Универзитет у Београду

Факултет организационих наука

Лабораторија за софтверско инжењерство

Предмет: Софтверски процес

Семинарски рад

Тема: Софтверски систем за управљање игрицом "Пут са замкама"

Професор:

др Синиша Влајић

Студент:

Лав Јовановић 3831/2024

Београд, 2025.

Садржај

1. Прикупљање корисничких захтева

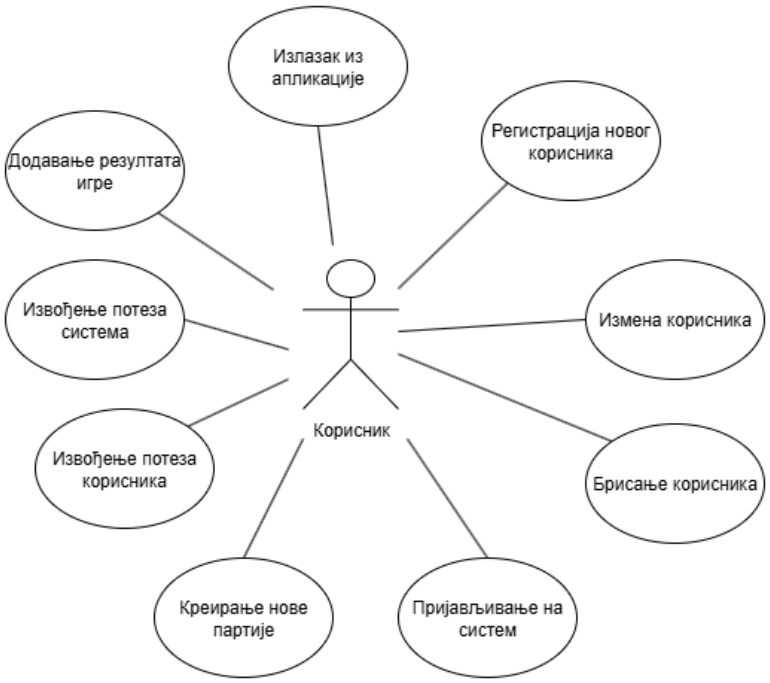
1.1. Вербални опис

Тема овог рада је израда информационог система који ће кориснику омогућити да игра игру **Пут са замкама**.
Систем треба да омогући кориснику регистрацију и пријављивање на систем. Након успешне пријаве корисника на систем, кориснику је омогућено да започне нову партију игре, као и да изврши увид у ранг листу на којој се налазе најбољи резултати (најмањи број потеза за долазак до циља). Подаци о кориснику, као и резултати партија, чувају се у бази података која се ажурира преко серверске стране.
Такође, корисник има могућност да започне нову игру кликом на дугме „Започни игру“. Када игра почне, на екрану се приказује матрица димензија 6x6, са јасно означеним пољем почетка.
У уводном потезу компјутер поставља замку на једно од поља матрице. Након тога игра се одвија наизменично по потезима — корисник прави свој потез тако што бира поље на које жели да се помери, а затим компјутер поставља нову замку на слободно поље.
Уколико корисник изабере поље на коме се налази замка, губи партију. Уколико корисник успе да стигне до крајњег (циљног) поља без да наиђе на замку, побеђује.
По завршетку партије, резултат (број потеза до циља) се уписује у базу података. Уколико је резултат бољи од неког постојећег на ранг листи, ранг листа се ажурира и корисник може видети свој пласман.
У оквиру менија система корисник може да приступи ранг листи, започне нову игру или изађе из апликације.

1.2. Случајеви коришћења

На основу вербалног описа, издвојени су следећи случајеви коришћења:

1. Регистрација новог корисника
2. Измена корисника
3. Брисање корисника
4. Пријављивање на систем
5. Креирање нове партије
6. Извођење потеза корисника
7. Извођење потеза система
8. Додавање резултата игре
9. Излазак из апликације



Слика 1. Модел случајева коришћења

СК 1: Случај коришћења – Регистрација новог корисника

Назив СК:
Регистрација новог корисника

Актори СК:
Корисник

Учесници СК:
Корисник и систем

Предуслов:
Систем је укључен и **корисник** није улогован.

Основни сценарио СК:

1. **Корисник** позива систем да региструје налог. (АПСО)
2. **Систем** генерише нови налог са јединственом корисничком шифром. (СО)
3. **Систем** приказује **кориснику** поруку: „**Корисник** је регистрован“. (ИА)

Алтернативни сценарио:

3.1. Уколико **систем** не може да региструје нови налог, он приказује **кориснику** поруку: „Систем не може да региструје новог корисника“ (ИА)

СК 2: Случај коришћења – Измена корисника

Назив СК:
Измена корисника

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и **корисник** је улогован на систем.

Основни сценарио СК:
1. **Корисник** уноси нове податке за постојећи налог. (АПУСО)
2. **Корисник** проверава тачност унетих података. (АНСО)
3. **Корисник** позива систем да ажурира налог. (АПСО)
4. **Систем** ажурира постојећи налог. (СО)
5. **Систем** приказује **кориснику** поруку: „**Корисник** је измењен“. (ИА)

Алтернативни сценарио:

5.1. Уколико **систем** не може да ажурира налог, он приказује **кориснику** поруку: „Систем не може да измени корисника“ (ИА)

СК 3: Случај коришћења – Брисање корисника

Назив СК:
Брисање корисника

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и **корисник** је улогован на систем.

Основни сценарио СК:
1. **Корисник** позива систем да обрише налог. (АПСО)
2. **Систем** брише постојећи налог. (СО)
3. **Систем** приказује **кориснику** поруку: „**Систем** је обрисао корисника“. (ИА)

Алтернативни сценарио:

3.1. Уколико **систем** не може да обрише налог, он приказује **кориснику** поруку: „Систем не може да обрише корисника“ (ИА)

СК 4: Случај коришћења – Пријављивање на систем

Назив СК:
Пријављивање на систем

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и **корисник** није улогован.

Основни сценарио СК:
1. **Корисник** уноси податке за пријављивање на **систем**. (АПУСО)
2. **Корисник** проверава тачност унетих података. (АНСО)
3. **Корисник** позива **систем** да га пријави на **систем**. (АПСО)
4. **Систем** проверава тачност унетих података. (СО)
5. **Систем** приказује **кориснику** поруку: „Корисник је успешно пријављен“. (ИА)

Алтернативни сценарио:

5.1. Уколико **систем** не може да пријави **корисника** на систем, он приказује **кориснику** поруку: „Систем не може да пријави корисника“ (ИА)

СК 5: Случај коришћења – Креирање нове партије

Назив СК:
Креирање нове партије

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и **корисник** је улогован на систем.

Основни сценарио СК:
1. **Корисник** уноси податке за креирање нове партије на **систем**. (АПУСО)
2. **Корисник** проверава тачност унетих података. (АНСО)
3. **Корисник** позива **систем** да креира нову партију. (АПСО)
4. **Систем** креира нову партију. (СО)
5. **Систем** приказује **кориснику** поруку „**Систем** је креирао нову партију”. (ИА)

Алтернативни сценарио:

5.1 Уколико **систем** не може да креира нову партију, он приказује **кориснику** поруку: „Систем не може да креира нову партију.“ (ИА)

СК 6: Случај коришћења – Извођење потеза корисника

Назив СК:
Извођење потеза корисника

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и **корисник** је улогован на систем. Приказан је кориснички интерфејс за игру "Пут са замкама".

Основни сценарио СК:
1. **Корисник** бира потез који жели да изведе. (АПУСО)
2. **Корисник** позива **систем** да изведе потез. (АПСО)
3. **Систем** проверава да ли је изабрано поље валидан потез. (СО)
4. **Систем** изводи потез и ажурира стање игре. (СО)
5. **Систем** приказује **кориснику** ажурирано стање игре и обавештава да је ред на систем. (ИА)

Алтернативни сценарио:

5.1 Уколико **систем** утврди да корисников потез није исправан, он приказује **кориснику** поруку: "Потез је илегалан". (ИА)

СК 7: Случај коришћења – Извођење потеза система

Назив СК:
Извођење потеза система

Актори СК:
Систем

Учесници СК:
Систем

Предуслов:
Систем је укључен и **корисник** је улогован на систем. Приказан је кориснички интерфејс за игру "Пут са замкама".

Основни сценарио СК:

1. **Систем** идентификује потез који жели да изведе. (СО)
2. **Систем** врши избор потеза. (СО)
3. **Систем** проверава да ли је изабрано поље валидан потез. (СО)
4. **Систем** изводи потез потез и ажурира стање игре. (СО)
5. **Систем** приказује **кориснику** ажурирано стање игре и обавештава да је ред на корисника. (ИА)

Алтернативни сценарио:

5.1 Уколико **систем** утврди његов потез није исправан, он приказује **кориснику** тренутно стање игре. (ИА)

СК 8: Случај коришћења – Додавање резултата игре

Назив СК:
Додавање резултата игре

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и игра је завршена.

Основни сценарио СК:
1. **Корисник** завршава игру одигравањем последњег потеза. (АПУСО)
2. **Корисник** позива **систем** да дода резултате игре. (АПСО)
3. **Систем** приказује **кориснику** резултат игре. (ИА)

СК 9: Случај коришћења – Излазак из апликације

Назив СК:
Излазак из апликације

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен.

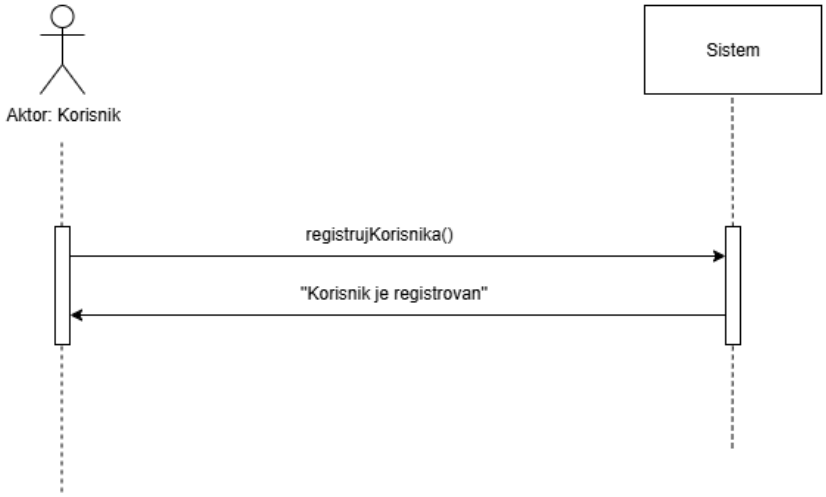
Основни сценарио СК:
1. **Корисник** позива **систем** да изађе из апликације. (АПСО)
2. **Систем** затвара апликацију и прекида своје извршавање. (СО)

2. Анализа

2.1. Системски дијаграм секвенци

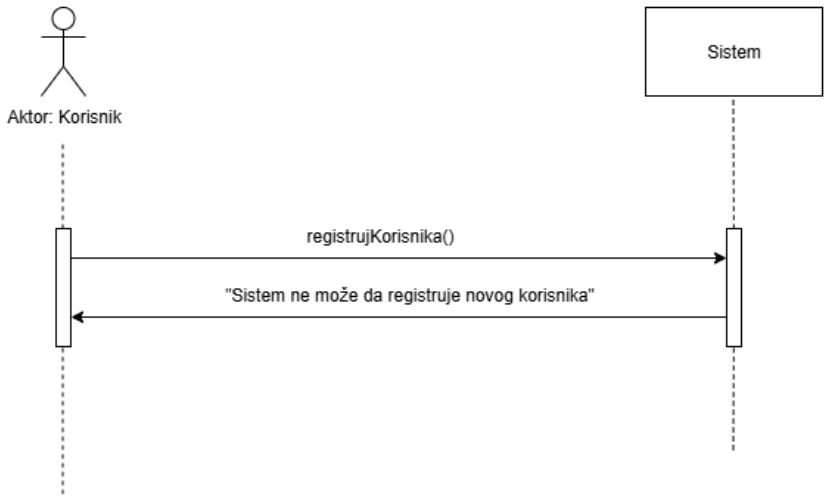
ДС1: Дијаграм секвенци случаја коришћења – Регистрација новог корисника

1. **Корисник** позива систем да региструје налог. (АПСО)
2. **Систем** приказује **кориснику** поруку: „**Корисник** је регистрован“. (ИА)



Алтернативни сценарио:

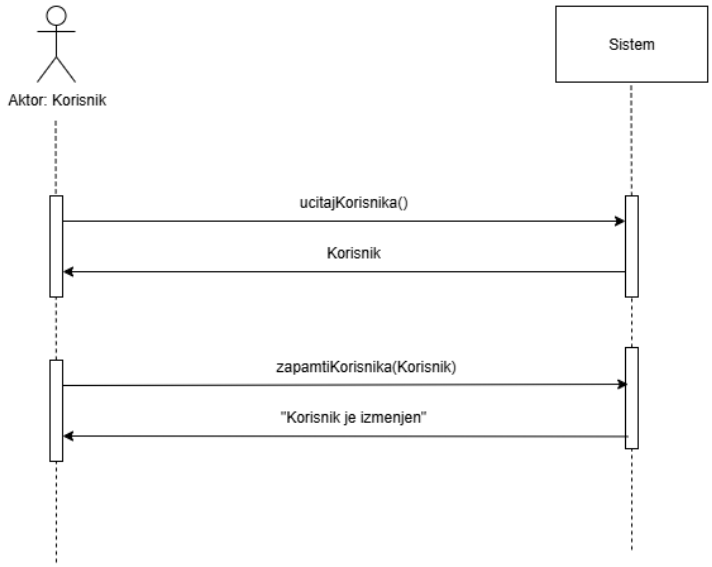
2.1. Уколико **систем** не може да региструје нови налог, он приказује **кориснику** поруку: „Систем не може да региструје новог корисника“ (ИА)



Са наведених секвенцних дијаграма уочава се 1 системска операција:
1. **Signal** registrujKorisnika()

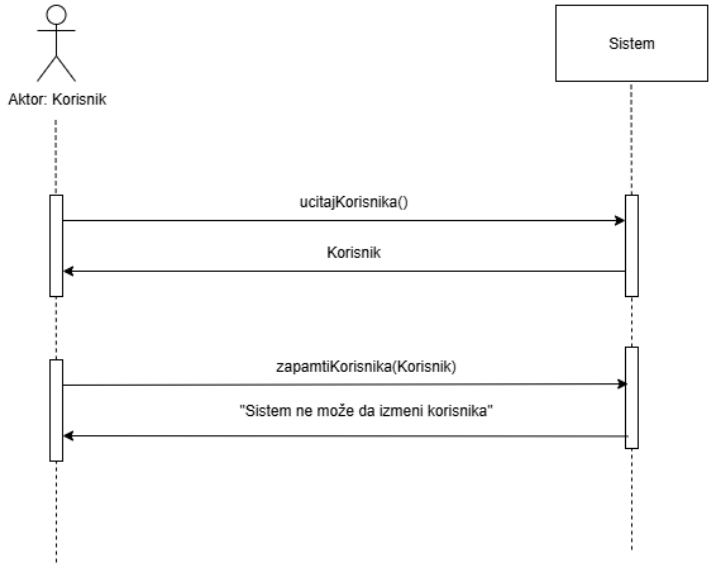
ДС2: Дијаграм секвенци случаја коришћења – Измена корисника

1. **Корисник** позива систем да ажурира налог. (АПСО)
2. **Систем** приказује **кориснику** поруку: „**Корисник** је измењен“. (ИА)



Алтернативни сценарио:

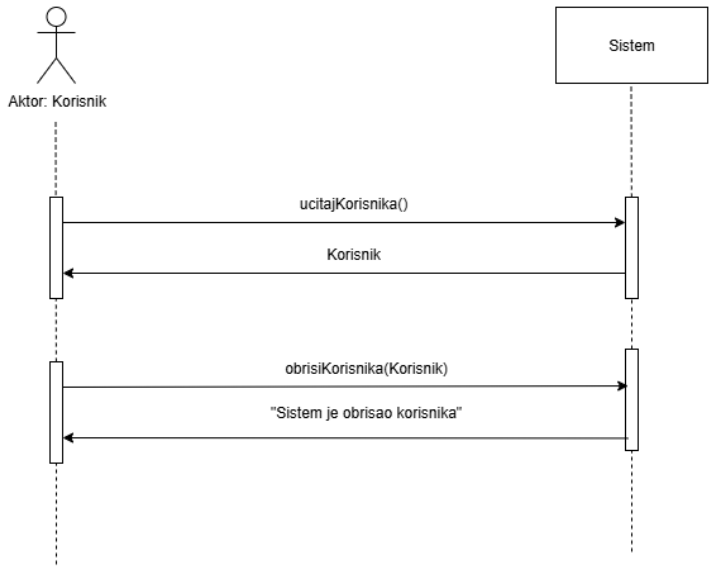
2.1. Уколико **систем** не може да ажурира налог, он приказује **кориснику** поруку: „Систем не може да измени корисника“ (ИА)



Са наведених секвенцних дијаграма уочава се 2 системске операције:
1. **Signal** zapamtiKorisnika(Korisnik)
2. **Signal** ucitajKorisnika()

