

УНИВЕРЗИТЕТ У БЕОГРАДУ
ФАКУЛТЕТ ОРГАНИЗАЦИОНИХ НАУКА
КАТЕДРА ЗА СОФТВЕРСКО ИНЖЕЊЕРСТВО

СЕМИНАРСКИ РАД
Софтверски процес

Тема:

Развој игре “Потапање бродова” применом
Ларманове методе

Ментор:
проф. др Синиша Влајић

Студент:
Милош Никић 3710/2019

Београд, 2020. године

Садржај

1. Кориснички захтеви.....	3
1.1. Вербални опис.....	3
1.2. Случајеви коришћења.....	4
1.2.1. СК1: Случај коришћења - Пријављивање корисника.....	5
1.2.2. СК2: Случај коришћења - Креирање игре.....	6
2. Анализа.....	11
2.1. Понашање софтверског система - Системски дијаграм секвенци.....	11
2.1.1. Дијаграм секвенци случаја коришћења 1 - Пријављивање играча.....	11
2.1.2. Дијаграм секвенци случаја коришћења 2 - Креирање игре.....	12
2.1.3. Дијаграм секвенци случаја коришћења 3 - Започињање игре.....	14
2.1.4. Дијаграм секвенци случаја коришћења 4 - Погађање поља.....	15
2.1.5. Дијаграм секвенци случаја коришћења 5 - Приказивање ранг листе.....	17
2.1.6. Дијаграм секвенци случаја коришћења 6 - Крај игре.....	18
2.2. Понашање софтверског система - Дефинисање уговора о системским операцијама.....	21
2.3. Структура софтверског система - Концептуални модел.....	23
2.4. Структура софтверског система - Релациони модел.....	23
3. Пројектовање.....	25
3.1. Архитектура софтверског система.....	25
3.2. Пројектовање корисничког интерфејса.....	25
3.2.1. СК1: Случај коришћења - Пријављивање играча.....	27
3.2.2. СК2: Случај коришћења - Креирање игре.....	29
3.2.3. СК3: Случај коришћења - Започињање игре.....	31
3.2.4. СК4: Случај коришћења - Погађање поља.....	34
3.2.5. СК5: Случај коришћења - Приказивање ранг листе.....	36
3.2.6. СК6: Случај коришћења - Прекид игре.....	38
3.3. Комуникација сервер - клијент.....	40
3.4. Контролер апликационе логике.....	43
3.5. Пројектовање структуре софтверског система - Доменске класе.....	45
3.6. Пројектовање понашања софтверског система.....	49
3.7. Пројектовање брокера базе података.....	53
3.8. Пројектовање складишта података.....	55

4. Имплементација.....	56
4.1. Механизам рефлексije.....	59
4.2. <i>Generics</i> механизам.....	60
5. Тестирање.....	61
6. Закључак.....	61
7. Принципи, методе и стратегије пројектовања софтверског система.....	62
7.1. Принципи пројектовања софтверског система.....	62
7.1.1. Апстракција.....	62
7.2. Стратегије пројектовања.....	73
7.3. Методе пројектовања.....	75
8. Примена патерна у пројектовању.....	81
8.1 Увод у патерне.....	81
8.2 Општи облик GOF патерна пројектовања.....	82
9. Литература.....	91

1. Кориснички захтеви

1.1. Вербални опис

Потребно је развити софтверску апликацију која врши симулацију игре “Потапање бродова”, где корисник игра против рачунара. Учеснику у игри је потребно омогућити пријављивање на систем уношењем корисничког имена и лозинке. Када се корисник улогује на систем потребно је омогућити започињање нове игре.

Када корисник изабере креирање нове игре, неопходно је обезбедити могућност постављања бродова на поља табле. Неопходно је обезбедити правилан распоред бродова на табли (без додиривања и у границама табле која има десет редова и исто толико колона). Након завршетка постављања свих бродова, кориснику је потребно омогућити потврду изабране формације. Када корисник потврди своју формацију, неопходно је обавестити корисника да је успешно потврђена формација и да игра може да почне.

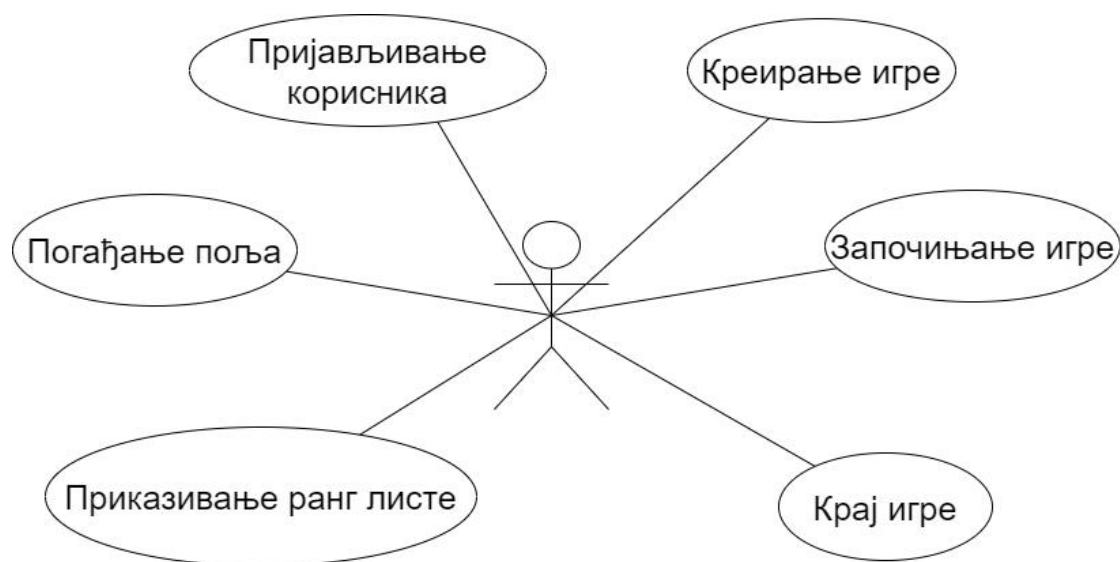
Такође, потребно је омогућити “гађање”, тј. одабир поља, противничке табле. Потребно је погођене бродове обележити зеленом бојом, а промашена поља црном бојом. Када корисник погоди брод, неопходно је омогућити да корисник настави са гађањем а да рачунар сачека на свој ред.

Омогућити да се игра заврши и прогласи победник након последњег потопљеног брода.

1.2. Случајеви коришћења

У конкретном случају су идентификовани следећи случајеви коришћења:

1. Пријављивање корисника
2. Креирање игре
3. Започињање игре
4. Погађање поља
5. Приказивање ранг листе
6. Крај игре



Слика 1 Дијаграм случаја коришћења

1.2.1. СК1: Случај коришћења - Пријављивање корисника

Назив СК

Пријављивање корисника

Актори СК

Корисник

Учесници СК

Корисници систем (програм)

Предуслов: Систем је укључен и приказује форму за пријављивање играча.

Основни сценарио СК:

1. **Корисник** уноси податке за пријављивање. (АПУСО)
2. **Корисник** контролише да ли је коректно унео податке. (АНСО)
3. **Корисник** позива систем да изврши пријављивање играча. (АПСО)
4. **Систем** врши пријављивање играча. (СО)
5. **Систем** приказује играчу поруку: “Uspesno ste se prijavili!!” и омогућава приступ систему. (ИА)

Алтернативна сценарија:

- 5.1. Уколико систем не пронађе играча са унетим подацима, приказује играчу поруку: “Neuspesno prijavljivanje!”. (ИА)

1.2.2. СК2: Случај коришћења - Креирање игре

Назив СК

Креирање игре

Актори СК

Играч

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен. Корисник је пријављен на систем.

Основни сценарио СК:

1. **Корисник** позива систем да изврши креирање игре. (АПСО)
2. **Систем** врши креирање игре. (СО)
3. **Систем** приказује играчу поруку: "Počnite sa postavljanjem brodova!". (ИА)

Алтернативна сценарија:

- 3.1. Уколико систем не може да креира игру, приказује поруку: "Nije moguće kreirati igru!". (ИА)

1.2.3. СКЗ: Случај коришћења - Започињање игре

Назив СК

Започињање игре

Актори СК

Играч

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен. Корисник је пријављен на систем и игра је успешно креирана.

Основни сценарио СК:

1. **Корисник** уноси податке потребне за започињање игре. (АПУСО)
2. **Корисник** контролише да ли је коректно унео податке. (АНСО)
3. **Корисник** позива систем да започне игру. (АПСО)
4. **Систем** врши започињање игре. (СО)
5. **Систем** приказује играчу поруку: "Protivnik je izabrao formaciju, igra moze da pocne!". (ИА)

Алтернативна сценарија:

5.1 Уколико систем не може да започне игру, приказује поруку: "Problem prilikom startovanja igre!". (ИА)

1.2.4. СК4: Случај коришћења - Погађање поља

Назив СК

Погађање поља

Актори СК

Играч

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен. Корисник је пријављен на систем и игра је успешно креирана и започета.

Основни сценарио СК:

1. **Корисник** бира поље које жели да гађа. (АПУСО)
2. **Корисник** контролише да ли је коректно унео податке. (АНСО)
3. **Корисник** позива систем да погађа поље. (АПСО)
4. **Систем** врши погађање поља. (СО)
5. **Систем** приказује играчу поруку: "Korisnik je gadjaо polje sa traženim koordinatama". (ИА)

Алтернативна сценарија:

- 5.1. Уколико систем не може да погађа поље, приказује поруку: "Greska prilikom gadjanja od strane korisnika!". (ИА)

1.2.5. СК5: Случај коришћења - Приказивање ранг листе

Назив СК

Приказивање ранг листе

Актори СК

Играч

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен и приказује форму за пријављивање играча. Корисник је пријављен на систем.

Основни сценарио СК:

1. **Корисник** позива систем да прикаже ранг листу. (АПСО)
2. **Систем** врши приказивање ранг листе. (СО)
3. **Систем** приказује играчу ранг листу. (ИА)

Алтернативна сценарија:

- 3.1. Уколико систем не може да прикаже ранг листу, приказује поруку: "Problem prilikom dovlacenja podataka o rang listi.". (ИА)

1.2.6. СК6: Случај коришћења - Крај игре

Назив СК

Крај игре

Актори СК

Играч

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен и приказује форму за пријављивање играча. Корисник је пријављен на систем и игра је успешно креирана.

Основни сценарио СК:

1. **Корисник** позива систем да заврши игру. (АПСО)
2. **Систем** врши крај игре. (СО)
3. **Систем** приказује играчу поруку о исходу игре. (ИА)

Алтернативна сценарија:

- 3.1. Уколико систем не може да прикаже ранг листу, приказује поруку: "Greska prilikom završavanja igre!". (ИА)

2. Анализа

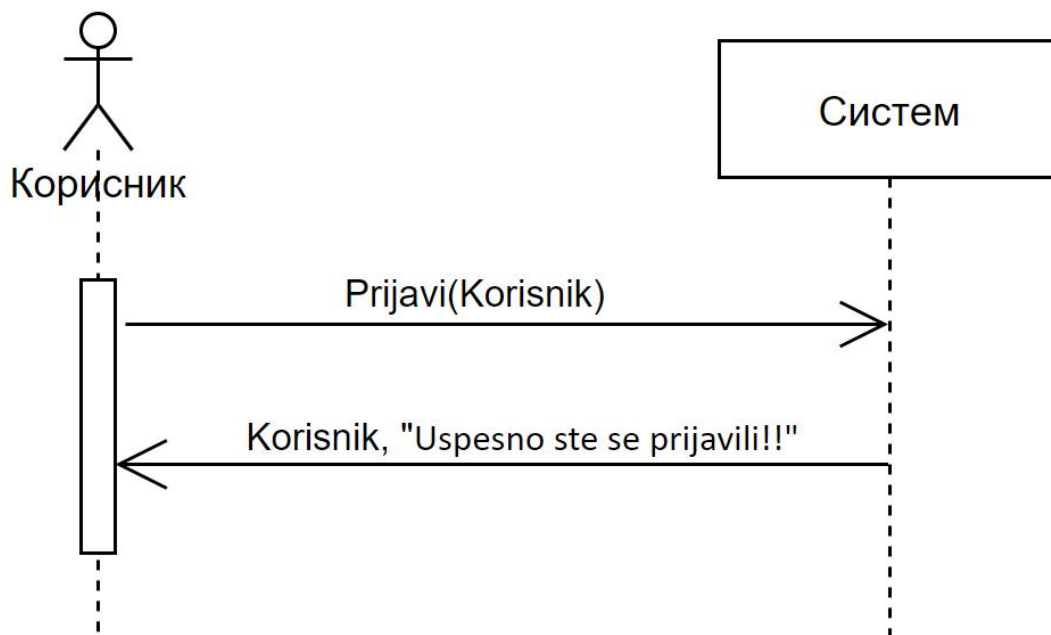
Фаза анализе описује логичку структуру и понашање софтверског система (пословну логику софтверског система). Понашање софтверског система је описано помоћу системских дијаграма секвенци и преко системских операција. Структура софтверског система се описује помоћу концептуалног и релационог модела.

2.1. Понашање софтверског система - Системски дијаграм секвенци

2.1.1. Дијаграм секвенци случаја коришћења 1 - Пријављивање играча

Основни сценарио СК:

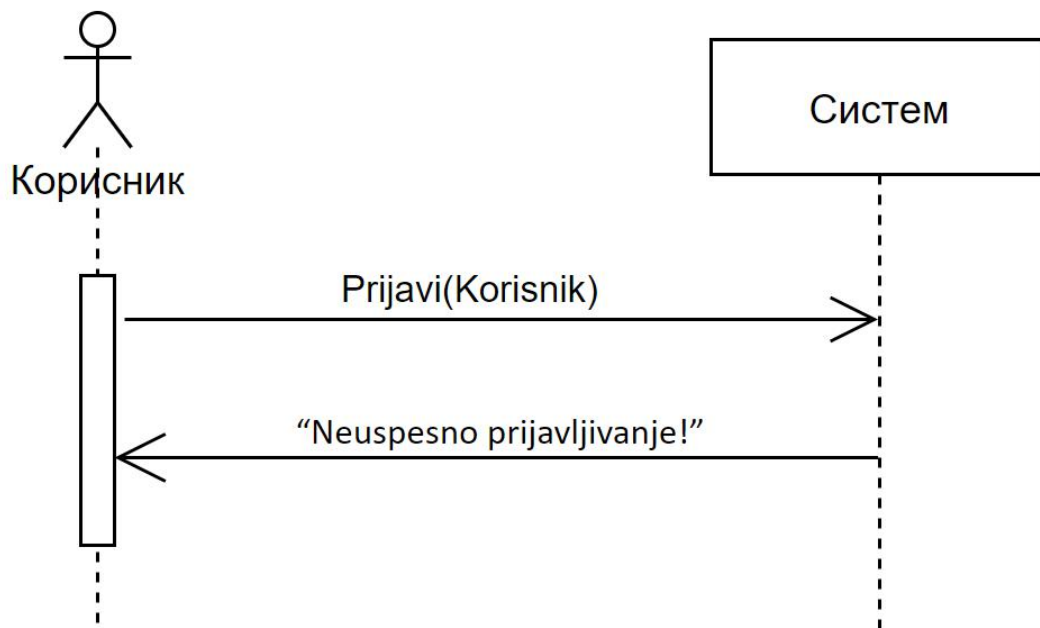
1. **Корисник** позива систем да изврши пријављивање играча. (АПСО)
2. **Систем** приказује играчу поруку: "Uspesno ste se prijavili!!" и омогућава приступ систему. (ИА)



Слика 2 Основни сценарио СК1

Алтернативна сценарија:

- 2.1. Уколико систем не пронађе играча са унетим подацима, приказује играчу поруку: "Neuspesno prijavljivanje!". (ИА)



Слика 3 Први алтернативни сценарио СК1

Са наведених секвенчних дијаграма уочава се једна системска операција коју треба пројектовати:

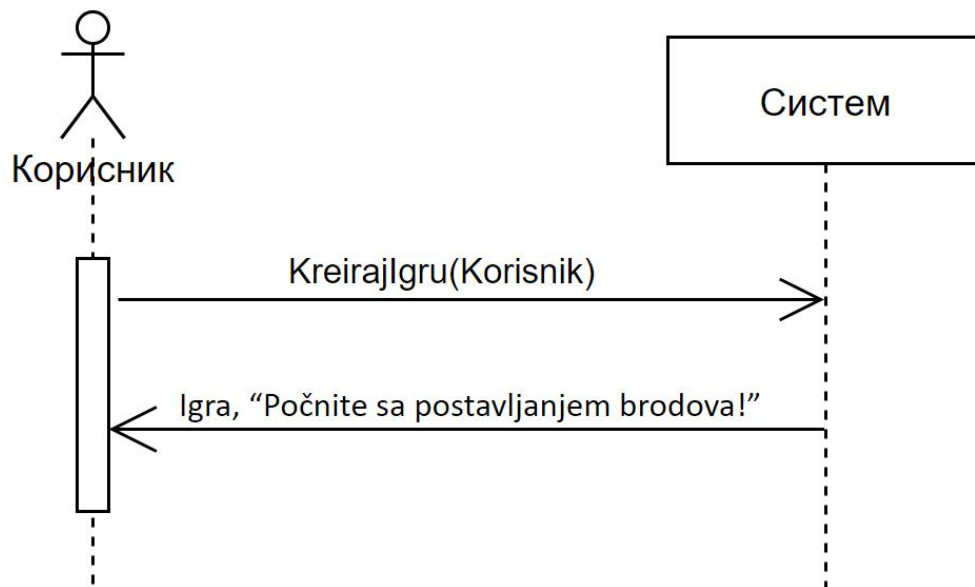
1. Сигнал `Prijavi(Korisnik)`

2.1.2. Дијаграм секвенци случаја коришћења 2 - Креирање игре

Основни сценарио СК:

1. **Корисник** позива систем да изврши креирање игре. (АПСО)

2. **Систем** приказује играчу поруку: "Počnite sa postavljanjem brodova!". (ИА)



Слика 4 Основни сценарио СК2

Алтернативна сценарија:

2.1. Уколико систем не може да креира игру, приказује поруку: "Nije moguće kreirati igru!". (ИА)



Слика 5 Први алтернативни сценарио СК2

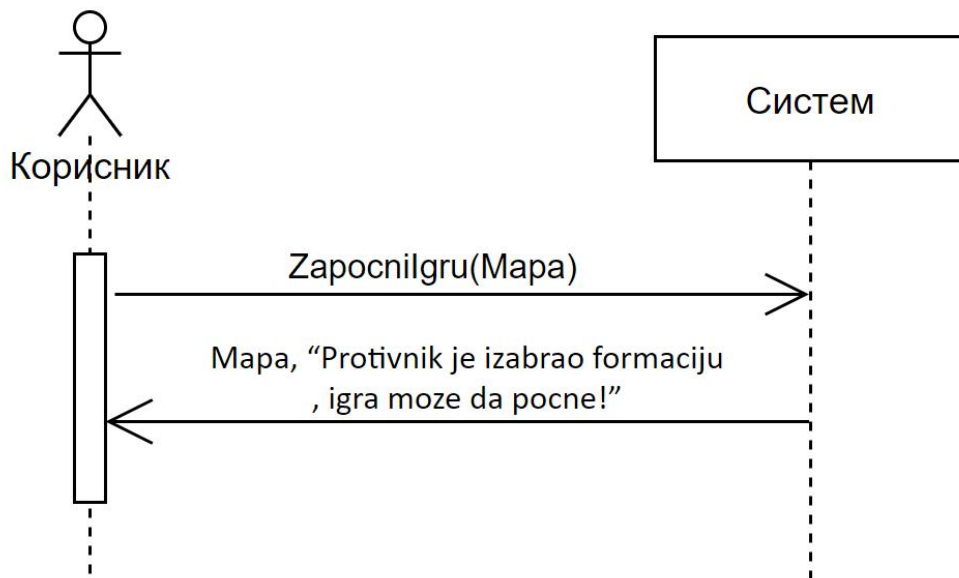
Са наведених секвенцих дијаграма уочава се једна системска операција коју треба пројектовати:

1. Сигнал KreirajIgru(Korisnik)

2.1.3. Дијаграм секвенци случаја коришћења 3 - Започињање игре

Основни сценарио СК:

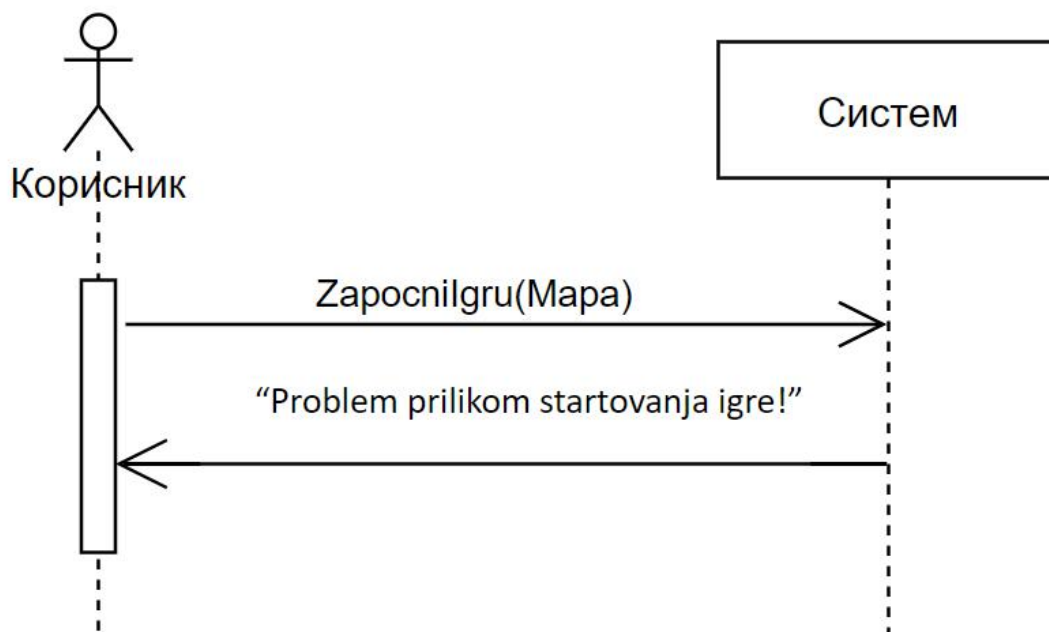
1. **Корисник** позива систем да започне игру. (АПСО)
2. **Систем** приказује играчу поруку: "Protivnik je izabrao formaciju, igra moze da pocne!". (ИА)



Слика 6 Основни сценарио СК3

Алтернативна сценарија:

- 2.1. Уколико систем не може да започне игру, приказује поруку: "Problem prilikom startovanja igre!". (ИА)



