VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Deklaratyviai apibrėžiamų komponentų architektūra: projektavimas ir realizacija Java sistemoje

The architecture for declarative components: design and Java implementation

Magistro baigiamasis darbas

Atliko:	Justas Valskis	(parašas)
Darbo vadovas:	Doc. dr. Rimantas Vaicekauskas	(parašas)
Recenzentas:	lekt. Mindaugas Plukas	(parašas)

Vilnius – 2012

Reziumė

Šio darbo tikslas yra sukurti metodą, kuris teiktų galimybę kuo didesnę vartotojo sąsajos dalį apibrėžti deklaratyviai (komponentų kūrimas, įvykių klausytojų registravimas, komponentų tarpusavio sąryšių apibrėžimas ir vizualus komponentų išdėstymas languose).

Darbe išanalizuotos įvairios komponentinės architektūros bei išraiškingą vartotojo sąsajos apibrėžimą įgalinantys sprendimai. Apibrėžta ir įgyvendinta architektūra leidžianti kurti deklaratyviai apibrėžiamus vartotojo sąsajos komponentus, kurie deklaruoja savo elgesį ir gali bendrauti tarpusavyje pranešimų pagalba.

Rezultate buvo sukurta sistema, kuri leidžia kurti taikomąsias programas su Java Swing vartotojo sąsaja, naudojantis beveik vien tik deklaratyviais taikomosios programos dalių apibrėžimais. Sukurtos sistemos konfigūracijos galimybės leidžia sumažinti komponentų manipuliacijai skirto Java kodo poreikį.

Summary

A part of a user interface can be created using declarative means. The main goal of this paper is to maximize that portion. To do that, said means would be used to create components, register event listeners, describe relationships between components and layout them in windows.

Various component-based architectures and existing solutions for declarative user interface definition were analyzed. As a result, an architecture was defined which allows user interface components to be created using a declarative approach. These components define their behavior and use messages to communicate with each other.

The outcome was a system, which can create Java Swing applications using mostly declarative definitions of various parts of said applications. This system provides various configuration options that reduce the need of writing Java code for component manipulation.

Turinys

Įvadas

Programinės įrangos komponentas yra modulis, kuris apima rinkinį semantiškai susijusių duomenų bei funkcijų. Komponentai yra nepriklausomi nuo kitų komponentų ir tarpusavyje bendrauja per sąsajas – komponentas siūlantis savo funkcionalumą kitiems komponentams paviešina sąsają per kurią yra pasiekiamas jo teikiamas funkcionalumas. Panašiai yra ir kai komponentui reikia kito komponento funkcionalumo tam kad galėtų funkcionuoti. Tokiu atveju komponentas paviešina sąsają, kuri apibrėžia, kokių paslaugų jam reikia.

Dėl išvardintų komponentų savybių, klientui naudojančiam komponento funkcionalumą nereikia nieko žinoti apie jo realizaciją. Taip pat vieni komponentai gali būti pakeisti kitais, jei naujieji komponentai tenkina tuos pačius reikalavimus kaip ir keičiami komponentai. Dėl tų pačių priežasčių komponentai taip pat gali būti ir pakartotinai panaudojami – tiesa tai jau priklauso nuo konkrečios komponento realizacijos.

Komponentinę sistemą sudaro iš anksto paruošti bei tarpusavyje komponuojami elementai. Šie komponentai turėtų būti nepriklausomi vieni nuo kitų ir sukurti taip, kad juos būtų galima pakartotinai naudoti.

Pritaikius šias savybes taikomajai programai su vartotojo sąsaja, ją sudarantys komponentai turėtų deklaruoti savo elgesį, o įvykių klausytojai (elgesys) neturėtų tiesiogiai manipuliuoti kitais komponentais. Dėl šios priežasties bendravimas tarp atskirų komponentų turėtų vykti sąsajų pagalba.

Vartotojo sąsajos komponentai yra naudojami įvairiais tikslais, tarp kurių yra duomenų rinkimas bei atvaizdavimas. Šie komponentai naudoja ne vien savo, tačiau ir kitų komponentų duomenis. Duomenys, kurie yra sukuriami vieno komponento pagalba, gali būti reikalingi kitiems komponentams – taip formuojamos tarpusavio priklausomybės.

Tokius vartotojo sąsajos komponentus būtų galima laikyti statybiniais blokais. Jų elgsena bei priklausomybės nuo kitų komponentų būtų nesunkiai konfigūruojamos, o tai įgalintų komponentų pritaikymą įvairiems naudojimo scenarijams, nekeičiant nei vienos kodo eilutės.

Darbo tikslas

Darbo tikslas yra sukurti mechanizmą leidžiantį kuo didesnę vartotojo sąsajos dalį (komponentų kūrimą, įvykių klausytojų registravimą, komponentų tarpusavio sąryšių apibrėžimą ir vizualų komponentų išdėstymą languose) apibrėžti deklaratyviai.

Tam tikslui būtina apibrėžti ir įgyvendinti architektūrą, leidžiančią kurti Java vartotojo sąsajos, įskaitant ir Swing, komponentus deklaruojančius savo elgseną bei nesančius glaudžiai susietiems su kitais komponentais nuo kurių jie priklauso. Kad komponentai būtų nepririšti

vienas prie kito, tačiau ir toliau galėtų sąveikauti, komponentams yra reikalinga sąsaja bendravimui ir atskiras mechanizmas kuris tą sąveiką reguliuotų.

Tuo siekiama sumažinti dažnai pasikartojančio kodo rašymą, kuris iš esmės skirtas komponentų reikšmių išgavimui ar nustatymui. Tuo pačiu, supaprastinti Java Swing vartotojo sąsajos kūrimo procesą, palengvinti sukurtos sąsajos priežiūrą bei leisti kurti grafines vartotojo sąsajas su nesunkiai konfigūruojamais komponentais, jų tarpusavio priklausomybėmis bei elgesiu.

Šaltiniai

Komponentinės sistemos nėra naujiena, todėl darbe buvo apžvelgtos įvairios jungtimis paremtomis komponentinės architektūros. Daugiausia dėmesio buvo skiriama komponentų sandarai, jų tarpusavio komunikacijai bei jungčių vaidmeniui dviejų komponentų kompozicijoje.

Išnagrinėtos galimos jungčių funkcijos bei galimybės, įvairūs komponentų sujungimo būdai ir iš to kylančios suderinamumo problemos bei jų rūšys, ir nesuderinamų komponentų adaptavimo strategijos.

Kadangi vienas iš tikslų yra deklaratyvus vartotojo sąsajos apibrėžimas, todėl buvo apžvelgti egzistuojantys deklaratyvų Java Swing vartotojo sąsajos apibrėžimą įgalinantys karkasai. Buvo apžvelgtos jų galimybės – kurie vartotojo sąsajos ir ją sudarančių komponentų kūrimo aspektai yra supaprastinti, kuri vartotojo sąsajos dalis gali būti išreikšta deklaratyviai bei kaip lengvai šie karkasai gali būti praplečiami.

Darbe įgyvendinta komponentinė architektūra pasižyminti tokiomis savybėmis: komponentai yra generuojami programos vykdymo metu, įvykių klausytojai gali būti kuriami dinamiškai bei komponentų tarpusavio priklausomybės išreiškiamos jungčių pagalba. Visos šios savybės detaliau aprašomos tolesniuose skyriuose.

Darbo struktūra

Darbas susideda iš įvado, keturių skyrių, išvadų, literatūros sąrašo ir priedų.

Pirmame skyriuje yra abstrakčiai aprašoma sukurta sistema – kokios yra pageidaujamos sistemos savybės, komponentų paskirtis bei funkcijos ir kaip deklaratyviai apibrėžiama vartotojo sąsaja.

Antrame skyriuje yra vykdoma įvairių komponentinių architektūrų stilių bei egzistuojančių karkasų, kurie leidžia Java Swing vartotojo sąsają apibrėžti deklaratyviai, apžvalga. Nagrinėta komponentų sandara, jų kompozicijos bei komunikavimo problemos.

Trečiame skyriuje yra detaliai aprašoma sukurtos sistemos ir jos sudedamųjų dalių paskirtis, sandara bei pateikiamas pavyzdinis deklaratyvus tų dalių apibrėžimas. Taip pat aprašomos ir įvairios sistemos funkcijos ir galimybės.

Ketvirtame skyriuje aprašomi naudoti bendro naudojimo komponentai ir demonstruojamas prototipas sukurtas naudojantis darbo metu sukurta sistema, kurios kūrimo procesas bei rezultatas yra palyginamas identišką funkcionalumą turinčia programa, kurioje nebuvo naudojama sukurta sistema.

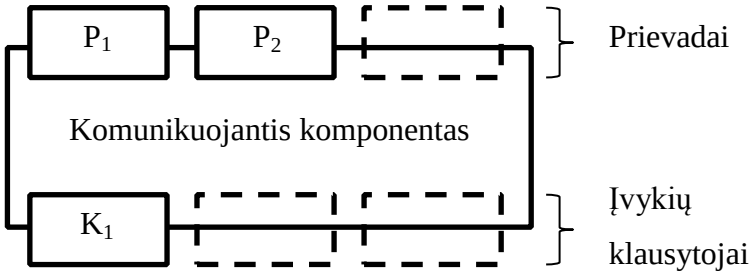
Išvadose pateikiami pagrindiniai magistrinio darbo rezultatai.

1. Sistemos aprašymas

Komponentus yra siekiama apibrėžti kaip programos statybinius blokus, todėl didelis dėmesys yra skiriamas komponentų tarpusavio komunikacijai, tam kad būtų išvengta glaudaus komponentų susiejimo (tiesioginės manipuliacijos komponentais). Dėl šios priežasties komponentų tarpusavio sąveika vyks tik pranešimų pagalba. Dėl savo veikimo principo šie komponentai bus vadinami komunikuojančiais komponentais.

Už komunikuojančių komponentų kompoziciją yra atsakingas atskiras mechanizmas. Komponentai tarpusavyje sujungiami konfigūruojamomis jungtimis. Tai leidžia komponentams tiesiogiai nepriklausyti vieniems nuo kitų bei palengvina jų tarpusavio priklausomybių konfigūravimą.

Komunikuojantis komponentas, tai paprastas (dažniausiai Java Swing) komponentas su konfigūruojama sąsaja (prievadai) bei elgesiu (įvykių klausytojai) (1 pav). Dėl elgesio ir priklausomybių konfigūravimo, kuris atliekamas deklaratyviai, komunikuojantys komponentai yra lengvai pakartotinai panaudojami – nereikia keisti komponento realizacijos.



1 pav. Komunikuojančio komponento sandara. P – prievadai, K – įvykių klausytojai.

Komunikuojančio komponento įvykių klausytojai moka reaguoti į vidinius, generuojamus originalaus komponento, ir išorinius įvykius (kitų komponentų atsiųsti pranešimai).

Prievadai gana glaudžiai susiję su įvykių klausytojais, kadangi komunikuojančio komponento bendravimas vyksta būtent per tuos prievadus. O visa komponento veikimo logika yra realizuojama įvykių klausytojais, ir tik klausytojai yra atsakingi už bendravimo iniciavimą, kaip kad, pavyzdžiui, visos, galimai kitiems komponentams įdomios, informacijos pasidalinimu (įvairūs būsenos pasikeitimai ir t.t.).

1.1. Reikalavimai komponentams bei sistemai

Pradedant projektuoti sistemą, buvo nustatytos pageidaujamos komponentų ir sistemos elgesio bei sandaros savybės:

1) Komponentai deklaruoja savo elgesį – į komponento aprašą įeina išorinių ir vidinių įvykių apdorojimas.

2) Susiję komponentai vienas nuo kitų yra atsieti (angl. Decoupled), todėl bendravimas tarpusavyje galimas tik pranešimų pagalba – per prievadus.
3) Pranešimų perdavimo valdymas – užlaikymas, filtravimas bei konvertavimas. Turi būti galima apibrėžti pranešimų perdavimo logiką.
4) Komponentų konstravimą, sąryšius tarp komponentų ir komponentų vizualinį išdėstymą turi būti galima išreikšti deklaratyviai.

1.2. Deklaratyvūs dalių aprašai

Apibrėžti langui naudojantis kuriama sistema teks paruošti trijų rūšių aprašus – visi jie bus išanalizuoti ir galutinis rezultatas bus apjungtas bei lems tai, kaip veiks taikomoji programa. Šios trys aprašų rūšys yra:

1) komunikuojančių komponentų aprašas (įeina prievadų ir elgesio apibrėžimas – detaliau 3.1.6).
2) komponentų tarpusavio sąryšių aprašas (3.2.4).
3) vizualaus komponentų išdėstymo languose bei tų komponentų išvaizdos nustatymų aprašas (3.3).

Šitoks interaktyvios vartotojo sąsajos apibrėžimo būdas buvo pasirinktas dėl to, kad tai supaprastina kiekvieną atskiros paskirties dokumentą. Taip pat dokumentai tampa nepriklausomi, todėl galima išskirstyti tų dokumentų atsakomybes pagal jų paskirtį. Pavyzdžiui, prireikus galima panaudoti jau egzistuojančius įrankius ar karkasus, konkrečios dalies interpretacijai atlikti – kaip tai buvo padaryta su 3 dalimi – atvaizdavimu.

2. Literatūros apžvalga

2.1. Komponentinių architektūrų stiliai

2.1.1. Bendro pobūdžio architektūrinis karkasas

Autoriai [OEK+10a, OEK+10b] pristato bendro pobūdžio architektūrinis karkasą, kuris yra skirtas modeliuoti komponentais paremtas sistemas. Komponentai tose sistemose yra sujungiami jungtimis. Šis autorių pristatomas karkasas remiasi [AG97] pristatomu architektūrinio apjungimo (angl. Architectural connection) sprendimu. Yra išskiriamos dvi esybės – komponentai ir jungtys.

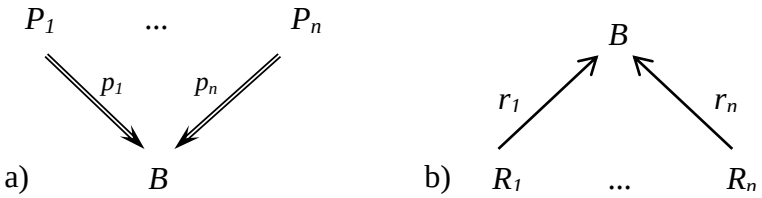
2.1.1.1. Komponentai

Komponentų paskirtis yra atlikti apibrėžtas funkcijas. Tarpusavyje komponentai gali būti sujungiami tik netiesiogiai – su jungčių pagalba. Komponentai tarpusavyje jungiami tam, kad gautų reikalaujamą funkcionalumą iš kitų komponentų.

Komponentas susideda iš kūno ir n ($n \geq 0$) sąsajų – prievadų (iliustruota 2 pav). Kūne realizuojamas komponento teikiamas funkcionalumas, o sąsaja nusako to komponento elgesį ir apibrėžia kaip pasiekti tą funkcionalumą.

2.1.1.2. Jungtys

Jungčių paskirtis yra sujungti komponentus; apsprendžia sujungtų komponentų sąveiką. Kaip ir komponentai, jungtys taip pat susideda iš 2 dalių: kūno ir n ($n \geq 2$) sąsajų, vadinamų rolėmis (iliustruota 2 pav).



2 pav. Komponento (a) ir jungties (b) diagramos (P – prievadai, R – rolės, B - kūnas). Šaltinis [OEK+10a].

Rolės nusako laukiamą komponentų, kurie prijungti prie konkrečios jungties, elgesį bei teikiamą funkcionalumą. Tuo tarpu kūnas apibrėžia tų rolių sąveiką – koordinuoja prijungtus komponentus.

2.1.1.3. Savybės

Komponentai sujungiami su jungtimis susiejant jungties roles su komponento prievadais. Viena jungtis gali sujungti daug komponentų. Susiejimas įmanomas tik tada kai rolės ir prievadai yra suderinami – prievadas turėtų būti tam tikra rolės specializacija.

Autoriai apibrėždami šį karkasą koncentravosi į šias tris jo savybes:

- Kompozicionalumas (angl. Compositionality) – architektūrinės sistemos semantika turėtų priklausyti tik nuo komponentų, jungčių bei sujungimų tarp jų.
- Hierarchinis projektavimas (angl. Hierarchical design) – turėtų egzistuoti galimybė turimą posistemę ar sistemą traktuoti kaip komponentą.
- Specializavimo galimybė – iš esmės ji reiškia tai, jog turint tam tikrą sistemą, ir joje pakeitus vieną tos sistemos komponentą kitu – labiau specializuotu – pati sistema taip pat tampa labiau specializuota.

2.1.2. C2 architektūrinis stilius

Šis architektūrinis stilius, priešingai nei prieš tai pristatytas, yra pritaikytas aplikacijoms turinčioms vartotojo sąsają. Autoriai [TMA+96] teigia jog šis stilius įgalina paprastesnį turimų komponentų pakartotinį panaudojimą.

Šiame architektūriniame stiliuje taip pat yra du (tokie patys) elementų tipai: komponentai ir jungtys. Komponentai tarpusavyje bendrauja tik netiesiogiai – siųsdami pranešimus jungčių pagalba. C2 stilius nedaro jokių prielaidų apie kalbas kuriomis realizuoti komponentai ir jungtys.

