**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**INFORMATIKOS FAKULTETAS**

**PROGRAMŲ SISTEMŲ INŽINERIJOS KATEDRA**

**Šarūnas Juškevičius**

# PROJEKTAVIMO ŠABLONŲ, BEI ARCHITEKTŪRINIŲ SPRENDIMŲ TYRIMAS IR ANALIZĖ SIEKIANT PROGRAMINĖS ĮRANGOS LANKSTUMO

Magistro darbas

Vadovas
doc. dr. V. Pilkauskas

KAUNAS, 2014

**KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**INFORMATIKOS FAKULTETAS**

**PROGRAMŲ SISTEMŲ INŽINERIJOS KATEDRA**

**TVIRTINU**
**Katedros vedėjas**
**doc. T. Blažauskas**
**2014-05-31**

# PROJEKTAVIMO ŠABLONŲ, BEI ARCHITEKTŪRINIŲ SPRENDIMŲ TYRIMAS IR ANALIZĖ SIEKIANT PROGRAMINĖS ĮRANGOS LANKSTUMO

Informacinių technologijų magistro baigiamasis darbas

Vadovas
doc. dr. V. Pilkauskas
2014-05-20

Recenzentas
prof. dr. R. Butleris
2014-05-20

Atliko
IFM 2/2 gr. stud.
Š. Juškevičius
2014-05-20

KAUNAS, 2014

# AUTORIŲ GARANTINIS RAŠTAS

# DĖL PATEIKIAMO KŪRINIO

**2014 – Gegužės - 23 d.**
**Kaunas**

**Autoriai,**_____ Šarūnas Juškevičius _____
(vardas, pavardė)

_____,

patvirtina, kad Kauno technologijos universitetui pateiktas baigiamasis bakalauro (magistro) darbas (toliau vadinama – Kūrinys)____PROJEKTAVIMO ŠABLONŲ, BEI ARCHITEKTŪRINIŲ SPRENDIMŲ TYRIMAS IR ANALIZĖ SIEKIANT PROGRAMINĖS ĮRANGOS LANKSTUMO_____
(kūrinio pavadinimas)
pagal Lietuvos Respublikos autorių ir gretutinių teisių įstatymą yra originalus ir užtikrina, kad

1) jį sukūrė ir parašė Kūrinyje įvardyti autoriai;
2) Kūrinys nėra ir nebus įteiktas kitoms institucijoms (universitetams) (tiek lietuvių, tiek užsienio kalba);
3) Kūrinyje nėra teiginių, neatitinkančių tikrovės, ar medžiagos, kuri galėtų pažeisti kito fizinio ar juridinio asmens intelektinės nuosavybės teises, leidėjų bei finansuotojų reikalavimus ir sąlygas;
4) visi Kūrinyje naudojami šaltiniai yra cituojami (su nuoroda į pirminį šaltinį ir autorių);
5) neprieštarauja dėl Kūrinio platinimo visomis oficialiomis sklaidos priemonėmis.
6) atlygins Kauno technologijos universitetui ir tretiesiems asmenims žalą ir nuostolius, atsiradusius dėl pažeidimų, susijusių su aukščiau išvardintų Autorių garantijų nesilaikymu;
7) Autoriai už šiame rašte pateiktos informacijos teisingumą atsako Lietuvos Respublikos įstatymų nustatyta tvarka.

**Autoriai**

_____Šarūnas Juškevičius_____ _____
(vardas, pavardė) (parašas)

# ABSTRACT

In today's software development industry flexibility and ability to adapt is crucial. Rapidly changing requirements, new features, and constant bug fixing makes software life cycle such a volatile environment. In addition to all of this most of the software is used for many customers who have their specific requirements thus software have to be flexible enough to fit for everyone or have the ability to be easily modified to do so. Key to software flexibility is within its design. Reasonable architecture can solve many problems before them even happened. Most of software design choices are well refined by experienced software developers and categorized into patterns. Experienced software developer almost always knows in which case what type of software structure to use, but rarely this decision can be justified scientifically. Purpose of this document is to research few of most used design patterns and try to improve them in order to achieve better software flexibility using design and metrics and experimental study.

# SANTRAUKA

Programinės įrangos aplinkoje kiekvienas elementas yra nuolatos besikeičiančioje būsenoje: nuolatiniai techninės įrangos pasikeitimai, besikeičiančios operacinės sistemos, integruotos ar bendradarbiaujančios programinės įrangos pokyčiai, klaidų taisymai, saugumo pažeidimai ir kiti aspektai verčia programinę įrangą nuolatos taikytis ir tobulėti [6]. Priežasčių kodėl reikia tobulinti sistemas yra tiek pat daug kaip ir būdų kaip jas tobulinti. Daugelį sistemų galima patobulinti skirtinguose lygmenyse, tačiau prireikus iš tiesų keisti pačią sistemą svarbu, jog ji būtų pakankamai lanksti ir jos pakeitimai užtruktų kuo mažesnę laiko dalį, bei reikalautų kuo mažiau resursų. Tai galima pasiekti pasitelkiant tinkamą sistemos architektūrą, tačiau iš ties sunku nuspręsti kokius privalumus viena ar kita architektūra turi sistemos išplečiamumo atžvilgiu. Tam bus atliekamas tyrimas, siekiant užtikrinti, jog pasirinkti architektūriniai sprendimai iš ties padidina sistemos lankstumą, bei išplečiamumo galimybes. Pasirinktos architektūros bus tiriamos keliais būdais: lyginamos jų diagramos pasinaudojant diagramų metrikomis, eksperimentiniu būdu realizuojant po vieną naują įrenginį sukurtai sistemai ir tikrinant laiką, atsižvelgiant į programinio kodo metrikas realizavimus šablonus. Sistemos vidus yra tiek pat svarbus kaip ir jo išorė ir todėl nereikėtų apleisti sistemos vidinės struktūros vien todėl, jog ekonominę vertę turi tik toji dalis, kurią mato klientai.

# TURINYS

# PAVEIKSLŲ SĄRAŠAS



# LENTELIŲ SĄRAŠAS

# 1 ĮVADAS

Programinės įrangos kūrimo pramonė nuolatos plečiasi ir tobulėja. Atsiranda vis daugiau ir kompanijų, bei individualių asmenų norinčių įsidiegti tam tikrą, jų reikalavimus tenkinančią programinę įrangą, bei kompanijų ir asmenų galinčių šiuos norus įgyvendinti. Plečiantis tiek paklausai tiek pasiūlai smarkiai keičiasi ir pačios pramonės struktūra. Užsakovai tampa reiklesni, nes žmonių galinčių atlikti jų užsakymą yra ne vienetai. Taip didėja realizacijos reikalavimai laikui, bei kokybei.

Dėl šių priežasčių kompanijos siekia rasti greitą ir efektyvų būdą programinei įrangai kurti, kuris leistų sukurti kuo lankstesnę programinę įrangą, leidžiančią prisitaikyti prie rinkos pasikeitimų su minimaliu kiekiu pakeitimų sistemoje, bei visiškai nepaliečiant arba minimaliai keičiant senus egzistuojančius funkcionalumus. Tokia politika dominuoja daugelyje kompanijų, niekas nenori keisti ar ardyti to kas jau veikia, o naujų funkcionalumų reikalavimai yra dažni ir nesiliaujantys. Tokios sąlygos verčia kompanijas ieškoti tinkamo architektūrinio sprendimo, kaip vienaip ar kitaip laksčiai prijungti naujus funkcionalumus prie egzistuojančios sistemos. Šiuo atveju, sprendimų aibė yra labai vidutinė, negalima teigti, jog nėra sprendimų šiam uždaviniui spręsti, tačiau ir nereikėtų svaidytis teiginiais apie gausybė egzistuojančių sprendimų. Taigi lankstumas ir gebėjimas prisitaikyti, kuriant programinę įrangą yra labai svarbus bruožas šių dienų programinės įrangos kūrimo rinkoje. Todėl savaime suprantama technologijų leidžiančių įgyvendinti šiuos sistemos privalumus paieška yra vienas iš svarbesnių faktorių programinės įrangos kūrime.

Kitas faktorius smarkiai įtakojantis programinės įrangos kūrimo procesą yra šių dienų fenomenas „programinė įranga kiekvienam“. Vis daugiau ir daugiau kompanijų renkasi biznio modelį, kuriame programinė įranga gamina pagal užsakymą, o ne išleidžiama į rinką kaip produktas potencialiai galintis turėti daugybė pirkėjų. Savaime suprantama, jog dirbant pagal tokį modelį dar dažniau tenka taikytis prie specifinių reikalavimų konkrečiam užsakymui ir netgi kuriant, ar tiesiog pritaikant esamą standartinę programinę įrangą, tenka atlikti pakeitimus kurių konkrečiai reikalauja užsakovas. Tokiu būdų net ir pati bendriausia programinė įranga tampa specializuota programine įranga. Žinoma toks modelis tenkina užsakovus, tuo pačiu ir pačias kompanijas. Kita vertus tai atsiliepia programinės įrangos kūrimo procese. Vėlgi iškyla problema kaip susieti pirmąją problemą su antrąją ir rasti joms sprendimą.

Susieti šias problemas ganėtinai nesunku, nes jos iš ties yra panašios ir reikalauja panašių spendimų kuriant programinė įrangą. Šiame dokumente ir pabandysime surasti optimalų sprendimą tokioms problemoms spęsti, bei pritaikyti jį kurtai programinei įrangai,

vadovaujantis užsakovo biznio modelio ir atsižvelgiant į atliktą analizę apie potencialius pokyčius, bei galimas rizikas.

## 1.1 DOKUMENTO PASKIRTIS

Šis dokumentas skirtas aprašyti atliktai sistemos analizei, siekiant išgauti lankstumą ir galimybė prisitaikyti, bei išspręsti iškilusias inžinerines problemas ir suformuluotą sistemos problemą. Inžinerinės problemos neretai užmirštamos dėl suteikiamos perdėtos svarbos galutiniam produktui. Koncentruojantis vien tik į rezultatus per mažai dėmesio skiriama pačiam procesui, o kaip tik proceso tobulinamas ir yra pagrindinis kelias link galutinio produkto sėkmės. Atliktame ir šiame dokumente aprašytame tyrime koncentruojamasi į sistemos architektūrinę pusę. Joje atliktus architektūrinius sprendimus, technologijų panaudojimą ir technologinių galimybių paiešką bei analizę. Taip pat pasinaudojant įvairiomis metrikomis palyginami plačiai naudojami projektavimo šablonai su sistemai pritaikytu nauju šablonu.

## 1.2 GAMYBOS PLANAVIMO SISTEMOS PASKIRTIS, BEI PANAUDOJIMO SRITIS

Gamybos planavimo sistema skirta aktyviai stebėti ir optimaliai paskirstyti turimų produkcijos gaminimo mašinų apkrovas, bei suplanuoti darbus tolesniems gamybos etapams. Taip pat ši sistema sumažina arba visiškai panaikina perėjimo laiką tarp skirtingų darbo stočių operacijų. Linijinio konvejerio gamybos principu dirbančioms gamykloms dažnai tenka nukentėti nuo prasto darbo dokumentavimo, operacijų perėjėjimo nuo vienos prie kitos žemos spartos ir kitų panašių problemų, kurios dažnai trukdo tinkamai laikytis siekiamų gamybos kokybės standartų.

Norint šių problemų išvengti reikia automatizuoti kuo didesnę gamybos proceso dalį. Tai žinoma galima padaryti pasitelkiant tokio tipo sistemas, kaip pristatoma gamybos planavimo sistema. Taipogi ši sistema leidžia sumažinti sunaudotų žaliavų kiekį, o tai yra labai svarbus punktas norint padidinti savo įmonės darbo našumą, bei galimą pelną. Kiekviena mašina turi minimalų reikalingą žaliavų kiekį norint pradėti gamybą, optimaliai paskirsčius darbus galima sumažinti ši kiekį, taip jog būtų pasiekiami ženklūs teigiami rezultatai.

# 2 PROGRAMINĖS ĮRANGOS ANALIZĖ SIEKIANT LANKSTUMO

Šis skyrius skirtas pristatyti sistemai, supažindinti su projekto eigoje iškilusiomis problemomis, bei jų sprendimų paieška. Taip pat bus apžvelgiama situacija rinkoje, bei sistemos platinimo politika ir tolesni plėtojimo planai, kadangi tai glaudžiai siejasi su pasirinktais sistemos architektūriniais sprendimais.

## 2.1 PROGRAMINĖS ĮRANGOS LANKSTUMO SVARBA ŠIUOLAIKINĖSE SISTEMOSE

Gamybos planavimo sistema reikalauja daugybės įrenginių sąsajų realizacijų, kurios leistų bendrauti su kiekvienu iš jų, paimti sistemai svarbius duomenis ir vėliau juos apdoroti. Vien tas faktas, jog įrenginių gali būti kelios dešimtys reiškia programavimų darbų gausą, tai savaime suprantama ilgina projekto trukmę, kaip ir jo kaštus. Papildomai atsižvelgiant į tolesnes kompanijos vizijas šio projekto atžvilgiu – plėsti produktą kaip vientisą sistemą ir kartu ją pritaikyti kiekvienam užsakovui pagal jo specifinius reikalavimus, galime spręsti, jog darbų kiekis dar daugiau prasiplėstų. Sistemos darbų kiekio augimas numatomas dėl skirtingų įrenginių realizacijų kiekvienam užsakovui. Savaime suprantama, jog viena kompanija gali naudoti vieno gamintojo įrenginius, kita kito.

Tokiam planui įgyvendinti reikalinga ypatingai laksti sistema, kuri leistų greitai ir lengvai įgyvendinti papildomų įrenginių realizavimą.

### 2.1.1 PROGRAMINĖS ĮRANGOS LANKSTUMO ĮGYVENDINIMO BŪDAI IR UŽDAVINIŲ FORMULUOTĖS

Sistemoje esančių įrenginių konfigūravimas turi būti ypatingai patogus ir paprastas norint pasiekti užsibrėžtą sistemos lankstumą. Kaip bebūtų paprastumas ir lankstumas dažnai viena su kita prasilenkiančios savybės. Siekiant padidinti sistemos architektūrinių sprendimų teorinės analizės ir praktinių tyrimų aiškumą suformuluosime siekiamą sistemos vaizdą. Įrenginių konfigūracija turėtų vykti atskirame lange skirtame būtent šiam reikalui. Kiekvienas įrenginys būtų aprašomas atskirai nurodant jo bendrus to tipo įrenginiui skirtus parametrus. Vėliau sistemoje tose vietose, kur reikalinga įrenginių informacija ar patiems įrenginiams reikalingas signalas iš sistemos, būtų atsižvelgiama į šias konfigūracijas.

Siekiant lanksčios sistemos tenka pasitelkti įvairiausias technologijas, bei architektūrines žinias. Taigi pagrindinis uždavinys yra ištirti sistemos lankstumą, gebėjimą prisitaikyti ir surasti sistemos architektūros variantą kurį lengvai galima tobulinti ir plėsti. Tam pasiekti reikės įgyvendinti keletą užduočių:

- Sukurti naują ar patobulinti egzistuojančius projektavimo šablonus, siekiant sistemos lankstumo.
- Palyginti egzistuojančius sprendimus su nauju projektavimo architektūriniu sprendimu konceptualiame projektavimo lygmenyje.
- Palyginti egzistuojančius sprendimus su nauju projektavimo architektūriniu sprendimu realiame įgyvendinimo lygmenyje.
- Įvertinti ir įvardinti sukurto arba patobulinto architektūrinio sprendimo geriausius panaudojimo atvejus, jo teigiamas ir neigiamas puses.
- Eksperimentiniu būdu patikrinti kokią naudą duoda sukurta ar patobulinta architektūra sistemos atnaujinimo atveju.

