

**Gruppe 9**

Anders Hvidgaard Poder, 19951439

Elund Christensen, 20074530

Kewin Peltonen, 20054669

Programming Project

HotTargui

1.0 D

07-04-2008



1 (51)

**Programming Project**

HotTargui

## Contents

<b>1 Introduction .....</b>	<b>3</b>
<b>2 AlphaTargui .....</b>	<b>3</b>
2.1 Initial class diagram .....	4
2.2 CRC-cards .....	4
2.3 Test-list .....	7
2.4 Test-driven development .....	8
2.4.1 Test Player Count .....	9
2.4.2 Iterate Players on board .....	10
2.4.3 Iterate tiles on board .....	12
2.4.4 Player turn ordering .....	14
2.4.5 Player Turn .....	15
2.4.6 Move .....	16
2.4.7 Attack .....	22
2.4.8 Buy .....	25
2.4.9 Revenue .....	26
2.4.10 Winner .....	31
2.5 Conclusion .....	33
<b>3 Systematic Test .....</b>	<b>35</b>
3.1 Introduction .....	35
3.2 Finding partition sets - distance .....	35
3.3 Result of attack .....	37
3.4 Equivalence classes .....	39
3.5 Test cases .....	39
3.6 Boundary analysis .....	42
<b>4 BetaTargui .....</b>	<b>43</b>
4.1 Introduction .....	43
4.2 Variability points .....	43
4.3 Class diagram for BetaTargui .....	45
<b>5 DeltaTargui .....</b>	<b>46</b>
5.1 New requirement: Randomized board .....	46
5.1.1 Analysis .....	46
5.1.2 Design choices .....	46
5.1.3 Position saltmine .....	47
5.1.4 Position tiles randomly on board .....	47
5.1.5 Position settlements .....	50
5.1.6 Conclusion .....	51
5.2 New requirement: Determine winner .....	52
5.2.1 Analysis .....	52

## 1 Introduction

As this is an academic report its focus is more on the learning objectives and process of development than on the finished project. For this reason the report may contain sections praising one approach, only to change it in the next section. In other words the report is an ongoing process, and conclusions drawn in previous sections may not be changes as the report progresses.

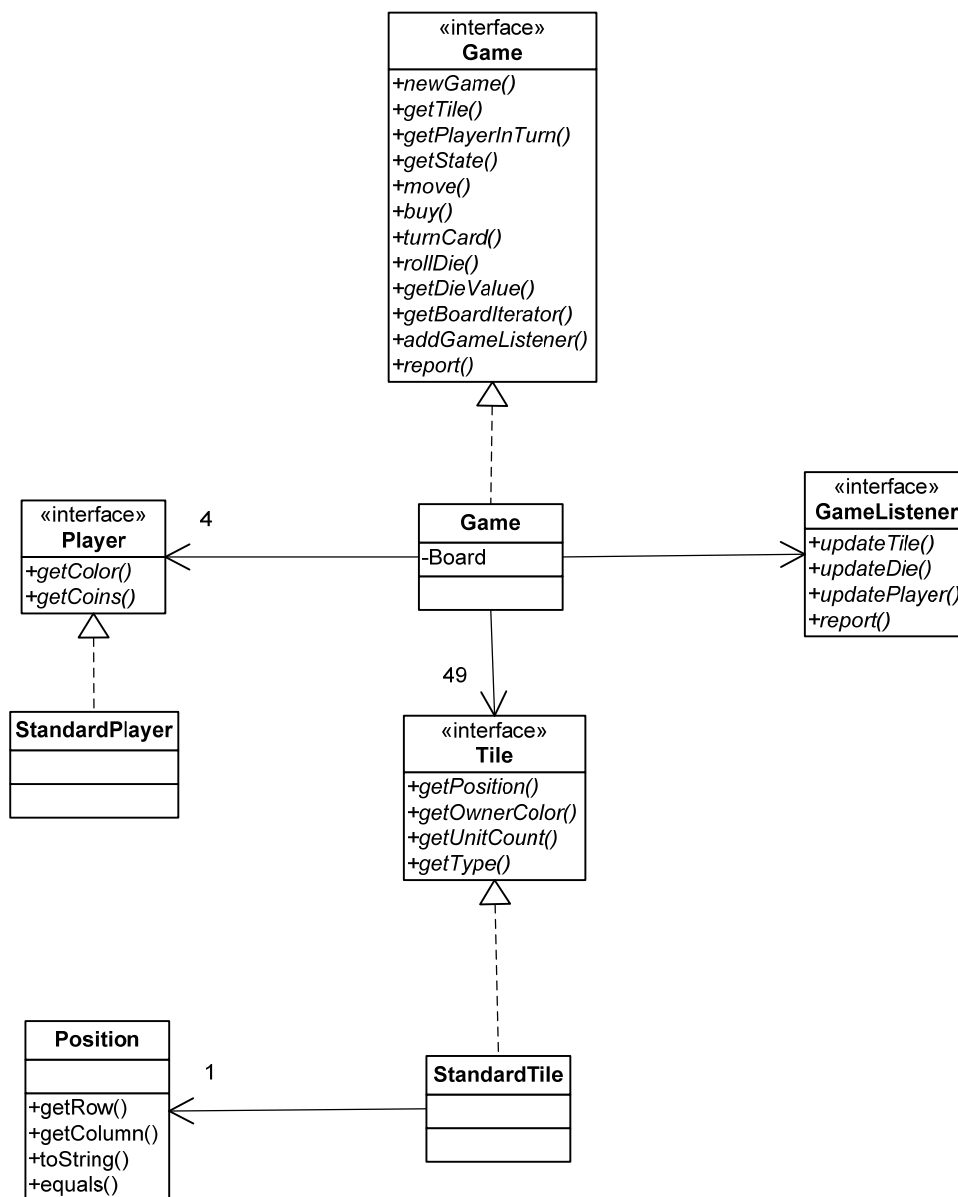
## 2 AlphaTargui

This product was begun at the first seminar, where we had to determine not only what to produce, but also how to break down the project into manageable pieces and figure out how to work geographically separated.

Test Driven Development is a very quick-decision process, and it is therefore exceptionally difficult to do when the development is performed not only separated in space, but also in time. For this reason we decided to use our time together to define the responsibilities of the different modules of the product, based on the framework handed out and our understanding of what needs to be implemented.

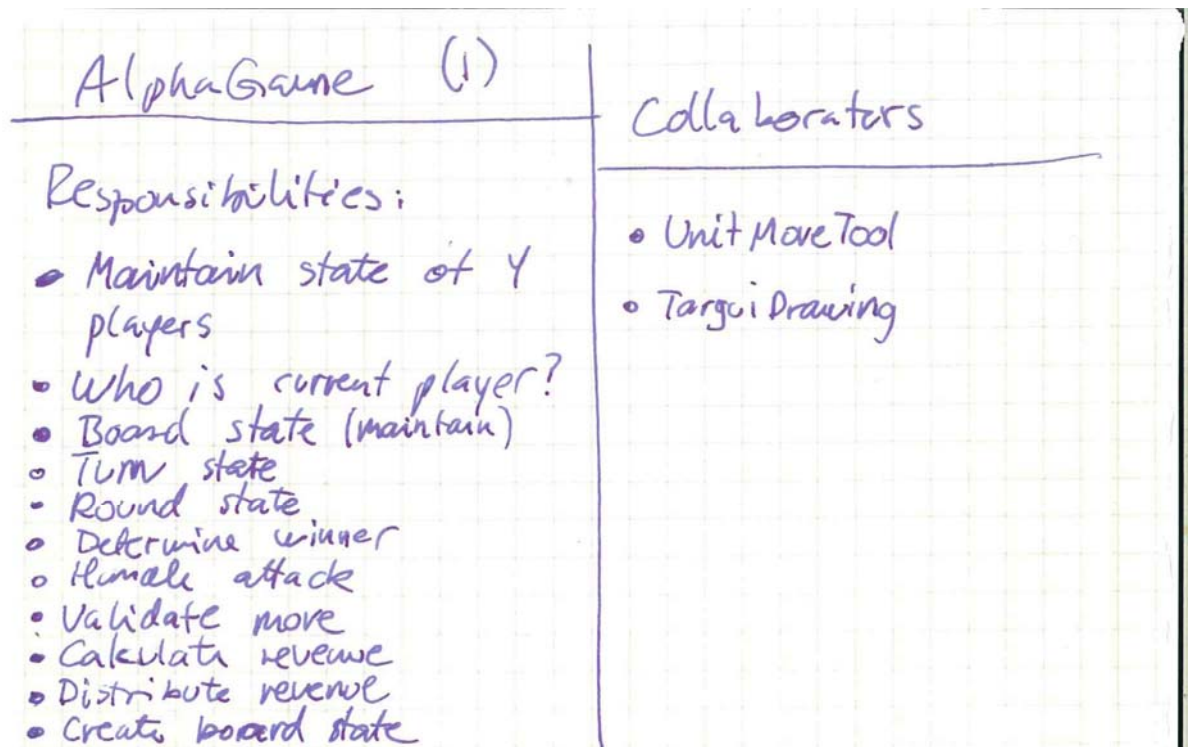
Defining responsibilities may be done in many different ways, yet as we have learned in the course "*Programming of Large Object-Oriented Systems*" it is beneficial to focus on roles, responsibilities and behaviour rather than data, which may be achieved using CRC-cards, so this is what we did. In order to determine the different interfaces/classes involved in the framework and their dependencies, we wrote a quick class diagram of the framework, which was later transcribed to electronic form to preserve it, yet for use with the CRC-cards it was simply a hand-written sketch.

## 2.1 Initial class diagram

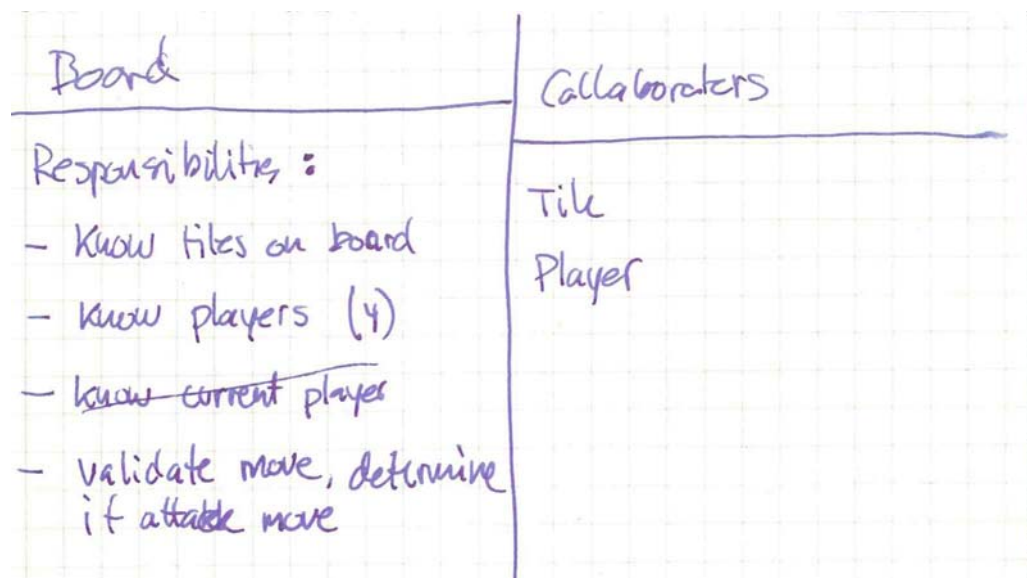


## 2.2 CRC-cards

We first defined the AlphaGame's responsibilities, yet quickly determined that it had way too many responsibilities and would quickly become "the blob".



For this reason we decided to split the *Game* implementation into two; one focusing on the static behaviour (the board), and one focusing on the dynamic behaviour (the *Game*). This resulted in the following responsibility for the *Board*.



As it may be seen some of the responsibility has been moved to the *Board*.

- State of the board → Delegated to "Board".
- Validate move → Delegated to "Board" (has knowledge of tiles and players, can determine if move is valid).

This changes the *AlphaGame* class implementation to

AlphaGame (2)	Collaborators
Respons: <ul style="list-style-type: none"> <li>- who is current player</li> <li>- turn state</li> <li>- round state</li> <li>- determine winner</li> <li>- handle attack</li> <li>- calculate revenue</li> <li>- distribute revenue</li> <li>- create board state</li> <li>- who is next player (new)</li> </ul>	<ul style="list-style-type: none"> <li>• Unit Move Tool</li> <li>• Targui Drawing</li> <li>• Board</li> <li>• Player</li> <li>• Tile</li> </ul>

The remaining interfaces have not been altered, and we only defined them in order to ensure that no responsibility was forgotten.

