System design document for Dance Fans

TDA367

Authors:

Hedy Pettersson, Jakob Persson, Joar Granström,

David Salmo, Johan Berg, Isabelle Ermeryd Tankred

Oct 24 2021

Version 3

1 Introduction

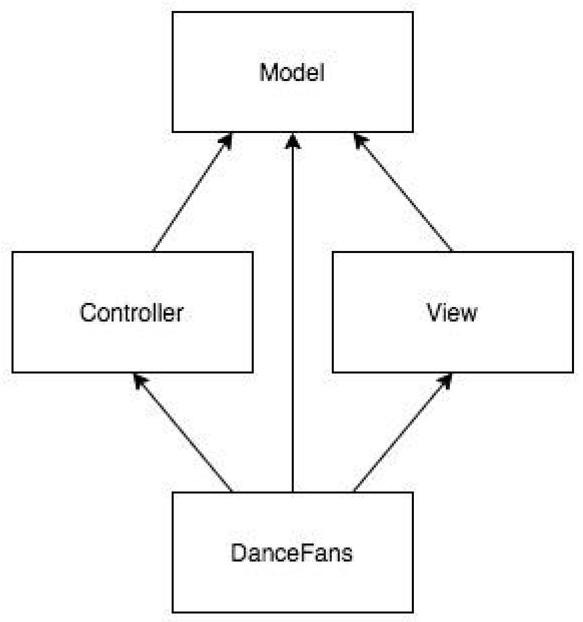
Dance fans is a strategy game where two players play as dancers who perform dance moves to get the most dance fans on the dancefloor. An object oriented programming approach was taken to implement the game, as is described in more detail in this document.

1.1 Definitions, acronyms, and abbreviations

GUI = Graphical User Interface.
MVC = Model-View-Controller. A design pattern which is commonly used if you develop applications with a GUI. It separates data and logic (model) from the view and controller.
LibGDX = A library for developing games in Java.
TexturePacker = A tool for creating sprite sheets.

Game specific definitions
Player = a user of the program and their corresponding representation in-game.
Main dancer = The dancer the user controls to make dance moves in order to gain dance fans and win the game.
Dance fan = One of the dance fans, which join the dance floor inspired by the Main Dancers dance moves. The player with the most dance fans at the end of the game wins.
Dancer = A dance fan or a main dancer
Dance floor = The world in which a dancer can be located and move around within, in the game.
Dance move = A pattern and a set number of steps a player can move across the dance floor. The pattern tells us how the dance floor will change after the dance move is performed. The dance move inspires dance fans to join in on the player’s dance team. These dance moves are represented in the game as cards in a deck of cards (a main dancer’s repertoire of dance moves).
Dance pattern = The pattern part of the dance move.
Card = Representation of dance moves in the game, to help the player choose strategy.
Preview = A special view of what the dance floor would look like if the player performed the dance move of the currently selected card at the location the selection marker is currently located on the dance floor.

2 System architecture



We use the MVC pattern. We have a module containing classes which handles data for the game. The model module is not depending on anything but itself. We also have a class which handles our GUI and a controller class which handles inputs from the user. We also have a class called Dance Fans which connects the parts and starts the program.

Parts in the model:

DanceFloor contains a list of tiles and the order of them. It is also responsible for changing the details of the specific tiles. Also handles counting what is on the tiles.

DanceFloorTile contains information about what is currently standing on the tile, and can tell us more details about the object on it.

Coordinates are used to navigate and keep track of locations on the danceFloor and consist of two ints.

CardDeck contains multiple lists of card:s, and in what order they are in. The different lists are the currently opened cards, the used cards, the unused cards, and a copy of the original list which was used to create the deck in the first place. It can shuffle cards in a list, and also handles moving the cards between the different lists.

Card contains a matrix which is used to represent the dance pattern, an int for the amount of steps and an image representation of the card. As of now the cards in the game have been manually added in, but automating the process is something we would like to do in the future.

FloorObject has coordinates and keeps track of different aspects of the object for now. (Like color and type)

Dancer extends FloorObject but does not do anything special as of now, but we think they might have some more specific behaviour later.

MainDancer extends Dancer and also has an extra set of coordinates, which are for its temporary position. Representation of the player on the dancefloor.

DanceFan extends Dancer, more functionality seems likely to be added in the future.

Player has a CardDeck, a MainDancer, and two DanceFans as of now. Player is what ties together many of the parts of the game.

3 System design

See the UML-diagram in appendix 1.1.

We have two packages as of now, the model and enums. The model does not access the view or the controller.

Some of the enums could possibly be removed later. They were introduced during refactoring to give better structure and more readable code, by removing representation of different things using ints and replacing them with enums.