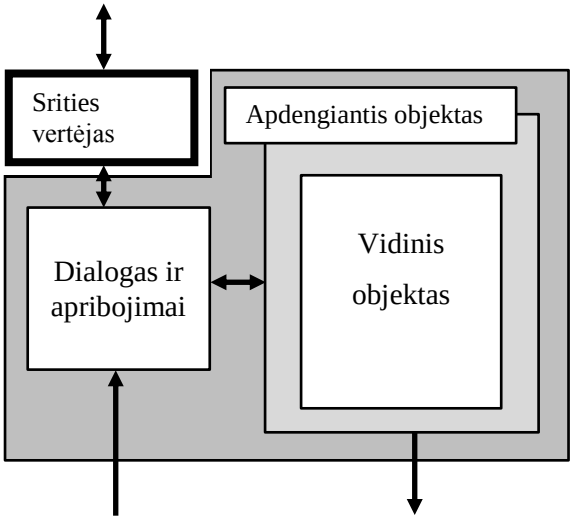
2.1.2.1. Komponentai

Komponentai – esybės galinčios atlikti tam tikras operacijas ir turėti būseną. Turi dvi komunikavimui skirtas sąsajas: viršutinę ir apatinę.

Viršutinė ir apatinė sąsajos nusako rinkinius pranešimų kuriuos siunčia ir priima komponentas. Per viršutinę sąsają priimami perspėjimai (angl. Notification) ir siunčiami prašymai (angl. Request). O per apatinę – atvirkščiai – priimami prašymai, o siunčiami perspėjimai.

Komponentai vienas su kitu tiesiogiai nesąveikauja. Vienas komponentas negali pasiekti kito komponento duomenų ar vykdyti pastarojo operacijas. Komponentas daugiausia gali būti prijungtas prie dviejų jungčių – viena jungtis prie viršutinės sąsajos, o kita prie apatinės.

2.1.2.1.1. Komponento sandara



3 pav. C2 stiliaus komponento sandara. Šaltinis [TMA+96].

Komponentas susideda iš šių dalių (iliustruota 3 pav):

- **Apdengtas vidinis objektas** - rūpinasi, kad kai yra iškviečiamas vidinio objekto sąsajos metodas, komponentas sukurtų perspėjimą su kvietimo informacija (kviečiamas metodas ir jo atributai) ir su apskaičiuotu rezultatu gautu po iškvietimo. Šis perspėjimas yra išsiunčiamas į jungtį prijungtą prie šio komponento apatinės sąsajos. Skleidžiamų perspėjimų rinkinį apibrėžia vidinio objekto sąsaja.
- **Dialogas ir apribojimai** (angl. Dialog and Constraint manager) – reaguoja į kitų komponentų siunčiamus prašymus. Taip pat kviečia vidinio objekto metodus:
    a) Reaguojant į iš viršutinės jungties atsiųstą perspėjimą
    b) Įvykdyti prašymą atsiųstą iš žemiau esančios jungties
    c) Palaikant tam tikrą apribojimą
- **Srities vertėjas** (angl. Domain translator).

2.1.2.2. Jungtys

Jungtys atlieka dvi funkcijas: susieja komponentus ir perduoda pranešimus, ir tuos pranešimus filtruoja bei prioretizuoja. Jie gali būti sujungti su kiek norima daug komponentų arba kitų jungčių. Kaip ir komponentai, jungtys turi viršutinę ir apatinę sąsajas, tačiau jos apsprendžiamos pagal prijungtus komponentus arba kitas jungtis.

Tam kad komponentai šio stiliaus architektūroje galėtų būti labiau nepriklausomi ir pakartotinai panaudojami, nėra garantuojama kad išsiųsti pranešimai pasieks jų gavėją. Tai yra, pranešimai gali būti pamesti arba atfiltruoti.

2.1.2.3. Pranešimai

Komponentai bendrauja asinchroniniais pranešimais. Pranešimai susideda iš jų vardo bei tipizuotų parametrų. Jie būna dviejų rūšių: perspėjimai ir prašymai.

Perspėjimai yra skirti pranešti apie komponento (tiksliau tai jame esančio vidinio objekto) būsenos pasikeitimą arba atliktą operaciją, o prašymų pagalba komponentams nurodoma įvykdyti tam tikrą operaciją.

C2 stilius taip pat reikalauja, kad komponentų siunčiami perspėjimai atitiktų to komponento atliekamas operacijas, o ne kitų komponentų, reaguojančių į tuos perspėjimus, poreikius. Šis apribojimas padeda komponentams būti labiau nepriklausomiems ir lengviau pakartotinai panaudojamais.

2.1.3. JavaBeans komponentų modelis

Architektūra skirta kurti bei naudoti komponentus Java kalboje. Naudojantis JavaBeans API (angl. Application programming interface) [Sun97] galima kurti pakartotinai panaudojamus komponentus.

Komponentai gali būti dinamiškai kontroliuojami bei surinkti tam kad suformuotų konkrečias taikomąsias programas. Komponentai privalo būti kuriami laikantis apibrėžtų kūrimo taisyklių – tai leidžia nustatyti to komponento ypatybes.

Įvairūs komponento atributai nusako komponento išvaizdą bei elgesio charakteristiką. Šios atributus komponentai gali paviešinti, tokiu būdu leisdami kūrimo metu juos keisti. Tokiu būdu yra supaprastinamas šių komponentų modifikavimas ar elgesio keitimas.

Komunikavimui tarpusavyje šie komponentai naudoja įvykius. Komponentai gali patys kurti bei priimti įvykius. Komponentas priimantis įvykius turi užsiregistruoti pas komponentą, kuris tuos įvykius siunčia.

Įvykus tam tikram įvykiui, komponentas A siųs pranešimus kitiems komponentams, kurie pareiškė susidomėjimą konkrečiu įvykiu – užsiregistravo pas komponentą A. Kokius įvykius komponentas moka apdoroti, ir kokius jis gali siųsti galima sužinoti „apžiūrėjus“ patį komponentą.

Komponentų būseną galima išsaugoti ir atstatyti. Šiam funkcionalumui užtikrinti JavaBeans architektūra naudoja Java Object Serialization.

2.1.4. Kitos jungtimis paremtos architektūros

2.1.4.1. Komponentai

Kitur darbuose [FDH08, BB07, AO09, BBC02], labiau tyrinėjama komunikacija tarp komponentų. Patys komponentai yra laikomi juodomis dėžėmis, kurios realizuoja tam tikrą funkcionalumą bei turi tam tikrus prievadus, per kuriuos jie reikalauja ir teikia paslaugas.

Teikiamos komponento paslaugos atspindi jo realizuojamą funkcionalumą, o reikalaujamos paslaugos yra tos, be kurių komponentui neitų teikti paslaugų. Komponento sąsaja susideda iš jo reikalaujamų ir teikiamų paslaugų specifikacijų.

2.1.4.1.1. Prievadai

Išskiriamos dvi komponentų prievadų rūšys – įvesties ir išvesties. Šie prievadai yra vienakrypčiai. Taip pat galimi ir dvikrypčiai prievadai. Dvikrypčiai prievadai suteikia tas pačias galimybes kaip ir vienakrypčiai, tačiau leidžia išreikšti sudėtingesnius komponentų kooperavimo scenarijus. Pavyzdžiui, komponentas A teikia paslaugą X, tačiau jam reikia paslaugos Y. Tai kitas komponentas prisijungęs prie komponento A prievado ir suteikdamas jam paslaugą Y, savo ruoštu gaus paslaugą X. Dvikryptės jungtys leidžia tiksliau išreikšti priklausomybes tarp komponentų [FDH08].

2.1.4.1.2. Apibrėžimas

Komponentų apibrėžimą galima padalinti į dvi dalis. Tai komponento prievadų apibrėžimas – komponento sąsaja. Ir komponento elgsenos aprašas, kurį apibrėžti yra daug sudėtingiau.

Pavyzdžiui [BBC02] siūlo komponentus apibrėžti naudojant IDL (angl. Interface Description Language). Sąsajoje atsispindėtų galimos sąveikos per konkrečius komponento prievadus. Tuo pačiu jos įgalintų automatinę komponentų veiklos patikrą bei komponentų dokumentavimą.

2.1.4.2. Komponentų tarpusavio sąveika

Dviejų komponentų tarpusavio sąveikai apibrėžti [WH05] siūlo pirmiausia atskirai apibrėžti kiekvieno iš tų komponentų elgesio ypatumus. Žinant tai, bus galima pasakyti ir kaip tie komponentai elgsis juos sujungus.

Pačią komponentų sąveiką autoriai laiko siunčiamų pranešimų ir sąveikoje dalyvaujančių komponentų sąveikavimo protokolą. Toliau detaliau apie kiekvieną iš jų.

2.1.4.2.1. Pranešimai

Pranešimas yra laikomas abstrakčiu ir tipizuotu pernešamų duomenų apibrėžimu – kiekvienas pranešimas perneša konkretų objektą.

Pranešimas susideda iš identifikatoriaus bei iš pernešamo objekto atributų poaibio (pranešimo turinys gali būti visas objektas arba dalis objekto). Pernešamo objekto atributai yra komponentų įvesties arba išvesties vektorių elementai.

2.1.4.2.2. Sąveikos protokolas

Sąveikos protokolas nusako komponentų bendravimo ypatumus. Jo paskirtis yra apibrėžti galimus komponentų komunikavimo veiksmus bei apribojimus teisingam pranešimų eiliškumui – kokia tvarka turi būti siunčiami bei priimami pranešimai.

2.2. Komponentų sujungimas

Ankstesniuose skyriuose buvo aprašyti įvairūs jungčių panaudojimo bei apibrėžimo būdai. Vadovaujantis netiesioginiu komponentų bendravimo principu, jungtis tampa lygiaverte esybe komponentui. Jei komponentas yra atsakingas tik už savo paties veikimo logiką – paslaugų teikimą, tai jungtis tampa atsakinga už visa kita – pranešimų perdavimą, jų koordinaciją ir, galimai, komponentų suderinimą.

Sąryšius tarp komponentų ne visada galima ar norima apibrėžti iš anksto. Tokiu atveju šie sąryšiai tarp komponentų turės būti sukuriami dinamiškai.

2.2.1. Dinaminis komponentų sujungimas

[WH05] yra tyrinėjamas dinaminis komponentų sujungimo mechanizmas, kuriame jungtys yra dinamiškai kuriamos „pagal pareikalavimą“. Šie ryšiai sudaromi tarp komponentų, ir yra trumpalaikiai – egzistuoja tol, kol vienam komponentui iš kito tebereikia pastarojo teikiamos paslaugos.

2.2.2. Derybos

Visas bendravimas tarp komponentų vyksta derybų principu. Derybų metu komponentai sprendžia ar jiems kito komponento siūloma paslauga tinka ir ar galės tinkamai bendrauti su tuo komponentu. Derybų eiga vyksta trimis etapais:

1) Komponentas A paskelbia kad jam reikia tam tikros paslaugos X. Skelbime taip pat nurodoma kaip šis komponentas elgsis kooperacijos su kitu komponentu metu – nurodoma komponento elgsena.
2) Komponentai kurie gavo šį skelbimą pirmiausia patikrina ar jie galės suteikti reikalaujamą paslaugą X. Tada nusprendžia ar jų elgesys yra suderinamas su komponento A elgesiu. Komponentams nustačius jog abu šiuos punktus jie įgyvendina, jie gali pareikšti jog kooperacija su komponentu A yra galima.
3) Komponentas A gavęs pareiškimus gali pasirinkti vieną komponentą B, su kuriuo ir sudarys dinaminį ryšį, tam kad galėtų vykti bendradarbiavimas.

Komponentams nutarus sudaryti ryšį, bus sukuriamas dinaminė jungtis, kuris egzistuos tol, kol komponentui A reikės paslaugos iš komponento B.

2.2.2.1. Dinaminės jungties sandara

Jungtis naudojama dinaminiams komponentų ryšiams sudaryti, laiko informaciją apie prie jos prijungtus komponentus bei atlieka tarpininko funkciją – adaptuoja komponentus, tam kad sąveika tarp komponentų vyktų sklandžiai.

2.2.2.2. Reikalavimai sujungimui

Jei laikysim, kad siunčiami duomenys identifikuojami vardais, tai tam kad eitų automatiškai generuoti dinamines jungtis, reikalingas toks metodas, kuris susietų tuos vardus tarp siuntėjo ir gavėjo. Todėl siunčiami pranešimai turi neštis ne tik turinį, bet ir turinio duomenų aprašą. Aprašu pasinaudojusi sujungimus organizuojanti jungties dalis – adapteris, gaunamą pranešimą iš vieno komponento perduoda jį kitam tokia forma, kokia gavėjas pasirengęs priimti.

2.2.3. Komponentų suderinamumas

Norint kad sąveika tarp tarpusavyje sujungtų komponentų vyktų sklandžiai, jie privalo būti suderinami. Tačiau gali nutikti taip, kad komponentai bandys vienas su kitu sąveikauti skirtingai, dėl šios priežasties jiems „susikalbėti“ nepavyks. [WH05] ir [BBC02] išskiria du skirtingus komponentų sąveikos nesutapimo lygius: sąsajos ir protokolo.

2.2.3.1. Sąsajos lygis

Šio lygio nesutapimai pasireiškia nesutapimais siunčiamuose pranešimuose. Pranešimą siunčiantis komponentas suformuoja ne tokį pranešimą, kokio tikisi gavėjas. Kaip pavyzdį galima pateikti [WH05] aprašytus šio lygio nesutapimo variantus:

1) **Vardų konfliktas** – pavadinimai naudojami identifikuoti tas pačias duomenų esybes nesutampa įeinančiame ir išeinančiame pranešimuose. Pavyzdžiui: komponentas A laukia duomenų pavadintų „kaina“, tačiau komponentas B išsiunčia duomenis pavadintus „mokestis“.
2) **Duomenų blokų nesutapimas** - galimas siunčiamų duomenų atributų skaičiaus bei jų išsidėstymo tvarkos nesutapimas.

Tinkamai apibrėžiant komponentų sąsajas, šio lygio nesutapimus galima aptikti bei ištaisyti. Tačiau netgi ir ištaisius šio lygio problemas, nėra jokios garantijos kad komponentai galės toliau sėkmingai komunikuoti. Kadangi gali būti nesutapimų ir protokolo lygyje.

2.2.3.2. Protokolo lygis

Šiame lygyje nesutapimai pasireiškia dėl skirtingų komponentų bendravimo ypatumų. [BBC02] ir [WH05] išskiria dvi priežastis, dėl ko gali kilti protokolo lygio nesutapimo problemos:

1) Blogas pranešimų eiliškumas – komponentas A siunčia pranešimus ne tokia tvarka, kokia tų pranešimų laukia komponentas B.
2) Aklavietė (angl. Deadlock) – komponentai užsiblokuoja. Pavyzdžiui: komponentas A laukia pranešimo X, tam kad galėtų išsiųsti Y, o komponentas B savo ruožtu laukia pranešimo Y, tam kad galėtų išsiųsti X.

Du komponentai yra laikomi suderinamais tada, kai neaptinkama jokių sąsajos ir protokolo lygių nesutapimų tarp šių komponentų. Jei komponentai visgi nėra suderinami, bet juos sujungti reikia, galima bandyti tuos nesutapimus taisyti – adaptuoti komponentus.

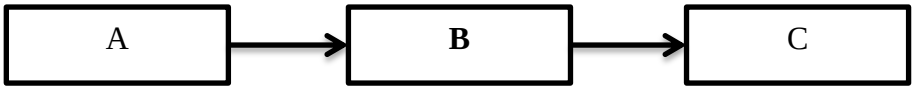
2.2.4. Komponentų adaptavimas

Nesutampant komponentų tarpusavio bendravimo ypatumams, tų komponentų paprastai neis sujungti. Iškyla komponentų adaptavimo problema – reikalingas mechanizmas kurio pagalba būtų įmanoma taip sujungti tuos komponentus, kad jie galėtų sėkmingai sąveikauti vienas su kitu.

Komponentų adaptavimas tai gana plati bei gana svarbi komponentais paremtos programinės įrangos kūrimo problema [Cam99, GS01]. Toliau bus aprašyti keli galimi komponentų adaptavimo sprendimų būdai, kuriems nereikia tiesioginės komponentų modifikacijos, nesileidžiant į jų realizacijos principus.

2.2.4.1. Adaptuojantis komponentas

Vienas iš būdų sujungti du nesutampančius komponentus, yra naudoti trečią komponentą, kurio paskirtis būtų tarpininkauti tarp jų (4 pav). Tarpininkaujančio (adaptuojančio) komponento paskirtis būtų ne tik rūpintis teisingu pranešimų konvertavimu (į kurį gali įeiti duomenų tipų konvertavimas, parametrų jei tokie yra, eiliškumo keitimas arba pašalinimas ir t.t), tačiau galimai ir protokolo nesutapimų taisymu. Toks komponentas iš išorės niekuo neišsiskiria iš kitų, tiek kad jis yra grynai pritaikytas konkrečiai kelių komponentų kompozicijai įgyvendinti.



4 pav. Komponentai A ir C sujungti adaptuojančio komponento B pagalba.

Galimybę tokius komponentus sukonstruoti dinamiškai, pagal tam tikrus komponentų požymius nagrinėja [BBC02]. Nagrinėjami tarpininkaujančių komponentų generavimo bei jų korektiško veikimo tikrinimo būdai.

Tam tikslui komponentams apibrėžti naudojama IDL. Apibrėžiamos komponentų įeigos ir išeigos bei sąveikos šablonas (angl. Behavioral pattern). Tai leidžia automatiškai surasti

nesutapimus tarp dviejų komponentų bei panaudoti tą informaciją dinaminiam adaptuojančio komponento sukūrimui.

