VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Funkcinio-reaktyvaus programavimo taikymas įvykių kaupimo sistemose

Functional Reactive Programming in Event Sourcing Systems

Mokslo tiriamasis darbas

Atliko: 1 kurso 7 grupės studentas
Žilvinas Kučinskas ..... (parašas)

Darbo vadovas:
Viačeslav Pozdniakov ..... (parašas)

Recenzentas:
prof. Rimantas Vaicekauskas ..... (parašas)

Vilnius, 2014 m.

Turinys

1. Magistro darbo objekto apžvalga bei tyrimo problemos aprašymas 3
1.1. Tyrimo objektas 3
1.2. Darbo tikslai ir uždaviniai 3
1.3. Tyrimo aktualumas 3
1.4. Pritaikymo pavyzdys 5
1.5. Tyrimo metodika 6
1.6. Laukiami rezultatai 7
2. Literatūros analizė 8
2.1. Funkcinis-reaktyvus programavimas 8
2.1.1. Įvadas 8
2.1.2. Pagrindinis tiklas 8
2.1.3. Sąvokos 8
2.1.4. Sąvybės 9
2.1.5. Įvykių srautas 9
2.1.6. Įvykių srautas funkciniame programavime 10
2.2. Įvykių kaupimas 15
2.2.1. Įvadas 15
2.2.2. Momentinė kopija 17
2.2.3. Įvykių kaupimo privalumai ir trūkumai 17
2.2.4. Įvykių kaupimas funkciniame programavime 18
2.3. Monados 18
2.3.1. Terminas 18
2.3.2. Taisyklės 19
2.3.3. Tolesnis darbas 19
2.4. Išvados 20
3. Sąvokų apibrėžimai 21
4. Santrumpos ir paaiškinimai 22
Literatūros sąrašas 23

1. Magistro darbo objekto apžvalga bei tyrimo problemos aprašymas

1.1. Tyrimo objektas

Tyrimo objektas yra funkcinio-reaktyvaus programavimo bei įvykių kaupimo principai.

1.2. Darbo tikslai ir uždaviniai

Darbo tikslas yra išnagrinėti funkcinio-reaktyvaus programavimo pritaikymo galimybes įvykių kaupimo sistemose.

Siekiant šio tikslo, turi būti išspręsti šie uždaviniai:

- įrodyti, kad funkcinį-reaktyvų programavimą įmanoma taikyti įvykių kaupimo sistemose;
- sukurti konkretizuotą kalbą (angl. domain specific language), apjungiančią funkcinio-reaktyvaus programavimo bei įvykių kaupimo principus;
- aprašyti konkretizuotuos kalbos kūrimo metodiką, apibrėžti gautų rezultatų apribojimus, suformuluoti iškilusias problemas bei paaiškinti jų priežastis.

1.3. Tyrimo aktualumas

Funkcinis reaktyvus programavimas integruoja laiko tėkmę bei sudėtinius įvykius į funkcinį programavimą. Šis principas suteikia elegantišką būdą išreikšti skaičiavimus interaktyvių animacijų, robotikos, kompiuterinio vaizdavimo, vartotojo sąsajos ir modeliavimo srityse [Cza12, p. 4]. Pagrindinės funkcinio reaktyvaus programavimo sąvokos:

- signalai arba elgsena - reikšmės, besikeičiančios bėgant laikui;
- įvykiai - momentinių reikšmių kolekcijos arba laiko-reikšmės poros.

Funkcinis-reaktyvus programavimas įgalina apsirašyti elgseną deklaratyviai naudojant imperatyvios programavimo kalbos struktūras [EH97, p.1]. Elgsena ir įvykiai

gali būti komponuojami kartu, išreikšti vienas per kitą. Funkcinis reaktyvus programavimas apibrėžia kaip signalai arba elgsena reaguoja į įvykius. [Ams, p. 1] Šį principą galima iliustruoti pavyzdžiu. Tarkime turime Excel lapą, kuriame yra trys laukai: darbuotojo pradirbtos valandos, valandinis užmokestis bei formulė, kuri paskaičiuoja konkretų darbuotojo atlyginimą. Darbuotojui pradirbus daugiau valandų, atnaujinamas pradirbtų valandų skaičius. Kartu atsinaujina ir pačios formulės reikšmė, tai yra konkretus užmokestis pakinta. Šiuo atveju įvykus reikšmės atnaujinimo įvykiui, nuo jos priklausomos formulės taipogi atsinaujina arba tam tikras įvykis iššaukia elgseną sistemoje.

