

Design of the ActiveNU Client

Jan Vaughan, 4111997

Contents

1	Introduction	3
1.1	Purpose	3
1.2	Document Overview	3
1.3	Conventions In This Document	3
1.4	Terminology	3
2	Requirement Modifications	4
3	Architecture	5
4	Module Decomposition	6
4.1	ActiveNUClient	6
4.2	Tools	6
4.2.1	Tools.MapTool	6
4.2.2	Tools.ContactsTool	6
4.2.3	Tools.ContactsTool.Commands	6
4.2.4	Tools.ZoneAlertTool	7
4.2.5	Tools.BulletinTool	7
4.3	Model	7
4.4	Event	7
4.4.1	Event.Tools	7
4.4.2	Event.Model	7
4.5	Data	7
4.5.1	Data.Alert	7
4.5.2	Data.Contacts	7
4.5.3	Data.Map	8
4.5.4	Data.Messages	8
4.5.5	Data.Persistent	8
4.5.6	Data.Position	8
4.6	Provided	8

5	Class Decomposition & Interfaces	8
5.1	ActiveNUClient	8
5.2	Tools	9
5.2.1	Tools.MapTool	10
5.2.2	Tools.ContactsTool	11
5.2.3	Tools.ContactsTool.Commands	13
5.2.4	Tools.ZoneAlertTool	14
5.2.5	Tools.BulletinTool	16
5.3	Model	16
5.4	Event	19
5.4.1	Event.Model	20
5.4.2	Event.Tools	20
5.5	Data	21
5.5.1	Data.Alert	21
5.5.2	Data.Contacts	22
5.5.3	Data.Map	23
5.5.4	Data.Messages	24
5.5.5	Data.Persistent	25
5.5.6	Data.Position	26
5.6	Provided	26
6	GUI Callbacks	27
6.1	MainView	27
6.2	MapTool	27
6.3	ContactsTool	27
6.4	ZoneAlertTool	28
6.5	BulletinTool	28
7	Example Control Flows	28
7.1	Create a Geobulletin	28
7.2	Select a ZoneAlert Area	29
8	Design Decisions	29
8.1	Tool Independent Model	29
8.2	Tool Interactions	29
8.3	Common Data Structures	30
8.4	PositionalObjects	30
8.5	Alert Areas	30
A	Alphabetical Class Listing	31

1 Introduction

1.1 Purpose

This document presents the design of the ActiveNU Client application. This includes the client's architecture, along with all classes in the design, giving their interfaces and key attributes. Explanation of how this design achieves the required functionality and of the reasoning behind this design are also included.

1.2 Document Overview

Following this introduction and the conventions and terminology that will be used, this document contains the requirements specifications that have been modified to improve the clarity and correctness of the ActiveNU Client's Software Requirements Specification (SRS). This is followed by a brief description of the ActiveNU Client Architecture, before explaining each module in the design. The document then moves on into more depth, describing each class and its interface. Callbacks from the GUI and some example control flows follow, before it concludes with reasoning on the benefits of this design.

1.3 Conventions In This Document

The following notational conventions are used in this document:

Constant Width Used for code such as module, class and function names.

Italic Used for terms from the terminology glossary.

Underlined In lists of figures a class is shown in, indicates figures in which the class' interface is defined.

1.4 Terminology

The following terms used within this document have specific meaning in the context of the ActiveNU Client.

contact A person or group of people known to the user and stored within the ActiveNU Client.

everyone group A special group representing all people and groups within its context. Rather than being an actual **Group** object, it is a rule and will behave differently to normal **Groups** in some situations.

geobulletin A bulletin that is positioned at some geographic location.

tool An independent section of the ActiveNU Client's user interface through which related tasks can be performed.

2 Requirement Modifications

2.3 The set of available tools is

1. map
2. contact manager
3. zone alert
4. geobulletin

Multiple instances are allowed for the map tool but not for any other tools. A further set of administration tools is needed elsewhere but is not part of this client application.

3.1 A map covers a geographic zone.

3.2 A geographic zone is a roughly rectangular¹ area of the Earth's surface defined by the location of two opposite corners. These corners may be specified absolutely or relatively to the client's current location.

3.5.1 The user shall be able to create a map tool with default scale and default zone.

3.6.1 A positioned object places an object at a geographic location. It has a centre location; an orientation (in degrees); a colour; an optional display font, size and style; and a string of characters or a graphic.

4.2 A friends and groups tool shall allow the user to manage an ordered contacts list.

4.3 A contacts list shall associate a name (and an optional graphic icon) with either an individual UniID or a group.

4.6 Groups shall contain ordered lists of individual names, UniIDs, and other named groups without cycles.

4.7 The friends and groups tool shall allow the user to specify that their location reports shall be private, or open to particular users or groups in the contacts list, or open to all other users. A personal setting takes precedence over a group setting, which takes precedence over the general setting. In the case where two settings of the same type conflict, the one occurring earlier in the list will take precedence.

6.2 In the case that the user does not have a global open visibility set and a permission setting is needed for a person not in the user's contacts list, the client will prompt the user for what visibility permission to use.

¹Rectangular were the map a flat projection.

7.4 Geobulletins shall be selectively visible in a map, provided that the geobulletin location is within the map's zone. The user can control which geobulletins are visible on a particular map with a timestamp range and by selection of authors and groups of authors from their contacts list. They may also select all or no authors.

3 Architecture

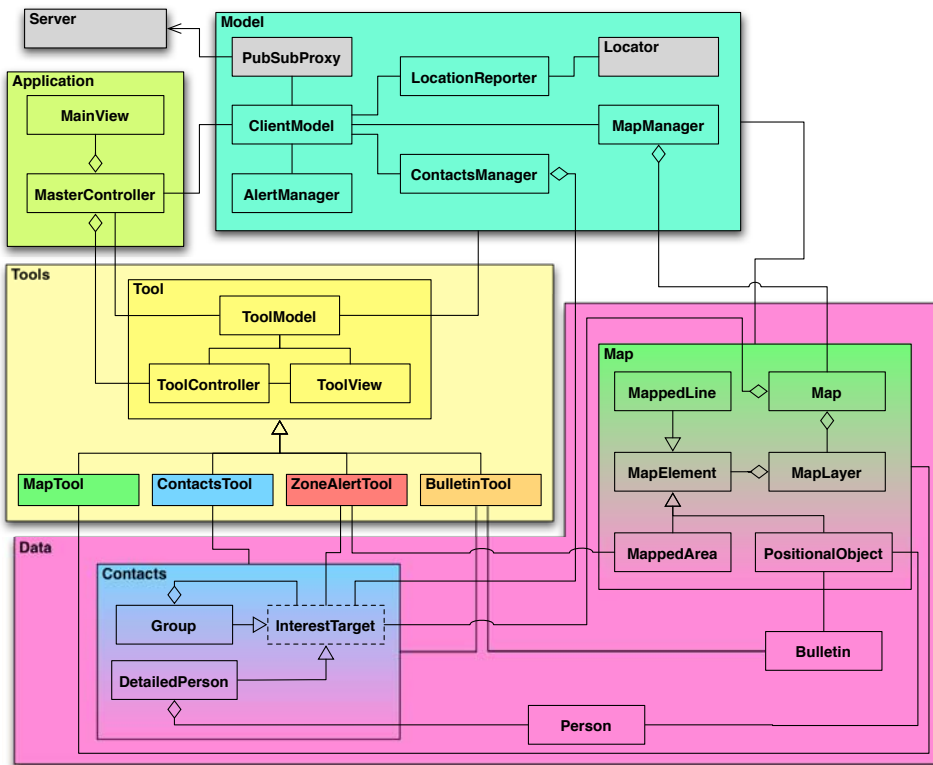


Figure 1: Architecture of the ActiveNU Client.

The architecture shown in figure 1 is designed to provide a simple yet powerful and easily modifiable structure for the ActiveNU Client. This was deemed important due to the high probability that the ActiveNU system will evolve substantially over time.

