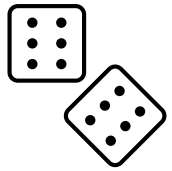
GOMOKU

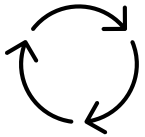
Massimiliano Cristarella • Matteo Ferfoglia • Giovanni Marchetto

Outline

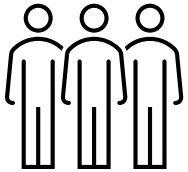


Gomoku

Project architecture and design choices



Project evolution



Team

Gomoku

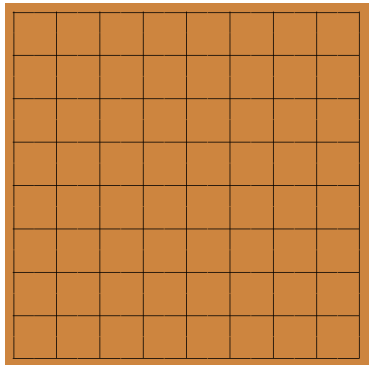
- The game
- Generalization
- Live demo

The game

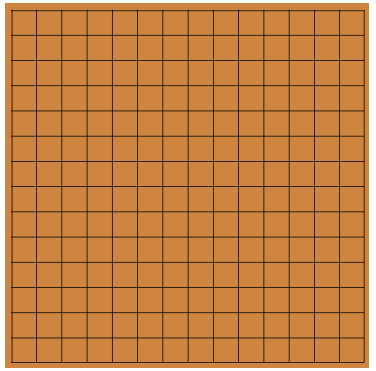
- Abstract strategy board game
- Board of size 19 x 19
- Two players: black and white
- Players alternate turns placing a stone of their color on an empty intersection
- Black player starts
- Player who places exactly 5 consecutive stones wins

Generalization

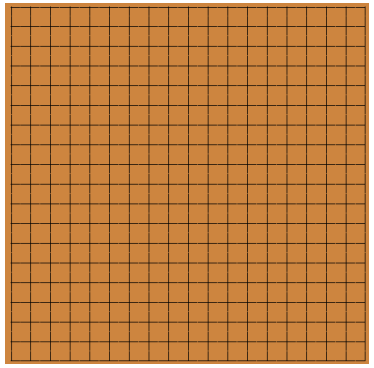
- Players play a number of games
- The set of games composes a match
- Color of players is inverted at each game
- Board size can be chosen
- Player who places *at least* 5 consecutive stones wins