## 2.2 PROJEKTAVIMO ŠABLONŲ APŽVALGA

Teisingai pasirinkti architektūriniai sprendimai gali labai teigiamai įtakoti tolesnį sistemos kūrimo ir gyvavimo procesą [15], todėl svarbu surasti atidžiai išanalizuoti ir surasti tinkamiausius sprendimus. Ieškant architektūrinių sprendimų galinčių tenkinti sistemai iškeltas sąlygas buvo aktyviai tyrinėjami projektavimo šablonai (angl. Design patterns). Kiekvienas programuotojas dažnai susiduria su vienu ar kitu populiaresniu projektavimo šablonu, neretai net pats to nežinodamas. Šie šablonai padeda pritaikyti sistemą prie objektinio programavimo principų, leidžia standartizuoti programinio kodo sritis ir neretai labai išplečia sistemos lankstumą ir išplečiamumą [12].

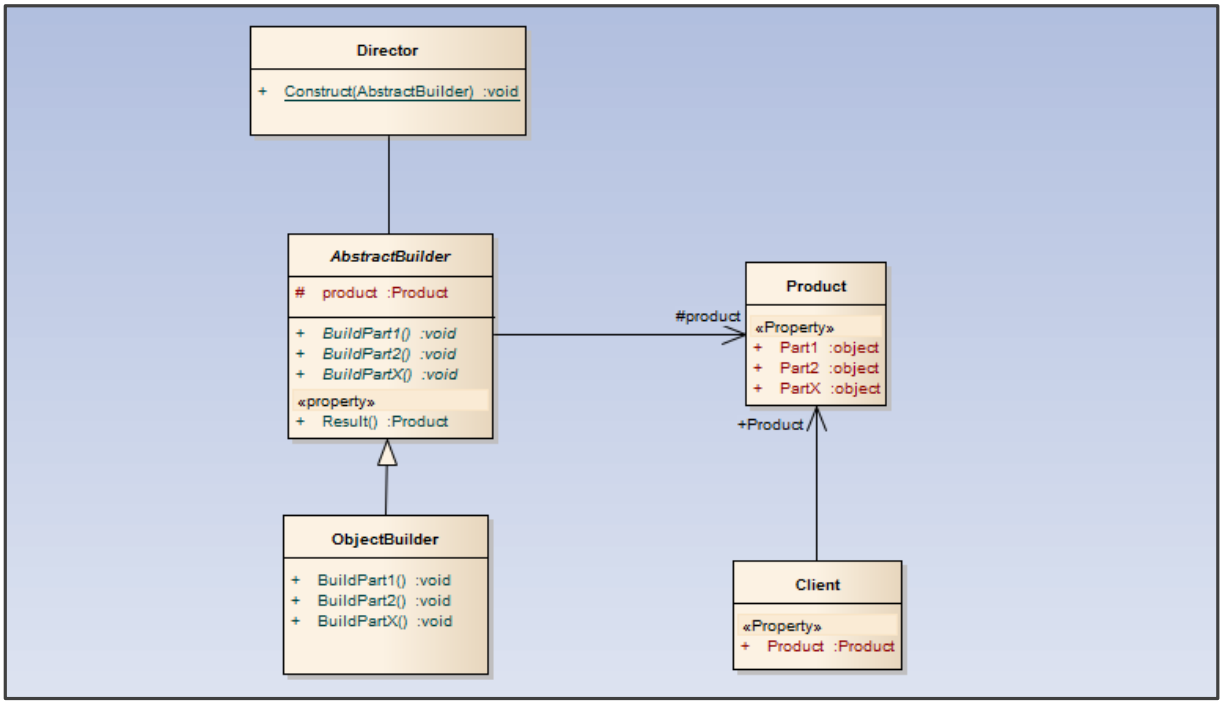
Aukštesniame lygyje sutinkama ir tokių šablonų, kurių sritis yra daug platesnė, neretai apimanti visą sistemą [1]. Projektavimo šablonai skirstomi į keletą kategorijų, skirtingi šaltiniai dažnai aprašo skirtingas kategorijas, o keletas iš jų netgi bando įvesti vieną ar kitą naują kategoriją. Vieną iš pirmųjų suskirstymų atliko keturi autoriai, dar vadinami „keturių žmonių gauja“ [3]. Savoje knygoje „Elements of Reusable Object-Oriented Software“ jie aprašė dvidešimt tris projektavimo šablonus ir suskirstė juos į tris kategorijas:

- Kuriamieji (angl. Creational)
- Struktūriniai (angl. Structural)
- Elgsenos (angl. Behavioral)

Nors nuo to laiko praėjo jau nemažai metų ir atsirado, tiek daug naujų projektavimo šablonų, tiek buvo pristatyta ne viena nauja kategorija. Daugelis iš jų yra tik smulkesnis suskirstymas, todėl tęsiant tyrinėtų architektūrinių sprendimų apžvalgą naudosimės pirmuoju pasiūlytų suskirstymu, kuriame naudojami tik trys tipai. Projektavimo šablonas pristatytus ir aprašytus po knygos išleidimo priskirsime vienai iš trijų kategorijų pagal jų raktines savybes.

### 2.2.1 STATYBININKO (ANGL. BUILDER) ŠABLONO PANAUDOJIMAS SUDĖTINGŲ OBJEKTŲ KŪRIMUI, JO PRIVALUMŲ BEI TRŪKUMŲ ANALIZĖ

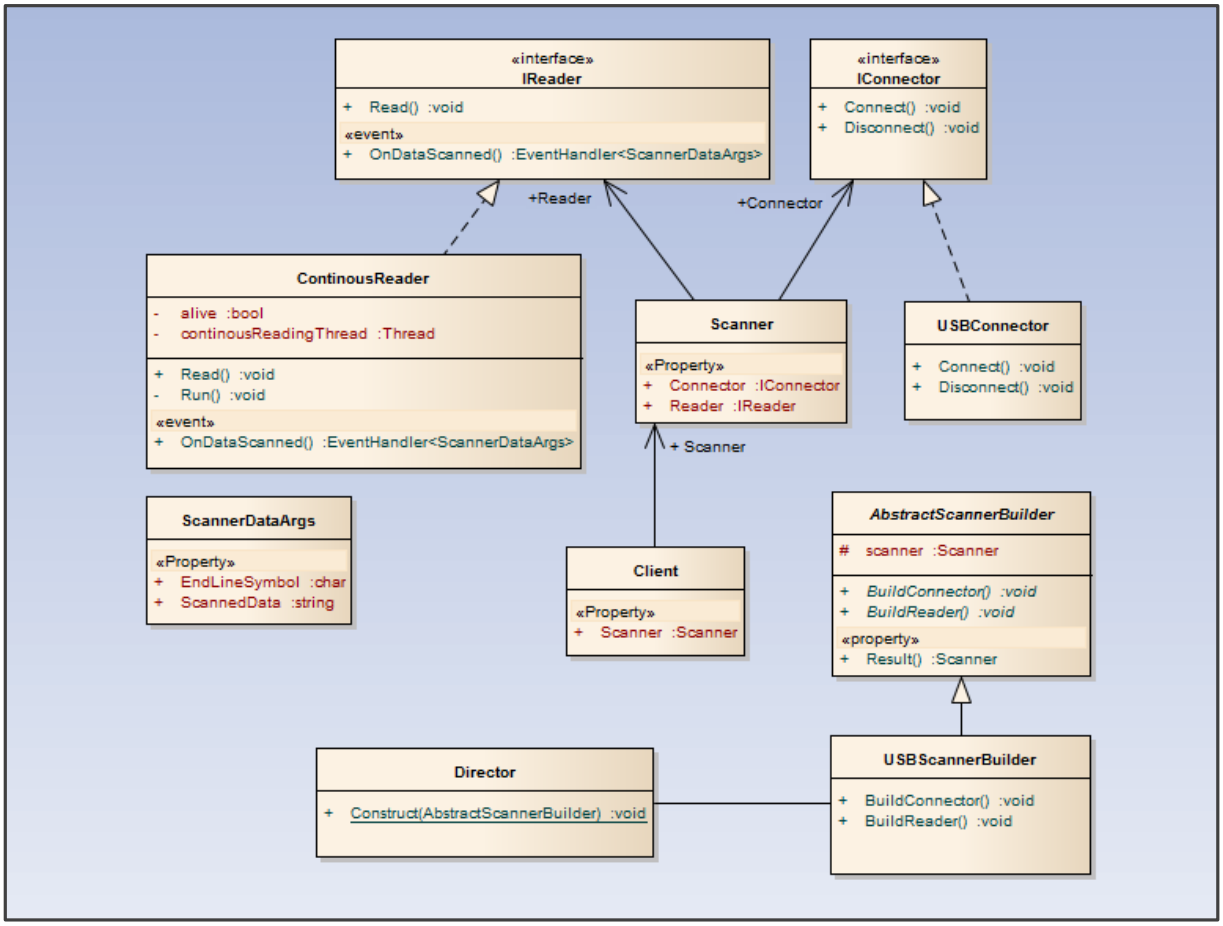
Statytojo šablonas remiasi objekto susidedančio iš kelių sudėtingų objektų išskaidymo į dalis principu. Kiekviena dalis kuriama atskirai, ir gali būti įgyvendinta visiškai skirtingai. Patį objektą visuomet sudaro tos pat dalys. Statybininko šablonas iš ties labai panašus į vėlesniuose skyriuose aprašomą gamyklos šabloną tiesa jis koncentruojasi į skirtingą atskirų dalių kūrimą, o ne skirtingų objektų kūrimą.




**Paveikslas nr. 1 Statybininko šablono abstraktaus pavyzdžio diagrama**

Šio šablonas naudojimas leidžia pagaminti vieną objekto tipą, susidedantį iš skirtingų dalių realiai sudarančių skirtingą objektą. Toks objektų kūrimo būdas turi vieną privalumą, vietoj bazinio objekto naudojimo ar sąsajos, turimas vienas tipas, kuris niekuomet nesikeičia, keičiasi tik jo komponentai. Protingai realizavus komponentų siūlomų funkcijų panaudojimą gaunamas unikalus atvejis kai to paties tipo objektas elgiasi visiškai skirtingai vienoje ar kitoje situacijoje. Be to visų šių objektų kūrimas puikiai inkapsuliuotas ir paslėptas nuo programuotojo. Tai leidžia lengvai ir be didesnių nesusipratimų sukurti reikiamus objektus.

Pritaikius šį šabloną gamybos planavimo sistemai jo struktūra pasidaro šiek tiek sudėtingesnė. Kiekvieną komponentą reikia pakeisti baziniu tipu, jog būtų galima įgyvendinti šio šabloną tikslą ir pasinaudoti vieno tipo objektu turinčiu kelis skirtingus funkcionalumus.




**Paveikslas nr. 2 Statybininko šablonas pritaikytas gamybos planavimo sistemai**

Pavyzdyje (žiūr. Paveikslas nr. 2 Statybininko šablonas pritaikytas gamybos planavimo sistemai) naudojamas brūkšninių kodų skanavimo įrenginio objektų kūrimo realizavimas. Klasė *Scanner* sudedama iš dviejų komponentų:

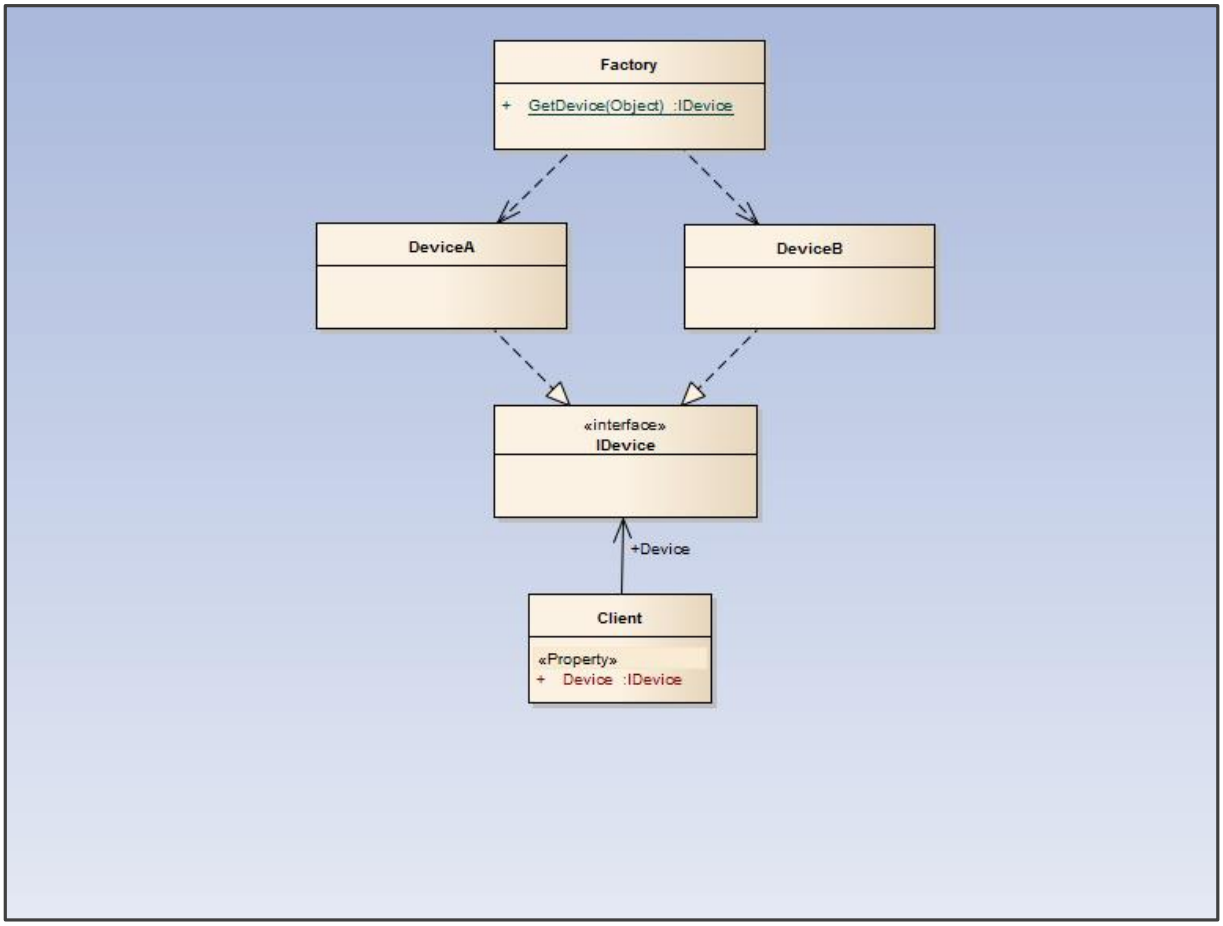
- *IConnector* sąsajos tipo komponento skirto prisijungimui prie įrenginio.
- *IReader* sąsajos tipo komponento skirto duomenų nuskaitymo mechanizmo realizavimui.