2.2.4.2. Adaptuojanti jungtis

Iš principo identiškas komponentų adaptavimo būdas kaip ir pristatytas prieš tai. Tik šiuo atveju, tarpininko vaidmenį atliktų pati jungtis (5 pav). Iš esmės tai reiškia, jog jungtis privalo turėti tokią pačią adaptavimo logiką kaip ir praeitame pavyzdyje pateiktas komponentas-tarpininkas.

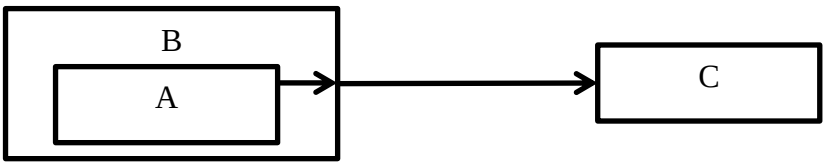
5 pav. Komponentai A ir C sujungti adaptuojančia jungtimi.

Esminis šių dviejų būdų skirtumas yra tas, kad pirmuoju būdu jungtys naudojamos tik informacijos pernešimui, visą kitą darbą atlieka komponentai. Kur šiuo atveju jungtis turi turėti papildomą tarpininkavimo funkcionalumą.

Toks komponentų sujungimo principas nagrinėjamas [WH05], kur ryšiai tarp komponentų yra kuriami dinamiškai. Kaip ir [BBC02], komponentų sąsaja bei sąveika yra aprašoma formaliai. Tuo naudojantis nauji ryšiai tarp nesutampančių komponentų kuriami adaptuojant jų sąveikas.

2.2.4.3. Apdengiantis komponentas

Šiuo atveju komponentas B taip pat atlieka tarpininko vaidmenį. B komponentas apdengia komponentą A ir tiesiogiai bendrauja su komponentu C (6 pav). B tarpininkauja perimdamas ir konvertuodamas visus iš A išeinančius ir A skirtus (ateinančius) pranešimus [AS02].



6 pav. B komponentas apdengia komponentą A, perimdamas A pranešimus ir tiesiogiai komunikuojantis su C.

Komponento apdengimas gali pasitarnauti ir dar vienu būdu. B turi galimybę pateikti kitokias nei A teikiamas paslaugas. Tai yra, naudojantis A teikiamomis paslaugomis, B realizuoja naują funkcionalumą ir paskelbia savo paslaugas.

Apdengiantis komponentas gali būti naudojamas ne tik tam kad paviešinti dalį, arba visą apdengiamo komponento funkcionalumą, tačiau ir dalinai jį pakeisti, ar pritaikyti konkrečiam scenarijui.

2.3. Deklaratyvūs karkasai

Darbo metu buvo ištirti keli egzistuojantys deklaratyvaus vartotojo sąsajos apibrėžimo karkasai. Visos aprašytos bibliotekos/karkasai leidžia naudoti visus Swing komponentus – keisti atributus, nurodyti įvykių klausytojus ir t.t. Taip pat leidžia sinchronizuoti meniu ir mygtukų būsenas bei yra lokalizuojami.

Visos bibliotekos yra plečiamos, tai yra, galima pridėti naujų komponentų (juos atitinkančias žymas) arba praplėsti egzistuojančias žymas.

2.3.1. CookSwing[1]

CookSwing tai biblioteka, kurios pagalba iš XML dokumentų sukonstruojama Java Swing vartotojo sąsaja. Biblioteka palaiko visus AWT/Swing išdėstymo valdytojus (angl. Layout manager). Taip pat palaiko šablonus – galima įtraukti (angl. Include) kitus xml failus.

XML Dokumente galima apibrėžti ne tik tai komponentus. Yra ir pagalbinių žymų, kuriomis galima apibrėžti įvairias struktūras (sąrašus, rinkinius ir t.t).

```
<frame title="Langas" defaultcloseoperation="EXIT_ON_CLOSE">
    <button text="Mygtukas" actionlistener="buttonAction" />
</frame>
```

7 pav. CookSwing deklaratyvus lango su mygtuku apibrėžimas

Įvairių atributų nustatymui naudojamos supaprastintos tekstinės išraiškos, kurios į reikiamą formatą transformuojamos konverterių pagalba. Egzistuoja jau sukurtas konverterių rinkinys, kurį taip pat galima praplėsti.

Visi įvykių klausytojai apibrėžiami kaip vieši klasės atributai, ir nurodomi kaip 7 pav., o pats tos klasės objektas paduodamas karkasui ir naudojamas kaip lango modelis.

2.3.2. SDL/Swing[2]

Karkasas vartotojo sąsajos apibrėžimui naudoja SDL (angl. Simple Declarative Language). Komponentų apibrėžimai, bei reikšmių perdavimai supaprastinti ir padaryti kaip įmanoma

[1] http://cookxml.yuanheng.org/cookswing/
[2] http://www.ikayzo.org/confluence/display/SSW/Home

glaustesniais. Pavyzdžiui, užtenka parašyti simbolių eilutę, ir karkasas tai interpretuos kaip `JLabel` komponentą.

```
frame {
    form ID="Langas" {
        button "Mygtukas"
    }
}
```

8 pav. SDL/Swing deklaratyvus lango su mygtuku apibrėžimas

Viena išskirtinė šio karkaso funkcija yra komponentų stilizavimo panašiu į CSS (angl. Cascading Style Sheet) būdu palaikymas – galima nurodyti stilius, kurie apibrėžiami atskirame dokumente. Karkasas palaiko beveik visus Java Swing išdėstymo valdytojus.

2.3.3. Swing JavaBuilder[3]

Naudojama YAML kalba. Šis karkasas išskirtinis tuo, kad siūlo papildomą deklaratyviai apibrėžiamą funkcionalumą, tokį kaip:

- Komponentų reikšmių tikrinimas (angl. Validation)
- Foninių užduočių vykdymo palaikymas su atvaizdavimu sąsajoje (naudojamas SwingWorker)
- Integruotą reikšmių susiejimą (angl. value binding).

Karkaso konstruojami objektai apibrėžiami naudojant du failus:

- YAML failas. Deklaratyviai apibrėžiama vartotojo sąsaja. Į tai įeina, komponentų sukūrimas, jų savybių nustatymas ir metodų kviečiamų iš įvykių klausytojų apibrėžimas, taip pat komponentų išdėstymo apibrėžimas, duomenų susiejimo apibrėžimas, komponentų ar jų reikšmių tikrinimas.
- Java klasė (`JFrame`, `JPanel` ir t.t) kurioje yra visas programinis kodas nusakantis konstruojamą langą. Įeina kviečiamų metodų realizacija bei suvedamų duomenų laikymas.

Karkasas įgyvendina MVC (angl. Model View Controller) modelį, kur YAML dokumentas atlieka vaizdo, o Java klasė – kontrolerio vaidmenį.

```
JFrame(name=frame, title=Langas):
    - JButton(name=button, text=Mygtukas)
```

[3] http://code.google.com/p/javabuilders

9 pav. Swing JavaBuilder deklaratyvus lango su mygtuku apibrėžimas

Skirtingai nei anksčiau minėtuose karkasuose, čia palaikoma žymiai mažiau išdėstymo valdytojų: `CardLayout`, `MiGLayout`[4] ir `FlowLayout`.

[4] http://miglayout.com

3. Sistemos sandara

Sistema veikia operuodama dviejų rūšių esybėmis. Pirma tai komunikuojantys komponentai – esybės, kurios tarpusavyje bendrauja, o antra – jungtys, apjungiantys komunikuojančius komponentus, formuojantys jų tarpusavio ryšius. Per šiuos tarpusavio sąryšius yra siuntinėjami pranešimai – bendravimas tarp komponentų vyksta pranešimų (3.1.4) pagalba.

Išskiriami trys sistemos veikimo etapai:

1) Komunikuojančių komponentų sukūrimas
2) sąryšių tarp jų sudarymas
3) lango su tais komponentais atvaizdavimas.

3.1. Komunikuojantys komponentai

Komunikuojantys komponentai yra Java Swing komponentai, turintys papildomą funkcionalumą, bendrą visiems komunikuojantiems komponentams. Tai yra galimybė būti apjungtiems tarpusavyje, su kitais komunikuojančiais komponentais bei, kaip jau minėta, keistis su jais pranešimais.

Kiekvienas komunikuojantis komponentas turi:

- prievadus, apibrėžiančius to komponento įvestį ir išvestį
- specifinius įvykius
- mechanizmą leidžiantį generuoti įvykių klausytojus
- komponento komunikavimo modelį

Visa tai yra paviešinta `CommunicatingComponent` sąsajos pagalba, todėl yra paprastai konfigūruojama.

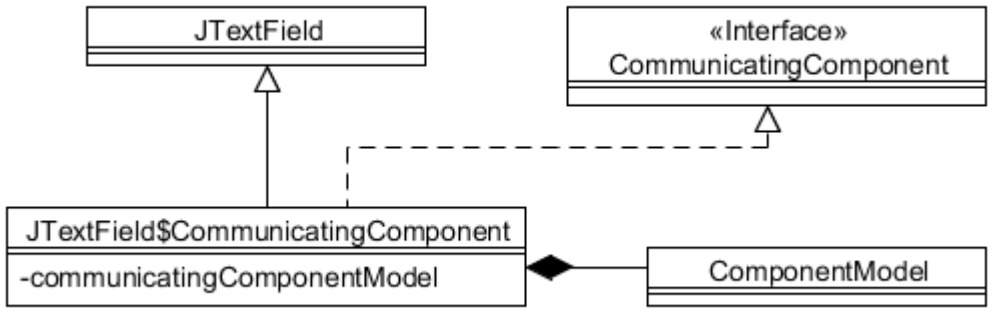
3.1.1. Sandara ir realizacija

Kiekvienas komunikuojantis komponentas turi komponento komunikavimo modelį. Ši dalis yra atsakinga už visas komunikuojančių komponentų funkcijas. Modelio paskirtis yra komponento inicijavimas – paruošimas darbui, įvykių klausytojų registravimas, įeinančių ir išeinančių pranešimų valdymas, įvykių, kurie yra specifiniai komunikuojantiems komponentams, generavimas.

Buvo keletas būdų kaip realizuoti komunikuojančius komponentus, tai yra, Java Swing komponentams suteikti galimybę komunikuoti tarpusavyje. Vienas iš jų buvo originalių Java Swing komponentų apdengimas, kaip tai buvo padaryta C2 architektūriniame stiliuje, aprašančiame komunikuojančius komponentus [TMA+96]. Tačiau šis metodas turi vieną minusą – atsiranda papildomas sluoksnis, kuris apsunkina priėjimą prie originalaus komponento.

Kitas būdas, kuris ir buvo pasirinktas, yra praplėsti komponentą (angl. Extend) (10 pav). Priešingai nei pirmu atveju, būtų išlaikomas originalaus komponento tipas – tokiu komponentu eitų manipuliuoti kaip ir originaliu Java Swing komponentu.

Tai įgalintų lengvesnę integraciją su kitais karkasais, kurie skirti palengvinti darbui su Java Swing komponentais bei vartotojo sąsajomis (pavyzdžiui karkasai leidžiantis vartotojo sąsają apibrėžti deklaratyviai).



10 pav. `JTextField` komponentas praplėstas realizuojant `CommunicatingComponent` sąsają.

Nors pastarasis metodas pranašesnis prieš pirmąjį, tačiau iškyla komponentų praplėtimo problema. Komponentų yra daug ir visokių, taip pat jų bet kada gali padaugėti, kadangi naudotojas gali sukurti naujus savo komponentus ir norėti juos panaudoti.

Apsiriboti praplečiant tik tam tikrus komponentus yra tikrai ne išeitis. Taip pat naudotojui neturėtų būti paliekama atsakomybė praplėsti savo sukurtus komponentus – toks procesas turėtų būti automatizuotas ir reikalaujantis kuo mažesnio naudotojo įsikišimo. Todėl reikalingas metodas leidžiantis tai atlikti dinamiškai, programos darbo metu. Sekančiame poskyryje detaliau apie tai, kaip komponentai yra dinamiškai praplečiami.

3.1.2. Dinaminis komponentų praplėtimas

10 pav. pavaizduotas komponentų praplėtimo principas gali būti pritaikomas bet kokiems komponentams (o tiksliau – objektams) bei yra atliekamas programos vykdymo metu. Tai yra, atsiradus poreikiui naudoti konkretų Java Swing komponentą, kaip komunikuojantį komponentą, nereikia kurti naujos klasės kuri realizuotų `CommunicatingComponent` sąsają. Naują komunikuojančio komponento klasę galima sukurti pagal pareikalavimą.

Komunikuojančių komponentų kūrimas vyksta keliais etapais.

1. Generuojama komponento išvestinė klasė, jei tokia dar nėra sukurta
2. Sugeneruota klasė užkraunama ir naudojama – kuriami jos objektai

3.1.2.1. Komunikuojančio komponento klasės generavimas

Klasių generavimui naudojamas ASM[5] karkasas. Jo pagalba yra generuojamas baitų masyvas, atitinkantis Java klasės failo formatą. Tokiu pačiu formatu Java klasės yra laikomos diske bei užkraunamos į JVM (angl. Java Virtual Machine). Šis karkasas suteikia galimybę aukštesniu lygiu rašyti Java byte kodą.

```
public void addMessageReceivedListener(MessageReceivedListener listener) {
  communicatingComponentModel.addMessageReceivedListener(listener);
}
```

```
ClassWriter cw = new ClassWriter(0);
...
MethodVisitor mv = cw.visitMethod(ACC_PUBLIC, "addMessageReceivedListener",
  "(Lcore/event/MessageReceivedListener;)V", null, null);
mv.visitCode();
mv.visitVarInsn(ALOAD, 0);
mv.visitFieldInsn(GETFIELD, "core/TestKlass",
  "communicatingComponentModel", "Lcore/component/CommunicatingComponentModel;");
mv.visitVarInsn(ALOAD, 1);
mv.visitMethodInsn(INVOKEVIRTUAL, "core/component/CommunicatingComponentModel",
  "addMessageReceivedListener", "(Lcore/event/MessageReceivedListener;)V");
mv.visitInsn(RETURN);
mv.visitMaxs(2, 2);
mv.visitEnd();
```

11 pav. Viršuje vienas iš komunikuojančio komponento metodų. Apačioje ekvivalentus tą metodą generuojantis ASM kodas.

ASM karkasas turi galimybę iš klasės failo sugeneruoti ekvivalentų ASM kodą, kuriuo ta klasė apsirašo. Todėl sistemos kūrimo metu buvo sukurta viena konkreti komunikuojančio komponento klasė (10 pav) ir išgautas ASM kodas tai klasei generuoti. Šis kodas vėliau buvo panaudotas apibrėžiant komunikuojančių komponentų klasėms sukurti reikalingus kodo generatorius (11 pav).

Beveik visi generuojami metodai deleguoja kvietimus komunikuojančio komponento modeliui (`CommunicatingComponentModel` objektui) bei kviečia „super“ metodus. Todėl generuojamas kodas (instrukcijos) yra palyginti paprastas, o visa veikimo logika lieka apibrėžta komunikuojančio komponento modelyje – todėl ir lengvai prižiūrima.

3.1.2.2. Sugeneruotos klasės naudojimas

Sugeneravus klasę aprašantį baitų masyvą, jis yra konvertuojamas į `Class` objektą. Tai atliekama naudojant `ClassLoader.defineClass()`. Ši klasė tampa pasiekiama ir pradedami kurti jos egzemplioriai.

[5] http://asm.ow2.org/

3.1.3. Prievadai

Kaip jau minėta, komunikuojantys komponentai turi įvestį ir išvestį. Paprasčiau tariant – tai kintamieji, kuriuos komponentas gauna iš išorės arba nustato pats. Kiekvienas prievadas atitinka komunikuojančio komponento kintamąjį – kintamieji identifikuojami vardu, kuris tuo pačiu yra ir prievado vardas, bei yra tipizuoti – prievadas nesugebės priimti nepalaikomo tipo kintamojo (detaliau apie suderinamumą 3.1.3.3).

Tam kad kintamieji ar jų duomenys galėtų būti perduodami tarp komunikuojančių komponentų, jie privalo būti tarpusavyje sujungti. Prievadai tarnauja kaip komunikuojančių komponentų sujungimo taškai – komponentams sujungti yra sujungiami atitinkami jų prievadai.

Bendravimas tarp sujungtų komponentų vyksta pranešimais ir tik per prievadus. Komponentas siunčiantis pranešimą nežino kas bus pranešimo gavėjas – jo gali ir nebūti. Tai reiškia, kad komponento žinutė gali ir niekur nenueiti, todėl komunikuojantis komponentas neturi daryti prielaidų apie prijungtų komponentų egzistavimą, nebent prievadas pažymėtas kaip būtinas.

