SCREW YOUR NEIGHBOR

Report for Milestone 3 of Sopra FS22, Group 36

Figure 1: cover (source B. Furrer)

Members

Carmen Kirchdorfer (20-720-132)
Salome Wildermuth (10-289-544)
Beat Furrer, group leader (07-542-392) Lucius Bachmann (11-060-274)
Moris Camporesi (19-764-349)

Diagrams

Database Schema

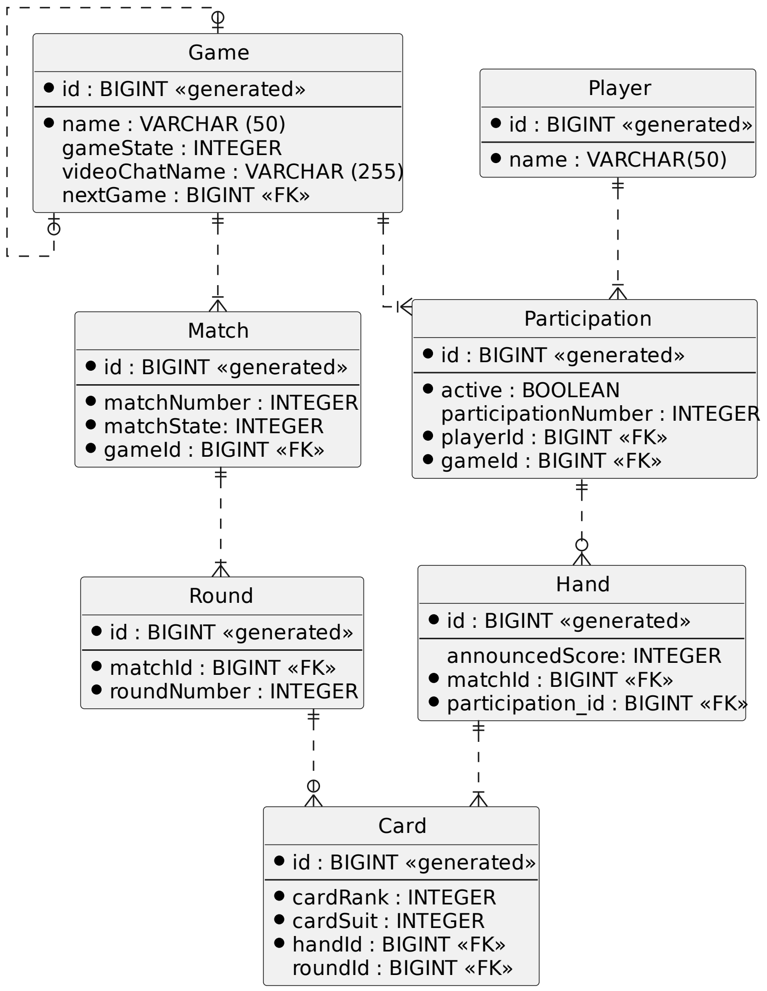


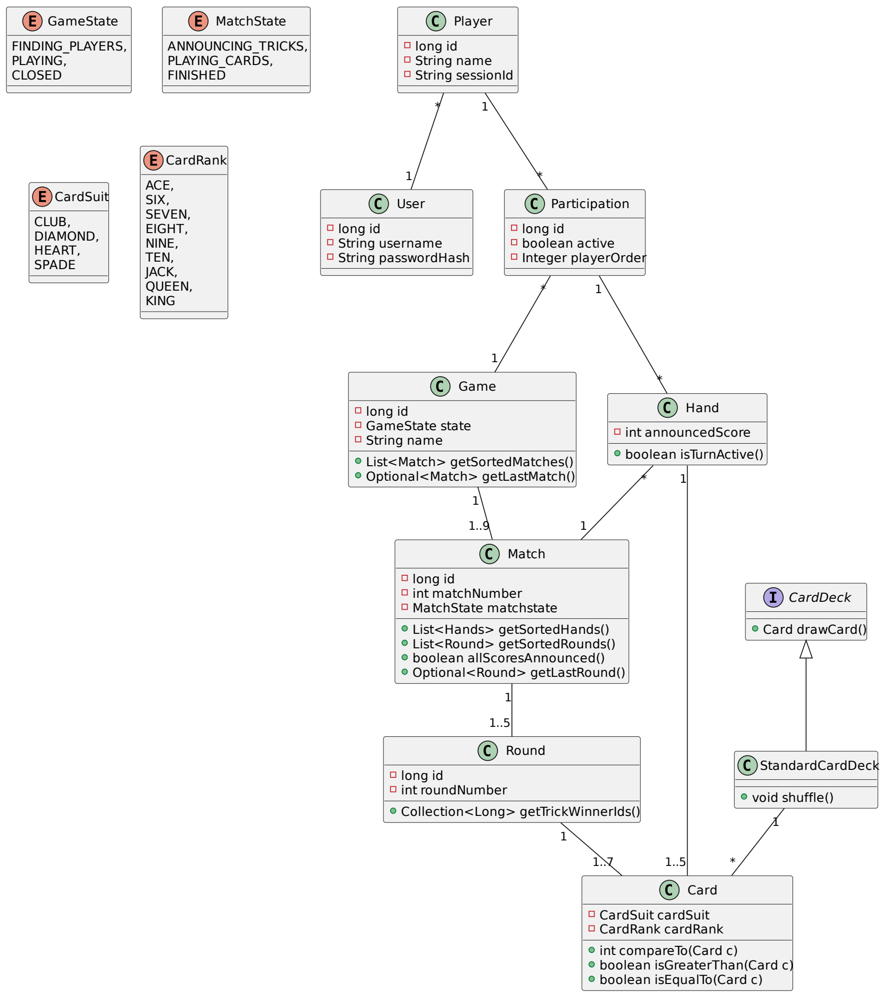
Figure 2: db_schema

Data Types

We were not sure how JPA stores Java data types in its tables. We found a mapping from basic Java data types to the respective standard SQL data types. We also found that enum values are stored by default by there ordinal i.e. with data type INTEGER. (Source)

Class Diagram

The class diagram that we handed in for M2 was quite sparse what was pointed out in the feedback. It has grown significantly (and also changed a bit) meanwhile the development process and because we use it permanently as a basis for our common understanding, we decided to hand in the current version again with the report for M3.



UI Screenshots

Tests

Complex unittest

Test class: HandTurnActiveTest, Code
Test method: the__first__player__must__play__a__card__when__round__starts()
Description: This test method verifies whether the first player in the playing order is the one who's turn is active while the other players turn is not active (assertion in block 2) and that the active turn changes (assertion in block 3), after this first player has played his card (block 3). For that we set up a small "toy game" with the help of the classes GameBuilder and MatchBuilder, which we implemented for the purpose to facilitate thecreation of game testing contexts (block 1). You find more details on the GameBuilder and MatchBuilder classes below.

For this specific test, a game with two players (PLAYER__1, PLAYER__2) was instantiated. The two players have two cards each (ace of clubs / queen of clubs and king of clubs / jack of clubs). There is another utility class CardValue which helps to instantiate cards easily. You find more details on the CardValue class below.

