

УНИВЕРЗИТЕТ У БЕОГРАДУ
ФАКУЛТЕТ ОРГАНИЗАЦИОНИХ НАУКА
КАТЕДРА ЗА СОФТВЕРСКО ИНЖЕЊЕРСТВО

СЕМИНАРСКИ РАД
Софтверски процес

Тема:

Развој игре “Минско поље”
применом Ларманове методе

Ментор:

проф. др Синиша Влајић

Студент:

Марија Јеремић 3720/2021

Београд, 2022. године

Садржај

1. Кориснички захтеви	2
1.1. Вербални опис	2
1.2. Модел случајева коришћења	3
1.3. Случајеви коришћења	4
1.3.1. СК1: Случај коришћења - Пријављивање корисника	4
1.3.2. СК2: Случај коришћења - Регистровање корисника	5
1.3.3. СК3: Случај коришћења – Учитавање нивоа	6
1.3.4. СК4: Случај коришћења – Чување резултата игре	7
1.3.5. СК5: Случај коришћења – Чување корисничког најбољег резултата	8
2. Анализа	9
2.1. Понашање софтверског система - Системски дијаграм секвенци	9
2.1.1. Дијаграм секвенци случаја коришћења 1 - Пријављивање корисника	10
2.1.2. Дијаграм секвенци случаја коришћења 2 – Регистровање корисника	12
2.1.3. Дијаграм секвенци случаја коришћења 3 - Учитавање нивоа	14
2.1.4. Дијаграм секвенци случаја коришћења 4 – Чување резултата игре	16
2.1.5. Дијаграм секвенци случаја коришћења 5 – Чување корисничког најбољег резултата	18
2.2. Понашање софтверског система - Дефинисање уговора о системским операцијама	20
2.3. Структура софтверског система - Концептуални модел	22
2.4. Структура софтверског система - Релациони модел	23
3. Пројектовање	26
3.1. Архитектура софтверског система	26
3.2. Пројектовање корисничког интерфејса	27
3.2.1. СК1: Случај коришћења - Пријављивање корисника	28
3.2.2. СК2: Случај коришћења - Регистровање корисника	30
3.2.3. СК3: Случај коришћења – Учитавање нивоа	32
3.2.4. СК4: Случај коришћења – Чување резултата игре	35
3.2.5. СК5: Случај коришћења – Чување корисничког најбољег резултата	37
3.3. Комуникација сервер - клијент	39
3.4. Контролер апликационе логике	42
3.5. Пројектовање структуре софтверског система - Доменске класе	43
3.6. Пројектовање пословне логике	51
3.7. Пројектовање брокера базе података	56
3.8. Пројектовање складишта података	58
4. Имплементација	59
4.1. Механизам рефлексije	62
5. Тестирање	63
6. Закључак	64
7. Принципи, методе и стратегије пројектовања софтверског система	65
7.1. Принципи пројектовања софтверског система	65
7.1.1. Апстракција	65
7.2. Стратегије пројектовања	82
7.3. Методе пројектовања	84
7.4. Принципи објектно оријентисаног пројектовања (Principles of object oriented class design)	86
8. Примена патерна у пројектовању	92
8.1 Увод у патерне	92
8.2 Општи облик GOF патерна пројектовања	92
9. Литература	104

1. Кориснички захтеви

1.1. Вербални опис

Потребно је направити софтверску апликацију која ће симулирати логичку видео игрицу „Минско поље”. Да би корисник могао да покрене игрицу неопходно је да се кориснику омогући пријављивање на систем уношењем корисничког имена и лозинке, за шта је неопходно да корисник постоји у систему. Уколико корисник нема налог, региструје се уношењем корисничког имена, лозинке и имејл адресе. Након пријаве корисника на систем потребно је омогућити започињање нове игре.

Приликом избора креирање нове игре од стране корисника, поставља се „нормална“ тежина игре. Кориснику је потребно омогућити промену тежине игре на „лако“ или „тешко“ уколико је тренутна претешка или прелака. Одабиром тежине од понуђених, игра се започиње.

У свакој игри потребно је да се бележи корисников резултат, уколико је победио. При чему је онда потребно чување његовог најбољег резултат.

1.2. Модел случајева коришћења

Функционални захтеви које софтверски систем треба да обезбеди се описује преко модела случајева коришћења. Овакав принцип се користи код упрошћене Ларамнове методе развоја софтвера.

На основу дефинисаног вербалног описа, у овом конкретном случају идентификовани су следећи случајеви коришћења:

1. Пријава корисника
2. Регистрација корисника
3. Учитавање нивоа
4. Чување резултата игре
5. Чување корисниковог најбољег резултата



Слика 1. Дијаграм случаја коришћења

1.3. Случајеви коришћења

1.3.1. СК1: Случај коришћења - Пријава корисника

Назив СК

Пријава корисника

Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

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

Основни сценарио СК:

1. **Корисник** уноси податке за пријаву. (АПУСО)
2. **Корисник** контролише да ли је коректно унео податке. (АНСО)
3. **Корисник** позива систем да изврши пријаву корисника. (АПСО)
4. **Систем** врши пријаву корисника. (СО)
5. **Систем** приказује кориснику поруку: „Login successful!” и омогућава приступ систему. (ИА)

Алтернативна сценарија:

- 5.1. Уколико систем не пронађе корисника са унетим подацима, приказује кориснику поруку: „Login unsuccessful!”. (ИА)

1.3.2. СК2: Случај коришћења - Регистрација корисника

Назив СК

Регистрација корисника

Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен и приказује форму за регистрацију корисника.

Основни сценарио СК:

1. **Корисник** уноси податке за регистрацију. (АПУСО)
2. **Корисник** контролише да ли је коректно унео податке. (АНСО)
3. **Корисник** позива систем да изврши регистрацију корисника. (АПСО)
4. **Систем** врши регистрацију корисника. (СО)
5. **Систем** приказује кориснику поруку: „Registration successful!” и омогућава приступ систему. (ИА)

Алтернативна сценарија:

- 5.1. Уколико систем не може да региструје корисника са унетим подацима, приказује кориснику поруку: „Registration unsuccessful!”. (ИА)

1.3.3. СК3: Случај коришћења – Учитавање нивоа

Назив СК

Учитавање нивоа

Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен. Корисник је пријављен на систем и приказује форму за бирање тежине.

Основни сценарио СК:

1. **Корисник** бира тежину нивоа која му одговара. (АПУСО)
2. **Корисник** контролише да ли бира коректну тежину. (АНСО)
3. **Корисник** позива систем да учита ниво. (АПСО)
4. **Систем** врши учитавање игре. (СО)
5. **Систем** приказује играчу поруку: „Your game has started!”. (ИА)

Алтернативна сценарија:

- 5.1. Уколико систем не може да започне игру, приказује поруку: „Your game cannot start.”. (ИА)

1.3.4. СК4: Случај коришћења – Чување резултата игре

Назив СК

Чување резултата игре

Актори СК

Корисник

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** позива систем да сачува резултат. (АПСО)
2. **Систем** чува резултат игре. (СО)
3. **Систем** приказује кориснику резултат и поруку: „Your game record has been saved.“. (ИА)

Алтернативна сценарија:

- 3.1. Уколико систем не може да сачува резултат, приказује поруку: „System cannot save the game record.“. (ИА)

1.3.5. СК5: Случај коришћења – Чување корисниковог најбољег резултата

Назив СК

Чување корисниковог најбољег резултата

Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

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

Основни сценарио СК:

1. **Корисник** позива систем да сачува нови најбољи резултат корисника. (АПСО)
2. **Систем** чува резултат игре. (СО)
3. **Систем** приказује кориснику резултат и поруку: „Your highscore has been saved.“. (ИА)

Алтернативна сценарија:

- 3.1. Уколико систем не може да сачува резултат, приказује поруку: „System cannot save your highscore.“. (ИА)

2. Анализа

Фаза анализе подразумева дефинисање логичке структуре и понашања софтверског ситета, односно дефинисање пословне логике ситета. [1]

За описивање понашања софтверског ситета користе се системски дијаграми секвенци и уговори о системским операцијама.

Структура софтверског ситета је описана помоћу концептуалног и релационог модела.

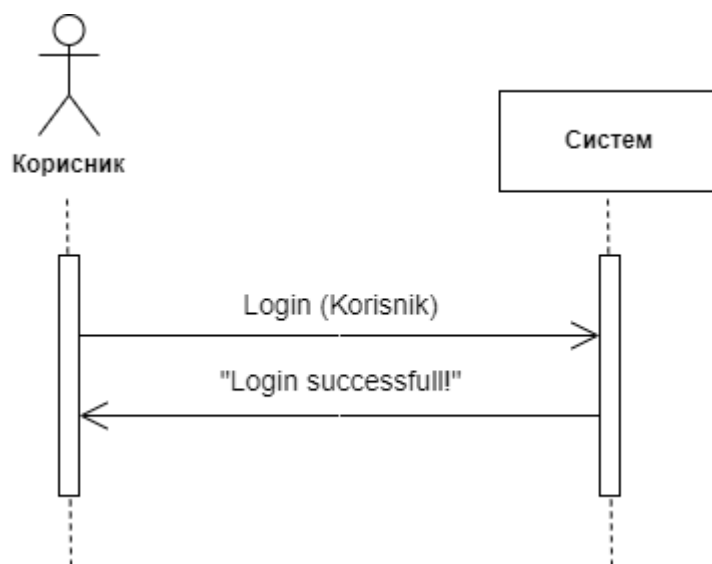
2.1. Понашање софтверског ситета - Системски дијаграм секвенци

За сваки случај коришћења се пројектује дијаграм секвенци случајева коришћења, који даје преглед секвенци порука које се размењују између актора и софтверског ситета.

2.1.1. Дијаграм секвенци случаја коришћења 1 - Пријава корисника

Основни сценарио СК:

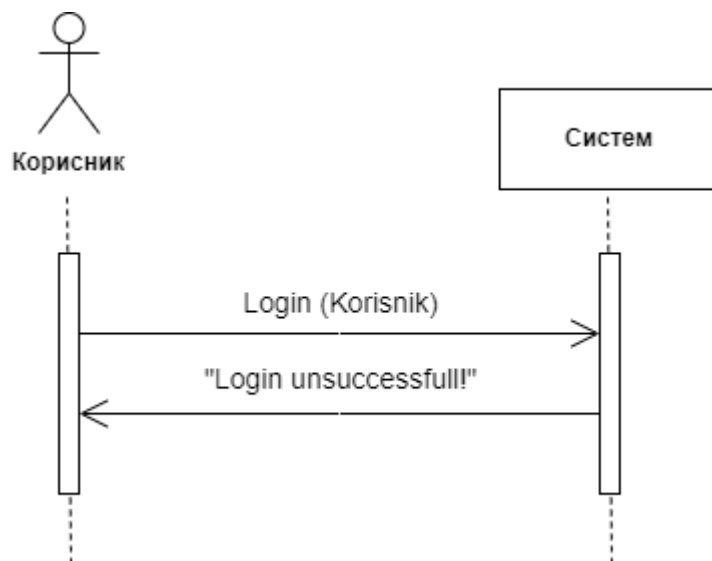
1. **Корисник** позива систем да изврши пријаву корисника. (АПСО)
2. **Систем** приказује кориснику поруку: „Login successful!” и омогућава приступ систему. (ИА)



Слика 2. Основни сценарио СК1

Алтернативна сценарија:

2.1 Уколико систем не пронађе корисника са унетим подацима, приказује кориснику поруку: „Login unsuccessful!”. (ИА)



Слика 3. Први алтернативни сценарио СК1

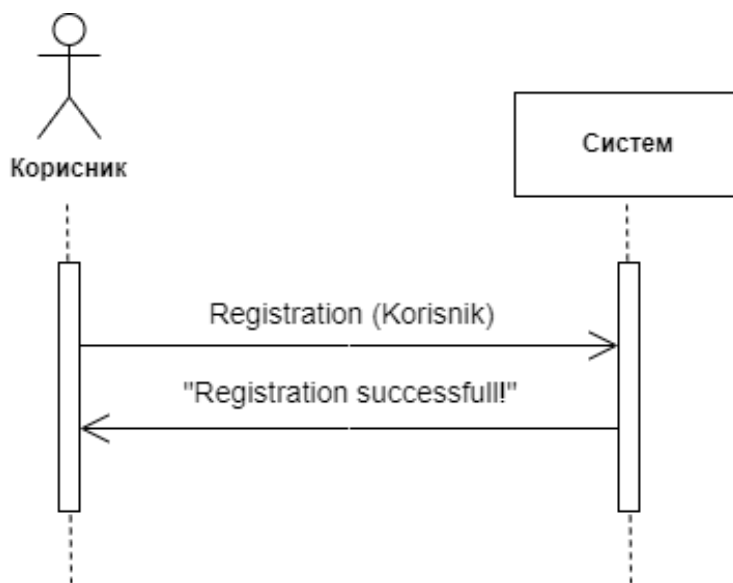
Са наведених секвенцих дијаграма уочава се једна системска операција коју треба пројектовати:

1. Сигнал Login(Korisnik);

2.1.2. Дијаграм секвенци случаја коришћења 2 – Регистрација корисника

Основни сценарио СК:

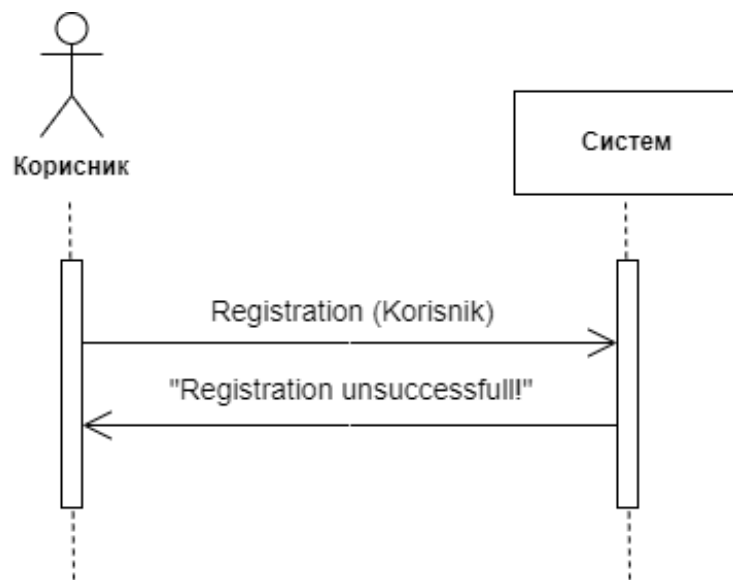
1. **Корисник** позива систем да изврши регистрацију корисника. (АПСО)
2. **Систем** приказује кориснику поруку: „Registration successful!” и омогућава приступ систему. (ИА)



Слика 4. Основни сценарио СК2

Алтернативна сценарија:

2.1. Уколико систем не може да региструје корисника са унетим подацима, приказује кориснику поруку: „Registration unsuccessful!”. (ИА)



Слика 5. Први алтернативни сценарио СК2

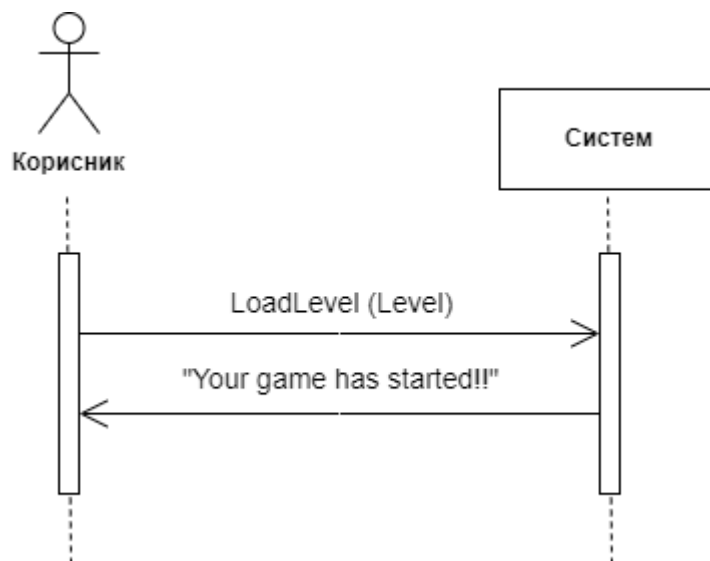
Са наведених секвенчних дијаграма уочава се једна системска операција коју треба пројектовати:

1. Сигнал Registration(Korisnik)

2.1.3. Дијаграм секвенци случаја коришћења 3 - Учитавање нивоа

Основни сценарио СК:

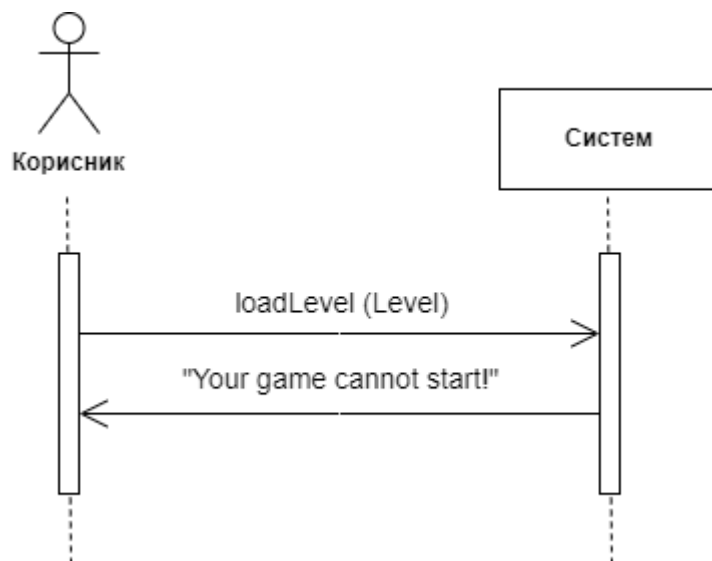
1. **Корисник** позива систем да учита ниво. (АПСО)
2. **Систем** приказује кориснику поруку: „Your game has started!“. (ИА)



Слика 6. Основни сценарио СК3

Алтернативна сценарија:

2.1. Уколико систем не може да започне игру, приказује поруку: „Your game cannot start!“. (ИА)



Слика 7. Први алтернативни сценарио СКЗ

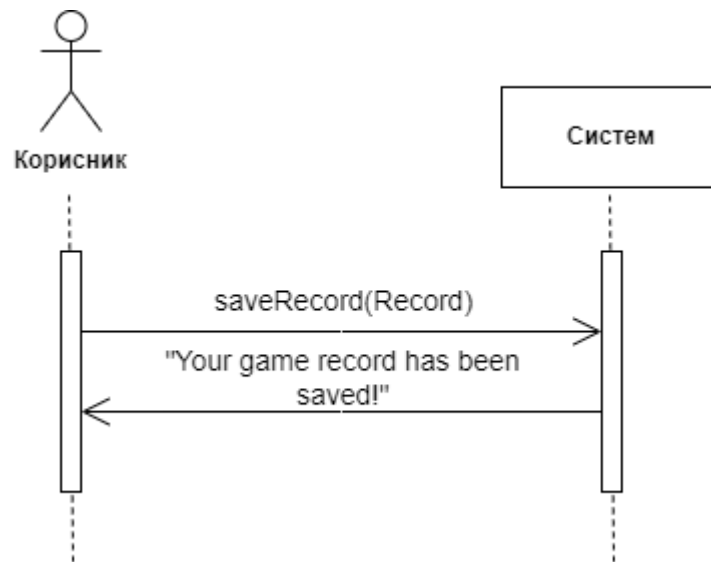
Са наведених секвенчних дијаграма уочава се једна системска операција коју треба пројектовати:

1. Сигнал loadLevel(Level);

2.1.4. Дијаграм секвенци случаја коришћења 4 – Чување резултата игре

Основни сценарио СК:

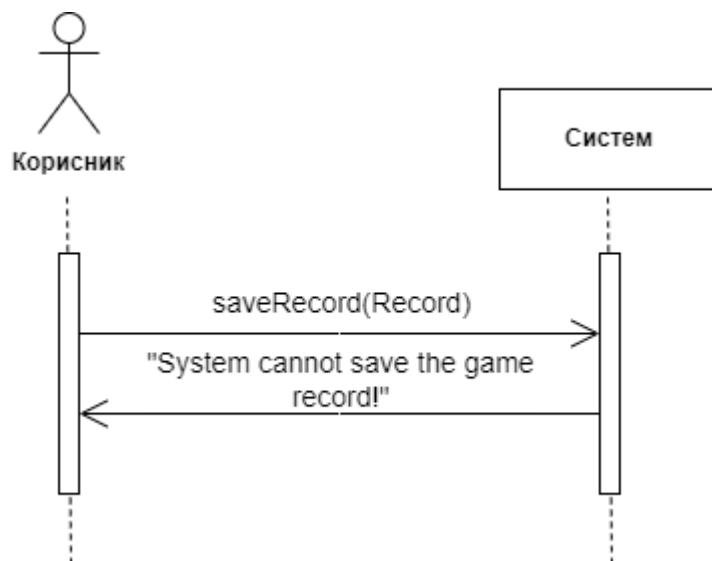
1. **Корисник** позива систем да сачува резултат. (АПСО)
2. **Систем** приказује кориснику резултат и поруку: „Your game record has been saved!“. (ИА)



Слика 8. Основни сценарио СК4

Алтернативна сценарија:

2.1. Уколико систем не може да сачува резултат, приказује поруку: „System cannot save the game record!“ (ИА)



Слика 9. Први алтернативни сценарио СК4

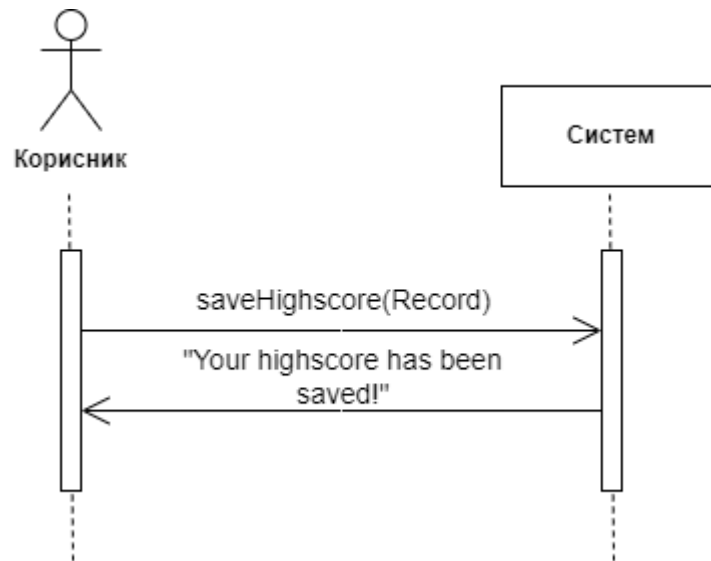
Са наведених секвенцих дијаграм уочава се једна системска операција коју треба пројектова:

1. Сигнал saveRecord(Record)

2.1.5. Дијаграм секвенци случаја коришћења 5 – Чување корисниковог најбољег резултата

Основни сценарио СК:

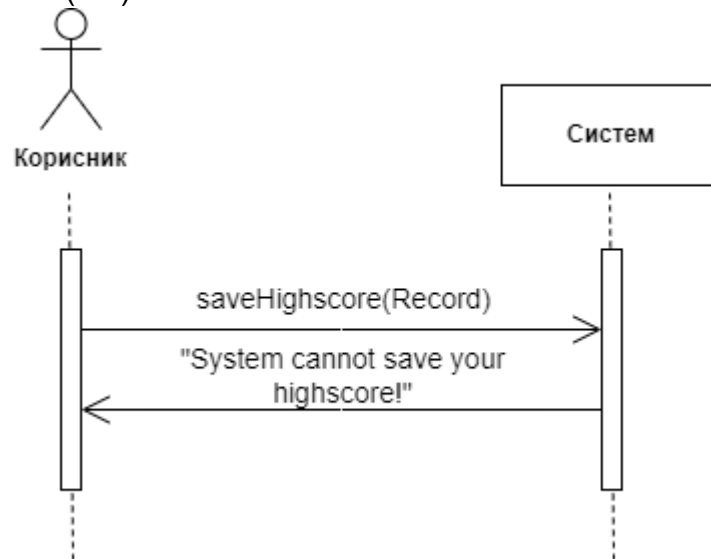
1. **Корисник** позива систем да сачува нови најбољи резултат корисника. (АПСО)
2. **Систем** приказује кориснику резултат и поруку: „Your highscore has been saved!“. (ИА)



Слика 10. Основни сценарио СК5

Алтернативна сценарија:

2.1. Уколико систем не може да сачува резултат, приказује поруку: „System cannot save your highscore!“. (ИА)



Слика 11. Први алтернативни сценарио СК5

Са наведених дијаграма уочене су следеће системске операције које треба пројектовати:

1. Сигнал saveHighscore(Record)

Као резултат анализе добијено је укупно 5 системских операција које треба пројектовати:

1. Сигнал Login(Korisnik)

2. Сигнал Registration(Korisnik)

3. Сигнал loadLevel(Level)

4. Сигнал saveRecord(Record)

5. Сигнал saveHighscore(Record)

2.2. Понашање софверског система - Дефинисање уговора о системским операцијама

1. Уговор УГ1: *Login*

Операција: Login(Korisnik): сигнал

Веза са СК: СК1

Предуслови: /

Постуслови: Корисник је пријављен.

2. Уговор УГ2: *Registration*

Операција: Registration(Korisnik): сигнал

Веза са СК: СК2

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

Постуслови: Корисник је регистрован.

3. Уговор УГ3: *LoadLevel*

Операција: loadLevel(Level): сигнал

Веза са СК: СК3

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

Постуслови: Игра је почела.

4. Уговор УГ4: *SaveRecord*

Операција: saveRecord(Record): сигнал

Веза са СК: СК4

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

Постуслови: Резултат је сачуван.

5. Уговор УГ5: *SaveHighscore*

Операција: saveHighscore(Record): сигнал

Веза са СК: СК5

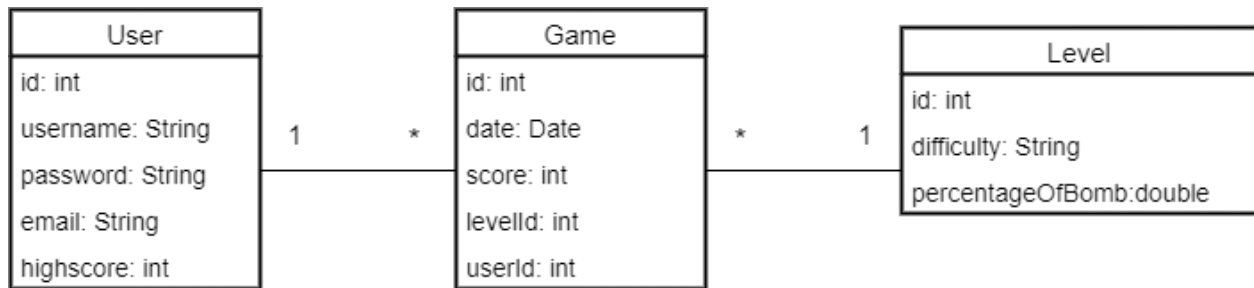
Предуслови: Вредносна и структурна ограничења над објектом Record морају бити задовољена.

Постуслови: Корисников најбољи резултат је сачуван.

2.3. Структура софтверског система - Концептуални модел

Концептуални (доменски) модел помоћу концептуалних класа – доменских објеката и асоцијација између концепталних класа, описује домен проблема.

Доменски модел студијског проблема наводим у наставку:



Слика 12. Структура софтверског система - Концептуални модел

2.4. Структура софтверског система - Релациони модел

На основу концептуалног модела се утврђују релације између објеката. У студијском примеру, утврђене су следеће релације између објеката:

User (id, username, password, email, highscore)

Level (id, difficulty, percentageOfBomb)

Game (id score, date, *levelId*, *userId*)

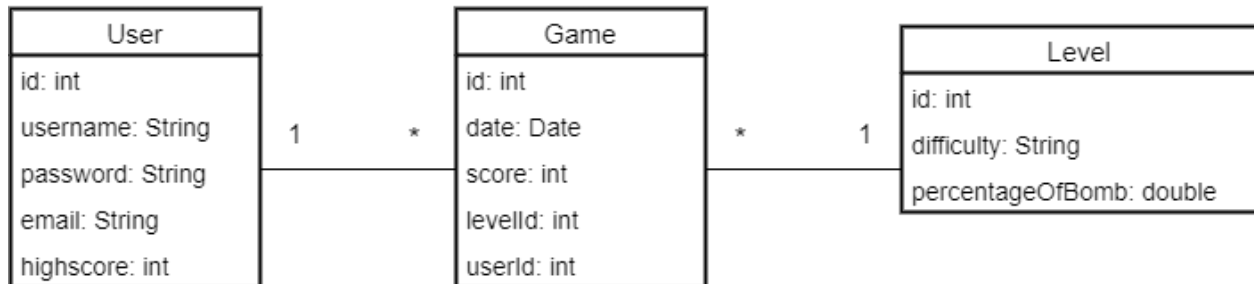
Табела User		Просто вредносно ограничење		Сложено вредносно ограничење		Структурно ограничење
Атрибути	Име	Тип атрибута	Вредност атрибута	Међузав. Атрибута једне табеле	Међузав. Атрибута више табела	INSERT /
	id	Integer	Not null and > 0			UPDATE /
	username	String	Not null			DELETE
	password	String	Not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

Табела Level		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 CASCADE Game
	id	Integer	Not null and > 0			
	difficulty	String	Not null			
	percentageOfBomb	Double	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 RESTRICT User Level
	id	Integer	Not null and > 0			
	score	Integer	Not null			UPDATE RESTRICT User Level
	date	Date	Not null			
	userId	Integer	Not null and > 0			
	levelId	Integer	Not null and > 0			DELETE RESTRICT User Level

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

Структура



Понашање система



Слика 13. Структура и понашање софтверског система

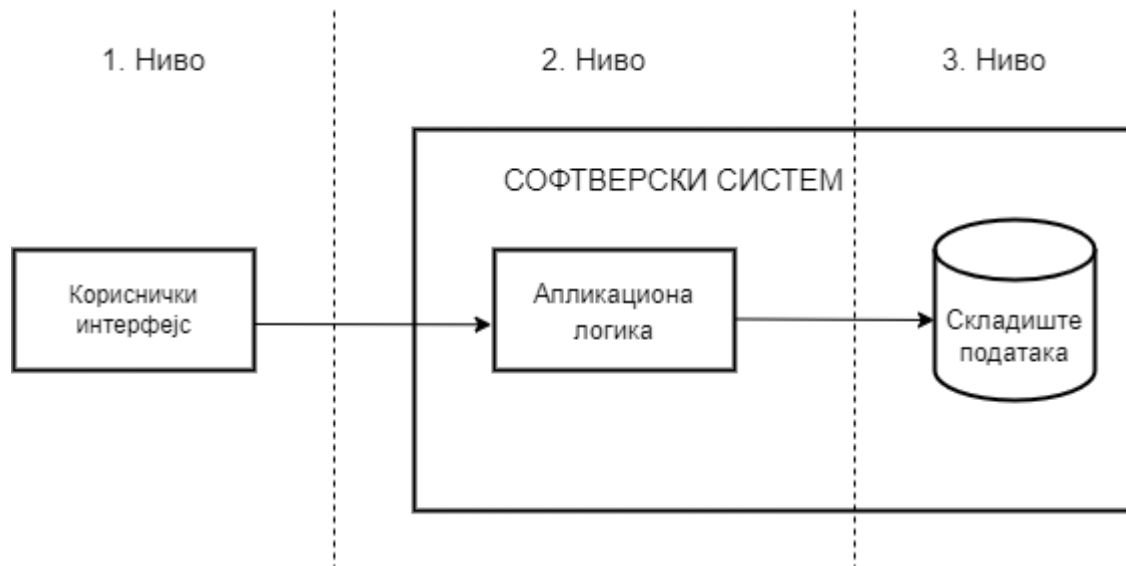
3. Пројектовање

Пројектовање софтверског ситета обухвата описивање физичке структуре и понашање софтверског ситета, тј архитектуре софтверског ситета. [1]

3.1. Архитектура софтверског система

Архитектура система је тронивојска и састоји се од следећих нивоа:

- ♦ Кориснички интерфејс
- ♦ Апликациона логика
- ♦ Складиште података



Слика 14. 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 15. Структура корисничког интерфејса

3.2.1. СК1: Случај коришћења - Пријава корисника

Назив СК

Пријава корисника

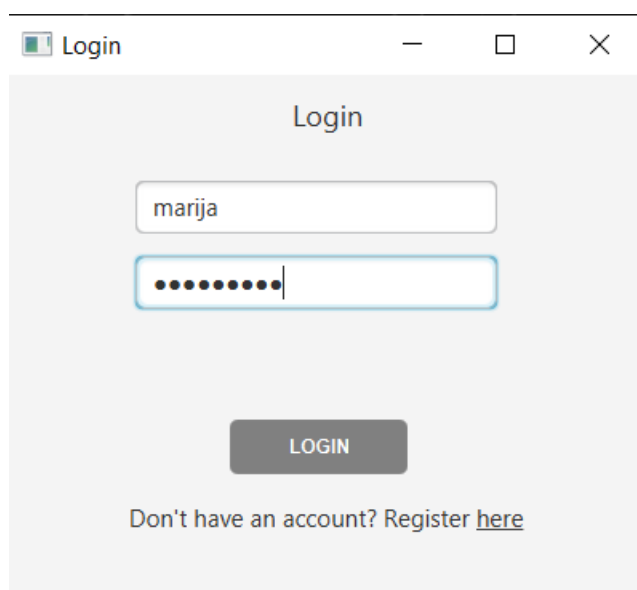
Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен и приказује форму за пријаву корисника.

