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

Gerai 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 patikima, suprantama ir lengviau palaikoma. Rekomenduojamas naudoti kodo skirstymo į paketus pagal dalykinės srities esybes 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 projektavimo 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 dalykinės srities esybes metodą ir spręsti projektavimo metu kylančias problemas.

Raktiniai žodžiai: Kodo skirstymas į paketus, Kodo skirstymo į paketus 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 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 and to recommend the most effective approach. To achieve this, a set of metrics evaluating readability, maintainability and stability were proposed.

Keywords: Code packaging, Code packaging patterns

Turinys

Įvadas

Gerai suprojektuota kompiuterinė sistema yra vienas iš kritinių sėkmingo ją naudojančio verslo elementų. Tam, jog kompiuterines sistemas naudojantis verslas išlaikytų stabilų augimą, yra būtina sukurti sistemą, kuri sumažintų atotrūkį tarp organizacijos tikslų ir jų įgyvendinimo galimybių. 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 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 našumui, 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 sluoksnį*, kur kiekvienam funkcionalumui arba kompiuterinės sistemos sluoksniui yra sukuriamas paketas, arba pagal *dalykinės srities esybes*, kur vienos esybės kodas, dalykinės srities esybės funkcionalumas skirtingose programiniuose sluoksniuose yra patalpintas viename pakete [Jan21]. Dažniausiai, prioritetas teikiamas kodo skirstymui pagal dalykinės srities esybes [Eva03] dėl lengvesnio skaitomumo, palaikomumo, mažesnės sankibos [Sad21]. Skirstymas pagal dalykinės srities esybes taip pat gali padidinti enkapsuliaciją naudojant prieigos modifikatorius *Java* programavimo kalboje.

Tačiau šis 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. 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. Šiame darbe minint *šabloną kodo skirstymui į paketus* turima omenyje taisykles, nurodančias, kaip grupuoti klases į paketus, sprendžiant iškilusią problemą bei užtikrintant nuoseklų stilių.

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

Tikslui pasiekti yra iškeliami 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 strukūrą

- Naudojant pertvarkytas sistemas, įvertinti kiekvieną šabloną kodo skirstymui į paketus pagal pasiūlytus kriterijus
- Pateikti rekomendacijas, ar aprašyti šablonai kodo skirstymui tinkami naudoti

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. Ieškomi kriterijai, kuriuos naudojant galima įvertinti kodo suskirstymo įtaką sistemos kokybei, pavyzdžiui - komponentų skaičius, tiesioginės ir netiesioginės priklausomybės, paketų stabilumas [Mar02]. Nagrinėjamos atviro kodo sistemos, pasirenkant skirtingo tipo projektus, siekiant objektyvesnės šablonų analizės skirtingose srityse. Galimi 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

Tyrinėjamų projektų paketų sturktūros pertvarkomos pagal pasirinktus skirstymo šablonus, pertvarkyti projektai įvertinti naudojant išskirtus kriterijus, nustatant, kokią įtaką skirtingi skirstymo šablonai turi sistemos kokybei.

Likusi šio dokumento dalis yra išdėstyta taip: pirmas skyrius tyrinėja praktikoje naudojamus šablonus klasėms į paketus skirstyti. Antrame skyriuje analizuojami išskirti šablonai. Trečias skyrius nagrinėja tvarkingos kompiuterinės sistemos sąvoką, kas ją sudaro, įgyvendinimo kokybę ir kaip galima ją įvertinti, aprašyti matai, vertinantys paketų struktūros įtaką sistemos kokybei. Ketvirtame skyriuje aprašomi sukurti įrankiai, reikalingi sistemų analizei ir šablonų įvertinimui. Penktame skyriuje analizuojamos pasirinktos atviro kodo sistemos - bandoma nustatyti jų naudojamus šablonus, vertinama sistemų kokybė. Šeštame skyriuje aprašomas procesas, kaip pasirinktos sistemos yra pertvarkomos, kad laikytųsi pirmame skyriuje aprašytų kodo skirstymų šablonų, įvertinama, kiek sudėtinga pasiekti kiekvieno šablono struktūrą. Nagrinėjamas pertvarkytų sistemų atitikimas trečiame skyriuje aprašytiems kriterijams.

1. Kodo skirstymo į paketus šablonai

Renkantis kodo skirstymo į paketus metodą, dažniausiai rekomenduojamas skirstymas pagal dalykinės srities esybes. 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, buvo nagrinėjamos atviro kodo sistemos, stebimi nukrypimai nuo bendresnio kodo skirstymo būdo ir identifikuojami klausimai arba problemos, kurias jais buvo bandoma išspresti.