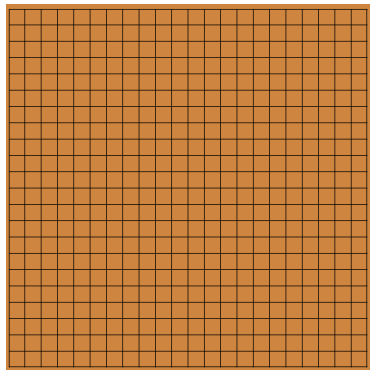
9 x 9



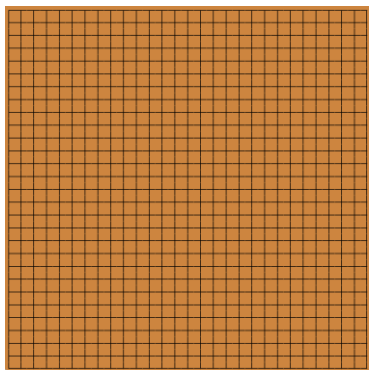
15 x 15



19 x 19



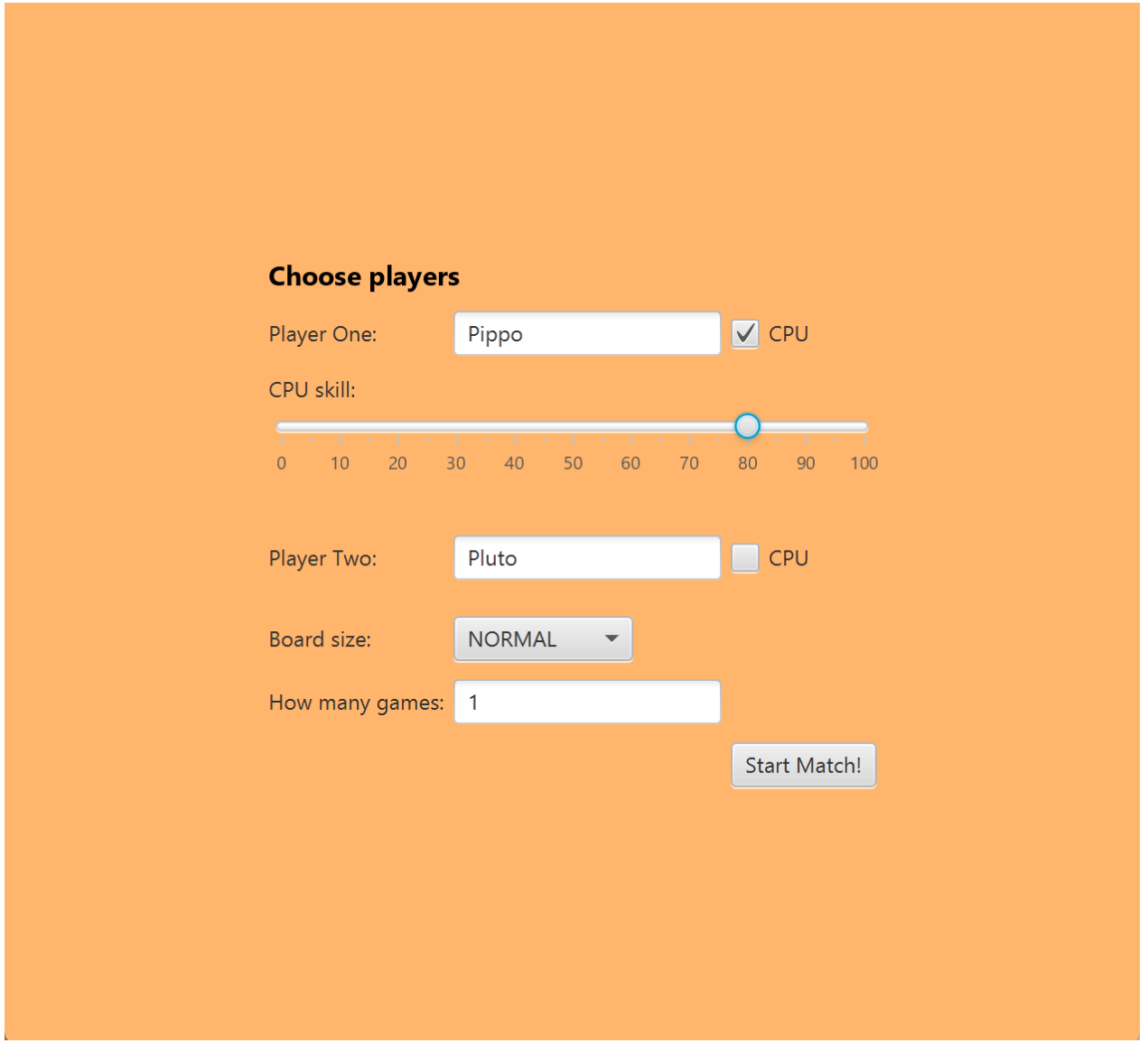
23 x 23



29 x 29

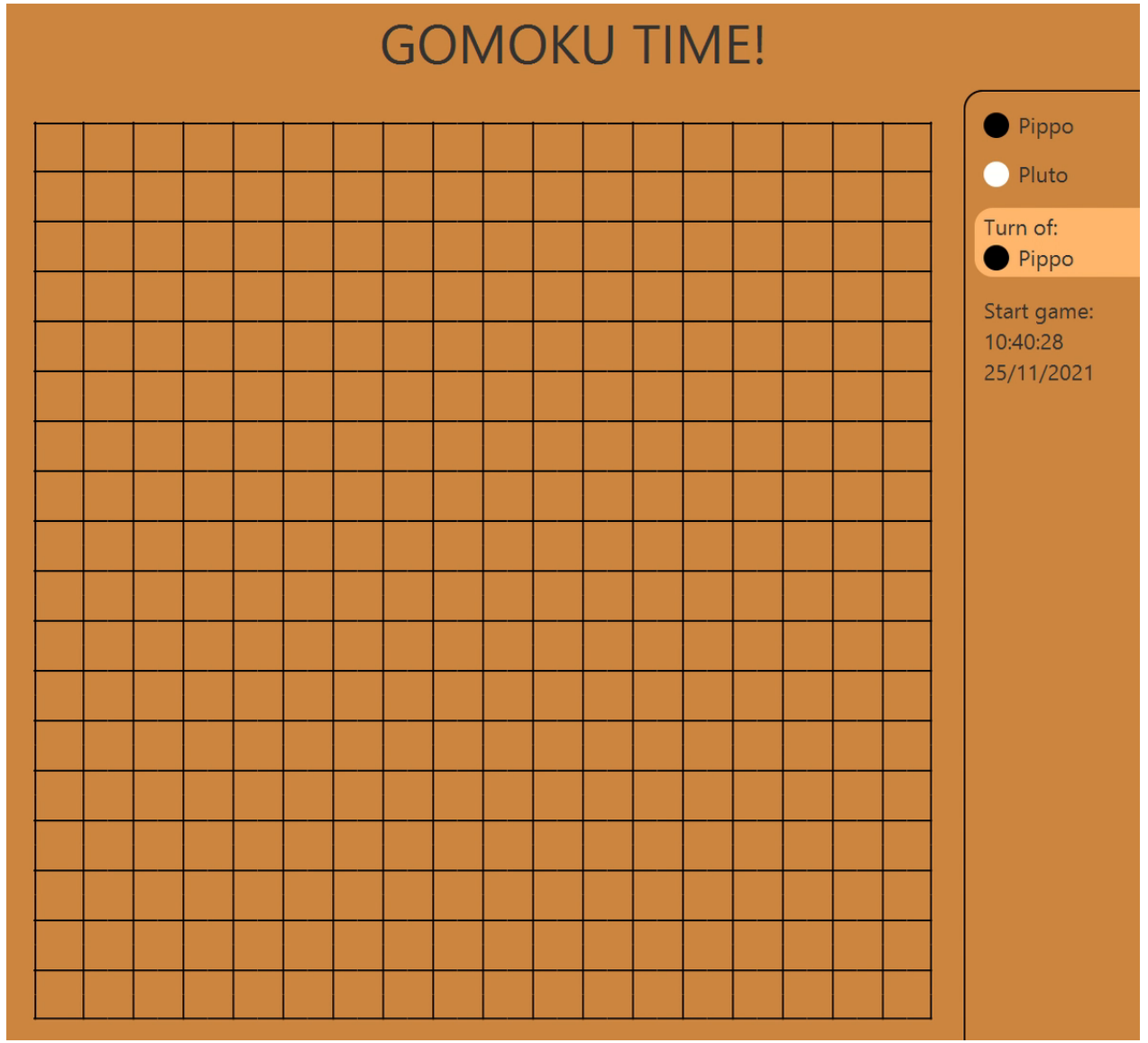
Live demo

- Start view allows to set
  - Player names
  - If adversary is CPU and difficulties
  - Board size
  - Number of games



