УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

Жарко Р. Мијаиловић

# Развој графичког корисничког интерфејса пословних апликација базиран на подели одговорности и објектном моделу података са *UML* семантиком

докторска дисертација

Београд, 2014.

УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

Жарко Р. Мијаиловић

# Развој графичког корисничког интерфејса пословних апликација базиран на подели одговорности и објектном моделу података са *UML* семантиком

докторска дисертација

Београд, 2014.

UNIVERSITY OF BELGRADE
FACULTY OF ELECTRICAL ENGINEERING

Žarko R. Mijailović

# Graphical User Interface Development for Business Applications Based on Separation of Concerns and Data Models with UML Semantics

Doctoral Dissertation

Belgrade, 2014

## Захвалнице

Захваљујем свим члановима комисије за преглед и оцену ове дисертације на корисним примедбама и сугестијама и на помоћи у изради ове дисертације.

Посебно захваљујем ментору, проф. др Драгану Милићеву, најпре на знању које ми је пружио и које ме је обликовало као инжењера и истраживача, затим на времену и труду уложеном у фази дизајна и развоја технике описане у овој дисертацији и коначно, на помоћи у писању научних радова који су неизоставни део ове докторске дисертације.

Цео подухват дизајна и имплементације технике за развој корисничких интерфејса не би био могућ без свих колега и сарадника у предузећу СОЛ Софтвер д.о.о. који су учествовали, заједно са мном, у развоју једне шире и свеобухватне (енгл. *end-to-end*) технологије за развој информационих система и пословних апликација под називом *SOLoist*. Техника описана у овој дисертацији је само један, али крупан и важан део тог ширег развоја. Ширина и контекст били су есенцијални састојци успеха ове технике како у индустријским пројектима, тако и у академским круговима. Прва верзија последње генерације ове технологије базиране на веб платформи настала је у врло специфичним околностима, у тромесечном бљеску продуктивности колега Николе Михајловића, Марка Нинковића, Немање Којића, Радоша Николића и моје маленкости, а под руководством проф. др Драгана Милићева. Као последица директне употребе у пракси у протеклом периоду, техника је више пута побољшавана. Извршена је и темељна експериментална евалуација технике коју је у једном делу, кроз своју мастер тезу, спровео Срђан Луковић. На томе му посебно захваљујем. Уживао сам у раду са свим споменутим колегама.

Захвалност дугујем својим родитељима, који су ме увек безусловно подржавали и пружили ми све услове да могу да се посветим докторским студијама и да их завршим, али и на подршци у животу и у току мог целокупног школовања.

На крају, захваљујем својој супрузи Снежани на подршци, стрпљењу и на сатима и данима које је провела поред мене, у радној соби, за радним столом, као у школској клупи, у последње три године заједничког живота. Њој је посвећена ова дисертација.

## Наслов докторске дисертације

Развој графичког корисничког интерфејса пословних апликација базиран на подели одговорности и објектном моделу података са UML семантиком

## Резиме

У овој дисертацији представљен је нови приступ моделовању и имплементацији графичких корисничких интерфејса информационих система и пословних апликација. Циљ дисертације је да предложи ефикасно решење за проблем комплексности у изради корисничких интерфејса. Предложено решење се темељи на неколико основних принципа: на раздвајању одговорности у процесу израде корисничких интерфејса, апстракцији, хијерархијској декомпозицији и енкапсулацији фрагмената корисничког интерфејса, као и на одговарајућој семантичкој спрези са објектним простором (подацима и пословном логиком).

Принцип раздвајања одговорности у развоју графичких корисничких интерфејса је детаљно елабориран у овој дисертацији. Најпре је извршена опширна анализа практичних проблема у програмирању корисничких интерфејса. На основу извршене анализе предложен је оквир основних одговорности који је затим емпиријски потврђен. Приступ моделовању и имплементацији корисничких интерфејса о коме је реч ослања се на предложени оквир одговорности и посебно адресира управљање структуром, понашање и приступ подацима као кључне одговорности у развоју корисничког интерфејса.

Са друге стране, потпуно ортогонално, принципи апстракције, хијерархијске декомпозиције и енкапсулације фрагмената корисничког интерфејса подржани су предложеним концептом тзв. *капсуле.* Капсула је профилисана структурирана класа из стандардног језика *UML* (енгл. *Unified Modeling Language*) са јасним интерфејсом. Капсула може да апстрахује како основне (примитивне) графичке елементе, тако и веће фрагменте корисничког интерфејса састављене од више логички и функционално повезаних графичких елемената или других капсула. Као последица тога, концепт капсуле може се применити на моделовање делова корисничког интефрејса на свим нивоима детаља, од најнижег нивоа појединачних графичких елемената до највишег нивоа архитектуре. Интерфејс капсула дефинише се уз помоћ посебног концепта *пина* (енгл. *pin*), док се функционална спрега капсула (понашање) реализује декларативним повезивањем пинова уз помоћ тзв. *жица* (енгл. *wire*). Пинови и жице преносе поруке између капсула и осигуравају стриктну енкапсулацију.

Техника представљена у овој дисертацији укључује и методу за спрезање капсула са доменским објектним простором која има одговарајући ниво апстракције и обезбеђује стриктно одвајање одговорности слоја корисничког интерфејса и слоја пословне логике.

У овој дисертацији представљена је имплементација ове технике са доменским језиком и напредном библиотеком графичких елемената за развој корисничких интерфејса. Језик и библиотека обезбеђују јасно раздвајање одговорности, решавају проблеме развоја и одржавања комплексних структура корисничког интерфејса, а при томе омогућавају непосредну и чврсту семантичку спрегу са моделом података пословне апликације на језику *UML*.

Коначно, спроведена је експериментална евалуација технике предложене у овој дисертацији у односу на неколико важних критеријума. Најпре је проверена њена практична употребљивост кроз анализу већег броја реалних софтверских пројеката и система који се успешно користе у пракси у последњих неколико година, као и перцепција и степен прихватања ове технике од стране програмера. Затим је ова техника упоређена са скупом изабраних технологија у широкој употреби у контексту оквира одговорности који

је изложен на почетку ове дисертације. На крају је извршена провера и поређење перформанси постојеће имплементације са перформансама других технологија.

## Кључне речи

Графички кориснички интерфејс, информациони системи, раздвајање одговорности, моделовање, *Unified Modeling Language (UML)*

### Title

Graphical User Interface Development for Business Applications Based on Separation of Concerns and Data Models with UML Semantics

### Abstract

This dissertation presents a novel approach to modeling and implementation of graphical user interfaces (GUI) of business applications and information systems. Its main goal is to propose a solution for efficient handling of complexity in user interface development. The solution presented in this dissertation is based on several core principles: separation of concerns in user interface implementation, abstraction, user interface decomposition and encapsulation, and adequate coupling with domain model (data and business logic).

The principle of separation of concerns has thoroughly been discussed in this dissertation. First, an extensive analysis of common issues in user interface development has been conducted. Based on that, a framework of most important aspects or concerns in user interface development has been proposed. After that, the framework has been confirmed empirically in a series of different experiments. This framework is a corner stone of the approach to user interface modeling and implementation presented in this dissertation. The emphasis is on handling structure, behavior, and data access being the most important aspects of user interface development.

In parallel, the principles of abstraction, hierarchical decomposition, and encapsulation are enforced by the concept of *capsule*. A capsule is a profiled structured class in UML (*Unified Modeling Language*), with a clear interface. A capsule abstracts both basic (primitive) user interface elements (widgets) as well as larger user interface fragments consisted of a number of functionally and logically coupled widgets and other capsules. Thus, a capsule can be applied in modeling user interface fragments on an arbitrary level of detail, starting from the lowest level of primitive graphical elements, up to the highest architectural levels. Capsule's interface is defined using a special concept of *pin*, while the functional coupling of capsules (behavior) is realized in a declarative way, using *wires* to interconnect pins. Pins and wires transfer messages between capsules and ensure strict encapsulation.

Finally, the technique presented in this dissertation ensures an adequate abstraction level of concepts used in the process of coupling capsules with the data model. This also helps to separate user interface layer from business logic and data model.

This dissertation presents a theoretical approach as well as an implementation of the approach. The implementation includes a domain-specific language and advanced library of user interface components. These two elements help programmers separate different user interface concerns, handle user interface complexity in an efficient way, and provide a semantic match between user interface and domain model specified in UML.

A thorough evaluation of the approach has been conducted. First, the practical applicability of the approach has been verified by analyzing a number of industrial systems developed with it. Second, programmers' satisfaction was analyzed. Third, the approach has been compared against a set of mainstream user interface development approaches in the context of a framework of GUI programming concerns presented in this dissertation. Finally, performance comparison has been conducted between the current implementation of the approach presented in this dissertation and several other GUI frameworks and technologies.

### Keywords

Graphical user interface, information systems, separation of concerns, modeling, *Unified Modeling Language (UML)*

**Кратак садржај**

Део 1: Увод ... 1
Увод ... 3
Дефиниција проблема ... 7
Преглед постојећих решења ... 9
Део 2: Принцип раздвајања одговорности ... 17
Раздвајање одговорности у развоју корисничких интерфејса ... 19
Поступак идентификације одговорности ... 27
Ретроспектива технологија за развој корисничких интерфејса ... 41
Део 3: Принцип хијерархијске декомпозиције ... 51
Основни концепти решења ... 53
Део 4: Имплементација ... 67
Језик за моделовање корисничких интерфејса ... 69
Библиотека за развој корисничких интерфејса ... 89
Имплементација и помоћни алати ... 145
Део 5: Анализа предложеног решења ... 151
Практична употребљивост и степен прихватања ... 153
Аналитичко поређење са другим технологијама ... 159
Евалуација перформанси ... 165
Део 6: Закључак ... 169
Закључак ... 171
Будуће активности и правци развоја и истраживања ... 173
Референце ... 175
Прилози ... 179
Профил језика UML за моделовање корисничких интерфејса ... 181
Биографија ... 189
Кратка биографија ... 191
Пројекти и професионалне активности ... 193
Списак објављених научних радова и учешћа на међународним конференцијама ... 195
Изјаве ... 197
Изјава о ауторству ... 199
Изјава о истоветности штампане и електронске верзије докторског рада ... 201
Изјава о коришћењу ... 203

## Садржај

Захвалнице........1
Наслов докторске дисертације........3
Резиме........3
Кључне речи........4
Title........5
Abstract........5
Keywords........5
Кратак садржај........7
Садржај........9
Део 1: Увод........1
Увод........3
Предмет и циљ рада........3
Структура и садржај рада........4
Дефиниција проблема........7
Преглед постојећих решења........9
Раздвајање одговорности у развоју корисничких интерфејса........9
Постојећи предлози начина раздвајања одговорности........9
Распоред........12
Валидација........12
Форматирање података........13
Стилизација........13
Употреба модела у развоју корисничких интерфејса........14
Део 2: Принцип раздвајања одговорности........17
Раздвајање одговорности у развоју корисничких интерфејса........19
Приступ подацима........21
Структура........21
Распоред........21
Понашање........22
Форматирање података........22
Валидација........23
Стилизација........23
Остале одговорности........24
Поступак идентификације одговорности........27
Опис апликативног програмског интерфејса........27
Опис итеративног поступка........28
Анализа апликативног програмског интерфејса........31
Студија коришћења........32
Анализа еволуције корисничког интерфејса........33

Поређење резултата ........ 34
Анкета ........ 35
Ретроспектива технологија за развој корисничких интерфејса ........ 41
Технологије оријентисане ка формуларима ........ 42
Технологије базиране на шаблонима ........ 42
Технологије базиране на принципима објектне оријентације ........ 43
Хибридне технологије ........ 43
Технологије базиране на декларативним језицима ........ 44
Технологије базиране на моделима ........ 45
Генерички приступи ........ 46
Критика, трендови и изазови ........ 47
Део 3: Принцип хијерархијске декомпозиције ........ 51
Основни концепти решења ........ 53
Идеја ........ 54
Структура ........ 57
Понашање ........ 60
Приступ подацима ........ 62
Део 4: Имплементација ........ 67
Језик за моделовање корисничких интерфејса ........ 69
Пример употребе ........ 69
Спецификација синтаксе језика *CAPWISE* ........ 83
Библиотека за развој корисничких интерфејса ........ 89
Основни пакети ........ 90
Пакет Construction ........ 90
Пакет Components ........ 92
Пакет Layout ........ 93
Пакет ElementComponents ........ 95
Пакет Style ........ 95
Пакет NonVisualComponents ........ 96
Начини распоређивања графичких елемената / таб елемент ........ 99
Опис нових графичких елемената ........ 99
Изворни код корисничког интерфејса ........ 100
Начини распоређивања графичких елемената / дек елемент ........ 101
Опис нових графичких елемената ........ 102
Изворни код корисничког интерфејса ........ 102
Менији ........ 104
Опис нових графичких елемената ........ 104
Изворни код корисничког интерфејса ........ 105
Чаробњак (енгл. wizard) ........ 106

Опис нових графичких елемената ........ 107
Изворни код корисничког интерфејса ........ 107
Графички елемент за приказ, унос и измену података ........ 109
Опис нових графичких елемената ........ 110
Модел на језику UML ........ 112
Изворни код корисничког интерфејса ........ 112
Форма за снимање података о особи ........ 118
Модел на језику UML ........ 118
Изворни код пословне логике ........ 119
Изворни код корисничког интерфејса ........ 119
Табела са могућностима измене детаља ........ 120
Модел на језику UML ........ 121
Изворни код корисничког интерфејса ........ 121
Креирање, читање, измена и брисање ........ 122
Опис нових графичких елемената ........ 123
Модел на језику UML ........ 123
Изворни код пословне логике ........ 124
Изворни код корисничког интерфејса ........ 124
Измена веза између ентитета ........ 125
Модел на језику UML ........ 126
Изворни код корисничког интерфејса ........ 126
Комбинација претходних примера ........ 127
Модел на језику UML ........ 128
Изворни код пословне логике ........ 129
Изворни код корисничког интерфејса ........ 129
Претрага особа ........ 130
Опис нових графичких елемената ........ 131
Модел на језику UML ........ 131
Изворни код пословне логике ........ 132
Изворни код корисничког интерфејса ........ 133
Полиморфни панел ........ 135
Модел на језику UML ........ 136
Изворни код корисничког интерфејса ........ 136
Динамички панел ........ 137
Опис нових графичких елемената ........ 138
Модел на језику UML ........ 138
Изворни код корисничког интерфејса ........ 139
Галерија фотографија ........ 140
Опис нових графичких елемената ........ 140
Модел на језику UML ........ 141
Изворни код пословне логике ........ 141

Изворни код корисничког интерфејса ... 142
Имплементација и помоћни алати ... 145
Помоћни алати ... 147
Део 5: Анализа предложеног решења ... 151
Практична употребљивост и степен прихватања ... 153
Аналитичко поређење са другим технологијама ... 159
Евалуација перформанси ... 165
Део 6: Закључак ... 169
Закључак ... 171
Будуће активности и правци развоја и истраживања ... 173
Референце ... 175
Прилози ... 179
Профил језика UML за моделовање корисничких интерфејса ... 181
Оквир ... 181
Капсула, интерна структура и конструктор ... 181
Додатна ограничења ... 183
Пинови ... 183
Додатна ограничења ... 185
Додатне операције ... 186
Жице ... 186
Додатне операције ... 187
Додатна ограничења ... 187
Биографија ... 189
Кратка биографија ... 191
Пројекти и професионалне активности ... 193
Списак објављених научних радова и учешћа на међународним конференцијама ... 195
Изјаве ... 197
Изјава о ауторству ... 199
Изјава о истоветности штампане и електронске верзије докторског рада ... 201
Изјава о коришћењу ... 203

# Део 1: Увод

# *Увод*

У овом поглављу биће укратко представљена тема и главни циљеви ове дисертације, као и структура и садржај самог документа.

## Предмет и циљ рада

Један од основних очекиваних доприноса докторске дисертације јесте идентификација најважнијих и најзаступљенијих одговорности (енгл. *concerns*) програмера у процесу имплементације графичког корисничког интерфејса (енгл. *graphical user interface - GUI*) пословних апликација и информационих система. Супротно досадашњим покушајима идентификације који су били засновани на интуицији, циљ ове дисертације, између осталог, јесте да спроведе систематичну емпиријску анализу аспеката, односно одговорности које треба адресирати приликом програмирања корисничког интерфејса. Кроз оквир (енгл. *framework*) дефинисан идентификованим одговорностима, циљ је и да се анализирају постојећи алати и открију могућности њиховог унапређења. Према томе, основна полазна претпоставка јесте да је могуће идентификовати и раздвојити аспекте, односно одговорности инжењера приликом развоја корисничког интерфејса, тако да финални скуп одговорности не буде превише широк, јер би тада захтевао напор да се усвоји и запамти, али ни превише узан јер се у том случају не би могли задовољити сви захтеви у имплементацији корисничких интерфејса и учинити напредак у решавању њихове комплексности. Идентификовање одговорности има за циљ и успостављање заједничког и препознатљивог вокабулара који би олакшао и унапредио комуникацију између софтверских инжењера који се баве развојем корисничких интерфејса. Додатно, прецизно идентификовање одговорности отворило би простор за унапређење самог процеса развоја корисничких интерфејса јер би омогућило паралелизацију активности током тог процеса.

Као потврда класификације одговорности очекује се и анализа технологија за развој корисничких интерфејса кроз призму претходно идентификованих одговорности. Анализа постојећих технологија треба да обухвати како комерцијалне технологије у широкој употреби у софтверској индустрији, тако и технологије развијене у академским институцијама у последње две деценије. Од анализе се очекује и класификација свих

анализираних технологија, као и евентуална критика и предвиђање будућих праваца њиховог развоја.

Раздвајање одговорности свакако олакшава решавање проблема комплексности у програмирању корисничких интерфејса. Међутим, одређене одговорности као што су структура, понашање и приступ подацима и даље представљају велики проблем, односно инхерентно поседују велику комплексност. Стога је од суштинске важности адекватно адресирати ове аспекте програмирања корисничких интерфејса и то је још један од најважнијих циљева ове дисертације. Очекује се да решење представљено у овој дисертацији буде техника која омогућава да се комплексне структуре, какви су кориснички интерфејси, могу декомпоновати на начин којим би се омогућио поглед на интерну организацију корисничког интерфејса на жељеном нивоу детаља, што подразумева све нивое, од најдетаљнијих до најапстрактнијих архитектуралних погледа.

Коначно, на основу свега реченог, очекује се предлог, пројекат и имплементација нове технике која уважава идентификоване одговорности, омогућава хијерархијску декомпозицију фрагмената корисничког интерфејса и нуди напредну библиотеку елемената за развој корисничких интерфејса. Аспекти развоја корисничких интерфејса који захтевају посебну пажњу и постављају највеће проблеме пред програмере корисничких интерфејса треба да буду посебно решавани. То су структура, понашање и приступ подацима. Основни захтев је, дакле, да техника и библиотека обезбеде јасно раздвајање одговорности, ефикасно управљање структуром и понашањем и чврсту семантичку спрегу са моделом података пословне апликације на језику *UML*[1] (енгл. *Unified Modeling Language*). Идеја је да библиотека унапреди развој корисничких интерфејса за сваку од идентификованих одговорности. Од ове технологије се очекује да омогући квалитетнији и бржи развој и олакша одржавање самог финалног производа.

### Структура и садржај рада

Ова дисертација састоји се из шест главних целина, односно делова. У уводном делу дисертације представљени су главна тема и циљеви дисертације, дефиниција проблема и преглед постојећих решења. Дат је преглед технологија за развој корисничких интерфејса како из угла раздвајања одговорности при програмирању корисничких интерфејса, тако и из угла развоја корисничких интерфејса уз помоћ модела.

Други део дисертације фокусиран је на принцип раздвајања одговорности у развоју корисничких интерфејса. У првом поглављу тог дела даје се преглед и објашњавају основне одговорности у развоју корисничких интерфејса. У наредном поглављу, кроз неколико експеримената, детаљно се описује процес идентификације одговорности описаних у првом поглављу. Треће поглавље даје историјски преглед и класификацију великог броја технологија за развој корисничких интерфејса кроз призму оквира кога чине идентификоване одговорности.

Трећи део дисертације обрађује принцип хијерархијске декомпозиције интерне структуре корисничких интерфејса. У првом поглављу овог дела неформално је изложена основна идеја технике за развој корисничких интерфејса кроз примере који објашњавају основне концепте технике као што су капсула, пин и жица. Ова техника првенствено адресира

1 http://www.uml.org/

најкомплексније одговорности у изградњи корисничких интерфејса, и то управљање структуром, понашање и приступ подацима.

У четвртом делу дисертације описана је имплементација предложене технике. Најпре је изложен доменски специфичан текстуални језик за програмирање корисничких интерфејса назван *CAPWISE.* Он је представљен кроз у уводном примеру употребе, а затим формално дефинисан у виду граматике у БНФ форми. Затим се детаљно и сликовито описује имплементирани апликациони програмски интерфејс који је претеча споменутог језика. Графички елементи овог интерфејса представљају поменуту библиотеку за изградњу корисничких интерфејса предложеном техником и приказани су кроз примере, моделе, изворни код и слике екрана. На крају се приказују и неки одабрани детаљи саме имплементације ове библиотеке и описују се помоћни алати који се могу користити у развоју.

У завршном делу дисертације описана техника се анализира и евалуира како би се поткрепиле тврдње о њеној примењивости, степену прихватања од стране програмера, успешности и перформансама.

Дисертација се завршава закључцима изнетим у последњем делу. У прилогу ове дисертације техника моделовања корисничких интерфејса описана у овој дисертацији формално се дефинише као профил језика *UML*.

# Дефиниција проблема

Развој графичких корисничких интерфејса укључује више активности и подразумева изградњу врло сложених и динамичних структура. Постоји мноштво научноистраживачких радова и извештаја из праксе који износе резултате и говоре о утрошеном времену, количини програмског кода и броју графичких елемената у пословним апликацијама и информационим системима данашњице [11, 18, 34, 51]. Прелиминарни резултати аутора ове дисертације показали су да данашње апликације садрже и преко 30.000 графичких елемената као што су панели, дугмад, иконице, лабеле, листе, табеле, менији итд, док се на изградњу корисничког интерфејса троши често и више од две трећине укупног времена развоја информационих система. Графички елементи данашњих апликација су често међусобно повезани са преко 10.000 функционалних или структурних веза које се манифестују као процедуре за обраду догађаја (енгл. *event handlers*) на једном графичком елементу које извршавају акције на другим елементима. Поред тога, графички елементи су повезани додатним везама у хијерархијску структуру која између осталог дефинише распоред елемената на екрану. У оваквој структури нису јасно и експлицитно представљене зависности, односи интеракције елемената, што отежава разумевање конструкције и начина функционисања корисничког интерфејса. Дакле, део проблема у развоју корисничких интерфејса представља велика сложеност структуре основних елемената графичког корисничког интерфејса.

Поред велике комплексности структуре корисничког интерфејса, постоји изражена комплексност и на нивоу појединачног елемента односно чвора те структуре. Прво, данашњи алати и апликативни програмски интерфејси нуде на десетине различитих типова графичких елемената. Друго, сви типови елемената захтевају мноштво подешавања приликом њихове уградње у кориснички интерфејс. На пример, дугме мора да се позиционира на екрану, да му се специфицира величина, да се дефинише прикладан натпис и опис одговарајуће акције, да се подеси боја, градијент, дебљина оквира, маргине, да се дефинише понашање (нпр. кад се стекну одређени услови, дугме треба да буде затамњено или недоступно) и на крају, када се притисне, дугме треба да покрене одређену акцију у апликацији. Сложеност ових подешавања слична је или иста за већину графичких елемената. Данашње технологије, међутим, нису направљене у складу са потребама подешавања тих различитих аспеката графичких елемената приликом њиховог коришћења.

Напротив, технологије за развој корисничких интерфејса су врло често пројектоване и имплементиране тако да буду у складу са потребама оних који их развијају и имплементирају, пре него њихових корисника – програмера [49].

Укратко, тема ове дисертације и основни проблем који решава јесте комплексност како интерне репрезентације односно саме графовске структуре графичких корисничких интерфејса, тако и самих градивних елемената ове структуре. Техника која претендује да се избори са проблемом комплексности у развоју графичких корисничких интерфејса треба да реши или ублажи сваки од ова два дела комплексности без увођења новог нивоа случајне комплексности (енгл. *accidental complexity*) у рад програмера корисничких интерфејса.

# *Преглед постојећих решења*

Као што је већ речено, фокус ове дисертације је на раздвајању одговорности (енгл. *separation of concerns*) као једном од основних принципа софтверског инжењерства и на примени моделовања са циљем апстракције, енкапсулације и хијерархијске декомпозиције фрагмената корисничког интерфејса. У складу са тим, у овом поглављу биће анализиране друге технологије и приступи и како они адресирају раздвајање одговорности и примену модела у развоју графичких корисничких интерфејса.

## Раздвајање одговорности у развоју корисничких интерфејса

У првом делу овог поглавља биће продискутовани постојећи напори да се оствари идеал раздвајања одговорности у домену имплементације корисничких интерфејса. Управо они представљају полазну тачку нашег истраживања. У другом делу овог поглавља биће представљена истраживања која се баве одговорностима у развоју али појединачно, фокусирајући се на тачно један аспект, односно одговорност.

### *Постојећи предлози начина раздвајања одговорности*

Сви начини аспектизације развоја корисничких интерфејса који ће бити споменути у пасусима који следе приказани су у табели 1, у свакој колони по један. Сваки ред табеле одговара једном аспекту из класификације одговорности предложене у овој дисертацији, док су бојама (нијансама сивог) и малим словима (а, б, в, итд.) означене одговорности идентификоване у другим студијама и њихово мапирање на одговорности из класификације предложене у овој дисертацији.

Такозвана *молекуларна архитектура* [21] предложена је са циљем да реши проблем флексибилности хијерархије наслеђивања у библиотекама графичких елемената. Основна идеја ове архитектуре јесте да карактеристике графичких елемената нису статички наметнуте њиховом класом, већ су резултат комбинације основних (атомских) и лако замењивих компонената од којих су састављени сви графички елементи. У овом смислу, класични графички елементи представљају молекуле састављене од базичних атома. Оно што је овде од интереса јесте да се спомиње коначан скуп категорија којима припадају атомски елементи. То су следеће категорије (видети табелу 1): а) понашање – обухвата превлачење (енгл. *drag and drop*), обележавање елемената, измену и обележавање текста, избор елемената у листама итд, б) графичке особине – изглед, фонтови, боје, маргине итд,

в) видљивост – слике, графика, симболи, г) повезивање – обухвата везу делова корисничког интерфејса са функционалним делом апликације, као и између себе и коначно д) груписање – обухвата формирање хијерархијске структуре графичких елемената са циљем њиховог визуелног распоређивања на екрану.

У интересантном раду [46] група аутора истражује могућности примене идеје пројектних образаца (енгл. *design patterns*) у развоју софтвера и корисничких интерфејса базираном на моделима. Аутори препознају три модела корисничких интерфејса. Модел дијалога (а) (енгл. *dialog model*) дефинише навигациону структуру и технике интеракције. Модел презентације (енгл. *presentation model*) (б) специфицира мапирање пословних задатака (енгл. *task*) и објеката (енгл. *business objects*) на графичке објекте за интеракцију као што су листе, стабла, дугмад, итд. На крају, модел распореда (енгл. *layout model*) (в) дефинише распоред графичких елемената као и фонтове, боје, и сл. (видети табелу 1).

Одговорности програмера корисничког интерфејса детаљно су анализиране у једној докторској дисертацији [11]. Ауторка те дисертације најпре анализира постојеће класификације одговорности, са посебним освртом на образац изградње интерфејса *model-view-controller - MVC*[2]. Она затим предлаже модуларизацију програмског кода корисничких интерфејса из перспективе програмера у виду а) презентационог аспекта (енгл. *presentation logic concern*), б) апликационог аспекта (енгл. *application logic concern*) и в) аспекта повезивања (енгл. *connection logic concern*) – видети табелу 1. Иако може да доведе у заблуду својим називом, апликациони аспект не представља саму апликацију нити њен слој пословне логике или неки други слој, већ је управо саставни део корисничког интерфејса. Он обухвата двосмерну логику која регулише како се акције у апликацији побуђују из корисничког интерфејса, али и обрнуто. Другим речима, у овом слоју региструју се апликационе акције (које се побуђују из корисничког интерфејса) и апликациони догађаји (који побуђују акције у корисничком интерфејсу). Презентациони аспект састоји се из визуелног дела и дела који обухвата понашање. Визуелни део обухвата дефинисање графичких елемената, њихових атрибута и распореда, као и стилска подешавања. Део за понашање обухвата регистрацију догађаја, али и акција у корисничком интерфејсу. Акције представљају понашање које мења сам кориснички интерфејс. Догађаји побуђују акције и у самом корисничком интерфејсу, али и у апликацији. Дакле, са једне стране постоје догађаји и акције регистроване у делу за понашање презентационог аспекта. Са друге стране постоје апликациони догађаји и акције регистроване у апликационом аспекту. Повезивање једних и других догађаја и акција обухваћено је аспектом повезивања. Према тврдњама ауторке, оваквим раздвајањем одговорности могуће је постићи потпуно независно и изоловано дефинисање сваког аспеката.

Још један истраживачки рад обрађује тематику раздвајања одговорности у развоју корисничких интерфејса [15]. Аутор тог рада препознаје три аспекта: а) аспект понашања, б) аспект презентације и в) аспект изгледа (табела 1). Аспект понашања обухвата навигацију, манипулисање подацима и помоћ корисницима. Навигација обухвата све операције које омогућавају кориснику навигацију кроз апликацију и сваки утицај на промену корисничког интерфејса без перзистентних последица. Примери понашања су навигација кроз странице апликације, сортирање података, прегледање страница табеле са пагинацијом, селекција ставки из менија, и сл. Манипулација података подразумева перзистентне последице и најчешће обухвата експлицитне операције снимања као и валидацију улазних података. Помоћ кориснику подразумева пружање статусних информација, валидационих порука, порука о грешкама, као и помоћ у смислу упутства за употребу апликације. Аспект презентације обрађује избор графичких елемената за

[2] Model-view-controller: http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

презентовање података и структуре корисничког интерфејса без обазирања на распоред или стилизацију тих елемената. Овај аспект се дефинише помоћу апстрактног и конкретног модела: најпре се бирају елементи концептуално, нпр. елемент за избор једне од опција, а затим се апстрактни елементи конкретизују, нпр. бира се радио дугме (енгл. *radio button*). На крају, аспект изгледа подразумева распоред графичких елемената, стилизацију и форматирање података.

У једном скорашњем раду на тему развоја корисничких интерфејса базираном на моделима (енгл. *model-based user interface development - MBUID*) [23], група аутора даје интересантан историјски преглед последње четири генерације технологија, од 1990. године до данас. Према њиховом мишљењу, основни модели/аспекти су а) модел задатака (енгл. *task model*), б) модел дијалога и в) модел презентације (табела 1). Модел задатака обухвата приступ подацима, тако што за сваки задатак из модела задатака дефинише графичке елементе и доменске апликационе објекте са којима се манипулише. Модел дијалога дефинише понашање корисничког интерфејса. Модел презентације обухвата избор, распоред и стилизацију графичких елемената.

Према свему до сада описаном у овом поглављу, јасно је да постоје бројне класификације и модели раздвајања одговорности у развоју корисничких интерфејса. Међутим, изгледа да не постоје конкретни научни докази који говоре о томе како су описане класификације добијене. Извесно је, дакле, да су оне базиране на искуству и неформалним инспекцијама изворног кода. Према мишљењу аутора ове тезе, ипак је важно пружити чвршће доказе ако се жели постићи завидан ниво разумевања у овој области. Са друге стране, приказани радови пружили су довољно почетног материјала за покретање систематске студије са циљем да се пруже поуздане информације о одговорностима у развоју корисничких интерфејса.

| | Lecolinet [21] | Sinnig et al. [46] | Goderis [11] | Holwerda [15] | Meixner et al. [23] |
|---|---|---|---|---|---|
| структура | - | | | | |
| распоред | | | | | |
| понашање | | | | | |
| приступ подацима | | | | | |
| форматирање података | ? | ? | | | ? |
| стилизација | | | | | |
| валидација | | | | | ? |
| легенда | а) | а) | а) | а) | а) |
| | б) | б) | б) | б) | б) |
| | в) | в) | в) | в) | в) |
| | г) | - | - | - | |
| | д) | - | - | - | |

Табела 1. Аспекти развоја корисничких интерфејса у постојећим приступима.

У наредним секцијама, биће анализирани приступи који адресирају одговорности развоја корисничких интерфејса појединачно.

### Распоред

Вероватно најпрепознатљивији аспект развоја корисничких интерфејса јесте распоред (енгл. *layout*). Сваки програмер који је икада имао прилику да развија апликацију која има кориснички интерфејс зна шта се под тим подразумева. Због тога није чудно што постоји велики број радова који разматрају проблематику распоређивања графичких елемената на екрану. Иако се већина приступа базира на ограничењима (енгл. *constraints*), још увек не постоји стандардан начин спецификације ограничења [22]. Овде ће бити споменуто неколико новијих научних радова из ове области.

Група аутора на челу са Џејкобсом (*Jacobs*) приказала је интересантну технику адаптивног распоређивања текста и графике у документима на вебу [16]. Техника решава проблем прилагођавања веб докумената различитим величинама екрана и значајно смањује количину мануелног посла у сређивању докумената у издаваштву. У великој мери, ова техника се ослања на шаблоне (енгл. *templates*) и ограничења.

Технике које се ослањају на ограничења су ниског нивоа, захтевају завидан ниво формалног знања (линеарно програмирање[3]) што их чини незграпним и тешким за разумевање [22]. Као последица тога, предложена је техника високог нивоа *Auckland Layout Model – ALM* која употребљава природне апстракције из домена корисничких интерфејса [22]. Њен циљ је да олакша прављење модела распоређивања. У понуди су апстракције правоугаоне области са минималном, максималном и жељеном величином, као и апстракције редова и колона. Техника је применљива на корисничке интерфејсе за екране променљиве величине, али и на штампане документе.

Као што је већ споменуто, један од основних проблема у распоређивању елемената јесте прилагодљивост интерфејса величини екрана. Додатни проблем су перформансе, због тога што претерана употреба ограничења захтева процесорско време и не може се извршити довољно брзо за корисника који мења величину прозора померајући миша у реалном времену. Дакле, решење треба да буде довољно једноставно за програмера и не превише захтевно за процесирање. Постоји један приступ који предлаже брз алгоритам који је уједно и лак за употребу [42]. Он се ослања на нову невидљиву компоненту корисничког интерфејса која служи само у сврху распоређивања названу „уметак“ (енгл. *spacer*).

### Валидација

Валидација података је такође препознатљив аспект развоја корисничких интерфејса. У једном скоријем раду објављеном на ову тему [12], говори се о интеграцији овог аспекта у процес развоја. Реч је о декларативном приступу валидацији улазних података који обухвата следеће: проверу формата улазних података (енгл. *well formedness*), рестрикције у виду операција на пољима модела података (енгл. *data invariants*), рестрикције у виду операција на елементима интерфејса (енгл. *input assertions*), предикатске провере акција (енгл. *action assertions*) и приказ валидационих порука кориснику.

Интересантно је споменути још један приступ валидацији веб образаца назван *PowerForms* [3]. У питању је декларативни, доменско-специфични језик (енгл. *domain-specific language*) високог нивоа за инкременталну валидацију уноса. Језик се састоји од регуларних израза које програмер уграђује у код веб странице на језику *HTML* и који се потом аутоматски преводе у комбинацију стандардног кода на језицима *HTML* и *JavaScript*. Аутори нуде

[3] Линеарно програмирање је техника оптимизације која минимизира или максимизира вредност линеарне функције *n* променљивих у складу са ограничењима постављеним у виду скупа линеарних неједначина.

богату синтаксу на језику *XML* за регуларне изразе као и за изражавање међусобне зависности улазних контрола које се валидирају. Корисник добија валидационе поруке континуирано, а улазни образац не може да се сними све док сви изрази не буду задовољени.

Коначно, класичан приступ валидацији података са бинарним исходом је проширен и унапређен у раду [41]. Поред дефинитивно валидног и дефинитивно невалидног уноса, уводи се и сумњив унос (енгл. *questionable*). На пример, за унос имена особе, чудан микс малих и великих слова „ЕТФ Београд" може бити сумњив. У том случају, апликација би могла да тражи од корисника да провери сумњив унос. У општем случају, сумњив унос захтева додатну проверу човека или програма. Додатно, ако постоји више од једног прихватљивог формата за неки податак, класични приступи валидацији оставиће податке у различитим форматима, док би у идеалном случају, подаци ипак били снимљени у једном, најпогоднијем формату за апликацију. У овом раду представљена је техника, укључујући и развојно окружење, за препознавање сумњивих података и снимање података у конзистентном формату.

### *Форматирање података*

Форматирање података није обрађено у научним радовима у већој мери. Овај аспект развоја корисничких интерфејса обрађен је у комерцијалним технологија. На пример, у библиотеци језика *Java*, класа *java.text.Format* представља основну апстрактну класу за форматирање бројева, датума, времена и порука. У технологији *JSP* постоје посебни тагови за форматирање података, као нпр. *<fmt:formatDate />*, *<fmt:formatNumber />*, итд. У другим програмским језицима и библиотекама ситуација је слична. Међутим, овакав приступ форматирању података може се у многоме унапредити. Први проблем јесте тај што се свако појављивање податка датума у апликацији мора форматирати на месту појављивања. На пример, на сваком месту у свакој страници мора се урадити следеће:

```
<fmt:formatDate value="${dateToBeFormatted}" dateStyle="short"/>.
```

Могуће је то учинити и негде пре у програму користећи подкласу поменуте класе за форматирање на следећи начин:

```
DateFormat.getDateInstance(DateFormat.SHORT).format(dateToBeFormatted);
```

Другим речима, немогуће је дефинисати формат датума на једном месту и означити за који део корисничког интерфејса (за које странице, или за које секције једне странице, или сл.) важи та дефиниција формата датума.

Други проблем јесте тај што поменуте библиотеке адресирају искључиво форматирање инстанци примитивних типова, као што су датуми, времена, бројеви, новчани износи, итд. Форматирање приказа инстанци сложених типова, као што су инстанце класа (нпр. особа, предузеће, рачун, итд.), није могуће остварити. Најчешће се програмски код који обрађује форматирање инстанци сложених типова налази расут по пројектним артефактима, а не сређен и локализован на једном месту. Изузетак је апликациони програмски интерфејс описан у књизи о развоју софтвера базираном на извршивим моделима на језику *UML* [26] чији су неки елементи преузети и унапређени у овој дисертацији.

### *Стилизација*

Начини за специфицирање визуелног стила апликације су добро подржани у апликационим програмским интерфејсима и практично нема академских извора који обрађују ову тему. Када је у питању развој веб сајтова и пословних веб апликација, без обзира на изабрану

технологију, превасходно се користи технологија *CSS* који је *de facto* стандардан начин за стилизацију веб докумената.

Остале одговорности из класификације одговорности предложене у овој дисертацији – структура, понашање и приступ подацима – према сазнању аутора ове дисертације такође нису појединачно били у фокусу истраживача.

## Употреба модела у развоју корисничких интерфејса

У претходном одељку фокус је био на разматрању принципа раздвајања одговорности у технологијама за развој корисничких интерфејса. У овом поглављу биће анализирани приступи из угла примене моделовања, односно употребе модела у развоју корисничких интерфејса.

Технологија *SUIDT* [2] ослања се врло интензивно на употребу модела у развоју корисничких интерфејса. Прецизније, у њој се користи концептуални модел података с једне стране, и апстрактни и конкретни модели задатака (енгл. *task model*) са друге. Да би направио кориснички интерфејс, програмер прво мора да га осмисли, затим да уложи значајан напор да преточи идеју у апстрактне и конкретне моделе задатака и на крају да генерише кориснички интерфејс из модела. На основу примера приложеног у раду изгледа да би било лакше да се кориснички интерфејс директно програмирао без употребе модела, јер се у примеру види колико компликовани могу да буду модели чак и за једну сасвим малу интерактивну апликацију.

*JUST-UI* [30] је техника прикупљања захтева и моделовања графичких корисничких интерфејса. Моделовање корисничких интерфејса се базира на концептуалном моделу података и скупу грубљих пројектних образаца корисничких интерфејса. Сваки од ових образаца описан је апстрактним објектима интеракције (енгл. *abstract interaction objects, AIO*). У фази генерисања корисничког интерфејса апстрактни објекти трансформишу се у конкретне објекте (енгл. *concrete interaction objects, CIO*) на основу одређених правила и табела трансформације. Конкретни објекти чине финални кориснички интерфејс. Техника *JUST-UI* има висок ниво апстракције (захваљујући пројектним обрасцима презентације и доменском моделу података), проширивост (захваљујући могућности прављења нових апстрактних модела) и висок ниво независности од циљне платформе. Међутим, није јасно како се тачно пројектни обрасци корисничког интерфејса мапирају у апстрактне објекте интеракције, нити како се комуницира са доменским објектним простором.

Апстрактни и конкретни модели корисничког интерфејса (енгл. *abstract user interface, AUI, concrete user interface, CUI*) заједно са моделима задатака користе се у још једном приступу [40]. Основни циљ у том приступу јесте да се адекватно обухвати дизајн, прављење прототипа и развој корисничких интерфејса у контексту разноликости уређаја и екрана (енгл. *multi-target, multi-device*). Главни допринос ове технике је платформска независност. Међутим, опет је остала нејасна веза између модела задатака и апстрактних модела корисничког интерфејса. Поврх тога, изгледа да је потребно уложити значајан напор да се ова два типа модела направе.

*UMLi* [9] је профил језика *UML* за моделовање корисничких интерфејса. Он нуди скуп стереотипа за моделовање апстрактних категорија графичких елемената као што су улазне контроле (енгл. *inputters*), едитори и контејнери, и тиме спречава прерано везивање за графичке елементе циљне платформе. У структурном делу моделују се стандардне релације садржавања графичких елемената. Међутим, понашање и интеракција елемената моделују се помоћу дијаграма активности. Због тога чак и најједноставније интеракције графичких елемената захтевају значајнији напор у фази моделовања. Такође није разјашњено ни како се апстрактне категорије елемената имплементирају.

*EUIS* (*Enterprise User Interface Specification*) [29] је још један пример доменски специфичног језика дефинисаног као профил на језику *UML* који се користи за спецификацију распореда и понашања графичких елемената пословних апликација. Овај приступ базира се на оквиру кога чине концепти и принципи за изградњу корисничких интерфејса оријентисаних ка формуларима из (уз помоћ стереотипа) анотираних доменских модела. Подржана је полу-генеричка навигација кроз странице као и декларативно спајање акција у корисничком интерфејсу са позадинском логиком. Комплетан изворни код корисничког интерфејса се генерише на основу анотираног модела класа на језику *UML*. Иако извођење корисничког интерфејса из модела класа има својих предности као што су униформност и комплетност, ипак остаје ограничење у смислу флексибилности за специфичне захтеве за кориснички интерфејс који се не могу једнозначно мапирати на класе из модела.

Још један пример употребе језика *UML* у развоју пословних апликација и корисничких интерфејса је и техника која се зове *ZOOM* (*Z-Based Object Oriented Modeling*) [13]. Ова техника обједињује и међусобно интегрише моделе структуре, понашања и презентације у један свеобухватан приступ развоју пословних апликација. Ова техника има формалну семантику, обезбеђује конзистентност споменутих модела и нуди нотацију за дизајн корисничких интерфејса. Међутим, библиотека графичких елемената је релативно једноставна и сиромашна. Понашање корисничког интерфејса је измештено у посебан слој између корисничког интерфејса и пословне логике, што има неколико негативних ефеката. Графички елементи постају „анемични“, повећава се концептуални простор између корисничког интерфејса и пословне логике и на крају, тривијалне међусобне интеракције између графичких елемената захтевају превише моделовања јер се морају моделовати у посебном слоју. Са друге стране, доста добро су раздвојене одговорности, а задовољавајући је ниво и платформске независности.

На крају, треба споменути и доменски специфичан језик за моделовање веб апликација назван *WebML* [4]. Део овог језика предвиђен за моделовање корисничких интерфејса нуди концепте за моделовање организације и структуре корисничког интерфејса на нивоу области и страница (енгл. *area, page*, респективно). Структура странице састоји се из графичких елемената који приказују или мењају делове доменског објектног простора или извршавају друге специфичне задатке. Постоје концепти који служе за моделовање навигације између страница и интеракцију између графичких елемената, укључујући прослеђивање аргумената. Библиотека уграђених графичких елемената садржи елементе који обављају генеричке операције над објектима из домена (на пример дохватање објекта по кључу, приказ листе објеката, мењање вредности атрибута објеката итд.). Библиотека се може проширити графичким елементима специфичним за циљни домен апликације. Овакви елементи се праве на платформски специфичан начин, али се интегришу у окружење врло формално и контролисано. На овај начин, приступ *WebML* постиже контролисану флексибилност: језик за моделовање дефинише семантику интерфејса графичких елемената и њихове интеракције, док је сама имплементација специфична за платформу (на пример, то може бити језик *JavaScript* на клијентској и *JEE* на серверској страни). Овакав начин размишљања је у потпуности сагласан са приступом представљеним у овој дисертацији, о чему ће бити више речи у наставку. Са друге стране, постоји један потенцијални проблем са техником *WebML*. Пословна логика није издвојена у посебан слој нити је уграђена у модел домена, већ је смештена у невидљивим елементима слоја графичког корисничког интерфејса. Овим се одваја пословна логика од модела домена и меша се са логиком корисничког интерфејса. Још једна битна разлика ове технике и технике представљене у овој дисертацији јесте та што прва има ограничен скуп концепата за моделовање различитих нивоа архитектуре корисничког интерфејса (област, страница и

графички елемент), док друга садржи концепт капсуле који се може примењивати рекурзивно на произвољном броју нивоа архитектуре. Ова разлика може да дође до изражаја приликом развоја корисничких интерфејса у десктоп стилу који немају изражену навигацију базирану на страницама и у сваком тренутку садрже велики број графичких елемената (еквивалентан броју свих графичких елемената свих страница корисничког интерфејса базираног на страницама).

Све технике и приступи споменути у овом поглављу имају добре концепте и јаку подршку али само за неке аспекте развоја корисничких интерфејса. Комплетна и свеобухватна техника која адекватно адресира сваку одговорност корисничког интерфејса још увек није предложена. Чак се и подршка за најосновније одговорности као што су структура, понашање и приступ подацима показује као врло проблематична, поготово када су у питању системи великих размера (енгл. *large-scale*). То је управо главна тема и предмет ове дисертације.

# Део 2: Принцип раздвајања одговорности

# *Раздвајање одговорности у развоју корисничких интерфејса*

Са циљем да се олакша и унапреди разумљивост и еволуција софтверских система, као и потенцијал за поновну употребу њихових делова, системи се морају правити као композиција јасно раздвојених и одвојено специфицираних аспеката од интереса [19]. Међутим, очигледно је да апликациони програмски интерфејси (енгл. *application programming interface, API*) за развој корисничких интерфејса које можемо пронаћи у широкој употреби нису у складу са основним одговорностима програмера корисничких интерфејса. Слика 1 илуструје како и за најмањи део корисничког интерфејса (формулар за претрагу аутомобила), без обзира на технологију која се користи, типичан пример кода најчешће садржи делове који се односе на неколико различитих аспеката корисничког интерфејса измешане на једном месту. Кривица за ово није само на програмерима, већ и на технологијама које не спречавају неадекватне стилове програмирања.

```
<div style="width: 100%; background-color: gray;">
    <form method="post" action="/searchcar">
        <h3>Search car by type</h3>
        <select id="carType">
            <option value="volvo">AUDI</option>
            <option value="mercedes">BMW</option>
            <option value="mercedes">FIAT</option>
            <option value="audi">OPEL</option>
        </select>
        <input type="submit" value="Search Car" />
    </form>
    <table id="resultTable">
        <tr>
            <td>Year</td><td>Color</td>
        </tr>
        <%
            List<Car> carList =
                    (List<Car>) request.getAttribute("cars");
            if (carList != null) {
                for (int i = 0; i < carList.size(); i++) {
        %>
            <tr>
                <td><%carList.get(i).getYear();%></td>
                <td><%carList.get(i).getColor();%></td>
            </tr>
        <%}}%>
    </table>
    <input type="button" value="Clear results" onclick="clearTable();">
</div>
```

а)

```
VerticalPanel vp = new VerticalPanel();
Label l = new Label("Searh car by type");
l.addStyleName("bold");
vp.add(l);
ListBox carType = new ListBox();
carType.addItem("AUDI");
carType.addItem("BMW");
carType.addItem("FIAT");
carType.addItem("OPEL");
carType.setVisibleItemCount(1);
vp.add(carType);
Button submitButton = new Button("Search Car");
vp.add(submitButton);
final Grid grid = new Grid();
vp.add(grid);
Button clearButton = new Button("Clear Results");
vp.add(clearButton);
clearButton.addClickHandler(new ClickHandler() {
  @Override
  public void onClick(ClickEvent arg0) {
     grid.clear();
  }
});
final Reader carReader = new Reader();
submitButton.addClickHandler(new ClickHandler() {
  @Override
  public void onClick(ClickEvent arg0) {
     List<Car> cars = carReader.readCars();
     for (Car car : cars)
     {
        grid.resizeRows(grid.getRowCount() + 1);
        grid.setWidget(grid.getRowCount() - 1, 0,
              new Label(car.getYear()));
        grid.setWidget(grid.getRowCount() - 1, 1,
              new Label(car.getColor()));
     }
  }
});
```

б)

Слика 1. Измешани аспекти у коду корисничког интерфејса: а) на језику *HTML* и б) на језику *Java*.

Поред тога, у највећем броју случајева, технологије за развој корисничких интерфејса, а посебно њихови апликативни програмски интерфејси и хијерархије наслеђивања, осмишљени су тако да буду лакши за имплементацију и одржавање, а не да буду лаки за употребу и усклађени са потребама и задацима њихових корисника, односно програмера. Прилагођавање дизајна апликативних програмских интерфејса потребама корисника (уместо потребама имплементатора) предложено је од стране неколико аутора. Хенинг (*Henning*) [14] открива тип понашања који доводи до лошег дизајна и скреће пажњу на важност одговорности и задатака корисника: „… имплементатор (апликативног програмског интерфејса) је фокусиран на решавање проблема, а задаци и одговорности корисника се брзо заборављају“. Слично томе, Кларк (*Clarke*) запажа следеће [6] : „Примена приступа у развоју апликативног програмског интерфејса који се заснива на сценаријима употребе осигурава да ће интерфејс исправно одговорити на задатке корисника, уместо да открива детаље имплементације“. Очигледно је да оба аутора наглашавају важност потреба корисника у дизајну апликативних програмских интерфејса. Како су технологије за развој корисничких интерфејса такође у форми апликативних програмских интерфејса, мора се обратити пажња на њихов дизајн. Да би се постигао завидан ниво употребљивости, морају се прво идентификовати основне одговорности и задаци програмера корисничких интерфејса и тек након и на основу тога ући у процес развоја технологије за развој корисничких интерфејса.

У току процеса идентификације ових одговорности, који ће бити описан у наредном поглављу, поштована су два важна принципа. С обзиром на то да класификација одговорности треба да буде основа за дизајн новог апликативног програмског интерфејса за изградњу графичког корисничког интерфејса, она мора бити природна и лако прихватљива за програмере. Осим тога, она не сме садржати велики број одговорности, јер би у том случају била тешка за памћење и одржавање. Према томе, дефиниција одговорности у развоју графичких корисничких интерфејса у контексту апликативних програмских интерфејса гласи:

> *Одговорност у развоју графичких корисничких интерфејса јесте подскуп елемената апликативног програмског интерфејса (поља, методе, конструктори, класе) који управљају једним истим аспектом програмирања корисничких интерфејса и заједно формирају интуитивну и кохерентну целину.*

У наставку текста ће бити изложена финална форма класификације одговорности како би се појединачне одговорности даље у тексту могле референцирати. У итеративном процесу који ће бити описан нешто касније, а у складу са подацима добијеним у експериментима, идентификоване су основне одговорности у развоју корисничких интерфејса: структура, распоред (енгл. *layout*), понашање, приступ подацима (енгл. *data access / data binding*), форматирање података, валидација података и стилизација.

### Приступ подацима

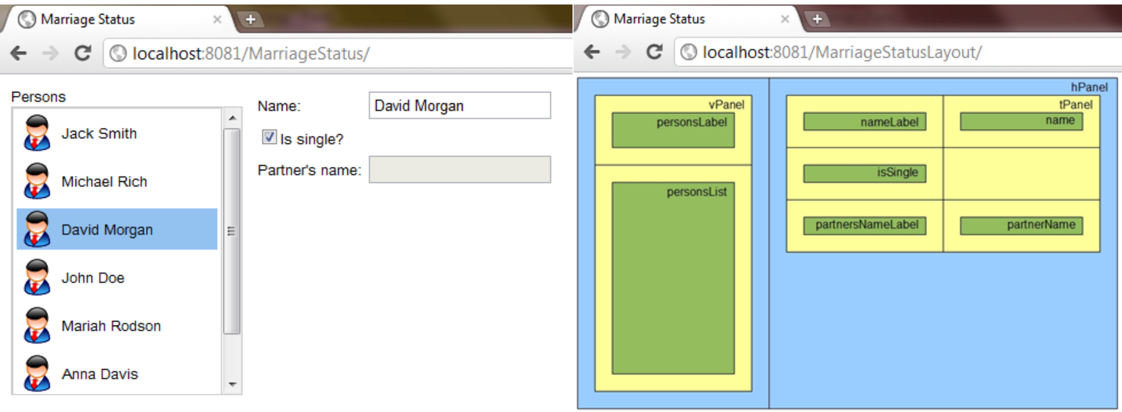
Један од основних задатака корисничког интерфејса јесте да прикаже релевантне податке кориснику апликације. Како би задовољио ову елементарну потребу сваког корисника, без обзира на тип окружења (десктоп, веб, клијент-сервер, мобилни телефон), програмер мора да приступи подацима користећи елементе датог апликативног програмског интерфејса за развој корисничких интерфејса. Према томе, одговорност приступа подацима подразумева елементе апликативног програмског интерфејса који служе за успостављање и одржавање комуникационих тачака у корисничком интерфејсу које обезбеђују двосмерну комуникацију са позадинском логиком (енгл. *backend logic*), моделом података или апликационим сервисима.

### Структура

Важан корак у процесу имплементације произвољног дела корисничког интерфејса представља избор тј. специфицирање градивних графичких елемената (енгл. *widget*). Ово подразумева избор и инстанцирање најприкладнијих типова графичких елемената и њихово постављање у жељено почетно стање (постављањем вредности поља и позивом метода). Структурни аспект програмирања корисничких интерфејса обухвата елементе апликативног корисничког интерфејса који се користе у фазама „креирања“ и „постављања“ (енгл. *create and set*) обрасца „креирај-постави-позови“ (енгл. *create-set-call*) [48]. Кроз остале аспекте програмирања корисничких интерфејса, као и у фази „позивања“, програмер се ослања на ову почетну иницијализовану структуру и додаје преостале функционалности. Другим речима, иницијализовани графички елементи морају се распоредити на екрану, обогатити динамичним понашањем, подаци које приказују се морају дохватити и форматирати и морају се стилски и визуелно прилагодити захтевима наручиоца.

### Распоред

Распоређивање структурних елемената корисничког интерфејса значи управљање њиховим положајем, величином и видљивошћу. Ово подразумева подешавање већ инстанцираних структурних графичких елемената, али и инстанцирање нових графичких елемената задужених искључиво за распоред (нпр. панели, табови, декови и друге врсте контејнерских компонената) и њихово подешавање са циљем постизања жељеног визуелног распореда. Одговорност распоређивања, према томе, подразумева подскуп елемената апликативног програмског интерфејса који се користе за извршавање поменутих акција. Слика 2 илуструје пример једне корисничке форме (лево) и њен аспект распореда графичких елемената (десно).




Слика 2. Пример формулара корисничког интерфејса (лево) и аспект распореда истог формулара (десно).

## Понашање

Кориснички интерфејс апликације је типичан систем базиран на догађајима (енгл. *event-driven*). Догађаје генерише корисник апликације (нпр. притискањем дугмета или хипер-линка на самом интерфејсу или тастера на тастатури), апликација (нпр. нотификацијом о промени стања података), или сам кориснички интерфејс интерно (нпр. активни графички елементи као што су тајмери). Програмски код који реагује на ове догађаје и управља догађајима иницира промене у корисничком интерфејсу и представља његово понашање. На пример, када један графички елемент региструје догађај селекције елемента у листи, други графички елемент треба да се ажурира у складу са селекцијом. Програмирање оваквог понашања спреже графичке елементе актере, односно креира структурне или функционалне везе између њих и потенцијално може да наруши енкапсулацију. Показало се да највише грешака у програмском коду корисничких интерфејса настаје управо у оваквим ситуацијама интеракције између графичких елемената [52]. Штавише, с обзиром на то да је структура корисничког интерфејса типично хијерархијска, у сложеним структурама врло је тешко разумети зависности које превазилазе однос „родитељ-дете“. С друге стране, овакве зависности су неизбежне, јер оне обезбеђују оно што кориснички интерфејс у суштини ради. Према томе, сваки апликативни програмски интерфејс за развој корисничких интерфејса садржи елементе који управљају интеракцијом графички елемента, и то су управо елементи који формирају понашање као посебан аспект програмирања корисничких интерфејса. Библиотека *Qt* [36] нуди посебан механизам „сигнала и слотова“ (енгл. *signals and slots*), који на експлицитан начин решава међусобну интеракцију графичких елемената, док већина осталих библиотека понашање ослања на процедуре за обраду догађаја (енгл. *event handler*).

## Форматирање података

Као што је већ споменуто, један од задатака корисничког интерфејса јесте да прикаже податке кориснику. Да би их приказао на начин који одговара кориснику, подаци се морају форматирати. Разни фактори утичу на то шта је „одговарајући начин“. Фактори могу бити специфични за домен апликације, специфични за географски регион, за циљно тржиште итд. На пример, број 25/100 може бити представљен на више начина: 0.250, 0,250, 0.25, .25 итд. Или, на пример, датум 12.7.1981. може бити приказан на један од следећих начина: 12-07-1981, 12/07/1981, 7/12/81, 12. јул 1981. итд.

Подаци се могу посматрати као инстанце типова података. Форматирање података се не односи само на форматирање инстанци примитивних или уграђених типова. Инстанце сложених типова (нпр. доменских класа као што су особа, предузеће, уговор итд.) такође се

манифестују у корисничком интерфејсу. Оне се појављују на различитим местима: у табелама, листама, контролама за селекцију итд. Манифестације инстанци сложених типова могу бити веома комплексне и садржати више вредности атрибута објеката које представљају. На пример, репрезентација инстанце особе може да се појави у графичком елементу за селекцију примаоца електронске поште као: „Марко Марковић – марко.марковић@абв.рс“. Једна од могућих дефиниција форматирања у овом случају би изгледала „#име #презиме - #имејл“.

Репрезентација исте инстанце може да се појави на више места у корисничком интерфејсу апликације. Ова појављивања могу да варирају, од међусобно сличних или чак истих, до потпуно различитих облика. Другим речима, на репрезентације исте инстанце у оквиру једне исте апликације могу бити примерљене различите дефиниције форматирања. Према томе, одговорност форматирања података обухвата елементе апликативног интерфејса који служе за а) креирање дефиниција форматирања за све типове података (примитивне и сложене, уграђене у доменске) и б) дефинисање правила применљивости за дефиниције форматирања, тј. мапирање дефиниција форматирања на делове корисничког интерфејса у којима дефиниције форматирања треба да важе.

У неким технологијама ова одговорност је адекватно препозната [26, 47]. У њима дефиниције форматирања покривају и примитивне и сложене типове података, док се правила применљивости дефинишу као подстабла хијерархијске структуре графичких елемената. Другим речима, дефиниција форматирања се додељује једном чвору стабла корисничког интерфејса и важи у целом подстаблу тог чвора, осим ако се негде у том подстаблу дефиниција форматирања не редефинише другом дефиницијом. Оваква правила применљивости су врло експресивна, јер се врло лако једна дефиниција форматирања може применити на цело стабло доделом те дефиниције кореном чвору стабла. Оваква примена дефиниција форматирања је и флексибилна, јер се свако правило применљивости може редефинисати новим правилом које може да важи чак и за један једини чвор стабла, уколико се правило применљивости веже за чвор који је лист у стаблу. Приказани начин наравно није једини начин примене правила форматирања. У другим технологијама срећу се други приступи.

## Валидација

Програмери обично морају да ограниче акције корисника и опсег података које корисници кроз кориснички интерфејс уносе у систем. Сваки елемент корисничког интерфејса који прихвата било какав вид интеракције са корисником је потенцијални валидациони пункт. Овакви елементи могу, и најчешће треба, да уведу ограничења у а) начин на који корисници интерагују са апликацијом и б) величину прихватљивог опсега података за унос. Валидација као одговорност обухвата елементе апликативног програмског интерфејса који се могу употребити за имплементацију валидационих правила, као и за презентацију валидационих порука кориснику апликације.

## Стилизација

Финално дотеривање и визуелна стилизација корисничког интерфејса подразумева примену визуелних и ергономских корекција на графичке елементе. Резултат ових корекција је жељени изглед и општи утисак (енгл. *look and feel*) о корисничком интерфејсу који се добрим делом преносе и на утисак о апликацији у целини. Ова подешавања укључују боје, фонтове, градијенте, сенке, маргине, ивице итд. Иако није кључан за само функционисање корисничког интерфејса, овај аспект програмирања је врло важан јер софтверу даје визуелни идентитет. Слично форматирању података, одговорност стилизације обухвата елементе апликативног програмског интерфејса који служе за а)

креирање дефиниција стилизације и б) дефинисање правила примењивости за дефиниције стилизације тј. мапирање дефиниција стилизације на делове корисничког интерфејса у којима дефиниције стилизације треба да важе. Правила примењивости могу да искористе хијерархијску организацију структуре корисничког интерфејса, тип графичких елемента, или било које друге карактеристике како би једнозначно дефинисала скуп графичких елемената за које треба да важи одређено правило стилизације. Типичан пример стилизације на основу типа елемента може се видети у технологијама *HTML*[4] и *CSS*[5] (енгл. *HyperText Markup Language* и *Cascading Style Sheets,* респективно) где се дефиниција стилизације на језику *CSS* може лако применити на све елементе језика *HTML* одређеног типа. На пример, дефиниција стилизације ивице може се једноставно применити на сву дугмад на екрану на следећи начин:

```
button { border: 1px solid black; }
```

Основна концептуална разлика између одговорности форматирања података и стилизације јесте у томе што се прва односи на податке, а друга на кориснички интерфејс тј. на саму апликацију. У табели 2 сумиране су све наведене одговорности и њихове кратке дефиниције.

| Одговорност | Дефиниција |
|---|---|
| Структура | Креирање и иницијализација градивних елемената корисничког интерфејса. |
| Распоред | Управљање положајем, величином и видљивошћу графичких елемената. |
| Приступ подацима | Креирање и одржавање двосмерне комуникације са позадинском логиком, моделом података или апликационим сервисима. |
| Понашање | Динамички аспекти корисничког интерфејса, спрега и интеракција графичких елемената. |
| Форматирање података | Креирање дефиниција форматирања за све типове података, интернационализација, и креирање правила примењивости за дефиниције форматирања. |
| Стилизација | Креирање дефиниција стилизације елемената корисничких интерфејса и креирање правила примењивости за дефиниције стилизације. |
| Валидација | Ограничавање начина на који корисници интерагују са апликацијом и прихватљивог опсега улазних података. Приказивање валидационих порука корисницима. |

Табела 2. Раздвајање одговорности у програмирању корисничких интерфејса.