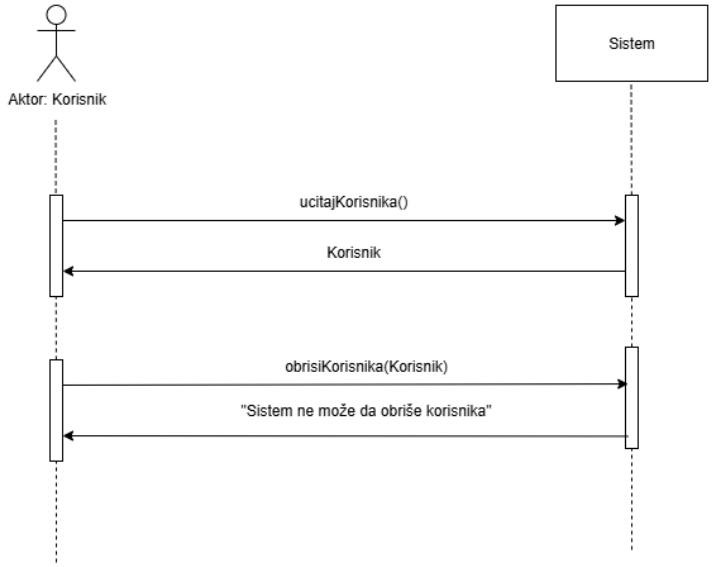
ДС3: Дијаграм секвенци случаја коришћења – Брисање корисника

1. **Корисник** позива систем да обрише налог. (АПСО)
2. **Систем** приказује **кориснику** поруку: „**Систем** је обрисао корисника“. (ИА)



Алтернативни сценарио:

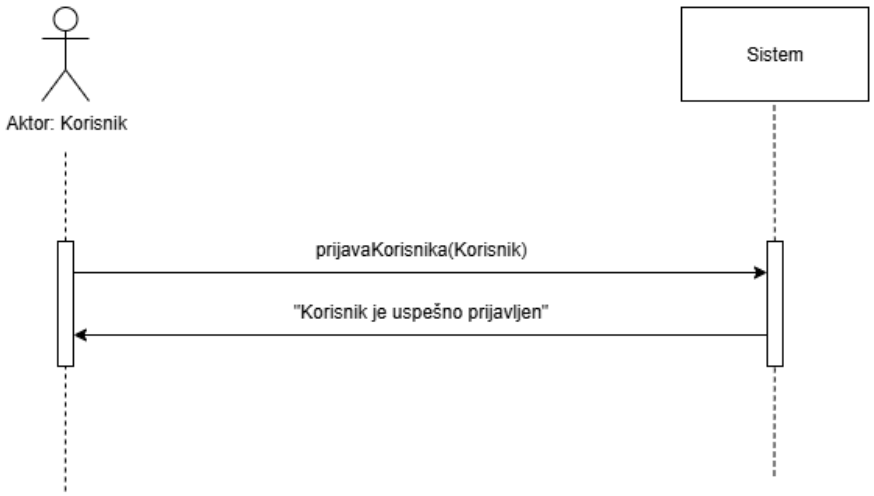
2.1. Уколико **систем** не може да обрише налог, он приказује **кориснику** поруку: „Систем не може да обрише корисника“ (ИА)



Са наведених секвенцних дијаграма уочава се 2 системске операције:
1. **Signal** obrisiKorisnika(Korisnik)
2. **Signal** ucitajKorisnika()

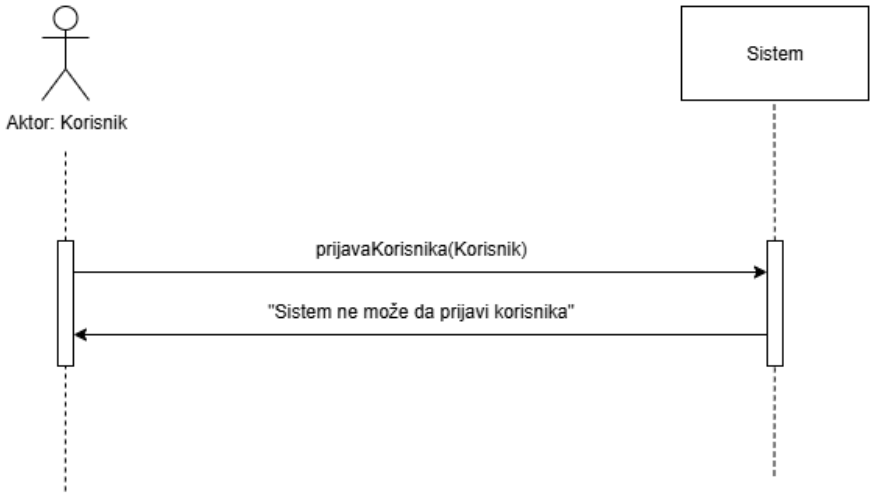
ДС4: Дијаграм секвенци случаја коришћења – Пријављивање корисника

1. **Корисник** позива **систем** да га пријави на **систем**. (АПСО)
2. **Систем** приказује **кориснику** поруку: „Корисник је успешно пријављен“. (ИА)



Алтернативни сценарио:

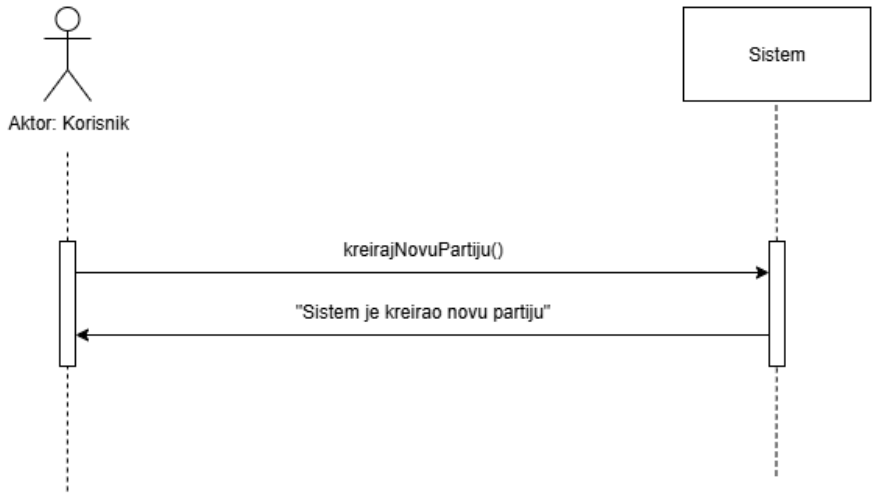
2.1. Уколико **систем** не може да пријави **корисника** на систем, он приказује **кориснику** поруку: „Систем не може да пријави корисника“ (ИА)



Са наведених секвенцних дијаграма уочава се 1 системска операција:
1. **Signal** prijavaKorisnika(Korisnik)

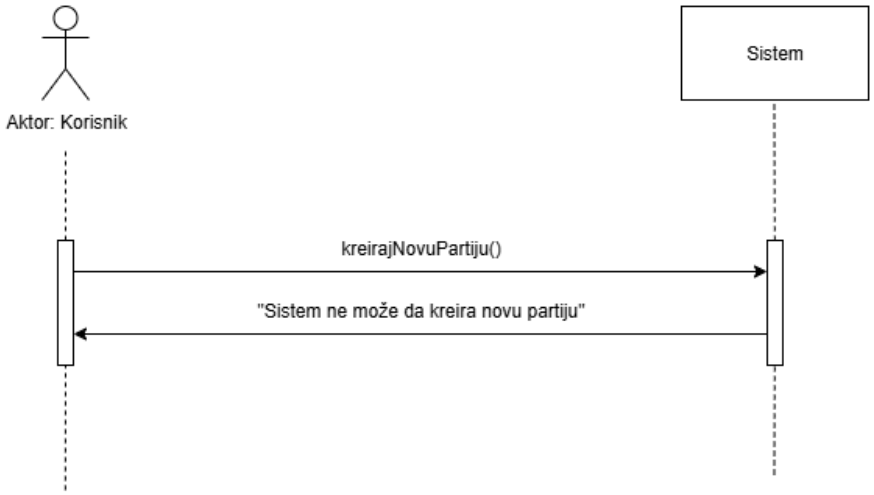
ДС5: Дијаграм секвенци случаја коришћења – Креирање нове партије

1. **Корисник** позива **систем** да креира нову партију. (АПСО)
2. **Систем** приказује **кориснику** поруку „**Систем** је креирао нову партију”. (ИА)



Алтернативни сценарио:

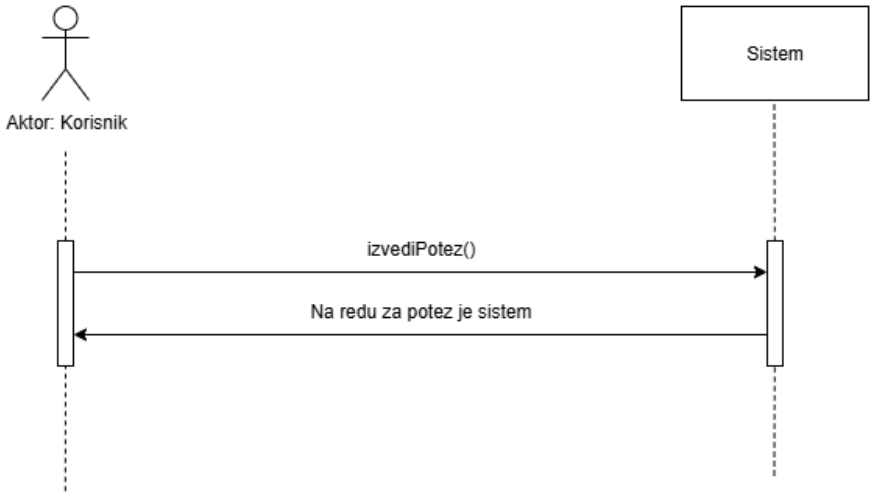
2.1. Уколико **систем** не може да креира нову партију, он приказује **кориснику** поруку: „Систем не може да креира нову партију.“ (ИА)



Са наведених секвенцних дијаграма уочава се 1 системска операција:
1. **Signal** kreirajNovuPartiju()

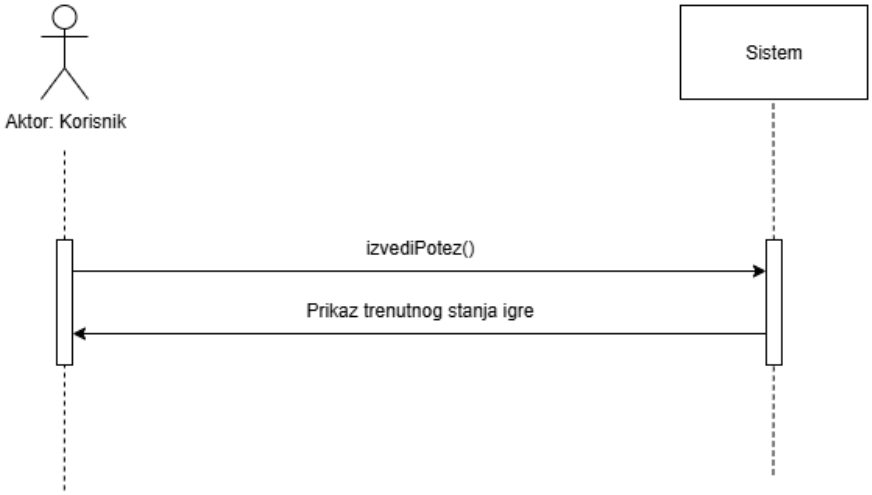
ДС6: Дијаграм секвенци случаја коришћења – Извођење потеза корисника

1. **Корисник** позива **систем** да изведе потез. (АПСО)
2. **Систем** приказује **кориснику** ажурирано стање игре и обавештава да је ред на систем. (ИА)



Алтернативни сценарио:

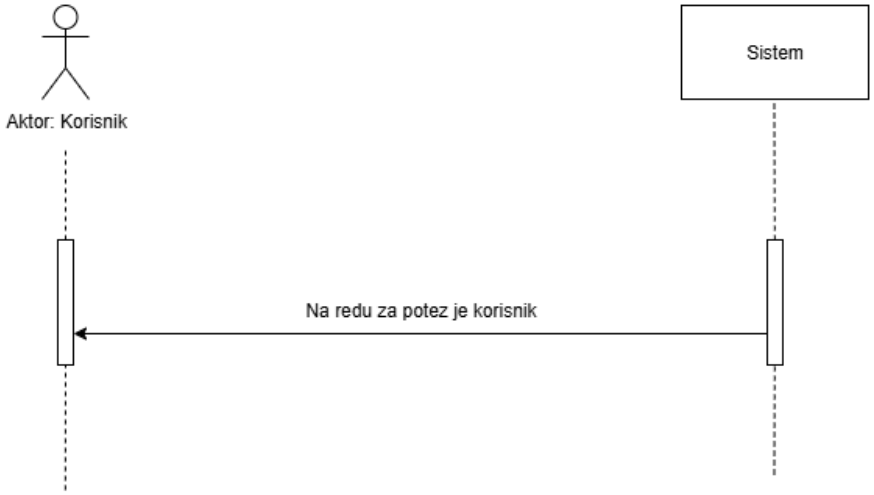
2.1 Уколико **систем** утврди да корисников потез није исправан, он приказује **кориснику** поруку: "Потез је илегалан". (ИА)



Са наведених секвенцних дијаграма уочава се 1 системска операција:
1. **Signal** izvediPotez()

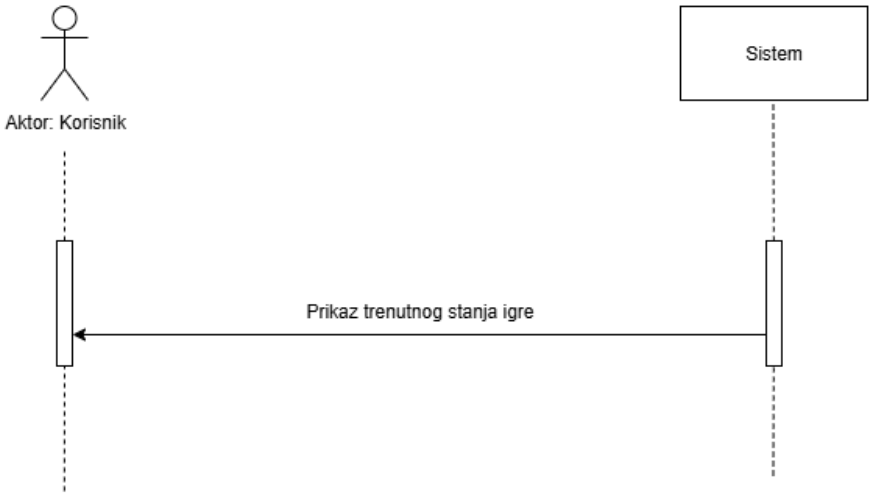
ДС7: Дијаграм секвенци случаја коришћења – Извођење потеза система

1. **Систем** приказује **кориснику** ажурирано стање игре и обавештава да је ред на корисника. (ИА)



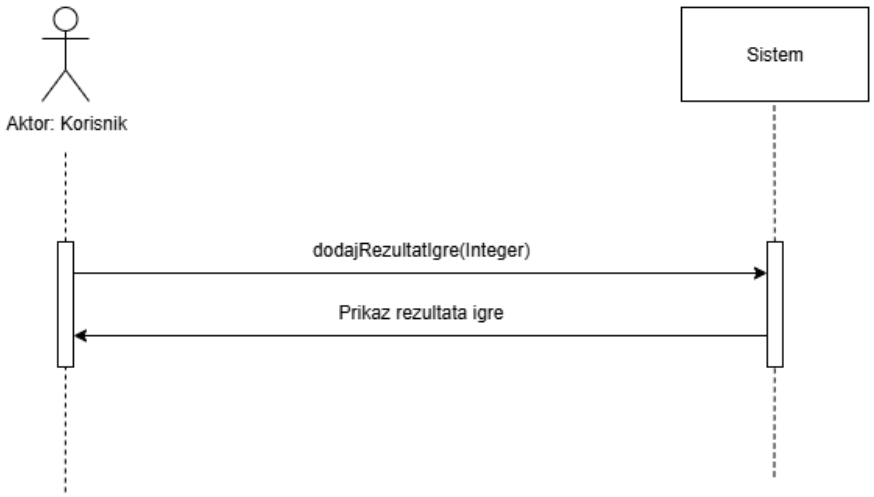
Алтернативни сценарио:

1.1 Уколико **систем** утврди његов потез није исправан, он приказује **кориснику** тренутно стање игре. (ИА)



ДС8: Дијаграм секвенци случаја коришћења – Додавање резултата игре

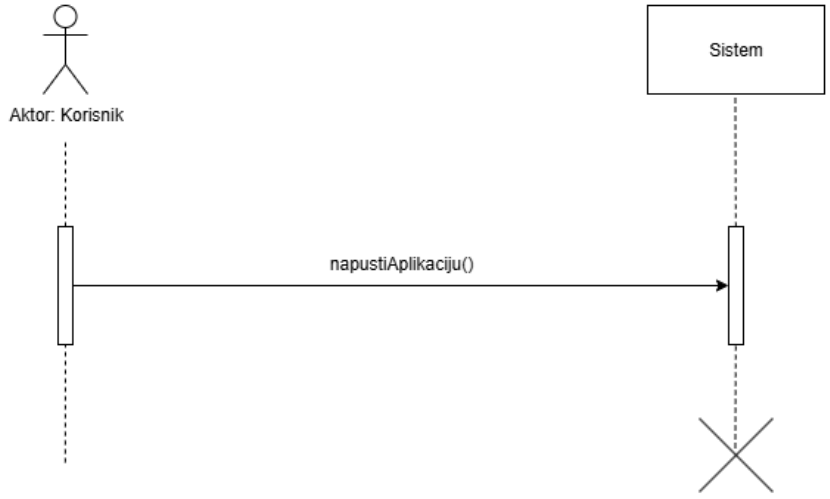
1. **Корисник** позива **систем** да дода резултате игре. (АПСО)
2. **Систем** приказује **кориснику** резултат игре. (ИА)