Pavyzdyje realizuotas įrenginys prie sistemos jungiamas per USB jungtį ir jame duomenų skaitymas vyksta nuolatos, nes per USB besijungiantys įrenginiai dažniausiai nesugeba pranešti apie atkeliavusią informaciją. *Director* klasė skirta sudėti reikiamoms dalims kaip ir standartinėje šio šablono variacijoje. Šablonas turi tiek privalumų, tiek trūkumų norint pasiekti sistemai užsibrėžtus tikslus. Daugiau apie juos bus papasakota skyriuje 5 Eksperimentinis šablonų palyginimas pasinaudojant diagramų metrikomis, kodo metrikomis ir Realizacijos laikų palyginimu.

### 2.2.2 GAMYKLOS (ANGL. FACTORY) ŠABLONO PRIVALUMAI IR TRŪKUMAI OBJEKTŲ KŪRIMO ATŽVILGIU

Šis projektavimo šablonas yra vienas iš senesnių projektavimo šablonų, tačiau labai dažnai naudojamas sistemose, kuriose reikia turėti daugybę klasių objektų. Gamyklos šablonas priskiriamas prie kuriamųjų šablonų kategorijos. Šiuo atveju kategorijos pavadinimas tiksliai nusako ką gi pats šablonas atlieka. Gamyklos šablonas skirtas kurti daugybę skirtingo tipo klasių objektų. Jis supaprastina kūrimo proceso ir sumažina rašomo kodo kiekį norint sukurti vieną ar kitą objektą. Žvelgiant į tai iš kitos pusės gamyklos šablonas ne tik leidžia greičiau sukurti pačius objektus, tačiau ir palengvina naujų tipų pridėjimą į gamyklos galimybes. Šis šablono privalumas yra puikiai tinkantis siekiant lengvai praplėsti turimų įrenginių realizacijų sąrašą nekeičiant jau egzistuojančio kodo.

Šablonas dažnai išskiriamas į du atskirus tipus abstraktų ir paprastą gamyklos šablonus. Kaip pirmasis kandidatas sistemos architektūriniai problemai spręsti bus pasirinktas paprastas gamyklos šablonas. gamyklos šablonas leistų centralizuoti sistemoje konfigūruojamų įrenginių kūrimą ir reikalui esant pridėti naujų įrenginių. Centralizavus kuriamus įrenginius ir suteikus jiems bendrą tipą pagal jų paskirtį sistemoje galima būtų panaudoti tik bendrinį tipą neatsižvelgiant į jo konkrečią realizaciją. Taip kiekvienas naujas įrenginys elgtųsi visiškai taip, kaip elgėsi prieš tai naudoti įrenginiai, tol kol jame būtų realizuotos visos pagrindinės ir būtinos funkcijos sistemos sklandžiam veikimui palaikyti.




**Paveikslas nr. 3 Bendra gamyklos diagrama**

Gamyklos šablonas gali būti realizuojamas pasinaudojant tiek bazinės klasės principu, kur visos reikalingos funkcijos, bei saugomi duomenys aprašomi bazinėje klasėje arba tam tikrose kalbose sąsajos (angl. interface) principu. Kadangi sistemoje naudojama C# programavimo kalba visos kalbos tendencijos ir apribojimai bus laikomi kaip numatyti dalykai kalbant apie šablonų realizaciją ir kitus techninius, su kalba susijusius, dalykus.

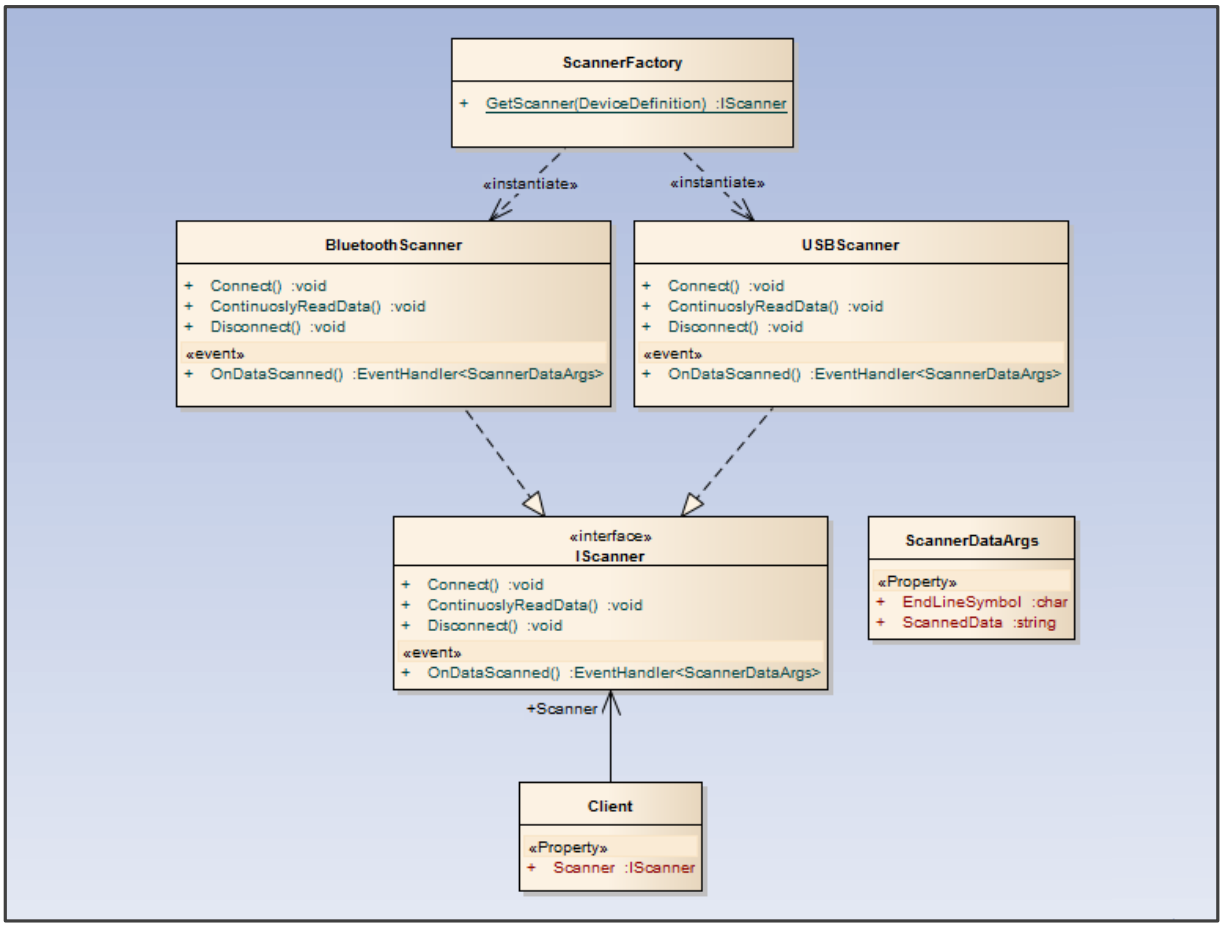
Pateiktame pavyzdyje pasirinktas sąsajos principas. Šį pasirinkimą įtakojo tiek kalbos limitacija leidžianti klasei turėti tik vieną tėvinę arba dar kitaip vadinamą super klasę, tiek J. Hyslop‘o ir H. Sutter‘io straipsnis[4] apie virtualiuosius metodus, bei abstrakčių klasių naudojimą. Kiekviena abstrakti klasė leidžia turėti tokių privalumų kaip jau įgyvendintas bazinis funkcionalumas, kuris nesikeistų arba turėtų galimybę nesikeisti visose kitose klasėse paveldėsiančiose ją. Šis privalumas dažnai leidžia įgyvendinti struktūras, kurios kitu atveju būtų neįmanomos arba bent jau nepriverstų kiekvieno programuotojų jas įgyvendinti būtent taip, kas sutrikdytų sistemos vientisumą ir sukeltų papildomų nesklandumų realizuojant pagrindinius gamyklos šablono privalumus. Nepaisant to, jog bazinė klasė turi savo privalumų, programuojant C# programavimo kalba jos panaudojimą reikia pagrįsti, o pačiu paprasčiausiu gamyklos atveju realaus pagrindimo nėra. Šiuo atveju puikiai tinka ir sąsaja,

kurių klasė gali realizuoti neribotą skaičių, taip daug aiškiau paskirstant kodą į skirtingus gabalėlius atliekančius tik tam tikras, tinkamai įvardintas funkcijas. Sakykime mūsų gamyklos sukuriamas įrenginys *IDevice* iš tiesų yra klasė DeviceA, kuri dar yra realizavus ir klonavimo sąsają *IClonable.* Tokia klasės realizacija aiškiai išskiria du skirtingus principus:

- Realizuota klasė yra įrenginys (*IDevice*)
- Realizuota klasė yra klonuojama (*IClonable*)

Tokiu būdu klasėms suteikiamos tik tokios savybės, kurių joms reikia. Visa tai palengvina kodo skaitomumą ir tinkamiau įgyvendina objektinio programavimo principus.

Performuojant bazinį šabloną į sistemai pritaikytą realizaciją gali pasitaikyti ženklių pasikeitimų. Šiuo konkrečių atveju pritaikant šabloną sistemai didesnių pasikeitimų nepasitaikė. Kaip pavyzdys naudojama bendravimo su brūkšninių kodų skaneriais objektų kūrimo gamyklos realizacija. Realizuojama sąsaja *IScanner* turi reikiamas funkcijas bendravimui su skaneriais ir jų gautam rezultatui atiduoti. Sistemoje pateiktos dvi iš daugybės galimų realizacijų. Toks objektų kūrimo būdas leidžia lengvai ir centralizuotai kurti bendrą bazinį tipą turinčius objektus, nesukant galvos dėl papildomų parametrų.




**Paveikslas nr. 4 Diagramos pavyzdys pritaikytas sistemai**

Sistemoje reikia realizuoti daugiau nei vienos rūšies įrenginius ir nors pasinaudojant gamyklos šablonu tai puikiai galima atlikti, iškyla papildomų problemų realizuojant įrenginius, kuriems reikia daugybės gijų palaikymų. Kitaip tariant, šie įrenginiai bendrauja nepriklausomais procesais nuolatos atnaujindami savo informaciją. Pasinaudojant gamyklos šablonu tokių įrenginių konfigūracijos pakeitimas reikalautų visų įrenginių objektų perkūrimo ir nors tai ir nėra sudėtinga, didėjant įrenginių kiekiui būtų susiduriama su nereikalingais sunkumais. Viso šių problemų galima išvengti pasinaudojant siūlomu „Reflective self-resolver“ šablonu, kuriam lietuviškas pavadinimas dar nesugalvotas.

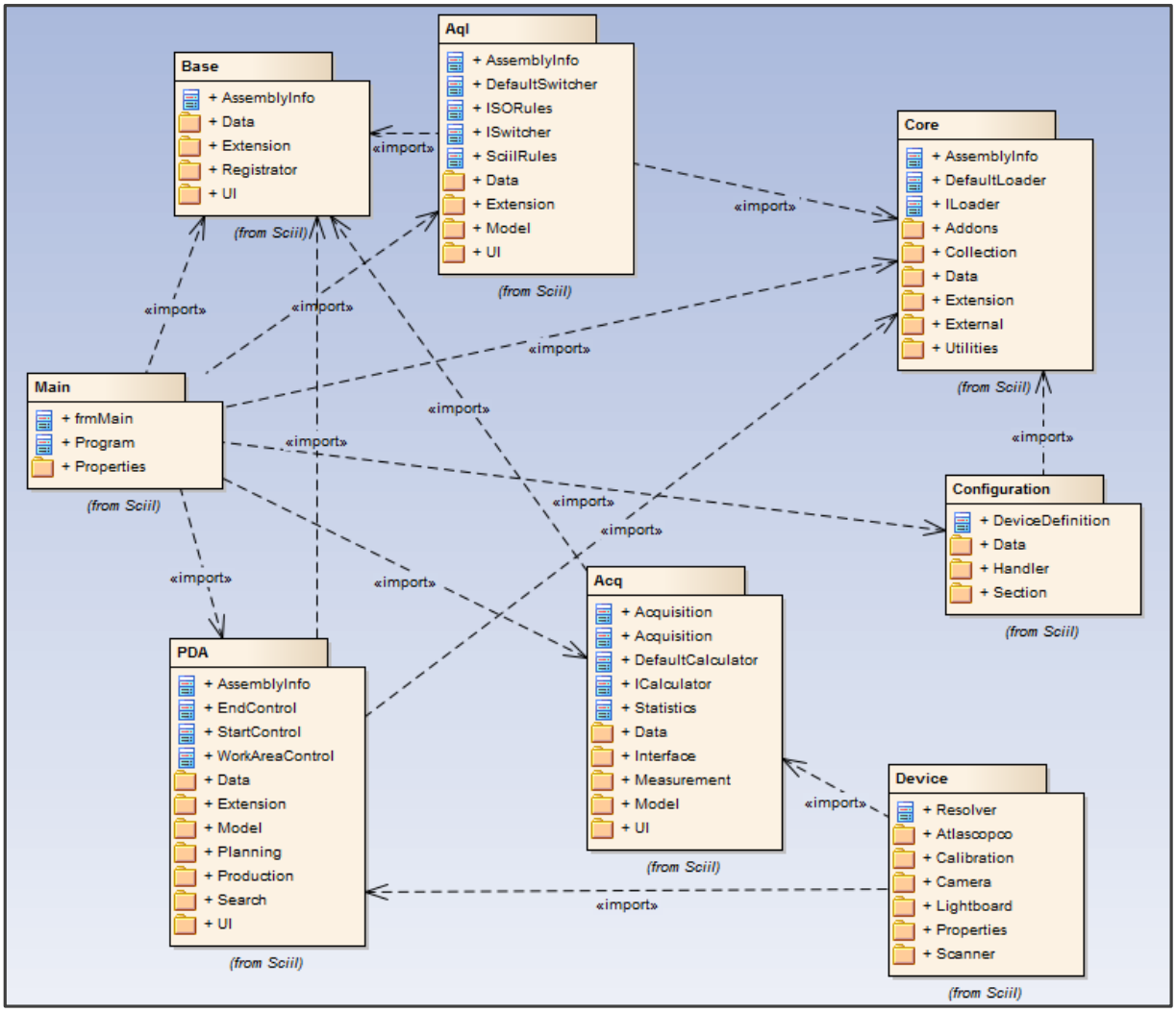
# 3 GAMYBOS PLANAVIMO SISTEMOS STRUKTŪRINIS VAIZDAS

Šioje dalyje bus supažindina su sistemos struktūra ir bandoma atsakyti į analizės metu iškeltus klausimus, bei pademonstruoti pasirinką realizaciją. Kadangi sistema ganėtinai plati, bus stengiamasi akcentuoti jos dalį susijusią su įrenginių naudojimu, jos konfigūravimu ir papildomų įrenginių įdiegimu į sistemą.

## 3.1 SUPAŽINDINIMAS SU SISTEMOS FUNKCIONALUMU IR ARCHITEKTŪRINIAIS SPRENDIMAIS

Apžvalgoje bus supažindinama su sistemos architektūra stambiu mastu. Nebus detalizuojamas klasių, bei funkcijų vidus. Taip pat apžvalgoje bus pavaizduojamas analizės metu aprašyto architektūrinio sprendimo įgyvendinimas.

