

VILNIAUS UNIVERSITETAS  
MATEMATIKOS IR INFORMATIKOS FAKULTETAS  
PROGRAMŲ SISTEMŲ BAKALAURO STUDIJŲ PROGRAMA

## **Kodo skirstymo į paketus šablonų tyrimas**

### **Analysis of code packaging patterns**

Bakalauro baigiamasis darbas

Atliko:	Martyna Ubartaitė
Darbo vadovas:	Gediminas Rimša
Darbo recenzentas:	doc. dr. Audronė Lupeikienė

Vilnius – 2024

## Santrauka

Gera suprojektuota kompiuterinė sistema yra vienas iš kritinių sėkmingo ją naudojančio verslo elementų. Mąstant apie programinio kodo projektavimą, kodo paketų kūrimas, klasių priskyrimas jiems ir paketų hierarchijos sudarymas paprastai nėra pagrindinis prioritetas, tačiau tai parodo praleistą galimybę padaryti sistemos struktūrą labiau stabilia, suprantama ir lengviau palaikoma. Rekomenduojamas naudoti kodo skirstymo į paketus pagal dalykinę sritį metodas negali būti vienareikšmiškai pritaikomas kiekvienoje situacijoje ir dažniausiai praktikoje jo nėra griežtai laikomasi – klasės būna išskaidytos remiantis papildomomis taisyklėmis, siekiant išspręsti sistemos planavimo metu kylančias problemas. Darbo metu išskirti planavimo metu kylančias problemas sprendžiantys kodo skirstymo į paketus šablonai, aprašyti sistemos kokybę vertinantys matai. Pagal rastus šablonus pertvarkytos atviro kodo sistemos, įvertinti pertvarkytų sistemų kokybės matai. Darbo rezultatuose išskirti šablonai galintys papildyti kodo skirstymo į paketus pagal dalykinę sritį metodą ir spręsti projektavimo metu kylančias problemas.

**Raktiniai žodžiai:** Kodo skirstymas į paketus, Kodo skirstymo į paketus šablonai, Java, Paketai, Šablonai

## Summary

Creating packages and package hierarchy is usually not a top priority in software design, but a correct approach to code packaging can make the system more readable, maintainable and improve the overall quality. The approach to divide code into packages by functionality may not be applicable in every situation. In an attempt to solve these issues, multiple code packaging patterns are used in real-world systems. This paper attempts to identify such issues together with patterns which solve them. To achieve this, a set of metrics evaluating readability, maintainability and stability were proposed. Identified patterns were verified against chosen metrics to evaluate their impact.

**Keywords:** Code packaging, Code packaging patterns, Java, Packaging, Patterns

# Turinys

ĮVADAS .....	5
1. KODO SKIRSTYMO Į PAKETUS ŠABLONAI .....	7
1.1. Šablonų paieškos procesas .....	7
1.2. Sistemos planavimo metu kylančios problemos .....	7
1.2.1. Pagalbinių, daugkartinio naudojimo klasių skirstymas .....	7
1.2.2. Didelis klasių skaičius pakete .....	10
1.2.3. Skirtingų sąsajų implementacijų skirstymas .....	12
1.2.4. Esybių versijavimo valdymas .....	14
1.3. Kodo skirstymo į paketus šablonų paieškos išvados .....	14
2. KODO SKIRSTYMO Į PAKETUS ŠABLONŲ ANALIZĖ .....	17
2.0.1. Pagalbinių, daugkartinio naudojimo klasių skirstymo šablonų analizė .....	17
2.0.2. Didelio klasių skaičius pakete skirstymo šablonų analizė .....	17
2.0.3. Skirtingų sąsajų implementacijų skirstymo šablonų analizė .....	19
2.0.4. Esybių pokyčių ir versijavimo skirstymo šablonų analizė .....	20
3. KOMPIUTERINĖS SISTEMOS VERTINIMAS .....	21
3.1. Kompiuterinės sistemos kokybė .....	21
3.2. Kodo skirstymo į paketus šablonų vertinimas .....	21
3.2.1. Bendro sąryšio principas .....	22
3.2.2. Aciklinių priklausomybių principas .....	22
3.2.3. Stabilių priklausomybių principas .....	22
3.2.4. Stabilių abstrakcijų principas .....	23
3.3. Paketų kokybės matai .....	23
4. ĮRANKIAI ŠABLONŲ ANALIZEI IR ĮVERTINIMUI .....	25
4.1. Reikalavimai įrankiams .....	25
4.2. Reikalavimai bendrinės sistemos analizės įrankiui .....	25
4.3. Reikalavimai įrankiui paketo kokybei skaičiuoti .....	25
4.4. Įrankių įgyvendinimas .....	26
5. ESAMŲ SISTEMŲ PERTVARKYMAS .....	28
5.1. Sistemų pasirinkimas .....	28
5.2. <i>crawler4j</i> sistema .....	28
5.3. <i>azure-sdk-for-java</i> sistema .....	35
REZULTATAI .....	39
IŠVADOS .....	40
ŠALTINIAI .....	41
PRIEDAI .....	42
1 priedas. Nuorodos į nagrinėtas atviro kodo sistemas .....	42

## Įvadas

Gera suprojektuota kompiuterinė sistema yra vienas iš kritinių sėkmingo ją naudojančio verslo elementų. Tam, jog kompiuterines sistemas naudojantis verslas išlaikytų stabilų augimą, jo sistema turėtų kiek įmanoma sumažinti atotrūkį tarp organizacijos tikslų ir jų įgyvendinimo galimybių. Mąstant apie programinio kodo projektavimą, paketų kūrimas, klasių priskyrimas jiems ir paketų hierarchijos sudarymas paprastai nėra pagrindinis prioritetas, tačiau tai parodo praleistą galimybę padaryti sistemos struktūrą labiau patikima, suprantama ir lengviau palaikoma. Modernios kompiuterinės sistemos yra didžiulės, programinis kodas yra padalintas į daugybę failų, kurie yra išskaidyti per skirtingo gylio direktorijas, todėl apgalvotai išskirstytas programinis kodas daro didesnę įtaką bendram sistemos suprantamumui bei palaikomumui, nei gali atrodyti iš pirmo žvilgsnio. Sistemos paketų struktūros analizė norint įvertinti programinės įrangos kokybę tampa vis svarbesne tema dėl augančio failų ir paketų skaičiaus [Eli10].

Diskusijose, kaip reikėtų skirstyti programinį kodą, paprastai akcentuojami du metodai – pagal *techninį funkcionalumą*, kur kiekvienai techninei funkcijai arba kompiuterinės sistemos sluoksniui yra sukuriamas paketas, arba pagal *dalykinę sritį*, kur vienos esybės kodas, dalykinės srities esybės funkcionalumas skirtingose programiniuose sluoksniuose yra patalpintas viename pakete [Jan21]. Dažniausiai, prioritetas teikiamas kodo skirstymui pagal dalykinę sritį [Eva03] dėl lengvesnio skaitomumo, palaikomumo, mažesnės sankibos [Sad21]. Skirstymas pagal dalykinę sritį taip pat gali padidinti inkapsuliaciją naudojant prieigos modifikatorius *Java* programavimo kalboje.

Tačiau šis metodas negali būti vienareikšmiškai pritaikomas kiekvienoje situacijoje ir dažniausiai praktikoje jo nėra griežtai laikomasi – klasės būna išskirstytos remiantis papildomomis taisyklėmis, siekiant išspręsti sistemos planavimo metu kylančias problemas. Kadangi nėra aiškiai apibrėžtų gairių, kaip skirstymo metodas gali būti pritaikomas kilus sudėtingesnei situacijai, praktikoje pasitaiko skirtingų jų sprendimo būdų, galinčių neigiamai paveikti sistemos kokybę. Norint išsiaiškinti, koks programinio kodo skirstymo būdas tinkamiausias tokiose situacijose, tam jog jo struktūra darytų teigiamą įtaką sistemai, reikalinga atlikti skirstymo į paketus šablonų analizę – išsiaiškinti galimus šablonus, naudojamus kylančių projektavimo problemų sprendimui, turėti aiškius šablonų apibrėžimus su jų privalumais bei trūkumais. Šio darbo metu nagrinėjami ir aprašomi gerai įgyvendinto kodo požymiai, užtikrinantys sistemos stabilumą ir palaikomumą, remiantis Martin Kleppmann *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, ir Robert C. Martin *Agile Software Development, Principles, Patterns, and Practices* knygomis. Šiame darbe minint *kodo skirstymo į paketus šabloną* turima omenyje taisyklės, nurodančias, kaip grupuoti klases į paketus, sprendžiant iškilusią problemą bei užtikrinant nuoseklų stilių.

Šio darbo tikslas – identifikuoti ir įvertinti praktikoje naudojamus šablonus kodo skirstymui į paketus.

Tikslui pasiekti yra iškeliama šie uždaviniai:

- Išskirti gerai įgyvendinto kodo požymius.
- Aprašyti skirstymo į paketus šablonus, remiantis praktikoje sutinkamais pavyzdžiais.

- Pasiūlyti kriterijus, įvertinančius kodo skirstymo šablono įtaką sistemos kokybei, remiantis rastais gerai įgyvendintos sistemos požymiais.
- Pasirinkti kelias sistemas ir pertvarkyti jų failų struktūrą pagal aprašytus šablonus, įvertinant, kiek sudėtinga pasiekti kiekvieno šablono struktūrą.
- Naudojant pertvarkytas sistemas, įvertinti kiekvieną šabloną kodo skirstymui į paketus pagal pasiūlytus kriterijus.
- Pateikti rekomendacijas, ar aprašyti šablonai kodo skirstymui tinkami naudoti.

# 1. Kodo skirstymo į paketus šablonai

Renkantis kodo skirstymo į paketus metodą, dažniausiai rekomenduojamas skirstymas pagal dalykinę sritį [Eva03]. Tokį skirstymą galima apibrėžti kaip paketų sudarymą iš klasių ar sąsajų, kurios tarpusavyje yra konceptualiai susijusios [Var12]. Šis skirstymo būdas dažniausiai bent jau iš dalies naudojamas net sistemose, kurios neturi aiškiai apibrėžto skirstymo būdo. Tačiau šis metodas negali būti vienareikšmiškai pritaikomas kiekvienoje situacijoje. Tai lemia, kad praktikoje jo nėra griežtai laikomasi – klasės būna išskaidytos remiantis papildomomis taisyklėmis, siekiant išspręsti sistemos planavimo metu kylančias problemas. Norint išskirti šias papildomas taisykles, šablonus, nagrinėjamos atviro kodo sistemos, stebimi nukrypimai nuo bendresnio kodo skirstymo būdo ir identifikuojami klausimai arba problemos, kurias jais buvo bandoma išspręsti.

## 1.1. Šablonų paieškos procesas

Praktikoje naudojamų kodo skirstymo į paketus šablonų paieška buvo atliekama stebint atviro kodo sistemų struktūrą ir bandant nustatyti, kodėl pasirenkama nukrypti nuo pagrindinio skirstymo būdo. Nagrinėjamos dvidešimt atviro kodo sistemų iš *trending GitHub* repozitorijų sąrašo, taip pat remtasi atviro kodo repozitorijų sąrašu<sup>1</sup>. Pasirenkami skirtingo tipo projektai, siekiant objektyvesnės šablonų analizės skirtingose srityse. Išskirti tipai:

- Taikomoji programinė įranga, teikianti paslaugas įrangos naudotojams. Pavyzdžiui, internetinė programėlė priminimams ir darbams užsirašyti
- Techninė programinė įranga, naudojama taikomosios programinės įrangos duomenų saugojimui, siuntimui, paieškai. Pavyzdžiui, duomenų bazės, pranešimų eilės, talpyklos (angl. cache)
- Programinės įrangos įrankiai, skirti naudoti kitose sistemose supaprastinant programinį kodą, naudojant jau įgyvendintas funkcijas. Pavyzdžiui, Java programavimo kalbos Spring karkasas internetinių programėlių kūrimui

Pastebėta, kad nukrypimai nuo skirstymo pagal dalykinę sritį – pagrindinio skirstymo būdo, dažnai pasikartodavo bandant išspręsti tas pačias problemas, tačiau jų sprendimo būdai buvo skirtingi.

## 1.2. Sistemos planavimo metu kylančios problemos

Šiame skyriuje pateikiamos analizuotose sistemose rasti kodo skirstymo į paketus šablonai, kartu su jais sprendžiamomis problemomis ir pavyzdžiais, kokiose sistemose jie yra. Pateiktas šablonų ir problemų sąrašas nėra baigtinis, tačiau minimi šablonai pasikartoja ne vienoje sistemoje.