Bendru atveju, panašiai kaip ir šaltiniuose [FDH08, TMA+96] prievadai pagal savo atliekamą funkciją yra skirstomi į dvi rūšis:

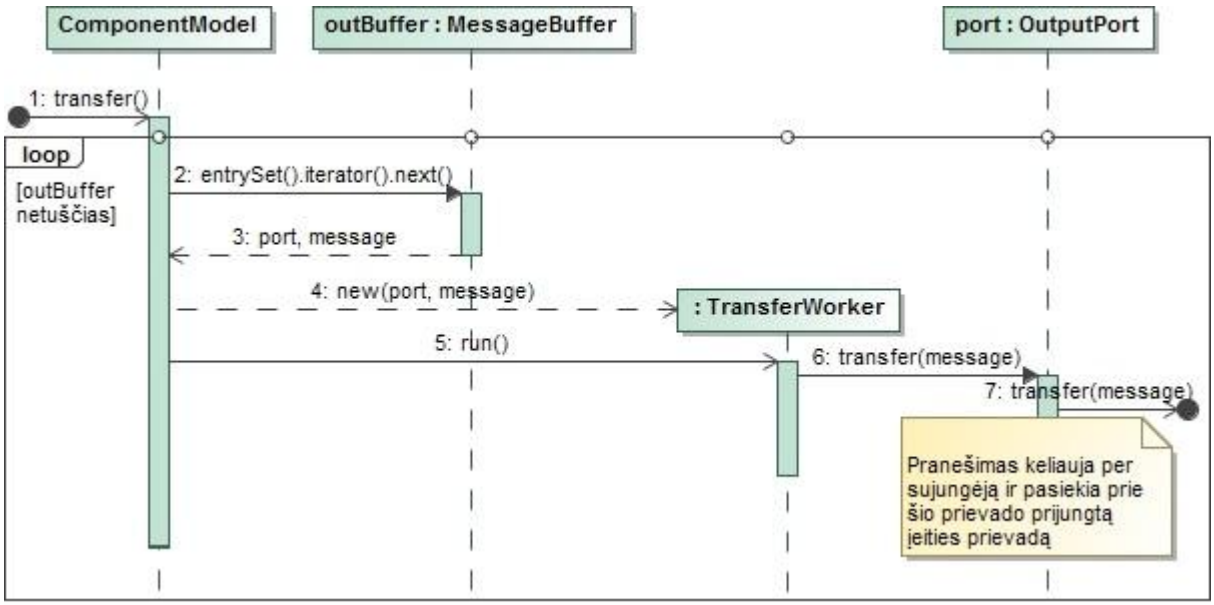
- **Siunčiantys** – siunčia pranešimus, o dvikrypčių prievadų atveju ir priima atsakymus.
- **Priimantys** – priima pranešimus, o dvikrypčių prievadų atveju siunčia atgal atsakymus.

Pagal bendravimo mechanizmą prievadai yra išskiriami į dvi grupes – vienakrypčius ir dvikrypčius. Toliau detaliau aprašomas kiekvienas iš jų.

3.1.3.1. Vienakrypčiai prievadai

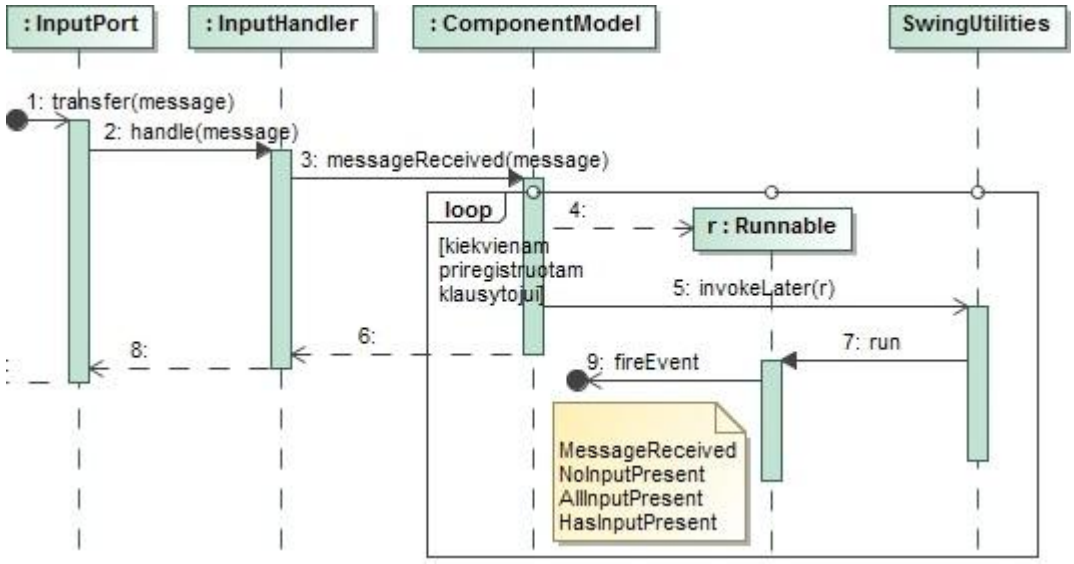
Šio tipo prievadai naudojami asinchroniniams pranešimams perduoti. Pranešimų siuntimas ir gavimas naudojantis šiais prievadais vyksta buferių pagalba.

Prieš siunčiami pranešimai padedami į siunčiamų pranešimų buferį, kadangi pranešimų išsiuntimas gali būti uždelstas (detaliau apie tai 3.4.5.1).



12 pav. Asinchroninių pranešimų siuntimas per išvesties prievadą.

Siuntimo momentu (12 pav) visi išvesties buferyje (`outBuffer`) esantys pranešimai išsiunčiami asinchroniškai – sukuriama nauja darbinė gija, kuri inicijuoja pranešimo siuntimą per nurodytą išvesties prievadą. Gijos užduotis yra pernešti pranešimą gavėjui.



13 pav. Asinchroninio pranešimo priėmimas per įvesties prievadą

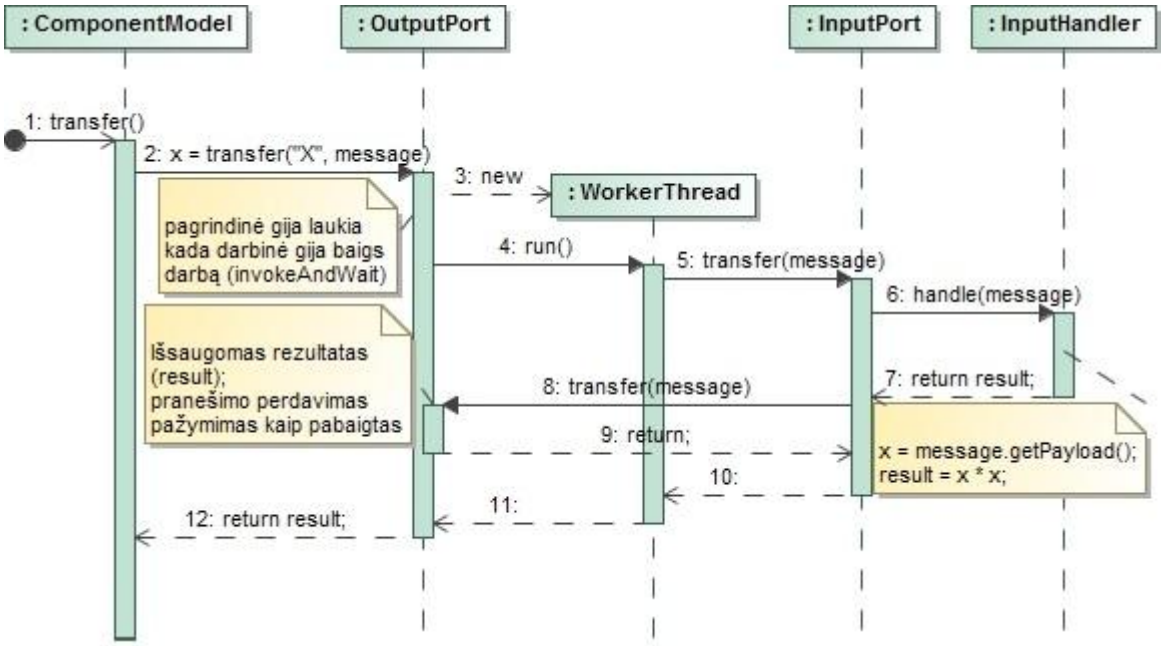
Toliau pranešimas keliauja per jungtį ir pasiekia gavėjo įvesties prievadą. Ten jį perima gavėjo komunikuojančio komponento modelis (13 pav). Gautas pranešimas padedamas į įvesties buferį. Toliau komunikavimo modelis sprendžia kokius įvykius sugeneruoti. Atitinkami įvykių klausytojų metodai iškviečiami jau iš kitos gijos – `EventQueue`. Darbinė gija sukurta 12 pav. baigia darbą.

Detaliau apie klausytojus 3.4.1 poskyryje.

3.1.3.2. Dvikrypčiai prievadai

Be vienakrypčių egzistuoja ir dvikrypčiai prievadai, jie skirti sinchroninei komunikacijai. Prieš pridedant naują prievado tipą buvo svarstytas ir kitas variantas: komponentai galėtų turėti papildomus prievadus, kuriuose jie lauktų grįžtančio rezultato. Tačiau tokiu atveju komponentų konfigūracija tampa sudėtingesnė, atsiranda perteklinių jungčių. Taip pat iškyla ir dar viena problema, tokiu atveju sinchroninį kvietimą inicijuojančiam objektui tektų pačiam rūpintis rezultato laukimu – užsiblokavimu iki gaunamas rezultatas.

Dvikrypčiai prievadai gali arba priimti pranešimą ir į jį atsakyti (išsiųsti atsakymą), arba atvirkščiai – išsiųsti pranešimą ir priimti atsakymą. Kaip ir vienakrypčių prievadų atveju, egzistuoja du vaidmenys – siunčiantis (pirmiau išsiunčia, paskui priima) ir priimantis (atvirkščiai siunčiančiam) prievadai.



14 pav. Dvikrypčių prievadų apsikeitimas pranešimais. Supaprastinta diagrama – išvesties ir įvesties prievadai pavaizduoti sujungti tiesiogiai – neįtraukta jungtis.

Priešingai nei vienakrypčių prievadų atveju, čia buferiai nei gautiems, nei siunčiamiems pranešimams laikyti nenaudojami. Pranešimai keliauja tiesiai gavėjui, o gautas rezultatas iš karto grąžinamas. Taip pat negeneruojami ir asinchroniniams pranešimams įprasti įvykiai.

Pradėjus siuntimą (14 pav.) ir pranešimui patekus į išvesties prievadą, sukuriama gija, kurios paskirtis bus atlikti pranešimo pernešimą iš siuntėjo išvesties prievado iki gavėjo įvesties prievado ir gavus apskaičiuotą rezultatą jį pargabenti atgal siuntėjui į išvesties prievadą.

Tuo tarpu pagrindinė gija, sukūrusi darbinę giją, laukia jos darbo pabaigos. Pasibaigus darbinės gijos darbui, pagrindinė gija atsiblokuoja, ir grąžina pargabentą reikšmę – rezultatą.

3.1.3.3. Prievadų suderinamumas

Komponentų sujungimas įmanomas tik tada kai prievadai yra suderinami. Priešingu atveju komponentų neis sujungti. Žemiau pateikti pagrindiniai punktai, kurie turi būti patenkinti, tam kad eitų sujungti du prievadus.

- **Prievadų grupė.** Vienkrypčiai ir dvikrypčiai prievadai nėra suderinami, kadangi jų komunikacijos principai skiriasi iš esmės.
- **Prievadų rūšis.** Suderinamais laikomi tik skirtingų rūšių prievadai – siunčiantis prievadas gali būti sujungiamas tik su priimančiu.
- **Pranešimų tipai.** Suderinamais laikomi prievadai, kurių tipai sutampa arba jei siunčiančio prievado tipas yra priimančio prievado potipis. Pvz.: siunčiantis prievadas yra `java.lang.String` tipo, o priimantis `java.lang.Object`. Taip pat egzistuoja galimybė sujungti du nesuderinamus prievadus, jei juos jungianti jungtis moka konvertuoti pranešimus (žr. 3.2.3.2).

3.1.3.4. Prievadų deklaratyvus apibrėžimas

Prievadas turi vardą yra tipizuotas bei turi rūšį ir kryptį. Prievado kryptį bei rūšį nusako konkretus prievado klasės įgyvendinimas. Egzistuoja 4 tokios klasės: po dvi vienakrypčiams ir dvikrypčiams prievadams – atitinkamai toms klasėms yra užregistruoti tokie Spring bean: `outputPort`, `inputPort`, `biOutputPort`, `biInputPort`. 15 pav. yra naudojamas `outputPort` – vienakryptis išvesties prievadas.

```
<bean parent="outputPort">
  <property name="payloadType" value="java.lang.String" />
  <property name="name" value="portName" />
</bean>
```

15 pav. Prievadų sąrašo deklaratyvus apibrėžimas

Toliau lieka apibrėžti tik:

- **name** - prievado vardą, kuris bus pasiekiamas iš kodo bei įvykių klausytojų konfigūracijos (3.4.1).
- **payloadType** – prievado priimamų arba siunčiamų pranešimų tipas (gali būti ir tuščias).

3.1.4. Pranešimai

Pranešimai naudojami komunikuojantiems komponentams tarpusavyje apsikeisti informacija. Pranešimo turinys gali būti bet koks objektas. Tarp komponentų pranešimai keliauja jungčių (3.2) pagalba.

Kaip jau buvo užsiminta 3.1.3 poskyryje, pranešimai yra dviejų tipų: asinchroniniai ir sinchroniniai.

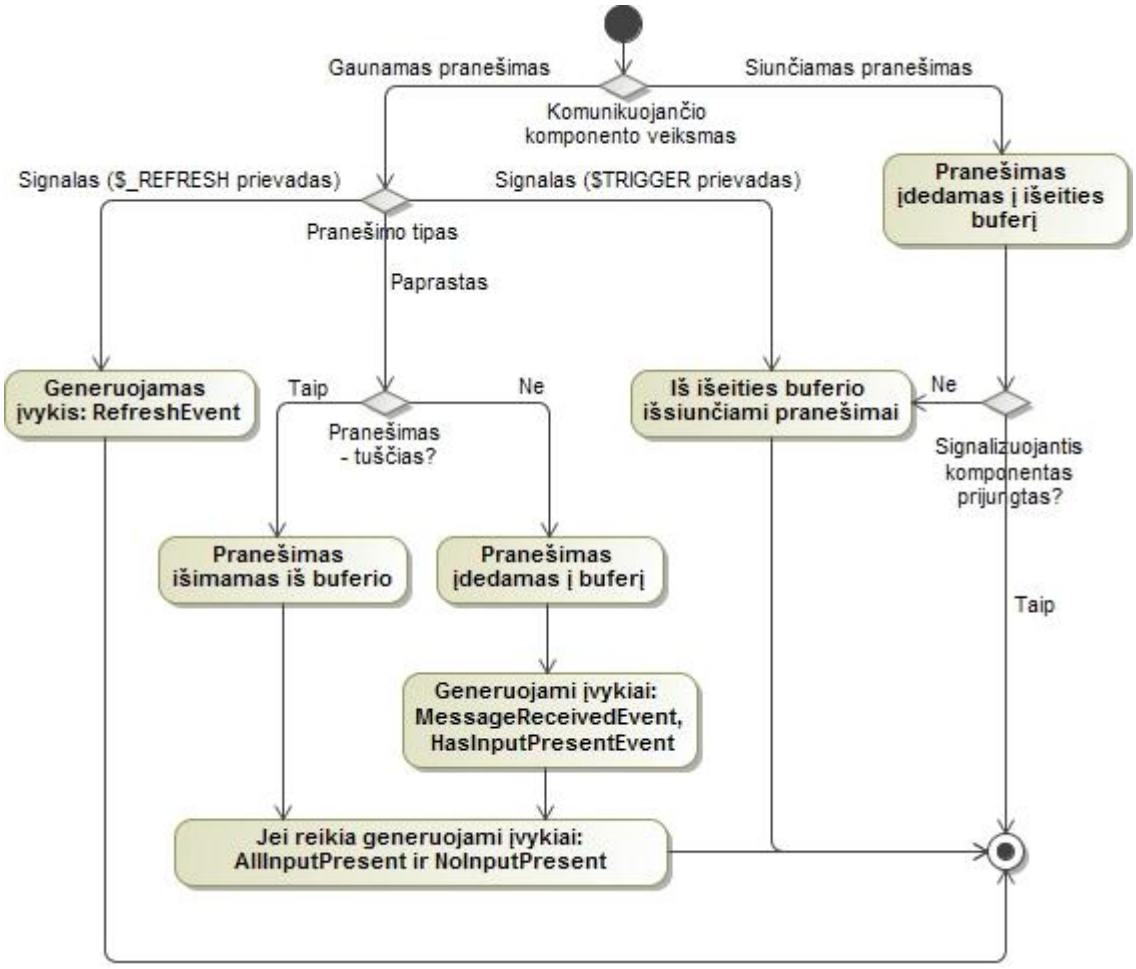
3.1.4.1. Asinchroniniai pranešimai

Asinchroniniai pranešimai yra skirti pranešti kitiems komunikuojantiems komponentams apie siuntėjo būsenos pasikeitimus (įvestas tekstas, paspaustas mygtukas), nelaukiant atsako iš gavėjo.

Asinchroninių pranešimų priėmimo metu yra generuojama aibė komunikuojančių komponentų įvykių, kurių kiekvienas detaliau aprašytas 3.1.5 – į gautus pranešimus reaguojama įvykių klausytojų pagalba.

Pranešimai nebūtinai turi nešti turinį. Gavus tuščią pranešimą, komunikuojančio komponento modelis pažymi pranešimą priėmusį prievadą (kintamąjį) kaip neturintį jokios reikšmės.

Signalas – tuščias pranešimas išsiųstas iš prievado be jokio tipo. Egzistuoja du specialūs prievadai, kurie priima tik signalus – `$_TRIGGER` ir `$_REFRESH`. Detaliau apie jų paskirtį aprašyta 3.4.5.