Слика 7 Први алтернативни сценарио СК3

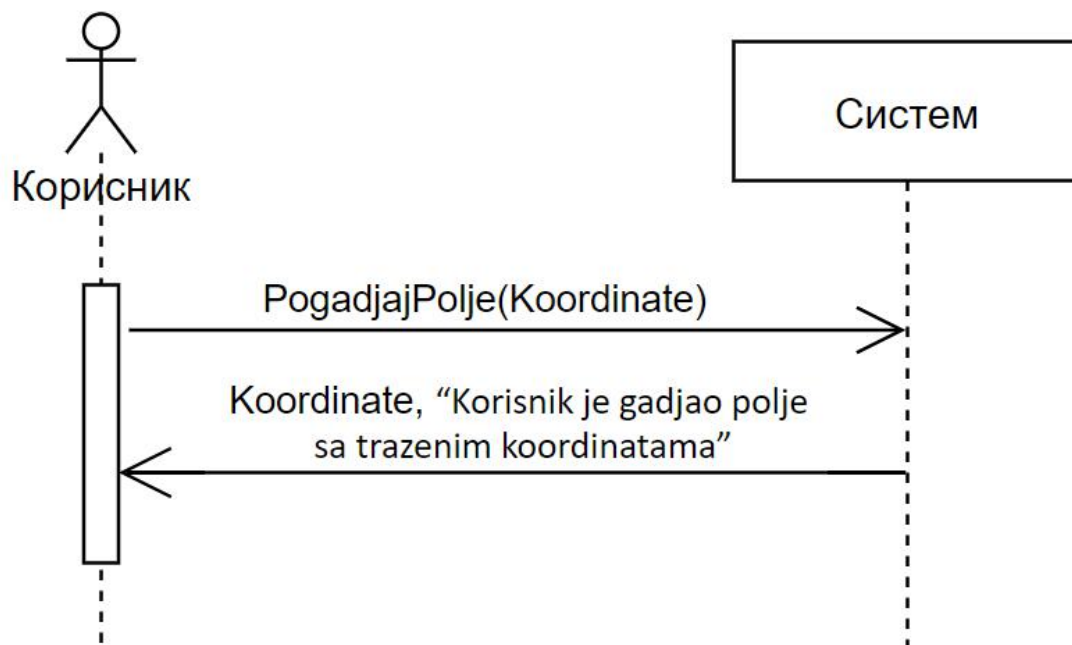
Са наведених секвенчних дијаграма уочава се једна системска операција коју треба пројектовати:

1. Сигнал ZapocniIgru(Mapa)

2.1.4. Дијаграм секвенци случаја коришћења 4 - Погађање поља

Основни сценарио СК:

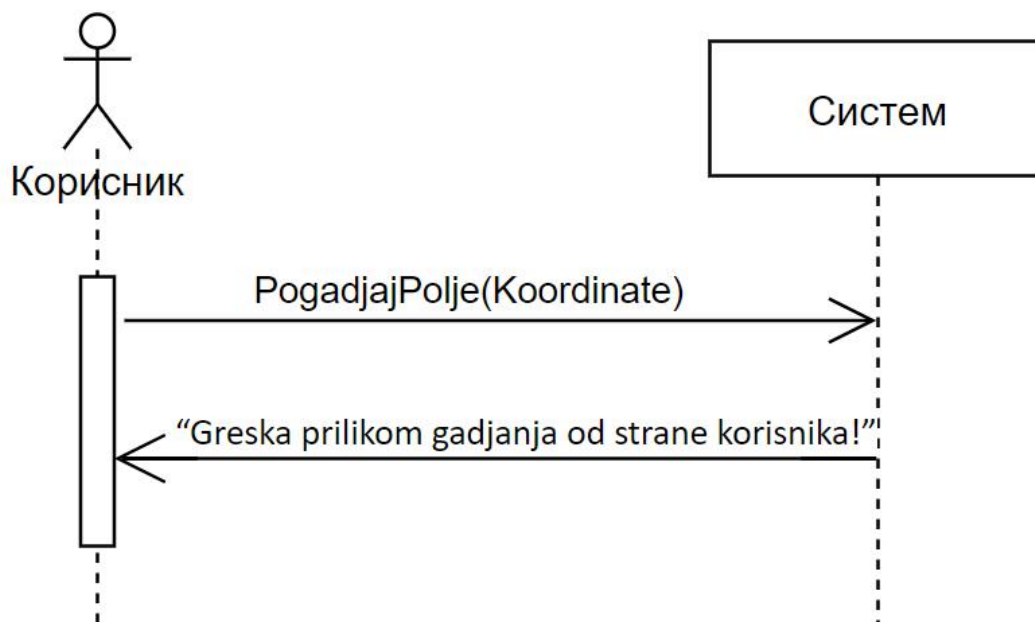
1. **Корисник** позива систем да погађа поље. (АПСО)
2. **Систем** приказује играчу поруку: "Korisnik je gadjao polje sa trazanim koordinatama". (ИА)



Слика 8 Основни сценарио СК4

Алтернативна сценарија:

2.1. Уколико систем не може да погађа поље, приказује поруку: "Greska prilikom gadjanja od strane korisnika!". (ИА)



Слика 9 Први алтернативни сценарио СК4

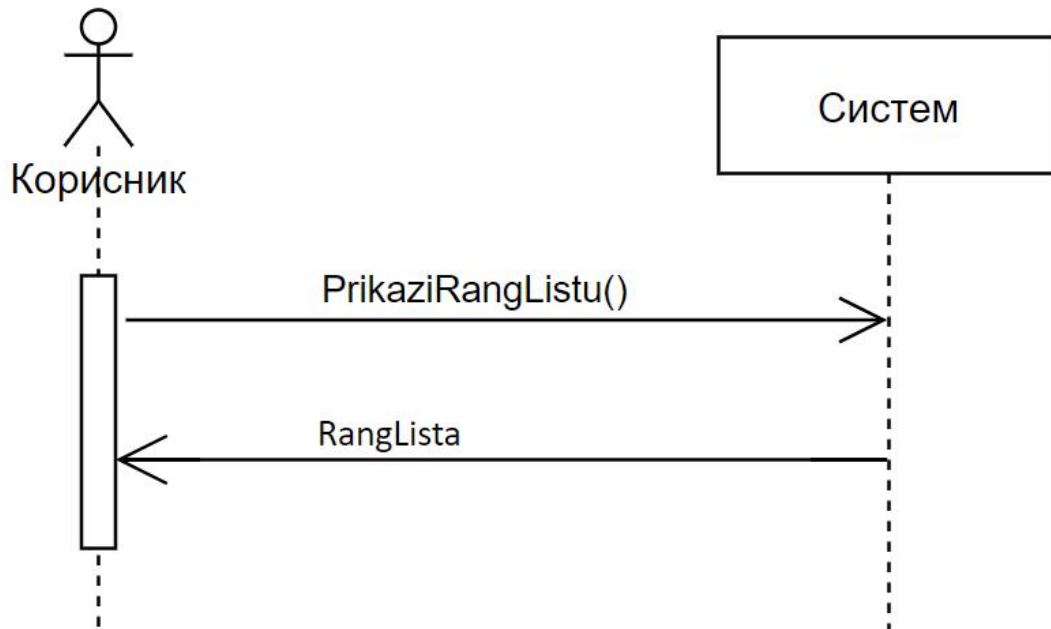
Са наведених секвенцих дијаграма уочава се једна системска операција коју треба пројектовати:

1. Сигнал PogadjajPolje(Koordinate)

2.1.5. Дијаграм секвенци случаја коришћења 5 - Приказивање ранг листе

Основни сценарио СК:

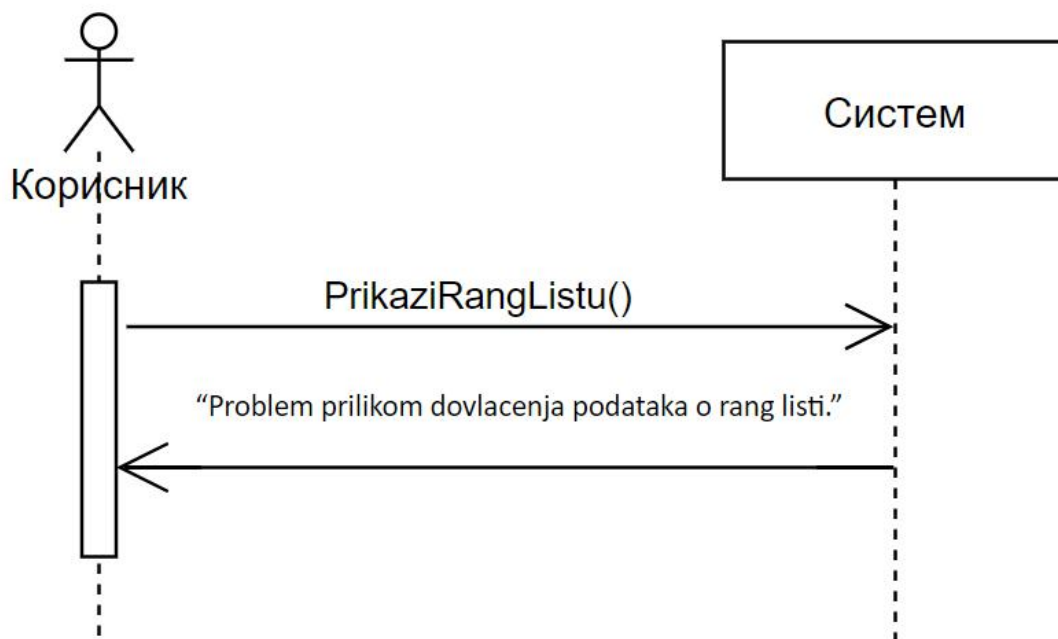
1. **Корисник** позива систем да прикаже ранг листу. (АПСО)
2. **Систем** приказује играчу ранг листу. (ИА)



Слика 10 Основни сценарио СК5

Алтернативна сценарија:

- 2.1. Уколико систем не може да прикаже ранг листу, приказује поруку: "Problem prilikom dovlacenja podataka o rang listi.". (ИА)



Слика 11 Први алтернативни сценарио СК5

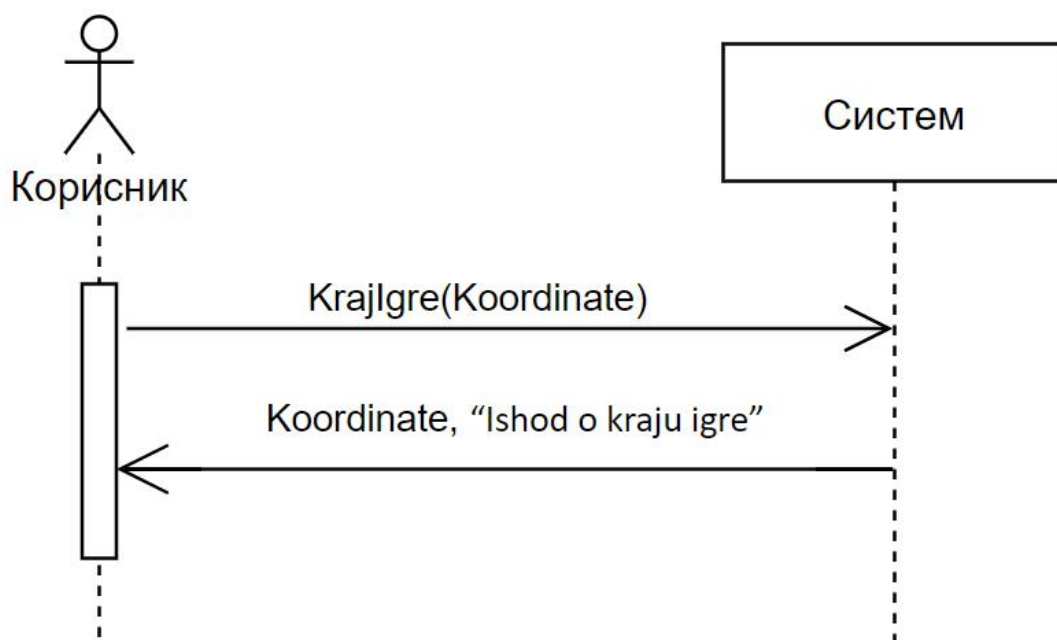
Са наведених секвенчних дијаграма уочава се једна системска операција коју треба пројектовати:

1. Сигнал PrikaziRangListu()

2.1.6. Дијаграм секвенци случаја коришћења 6 - Крај игре

Основни сценарио СК:

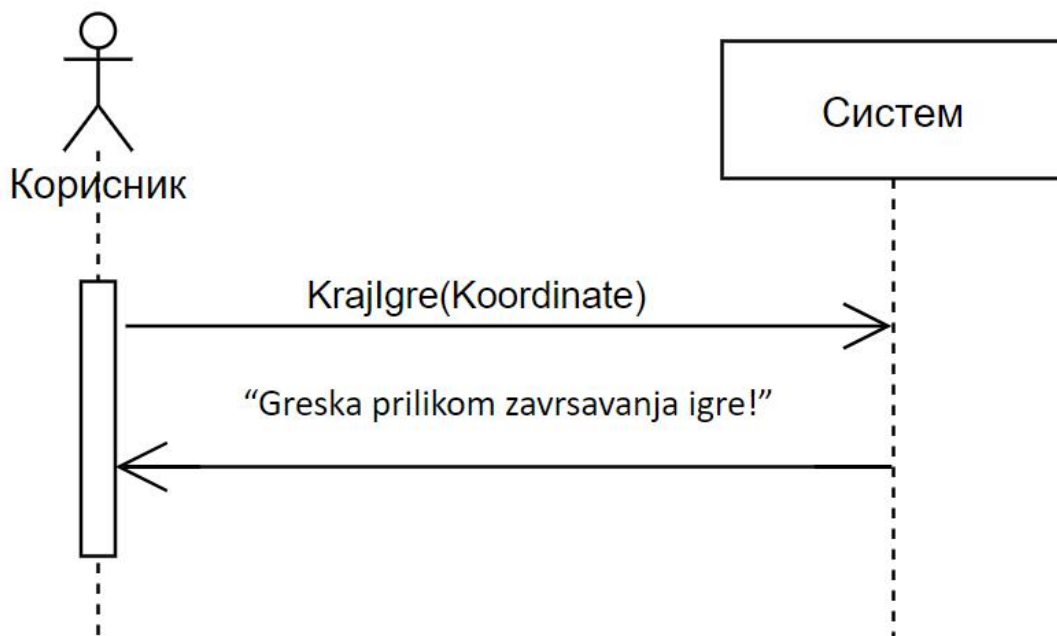
1. **Корисник** позива систем да заврши игру. (АПСО)
2. **Систем** приказује играчу поруку о исходу игре. (ИА)



Слика 12 Основни сценарио СК6

Алтернативна сценарија:

2.1. Уколико систем не може да прикаже ранг листу, приказује поруку: "Greska prilikom završavanja igre!". (ИА)



Слика 13 Први алтернативни сценарио СК6

Са наведених секвенцих дијаграм уочава се једна системска операција коју треба пројектова:

1. Сигнал KrajIgre(Koordinate)

Након анализе сценарија случајева коришћења уочено је 6 системских операција које треба пројектовати:

1. Сигнал Prijavi(Korisnik)
2. Сигнал KreirajIgru(Korisnik)
3. Сигнал ZapocniIgru(Koordinate)
4. Сигнал PogadjajPolje(Koordinate)
5. Сигнал PrikaziRangListu()
6. Сигнал KrajIgre(Koordinate)

2.2. Понашање софверског система - Дефинисање уговора о системским операцијама

Уговор УГ1: *Prijavi*

Операција: *Prijavi(Korisnik)*: сигнал

Веза са СК: СК1

Предуслови: /

Постуслови: /

Уговор УГ2: *KreirajIgru*

Операција: *KreirajIgru(Korisnik)*: сигнал

Веза са СК: СК2

Предуслови: Вредносна и структурна ограничења над објектом *Korisnik* морају бити задовољена.

Постуслови: Игра је креирана.

Уговор УГ3: *ZapocniIgru*

Операција: *ZapocniIgru(Mapa)*: сигнал

Веза са СК: СК3

Предуслови: Вредносна и структурна ограничења над објектом *Mapa* морају бити задовољена.

Постуслови: Игра је започета.

Уговор УГ4: *PogadjajPolje*

Операција: *PogadjajPolje(Koordinate)*: сигнал

Веза са СК: СК4

Предуслови: Вредносна и структурна ограничења над објектом *Координате* морају бити задовољена.

Постуслови: Мапа рачунара је ажурирана.

Уговор УГ5: *PrikaziRangListu*

Операција: *PrikaziRangListu()*: сигнал

Веза са СК: СК5

Предуслови: /

Постуслови: /

Уговор УГ6: *Krajlgre*

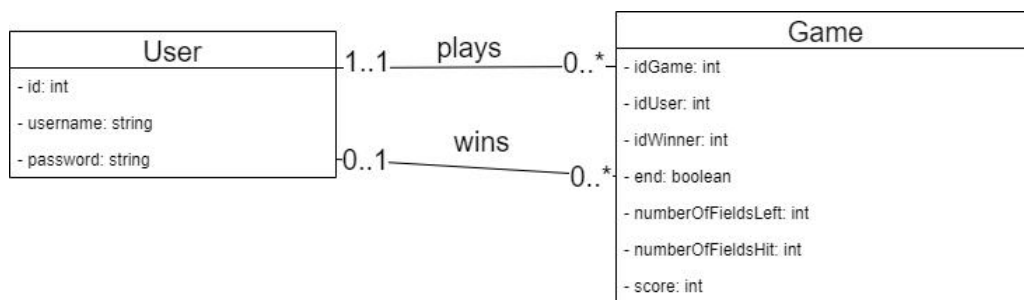
Операција: *Krajlgre(Koordinate)*: сигнал

Веза са СК: СК6

Предуслови: Вредносна и структурна ограничења над објектом *Координате* морају бити задовољена.

Постуслови: Игра је завршена.

2.3. Структура софтверског система - Концептуални модел



Слика 14 Структура софтверског система - Концептуални модел

2.4. Структура софтверског система - Релациони модел

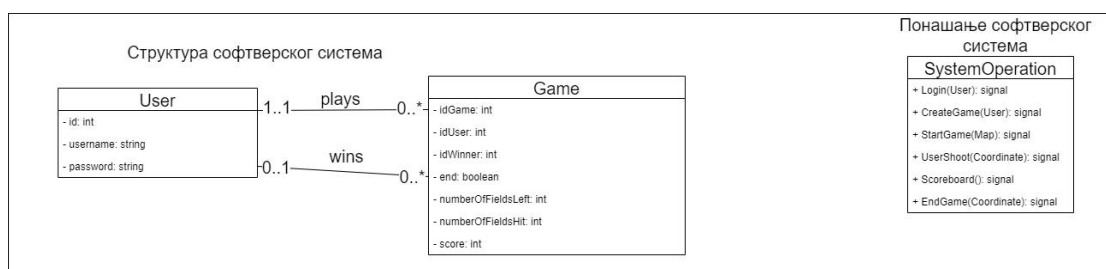
User (id, username, password)

Game (idGame, idUser, idWinner, end, numberOfFieldsLeft, numberOfFieldsHit, score)

Табела User		Просто вредносно ограничење		Сложено вредносно ограничење		Структурно ограничење
Атрибути	Име	Тип атрибута	Вредност атрибута	Међузав. Атрибута једне табеле	Међузав. Атрибута више табела	INSERT/ UPDATE/ DELETE/
	id	Integer	Not null and > 0			
	username	String	Not null			
	password	String	Not null			

Табела Game		Просто вредносно ограничење		Сложено вредносно ограничење		Структурно ограничење
Атрибути	Име	Тип атрибута	Вредност атрибута	Међузав. Атрибута једне табеле	Међузав. Атрибута више табела	INSERT/ UPDATE/ DELETE/
	idGame	Integer	Not null and > 0			
	idUser	Integer	Not null and > 0			
	idWinner	Integer	Not null and > 0			
	end	Boolean				
	numberOfFieldsLeft	Integer	> 0			
	numberOfFieldsHit	Integer	> 0			
	score	Integer	> 0			

Као резултат анализе сценарија СК и креирања концептуалног модела добија се логичка структура и понашање софтверског система:



Слика 15 Структура и понашање софтверског система

3. Пројектовање

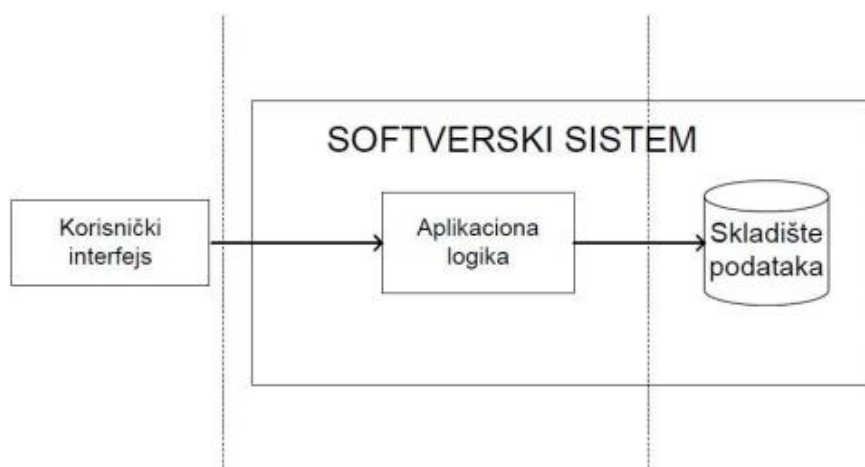
Фаза пројектовања описује физичку структуру и понашање софтверског система - архитектуру софтверског система.

3.1. Архитектура софтверског система

Архитектура система је тронивојска и састоји се из следећих нивоа:

- ♦ Кориснички интерфејс
- ♦ апликациона логика
- ♦ складиште података

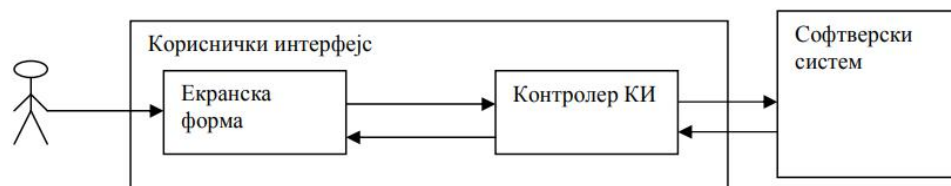
Кориснички интерфејс се налази на страни клијента, а апликациона логика и складиште података на страни сервера. Уколико постоји потреба, и серверска страна може поседовати кориснички интерфејс.



Слика 16 Тронивојска архитектура

3.2. Пројектовање корисничког интерфејса

Пројектовање корисничког интерфејса представља реализацију улаза и/или излаза софтверског система. Екранска форма има улогу да прихвати податке које корисник уноси, прихвата догађаје које прави корисник, позива контролера корисничког интерфејса како би му проследила податке и приказује податке добијене од контролера корисничког интерфејса.