### Остале одговорности

Постоји и релативно мали и не тако често употребљаван скуп елемената апликативних програмских интерфејса који се не могу сврстати ни у једну од споменутих одговорности. Ти елементи се односе на ауторизацију, приказ порука о статусу, приступачност (енгл. *accessibility*) итд. Адаптивност различитим уређајима и контекстима је аспект за који се

[4] http://www.w3.org/html/
[5] http://www.w3.org/Style/CSS/Overview.en.html

показало да се не огледа у превеликом броју елемената апликативног корисничког интерфејса који адресирају адаптивност, већ у примени осталих стандардних елемената на различите начине у зависности од контекста или уређаја.

# *Поступак идентификације одговорности*

У овом поглављу биће приказана емпиријска студија спроведена са циљем утврђивања релевантних аспеката, односно одговорности у развоју корисничких интерфејса приказаних у претходном одељку. Насупрот другим приступима који се претежно заснивају на интуицији и искуству, приступ који ће бити приказан овде представља систематичну емпиријску анализу једног постојећег апликативног програмског интерфејса и његове употребе у индустријским пројектима. Ово поглавље почиње описом програмског интерфејса који је анализиран. Затим се даје опис итеративног поступка идентификације аспеката. Коначно, у последњем делу објашњени су сви конкретни експерименти који су спроведени у процесу идентификације одговорности.

## Опис апликативног програмског интерфејса

У поступку идентификације одговорности у развоју корисничких интерфејса анализиран је апликативни програмски интерфејс *SOLoist* [47]. У питању је била претеча верзије која ће бити приказана у овој дисертацији. Верзија која је коришћена била је погодна за раздвајање одговорности (у наставку текста су објашњене погодности), али не и специјално направљена да буде у складу са принципом раздвајања одговорности. Неки елементи коришћене верзије били су већ најављени у литератури [26].

*SOLoist* је веб оријентисан и базиран на језику *UML* и технологији *Java*[6]. Коришћен је за развој десетак средњих и великих пројеката у последње три године. Најновија верзија је направљена за веб окружење. Она акумулира сва искуства, концепте и методе из неколико претходних генерација ове технологије која је укупно гледано била коришћена у неколико десетина пројеката у последњој деценији. У овом поглављу биће детаљније описан апликативни програмски интерфејс ове технологије као и мотивација за коришћење баш њега у процесу идентификације одговорности у развоју графичких корисничких интерфејса.

Апликативни програмски интерфејс технологије *SOLoist* који служи за изградњу корисничких интерфејса садржи 110 класа и нуди преко 70 различитих типова графичких елемената. У одређеном смислу врло је сличан стандардним технологијама за развој

[6] Java: http://www.oracle.com/technetwork/java/index.html

корисничких интерфејса базираним на компонентама. Наиме, он прописује прављење корисничког интерфејса у виду хијерархије, инстанцирањем графичких елемената и њиховим повезивањем уз помоћ веза „родитељ-дете“. Он нуди стандардне механизме за распоређивање, тј. позиционирање елемената: хоризонтално, вертикално, апсолутно, слојевито итд. У смислу стилизације, користи се још један стандардан приступ, технологија *CSS*.

Међутим, постоји неколико аспеката који су адресирани на посебан начин. Прво, управљање догађајима и динамичност се третира посебним механизмом који превазилази стандардни начин (тзв. енгл. *event-handlers*). Овај механизам се назива „пинови и жице“ (енгл. *pins and wires*) и сличан је механизму сигнала и слотова технологије *Qt*. Механизам је изузетно изражајан и поједностављује програмирање интеракције графичких елемената. Као последица тога, процес идентификације аспекта понашања је значајно поједностављен јер се своди на идентификацију „жица“ између графичких елемената које су врло експлицитне. Друго, овај апликативни програмски интерфејс подиже ниво апстракције у смислу приступа подацима. Уместо са техничког нивоа, он се ослања на модел података на језику *UML* и приступа подацима декларативно. Овакав начин приступа се такође лакше може идентификовати и анализирати. Коначно, део за форматирање података, поред примитивних, подржава и комплексне типове података и омогућава лакше проналажење и анализирање.

Важно је напоменути и то да је процес идентификације свакако могао бити спроведен и коришћењем неког другог програмског интерфејса и анализом изворног кода других пројеката. Међутим, посебне особине технологије *SOLoist,* које су наведене у претходном пасусу, поткрепиле су одлуку да се баш он користи у процесу идентификације. Процењено је да ће сам поступак бити мање подложан грешкама. У сваком случају, избор самог програмског интерфејса није кључан, уколико се покаже примењивост идентификованих одговорности, тј. предложене класификације.

### Опис итеративног поступка

Први корак студије био је преглед постојећих научно-истраживачких радова који се баве темом одговорности, односно одвајања одговорности у развоју корисничких интерфејса. Овај преглед пружио је довољно полазног материјала да се формира прелиминарна класификација одговорности, коју је касније кроз експерименте требало обликовати у стабилну и утемељену класификацију. Уследила је анализа постојећег апликативног програмског интерфејса [26, 47] на нивоу сваког појединачног елемента. Сваки елемент је додељен једној или више одговорности из полазне класификације и онда је извршено пребројавање елемената који припадају свакој од одговорности.

Потом је уследила опсежна анализа изворног кода три велика пројекта са динамичним и захтевним корисничким интерфејсима имплементираним уз помоћ истог програмског интерфејса. Овај експеримент је имао за циљ да утврди колико често се сваки од елемената програмског интерфејса користи у реалним пројектима пребројавањем референци позивања у изворном коду споменутих пројеката. На основу тога и на основу резултата првог експеримента где је утврђено које одговорности покрива сваки елемент програмског интерфејса, утврђено је колико се процентуално посвећује програмског кода свакој од одговорности појединачно.

Као трећи корак, анализиран је репозиторијум пријављених проблема (енгл. *issue tracker*) за иста три пројекта. Циљ је био да се испита фаза еволуције корисничког интерфејса и да се провере резултати прва два експеримента.

Поступак идентификације одговорности био је суштински итеративан. Као увод у анализу, прво су прегледане постојеће класификације одговорности у развоју корисничких интерфејса, што је помогло да се формира прелиминарна класификација. Прелиминарна класификација је коришћена као полазна тачка у поступку. Експерименти су морали да потврде постојање или непостојање појединачних одговорности из прелиминарне класификације и евентуално да утврде постојање нових. Другим речима, требало је да експерименти обликују прелиминарну класификацију и учине је стабилном. Овде треба нагласити да се током тог итеративног поступка не само променила прелиминарна листа одговорности, већ и дефиниције самих одговорности. Прелиминарна класификација одговорности је неколико пута модификована и проширивана у току експеримената у складу са новодобијеним експерименталним резултатима. Након сваке модификације, експерименти су понављани како би се од самог почетка свих експеримената узеле у обзир све потенцијалне одговорности.

Свака итерација у процесу идентификације имала је улазни скуп одговорности, који је требало испитати, односно потврдити или дисквалификовати одговорности из тог скупа или увести нове на основу довољне количине доказа из експеримената. Табела 3 приказује улазни скуп одговорности за сваку итерацију. Одговорности означена знаком „+“ су оне које су биле укључене у полазни скуп одговорности за дату итерацију.

| Број итерације<br>Одговорност | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Структура | + | + | + | + |
| Распоред | + | + | + | + |
| Приступ подацима | + | + | + | + |
| Стилизација | + | + | + | + |
| Форматирање података | - | + | + | + |
| Понашање | - | - | + | + |
| Валидација | - | - | - | + |
| Број спроведених експеримената | 1 | 1 | 3 | 3 |
| Коментар | 18 од 113 класа није класификовано. 12 од тих 18 класа задужено за форматирање података. | 112 пинова није класификова но. Пронађени докази за увођење понашања. | Пронађени докази за увођење валидације. | Нису пронађени докази о новој одговорности. Улазни скуп одговорности потврђен као релевантан. |

Табела 3. Итерације у процесу идентификације одговорности.

Улазни скуп прве итерације укључивао је структуру, распоред, приступ подацима и стилизацију. Скуп је са намером био најужи могући, како би се избегло понављање итерације због евентуалног дисквалификовања неке од полазних одговорности. Итерација је започета анализом апликативног програмског интерфејса технологије *SOLoist* (први експеримент). У овом експерименту анализирано је пет категорија елемената програмског интерфејса: класе, конструктори, методе, поља и пинови (специјални тип поља за управљање догађајима). Поступак је започет над класама тако што је свакој класи додељен подскуп одговорности из полазног скупа које та класа покрива. Међутим, од 113 класа, њих 18 (16%) није имало ниједну додељену одговорност из полазног скупа. Ово је био јасан индикатор недостатака полазног скупа. Са циљем да се полазни скуп допуни и прошири и да се експерименти понове, анализирано је споменутих 18 класа и утврђено је да је 12 класа међусобно сродно. Неке од тих 12 класа су биле стриктно намењене форматирању инстанци примитивних типова (датума, бројева, новчаних износа итд.). Међутим, велика већина је била везана за форматирање инстанци сложених типова. Ово је био разлог да дефинишемо нову одговорност форматирања података која покрива и примитивне и сложене типове података. Ова одговорност је укључена у полазни скуп одговорности и поступак је враћен на почетак.

Друга итерација започета је са полазним скупом кога су чиниле следеће одговорности: структура, распоред, приступ подацима, форматирање података и стилизација. Поново је покренут експеримент анализе програмског интерфејса и завршен за класе, методе, конструкторе и поља. Међутим, пинови нису могли бити класификовани, односно, није било одговарајуће одговорности којој су могли бити додељени. Свих 112 пинова су очигледно били повезани са одговорношћу која недостаје полазном скупу: понашање. Ова одговорност је додата полазном скупу и процес је поново враћен на почетак. Овде треба нагласити да се процес идентификације мора вратити на почетак након сваке измене полазног скупа одговорности или дефиниције одговорности. Разлог је у томе што би додељивање одговорности елементима програмског интерфејса можда било урађено другачије да је новоуведена одговорност била на располагању за додељивање.

Полазни скуп за трећу итерацију чиниле су структура, распоред, понашање, приступ подацима, форматирање података и стилизација. У оквиру ове итерације урађена су сва три експеримента: анализа апликационог програмског интерфејса, анализа употребе програмског интерфејса и анализа пријављених проблема. Тек након завршетка свих експеримената утврђено је да се елементи програмског интерфејса који су задужени за валидацију уноса и акција корисника појављују често као некласификовани. Ово је посебно било уочљиво у резултатима трећег експеримента, односно у резултатима анализе проблема. С обзиром на то да је велики део проблема био везан за валидацију, одлучено је да се уведе нова одговорност и да се експерименти још једном понове.

Полазни скуп четврте и последње итерације састојао се из следећих одговорности: структура, распоред, понашање, приступ подацима, форматирање података, стилизација и валидација. Спроведена су сва три експеримента до краја. Није утврђено постојање доказа о новим одговорностима, а истовремено, довољно доказа је било да се оправда постојање сваке од одговорности из полазног скупа.

С обзиром на то да је било неопходно да се експерименти понављају, у наредним деловима овог поглавља биће приказани резултати експеримената из последње, четврте итерације која је завршена без додавања или уклањања одговорности и без измена дефиниција одговорности.

### Анализа апликативног програмског интерфејса

Анализа апликативног програмског интерфејса подразумевала је испитивање сваког његовог елемента и то сваке класе, конструктора, методе, поља и пина. Сваком елементу додељена је једна или више од одговорности из полазног скупа. Штавише, врло често се дешавало да један елемент програмског интерфејса покрива више одговорности. На пример, конструктор *Label()* би могао да се класификује у структуру. Међутим, конструктор *Label(GUIComponent parent)* би морао да се класификује и у структуру али и у распоред, јер дефинисање родитеља графичког елемента у одређеној мери одређује и распоред тог елемента на екрану. У таквим случајевима вишеструке одговорности вршена је процена удела одговорности. У овом примеру, конструктору са параметром додељена је једна половина одговорности за структуру и једна половина за распоред. У табели 4 дати су резултати овог експеримента за последњу, четврту, итерацију.

Најпре је пребројано колико је елемената апликативног програмског интерфејса везано за сваку од одговорности. То је приказано у колонама са знаком „#“ у заглављу табеле 4. Због чињенице да један елемент програмског интерфејса може имати више одговорности, сума свих вредности у ћелијама ових колона за један тип елемената програмског интерфејса превазилази одговарајући укупан број таквих елемената. На пример, укупно постоји 315 конструктора, а збир вредности за све одговорности је 101 + 151 + 63 + 145 + 55 + 8 + 1 + 19 = 543. Захваљујући проценама удела одговорности за сваки елемент програмског интерфејса издвојени су удели сваке одговорности за сваки тип елемената (колоне означене тачком „.“ за номиналне вредности, односно процентом „%“ за процентуалне). На пример, за поменутих 315 конструктора у програмском интерфејсу, 101 конструктор је био одговоран за приступ подацима бар делимично (обратити пажњу на ћелије табеле 4 са подебљаним оквиром). Када су сумирани удели одговорности за тај 101 конструктор добијена је номинална вредност 56,23. Ова вредност представља 17,9% укупног броја конструктора (315).

| Анализа програмског интерфејса | | Елементи технологије *SOLoist* | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Конструктори | | | Методе | | | Поља | | | Пинови | | | Укупно | | |
| | Број | 315 | | | 179 | | | 197 | | | 112 | | | 803 | | |
| | | # | . | % | # | . | % | # | . | % | # | . | % | # | . | % |
| Одговорности | Приступ подацима | 101 | 56.23 | 17.9 | 34 | 30.83 | 17.2 | 48 | 43.83 | 22.2 | 0 | 0.00 | 0.0 | 183 | 130.89 | 16.3 |
| | Структу-ра | 151 | 93.14 | 29.6 | 8 | 7.20 | 4.0 | 23 | 21.83 | 11.1 | 0 | 0.00 | 0.0 | 182 | 122.17 | 15.2 |
| | Понаша-ње | 63 | 24.58 | 7.8 | 32 | 30.70 | 17.2 | 24 | 12.58 | 6.4 | 112 | 112.00 | 100.0 | 231 | 179.86 | 22.4 |
| | Распоред | 145 | 89.89 | 28.5 | 48 | 46.70 | 26.1 | 60 | 59.50 | 30.2 | 0 | 0.00 | 0.0 | 253 | 196.09 | 24.4 |
| | Формати-рање података | 55 | 45.22 | 14.4 | 57 | 53.03 | 29.6 | 36 | 31.92 | 16.2 | 0 | 0.00 | 0.0 | 148 | 130.17 | 16.2 |
| | Стилиза-ција | 8 | 2.67 | 0.8 | 3 | 2.20 | 1.2 | 14 | 12.33 | 6.3 | 0 | 0.00 | 0.0 | 25 | 17.20 | 2.2 |
| | Валида-ција | 1 | 1.00 | 0.3 | 9 | 8.33 | 4.7 | 15 | 14.22 | 7.2 | 0 | 0.00 | 0.0 | 25 | 23.55 | 2.9 |
| | Остало | 19 | 2.28 | 0.7 | 0 | 0 | 0 | 7 | 0.78 | 0.4 | 0 | 0.00 | 0.0 | 26 | 3.06 | 0.4 |

Табела 4. Резултати анализе апликативног програмског интерфејса.

Након анализе целог програмског интерфејса и свих типова елемената, може се констатовати да свега неколико елемената програмског интерфејса није везано ни за једну

од полазних одговорности. Ово значи да предложена листа одговорности покрива велику већину елемената програмског интерфејса. У најмању руку, бар у анализираном програмском интерфејсу је тако. Такође треба приметити апсолутно униформне податке за једну врсту елемената програмског интерфејса – пинове. Ово је последица природе пинова, тј. њихове искључиве одговорности у генерисању догађаја и обради догађаја. Коначно, примећује се да је релативно мали део програмског интерфејса посвећен стилизацији и валидацији. Међутим, тек у наредним експериментима показаће се да ли је овај пропорционално мали део заиста толико малог значаја, односно, показаће се колико се често користе елементи програмског интерфејса за ове две одговорности. Највећи део програмског интерфејса је, према приложеним подацима, резервисан за распоред, понашање, структуру, приступ подацима и форматирање података.

## Студија коришћења

Резултате добијене у претходном експерименту треба пажљиво узети у обзир. Уколико се елемент апликативног програмског интерфејса не користи у пракси, онда је врло вероватно да се на основу њега не може закључити било шта о одговорностима које испитујемо. То је разлог због кога је спроведен експеримент у ком је анализирана употреба свих елемената датог програмског интерфејса.

У овом експерименту, анализирана су три средња и велика пројекта[7] развијена од стране различитих тимова програмера коришћењем истог програмског интерфејса. За сваки елемент утврђен је број места коришћења, односно број референци у изворном коду ових пројеката. Коришћено је развојно окружење *Eclipse*[8], односно његова алатка за претраживање места позивања. Добијени број референци датог елемента програмског интерфејса узет је у разматрање заједно са процењеним уделом одговорности тог елемента из првог експеримента (другим речима, које су одговорности датог елемента). На основу ова два податка о сваком елементу програмског интерфејса, могло се проценити колико често и колико интензивно се свака од одговорности референцира у изворном коду ових пројеката. Табела 5 даје резултате овог експеримента у последњој итерацији.

| Анализа коришћења | | Информациони систем | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | CRMS | | | HRMS | | | RECS | | | Укупно | | |
| | Број референци | 5284 | | | 26512 | | | 20021 | | | 51817 | | |
| | | # | . | % | # | . | % | # | . | % | # | . | % |
| Одговорности | Приступ подацима | 596 | 262.52 | 5.0 | 3602 | 2284.65 | 8.6 | 2478 | 1504.85 | 7.5 | 6676 | 4052.02 | 7.8 |
| | Структу-ра | 1064 | 507.99 | 9.6 | 4793 | 2978.71 | 11.2 | 3358 | 1686.63 | 8.4 | 9215 | 5173.33 | 10.0 |
| | Понаша-ње | 2663 | 2455.62 | 46.5 | 12146 | 11142.89 | 42.0 | 10440 | 9861.56 | 49.3 | 25249 | 23460.07 | 45.3 |
| | Распоред | 1636 | 1099.88 | 20.8 | 8482 | 6622.84 | 25.0 | 6317 | 4729.48 | 23.6 | 16435 | 12452.20 | 24.0 |
| | Формати-рање података | 88 | 73.25 | 1.4 | 911 | 760.47 | 2.9 | 317 | 266.97 | 1.3 | 1316 | 1100.69 | 2.1 |
| | Стилиза-ција | 752 | 714.83 | 13.5 | 2317 | 1977.58 | 7.5 | 1682 | 1447.92 | 7.2 | 4751 | 4140.33 | 8.0 |
| | Валида- | 149 | 108.44 | 2.1 | 875 | 530.89 | 2.0 | 520 | 323.44 | 1.6 | 1544 | 962.77 | 1.9 |

[7] Детаљи о комплексности пројеката могу се пронаћи у поглављу „Анализа предложеног решења“ ове дисертације.
[8] http://www.eclipse.org/

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ција | | | | | | | | | | | | |
| | Остало | 537 | 61.47 | 1.2 | 1897 | 213.96 | 0.8 | 1737 | 200.15 | 1.0 | 4171 | 475.58 | 0.9 |

Табела 5. Анализа коришћења елемената апликативног програмског интерфејса у реалним пројектима.

Резултати показују да предложена листа одговорности практично у потпуности задовољава захтеве у домену корисничког интерфејса које су пред програмере поставили анализирани пројекти. Ово је било очекивано, с обзиром на то да само 0,4% елемената апликативног програмског интерфејса није имало ниједну од предложених одговорности (видети резултате претходног експеримента), већ је завршило у групи „осталих“ одговорности. Закључак би био другачији да је анализа коришћења показала да се елементи програмског интерфејса који порkивају „остале“ одговорности интензивно користе у пракси, што се није догодило. Другим речима, удео осталих одговорности је само 0,9%.

Треба приметити да је удео предложених одговорности релативно стабилан од пројекта до пројекта, упркос различитостима њиховим домена. На крају, с обзиром на то да је овај експеримент дао информације о коришћењу сваког елемента апликативног програмског интерфејса, то може бити од помоћи у изради његових наредних верзија. Наиме, на основу тога се може одлучити које елементе програмског интерфејса задржати у наредној верзији, које прогласити застарелим (енгл. *deprecate*), а које елементе додатно елаборирати. То је у складу са генералним правилом које каже „да одлука о увођењу нове методе у апликативни програмски интерфејс зависи од процене колико ће та метода бити коришћена“ [14].

## Анализа еволуције корисничког интерфејса

Последњи експеримент је анализирао одржавање софтвера и корисничких интерфејса. Циљ је био да се утврди како се кориснички интерфејс мења након иницијалне испоруке софтвера. У овом експерименту коришћене су информације из репозиторијума за складиштење проблема (енгл. *issue/bug tracking*). Резултати су приказани у табели 6. Бројеви у табели су добијени анализом проблема из репозиторијума на дан 15. фебруара 2012. године за *CRMS*, 4. марта 2012. године за *HRMS*, и 15. марта 2012. године за *EMS* пројекат.

| Анализа проблема | | Информациони систем | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | EMS | | | HRMS | | | CRMS | | | Укупно | | |
| | Проблеми (везани за кориснички интерфејс / укупно) | 140 / 197 | | | 418 / 596 | | | 146 / 198 | | | 704 / 991 | | |
| | | # | . | % | # | . | % | # | . | % | # | . | % |
| Одговорности | Приступ подацима | 15 | 14.25 | 10.2 | 149 | 38.55 | 9.2 | 25 | 8.33 | 5.7 | 189 | 61.13 | 8.7 |
| | Структура | 45 | 29.78 | 21.3 | 240 | 110.72 | 26.5 | 80 | 35.06 | 24.0 | 365 | 175.56 | 25.0 |
| | Понашање | 40 | 27.88 | 19.9 | 185 | 60.38 | 14.4 | 72 | 32.39 | 22.2 | 257 | 120.65 | 17.1 |
| | Распоред | 46 | 29.42 | 21.0 | 235 | 107.22 | 25.6 | 72 | 25.47 | 17.4 | 353 | 162.11 | 23.0 |
| | Формати-рање података | 10 | 9.50 | 6.8 | 34 | 30.00 | 7.2 | 14 | 9.82 | 6.7 | 58 | 49.32 | 7.0 |
| | Стилиза-ција | 20 | 12.17 | 8.7 | 42 | 11.07 | 2.6 | 22 | 11.37 | 7.8 | 84 | 34.61 | 4.9 |
| | Валидација | 7 | 7.00 | 5.0 | 44 | 38.57 | 9.2 | 12 | 10.33 | 7.1 | 63 | 55.90 | 7.9 |
| | Остало | 10 | 10.00 | 7.1 | 22 | 21.50 | 5.3 | 16 | 13.23 | 9.1 | 48 | 44.73 | 6.4 |

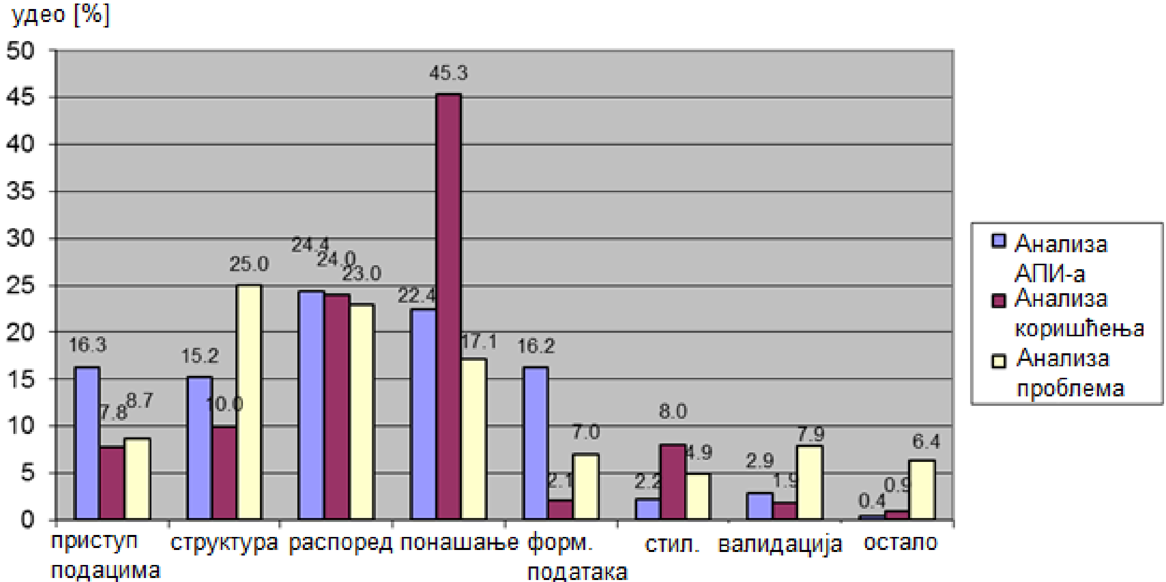
Табела 6. Удели одговорности у развоју корисничких интерфејса у проблемима из репозиторијума проблема.

У експерименту су мануелно анализирани сви проблеми и захтеви за изменама пријављени за ова три пројекта. Они проблеми који нису имали директне везе са корисничким интерфејсом су претходно исфилтрирани. За сваки пријављени проблем процењено је на које одговорности у развоју корисничких интерфејса утиче решавање тог проблема. За оне проблеме који су покривали више одговорности процењен је удео сваке од релевантних одговорности слично као у првом експерименту. На пример, постојао је проблем који је гласио: „Смањити секције за техничке документе и променити фонт који се користи“. Овом проблему додељен је удео од једне половине за стилизацију (фонта) и једна половина за распоред (смањење висине секције). На основу прикупљених података за сваки од проблема из репозиторијума утврђено је колико проблема је везано за сваку од одговорности (колоне са знаком „#“ у заглављу табеле 6). Због чињенице да један проблем може имати више одговорности, сума свих вредности у ћелијама ових колона за један пројекат превазилази одговарајући укупан број проблема, баш као у првом експерименту. На основу података о уделима сваке од одговорности за сваки проблем, утврђени су укупни удели сваке од одговорности за сваки пројекат (колоне означене тачком „.“ за номиналне вредности и процентом „%“ за процентуалне).

Потврђено је да предложени скуп одговорности покрива више од 90% укупних напора у фази одржавања корисничког интерфејса. Такође је закључено да се структура, распоред и понашање најчешће поправљају у тој фази. На крају, као и у претходном експерименту, потврђено је да одговорности задржавају сличан удео без обзира на врсту пројекта.

### Поређење резултата

Слика 3 графички приказује резултате три спроведена експеримента. Она даје процентуални удео за сваку од одговорности и за сваки спроведени експеримент. На пример, распоред има удео од 24,4% експерименту анализе апликативног програмског интерфејса, 24,0% у анализи коришћења елемената програмског интерфејса и 23,0% у анализи проблема.




Слика 3. Резултати анализе апликативног програмског интерфејса, анализе коришћења елемената истог интерфејса и анализе проблема из репозиторијума.

Према овим резултатима, одмах се може констатовати да не постоји превелики број одговорности у развоју корисничких интерфејса или бар да се не улаже превише напора у њиховом испуњавању. Другим речима, предложени скуп одговорности покрива скоро све напоре који се улажу у развоју корисничких интерфејса.

Друго, очигледно постоје јаки докази у корист одговорности приступа подацима, структуре, распореда и понашања. На основу приложених резултата, ове одговорности су јако присутне у апликативном програмском интерфејсу, одговарајући елементи програмског интерфејса се врло често користе и, на крају, кориснички интерфејси се врло често поправљају у домену ових одговорности. Може се чак рећи да ове четири одговорности представљају језгро свих напора који се морају уложити како би се добио кориснички интерфејс који функционише. Другим речима, не постоји кориснички интерфејс без података а ни презентације података без приступа подацима. Такође нема корисничког интерфејса без градивних делова (структура), нити без најосновнијег распореда тих делова. На крају, динамичност корисничког интерфејса, што је једна од основних карактеристика, не може се постићи без програмирања понашања.

Форматирање података, стилизација и валидација уноса и акција корисника су значајни, али не у тој мери као претходно описане четири одговорности. У суштини, ове три одговорности представљају напоре које треба уложити како би се завршио кориснички интерфејс и припремио за употребу. Другим речима, ово подразумева унапређење визуелног стила и идентитета, припрему формата података за циљни географски регион или тржиште, унапређење робусности, безбедности и смањење броја грешака.

## Анкета

Са циљем да се прикупе и мишљења програмера о одговорностима представљеним у претходном делу дисертације, спроведена је анкета базирана на Ликертовој скали (енгл. *Likert scale*[9]). Упитник је имао 25 питања. Свако питање било је у форми исказа. Задатак кандидата који су одговарали на питања био је да изразе своје мишљење о сваком исказу и то специфицирајући степен слагања или неслагања са исказом. Коришћена је симетрична скала са пет могућих одговора, односно степени слагања: потпуно се слажем (енгл. *strongly agree*), слажем се (енгл. *agree*), имам неутралан став (енгл. *neutral*), не слажем се (енгл. *disagree*) и потпуно се не слажем (енгл. *strongly disagree*).

Искази из упитника представљали су дефиниције аспеката, односно одговорности, у развоју графичких корисничких интерфејса. Од двадесет пет дефиниција, седам дефиниција су биле дефиниције одговорности представљене у овој дисертацији. Осталих осамнаест дефиниција су узете из екстерних извора (већина се може пронаћи у научним радовима цитираним у прегледу постојећих решења ове дисертације). Свих двадесет пет дефиниција су најпре намерно измешане, а затим у том измешаном редоследу понуђене кандидатима да се о свакој од њих изјасне. Дефиниције су приказане у табели 7, у колони „дефиниције“ и то у редоследу појављивања у упитнику. Анкета је спроведена на енглеском језику, па су дефиниције на енглеском језику дате у угластим заградама одмах испод превода. Седам дефиниција из ове дисертације су у табели обележене посебном бојом, док се у упитнику, наравно, нису разликовале од других дефиниција.

| | Дефиниције | Укупан скор (T) |
|---|---|---|
| 1 | Распоред видим као управљање положајем, величином и | 28 |

[9] http://en.wikipedia.org/wiki/Likert_scale

| | | |
|---|---|---|
| | видљивошћу графичких елемената. [I see layout as management of position, size, and visibility of GUI widgets.] | |
| 2 | Интеракцију видим као „ослушкиваче“ и управљање догађајима. [I see interaction as listeners and event handling.] | 29 |
| 3 | Презентацију видим као средства за представљање садржаја (података из апликације) и навигационе структуре / хијерархије. [I see presentation as means for presenting content (application data) and navigational structure / hierarchy.] | 30 |
| 4 | Понашање видим као управљање навигацијом, променама и информисање корисника о статусу апликације. [I see behavior as dealing with navigation, editing, and informing users about application status.] | 27 |
| 5 | Валидацију видим као рестрикцију уноса корисника и опсега прихватљивих података, као и презентацију валидационих одговора кориснику. [I see validation as restriction of users' input and the acceptable data extent, as well as presentation of post-validation feedback to the users.] | 35 |
| 6 | Стилизацију видим као прављење дефиниција стила (боје, фонтови, градијенти, сенке, ивице итд.) и прављење правила важења за те дефиниције (односно мапирања између дефиниција стила и делова корисничког интерфејса у ком те дефиниције важе). [I see styling as creation of styling definitions (colors, fonts, gradients, shadings, borders, etc.) and creation of applicability policies for styling definitions (mapping between styling definitions and parts of the GUI that these definitions should be applied to).] | 34 |
| 7 | Приказивање видим као распоред, стил (боје, фонтови, ивице) и текст (управљање свим елементима базираним на тексту, укључујући и интернационализацију). [I see appearance as layout, style (colors, fonts, borders), and text (dealing with all language-based elements including internationalization).] | 29 |
| 8 | Груписање видим као управљање хијерархијом графичких елемената. [I see grouping as management of parent-child hierarchy of GUI widgets.] | 25 |
| 9 | Распоред видим као управљање позицијом графичких контрола како релативно једне у односу на другу, тако и у односу на прозор. [I see layout as management of position of controls relative to each other and/or directly in the window.] | 32 |
| 10 | Видим одговорност дијалога као елементе интеракције који | 23 |

| | | |
|---|---|---|
| | формирају везу између задатака корисника и презентације.<br>[I see dialog concern as interaction elements forming a bridge between users' tasks and the presentation.] | |
| 11 | Видим понашање као промене текста, превлачење елемената, активацију, селекцију, означавање итд.<br>[I see behavior as text editing, drag and drop, activation, selection, highlighting, etc.] | 22 |
| 12 | Видим „callback" одговорност као скуп елемената и објеката који повезују делове корисничког интерфејса са функционалним делом апликације и између себе.<br>[I see callback concern as a set of elements or objects for connecting GUI parts with functional part of application and between themselves.] | 31 |
| 13 | Видим графички апсект као управљање изгледом, фонтовима и бојама.<br>[I see graphical concern as a management of appearance, fonts, and colors.] | 26 |
| 14 | Видим понашање као динамичке аспекте корисничког интерфејса и интеракцију између графичких елемената.<br>[I see behavior as dynamic aspects of GUI and coupling and interaction between GUI widgets.] | 33 |
| 15 | Видим апсект презентације као хијерархијску композицију елемената интеракције.<br>[I see presentation concern as hierarchical composition of interaction elements.] | 22 |
| 16 | Видим одговорност визуализације као избор графичких елемената, њихово подешавање и специфицирање њиховог распореда.<br>[I see visualization concern as defining GUI components, setting their properties, and specifying layout.] | 28 |
| 17 | Видим одговорност дијалога као навигациону структуру корисничког интерфејса и технике интеракције.<br>[I see dialog concern as a navigational structure of the GUI and interaction techniques.] | 25 |
| 18 | Видим аспект графичких елемената као графичке симболе, текстове и слике.<br>[I see viewable elements as graphical symbols, strings, and images.] | 29 |
| 19 | Форматирање података видим као прављење дефиниција форматирања за све типове података, интернационализацију и прављење правила важења за те дефиниције (односно мапирања између дефиниција форматирања и делова корисничког интерфејса у ком те дефиниције важе).<br>[I see data formatting as creation of format definitions for all types of | 34 |

| | | |
|---|---|---|
| | data, internationalization, and creation of applicability policies for format definitions (mapping between format definitions and parts of the GUI that these definitions should be applied to).] | |
| 20 | Видим приступ подацима као прављење и одржавање тачака комуникације корисничког интерфејса које омогућавају бидирекциону комуникацију са позадинском логиком, моделом података и апликативним сервисима.<br>[I see data binding as creation and maintenance of communication points in the GUI that provide bidirectional communication with the backend logic, data model, and application services.] | 32 |
| 21 | Видим структуру као прављење и иницијализацију структурних делова (графичких елемената) корисничког интерфејса.<br>[I see structure as creation and initialization of structural parts (widgets) of the GUI.] | 27 |
| 22 | Видим презентацију као мапирање између задатака и доменских елемената са једне стране и објеката интеракције као што су листе, стабла, дугмићи итд. са друге.<br>[I see presentation concern as a mapping between tasks and domain elements with interaction objects such as lists, buttons, trees, etc.] | 29 |
| 23 | Видим приступачност као степен прихватљивости корисничког интерфејса од стране људи са специјалним потребама.<br>[I see accessibility as degree to which GUI fits people with special needs or disabilities.] | 29 |
| 24 | Видим понашање као повезивање догађаја и акција.<br>[I see behavior as linking events and actions.] | 32 |
| 25 | Видим распоред као распоређивање објеката интеракције као и стилизацију (фонтови, боје итд.).<br>[I see layout as arrangement of the interaction objects as well as the style (fonts, colors, etc.).] | 23 |

Табела 7. Дефиниције из упитника.

Пре објашњења резултата упитника, треба нагласити да кандидати који су одговарали на упитник нису били упознати са резултатима претходног истраживања одговорности у развоју корисничких интерфејса које је изложено у овој дисертацији. Они нису знали који су искази предложени у овом истраживању, а који су узети из екстерних извора. Штавише, нису били упознати ни са чињеницом да у листи исказа уопште има исказа које треба проверити у односу на друге исказе. На тај начин, кандидати нису могли намерно одговарати у корист исказа предложених у овој дисертацији, јер нису знали ни да има таквих исказа, а ни који су то искази.

Упитник је попунило осам програмера различитог програмерског искуства, али са познавањем бар неколико технологија за развој корисничких интерфејса. Резултати су сабрани за сваки исказ, односно за свако питање. У сабирању резултата претпостављено је да је идентична удаљеност између два суседна одговора Ликерт скале. На пример, разлика између одговора „потпуно се слажем“ и одговора „слажем се“ иста је као разлика између

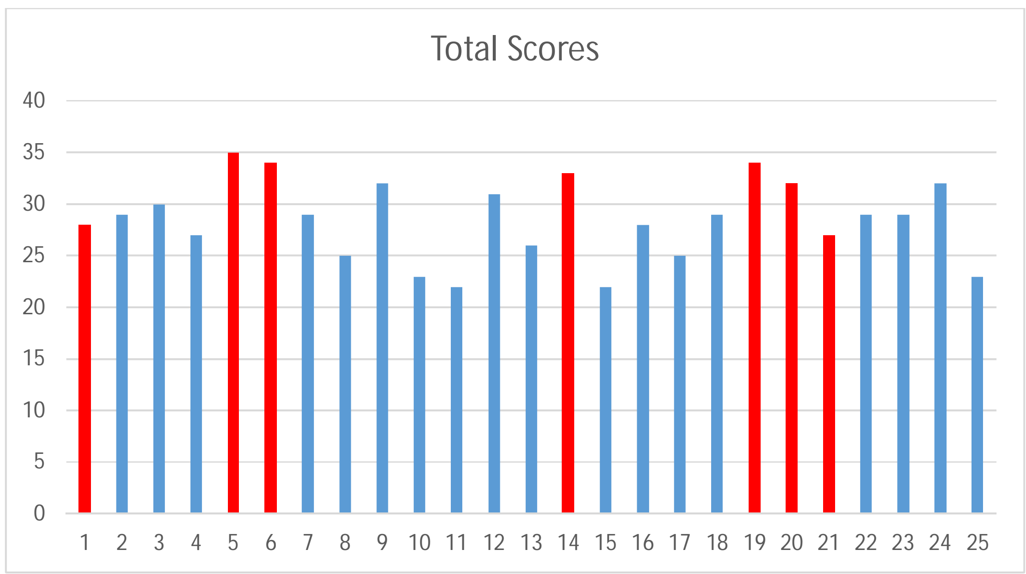
одговора „имам неутралан став“ и одговора „не слажем се“. У складу са тим, приликом рачунања скора за појединачно питање, односно за појединачну дефиницију одговорности у развоју корисничког интерфејса коришћена је следећа формула:

$$T(s) = 1 * A(s) + 2 * B(s) + 3 * C(s) + 4 * D(s) + 5 * E(s)$$

Овде је:

- A(s) број кандидата који су одговорили „потпуно се не слажем“ за исказ *s*,
- B(s) број кандидата који су одговорили „не слажем се“ за исказ *s*,
- C(s) број кандидата који су одговорили „имам неутралан став“ за исказ *s*,
- D(s) број кандидата који су одговорили „слажем се“ за исказ *s* и
- E(s) број кандидата који су одговорили „потпуно се слажем“ за исказ *s*.

У табели 7, у колони „укупан скор“ приказани су резултати за сваки исказ из упитника. Како би се резултати лакше могли упоредити, на слици 4 резултати упитника су приказани графички (вредности које одговарају дефиницијама из истраживања описаног у овој дисертацији обележене су црвеном бојом).




Слика 4. Графички приказ резултата упитника (хоризонатална оса: редни број питања из упитника; вертикална оса: број освојених поена сваког исказа / питања).

Са циљем да се провери ниво слагања кандидата (енгл. *inter-rater agreement*), спроведен је прорачун коефицијента *kappa*[10]. С обзиром на чињеницу да основни прорачун овог коефицијента мери слагање између само два кандидата, у прорачуну је коришћен генерализовани приступ који подржава више кандидата [1]. С обзиром на то да целокупан прорачун овог коефицијента излази из оквира ове тезе, овде ће бити представљени само коначни резултати. Наиме, добијена вредност генерализованог коефицијента *kappa* је 0,15. Ово се може посматрати као „благо слагање“ између кандидата [20].

На основу добијених резултата упитника и коефицијента *kappa* може се закључити да се на основу двадесет пет питања из упитника кандидати у мањој мери ипак слажу. Такође, може се закључити да дефиниције одговорности изложене у овој дисертацији програмери генерално виде као прихватљивије и интуитивније од дефиниција предложених у другој литератури. Према добијеним резултатима, програмери јасно подржавају дефиниције

[10] http://en.wikipedia.org/wiki/Cohen's_kappa

валидације (укупан скор: 35), стилизације (34), понашања (33), форматирања података (34) и приступа подацима (32). Истовремено, дефиниције распореда (28) и структуре (28) препознате су делимично.

Са друге стране, међу осамнаест екстерних дефиниција одговорности постоји неколико које су програмери препознали и јако подржали. Наиме, програмери виде распоред као „управљање позицијом графичких контрола како релативно једне у односу на другу, тако и у односу на прозор“ (укупан скор 32), док понашање виде просто као „повезивање догађаја и акција“ (32).

# *Ретроспектива технологија за развој корисничких интерфејса*

Са циљем да се испита примењивост предложене класификације одговорности у развоју корисничких интерфејса, анализирани су бројни постојећи апликативни програмски интерфејси за изградњу десктоп, веб и мобилних корисничких интерфејса. Осим примарног циља утврђивања примењивости класификације одговорности, у овом поглављу предложена је и класификација технологија за развој корисничких интерфејса, која је сама по себи својеврстан допринос. У овом поглављу представљени су резултати ове студије.

Фокус спроведене студије је био да се утврди да ли су и како предложене одговорности обухваћене у технологијама за развој корисничких интерфејса. Анализирано је преко 50 технологија у широкој употреби. Технологије су анализиране на бази предложене класификације одговорности. За сваку од технологија, односно за сваку од група технологија, грубо су приказани начини на које се адресирају све одговорности из предложене класификације.

Ово поглавље конципирано је као кратак преглед технологије развоја корисничких интерфејса од времена кад су персонални рачунари и десктоп апликације са графичким корисничким интерфејсима ушли у широку употребу (почетком деведесетих година 20. века) до данас. Елаборирају се промене и кретања у развоју корисничких интерфејса у току тог периода и дискутују текући проблеми и потенцијални правци будућег развоја. Према томе, овај преглед даје ретроспективу главних категорија или праваца у развоју корисничких интерфејса.

Анализиране су следеће групе технологија:

- технологије оријентисане ка формуларима (енгл. *form-oriented*),
- технологије базиране на шаблонима (енгл. *template-based*),
- технологије базиране на принципима објектне оријентације (енгл. *object-oriented*),
- хибридне технологије
- технологије базиране на декларативним језицима
- технологије за развој базиране на моделима (енгл. *model-driven*) и

- генерички приступи за развој, односно за генерисање корисничких интерфејса.

За сваку категорију технологија дат је кратак опис и идентификоване су заједничке карактеристике и најпознатији представници. Као што је већ речено, дискутовани су и начини на које свака група технологија покрива одговорности из предложене класификације одговорности.

### Технологије оријентисане ка формуларима

Технологије базиране на формуларима, чврсто спрегнутим са релационим моделима података, представљају групу врло успешних и у индустрији често примењиваних технологија [24]. Њихов приступ изградњи корисничких интерфејса за апликације са релационим базама података подразумева организацију корисничког интерфејса у виду формулара за унос података. Формулари приказују вредности колона слогова из базе података у одговарајућим графичким елементима као што су поља за текст (енгл. *text box*), листе (енгл. *list box*), поља за потврду (енгл. *check box*), табеле (енгл. *grid*), и сл. Алати и друге компоненте ових технологија омогућавају генеричку навигацију кроз формуларе, као и директну спрегу графичких елемената са базом података. Програмер је ослобођен бриге о закључавању података, трансферу, трансформацији и ажурирању података у формуларима. На пример, када корисник промени селекцију слога у главном делу формулара (енгл. *master-details form*), вредности у делу формулара са детаљима се аутоматски освежавају захваљујући механизму већ уграђеном у формулар. Слично томе, када корисник мења вредност у графичком елементу спрегнутом са одговарајућом колоном табеле базе података, нова вредност се аутоматски снима у табелу. За ову врсту понашања није потребно мануелно програмирање осим декларативног спрезања графичког елемента или целог формулара са базом података. У контексту предложене класификације, одговорност приступа подацима је снажно подржана јер технологија пружа униформан поглед на развој апликација и јаку, декларативну спрегу са базом података. За дефинисање структуре, распореда и стилизације, ове технологије нуде алате за дизајн формулара, као и специфичне објектно-оријентисане апликативне програмске интерфејсе. Понашање се решава на класичан начин, уз помоћ парадигме обраде догађаја (енгл. *event-handlers*), док се за валидацију уноса и акција корисника користе изрази (енгл. *expressions*), укључујући и регуларне изразе (енгл. *regular expressions*). Међу многим технологијама из ове групе издвајају се технологије *Microsoft Access*, *Oracle Forms* и *Oracle Application Express – APEX* као типични представници.

### Технологије базиране на шаблонима

Као последица уласка веба у широку употребу појавио се читав низ технологија за развој корисничких интерфејса које се базирају на текстуалним шаблонима. Главна мотивација за приступ базиран на шаблонима јесте флексибилан начин изградње корисничког интерфејса за веб окружење. На самом почетку, ове технологије су диктирале развој помоћу шаблона који дефинишу структуру и распоред графичких елемената. За приступ подацима у споменуте шаблоне уграђивани су делови програмског кода нпр. на језику *Java, PHP, Python, C#*. Мешање шаблонског и програмског кода у веб страницама (на пример, уграђивање шаблонског кода у петље са циљем добијања итеративних графичких елемената као што су листе, табеле, стабла итд.) имало је за последицу честу промену синтаксног контекста. Овакав начин синтаксно хетерогеног програмирања показао се као напоран за програмере. Из тог разлога у ове технологије уведени су посебни тагови (енгл. *tag*) и језици израза (енгл. *expression language*). Циљ је био да се у шаблонима замени програмски код а) шаблонским таговима за стандардне функционалности веб апликација, б) изразима за приступ подацима из апликације и функцијама смештеним у пословним

објектима (енгл. *business objects*). Типичан представник ове групе технологија је технологија *JavaServer Pages – JSP*. Библиотека тагова ове технологије *JSP Standard Tag Library – JSTL* подржава манипулацију документима на језику *XML*, итерације, условне изразе, приступ подацима у релационој бази података, интернационализацију итд. Ова технологија има и свој посебан језик израза назван *Expression Language – EL*. Овај језик служи као одвојен механизам за приступ подацима са једноставном нотацијом за приступ апликационим објектима, њиховим атрибутима и операцијама. Много других технологија приступа развоју корисничких интерфејса за веб на овај начин. Све оне су базиране на шаблонима за дефинисање структуре и распореда, користе језике израза за приступ подацима, док се динамички аспекти корисничког интерфејса постижу класичним начином обраде догађаја користећи апликационе програмске интерфејсе описане у стандардима *ECMAScript*[11] и *Document Object Model – DOM*[12]. Форматирање података се решава помоћу библиотека тагова специфичних за сваку технологију. Стилизација корисничког интерфејса се остварује готово увек на бази стандарда *CSS*. Типични представници ових технологија су *ASP, PHP, Apache Struts, WebWork, Struts2, Spring MVC, Spyce (Python Server Pages), Ruby on Rails* и многе друге.

### Технологије базиране на принципима објектне оријентације

Значајан део технологија за развој корисничких интерфејса ослања се на основне принципе објектне оријентације. Ове технологије најчешће долазе у виду библиотека графичких елемената. Њихова највећа предност јесте могућност компоновања графичких елемената у веће, поновно употребљиве делове корисничког интерфејса и интуитивно и флексибилно управљање догађајима. Конкретно, све врсте понашања и интеракције графичких елемената дефинишу се посебним методама за обраду догађаја које су заправо обичне методе циљног програмског језика за који је дата технологија за развој корисничких интерфејса намењена. Сви аспекти развоја се решавају употребом објектних апликативних програмских интерфејса специфичних за дату технологију. Технологије *Visual Basic, MFC, AWT, Swing, SWT, Delphi, Google Web Toolkit, Cocoa Touch UIKit, Vaadin,* и многе друге технологије сврставају се у ову групу.

Иако спада у исту групу, библиотека *Qt* [36] доприноси овој групи својим оригиналним концептима. Упркос значајној комплексности интеракције између делова структуре корисничког интерфејса, друге технологије нису препознале потребу за посебним механизмима и концептима осим класичног приступа обради догађаја. Са друге стране, технологија *Qt* нуди већ споменути механизам сигнала и слотова. Приликом сваког догађаја из одговарајуће компоненте се емитује сигнал. Слот је метода која се позива као последица појаве неког сигнала. Сигнали и слотови се повезују међусобно на декларативан начин. Сигнал се може повезати декларацијама са произвољно много слотова, и обрнуто. Графички елемент који емитује сигнал не зна, односно није директно спрегнут са елементом чији ће слот бити позван. На овај начин, графички елементи (компоненте) су само лабаво спрегнути декларацијама веза између сигнала и слотова. Овакав механизам побољшава енкапсулацију и омогућава решавање понашања и интеракције графичких елемената на декларативан начин.

### Хибридне технологије

Хибридне технологије представљају релативно нов тренд технологија за развој корисничких интерфејса у широкој употреби. Ове технологије комбинују шаблоне и језике

---

[11] Стандард *ECMAScript* је основни стандард за неколико језика: *JavaScript, ActionScript, QML, JScript, QtScript, InScript.* http://www.ecma-international.org/publications/standards/Ecma-262.htm

[12] http://www.w3.org/DOM/

израза (који се интензивно користе у технологијама базираним на шаблонима) са објектним апликативним програмским интерфејсима (који се користе у технологијама базираним на принципима објектне оријентације). Технологија *JavaServer Faces – JSF* је типичан представник ове групе технологија. Ова технологија се састоји из а) библиотеке тагова за дефинисање структуре, распореда и форматирања података, б) језика израза за спрезање графичких елемената и догађаја са апликационим објектима и сервисима (приступ подацима) и в) објектним апликативним програмским интерфејсом за управљање стањем графичких елемената, догађајима и за валидацију уноса и акција корисника. Други карактеристични представници ове групе технологија су технологије: *ASP.NET MVC, Apache Wicket, Apache Tapestry, Apache Click*, *ZK* и друге. Такође треба споменути и технологију *Adobe Flex,* која је концептуално блиска овим технологијама и која користи шаблоне за дефинисање структуре и распореда графичких елемената. Слично као код технологије *Qt*, постоји интересантан механизам за управљање понашањем и приступом подацима. Циљни језик ове технологије је језик *ActionScript*.

### Технологије базиране на декларативним језицима

Декларативни приступи су најновији тренд у развоју корисничких интерфејса. За креирање структуре корисничког интерфејса махом се користе нотације *XML* и *JavaScript Object Notation - JSON* и генерално декларативне нотације за адресирање свих аспеката развоја корисничких интерфејса. Технологије из ове групе су се појавиле релативно скоро, заједно са уласком мобилних и „паметних" телефона (енгл. *smart phones*) у широку употребу. За разлику од хибридних приступа који се базирају на веб окружењу, декларативне технологије покривају како мобилне тако и десктоп платформе. Узимајући у обзир њихову тренутну популарност, биће им посвећено мало више пажње у овом делу дисертације.

Технологија *Android,* односно њен апликативни програмски интерфејс за развој корисничких интерфејса могао би се класификовати као објектно оријентисан. Међутим, он доноси додатни апликативни програмски интерфејс базиран на језику *XML* за спецификацију структуре, распореда и стилизације графичких елемената. Декларативни начин спецификације структуре и распореда повећава моћ визуелизације структуре, проналажење грешака, омогућава лакше модификовање без поновне компилације кода и помаже у прилагођавању апликације различитим величинама и оријентацијама екрана. Када је потребно динамички мењати структуру корисничког интерфејса и даље је могуће користити објектни апликативни програмски интерфејс. Приступ подацима није подржан на декларативан начин[13].

Прављење корисничког интерфејса за апликације базиране на оперативном систему *Windows* и за веб апликације на платформама *Windows Platform Foundation – WPF* и *Microsoft Silverlight*, могуће је коришћењем посебног језика *Extensible Application Markup Language – XAML.* Овај језик омогућава дефинисање структуре, распореда и стилизације. Међутим, за разлику од технологије *Android,* подржан је приступ подацима као и дефинисање догађаја у коду.

Компанија *Nokia* тренутно препоручује технологију *QtQuick*, мултиплатформско решење за десктоп и мобилне оперативне системе. Ова технологија укључује декларативни језик *QML* базиран на нотацији *JSON*. Графички елементи се ређају хијерархијски угнежђивањем спецификација конструкције. Стандард *ECMAScript* се користи као језик за програмирање понашања које је олакшано механизмом сигнала и слотова. Ова технологија омогућава

[13] Технологија *Android-binding*, екстерно решење отвореног кода за приступ моделу података, може се пронаћи овде: http://code.google.com/p/android-binding/.

повезивање графичких елемената са моделом података као и употребу машина стања (енгл. *state machines*) за управљање променама вредности атрибута графичких елемената.

Технологија *Enyo* је мултиплатформски апликативни програмски интерфејс базиран на синтакси *ECMAScript*. Он промовише декларативну спецификацију структуре графичких елемената и повезивања догађаја са операцијама за управљање догађајима. Понашање се програмира на језику заснованом на поменутом стандарду *ECMAScript*. Постоје три начина за управљањем догађајима: а) на нивоу једног графичког елемента, б) прослеђивањем догађаја у смеру од компоненте која је примила догађај ка родитељу без директног везивања и в) механизмом за пријављивање на врсту догађаја (енгл. *subscribe mechanism*), такође без директног везивања компонената. Овакав начин управљања догађајима без јаке међусобне спреге графичких елемената доприноси већем степену поновне употребе и енкапсулације фрагмената корисничког интерфејса. Главни допринос ове технологије управо и лежи у моделу енкапсулације који помаже у распоређивању функционалности у независне, самодовољне градивне блокове са јасно дефинисаним интерфејсима који се могу лако поново употребити. Овакав модел форсира апстракцију и адресира све архитектурне нивое корисничког интерфејса. Тим који развија ове технологије тренутно ради на подршци за приступ подацима.

Технологија *XML Window Toolkit - XWT* је још један апликативни програмски интерфејс оријентисан ка декларативним начинима спецификације корисничких интерфејса. Овај програмски интерфејс унифицира неколико програмских интерфејса из породице *Eclipse* мапирајући њихове елементе структуре и распореда на језик *XML*: *SWT, JFace, Eclipse Forms* и други. Приступ подацима решен је уз помоћ језика израза док се понашање програмира на програмском језику *Java*. За екстерну стилизацију графичких елемената користи се технологија *CSS*. Апликације направљене у овој технологији ослањају се на посебну виртуелну машину, што им омогућава да раде у веб окружењу уз помоћ технологија *Java Applet* и *ActiveX*.

Постоји још неколицина апликативних програмских интерфејса у овој групи. На пример, *AmpleSDK,* који користи језик *XUL* као декларативни језик, технологију *CSS* за стилизацију и језик *ECMAScript* за понашање и интеракцију графичких елемената. Програмски интерфејс *Dojo Toolkit* промовише декларативни начин састављања корисничког интерфејса као и напредне механизме распоређивања графичких елемената, приступа подацима и интернационализације.

### Технологије базиране на моделима

Осим већ описаних технологија за развој корисничких интерфејса, постоји велика група технологија која се базира на моделима и језицима за специфичне домене (енгл. *domain-specific languages*). У већини случајева користе се посебни модели корисничког интерфејса, мада постоје и приступи који користе моделе апликације за добијање корисничког интерфејса (они се називају генеричким и биће представљени у наредном одељку). У обе групе, модели се користе или за генерисање корисничког интерфејса, или као основа за интерпретацију у време извршавања.

Преглед прве две генерације окружења за развој корисничких интерфејса помоћу модела (енгл. *model-based user interface environments, MBUIDE*) закључно са 2000. годином дао је Да Силва (*Da Silva*) [8]. Према његовим речима, осим подизања нивоа апстракције, унапређења систематичности и аутоматизације метода за дизајн и имплементацију корисничких интерфејса, технологије из овог периода нажалост нису успеле у решавању проблема интеграције корисничког интерфејса са апликацијом, нити у постизању консензуса око скупа модела који би био најприкладнији за описивање корисничког

интерфејса. У контексту предложене класификације одговорности, ове технологије нису понудиле решење за одговорност приступа подацима нити су пружиле јединствен скуп модела који адресирају остале одговорности.

У скорашњем раду Мајкснера (*Meixner*) и његових колега [23] елабориране су четири генерације приступа за развој корисничких интерфејса базираних на моделима од 1990. године па до данас. Према ауторима ове студије, данас постоји сагласност око нивоа апстракције и типа модела за развој корисничких интерфејса, али још увек нема консензуса ни стандарда који прецизно дефинишу семантику ових модела. Дакле, основни модели су модел задатака, модел дијалога и модел презентације (енгл. *task model, dialog model, presentation model,* респективно). Модел презентације подразумева дефинисање структуре, распореда и стилизације. Модел задатака обухвата приступ подацима тако што за сваки задатак (енгл. *task*) дефинише графичке елементе и доменске апликационе објекте са којима манипулише. Модел дијалога дефинише аспект понашања у корисничком интерфејсу. Најистакнутији пример модела задатака су „стабла конкурентних задатака“ (енгл. *ConcurrentTaskTrees, CTT*) [32]. Ова технологија је потпомогнута језиком *MARIA* [39] који пружа остале моделе корисничког интерфејса. Заједно са одговарајућим развојним окружењима *CTTE* и *MARIAE* ове технологије нуде комплетан приступ развоју корисничких интерфејса базиран на моделима.

Осим модела корисничких интерфејса базираних на задацима, постоји велики број приступа који се базирају на језику *UML*, моделу ентитета и веза (енгл. *entity-relationship*), или сличним моделима. Типичан пример употребе профила језика *UML* у развоју корисничких интерфејса пословних апликација предложио је Милосављевић са својим колегама [29]. Треба споменути и друге представнике као што су *WebRatio*[14], *UMLi*[15] [9], *Intelliun Virtual Enterprise*[16] и *SOLoist*[17] [28] који су више пута примењивани у индустријским пројектима са информативним веб сајтовима, доступним извршним верзијама и помоћним алатима.

### Генерички приступи

Мањи, али значајан подскуп технологија за развој корисничких интерфејса представљају оне које омогућавају генерисање корисничког интерфејса из модела података, задатака или из других модела (али не непосредних модела корисничког интерфејса). Ове технологије најчешће подразумевају потпуни или полуаутоматски процес генерисања корисничког интерфејса из модела домена. Због тога се овакав приступ понекад назива аутоматско генерисање корисничког интерфејса (енгл. *automatic user interface generation*). У неким случајевима, уместо генерисања корисничког интерфејса, модели се интерпретирају у време извршавања апликације. У сваком случају, захваљујући аутоматизацији у процесу добијања корисничког интерфејса, овакви приступи штеде време и смањују количину грешака (енгл. *bug*). При томе, добијени кориснички интерфејси су јасно и униформно структурирани. Постоје, међутим, генерални проблеми са оваквим приступом јер се њима значајно смањује флексибилност добијених корисничких интерфејса. Овакви приступи пате и од синдрома „кратког домета“ (енгл. *low ceiling*) и непредвидивости резултата процеса генерисања [33]. Ипак, показало се да због своје јаке спреге и приступа подацима овакве технологије могу бити врло успешне и употребљиве. У том смислу, ове технологије показују евентуални будући правац кретања за технологије базиране на моделима, а пре

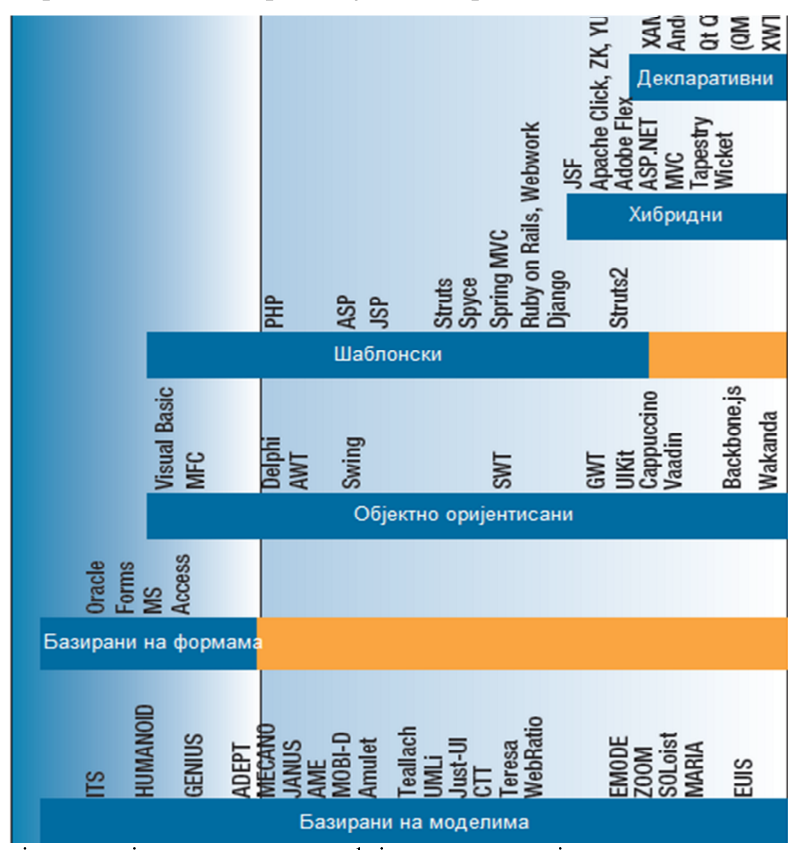
[14] http://www.webratio.com
[15] http://trust.utep.edu/umli/
[16] http://www.intelliun.com
[17] http://www.soloist4uml.com

свега за технологије у широкој употреби. На пример, технологија „огољених објеката“[18] (енгл. *Naked Objects*), показала се као врло успешна у реалним системима. У литератури се могу пронаћи и други генерички приступи [7, 26, 31].

Под условима конкретног и ограниченог домена примене, генерички приступи имају перспективу. Изазови који предстоје подразумевају проналажење нових домена примене, унапређење техника моделовања, као и истраживање нових начина комбиновања модела са циљем унапређења лакоће употребе [35].

## Критика, трендови и изазови

На бази времена појављивања, као и периода доминантног коришћења све споменуте категорије технологија за развој корисничких интерфејса су приказане на временској скали на слици 5. У табели 8 су приказани начини на које свака од група технологија покрива сваку од одговорности из предложене класификације одговорности.




