SmarterCities

Federico Bellini & Francesco Capponi

26 febbraio 2015

Indice

Capitolo 1

Analisi

1.1 Introduzione

L'idea

L'idea di questo progetto è venuta osservando il sistema stradale di controllo e prevenzione che negli Stati esteri sta diventando sempre più presente e capillare. Ovviamente tutto questo ha i suoi lati positivi e negativi, a seconda del fine che il controllo ha. Il nostro software, denominato SmarterCities, si limiterà ad effettuare una dimostrazione pratica di alcune azioni che potrebbero esser fatte se potessimo essere in possesso di suddetti dati, senza esprimere giudizi etici o morali al riguardo.

Perché è possibile

E' da tenere in considerazione che in Italia già esiste una fitta rete di controllo stradale automatica, composta da autovelox, tutor etc. Questi enti distribuiti su tutto il territorio generano enormi quantità di dati quotidianamente, che vengono già utilizzati per scopi legislativi e regolamentari, ma sono gestiti come singole entità e quindi quasi totalmente privi di elaborazione.
Di conseguenza è già idealmente possibile offrire alla comunità il servizio che la nostra applicazione propone.

Figura 1.1: Mappa degli Autovelox in Italia

Il risultato desiderato

Con questa applicazione si potrebbero offrire ad esempio servizi di monitoraggio del traffico, analizzando le strade con maggior affluenza di una città; calcolare la velocità media realmente rispettata su ogni strada e prendere provvedimenti di conseguenza(aggiunta dossi ecc..). In aggiunta con un piccolo upgrade si potrebbe offrire un servizio di vigilanza, controllando le strade ed

avvertendo le autorità quando viene avvistato un mezzo rubato.
Lo scopo dell'applicazione è quindi quello di mostrare come si possa creare un server che gestisca le informazioni raccolte per le strade dai vari rilevatori con fini utili alla comunità.

1.2 Funzionalità

Il software è composto da tre funzionalità principali ed un esempio di estensione: dovrà permettere di raccogliere i dati dalle varie fonti, salvare i suddetti dati, simulare una situazione pseudo reale di traffico cittadino e fornire un esempio di elaborazione dei dati ottenuti.

1.2.1 Ricezione dati

La funzionalità core per una Smart City è proprio quella di porter ricevere i dati da più fonti. Per questo è stata implementata una specifica interfaccia che consente ai Tutor, Autovelox e simili (che da ora in poi per generalizzare chiameremo "Street Observers" o "Osservatori"), di inviare i dati acquisiti (denominati "Sightings", o "Avvistamenti").

1.2.2 Salvataggio dati

Essendo un software finalizzato all'analisi dei dati, è fornita un'interfaccia che consente il salvataggio delle informazioni ricevute su una memoria stabile.

1.2.3 Simulazione

Non avendo a disposizione degli Osservatori reali che ci inviano dati, ai fini di una dimostrazione pratica, abbiamo implementato un'interfaccia che genera dati pseudo-casuali.

1.2.4 Elaborazione

E' possibile creare infiniti esempi di elaborazione dei dati. Nel nostro specifico caso abbiamo scelto di creare tre differenti interfacce per l'elaborazione delle informazioni:

- Una prima interfaccia che consente di ottenere informazioni generali, quali la frequenza giornaliera/mensile ecc. di auto avvistate da un Osservatore, la velocità media rilevata ed ulteriori parametri ricavati dai dati spediti dagli Osservatori.
- Una seconda interfaccia che mostra su di una mappa la posizione effettiva degli Osservatori;
- Una terza interfaccia che verifica ad ogni Avvistamento, che la macchina passata non sia stata rubata. Se così fosse dovrà essere segnalata questa evenienza all'operatore.