Слика 17 Структура корисничког интерфејса

3.2.1. СК1: Случај коришћења - Пријављивање играча

Назив СК

Пријављивање корисника

Актори СК

Корисник

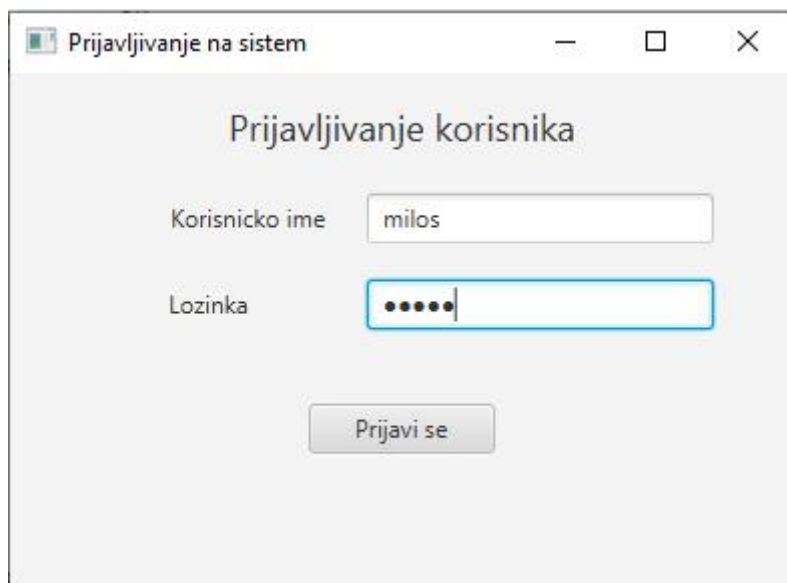
Учесници СК

Корисници систем (програм)

Предуслов: Систем је укључен и приказује форму за пријављивање играча.

Основни сценарио СК:

1. **Корисник** уноси податке за пријављивање. (АПУСО)

A screenshot of a graphical user interface for a login system. The window has a title bar that says "Prijavljivanje na sistem". Inside the window, the title "Prijavljivanje korisnika" is centered. Below the title, there are two input fields. The first is labeled "Korisnicko ime" and contains the text "milos". The second is labeled "Lozinka" and contains five dots, indicating a password. Below these fields is a button labeled "Prijavi se".

Слика 18 Пријављивање корисника - унос података

2. **Корисник** контролише да ли је коректно унео податке. (АНСО)

3. **Корисник** позива систем да изврши пријављивање играча. (АПСО)

4. **Систем** врши пријављивање играча. (СО)

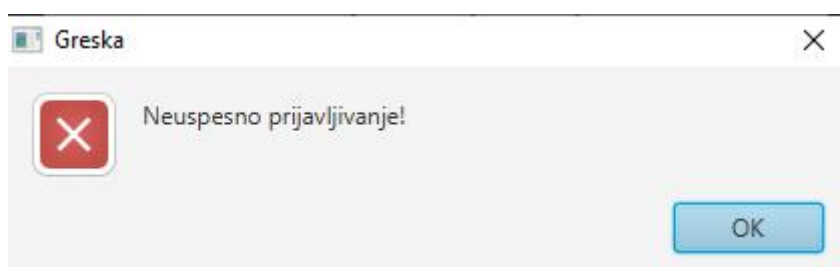
5. **Систем** приказује играчу поруку: "Uspesno ste se prijavili!!" и омогућава приступ систему. (ИА)



Слика 19 Пријављивање корисника - основни сценарио

Алтернативна сценарија:

5.1. Уколико систем не пронађе играча са унетим подацима, приказује играчу поруку: "Neuspesno prijavljivanje!". (ИА)



Слика 20 Пријављивање корисника - алтернативни сценарио

3.2.2. СК2: Случај коришћења - Креирање игре

Назив СК

Креирање игре

Актори СК

Играч

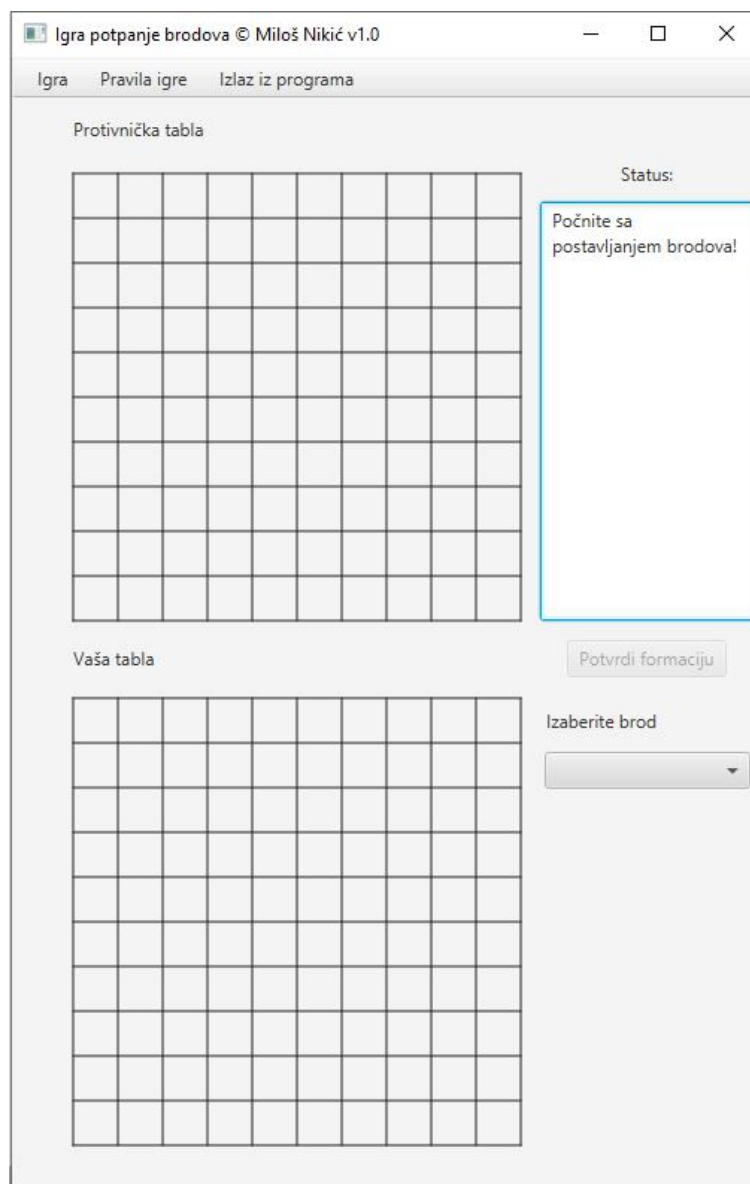
Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен. Корисник је пријављен на систем.

Основни сценарио СК:

1. **Корисник** позива систем да изврши креирање игре. (АПСО)
2. **Систем** врши креирање игре. (СО)
3. **Систем** приказује играчу поруку: "Počnite sa postavljanjem brodova!". (ИА)



Слика 21 Креирање игре - основни сценарио

Алтернативна сценарија:

3.1. Уколико систем не може да креира игру, приказује поруку: "Nije moguće kreirati igru!". (ИА)



Слика 22 Креирање игре - алтернативни сценарио

3.2.3. СКЗ: Случај коришћења - Започињање игре

Назив СК

Започињање игре

Актори СК

Играч

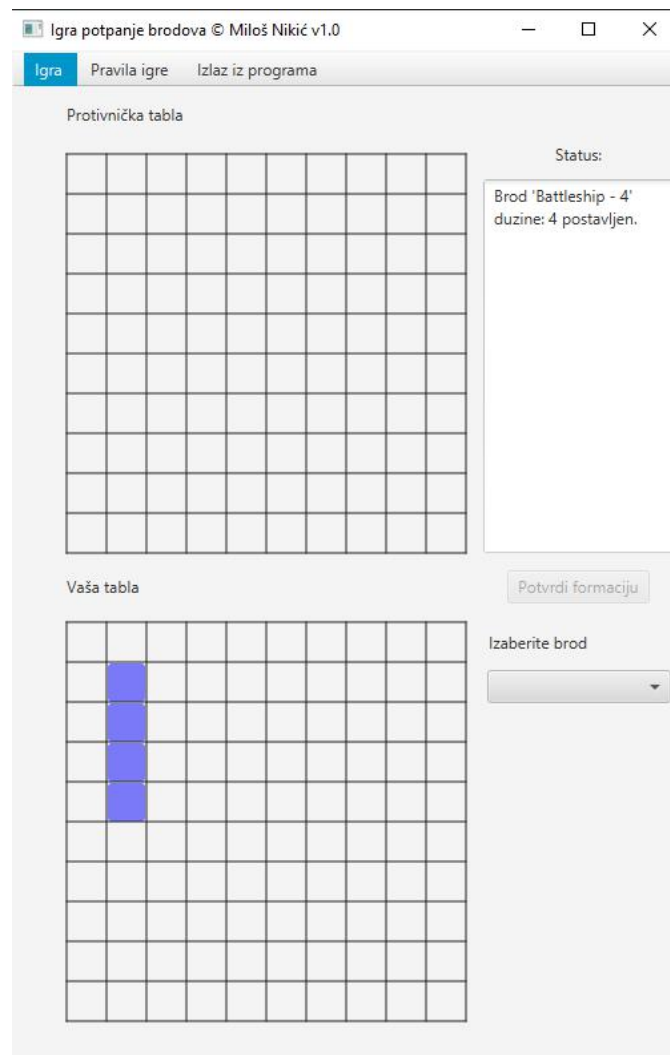
Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен. Корисник је пријављен на систем и игра је успешно креирана.

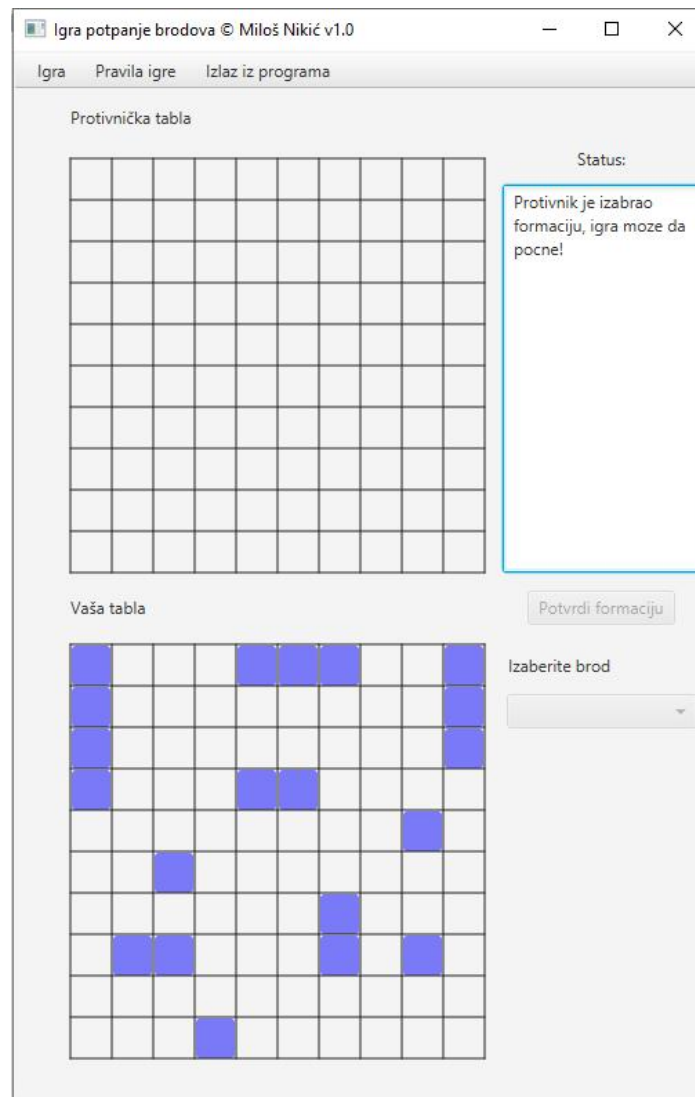
Основни сценарио СК:

1. **Корисник уноси** податке потребне за започињање игре. (АПУСО)



Слика 23 Започињање игре - унос података

2. **Корисник** контролише да ли је коректно унео податке. (АНСО)
3. **Корисник** позива систем да започне игру. (АПСО)
4. **Систем** врши започињање игре. (СО)
5. **Систем** приказује играчу поруку: “Protivnik je izabrao formaciju, igra moze da pocne!”. (ИА)



Слика 24 Започињање игре - основни сценарио

Алтернативна сценарија:

- 5.1 Уколико систем не може да започне игру, приказује поруку: “Problem prilikom startovanja igre!”. (ИА)



Слика 25 Започињање игре - алтернативни сценарио

3.2.4. СК4: Случај коришћења - Погађање поља

Назив СК

Погађање поља

Актори СК

Играч

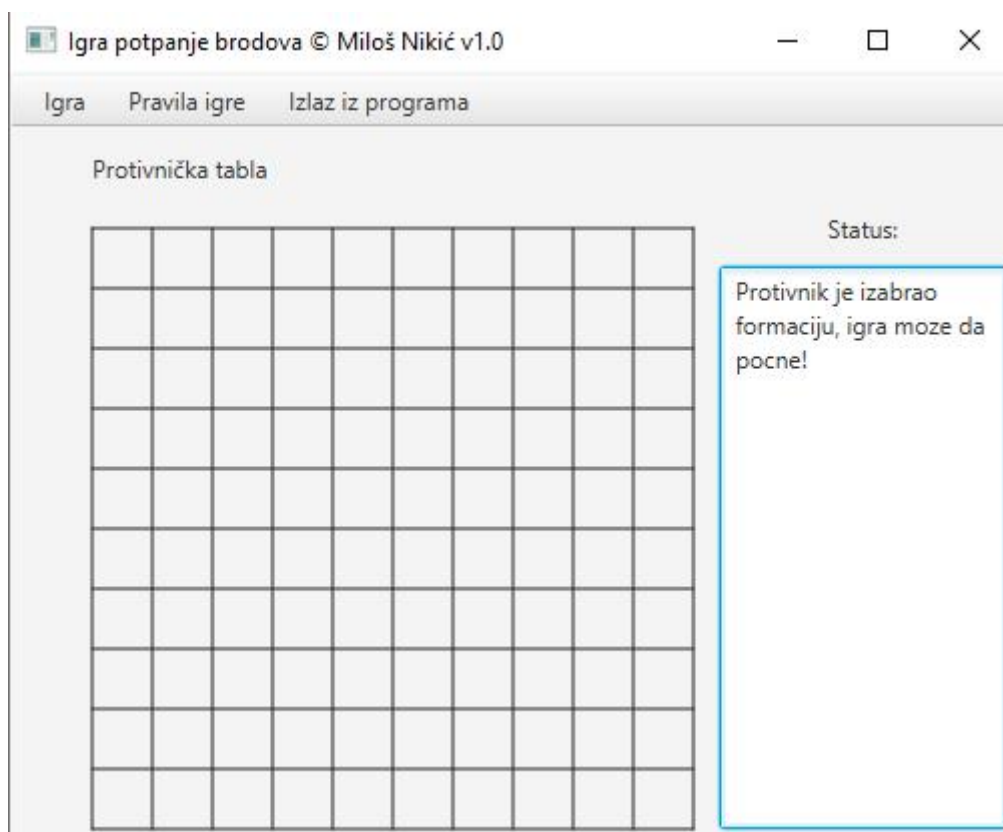
Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен. Корисник је пријављен на систем и игра је успешно креирана и започета.

Основни сценарио СК:

1. **Корисник** бира поље које жели да гађа. (АПУСО)



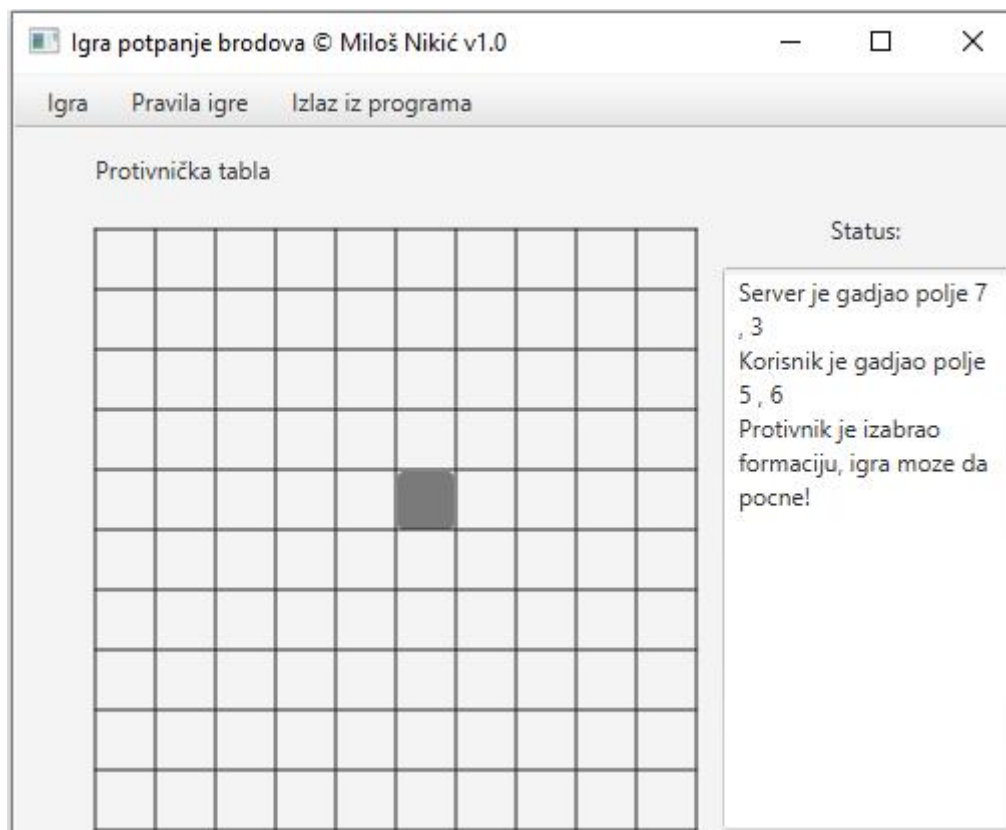
Слика 26 Погађање поља - одабир поља

2. **Корисник** контролише да ли је коректно унео податке. (АНСО)

3. **Корисник** позива систем да погађа поље. (АПСО)

4. **Систем** врши погађање поља. (СО)

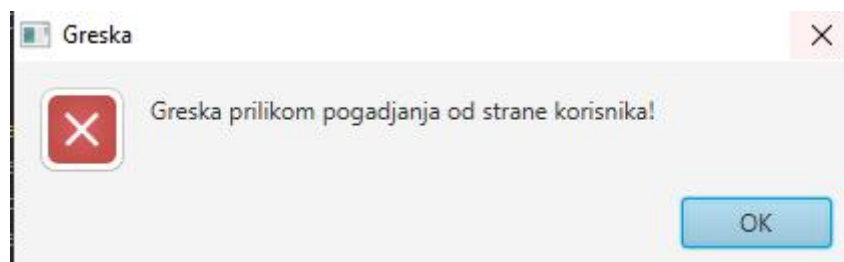
5. **Систем** приказује играчу поруку: "Korisnik je gadjaо polje sa trazanim koordinatama". (ИА)



Слика 27 Погађање поља - основни сценарио

Алтернативна сценарија:

5.1. Уколико систем не може да погађа поље, приказује поруку: “Greska prilikom gadjanja od strane korisnika!”. (ИА)



Слика 28 Погађање поља - алтернативни сценарио

3.2.5. СК5: Случај коришћења - Приказивање ранг листе

Назив СК

Приказивање ранг листе

Актори СК

Играч

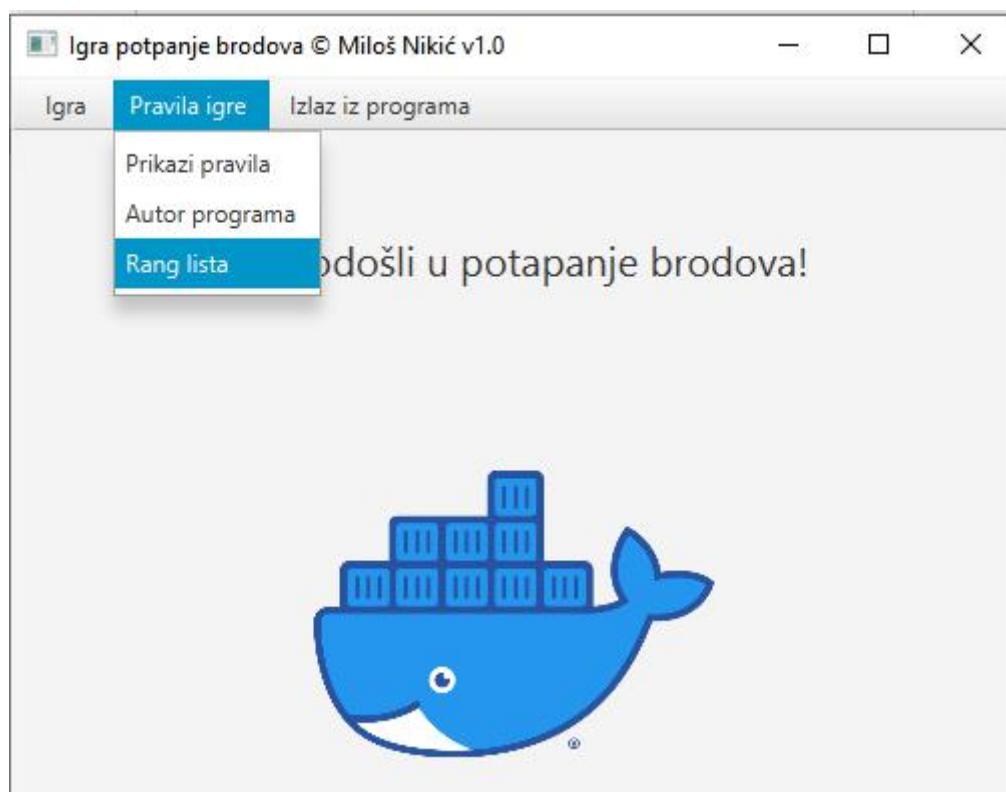
Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен и приказује форму за пријављивање играча. Корисник је пријављен на систем.