Слика 5. Категорије технологија за развој корисничких интерфејса на временској скали.

| Аспект / Технолог. | Структура и распоред | Приступ подацима | Понашање | Формат. Података | Стилизација | Валидација |
|---|---|---|---|---|---|---|
| | К Д | S Д | К И | К И | К И | К Д/И |
| **Базиране на формама** | Дизајнери форми, конкретни објектни програмски | *SQL*, спрезање контрола са базом података | Обрада догађаја на циљном језику и конкретни објектни | Циљни језик | Дизајнери форми, конкретни објектни програмски | Конкретни језици израза и управљање догађајима на |

[18] http://en.wikipedia.org/wiki/Naked_objects

| | интерфејси | | | | програмски интерфејси | | | | интерфејси | | циљном језику | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Шаблонске технолог.** | С | Д | К | Д | С | И | К | Д/И | С | Д | С | И/Д |
| | *(X)HTML*, *CSS*, конкретне библ. тагова | | *EL*, *OGNL*[19], циљни језик | | Обрада догађаја, *ECMA*, *DOM* програмски интерфејси | | Конкретне библ. тагова, циљни језик | | *CSS*, *DOM* програмски интерфејси | | *ECMA*, *(X)HTML* атрибути | |
| **Објектне технолог.** | К | И | К | И | К | И | К | И | К | И | К | И |
| | Конкретни објектни програмски интерфејси | | Циљни језик | | Обрада догађаја на циљном језику и конкретни објектни програмски интерфејси | | Конкретни објектни програмски интерфејси, циљни језик | | Конкретни објектни програмски интерфејси, *CSS* (*GWT*, *Vaadin*) | | Циљни језик, конкретни објектни програмски интерфејси | |
| **Хибридне технолог.** | С | Д | К | Д | С | И | К | Д/И | С | Д | С | И/Д |
| | *(X)HTML*, *CSS*, конкретне библ. тагова, конкретни објектни програмски интерфејси | | *EL*, *UEL*[20], *OGNL*, *MVEL*[21], циљни језик | | Обрада догађаја, *ECMA*, *DOM* програмски интерфејси, циљни језик, конкретни објектни програмски интерфејси | | Конкретне библ. тагова, циљни језик | | *CSS*, *DOM* програмски интерфејси, конкретни објектни програмски интерфејси | | *ECMA*, *(X)HTML* атрибути, конкретни објектни програмски интерфејси | |
| **Деклара-тивне** | С | Д | К | Д | С | И | С/К | Д/И | С | Д | С/К | И |
| | XML, ECMA, JSON, (X)HTML, XUL[22], XAML | | Програмски интерфејси за приступ подацима и језици израза, циљни језик | | Обрада догађаја, *ECMA*, *DOM* програмски интерфејси, циљни језик | | *ECMA*, програмски интерфејс за приступ и форм. података, циљни језик | | *CSS*, *ECMA*, *DOM* програмски интерфејси, конкретни објектни програмски интерфејси | | ECMA, циљни језик | |
| **Базиране на моделима** | К | Д | К | Д | К | Д | К | И/Д | К | Д | К | И/Д |
| | *DSL*[23], *UML* профили, модели задатака | | *DSL* | | Дијаграми активности, машине стања, *UML* профили, *DSL*, циљни језици | | Непознато | | *UML* профили, *DSL*, *CSS* | | Регуларни изрази, извођење из модела домена, циљни језици | |

Табела 8. Начини покривања одговорности у развоју корисничких интерфејса (С – базирано на стандардима, К – конкретни, специфични начин, Д – декларативни начин, И – императивни начин).

Кад је у питању развој веб базираних корисничких интерфејса, најновија историја технологија у широкој употреби показује два паралелна, али супротстављена правца. Приступи базирани на шаблонима бивају потиснути приступима који нуде објектно оријентисане апликативне програмске интерфејсе. То је због а) допуне шаблонских приступа објектним програмским интерфејсима што је случај са хибридним технологијама

[19] *Object-Graph Navigation Language - OGNL*: http://commons.apache.org/ognl/
[20] *Unified Expression Language - UEL*: http://download.oracle.com/otndocs/jcp/jsp-2.1-fr-eval-spec-oth-JSpec/
[21] *Powerful expression language - MVEL*: http://mvel.codehaus.org/
[22] *XML User Interface Language - XUL*: https://developer.mozilla.org/en-US/docs/XUL
[23] Језик специфичан за домен: *Domain-Specific Language - DSL*

или б) потпуне замене шаблонских приступа технологијама које нуде искључиво објектне програмске интерфејсе као што су технологије *Google Web Toolkit – GWT* и *Vaadin*. Овакав тренд је врло логичан ако се узме у обзир а) супериорност објектних језика у односу на шаблонске (наслеђивање, полиморфизам, енкапсулација, параметризација, поновна употреба итд.), б) потреба за моћним и експресивним концептима за изградњу великих и комплексних структура корисничких интерфејса и в) успех објектних програмских интерфејса у десктоп ери.

Са друге стране, уместо императивних, објектних програмских интерфејса, тренутно су ипак актуелни декларативни приступи за развој корисничких интерфејса. Технологије *HTML, XML, XPath, CSS, JSON* и сличне декларативне нотације преузимају примат у развоју корисничких интерфејса. С обзиром на то да је највећи део типичне структуре корисничког интерфејса статичан, декларативни приступи су врло употребљиви у спецификацији таквих делова. Они доносе бољу прегледност програмског кода, али и делимично одвајање одговорности структуре и распореда од стилизације и понашања. Ипак, понашање и динамични аспекти корисничких интерфејса се још увек програмирају углавном уз помоћ императивних језика и класичне парадигме обраде догађаја. Из овог разлога, објектни апликативнни програмски интерфејси још увек трају.

Може се уочити још један тренд: свеопшта оријентација ка стандардним технологијама и платформама. Стандарди *XML* и *ECMAScript* се интензивно користе и врло су популарни. У исто време, нестандардне технологије, поготово из групе технологија које се базирају на моделима, боре се за простор у великим технолошким стандардима.

Пред произвођаче технологија за развој корисничких интерфејса се тако поставља неколико будућих изазова. Један је посебно критичан за технологије базиране на шаблонима и за хибридне технологије. Овим технологијама недостају концепти високог нивоа апстракције за спецификацију архитектуре и распоређивање страница у веће енкапсулиране модуле са јасно дефинисаним интерфејсима. Узимајући у обзир чињеницу да данашње пословне апликације имају по неколико стотина страница, јасан архитектурни поглед на систем је апсолутно неопходан. У овој дисертацији, биће посебно обрађен управо овај проблем. Од постојећих технологија, технологије *Enyo* и *WebML* до одређеног степена препознају и решавају овај проблем.

Иако су флексибилност, проширивост и подршка развојних алата највећи адути технологија у широкој употреби, низак ниво апстракције, као и слаба изражајност су највећи недостаци који се морају елиминисати у будућности како би се повећала продуктивност програмера. Кад су у питању технологије које се базирају на моделима, највећи изазов јесте избегавање преузимања семантике нивоа имплементације у моделима корисничких интерфејса. У супротном, апстрактни модели могу постати комплексни као сама и имплементација.

Коначно, можда и највећи изазов тренутно представљају захтеви адаптације корисничких интерфејса различитим контекстима употребе (енгл. *multi-context, multi-target*). Контекст подразумева: а) корисника апликације, б) софтверско-хардверску платформу и в) комплетно физичко окружење у ком корисник интерагује са апликацијом [5]. Неки резултати у овом домену већ постоје. 2003. године предложен је јединствени свеобухватни оквир за процесе, моделе и методе за изградњу корисничких интерфејса који се адаптирају према контексту [5]. Такође, једна скорашња студија сумира десет димензија дизајна алата и апликација за вишеконтекстна окружења у форми логичког оквира [38]. Ова студија помаже да се боље разумеју оваква окружења, као и да се постојећи истраживачки напори могу лакше међусобно упоредити. Врло често данашње апликације занемарују корисника и

окружење и одговарају само на контекст софтверско-хардверске платформе, на пример, прилагођавајући кориснички интерфејс величини и облику екрана.

На крају, присутан је и проблем пролиферације различитих уређаја и платформи који неодољиво подсећа на десктоп еру са мноштвом нестандардних технологија за развој десктоп корисничких интерфејса. Стандард *HTML5* још увек не успева да разреши постојећи проблем, највише из разлога ограничене подршке, односно ограниченог апликативног програмског интерфејса према хардверским уређајима као што су камере, фото апарати, навигациони уређаји итд. Борба са свеприсутном разноликошћу уређаја и платформи описана је у једном новијем чланку о корисничким интерфејсима за мобилне телефоне [45].

# Део 3: Принцип хијерархијске декомпозиције

# *Основни концепти решења*

У претходним поглављима постављен је генерални оквир који дефинише основне одговорности у развоју корисничких интерфејса. Показано је и како се дошло то предложеног скупа одговорности, али и како је свака од одговорности обухваћена у технологијама у широкој употреби. На основу изложеног скупа одговорности, предложена је и категоризација технологија за развој корисничких интерфејса која групише технологије које на сличан или исти начин покривају одговорности из класификације одговорности. На крају, изнета је и дискусија и критика тренутног стања у развоју, опет кроз призму оквира, односно класификације одговорности.

У овом и наредном поглављима биће предложена технологија за развој корисничких интерфејса која решава неке од споменутих проблема и адресира све споменуте одговорности на адекватан начин. Нагласак решења које ће бити представљено је на подскупу одговорности за који се може рећи да није адекватно подржан у већини доступних технологија, а не на свим одговорностима из предложене класификације и свим проблемима који су споменути.

Најскорији трендови у развоју софтвера базираном на моделима охрабрују моделовање у високо апстрактним језицима као што је језик *UML* [26, 43, 50]. При томе, задржавају се предности јаке и формалне спреге између артефаката високог нивоа (нпр. поглед на архитектуру или дизајн система) и извршивих артефаката ниског нивоа. Да Силва и Патон (*Da Silva, Paton*) наглашавају да коришћење модела као саставни део развоја корисничких интерфејса помаже у усвајању захтева корисника, ослобађа од прераног везивања за конкретне распореде и типове графичких елемената, а везу између различитих делова корисничког интерфејса чини експлицитном [9]. У овој дисертацији прати се управо овај правац и предлаже се нови приступ моделовању и изградњи корисничких интерфејса пословних апликација. Циљ је да се, између осталих, постигну следеће карактеристике развоја:

1. Моделовање корисничког интерфејса са свим најважнијим особинама извршивих модела [43]. У предложеном приступу користиће се извршиви модели, што значи да се извршиви кориснички интерфејс добија из модела без икакве мануелне трансформације. Користиће се експресивни и декларативни концепти за решавање природне комплексности корисничког интерфејса. Предложено решење помаже у

смањењу напора који се морају уложити у развој, а пре свега у одржавање и то уз помоћ декларативног спрезања фрагмената корисничког интерфејса које замењује традиционални императивни начин обраде догађаја.

2. Поштовање принципа апстракције. Предложени приступ омогућава третирање како појединачног графичког елемента тако и произвољно великог и сложеног фрагмента логички спрегнутих графичких елемената као апстракције.
3. Употреба принципа поновне употребе (енгл. *reuse*). Предложена техника омогућава поновну употребу већ дефинисаних фрагмената корисничког интерфејса уз опциону измену структуре и понашања уз помоћ параметризације и наслеђивања са редефинисањем.
4. Предложено решење подржава енкапсулацију тако што јасно идентификује и одваја интерфејс апстракције и енкапсулира имплементацију (односно интерну структуру и понашање). За разлику од других техника енкапсулације, овде се поред изолације унутрашњости фрагмента од његове спољашњости, изолује и спољашњост од унутрашњости, чиме се додатно смањују међузависности делова корисничког интерфејса. Исти принципи декомпозиције се могу примењивати на свим нивоима детаља модела, од највиших нивоа архитектуре, до најнижег нивоа моделовања појединачних графичких елемената.
5. Правилна спрега корисничког интерфејса са пословном логиком и моделом података која подразумева:
    a. семантичку компатибилност слоја корисничког интерфејса и слоја података и
    b. лабаву спрегу ова два слоја са циљем забране мешања логике једног и другог.

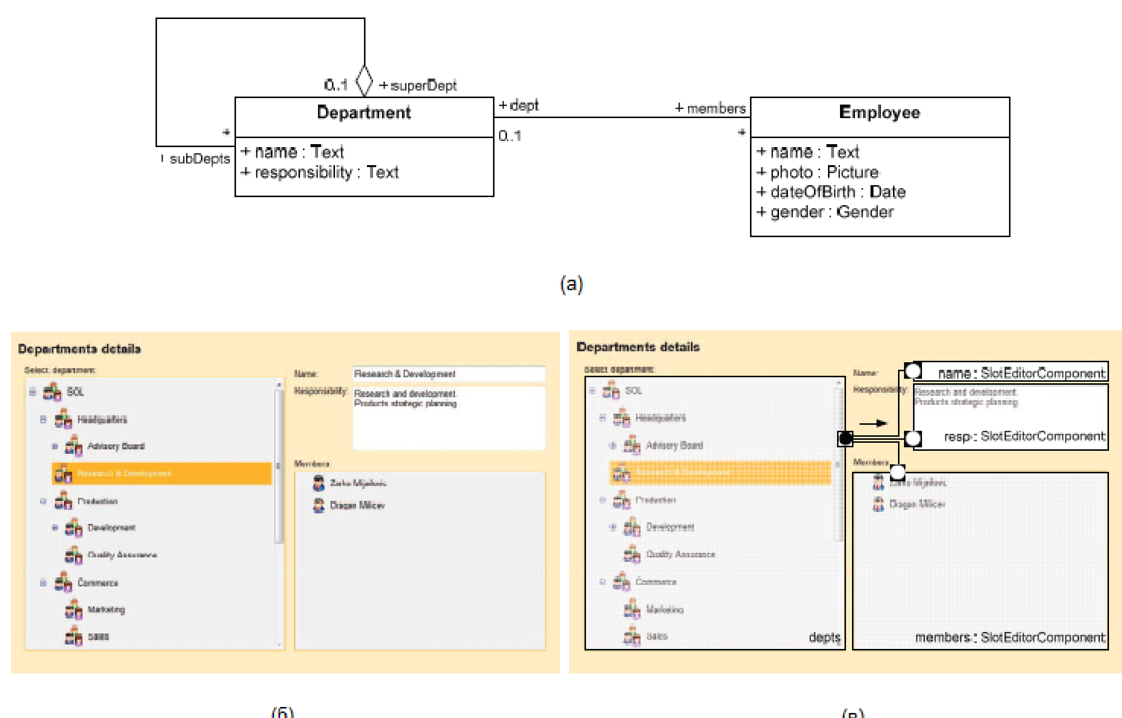
Наведена листа свакако није комплетна листа свих особина које једна техника за моделовање корисничког интерфејса треба да има. Међутим, ово јесу најважније карактеристике решења предложеног у овој дисертацији.

Предложена техника за развој корисничког интерфејса део је извршивог профила језика *UML* за моделовање објектно оријентисаних информационих система (енгл. *object-oriented information systems, OOIS*) названог *OOIS UML* и у књизи [26]. Она је инспирисана концептима осмишљеним у једној другој техници моделовања у потпуно другом домену [44], која је уврштена и у стандард *UML* [37].

Техника која је описана у овој дисертацији намењена је за израду пословних апликација великих размера фокусираних на приказ података у којима су софтверска архитектура, дизајн, ефикасност развоја и одржавања и структурираност важнији од естетског изгледа и атрактивности као што је случај са веб сајтовима и порталима. С друге стране, веб сајтови и портали имају своје специфичне карактеристике и аспекте који су подржани другим технологијама које су ортогоналне на технику која ће овде бити описана. У наставку текста биће описана мотивација и основна идеја предложеног решења на неформалан начин. Потом су детаљније елаборирани структура, понашање и приступ подацима. Остале одговорности неће бити посебно разматране.

### Идеја

Слика 6 илуструје главну идеју предложеног решења. На слици 6a приказан је исечак модела података на језику *UML* неке замишљене пословне апликације. Овај модел описује хијерархијску структуру одељења неког предузећа као и запослене и њихову припадност одељењима.




Слика 6. Једноставан извршив пример корисничког интерфејса. а) Исечак модела података на језику *UML*. б) Пример формулара са спрегнутим графичким елементима. в) Функционална спрега графичких елемената из перспективе размене порука.

Један једноставан фрагмент корисничког интерфејса, односно формулар из апликације приказан је на слици 6б. Стабло са леве стране слике је подешено да приказује хијерархију одељења у предузећу. Свако одељење је објекат класе *Department* из модела са слике 6а. Три графичка елемента (контроле) на десној страни слике 6б приказују особине одељења изабраног у стаблу са леве стране. Прва два графичка елемента су текстуална поља која приказују, али и нуде могућност измене вредности атрибута *name* и *responsibility* класе *Department*. Трећи графички елемент је листа која приказује све запослене у предузећу који су чланови одељења изабраног у стаблу са леве стране, односно, приказује скуп објеката класе *Employee* који су везани за изабрани објекат класе *Department* преко асоцијационог краја *members*.

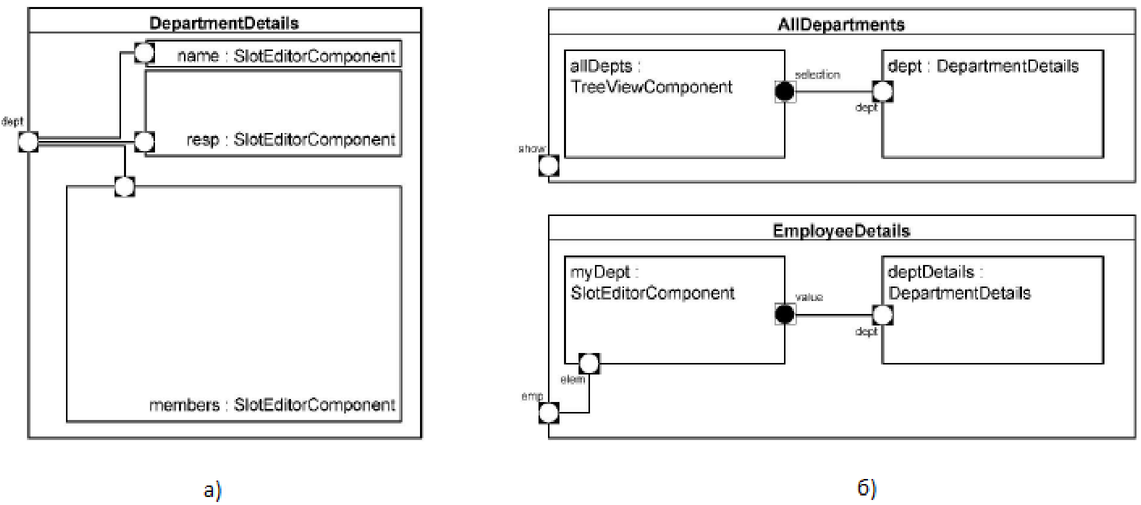
Координирано понашање приказаних графичких елемената може се описати на следећи начин. Кад год је у стаблу са леве стране изабран неки објекат класе *Department*, сваки од три графичка елемента са десне стране треба да прикаже вредност одговарајућег атрибута/асоцијације. Очигледно је да изабрано одељење мора да буде динамички прослеђено као улазни параметар сваком од три графичка елемента. Сваком од три графичка елемента може се доделити одговарајући атрибут чију вредност тај елемент треба да прикаже. То се може учинити још за време моделовања, односно конструкције корисничког интерфејса одговарајућом конфигурацијом датог елемента. Једино што се мења за време извршавања корисничког интерфејса (као последица акције корисника апликације) јесте инстанца, односно објекат класе *Department* чији атрибути треба да се прикажу.

На основу овога се може замислити функционална спрега графичких елемената као на слици 6в. Графички елементи могу се логички сматрати компонентама са "*пиновима*" који формирају њихов интерфејс. Пин је тачка везивања графичког елемента са осталим

графичким елементима преко које елемент прима и шаље сигнале или податке. У споменутом примеру, стабло има један излазни пин преко ког шаље објекат (референцу), односно одељење изабрано од стране корисника, и то сваки пут кад корисник промени селекцију. Сваки од три графичка елемента са десне стране има један улазни пин који прима објекат (референцу) за који треба да прикаже вредност одговарајућег атрибута. Интерно понашање ових елемената обезбеђује да сваки пут када се појави нови објекат на улазном пину, графички елемент приступа објектном простору домена, односно чита и приказује вредност одговарајућег атрибута објекта пристиглог на улазном пину.

Да би обезбедио овакав начин функционисања графичких елемената, програмер једино треба да повеже излазни пин стабла са улазним пиновима остала три графичка елемента помоћу такозваних „жица“ (енгл. *wire*). Жице представљају везе кроз које „пролазе“ подаци од излазног ка једном или више улазних пинова, и то сваки пут кад се на излазном пину појави нова вредност.

У складу са објашњеним приступом, осмишљена је апстрактна и генерална техника моделовања у којој појединачни графички елементи или цели фрагменти логички груписаних и спрегнутих графичких елемената представљају апстракције које се моделују уз помоћ концепта *капсуле* (енгл. *capsule*). Капсула је у моделу представљена класом са опционом интерном структуром. Интерна структура се састоји од других капсула као под-елемената и жица (енгл. *wire*) које повезују њихове пинове. На пример, цела група графичких елемената на десној страни слике 6в може да се сматра компактном целином, односно фрагментом одговорним за приказивање детаља одељења са улазног пина. Такав фрагмент се може поново употребити на другим местима у корисничком интерфејсу апликације. Према томе, ова група графичких елемената формира апстракцију која се моделује капсулом *DepartmentDetails*, чија је дефиниција приказана на слици 7а[24]. На овај начин дефинисана капсула се лако може поново употребити, односно референцирати у дефиницијама других капсула, као што је илустровано на слици 7б.




Слика 7. а) Капсула је структурирана класа (енгл. *structured class*) која апстрахује скуп логички и по понашању груписаних под-елемената. б) Овако дефинисана капсула се може поново употребљавати, односно референцирати из других капсула и самим тим се појављивати на различитим местима у корисничком интерфејсу.

---

[24] Због боље прегледности статичке лабеле које се појављују на формулару нису приказане у дефиницији капсуле.

Једно такво појављивање је у капсули *AllDepartments* (слика 7б). Друго појављивање је приказано на другом формулару моделованом капсулом *EmployeeDetails* на слици 7б, а која приказује детаље објекта класе *Employee* са свог улазног пина. Једна од особина запосленог јесте одељење у ком тај запослени ради и које је приказано компонентом *myDept*. Ова компонента приказује вредност слота *dept* (видети модел са слике 6а) класе *Employee*, односно одељење у ком ради запослени. Та вредност се преноси на улазни пин *dept* капсуле *deptDetails*. Према томе, фрагмент корисничког интерфејса се дефинише само једном, а може се користити на више места у корисничком интерфејсу. Осим тога, имплементација капсуле се може изменити или проширити изменом дефиниције интерне структуре капсуле. С обзиром на то да је интерна структура енкапсулирана иза интерфејса дефинисаног пиновима капсуле, овакав тип измене неће утицати на окружење капсуле.

Једна очигледна и важна последица овог приступа јесте то што овакви модели корисничког интерфејса имају извршиву семантику. То значи да се извршиви кориснички интерфејс може добити директно, једнозначно и аутоматски из модела капсула. Зато овакав модел не представља само апстрактни архитектурни поглед већ реалну, извршиву спецификацију и прецизну документацију дизајна. Ово има много важних импликација, од којих је једна спречавање синдрома преураног почетка имплементације (енгл. *rush-to-code syndrome*) који није превазиђен у многим другим приступима за пројектовање и моделовање [26, 44].

Представљена парадигма заједно са објашњеним концептима инспирисана је једним старијим приступом за моделовање за други домен примене – објектно оријентисано моделовање у реалном времену (енгл. *Real-time Object-Oriented Modeling, ROOM*) [44]. Аналогија овог домена са доменом корисничких интерфејса није случајна, јер се оба домена баве системима вођеним догађајима (енгл. *event-driven*). Наравно, овим доменима управљају другачији типови догађаја, али су принципи слични. Пошто је већина фундаменталних концепата из технике *ROOM* прихваћена у стандарду *UML2*, парадигма и концепти представљени у овој дисертацији су такође усклађени са тим стандардом [37]. Формалније речено, представљена техника се може посматрати као профил стандарда *UML2* који се односи на моделовање корисничких интерфејса пословних апликација великих димензија. На пример, капсула одговара концепту актера (енгл. *actor*) у техници *ROOM,* односно структурираној класи (енгл. *structured class*) на језику *UML2*. Пинови и жице се зову портови и везе (енгл. *ports, bindings,* респективно) у техници *ROOM*, односно портови и конектори (енгл. *ports, connectors*, респективно) на језику *UML2*. У техници представљеној у овој дисертацији намерно су коришћени другачији називи како би се јасно раздвојила профилисана семантика и сврха споменутих концепата. Комплетна спецификација овог профила дата је као прилог ове тезе.

У наставку текста биће додатно и детаљније елаборирани основни концепти и принципи описане технике. Фокус ће бити на структури, понашању и приступу подацима. Остале одговорности неће бити посебно разматране.

### Структура

Капсула је класа која апстрахује или елементарну (примитивну, атомску) компоненту корисничког интерфејса, или целу групу логички и функционално спрегнутих компонената и других капсула. Она представља интегралан и поново употребљив фрагмент корисничког интерфејса. Капсула има интерну структуру која је дефинисана референцирањем других капсула и повезивањем жица између њихових пинова као на слици 7. Интерне капсуле су практично делови окружујуће капсуле. На пример, *deptDetails* као део интерне структуре капсуле *EmployeeDetails* са слике 7б је референца на капсулу *DepartmentDetails*. Семантика референци капсула је иста као семантика актера језика *ROOM*, односно као семантика делова структурираних класа језика *UML2*. Наиме, сваки пут када се прави инстанца

капсуле *EmployeeDetails,* биће направљена по једна инстанца интерних капсула (класа *SlotEditorComponent* и *DepartmentDetails*) заједно са њиховом интерном структуром (ако постоји), и оне ће бити повезане жицама у складу са дефиницијом интерне структуре капсуле *EmployeeDetails*. Креиране инстанце подкомпонената могу да се референцирају из опсега дефинисаног окружујућом капсулом преко референци *myDept* и *deptDetails*.

Описана подршка за апстракцију је круцијална особина предложене технике за моделовање корисничких интерфејса из неколико разлога. Прво, она нуди исти концепт капсуле за моделовање корисничког интерфејса на (високом) нивоу архитектуре, као и на (ниском) нивоу детаља. Другим речима, моделовање помоћу интерне структуре капсуле је рекурзивно и омогућава примену истог концепта на различитим нивоима детаља у моделу [44]. Друго, дефинисање произвољног фрагмента корисничког интерфејса као капсуле омогућава лаку поновну употребу тог фрагмента на различитим местима у корисничком интерфејсу.

Интерфејс капсуле је дефинисан помоћу пинова. За разлику од нешто напреднијих особина портова у техници *ROOM*, где портови могу да преносе вредности произвољних типова података у складу са протоколима у оба правца, или портова на језику *UML2* који могу да нуде и захтевају интерфејсе који дефинишу било какву врсту интеракције (позив операције или слање порука), у техници из ове дисертације порт је поједностављен и профилисан (и преименован у пин) на следећи начин: пин може да преноси само једну референцу или колекцију више референци ка објектима класа или типова података доменског модела података на језику *UML*. Појављивање референце или колекције референци на пину се третира као сигнал (порука). Сигнал може да има и вредност *null,* што значи да је важна сама његова појава и да она не носи никакве додатне информације. Пин може бити типизиран класом или типом података из модела домена, што значи да тај пин може да преноси само референце на објекте тог конкретног типа (укључујући и подтипове). Пин може бити и нетипизиран, или типизиран заједничким најопштијим типом из модела, што значи да може да преноси референце било ког типа; ово је и подразумевано понашање пина. Пин такође може да има дефинисану мултипликативност која ограничава величину колекције референци коју пин може да пренесе. Подразумевана мултипликативност пина је [0..1]. Пин може бити или улазни или излазни, што значи да може да преноси референце само у једном смеру. Назив „пин“ управо је и уведен због описаних семантичких поједностављења. Поред тога, уведена је и посебна нотација за улазни пин (бели круг у црном квадрату) као и за излазни пин (црни круг у белом квадрату) као што је приказано на слици 7. Ова нотације је преузета из технике *ROOM*.

Интерфејс капсуле дефинисан је помоћу пинова. Он обезбеђује стриктну енкапсулацију капсула: не само да је унутрашњост капсуле заштићена од њене спољашности, већ је енкапсулација обезбеђена и у супротном смеру. Наиме, интерна имплементација капсуле не може директно приступити или зависити ни од једне компоненте из окружења капсуле; она само може примати и слати сигнале/поруке преко својих пинова. Ово гарантује да се капсула може уградити у било које окружење које је усклађено са њеним интерфејсом.

Излазни пин унутрашње капсуле може се повезати са једним или више компатибилних улазних пинова других унутрашњих капсула у оквиру исте окружујуће капсуле. Компатибилност пинова значи да се тип и мултипликативност излазног пина слаже са типом и мултипликативношћу улазног пина. У време извршавања, када инстанца капсуле са излазним пином пошаље сигнал на тај пин, сигнал се прослеђује свим повезаним улазним пиновима других капсула.

Улазни пин из интерфејса окружујуће капсуле може се повезати жицама са једним или више компатибилних улазних пинова капсула из њене унутрашње структуре. На пример,

пин *emp* је тако повезан са пином *elem* у капсули *EmployeeDetails* на слици 7б. У овом случају, у време извршавања, свако појављивање сигнала на пину из интерфејса окружујуће капсуле се одмах прослеђује свим повезаним улазним пиновима из интерне структуре те капсуле. Слично важи и за излазне пинове капсула из интерне структуре који се могу повезати са излазним пином из интерфејса окружујуће капсуле. Прецизна семантика управљања сигналима у смислу тока контроле и синхронизације биће анализирана касније у тексту.

Конектори (као концепт везан за структуриране класе на језику *UML*) између пинова названи су жицама због своје семантичке профилизације. Ова терминологија (капсуле, пинови и жице) није изабрана случајно: семантика капсула и њихова интеракција (слање сигнала преко пинова и жица) је врло слична оној из дигиталних електронских кола. Управо због тога модели корисничких интерфејса из праксе у многоме подсећају на оваква електронска кола.

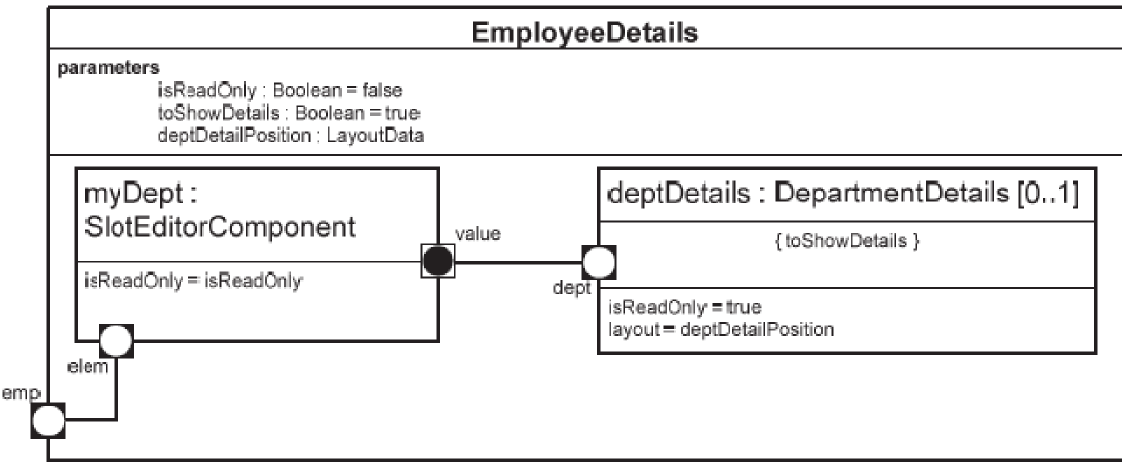
Објашњена парадигма моделовања објеката који сарађују (енгл. *collaborative objects*) има више генералних предности [26, 44]. У контексту развоја корисничких интерфејса, њоме се елиминише потреба за применом одређених пројектних образаца (енгл. *design patterns*) који се често користе како би се смањиле зависности између функционално спрегнутих елемената корисничког интерфејса и унапредио ниво поновне употребљивости, као што су обрасци „убризгавање зависности“ и „посредник“ (енгл. *dependency injection, mediator*, респективно) [10]. Правилна примена пројектних образаца је добра пракса, али она у потпуности зависи од вештине и дисциплине програмера, што може да представља проблем. Због тога је честа последица коришћења класичних технологија за развој корисничких интерфејса врло чврста спрега елемената корисничког интерфејса имплементирана у методама за обраду догађаја (енгл. *event-handlers*). Интеракцију елемената корисничког интерфејса направљених на овај начин је често врло тешко разумети, што може довести у питање и коректност самог понашања: „Највећи број грешака које се појављују приликом тестирања корисничких интерфејса настају у интеракцијама у методама за обраду догађаја“ [52]. Разлог овакве појаве је у томе што су услуге и својства компонената које окружују неку компоненту на неком формулару корисничког интерфејса најчешће директно доступни преко референци из кода метода за обраду догађаја дате компоненте (погледати дискусију везану за пројектни образац „посредник“, енгл. *Mediator* [10]). Са друге стране, техника предложена у овој дисертацији намеће енкапсулацију и слабу спрегу, јер су пинови и жице једини доступни механизам интеракције елемената корисничког интерфејса.

Осим тога, уместо незграпног кода метода за обраду догађаја који служи за функционално спрезање елемената корисничког интерфејса, у представљеној парадигми, програмер, односно моделар, једноставно и елегантно жицама повезује елементе и то на декларативан начин који се може и визуализовати на дијаграмима. Механизам прослеђивања порука се при томе подразумева (уграђен је у имплементацију, односно у извршно окружење корисничког интерфејса) и обезбеђује сво неопходно понашање.

Концепт капсуле апстрахује понављајуће фрагменте корисничког интерфејса у структуре које се могу лако користити на више места. Међутим, фрагменти који се понављају често нису потпуно идентични на сваком месту у корисничком интерфејсу, већ се незнатно разликују од места до места. На пример, фрагмент апстрахован капсулом *EmployeeDetails* са слике 7б ће се можда понављати у корисничком интерфејсу са извесним разликама у интерној структури: компонента *myDept* ће бити потребна али без могућности мењања (енгл. *read-only*), или ће компонента *deptDetails* бити постављена другачије, или ће чак

бити изостављена на неким местима. Да би се подржале овакве могућности, потребно је омогућити дефинисање параметризоване и варијабилне интерне структуре капсуле.

Слика 8 приказује спецификацију параметризованог креирања интерне структуре капсуле *EmployeeDetails*. Ова спецификација има семантику креационих објектних структура [25, 26].




Слика 8. Параметризовани конструктор капсуле *EmployeeDetails*.

У суштини, ово представља спецификацију имплицитног конструктора капсуле. Приликом прављења једне инстанце дате капсуле извршава се конструктор који иницијализује интерну структуру те инстанце капсуле. Параметри конструктора су наведени у секцији за параметре дефиниције капсуле. Осим тога, неки елемент интерне структуре се може креирати условно, ако израз додељен спецификацији елемента има вредност *true*. Ово је случај са интерним елементом *deptDetails*. Условни израз *{toShowDetails}* који му је додељен је тривијални Булов израз који референцира вредност формалног параметра конструктора. Приликом креирања инстанце капсуле *EmployeeDetails,* интерна капсула *deptDetails,* као и све жице повезане са њом биће креирани једино ако параметар конструктора има вредност *true*. Због овога интерна капсула *deptDetails* има мултипликативност [0..1], односно опциона је. Вредности параметара прослеђене конструкторима капсула могу се специфицирати у форми *параметар = израз*, где израз добија вредност у опсегу важења окружујућег конструктора. На пример, вредност параметра *isReadOnly* конструктора капсуле *EmployeeDetails* се једноставно прослеђује истоименом параметру конструктора интерне капсуле *deptDetails*. И други софистицирани концепти креационих спецификација могу бити подржани [25, 26], али се у практичној употреби у домену корисничких интерфејса условно и параметризовано креирање показало као довољно.

## Понашање

У највећем броју случајева композитних капсула, интерне структуре оваквих капсула обезбеђују сво неопходно понашање структуром својих интерних капсула чији су пинови повезани жицама; композитне капсуле не морају да имају додатно, сопствено понашање. Међутим, постоје ситуације када је и то неопходно, и када капсула мора да дефинише сопствено понашање које превазилази понашање њене интерне структуре.

Први такав пример је релативно честа потреба да капсула мења своју унутрашњу структуру након иницијалне конструкције, тако што динамички креира и брише своје унутрашње делове и жице између њих. У најзахтевнијем случају, цела интерна структура капсуле се мора динамички конструисати у складу са одређеним правилима пословне логике. Захтев је, дакле, да капсула буде у стању да динамички мења своју интерну структуру.

Други пример када је неопходно обезбедити сопствено понашање капсула јесте онај када је потребно логички трансформисати поруке које капсула добија на својим улазним пиновима пре прослеђивања тих порука пиновима унутрашњих капсула, и супротно, кад је потребно трансформисати поруке са излазних пинова унутрашњих капсула пре него што се оне проследе на излазне пинове окружујуће капсуле.

Трећи пример представља имплементација примитивних, атомских, односно недељивих капсула које немају интерну структуру у виду других капсула, већ им се имплементација ослања на апликативни програмски интерфејс имплементационе платформе, односно имплементационе библиотеке компонената корисничког интерфејса. Библиотека описана у овој дисертацији садржи десетине уграђених (енгл. *built-in*) примитивних компонената, али често постоји потреба и да се направе примитивне компоненте које су најчешће доста специфичне за сам пројекат или апликацију.

У свим споменутим случајевима, потребно је имплементирати сопствено понашање капсула. Ово понашање мора да оркестрира поруке добијене из различитих извора. Један извор представљају екстерне капсуле; овакве поруке долазе кроз улазне пинове интерфејса саме посматране капсуле. Други извор су интерне капсуле чије се поруке преузимају са њихових излазних пинова. Трећи извор су поруке и догађаји које генерише корисник, као што су притисци мишем или на тастере тастатуре, а које капсула чије сопствено понашање треба имплементирати мора да обради. Коначно, поруке долазе и из објектног простора (позадинске пословне логике) апликације. Ове последње поруке могу бити одговори на захтеве капсуле за читањем података које капсула приказује, или обавештења о изменама у делу објектног простора који је релевантан за капсулу. У општем случају, поруке долазе и обрађују се асинхроно (у непредвидивим тренуцима) и независно (у непредвидивом редоследу). На пример, захтеви за читањем објектног простора могу да се шаљу асинхроно на удаљени сервер као у случају веб базираних апликација које користе механизам *AJAX* за ове потребе.

Због овакве врсте непредвидивости, за моделовање сопственог понашања капсула могу се користити машине стања (енгл. *state machines*). У овом раду преузета је семантика машина стања из технике *ROOM* и језика *UML2*. Прецизније речено, подразумева се семантика „извршавања до краја“ (енгл. *run-to-completion*) за процесирање прелаза стања. Кад једна порука почне да се процесира у машини стања, процесирање се неће прекинути другом поруком у истој машини стања све док се процесирање прве поруке комплетно не заврши, тј. док се машина стања не „смири“ у циљном стању. У случају када понашање капсуле нема стања (енгл. *stateless*), реакције на поруке се моделују као прелази из једног јединог стања у самог себе. У суштини, акције за овакве прелазе стања играју улогу метода за обраду догађаја (енгл. *event handlers*). На тај начин се модел машина стања поједностављује до једноставног модела понашања операција и њихових имплементација.

Услови (енгл. *guard conditions*) и акције на прелазима стања су изрази и искази (респективно) који се пишу на језику акција који се бави детаљима (енгл. *detail-level action language*). То може бити било који традиционални објектно оријентисани језик који служи као имплементациони језик. Типично, акције представљају позиве интерних операција дефинисаних у капсули. Њихова семантика је идентична семантици класичних операција у језицима објектне оријентације, осим што су оне увек приватне или заштићене (енгл.

*private, protected*, респективно) и нису доступне из спољашњости капсуле. Њима се може приступити само из сопственог понашања капсуле. Ово осигурава потпуну и стриктну енкапсулацију понашања капсула.

Имплементација ових операција се такође пише у акционом језику детаља. Њихов код може да садржи стандардне исказе контроле тока (услове, петље, позиве операција) и акције. Акције укључују приступ следећим сервисима доступним сопственом понашању капсуле:

1. Одржавање интерне структуре капсуле креирањем и брисањем њених унутрашњих делова и/или жица, а све у складу са моделом интерне структуре. Ово је доступно кроз посебан програмски интерфејс.
2. Слање порука на излазне пинове. Као саставни део капсуле, сваки пин је доступан као објекат у опсегу методе, а порука се шаље позивом на конкретне операције на том објекту. Слање поруке може бити асинхроно (неблокирајуће) или синхроно (блокирајуће). На пример, синхроно слање поруке која носи податке *val* преко пина *p* пише се
   ```
   p.send(val);
   ```
   Семантика слања порука преузета је из језика *UML2*. Имплементацији је остављена могућност да подржи један или оба начина слања порука.
3. Читање или писање интерних атрибута капсуле. Свака капсула може имати интерне атрибуте чије вредности доприносе укупном стању капсуле. Атрибути су преко својих имена директно доступни у телима метода.
4. Приступ објектном простору на строго контролисан начин као што ће бити показано у следећем одељку.

Пинови интерфејса капсула увек прослеђују поруке или у сопствено понашање капсуле, или до пинова капсула које формирају интерну структуру те капсуле. Сопствено понашање капсуле, према томе, ради као крајњи генератор, односно извор али и као крајњи понор порука.

### Приступ подацима

Оријентација пословних апликација ка вебу постала је свеприсутна у последњој деценији. Таква оријентација је умногоме унапредила доступност апликација, али је и утицала на то да основна карактеристика традиционалних технологија за развој корисничких интерфејса оријентисаних ка формуларима – чврста спрега са позадинским (релационим) моделом података – буде напуштена. Тако су кориснички интерфејси и слој пословне логике постали концептуално удаљени, а управо та удаљеност узрок је значајне количине случајне комплексности у развоју корисничких интерфејса. Треба споменути и чињеницу да програмирање корисничких интерфејса подразумева и значајну природну комплексност, што само по себи представља велики изазов. Кориснички интерфејс апликације свакако мора приступити објектном простору, односно подацима, како би приказао те податке и промене на њима. Међутим, начин на који се приступа подацима је врло осетљива тема због тога што мора да помири два донекле супротстављена захтева.

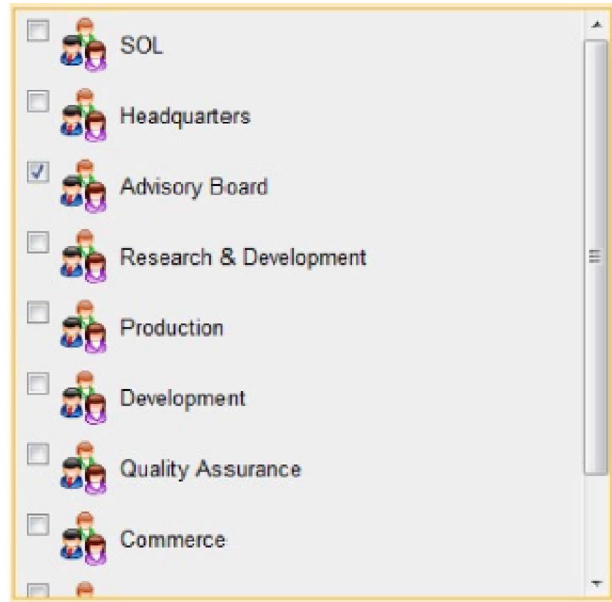
С једне стране, за ефикасан развој корисничких интерфејса од изузетне је важности да елементи корисничког интерфејса буду добро спрегнути са објектним простором апликације. Мора да постоји поклапање нивоа апстракције слоја података, односно објектног простора и корисничког интерфејса. Другим речима, елементи корисничког интерфејса морају да рефлектују парадигму која се користи у објектном простору. У супротном програмери морају да пишу много назграпног и непотребног кода како би

надоместили концептуалне разлике и спрегнули елементе корисничког интерфејса са подацима.

Са друге стране, ако је објектни простор у потпуности отворен и доступан слоју корисничког интерфејса, програмери лако упадају у замку мешања пословне логике у слој корисничког интерфејса. Практично све традиционалне технологије за развој корисничких интерфејса омогућавају неограничен приступ простору података. Управо због тога је поштовање правила развајања одговорности (енгл. *separation of concerns*) и примена пројектних образаца који одвајају ова два слоја апликације у потпуности препуштено дисциплини и овбучености програмера. Међутим, програмери ће увек правити грешке и нарушавати дисциплину, па и мешати код ова два аспекта развоја апликација. Врло чест пример који се среће у пракси јесте писање кода пословне логике у методама за обраду догађаја. Овај симптом је у литератури познат под називом ”анти-образац магичног дугмета” (енгл. *magic pushbutton anti-pattern*). Чим методе за обраду догађаја (акције прелаза у случају технике представљене у овој дисертацији) дозволе 1) исказе контроле тока (петље и услове) и 2) потпун и отворен приступ подацима (читање и снимање вредности и позивање операција), отвара се прилика да се помешају процедуре које логички припадају слоју пословне логике у методе за обраду догађаја слоја корисничког интерфејса. У посебном поглављу ове дисертације биће дат извештај о учестаности ове појаве у реалним пројектима из праксе.

У техници представљеној у овој дисертацији, први захтев за добром спрегом са подацима остварује се уз помоћ богате библиотеке капсула за приступ подацима. Ове капсуле с једне стране изгледају као стандардни елементи корисничког интерфејса (листе, табеле, стабла, поља за текст, за чекирање, за избор опције, слике итд.) а такође садрже и пинове и нуде стандардну семантику капсула другим капсулама из окружења. Са друге стране, ове капсуле подижу ниво апстракције тако да одговара нивоу апстракције објектног простора. Слика 9 илуструје једну такву капсулу за приступ подацима.

Капула на слици 9 је пример капсуле за измену слота објекта који се доставља на улазни пин *elem*. У овом примеру, објекат припада класи *Employee*. У наставку ће тај објекат бити називан објектом-домаћином. Слот је инстанца краја асоцијације из модела домена на језику *UML*. Овај крај асоцијације је дат као параметар конструкције капсуле са слике и чува се у једном атрибуту те капсуле. У овом случају у питању је асоцијациони крај *dept* са слике 6а који повезује запосленог са одељењем у ком ради. Сама капсула приказује колекцију објеката одређене класе који су кандидати за креирање веза (линкова) са објектом домаћином преко асоцијационог краја конфигурисаног приликом конструкције капсуле. Сама колекција објеката кандидата прослеђује се капсули преко другог улазног пина *collection* и та колекција може бити приказана као листа или стабло. Сваки пут када се нови објекат домаћин појави на улазном пину *elem*, капсула аутоматски дохвата везу (или у општем случају, везе) објекта домаћина на слоту *dept* из објектног простора и освежава кућице за чекирање (енгл. *checkbox*) тако да рефлектују прочитану везу (односно везе). Кад год корисник апликације промени стање неке кућице, односно чекира један од објеката из колекције објеката кандидата, капсула аутоматски ажурира објектни простор креирањем или брисањем одговарајуће везе. Као резултат, програмер не мора да уради ништа посебно да би обезбедио ово типично понашање. Све се ради декларативно, конфигурисањем капсуле и повезивањем њених пинова са другим капсулама из окружења. Библиотека технике предложене у овој дисертацији обилује софистицираним капсулама које прате описану парадигму. Неке од њих биће описане у наредним поглављима, а комплетан каталог као и примери апликација могу се наћи на www.soloist4uml.com.

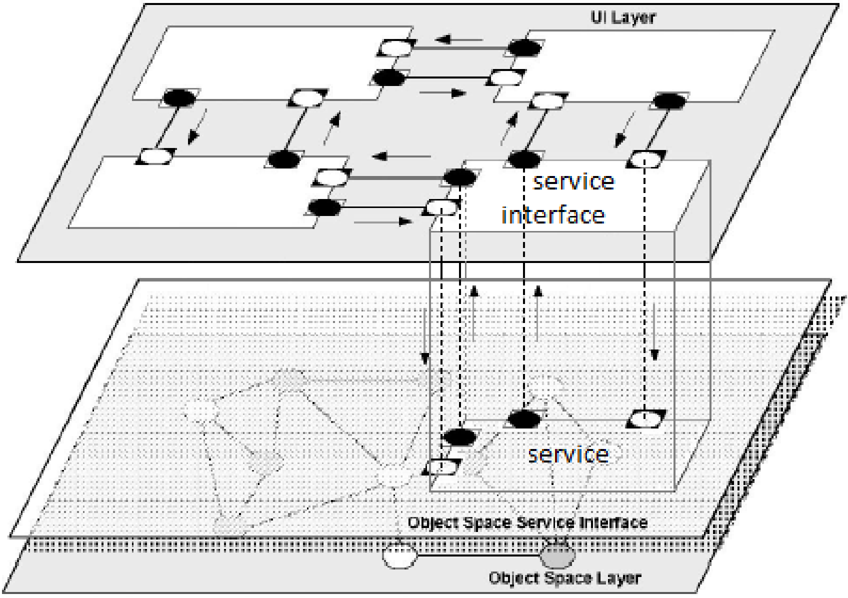



Слика 9. Пример капсуле за промену одељења запосленог (измена везаног објекта).

Други захтев који има за циљ јасно раздвајање одговорности реализује се одвајањем слојева објектног простора и корисничког интерфејса у два логичка слоја као на слици 10. Слој корисничког интерфејса је горњи слој који се састоји од капсула и сервиса. Сервиси су посебна врста капсула. Они се разликују од обичних капсула по томе што немају познату (дефинисану) интерну структуру, већ само интерфејс. Њихов интерфејс се, као код капсула, састоји од пинова. Сервис се може схватити као екстерна класа чије инстанце повезују слој корисничког интерфејса са слојем објектног простора. У слоју корисничког интерфејса, капсуле и сервиси „хоризонтално“ размењују поруке (горњи слој приказан на слици 10). Сервис прима вредности својих параметара преко својих улазних пинова, док се резултати извршавања сервиса враћају слоју корисничког интерфејса преко излазних пинова сервиса. У слоју корисничког интерфејса, сервиси једино имају стање које чува вредности параметара добијене преко улазних пинова сервиса.

Нижи слој је објектни простор домена који обезбеђује одговарајућу структуру података, у облику графа повезаних објеката, као и пословну логику, типично у облику операција на објектима (обратити пажњу на нижи доњи слој на слици 10). Слој корисничког интерфејса и слој објектног простоја комуницирају тако што размењују поруке „вертикално“. Ове поруке прихватају и обрађују сервиси, који их прослеђују окружујућим капсулама. Поруке имају исту семантику и капсуле их третирају на исти начин као поруке које капсуле размењују између себе у слоју корисничког интерфејса. Овакво логичко одвајање омогућава да се ови слојеви могу лако одвојити и физички, нпр. у клијент-сервер архитектури.Помоћу сервиса који се налазе у слоју корисничког интерфејса, слој објектног простора нуди посебан интерфејс својих услуга;имплементација тих услуга, очекивано, налази се у слоју објектог простора (обратити пажњу на виши доњи слој на слици 10).

У техници предложеној у овој дисертацији налази се већи број предефинисаних сервиса од којих ће већина бити објашњена у наредним поглављима. Укратко речено, постоје сервиси од којих капсуле могу да захтевају читање вредности слота објекта (нпр. дохватање колекције објеката преко инстанци асоцијација или вредности атрибута), упис вредности атрибута, креирање или брисање везе између објеката, итд. Одговор на овакве захтеве стиже преко интерфејса сервиса који су декларисани слоју корисничког интерфејса.




Слика 10. Слојевита архитектура апликације.

Још један важан механизам уграђен у ову парадигму јесте механизам нотификација (обавештавања, енгл. *notifications mechanism*) слоја корисничког интерфејса о променама у слоју објектног простора. Наиме, за сваку промену у објектном простору, нпр. креирање или брисање објекта или промена вредности слота, слој објектног простора шаље нотификациону поруку слоју корисничког интерфејса преко посебног сервиса. Истим механизмом комуникације између слојева, нотификациона порука се прослеђује свим капсулама које су преко пинова повезане са тим сервисом[25]. Према томе, капсуле могу да реагују на нотификације одређене врсте тако што ажурирају своје репрезентације на екрану корисника у складу са нотификацијама, односно са променама објектног простора. У суштини, ово је врло једноставан сурогат пројектног обрасца „посматрач" (енгл. *observer*) [10]. Капсула се пријављује за нотификације одређене врсте повезивањем са одговарајућим сервисом. Капсула треба да обезбеди реакцију на нотификационе поруке у свом сопственом понашању. Уграђене капсуле су већ повезане са овим сервисом и већ поседују овакве реакције у себи. У пракси се показало да је овај механизам нотификације од непроцењиве вредности јер значајно смањује комплексност приликом изградње корисничких интерфејса. Програмер се ослобађа бриге о освежавању појединих делова корисничког интерфејса у зависности од промена у објектном простору. Уместо тога, програмер може да се фокусира на концептуални модел домена и пословну логику, док се кориснички интерфејс аутоматски ажурира јер се уграђене капсуле обавештавају и реагују на промене у објектном простору. На пример, сваки пут када се обрише објекат из домена у било којој процедури пословне логике, све капсуле које приказују тај објекат или његове слотове биће аутоматску ажуриране (уклањањем објекта или вредности његових слотова са екрана).

[25] Из практичних разлога, у тренутној имплементацији, нотификације се шаљу корисничком интерфејсу који одговара корисничкој сесији само за оне измене које су проузроковане трансакцијама извршеним из те исте корисничке сесије. Међутим, концепт нотификација је генералан и могућа је и реализација у којој се може конфигурисати којим корисничким сесијама се прослеђују нотификације.

Са циљем да се спречи или бар драстично отежа и тиме обесхрабри уградња кода пословне логике у слој корисничког интерфејса, примењен је следећи приступ. Понашању капсула доступне су само типизиране референце на доменске објекте. Оне се могу чувати у атрибутима капсула, прослеђивати преко пинова и користити као параметри конструктора. Оне подлежу класичним правилима типизирања као што су конверзија типова, субституција и подтипови (енгл. *conversion, substitution, subtyping*, респективно). Међутим, особине референцираних објеката, односно њихови атрибути, крајеви асоцијација и операције су недоступни у слоју корисничког интерфејса. (Инстанце примитивних типова података су доступне због тога што се ови типови података сматрају непромењљивим (енгл. *immutable*) [26, 27] па их самим тим ни слој корисничког интерфејса не може променити.) Да би се приступило објекту из домена или променила било која његова особина, понашање капсуле мора послати поруку ка сервису и по потреби сачекати одговор. У суштини, капсуле и објекти из домена постоје у различитим адресним просторима, због чега је програмеру значајно отежано да меша код пословне логике и код корисничког интерфејса, односно понашања капсула. Врло је тешко приступити било којој операцији или својству објекта из домена (прочитати један или више слотова или позвати једну или више операција) у оквиру секвенцијалног кода метода за обраду догађаја, односно метода капсула. Уместо тога, мора се послати (типично асинхрони) захтев сервису из објектног простора и уградити свака компликованија процедура пословне логике у слој објектног простора, односно у операције доменских класа. Према томе, слој корисничког интерфејса је фокусиран на интеракцију између капсула и сервиса, обраду догађаја, размену сигнала и параметара, док је одговорност доменске логике смештена у доменским класама.

Постоје случајеви када се структура неког фрагмента корисничког интерфејса мора динамички саставити према тренутном стању дела објектног простора или према резултату неке процедуре пословне логике. На пример, често се на зна тачан број елемената корисничког интерфејса које треба приказати, али се зна да сваки од њих одговара једном објекту из колекције објеката која је резултат одређене процедуре пословне логике. За овакве потребе, у техници предложеној у овој дисертацији постоји концепт капсуле са динамичком интерном структуром. Интерфејс и понашање оваквих капсула дефинише се на исти начин као код осталих капсула, али дефиниција интерне структуре у конструктору може бити празна или дефинисати само иницијалну или парцијалну интерну структуру која се касније, у току извршавања програма, може променити. У том случају се мора дефинисати и имплементирати сервис за сваку овакву капсулу: сервис прихвата захтев и испоручује одговор. Понашање динамичке капсуле користи сервис да пошаље захтев за дефиницијом, односно креационом спецификацијом своје интерне структуре. Ова дефиниција се конструише у имплементацији сервиса у слоју објектног простора. За конструкцију се користи апликативни програмски интерфејс уз помоћ ког се конструише интерна структура у складу са стањем објектног простора или резултатом процедуре пословне логике. Када се дефиниција интерне структуре пошаље назад капсули која је захтевала, конструише се интерна структура капсуле на исти начин као приликом иницијализације. Процедура дохватања интерне структуре може се понављати неограничен број пута у току животног циклуса капсуле, сваки пут мењајући стару интерну структуру новом. На овај начин се интерна структура капсуле конструише динамички, у време извршавања програма, у складу са стањем објектног простора и пословном логиком, при чему је раздвајање одговорности у потпуности очувано. Пословна логика остаје у слоју објектног простора и не меша се са слојем корисничког интерфејса; она само конструише добро дефинисане и енкапсулиране фрагменте модела корисничког интерфејса, али се не меша у њихову динамику и понашање.

# Део 4: Имплементација

# *Језик за моделовање корисничких интерфејса*

Након представљања идејног решења и основних принципа на којима почива техника представљена у овој дисертацији, у овом делу, фокус је стављен на њен практични аспект. Конкретно, у овом поглављу биће представљен нов, текстуални, доменски језик (енгл. *domain-specific language, DSL*), радно назван *CAPWISE* (скраћено, од енглеске фразе *CApsules, Pins, WIres and SErvices*). У овом конкретном случају циљни домен јесте конструкција графичких корисничких интерфејса. Један од важних принципа у дизајну језика *DSL* подразумева увођење само неопходних концепата и избегавање непотребне општости [17]. У случају језика *CAPWISE* најважнији концепти су капсула, пин, жица, сервис, као и неки концепти преузети из језика опште намене, као што су конструктори, наслеђивање, права приступа итд. У наставку текста биће објашњена синтакса и семантика језика неформално, кроз један пример употребе, а затим ће и формално бити представљени детаљи синтаксе језика.

## Пример употребе

У овом примеру употребе биће демонстрирана употреба језика у имплементацији дела једне апликације. Циљ је да се покаже неколико важних сценарија употребе језика и његових уграђених елемената и то:

- употреба примитивних, односно уграђених, капсула и повезивање капсула жицама,
- прављење нових сложених капсула од постојећих и
- употреба сервиса.

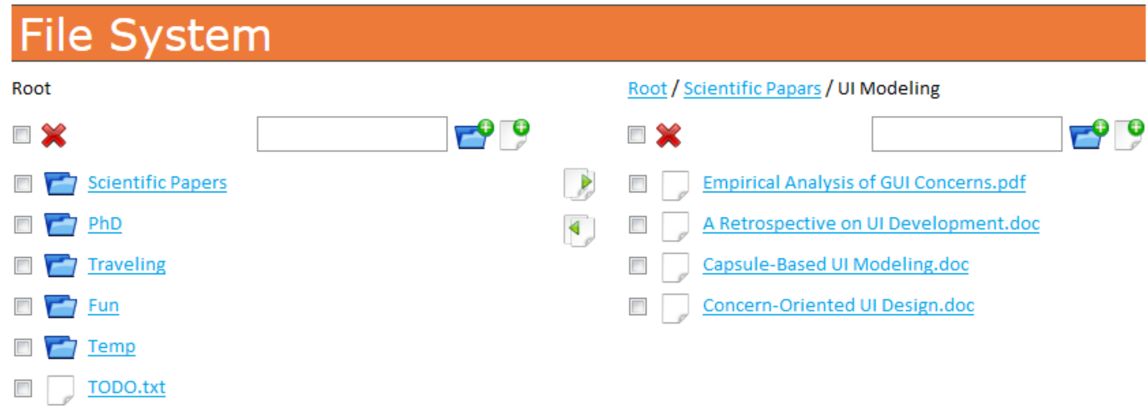
Пре него што се изложе детаљи језика и његова употреба, укратко ће бити објашњен кориснички интерфејс циљне апликације која се конструише у овом примеру, концептуални модел те апликације на језику *UML* и сервиси за приступ подацима неопходни за рад апликације.

Предмет имплементације у овом примеру је апликација чији је кориснички интерфејс приказан је на слици 11. Реч је о апликацији за преглед и управљање системом фајлова, односно за преглед хијерархијске структуре фолдера, као и за копирање, прављење и брисање фолдера и фајлова. Кориснички интерфејс ове апликације подсећа на поједностављен и измењен кориснички интерфејс познатих програма за управљање системом фајлова, као што су програми *Norton Commander, Free Commander* и други

слични. На слици 11 приказан је карактеристичан двоструки поглед на систем фајлова који кориснику омогућава комфорнији преглед система фајлова и бржи рад са фајловима и фолдерима.

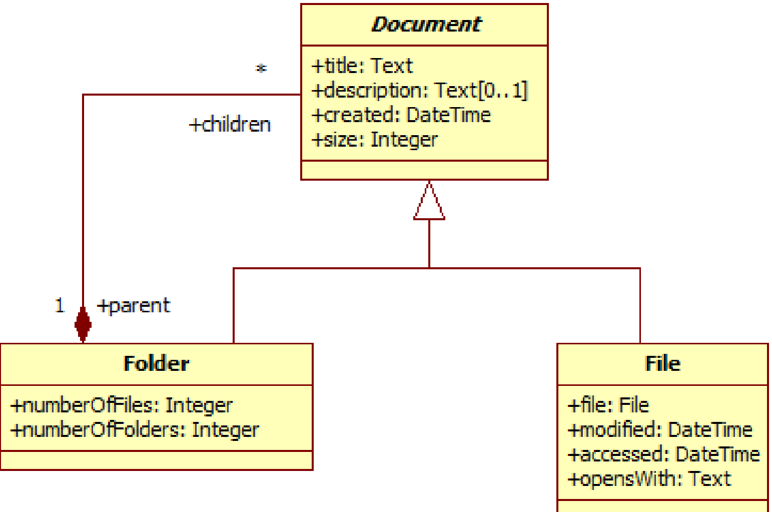
Кориснички интерфејс са слике 11 састоји се из више делова:

- заглавље апликације са лабелом *File System*,
- секције са линковима за хијерархијски преглед текућих фолдера,
- кућице за чекирање (енгл. *checkbox*) за обележавање фајлова и фолдера једном потезу,
- дугмићи за брисање обележених фајлова и фолдера,
- дугмићи за прављење фајлова и фолдера са задатим именом,
- секције за преглед садржаја текућих фолдера и
- дугмићи за копирање фајлова и фолдера из текућег фолдера првог погледа у текући фолдер другог погледа и обрнуто.




Слика 11. Кориснички интерфејс апликације *File System*.

Дијаграм који приказује модел података апликације приказан је на слици 12. Модел чине три класе. Основна класа је апстрактна класа *Document*. Сваки објекат ове класе (односно, сваки документ) има име, време креирања и величину, док је опис документа опциона особина (видети атрибуте *title, created, size* и *description*). Документ је заправо апстрактна генерализација појмова фајл и фолдер. Ови појмови су представљени у моделу класама *File* (фајл) и *Folder* (фолдер). Сваки фолдер може да садржи више докумената, односно фајлова и фолдера (асоцијациони крај *children*) и чува број фајлова и фолдера које садржи (атрибути *numberOfFiles* и *numberOfFolders*). Са друге стране, објекти класе *File* у свом атрибуту *file* чувају бинарни садржај фајла, док у атрибутима *modified*, *accessed* и *opensWith* чувају информације о томе време када је фајл последњи пут модификован, време када је последњи пут отворен, као и назив програма који може да отвори дати фајл.




Слика 12. Модел података апликације *FileSystem*.

Сервиси који обезбеђују везу корисничког интерфејса ове апликације са моделом података су следећи[26]:

- сервис за креирање фолдера у изабраном родитељском фолдеру (*CreateFolder*),
- сервис за креирање фајла у изабраном родитељском фолдеру (*CreateFile*),
- сервис за брисање изабраних фајлова и фолдера (*DeleteDocs*),
- сервис за копирање изабраних фајлова и фолдера из изворишног у одредишни фолдер (*CopyDocs*),
- сервис за читање фајлова и фолдера за изабрани родитељски фолдер (*ReadDocs*),
- сервис за читање свих фолдера из хијерархије изабраног фолдера почевши од изабраног фолдера све до кореног фолдера хијерархије укључујући и њега (*ReadHierarchy*) и
- сервис за нотификације о променама у простору доменских објеката (*Notifier*).

