Programming Project

AlphaTargui

Contents

1 Introduction

In the course "*Programming of Large Object-Oriented Systems*" we have learned to focus on roles, responsibilities and behavior rather than data. CRC-diagrams and sequence diagrams are good ways to explore object interactions and to understand domain problems. Of course we cannot do without a class diagram.

We have also learned to develop code using test-driven development process. Thus we have produced a test list to support this process.

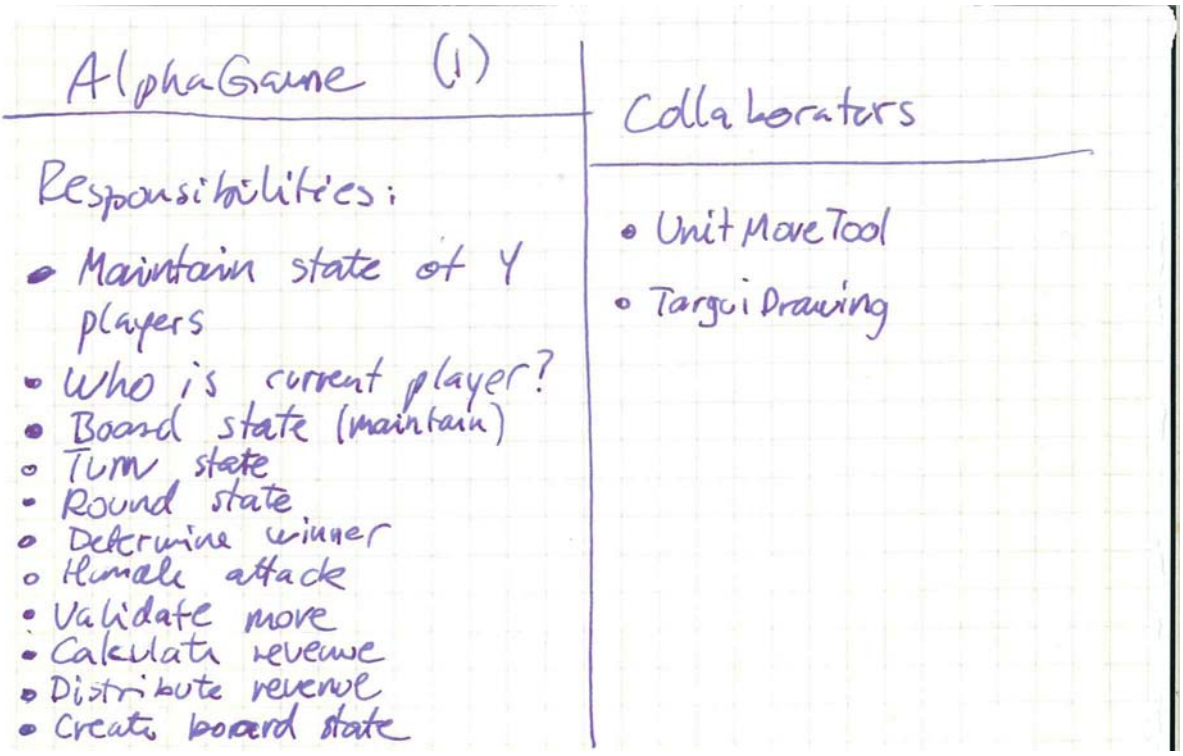
Finally we also wanted to get a better understanding of quality attributes.

To be continued…☺

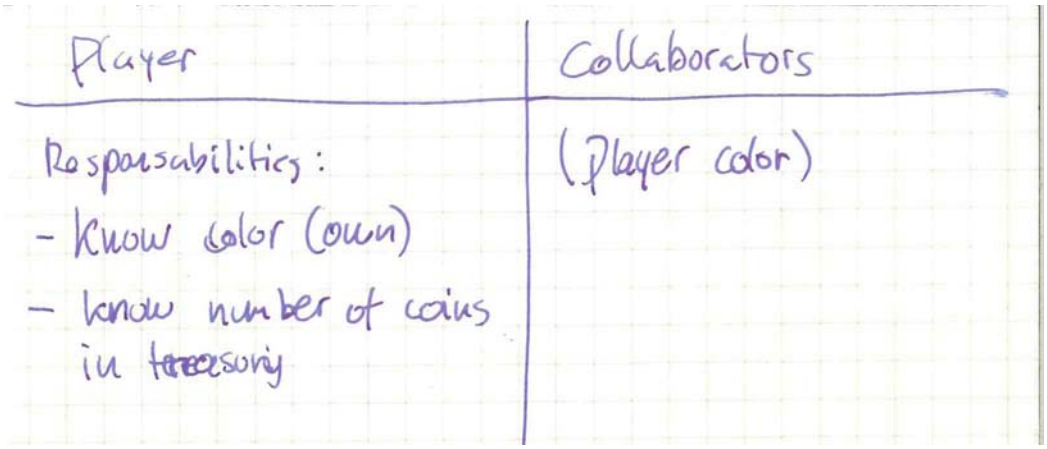
NB: This first version of the report suffer of lack of text