Основни сценарио СК:

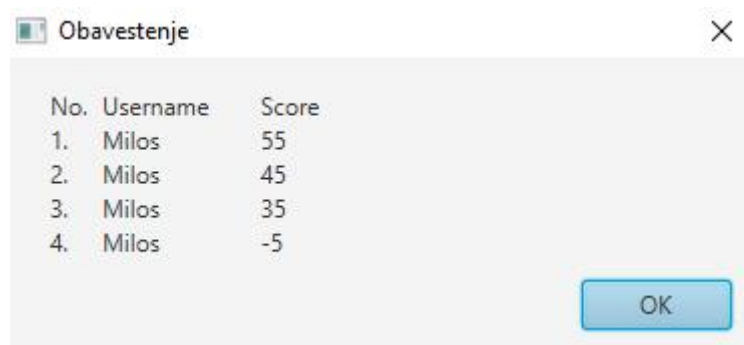
1. **Корисник** позива систем да прикаже ранг листу. (АПСО)



Слика 29 Приказивање ранг листе - позивање приказивања

2. **Систем** врши приказивање ранг листе. (СО)

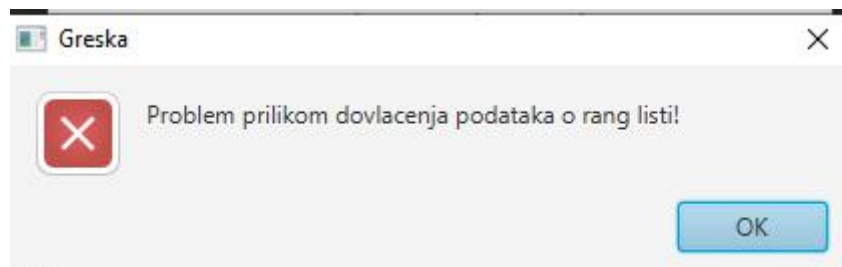
3. **Систем** приказује играчу ранг листу. (ИА)



Слика 30 Приказивање ранг листе - основни сценарио

Алтернативна сценарија:

3.1. Уколико систем не може да прикаже ранг листу, приказује поруку: "Problem prilikom dovlacenja podataka o rang listi.". (ИА)



Слика 31 Приказивање ранг листе - алтернативни сценарио

3.2.6. СК6: Случај коришћења - Прекид игре

Назив СК

Крај игре

Актори СК

Играч

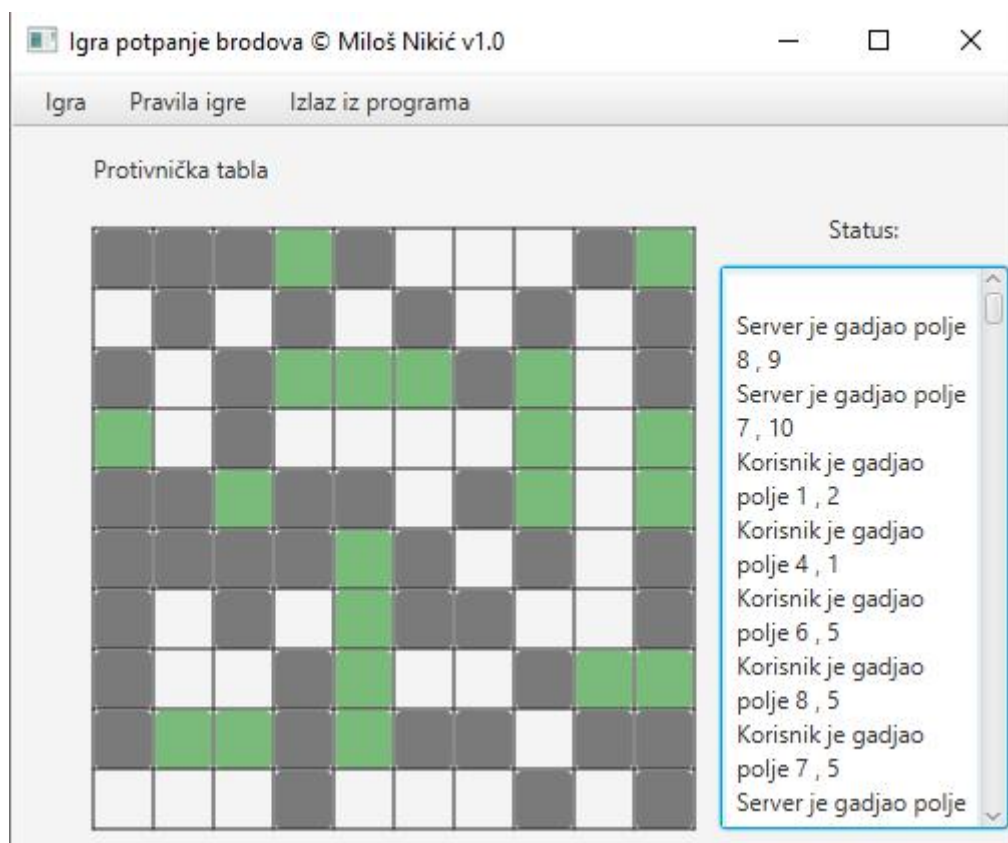
Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен и приказује форму за пријављивање играча. Корисник је пријављен на систем и игра је успешно креирана.

Основни сценарио СК:

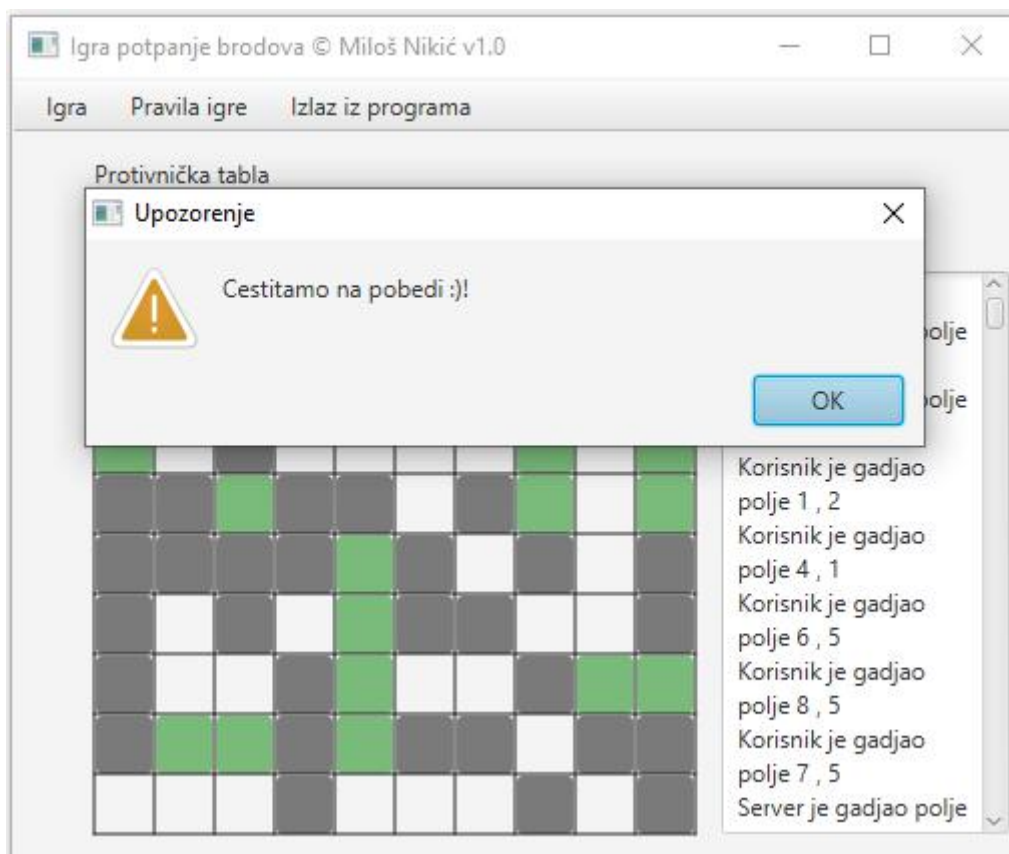
1. **Корисник** позива систем да заврши игру. (АПСО)



Слика 32 Крај игре - позивање краја игре

2. **Систем** врши крај игре. (СО)

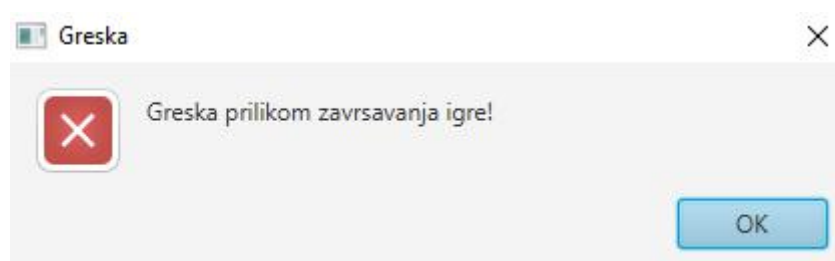
3. **Систем** приказује играчу поруку о исходу игре. (ИА)



Слика 33 Крај игре - основни сценарио

Алтернативна сценарија:

3.1. Уколико систем не може да прикаже ранг листу, приказује поруку: "Greska prilikom završavanja igre!". (ИА)



Слика 34 Крај игре - алтернативни сценарио

3.3. Комуникација сервер - клијент

На серверској страни се покреће веб сервис и серверски сокет који ослушкује мрежу. Када клијент успешно изврши пријављивање на систем, успоставља се конекција између клијентског сокета и серверског. Тада сервер генерише посебну нит која је задужена за обраду корисничких захтева.

Слање и примање података се одвија помоћу класа *Request* и *Response*. Приликом слања слања потребно је у *Request* дефинисати системску операцију као и податке неопходне за њено извршење. Када сервер прими и обради захтев, он креира нову инстанцу *Response* класе и шаље одговор клијенту, који он даље обрађује.

```
package transfer;
import domain.Coordinates;
import domain.Map;
import domain.User;
import java.io.Serializable;
import util.Operation;

public class Request implements Serializable {

    private Coordinates coordinates;
    private User user;
    private Map map;
    private Operation operation;

    public Request() {
    }

    public Coordinates getCoordinates() {
        return coordinates;
    }

    public void setCoordinates(Coordinates coordinates) {
        this.coordinates = coordinates;
    }

    public Map getMap() {
        return map;
    }

    public void setMap(Map map) {
        this.map = map;
    }

    public User getUser() {
        return user;
    }
}
```

```

    public void setUser(User user) {
        this.user = user;
    }

    public void setOperation(Operation operation) {
        this.operation = operation;
    }

    public Operation getOperation() {
        return operation;
    }
}

package transfer;

import domain.Coordinates;
import domain.Map;
import domain.RankItem;
import domain.Ship;
import util.ResponseStatus;
import domain.User;
import java.io.Serializable;
import java.util.List;
import util.Operation;

public class Response implements Serializable {

    private User user;
    private Operation operation;
    private ResponseStatus responseStatus;
    private Boolean hit;
    private Ship ship;
    private Boolean userPlaying;
    private Coordinates coordinates;
    private Map map;
    private List<RankItem> rankList;

    public Response() {
    }

    public List<RankItem> getRankList() {
        return rankList;
    }
}

```

```

public void setRankList(List<RankItem> rankList) {
    this.rankList = rankList;
}

public Map getMap() {
    return map;
}

public void setMap(Map map) {
    this.map = map;
}

public Coordinates getCoordinates() {
    return coordinates;
}

public void setCoordinates(Coordinates coordinates) {
    this.coordinates = coordinates;
}

public Ship getShip() {
    return ship;
}

public void setShip(Ship ship) {
    this.ship = ship;
}

public Boolean getUserPlaying() {
    return userPlaying;
}

public void setUserPlaying(Boolean userPlaying) {
    this.userPlaying = userPlaying;
}

public Boolean getHit() {
    return hit;
}

public void setHit(Boolean hit) {
    this.hit = hit;
}

public ResponseStatus getResponseStatus() {
    return responseStatus;
}

```

```
public void setResponseStatus(ResponseStatus responseStatus) {
    this.responseStatus = responseStatus;
}

public User getUser() {
    return user;
}

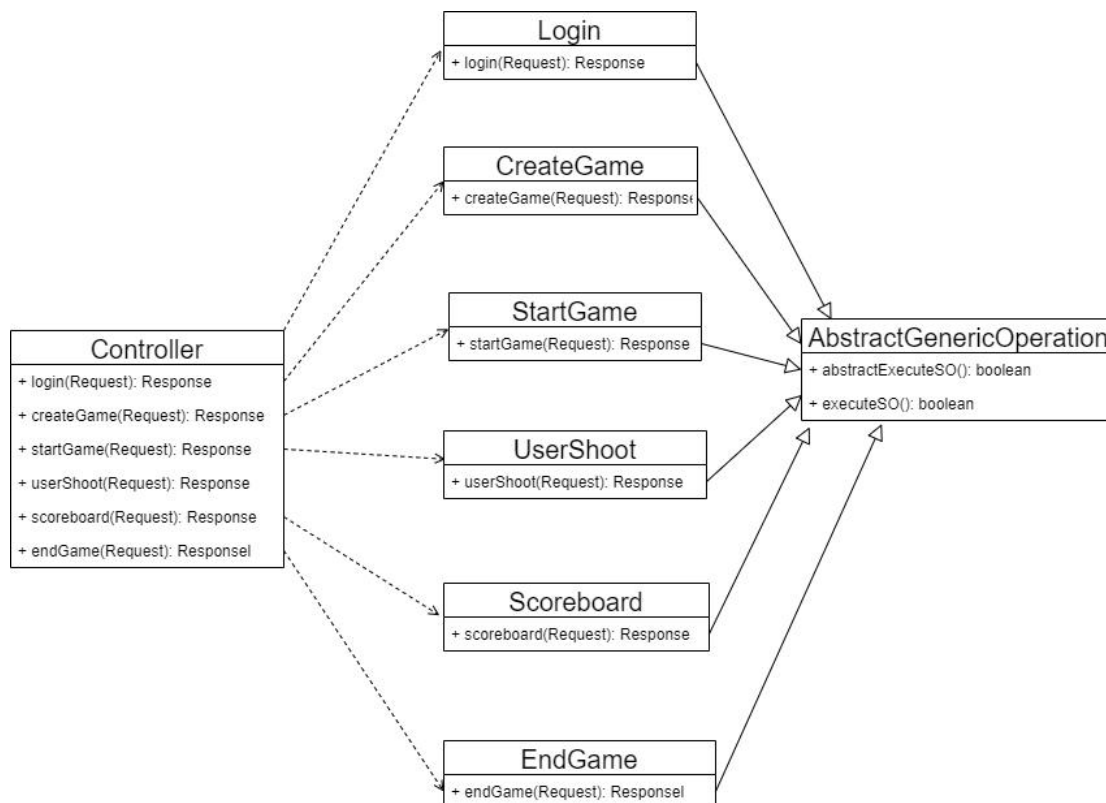
public void setUser(User user) {
    this.user = user;
}

public Operation getOperation() {
    return operation;
}

public void setOperation(Operation operation) {
    this.operation = operation;
}
}
```

3.4. Контролер апликационе логике

Контролер апликационе прихвата захтев за извршење системске операције од нити клијента и даље га преусмерава до класе које су одговорне за извршење конкретних системских операција. Након извршења системске операције контролер апликационе логике прихвата резултат и прослеђује га позиваоцу, односно нити која је задужена за обраду клијентских захтева.



Слика 35 Приказ зависности контролера и класа одговорних за извршење системских операција

Све класе које се задужене за извршење системских операција наслеђују класу **AbstractGenericOperation**, па самим тим и имплементирају методу `executeSO`, која је апстрактна. Метода **abstractExecuteSO** није апстрактна, па представља шаблон по ком редоследу се операције морају извршавати, а све подкласе, дају конкретну имплементацију апстрактне методе.

```

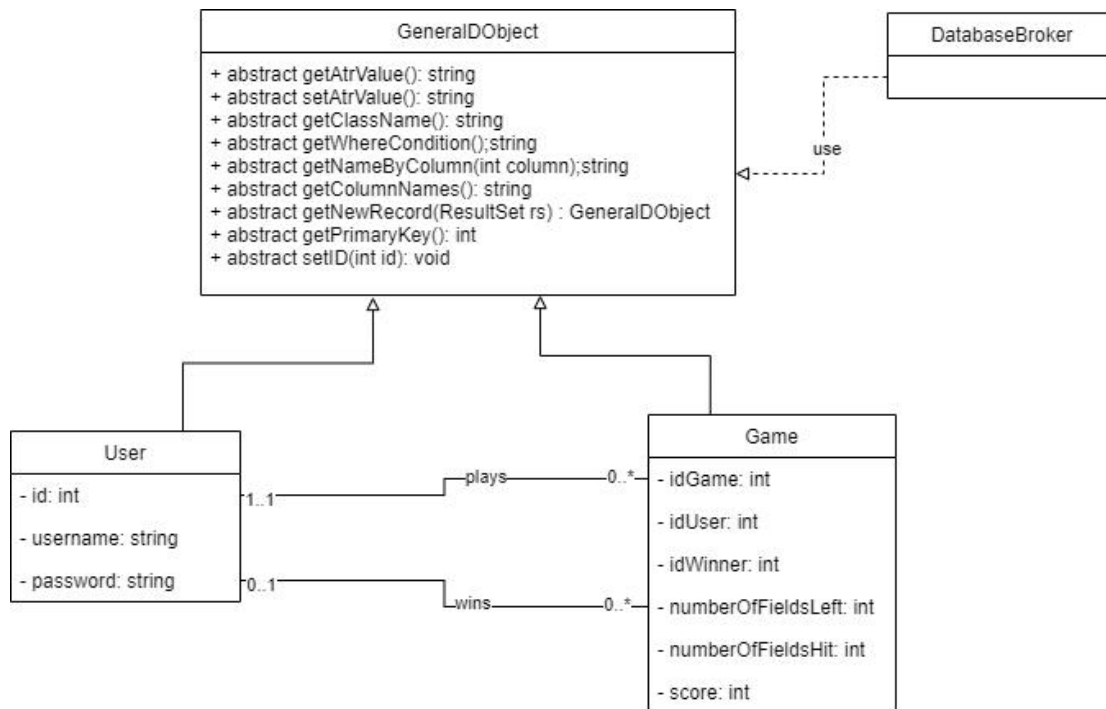
public abstract class AbstractGenericOperation {
    static public IDatabaseBroker bbp = new DatabaseBroker();
    GeneralDBObject gdo;

    synchronized public boolean abstractExecuteSO() {
        bbp.makeConnection();
        boolean signal = executeSO();
        if (signal == true) {
            bbp.commitTransation();
        } else {
            bbp.rollbackTransation();
        }
        bbp.closeConnection();
        return signal;
    }

    abstract public boolean executeSO();
}
  
```

3.5. Пројектовање структуре софтверског система - Доменске класе

Све доменске класе имплементирају интерфејс **GeneralDBObject**, који се користи као параметар у методама брокера базе података, како би се обезбедила инверзија зависности и омогућила имплементација генеричких метода.



Слика 36 Доменске класе и зависност брокера од GeneralDBObject

Дат је пример класе User која наслеђује класу **GeneralDBObject**:

```
/* User.java
 * @autor Milos Nikic,
 * Univerzitet u Beogradu
 * Fakultet organizacionih nauka
 * Katedra za softversko inzenjerstvo
 * Laboratorija za softversko inzenjerstvo
 * Datum:2020-08-14 Problem oko brojaca korisnika.
 */
package domain;

import java.io.Serializable;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;

@XmlAccessorType(XmlAccessType.FIELD)
public class User extends GeneralIDObject implements Serializable {

    public int id;
    public String username;
    public String password;

    public User() {
        id = 0;
        username = "";
        password = "";
    }

    public User(int idUser, String username, String password) {
        this.id = idUser;
        this.username = username;
        this.password = password;
    }

    // primarni ključ
    public User(int idUser) {
        this.id = idUser;
    }

    @Override
    public void setID(int id) {
        this.id = id;
    }
}

```

@Override

```
public int getPrimaryKey() {  
    return this.id;  
}
```

```
public int getIdUser() {  
    return id;  
}
```

```
public String getUsername() {  
    return username;  
}
```

```
public String getPassword() {  
    return password;  
}
```

```
public void setIdUser(int idUser) {  
    this.id = idUser;  
}
```

```
public void setUsername(String username) {  
    this.username = username;  
}
```

```
public void setPassword(String password) {  
    this.password = password;  
}
```

@Override


```

public GeneralDBObject getNewRecord(ResultSet rs) throws SQLException {
    return new User(rs.getInt("id"), rs.getString("username"), rs.getString("password"));
}

@Override
public String getAtrValue() {
    return (username == null ? null : "" + username + "") + ", " + (password == null ? null : "" +
password + "");
}

@Override
public String setAtrValue() {
    return "username=" + (username == null ? null : "" + username + "") + ", " + "password=" +
(password == null ? null : "" + password + "");
}

@Override
public String getClassName() {
    return "User";
}

@Override
public String getWhereCondition() {
    return "username = " + username + " AND " + "password = " + password + "";
}

@Override
public String getNameByColumn(int column) {
    String names[] = {"idUser", "username", "password"};
    return names[column];
}

```

```

public String[] getNameAtributes() {

    String names[] = {"idUser", "username", "password"};

    return names;

}

```

```

@Override

public String getColumnNames() {

    return " (username, password) ";

}

```

```

}

```

3.6. Пројектовање понашања софтверског система

За сваку системску операцију треба направити концептуална решења која су директно повезана са логиком проблема. За сваки од уговора пројектује се концептуално решење.

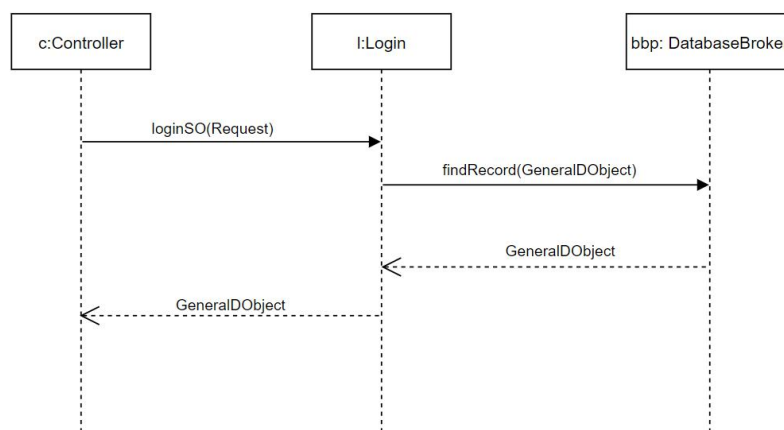
Уговор УГ1: *Prijavi*

Операција: *Prijavi(Korisnik)*: сигнал

Веза са СК: СК1

Предуслови: /

Постуслови: /



Слика 37 Уговор УГ1

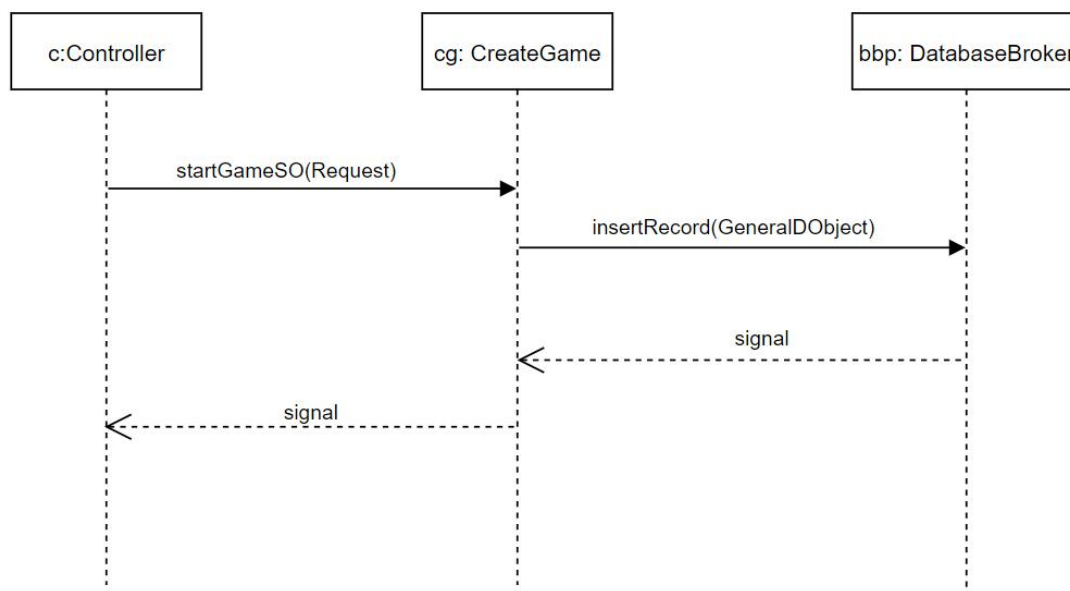
Уговор УГ2: *KreirajIgru*

Операција: *KreirajIgru(Korisnik)*: сигнал

Веза са СК: СК2

Предуслови: Вредносна и структурна ограничења над објектом *Korisnik* морају бити задовољена.