Изворни код на језику *CAPWISE* са дефиницијама побројаних сервиса приказан је у листингу 1. Зеленом бојом означене су кључне речи језика, црном бојом идентификатори, црвеном терминални симболи, а плавом коментари. Ова конвенција обележавања изворног кода биће коришћена у свим листинзима у овом одељку.

```
service CreateFolder {
	public input pin folderName : Text [1];
	public input pin parentFolder : Folder [1];
	public input pin execute : void [*];
}

service CreateFile {
```

[26] Уместо набројаних сервиса могу се искористити и генерички сервиси из библиотеке језика. На пример, у библиотеци постоје следећи сервиси: сервис за прављење објекта једне класе и везивање са објектом друге класе, сервис за брисање објекта, сервис за копирање објекта, сервис за читање објеката везаних за други објекат итд.

```
    public input pin fileName : Text [1];
    public input pin parentFolder : Folder [1];
    public input pin execute : void [*];
}

service DeleteDocs {
    public input pin docs : Document [*];
    public input pin execute : void [*];
}

service CopyDocs {
    public input pin docs : Document [*];
    public input pin srcFolder : Folder [1];
    public input pin dstFolder : Folder [1];
    public input pin execute : void [*];
}

service ReadDocs {
    public input pin folder : Folder [1];
    public input pin execute : void [*];
    public output pin docs : Document [*];
}

service ReadHierarchy {
    public input pin folder : Folder [1];
    public input pin execute : void [*];
    public output pin folders : Folder [*] {ordered, unique};
}

service Notifier {
    public output pin notifications : Notification [*];
}
```

Листинг 1. Изворни код за дефиницију сервиса.

Важно је напоменути да листинг 1 приказује само интерфејс сервиса, док се њихова имплементација спроводи на другом програмском језику (на пример, на језику *Java*), а не на језику *CAPWISE*.

У имплементацији ове апликације, поред описаних сервиса биће коришћене и библиотечке капсуле које су приказане у листингу 2. Читалац ће приметити кључну реч *native* у декларацији ових капсула која сугерише на то да се ради о примитивним капсулама чија је

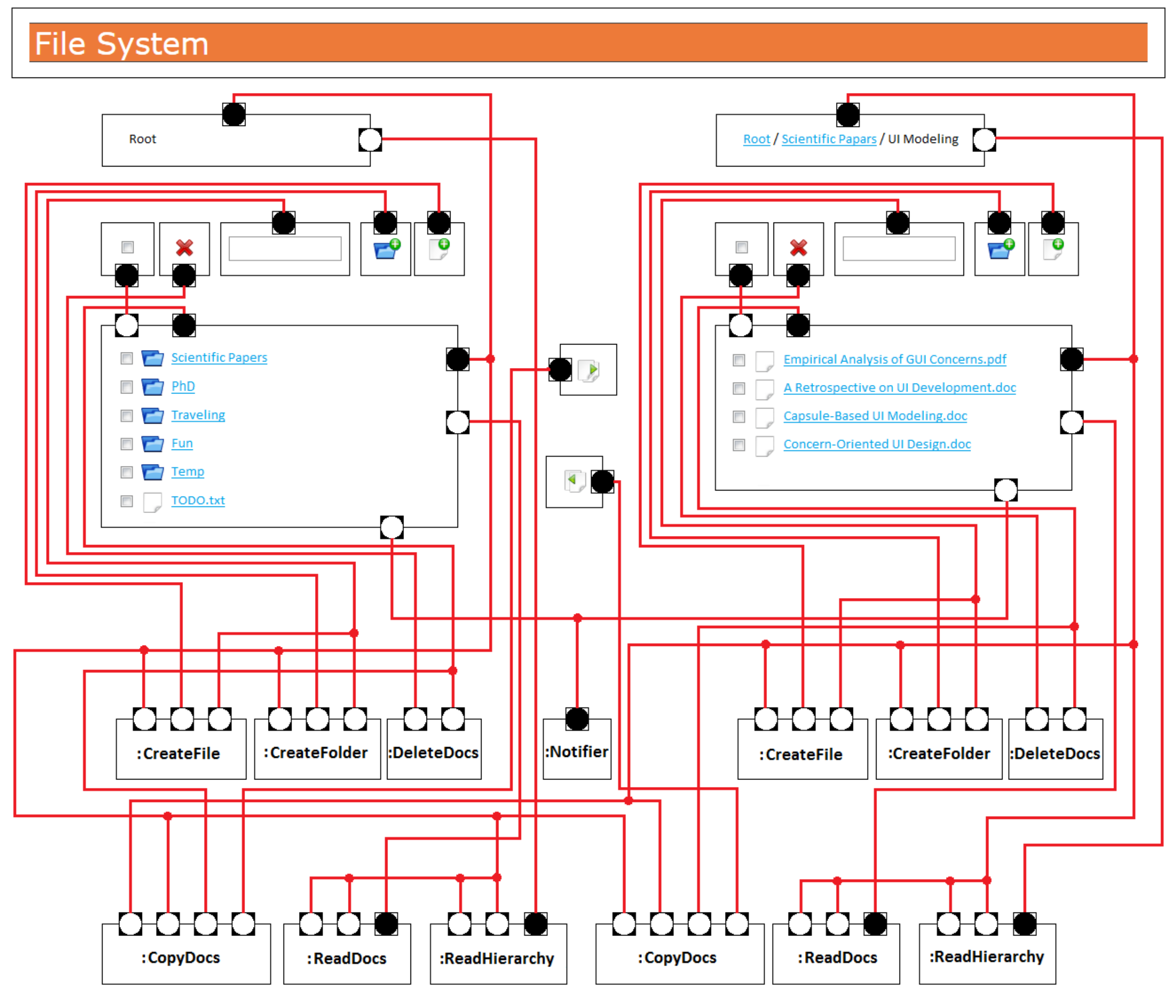
имплементација, слично као код сервиса, остављена другом програмском језику. Као што је већ речено, поред примитивних капсула постоје и сложене капсуле које се састоје од других (примитивних и сложених) капсула и сервиса. Свака капсула (и примитивна и сложена), поред специфичних пинова, инхерентно поседује и пин *parent* (видети листинг 2) који служи за повезивање дате капсуле са окружујућом (родитељском), сложеном капсулом. Аналогно томе, свака сложена капсула инхерентно поседује пин *children* који служи за повезивање дате капсуле са капсулама које она окружује. У листингу 2 декларације капсула садрже само оне елементе (пинове, конструкторе итд.) који ће бити коришћени у овом примеру, док су други пинови изостављени због прегледности.

```
public native capsule Label {
	public Label (Text caption, Text style);
	public input pin parent : void [*] @ layout;
}

public native capsule Button {
	public Button (Text caption, Text tooltip, Picture icon);
	public output pin click : void [*];
	public input pin parent : void [*] @ layout;
}

public native capsule CheckBox {
	public CheckBox ();
	public output pin value : Boolean [1];
	public input pin parent : void [*] @ layout;
}

public native capsule TextBox {
	public TextBox ();
	public output pin value : Text [1];
	public input pin parent : void [*] @ layout;
}

public native capsule List {
	public List (Boolean showCheckBoxes, Boolean isMultiselect1, Boolean
isMultiselect2);
	public input pin notifications : Notification [*] {unique, ordered};
	public input pin objects : void [*];
	public input pin selectAll : Boolean [1];
	public output pin selection1 : void [*];
	public output pin selection2 : void [*];
	public input pin parent : void [*] @ layout;
```

```
}

public native capsule BreadCrumbs {
	public BreadCrumbs (Text separator);
	public input pin objects : void [*];
	public output pin selection : void [1];
	public input pin parent : void [*] @ layout;
}
```

Листинг 2. Изворни код уграђених капсула.

Кориснички интерфејс дате апликације могуће је направити употребом сервиса из листинга 1 и капсула из листинга 2. На слици 13 графички је приказан један такав начин имплементације. Капсуле и сервиси су означени црним правоугаоницима са пиновима (обележавање пинова је у складу са нотацијом из дела 3 ове дисертације), а жице црвеним линијама које спајају пинове. Сервиси на слици имају текстуални натпис који означава тип сервиса, како би се лакше разликовали једни од других. Капсуле имају изглед који иначе имају на корисничком интерфејсу па се такође могу лако препознати и међусобно разликовати. Из разлога побољшања читљивости, на слици 13 су изостављени пинови *parent* и *children* и одговарајуће жице. Њихова улога није кључна и односни се искључиво на распоред капсула на екрану.




Слика 13. Једна могућа варијанта имплементације корисничког интерфејса дате апликације.

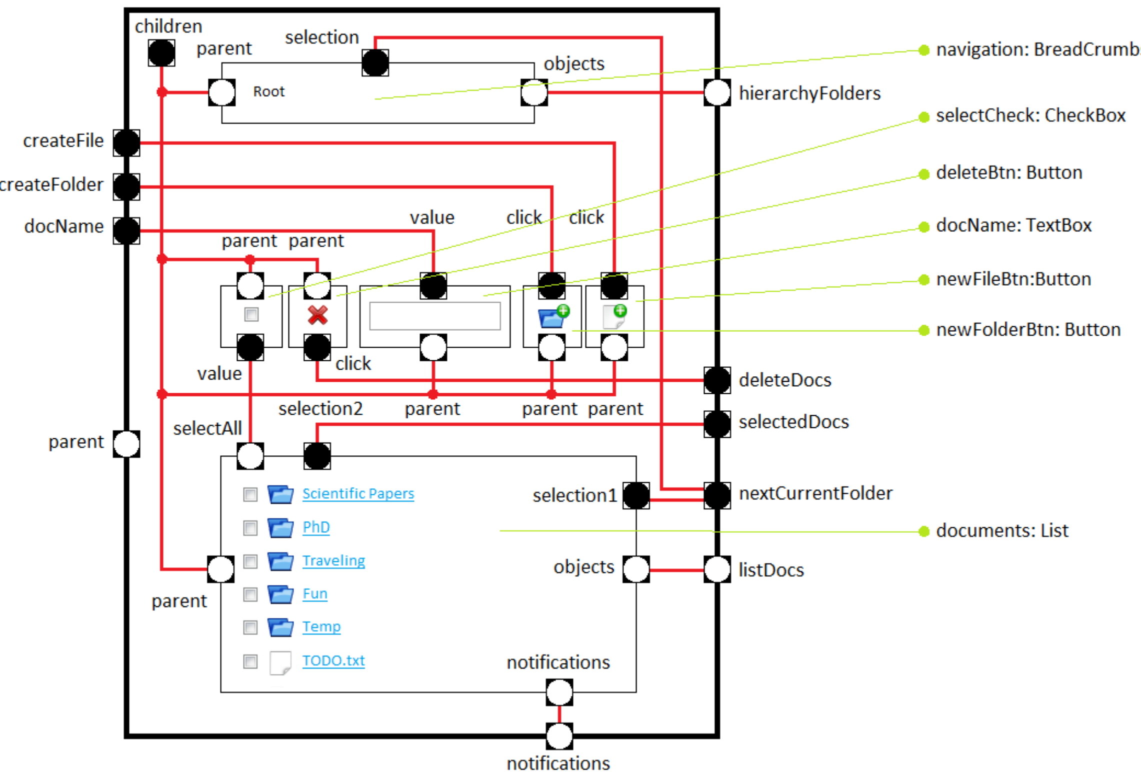
Иако је решење приказано на слици 13 сасвим легитимно и исправно, оно је прилично непрегледно, односно непотребно сложено и због тога изворни код тог решења неће бити приложен. Разлог лежи у томе што се кориснички интерфејс дате апликације може имплементирати једноставније и то управо захваљујући принципима објашњеним у овој дисертацији и правилном употребом језика *CAPWISE* који те принципе инхерентно подржава и фаворизује.

Једноставност решења које ће бити приказано у наставку лежи у примени принципа апстракције, енкапсулације и поновне употребе. Генерално говорећи, одабир дела корисничког интерфејса који треба издвојити и апстраховати и енкапсулирати (односно промовисати у капсулу) јесте важна пројектна одлука у процесу програмирања корисничког интерфејса. Ова одлука зависи од више фактора. Најочигледнији разлог за увођење нове капсуле је њен потенцијал за поновну употребу. Другим речима, ако програмер корисничког интерфејса процени да ће се један део корисничког интерфејса користити више пута на сличан или идентичан начин, разумно је заокружити тај део корисничког интерфејса у нову капсулу. Поред тога, увођење нове капсуле оправдано је и у случајевима када се процени да део корисничког интерфејса није прецизно дефинисан и да је велика вероватноћа да ће претрпети промене у будућности. Тиме се окружење штити од промена у том делу корисничког интерфејса на начин што му приступа искључиво преко

пинова, односно преко прецизно дефинисаног интерфејса, па евентуалне промене у унутрашњости тог дела корисничког интерфејса не могу имати утицај на његово окружење. Увођење нове капсуле има смисла и када се жели изоловати неки специфичан или сложен део корисничког интерфејса, како се сложеност не би „разлила“ по изворном коду окружења. Ово свакако нису сви разлози за увођење нових капсула. Програмер треба да одлучи сам, на основу свог инжењерског знања и искуства, који делови корисничког интерфејса представљају добре кандидате за енкапсулацију.

У конкретном случају корисничког интерфејса приказаног на слици 11, изузев заглавља и дугмића за копирање фајлова и фолдера сви остали делови појављују се по два пута на екрану. То их чини идеалним кандидатима за поновну употребу (енгл. *reuse*). Поред тога што се понављају секције корисничког интерфејса, понављају се и сервиси који се користе у тим секцијама. Управо из тог разлога, у примеру изворног кода који следи, ове секције и сервиси биће имплементирани као посебне капсуле који се могу више пута употребити, односно уградити у кориснички интерфејс.

На слици 14 приказан је дијаграм капсуле која садржи графичке елементе (такође капсуле) које се налазе на једној страни екрана. Другим речима, део корисничког интерфејса је промовисан у сложену капсулу која се састоји од других капсула. Промоција дела корисничког интерфејса у капсулу поред избора садржавајућих капсула подразумева и спецификацију интерфејса, односно пинова сложене капсуле. У случају капсуле са слике 14 у интерфејсу се налази шест излазних пинова (*createFile, createFolder, docName, deleteDocs, selectedDocs,* и *nextCurrentFolder*), три улазна пина (*hierarchyFolders, listDocs,* и *notifications*) и један инхерентни улазни пин који има свака капсула: *parent*. На овој слици приказани су поред осталог и пинови *parent* примитивних капсула, пинови *children* и *parent* сложене капсуле, као и одговарајуће жице.




Слика 14. Капсула која енкапсулира графичке елементе на једној страни екрана.

Изворни код на језику *CAPWISE* капсуле са слике 14 приказан је у листингу 3. Изворни код састоји се из два дела: конструктора и интерфејса. У конструктору се инстанцирају, иницијализују и међусобно жицама повезују пинови капсула које садржи сложена капсула и пинови сложене капсуле. У другом делу, спецификација интерфејса обухвата дефиницију пинова сложене капсуле.

На самом крају конструктора у листингу 3, у последњој секцији, пин *children* сложене капсуле је жицама повезан са пиновима *parent* осталих капсула. Ове жице дефинишу одговорност сложене капсуле за распоред унутрашњих капсула на екрану. У изворном коду у листингу 3 изостављена су подешавања унутрашњег распореда сложене капсуле, као и фина подешавања позиција унутрашњих капсула. Другим речима, изостављено је да је распоред унутрашњих капсула (на пример) табеларни, као и координате и простирање у редовима и колонама појединачних капсула у таквом распореду.

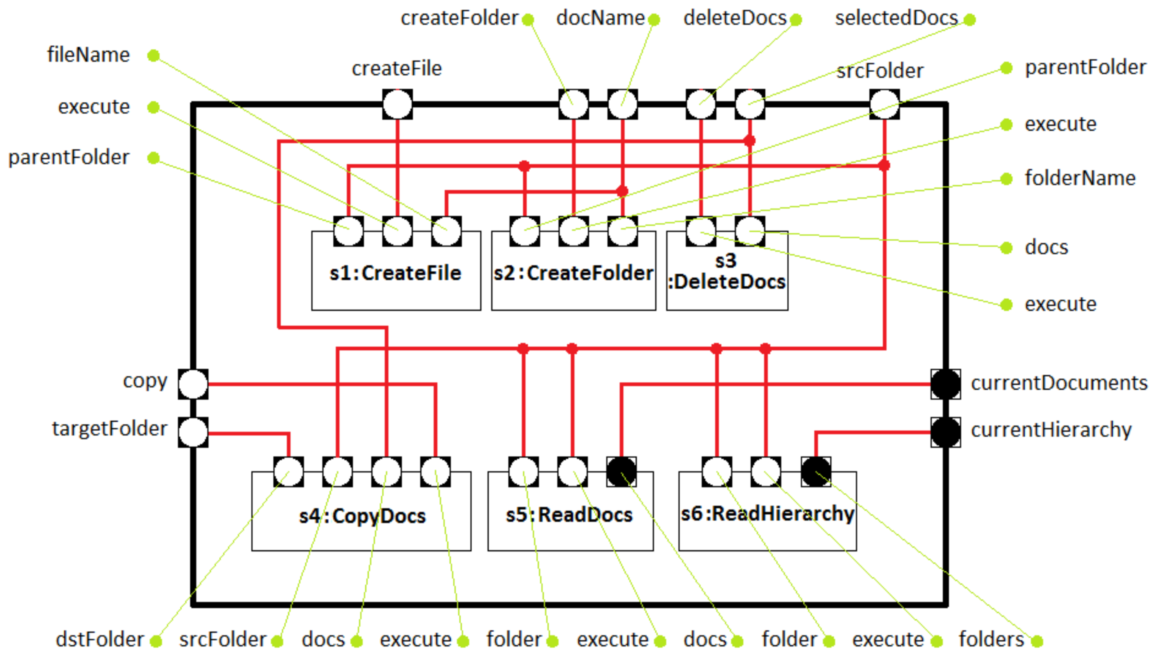
Иако је интуитивно логично да капсула која енкапсулира друге капсуле буде одговорна за њихов распоред на екрану, могућа су и друга решења. На пример, сложена капсула би могла да има улазни пин у свом интерфејсу који би (уместо пина *children* сложене капсуле) жицама био спојен са пиновима *parent* осталих капсула. Тиме би се одговорност за распоред унутрашњих капсула могла делегирати некој другој сложеној капсули која се налази споља и која би преко пина у интерфејсу дате сложене капсуле могла да буде одговорна за распоред капсула у њеној унутрашњости. Ово је врло интересантна могућност која повећава флексибилност. У пракси се врло често дешава да део корисничког интерфејса треба распоредити на екрану далеко од места где по логици понашања, стилизације или неког другог критеријума припада. Овај механизам омогућава управо то да се распоред, као и други аспекти, може делегирати капсулама изван енкапсулираног фрагмента корисничког интерфејса.

```
public capsule Browser {
	public Browser() {
		BreadCrumbs navigation = new BreadCrumbs("/");
		Button newFileBtn = new Button("", "Create File", new
Picture("file.png"));
		Button newFolderBtn = new Button ("", "Create Folder", new
Picture("folder.png"));
		Button deleteBtn = new Button ("", "Delete Documents", new
Picture("delete.png"));
		TextBox docName = new TextBox();
		CheckBox selectCheck = new CheckBox();
		List documents = new List(true, false, true);


		wire (newFolderBtn.click, this.createFolder);
		wire (newFileBtn.click, this.createFile);
		wire (deleteBtn.click, this.deleteDocs);
		wire (docName.value, this.docName);
		wire (selectCheck.value, documents.selectAll);
		wire (documents.selection1, this.nextCurrentFolder);
		wire (navigation.selection, this.nextCurrentFolder);
```

```
        wire (documents.selection2, this.selectedDocs);
        wire (this.listDocs, documents.objects);
        wire (this.hierarchyFolders, navigation.objects);
        wire (this.notifications, documents.notifications);

        wire (this.children, navigation.parent);
        wire (this.children, selectCheck.parent);
        wire (this.children, deleteBtn.parent);
        wire (this.children, docName.parent);
        wire (this.children, newFileBtn.parent);
        wire (this.children, newFolderBtn.parent);
        wire (this.children, documents.parent);
    }
    public input pin hierarchyFolders : Folder [1..*] {unique, ordered};
    public input pin listDoc : Document [*] {unique};
    public input pin notifications : Notification [*] {unique, oredered};
    public output pin createFile : void [*];
    public output pin createFolder : void [*];
    public output pin docName : Text [1];
    public output pin deleteDocs : void [*];
    public output pin selectedDocs : Document [*] {unique};
    public output pin nextCurrentFolder : Folder [1];

    protected output pin children : void [*] @ layout;
    public input pin parent : void [*] @ layout;
}
```

Листинг 3. Изворни код капсуле која енкапсулира графичке елементе на једној страни екрана.

Група сервиса који се понављају у овом примеру такође се може енкапсулирати у посебну сложену капсулу. Конструкција ове капсуле врши се аналогно претходно приказаном начину конструкције капсуле *Browser*. На слици 15 приказан је дијаграм структуре капсуле састављене искључиво од сервиса. Изворни код тог ове капсуле приложен у листингу 4.



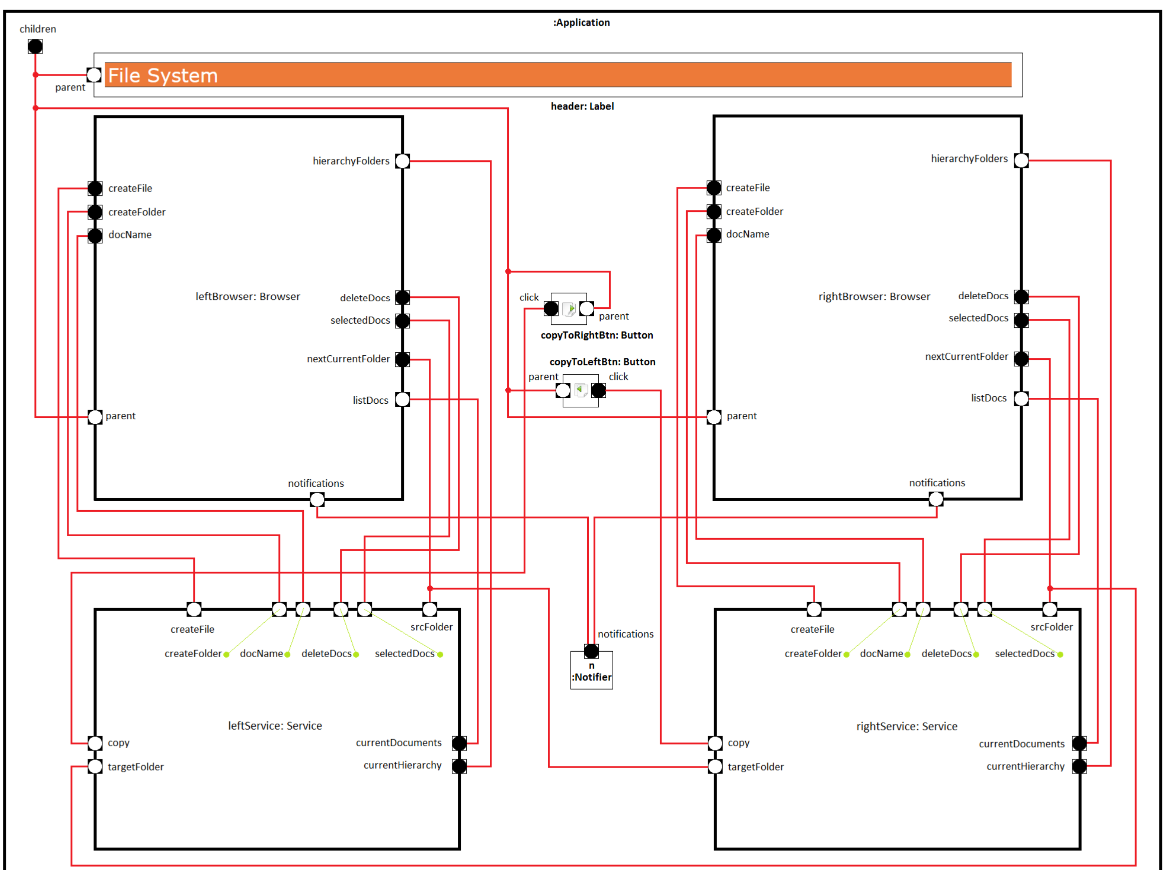

Слика 15. Графички приказ структуре капсуле која енкапсулира сервисе.

```
public capsule Service {
      public Service () {
            CreateFile s1 = new CreateFile();
            CreateFolder s2 = new CreateFolder();
            DeleteDocs s3 = new DeleteDocs();
            CopyDocs s4 = new CopyDocs();
            ReadDocs s5 = new ReadDocs();
            ReadHierarchy s6 = new ReadHierarchy();

            wire (this.srcFolder, s1.parentFolder);
            wire (this.createFile, s1.execute);
            wire (this.docName, s1.folderName);
            wire (this.srcFolder, s2.parentFolder);
            wire (this.createFolder, s2.execute);
            wire (this.docName, s2.fileName);
            wire (this.selectedDocs, s3.docs);
            wire (this.deleteDocs, s3.execute);
            wire (this.targetFolder, s4.dstFolder);
            wire (this.srcFolder, s4.srcFolder);
            wire (this.selectedDocs, s4.docs);
            wire (this.copy, s4.execute);
            wire (this.srcFolder, s5.folder, s5.execute);
```

```
            wire (this.srcFolder, s6.folder, s6.execute);
            wire (s5.docs, this.currentDocuments);
            wire (s6.folders, this.currentHierarchy);
      }
      public input pin createFile : void [*];
      public input pin createFile : void [*];
      public input pin docName : Text [1];
      public input pin deleteDocs : void [*];
      public input pin selectedDocs : Document [*] {unique};
      public input pin srcFolder : Folder [1];
      public input pin copy : void [*];
      public input pin targetFolder : Folder [1];
      public output pin currentDocuments : Document [*] {unique};
      public output pin currentHierarchy : Folder [1..*] {unique, ordered};
}
```

Листинг 4. Изворни код капсуле која енкапсулира сервисе.

Коначно, на слици 16 приказан је блок дијаграм комплетног корисничког интерфејса апликације *File System.* Као што се може приметити на слици, уз помоћ принципа хијерархијске декомпозиције и енкапсулације и правилном употребом језика *CAPWISE,* структура корисничког интерфејса апликације на највишем нивоу апстракције постаје практично тривијална. Она се састоји од четири сложене капсуле, две примитивне капсуле и једног сервиса чији су пинови међусобно повезани жицама. У листингу 5 дат је целокупни изворни код апликације. Приложени изворни код од свега тридесетак програмских линија, такође одсликава једноставност која се уочава и на блок дијаграму. Наравно, изворни код апликације чини и код приказан у претходним листинзима, међутим важно је напоменути да је сваки од нивоа апстракције на које је сведен кориснички интерфејс апликације врло једноставан и прегледан, па га је самим тим лакше направити и одржавати.




Слика 16. Дијаграм корисничког интерфејса апликације.

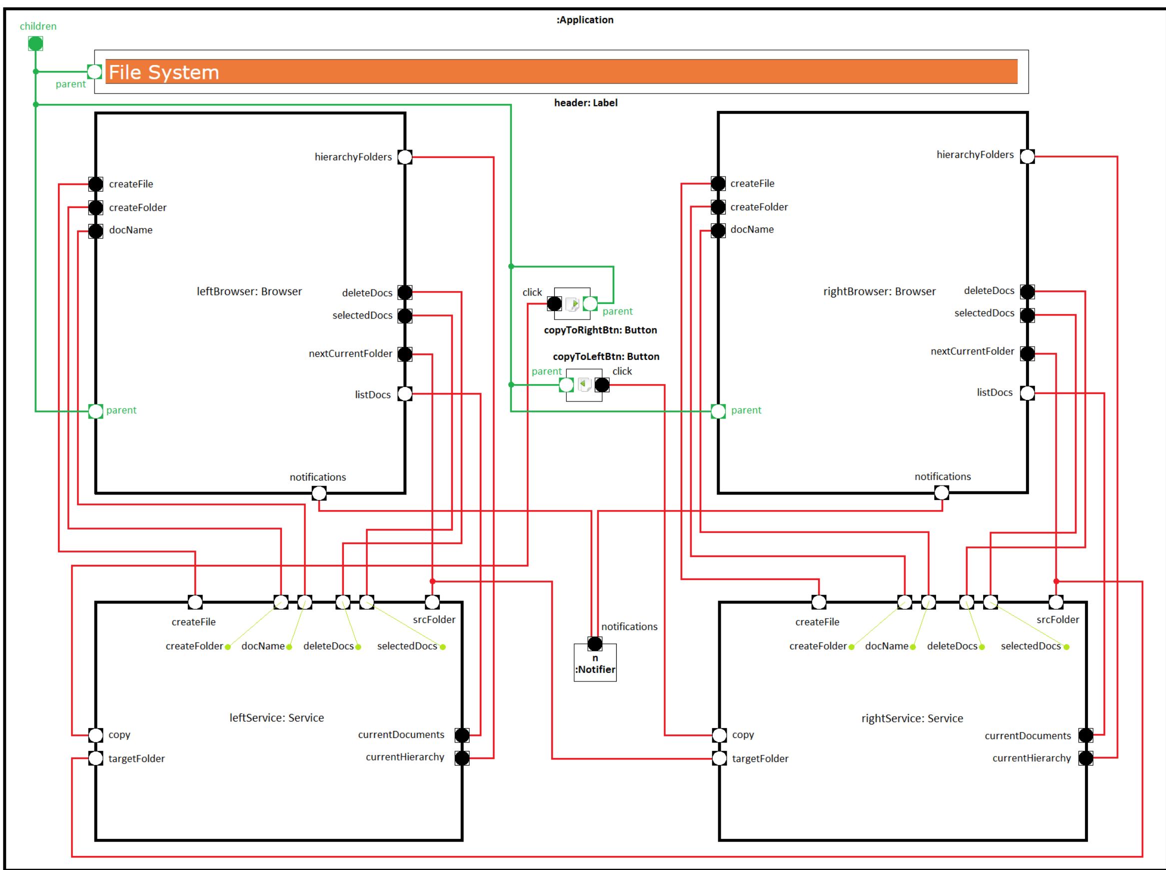
```
public capsule Application {
      public Application() {
            Label title = new Label("File System", "orange top-border");
            Browser leftBrowser = new Browser();
            Browser rightBrowser = new Browser();
            Service leftService = new Service();
            Service rightService = new Service();
            Button copyToRightBtn = new Button("", "Copy documents", new
Picture("copy-right.png"));
            Button copyToLeftBtn = new Button("", "Copy documents", new
Picture("copy-left.png"));
            Notifier n = new Notifier();

            wire(leftBrowser.createFile, leftService.createFile);
            wire(leftBrowser.createFolder, leftService.createFolder);
            wire(leftBrowser.docName, leftService.docName);
            wire(leftBrowser.deleteDocs, leftService.deleteDocs);
            wire(leftBrowser.selectedDocs, leftService.selectedDocs);
```

```
            wire(leftBrowser.nextCurrentFolder, leftService.srcFolder);
            wire(leftService.currentDocuments, leftBrowser.listDocs);
            wire(leftService.currentHierarchy, leftBrowser.hierarchyFolders);
            wire(n.notifications, leftBrowser.notifications);
            wire(leftBrowser.nextCurrentFolder, rightService.targetFolder);
            wire(copyToRightBtn.click, leftService.copy);

            wire(rightBrowser.createFile, rightService.createFile);
            wire(rightBrowser.createFolder, rightService.createFolder);
            wire(rightBrowser.docName, rightService.docName);
            wire(rightBrowser.deleteDocs, rightService.deleteDocs);
            wire(rightBrowser.selectedDocs, rightService.selectedDocs);
            wire(rightBrowser.nextCurrentFolder, rightService.srcFolder);
            wire(rightService.currentDocuments, rightBrowser.listDocs);
            wire(rightService.currentHierarchy, rightBrowser.hierarchyFolders);
            wire(n.notifications, rightBrowser.notifications);
            wire(rightBrowser.nextCurrentFolder, leftService.targetFolder);
            wire(copyToLeftBtn.click, rightService.copy);

            wire(this.children, leftBrowser.parent);
            wire(this.children, rightBrowser.parent);
            wire(this.children, copyToRightBtn.parent);
            wire(this.children, copyToLeftBtn.parent);
            wire(this.children, header.parent);
      }

      protected output pin children : void [*] @ layout;
}
```

Листинг 5. Изворни код корисничког интерфејса апликације.

На самом крају овог одељка, скреће се пажња читаоцу на квалификаторе аспеката односно одговорности који се појављују у декларацијама пинова *parent* и *children* одмах иза нетерминала @. У питању су квалификатори одговорности распореда (енгл. *layout*) који сугеришу да обележени (квалификовани) пинови имају одговорност распоређивања капсула на екрану. Поред квалификатора распореда постоје још и квалификатори стилизације, приступа подацима, форматирања података и валидације (енгл. *style, access, formatting* и *validation,* респективно). Структура и понашање не захтевају посебне квалификаторе. Структура корисничког интерфејса се огледа у самим капсулама, док је понашање издвојено у виду жица. На овај начин су све одговорности из оквира одговорности представљеног у овој дисертацији раздвојене у самом језику *CAPWISE.* На слици 17 приказан је дијаграм корисничког интерфејса апликације *FileSystem* на ком су

зеленом бојом обележени пинови који су квалификовани за распоред капсула на екрану, као и одговарајуће жице.




Слика 17. Дијаграм апликације *File System* са издвојеним пиновима и жицама за распоред.

## Спецификација синтаксе језика *CAPWISE*

У овом одељку, на формалан начин, биће представљена синтакса језика *CAPWISE*. Спецификација синтаксе изложена је у листингу 6 у БНФ форми (енгл. *Backus-Naur Form, BNF*[27]) са једним проширењем (зарад концизнијег записа) у виду опционог елемента '?' који се може наћи као суфикс терминалним или нетерминалним симболима. Кључне речи написане су зеленом, док су остали терминални симболи написани црвеном бојом. Како би се читалац задржао на суштинским елементима спецификације, односно због побољшања читљивости спецификације, поједини нетерминални симболи нису разрађени до краја. Они су написани бледо сивом бојом. У наредним параграфима биће описани детаљи спецификације синтаксе језика *CAPWISE* приказане у листингу 6.

```
<compilation-unit> ::= <package-declaration> <import-declarations>? <type-
declarations>

<package-declaration> ::= package <package-name> ;

<import-declarations> ::= <import-declaration> | <import-declarations> <import-
declaration>
```

[27] http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form

```
<type-declarations> ::= <type-declaration> | <type-declarations> <type-declaration>

<type-declaration> ::= <capsule-declaration> | <service-declaration>


<capsule-declaration> ::= <external-visibility-modifier> <native>? capsule <identifier> <super>? <capsule-body>

<super> ::= extends <type>

<capsule-body> ::= { <capsule-body-declarations>? }

<capsule-body-declarations> ::= <capsule-body-declaration> | <capsule-body-declarations> <capsule-body-declaration>

<capsule-body-declaration> ::= <capsule-member-declaration> | <constructor-declaration>

<constructor-declaration> ::= <constructor-header> <constructor-body>?

<constructor-header> ::= <visibility-modifier> <identifier> ( <parameter-list>? ) ;?

<capsule-member-declaration> ::= <pin-declaration> | <field-declaration> | <method-declaration>

<native> ::= native


<service-declaration> ::= service <identifier> <service-body>

<service-body> ::= { <service-body-declarations> }

<service-body-declarations> ::= <service-body-declaration> | <service-body-declarations> <service-body-declaration>

<service-body-declaration> ::= <pin-declaration>


<pin-declaration> ::= <visibility-modifier> <direction-modifier> pin <identifier> : <type> <pin-multiplicity> <pin-options>? <concern-modifiers>? ;

<pin-multiplicity> ::= [<exact-multiplicity> | <lower-multiplicity> .. <upper-multiplicity> ]

<exact-multiplicity> ::= <upper-multiplicity>

<lower-multiplicity> ::= <number>

<upper-multiplicity> ::= <number> | *

<pin-options> ::= { <uniqueness-modifier> } | { <ordering-modifier> } | { <uniqueness-modifier>, <ordering-modifier> } | { <ordering-modifier>, <uniqueness-modifier> }

<uniqueness-modifier> ::= unique | many

<ordering-modifier> ::= ordered | unordered

<direction-modifier> ::= input | output


<method-declaration> ::= <method-header> <method-body>

<method-header> ::= <internal-visibility-modifier> <type> <method-declarator>

<field-declaration> ::= <internal-visibility-modifier> <type> <variable-declarators> ;


<constructor-invocation> ::= new <type> ( <argument list>? ) ;
```

```
<service-invocation> ::= new <type> ();

<wire-invocation> ::= wire (<variable-initializer>? <pin-access-list> );
<variable-initializer> ::= new <type> ( <instances> ) ,
<pin-access-list> ::= <pin-access> | <pin-access-list> , <pin-access>
<pin-access> ::= <primary>.<identifier>

<type> ::= <type-name>

<concern-modifiers> ::= @ <concern-modifier> | <concern-modifiers>, <concern-modifier>
<concern-modifier> ::= style | access | formatting | validation | layout

<visibility-modifier> ::= <internal-visibility-modifier> | <external-visibility-
modifier>
<external-visibility-modifier> ::= public | package
<internal-visibility-modifier> ::= private | protected

<keyword> ::= capsule | extends | false | formatting | input | layout | many | native
| new | ordered | output | package | pin | private | protected | public | sap |
service | spp | style | this | true | unique | unordered | validation | void | wire
```

Листинг 6. Спецификација синтаксе језика *CAPWISE* у БНФ форми.

Језик *CAPWISE* подржава концепт пакета на начин врло сличан програмским језицима опште намене. Програмски код написан на језику *CAPWISE* распоређује се у компилационе целине (нетерминал `<compilation-unit>`) од којих свака припада тачно једном пакету. Као последица тога, свака компилациона целина мора имати декларацију припадајућег пакета на свом почетку. Коришћење елемената (односно типова) из других компилационих целина може се остварити уз претходно декларисање увоза коришћених елемената (нетерминал `<import-declarations>`).

Компилациона целина се састоји из једне или више декларација (нетерминал `<type-declarations>`) типова који могу бити или типови капсула (нетерминал `<capsule-declaration>`) или типови сервиса (нетерминал `<service-declaration>`). Декларација капсуле може бити јавна (видљива у свим пакетима) или пакетска (видљива само на нивоу припадајућег пакета). Овај тип видљивости дефинисан је у нетерминалу `<external-visibility-modifier>` кога, из описаних разлога, садржи декларација капсуле. Декларација сервиса је увек подразумевано јавна и може се користити у свим капсулама. Разлог за ово је чињеница да сервиси представљају јавни интерфејс слоја објектног простора.

Декларација капсуле садржи још делова. Кључна реч *capsule* је неизоставна и следи након описане декларације видљивости капсуле. Затим следи обавезан идентификатор (типа) капсуле који није даље елабориран. Следећи елемент декларације капсуле је опциона декларација наслеђивања. Језик *CAPWISE* подржава наслеђивање капсула, које подразумева наслеђивање поља, метода, конструктора и пинова. На крају декларације капсуле налази се декларација тела капсуле (нетерминал `<capsule-body>`) које се састоји из

произвољног броја декларација чланова (поља, метода и пинова) и декларација конструктора.

Декларација конструктора подразумева декларацију видљивости сложенију од оних код капсула (видети нетерминал `<visibility-modifier>`). Наиме, конструктори могу бити део интерфејса капсуле; у том случају имају или јавну или пакетску видљивост, као капсуле. Конструктори не морају увек бити део интерфејса капсуле; у том случају имају или заштићену или приватну видљивост која је дефинисана у нетерминалу `<internal-visibility-modifier>`. Заштићена и приватна видљивост се односе на механизам наслеђивања, односно на изведене капсуле, на начин уобичајен у објектно оријентисаним програмским језицима. Декларација конструктора капсуле подразумева и обавезан идентификатор конструктора и листу формалних параметара која може бити празна или садржати произвољан број формалних параметара. На крају, декларација конструктора капсуле подразумева и тело конструктора које садржи низ исказа који служе за креирање интерне структуре капсуле (не види се у листингу 7).

Поља и методе, чије су декларације, поред декларација конструктора и пинова, саставни делови декларације капсуле, не припадају интерфејсу капсуле. Према томе, видљивост ових елемената може бити или приватна или заштићена (видети нетерминале `<method-declaration>`, `<field-declaration>` и `<internal-visibility-modifier>`).

Са друге стране, пинови могу да буду (мада не морају, иако најчешће у пракси јесу) део интерфејса капсуле. Из тог разлога, као и конструктори, имају сложенију видљивост (видети нетерминал `<visibility-modifier>`). Декларације пинова имају још неколико важних елемената:

- Сваки пин има смер у ком преноси поруке (видети нетерминал `<direction-modifier>`) и који се дефинише у односу на капсулу или сервис којој пин припада. Смер може бити улазни или излазни (али не и један и други).
- Декларација пина укључује и обавезан идентификатор пина.
- Декларација пина укључује обавезно специфицирање типа порука које се преносе преко датог пина (видети нетерминал `<type>`). Сваки пин преноси поруке одређеног типа или свих могућих типова (тип *void*).
- Сваки пин има мултипликативност (видети нетерминал `<pin-multiplicity>`), што значи да у једној поруци може да пренесе онолико референци колико је дефинисано кроз мултипликативност пина.
- Декларација пина подразумева и неке опционе елементе. Редослед референци у колекцији која се преноси у поруци преко датог пина може бити уређен или неуређен (видети нетерминал `<ordering-modifier>`), како је то дефинисано језиком *UML2*.
- Такође, колекција референци које се преносе у порукама преко датог пина може садржати више референци ка истом објекту или то може бити забрањено (видети нетерминал `<uniqueness-modifier>`), као на језику *UML2*.
- Декларација пина може (и не мора) садржати један или више типова одговорности датог пина. Видети нетерминал `<concern-modifier>`. Овде треба напоменути да понуђене одговорности пина не садрже опције за структуру и понашање зато што структуру чине саме капсуле, док је понашање у корисничком интерфејсу представљено жицама.

Као што је већ речено, компилационе целине садрже декларације капсула и сервиса. Декларације сервиса (видети нетерминал `<service-declaration>`) односе се на елементе језика *CAPWISE* који служе за приступ подацима. Она садржи обавезну кључну реч *service*,

идентификатор сервиса и тело сервиса. Декларација тела сервиса садржи једну или више декларација пинова.

У оквиру тела конструктора и метода креира се структура капсула позивањем (енгл. *invocation*) конструктора капсула (видети нетерминал `<constructor-invocation>`) и креирањем инстанци сервиса (нетерминал `<service-invocation>`). Понашање капсула постиже се креирањем жица (нетерминал `<wire-invocation>`) које могу да спајају два компатибилна пина или вредност примитивног типа и компатибилног пина.

Систем типова података је саставни део сваког програмског језика. Језик *CAPWISE* подржава неколико уграђених типова података, као и сложене типове примитивних и композитних капсула и сервиса.

Као што се може видети на самом крају спецификације, језик *CAPWISE* садржи тридесетак кључних речи (нетерминал `<keyword>`). Нетерминални симболи који су у листингу 8 изостављени из спецификације синтаксе језика *CAPWISE* су: `<constructor-body>`, `<instances>`, `<identifier>`, `<method-body>`, `<method-declarator>`, `<number>`, `<parameter-list>`, `<primary>`, `<type name>`, `<variable-declarators>`.

Један од врло важних принципа у дизајну језика је подршка за коментаре [17]. Постоје три врсте коментара као у језику *Java*.

# *Библиотека за развој корисничких интерфејса*

У овом поглављу биће представљена претеча претходно описаног језика, односно библиотека графичких елемената имплементирана на језику *Java*, која у значајној мери задовољава захтеве који су постављени у уводном делу ове дисертације. Извесна одступања од описаног доменски специфичног језика постоје и последица су чињенице да је формализација приступа моделовању корисничких интерфејса настајала упоредо са првобитном имплементацијом ове библиотеке. Одступања су, у извесној мери, и последица незрелости саме имплементације. Наиме, имплементација ове библиотеке је управо служила као потврда изводљивости почетних концепата и средство за прикупљање искустава у пракси, из које су потом проистекла унапређења концепата описана у претходним поглављима ове тезе. Ипак, библиотека која ће бити описана у овом поглављу итекако одговара на већину пред њу постављених захтева, као и на већину критеријума постављених у овој дисертацији.

Поглавље почиње описом најважнијих пакета библиотеке, односно општим и релативно кратким прегледом њеног садржаја. Овде ће бити објашњене само најважније класе библиотеке. Објашњење конкретних графичких елемената из библиотеке биће спроведено након тога, кроз неколико примера фрагмената корисничког интерфејса различите сложености. Циљ сваког од примера је да прикаже могућности библиотеке, њену експресивност и ефикасност развоја, али и да послужи као материјал за учење како се типичне функционалности корисничког интерфејса могу направити коришћењем ове библиотеке.

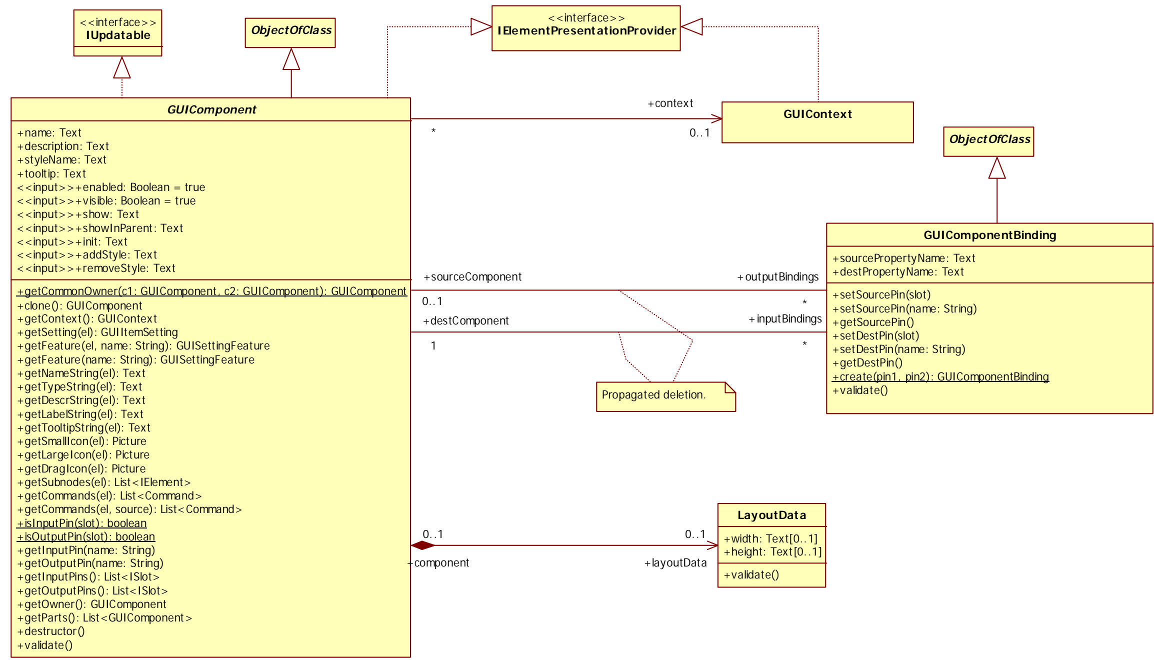
Примери ће бити изложени редом, од једноставнијих ка сложенијим. Сваки пример биће детаљно објашњен уз обавезан приказ екрана, модел на језику *UML* (уколико је потребан), а биће дат и изворни код. Уколико постоји, изворни код задужен за пословну логику биће дат одвојено од изворног кода корисничког интерфејса. У сваком наредном примеру биће поступно уведено неколико нових графичких елемената, а сваки од њих биће посебно објашњен приликом првог спомињања. Сваки пример биће изложен у виду засебне апликације. Важно је напоменути да софтверски артефакти изложени у примерима (модели и изворни код) представљају само оне и све оне артефакте које софтверски инжењер мора мануелно да направи како би добио извршиву верзију апликације.

## Основни пакети

Библиотека графичких елемената о којој је реч у овом поглављу састоји се неколико основних пакета. У наставку текста биће укратко описан сваки од њих.

### Пакет Construction

Пакет *Construction* је основни пакет који садржи апстрактну корену класу хијерархије класа графичких елемената *GUIComponent*, као и класу *GUIComponentBinding* чије инстанце представљају жице (енгл. *wire*) о којима је већ било говора у овој дисертацији. Слика 18 приказује модел овог пакета на језику *UML*.




Слика 18. Модел основног пакета библиотеке графичких елемената на језику *UML*.

Основна класа свих графичких елемената библиотеке садржи неколико заједничких атрибута и асоцијација, као и улазних пинова (у моделу, пинови су моделовани атрибутима означеним стереотипом <<*input*>> или <<*output*>>).

Сваки графички елемент има атрибуте за назив и опис (енгл. *name, description*) који немају посебан значај и препуштени су програмеру да их користи према својим потребама, на пример за анализу стабла графичких елемената или слично. Атрибут *styleName* има важну улогу у стилизацији графичких елемената јер чува назив класе из језика *CSS* помоћу које се врши стилизација графичког елемента екстерно од ове библиотеке. Подешавање овог атрибута могуће је позивом методе *setStyle(String)*. Атрибут *tooltip* подешава се методом *setTooltip(String)* и подразумева текст који ће се појавити у такозваном „облачићу“ поред графичког елемента сваки пут када корисник пређе мишем преко њега. Овај атрибут је користан ако кориснику треба објаснити сврху датог графичког елемента или начин употребе.

Графички елемент може се повезати са објектом класе *GUIContext* чиме преузима дефиниције форматирања података садржане у том објекту да важе за њега и његово подстабло уколико га има. За ово се користи метода *setContext(GUIContext)*. Више речи о класи *GUIContext* биће у наставку, у одељку које говори о њеном матичном пакету *style*.

Слично томе, графички елементи повезују се и са објектима класе *LayoutData* који садрже дефиницију позиције тих елемената на екрану. Више о томе у одељку о пакету *layout*.

Сваки графички елемент наслеђује неколико улазних пинова од класе *GUIComponent*. Вредности (*true* или *false*) на улазном пину који се може дохватити методом *ipEnabled()* дефинишу да ли ће графички елемент бити активан или не. Активно и пасивно стање графичког елемента подразумевају специфично понашање елемента које зависи од типа елемента. Вредности (*true* или *false*) на улазном пину који се може дохватити методом *ipVisible()* дефинишу да ли ће графички елемент бити видљив или не. То важи за све графичке елементе. Као што је већ споменуто, класа за стилизацију на језику *CSS* иницијално се подешава постављањем одговарајућег атрибута. Међутим, често постоји потреба да се ова класа мења у току извршавања програма, због чега класа *GUIComponent* садржи два улазна пина који се могу дохватити методама *ipAddStyle()* и *ipRemoveStyle()*. Ови улазни пинови омогућавају графичком елементу додавање или одузимање класе за стилизацију на језику *CSS* у току извршавања програма. У оба случаја, назив класе за стилизацију долази преко ових пинова као текстуална вредност. Важно је споменути још два улазна пина која контролишу видљивост графичких елемената и врло често се користе. То су пинови који се могу дохватити методама *ipShow()* и *ipShowInParent()*. Сигнал на било ком од ових пинова обезбеђује да графички елемент који га прими „исплива" из евентуално сакривеног стања. Ово је обезбеђено на тај начин што одговарајући графички елементи који имају више поделемената од којих је увек видљив само један, промене стање. Разлика је само у томе да ли цело стабло треба да промени стање (у случају *ipShow()*) или само родитељ графичког елемента који је примио сигнал (у случају *ipShowInParent()*).

За повезивање пинова графичких елемената користе се статичке методе класе *GUIComponentBinding*. Као што је већ речено, повезивањем два пина обезбеђује се да се сваки догађај и порука на излазном (изворишном) пину проследи улазном (одредишном) пину. Програмер је дужан само да повеже два, по типу компатибилна, пина. Генерално, излазни пинови свих графичких елемената се дохватају методама чији називи почињу префиксом *op,* док се улазни пинови дохватају методама са називом који почиње са *ip*. Постоје четири статичке методе у класи *GUIComponentBinding* које праве објекте ове класе и уз помоћ којих се повезују пинови:

а) *create(ISlot, ISlot)* – Користи се за повезивање два пина графичких елемената. Овде треба скренути пажњу на тип *ISlot* који представља тип пина у овој библиотеци.

б) *create(ISlot, GUIComponent, IProperty)* – Користи се за повезивање излазног пина графичког елемента (параметар типа *ISlot*), са улазним пином такозване команде (која је коришћена као нека врста претече концепта сервиса)[28]. У овом случају користи се посебна класа графичких елемената *GUICommandComponent* (са својим поткласама *GUIButtonComponent* и *GUIMenuItemComponent* о којима ће бити речи у наставку) која представља сервис (параметар типа *GUIComponent*), као и пин сервиса (параметар типа *IProperty*).

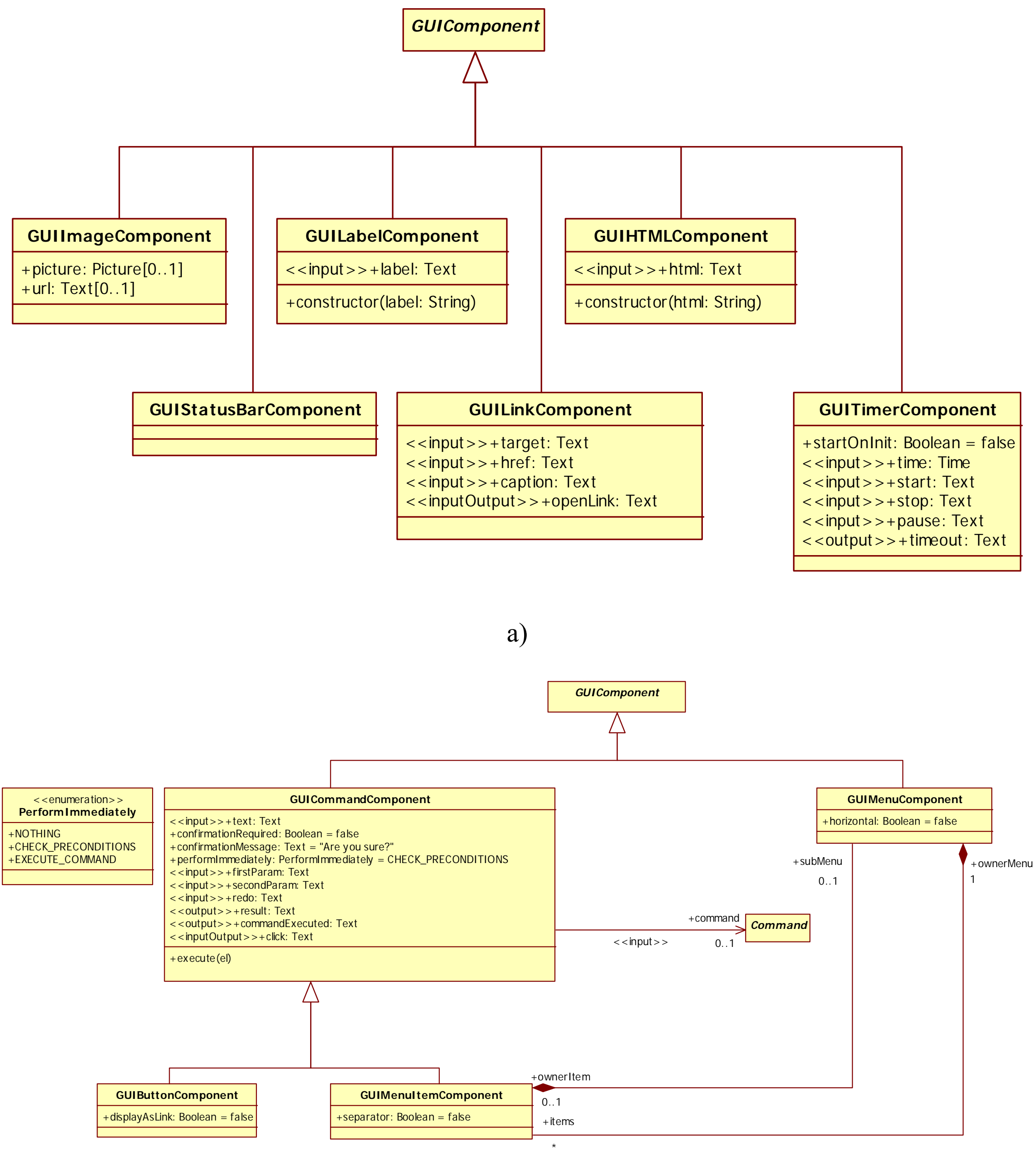
в) *create(GUIComponent, IProperty, ISlot)* – Слично као под б), али се користи за повезивање излазног пина команде са улазним пином графичког елемента.

г) *create(GUIComponent, IProperty, GUIComponent, IProperty)* – Користи се за повезивање пинова две команде.

[28] Пројектни образац „команда" (енгл. *command design pattern*) [10].

### Пакет Components

Пакет *Components* садржи већину класа видљивих графичких елемената као што су лабеле, дугмад, линкови, менији, ставке менија, дијалози, панели, табови, декови итд. Слика 19 приказује модел дела овог пакета на језику *UML*.




а)

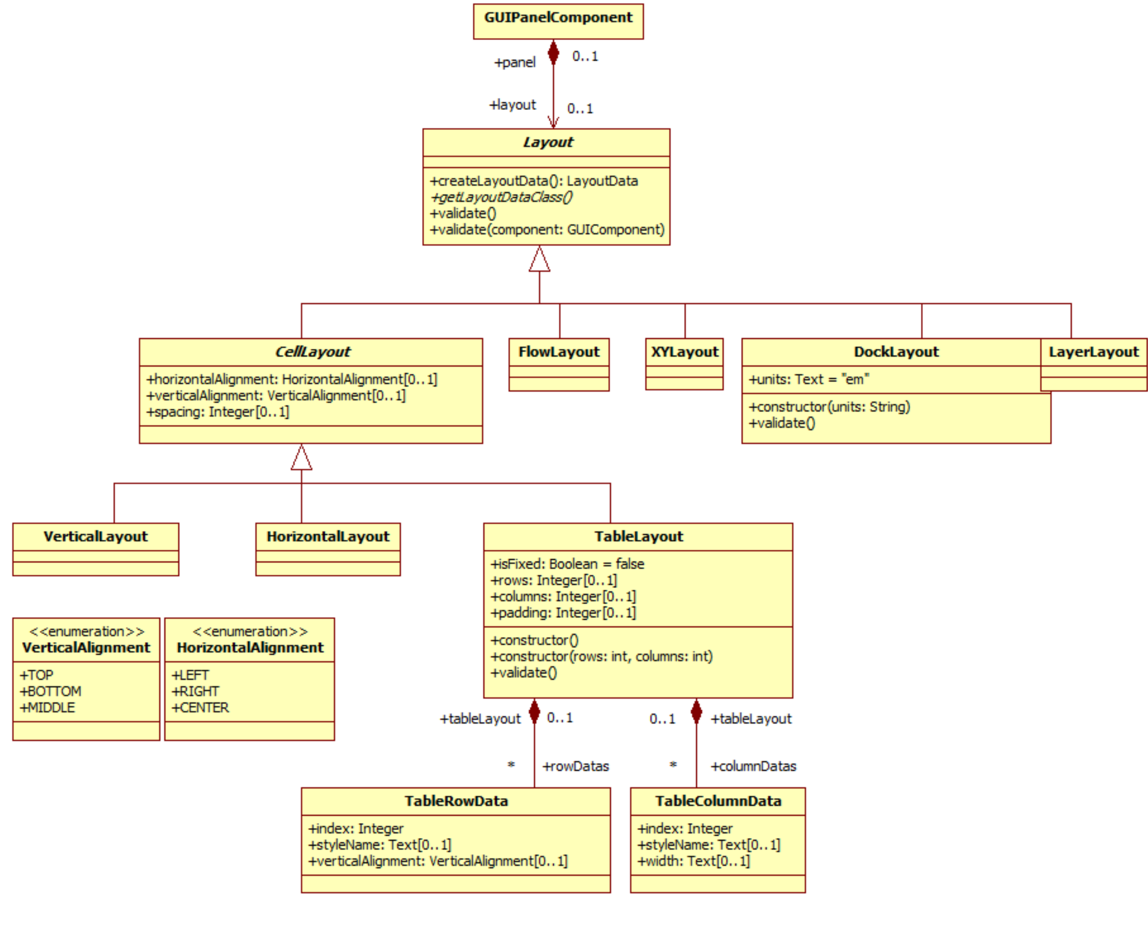
б)

в)

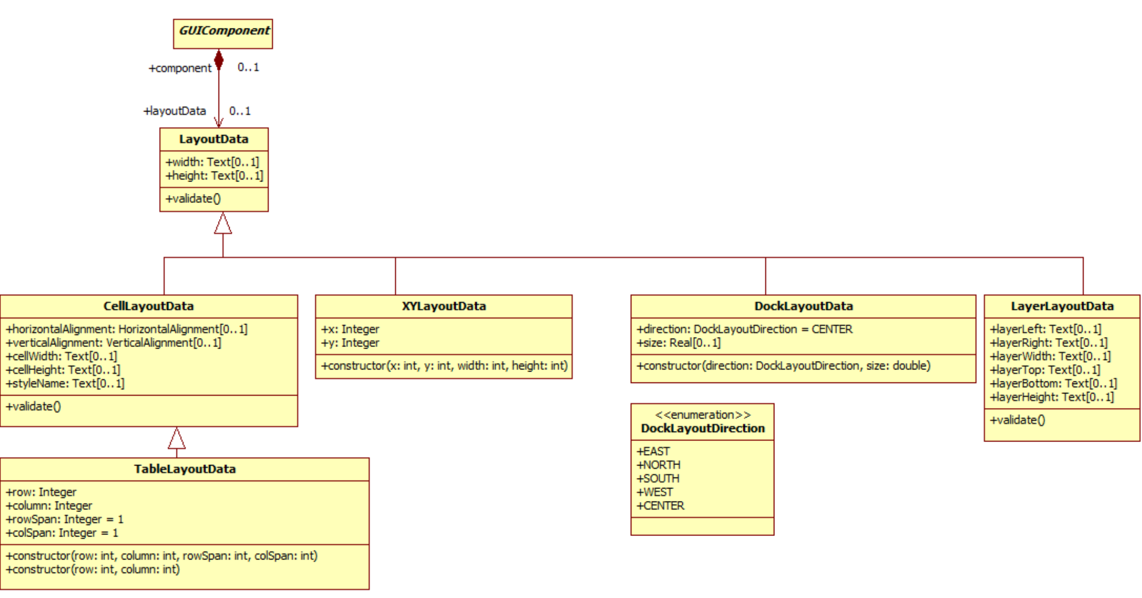
Слика 19. Модел дела библиотеке графичких елемената на језику *UML*: а) графички елементи за приказ лабела, слика, ликова, кода на језику *HTML* итд, б) графички елементи (сервиси) за извршавање акција – дугме, ставка менија итд, в) контејнерски графички елементи као што су разне врсте панела, декови, табови, дијалози итд.