The view accesses the model and uses information to display information, like which players turn it is, if someone has won yet. It also uses a dancefloor to display the characters and the dancefloor itself.

The controller calls on methods in the model. We have thought about using listeners, but have not implemented any yet.

4 Patterns

Chain of responsibility: in order to not break the law of demeter we have tried to implement the chain of responsibility pattern. For example, calling getCoordinates on a player is then handled in the player to get the coordinates from the mainDancer (which is the representation of the player on the dancefloor), which is an extension of Dancer, which is an extension of FloorObject, which has the method of getting coordinates.

Bridge pattern: In the project we use different types of dancer, main dancer and dance fan. These could have different colors depending on which player they belong to, since it’s a two-person game we have two colors right now, red and green. The colors could be extended in the future. The colors and dance types could be combined in indepently. To apply the colors, independently on what type the dancer is, we use the bridge pattern.

Prototype: We have created methods for creating copies of danceFloor, which we think is implemented in a similar way to how the prototype pattern is usually implemented.

Iterator: The iterator pattern would probably work really well in our game and work to apply this pattern has been started. Unfortunately, due to the time limit of this project, we never finished the work and the pattern has not been implemented in the latest version of our game. However there is a branch which has started the implementation and the work could be continued in the future.

Observer: Quite late in the development we realised it would have been good to use listeners and observer for the communication between our MVC-parts. This would have reduced dependencies a lot but once again, this is something we didn’t have time to implement.

5 Persistent data management

The game does not need the players to have profiles or similar data stored.

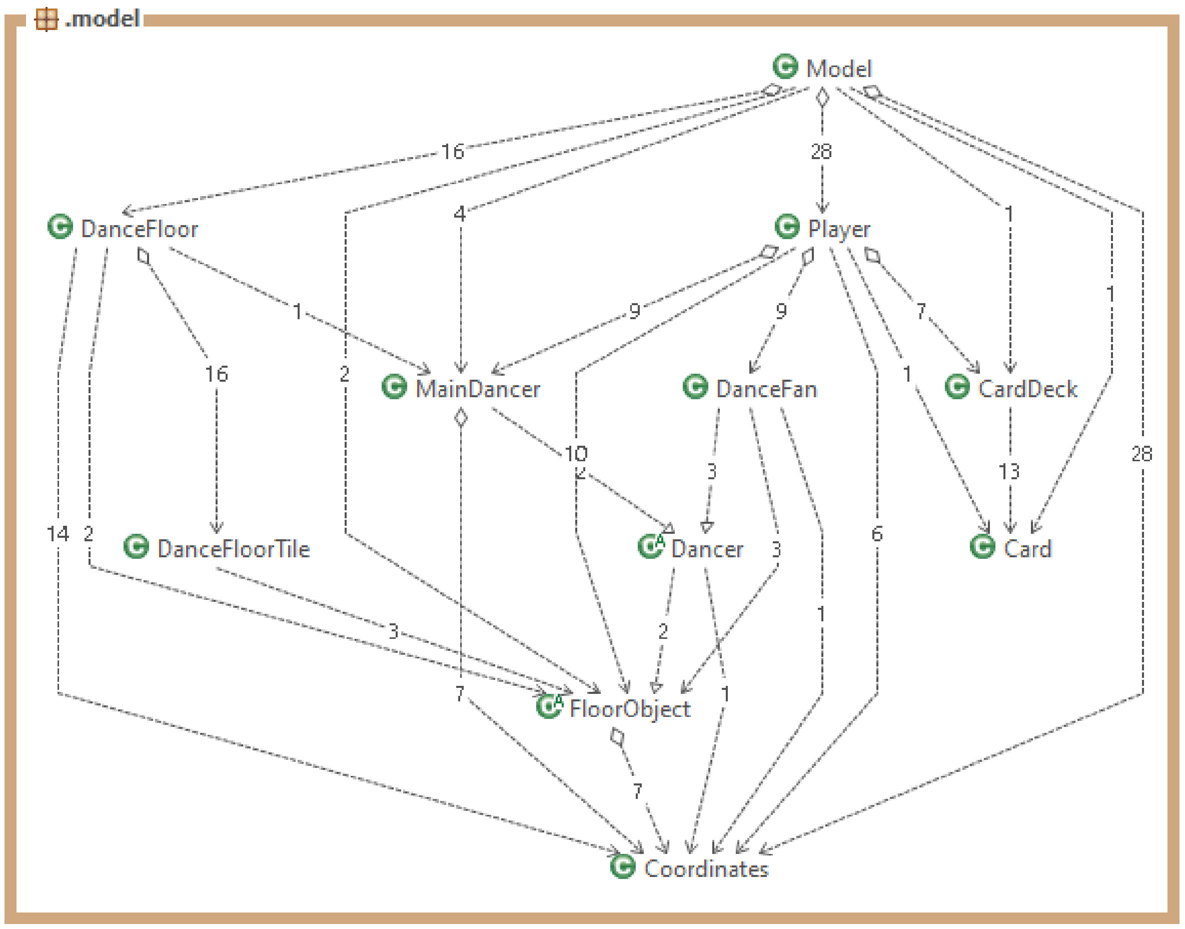
Icons and images have been saved in sprite sheets using TexturePacker. These are paired up with text files which describe the location of each image and using atlases this makes it easier to store many images, and will hopefully make it easier to add many new icons and images.