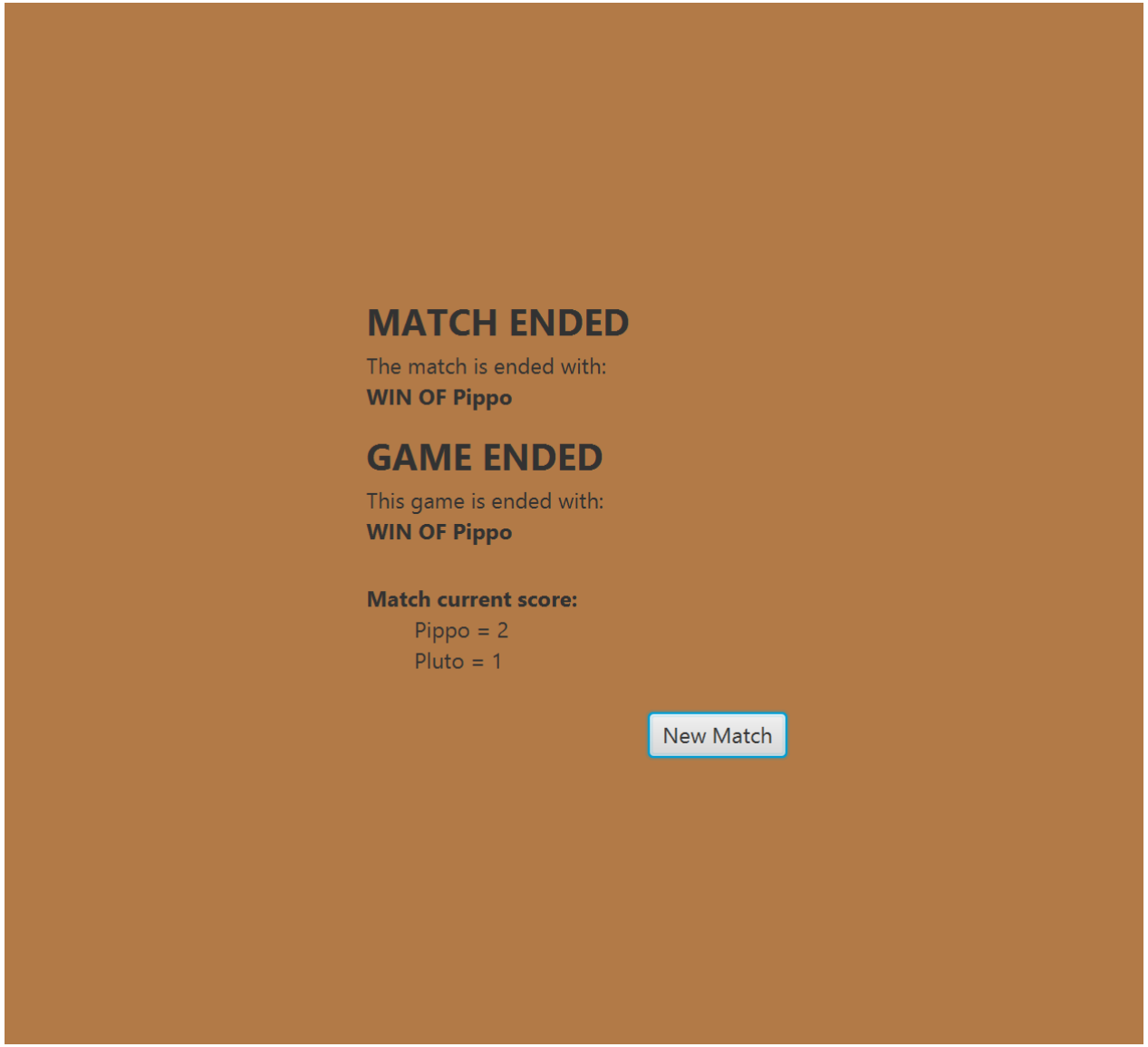
Live demo

- Players dispute the game
  - Red stroke on the stone indicates the last move



Live demo

- Summary view is shown at the end of each game, allows to:
  - Continue the match (if it is not ended)
  - Start new match (if the match is ended)
  - Start an extra game (if the match is ended with draw)



Live demo

- Application also runs with CLI

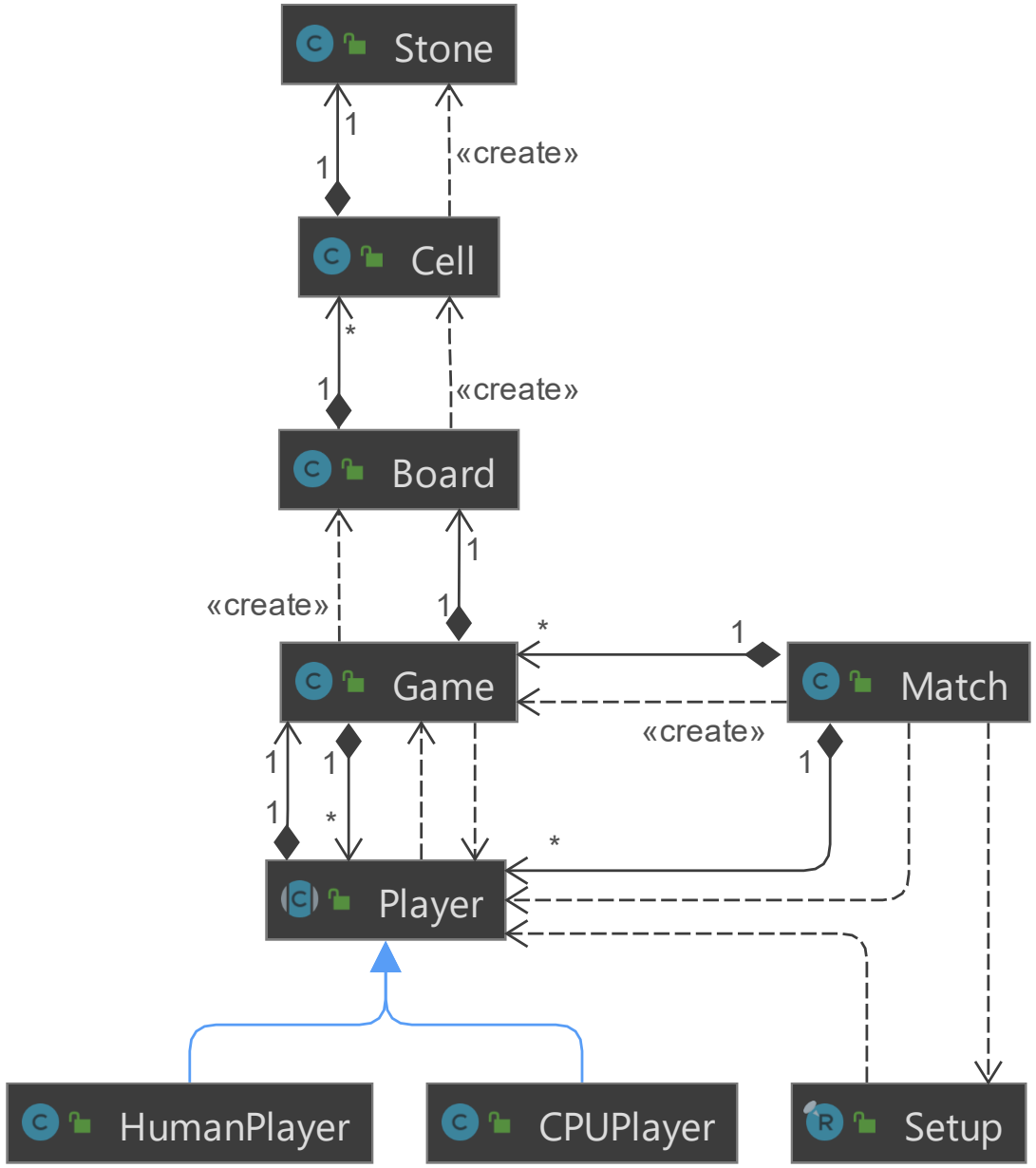
```
Do you want to play with GUI or CLI? cli
How many players? Choose 0 for CPU VS CPU, 1 for PERSON VS CPU, 2 for PERSON VS PERSON: 1
Name of player 1: Mark
Skill factor of player 2 (between 0 and 1): 0.6
Choose the board size (1 for VERY SMALL, 2 for SMALL, 3 for NORMAL, 4 for BIG, 5 for VERY BIG): 3
How many games? 2


New game!
    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 0|
 1|
 2|
 3|
 4|
 5|
 6|
 7|
 8|
 9|
10|
11|
12|
13|
14|
15|
16|
17|
18|

Turn of Mark
Insert next move:
   Row coordinate:
```

