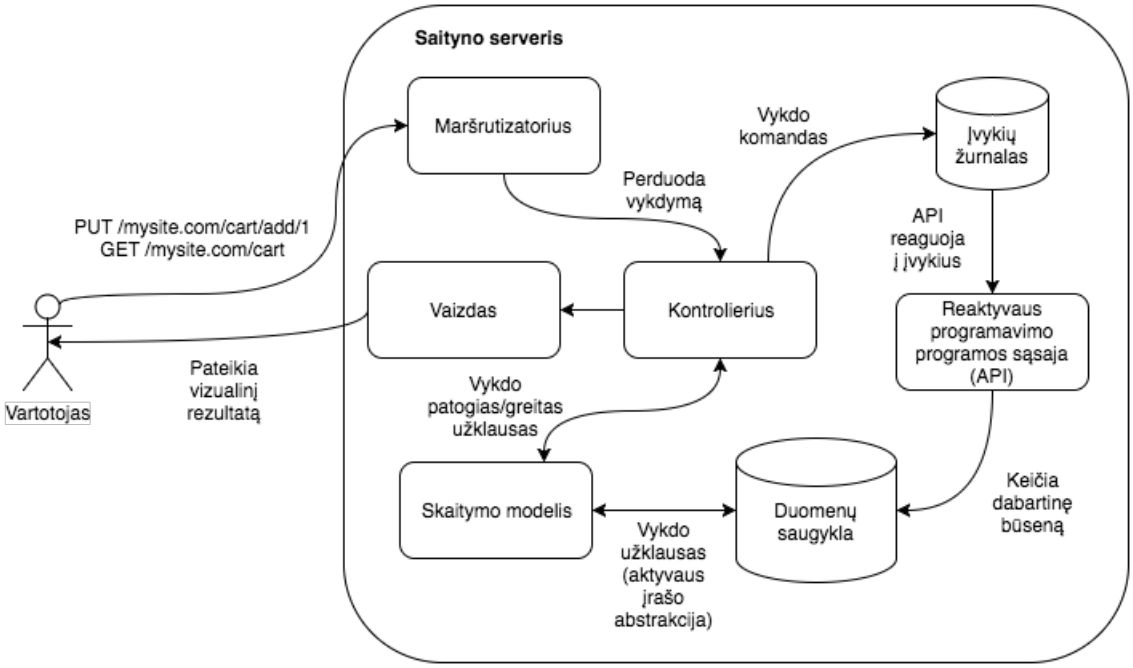
4.4. Įprastas būdas kurti įvykių kaupimo sistemas IT industrijoje

Kad būtų lengviau suprasti įvykių kaupimo sistemas bei jos komponentus, šiame poskyryje sukursime pavyzdinę įvykių kaupimo sistemos architektūros modelį, apjungiantį modelio-kontrolieriaus-vaizdo projektavimo šabloną, komandų-užklausų atskyrimo principą bei įvykių kaupimą. Šiame poskyryje bus demonstruojama kaip įprasta kurti tokias sistemas informacinių technologijų industrijoje bei bus gilinamąsi į kiekvieną komponentą žemesniame, kodo lygmenyje.

4.4.1. Pavyzdinė įvykių kaupimo sistemos architektūra

Tarkime turime tipinę saityno serverio sistemą, paremtą modelio-vaizdo-kontrolieriaus (MVC) projektavimo šablonu, kurį dar 9 dešimtmetyje aprašė Glenn Krasner ir Stephen Pope kaip bendrinę sąvoką [KP88].

5 paveikslėlyje pavaizduotas architektūros modelis naudojant MVC projektavimo šabloną bei apjungiantis komandų-užklausų atskyrimo , įvykių kaupimo principus bei reaktyvųjį programavimą. Toliau ši diagrama bus išsamiau paaiškinta bei aprašyta pagrindiniai proceso tėkmės atvejai, komandų operacijos (keičiančios būseną) bei užklausų operacijos (nekeičiančios būsenos).



5 pav. MVC paremtas architektūros modelis, apjungiantis CQRS, ES bei reaktyvaus programavimo principus

Pavyzdinis procesas - vartotojas įsideda daiktą į pirkinių krepšelį

1. Vartotojas pateikia užklausą programai įdėti pirkinį į pirkinių krepšelį. Pavyzdinė užklausos semantika - `https://mysite.com/cart/add/1`.

2. Saityno serveris gauna užklausą. Maršrutizatorius naudoja maršrutų failą `config/routes.rb` ir identifikuoja kur reikia nusiųsti užklausą. Mūsų atveju `cart` yra kontrolierius, `add` yra norimas atlikti veiksmas, o `1` yra parametras, nurodantis konkretų/unikalų pirkinį.

3. Kontrolierius iškviečia komandą.

4. Komanda gali patikrinti ar užklausa buvo teisinga. Jei užklausa buvo teisinga, kontrolierius įrašo sėkmingą arba nesėkmingą įvykį į įvykių kaupimo žurnalą. Taipogi komanda gali atlikti tam tikrus šalutinius efekus, tokius kaip elektronių žinučių siuntimas vartotojui.

5. Kai įvykis įrašomas į įvykių kaupimo žurnalą, reaktyvaus programavimo programos sąsaja (API) reaguoja į įvykį pagal aprašytas taisykles. Taisyklės aprašo ką su tuo įvykiu reikia padaryti, bendru atveju tai dažniausiai yra dabartinės būsenos atnaujinimas tam tikroje kitoje duomenų saugykloje.

6. Kai tik įvykis jau įrašytas į įvykių žurnalą, galima vartotojui pranešti apie sėkmingą rezultatą.

Pavyzdinis procesas - vartotojas peržiūri savo pirkinių krepšelį

1. Vartotojas pateikia užklausą programai peržiūrėti savo pirkinių krepšelį. Pavyzdinė užklausos semantika - `https://mysite.com/cart`.

2. Saityno serveris gauna užklausą. Maršrutizatorius naudojo maršrutų failą `config/routes.rb` ir identifikuoka kuris kontrolierius atsakys į užklausą.

3. Kontrolierius iškviečia skaitymo modelį, kuris naudojant Ruby on Rails karkasą įprastai būna supaprastintas (naudojant aktyvų įrašą nereikia rašyti SQL, jis yra sugeneruojamas). Pavyzdinė užklausa `@cart =current_user.cart.all`.

4. Dabartinis vartotojo pirkinių krepšelis yra nuskaitomas iš duomenų saugyklos. Čia reikia prisiminti galutinės darnos sąvybę. Esant didelei apkrovai, egzistuoja galimybė, jog bus nuskaitoma krepšelio būsena, kai pirkinys dar nepridėtas į krepšelį. Todėl toks sistemos projektavimas verčia labiau mąstyti apie pačią sistemą. Įprastiniu atveju, pirkinių krepšelis įvykdžius daikto pridėjimo scenarijų, aprašytą aukščiau, galėtų būti pridedamas vartotojo sąsajoje. Vartotojas matytų, jog pirkinys pridėtas į krepšelį, tačiau įvykių kaupimo programavimo programos sąsaja (API) dar galbūt nebūtų baigusi darbo. Kai vartotojas norėtų pirkti visus krepšelio pirkinius, reaktyviojo programavimo programos sąsaja (API) jau būtų baigusi darbą. Įprastiniu atveju tai trunka keliasdešimt milisekundžių.

5. Galiausiai vartotojui pateikiamas krepšelio vaizdas.

4.4.2. Domeno modelis

Remiantis literatūros analizėje aptartu domenu pagrįstu projektavimu - pradedant kurti įvykių kaupimo sistemą, pirmiausia reikia apsibrėžti domeno modelį, arba tiksliau agregatus, kuriuos aprašo Martin Fowler [Fow12]. Naudojant įvykių kaupimą, domeno modelis yra sukuriamas remiantis domeno įvykiais. Duomenų struktūra nesaugo dabartinės būsenos, o saugo eilę domeno įvykių, kurie buvo pritaikyti agregatui nuo sistemos gyvavimo pradžios. Tai leidžia konstruoti agregatą, neatskleidžiant jo būsenos ir taip apsaugant jo invariantus.

Agregatas gali būti naujų domeno įvykių šaltinis - kiekvienas agregato metodo kvietimas gali publikuoti naują domeno įvykį. Kiekvienas agregato vidinės būsenos pasikeitimas privalo būti realizuotas publikuojant arba pritaikant domeno įvykį. Tik taip galima užtikrinti, jog taikant įvykius agregatas visada bus atkurtas į tą pačią būseną.

Pirmiausia apsibrėžkime domeno įvykius:

```
module Events
  ProductAddedToCart = Class.new(RailsEventStore::Event)
  ProductRemovedFromCart = Class.new(RailsEventStore::Event)
  OrderCreated = Class.new(RailsEventStore::Event)
end
```

Pagal nagrinėjamą scenarijų minėti įvykiai apibrėžia šiuos veiksmus:

- `ProductAddedToCart` - pirkinys pridėtas į krepšelį.
- `ProductRemovedFromCart` - pirkinys išimtas iš krepšelio.
- `OrderCreated` - įvykdytas užsakymas.

