QT Solitaire
Group 3

Project Plan

Project Scope:

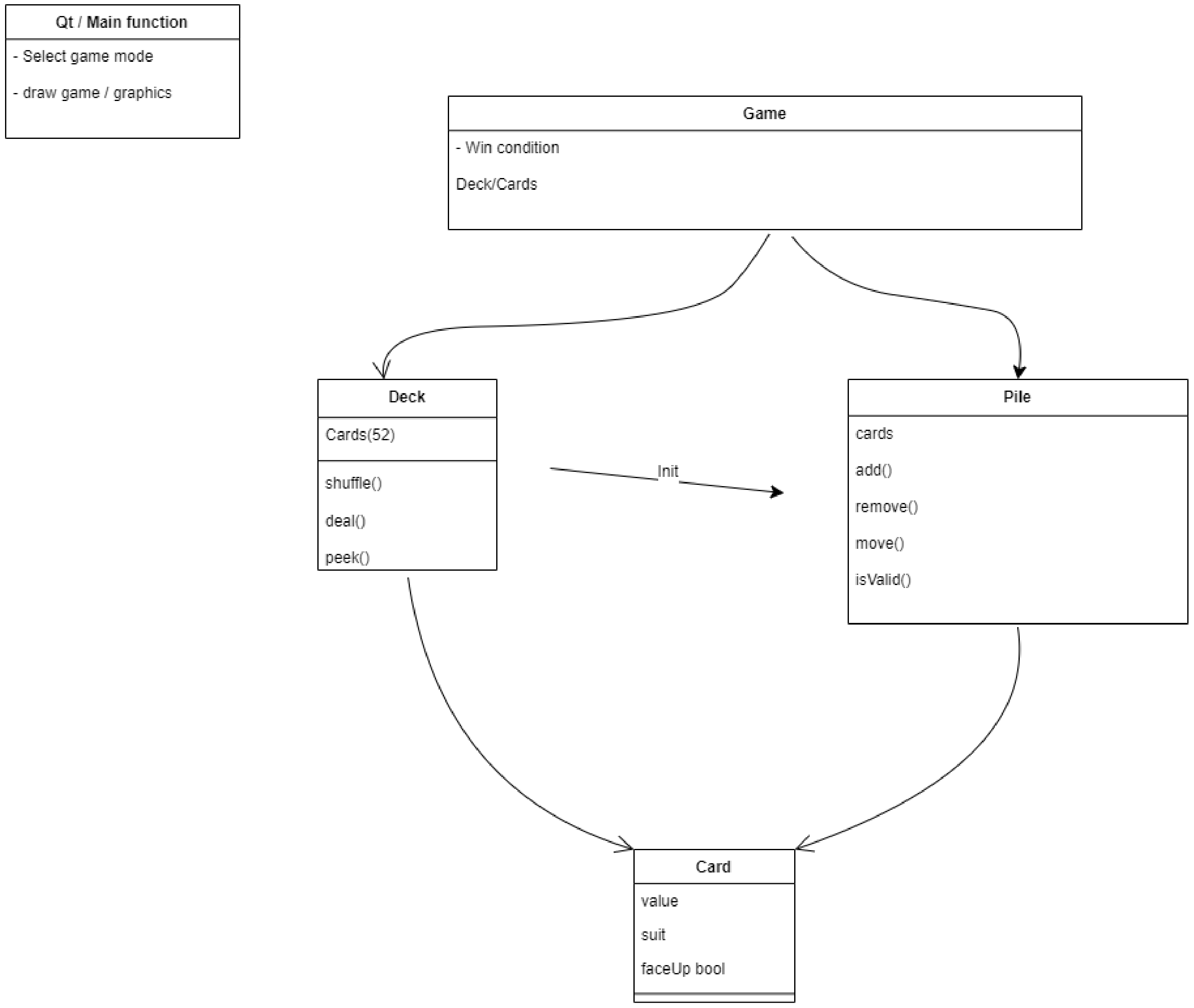
The initial plan for the project is to at least have a fully functioning "Klondike" solitaire game-mode, as well as a minimal 2D graphics implementation using Qt. The main goal regarding building the program is firstly to get it running on a Linux machine, since this is what the advisor will be using. Ideally, however, instructions should be provided for Mac and Windows as well.

The program is structured around a single process workflow, beginning with the user starting a new game. The program automatically initializes a game session with the board, deck, and relevant card piles. The game automatically shuffles the deck and splits the cards into the piles. After the initialization, the game is driven by a user-action loop, consisting of the user making a valid move, and the game reacting to it and updating its state. This mainly consists of checking the validity of the moves, as well as checking whether the win or loss conditions evaluate to true. This loop continues until the player either wins or loses, after which the game is over, or runs into a temporary roadblock that can be resolved by e.g. reshuffling the deck. The user actions will be registered through simple inputs, e.g. via mouse clicks.

Additional features are to be implemented if time and complexity allow for it. The highest-priority extra feature is the addition of other solitaire game modes, where the user can select a game mode upon starting a game. The software should therefore be developed in a modular enough manner to allow for this extension, and the new game modes would simply restructure some of the rules, conditions and deck setup. User statistics tracking is also on the list of extra features, along with smaller add-ons for a better user experience like sound effects and neater graphics.