Project architecture and design choices

- Model
- Custom types and exception
- User interface

Model

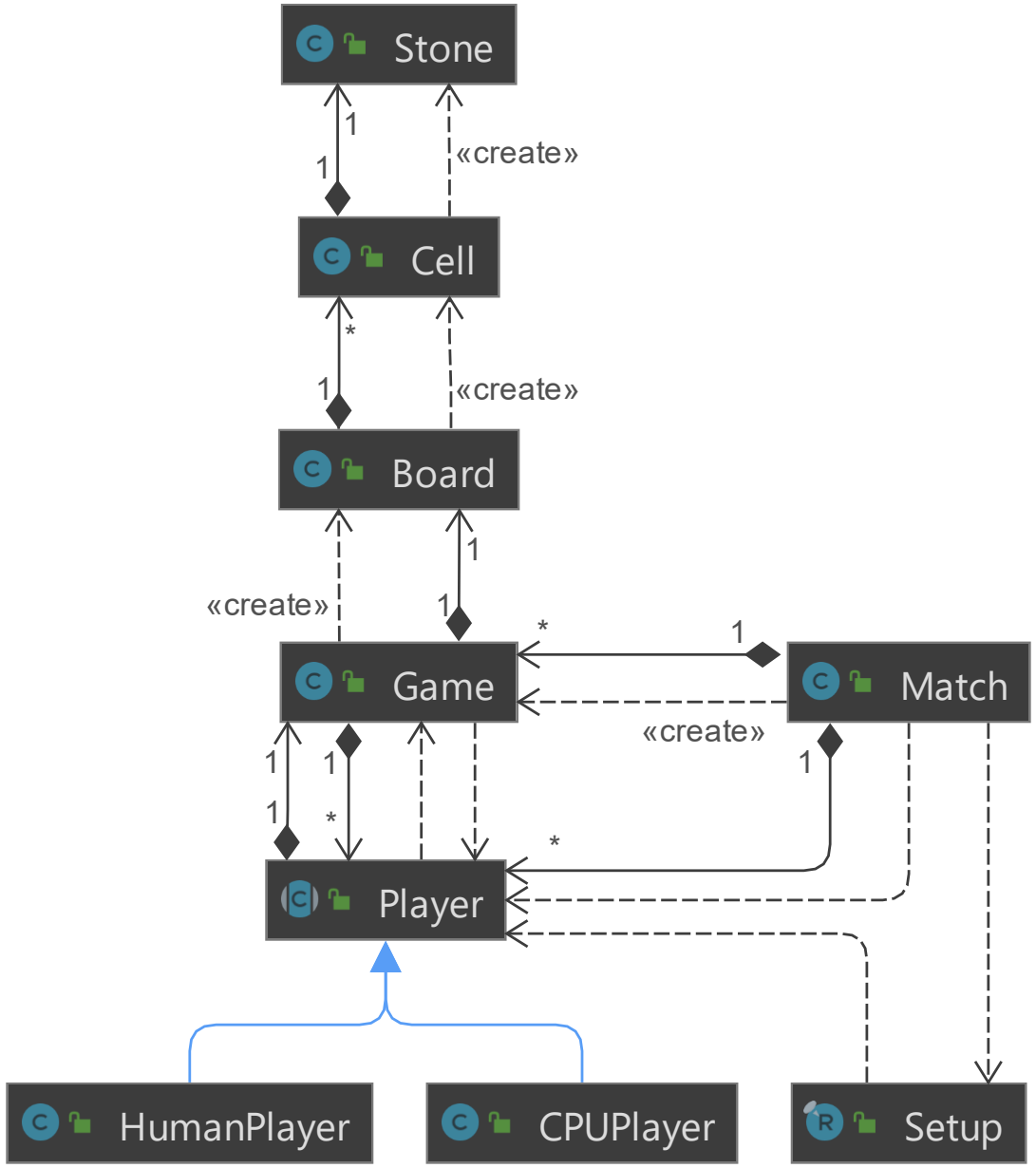
- Stone
  placed by a player, can be black or white
- Cell
  can contain a stone
- Board
  with the matrix of cells
- Player
  can be human or CPU
- Game
  records turns of players till the end
- Match
  is the set of games



A ⇢ B means that A «uses» B (e.g., a method invocation)

Model

- Entities follow the Single Responsibility principle
- High cohesion
- Low coupling



A ⇢ B means that A «uses» B (e.g., a method invocation)

Custom types and exceptions

Custom Types:

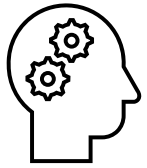
- Color
- NonNegativeInteger
- PositiveInteger
- Coordinates
- Buffer

Exceptions:

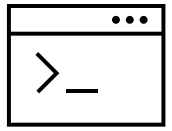
Set of model-specific exceptions:

- BoardIsFullException
- CellAlreadyOccupiedException
- CellOutOfBoardException
- GameNotEndedException
- …

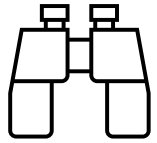
User Interface



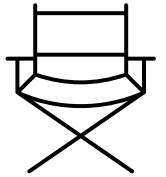
Model View ViewModel



CLI and GUI

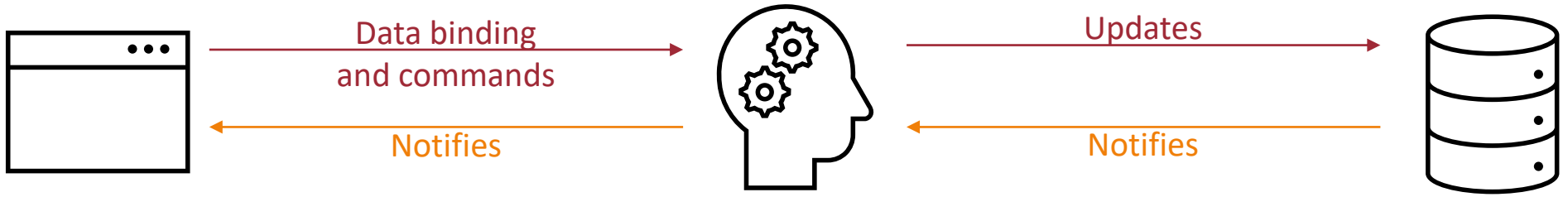


Observer Pattern



Scene Controller

Model-View-ViewModel (MVVM)



View

- Defines layout and appearance
- Platform dependent
- Observes the ViewModel for data binding
- Accepts commands from the user and forwards to the ViewModel

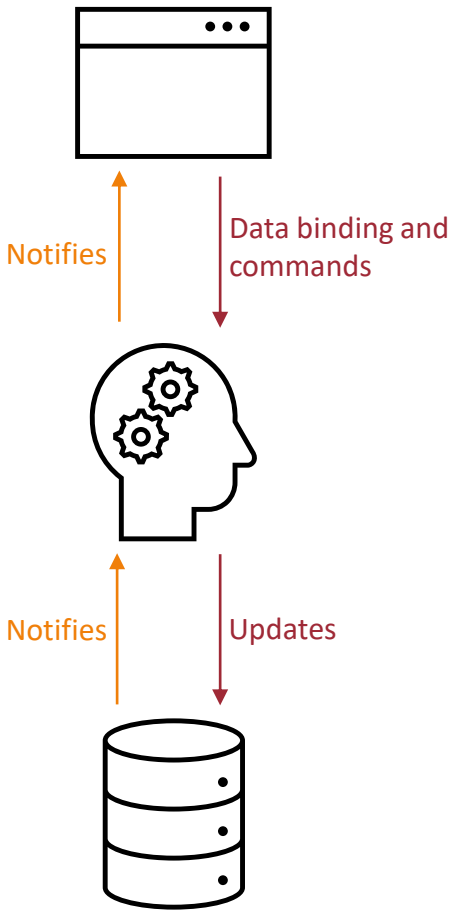
ViewModel

- Interfaces with the model
- Defines the functionality to be offered by the UI
- Implements properties and commands to which the view can data bind to
- Notifies the view for property changes

Model

- Set of non-visual classes that encapsulate the app's data and functionalities

Model-View-ViewModel (MVVM)



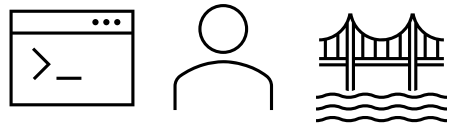
View

- StartView — collect setups operations
- MainView — shows the current board and allows to place stones
- SummaryView — summarizes the result of the game when it ends

ViewModel

- StartViewModel: — handles setup operations
- MainViewModel: — handles the progress of the entire match

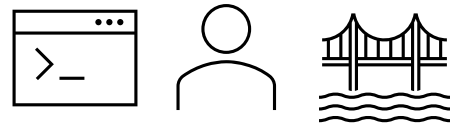
Model



CLI and GUI

- GUI
  - Graphic User Interface
  - Realized with JavaFX
  - Each view is associated to a markup FXML file

- CLI
  - Command Line Interface



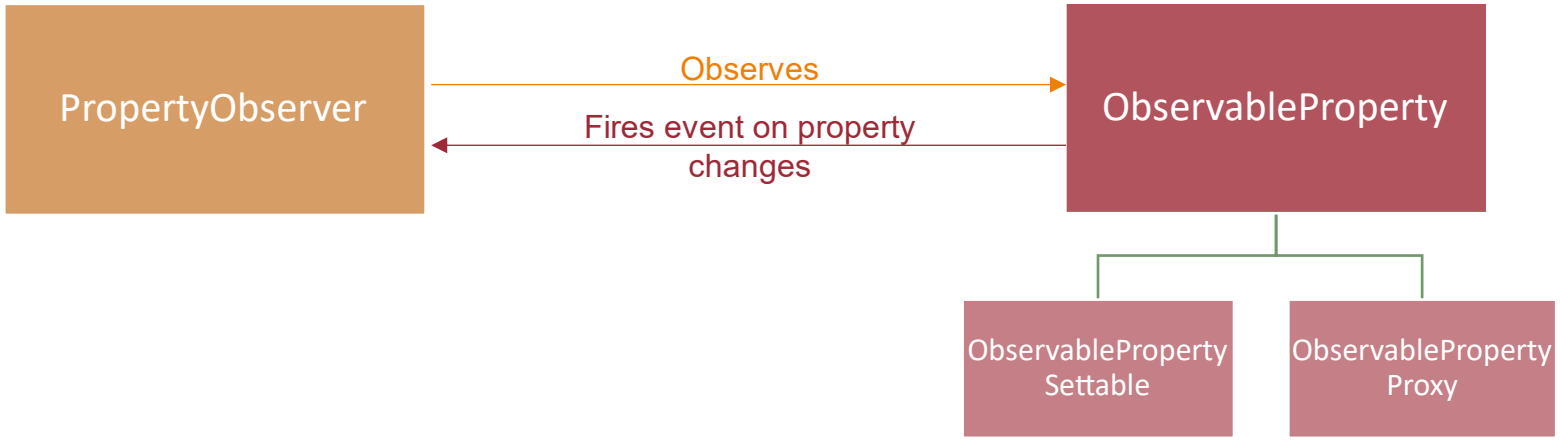
CLI and GUI

- MVVM allows to make the UI independent from the logic
  - CLI and GUI implement the views of the MVVM
  - Asynchronous property-change notifications
    - To keep the UI responsive
    - Event-based inter-thread communication