Kai turime apsibrėžtus domeno įvykius, galime juos pritaikyti domeno objektui. Panagrinėkime 6 kodo pavyzdį (priedas nr. 2). `Domain::Order` klasė turi metodą `initialize`, kuris sukuria pradinę agregato būseną. Taip pat, klasė turi `create` metodą. Pastarasis turėtų būti naudojamas kitų objektų, iškviečiant agregato funkcijas. Čia turėtų būti apsaugomi invariantai, tikrinama verslo logika, teisingumo taisyklės. Remiantis CQRS principu, šis metodas neturėtų grąžinti reikšmės - jis privalo būti tik įvykdomas arba nutraukti vykdymą iškeliant išimtinę situaciją. Taipogi šiame metode agregato būsena niekada nėra keičiama. Vietoje to yra sukonstruojamas ir pritaikomas naujas domeno įvykis (kurį apsirašėme anksčiau).

Reiktų detaliau panagrinėti iš kur įtraukiamas `AggregateRoot::Base` modulis. Tai yra "aggregate_root"[29] biblioteka, kuri atkeliauja kartu su "RailsEventStore biblioteka". 7 kodo pavyzdyje (priedas nr. 2) demonstruojamas svarbiausias funkcionalumas. Modulis įterpia metodą `apply`. Kviečiant `apply Events::OrderCreated.create(@id, order_number, customer_id)` yra sukuriamas naujas domeno įvykis ir pritaikomas agregatui - būsena pasikeičia ir `@unpublished_events` kintamasis saugo pakeitimus. Jis saugo visus agregato sukurtus domeno įvykius, kol metodas yra vykdomas.

[29] https://github.com/arkency/aggregate_root

Paskutinis svarbus momentas yra `apply_order_created` metodas. Pastarasis atnaujina agregato būseną priklausomai nuo įvykusio domeno įvykio. Nesvarbu ar pats agregatas publikavo įvykį, ar įvykis buvo nuskaitytas iš įvykių žurnalo ir pritaikytas atkuriant dabartinę agregato būseną, šioje vietoje negali būti verslo logikos ar teisingumo taisyklių. Tai tik būsenos atkūrimas, o istorijos keisti negalima, kaip buvo nagrinėta literatūroje.

4.4.3. Komandos

Komanda yra paprastas objektas, kuris apibrėžia parametrus veiksmo vykdymui. Remiantis domenu paremtu projektavimu, komanda turėtų būti pavadinta bendrąja verslo kalba (skvarbiąja kalba) ir išreikšti vartotojo ketinimą. Prieš komandos vykdymą turėtų būti užtikrintos teisingumo taisyklės. Teisingumo taisyklės turėtų būti paprastos, pagrįstos pačiais duomenimis, bet ne verslo logika (Kaip jau minėjome praeitame skirsnyje, domeno objektas tikrina verslo logiką).

Pažvelkime į komandos pavyzdį:

```
module Command
  class CreateOrder < Base
    attr_accessor :order_id
    attr_accessor :customer_id

    validates :order_id, presence: true
    validates :customer_id, presence: true

    alias :aggregate_id :order_id
  end
end
```

"Ruby" programavimo kalboje tai dar žinoma kaip formos objektai (angl. Form Objects)[30]. Iš esmės, norint leisti patikrinti teisingumo taisykles, tėvinė klasė `Base` naudoja dalį `ActiveModel` funkcijų. Išeities kodas demonstruojamas 8 kodo pavyzdyje (priedas nr. 2).

4.4.4. Komandų apdorojimas

Komandų doroklė yra pradinis domeno taškas, turintis instrumentuoti domeno objektus ir domeno tarnybas bei vykdyti domeno objektų metodus. Komandų doroklės išeities kodas yra pateiktas 9 kodo pavyzdyje (priedas nr. 2). Metodas `build` ištraukia įvykius iš įvykių saugyklos, naudodamasis agregato unikaliu identifikatoriumi ir paduodamas įvykių srauto vardą. Komandų doroklė iš pradžių atkuria agregato būseną ir tada išsaugo pakeitimus įvykių žurnale. Prisiminkime praeitame skirsnyje aprašytą `unpublished_events` kintamąjį. Štai čia atsiskleidžia tikroji jo paskirtis, visi dar nepublikuoti įvykiai yra išsaugomi. Kai įvykis yra išsaugomas, visi prenumeratoriai, kurie stebi ši įvykį (arba temą) yra informuojami apie pasikeitimus.

Taigi išplečiant šį atvejį, galime pavaizduoti užsakymo komandų doroklę:

[30] https://webuild.envato.com/blog/creating-form-objects-with-activemodel-and-virtus/

```
module CommandHandlers
  class CreateOrder < Command::Handler
    def call(command)
      with_aggregate(command.aggregate_id) do |order|
        order.create(command.customer_id)
      end
    end

    private

    def aggregate_class
      Domain::Order
    end
  end
end
```

Čia `with_aggregate(command.aggregate_id)` metodas grąžina domeno objektą `Domain::Order`, atkurtą iš anksčiau sistemoje įvykusių įvykių, kurie skaitomi iš įvykių žurnalo. Tada komandų doroklė iškviečia `create` metodą perduodama parametrus, kuriuos gavo iš komandos.

Supratus domeno įvykius, komandas bei komandų doroklę iškyla klausimas: o kaip kontrolierius iškviečia komandą?

Paprastesniam naudojimui galima sukurti atskirą modulį, kurį galima labai paprastai įtraukti į `ApplicationController`, jame prirašius `include Command::Execute`. `Command::Execute` modulis pavaizduotas 10 kodo pavyzdyje (priedas nr. 2). Šiuo atveju iš pradžių yra tikrinamos komandos teisingumo taisyklės. Sukuriamas objektas prieigai prie įvykių žurnalo skaitymo. `handler_for` metodas tiesiog aprašo taisykles kuriomis komandos yra priskiriamos jų doroklėms.

Galiausiai iš kontrolieriaus galime sukurti užsakymą:

```
def create
  cmd = Command::CreateOrder.new(order_params)
  execute(cmd)

  redirect_to Order.find_by_uid(cmd.order_id), notice: 'Order was successfully
      created.'
end
```

4.4.5. Skaitymo modelio kūrimas

Iki šiol aprašėme kaip vartotojas vykdo komandas, kaip jos apdorojamos, kur įvykdoma domeno logika, kaip publikuojami ir saugomi domeno įvykiai. Prisiminkime, jog domeno objektai neleidžia paimti (arba nerodo) dabartinės būsenos, todėl reikia kito kelio konstruoti duomenis pa-

teikimui vartotojui (vaizde), tai yra skaitymo pusei. Čia verta pažymėti, jog duomenys yra denormalizuoti, jie yra būtent tokie, kokių reikia tam vaizdui paruošti. Šis sprendimas turi didelį našumo privalumą, kai reikalingos labai greitos užklausos, nes nereikia vykdyti sudėtingų užklausų (pvz.: sujungimo). Duomenų saugykla šiuo atveju gali būti bet kokia duomenų bazė tiek reliacinė, tiek grafų ar NoSQL.

Kad galėtume naudotis šiomis greitomis užklausomis, pirmiausia reikia paruošti skaitymo pusės duomenis. Reikia apdoroti įrašytus įvykius ir juos transformuoti į norimą vaizdą.

Taigi galime apsirašyti įvykių kaupimo sistemos maršrutizatorių:

```
module EventStoreSetup
  def event_store
    @event_store ||= RailsEventStore::Client.new.tap do |es|
      es.subscribe(Denormalizers::OrderCreated.new, [Events::OrderCreated])
      es.subscribe(Denormalizers::ProductAddedToCart.new,
          [Events::ProductAddedToCart])
      es.subscribe(Denormalizers::ProductRemovedFromCart.new,
          [Events::ProductRemovedFromCart])
    end
  end
end
```

Įvykių maršrutizatorius aprašo koks duomenų denormalizatorius apdoroja domeno įvykį arba aibę domeno įvykių. Iš tikrųjų tai ta pati pagrindinė paskirtis kaip ir maršrutizatoriaus MVC projektavimo šablone, kuris pasako kokią užklausą kuris kontrolierius apdoros.

Pažiūrėkime į užsakymo denormalizatoriaus kodą:

```
module Denormalizers
  class OrderCreated
    def call(event)
      return if Order.where(uid: event.data.order_id).exists?
      order = ::Order.new.tap do |o|
        o.uid = event.data.order_id
        o.number = event.data.order_number
        o.customer = Customer.find(event.data.customer_id).name
        o.state = "Created"
      end
      order.save!
    end
  end
end
```

Pagrindinė jo paskirtis išsaugoti gautus duomenis į tam tikrus skaitymo duomenų saugyklos stulpelius (ar įrašus - priklausomai nuo duomenų saugyklos tipo). Tačiau atidžiau pažvelkime į šią eilutę - `return if Order.where(uid: event.data.order_id).exists?`. Jeigu įvykis jau eg-

zistuoja - jis nėra atnaujinamas. Tai jau tam tikra teisingumo taisyklė, kuri yra patikrinama prieš įrašant duomenis į duomenų saugyklą. Taigi denormalizatorius šiuo atveju turi dvi atsakomybes - filtruoti duomenis ir išsaugoti duomenis į pasirinktą duomenų saugyklą. Kaip pamatysime vėliau darbe, kuriant biblioteką, apjungiančią reaktyvaus programavimo ir įvykių kaupimo principus, skaitymo modelio kūrimas gali būti supaprastintas.