The architecture contains a single model at the core, with a series of independent *tools* allowing the user to view and manipulate the model. Each *tool* follows the idea of a model-view-controller, with the model of each *tool*

containing information specific to that *tool* and also acting as a façade in front of the core model.

4 Module Decomposition

From the architecture in figure 1, the ActiveNU Client is further decomposed into the following modules. This listing includes modules that are specific to this implementation of the architecture and hence not shown in figure 1.

4.1 ActiveNUClient

Classes of the application that gather together the other modules to provide the ActiveNU Client's full functionality. The main entry point for the program is within this module.

4.2 Tools

A collection of independent UI modules for viewing and interacting with the model. Each *tool* is focused on a particular part of the application's functionality.

4.2.1 Tools.MapTool

The `MapTool` renders and displays a `Map` that exists in the `ClientModel`, along with allowing that `Map` to be modified such as zoomed and panned. It also allows the visibility of items on the `Map` such as people and *geobulletins* to be controlled. Finally, it provides events to the `MasterController` when the `Map` is clicked so that other *tools* can perform any required actions.

4.2.2 Tools.ContactsTool

The `ContactsTool` lets the user organise their contacts within the system. This includes a structure of `Groups` that can contain people and other `Groups`, and also allows names and pictures to be associated with people and groups.

This *tool* also controls the user's visibility to other people using the ActiveNU system. This can be done on a person by person basis, or to whole `Groups` at a time. There is also an *everyone group* which allows global privacy options to be set.

4.2.3 Tools.ContactsTool.Commands

Classes representing the various commands that can be performed through the `ContactsTool`.

4.2.4 `Tools.ZoneAlertTool`

The `ZoneAlertTool` allows the user to set up alerts for when events of interest occur. This includes being alerted when certain `InterestTargets` enter a chosen area of the map and when a certain threshold number of `InterestTargets` are present in an area. The area may be any `MappedArea`.

4.2.5 `Tools.BulletinTool`

The `BulletinTool` allows the user to view the content of `Bulletins`, along with editing the content of their own `Bulletins`.

4.3 `Model`

The `Model` controls the application's core data and operations. The *tools* indirectly use it as much of the model in their model-view-controller architectures via the façade provided by each `ToolModel`. It also conceals the manner in which clients exchange information from the rest of the application — the classes outside the `Model` do not need to know, for instance, that a single server is in use rather than multiple servers, or even what data is provided by the server and what is provided locally in the `Model`.

4.4 `Event`

Classes from the event-listener system used for communication of events that occur within the client.

4.4.1 `Event.Tools`

Events used for communication between the *tools*.

4.4.2 `Event.Model`

Events used for communication between parts of the model and data structures, and from the model up to the *tools*.

4.5 `Data`

Data structures for the information that the ActiveNU Client deals with.

4.5.1 `Data.Alert`

Classes to hold information about events the user would like to receive alerts for, and also to determine when these events occur.

4.5.2 `Data.Contacts`

Classes to organise the user's contacts and store settings such as whether these people can see the user.

4.5.3 Data.Map

Data structures to describe the maps and their content.

4.5.4 Data.Messages

Object oriented forms of the messages that go between the client and server.

4.5.5 Data.Persistent

Data structures to serialise for persistent storage of client information.

4.5.6 Data.Position

Various ways to describe a position on the map, such as absolutely, in relation to another position or as the user's current location.

4.6 Provided

All parts of the system that already exist and are provided.

5 Class Decomposition & Interfaces

From the modules listed in section 4, the design of the ActiveNU Client breaks down into the following classes. For an alphabetical listing of classes showing which class appears in which module, see appendix A. Methods of interest are explained here, for a full listing of the class interfaces see the UML diagram whose number is underlined in the figures list for that class.

5.1 ActiveNUClient

MasterController

Figures 2, 3

The main entry point of the ActiveNU Client application. It sets up the model and all *tools*. It also controls the `MainView`, performing actions such as creating new maps and changing between *tools*.

The `MasterController` also controls the connection between *tools* in such a way that no *tool* needs to know about any of the other *tools*. It receives events when actions that do not affect the model but may be of interest to other *tools* occur, and sends appropriate messages to the *tools* it knows are interested in such actions.

MainView

Figures 2

The main view of the ActiveNU Client, providing widgets to create new maps and move between the various *tools*.

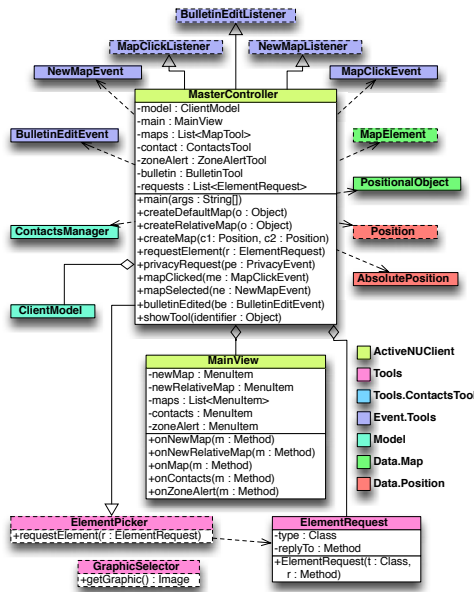


Figure 2: General classes of the ActiveNUClient and Tools modules.

5.2 Tools

ToolController

Figures 3

The controller for a *tool*, responsible for creating and controlling the *tool*'s *ToolView*. It is given the *ToolModel* rather than creating it itself, as it does not know of the *ClientModel*.

ToolModel

Figures 3

The model for a *tool*. It contains data specific to that *tool*, along with acting as a façade in front of the *ClientModel*.

ToolView

Figures 3

The view for a *tool*, displaying data from that *tool*'s *ToolModel*.

ElementPicker

Figures 2, 8

A class that provides a service of allowing the user to, by some means, pick an element.

`requestElement(r : ElementRequest)` Request that the user pick an element of the type specified in *r*.

ElementRequest

Figures 2, 8

A form in which it can be requested that the user pick an element.

`ElementRequest(t : Class, r : Method)` Create a new message that can be used to request an element of class *t* be picked by the user, specifying that the element the user picks be sent to the method *r*.

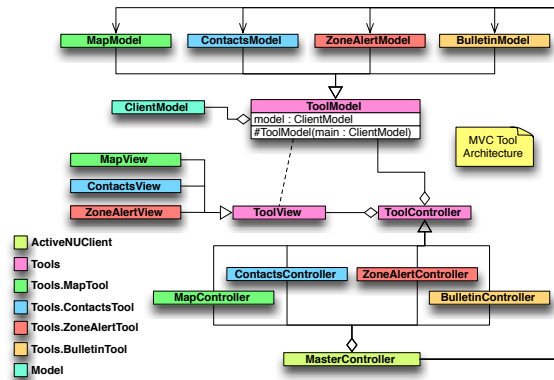


Figure 3: Structure of the *tools*.

GraphicSelector

Figures 2, 6, 9

A class that allows an image to be picked. The details of how this is actually done are left unspecified, as it will depend substantially on the platform the ActiveNU Client is running on. Examples of possible methods are through a standard “open file” dialog or by selecting from a list of default images within the application.

5.2.1 Tools.MapTool

MapController

Figures 3, 5

The controller for the MapTool. It will fire `MapClickEvents` when the map is clicked and `NewMapEvents` when the user selects a new zone from within the current map.

`pickElement(type : Class)` Ask the MapTool to get the user to pick an object of class `type`. A `MapClickEvent` will be fired when the user picks one.

MapModel

Figures 3, 4, 5

The model for a single MapTool. It acts partially as a façade in front of a `Map`, with a number of methods of the same names. It also contains some information specific to the display of the Map, such as the map’s zoom and *geobulletin* interest settings.

MapView

Figures 3, 5

A display of a `Map`, along with providing access to a set of controls that can be used to adjust the Map.