### *Пакет Layout*

Пакет *layout* садржи помоћне класе одговорне за подешавање распореда графичких елемената на екрану. Класе из овог пакета грубо су подељене у две хијерархије класа. Прва (слика 20а) садржи класе свих подржаних распореда као што су вертикални (*VerticalLayout),* хоризонтални (*HorizontalLayout),* текући (*FlowLayout),* табеларни (*TableLayout)* итд, са класом *Layout* на самом врху хијерархије. Приликом програмирања корисничког интерфејса, објекти ових класа додељују се објектима контејнерских графичких елемената као што су панели (обратити пажњу на асоцијацију између класа *GUIPanelComponent* и *Layout*) и тиме се дефинише распоред елемената у панелима. Друга хијерархија садржи поткласе класе *LayoutData* чији се објекти додељују објектима графичких елемената који се налазе у оквиру неког панела да би одредили њихове специфичне особине везане за распоред дефинисан у панелу, као на пример координате ћелије табеларног распореда у којој се налази произвољни графички елемент.




а)

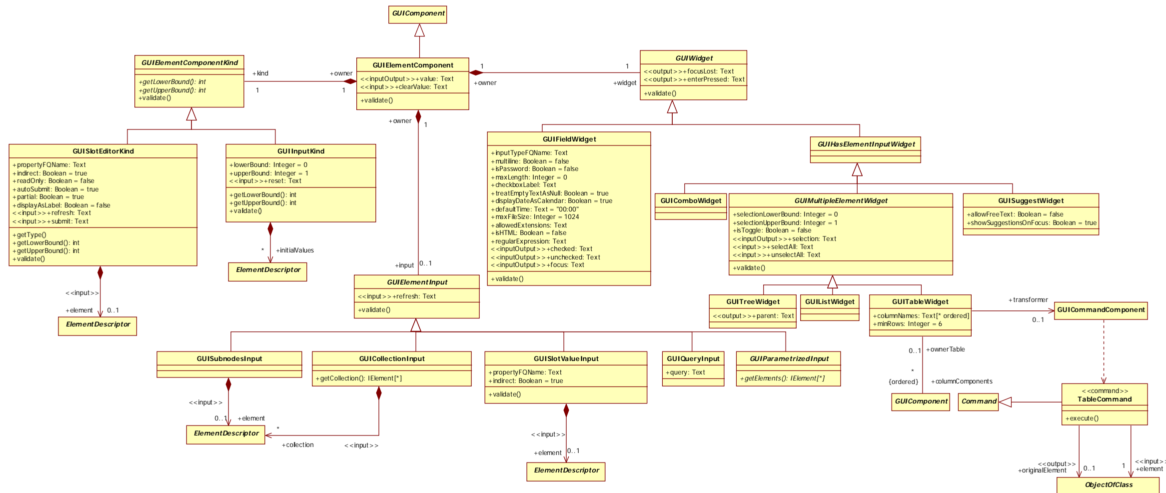



б)

Слика 20. Модел дела библиотеке са помоћним класама за дефиницију распореда на језику *UML*: а) класе распореда који могу важити у графичком елементу – панелу и б) класе које дефинишу посебна подешавања графичких елемената у смислу распореда и у зависности од панела у ком се налазе.

## Пакет ElementComponents

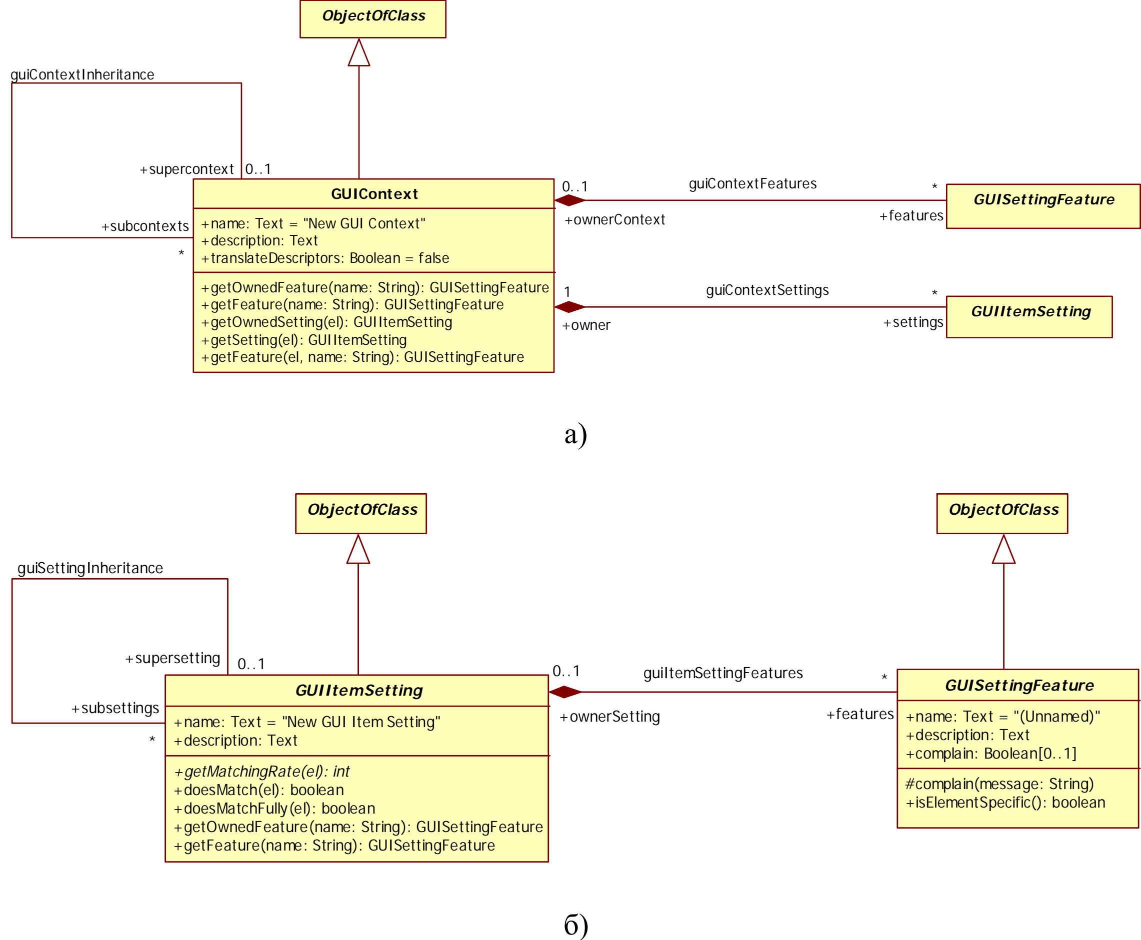
Пакет *ElementComponents* садржи једну централну класу *GUIElementComponent* која представља велики број графичких елемената који приказују, али и мењају вредности атрибута и инстанци крајева асоцијација објеката из доменског објектног простора. То су графички елементи као што су листе, стабла, табеле, поља за унос текста и многи други. С обзиром на то да ова класа има много различитих манифестација у корисничком интерфејсу, њено подешавање у жељено стање је мало захтевније и остварује се подешавањем објеката помоћних класа и њиховим везивањем за објекат ове класе. Слика 21 приказује модел у чијем је центру управо класа *GUIElementComponent,* док се око ње налазе помоћне класе. С обзиром на то да је овакав начин конфигурације графичких елемената ове класе тежак за разумевање и захтева познавање свих класа из модела са слике 21, у библиотеку су уведене две нове класе које сакривају детаље имплементације ове класе као и њену непотребну сложеност. То су класе *GUIInput* и *GUIEdit.* Ове две класе биће посебно обрађене и представљене у једном од примера који следе.




Слика 21. Модел графичког елемента *GUIElementComponent* (односно, надкласе класа *GUIInput* и *GUIEdit*) са помоћним класама на језику *UML.*

## Пакет Style

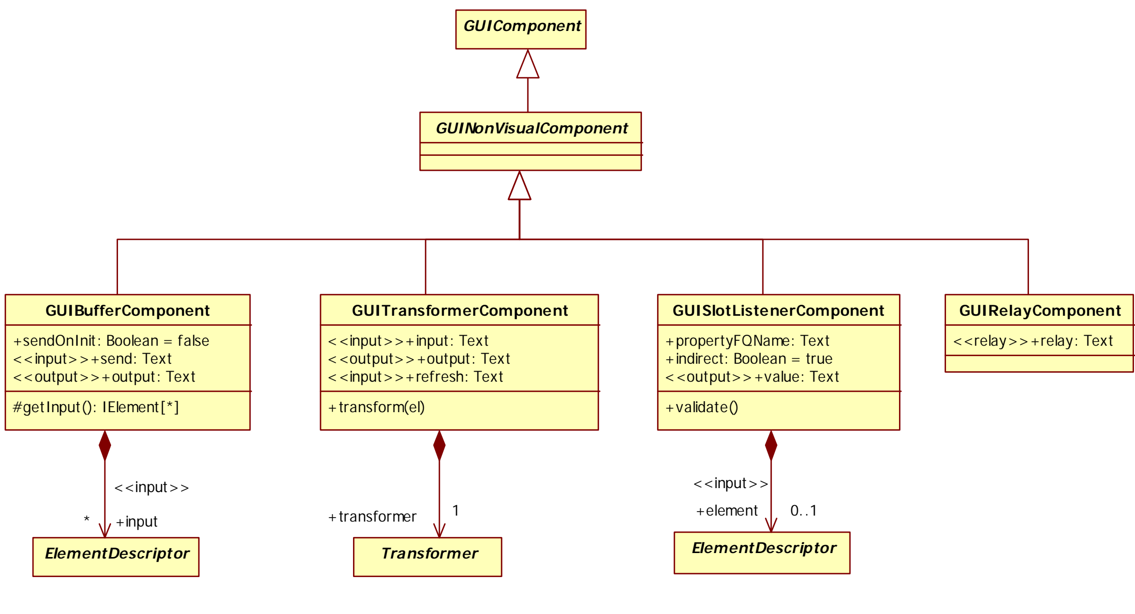
Иако сам назив пакета може да наведе на погрешан закључак да се ради о делу библиотеке који обрађује стилизацију, овде је ипак реч о пакету који обавља одговорност форматирања података. Основне класе пакета су *GUIContext, GUIItemSetting* и *GUISettingFeature.* Ове класе приказане су на слици 22.



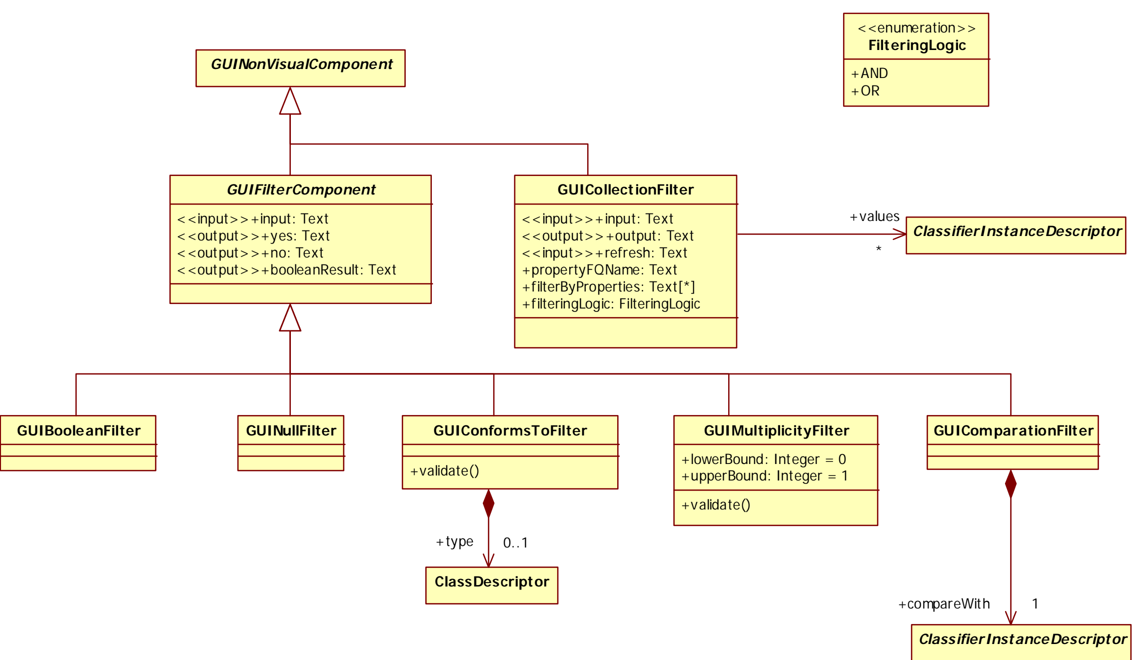

Слика 22. Модел пакета за форматирање података на језику *UML*.

### Пакет NonVisualComponents

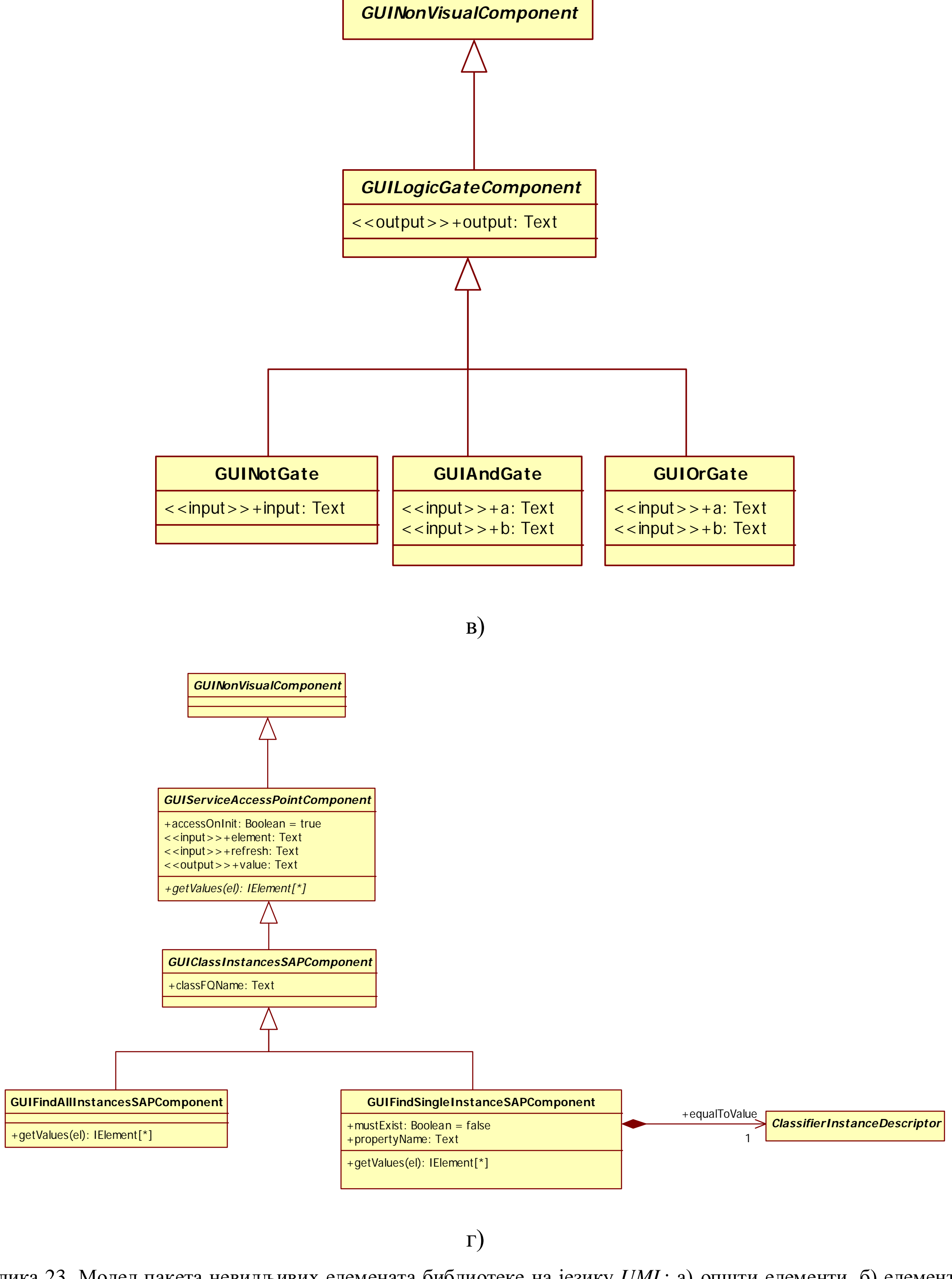
Посебан пакет елемената библиотеке представљају такозвани „невидљиви“ елементи. Овакви елементи не могу се назвати „графичким“ али свакако јесу део корисничког интерфејса. Они немају визуелну манифестацију (самим тим њихов распоред на екрану није битан, као ни форматирање података, валидација и стилизација), али с друге стране, они имају изражену компоненту понашања и динамике. Класе оваквих елемената корисничког интерфејса приказане су на слици 23.




а)




б)

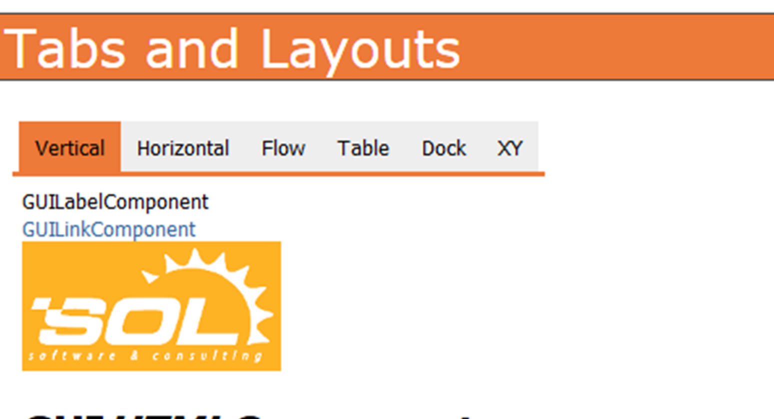



Слика 23. Модел пакета невидљивих елемената библиотеке на језику *UML*: а) општи елементи, б) елементи филтери, в) логички елементи и г) уграђени сервиси.

У наставку текста биће изложени примери употребе описане библиотеке графичких елемената као и детаљнији описи појединих класа ове библиотеке приликом првог спомињања.

### Начини распоређивања графичких елемената / таб елемент

У овом примеру биће приказано шест начина распоређивања графичких елемената. Сваки начин распоређивања биће приказан одвојено и доступан кликом на одговарајући „језичак“ графичког елемента *таба*. Слика 24 приказује један од екрана из овог примера. У овом примеру се не приступа подацима из доменског објектног простора, па према томе нема потребе за моделом објектног простора на језику *UML*. Такође, неће бити потребе ни за програмирањем пословне логике.




Слика 24. Екран који приказује вертикални распоред графичких елемената у тренутно видљивом делу графичког елемента таба.

#### Опис нових графичких елемената

Класа *GUIApplicationComponent* представља графичке елементе који се могу посматрати као веб странице или чак као целе веб апликације. Објекти ове класе увек су корен стабла графичких елемената веб странице и увек имају само један графички елемент дете – елемент који садржи све остале графичке елементе на страници.

Класа *GUIPanelComponent* представља графичке елементе који служе да садрже друге графичке елементе на корисничком интерфејсу ради њиховог распоређивања. Подржано је више начина распоређивања: хоризонтално, вертикално, табеларно, апсолутно, слојевито итд. Распоред се дефинише позивом методе *setLayout(Layout)* на објекту панела.

Класа *GUILabelComponent* представља графичке елементе који садрже и приказују произвољан текст на екрану. Текст се иницијално поставља позивом методе

*setLabel(String)*. Међутим, врло често постоји потреба да се текст лабеле мења у току извршавања апликације. То се постиже повезивањем излазног пина текстуалног типа неког другог графичког елемента са улазним пином лабеле који се дохвата методом *ipLabel()*.

Класа *GUITabComponent* представља графичке елементе сличне панелима, с том разликом да се код ове класе елемената увек види само једно дете-елемент. Промена елемента који је видљив постиже се кликом на одговарајући „језичак“ у заглављу ове компоненте. Исто се може учинити и програмски, преко пинова. Контрола видљивости поделемената базирана на индексу поделемента постиже се уз помоћ пинова који се дохватају методама *ipVisibleChild()* и *opVisibleChild()*. Контрола видљивости поделемената базирана на редоследу поделемента постиже се уз помоћ пинова који се дохватају методама *ipNext()* и *ipPrevious()*. Контрола видљивости поделемената базирана на историји видљивости поделемената постиже се уз помоћ пинова који се дохватају методама *ipBack()* и *ipForward()*. Иницијално видљив поделемент одређује се позивом методе *setVisibleChild(int)*.

Класа *GUILinkComponent* представља класичне хиперлинкове из језика *HTML*. Основне методе за конфигурацију елемената ове класе су *setHref(String), setCaption(String), setTarget(String)*. Уколико корисник кликне на овај графички елемент, он ће генерисати догађај на пину који се може дохватити позивом методе *opOpenLink()*. Преко улазних пинова који се дохватају методама *ipHref(), ipCaption()* и *ipTarget()* понашање овог графичког елемента може се мењати у току извршавања програма.

Класа *GUIImageComponent* представља елементе који приказују слике. Садржај, односно слика која се приказује постављају се једном од метода *setPicture(File)* или *setURL(String)*.

Класа *GUIHTMLComponent* представља графичке елементе који могу да садрже и прикажу произвољан сегмент кода на језику *HTML*. Иницијално, код се поставља позивом методе *setHTML(String)*, а може се променити и у току извршавања програма преко пина који се може дохватити позивом методе *ipHTML()*.

### Изворни код корисничког интерфејса

```
GUIApplicationComponent application = new GUIApplicationComponent();
application.setName("TabLayoutStaticComponents");
SoloistServiceServlet.registerApplication(application);

GUIPanelComponent root = GUIPanelComponent.createFlow(application);

GUILabelComponent title = GUILabelComponent.create(root, "Tabs and Layouts");
title.setStyle("titleStyle"); // setting CSS class name (.titleStyle)

GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

GUITabComponent tabs = GUITabComponent.create(topPanel, new String[] { "Vertical", "Horizontal", "Flow",
        "Table", "Dock", "XY" });

GUIPanelComponent verticalPanel = GUIPanelComponent.createVertical(tabs); // vertical layout
createStaticComponents(verticalPanel);

GUIPanelComponent horizontalPanel = GUIPanelComponent.createHorizontal(tabs); // horizontal layout
createStaticComponents(horizontalPanel);

GUIPanelComponent flowPanel = GUIPanelComponent.createFlow(tabs); // flow layout (HTML normal flow)
createStaticComponents(flowPanel);

GUIPanelComponent table = GUIPanelComponent.createTable(tabs); // Table Layout
int row = 0, col = 0;
GUILabelComponent.create(table, "GUILabelComponent", row, col++);
GUILinkComponent.create(table, "GUILinkComponent", "http://www.soloist4uml.com",
        TableLayoutData.create(row++, col--));
GUIImageComponent.create(table, "images/logo.png", TableLayoutData.create(row, col++));
```

```
GUIHTMLComponent.create(table, "<h1>GUI<i>HTMLComponent</i></h1>", TableLayoutData.create(row, col));

GUIPanelComponent dock = GUIPanelComponent.createDock(tabs); // Dock layout
dock.setStyle("box");
GUILabelComponent.create(dock, "GUILabelComponent", DockLayoutData.create(DockLayoutDirection.NORTH, 5));
GUILinkComponent.create(dock, "GUILinkComponent", "http://www.soloist4uml.com", "_blank",
        DockLayoutData.create(DockLayoutDirection.EAST, 9));
GUIImageComponent.create(dock, "images/logo.png", DockLayoutData.create(DockLayoutDirection.WEST, 12));
GUIHTMLComponent.create(dock, "<h1>GUI<i>HTMLComponent</i></h1>",
        DockLayoutData.create(DockLayoutDirection.SOUTH, 5));

GUIPanelComponent xy = GUIPanelComponent.createAbsolute(tabs); // XY layout
xy.setStyle("box");
GUILabelComponent.create(xy, "GUILabelComponent").setLayoutData(XYLayoutData.create(5, 10, "120px",
        "30px"));
GUILinkComponent.create(xy, "GUILinkComponent", "http://www.soloist4uml.com",
        "_blank").setLayoutData(XYLayoutData.create(155, 20, "120px", "30px"));
GUIImageComponent.create(xy, "images/logo.png").setLayoutData(XYLayoutData.create(55, 110, "160px",
        "80px"));
GUIHTMLComponent.create(xy, "<h1>GUI<i>HTMLComponent</i></h1>").setLayoutData(XYLayoutData.create(205, 50,
        "280px", "40px"));

private void createStaticComponents(GUIPanelComponent parent) {
        GUILabelComponent.create(parent, "GUILabelComponent");
        GUILinkComponent.create(parent, "GUILinkComponent", "http://www.soloist4uml.com", "_blank");
        GUIImageComponent.create(parent, "images/logo.png");
        GUIHTMLComponent.create(parent, "<h1>GUI<i>HTMLComponent</i></h1>");
}
```

Листинг 9.

### Начини распоређивања графичких елемената / дек елемент

Слично претходном примеру, и овде се приказује шест начина распоређивања графичких елемената при чему су различити распореди постављени у дек, уместо у таб графички елемент. Разлика између ова два графичка елемента је у томе што дек елемент не поседује „језичке“ на врху, па се промена видљивости графичких поделемената може остварити само програмски, односно преко пинова, а не и од стране корисника апликације директно. Због тога су у овом примеру уведени нови графички елементи – дугмићи, који примају акције корисника и прослеђују их преко пинова графичком елементу деку. Дакле, у овом примеру, између осталог, биће приказано како се спрежу пинови два графичка елемента. На слици 25 је приказан један од екрана из овог примера. У овом примеру нема пословне логике нити модела на језику *UML*.

Deck and Layouts

GUILabelComponent GUILinkComponent

**GUI*HTMLComponent***

Vertical Horizontal Flow Table Dock XY << >>

Слика 25. Екран који приказује апсолутни начин распоређивања графичких елемената у деку.

## *Опис нових графичких елемената*

Класа *GUIDeckComponent* представља графичке елементе сличне елементима класе *GUITabComponent*, с том разликом да се код ове класе елемената не виде језичци у заглављу, па се контрола видљивости одвија искључиво програмски. Контрола видљивости поделемената базирана на индексу поделемента постиже се уз помоћ пинова који се дохватају методама *ipVisibleChild()* и *opVisibleChild()*. Контрола видљивости поделемената базирана на редоследу поделемента постиже се уз помоћ пинова који се дохватају методама *ipNext()* и *ipPrevious()*. Контрола видљивости поделемената базирана на историји видљивости поделемената постиже се уз помоћ пинова који се дохватају методама *ipBack()* и *ipForward()*. Иницијално видљив поделемент одређује се позивом методе *setVisibleChild(int)*.

Класа *GUIButtonComponent* представља класичне графичке елементе који се популарно називају дугмићи. Улога овог елемента јесте да корисничке акције, конкретно клик миша на овом елементу, претвори у акцију било у самом корисничком интерфејсу, било на серверској страни (у доменској логици), или чак на оба места. Дугме сигнализира догађај клика мишем слањем поруке на излазни пин који се може дохватити методом *opClick()*. У случају када је потребно покренути акцију на серверској страни, дугме може да представља сервис. Имплементација сервиса додаје се дугмету позивом методе *setCommand(Command)* или кроз конструктор. Резултат извршавања серверске акције дугме сигнализира свом окружењу преко излазног пина *opResult()*, а сам догађај завршетка извршавања серверске акције преко излазног пина *opCommandExecuted()*.

## *Изворни код корисничког интерфејса*

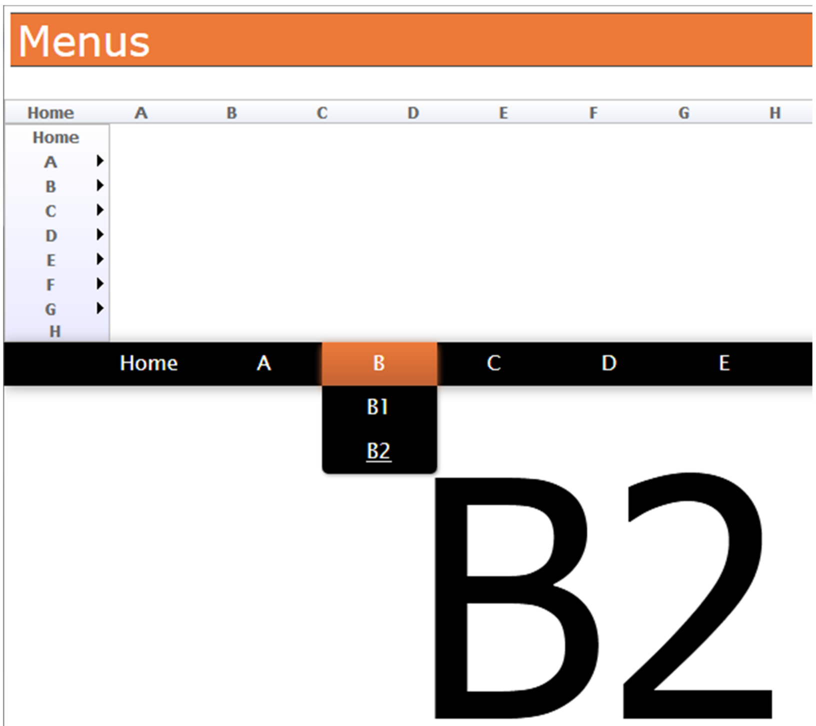
```
GUIApplicationComponent application = new GUIApplicationComponent();
application.setName("DeckLayoutStaticComponents");
```

```
SoloistServiceServlet.registerApplication(application);
GUIPanelComponent root = GUIPanelComponent.createFlow(application);

GUILabelComponent title = GUILabelComponent.create(root, "Deck and Layouts");
title.setStyle("titleStyle");

GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

GUIDeckComponent deck = GUIDeckComponent.create(topPanel);
deck.setStyle("deckH");

GUIPanelComponent verticalPanel = GUIPanelComponent.createVertical(deck);
createStaticComponents(verticalPanel);

GUIPanelComponent horizontalPanel = GUIPanelComponent.createHorizontal(deck);
createStaticComponents(horizontalPanel);

GUIPanelComponent flowPanel = GUIPanelComponent.createFlow(deck);
createStaticComponents(flowPanel);

GUIPanelComponent table = GUIPanelComponent.createTable(deck);
int row = 0, col = 0;
GUILabelComponent.create(table, "GUILabelComponent", TableLayoutData.create(row, col++));
GUILinkComponent.create(table, "GUILinkComponent", "http://www.soloist4uml.com",
        TableLayoutData.create(row++, col--));
GUIImageComponent.create(table, "images/logo.png", TableLayoutData.create(row, col++));
GUIHTMLComponent.create(table, "<h1>GUI<i>HTMLComponent</i></h1>", TableLayoutData.create(row, col));

GUIPanelComponent dock = GUIPanelComponent.createDock(deck);
dock.setStyle("box");
GUILabelComponent.create(dock, "GUILabelComponent", DockLayoutData.create(DockLayoutDirection.NORTH, 5));
GUILinkComponent.create(dock, "GUILinkComponent", "http://www.soloist4uml.com", "_blank",
        DockLayoutData.create(DockLayoutDirection.EAST, 9));
GUIImageComponent.create(dock, "images/logo.png", DockLayoutData.create(DockLayoutDirection.WEST, 12));
GUIHTMLComponent.create(dock, "<h1>GUI<i>HTMLComponent</i></h1>",
        DockLayoutData.create(DockLayoutDirection.SOUTH, 5));

GUIPanelComponent xy = GUIPanelComponent.createAbsolute(deck);
xy.setStyle("box");
GUILabelComponent.create(xy, "GUILabelComponent", XYLayoutData.create(5, 10, "120px", "30px"));
GUILinkComponent.create(xy, "GUILinkComponent", "http://www.soloist4uml.com", "_blank",
        XYLayoutData.create(155, 20, "120px", "30px"));
GUIImageComponent.create(xy, "images/logo.png", XYLayoutData.create(55, 110, "160px", "80px"));
GUIHTMLComponent.create(xy, "<h1>GUI<i>HTMLComponent</i></h1>", XYLayoutData.create(205, 50, "280px",
        "40px"));

GUIButtonComponent verticalBtn = GUIButtonComponent.create(topPanel, "Vertical");
GUIButtonComponent horizontalBtn = GUIButtonComponent.create(topPanel, "Horizontal");
GUIButtonComponent flowBtn = GUIButtonComponent.create(topPanel, "Flow");
GUIButtonComponent tableBtn = GUIButtonComponent.create(topPanel, "Table");
GUIButtonComponent dockBtn = GUIButtonComponent.create(topPanel, "Dock");
GUIButtonComponent xyBtn = GUIButtonComponent.create(topPanel, "XY");

GUIComponentBinding.create(verticalBtn.opClick(), verticalPanel.ipShow());
GUIComponentBinding.create(horizontalBtn.opClick(), horizontalPanel.ipShow());
GUIComponentBinding.create(flowBtn.opClick(), flowPanel.ipShow());
GUIComponentBinding.create(tableBtn.opClick(), table.ipShow());
GUIComponentBinding.create(dockBtn.opClick(), dock.ipShow());
GUIComponentBinding.create(xyBtn.opClick(), xy.ipShow());

GUIButtonComponent backBtn = GUIButtonComponent.create(topPanel, "<<");
GUIButtonComponent forwardBtn = GUIButtonComponent.create(topPanel, ">>");
GUIComponentBinding.create(forwardBtn.opClick(), deck.ipForward());
GUIComponentBinding.create(backBtn.opClick(), deck.ipBack());

private void createStaticComponents(GUIPanelComponent parent) {
        GUILabelComponent.create(parent, "GUILabelComponent");
        GUILinkComponent.create(parent, "GUILinkComponent", "http://www.soloist4uml.com", "_blank");
        GUIImageComponent.create(parent, "images/logo.png");
        GUIHTMLComponent.create(parent, "<h1>GUI<i>HTMLComponent</i></h1>");
}
```

Листинг 10.

## Менији

Овај врло једноставан пример пружа увид у начине вертикалне и хоризонталне организације менија, укључујући и наравно и начин програмирања подменија. У овом примеру нема пословне логике као ни модела на језику *UML*. Типичан екран из овог примера дат је на слици 26.




Слика 26. Екран који приказује начине организације и стилизације менија апликације.

### *Опис нових графичких елемената*

Класа *GUIMenuComponent* представља меније – графичке елементе који могу да садрже један или више ставки менија или сепаратора. Додавање и уклањање ставке менија постиже се позивима метода *addItem(GUIMenuITemComponent)* и *removeItem (GUIMenuITemComponent)*. Менији могу да се рендерују хоризонтално или вертикално. Начин рендеровања се подешава у методи *setHorizontal(Boolean)*.

Класа *GUIMenuItemComponent* представља графичке елементе – ставке менија. Ови елементи могу да покрену акцију у самом корисничком интерфејсу, на серверској страни, а могу чак да садрже и читаве подменије. Додавање подменија постиже се позивом методе *setSubMenu(GUIMenuComponent)*. Ставка менија сигнализира догађај клика мишем слањем поруке на излазни пин који се може дохватити методом *opClick()*. У случају када је потребно покренути акцију на серверској страни, ставка менија може да представља сервис. Имплементација сервиса додаје се ставци менија позивом методе *setCommand(Command)* или кроз конструктор. Резултат извршавања серверске акције ставка менија сигнализира свом окружењу преко излазног пина (*opResult()*) а сам догађај завршетка извршавања серверске акције преко излазног пина (*opCommandExecuted()*).

Класа *GUIBufferComponent* представник је једног дела „невидљивих“ графичких елемената. У питању је врло једноставан графички елемент, који служи да чува вредности које добије преко свог улазног пина који се може дохватити помоћу методе *ipInput()*. Вредност коју

чува овај графички елемент шаље се окружењу преко излазног пина који се може дохватити методом *opOutput()* и то само у тренутку када добије сигнал за слање преко свог улазног пина који се може дохватити методом *ipSend()*.

### *Изворни код корисничког интерфејса*

```
GUIApplicationComponent application = new GUIApplicationComponent();
application.setName("MenusSample");
SoloistServiceServlet.registerApplication(application);
GUIPanelComponent root = GUIPanelComponent.createFlow(application);

GUILabelComponent title = GUILabelComponent.create(root, "Menus");
title.setStyle("titleStyle");

GUIPanelComponent menuPanel = GUIPanelComponent.createFlow(root);
GUIDeckComponent deck = GUIDeckComponent.create(root);

strings = new String[] { "", "A", "A1", "A2", "B", "B1", "B2", "C", "C1", "C2", "D", "D1", "D2", "E", "E1",
        "E2", "F", "F1", "F2", "G", "G1", "G2", "H" };
array = new GUIPanelComponent[strings.length];
for (int i = 0; i < strings.length; i++) {
        array[i] = GUIPanelComponent.createFlow(deck);
        GUILabelComponent.create(array[i], strings[i]).setStyle("largeText");
}

createGWTMenu(menuPanel, true);
createGWTMenu(menuPanel, false);
createDropDownMenu(menuPanel);

private void createDropDownMenu(GUIPanelComponent mainPanel) {
        GUIPanelComponent menu = GUIPanelComponent.createFlow(mainPanel);
        menu.setStyle("#menu");

        GUIPanelComponent menuWrapper = GUIPanelComponent.createFlow(menu);
        menuWrapper.setStyle("#menu_wrapper");

        GUILinkComponent[] links = new GUILinkComponent[strings.length];

        for (int i = 0; i < strings.length; i++) {
                if (i == 0) {
                        links[i] = GUILinkComponent.create(menuWrapper, "Home", "javascript:void");
                        links[i].setStyle("sol-Link-active");
                } else if (i == strings.length - 1) {
                        links[i] = GUILinkComponent.create(menuWrapper, "H", "javascript:void");
                } else if (i % 3 == 1) {
                        GUIPanelComponent top = GUIPanelComponent.createFlow(menuWrapper);
                        top.setStyle("#topmenu");
                        links[i] = GUILinkComponent.create(top, strings[i], "javascript:void");
                        GUIPanelComponent sub = GUIPanelComponent.createFlow(top);
                        sub.setStyle("#submenu");
                        links[i + 1] = GUILinkComponent.create(sub, strings[i + 1], "javascript:void");
                        links[i + 2] = GUILinkComponent.create(sub, strings[i + 2], "javascript:void");
                }
                GUIComponentBinding.create(links[i].opOpenLink(), array[i].ipShow());
        }

        menuHighlighter(mainPanel, links[0], (GUILinkComponent[]) ArrayUtils.subarray(links, 1,
        strings.length));

private void menuHighlighter(GUIPanelComponent container, GUILinkComponent home, GUILinkComponent...
        orderedMenuLinks) {
        GUIBufferComponent[] highlightBuffers = new GUIBufferComponent[orderedMenuLinks.length / 3 + 1];
        GUIBufferComponent homeBuffer = GUIBufferComponent.create(container, false,
        Text.fromString("active"));
        GUIComponentBinding.create(homeBuffer.opOutput(), home.ipAddStyle()); // addStyle pin changes CSS
        class adding "active"
        GUIComponentBinding.create(home.opOpenLink(), homeBuffer.ipSend());
        for (int i = 0; i < orderedMenuLinks.length; i++) {
                if (i % 3 == 0) {
                        highlightBuffers[i / 3] = GUIBufferComponent.create(container, false,
        Text.fromString("active"));
                        GUIComponentBinding.create(highlightBuffers[i / 3].opOutput(),
        orderedMenuLinks[i].ipAddStyle());
```

```
                GUIComponentBinding.create(orderedMenuLinks[i].opOpenLink(), highlightBuffers[i / 
        3].ipSend());
                for (int j = 0; j < orderedMenuLinks.length; j += 3)
                        if (!orderedMenuLinks[j].equals(orderedMenuLinks[i]))
                                GUIComponentBinding.create(highlightBuffers[i / 3].opOutput(), 
        orderedMenuLinks[j].ipRemoveStyle());
                GUIComponentBinding.create(homeBuffer.opOutput(), 
        orderedMenuLinks[i].ipRemoveStyle());
                GUIComponentBinding.create(highlightBuffers[i / 3].opOutput(), 
        home.ipRemoveStyle());
            } else {
                GUIComponentBinding.create(orderedMenuLinks[i].opOpenLink(), highlightBuffers[i / 
        3].ipSend());
            }
        }
}

private void createGWTMenu(GUIPanelComponent mainPanel, boolean isHorizontal) {
        GUIMenuComponent myMainMenu = GUIMenuComponent.create(mainPanel, isHorizontal);
        GUIMenuItemComponent[] items = new GUIMenuItemComponent[strings.length];

        for (int i = 0; i < strings.length; i++) {
            if (i == 0) {
                items[i] = GUIMenuItemComponent.create(myMainMenu, "Home");
            } else if (i == strings.length - 1) {
                items[i] = GUIMenuItemComponent.create(myMainMenu, strings[i]);
            } else if (i % 3 == 1) {
                items[i] = GUIMenuItemComponent.create(myMainMenu, strings[i]);
                GUIMenuComponent subA = GUIMenuComponent.createSubMenu(items[i], !isHorizontal);
                items[i + 1] = GUIMenuItemComponent.create(subA, strings[i + 1]);
                items[i + 2] = GUIMenuItemComponent.create(subA, strings[i + 2]);
            }
            GUIComponentBinding.create(items[i].opClick(), array[i].ipShow());
        }
}
```

Листинг 11.

### Чаробњак (енгл. wizard)

Кроз један сложенији пример у овом поглављу биће представљен дијалог (прозор), као један важан графички елемент који се често користи у програмирању корисничких интерфејса. У овом примеру биће направљен „чаробњак“ (енгл. *wizard*) – пројектни образац који се често употребљава у случајевима када одређену функционалност корисник треба да прође у неколико корака. Овде неће бити потребан модел на језику *UML,* нити код пословне логике: све се дешава искључиво у слоју корисничког интерфејса. На слици 27 приказан је један од екрана из овог примера.

Слика 27. Екран који приказује један корак „чаробњака“ (енгл. *wizard*).

## Опис нових графичких елемената

Класа *GUIDialogComponent* представља графичке елементе – дијалоге (енгл. *pop-up dialog*) са насловом. Ови графички елементи увек имају један подelement и то панел, који садржи све остале елементе који треба да се прикажу у дијалогу. Дијалози се додају директно графичком елементу апликације позивом методе *addDialog(GUIDialogComponent)* на објекту графичког елемента апликације који је претходно објашњен.

## Изворни код корисничког интерфејса

```
GUIApplicationComponent application = new GUIApplicationComponent();
application.setName("WizardSample");
SoloistServiceServlet.registerApplication(application);

GUIPanelComponent root = GUIPanelComponent.createFlow(application);

GUILabelComponent title = GUILabelComponent.create(root, "Wizard");
title.setStyle("titleStyle");

GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

// background panel which is shown when wizard dialog pops up, aids visual effect
GUIPanelComponent blackScreen = GUIPanelComponent.createFlow(root);
blackScreen.setStyle("blackScreen");

GUIDialogComponent dialog = new GUIDialogComponent();
dialog.setModal(true);
application.addDialog(dialog);
dialog.setCaption("SOLoist Wizard");

GUIPanelComponent dialogPanel = GUIPanelComponent.createFlow(dialog);
dialogPanel.setStyle("dialogPanel");

Wizard wizard = new Wizard(dialogPanel, dialog, blackScreen);

// Panels which will be shown as steps in wizard
GUIPanelComponent p1 = new GUIPanelComponent();
p1.setLayout(new FlowLayout());
GUIPanelComponent p2 = new GUIPanelComponent();
p2.setLayout(new FlowLayout());
GUIPanelComponent p3 = new GUIPanelComponent();
p3.setLayout(new FlowLayout());
```

```
// The content can arbitrarily complex
GUILabelComponent.create(p1, "1/3").setStyle("largeText");
GUILabelComponent.create(p2, "2/3").setStyle("largeText");
GUILabelComponent.create(p3, "3/3").setStyle("largeText");

// Add pages to wizard in order
wizard.addPage(p1);
wizard.addPage(p2);
wizard.addPage(p3);

// Generate the wizard GUI
wizard.generate();

GUIButtonComponent cmdStart = GUIButtonComponent.create(topPanel, "Start wizard");

GUIComponentBinding.create(cmdStart.opClick(), wizard.getBufferDarkenBackground().ipSend());
GUIComponentBinding.create(cmdStart.opClick(), wizard.getFirstChild().ipShowInParent());
GUIComponentBinding.create(cmdStart.opClick(), wizard.getFirstButtonChild().ipShowInParent());

GUIComponentBinding.create(cmdStart.opClick(), dialog.ipShow());
GUIComponentBinding.create(cmdStart.opResult(), dialogPanel.ipRelay1());

GUIComponentBinding.create(cmdStart.opResult(), p1.ipRelay1());
GUIComponentBinding.create(cmdStart.opResult(), p2.ipRelay1());
GUIComponentBinding.create(cmdStart.opResult(), p3.ipRelay1());

public class Wizard {

        private List<GUIPanelComponent> pages = new ArrayList<GUIPanelComponent>();
        private GUIDeckComponent mainDeck;
        private GUIDeckComponent buttonDeck;
        private GUIDialogComponent dialog;
        private GUIBufferComponent bufferSet;
        private GUIBufferComponent bufferRemove;
        private GUIPanelComponent blackScreen;
        private GUIPanelComponent rootPanel;

        public Wizard(GUIPanelComponent rootPanel, GUIDialogComponent dialog, GUIPanelComponent
        blackScreen) {
                this.rootPanel = rootPanel;
                this.dialog = dialog;
                this.blackScreen = blackScreen;
        }

        public void addPage(GUIPanelComponent page) {
                pages.add(page);
        }

        public void generate() {
                bufferSet = GUIBufferComponent.create(rootPanel, false,
        Text.fromString("darkenBackground"));
                bufferRemove = GUIBufferComponent.create(rootPanel, false,
        Text.fromString("darkenBackground"));

                GUIComponentBinding.create(bufferSet.opOutput(), blackScreen.ipAddStyle());
                GUIComponentBinding.create(bufferRemove.opOutput(), blackScreen.ipRemoveStyle());

                GUIPanelComponent mainPanel = GUIPanelComponent.createFlow(rootPanel);
                GUIPanelComponent btnPanel = GUIPanelComponent.createFlow(rootPanel);

                mainDeck = GUIDeckComponent.create(mainPanel);
                mainPanel.setStyle("wizardMain");

                buttonDeck = GUIDeckComponent.create(btnPanel);
                btnPanel.setStyle("wizardButtons");

                for (int i = 0; i < pages.size(); i++) {
                        mainDeck.add(pages.get(i));
                        GUIPanelComponent.createFlow(buttonDeck);
                }

                for (int i = 0; i < pages.size(); i++) {
```

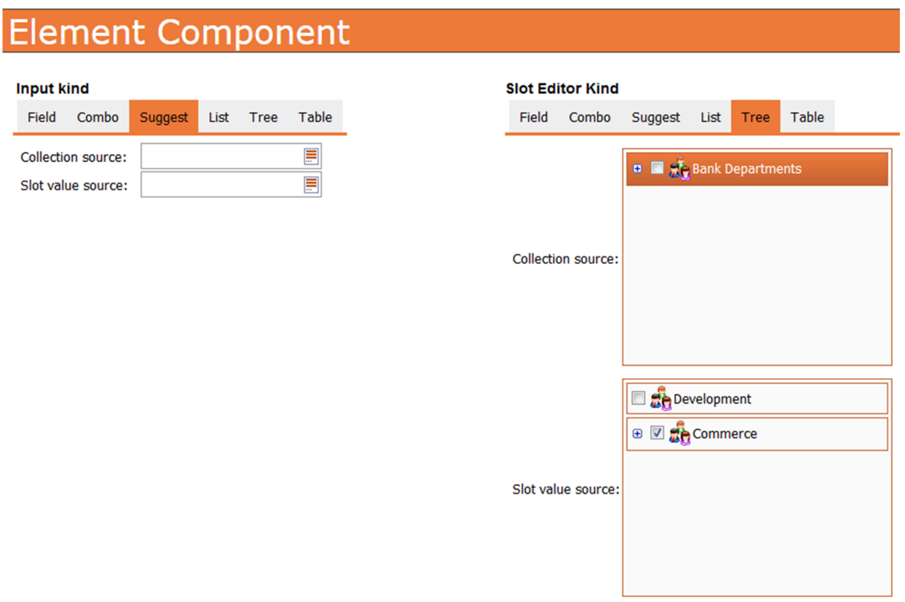
```
                    GUIPanelComponent buttonPanel = (GUIPanelComponent) buttonDeck.get(i);

                    GUIButtonComponent btnCancel = GUIButtonComponent.create(buttonPanel, "Cancel");
                    btnCancel.setStyle("cancelButton");
                    btnCancel.setConfirmationRequired(true);
                    btnCancel.setConfirmationMessage("Cancel?");

                    GUIComponentBinding.create(btnCancel.opClick(), dialog.ipHide());
                    GUIComponentBinding.create(btnCancel.opClick(), bufferRemove.ipSend());

                    if (i != 0) {
                            GUIButtonComponent btnPrev = GUIButtonComponent.create(buttonPanel, "<
        Back");
                            GUIComponentBinding.create(btnPrev.opClick(), mainDeck.get(i -
        1).ipShow());
                            GUIComponentBinding.create(btnPrev.opClick(), buttonDeck.get(i -
        1).ipShow());
                            btnPrev.setStyle("previousButton");
                    }

                    if (i == pages.size() - 1) {
                            GUIButtonComponent btnFinish = GUIButtonComponent.create(buttonPanel,
        "Finish");
                            btnFinish.setStyle("finishButton");
                            btnFinish.setConfirmationMessage("Finish?");
                            btnFinish.setConfirmationRequired(true);
                            GUIComponentBinding.create(btnFinish.opClick(), dialog.ipHide());
                            GUIComponentBinding.create(btnFinish.opClick(), bufferRemove.ipSend());
                    } else {
                            GUIButtonComponent btnNext = GUIButtonComponent.create(buttonPanel, "Next
        >");
                            btnNext.setStyle("nextButton");
                            GUIComponentBinding.create(btnNext.opClick(), mainDeck.get(i +
        1).ipShow());
                            GUIComponentBinding.create(btnNext.opClick(), buttonDeck.get(i +
        1).ipShow());
                    }
            }
        }

        public GUIBufferComponent getBufferDarkenBackground() {
                return bufferSet;
        }

        public GUIPanelComponent getFirstChild() {
                return (GUIPanelComponent) mainDeck.get(0);
        }

        public GUIPanelComponent getFirstButtonChild() {
                return (GUIPanelComponent) buttonDeck.get(0);
        }
}
```

Листинг 12.

### Графички елемент за приказ, унос и измену података

Сви графички елементи који се могу добити употребом класа *GUIInput* и *GUIEdit* (фасаде око класе *GUIElementComponent*) приказани су у овом примеру на систематичан начин. У левом делу екрана (слика 28) приказане су све примене класе *GUIInput,* а у десном примене класе *GUIEdit*; и то поља за унос, комбо поља, поља за сугестију, листе, стабла и табеле. Разлика између графичких елемената са једне и са друге стране екрана није у визуелној презентацији већ у приступу подацима. Наиме, елементи са леве стране екрана само читају вредности атрибута и инстанци крајева асоцијација објеката из домена, док елементи са десне стране поред приказивања (читања) раде и измене вредности атрибута и инстанци крајева асоцијација. У овом примеру неће бити потребе за програмирањем пословне

логике, док је модел на језику *UML* приказан одмах након описа графичких елемената који се први пут појављују у овом примеру.




Слика 28. Екран који приказује неколико варијација графичког елемента за приказ, унос и измену података.

### Опис нових графичких елемената

Класа *GUIFindAllInstancesSAPComponent* представља сервис који дохвата све објекте једне класе. Класа објеката који се желе дохватити поставља се у методи фабрици (енгл. *factory method*): *create(GUIContainerComponent parent, IClass cls)*. Дохваћени објекти прослеђују се окружењу овог графичког елемента преко његовог излазног пина који се може дохватити позивом методе *opValue()*.

Класа *GUIFindSingleInstanceSAPComponent* представља сервис који дохвата тачно један објекат једне доменске класе. Доменска класа поставља се у методи фабрици слично као код претходно описане класе графичких елемената. Јединствени објекат дефинише се вредношћу једног од атрибута. Атрибут се поставља позивом методе *setProperty(IProperty)*, док се вредност поставља позивом методе *setEqualToValue(IClassifierInstance)*. Дохваћени објекат прослеђује се окружењу овог графичког елемента преко његовог излазног пина који се може дохватити позивом методе *opValue()*.

Класа *GUICommandComponent* представља сервис коме се мора доделити класа која га имплементира. Ово се постиже позивом методе *setCommand(Command)* или кроз конструктор. Резултат извршавања серверске акције овај елемент сигнализира свом окружењу преко излазног пина *opResult()*, а сам догађај завршетка извршавања серверске акције преко излазног пина *opCommandExecuted()*.

Класа *GUIComparationFilter* представља једну од класа невидљивих елемената који обављају функције филтрирања вредности које добију преко улазног пина који се може дохватити методом *ipInput()*. Ако вредност са овог улазног пина задовољава услов који зависи од типа елемента за филтрирање (у овом случају тип је *GUIComparationFilter*), онда се *boolean* вредност *true* сигнализира на излазном пину *opBooleanResult();* у супротном, сигнализира се *false*. Такође, та иста вредност са улазног пина се прослеђује у случају задовољења услова на излазни пин *opYes()*, а у случају да услов није задовољен на излазни пин *opNo()*. Филтрирање се обавља сваки пут када се нова вредност појави на улазном пину *ipInput()*. Ово понашање је заједничко за све врсте елемената за филтрирање, једино се

разликују услови који зависе од типа елемента. У случају типа *GUIComparationFilter,* вредност на улазном пину пореди се по једнакости са вредношћу која се поставља позивом методе *setComparedWith(IClassifierInstance).*

У случају типа *GUINullFilter,* вредност са улазног пина пореди се са вредношћу *null.*

Код типа *GUIMultiplicityFilter* проверава се да ли број вредности на улазном пину (као што је објашњено у претходним поглављима, преко пинова се преносе колекције вредности) одговара опсегу дефинисаном вредностима које се постављају позивом метода *setLowerBound(int)* и *setUpperBound(int).*

У случају типа *GUIBooleanFilter* проверава се да ли је *boolean* вредност са улазног пина једнака вредности *true* или *false*.

Код типа *GUIConformsToFilter* врши се провера типа улазне вредности, тако што се проверава да ли је у питању инстанца типа који се поставља позивом методе *setType(IClassifier).*

Класа *GUIInput,* заједно са класом *GUIEdit* која ће бити објашњена у наставку, представља једну од најважнијих класа графичких елемената ове библиотеке. Ова класа обухвата широку лепезу графичких елемената као што су поља (енгл. *fields*), листе, табеле, стабла, поља за сугестију (енгл. *suggestion box*) итд. Ови графички елементи служе кориснику апликације да изабере или дефинише једну или више вредности из доменског објектног простора апликације. Вредности се могу изабрати из колекције вредности, као на пример избором вредности у листи објеката, или се могу дефинисати од стране корисника, на пример куцањем броја или неког текста у текстуалном пољу. Изабране вредности се сигнализирају окружењу овог графичког елемента преко излазног пина који се може дохватити методом *opValue(),* и то сваку пут када се изабрана или дефинисана вредност промени.

Класа *GUIEdit* представља графичке елементе који визуелно изгледају исто као њихови парњаци из класе елемената *GUIInput.* Међутим, елементи класе *GUIEdit* имају додатно својство да уписују новоизабране вредности директно у доменски објектни простор. То уписивање се подразумевано догађа након сваке измене (промена селекције, промена текста у пољу итд.), што се може променити позивом методе *setAutoSubmit(boolean)* и тиме одложити до тренутка пријема сигнала на улазном пину *ipSubmit().*

Наредних неколико класа представљају подршку за форматирање података, како инстанци примитивних, тако и инстанци сложених типова. Објекти ових класа не представљају графичке елементе, већ конфигурацију форматирања инстанци типова података које се појављују у графичким елементима као објекти у листи, стаблу, табели, или као вредности (нпр. бројеви, новчани износи итд.) у пољима за унос или у тексту негде на страници.

Објекат класе *GUIContext* користи се као корени чвор или садржалац једне групе дефиниција форматирања података [26]. Додавањем овог објекта било ком чвору, односно графичком елементу из стабла графичких елемената, постиже се да дефиниције форматирања које садржи објекат класе *GUIContext* постану важеће управо у подстаблу графичког елемента ком је додељен.

Класа *GUIObjectSetting* [26] најчешће се употребљава за дефинисање форматирања објеката једне класе, али може да послужи и за дефинисање форматирања свих објеката или само једног објекта из доменског објектног простора. Објекат ове класе додаје се у контекст форматирања позивом методе *addSetting(GUIItemSetting)* на објекту контекста. Дефиниција форматирања састоји се од једног или више објеката апстрактне класе *GUISettingFeature* који, у зависности од типа конкретне класе, одређују један аспект форматирања. Ови

објекти додају се објекту класе *GUIObjectSetting* позивом методе *addFeature(GUISettingFeature)*. У наставку биће објашњене конкретне поткласе ове класе.

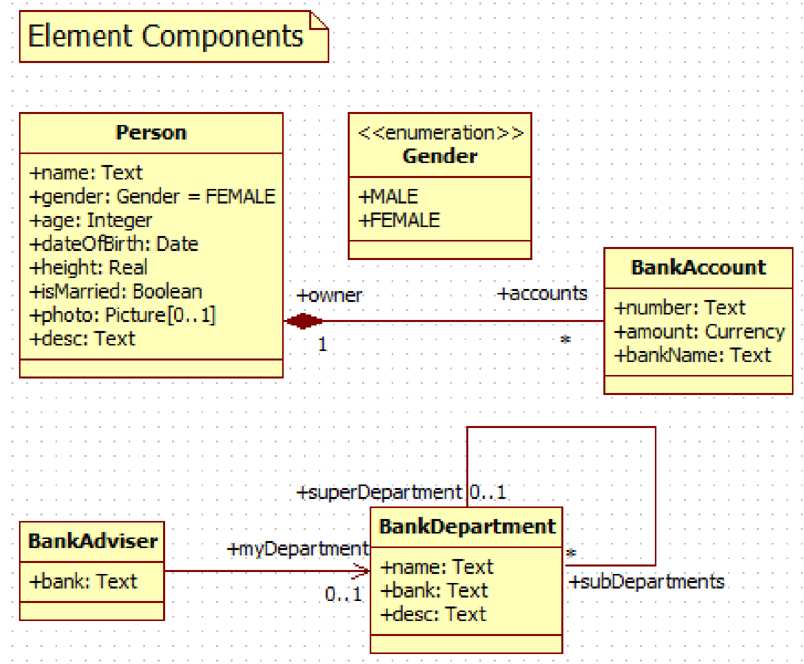
Објекат класе *GUITextFeature* дефинише начин текстуалне презентације објекта неке класе тако што у најчешћем броју случајева дефинише атрибут те класе из ког ће бити прочитан текст који ће бити употребљен као део те текстуалне репрезентације. Репрезентација једног објекта може имати више делова, па самим тим и више текстуалних елемената. Поред текста, репрезентација објекта неке класе може садржати и слику, иконицу, или сл.

У том случају, користи се класа *GUIPictureFeature* која дефинише како се дохвата слика или иконица која ће бити употребљена у репрезентацији објекта на екрану. Најчешће је у питању иконица која је иста за све објекте једне класе, али понекад се слика чита и из самог објекта, па тако сваки објекат те класе може имати посебну иконицу. На пример, особе врло често приказујемо личном фотографијом уместо генеричком иконицом која је иста за све особе.

Класа *GUINavigatorFeature* се користи за форматирање објеката који се појављују у оквиру графичких елемената типа стабла. Прецизније речено, дефинише се који доменски објекти треба да се појаве у подстаблу неког објекта из стабла. На пример, објекат класе Одељење (енгл. *Department*) треба да садржи у свом подстаблу објекте исте класе који представљају пододељења тог одељења.

### *Модел на језику UML*

На слици 29 приказан је дијаграм класа на језику *UML*. Овај модел приказује особе са својим рачунима у банци, као и запослене у банци и њихову припадност одељењима која формирају хијерархију. У изворном коду овог примера могу се уочити места где се директно реферишу елементи приказаног модела.




Слика 29. Модел података на језику *UML*: класе особа, рачун, запослени (у банци) и одељење (енгл. *Person, BankAccount, BankAdviser, BankDepartment,* респективно).

