COMP-361 Project
Design Models

(12% of final grade)

Hand-in Date

Please submit your models (upload on myCourses) by Friday February 13th 2015. The handin consists of 2 models: the Interaction Model and the Design Class Model.

Scope

Some groups are going for a peer-to-peer implementation of Medieval Warfare (i.e., all nodes in the distributed system run the same application), while others are planning to use a client-server architecture (i.e., player nodes run a "game client" application that connects to a single "game server"). Whichever architecture you use, there is a node in your distributed system that holds the complete game state[1].

For this hand-in you must only develop the design models for the node that stores the entire game state. A partial concept model that specifies the conceptual game state that this node (peer or server) must minimally contain is presented in Figure 1. The information about external actors that the system communicates with was intentionally omitted from the diagram, since it would vary depending on the chosen architecture.

The following conceptual enumeration types are used in the concept model.

```
type PlayerStatus is enum {Offline, Online};
type LandType is enum {Sea, Grass, Tree, Meadow};
type VillageType is enum {Hovel, Town, Fort};
type UnitType is enum {Peasant, Infantry, Soldier, Knight}
type ActionType is enum {ReadyForOrders, Moved, BuildingRoad, ChoppingTree,
  ClearingTombstone, UpgradingCombining, StartCultivating, FinishCultivating}
```

Interaction Model / Behavioural Design

The behavioural design consists of 7 operations. The behaviour of each operation is described below with an operation schema that describes the conceptual state changes that this operation is supposed to achieve. Please provide an Interaction Model in form of a Sequence Diagram (Communication Diagrams are not acceptable) that depicts *all method invocations* (even invocations of *getters* and *setters*) needed to achieve the functionality described for each of the operations.

Of course you are allowed to submit additional sequence diagrams to describe the behaviour of design methods that you introduce. This is useful, for example, when several operations share common methods, or if the design of a complex operation becomes too big. In this case, simply call the new method in one design, and then specify the behaviour of the method in a separate diagram.

Note that output messages have been intentionally omitted from the operation schemas, because depending on if your game runs on a server or a peer, outputs might have to be sent over the network or displayed by means of a GUI. We ask you therefore to not include any communication design in the models you are handing in. In other words, you do not have to show how you design your GUI and your network communication.

[1]Typically, all peers in a peer-to-peer system store the entire game state. In a client-server setting, the full game state is stored on the server.