### 1.2.1. Pagalbinių, daugkartinio naudojimo klasių skirstymas

Pagalbinės klasės galėtų būti apibrėžiamos kaip tiesiogiai su dalykine sritimi nesusijusios, techninių funkcijų klasės, pavyzdžiui, atsakingos už masyvų filtravimą, darbą su failais. Pagalbinės ir daugkartinio naudojimo klasės dažniausiai negali būti patogiai grupuojamos skirstant pagal

---

<sup>1</sup><https://github.com/akullpp/awesome-java>

dalykinę sritį – pagalbinės klasės gali naudoti kelios esybės, todėl neaišku, prie kurios jas reikėtų priskirti. Vienas iš šablonų, sprendžiančių šią problemą – turėti vieną paketą, skirtą visoms pagalbinėms klasėms. Pakete galima turėti atskiras klases kiekvienai bendrinei dalykinei sričiai, iš kurios pavadinimo programuotojas galėtų nuspresti, kad jo ieškomas funkcionalumas bus būtent toje klasėje (pavyzdys 1).

```
/
├── users
│   └── UserRolesGrouping
├── ...
├── payments
│   └── PaymentScheduler
├── ...
├── transactions
│   └── TransactionBatching
├── utils
│   ├── Arrays
│   ├── Maps
│   ├── SqlQueries
│   └── Users
```

1 pav. Sistemos pavyzdys, kur visas bendrinio panaudojimo kodas guli *utils* pakete, pirmame sistemos paketų lygyje

Jei pagalbinių klasių dydis labai išauga, galima vietoje vienos atskiros klasės vienai bendrinei sričiai sukurti vieną paketą, ir jame turėti kelias pagalbinės klases, susijusias su ta dalykine sritimi (pavyzdys 2).

```
/common
├── arrays
│   ├── ArrayFilters
│   └── ArrayComparators
├── ...
├── maps
│   ├── MapTransformations
│   └── MapJoining
├── json
│   └── JsonParser
├── ...
├── database
│   ├── DatabaseConnection
│   └── DatabaseQueries
```

2 pav. Sistemos pavyzdys, kur bendrinio panaudojimo kodas guli *common* pakete, po dalykinės srities subpaketais, taip sumažinant klasių dydį

Tokį pagalbinių klasių skirstymo metodą naudoja keletas repozitorijų – pavyzdžiui, užrašinės



aplikacijos *Omni-Notes*<sup>2</sup> bei *Fire Sticker*<sup>3</sup>. *Omni-Notes* atveju, *utils* pakete esančios klasės papildomai skirstomos pagal atitinkamą dalykinę sritį. Šio šablono aprašymas (3)

Šablonas	Atskiras pagalbinių klasių paketas
Sprendžiama problema	Pagalbinių klasių skirstymas
Siekiamybė	Mažesnis kodo dubliavimas, lengvai randamos pagalbinės klasės
Siūlomas sprendimas	Sukurti atskirą pagalbinių klasių paketą dalykinės srities esybių lygmenyje
Galimos variacijos	Gali būti papildomai sukuriami subpaketai pagal pagalbinių klasių teikiamą funkcionalumą

3 pav. Atskiros pagalbinių klasių paketo šablonas

Šiai problemai spręsti galimas ir kitoks būdas – turėti keletą pagalbinių klasių paketų, patalpintų po atitinkamu bendresniu dalykinės srities paketu. Tokį skirstymo į paketus šabloną naudoja transakcijoms skirta programinė įranga *Seata*<sup>4</sup>. Čia pagalbinės klasės yra *utils* pakete žemesniame sistemos lygmenyje (pavyzdys 4), pavyzdžiui, po *engine* paketu.

```

/engine
├── strategy
├── sequence
├── ...
├── utils
│   ├── ExceptionUtils
│   └── ...

```

4 pav. Sistemos pavyzdys, kur bendrinio panaudojimo kodas guli *utils* pakete esančiame po dalykinės srities subpaketais

Šio šablono aprašymas (5)

<sup>2</sup><https://github.com/federicoiosue/Omni-Notes/tree/develop/omniNotes/src/main/java/it/feio/android/omninotes/utils>

<sup>3</sup>[https://github.com/hackjutsu/Fire\\_Sticker/tree/master/app/src/main/java/com/gogocosmo/cosmoqiu/fire\\_sticker/Utils](https://github.com/hackjutsu/Fire_Sticker/tree/master/app/src/main/java/com/gogocosmo/cosmoqiu/fire_sticker/Utils)

<sup>4</sup><https://github.com/apache/incubator-seata/tree/2.x/saga/seata-saga-engine/src/main/java/org/apache/seata/saga/engine/utils>

Šablonas	Pagalbinių klasių priskyrimas dalykinės srities paketui
Sprendžiama problema	Pagalbinių klasių skirstymas
Siekiamybė	Kiekvienai esybei priskirtos pagalbinės klasės
Siūlomas sprendimas	Klases laikyti <i>util</i> paketuose, žemesniame lygmenyje nei dalykinės srities esybės

5 pav. *Pagalbinių klasių priskyrimo dalykinės srities paketui šablonas*

### 1.2.2. Didelis klasių skaičius pakete

Sistamai plečiantis, neišvengiamai didėja ir klasių skaičius. Net pasirinkus klasių skirstymo į paketus metodą ir palaikant šią tvarką, klasių skaičius pakete gali išaugti. Tokį pavyzdį galima matyti paieškos įrankio *Elasticsearch*<sup>5</sup> *transport* pakete. Nors klasės yra sugrupuotos pagal dalykinę sritį, bendras jų skaičius pakete siekia beveik devyniasdešimt. Tokiame pakete naviguoti sunku, jis taip pat gali turėti daug priklausomybių. Jei tokia tendencija yra būdinga visai sistemai, sistema tampa mažiau lanksti pokyčiams, kadangi net ir paprastas pakeitimas daro įtaką reišmingai sistemos daliai, pokyčiai yra labiau linkę keisti bendrą sistemos architektūrą. Taip pat naujos sistemos versijos išleidimo (angl. *release*) procesas tampa sudėtingesnis, kadangi yra paveikiama daugiau klasių. Dažnai matoma, kad susidarius šiai situacijai, nukrypstama nuo skirstymo pagal dalykinę sritį, pradedamos grupuoti techninio funkcionalumo klasės. Tokį skirstymą galima matyti *DBeaver* duomenų bazės įrankyje<sup>6</sup> – aukštesniame lygmenyje grupuojant pagal dalykinę sritį, pavyzdžiui, *MsSql*, vėliau pereinama prie skirstymo pagal techninį funkcionalumą pakete *model*. Šio šablono aprašymas (6)

Šablonas	Žemesnio lygio paketų sudarymas grupuojant pagal techninį funkcionalumą
Sprendžiama problema	Didelis klasių skaičius pakete
Siekiamybė	Išskirti dalykinės srities esybės techninius funkcionalumus
Siūlomas sprendimas	Dalykinės srities esybių klases skirstyti pagal techninį funkcionalumą

6 pav. *Žemesnio lygio paketų sudarymas grupuojant pagal techninį funkcionalumą šablonas*

Kitą skirstymo būdą galima stebėti *IntelliJ Source Wizard* įskiepio kode<sup>7</sup>. Čia *plugin* paketo esybė yra papildomai išskirstyta pagal *IntelliJ* sistemos dalykinės srities esybes. Vadovaujantis šiuo

<sup>5</sup><https://github.com/elastic/elasticsearch/tree/main/server/src/main/java/org/elasticsearch/transport>

<sup>6</sup><https://github.com/dbeaver/dbeaver/tree/devel/plugins/org.jkiss.dbeaver.ext.mssql/src/org/jkiss/dbeaver/ext/mssql>

<sup>7</sup><https://github.com/wix-incubator/source-wizard/tree/master/src/main/kotlin/plugin>

šablonu, klasės skirstomos pagal smulkesnį jų teikiamą funkcionalumą – *actions*, *extensions*, *ui*, šiame kontekste yra ne techniniai sluoksniai, o *Intellij* įskiepio dalykinės srities dalis (pavyzdys 7).

```
/plugin
├─ actions
├─ extensions
└─ ui
```

7 pav. *Source Wizard* sistemos dalis, kurioje matomi smulkiau išskirstyti moduliai

Šio šablono variacija – smulkūs moduliai su atskirose klasėse aprašytais kontraktais. Funkcionalumas kitiems paketams galėtų būti pasiekiamas per vieną minimalią sąsają (angl. *interface*), kuri atskleidžia tik konceptus (metodus arba duomenų tipus), kurie yra glaudžiai susiję su komponento teikiama paslauga, bei klase, grąžinančią minėtos sąsajos įgyvendinimą. Tai gali būti paprasta klasė, su statine funkcija, kurios rezultatas yra ši sąsaja, arba, esant keliomis sąsajos implementacijomis, *Static factory* dizaino šablonas, kuris nusprendžia, kurį įgyvendinimą reikėtų grąžinti pagal paduotus argumentus. Visos kitos klasės naudoja *package* pasiekiamumo modifikatorių, taip kompiliatoriaus pagalba užtikrinant, kad jos nebus pasiektos iš išorės. Toks principas yra sutinkamas *Typescript* programavimo kalboje, kur modulis (šios kalbos paketo atitikmuo) gali turėti *index.ts* failą, veikiantį kaip sąsaja, apibrėžianti, kokios modulio klasės bei funkcijos gali būti pasiekiamos už modulio ribų. Toki šabloną naudoja turinio valdymo sistema *Keystone*<sup>8</sup>, kur kiekviena esybė turi savo modulį, atliekantį vieną funkciją, kurios kontraktas aprašytas *index.ts* faile (pavyzdys 8).

```
/types
├─ image
│   ├── views
│   ├── utils
│   └─ index
├─ text
│   ├── views
│   └─ index
├─ ...
└─ checkbox
    ├── views
    └─ index
```

8 pav. *Keystone* sistemos dalis, kurioje matomi smulkūs moduliai su *index.ts* failuose aprašytais kontraktais

Šio šablono aprašymas (9)

---

<sup>8</sup><https://github.com/keystonejs/keystone/tree/main/packages/core/src/fields/types>

Šablonas	Skirstymas pagal smulkų funkcionalumą
Sprendžiama problema	Didelis klasių skaičius pakete
Siekiamybė	Maži paketai, turintys aiškiai apibrėžtą funkcionalumą
Siūlomas sprendimas	Klases suskirstyti pagal smulkesnį jų teikiamą funkcionalumą
Galimos variacijos	Smulkūs moduliai su atskirose klasėse aprašytais kontraktais

9 pav. Skirstymo pagal smulkų funkcionalumą šablonas

### 1.2.3. Skirtingų sąsajų implementacijų skirstymas

Sistemai plečiantis galima susidurti su problema, kad išauga sąsajos įgyvendinimų skaičius. Ši problema aktuali skirstant klases pagal dalykinę sritį – laikant sąsajų įgyvendinimus skirtinguose esybių paketuose (kai sąsaja nėra susijusi su viena esybe, pavyzdžiui, ją įgyvendina kelios esybės), navigacija paketuose pasidaro sudėtinga, neaišku, kas kurią klasę įgyvendina. Galimas šios problemos sprendimas – sukurti sąsajos implementacijas tame pačiame pakete kaip ir pati sąsaja. Toks skirstymo į paketus šablonas yra sutinkamas *Java* standartinėje bibliotekoje<sup>9</sup>, kur visos standartinės sąsajos bei jų implementacijos yra tame pačiame pakete module (pavyzdys 10).

```

/util
├── Map
│   ├── HashMap
│   ├── LinkedHashMap
│   ├── WeakHashMap
│   └── TreeMap
├── ...
├── List
│   ├── LinkedList
│   └── ArrayList

```

10 pav. *Java* standartinės bibliotekos pavyzdys, kuriame sąsajų įgyvendinimai guli tame pačiame pakete, kaip ir sąsajos

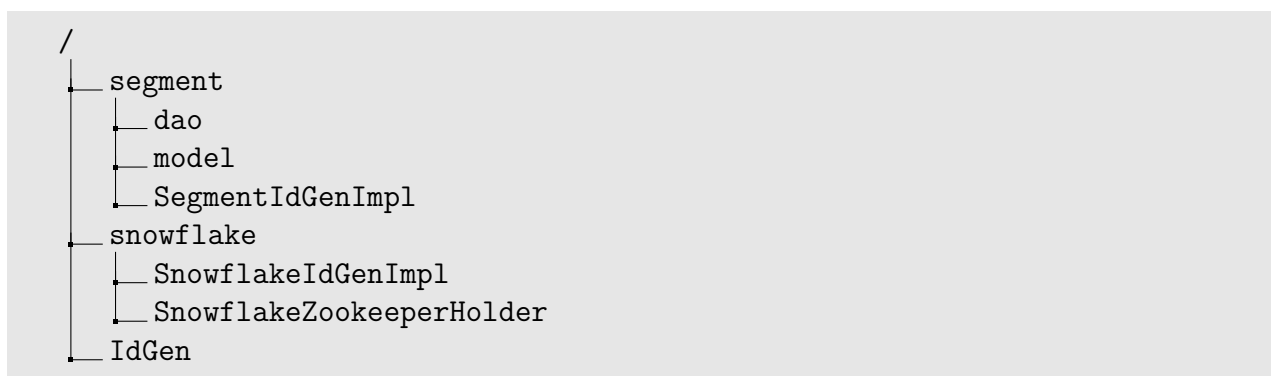
Šio šablono aprašymas (11)

<sup>9</sup><https://github.com/AdoptOpenJDK/openjdk-jdk11/tree/master/src/java.base/share/classes/java/util>

Šablonas	Sąsajų ir implementacijų grupavimas
Sprendžiama problema	Daug skirtingų sąsajų implementacijų
Siekiamybė	Susieti sąsają ir jos implementacijas lengvesnei jų paieškai
Siūlomas sprendimas	Sąsajų įgyvendinimus laikyti tame pačiame pakete, kaip ir sąsajas

11 pav. Sąsajų ir implementacijų grupavimo šablonas

Kitas galimas sprendimo būdas – sąsaja, kuri turi daug skirtingų įgyvendinimų, turėtų turėti specifiškai jai sukurtą paketą, o kiekvienam paketo įgyvendinimui sukuriamas po subpaketą, kuriame yra tiek klasę įgyvendinanti sąsaja, tiek kitos, įgyvendinimui reikalingos klasės. Tokia paketų struktūra buvo sutikta tokiose sistemose kaip *HikariCP*<sup>10</sup>, skirtoje efektyviam *Java* programos prisijungimui prie duomenų bazių, bei *Leaf*<sup>11</sup> (pavyzdys 12), naudojamoje unikalaus identifikatoriaus generavimui.