InterestEditorController

Figures 5

Allows the user to control the map’s settings for the visibility of people and *geobulletins*.

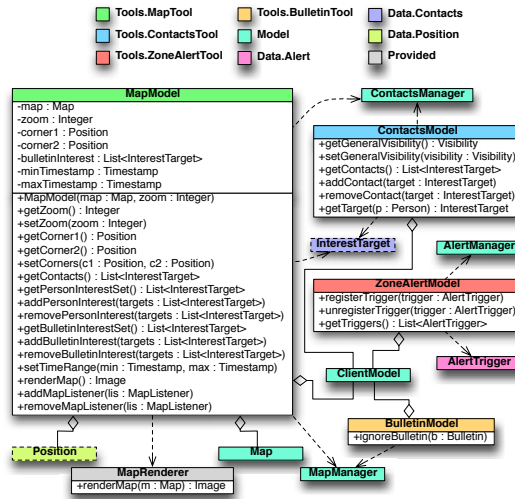


Figure 4: Definitions of the ToolModels.

InterestEditorView

Figures 5

A view of the map’s settings for the visibility of people and *geobulletins*. It contains lists of interest sets for people and *geobulletin* authors, along with a timestamp range that *geobulletins* must fall within to be displayed.

ClickInfo

Figures 5

Information about a click on the map, including the **Position** that was clicked and a list of **MapElements** at that location.

DragInfo

Figures 5

Information about the start and end **Positions** of a drag of the map.

5.2.2 Tools.ContactsTool

ContactsController

Figures 3, 6

The controller of the **ContactsTool**. It is an **Invoker**, populating and performing commands given to it by other parts of the **ContactsTool**. To do this, it tracks what **InterestTarget** is currently selected in the **ContactsList**. It also maintains a history of **Commands** that can then be undone.

ContactsModel

Figures 3, 4, 6

The model for the **ContactsTool**, which is implemented as a façade in front of the **ContactsManager**.

ContactsView

Figures 3, 6

The view of the **ContactsTool**, showing the user’s *contacts* and allowing them to view and edit individual *contacts*’ details.

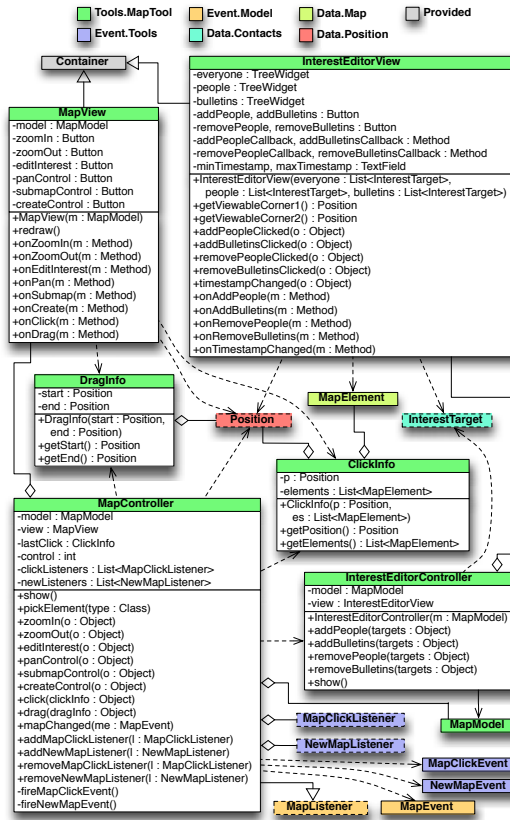


Figure 5: MapTool design and details.

ContactsList

Figures 6

A list of **InterestTargets**, allowing the user to delete **InterestTargets** from the list and create and add new **Groups** and **DetailedPersons** to it.

`restrict(search : String)` Restrict the people shown in the list to those matched by the search string.

ContactsEditor

Figures 6

A container holding the GUI widgets required to show and edit an **Interest-Target**, including extra widgets for **Groups** and **DetailedPersons**.

GroupContentEditor

Figures 6

A container holding the GUI widgets required to show and edit the content of a **Group**.

PrivacyRequestDialog

Figures 6

A dialog box requesting that the user set a privacy preference for a certain **InterestTarget**.

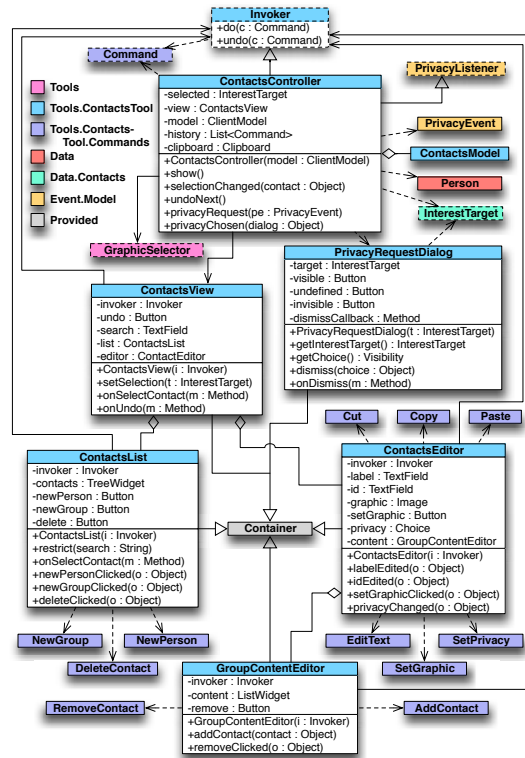


Figure 6: ContactsTool design and details.

Invoker Figures 6

A class that can fill in any information missing from a Command and then do() or undo() it.

5.2.3 Tools.ContactsTool.Commands

Command Figures 6, 7

The command class within the command pattern, for operations that can be performed within the ContactsTool.

NewContact Figures 7

Command to create a new InterestTarget of some form.

NewGroup, NewPerson Figures 6, 7

Commands to create new Group and DetailedPerson contacts.

DeleteContact Figures 6, 7

Command to delete an InterestTarget.

ModifyGroup Figures 7

Command that modifies the content of a Group.

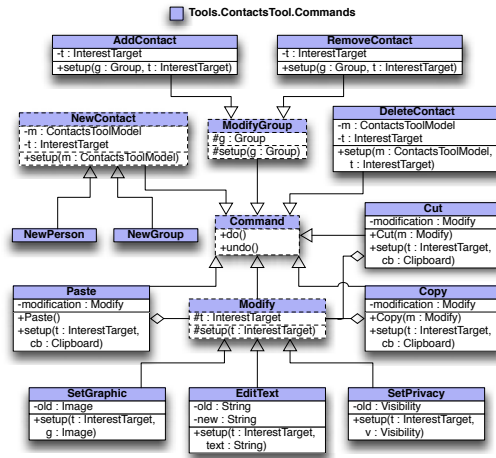


Figure 7: Commands available in the ContactsTool.

- AddContact, RemoveContact** Figures 6, 7
 Commands to add/remove an InterestTarget to/from a Group.
- Modify** Figures 7
 Command that modifies a property common to all InterestTargets.
- EditText** Figures 6, 7
 Command that edits a textual part of an InterestTarget.
- SetGraphic, SetPrivacy** Figures 6, 7
 Commands that change the graphic and privacy preference associated with an InterestTarget.
- Cut** Figures 6, 7
 Command that performs some Modify command, placing content modified by that command on a clipboard.
- Copy** Figures 6, 7
 Command that places the content that would be modified by some Modify command on a Clipboard.
- Paste** Figures 6, 7
 Command that performs some Modify command, determining the details of what is modified by using data on a Clipboard.

5.2.4 Tools.ZoneAlertTool

ZoneAlertController Figures 3, 8
 The controller of the ZoneAlertTool. Upon creation it is given an ElementPicer so that it can use another part of the ActiveNU Client to pick MappedAreas for the AlertProcessors.