1.1. Sistemų analizės procesas

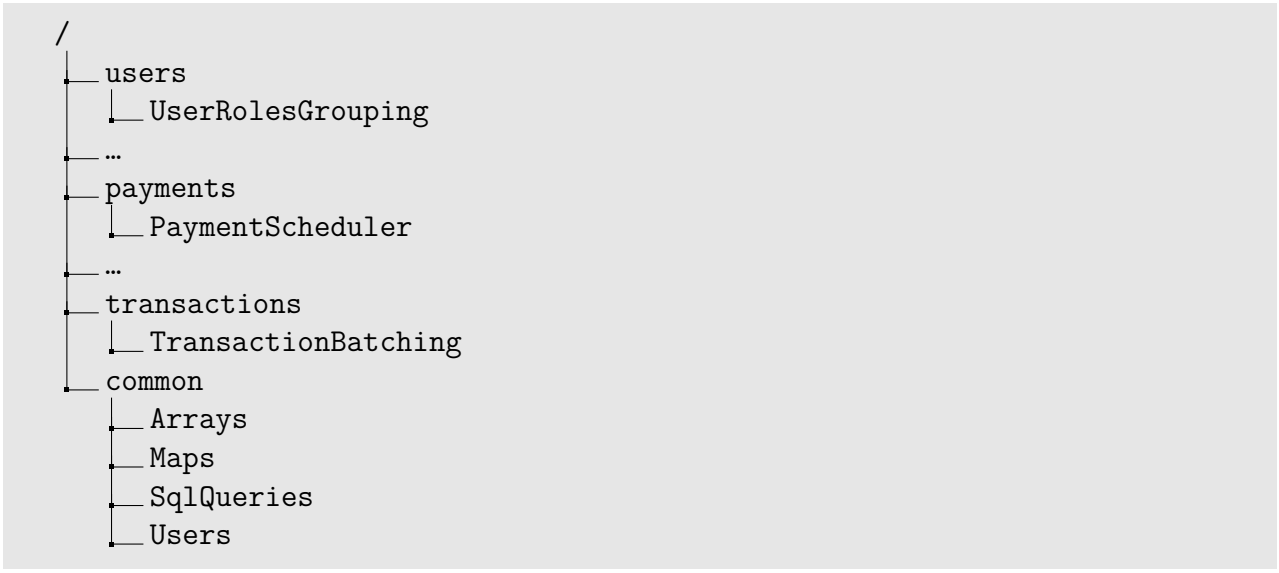
Praktikoje naudojamų šablonų paieška buvo atliekama stebint atviro kodo sistemų struktūrą ir bandant nustatyti, kodėl pasirenkama nukrypti nuo pagrindinio skirstymo būdo. Nagrinėtos įvairių tipų sistemos - tiek taikomoji, tiek techninė programinė įranga. Pastebėta, kad nukrypimai nuo pagrindinio skirstymo būdo dažnai pasikartodavo bandant išspręsti tas pačias problemas, tačiau jų sprendimo būdai gali būti skirtingi.

1.2. Problemos

Šiame skyriuje pateikiamos analizuotose sistemose rastos problemos, kartu su jas sprendžiančiais šablonais ir pavyzdžiais, kokiose sistemose jie yra.

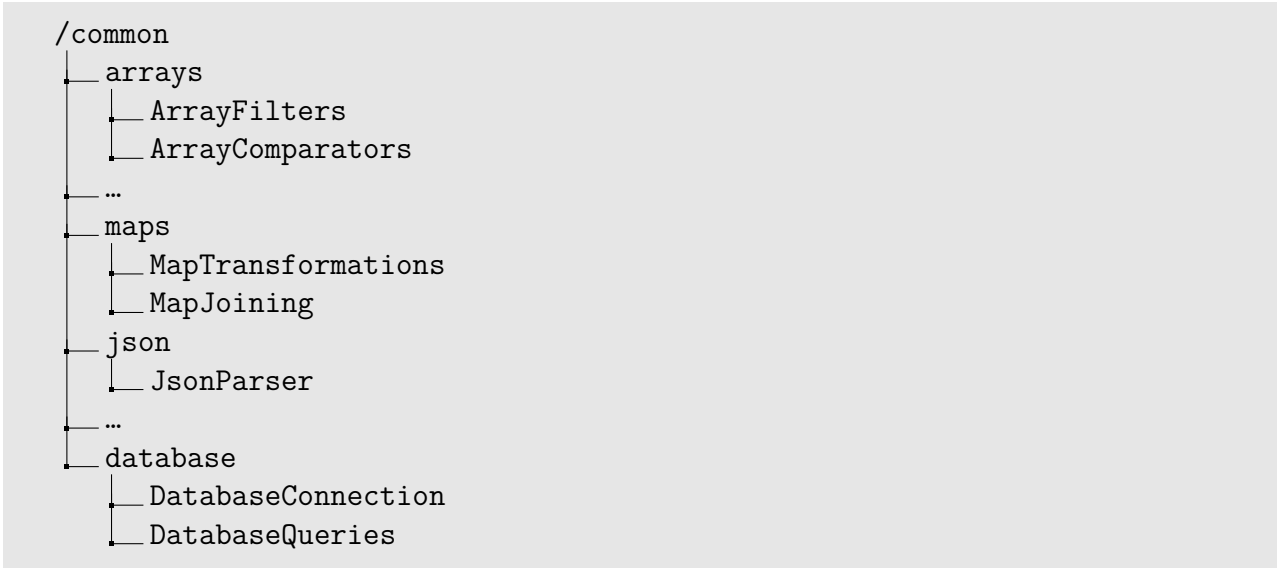
1.2.1. Pagalbinių, daugkartinio naudojimo klasių skirstymas

Pagalbinės klasės galėtų būti apibrėžiamos kaip tiesiogiai su domeno 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 dalykinės srities esybes - pagalbines klases gali naudoti kelios esybės, todėl neaišku, prie kurios jas reikėtų priskirti. Vienas iš šablonų, sprendžiančių šią problemą - turėti vieną paketą, skirtą visoms pagalbinėms klasėms, kuris yra paminėtas sistemos dokumentacijoje ir apie jo egzistavimą teoriškai žino visi komandos nariai. Pakete galima turėti atskiras klases kiekvienai bendrinei dalykinei sričiai, iš kurios pavadinimo programuotojas galėtų nuspresti, kad jo ieškomas funkcionalumas bus būtent toje klasėje.



1 pav. Sistemos pavyzdys, kur visas bendrinio panaudojimo kodas guli *common* 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 pagalbines klases, susijusias su ta dalykine sritimi.