12 pav. *Leaf* sistema, kur kiekvienas sąsajos įgyvendinimas yra aprašytas subpakuete

Šio šablono aprašymas (13)

Šablonas	Įgyvendinimų atskyrimas
Sprendžiama problema	Daug skirtingų sąsajų implementacijų
Siekiamybė	Skirtingų sąsajų bei implementacijų atskyrimas
Siūlomas sprendimas	Kiekvienam paketo įgyvendinimui sukurti subpaketą, kuriame yra tiek klasę įgyvendinanti sąsaja, tiek kitos, įgyvendinimui reikalingos klasės

13 pav. Įgyvendinimų atskyrimo šablonas

<sup>10</sup><https://github.com/brettwooldridge/HikariCP/tree/dev/src/main/java/com/zaxxer/hikari>

<sup>11</sup><https://github.com/Meituan-Dianping/Leaf/tree/master/leaf-core/src/main/java/com/sankuai/inf/leaf>

#### 1.2.4. Esybių versijavimo valdymas

Sistamai egzistuojant ilgesnį laiką, jos pokyčiai pasidaro neišvengiami. Smulkūs pokyčiai nebūtinai paveikia bendrą sistemos struktūrą, tačiau didesniems pokyčiams kartais reikia sukurti naujas klasių ar funkcionalumų versijas. Jei reikia palaikyti atgalinį suderinamumą (angl. backward compatibility), sistemoje gali atsirasti kelios tų pačių esybių versijos. Tokiu atveju sistemoje egzistuojančios kelios tų pačių esybių versijos gali pridėti painumo. Ši problema gali būti sprendžiama sukuriant atskirus paketus skirtingoms versijoms (v1, v2, v3, ...) bei iškeliant bendrą, abiejų esybių naudojamą kodą į paketus taip, kad jas galėtų pasiekti abi versijos.

```
/regex
├── common
├── pending
├── v3
├── v4
└── ...
```

14 pav. *Mongo* duomenų bazės pavyzdys, kuriame skirtingos versijos patalpintos atskiruose paketuose

Toks skirstymo būdas matomas keliose repozitorijose – *Mongo*<sup>12</sup> duomenų bazėje (pavyzdys 14), API skirtame kelionių valdymui *travels-java-api*<sup>13</sup> bei duomenų saugojimo įrankyje *nocodb*<sup>14</sup>. Šiose repozitorijose v1, v2 (*Mongo* atveju – v4, v5) pavadinti paketai saugo skirtingas esybių versijas. Šio šablono aprašymas (15)

Šablonas	Atskirų versijų skirstymas
Identifikatorius	<i>versijos-1</i>
Sprendžiama problema	Esybių pokyčių ir versijavimo valdymas
Siekiamybė	Palaikyti skirtingas esybių versijas
Siūlomas sprendimas	Sukurti atskirus paketus skirtingoms versijoms bei iškelti bendrą, abiejų esybių naudojamą kodą į paketus

15 pav. *Atskirų versijų skirstymo* šablonas

### 1.3. Kodo skirstymo į paketus šablonų paieškos išvados

Išanalizavus dividešimt pasirinktų sistemų matoma, kad kodo skirstymo į paketus šablonai bei jų sprendžiamos problemos pasikartoja – išskirtus pagalbinių, daugkartinio naudojimo klasių

<sup>12</sup>[https://github.com/mongodb/mongo/tree/master/src/third\\_party/boost/boost/regex](https://github.com/mongodb/mongo/tree/master/src/third_party/boost/boost/regex)

<sup>13</sup><https://github.com/mariazevedo88/travels-java-api/tree/master/src/main/java/io/github/mariazevedo88/travelsjavaapi/controller>

<sup>14</sup><https://github.com/nocodb/nocodb/tree/c7cc1f92fd77f8b5daefceb7148aab4a69cb9b4e/packages/nocodb/src/meta/migrations>

skirstymo šablonus naudoja aštuoniolika sistemų, didelio klasių skaičiaus pakete šablonus – devynios, skirtingų sąsajų implementacijų skirstymo šablonus – dešimt, o esybių versijavimo valdymo – penkios. Tai parodo, kad šios problemos sistemose yra aktualios.

Lentelėje 16 pateiktas nagrinėtų sistemų sąrašas, bei nurodyta, ar sistemos įgyvendina aprašytus šablonus. Ši lentelė padeda suprasti šablonų pasikartojimą ribotos apimties sistemų imtyje.

Šablonų pavadinimai lentelės antraštėje yra sutrumpinti siekiant sutalpinti lentelę į dokumento puslapį, todėl pateikiamas lentelės antraštės reikšmių aprašas:

- *P1 – atskiro pagalbinių klasių paketo* šablonas (aprašytas pavyzdyje 3)
- *P2 – pagalbinių klasių priskyrimo dalykinės srities paketui* šablonas (aprašytas pavyzdyje 5)
- *D1 – žemesnio lygio paketų sudarymo grupuojant pagal techninį funkcionalumą* šablonas (aprašytas pavyzdyje 6)
- *D2 – skirstymo pagal smulkų funkcionalumą* šablonas (aprašytas pavyzdyje 9)
- *S1 – sąsajų ir implementacijų grupavimo* šablonas (aprašytas pavyzdyje 11)
- *S2 – įgyvendinimų atskyrimo* šablonas (aprašytas pavyzdyje 13)
- *V – atskirų versijų skirstymo* šablonas (aprašytas pavyzdyje 15)

Sistema	Sistemos tipas	$P1$	$P2$	$D1$	$D2$	$S1$	$S2$	$V$
HikariCP	Įrankis	Taip	Ne	Ne	Taip	Ne	Taip	Ne
H2	Techninis	Taip	Ne	Ne	Ne	Taip	Ne	Ne
Leaf	Techninis	Taip	Ne	Ne	Taip	Ne	Taip	Ne
crawler4j	Taikomasis	Taip	Ne	Ne	Ne	Ne	Ne	Ne
Chronicle-Map	Techninis	Ne	Ne	Taip	Ne	Taip	Ne	Ne
RxJava	Įrankis	Ne	Ne	Ne	Ne	Taip	Ne	Ne
guice	Įrankis	Taip	Taip	Ne	Ne	Taip	Ne	Ne
azure-storage	Įrankis	Ne	Ne	Ne	Ne	Ne	Ne	Ne
elasticsearch	Techninis	Taip	Taip	Ne	Ne	Ne	Ne	Ne
incubator-seata	Techninis	Ne	Taip	Ne	Ne	Ne	Ne	Taip
mongo	Techninis	Taip	Ne	Ne	Ne	Ne	Ne	Taip
nocodb	Taikomasis	Taip	Ne	Ne	Taip	Ne	Ne	Taip
travels-java-api	Taikomasis	Taip	Ne	Ne	Taip	Ne	Taip	Taip
openjdk-jdk11	Techninis	Taip	Ne	Ne	Ne	Taip	Ne	Taip
keystone	Taikomasis	Taip	Taip	Ne	Taip	Ne	Ne	Ne
source-wizard	Taikomasis	Ne	Ne	Ne	Taip	Ne	Taip	Ne
dbeaver	Techninis	Ne	Taip	Taip	Taip	Ne	Ne	Ne
Fire Sticker	Taikomasis	Taip	Ne	Ne	Taip	Ne	Ne	Ne
Omni-Notes	Taikomasis	Taip	Ne	Ne	Ne	Ne	Ne	Ne
Apache Cassandra	Techninis	Taip	Ne	Ne	Ne	Ne	Taip	Ne
Bendras sistemų skaičius		$\Sigma P1$	$\Sigma P2$	$\Sigma D1$	$\Sigma D2$	$\Sigma S1$	$\Sigma S2$	$\Sigma V$
19		13	5	2	8	5	4	5

16 pav. Nagrinėtų sistemų sąrašas ir jų šablonai



## 2. Kodo skirstymo į paketus šablonų analizė

Atviro kodo analizės metu buvo išskirti septyni kodo skirstymo į paketus šablonai. Norint įvertinti, kuris šablonas duotai problemai spręsti yra tinkamiausias, reikalinga nagrinėti, kaip siūlomas sprendimas sprendžia atitinkamą problemą ir kokias pasekmes gali turėti jo taikymas.

### 2.0.1. Pagalbinių, daugkartinio naudojimo klasių skirstymo šablonų analizė

Pagalbinių klasių skirstymui buvo išskirti du šablonai – kiekvienai esybei priskirtos pagalbinės klasės bei atskiras pagalbinių klasių paketas. Svarbi problema, susijusi su pagalbinių klasių – su sistema dirbantys inžinieriai nežino apie jų egzistavimą, todėl jų nenaudoja, tai veda prie didesnio kodo pasikartojimo arba kelių skirtingų to paties pagalbinių funkcionalumo įgyvendinimų [Ale23]. Šią problemą sprendžia atskiras pagalbinių klasių paketas – naudojant tokį šabloną, programuotojas, susiduriantis su bendrine problema, kuri, tikėtina, jau yra išspręsta sistemoje, turėtų aiškų procesą, kaip elgtis šioje situacijoje:

1. Atsidaryti vieną paketą, skirtą pagalbinėms, daugkartinio naudojimo klasėms
2. Pakete surasti klasę, kurios pavadinimas būtų susijęs su jo problema
3. Klasės funkcijų saraše surasti jam tinkamą funkciją.
4. Jei reikalingas funkcionalumas nerastas, įgyvendinti jį pasirinktoje klasėje.
5. Iškviesti rastą arba sukurtą funkciją iš bendrinio panaudojimo kodo paketo savo funkcionalume

Pakete reikėtų turėti atskiras klases kiekvienai bendrinei dalykinei sričiai, iš kurios pavadinimo programuotojas galėtų nuspresti, kad jo ieškomas funkcionalumas bus būtent toje klasėje. Tokį skirstymo būdą siūlo ir *Domain-Driven Design: Tackling Complexity in the Heart of Software* autorius – šiuo atveju pagalbinių klasių paketas galėtų būti laikomas palaikymui skirta dalykinės srities esybe. Tokiu atveju svarbu užtikrinti, kad iš klasių pavadinimo aišku, kokią smulkesnę dalykinės srities sritį padengia klasė, bei kad šios klasės neturėtų priklausomybių nuo jas naudojančių klasių – kitu atveju sudaromos ciklinės priklausomybės.

Skirstant pagalbinės klases po ją naudojančios dalykinės srities esybės paketais, kyla kodo pasikartojimo rizika – nepatikrinus skirtingoms esybėms skirtų pagalbinių klasių, sunku žinoti, ar toks funkcionalumas jau buvo įgyvendintas. Išanalizavus galimus kodo skirstymo į paketus šablonus pagalbinių klasių skirstymui, galima teigti, kad tinkamesnis sprendimas šiai problemai yra atskiras pagalbinių klasių paketas.