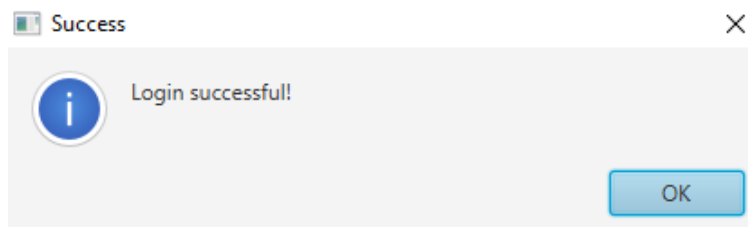


The image shows a web browser window with a title bar that says 'Login'. Inside the window, there is a form with the title 'Login' centered at the top. Below the title, there are two input fields. The first field contains the text 'marija'. The second field contains a series of dots, indicating a password. Below these fields is a button labeled 'LOGIN'. At the bottom of the form, there is a link that says 'Don't have an account? Register [here](#)'.

Слика 16. Пријава корисника - форма

Основни сценарио СК:

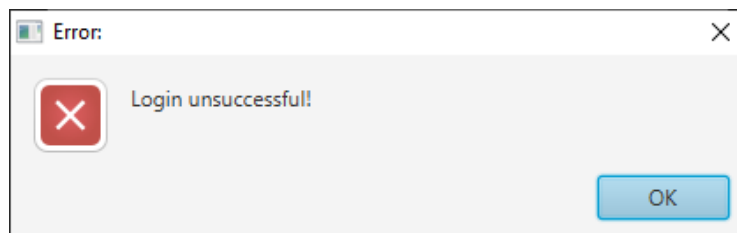
1. **Корисник** уноси податке за пријаву. (АПУСО)
2. **Корисник** контролише да ли је коректно унео податке. (АНСО)
3. **Корисник** позива систем да изврши пријаву корисника. (АПСО)
4. **Систем** врши пријављивање корисника. (СО)
5. **Систем** приказује кориснику поруку: „Login successful!” и омогућава приступ систему. (ИА)



Слика 17. Пријава корисника – основни сценарио

Алтернативна сценарија:

- 5.1. Уколико систем не пронађе корисника са унетим подацима, приказује кориснику поруку: „Login unsuccessful!”. (ИА)



Слика 18. Пријава корисника – алтернативни сценарио

3.2.2. СК2: Случај коришћења - Регистрација корисника

Назив СК

Регистрација корисника

Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен и приказује форму за регистрацију корисника.

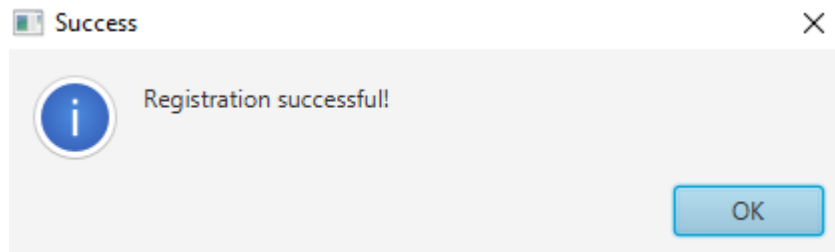
here'." data-bbox="298 363 689 645"/>

The image shows a web browser window with the title 'Login'. Inside the window, there is a 'Registration' form. The form consists of three input fields stacked vertically. The first field contains the text 'marija'. The second field contains a series of dots, indicating a masked password. The third field is labeled 'Email address'. Below these fields is a dark button with the text 'REGISTER' in white. At the bottom of the form, there is a link that says 'You have an account? Login [here](#)'.

Слика 19. Регистрација корисника – форма

Основни сценарио СК:

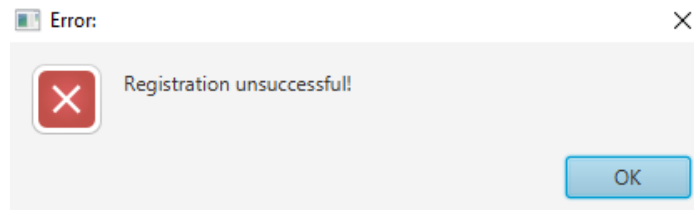
1. **Корисник** уноси податке за регистрацију. (АПУСО)
2. **Корисник** контролише да ли је коректно унео податке. (АНСО)
3. **Корисник** позива систем да изврши регистрацију корисника. (АПСО)
4. **Систем** врши регистрацију корисника. (СО)
5. **Систем** приказује кориснику поруку: „Registration successful!” и омогућава приступ систему. (ИА)



Слика 20. Регистрација корисника – основни сценарио

Алтернативна сценарија:

- 5.1. Уколико систем не може да региструје корисника са унетим подацима, приказује кориснику поруку: „Registration unsuccessful!”. (ИА)



Слика 21. Регистрација корисника – алтернативни сценарио

3.2.3. СК3: Случај коришћења – Учитавање нивоа

Назив СК

Учитавање нивоа

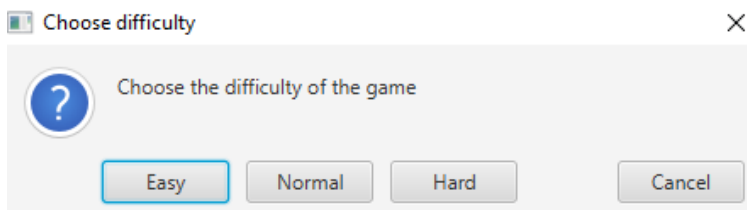
Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

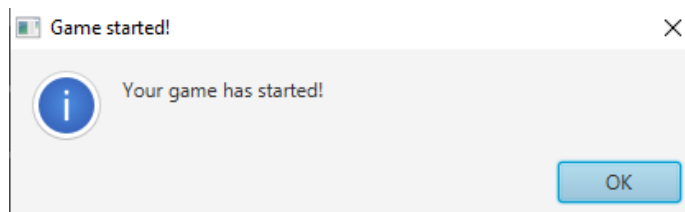
Предуслов: Систем је укључен. Корисник је пријављен на систем и приказује форму за бирање тежине.



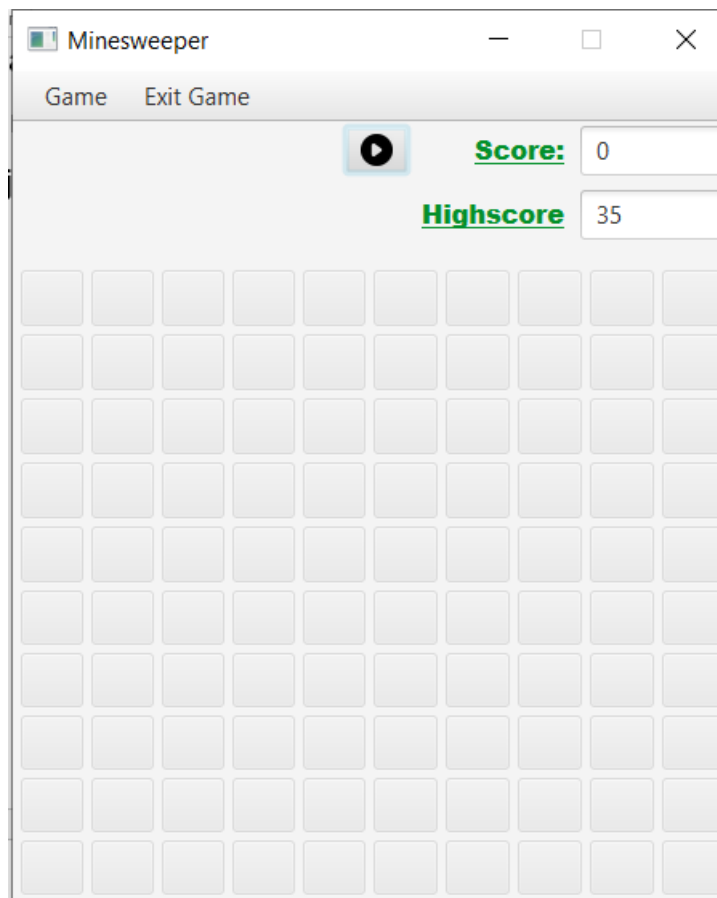
Слика 22. Учитавање нивоа

Основни сценарио СК:

1. **Корисник** бира тежину нивоа која му одговара. (АПУСО)
2. **Корисник** контролише да ли бира коректну тежину. (АНСО)
3. **Корисник** позива систем да учита ниво. (АПСО)
4. **Систем** врши учитавање игре. (СО)
5. **Систем** приказује играчу поруку: „Your game has started!”. (ИА)



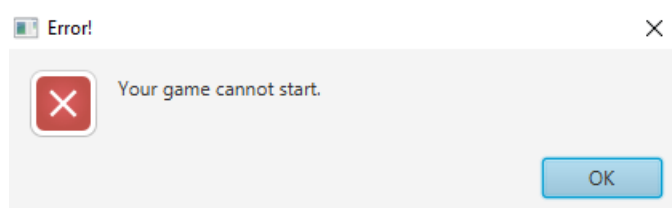
Слика 23. Учитавање нивоа – основни сценарио



Слика 24. Учитавање нивоа – основни сценарио

Алтернативна сценарија:

5.1. Уколико систем не може да започне игру, приказује поруку: „Your game cannot start!”. (ИА)



Слика 25. Учитавање нивоа – алтернативни сценарио

3.2.4. СК4: Случај коришћења – Чување резултата игре

Назив СК

Чување резултата игре

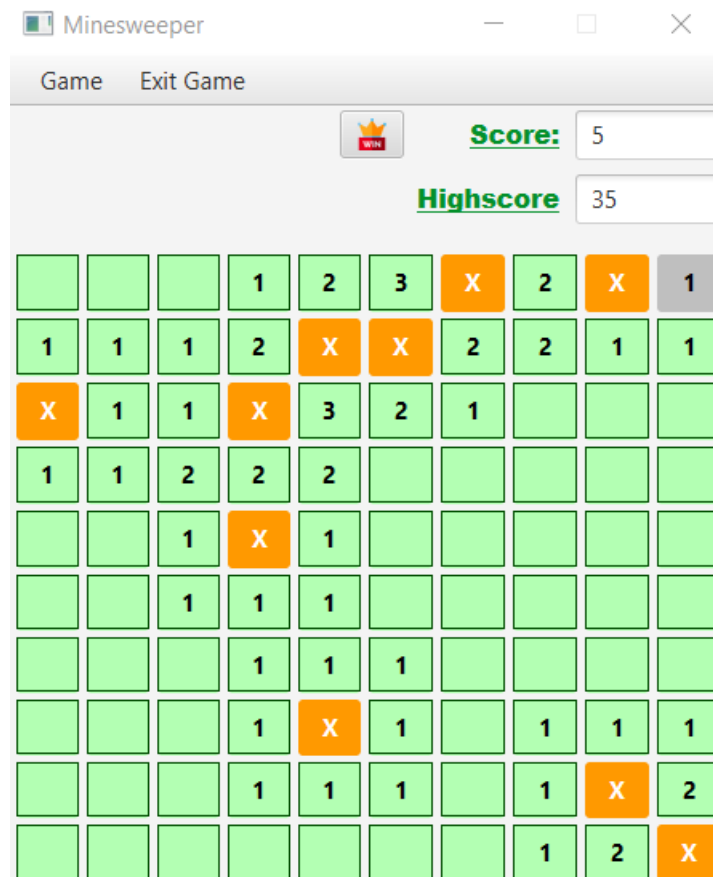
Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

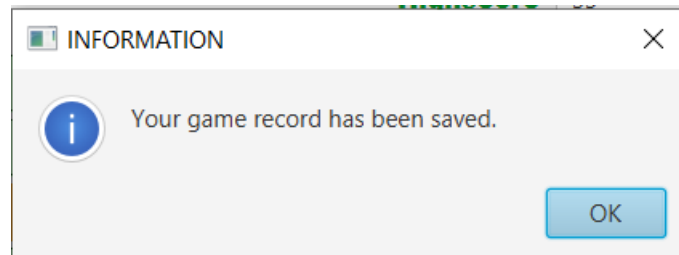
Предуслов: Систем је укључен. Корисник је пријављен на систем и игра је успешно креирана и започета.



Слика 26. Чување резултата игре – крај игре

Основни сценарио СК:

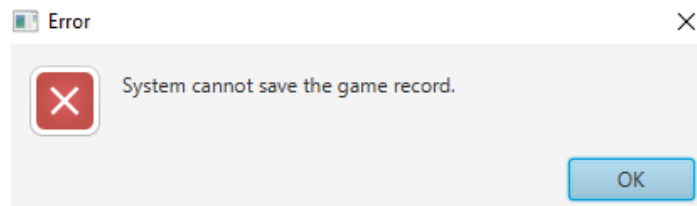
1. **Корисник** позива систем да сачува резултат. (АПСО)
2. **Систем** чува резултат игре. (СО)
3. **Систем** приказује кориснику резултат и поруку: „Your game record has been saved!“. (ИА)



Слика 27. Чување резултата игре – основни сценарио

Алтернативна сценарија:

- 3.1. Уколико систем не може да сачува резултат, приказује поруку: „System cannot save the game record!“. (ИА)



Слика 28. Чување резултата игре – алтернативни сценарио

3.2.5. СК5: Случај коришћења – Чување корисниковог најбољег резултата

Назив СК

Чување корисниковог најбољег резултата

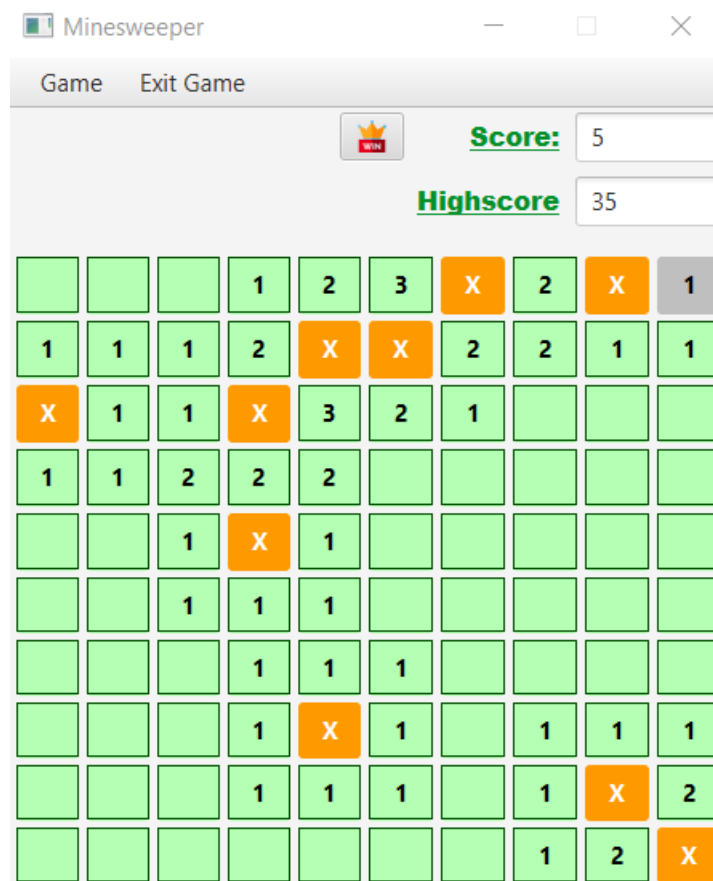
Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

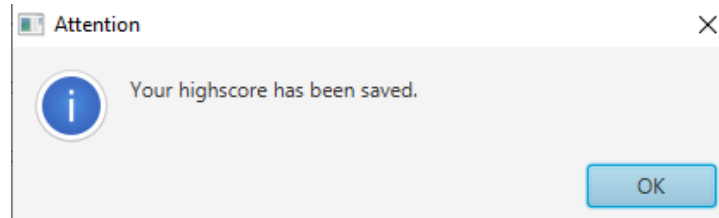
Предуслов: Систем је укључен. Корисник је пријављен на систем и игра је успешно завршена.