Sistemą sudaro paketų rinkinys, kurio pagrindinė dedamoji dalis yra PDA paketas. Šis paketas naudos, kitus šios sistemos paketus, apjungdamas juose realizuotą logiką į visumą. Kiekvienas paketas turi unikalų funkcijų rinkinį leidžiantį atlikti specifinius jam būdingus veiksmus ir vienaip ar kitaip yra nepriklausomas. Taip paketai tampa lankstesni ir juos gali panaudoti kituose projektuose be visiškai jų nemodifikuojant arba minimaliai išplečiant jau esantį funkcionalumą. Išimtis yra PDA paketas, jis skirtas būtent gamyklos planavimo sistemai reikalingoms funkcijoms realizuoti ir yra priklausomas nuo likusių paketų.




**Paveikslas nr. 5 Sistemos paketų vaizdas**

Nors diagramoje rodomi ryšiai to ir nerodo, nes tiesioginiai ryšiai yra tik su nubrėžtais paketais, PDA paketas yra priklausomas nuo visų paketų išskyrus Main. Sistemoje realizuoti paketai yra atsakingi už šias funkcijas:

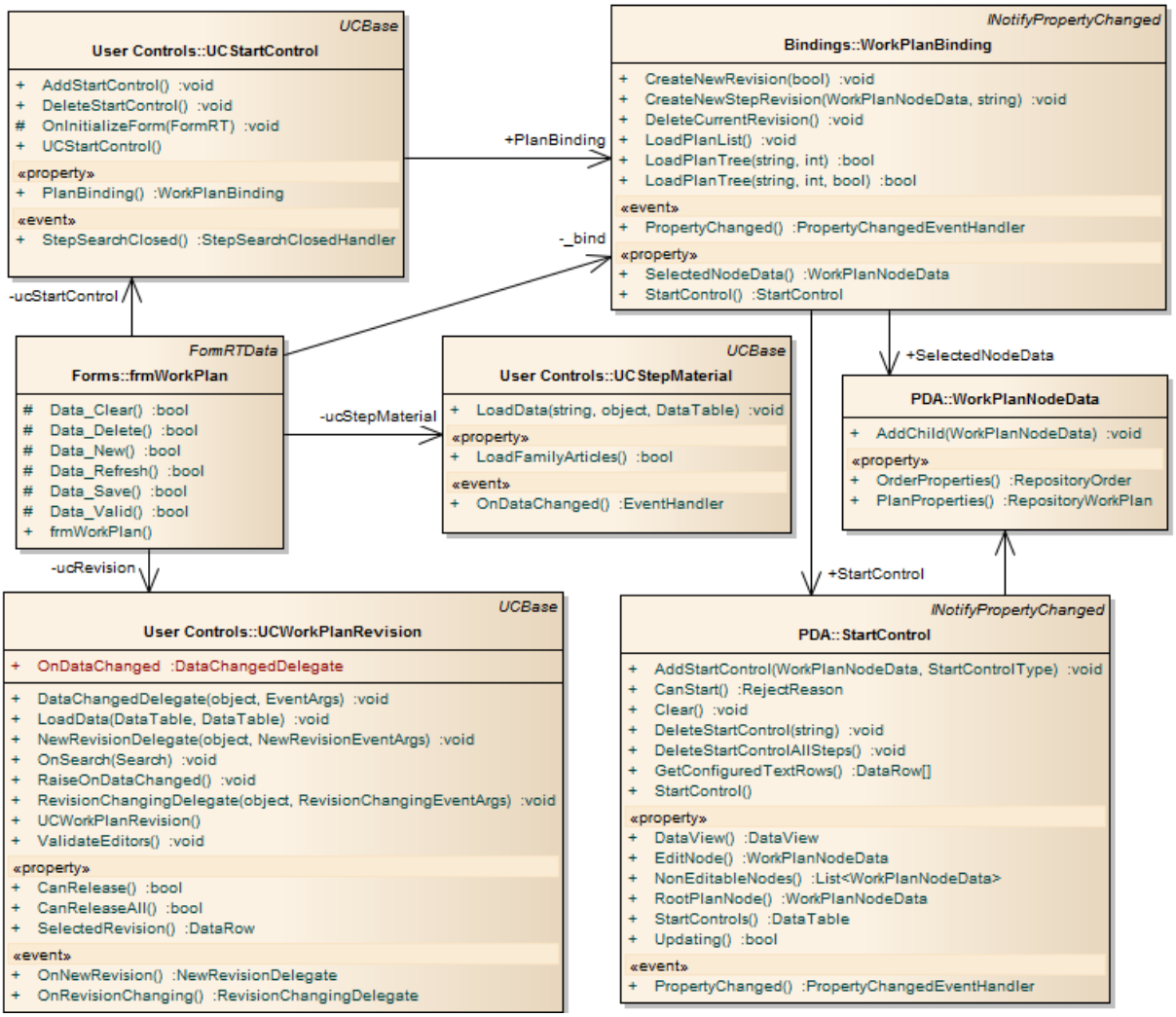
- Main – Pagrindinis langas ir sistemos paleidžiamasis failas.
- Acq – atsakingas už duomenų surinkimą, kokybės apžiūros duomenis, jų analizę, bei rezultatus.
- Aql – matavimų kiekių apskaičiavimui skirtas paketas, kuris skaičiuoja kokią dalį produkcijos matuoti priklausimai nuo partijos dydžio.
- Base – bazinių duomenų paketas, kadangi yra importuojamas į pačią sistemą gali būti papildytais reikalingais duomenimis šiai konkrečiai sistemai.
- Core – pagrindines sistemos funkcijas, bei neutralias klases laikantis paketas.
- Configuration – konfigūracinio failo nuskaitymo ir nuskaitytų duomenų saugojimo paketas, kuriame tai pat realizuotos ir visos konfigūracinės formos.
- Device – bendravimui su įrenginiais ir jų valdymui skirtas paketas.

- PDA – pagrindinis paketas, kuriame konkretizuojamas sistemos darbas ir įgyvendinamos visos gamybos planavimo sistemai reikalingos funkcijos.

Nors PDA paketas ir yra priklausomas nuo kitų paketų, o jie gali veikti kaip nepriklausomos bibliotekos kitiems projektams. Paketai buvo projektuojami būti Gamybos planavimo sistemos dalimi ir yra orientuoti į ją. Tiesa kaip vienas iš objektinio programavimo principų buvo pasiektas gana aukštas klasių ir modulių atskirtumas (angl. de-coupling). Tai ir leido šiems paketams tapti nepriklausomais moduliais.

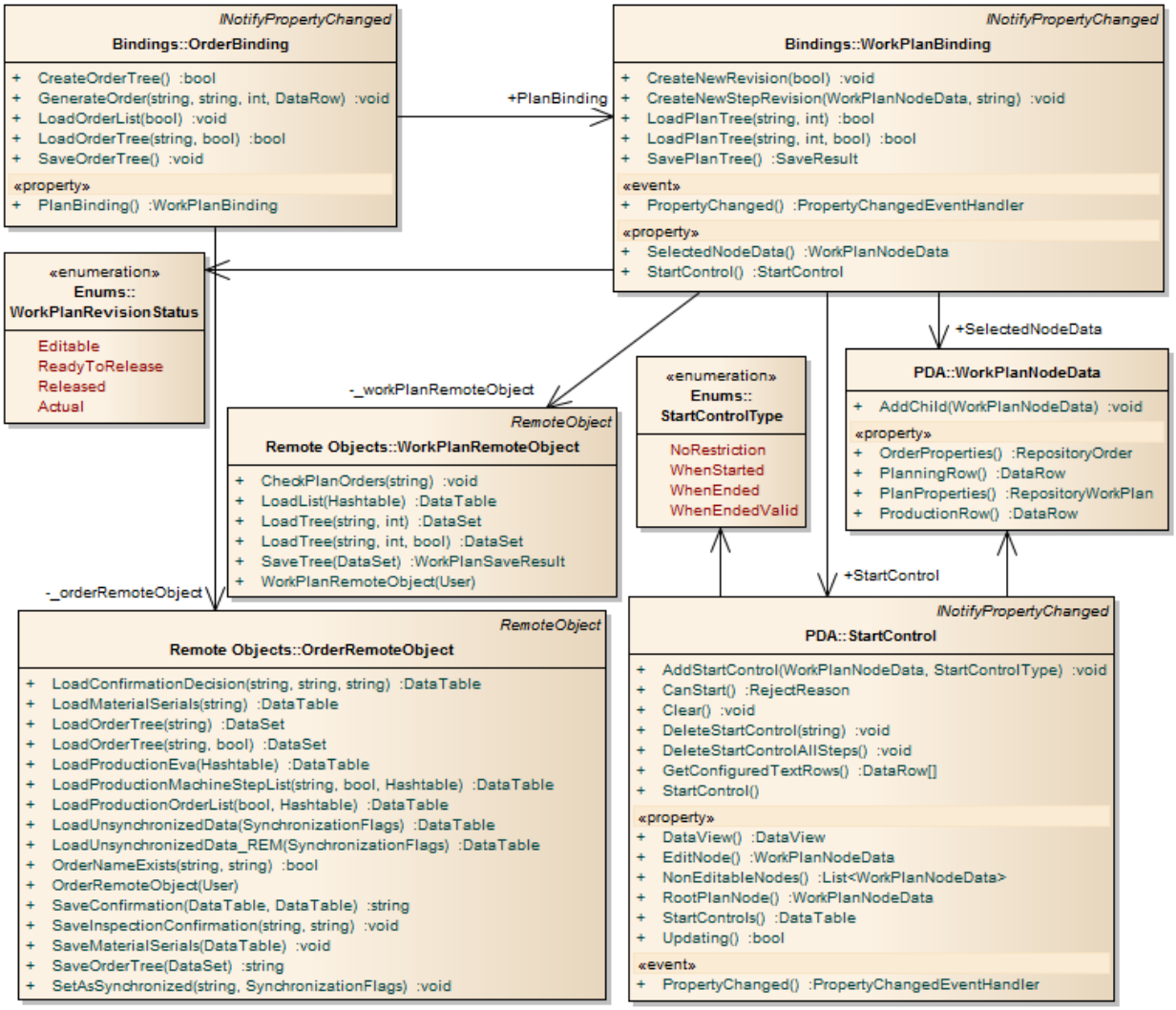
### 1.1.1 GAMYBOS PLANAVIMO SISTEMOS KLASIŲ DIAGRAMOS

Šiame skyrelyje bus pavaizduotos klasių diagramos orientuotos į didžiausias formas, procesus ir logiką įgyvendinančias klases. Kadangi bendra klasių diagrama per didelė ir netilptų į kelis lapus bus braižomos kelios klasių diagramos su pasikartojančiomis dalimis iš kiekvieno pagrindinio elemento perspektyvos. Taip pat bus supažindinama su sistemos paskirtimi ir daugelio klasių atsakomybe siekiant įvykdyti sistemos reikalavimus. Kiekvienos diagramos ryšiai leidžia atpažinti siekiamus tikslus ir kelią kaip juos pasiekti.




**Paveikslas nr. 6 frmWorkPlan formos klasių diagrama**

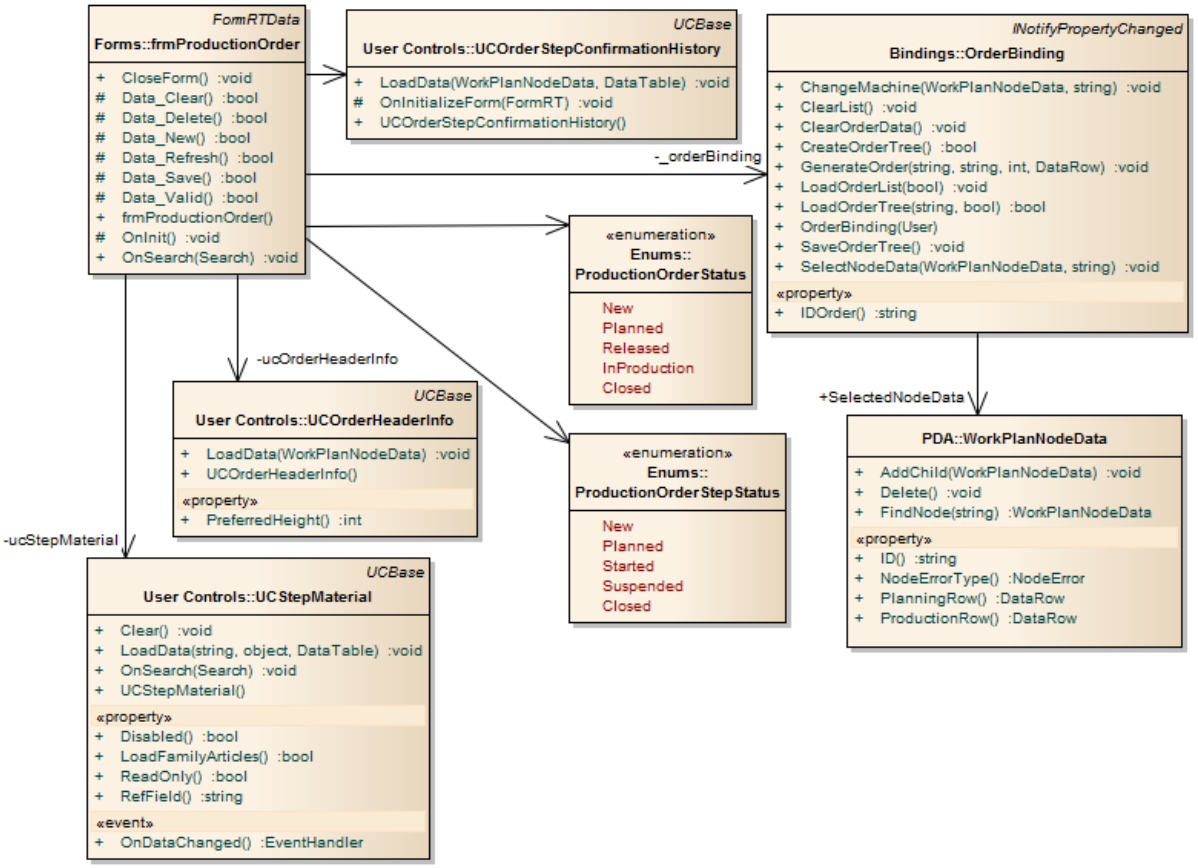
Diagramoje (žiūr. Paveikslas nr. 6 frmWorkPlan formos klasių diagrama) matomas sistemos vaizdas iš *frmWorkPlan* klasės perspektyvos. Ryšiai tarp klasių gana paprasti, naudojamas tik asociacijos ryšys, nes klasės siejasi tik vieno kintamoje pagalba. Diagramoje pavaizduota forma, joje esantys pagrindiniai komponentai ir pagalbinės klasės siejančios duomenų apdorojimą ir grafinę sąsają (bindings). Jie tarsi suriša šias dvi skirtingas sistemos dalis, taip padidina sistemos išplečiamumą ir kokybę.