```
void the_first_player_must_play_a_card_when_round_starts() {
    // block 1
    Game game =
    GameBuilder.builder("game1")
        .withParticipation(PLAYER_1)
        .withParticipation(PLAYER_2)
        .withMatch()
        .withMatchState(MatchState.PLAYING)
        .withHandForPlayer(PLAYER_1)
        .withCards(ACE_OF_CLUBS, QUEEN_OF_CLUBS)
        .withAnnouncedScore(1)
        .finishHand()
        .withHandForPlayer(PLAYER_2)
        .withCards(KING_OF_CLUBS, JACK_OF_CLUBS)
        .withAnnouncedScore(0)
        .finishHand()
        .withRound()
        .finishRound()
        .finishMatch()
        .build();

    // block 2
    Match activeMatch = game.getLastMatch().orElseThrow();
    List<Hand> hands = activeMatch.getSortedHands();
    Hand firstHand = hands.get(0);
    Hand secondHand = hands.get(1);

    assertThat(firstHand.isTurnActive(), is(true));
    assertThat(secondHand.isTurnActive(), is(false));

    // block 3
    Round activeRound = activeMatch.getLastRound().orElseThrow();
    Card cardToPlay = firstHand.getCards().iterator().next();
    cardToPlay.setRound(activeRound);
    activeRound.getCards().add(cardToPlay);

    assertThat(firstHand.isTurnActive(), is(false));
    assertThat(secondHand.isTurnActive(), is(true));
}
```

Integrationtest

Test class: CardEventHandlerTest, Code
Test method: play_last_card_new_round_new_match()
Description: This test verifies whether a new round and a new match are created and saved in the Hibernate database when all cards have been played in a match. For sake of completeness and understandability, we not only list the test but also the setup method which is executed before the test. For that, a game with three players, each of them having two cards is instantiated. A match with two rounds where all players have played their cards is then added and all of it is saved in the database (note: we only have to save the game thanks to the JPA cascade type "ALL").