### *Изворни код корисничког интерфејса*

```
GUIApplicationComponent page = new GUIApplicationComponent();
```

```
page.setName("ElementComponent");
SoloistServiceServlet.registerApplication(page);
page.setContext(createContextAndStyles());

GUIPanelComponent root = GUIPanelComponent.createFlow(page);

GUILabelComponent title = GUILabelComponent.create(root, "Element Component");
title.setStyle("titleStyle");

GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

// Service Access Points
GUIFindAllInstancesSAPComponent allPersons = GUIFindAllInstancesSAPComponent.create(root,
        Person.CLASSIFIER);
GUIFindSingleInstanceSAPComponent findJohnDoe = GUIFindSingleInstanceSAPComponent.create(root,
        Person.FQ_TYPE_NAME, Person.FQPropertyNames.desc,
        (ClassifierInstanceDescriptor)ElementDescriptor.create(Text.fromString(ObjectSpaceInit.JOHN_DOE_IDE
        NTIFIER)));
GUIFindSingleInstanceSAPComponent findJohnDeoRootDepartment =
        GUIFindSingleInstanceSAPComponent.create(root, BankDepartment.FQ_TYPE_NAME,
        BankDepartment.FQPropertyNames.desc,
        (ClassifierInstanceDescriptor)ElementDescriptor.create(Text.fromString(ObjectSpaceInit.JOHN_DOE_ROO
        T_DEPARTMENT)));
GUICommandComponent allPersonsNotAdvisers = GUICommandComponent.create(root, new FetchPersonsOnly(),
        PerformImmediately.NOTHING);
GUIComponentBinding.create(page.opStart(), allPersonsNotAdvisers.ipClick());
GUIFindSingleInstanceSAPComponent findJanneRoe = GUIFindSingleInstanceSAPComponent.create(root,
        Person.FQ_TYPE_NAME, Person.FQPropertyNames.desc,
        (ClassifierInstanceDescriptor)ElementDescriptor.create(Text.fromString(ObjectSpaceInit.JANNE_ROE_ID
        ENTIFIER)));

// Inputs
GUIPanelComponent inputs = GUIPanelComponent.createFlow(topPanel);
inputs.setStyle("leftSide");
GUILabelComponent l1 = GUILabelComponent.create(inputs, "Input kind");
l1.setStyle("subtitle");
GUITabComponent tabInput = GUITabComponent.create(inputs, "Field", "Combo", "Suggest", "List", "Tree",
        "Table");
GUIPanelComponent inputFieldsTable = GUIPanelComponent.createTable(tabInput);
GUIPanelComponent inputComboTable = GUIPanelComponent.createTable(tabInput);
GUIPanelComponent inputSuggestTable = GUIPanelComponent.createTable(tabInput);
GUIPanelComponent inputListsTable = GUIPanelComponent.createTable(tabInput);
GUIPanelComponent inputTreesTable = GUIPanelComponent.createTable(tabInput);
GUIPanelComponent inputTablesTable = GUIPanelComponent.createTable(tabInput);

int i = 0;
GUIComparationFilter gcf0 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabInput.opVisibleChild(), gcf0.ipInput());
GUIComponentBinding.create(gcf0.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcf1 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabInput.opVisibleChild(), gcf1.ipInput());

GUIComponentBinding.create(gcf1.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcf2 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabInput.opVisibleChild(), gcf2.ipInput());
GUIComponentBinding.create(gcf2.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcf3 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabInput.opVisibleChild(), gcf3.ipInput());
GUIComponentBinding.create(gcf3.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcf4 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabInput.opVisibleChild(), gcf4.ipInput());
GUIComponentBinding.create(gcf4.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcf5 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabInput.opVisibleChild(), gcf5.ipInput());
GUIComponentBinding.create(gcf5.opYes(), allPersons.ipRefresh());

// Input fields
int row = 0, col = 0;
GUILabelComponent.create(inputFieldsTable, "Text: ", row, col);
GUIInput.createField(inputFieldsTable, Text.CLASSIFIER, row++, col + 1);

GUILabelComponent.create(inputFieldsTable, "Integer: ", row, col);
```

```
GUIInput.createField(inputFieldsTable, Integer.CLASSIFIER, row++, col + 1);

GUILabelComponent.create(inputFieldsTable, "Real: ", row, col);
GUIInput.createField(inputFieldsTable, Real.CLASSIFIER, row++, col + 1);

GUILabelComponent.create(inputFieldsTable, "Date: ", row, col);
GUIInput.createField(inputFieldsTable, Date.CLASSIFIER, row++, col + 1);

GUILabelComponent.create(inputFieldsTable, "Boolean: ", row, col);
GUIInput checkBox = GUIInput.createField(inputFieldsTable, Boolean.CLASSIFIER, row++, col + 1);
checkBox.setLowerBound(1);
checkBox.addInitialValue(Boolean.FALSE);

GUILabelComponent.create(inputFieldsTable, "Enum: ", row, col);
GUIInput.createField(inputFieldsTable, Repository.getRepository().getEnumeration(Gender.FQ_TYPE_NAME),
        row++, col + 1);

// Input combo
row = 0; col = 0;
GUILabelComponent.create(inputComboTable, "Collection source: ", row, col);
GUIInput collectionInputCombo = GUIInput.createCombo(inputComboTable, TableLayoutData.create(row++, col +
        1));
GUIComponentBinding.create(allPersons.opValue(), collectionInputCombo.ipCollection());

GUILabelComponent.create(inputComboTable, "Slot value source: ", row, col);
GUIInput slotValueInputCombo = GUIInput.createCombo(inputComboTable, Person.PROPERTIES.accounts,
        TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJohnDoe.opValue(), slotValueInputCombo.ipSlotValueElement());

// Input suggest
row = 0; col = 0;
GUILabelComponent.create(inputSuggestTable, "Collection source: ", row, col);
GUIInput collectionInputSuggest = GUIInput.createSuggest(inputSuggestTable, TableLayoutData.create(row++,
        col + 1));
GUIComponentBinding.create(allPersons.opValue(), collectionInputSuggest.ipCollection());

GUILabelComponent.create(inputSuggestTable, "Slot value source: ", row, col);
GUIInput slotValueInputSuggest = GUIInput.createSuggest(inputSuggestTable, Person.PROPERTIES.accounts,
        TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJohnDoe.opValue(), slotValueInputSuggest.ipSlotValueElement());

// Input lists
row = 0; col = 0;
GUILabelComponent.create(inputListsTable, "Collection source: ", row, col);
GUIInput collectionInputList = GUIInput.createList(inputListsTable);
collectionInputList.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(allPersons.opValue(), collectionInputList.ipCollection());
collectionInputList.setSize("250px", "200px");
collectionInputList.setStyle("listWidget");

GUILabelComponent.create(inputListsTable, "Slot value source: ", row, col);
GUIInput slotValueInputList = GUIInput.createList(inputListsTable, Person.PROPERTIES.accounts);
slotValueInputList.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJohnDoe.opValue(), slotValueInputList.ipSlotValueElement());
slotValueInputList.setSize("250px", "200px");
slotValueInputList.setStyle("listWidget");

// Input trees
row = 0; col = 0;
GUILabelComponent.create(inputTreesTable, "Collection source: ", row, col);
GUIInput personsTree = GUIInput.createTree(inputTreesTable);
personsTree.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(allPersons.opValue(), personsTree.ipCollection());
personsTree.setSize("250px", "200px");
personsTree.setStyle("treeWidget");

GUILabelComponent.create(inputTreesTable, "Slot value source: ", row, col);
GUIInput slotValueInputTree = GUIInput.createTree(inputTreesTable,
        BankDepartment.PROPERTIES.subDepartments);
slotValueInputTree.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJohnDeoRootDepartment.opValue(), slotValueInputTree.ipSlotValueElement());
slotValueInputTree.setSize("250px", "200px");
slotValueInputTree.setStyle("treeWidget");
```

```
// Input tables
row = 0; col = 0;
GUILabelComponent.create(inputTablesTable, "Collection source: ", row, col);

GUIEdit genderValue = GUIEdit.createField(null, Person.PROPERTIES.gender);
genderValue.setReadOnly(true);
genderValue.setDisplayAsLabel(true);

GUIEdit dateOfBirthValue = GUIEdit.createField(null, Person.PROPERTIES.dateOfBirth);
dateOfBirthValue.setReadOnly(true);
dateOfBirthValue.setDisplayAsLabel(true);

GUIEdit isMarriedValue = GUIEdit.createField(null, Person.PROPERTIES.isMarried);
isMarriedValue.setReadOnly(true);
isMarriedValue.setDisplayAsLabel(true);

GUIInput collectionInputTable = GUIInput.createTable(inputTablesTable, "Person", "Gender", "Date of Birth",
        "Is Married");
collectionInputTable.setColumnComponents(genderValue, dateOfBirthValue, isMarriedValue);
collectionInputTable.setMinRows(3);
collectionInputTable.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(allPersons.opValue(), collectionInputTable.ipCollection());

GUILabelComponent.create(inputTablesTable, "Slot value source: ", row, col);

GUIEdit bankNameValue = GUIEdit.createField(null, BankAccount.PROPERTIES.bankName);
bankNameValue.setReadOnly(true);
bankNameValue.setDisplayAsLabel(true);

GUIInput slotValueInputTable = GUIInput.createTable(inputTablesTable, Person.PROPERTIES.accounts, "Bank
        Account", "Bank Name");
slotValueInputTable.setColumnComponents(bankNameValue);
slotValueInputTable.setMinRows(3);
slotValueInputTable.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJohnDoe.opValue(), slotValueInputTable.ipSlotValueElement());

// Edits
GUIPanelComponent edits = GUIPanelComponent.createFlow(topPanel);
GUILabelComponent l2 = GUILabelComponent.create(edits, "Slot Editor Kind");
l2.setStyle("subtitle   ");
GUITabComponent tabEdit = GUITabComponent.create(edits, "Field", "Combo", "Suggest", "List", "Tree",
        "Table");
GUIPanelComponent editFieldsTable = GUIPanelComponent.createTable(tabEdit);
GUIPanelComponent editComboTable = GUIPanelComponent.createTable(tabEdit);
GUIPanelComponent editSuggestTable = GUIPanelComponent.createTable(tabEdit);
GUIPanelComponent editListsTable = GUIPanelComponent.createTable(tabEdit);
GUIPanelComponent editTreesTable = GUIPanelComponent.createTable(tabEdit);
GUIPanelComponent editTablesTable = GUIPanelComponent.createTable(tabEdit);

i = 0;
GUIComparationFilter gcfe0 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabEdit.opVisibleChild(), gcfe0.ipInput());
GUIComponentBinding.create(gcfe0.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcfe1 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabEdit.opVisibleChild(), gcfe1.ipInput());
GUIComponentBinding.create(gcfe1.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcfe2 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabEdit.opVisibleChild(), gcfe2.ipInput());
GUIComponentBinding.create(gcfe2.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcfe3 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabEdit.opVisibleChild(), gcfe3.ipInput());
GUIComponentBinding.create(gcfe3.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcfe4 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabEdit.opVisibleChild(), gcfe4.ipInput());
GUIComponentBinding.create(gcfe4.opYes(), allPersons.ipRefresh());
GUIComparationFilter gcfe5 = GUIComparationFilter.create(root, Integer.valueOf(i++));
GUIComponentBinding.create(tabEdit.opVisibleChild(), gcfe5.ipInput());
GUIComponentBinding.create(gcfe5.opYes(), allPersons.ipRefresh());

// Edit fields
row = 0; col = 0;
GUILabelComponent.create(editFieldsTable, "Text: ", row, col);
```

```
GUIEdit textEditor = GUIEdit.createField(editFieldsTable, Person.PROPERTIES.name, row++, col + 1);

GUILabelComponent.create(editFieldsTable, "Integer: ", row, col);
GUIEdit integerEditor = GUIEdit.createField(editFieldsTable, Person.PROPERTIES.age, row++, col + 1);

GUILabelComponent.create(editFieldsTable, "Real: ", row, col);
GUIEdit realEditor = GUIEdit.createField(editFieldsTable, Person.PROPERTIES.height, row++, col + 1);

GUILabelComponent.create(editFieldsTable, "Date: ", row, col);
GUIEdit dateEditor = GUIEdit.createField(editFieldsTable, Person.PROPERTIES.dateOfBirth, row++, col + 1);

GUILabelComponent.create(editFieldsTable, "Boolean: ", row, col);
GUIEdit booleanEditor = GUIEdit.createField(editFieldsTable, Person.PROPERTIES.isMarried, row++, col + 1);

GUILabelComponent.create(editFieldsTable, "Enum: ", row, col);
GUIEdit enumEditor = GUIEdit.createField(editFieldsTable, Person.PROPERTIES.gender, row++, col + 1);

GUILabelComponent.create(editFieldsTable, "Picture: ", row, col);
GUIPanelComponent photoUpload = GUIEdit.createFile(editFieldsTable, Person.PROPERTIES.photo, true, false);
photoUpload.setRowColumn(row++, col + 1);

GUILabelComponent.create(editFieldsTable, "Picture Value: ", row, col);
GUIPanelComponent photoValue = GUIEdit.createPicture(editFieldsTable, Person.PROPERTIES.photo, false);
photoValue.setRowColumn(row++, col + 1);

GUIComponentBinding.create(findJohnDoe.opValue(), textEditor.ipElement());
GUIComponentBinding.create(findJohnDoe.opValue(), integerEditor.ipElement());
GUIComponentBinding.create(findJohnDoe.opValue(), realEditor.ipElement());
GUIComponentBinding.create(findJohnDoe.opValue(), dateEditor.ipElement());
GUIComponentBinding.create(findJohnDoe.opValue(), booleanEditor.ipElement());
GUIComponentBinding.create(findJohnDoe.opValue(), enumEditor.ipElement());
GUIComponentBinding.create(findJohnDoe.opValue(), photoUpload.ipRelay1());
GUIComponentBinding.create(findJohnDoe.opValue(), photoValue.ipRelay1());

// Edit lists
row = 0; col = 0;
GUILabelComponent.create(editListsTable, "Collection source: ", row, col);
GUIEdit collectionEditList = GUIEdit.createList(editListsTable, BankAdviser.PROPERTIES.myClients);
collectionEditList.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(allPersonsNotAdvisers, FetchPersonsOnly.PROPERTIES.fetchedPersons,
        collectionEditList.ipCollection());
GUIComponentBinding.create(findJohnDoe.opValue(), collectionEditList.ipElement());
collectionEditList.setSize("250px", "200px");
collectionEditList.setStyle("listEditor");

GUILabelComponent.create(editListsTable, "Slot value source: ", row, col);
GUIEdit slotValueEditList = GUIEdit.createList(editListsTable, Person.PROPERTIES.successfulAdvisers,
        Person.PROPERTIES.myBankAdvisers);
slotValueEditList.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJanneRoe.opValue(), slotValueEditList.ipElement());
GUIComponentBinding.create(findJanneRoe.opValue(), slotValueEditList.ipSlotValueElement());
slotValueEditList.setSize("250px", "200px");
slotValueEditList.setStyle("listEditor");

// Edit combos
row = 0; col = 0;
GUILabelComponent.create(editComboTable, "Collection source: ", row, col);
GUIEdit collectionEditCombo = GUIEdit.createCombo(editComboTable, BankAdviser.PROPERTIES.myClients,
        TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(allPersonsNotAdvisers, FetchPersonsOnly.PROPERTIES.fetchedPersons,
        collectionEditCombo.ipCollection());
GUIComponentBinding.create(findJohnDoe.opValue(), collectionEditCombo.ipElement());

GUILabelComponent.create(editComboTable, "Slot value source: ", row, col);
GUIEdit slotValueEditCombo = GUIEdit.createCombo(editComboTable, Person.PROPERTIES.successfulAdvisers,
        Person.PROPERTIES.myBankAdvisers, TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJanneRoe.opValue(), slotValueEditCombo.ipElement());
GUIComponentBinding.create(findJanneRoe.opValue(), slotValueEditCombo.ipSlotValueElement());

// Edit suggests
GUILabelComponent.create(editSuggestTable, "Collection source: ", row, col);
GUIEdit collectionEditSuggest = GUIEdit.createSuggest(editSuggestTable, BankAdviser.PROPERTIES.myClients,
        TableLayoutData.create(row++, col + 1));
```

```
GUIComponentBinding.create(allPersonsNotAdvisers, FetchPersonsOnly.PROPERTIES.fetchedPersons,
        collectionEditSuggest.ipCollection());
GUIComponentBinding.create(findJohnDoe.opValue(), collectionEditSuggest.ipElement());

GUILabelComponent.create(editSuggestTable, "Slot value source: ", row, col);
GUIEdit slotValueEditSuggest = GUIEdit.createSuggest(editSuggestTable,
        Person.PROPERTIES.successfulAdvisers, Person.PROPERTIES.myBankAdvisers,
        TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJanneRoe.opValue(), slotValueEditSuggest.ipElement());
GUIComponentBinding.create(findJanneRoe.opValue(), slotValueEditSuggest.ipSlotValueElement());

// Edit trees
row = 0; col = 0;
GUILabelComponent.create(editTreesTable, "Collection source: ", row, col);
GUIEdit collectionEditTree = GUIEdit.createTree(editTreesTable, BankAdviser.PROPERTIES.myDepartment);
collectionEditTree.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJohnDoe.opValue(), collectionEditTree.ipElement());
GUIComponentBinding.create(findJohnDeoRootDepartment.opValue(), collectionEditTree.ipCollection());
collectionEditTree.setSize("250px", "200px");
collectionEditTree.setStyle("treeEditor");

GUILabelComponent.create(editTreesTable, "Slot value source: ", row, col);
GUIEdit slotValueEditTree = GUIEdit.createTree(editTreesTable, BankAdviser.PROPERTIES.myDepartment,
        BankDepartment.PROPERTIES.subDepartments);
slotValueEditTree.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJohnDoe.opValue(), slotValueEditTree.ipElement());
GUIComponentBinding.create(findJohnDeoRootDepartment.opValue(), slotValueEditTree.ipSlotValueElement());
slotValueEditTree.setSize("250px", "200px");
slotValueEditTree.setStyle("treeEditor");

// Edit tables
row = 0; col = 0;
GUILabelComponent.create(editTablesTable, "Collection source: ", row, col);
GUIEdit genderEditor = GUIEdit.createField(null, Person.PROPERTIES.gender);
GUIEdit dateOfBirthEditor = GUIEdit.createField(null, Person.PROPERTIES.dateOfBirth);
GUIEdit isMarriedEditor = GUIEdit.createField(null, Person.PROPERTIES.isMarried);

GUIInput collectionEditTable = GUIInput.createTable(editTablesTable, "Person", "Gender", "Date of Birth",
        "Is Married");
collectionEditTable.setColumnComponents(genderEditor, dateOfBirthEditor, isMarriedEditor);
collectionEditTable.setMinRows(3);
collectionEditTable.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(allPersons.opValue(), collectionEditTable.ipCollection());

GUILabelComponent.create(editTablesTable, "Slot value source: ", row, col);
GUIEdit bankNameEditor = GUIEdit.createField(null, BankAccount.PROPERTIES.bankName);

GUIInput slotValueEditTable = GUIInput.createTable(editTablesTable, Person.PROPERTIES.accounts, "Bank
        Account", "Bank Name");
slotValueEditTable.setColumnComponents(bankNameEditor);
slotValueEditTable.setMinRows(3);
slotValueEditTable.setLayoutData(TableLayoutData.create(row++, col + 1));
GUIComponentBinding.create(findJohnDoe.opValue(), slotValueEditTable.ipSlotValueElement());

private GUIContext createContextAndStyles() {
        GUIContext context = new GUIContext();
        DefaultContextInit.getRoot().addContext(context);

        GUIObjectSetting person = GUIObjectSetting.create(context, Person.CLASSIFIER);
        GUITextFeature.createName(person, Person.PROPERTIES.name);
                GUITextFeature.createFixedSeparator(person, ":");
        GUITextFeature.createDescription(person, Person.PROPERTIES.age);
        GUIPictureFeature.createSmallIcon(person, "images/icons/person.png");
        GUINavigatorFeature.createSubnodes(person, Person.FQPropertyNames.accounts);

        GUIObjectSetting bankAccount = GUIObjectSetting.create(context, BankAccount.CLASSIFIER);
        GUITextFeature.createName(bankAccount, BankAccount.PROPERTIES.number);
        GUITextFeature.createFixedSeparator(bankAccount, "/");
        GUITextFeature.createDescription(bankAccount, BankAccount.PROPERTIES.amount);
        GUIPictureFeature.createSmallIcon(bankAccount, "images/icons/bankaccount.png");

        GUIObjectSetting department = GUIObjectSetting.create(context, BankDepartment.CLASSIFIER);
        GUITextFeature.createName(department, BankDepartment.PROPERTIES.name);
```

```
        GUIPictureFeature.createSmallIcon(department, "images/icons/bankdepartment.png");
        GUINavigatorFeature.createSubnodes(department, BankDepartment.FQPropertyNames.subDepartments);

        return context;
}
```

Листинг 13.

### Форма за снимање података о особи

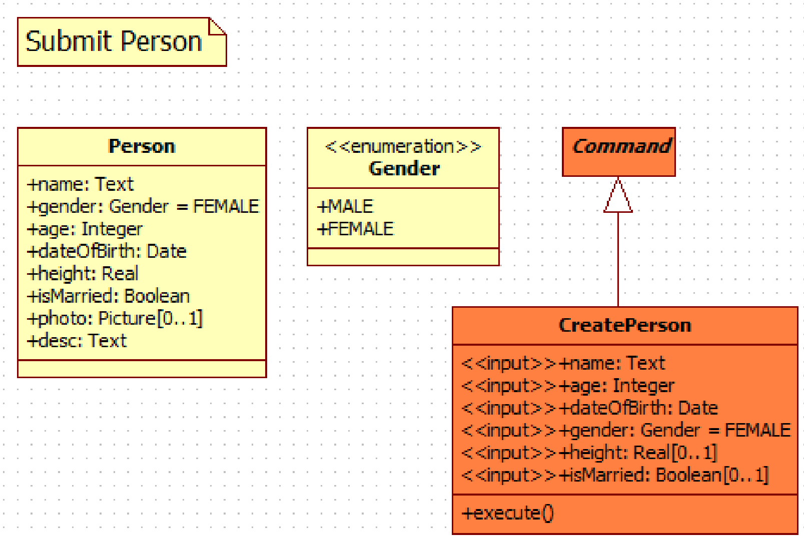
Након детаљног упознавања са графичким елементима класа *GUIInput* и *GUIEdit,* следи један једноставан пример који демонстрира употребу сервиса и њихове сарадње са елементима класе *GUIInput.* Укратко, у овом примеру демонстрира се начин прављења форме за унос података о особи, као и серверски део целог случаја употребе. Сервис је представљен дугметом (са лабелом *Create Person*) док се његова имплементација налази у класи *CreatePerson* из модела на језику *UML* (видети слику 31). На слици 30 приказан је екран из овог примера.



Слика 30. Екран који приказује форму за унос особе.

#### *Модел на језику UML*

Дијаграм класа са слике 31 приказује врло једноставан модел – класу особа и командну класу – сервис.




Слика 31. Модел података на језику *UML*: класа особа и команда за креирање особе (енгл. *Person, CreatePerson,* респективно).

## Изворни код пословне логике

```
class CreatePerson {

public void execute() {
        Text nameVal = name.val();
        Integer ageVal = age.val();
        Date dateOfBirthVal = dateOfBirth.val();
        Gender genderVal = gender.val();
        Real heightVal = height.val();
        Boolean isMarriedVal = isMarried.val();

        if (nameVal == null || nameVal.isEmpty()) {
                throw new CommandPreconditionsException("Please, type in the name for new person.");
        }

        Person p = new Person();
        p.name.set(nameVal);
        p.age.set(ageVal);
        p.dateOfBirth.set(dateOfBirthVal);
        p.gender.set(genderVal);
        p.height.set(heightVal);
        p.isMarried.set(isMarriedVal);
}
}
```

Листинг 14.

## Изворни код корисничког интерфејса

```
GUIApplicationComponent page = new GUIApplicationComponent();
page.setName("SubmitPerson");
SoloistServiceServlet.registerApplication(page);
page.setContext(DefaultContextInit.getRoot());

GUIPanelComponent root = GUIPanelComponent.createFlow(page);

GUILabelComponent title = GUILabelComponent.create(root, "Submit Person");
title.setStyle("titleStyle");

GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

GUIPanelComponent table = GUIPanelComponent.createTable(topPanel);

int row = 0;
```

```
GUILabelComponent.create(table, "Name: ", row, 0);
GUIInput nameInput = GUIInput.createField(table, Text.CLASSIFIER, row++, 1);
nameInput.addInitialValue(Text.fromString("New Person"));

GUILabelComponent.create(table, "Gender: ", row, 0);
GUIInput genderInput = GUIInput.createField(table,
        Repository.getRepository().getEnumeration(Gender.FQ_TYPE_NAME), row++, 1);

GUILabelComponent.create(table, "Age: ", row, 0);
GUIInput ageInput = GUIInput.createField(table, Integer.CLASSIFIER, row++, 1);

GUILabelComponent.create(table, "Date of birth: ", row, 0);
GUIInput dateOfBirthInput = GUIInput.createField(table, Date.CLASSIFIER, row++, 1);

GUILabelComponent.create(table, "Height [m]: ", row, 0);
GUIInput heightInput = GUIInput.createField(table, Real.CLASSIFIER, row++, 1);

GUILabelComponent.create(table, "Is married: ", row, 0);
GUIInput isMarriedInput = GUIInput.createField(table, Boolean.CLASSIFIER, row++, 1); // true is initial
        value
isMarriedInput.setLowerBound(1);
isMarriedInput.addInitialValue(Boolean.TRUE);

// Now let's create the buttons
GUIPanelComponent hp = GUIPanelComponent.createHorizontal(topPanel);

GUIButtonComponent createButton = GUIButtonComponent.create(hp, "Create Person", new CreatePerson());
createButton.setConfirmationRequired(true);
createButton.setConfirmationMessage("Are you sure you want to create a person?");

// Let's now bind the button with input parameters' provider components
GUIComponentBinding.create(nameInput.opValue(), createButton, CreatePerson.PROPERTIES.name);
GUIComponentBinding.create(ageInput.opValue(), createButton, CreatePerson.PROPERTIES.age);
GUIComponentBinding.create(genderInput.opValue(), createButton, CreatePerson.PROPERTIES.gender);
GUIComponentBinding.create(dateOfBirthInput.opValue(), createButton, CreatePerson.PROPERTIES.dateOfBirth);
GUIComponentBinding.create(heightInput.opValue(), createButton, CreatePerson.PROPERTIES.height);
GUIComponentBinding.create(isMarriedInput.opValue(), createButton, CreatePerson.PROPERTIES.isMarried);

// Let's reset input fields after command execution
GUIComponentBinding.create(createButton.opCommandExecuted(), nameInput.ipReset());
GUIComponentBinding.create(createButton.opCommandExecuted(), ageInput.ipReset());
GUIComponentBinding.create(createButton.opCommandExecuted(), genderInput.ipReset());
GUIComponentBinding.create(createButton.opCommandExecuted(), dateOfBirthInput.ipReset());
GUIComponentBinding.create(createButton.opCommandExecuted(), heightInput.ipReset());
GUIComponentBinding.create(createButton.opCommandExecuted(), isMarriedInput.ipReset());

// Let's now create clear and reset form buttons
// Note the difference between reset and clear: reset re-sets the initial values in the input fields, while
        clear actually makes them empty
GUIButtonComponent clearButton = GUIButtonComponent.create(hp, "Clear Form");
GUIComponentBinding.create(clearButton.opClick(), nameInput.ipClearValue());
GUIComponentBinding.create(clearButton.opClick(), ageInput.ipClearValue());
GUIComponentBinding.create(clearButton.opClick(), genderInput.ipClearValue());
GUIComponentBinding.create(clearButton.opClick(), dateOfBirthInput.ipClearValue());
GUIComponentBinding.create(clearButton.opClick(), heightInput.ipClearValue());
GUIComponentBinding.create(clearButton.opClick(), isMarriedInput.ipClearValue());

GUIButtonComponent resetButton = GUIButtonComponent.create(hp, "Reset Form");
GUIComponentBinding.create(resetButton.opClick(), nameInput.ipReset());
GUIComponentBinding.create(resetButton.opClick(), ageInput.ipReset());
GUIComponentBinding.create(resetButton.opClick(), genderInput.ipReset());
GUIComponentBinding.create(resetButton.opClick(), dateOfBirthInput.ipReset());
GUIComponentBinding.create(resetButton.opClick(), heightInput.ipReset());
GUIComponentBinding.create(resetButton.opClick(), isMarriedInput.ipReset());
```

Листинг 15.

### Табела са могућностима измене детаља

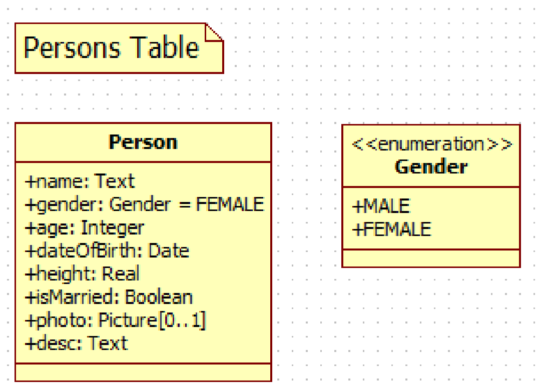
Претходни пример приказао је како се може направити фрагмент корисничког интерфејса за измену детаља једне особе. У овом примеру се демонстрира како се могу мењати детаљи више особа истовремено. На слици 32 приказан је екран из овог примера. Као што се може

приметити, у примеру је показано како се праве табеле са детаљима особа, при чему се у горњој табели детаљи особа могу чак и мењати. Свака измена вредности у графичком елементу и напуштање фокуса снимиће нову вредност атрибута у објекат, односно у особу.

Слика 32. Екран који приказује табелу за измену података о особи (горе) и приказ података о особи (доле).

### *Модел на језику UML*

На дијаграму са слике 33 представљен је модел на језику *UML* идентичан оном из претходног примера који приказује само класу *Person* (особа) и енумерацију *Gender* (пол).




Слика 33. Модел података на језику *UML:* класа Особа (енгл. *Person*).

### *Изворни код корисничког интерфејса*

```
GUIApplicationComponent page = new GUIApplicationComponent();
page.setName("PersonsTable");
SoloistServiceServlet.registerApplication(page);
GUIContext context = createContextAndStyles();
```

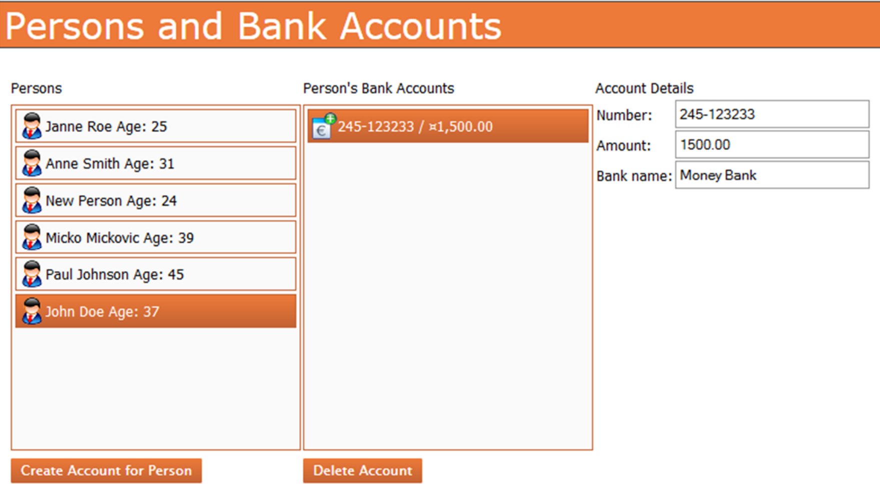
```
page.setContext(context);

GUIPanelComponent root = GUIPanelComponent.createVertical(page);

GUILabelComponent title = GUILabelComponent.create(root, "Persons Table");
title.setStyle("titleStyle");

GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

// This component fetches all persons
GUIFindAllInstancesSAPComponent allPersons = GUIFindAllInstancesSAPComponent.create(root,
        Person.CLASSIFIER);

GUILabelComponent.create(topPanel, "Editable fields").setStyle("subtitle");
createPersonsTable(topPanel, allPersons, false, false);
GUILabelComponent.create(topPanel, "Readonly fields").setStyle("subtitle");
createPersonsTable(topPanel, allPersons, true, false);
GUILabelComponent.create(topPanel, "Readonly fields as labels").setStyle("subtitle");
createPersonsTable(topPanel, allPersons, true, true);

private void createPersonsTable(GUIPanelComponent root, GUIFindAllInstancesSAPComponent allPersons, boolean
        readonly, boolean displayAsLabel) {
// column components
GUIEdit nameEditor = GUIEdit.createField(null, Person.PROPERTIES.name);
nameEditor.setReadOnly(readonly);
nameEditor.setDisplayAsLabel(displayAsLabel);

GUIEdit genderEditor = GUIEdit.createField(null, Person.PROPERTIES.gender);
genderEditor.setReadOnly(readonly);
genderEditor.setDisplayAsLabel(displayAsLabel);

GUIEdit ageEditor = GUIEdit.createField(null, Person.PROPERTIES.age);
ageEditor.setReadOnly(readonly);
ageEditor.setDisplayAsLabel(displayAsLabel);

GUIEdit dateOfBirthEditor = GUIEdit.createField(null, Person.PROPERTIES.dateOfBirth);
dateOfBirthEditor.setReadOnly(readonly);
dateOfBirthEditor.setDisplayAsLabel(displayAsLabel);

GUIEdit heightEditor = GUIEdit.createField(null, Person.PROPERTIES.height);
heightEditor.setReadOnly(readonly);
heightEditor.setDisplayAsLabel(displayAsLabel);

GUIEdit isMarriedEditor = GUIEdit.createField(null, Person.PROPERTIES.isMarried);
isMarriedEditor.setReadOnly(readonly);
isMarriedEditor.setDisplayAsLabel(displayAsLabel);

GUIInput editableTable = GUIInput.createTable(root, "Person", "Name", "Gender", "Age", "Date of Birth",
        "Height", "Is Married");
editableTable.setColumnComponents(nameEditor, genderEditor, ageEditor, dateOfBirthEditor, heightEditor,
        isMarriedEditor);
editableTable.setMinRows(3); // min rows to show if there is no data in the table
GUIComponentBinding.create(allPersons.opValue(), editableTable.ipCollection());

private GUIContext createContextAndStyles() {
        GUIContext context = new GUIContext();
        DefaultContextInit.getRoot().addContext(context);

        GUIObjectSetting person = GUIObjectSetting.create(context, Person.CLASSIFIER);
        GUITextFeature.createName(person, Person.PROPERTIES.name);
        GUIPictureFeature.createSmallIcon(person, "images/icons/person.png");
        return context;
}
```

Листинг 16.

### Креирање, читање, измена и брисање

У овом примеру биће представљен један врло интерактиван и функционално богат кориснички интерфејс. Пример приказује особе са њиховим банкарским рачунима и

детаљима рачуна који се могу мењати. Такође, могуће је направити нови рачун за особу, као и избрисати рачун. На слици 34 приказан је екран из овог примера.



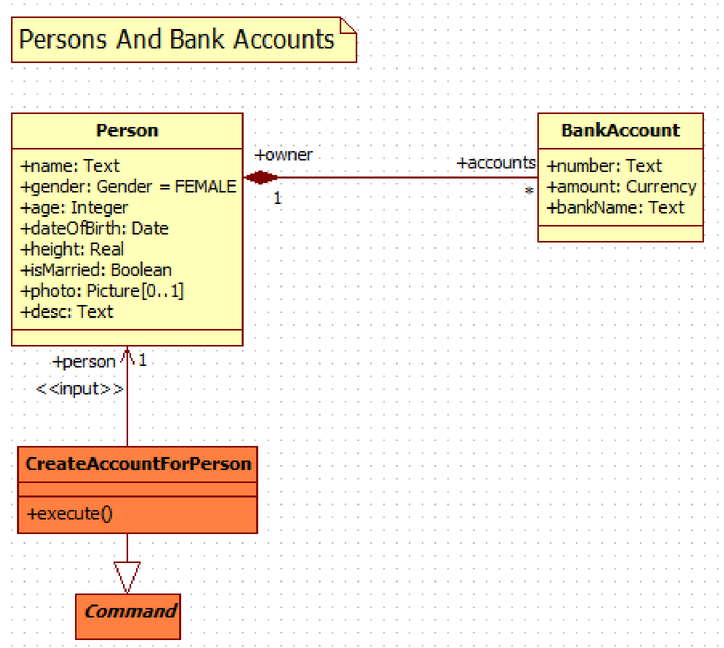

Слика 34. Екран који приказује податке о особама и њиховим рачунима у банци, са могућношћу креирања и брисања рачуна и измене детаља рачуна.

### Опис нових графичких елемената

Класа *GUINumberTextFeature* је још једна класа из пакета класа задужених за форматирање података. Она се користи као и остале сличне класе изведене из *GUISettingFeature*, с тим да је њена сврха да дефинише формат бројних вредности података на корисничком интерфејсу.

### Модел на језику UML

Слика 35 приказује једноставан модел података, односно класе *Person* (особа), *BankAccount* (банкарски рачун) и командну класу *CreateAccountForPerson* као и међусобне асоцијације. Командна класа за брисање рачуна није приказана због тога што већ постоји у библиотеци команди (која неће бити посебно обрађена јер излази из оквира ове дисертације).



Слика 35. Модел података на језику *UML*: класе особа, рачун и команда за креирање рачуна особе (енгл. *Person, BankAccount, CreateAccountForPerson,* респективно).

## Изворни код пословне логике

```
class CreateAccountForPerson {

public void execute() {
        Person p = person.val();
        if (p == null) {
        throw new CommandPreconditionsException("Please, select person first.");
        }

        BankAccount ba = new BankAccount();
        ba.number.set(Text.fromString("xxx-xxx-xxx"));
        ba.amount.set(Currency.fromBigDecimal(BigDecimal.valueOf(0.)));
        ba.bankName.set(Text.fromString("New Bank"));

        p.accounts.add(ba);
}
}
```

Листинг 17.

## Изворни код корисничког интерфејса

```
GUIApplicationComponent page = new GUIApplicationComponent(); // new web page
page.setName("PersonsAndBankAccounts");
SoloistServiceServlet.registerApplication(page);

GUIContext context = createContextAndStyles();
page.setContext(context);

GUIPanelComponent root = GUIPanelComponent.createFlow(page);

GUILabelComponent title = GUILabelComponent.create(root, "Persons and Bank Accounts");
title.setStyle("titleStyle");

GUIPanelComponent topPanel  = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

GUIPanelComponent table = GUIPanelComponent.createTable(topPanel);

// first row
GUILabelComponent.create(table, "Persons", 0, 0);
GUILabelComponent.create(table, "Person's Bank Accounts", 0, 1);
GUILabelComponent.create(table, "Account Details", 0, 2);

// second row
// this component fetches all persons, it is invisible component but still, it has to have a parent
GUIFindAllInstancesSAPComponent allPersons = GUIFindAllInstancesSAPComponent.create(root,
        Person.CLASSIFIER);
GUIInput personsList = GUIInput.createList(table);
personsList.setLayoutData(TableLayoutData.create(1, 0)); // shows all persons
GUIComponentBinding.create(allPersons.opValue(), personsList.ipCollection());
personsList.setSize("250px", "300px");
personsList.setStyle("listWidget");

GUIInput accountsList = GUIInput.createList(table, Person.PROPERTIES.accounts);
accountsList.setLayoutData(TableLayoutData.create(1, 1));
accountsList.setSize("250px", "300px");
accountsList.setStyle("listWidget");
GUIPanelComponent accountDetailsTable = GUIPanelComponent.createTable(table);
accountDetailsTable.setRowColumn(1, 2); // place the accountDetailsTable
accountDetailsTable.setCellAlignment(null, VerticalAlignment.TOP);

// third row
GUIButtonComponent createAccountButton = GUIButtonComponent.create(table, "Create Account for Person", new
        CreateAccountForPerson());
createAccountButton.setLayoutData(TableLayoutData.create(2, 0));
CmdDestroyObject destroyCmd = new CmdDestroyObject();
destroyCmd.setType(BankAccount.CLASSIFIER);
```

```
GUIButtonComponent destroyAccountButton = GUIButtonComponent.create(table, "Delete Account", destroyCmd);
destroyAccountButton.setLayoutData(TableLayoutData.create(2, 1));

// let's fill in the accoutDetailsTable
GUILabelComponent.create(accountDetailsTable, "Number:", 0, 0);
GUILabelComponent.create(accountDetailsTable, "Amount:", 1, 0);
GUILabelComponent.create(accountDetailsTable, "Bank name:", 2, 0);

GUIEdit numberEditor = GUIEdit.createField(accountDetailsTable, BankAccount.PROPERTIES.number, 0, 1);
GUIEdit amountEditor = GUIEdit.createField(accountDetailsTable, BankAccount.PROPERTIES.amount, 1, 1);
GUIEdit bankNameEditor = GUIEdit.createField(accountDetailsTable, BankAccount.PROPERTIES.bankName, 2, 1);

// let's bind the components
// account list will be populated for the selected person in the persons list
GUIComponentBinding.create(personsList.opValue(), accountsList.ipSlotValueElement());
// this specifies that account will be created for selected person
GUIComponentBinding.create(personsList.opValue(), createAccountButton,
        CreateAccountForPerson.PROPERTIES.person);
// this specifies that selected account will be destroyed
GUIComponentBinding.create(accountsList.opValue(), destroyAccountButton,
        CmdDestroyObject.PROPERTIES.input);
GUIComponentBinding.create(accountsList.opValue(), numberEditor.ipElement());
GUIComponentBinding.create(accountsList.opValue(), amountEditor.ipElement());
GUIComponentBinding.create(accountsList.opValue(), bankNameEditor.ipElement());

private GUIContext createContextAndStyles() {
        GUIContext context = new GUIContext();
        DefaultContextInit.getRoot().addContext(context);

        GUIObjectSetting bankAccount = GUIObjectSetting.create(context, BankAccount.CLASSIFIER);
        GUITextFeature.createName(bankAccount, BankAccount.PROPERTIES.number);
        GUITextFeature.createFixedSeparator(bankAccount, "/");
        GUINumberTextFeature.createDescription(bankAccount, BankAccount.PROPERTIES.amount);
        GUIPictureFeature.createSmallIcon(bankAccount, "images/icons/bankaccount.png");

        GUIObjectSetting person = GUIObjectSetting.create(context, Person.CLASSIFIER);
        GUITextFeature.createName(person, Person.PROPERTIES.name);
        GUITextFeature.createFixedSeparator(person, "Age:");
        GUITextFeature.createDescription(person, Person.PROPERTIES.age);
        GUIPictureFeature.createSmallIcon(person, "images/icons/person.png");
        return context;
}
```

Листинг 18.

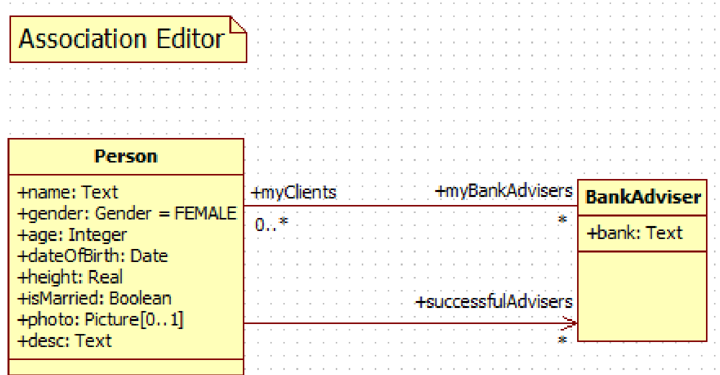
### Измена веза између ентитета

У овом примеру биће приказана употреба графичког елемента за измену вредности инстанце асоцијационог краја. На слици 36 приказан је екран из овог примера. На слици је један графички елемент – листа са пољима за чекирање. Овај елемент приказује све банкарске саветнике особе *Jane Roe* (са асоцијационог краја *Person::myBankAdvisers*) и нуди могућност обележавања оних који се сматрају успешним (односно везивање приказаних банкарских саветника по још једном асоцијационом крају: *Person::successfulAdvisers*).

Слика 36. Екран који приказује графички елемент за избор клијената запосленог у банци.

### *Модел на језику UML*

Дијаграм са слике 37 приказује модел на језику *UML* класа *Person* и *BankAdviser* као и зависности између њих.




Слика 37. Модел података на језику *UML:* класе особа и запослени (енгл. *Person, BankAdviser*, респективно).

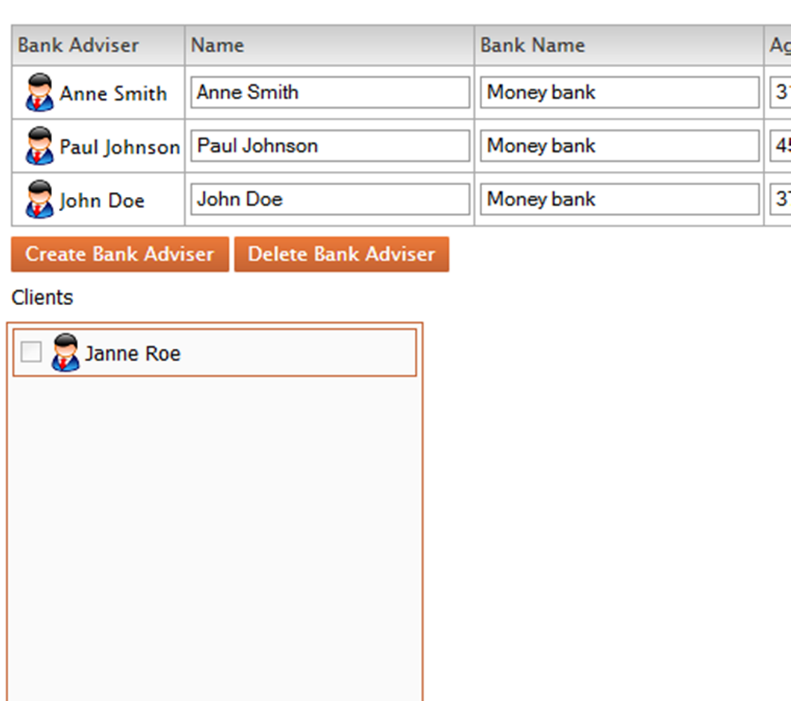
### *Изворни код корисничког интерфејса*

```
GUIApplicationComponent page = new GUIApplicationComponent();
page.setName("AssociationEditor");
SoloistServiceServlet.registerApplication(page);
page.setContext(createContextAndStyles());

GUIPanelComponent root = GUIPanelComponent.createFlow(page);

GUILabelComponent title = GUILabelComponent.create(root, "Association Editor");
title.setStyle("titleStyle");
```

```
GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");
GUIPanelComponent table = GUIPanelComponent.createTable(topPanel); // panel with table layout

GUIFindSingleInstanceSAPComponent findJanneRoe = GUIFindSingleInstanceSAPComponent.create(table,
        Person.FQ_TYPE_NAME, Person.FQPropertyNames.desc,
        (ClassifierInstanceDescriptor)ElementDescriptor.create(Text.fromString(ObjectSpaceInit.JANNE_ROE_ID
        ENTIFIER)));

GUIEdit editList = GUIEdit.createList(table, Person.PROPERTIES.successfulAdvisers,
        Person.PROPERTIES.myBankAdvisers);
editList.setLayoutData(TableLayoutData.create(0, 0));
GUIComponentBinding.create(findJanneRoe.opValue(), editList.ipElement());
GUIComponentBinding.create(findJanneRoe.opValue(), editList.ipSlotValueElement());

editList.setSize("250px", "200px");
editList.setStyle("listEditor");

private GUIContext createContextAndStyles() {
        GUIContext context = new GUIContext();
        DefaultContextInit.getRoot().addContext(context);

        GUIObjectSetting person = GUIObjectSetting.create(context, Person.CLASSIFIER);
        GUITextFeature.createName(person, Person.PROPERTIES.name);
        GUITextFeature.createFixedSeparator(person, ":");
        GUITextFeature.createDescription(person, Person.PROPERTIES.age);
        GUIPictureFeature.createSmallIcon(person, "images/icons/person.png");
        GUINavigatorFeature.createSubnodes(person, Person.FQPropertyNames.accounts);

        return context;
}
```
Листинг 19.

### Комбинација претходних примера

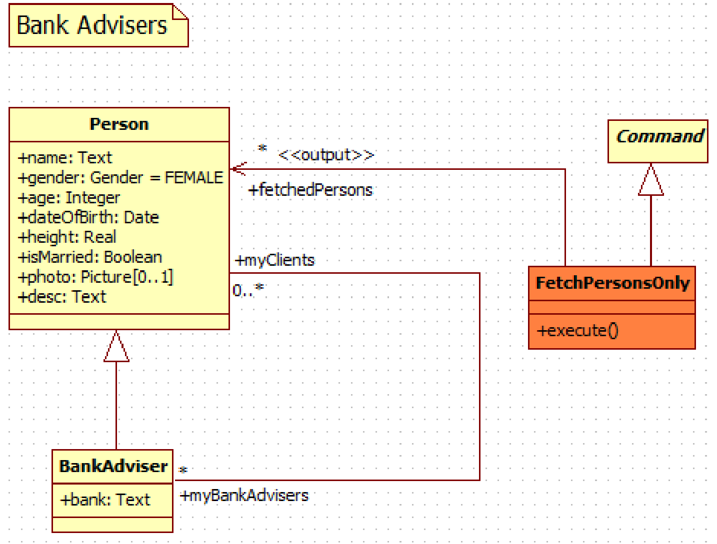
Пример чији је екран приказан на слици 38 приказује све банкарске саветнике у табели, омогућава промену њихових атрибута директно у контролама у табели, креирање и брисање банкарских саветника, као и приказ клијената у табели изабраног банкарског саветника.




Слика 38. Екран који приказује запослене у банци са могућношћу креирања и брисања истих и ажурирања листе клијената.

### *Модел на језику UML*

На дијаграму са слике 39 приказан је модел података који дефинише класе *Person* (особа), *BankAdviser* (банкарски саветник) и командну класу *FetchPersonsOnly*.




Слика 39. Модел података на језику *UML:* класе особа, запослени и команда за читање особа (енгл. *Person, BankAdviser*, *FetchPersonsOnly,* респективно).

### *Изворни код пословне логике*

```
class fetch PersonsOnly {

public void execute() {
        List<Person> allPersons = QueryUtils.getAllInstances(Person.CLASSIFIER);
        List<Person> justPersons = new ArrayList<Person>();
        for (Person person : allPersons) {
                if (person instanceof BankAdviser) {
                        continue;
                }
                justPersons.add(person);
        }
        fetchedPersons.set(justPersons);
}
}
```

Листинг 20.

### *Изворни код корисничког интерфејса*

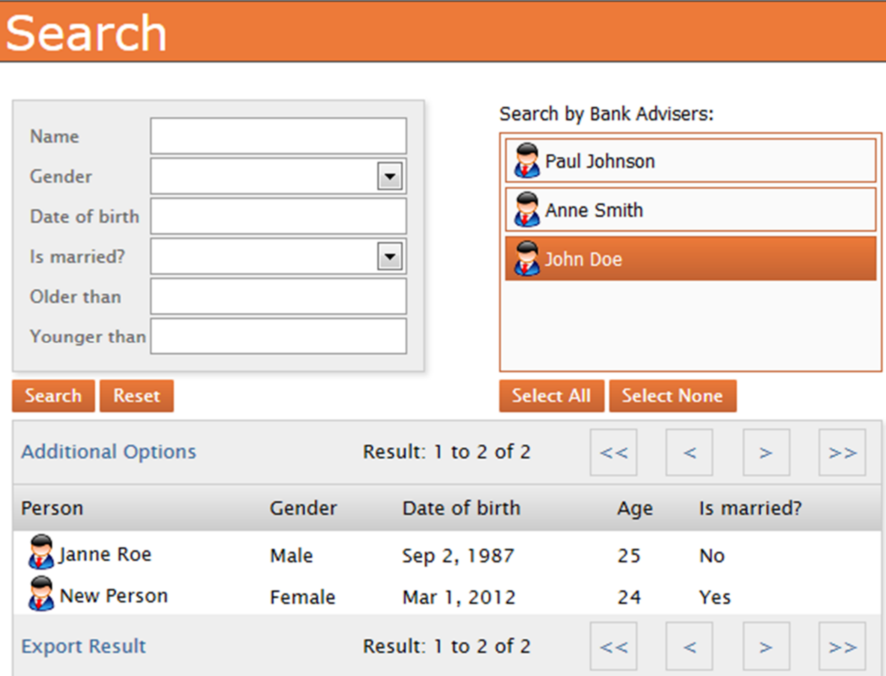
```
GUIApplicationComponent page = new GUIApplicationComponent();
page.setName("BankAdvisers");
SoloistServiceServlet.registerApplication(page);
GUIContext context = createContextAndStyles();
page.setContext(context);

GUIPanelComponent root = GUIPanelComponent.createFlow(page);

GUILabelComponent title = GUILabelComponent.create(root, "Bank Advisers");
title.setStyle("titleStyle");

GUIPanelComponent topPanel  = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

GUIFindAllInstancesSAPComponent allBankAdvisers = GUIFindAllInstancesSAPComponent.create(root,
        BankAdviser.CLASSIFIER);

// Column components
GUIEdit nameEditor = GUIEdit.createField(null, Person.PROPERTIES.name);
GUIEdit bankNameEditor = GUIEdit.createField(null, BankAdviser.PROPERTIES.bank);
GUIEdit genderEditor = GUIEdit.createField(null, Person.PROPERTIES.gender);
GUIEdit ageEditor = GUIEdit.createField(null, Person.PROPERTIES.age);
GUIEdit isMarriedEditor = GUIEdit.createField(null, Person.PROPERTIES.isMarried);

// Column header texts and oredering of column components
GUIInput editableTable = GUIInput.createTable(topPanel, "Bank Adviser", "Name", "Bank Name", "Age",
        "Gender", "Is married");
editableTable.setColumnComponents(nameEditor, bankNameEditor, ageEditor, genderEditor, isMarriedEditor);
editableTable.setMinRows(3); // min rows to show if there is no data in the table

GUIComponentBinding.create(allBankAdvisers.opValue(), editableTable.ipCollection());

GUIPanelComponent buttonsPanel  = GUIPanelComponent.createHorizontal(topPanel, VerticalAlignment.MIDDLE);
buttonsPanel.setStyle("margin3");

CmdCreateObjectOfClass createCmd = new CmdCreateObjectOfClass();
createCmd.setClass(BankAdviser.CLASSIFIER);
GUIButtonComponent createAdviserButton = GUIButtonComponent.create(buttonsPanel, "Create Bank Adviser",
        createCmd);

CmdDestroyObject destroyCmd = new CmdDestroyObject();
destroyCmd.setType(BankAdviser.CLASSIFIER);
GUIButtonComponent destroyAdviserButton = GUIButtonComponent.create(buttonsPanel, "Delete Bank Adviser",
        destroyCmd);

GUIComponentBinding.create(editableTable.opValue(), destroyAdviserButton,
        CmdDestroyObject.PROPERTIES.input);

GUIComponentBinding.create(createAdviserButton.opCommandExecuted(), allBankAdvisers.ipRefresh()); // this
        is to tell SAP to read advisers again
```

```
GUIComponentBinding.create(destroyAdviserButton.opCommandExecuted(), allBankAdvisers.ipRefresh()); // this
        is to tell SAP to read advisers again

GUILabelComponent clientsLabel = GUILabelComponent.create(topPanel, "Clients");
clientsLabel.setStyle("padding3");

GUIEdit clients = GUIEdit.createList(topPanel, BankAdviser.PROPERTIES.myClients);
clients.setSize("250px", "300px");
clients.setStyle("listEditor");

// invisible command: fetches all persons, but not bank advisers
GUICommandComponent allPersons = GUICommandComponent.create(root, new FetchPersonsOnly(),
        PerformImmediately.NOTHING);
GUIComponentBinding.create(allPersons, FetchPersonsOnly.PROPERTIES.fetchedPersons, clients.ipCollection());
GUIComponentBinding.create(editableTable.opValue(), clients.ipElement());
GUIComponentBinding.create(page.opStart(), allPersons.ipClick());

private GUIContext createContextAndStyles() {
        GUIContext context = new GUIContext();
        DefaultContextInit.getRoot().addContext(context);

        // This is how Person objects will look like in the GUI.
        GUIObjectSetting person = GUIObjectSetting.create(context, Person.CLASSIFIER);
        GUITextFeature.createName(person, Person.PROPERTIES.name);
        GUIPictureFeature.createSmallIcon(person, "images/icons/person.png");

        return context;
}
```

Листинг 21.

### Претрага особа

Овај пример приказује прототип странице за претрагу особа на основу критеријума задатих од стране корисника. У горњем делу странице задају се критеријуми претраге уносом текста у одговарајућа поља, али и избором понуђених вредности. Резултати претраге се приказују у табели у доњем делу екрана. Уколико се претраживање врши без унетих параметара, резултат претраге биће све особе. У табели са резултатима могу се изабрати додатне опције претраге кликом на линк *Additional Options* и то: које колоне треба да садржи табела са резултатима претраге, величину једне странице са резултатима претраге и да ли у резултатима претраге треба да пише број особа које задовољавају критеријуме (израчунавање броја редова у резултату може да има велики утицај на време претраживања у случају велике количине података у бази података). Кликом на заглавље неке колоне у табели са резултатима врши се сортирање резултата по тој колони. Резултати се могу извести у документ *Excel*. Коначно, кликом на један од резултата претраге, односно на особу, корисник се прослеђује на страницу са детаљима о тој особи. Екран који одговара овом примеру приказан је на слици 40.




Слика 40. Екран који приказује претрагу особа.

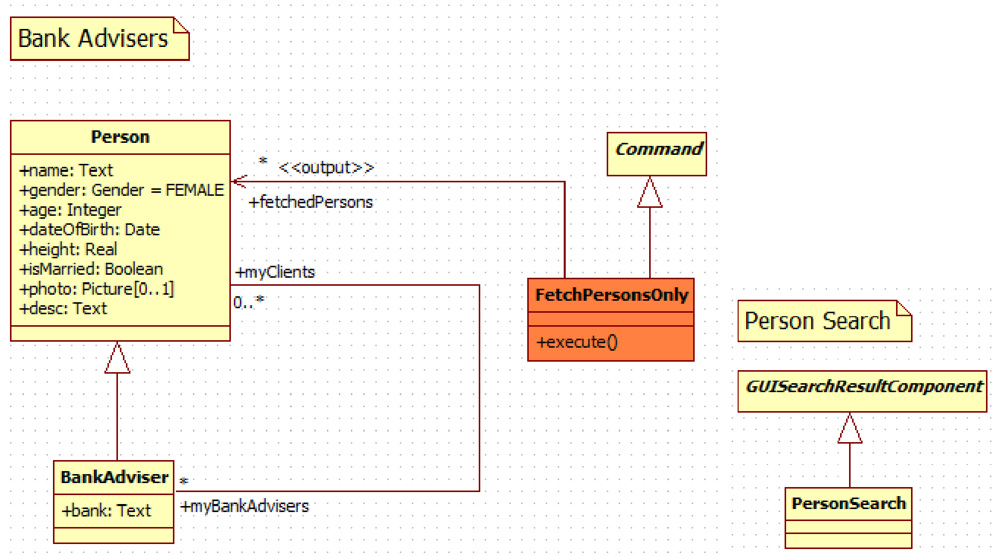
## Опис нових графичких елемената

Класа *GUISearchPanelComponent* представља посебну врсту панела који служе за прикупљање параметара претраге. Другим речима, панели овог типа садрже графичке елементе као што су поља, листе, стабла итд, помоћу којих корисник дефинише параметре за претраживање. Улога панела јесте да једном жицом између овог панела и циљног графичког елемента која приказује резултате претраживања обезбеди пренос параметара за претраживање. Без овог панела било би неопходно правити жице између сваког од графичких елемената и циљног елемента за претраживање.

Класа *GUISearchResultComponent* је апстрактна класа која представља све графичке елементе који приказују резултате претраживања објектног простора. Класа је направљена да буде наслеђена конкретним класама које служе за приказ резултата конкретног упита. Пријемом сигнала на пину који се може дохватити методом *ipSearch()* иницира се претрага објеката. Брисање резултата претраге врши се након пријема сигнала на улазном пину који се може дохватити методом *ipClearContents(),* а брисање параметара претраге након пријема сигнала на пину који се може дохватити методом *ipClearParameters().*

## Модел на језику UML

Модел приказан на слици 41 садржи класу *Person* (која се претражује), класу *BankAdviser* (чији објекти служе као параметри за претраживање), као и конкретну класу *PersonSearch* која представља специјализацију апстрактне класе *GUISearchResultComponent.*




Слика 41. Модел података на језику *UML:* класе особа, запослени, команда за читање особа и класа графичког елемента за претрагу (енгл. *Person, BankAdviser*, *FetchPersonsOnly, PersonSearch,* респективно).

## *Изворни код пословне логике*

```
public class PersonQueryBuilder extends QueryBuilder {

	private static final SimpleQueryDefinition prototypeQuery;

	@Parameter // @Parametar annotation means that QueryBuilder expects a parameter value from the
	element component named as the annotated string
	@Result("Name") // @Result annotation means that the column "Name" will be in the result table
	public static final String NAME = "NAME";
	@Result("Person") @OrderByAnother(NAME) // @OrderByAnother - objects of Person will be ordered by
	the name attribute
	public static final String PERSON = "PERSON";
	@Result("Age")
	public static final String AGE = "AGE";
	@Parameter @Result("Gender")
	public static final String GENDER = "GENDER";
	@Parameter @Result("Date of birth")
	public static final String DATE_OF_BIRTH = "DATE_OF_BIRTH";
	@Parameter @Result("Is married?")
	public static final String IS_MARRIED = "IS_MARRIED";
	@Parameter
	public static final String OLDER_THAN = "OLDER_THAN";
	@Parameter
	public static final String YOUNGER_THAN = "YOUNGER_THAN";
	@Parameter
	public static final String BANK_ADVISOR = "BANK_ADVISOR";

	public PersonQueryBuilder() {
		super(prototypeQuery);

		contributions.addAll(Arrays.asList(
			new ContributePrefixMatch(NAME),
			new ContributeEqual(IS_MARRIED),
			new ContributeEqual(GENDER),
			new ContributeEqual(DATE_OF_BIRTH),
			new ContributeGreaterOrEqual(OLDER_THAN),
			new ContributeLessOrEqual(YOUNGER_THAN),
			new ContributeIn(BANK_ADVISOR)
		));

		includeInResultOnce( // Columns in the result table, in this order
			PERSON,
			GENDER,
			DATE_OF_BIRTH,
```

```
                        AGE,
                        IS_MARRIED
                );
        }

        @Override
        public SimpleQueryDefinition buildCountQuery() {
                require(PERSON);
                return super.buildCountQuery();
        }

        @Override
        public SimpleQueryDefinition buildQuery() {
                require(PERSON);
                return super.buildQuery();
        }

        static {
                ObjectTerm root = new ObjectTerm(Person.FQ_TYPE_NAME).as(PERSON);
                new AttributeTerm(root, Person.PROPERTIES.name).as(NAME);
                new AttributeTerm(root, Person.PROPERTIES.gender).as(GENDER);
                new AttributeTerm(root, Person.PROPERTIES.age).as(AGE);
                new AttributeTerm(root, Person.PROPERTIES.dateOfBirth).as(DATE_OF_BIRTH);
                new AttributeTerm(root, Person.PROPERTIES.isMarried).as(IS_MARRIED);
                new AttributeTerm(root, Person.PROPERTIES.age).as(OLDER_THAN);
                new AttributeTerm(root, Person.PROPERTIES.age).as(YOUNGER_THAN);
                new AssocEndTerm(root, Person.PROPERTIES.myBankAdvisers).as(BANK_ADVISOR);

                prototypeQuery = new SimpleQueryDefinition(root).from(root);
        }
}
```

Листинг 22.