Са наведених секвенцних дијаграма уочава се 1 системска операција:
1. **Signal** dodajRezultatIgre(Integer)

ДС9: Дијаграм секвенци случаја коришћења – Излазак из апликације

1. **Корисник** позива **систем** да изађе из апликације. (АПСО)



Са наведених секвенцних дијаграма уочава се 1 системска операција:
1. **Signal** napustiAplikaciju()

На основу анализе сценарија уочено је 9 системских операција:

1. **Signal** registrujKorisnika()
2**. Signal** zapamtiKorisnika(Korisnik)
3. **Signal** ucitajKorisnika()
4**. Signal** obrisiKorisnika(Korisnik)
5**. Signal** prijavaKorisnika(Korisnik)
6. **Signal** kreirajNovuPartiju()
7. **Signal** izvediPotez()
8**. Signal** dodajRezultatIgre(Integer)
9**. Signal** napustiAplikaciju()

2.2. Понашање софтверског система – Дефинисање уговора о системским операцијама

1. Уговор УГ1: registrujKorisnika(): Signal;
   Веза са СК: СК1
   Предуслови: /
   Постуслови: Корисник је регистрован

2. Уговор УГ2: zapamtiKorisnika(Korisnik): Signal;
   Веза са СК: СК2
   Предуслови: Вредносна и структурна ограничења над објектом Korisnik морају бити задовољена
   Постуслови: Корисник је запамћен

3. Уговор УГ3: ucitajKorisnika(): Signal;
   Веза са СК: СК2, СК3
   Предуслови: Корисник је претходно креиран
   Постуслови: /

4. Уговор УГ4: obrisiKorisnika(Korisnik): Signal;
   Веза са СК: СК3
   Предуслови: Структурна ограничења над објектом Korisnik морају бити задовољена
   Постуслови: Корисник је обрисан

5. Уговор УГ5: prijavaKorisnika(Korisnik): Signal;
   Веза са СК: СК4
   Предуслови: /
   Постуслови: Корисник је пријављен на систем

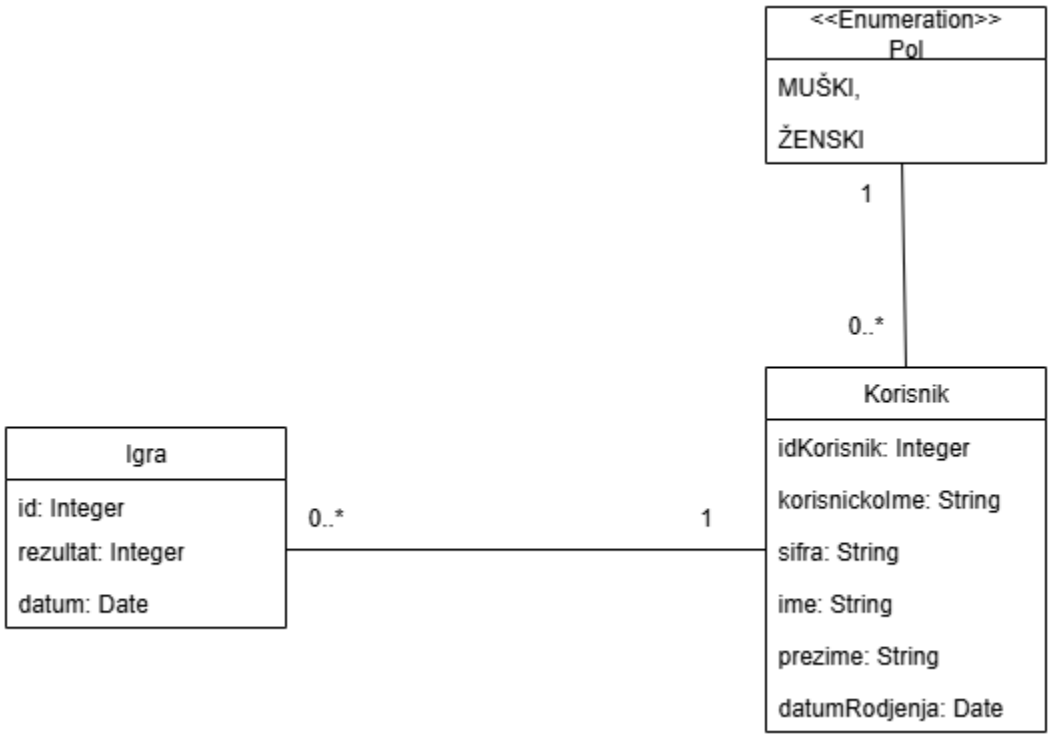
6. Уговор УГ6: kreirajNovuPartiju(): Signal;
   Веза са СК: СК5
   Предуслови: Корисник је претходно пријављен на систем
   Постуслови: Нова партија је креирана

7. Уговор УГ7: izvediPotez(): Signal;
   Веза са СК: СК6
   Предуслови: Корисник је претходно пријављен на систем и нова партија је креирана
   Постуслови: /

8. Уговор УГ8: dodajRezultatIgre(Integer): Signal;
   Веза са СК: СК8
   Предуслови: Корисник је претходно пријављен на систем, нова партија је креирана и корисник је одиграо последњи потез
   Постуслови: Подаци о резултату игре су запамћени

9. Уговор УГ9: napustiAplikaciju(): Signal;
   Веза са СК: СК9
   Предуслови: /
   Постуслови: Корисник је одјављен из апликације и корисничка апликација се затвара

2.3. Структура софтверског система – Концептуални(доменски) модел



2.4. Структура софтверског система – Релациони модел

Korisnik (idKorisnik, korisnickoIme, sifra, ime, prezime, datumRodjenja, pol)
Igra (id, rezultat, datum, *korisnikId*)

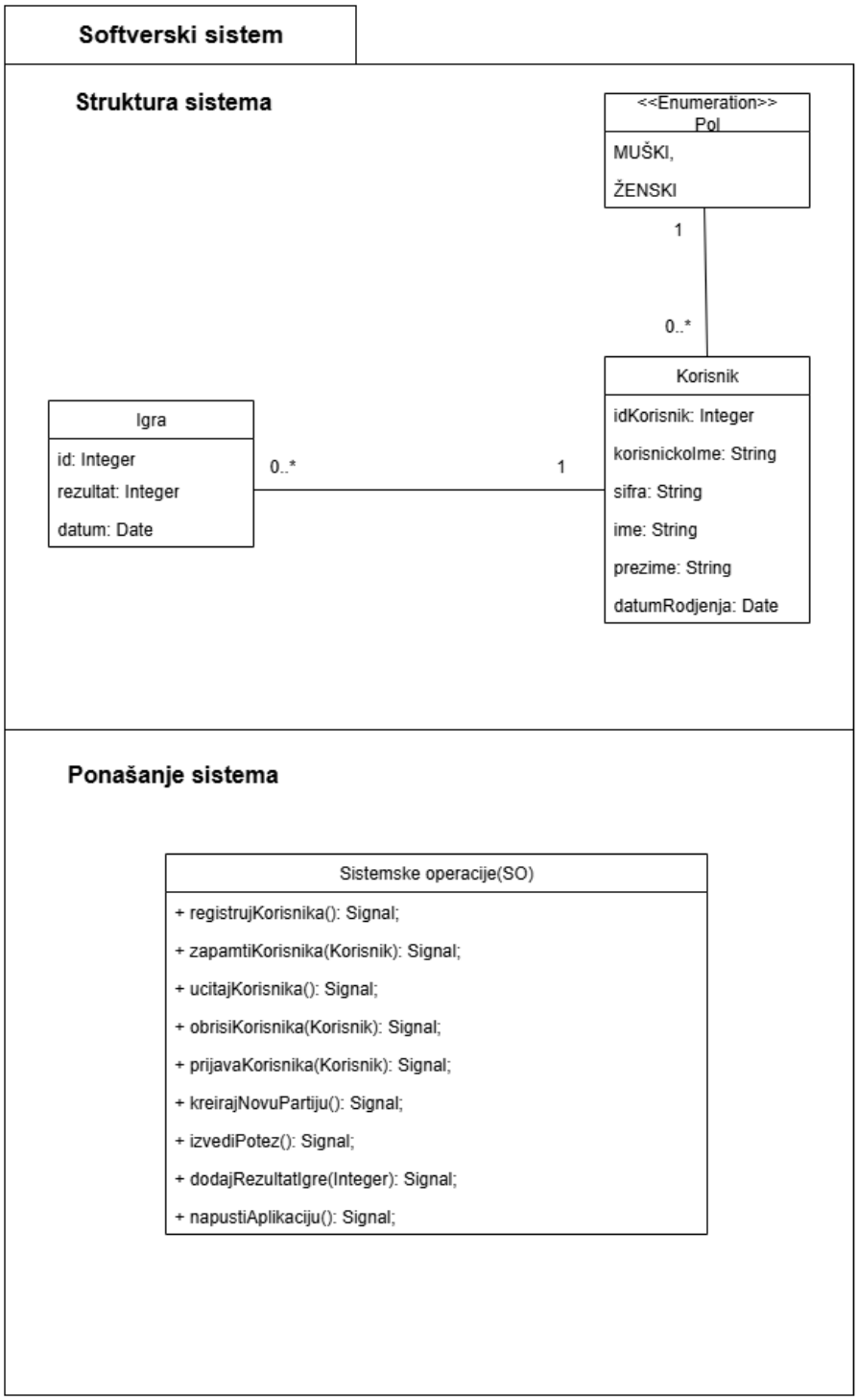
<table>
<tr><td colspan="2">Tabela Korisnik</td><td colspan="2">Prosto vrednosno ograničenje</td><td>Struktuno ograničenje</td></tr>
<tr><td rowspan="8">Atributi</td><td>Ime</td><td>Tip</td><td>Vrednost</td><td rowspan="8">INSERT /

UPDATE CASCADES Igra

DELETE RESTRICTED Igra</td></tr>
<tr><td>idKorisnik</td><td>Integer</td><td>NOT NULL AND >0</td></tr>
<tr><td>korisnickoIme</td><td>String</td><td></td></tr>
<tr><td>sifra</td><td>String</td><td></td></tr>
<tr><td>ime</td><td>String</td><td></td></tr>
<tr><td>prezime</td><td>String</td><td></td></tr>
<tr><td>datumRodjenja</td><td>Date</td><td></td></tr>
<tr><td>pol</td><td>Enum</td><td></td></tr>
</table>

Tabela Igra		Prosto vrednosno ograničenje		Strukturno ograničenje
Atributi	Ime	Tip	Vrednost	INSERT RESTRICTED Korisnik UPDATE RESTRICTED Korisnik DELETE /
	id	Integer	NOT NULL AND >0	
	rezultat	Integer		
	datum	Date		
	korisnikId	Integer	NOT NULL AND >0	

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



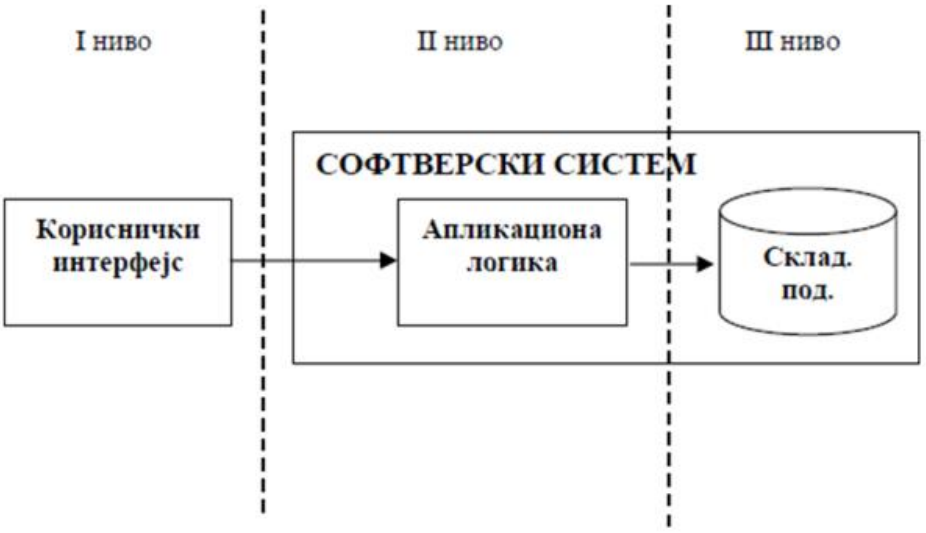
3. Пројектовање

Фаза пројектовања описује физичку структуру и понашање софтверског система. Пројектовање архитектуре софтверског система обухвата пројектовање корисничког интерфејса (пројектовање контролера корисничког интерфејса и екранских форми), апликационе логике (пројектовање контролера апликационе логике и пословне логике) и складишта података (брокер базе података).

Архитектура система је тронивојска и састоји се од следећих нивоа:

- кориснички интерфејс
- апликациона логика
- складиште података

Ниво корисничког интерфејса ја на страни клијента, док су апликациона логика и складиште на страни сервера.

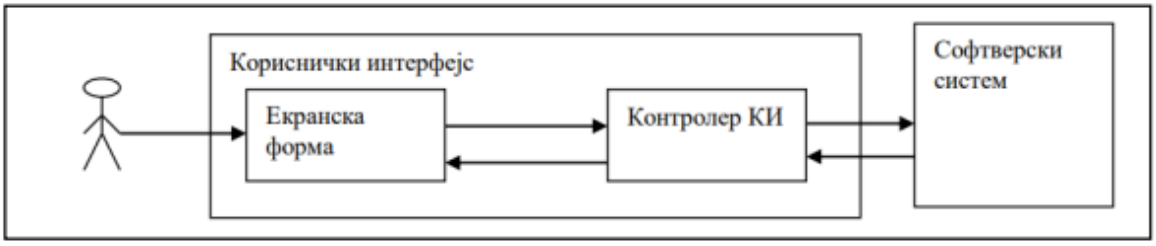


3.1 Пројектовање корисничког интерфејса

Кориснички интерфејс представља улазно-излазну реализацију софтверског система.

Састоји се од:

1. Екранске форме
2. Контролера корисничког интерфејса



3.1.1 Пројектовање екранских форми

Кориснички интерфејс је дефинисан преко скупа екранских форми. Сценарио коришћења екранских форми је директно повезан са сценаријима случајева коришћења.

На серверској страни није пројектована ниједна екранска форма. Серверска апликација се повезује са web serverom и након тога ни администратор ни корисник немају више било какву повезаност са серверском страном.

Приликом покретања клијентске стране добија се екранска форма која тражи пријављивање корисника.

СК 1: Случај коришћења – Регистрација новог корисника

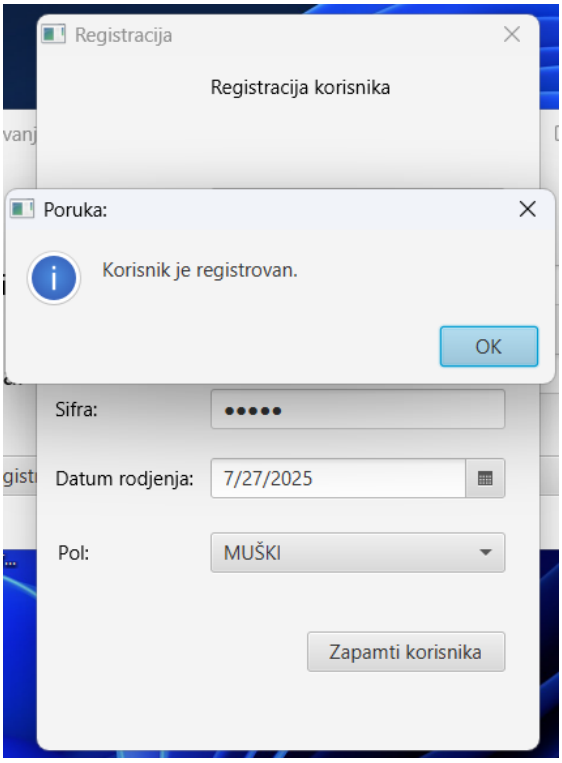
Назив СК:
Регистрација новог корисника

Актори СК:
Корисник

Учесници СК:
Корисник и систем

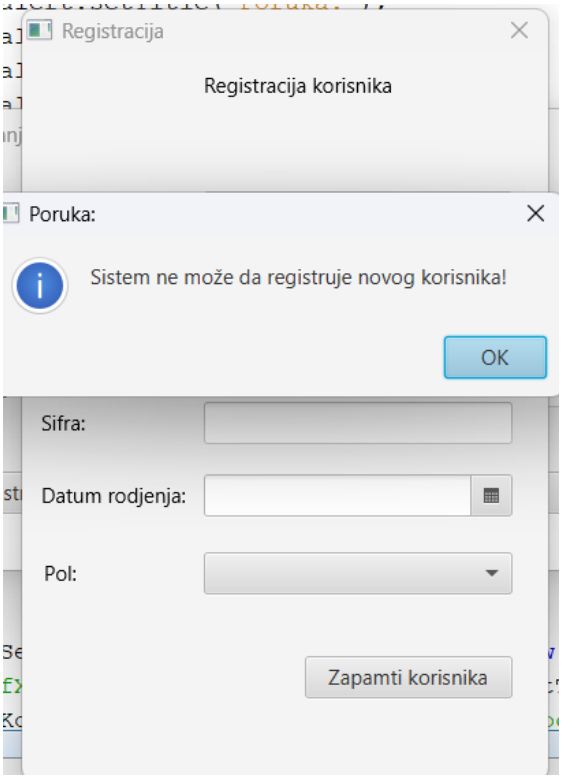
Предуслов:
Систем је укључен и **корисник** није улогован.

