DESIGN

LINKÖPINGS TEKNISKA HÖGSKOLA | IDA | TDDD09 | GRUPP 3

REVIEWED	Sebastian Wihlborg	2013-05-05
APPROVED	Josef Gustafsson	2013-05-05

TABLE OF CONTENTS

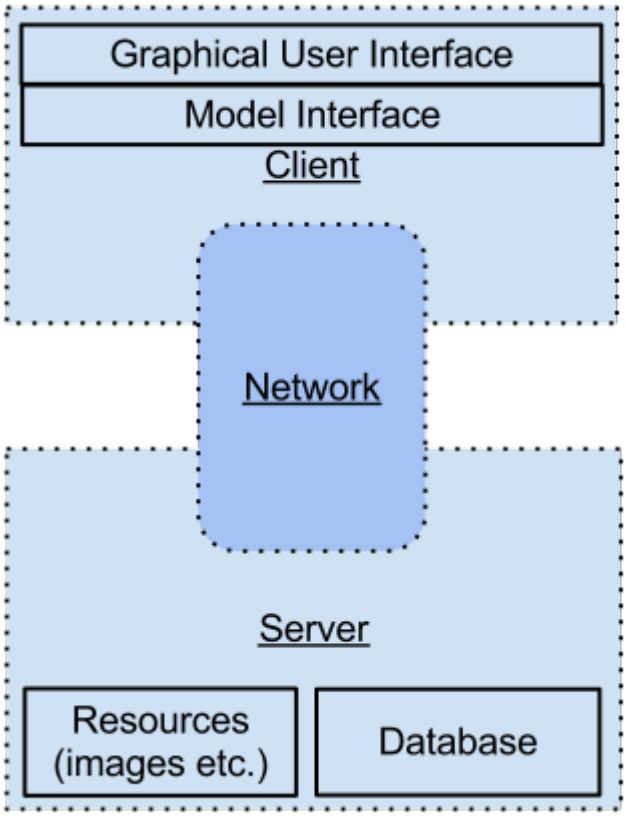
DOCUMENT HISTORY

VERSION	DATE	CHANGES	REVIEWED
1.0	2013-02-06	First version	Josef Gustafsson
1.1	2013-03-10	Updated	Sebastian Karlsson
2.0	2013-05-05	More info added and better reflect the software	Sebastian Wihlborg

1. INTRODUCTION

The purpose of this document is to describe the system elements as an abstraction of the source code. The goal of this document is to avoid poorly chosen, tactical decision regarding the system when implementing the solution.