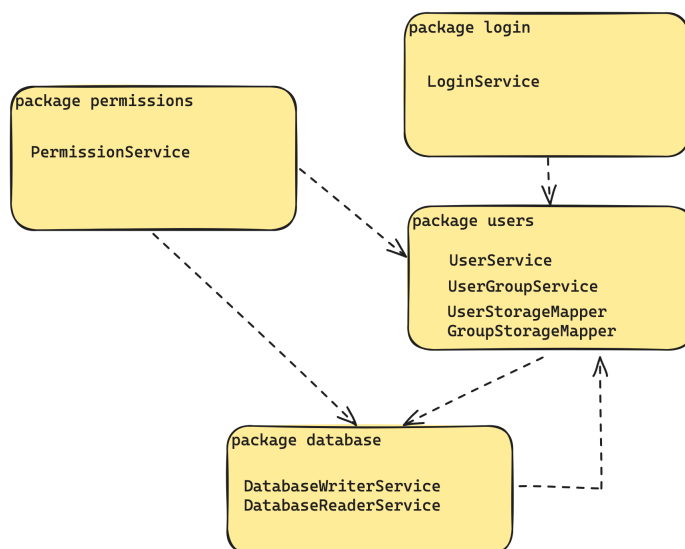
### 2.0.2. Didelio klasių skaičius pakete skirstymo šablonų analizė

Dideliam klasių pakete skaičiui spręsti buvo išskirti du šablonai – žemesnio lygio paketų sudarymas grupuojant pagal techninį sluoksnį ir skirstymas pagal smulkų funkcionalumą. Klasių skirstymas į paketus pagal techninį funkcionalumą dažnu atveju yra mažiau pastangų reikalaujantis metodas – atskirti, pavyzdžiui, kurios klasės priklauso *service* sluoksniui yra paprasčiau, nei įvertinti, kurių klasių teikiamas funkcionalumas yra glaudžiau susijęs. Tačiau šis būdas ne taip efektyviai prisideda prie lengviau suprantamos sistemos struktūros – hibridinis skirstymo metodas nesuteikia

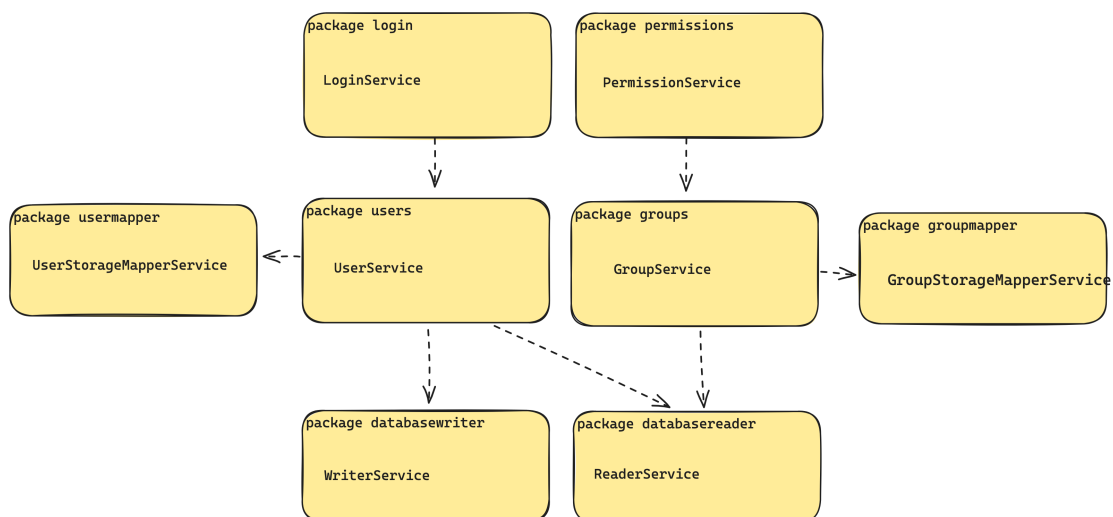
pilnos informacijos nei apie dalykinės srities esybes, nei apie technines sistemos dalis. Taip pat, taikant šį metodą, gali kilti papildomų problemų – ne visas klases galima užtikrintai priskirti vienam sluoksniui. Tokiu atveju, pavyzdžiui, esybės pagalbinėms klasėms, *orchestrator* tipo klasėms reikėtų kurti papildomus paketus.

Skirstymą pagal smulkų funkcionalumą pagrindžia Robert C. Martin bendro sąryšio principas, kuris teigia, kad visos tarpusavyje susijusios klasės turėtų būti viename pakete ir akcentuoja siekiamybę turėti gan mažus paketus, turinčius aiškiai apibrėžtą funkcionalumą, priežastį egzistuoti, taip užtikrinant ir glaudų tarpusavyje susijusių klasių sąryšį. Šis principas taip pat gali padėti sumažinti aferentinių jungčių skaičių paketuose. Didelis priklausomybių nuo specifinio paketo skaičius (arba aferentinės jungtys), reiškia, kad pokyčiai tame pakete turės įtaką kelioms klasėms. Funkcionalumas kitiems paketams galėtų būti pasiekiamas per vieną minimalią sąsają (angl. *interface*), kuri atskleidžia tik konceptus (metodus arba duomenų tipus), kurie yra glaudžiai susiję su komponento teikiama paslauga, bei klase, grąžinančią minėtos sąsajos įgyvendinimą. Paketas, turintis vieną funkciją, yra naudojamas tik tų paketų, kuriems reikia būtent tos funkcijos, taip užtikrinant tik mažos sistemos dalies priklausomybę nuo vieno paketo. Taip pat mažas paketo funkcionalumas reiškia, kad minėtas paketas skirtas funkcionalumui įgyvendinti naudos minimalų kitų sistemos esybių skaičių, taip sumažinant ir eferentinių jungčių skaičių.

Žemiau esančiuose paveikslėliuose galima matyti, kaip išskaidant paketus, turinčius kelis funkcionalumus (pavyzdys 17), yra sumažinamas paketų priklausomybių skaičius (pavyzdys 18).



17 pav. Sistemos pavyzdys su kelias funkcijas atliekančiais paketais



18 pav. Sistemos pavyzdys su aiškia, vieną funkciją turinčiais paketais

Toks skirstymo būdas taip pat sprendžia ciklinių priklausomybių problemą – pavyzdžiui, įrankio, skirto spręsti ciklinių priklausomybių problemą, kūrimo aprašas [Aga15] teigia, kad ciklinių priklausomybių problemą galima spręsti laikantis trijų principų – bendro panaudojimo, bendro keitimosi bei paleidimo ir pernaudojimo ekvivalentumo. Šie principai yra išvesti iš bendro sąryšio principo ir akcentuoja, kad kartu besikeičiančios klasės turėtų būti viename pakete. Tokiu atveju ciklinių priklausomybių tikimybė sumažėja.

Šio šablono pritaikymas gali būti sudėtingesnis – norint išskaidyti didesnės apimties paketą į kelis mažesnius, gali būti sudėtinga atskirti, koks grupavimas tinkamesnis. Tačiau, nors šį principą sunkiau pritaikyti, jis gali padėti išspręsti ne tik didelio klasių skaičiaus pakete problemą, bet ir prisidėti prie mažesnio priklausomybių skaičiaus, taip užtikrinant sistemos tvirtumą.

Išanalizavus galimus kodo skirstymo į paketus šablonus didelio klasių skaičiaus pakete skirstymui, galima teigti, kad tinkamesnis sprendimas šiai problemai yra skirstymas pagal smulkų funkcionalumą.

### 2.0.3. Skirtingų sąsajų implementacijų skirstymo šablonų analizė

Skirtingų sąsajų implementacijų skirstymui buvo išskirti du šablonai – sąsajų ir implementacijų grupavimas bei įgyvendinimų atskyrimas. Naudojant sąsajų ir implementacijų grupavimą, lengva rasti reikalingas implementacijas, tačiau šis būdas turi vieną trūkumą – pakete su dideliu klasių kiekiu bei keliomis skirtingomis sąsajomis sunku suprasti, kuri klasė kurią sąsają įgyvendina.

Naudojant įgyvendinimų atskyrimą, galima labai greitai rasti sąsajas bei galimus jos įgyvendinimus beveik netyrinėjant sistemos struktūros bei klasių kodo. Šis šablonas taip pat užtikrina, kad paketas turi vieną aiškų funkcionalumą, todėl laikosi bendro sąryšio principo.

Išanalizavus galimus kodo skirstymo į paketus šablonus skirtingų sąsajų implementacijų skirstymui, galima teigti, kad tinkamesnis sprendimas šiai problemai yra įgyvendinimų atskyrimas.

#### **2.0.4. Esybių pokyčių ir versijavimo skirstymo šablonų analizė**

Esybių versijavimo skirstymui buvo išskirtas skirtingų versijų grupavimo į paketus šablonas. Toks skirstymo būdas leidžia išvengti kodo duplikacijos bei perteklinio klasių skaičiaus paketuose, bei užtikrina aiškią skirtingų versijų atskirtį. Toks būdas gali praplėsti skirstymą pagal dalykinę sritį – skirstomos versijos gali atspindėti reikalingus palaikyti besikeičiančius verslo reikalavimus.

### 3. Kompiuterinės sistemos vertinimas

#### 3.1. Kompiuterinės sistemos kokybė

Išskyrus tinkamiausius naudoti kodo skirstymo į paketus šablonus, reikėtų objektyviai pamatuoti jų įtaką sistemos kokybei ir užtikrinti, kad jie prisideda prie labiau suprantamos ir lengviau palaikomos sistemos struktūros [Ala07]. Tam reikėtų apibrėžti, kokiais požymiais pasižymi gerai įgyvendinta kompiuterinė sistema. Martin Kleppmann savo knygoje *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems* [Kle17] išskiria šiuos pagrindinius kriterijus:

- Patikimumas, reiškiantis, kad net ir klaidų (įrangos, programinių ar žmogiškųjų) atveju, sistema veikia stabiliai ir patikimai, paslepiančiam tikras klaidas nuo vartotojo [Kle17].
- Prižiūrimumas, reiškiantis, jog skirtingų abstrakcijų pagalba sumažintas sistemos kompleksiskumas. Dėl to nesunku keisti esančią sistemos funkcionalumą bei pritaikyti naujiems verslo naudojimo atvejams. Tai supaprastina darbą inžinierių ir operacijų komandoms, dirbančioms su šia sistema, taip pat leidžia prie sistemos prisidėti naujiems žmonėms, o ne tik jos ekspertams. Tai ypač aktualu atviro kodo sistemoms [Kle17].
- Plečiamumas, reiškiantis, jog sistema turi strategijas, kaip išlaikyti gerą našumą užklausių srautui didėjant ir sistemai augant, tai atliekant su pagrįstais kompiuteriniais resursais ir priežiūros kaina [Kle17].

Yra daug skirtingų elementų, sudarančių sistemą, kuri tenkintų aukščiau paminėtus kriterijus, pavyzdžiui, pasirinktos technologijos, aukšto lygio architektūra, dokumentacija, sistemos testavimo procesai, jų kiekis ir pan. Vienas iš svarbių elementų, prisidedančių prie gerai įgyvendintos sistemos yra programinio kodo struktūra, todėl laikas, skirtas rasti sistemai tinkamus skirstymo į paketus šablonus ir tų šablonų laikytis, atsiperka padarant programinį kodą geriau suprantamu, taip prisidedant prie bendro sistemos patikimumo, plečiamumo ir lengvesnio palaikomo.

Straipsnyje *Investigating The Effect of Software Packaging on Modular Structure Stability* [Sho19], autoriai akcentuoja, kad gerai įgyvendintos objektinio stiliaus sistemos turėtų vystytis be didelių pakeitimų jų architektūroje. To siekiama, nes architektūriniai pakeitimai paveikia didelę sistemos dalį ir jų įgyvendinimo ir sistemos priežiūros kaštai yra žymiai didesni [Sho19]. Paketų struktūra, kuri užtikrina atsietą (angl. *decoupled*) komunikavimą tarp paketų, inkapsuliuoja paketų vidinius elementus, neleidžiant pakeitimais išplisti už paketų ribų yra pagrindas stabiliai sistemos architektūrai, gebančiai efektyviai plėstis, neįnešant nebūtinų pokyčių ir sutaupant programos priežiūros kaštus.

#### 3.2. Kodo skirstymo į paketus šablonų vertinimas

Ankstesniame skyriuje buvo analizuojami kodo skirstymo į paketus šablonai, tačiau lieka neatsakytas klausimas – kaip įvertinti šabloną kodui į paketus grupuoti ir kaip užtikrinti, jog pasirinktas sprendimas yra būtent toks, kokio reikia, ir kokia jo įtaka kompiuterinei sistemai? Tvarkingai bei aiškiai išskirstytų klasių apibrėžimas gali priklausyti nuo komandos, naudojamos

programavimo kalbos ar programinių įrankių bei programinės sistemos dalykinės srities. Kodo grupavimo į paketus metodai, taip, kaip ir bendros tvarkingo kodo praktikos, gali būti subjektyvūs ir patogūs tik metodą formavusiam asmeniui. Tam, kad būtų galima pagrįstai įvertinti skirtingus kodo skirstymo šablonus, pasiekiant kuo objektyvesnį, įtaką sistemos skaitomumui bei palaikomumui nusakantį rezultatą, reikėtų aprašyti kriterijus, nusakančius, ko tikimasi iš paketų struktūros.

Robert C. Martin savo knygoje *Agile Software Development, Principles, Patterns, and Practices* [Mar02] aprašo principus, padedančius grupuoti klases į paketus. Rodikliai, kiek paketas laikosi kiekvieno iš nurodytų principų naudojant tam tikrą paketų skirstymo šabloną, gali būti kriterijus vertinant to šablono kokybę ir įtaką bendrai sistemos struktūrai.

### 3.2.1. Bendro sąryšio principas

*Klasės pakete turėtų būti susietos kartu, kad turėtų tą pačią prielastį pasikeisti. Pakeitimas, kuris paveikia paketą, paveikia visas to paketo klases ir jokių kitų paketų.*

Kaip teigia vienos atsakomybės principas (angl. *Single responsibility principle*), klasė turėtų neturėti skirtingų priežasčių keistis. Šis principas taip pat teigia, kad paketas taip pat neturėtų turėti skirtingų priežasčių pasikeisti. Principas ragina suburti visas klases, kurios gali keistis dėl tų pačių priežasčių, į vieną vietą. Jei dvi klasės yra taip stipriai susietos, kad jos įprastai keičiasi kartu, jos turėtų būti tame pačiame pakete. Kai reikia išleisti pakeitimus, geriau, kad visi pakeitimai būtų viename pakete. Tai sumažina darbo krūvį, susijusį su pakeitimu išleidimu, pakartotiniu patvirtinimu ir programinės įrangos perskirstymu, be reikalo neatliekant validacijos ir neleidžiant kitų, nesusijusių modulių [Mar02].