Слика 29. Чување корисниковог најбољег резултата

Основни сценарио СК:

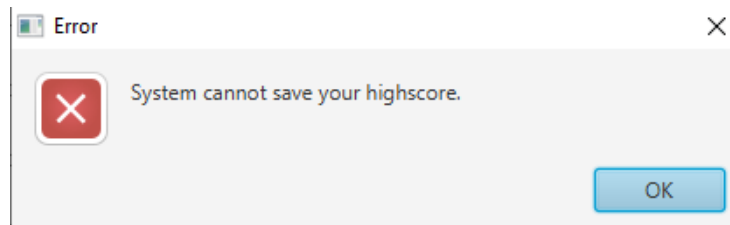
1. **Корисник** позива систем да сачува нови најбољи резултат корисника. (АПСО)
2. **Систем** чува резултат игре. (СО)
3. **Систем** приказује кориснику резултат и поруку: „Your highscore has been saved!“. (ИА)



Слика 30. Чување најбољег корисниковог резултата – основни сценарио

Алтернативна сценарија:

- 3.1. Уколико систем не може да сачува резултат, приказује поруку: „System cannot save your highscore “. (ИА)



Слика 31. Чување најбољег корисниковог резултата – алтернативни сценарио

3.3. Комуникација сервер - клијент

На серверској страни се покреће 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 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.

```
public class Request implements Serializable {

    private Game game;
    private User user;
    private Level level;
    private int operation;

    public Request() {
    }

    public Game getGame() {
        return game;
    }

    public void setGame(Game game) {
        this.game = game;
    }

    public User getUser() {
        return user;
    }

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

    public Level getLevel() {
        return level;
    }

    public void setLevel(Level level) {
        this.level = level;
    }
}
```



```

    }

    public int getOperation() {
        return operation;
    }

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

```

```

public class Response implements Serializable {

    private Game game;
    private User user;
    private Level level;
    private Object error;
    private ResponseStatus status;

    public Response() {
    }

    public ResponseStatus getStatus() {
        return status;
    }

    public void setStatus(ResponseStatus status) {
        this.status = status;
    }

    public Object getError() {
        return error;
    }

    public void setError(Object error) {
        this.error = error;
    }

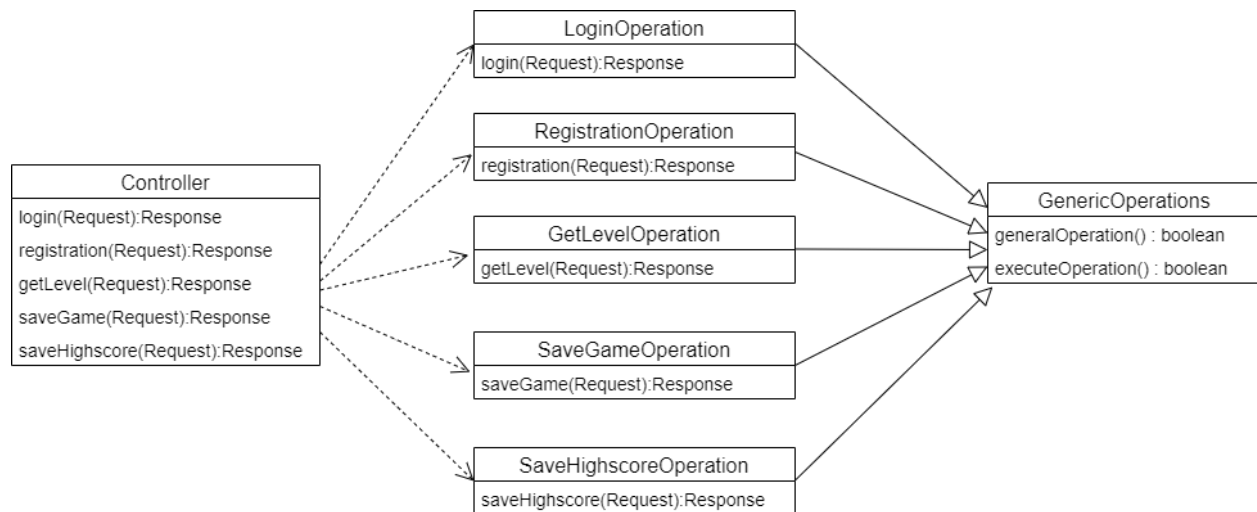
    public Game getGame() {
        return game;
    }
}

```

```
public void setGame(Game game) {  
    this.game = game;  
}  
  
public User getUser() {  
    return user;  
}  
  
public void setUser(User user) {  
    this.user = user;  
}  
  
public Level getLevel() {  
    return level;  
}  
  
public void setLevel(Level level) {  
    this.level = level;  
}  
}
```

3.4. Контролер апликационе логике

Контролер апликационе логике 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 32. Приказ зависности контролера и класа одговорних за извршење системских операција

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

```
public abstract class GenericOperations {

static public DbBroker dbb = new DbBroker1();
    GeneralDomainObject gdo;
    synchronized public boolean generalOperation(){
        dbb.makeConnection();
        boolean signal = executeOperation();
        if (signal == true){
            dbb.commitTransation();
        }else{
            dbb.rollbackTransation();
        }
    }
}
```

```

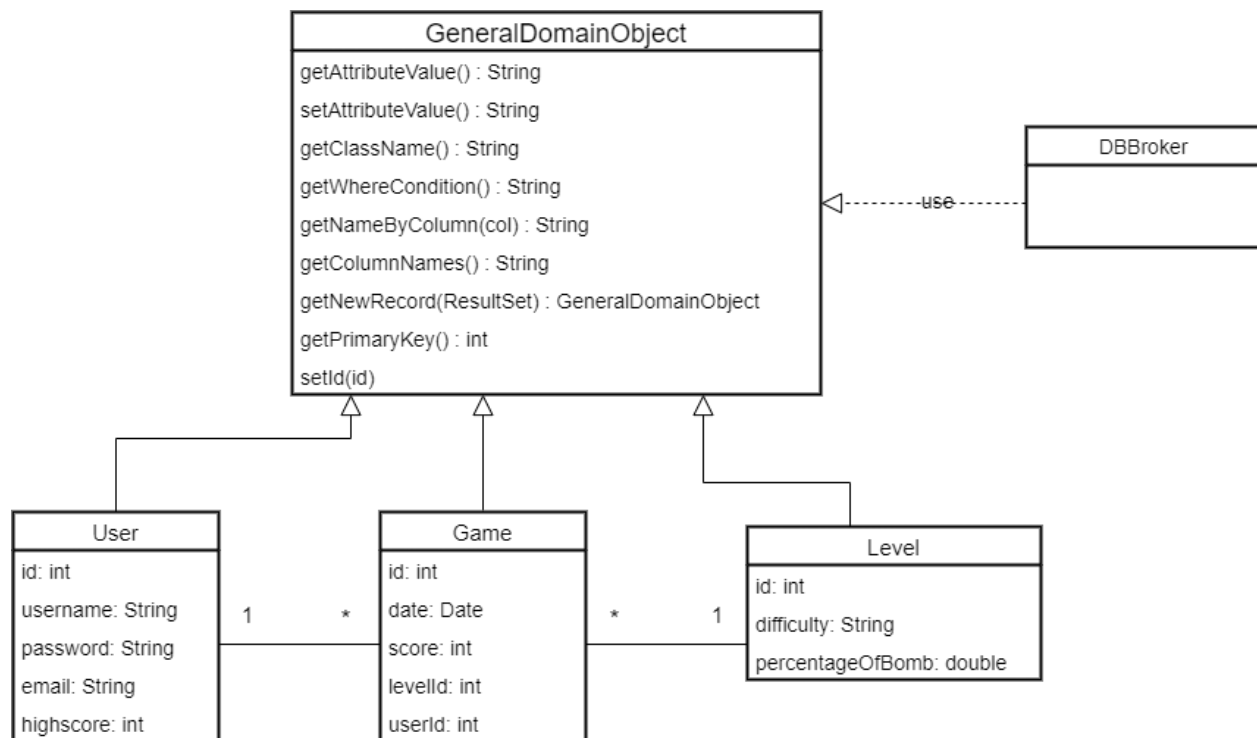
    }
    dbb.closeConnection();
    return signal;
}

abstract public boolean executeOperation();
}

```

3.5. Пројектовање структуре софтверског система - Доменске класе

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



Слика 33. Доменске класе и зависност брокера од GeneralDomainObject

```

public class Game extends GeneralDomainObject implements Serializable {

    private int id;
    private Date date;
    private int score;
    private int levelId;
    private int userId;

    public Game(){

        id = 0;
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        date = new Date();
        date = java.sql.Date.valueOf(sdf.format(date));
        score = 0;
        levelId = 0;
        userId = 0;
    }

    public Game(int id, Date date, int score, int levelId, int userId){

        this.id = id;
        this.date = date;
        this.score = score;
        this.levelId = levelId;
        this.userId = userId;
    }

    public Game( int id){

        this.id = id;
    }

    public void setId(int id){

        this.id = id;
    }

    public int getPrimaryKey(){

        return this.id;
    }

    public Date getDate(Date date){

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

```

```

        this.date = java.sql.Date.valueOf(sdf.format(date));
        return this.date;
    }

    public void setDate(Date date){

        this.date = date;
    }

    public void setScore(int score){

        this.score = score;
    }

    public int getLevel(){

        return levelId;
    }

    public void setLevel(int level){

        this.levelId = level;
    }

    public int getUserId(){

        return userId;
    }

    public void setUserId(int userId){

        this.userId = userId;
    }

    @Override
    public String getAttributeValue() {
        return score + ", " + "" + date + "" + ", " + levelId + ", " + userId;
    }

    @Override
    public String setAttributeValue() {
        return "GameID="+ id + ", " + "Score=" + score + ", " + "Date=" + "" + getDate(date)
+ "" + ", " + "LevelId=" + levelId + ", " + "UserId=" + userId;
    }

```

```

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

@Override
public String getWhereCondition() {
    return "GameID = " + id;
}

@Override
public String getNameByColumn(int col) {
    String names[] = {"id", "score", "date", "levelId", "userId"};
    return names[col];
}

@Override
public String getColumnNames() {
    return " (score, date, levelId, userId) ";
}

@Override
public GeneralDomainObject getNewRecord(ResultSet rs) throws SQLException {
    return new Game(rs.getInt("id"),rs.getDate("date"),    rs.getInt("score"),
rs.getInt("levelId"), rs.getInt("userId"));
}
}

public class Level extends GeneralDomainObject implements Serializable {
    public int id;
    public String difficulty;
    public double percentageOfBomb;

    public Level(){

        id = 0;
        difficulty = "";
        percentageOfBomb = 0;
    }

    public Level(int id, String difficulty, double percentageOfBomb){

        this.id = id;
        this.difficulty = difficulty;
        this.percentageOfBomb = percentageOfBomb;
    }
}

```

```

}

public Level(int id){

    this.id = id;
}

public void setId(int id){

    this.id = id;
}

public int getPrimaryKey(){

    return this.id;
}

@Override
public String getAttributeValue() {
    return id + ", " + (difficulty == null ? null : "" + difficulty + "") + ", " +
percentageOfBomb;
}

@Override
public String setAttributeValue() {
    return "LevelID=" + id + ", " + "difficulty=" + (difficulty == null ? null : "" + difficulty +
"") + ", " + "percentageOfBomb=" + percentageOfBomb;
}

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

@Override
public String getWhereCondition() {
    return "difficulty = " + difficulty + "";
}

@Override
public String getNameByColumn(int col) {
    String names[] = {"id", "difficulty", "percentageOfBomb"};
    return names[col];
}

```



```

@Override
public String getColumnNames() {
    return " (difficulty,percentageOfBomb) ";
}

@Override
public GeneralDomainObject getNewRecord(ResultSet rs) throws SQLException {
    return new Level(rs.getInt("id"),
rs.getString("difficulty"),rs.getDouble("percentageOfBomb"));
}
}

public class User extends GeneralDomainObject implements Serializable {

    public int id;
    public String username;
    public String password;
    public String email;
    public int highscore;

    public User(){

        id = 0;
        username = "";
        password = "";
        email = "";
        highscore = 0;
    }

    public User(int id, String username, String password, String email, int highscore){

        this.id = id;
        this.username = username;
        this.password = password;
        this.email = email;
        this.highscore = highscore;
    }

    public String getUsername() {
        return username;
    }

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

```

```

public String getPassword() {
    return password;
}

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

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public int getHighscore() {
    return highscore;
}

public void setHighscore(int highscore) {
    this.highscore = highscore;
}

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

@Override
public String setAttributeValue() {
    return "Username=" + (username == null ? null : "" + username + "") + ", " +
"Password=" + (password == null ? null : "" + password + "") + ", " + "Email=" + (email ==
null ? null : "" + email + "") + ", " + "Highscore=" + highscore;
}

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

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

```

```

    }

    @Override
    public String getNameByColumn(int col) {
        String names[] = {"id", "username", "password", "email", "highscore"};
        return names[col];
    }

    @Override
    public String getColumnNames() {
        return " (username, password, email, highscore) ";
    }

    @Override
    public GeneralDomainObject getNewRecord(ResultSet rs) throws SQLException {
        return new User(rs.getInt("id"), rs.getString("username"),rs.getString("password"),
rs.getString("email"), rs.getInt("highscore"));
    }

    @Override
    public int getPrimaryKey() {
        return id;
    }

    @Override
    public void setId(int id) {
        id = id;
    }
}

```

3.6. Пројектовање пословне логике

У фази анализе одредили смо уговоре о системским операцијама, при чему смо рекли да један уговор описује понашање једне системске операције, тако што описује **ШТА** операција треба да ради, **али НЕ и КАКО**.

У фази пројектовања за сваки од уговора се пројектује концептуално решење (реализација) СО. То значи да ћемо за сваку класу одговорну за извршење СО дефинисати **КАКО** ће се СО извршити.

Аспекти реализације који се односе на конекцију са базом, перзистентност и трансакције треба избећи у почетку пројектовања СО.

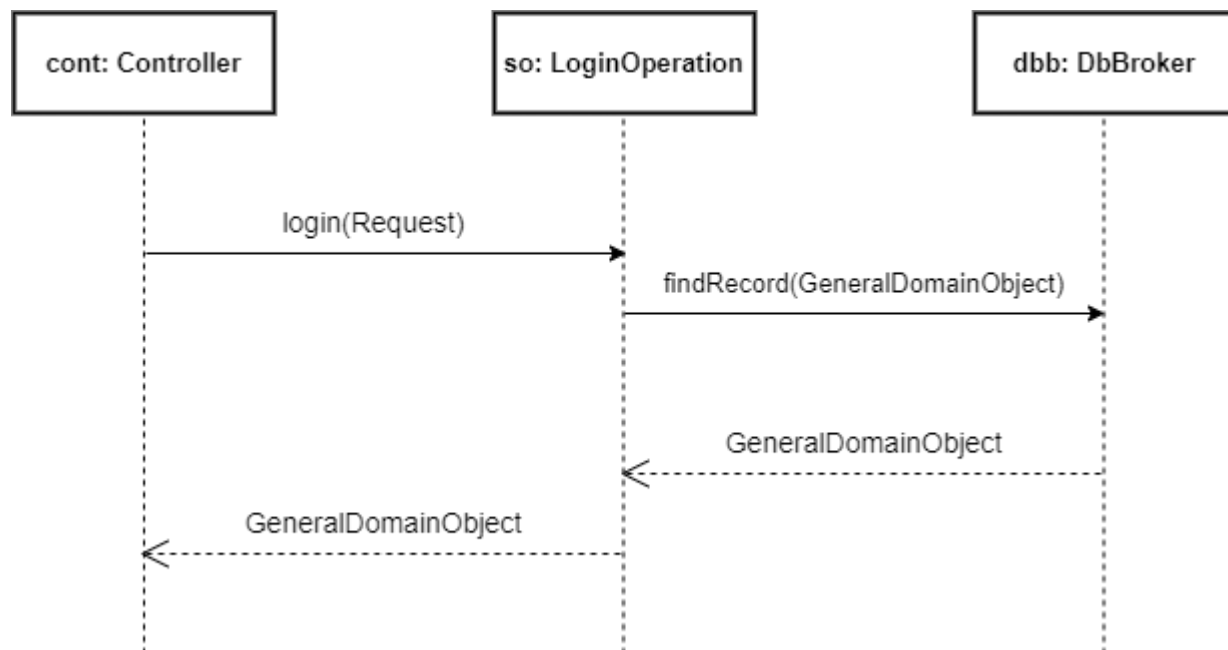
1. Уговор УГ1: *Login*

Операција: *Login(Korisnik)*: сигнал

Веза са СК: СК1

Предуслови: /

Постуслови: Корисник је пријављен.