Player	Collaborators
Responsibilities: <ul style="list-style-type: none"> <li>- know color (own)</li> <li>- know number of coins in treasury</li> </ul>	(Player color)

Tile	Collaborators
Responsibilities: <ul style="list-style-type: none"> <li>- know type</li> <li>- know position on board</li> <li>- know owner (if any)</li> <li>- know number of units on tile (if any)</li> </ul>	Tile type (Position) (Player color)

Before beginning the actual development process, we also discussed quality attributes and it was determined that there are many important quality attributes, yet when we look at the AlphaTargui game by itself, with no GUI support and no knowledge of products to

come, there is actually only one quality attribute of any real importance, and that is testability, which fits well with the test driven development. Furthermore, as we all know, servicing the testability quality attribute means programming to an interface, in order to allow unit testing where dependent units are stubbed away. It also means opening up some of the state information of the class for testing, which might otherwise have been hidden. This means walking that fine line between testability and information hiding, for even though we focus on testability, this does not mean simply opening up all private data to external modification. We have to remember that the design/framework may have to be used by others, and if a private member can be modified, someone is bound to modify it, even if it is not suppose too and no matter how many doxygen warnings are written to this affect.

As with all test it is important to know the requirements, as they are the bases of the tests. Test Driven Development specifies that it is a good idea to begin with an overall test-list, which simply states the areas that needs to be tested.

We did this also in plenum and came up with a test-list which may be a little more detailed than is the initial intention of TDD, yet as we are to perform the TDD as a distributed effort it helped to ensure a more unilateral approach. This also showed the first ambiguity, as we discovered that we could not determine if it was required to own the settlement that was originally allocated to the player, or if it was enough to simply own a settlement.

## 2.3 Test-list

- *Player* count. (4)
- *Tile* positions matches specification.
- Round (red first, green second, blue third, yellow fourth, red first).
- Turn (expect move first, expect buy second).
- Move
  - Valid: Move to unoccupied tile.
  - Valid: Move to occupied tile where there are 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 (tested during test of Turn)
  - Attack: Attacker has fewer camels than defender (attacker is defeated).
  - Attack: Attacker and defender have 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.

After writing the overall test-list we went our separate ways, and the following parts of the AlphaTargui was written by individual contributions.

## 2.4 Test-driven development

Before beginning the actual TDD we had a look at the existing implementations of *Tile* and *Player*, and it was determined that they may be used as-is; at least until a situation arise where it is no longer beneficial – there is no reason to re-invent the wheel.

The existing interfaces (*Tile*, *Player*, *Game* and the value-types they need) will be preserved as much as possible. This is based on the belief that the interfaces was designed for a reason and not simply thrown together. The value types, e.g. *Position*, are believed so simple, and already validated, that they are used without further testing.

Test Driven Development is designed to follow a rhythm with five overall stages; add test; see that the new test fail, but not the previous; make a little change; see that all the test now succeed; refactor to ensure good code.

This rhythm will naturally also be followed in this project, though as it may be seen the small changes can result in extensive refactoring in order to ensure good code.

Test Driven Development is also designed on the premise that it is possible to start with any feature and begin the implementation, but it is required that the feature chosen is implemented fully.



We decide to start with the player count. This is part of the static behaviour and is therefore part of the *Board*. Even though it is not required to create an interface to specify the protocol between the *Game* implementation and the *Board*, from a testing point of view it is a very good idea; programming to an interface.

For this reason the *Board* interface is defined and given the static behaviour from the *Game* interface. This may be done one method at a time or all at once. We have decided to let the interface grow.

In order to service the existing GUI implementation it has been decided not to remove the methods from the *Game* interface, yet simply use the *Game* as a *Façade pattern* forwarding all calls of a static nature to the *Board* interface.

As *Board* is an interface it is naturally not testable, but it does not mean we cannot use TDD, as will be shown in the first iteration, as shown below.

#### 2.4.1 Test *Player* Count

1. Add test case *playerCountEqualsFour*
2. Test case fails to compile as there is no such method.
3. Implement the method *getPlayerCount* in the interface.
4. Test case fail (there is no implementation only an interface).
5. Create class *AlphaBoard* and implement method (obvious implementation, simply return 4)
6. Test case succeeds.
7. Refactoring – No refactoring needed

Already here it may be seen that the rhythm is not quite followed, as the implementation phase requires two steps in order to make the tests pass. As the sub-iterations are very small and the focus is never lost, nor is the overview, it is acceptable.

Another important aspect here is actually the test itself. Tests may be both white-box and black-box, and as TDD is an iterative development process it follows that the implementation is known. For this reason it must therefore be decided if the type that is used in the test-cases is the specific type (*AlphaBoard*) or the interface *Board*. It has been decided to ensure a clear interface that the tests shall as default know only the interface, and then if a more special access is required, cast the instance as needed. Such casting is not required in the first test-case.

As we have now begun the *Player* access implementation we should finish it, and we therefore continue with accessing the actual *Players*. We are here faced with a decision; should the *Board* know anything about the ordering of the players? (The order in which they play their turns.) This is a very good question. It would be very easy for the *Board* to simply apply significance to the ordering in which it returns the *Players*, yet this is an

issue of responsibility-creep, where extra responsibility creeps in without is being actively chosen. The CRC-cards says nothing about the *Board* knowing whose turn it is or the order of turns – that is dynamic behaviour.

For this reason the interface is created to be independent of the ordering, and the user of the *Board* must request the player of interest. It is here important to note that a potential problem is discovered; what if a player is dead? Should null be returned, is an *isAlive* method required? Etc. This is a not an issue we wish to address now, but at the same time it must be remembered, and a test *getDeadPlayerFails* is added to the test-list for later implementation (and potential name updating).

#### 2.4.2 Iterate Players on board

1. Create test *redPlayerExistsInGame*.
2. Test case fails to compile as there is no such method.
3. Implement the method *getPlayer(PlayerColor)* in the interface and in the class (Fake-it, simply return Dummy *Player* no matter what).
4. Test case succeeds.
5. Refactoring – No refactoring required.

This test case naturally do not test that the *Player* returned is correct, so we add another test case.

1. Create test *redPlayerHasRedPlayerColour*.
2. Test case fails.
3. Update the method to return a *StandardPlayer(PlayerColor.Red)* – also a Fake-it implementation.
4. Test case succeeds.
5. Refactoring – No refactoring required.

We still have a fake-it implementation so obviously we are not done. We therefore add the method *greenPlayerHasGreenPlayerColour*, as the *greenPlayerExists* is indirectly tested in this and do not give anything new.

1. Create test *greenPlayerHasGreenPlayerColour*.
2. Test case fails (returns same *Player* as for Red)
3. We are now faced with the option of using the parametric approach and simply introduce an “if”. This is a quick and easily implementable solution, yet it also has some disadvantages. It does not grow well, it opens up for a potential switch-creep (or if-creep in this case) and as the Java framework introduces a much

more elegant solution and it is chosen here. A *Dictionary<PlayerColor, Player>* will allow us to perform a fast and type-safe (due to the generics) look-up. The Dictionary is default filled with four *StandardPlayers* with matching Key and *PlayerColor*.

4. Test cases succeed.
5. Refactoring 1. Firstly there is a potential for error, as we have a hard-coded player's count (4), and a dictionary containing 4 entries, but what if they came out of sync. We therefore change the *getPlayersCount* to return the number of elements in the dictionary.
6. Test cases still succeed.
7. Refactoring 2. Here we hit our first big refactoring, as we have a problem. The *Board* implementation, according to the CRC-cards, is supposed to know the current set-up of the board, not create the players. This is an example of responsibility-creep, where extra responsibility creeps in without is being actively chosen. There are two options here; supplying the Players in the constructor or implement the AbstractFactory-pattern. This first is most definitely the simplest, and as we presently do not "know" that there will be other "creation" required it is sufficient. Moving creation away from the class itself is also a good idea from a testing point of view, as we may now replace the *Player* implementation with a Mock object. In the present test this was not necessary, as we do not need to stimulate any indirect input (*PlayerColor*) in ways we cannot do with the *StandardPlayer*. From a pure Unit-testing point of view we should naturally stub away all functionality other than the UUT, yet we must weigh the advantages and disadvantages. The *StandardPlayer* is a relatively simple implementation, and it is part of the framework, so we can be fairly certain it has been tested and by simply code-inspection convince ourselves that it is correct. The risk of introducing defects in the stub is simply greater, and it is therefore not in the interest of the test to create a stub. The test is supposed to find defects, not service the unit-testing theorem to the letter.
8. Test cases still succeed.

We now implement the remaining tests methods for testing the blue and yellow, as well as test cases for the amount of money initially had (10 as was told at last seminar, as it is missing in specification). These are not shown here as they are of no academic interest. Even though these tests are not made directly with systematic testing, it is still used, as we test all possible elements in a set, and as it is not possible to test anything outside of the set (outside the enum) it is naturally not tested. Even the value 10 is in a way a systematic test, as it is equivalent to any other positive value up to `int.MAX`. It is not possible to enter a value larger than `int.MAX`, and negative numbers are simply not allowed (we see this when attempting to buy more camels than we have money).

After implementing the *Player* section we move on to the *Tile* section, and implement the tiles, beginning with the following iteration. The reason for this is not because it has to be that way – we could have chosen any functionality to implement, but we chose to implement the static functionality first, followed by the dynamic.

### 2.4.3 Iterate tiles on board

1. Create test *iterateTilesCountEqualsSevenTimesSeven*.
2. Test case fails to compile as there is no such method.
3. Implement the method *getBoardIterator* in the interface and in the class (Fake-it, simply return a 49 *Tile* dummy iterator).
4. Test case succeeds.
5. Refactoring – Already here we can see that we are faced with the same problem as we were with the *Player*, the *Board* implementation is not suppose to create the Tiles. Again we could choose to supply the tiles in the constructor, yet the constructor is getting rather cluttered, so instead we use the Abstract Factory pattern, and introduce a *BoardFactory* interface which can create not only the required tiles, but also the Players which was previously supplied in the constructor. This is also done in small stages.
  1. Introduce *BoardFactory* interface and method *createPlayers*. Use method to create Players in *AlphaBoard*.
  2. Test cases fail.
  3. Create *AlphaBoardFactory* (Obvious implementation – move code from test constructor to factory. We could have created a *BoardFactoryStub* for use in the testing, yet it would have to have the same functionality as the real *AlphaBoardFactory*, so it would merely be a waste. Naturally this goes against the rules of Unit-test, as we test more than one unit (both *AlphaBoardFactory* and *AlphaBoard*), yet *AlphaBoardFactory* is deemed as simple as any stub and it is therefore acceptable.
  4. Test cases succeed.
  5. Update *BoardFactory* interface with *createTiles* method, and use it in *AlphaBoard*.
  6. Test cases fail (compile error *AlphaBoardFactory*)
  7. Update *AlphaBoardFactory* with dummy implementation from *AlphaBoard*.
  8. Test cases succeed.

Here it may be seen again that the refactoring becomes more than just removing duplicate code, in order to keep the code good. This is not just a single occurrence. It seems that when using TDD, just like with XP, the rules should be followed judiciously. It is sometimes necessary to think a few steps ahead to avoid major refactoring. This does not mean design everything in advance, but simply that sound judgement also has a place in TDD.

Moving on to the next iteration we could validate the content of a single of the tiles, or we could validate the layout of all tiles. Taking small steps we should create test cases for each tile, like *tiler0c0EqualsSettlement*, *tiler0c0OwnedByRed*, *tiler0c0Has10Units*. One may argue whether this is beneficial, as it does create a lot of test cases ( $49 \times 3 = 147$  to be exact). In practice we believe this would never be an advantage, as it is quite possible to handle adding the entire tile-setup in one goes. It is, as can be seen in the code, only a single line per tile. We will however do it properly, and show the first iteration below.