Основни сценарио СК:
1. **Корисник** позива систем да региструје налог. (АПСО)
2. **Систем** генерише нови налог са јединственом корисничком шифром. (СО)
3. **Систем** приказује **кориснику** поруку: „**Корисник** је регистрован“. (ИА)



Алтернативни сценарио:

1.1. Уколико **систем** не може да региструје нови налог, он приказује **кориснику** поруку: „Систем не може да региструје новог корисника“ (ИА)



СК 2: Случај коришћења – Измена корисника

Назив СК:
Измена корисника

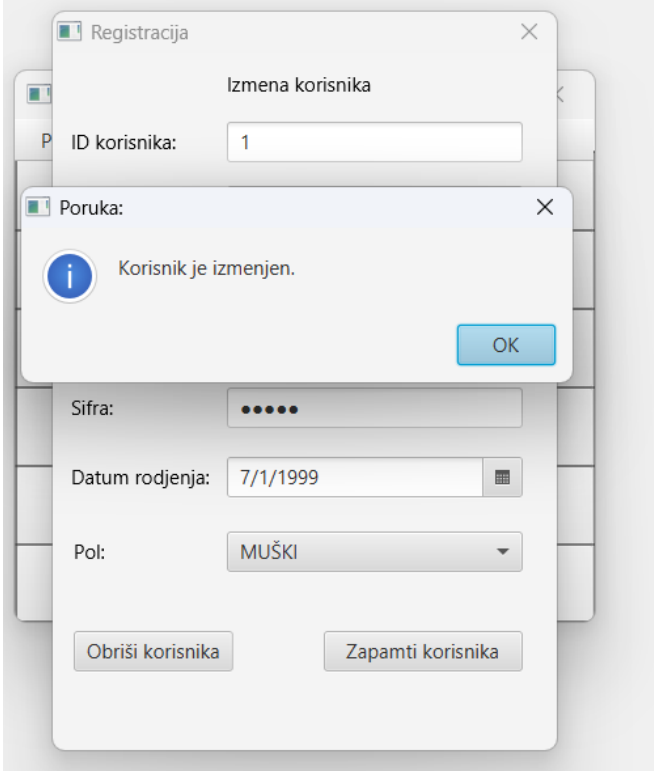
Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и **корисник** је улогован на систем.

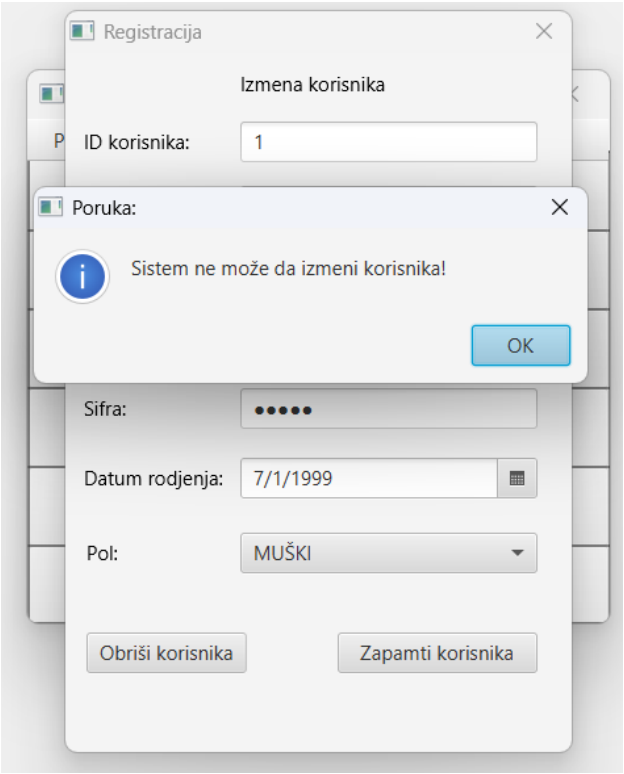
Основни сценарио СК:

1. **Корисник** уноси нове податке за постојећи налог. (АПУСО)
2. **Корисник** проверава тачност унетих података. (АНСО)
3. **Корисник** позива систем да ажурира налог. (АПСО)
4. **Систем** ажурира постојећи налог. (СО)
5. **Систем** приказује **кориснику** поруку: „**Корисник** је измењен“. (ИА)



Алтернативни сценарио:

5.1. Уколико **систем** не може да ажурира налог, он приказује **кориснику** поруку: „Систем не може да измени корисника“ (ИА)



СК 3: Случај коришћења – Брисање корисника

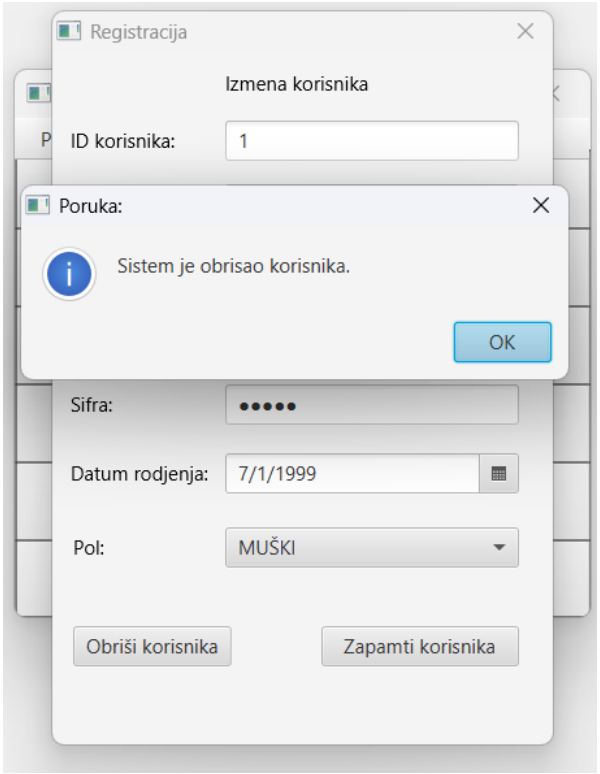
Назив СК:
Брисање корисника

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

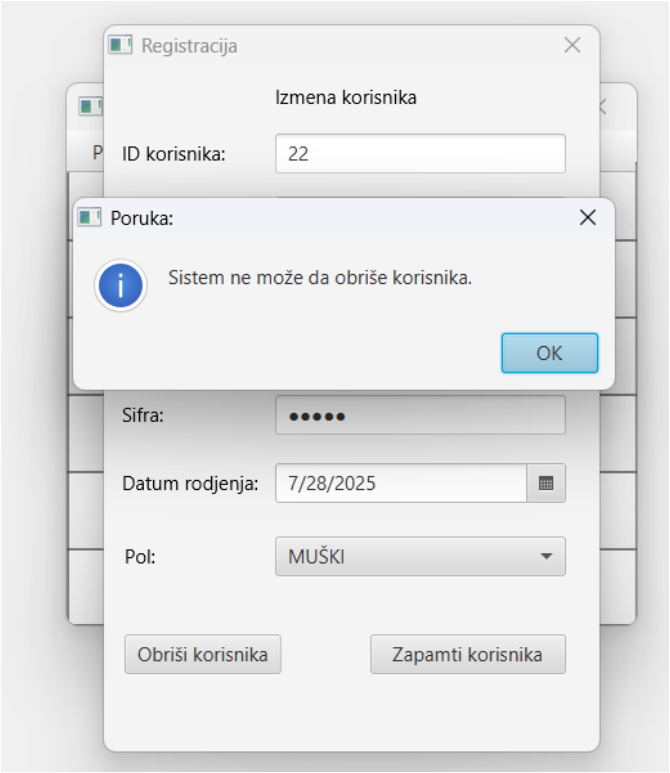
Предуслов:
Систем је укључен и **корисник** је улогован на систем.

Основи сценарио СК:
1. **Корисник** позива систем да обрише налог. (АПСО)
2. **Систем** брише постојећи налог. (СО)
3. **Систем** приказује **кориснику** поруку: „**Систем** је обрисао корисника“. (ИА)



Алтернативни сценарио:

3.1. Уколико **систем** не може да обрише налог, он приказује **кориснику** поруку: „Систем не може да обрише корисника“ (ИА)



СК 4: Случај коришћења – Пријављивање на систем

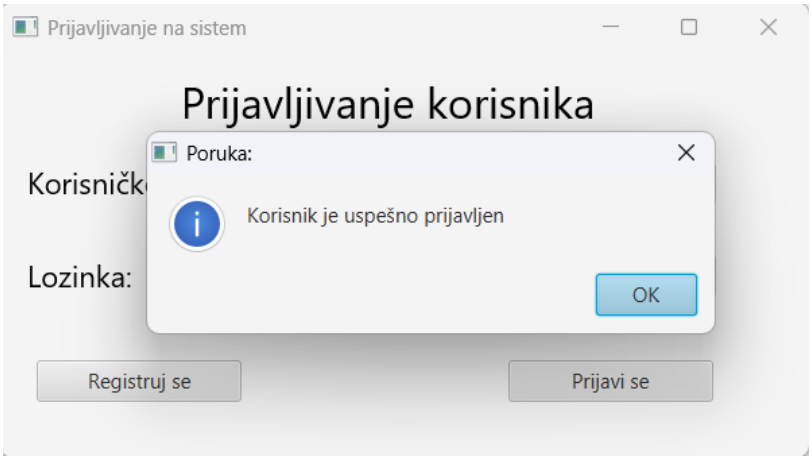
Назив СК:
Пријављивање на систем

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

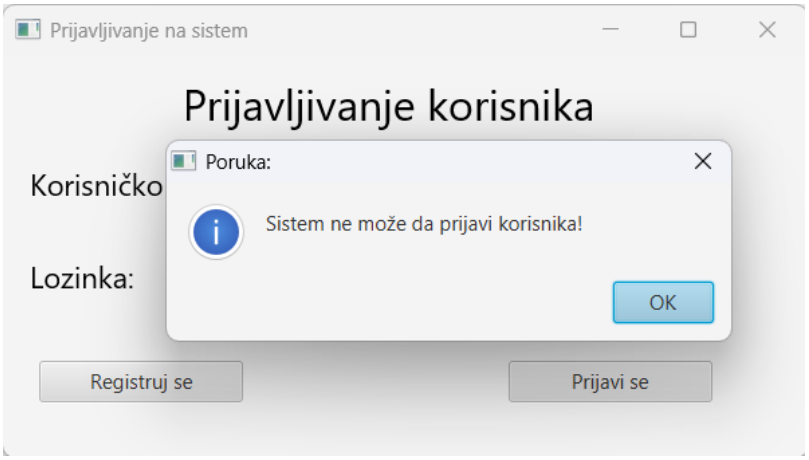
Предуслов:
Систем је укључен и **корисник** није улогован.

Основни сценарио СК:
1. **Корисник** уноси податке за пријављивање на **систем**. (АПУСО)
2. **Корисник** проверава тачност унетих података. (АНСО)
3. **Корисник** позива **систем** да га пријави на **систем**. (АПСО)
4. **Систем** проверава тачност унетих података. (СО)
5. **Систем** приказује **кориснику** поруку: „Корисник је успешно пријављен“. (ИА)



Алтернативни сценарио:

5.1. Уколико **систем** не може да пријави **корисника** на систем, он приказује **кориснику** поруку: „Систем не може да пријави корисника“ (ИА)



СК 5: Случај коришћења – Креирање нове партије

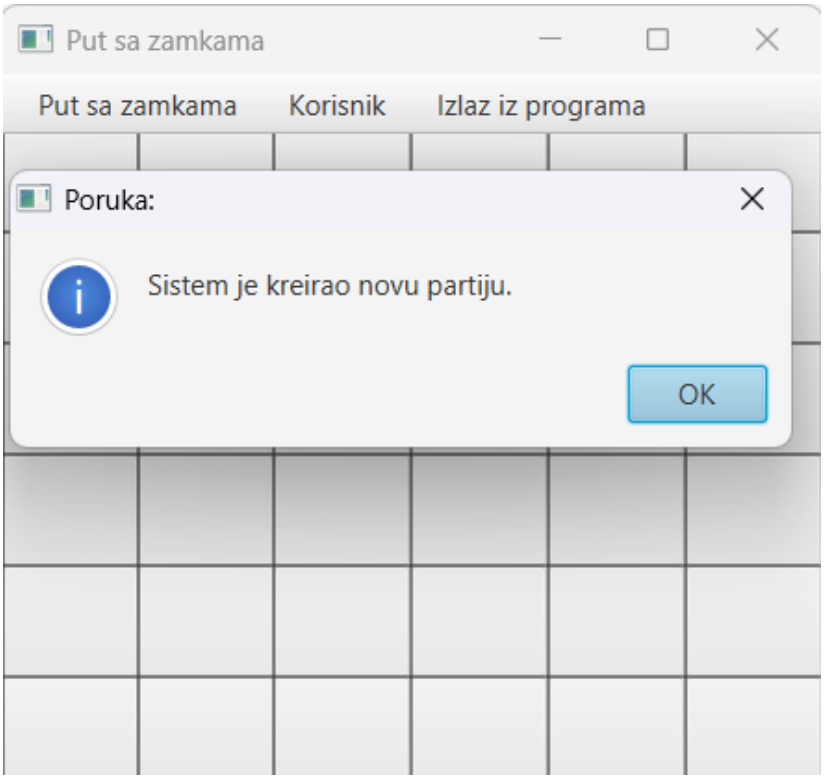
Назив СК:
Креирање нове партије

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и **корисник** је улогован на систем.

Основни сценарио СК:
1. **Корисник** уноси податке за креирање нове партије на **систем**. (АПУСО)
2. **Корисник** проверава тачност унетих података. (АНСО)
3. **Корисник** позива **систем** да креира нову партију. (АПСО)
4. **Систем** креира нову партију. (СО)
5. **Систем** приказује **кориснику** поруку „**Систем** је креирао нову партију”. (ИА)



СК 6: Случај коришћења – Извођење потеза корисника

Назив СК:
Извођење потеза корисника

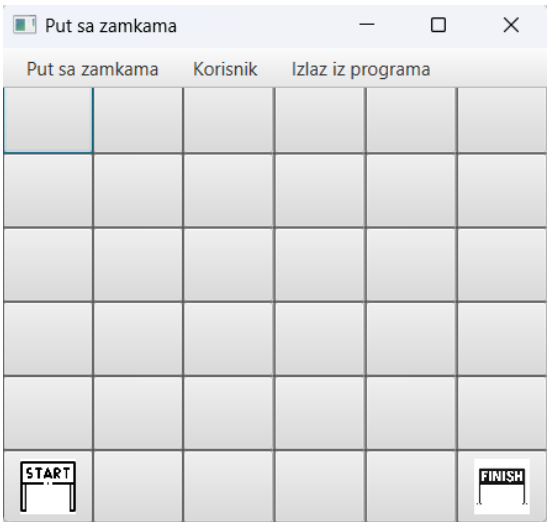
Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и **корисник** је улогован на систем. Приказан је кориснички интерфејс за игру "Пут са замкама".

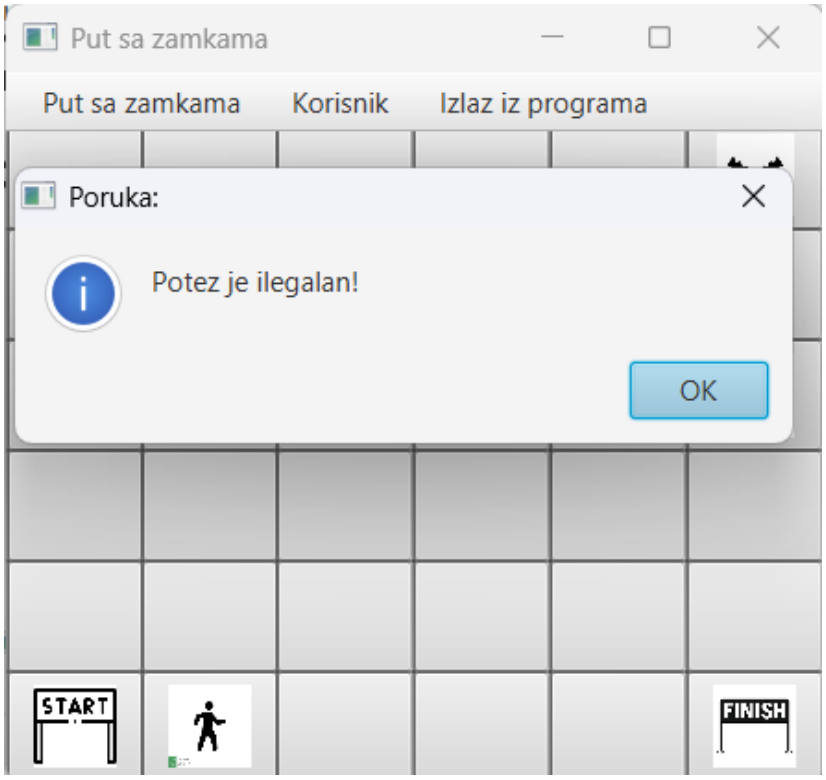
Основни сценарио СК:

1. **Корисник** бира потез који жели да изведе. (АПУСО)
2. **Корисник** позива **систем** да изведе потез. (АПСО)
3. **Систем** проверава да ли је изабрано поље валидан потез. (СО)
4. **Систем** изводи потез и ажурира стање игре. (СО)
5. **Систем** приказује **кориснику** ажурирано стање игре и обавештава да је ред на систем. (ИА)



Алтернативни сценарио:

5.1 Уколико **систем** утврди да корисников потез није исправан, он приказује **кориснику** поруку: "Потез је илегалан". (ИА)



СК 8: Случај коришћења – Додавање резултата игре

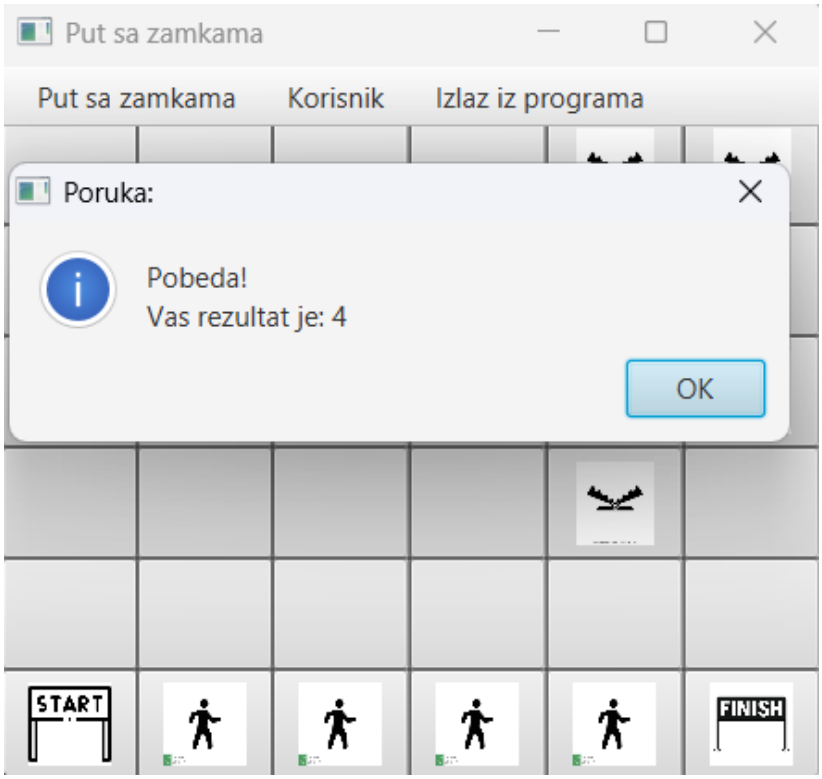
Назив СК:
Додавање резултата игре

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен и игра је завршена.