2. DESIGN STRUCTURE



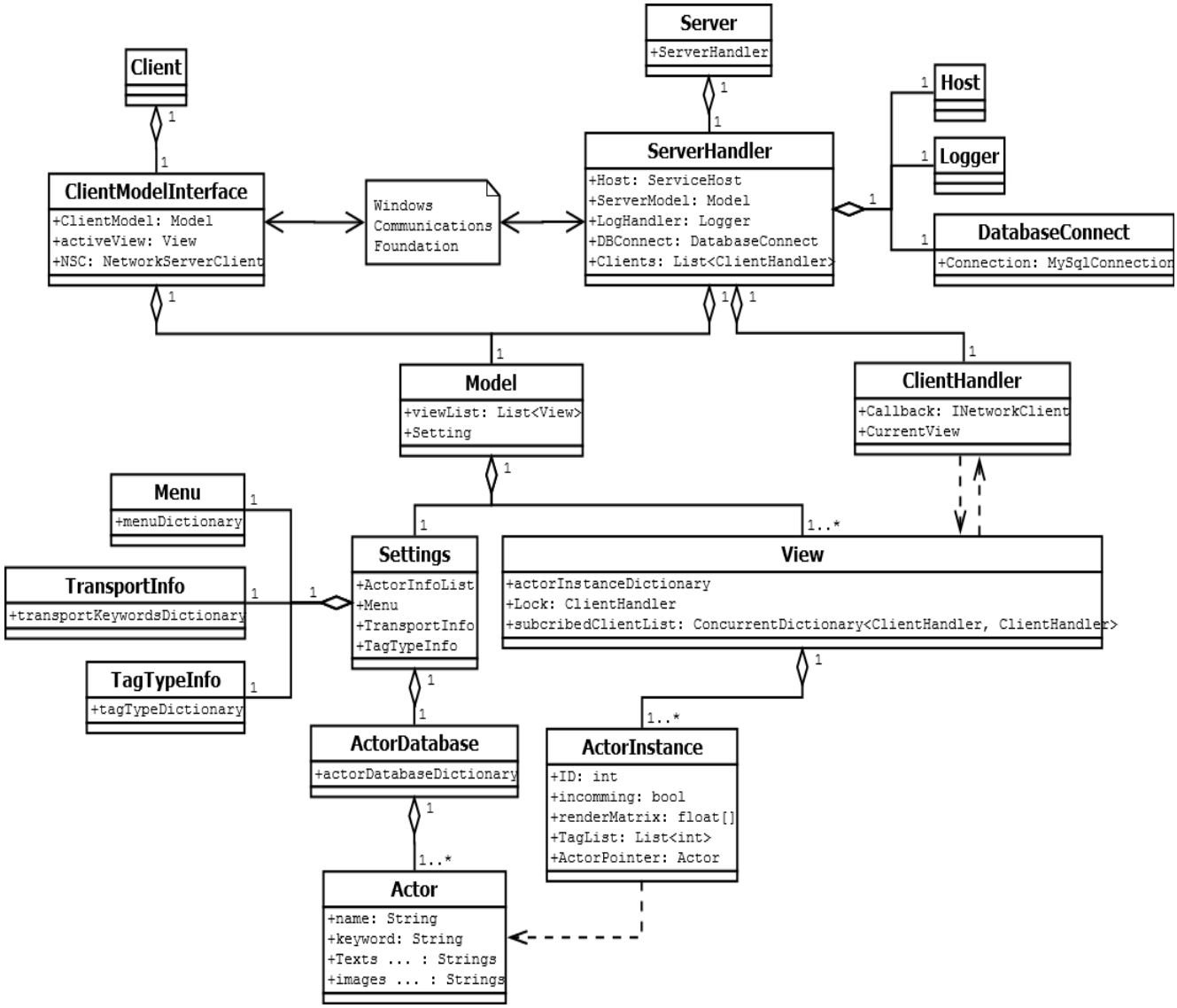
The drawing explains the basic concept of the system with three major subsystems and how they're interconnected. It also shows the different layers make up the client.

The model used in the MVC pattern is located in the server, however, there exists a local cache model in each client. The local model is used to manage all underlying information, but no changes are made to the model located in the server from the client. Instead the client makes a request to change an actors' location and the interface make the server request over the network. The server now decides whether or not the client is allowed to move the actor and if so send an update to all subscribed (explained later) clients.

3. SUBSYSTEMS

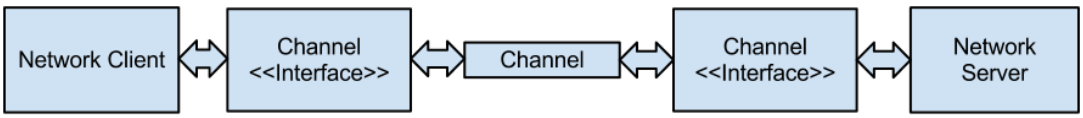
The system is divided into a back-end part, handling the server etc. and a front-end part, handling the GUI etc. Both the back-end part and the front-end part contain several subsystems. In this section all the subsystem from each part is listed and described.

3.1 Back-End



The UML mock-up shows how the different classes of the back-end system are interconnected.

3.1.1 Network



(For the channel interface please se Appendix)

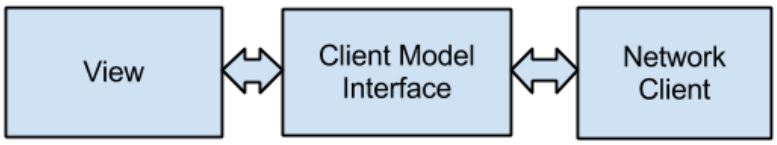
For this subsystem we are using WCF (Windows Communication Foundation). The network subsystem shall transfer information between client and server in a safe and efficient manner. The above drawing show how the channel interfaces moves the channel a layer down to simplify client and server communication by using a pre-specified protocol. The abstraction let's us specify which data to be sent

using a specific function. The channel is specified as full duplex which allows us to send and receive network messages asynchronously to each client.

Most of the interface methods does not return a value (they're void), which allows us to make a request and immediately forget about it on the client. The server responses are implemented using a callback channel, the methods in this channel are mostly all void too which the server uses when sending updates without a client requesting them.