Before we call the method under test *handleAfterSave(Card card)*, we verify whether the match has been saved with its rounds and the played cards as expected (block 2). Note that we need a JPA queries for retrieving matches and cards, while the rounds are attached to the retrieved match object. This is because we do not have implemented JPA fetch type "EAGER" for all associations. With one of the played cards (it doesn't matter which one) we call the method under test and after that we read all rounds and matches that are available in the database (block 3).

There should be three rounds, (two have already been saved before and now another one should have been created) and two matches. In addition we check, whether the match and round numbers are assigned correctly (note: no round has the number 3, because for each new match, the numbers start again with 1).

```
void setup() {
    matchBuilder =
        GameBuilder.builder("game1", gameRepository, participationRepository, playerRepository)
            .withParticipation(PLAYER_NAME_1)
            .withParticipation(PLAYER_NAME_2)
            .withParticipation(PLAYER_NAME_3)
            .withGameState(GameState.PLAYING)
            .withMatch()
            .withMatchState(MatchState.ANNOUNCING)
            .withHandForPlayer(PLAYER_NAME_1)
            .withCards(ACE_OF_CLUBS, QUEEN_OF_CLUBS)
            .finishHand()
            .withHandForPlayer(PLAYER_NAME_2)
            .withCards(KING_OF_CLUBS, JACK_OF_CLUBS)
            .finishHand()
            .withHandForPlayer(PLAYER_NAME_3)
            .withCards(QUEEN_OF_HEARTS, KING_OF_HEARTS)
            .finishHand();
}

void play_last_card_new_round_new_match() {
    // block 1
    Game game =
    matchBuilder
    .withRound()
    .withPlayedCard(PLAYER_NAME_1, ACE_OF_CLUBS)
    .withPlayedCard(PLAYER_NAME_2, JACK_OF_CLUBS)
    .withPlayedCard(PLAYER_NAME_3, QUEEN_OF_HEARTS)
    .finishRound()
    .withRound()
    .withPlayedCard(PLAYER_NAME_1, QUEEN_OF_CLUBS)
    .withPlayedCard(PLAYER_NAME_2, KING_OF_CLUBS)
    .withPlayedCard(PLAYER_NAME_3, KING_OF_HEARTS)
    .finishRound()
    .finishMatch()
    .build();

    Iterable<Game> savedGames = gameRepository.saveAll(List.of(game));

    // block 2
    match = savedGames.iterator().next().getLastMatch().get();
    round = match.getLastRound().get();
    card1 = round.getCards().iterator().next();
    Collection<Round> savedRounds1 = roundRepository.findAll();
    assertEquals(2, savedRounds1.size());
    cardEventHandler.handleAfterSave(card1);
    Collection<Round> savedRounds = roundRepository.findAll();
    Collection<Match> savedMatches = matchRepository.findAll();
    assertEquals(3, savedRounds.size());
    assertEquals(2, savedMatches.size());
    assertTrue(savedRounds.stream().anyMatch(r -> r.getRoundNumber() == 1));
    assertTrue(savedRounds.stream().anyMatch(r -> r.getRoundNumber() == 2));
    assertTrue(savedMatches.stream().anyMatch(m -> m.getMatchNumber() == 1));
    assertTrue(savedMatches.stream().anyMatch(m -> m.getMatchNumber() == 2));
}
```

REST interface test

Test class: GameIntegrationTest Code
Test method: change_gameState_to_playing()
Description: This test verifies whether all required activities have been executed after the GameState attribute's value has been set to "PLAYING" by a patch request. First there is a post on the game endpoint to ensure we have a game to patch (block 1). After that, the game is patched with a new value for the GameState attribute (block 2). But to post a game, we first need a player instance for a valid security context (block 1). A patch on the game entity triggers the *handleAfterSave* method in the GameEventHandler class which in case of a value change from "WAIT" to "PLAYING" builds up the initial game context. As the whole game context is returned in the response, the successful instantiation can be verified by checking the response values of the patch request (block 3).