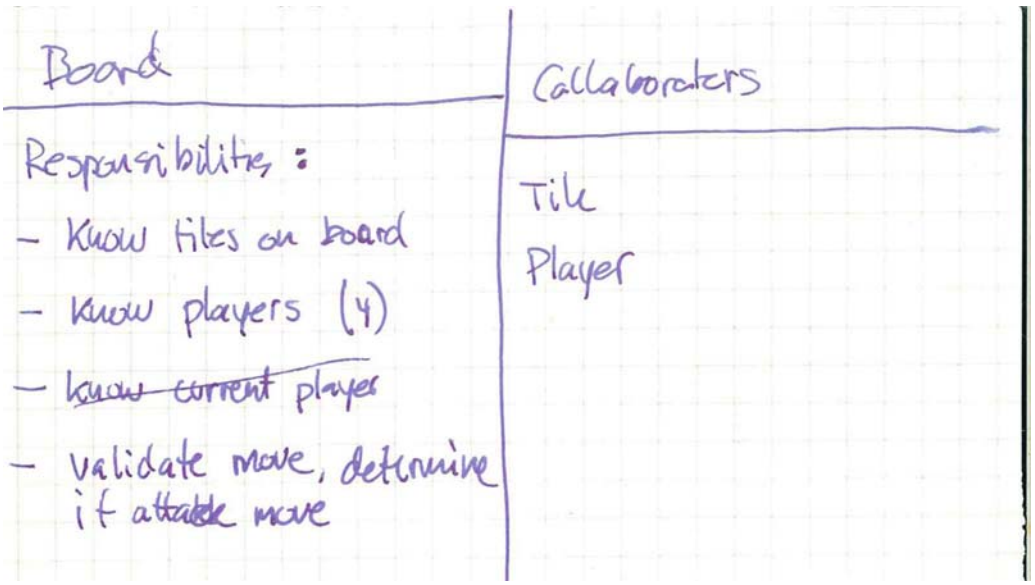
2 CRC-cards

1) Make AlphaGame CRC – determine responsibilities.



2) Too much responsibility in the AlphaGame. “Board” and “Player” separated out from AlphaGame because of complexity. Used OO-centered thinking – “The physical model is a simulation of the real world”.





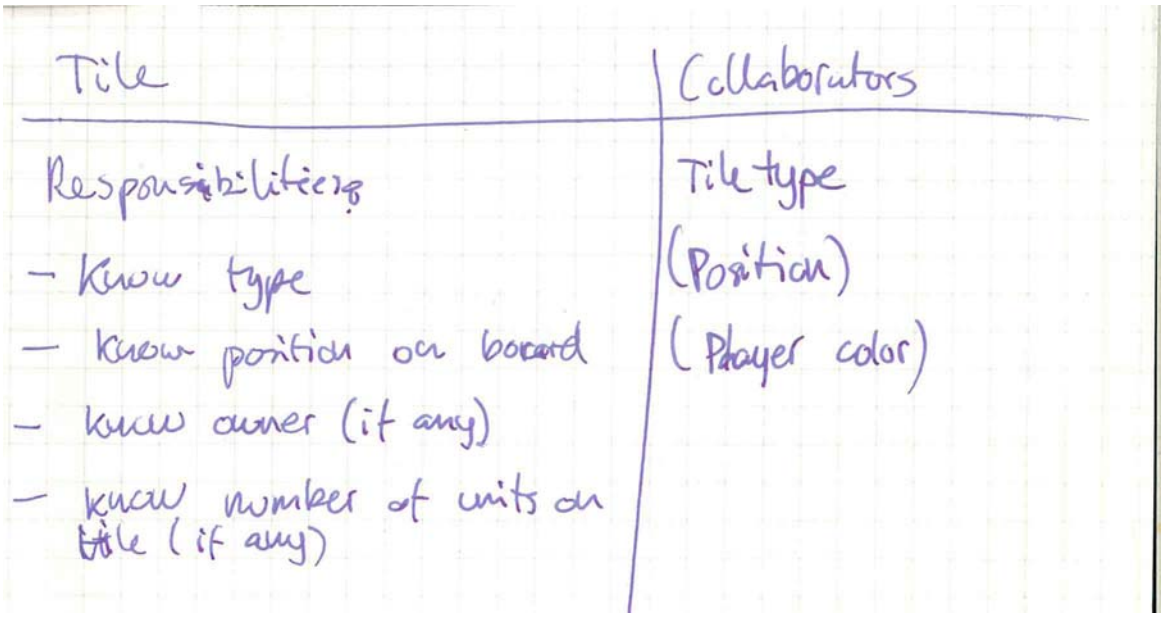
3) Distribute AlphaGame responsibilities onto the new collaborators