      → *Observer Pattern*

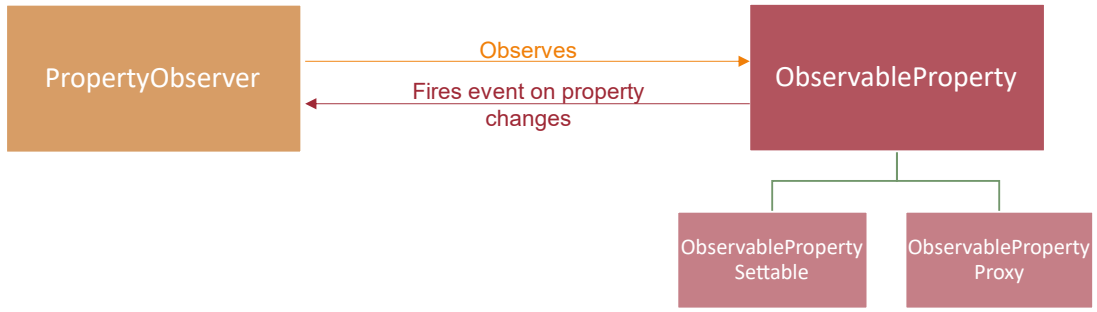
Observer pattern

- PropertyObserver observes ObservableProperty
  - A property observer observes exactly one observable property
  - When the observable property values changes
    - ObservableProperty fires a property-change event
    - only the corresponding PropertyObserver is notified and handles the change

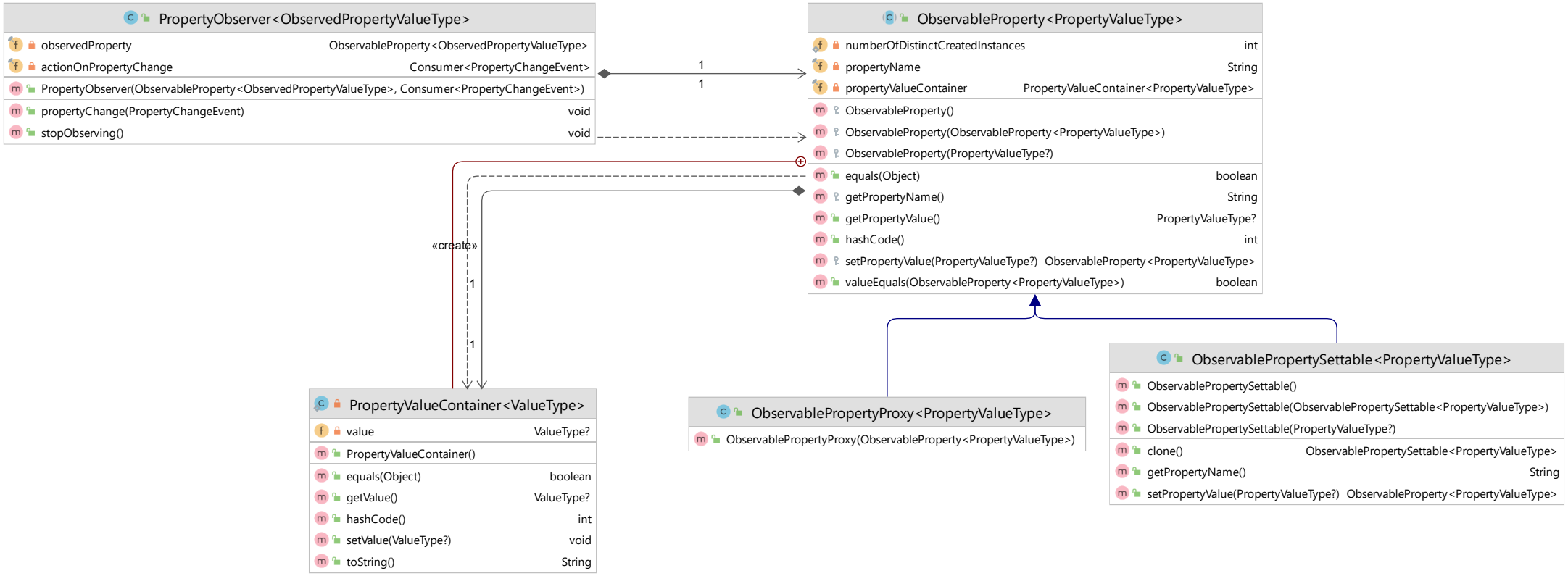


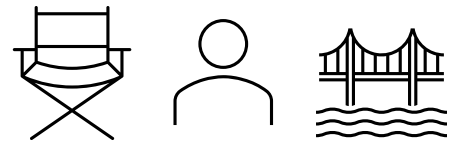
Observer pattern

- ObservableProperty
  - is the base abstract class
- ObservablePropertySettable
  - can read and set the property value and notify
- ObservablePropertyProxy
  - cannot neither set the value nor notify
  - can only read the the value
  - needed because the PropertyObserver
    - “contains” the ObservableProperty instance to be directly bound with it
    - must not be able to “use” the ObservableProperty instance
      - otherwise, the PropertyObserver would be able to set and notify property changes → *privacy leak*



Observer pattern

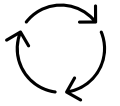




Scene controller

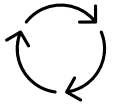
- Manages the navigation between views
  - Lazy initialization of views (scenes in JavaFX)

- Enforces the single responsibility principle
  - Navigation between views is
    - neither responsibility of the view itself
    - nor of the viewmodel