Capitolo 2

Design

2.1 Architettura complessiva

Si nota da subito guardando la figura 2.1 dello schema generale che segue, che l'applicazione ha più di un punto di entrata: Utente e Network. In questa situazione pare ovvio usare il pattern *Model-View-Controller*. Così facendo infatti, sia la *GUI* che il *Network* sono trattati come due differenti *View*. Infatti la *GUI* sarà un'interfaccia View per gli "operatori umani", mentre il *Network* sarà un'interfaccia per gli operatori meccanici, come gli *Street Observers*. Con questo genere di implementazione possiamo quindi avere conferma della corretta implementazione del Controller, dimostrando che esso può gestire più di una singola View.
Di conseguenza il Controller sarà punto cardine dell'applicazione, processando gli eventi provenienti dalle Views, ed interfacciandosi con il Model, ossia il Database contenente tutti i dati ricevuti. In particolare visto che il *Network*, ricevendo messaggi da parte degli Street Observers, genera a sua volta dei messaggi che andranno ad influire sulla *GUI*, il *Controller* si dovrà preoccupare anche di sincronizzare le due parti.

2.2 Schema Generale

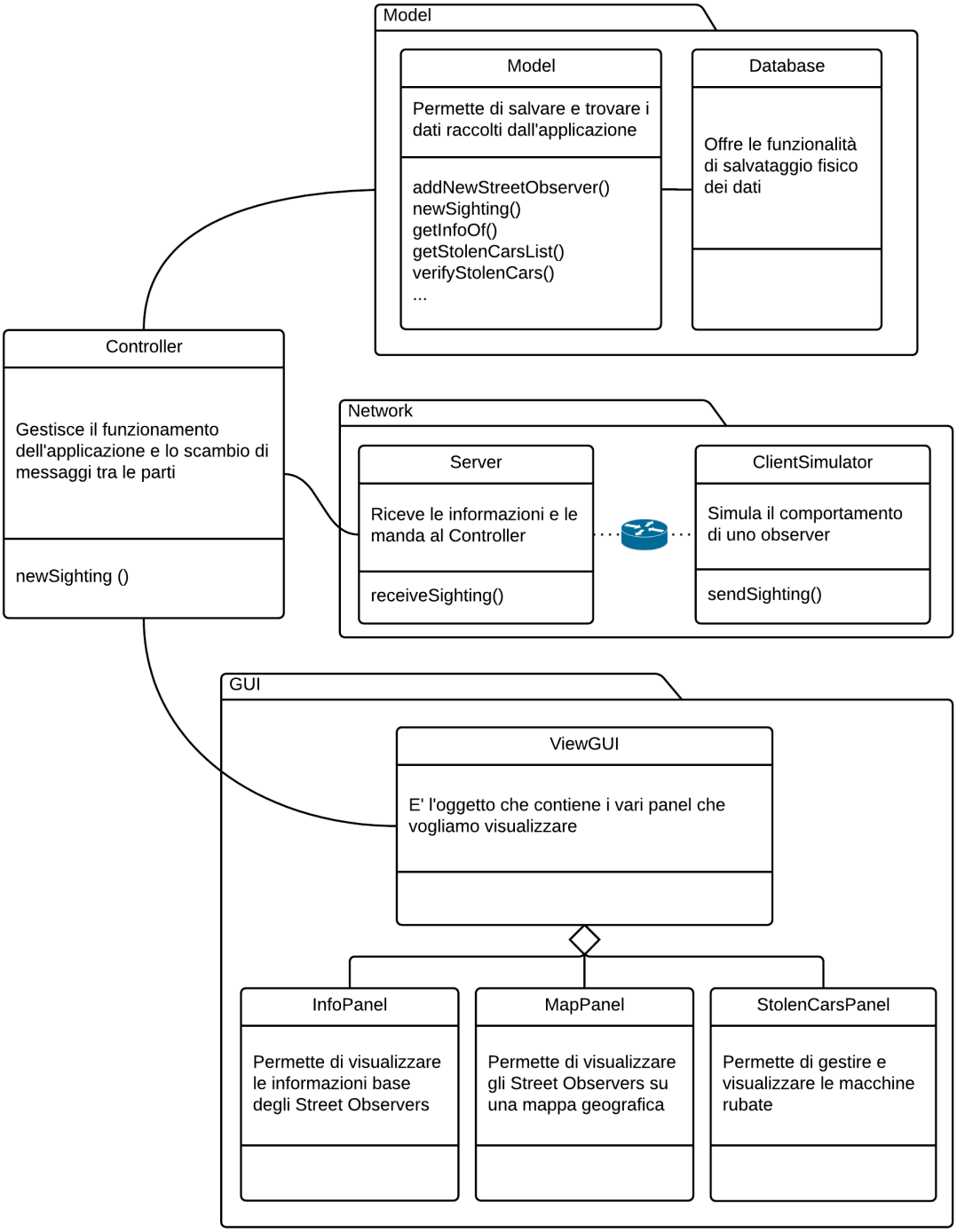

Figura 2.1: Descrizione dell'architettura per macroblocchi

2.3 Pattern Utilizzati

2.3.1 Controller

Decorator

Di default il Controller mette a disposizione solo l'interfaccia per la gestione degli Street Observers e dei Sighting.
Per mostrare le possibilità di future estensioni dell'applicazione, abbiamo voluto aggiungere delle funzioni all'*IController* utilizzando il pattern *Decorator*, implementato dalla classe *AbstractControllerDecorator*. In questo modo tramite l'implementazione dello *StolenCarsController* che decora un *IController*, in un solo Controller siamo in grado di gestire anche le funzionalità riguardanti le macchine rubate.