3.1.2 Client (model interface)

(This part will not describe the graphical user interface, please look under the heading Graphical user interface)



The client consists of three main components; the network client, the model interface and the GUI. The networking parts' main focus is keeping local changes on the client up-to date on the server. The network client is then controlled and used by the model interface, which will make decision of what and when to send.

When connecting to the server the client receives a list of available views, these represent all (excluding special view, e.g. transport view) the available views, which can be subscribed and locked to.

Subscribing to a view means that this client will receive all changes made to the view, however subscribing to a view will not grant the client editing permission. To be able to change the model a client must first lock the view (Locking is described below). A client can be subscribed to several views at the same time, this gives us the ability to monitor any change, on any view. Using that feature one can construct a graphical representation of resources used on each view.

Locking a view is exclusive, meaning that no more than one client can have a lock on any view at any given time. The client may not lock more than one view at a time. Locking a view grants the client the rights to edit and change the view, or rather make the request to edit and change (the server is the only one which directly changes the model and the local cached model).

3.1.3 Server

The server responsibilities includes, but is not limited to; coordinating and controlling all connected clients, log actions, synchronize relevant model data to subscribed clients. The system uses server-based logging, a centralized log file is used to help a user analyze other users input. The server is responsible for parsing through all the posts in the database and dynamically creating needed objects using the flyweight pattern. Also the MVC model resides on the server, even though there's a local model on each client. The model on the clients only contains a subset of information contained in the servers complete model.

3.1.4 Database

As a part of the server, this system shall contain all information required to use the system for the specified scenario. The scenario is not part of the database, but all information, except scenario specific values, are contained within the database. The pattern flyweight is used for the connection between scenario and database (how is explained later on in this document).

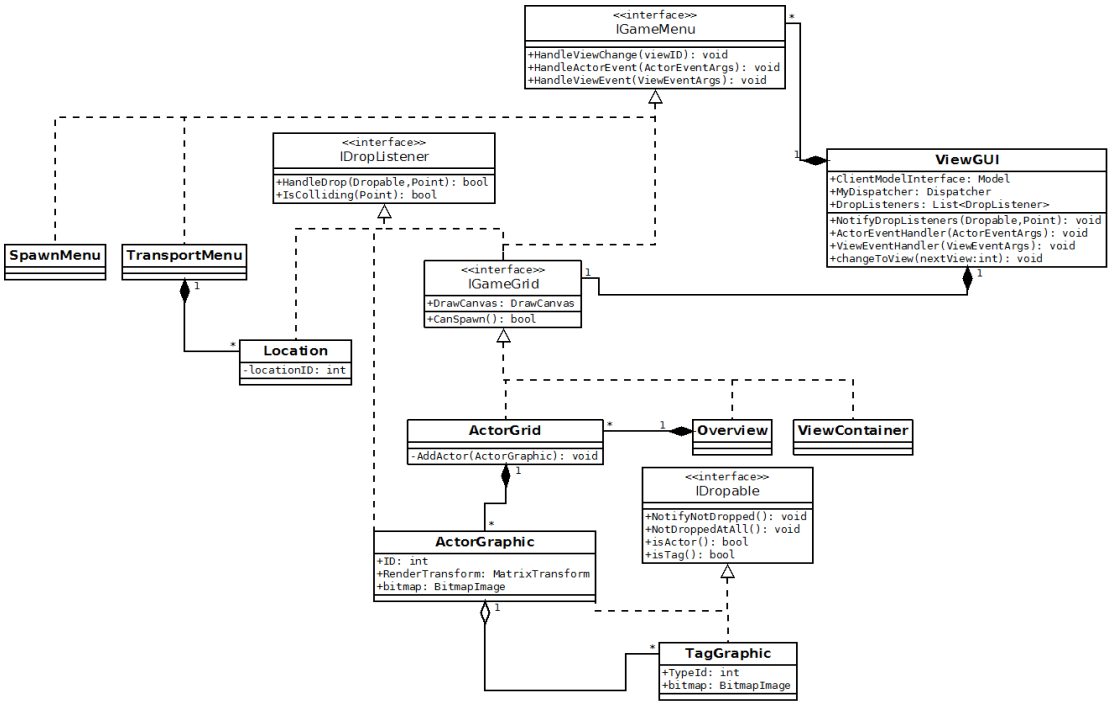
3.1.5 Scenario

This scenario file is located on the server. The information contained within this file describes scenario specific settings, e.g. which actor at which position, but does not contain information about the actor itself, that data resides inside the database.