Постуслови: Игра је креирана.



Слика 38 Уговор УГ2

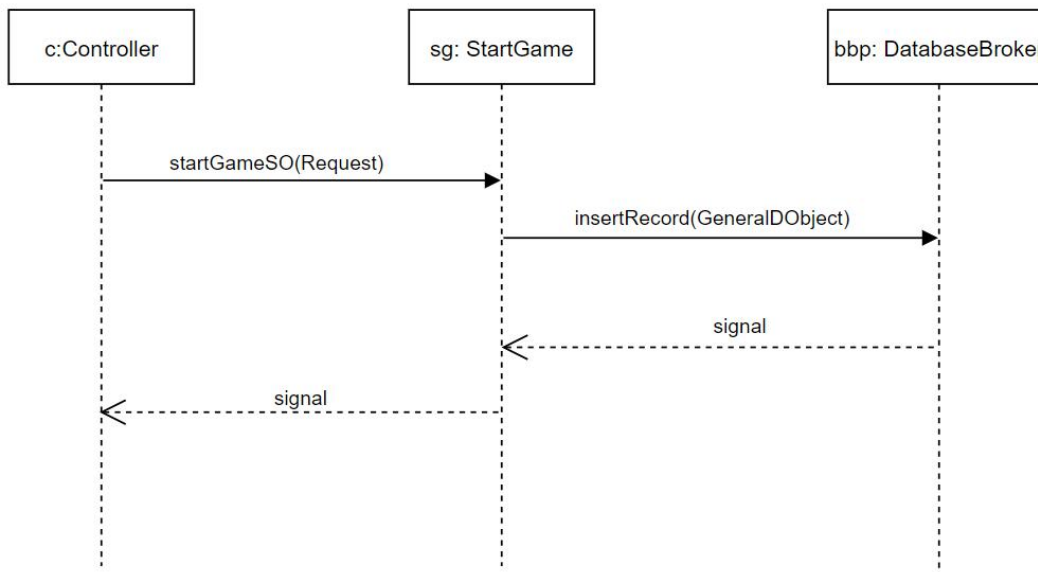
Уговор УГ3: *ZapocniIgru*

Операција: *ZapocniIgru(Мара)*:сигнал

Веза са СК: СК3

Предуслови: Вредносна и структурна ограничења над објектом *Мара* морају бити задовољена.

Постуслови: Игра је започета.



Слика 39 Уговор УГ3

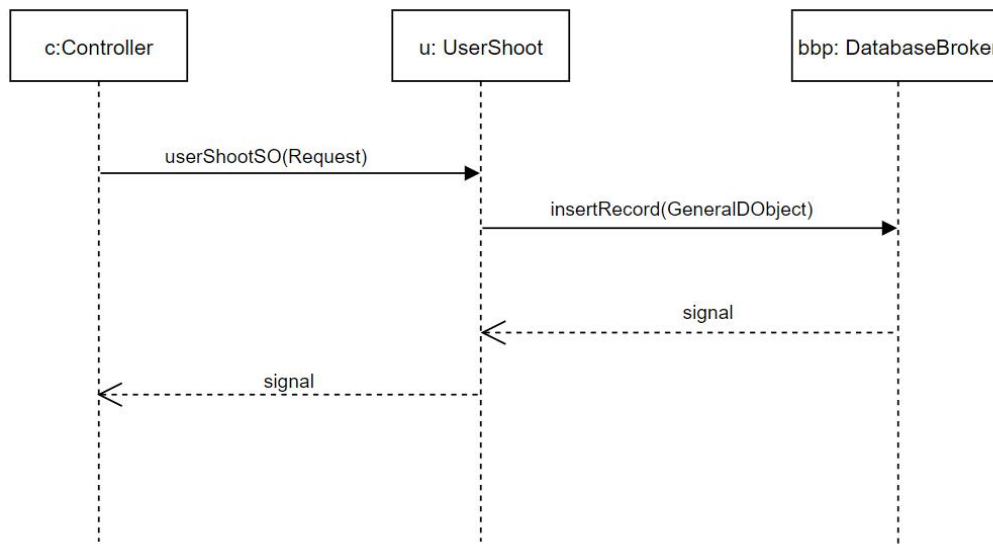
Уговор УГ4: *PogadjajPolje*

Операција: *PogadjajPolje(Koordinate)*: сигнал

Веза са СК: СК4

Предуслови: Вредносна и структурна ограничења над објектом *Координате* морају бити задовољена.

Постуслови: Мапа рачунара је ажурирана.



Слика 40 Уговор УГ4

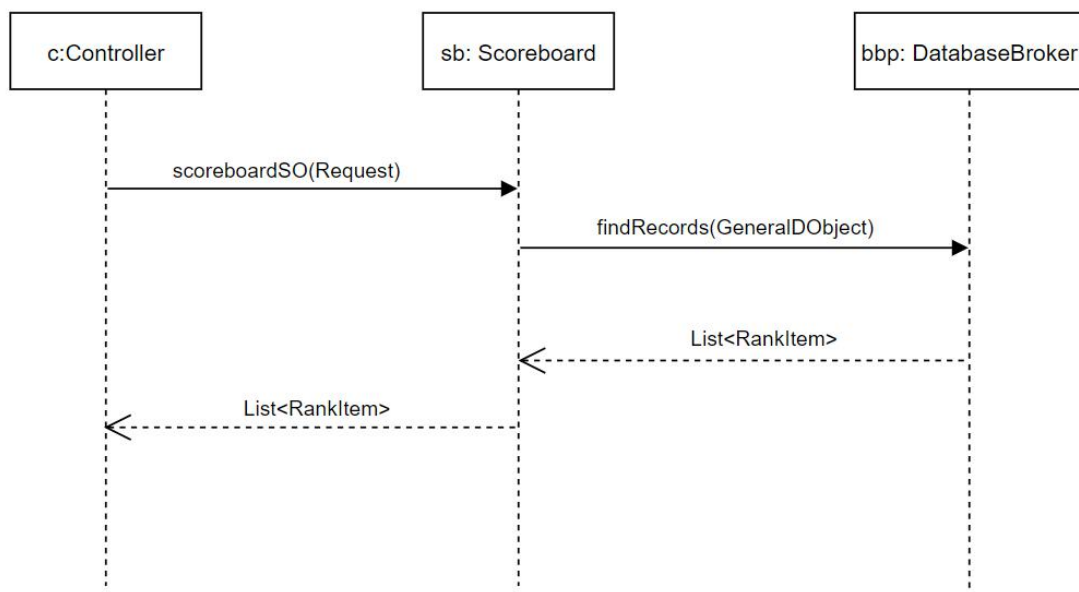
Уговор УГ5: *PrikaziRangListu*

Операција: *PrikaziRangListu()*: сигнал

Веза са СК: СК5

Предуслови: /

Постуслови: /



Слика 41 Уговор УГ5

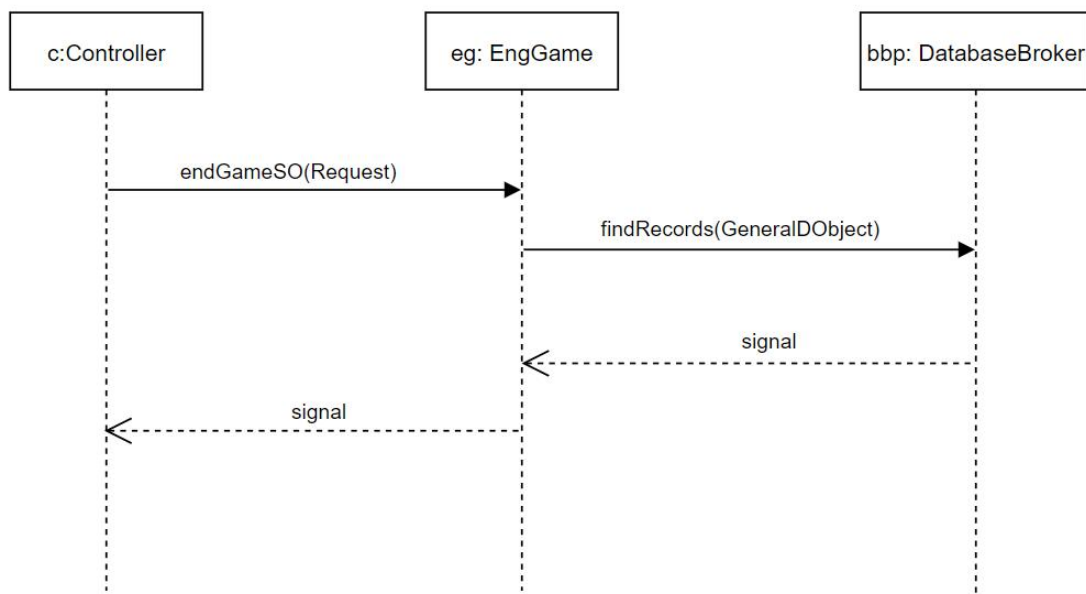
Уговор УГ6: *KrajIgre*

Операција: *KrajIgre(Koordinate)*: сигнал

Веза са СК: СК6

Предуслови: Вредносна и структурна ограничења над објектом *Координате* морају бити задовољена.

Постуслови: Игра је завршена.



3.7. Пројектовање брокера базе података

Класа *DatabaseBroker* је пројектована како би обезбедила перзистенцију објеката доменских класа и након завршетка рада програма.

Методе ове класе су генерички пројектоване јер као параметар примају **GeneralDObject**, па самим тим и све подкласе. Методе брокера базе података су:

```
public abstract boolean makeConnection();

public abstract boolean insertRecord(GeneralDObject odo);

public abstract boolean updateRecord(GeneralDObject odo,GeneralDObject odoold);

public abstract boolean updateRecord(GeneralDObject odo);

public abstract boolean deleteRecord(GeneralDObject odo);

public abstract boolean deleteRecords(GeneralDObject odo,String where);

public abstract GeneralDObject findRecord(GeneralDObject odo);

public abstract List<GeneralDObject> findRecords(GeneralDObject odo,String where);

public abstract boolean commitTransation();

public abstract boolean rollbackTransation();

public abstract boolean increaseCounter(GeneralDObject odo,AtomicInteger counter);

public abstract void closeConnection();

public abstract GeneralDObject getRecord(GeneralDObject odo,int index);

public abstract int getRecordsNumber(GeneralDObject odo);
```

Свака од наведених метода при извршењу позивају методе класе **GeneralDObject**, чије подкласе обезбеђују сопствене имплементације. Методе класе **GeneralDObject**:

```
abstract public String getAtrValue();

abstract public String setAtrValue();

abstract public String getClassName();

abstract public String getWhereCondition();

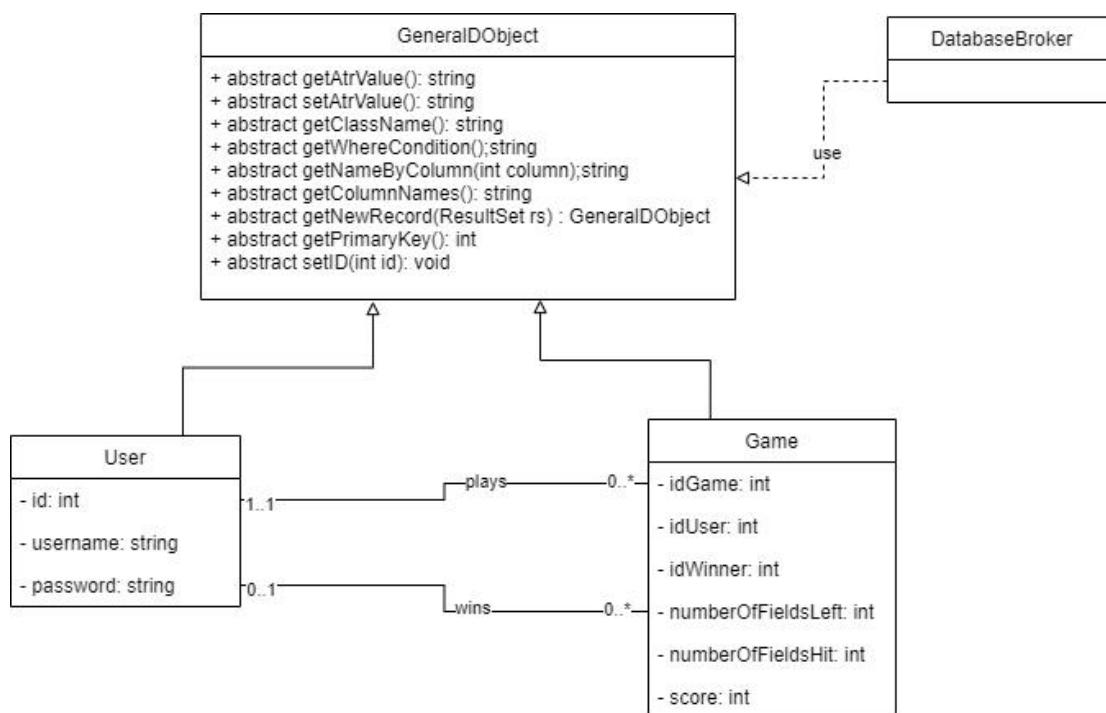
abstract public String getNameByColumn(int column);

abstract public String getColumnNames();

abstract public GeneralDObject getNewRecord(ResultSet rs) throws SQLException;

abstract public int getPrimaryKey();

abstract public void setID(int id);
```

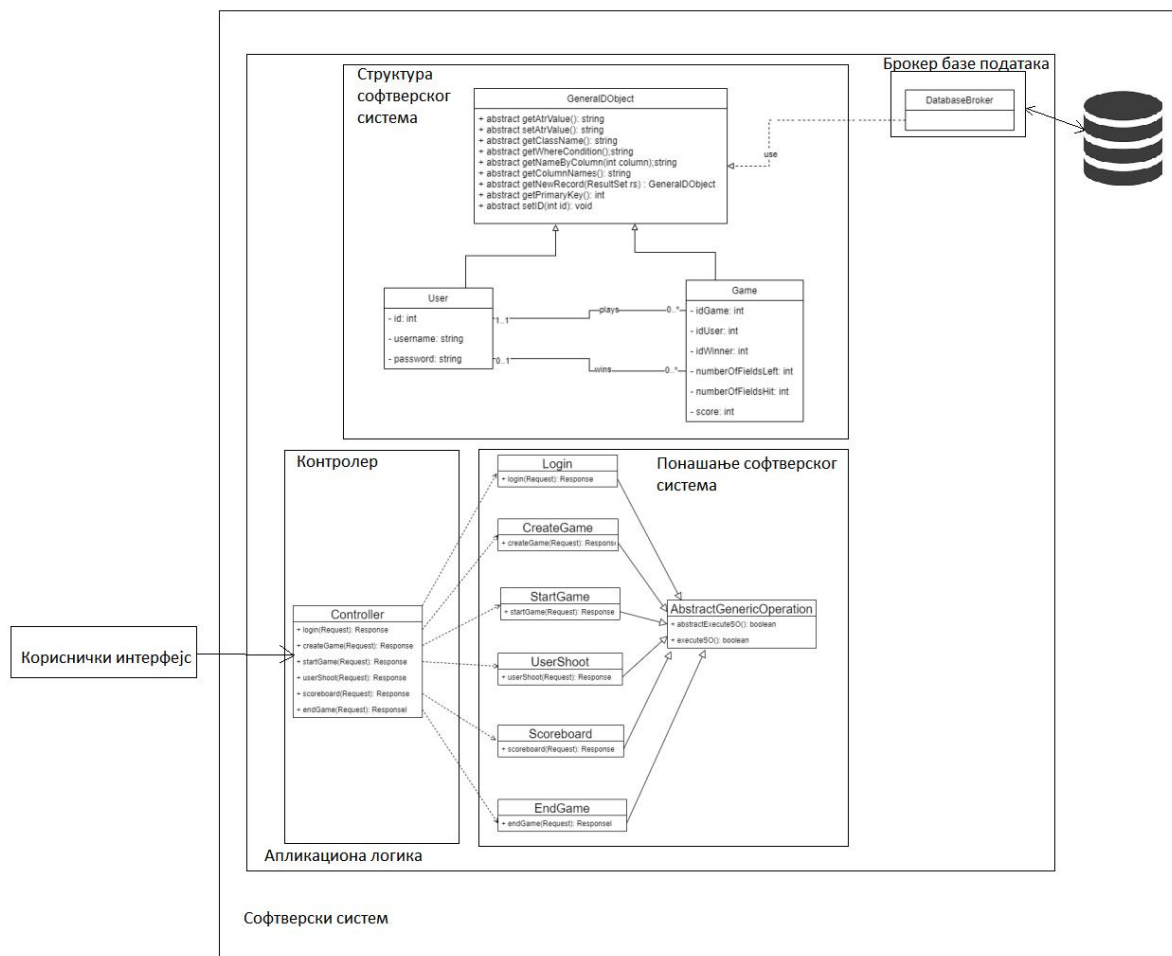


Слика 43 Веза брокера базе података са доменским класама

3.8. Пројектовање складишта података

Табела: User		
Име поља	Тип	Величина
id	int	20
username	varchar	50
password	varchar	50

Табела: Game		
Име поља	Тип	Величина
idGame	int	20
idUser	int	20
idWinner	int	20
end	boolean	1
numberOfFieldsLeft	int	11
numberOfFieldsHit	int	11
score	int	11



Слика 44 Архитектура софтверског система

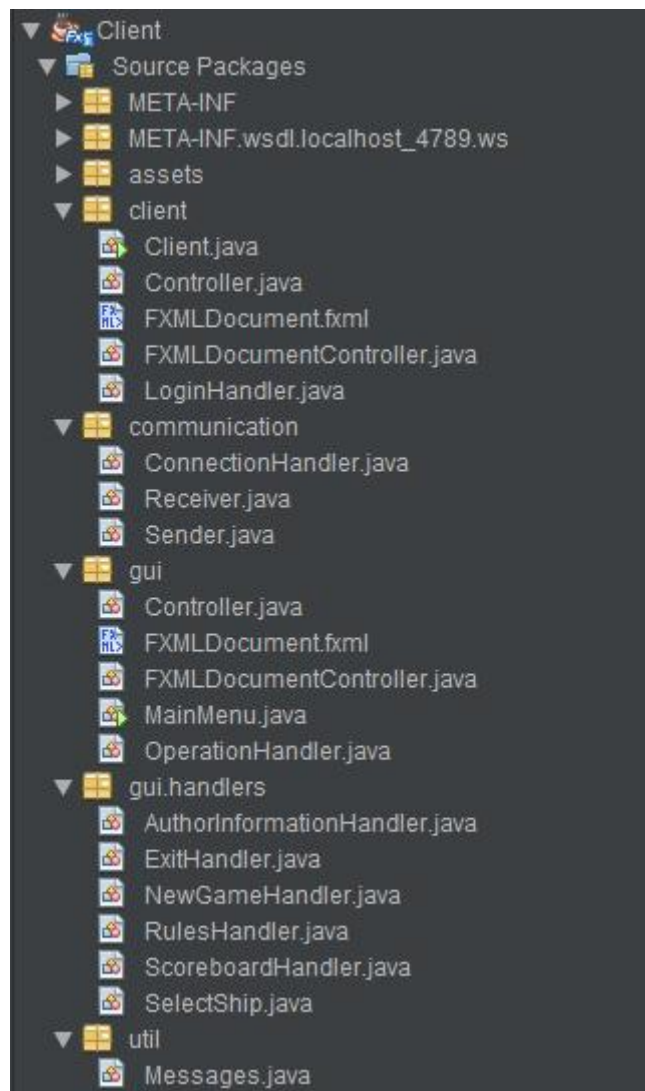
4. Имплементација

Софтверски систем који је описан у овом раду, имплементиран је у програмском језику *Java* као клијент-сервер систем.

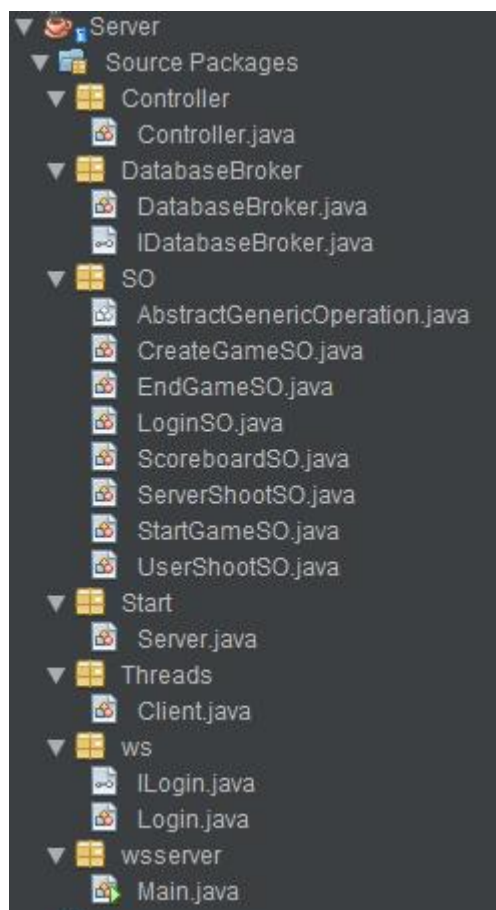
Као развојно окружење коришћен је *NetBeans 8.2*. Као систем за управљање базом података коришћен је *MySQL* систем за управљање базом података.

Систем се састоји из три пројекта: серверског дела апликације, клијентског дела и заједничке библиотеке.

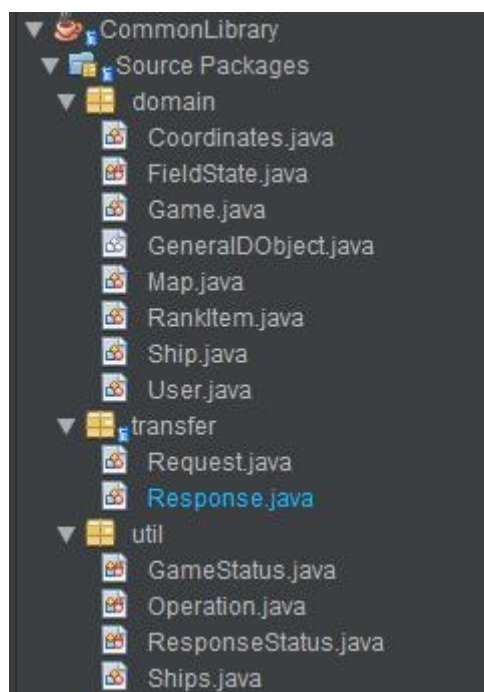
Клијентски пројекат користи *JavaFX* платформу која омогућава олакшан рад и креирање графичких компонената.