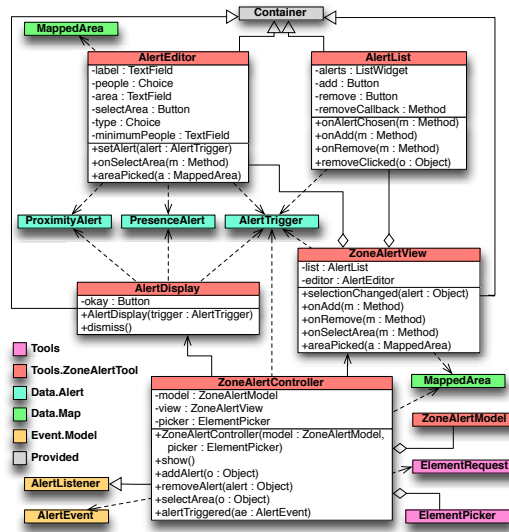


Figure 8: ZoneAlertTool design and details.

`selectArea(o : Object)` Creates an `ElementRequest` for a `MappedArea`, requesting that the selected `MappedArea` be sent to the `ZoneAlertView.areaPicked(Object)` method.

ZoneAlertModel

Figures 3, 4, 8

The model for the `ZoneAlertTool`, implemented as a façade in front of the `AlertManager`.

ZoneAlertView

Figures 3, 8

The view of the `ZoneAlertTool`, presenting the user's `AlertTriggers` along with allowing them to be edited and new ones to be created.

AlertList

Figures 8

A list of `AlertTriggers` that have been set up, allowing new triggers to be created along with selecting old triggers for editing or deletion.

AlertEditor

Figures 8

A container holding GUI widgets to display and allow the editing of an `AlertTrigger`. This includes editing the `AlertProcessor` data of `AlertTriggers` that are using `ProximityAlerts` and `PresenceAlerts`.

AlertDisplay

Figures 8

A dialog for displaying that an `AlertEvent` has occurred, along with some details from that `AlertEvent`.

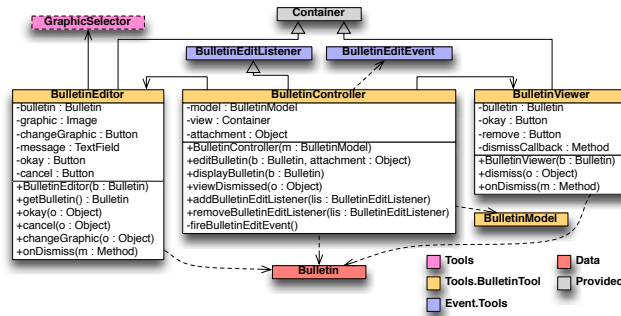


Figure 9: BulletinTool design and details.

5.2.5 Tools.BulletinTool

BulletinController

Figures 3, 9

The controller for viewing and editing Bulletins.

`editBulletin(b:Bulletin, attachment:Object)` Allow the user to edit `b`. A `BulletinEditEvent` is fired when the user has finished editing `b`. The `attachment` is returned with that event and is otherwise unused.

`displayBulletin(b:Bulletin)` Show the user the content of `b`.

`viewDismissed(o:Object)` Called when the `BulletinViewer` or `BulletinEditor` is dismissed, to remove them from the main `BulletinTool` view.

BulletinModel

Figures 3, 4, 9

A façade in front of the Model. The only method, `ignoreBulletin(Bulletin)`, corresponds to same method in the `MapManager`.

BulletinEditor

Figures 9

A view for editing the author, text and graphic of a `Bulletin`.

BulletinViewer

Figures 9

A view for displaying a `Bulletin`, allowing the view to be dismissed or closed in such a way that the `Bulletin` it was showing will no longer appear on Maps.

5.3 Model

ClientModel

Figures 2, 3, 4, 10, 18

The main part of the model that groups together all general data in the ActiveNU Client. It is a `MessageHandler` to deal with messages it gets from the `ServerProxy` it creates. It can be constructed from scratch or from an existing `PersistentModel`.

`onMessage(message:IncomingMessage)` Receive `IncomingMessage` and forward it to any other parts of the model that are interested in messages of that type.

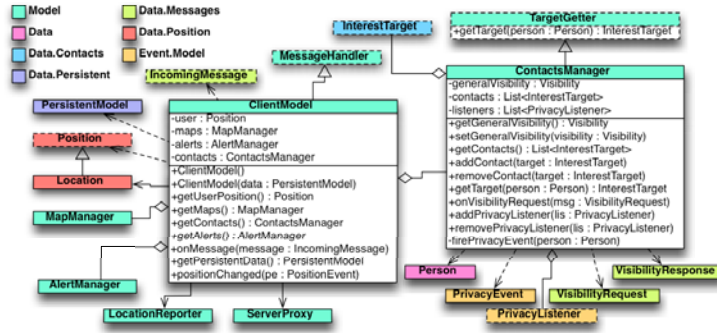


Figure 10: ClientModel and ContactsManager interactions.

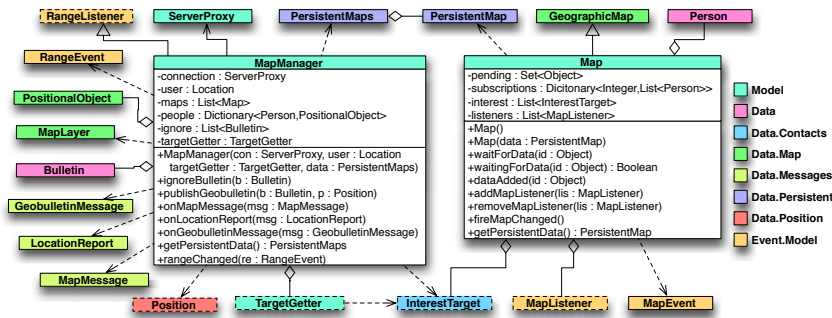


Figure 11: Map handling in the Model.

MapManager

Figures 4, 10, 11

Organises all Maps in the system. This includes populating them with MapElements, including PositionalObjects for people and *geobulletins*.

`ignoreBulletin(b : Bulletin)` Add the given Bulletin to a list of ones to ignore and hence not place on Maps, generally as it has already been viewed by the user.

`onMapMessage(msg : MapMessage)` Receive a MapMessage from the server. It places any data from that message into the Maps that were waiting for it, as determined by the `Map.waitForData(Object) : boolean` method.

`onLocationReport(msg : LocationReport)` Receive a LocationReport from the server. The MapManager updates the position of any existing PositionalObjects it knows of that represent the Person in the message, and creates and stores a new PositionalObject for that Person if they now fall within one of the MapManager's Maps.

`rangeChanged(re : RangeEvent)` Called when the range of a Map the MapManager is organising changes. It requests any extra information from the server that is needed for this new range, marking the map as needing that information by using the `Map.waitForData(Object)` method.

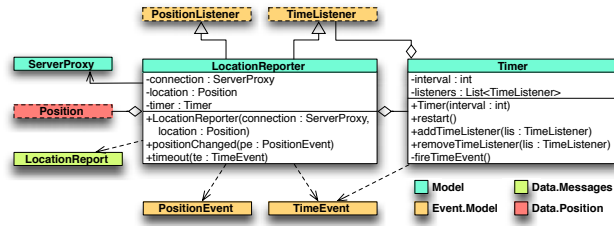


Figure 12: Model section to report the user’s location to the server.

Map

Figures 4, 11

Map is a subclass of GeographicMap that contains information more specific to this implementation. A Map can be constructed from scratch, or from existing persistent information. Due to the asynchronous manner in which map information is retrieved from the server, Map includes the following methods for tracking data requests:

`waitForData(id: Object)` Mark that the Map is waiting for a data message with the given id.

`waitingForData(id: Object): Boolean` Check if the Map is waiting for a data message with the given id.

`dataAdded(id: Object)` Indicate that data from a message with the given id has been added to the Map and it no longer needs to wait for it.