### 3.2.2. Aciklinių priklausomybių principas

*Paketo priklausomybės diagramoje neturi būti ciklų.*

Priklausomybių ciklai sukuria neatidėliotinų problemų. Žiedinės priklausomybės gali sukelti domino efektą, kai nedidelis, lokalus vieno modulio pokytis išplinta į kitus modulius, dėl to, norit testuoti vieną nedidelį modulį, reikia iš naujo sukompiliuoti didžiulę sistemos dalį. Taip pat žiedinės priklausomybės lemia programos ir kompiliavimo klaidas, kadangi pasidaro labai sunku sudaryti tvarką, kaip kompiliuoti paketus. Žiedinės priklausomybės taip pat gali sukelti begalinę rekursiją, kuri sukelia nemalonių problemų tokioms kalboms kaip Java, kurios skaito savo deklaracijas iš sukompiliuotų dvejetainių failų [Mar02].

### 3.2.3. Stabilių priklausomybių principas

*Paketų priklausomybės turėtų laikytis stabilumo krypties.* Sistema negali būti visiškai statiška. Norint jai plėstis būtinas tam tikras nepastovumas. Kai kurie paketai yra sukurti taip, kad būtų nepastovūs, tikimasi, kad sistemai plečiantis, jie keisis. Nuo paketo, kuris, manoma, yra nepastovus, neturėtų priklausyti sunkiai pakeičiami paketai, priešingu atveju nepastovų paketą taip pat bus sunku pakeisti. Gali susiklostyti situacija, kad modulis, sukurtas taip, kad jį būtų lengva pakeisti, kartais tampa sunkiai keičiamu, pridėjus netinkamą priklausomybę nuo jo [Mar02].

### 3.2.4. Stabilių abstrakcijų principas

*Paketai turi būti tiek abstraktūs, kiek ir stabilūs.* Šis principas nustato ryšį tarp stabilumo ir abstraktumo. Stabilūs paketai turėtų būti abstraktūs, todėl ir lengvai praplečiami. Tai pat šis principas teigia, kad nestabilus paketas turi būti konkretus, nes jo nestabilumas leidžia lengvai pakeisti jo turinio kodą. Taigi, jei paketas yra stabilus, jį taip pat turėtų sudaryti abstrakčios klasės, užtikrinant jo išplečiamumą. Stabilūs paketai, kurie yra lengvai išplečiami, yra lankstūs pakeitimams, nedarant didelės įtakos sitemos struktūrai [Mar02].

### 3.3. Paketų kokybės matai

Beveik visi autoriaus aprašyti principai turi aiškiai apibrėžtus matus, kuriais galima patikrinti, kiek paketas laikosi šių principų. Būtent šie matai bus naudojami įvertinti paketus analizuojamuose šablonuose norint patikrinti šablonų poveikį sistemai:

- **Klasių skaičius** ( $N$ ) – klasių skaičiaus matas paketui nurodo, kiek klasių (konkrečių ir abstrakčių) yra pakete. Šis matas matuoja paketo dydį.
- **Aferentinės jungtys (angl. *Afferent Couplings*)** ( $Af$ ) – aferentinių jungčių matas nurodo skaičių paketų, kurie priklauso nuo klasių, esančių pasirinktame pakete. Šis matas matuoja ateinančias priklausomybes.
- **Eferentinės jungtys (angl. *Efferent Couplings*)** ( $Ef$ ) – eferentinių jungčių matas nurodo skaičių kitų paketų, nuo kurių priklauso klasės pasirinktame pakete. Šis matas matuoja išeinančias priklausomybes.
- **Stabilumas** ( $S$ ) – stabilumo matas nurodo santykį tarp eferentinių jungčių ir visų jungčių (aferentinės + eferentinės) pakete. Šis matas matuoja paketo atsparumą pokyčiams, kuris buvo akcentuojamas stabilų priklausomybių principu:

$$Stabilumas(S) = \frac{Jungtys_{eferentines}}{Jungtys_{eferentines} + Jungtys_{aferentines}} \quad (1)$$

Reikšmės režiai – nuo nulio iki vieno, kur nulis nurodo visiškai stabilų paketą, o vienetą – visiškai nestabilų.

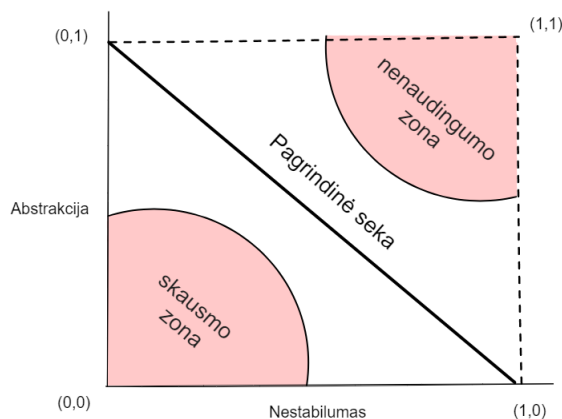
- **Abstrakcija** ( $A$ ) – paketo abstrakcijos matas nurodo santykį tarp abstrakčių klasių (arba sąsajų (angl. *interface*)) pakete ir bendro klasių skaičiaus:

$$Abstrakcija(A) = \frac{N_{abstrakcios}}{N_{visos}} \quad (2)$$

Šis matas nurodo paketo abstraktumą. Abstrakcijos reikšmė gali būti tarp nulio ir vieno. Nulis reiškia, kad paketas neturi jokių abstrakčių klasių, o vienetą nurodo, kad pakete yra tik abstrakčios klasės.

- **Atstumas nuo pagrindinės sekos** ( $D$ ) – pagrindinė seka yra sąryšis tarp stabilumo ir abstrakcijos. Pagrindinė seka – idealizuota linija ( $Abstrakcija + Nestabilumas = 1$ ) kurią galima vaizduoti kaip kreivę, su abstrakcijos dydžiu  $y$  ašyje ir nestabilumu  $x$  ašyje. Paketas tiesiai

ant pagrindinės sekos yra optimaliai subalansuotas atsižvelgiant į jo abstraktumą ir stabilumą. Idealūs paketai yra arba visiškai abstraktūs ir stabilūs (Stabilumas=0, Abstrakcija=1) arba visiškai konkretūs ir nestabilūs (Nestabilumas=1, Abstrakcija=0) (pavyzdys 19).



19 pav. Pagrindinės sekos kreivė

Atstumą nuo pagrindinės sekos galima apskaičiuoti kaip:

$$Atstumas = |Abstrakcija + Nestabilumas - 1| \quad (3)$$

Šis matas yra paketo abstraktumo ir stabilumo pusiausvyros rodiklis. Šio mato diapazonas yra nuo nulio iki vieno, kur nulis reiškia paketą, sutampantį su pagrindine seka, o vienas – paketą, maksimaliai nutolusį nuo pagrindinės sekos.

- **Žiedinės priklausomybės (C)** – žiedinių priklausomybių matas skaičiuoja atvejus, kur pasirinkto paketo išeinančios priklausomybės taip pat yra paketo ateinančios priklausomybės (tiesiogiai arba netiesiogiai). Šis matas – aciklinių priklausomybių rodiklis, minėtas aciklinių priklausomybių principu.

Šių matų patikimumas buvo įvertintas atvejo analizėje *Exploring the Relationships between Design Metrics and Package Understandability: A Case Study* [Eli10], kurioje autoriai tyrinėjo sąryšį tarp minėtų matų ir vidutinių pastangų, reikalingų suprasti objektinio dizaino paketą. Tyrimas atliktas naudojant aštuoniolika paketų, paimtų iš dviejų atviro kodo programinės įrangos sistemų. Paskaičiuoti pastangas, reikalingas paketui suprasti, buvo pasitelktos trys skirtingos komandos, turinčios po tris panašią patirtį turinčius programuotojus. Jų buvo paprašyta pilnai suprasti paketų funkcionalumą ir nuo vieno iki dešimt įvertinti pastangas, reikalingas suprasti kiekvieną paketą. Rezultatai gauti iš šio tyrimo rodo statistškai reikšmingą koreliaciją tarp daugumos matų ir paketų suprantamumo. Sistemos suprantamumui įtakos nedarė tik eferentinių jungčių skaičius, tačiau šis matas reikalingas apskaičiuoti paketų stabilumo matą.



## 4. Įrankiai šablonų analizei ir įvertinimui

Kompiuterinės sistemos, kurioms yra aktualu klasių ir paketų skirstymo metodai, paprastai tokio dydžio, kad pilnai jas perprasti, nustatyti jų įgyvendinimo kokybę, naudojamus šablonus kodui skirstyti, apskaičiuoti paketų kokybės matus yra sudėtingas procesas. Daryti tai rankiniu būdu užtrunka daug laiko bei yra paliekama daug vietos potencialioms klaidoms, todėl yra būtina šį procesą optimizuoti, skaitmenizuoti analizės procedūras pasitelkiant programinių įrankių pagalbą.

### 4.1. Reikalavimai įrankiams

Įrankių, leidžiančių paprasčiau atlikti sistemų analizę, atsakomybes galima suskirstyti į dvi grupes:

- Bendrinė sistemos analizė – įrankis ar įrankiai padeda atlikti bendrinę sistemos analizę. Šios atsakomybių grupės įrankių išvestis nėra objektyvūs, tiksliai apibrėžtų formulių rezultatai, o papildoma, aiškiai pavaizduota meta informacija apie sistemą – paketų struktūrą, juose esančių klasių priklausomybes, dažnai pasikartojančius paketų bei klasių vardus. Ši papildoma informacija nėra aiškūs teiginiai, o tik pagalba analizę atliekančiams asmeniui, leidžianti priimti įžvalgas apie sistemą, kaip pavyzdžiui, kokiam paketų skirstymo šablonui yra atimčiausia sistemos struktūra, arba kaip lengvai sistema yra suprantama.
- Paketų kokybės matų skaičiavimas – įrankis ar įrankiai turi padėti apskaičiuoti aprašytus paketų kokybės matus. Šių įrankių išvestis – tikslūs, formulėmis pagrįstų skaičiavimų rezultatai apie paketų atitikimą kriterijams, kuriuos galima lyginti tarpusavyje.

### 4.2. Reikalavimai bendrinės sistemos analizės įrankiui

Įrankis bendrinei sistemos analizei atlikti turėtų suteikti galimybę naudotojui nurodyti kelią iki *Java* programavimo kalba parašytos sistemos arba posistemės ir joje atlikti jos turinio analizę bei naudotojui pateikti naudingas išvadas, sudarytas iš:

- Klasių ir paketų skaičiaus
- Vidutinio klasių pakete skaičiaus
- Paketų ir klasių medžio, identifikuojančio abstrakčias klases ar sąsajas
- Paketų priklausomybių grafiką

Gautą rezultatą išvesti suprantamu formatu, leidžiant vartotojui susidaryti išvadas apie sistemos arba tam tikros posistemės struktūrą, naudojamus įrankius bei kokybę.

### 4.3. Reikalavimai įrankiui paketo kokybei skaičiuoti

Įrankis paketo kokybei skaičiuoti turėtų suteikti galimybę naudotojui nurodyti kelią iki *Java* programavimo kalba parašytos sistemos arba posistemės ir joje apskaičiuoti kiekvieno paketo kokybės matus:

- Klasių skaičių

- Aferentinių jungčių skaičių
- Eferentinių jungčių skaičių
- Nestabilumo santykį
- Abstrakcijos santykį
- Atstumo nuo pagrindinės sekos santykį
- Žiedinių priklausomybių skaičių

Gautą rezultatą išvesti vartotojui suprantamu formatu, kuriame matytųsi individualių paketų matai bei šių matų vidurkis sistemoje (arba posistemėje). Išvedimo formatas turėtų būti toks, jog skirtingų analizių rezultatai būtų lengvai palyginami su kitais.

Abiejuose įrankiuose vienas iš palaikomų išvesties formatų turėtų būti *latex*, taip suteikiant galimybę analizės rezultatus pateikti tolesniame šio darbo turinyje.

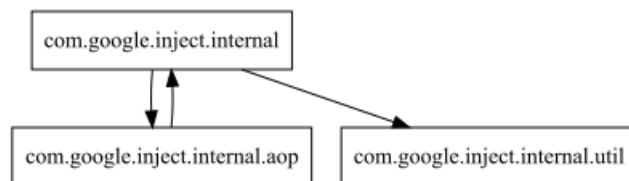
#### 4.4. Įrankių įgyvendinimas

Nors beveik visiems reikalavimuose minimiems funkcionalumams galima rasti jau sukurtų įrankių, greit ir efektyviai pritaikyti juos skirtingoms sistemoms (arba posistemėms) nėra patogu – kiekvieną įrankį reikėtų vykdyti atskirai, skirtingais vykdymo procesais ir argumentais. Taip išvestų rezultatų formatai yra skirtingi. Todėl, norint palengvinti šį procesą – suvienodinti procesų vykdymą bei gautus rezultatus – visi analizei reikalingi įrankiai yra įgyvendinti kaip viena programa, kuri apdoroja *java* sistemos failus ir, naudojant trečiųjų šalių įrankius, suskaičiuoja reikiamus rezultatus apie analizuotą sistemą.