   a. State of the board → Delegated to “Board”.

   b. Validate move → Delegated to “Board” (has knowledge of tiles and players, can determine if move is valid).

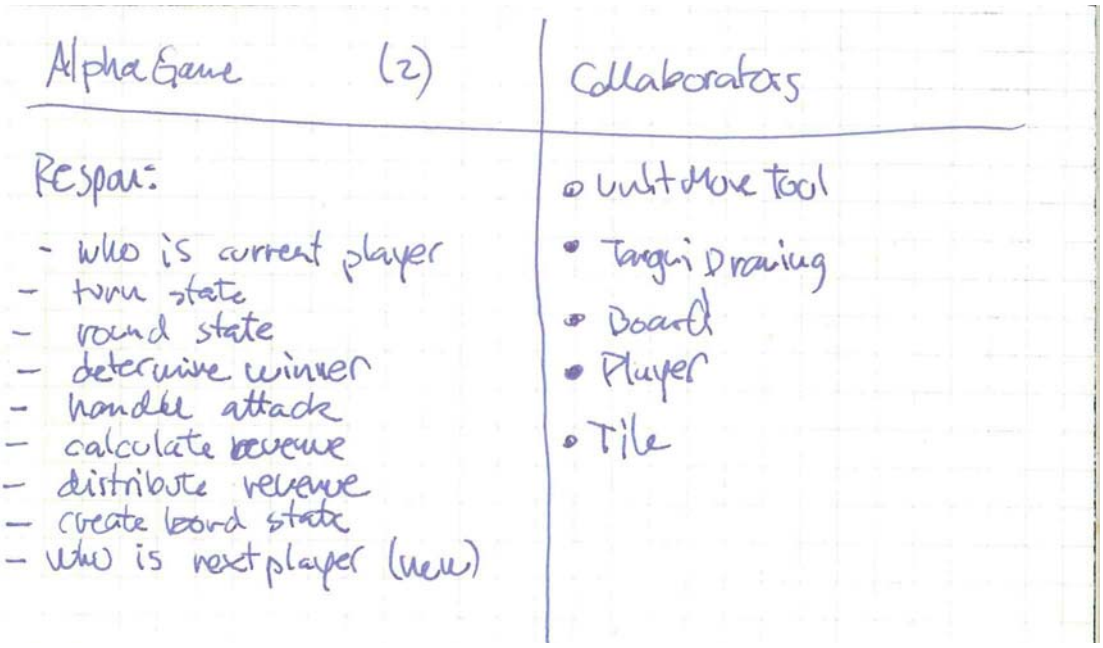
   c. Who is current player → “Board” know players.

4) Introduce “Tile”.



5) Move “currentPlayer()” back to “AlphaGame” from “Board”.

6) AlphaGame CRC update.


