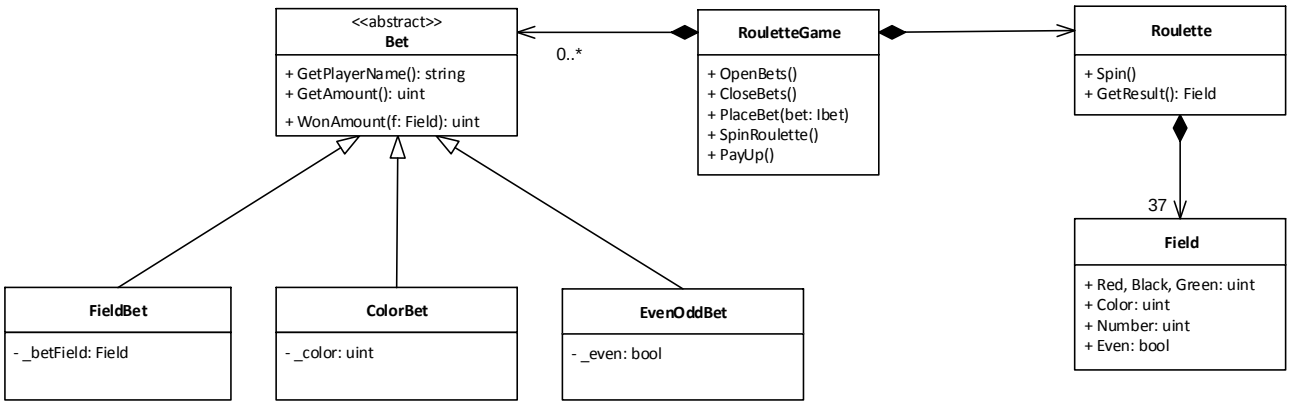
Lab exercise 10: Fakes 1 (primarily stubs) – The Roulette Game

In this exercise, you will create a Roulette Game. You will create fakes, most prominently *stubs* (configurable ones where applicable) and put them to use in the unit test of the system before you put the modules together and play the game.

Note: You shall solve this exercise in your teams – *not* alone.

Note: This is a fairly big exercise. Do yourself a favor: Work in parallel and *get started in time!*

A sketch of the system structure is given below. Note that the UML class diagram is only a sketch designed to introduce the concepts of the game.



The Roulette game consists of various concepts as described below:

The *Roulette* itself, which can be spun and which can be queried for the most recent result. The roulette consists of 37 *Fields*, each with a number (0..36) and color (0 is green, 1..36 is red or black). The roulette can be spun which results in a random result, namely 1 of the defined 37 fields.

A *Bet*, which is identified by the name of the betting player, the amount to bet, and the kind and value of bet (e.g. “Pete Mitchell”, “1000”, “Red/black”, “Red”). Three different kinds of bets are allowed:

- Betting on individual numbers (0-36) – pays back 36 times the amount
- Betting on even/odd numbers - pays back 2 times the amount
- Betting on Red/black - pays back 2 times the amount

The *Roulette Game*, which controls the opening and closing for bets, and placing of bets, spinning the roulette, payment of wins to players etc. The roulette game should allow several *bets* to be placed on the roulette.

The roulette game runs in rounds. When a round starts, the bets are opened and bets can be placed. Sometime later, the bets are closed (“*rien ne vas plus*”) - after this time, placing a bet is considered an error. After the bets are closed, the roulette game will spin the roulette and subsequently check if any bets are won. If so, a notification of the player name, kind of bet and amount won should be made e.g. to the console. Initially, consider the game to have an unlimited amount of money.

You are provided with a working implementation of the game, but this code is neither tested nor designed for test. Your task, in your group, is to refactor and test, using fakes where applicable.

Exercise 1
In the group: Discuss the work to be done, e.g.:

- How do the individual objects work together to create a "gaming experience"? Draw a sequence diagram if you need to.
- How should the system be refactored so that the design is more testable than the one implemented?
- How will we use fakes (especially configurable stubs) in the unit test of the classes? Do we need a boundary value analysis for any of them?
- How will we communicate with the user? Using the console? Should we have some sort of "façade" for this between the console and the Roulette game?
- How will we divide work between us? Who implements and tests what?

Set up a Jenkins build connected to the Git repo for the exercise and use it during your implementations.

Exercise 2:
Individually: Implement and test the classes assigned to you. Be *sure* that the tests are thorough and complete, and be *sure* to communicate with your team mates along the way. Whenever you have something worth sharing (i.e. working and tested), commit and push.

Exercise 3:
In the group: Put the system together, make it work and play the game. Have fun!

Exercise 4:
If you have the time, introduce the concept of a *Bank*. A Bank is where the Roulette game deposits bets from players and withdraws money when a player wins. The bank is initialized with a certain amount of money and if/when this is depleted, the game should be so notified. The bank should be notified when a round is opened. At this time, the bank should store the present amount in a list, so that the casino can later retrieve statistics on the income for the day.

Implement the Bank, test it, and integrate it into the game.

Exercise 5:
If you still have time, think of what else could be introduced. One obvious idea would be the concept of a *Player* class The player could have a certain amount of money initially, and could join and leave the game. The player – like the bank – could also maintain statistics of his amount left through rounds. A player could be used for automated testing of the game, etc. You could also consider introducing some sort of GUI for placing bets, showing results etc., or maybe a *Bankier* who could open and close bets, spin the roulette etc.

Again, implement, test, and integrate the things you come up with.

Exercise 6:
If you still have time, you can leave early today!