Sistema parašyta *Kotlin* programavimo kalba, o jos vykdomas kodas yra *jar* programėlė paleidžiama iš komandinės eilutės:

```
java -jar system-analyzer.jar -output /isvesties/direktorija -
system kelias/iki/sistemas
```

Įvykdyta programa rekursyviai nuskaityti visus failus bei paketus sistemos direktorijoje, nurodytoje *system* argumentu. Nuskaitytų failų struktūros yra apdorojamos įrankiu *javaparser*<sup>15</sup>, kuris leidžia programiškai pasiekti *Java* klasės failo elementus – klasių, paketų vardus, metodus. Surinkus informaciją apie paketus bei jų priklausomybes, jie naudojami paskaičiuoti kiekvieno paketo kokybės matus. Paketų informacija taip pat yra paduodama įrankiui *graphviz*<sup>16</sup>, kuris iš jų sukuria paveikslėlius, vaizduojančius paketų priklausomybių grafiką (pavyzdys 20)



20 pav. *graphviz* įrankio sugeneruotas priklausomybių grafikas

<sup>15</sup><https://github.com/javaparser/javaparser>

<sup>16</sup><https://graphviz.org/>

Gauti matų įverčiai, priklausomybių grafikas bei bendras sistemos paketų medis yra išvedami į išvesties direktoriją, nurodytą *output* argumentu. Rezultatai grąžinami kaip *latex* formato failai, todėl juos galima įdėti į darbo projektą, taip padarant sistemų analizę greitesnę bei efektyvesnę. Įrankio programinis kodas yra pasiekiamas viešojoje *GitHub*<sup>17</sup> repozitorijoje.

---

<sup>17</sup><https://github.com/MartynaUb/bachelor-thesis-analysis-of-code-packaging-patterns/tree/main/code/analyzer>

## 5. Esamų sistemų pertvarkymas

Išskyrus šablonus kodo skirstymui į paketus ir juos išanalizavus, reikia įvertinti jų įtaką, juos pritaikant esamoms sistemoms. Šio skyriaus tikslas – pasirinkti ir perrašyti kelias sistemas, naudojant gautą šablonų imtį. Atlikus pertvarkymus sistemose, dar paskaičiuoti paketų kokybės matavimus, taip gaunant įrodymus, ar atrinkti šablonai iš tiesų sprendžia jiems priskirtas problemas ir prisideda prie sistemos kokybės.

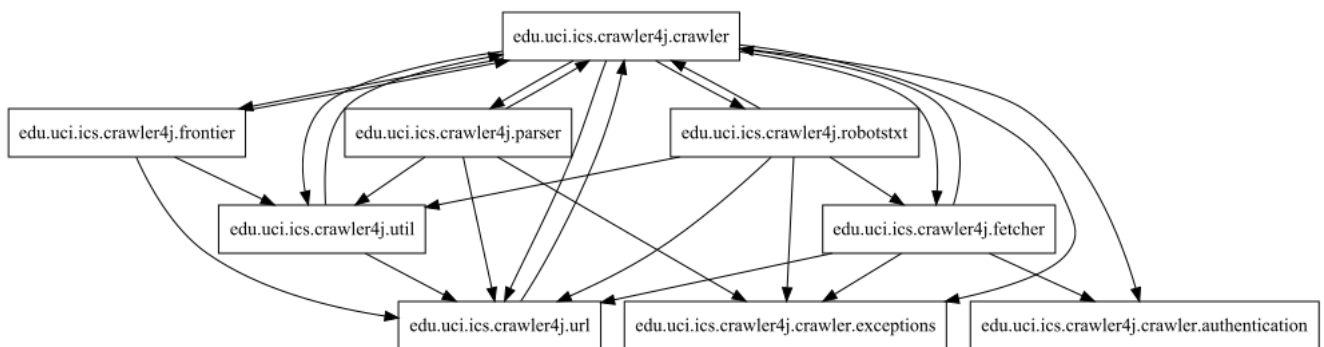
### 5.1. Sistemų pasirinkimas

Sistemų pertvarkymui pasirinktos atviro kodo sistemos, kurių kodas yra viešai prieinamas *GitHub* platformoje. Pasirinktos sistemos yra vidutinio dydžio, todėl nėra labai sudėtinga jas suprasti ir pertvarkyti, bet taip pat jos nėra tokios paprastos, kad neturėtų sistemos projektavimo problemų. Pasirinktos sistemos yra skirtingo tipo projektai, taip užtikrinant didesnę problemų įvairovę ir objektyvesnius įvertinimus. Sistemos yra neįdegalios ir jose matomos problemos, kurias teoriškai turėtų išspręsti atrinkti šablonai. Per pasirinktų sistemų imtį yra sutinkamos visos aprašytos problemos, taip įvertinant visus pasirinktus šablonus.

### 5.2. *crawler4j* sistema

**crawler4j**<sup>18</sup> yra atvirojo kodo žiniatinklio tikrinimo (angl. *crawling*) programa parašyta *Java* programavimo kalba, leidžianti efektyviai tikrinti žiniatinklį naudojant daugiagijį (angl. *multithreaded*) procesą. Ši sistema yra taikomosios programinės įrangos tipo, skirta naudoti vartotojų, o ne kitų sistemų.

Prieš visus pakeitimus *crawler4j* sistemos paketų struktūra atrodo taip (pavyzdys 21)



21 pav. *crawler4j* sistemos struktūra

Atlikus sistemos kokybės matų analizę, gautą rezultatą galima naudoti problemų paieškai sistemoje (pavyzdys 22).

<sup>18</sup><https://github.com/yasserg/crawler4j>

Paketo vardas	$N$	$A$	$E$	$S$	$A$	$D$	$C$
edu.uci.ics.crawler4j.crawler.authentication	4	2	0	0.0	0.25	0.75	0
edu.uci.ics.crawler4j.url	4	6	1	0.143	0.0	0.857	1
edu.uci.ics.crawler4j.crawler.exceptions	3	4	0	0.0	0.0	1.0	0
edu.uci.ics.crawler4j.crawler	5	6	8	0.571	0.2	0.229	5
edu.uci.ics.crawler4j.robotstxt	6	1	5	0.833	0.0	0.167	1
edu.uci.ics.crawler4j.frontier	6	1	3	0.75	0.0	0.25	1
edu.uci.ics.crawler4j.fetcher	6	2	4	0.667	0.0	0.333	1
edu.uci.ics.crawler4j.util	3	4	2	0.333	0.0	0.667	0
edu.uci.ics.crawler4j.parser	12	1	4	0.8	0.167	0.033	1
Vidurkiai	$\bar{N}$	$\bar{A}f$	$\bar{E}f$	$\bar{S}$	$\bar{A}$	$\bar{D}$	$\Sigma C$
	5	3	3	0.455	0.069	0.476	5

22 pav. *crawler4j* paketų kokybės matai

#### Matų lentelės paaiškinimas:

- $N$  – klasių skaičių pakete.
- $Af$  – aferentinės jungtys.
- $Ef$  – eferentinės jungtys.
- $S$  – stabilumas.
- $A$  – paketo abstraktuma.
- $D$  – atstumas nuo pagrindinės sekos.
- $C$  – ciklinių priklausomybių skaičius.

Kaip skaičiuojami matai aprašyta paketų kokybės matų skyriuje 3.3, puslapyje 23.

Iš bendros sistemos analizės bei matų lentelės rezultatų matomos kelios sistemos problemos –

- **Problema numeris 1** – atlikus bendrą sistemos kodo analizę matoma, kad sistemos paketuose yra sąsajos kurios turi daugiau nei vieną įgyvendinimą, neturint aiškiau šablono, kaip skirstyti skirtingus įgyvendinimus pasidaro sudėtinga rasti tinkamą specifinės sąsajos įgyvendinimą.
- **Problema numeris 2** – paketo *edu.uci.ics.crawler4j.parser* viduje yra 12 klasių, turinčių skirtingas atsakomybes, todėl yra sunku suprasti, kokia būtent funkciją atlieka šis paketas, kokie būtent pokyčiai produkte gali pareikalausiti jame pakeitimų.
- **Problema numeris 3** – sistema turi daug ciklinių priklausomybių, kur du arba daugiau paketų priklauso vienas nuo kito. Tai matoma sistemos paketų diagramoje pavyzdyje 21, ciklinėse priklausomybėse dalyvauja 10 paketų ir kartu jie sudaro 5 ciklinių priklausomybių sekas, kurios turėtų būti pašalintos.

Sistemos paketas *edu.uci.ics.crawler4j.parse*, turi sąsają *ParseData* bei 4 skirtingus jos įgy-

vendinimus – *CssParseData*, *TextParseData*, *HtmlParseData*, *BinaryParseData*, šios klasės pakete dalinasi vieta dar su 9 klasėmis, todėl yra nepatogu rasti ieškomą įgyvendinimą (pavyzdys 23).

```
/parser
├─ AllTagMapper
├─ ParseData
├─ TextParseData
├─ HtmlParseData
├─ CssParseData
├─ BinaryParseData
├─ ExtractedUrlAnchorPair
├─ HtmlParser
├─ HtmlContentHandler
├─ TikaHtmlParser
└─ Parser
```

23 pav. *parser* paketo struktūra

Šiam paketui sutvarkyti taikomas *įgyvendinimų atskyrimo* šablonas – sąsaja iškeliamą į atskirą paketą *parse*, jos įgyvendinimai perkeliama į minėto paketo subpaketus. Šie pertvarkymai pakeičia *parser* paketo struktūrą (pavyzdys 24).

```
/parser
├─ parse
│   ├── ParseData // Sąsaja
│   ├── text
│   │   └─ TextParseData // Pirmas sąsajos įgyvendinimas
│   ├── html
│   │   └─ HtmlParseData // Antras sąsajos įgyvendinimas
│   ├── css
│   │   └─ CssParseData // Trečias sąsajos įgyvendinimas
│   └─ binary
│       └─ BinaryParseData // Ketvirtas sąsajos įgyvendinimas
├─ AllTagMapper
├─ ExtractedUrlAnchorPair
├─ HtmlParser
├─ HtmlContentHandler
├─ NotAllowedContentException
├─ TikaHtmlParser
└─ Parser
```

24 pav. *parser* paketo struktūra po pirmojo pertvarkymo

Dėl atskirtos sąsajos ir jos įgyvendinimų tampa paprasčiau ją rasti bei suprasti sistemoje egzistuojančius jos įgyvendinimus. Tokia struktūra išsprendžia sistemos **problemą numeris 1**. Šis pertvarkymas taip pat dalinai sumažino **problemą numeris 2**, sumažinant klasių skaičių pakete į

7. Palyginus pertvarkytos ir buvusios sistemos matus matome, kad sistema tapo aiškesnė – sumažėjo vidutinis klasių skaičius paketuose, todėl yra paprasčiau suprasti pagrindinės paketų funkcijas. Nors nežymiai sumažėjo vidutinis stabilumas, bet pakilo abstrakcijos lygis, todėl atstumas nuo pagrindinės sekos sumažėjo (pavyzdys 25).

<i>crawler4j.parser</i>	<i>N</i>	<i>A</i>	<i>E</i>	<i>S</i>	<i>A</i>	<i>D</i>	<i>C</i>
Prieš	12	1	4	0.8	0.167	0.033	1
Po	7 (-5)	1	9	0.9 + (0.1)	0.143 (-0.024)	0.043 (+0.01)	1
Vidurkiai	$\bar{N}$	$\bar{A}f$	$\bar{E}f$	$\bar{S}$	$\bar{A}$	$\bar{D}$	$\Sigma C$
Prieš	5	3	3	0.455	0.069	0.476	5
Po	4 (-1)	3	3	0.488 (+0.033)	0.114 (+0.045)	0.425 (-0.051)	5

25 pav. *crawler4j* paketų kokybės matai po pirmo pertvarkymo

Matų įverčio didėjimas nebūtinai susijęs su sistemos kokybės gerėjimu, todėl teigiamas pokytis paženklintas žaliai, o neigiamas – raudonai. Norint išspręsti **problemą numeris 3** – ciklines priklausomybes, reikia identifikuoti klases, kurių priklausomybės sudaro ciklus ir iškelti jas į atskirus paketus.