### *Изворни код корисничког интерфејса*

```
GUIApplicationComponent application = new GUIApplicationComponent();
application.setName("SearchSample");
SoloistServiceServlet.registerApplication(application);

GUIPanelComponent root = GUIPanelComponent.createFlow(application);

GUILabelComponent title = GUILabelComponent.create(root, "Search");
title.setStyle("titleStyle");

GUIPanelComponent topPanel  = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

GUIDeckComponent mainDeck = GUIDeckComponent.create(topPanel);

GUIPanelComponent wrapPanel  = GUIPanelComponent.createFlow(mainDeck);
GUISearchPanelComponent searchForm = GUISearchPanelComponent.create(wrapPanel, new FlowLayout());
// GUISearchPanelComponent binds element components implicitly between the search form and the query
        builder based on the components' names
searchForm.setStyle("form searchForm");

GUIPanelComponent table = GUIPanelComponent.createTable(searchForm);
table.setStyle("table");
table.setSize("100%", null);

int row = 0;
// name needs to be provided, in this case PersonQueryBuilder.NAME constant
GUIInput.createField(table, Text.CLASSIFIER, row++, 1).setName(PersonQueryBuilder.NAME);
GUIInput.createField(table, Repository.getRepository().getEnumeration(Gender.FQ_TYPE_NAME), row++,
        1).setName(PersonQueryBuilder.GENDER);
GUIInput.createField(table, Date.CLASSIFIER, row++, 1).setName(PersonQueryBuilder.DATE_OF_BIRTH);
GUIInput.createField(table, Boolean.CLASSIFIER, row++, 1).setName(PersonQueryBuilder.IS_MARRIED);
GUIInput.createField(table, Integer.CLASSIFIER, row++, 1).setName(PersonQueryBuilder.OLDER_THAN);
GUIInput.createField(table, Integer.CLASSIFIER, row++, 1).setName(PersonQueryBuilder.YOUNGER_THAN);

row = 0;
GUILabelComponent.create(table, "Name", row++, 0).setStyle("formLabel");
GUILabelComponent.create(table, "Gender", row++, 0).setStyle("formLabel");
```

```
GUILabelComponent.create(table, "Date of birth", row++, 0).setStyle("formLabel");
GUILabelComponent.create(table, "Is married?", row++, 0).setStyle("formLabel");
GUILabelComponent.create(table, "Older than", row++, 0).setStyle("formLabel");
GUILabelComponent.create(table, "Younger than", row++, 0).setStyle("formLabel");

GUILabelComponent.create(wrapPanel, "Search by Bank Advisers:");
GUISearchPanelComponent searchList = GUISearchPanelComponent.create(wrapPanel, new FlowLayout());
GUIFindAllInstancesSAPComponent advisersSource = GUIFindAllInstancesSAPComponent.create(searchList,
        BankAdviser.CLASSIFIER);
GUIInput bankAdvisors = GUIInput.createList(searchList);
GUIComponentBinding.create(advisersSource.opValue(), bankAdvisors.ipCollection());
bankAdvisors.setName(PersonQueryBuilder.BANK_ADVISOR);
bankAdvisors.setUpperBound(-1);
bankAdvisors.setSize("250px", "154px");
bankAdvisors.setStyle("listWidget");

GUIButtonComponent btnAll = GUIButtonComponent.create(wrapPanel, "Select All");
GUIButtonComponent btnNone = GUIButtonComponent.create(wrapPanel, "Select None");
GUIComponentBinding.create(btnAll.opClick(), bankAdvisors.ipSelectAll());
GUIComponentBinding.create(btnNone.opClick(), bankAdvisors.ipUnselectAll());

GUIPanelComponent buttonPanel = GUIPanelComponent.createFlow(wrapPanel);
buttonPanel.setStyle("searchButtons");

GUIButtonComponent searchButton = GUIButtonComponent.create(buttonPanel, "Search");
searchButton.setStyle("submitButton");

GUIButtonComponent resetButton = GUIButtonComponent.create(buttonPanel, "Reset");
resetButton.setStyle("submitButton");

GUIPanelComponent searchTable = GUIPanelComponent.createFlow(wrapPanel);
searchTable.setStyle("searchPanel");

PersonSearch searchResult = new PersonSearch();
searchTable.add(searchResult);
searchResult.setDoInitialSearch(false);
searchForm.addDynamicBindingDest(searchResult); // binds all fields in search form with corresponding query
        builder
searchList.addDynamicBindingDest(searchResult);

GUIComponentBinding.create(searchButton.opClick(), searchResult.ipSearch());
GUIComponentBinding.create(resetButton.opClick(), searchForm.ipReset());
GUIComponentBinding.create(resetButton.opClick(), searchResult.ipClearContents());

GUIComponentBinding.create(searchButton.opClick(), searchResult.ipSearch());
GUIComponentBinding.create(resetButton.opClick(), searchForm.ipReset());
GUIComponentBinding.create(resetButton.opClick(), searchResult.ipClearContents());

GUIComponentBinding.create(wrapPanel.opRelay2(), searchResult.ipSearch());

GUIPanelComponent personDetails = GUIPanelComponent.createTable(mainDeck);
personDetails.setStyle("searchDetails");

GUIPanelComponent detailsTable = GUIPanelComponent.createTable(personDetails);
application.setContext(createContextAndStyles(detailsTable));

createDetails(detailsTable, wrapPanel);

private void createDetails(GUIPanelComponent personDetails, GUIPanelComponent backTo) {
        int row = 0;
        GUILabelComponent.create(personDetails, "Name: ", row, 0);
        GUIEdit nameEditor = GUIEdit.createField(personDetails, Person.PROPERTIES.name, row++, 1);

        GUILabelComponent.create(personDetails, "Gender: ", row, 0);
        GUIEdit genderEditor = GUIEdit.createField(personDetails, Person.PROPERTIES.gender, row++, 1);

        GUILabelComponent.create(personDetails, "Age: ", row, 0);
        GUIEdit ageEditor = GUIEdit.createField(personDetails, Person.PROPERTIES.age, row++, 1);

        GUILabelComponent.create(personDetails, "Date of birth: ", row, 0);
        GUIEdit dateOfBirthEditor = GUIEdit.createField(personDetails, Person.PROPERTIES.dateOfBirth,
        row++, 1);
```

```
        GUILabelComponent.create(personDetails, "Height [m]: ", row, 0);
        GUIEdit heightEditor = GUIEdit.createField(personDetails, Person.PROPERTIES.height, row++, 1);

        GUILabelComponent.create(personDetails, "Is married: ", row, 0);
        GUIEdit isMarriedEditor = GUIEdit.createField(personDetails, Person.PROPERTIES.isMarried, row++,
        1);

        GUIComponentBinding.create(personDetails.opRelay1(), nameEditor.ipElement());
        GUIComponentBinding.create(personDetails.opRelay1(), genderEditor.ipElement());
        GUIComponentBinding.create(personDetails.opRelay1(), ageEditor.ipElement());
        GUIComponentBinding.create(personDetails.opRelay1(), dateOfBirthEditor.ipElement());
        GUIComponentBinding.create(personDetails.opRelay1(), heightEditor.ipElement());
        GUIComponentBinding.create(personDetails.opRelay1(), isMarriedEditor.ipElement());

        GUIButtonComponent backButton = GUIButtonComponent.create(personDetails, "Back");
        backButton.setLayoutData(TableLayoutData.create(row++, 0));
        GUIComponentBinding.create(backButton.opClick(), backTo.ipShow());
}

private GUIContext createContextAndStyles(GUIPanelComponent personDetails) {
        GUIContext context = new GUIContext();
        DefaultContextInit.getRoot().addContext(context);

        // how will Person objects look like in the GUI?
        GUIObjectSetting person = GUIObjectSetting.create(context, Person.CLASSIFIER);
        GUITextFeature.createName(person, Person.PROPERTIES.name);
        GUIPictureFeature.createSmallIcon(person, "images/icons/person.png");

        GUIBindingsFeature bf = GUIBindingsFeature.create(person);
        // doubleClick signal carries a Person object
        GUIComponentBinding.create(bf.opDoubleClick(), personDetails.ipRelay1());
        // double-clicking on a person in the result table will show the person's details
        GUIComponentBinding.create(bf.opDoubleClick(), personDetails.ipShow());

        return context;
}
```

Листинг 23.

### Полиморфни панел

У овом примеру биће приказано како се део корисничког интерфејса може прилагођавати специфичностима класе доменских објеката чије детаље треба да прикаже. Екран из овог примера приказан је на слици 42. У зависности од тога да ли се приказују детаљи објекта класе *Person* или *BankAdviser*, фрагмент корисничког интерфејса имаће различит скуп графичких елемената.

Polymorphic

Choose person:

Name: Anne Smith

Gender: Female

Age: 31

Date of birth: 6/10/81

Height [m]: 1.74

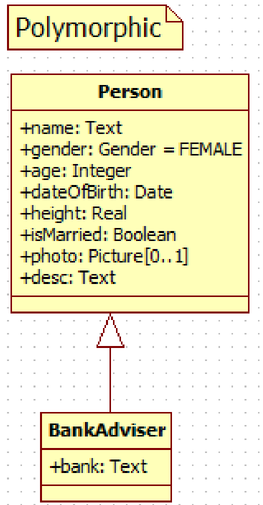
Is married:

Bank Name: Money bank

Слика 42. Екран који приказује измену података о изабраној особи. Количина и тип података зависи од типа особе.

## *Модел на језику UML*

Дијаграм на слици 43 приказује класе *Person* и *BankAdviser* које се користе у овом примеру.




Слика 43. Модел података на језику *UML:* класе особа и запослени (енгл. *Person, BankAdviser*, респективно).

## *Изворни код корисничког интерфејса*

```
GUIApplicationComponent page = new GUIApplicationComponent();
page.setName("Polymorphic");
SoloistServiceServlet.registerApplication(page);
page.setContext(DefaultContextInit.getRoot());
```

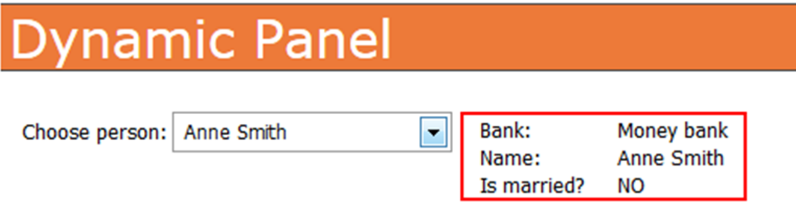
```
GUIPanelComponent root = GUIPanelComponent.createFlow(page);

GUILabelComponent title = GUILabelComponent.create(root, "Polymorphic");
title.setStyle("titleStyle");

GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

GUIPanelComponent wrap = GUIPanelComponent.createFlow(topPanel);

GUILabelComponent.create(wrap, "Choose person:");

GUIFindAllInstancesSAPComponent allPersons = GUIFindAllInstancesSAPComponent.create(wrap,
        Person.CLASSIFIER);
GUIInput comboBox = GUIInput.createCombo(wrap);
GUIComponentBinding.create(allPersons.opValue(), comboBox.ipCollection());

GUIDeckComponent deck = GUIDeckComponent.create(wrap);

GUIPanelComponent personDetailsTable = GUIPanelComponent.createTable(deck);
createPersonsDetails(comboBox, personDetailsTable);

GUIPanelComponent bankAdviserDetailsTable = GUIPanelComponent.createTable(deck);
int row = createPersonsDetails(comboBox, bankAdviserDetailsTable);
GUILabelComponent.create(bankAdviserDetailsTable, "Bank Name: ", row, 0);
GUIEdit nameEditor = GUIEdit.createField(bankAdviserDetailsTable, BankAdviser.PROPERTIES.bank, row++, 1);
GUIComponentBinding.create(comboBox.opValue(), nameEditor.ipElement());

// Is the selected object an instance of the BankAdviser class?
GUIConformsToFilter isItBankAdviser = GUIConformsToFilter.create(deck, BankAdviser.CLASSIFIER);
GUIComponentBinding.create(comboBox.opValue(), isItBankAdviser.ipInput());
GUIComponentBinding.create(isItBankAdviser.opYes(), bankAdviserDetailsTable.ipShow());
GUIComponentBinding.create(isItBankAdviser.opNo(), personDetailsTable.ipShow());

private int createPersonsDetails(GUIElementComponent comboBox, GUIPanelComponent detailsTable) {
        int row = 0;
        GUILabelComponent.create(detailsTable, "Name: ", row, 0);
        GUIEdit nameEditor = GUIEdit.createField(detailsTable, Person.PROPERTIES.name, row++, 1);

        GUILabelComponent.create(detailsTable, "Gender: ", row, 0);
        GUIEdit genderEditor = GUIEdit.createField(detailsTable, Person.PROPERTIES.gender, row++, 1);

        GUILabelComponent.create(detailsTable, "Age: ", row, 0);
        GUIEdit ageEditor = GUIEdit.createField(detailsTable, Person.PROPERTIES.age, row++, 1);

        GUILabelComponent.create(detailsTable, "Date of birth: ", row, 0);
        GUIEdit dateOfBirthEditor = GUIEdit.createField(detailsTable, Person.PROPERTIES.dateOfBirth, row++,
        1);

        GUILabelComponent.create(detailsTable, "Height [m]: ", row, 0);
        GUIEdit heightEditor = GUIEdit.createField(detailsTable, Person.PROPERTIES.height, row++, 1);

        GUILabelComponent.create(detailsTable, "Is married: ", row, 0);
        GUIEdit isMarriedEditor = GUIEdit.createField(detailsTable, Person.PROPERTIES.isMarried, row++, 1);

        GUIComponentBinding.create(comboBox.opValue(), nameEditor.ipElement());
        GUIComponentBinding.create(comboBox.opValue(), genderEditor.ipElement());
        GUIComponentBinding.create(comboBox.opValue(), ageEditor.ipElement());
        GUIComponentBinding.create(comboBox.opValue(), dateOfBirthEditor.ipElement());
        GUIComponentBinding.create(comboBox.opValue(), heightEditor.ipElement());
        GUIComponentBinding.create(comboBox.opValue(), isMarriedEditor.ipElement());
        return row;
}
```

Листинг 24.

### Динамички панел

У овом примеру описана је класа графичких елемената која се врло често користи кад се жели да кориснички интерфејс у потпуности зависи од података, односно кад је његова динамичност изражена. На слици 44 приказан је један врло једноставан пример који

подсећа на пример из претходног поглавља, али се реализује не један потпуно другачији начин.



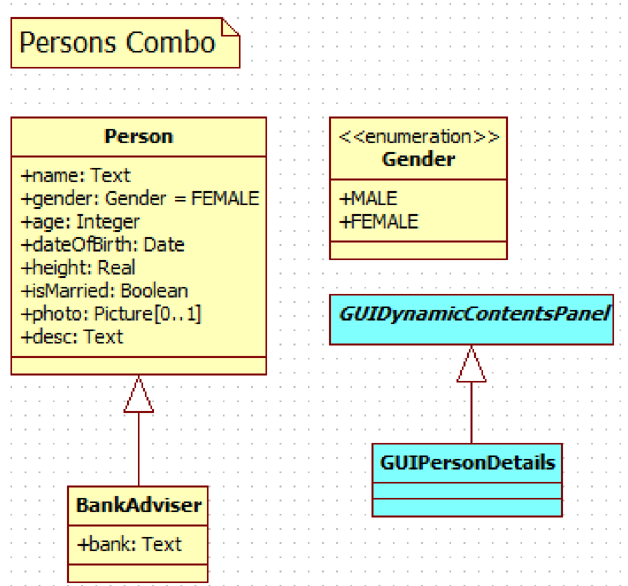

Слика 44. Екран који приказује приказ података о изабраној особи. Количина и тип података зависи од типа особе. Решење добијено уз помоћ динамичког панела.

### *Опис нових графичких елемената*

Класа *GUIDynamicPanelComponent* је апстрактна генерализација свих динамичких панела, односно панела чија подстабла графичких елемената директно зависе од података и израчунавају се више пута у току извршавања апликације. Ова класа је направљена да буде наслеђена од стране конкретних класа динамичких панела. Потребно је само направити конкретну поткласу и имплементирати методу *getDynamicContentns(IElement)* у којој се програмски израчунава подстабло овог графичког елемента и то у зависности од стања објектног простора, односно података. Параметар методе је вредност коју овај графички елемент добија преко улазног пина који се може дохватити позивом методе *ipElement()*. Подстабло се израчунава сваки пут када се појави нова вредност на овом улазном пину.

### *Модел на језику UML*

На слици 45 приказан је дијаграм модела на језику *UML* на ком су приказане две доменске класе као и једна конкретна класа динамичког панела *GUIPersonDetails.*




Слика 45. Модел података на језику *UML:* класе особа, запослени и динамички панел за приказ детаља особе (енгл. *Person, BankAdviser, GUIPersonDetails*, респективно).

## *Изворни код корисничког интерфејса*

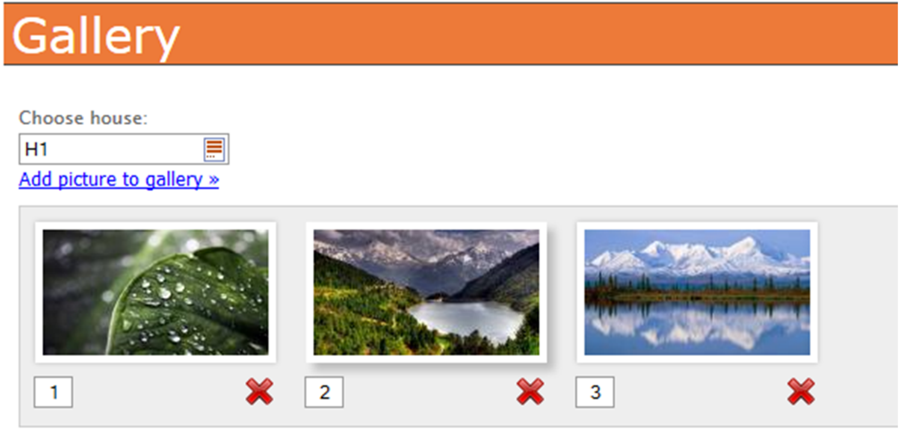
```
GUIApplicationComponent page = new GUIApplicationComponent();
page.setName("PersonCombo");
SoloistServiceServlet.registerApplication(page);
page.setContext(DefaultContextInit.getRoot());

GUIPanelComponent root = GUIPanelComponent.createFlow(page);

GUILabelComponent title = GUILabelComponent.create(root, "Dynamic Panel");
title.setStyle("titleStyle");

GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

GUIPanelComponent horizontal = GUIPanelComponent.createHorizontal(topPanel, VerticalAlignment.TOP);

GUILabelComponent.create(horizontal, "Choose person:").setStyle("margin3");
GUIFindAllInstancesSAPComponent allPersons = GUIFindAllInstancesSAPComponent.create(horizontal,
        Person.CLASSIFIER);
GUIInput comboBox = GUIInput.createCombo(horizontal);
GUIComponentBinding.create(allPersons.opValue(), comboBox.ipCollection());

GUIPersonDetails pd = new GUIPersonDetails();
horizontal.add(pd);

GUIComponentBinding.create(comboBox.opValue(), pd.ipElement());

class GUIPersonDetails extends GUIDynamicPanelComponent {

@Override
protected GUIContainerComponent getDynamicContents(IElement el) {
        GUIPanelComponent rootPanel = GUIPanelComponent.createFlow(null);
        rootPanel.setStyle("person_details");

        Person p = (Person) el;
        if (p == null) {
                GUILabelComponent.create(rootPanel, "No Persons selected.");
                return rootPanel;
        }

        GUIPanelComponent table = GUIPanelComponent.createTable(rootPanel);

        int row = 0;
```

```
        if (p instanceof BankAdviser) {
                GUILabelComponent.create(table, "Bank: ", row, 0);
                GUILabelComponent.create(table, Text.stringValue(((BankAdviser) p).bank), row++, 1);
        }

        GUILabelComponent.create(table, "Name: ", row, 0);
        GUILabelComponent.create(table, Text.stringValue(p.name), row++, 1);
        GUILabelComponent.create(table, "Is married? ", row, 0);
        GUILabelComponent.create(table, p.isMarried.val().toBoolean() ? "YES" : "NO", row++, 1);

        if (Gender.MALE.equals(p.gender.val())) {
                GUILabelComponent.create(table, "Age: ", row, 0);
                GUILabelComponent.create(table, p.age.val().toString(), row++, 1);
                table.setStyle("male");
        } else {
                table.setStyle("female");
        }

        return rootPanel;
}
}
```

Листинг 25.

### Галерија фотографија

На слици 46 приказан је један типичан екран – галерија фотографија која је имплеметирана у овом примеру. У примеру се најпре бира кућа, а након тога кориснички интерфејс приказује све слике из галерије и даје могућности да се додају нове слике а старе бришу.




Слика 46. Екран који приказује галерију фотографија везаних за изабрану кућу.

#### *Опис нових графичких елемената*

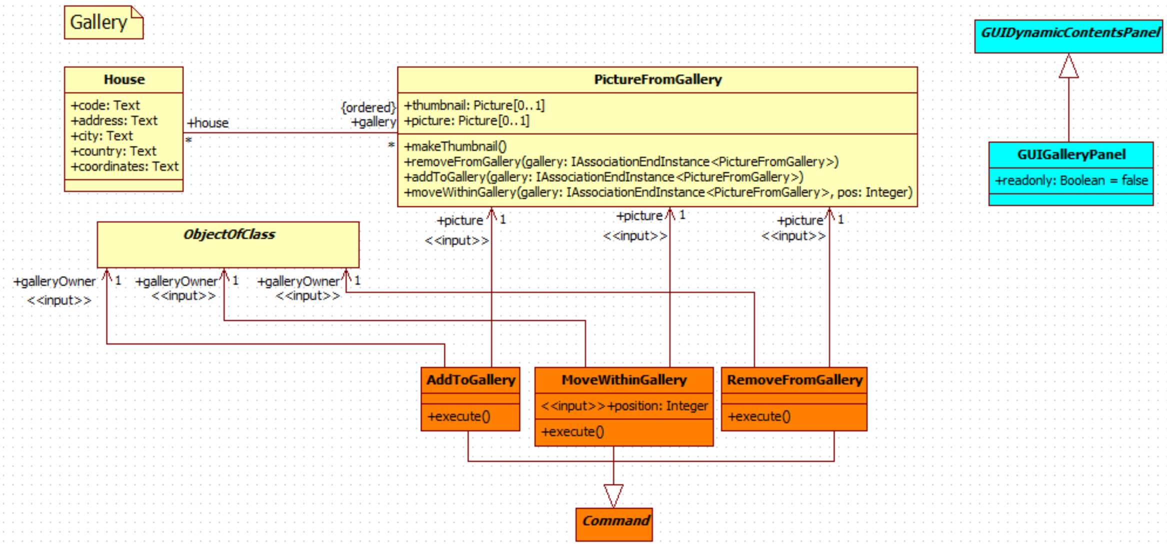
Класа *GUIRelayComponent* представља невидљиве елементе са врло једноставном логиком понашања. Вредности које приме преко улазног пина који се може дохватити позивом методе *ipRelay()* одмах се прослеђују на излазни пин (метода *opRelay()*). Овај елемент корисничког интерфејса може да буде врло користан као „сабирач“ сигнала и помаже приликом енкапсулације фрагмената корисничког интерфејса.

Класа *GUIDisclosurePanel* представља контејнерске графичке елементе који своје подстабло графичких елемената могу да прикажу и сакрију. Контрола приказа је препуштена кориснику апликације који има на располагању заглавље овог панела на које може да кликне и да наизменично приказује, односно сакрива садржај панела. Такође, контрола приступа може се остварити и програмски, преко улазног пина (метода *ipOpen()*)

на ком се очекује вредност типа *boolean* која дефинише да ли ће се садржај приказати (*true*) или сакрити (*false*).

## Модел на језику UML

На дијаграму са слике 47 приказан је модел на језику *UML* који приказује доменске класе, команде и један динамички панел.




Слика 47. Модел података на језику *UML:* класе кућа и фотографија, команде за додавање фотографије у галерију, премештање и уклањање из галерије и динамички панел за приказ галерије (енгл. *House, PictureFromGallery, AddToGallery, MoveWithinGallery, RemoveFromGallery, GUIGalleryPanel,* респективно).

## Изворни код пословне логике

```
class AddToGallery extends Command {

public void execute() {
        ObjectOfClass owner = galleryOwner.val();
        IAssociationEndInstance<PictureFromGallery> gallery = ((House) owner).gallery;
        picture.val().addToGallery(gallery);
}

@Override
protected void doCheckPreconditions() throws CommandPreconditionsException {
        super.doCheckPreconditions();
        if (galleryOwner.val() == null) {
                throw new CommandPreconditionsException("Please select the house.");
        }
}
}

class MoveWithinGallery extends Command {

public void execute() {
        ObjectOfClass owner = galleryOwner.val();
        IAssociationEndInstance<PictureFromGallery> gallery = ((House) owner).gallery;
        PictureFromGallery pfg = picture.val();
        pfg.moveWithinGallery(gallery, position.val());
}
}

class RemoveFromGallery extends Command {

public void execute() {
        ObjectOfClass owner = galleryOwner.val();
        IAssociationEndInstance<PictureFromGallery> gallery = ((House) owner).gallery;
        picture.val().removeFromGallery(gallery);
}
```

```
}

class PictureFromGallery {

public void makeThumbnail() {
	if (picture.val() != null) {
		try {
thumbnail.set(Picture.fromBufferedImage(Thumbnails.of(picture.val().toBufferedImage()).size(150,
	150).outputFormat("jpg").asBufferedImage()));
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}

public void removeFromGallery(IAssociationEndInstance<PictureFromGallery> gallery) {
	gallery.remove(this);
	if (this.house.read().isEmpty())
	this.destroy();
}


public void addToGallery(IAssociationEndInstance<PictureFromGallery> gallery) {
	this.makeThumbnail();
	this.submitted.set(Boolean.TRUE);
	gallery.addLast(this);
}

public void moveWithinGallery(IAssociationEndInstance<PictureFromGallery> gallery, Integer pos) {
	int i = gallery.read().indexOf(this);
	int position = pos.toInt() - 1;
	if (position != -1) {
		if (position >= gallery.size())
			position = (int) (gallery.size() - 1);
		else if (position < 0)
			position = 0;
		gallery.reorder(i, position);
	}
}
}
```

Листинг 26.

### *Изворни код корисничког интерфејса*

```
GUIApplicationComponent application = new GUIApplicationComponent();
application.setName("GallerySample");
SoloistServiceServlet.registerApplication(application);
application.setContext(createContextAndStyles());

GUIPanelComponent root = GUIPanelComponent.createFlow(application);

GUILabelComponent title = GUILabelComponent.create(root, "Gallery");
title.setStyle("titleStyle");

GUIPanelComponent topPanel = GUIPanelComponent.createFlow(root);
topPanel.setStyle("topPanel");

GUILabelComponent.create(topPanel, "Choose house:").setStyle("formLabel");

GUIFindAllInstancesSAPComponent allHouses = GUIFindAllInstancesSAPComponent.create(topPanel,
	House.CLASSIFIER);
GUIInput suggestBox = GUIInput.createSuggest(topPanel);
suggestBox.setLowerBound(1);
GUIComponentBinding.create(allHouses.opValue(), suggestBox.ipCollection());

GalleryFragment gf = new GalleryFragment(topPanel);
GUIComponentBinding.create(suggestBox.opValue(), gf.ipOwner());

private GUIContext createContextAndStyles() {
	GUIContext context = new GUIContext();
	DefaultContextInit.getRoot().addContext(context);
	GUIObjectSetting person = GUIObjectSetting.create(context, House.CLASSIFIER);
```

```
        GUITextFeature.createName(person, House.PROPERTIES.code);
        return context;
}

public class GalleryFragment {

        private ISlot<?> owner;

        public ISlot<?> ipOwner(){
                return owner;
        }

        public static MoveWithinGallery cmdMoveWithinGallery = new MoveWithinGallery();
        public static RemoveFromGallery cmdRemoveFromGallery = new RemoveFromGallery();

        public GalleryFragment(GUIPanelComponent rootPanel) {
                GUIRelayComponent ownerRelay = GUIRelayComponent.create(rootPanel);
                owner = ownerRelay.ipRelay();

                GUIDisclosurePanel newPictureDisclosure = GUIDisclosurePanel.create(rootPanel, "Add picture
        to gallery");

                GUIPanelComponent subDisclosure = GUIPanelComponent.createFlow(newPictureDisclosure);
                subDisclosure.setStyle("disclosureForm");
                GUIPanelComponent table = GUIPanelComponent.createTable(subDisclosure);
                table.setStyle("table");

                CmdCreateObjectOfClass cmd = new CmdCreateObjectOfClass();
                cmd.setClass(PictureFromGallery.CLASSIFIER);
                GUICommandComponent cmdNewBlank = GUICommandComponent.create(rootPanel, cmd,
        PerformImmediately.NOTHING);
                GUIBooleanFilter gbf = GUIBooleanFilter.create(rootPanel);
                GUIComponentBinding.create(newPictureDisclosure.opOpen(), gbf.ipInput());
                GUIComponentBinding.create(gbf.opYes(), cmdNewBlank.ipClick());

                int row = 0;
                GUILabelComponent.create(table, "Title", row++, 0).setStyle("formLabel");
                GUILabelComponent.create(table, "Description", row++, 0).setStyle("formLabel");
                GUILabelComponent.create(table, "Picture", row++, 0).setStyle("formLabel");

                row = 0;
                GUIEdit title = GUIEdit.createField(table, Document.PROPERTIES.title, row++, 1);
                GUIEdit descrip = GUIEdit.createField(table, Document.PROPERTIES.description, row++, 1);
                descrip.setMaxLength(2000);
                descrip.setMultiline(true);
                descrip.setSize("265px", "192px");
                GUIPanelComponent picUpload = GUIEdit.createPicture(table,
        PictureFromGallery.PROPERTIES.picture, true);
                picUpload.setRowColumn(row++, 1);

                GUIPanelComponent picture = GUIEdit.createPicture(table,
        PictureFromGallery.PROPERTIES.picture, false);
                picture.setRowColumn(0, 2, 4, 1);
                picture.setCellAlignment(null, VerticalAlignment.TOP);

                GUIComponentBinding.create(cmdNewBlank, CmdCreateObjectOfClass.PROPERTIES.output,
        title.ipElement());
                GUIComponentBinding.create(cmdNewBlank, CmdCreateObjectOfClass.PROPERTIES.output,
        descrip.ipElement());
                GUIComponentBinding.create(cmdNewBlank, CmdCreateObjectOfClass.PROPERTIES.output,
        picUpload.ipRelay1());
                GUIComponentBinding.create(cmdNewBlank, CmdCreateObjectOfClass.PROPERTIES.output,
        picture.ipRelay1());

                GUIButtonComponent cmdAdd = GUIButtonComponent.create(table, "Add picture", new
        AddToGallery());
                cmdAdd.setLayoutData(TableLayoutData.create(row++, 0));
                GUIComponentBinding.create(ownerRelay.opRelay(), cmdAdd,
        AddToGallery.PROPERTIES.galleryOwner);
                GUIComponentBinding.create(cmdNewBlank, CmdCreateObjectOfClass.PROPERTIES.output, cmdAdd,
        AddToGallery.PROPERTIES.picture);

                GUIGalleryPanel galleryPanel = new GUIGalleryPanel();
```

```
                rootPanel.add(galleryPanel);
                GUIComponentBinding.create(ownerRelay.opRelay(), galleryPanel.ipElement());
                GUIComponentBinding.create(cmdAdd.opCommandExecuted(), galleryPanel.ipRefreshContents());
                GUIBufferComponent gbc = GUIBufferComponent.create(rootPanel, false, Boolean.FALSE);
                GUIComponentBinding.create(gbc.opOutput(), newPictureDisclosure.ipOpen());
                GUIComponentBinding.create(cmdAdd.opCommandExecuted(), gbc.ipSend());
        }
}

extension.js
function htmlComponentChanged(id) {
        if (id == "fancyGallery") {
                setTimeout(function() {
                        $("a.fancy_group").fancybox({
                                'titlePosition' : 'over'
                        });
                }, timeout);
        }
}
```

Листинг 27.

# *Имплементација и помоћни алати*

У оквиру ове докторске дисертације имплементирано је и извршно окружење за описану технологију за развој корисничких интерфејса које чини део технологије *SOLoist* [47]. То је технологија за брзи развој информационих система вођен извршивим моделима на језику *UML*. Имплементација подржава описане механизме инстанцирања компонената из описане библиотеке, комуникације између компонената преко пинова и жица, приступ објектном простору, као и механизам аутоматске нотификације корисничког интерфејса о променама у објектном простору.

Имплементација је веб базирана. Слој веб клијента садржи комплетан слој корисничког интерфејса описаног у претходним поглављима, док је слој објектног простора постављен на веб серверу. Клијентски слој користи технологију *Google Web Toolkit – GWT,* као имплементациону платформу. Прецизније речено, уграђене библиотечке компоненте омотавају (енгл. *wrap up*) елементе корисничког интерфејса технологије *GWT* и додају им описану семантику капсула. У складу са технологијом *GWT,* програмски језик који је коришћен за имплементацију је језик *Java.* Имплементациони код који је написан на језику *Java* преводи се на језик *JavaScript* уз помоћ посебног компајлера који се испоручује уз технологију *GWT*. Добијени код на језику *JavaScript* погодан је извршавање у свеприсутним веб претраживачима (енгл. *web browser*), односно веб клијентима.

Серверски слој одговоран је да обезбеди семантику профила *OOIS UML* над објектним простором. Он садржи сопствено извршно окружење за објектно-релационо мапирање, како би се објектни простор са описаном семантиком адаптирао на релациону базу података која се користи као медијум за перзистенцију података. Серверски слој такође обезбеђује приступне тачке клијентском слоју имплементиране на језику *Java* као класе типа *Servlet*. Клијентски слој приступа свим сервисима преко позива типа *AJAX*. Резултати захтева, као и нотификационе поруке, враћају се клијентском слоју у виду одговора на протоколу *HTTP*.

Показало се да филозофија технологије *GWT,* као имплементационе платформе нижег нивоа, одлично одговара описаној парадигми за развој корисничких интерфејса. Ова парадигма доноси нове апстракције и комплементарна је споменутој имплементационој технологији. Као резултат тог доброг слагања технологија, апликације дизајниране уз помоћ ове технологије су посебно атрактивне. Наиме, када се апликација покрене (уносом

одговарајуће веб адресе у веб претраживач), цела апликација – све компоненте (осим оних посебно означених да имају одложено учитавање или динамичку интерну структуру) учитавају се (енгл. *download*) на веб клијента. Ово обично траје једну до неколико секунди. Од тог тренутка, цео кориснички интерфејс апликације – сви панели, форме, контроле и др. остају на клијенту. Неки делови корисничког интерфејса се неће одмах видети; они остају сакривени иза тренутно приказаног (иницијалног) панела. Међутим, они ће се, према потреби, практично тренутно појавити када добију сигнал на улазном пину *show*. Ово оставља утисак тренутног одзива на акције корисника као код десктоп апликација што је, као што је већ објашњено, последица чињенице да се виртуелна машина (написана на језику *JavaScript*) која интерпретира модел компонената, као и сам модел компонената већ налазе на клијенту, односно да су учитани на самом почетку приликом учитавања апликације. Другим речима, не постоји никаква додатна комуникација се сервером зарад дохватања страница корисничког интерфејса (осим у посебним случајевима одложеног или динамичког учитавања). Једина комуникација са сервером дешава се кроз позиве типа *AJAX* и то зарад дохватања, освежавања и модификовања података, затим извршавања команди и слања других захтева приступним тачкама као што су одложено и динамичко учитавање. Осим тога, треба имати у виду да комуникација се сервером не зависи од тога шта се тренутно приказује на екрану. Она се дешава у позадини и иницирана је током порука између компонената независно од тога да ли су компоненте које комуницирају видљиве на екрану или не.

Ова имплементација садржи неколико техничких решења и оптимизација које је чине скалабилном и примењивом у развоју великих индустријских система. На пример, постоји оптимизација комуникације између клијента и сервера која сакупља више индивидуалних захтева послатих независно и од стране различитих компонената у један заједнички захтев протокола *HTTP* који се шаље на сервер. Овакав захтев се отпакује и процесира на серверу. Уколико захтев садржи више индивидуалних захтева истог садржаја нпр. за читање истог атрибута, извршиће се само један приступ објектном простору, а резултат ће бити враћен као одговор на све идентичне захтеве. Овај цео процес је потпуно транспарентан за апликацију и програмера.

Резултат целокупне имплементационе архитектуре јесте да се већина процесирања везаног за кориснички интерфејс и презентацију премешта са сервера на клијента, док сервер брине само о одржавању објектног простора и извршавању пословне логике.

Поред механизма комуникације са сервером, имплементиран је и механизам међусобне комуникације између компонената који је у својој суштини механизам прослеђивања порука (енгл. *message passing*) са једноставним синхроним позивима операција на објектима језика *Java* (односно језика *JavaScript* након компајлирања). Ови позиви операција омогућавају да се извршавање понашања и компонената које шаљу, али и оних које примају поруке, дешава у истој нити и адресном простору веб претраживача без икаквог додатног деградирања перформанси у клијентском слоју у поређењу са класичним системима са обрадом догађаја (енгл. *event-driven systems*) и без додатног процесирања на серверу. Због тога се реакције компонената које преко порука реагују на догађаје из других компонената одликују брзим одзивом, као код десктоп апликација.

Треба напоменути да би се исти механизам прослеђивања порука може употребити и за комуникацију између слоја корисничког интерфејса и слоја објектног простора у ситуацијама када су та два слоја постављена у истом физичком окружењу (за разлику од описан клијент-сервер окружења). У том случају описана парадигма обезбеђује адекватно логичко одвајање слојева, при чему би имплементација користила обичне позиве операције

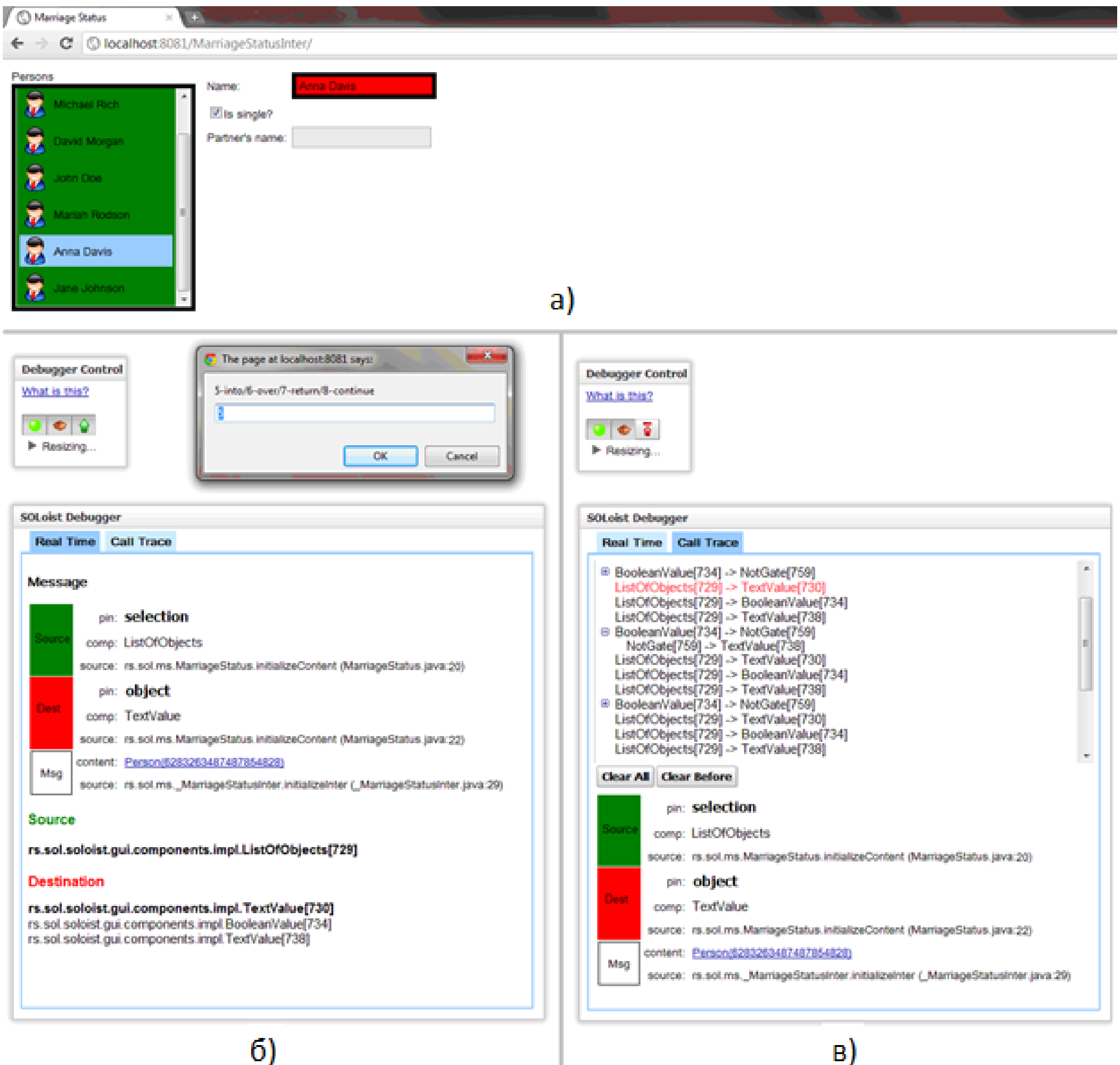
да врло ефикасно споји два слоја. Са друге стране, као што је већ показано, ова парадигма омогућава дистрибуцију слојева и у случају слојевите архитектуре као што је веб.

Као доказ примењивости парадигме у развоју савремених корисничких интерфејса за мобилне уређаје имплементиран је и прототип технологије за оперативни систем *Android*. Апликације развијене на прототипу остављају уобичајен утисак и изглед *Android* апликација.

У наставку ће бити приказан начин имплементације нове капсуле кроз који ће бити изнети конкретнији имплементациони детаљи. Више информација о имплементацији може се пронаћи и на веб сајту технологије *SOLoist* [47]. Ново окружење за моделовање је још увек у развоју.

### Помоћни алати

Као што је већ објашњено, интеракција компонената корисничког интерфејса обавља се преко пинова и жица. У пракси се показало да она може бити врло сложена и појавила се потреба за алатом који би помогао програмеру да лакше уочи евентуалне проблеме и грешке у свом моделу. То је управо била мотивација за развој једног таквог алата који ће укратко бити приказан у овом поглављу на примеру једне апликације (слика 48а).




Слика 48. Визуелни алат за проналажење проблема у интеракцији елемената корисничког интерфејса.

То је типична страница која приказује листу објеката особа на левој страни, и детаље изабране особе на десној. Пинови листе са леве стране су спрегнути жицама са пиновима компонената са десне стране. Приликом избора особе у листи са леве стране, порука са типизираном референцом на објекат особе шаље се преко пинова и жица свим компонентама које приказују детаље (у овом случају има три такве компоненте и према томе три поруке). Уз помоћ овог алата, за сваку поруку могуће је видети све потребне детаље.

Алат се интерно назива „визуелни дебагер“. Покреће се заједно са апликацијом, под условом да је подешена одговарајућим параметром за дебаговање. Када се апликација иницијално учита, алат се појављује на екрану у виду малог дијалога (означеног лабелом *Debugger Control* слике 48б и 48в) са основним функцијама за покретање, заустављање, подешавање величине дијалога у пикселима и за избор једног од два мода за праћење интеракције капсула. Приликом употребе визуелног дебагера отвара се нови дијалог са информацијама о току порука кроз пинове и жице (означен лабелом *SOLoist Debugger*). У зависности од режима рада, активан је један од два таба тог дијалога.

Мод праћења интеракције у реалном времену (слика 48б) омогућава да се конкретно у тренуцима слања сваке поруке од изворишног ка одредишном пину прикажу информације о поруци, компонентама које интерагују и пиновима у тренутку слања поруке. Другим речима, програм, односно интеракција у слоју корисничког интерфејса се паузира у тренутку слања поруке, како би програмер имао времена да анализира све информације. За време паузе програма, програмер анализира информације и одлучује се за једну од следећих акција којима се наставља извршавање програма и зауставља у наредном тренутку који је дефинисан изабраном акцијом:

- Притиском на тастер 5 (аналогно тастеру *F5 – step into,* алата за дебаговање у програму *Eclipse*) програмер прелази на прву следећу поруку (односно програм наставља да извршава и зауставља се код прве следеће поруке) коју шаље компонента којој припада одредишни пин ка било којој другој компоненти. Другим речима, програмер прати пропагацију порука „у дубину“.
- Притиском на тастер 6 (аналогно тастеру *F6 – step over,* алата за дебаговање у програму *Eclipse*) програмер прелази на прву следећу поруку коју шаље компонента којој припада изворишни пин ка било којој другој компоненти. Другим речима, програмер прати пропагацију порука „у ширину“.
- Притиском на тастер 7 (аналогно тастеру *F7 – step out / return,* алата за дебаговање у програму *Eclipse*) програмер прелази на прву следећу поруку коју шаље компонента родитељ (у стаблу пропагације порука) компоненте којој припада изворишни пин. Ово је аналогно изласку из методе, односно преласку на следећу наредбу у телу методе позиваоца, приликом дебаговања извршавања програмског кода.
- Притиском на тастер 8 (аналогно тастеру *F8 – continue,* алата за дебаговање у програму *Eclipse*) програм се наставља без заустављања, све док се пропагација порука не заврши.

Режим накнадног анализирања порука приказан је на слици 48в. Извршавање програма се не паузира, већ се након завршетка пропагације порука анализира стабло пропагације.

У оба мода анализе пропагације порука, алат приказује следеће информације о поруци, компонентама које интерагују, њиховим пиновима итд:

- назив изворишног пина,
- класу изворишне компоненте,
- број линије у коду модела где је инстанцирана изворишна компонента,
- назив одредишног пина,
- класу одредишне компоненте,
- број линије у коду модела где је инстанцирана одредишна компонента,
- садржај поруке и
- број линије у коду модела где је инстанцирана жица која спаја изворишни и одредишни пин.

# Део 5: Анализа предложеног решења

Провера и потврда предложеног решења спроведена је кроз три кампање. Прва је имала за циљ да провери практичну употребљивост предложене технике у великим софтверским пројектима, као и перцепцију и степен прихватања ове технологије од стране програмера. Друга кампања била је усмерена ка аналитичком поређењу предложеног решења са скупом изабраних технологија у широкој употреби, а све у контексту оквира одговорности у развоју корисничких интерфејса који је изложен на почетку ове дисертације. Трећа кампања је имала за циљ да провери и упореди перформансе постојеће имплементације са перформансама других технологија.

# *Практична употребљивост и степен прихватања*

Циљеви ове кампање били су следећи:

1. Проценити практичну примењивост технике предложене у овој дисертацији у великим софтверским пројектима тако што треба утврдити:
    a. да ли се коришћењем ове технике може успешно одговорити на широк спектар и велику већину захтева и потреба реалних апликација у домену корисничког интерфејса и
    b. да ли се предложена техника може применити на велике апликације и системе, уз обавезно пружање адекватних метрика које говоре о величини испитаних система.
2. Проценити степен прихватања технике предложене у овој дисертацији од стране програмера и инжењера (искључујући аутора) тако што треба:
    a. утврдити да ли програмери могу да усвоје и са успехом користе предложену технику и
    b. обезбедити повратне информације о коришћењу предложене технике

Интеракција графичких елемената преко пинова и жица, као идејно решење, осмишљена је још 2004. године [26]. Пре него што је направљена прва верзија технологије која подржава ову парадигму, аутор ове дисертације и колеге из развојног тима, пре имплементације циљног корисничког интерфејса неке апликације, користили су ову идеју за описивање и дизајн корисничког интерфејса само концептуално, на скицама и дијаграмима. Овакав приступ коришћен је у неколико мањих и пројеката средње величине. Након овог периода позитивног прелиминарног искуства, била је направљена прва верзија технологије за десктоп окружење. Ова верзија технологије коришћена је од стране неколико програмера у неколицини пројеката средње величине у различитим доменима као што су комерцијала, администрација, инжењерство. У том периоду унапређено је разумевање главних концепата и повећана је база уграђених графичких компонената. Веб базирана имплементација описана у овој тези је, према томе, друга генерација ове технологије развијана почев од 2009. године. Од тада је технологија коришћена у четири већа индустријска пројекта (који ће овде бити укратко приказани) у периоду 2009.-2012. године, као и у низу других пројеката након тог периода до данас.

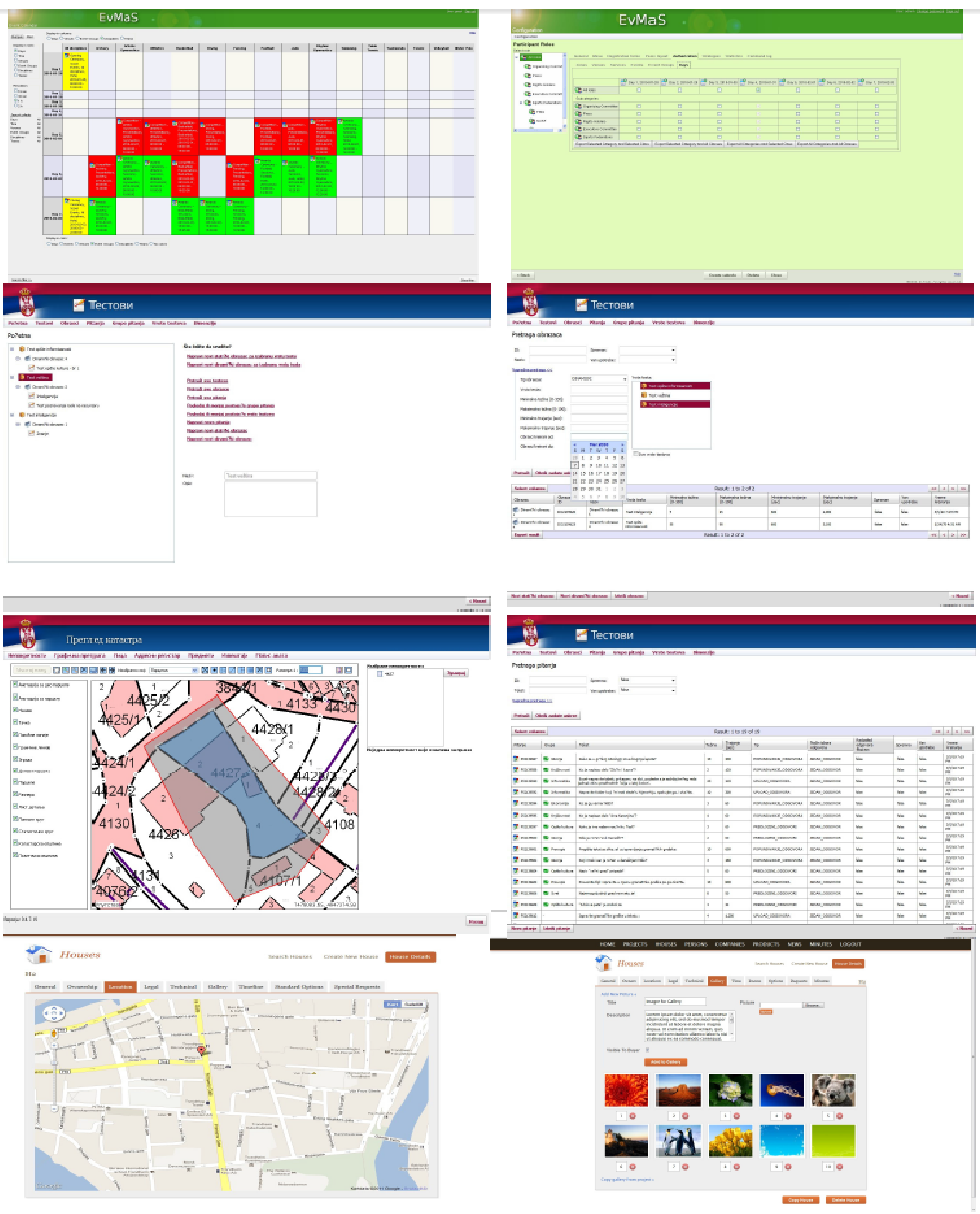
Први пројекат који је анализиран овде, систем за управљање друштвеним догађајима (ЕМС), јесте изузетно флексибилан производ за информатичку подршку великим спортским догађајима, симпозијумима, конвенцијама, конференцијама, такмичењима итд. Овај систем подржава регистрацију учесника, одобравање регистрације, акредитације, права приступа местима одржавања догађаја, подешавање и дизајн, штампање и издавање пропусница, управљање салама и местима одржавања догађаја, управљањем временским распоредом догађаја итд.

Други пројекат, државни систем за управљање кадровима (ХРМС), је систем за подршку процеса запошљавања у државним органима и институцијама. Систем подржава централизовано управљање државним органима, радним местима и отвореним конкурсима свих државних институција. Систем омогућава формирање конкурса за радна места, пријављивање кандидата на конкурсе и вођење тока конкурса, тестирање вештина и знања кандидата, оцењивање и рангирање итд.

Трећи пројекат је национални систем катастра непокретности (РЕЦС). Овај систем подржава све стандардне концепте и функционалности које један такав систем треба да има, укључујући катастарске ентитете као што су парцеле, потпарцеле, објекти, подобјекти, али и правне ентитете као што су власништва, терети и хипотеке.

Четврти пројекат је информациони систем за односе за клијентима, развијен за једно предузеће за трговину некретнинама. Систем подржава процес продаје некретнина, односно администрацију учесника процеса, пројеката, кућа и станова, опремање некретнина, комуникацију са клијентима, управљање документацијом, управљање опцијама везаним за опрему некретнине, измене захтева клијената, ценовнике итд. Систем је био интегрисан са неколико екстерних производа у домену корисничког интерфејса, што је представљало изазов у смислу интеграције предложене технологије са екстерним технологијама.

Све описане апликације имале су изузетно захтевне корисничке интерфејсе са десетинама или стотинама форми и екрана. Разноликост њихових корисничких интерфејса је велика, а превасходно је последица жеља корисника као и потреба њихових домена примене. У апликацијама је била уграђена и неколицина графичких компонената специфичних за сваку од апликација, од којих су неке служиле као компоненте за интеграцију са компонентама екстерних технологија. Слика 49 илуструје разноликост ових апликација.




Слика 49. Слике екрана из индустријских система.

Табела 9 даје преглед величине модела домена на језику *UML* за описана четири пројекта. Метрика се односи на структурне делове модела (класе, атрибуте и асоцијације), као и нетривијалне операције пословне логике система (класе команди). Метрика се односи само на доменске (перзистентне) класе концептуалног модела, а не на класе везане за кориснички интерфејс. Бројеви дати у табели практично дају утисак о природној комплексности ова четири система.

| Метрика | Систем | | | |
|---|---|---|---|---|
| | ЕМС | ХРМС | РЕЦС | ЦРМС |
| Број доменских класа – укупно / апстрактних / конкретних | 81/13/68 | 86/5/81 | 196/17/179 | 28/4/24 |
| Број доменских атрибута – укупно / макс. / просечно по класи | 218/13/2.69 | 566/52/6.58 | 280/17/1.43 | 130/22/4.64 |
| Број доменских асоцијација / асоцијационих крајева | 79/158 | 132/264 | 254/508 | 32/64 |
| Број навигабилних доменских асоцијационих крајева – укупно / макс. / просечно по класи | 139/11/1.72 | 228/18/2.65 | 336/29/1.71 | 50/6/1.79 |
| Број атрибута и навигабилних асоцијационих крајева – укупно / макс. / просечно по класи | 357/18/4.41 | 794/54/9.23 | 616/46/3.14 | 180/27/6.43 |
| Величина хијерархије класа – максимална / просечна дубина | 4/1.95 | 3/1.24 | 5/2.36 | 3/1.50 |
| Број доменских енумерација | 10 | 32 | 15 | 13 |
| Број командних класа | 63 | 186 | 193 | 70 |
| Укупан број класа (доменских и командних) | 144 | 272 | 389 | 98 |

Табела 9. Метрика модела домена за четири индустријска пројекта.

Табела 10 приказује метрику слоја корисничког интерфејса за описана четири информациона система. Бројеви у табели одговарају једној инстанци целокупног модела корисничког интерфејса, односно величини корисничког интерфејса који одговара једној корисничкој сесији а покрива целокупну функционалност система.

| Метрика | Систем | | | |
|---|---|---|---|---|
| | ЕМС | ХРМС | РЕЦС | ЦРМС |
| Укупан број графичких компонената | 16,883 | 6,925 | 59,448 | 1698 |
| Хијерархија садржавања компонената – максимална дубина / просечна дубина | 15/8.55 | 21/11.23 | 32/16.40 | 16/10.65 |
| Број контејнерских компонената | 2,058 | 1,928 | 18,046 | 395 |
| Број примитивних компонената | 14,825 | 4,997 | 41,402 | 1303 |
| Број жица | 12,104 | 4,255 | 31,285 | 1221 |
| Макс. / просечан број компонената делова које садржи једна контејнерска компонента | 101/8.22 | 65/3.60 | 57/3.28 | 28/4.29 |
| Број програмера укључених у развој корисничког интерфејса | 5 | 7 | 10 | 3 |

Табела 10. Метрика слоја корисничког интерфејса за четири индустријска пројекта.

Према резултатима приказаним горе, може се закључити да се техника предложена у овој дисертацији може применити на пројекте различитих величина. У наставку ће бити изнето још неколико интересантних запажања добијених кроз искуство примене ове технике.

У прва два пројекта (ЕМС и ХРМС) приступ развоју није наметао употребу капсула као апстракција са интерном структуром и стриктном енкапсулацијом. Користио се само механизам хијерархијске уређености инстанци компонената, као и повезивање пинова компонената преко жица. Иако је такав развој био задовољавајуће ефикасан, појавили су се проблеми са одржавањем: чим је структура корисничког интерфејса постала већа и

сложенија, жице су се испреплетале и постале тешке за разумевање, дебаговање и одржавање. Разлог је тај што је било дозвољено да жице прелазе границе фрагмената корисничког интерфејса и било је тешко да се у потпуности разуме повезивање фрагмента корисничког интерфејса са осталим фрагментима из окружења.

Ово је разлог зашто је уведен концепт капсуле као апстракције са стриктном енкапсулацијом и потенцијалом за поновну употребу. Овај приступ је коришћен (у једном рудиментарном и неформалном облику) у два последња описана система (РЕЦС и ЦРМС) и одмах је примећено значајно побољшање и решење споменутог проблема. Концепт капсуле је такође значајно унапредио потенцијал за поновном употребом фрагмената корисничког интерфејса.

На основу описаног искуства дошло се такође до закључка да је механизам нотификација још једна врло важна компонента целе технологије. У поређењу са другим технологијама, развој корисничких интерфејса је лакши и комфорнији захваљујући овом механизму.

Показало се и да модели корисничког интерфејса, као и сам начин размишљања врло често подсећају на дизајн дигиталних електронских кола. Ово је сасвим разумљиво, имајући у виду сличну семантику и чињеницу да се капсуле понашају као електронске компоненте које шаљу и примају сигнале преко пинова и жица. Овај ефекат још више подржава јасно одвајање одговорности слојева корисничког интерфејса и пословне логике апликације. Разлог је у томе што програмери лако повезују капсуле како би реализовали једноставнију функционалност, док комплексније процедуре остављају слоју пословне логике.

У последњем реду табеле 10 дат је број програмера који су учествовали у развоју корисничког интерфејса споменутих система (ово није укупан број програмера који су учествовали у развоју ових система). Укупан број различитих особа које су учествовале у развоју корисничких интерфејса ових система, искључујући аутора, је 15. У развој су били укључени како млађи, тако и старији програмери са искуством рада са неколико технологија за развој корисничких интерфејса и укупно три предузећа за развој софтвера.

Групи од ових 15 програмера понуђен је упитник како би се утврдило њихово мишљење и искуство у раду са техником предложеном у овој дисертацији. У упитнику је од програмера најпре затражено да изаберу једну другу технологију за развој корисничких интерфејса коју најбоље познају или коју сматрају најбољом од технологија које познају. У наставку се тражило да се изабрана технологија упореди на бази више одговорности / аспеката у развоју корисничких интерфејса са технологијом предложеној у овој дисертацији. За сваки аспект развоја, обе технике су оцењиване оценама од 1 до 5 (1 – незадовољавајуће, 2 – задовољавајуће, 3 – добро, 4 – врло добро и 5 – одлично).

Попуњени упитници стигли су од 9 програмера, док је један програмер послао чак 5 попуњених упитника, односно поређења технике предложене у овој дисертацији са 5 других популарних технологија. Укупно је било 8 различитих технологија са којима је вршено поређење: *Swing, GWT+MVP+UiBinder, JFace+SWT, JSP 2, Ruby on Rails, C#.net+Visual Studio, AndroidUI,* и *Spring.* Табела 11 даје преглед просечних оцена.

| Аспект | Описана технологија – просек (А) | Остале технологије – просек (Б) | Разлика (А – Б) |
|---|---|---|---|
| Енкапсулација (подршка / спровођење) | 3.38 | 2.55 | 0.83 |
| Читљивост кода и организација | 2.89 | 3.17 | -0.28 |
| Поновна употреба фрагмената | 3.44 | 3.75 | -0.31 |
| Квалитет уграђених компонената | 3.75 | 3.36 | 0.39 |
| Лакоћа имплементације интеракције и | 4.44 | 3.17 | 1.28 |

| | | | |
|---|---|---|---|
| понашања | | | |
| Лакоћа имплементације спреге са пословном логиком | 4.44 | 3.75 | 0.69 |
| Лакоћа дефинисања распореда | 3.11 | 4.25 | -1.14 |
| Лакоћа стилизације | 2.89 | 4.17 | -1.28 |
| Лакоћа прављења специфичних компонената / проширивање | 2.67 | 4.42 | -1.75 |
| Подршка за одвајање одговорности корисничког интерфејса и пословне логике (забрана мешања) | 4.63 | 2.82 | 1.81 |

Табела 11. Упоредни преглед просечних оцена технологија.

Овде је важно напоменути да је су програмери највећи део искуства рада са техником предложеном у овој дисертацији добили у периоду пре него што је уведен концепт капсуле са подршком за стриктну енкапсулацију, поновну употребу фрагмената и бољу организацију кода комплексних делова корисничког интерфејса. Ово је очигледно утицало на оцене за прва три аспекта у табели 11. Такође, ово може да објасни и мало лошије оцене за читљивост кода (2,89 наспрам 3,17) и за подршку поновној употреби (3,44 наспрам 3,75). Упркос томе, техника предложена у овој дисертацији је добила значајно већу просечну оцену за енкапсулацију (3,32 наспрам 2,55). Поред тога, и за остале аспекте развоја корисничког интерфејса који су разматрани у овој дисертацији добијене су боље просечне оцене: за квалитет уграђених компонената (3,75 наспрам 3,36), лакоћу имплементације интеракције и понашања (4,44 наспрам 3,17), лакоћу имплементације спреге са пословном логиком (4,44 наспрам 3,75), и подршку одвајању одговорности корисничког интерфејса и пословне логике (4,63 наспрам 2,82).

Ниже просечне оцене добијене су за аспекте који јесу важни, али нису карактеристични за приступ предложен у овој дисертацији: дефинисање распореда, стилизације и прављење специфичних компонената. Ово се може објаснити незрелошћу тренутне имплементације и недостатком помоћних алата који још увек не подржавају споменуте техничке аспекте као што је случај са технологијама у широкој употреби. У наредним фазама развоја и ови аспекти ће бити унапређени.

У делу упитника где су програмери могли да унесу свој коментар о предностима и манама технологије предложене у овој дисертацији у виду слободног текста појавили су се и следећи позитивни коментари и похвале: за декларативан и лак начин повезивања компонената, лак приступ подацима и пословној логици и добру спрегу са моделом на језику *UML,* стриктну одвојеност пословне логике и слоја корисничког интерфејса, одличан уграђени механизам нотификације и транспарентну подршку за протокол *AJAX*, интуитивни концепти и лако усвајање. Са друге стране, критикована је имплементација и неодговарајућа подршка алата, смањена флексибилност у смислу конфигурације графичких елемената и ограничења у решавању једноставних задатака ниског нивоа специфичних за веб окружење.

# *Аналитичко поређење са другим технологијама*

Ова кампања имала је за циљ да упреди неке аспекте развоја корисничких интерфејса технике предложене у овој дисертацији са другим технологијама, и то анализом већ развијених апликација. Анализа се базирала на:

1. процени квалитета програмског кода, са циљем да се утврди степен којим сама технологија спречава појаву лошег кода и елемената лоше програмерске праксе и
2. поређењу експресивности технологије, поређењем величине програмског кода неопходног да се имплементира одређени задатак.

Овде вреди напоменути да циљ није био да се утврди да је било која технологија генерално боља од других или да је сама по себи довољна да се добије квалитетан интерни дизајн програма са мало кода. Штавише, свака технологија омогућава да се направе добри програми. Образован, вешт и дисциплинован програмер направиће добар програм у свакој технологији коју познаје. Уместо тога, циљ је био да се испита квалитет кода реалних, већ развијених апликација, независно и непристрасно.

Због свега тога, како би се постигао највиши ниво објективности и поузданости резултата, анализиране апликације из других технологија нису развијане специјално за потребе овог експеримента, јер би се тиме угрозила непристрасност или због недовољног познавања других технологија од стране извођача, или због чињенице да би приликом развијања програмери били свесни да ће програмски код бити накнадно оцењиван и анализиран. Уместо тога, узете су у разматрање већ постојеће апликације из других технологија. Одабране су следеће три групе технологија:

1. Група *Django/Pyhton,* као пример технологија базираних на шаблонима,
2. Група *GWT/Java,* као пример објектно оријентисаних технологија и
3. Група *ASP.net/C#,* као пример хибридних технологија.