This test is very crucial because if the context of the game is not correctly set up, there will be unexpected behavior earlier or later during the game and it may be hard to trace back on where the error happend. So we decided to define clearly on what has to happen, when the game is started, by setting the GameState value to "PLAYING" and setting up an appropriate test with a large list of assertions, whether all associated entities have been created (by checking if every url path is there in the response).

```
void change_gameState_to_playing() {
    // block 1
    HttpHeaders responseHeaders =
        webTestClient
            .post()
            .uri("/players")
            .body(BodyInserters.fromValue(PLAYER_1))
            .exchange()
            .expectStatus()
            .isCreated()
            .expectBody()
            .returnResult()
            .getResponseHeaders();

    String sessionId = getSessionIdOf(responseHeaders);
    GAME_1.setName("game_1");
    String sessionId = getSessionIdOf(responseHeaders);
    GAME_1.setName("game_1");

    // block 2
    webTestClient
        .post()
        .uri("/games")
        .body(Mono.just(GAME_1), Game.class)
        .header(HttpHeaders.COOKIE, "JSESSIONID=%s".formatted(sessionId))
        .exchange()
        .expectStatus()
        .isCreated()
        .expectBody()
        .jsonPath("name")
        .isEqualTo(GAME_1.getName())
        .jsonPath("_embedded.participations")
        .isNotEmpty()
        .jsonPath("_embedded.participations[0].player.name")
        .isEqualTo(PLAYER_1.getName());

    // block 3
    Long id = gameRepository.findAllByName("game_1").get(0).getId();
    String uri = "games/" + id.toString();
    GAME_1.setGameState(GameState.PLAYING);

    Map<String, GameState> patchBody = Map.of("gameState", GameState.PLAYING);
    // Without check whether the game exists (no get()) change the gameState with patch() request
    webTestClient
        .patch()
        .uri(uri)
        .contentType(MediaType.APPLICATION_JSON)
        .header(HttpHeaders.COOKIE, "JSESSIONID=%s".formatted(sessionId))
        .body(BodyInserters.fromValue(patchBody)) // Game 2 has different gameState = PLAYING
        .exchange()
        .expectStatus()
        .isOk()
        .expectBody()
        .jsonPath("_embedded.matches")
        .value(hasSize(1))
        .jsonPath("_embedded.matches[0].rounds")
        .value(hasSize(1))
        .jsonPath("_embedded.matches[0].rounds[0].roundNumber")
        .isEqualTo(1)
        .jsonPath("_embedded.matches[0].rounds[0].cards")
        .value(hasSize(0))
        .jsonPath("_embedded.matches[0].matchNumber")
        .isEqualTo(1)
        .jsonPath("_embedded.matches[0].matchState")
        .isEqualTo(MatchState.ANNOUNCING.name())
        .jsonPath("_embedded.matches[0].hands")
        .value(hasSize(1))
        .jsonPath("_embedded.matches[0].hands[0].announcedScore")
        .value(nullValue())
        .jsonPath("_embedded.matches[0].hands[0].cards")
        .value(hasSize(5))
        .jsonPath("_embedded.matches[0].hands[0].participation")
        .value(notNullValue());
}
```

Future regressions

The three test expamles make verify the behavior of the Screw-Your-Neighbor system on basis of game rules. These rule will never change, as the game rules have been agreed and specified in advance. So no matter how the method under test is being changed, the result of their executions must always be the same. These methods are therefore very important for regression testing, to ensure no new code breaks a working implementation of the game rules.

All three tests cover core functionalities of the game system. The methods are all very often called during the game, so thorough testing is of high interest.

GameBuilder class

The GameBuilder allows to instantiate a game at any point in time resp. possible state during the game. As a test writer you are responsible of setting up the game according to the rules and with consistent data (i.e. not creating two players and distributing one of them three and the other one only two cards.) Code

MatchBuilder class

CardValue class

With this class we can very easily instantiate cards and it provides in addition a method to compare them by their identity (i.e on their rank **and** suit, not only on their rank, like the original method from the Card class does). This feature is needed for some tests, where we want to verify which card has been played. Code