Основни сценарио СК:
1. **Корисник** завршава игру одигравањем последњег потеза. (АПУСО)
2. **Корисник** позива **систем** да дода резултате игре. (АПСО)
3. **Систем** приказује **кориснику** резултат игре. (ИА)



СК 9: Случај коришћења – Излазак из апликације

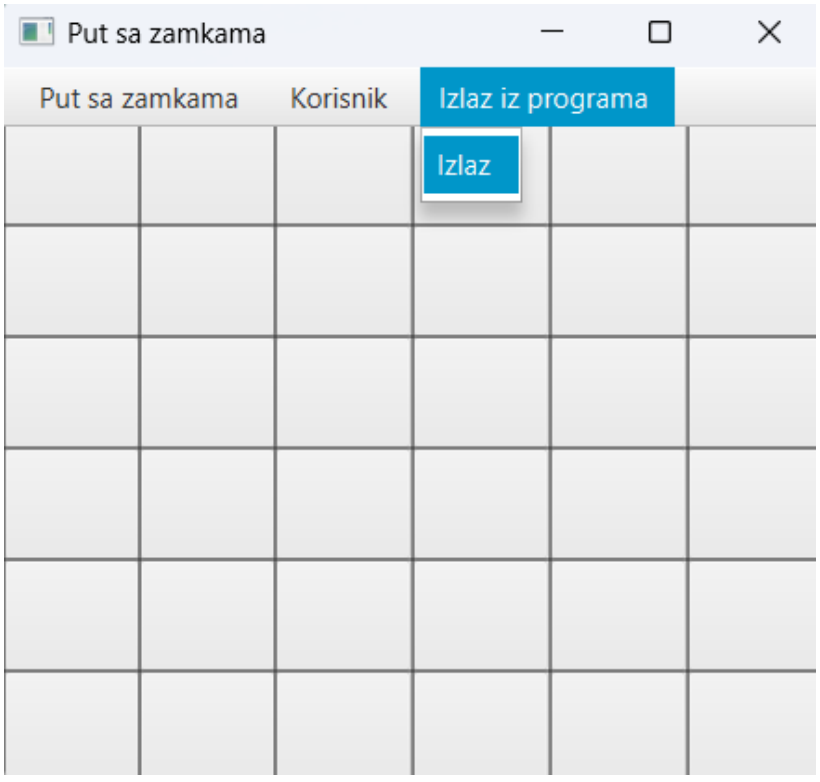
Назив СК:
Излазак из апликације

Актори СК:
Корисник

Учесници СК:
Корисник и **систем**

Предуслов:
Систем је укључен.

Основни сценарио СК:
1. **Корисник** позива **систем** да изађе из апликације. (АПСО)
2. **Систем** затвара апликацију и прекида своје извршавање. (СО)



3.2 Комуникација са клијентом

Два начина комуникације са клијентом су:

- Преко Web сервиса
- Преко socket-a

Kod Web сервиса, комуникација се врши коришћењем SOAP транспортног протокола. SOAP протокол је заснован на http захтевима где се поруке преводе у XML формат преко којих клијент и сервер међусобно комуницирају.

На серверској страни најпре се креира серверски socket који ослушкује мрежу. Када је он активан, клијенти покретањем програма успостављају везу са сервером. Након тога, сервер покреће посебну нит помоћу које се остварује двосмерна комуникација између клијента и сервера.

3.2.1. Комуникација преко Web сервиса

Комуникација преко web сервиса се извршава преко SOAP протокола у оквиру кога се шаљу HTTP zahtevi, а објекат који прослеђујемо се преводи у XML поруку.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "generickiTransferObjekat", propOrder = {
    "currentRecord",
    "gdo",
    "poruka",
    "signal"
})
@XmlSeeAlso({
    TransferObjekatKorisnik.class,
    TransferObjekatIgra.class
})
public abstract class GenerickiTransferObjekat {

    protected int currentRecord;
    protected GeneralDObject gdo;
    protected String poruka;
    protected boolean signal;

    /**
     * Gets the value of the currentRecord property.
     *
     */
    public int getCurrentRecord() {
        return currentRecord;
```

```
}

/**
 * Sets the value of the currentRecord property.
 *
 */
public void setCurrentRecord(int value) {
    this.currentRecord = value;
}

/**
 * Gets the value of the gdo property.
 *
 * @return
 *     possible object is
 *     {@link GeneralDObject }
 *
 */
public GeneralDObject getGdo() {
    return gdo;
}

/**
 * Sets the value of the gdo property.
 *
 * @param value
 *     allowed object is
 *     {@link GeneralDObject }
 *
 */
public void setGdo(GeneralDObject value) {
    this.gdo = value;
}

/**
 * Gets the value of the poruka property.
 *
 * @return
 *     possible object is
 *     {@link String }
 *
 */
public String getPoruka() {
    return poruka;
```

```
    }

    /**
     * Sets the value of the poruka property.
     *
     * @param value
     *     allowed object is
     *     {@link String }
     *
     */
    public void setPoruka(String value) {
        this.poruka = value;
    }

    /**
     * Gets the value of the signal property.
     *
     */
    public boolean isSignal() {
        return signal;
    }

    /**
     * Sets the value of the signal property.
     *
     */
    public void setSignal(boolean value) {
        this.signal = value;
    }

}
```

На страни клијента у пакету Server_client се налази трансфер објекат, тј. објекат намењен размени података између клијента и сервера. Након тога се учитава операција коју је корисник покренуо, а преко GenerickiKontrolerServer се позива одговарајућа системска операција из пакета SO. Та системска операција изводи одређену функционалност. Ако је потребна комуникација са базом података, системска операција користи brokerBazaPodataka који спроводи конкретну операцију над базом. Када се операција успешно изврши, брокер враћа сигнал системској операцији, а она даље генерише повратну поруку намењену кориснику. Том поруком корисник је обавештен о успешности извршене акције на серверу. На крају, порука се са сервера враћа клијенту истим путем којим је и стигла до сервера, помоћу трансфер објекта и SOAP протокола.

3.2.2 Комуникација преко socket-а

Комуникација преко socket-а захтева ослушкивач на страни сервера, који ће ослушкивати мрежу константно. Ово омогућава да се корисник у било ком тренутну повеже на сервер, повезујући се на серверски сокет.

```
public class ServerSoket {

    static ServerSocket ss;
    static int brojKlijenata;

    public static void main(String[] args) throws Exception {
        ss = new ServerSocket(8189);
        while (true) {
            Socket soketS = ss.accept();
            Klijent kl = new Klijent(soketS, ++brojKlijenata);
        }
    }


}

public class Klijent extends Thread{

    private final Socket soketS;
    ObjectOutputStream out;
    ObjectInputStream in;

    public Klijent(Socket soketS1, int brojKlijenta) {
        soketS = soketS1;
        start();
    }

    @Override
    public void run() {

        try {
            out = new ObjectOutputStream(soketS.getOutputStream());
            in = new ObjectInputStream(soketS.getInputStream());

            while (true) {
                Object receivedObject = in.readObject();

                Object returnObject = executeOperation(receivedObject);
```

```
            if (returnObject != null) {
                sendObject(returnObject);
            }
        }
    } catch (IOException ex) {
        Logger.getLogger(Klijent.class.getName()).log(Level.SEVERE, null, ex);
    } catch (ClassNotFoundException ex) {
        Logger.getLogger(Klijent.class.getName()).log(Level.SEVERE, null, ex);
    }

}

private Object executeOperation(Object receivedObject) {
    if (receivedObject instanceof TransferObjekatKorisnikLogin) {

        TransferObjekatKorisnikLogin tokl = (TransferObjekatKorisnikLogin) receivedObject;
        new KPrijaviDK().prijaviDK(tokl);
        return tokl;
    }
    return null;
}

private void sendObject(Object object) throws IOException {
    out.writeObject(object);
    out.flush();
    out.reset();
}


}
```

Из датог кода, уочава се да одмах при покретању серверског дела се подиже серверски socket на порту 8189, и да се одмах након тога позива бесконачна петља у којој серверски socket непрестано ослушкује мрежу и прихвата захтеве клијената. Клијент је представљен као thread, односно нит и видимо да по његовом покретању се успостављају out и in објекти који представљају излазни и улазни ток података. Објекат који се шаље кроз мрежу јесте TransferObjekatKorisnikLogin, на основу чега се извршава пријава корисника. Након извршења операције, клијенту се прослеђује трансфер објекат назад, који садржи информације о успешности извршења операције и адекватним повратним информацијама.

3.3 Контролер апликационе логике

Након успостављања комуникације између клијента и сервера, потребно је обрадити захтеве. Контролер апликационе логике омогућава серверској страни да прихвати захтев преко socket-а или web сервиса (у зависности од тога која се операција шаље). Након што захтев дође до контролера, у зависности од операције која се налази у трансфер објекту, он позива одговарајућу системску операцију да се изврши. По завршетку системске операције, контролер је дужан да врати одговарајући одговор кориснику.

```
@WebService(serviceName = "GenerickiKontrolerServer")
public class GenerickiKontrolerServer {

   public  GenerickiTransferObjekat  kreirajDK(GenerickiTransferObjekat gto)
   {
     new KKreirajDK().kreirajDK(gto);

     return gto;
   }

    public GenerickiTransferObjekat izmeniDK(GenerickiTransferObjekat gto)
    { new KIzmeniDK().izmeniDK(gto);
      return gto;
    }


    public GenerickiTransferObjekat obrisiDK(GenerickiTransferObjekat gto)
    { new KObrisiDK().obrisiDK(gto);
      return gto;
    }

     public TransferObjekatKorisnikLogin prijaviDK(TransferObjekatKorisnikLogin tok)
    {
      KPrijaviDK kPrijaviDK = new KPrijaviDK();
      kPrijaviDK.prijaviDK(tok);
      return kPrijaviDK.getGto();
    }

     public TransferObjekatPartija izvediPotez(TransferObjekatPartija top)
    {
        new KIzvediPotez().izvediPotez(top);
        return top;

    }
```

```
    public  GenerickiTransferObjekat  nadjiDk(GenerickiTransferObjekat gto)
   {
     new KNadjiDK().nadjiDK(gto);
     return gto;
   }

    public TransferObjekatRangLista vratiRangListu(TransferObjekatRangLista torl){

        new KVratiRangListu().vratiRangListu(torl);
        return torl;

    }
}
```

Свака системска операција наслеђује апстрактну класу OpsteIzvrsenjeSO, која дефинише структуру сваке системске апликације.

```
public abstract class OpsteIzvrsenjeSO {
   static public BrokerBazePodataka bbp = new BrokerBazePodatakaImpl();
   int recordsNumber;
   int currentRecord = -1;
   GeneralDObject gdo;

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


   abstract public boolean izvrsiSO();

}
```

3.4 Пројектовање структуре софтверског система

На основу концептуалног(доменског) модела потребно је пројектовани структуру софтверског система. Свака класа има своје поља, одговарајуће get i set методе као и одређене поруке које се приказују у случају различитих операција. У наставку је приказана доменска класа Korisnik:

```
public class Korisnik  extends GeneralDObject implements Serializable{
    public int idKorisnik;
    String korisnickoIme;
    String sifra;
    String ime;
    String prezime;
    public Date datumRodjenja;
    Pol pol;

    public Korisnik(int idKorisnik, String korisnickoIme, String sifra, String ime, String prezime, Date
datumRodjenja, Pol pol) {
        this.idKorisnik = idKorisnik;
        this.korisnickoIme = korisnickoIme;
        this.sifra = sifra;
        this.ime = ime;
        this.prezime = prezime;
        this.datumRodjenja = datumRodjenja;
        this.pol = pol;
    }



    public Korisnik()
      { this.idKorisnik = 0;
        this.korisnickoIme = "";
        this.sifra = "";
        this.ime = "";
        this.prezime = "";
        SimpleDateFormat sm = new SimpleDateFormat("yyyy-MM-dd");
        Date dDatum = new Date();
        datumRodjenja = java.sql.Date.valueOf(sm.format(dDatum));
      }


    public int getIDKorisnik(){

        return this.idKorisnik;
```

```
}

public void setIdKorisnik(int idKorisnik) {
    this.idKorisnik = idKorisnik;
}

public Pol getPol() {
    return pol;
}

public void setPol(Pol pol) {
    this.pol = pol;
}


public Korisnik(int idKorisnik)
  { this.idKorisnik = idKorisnik;
  }

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


public int getPrimaryKey()
  {return this.idKorisnik;}

public String getKorisnickoIme()
  {return this.korisnickoIme;}

public String getSifra()
  {return this.sifra;}

public String getIme()
  {return this.ime;}

public String getPrezime()
  {return this.prezime;}

public Date getDatumRodjenja()
  {return this.datumRodjenja;}

public void setIDKorisnika(int idKorisnik)
```

```
    {this.idKorisnik = idKorisnik;}

   public void setKorisnickoIme(String korisnickoIme)
    {this.korisnickoIme = korisnickoIme;}

   public void setSifra(String sifra)
    {this.sifra = sifra; }

   public void setIme(String ime)
    {this.ime = ime;}

   public void setPrezime(String prezime)
    {this.prezime = prezime;}

   public void setDatumRodjenja(java.sql.Date datumRodjenja)
    {this.datumRodjenja = datumRodjenja;}

    public java.util.Date getDatumRodjenja1(java.util.Date datumRodjenja)
    {SimpleDateFormat sm = new SimpleDateFormat("yyyy-MM-dd");
     this.datumRodjenja = java.sql.Date.valueOf(sm.format(datumRodjenja));
     return this.datumRodjenja;
    }


   @Override
    public String getAtrValue() { return  idKorisnik + ", '" + korisnickoIme + "', '" + sifra + "', '" + ime
+ "', '" + prezime + "', '" + getDatumRodjenja1(datumRodjenja) + "', '" + pol + "'"; }

   @Override
   public String setAtrValue() {
       return "idKorisnik=" + idKorisnik + ", " + "korisnickoIme='" + korisnickoIme + "', " + "sifra='" +
sifra + "', ime='" + ime + "', prezime='" + prezime + "', datumRodjenja='" +
getDatumRodjenja1(datumRodjenja) + "', " + "pol='" + pol + "'";
   }

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

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

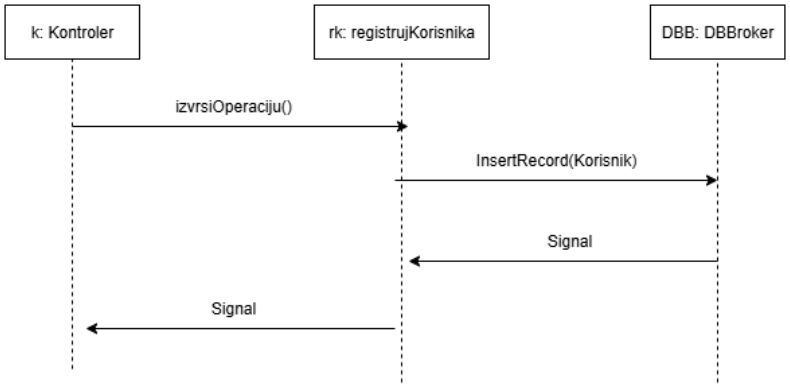
```
    @Override
    public String getNameByColumn(int column) {
        String names[] =
{"idKorisnik","korisnickoIme","sifra","ime","prezime","datumRodjenja","pol"};
        return names[column];
    }

    @Override
    public GeneralDObject getNewRecord(ResultSet rs) throws SQLException {
      return new
Korisnik(rs.getInt("idKorisnik"),rs.getString("korisnickoIme"),rs.getString("sifra"),rs.getString("ime
"),rs.getString("prezime"),(rs.getDate("datumRodjenja")), Pol.valueOf(rs.getString("pol")));
    }

    @Override
    public String poruka1() {
        return "Problem oko brojaca korisnika.";
    }

    @Override
    public String poruka2() {
        return "Ne moze da se poveca brojac korisnika.";
    }

    @Override
    public String poruka3() {
        return "Korisnik je registrovan.";
    }

    @Override
    public String poruka4() {
        return "Sistem ne može da registruje novog korisnika.";
    }

    @Override
    public String poruka5() {
        return "Sistem je obrisao korisnika.";
    }

    @Override
    public String poruka6() {
        return "Sistem ne može da obriše korisnika.";
    }
```

```
    @Override
    public String poruka7() {
        return "Ne moze se obrisati korisnik jer ne postoji.";
    }

    @Override
    public String poruka8() {
        return "Korisnik je izmenjen.";
    }

    @Override
    public String poruka9() {
        return "Sistem ne može da izmeni korisnika.";
    }

    @Override
    public String poruka10() {
        return "Ne moze se promeniti korisnik jer ne postoji.";
    }

}
```

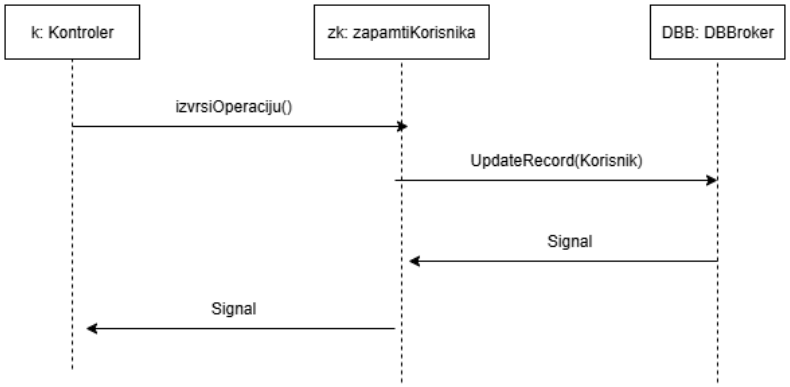
3.5 Пројектовање системских операција

За сваку системску операцију кључно је пројектовати концептуално решење које је директно повезано са логиком проблема и сходно томе се за сваки уговор пројектује концептуално решење.