Įvykių kaupimo principo esmė – objektas yra atvaizduojamas kaip įvykių seka. Kaip pavyzdį tai galima parodyti remiantis banko sąskaita. Tarkime vartotojas, banko klientas, turi 100 litų sąskaitos balansą. Tarkime vartotojas nusipirko prekę už 20 litų, tada įnešė į savo sąskaitą 15 litų ir galiausiai nusipirko tam tikrą paslaugą už 30 litų. Akivaizdu, jog turint šią įvykių seką, galima atvaizduoti dabartinę objekto būseną - tai yra 65 litai vartotojo sąskaitoje. Įvykių kaupimo principas užtikrina, jog visi būsenos pasikeitimai yra saugomi įvykių žurnale kaip įvykų seka [Ver13]. Įvykių kaupimo principui yra būdinga, jog įvykių negalimą 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 funkcinį-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 viename faile 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.

1.4. Pritaikymo pavyzdys

Tarkime turime domeno sritį - bankininkystė. Turime įvykių srautą - vartotojų sukūrimas. Naudojant įsivaizduojamą Scala API galima sukurti vartotojų paieškos puslapį pagal vardą ir pavardę. Demonstacija pateikta 3 kodo pavyzdyje.

```
// stream model
case class CustomerCreate (val name: String, val surname: String, val personalNum:
    String)

val es = EventSourceConnection(url)
val createStream = Stream(es, "customerCreate")

class case CustomerModel(val name: String, val surname: String, val personalNum:
    String) extends ViewableModel

trait CustomerArgModel extends Arg2Model[String,String]{
  val name: Option[String]
  val surname: Option[String]
}

//args are passed on form view/post
val customerView = View(args: CustomerArgModel).foldLeft(
  (acc,event) => event match {
    case CustomerCreate(name, surname, personalNum) =>
      for {
        newName <- name
        newSurname <- surname
        newPersNum <- personalNum
        if (args.name == name && args.surname == surname)
      } yield CustomerModel(newName, newSur, newPersNum)
  }
)
// getting data for all Kucinskai
val specificData = customerView(None, Some("Kucinskas")):
    Option[List[CustomerModel]]
```

Kodo pavyzdys 1 - vartotojų paieškos puslapis naudojant įsivaizduojamą Scala API

Šiuo atveju veiksmai peržiūrint duomenis yra sumaišomi kartu su veiksmais gaunant įvykius. Verta pastebėti, jog lokali duomenų saugykla nebuvo paminėta arba apibrėžta. Pastaroji gali būti sugeneruota bei valdoma automatiškai.

Antruoju pavyzdiniu atveju turimas vartotojo balanso(įplaukiančios/išplaukiančios lėšos) įvykių srautas ir norima gauti vartotojo, kurio asmens kodas yra *39008226547*, einamosios savaitės sąskaitos balansą. Demonstacija pateikta 2 kodo pavyzdyje.

```
val duration = 1.weeks
val personalNum = "39008226547"
val balanceStream = Stream(es, "customerBalance")
val notOlderThanOneWeek =
    for {
        event <− balanceStream
        filtered  <− event.filter(_.personalNum == personalNum
            && (DateTime.now − _.timeStamp) >= duration)
    } yield  filtered
val sum = notOlderThanOneWeek.toList.sum
```

Kodo pavyzdys 2 - vartotojo einamosios sąskaitos balansas naudojant įsivaizduojamą Scala API

Šiuo atveju *event* kintamasis yra įvykių srauto monada (terminas vartojamas funkciniame programavime, kilęs iš kategorijų teorijos ir turi savas taisykles), o *filtered* kintamasis - duomenų saugyklos monada. Bendruoju atveju skirtingos monados tarpusavyje nesiderina [KW92]. Dėl to reikia išsiaiškinti - ar yra įmanoma ir kaip šias monadas suderinti.

1.5. Tyrimo metodika

Darbo analitinėje dalyje bus naudojami tradiciniai bibliotekinio tyrimo metodai. Darbo tikslui pasiekti tiriamojoje dalyje bus pasirinkta konkreti funkcinė programavimo kalba (pvz.: Haskell, Scala) bei aprašoma kūrimo metodika.

Papildomai magistro darbo rašymo metu ketinama užbaigti Erik Meijer, Martin Odersky, Roland Kuhn dėstomą reaktyvaus programavimo kursą (Coursera). Esant galimybėms žadama apsilankyti į funkcinį programavimą orientuotose konferencijose Scala Days (http://scaladays.org/), Scalar (http://scalar-conf.com/) ar kitose.

1.6. Laukiami rezultatai

Magistrinio darbo metu planuojama išnagrinėti funkcinio-reaktyvaus programavimo ir įvykio kaupimo principus, įrodyti, jog šie principai gali būti panaudoti kartu bei suderinti, sukurti konkretizuotą kalbą (angl. domain specific language), apjungiančią šiuos principus, bei aprašyti kūrimo eigos metodiką, apibrėžti gautus rezultatus, suformuluoti apribojimus, iškilusias problemas bei paaiškinti jų priežastis.

2. Literatūros analizė

2.1. Funkcinis-reaktyvus programavimas

2.1.1. Įvadas

Anot [Ams], funkcinis-reaktyvus programavimas, toliau dažnai vadinamas tiesiog FRP, yra būdas modeliuoti reaktyvų - besikeičiančius laiko tekmėje bei reaguojančius į išorinį stimulą - elgesį visiškai funkcinėse programavimo kalbose. FRP leidžia deklaratyviu ir paprastu būdu modeliuoti sistemas, kurios turi reaguoti į duomenis bėgant laikui.