1. Create test *tiler0c0EqualsSettlement*. We here had to decide if Position should start at 0 or 1. There does not have to be a one-to-one between the test name and the code (we could extract position 0,0 even though the name of the test is 1,1), but it is a good idea for readability. Since Position is defined to begin from 0, we decide to also use this.
2. Test case fail (Dummy *Tile* not equal to Settlement)
3. Update *AlphaBoardFactory* to return Settlement for tile 0,0 instead of dummy.
4. Test case succeed
5. Refactoring – Update the *Board*-interface with new method to return a specific *Tile* based on row and column. This saves iteration each time to find the correct *Tile* (at least in the calling code).

The remaining 146 iterations are not shown, as they are simply a matter of doing the above for each tile, and then updating the returned tile to match unit count and owner.

After these test-cases the board should be fully tested for initial static behaviour. Naturally, as the factory is called every time this is a stateless implementation, which is completely useless in a *Game*, but this will not be discovered until the implementation of the dynamic behaviour where the content of the *Board* has to change.

We choose to start with the *Player* ordering first (Red, Green, Blue than Yellow), but first we have to decide if we want to follow the pure unit-test approach and stub away the board (or from a pure TDD point of view, not use it). As the board is pure static functionality stubbing it will introduce a risk of generating new errors far greater than the risk of using the implementation, so this is not wise. We therefore choose to use the *Board* implementation in the *AlphaGame*. How is the *Board* implementation supplied to the *AlphaGame*? Well, to facilitate testability we supply it in the constructor, so it may be stubbed away should the need arise and also allow for replacing the board implementation. For now there is no need for a *GameFactory*.

#### 2.4.4 Player turn ordering

1. Create test *redHasTurnFirst*
2. Unit test fail (null reference exception – *getPlayerInTurn* return null)

3. Fake-it. We implement the *getPlayerInTurn* to simply return *Red* player from *Board*.
4. Test case succeeds.
5. Refactoring – none needed.

Then we want to implement the second player turn, and now we hit a snag. How do we change turns when we have not implemented the die roll, move or buy? Perhaps the turn was not the best suggestion? Is it not possible to choose any test to implement? Of course it is.

1. Create test *greenHasTurnAfterRed*
2. Unit test fail (always return Red).
3. Create fake-it *buy* implementation, which simply changes the current player to green (it is allowed to buy without moving, which ends the turn)
4. Test case succeeds
5. Refactoring – We create method *goToGreenTurn* in order to increase readability.

We follow this approach to *Blue*, where we see an option for refactoring.

1. Create test *blueHasTurnAfterGreen*
2. Unit test fail (return yellow)
3. Here we could simply implement another change, but with our triangulation we can see that the fake-it does not make sense any more. We therefore implement a buffer and an index which is updated every time a buy is processed.
4. Test case succeeds.
5. Refactoring – We create method *goToBlueTurn* in order to increase readability and avoid multiple *buys* under each other

For *Yellow* it is merely a matter of adding yellow to the buffer so we do not show it here, but for the final one we see a possibility for improvement.

1. Create test *redHasTurnAfterYellow*
2. Unit test fail (index out of bounce exception)
3. Implement resetting of the index when index reaches 4 (turning the buffer into a cyclic buffer). This is also the completion of a round, but it is of no importance now.
4. Test case succeeds.

5. Refactoring – Here we see the risk of switch-creep. We have to manually ensure that the index do not exceed 4. By encapsulating this it makes the code much more readable, and we using a strategy pattern to encapsulate the cyclic buffer. We introduce the *PlayerTurnStrategy* interface and *SimpleTurnStrategy* implementation where we move the implementation of the cyclic buffer too.
6. Test case succeeds.
7. Now of course we have two arguments for our constructor. Is this acceptable? Well, it may be and it may not - it is very much up to the individual. We however decide to create a *GameFactory* interface to create the board and the *PlayerTurnStrategy* and create an *AlphaGameFactory* with the creating code from the test and the *SimpleTurnStrategy*.
8. We move the code for going to red after having been to yellow, or “completing a round” in a method called *completeRound* to increase readability.

For the next iteration we choose the ordering within the turn (move (optional) than buy (may purchase 0)).

#### 2.4.5 Player Turn

Since it is allowed to not move, the buy-buy is allowed, just like move-buy and buy-move. The only illegal option is move-move. Naturally the first command is always valid and is not part of the test case (it is not validated).

1. Create test *buyBuyAllowed*
2. Test case succeeds

The same is true for move-buy and buy-move and we therefore skip them here and move on to move-move. We also realise that as it is not allowed we have to be careful with error hiding. If move incorrectly change to the next player then a following move would of the same players position would be invalid due to error-hiding, and if an incorrect move impl. kept the current player, and simply allowed multiple moves, then it would be error hiding if we moved the next players camels. Other errors could occur, and the question is, should we try and predict all the possible move errors. We do not believe so, and choose the most likely, where the move do not change state at all (allow multiple moves on same player)

1. Create test *moveMoveSamePlayerNotAllowed*
2. Test case fails
3. Triangulating with the previous means we cannot simply return false, so we implement a switch-based state pattern where buy always change state to move and move only succeed if state is move and then change to buy. There is a problem here, though. The CRC-cards state that the Board is responsible for move validation, and yet this implementation is in Board. We decide to postpone



the issue until we reach the invalid moves implementation later, and simply make a note of it in our test-list.

4. Test case succeeds.
5. Refactoring – Well, this is a parametric implementation of the state pattern, as oppose to the fully fledged implementation with sub-classes. It is however deemed acceptable here, at least for now.

#### 2.4.6 Move

Here is an interesting difference between systematic testing and TDD, as the breakdown of the tests into valid and invalid moves looks very much like Systematic testing, yet it is important to realise that it is not. These are overall TDD test cases, and we should therefore not attempt to test as many valid situations in one test case as possible. We should instead take small steps and create a single test case for each. In TDD the issue is not an explosion of test-cases – quite the contrary. It may be a little much to claim that more is better in TDD, but it is close. The smaller steps, the better and smaller steps mean more test cases. This is naturally only true to the point where the steps become smaller than what is beneficial, as TDD does advice using common sense when breaking down the overall tests into their individual steps.

This section also present an important possibility, as it will allow for the use of test-stubs. Instead of choosing the *AlphaBoard*, we can choose a *TestBoard* which can be configured to have the setup required. This same situation can be achieved by choosing a different *BoardFactory*, as it is the factory that determines the initial layout. The first is most unittest-like, as it means the least amount of units (only the *AlphaGame*) are a part of the test. Unfortunately it also has some disadvantages, as more functionality has to be introduced in the *Board* interface.

We also have to decide how much we want to test in a single test case; e.g. move 2 camels means validate move and extract two camels from current location, add two camels to destination and update ownership. Based on the “take-small-steps” principle we naturally create test-cases for each situation.

1. Create test  
*redMoveTwoCamelsFromRedSettlementWithTenCamelsToEmptyUnownedErgValid*
2. Test case succeed – always return true

We could now move on to other valid moves, but that would not make much sense, as they would all succeed. Instead we finish the move to Erg.

1. Create test  
*redMoveTwoCamelsFromSettlementWithTenCamelsToEmptyUnownedErgGivesOwnershipToRed*
2. Test case fails (the tile moved to does not change the Erg tile)



3. We could now fake it and simply set ownership to Red, but this would go against the test description, as it states an empty unowned Erg. We therefore implement the beginning move facility where it simply updates the tile owner to the owner of from – no matter what. This is true TDD, as we do not consider “what if you move from an unowned tile?” or “what if it is occupied?” etc. we only look at the test. Implementing this gives us a problem. The *Tile* does not have any way of being updated. We could cast it to *StandardTile*, but that is not very nice, as it results in a hard binding between *AlphaGame* and *StandardTile*, making it impossible to change the *Tile* implementation, in which case we might as well add the update methods from *StandardTile* to the *Tile* interface. This is also a solution, yet we do not really want future programmers, and users of the framework to start modifying the Tiles – they are mutable, just like Java strings. Well, how has Java then solved this problem? Well, yet have implemented *String* so that any methods which modify a string return a new instance of *String* equal to the modified string. Doing this is of course a creational matter, and not one we have to consider right here, as we have another layer of abstraction – the *Board*. The board may simply present a *Façade* for the move functionality, and then the actual implementation is done in *AlphaBoard* (see later). We therefore create a new method on the *Board* interface called *updateTileOwnership(Tile t, PlayerColor pc)*. In order for the test case to succeed we have to implement this method. We could implement it in the *TestBoard*, yet the problem is. How different will it be from the actual implementation needed in the *AlphaBoard*? We have to decide if we want to continue using the *TestBoard*, or if we should choose the alternate solution of creating a *TestBoardFactory* and use the real *AlphaBoard*. We choose the latter, as we believe it is more important to avoid duplicate code and thereby duplicate risk of error, than it is to adhere to the Unit-testing principles. This can also be seen from the fact that the *Board* functionality is tested via the *AlphaBoard* test cases. This is due to the fact that we are doing TDD and not unit testing; we believe that they are not the same. For this reason we implement the *updateTileOwnership* method to simply replace all tiles with Red ownership (fake-it), and yet we now have the problem of responsibility again. The *Board* implementation is not supposed to create Tiles; it is only supposed to hold them. How do we fix this? We simply add a *createTile* method on the *BoardFactory* interface which creates a new tile, identical to the previous except with the updated ownership and then have the *Board* implementation call this. This resulted in changes to *AlphaGame*, *Board*, *AlphaBoard*, *BoardFactory* and *AlphaBoardFactory*. Is this a small step? Not really, but there is no easy way around it and it is only small changes in the different locations. Is it good TDD? Good question.
4. Test case succeed.
5. Refactoring – We can see that in the *AlphaBoardFactory* implementation we have two places where Tiles are created; *createTile* and *createTiles*. We can avoid this by having *createTiles* call *createTile*, and this refactoring is implemented.

We could see from this iteration that it was not possible to keep nice responsibilities and at the same time take small steps (depending of course on the definition of “small”). This seems to be one of the problems with TDD, as it relies very much on a form of unit-

testing, which do not support interdependencies between units very much – just like unit-testing also does not. It may also be due to a poor use of TDD that the situation arose, or to a bad distribution of responsibility, yet whatever the reason, the problem exists and focus is easily lost. With the interface update in place we continue with the remaining test cases for the move, which should hopefully fit better with the TDD rhythm.

1. Create test  
*redMoveTwoCamelsFromSettlementWithTenCamelsToEmptyUnownedErgResultsInEightCamelsOnSettlement*
2. Test case fail (the from *Tile* is not changed)
3. Update *Board* interface with *updateUnitsOnTile(Tile t, int newCount)* and implement this method (done by using the *createTile* method from *BoardFactory*). Finally set the *AlphaGame* move to use this command. We could choose to hard-code the number 2, yet as it is an argument directly accessible and nothing is gained by faking it, obvious implementation is used.
4. Test case succeeds.
5. Refactoring – none needed.

Here we again see that the rhythm is not quite as simple. We have to change an interface and two implementations, which may be simple updates, but is still multiple files, and it is easy to lose focus.

1. Create test  
*redMoveTwoCamelsFromSettlementWithTenCamelsToEmptyUnownedErgResultsInTwoCamelsOnErg*
2. Unit test fails (only from is updated)
3. Call *updateUnitsOnTile* on the from *Tile* and extract the 2 units.
4. Test case succeeds
5. Refactoring – no refactoring needed.

These four test cases showed a valid Red move to Erg. To be completely exhaustive we should test to all player colours and to all valid types of tiles. We do however have an advantage of the white-box knowledge. If we implement the invalid moves as an exclusion algorithm, meaning we test for equality with invalid tiles, then we can put all valid tiles in the same equivalence class and therefore only have to test one. Naturally as the Tiles are a set, they should all be tested according to systematic testing, which is where equivalence classes comes from, so it is slightly contradictory. As is always the case with Equivalence classes. When in doubt, divide into two, and we should therefore create tests for all the valid Tiles. This is more systematic testing than TDD, though. The same is true for the players, even though it is pretty obvious that the validity-algorithm for empty unowned tiles is not going to be dependent on player colour. These test-cases

are not included in the report, as they do not contribute to the understanding of the process/product and we continue with move onto occupied empty tiles.