ContactsManager

Figures 2, 4, 10

Organises the user’s contacts, in particular tracking the user’s global visibility setting and their visibility preference for all their *contacts*.

`onVisibilityRequest(msg: VisibilityRequest)` Receive a `VisibilityRequest` from the server. If a visibility is already specified for that *contact* then it sends that choice to the server. Otherwise it fires a `PrivacyEvent` to all subscribed `PrivacyListeners` so that another part of the ActiveNU Client can determine a visibility preference for that *contact*.

AlertManager

Figures 4, 10, 16

Organises alerts the user has set up in the form of `AlertTriggers`, and provides these triggers with the information to determine when the situation they are looking for occurs.

`onLocationReport(msg: LocationReport)` Receive a `LocationReport` message, forwarding it to all triggers via the `AlertTrigger.processMove(LocationReport)` method so they can determine if the new location of people sets off the trigger.

LocationReporter

Figures 10, 12

Reports heartbeats and changes in the client’s location to the server.

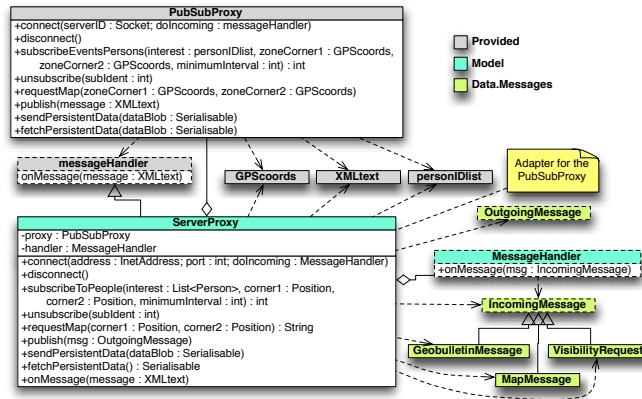


Figure 13: Design of the server interaction interface.

TargetGetter

Figures 10, 11

A class that can retrieve the InterestTarget for a Person.

`getTarget(person : Person) : InterestTarget` Return an InterestTarget corresponding to this Person, or null if one does not exist.

Timer

Figures 12

A timer that fires TimeEvents to all registered TimeListeners at a specified interval.

`Timer(interval : int)` Start a new Timer firing events every interval seconds.

`restart()` Restart the Timer such that the next event occurs interval seconds from the time this method was called.

ServerProxy

Figures 10, 11, 12, 13

An adapter in front of PubSubProxy that brings the interface to the server into conformance with the ActiveNU Client conventions and data structures. It also may fake some functionality expected of the server that is not available through the given PubSubProxy.

`requestMap(corner1 : Position, corner2 : Position) : String` Request map information within the given area. An identifier that the MapMessage reply will have is returned.

MessageHandler

Figures 10, 13

A class that can receive IncomingMessages.

5.4 Event

Event

Figures 14

A class of objects indicating when events occur within the system.

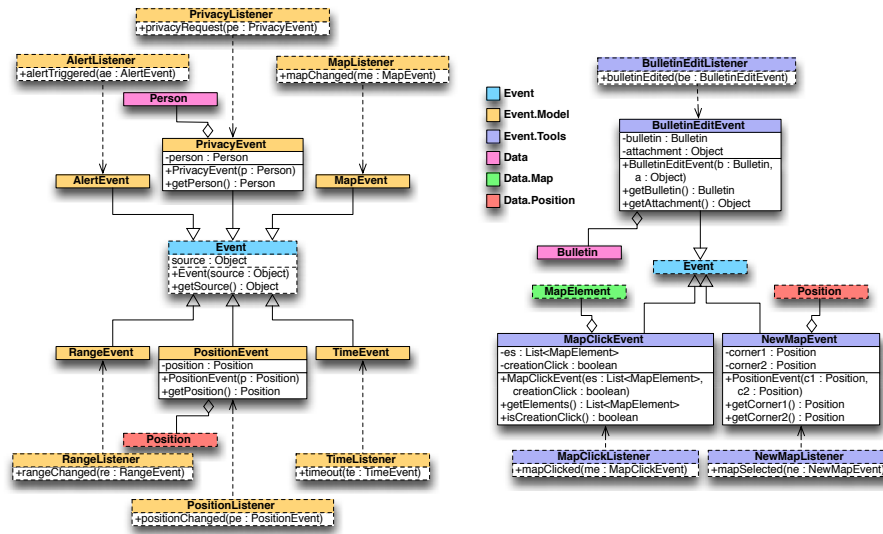


Figure 14: Model and Tools events in the ActiveNU Client design.

5.4.1 Event.Model

- PositionEvent, PositionListener** Figures 12, 14, 20
 Events and listener relating to Positions.
- MapEvent, MapListener** Figures 5, 11, 14
 Events and listener relating to a Map changing.
- RangeEvent, RangeListener** Figures 11, 14, 17
 Events and listener relating to a range changing, such as a Map's range.
- PrivacyEvent, PrivacyListener** Figures 6, 10, 14
 Event that the user needs to specify their visibility to a Person, along with an associated listener.
- AlertEvent, AlertListener** Figures 8, 14, 16
 Event that an alert condition has occurred, such as an AlertTrigger going off, along with an associated listener.
- TimeEvent, TimeListener** Figures 12, 14
 An event and listener relating to time.

5.4.2 Event.Tools

- MapClickEvent, MapClickListener** Figures 2, 5, 14
 Event indicating that the map was clicked, along with a listener for it.
- `getElements() : List<MapElement>` Get a list of the MapElements on the map at the location that was clicked.

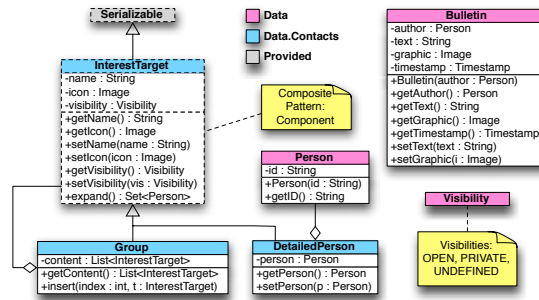


Figure 15: **Contacts** and other basic data structures.

`isCreationClick() : boolean` Find out if the click should create a new object of some form on the map.

`NewMapEvent`, `NewMapListener` Figures 2, 5, 14

Event indicating that a new map should be created, with two of its corners as those in the `NewMapEvent`, along with an associated listener.

`BulletinEditEvent`, `BulletinEditListener` Figures 2, 9, 14

Event indicating that a bulletin has been edited, along with an associated listener. The event also has room to hold some extra information, retrievable using the `getAttachment() : Object` method, that may be relevant to the `Bulletin`.

5.5 Data

`Person` Figures 6, 10, 11, 14, 15, 16, 18

A data structure for a basic person, not necessarily one of the user's `contacts`. The physical person it corresponds to is uniquely identified by their university id.

`Visibility` Figures 15

A simple enumerated type of the possible settings for whether the user may be seen or not. The possible values are `OPEN`, `UNDEFINED` and `PRIVATE`. `UNDEFINED` indicates that a more general setting should be used to determine if the user may be seen or not.

`Bulletin` Figures 9, 11, 14, 15

A message from some author that has a timestamp and can contain text and a graphic.

5.5.1 Data.Alert

`AlertTrigger` Figures 4, 8, 16

Represents an alert that should fire an event to all subscribed listeners when some condition described by an `AlertProcessor` is met.

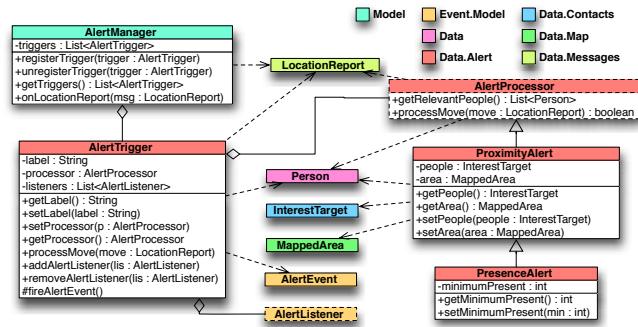


Figure 16: Data structure for tracking and processing alerts.