2.1.2. Pagrindinis tiklas

Pagrindinis funkcinio-reaktyvaus programavimo tikslas [Ams]:

- saugus programavimas - kompiliatorius turi kiek įmanoma patikrinti programų korektiškumą;
- efektyvus programavimas - programos turėtų veikti realiu laiku, todėl efektyvios ir optimizuotos operacijos yra būtinos;
- komponavimas - FRP leidžia kurti programas iš smulkesnių programų, o ne orientuotą į problemą, vientisą kodą.

2.1.3. Sąvokos

Pagrindinės FRP sąvokos yra:

- signalai arba elgsena - besikeičiančios laike reikšmės;
- įvykiai - kolekcija momentinių reikšmių arba laiko-reikšmės poros.

FRP pasiekia reaktyvumą naudodamas konstrukcijas, kurios tiksliai apibrėžia kaip signalai arba elgsena pasikeičia reaguodami į įvykius. Tai yra pagrindinis būdas išreiškiant bei realizuojant elgseną. Kitu būdu, elgsena gali būti laiko semantinės funkcijos[1], kuriose laikas yra pakeičiamas kilus įvykiui [NCP02].

[1] http://msdl.cs.mcgill.ca/people/tfeng/docs/as/node5.html

2.1.4. Sąvybės

Anot anksčiausios funkcinio-reaktyvaus programavimo formuluotės [EH97], pagrindinės sąvybės, kuriomis pasižymi FRP:

- elgsenos arba singalų modeliavimas bėgant laikui,
- įvykių, kurie turi baigtinį skaičių atsitikimų daugelyje laiko taškų, modeliavimas,
- perjungimas (angl. switching) - sistema gali pasikeisti dėl atsitikusių įvykių,
- analizės detalių, tokių kaip reaktyvaus modelio įvykių ėmimo dažnis, atskyrimas.

2.1.5. Įvykių srautas

Pagal [Bas], įvykių srautas yra eilė pagal laiką surikiuotų įvykių, pavyzdžiui akcijų rinkos srautas.

Įvykių srautas kaip duomenų srauto tipas formaliai atrodo kaip pora (s, t), kur s yra seka surikiuotų sąrašo įvykių, o t yra seka laiko intervalų ir kiekvienas intervalas yra netuščias.

Tokio duomenų srauto pavyzdžiai gali būti:

- akcijų kursas,
- paspaudimų srautas,
- tinklo srautas,
- GPS[2] duomenys.

Įvykių srauto apdorojimas pagal atsitikimo laiką turi privalumų:

- įvykių apdorojimo algoritmai naudoja mažai atminties, nes jiems nereikia prisiminti daug įvykių;
- algoritmai gali būti labai greiti;

[2]http://en.wikipedia.org/wiki/Global_Positioning_System

- gavus įvykį, skaičiavimai atliekami iškart, todėl galima perduoti rezultatą kitam skaičiavimui ir pamiršti įvykį.

Įvykių srauto apdorojimas labiau akcentuoja didelio našumo duomenų gavimą ir matematinių algoritmų pritaikymą įvykių duomenims. Taip pat įvykių srautai įprastai pritaikomi konkrečiai sistemai ar organizacijai.

2.1.6. Įvykių srautas funkciniame programavime

Vietoje įvykių srauto, galima naudoti klausytojo projektavimo šabloną [OSG04], tačiau ilgame programinės įrangos kūrimo gyvavimo cikle įvykių srauto naudojimas palengvina kai kuriuos dalykus. Pavyzdžiui, imperatyvaus programavimo atžvilgiu, viename metode turime klausytoją, kuris reaguoją į situaciją /textitA, tačiau iškilus situacijai /textitB, šis klausytojas yra pašalinamas. Šiuo atveju programinis kodas, kuris valdo klausytojo gyvavimo ciklą yra įsipynęs keliose skirtingose kodo vietose, ko pasekoje tai reiškia, jog yra sunkiau palaikyti, stebėti, keisti bei suprasti šias vietas. Įvykių srautas yra pirmosios klasės reikšmės, todėl jį galima abstrahuoti [MO12]. Abstrahuojant klausytojų gyvavimo ciklo valdymo idėją tampa lengviau programuoti pagal tai kas turėtų nutikti, o ne pagal tai, ką kompiuteris turėtų daryti toliau. Tačiau turbūt didesnis įvykių srauto pranašumas yra tai, jog pastarasis gali būti transformuojamas ir komponuojamas. Toliau bus aprašomi operacijos arba veiksmai bei sutrumpinti išeities kodo pavyzdžiai, kuriuos galima atlikti su įvykių srautu remiantis Reactive Web[3] karkasu, skirtu Scala[4] programavimo kalbai.