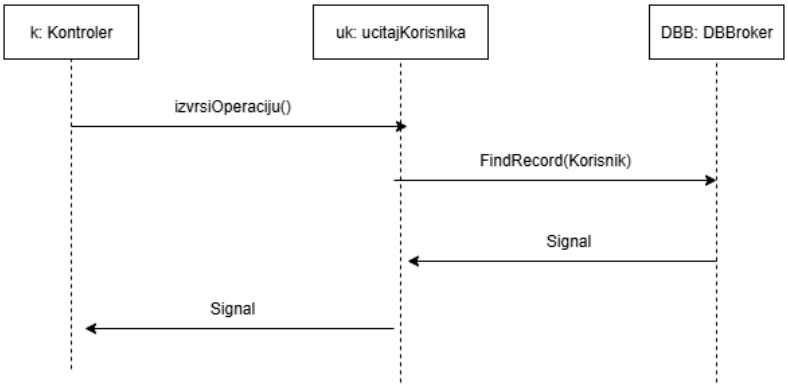
1. Уговор УГ1: registrujKorisnika(): Signal;
   Веза са СК: СК1
   Предуслови: /
   Постуслови: Корисник је регистрован



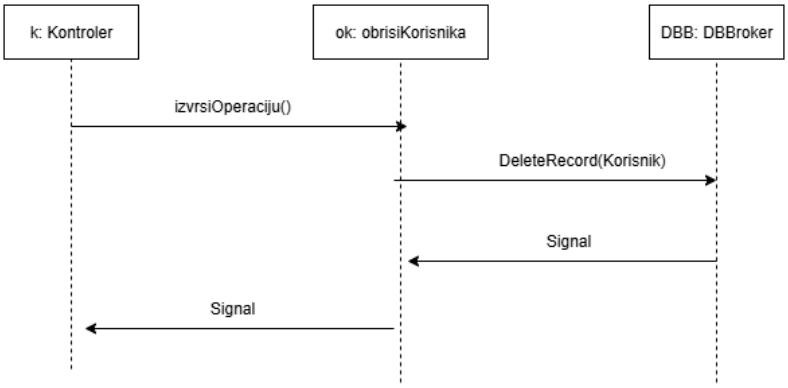
2. Уговор УГ2: zapamtiKorisnika(Korisnik): Signal;
   Веза са СК: СК2
   Предуслови: Вредносна и структурна ограничења над објектом Korisnik морају бити задовољена
   Постуслови: Корисник је запамћен



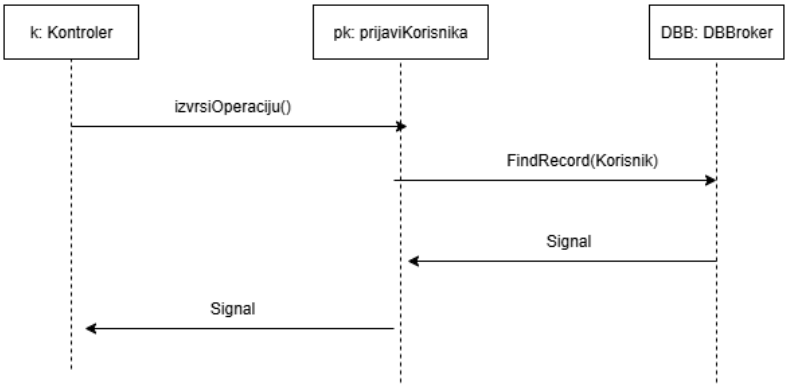
3. Уговор УГ3: ucitajKorisnika(): Signal;
   Веза са СК: СК2, СК3
   Предуслови: Корисник је претходно креиран
   Постуслови: /



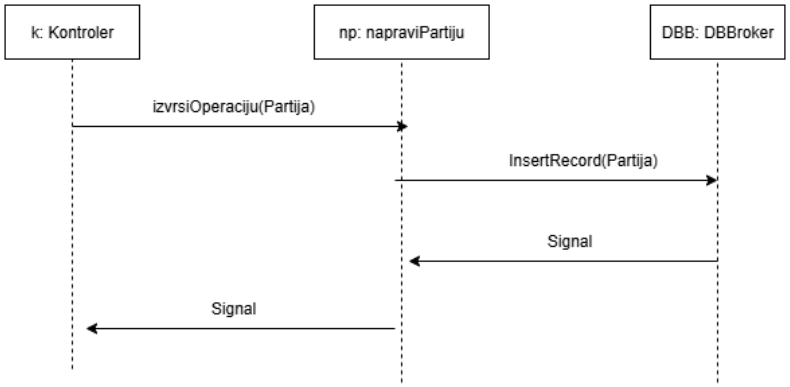
4. Уговор УГ4: obrisiKorisnika(Korisnik): Signal;
   Веза са СК: СК3
   Предуслови: Структурна ограничења над објектом Korisnik морају бити задовољена
   Постуслови: Корисник је обрисан



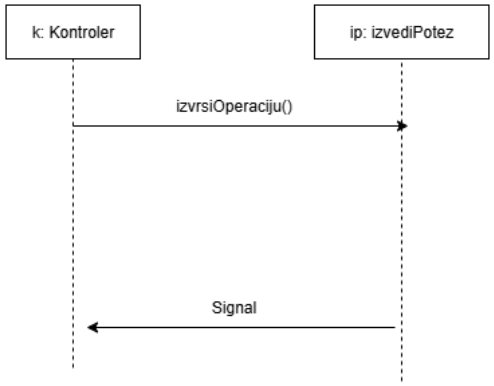
5. Уговор УГ5: prijavaKorisnika(Korisnik): Signal;
   Веза са СК: СК4
   Предуслови: /
   Постуслови: Корисник је пријављен на систем



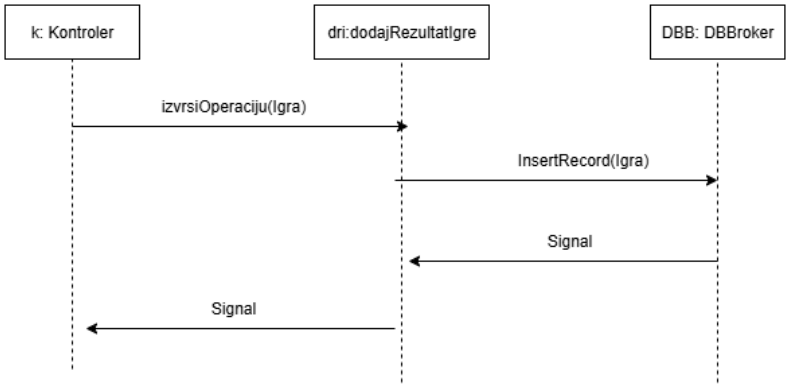
6. Уговор УГ6: kreirajNovuPartiju(): Signal;
   Веза са СК: СК5
   Предуслови: Корисник је претходно пријављен на систем
   Постуслови: Нова партија је креирана



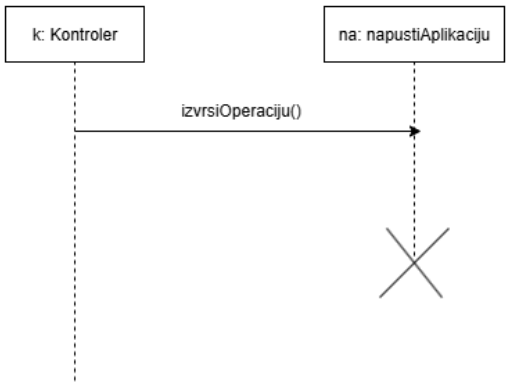
7. Уговор УГ7: izvediPotez(): Signal;
   Веза са СК: СК6
   Предуслови: Корисник је претходно пријављен на систем и нова партија је креирана
   Постуслови: /



8. Уговор УГ8: dodajRezultatIgre(Integer): Signal;
   Веза са СК: СК8
   Предуслови: Корисник је претходно пријављен на систем, нова партија је креирана и корисник је одиграо последњи потез
   Постуслови: Подаци о резултату игре су запамћени



9. Уговор УГ9: napustiAplikaciju(): Signal;
   Веза са СК: СК9
   Предуслови: /
   Постуслови: Корисник је одјављен из апликације и корисничка апликација се затвара



3.6 Брокер базе података

Брокер базе података представља међуслој између апликације и саме базе података, обезбеђујући перзистентни сервис свим објектима доменских класа који се чувају у бази. На тај начин, он централизује и контролише све операције приступа и манипулације подацима, спречавајући вишеструко или неконзистентно мењање базе у истом тренутку.

```
public abstract class BrokerBazePodataka
{   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> findRecord(GeneralDObject odo,String where);
    public abstract boolean commitTransation();
    public abstract boolean rollbackTransation();
    public abstract boolean getCounter(GeneralDObject odo,AtomicInteger counter);
    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);
    public abstract int findRecordPosition(GeneralDObject odo);
    public abstract Connection getConnection();
    public abstract void close(Connection conn, Statement st, ResultSet rs);
}
```

Брокер базе података је апстрактна класа, што подразумева да је омогућено позивање његових метода без увида у њихове конкретне имплементације (које се налазе у класи *BrokerBazePodatakaImpl*). Поред тога, методе прихватају и враћају генеричке објекте. На овај начин се омогућава да све доменске класе користе исти брокер базе података. Овим приступом значајно се умањује количина кода која би постојала уколико би се за сваку доменску класу засебно програмирао брокер базе података.

Да би овај приступ имао смисла, потребно је да све доменске класе наследе апстрактну класу GeneralDObject:

```
public abstract class GeneralDObject implements Serializable {

    abstract public String getAtrValue();//{return "";}

    abstract public String setAtrValue();//{return "";}
```

```
    abstract public String getClassName();//{return "";}

    abstract public String getWhereCondition();//{return "";}

    abstract public String getNameByColumn(int column);//{return "";}

    abstract public GeneralDObject getNewRecord(ResultSet rs) throws SQLException; //{return
null;}

    abstract public int getPrimaryKey();

    abstract public void setID(int id);

    abstract public String poruka1();

    abstract public String poruka2();

    abstract public String poruka3();

    abstract public String poruka4();

    abstract public String poruka5();

    abstract public String poruka6();

    abstract public String poruka7();

    abstract public String poruka8();

    abstract public String poruka9();

    abstract public String poruka10();
}
```

3.7 Пројектовање складишта података

На основу релационог модела структуре софтверског система, коришћењем MySQL-a, пројектоване су табеле релационог система за управљање базом података.

Tabela Korisnik		
Ime polja	Tip	Veličina
idKorisnik	int	10
korisnickoIme	varchar	32
sifra	varchar	32
ime	varchar	32
prezime	varchar	32
datumRodjenja	date	/
pol	enum	’MUŠKI’, ’ŽENSKI’

Tabela Igra		
Ime polja	Tip	Veličina
id	int	11
rezultat	int	11
datum	Date	/
korisnikId	Int	10

3.8 Принципи, методе и стратегије пројектовања софтвера

3.8.1 Принципи пројектовања софтвера

Апстракција

Апстракција је процес који подразумева издвајање најважнијих карактеристика везаних за неку конкретну појаву.

У контексту пројектовања апстракције постоје два кључна механизма:

- Спецификација
- Параметризација

Спецификација

Спецификација је механизам апстракције који подразумева издвајање општи својстава из неког скупа елемената. Та својства се могу представити преко процедуре, података или контроле.

Процедурална апстракција

Процедурална апстракција помаже да издвојимо општа својства из неког скупа процедура:

- Тип онога што враћа процедура
- Назив процедуре
- Улазни аргументи процедуре

Општи случај процедуралне апстракције према упрошћеној Ларамановој методологији садржи:

- Потпис процедуре
- Веза са случајевима коришћења
- Предуслови и Постуслови

Као пример процедуралне апстакције биће коришћен пример једне системске операције:

> Операција: obrisiKorisnika(Korisnik): Signal;
> Веза са СК: СК3
> Предуслови: Структурна ограничења над објектом Korisnik морају бити задовољена
> Постуслови: Корисник је обрисан

Апстракција података

Апстракцијом података из скупа се издвајају његова општа својства. Спецификација скупа врши се навођењем општих елемената (ID, корисничко име, име, презиме). Апстракција података је одређена именом скупа – *Korisnik* и његовом спецификацијом. Када елементи поседују и структуру и понашање, примењује се како апстракција података тако и процедурална апстракција. Крајњи резултат оваквог приступа је класа која обједињује атрибуте и процедуре.

Апстракција контролом

Постоје два облика апстракције контролом:

- Апстракција наредби
- Апстракција структура података

Апстракцијом наредби се издвајају заједничка својства из скупа наредби и она се представљају кроз контролне структуре и опште наредбе. Овај начин омогућава се да се различите наредбе посматрају и користе на јединствен начин, без потребе за појединачним специфичним имплементацијама.

Апстракцијом структура података издвајају се општа својства различитих структура, а она се затим представљају преко итератора. Итератор у општем смислу служи да контролише кретање кроз елементе структуре података.

Параметризација

Параметризација је механизам апстракције којим се из скупа елемената издвајају њихова општа својства и та својства се изражавају помоћу параметара. Параметризација се може посматрати кроз пет случајева:

- Параметризација скупа елемената простог типа
- Параметризација скупа елемената сложеног типа
- Параметризација скупа операција
- Параметризација скупа процедура
- Параметризација скупа наредби

Параметризација скупа елемената простог типа

Под претпоставком да имамо скуп целих бројева -1, 0 , 1, 2, ... , цео скуп можемо представити помоћу једног општег представника целих бројева. У програмерском смислу, тај представник свих карактера може се описати параметром:

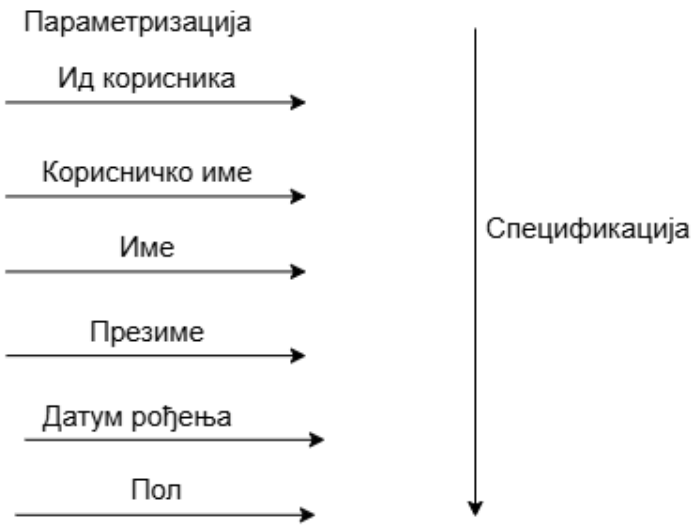
long a;

Овим изразом декларишемо променљиву *a* која је целобројног типа. На тај начин, самим чином декларисања, имплицитно дефинишемо и скуп операција које је могуће извршити над овом променљивом.

Параметризација скупа елемената сложеног типа

У оквиру овог софтверског система постоји следећи скуп података: (1, марко, Марко, Марковић, 2000-03-04, Мушки), (2, петар12, Петар, Петровић, 1999-08-20, Мушки) и (3, николина01, Николина, Јовановић, 2001-07-07, Женски). Овај скуп корисника можемо представити са к1,к2,к3. Након тога се може извршити параметризација за сваки индивидуални скуп: скуп ид корисника, скуп корисничких имена, скуп имена, скуп презимена, скуп датума рођења и скуп полова.