1. Create test

*redMoveTwoCamelsFromSettlementWithTenCamelsToEmptyYellowErgValid*

2. Unit test succeeds

The big question is – do we want to continue writing tests we know is going to succeed? We should, as they may begin failing when we implement the attack algorithm and yet is there a difference (equivalence class difference) in an unowned *Tile* and an Owned but empty *Tile* from a move point of view. Well, as the rule goes: “If you ask the question then there is a difference”, so we implement the remaining three test cases. These are not included in the report as they have no academic merit. The same goes for testing an owned empty tile of each *TileType*, as this should, from a systematic testing point of view, also be included. This also goes for the from tiles. Is there a difference between moving from a Settlement or moving from an Erg? As this is TDD, and we know the implementation, we have decided to say; no, there is no difference, and for this reason no other valid from tile type is tested.

We now go to the invalid moves, and start with

1. Create test

*redMoveTwoCamelsFromSettlementWithTenCamelsToSaltLakeInvalid*

2. Test case fail (we faked it so it simply returned true)

3. Triangulate – we now has to implement the validity-check, and as the CRC cards specify that it is the *Board*'s responsibility to determine validity, we have to update this interface with a method `validateMove(Position, Position, PlayerColor)` which indicate if the move is valid or not. This may be used directly in the *AlphaGame* implementation. Implementing the *validateMove* is simply return true in all instances except if “to” position point to a Salt lake.

4. Test case succeeds

5. Refactor – no refactoring needed.

1. Create test

*redMoveTwoCamelsFromGreenSettlementWithTenCamelsToErgInvalid*

2. Test case fails

3. The validation of move algorithm only looks at Salt Lake, and not at whose turn it is. Unfortunately the *Board* does not know whose turn it is, and can therefore not perform the validation. This is called a responsibility conflict, and we have to decide if we want to teach the *Board* whose turn it is, move all validation to the game, have validation code in both the *Game* and the *Board* or move validation to a separate class. The simplest is naturally to validate the player-turn in the *Game* and leave the rest in the *Board*, but this is not very good design. It is

difficult to understand and maintain. Teaching the board about which turn it is, e.g. by supplying it with a *Game* reference is possible, but also causes a more complex design, as there is now a two-way reference between *Game* and *Board*, and that should be avoided. We can move the validation to the *Game* implementation, but it already have plenty of responsibility, and we do not want "the blob". We therefore begin a Refactoring using a strategy pattern:

1. Create *MoveValidationStrategy* interface by moving method from *Board*.
2. Update *AlphaGameFactory* to create instance.
3. Update *AlphaGame* to use the factory to create strategy
4. Update *AlphaGame* to use strategy to validate move
5. Create *StandardMoveValidationStrategy* class and move the validation code from the *Board* implementation. We also remember that we found a problem during the move-move test, with some validation code that was left in *Game* (move only in Move state), and move the state validation too. This requires that *StandardMoveValidationStrategy* knows both *Board* and *Game*, and that is a problem. *Game* takes a *GameFactory* reference, so *GameFactory* does not know *Game*. How do we get a reference to *Game* inside *GameFactory*? There are several possibilities.
  1. We can move the *GameFactory* out of the *Game*'s constructor, so it can be created first and then use a simple set-method. We look at the code to validate, and it is possible as we have *newGame* to do all initialization.
  2. We can also set the *Game* on the *MoveValidationStrategy* after the fact. The key is to make sure it cannot be forgotten - this means not leaving it up to the programmer.
  3. We can implement a publish-subscribe system thereby allowing loose binding eventing of information, but it requires quite a lot of refactoring.

The disadvantage of set-methods is that they can be forgotten, Set-methods have the disadvantage that they can be forgotten. We believe that it is most likely to be remembered where configuration is done anyway, and that is at the creation of the Factory, and we therefore choose to create a set method on *AlphaGame* and update *AlphaGameFactory* to take *Game* in its constructor.

6. There is another serious issue. An abstract factory is suppose to create new instances every time, but it is not possible, as the *Board* returned by *createBoard* MUST be the same as is supplied to *StandardMoveValidationStrategy*. Luckily the *Game* acts as a proxy for the *Board* and we can avoid supplying the *StandardMoveValidationStrategy* with the *Board* at all.

7. As we have no systematic testing of the *Board* interface this breaks none of the test-cases, as the test cases test functionality and not interfaces, and that is nice. We then introduce the test of the *Tile* is owned by the *Player* who's turn it is in *StandardMoveValidationStrategy* and return false if not.
4. Test case succeeds.
5. Refactoring – none, as this was done during implementation.

Here we saw a different aspect of the TDD process. What to do when you realize a bad design. Does it follow the rhythm? No, we do not think so, but the focus on functionality in the testing does ensure that the test cases are still valid, which is very nice.

Here it may also be argued, according to systematic testing, that we need to test that the move was not performed anyway. This could be tested by simply also validating that the tiles have not changed, yet as we know the code, we know that the false is returned before any move-code, so it is not possible. Of course saying something is not possible is the same as saying it is going to happen, but this code is so simple we are confident in it. There is another aspect that should be tested, and that is whether a state-change occurs for an invalid move, which means that a buy is expected to follow an invalid move. This is done by writing a simple test; *moveAfterInvalidMoveIsValid* (iteration not shown here).

We now move to the next invalid, and that is moving units that are not there (or more than there are). In order to ensure that it is not an defect masking the test must ensure that the *Tile* is owned by the given colour, so it is not the owner-validation that catches (only test one invalid at a time => systematic testing; always a good idea).

1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithNoCamelsToBoardingEmptyUnownedErgInvalid*
2. Test case fails (move performed – results in negative camels at “from”)
3. Implement validation of Camel number (Obvious implementation, as oppose to Fake-it, which only checks if there are any camels. These two implementations are very similar, and it would be a waste to fake it).
4. Test case succeeds.
5. Refactoring – none needed.

We should also test for not enough Camels (other Equivalence and boundary value test), yet this is not included in the report, as only that we should is important, how is easy. We now realise that there is an invalid missing in the original test list. What if we move to a *Tile* that is not bordering? As we see it is important to not simply implement the overall test-list, but to insure that all functionality is tested. The original test-list is merely a guideline. We also realise that all the previous tests are missing some information in their names, namely “Bordering”. We simply assumed this, and we all know what happens with assumptions. This will be fixed in the following refactoring.

Some assumption we have permitted, as e.g. it is assumed that it is Red's turn when all these tests are being run. Is this correct? Well, we think it is a matter of readability, and that it will merely clutter the test-case name even more without contributing to understanding the test; it is a balance.

1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithTenCamelsToNonBoardingEmptyUnownedErgInvalid*
2. Test case fails (move performed)
3. Implement validation that the “to” *Tile* is bordering the “from”.
4. Test case succeeds
5. Refactoring – Update all move tests to specify that the *Tile* is bordering, except of course this one.

Is it likely that the code has been written to distinguish between *PlayerColour* or *TileType* when checking whether a *Tile* is *Bordering*? No, not really and we therefore do not need to test the above for the other types and the other colours. We therefore move on to the attacks, which we split up into two; determine if attack is needed and then performing the attack (small steps).

#### 2.4.7 Attack

As for the attack it is simply a variation of a move we can continue along the same lines.

1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithTenCamelsToBoardingYellowOwnedErgWithEightCamlesValid*
2. Test case succeed (it is a valid move, and the move is simply performed).
1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithTenCamelsToBoardingYellowOwnedErgWithEightCamlesResultsInRedOwnershipOfErg.*
2. Test case succeeds (we also not simply move the ownership).
1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithTenCamelsToBoardingYellowOwnedErgWithEightCamlesResultsInTwoCamelsOnErg.* Here we actually had to make a decision, as the specification does not show how we should distribute the remaining camels on the two tiles, we decide to move in with all of them, as you can only attack with all Camels (not the two that you moved – that number is irrelevant).
2. Test case fails (we actually end up with 10 camels (the 2 moved + the 8 yellow)



3. Implement fake-it attack, which identify if an attack is warranted (different owner) and if so always subtracts the "to"-tiles unit number from the "from"-tiles total number. This breaks the test of moving a number of camels to an owned *Tile* with no camels, as here all camels are moved. So we have to update the attack needed algorithm to return true when different owner AND camels on "to"-tile. And then perform the subtraction.
4. Test case succeeds
5. Refactoring – Moving the `isAttackNeeded` algorithm to separate method to ease readability. There are still a few ifs in the move algorithm, so we might think of a redesign/refactor, yet for now it is still readable, yet switch-creep is a risk.
1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithTenCamelsToBoarderingYellowOwnedErgWithEightCamlesResultsInZeroCamelsOnOrriginalRedTile.*

2. Test case succeed

Moving on to the loosing attacks. As all attacks are valid we

1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithTenCamelsToBoarderingYellowOwnedErgWithElevenCamlesResultsInYellowOwnershipOfErg*
2. Test case fails
3. Triangulate – We have already implemented the `isAttackNeeded`, and we now need to determine a winner. To increase readability we again move it to a separate method; `isAttackSuccessful`, and if not we do not change the ownership of the tiles (the move-algorithm is unchanged).
4. Test case succeeds
5. Refactoring – none needed
1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithTenCamelsToBoarderingYellowOwnedErgWithElevenCamlesResultsInOneCamelOnErg.*
2. Test case fails (we still subtract as if the attacker wins => negative units)
3. Triangulate. We need to differentiate between a winning and a loosing when moving and use the above method again to determine this, and reverse the subtraction in this case.
4. Test case succeeds
5. Refactoring – none needed.

And finally testing the loser's camels

1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithSevenCamelsToBoarderingYellowOwnedErgWithElevenCamlesResultsInZeroCamelsOnRedTile.*
2. Test case succeed (was implemented above as obvious implementation).

We have not dealt with a draw, so we will do so.

1. Create test  
*redMoveTwoCamelsFromRedOwnedTileWithTenCamelsToBoarderingYellowOwnedErgWithTenCamlesResultsInYellowOwnershipOfErg.* Here the specification was also a little vague. We decided to implement so that the attacking unit needs camel's to enter the *Tile*, and in a draw has none, so the ownership remains in the hands of the attacked.
2. Test case fails (isAttackSuccessful was implemented to return true is equal)
3. Fix algorithm
4. Test case succeeds
5. Refactoring – none needed

Tests for the remaining camels on the "to" and "from" tiles is also performed, yet these are left out of the report as they do not supply extra information. Now that we have finished the attack algorithm, we realise that there is a not-attack that has never been tested, and not until now do we realise the potential problem. What if a player moves camel's onto a *Tile* that is already owned by this player and has units on it. In the existing algorithm it is overridden, is it not? The simplest way to find out; write a test.

1. Create test  
*redMoveTwoCamelsToBoarderingRedOwnedTileWithTwoUnitsResultsInFourCamelsOnTo\_Tile*
2. Test case fails. We were right, there is a problem.
3. Triangulate. With the existing implementation we simply change a standard move (non-attack) to add the move number of units to the number of units on the "to"-*Tile*, and this is the number the tile is updated with.
4. Test case succeeds. The reason this is so simple to fix is that we were lucky with our *isAttackNeeded* algorithm, as we looked at ownership first, and not Camel count, as we might in that case only have looked at the Camel-count, and then the implementation would have required more.
5. Refactoring – none needed.



This is the completion of the attack. As we have all our previous move-tests we know that the attack-algorithm has not messed with the standard move, and that is good to know.

#### 2.4.8 Buy

We begin the buy procedure, or rather continue, as we have already implemented the state-change. When we look at the original test-list we realise that it has been completely forgotten. This is naturally not a reason to leave it out, so we begin the process.

1. Create test *redWithTenCoinsBuyFiveCamelsForRedSettlementValid*
2. Test case succeeds (We have already implemented that buy is valid)
1. Create test  
*redWithTenCoinsBuyFiveCamelsForRedSettlementResultsInFifteenCamelsOnSettlement*
2. Test case fails
3. We First fake-it and simply deploy the specified amount of coins to the indicated Tile. This is the highest level of fake-it we can do to avoid breaking previous tests.
4. Test case succeeds
5. Refactoring – none needed.

Well, how about an invalid purchase

1. Create test *redWithTenCoinsBuyElevenCamelsForRedSettlementInvalid*
2. Test case fails (we always allow buy)
3. Triangulate – We validate that the player in turn has the specified amount of money and if not return false.
4. Test case succeeds
5. Refactoring – We have to consider if we want to move validation out as a strategy-pattern just like with move. We decide to postpone it, as there is no responsibility conflict, and the code is still readable.
1. Create test *redWithTenCoinsBuyTwoCamelsForGreenSettlementInvalid*
2. Test case fails (we only validate sufficient coins)

3. More triangulation – We validate that the deployment-*Tile* is owned by the current player. We could fake-it and say it should be owned by Red, but it would be fake-it for the sake of faking.
4. Test case succeeds
5. Refactoring – Is it now we should move the validation out of Game? We decide to wait, as it is still quite readable.