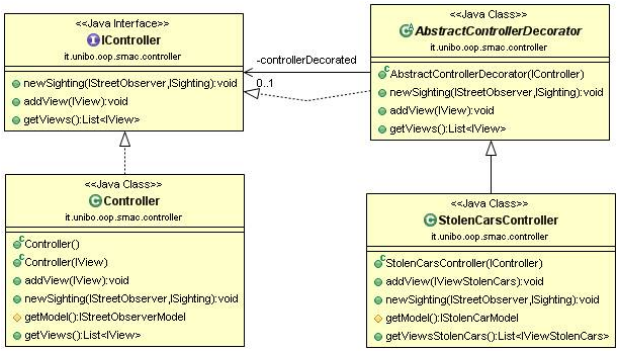

Figura 2.2: UML descrittivo del decorator del Controller

Observer

Il Controller, è strettamente legato alle Views, poiché deve gestirne gli eventi. Per garantire un maggior indipendenza tra le due parti, alla View non è permesso di richiamare direttamente i metodi del Controller, ma ogni View scatenerà degli eventi su un'ulteriore classe che ne fa da Observer, la quale poi si occuperà di richiamare il giusto metodo del Controller per ogni differente evento. In questo modo se in futuro avverrà un cambiamento della ai metodi del controller, non causerà la modifica della View, che farà riferimento univocamente alla classe Observer (*InfoStreetObserverObserver*. Le classi a cui vengono *attached* questi Observer sono: *IView* (con la funzione attachStreetObserverObserver) e *IViewStolenCars* (con la funzione attachStolenCarsObserver). Nella figura 2.3 abbiamo mostrato soltanto il caso del *Controller* e relativa *IView*. Il caso dello *StolenCarsController*, è del tutto analogo.

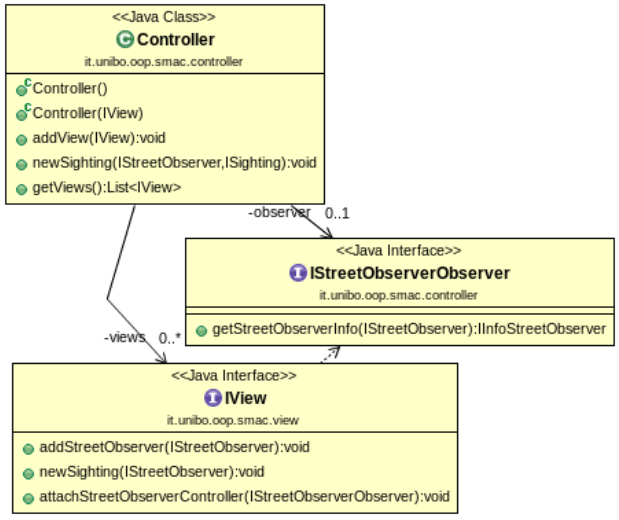

Figura 2.3: UML descrittivo degli obsever del Controller

2.3.2 GUI

ViewGUI: Façade

La GUI è composta da una una serie di panel per la gestione della visualizzazione delle varie componenti dell'applicazione. Per rendere più semplice e leggibile il codice, tutti i panel vengono gestiti da un'unica classe, implementata secondo il pattern Façade, che quindi gioca un ruolo di 'facciata' per tutte le altre, permettendo in questo modo un accesso più facile a tutte le sue sottoclassi.