Project evolution

- Pair programming
- Test
- Continuous integration
- Growth direction

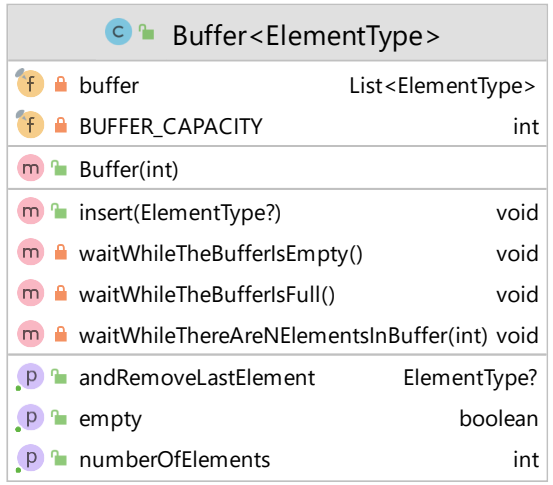


Pair programming

- Difficult and expensive but effective
- Both in presence and online
- “Trio” programming
  - Frequent rotation between drivers and navigators

Test

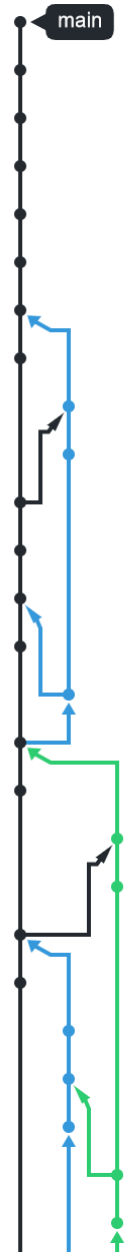
- Run before each commit
  - Helpful for discovering errors
- Junit
  - Testing framework
- TDD
  - e.g., class Buffer
- Property-based tests
- Test Doubles
  - Use of fake objects



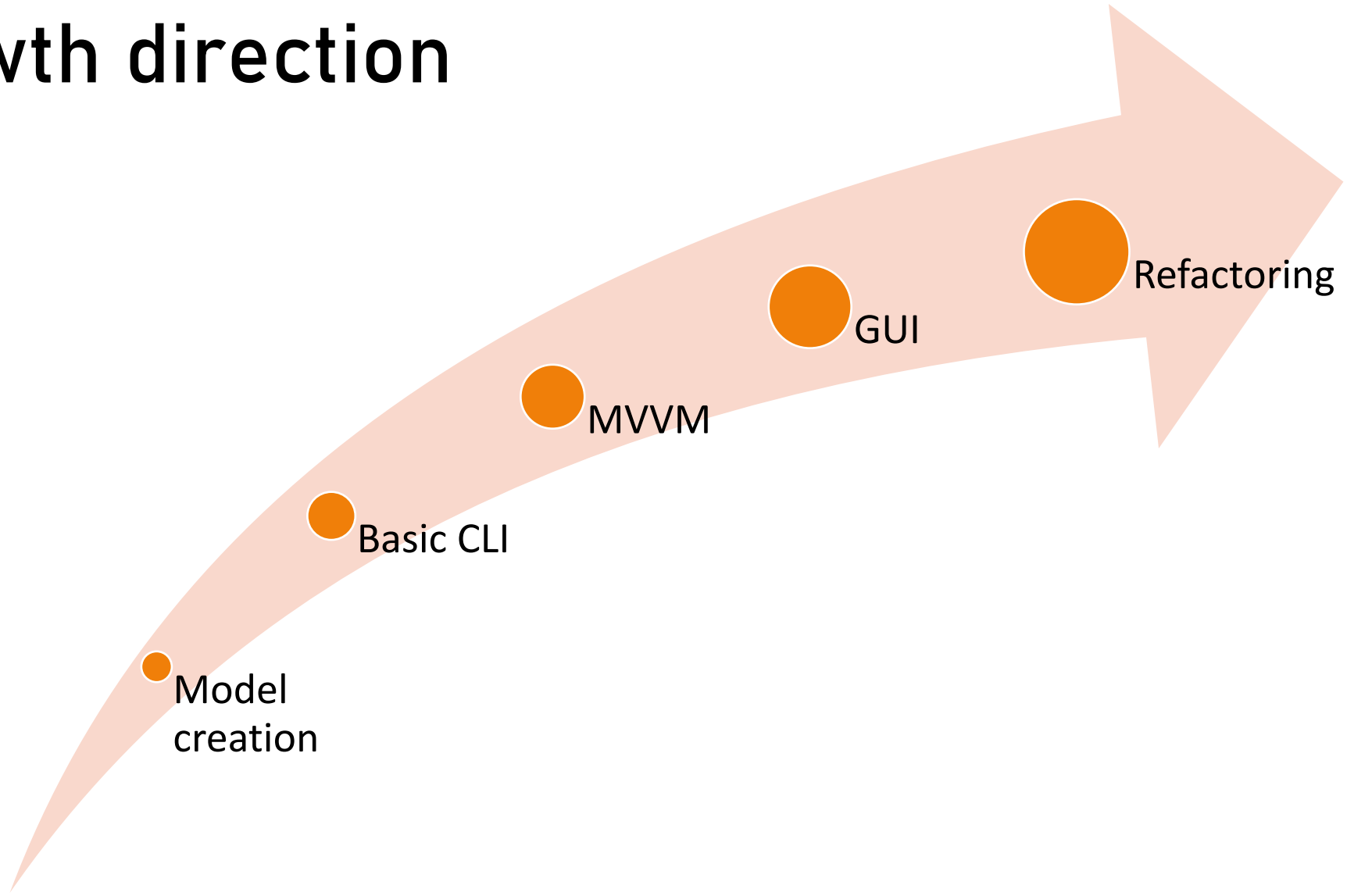
```
class BufferTest {

    @NotNull
    private Buffer<Integer> bufferOfIntegerUsedInTests;
                    . . .
    @BeforeEach
    void setUp() {
      bufferOfIntegerUsedInTests = new Buffer<>(ARBITRARY_CHOSEN_SIZE);
    }

    @ParameterizedTest
    @CsvFileSource(resources = EnvVariables.POSITIVE_INTS_LOWER_THAN_10000_PROVIDER_RESOURCE_LOCATION)
    void insertOneElementWhenBufferIsEmpty(int element) {
      assert isBufferEmpty(bufferOfIntegerUsedInTests);
      bufferOfIntegerUsedInTests.insert(element);
      final int NUMBER_OF_INSERTED_ELEMENTS = 1;
      assertEquals(NUMBER_OF_INSERTED_ELEMENTS, bufferOfIntegerUsedInTests.getNumberOfElements());
    }
                    . . .
    private <ElementType> boolean isBufferEmpty(@NotNull final Buffer<ElementType> buffer) {
      return Objects.requireNonNull(buffer).getNumberOfElements() == 0;
    }

}
```