Pirmas veiksmas - įvykio srauto sukūrimas. Jis pavaizduotas 3 kodo pavyzdyje. Pirmiausia sukuriamas įvykių šaltinis, kuris po to priskiriamas įvykių srautui.

```
val eventSource = new EventSource[String]{}
scheduleTask(10000) {
        eventSource.fire("Event after 10 seconds")
}

val eventStream: EventStream[String] = eventSource
```

Kodo pavyzdys 3 - įvykių srauto sukūrimas

[3]http://scalareactive.org
[4]http://www.scala-lang.org

Taip pat *EventSource* (įvykių šaltinis) turi naudingą poklasį *Timer* (laikrodis). Jo realizacija pavaizduota 4 kodo pavyzdyje. Šis sukuria tiksėjimo įvykius duotu laiko intervalu.

```
val timer = new Timer(0, 2000, {t => t >= 32000})


for(t <- timer)
yield "timer: " + t.toString
```

Kodo pavyzdys 4 - įvykių srauto sukūrimas

Įvykių srautas valdo kolekciją klausytojų, tačiau konceptualiai reikėtų mąstyti kitaip. Klausytojo pridėjimas iš tikrųjų reiškia funkcijos iškvietimą kiekvienam kilusiui įvykiui, kitais žodžiais kiekvienam įvykių srauto įvykiui. Lygiai taip pat kaip Scala programavimo kalboje norint įvykdyti funkciją kiekvienai kolekcijos reikšmei yra iškviečiama *foreach* funkcija. Tačiau įvykių srauto atveju, *foreach* grąžina rezultatą akimirksniu, o funkcija yra išsaugoma ir vėliau įvykdoma kai tik gaunamas koks nors įvykis. Šio atvejo praktinis variantas pademonstruotas 5 kodo pavyzdyje.

```
val eventSource = new EventSource[String] {}


// The following is syntactic sugar for
// eventSource.foreach(event => alert("You fired: '" + event + "'"))
for(event <- eventSource) {
        alert("You fired: '" + event + "'")
}
```

Kodo pavyzdys 5 - klausytojų pridėjimas

Rinkinys transformacijų metodų įvykių srautą padaro labai universaliu. Šios transformacijos grąžina modifikuotą įvykių srautą. Transformacijas galima atlikti viena po kitos. Tai primena transformacijas galimas su Scala kolekcijomis, pavyzdžiui 6 kodo pavyzdyje. Kai tik originalus įvykių srautas gauna įvykį, transformuotas įvykių srautas gauna savo įvykį pritaikant klausytojo funkciją.

```
List(1,2,3).map(_ * 10).filter(_ < 25)
```

Kodo pavyzdys 6 - kolekcijos transformacijos

Jeigu įvykių srautas gauna labai daug įvykių, tačiau aktualūs yra tik dalis jų, galima naudoti filtravimą, tai yra *filter* metodą. Tai demonstruojama 7 kodo pavyzdyje. Čia kiekvienam įvykiui yra pritaikomas predikatas. Jeigu jis įvertinimas kaip teisingas arba kitaip *true*, įvykis yra gaunamas transformuotame įvykių sraute.

```
val eventSource = new EventSource[String] {}
eventSource. filter (_.length < 5)
```

Kodo pavyzdys 7 - įvykių srauto filtravimas

Kitas esminis kolekcijų metodas yra *map*. Jis leidžia transformuoti kolekciją pritaikant tam tikrą funkciją kiekvienam kolekcijos elementui. Pritaikius yra grąžinama nauja modifikuota kolekcija. Lygiai taip pat šis metodas gali būti pritaikytas įvykių srautui. Šis atvejis demonstruojamas 8 kodo pavyzdyje.

```
val eventSource = new EventSource[String] {}
eventSource.map(_.reverse)
```

Kodo pavyzdys 8 - įvykių visiškas transformavimas

Įvykių srautam taip pat galima pritaikyti *flatmap* metodą. Scala kolekcijose šis metodas pritaiko funkciją visiems elementams, kuri grąžina naują kolekciją, kurioje sujungiamos visos grąžintų kolekcijų reikšmės. Pavyzdys pateiktas kodo pavyzdyje 9. Operacijos *map* atveju būtų grąžinta reikšmė *List(List(10, 11, 12), List(20, 21, 22), List(30, 31, 32))*, tačiau operacija *flatMap* sujungia grąžintų kolekcijų reikšmes.

```
val original  = List(1,  2,  3)
val flatMapped = original.flatMap(x => List(x*10,x*10+1,x*10_2))
flatMapped == List(10,11,12,  20,21,22,    30,31,32)
```