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*[1] bei *Fire Sticker*[2]. *Omni-Notes* atveju, *utils* pakete esančios klasės papildomai skirstomos pagal atitinkamą dalykinę sritį. Šio šablono aprašymas -

[1] `https://github.com/federicoiosue/Omni-Notes/tree/develop/omniNotes/src/main/java/it/feio/android/omninotes/utils`

[2] `https://github.com/hackjutsu/Fire_Sticker/tree/master/app/src/main/java/com/gogocosmo/cosmoqiu/fire_sticker/Utils`

Šablonas	Atskiras pagalbinių klasių paketas
Sprendžiama problema	Pagalbinių klasių skirstymas
Siekiamybė	Mažesnė kodo duplikacija, 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 dengiamą funkcionalumą

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*[3]. Čia pagalbinės klasės yra *utils* pakete žemesniame sistemos lygmenyje, pavyzdžiui, po *engine* paketu.

```
/engine
├── strategy
├── sequence
├── …
└── utils
    ├── ExceptionUtils
    └── …
```

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

Šio šablono aprašymas -

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

1.2.2. Didelis klasių skaičius pakete

Sistemai plečiantis, neišvengiamai didėja ir klasių skaičius. Net pasirinkus tinkamą klasių skirstymo į paketus metodą ir palaikant šią tvarką, klasių skaičius pakete gali išaugti. Tokį pavyzdį galima matyti paieškos įrankio *Elasticsearch*[4] *transport* pakete. Nors klasės yra sugrupuotos pagal dalykinės srities esybę, bendras jų skaičius pakete siekia beveik devyniasdešimt. Tokiame pakete

[3] https://github.com/apache/incubator-seata/tree/2.x/saga/seata-saga-engine/src/main/java/org/apache/seata/saga/engine/utils

[4] https://github.com/elastic/elasticsearch/tree/main/server/src/main/java/org/elasticsearch/transport

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 sudetingesnis, kadangi yra paveikiama daugiau klasių. Dažnai matoma, kad susidarius šiai situacijai, nukrypstama nuo skirstymo pagal dalykinės srities esybes, pradedamos grupuoti techninio sluoksnio klasės. Tokį skirstymą galima matyti *DBeaver* duomenų bazės įrankyje [5] - Aukštesniame lygmenyje grupuojant pagal dalykinės srities esybes, pavyzdžiui, *MsSql*, vėliau pereinama prie skirstymo pagal techninį sluoksnį pakete *model*. Šio šablono aprašymas -

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

Kitą skirstymo būdą galima stebėti *IntelliJ Source Wizard* įskiepio kode [6]. Čia *plugin* paketo esybė yra papildomai išskirstyta pagal *IntelliJ* sistemos domeno esybes. Vadovaujantis šiuo šablonu, klasės skirstomos pagal smulkesnį jų teikiamą funkcionalumą - *actions*, *extensions*, *ui*, šiame kontekste yra ne techniniai sluoksniai, o *Intellij* įskiepio domeno dalis.

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

4 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 kiekvienas modulis (šios kalbos paketo atitikmuo) turi *index.ts* failą, veikiantį kaip sąsaja, apibrėžianti, kokios modulio klasės bei funkcijos gali

[5] `https://github.com/dbeaver/dbeaver/tree/devel/plugins/org.jkiss.dbeaver.ext.mssql/src/org/jkiss/dbeaver/ext/mssql`

[6] `https://github.com/wix-incubator/source-wizard/tree/master/src/main/kotlin/plugin`

būti pasiekiamos už paketo ribų. Toki šabloną naudoja turinio valdymo sistema *Keystone*[7], kur kiekviena esybė turi savo modulį, atliekantį vieną funkciją, kurios kontraktas aprašytas *index.ts* faile.

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

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

Šio šablono aprašymas -

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

1.2.3. Skirtingų sąsajų implementacijų skirstymas

Sistemai plečiantis galima susidurti su problema, kad išauga sąsajos įgyvendinimų skaičius. Ši problema aktuali skirstant klases pagal dalykinės srities esybes - 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[8], kur visos standartinės sąsajos bei jų implementacijos yra tame pačiame pakete.

[7] https://github.com/keystonejs/keystone/tree/main/packages/core/src/fields/types

[8] https://github.com/AdoptOpenJDK/openjdk-jdk11/tree/master/src/java.base/share/classes/java/util

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

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

Šio šablono aprašymas -

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

Kitas galimas sprendimo būdas - sąsaja, kuri turi daug skirtingų įgyvendinimų, turėtų turėti specifiškai jai sukurtą paketą, o kiekvienam paketo įgyvendinimui sukuriama po subpaketą, kuriame yra tiek klasę įgyvendinanti sąsają, tiek kitos, įgyvendinimui reikalingos klasės. Šablone aprašyta paketų struktūra buvo sutikta tokiose sistemose kaip *HikariCP*[9], skirtoje efektyviam *Java* programos prisijungimui prie duomenų bazių, bei *Leaf*[10], naudojamoje unikalaus identifikatoriaus generavimui.

```
/
├── segment
│   ├── dao
│   ├── model
│   └── SegmentIdGenImpl
├── snowflake
│   ├── SnowflakeIdGenImpl
│   └── SnowflakeZookeeperHolder
└── IdGen
```

7 pav. *Leaf* sistema, kur kiekvienas sąsajos įgyvendinimas yra aprašytas subpakete

Šio šablono aprašymas -

[9] https://github.com/brettwooldridge/HikariCP/tree/dev/src/main/java/com/zaxxer/hikari

[10] https://github.com/Meituan-Dianping/Leaf/tree/master/leaf-core/src/main/java/com/sankuai/inf/leaf