Kuriant naujus paketus vadovaujamosi *skirstymo pagal smulkų funkcionalumą* šablonu, užtikrinant, kad kiekvienas paketas teikia vieną, aiškiai apibrėžtą funkciją, kuri yra pasiekama per pakete aprašytą sąsają. Paketo vidus yra paslėptas su *Java* kalbos pasiekiamumo modifikatoriais – konkrečių klasių negalima inicializuoti arba paveldėti už paketo ribų, nes jų konstruktoriai privati (pavyzdys 26). Klasės inicializuojamos statinių metodų pagalba, kurie grąžina inicializuotas konkrečias klases, įgyvendinančias reikiamą sąsają.

```

public interface HtmlParser {

    HtmlParseData parse(ParsedPage page, String contextURL)
        throws ParseException;

    // Grazinamas sasajos igyvendinimas
    static HtmlParser newHtmlParser(CrawlConfig config, TLDList
        tldList) throws InstantiationException,
        IllegalAccessException {
        return new TikaHtmlParser(config, tldList);
    }
}

// Sasajos igyvendinimas. Nera viesas, pasiekiamas tik paketo
viduje, nes klase ir konstruktorius nenaudoja public
raktazodziu
class TikaHtmlParser implements HtmlParser {
    ...
    TikaHtmlParser(CrawlConfig config, TLDList tldList) throws
        InstantiationException, IllegalAccessException {
        ...
    }
}

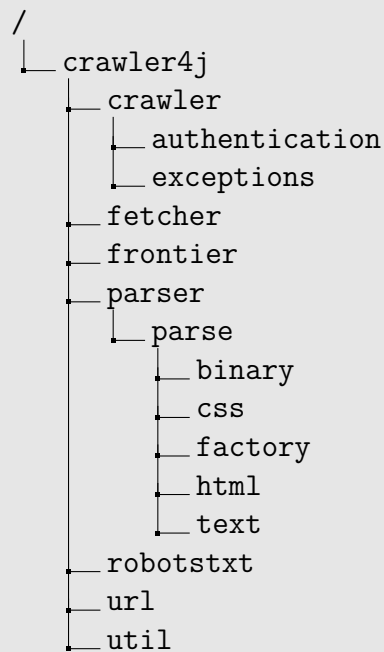
```

26 pav. Sąsaja parašyta laikantys *skirstymo pagal teikiamą funkcionalumą* šablono gaires

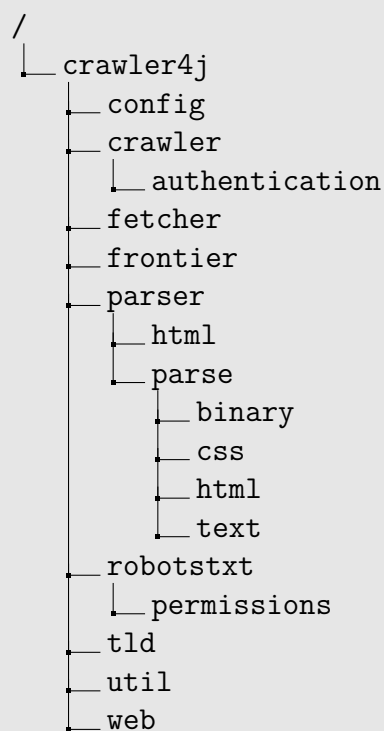
Peržiūrėjus paketus su ciklinėmis priklausomybėmis, iš jų buvo išskirti šie nauji funkcionalumai, kuriems buvo sukurti atskiri paketai (pavyzdys 27, 28).

- Iš *parser* paketo iškeltas funkcionalumas *html* turiniui apdoroti, patalpintas į *parser/html* paketą. Po šio pertvarkymo *parser* paketas pasidaro mažesnis ir turi vieną funkcionalumą – priimti puslapį ir deleguoti jį apdorojimui kitai klasei pagal puslapio tipą.
- Iš *robotstxt* paketo funkcionalumas patikrinti, ar sistema autorizuota apdoroti pasirinktą puslapį, iškeltas į *robotstxt/permissions* paketą.
- Iš *crawler*, bei *parser* paketų funkcionalumas, aprašantis internetinio puslapio elementus, iškeltas į *web* paketą.
- Iš *crawler* paketo funkcionalumas, nustatantis įrankio konfigūraciją, iškeltas į *config* paketą.
- Iš *url* paketo funkcionalumas, gaunantis internetinių domenų pavadinimus, iškeltas į *tld* paketą.



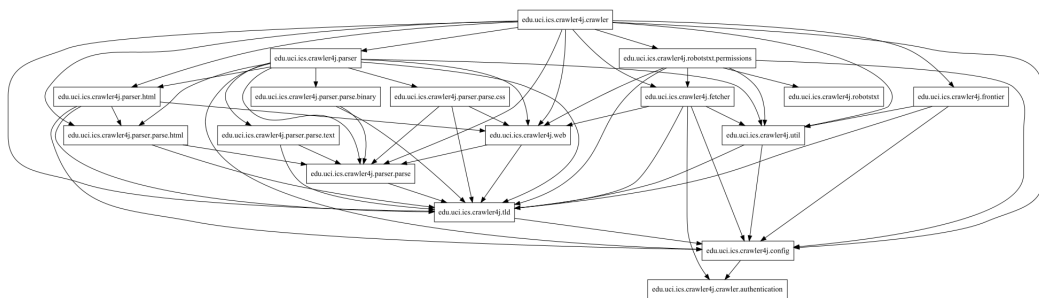


27 pav. *crawler4j* paketų medis prieš iškeliant minėtus funkcionalumus



28 pav. *crawler4j* paketų medis iškėlus minėtus funkcionalumus

Atlikus šiuos funkcionalumų suskaidymus į atskirus paketus buvo panaikintos žiedinės priklausomybės. Naujoje paketų diagramoje galime matyti, jog dabar sistema turi daugiau paketų, tačiau jų priklausomybių kryptys yra aiškesnės (pavyzdys 29).



29 pav. *crawler4j* sistemos struktūra po antro pertvarkymo

Atlikus matų analizę pertvarkytai sistemai yra matoma, kad sistemos kokybė pagerėjo: sumažėjo vidutinis klasių skaičius pakete, ciklinių priklausomybių rodiklis tapo lygus nuliui, padidėjo vidutinis abstrakcijos lygis (pavyzdys 30).

Paketo vardas	$N$	$Af$	$Ef$	$S$	$A$	$D$	$C$
edu.uci.ics.crawler4j.parser.parse.binary	1	1	2	0.667	0.0	0.333	0
edu.uci.ics.crawler4j.crawler	4	0	11	1.0	0.25	0.25	0
edu.uci.ics.crawler4j.tld	3	13	1	0.071	0.333	0.596	0
edu.uci.ics.crawler4j.parser.html	6	2	4	0.667	0.167	0.166	0
edu.uci.ics.crawler4j.robotstxt.permissions	2	1	6	0.857	0.5	0.357	0
edu.uci.ics.crawler4j.config	1	8	1	0.111	0.0	0.889	0
edu.uci.ics.crawler4j.frontier	6	1	3	0.75	0.0	0.25	0
edu.uci.ics.crawler4j.parser	2	1	10	0.909	0.0	0.091	0
edu.uci.ics.crawler4j.crawler.authentication	4	2	0	0.0	0.25	0.75	0
edu.uci.ics.crawler4j.parser.parse.css	1	1	3	0.75	0.0	0.25	0
edu.uci.ics.crawler4j.robotstxt	5	1	0	0.0	0.0	1.0	0
edu.uci.ics.crawler4j.parser.parse.html	1	3	2	0.4	0.0	0.6	0
edu.uci.ics.crawler4j.parser.parse	1	7	1	0.125	1.0	0.125	0
edu.uci.ics.crawler4j.fetcher	6	2	5	0.714	0.0	0.286	0
edu.uci.ics.crawler4j.util	4	5	2	0.286	0.0	0.714	0
edu.uci.ics.crawler4j.web	5	6	2	0.25	0.4	0.35	0
edu.uci.ics.crawler4j.parser.parse.text	1	1	2	0.667	0.0	0.333	0
Vidurkiai	$\bar{N}$	$\bar{Af}$	$\bar{Ef}$	$\bar{S}$	$\bar{A}$	$\bar{D}$	$\Sigma C$
Pradiniai	5	3	3	0.455	0.069	0.476	5
Po	3 (-2)	3	3	0.452 (-0.03)	0.183 (+0.114)	0.471 (-0.005)	0 (-5)

30 pav. *crawler4j* paketų kokybės matai po galutinių pertvarkymų

Iš gautų rezultatų galime teigti, jog po pertvarkymo šablonai *skirstymas pagal smulkų funkcionalumą* ir *įgyvendinimų atskyrimas*, išsprendė visas 3 sistemoje matytas problemas, padarė sistemą lengviau suprantama, panaikino ciklines priklausomybes ir turėjo teigiamą įtaką matams, nurodantiems sistemos įgyvendinimo kokybę.

### 5.3. *azure-sdk-for-java* sistema

**azure-sdk-for-java**<sup>19</sup> yra atviro kodo biblioteka, parašyta *Java* programavimo kalba, leidžianti programiškai bendrauti su *Microsoft Azure* debesų kompiuterijos platforma. Tai programinės įrangos įrankis, skirtas naudoti kitose sistemose, supaprastinant programinį kodą. Ši biblioteka yra labai didelė ir šio pertvarkymo metu yra dirbama tik su posisteme *azure-storage-blob-cryptography* kuri atsakinga už duomenų kriptografiją komunikacijos su *Azure Blob* nestruktūrizuota failų saugykla metu.

Prieš visus pakeitimus *azure-storage-blob-cryptography* sistema turi vieną paketą *com.azure.storage.blob.specialized.cryptography*, kuriame guli visos jos klasės.

Sistemos kokybės matai nėra naudingi dirbant su vienu paketu, nes trūksta konteksto suprasti, kaip paketas yra naudojamas. Tačiau iš bendrinės sistemos analizės galima pamatyti pagrindines sistemos problemas – viename pakete yra 21 skirtinga klasė, tiek sąsajos, tiek konkrečios klasės. Taip pat sistemoje yra kelios skirtingos tos pačios esybės versijos – *DecryptorV1* ir *DecryptorV2*, *EncryptorV1* ir *EncryptorV2*, todėl sunku suprasti, su kokia esybės versija yra susijusios kitos klasės bei yra kodo pasikartojimo. Ši sistema būtų aiškesnė ir greičiau perprantama, jei būtų išskirti mažesni funkcionalumai ir kodas būtų išskaidytas į smulkesnius paketus, taip pat atskiriant juos pagal esybių versijas. Norint atlikti esybių versijavimą naudojamas *skirtingų versijų grupavimo į paketus* šablonas, pagal jį kiekvienai esybės versijai sukuriamas atskiras paketas, taip pat bendras, pasikartojantis kodas iškeliamas į abstrakčią klasę, kuri yra pakete vienu lygiu aukščiau nei versijų paketai (pokyčių pavyzdžiai 31, 32).

```
/cryptography
├─ Decryptor
├─ DecryptorV1 // Pirma esybės versija
├─ DecryptorV2 // Antra esybės versija
├─ Encryptor
├─ EncryptorV1 // Pirma esybės versija
├─   └─ EncryptorV2 // Antra esybės versija
└─ ...// Likusios klasės lieka pagrindiniame pakete
```

31 pav. *azure-storage-blob-cryptography* kriptografijos klasių struktūra prieš pertvarkymą

<sup>19</sup><https://github.com/Azure/azure-sdk-for-java/tree/main/sdk/storage/azure-storage-blob-cryptography>

```

/cryptography
├── decryptor
│   ├── Decryptor // Abstrakti klasė su bendru esybės kodu
│   ├── v1
│   │   └── DecryptorV1 // Pirma esybės versija
│   ├── v2
│   │   └── DecryptorV2 // Antra esybės versija
├── encryptor
│   ├── Encryptor // Abstrakti klasė su bendru esybės kodu
│   ├── agent // Papildomas, su esybe susijęs kodas, naudojamas abiejų
│   │   esybių
│   ├── v1
│   │   └── EncryptorV1 // Pirma esybės versija
│   ├── v2
│   │   └── EncryptorV2 // Antra esybės versija
└── ...// Likusios klasės lieka pagrindiniame pakete

```

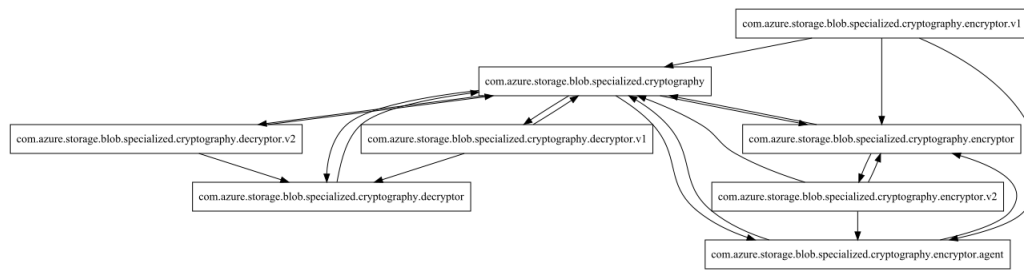
32 pav. *azure-storage-blob-cryptograph* kriptografijos klasių struktūra po jų iškelimo į paketus

Šis pertvarkymas pagal *skirtingų versijų grupavimo į paketus* šabloną, atskiria esybės versijas ir sukuria naujus paketus, apibrėžiančius aiškesnius funkcionalumus. Atsiradus daugiau paketų bei jų sąsajų, galima paskaičiuoti paketų kokybės matus:

Paketo vardas	$N$	$Af$	$Ef$	$S$	$A$	$D$	$C$
specialized.cryptography	13	7	5	0.417	0.0	0.583	5
specialized.cryptography.encryptor.v2	1	1	3	0.75	0.0	0.25	0
specialized.cryptography.encryptor.v1	1	0	3	1.0	0.0	0.0	1
specialized.cryptography.encryptor	1	4	2	0.333	1.0	0.333	2
specialized.cryptography.decryptor.v2	1	1	2	0.667	0.0	0.333	1
specialized.cryptography.decryptor.v1	1	1	2	0.667	0.0	0.333	1
specialized.cryptography.encryptor.agent	2	3	2	0.4	0.0	0.6	1
specialized.cryptography.decryptor	2	3	1	0.25	0.5	0.25	1
Vidurkiai	$\bar{N}$	$\bar{A}g$	$\bar{E}g$	$\bar{S}$	$\bar{A}$	$\bar{D}$	$\Sigma C$
	3	3	3	0.561	0.188	0.335	6

33 pav. *azure-storage-blob-cryptograph* paketų kokybės matai po pirmo pertvarkymo

Apskaičiuoti matai (pavyzdys 33) rodo nedidelį nuokrypį nuo pagrindinės sekos, todėl galima teigti, kad sistemos abstraktumas bei stabilumas yra sąlyginai subalansuoti, tačiau šis skirstymas turėjo ir neigiamų padarinių – sukėlė 6 ciklines priklausomybes. Šiais priklausomybes galima matyti sistemos paketų diagramoje (pavyzdys 34).