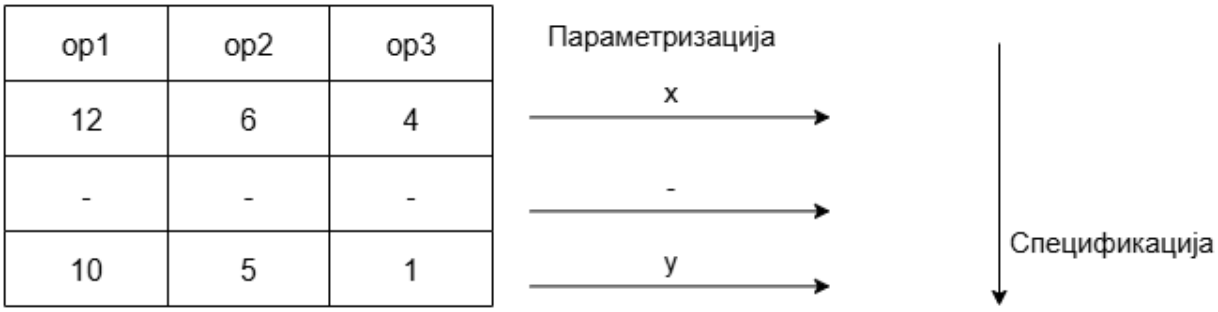
k1	k2	k3
1	2	3
марко	петар12	николина01
Марко	Петар	Николина
Марковић	Петровић	Јовановић
2000-03-04	1999-08-20	2001-07-07
Мушки	Мушки	Женски



Параметризацијом се издвајају општа својства елемената једног скупа. Спецификација скупа представља управо опис тих општих својстава. Апстракција података се дефинише именом и спецификацијом скупа, на пример: Корисник (ид, корисничко име, име, презиме, датум рођења, пол). Из овога следи да спецификација произилази из параметризације, јер се заснива на њеним резултатима.

Параметризација скупа операција

Параметризација се врши и над скупом операција. При датом скупу операција ((12-10), (6-5), (4-1)), параметризацијом добијамо општи скуп операција x-y. Елементи над којима се врши операција су x и y, као и операнд који показује шта операција ради, у овом случају -.



Резултат параметризације је x-y који представља општу операцију за одузимање два цела броја.

Параметризација скупа процедура

Параметризацијом се добија општа процедура чији елементи укључују: повратну вредност процедуре, назив процедуре, аргументе (параметре) и тело процедуре. Резултат параметризације је потпис процедуре, који представља један од кључних делова спецификације процедуре (процедурална апстракција).

Параметризација скупа наредби

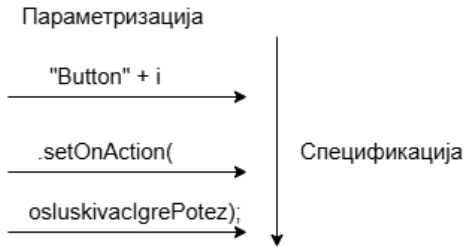
За следеће наредбе:

```
Button1.setOnAction(osluskivacIgrePotez);
Button2.setOnAction(osluskivacIgrePotez);
Button3.setOnAction(osluskivacIgrePotez);
```

које на дугме постављају ослушкивач(OsluskivacIgrePotez), параметризацијом се добија следећа општа наредба:

```
button.setOnAction(osluckivacIgrePotez);
```

Наредба 1	Наредба 2	Наредба 3
Button1	Button2	Button3
.setOnAction(	.setOnAction(	.setOnAction(
osluskivaclgrePotez);	osluskivaclgrePotez);	osluskivaclgrePotez);



На основу тога као резултат параметризације се добија:

```
for(Button button : buttons){
        button.setOnAction(osluckivaclgrePotez);
}
```

Спојеност и кохезија

При развоју објектно-оријентисаног софтверског система потребно је тежити ка постизању два важна циља:

- Висока кохезија(high cohesion)
- Слаба повезаност између класа(weak coupling)

Спојеност

Спојеност означава међусобну зависност класа, односно способност једне класе да изврши све своје операције без утицаја других класа. Ако класа A не може да изврши методу m1() без позива методе m2() која је дефинисана у класи B, онда је класа A зависна од класе B. У овом софтверском систему, код класа из пакета SO (системске операције), уочава се слаба повезаност, јер ове класе зависе само од класе BrokerBazePodataka.

Кохезија

Кохезија представља међусобну повезаност метода унутар класе. Ако класа A у себи има само једну сложену методу m1() у оквиру које се извршава цела логика, онда та класа има ниску кохезију. Ако ову сложену методу поделимо на више мањих метода m2(), m3(), m4() и учинимо да извршавање методе m1() изгледа на следећи начин:

```
m1(){
        m2();
        m3();
        m4();
}
```

На овај начин се постиже висок ниво кохезије. На следећем примеру у контролеру ранг листе постигнут је висок ниво кохезије у методи врати ранг листу:

```
public boolean vratiRangListu(){

    TransferObjekatRangLista torl = new TransferObjekatRangLista();
    torl = generickiKontrolerServer.vratiRangListu(torl);
    if(!torl.isSignal()){

       poruka(torl.getPoruka());
       return false;
    }

    KonverterGUIDK.popuniTabelu(torl.getRangLista(), fXMLDocumentController);
    poruka(torl.getPoruka());


    return true;
  }
```

Такође, с обзиром на то да су све операције за рад са ранг листом дефинисане у оквиру истог метода у класи контролера (нпр. vratiRangListu() у FXMLDocumentController), можемо утврдити да је ова класа уједно слабо повезана са другим класама . Разлог зашто желимо да постигнемо висок ниво кохезије и низак ниво спојености је тај што се на тај начин лакше управља класом, а код постаје одржив и једноставан за одржавање.

Декомпозиција и модуларизација

Декомпозиција је процес разлагања сложеног проблема на више мањих, међусобно независних проблема који се лакше могу решавати. На овај начин се поједностављује решавање почетног, комплексног проблема јер се сваки подпроблем може обрадити одвојено. У области развоја софтвера, декомпозиција се примењује тако што се систем дели на више функционалних модула, чиме настаје модуларизација. Декомпозицију је могуће спроводити у различитим фазама развоја, укључујући прикупљање захтева, пројектовање система и саму имплементацију, како би се осигурала боља организованост и лакше одржавање кода.

Декомпозицију је могуће посматрати кроз три аспекта:

- Декомпозиција код прикупљања захтева
- Декомпозиција код пројектовања софтвера
- Декомпозиција функција (метода)

Декомпозиција код прикупљања захтева

Кориснички захтеви обухватају сложене целине чије пројектовање захтева значајно време и пажњу, због чега је веома важно да се још од најранијих фаза развоја софтверског система примени декомпозиција. Конкретно, један кориснички захтев се разлаже на скуп мањих захтева који се могу приказати кроз случајеве коришћења. У нашем примеру, кориснички захтев се декомпонује на следеће захтеве:

1. Регистрација новог корисника
2. Измена корисника
3. Брисање корисника
4. Пријављивање на систем
5. Креирање нове партије
6. Извођење потеза корисника
7. Извођење потеза система
8. Додавање резултата игре
9. Излазак из апликације

Декомпозиција код пројектовања софтвера

Фаза пројектовања где се примењујући поједностављену Ларманову методу развоја софтвера, софтверски систем декомпонује на три основне компоненте: кориснички интерфејс, апликациону логику и складиште података. Због сложености апликационе логике, она се често декомпонује на контролер корисничког интерфејса, пословну логику и брокер базе података. Приликом декомпозиције важно је водити рачуна да се не пређе мера, односно да се задржи принцип јаке кохезије и слабе спојености. Поред тога, потребно је обезбедити и енкапсулацију, односно сакрити интерне информације сваког модула.

Декомпозиција функција (метода)

Декомпозиција сложене функције обухвата њено декомпоновање на више подфункција. Ове подфункције се затим независно реализују, а на крају се интегришу у једну целину како би се постигло понашање почетне функције. На овај начин испуњавамо принципе јаке кохезије и слабе спојености, избегавајући сложене методе које самостално извршавају целокупно понашање класе и добијајући број мањих метода које независно решавају делове проблема.

Учаурење/Сакривање информација

Учаурење је процес у којем се разликују особине модула (нпр. класе) које су доступне другим модулима од особина које су скривене од других модула система. Јавне особине модула указују на суштинске карактеристике модула, док скривене особине приказују његове интерне детаље. Јавне особине могу се користити у другим модулима, док скривене особине нису доступне другим модулима. Ово сакривање информација је последица процеса учаурења, којим се интерне особине модула чувају ван домашаја корисника модула.

Одвајање интерфејса од имплементације

Интерфејс се одваја од имплементације и представља кориснику само доступне операције, без откривања начина њихове реализације. У нашем примеру, корисник је genericki Kontroler Server, док апстрактне класе BrokerBazePodataka и GeneralDObject представљају интерфејс који излаже методе кориснику.

3.8.2 Стратегије пројектовања софтвера

Divide and Conquer(Подели и освоји)

Стратегија подели и освоји, заснива се на принципу декомпозиције, при којој се сложени почетни проблем дели на више независних подпроблема. Ови подпроблеми се затим самостално решавају, што значајно олакшава решавање целокупног, комплексног проблема.

Примена ове стратегије у развоју софтвера може се уочити у различитим фазама:

- Прикупљање захтева – кориснички захтеви, који су често сложени, декомпонују се на једноставније захтеве и описују се кроз случајеве коришћења.
- Анализа – понашање система се представља кроз скуп независних системских операција, а структура се описује концептима који чине концептуални модел.
- Пројектовање – архитектура система се дели на три основне компоненте: кориснички интерфејс, апликациону логику и складиште података. Даље, кориснички интерфејс се дели на екранске форме и контролер, док се апликациона логика дели на контролер апликационе логике, пословну логику и брокер базе података.

На овај начин се обезбеђује модуларност, лакше одржавање и боља организованост софтверског система.

Top down(S vrha na dole)

Овај приступ подразумева остваривање јаке кохезије у пројектовању софтвера. Основна идеја је да главна метода не садржи сву логику извршавања, већ да њена улога буде позивање једноставнијих метода које решавају појединачне подпроблеме. Поред тога, подметоде се могу даље декомпоновати на мање јединице све док се не дође до метода које решавају потпуно једноставне проблеме. Овај приступ се најбоље примењује унутар класе OpsteIzvrsenjeSO.

```
public abstract class OpsteIzvrsenjeSO {
   static public BrokerBazePodataka bbp = new BrokerBazePodatakaImpl();
   int recordsNumber;
   int currentRecord = -1;
   GeneralDObject gdo;

   synchronized public boolean opsteIzvrsenjeSO()
    {  bbp.makeConnection();
       boolean signal = izvrsiSO();
```

```
        if (signal==true)
          bbp.commitTransation();
        else
         bbp.rollbackTransation();
        bbp.closeConnection();
        return signal;
      }


    abstract public boolean izvrsiSO();

}
```

Bottom-up(S dna na gore)

Следећа стратегија за постизање принципа јаке кохезије у пројектовању софтвера је да се у оквиру једне функције идентификују логичке целине, које се затим издвајају као независне функције. На тај начин се врши декомпозиција функције, при чему се задржава иста функционалност. У наставку је приказан пример пре и после примене ове стратегије.

```
public abstract class OpsteIzvrsenjeSO {
    static public BrokerBazePodataka bbp = new BrokerBazePodataka1();
    int recordsNumber;
    int currentRecord = -1;
    GeneralDObject gdo;

    synchronized public boolean opsteIzvrsenjeSO() {
        try {
            bbp.makeConnection();

            boolean signal = izvrsiSO();
            if (signal == true) {
                bbp.commitTransation();
            } else {
                bbp.rollbackTransation();
            }

            bbp.closeConnection();

            return signal;
        } catch (Exception ex) {
```

```
            Logger.getLogger(OpsteIzvrsenjeSO.class.getName()).log(Level.SEVERE, null, ex);
            return false;
        }

    }

    abstract public boolean izvrsiSO();
}
```

После примене стратегије:

```
public abstract class OpsteIzvrsenjeSO {

    static public BrokerBazePodataka bbp = new BrokerBazePodataka1();
    int recordsNumber;
    int currentRecord = -1;
    GeneralDObject gdo;

    synchronized public boolean opsteIzvrsenjeSO() {
        try {
            uspostaviKonekciju();
            boolean signal = izvrsiSO();
            if (signal == true) {
                potvrdiTransakciju();
            } else {
                odustaniOdTransakcije();
            }
            prekiniKonekciju();
            return signal;
        } catch (Exception ex) {
            Logger.getLogger(OpsteIzvrsenjeSO.class.getName()).log(Level.SEVERE, null, ex);
            return false;
        }
    }

    abstract public boolean izvrsiSO();

    private void uspostaviKonekciju() throws Exception {
        bbp.makeConnection();
    }

    private void potvrdiTransakciju() throws Exception {
        bbp.commitTransation();
```

```
    }

    private void odustaniOdTransakcije() throws Exception {
        bbp.rollbackTransation();
    }

    private void prekiniKonekciju() throws Exception {
        bbp.closeConnection();
    }

}
```

Iterative incremental approach (Итеративно инкрементални приступ)

Приликом развоја софтвера коришћена је поједностављена Ларманова метода, заснована на итеративно-инкременталном приступу. То значи да је систем развијан кроз више итерација, при чему је у свакој итерацији реализована одређена надоградња пројекта, односно инкремент. У оквиру једне итерације најчешће се развија појединачна функционалност система. На пример, у једној итерацији развијала се системска операција за креирање корисника, у другој за измену корисника, итд. Итерације су обично независне, тако да се у већим тимовима различите операције могу налазити у различитим фазама развоја – једна операција се имплементира, друга је у фази анализе, а трећа у фази тестирања.

3.8.3 Методе пројектовања софтвера

Објектно оријентисано пројектовање (Object-oriented design)

На овом софтверском систему примењена је метода објектно-оријентисаног пројектовања. Основна карактеристика ове методе је заснивање на објектима који имају одређену структуру и понашање. Структура је приказана путем атрибута, док је понашање представљено методама класе.

Као резултат фазе анализе добија се логичка структура и понашање софтверског система. Логичка структура је приказана концептуалним моделом, а понашање система дијаграмима секвенци.

Принципи објектно оријентисаног пројектовања класа

Принципи који постоје код објектно оријентисаног пројектовања класу јесу:

- Принцип Отворено-Затворено
- Принцип замене Барбаре Лисков
- Принцип инверзије зависности
- Принцип уметања зависности
- Принцип издвајања интерфејса

Принцип Отворено-Затворено

Класа треба да буде отворен за проширење, али затворен за измене. То подразумева спречавање мењања понашања већ имплементираних и тестиранах метода. Овај принцип се најбоље остварује коришћењем апстрактних класа, конкретно у овом случају кроз класу ОпштаSO.

```
public abstract class OpsteIzvrsenjeSO {
   static public BrokerBazePodataka bbp = new BrokerBazePodatakaImpl();
   int recordsNumber;
   int currentRecord = -1;
   GeneralDObject gdo;

   synchronized public boolean opsteIzvrsenjeSO()
    {  bbp.makeConnection();
       boolean signal = izvrsiSO();
      if (signal==true)
```

```
            bbp.commitTransation();
        else
            bbp.rollbackTransation();
        bbp.closeConnection();
        return signal;
    }


    abstract public boolean izvrsiSO();

}
```

Принцип замене Барбаре Лисков

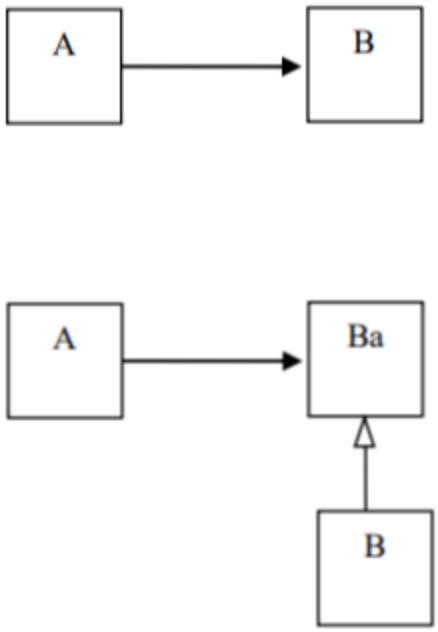
Подкласе морају бити међусобно заменљиве тако да њихова употреба не мења понашање програма. Овај принцип се најбоље илуструје на примеру општег доменског објекта и брокера базе података. Имплементација брокера је заснована на апстрактној класи GeneralDObject, па се у базу могу уписивати сви доменски објекти који наслеђују ову класу. На тај начин обезбеђује се да су све подкласе класе GeneralDObject у потпуности заменљиве, без икаквог утицаја на функционисање програма.

Принцип инверзије зависности

Модули вишег нивоа не треба директно да зависе од модула нижег нивоа – и једни и други треба да зависе од апстракција.

У овом раду постоји примеер брокера базе података који је модул вишег нивоа и доменских класа који су модули нижег нивоа. Суштина јесте да се избегава директна зависност између та два модула. Као решење брокер базе података комуницира само са GeneralDObject, којег наслеђују све доменске класе. На тај начин брокер базе података је индиректно повезан са сваком класом која наслеђује ову апстрактну класу.

У наставку биће приказани примери јаке и слабе зависности модула.



На приказаној слици, класа А представља брокера базе података који је у горњем делу дијаграма директно зависан од класе В, односно конкретног доменског објекта. У доњем делу слике приказано је решење овог проблема: увођењем апстрактне класе Ва уклања се директна зависност између брокера базе података и конкретног доменског објекта. На тај начин софтверски систем постаје одрживији и једноставнији за даљи развој.

Принцип уметања зависности

Принцип уметања зависности подразумева да се зависност између две компоненте програма успоставља у току извршавања преко треће компоненте. У приказаном у овом раду, улогу те треће компоненте има класа ОпштаСО, која омогућава повезивање брокера базе података са конкретном доменском класом.