`processMove(move : LocationReport)` Determine if the given move triggers the alert, firing an `AlertEvent` to all subscribed `AlertListeners` and disabling this alert from going off again if it does.

AlertProcessor

Figures 16

A class that can determine when some form of alert is triggered.

`getRelevantPeople() : List<Person>` Get a list of the people whose positions influence the triggering of this alert.

`processMove(move : LocationReport) : boolean` Determine if the given move triggers the alert, returning true if it does or false otherwise.

ProximityAlert

Figures 8, 16

An `AlertProcessor` that determines when a `Person` from the `ProximityAlert`'s `InterestTarget` moves within the bounds of the `ProximityAlert`'s `MappedArea`.

PresenceAlert

Figures 8, 16

An `AlertProcessor` that determines when at least some minimum number of people (`minimumPresent`) from the `PresenceAlert`'s `InterestTarget` are within the bounds of its `MappedArea`.

5.5.2 Data.Contacts

InterestTarget

Figures 4, 5, 6, 10, 11, 15, 16, 19

Some set of *contacts* that can be expanded to a `Set` of `Person` objects. The `InterestTarget` also tracks whether these people may see the user's location, along with associating optional extra information such as a name and icon with those people. `InterestTarget` plays the role of component in the composite pattern.

Group

Figures 15

An `InterestTarget` that groups together other `InterestTargets`.

DetailedPerson

Figures 15

An `InterestTarget` representing a single `Person`.

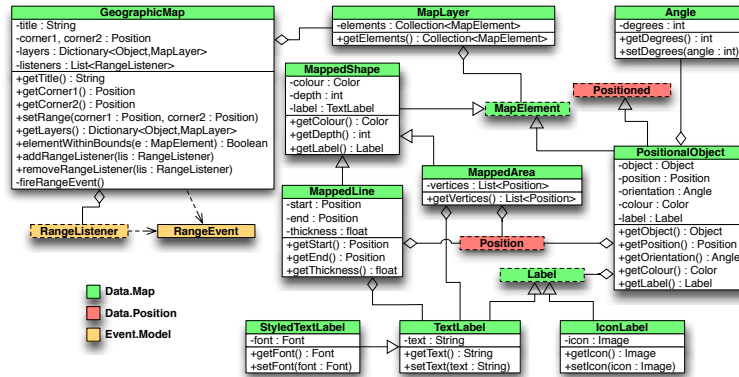


Figure 17: Basic data structures for geographic maps.

5.5.3 Data.Map

GeographicMap

Figures 11, 17

A data structure representing a basic, geographic map. This does not include implementation specific details such as the map’s interest set — for that, see the Map class.

`elementWithinBounds(e : MapElement) : boolean` Return true if the given MapElement is within this GeographicMap’s range, false otherwise.

MapLayer

Figures 11, 17

A grouping of MapElements.

MapElement

Figures 2, 5, 14, 17, 18

An element that can occur in a map.

MappedShape

Figures 17

A shape that can occur in a map, with the visual properties of colour, depth and an optional textual label in the form of a TextLabel. The label will be null to indicate no label.

MappedArea

Figures 16, 17

A polygonal area on a map, specified by its vertices.

MappedLine

Figures 17

A line on a map, specified by its start and end.

PositionalObject

Figures 11, 17, 18

A PositionalObject places any object on the map at a Position and with a Angle of orientation, also associating visual properties such as a label (in the form of a Label) and colour with it.

Angle

Figures 17

A simple data structure representing an angle in degrees.

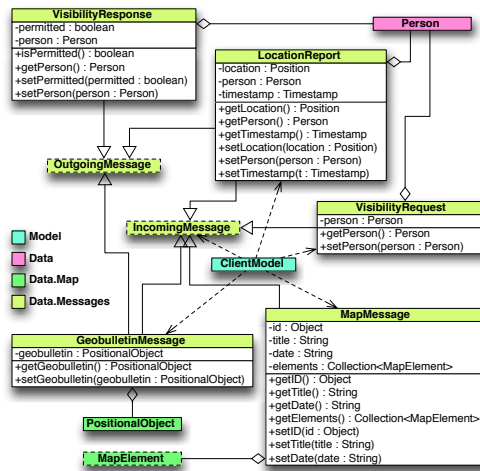


Figure 18: Classes of client/server communication messages.

Label Figures [17](#)

A Label is an object that labels things in some way.

TextLabel, StyledTextLabel Figures [17](#)

Plain and styled text Labels.

IconLabel Figures [17](#)

A Label that is a single image.

5.5.4 Data.Messages

IncomingMessage Figures [10](#), [13](#), [18](#)

An object oriented representation of a message coming from the server to the client.

OutgoingMessage Figures [13](#), [18](#)

An object oriented representation of a message transmitted from the client to the server.

MapMessage Figures [11](#), [13](#), [18](#)

A IncomingMessage with information for a map.

VisibilityRequest Figures [10](#), [13](#), [18](#)

A IncomingMessage asking whether a person should be able to see the user's location.

LocationReport Figures [11](#), [12](#), [16](#), [18](#)

A message sent by both the client and the server reporting a person's location.

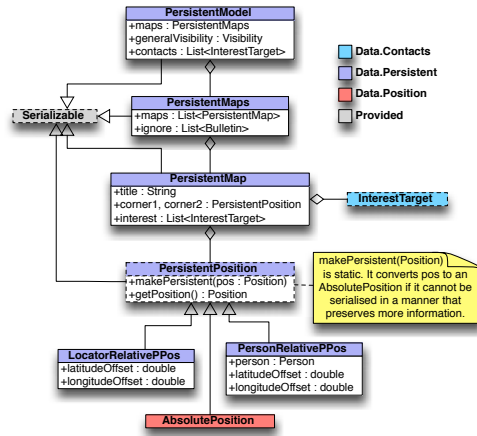


Figure 19: Persistent data structures for preserving the Model.

GeobulletinMessage

Figures 11, 13, 18

A message sent by both the client and the server specifying the location and content of a *geobulletin*.

VisibilityResponse

Figures 10, 18

An *OutgoingMessage* informing the server of whether a person should be able to see the user’s location.

5.5.5 Data.Persistent

PersistentModel

Figures 10, 19

Data from the ActiveNU Client that must persist over multiple sessions.

PersistentMaps

Figures 11, 19

Data from the *MapManager* that must persist over multiple sessions.

PersistentMap

Figures 11, 19

Data of a Map that must persist over multiple sessions.

PersistentPosition

Figures 19, 20

A *Position* in a form able to be serialised.

makePersistent(pos : Position) A static method that converts *pos* to a *PersistentPosition* best describing the type of *Position* *pos* is. This will default to being an *AbsolutePosition* of *pos*’s current coordinates if it cannot be converted to a serialisable form that preserves more information. This is a factory method so that *Position* objects can be converted to *PersistentPosition* objects without the original *Position* objects needing to have any knowledge of *PersistentPositions*.

getPosition() : Position Get a *Position* of the type that this *PersistentPosition* describes.

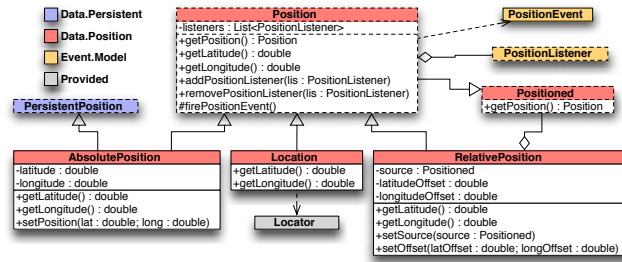


Figure 20: Various representations of locations on the Earth’s surface.