What's next – well, we can only place on the Settlement, so we should try and place some Camels where it would normally be valid (by the overall rules), like an owned tile that is not a *Settlement*.

1. Create test *redWithTenCoinsBuyTwoCamelsForRedOwnedErgInvalid*
2. Test case fails
3. Triangulate – update test to only allow purchases to be placed on Settlements (must naturally also meet other requirements)
4. Other test case fails. We realise we have a problem with the yellow revenue test, as we cannot purchase anything while not having a settlement, and therefore not get a revenue calculation. What to do? We cannot use another player, as there is the same problem. He can never complete his turn without a *Settlement*? We decide for a quick-fix, and note the error in the requirements, and simply make an exception for buy-count = 0. This is not a proper solution, though, it is a nasty HACK. The requirements needs to be updated to specify what should happen, as we otherwise have a dead-lock.
5. Test cases succeed.
6. Refactoring – Here we could argue that it is getting so complex it warrants a separate class, but we believe it is still easily readable.

We have also not validated that the other *TileTypes* are not different, but we are confident they are handled the same (same EC). We could try a buy outside the board, but just like the previous the GUI would not allow it, so it is not a “valid” test - it is sufficient to simply document it, as is done in *Position* (only 0 – 6 is valid). We should naturally perform the same buy tests for the other players, but this has been left out of the report as it is similar to the above, except they succeed.

#### 2.4.9 Revenue

With the revenue calculation we could look at all possible combinations of all tiles and ownership and write a test case for it, but that would give more test-cases than we care to think off, and it would not be in tune with the idea of using logic, as it does not take that long. Just like with the *Tile*-layout we have the option of using systematic testing and test all valid in one test case or TDD and test one at a time. We need at least one test case for each *Tile*-type with a given ownership, or one test case put together so that all tiles types has an owner, and so that the sums may only come from a given combination of Tiles. Naturally this will allow for error-masking, but only if two errors

cancel out each other, e.g. settlement is 5 should be 4 and erg is 2 should be 3. If the same player owns these two we cannot see the error. How likely is this? Many orders of magnitude are smaller than a single *Tile* type failure. We do however decide to take small steps and implement a test for each *TileType*. We do however deem the colours to be handled identically, and we therefore only test one colour for all Tiles, as will be seen.

1. Create test *revenueRedWithSettlementGivesFourteenUnitsToRed*. Fourteen is due to the initial 10.
2. Test case fail
3. As we have already dealt with the units before (during player test we tested the initial value) we cannot simply fake it and return 14 (would make initial count test fail), we have to triangulate. We calculate revenue at end round (when circular-buffer turns). We find this out by updating the circular buffer to signal when it has completed a round. We decide to do this with an *observer pattern*, where the *Game* implementation may listen for a round completion. We could of course have kept a secondary counter or hard-coded that when the player returned to Red we should calculate revenue, but that would violate the entire principle of moving the round-maintenance to the circular buffer. We therefore implement the observer pattern and realise another advantage. As the circular buffer is a strategy pattern it is used via an interface, and is therefore under testing control, and we write a test version which trigger the revenue (by eventing round-done) the first time a turn is completed. This simplifies the unit test greatly, as we do not have to complete a whole round. We fake-it the revenue calculation and simply giving red 4 units.
4. Test case succeeds.
5. Refactoring - none needed.

We once again got stuck with an implementation that looks bigger than small. When one realise the potential for the use of a design pattern it is often a larger undertaking to implement it, and it is rare that the first test in the TDD will reveal this. There seems to be an inherent un-will to user overall design in the TDD process – very much like the XP process, which may result in well tested and quickly written code, but also makes it difficult to get an over-all understanding (no architectural documentation, as the architecture is not understood). Moving on.

1. Create test *revenueRedWithSettlementAndSaltMineGivesNineteenUnitsToRed*. Here we take advantage of the *BoardFactory* interface, and insert a *TestBoardFactory* that return the ownership we want, and therefore do not have to move around in order to test revenue.
2. Test case fail – only get 14.
3. Triangulate – we need a matching between *TileType* and Economic value. There are several ways of doing this;
  1. Create a big switch and let the *Game* implementation decide.

2. Create a method on *Board* to retrieve an Economic value from a *TileType*.
3. Create a strategy-pattern and have this interface map the *TileType* to an Economic value.
4. Making it part of the *TileType* enum (due to Java's implementation of enums)
5. Create method on *Tile* and retrieve from there.

There are plenty of advantages and disadvantages to these. From a logic standpoint the Economic value follow the *TileType*, indicating the enum solution. The problem with this one is mostly readability. Will a future reader understand what is going on, as it is a bit of a Java-trick? We fear not, so we choose against this. The strategy pattern allows for the replacement of economic values at runtime, yet as we have no such requirement at the moment this may be overkill. This leaves us with the switch solutions and the *Tile* interface update. As it is not the responsibility of neither *Game* nor *Board* to know determine the monetary value of a *Tile* we choose the *Tile* update and add the method *getEconomicValue* to the interface. The implementation may also be a switch or a value. If we choose a value it may be replaced at runtime, yet as we have no such requirement we choose a simple switch solution. The first implementation contains only *Settlement*. The implementation in *Game* is a simple iteration and accumulation of all (obvious implementation)

4. Test case succeeds.
5. Refactoring - none needed.

The remaining *TileType* test-cases are simply adding to the switch in the *StandardTile* implementation, so they are not included here, but they follow the rhythm quite well. These test-cases are left out of the report. We continue with the other colours.

1. Create test *revenueGreenWithSettlementGivesFourteenUnitsToGreen*.
2. Test case fails (only Red got revenue)
3. Triangulate – we update the code to also give Green revenue (obvious implementation to include the actual revenue calculation as for Red).
4. Test case succeeds.
5. Refactoring – none needed.

This is continued for the other two colours, yet this gives nothing to the academic knowledge so it is left out. We continue with the odd-situations; no settlement.

1. Create *revenueYellowWithErgButNoSettlementGivesTenUnitsToYellow*, i.e. no revenue (again this is simply done with our *TestBoardFactory*).
2. Test case fails.

3. Triangulate. We need to implement a check of whether Yellow has a settlement, and we do this during the revenue-calculation (simply set a flag if a settlement is found). This is relatively simple, and we do it for all four *PlayerColor* as obvious implementation
4. Test case succeeds
5. Refactoring – none needed.

Here we see a nice rhythm again, and with the obvious implementation the test of each of the Four *PlayerColor* missing a Settlement will succeed. We have left these iterations out of the report.

We move on to the other special case; a dead player. This turns out to be a little more difficult to implement, or rather requires some significant changes, as will be shown.

1. Create test *yellowLoosesOnlyTileResultsInPlayerCountThree* (again using *TestBoardFactory*). As the player-count is only available in the *Board*, and the board is the one who maintains the players, it is tested in the *TestAlphaBoard*
6. Test case fails (Yellow is still include in the *Player* count)
7. Well, we need to update the player count under a special condition, which is that there are no Tiles owned by Yellow. This is easily done by iterating all Tiles, but when do we need to perform this. The easiest is simply to have *Board* do the evaluation every time a *Tile* is updated. You could also do it only when a *Tile* changes owner, but as we have plenty of processing power we can afford to splurge and get some readability instead. As we do not test the player count after a move at any point in time we could simply remove read after the first move, but that is more like looking for a way to fake-it that we know will cost us extra work. We therefore take the high road and implement the iteration correctly. We create a private method on the *Board* implementation; *updatePlayers()*, which scans the Tiles for the *PlayerColors* in the collection of players and remove the ones that are not found. This will also affect the count.
8. Test case succeed
9. Refactoring 1 – We realise that there is a potential mismatch between the players returned by the *BoardFactory* and the *Board* layout returned by the same factory. We therefore perform an *updatePlayers* in the constructor after creating the *Board* layout, thereby erasing any players that should never have been on the board.
10. Refactoring 2 – There seems to be a funny situation with the player count. We have a requirement that there are initially four players, but there is no need for a method returning a player-count – at least the method is never used outside the tests. It is dead-code, and it should be removed, yet we cannot easily replace it with something else. We have methods to get a specific *Player* given his or her colour, but the number of players hidden behind the *Board* may be more (if there were more than four colours). This means that a unit test that checked that there

is a Red, Green, Blue and Yellow player is insufficient, as it will not catch if a Pink player is added (requires that the enum is expanded). We therefore choose to move the method from the interface and create special *TestBoard* interface which has the method, and let *AlphaBoard* inherit from both *TestBoard* (which extends *Board*). This is an acceptable solution that keeps the code for testing purposes, but does not show it to the production code.

We have now implemented the death of a player, but have not considered the game flow. What happens to the rounds? Do we update the Round algorithm in the strategy pattern and teach it about Players? Bad idea as it mixes responsibilities. We need to code to skip the dead player, though. Let's see what TDD would do.

1. Create test *currentPlayerAfterRedWithGreenDeadIsBlue*. Here we again use the *TestBoardFactory* to create a board with no Green Tiles, i.e. Dead Green player. The refactoring above makes this work.
2. Test case fails – (null reference exception)
3. We cannot allow the dead players to get their turn, as they have nowhere to place bought camels, even if they have units, and as the *Player* has been removed from the *Board* it cannot be retrieved. Presently the Java implementation of the *HashMap* return null if the key is not found. Some languages see this as an exception, as they specify that the key *MUST* be present. We could handle this null-value and use it to ignore the player, but that is placing a protocol into the return value object and that always makes for unreadable code. We prefer to see the returned null-value more like an exception and exceptions are meant to deal with exceptional situations, and a dead player is not exceptional, it is quite predictable. We therefore use the same trick as most programming languages use to at least limit the likelihood of an exception; first ask if the call is valid and then, if it is, perform it. E.g. If key exist in hash-table get value, otherwise don't. We update the *Board* interface with a method *hasPlayer(PlayerColor)* which returns if the player is in the Map. (Obvious implementation). If this method returns false we simply ask for the next *PlayerColor* from the *PlayerTurnStrategy*.
4. Test case succeeds.
5. Refactoring – We realise that in one of the very first test cases we actually did make the assumption that null meant player do not exist. This is of course in test code so the rules are a little more lenient, but at the same time we should not have un-pretty code. We could change the test to use the *hasPlayer* check, but then we would not actually verify that the player is returned in that test case, but since several other (e.g. testing that the player returned on Green also has a Green *PlayerColor*) it is not a problem. We should update with test cases for *hasPlayer* for Green, Blue and Yellow, as we left these out.

This may look like a victory, but what if two players die very unfortunately; Yellow kill Green and Red kill Blue?

1. Create test *currentPlayerAfterRedWithGreenAndBlueDeadIsYellow*.

2. Test case fails – (null reference exception)
3. Why did this explode? Because it is assumed that the next player in line is valid, it is not validated. How do we fix this? The simplest is a while loop that continues asking for the next *Player* from the *PlayerTurnStrategy* until it gets one that is valid. Is there a risk with such a while-loop? As with all self-controlling while-loops it can create an infinite loop, in this case if player count reaches 0. As this is an impossibility, unless the *BoardFactory* returns it, this is handled by simply making initial player count = 0 illegal by convention. It would also be pretty boring game.
4. Test case succeeds.
5. Refactoring – none needed.

We have now handled the dying players and we move on to the last part of the *Game*; the winning. We hit our first snag in writing the test. How do we determine if there is a winner? There is an eventing system which can send a text-string which we can then parse. This is a solution, yet not a preferable one. We use TDD to solve the problem.

#### 2.4.10 Winner

1. Create test *winnerInFirstRoundIsNone*
2. Test case fails (no such method)
3. We need a method to determine the winner, and create *getWinner()*, which returns a *PlayerColor* if a winner is found or *PlayerColor.None* if the game is ongoing. We fake-it implements this to return *PlayerColor.None*.
4. Test case succeeds.
5. Refactoring – none needed.

We could test for a heap of other none-win situations, but they would all succeed, so what is the fun in that. We move on to the win situation. We realise that there is no way to set round count, and we have no interest in performing  $25 * 4$  buy. Naturally this can be done quickly in a loop, but still; there must be a more elegant solution, and there is. The win is determined by a round count and the rounds are controlled by the *PlayerTurnStrategy*. It is therefore a brilliant place to store the round count and then simply create a *TestStub* for the test. We use this idea to write the test, inventing the names as we write the test.