Слика 45 Имплементација клијентског дела апликације



Слика 46 Имплементација серверског дела апликације



Слика 47 Имплементација заједничке библиотеке

4.1. Механизам рефлексije

Механизам рефлексije представља начин испитивања или модификовања понашања метода, класа или интерфејса у време извршења програма. Овај механизам се реализује помоћу метаподатака класа.

На следећем примеру извршења системске операције се може видети употреба рефлексije.

```
@Override
```

```
public boolean executeSO() {
    try {
        Field game = Client.class.getDeclaredField("game");
        game.setAccessible(true);
        Game gameReflection = (Game) game.get(new Client());
        gameReflection.setend(true);
        if (this.request.getOperation() == Operation.SERVER_WIN) {
            gameReflection.setidWinner(-1);
            // 20 is number of total fields
            gameReflection.setnumberOfFieldsHit(20 - Client.serverMap.countAliveFields());
            gameReflection.setnumberOfFieldsLeft(Client.serverMap.countAliveFields());
            gameReflection.setscore(Client.serverMap.countAliveFields() * 5 - (20 -
Client.serverMap.countAliveFields()) * 2);
        } else {
            gameReflection.setidWinner(request.getUser().getIdUser());
            // 20 is number of total fields
            gameReflection.setnumberOfFieldsHit(20 - Client.userMap.countAliveFields());
            gameReflection.setnumberOfFieldsLeft(Client.userMap.countAliveFields());
            gameReflection.setscore(Client.userMap.countAliveFields() * 5 - (20 -
Client.userMap.countAliveFields()) * 2);
        }

        if (bbp.updateRecord(gameReflection)) {
            response.setResponseStatus(ResponseStatus.OK);
        } else {
            response.setResponseStatus(ResponseStatus.ERROR);
        }
        response.setOperation(Operation.END);
    } catch (NoSuchFieldException ex) {
        Logger.getLogger(EndGameSO.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

```

    } catch (SecurityException ex) {
        Logger.getLogger(EndGameSO.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IllegalArgumentException ex) {
        Logger.getLogger(EndGameSO.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IllegalAccessException ex) {
        Logger.getLogger(EndGameSO.class.getName()).log(Level.SEVERE, null, ex);
    }
    return true;
}

```

На овај начин се добија објекат доменске класе преко рефлексије поља класе *Client*. Овакав приступ је од великог значаја када је потребно позвати методу над објектом, чији тип није познат.

4.2. *Generics* механизам

По аналогiji са механизмом апстракције, о којем је раније било речи, овај механизам омогућава параметризовање метода, класа и интерфејса. Када је потребно креирати методу која врши операције над различитим типовима података, а чије извршење није зависно од улазних параметара, могуће је креирати генеричку методу.

Генеричка метода представља методу која је инваријантна у односу на параметре. На следећем примеру биће приказан начин на који је могуће убацити елементе у графички елемент *ComboBox* независно од њиховог типа.

```

private <T> void putItemsIntoCombobox(List<T> elements) {
    ObservableList<T> options
        = FXCollections.observableArrayList();
    for (T t : elements) {
        options.add(t);
    }

    this.document.shipsList.setItems(options);
}

private <T> void removeFromCombobox(T item, List<T> items) {
    ObservableList<T> options
        = FXCollections.observableArrayList();
    if (item instanceof Ship

```

```

        && items.contains(item)) {
            items.remove(item);
        }
        for (T t : items) {
            options.add(t);
        }

        this.document.shipsList.setItems(options);
    }

```

На претходним примерима уочава се издвајање општег и уклањање специфичности. Помоћу ових метода могуће је попунити графички елемент објектима независно од њиховог типа. Исти сценарио важи и за избацивање.

5. Тестирање

За сваки од идентификованих случајева коришћења је вршено тестирање над скупом исправних, али и неисправних података, како би се утврдили резултати извршења. На основу идентификованих проблема извршена је корекција проблема и отклоњени су уочени недостаци.

6. Закључак

Кроз овај рад је описан поступак развоја софтверског система помоћу упрошћене Ларманове методе, која се састоји од пет фаза:

1. Опис и захтева и случајеви коришћења
2. Анализа
3. Пројектовање
4. Имплементација
5. Тестирање

Приликом развоја разматрани су принципи, методе као и стратегије пројектовања софтвера. Да би софтверски систем могао лако да се одржава и надграђује, коришћени су узорни пројектовања, који раздвајају генералне делове од специфичних. Показан је пример MVC макро-архитектуралног утора. Такође, коришћен је и механизам рефлексije.

Како је било потребно омогућити да рачунар игра против играча, метод за одабир поља рачунара представља прилично наиван приступ, односно рачунар погађа насумично слободна поља на табли без памћења статуса претходног гађања.

7. Принципи, методе и стратегије пројектовања софтверског система

7.1. Принципи пројектовања софтверског система

7.1.1. Апстракција

“Апстракција је процес свесног заборављања информација, тако да ствари које су различите могу бити третиране као да су исте”.

Под апстракцијом подразумевамо издвајање општих, генеричких особина, не обраћајући пажњу на детаље и специфичности.

У контексту пројектовања софтвера, постоје два кључна механизма апстракције:

1. Параметризација
2. спецификација

Апстракција спецификацијом води до три главне врсте апстракција:

1. процедурална апстракција
2. апстракција података
3. апстракција контролом

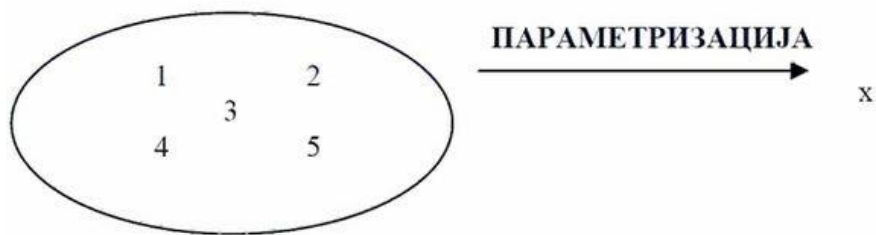
Параметризација је апстракција која издваја из неког скупа елемената њихова општа својства која су представљена преко параметара.

Постоји пет случајева параметризације:

1. параметризација скупа елемената простог типа
2. параметризација скупа елемената сложеног типа
3. параметризација скупа операција
4. параметризација скупа процедура
5. параметризација скупа наредби

1. Параметризација скупа елемената простог типа

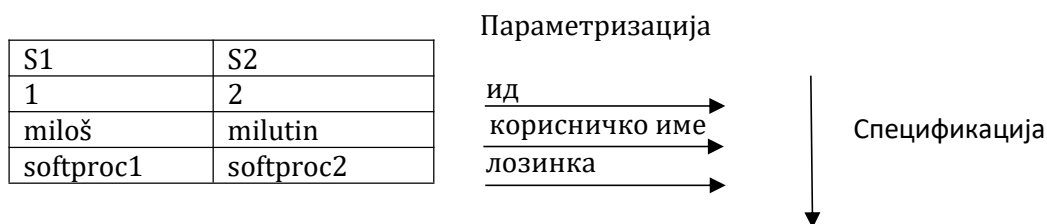
Уколико посматрамо скуп целих бројева 1,2,3..њих можемо представити преко неког општег представника скупа целих бројева. На пример помоћу параметра x . У приказаном случају, програмски би то записали као: **int x**;



Слика 48: Параметризација скупа елемената простог типа

2. Параметризација скупа елемената сложеног типа

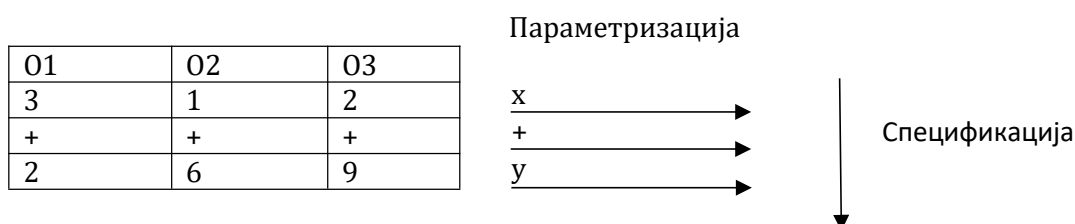
Уколико посматрамо неки скуп сложених објеката, нпр. скуп играча: (1, "miloš", "softproc1"), (2, "milutin", "softproc2"), тада параметризацијом добијамо општа својства тог скупа, односно њихове атрибуте: ИД, корисничко име и лозинку. Наведени скуп објеката означимо са S , а елементе скупа са $s1, s2$.



- ❖ Параметризацијом добијамо општа својства елемената скупа
- ❖ Пример је класа у објектно оријентисаним програмским језицима, јер представља представника неког скупа, а параметризацијом долазимо до његових својстава, односно атрибута класе
- ❖ Спецификација следи параметризацију, јер наводи општа својства елемената скупа, односно користити резултате параметризације
- ❖ Апстракција је дефинисана именом и спецификацијом скупа:
Корисник (ид, корисничко име, лозинка).

3. Параметризација скупа операција:

Уколико посматрамо неки S , скуп операција: { (3+2), (1+6), (2+9)}, уочавамо општу операцију сабирања над два елемента и можемо је дефинисати са $x+y$. Бројеви x и y представљају операнде над којима се операција извршава, док $+$ представља оно што операција ради. Наведени скуп елемената означен је са $O1, O2$ и $O3$:



- ❖ Специфицирањем општих својстава операције, добијамо њене елементе: операнде и оператор
- ❖ Специфицирањем неког скупа операција, из њега добијамо општу операцију.
- ❖ Спецификацијом операције, за наведени пример, се добија општа операција за сабирање два броја.

4. Параметризација скупа процедура

Уколико посматрамо неки скуп S процедура:

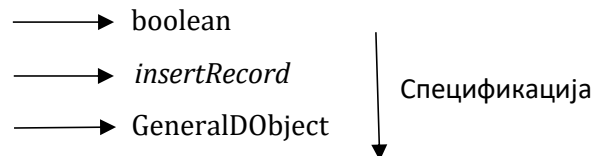
$s1: \text{boolean insertRecord}(\text{GeneralDObject gdo})$ – метода која чува игру

$s2: \text{boolean insertRecord}(\text{GeneralDObject gdo})$ – метода која чува играча,

учавамо да параметризацијом добијамо општу процедуру: $\text{boolean insertRecord}(\text{GeneralDObject gdo})$. Елементи који чине процедуру су:

повратна вредност процедуре, назив процедуре, аргументи (параметри) процедуре и тело процедуре.

	S1	S2
Повратна вредност процедуре	boolean	boolean
назив	<i>insertRecord</i>	<i>insertRecord</i>
аргументи	GeneralDObject	GeneralDObject
Тело процедуре	Метода која чува игру	Метода која чува играча



- ❖ Навођење општих својстава процедуре представља спецификацију процедуре
- ❖ Спецификацијом процедуре се добија општа процедура над неким скупом процедура
- ❖ Резултат спецификације процедуре је њен потпис
- ❖ Као такав, потпис процедуре представља процедуралну апстракцију неког скупа процедура.

У нашем примеру добијамо следећи потпис: $\text{boolean insertRecord}(\text{GeneralDObject gdo})$.

5. Параметризација скупа наредби

Уколико посматрамо скуп S наредби:

1. $S1: \text{System.out.println}(\text{"Korisnik " + users[0].getUsername() + " se prikljucio igri!"})$;

2. $S2: \text{System.out.println}(\text{"Korisnik " + users[1].getUsername() + " se prikljucio igri!"})$;

Уочићемо да се параметризацијом истичу општа својства ових наредби, а то су:

- ❖ $\text{System.out.println}$
- ❖ "Igrac "
- ❖ $\text{+igraci[i].getKorisnickoIme() +}$
- ❖ $\text{" se prikljucio igri!"}$
- ❖);

S1	S2
System.out.printl(System.out.printl(
"Korisnik "	"Korisnik "
+users[0].getUsername()+	+users[1].getUsername()+
" se prikljucio igri!"	" se prikljucio igri!"
););

Параметризација:

```
-----> System.out.printl(
-----> "Korisnik "

-----> +users[0].getUsername()+
-----> " se prikljucio igri!"
-----> ");
```

Навођење општих својстава наредбе представља спецификацију наредбе.

Као резултат спецификације наредбе добија се:

```
for(int i=0;i<2;i++){ System.out.println("Korisnik "+users[i].getUsername()+" se prikljucio igri!");}
```

Спецификацијом наредби из скупа наредби, добијамо општу наредбу.

- ❖ Спецификација наредби представља апстракцију наредби, један од облика **апстакције контролом**.

Спецификација је апстракција која издваја из неког скупа елемената њихова општа својства, која могу бити представљена преко процедуре, податка или контроле.

1. Процедурална апстракција

Процедуралном апстракцијом издвајамо из неког скупа процедура оно што су њихова општа својства:

- ❖ Тип повратне вредности
- ❖ име процедуре
- ❖ аргумент процедуре

Као што је већ речено, резултат процедуралне апстракције је потпис процедуре. Њен резултат, поред потписа, могу бити додатне информације о процедури, као што су услови извршења и/или додатни опис шта она ради.

Предности процедуралне апстракције су да се фокусира на то шта процедура ради, не бави се детаљима њене имплементације и начином реализације.

2. Апстракција података

- ❖ Апстракцијом података издвајамо општа својства неког датог скупа података

Уколико имамо нпр. скуп играча {(1, "miloš", "softproc1"), (2, "milutin", "softproc2")}, Тада се параметризацијом добијају општа својства скупа: ид, корисничко име, лозинка.

Параметризацију следи спецификација, па се навођењем ових општих својстава добија спецификација скупа.

Апстракција података је дефинисана именом (Korisnik) скупа и спецификацијом скупа: ид, корисничко име, лозинка.

Уколико елементи скупа имају и структуру и понашање – објекти, тада се над њима ради и процедурална апстракција и апстракција података. Резултат примене ових апстракција је класа која има своја стања – *атрибуте* и понашање-*методе*. Уколико

елементи скупа имају само понашање, резултат процедуралне апстракције ће бити интерфејс који садржи скуп потписа процедура.

3. Апстракција контролом

Постоје два вида апстракције контролом:

1. Апстракција наредби

Овом апстракцијом се из неког скупа наредби издваја оно што је опште и представља се помоћу контролне структуре и опште наредбе.

2. Апстракција структура података

Овом апстракцијом се из неког скупа структура издвајају њихове опште особине и представљају се помоћу итератора који контролише пролазак кроз структуру података.

Спојеност (coupling) и кохезија (cohesion)

Оно чему се тежи при развоју софтверских система је остваривање:

- ❖ Што веће кохезије – *high cohesion*
- ❖ Што мање повезаних класа – *low coupling*

Кохезија

Класа X треба да обезбеди неко понашање `m1()`.

```
class X{  
    public m1(){}  
}
```

Уколико се при извршењу `m1` позивају друге методе класе X, `m11`, `m12` и `m13`:

```
class X{  
    public m1(){m11(); m12(); m13();}  
    private m12(){...}  
    private m13(){...}  
}
```

онда се може рећи да наведена класа има високу кохезију. Кохезија нам говори о томе колико су методе унутар класе међусобно повезане.

Губитак кохезије значи да је нека класа одговорна да обезбеди више различитих понашања, која између себе нису повезана.

```
class Y{  
    public m1(){m11(); m12(); m13();}  
    public m2(){m21(); m22(); m23();}
```

```

public m3(){m31(); m32(); m33();}
private m12(){...}
private m13(){...}
private m21(){
}

```

Оваква класа је тешка за одржавање и надградњу.

У овом примеру видимо да међу приватним методама које спадају под извршење једне операције постоји повезаност (m11,m12,m12), али да између метода које припадају различитим извршењима (m12, m23), не постоји повезаност.

Самим тим, закључујемо да треба правити класе које имају високу кохезију, ради лакше надградње и одржавања,

Повезаност – Coupling

Повезаност – купловање значи да класе међусобно зависе једна од друге. То значи да класа не може да обави неку операцију без присуства друге класе.

На пример, класа X, има методу m1, која при извршењу позива методу m2, класе Y.

У том случају, класа X зависи од класе Y.

```

class X{
    Yu;
    public m1(){y=new Y(); y.m2();}
}
class Y{
    public m2(){..}
}

```

Купловање представља меру повезаности класе са другим класама, којом се утврђује колико је једна класа зависна од других.

Класа која јако зависи од других класа је класа са снажном повезаношћу – *High/strong coupling*.

Закључак је да треба правити класе са слабом повезаношћу са другим класама, иако је прихваћено да је међусобна повезаност класа неизбежна.

Пример међусобне повезаности класа је класа *OpstelzvršenjeSO*, која у својој *template* методи, иако добро пројектованој, приказује зависност од класе *BrokerBazePodataka*.

```

public abstract class AbstractGenericOperation {

    static public IDatabaseBroker bbp = new DatabaseBroker();
    GeneralDBObject gdo;

    synchronized public boolean abstractExecuteSO() {
        bbp.makeConnection();
        boolean signal = executeSO();
        if (signal == true) {
            bbp.commitTransation();
        } else {
            bbp.rollbackTransation();
        }
        bbp.closeConnection();
        return signal;
    }

    abstract public boolean executeSO();
}

```

Слика 48 Повезаност класа AbstractGenericOperation и DatabaseBroker

Међутим, овакав ниво повезаности је низак, прихватљив и незбежан, јер класа *OpstelzvrjenjeSO* зависи само од једне друге класе.

Декомпозиција и модуларизација

Декомпозиција представља процес рашчлањивања, процес који полазни проблем дели у скуп потпроблема, који се независно решавају. Када су потпроблеми решени на тај начин, омогућавају да се полазни проблем лакше реши.

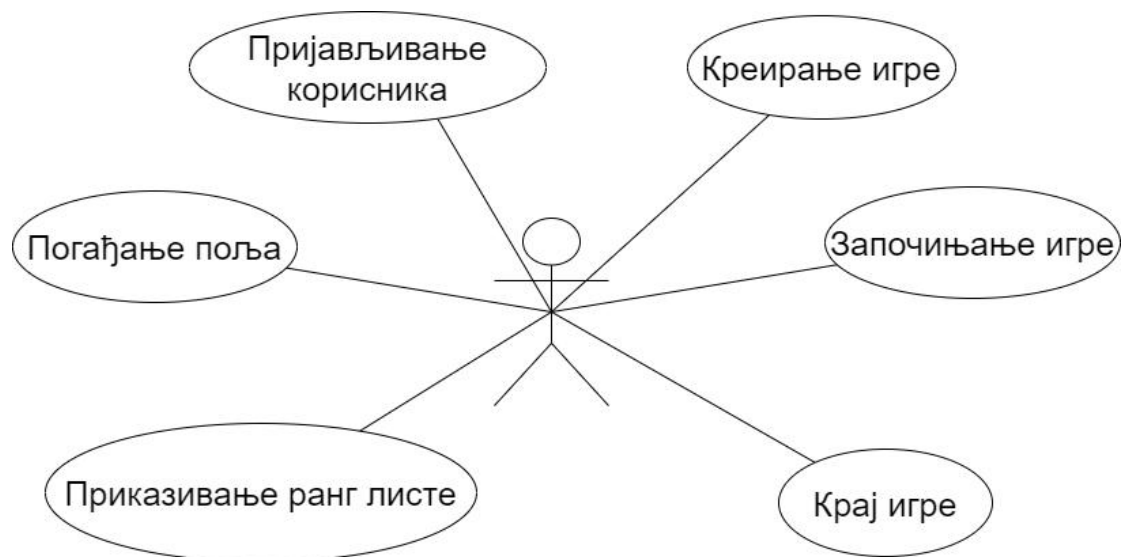
Уколико примењујемо принцип декомпозиције при развоју софтверског система, доћи ће до модуларизације софтверског система.

Закључак је да модуларизација софтверског система настаје као последица процеса декомпозиције.

Декомпозиција може бити објашњена са више аспеката при развоју софтвера:

1. Декомпозиција код прикупљања захтева

Кориснички захтев се код прикупљања декомпонује на скуп захтева који се описују преко случајева коришћења: Пријављивање, креирање игре, ажурирање игре, претрага игара, погађање слова, прекид игре.

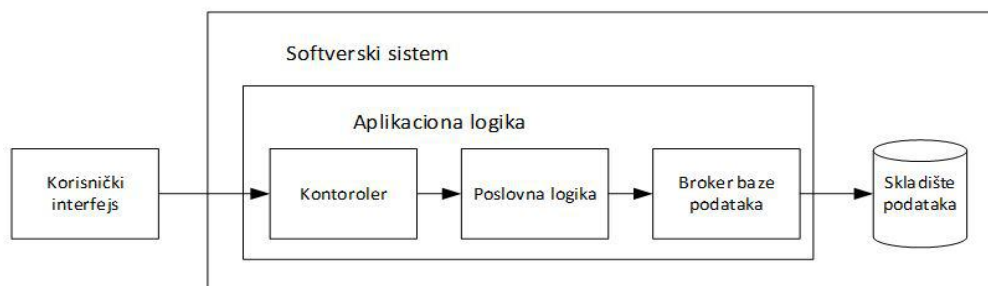


Слика 49 Декомпозиција код прикупљања захтева

2. Декомпозиција код пројектовања софтвера

При развоју софтверског система, применом Ларманове методе се, у трећој фази - фази пројектовања, врши пројектовање архитектуре софтверског система која је модуларна:

1. Модул – кориснички интерфејс
2. Модул – апликациона логика
3. Модул – складиште података



Слика 50 Декомпозиција код пројектовања софтвера

Сваки модул је добијен декомпозицијом, може се независно пројектовати и имплементирати.

Постоје принципи који морају бити испуњени да би софтверски систем могао да се декомпонује у модуле:

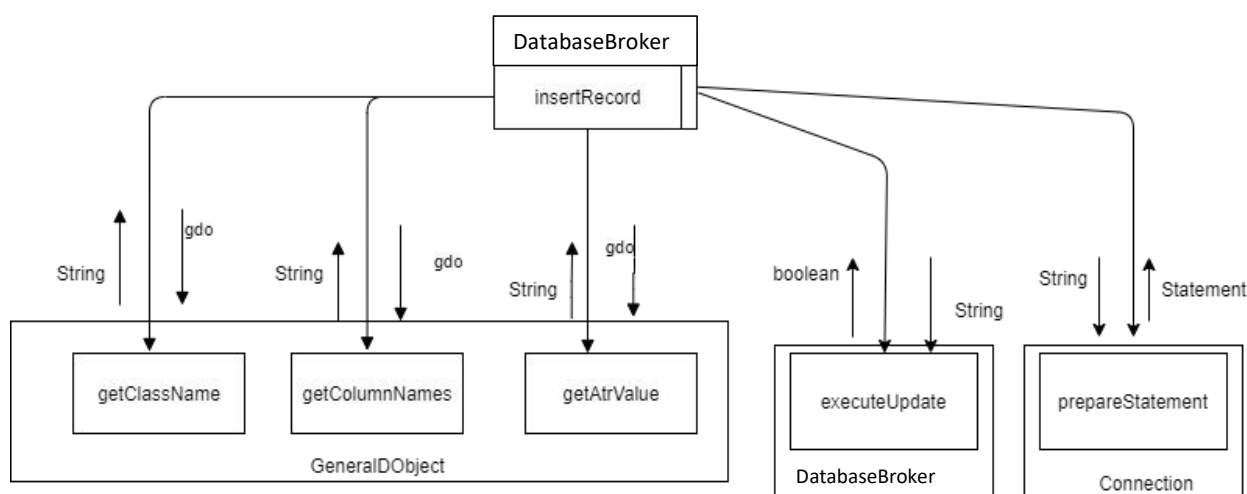
- ❖ Модули треба да имају јаку кохезију (*high cohesion*)
- ❖ Модули треба буду што је могуће слабије везани (*low coupling*)
- ❖ Сваки модул треба да чува своје интерне информације (*information hiding*)

Резултат процеса декомпозиције је софтверски систем који је модуларизован.

4. Декомпозиција функција (метода)

Функција – метода неке класе која је сложена, може бити декомпонована на више подфункција. Ове подфункције се независно решавају да би се на крају све интегрисале у једну целину како би се реализовала почетна функција.

У примеру, метода брокера базе података *insertRecord*.



Слика 52: Декомпозиција функција

Учаурење – енкапсулација / Сакривање информација – *information hiding*

Учаурење је процес којим се раздвајају:

- ❖ Особине модула – класе, које су јавне за друге модуле
- ❖ Од особина модула које су скривене за друге модуле система

Особине које су јавне, могу користити други модули, за разлику од оних које нису јавне.

Сакривање информација представља резултат енкапсулације, јер се сакривају информације које други модули не смеју користити.

Као пример ће послужити класа *Controller*:

```

public class Controller {

    private static Controller instance;
    private AbstractGenericOperation so;

    private Controller() {

    }

    public static Controller getInstance() {
        if (instance == null) {
            instance = new Controller();
        }
        return instance;
    }

    public Response login(Request request) {
        so = new LoginSO();
        return ((LoginSO) so).login(request);
    }

    public Response createGame(Request request) {
        so = new CreateGameSO();
        return ((CreateGameSO) so).createGame(request);
    }

    public Response startGame(Request request) {
        so = new StartGameSO();
        return ((StartGameSO) so).startGame(request);
    }

    public Response userShoot(Request request) {
        so = new UserShootSO();
        return ((UserShootSO) so).userShoot(request);
    }

    public Response serverShoot(Request request) {
        so = new ServerShootSO();
        return ((ServerShootSO) so).serverShoot(request);
    }

    public Response endGame(Request request) {
        so = new EndGameSO();
    }
}

```

Слика 51 Енкапсулација

Ова класа је имплементирана као *Singleton* класа, што значи да ће само једном бити инстанцирана при првом позиву њене јавне методе *getInstance()*.

Наиме, она ставља на располагање своју јавну методу другим класама којима је потребна њена инстанца, али им не дозвољава креирање. Као што је приказано на слици 51. , конструктор ове класе је приватан, што значи да друге класе којима је потребна инстанца класе *Controller*, не знају да ли је она тог тренутка била креирана или не. Оно што је сигурно, је, да ће ту инстанцу добити и да ће инстанца бити јединствена током целог њеног постојања.

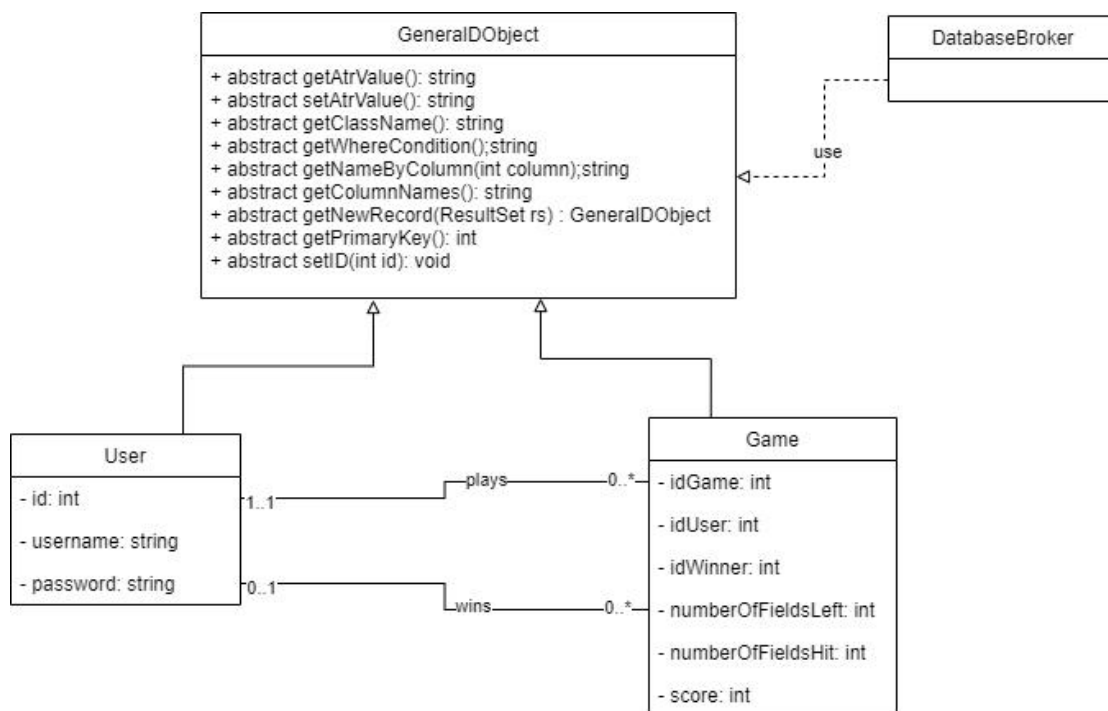
За енкапсулацију можемо рећи да је такође апстракција, јер раздваја нешто што је опште (јавно) од специфичног (приватно).

Такође, применом Ларманове методе у фази анализе описујемо системске операције на високом нивоу апстракције, односно енкапсулирамо начин имплементације који у том моменту још увек не знамо, од онога што у том моменту знамо, а то је – шта та системска операција треба да ради.

Одвајање интерфејса од имплементације

Интерфејс се одваја од имплементације и излаже се клијенту.

Клијент не треба да зна како су операције имплементирани. У примеру, клијент је класа *DatabaseBroker*, а сервер је класа *GeneralDObject*.



Слика 52 Одвајање интерфејса од имплементације

Брокер је тај који позива методе класа које имплементирају класу *GeneralDObject*, не знајући како су методе имплементирани.

Довољност, комплетност и једноставност

Довољност, комплетност и једноставност указују на особине софтверске компоненте коју карактерише једноставан начин одржавања као и надградње, а са друге стране је довољна и комплетна да обезбеди функционалност.

7.2. Стратегије пројектовања

Најпознатије стратегије пројектовања су:

- 1) Подели и победи
- 2) С врха на доле
- 3) Одоздо на горе
- 4) Итеративно-инкрементални приступ

Подели и победи – divide and conquer

Ова стратегија се занима на претходно поменутом принципу декомпозиције који разлаже полазни проблем на потпроблеме, који се независно решавају како би се полазни проблем лакше решио.

Ова стратегија се најчешће примењује у прве три фазе развоја софтвера:

1. *Прикупљање корисничких захтева* – захтеви се описују преко скупа независних случајева коришћења
2. *Анализа* – структура се описује помоћу концептуалног модела, а понашање преко скупа независних системских операција.
3. *Пројектовање* – архитектура софтверског система се дели на три дела: **кориснички интерфејс, апликациону логику и складиште података**. Кориснички интерфејс се даље дели на екранске форме и контролер корисничког интерфејса, а апликациона логика на: контролер апликационе логике, пословну логику и складиште података.

С врха на доле – top down

Ова стратегија се заснива на принципу декомпозиције функција и дели почетну функцију на више подфункција.

Ове подфункције се независно решавају како би олакшале решавање полазне функције.

Пример који је дат се односи на класу *AbstractGenericOperation*:

```
public abstract class AbstractGenericOperation {  
  
    static public IDatabaseBroker bbp = new DatabaseBroker();  
    GeneralDObject gdo;  
  
    synchronized public boolean abstractExecuteSO() {  
        bbp.makeConnection();  
        boolean signal = executeSO();  
        if (signal == true) {  
            bbp.commitTransation();  
        }  
    }  
}
```

```

    } else {
        bbp.rollbackTransation();
    }
    bbp.closeConnection();
    return signal;
}

abstract public boolean executeSO();

}

```

Одоздо на горе

Ова стратегија се заснива на принципу генерализације, који у некој сложеној функцији уочава једну или више логичких целина, које проглашава за функције.

На тај начин је сложена функција декомпонована на више независних функција.

Композиција тих функција треба да обезбеди исту функционалност као и сложена функција.

У овој стратегији као и у претходној, добићемо исто решење. Оно што се разликује је начин доласка до решења. У претходној стратегији уочавамо одмах генералне делове, односно сложене функције које морамо разложити, док у оваквом приступу прво полазимо од веома специфичних делова.

Након што уочимо логичке делове који су специфични, можемо их прогласити за функције.

Итеративно-инкрементални приступ

У итеративно-инкременталном приступу је циљ да се релативно брзо реализује решење, које није неопходно да буде сасвим комплетно. Циљ је да корисник може да види готов производ, поједине функционалности и да изрази своје задовољство односно незадовољство, што би значило да се та функционалност мора мењати.

Овакав приступ је свакако боља опција од тога да се стигне до краја пројекта и да се касно схвати да можда кориснички захтеви нису добро прикупљени или схваћени, што би резултирало незадовољством корисника и великим проблемом у смислу измене целог пројекта.

Како би се проблеми избегли, систем који се развија се дели на више делова (потпројеката), који могу представљати системске операције које се развијају или потпуно независне случајеве коришћења. Сваки део система, подпројекат, пролази кроз више итерација и као резултат једне итерације, добија се инкремент за систем.

На крају се сви потпројекти интегришу у један софтверски систем.

Пример: Прикупљањем корисничких захтева, учили смо системске операције које треба пројектовати тако да су потпуно независне једна од друге. На овакав начин омогућавамо и појаву нових системских операција које неће утицати на постојеће, као и измену постојећих, а да се измене не одразе на друге системске операције.

7.3. Методе пројектовања

Најважније методе пројектовања су:

1. Функционо оријентисано пројектовање
2. Објектно оријентисано пројектовање
3. Пројектовање засновано на структури података
4. Пројектовање засновано на компонентама

1. Функционо оријентисано пројектовање:

Проблем се посматра из перспективе његовог понашања, функционалности.

На овај начин се прво уочавају функције система, затим се одређују структуре података над којима се извршавају те функције.

2. Објектно оријентисано пројектовање:

Засновано је на објектима (класама). Објекти могу да представљају и структуру и понашање софтверског система. Код објектно оријентисаног пројектовања паралелно се развијају и структура и понашање.

Тежи се раздвајању структуре и понашања, јер се жели постићи ефекат независног извршења системских операција над независном структуром система.

3. Пројектовање засновано на структури података:

Проблем посматра из перспективе структуре. Прво се уочава структура система, а затим се дефинишу функције које се извршавају над том структуром.

4. Пројектовање засновано на компонентама:

Проблем посматра из перспективе постојећих компоненти које се могу (поново) користити у решавању проблема. Прво се уочавају делови проблема који се могу реализовати постојећим компонентама, а након тога се имплементирају они делови за које није постојало решење.

Принципи објектно оријентисаног пројектовања (Principles of object orientied class design)

Постоје следећи принципи код објектно оријентисаног пројектовања класа:

1. Принцип отворено затворено
2. принцип замене Барбаре Лисков
3. принцип инверзије зависности
4. принцип уметања зависности
5. принцип издвајања интерфејса

1. *Принцип отворено затворено (Open – closed principle)* : Модул треба да буде отворен за проширење, али и затворен за модификацију.

Пример је класа *AbstractGenericOperation*, која као што је већ речено, има *template* методу, у којој даје редослед извршења операција на вишем нивоу апстракције, а детаље извршења оставља подкласама.

На овај начин, редослед извршења операција је непроменљив, а оно што је отворено за промене је апстрактна метода која дозвољава подкласама да је прошире. Приметићемо да постоји директна веза између *Template Method patterna* и *open-closed* принципа, односно можемо рећи да је *template method pattern* заснован на *open-closed* принципу.

```
public abstract class AbstractGenericOperation {

    static public IDatabaseBroker bbp = new DatabaseBroker();
    GeneralDBObject gdo;

    synchronized public boolean abstractExecuteSO() {
        bbp.makeConnection();
        boolean signal = executeSO();
        if (signal == true) {
            bbp.commitTransation();
        } else {
            bbp.rollbackTransation();
        }
        bbp.closeConnection();
        return signal;
    }

    abstract public boolean executeSO();

}
```

2. *Принцип замене Барбаре Лисков (The Liskov substitution principle):*

подкласе треба да буду заменљиве са њиховим надкласама.

Пример се односи на доменске класе *Игра* и *Играч*, које могу бити заменљиве њиховом надкласом *GeneralDBObject*. Ово се конкретно види у методама брокера базе података који као параметар прима *GeneralDBObject*.

Обезбеђивањем оваквог параметра, обезбеђујемо да нам класа брокера базе података не буде чврсто везана за све доменске класе, већ за њихову апстракцију, што је основна идеја свих патерна пројектовања.

Брокер базе података:

```
public List<GeneralDBObject> findRecords(GeneralDBObject odo, String where) {
    ResultSet rs = null;
    Statement st = null;
    String query = "SELECT * FROM " + odo.getClassName() + " " + where;
    List<GeneralDBObject> ls = new ArrayList<>();

    try {
        st = connection.prepareStatement(query);
        rs = st.executeQuery(query);
        while (rs.next()) {
            ls.add(odot.getNewRecord(rs));
        }
    } catch (SQLException ex) {
        Logger.getLogger(DatabaseBroker.class.getName()).log(Level.SEVERE, null, ex);
    } finally {
        close(null, st, rs);
    }
    return ls;
}
```

Системска операција КреирајИгру:

```
public boolean executeSO() {
    User user = request.getUser();
    Game game = new Game();
}
```

```

game.setidUser(user.getIdUser());
if (bbp.insertRecord(game)) {
    // Here we need to get record from database and set it to game
    List<GeneralDObject> games = bbp.findRecords(new Game(), "WHERE idGame =
(SELECT MAX(idGame) FROM game)");
    if (games.size() != 0) {
        // We take first record
        Client.game = (Game) games.get(0);
        response.setResponseStatus(ResponseStatus.OK);
    } else {
        response.setResponseStatus(ResponseStatus.ERROR);
    }
} else {
    response.setResponseStatus(ResponseStatus.ERROR);
}
response.setOperation(Operation.CREATE_GAME);