Kodo pavyzdys 9 - flatmap Scala kolekcijose

Panašiai veikia *flatMap* metodas įvykių srautuose. Pastarasis leidžia sukurti įvykių srautą, kuris iššauna įvykius, kurie yra iššauti kitų įvykių srautų. Metodas demonstruojamas 10 kodo pavyzdyje. Čia figūra yra išplečiama, o po to išnyksta.

```
// Assuming Shape is a case class with  scale  and opacity values
// and millisTimer fires  events  once per millisecond,  starting
// at  zero.
// Scale  should animate from 0 to 1 over the  first  second,
```

```
// and opacity should animate from 1 to 0 over the next.
def compositeAnimation(millisTimer: EventStream[Long], shape: Shape):
    EventStream[Shape] = {
  val scale: EventStream[Double] =
    millisTimer.map(m => m/1000.0)
  val opacity: EventStream[Double] =
    millisTimer.map(m => 1 - (m-1000)/1000.0)
  val seconds = millisTimer.filter(_ % 1000 == 0).map(_ / 1000).
    takeWhile(_ < 2000)

  seconds.flatMap {
    case 0 => scale
    case 1 => opacity
  }
}
```

Kodo pavyzdys 10 - flatmap įvykių sraute

Imperatyviose kalbose, dažna užduotis yra iteruoti per masyvą ir atlikti tam tikrus veiksmus. Šiam veiksmui atlikti naudojamas tam tikras skaičius kintamųjų veiksmam atlikti, pavyzdžiui skaičiavimam. Funkciniame programavime šiam tikslui dažnai yra naudojamas *foldLeft* metodas. Pastarasis priima pradinę reikšmę arba kitaip būseną ir funkciją, kuri priima paskutinę reikšmę arba būseną ir kitą kolekcijos elementą. Kiekvienam elementui funkcija grąžina reikšmę arba būseną, kuri naudojama sekančiam funkcijos kvietimui. Verta pastebėti, jog funkcinis programavimas čia leidžia kodą vykdyti lygiagrečiai jo visiškai nekeičiant. 11 kodo pavyzdys demonstruoja sąrašo elementų sumos skaičiavimą.

```
list.foldLeft(0){(totalSoFar, nextElement) => totalSoFar + nextElement}
//more commonly written as list.foldLeft(0)(_ + _)
```

Kodo pavyzdys 11 - sąrašo elementų skaičiavimas naudojant flatmap

Panašiai *foldleft* metodą galima pritaikyti įvykių srautam. Demonstracija 12 kodo pavyzdyje.

```
case class AvgState(total: Double, count: Int)
```

```
    val eventSource = new EventSource[String] {}


    eventSource.foldLeft(AvgState(0,0)){
  case (AvgState(total, count), s) => AvgState(total+s.length,count+1)
} map {
  case AvgState(total, count) =>
    "Average length so far: " + (total/count)
}
```

Kodo pavyzdys 12 - foldleft metodas įvykių sraute

Įvykių srautus galima ne tik transformuoti, bet ir komponuoti, tai yra sujungti. Demonstracija 13 kodo pavyzdyje. Įvykių srautas *allClicks* iššauna įvykį kai bet kuris iš kitų įvykių srautų iššauna įvykį.

```
    val allClicks = leftClicks | middleClicks | rightClicks
```

Kodo pavyzdys 13 - įvykių srautų sujungimas

Įvykių srautą taip pat galima paversti signalu. Įvykių srautas atvaizduoja įvykius kaip srautą diskrečių reikšmių laiko tekmėje, taigi šios reikšmės egzistuoja momentiškai. Signalas atvaizduoja besitęsiančią reikšmę, tai yra turi dabartinę reikšmę. Kai signalo reikšmė pasikeičia, įvykių srautas iššauna naują įvykį. Įvykių srautą norint paversti signalu galima naudoti *hold* metodą, kuriam yra paduodama pradinė reikšmė. Demonstracija 14 kodo pavyzdyje.

```
    val eventSource = new EventSource[String] {}
    val signal = eventSource.hold("(initial value of signal)")


    def render = Demos.eventSourceInput(eventSource) ++
    <p>Signal value: '{Demos.signalOutput(signal)}'</p>
```

Kodo pavyzdys 14 - įvykių srautų pavertimas signalu

2.2. Įvykių kaupimas

Šiame skyriuje yra aprašomos žinios apie įvykių kaupimą, pliusus ir minusus, įvykių srautus bei įvykių kaupimą funkciniame programavime remiantis Vaughn Vernon surinkta ir aprašyta informacija [Ver13].

2.2.1. Įvadas

Kartais verslui svarbu fiksuoti objekto pasikeitimus domeno modelyje[5]. Šiuos pasikeitimus galima stebėti skirtingais būdais. Įprastai yra pasirenkama stebėti kai esybė[6] yra:

- sukurta,
- paskutinį kartą modifikuota
- bei kas atliko modifikaciją.

Tačiau šis būdas nepateikia jokios informacijos apie vienkartinius pasikeitimus.