1. Create test *redOwnsSaltMineAfterTwentyfiveTurnsGivesRedWinner*.
2. Test case fails. (compile error, no such method)
3. We first implement the method *getRoundCount* on the *PlayerTurnStrategy* interface. As we already event the revenue calculation on a completed round we have the perfect place to update our count. This is an obvious implementation.



As to the winner we fake-it and return Red if round count  $\geq 25$  (we could probably have obviously implemented this, but let's do one more iteration with a nice rhythm). We then set the current round count in the *TestPlayerTurnStrategy* implementation.

4. Test case succeeds
5. Refactoring – none needed.

As we still have one fake-it we create.

1. Create test *yellowOwnsSaltMineAfterTwentyfiveTurnsGivesYellowWinner*.
2. Test case fails
3. Triangulate – Now we have the problem. How do we get the SaltMine, and which one (only the board knows how many there is). The requirement places the salt-mine at 3,3, but the game does not know this. We now have to decide if we want to implement a strategy-pattern to place the “determine winner”-strategy in, or we can place it in the Game implementation. We decide on the latter strategy, as we do not have a responsibility conflict (and do not know of any reason to have a variability-point). We do not want to hard-code the Position(3,3) as the winner, and instead iterate the Board for TileType.SaltMine. This means the result is not the owner of Position(3,3), but rather the owner of the first SaltMine, which should be the same.
4. Test case succeeds.
5. Refactoring – none needed.

As it may be seen we have decided to continue the game if no-one owns the SaltMine after 25 turns, instead of terminating it. This means that if anyone gets the SaltMine after the 25<sup>th</sup> turn they instantly win. This is also why we chose to make the code round count  $\geq 25$  and not  $== 25$ .



## 2.5 Conclusion

That is the implementation of the AlphaTargui based on TDD. We have made some comments during the process, and here we sum up on a few of our points and also add some more thoughts.

As it may be seen we have not included any diagrams, as we believed it would take focus away from the process. The document and code has been written as an ongoing process with one iteration leading to the next, and with very little stopping up to look at the big picture – keeping focus. We have, however, as part of the conclusion created a class diagram showing the different static parts of the system, just to give an overview. This may be seen in Figure 2.

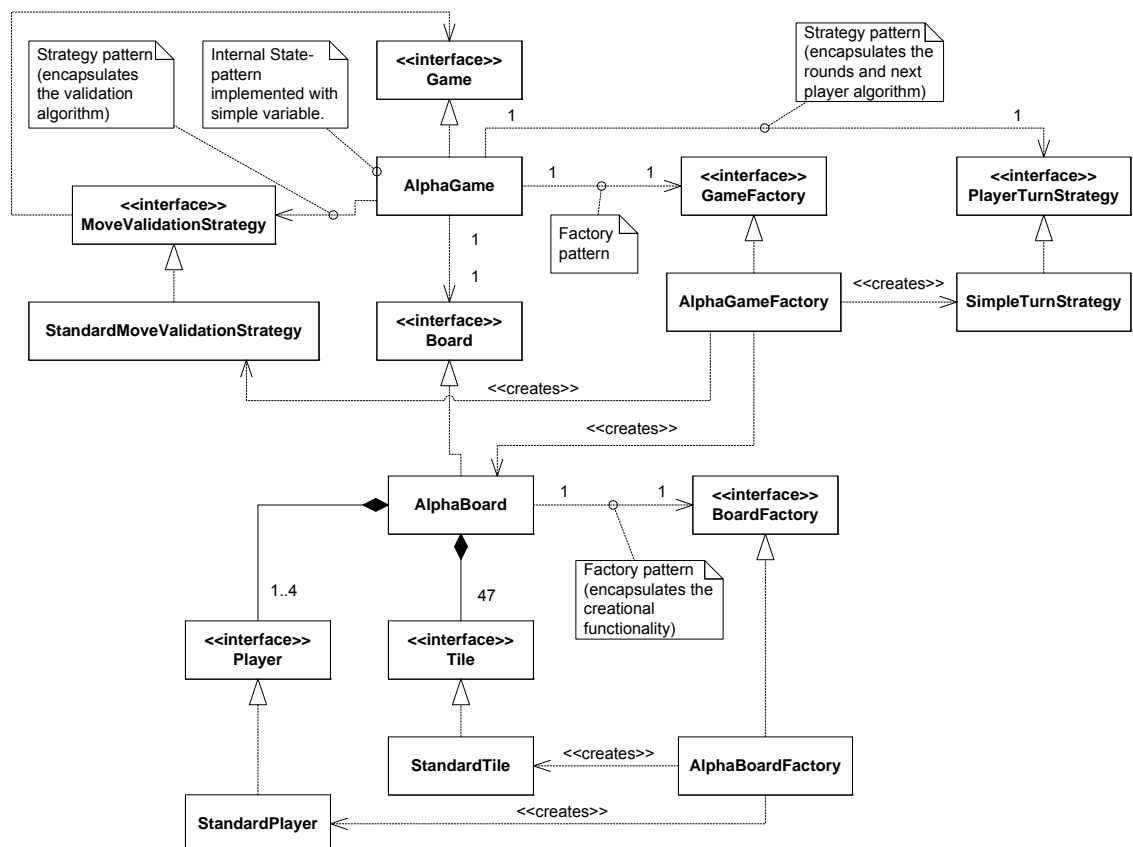


Figure 2

It is interesting to see how the TDD rhythm works much better when the interfaces (and implementation) of the types the UUT is dependent on is defined and implemented and we only have to alter the UUT itself. This can be seen as the different overall tests (each including several test-cases) reaches their conclusion. The last test-cases are always following the rhythm quite nicely, where the first has “larger” implementation or refactoring. We believe this is to be expected when TDD is used to define the interfaces as the process progress, just like in a traditional design phase, it is always important to define the interfaces and responsibilities first, and make sure that this is complete, in

order to avoid major refactoring during implementation (here code is included in the design phase, as the proof of concept usually means writing production code where needed, and faking it where possible until the chain is complete). After that all the in depth implementation starts; the kind that does not require in-depth understanding of the design.

We have considered if a test perhaps should be removed if it seems too big, but it creates tests that does not have any hold in the requirements. This may be a better approach, in hindsight, as some of the implementations actually contained iterations inside them (at least from a coding point of view). Of course having to stop mid-iteration to break the test up into smaller tests also interrupt the rhythm, so we would still have the problem.

We can also see that we have reused many of the same TileTypes, e.g. the Erg is used almost everywhere. From a Systematic Testing point of view this is silly, as we should have used as many different as possible to get maximum chance of finding defects.

The assignment itself may also not be the best suited for TDD. At a seminar it was once pointed out that “if you knew exactly what you were doing, then why waste time on iterative processes, when the waterfall-model works perfectly for those projects”, and also “if you know all the requirements in detail then why are you looking at the project, send it to India”. The point is that most projects are not as well specified as this one, and TDD gives some good possibilities of implementing what you can and postponing or encapsulating what you do not know, giving it more merit in those projects.

An alternative to TDD which it is believed would also have give a good result with less overhead is Responsibility Driven Design, and after the CRC cards design the Software Architecture, thereby getting a full understanding of dependencies, which would have discovered problems like the *Board* needing to know who's turn it is in order to validate move. This would have defined the interfaces, and Systematic Test could have been used on the individual interfaces and also there implementations (in same test perhaps), as fault detection, giving high reliability and readability (the design).

### 3 Systematic Test

#### 3.1 Introduction

We are going to make a systematic black-box testing of the BetaTargui attack algorithm. In black-box testing we don't have access to the production code.

A hint was given to this exercise that the best test investment is to concentrate on the kill algorithm.

We should consider the BetaTargui function to be tested to be:

$$\text{killed} = k(f, t, Uf, Ut, Sf, St, d);$$

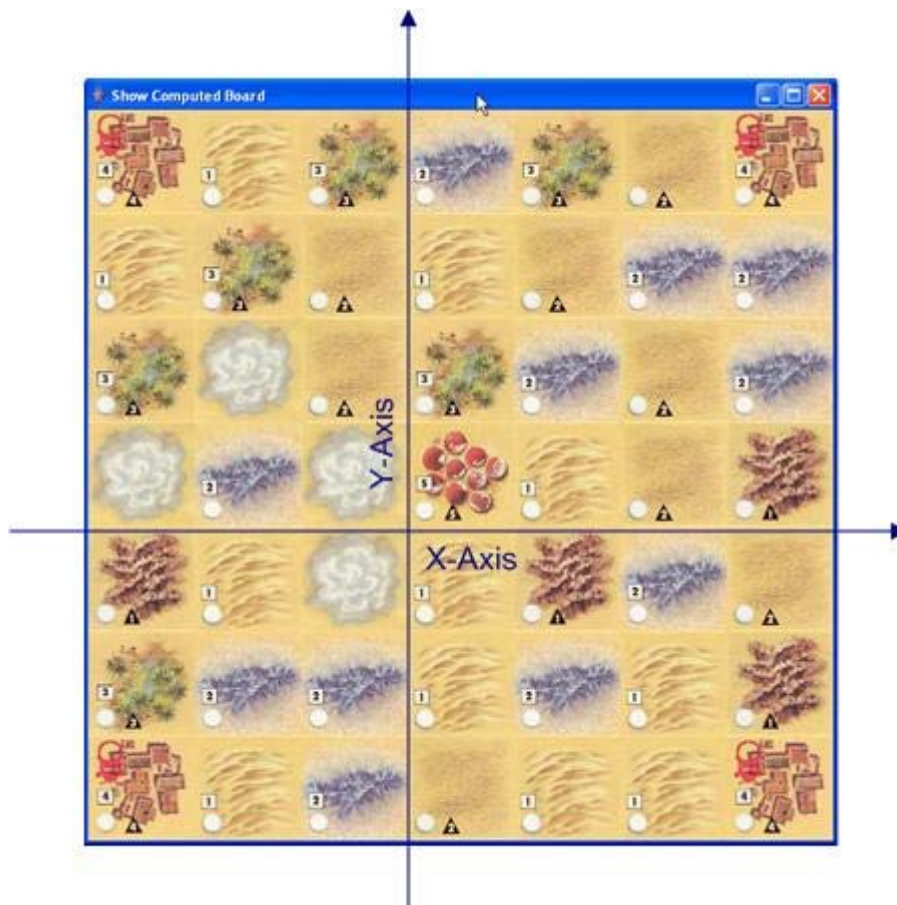
where 'killed' is the expected answer according to the Targui rules, 'k' the computed answer - our system testing effort must make us confident that the above equation is true under all circumstances. The parameters have the following domains:

- f: from tile  $\{(0,0), \dots (6,6)\}$
- t: to tile  $\{(0,0), \dots (6,6)\}$
- Uf: units on from tile  $\{1, \dots \text{infinity}\}$
- Ut: units on to tile  $\{1, \dots \text{infinity}\}$
- Sf: strategic value of from tile  $\{0,1,2,3,4,5\}$
- St: strategic value of to tile  $\{0,1,2,3,4,5\}$
- d: die roll value  $\{1,2,3,4,5,6\}$

We will use the abbreviations and the domain delimitations in the following assignment. For instance we will not waste time on testing if there are any units on from tile.

#### 3.2 Finding partition sets - distance

The figure below should illustrate that you could move the camels either in the X-axis direction (horizontal), in the Y-axis direction (vertical) or in both direction at the same (diagonal).



The first considerations regarding finding partition sets:

Dimensions	Type	Values	No. of partition sets
X-axis	range	-6 to 6	3
Y-axis	range	-6 to 6	3

Defining X-axis and Y-axis as separate range of values are not very useful. Let's try another way.

As when you just move camels we may only attack neighbour tiles. Thus we have to test if you are allowed to attack the attacked tile. Or in other words: is the distance between the tiles correct?

The distances can be calculated as:

$$X_{\text{dist}} = X_f - X_t$$

$$Y_{\text{dist}} = Y_f - Y_t$$

where  $X_f$  and  $Y_f$  specifies the tile where you attack from and  $X_t$  and  $Y_t$  specifies the tile you attack.