4.4.6. Skaitymo pusės asinchroninis kūrimas

Iš tiesų "RailsEventStore" biblioteka veikia sinchroniškai, tačiau skaitymo pusę generuoti galima ir asinchroniškai pasitelkiant papildomas bibliotekas tokias kaip "Sidekiq"[31] ar "Rescue"[32]. Šios bibliotekos pasižymi bendru bruožu - naudoja gijas apdoroti daugybę darbų tuo pačiu metu. Parametrai perduodami šiems darbams dažniausiai saugomi duomenų saugyklose atmintyje, tokiose kaip "Redis"[33] ar "Memcached"[34] našumui užtikrinti.

Parodysime kodo pavyzdį kaip atrodytų asinchroninis užsakymo kūrimas panaudojant "Sidekiq":

```
module Workers
  class OrderExists
    include Sidekiq::Worker

    def perform(order_id)
      Order.where(uid: order_id).exists?
    end
end

module Filters
  class OrderExists
    def self.execute(event)
      Workers::OrderExists.perform_async(event.data.order_id)
    end
  end
end
```

Čia matome, jog darbininko klasėje tereikia pridėti `include Sidekiq::Worker` modulį ir įgyvendinti `perform` metodą. Įvykdžius `Workers::OrderExists.perform_async(event.data.order_id)`, parametrai (unikalus darbo identifikatorius, perduoti parametrai) yra išsaugomi atskiroje duomenų saugykloje (skirtoje būtent tam) į eilę. "Sidekiq" procesas tikrina atsiradusius naujus darbus šioje duomenų saugykloje, ir bendru atveju ima juos iš eilės ir priskiria laisvai gijai, kuri apdoroja rezultatą. Esant didesniai

[31] https://github.com/mperham/sidekiq
[32] https://github.com/resque/resque
[33] https://redis.io/
[34] https://memcached.org/

sistemos apkrovai yra ganėtinai nesunku pridėti dar vieną serverį, apdorojantį tokius asinchroninius darbus.

Toks asinchroninis darbų apdorojimas padidina sistemos našumą, padaro ją labiau išplečiama, tačiau kaip buvo minėta literatūroje, priverčia atidžiau mąstyti apie pačią sistemą dėl galutinės darnos principo.

Toks būdas gali būti naudingas darbams, kurie užtrunka ilgiau. Pavyzdžiui, sistemoje yra galimybė įkelti prekes ir jų paveikslėlius. Paveikslėlius reikia sumažinti, sukarpyti, suprastinti kokybę ar dar kitaip apdoroti. Toks procesas trunka ilgiau nei paprastas įrašymas į duomenų bazę, todėl iškėlimas būtų labai naudingas didinant sistemos našumą.

4.5. Reaktyvus programavimas bei įvykių kaupimas kartu

Šiame poskyryje bus siekiama pasiekti darbo tikslą - pritaikyti reaktyvaus programavimo principus įvykių kaupimo sistemose taip, jog skaitymo modelis būtų kuriamas tik komponavimo būdu, neturėtų būsenos, tai yra visos operacijos su duomenų baze būtų paslėptos. Pirmiausia bus aprašyta problema, su kuria susiduriama kuriant įvykių kaupimo sistemos skaitymo modelį naudojantis "RailsEventStore" biblioteka. Vėliau bus pristatyti kuriamos patobulintos bibliotekos reaktyvūs operatoriai bei jų panaudojimo atvejai, pagrindinės realizacijos problemos bei jų sprendimai, suformuluoti apribojimai.

4.5.1. Įprasto būdo kurti įvykių kaupimo sistemas problema

Praeitame poskyryje buvo pademonstruotas įvykių kaupimo sistemos skaitymo modelio kūrimas panaudojant egzistuojančią biblioteką. Tačiau toks bibliotekos panaudojimas nėra deklaratyvus, programuotojui tenka rašyti darbo su duomenų baze operacijas. Norima tokį pašalinį efektą paslėpti, jog būtų galima koncentruotis tik ties sprendžiama problema.

Remiantis G. Salvaneschi empiriniu tyrimu apie programos, paremtos reaktyviu programavimu, suprantamumą [SAPM14], tai galėtų sumažinti programuotojo daromų klaidų skaičių bei pagerinti programos suprantamumą.

4.5.2. Pasiruošimas kurti biblioteką

Norint išspręsti minėtą problemą ir sukurti patogią naudotis biblioteką, pirmiausia reikia atsakyti į šiuos klausimus:

- Kaip įvykiai bus saugojami ir publikuojami?
- Kaip atrodys skaitymo modelio kvietimas?
- Kaip atrodys skaitymo modelio kūrimo aprašas?
- Kokie reaktyvūs operatoriai bus realizuoti?
- Kurie reaktyvūs operatoriai turi paslėpti operacijas su duomenų saugykla?

Sekančiuose skyriuose atsakysime į šiuos klausimus.

4.5.3. Įvykių saugojimas ir publikavimas

Įvykių saugojimui pasirenkamas aktyvaus įrašo projektavimo šablonas, kuris buvo nagrinėtas literatūros analizėje. Įvykių publikavimui galima panaudoti "rails_event_store_active_record"[35] biblioteką. Ją pagal nutylėjimą naudoja "RailsEventStore" biblioteka.

Tarkime turime domeno sritį bankininkystė ir vartotojas gali atlikti šias operacijas, kurias atvaizduotų atitinkami įvykiai sistemoje:

- `AccountCreated` - susikurti sąskaitą, kuri turėtų unikalų sąskaitos identifikatorių ir einamąjį balansą.
- `MoneyDeposited` - įnešti pinigus į sąskaitą.
- `MoneyWithdrawn` - išsiimti pinigus.

Įvykio publikavimo pavyzdys:

```
stream_name = "account"
event = AccountCreated.new(data: {
         account_id: ``LT121000011101001000''
       })
EventStore::EventRepository.new.create(event, stream_name)
```

4.5.4. Skaitymo modelio kvietimas

Patogiam užklausų rašymui, pasirenkamas aktyvaus įrašo projektavimo šablonas, kuris buvo nagrinėtas literatūros analizėje.

Norint surasti sąskaitą `Account` pagal unikalų identifikatorių užtenka iškviesti:

```
  Account.find_by(account_id: 'LT121000011101001000')
```

Norint atvaizduoti sąskaitas, kurios buvo atidarytos paskutinį mėnesį ir išrūšiuoti pagal naujumą užtenka iškviesti:

```
  Account.filter(created_at: 1.month.ago..Time.current).order(created_at: :desc)
```

4.5.5. Skaitymo modelio kūrimo aprašas

Tarkime, jog norime sukurti skaitymo modelį, kuris atvaizduotų dabartinį vartotojo sąskaitos balansą. Skaitymo modelio kūrimo kodas turėtų atrodyti taip:

```
account_view =
  Stream.new(AccountCreated, MoneyDeposited, MoneyWithdrawn).
    as_persistent_type(Account).
```

[35] https://github.com/arkency/rails_event_store_active_record

```
init( -> (state) { state.balance = 0} ).
when(MoneyDeposited, -> (state, event) { state.balance +=
    event[:data][:amount] }).
when(MoneyWithdrawn, -> (state, event) { state.balance -=
    event[:data][:amount] })
```

Verta pažymėti, jog čia nėra nė vienos operacijos su duomenų saugykla. Jeigu būtų sukurta vartotojo sąskaita su kodu "LT121000011101001000", dabartinį sąskaitos balansą galėtume gauti tiesiog iškvietę `Account.find_by(account_id: 'LT121000011101001000').balance`

4.5.6. Reaktyvūs operatoriai

Kuriant biblioteką pasirinkta įgyvendinti šiuos operatorius:

- `merge(another_stream)` - srautų sujungimo operatorius. Šis operatorius sujungia 2 srautus į vieną.
- `filter(predicate_function)` - filtravimo operatorius. Operatorius tikisi funkcijos, kuri pritaiko įvykį šiai funkcijai ir tikisi `boolean` tipo atsakymo.
- `map(transform_function)` - transformavimo operatorius. Operatorius tikisi funkcijos, kuri transformuoja įvykį.
- `init(initial_state_function)` - pradinės reikšmės operatorius. Operatorius išsaugo pradinę skaitymo modelio būseną duomenų saugykloje, jei ji dar nesukurta.
- `when(event_type, state_change_function)` - tipo atitikimo operatorius. Nutikus tam tikram tipo `event_type` įvykiui, duomenų saugykloje išsaugoma tarpinė skaitymo modelio būsena pritaikant būsenos pakeitimo funkciją.
- `each(state_change_function)` - iteratoriaus operatorius. Funkcionalumas identiškas tipo atitikimo operatoriui `when`, tačiau būsenos keitimo funkcija pritaikoma bet kokiam įvykiui.