Atsiradus poreikiu stebėti pasikeitimus detaliau, verslas reikalauja dar daugiau metaduomenenų[7], ko pasekoje tokie faktai kaip individualios operacijos laiko tekmėje bei jų įvykdymo laikas tampa svarbūs. Šie poreikiai verčia įvesti audito žurnalą fiksuoti labai tikslias panaudojimo atvejų metrikas, tačiau pastarasis būdas turi apribojimų. Jis gali atskleisti dalį informacijos apie tai kas nutiko sistemoje, leisti rasti bei ištaisyti dalį riktų bei klaidų programinėje įrangoje, bet audito žurnalas neleidžia patikrinti domeno objekto būsenos prieš ir po tam tikrų pasikeitimų. O jeigu būtų galima išgauti daugiau informacijos iš pasikeitimų stebėjimo?

Visi programinės įrangos kūrėjai susiduria su labai tiksliu pasikeitimų stebėjimu. Įprastas ir populiarus pavyzdys yra išeities kodo saugyklos, tokios kaip CVS[8], Subversion[9], Git[10] arba Mercurial[11]. Visos šios pataisų valdymo sistemos leidžia stebėti pirminių failų pasikeitimus. Įrankiai leidžia peržiūrėti išeities kodo artefaktus nuo

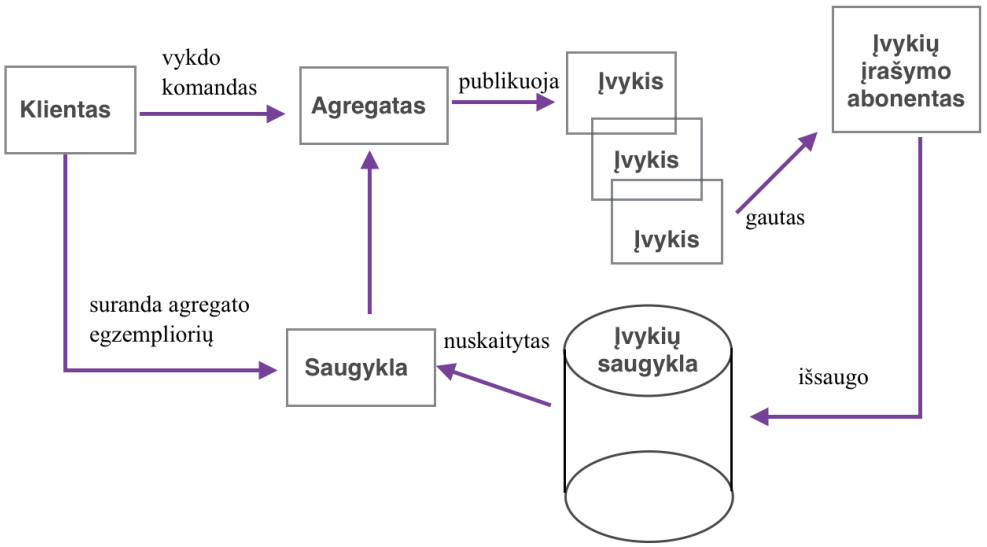
[5] http://en.wikipedia.org/wiki/Domain_model
[6] http://en.wikipedia.org/wiki/Entity
[7] http://en.wikipedia.org/wiki/Metadata
[8] http://www.nongnu.org/cvs/
[9] http://subversion.apache.org/
[10] http://git-scm.com/
[11] http://mercurial.selenic.com/

pačios pirmosios pataisos iki paskutinės. Kai visi išeities failai yra nusiunčiami į pataisų kontrolės sistemą, ši gali stebėti pasikeitimus viso programinės įrangos kūrimo gyvavimo ciklo metu.

Jeigu šis principas būtų pritaikytas vienai esybei, tada vienam agregatui[12] bei galiausiai kiekvienam modelio agregatui, galima suprasti kokią naudą atneša sistemos objektų pasikeitimų stebėjimas:

- Kas būtent nutiko modelyje, jog agregato egzempliorius buvo sukurtas?
- Kas nutiko agregato egzemplioriui bėgant laikui? (Operacijų požiūriu)

Turint visų atliktų operacijų istoriją, galima palaikyti laikinus modelius. Toks kaitos stebėjimas yra įvykių kaupimo principas. 1 diagramoje pateika šio principo aukšto lygio reprezentacija. Agregatai publikuoja įvykius, kurie yra išsaugomi įvykių saugykloje ir naudojami sekti modelio būsenos pasikeitimus. Verta paminėti, jog įvykiai reprezentuoja tam tikrą būsenos pasikeitimą bėgant laikui, todėl jie nėra atnaujinami arba ištrinami. Saugykla nuskaito įvykius iš įvykių saugyklos ir pritaiko juos vieną po kito taip atkurdama agregato būseną.



1 pav. Įvykių kaupimo aukšo lygio reprezentacija

[12] http://martinfowler.com/bliki/DDD_Aggregate.html

2.2.2. Momentinė kopija

Ilgame periode sistemoje susikaupia daugybė įvykių. Atkuriant agregato būseną reikia atkartoti šimtus, tūkstančius ar net milijonus įvykių. Tai tampa šio modelio silpnąja puse, nes įvykių atkartojimas užtrunka vis ilgiau sistemai plečiantis.