We can either test each dimension ( $X_{\text{dist}}$  and  $Y_{\text{dist}}$ ) separately or on a function of the two dimensions. To reduce the number of partitions we choose to test on both dimensions at the same time. It gives the following partition table with 13 partitions ([A1] to [A13]):

Dimensions	Invalid partition	Valid partition
"X <sub>dist</sub> , Y <sub>dist</sub> "	X <sub>dist</sub> < -1 [A1] Y <sub>dist</sub> < -1 [A2] X <sub>dist</sub> = 0; Y <sub>dist</sub> = 0 [A3] X <sub>dist</sub> > 1 [A4] Y <sub>dist</sub> > 1 [A5]	X <sub>dist</sub> = -1; Y <sub>dist</sub> = -1 [A6] X <sub>dist</sub> = -1; Y <sub>dist</sub> = 0 [A7] X <sub>dist</sub> = -1; Y <sub>dist</sub> = 1 [A8] X <sub>dist</sub> = 0; Y <sub>dist</sub> = -1 [A9] X <sub>dist</sub> = 0; Y <sub>dist</sub> = 1 [A10] X <sub>dist</sub> = 1; Y <sub>dist</sub> = -1 [A11] X <sub>dist</sub> = 1; Y <sub>dist</sub> = 0 [A12] X <sub>dist</sub> = 1; Y <sub>dist</sub> = 1 [A13]

### 3.3 Result of attack

We don't need to test if attacker has some camels to attack with, because the Targui domain code will never throw a value less than 1. It gives us one new valid partition:

Dimensions	Invalid partition	Valid partition
Uf		Uf > 0 [B1]

Next we have to test how many camels are killed and how many camels are left after the attack.

The number of camels killed by an attack can be calculated as:

$$F1_{\text{killed}} = \text{Math.round}((d_1 + Sf)/2)$$

where  $d_1$  = "die roll value" and  $Sf$  = "strategic value of from tile".  $\text{Math.round}()$  is function that rounds a number downwards towards the next lowest number.

Because the number of killed camels depends on  $d_1$  and  $Sf$  we get two new dimensions. Both  $d_1$  and  $Sf$  are set of values, which leads to a partition for each member and normally one for non-members. But because a die can't throw a value outside the set of values (1-6) and because the Targui domain code will never throw a value outside the set of values (0-5), we don't have any invalid partition.

Dimensions	Invalid partition	Valid partition
$d_1$		1 [C1] 2 [C2] 3 [C3] 4 [C4] 5 [C5] 6 [C6]
$Sf$		0 [D1] 1 [D2] 2 [D3] 3 [D4] 4 [D5] 5 [D6]

The next thing we have to test is if there are any camels left on the attacked tile:

$$U_t > F1_{\text{killed}}$$

And it gives the following dimension (or two new partitions [E1] and [E2]).

Dimensions	Invalid partition	Valid partition
$F1_{\text{killed}}, d_1, S_f$		$U_t - F1_{\text{killed}} > 0$ [E1] $U_t - F1_{\text{killed}} \leq 0$ [E2]

If  $U_t > F1_{\text{killed}}$  (partition [E1]) the defender is allowed to re-attack the attacker.

The numbers of camels killed by a re-attack are calculated on a similar way as by the first attack.

$$F2_{\text{killed}} = \text{Math.round}((d_2 + S_t)/2)$$

where  $d_2$  = “defenders die roll value” and  $S_t$  = “strategic value of to tile”.

Because the number of killed camels depends on  $d_2$  and  $S_t$  we get two new dimensions and a similar set of partition as for the first attack:

Dimensions	Invalid partition	Valid partition
$d_2$		1 [F1] 2 [F2] 3 [F3] 4 [F4] 5 [F5] 6 [F6]
$S_t$		0 [G1] 1 [G2] 2 [G3] 3 [G4] 4 [G5] 5 [G6]

### 3.4 Equivalence classes

Equivalence classes are the cross product of the partition sets. In our case it would give  $13 \times 1 \times 6 \times 6 \times 2 \times 6 \times 6 = 33.696$  combinations. Of course we can't test all these combinations so we will apply Myers recommendations for finding test cases.

According to Myers formulation you only have to test a single invalid partition one time and that it should be in combination with only valid partitions. 5 invalid partitions ([A1] to [A5]) give 5 test cases.

[A1] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]  
 [A2] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]  
 [A3] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]  
 [A4] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]  
 [A5] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]

Myers also says that you need to cover all valid partitions. To do that you need 8 test cases to cover the dimension " $X_{dist}, Y_{dist}$ " (partition [A6] to [A13]). All other partitions are covered at least one time.

[A6] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]  
 [A7] x [B1] x [C2] x [D2] x [E2] x [F2] x [G2]  
 [A8] x [B1] x [C3] x [D3] x [E1] x [F3] x [G3]  
 [A9] x [B1] x [C4] x [D4] x [E1] x [F4] x [G4]  
 [A10] x [B1] x [C5] x [D5] x [E1] x [F5] x [G5]  
 [A11] x [B1] x [C6] x [D6] x [E1] x [F6] x [G6]  
 [A12] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]  
 [A13] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]

Using Meyers recommendation we have now only 13 cases to test.

### 3.5 Test cases

Equivalence class	Inputs	Expected output
[A1] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]	[A1]: $X_{dist} = -2; Y_{dist} = 0; (X_f=4; X_t=2; Y_f=4; Y_t=4)$ [B1]: $U_f = 2$ [C1]: $d_1 = 1$ [D1]: $S_f = 0$ [E1]: $U_t - F1_{Killed} > 0$ where $F1_{Killed} = \text{Math.round}((d_1 + S_f)/2)$ $U_t = 2 \Rightarrow 2 - \text{Math.round}((1 + 0)/2) > 0$ [F1]: $d_2 = 1$ [G1]: $S_t = 0$	Error (Attack not carried out)
[A2] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]	[A2]: $X_{dist} = 0; Y_{dist} = -2$ [B1]: $U_f = 2$ [C1]: $d_1 = 1$	Error (Attack not carried out)

	[D1]: $S_f = 0$ [E1]: $U_t = 2 \Rightarrow$ $2 - \text{Math.round}((2 + 0)/2) > 0$ [F1]: $d_2 = 1$ [G1]: $St = 0$	
[A3] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]	[A3]: $X_{\text{dist}} = 0; Y_{\text{dist}} = 0$ [B1]: $U_f = 2$ [C1]: $d_1 = 1$ [D1]: $S_f = 0$ [E1]: $U_t = 2 \Rightarrow$ $2 - \text{Math.round}((2 + 0)/2) > 0$ [F1]: $d_2 = 1$ [G1]: $St = 0$	Error (Attack not carried out)
[A4] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]	[A4]: $X_{\text{dist}} = 2; Y_{\text{dist}} = 0$ [B1]: $U_f = 2$ [C1]: $d_1 = 1$ [D1]: $S_f = 0$ [E1]: $U_t = 2 \Rightarrow$ $2 - \text{Math.round}((2 + 0)/2) > 0$ [F1]: $d_2 = 1$ [G1]: $St = 0$	Error (Attack not carried out)
[A5] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]	[A5]: $X_{\text{dist}} = 0; Y_{\text{dist}} = 2$ [B1]: $U_f = 2$ [C1]: $d_1 = 1$ [D1]: $S_f = 0$ [E1]: $U_t = 2 \Rightarrow$ $2 - \text{Math.round}((2 + 0)/2) > 0$ [F1]: $d_2 = 1$ [G1]: $St = 0$	Error (Attack not carried out)
[A6] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]	[A6]: $X_{\text{dist}} = -1; Y_{\text{dist}} = -1$ ( $X_f=4;$ $X_t=3; Y_f=4; Y_t=3$ ) [B1]: $U_f = 2$ [C1]: $d_1 = 1$ [D1]: $S_f = 0$ [E1]: $U_t - F1_{\text{Killed}} > 0$ where $F1_{\text{Killed}} = \text{Math.round}((d_1 + S_f)/2)$ $U_t = 2 \Rightarrow$ $2 - \text{Math.round}((1 + 0)/2) > 0$ [F1]: $d_2 = 1$ [G1]: $St = 0$	1. Attack carried out $U_f = 2; U_t = 2$ 2. Re-attack carried out $U_f = 2; U_t = 2$ 3. Attacker gives up $U_f = 2; U_t = 2$
[A7] x [B1] x [C2] x [D2] x [E2] x [F2] x [G2]	[A7]: $X_{\text{dist}} = -1; Y_{\text{dist}} = 0$ [B1]: $U_f = 2$ [C2]: $d_1 = 2$ [D2]: $S_f = 1$ [E2]: $U_t = 2 \Rightarrow$ $2 - \text{Math.round}((2 + 2)/2) \leq 0$ [F2]: $d_2 = 2$ [G2]: $St = 1$	Attack carried out $U_f = 2; U_t = 0$
[A8] x [B1] x [C3] x [D3] x [E1] x [F3] x	[A8]: $X_{\text{dist}} = -1; Y_{\text{dist}} = 1$ [B1]: $U_f = 2$	1. Attack carried out $U_f = 2; U_t = 1$



[G3]	[C3]: $d_1 = 3$ [D3]: $S_f = 2$ [E1]: $U_t = 3 \Rightarrow$ $3 - \text{Math.round}((3 + 2)/2) > 0$ [F3]: $d_2 = 3$ [G3]: $St = 2$	2. Re-attack carried out $U_f = 0; U_t = 1$
[A9] x [B1] x [C4] x [D4] x [E1] x [F4] x [G4]	[A9]: $X_{\text{dist}} = 0; Y_{\text{dist}} = -1$ [B1]: $U_f = 2$ [C4]: $d_1 = 4$ [D4]: $S_f = 3$ [E1]: $U_t = 4 \Rightarrow$ $4 - \text{Math.round}((4 + 3)/2) > 0$ [F4]: $d_2 = 4$ [G4]: $St = 3$	6. Attack carried out $U_f = 2; U_t = 1$ 7. Re-attack carried out $U_f = 0; U_t = 1$
[A10] x [B1] x [C5] x [D5] x [E1] x [F5] x [G5]	[A10]: $X_{\text{dist}} = 0; Y_{\text{dist}} = 1$ [B1]: $U_f = 2$ [C5]: $d_1 = 5$ [D5]: $S_f = 4$ [E1]: $U_t = 5 \Rightarrow$ $5 - \text{Math.round}((5 + 4)/2) > 0$ [F5]: $d_2 = 5$ [G5]: $St = 4$	1. Attack carried out $U_f = 2; U_t = 1$ 2. Re-attack carried out $U_f = 0; U_t = 1$
[A11] x [B1] x [C6] x [D6] x [E1] x [F6] x [G6]	[A11]: $X_{\text{dist}} = 1; Y_{\text{dist}} = -1$ [B1]: $U_f = 2$ [B6]: $d_1 = 6$ [C6]: $S_f = 5$ [D1]: $U_t = 6 \Rightarrow$ $6 - \text{Math.round}((6 + 5)/2) > 0$ [E1]: $d_2 = 6$ [F6]: $St = 5$	1. Attack carried out $U_f = 2; U_t = 1$ 2. Re-attack carried out $U_f = 0; U_t = 1$
[A12] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]	[A12]: $X_{\text{dist}} = 1; Y_{\text{dist}} = 0$ [B1]: $U_f = 2$ [C1]: $d_1 = 1$ [D1]: $S_f = 0$ [E1]: $U_t = 2 \Rightarrow$ $2 - \text{Math.round}((1 + 0)/2) > 0$ [F1]: $d_2 = 1$ [G1]: $St = 0$	6. Attack carried out $U_f = 2; U_t = 2$ 7. Re-attack carried out $U_f = 2; U_t = 2$ 8. Attacker gives up $U_f = 2; U_t = 2$
[A13] x [B1] x [C1] x [D1] x [E1] x [F1] x [G1]	[A13]: $X_{\text{dist}} = 1; Y_{\text{dist}} = 1$ [B1]: $U_f = 2$ [C1]: $d_1 = 1$ [D1]: $S_f = 0$ [E1]: $U_t = 2 \Rightarrow$ $2 - \text{Math.round}((1 + 0)/2) > 0$ [F1]: $d_2 = 1$ [G1]: $St = 0$	1. Attack carried out $U_f = 2; U_t = 2$ 2. Re-attack carried out $U_f = 2; U_t = 2$ 3. Attacker gives up $U_f = 2; U_t = 2$

### 3.6 Boundary analysis

Referring to the domain delimitations:

- f: from tile  $\{(0,0), \dots (6,6)\}$
- t: to tile  $\{(0,0), \dots (6,6)\}$
- Uf: units on from tile  $\{1, \dots \text{infinity}\}$
- Ut: units on to tile  $\{1, \dots \text{infinity}\}$
- Sf: strategic value of from tile  $\{0, 1, 2, 3, 4, 5\}$
- St: strategic value of to tile  $\{0, 1, 2, 3, 4, 5\}$
- d: die roll value  $\{1, 2, 3, 4, 5, 6\}$

we can't find any boundary cases.