Šie operatoriai bus plačiau paaiškinti ir pademonstruoti aprašant realizacijos detales.

4.5.7. Pagrindinės realizacijos problemos

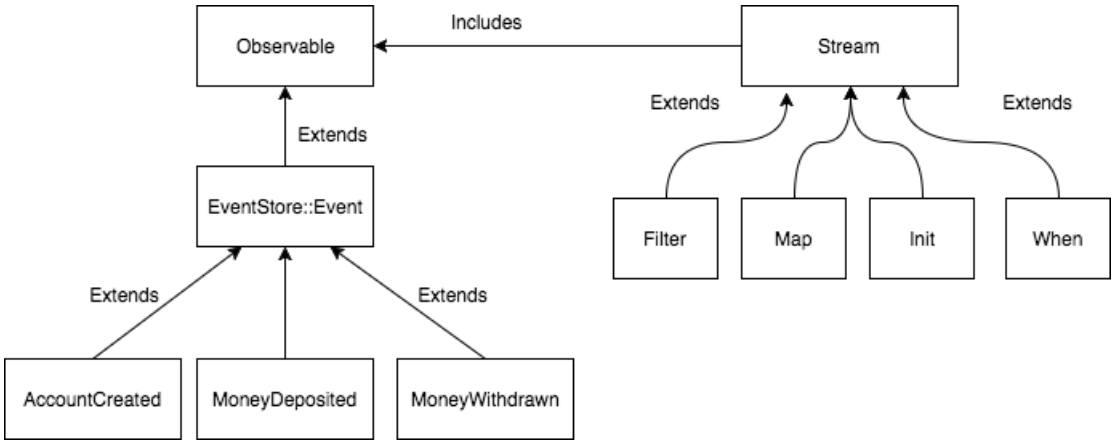
Apsibrėžus kuriamos bibliotekos norimą pasiekti funkcionalumą bei naudojimosi sintaksę, kyla daugiau klausimų:

- Kaip stebėti įvykius sistemoje?
- Kaip realizuoti srautą bei reaktyvius operatorius?
- Kaip išsaugoti tarpinę skaitymo modelio būseną paslepiant operacijų su duomenų saugykla detales?

Sekančiame skyriuje aptarsime bibliotekos realizacijos detales bei kertinius sprendimus, atsakančius į išsikeltas problemas.

4.5.8. Realizacijos detalės

Analizuojant reaktyvaus programavimo bibliotekas, buvo pastebėta, jog jos naudoja stebėtojo projektavimo šabloną. Remiantis analizės rezultatais apsibrėšime klases, kurių reikia funkcionalumui realizuoti. Kuriamos bibliotekos klasių priklausomybių diagrama pavaizduota 6 paveikslėlyje.



6 pav. Kuriamos bibliotekos klasių priklausomybių diagrama

Įvykių stebėsenos realizacija

`EventStore::Event` išplečia `Observable` modulį ir šiuo atveju yra tema. Kiekvienas įvykio tipas sistemoje yra kuriamas paveldint `EventStore::Event` klasę:

```
class AccountCreated < EventStore::Event
end
# arba
AccountCreated = Class.new(EventStore::Event)
```

Kadangi tema yra klasė, o ne objektas (tokiu atveju programuotojui pačiam tektų informuoti stebėtojus), verta atskirai panagrinėti `EventStore::Event` klasės objekto metodą `emit`:

```
def emit
  self.class.changed
  self.class.notify_observers(self.to_h)
end
```

Išplečiant klasę `Observable` moduliu, pastarosios metodai yra pridedami klasei, o ne tos klasės objektui. Čia galima panaudoti introspekciją ir sužinoti kokio tipo objektui metodas yra kviečiamas. Taip informuojami visi stebėtojai, kurie prenumeruoja šią temą. Šis metodas yra iškviečiamas iškart po to kai įvykis būna įrašytas į duomenų saugyklą naudojantis anksčiau minėtu įvykių publikavimo mechanizmu. Galime pažvelgti į klasės `EventRepository`, atsakingos už įvykių publikavimą, metodą `create`:

```
def create(event, stream_name)
  data = event.to_h.merge!(stream: stream_name)
  adapter.create(data)
```

```
  # Notify observers of new event
  event.emit if event.respond_to?(:emit)

  event
end
```

Kai tik įvykis išsaugomas duomenų saugykloje, visi įvykio prenumeratoriai yra informuojami apie naują įvykį sistemoje.

Skaitymo modelio kūrimo aprašo realizacija

Skaitymo modelis yra kuriamas pasinaudojant duomenų srautais. Inicializuojant duomenų srautą `Stream.new(*sources)`, srautas geba pats pasirūpinti, jog būtų informuotas apie naujus įvykius sistemoje. Tam pasiekti, konstruktoriuje apibrėžiami prenumeruojami įvykiai:

```
sources.each { source.add_observer(self) }
```

Kiekvienas reaktyvus operatorius yra kuriamas labai panašiai, todėl pateiksime tik porą operatorių realizaciją:

```
def filter(blk)
  Filter.new(self, blk)
end

def when(event_type, blk)
  When.new(self, event_type, blk)
end
#...
```

Grąžinama nauja atitinkama klasė, kuri paveldi `Stream` srautą. Verta pastebėti, jog perduodamas ir pradinis srautas. Reaktyvaus operatoriaus klasė inicializavimo metu iškviečia `source.add_observer(self)` ir taip prenumeruoja srautą, kuris sukūrė šį reaktyvų operatorių. Taigi srautas `Stream` tuo pačiu metu gali būti ir stebėtojas, ir tema.

Srautas `Stream` realizuoja svarbų metodą `update`, kuris yra visada iškviečiamas gavus naują įvykį. Šis metodas pažymi, jog įvyko pasikeitimai ir perduoda pakeitimus visiems stebėtojams. Reaktyvių operatorių klasės pasinaudoja objektinio programavimo sąvybe - polimorfizmu ir perrašo šį funkcionalumą priklausomai nuo operatoriaus. Šis funkcionalumas bus pademonstruotas aprašant reaktyvius operatorius.

Kuriamo skaitymo modelio tipo operatorius

Norint išsaugoti tarpinę skaitymo modelio būseną duomenų saugykloje, reikia žinoti skaitymo modelio tipą bei jo pirminį raktą.

`as_persistent_type(resource_type, unique_resource_identifier)` yra srauto metodas, kuris įsimena kuriamo skaitymo modelio tipą bei jo pirminį raktą. Naudojant reaktyvius operatorius, modelio tipas ir pirminis raktas yra automatiškai perduodami reaktyvaus operatoriaus

srautui. Bendruoju atveju, pirminis raktas gali būti nuspėtas. Tarkime, jeigu kuriamas vartotojo sąskaitos skaitymo modelis `Account`, pirminis raktas greičiausiai bus `account_id`. Kartais gali atsirasti noras kurti sudėtingesnius skaitymo modelio vaizdus. Tarkime, programos vartotojas gali turėti kelias sąskaitas. Tokius atveju pirminis raktas turėtų būti pora `user_id, account_id` - vartotojo kodas ir sąskaitos kodas. Norint sukurti tokį skaitymo modelį, užtektų kuriamo skaitymo modelio aprašo srautui iškviesti `as_persistent_type(Account, [:account_id, :user_id])`. Pirminio rakto sudedamųjų dalių tvarka šiuo atveju nesvarbi. Metodas grąžina pasinaudoja metodų apjungimo principu, tai yra grąžina objektą, kuriam buvo iškviestas.