PersonRelativePPos

Figures 19

A `PersistentPosition` whose position is relative to the contained person’s current location.

LocatorRelativePPos

Figures 19

A `PersistentPosition` whose position is always relative to the user’s current location.

5.5.6 Data.Position

Positioned

Figures 17, 20

Objects of class `Positioned` have a `Position`.

Position

Figures 2, 4, 5, 10, 11, 12, 14, 17, 20

A data structure representing a physical location on the Earth’s surface that can be retrieved as decimalised latitude and longitude coordinates to an accuracy of two metres. It allows `PositionListeners` to subscribe to receive events when this location changes.

AbsolutePosition

Figures 19, 20

A `Position` specifying a fixed location on the Earth’s surface.

RelativePosition

Figures 20

A `Position` specified by an offset from some `Positioned` object.

Location

Figures 10, 20

A `Position` for the current location of the user.

5.6 Provided

Provided as part of the server communication proxy are `PubSubProxy`, `GPSCoords`, `messageHandler`, `personIDlist` and `XMLtext`. A `MapRenderer` is provided to create the map images, a `Locator` is given for finding the user’s location, and a `Clipboard`, along with associated concepts such as data flavours, is available.

Basic elements of the language's APIs that were assumed are collection classes (**Lists**, **Sets** and **Maps**), various GUI widgets including list and tree widgets along with associated properties such as **Colors** and **Fonts**, **Serializable** classes, and some form of **Timestamp**.

6 GUI Callbacks

6.1 MainView

```
MainView.newMap ⇒ MasterController.createDefaultMap()
MainView.newRelativeMap ⇒ MasterController.createRelativeMap()
MainView.maps ⇒ MasterController.showTool()
MainView.contacts ⇒ MasterController.showTool()
MainView.zoneAlert ⇒ MasterController.showTool()
```

6.2 MapTool

```
MapView.zoomIn ⇒ MapController.zoomIn()
MapView.zoomOut ⇒ MapController.zoomOut()
MapView.editInterest ⇒ MapController.editInterest()
MapView.panControl ⇒ MapController.panControl()
MapView.submapControl ⇒ MapController.submapControl()
MapView.createControl ⇒ MapController.createControl()

InterestEditorView.addPeople ⇒ InterestEditorView.addPeopleClicked()
    ⇒ InterestEditorController.addPeopleClicked()
InterestEditorView.addBulletins ⇒ InterestEditorView.addBulletins-
    Clicked() ⇒ InterestEditorController.addBulletinsClicked()
InterestEditorView.removePeople ⇒ InterestEditorView.removePeople-
    Clicked() ⇒ InterestEditorController.removePeopleClicked()
InterestEditorView.removeBulletins ⇒ InterestEditorView.removeBulle-
    tinsClicked() ⇒ InterestEditorController.removeBulletinsClicked()
```

6.3 ContactsTool

```
ContactsView.search ⇒ ContactsList.restrict()
ContactsView.undo ⇒ ContactsController.undoNext()

ContactsList.contacts ⇒ ContactsController.selectionChanged()
ContactsList.newPerson ⇒ ContactsController.newPersonClicked()
ContactsList.newGroup ⇒ ContactsController.newGroupClicked()
ContactsList.delete ⇒ ContactsController.deleteClicked()

ContactsEditor.label ⇒ ContactsEditor.labelEdited()
ContactsEditor.id ⇒ ContactsEditor.idEdited()
ContactsEditor.setGraphic ⇒ ContactsEditor.setGraphicClicked()
ContactsEditor.privacy ⇒ ContactsEditor.privacyChanged()

GroupContentEditor.content ⇒ GroupContentEditor.addContact()
GroupContentEditor.remove ⇒ GroupContentEditor.removeClicked()
```

```

PrivacyRequestDialog.visible ⇒ PrivacyRequestDialog.dismiss()
    ⇒ ContactsController.privacyChosen()
PrivacyRequestDialog.undefined ⇒ PrivacyRequestDialog.dismiss()
    ⇒ ContactsController.privacyChosen()
PrivacyRequestDialog.invisible ⇒ PrivacyRequestDialog.dismiss()
    ⇒ ContactsController.privacyChosen()

```

6.4 ZoneAlertTool

```

AlertList.alerts ⇒ ZoneAlertView.selectionChanged()
AlertList.add ⇒ ZoneAlertController.addAlert()
AlertList.remove ⇒ AlertList.removeClicked() ⇒ ZoneAlertControl-
    ler.removeAlert()

AlertEditor.selectArea ⇒ ZoneAlertController.selectArea()

AlertDisplay.okay ⇒ AlertDisplay.dismiss()

```

6.5 BulletinTool

```

BulletinEditor.changeGraphic ⇒ BulletinEditor.changeGraphic()
BulletinEditor.okay ⇒ BulletinEditor.okay() ⇒ BulletinControl-
    ler.viewDismissed()
BulletinEditor.cancel ⇒ BulletinEditor.cancel() ⇒ BulletinCont-
    roller.viewDismissed()

BulletinViewer.okay ⇒ BulletinViewer.dismiss() ⇒ BulletinControl-
    ler.viewDismissed()
BulletinViewer.remove ⇒ BulletinViewer.dismiss() ⇒ BulletinCon-
    troller.viewDismissed()

```

7 Example Control Flows

The following are examples of control flows demonstrating some of the most complicated parts of the design.

7.1 Create a Geobulletin

User selects the create control in the `MapView`.
`MapController` receives callback and sets the current control to `create`.
User clicks on the map in the `MapView`.
`MapController` receives callback with a `ClickInfo` object and fires a `MapClickEvent` with `creationClick` set to `true`.
`MasterController` receives `MapClickEvent`, finds it's a creation click and calls `BulletinController.editBulletin()`, passing it a new `Bulletin` with the user as its author and the `Position` from the `MapClickEvent`.
`BulletinController` brings up a new `BulletinEditor` for the bulletin.

User edits the bulletin and clicks the `okay` button.
`BulletinController` receives a callback and fires a `BulletinEditEvent` with the bulletin and position it was given previously.
`MasterController` receives `BulletinEditEvent` and sends bulletin and position to `MapManager.publishGeobulletin()`.
`MapManager` sends the geobulletin off via the `ServerProxy`.

7.2 Select a ZoneAlert Area

User clicks the `selectArea` button in the `AlertEditor`.
`ZoneAlertController` receives a callback. It creates an `ElementRequest` for a `MappedArea`, calling back to `ZoneAlertView.areaPicked()`. It passes this to the picker.
`MasterController` calls `MapController.pickElement()` with type `MappedArea`, which prompts the user to pick one.
User clicks the map in the `MapView`.
`MapController` receives a callback with a `ClickInfo` object and creates a `MapClickEvent` out of it.
`MasterController` receives the `MapClickEvent`, notes it fits with an outstanding request, and calls the `ElementRequest`'s `replyTo()` method.
`ZoneAlertView` receives a callback with the `MappedArea`, and forwards it to `AlertEditor.areaPicked()`.

8 Design Decisions

8.1 Tool Independent Model

There is no direct correspondence between the way in which the `Model` is split up and the way in which the *tools* are split up. This is due to the fact that composing the `Model` from the individual `ToolModels` would restrict modification of the user's toolset and make sharing of model information between the *tools* more difficult. The `Model` is instead designed to provide the core of the system regardless of what *tools* the user may use to manipulate it. A façade is then used for each *tool* to pare down the `Model` to what is relevant to that *tool*. This extra level of indirection could later be turned into an adapter for a heavily modified model without affecting the remainder of the *tool*'s code.

8.2 Tool Interactions

The MVC architecture provides a basis for *tools* to function independently. By being distinct views and controllers of the `ClientModel`, one *tool* can

adjust the `ClientModel`, such as change the contacts list, and this change will be reflected in other *tools* without them requiring knowledge about the first *tool*.