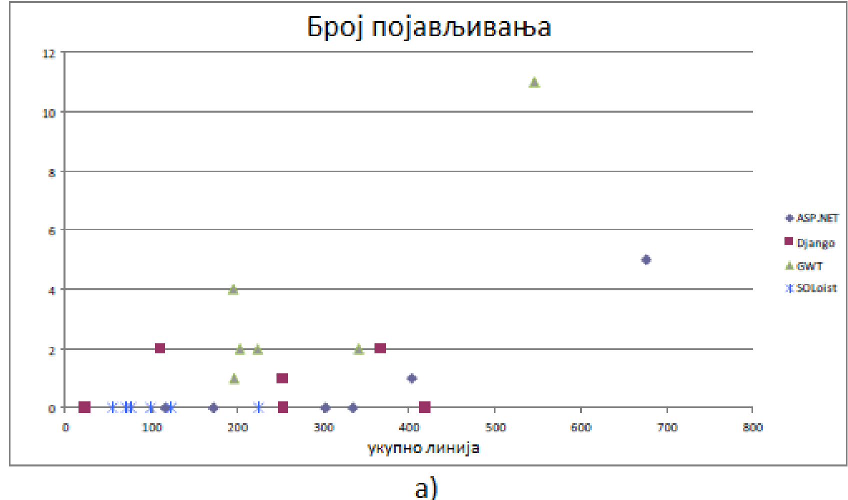
За сваку од ове три групе технологија изабране су по две пословне апликације отвореног кода доступне на вебу и развијене од аутору потпуно непознатих група програмера. Избором по две апликације развијене од стране независних програмерских тимова умањен је утицај индивидуалних вештина и дисциплине на спроведену анализу. Поред ових изабраних апликација, изабране су и две у том тренутку најновије апликације развијене помоћу технике описане у овој дисертацији. У овим апликацијама анализирани су модели

преведени у код на језику *Java*: капсуле су репрезентоване класама, док је за конструкторе писан иницијализациони код на језику *Java*.

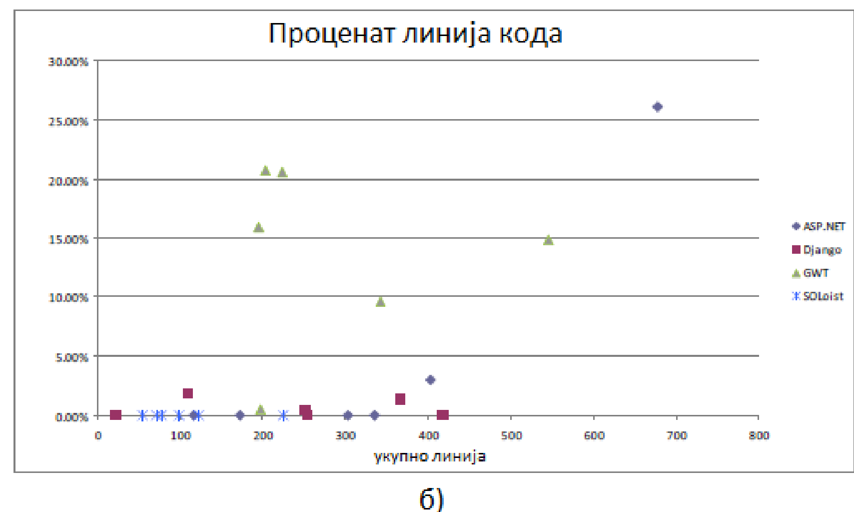
За сваку од осам изабраних апликација изабрана су по три репрезентативна модула корисничког интерфејса. Бирани су они модули који представљају типичне примере форми и страница корисничког интерфејса. За сваки од модула анализиран је програмски код и пребројаване линије кода које припадају различитим одговорностима и развоју корисничких интерфејса (једна линија може да припада једној или више одговорности). Иако је број линија кода дискутабилна метрика, разумно је претпоставити да он ипак одражава неке аспекте комплексности.

Прва анализа кода имала је за циљ да утврди присуство образаца лоше праксе мешања кода пословне логике и корисничког интерфејса (енгл. *magic pushbutton anti-pattern*[29]). У сваком модулу пробројана су оваква појављивања, као и број линија кода које су биле заслужне за ову појаву. Слика 50 приказује резултате. На оба дијаграма на оси *x* дат је укупан број линија кода у модулу (величина модула). На слици 50а на оси *y* приказан је број појављивања овог обрасца, док је на слици 50б на оси *y* приказан проценат инволвираних линија програмског кода.

[29] http://en.wikipedia.org/wiki/Magic_pushbutton




а)



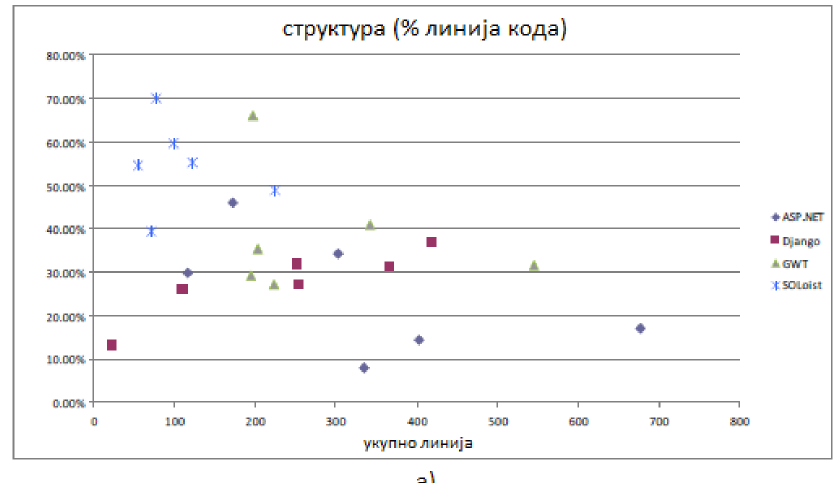

б)

Слика 50. Присуство анти-обрасца *Magic pushbutton.* а) број појављивања, б) проценат линија програмског кода.

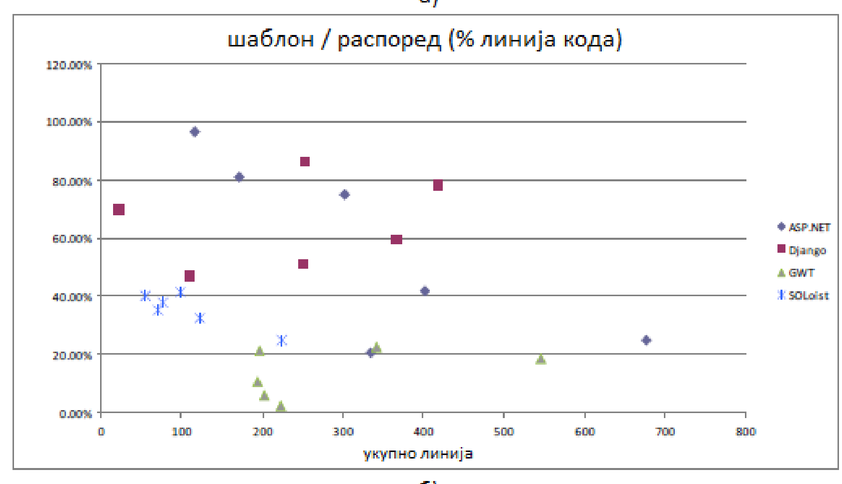
Из резултата датих на слици 50 може се закључити да програмери заиста падају у замку овог анти-обрасца уколико технологија не спречи овај неадекватни стил програмирања. За све узете апликације, осим оних направљених уз помоћ технике представљене у овој дисертацији, овај проблем појављивао се у више од једног анализираног модула, док је у групи модула технологије *GWT/Java* ова појава пронађена у сваком модулу. У модулима у којима је пронађена, појављивала се од један до једанаест пута узимајући при томе од неколико, па све до двадесет процената линија кода. У модулима апликација развијених уз помоћ технике представљене у овој дисертацији, ова појава није пронађена ниједном.

Слика 51 приказује још неке интересантне карактеристике садржаја анализираних модула. Модули направљени у технологији представљеној у овој дисертацији имају највећи проценат линија кода који адресира структуру, односно креирање (инстанцирање) графичких елемената (увек изнад 40%, често и изнад 50%) за разлику од осталих технологија које углавном имају мање од 40% линија кода посвећених структури. Са друге стране, кад су у питању друге одговорности, распоред (40% и мање), понашање (15% и мање) и посебно приступ подацима (мање од 20%), ситуација је обрнута: друге технологије

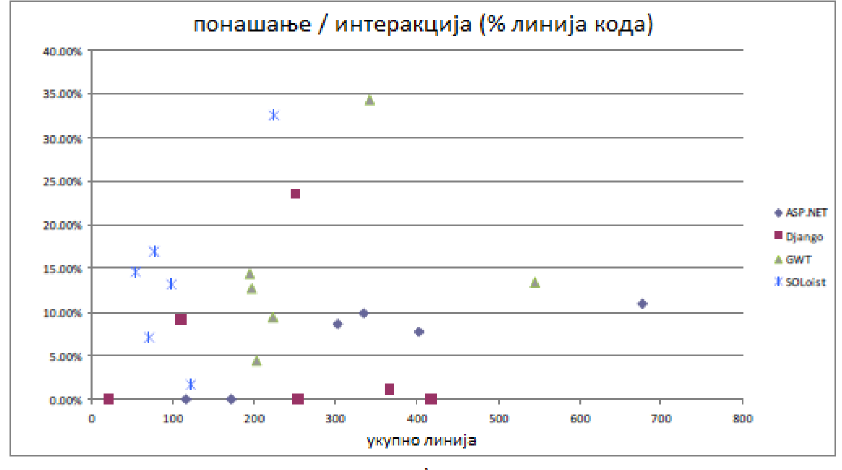
посвећују знатно више програмског кода овим аспектима програмирања корисничких интерфејса.



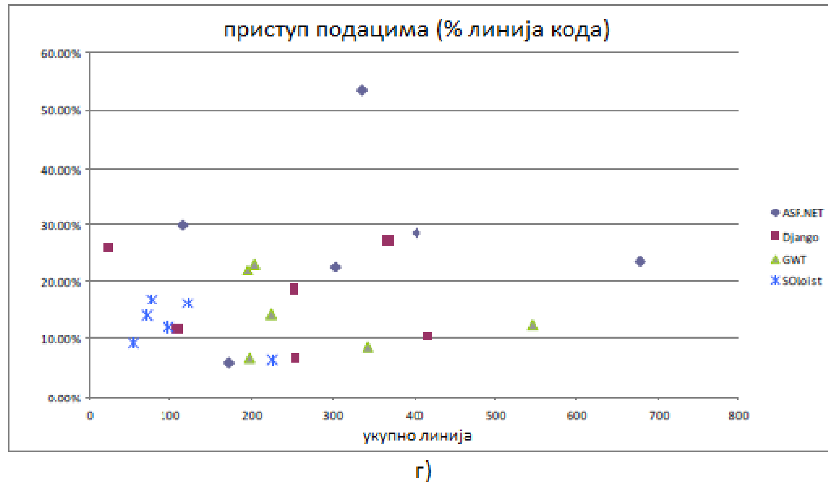

а)




б)

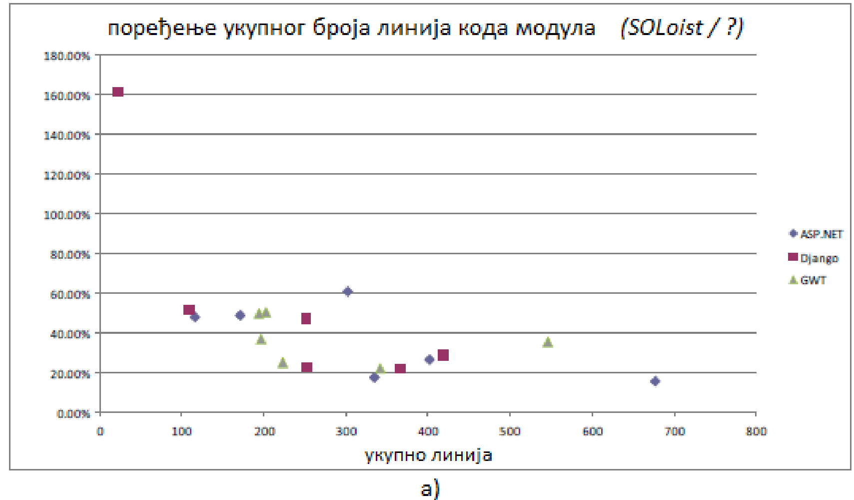



в)

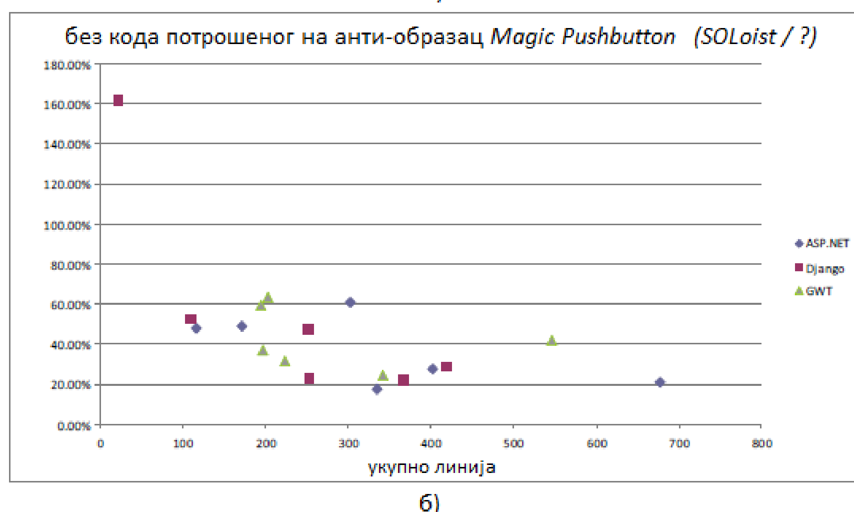



Слика 51. Поређење различитих аспеката анализираних апликација. Проценат линија програмског кода потрошен на дефинисање а) графичких елемената у модулу, б) распореда, в) понашања и г) приступа подацима.

Други експеримент ове кампање је подразумевао да се направи функционални и визуелни еквивалент за сваки од модула из других технологија уз помоћ технике представљене у овој дисертацији и да се упореди укупан број линија кода у обе изведбе. Претпоставка приликом прављења еквиваленатa је била да већ постоје одговарајући концептуални модел и позадинска пословна логика, тако да је фокус био само на слоју корисничког интерфејса. Слика 52 приказује однос броја линија кода еквиваленатa направљених у техници предложеној у овој дисертацији и одговарајућих модула направљених у другим технологијама. Осим појединачног случаја изузетно малог модула са свега шеснаест линија кода, модул еквивалент је у свим случајевима био свега 20% до 60% величине свог парњака, са очигледном тенденцијом опадања са порастом величине модула. Може се закључити да се техника представљена у овој дисертацији скалира боље од других технологија анализираних у овом поглављу.




а)




б)

Слика 52. Поређење величине (у броју линија кода) сваког модула из других технологија са својим функционалним и визуелним еквивалентом направљеним у технологији представљеној у овој дисертацији: а) када модули других технологија садрже код анти-обрасца *Magic Pushbutton* и б) када је код анти-обрасца *Magic Pushbutton* изузет из кода модула других технологија.

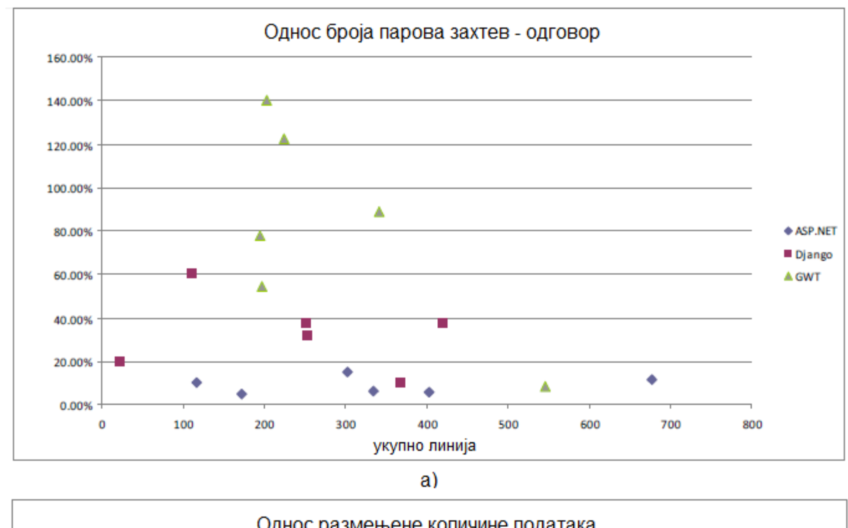
## *Евалуација перформанси*

Циљ ове кампање био је да се процене одређени аспекти тренутне имплементације који утичу на перформансе и одзив апликација направљених уз помоћ технике представљене у овој дисертацији, као и да се у контексту тих аспеката тренутна имплементација упореди са другим технологијама. Конкретно, циљ је био да се упореди количина размене информација између клијента и сервера у тренутној имплементацији и у другим технологијама.

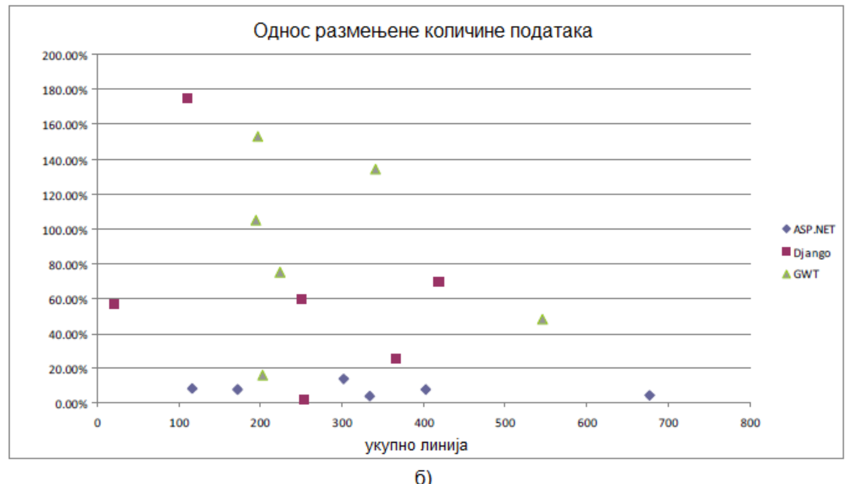
У овој кампањи коришћен је исти скуп апликација и модула као и у претходној. За сваки модул, односно веб страницу, измерена је укупна количина информација која се размени између клијента и сервера у сценарију типичном за ту страницу. Сценарио почиње учитавањем странице, наставља се евентуалним прегледом података на страници и изменом података на форми, и завршава се командом која је карактеристична за ту страницу и која учитава исту или неку другу страницу. За такав сценарио измерено је следеће:

1. број порука, односно број парова захтев-одговор размењених између клијента и сервера и
2. количина података (у бајтовима) пренетих у оба смера у току сценарија; овде се нису рачунали подаци који се кеширају у веб претраживачу, јер се они не преносе преко мреже.

За сваки модул и карактеристичан сценарио направљено је поређење резултата са резултатима одговарајућих еквивалената направљених у техници представљеној у овој дисертацији. Слика 53 приказује резултате мерења.




а)




б)

Слика 53. Поређење количине информација размењених између клијента и сервера за сваки модул направљен у другој технологији и одговарајућих функционалних и визуелних еквивалената направљених у техници описаној у овој дисертацији и то у виду а) броја парова захтев-одговор и б) количине података пренетих преко мреже.

На слици 53а види се да у већини случајева модули еквиваленти размене мање порука него модули из других технологија, осим у два случаја модула направљених у технологији *GWT* (120% и 140%). У случају модула направљених у технологији *ASP.NET* еквиваленти направљени у техници описаној у овој дисертацији размене више од пет пута мање порука.

На слици 53б приказани су односи количине података размењених преко мреже. У већини случајева, тренутна имплементација технике описане у овој дисертацији, односно модули еквиваленти, се показују ефикаснији од својих парњака (поново су ефикаснији више од пет пута од модула направљених у технологији *ASP.NET*). Само у неколико случајева, модули еквиваленти су се показали лошије, али у границама прихватљивог (између 100% и 180%). Овде треба напоменути да постоји још простора за смањење количине података у тренутној имплементацији, захваљујући чињеници да одређени делови порука које се размењују још увек нису компримовани.

Може се закључити да, иако на први поглед техника описана у овој дисертацији уноси поплаву порука које се размењују између слоја корисничког интерфејса и слоја пословне логике, у реалности то није случај. Захваљујући интерним оптимизацијама, размера

комуникације између клијента и сервера је у већини случајева мања него код других технологија, или ако није мања, ипак је у границама прихватљивог.

# Део 6: Закључак

# *Закључак*

У овој дисертацији приказана је техника за моделовање и имплементацију графичких корисничких интерфејса пословних апликација. Описана техника је базирана на раздвајању одговорности, апстракцији, хијерархијској декомпозицији и одговарајућој спрези са објектним простором са семантиком језика *UML*.

Принцип раздвајања одговорности формулисан је кроз оквир (енгл. *framework*) кога чине најважније одговорности у развоју графичких корисничких интерфејса: структура, распоред, понашање, приступ подацима, форматирање података, валидација и стилизација. Спроведене су емпиријске студије које поткрепљују управо овакав начин раздвајања одговорности. Иако је употребљен као основа за технику описану у овој дисертацији, овај оквир има и ширу примену. У посебној студији изложеној у другом делу дисертације, дошло се до резултата који говоре у прилог чињеници да је описани оквир одговорности применљив на већину комерцијалних технологија за развој корисничких интерфејса.

У овој дисертацији пропагира се раздвајање свих идентификованих одговорности у развоју корисничких интерфејса. Међутим, у пракси се врло често дешава да одговорности које уопште нису део слоја корисничког интерфејса ипак нађу своје место у том слоју. Како би искоренила ову лошу праксу, техника представљена у овој дисертацији спречава мешање слоја корисничког интерфејса и слоја пословне логике (кога не треба поистоветити са делом слоја корисничког интерфејса за приступ подацима), односно мешање пословне логике у изворни код корисничког интерфејса.

Осим оквира одговорности који представља најширу основу на којој се базира описана техника, у трећем делу изложени су неки врло специфични концепти који чине саму суштину описане технике. Наиме, техника се заснива на кључном концепту капсуле. Процес апстракције помоћу капсуле са интерном структуром може се применити рекурзивно и кохерентно на свим нивоима детаља у оквиру модела или програма корисничког интерфејса, почевши од највишег нивоа архитектуре, па све до најнижег нивоа моделовања појединачних графичких елемената. Концепт капсуле обезбеђује и подржава апстракцију како појединачних графичких елемената, тако и целих фрагмената корисничког интерфејса са наглашеном енкапсулацијом и подршком за поновну употребу. Управо ово представља фундаменталне предуслове скалабилности технике описане у овој дисертацији.

Интерфејс капсуле дефинише се помоћу пинова, који представљају други важан концепт ове технике. Интеракција капсула дефинише се декларативно, повезивањем пинова жицама. Декларативни приступ програмирању и моделовању интеракције смањује количину уложеног труда у развоју корисничких интерфејса, као и случајну комплексност коју има сада већ превазиђени начин функционалног спрезања графичких елемената у методама за управљање догађајима.

Сви концепти технике описане у овој дисертацији су формално дефинисани и међусобно добро спрегнути. Техника подразумева лингвистичку кохерентност, односно унифицираност синтакси језика за моделовање и програмирање. Према томе, техника није подложна семантичким, фазним или размимоилажењима опсега [44].

Техника описана у овој дисертацији има вишегодишњу историју развоја и примене. Неки њени саставни делови, као што је библиотека графичких елемената описана у делу о имплементацији, практично су потврђени у индустријским пројектима у земљи и иностранству. Показало се да техника омогућава, али и значајно олакшава, развој најсложенијих и најзахтевнијих корисничких интерфејса пословних апликација. С друге стране, неки елементи ове технике су релативно млади и још чекају на практичну потврду и додатна унапређења.

У петом делу ове дисертације изложени су резултати експеримената који су потврдили супериорност описане технике у односу на технике у широкој употреби у одређеним аспектима развоја корисничких интерфејса. Експерименти су показали предности ове технике у спровођењу принципа енкапсулације, као и у домену имплементације интеракције између графичких елемената и понашања. Такође, резултати говоре у прилог квалитету библиотечких графичких елемената ове технике, али и у прилог подршци за раздвајање слојева корисничког интерфејса и пословне логике.

На крају, важно је напоменути да описана техника не преферира ниједан специјални стил интеракције или тип корисничког интерфејса. Другим речима, техника подржава развој корисничких интерфејса базираних како на задацима и такозваним „чаробњацима“ (енгл. *wizard*), тако и развој интерфејса оријентисаних ка подацима (енгл. *data-centric*), па чак и комбиноване варијанте. Другим речима, јака спрега са доменским објектним простором не имплицира да кориснички интерфејс мора представљати податке у изворном облику, иако такав облик може да буде прикладан у неким случајевима. Напротив, техника је општа и довољно флексибилна да подржи развој како интерфејса оријентисаних ка подацима (енгл. *object-action, noun-verb*) тако и оних функционално оријентисаних (енгл. *verb-noun*).

# *Будуће активности и правци развоја и истраживања*

Ова дисертација заснива се на принципима раздвајања одговорности, енкапсулације и хијерархијске декомпозиције у домену развоја и имплементације графичких корисничких интерфејса. На основу ових принципа развијени су, и добрим делом у пракси проверени, оквир одговорности у развоју корисничких интерфејса, техника моделовања, доменски језик и апликативни програмски интерфејс за развој корисничких интерфејса. Свака од ових целина отвара нове могућности за истраживање и развој.

Оквир одговорности за развој графичких корисничких интерфејса описан у овој дисертацији, иако заснован на опсежној емпиријској анализи (апликативног програмског интерфејса, изворног кода и пријављених проблема), ипак захтева додатни напор у смислу провере, тестирања и усклађивања са потребама и очекивањима програмера корисничких интерфејса. У току процеса идентификације одговорности описаног у другом делу ове дисертације, спорадично су се појављивали елементи који су указивали на постојање још неких одговорности. Другим речима, неки елементи нису могли бити објашњени приказаним оквиром одговорности. Као један од примера одговорности које се потенцијално могу уврстити у оквир јесте приступачност (енгл. *accessibility*), односно прилагођавање корисничких интерфејса корисницима са специјалним потребама. Са друге стране, потребно је преиспитати одговорности које се већ налазе у оквиру и додатно их емпиријски проверити. Један од начина преиспитивања предложеног оквира подразумева анализу дефиниција одговорности и евентуално спајање или додатно елаборирање појединих одговорности. У сваком случају, може се рећи да вреди учинити додатни напор и потенцијално кориговати предложени оквир у складу са резултатима будућих анализа и промишљања.

Техника моделовања која је представљена у трећем делу ове дисертације отвара пут развоју нових алата и окружења за моделовање корисничких интерфејса. С обзиром на своју превасходно декларативну природу, ова техника нуди велике могућности у контексту развоја помоћних алата и окружења који би били засновани на принципу *WYSIWYG* (енгл. *what you see is what you get*) и који би значајно убрзали процес моделовања корисничких интерфејса у односу на данашње језике, окружења и алате који су у великој мери базирани на тексту.

С друге стране, као противтежа принципу графичког моделовања корисничких интерфејса, у овој дисертацији описан је и доменски језик који ипак нуди могућност развоја корисничких интерфејса на бази текста. Описани доменски језик очекује провера у пракси и сазревање. Пре тога, у блиској будућности очекује се унапређење компајлера, проширивање библиотеке капсула, као и развој помоћних алата и окружења.

Као што је већ речено, апликативни програмски интерфејс описан у делу *Имплементација,* интензивно је употребљаван у индустријским пројектима у земљи и иностранству. Фаза концептуалног сазревања је добрим делом завршена, а библиотека је већ претрпела неколико итерација суштинског рефакторисања (енгл. *refactoring*) и то највише у домену унапређења употребљивости (енгл. *usability*) и редизајна јавних класа. Ипак, остаје да се имплементира неколико важних оптимизација и унапређења. Најважнија системска оптимизација подразумева рестриктивније коришћење меморијских ресурса. Поред тога, потребно је унапредити и значајно олакшати процес имплементације специфичних капсула, односно нових елемената библиотеке, пошто се у пракси показало да, без обзира на богатство библиотеке уграђених капсула, увек постоји потреба за новим, изразито специфичним капсулама.

У будућности, аутор ће наставити развој, истраживање и прикупљање нових искустава у примени ове технике у текућим и будућим индустријским пројектима.

# *Референце*


[1] Abraira, V., Pérez de Vargas, A., “Generalization of the Kappa Coefficient for Ordinal Categorical Data, Multiple Observers, and Incomplete Designs”, Qüestiió, Vol. 23, No. 3, pp. 561-571, 1999

[2] Baron, M., Girard, P., “SUIDT: A Task Model Based GUI-Builder,” *Proc. Task Models and Diagrams for User Interface Design (TAMODIA)*, July 2002, pp. 64-71

[3] Brabrand, Claus, Anders Møller, Mikkel Ricky, and Michael I. Schwartzbach. "Powerforms: Declarative client-side form field validation." World Wide Web, Vol. 3, No. 4, 2000, pp. 205-214.

[4] Brambilla, M., Comai, S., Fraternali, P., Matera, M., ”Designing Web Applications with WebML and WebRatio,” in G. Rossi, O. Pastor, D. Schwabe, L. Olsina (eds.), *Web Engineering: Modelling and Implementing Web Applications (Human-Computer Interaction Series)*, Springer, October 2007

[5] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J., “A Unifying Reference

Framework for Multi-Target User Interfaces”, *Interacting with Computers*, Vol.15, No. 3, June 2003, pp. 289-308

[6] Clarke, S., “Measuring API Usability”, Doctor Dobbs Journal, May 2004

[7] Da Cruz, A. M. R., Faria, J., P., “A Metamodel-Based Approach for Automatic User Interface Generation,” *Proc. 13th ACM/IEEE Int’l Conf. Model-Driven Engineering Languages and Systems (MODELS)*, October 2010, pp. 256-270

[8] Da Silva, P. P., “User Interface Declarative Models and Development Environments: A Survey,” Proc. Interactive Systems: Design, Specification, and Verification, June 2000

[9] Da Silva, P. P., Paton, N. W., "User Interface Modeling in UMLi," *IEEE Software*, Vol. 20, No. 4, July/Aug. 2003, pp. 62-69

[10] Gamma, E., Helm, R., Johnson, R., Vlissides, J, Design Patterns. Addison-Wesley Longman, 1995.

[11] Goderis, S., “On the Separation of User Interface Concerns,” Ph.D Thesis, Vrije Universiteit Brussel, Brussels, 2008

[12] Groenewegen, D., M., Visser, E., “Integration of data validation and user interface concerns in a DSL for web applications“, Software & Systems Modeling, Vol. 12, No. 1, pp. 35-52, 2013

[13] Jia, X., Steele, A., Qin, L., Liu, H., Jones, C., "Executable Visual Software Modeling – the ZOOM Approach," *Software Quality Journal*, Vol. 15, No. 1, 2007, pp. 27-51

[14] Henning, M., "API Design Matters", ACM QUEUE, Vol. 5, Issue 4, May/June 2007

[15] Holwerda, J., "Separation of Concerns in Web User Interface Design", Master Thesis, Faculty EEMCS, Delft University, Netherlands, 2008

[16] Jacobs, C., Li, W., Schrier, E., Bargeron, D., & Salesin, D. "Adaptive Document Layout", Communications of the ACM, Vol. 47 No. 8, 2004, pp. 60-66

[17] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., & Völkel, S. "Design Guidelines for Domain Specific Languages", In The 9th OOPSLA workshop on domain-specific modeling, pp. 7-13, October 2009

[18] Kovacevic, S., "UML and User Interface Modeling," Proc. 1st Int'l Workshop on The Unified Modeling Language (UML '98), 1998, pp. 253-266

[19] Kulkarni, V., Reddy, S, "Separation of Concerns in Model-Driven Development", IEEE Software, Vol.20, No. 5, September/October 2003, pp. 64-69

[20] Landis, J., Richard, and Gary G. Koch. "The measurement of observer agreement for categorical data." Biometrics, Vol. 33, No. 1, pp. 159-174, 1977

[21] Lecolinet, E., "A Molecular Architecture for Creating Advanced GUIs", Proc. 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03), 2003

[22] Lutteroth, C., Strandh, R., Weber, G., "Domain Specific High-Level Constraints for User Interface Layout", Constraints Journal, Vol. 13, Issue 3, September 2008

[23] Meixner, G., Paterno, F., Vanderdonckt, J., "Past, Present, and Future of Model-Based User Interface

Development", i-com, Vol. 10, No. 3, pp. 2-11, 2011

[24] Mijailović, Ž., Milićev, D., "A Retrospective on User Interface Development Technology", IEEE Software, Vol. 30, No. 6, pp. 76-83, November / December 2013.

[25] Milićev, D., "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments," IEEE Trans. Software Eng., vol. 28, no. 4, pp. 413-431, Apr. 2002.

[26] Milićev, D., "Model-Driven Development with Executable UML", John Wiley & Sons (WROX), 2009

[27] Milicev, D., "Towards Understanding of Classes versus Data Types in Conceptual Modeling and UML," Computer Science and Information Systems, vol. 9, no. 2, pp. 506-538, June 2012.

[28] Milićev, D., Mijailović, Ž., "Capsule-Based User Interface Modeling for Large-Scale Applications", IEEE Transactions on Software Engineering, Vol. 39, No. 9., pp. 1190-1207, September 2013.

[29] Milosavljević, G., Dejanović, I., Perišić, B., Milosavljević, B., "UML Profile for Specifying User Interfaces of Business Applications," *Computer Science and Information Systems*, Vol. 8, No. 2, May 2011, pp. 405-426

[30] Molina, P. J., Meliá, S., Pastor, O., "Just-UI: A User Interface Specification Model", *Proc. CADUI 2002*, pp. 63-74

[31] Monteiro, M., Oliveira, P., Gonçalves, R., "GUI Generation Based on Language Extensions: A Model to Generate GUI, based on Source Code with Custom Attributes," *Proc. ICEIS*, June 2008

[32] Mori, G., Paterno, F., Santoro, C., "CTTE: Support for Developing and Analyzing Task Models for Interactive Systems Design", IEEE Transactions on Software Engineering, Vol. 28, Issue 8, August 2002, pp. 797-813

[33] Myers, B. A., Hudson, S. E., Pausch, R., "Past, Present and Future of User Interface Software Tools," ACM

Transactions on Computer-Human Interaction (TOCHI), Vol. 7, No. 1, March 2000

[34] Myers, B., A., Rosson, M., B., "Survey on User Interface Programming", CHI '92 Proceedings of the SIGCHI conference on Human factors in computing systems, 1992

[35] Nichols, J., Faulring, A., "Automatic Interface Generation and Future User Interface Tools," ACM SIGCHI

Workshop on the Future of User Interface Design Tools, 2005

[36] Nokia Qt, http://qt.nokia.com/

[37] Object Management Group, UML 2.2 Superstructure Specification, http://www.omg.org, Feb. 2009.

[38] Paterno, F., Santoro, C., "A Logical Framework for Multi-Device User Interfaces", *Proc. 4th ACM SIGCHI*

*Symposium on Engineering Interactive Computing Systems (EICS '12)*, 2012, pp. 45-50

[39] Paterno, F., Santoro, C., Spano, L., D., "MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments", *ACM Transactions on Computer-Human Interaction (TOCHI)*, Vol. 16, Issue 4, November 2009, article No. 19

[40] Pitkänen, A., Hickey, S., "Implementation of Model-based User Interfaces for Pictorial Communication Systems Using UsiXML and Flex", *Proc. 1st Int'l Workshop USer Interface eXtensible Markup Language (UsiXML)*, 2010, pp. 65-74

[41] Scaffidi, C., Myers, B., Shaw, M., "Topes: Reusable Abstractions for Validating Data", In Proceedings of the 30th international conference on Software engineering, ACM. 2008, pp. 1-10

[42] Scoditti, A., Stuerzlinger, W., "A New Layout Method for Graphical User Interfaces", IEEE Toronto International Conference on Science and Technology for Humanity (TIC-STH '09), pp. 643-647, September 2009

[43] Selic, B., "The Pragmatics of Model-Driven Development," IEEE Software, vol. 20, no. 5, pp. 19-25, Sept./Oct. 2003.

[44] Selic, B, Gullekson, G., Ward, P.T., Real-Time Object-Oriented Modeling. John Wiley and Sons, 1994.

[45] Shull, F., "Designing a World at Your Fingertips: A Look at Mobile User Interfaces", IEEE Software, Vol. 29, No. 4, July/Aug. 2012, pp. 4-7

[46] Sinnig, D., Gaffar, A., Seffah, A., Forbrig, P., "Patterns, Tools and Models for Interaction Design", Workshop on Making Model-based UI Design Practical:Usable and Open Methods and Tools, (IUI-CADUI '04), 2004

[47] SOLoist framework, http://www.soloist4uml.com/

[48] Stylos, J., Clarke, S., "Usability Implications of Requiring Parameters in Objects' Constructors", Proc. 29th International Conference on Software Engineering (ICSE '07), 2007

[49] Tulach, J., "Practical API Design: Confessions of a Java Framework Architect", Apress, 2008

[50] Woods, E., Emery, D., Selic, B., "Point/Counterpoint," IEEE Software, vol. 27, no. 6, pp. 54-57, Nov./Dec. 2010.

[51] Xie, Q., Grechanik, M., Fu, C., "Guide: A GUI Differentiator," International Conference on Software Maintenance – ICSM, pp. 395-396, 2009

[52] Yuan, X., Memon, A., M., “Alternating GUI Test Generation and Execution,” TAIC PART '08. Testing: Academic & Industrial Conference, 2008, pp. 23-32

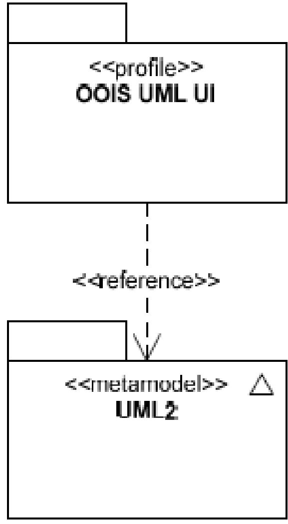
# Прилози

# *Профил језика UML за моделовање корисничких интерфејса*

Концепти описани у тези овде ће бити представљени на формалан начин у виду спецификације профила језика *UML*. Спецификација је у складу са документом *UML Superstructure Specification 2.2* [37].

## Оквир

Профил језика *UML* који је предмет ове спецификације референцира мета-модел језика *UML* (слика 54).

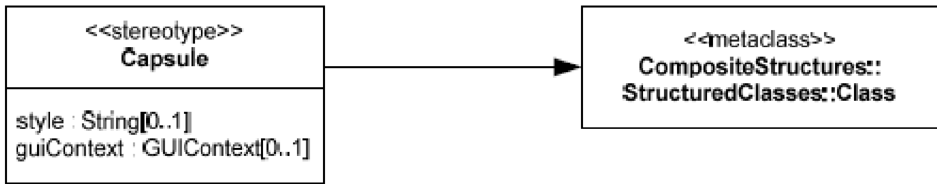



Слика 54.

## Капсула, интерна структура и конструктор

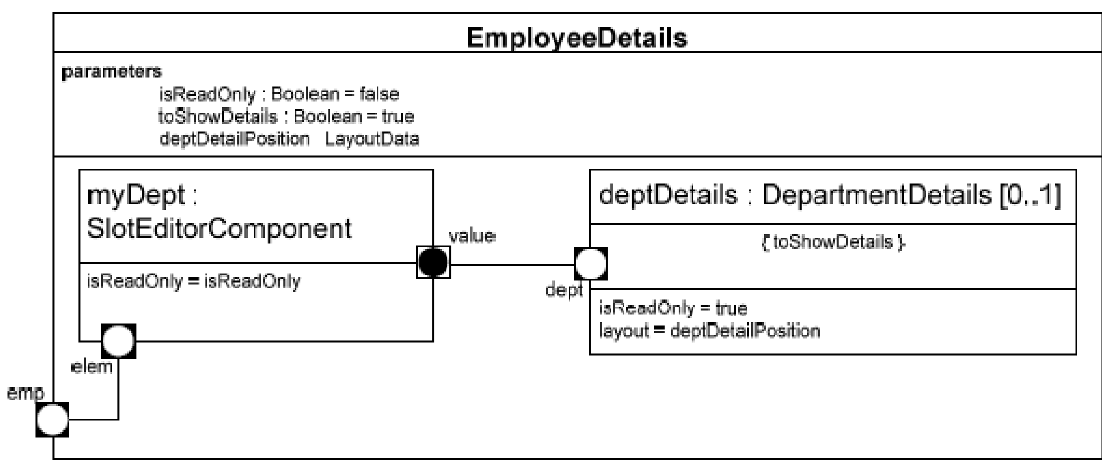
Капсула је опциони стереотип (енгл. *stereotype*) који се може доделити класи (енгл. *Class*) (из пакета *CompositeStructures::StructuredClasses*) као што је приказано на слици 55. Капсула има опциони стил презентације који се чува у атрибуту *style*. Он дефинише форматирање и визуелни изглед капсуле корисничког интерфејса као што су боје, фонтови,

ивице, маргине итд. Капсула такође има и опциони контекст који се чува атрибуту *guiContext* и који дефинише презентацију и понашање елемената корисничког интерфејса који су садржани у капсули и који директно одговарају инстанцама типова података [26]. Ако контекст није дефинисан у капсули, биће наслеђен од окружујућих капсула.

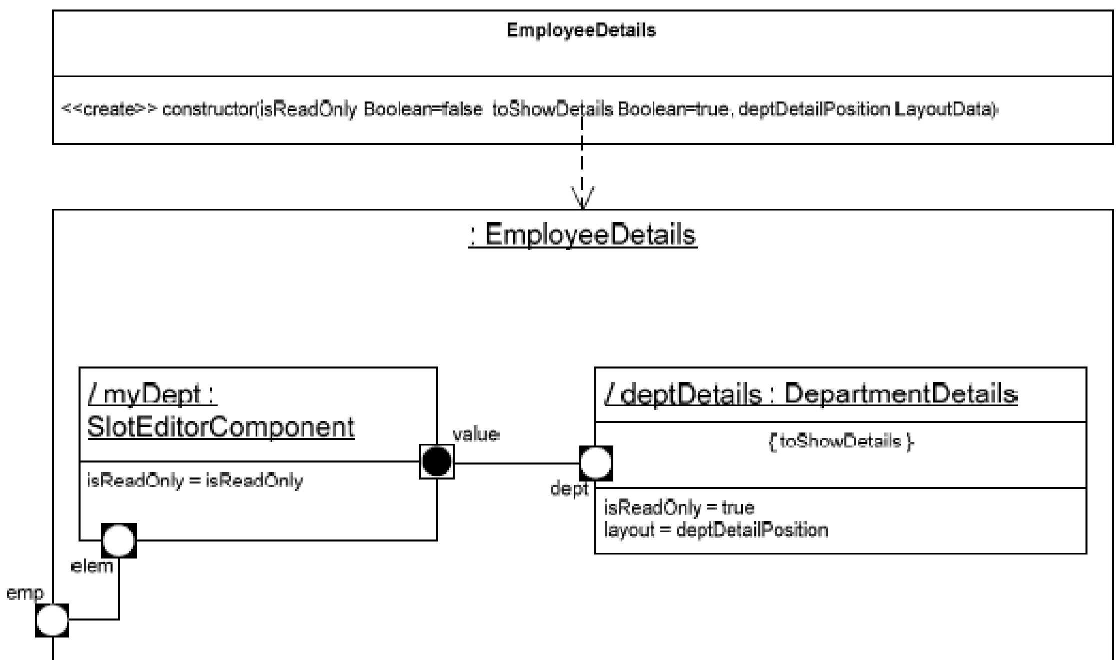



Слика 55.

Интерна структура капсуле, укључујући њене делове (референце), портове, жице, условна ограничења креирања и секцију са параметрима на дијаграму са слике 56 је само нотациона скраћеница дефиниције конструктора са датом креационом спецификацијом (као у профилу *OOIS UML* [26]) за спецификацију на језику *UML* приказану на слици 57. Спецификације инстанци, укључујући линкове (жице) имају семантику креационих спецификација у профилу *OOIS UML*.




Слика 56.




Слика 57.

*Додатна ограничења*

1. Сви атрибути капсуле, осим портова, морају бити или заштићени или приватни (енгл. *protected, private*, респективно):

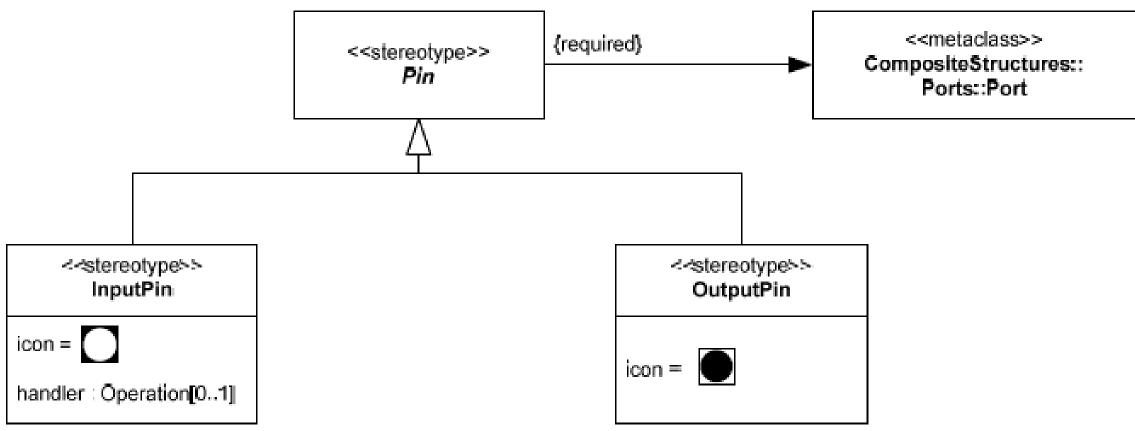
```
context Capsule:
self.base_Class.ownedAttribute ->
forAll(m | not m.oclIsKindOf(Port) implies
(m.visibility=protected or m.visibility=private ))
```

2. Све операције капсуле, осим конструктора, морају бити заштићене или приватне:

```
context Capsule:
self.base_Class.ownedOperation ->
forAll(m | m.extension$_create->size()=0 implies
(m.visibility=protected or m.visibility=private ))
```

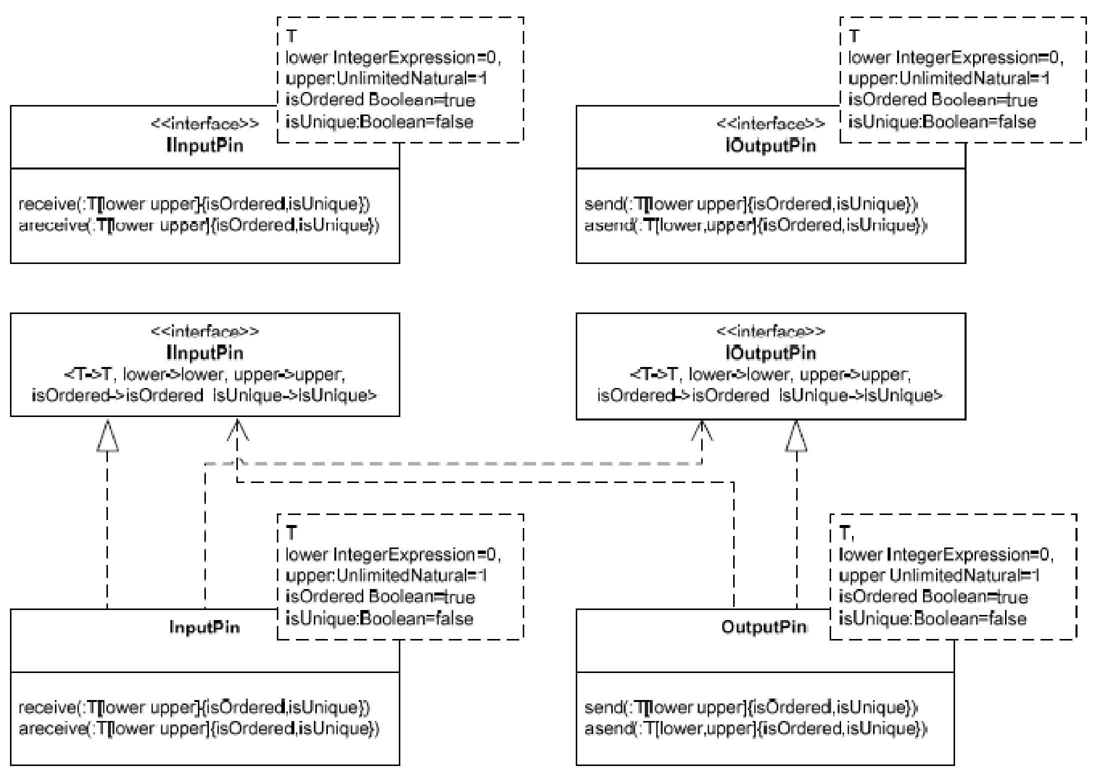
## Пинови

Пин је обавезан стереотип који се додељује порту језика *UML* (слика 58).




Слика 58.

Улазни пин (`<<inputPin>>`) је типизиран параметризованом шаблонском класом *InputPin*, док је излазни пин типизиран параметризованом шаблонском класом *OutputPin* као што је приказано на слици 59. Ове две класе су библиотечке класе овог профила. Шаблонски параметри дефинишу тип (T), мултипликативност, уређеност и јединственост колекције објеката која се транспортује преко датог пина.




Слика 59.

Класа *OutputPin* реализује параметризовани интерфејс *IOutputPin* (слика 59). Ово је једини интерфејс који нуди излазни пин. Он захтева интерфејс *IInputPin* кога нуде улазни пинови. Операција *send* служи за синхроно слање колекције објеката типа T преко излазног пина.

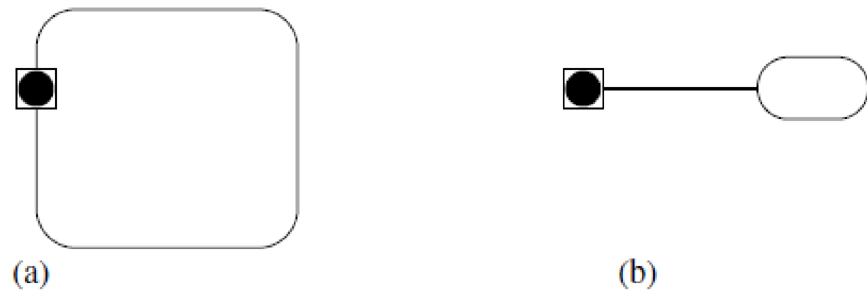
Операција *asend* служи за асинхроно слање колекције објеката типа T преко тог излазног пина.

Аналогно, класа *InputPin* реализује параметризовани интерфејс *IInputPin* (слика 59). Ово је једини интерфејс који нуди улазни пин. Он захтева интерфејс *IOutputPin* кога нуде излазни пинови. Операција *receive* се интерно позива када се колекција објеката типа T синхроно појави на улазном пину. Операција *areceive* се интерно позива када се колекција објеката типа T асинхроно појави на улазном пину.

*Пин интерфејса* је синоним за сервисни порт класе. *Интерни пин* је синоним за не-сервисни порт класе.

Атрибут *handler* стереотипа *InputPin* је опциона спецификација операције класе која садржи пин и која се позива да обради улазну поруку на том пину.

*Пин понашања* је нотациона скраћеница за интерни пин понашања. Нотација приказана на слици 60а је скраћеница за модел приказан на слици 60б.




Слика 60.

#### Додатна ограничења

3. Пин мора бити својство (енгл. *property*) или капсуле или понуђача сервиса (који ће бити дефинисан у наставку) али не и једног и другог истовремено:

```
context Pin:
self.base_Port.class.extension$_capsule->size()=1 xor
self.base_Port.class.extension$_serviceProvider->size()=1
```

4. Пин интерфејса мора имати недефинисану, јавну или видљивост на нивоу пакета:

```
context Pin:
self.base_Port.isService implies
(self.base_Port.visibility.size()=0 or self.base_Port.visibility=public or
self.base_Port.visibility=package)
```

5. Интерни пин мора имати недефинисану, приватну или заштићену видљивост:

```
context Pin:
not self.base_Port.isService implies
(self.base_Port.visibility.size()=0 or self.base_Port.visibility=private or
```

```
self.base_Port.visibility=protected)
```

6. Тип улазног пина мора бити класа *InputPin*:

```
context <<stereotype>>InputPin:
self.base_Port.type=InputPin
```

7. Тип излазног пина мора бити класа *OutputPin*:

```
context <<stereotype>>OutputPin:
self.base_Port.type=OutputPin
```

8. Само пин понашања може имати операцију *handler*:

```
context Pin:
self.handler->size()>0 implies self.base_Port.isBehavior
```

9. Операција *handler* датог пина мора бити спецификација операције класе која садржи тај пин:

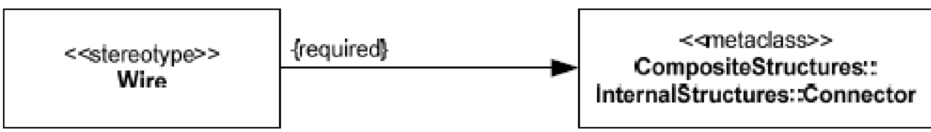
```
context Pin:
self.handler->size()>0 implies
self.base_Port.class->allFeatures()->includes(self.handler)
```

#### Додатне операције

1. `Pin::conformsTo(other:Pin):Boolean`: Да ли је пин у складу са другим датим пином? Усклађеност пинова се базира на типу пина (T), мултипликативности, уређености и јединствености параметризоване класе која типизира порт (*self.type*) [26].

### Жице

Жица је обавезни стереотип који се додељује конектору језика *UML* из пакета *CompositeStructures::InternalStructures* (слика 61).




Слика 61.

Жица повезује пинове у складу са основним правилима језика *UML* али и додатним правилима дефинисаним у табели 12. Ова табела дефинише да ли одређени пин *X* одређене врсте (улазни или излазни) може да се повеже жицом са пином *Y* одређене врсте. Лабела „понашање“ означава интерни пин понашања, док лабела „део“ означава пин дела (односно интерне капсуле) окружујуће капсуле. Лабела „-“ значи да одговарајућа веза пинова није дозвољена. Симбол „*X*→*Y*“ значи да је одговарајућа веза дозвољена и да поруке теку кроз ту везу од *X* ка *Y*. У овом случају, пин *X* се назива изворишни, а пин *Y* одредишни пин. Из

ове табеле се може закључити да се интерни пинови и пинови делова понашају на исти начин, односно као пинови интерне комуникације.

| пин *X* | | пин *Y* | | | | | |
|---|---|---|---|---|---|---|---|
| | | Улазни | | | излазни | | |
| | | интерфејс | понашање | део | интерфејс | понашање | део |
| улазни | интерфејс | - | $X \to Y$ | $X \to Y$ | $X \to Y$ | - | - |
| | понашање | $Y \to X$ | - | - | - | $Y \to X$ | $Y \to X$ |
| | део | $Y \to X$ | - | - | - | $Y \to X$ | $Y \to X$ |
| излазни | интерфејс | $Y \to X$ | - | - | - | $Y \to X$ | $Y \to X$ |
| | понашање | - | $X \to Y$ | $X \to Y$ | $X \to Y$ | - | - |
| | део | - | $X \to Y$ | $X \to Y$ | $X \to Y$ | - | - |

Табела 12.

### *Додатне операције*

1. `Wire::sourcePin():Pin[0..1]`, `Wire::destPin():Pin[0..1]`: Ове операције враћају изворишни и одредишни пин жице у складу са табелом 12.

### *Додатна ограничења*

1. Жица мора имати тачно два краја:

```
context Wire:
self.base_Connector.end->size()=2
```

2. Жица може повезивати само пинове:

```
context Wire:
self.base_Connector.end-> forEach(ce| ce.role.oclIsKindOf(Port) and
ce.role.oclAsType(Port).extension$_pin->size()=1)
```

3. Жица може повезивати пинове у складу са табелом 12:

```
context Wire:
self.sourcePin()->size()=1 and self.destPin()->size()=1
```

4. Жица може повезивати само компатибилне пинове:

```
context Wire:
self.sourcePin().conformsTo(self.destPin())
```

# Биографија

# *Кратка биографија*

Жарко Мијаиловић, дипломирани инжењер електротехнике, рођен је 12.07.1981. године у Шапцу, република Србија, од оца Радована и мајке Јасмине. Основну и средњу школу завршио је у Шапцу као један од најбољих ученика. Гимназију је такође завршио у Шапцу. У току школовања учествовао је на многобројним такмичењима из природних наука, од школских, до републичких и савезних. Поред природних наука показивао је интересовање за шах, али и за друге спортске активности. Електротехнички факултет у Београду уписао је 2000/2001. школске године. Дипломирао је 2006. године је са просечном оценом 8.50 током студија и оценом 10 на дипломском. Исте године, засновао је радни однос у ИТ сектору Комерцијалне банке, а.д. Београд као софтверски инжењер. Од новембра 2007. године до данас запослен је у предузећу СОЛ Софтвер д.о.о. Школске 2007/2008. године уписао је докторске студије на Електротехничком факултету у Београду. Област научног истраживања кандидата обухвата технологије и софтверске алате за развој инфромационих система, а посебно графичких корисничких интерфејса, као и развој софтвера и информационих система на основу извршивих UML модела.

## *Пројекти и професионалне активности*

| Датум | Предузеће | Радно место | Опис |
|---|---|---|---|
| тренутно | СОЛ Софтвер д.о.о. | пројектант софтвера, програмер | • Регистар новчаних казни и прекршајне евиденције Босне и Херцеговине<br>• Наручилац: Агенција за идентификациона документа, евиденцију и размјену података (IDDEEA)<br>• Oracle Database, Oracle Apex |
| 2013. | СОЛ Софтвер д.о.о. | пројектант софтвера, програмер | • Веб базирана берза послова транспорта људи и робе<br>• Наручилац: TransportBidder.com<br>• UML, SOLoist, Java, JSP, MySQL. |
| 2012.–2013. | СОЛ Софтвер д.о.о. | пројектант софтвера, програмер | • Систем за оцењивање студената заснован на учењу<br>• Наручилац: HiST (Норвешка)<br>• UML, SOLoist, Java, MySQL.<br>• Успешно инсталиран 2013. године |
| 2011.–2012. | СОЛ Софтвер д.о.о. | руководилац пројекта, пројектант софтвера, програмер | • Систем за односе са клијентима (енг. *CRM*) Наручилац: Industry Media Global (Норвешка)<br>• UML, SOLoist, Java, MySQL.<br>• Успешно инсталиран 2012. године |
| 2011. | СОЛ Софтвер д.о.о. | програмер | • Систем за за прву помоћ, апликација за мобилне уређаје, централна диспечерска веб апликација и административна веб апликација<br>• UML, SOLoist, Java, Android SDK, MySQL.<br>• Наручилац: Quality Management Software AS (Норвешка)<br>• Успешно инсталиран 2011. године |
| 2010.-2011. | СОЛ Софтвер д.о.о. | руководилац развојног тима, програмер | • Катастарски информациони систем<br>• Наручилац: Siemens, Београд.<br>• UML, SOLoist, Java, Oracle. |
| 2009. | СОЛ Софтвер | руководилац пројекта, | • Информациони систем Службе за управљање кадровима Републике Србије |

| | | | |
|---|---|---|---|
| | д.о.о. | пројектант софтвера, програмер | • Наручилац: MD&Profy, Београд.<br>• UML, SOLoist, Java, Sybase ASA.<br>• Успешно инсталиран 2010. године |
| 2009. | СОЛ Софтвер д.о.о. | програмер | • Информациони систем за праћење великих догађаја, скупова и конференција<br>• Наручилац: MD&Profy, Београд.<br>• UML, SOLoist, Java, MySQL.<br>• Успешно инсталиран 2009. године |
| 2009.-2011. | СОЛ Софтвер д.о.о. | пројектант софтвера, програмер | • Развој технологије за развој информационих система уз помоћ извршивих UML модела: SOLoist<br>• Java, GWT<br>• Успешно употребљен у неколицини индустријских пројеката и система |
| 2008.-2009. | СОЛ Софтвер д.о.о. | извођач | • Студија изводљивости дигитализације метеоролошких података Републичког хидрометеоролошког завода Србије<br>• Abbyy Form Reader |
| 2007.-2008. | ЕТФ | програмер | • Систем за прорачун и графички приказ јачине електричног поља<br>• Наручилац: Агенција за контролу летења Србије и Црне Горе.<br>• UML, SOLoist, C++, Qt, PostgreSQL.<br>• Успешно инсталиран 2008. године |
| 2006.-2007. | Комерцијалн а банка а.д. Београд | програмер | • Брокерска апликација за управљање берзанским налозима<br>• Java, JSP, Struts, Spring, IBM Rational Application Developer, IBM DB2<br>• Успешно инсталиран 2007. године |
| 2006. | Комерцијалн а банка а.д. Београд | програмер | • Систем - База знања<br>• Java, JSP, Struts, IBM Rational Application Developer, SQLServer<br>• Успешно инсталиран 2006- године |
| 2006. | Комерцијалн а банка а.д. Београд | програмер | • Апликација за миграцију података Агенције за привредне регистре Републике Србије<br>• Java, IBM DB2<br>• Успешно инсталиран 2006. године |

# *Списак објављених научних радова и учешћа на међународним конференцијама*

1. Mijailović, Ž, Milićev, D., "A Retrospective on User Interface Development Technology", *IEEE Software*, Vol. 30, No. 6, Nov. – Dec. 2013, pp. 76 – 83, DOI: 10.1109/MS.2013.45, ISSN: 0740-7459, IF: 1.616, 5-year IF: 1.584, M21
2. Milićev, D., Mijailović, Ž, "Capsule-Based User Interface Modeling for Large-Scale Applications“, *IEEE Transactions on Software Engineering*, Vol. 39, No. 9, Sept. 2013, pp. 1190 - 1207, DOI: 10.1109/TSE.2013.8, ISSN: 0098-5589, IF: 2.588, 5-year IF: 3.371, M21
3. Mijailović, Ž, Milićev, D., "Empirical Analysis of GUI Programming Concerns", послат за публиковање у *International Journal of Human Computer Studies (Elsevier)*, транутни статус: прихваћен уз захтев за мање измене (енг. *accept with minor revision*), ISSN: 1071-5819, IF: 1.415, 5-year IF: 2.003, M21
4. Mijailović, Ž, Luković, S., Milićev, D., "FastOQL – Fast Object Queries for Hibernate“, Међународна конференција *Devoxx 2012*, снимак доступан на сајту конференције: http://www.devoxx.com/display/DV12/Home

# Изјаве

# *Изјава о ауторству*

**Прилог 1.**

# Изјава о ауторству

Потписани-а Мијаиловић Жарко

број индекса 5060 / 2007

**Изјављујем**

да је докторска дисертација под насловом

Развој графичког корисничког интерфејса пословних апликација базиран на подели одговорности и објектном моделу података са UML семантиком

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

**Потпис докторанда**

У Београду, 4.8.2014. године

## *Изјава о истоветности штампане и електронске верзије докторског рада*

**Прилог 2.**

# Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора __Мијаиловић Жарко

Број индекса 5060 / 2007

Студијски програм Софтверско инжењерство

Наслов рада Развој графичког корисничког интерфејса пословних апликација базиран на подели одговорности и објектном моделу података са UML семантиком

Ментор проф. др Драган Милићев

Потписани/а Мијаиловић Жарко

Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

**Потпис докторанда**

У Београду, 8.4.2014. године

# *Изјава о коришћењу*

**Прилог 3.**

## Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић" да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Развој графичког корисничког интерфејса пословних апликација базиран на подели одговорности и објектном моделу података са UML семантиком

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ла.

1. Ауторство

2. Ауторство - некомерцијално

3. Ауторство – некомерцијално – без прераде

4. Ауторство – некомерцијално – делити под истим условима

5. Ауторство – без прераде

6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

**Потпис докторанда**

У Београду, 8.4.2014. године

1. Ауторство - Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. Ауторство – некомерцијално. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. Ауторство - некомерцијално – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

4. Ауторство - некомерцијално – делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. Ауторство – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. Ауторство - делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.