Слика 34. Уговор УГ1

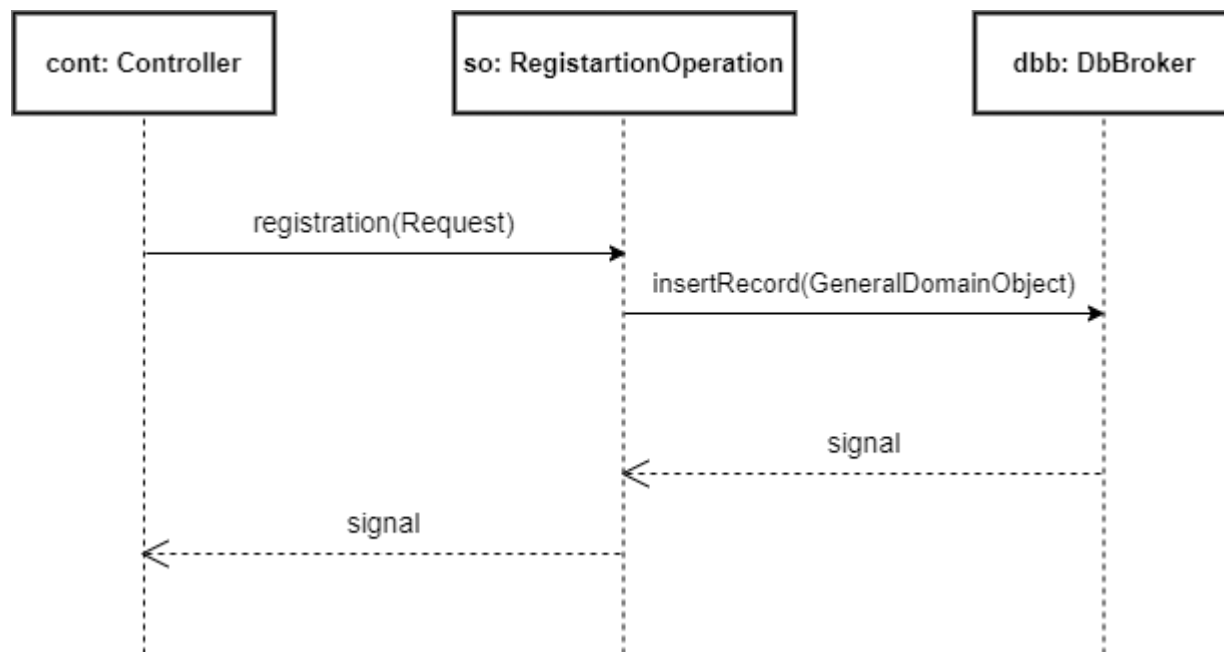
2. Уговор УГ2: *Registartion*

Операција: *Registration(Korisnik)*: сигнал

Веза са СК: СК2

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

Постуслови: Корисник је регистрован.



Слика 35.. Уговор УГ2

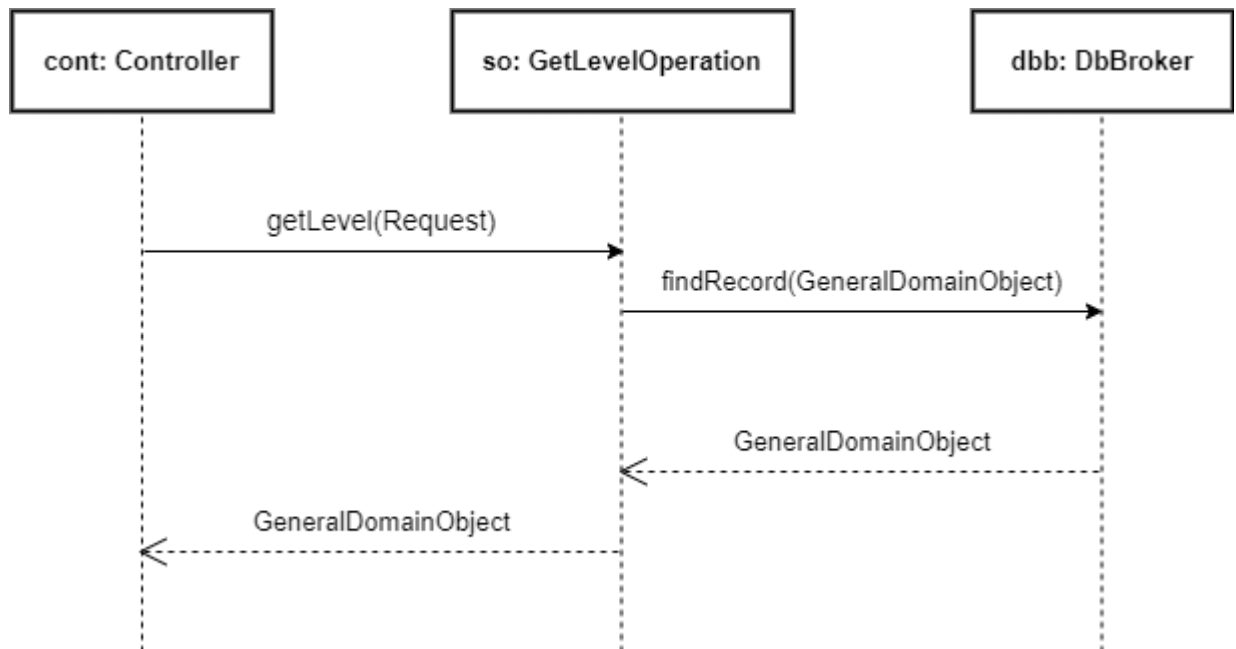
3. Уговор УГ3: *LoadLevel*

Операција: *loadLevel(Level)*: сигнал

Веза са СК: СК3

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

Постуслови: Игра је почела.



Слика 36. Уговор УГ3

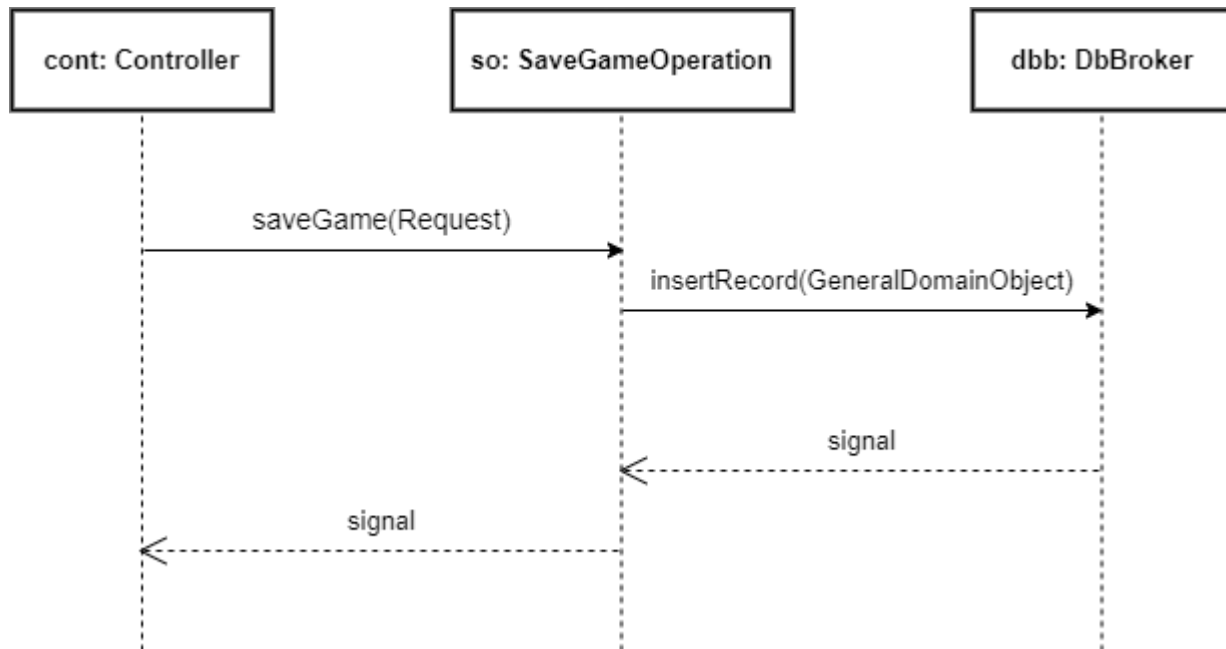
4. Уговор УГ4: *SaveRecord*

Операција: *SaveRecord(Record)*: сигнал

Веза са СК: СК4

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

Постуслови: Резултат је сачуван.



Слика 37. Уговор УГ4

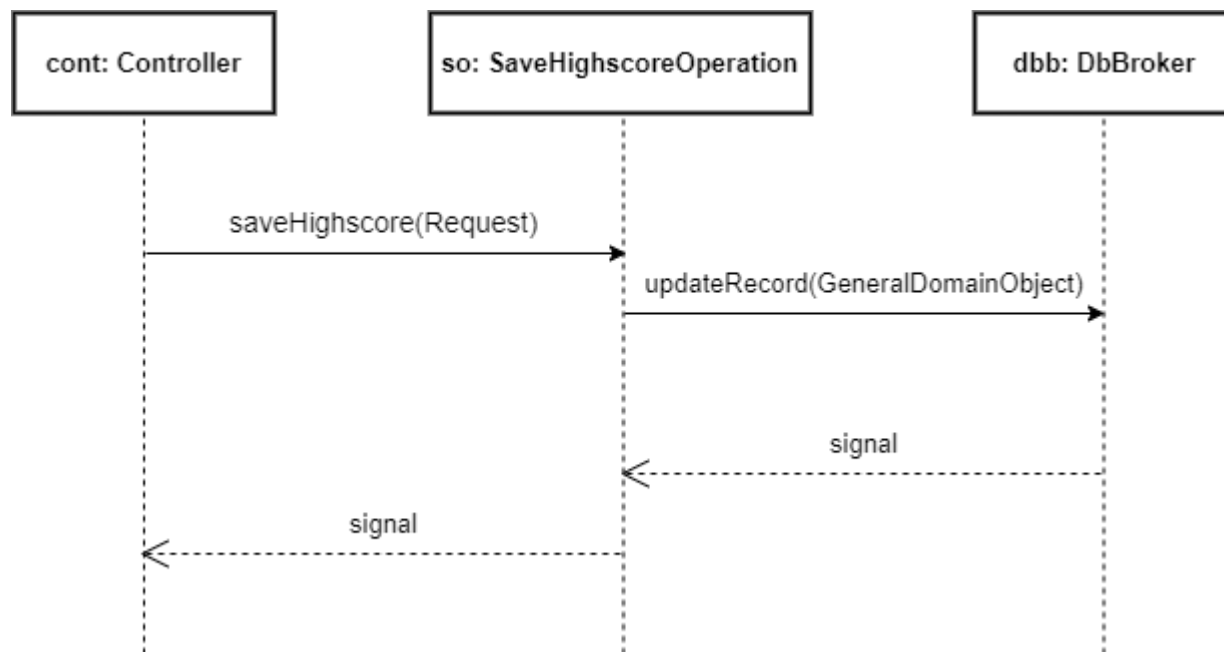
5. Уговор УГ5: *SaveHighscore*

Операција: *SaveHighscore* (*Record*): сигнал

Веза са СК: СК5

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

Постуслови: Корисников најбољи резултат је сачуван.



Слика 38. Уговор УГ5

3.7. Пројектовање брокера базе података

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

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

```
public abstract boolean makeConnection();
public abstract boolean insertRecord(GeneralDomainObject gdo);
public abstract boolean updateRecord(GeneralDomainObject gdo);
public abstract boolean updateRecord(GeneralDomainObject gdo, GeneralDomainObject gdoid);
public abstract boolean deleteRecord(GeneralDomainObject gdo);
public abstract boolean deleteRecords(GeneralDomainObject gdo, String where);
public abstract GeneralDomainObject findRecord(GeneralDomainObject gdo);
public abstract List<GeneralDomainObject> findRecord(GeneralDomainObject gdo, String where);
public abstract boolean commitTransation();
public abstract boolean rollbackTransation();
public abstract boolean increaseCounter(GeneralDomainObject gdo, AtomicInteger counter);
public abstract void closeConnection();
public abstract GeneralDomainObject getRecord(GeneralDomainObject gdo,int index);
public abstract int getRecordsNumber(GeneralDomainObject gdo);
```

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

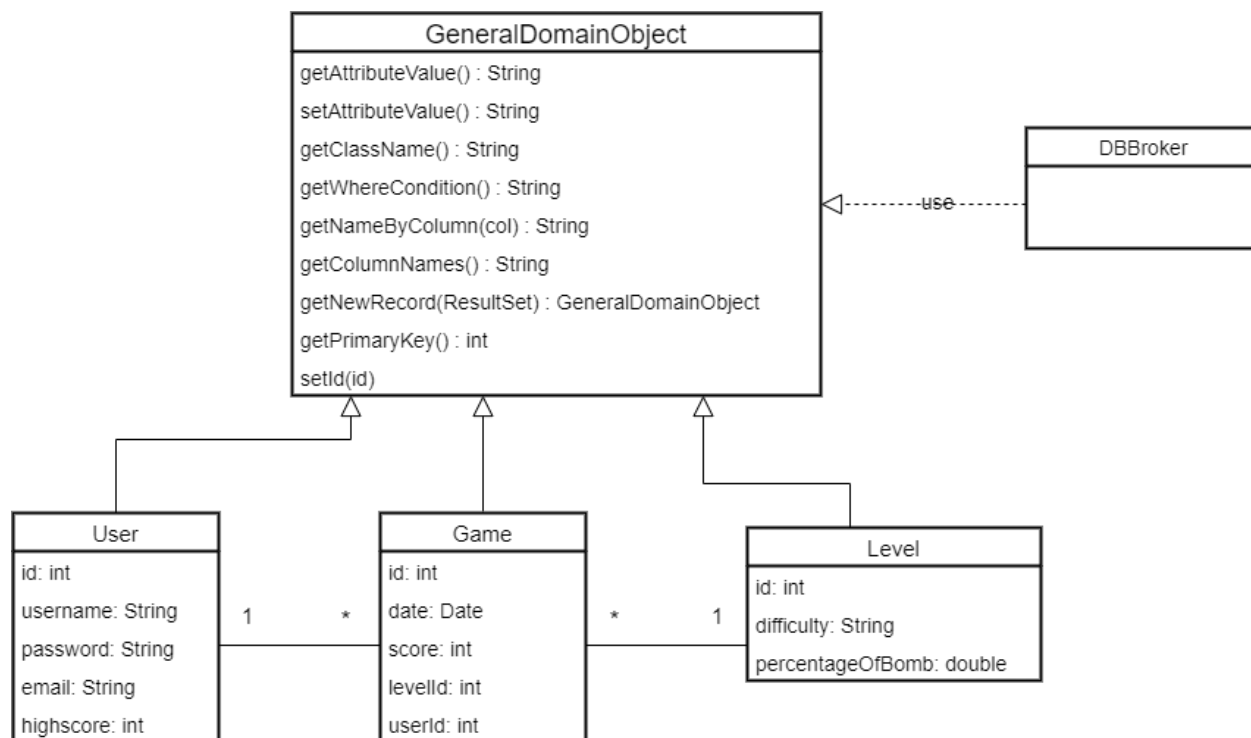
```
abstract public String getAttributeValue();
abstract public String setAttributeValue();
abstract public String getClassName();
abstract public String getWhereCondition();
abstract public String getNameByColumn(int col);
```

abstract public String getColumnNames();

abstract public GeneralDomainObject getNewRecord(ResultSet rs) throws SQLException;

abstract public int getPrimaryKey();

abstract public void setId(int id);



Слика 39. Веза брокера базе података са доменским класама

3.8. Пројектовање складишта података

На основу софтверских класа структуре пројектоване су табеле (складишта података) релационог система за управљање базом података. У овом раду коришћен је СУБП „MySQL”:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default	Comment	Collation
1	id	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREMENT		
2	username	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	No default		utf8_unicode_ci
3	password	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	No default		utf8_unicode_ci
4	email	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	No default		utf8_unicode_ci
5	highscore	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	No default		

Слика 40. Табела *User*

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default	Comment	Collation
1	id	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	No default		
2	difficulty	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	No default		utf8_unicode_ci
3	percentageOfBomb	DOUBLE		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	No default		

Слика 41. Табела *Level*

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default	Comment
1	id	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREMENT	
2	date	DATE		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL	
3	score	INT	11	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL	
4	userId	INT	11	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL	
5	levelId	INT	11	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL	

Слика 42. Табела *Game*

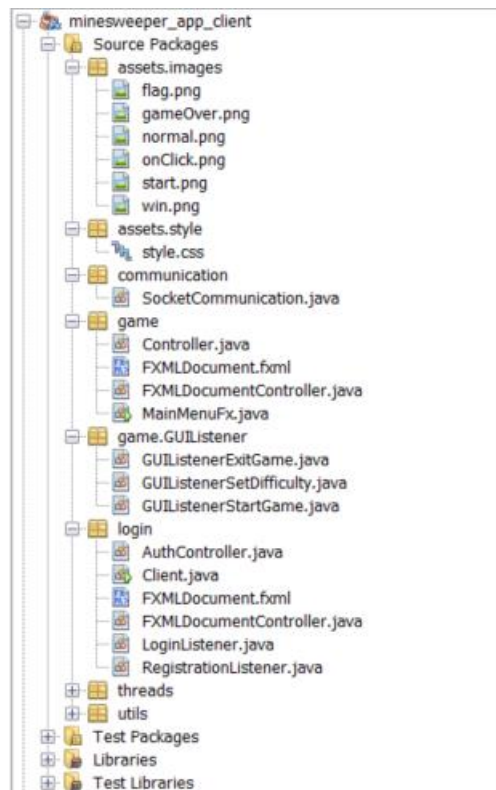
4. Имплементација

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

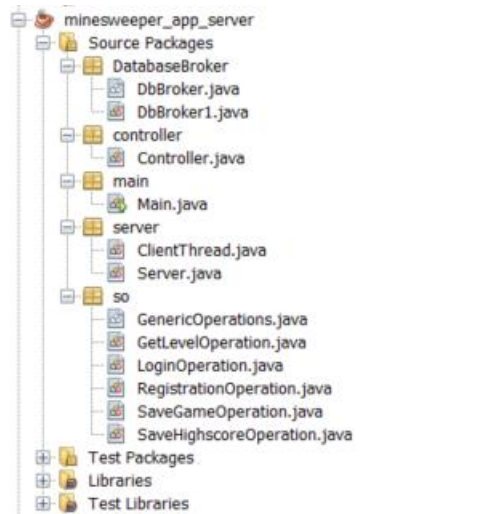
Као развојно окружење коришћен је *NetBeans 8.2*. Као систем за управљање базом података коришћен је *MySQL* систем за управљање базом података.