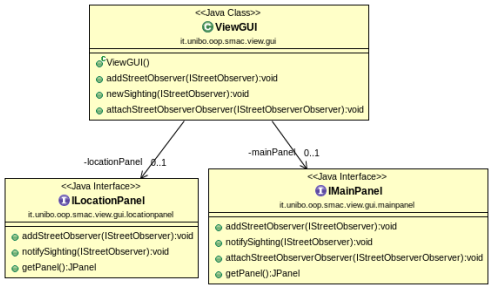

Figura 2.4: UML descrittivo del facade nella GUI

AbstractMapMarker: Template Method

La classe *AbstractMapMarker* implementa un generico *MapMarker* (ossia un'icona che deve essere visualizzata su di una mappa per marcarne una posizione), lasciando il compito alle classi che la estendono di scegliere la specifica icona che deve essere visualizzata. Come si può vedere nella figura 2.5, in questa applicazione ce ne sono di due tipi: *RedMarker* e *GreenMapMarker*, che utilizzano come marker rispettivamente un pin di colore rosso e verde.

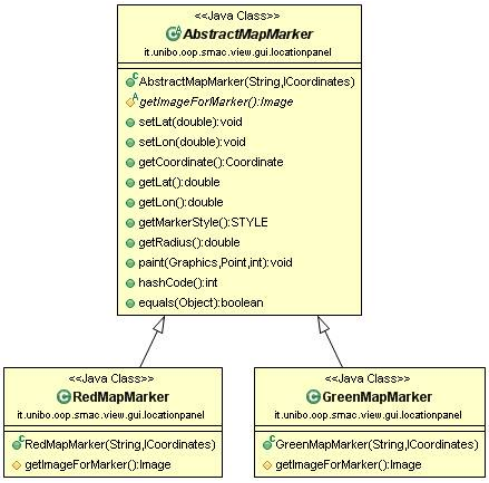

Figura 2.5: UML descrittivo del Template Method dell'AbstractMapMarker

2.3.3 Model

Singleton

Il *Singleton* è stato utilizzato per cercare di risolvere uno specifico problema: forzare l'applicazione a non avere più di una classe che gestisca le connessioni. Infatti le classi *Connection*, *StreetObserverModelDatabase* e *StolenCarModelDatabase* devono istanziare tutte al più una classe. Questo è fatto per ovviare principalmente due problemi:

- per mantenere al più una connessione attiva;
- per permettere alle query di essere eseguite con transizaoni atomiche. Infatti se avessimo più istanze, visto che il DataBase considerato non supporta le transazioni in maniera nativa, si rischierebbe di entrare in una sezione critica senza mutua esclusione, causando *race condition*.

Builders

L'applicazione deve gestire in differenti parti, alcuni tipi di classi con una grande abbondanza di campi da inizializzare. L'eterogeneità delle possibilità di inizializzazione e il numero dei campi, ci hanno portato ad usare il pattern *Builder* per facilitare l'inizializzazione e la leggibilità di esse.

In particolare è stato utilizzato questo pattern nel package *datatypes* per le classi: *InfoStreetObserver*, *Sighting* e *StolenCars*.

2.3.4 Network

Observer

Anche la Network è stata progettata per essere estesa. Per questo alla ricezione dei messaggi da parte del *NetServer*, questi ultimi verranno passati al *Dispatcher* (che implementa la classe *Observable*), il quale provvederà a richiamare tutti i Job (Observer) che si sono messi in ascolto. In questo modo basta aggiungere un nuovo Job che gestirà automaticamente i nuovi messaggi che si prevede vengano ricevuti dalla rete.
Esempio: il job *ControllerSightingSender* gestirà l'arrivo dei messaggi *PlainSighting*. Una volta aggiunto, si occuperà di tradurre il messaggio dal formato di rete (come *PlainSighting*), al formato compatibile con l'applicazione (*Sighting*), e informare il controller generando un evento nell'osservatore della view (*IStolenCarObserver*).