**Paveikslas nr. 7 Order Binding klasių diagrama**

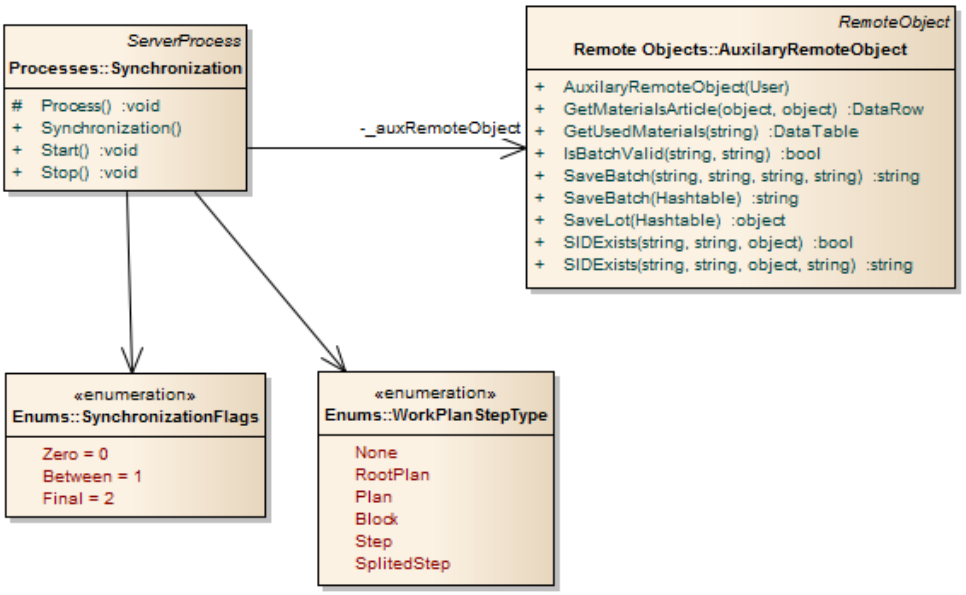
*OrderBinding* klasių diagramoje pavaizduoti ryšiai tarp pagrindinių binding objektų ir remote object objektų. Vaizdas rodomas iš *OrderBinding* klasės perspektyvos, kuri turi ryšį su planavimo binding klase, *WorkPlanBinding*. Iš tiesų *OrderBinding* klasė savyje turi *WorkPlanBinding* tipo objektą, dėl to, jog vykstant konkretaus užsakymo plano generavimui nereikėtų iš duomenų bazės atskirai gauti tų pat duomenų, kurių struktūrą galima pasiimti iš *WorkPlanBinding* klasės. *WorkPlanBinding* klasė savyje turi *WorkPlanNodeData* tipo klasių medžio struktūra, kurioje saugomos reikiamos atlikti operacijos. Ši struktūra leidžia lengvai nupiešti medžio tipo operacijų rinkinį binding klases turinčiuose komponentuose.

*StartControl* klasė nusako ryši tarp operacijų, tai reiškia ar yra apribojimų operacijai pradėti (turi būti pasibaigus arba prasidėjus praeita operacija). Skaitiklis *StartControlType* nurodo koks ryšis yra nustatytas tarp operacijų.

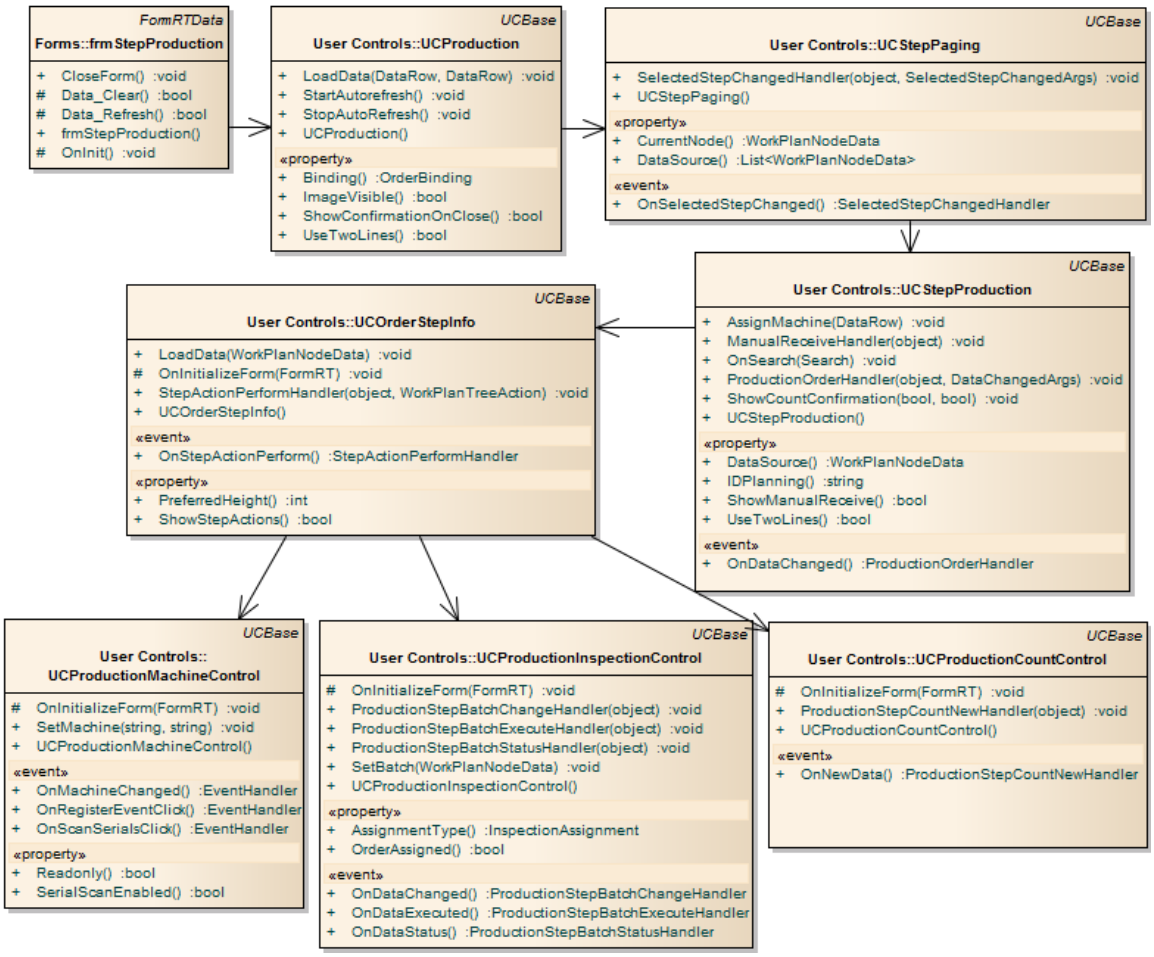



**Paveikslas nr. 8 frmProductionOrder klasių diagrama**

Paveiksle (žiūr. Paveikslas nr. 8 frmProductionOrder klasių diagrama) pavaizduota konkrečių užsakymo planavimo formos sąveika su joje esančiais komponentais ir jos ryšys su *OrderBinding* klase. *UCOrderHeaderInfo* atsakingas už visą informaciją apie patį užsakymą, neįskaitant informacijos apie jame sukonfigūruotas operacijas. UCOrderStepConfirmationHistory atsakingas už jau vykdomų užsakymo planų operacijų rezultatų įvedimo peržiūrą. *UCStepMaterial* leidžia sukonfigūruoti reikalingas žaliavas vienai ar kitai operacijai ir būna aktyvus tik tuomet kai yra parinkta operacija. Skaitikliai *ProductionOrderStepStatus* ir *ProductionOrderStatus* yra skirti nustatyti tiek operacijos tiek pačio plano statusui, nuo kurio vėliau priklauso galimybės redaguoti tas medžio šakas.




**Paveikslas nr. 9 Data.Synchronization klasių diagrama**




**Paveikslas nr. 10 frmStepProduction klasių diagrama**

Paveiksle (žiūr. Paveikslas nr. 9 Data.Synchronization klasių diagrama) pavaizduotas sinchronizacijos proceso veikimo principas, kuris skirtas susinchronizuoti duomenims apie

gamybos partiją, su išorine sistema. Proceso architektūra ganėtinai paprasta, beveik visa logika įgyvendinta pačioje sinchronizacijos klasėje ir papildomos klasės, kurias šis procesas naudoja skirtos tik bendravimui su nuotoliniu serveriu, bei duomenų baze. Sisteminiai skaitikliai padeda atskirtu kokiu rėžimu reikia vykdyti sinchronizaciją.

Paveiksle (žiūr. Paveikslas nr. 10 frmStepProduction klasių diagrama) pavaizduota aktualios produkcijos stebėjimo formos klasių diagrama ir sąryšiai tarp pačios klasės ir joje panaudotų komponentų.

# 4 ATLIKTŲ TYRIMŲ APRAŠYMAS IR SIŪLOMO ARCHITEKTŪRINIO PAKEITIMO PRISTATYMAS

Sistemos dažnai keičiamos ir tobulinamos. Kiekvienas pakeitimas reikalauja laiko ir kitų resursų ir norint sumažinti šiuos kaštus ieškoma pradinių sprendimų, kurie galėtų tą atlikti. Yra daug patogiau sukurti sistemą gebančią prisitaikyti prie aplinkos pasikeitimų vietoj to, jog atėjus laikui ją keisti ir bandyti pritaikyti. Išmatuoti sistemos lankstumą yra ganėtinai sudėtinga, nes lankstumas yra abstrakti sąvoka. Tačiau sistemos lankstumą gali išskaidyti į kitas sąvokas, kurios tikrai yra lankstumo dalis, tačiau yra daug labiau apčiuopiamos ir netgi išmatuojamos. Gamybos planavimo sistemoje lankstumas bus apibrėžiamas šiomis sąvokomis:

- Vienetų testų rašymo sudėtingumu. (gali būti matuojamas kodo metrikomis)
- Reikiamu kodo kiekiu sistemai praplėsti. (kuo mažiau tuo geriau)
- Paveldėjimo laipsnio metrika. (kuo mažesnis paveldimumo laipsnis, tuo suprantamesnis ir skaitomesnis tampa programinis kodas)
- Galimų rizikų kiekis. (paprastas skaičius galimų rizikų, šiek tiek sunkiau išmatuojamas ir kiek abstraktus, tačiau lengviau įvardijamas negu lankstumo sąvoka)

Šių savybių matavimai bus aprašomi lentelėse lyginančiose analizės dalyje aprašytus metodus su 4.1 Reflective self-resolver šablono aprašymas skyriuje aprašomu projektavimo šablonu, bei jo realizacija.

## 4.1 REFLECTIVE SELF-RESOLVER ŠABLONO APRAŠYMAS

Kaip jau minėta ankstesniuose skyriuose sistemos realizacijos metu siekiama surasti optimalų bendravimo su įrenginiais objektų kūrimo mechanizmą. Apžvelgus keliamas reikalavimus atsiradusius sistemos analizės metu. Matome, jog sklandžiai sukurti bendravimui su įrenginiais skirtus objektus vien tik prieš tai apžvelgtų projektavimo šablonų neužtenka. Atsižvelgiant į šiuos teiginius:

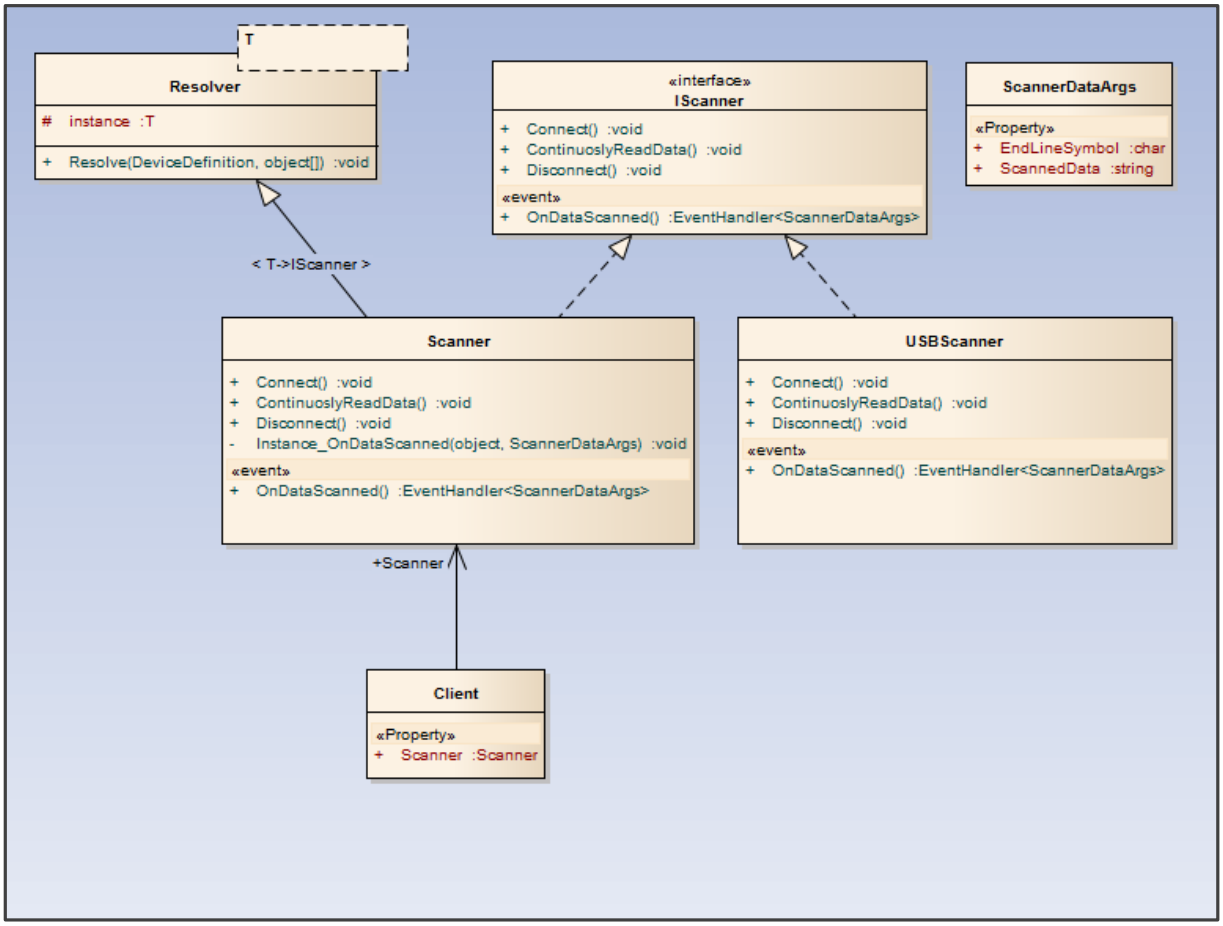
- Reikalinga galimybė centralizuotai kurti bendravimo su įrenginiais objektus pasinaudojant viena funkcija.
- Įrenginiai turi būti naudojama tik kaip baziniai tipai arba sąsajos, jog būtų galima įgyvendinti skirtingas realizacijas kuo mažiau arba išvis nekeičiant programinio kodo.
- Sistemos sustabdymas yra sąlyginai neigiamas dalykas, nes stabdoma gamyba ir jeigu yra galimybė būtų privalu keisti įrenginių konfigūraciją neperkraunant visos sistemos.

- Dinaminis sistemos plėtimas būtų didelis privalumas, taip pat kaip ir sistemos plėtimas iš išorės.

Šie kiek vartotojiško, kiek techninio pobūdžio teiginiai nusako siekiamus įgyvendinti sistemos funkcionalumus, tačiau jiems trūksta konkretumo realizacijos atžvilgiu. Norint sukurti sistemą tenkinančią visus šiuos punktus reikėtų pasinaudoti C# programavimo kalbos teikiamais privalumais, bei visomis architektūrinėmis žiniomis. Visus šiuos teiginius galima užrašyti kitu teiginių sąrašu, kuriame būtų daugiau techniniai teiginiai susiję su realizacija:

- Reikalingas objektas galintis keisti būseną.
- Objektas turi būti paremtas baziniu tipu. Išlieka reikalavimas sistemoje turėti vien bazinius tipus.
- System.Reflection paketo teikiamų funkcionalumų panaudojimas.
- Reikia įgyvendinti mutex principus objekto viduje, taip apsaugant nuo galimų problemų daugelio gijų atveju.