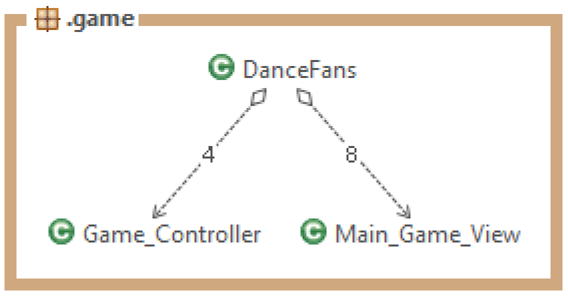
5.1 Access control and security

Not Applicable

6 Quality

During this project we have used JUnit tests for our project. The tests cover all most all parts of the model.
Running a dependency analysis with Stan4j returns the following graph.





Current issues:
Some analytical tools have been used (STAN), but we should use more.
As mentioned in previous chapters, more design patterns could definitely have been implemented to reduce dependencies and get a better code structure.
If we had more time we would have liked to split our view class into several to handle different parts of the screen and maybe also add a start screen to our game.

7 References

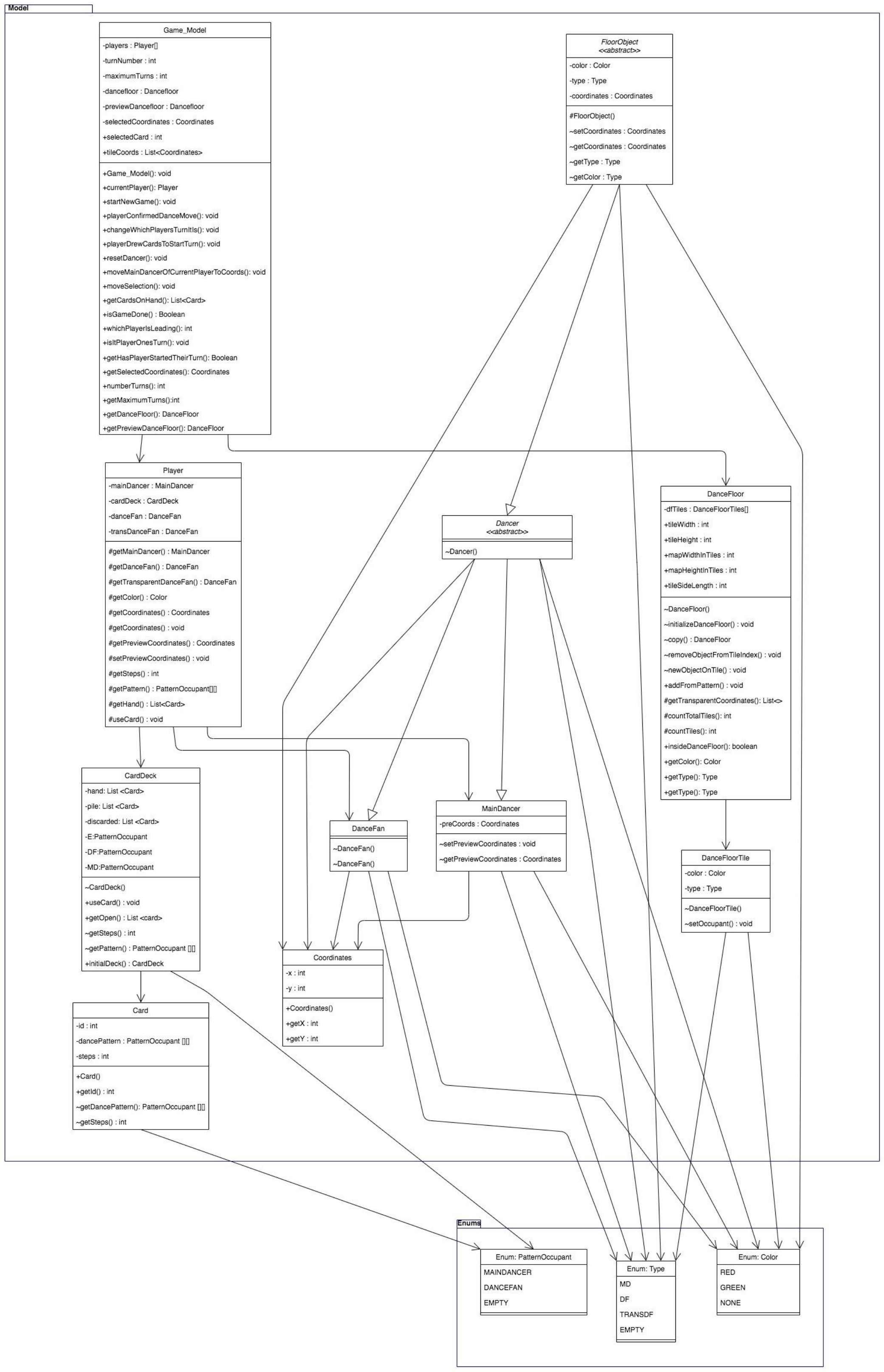
LibGDX
https://github.com/rafaskb/awesome-libgdx#readme