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

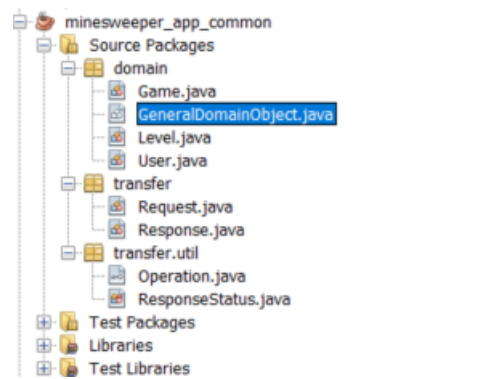
Клијентски пројекат користи *JavaFX* платформу која омогућава олакшан рад и креирање графичких компонената.



Слика 43. Имплементација клијентског дела апликације

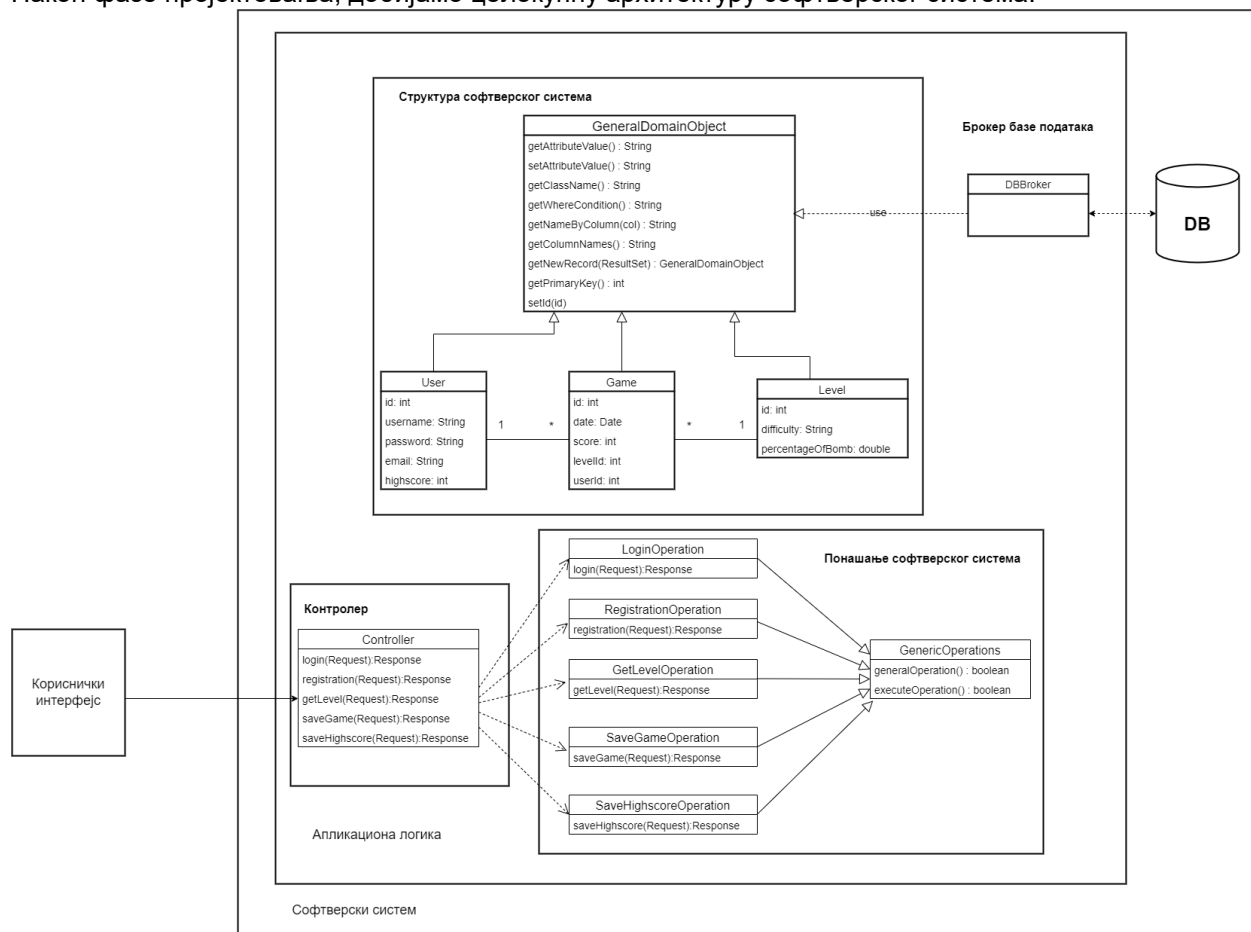


Слика 44. Имплементација серверског дела апликације



Слика 45. Имплементација заједничког дела апликације

Након фазе пројектовања, добијамо целокупну архитектуру софтверског система:



Слика 46. Архитектура софтверског система

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 у класи FXMLDocumentController.

@FXML

```
public void initialize() throws NoSuchFieldException, IllegalArgumentException,
IllegalAccessException {
```

```
    con = new Controller(this);
```

```
    for( int x = 0; x < fieldRows; x++){
```

```
        for(int y = 0; y < fieldColumns; y++){
```

```
            String nameBtn = "p" + x + y;
```

```
            Class cls = this.getClass();
```

```
            Field field = cls.getDeclaredField(nameBtn);
```

```
            Button btn = (Button) field.get(this);
```

```
            btn.setDisable(true);
```

```
            btn.setOnMouseClicked(e ->{
```

```
                if(e.getButton() == MouseButton.PRIMARY){
```

```
                    con.open((Button) e.getSource());
```

```
                }else if(e.getButton() == MouseButton.SECONDARY){
```

```
                    con.setFlag((Button) e.getSource());
```

```
                }
```

```
            });
```

```
        }
```

```
    }
```

```
}
```

```
public Button getButton(int x, int y) throws NoSuchFieldException,
IllegalArgumentException, IllegalAccessException{
```

```
    Button btn = null;
```

```
    String nameBtn = "p" + x + y;
```

```
    Class cls = this.getClass();
```

```
    Field field = cls.getDeclaredField(nameBtn);
```

```
    btn = (Button) field.get(this);
```

```
    return btn;
```

```
}
```

На овај начин се добија објекат доменске класе преко рефлексije поља класе *FXMLDocumentController*. Овакав приступ је од великог значаја када је потребно позвати објекат кад имамо само координате поља која су „погођена“.

5. Тестирање

Сваки од имплементираних случајева коришћења је тестиран. Приликом тестирања сваког случаја коришћења, поред унетих правилних података, уношени су и неправилни подаци да би се утврдило какав ће бити резултат извршења.

На основу извршених тестирања 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. Прикупљање захтева од корисника
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-архитектуре. Такође, коришћен је и механизам рефлексije.

Овако изграђен систем представља модуларан систем који је надградив, једноставнији за одржавање и који треба да омогући ефикасно вођење евиденције о играчима и самој игри.

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.

Можемо приметити да се и коришћена метода развоја софтвера, упрошћена Ларманова метода, заснива на принципу апстракције. Кроз све фазе развоја, крећемо се од генералног описа система, интеракција корисника и система, да би у каснијим фазама постепено уводили специфичности које се тичу самих акција, изгледа екранских форми, системских операција.

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

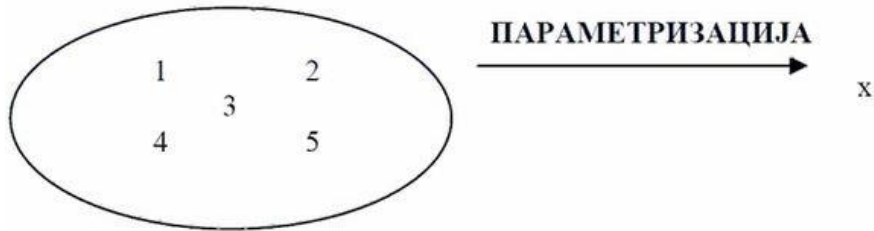
Параметризација је апстракција која издваја из неког скупа елемената њихова општа својства која су представљена преко параметара.

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

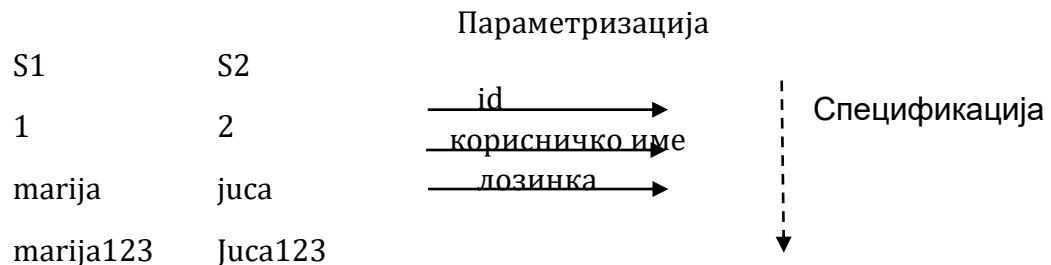
Уколико посматрамо скуп целих бројева 1,2,3... њих можемо представити преко неког општег представника скупа целих бројева. На пример помоћу параметра x . У приказаном случају, програмски би то записали као: **int x;**



Слика 47. Параметризација скупа елемената простог типа

2. Параметризација скупа елемената сложеног типа

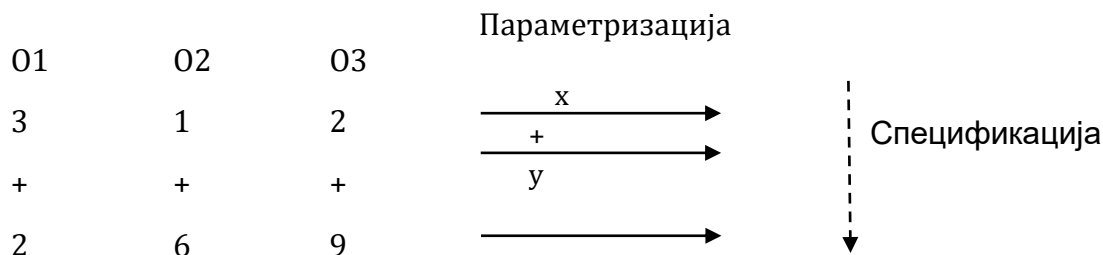
Уколико посматрамо неки скуп сложених објеката, нпр. скуп играча: (1, "marija", "marija123"), (2, "jusa", "jusa123"), тада параметризацијом добијамо општа својства тог скупа, односно њихове атрибуте: *ID*, корисничко име и лозинку. Наведени скуп објеката означимо са S , а елементе скупа са $s1, s2$.



- 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 (id, 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

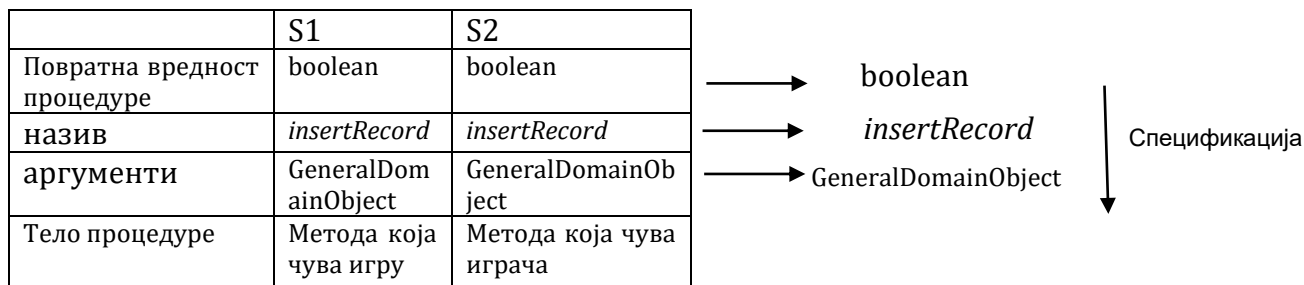
Уколико посматрамо неки скуп S процедура:

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

$s2: \text{boolean insertRecord}(\text{GeneralDomainObject } gdo)$ – метода која чува корисника.

Уочавамо да параметризацијом добијамо општу процедуру:

$\text{boolean insertRecord}(\text{GeneralDomainObject } gdo)$. Елементи који чине процедуру су: повратна вредност процедуре, назив процедуре, аргументи (параметри) процедуре и тело процедуре.



- 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: *Boolean insertRecord(GeneralDomainObject gdo).*

5. Параметризација скупа наредби

Уколико посматрамо скуп S наредби:

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

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


- System.out.println(
- "Igrac "
- +igraci[i].getKorisnickolme()+
- " se prikljucio igri!"
-);

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

Параметризација

-----> System.out.println(Спецификација
-----> "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 **апстракције 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 $\{(1, \text{"marija"}, \text{"marija123"}), (2, \text{"jusa"}, \text{"jusa123"})\}$, 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,m13), 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{
    Y y;
    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 *GenericOperations*, која у својој *generalOperation* методи, иако добро пројектованој, приказује зависност од класе *DbBroker*.

```

public abstract class GenericOperations {

    static public DbBroker dbb = new DbBroker1();
    GeneralDomainObject gdo;
    synchronized public boolean generalOperation() {
        dbb.makeConnection();
        boolean signal = executeOperation();
        if (signal == true){
            dbb.commitTransation();
        }else{
            dbb.rollbackTransation();
        }
        dbb.closeConnection();
        return signal;
    }

    abstract public boolean executeOperation();

}

```

Слика 48. Повезаност класа *GenericOperations* и *DbBroker*

Међутим, овакав ниво повезаности је низак, прихватљив и неизбежан, јер класа *GenericOperations* 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, регистравање, читавање нивоа, чување резултата игре, чување корисничког најбољег резултата.

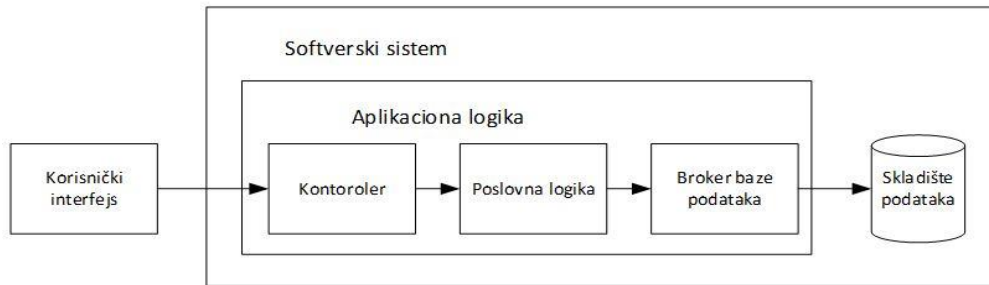


Слика 49. Декомпозиција код прикупљања захтева

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:

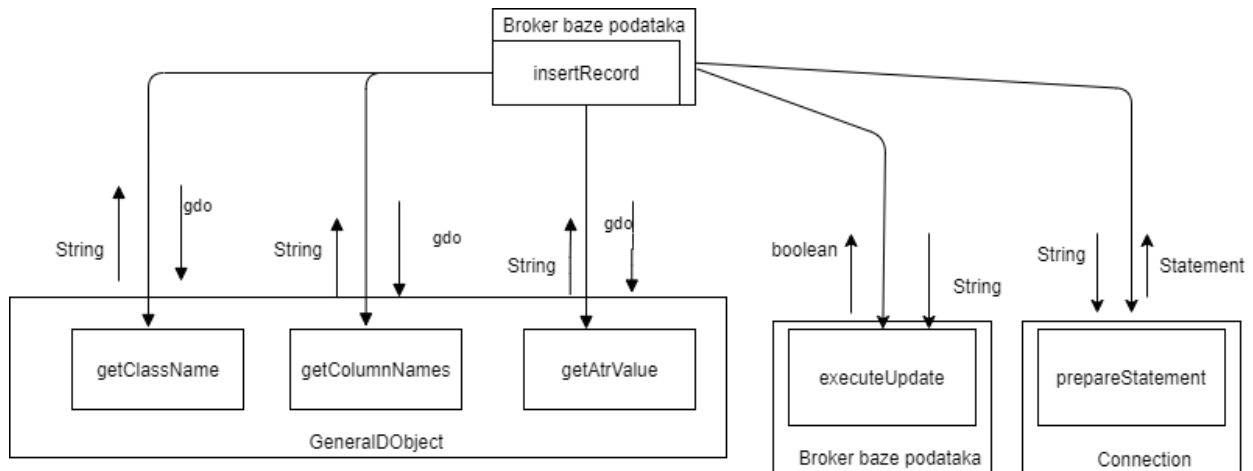
- Модули треба да имају јаку кохезију (*high cohesion*)
- Модули треба буду што је могуће слабије везани (*low coupling*)
- Сваки модул треба да чува своје интерне информације (*information hiding*)

Резултат процеса декомпозиције је софтверски систем који је модуларизован.