Įprastai programuotojas turi apsirašyti aktyvaus įrašo modelį pats `class Account < ActiveRecord::Base; end`. Tačiau čia gali pasitarnauti meta programavimas [Ols], kuris leidžia programuotojams būti produktyvesniems generuojant dalį kodo. Šiuo atveju biblioteka leidžia paduoti ne tik jau aprašytą aktyvaus įrašo tipą, bet priima tiek eilutės, tiek simbolio tipą ir gali dinamiškai paveldėti aktyvaus įrašo bazinį tipą panaudojant refleksija:

```
@resource_type =
  if defined_stream_type.is_a? String
    Object.const_set(dynamic_name, Class.new(ActiveRecord::Base))
  elsif defined_stream_type.is_a? Symbol
    Object.const_set(defined_stream_type.to_s.capitalize,
        Class.new(ActiveRecord::Base))
  else
    defined_stream_type
  end
```

Filtravimo operatorius

Filtravimo operatorius `filter` priima predikatą, tai yra tam tikrą sąlygą. Jeigu ši sąlyga yra teisinga, įvykis yra perduodamas toliau visiem jo prenumeratoriams. Šis funkcionalumas realizuojamas pasinaudojant objektinio programavimo sąvybe polimorfizmu. Tėvinės klasės metodas `update` yra perrašomas ir pakeitimai perduodami tik tada kai sąlyga yra išpildyta:

```
class Filter < Stream
  def initialize(source, blk)
    @resource_type = source.resource_type
    @unique_resource_identifier = source.unique_resource_identifier
    @block = blk
    source.add_observer(self)
  end

  def update(event)
    occur(event) if @block.call(event)
  end
end
```

Galime panagrinėti šio reaktyvaus operatoriaus panaudojimo atvejį. Tarkime vartotojas sistemoje gali tiek nusipirkti prekę, tiek užsisakyti, jog prekė būtų pagaminta pagal užsakymą. Gali kilti noras konstruoti skaitymo modelį, apjungiantį juos abu bei turint papildomų sąlygų. Pavyzdžiui vartotojui gali būti suteikiami kreditai už sėkmingą pirkimą tik tada, kai:

- Pirkinio vertė yra didesnė nei 100 eurų.
- Specialaus užsakymo vertė yra daugiau nei 50 eurų.

Tokį duomenų srautą galima konstruoti panaudojant reaktyvų filtravimo operatorių `filter` kaip:

```
product_orders_eligible_for_bonus =
  Stream.new(ProductPurchased).
            filter(-> (event) event[:data][:price_paid] > 100)

job_orders_eligible_for_bonus =
  Stream.new(JobOrderPurchased).
           .filter(-> (event) event[:data][:price_paid] > 50)
```

Srautų sujungimo operatorius

Apibrėžiant filtravimo operatorių, buvo sukurti 2 srautai, kurie yra filtruojami skirtingi. Norėdami pritaikyti bendrus veiksmus jiems, turime turėti galimybę juos sujungti. Šių filtruotų srautų sujungimo operatorius `merge` atrodytų kaip:

```
merged_stream =
    product_orders_eligible_for_bonus.merge(job_orders_eligible_for_bonus)
```

Metodo `merge` realizacija yra naujo srauto sukūrimas, panaudojant 2 srautus:

```
def merge(another_stream)
  Stream.new(self, another_stream)
end
```

Pradinės reikšmės operatorius

`init` metodas veikia kaip inicializatorius. Jeigu skaitymo modelis dar nėra saugomas duomenų saugykloje, sukuriant įrašą bus nustatoma pradinės įrašo reikšmės. Šiuo atveju sąskaitos balansas bus 0. Metodas priima `lambda` funkciją, kuri bus iškviesta inicializavimo metu.

```
def update(event)
  check_resource_type_presence

  entity_id_hash = extract_entity_id(event)

  if !@resource_type.where(entity_id_hash).exists?
    resource = @resource_type.new(event[:data])
```

```
    @block.call(resource)
    resource.save!
  end

  occur(event)
end
```

Čia `update` metodas yra iškviečiamas kiekvieną kartą, kai duomenų srautas gauna naują įvykį, o stebėtojas jį prenumeruoja. `Init` tipo srautas saugo informaciją apie kuriamą skaitymo modelio tipą, todėl iš gauto įvykio gali išgauti pirminio rakto informaciją. Jeigu toks skaitymo modelis dar nėra sukurtas sistemoje, naujas įrašas yra sukonstruojamas, pritaikoma būsenos keitimo funkcija ir programos tarpinė būsena yra išsaugoma duomenų saugykloje. Įvykis visada perduodamas toliau visiems prenumeratoriams.

Tipo atitikimo operatorius

Operatorių `when(event_type, blk)` galima vadinti tipo atitikimo operatoriumi. Perduodamas `lambda` blokas bus iškviestas tik tada kai sistemoje įvykusio įvykio tipas bus toks, koks yra apibrėžtas.

Jeigu tipai atitinka ir skaitymo modelio tarpinė būsena dar nėra saugoma duomenų saugykloje, šio operatoriaus realizacija yra identiška `init` operatoriaus realizacijai.

Jeigu tipai atitinka ir skaitymo modelio tarpinė būsena jau yra saugoma duomenų saugykloje, būsena užklausiama, modifikuojama panaudojus būsenos keitimo funkciją, o gautas tarpinis rezultatas išsaugomas duomenų saugykloje:

```
resource = @resource_type.where(entity_id_hash).first
@block.call(resource)
resource.save!
```

Iteratoriaus operatorius

Kartais gali vertėti klausytis kelių įvykių šaltinių ir jiems pritaikyti bendras operacijas. Tarkime norime atnaujinti informaciją paieškos duomenų saugykloje, kai pasikeičia kokia nors svarbi informacija apie produktą. Tokį funkcionalumą, galėtume išreikšti kaip:

```
  Stream.new(ProductImageUploaded, ProductInformationChanged)
      .as_persistent_type(Product)
      .each(-> (state, event) state.reindex )
```

Iteratoriaus operatoriaus `each` realizacija yra identiška tipo atitikimo operatoriui. Vienintelis skirtumas, jog būsenos keitimo funkcija yra pritaikoma visems prenumeruojamiems įvykiams.

Skirtingi būdai struktūrizuoti skaitymo modelio aprašą

Iš esmės `lambda` blokas yra vykdomas, kai jam iškviečiamas `call` metodas. Kadangi aprašyti operatoriai priima `lambda` bloką kaip parametrą, esti papildomi būdai perduoti juos.

Galima apsirašyti kintamąjį:

```
  variable = -> (event) { }
```

Taip pat galima apsirašyti klasę, kuri turi `call` metodą:

```
class Denormalizers::ReadModelType::Event
  def call(state, event)
    # implementation
  end
end
```

Lankstumas suteikia galimybę laisvai struktūrizuoti kodą, sutrumpinti skaitymo modelio kūrimo aprašą, kas gali pagerinti projekto priežiūrą, skaitomumą bei programuotojo produktyvumą.

4.5.9. Konkretizuotas taisyklių rinkinys

Sukūrus biblioteką, galime konkretizuoti taisykles, reikalingas srauto ir reaktyvių operatorių realizacijai. Labiausiai dominantys klausimai yra:

- Kaip reaktyvūs operatoriai transformuoja srauto žinutes į tarpinę būseną?
- Kokios taisyklės transformuoja tarpinę būseną į skaitymo modelį.

Atsakymas į pirmąjį klausimą pateiktas diagramoje, kuri pavaizduota 7 paveikslėlyje. Ši diagrama parodo ryšį tarp srautų ir reaktyvių operatorių bei aprašo kokie veiksmai yra atliekami su gauta srauto žinute, tai yra tam tikru prenumeruojamu įvykiu. Iteratoriaus operatorius `each(state_change_function)` yra praleistas, nes yra labai panašus į tipo atitikimo operatorių `when(type, state_change_function)`. Vienintelis skirtumas, jog iteratoriaus operatorius netikrina tipo.

Atsakant į antrąjį klausimą, verta panagrinėti, kaip tarpinė būsena yra sukuriama, kaip jai pritaikoma būsenos keitimo funkcija ir kaip ši būsena yra išsaugoma. Čia būtina sąlyga yra kuriamo skaitymo modelio tipo bei pirminio rakto žinojimas. Toliau paaiškinsime diagramoje aprašytus žingsnius:

- **Tarpinės būsenos sukūrimas** - jei tarpinė būsena neegzistuoja duomenų saugykloje, sukuriamas objektas, turintis tokius pačius laukus kaip ir kuriamas skaitymo modelis. Skaitymo modelio laukai gaunami žinant kuriamo skaitymo modelio tipą. Sukurti objekto laukai privalo turėti tiek rašymo tiek skaitymo prieigą.
- **Tarpinės būsenos suradimas** - jei tarpinė būsena egzistuoja duomenų saugykloje, reikia surasti šią būseną. Būsena randama padarius užklausą į duomenų saugyklą panaudojant skaitymo modelio tipą bei pirminį raktą. Sukuriamas objektas, turintis tokius pačius laukus kaip ir kuriamas skaitymo modelis. Sukurti objekto laukai privalo turėti tiek rašymo tiek skaitymo prieigą. Sukurtam objektui priskiriami užklausos metu gauta informacija apie esamą tarpinę būseną.

- **Tarpinės būsenos keitimo funkcijos pritaikymas** - turint praeituose žingsniuose sukurtą objektą, kuris turi visus skaitymo modelio laukus, iškviečiama tarpinės būsenos keitimo funkcija. Sukurtas objektas yra perduodamas kaip argumentas šiai funkcijai. Funkcija modifikuoja pastarojo objekto laukus.

- **Tarpinės būsenos išsaugojimas** - praeituose žingsniuose buvo sukurtas ir modifikuotas objektas, saugantis tarpinę skaitymo modelio būseną. Kadangi visi objekto laukai atitinka skaitymo modelio duomenų saugyklos laukus, padaroma užklausa, kuri išsaugo (jei tarpinė būsena nesaugoma duomenų saugykloje) arba atnaujina (jei tarpinė būsena jau saugoma duomenų saugykloje) šią informaciją duomenų saugykloje.