TexturePacker
https://www.codeandweb.com/texturepacker

STAN
http://stan4j.com/

7 Appendix

1.1


Model
Game_Model
-players : Player[]
-turnNumber : int
-maximumTurns : int
-dancefloor : Dancefloor
-previewDancefloor : Dancefloor
-selectedCoordinates : Coordinates
+selectedCard : int
+tileCoords : List<Coordinates>
+Game_Model(): void
+currentPlayer(): Player
+startNewGame(): void
+playerConfirmedDanceMove(): void
+changeWhichPlayersTurnItIs(): void
+playerDrewCardsToStartTurn(): void
+resetDancer(): void
+moveMainDancerOfCurrentPlayerToCoords(): void
+moveSelection(): void
+getCardsOnHand(): List<Card>
+isGameDone() : Boolean
+whichPlayerIsLeading(): int
+isItPlayerOnesTurn(): void
+getHasPlayerStartedTheirTurn(): Boolean
+getSelectedCoordinates(): Coordinates
+numberTurns(): int
+getMaximumTurns():int
+getDanceFloor(): DanceFloor
+getPreviewDanceFloor(): DanceFloor
FloorObject
<<abstract>>
-color : Color
-type : Type
-coordinates : Coordinates
#FloorObject()
~setCoordinates : Coordinates
~getCoordinates : Coordinates
~getType : Type
~getColor : Type
Player
-mainDancer : MainDancer
-cardDeck : CardDeck
-danceFan : DanceFan
-transDanceFan : DanceFan
#getMainDancer() : MainDancer
#getDanceFan() : DanceFan
#getTransparentDanceFan() : DanceFan
#getColor() : Color
#getCoordinates() : Coordinates
#getCoordinates() : void
#getPreviewCoordinates() : Coordinates
#setPreviewCoordinates() : void
#getSteps() : int
#getPattern() : PatternOccupant[][]
#getHand() : List<Card>
#useCard() : void
Dancer
<<abstract>>
~Dancer()
DanceFloor
-dfTiles : DanceFloorTiles[]
+tileWidth : int
+tileHeight : int
+mapWidthInTiles : int
+mapHeightInTiles : int
+tileSideLength : int
~DanceFloor()
~initializeDanceFloor() : void
~copy() : DanceFloor
~removeObjectFromTileIndex() : void
~newObjectOnTile() : void
+addFromPattern() : void
#getTransparentCoordinates(): List<>
#countTotalTiles(): int
#countTiles(): int
+insideDanceFloor(): boolean
+getColor(): Color
+getType(): Type
+getType(): Type
CardDeck
-hand: List <Card>
-pile: List <Card>
-discarded: List <Card>
-E:PatternOccupant
-DF:PatternOccupant
-MD:PatternOccupant
~CardDeck()
+useCard() : void
+getOpen() : List <card>
~getSteps() : int
~getPattern() : PatternOccupant [][]
+initialDeck() : CardDeck
MainDancer
-preCoords : Coordinates
~setPreviewCoordinates : void
~getPreviewCoordinates : Coordinates
DanceFan
~DanceFan()
~DanceFan()
DanceFloorTile
-color : Color
-type : Type
~DanceFloorTile()
~setOccupant() : void
Coordinates
-x : int
-y : int
+Coordinates()
+getX : int
+getY : int
Card
-id : int
-dancePattern : PatternOccupant [][]
-steps : int
+Card()
+getId() : int
~getDancePattern(): PatternOccupant [][]
~getSteps() : int
Enums
Enum: PatternOccupant
MAINDANCER
DANCEFAN
EMPTY
Enum: Type
MD
DF
TRANSDF
EMPTY
Enum: Color
RED
GREEN
NONE