Šis sąrašas apibūdina visas „*Reflective self-resolver*“ siūlomas galimybes. Šio šablono realizacija pasinaudojant System.Reflection paketo teikiamu funkcionalumu leidžia ne tik kurti objektus nieko apie juos nežinant, bet ir pridėti papildomų klasių, kurios realizuoja bazinį tipą išoriškai, nekeičiant programinio kodo.




**Paveikslas nr. 11 Self-resolver šablono diagrama**

Diagramoje (žiūr. Paveikslas nr. 11 Self-resolver šablono diagrama) matomas self-resolver šablonas susideda iš kelių pagrindinių dalių:

- *Resolver* klasės su jau įgyvendinta Resolve funkcijos logika.
- *IScanner* sąsajos leidžiančios apibendrinti realizuojamus tipus į vieną.
- *Scanner* klasės veikiančios kaip tarpininkės, galinčios keisti savo tipą.

Kalbant apie gamyklos šabloną buvo paminėti skirtumai tarp sąsajų naudojimo ir paprastų klasių arba abstrakčių klasių paveldėjimo. Šiuo atveju pačiam šablonui apibrėžti galima naudoti tiek sąsają tiek paveldimą klasę. Šablonas pats neaprašo įgyvendinimo logikos jis tarsi gidas pačiai struktūrai, todėl tokiam atvejui daug tinkamiau būtų naudoti sąsają. Kita vertus kalbant apie šį šabloną kaip architektūrinį sprendimą ir gilinantis į jo realizaciją pasirinktas paveldimos klasės kelias, nes jis leidžia iš anksto realizuoti *Resolve* funkciją. Pati funkcija realizuota pasinaudojant System.Reflection paketo teikiamomis galimybėmis, todėl galimas dinaminis objektų inicializavimas sistemos veikimo metu. Taip išpildomi reikalavimai keisti objektų būsenas neperkraunant sistemos, suteikiamos galimybės išoriniam sistemos plėtimui ir viskas yra palaikoma bazinio tipo pagrindu.

```
public virtual void Resolve(DeviceDefinition definition, params object[] args)
{
     Type type = Type.GetType(definition.Type, false, true);
     instance = (T)Activator.CreateInstance(type, args);
}
```

**Paveikslas nr. 12 Bazinis Resolve funkcijos realizavimas**

Ši realizacija buvo pavadinta „*Reflective self-resolver*“ dėl jos *Resolve* funkcijos įgyvendinimo pasinaudojant System.Reflection paketo funkcionalumais. Funkcijos vidus labai paprastas, tačiau daug keičiantis sistemos veikime:

Šis iš pirmos pažiūros paprastas mechanizmas galį būti pritaikytas, bet kokiai sąsajai ir ją realizuojančiam objektui. Taip pat ši funkcija negrąžina jokio tipo, o tik inicializuoja klasės viduje esantį objektą, kuris paveldėjusioje klasėje naudojamas kaip tarpininkas. Realizuojanti klasė *Scanner* kartu realizuoja ir *IScanner* sąsają ir paveldi Resolver klasę, kurios tipas yra *IScanner*.

```
public class Scanner : Resolver<IScanner>, IScanner
{
        //....
}
```

**Paveikslas nr. 13 Self resolver klasės objekto aprašas pasinaudojant IScanner sąsaja**

Norint užtikrinti išorinį sistemos plėtimą pagrindai jau padėti, tačiau reikia lanksčios konfigūracijos, kuri leistų įdiegti naujas realizacijas neperkraunant sistemos. Jau nuspręsta, jog sistemai kaip parametras bus paduodama *DeviceDefinition* klasė, kurios struktūra pavaizduota paveiksle (žiūr. Paveikslas nr. 14 Device definition klasės aprašas). Ši struktūra susideda iš paprasto pavadinimo ir pilno reflection kelio į klasę.

```
public sealed class DeviceDefinition
{
    /// <summary>
    /// Gets or sets full type definition: namespace, class and assembly.
    /// </summary>
    [XmlAttribute("type")]
    public string Type { get; set; }
    /// <summary>
    /// Gets or sets display name in device basedata form.
    /// </summary>
    [XmlAttribute("name")]
    public string Name { get; set; }
}
```

**Paveikslas nr. 14 Device definition klasės aprašas**

Žinoma klasė turi būti sistemos dalis, todėl tam tikslui bus skenuojamas sistemos aplankas ir užkraunamos visos išorinės klasės įgyvendinančios tam tikras sąsajas ir aprašytos konfigūraciniame faile. Konfigūracinis failas kartu naudojamas ne tik kaip papildomas kriterijus atskirti kurias bibliotekas įkelti į sistemos atminties erdvę, tačiau ir sistemoje esančios konfigūracinės formos įrenginių parinkimo laukelio dinaminiam užpildymui. Visos galimos įrenginių realizacijos atsiras kaip sąrašas sistemos konfigūracinėje formoje, kur vėliau bus galima sukonfigūruoti naudoti vieną ar kitą realizaciją priklausomai nuo vietos ir nuo reikalaujamos logikos.

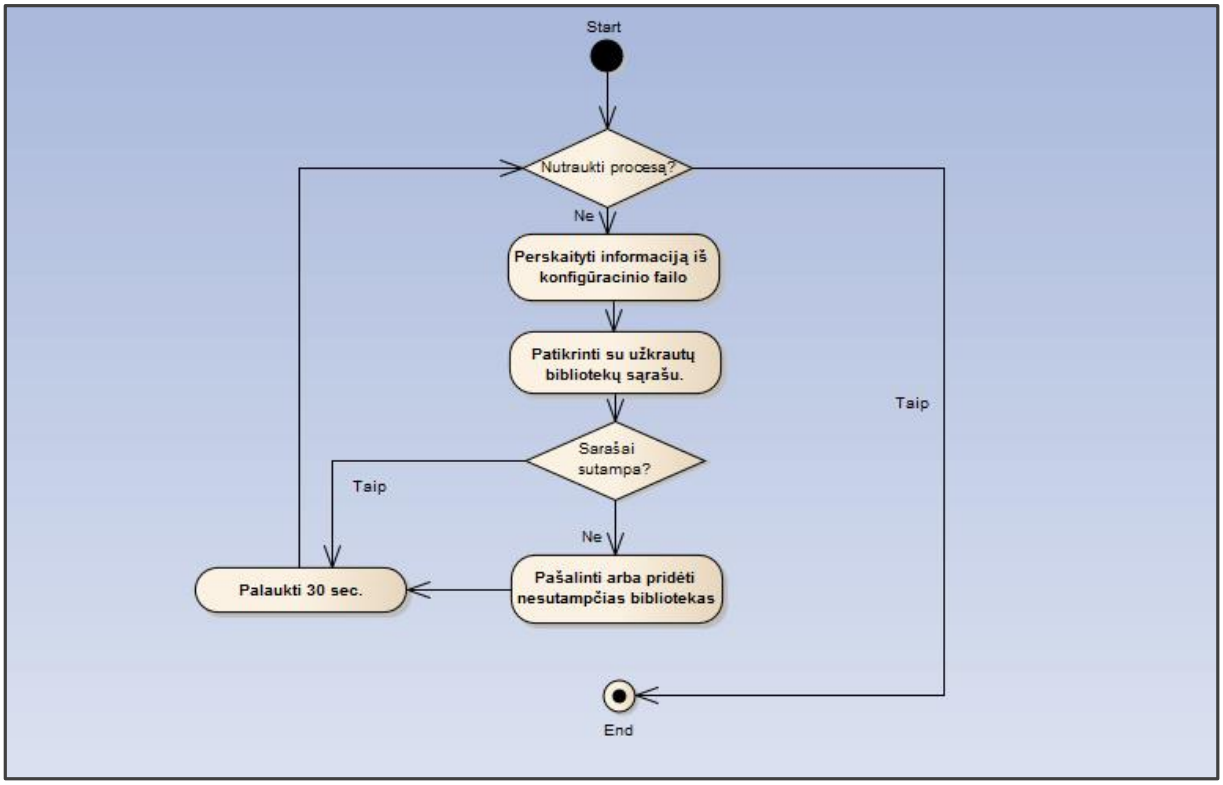
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="Devices" type="Sciil.Device, Sciil.Device.v2.9"/>
  </configSections>