3. 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 51. 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 GenericOperations go;
        private List<Socket> clients;

        private Controller() {
        }

        public static Controller getInstance() {

            if (instance == null) {
                instance = new Controller();
            }

            return instance;
        }

        public Response login(Request request) throws Exception{
            go = new LoginOperation();
            return ((LoginOperation) go).login(request);
        }

        public Response registration(Request request) throws Exception{
            go = new RegistrationOperation();
            return ((RegistrationOperation) go).registration(request);
        }

        public Response getLevel(Request request){
            go = new GetLevelOperation();
            return ((GetLevelOperation) go).getLevel(request);
        }

        public Response saveGame(Request request) throws Exception{
            go = new SaveGameOperation();
            return ((SaveGameOperation) go).saveGame(request);
        }

        public Response saveHighscore(Request request) throws Exception{
            go = new SaveHighscoreOperation();
            return ((SaveHighscoreOperation) go).saveHighscore(request);
        }

    }
}

```

Слика 52. 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 52. , 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.

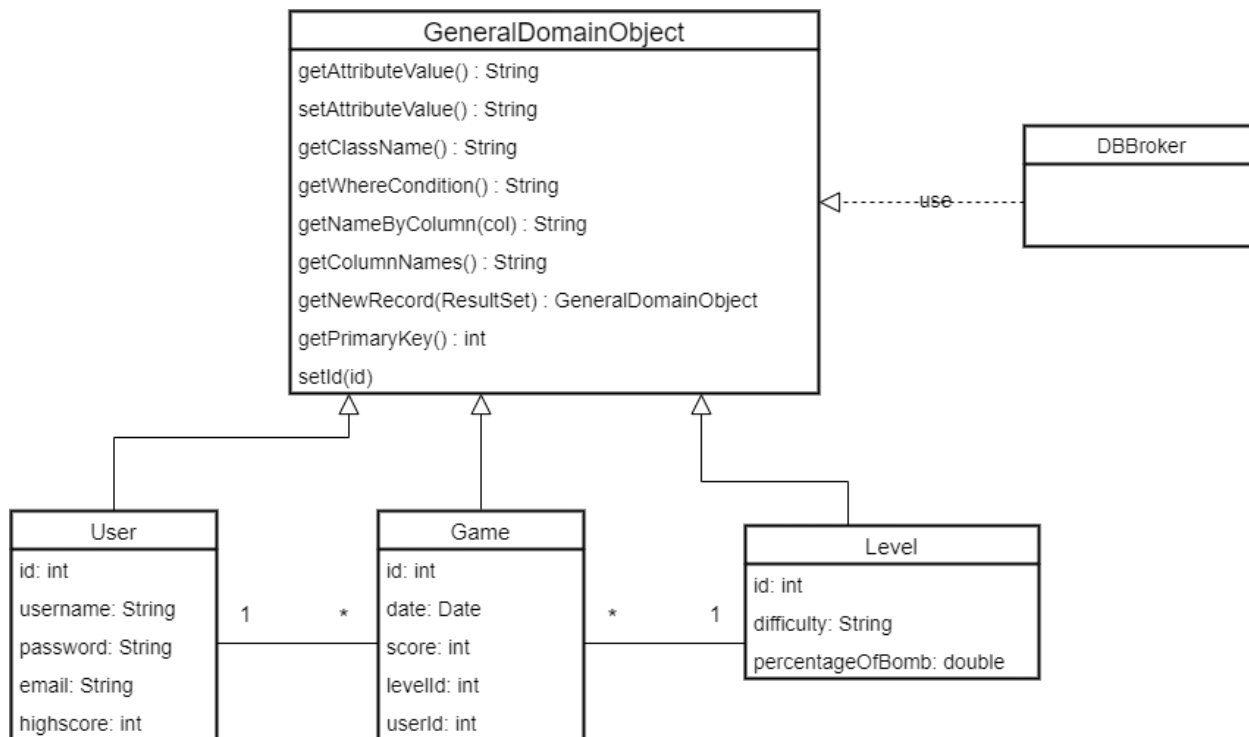
За енкапсулацију можемо рећи да је такође апстракција, јер раздваја нешто што је опште (јавно) од специфичног (приватно).

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 *DbBroker*, а сервер је класа *GeneralDomainObject*.



Слика 52. Одвајање интерфејса од имплементације

Брокер је тај који позива методе класа које имплементирају класу *GeneralDomainObject*, 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 *GenericOperations*:

```
public abstract class GenericOperations {  
  
    static public DbBroker dbb = new DbBroker1();  
    GeneralDomainObject gdo;  
    synchronized public boolean generalOperation(){  
        dbb.makeConnection();  
        boolean signal = executeOperation();  
        if (signal == true){  
            dbb.commitTransation();  
        }else{
```

```

        dbb.rollbackTransation();
    }
    dbb.closeConnection();
    return signal;
}

abstract public boolean executeOperation();
}

```

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, уочили 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.

7.4. Принципи објектно оријентисаног пројектовања (*Principles of object oriented 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 *OpstelzvršenjeSO*, која као што је већ речено, има *opstelzvršenjeSO* 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 GenericOperations {  
  
    static public DbBroker dbb = new DbBroker1();  
    GeneralDomainObject gdo;  
    synchronized public boolean generalOperation(){  
        dbb.makeConnection();  
        boolean signal = executeOperation();  
        if (signal == true){  
            dbb.commitTransation();  
        }else{  
            dbb.rollbackTransation();  
        }  
        dbb.closeConnection();  
        return signal;  
    }  
  
    abstract public boolean executeOperation();  
}
```

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

подкласе треба да буду заменљиве са њиховим надкласама.

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

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 GeneralDomainObject findRecord(GeneralDomainObject gdo) {  
  
    ResultSet rs = null;  
    Statement st = null;  
    String query = "SELECT * FROM " + gdo.getClassName() + " WHERE " +  
gdo.getWhereCondition();  
    boolean signal;  
    try{  
        st = conn.prepareStatement(query);  
        rs = st.executeQuery(query);  
        signal = rs.next();  
        if(signal == true){  
            gdo = gdo.getNewRecord(rs);  
        }else{  
            gdo = null;  
        }  
    }catch (SQLException ex){  
        Logger.getLogger(DbBroker1.class.getName()).log(Level.SEVERE, null, ex);  
    }finally{  
        close(null, st, rs);  
    }  
    return gdo;  
}
```


Системска операција GetLevelOperation:

```
public class GetLevelOperation extends GenericOperations{

    Request request;
    Response response;

    public Response getLevel(Request request){
        this.request = request;
        this.response = new Response();
        generalOperation();
        return response;
    }

    @Override
    public boolean executeOperation() {
        System.out.println(request.getLevel().getWhereCondition());
        Level level = (Level) dbb.findRecord(request.getLevel());
        response.setLevel(level);
        return true;
    }
}
```

Клијентски контролер:

```
Level getLevel(String difficulty) {

    Request request = new Request();
    level = new Level();
    level.difficulty = difficulty;

    request.setLevel(level);
    request.setOperation(Operation.OPERATION_GET_LEVEL_DIFFICULTY);

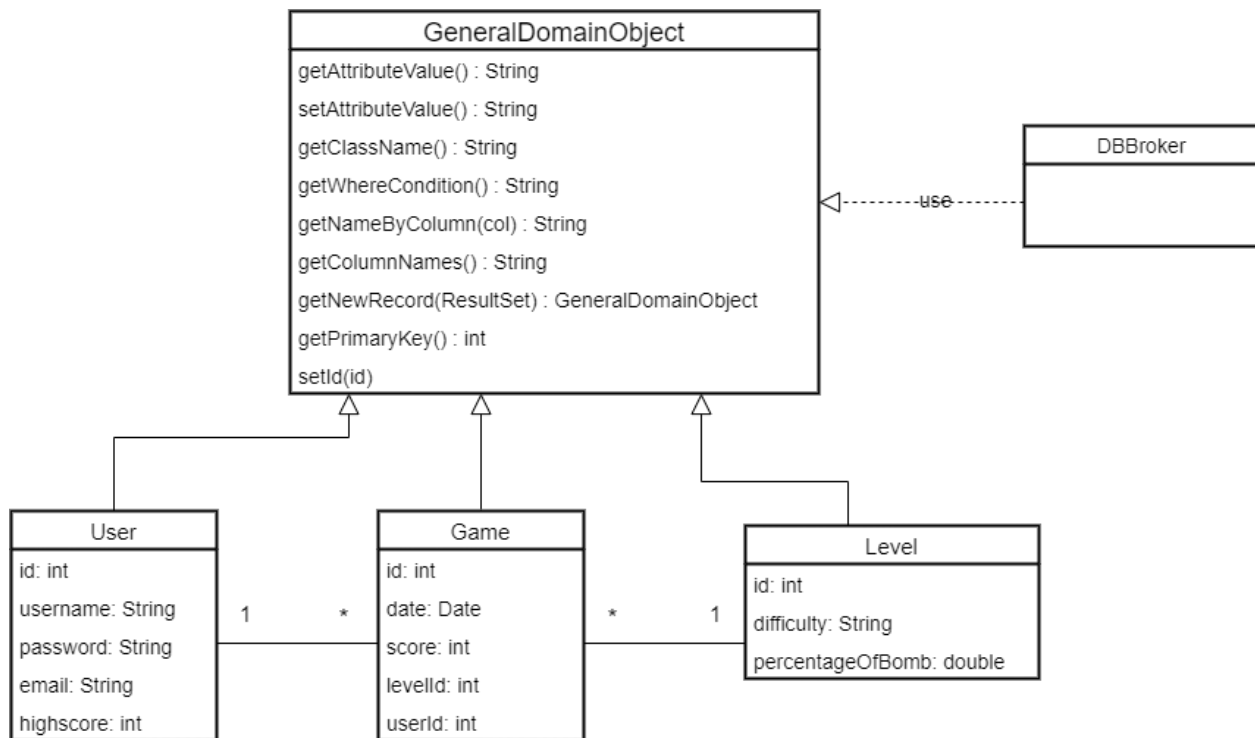
    SocketCommunication.getInstance().sendRequest(request);
    Response response = SocketCommunication.getInstance().readResponse();