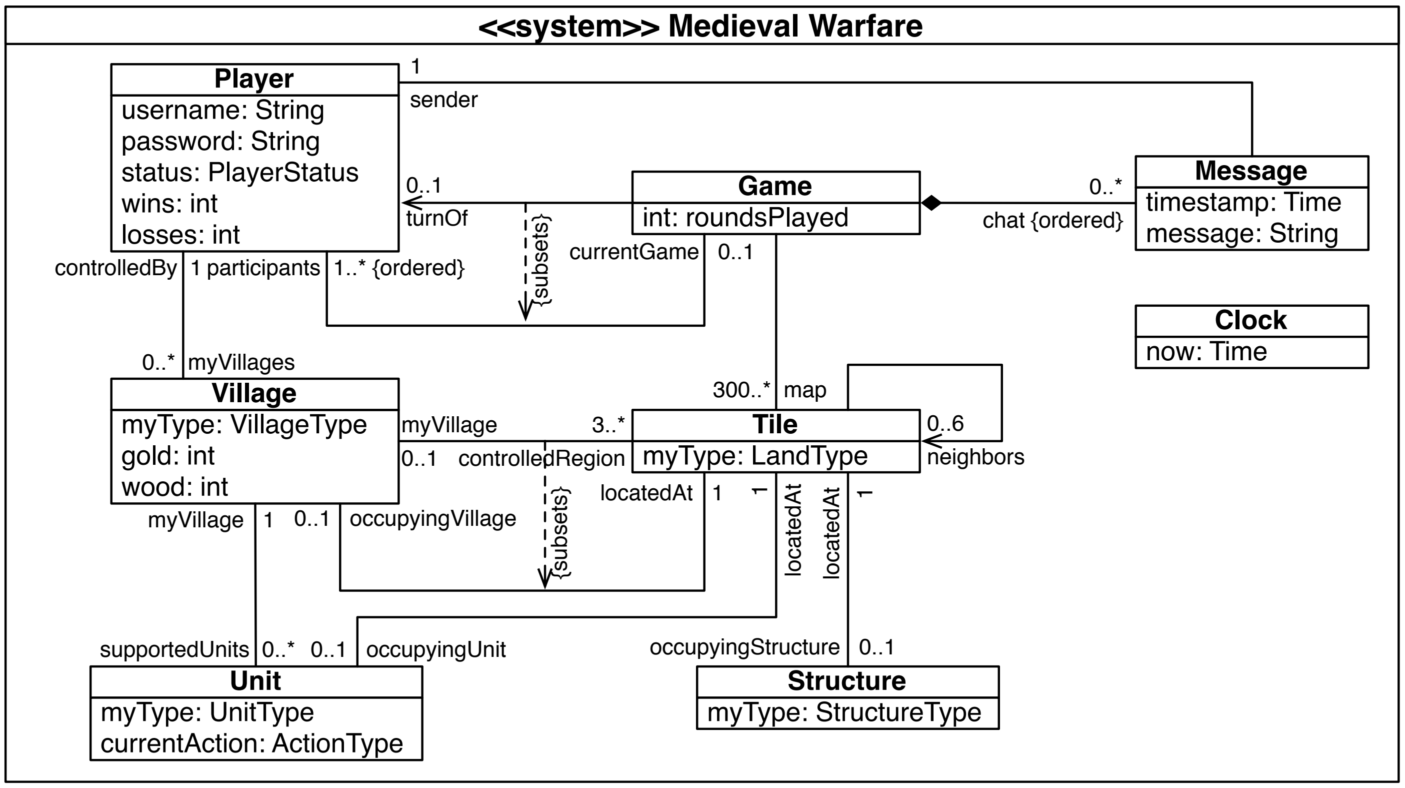

Figure 1: Medieval Warfare Conceptual Game State

Game Management (2 operations)

Provide an Interaction Model (in form of a Sequence Diagram) for the *newGame* and *beginTurn* operations described conceptually below. For each of the operations you must decide:

- What class will be the controller of this operation. Please justify this choice with one or two short sentences.
- How to encode the conceptual parameters of the operation (please explain the encoding, if it is not obvious).

Operation: MedievalWarfare::newGame(participants: Set<Player>) or
MedievalWarfare::newGame(participants: Set<Player>, String mapToLoad)
Scope: Game, Tile, Village;
New: newGame: Game; newTiles: Set<Tile>; newVillages: Set<Village>
Description: The *newGame* operation creates/initializes all data structures needed to play a new game of Medieval Warfare between (at least) two players (identified by the participants conceptual parameter). This includes creating a new game, creating the tiles, and initializing the structure and topology of the battlefield (grass, meadow, sea, trees) either by generating the map or by loading a preexisting one (identified by the mapToLoad parameter) and assigning initial regions to players. It also positions the starting villages at some random positions within each region.

Operation: MedievalWarfare::beginTurn(g: Game, p: Player)
Scope: Game, Player, Village, Tile, Unit, Structure;
New: newStructures: Set<Structure>
Description: The *beginTurn* operation performs all game state updates that need to be done at the start of player p's turn. This includes replacing tombstones on tiles owned by p with trees, producing meadows and roads, updating the gold stock of each village with the income and paying the wages of the villagers, potentially replacing villagers by tombstones if their wages cannot be paid.

Game Moves (5 operations)

Provide an Interaction Model (in form of a Sequence Diagram) for the operations *buildRoad, upgradeVillage*, *upgradeUnit*, *takeoverTile* and *moveUnit* (ordered in increasing complexity of conceptual change to the game state). For each of the operations you must decide:

- What class will be the controller of this operation. Please justify this choice with one or two short sentences.
- How to encode the conceptual parameters of the operation (please explain the encoding, if it is not obvious).

Operation: MedievalWarfare::buildRoad(u: Unit)
Scope: Unit;
Description: If the unit *u* is a peasant, then *buildRoad* instructs *u* to start building a road at the current location. Otherwise the game state is left unchanged.

Operation: MedievalWarfare::upgradeVillage(v: Village, newLevel: VillageType)
Scope: Village;
Description: The *upgradeVillage* operation upgrades the village (identified by the parameter *v*) to a new level (identified by parameter *newLevel*), using the wood of the corresponding village to pay for the upgrade. In case there is not enough wood, the game state is left unchanged.

Operation: MedievalWarfare::upgradeUnit(u: Unit, newLevel: UnitType)
Scope: Unit, Village;
Description: The *upgradeUnit* operation upgrades the villager (identified by the parameter *u*) to a certain level (identified by parameter *newLevel*), using the gold of the corresponding village to pay for the upgrade. In case there is not enough gold, the game state is left unchanged.