To provide a convenient interface to the user and avoid duplication, there must be some means for communication between *tools*. For instance, upon clicking a geobulletin in the `MapTool` it is brought up in the `BulletinTool`. This is handled by the `MasterController`, completely removing the need for *tools* to require the existence of one another. This is achieved by having *tools* inform the `MasterController` when actions occur, via an event-listener system. The `MasterController`, being the part of the program that creates the Tools in use, is aware of what *tools* need to know about the event under the current toolset and will subsequently call appropriate methods in those *tools*.

The combination of the `ClientModel` and `MasterController` results in existing Tools begin able to continue functioning without modification even if Tools are added, removed or replaced. Furthermore, the knowledge of what Tools exist and any interactions between them is collected and isolated in the `MasterController`, meaning only a very specific part of the system needs to be modified upon changing what Tools exist or the functionality they provide.

8.3 Common Data Structures

The same data structures are used by many parts of the ActiveNU Client to avoid duplication. It should be noted, however, that even if many parts of the application use the same data structures these parts are not necessarily coupled with one another. Using common data structures rather than independent ones simply reduces duplication.

8.4 PositionalObjects

A `PositionalObject` allows any object to be placed at a location on the `Map`. It does this by containing positional data, along with a reference to the object that is placed there. This completely separates the information stored at a location from the location itself, making it easier to reuse those objects or later incorporate objects from another program into the map.

8.5 Alert Areas

The areas used by alerts will be any `MappedAreas`. This means that in the future it would be simple to add the ability for users to define their own areas without having to modify the `ZoneAlertTool`. An area creation *tool* could simply be made that creates new `MappedAreas` on the `Map`.

A Alphabetical Class Listing

Class	Section	Module
AbsolutePosition	5.5.6	Data.Position
AddContact	5.2.3	Tools.ContactsTool.Commands
AlertDisplay	5.2.4	Tools.ZoneAlertTool
AlertEditor	5.2.4	Tools.ZoneAlertTool
AlertEvent	5.4.1	Event.Model
AlertList	5.2.4	Tools.ZoneAlertTool
AlertListener	5.4.1	Event.Model
AlertManager	5.3	Model
AlertProcessor	5.5.1	Data.Alert
AlertTrigger	5.5.1	Data.Alert
Angle	5.5.3	Data.Map
Bulletin	5.5	Data
BulletinController	5.2.5	Tools.BulletinTool
BulletinEditEvent	5.4.2	Event.Tools
BulletinEditListener	5.4.2	Event.Tools
BulletinEditor	5.2.5	Tools.BulletinTool
BulletinModel	5.2.5	Tools.BulletinTool
BulletinViewer	5.2.5	Tools.BulletinTool
ClickInfo	5.2.1	Tools.MapTool
ClientModel	5.3	Model
Command	5.2.3	Tools.ContactsTool.Commands
ContactsController	5.2.2	Tools.ContactsTool
ContactsEditor	5.2.2	Tools.ContactsTool
ContactsList	5.2.2	Tools.ContactsTool
ContactsManager	5.3	Model
ContactsModel	5.2.2	Tools.ContactsTool
ContactsView	5.2.2	Tools.ContactsTool
Copy	5.2.3	Tools.ContactsTool.Commands
Cut	5.2.3	Tools.ContactsTool.Commands
DeleteContact	5.2.3	Tools.ContactsTool.Commands
DetailedPerson	5.5.2	Data.Contacts
DragInfo	5.2.1	Tools.MapTool
EditText	5.2.3	Tools.ContactsTool.Commands
ElementPicker	5.2	Tools
ElementRequest	5.2	Tools
Event	5.4.1	Event.Model
GeobulletinMessage	5.5.4	Data.Messages
GeographicMap	5.5.3	Data.Map
GPScoords	5.6	Provided
GraphicSelector	5.2	Tools
Group	5.5.2	Data.Contacts
GroupContentEditor	5.2.2	Tools.ContactsTool
IconLabel	5.5.3	Data.Map
IncomingMessage	5.5.4	Data.Messages
InterestEditorController	5.2.1	Tools.MapTool
InterestEditorView	5.2.1	Tools.MapTool

Class	Section	Module
InterestTarget	5.5.2	Data.Contacts
Invoker	5.2.2	Tools.ContactsTool
Label	5.5.3	Data.Map
Location	5.5.6	Data.Position
LocationReport	5.5.4	Data.Messages
LocationReporter	5.3	Model
Locator	5.6	Provided
LocatorRelativePPos	5.5.5	Data.Persistent
MainView	5.1	ActiveNUClient
Map	5.3	Model
MapClickEvent	5.4.2	Event.Tools
MapClickListener	5.4.2	Event.Tools
MapController	5.2.1	Tools.MapTool
MapElement	5.5.3	Data.Map
MapEvent	5.4.1	Event.Model
MapLayer	5.5.3	Data.Map
MapListener	5.4.1	Event.Model
MapManager	5.3	Model
MapMessage	5.5.4	Data.Messages
MapModel	5.2.1	Tools.MapTool
MappedArea	5.5.3	Data.Map
MappedLine	5.5.3	Data.Map
MappedShape	5.5.3	Data.Map
MapRenderer	5.6	Provided
MapView	5.2.1	Tools.MapTool
MasterController	5.1	ActiveNUClient
MessageHandler	5.3	Model
messageHandler	5.6	Provided
Modify	5.2.3	Tools.ContactsTool.Commands
ModifyGroup	5.2.3	Tools.ContactsTool.Commands
NewContact	5.2.3	Tools.ContactsTool.Commands
NewGroup	5.2.3	Tools.ContactsTool.Commands
NewMapEvent	5.4.2	Event.Tools
NewMapListener	5.4.2	Event.Tools
NewPerson	5.2.3	Tools.ContactsTool.Commands
OutgoingMessage	5.5.4	Data.Messages
Paste	5.2.3	Tools.ContactsTool.Commands
PersistentMap	5.5.5	Data.Persistent
PersistentMaps	5.5.5	Data.Persistent
PersistentModel	5.5.5	Data.Persistent
PersistentPosition	5.5.5	Data.Persistent
Person	5.5	Data
personIDlist	5.6	Provided
PersonRelativePPos	5.5.5	Data.Persistent
Position	5.5.6	Data.Position
PositionalObject	5.5.3	Data.Map
Positioned	5.5.6	Data.Position
PositionEvent	5.4.1	Event.Model

Class	Section	Module
PositionListener	5.4.1	Event.Model
PresenceAlert	5.5.1	Data.Alert
PrivacyEvent	5.4.1	Event.Model
PrivacyListener	5.4.1	Event.Model
PrivacyRequestDialog	5.2.2	Tools.ContactsTool
ProximityAlert	5.5.1	Data.Alert
PubSubProxy	5.6	Provided
RangeEvent	5.4.1	Event.Model
RangeListener	5.4.1	Event.Model
RelativePosition	5.5.6	Data.Position
RemoveContact	5.2.3	Tools.ContactsTool.Commands
Serializable	5.6	Provided
ServerProxy	5.3	Model
SetGraphic	5.2.3	Tools.ContactsTool.Commands
SetPrivacy	5.2.3	Tools.ContactsTool.Commands
StyledTextLabel	5.5.3	Data.Map
TargetGetter	5.3	Model
TextLabel	5.5.3	Data.Map
TimeEvent	5.4.1	Event.Model
TimeListener	5.4.1	Event.Model
Timer	5.3	Model
ToolController	5.2	Tools
ToolModel	5.2	Tools
ToolView	5.2	Tools
Visibility	5.5	Data
VisibilityRequest	5.5.4	Data.Messages
VisibilityResponse	5.5.4	Data.Messages
XMLtext	5.6	Provided
ZoneAlertController	5.2.4	Tools.ZoneAlertTool
ZoneAlertModel	5.2.4	Tools.ZoneAlertTool
ZoneAlertView	5.2.4	Tools.ZoneAlertTool