AlphaGame (2)
Collaborators
Respon:
- who is current player
- turn state
- round state
- determine winner
- handle attack
- calculate revenue
- distribute revenue
- create board state
- who is next player (new)
• UnitMoveTool
• TarguiDrawing
• Board
• Player
• Tile

3 Test-list

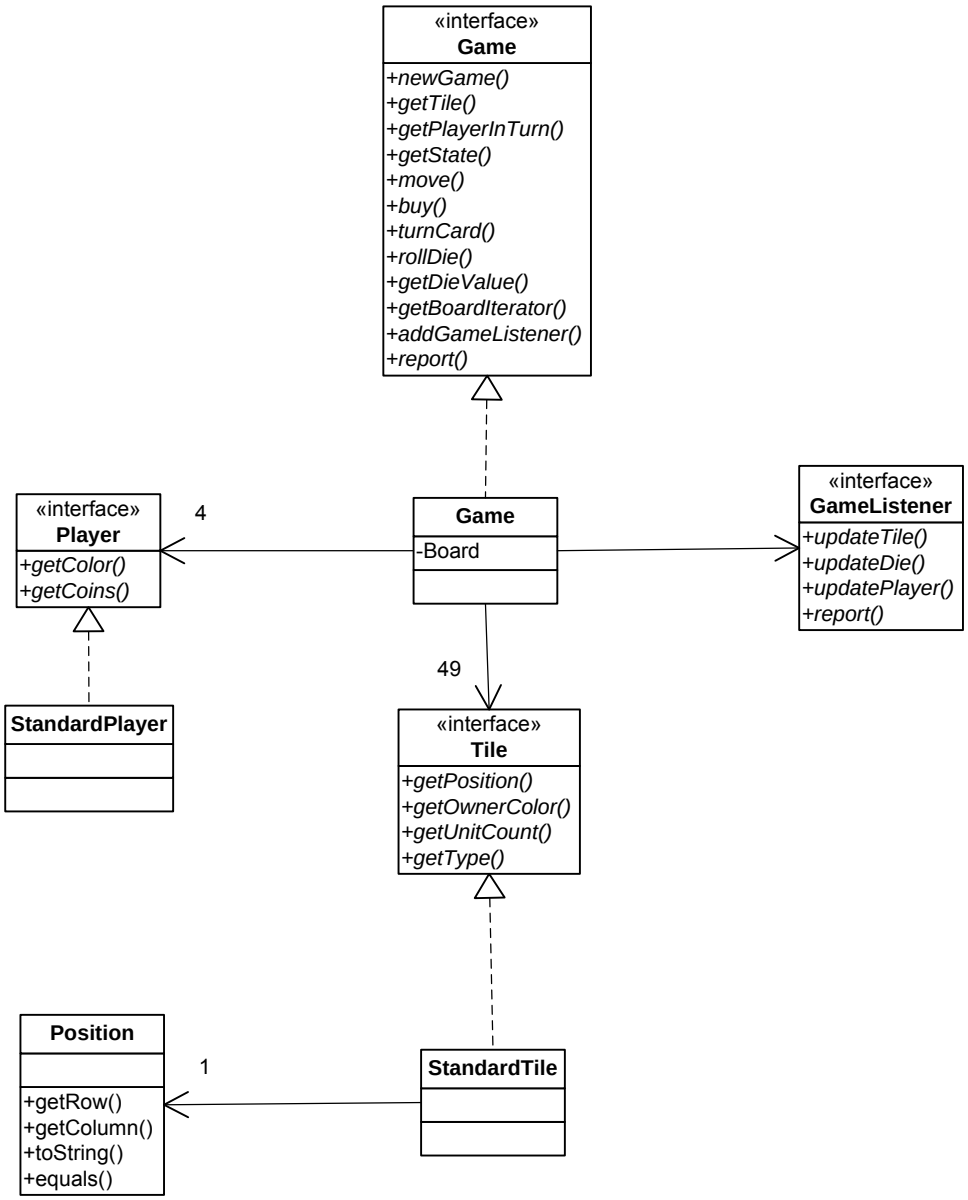
The following section contains a test-list, who describes the tests we have identified:

- Player count (obvious - hardcoded).
- Tile positions matches specification (review manually).
- Round (red first, green second, blue third, yellow fourth, red first).
- Turn (expect move first, expect buy second).
- Move
    - Valid: Move to un-occupied tile.
    - Valid: Move to occupied tile where there is no camels.
    - Valid: Move to each of the tile types with no camels.
    - Invalid: Move with other player camels.
    - Invalid: Move without camels.
    - Invalid: Move to the “Salt Lake”.
    - Invalid: Move outside the board (cannot be tested as the GUI is unable to make this move. Preconditions should be noted, so this test can be avoided).
    - Invalid: Move more than one tile in a turn.
    - Attack: Attacker has less camels than defender (attacker is defeated).
    - Attack: Attacker and defender has equal numbers of camels (status quo).
    - Attack: Attacker has more camels than defender (defender is defeated).
- Revenues
    - Calculate revenue for all players.
    - Calculate revenue, where one or more players have died.
    - Calculate revenue, where one or more players have lost own settlement.
    - Sums revenue correctly.