    return response.getLevel();
}
```

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

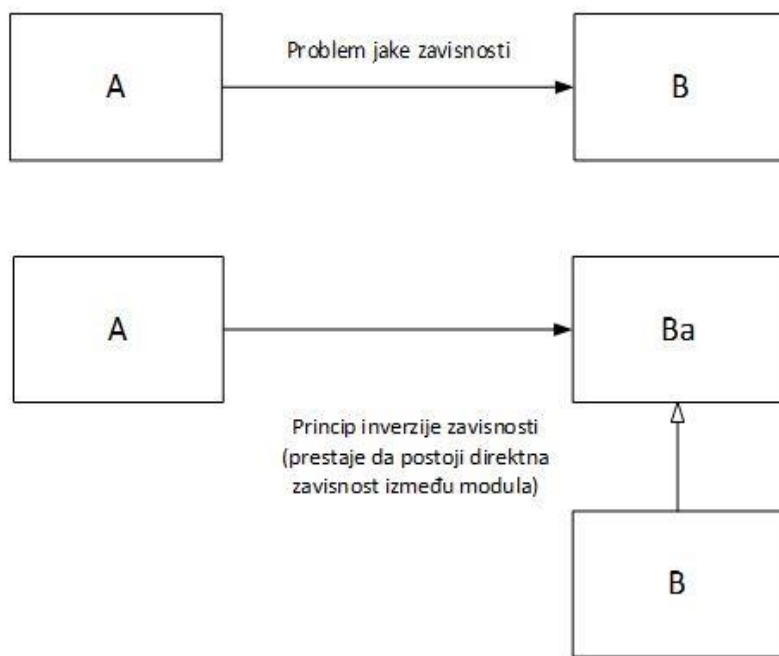
- Принцип инверзије зависности: зависи од апстракције а не од конкретизације.
- Модули вишег нивоа не треба да зависе од модула нижег нивоа
- Оба треба да зависе од апстракције
- Апстракције не треба да зависе од детаља
- Детаљи треба да зависе од апстракције

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



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.

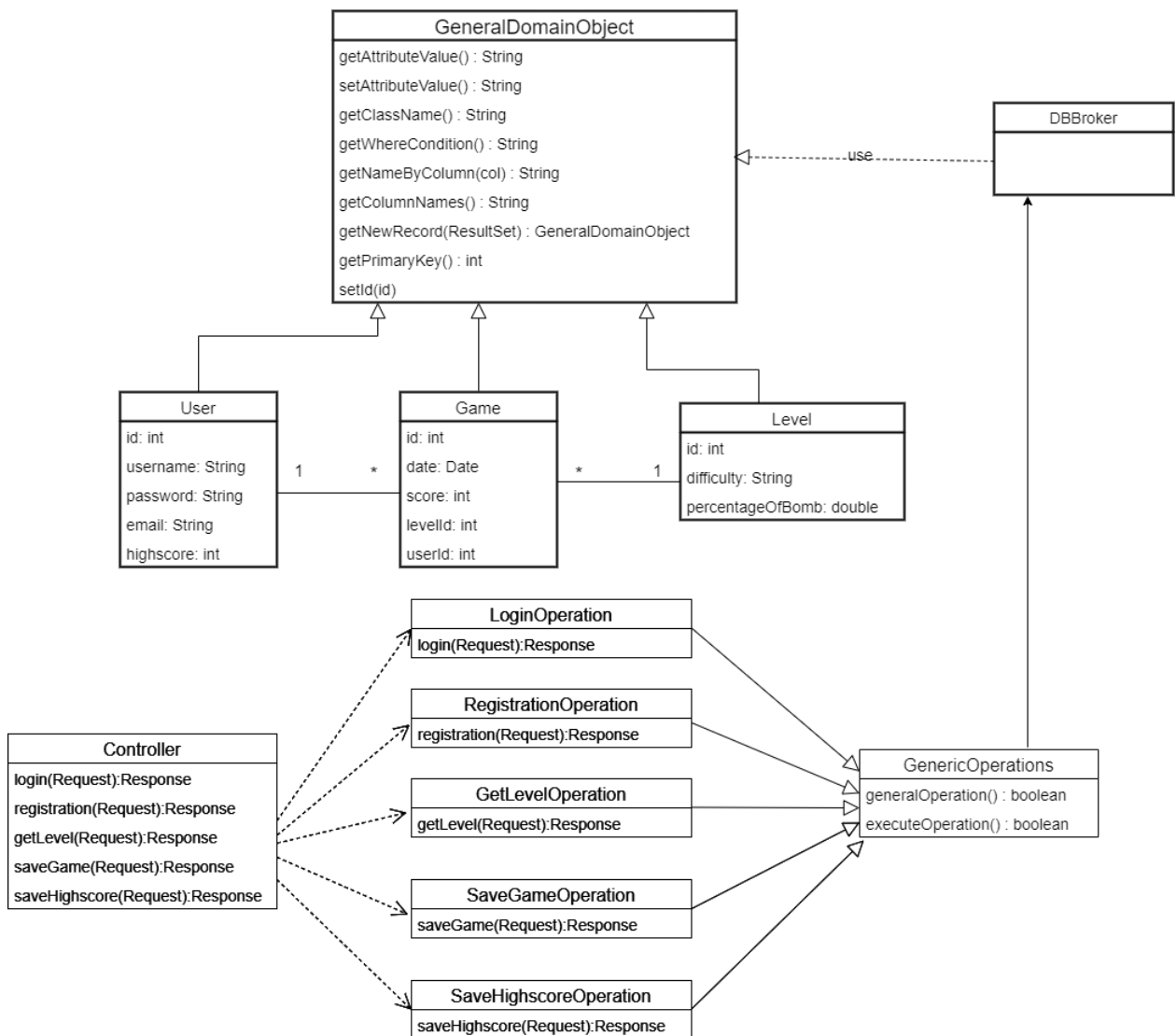
Директна веза између модула А и В се прекида и између њих се поставља модул вишег нивоа – модул Ва.

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

4. Принцип уметања зависности – *The dependency injection principle* :

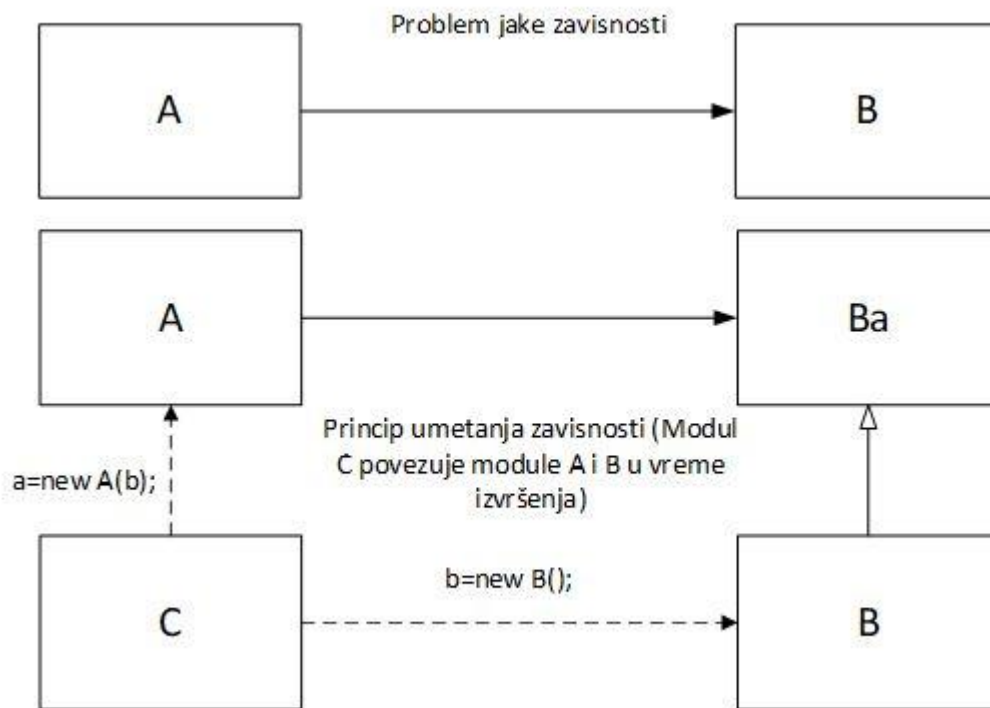
Зависности између две компонентне програма се успостављају у време извршења програма преко неке треће компоненте.

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



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.

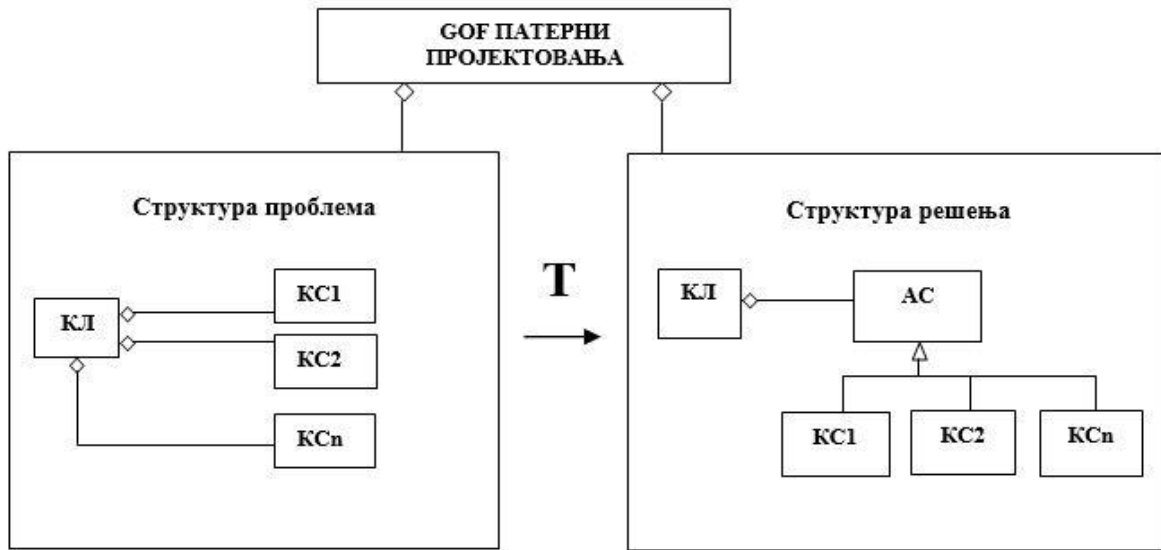
Оно што изводимо као закључак је да је патерн “троделно правило” које успоставља корелацију између одређеног контекста као система ограничења, проблема који делује у том контексту и решења, као структуре софтверског система која омогућава да односи између елемената тог система буду поново употребљиви.

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.



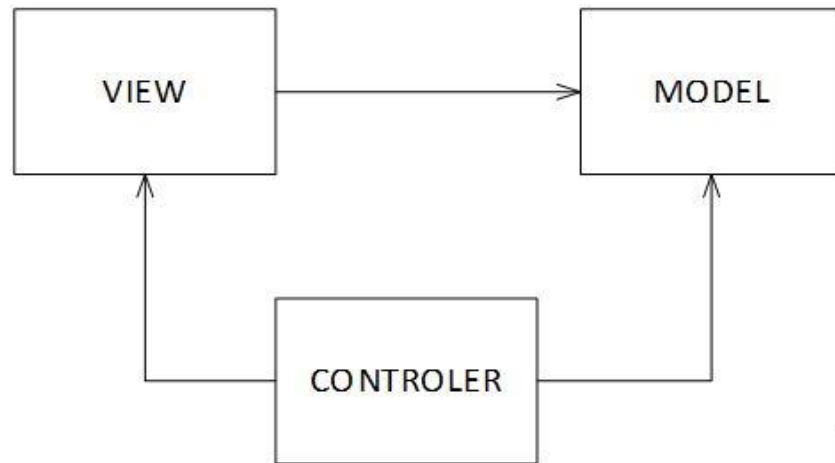
Слика 57. Општи облик GOF патерна пројектовања

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.

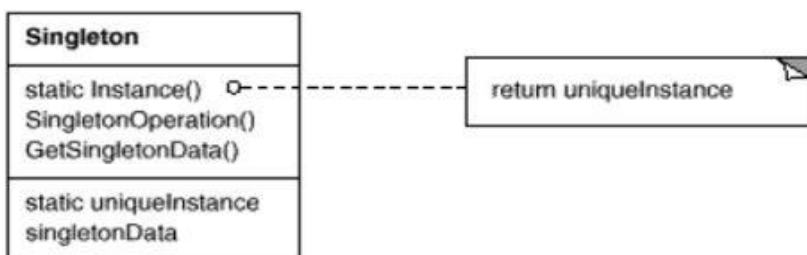
Патерни понашања описују начин на који класе и објекти сарађују и распоређују одговорности. У њих се сврставају *Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor* патерн.

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 GenericOperations go;  
    private List<Socket> clients;  
  
    private Controller() {  
    }  
  
    public static Controller getInstance() {  
  
        if (instance == null) {  
            instance = new Controller();  
        }  
  
        return instance;  
    }  
}
```

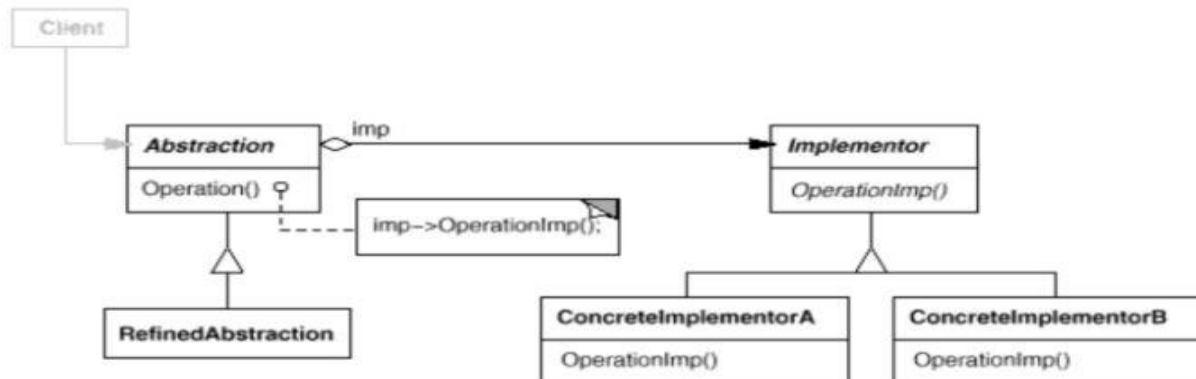
Слика 60. Singleton патерн – Controller

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

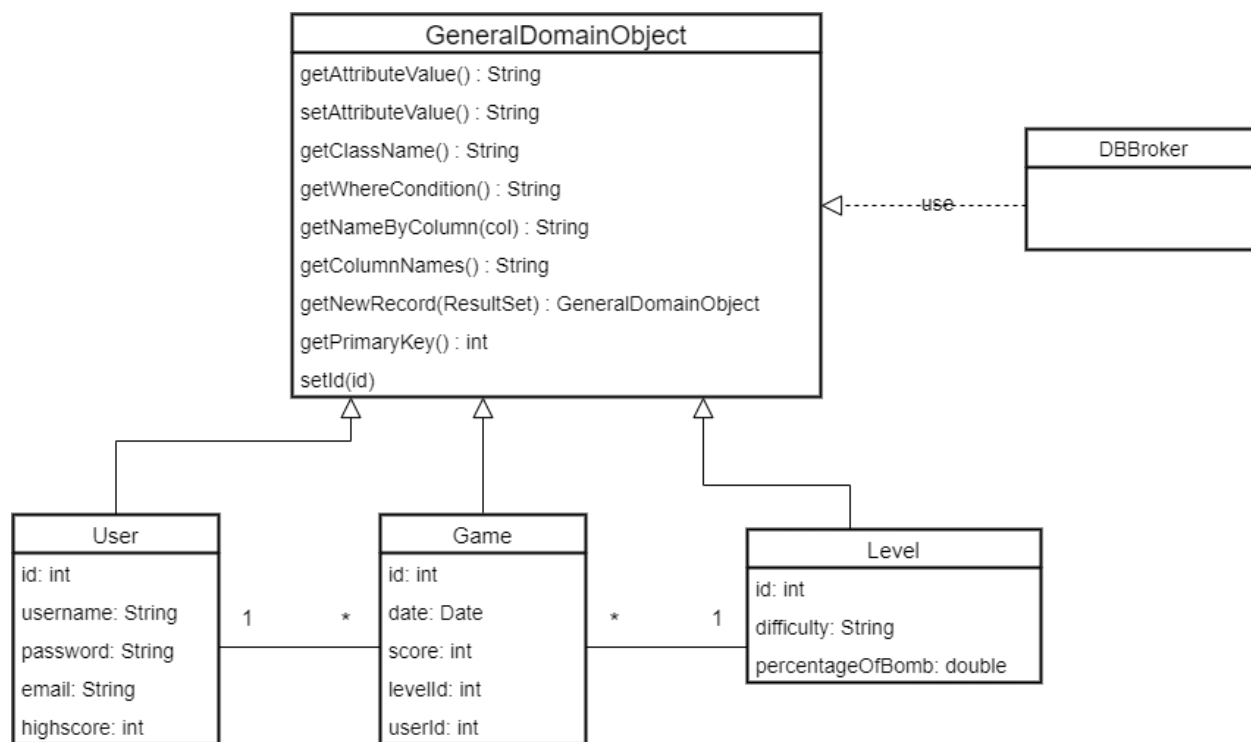
Захтев: Потребно је обезбедити апстрактне операције у класи брокера базе података, за које ће се тек у време извршења програма везати конкретне имплементације. Обезбедити генеричке методе за све доменске класе.

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



Слика 61: Bridge патерн

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



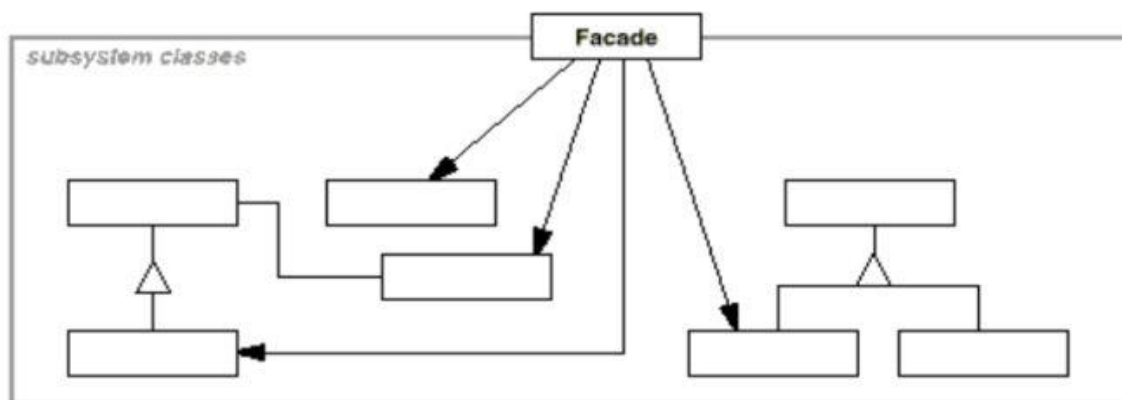
Слика 62 Bridge патерн – брокер базе података

Са приложених слика можемо закључити да је однос између брокера базе података и класе *GeneralDomainObject*, заправо однос између *Abstraction*-а и *Implementor*-а, односно закључујемо да брокер базе података има улогу *Abstraction*-а, а доменске класе улогу *ConcreteImplementor*-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 63. 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 уласка у систем и у којој се делегирају захтеви ка одговорним класама за њихову обраду, тј. класи *ClientThread*.

```

public class Server extends Thread {

    private boolean active = true;
    private int i;
    private List<ClientThread> clients;
    private List<Socket> clientSocket;
    private ServerSocket ss;

    private static Server instance;

    public List<ClientThread> getClients() {
        return clients;
    }

    public void setClients(List<ClientThread> clients) {
        this.clients = clients;
    }
}
  
```

```

@Override
public void run(){
    try {
        ss = new ServerSocket(9000);
        System.out.println("Server is up and running.");
        clients = new LinkedList<>();
        clientSocket = new LinkedList<>();
        this.i = 0;
        while(active){
            Socket socket = ss.accept();
            System.out.println("IP: " + socket.getInetAddress());
            System.out.println("Socket port: " + socket.getPort());
            ClientThread clientThread = new ClientThread(socket, this, i);
            clientThread.start();
            System.out.println("New Client");
            i++;
            clients.add(clientThread);
            clientSocket.add(socket);
        }
    } catch (IOException ex) {
        System.out.println("Server is stop working!");
    }
}

    public void stopServer() {
    try {
        active = false;
        ss.close();
        for (Socket socket : clientSocket) {
            socket.close();
        }

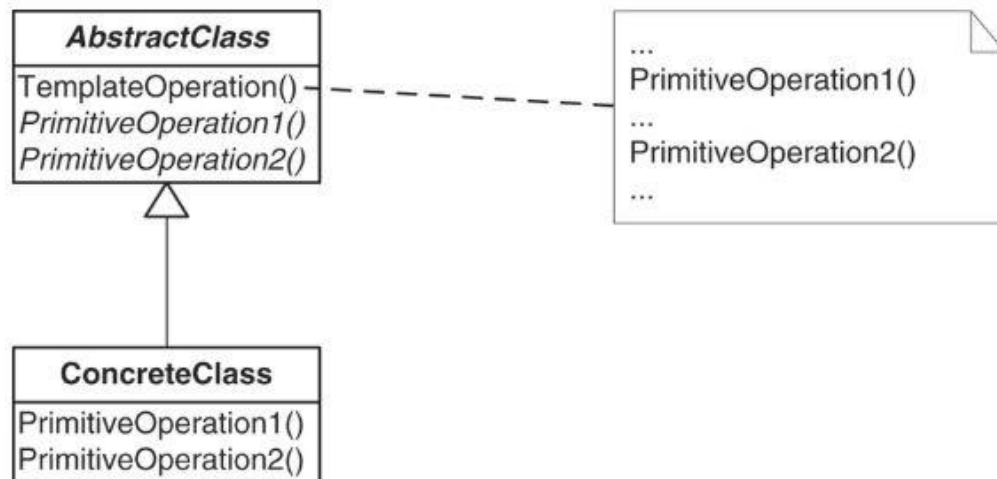
        System.out.println("Server socket is closed");
    } catch (IOException ex) {
    }
}
}

```

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 64. 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 *GenericOperations* и њена *template* метода - *generalOperation()*:


```

public abstract class GenericOperations {

static public DbBroker dbb = new DbBroker1();
    GeneralDomainObject gdo;
    synchronized public boolean generalOperation(){
        dbb.makeConnection();
        boolean signal = executeOperation();
        if (signal == true){
            dbb.commitTransation();
        }else{
            dbb.rollbackTransation();
        }
        dbb.closeConnection();
        return signal;
    }

    abstract public boolean executeOperation();
}

```

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

```

public class LoginOperation extends GenericOperations{

    Request request;
    Response response;

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

    @Override
    public boolean executeOperation() {
        System.out.println(request.getUser().getAttributeValue());
        User user = (User) dbb.findRecord(request.getUser());
        response.setUser(user);
        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.