QT Solitaire
Group 3

High-level Structure:



Setup

1. Init game
2. Init deck
3. Move cards into piles (init piles)

Game loop

Move (draw, move)

Check validity / win & lose condition

The main classes as of now, from top down are Game, Deck, Pile and Card. The QT application functions as the entry point of the program, as well as the driver of the entire application, reacting to user inputs and initializing and interacting with these classes. It is also, naturally, responsible for the graphics implementation of the game.

The game class contains fields and methods relevant to a game instance itself, main ones being the specific conditions for winning or losing the game, and the relevant data structure for setting up the board with decks and cards/piles. The deck, which is composed of the 52 cards (and possibly more depending on game mode), is initialized and shuffled, after which it is divided into a set number of piles on the game board. The concrete implementation of this again depends on the game mode. The user is intended to play the game by interacting with the specific piles on the board. Cards in piles can be added, removed, and moved between piles by utilizing the class methods. The piles should also implement validity checks for each game mode to allow or deny certain compositions or states. The cards stored in piles contain general information fields like suit, number and whether it is face-up or face-down, as well as possible related methods for modifying this orientation.

There exists no specific board class for simplicity, as our current understanding is that it only complicates the development process through unnecessary coupling. The board is inherently a graphical construct not needed for the game logic, and will be implemented graphically in the QT app.

External Libraries:

As of now, QT is the main external library, which will be used for all things graphics-related. It will implement the main app/driver functionality and GUI, reacting to user-events and displaying the subsequent changes in state through images and animations. Furthermore, QT Multimedia can provide audio-related functionality for the extra features.

We have decided to use some statistics-related library for simplifying development and perhaps providing advanced statistics that go beyond curiosity, but can actually help the user improve. The choice of a specific library, however, has not yet been made.

QT Solitaire
Group 3

Sprint Plan Details:

Important dates / target dates for meetings:

Sprint 0 goal: project plan done, establish working practices, set up dev env

Sprint 1 goal: basic GUI, game logic/rules implementation

X.11.2024: Sprint 1 planning

Wed 6.11.2024: Sprint 1 weekly 1

Wed 13.11.2024: Sprint 1 weekly 2

X Y.11.2024: Sprint 1 Review with Advisor

Fri 15.11.2024: Sprint 1 DL

Sprint 2 goal: connecting GUI with logic, statistics, abstractions and optimizations

Monday 18.11.2024: Sprint 2 planning

Wed 20.11.2024: Sprint 2 weekly 1

Wed 27.11.2024: Sprint 2 weekly 2

X Y.11.2024: Sprint 2 Review with Advisor

Fri 29.11.2024: Sprint 2 DL

Sprint 3 goal: finish up project, documentation, final commits

Monday 2.12.2024: Sprint 3 planning

Wed 4.12.2024: Sprint 3 weekly 1

Wed 11.12.2024: LAST DATE TO DEMO PROJECT TO ADVISOR

Fri 13.12.2024: final project DL, last push to git!

QT Solitaire
Group 3

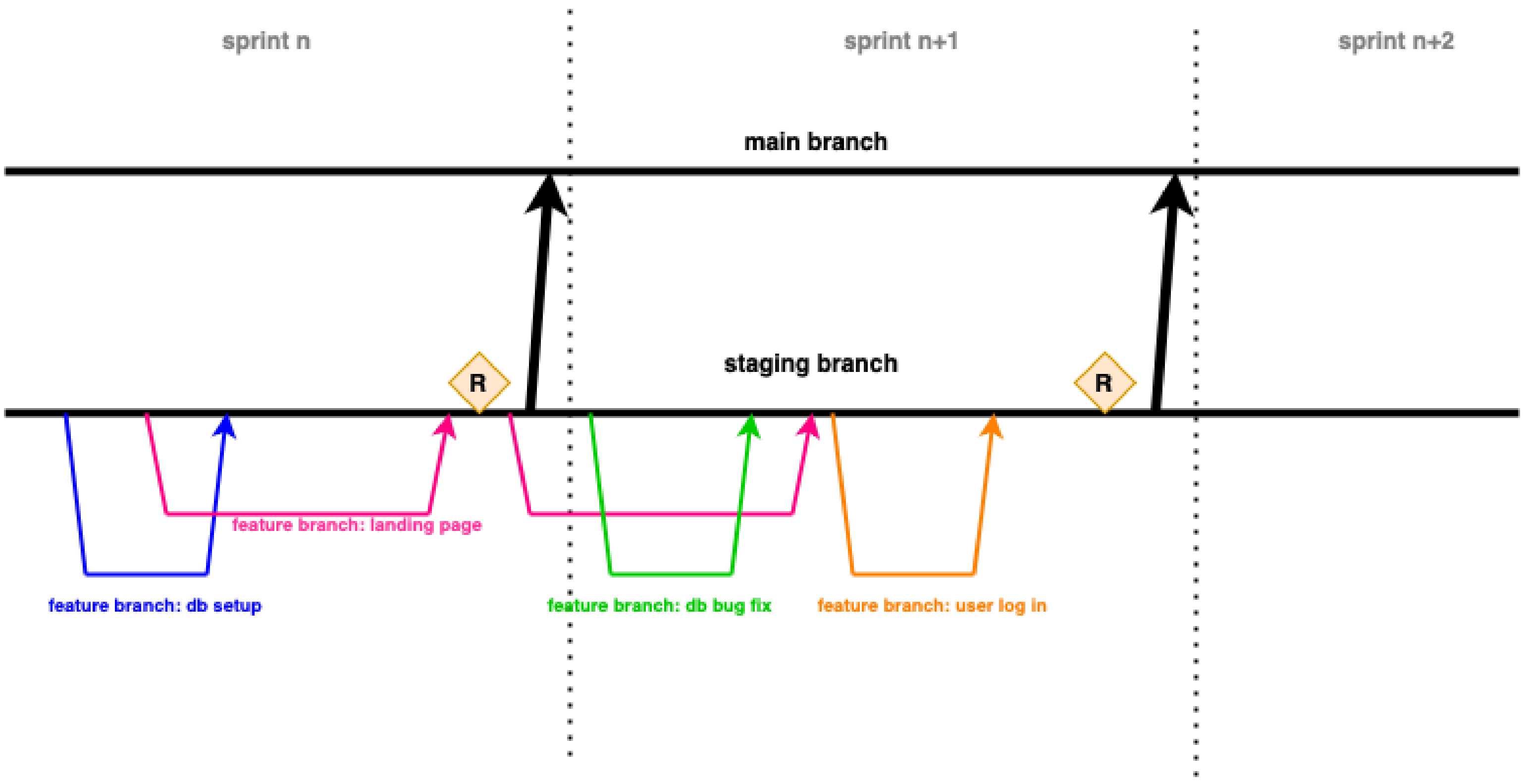
Sprint "dailies"/weekly meeting times: Wednesdays 16:30