Tačiau šio duomenų kamsčio galima išvengti naudojant agregato būsenos momentines kopijas. Tam tikrame įvykių saugyklos istorijos taške yra padaroma agregato būsenos kopija. Serializuota agregato būsena yra įrašoma į įvykių saugyklą. Nuo to momento, agregatas yra atkuriamas pirmiausia naudojantis naujausia jo būsenos momentine kopija ir tik po to atkartojami visi naujesni įvykiai.

Momentinės kopijos nėra atkuriamos atsitiktinai. Jos gali būti kuriamos kas apibrėžtą skaičių įvykių. Šis skaičius turėtų būti parinktas analizuojant domeno sritį bei sistemą ir radus optimalų variantą. Tikėtinai gali būti 50 arba 100 įvykių tarp momentinių kopijų.

2.2.3. Įvykių kaupimo privalumai ir trūkumai

Kaip saugojimo mechanizmas, įvykių kaupimas stipriai skiriasi ir pakeičia ORM[13] įrankį. Kadangi įvykiai dažnai įrašomi kaip dvejetainės reprezentacijos, jie negali būti optimaliai naudojami užklausoms atlikti. Faktiškai įvykių kaupimu pagrįstoms saugykloms tereikia vienos operacijos - gauti įrašus pagal unikalią agregato tapatybę [You10]. To pasekoje užklausom daryti reikia kito kelio. Dažniausiai tam pasirenkamas CQRS[14] principas [BDM+13].

Įvykių kaupimas verčia kitaip mąstyti apie domeno modelį. Įvykių istorija gali padėti surasti bei ištaisyti sistemos defektus bei klaidas [Fit12]. Derinimas naudojant istoriją visų veiksmų, kurie nutiko sistemoje, turi didžiulį pranašumą. Įvykių kaupimas gali vesti prie didelio našumo domeno modelių, tai yra palaikyti ypač didelį skaičių operacijų per sekundę. Pavyzdžiui, įrašymas į vieną duomenų saugyklos lentelę yra ypač greitas. Negana to, tai leidžia CQRS užklausų modelį išplėsti horizontaliai, nes duomenų šaltinio atnaujinimai įvykdomi fone, kai įvykių saugykla yra atnaujinama naujais įvykiais.

[13] http://www.orm.net/
[14] http://martinfowler.com/bliki/CQRS.html

2.2.4. Įvykių kaupimas funkciniame programavime

Vaughn Vernon pateikia keletą pastebėjimų apie įvykių kaupimą funkciniame programavime, kurie gali būti naudingi atliekant projektinius sprendimus bei eksperimentinį tyrimą:

- Agregatas projektuojamas kaip nekintantis būsenos įrašas kartu su funkcijomis, kurios keičia būseną. Šios funkcijos paprasčiausiai priima būsenos įrašą ir įvykių argumentus ir gražina naują būsenos įrašą kaip rezultatą. Tokia funkcija pavaizduota **??** kodo pavyzdyje.

```
Funkcija<Busena, Ivykis, Busena>
```

- Dabartinė agregato būsena gali būti apibrėžta kaip suskleidimas į kairę visų praeities įvykių, kurie yra perduodami būseną keičiančiai funkcijai.
- Agregato metodai gali būti išreikšti kaip kolekcija funkcijų be būsenos.
- Įvykių saugykla gali būti suvokiama bei naudojama kaip funkcinė duomenų bazė, nes ji perduoda argumentus funkcijoms, kurios keičia agregato būseną. Momentinės kopijos įvykių saugykloje primena įsiminimą atmintyje[15] funkciniame programavime.

2.3. Monados

2.3.1. Terminas

Funkciniame programavime monada yra struktūra, kuri atspindi skaičiavimus, apibrėžtus kaip seka žingsnių. Tipas kartu su monados struktūra apibrėžia ką reiškia vykdyti operacijas viena po kitos bei naudoti to pačio tipo įdėtines funkcijas. Tai leidžia kurti komandų grandines, kurios pažingsniui apdoroja informaciją, kur kiekvienas veiksmas yra dekoruojamas naujomis apdorojimo taisyklėmis, kurias apibrėžia monada [OGS08]. Taip pat monada gali būti laikoma kaip funkcinis projektavimo šablonas, skirtas kurti daugybinius tipus [Mer14].