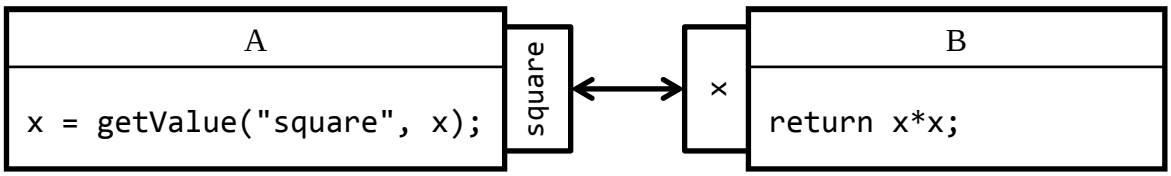
16 pav. Asinchroninių pranešimų priėmimas ir siuntimas

16 pav. pavaizduotas pranešimų priėmimams ir išsiuntimas. Išskirti pranešimai ateinantys į `$_TRIGGER` ir `$_REFRESH` prievadus bei parodytas jų ryšys su komponento pranešimų valdymu bei kada ir kokie įvykiai yra generuojami.

3.1.4.2. Sinchroniniai pranešimai

Sinchroniniai pranešimai yra siunčiami per dvikrypčius prievadus (3.1.3.2). Šių pranešimų siuntimas bei apdorojimas atliekamas tik tam tikslui naudotojo pateiktame Java kode – nėra galimybės jų siuntimą inicijuoti taip kaip tai galima daryti su asinchroniniais pranešimais ir `EventListenerInfo` (3.4.1.2). Taip yra todėl, kad sinchroninių pranešimų poreikis gali atsirasti tik kai prireikia atlikti skaičiavimus ar panašias operacijas, kurių rezultatą reikia iškart gauti atgal.

Pavyzdžiui kiekvieną kartą paspaudžiant mygtuką, komponentas A atlieka skaičiavimus, į kuriuos įeina skaičiaus kėlimas kvadratu, ir tas rezultatas atvaizduojamas komponente (iliustruota 17 pav.).



17 pav. Sinchroninis komponentų komunikavimas. A komponentas – siunčiantis, o B – priimantis (žr. prievadų rūšis 3.1.3)

Tarkime, kad skaičiaus kvadratą apskaičiuoja komponentas B. Komponentas A priklausys nuo B. Mygtuko paspaudimo momentu A turės netik kad perduoti skaičių x komponentui B, tačiau ir sulaukti iš jo atsakymo, netęsiant skaičiavimų – reikalingas sinchroninis pranešimo perdavimas (išsamiau apie dvikrypčių prievadų veikimo principą 3.1.3.2).

3.1.5. Komunikuojančių komponentų įvykiai

Be originalių Java Swing komponentų įvykių, komunikuojantys komponentai komponentų komunikacijos arba sujungimo metu gali generuoti ir specifinius įvykius.

Pranešimų gavimo metu yra generuojami kelių rūšių įvykiai, priklausomai nuo įvesties prievadų bei įvykių klausytojų konfigūracijos (bendra informacija apie įvykių klausytojų registravimą bei generuojamų klausytojų galimybes – 3.4.1).

Pranešimo gavimo metu generuojami įvykiai:

1) `MessageReceivedEvent` – gaunamas pranešimas
2) `HasInputPresentEvent` – užpildytas bent vienas klausytojo reikalaujamas kintamasis

3) `AllInputPresentEvent` – užpildyti visi klausytojo reikalaujami kintamieji
4) `NoInputPresentEvent` – neužpildytas nei vienas klausytojo reikalaujamas kintamasis
5) `RefreshEvent` – į `$_REFRESH` (prievado paskirtis aprašyta 3.4.5.2) prievadą atėjo pranešimas.

Komponentų sujungimo metu generuojamas įvykis:

6) `ConnectedToPortEvent` – prie prievado prijungtas kito komunikuojančio komponento prievadas.

3.1.6. Komponentų deklaratyvus apibrėžimas

Komponentų apibrėžimas yra vykdomas Spring[6] karkaso pagalba. Kiekvieną komponentą aprašo vienas Spring bean. 18 pav. pavaizduotas mygtuko – `javax.swing.JButton` komunikuojančio komponento apibrėžimas su vienu išvesties prievadu ir `ActionListener` klausytoju.

```
<bean id="jButton" parent="abstractComponent">                          1
  <constructor-arg value="javax.swing.JButton" />
  <property name="name" value="mygtukas" />
  <property name="connectionPorts">                                     2
    <map>
      <entry key="actionPerformed" value-ref="outputTrigger" />
    </map>
  </property>
  <property name="eventListenerInfoList">                               3
    <list>
      <bean id="actionListener" parent="eventListenerInfo">
        <property name="listenerName" value="action" />
        <property name="targetPortName" value="actionPerformed" />
      </bean>
    </list>
  </property>
```

18 pav. Komunikuojančio komponento apibrėžimas Spring karkaso pagalba.

Komunikuojančio komponento apibrėžimas susideda iš 3 dalių:

1) Komunikuojančio komponento tipo (šiuo atveju `javax.swing.JButton`) ir įvairių atributų apibrėžimas. Šioje dalyje galima nustatyti komponento pradinę reikšmę, išvaizdą ir t.t. – sukurti norimą komponento būseną.
2) Prievadų sąrašo apibrėžimas. Nusako per kokius taškus komunikuojantis komponentas sąveikaus su išore. Daugiau apie prievadus ir jų apibrėžimą - 3.1.3.
3) Įvykių klausytojo sąrašo apibrėžimas. Nusako komponento reakciją į vidinius bei išorinius įvykius. Daugiau apie įvykių klausytojų apibrėžimą - 3.4.1.

[6] http://www.springsource.org/

3.2. Jungtys

Jungčių paskirtis yra apjungti komunikuojančias esybes, tam kad galėtų vykti jų tarpusavio komunikacija – kintamųjų reikšmių apsikeitimas. Taip pat, jungtys atlieka ir dar vieną svarbią funkciją – apibrėžia komponentų tarpusavio sąryšius (priklausomybes). Jų pagalba įmanoma nesunkiai nusakyti, kurių komunikuojančių komponentų pasikeitimai gali įtakoti kitus komunikuojančius komponentus.

Be to, kadangi komunikuojančių komponentų tarpusavio sąryšiai yra atskirti nuo komponentų veikimo logikos, jungčių pagalba įmanoma nesunkiai keisti komunikuojančių komponentų tarpusavio priklausomybes.

3.2.1. Jungčių funkcijos

Be pranešimų perdavimo jungtys turi ir kitas funkcijas, tokias kaip:

- Pranešimų filtravimas
- Komponentų adaptavimas – pranešimų keliaujančių tarp nesuderinamų prievadų konvertavimas.
- Pranešimų užlaikymas – galimybė nesiųsti pranešimo tol kol nėra gaunamas atitinkamas signalas.

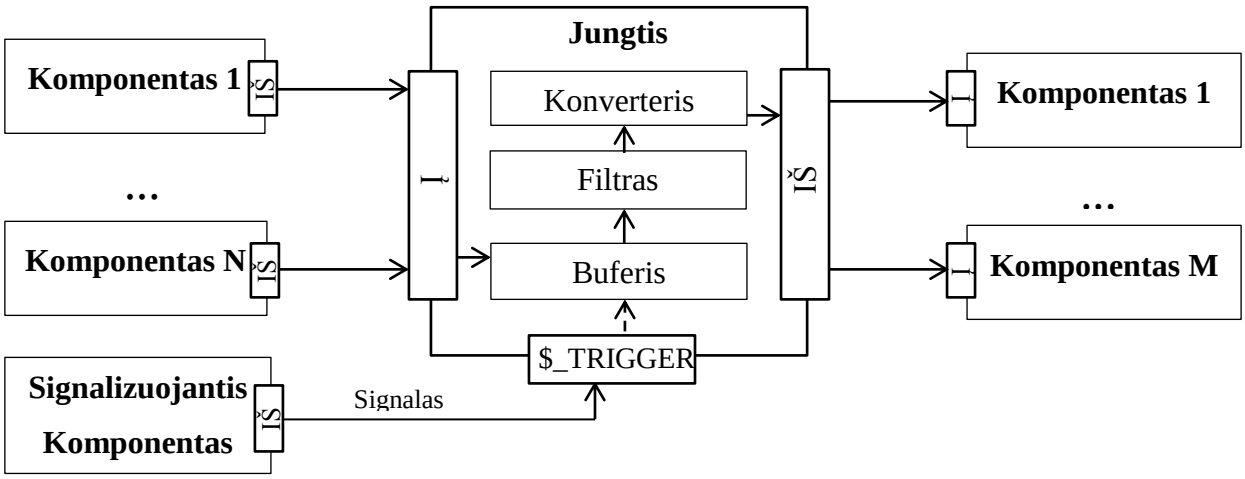
3.2.2. Jungties schema

Jungtis sujungia 2 arba daugiau komponentų, kurie jungiasi prie jungties prievadų. Iš viso jungtis turi tris prievadus (19 pav):

1) įvesties
2) išvesties
3) signalizavimo

Jungties įvesties ir išvesties prievadai yra sukuriami dinamiškai – pagal prie jų jungiamus kitos esybės prievadus. Sukurti prievadai visada yra suderinami su prie jų prijungtais prievadais, todėl jų nereikia apibrėžti atskirai.

Visi prievadai prijungti prie jungties įeinančio arba išeinančio prievado privalo būti vienodo tipo.



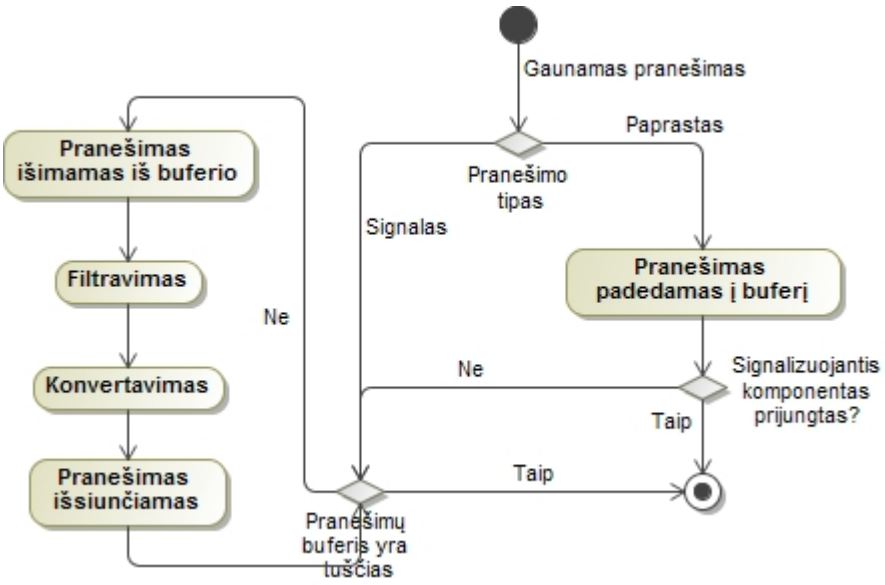
19 pav. Jungties schema

Jungties įvestis jungiasi prie komponento išvesties ir atvirkščiai. Tokių jungčių skaičius yra neribojamas – galima prijungti daug komponentų. Kiekvienas komponentas prijungtas prie jungties įvesties išsiuntęs pranešimą perduos visiems komponentams prijungtiems prie jungties išvesties (20 pav).

20 pav. 3 komponentai prijungti prie jungties įvesties, o 2 prie išvesties. Vieno komponento išsiųstas pranešimas pasiekia abu komponentus-gavėjus.

3.2.3. Veikimo principas

Normaliu atveju jungtis pranešimą perduos nedelsiant – vos jam atėjus. Tačiau pranešimus galima užlaikyti – perduoti ne iš karto. Tam skirtas signalizavimo prievadas (kaip ir komunikuojantys komponentai, pagal nutylėjimą jį turi visos jungtys). Jei prie šio prievado yra prijungtas bent vienas komponentas, pranešimai bus perduodami tik tada, kai jungtis gaus signalą (21 pav).



21 pav. Jungties veikimo schema – pranešimų perdavimas

Pranešimo pernešimas vyksta keliais etapais, detali schema pavaizduota (21 pav). Pranešimai ateina iš komponentų prievadų. Priklausomai nuo gautų pranešimų rūšies jungtis elgiasi skirtingai:

- **Paprastas pranešimas** – pranešimas padedamas į buferį, ir jei signalizuojantis prievadas neturi prijungto komponento, pranešimas – išsiunčiamas.
- **Signalas** – jungtis išsiunčia buferyje sukauptus pranešimus.

Pranešimo persiuntimas vyksta keliais etapais:

1) Pranešimas išimamas iš buferio
2) Pranešimų filtravimas (3.2.3.1)
3) Pranešimų konvertavimas (3.2.3.2)

Tik atlikus visus šiuos žingsnius pranešimas yra persiunčiamas gavėjui prijungtam prie jungties išvesties prievado.

3.2.3.1. Filtravimas

Pradėjus pranešimų persiuntimą pirmiausia vykdomas pranešimų išfiltravimas. Esant poreikiui tam tikrus pranešimus (filtruojama pagal pernešamą turinį) jungčiai galima neleisti perduoti. Pavyzdžiui neleisti perduoti skaičių, kurie mažesni už X. Pagal nutylėjimą joks filtravimas nevykdomas.

3.2.3.2. Konvertavimas

Kartais gali atsirasti poreikis sujungti du komunikuojančius komponentus, kurių prievadai yra nesuderinami (žr. 3.1.3.3) – pavyzdžiui vienas siunčia skaičių, o kitas priima tekstą. Tokiu atveju, jungčiai yra būtinas metodas, kurio pagalba pranešimai jų perdavimo metu būtų konvertuojami į reikiamą formatą ar tipą.

Jei bet kokio tipo prievadas yra sujungtas su `java.lang.String` tipą priimančiu prievadu, tada pagal nutylėjimą pranešimo turinys konvertuojamas yra konvertuojamas `.toString()` metodo pagalba.

3.2.4. Jungties deklaratyvus apibrėžimas

Kaip ir komponentai, jungtys apibrėžiamos Spring karkaso pagalba. Jungties aprašą sudaro trys dalys:

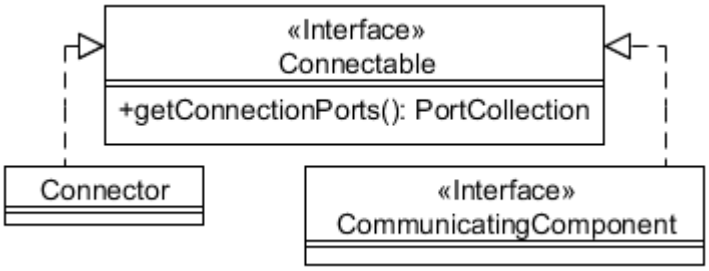
1) **Įvesties ir išvesties prievadai** – perduodami kaip konstruktoriaus argumentai. Konkretus prievadas identifikuojamas nurodant esybę kuriai jis priklauso ir to prievado vardą. Galimi du būdai nurodyti sujungiamiems prievadams (iliustruota 22 pav.). Sujungiami prievadai nurodomi dviem skirtingais būdais: išraiška `esybė:prievadas`, arba kaip esybių ir prievadų vardų poros.
2) **converter** – objektas atsakingas už pranešimų konvertavimą tarp nesuderinamų prievadų (3.2.3.2).
3) **filter** – pranešimų keliaujančių per jungtį filtravimas (3.2.3.1).

```
<bean id="connectorId" parent="connector">
  <constructor-arg value="component1:outgoingPort" />
  <constructor-arg value="component2:incomingport" />
  <property name="converter" ref="defaultConverter" />
  <property name="filter" ref="defaultFilter" />
</bean>
```

```
<bean id="connectorId" parent="connector">
  <constructor-arg ref="component1" />
  <constructor-arg value="outgoingPort " />
  <constructor-arg ref="component2" />
  <constructor-arg value="incomingport " />
  <property name="converter" ref="defaultConverter" />
  <property name="filter" ref="defaultFilter" />
</bean>
```

22 pav. Apibrėžta jungtis, sujungianti „component1“ ir „component2“, per jų „outgoingPort“ ir „incomingPort“ prievadus.

Komunikuojantys komponentai ir jungtys yra esybės galinčios būti sujungtos tarpusavyje – realizuoja `Connectable` sąsają (23 pav). Dėl šios priežasties, sujungimai gali nebūtinai būti vien tik tarp komunikuojančių komponentų (pavyzdžiui, mygtukas gali būti sujungiamas su jungtimi – paspaudimo metu jungtis praleidžia susikaupusį pranešimų srautą).



23 pav. Jungtys ir komunikuojantys komponentai yra esybės kurios gali būti sujungtos tarpusavyje.

3.3. Lango atvaizdavimas

Komunikuojančių komponentų atvaizdavimas vyksta atskirai nuo jų kūrimo bei konfigūravimo, todėl dėl šios priežasties galima šią užduotį atlikti grynai su Java kalba, arba palikti jau egzistuojantiems įrankiams.

Lango apibrėžimui buvo nutarta rinktis jau egzistuojantį įrankį. Renkantis įrankį buvo atsižvelgiama į keletą savybių:

- Deklaratyvus vartotojo sąsajos aprašas
- Nesunkus konfigūravimas – galimybė programos veikimo metu užregistruoti vartotojo sukurtus komponentus.
- Išsami dokumentacija

Rezultate buvo pasirinktas Swing JavaBuilder (detaliau aprašytas 2.3.3) karkasas. Šis karkasas leidžia vartotojo sąsają apibrėžti deklaratyviai – tam naudojama YAML (angl. YAML Ain't Markup Language).

Palaikoma iš tiesų nemažai įvairių funkcijų, tačiau šio darbo kontekste svarbiausios yra: galimybė komponentus kurti bei išdėstyti lange deklaratyviai ir komponentų savybių nustatymas.