Toliau aprašysime, papildomas taisykles, aprašančias srauto ir reaktyvių operatorių panaudojimą:

- Kad galėtume naudoti reaktyvius operatorius, pirmiausia reikia sukurti srauto objekto egzemplioriu.

- Galima jungti skirtingus srauto objektų egzempliorius.

- Srauto objekto egzemplioriui galima kviesti reaktyvius operatorius.

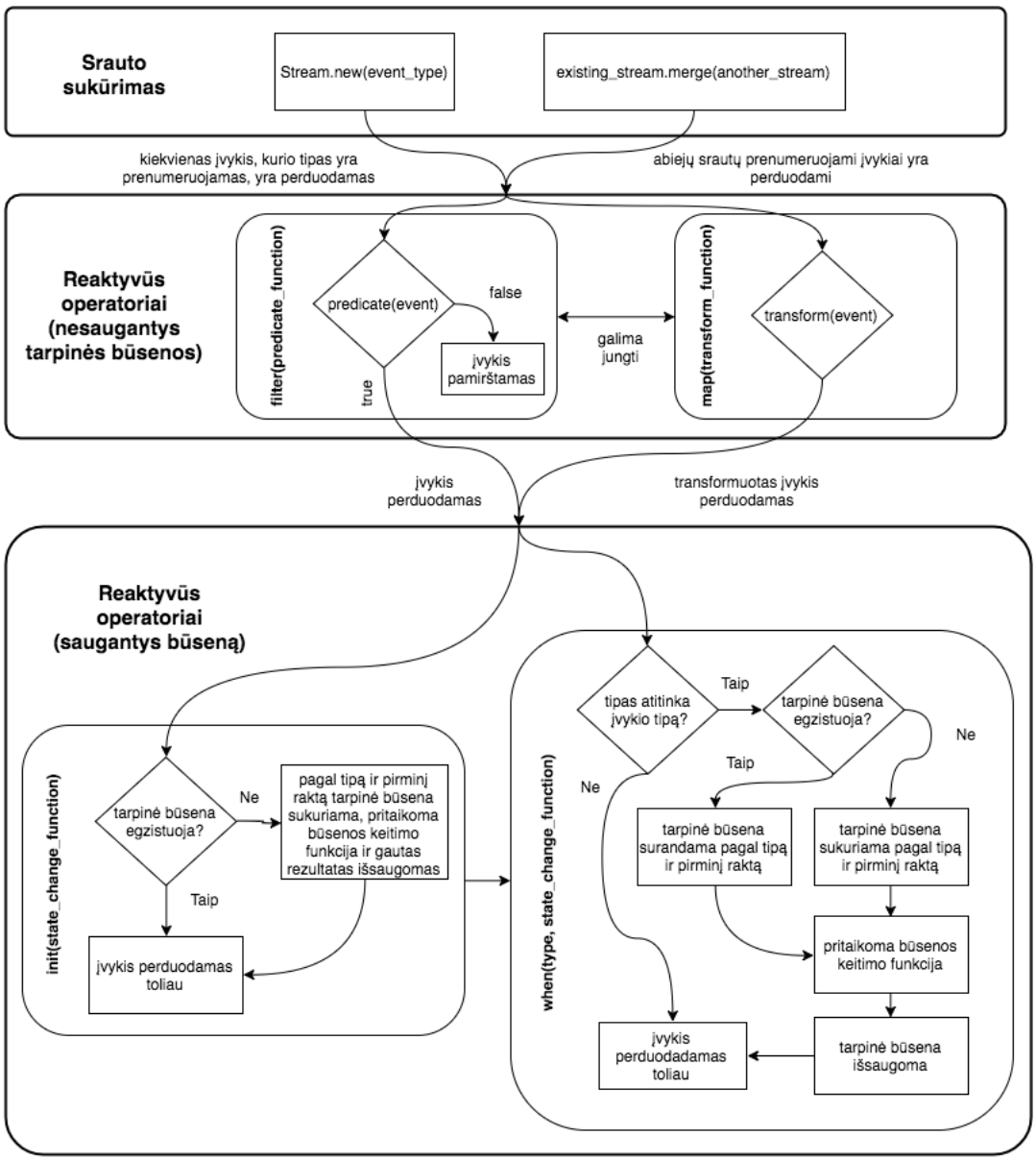
- Vienam srauto objekto egzemplioriui rekomenduojama kurti tik vieną skaitymo modelį.

- Tarpinę būseną saugančius operatorius galima kviesti tik tada, kai srautui būna perduotas tipas ir pirminis raktas. Sukurtoje bibliotekoje šią atsakomybę įgyvendina tipo apibrėžimo operatorius `as_persistent_type(type, primary_key)`.

- Kiekvienas įvykis privalo turėti kuriamo skaitymo modelio pirminio rakto informaciją, saugomą tokiu pačiu vardu, kokį turi pirminis raktas.

- Pradinės būsenos operatorius `init(state_change_function)` privalo būti kviečiamas prieš iteratoriaus `each(state_change_function)` bei tipo atitikimo operatorių `when(type, state_change_function)`. Pastarieji operatoriai sukuria tarpinę būseną nepritaikant pradinės būsenos, todėl pradinės būsenos operatorius prarastų prasmę.

- Kuriamas skaitymo modelis turi būti denormalizuotas, tai yra turi turėti visą informaciją, reikalingą jo panaudojimo atvejui.



7 pav. Srauto ir reaktyvių operatorių realizacijos taisyklių diagrama

4.5.10. Apribojimai

Sujungus reaktyvaus programavimo ir įvykių kaupimo principus ir sukūrus biblioteką, pastebėti tam tikri apribojimai, kuriuos verta paminėti:

- Saugant tarpinę skaitymo modelio tipo būseną reikalingas ne tik tipas, bet ir to skaitymo modelio pirminis raktas. Šis identifikatorius paimamas iš prenumeruojamo įvykio nešamos informacijos. Dėl šio priežasties kiekvienas prenumeruojamas įvykis privalo turėti šią informaciją.
- Analizuojant įvykių kaupimo sistemų kūrimą bei teoriją, buvo pastebėta, jog kuriamas skaitymo modelis yra denormalizuotas, tai yra jis turi visą informaciją, reikalingą vaizdui pateikti. Dėl šios priežasties, biblioteka nepalaiko kelių skaitymo modelių, sujungtų išoriniais raktais, kūrimo.

- Srautas nėra baigtinis, operatorių funkcijos pritaikomos tik nutikus įvykiui, todėl neįmanoma užklausti pagal kintantį laiką. Pavyzdžiui, negalėtume sukurti tam tikro skaitymo modelio, kuris vaizduotų operacijas, įvykusias per paskutines 24 valandas. Norint tikslumo, tam derėtų pasinaudoti įvykių saugyklos adapteriu `EventRepository`, skaityti paskutinius įvykius sistemoje ir kurti laikiną skaitymo modelį nesinaudojant biblioteka.
- Sukurta biblioteka niekada nesunaikina tarpinės skaitymo modelio būsenos. Esant reikalui identifikuoti sunaikintą būseną, galima prenumeruoti naikinimo įvykį ir turėti tam tikrą `status` identifikatorių, kuris pažymėtų skaitymo modelio būseną sistemoje. Pavyzdžiui, elektroninėje parduotuvėje gali tekti žinoti ar produktas pradedamas kelti, įkeltas, suspenduotas ar ištrintas.

4.5.11. Pavyzdinis projektas panaudojantis aprašytą biblioteką

Pilnas išeities kodas yra prieinamas atviro kodo talpykloje "Github", projektas pavadintas "FRP-EventSourcing"[36] vardu. Pagrindinės nuorodos:

- `README.md` - aprašytos diegimo instrukcijos, testų paleidimo instrukcijos bei bibliotekos naudojimosi instrukcijos.
- `app/lib` - bibliotekos failai, reikalingi įvykių publikavimui ir skaitymo modelio kūrimui panaudojant reaktyvųjį programavimą.
- `spec/lib` - testai, užtikrinantys korektišką bibliotekos elgesį, tai yra validuojantys funkcionalumą.

[36] https://github.com/ZilvinasKucinskas/FRP-EventSourcing

Rezultatai ir išvados

Darbo rezultatai:

- Sukurtas pavyzdinis architektūros modelis, apjungiantis reaktyvaus programavimo bei įvykių kaupimo principus, aprašytos praktinės įvykių kaupimo sistemų kūrimo gairės.
- Sukurta biblioteka, įrodanti, jog tikslas yra pasiekiamas. Biblioteka gali tiek publikuoti įvykius, tiek kurti skaitymo modelį naudojantis reaktyviais operatoriais ir paslepiant operacijų su duomenų saugykla realizacijos detales.
- Aprašyta sukurtos bibliotekos realizacijos detalės, taisyklės ir suformuluoti apribojimai. Įgyvendinti srautų sujungimo `merge`, filtravimo `filter`, transformavimo `map`, kuriamo skaitymo modelio tipo apibrėžimo `as_persistent_type`, pradinės reišmės `init`, tipo atitikimo `when` bei iteratoriaus `each` deklaratyvūs operatoriai, įgalinantys skaitymo modelio kūrimą paslepiant operacijų su duomenų saugykla realizacijos detales.g
- Sukurtos bibliotekos funkcionalumas validuotas parašant jai naudojimosi testus.