  <Devices>
    <Scanners>
      <Device name="Honeywell MS9520" type="Honeywell.MS9520, Honeywell.MS9520.v1.0"/>
    </Scanners>
    <Lightboards>
      <Device name="TONSEC 18SMDS" type="Tonesec.Lightboards, Tonesec.v1.1"/>
    </Lightboards>
  </Devices>
</configuration>
```

**Paveikslas nr. 15 Konfigūracinio failo pavyzdys**

### 4.1.1 ĮRENGINIŲ IŠORINIO PRIJUNGIMO ARCHITEKTŪRA IR PRIVALUMAI

Sistemoje įgyvendinta unikali struktūra leidžianti išoriškai prijungti įrenginius prie sistemos nereikalaujant, jokio jos atnaujinimo ir net perkrovimo. Šis mechanizmas susideda iš kelių sudėtingų procesų, bei struktūrinių sprendimų, tačiau savo lankstumu suteikia didelę galimybių įvairovę naudojantis sistema.

Nuolatos besisukantis procesas tikrina sukonfigūruotą įrenginių sąrašą (šis yra aprašomas konfigūraciniame faile) su sistemoje užkrautu įrenginių sąrašu. Sistemoje veikiantis įrenginių sąrašas yra saugojamas bendroje sistemai operacinės sistemos išskirtoje operatyviojoje atmintyje. Norint užtikrinti išimti vieną ar kitą biblioteką iš atminties, reikia užtikrinti, jog jos nėra naudojamos, tai įgyvendinti padeda Sciil.Core realizuota *ILoader* sąsaja *DefaultLoader*. Kadangi nėra universalaus būdo kaip patikrinti ar biblioteka yra šiuo metu naudojama ar ne. Kviečiant *Resolver* metodą *Resolve* kiekvienam įrenginio objektui išskiriamas unikalus raktas saugomas specialioje kolekcijoje, su kuria bendrauja *DefaultLoader* klasė norėdama pašalinti vieną ar kitą biblioteką.




**Paveikslas nr. 16 Išorinių įrenginių valdymo veiklos diagrama**

Jeigu objektas tuo metu naudojamas bandoma tol, kol jis yra atlaisvinamas. Neatlaisvinus objekto ilgiau negu 5 minutes yra išsaugojamas pranešimas, kuris gali būti išsiunčiamas elektroniniu paštu, sistemos pranešimų žurnale kaip įspėjimas apie potencialiai klaidingą konfigūraciją.

# 5 EKSPERIMENTINIS ŠABLONŲ PALYGINIMAS PASINAUDOJANT DIAGRAMŲ METRIKOMIS, KODO METRIKOMIS IR REALIZACIJOS LAIKŲ PALYGINIMU

Skyriuje 4.1 Reflective self-resolver šablono aprašymas pristatytas šablonas turi savų privalumų, tačiau turi ir savų trūkumų. Norint palyginti šį šabloną, bus sudaromos privalumų ir trūkumų lentelės leidžiančios lengviau nustatyti vienos ar kitos architektūros teorinį pranašumą. Tai turėtų padėti lengviau įsivaizduoti apie kokiu skirtumus tarp šių architektūrų yra kalba ir kokioje situacijoje reikėtų naudoti vieną ar kitą variantą. Savybių lentelėje (žiūr. Lentelė nr. 1 Šablonų savybių palyginimas) kalbama apie savybes, kurias turi pats šablonas arba tam tikra jo realizacija. Beveik kiekvieną architektūrinį sprendimą galima pasukti viena ar kita linkme, jog jis turėtų vieną ar kitą savybę, tačiau čia lyginamos tik tos savybes, kurios gali būti realizuotos paties šablono viduje. Kaip pavyzdys šiuo atveju naudojama lentelėje nurodyta daugelio gijų palaikymo savybė. Gamyklos atveju objektą gali prieiti daugelis gijų tuo pat metu, nebent jis bus išskiriamas mutex principu iš išorės. Tuo tarpu tiek statybininkas tiek self-resolver objektus, naudoja tarsi per tarpininką, kurio viduje gali būti realizuotas mutex principus taip leidžiant pačiam šablonui tapti draugišku daugybės gijų atžvilgiu.

**Lentelė nr. 1 Šablonų savybių palyginimas**

| Savybė | Statybininkas (angl. Builder) | Gamykla (angl. Factory) | Reflective Self Resolver |
|---|---|---|---|
| Paremtas bazinio tipo naudojimu | ✓ | ✓ | ✓ |
| Galima užtikrinti, jog objektą vienu metu pasieks tik viena gija | ✓ | | ✓ |
| Reikia realizuoti ne daugiau kaip vieną klasę | | ✓ | ✓ |
| Naudojamos tik sąsajos, be jokių abstrakčių klasių | | ✓ | |
| Sistemos praplėtimas, nekeičiant egzistuojančio kodo | | | ✓ |
| Centralizuotas objektų kūrimas | ✓ | ✓ | ✓ |

Lyginant bendrąsias savybes ir žiūrint į šiuos teiginius, kaip į teiginius leidžiančius sistemai tapti lankstesne akivaizdu, jog pagal kiekį lanksčiausias šablonas šiuo atveju būtų „Reflective self-resolver“. Tiesa kiekvieno šablono lankstumas labai priklauso nuo jo realizacijos ir čia

lyginamos tik galimybės realizacijoms, norint nubrėžti tam tikras gaires ieškant geresnio architektūrinio sprendimo.

Projektavimo šablonai yra ganėtinai paprasti, tačiau kartu ir sudėtingi pagalbiniai architektūriniai sprendimai. Jų paskirtis yra leisti projektuotojui ar programuotojui pasinaudoti jau kitų jau sukaupta patirtimi sprendžiant vienas ar kitas problemas. Ir nors teoriškai ir skirti konkrečioms problemos spręsti, tos pat grupės šablonas galima pritaikyti panašiems sprendimams. Tačiau kaip nuspręsti, jog vienas ar kitas šablonas yra labiau tinkamas tam tikroje situacijoje. Bet kurį sprendimą galima paremti patirtimi ir tai greičiausiai nebūtų blogai, tačiau empirinis sprendimo pagrindimas yra vienas iš svarbiausių, bet kokio programinės įrangos projekto aspektų[10].

## 5.1 Šablonų klasių diagramų statinių metrikų tyrimas pasinaudojant M.Genero metodika šablonų palaikomumui nustatyti

Dažniausiai projektavimo šablonai vaizduojami klasių diagramose ir nesigilinant į jų realizaciją vienintelis tinkamas būdas juos tirti yra tirti jų klasių diagramas. Projektavimo lygmens metrikos kol kas nėra pilnai išvystytos kai kalbama apie ryši su programos veikimo charakteristikomis, tačiau yra atliktų tyrimų, kurie gali pagelbėti šioje srityje[13]. Pasinaudosime M. Genero ir jo kolegų aprašytomis klasių diagramos metrikomis [7] [8] (žiūr. Lentelė nr. 2 UML klasių diagramos metrikos).

**Lentelė nr. 2 UML klasių diagramos metrikos[7]**

| Metrika | Apibūdinimas |
|---|---|
| **Klasių kiekis (angl. Number or classes) [NC]** | Bendras klasių ir sąsajų kiekis diagramoje. |
| **Atributų kiekis (angl. Number of attributes) [NA]** | Bendras atributų kiekis diagramoje. |
| **Metodų kiekis (angl. Number of methods) [NM]** | Bendras metodų kiekis diagramoje. |
| **Asociacijų kiekis (angl. Number of Associations) [NAssoc]** | Asociacijos ryšių kiekis diagramoje. |
| **Agregavimų kiekis (angl. Number of aggregation) [NAgg]** | Bendras agregavimo ryšių kiekis diagramoje. |
| **Priklausomybių kiekis (angl. Number of dependencies) [NDep]** | Bendras priklausomybės ryšių kiekis diagramoje. |
| **Apibendrinimų kiekis (angl. Number of generalizations)** | Bendras apibendrinimo ryšių kiekis. |

| | |
|---|---|
| **[NGen]** | |
| **Agregacijos hierarchijų kiekis (angl. Number of aggregation hierarchies) [NAggH]** | Agregacijos hierarchijų skaičius diagramoje. |
| **Generalizacijų hierarchijų kiekis (angl. Number of generalizations hierarchies) [NGenH]** | Bendras generalizacijų hierarchijų skaičius diagramoje |
| **Didžiausias DIT (angl. Maximum DIT) [DIT]** | Didžiausias paveldimumo lygis. Atstumas nuo vienos tėvinės klasės iki jos paskutinio giminaičio. |
| **Didžiausias HAGG (angl. Maximum HAGG) [HAGG]** | Ilgiausias kelias agregavimo hierarchijose. |

Šios metrikos nurodo vienos ar kitos klasių diagramos sudėtingumą ir turi įtakos jų palaikomumui. Kadangi savo eksperimentu siekiame ištirti projektavimo šablonus ieškant lankstesnio sprendimo, kaip ir norime tokio šablono, kurio realizacijos palaikomumo laikas yra mažiausias. Žinoma reiktų atsižvelgti ir į kitus aspektus, kaip paties realizacijos laiką ir papildomus paslėptus aspektus, kuriuos pasirinktas sprendimas teikia realizuotai sistemai.

M. Genero aprašo metodiką [9], kurios pagalba pateiktos metrikos įgyja palaikomumo koeficientus, pagal kuriuos galima vertinti projektavimo šablono įtaką sistemos palaikymui skiriamam laikui (žiūr. Lentelė nr. 3 Koeficientai tarp metrikų ir palaikomumui skiriamo laiko).

**Lentelė nr. 3 Koeficientai tarp metrikų ir palaikomumui skiriamo laiko[7]**

| Metrika | NC | NA | NM | NAssoc | NAgg | NDep | NGen | NAggH | NGenH | Max DIT | Max Hagg |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Palaikomumo koeficientas** | 0.895 | 0.753 | 0.828 | 0.557 | 0.547 | 0.411 | 0.575 | 0.675 | 0.696 | 0.719 | 0.555 |

Pasinaudojant šiais koeficientais patikrinsime analizuojamus šablonus, bei siūlomą architektūrinį sprendimą sistemos lakstumo problemai spręsti. Šablonų metrikų rezultatai rodo, jog paprasčiausia diagrama yra gamyklos šablono (žiūr. Lentelė nr. 4 Klasių diagramų metrikų rezultatai).

**Lentelė nr. 4 Klasių diagramų metrikų rezultatai**

| Metrika | Statybininkas (angl. Builder) | Gamykla (angl. Factory) | Reflective Self Resolver |
|---|---|---|---|
| **Klasių kiekis [NC]** | 10 | 5 | 6 |
| **Atributų kiekis [NA]** | 9 | 3 | 4 |
| **Metodų kiekis [NM]** | 14 | 9 | 10 |
| **Asociacijų kiekis [NAssoc]** | 3 | 1 | 1 |
| **Agregavimų kiekis [NAgg]** | 0 | 0 | 0 |
| **Priklausomybių kiekis [NDep]** | 0 | 1 | 0 |
| **Apibendrinimų kiekis [NGen]** | 3 | 1 | 1 |
| **Agregacijos hierarchijų kiekis [NAggH]** | 0 | 0 | 0 |
| **Generalizacijų hierarchijų kiekis [NGenH]** | 2 | 1 | 1 |
| **Didžiausias DIT [DIT]** | 1 | 1 | 1 |

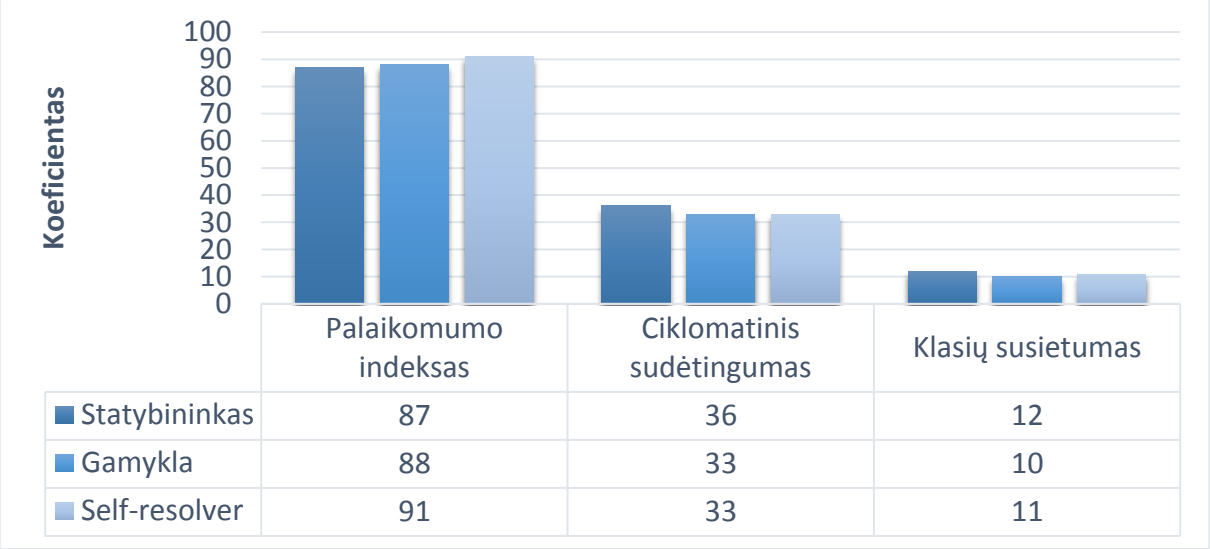
| | | | |
|---|---|---|---|
| Didžiausias HAGG [HAGG] | 0 | 0 | 0 |

Sudauginus lentelėje (žiūr. Lentelė nr. 4 Klasių diagramų metrikų rezultatai) aprašytus duomenis su koeficientų lentelėje (žiūr. Lentelė nr. 3 Koeficientai tarp metrikų ir palaikomumui skiriamo laiko) esančiais duomenimis gauta, jog gamyklos šablonas turi mažiausią palaikomumo laiką programavimo atžvilgiu. Taigi siūlomas self-reflective resolver šablonas atlikus statinę diagramos analizę šiek tiek nusileidžia gamyklos šablonui ir yra pranašesnis už statybininko šabloną. Kita vertus reikėtų nepamiršti, jog statinė diagramos analizės neįvertina tokių dalykų kaip klasių, kurių nereikia realizuoti ir jos skirtos tik įgyti vienam ar kitam funkcionalumui [11], kaip tarkim *Resolver* klasė, bei tiek statybininko tiek self-resolver šablonuose esančių objektų kiautų klasių skirtų tik perduoti vidinių komponentų funkcionalumui, paprastumo.

## 5.2 Šablonų kodo metrikų tyrimas ir jų įtaka sistemos lankstumui, bei palaikomumui

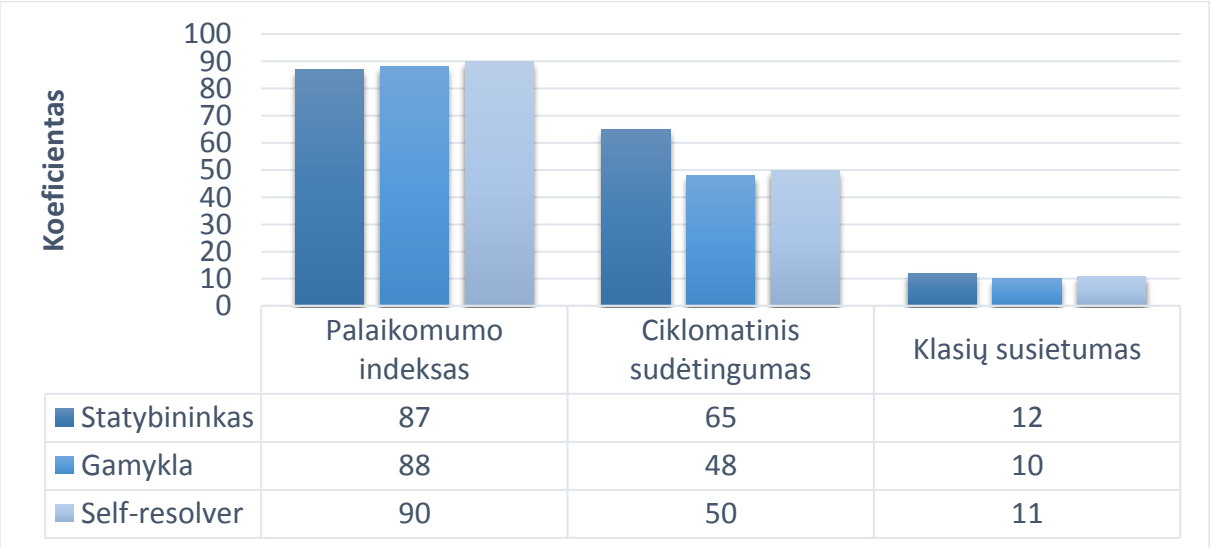
Be statinės diagramų analizės buvo atliktas ir kodo metrikų tyrimas, kurio metu siekta sutvirtinti diagramų metrikų tyrimo metu apskaičiuotus rezultatus, bei patikrinti ar patikslinti juos. Kodo metrikų tyrimo papildomai buvo atliekamas ne tik dėl anksčiau minėtų priežasčių, tačiau ir dėl to, jog keletas diagramų metu į skaičiavimus įtrauktų elementų iš tiesų neturėtų turėti, tokios reikšmės, kokia buvo skiriama skaičiuojant galutinius palaikymui skiriamo laiko rezultatus. Taip pat kodo metrikos nors ir dažnai abejingai sutinkamos kaip programos kokybės ar jos lankstumo matas, tačiau gali nemažai pasakyti ir potencialiai nuvesti tinkamu keliu programos tobulinimo procese [14]. Anksčiau minėta, jog tiek statybininko šablonai tiek self-resolver turi klasių, kurių iš ties nereiktų realizuoti, taigi kuriant naują bazinio objekto realizaciją jos nebūtų paliečiamos, nors diagramų analizės metu į jas buvo atsižvelgiama ir jos ganėtinai nemažai įtakojo galutinius rezultatus.

Kiekvienas iš šablonų buvo realizuojamas dvejais variantais: a) įgyvendinant tik jo struktūrą b) realizuojant pilną funkcionalumą.




**Paveikslas nr. 17 Šablonų be pilnos realizacijos kodo metrikų rezultatai**

Iš lentelės (žiūr. Paveikslas nr. 17 Šablonų be pilnos realizacijos kodo metrikų rezultatai) matome, jog didžiausią palaikomumą indeksą turi self-resolver šablonas. Taip yra todėl, jog vos didesniame kodo kiekyje ciklomatinis sudėtingumas išlaikomas toks pat, be to skaičiuojant Halstead'o metrikas gaunami palankesni rezultatai. Lyginant ciklomatinį sudėtingumą pastebime, jog gamyklos ir self-resolver šablonai turi vienodus ciklomatinio sudėtingumo matavimus ir kiek lenkia statybininko šabloną. Ši metrika ypač svarbi rašant vienetų testus funkcijoms, nes tiesiogiai nusako kelių kiekį kurį reikia padengti norint pasiekti pilną funkcijos padengiamumą. Iš ties daug apie pačius šablonus pasako metrikų prieaugis tarp realizuotos versijos ir tiesiog šablono kiauto. Nepaisant to, jog viena ar kita metrika gali rodyti tam tikro šablono pranašumą prieaugis parodytų realizacijos sudėtingumą ir leistų spręsti apie realią šabloną vertę programiniame kode.