3.3.1. Lango sukūrimas

Kuriama sistema priima lango aprašą, apibrėžtus komunikuojančius komponentus bei lokalizuotų resursų sąrašą ir grąžina sukonstruotą lango objektą. Lango sukūrimas ir atvaizdavimas įvykdomas užrašius vos keletą eilučių (24 pav).

```
SwingJavaBuilderLauncher launcher = new SwingJavaBuilderLauncher();

launcher.initialize("classpath:MainWindow.yml").setVisible(true);
```

24 pav. Programos naudojančios `MainWindow.yml` kaip lango apibrėžimą paleidimas

Šiuo atveju lango objektas bus sukurtas pagal nutylėjimą, o jei tas objektas netinka, yra galimybė paduoti norimą objektą.

3.3.2. Komponentų naudojimas lango apraše

Svarbiausia bei didžiausia šio karkaso integravimo problema buvo komunikuojančių komponentų registravimas, kaip vartotojo apibrėžtų komponentų. Tai būtina tam, kad būtų galima juos įterpti į vartotojo sąsajos aprašą.

```
<bean id="button" parent="jButton">
```

25 pav. Mygtuko apibrėžimas

Komunikuojantys komponentai yra apibrėžiami Spring karkaso pagalba ir identifikuojami `id` atributu. Tam, kad būtų galima naudoti 25 pav. aprašytą komunikuojantį komponentą 26 pav. pavaizduotu būdu, teko parūpinti specializuotą komponentų kūrimo metodą Swing JavaBuilder karkasui.

```
JFrame(name=frame, title=Langas):
    - button(name=button, text=Mygtukas)
```

26 pav. `button` komunikuojantis komponentas (25 pav) naudojamas YAML faile.

Swing JavaBuilder leidžia užregistruoti naujus komponentų tipus, ir sukurti jiems sinonimus. Tačiau karkasas pats kuria naujus komponentų egzempliorius, pagal jų nurodytus klasių vardus. Todėl reikėjo pakeisti objektų kūrimo logiką.

Kiekvieną kartą karkasui aptikus naudojamą klasės vardo sinonimą (26 pav. tai bus `JFrame` ir `button`) pagal jį bandys išgauti klasę ir sukurti jos objektą. Šią rutiną teko papildyti taip, kad visų pirma būtų patikrinama, ar ieškomas klasės vardo sinonimas iš tikro nėra jau sukonstruoto ir Spring aplikacijos kontekste užregistruoto komunikuojančio komponento id (25 pav. tai bus `button`). Jei taip, grąžinamas tas komunikuojantis komponentas, priešingu atveju karkasas klasės vardo sinonimą apdoroja kaip ir anksčiau.

3.4. Galimybės

3.4.1. Įvykių klausytojai

Kaip jau minėta, be tipinių Java Swing komponentų įvykių, komunikuojantys komponentai taip pat generuoja ir specifinius įvykius. Visiems šitiems įvykiams apdoroti galima užregistruoti klausytojus.

3.4.1.1. Įvykių klausytojų registravimas

Klausytojų registracija vyksta komponento inicializavimo metu. Norint užregistruoti klausytoją, jis aprašomas pagalbine struktūra – `EventListenerInfo` (27 pav), kurioje ne tik nusirodo koks konkretus klausytojas yra registruojamas konkrečiam įvykiui, tačiau ir kiti parametrai naudojami nusprendžiant, kokie įvykiai domina šį klausytoją.

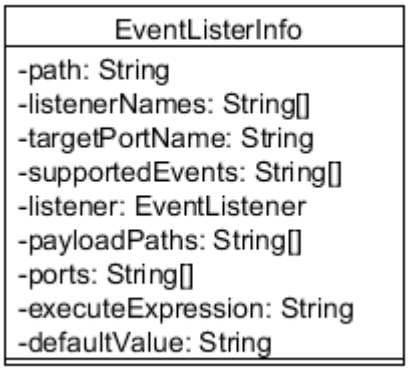
Sistema leidžia registruoti konkrečius įvykių klausytojus – Java klases realizuojančias reikiamas sąsajas bei dinamiškai sukuriamus klausytojus, kurių paskirtis reikšmių išgavimas iš komponento ir jų išsiuntimas, arba reikšmės priėmimas ir nustatymas.

Jei yra klausomasi komunikuojančių komponentų kuriamų įvykių, galima nurodyti dominančius prievadus – tik į nurodytus prievadus bus kreipiamas dėmesys kai bus kuriami įvykiai (`ports` atributas – 3.4.1.2).

Tarkime klausytojui įdomus įvykis kai visi prievadai yra gavę pranešimus (`AllInputPresent` įvykis – detaliau 3.1.5). Nenurodžius prievadų, šis klausytojas bus kviečiamas kai visi egzistuojantys įvesties prievadai bus užpildyti. Tuo tarpu nurodžius dominančius prievadus, bus tikrinami tik tie prievadai. Tokiu būdu konfigūracija leidžia klausytojams apdoroti įvykius tada, kai pranešimai ateina tik iš dominančių prievadų.

Nenurodžius konkretaus klausytojo objekto, jis yra generuojamas – detaliau apie tai 3.4.1.3.

3.4.1.2. Įvykių klausytojų informacija



27 pav. `EventListenerInfo` klasė naudojama aprašyti registruojamo įvykio klausytojo informaciją.

- **path** – kelias iki objekto, kuriam bus kviečiamas `addXXXListener` metodas. Užrašoma MVEL[7] išraiška.

[7] http://mvel.codehaus.org/

- **listenerNames** – klausytojų vardai, naudojami išskaičiuoti `addXXXListener` metodams (pvz.: mouse – `addMouseListener`, action – `addActionListener`).
- **supportedEvents** – įvykių tipai, į kurių klausysis sugeneruotas įvykių klausytojas (Pvz.: `DocumentListener` klausosi 3 įvykių: `insertUpdate`, `removeUpdate`, `changedUpdate`).
- **listener** – konkretus įvykių klausytojo objektas.
- **ports** – dominantys įvesties prievadai. Jei nurodytas bent vienas prievadas, įvykis bus generuojamas atsižvelgiant tik į nurodytų prievadų reikšmes, priešingu atveju – į visų įvesties prievadų.
- **targetPortName** – prievado, kuriuo bus siunčiamas pranešimas (išskaičiuota komponento reikšmė), vardas.
- **payloadPaths** – jei netenkina išskaičiuota komponento reikšmė, šiuo laukeliu galima sukurti siunčiamą reikšmę. Užrašoma MVEL išraiška.
- **executeExpression** – įvykus įvykiui kuriam klausytis skirtas šis įvykių klausytojas yra vykdoma ši išraiška. Užrašoma MVEL išraiška.
- **defaultValue** – įvykus įvykiui, komunikuojančiam komponentui nustatoma nurodyta reikšmė. Užrašoma MVEL išraiška.

3.4.1.3. Įvykių klausytojų generavimas

Sugeneruotų įvykių klausytojų paskirtis yra išgauti komponento reikšmę (lentelės duomenys, pasirinktas sąrašo elementas ir t.t) arba ją nustatyti. Nesant nurodytam konkrečiam įvykio klausytojo objektui, klausytojas generuojamas pagal tam tikrus `EventListenerInfo` atributus.

```
<bean parent="eventListenerInfo">
  <property name="path" value="document" />
  <property name="listenerName" value="document" />
  <property name="supportedEvents" value="insertUpdate" />
  <property name="targetPortName" value="outgoingPort" />
</bean>
```

28 pav. `DocumentListener` klausytojo reaguojančio tik į vieną įvykį – `insertUpdate` – deklaratyvus apibrėžimas.

```
getDocument().addDocumentListener(new DocumentListener() {
  @Override
  public void insertUpdate(DocumentEvent e) {
    // 1. Išskaičiuojama reikšmė
    Object value = resolveValue();
    // 2. ir siunčiama pranešimu per nurodytą prievadą
    transfer("outgoingPort", value);
  }
  // kiti klausytojo metodai tušti (supportedEvents)
});
```

29 pav. Sugeneruotam įvykių klausytojui, kuris apibrėžtas 28 pav., ekvivalentaus kodo ištrauka.

Klausytojo generavimas pagal `EventListenerInfo` nurodytus atributus vyksta tokia tvarka:

1. Surandami visi `addXXXListener` metodai, kiekvienam `listenerNames` nurodytam įvykio vardui. Metodų paieškoje taip pat naudojamas `path` atributas.
2. Surandamos norimų užregistruoti įvykių klausytojų sąsajos. 1 žingsnyje rasti metodai turi po vieną parametrą, kuris ir yra klausytojo sąsaja.
3. Įvykio klausytojo generavimas. Sugeneruojamas vienas klausytojo objektas (dinaminė proxy klasė) realizuojantis visas 2 žingsnyje rastas sąsajas.
4. Klausytojo elgsena priklauso nuo `supportedEvents` atribute nurodytų palaikomų įvykių. Klausytojas reaguos tik į šiuos įvykius, o kitų įvykių metodų kvietimai bus ignoruojami.

   Pagal nutylėjimą sugeneruotas klausytojas reaguoja į visus realizuotų sąsajų palaikomus įvykius.

3.4.2. Sugeneruotų klausytojų paskirtis

Generuojami įvykių klausytojai skirti atlikti dvi funkcijas. Įvykus įvykiams (Java Swing komponentų įvykiai ir specialūs komunikuojančių komponentų įvykiai) automatiškai siųsti pranešimus per nurodytus prievadus prijungtiems komponentams, arba nustatyti komponento reikšmę.

Siunčiami pranešimai yra konfigūruojami. Pagal nutylėjimą, komponento reikšmei išgauti yra naudojamos reikšmių išgavimo strategijos (žr. 3.4.3). Tačiau jei netenkina išgauta reikšmė, galima, `payloadPaths` atributo pagalba, nurodyti visiškai naują reikšmę. Pranešimas išsiunčiamas per `targetPortName` prievadą.

Tuo atveju kai įvykus įvykiui yra norima tiesiog pakeisti šio komunikuojančio komponento reikšmę, tereikia užrašyti jos išgavimo išraišką (išraiškų pavyzdžiai 31 pav.) – `defaultValue` atributas. Reikšmei nustatyti bus naudojama komponentų reikšmių nustatymo strategija (žr. 3.4.3).

Taip pat, jei netuščias `executeExpression` atributas, įvykus įvykiui bus vykdoma ši išraiška. Vienas iš galimų šio funkcionalumo panaudojimo būdų yra kitoks komponento reikšmės nustatymas – jei netenkina komponento reikšmių išgavimo strategijos realizacija.

3.4.3. Komponentų reikšmių radimas ir nustatymas

Pagrindiniams komponentams (1 priedas) yra paruoštos strategijos reikšmių gavimui bei nustatymui. Strategijos yra patalpintos `java.util.Map` struktūroje, kur identifikuojamos pagal klasę kuriai yra skirta strategija (30 pav).

Paieška vykdoma keliaujant per visas strategijas ir ieškant kuri iš jų yra tinkama dabartiniam komunikuojančiam komponentui. Tinkama strategija išrenkama patikrinus ar dabartinio komponento klasė yra tokia pati, arba poklasis tos klasės kuriai ta strategija yra skirta. Esant kelioms tinkamoms strategijoms naudojama ta kuri buvo rasta pirmiau.

```
<util:map id="commonComponentValueProviders" map-class="java.util.HashMap">
  <entry key="javax.swing.JList" value-ref="jListComponentValueProvider" />
  ...
</util:map>
<alias name="commonComponentValueProviders" alias="componentValueProviders"/>
```

30 pav. Visos reikšmių išgavimo strategijos patalpintos `java.util.Map` struktūroje ir identifikuojamos pagal komponento klasę, kuriai ir yra skirtos.

3.4.4. Kintamieji

Įvairiems atributams, kurie priima MVEL išraiškas (atributai pateikti 3.4.1.2. Pavyzdžiui `payloadPaths`), perduodamos reikšmės gali naudoti kintamuosius, pasiekiamus konfigūruojamo komunikuojančio komponento kontekste. Šie kintamieji yra tokie:

- **$component** – konfigūruojamas komunikuojantis komponentas.
- **$buffer** – įvesties buferis. Gražinamas `java.util.Map` objektas, kurio raktai yra įvesties prievadų vardai, o reikšmės – prievadais gautų pranešimų turinys.
- **$value** – komunikuojančio komponento reikšmė, išgauta pasitelkiant į pagalbą 3.4.3 poskyryje aprašytas reikšmių išgavimo strategijas.
- **$kintamojo_vardas** – bet kurio įvesties prievado vardas su `$` pradžioje. Gražinama reikšmė iš įvesties buferio, kuri yra identifikuojama nurodyto prievado vardu.

Keletas skirtingų kintamųjų panaudojimo scenarijų pademonstruota 31 pav. Pavyzdžiui viršutiniame pavyzdyje nurodyta išraiška gražinanti lentelės pirmos pasirinktos eilutės „TABLE_NAME“ stulpelio reikšmę (kur `$value` gražina visas pasirinktas lentelės ląsteles), jei

ta eilutė egzistuoja. O apačioje pateiktas užklausos šablonas su dviem naudojamais kintamaisiais – prievadų „schema“ ir „tableName“ reikšmėmis.

```
<property name="payloadPaths"
  value="$value != null and $value.size() &gt; 0 ? $value[0].TABLE_NAME : null" />
```

```
<property name="sql" value="SELECT * FROM @{$schema}.@{$tableName}" />
```

31 pav. Du skirtingi kintamųjų naudojimo scenarijai.

3.4.5. Specialūs komunikuojančių komponentų prievadai

Visi komponentai pagal nutylėjimą turi du specialius įvesties prievadus – `$_TRIGGER` (šį prievadą taip pat turi ir jungtys) ir `$_REFRESH`. Šie prievadai priima signalus, tai yra – tokie prievadai neturi tipo, ir jų priimami bei siunčiami pranešimai yra tušti – be tipo ir be turinio.

3.4.5.1. Pranešimų užlaikymas ir siuntimas

`$_TRIGGER` prievadas yra skirtas siunčiamiems pranešimams užlaikyti iki gaunamas signalas. Jei prie `$_TRIGGER` nėra prijungtų komponentų, pranešimų siuntimas vyksta iškart. Priešingu atveju, jei prie prievado yra prijungtų komponentų, šio prievado savininkas komunikuojantis komponentas arba jungtis užlaiko siunčiamus pranešimus – deda juos į buferį, iki bus sulauktas signalas – pranešimas – iš šio prievado. Atėjus signalui visi pranešimai iš išvesties buferio yra išsiunčiami – buferis tampa tuščias (iliustruota 16 pav. – komponentų atveju ir 21 pav. – jungčių atveju).

3.4.5.2. Komunikuojančio komponento atsinaujinimas

Tam tikriems komponentas gali kartas nuo karto reikėti atsinaujinti (iš naujo atlikti skaičiavimus, nuskaityti duomenis ir t.t). Pavyzdžiui, turimas komponentas atvaizduojantis failus esančius nurodytoje direktorijoje. Iš išorės pridedami nauji failai, tačiau komponentas nenuskaitęs direktorijos iš naujo, tų failų neatvaizduos. Reikalingas signalas leidžiantis pranešti komponentui apie būtinybę atsinaujinti.

Komunikuojančiam komponentui kiti komponentai gali liepti atsinaujinti, pasiųsdami pranešimą į `$_REFRESH` prievadą. Į šį prievadą patekę pranešimai yra apdorojami skirtingai, komunikuojantis komponentas nekurs įvykio, kad gautas pranešimas, tačiau sukurs atsinaujinimo įvykį – `RefreshEvent` (iliustruota 16 pav). Norint, kad komunikuojantį komponentą būtų galima priversti atsinaujinti iš išorės, teks užregistruoti tam skirtą klausytoją.

4. Realizacija ir testavimas

4.1. Komponentų ir klausytojų biblioteka

Kuriant demonstracinę taikomąją programą buvo naudojamasi keletu iš anksto paruoštų bendro naudojimo komponentų ir klausytojų.

Naudojamas komponentas valdantis prisijungimą prie duomenų bazės, ir jį perduodantis visiems jo prašantiems komponentams – `DBConnectionComponent` (5 priedas).

Taip pat naudojami du klausytojai:

1. Klausytojas, kuris komunikuojančius komponentus padaro neaktyviais, jei jie neturi nei vieno reikalaujamo kintamojo – `disableWhenNoInputPresent` (3 priedas).
2. Klausytojas kuris skirtas vykdyti SQL užklausoms įvykus laukiamiems įvykiams – `queryExecution` (4 priedas, 8 priedas - 11 priedas).

4.2. Demonstracinė taikomoji programa

Sistemos galimybių pademonstravimui buvo sukurta demonstracinė taikomoji programa, kurios iškvietimas pavaizduotas 32 pav. Programą sudaro 11 failų: 1 Java klasė (32 pav) su tipiniu sistemos iškvietimu, 1 deklaratyvus lango apibrėžimas (2 priedas) ir 9 deklaratyvūs komponentai bei jų sąryšių aprašai (3 priedas - 11 priedas).

Iš viso 382 kodo eilutės, kur Java kodas užima 18 eilučių, XML – deklaratyvus komponentų ir jungčių apibrėžimas – 312 eilučių, ir YAML lango deklaratyvus aprašas – 52 eilutes.

```
public class Main {

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        SwingJavaBuilderLauncher launcher = new SwingJavaBuilderLauncher();

        try {
          UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
          launcher.initialize("MainWindow.yml").setVisible(true);
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    });
  }
}
```

32 pav. Tipinis Java kodas reikalingas langui sukurti. Šiame pavyzdyje sukuriamas 33 pav. pavaizduotas langas.