Assumptions: We assume that having a settlement is enough to get revenue compared to having own settlement.

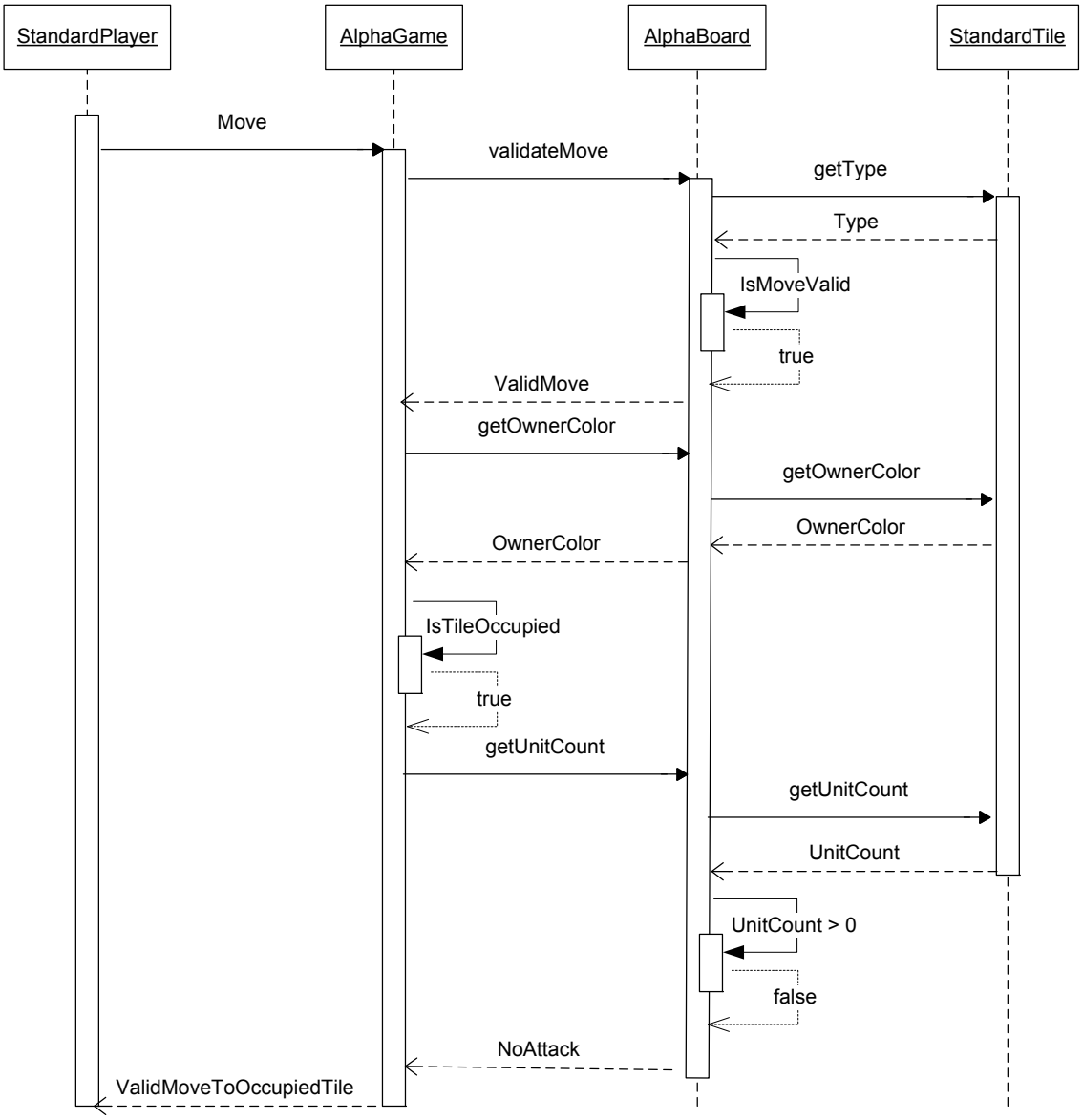
- Determine winner: Play 25 rounds, one must own “Salt Mine”, determine winner.