Continuous integration

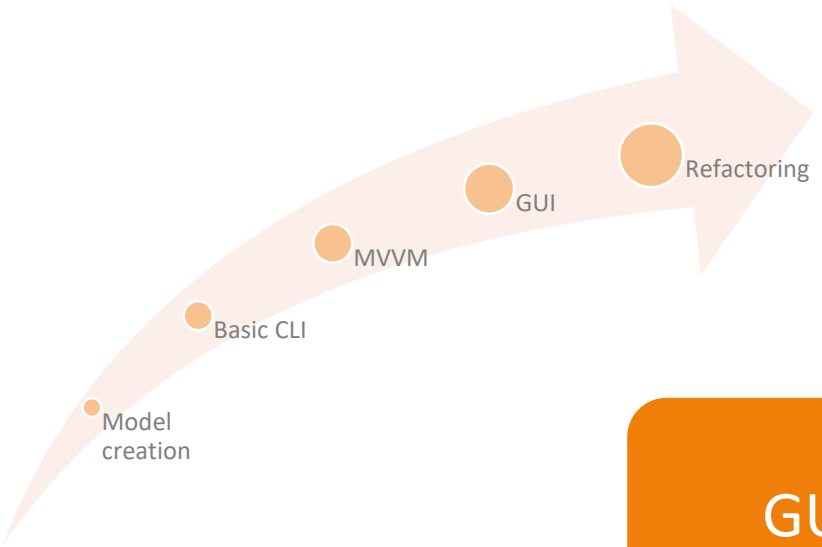
- Only one branch
- Growing the system
- Integration to the main branch as soon as possible
  - Merge and solve small conflicts often

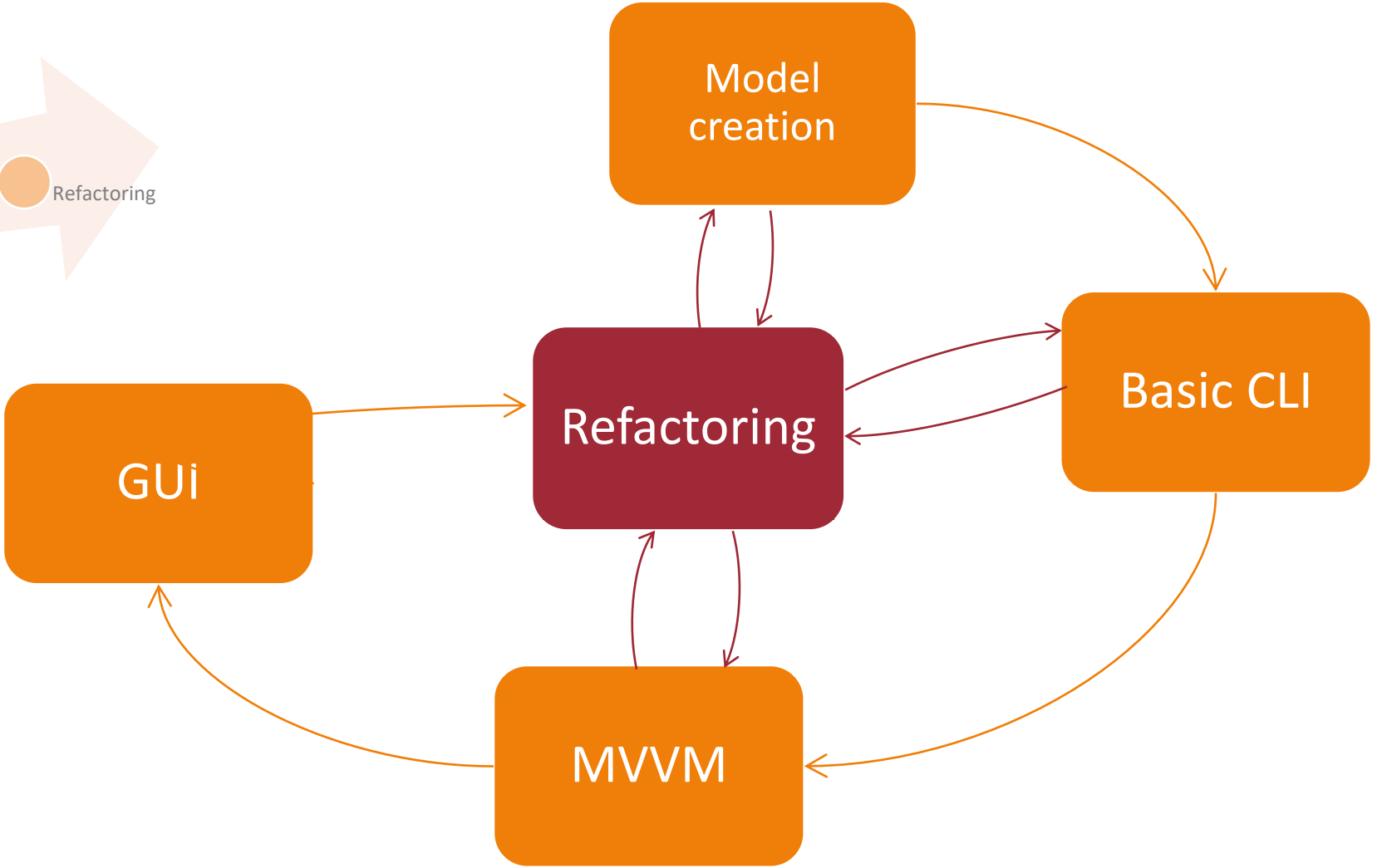


Growth direction



Growth direction





Continuous Refactoring

Team

Massimiliano Cristarella

Matteo Ferfoglia

Giovanni Marchetto

One for all, all for one!

Brainstorming, pair programming, continuous integration and TDD

Thank you