Принцип издвајања интерфејса

Принцип издвајања интерфејса подразумева да интерфејс треба да садржи само мали број метода. Класе које га имплементирају морају реализовати све његове методе. Уколико постоји метода коју нека класа не жели или не треба да имплементира, та метода би требало да буде издвојена у посебан интерфејс, а не укључена у интерфејс који дата класа користи.

3.9 Примена патерна у пројектовању

Макро архитектура

Макроархитектура представља приказ структуре и организације софтверског система на најопштијем нивоу. Она се може описати применом различитих архитектонских стилова, који дефинишу основни начин организације и функционисања система.

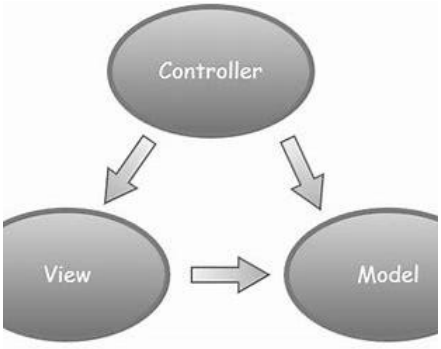
MVC патерн (Model-View-Controller)

MVC је патерн макро архитектуре који дели систем на три дела:

- Model – представља стање система које се може мењати уз помоћ неки операција модела

- View – приказује кориснику стање модела, односно обезбеђује кориснику интерфејс (екранску форму), које му омогућавају да уноси податке и позива одређене операције које се врше над моделом

- Controller – ослушкује и прихвата захтев од клијента за извршење операције. Након тога позива одговарајућу операцију модела и уколико модел промени своје стање обавештава view да је стање промењено

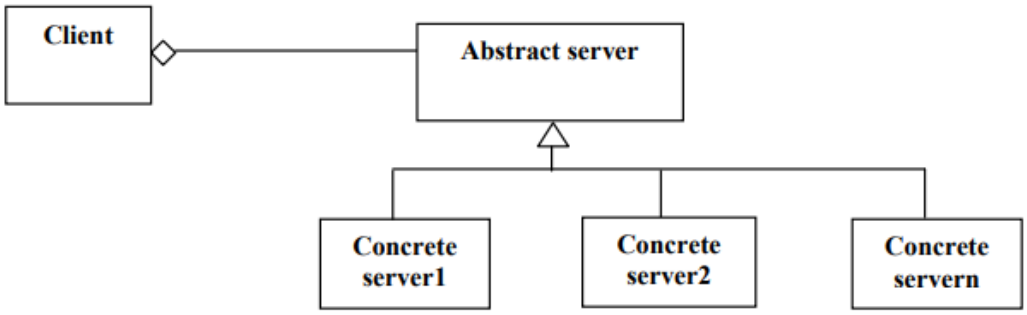


Мирко архитектура

У оквиру фазе пројектовања софтвера се користе три групе патерна: креациони, структурни и патерни понашања.
У најширем смислу, сваки patern може се посматрати као комбинација три целине:

- проблем који је потребно решити
- контекст који описује услове и ограничења у којима решење мора да функционише
- решење које одговара на тај проблем

Основна вредност патерна лежи у његовој могућности поновне примене. Захваљујући њиховој општости, исти патерн можемо применити у различитим ситуацијама, што олакшава адаптацију система новим корисничким захтевима и убрзава развој.
Књига Design Patterns коју су објавили „Gang of Four“ аутори описује укупно 23 пројектна патерна. Већина тих патерна има сличну структуру у основи, али сваки нуди специфичан начин за ефикасно решавање одређене врсте проблема у пројектовању.



Примењени патерни за решавање проблема пројектовања

Први захтев: На серверској страни потребно је обрадити све захтеве који стижу од клијента. Проблем настаје у томе што би, да би се могле извршити све функционалности система, клијент морао да зна све класе и потписе метода које сервер користи.

У овом раду, у класи Klijent на серверској страни система, приликом читања захтева клијента путем in.readObject(), примењује се Command патерн како би се позвала одговарајућа класа задужена за конкретну функционалност. Испод су приказане класе: Klijent, интерфејс Command и једна класа која имплементира тај интерфејс PrijaviKorisnikaCommand.

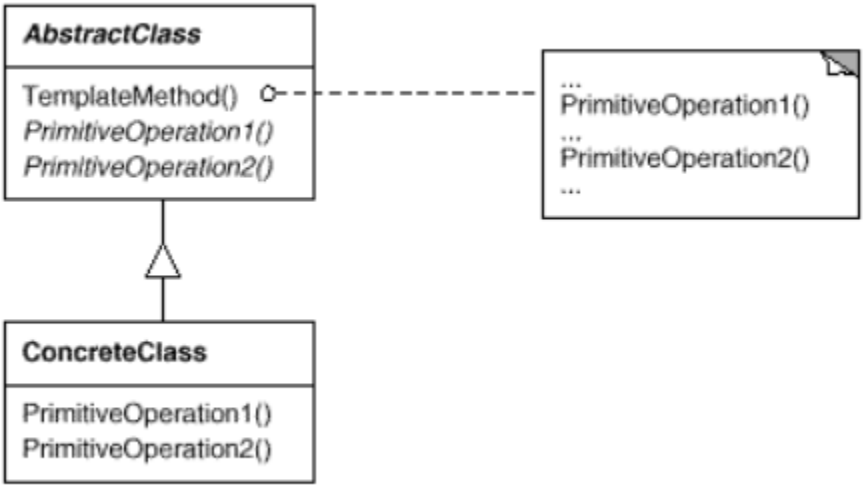
```
public class Klijent extends Thread{

    private final Socket soketS;
    ObjectOutputStream out;
    ObjectInputStream in;

    public Klijent(Socket soketS1, int brojKlijenta) {
        soketS = soketS1;
        start();
    }

    @Override
    public void run() {

        try {
            out = new ObjectOutputStream(soketS.getOutputStream());
            in = new ObjectInputStream(soketS.getInputStream());

            while (true) {
                Object receivedObject = in.readObject();

                Object returnObject = executeOperation(receivedObject);

                if (returnObject != null) {
                    sendObject(returnObject);
                }
            }
        } catch (IOException ex) {
            Logger.getLogger(Klijent.class.getName()).log(Level.SEVERE, null, ex);
        } catch (ClassNotFoundException ex) {
            Logger.getLogger(Klijent.class.getName()).log(Level.SEVERE, null, ex);
        }

    }

    private Object executeOperation(Object receivedObject) {

        return CommandRegistry.executeCommand(receivedObject);

    }

    private void sendObject(Object object) throws IOException {
        out.writeObject(object);
        out.flush();
```

```
        out.reset();
    }


}

public interface Command {

    public Object izvrsi(Object primljenObjekat);

}

public class PrijaviKorisnikaCommand implements Command{



    @Override
    public Object izvrsi(Object primljenObjekat) {

        TransferObjekatKorisnikLogin tokl = (TransferObjekatKorisnikLogin) primljenObjekat;

        KPrijaviDK kPrijaviDK = new KPrijaviDK();
        kPrijaviDK.prijaviDK(tokl);
        return primljenObjekat;


    }

}
```

Клијент шаље позива операцију и шаље одговарајући трансфер објекат серверу. Сервер на основу типа објекта који се шаље позива одговарајућу системску операцију.

Други захтев: На серверској страни потребно је обрадити захтеве за учитавање, чување, брисање и измену података који стижу од клијента. Свака од ових операција подразумева исти низ корака над базом података: отварање конекције, извршење тражене операције и, у случају успешног извршења, потврду трансакције; у супротном, трансакција се поништава, након чега се конекција затвара. Потенцијални проблем је понављање истог кода за сваки тип операције над базом.

За решење овог проблема коришћен је template патерн. Ово подразумева да сву све заједничке особине и операције за једну системску операцију смештени у апстрактну класу OpsteIzvrsenjeSO. Затим свака системска операција наслеђује ту апстрактну класу и према својим специфичностима имплементира методу izvrsiSO().



У наставку је приказана апстрактна класа OpsteIzvrsenjeSO као и системска операција ObrisiDK.

```
public abstract class OpsteIzvrsenjeSO {
   static public BrokerBazePodataka bbp = new BrokerBazePodatakaImpl();
   int recordsNumber;
   int currentRecord = -1;
   GeneralDObject gdo;

   synchronized public boolean opsteIzvrsenjeSO()
    {  bbp.makeConnection();
       boolean signal = izvrsiSO();
      if (signal==true)
        bbp.commitTransation();
      else
       bbp.rollbackTransation();
```

```
        bbp.closeConnection();
        return signal;
      }


    abstract public boolean izvrsiSO();

}



public class KObrisiDK extends OpsteIzvrsenjeSO {
    GenerickiTransferObjekat gto;

     public void obrisiDK(GenerickiTransferObjekat gto)
     { this.gto = gto;
       opsteIzvrsenjeSO();
     }

    @Override
    public boolean izvrsiSO()
      { gto.setSignal(false);
          GeneralDObject gdo1 = bbp.findRecord(gto.getGdo());
        if (gdo1!=null)
         {  if (bbp.deleteRecord(gto.getGdo()))
            { gto.setPoruka(gto.getGdo().poruka5());
              gto.setSignal(true);
            }
           else
            { gto.setPoruka(gto.getGdo().poruka6());
            }
         }
        else
         { gto.setPoruka(gto.getGdo().poruka7());
         }

       return gto.isSignal();
      }
}
```

4. Имплементација

Коришћени софтверски систем је развијен у програмском језику Јава (верзија 8), док је за израду корисничког интерфејса примењена технологија JavaFX. Као систем за управљање базама података коришћен је MySQL, док је као web сервис серверског дела апликације примењен WildFly web сервер. Поред тога, у имплементацији је искоришћен и програм Luca Generator (верзија 1.0), чији аутор је професор Синише Влајића, који је послужио за аутоматско креирање корисничког менија.

Коришћена је клијент-сервер архитектура софтверског система. У наставку су приказане класе клијентске стране:

- DomenskeKlase
  - Korisnik.java
  - KorisnikLogin.java
  - TrenutniKorisnik.java
- Enumi
  - Mod.java
- GUIIgra
  - FXMLDocument.fxml
  - FXMLDocumentController.java
  - Igra.java
  - KontrolerIgra.java
- GUIIgra.Osluskivaci
  - OsluskivacIgraPocetak.java
  - OsluskivacIgrePotez.java
- GUIKorisnik
  - FXMLDocument.fxml
  - FXMLDocumentController.java
  - Korisnik.java
  - KonverterGUIDK.java
  - KontrolerKorisnik.java
- GUIKorisnik.Osluskivaci
  - OsluskivacKorisnik.java
  - OsluskivacRegistracija.java
- GUIPrijavljivanje
  - FXMLDocument.fxml
  - FXMLDocumentController.java
  - Prijava.java
  - KonverterGUIDK.java
  - KontrolerPrijavljanje.java
- GUIPrijavljivanje.Osluskivaci
  - OsluskivacPrijavljivanje.java

- GUIRangLista
  - FXMLDocument.fxml
  - FXMLDocumentController.java
  - RangLista.java
  - KontrolerRangLista.java
- GUIRangLista.Osluskivaci
  - OsluskivacRangLista.java
- Server_client
  - GeneralDObject.java
  - GenerickiKontrolerServer.java
  - GenerickiKontrolerServer_Service.java
  - GenerickiTransferObjekat.java
  - Igra.java
  - IntArray.java
  - IzmeniDK.java
  - IzmeniDKResponse.java
  - IzvediPotez.java
  - IzvediPotezResponse.java
  - Korisnik.java
  - KorisnikLogin.java
  - KreirajDK.java
  - KreirajDKResponse.java
  - NadjiDK.java
  - NadjiDKResponse.java
  - ObjectFactory.java
  - ObrisiDK.java
  - ObrisiDKResponse.java
  - Partija.java
  - Pol.java
  - PrijaviDK.java
  - PrijaviDKResponse.java
  - TransferObjekatIgra.java
  - TransferObjekatKorisnik.java
  - TransferObjekatKorisnikLogin.java
  - TransferObjekatPartija.java
  - TransferObjekatRangLista.java
  - VratiRangListu.java
  - VratiRangListuResponse.java
  - package-info.java

- Slike
  - cikica.png
  - finish.png

    - sistem.png
    - start.png
- TransferObjekat
    - TransferObjekatKorisnikLogin.java

Класе на серверској страни:

- BrokerBazePodataka
    - BrokerBazePodataka.java
    - BrokerBazaPodatakaImple.java
- DomenskeKlase
    - GeneralDObject.java
    - Igra.java
    - Korisnik.java
    - KorisnikLogin.java
    - Partija.java
- LogikaIgre
    - Logika.java
- SO
    - KDodajRezutlatIgre.java
    - KIzmeniDK.java
    - KIzvediPotez.java
    - KKreirajDK.java
    - KNadjiDK.java
    - KObrisiDK.java
    - KPrijaviDK.java
    - KVratiRangListu.java
    - OpsteIzvrsenjeSO.java
- Server
    - GenerickiKontrolerServer.java
    - GenerickiKontrolerServerPort.java
    - Klijent.java
    - ServerSoket.java
- TransferObjekat
    - GenerickiTransferObjekat.java
    - TransferObjekatIgra.java
    - TransferObjekatKorisnik.java
    - TransferObjekatKorisnikLogin.java
    - TransferObjekatPartija.java
    - TransferObjekatRangLista.java
- Enumeracija
    - Pol.java

- Util
    - Command.java

    - CommandRegistry.java
- Util.commands
    - PrijaviKorisnikaCommand.java

4.1 Рефлексија

Рефлексија представља механизам који омогућава приступ основним информацијама о класи и њеним члановима. Такве информације називају се метаподаци, а у објектно оријентисаном програмирању они се чувају у оквиру мета-објеката.

Пример: Уклањање слика са свих поља

FXMLDocumentController класа:

```
public class FXMLDocumentController {

    @FXML
    public GridPane grid;

    @FXML
    public MenuBar menuBar;

    @FXML
    public Menu igra;

    @FXML
    public Menu korisnik;

    @FXML
    public Menu izlazIzPrograma;


    @FXML
    public MenuItem zapocniIgru;
    @FXML
    public MenuItem rangLista;
    @FXML
    public MenuItem nalogKorisnika;
    @FXML
    public MenuItem izlaz;

    @FXML
    public Button p00;
    @FXML
    public Button p10;
    @FXML
    public Button p20;
    @FXML
```

```
public Button p30;
@FXML
public Button p40;
@FXML
public Button p50;

@FXML
public Button p01;
@FXML
public Button p11;
@FXML
public Button p21;
@FXML
public Button p31;
@FXML
public Button p41;
@FXML
public Button p51;

@FXML
public Button p02;
@FXML
public Button p12;
@FXML
public Button p22;
@FXML
public Button p32;
@FXML
public Button p42;
@FXML
public Button p52;

@FXML
public Button p03;
@FXML
public Button p13;
@FXML
public Button p23;
@FXML
public Button p33;
@FXML
public Button p43;
@FXML
public Button p53;
```

```
@FXML
public Button p04;
@FXML
public Button p14;
@FXML
public Button p24;
@FXML
public Button p34;
@FXML
public Button p44;
@FXML
public Button p54;

@FXML
public Button p05;
@FXML
public Button p15;
@FXML
public Button p25;
@FXML
public Button p35;
@FXML
public Button p45;
@FXML
public Button p55;

@FXML
public void initialize(){

    KontrolerIgra kontrolerIgra = new KontrolerIgra(this);

}

public List<Button> vratiDugmice(){

    return Arrays.asList(
    p00, p01, p02, p03, p04, p05,
    p10, p11, p12, p13, p14, p15,
    p20, p21, p22, p23, p24, p25,
    p30, p31, p32, p33, p34, p35,
    p40, p41, p42, p43, p44, p45,
    p50, p51, p52, p53, p54, p55
);
```

```
    }
}
```

Метода која уклања слике из класе KontrolerIgra.java:

```
public void ocistiTablu(){

        Field[] polja = fXMLDocumentController.getClass().getDeclaredFields();

        for(Field f : polja){

            if(Button.class.isAssignableFrom(f.getType())){

                f.setAccessible(true);
            try {
                Button b = (Button) f.get(fXMLDocumentController);
                b.setGraphic(null);
                b.setDisable(true);
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }

            }

        }

    }
```

5. Тестирање

Сваки реализован случај коришћења је тестиран мануелно. Током тестирања проверено је како очекивано понашање система, тако и алтернативни токови извршавања, као и ситуације у којима су унети неправилни подаци. На основу спроведеног тестирања откривени су недостаци у систему који су накнадно отклоњени.

6. Закључак

Развој софтверског система спроведен је у складу са принципима објектно-оријентисаног пројектовања и уз примену одговарајућих принципа пројектовања као што су декомпозиција, модуларизација и употреба патерна пројектовања. Систем је реализован у више итерација, где је свака итерација допринела постепеној изградњи функционалности и побољшању квалитета решења. Посебан акценат стављен је на примену принципа јаке кохезије и слабе повезаности, чиме је обезбеђена већа одрживост и лакше одржавање кода.

Мануелним тестирањем верификовани су сви случајеви коришћења, укључујући и алтернативне токове и сценарије са неправилним уносима, што је допринело откривању и благовременом исправљању недостатака. Захваљујући примени оваквог приступа, добијен је систем који је стабилан, проширив и прилагођен будућем развоју.

7. Литература

Др Синиша Влајић, *Софтверски процес (скрипта)*, Београд, 2016.
Др Синиша Влајић, *Пројектовање софтвера (скрипта)*, Београд, 2020.