Išvados:

Įmanoma pritaikyti reaktyvaus programavimo principus įvykių kaupimo sistemose taip, jog skaitymo modelis būtų kuriamas tik komponavimo būdu, paslepiant operacijų su duomenų saugykla realizacijos detales griežtai tipizuotoje programavimo kalboje. Tikslui pasiekti tinka griežtai tipizuota “Ruby” programavimo kalba.

Siekiant tikslo susidurta su dvejomis pagrindinėmis problemomis: kaip realizuoti reaktyvius operatorius ir kaip gudriai išsaugoti tarpinę programos būseną paslepiant operacijų su duomenų baze detales. Pastarosios problemos yra išsprendžiamos:

- Reaktyvūs operatoriai gali būti realizuojami pasinaudojant stebėtojo projektavimo šablonu, objektinio programavimo sąvybėmis (paveldėjimu, polimorfizmu) ir introspekcija.
- Tarpinės būsenos saugojimo problema yra išsprendžiama pasinaudojant metaprogramavimu (introspekcija ir refleksija) ir metodų jungimo principu (angl. method chaining).

Pasiūlymai tolimesniam darbui:

- Plečiant ir modifikuojant kuriamą įvykių kaupimo sistemą arba įvykus nenumatytai klaidai, gali tekti atkurti skaitymo modelį iš naujo. Dėl šios priežasties būtų verta atlikti skaitymo modelio būsenos atkūrimo būdų lyginamąją analizę. Reikėtų išnagrinėti ką siūlo esamos įvykių kaupimo bibliotekos bei pritaikyti tinkamą sprendimą šiame darbe sukurtai bibliotekai.
- Įrodyti koks rinkinys reaktyvių operatorių yra pakankamas, norint užtikrinti denormalizuoto skaitymo modelio kūrimo išsprendžiamumą tik komponavimo būdu.

Literatūra

[Arm97] Joe Armstrong. The development of erlang. *Proceedings of the second acm sigplan international conference on functional programming*. ICFP '97. ACM, Amsterdam, The Netherlands, 1997, p. 196–203. ISBN: 0-89791-918-1. DOI: 10.1145/258948.258967. URL: http://doi.acm.org/10.1145/258948.258967.

[BC13] Jafar Husain Ben Christensen. Reactive programming in the netflix api with rxjava. http://techblog.netflix.com/2013/02/rxjava-netflix-api.html. 2013.

[BCCMM13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx ir Wolfgang de Meuter. A survey on reactive programming. *Acm comput. surv.*, 45(4):52:1–52:34, 2013-08. ISSN: 0360-0300. DOI: 10.1145/2501654.2501666. URL: http://doi.acm.org/10.1145/2501654.2501666.

[BDMSS13] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi ir Mani Subramanian. *Exploring cqrs and event sourcing: a journey into high scalability, availability, and maintainability with windows azure*. Microsoft patterns ir practices, 1st leid., 2013. ISBN: 1621140164, 9781621140160.

[BFKT14] Jonas Bonér, Dave Farley, Roland Kuhn ir Martin Thompson. The reactive manifesto, 2014. URL: http://www.reactivemanifesto.org/.

[CC13] Evan Czaplicki ir Stephen Chong. Asynchronous functional reactive programming for GUIs. *Proceedings of the 34th acm sigplan conference on programming language design and implementation*. ACM Press, New York, NY, USA, 2013-06, p. 411–422.

[CK06] Gregory H. Cooper ir Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. *Proceedings of the 15th european conference on programming languages and systems*. ESOP'06. Springer-Verlag, Vienna, Austria, 2006, p. 294–308. ISBN: 3-540-33095-X, 978-3-540-33095-0. DOI: 10.1007/11693024_20. URL: http://dx.doi.org/10.1007/11693024_20.

[Doc15] Event Store Documentation. Event sourcing basics. http://docs.geteventstore.com/introduction/3.9.0/event-sourcing-basics/. 2015.

[EH97] Conal Elliott ir Paul Hudak. Functional reactive animation. *International conference on functional programming*, 1997. URL: http://conal.net/papers/icfp97/.

[EK14] Benjamin Erb ir Frank Kargl. Combining discrete event simulations and event sourcing. *Proceedings of the 7th international icst conference on simulation tools and techniques*. SIMUTools '14. ICST (Institute for Computer Sciences, Social-Informatics ir Telecommunications Engineering), Lisbon, Portugal, 2014, p. 51–55. ISBN: 978-1-63190-007-5. DOI: 10.4108/icst.simutools.2014.254624. URL: https://doi.org/10.4108/icst.simutools.2014.254624.

[Ell15] Conal Elliott. The essence and origins of functional reactive programming. https://youtu.be/j3Q32brCUAI. 2015.

[Eva04] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley, Boston, 2004. ISBN: 0321125215 9780321125217.

[FB99] Armando Fox ir Eric A. Brewer. Harvest, yield, and scalable tolerant systems. *Proceedings of the the seventh workshop on hot topics in operating systems*. HOTOS '99. 174-178 psl. IEEE Computer Society, Washington, DC, USA, 1999, p. 174–. ISBN: 0-7695-0237-7. URL: http://dl.acm.org/citation.cfm?id=822076.822436.

[Fow02] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN: 0321127420.

[Fow11] Martin Fowler. The lmax architecture. http://martinfowler.com/articles/lmax.html. 2011.

[Fow12] Martin Fowler. Ddd aggregate. https://martinfowler.com/bliki/DDD_Aggregate.html. 2012.

[Fow15a] Martin Fowler. Event sourcing. http://martinfowler.com/eaaDev/EventSourcing.html. 2015.

[Fow15b] Martin Fowler. Retroactive event. http://martinfowler.com/eaaDev/RetroactiveEvent.html. 2015.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson ir John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN: 0-201-63361-2. 326 psl.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? *Symposium on reliability in distributed software and database systems*. IEEE Computer Society, 1986, p. 3–12. ISBN: 0-8186-0690-8. URL: http://dblp.uni-trier.de/db/conf/srds/srds86.html#Gray86.

[Gun98] Neil J. Gunther. *The practical performance analyst : performance-by-design techniques for distributed systems. McGraw-Hill series on computer communications*. MCGRAW-HILL, New York, 1998. ISBN: 0-07-912946-3. URL: http://opac.inria.fr/record=b1118020.

[JC15] A. S. A. Jeffrey ir T. Van Cutsem. Functional reactive programming with nothing but promises: implementing push/pull frp using javascript promises. *Proc. workshop on reactive and event-based languages and systems*, 2015.

[Kla13a] Steve Klabnik. Functional reactive programming. `https://www.youtube.com/watch?v=0qv3hWgC950`. Euruko conference. 2013.

[Kla13b] Steve Klabnik. Functional reactive programming in ruby with frappuccino. `https://www.youtube.com/watch?v=XlyQOLRiEPs`. RubyConf India conference. 2013.

[KP88] Glenn E. Krasner ir Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. object oriented program.*, 1(3):26–49, 1988-08. ISSN: 0896-8438. URL: `http://dl.acm.org/citation.cfm?id=50757.50759`.

[Kra13] Greg Krasser. Introduction to akka persistence. `http://krasserm.blogspot.lt/2013/12/introduction-to-akka-persistence.html`. 2013.

[Kra15] Greg Krasser. Event sourcing and cqrs with akka persistence and eventuate. `https://www.youtube.com/watch?v=vFVry457XLk`. 2015.

[Llo94] JW Lloyd. *Practical advantages of declarative programming*. *Unknown*. Conference Proceedings/Title of Journal: Joint Conference on Declarative Programming, 1994, p. 3 –17.

[Mey88] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st leid., 1988. ISBN: 0136290493. 747 psl.

[MGBCGBK09] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield ir Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. *Sigplan not.*, 44(10):1–20, 2009-10. ISSN: 0362-1340. DOI: `10.1145/1639949.1640091`. URL: `http://doi.acm.org/10.1145/1639949.1640091`.

[Ols] Russ Olsen. *Design patterns in ruby (addison-wesley professional ruby series)*. Addison-Wesley Professional, 1 leid. ISBN: 0321490452, 9780321490452.

[Rod85] David P. Rodgers. Improvements in multiprocessor system design. *Sigarch comput. archit. news*, 13(3):225–231, 1985-06. ISSN: 0163-5964. DOI: `10.1145/327070.327215`. URL: `http://doi.acm.org/10.1145/327070.327215`.