3.2 Front-End

3.2.1 Graphical user interface

The GUI is built in WPF, Windows Presentation Foundation. It is quite complex and contains a lot of classes and interfaces. These are however the most important ones.



The base class of our client is ViewGUI, this object handles the communication with our model. ViewGUI holds a collection of WPF components implementing the IGameMenus interface. It is through these menus the player can interact with the game and get information from. When the model is changed the ViewGUI is notified via ActorEventHandler and ViewEventHandler, it relays these events to all of it's IGameMenus so they can react to it. A normal case of the listener pattern.
A special case of IGameMenu is the IGameGrid interface, this is the main grid; the game area. It can be either a normal game grid (ActorGrid) or a special grid like overview or ViewContainer. These objects are supposed to be WPF components one can put into a ScrollPanel.
In the case of a normal game grid then the IGameGrid can contain ActorGraphic objects. These are our visual representation of actors from the model. When we interact with them, like moving them, we notify the model so it changes itself and sync the change to the server. ActorGraphic objects can also have TagGraphic objects attached to them which are our visual representation for tags.

A system running through the whole GUI is the drop system. ActorGraphic and TagGraphic implements the IDropable interface, this means that when they think that they are dropped (last manipulator leaves them) they notify our ViewGUI's NotifyDropListeners function to tell that they have been dropped. This function iterates through all IDropListeners the ViewGUI knows of and ask them if they collides with the dropped object and if so if they want to do something with it and thus end the iteration. Many functions like tagging, removing tags, moving actors, transporting actors to other views and grouping actors are done via this system. This system can also be seen as a special form of the listener pattern.

4. PATTERNS

4.1 Observer

4.1.1 Overview
The listener pattern removes the need to polling. An object is a list of other objects that are interested in when it changes itself. When this happens it notifies them.

4.1.2 Structure
The object that we want to listen to has a list of listeners. In most cases the listeners are an interface with a function, notifyChanged(). All functions that change the listened object also calls a function that notifies all the listeners.

4.1.3 Behavior
Example: Change view
The user presses a Location button, which tells the ViewGUI object that we want to change view. Unless something goes wrong ViewGUI recreates it's ActorGrid and notifies all IGameGrid listeners by iterating through them and calling HandleViewChange(ID) so the menus can update to reflect that.

4.2 Flyweight

4.2.1 Overview
The flyweight pattern minimizes memory usage by sharing as much common information as possible between objects. This is possible because many objects will have data fields that are the same, for example all fireman instances will have the "fireman.pgn" string as an image name field.

4.2.2 Structure
Memory usage is minimized by using two objects instead of one; one "base" object and one "instance" object. The instance object knows what base object it's an instance of and ask that object when shared data needs to be accessed. For example, we have the base object "fireman", we can have several instance objects pointing to this object containing only unique data like position and rotation themselves. When we need to access shared data we access the base object from our instance object.

4.2.3 Behavior
We use the flyweight pattern for one of its' nice side effects, that the base data and the instance data are separated. This partitioning of data is natural when we have a database for storing the actor's base data and a scenario file where the instance data is stored, like starting position.
When we load a scenario the base objects are loaded from the database and the instance objects are loaded from the scenario file.

4.3 Model View Controller

4.3.1 Overview
Model View Controller is a popular pattern when one has a collection of data and several objects which can modify it or display it. The model has two interfaces, a view interface which provides a read only interface to the data contained in the model and a controller interface which can modify the data. All functions which modifies the data also notifies all Views listening to the model via the Observer pattern so they can update themselves to reflect the changes.

In our case the View and Controller has been merged into one large facade holding everything from data reading, data manipulating and some internal server communication functions.
The users interaction is separated from the information of the model. The model can only be changed by using the model interface layer in the client. The interface will also notify the view on any changes made to the model using events.