Sprint Planning: ≈ first date of each sprint

Sprint review: ≈ last date of each sprint (depending on Advisor's availability)

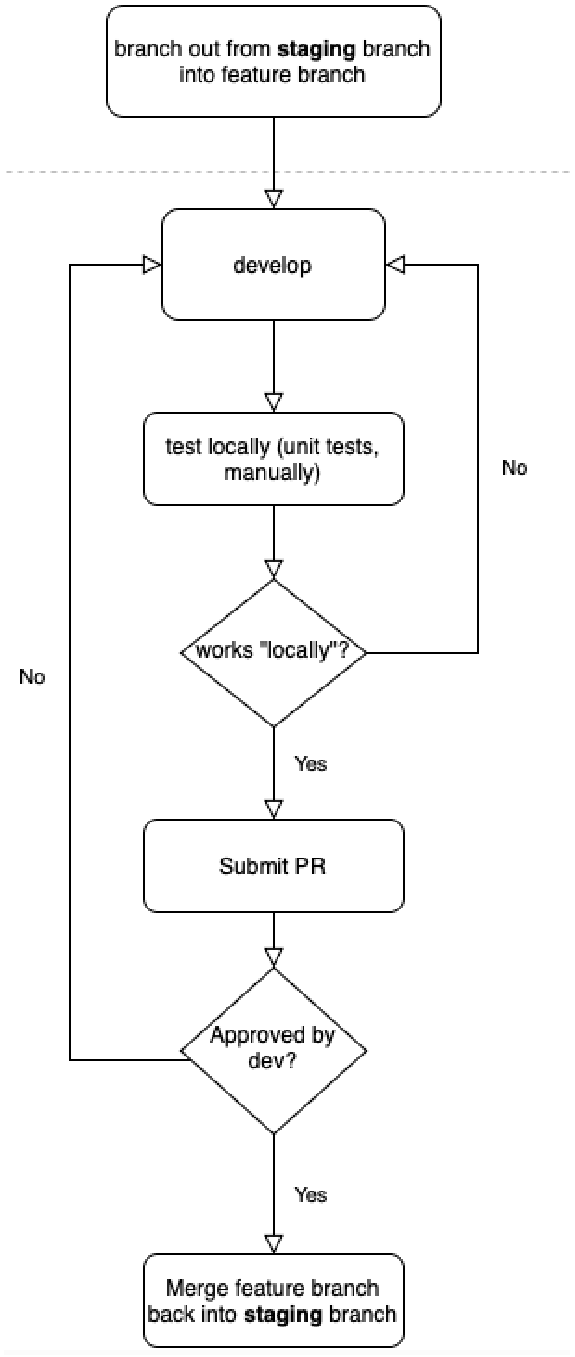
Documentation: rotating documenter each meeting (also dailies)

Git working practices:


sprint n
sprint n+1
sprint n+2
main branch
staging branch
R
R
feature branch: landing page
feature branch: db setup
feature branch: db bug fix
feature branch: user log in

QT Solitaire
Group 3

Sprint development workflow:


branch out from **staging** branch into feature branch
develop
test locally (unit tests, manually)
works "locally"?
No
Yes
Submit PR
Approved by dev?
No
Yes
Merge feature branch back into **staging** branch