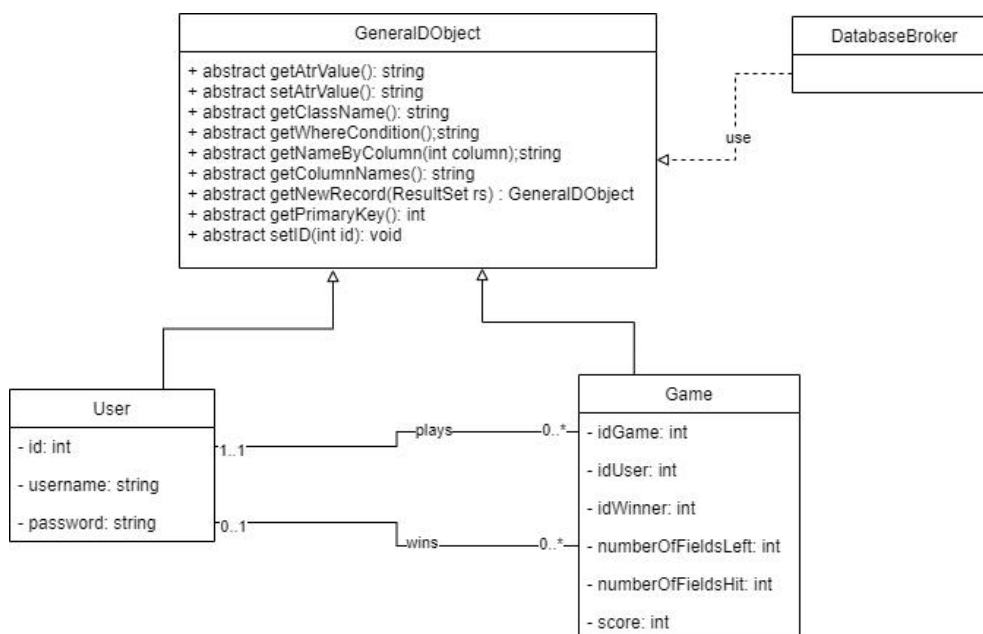
return true;
}

```

3. Принцип инверзије зависности – *The Dependancy inversion principle*

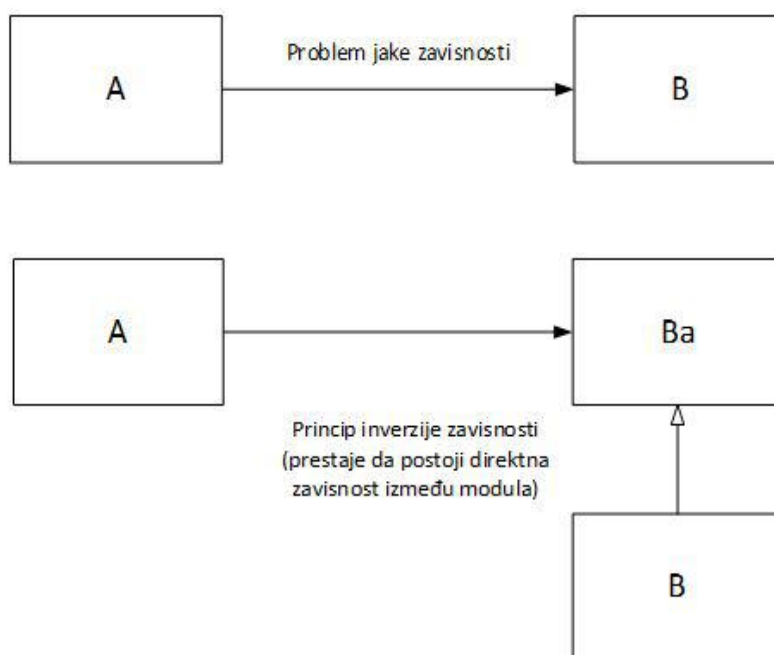
- ❖ Принцип инверзије зависности: зависи од апстракције а не од конкретизације.
- ❖ Модули вишег нивоа не треба да зависе од модула нижег нивоа
- ❖ Оба треба да зависе од апстракције
- ❖ Апстракције не треба да зависе од детаља
- ❖ Детаљи треба да зависе од апстракције

У нашем примеру се избегава зависност модула вишег нивоа: Брокера базе података од модула нижег нивоа – доменских класа. Повезаност се остварује преко класе *GeneralDObject*, коју реализују све доменске класе. На тај начин се брокер базе података посредно повезује са свим доменским класама које реализују класу *GeneralDObject*.



Слика 53 Пример принципа инверзије зависности

У општем случају:



Слика 54 Принцип инверзије зависности – општи случај

Модул А је модул вишег нивоа и он у првом делу слике зависи од модула нижег нивоа – модула В, што је противно овом принципу.

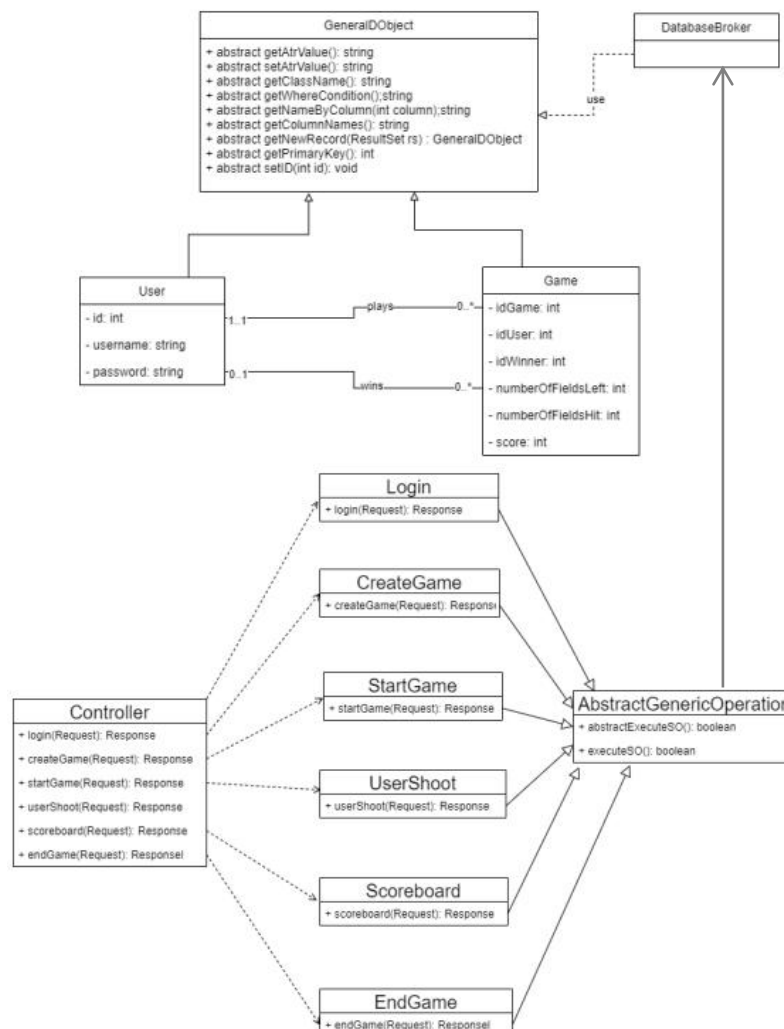
Директна веза између модула А и В се прекида и између њих се поставља модул вишег нивоа – модул Ва.

Можемо да закључимо да постоји велика сличност између принципа инверзије зависности и опште дефиниције патерна.

4. Принцип уметања зависности – The dependency injection principle :

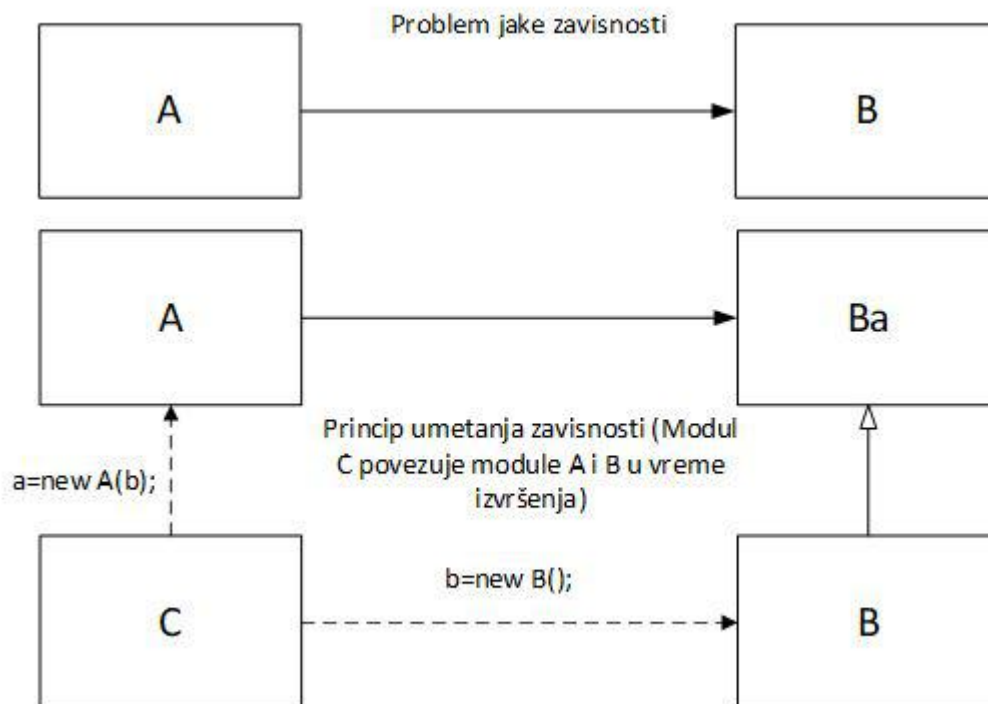
Зависности између две компонентне програма се успостављају у време извршења програма преко неке треће компоненте.

У наведеном примеру, веза између брокера базе података и конкретних доменских класа се успоставља у време извршења програма, преко класа које реализују класу AbstractGenericOperation.



Слика 55 Принцип уметања зависности

Општи случај:



Слика 56 Принцип уметања зависности – општи случај

8. Примена патерна у пројектовању

8.1 Увод у патерне

Прве дефиниције патерна настају опажањем структура градова и грађевина, дакле у грађевинарству, заслугом Кристофера Александера који је и дао први значајан допринос у дефинисању патерна.

Једна од његових дефиниција гласи: *“Сваки патерн је троделно правило, које успоставља релацију између неког проблема, његовог решења и његовог контекста.*

Патерн је у исто време и ствар, која се дешава у стварности, и правило које говори када и како се креира наведена ствар”.

Оно што је Александер открио је да се за сваки проблем који се више пута понавља на различит начин, постоји неко решење које је применљиво за цео скуп сличних проблема.

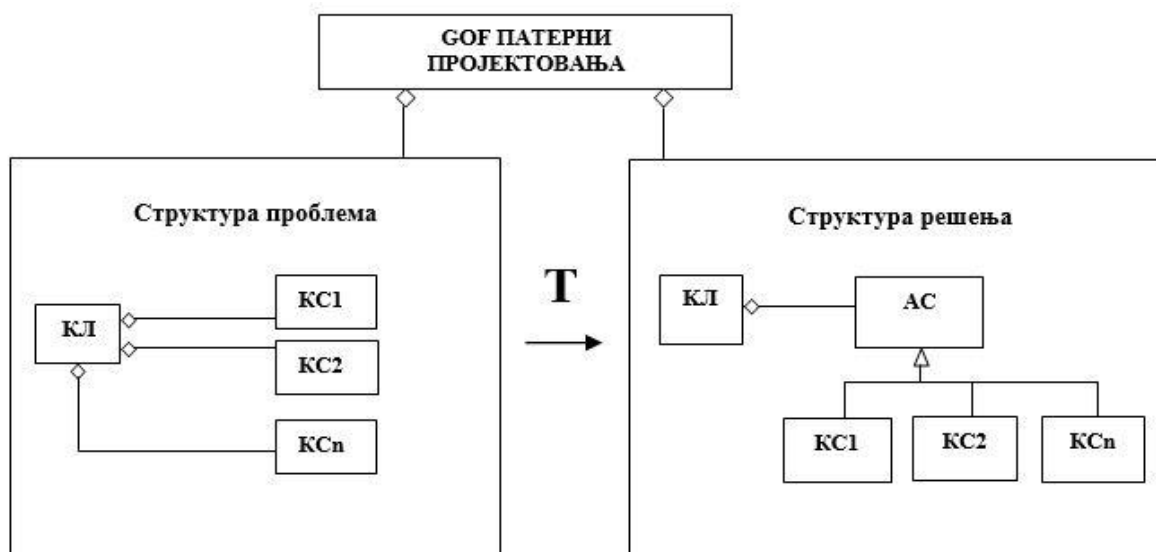
Оно што изводимо као закључак је да је патерн “троделно правило” које успоставља корелацију између одређеног контекста као система ограничења, проблема који делује у том контексту и решења, као структуре софтверског система која омогућава да односи између елемената тог система буду поново употребљиви.

8.2 Општи облик GOF патерна пројектовања

Када говори о претходно наведеним дефиницијама патерна, Кристофер Александер инсистира и на особини поновне употребљивости патерна и каже: *“Сваки патерн описује проблем који се јавља изнова (непрестано) у нашем окружењу, а затим описује суштину решења тог проблема на такав начин да ви можете користити ово решење милион пута а да никада то не урадите на два пута на исти начин”* [2], што значи да се већ једном пронађено решење може применити на више различитих проблема, који су из истог контекста.

Патерн као процес, трансформацијом структуре проблема у структуру решења обезбеђује дугорочност, стабилност, флексибилност и могућност даљег развоја, што и јесте примарна дефиниција одрживости, у контексту софтверских система.

Оваквом трансформацијом, раздвајају се специфичности које обезбеђују различитости у софтверском систему, и које су на самом почетку развоја софтверског система биле помешане са генералним деловима, које програму обезбеђују универзалност, што и јесте нешто чему временом тежи сваки софтверски систем.



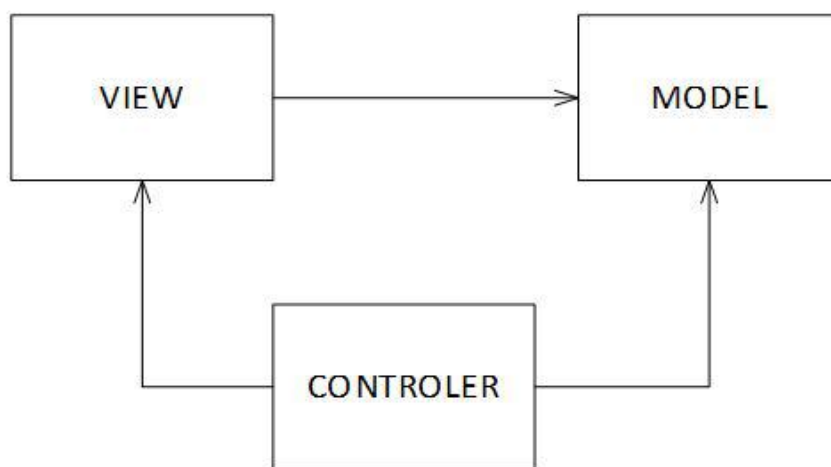
Слика 57 Општи облик GOF патерна пројектовања

Постоје патерни *макро* архитектуре и *микро* архитектуре.

Макро архитектура описује структуру и организацију софтверског система на највишем нивоу.

MVC је макроархитектурни патерн који дели софтверски систем на три дела:

1. View – обезбеђује кориснику интерфејс (екранску форму) помоћу које ће корисник да унесе податке и позива одговарајуће операције које треба да се изврше над моделом
2. Controller – ослушкује и прихвата захтев од клијента за извршење операције. Након тога позива операцију која је дефинисана у моделу. Уколико модел промени стање, *controller* извештава *view* о томе.
3. Model – представља стање система. Стање модела мењају неке од операција модела.



Слика 58 MVC патерн

Поглед (*view*) је задужен да омогући кориснику позив операција над моделом, има информације о моделу, док контролер има информацију и о погледу и о моделу. Он извршава промене над моделом које поглед захтева и обавештава поглед о тим променама.

Модел не мора да има било какву информацију о контролеру и погледу, он само служи да чува информације о својим стањима. Може се и рећи да модел представља апликациону логику, поглед екранску форму, а контролер представља “лепак” између њих.

Уколико говоримо у контексту клијента и сервера, код MVC патерна, модел би представљао сервер, док су контролер и поглед клијенти.

Микро архитектурни патерни се сврставају у три категорије:

- ❖ Креациони патерни
- ❖ Патерни структуре
- ❖ Патерни понашања

Креациони патерни апстрахују процес креирања објеката. Они дају велику флексибилност у томе шта ће бити креирано, ко ће то креирати, како и када.

У ову подврсту патерна се сврставају: *Abstract Factory, Builder, Factory Method, Prototype, Singleton*.

Структурни патерни описују сложене структуре међусобно повезаних класа и објеката. У њих се сврставају: *Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy* патерн.

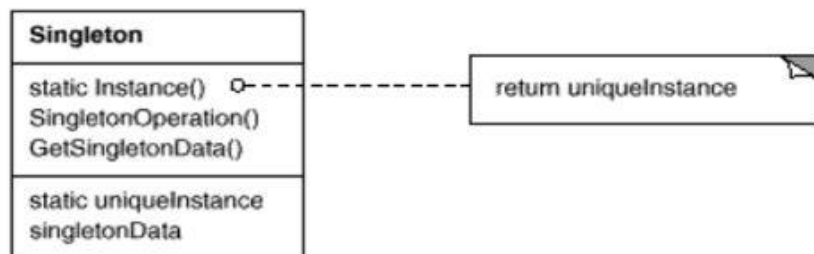
Патерни понашања описују начин на који класе и објекти сарађују и распоређују одговорности. У њих се сврставају *Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor* патерн.

У наставку ће бити дати примери примена неких од наведених патерна:

Singleton патерн

Захтев: Потребно је обезбедити јединствену комуникацију између клијента и сервера. Неопходно је да се клијент само једном повеже са сервером, остварујући комуникацију са сервером путем класе *Controller*. Онемогућити поновно повезивање у току једног покретања програма.

Решење: *Singleton* патерн обезбеђује класи само једно појављивање и јединствен приступ до ње. То омогућава статичка метода *Instance()*.



Слика 59 Singleton патерн

У примеру је дата имплементација класе *Controller*:

```

public class Controller {

    private static Controller instance;
    private AbstractGenericOperation so;

    private Controller() {

    }

    public static Controller getInstance() {
        if (instance == null) {
            instance = new Controller();
        }
        return instance;
    }
}

```

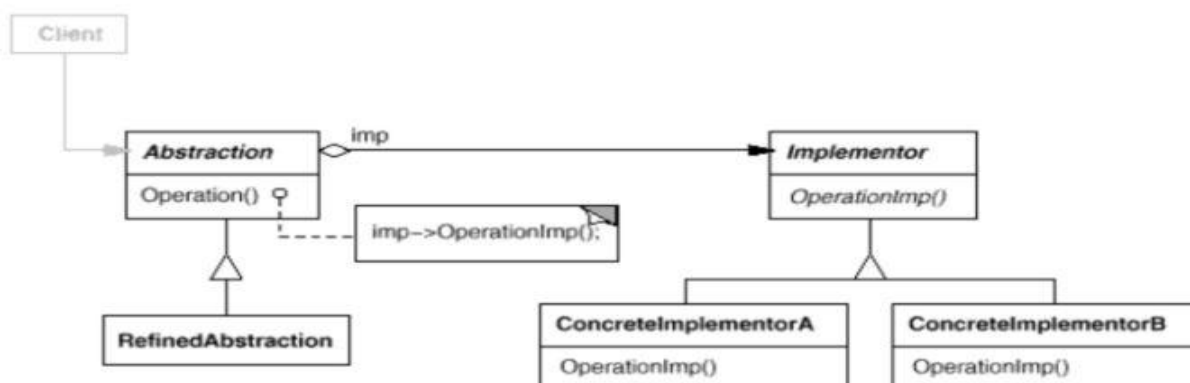
Слика 60 Singleton патерн – Controller

Ова класа је имплементирана преко *Singleton* патерна и на тај начин је омогућено да се при сваком позиву методе *getInstance()*, добије једна иста инстанца ове класе, која ће бити креирана при првом позиву а помоћу које ће се вршити позивање системских операција.

Bridge патерн

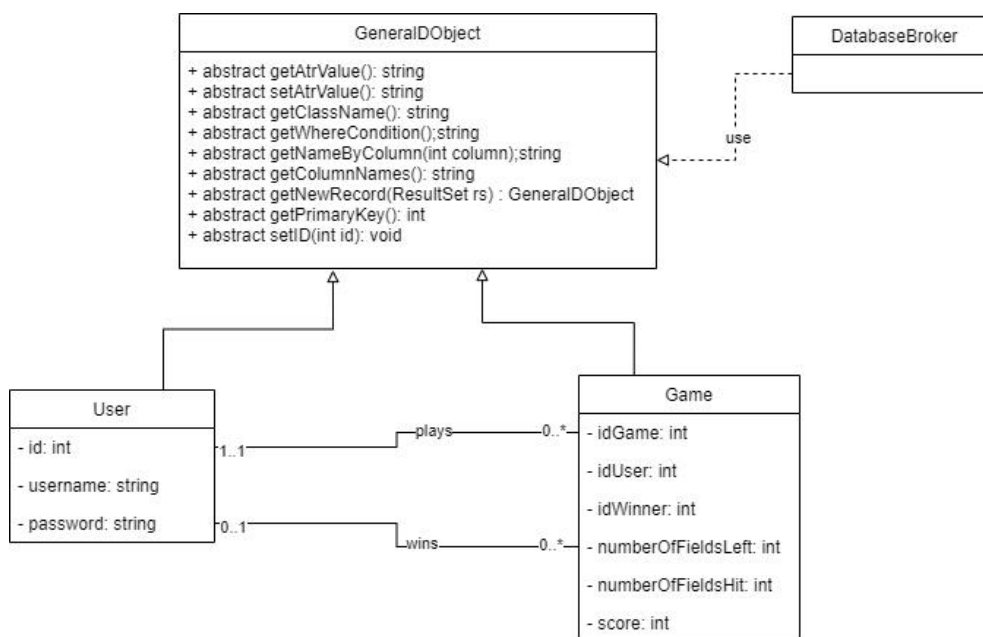
Захтев: Потребно је обезбедити апстрактне операције у класи брокера базе података, за које ће се тек у време извршења програма везати конкретне имплементације. Обезбедити генеричке методе за све доменске класе.

Решење: *Bridge* патерн - декуплује (одваја) апстракцију од њене имплементације, тако да се оне могу мењати независно.



Слика 63: Bridge патерн

У примеру наводимо класу *DatabaseBroker*, која као параметар у својим методама прима генерички објекте класе *GeneralDBObject*, над којима позива одређене методе при извршењу.



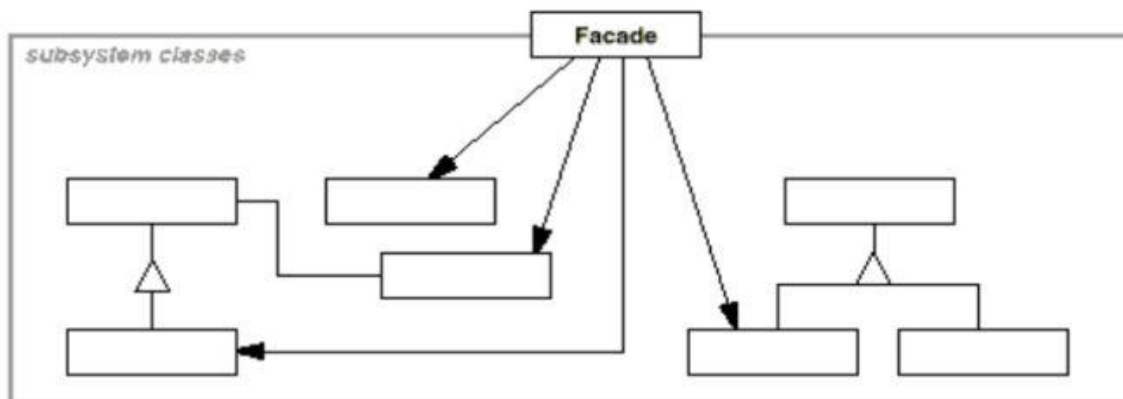
Слика 61 Bridge патерн – брокер базе података

Са приложених слика можемо закључити да је однос између брокера базе података и класе *GeneralDBObject*, заправо однос између *Abstraction*-а и *Implementora*, односно закључујемо да брокер базе података има улогу *Abstraction*-а, а доменске класе улогу *ConcreteImplementora*-а.

Facade патерн

Захтев: Потребно је обезбедити корисницима да помоћу екранских форми позивају различите операције система. Сервер треба да ослушкује повезивање клијента и креира посебну клијентску нит која ће наставити да ослушкује захтеве од повезаног клијента. На овај начин је потребно обезбедити да сервер само усмерава захтеве ка клијентској нити, не да их обрађује.

Решење: **Facade** патерн – обезбеђује јединствен интерфејс за скуп интерфејса неког подсистема. *Facade* патерн дефинише интерфејс високог нивоа који омогућава да се систем лакше користи.



Слика 62 Facade патерн

Овај патерн омогућава да се обезбеди јединствена тачка уласка у неки подсистем и на тај начин олакшава клијенту да не мора водити рачуна о ономе што се дешава иза те тачке. Све што је иза фасаде, клијент не треба да зна.

Пример који је дат се односи на класу *Server*, која представља јединствену тачку улаза у систем и у којој се делегирају захтеви ка одговорним класама за њихову обраду, тј. класи *Client*.

```
public class Server extends Thread {

    private int portNumber;

    public Server(int portNumber) {
        this.portNumber = portNumber;
    }

    @Override
    public void run() {
        try {
            // Listening to users connections
            ServerSocket serverSocket = new ServerSocket(portNumber);
            System.out.println("Listening for connections...");

            while (true) {
                // Accept client
                Socket clientSocket = serverSocket.accept();
                System.out.println("User accepted: " + clientSocket.getInetAddress().toString());

                // Now with client socket we can create new client thread
                Client client = new Client(clientSocket);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```

        client.start();
    }
} catch (IOException ex) {
    // Error creating server socket
    Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
}
}
}
}

public class Server extends Thread {

    private int portNumber;

    public Server(int portNumber) {
        this.portNumber = portNumber;
    }

    @Override
    public void run() {
        try {
            // Listening to users connections
            ServerSocket serverSocket = new ServerSocket(portNumber);
            System.out.println("Listening for connections...");

            while (true) {
                // Accept client
                Socket clientSocket = serverSocket.accept();
                System.out.println("User accepted: " + clientSocket.getInetAddress().toString());

                // Now with client socket we can create new client thread
                Client client = new Client(clientSocket);
                client.start();
            }
        } catch (IOException ex) {
            // Error creating server socket
            Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

```

```

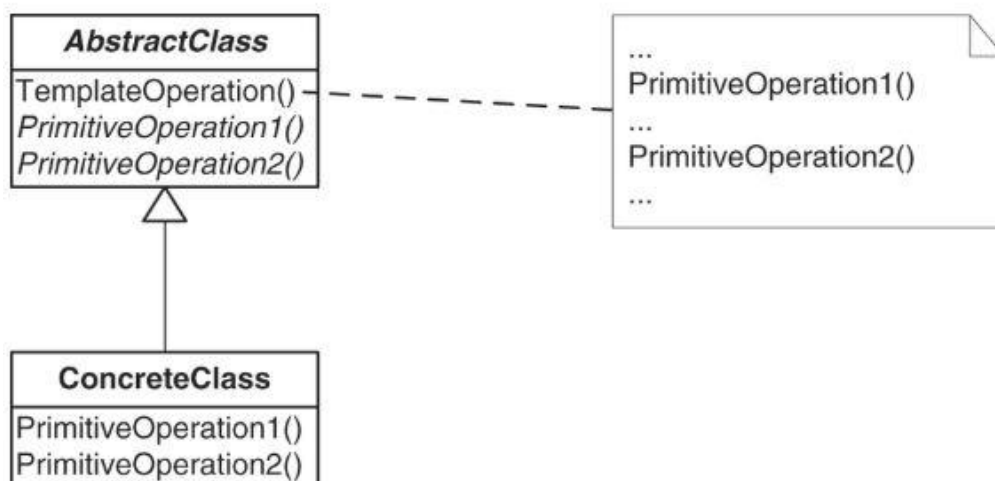
    }
}
}

```

Template method патерн

Захтев: Потребно је логички груписати све методе које се понављају при извршењу сваке системске операције у једну апстрактну методу и дефинисати редослед извршења. Неке методе је потребно имплементирати, јер је понашање исто за све системске операције, а неке методе је потребно оставити подкласама да имплементирају.

Решење: **Template method** патерн- Дефинише скелет алгоритма у операцији, препуштајући извршење неких корака операција подкласама. Овај патерн омогућава подкласама да редефинишу неке од корака алгоритма без промене алгоритамске структуре.



Слика 63 Template method патерн

На овај начин, апстрактна класа дефинише неку *template* (шаблон) операцију која садржи друге примитивне операције, које се проглашавају апстрактним како би се омогућило да их подкласе редефинишу. Примећујемо да на овај начин обезбеђујемо да ниједна подкласа не може променити редослед операција – затворен је за промену, али може проширити примитивне операције -отворене за проширења, што је заправо прави показатељ директне везе између *open-closed* принципа и *template method* патерна.

У примеру је приказана класа *AbstractGenericOperation* и њена *template* метода - *abstractExecuteSO()*:

```

public abstract class AbstractGenericOperation {

    static public IDatabaseBroker bbp = new DatabaseBroker();

```

```

GeneralDObject gdo;

synchronized public boolean abstractExecuteSO() {
    bbp.makeConnection();
    boolean signal = executeSO();
    if (signal == true) {
        bbp.commitTransation();
    } else {
        bbp.rollbackTransation();
    }
    bbp.closeConnection();
    return signal;
}
abstract public boolean executeSO();
}

```

Класе које представљају конкретне системске операцију редефинишу само методу *executeSO()* – примитивну операцију.

```

public class LoginSO extends AbstractGenericOperation {

    Request request;
    Response response;

    public Response login(Request request) {
        this.request = request;
        this.response = new Response();
        abstractExecuteSO();
        return response;
    }

    @Override
    public boolean executeSO() {
        User user = (User) bbp.findRecord(this.request.getUser());
        this.response.setUser(user);
        this.response.setOperation(Operation.LOGIN);
        return true;
    }
}

```

9. Литература

[1] др Синиша Влајић, *Софтверски процес (Скрипта)*, Београд, 2016

[2] Синиша Влајић: *Софтверски патерни*, Издавач Златни пресек, ISBN: 978-86-86887-30-6, Београд, 2014.