Tokią pačią taikomąją programą sukūrus nenaudojant komunikuojančių komponentų – iš viso susidarė 315 kodo eilučių. Iš jų Java kodas užima 263 eilutes, o YAML lango deklaratyvus aprašas tiek pat – 52 eilutes. Matuota neįskaičiuojant minėtų pagalbinių įvykių klausytojų bei komponentų.

Bendru atveju, didžiausią įtaką taikomosios programos kodo kiekio sumažėjimui turėjo tai, jog:

- Karkasas atskiria komponentų reprezentaciją nuo jų veikimo logikos – komponentų savybės ir jų išdėstymas apibrėžiamas deklaratyviai
- Dinamiškai kuriami karkaso įvykių klausytojai
- Generuojamas komponentų kodas, skirtas reikšmių apsikeitimui tarp komponentų

4.2.1. Naudojamų komponentų sąrašas

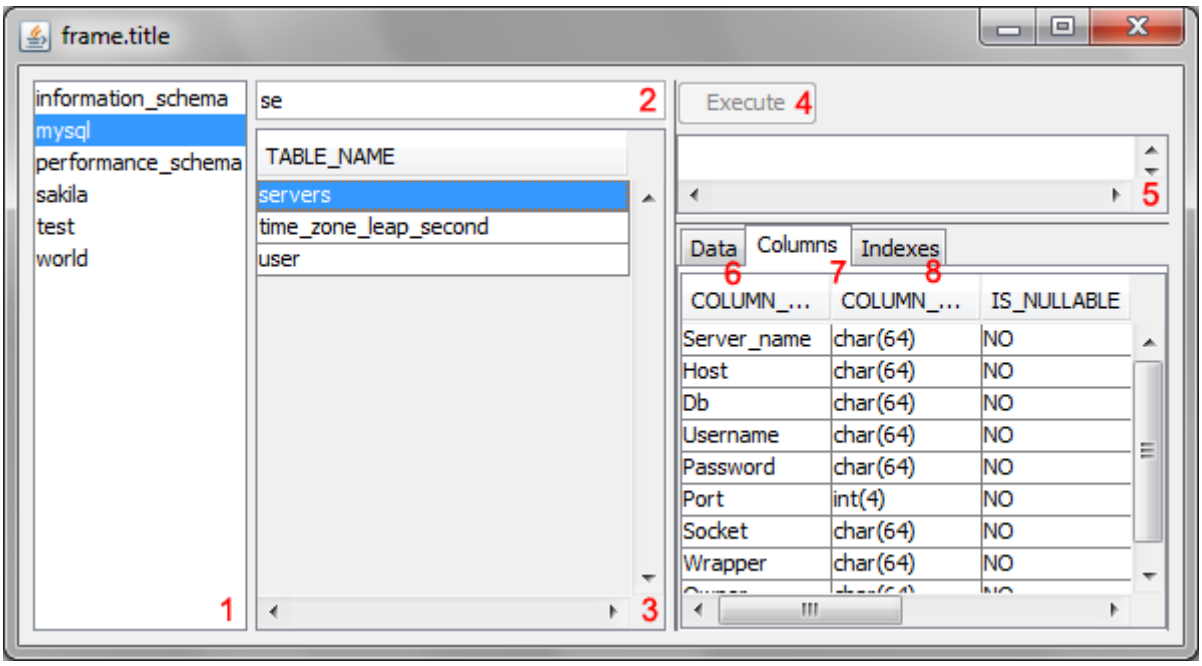
33 pav. pavaizduotas langas yra sudarytas iš 8 matomų komponentų ir 1 nematomo. Šie komponentai yra tokie:

1. Schemų sąrašas (4 priedas).
2. Lentelių vardų filtravimo laukelis (6 priedas).
3. Lentelių (išfiltruotų) sąrašas (10 priedas).
4. Užklausų vykdymo („Execute“) mygtukas (3 priedas).
5. Laukelis skirtas SQL užklausoms įvesti (7 priedas).
6. Duomenų lentelė (9 priedas).
7. Stulpelių lentelė (8 priedas).
8. Indeksų lentelė (11 priedas).
9. Nematomas komponentas atsakingas už susijungimą su duomenų baze (5 priedas).

Sąraše, prie kiekvieno komunikuojančio komponento pateikta nuoroda į priedą kuriame jis yra apibrėžtas. Tame pačiame priede po komponento aprašo yra jungties aprašas, kuris parodo, kokie komponentai priklauso nuo šio komponentas teikiamų reikšmių.

4.2.2. Programos veikimo scenarijus

Ši (33 pav) taikomoji programa leidžia peržiūrėti duomenų bazės meta-informaciją. Duomenų bazės duomenys nurodomi specialiam komunikuojančiam komponentui (5 priedas), kuris pasirūpina prisijungimu prie duomenų bazės ir aprūpina visus prisijungimo prie duomenų bazės prašančius komponentus.



33 pav. Taikomosios programos langas

Užsikrovus programai 1 komponentas atvaizduoja visas egzistuojančias duomenų bazės schemas. Pasirinkus kurią nors vieną iš schemų, 3 komponentas atvaizduoja visas toje schemoje sukurtos lentelės, išfiltruotas pagal 2 komponente įvestą reikšmę. Pasirinkus kurią nors vieną iš pateiktų lentelių, 6 komponentas atvaizduoja toje lentelėje esančius duomenis, 7 – lentelės stulpelius ir 8 – indeksus.

5 komponentas SQL užklausas siunčia 6 komponentui, kuris jas įvykdo ir atvaizduoja rezultatą. Užklausų siuntimas vykdomas tik paspaudus 4 komponentą – mygtuką, kuris būna neaktyvus, kol nėra įvesta jokio teksto į 5 komponentą.

4.3. Sprendimo palyginimas su kitais karkasais

Kaip jau minėta, sistemos veikimas skirstomas į 3 dalis. Dvi iš jų, tai yra komponentų kūrimas, su savybių nustatymu bei jų vizualus išdėstymas atlieka tą pačią funkciją kaip ir sąsajos kūrimas naudojantis WYSIWYG (angl. What You See Is What You Get) įrankiais. Komunikuojančių komponentų karkasas iš aprašų kuria komponentus ir nustato jų savybės, o WYSIWYG įrankių atveju kuriamas specifinis aprašas kuriamo lango atvaizdavimui kūrimo metu bei iš karto generuojamas tą langą galintis atvaizduoti Java kodas. Abiem atvejais naudojami aprašai, tačiau jų paskirtis yra skirtinga – vienas interpretuojamas programos vykdymo metu, o kitas naudojamas peržiūrai ir kodo generavimui.

Apžvalgoje minėti deklaratyvūs karkasai taip interpretuoja aprašus programos vykdymo metu ir atlieka tas pačias funkcijas, kaip ir komunikuojančių komponentų karkaso minėtos dvi dalys – komponentų kūrimas, jų savybių nustatymas ir jų išdėstymas languose.

Sukurto karkaso skirtumai nuo kitų karkasų pasireiškia dalyje kur apibrėžiamos komponentų tarpusavio priklausomybės (tam kad įgalinti tų priklausomybių apibrėžimą, atitinkamai yra naudojami ir modifikuoti komponentai). Komunikuojančių komponentų karkase, komponentai konfigūruojami jiems priskyrus prievadus ir įvykių klausytojus. Prievadai iš esmės yra ne kas kita, kaip įvykių klausytojuose naudojamų išorės (kitų komponentų) kintamųjų sąrašas.

4.3.1. Komponentai ir įvykių klausytojai

Kadangi komponentai deklaruoja savo elgesį, todėl net ir įvykių klausytojai skirti reaguoti į kitų komponentų pasikeitimus yra registruojami tam pačiam komponentui. Tai šiek tiek skiriasi nuo to, kaip panašus funkcionalumas galėtų būti įgyvendintas nenaudojant šio karkaso.

Pavyzdžiui, komponentų K_1, K_2, ..., K_n klausytojai priklauso nuo išorinio kintamojo – komponento K_0 reikšmės. Paprastai, nenaudojant karkaso, įvykių klausytojas, užregistruotas reaguoti į komponento K_0 reikšmės pasikeitimą, galėtų apdoroti visus priklausančius (K_1, K_2, ..., K_n) komponentus.

Tačiau, komunikuojančių komponentų sistema išvengdama tiesioginių priklausomybių tarp komponentų registruoja klausytojus kiekvienam iš komponentų K_1, K_2, ..., K_n. Šie klausytojai reaguos į komponento K_0 reikšmės pasikeitimą (K_0 siųs pranešimą pasikeitus jo reikšmei). Dėl šios priežasties reikia sukurti n + 1 klausytojų – vienas K_0 komponento reikšmės pasikeitimo metu siųs pranešimą, o likę n klausytojų skirti K_1, K_2, ..., K_n komponentams – klausysis pranešimų iš K_0 komponento.

4.3.2. Įvykių klausytojai

Įvykių klausytojai sistemoje gali būti registruojami dviem būdais – nurodant konkretų įvykių klausytoją, arba apibrėžiant geidžiamas klausytojo savybes, pagal kurias jis bus sukurtas dinamiškai (3.4.1).

4.3.2.1. Konkrečių klausytojų registravimas

Specialūs klausytojai – konkrečios Java klasės, yra registruojami tiesiogiai komponentui. Karkasas pasirūpina šių klasių objektų sukūrimu (Komponento su klausytoju apibrėžimas iliustruotas 34 pav). Klausytojų registravimas iš esmės niekuo nesiskiria nuo tokių pačių klausytojų registravimo nesinaudojant komunikuojančių komponentų karkasu, ir tai darant su Java Swing priemonėmis. Tiesa, sistema leidžia užregistruotiems klausytojams pasiekti prievadų reikšmes (kintamuosius) bei jais manipuliuoti (3.4.4).

```
<bean id="button" parent="jButton">
  <property name="eventListenerInfoList">
    <list>
      <bean parent="actionListener">
        <property name="listener">
          <bean class="ListenerClass" />
        </property>
      </bean>
    </list>
  </property>
</bean>
```

34 pav. Įvykio klausytojo registravimas mygtuko komponentui. `ListenerClass` yra pilnas klasės, kuri realizuoja `ActionListener` sąsają, vardas.

Abiem atvejais turi būti paruošta klasė apdorojanti įvykį(-ius) bei užregistruota komponentui. Skiriasi tik tos klasės užregistravimo būdai. Vienu atveju tai atliekama deklaratyviai, kitu rašant Java kodą. Tai leidžia išvengti Java kodo skirto klausytojų registravimui rašymo.

Panašiai įvykių klausytojai registruojami ir CookSwing (detaliau aprašytas 2.3.1) karkase (35 pav). Klausytojas, šiuo atveju `buttonAction`, privalo būti sukurtas ir paviešintas (`public` atributas).

```
public ActionListener buttonAction = new ActionListener () { ... };
```

```
<button actionlistener="buttonAction" />
```

35 pav. CookSwing karkaso įvykių klausytojo apibrėžimas (viršuje) ir jo deklaratyvus priskyrimas mygtukui (apačioje).

SDL/Swing karkasas įvykių apdorojimą atlieka šiek tiek kitaip. Registruojamas ne konkretus klausytojas, o metodas be parametrų (36 pav), kuris yra iškviečiamas iš, šiuo atveju `ActionListener`, klausytojo.

Toks įvykių apdorojimo būdas turi ir trūkumų: negalima pasiekti originalaus įvykio objekto ir komponento sugeneravusio įvykį, taip pat negalima klausytis visų komponentų generuojamų įvykių, kadangi tik tam tikra dalis jų yra leidžiama registruoti deklaratyviai.

```
public void action();
```

```
button do="action"
```

36 pav. SDL/Swing karkaso įvykio apdorojimo metodas (viršuje) ir jo deklaratyvus priskyrimas mygtukui (apačioje)

Swing JavaBuilder įvykius apdoroja labai panašiai į SDL/Swing taikomą metodą – įvykių apdorojimą atlieka ne konkretūs klausytojai o metodai, kurie kviečiami iš karkaso užregistruoto klausytojo. Šis karkasas siūlo didesnį įvykius apdorojančių metodų pasirinkimą (37 pav).

```
private void save(JButton button, ActionEvent event) {}
private void save(ActionEvent event) {}
private void save(JButton button) {}
private void save() {}
```

```
JButton(onAction=save)
```

37 pav. Swing Javabuilder karkaso įvykio apdorojimo metodai (viršuje) ir jų deklaratyvus priskyrimas mygtukui (apačioje).

Tačiau vis tiek išlieka ta pati problema, kad galima klausytis tik tam tikrų įvykių. Tiesa, karkasas gali būti praplėstas ir papildytas palaikomų įvykių sąrašu. Taip pat kaip ir SDL/Swing karkase, negalima registruoti konkrečių klausytojų klasių ar objektų.

4.3.2.2. Dinamiškai kuriamų klausytojų registravimas

Komunikuojančių komponentų karkasas turi vieną ypatingą savybę, kurios atitikmens nėra nagrinėtuose karkasuose – iš deklaratyvaus aprašo dinamiškai sukuriami įvykių klausytojai. Šių klausytojų funkcijos aprašytos 3.4.1 ir 3.4.2. Dinaminiai įvykių klausytojai atstoja komponentų komunikavimo logiką – reikšmių apsikeitimą (priėmimą ir išsiuntimą), ji išreiškiama deklaratyviai (28 pav ir 29 pav iliustruotas deklaratyvus įvykio klausytojo priskyrimas komponentui ir jam ekvivalentus Java kodas).

Šie klausytojai naudojasi komponentų reikšmių išgavimo ir nustatymo strategijomis, kurios turi būti realizuotos Java kodu. Žinoma karkasas leidžia nurodyti ir kitokias siunčiamas reikšmes, jei tam yra poreikis.

Nemažai komponentų naudojami tik jų būsenai išgauti ar panašioms trivialioms operacijoms atlikti. Tokie klausytojai leidžia išvengti pasikartojančio kodo, skirto toms operacijoms atlikti, rašymo.

Kadangi reikšmes siunčiančių ir nustatančių klausytojų elgesys dar papildomai konfigūruojamas, todėl šių klausytojų elgesį galima pritaikyti beveik visiems norimiems atvejams.

4.3.2.3. Įvykių klausytojų daugkartinis panaudojimas

Karkasas leidžia kurti įvykių klausytojus, kurie gali priklausyti nuo išorinių parametrų (prievadų reikšmių), o jei tai dinaminiai klausytojai, tai dar gali būti ir labai lanksčiai konfigūruojami. Tai leidžia kurti bendresnio pobūdžio klausytojus, kurie yra lengviau panaudojami skirtinguose scenarijuose.

4.3.3. Trūkumai

Karkasas turi ir keletą neesminių trūkumų. Kaip ir kiti deklaratyvūs karkasai aptarti darbe, kelių komponentų kompozicijos negalima apibrėžti kaip naujo komponento, kas yra nesunkiai realizuojama naudojantis Java kalbos priemonėmis.

Kadangi komponentai negali tiesiogiai priklausyti vieni nuo kitų, yra kuriami papildomi įvykių klausytojai. Kitaip tariant, visiems susijusiems komunikuojantiems komponentams reikalingi reikšmių siuntimui ir jų priėmimui bei apdorojimui skirti įvykių klausytojai.

Išlaikant karkaso lankstumą nebuvo supaprastinta klausytojų registracija, todėl registruojant klausytojus reikia daugiau žinių apie naudojamus komponentus ir jų generuojamus įvykius.

4.4. Apibendrinimas

Sistema nebuvo išsamiai ištestuota kuriant sudėtingas taikomąsias programas, todėl nevisai aišku, kaip tam tikri sprendimai atsilieps patirčiai naudojant karkasą tokiose programose. Remiantis bendrais sukurto karkaso realizacijos principais buvo atliktas palyginimas su kitais deklaratyviais karkasais.

Nors vartotojo sąsajos apibrėžimas ir susideda iš trijų dalių (komponentų aprašas, jungčių aprašas, vartotojo sąsajos aprašas), tačiau tai neapsunkina taikomosios programos kūrimo. Kaip tik, kadangi šių dalių atsakomybės skiriasi, labai paprasta atlikinėti pakeitimus vienoje iš dalių, nesibaiminant ką nors sugadinti kitoje – jos nesusijusios.

Pavyzdinėje programoje naudojantis sistema, komponentų ir jungčių apibrėžimai užėmė didžiąją dalį viso kodo – daugiau nei Java kodas taikomojoje programoje nesinaudojant sistema. Tačiau svarbiausia tai, kad dalys, kurios tiesiogiai įtakoja komponento veikimo logiką yra apibrėžiamos gana paprastai bei yra konfigūruojamos.

Sukurtoji sistema skatina naudotis pakartotinai panaudojamais ir konfigūruojamais klausytojais bei komponentais. Vien šiame pavyzdyje užklausoms vykdyti skirtas klausytojas (4.1) buvo panaudotas keliuose skirtinguose komponentuose. Turint išsamią panašaus pobūdžio konfigūruojamų klausytojų ir komponentų biblioteką pritaikytą konkrečiai sričiai, būtų visai įmanoma išsiversti su minimaliu Java kodo rašymu (klausytojų su specifiniu funkcionalumu kūrimas) net ir kuriant sudėtingesnes taikomąsias programas.

Didžiausias sukurtos sistemos privalumas yra tai, jog ji leidžia lengviau pamatyti kaip veikia atskiri taikomosios programos komponentai ir kokie tų komponentų sąryšiai su kitais komponentais. Kadangi komponentai deklaruoja savo elgseną, tai pažvelgus į komponento aprašą galima tiksliai pasakyti, kokių įvykių laukia šis komponentas bei kaip jis elgsis jiems įvykus.