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

1.2.4. Esybių pokyčių ir versijavimo valdymas

Sistemai 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
└── ...
```

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

Toks skirstymo būdas matomas keliose repozitorijose - *Mongo*[11] duomenų bazėje, API skirtame kelionių valdymui *travels-java-api*[12] bei duomenų saugojimo įrankyje *nocodb*[13]. Šiose repozitorijose v1, v2 pavadinti paketai saugo skirtingas esybių versijas. Šio šablono aprašymas -

Šablonas	Atskirų versijų skirstymas
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

[11] `https://github.com/mongodb/mongo/tree/master/src/third_party/boost/boost/regex`

[12] `https://github.com/mariazevedo88/travels-java-api/tree/master/src/main/java/io/github/mariazevedo88/travelsjavaapi/controller`

[13] `https://github.com/nocodb/nocodb/tree/c7cc1f92fd77f8b5daefceb7148aab4a69cb9b4e/packages/nocodb/src/meta/migrations`

2. Kodo skirstymo į paketus šablonų analizė

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 pagalbinėmis klasėmis - su sistema dirbantys inžinieriai nežino apie jų egzistavimą, todėl jų nenaudoja, tai veda prie didesnio kodo pasikartojimo arba kelių skirtingų to paties pagalbinio funkcionalumo įgyvendinimų. 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 bendrinio panaudojimo kodui
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, padengti jį testais, bei aprašyti dokumentaciją, kaip funkcija turėtų būti naudojama.
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. 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 pagalbines klases po ja dengiamos esybės paketais, kyla kodo pasikartojimo rizika - nepatikrinus skirtingoms esybėms skirtų pagalbinių klasių, sunku žinoti, ar toks funkcionalumas jau buvo įgyvendintas.

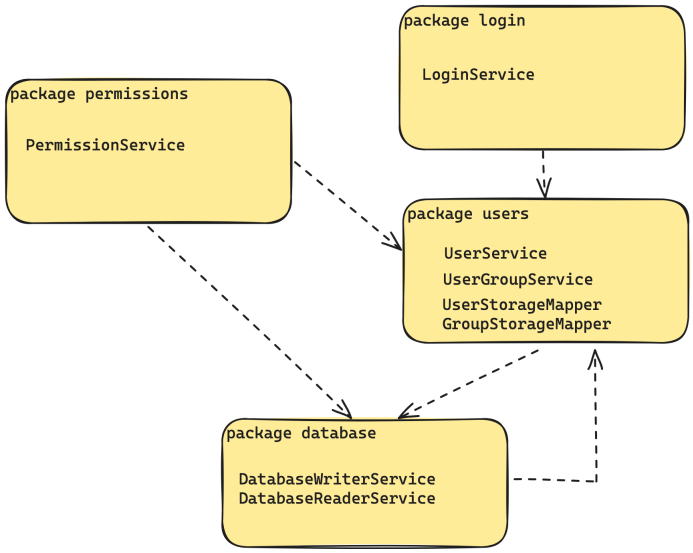
2.0.2. Didelio klasių skaičius pakete skirstymo šablonų analizė

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 sluoksnį 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 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

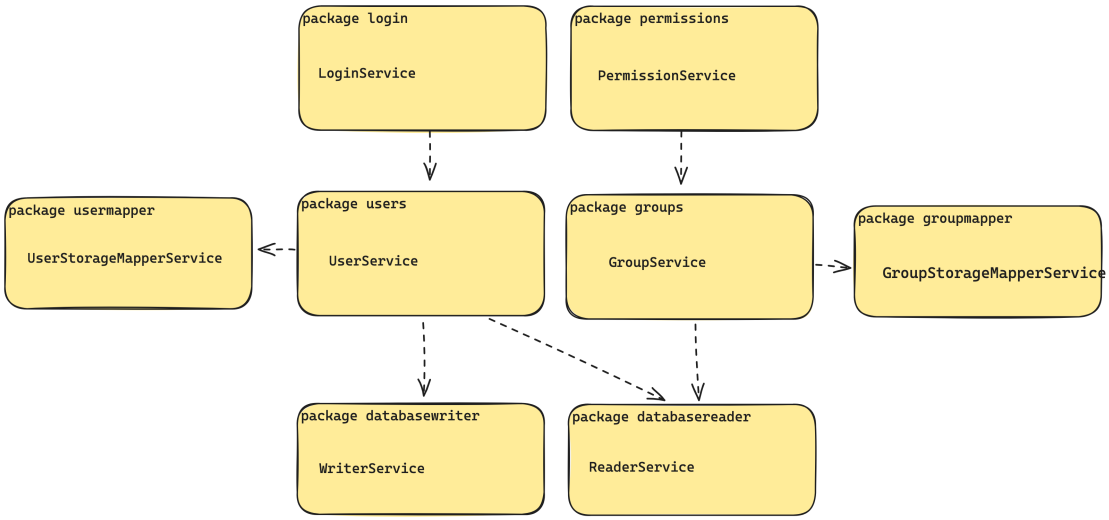
atveju, pavyzdžiui, esybės pagalbinėms klasėms, *orchestrator* tipo klasėms reikėtų kurti papildomus paketus.

Skirstymą pagal smulkų funkcionalumą pagrindžia Robert C. Martin bendro sąryšio principas, kuris teigia, kad visos tarpusavyje susijusios klasės turėtų būti vienam pakete ir akcentuoja siekiamybę turėti gan mažus paketus, turinčius aiškiai apibrėžtą funkcionalumą, priežastį egzistuoti, taip užtikrinant ir glaudų tarpusavyje susijusių klasių saryšį. Šis principas taip pat gali padėti užtikrinti 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 funckionalumus, yra sumažinamas paketų priklausomybių skaičius.



9 pav. Sistemos pavyzdys su kelias funkcijas atliekančiais paketais



10 pav. Sistemos pavyzdys su aiškią, 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 saryšio 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.

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.

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 dalykinės srities esybes - skirstomos versijos gali atspindėti reikalingus palaikyti besikeičiančius verslo reikalavimus.

2.1. Kodo skirstymo į paketus šablonų analizės išvados

Išanalizavus skirtingus kylančias problemas sprendžiančius šablonus matoma, kad pagalbinių klasių skirstymui tinkamesnis naudoti atskiras pagalbinių klasių paketas, didelio klasių skaičiaus pakete skirstymui – skirstymas pagal smulkų funkcionalumą, o skirtingų sąsajų implementacijoms – įgyvendinimų atskyrimą.

3. Kompiuterinės sistemos vertinimas

3.1. Kompiuterinės sistemos kokybė

Išskyrus tinkamesnius naudoti kodo skirstymo į paketus šablonus, reikėtų objektyviai pamatuoti jų įtaką sistemos kokybei ir užtikrinti, kad jie prisideda prie labiau patikimos, suprantamos ir lengviau palaikomos sistemos struktūros. Tam reikėtų apsibrėžti, 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, paslepiant tam tikras klaidas nuo vartotojo [Kle17].
- Prižiūrimumas, reiškiantis, jog skirtingų abstrakcijų pagalba sumažintas sistemos kompleksiškumas. Dėl to nesunku keisti esamą 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 užklausų 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 paketų skirstymo šablonus ir tų šablonų laikytis, atsiperka padarant programinį kodą geriau suprantamu, taip prisidedant prie bendro sistemos patikimumo, plečiamumo ir lengvesnio palaikomumo.

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, enkapsuliuoja paketų vidinius elementus, neleidžiant pakeitimais išplisti už paketų ribų yra pagrindas tvirtai sistemos architektūrai, gebančiai efektyviai plėstis, nepristatant nebūtinų pokyčių ir sutaupant programos priežiūros kaštus.

3.2. Kodo skirstymo į paketus šablonų vertinimas

Ankstesniame skyriuje buvo nagrinėjama gerai įgyvendintos paketų struktūros įtaka gerai kompiuterinės sistemos struktūrai, 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? Tvarkingas, aiškiai suprantamas kodas yra subjektyvi

tema, priklausanti 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 labai subjektyvūs ir patogūs tik metodą formavusiam asmeniui. Tam, kad būtų galima pagrįstai įvertinti skirtingus kodo skirstymo šablonus, pasiekiant kuo objektyvesnį, įtaką sistemos kokybei 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 teisingai grupuoti klases į paketus. Rodiklis, kiek kiekvienas paketas sistemoje laikosi 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 priežastį 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 visada keičiasi kartu, tada 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 žiedinių 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 turetų 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 pasirintame 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 stabilių priklausomybių principe:

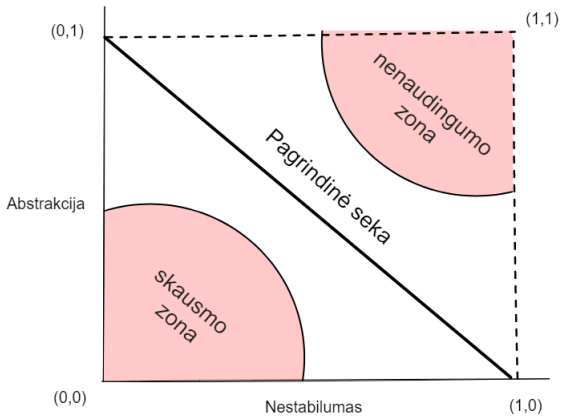
$$Stabilumas(S) = \frac{Jungtys_{eferentines}}{Jungtys_{eferentines} + Jungtys_{aferentines}} \quad (1)$$

  Reikšmės rėžiai - nuo nulio iki vieno, kur nulis nurodo visiškai stabilų paketą, o vienetas - 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 vienetas 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).



11 pav. Pagrindinės sekos kreivė

Atstumą nuo pagrindinės sekos galima apskaičiuoti kaip:

$$Atstumas = |Abstracija + Nestabilumas - 1| \tag{3}$$

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 principe.

Šių matų patikimumas buvo ivertintas atvejo analizėje *Exploring the Relationships between Design Metrics and Package Understandability: A Case Study*, 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 statistiškai reikšmingą koreliaciją tarp daugumos matų ir paketų suprantamumo [Eli10].

4. Įrankiai šablonų analizei ir įvertinimui

Kompiuterinės sistemos, kurioms yra aktualu klasių ir paketų skirstymo metodai, paprastai yra labai didelės. Pilnai perprasti tokias sistemas, 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, figūruojančių paketų bei klasių vardus. Ši papildoma informacija nėra aiškūs teiginiai, o tik pagalba analizę atliekančiams asmeniui, leidžianti priimti išvalgas apie sistemą, kaip pavyzdžiui, kokiam paketų skirstymo šablonui yra atimiausia sistemos strukūra, 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.

Abiejose įrankiuose vienas iš palaikomų išvesties formatų turėtų būti *latex*, taip suteikiant galimybę analizės rezultatus pateikti tolesniame šio dokumento 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 ši procesą – suvienodinti procesų vykdymą bei gautus rezultus, visi įrankiai, reikalingi analizei, įgyvendinti kaip viena programa, kuri apdoroja failus nurodytoje direktorijoje į analizuojamą sistemą – nuskaito *Java* failų turinį ir išveda informaciją apie sistemos paketus bei klases. Surinkta informacija naudojama įgyvendinti kiekvienam aprašytam įrankio funkcionalumui, ten, kur galima, naudojant jau parašytus įrankius, taip programiškai supaprastinant skirtingų įrankių vykdymą. Gauti atliktos analizės rezultatai išvedami į failus nurodytoje direktorijoje. Rezultatus sudaro apdorotos sistemos paketų kokybės matai, paketų struktūros medis, paketų priklausomybės grafikas. Gauti rezultatai išvesti *latex* formatu, todėl gali būti pridedami į darbo dokumentą, taip padarant sistemų analizę greitesne bei efektyvesne.

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 išnagrinėti kelias sistemas pasitelkus paketų kokybės matus bei bendrą sistemos struktūros analizę, taip identifikuojant programiniam kodui būdingas problemas bei rastas problemas išspręsti pritaikant aprašytus šablonus. Atlikus pertvarkymus sistemose, dar kartą paskaičiuoti paketų kokybės matus, taip gaunant įrodymus, ar gauti šablonai yra efektyvūs ir iš tiesų sprendžia jiems priskirtas problemas.

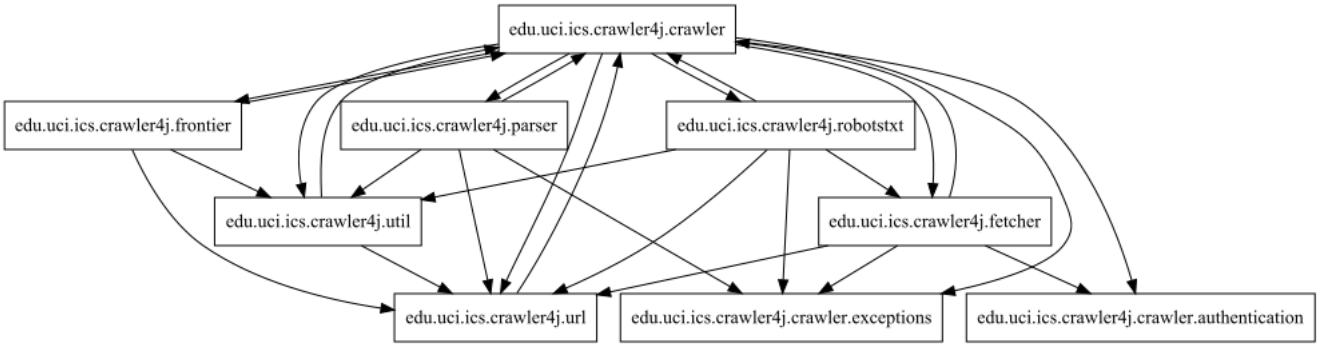
5.1. Sistemų pasirinkimas

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 ivairovę ir objektyvesnius įvertinimus. Per visą pasirinktų sistemų imtį yra sutinkamos visos aprašytos problemos, taip įvertinant visus pasirinktus šablonus.

5.2. *crawler4j* sistema

crawler4j[14] yra atvirojo kodo žiniatinklio tikrinimo (angl. *crawling*) programa parašyta *Java* programavimo kalba, leidžianti efektyviai tikrinti žiniatinklį naudojant daugiagijį anglmultithreaded metodą. Tai taikomosios programinės įrangos tipas, skirtas vartotojams.

Prieš visus pakeitimus *crawler4j* sistemos paketų struktūra atrodo taip:



12 pav. *crawler4j* sistemos struktūra

Atlikus sistemos kokybės matų analizę galima pamatyti sistemai būdingas problemas:

[14] `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

$\bar{N}$	$\bar{A}f$	$\bar{E}f$	$\bar{S}$	$\bar{A}$	$\bar{D}$	ΣC
5	3	3	0.455	0.069	0.476	5

Matų lentelės paaiškinimas:

- N - matas žymintis klasių skaičių pakete.
- Af - aferentinės jungtys. Nurodo, kiek kitų paketų priklauso nuo pasirinkto paketo.
- Ef - eferentinės jungtys. Nurodo, nuo kiek kitų paketų priklauso pasirinktas paketas.
- S - stabilumas. Rodo paketo atsparumą pokyčiams, reikšmės rėžiai - nuo nulio iki vieno, nulis visiškai stabilus paketas, o vienetas - visiškai nestabilus.
- A - nurodo paketo abstraktumą, reikšmės rėžiai - nuo nulio iki vieno, nulis reiškia, kad paketas neturi abstrakčių klasių, o vienetas nurodo, kad pakete yra tik abstrakčios klasės.
- D - Atstumas nuo pagrindinės sekos arba atstumas nuo sąryšio tarp stabilumo ir abstrakcijos, reikšmės rėžiai - nuo nulio iki vieno, nulis reiškia paketą, sutampantį su pagrindine seka,vienas - paketą, maksimaliai nutolusį nuo pagrindinės sekos, siekiama reikšmė - 0.
- C - ciklinių priklausomybių matas, rodo ciklines priklausomybės, siekiama reikšmė - 0.

Kaip skaičiuojami matai aprašyta paketų kokybės matų skyriuje.

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 daug sąsajų su skirtingais jų įgyvendinimais, todėl yra sunku rasti visus specifinės sąsajos įgyvendinimus.
- **Problema numeris 2** - paketo *edu.uci.ics.crawler4j.parser* viduje yra 12 klasių, todėl sunku suprasti, už kokius funkcionalumus jis atsakingas ir kada jį reikėtų nagrinėti bei keisti.
- **Problema numeris 3** - Sistema turi daug ciklinių prilausomynbių, kur du arba daugiau paketų priklauso vienas nuo kito. Tai matoma sistemos paketų diagramoje12, 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.

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

13 pav. *parser* paketo struktūra

Šiam paketui sutvarkyti taikomas *įgyvendinimų atskyrimo* šablonas - sąsaja iškeliama į atskirą paketą *parse*, jos įgyvendinimai perkeliami į minėto paketo subpaketus. Šie pertvarkymai pakeičia *parser* paketo struktūrą:

```
/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
```

14 pav. *parser* paketo struktūra po pirmojo pertvarkymo

Dėl atskirtos sąsajos ir jos įgyvendinimų tampa daug 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 sitemos matus matome, kad sistema tapo aiškesnė, sumažėjo vidutinis klasių skaičius paketuose, nežymiai sumažėjo vidutinis stabilumas, bet pakilo abstrakcijos lygis, todėl atstumas nuo pagrindinės sekos sumažėjo.

crawler4j.parser	N	A	E	S	A	D	C
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	$\bar{N}$	$\bar{A}f$	$\bar{E}f$	$\bar{S}$	$\bar{A}$	$\bar{D}$	ΣC
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

Matų pokyčio 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 vadovaujamasi *skirstymo pagal smulkų funkcionalumą* šablonu, užtikrinant, kad kiekvienas paketas teikia vieną, aiškiai apibrėžtą funkciją, kuri yra pasiekiama per pakete aprašytą sąsają. Paketo vidus yra paslėptas su *Java* kalbos pasiekiamumo modifikatoriais - konkrečių klasių negalima inicializuoti už paketo ribų, nes jų konstruktoriai privatūs.

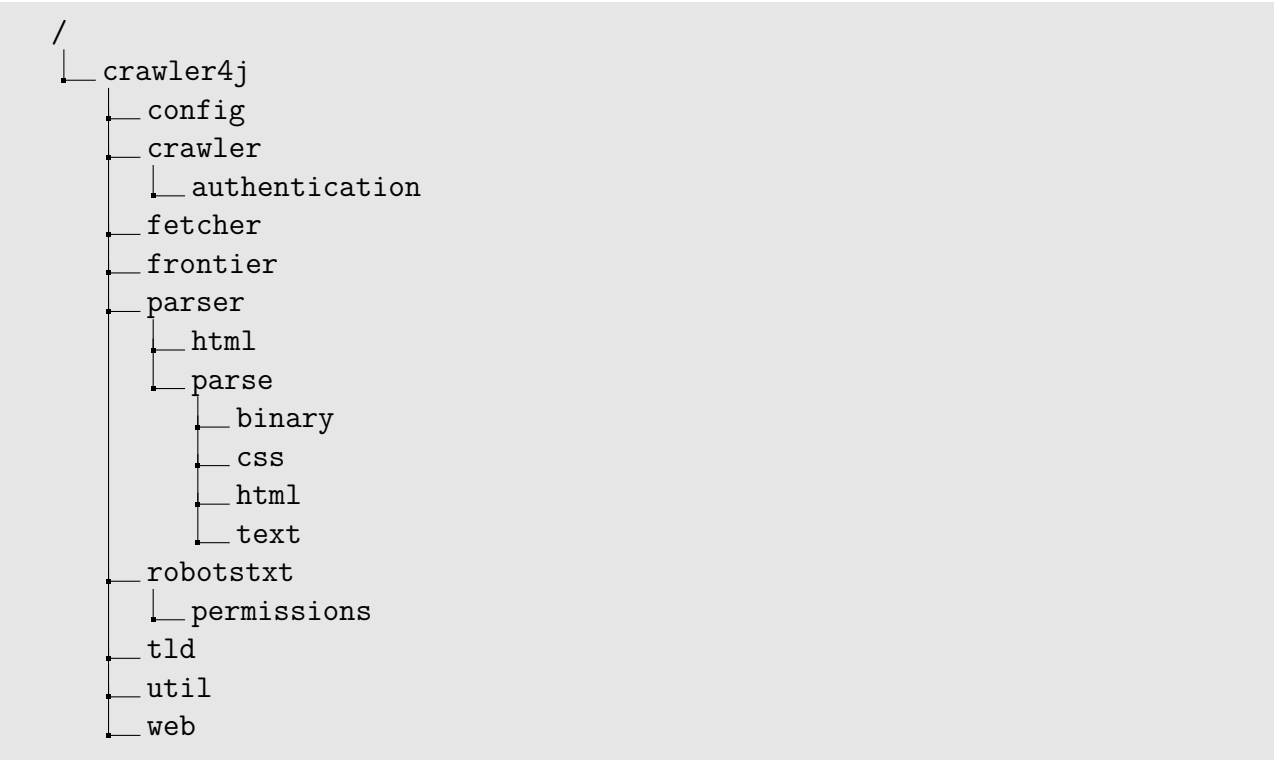
```
public interface HtmlParser {