**Paveikslas nr. 18 Šablonų ir jų pilkų realizacijų kodo metrikų rezultatai**

Perskaičiavus rezultatus po pilnos realizacijos matome ryškų pasikeitimą šablonų metrikose. Labiausiai į teigiamąją pusę pasikeitė gamyklos šablonas. Pastebime, jog didėjant jo realizacijos apimčiai jo palaikomumas auga mažiausiai. Sudėtingiausiu vėlgi liko statybininko šablonas, o self-resolver parodė vidutinius rezultatus. Iš to galime spręsti, jog kuo didesnė realizacija tuo labiau mažėja šansas, jog self-resolver šablonas yra lengviau palaikomas už kitus, jeigu jums nėra reikalingi privalumai, kuriuos siūlo šis šablonas.

## 5.3 ŠABLONŲ REALIZAVIMO IR SISTEMOS ATNAUJINIMO TIKRINANT SUGAIŠTĄ LAIKĄ EKSPERIMENTINIS TYRIMAS

Tiksliai išmatuoti šabloną vertę yra sudėtinga net ir remiantis daugybe įvairių metrikų, tačiau visuomet galima atlikti praktinį vieno ar kito sprendimo įvertį konkrečios sistemos lankstumui patikrinti. Galima drąsiai teigti, jog vienas ar kitas architektūrinis sprendimas yra lankstesnis, jeigu su tokia realizacija įmanoma sistemą praplėsti ir atnaujinti greičiau negu naudojant kitą architektūrinį sprendimą.

### 5.3.1 INTERNETINĖS KAMEROS „BIP2-1600-25C-DN“ FUNKCIONALUMO REALIZAVIMO ESANT SKIRTINGOMS SISTEMOS ARCHITEKTŪROMS EKSPERIMENTO APRAŠAS

Eksperimentas atliekamas realizuojant po vieną papildomą įrenginį ir įdiegiant jį į sistemą. Eksperimento metu skaičiuojamas realizavimui skiriamas laikas ir sistemos atnaujinimui skiriamas laikas. Eksperimento metu atliekami žingsniai.

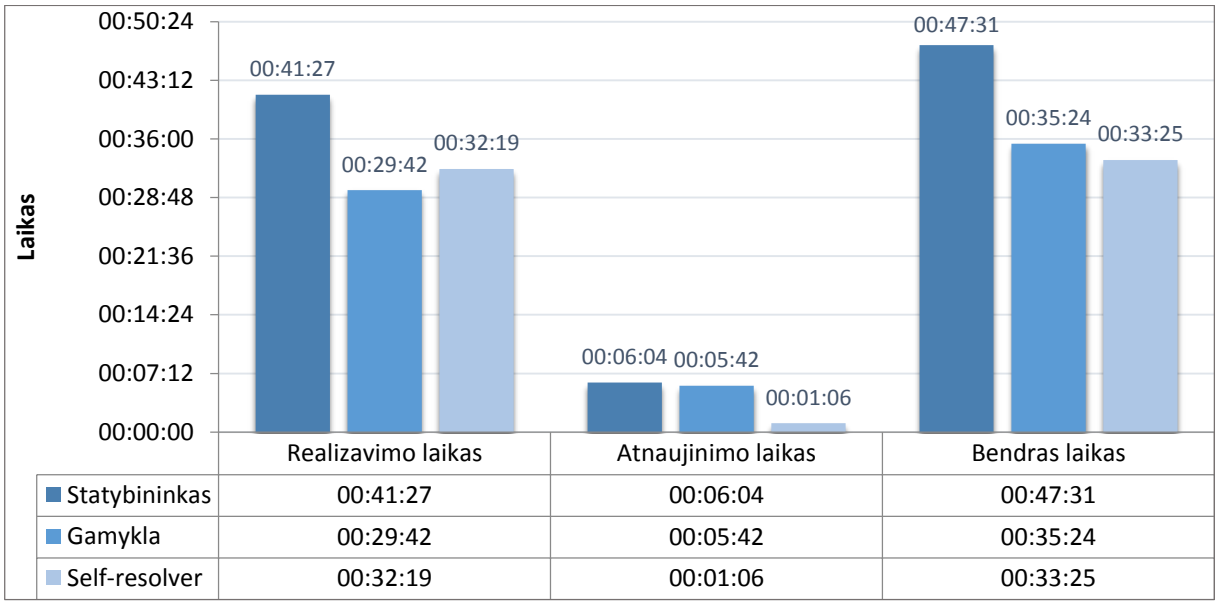
1) Užrašomas pradžios laikas (hh:mm:ss).
2) Įgyvendinama *„BIP2-1600-25c-dn“* internetinės kameros realizaciją. (Eksperimento sumetimais reikalaujama tik vaizdo perdavimo funkcija su išsaugojimo galimybe).
3) Užrašomas tarpinis įgyvendinimo laikas (hh:mm:ss).
4) Atnaujinama sistema (įvykdomi visi reikalingi pakeitimai, jog nauja kameros realizacija būtų prieina ir pilnai funkcionali).
5) Užrašomas galutinis eksperimento laikas (hh:mm:ss).

Šie žingsniai kartojami tris kartus su trimis skirtingomis sistemos versijomis, kuriose yra įgyvendinti skirtingi šablonai įrenginių struktūrai valdyti.

### 5.3.2 „BASLER“ KAMEROS REALIZACIJŲ LAIKŲ APŽVALGA ESANT SKIRTINGOMS SISTEMOS ARCHITEKTŪROMS

Realizuojant šablonus buvo pasirinktas įrenginys, su kuriuo visa komanda puikiai susipažinus, todėl realizacijų tvarka neturėjo didelės įtakos. Eksperimentiniu būdų patikrinus šablonų

realizacijos ir sistemos atnaujinimo laikus pastebėta (žiūr. Paveikslas nr. 19 Šablonų realizacijos ir sistemos atnaujinimo vidutiniai laikai), jog sparčiausiai programuotojams pavyko realizuoti gamyklos šabloną, tačiau mažiausias vidutinis laikas sugaištas sistemos atnaujinimui buvo pasiektas dirbant su reflective self-resolver sistemos architektūra.




| | Realizavimo laikas | Atnaujinimo laikas | Bendras laikas |
|---|---|---|---|
| Statybininkas | 00:41:27 | 00:06:04 | 00:47:31 |
| Gamykla | 00:29:42 | 00:05:42 | 00:35:24 |
| Self-resolver | 00:32:19 | 00:01:06 | 00:33:25 |

**Paveikslas nr. 19 Šablonų realizacijos ir sistemos atnaujinimo vidutiniai laikai**

Reflective self-resolver šablonu pagrįstoje sistemos architektūroje naujo įrenginio realizavimas užtruko 8,82% procento lėčiau negu gamyklos šablonu pagrįstoje architektūroje ir 22,02% procentais greičiau negu statybininko šablonu pagrįstoje architektūroje. Kita vertus sudėjus tiek naujo įrenginio realizavimo, tiek sistemos atnaujinimo laikus gauti rezultatai rodo, jog reflective self-resolver šablonas 5,62% sparta lenkia gamyklos šabloną ir 29,67% statybininko. Taigi sistemose turinčiose didelį kiekį nedidelės apimties objektų, bei dažnai atnaujinamose sistemos ši architektūra gali būti taikoma kaip efektyvi sistemos lankstumą padidinanti architektūra.

# 6 IŠVADOS

Tyrimo metu išspręsti *2.1.1 Programinės įrangos lankstumo įgyvendinimo būdai ir uždavinių formuluotės* skyriuje suformuluoti uždaviniai, bei įvykdyti tikslai parodė:

1. Sukurtas šablonas pakeičiantis objektų turinčių bendrą bazinį tipą kūrimą, bei realizuota architektūra pasinaudojanti dinaminiu bibliotekų užkrovimu suteikia sistemai papildomo lankstumo jos atnaujinimo ir plėtimo atvejais.
2. Eksperimentinio tyrimo metu apskaičiuota, jog esant siūlomai architektūrai sistemos pilnas atnaujinimas sutrumpėja apie 9% procentus.
3. Dėl šios architektūros privalumų atnaujinant sistemą ji ypatingai naudinga sistemose, kuriose vyksta dažni sistemos atnaujinimai.
4. Galimybė atnaujinti sistemą nestabdant jos darbo leidžia ne tik sutaupyti laiko ją atnaujinant, tačiau ir mažinami kaštai, kurie atsiranda linijos sustabdymo metu.
5. Šio šablono realizacijos laikas auga greičiau negu kitų augant realizuojamo objekto sudėtingumui, todėl jis yra pranašesnis, kuriant daugiau, tačiau paprastesnių objektų.

# 7 LITERATŪRA


1. Martin, Robert C. „Design Principles and Design Patterns“ 2000.
2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides „Elements of Reusable Object-Oriented Software“, 1994.
3. http://c2.com/cgi/wiki?GangOfFour.
4. J. Hyslop and H. Sutter. „Virtually Yours“ (C/C++ Users Journal Experts Forum, 18(12), December 2000).
5. Amnon H. Eden, Tom Mens. „Measuring Software Flexibility” IEE Software, Vol. 153, No. 3 (Jun. 2006), pp. 113–126. London, UK: The Institution of Engineering and Technology.
6. M. Lehman. „Laws of Software Evolution Revisited“ Lecture Notes in Computer Science 1149 (Proc. 5th European Workshop on Software Process Technology), pp. 108–124. Berlin: Springer, 1996.
7. M. Genero, M. Piattini, and C. Calero, „Early Measures for UML class diagrams“, L´Objet. 6(4), Hermes Science Publications, (2000), pp. 489-515.
8. M. Genero, “Defining and Validating Metrics for Conceptual Models”, Ph.D. Thesis, Dept. of Computer Science, University of Castilla-La Mancha. Ciudad Real, Spain, (2002).
9. Marcela Genero, Mario Piattini, Coral Calero „An Empirical Study to Validate Metrics for Class Diagrams” Dept. of Computer Science, University of Castilla-La Mancha. Ciudad Real, Spain (2009).
10. N. Schneidewind, „Methodology For Validating Software Metrics”, IEEE Transactions of Software Engineering, 18(5), (1992), pp. 410-422.
11. V. Basili and D. Weiss, “A Methodology for Collecting Valid Software Engineering Data”, IEEE Transactions on Software Engineering, 10, (1984), pp. 728-738.
12. F. Ververs, C. Pronk, „On the interaction between metrics and patterns” In proceedings of OOIS ’95 Dublin, 1995, pp. 303-314.
13. N.E. Fenton, M. Neil, „Software metrics: roadmap“. ICSE-Future of SE, pp 357-370, 2000.
14. N. Subramanian, L. Chung, „Metrics for Software Adaptability“, Applied Technology Division, Anritsu Company, Richardson, TX, USA, 2000.
15. L. Bass, P. Clements and R. Kazman, „Software Architecture in Practice“, SEI Series in Software Engineering, Addison-Wesley, Massachusetts, 1998.

# 8 TERMINŲ IR SANTRUMPŲ ŽODYNAS

**IP** (angl. Inspection Plan) – Kalbama apie tikrinimų planą skirtą apibrėžti, kaip bus tikrinamas vienas ar kitas objektas (tai gali būti gaminys, jo dalis ar net tam tikras procesas).

**SPC** (angl. Statistical Process Control) – tai proceso kontrolės metodologija, leidžianti nustatyti specifines nukrypimų priežastis ir įspėjanti, kad reikalingi korekciniai veiksmai.

**IO** (angl. Inspection Order) – Tikrinimo užsakymas, egzistuojančio tikrinimo plano realizavimas atliekant visus plane nurodytus veiksmus konkrečiai produktų partijai.

**Partija** (angl. Batch) – produkcijos arba žaliavimų partija, tai tarsi grupavimo vienetas skirtas produkcijai bei žaliavoms.

**QMS** (angl. Quality Management System) – Kokybės valdymo sistemos skirtos produkcijos sekimui ir jos kokybės užtikrinimui.

**MES** (angl. Manufacturing Execution System) – Informacinės sistemos skirtos gamyklose vykdomų operacijų valdymui automatizuoti.

**Baziniai Duomenys** (angl. Base data) – bendrai visoje sistemoje naudojami duomenys, turintys antrinę panaudos vertę ir nepriklausomi nuo modulio specifikos.

**Standartinis panaudojimo kelias** - Veiksmų ar operacijų seka atliekama norint sėkmingai pasinaudoti pagrindinėmis sistemos funkcijomis (plano sukūrimas, produkcijos užsakymo sukūrimas, darbų paskirstymas, operacijų pradėjimas/sustabdymas)

**Beta versijos testavimas** – Testavimo tipas kai ne iki galo baigta sistema duodama klientams ir patys sistemos vartotojai atlieka testavimą ja naudodamiesi.

**PLC** (Programmable Logic Controller) – Kompiuterinis valdiklis skirtas automatizuotis elektromechaniniams veiksmas gamyboje.

**Klientinė dalis** – sistemos dalis Server/Client tipo architektūroje, kuria naudojasi sistemos vartotojai ir kuri gauna informaciją iš serverio.

**Projektavimo šablonas** (angl. Design pattern) – architektūrinio sprendimo šablonas, kurį galima taikyti įvairiose situacijose, siekiant gauti vienokią ar kitokią jo siūlomą naudą.

**Sąsaja** (angl. Interface) – bazinis klasės aprašas nusakantis visas būtinas funkcijas bei duomenis, kurie turi būti aprašyti norint realizuoti šią sąsają.

# 9 PRIEDAI

## 9.1 ŠABLONŲ REALIZACIJOS LAIKŲ REZULTATAI

| Builder | | | | | |
|---|---|---|---|---|---|
| Programuotojas | Pradžios laikas | Realizacijos pabaigos laikas | Pabaigos laikas | Realizacija užtruko | Atnaujinimas užtruko |
| | | | | | |
| Programuotojas 1 | 18:07:52 | 18:51:43 | 18:57:36 | 00:43:51 | 00:05:53 |
| Programuotojas 2 | 12:03:49 | 12:43:45 | 12:50:39 | 00:39:56 | 00:06:54 |
| Programuotojas 3 | 17:58:48 | 18:45:04 | 18:51:56 | 00:46:16 | 00:06:52 |
| Programuotojas 4 | 17:58:13 | 18:39:44 | 18:45:41 | 00:41:31 | 00:05:57 |
| Programuotojas 5 | 18:01:22 | 18:38:06 | 18:44:02 | 00:36:44 | 00:05:56 |
| Programuotojas 6 | 18:17:07 | 18:55:15 | 19:00:18 | 00:38:08 | 00:05:03 |
| Programuotojas 7 | 12:05:14 | 12:44:47 | 12:50:36 | 00:39:33 | 00:05:49 |
| Programuotojas 8 | 18:03:37 | 18:41:59 | 18:47:53 | 00:38:22 | 00:05:54 |
| Programuotojas 9 | 18:01:56 | 18:45:10 | 18:50:59 | 00:43:14 | 00:05:49 |
| Programuotojas 10 | 18:12:35 | 18:55:43 | 19:01:30 | 00:43:08 | 00:05:47 |
| Programuotojas 11 | 17:59:20 | 18:44:31 | 18:51:19 | 00:45:11 | 00:06:48 |

| Factory | | | | | |
|---|---|---|---|---|---|
| Programuotojas | Pradžios laikas | Realizacijos pabaigos laikas | Pabaigos laikas | Realizacija užtruko | Atnaujinimas užtruko |
| | | | | | |
| Programuotojas 1 | 18:15:56 | 18:42:49 | 18:47:56 | 00:26:53 | 00:05:07 |
| Programuotojas 2 | 18:12:42 | 18:45:36 | 18:52:10 | 00:32:54 | 00:06:34 |
| Programuotojas 3 | 17:59:13 | 18:34:05 | 18:39:58 | 00:34:52 | 00:05:53 |
| Programuotojas 4 | 17:55:56 | 18:24:53 | 18:30:00 | 00:28:57 | 00:05:07 |
| Programuotojas 5 | 18:15:11 | 18:46:07 | 18:51:52 | 00:30:56 | 00:05:45 |
| Programuotojas 6 | 18:10:42 | 18:37:45 | 18:44:01 | 00:27:03 | 00:06:16 |
| Programuotojas 7 | 18:16:36 | 18:43:25 | 18:49:13 | 00:26:49 | 00:05:48 |
| Programuotojas 8 | 18:01:08 | 18:28:02 | 18:33:11 | 00:26:54 | 00:05:09 |
| Programuotojas 9 | 18:10:06 | 18:40:55 | 18:46:34 | 00:30:49 | 00:05:39 |
| Programuotojas 10 | 18:06:53 | 18:34:40 | 18:40:19 | 00:27:47 | 00:05:39 |
| Programuotojas 11 | 18:01:42 | 18:34:30 | 18:40:19 | 00:32:48 | 00:05:49 |

| Reflective resolver | | | | | |
|---|---|---|---|---|---|
| Programuotojas | Pradžios laikas | Realizacijos pabaigos laikas | Pabaigos laikas | Realizacija užtruko | Atnaujinimas užtruko |
| | | | | | |
| Programuotojas 1 | 18:09:32 | 18:37:53 | 18:38:44 | 00:28:21 | 00:00:51 |
| Programuotojas 2 | 12:07:32 | 12:44:04 | 12:45:36 | 00:36:32 | 00:01:32 |
| Programuotojas 3 | 18:06:30 | 18:38:27 | 18:39:24 | 00:31:57 | 00:00:57 |
| Programuotojas 4 | 18:06:53 | 18:35:35 | 18:36:27 | 00:28:42 | 00:00:52 |
| Programuotojas 5 | 18:05:57 | 18:40:31 | 18:41:55 | 00:34:34 | 00:01:24 |
| Programuotojas 6 | 18:08:48 | 18:37:44 | 18:38:40 | 00:28:56 | 00:00:56 |
| Programuotojas 7 | 12:03:52 | 12:35:44 | 12:36:36 | 00:31:52 | 00:00:52 |
| Programuotojas 8 | 18:02:42 | 18:33:30 | 18:34:18 | 00:30:48 | 00:00:48 |
| Programuotojas 9 | 17:59:05 | 18:33:53 | 18:35:11 | 00:34:48 | 00:01:18 |
| Programuotojas 10 | 12:09:31 | 12:45:02 | 12:46:23 | 00:35:31 | 00:01:21 |
| Programuotojas 11 | 18:05:22 | 18:38:52 | 18:40:04 | 00:33:30 | 00:01:12 |