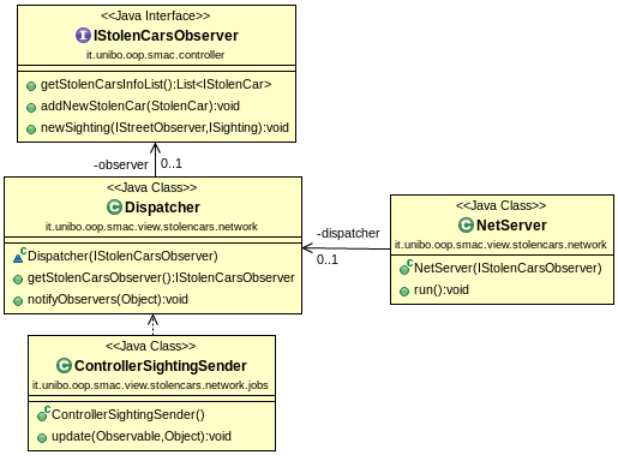

Figura 2.6: UML descrittivo degli obsever nel Network

Capitolo 3

Sviluppo

3.1 Testing

I punti critici dell'applicazione di cui si deve verificare il funzionamento, sono 3, e per ognuno è stato adottato uno specifico metodo di test:

- il database che si occupa della memorizzazione dei dati;
- il network che deve interagire con il server;
- la GUI che deve interagire con l'utente.

3.1.1 Test Automatico, Model

Per ogni Model, abbiamo creato una serie di test che andassero a verificare ogni operazione eseguibile sulla base di dati.
JUnit si è dimostrando estremamente utile anche in fase di sviluppo, permettendoci di controllare che le modifiche effettuate non andassero ad impattare sul risultato aspettato. Inoltre sono stati controllati non solo i casi in cui le funzioni avrebbero dovuto gestire i casi di successo (passando sempre i parametri corretti) ma anche i casi in cui avrebbero dovuto restituire un'eccezione, in modo da avere una copertura a tutto tondo dei casi d'utilizzo.

3.1.2 Test SemiAutomatico, Network

Volendo dare prova pratica del funzionamento dell'interaccia del Network abbiamo dovuto provvedere alla creazione di un generatore di messaggi, che simula il comportamento di uno Street Observer.
La verifica della simulazione verrà effettuata manualmente dall'utente, controllando che all'apertura dell'applicazione vengano segnalati i *Sightings* sulla mappa, e nel pannello delle informazioni, i passaggi delle macchine. Se la GUI non dovesse visualizzare alcuna informazione, ovviamente questo Test dovrebbe essere ritenuto fallito.

3.1.3 Test Manuale, GUI

Il test delle funzionalità della GUI, in cui si sarebbe dovuto simulare il comportamento dell'utente, è stato di tipo manuale.

3.2 Librerie

Per migliorare le condizioni di sviluppo, e per offrire funzionalità grafiche avanzate, ci siamo appoggiati su 5 librerie:

- *slf4j* (Simple Loggin Facade for Java), per ottenere la più assoluta libertà nel Logging. Infatti slf4j permette di scrivere i log su molteplici dispositivi, dalla semplice console, a files.

- *ormlite*, il quale offre un'interfaccia ORM (Object-Relational Mapping) per accedere e salvare semplicemente i dati. In particolare in questa versione del software, abbiamo deciso che lo store dei dati sarà effettuato in memoria RAM. Questa decisione è stata presa in particolare per non gravare sul setup dell'applicazione (se avessimo deciso che si doveva collegare a un database MySQL o Oracle, ovviamente era necessaria anche la relativa configurazione).

- *JMapViewer*, che permette di implementare facilmente una schermata con mappa scorrevole e autocaricante in stile Google Maps, ma con la ampia liberà di utilizzo che il servizio messo a dispozione da OpenStreetMap. Inizialmente infatti avevamo pensato di usare proprio le mappe di Google come dichiarato nella fase iniziale del progetto, ma successivamente abbiamo trovato una serie di problemi: la licenza pone forti condizioni sull'utilizzo; la libreria non è implementata in Java nativo, ma attraverso un wrapper di una WebView, peggiorando così le prestazioni e le possibilità di utilizzo, ed è in oltre necessario inserire all'interno dell'applicazione la propria chiave personale di sviluppo Google per usufruire dei servizi, generando una potenziale divulgazione che non è auspicabile.