    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 {
      ...
    }
```

15 pav. *skirstymo pagal teikiamą funkcionalumą* šablono sąsaja

Peržiūrėjus paketus su ciklinėmis priklausomybėmis, iš jų buvo išskirti šie nauji funkcionalumai, kuriems buvo sukurti atskiri paketai:

- 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.

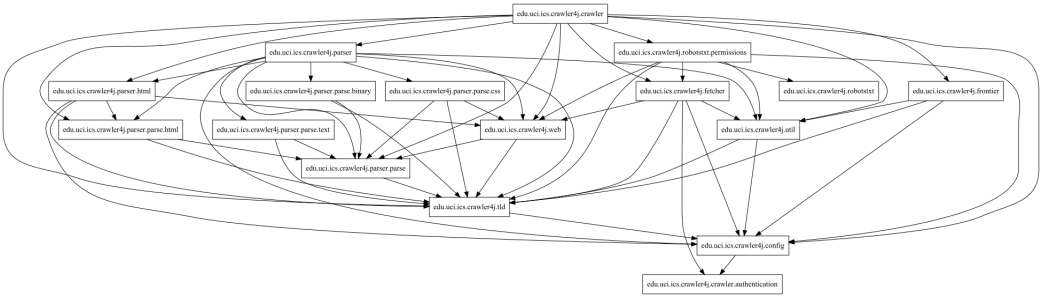
16 pav. *crawler4j* paketų medis prieš iškeliant minėtus funkcionalumus



17 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 daug aiškesnės:

5.3. *Crawler* sistema



18 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:

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	$\bar{N}$	$\bar{Af}$	$\bar{Ef}$	$\bar{S}$	$\bar{A}$	$\bar{D}$	ΣC
Prieš	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)

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.

5.4. *azure-sdk-for-java* sistema

azure-sdk-for-java[15] yra atviro kodo biblioteka, parašyta *Java* programavimo kalba, leidžianti programiškai bendrauti su *Microsofl 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-cryptograph*, kuri atsakinga už duomenų kriptografiją komunikacijos su *Azure Blog* nestruktūrizuota failų saugykla metu.

Prieš visus pakeitimus *azure-storage-blob-cryptograph* 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 funcionalumai 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:

```
/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
```

19 pav. *azure-storage-blob-cryptograph* paketų medis su paketais, skirtais esybių versijų valdymui

Šis pertvarkymas pagal *skirtingų versijų grupavimo į paketus* šabloną, atskiria esybės versijas

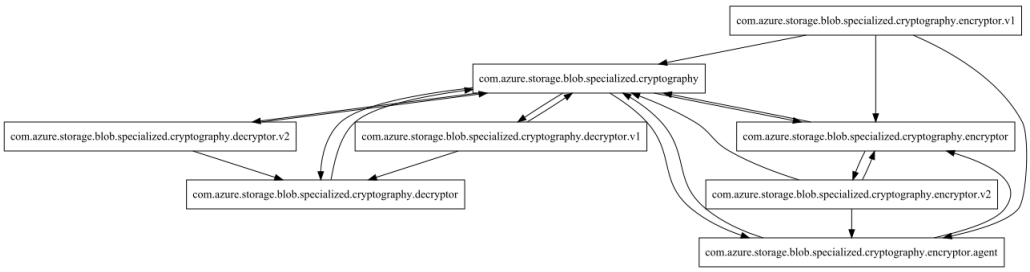
[15] `https://github.com/Azure/azure-sdk-for-java/tree/main/sdk/storage/azure-storage-blob-cryptography`

ir pristato kelis 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

$\bar{N}$	$\bar{A}g$	$\bar{E}g$	$\bar{S}$	$\bar{A}$	$\bar{D}$	ΣC
3	3	3	0.561	0.188	0.335	6

Apskaičiuoti matai rodo nedidelį nuokrypį nuo pagrindinės sekos, todėl galima teigti, kad sistemos abstraktumas bei stabilumas yra subalansuoti, tačiau šis skirstymas turėjo ir neigiamų padarinių - sukelė 6 ciklines priklausomybes. Šiais priklausomybes galima matyti sistemos paketų diagramoje:

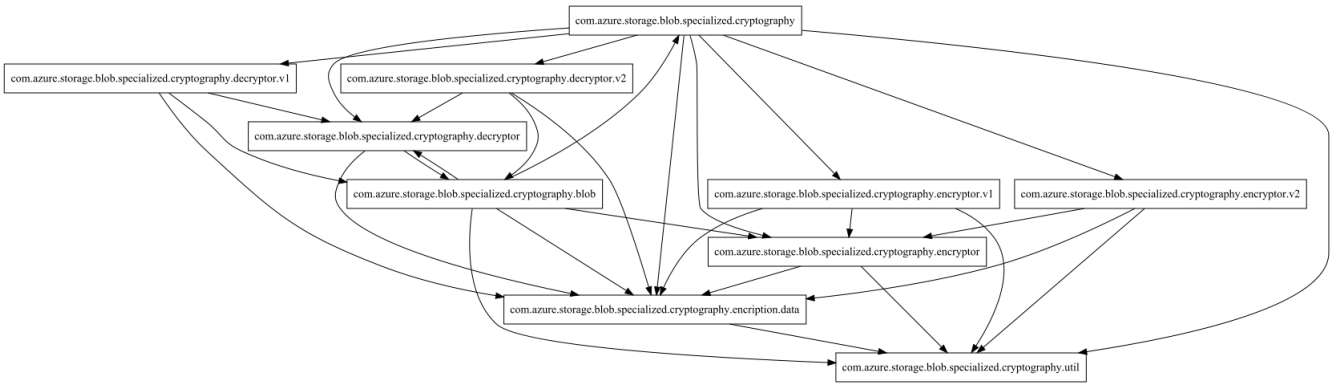


20 pav. *azure-storage-blob-cryptograph* sistemos struktūra po pirmojo pertvarkymo

Norint panaikinti ciklines priklausomybes, reikia panaudoti šabloną *skirstymas pagal smulkų funkcionalumą* ir, išskaidant pagrindinį paketą į dar smulkesnius vienetus - yra sukuriamas *encription.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 funcionalumus 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:



21 pav. Galutinė *azure-storage-blob-cryptograph* sistemos struktūra

Pakartotinai įvertinus matus 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, išlaikant objektyviai gerus sistemos kokybės matus.

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.encription.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	$\bar{N}$	$\bar{A}$	$\bar{E}$	$\bar{S}$	$\bar{A}$	$\bar{D}$	ΣC
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)

Rezultatai

1. Analizuojant industrijoje egzistuojančias kompiuterines sistemas, išskirti kodo skirstymo šablonai projektavimo metu kylančioms problemoms spręsti.
2. Aprašyti gerai įgyvendintos sistemos reikalavimai bei matai jų įvertinimui.
3. Sukurta programa išvedanti informaciją apie sistemą bei matų įverčius.
4. Pagal šablonus pertvarkytos sistemos, šablonai įvertinti pagal išskirtus kriterijus.
5. Iš atrinktų šablonų išskirti geriausiai atitinkantys kriterijus.

Išvados

1. Skirstant kodą pagal dalykinės srities esybes gali iškilti situacijų, kurioms šis metodas nėra pakankamas.
2. Problemų, kurias sudėtinga išspręsti skirstant kodą į paketus pagal dalykinės srities esybes, sąrašas nėra baigtinis, tačiau aprašytos problemos pasikartoja ne vienoje sistemoje, ir yra sprendžiamos skirtingais būdais.
3. Rekomenduojami naudoti šablonai prisideda prie didesnio sistemos našumo, suprantamumo bei palaikomumo.

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.

[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.

[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.

[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.