Tiesa, karkaso lankstumas turi ir trūkumų, kadangi tam tikros karkaso vietos reikalauja daugiau žinių apie naudojamus komponentus, priešingai nei kituose karkasuose, kur nepalaikomas rečiau naudojamas funkcionalumas, tačiau kitos dalys yra stipriai supaprastintos.

Rezultatai ir išvados

Darbo tikslas buvo sukurti mechanizmą leidžiantį kuo didesnę vartotojo sąsajos dalį apibrėžti deklaratyviai. Tam tikslui pasiekti reikėjo apibrėžti ir įgyvendinti architektūrą, leidžiančią kurti Java vartotojo sąsajos, įskaitant ir Swing, komponentus deklaruojančius savo elgseną.

Darbo metu buvo išanalizuoti įvairūs komponentinių architektūrų stiliai, juose naudojamų komponentų paskirtis, sandara, jų komunikavimo ypatumai bei jungčių vaidmuo komponentų kompozicijoje. Taip pat apžvelgtos deklaratyvų Java Swing vartotojo sąsajos apibrėžimą leidžiančių karkasų funkcijos bei galimybės.

Remiantis išnagrinėtais šaltiniais buvo sukurta sistema, leidžianti kurti taikomąsias programas su Java Swing vartojo sąsaja naudojantis beveik vien tik deklaratyviais taikomosios programos dalių apibrėžimais. Taip pat integruotas deklaratyvų vartotojo sąsajos apibrėžimą įgalinantis karkasas – Swing JavaBuilder. Jis atsakingas už vartotojo sąsajos komponentų savybių nustatymą ir vizualų jų išdėstymą. Komponentai ir jungtys apibrėžiami Spring karkaso pagalba.

Pavyko įgyvendinti sistemą, kurioje Java Swing komponentai deklaruoja savo elgesį, yra apibrėžiami deklaratyviai ir tarpusavyje komunikuoja tik per sąsajas – prievadus. Sukurtoji sistema lengvai plečiama – naujų komponentų naudojimas nereikalauja jokio papildomo konfigūravimo, kadangi komponentai yra automatiškai paruošiami programos darbo metu.

Nors komunikuojančių komponentų aprašai ir užima daugiau vietos bei kartais gali pasirodyti pernelyg išpūsti, tačiau didelis architektūros privalumas yra tai, jog ji leidžia lengviau suprasti kaip veikia atskiri taikomosios programos komponentai. Užtenka pažvelgti į komponento aprašą tam kad būtų galima tiksliai pasakyti, kokių įvykių laukia šis komponentas bei kaip jis elgsis jiems įvykus.

Šią sistemą būtų galima išplėsti, įvedus automatinį komponentų sujungimo mechanizmą. Tokiu atveju būtų galima bent jau dalinai atsisakyti jungčių apibrėžimo. Šis mechanizmas galėtų analizuoti visus prieinamus duomenis apie komunikuojančių komponentų siūlomus bei reikalaujamus kintamuosius (tipas, vardas, rūšis ir t.t.), ir rezultate sukurti jungtį, kuri prireikus taip pat mokėtų ir adaptuoti nesuderinamus komponentus.

Šaltinių sąrašas

[AG97] Robert Allen, David Garlan. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, 6(3), July 1997, pp. 213-249.

[AO09] Abdelkrim Amirat, Mourad Oussalah. First-Class Connectors to Support Systematic Construction of Hierarchical Software Architecture. In: Journal of Object Technology, Vol. 8, No. 7, November - December 2009, pp. 107-130.

[AS02] Matthias Anlauff, Asuman Sunbul. Towards Component Based Systems: Refining Connectors. In: Electronic Notes in Theoretical Computer Science, 70, Elsevier, 2002.

[BBC02] Andrea Bracciali, Antonio Brogi, Carlos Canal. Systematic Component Adaptation, Electronic Notes in Theoretical Computer Science, 66, Elsevier, 2002.

[BB07] Marco Antonio Barbosa, Luis Soares Barbosa. An Orchestrator for Dynamic Interconnection of Software Components, Electronic Notes in Theoretical Computer Science, 181, 2007, pp. 49–61.

[Cam99] Grady. H. Campbell. Adaptable components. In: ICSE 1999, IEEE Press, 1999, pp. 685 – 686.

[EL10] Perla Velasco Elizondo, Kung-Kiu Lau. A catalogue of component connectors to support development with reuse. In: Journal of Systems and Software, 83(7) 1165-1178, 2010.

[FDH08] Luc Fabresse, Christophe Dony, Marianne Huchard. Foundations of a simple and unified component-oriented language. In: Journal of Computer Languages, Systems and Structures, 34(2–3), 2008, pp. 130–149.

[GS01] D. Garlan and B. Schmerl. Component-based software engineering in pervasive computing environments. In: 4th ICSE Workshop on Component-Based Software Engineering, 2001.

[LW07] Kung-Kiu Lau, Zheng Wang, Software component models, IEEE Transactions on Software Engineering, 33(10), 2007, pp. 709–724.

[OEK+10a] Fernando Orejas, Hartmut Ehrig, Markus Klein, Julia Padberg, Elvira Pino, Sonia Pérez. A Generic Approach to Connector Architectures Part I: The General Framework. Fundamenta Informaticae, 99(2010), pp. 63–93.

[OEK+10b] Fernando Orejas, Hartmut Ehrig, Markus Klein, Julia Padberg, Elvira Pino, Sonia Pérez. A Generic Approach to Connector Architectures Part II: Instantiation to Petri Nets and CSP. Fundamenta Informaticae, 99(2010), pp. 95-124.

[Sun97] Sun Microsystems. JavaBeans™ API specification, Version 1.01-A. 1997. http://download.oracle.com/otn-pub/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/beans.101.pdf

[Sha93] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. Proc. Workshop Studies of Software Design, May 1993, pp. 17-32.

[TMA+96] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr. A Component - and Message-based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, 22(6), June 1996, pp. 390–406.

[WH05] Wenpin Jiao, Hong Mei. Dynamic Architectural Connectors in Cooperative Software Systems. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005). IEEE Computer Society, pp. 477–486.

Priedai

1 priedas. Sąrašas Java Swing komponentų, kuriems paruoštos reikšmių išgavimo ir nustatymo strategijos

javax.swing.JLabel

javax.swing.text.JTextComponent

javax.swing.JList

javax.swing.JTable

javax.swing.JComboBox

javax.swing.JToggleButton

javax.swing.JSpinner

javax.swing.JProgressBar

javax.swing.JSlider

2 priedas. Langa aprašas MainWindow.yml

```
JFrame(name=frame, title=frame.title, size=packed,
defaultCloseOperation=exitOnClose):
  - JSplitPane(name=split1,orientation=horizontalSplit,border=0):
    - JPanel():
      - tableNameFilter()
      - JScrollPane(name=listSchemasScroll): listSchemas()
      - JScrollPane(name=scroll1, horizontalScrollBarPolicy=always,
verticalScrollBarPolicy=always): filteredTables()
      - MigLayout: |
        [[insets 0]]
        [grow]                    [grow]
        listSchemasScroll+1+*|    tableNameFilter                  []
                                  ^scroll1                      [grow]
    - JPanel(name=pane2):
      - JSplitPane(name=split2,orientation=verticalSplit,border=0):
        - JPanel():
          - JScrollPane(name=queryAreaScroll, horizontalScrollBarPolicy=always,
verticalScrollBarPolicy=always): queryArea(font=12pt monospaced)
          - buttonExecuteQuery(text=Execute)
          - MigLayout: |
            [[insets 0]]
            [grow]
            buttonExecuteQuery      []
            queryAreaScroll         [grow]
        - JPanel():
          - JTabbedPane(name=tabs):
            - JPanel(tabTitle=Data):
              - JScrollPane(name=tableDataScroll, horizontalScrollBarPolicy=always,
verticalScrollBarPolicy=always): tableData(autoResizeMode=0)
              - MigLayout: |
                [[insets 0]]
                [grow]
                tableDataScroll     [grow]
            - JPanel(tabTitle=Columns):
              - JScrollPane(name=tableColumnsScroll,
horizontalScrollBarPolicy=always, verticalScrollBarPolicy=always):
tableColumns(autoResizeMode=0)
              - MigLayout: |
                [[insets 0]]
                [grow]
                tableColumnsScroll  [grow]
            - JPanel(tabTitle=Indexes):
              - JScrollPane(name=tableIndexesScroll,
horizontalScrollBarPolicy=always, verticalScrollBarPolicy=always):
tableIndexes(autoResizeMode=0)
              - MigLayout: |
                [[insets 0]]
                [grow]
```

```
                tableIndexesScroll  [grow]
          - MigLayout: |
            [[insets 0]]
            [grow]
            Tabs  [grow]
      - MigLayout: |
        [[insets 0]]
        [grow]
        split2 [grow]
  - MigLayout: |
      [grow]
      split1 [grow]
```

3 priedas. „Execute“ mygtuko apibrėžimas faile button-execute-query-component-beans.xml

```
<bean id="buttonExecuteQuery" parent="jButton">
  <property name="name" value="buttonExecuteQuery" />
  <property name="eventListenerInfoList">
    <list merge="true">
      <bean parent="disableComponentWhenNoInputPresent" />
    </list>
  </property>
</bean>

<bean id="buttonExecuteQuery_queryArea_Connector" parent="abstractConnector">
  <constructor-arg value="buttonExecuteQuery:actionPerformed" />
  <constructor-arg value="queryArea_tableData_Connector:$_TRIGGER" />
</bean>
```

4 priedas. Schemas atvaizduojančio komponento apibrėžimas faile list-schemas-component-beans.xml

```
<bean id="listSchemas" parent="jList">
  <property name="name" value="listSchemas" />
  <property name="connectionPorts">
    <map>
      <entry key="connection">
        <bean parent="inputPort">
          <property name="payloadType" value="java.sql.Connection" />
        </bean>
      </entry>
      <entry key="selectedSchema" value-ref="outputString" />
    </map>
  </property>
  <property name="eventListenerInfoList">
    <list>
      <bean parent="allInputPresentListener">
        <property name="listener">
          <bean parent="queryExecution">
            <property name="query" value="show databases;" />
          </bean>
        </property>
      </bean>

      <bean parent="listSelectionListener">
        <property name="targetPortName" value="selectedSchema" />
      </bean>
    </list>
  </property>
</bean>

<bean id="listSchemas_Connector" parent="abstractConnector">
  <constructor-arg value="listSchemas:selectedSchema" />
```

```
    <constructor-arg value="filteredTables:schema, tableColumns:schema,
tableData:schema, tableIndexes:schema" />
  </bean>
```

5 priedas. Duomenų bazės susijungimą valdančio komponento apibrėžimas faile mysql-db-connection-component-beans.xml

```
  <bean id="dbConnection" parent="dbConnectionComponent">
    <property name="name" value="mysqlDBConnection" />
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="dbUrl" value="jdbc:mysql://localhost:3306/test" />
    <property name="dbUser" value="root" />
    <property name="dbPassword" value="pass" />
    <property name="eventListenerInfoList">
      <list>
        <bean parent="connectedToPortListener">
          <property name="path" value="$ports.connection" />
          <property name="targetPortName" value="connection" />
          <property name="payloadPaths" value="connection" />
        </bean>
      </list>
    </property>
  </bean>

  <bean id="dbConnection_Connector" parent="abstractConnector">
    <constructor-arg value="dbConnection:connection" />

    <constructor-arg value="filteredTables:connection, tableData:connection,
tableColumns:connection, tableIndexes:connection, listSchemas:connection" />
  </bean>
```

6 priedas. Laukelio filtruojančio lentelių vardus apibrėžimas faile name-filter-component-beans.xml

```
  <bean id="tableNameFilter" parent="jTextField">
    <property name="name" value="tableNameFilter" />
    <property name="eventListenerInfoList">
      <list>
        <ref bean="documentListener" />
        <bean parent="connectedToPortListener">
          <property name="path" value="$ports.text" />
          <property name="targetPortName" value="text" />
        </bean>
      </list>
    </property>
  </bean>

  <bean id="tableNameFilter_filteredTables_Connector" parent="abstractConnector">
    <constructor-arg value="tableNameFilter:text" />
    <constructor-arg value="filteredTables:tableName" />
  </bean>
```

7 priedas. Užklausai įvesti skirto laukelio apibrėžimas faile query-component-beans.xml

```
  <bean id="queryArea" parent="jTextArea">
    <property name="name" value="queryArea" />
    <property name="eventListenerInfoList">
      <list>
        <ref bean="documentListener" />
      </list>
    </property>
  </bean>

  <bean id="queryArea_tableData_Connector" parent="abstractConnector">
    <constructor-arg value="queryArea:text" />
```

```
    <constructor-arg value="tableData:query" />
  </bean>

  <bean id="queryArea_buttonExecuteQuery_Connector" parent="abstractConnector">
    <constructor-arg value="queryArea:text" />
    <constructor-arg value="buttonExecuteQuery:defaultInput" />
  </bean>
```

8 priedas. Lentelės stulpelius atvaizduojančio komponento apibrėžimas faile table-columns-component-beans.xml

```
  <bean id="tableColumns" parent="jTable">
    <property name="name" value="tableColumns" />
    <property name="connectionPorts">
      <map>
        <entry key="connection">
          <bean parent="inputPort">
            <property name="payloadType" value="java.sql.Connection" />
          </bean>
        </entry>
        <entry key="schema" value-ref="inputString" />
        <entry key="tableName" value-ref="inputString" />
      </map>
    </property>
    <property name="eventListenerInfoList">
      <list>
        <bean parent="allInputPresentListener">
          <property name="ports" value="tableName" />
          <property name="listener">
            <bean parent="queryExecution">
              <property name="query"
                        value="SHOW COLUMNS FROM @{$schema}.@{$tableName};" />
            </bean>
          </property>
        </bean>
      </list>
    </property>
  </bean>
```

9 priedas. Lentelės duomenis atvaizduojančio komponento apibrėžimas faile table-data-component-beans.xml

```
  <bean id="tableData" parent="jTable">
    <property name="name" value="tableData" />
    <property name="connectionPorts">
      <map>
        <entry key="connection">
          <bean parent="inputPort">
            <property name="payloadType" value="java.sql.Connection" />
          </bean>
        </entry>
        <entry key="schema" value-ref="inputString" />
        <entry key="tableName" value-ref="inputString" />
        <entry key="query" value-ref="inputString" />
      </map>
    </property>
    <property name="eventListenerInfoList">
      <list>
        <bean parent="allInputPresentListener">
          <property name="ports" value="schema, tableName" />
          <property name="listener">
            <bean parent="queryExecution">
              <property name="query" value="SELECT * FROM @{$schema}.@{$tableName}"
/>
            </bean>
          </property>
```

```
        </bean>
        <bean parent="hasInputPresentListener">
          <property name="ports" value="query" />
          <property name="listener">
            <bean parent="queryExecution" />
          </property>
        </bean>
      </list>
    </property>
  </bean>
```

10 priedas. Išfiltruotų duomenų bazės lentelių sąrašą atvaizduojančio komponento apibrėžimas faile table-filtered-tables-component-beans.xml

```
  <bean id="filteredTables" parent="jTable">
    <property name="name" value="filteredTables" />
    <property name="selectionMode">
      <util:constant static-field="javax.swing.ListSelectionModel.SINGLE_SELECTION"/>
    </property>
    <property name="connectionPorts">
      <map>
        <entry key="connection">
          <bean parent="inputPort">
            <property name="payloadType" value="java.sql.Connection" />
          </bean>
        </entry>
        <entry key="tableName" value-ref="inputString" />
        <entry key="schema" value-ref="inputString" />
        <entry key="selectedTableName" value-ref="outputString" />
      </map>
    </property>
    <property name="eventListenerInfoList">
      <list>
        <bean parent="allInputPresentListener">
          <property name="ports" value="connection, tableName, schema" />
          <property name="listener">
            <bean parent="queryExecution">
              <property name="query" value="SELECT DISTINCT TABLE_NAME FROM
INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME LIKE '%@{$tableName}%' AND
TABLE_SCHEMA='@{$schema}';" />
            </bean>
          </property>
        </bean>
        <bean parent="listSelectionListener">
          <property name="targetPortName" value="selectedTableName" />
          <property name="payloadPaths" value="$value != null and $value.size() &gt;
0 ? $value[0].TABLE_NAME : null" />
        </bean>
      </list>
    </property>
  </bean>

  <bean id="filteredTables_Connector" parent="abstractConnector">
    <constructor-arg value="filteredTables:selectedTableName" />

    <constructor-arg value="tableData:tableName, tableColumns:tableName,
tableIndexes:tableName " />
  </bean>
```

11 priedas. Lentelės indeksus atvaizduojančio komponento apibrėžimas faile table-indexes-component-beans.xml

```
  <bean id="tableIndexes" parent="jTable">
    <property name="name" value="tableIndexes" />
```

```
    <property name="connectionPorts">
      <map>
        <entry key="connection">
          <bean parent="inputPort">
            <property name="payloadType" value="java.sql.Connection" />
          </bean>
        </entry>
        <entry key="schema" value-ref="inputString" />
        <entry key="tableName" value-ref="inputString" />
      </map>
    </property>
    <property name="eventListenerInfoList">
      <list>
        <bean parent="allInputPresentListener">
          <property name="ports" value="schema, tableName" />
          <property name="listener">
            <bean parent="queryExecution">
              <property name="query" value="SHOW INDEX FROM
@{$schema}.@{$tableName};" />
            </bean>
          </property>
        </bean>
      </list>
    </property>
  </bean>
```