4.3.2 Structure
This pattern is made out of three participants; The model, the view and the controller. The controller is the component which updates and changes the model. The model in return updates the users view. Often the view and controller are embedded closely with each other, for example the GUI both displays (the view) and controls the changes (controller). In DigiMergo the underlying model will be made private and its data cannot directly be accessed and changed. For that we'll have the model interface which is used as a point of synchronization and buffer between the GUI and the server.
This implementation differs somewhat from the definition of the MVC pattern as the controller (in respect to the GUI) isn't really controlling the model. It's possible that we might have a controller controlling another controller inside the model interface, however this is a conscious choice as all changes will be controlled by the same portion of code that's keeping the rest of the system synchronized.

4.3.3 Behavior
The pattern radiates through the whole system and every significant action runs through it. Say for example that we want to have an actor we can move to another view. First, we have the model downloaded from the server containing a view with an actor on. The GUI creates a visual representation of this actor from data accessed via the view/controller facade. Using some WPF magic we can detect when the player have moved the actor into a new location. When this happens we send a *moveActor* function to our facade with the changed actor's ID and new position. If we have moved the actor into a component representing another location then we send a *sendActor* command instead. The *sendActor* command replies to our view via a *ACTOR_REMOVE* event. When this event is triggered we remove the actor from our view.
If another client is connected to the other view then it is also notified via an *actorEvent* so it can display the new transported actor as incoming.

Example: Moving an actor
The user sees an actor on the screen that he wants to move. This view is given from the model which contains the location of all screen objects. The user points and drags the object to its desired location. The controller updates the model with the required changes and the model in return notifies the view of a change and the object has been moved to its new position.

5. REQUIREMENT REALIZATIONS

5.1 The gameplay requirements

5.1.1 View of participants
The gameplay requirements are very much linked to the MVC pattern. The interface between the controller and the model is what decide what you are allowed to do with the model. However the model must have the capability of changing in the way that the controller wants to.

5.1.2 Basic scenario
Requirement 32.2: Patients and staff shall be able to be placed in a transport vehicle.

To realize this requirement the view needs to provide the user with the option to place a patient or staff into a transport vehicle. Then there must be command in the interface relating to that action. Finally the model of the transport vehicle must be able to have a patient or staff placed into it.

5.1.3 Additional scenarios
Requirement 29.2: Have a backside with information (ABCDE) to be visible by the action "examine".

This requirement refers to the patient actor. First we need a way for the user to perform the action "examine". Then we need the interface to be able to handle that action. Finally we need to be able to extract the ABCDE-information of that patient from the model and put it on the view.

5.2 The log requirements

5.2.1 View of participants

Almost every action that needs to be logged happens on a client. However the log is on the server. Therefore the client needs to notify the server when any significant action is performed.

5.2.2 Basic scenario

Requirement 45.1: Contain all the information including the start time, and end time for actions which take time, for practice starting, resource arrival, when sectors become available, treatment decisions, transports, treatments, priorities (triage), patient examinations, patient complications.

Since the system is built up the way it is. The client notifies the server when it changes the local model so that it can change the "real" model. Therefore we keep a log on the server that we update when we update the "real" model. The server also keeps track of the current game time and can easily put that into the log as well.

6. APPENDIX A

6.1 Network Client Interface (methods called by the server)

```
void pingClient();
void sendView(View view);
void addView(View view);
void removeView(int viewID);
void setActorListMatrices(List<int> actorIDList, List<float[]> renderMatrixList);
void addActorList(List<ActorInstance> actorList, int viewID);
void removeActorList(List<int> actorIDList, int viewID);
void setActorListActive(List<int> actorIDList);
void setActorTags(int actorID, List<int> tagList);
void setSettings(Settings Setting);
void setTime(DateTime serverTime);
void addMovingActors(MovingActors movingActors);
void removeMovingActors(MovingActors movingActors);
```

6.2 Network Server Interface (methods called by the client)

```
bool connectToServer();
void pingServer();
List<ViewInfo> getViewsInfo();
bool lockView(int viewID);
void unlockView(int viewID);
bool subscribeView(int viewID);
void unsubscribeView(int viewID);
void requestMoveActorList(List<int> actorIDList, List<float[]> matrix);
void requestSendActorList(List<int> actorIDList, int viewID);
void requestMoveActorListIntoView(List<int> actorID);
void requestAddTagToActor(int actorID, int tagType);
void requestRemoveTagFromActor(int actorID, int tagType);
void requestCreateNewActor(string name);
void requestDestroyActorList(List<int> actorIDList);
void requestAddActorListToRoom(List<int> actorIDList, int roomID);
void requestRemoveActorListFromRoom(List<int> actorIDList, int roomID);
```