[SAPM14] Guido Salvaneschi, Sven Amann, Sebastian Proksch ir Mira Mezini. An empirical study on program comprehension with reactive programming. *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*. FSE 2014. ACM, Hong Kong, China, 2014, p. 564–575. ISBN: 978-1-4503-3056-5. DOI: `10.1145/2635868.2635895`. URL: `http://doi.acm.org/10.1145/2635868.2635895`.

[SC03] U.S. Securities and Exchange Commission. Electronic storage of broker-dealer records. https://www.sec.gov/rules/interp/34-47806.htm. 2003.

[SHM14] Guido Salvaneschi, Gerold Hintz ir Mira Mezini. Rescala: bridging between object-oriented and functional style in reactive applications. *Proceedings of the 13th international conference on modularity*. MODULARITY '14. ACM, Lugano, Switzerland, 2014, p. 25–36. ISBN: 978-1-4503-2772-5. DOI: 10.1145/2577080.2577083. URL: http://doi.acm.org/10.1145/2577080.2577083.

[Ste13] Stoyan Stefanov. Remarkable react. http://www.phpied.com/remarkable-react/. BrazilJS conference. 2013.

[Ver12] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley Professional, [S.l.], 2012. URL: https://cds.cern.ch/record/1555590.

[Vie15] M. Viering. An efficient runtime system for reactive programming. Magistro darbas. Technische Universität Darmstadt, 2015.

[WTH02] Zhanyong Wan, Walid Taha ir Paul Hudak. Event-driven FRP. *Practical aspects of declarative languages (PADL'02)*. 155–172 psl., 2002-01.

[You10a] Greg Young. Cqrs documents by greg young. Technical Whitepaper. 2010.

[You10b] Greg Young. Cqrs, task based uis, event sourcing agh! http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/. 2010.

Sąvokų apibrėžimai

- **Agregatas** (angl. aggregate) - DDD modelis, rinkinys domeno objektų, kurie gali būti laikomi kaip visuma.
- **Nuoseklumas arba darna** (angl. consistency) - Duomenų ir jų savybių, esančių toje pačioje sistemoje, logiškumas, vieno su kitu suderintumas.
- **Derinimas** (angl. debugging) - riktų bei klaidų paieška programinėje įrangoje bei jų taisymas.
- **Išplečiamumas** (angl. scaling) – galimybė sujungti daugybę techninės ar programinės įrangos esybių taip, jog jos dirbtų kaip visuma. Pavyzdžiui, galima pridėti keletą serverių pasinaudojant grupavimu arba apkrovos paskirstymu taip pagerinant sistemos našumą bei prieinamumą.
- **Introspekcija** (angl. introspection) - programos gebėjimas nagrinėti objekto tipą ir sąvybes programos vykdymo metu.
- **Metodų jungimo principas** (angl. method chaining) - objektinės programavimo kalbos sintaksė, leidžianti iškviesti kelis metodus iš eilės. Kiekvienas metodas grąžina objektą, taip leidžiant jungti metodų kvietimus viename apraše ir nereikalaujant papildomų kintamųjų tarpinės būsenos saugojimui.
- **Modulis** (angl. module) - savarankiška programos dalis, į kurią sudėti duomenys ir veiksmai su jais.
- **Refleksija** (angl. reflection) - programos gebėjimas tiek nagrinėti objektus (introspekcija), tiek modifikuoti save (elgesį arba būseną) programos vykdymo metu.

Santrumpos ir paaiškinimai

- **API** (angl. Application programming interface) - rinkinys funkcijų ar procedūrų, leidžiančių kurti programas, kurios turi prieigą prie kitos operacinės sistemos, programos ar paslaugos informacijos ar tam tikro funkcionalumo.
- **CQS** (angl. Command Query Separation) - komandų-užklausų atskyrimas.
- **CQRS** (angl. Command Query Responsibility Segregation) – komandų-užklausų atsakomybių atskyrimas.
- **DDD** (angl. Domain-Driven Design) – domenu pagrįstas projektavimas - būdas kurti programinę įrangą, skirtą spręsti sudėtingus uždavinius, bei apjungti realizaciją kartu su augančiu domeno modeliu.
- **ES** (angl. Event Sourcing) - įvykių kaupimo principas.
- **ES+CQRS** - įvykių kaupimo bei komandų-užklausų atskyrimo principai, kurie dažniausiai yra naudojami kartu projektuojant sistemą.
- **FRP** (angl. Functional Reactive Programming) - funkcinis reaktyvus programavimas.
- **RP** (angl. Reactive Programming) - reaktyvus programavimas.

Priedas nr. 1

“Frappuccino” bibliotekos išeities kodo pavyzdžiai

```
class Button
  def push
    emit(:pushed) # emit sends a value into the stream
  end
end

button = Button.new
stream = Frappuccino::Stream.new(button)

counter = stream
          .map { |event| event == :pushed ? 1 : 0 } # convert events to ints
          .inject(0) { |sum, n| sum + n } # add them up

counter.now # => 0

button.push
button.push
button.push

counter.now # => 3

button.push

counter.now # => 4
```

Kodo pavyzdys 3: “Frappuccino” bibliotekos panaudojimo pavyzdys

Priedas nr. 2

Įvykių kaupimo sistemos išeities kodo pavyzdžiai

```
    create_table(:event_store_events) do |t|
      t.string    :stream,    null: false
      t.string    :event_type, null: false
      t.string    :event_id,  null: false
      t.text      :metadata
      t.text      :data,      null: false
      t.datetime  :created_at, null: false
    end
    add_index :event_store_events, :stream
    add_index :event_store_events, :created_at
    add_index :event_store_events, :event_type
    add_index :event_store_events, :event_id, unique: true
```

Kodo pavyzdys 4: Įvykių saugojimo migracija

```
class CartReadModel
    def call(event)
        if event.event_type == 'AddToCart'
            add_to_cart(event.data)
        end
        if event.event_type == 'RemoveFromCart'
            remove_from_cart(event.data)
        end
    end
    private
    def add_to_cart(event_data)
        #Implementation here
    end
    def remove_from_cart(event_data)
        #Implementation here
    end
end

cart = CartReadModel.new
client.subscribe(cart, [AddToCart, RemoveFromCart])
```

Kodo pavyzdys 5: “EventStore” bibliotekos prenumeratorių mechanizmo panaudojimas

```
module Domain
  class Order
```

```
    include AggregateRoot::Base

    AlreadyCreated  = Class.new(StandardError)
    MissingCustomer = Class.new(StandardError)

    def initialize(id = SecureRandom.uuid)
      @id = id
      @state = :draft
    end

    def create(order_number, customer_id)
      raise AlreadyCreated unless state == :draft
      raise MissingCustomer unless customer_id
      apply Events::OrderCreated.create(@id, order_number, customer_id)
    end

    # ...

    def apply_order_created(event)
      @customer_id = event.customer_id
      @number = event.order_number
      @state = :created
    end

    private
    attr_accessor :id, :customer_id, :order_number, :state

    # ...
  end
end
```

Kodo pavyzdys 6: Įvykių pritaikymas domeno objektui

```
module AggregateRoot
  class DefaultApplyStrategy
    def call(aggregate, event)
      event_name_processed = event.class.name.demodulize.underscore
      aggregate.method("apply_#{event_name_processed}").call(event)
    end
  end
end

module AggregateRoot
  def apply(event)
```

```
    apply_strategy.(self, event)
    unpublished_events << event
  end

  private

  def unpublished_events
    @unpublished_events ||= []
  end

  def apply_strategy
    DefaultApplyStrategy.new
  end
end
```

Kodo pavyzdys 7: “AggregateRoot” modulis

```
module Command
  class Base
    include ActiveModel::Model
    include ActiveModel::Validations
    include ActiveModel::Conversion

    def initialize(attributes={})
      super
    end

    def validate!
      raise ValidationError, errors unless valid?
    end

    def persisted?
      false
    end
  end
end
```

Kodo pavyzdys 8: Tėvinė komandų klasė

```
module Command
  class Handler
    def initialize(repository:, **_)
      @repository = repository
    end
```

```
    protected
    def with_aggregate(aggregate_id)
      aggregate = build(aggregate_id)
      yield aggregate
      repository.store(aggregate)
    end

    private
    attr_accessor :repository

    def build(aggregate_id)
      aggregate_class.new(aggregate_id).tap do |aggregate|
        repository.load(aggregate)
      end
    end
  end
end
```

Kodo pavyzdys 9: Komandų doroklė

```
module Command
  module Execute
    def execute(command, **args)
      command.validate!
      args = AggregateRoot::Repository.new(event_store) if args.empty?
      handler_for(command).new(**args).call(command)
    end

    private
    def handler_for(command)
      {
        Command::CreateOrder      => CommandHandlers::CreateOrder,
        Command::AddItemToBasket  => CommandHandlers::AddItemToBasket,
        Command::RemoveItemFromBasket => CommandHandlers::RemoveItemFromBasket,
      }.fetch(command.class)
    end
  end
end
```

Kodo pavyzdys 10: Komandų priskyrimas doroklėms