The coordinates for the tiles have been defined to be between (0,0) and (6,6) and distance ( $X_{\text{dist}}$  or  $Y_{\text{dist}}$ ) can be either 0, 1 or greater than the numeric value of 1. We have tested all 3 cases regarding distance.

The strategic values and the die roll values have been defined to be between 0 and 5 and between 1 and 6 respectively. All cases have been covered by the test cases.

Re-attack has also been tested, so we can't find any further things to test.

## 4 BetaTargui

### 4.1 Introduction

BetaTargui is identical to AlphaTargui except for the 3 variability points “Turn”, “Attack” and “Winner”

### 4.2 Variability points

#### **Turn**

Responsibility: Putting units after buy

```
Public interface PutUnitsStrategy{
    Boolean isPutValid();
}
```

#### **Attack**

Responsibility: Attack

```
Public interface AttackStrategy{
    // return the number of surviving units from attack...???
    public int attack(Tile tFrom, Tile tTo, int dieValue);
}
```

#### **Winner**

Responsibility: Find winner

```
Public interface WinnerStrategy{
    PlayerColor getWinner(Player p, Tile t);
}
```

To create the different game variants with different strategies we use the Abstract Factory Pattern:

```
Public interface GameFactory{
    createBoard();
    createPutUnitsStrategy();
    createAttackStrategy();
    createWinnerStrategy();
}
```

We also replace AlphaGame with StandardGames. It just looks nicer when creating the different games.

*Consider using State Pattern for variables implementing internal state...???*

To be continued...



## 5 DeltaTargui

### 5.1 New requirement: Randomized board

#### 5.1.1 Analysis

In the AlphaTargui variant we introduced the BoardFactory interface. The BoardFactory describes the responsibilities of creating a Board. In AlphaTargui, we created an implementation of the BoardFactory named AlphaBoardFactory. The implementation placed all tiles on the board in a predefined way.

In the DeltaTargui, we want to randomize the positioning of the tiles on the board; hence we create a new implementation of the BoardFactory named DeltaBoardFactory. DeltaBoardFactory implements the algorithm defined in the project description. The algorithm is stated as:

1. The saltmine is positioned in the center of the board.
2. The different types of tiles are placed randomly on the board. The number of each tile type is predefined.
3. Each corner tile is replaced with one of the four settlement tiles. The position is as follows:
  - Red settlement – Upper left corner (0, 0).
  - Green settlement – Upper right corner (0, 6).
  - Blue settlement – Lower left corner (6, 0).
  - Yellow settlement – Lower right corner (6, 6).

To take small steps and keep focus, the above steps are implemented in multiple iterations. In each step a requirement is implemented.

The test list contains a test that reads: "Tile positions matches specification". This test definition is not very concrete and the test is therefore broken down into smaller tests, who match each of the three identified iterations.

#### 5.1.2 Design choices

In AlphaTargui we already used the Abstract Factory pattern to support the implementation of the AlphaBoard. In DeltaTargui, we will simply implement the BoardFactory interface to support another variant of the board – DeltaBoard. The DeltaBoardFactory has responsibility of creating the board according to the DeltaTargui specification.

For the best implementation, we will analyze the AlphaTargui implementation to determine the variability points and how to create the new implementation. For this purpose, we will use the 3-1-2 process.

Variability points:

1. Position of tile – random or fixed.
2. Type of tile – differs between the types given in the requirements.

### 5.1.3 Position saltmine

For this iteration, we need to position the saltmine in the center position of the board according to the specification. For this purpose we add the test case 'testSaltmineCenterPosition' to the newly created TestDeltaBoard class. Testing for the position of the saltmine is simple:

```
/**
 * Test that the Saltmine is in the center position (3,3)
 */
@Test
public void testSaltmineCenterPosition() {
    // Get tile at center position
    Tile tile = board.getTile(new Position(3,3));
    // Get tile type
    TileType tileType = tile.getType();
    // Assert it's the Saltmine
    assertTrue(tileType.equals(TileType.Saltmine));
}
```

### 5.1.4 Position tiles randomly on board

In AlphaTargui, we created an instance of the BoardFactory with fixed positioning of the tiles. We generated test cases for the fixed positions.

Now, we have to generate random positions for the tiles, who we add to the board. Compared to the AlphaTargui, this serves a new problem. In AlphaTargui, a tile has the responsibility of knowing its own position. This means, that the board does not have any knowledge about where each tile type is positioned. The board can determine the type of the tile, position, owner and units on the tile by using the tile interface.

When we need to add tiles randomly to the board, we should not place a tile on the board, where a tile previously has been positioned. This is only the case when placing the settlements.

## Lessons learned

When implementing this requirement, the first approach was to generate the positions, available for the tiles that should be positioned. Generating the positions was fairly simple with a double loop:

```
Position position;
for (int i = 0; i < 7; i++) {
    for (int j = 0; j < 7; j++) {
        position = new Position(i, j);
    }
}
```

This loop created all the positions on the board. The positions were held in a collection, where each position was removed and assigned to a tile. Creating the tiles stated in table 2.1 in the requirements specification was simply a question of looping the number of times according to each specific tile and create the tile.

The created tiles were held in a collection. This collection was iterated and assigned a position from the available positions.

Although the algorithm for filling the board, there was not paid so much attention to the sequence of the steps, hence they were altered to the below mentioned for ease of implementation.

1. Place the 'Saltmine'
2. Place the settlements on the predefined positions.
3. Place the randomized tiles.

There were 49 available positions. Suddenly the createTiles() method started throwing IndexOutOfBoundsException exceptions. This was strange; because the test cases created for probing the number of each tile type was not revealing any defects. The correct number of tiles existed. Debugging the loops and reading the table again didn't reveal anything.

OK, back to good old system out. Everytime a tile was created, a line with a description of the tile was printed. Before assigning positions to the tiles, the line count was 53. Not 49 as the available positions. Summing the numbers in table 2.1 also gave 53.

Reading the algorithm for populating the board revealed the defect; the settlements were to replace the four corner tiles, meaning that four random tiles were discarded from the game. Implementing this was thought to be simple, but later proven otherwise. As we removed positions from a collection upon tile creation, these positions were used and no longer available. How to reassign the pre-defined positions for the settlements?

A new solution was made: We created a simple deck interface with the responsibility of creating a deck of tiles according to the DeltaTargui specification.



```
public interface Deck {  
  
    public Tile getTile(Position position);  
    public void shuffleDeck();  
    public boolean isEmpty();  
}
```

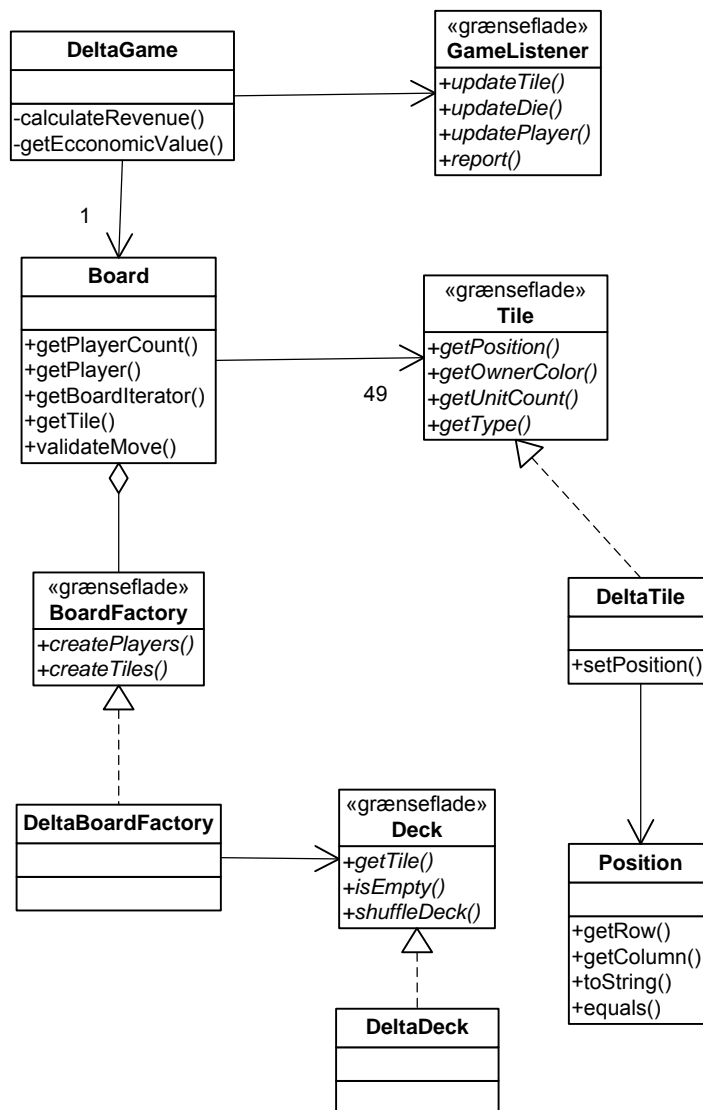
Internally, the deck implementation (DeltaDeck) consists of stack, which is given the tiles for the DeltaTargui specification.

A new implementation of the Tile interface was made (DeltaTile), where it's possible to assign a position to a tile after creation. Randomness was made by using the Collections.shuffle method in Java. This method applies standard random behavior. The original object-oriented thinking is very much applied to this implementation. The deck is seen as simulation of a part of the physical world.

Another advantage of the new deck interface is that responsibility for creating the deck is separated out from the DeltaBoardFactory.

What we learned from this episode are:

1. Read the specification, even small changes can mean a lot to the architecture.
2. Making mistakes might lead to better software, as we were forced to think the things over again.
3. Place test cases at the "right level". In this case it was good that we tested for the amount of each tile type. This eventually led to a reveal of a defect. If we had tested for the amount of tiles on the entire board, this would probably not have happened.
4. Test cases make code evolve and vice versa. The test cases we initially created revealed a defect. But as the code changed, some of the test became outdated. It's not possible to test for a specific number of tiles, when some of tiles are randomly replaced. Though, the test cases are OK, when they are run before the tiles are replaced.
5. Keep test cases isolated.
6. Refactor step in TDD is important, as the "gold" is here. The reflections made here are very important for your code and often provides efficient code changes.



### 5.1.5 Position settlements

Position of the settlements was done as a step in the algorithm, where the randomized tiles were created. To place the settlements, each tile in the collection of tiles was tested for match of the settlements positions. The easiest solution here was to parameterize the positioning by using some conditional statements:

```
if (row==0 && column==0) {
    tilesIterator.remove();
    DeltaTile redSettlement = new DeltaTile(TileType.Settlement, PlayerColor.Red);
    redSettlement.setPosition(new Position(0,0));
    tilesIterator.add(redSettlement);
} else if (row==0 && column==6) {
    tilesIterator.remove();
    DeltaTile greenSettlement = new DeltaTile(TileType.Settlement, PlayerColor.Green);
    greenSettlement.setPosition(new Position(0,6));
    tilesIterator.add(greenSettlement);
} else if (row==6 && column==0) {
```

### 5.1.6 Conclusion

The new requirements for DeltaTargui was fairly easy to implement as we have made some good design choices when implementing AlphaTargui.

Benefits:

1. Change by addition, as we add DeltaBoardFactory to create the tiles.
2. Centralized responsibility for creating tiles and positions. The responsibility lies within the concrete implementation of the BoardFactory. This enhances code readability as the configuration code is defined in a single place.

Liabilities:

1. Creating new variants based upon code in the DeltaBoardFactory (e.g. randomizing the tiles in another way) result in either too much responsibility placed in DeltaBoardFactory or code duplication.

New class diagram for DeltaTargui:

**Gruppe 9**

Anders Hvidgaard Poder, 19951439

Elund Christensen, 20074530

Kewin Peltonen, 20054669

Programming Project

HotTargui

1.0 D

07-04-2008



51 (51)

## 5.2 New requirement: Determine winner

### 5.2.1 Analysis