4 Class diagram for initial version of the game

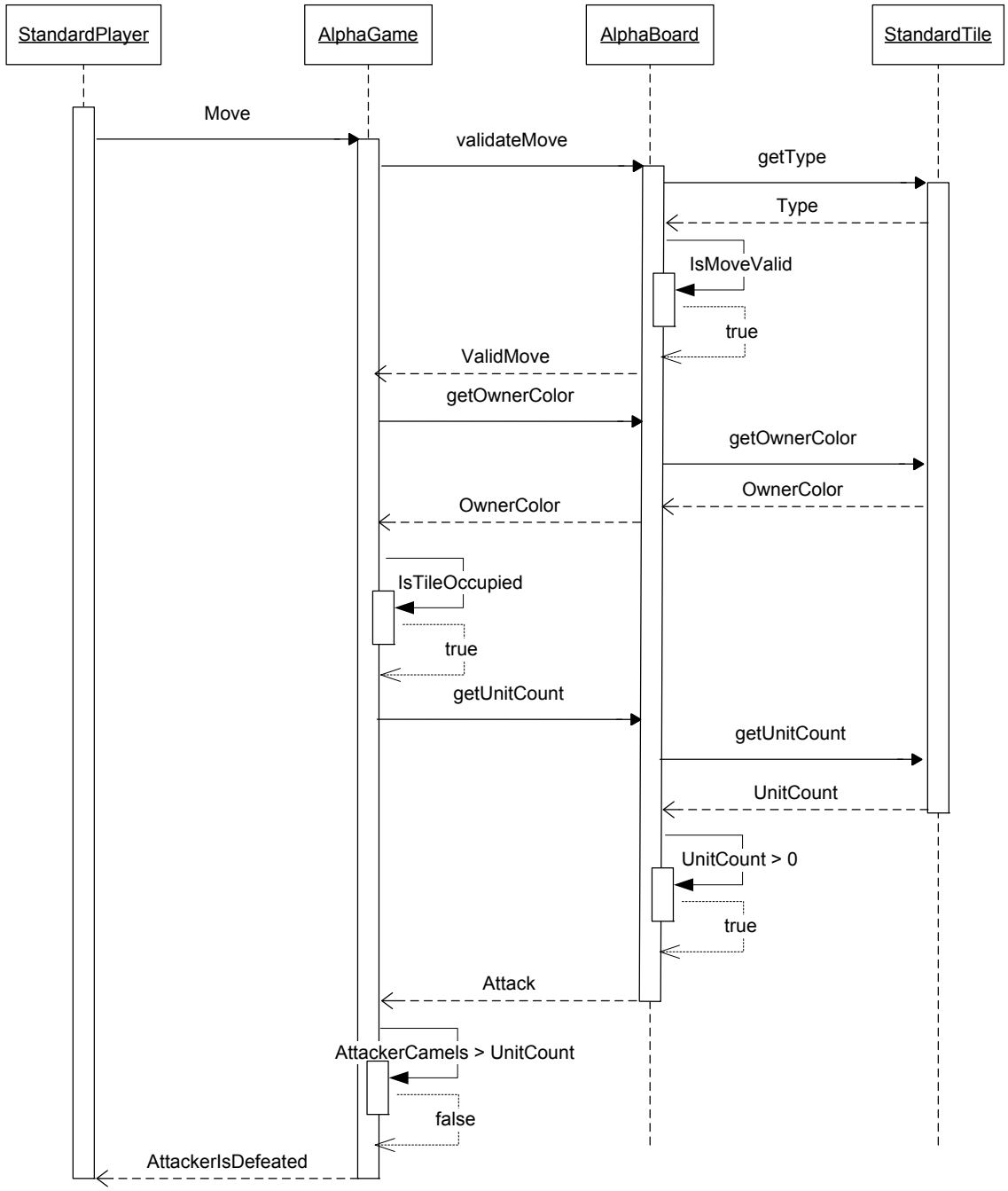


5 Sequence diagram for “Moves”

5.1 Valid move to occupied tile with no camels



5.2 Attacker has less camels than defender (Attacker is defeated)



6 Software Quality Attributes for the implementation of HotTargui

6.1 Modifiability Scenario

HotTargui can be played by 2 to 4 players depending on the game variant. Thus we have found something that varies in our system. It shall be easy to modify the software to implement the different variants. This means that the modifiability quality attributes is very important for our system.