[15] http://en.wikipedia.org/wiki/Memoization

2.3.2. Taisyklės

Anot [Wad95], monadų operacijos turi tenkinti tris taisykles:

- Kairiojo vienetotapatybės (angl. left unit/identity) - suskaičiuojama reikšmė a, rezultatui priskiriamas b ir suskaičiujama n reikšmė. Rezultatas yra toks pat kaip n kai a reikšmė pakeičiama b reikšme. Parodyta 2 paveikslėlio formulėje.

$$unit\ a \star \lambda b.\ n = n[a/b].$$

2 pav. Perkėlimo į kairę taisyklė

- dešiniojo vieneto/tapatybės (angl. right unit/identity) - suskaičiuojama reikšmė m, rezultatas priskiriamas a ir grąžinama a. Rezultatas yra toks pat kaip m. Parodyta 3 paveikslėlio formulėje.

$$m \star \lambda a.\ unit\ a = m.$$

3 pav. Perkėlimo į dešinę taisyklė

- Asociatyvumas - suskaičiuojama reikšmė m, rezultatas priskiriamas a, suskaičiuojamas n, rezultatas priskiriamas b, suskaičiuojama o. Skliaustelių padėtis šiuo atveju yra nesvarbi. Kintamojo a apimtis įtraukia o kairėje, tačiau neįtraukia o dešinėje, todėl ši taisyklė galioja tik tada kai a nėra laisvas nuo o. Parodyta 4 paveikslėlio formulėje.

$$m \star (\lambda a.\ n \star \lambda b.\ o) = (m \star \lambda a.\ n) \star \lambda b.\ o.$$

4 pav. Asociatyvumas

2.3.3. Tolesnis darbas

Kitame mokslinio tyrimo etape ruošiamąsi įrodyti, kad funkcinį-reaktyvų programavimą įmanoma taikyti įvykių kaupimo sistemose. Šiam tikslui pasiekti toliau bus nagrinėjamos monados, jų komponavimo būdai. Kad geriau suprasti monadas turbūt prireiks įgauti žinių kategorijų teorijoje, daugiau žinių apie funkcinį programavimą, pavyzdžiui tipizuotos klasės, parametrizuoti tipai ir t.t.

2.4. Išvados

Literatūros analizės metu remiantis kitų autorių patirtimi:

- išnagrinėtas funkcinis-reaktyvus programavimas,
- išnagrinėtas įvykių kaupimo principas,
- išnagrinėti įvykių srautai bei operacijos su jais,
- susipažinta su įvykių kaupimu funkciniame programavime,
- susipažinta su monadomis,
- įvaldyta sąvokų sistema, susijusi su nagrinėjama tematika.

3. Sąvokų apibrėžimai

- **Agregatas** (angl. aggregate) - DDD modelis, rinkinys domeno objektų, kurie gali būti laikomi kaip visuma.
- **Derinimas** (angl. debugging) - riktų bei klaidų paieška programinėje įrangoje bei jų taisymas.
- **Esybė** (angl. entity) - kažkas, kas egzistuoja pats savaime, faktiškai arba hipotetiškai.
- **Horizontalus išplečiamumas** (angl. horizontal 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.
- **Metaduomenys** (angl. metadata) - duomenys apie kitus duomenis.

4. Santrumpos ir paaiškinimai

- **CQRS** (angl. Command Query Responsibility Segregation) – komandų-užklausų atsakomybių atskyrimas.
- **DDD** (angl. Domain-Driven Design) – būdas kurti programinę įrangą, skirtą spręsti sudėtingus uždavinius, bei apjungti realizaciją kartu su augančiu domeno modeliu.
- **NoSQL** – duomenų bazė, skirta architektūriniams modeliams, kuriems nereikia palaikyti stiprios darnos principo, kuris naudojamas reliacinėse duomenų bazėse. Tai įgalina horizontalų išplečiamumą bei aukštesnį prieinamumą.
- **ORM** (angl. Object-Relational Mapping) – programavimo technika duomenų konvertavimui tarp nesuderinamų sistemų tipų, naudojama objektinio programavimo kalbose. Pavyzdys: JAVA programavimo kalba bei Oracle duomenų bazė.

Literatūros sąrašas

[Ams] Edward Amsden. A survey of functional reactive programming. *Rochester Institute of Technology.*

[Bas] Tim Bass. Mythbusters: Event Stream Processing Versus Complex Event Processing. page 1. Invited talk.

[BDM+13] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure.* Microsoft patterns & practices, 1st edition, 2013.

[Cza12] Evan Czaplicki. Elm: Concurrent frp for functional guis. Master's thesis, 30 March 2012.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[Fit12] Sean Fitzgerald. A pattern for state machine persistence using event sourcing, cqrs and a visual workbench. Master's thesis, 21 March 2012.

[KW92] David J. King and Philip Wadler. Combining monads. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 134–143. Springer, 1992.

[Mer14] L.G. Meredith. *Pro Scala: Monadic Design Patterns for the Web.* Artima, 2014.

[MO12] Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM.

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell.* O'Reilly Media, Inc., 1st edition, 2008.

[OSG04] OSGi Alliance. Listener Pattern Considered Harmful: The "Whiteboard"Pattern. Technical whitepaper, OSGi, 2004.

[Ver13] V. Vernon. *Implementing Domain-Driven Design*. Pearson Education, 2013.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag.

[You10] Greg Young. Cqrs documents by greg young. Technical whitepaper, 2010.