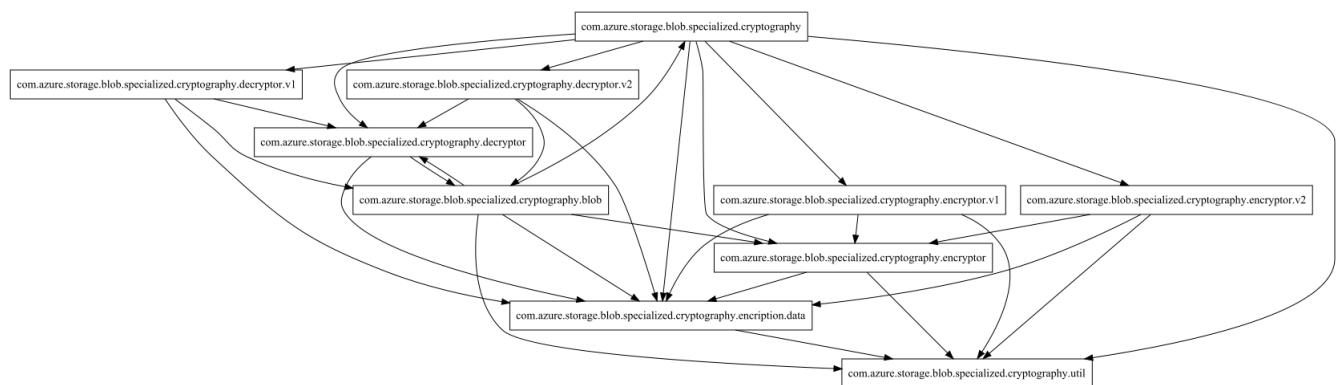


34 pav. *azure-storage-blob-cryptography* sistemos struktūra po pirmojo pertvarkymo

Norint panaikinti ciklines priklausomybės, reikia panaudoti šabloną *skirstymas pagal smulkų funkcionalumą* ir, išskaidant pagrindinį paketą į dar smulkesnius vienetus – yra sukuriamas *encryption.data* paketas, atsakingas už užkoduotų duomenų vaizdavimą, taip pat aprašomas paketas *blob* skirtas *blob* esybių, reprezentuojančių saugyklos duomenų formatą, kriptografijai. Taip pat vadovaujantis *atskiro pagalbinių klasių paketo* šablonu, visos pagalbinės klasės, neturinčios priklausomybių į kitus paketus ir padedančios vykdyti kertinius techninius funkcionalumus yra iškeliamos į *util* paketą. Minėtame pakete atsiranda tokios klasės:

- *CryptographyConstants* – klasė, saugojanti bendras, su kriptografija susijusias konstantas, kurios naudojamos skirtingose sistemos vietose.
- *EncryptionVersion* – aprašo kriptografijos versijas ir jų naudojamus protokolus
- *WrappedKeyJson* – palengvina darbą su serializuojant ir deserializuojant kriptografijos raktus į *json* formatą.

Pritaikius visus šiuos šablonus, gaunama sistemos struktūra turinti nedidelius, aiškesniais funkcionalumais apibrėžtus paketus, taip pat pašalinamos ciklinės priklausomybės (pavyzdys 35).



35 pav. Galutinė *azure-storage-blob-cryptography* sistemos struktūra

Pakartotinai įvertinus matoma, kad nuokrypis nuo pagrindinės sekos šiek tiek padidėjo, tačiau vis dar yra gan mažas, ciklinės priklausomybės išnyko, todėl galime teigti, kad *skirstymo pagal smulkų funkcionalumą*, *atskiro pagalbinių klasių paketo* ir *skirtingų versijų grupavimo į paketus* šablonai prisidėjo prie bendro sistemos suprantamumo, įvedė aiškesnę jos struktūrą, nesukuriant sistemos kokybės problemų, aptinkamų paketų kokybės matuose (pavyzdys 36).

Paketo vardas	$N$	$Af$	$Ef$	$S$	$A$	$D$	$C$
specialized.cryptography	3	1	8	0.889	0.0	0.111	0
specialized.cryptography.blob	6	3	5	0.625	0.0	0.375	0
specialized.cryptography.encryption.data	4	8	1	0.111	0.0	0.889	0
specialized.cryptography.encryptor.v2	1	1	3	0.75	0.0	0.25	0
specialized.cryptography.encryptor.v1	1	1	3	0.75	0.0	0.25	0
specialized.cryptography.encryptor	1	4	2	0.333	1.0	0.333	0
specialized.cryptography.decryptor.v2	1	1	3	0.75	0.0	0.25	0
specialized.cryptography.decryptor.v1	1	1	3	0.75	0.0	0.25	0
specialized.cryptography.decryptor	2	4	2	0.333	0.5	0.167	0
specialized.cryptography.util	3	6	0	0.0	0.0	1.0	0
Vidurkiai	$\bar{N}$	$\bar{A}$	$\bar{E}$	$\bar{S}$	$\bar{A}$	$\bar{D}$	$\Sigma C$
Pirmas pertvarkymas	3	3	3	0.561	0.188	0.335	6
Galutinis variantas	2 (-1)	3	3	0.529 (-0.032)	0.15 (-0.03)	0.388 (+0.053)	0 (-5)

36 pav. *azure-storage-blob-cryptograph* paketų kokybės matai po visų pertvarkymų

Pavyzdyje 37 matomos pertvarkytos sistemos, jos rasto problemos ir šablonų, panaudotų problemų išsprenimui sąrašas. Pertvarkytų programinis kodas yra pasiekiamas viešojoje *GitHub*<sup>20</sup> repozitorijoje.

Sistema	Išspręstos problemos	Panaudoti šablonai
crawler4j	dideli paketai su skirtingomis atsakomybėmis, neaiški struktūra sąsajos įgyvendinimams saugoti, ciklinės priklausomybės	skirstymas pagal smulkų funkcionalumą, įgyvendinimų atskyrimas
azure-storage-blob-cryptography	dideli paketai su skirtingomis atsakomybėmis, nėra pagalbinių klasių paketų, skirtingos tos pačios esybės versijos	skirstymas pagal smulkų funkcionalumą, atskirų versijų skirstymas, atskiras pagalbinių klasių paketas

37 pav. Pertvarkytos sistemos ir jų panaudoti šablonai

<sup>20</sup>[https://github.com/MartynaUb/bachelor-thesis-analysis-of-code-packaging-patterns/tree/main/code/refactored\\_systems](https://github.com/MartynaUb/bachelor-thesis-analysis-of-code-packaging-patterns/tree/main/code/refactored_systems)

## Rezultatai

1. Išanalizavus atviro kodo kompiuterines sistemas, išskirti kodo skirstymo šablonai projektavimo metu kylančioms problemoms spręsti.
2. Sukurta programa, išvedanti bendrą informaciją apie sistemą bei matų įverčius.
3. Pagal šablonus pertvarkytos sistemos, šablonai įvertinti pagal pasirinktus matus.
4. Pateiktos rekomendacijos šablonų taikymui

## **Išvados**

1. Skirstant klases į paketus pagal dalykinę sritį gali iškilti situacijų, kurioms šis metodas nėra pakankamas.
2. Problemų, kurias sudėtinga išspręsti skirstant kodą į paketus pagal dalykinę sritį, sąrašas nėra baigtinis, tačiau aprašytos problemos pasikartoja ne vienoje sistemoje, ir yra sprendžiamos skirtingais būdais.
3. Rekomenduojami naudoti šablonai turi teigiamos įtakos vertinant pagal matus, siejamus su lengvesniu sistemos suprantamumu bei palaikomumu.



## Šaltiniai

- [Aga15] B. Aga. A Tool for Breaking Dependency Cycles Between Packages. 2015 [žiūrėta 2024-04-04]. Prieiga per internetą: <https://scg.unibe.ch/archive/masters/Aga15a.pdf>.
- [Ala07] C. B. Alan MacCormack John Rusnak. The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry. 2007 [žiūrėta 2024-03-04]. Prieiga per internetą: [https://www.hbs.edu/ris/Publication%20Files/08-038\\_187f1243-7d1e-464a-bddd-bf7c09873284.pdf](https://www.hbs.edu/ris/Publication%20Files/08-038_187f1243-7d1e-464a-bddd-bf7c09873284.pdf).
- [Ale23] Y. Alemayehu. The Power of Utility Classes in Dart: A Comprehensive Guide. 2023 [žiūrėta 2024-05-15]. Prieiga per internetą: <https://medium.com/@yetesfadev/the-power-of-utility-classes-in-dart-a-comprehensive-guide-f1dcab5edd7>.
- [Eli10] M. Elish. Exploring the Relationships between Design Metrics and Package Understandability: A Case Study. 2010 [žiūrėta 2024-03-04]. Prieiga per internetą: [https://www.researchgate.net/publication/221219583\\_Exploring\\_the\\_Relationships\\_between\\_Design\\_Metrics\\_and\\_Package\\_Understandability\\_A\\_Case\\_Study](https://www.researchgate.net/publication/221219583_Exploring_the_Relationships_between_Design_Metrics_and_Package_Understandability_A_Case_Study).
- [Eva03] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2003.
- [Jan21] M. Jang. Two Different Ways To Package Your Code. 2021 [žiūrėta 2024-04-04]. Prieiga per internetą: <https://medium.com/ryanjang-devnotes/two-different-ways-to-package-your-code-e9cb12d1b6ea>.
- [Kle17] M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.
- [Mar02] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002.
- [Sad21] M. Sadowski. Divide Your Codebase by Domains and Features To Keep It Scalable. 2021 [žiūrėta 2024-05-04]. Prieiga per internetą: <https://betterprogramming.pub/divide-code-by-domains-and-features-and-keep-it-scalable-bb5bd66cf3d2>.
- [Sho19] M. A. Shouki A. Ebad. Investigating The Effect of Software Packaging on Modular Structure Stability. 2019 [žiūrėta 2024-03-04]. Prieiga per internetą: [https://www.researchgate.net/publication/348654540\\_Investigating\\_the\\_Effect\\_of\\_Software\\_Packaging\\_on\\_Modular\\_Structure\\_Stability](https://www.researchgate.net/publication/348654540_Investigating_the_Effect_of_Software_Packaging_on_Modular_Structure_Stability).
- [Var12] J. K. C. Varun Gupta. Package level cohesion measurement in object-oriented software. 2012 [žiūrėta 2024-05-04]. Prieiga per internetą: <https://link.springer.com/content/pdf/10.1007/s13173-011-0052-4.pdf>.

# Priedai

## Priedas nr. 1

### Nuorodos į nagrinėtas atviro kodo sistemas

- <https://github.com/brettwooldridge/HikariCP>
- <https://github.com/h2database/h2database>
- <https://github.com/Meituan-Dianping/Leaf>
- <https://github.com/OpenHFT/Chronicle-Map>
- <https://github.com/ReactiveX/RxJava>
- <https://github.com/yasserg/crawler4j/tree/master>
- <https://github.com/google/guice>
- <https://github.com/Azure/azure-sdk-for-java/tree/main/sdk/storage/azure-storage-blob-cryptography>
- <https://github.com/elastic/elasticsearch>
- <https://github.com/apache/incubator-seata>
- <https://github.com/mongodb/mongo>
- <https://github.com/nocodb/nocodb>
- <https://github.com/mariazevedo88/travels-java-api>
- <https://github.com/AdoptOpenJDK/openjdk-jdk11>
- <https://github.com/keystonejs/keystone>
- <https://github.com/wix-incubator/source-wizard>
- <https://github.com/dbeaver/dbeaver>
- [https://github.com/hackjutsu/Fire\\_Sticker](https://github.com/hackjutsu/Fire_Sticker)
- <https://github.com/federicoiosue/Omni-Notes>