Portions of scenario	**Values**
Source	Developer.
Stimulus	Wish to make a new game variant
Artifact	Code
Environment	Design Time.
Response	The modification is made with no side effects.
Response measure	The time to make the modification should be less than 3 days.

6.2 Testability Scenario

It is mandatory that code is developed using test-driven development process. Thus testability quality attributes is also very important for our system.

Portions of scenario	**Values**
Source	Unit developer
Stimulus	Design (writing code)
Artifact	Piece of code (Component of the system)
Environment	At development time
Response	Unit can be controlled and its responses captured
Response measure	Green bar (or red bar)

6.3 Usability Scenario

Since we are in the gaming world usability is important. To become popular, a game should be easy to learn and easy to use. To support “use system efficiently”:

Portions of scenario	**Values**
Source	End user
Stimulus	Wish to learn to use the system
Artifact	The system (game)
Environment	Normal operation at runtime
Response	User gets instructions that he/she can read
Response measure	After 30 minutes the user is familiar with playing the game

6.4 Performance Scenario

Performance is often an important part of usability. The system must respond requests from user (mouse click or key press) within a reasonable time.

Portions of scenario	**Values**
Source	Users
Stimulus	Move, attack or buy units
Artifact	The system (game)
Environment	Normal operation at runtime
Response	Move, attack or buy units are processed by the system.
Response measure	Within 2 seconds

6.5 Availability Scenario

Notifying end user about faults is also a parameter to improve usability. In our case a restart of the system (game) would probably solve all faults. Thus the availability quality attributes are not very important for our system.

Portions of scenario	**Values**
Source	User input
Stimulus	A component fails to respond to an input
Artifact	The system
Environment	Normal operation
Response	Notify end user about the problem.
Response measure	Restart of system must only take 5 seconds

6.6 Security Scenario

Not important for our system and therefore there are no description of this scenario.

7 Test-driven development

7.1 Iteration 1: Iterating tiles on Board

1. Add test case ‘testTileIterationCount’

2. Compile error – no implementation

3. Create getBoardIterator method on Board – fake-it, return null

4. Unit test fails – null reference exception

5. AlphaBoardFactory return correct array of tiles – Fake-it implementation of Tile interface used (anonymous inner class implementation of Tile with exception throwing methods)

6. Test cases succeed

7. Refactoring – no change

7.2 Iteration 2: Validating Board layout

1. Add test case ‘testTileIterationLayout’

2. Unit test fails – unimplemented method exception

3. Triangulate with Iteration 2 and 3 and use Factory pattern and create BoardFactory interface which may create the tile layout.

4. Same unit test fail (no extra errors after update)

5. Update Board to use BoardFactory interface to create Tile layout during construction

6. Compile error – Board construction fails.

7. Update Unit test to Take an instance of BoardFactory called AlphaBoardFactory with Fake-it implementation.

8. Unit test fail as Board constructor fails with null reference exception.

9. Implement AlphaBoardFactory with code from Iteration 2 – simply create collection of Fake-it tiles.

10. Same unit-test as first now fails (back to square one).

11. Investigation of the StandardTile implementation reveals that it is sufficient. This tile-type is used instead with the correct setup.

12. Test cases succeed

13a. Refactoring: Move validation code to separate utility method to save test space.

13b. Move utility method which extracts a specific Tile based on its position (used in unit test) to the Board class, as it will be needed elsewhere.

13c. Update unit test to use this method rather than it own.