Operation: MedievalWarfare::takeoverTile(dest: Tile)
Scope: Player, Village, Tile, Unit, Game
New: newVillages: Set<Village>
Description: The behaviour of the *takeoverTile* operation is to remove the *dest* tile from the region of the enemy player. It is assumed that the unit that is capturing the tile is already on the *dest* tile (i.e., another operation (see *moveUnit* below) has already checked that this tile can be captured by this unit, and moved the unit to the *dest* tile, and destroyed any enemy units that might have been on the tile). Depending on the situation, one or several of the following game state updates are additionally performed:
- if the *dest* tile has an enemy village on it, then it is invaded (i.e. the gold and wood of the enemy village are added to the village of the capturing unit) and then destroyed. If the enemy region is still big enough to support a village, a new hovel is placed at a random place on the enemy region.
- if capturing the dest tile split the enemy region into two or three unconnected regions, then the region that has the enemy village on it loses all units that are not connected to the village anymore. Furthermore, if any of the new regions is big enough to support a village, a new enemy hovel is created on a random tile in that region, and the units that are located in that region are assigned to the new region. Any units located on regions that are too small to support a village are destroyed, and all tiles belonging to the region are converted to neutral tiles.
- if the enemy player has no regions left under his control, he is eliminated from the game and his loss statistics are updated.
- if the last enemy player is eliminated, the win statistics for the current player are updated.

Operation: MedievalWarfare::moveUnit(u: Unit, dest: Tile)
Scope: Game, Player, Village, Tile, Unit, Structure;
New: newStructures: Set<Structure>
Description: The behaviour of the *moveUnit* operation is to move a unit (identified by parameter *u*) to some new position on the battlefield (identified by the *dest* parameter). If the unit can not be moved or if it is impossible for the unit to reach the destination tile (e.g., because the destination tile is not a neighboring tile) or if the unit is not allowed to walk on the *dest* tile (e.g., because it is a sea tile, or because *u* is a knight and *dest* contains a tree, or because the tile is protected by an enemy unit of equal or greater level, ...), the game state is left unchanged. If it is possible for *u* to reach *dest*, the current location of the unit is set to *dest*. Depending on the destination tile, one or several of the following game state updates are performed:
- if the *dest* tile has a tombstone, the tombstone is removed.
- if the *dest* tile has a tree, the tree is removed and the wood reserve of the village is updated.
- if the *dest* file has a meadow without road and *u* is a knight, then the meadow is reverted to grass.
- if the *dest* tile contained an enemy unit, then the enemy unit is destroyed.
- if the *dest* tile is neutral or enemy territory, it is added to the region controlled by the village. In the case where the acquisition of this tile results in two regions of the same player touching each other, then the regions are combined into one, and so are the two villages (the village of lower level is removed, and its gold and wood reserve are added to the reserve of the other village, and so is the control of its units).
- if the *dest* tile is enemy territory, then the *takeoverTile* operation effects (see above) are also applied.

Design Class Model / Structural Design

Elaborate a design class model that comprises all the structural properties needed to realize the behaviour that you described in the interaction model for the 7 operations. Remember that the design class model must show *all attributes* (all with visibility *private*) and *methods for each class* you used in your interaction model, as well as *navigable associations with role names* between them, if any. Also, any inheritance relationships between classes, if any, need to be shown. *You do not have to show usage dependencies in the design class model.*

Note that, since all attributes have *private* visibility, appropriate *getter* and/or *setter* methods have to be defined whenever an attribute's value needs to be made available to other objects. You can assume that template collection classes like the ones that Java provides are available in order to implement any multi-objects you need. Simply name them *CollectionOfX* (where *X* stands for the classname of the elements that the collection contains).

Tool Support

We believe it should be possible to model the design with TouchCORE (http://www.cs.mcgill.ca/~joerg/SEL/TouchCORE.html). The advantage of using TouchCORE is that by construction your design model and interaction model will be consistent. Also, if you are using Java as an implementation language, TouchCORE allows you to import Java classes into your design (such as, for example, ArrayList), and eventually generate Java code from your design. On the downside: the tool is unfortunately still not very stable. We will promise, though, to address bugs as fast as possible.

In case you choose to use TouchCORE, please upload a zip file to myCourses that contains the .core and .ram files produced by TouchCORE and a pdf that justifies the choice of controllers.