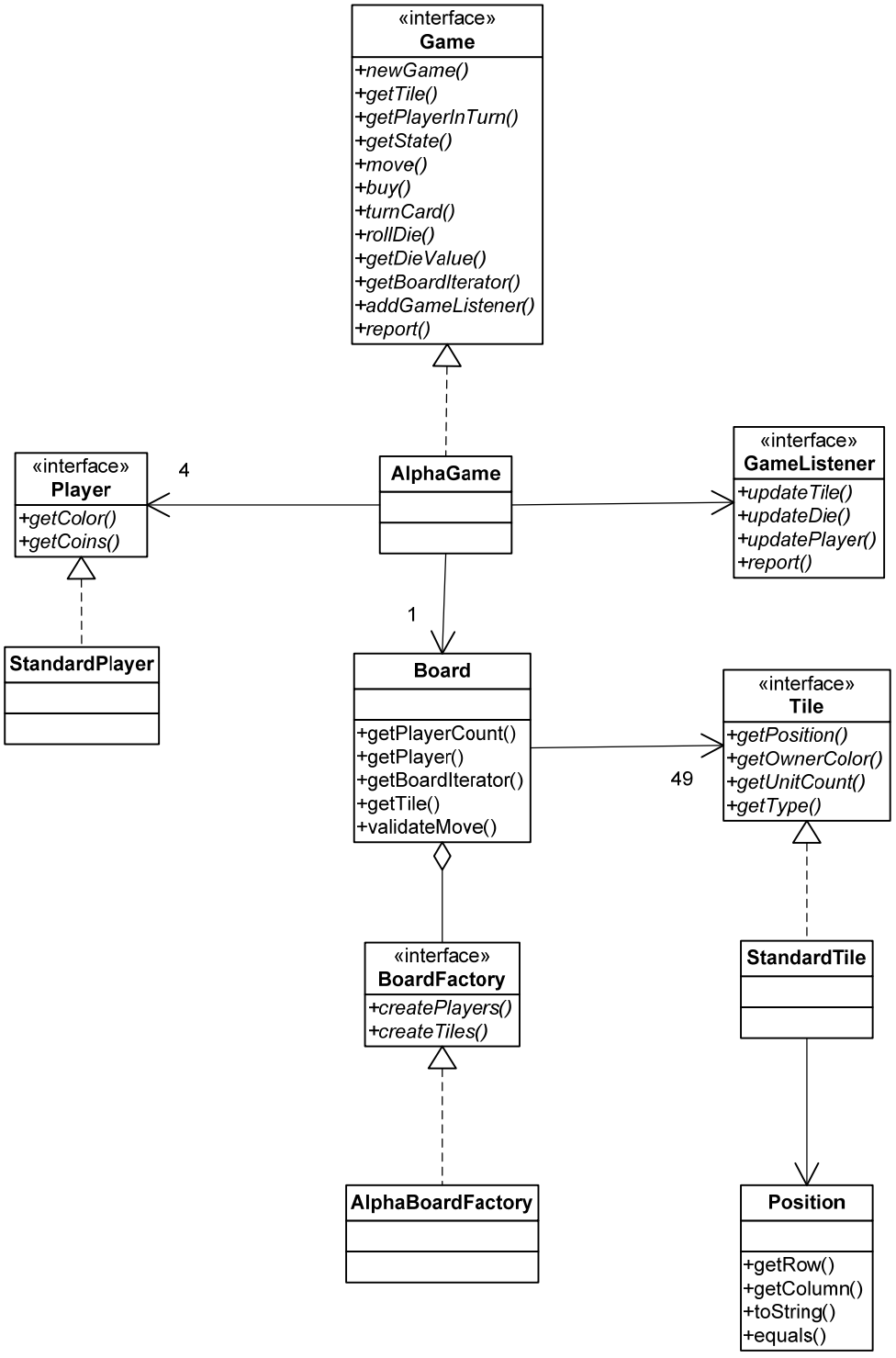
14. Extend with test of invalid position (one that does not exist) “testFindInvalidPosition’ – Here the Equivalence Classes suggests testing above and below (negative and above 6), and as the row/column is an integer, then it is possible to test. Whether it should be tested may be debated, as one may simply stipulate that the Position column and row must always lie in the range 0-6.

15. Test case failed – null returned when expecting exception.

16. Update code so ‘IllegalArgumentException’ is thrown if Position not found.

17. Test cases succeed

8 Class diagram for AlphaTargui



9 TODO

This section contains our TODO's:

- Make CRC-cards to place responsibility.
- Test-list
- Template.
- Class diagram for AlphaTargui.
- Sequence diagram for "Moves".
- Software Quality Attributes for the implementation.
- Start test-driven development.