- *Netty*, una libreria per gestire il networking dalle enormi possibilità, la quale sta venendo sempre più utilizzata anche da colossi dell'informatica come Twitter, Apple ed altri (per una lista completa (http://goo.gl/jRb69k).
  Il suo successo probabilmente è derivato dall'alta facilità d'utilizzo, la grande estendibilità e performance. Infatti grazie a questa libreria, che implementa un sistema ad-hoc di gestione delle connessioni basandosi sulla strategia NIO (Non-Blocking IO), ci permette di astrarci da problematiche di performance relative alla rete, concentrandoci solo sul message passing.

- *gson*, è stata utilizzata per configurare il simulatore degli street observer. Infatti, tramite file JSON all'interno del progetto, è possibile specificare, in quali punti passeranno le macchine, senza dover andare a specificare queste informazioni programmaticamente.

3.3 Divisione dei compiti e metodologia di lavoro

I primi giorni di sviluppo sono stati investiti per cercare di creare un'architettura UML più coerente possibile al progetto finale. Data la dimensione del progetto successivamente sono state fatte alcune modifiche, ma lasciando completamente invariata la relazione che c'era tra i macroblocchi.
Per quanto riguarda la divisione compiti in linea generale è stata totalmente rispettata, tranne per la creazione dell'interfaccia della mappa che era stata affidata ad entrambi, ma alla fine trattandosi di un miniblocco, Bellini ha completato senza l'apporto di Capponi. La parte di database inoltre è stata creata in stretta collaborazione tra i due sviluppatori, in quanto comune ad entrambi, anche se legata ad aspetti differenti(salvataggio dati degli Osservatori/auto rubate). Le parti per la correzione di bugs e warnings generati dai plug-in checkStyle, PMD, FindBugs e qualche refactoring sono stati svolti da entrambi gli sviluppatori.
La divisione dei compiti era:

- Bellini: parte server: elaborazione dati ricevuti (controller), GUI del server (parte degli Osservatori), salvataggio dei dati ottenuti dai client su un DB (Model), interfaccia che utlizza APIs Google Maps(poi APIs del progetto OpenStreetMap).
- Capponi: parte client: invio di dati pseudo-casuali per simulazione, creazione di una connessione tra server e clients, gestione del database di auto rubate (ModelStolenCars), GUI del server (parte delle auto rubate), interfaccia che utlizza API Google Maps (solo Bellini), eventuale elaborazione del flusso di veicoli dentro e fuori dalla città (non fatta poiché fuori monte ore).

Come autocritica possiamo dire che la lista di compiti originariamente stilata, non era molto dettagliata e precisa, e ciò che non era stato scritto (come la creazione del Network, o la GUI per la gestione delle Stolen Cars, sviluppate entrambe da Capponi) può essere inteso come la diretta conseguenza della creazione delle altre parti del software.

3.4 Criticità

In una prima fase, avevamo progettato di implementare la mappa degli Street Observers tramite le librerie di Google Maps, pensando che fossero perlomeno simili a quelle della versione Android. Per nostra sfortuna, erano completamente diverse, e il supporto ad esse era minimo. Per questo motivo, abbiamo deciso di passare alla seconda scelta basata sul progetto Open-Source OpenStreetMap, che invece offriva un ottima libreria, che ha velocizzato lo sviluppo enormemente.
Un'altra question riguarda la *ViewGUI*: l'idea iniziale era quella di renderla una classe decorabile, in modo da poter aggiungere pannelli man mano, come ad esempio *ViewGUIStolenCars*. La problematica è venuta fuori quando ci siamo resi conto che non poteva essere un vero e proprio decorator, poiché ognuno di questi decorator avrebbero aggiunto dei metodi, che sarebbero dovuti essere richiamati dopo l'inizializzazione nel controller (come *attachStolenCarsController* all'interno di *ViewGUIStolenCars*). Per questo abbiamo preferito non "inventarci" un pattern ad hoc, ma lasciare la classe come estensione della precedente.

Capitolo 4

Commenti finali

Il progetto nel complesso è stato sviluppato secondo le idee iniziali che ci eravamo prosti, incontrando soltanto piccole difficoltà con l'utilizzo delle APIs messe a disposizione da Google e con qualche difficoltà sulle librerie di database. Osservando il progetto nella sua interezza si può notare che il suo punto di forza risiede nella facilità di annettere future estensioni del software, senza apportare radicali cambiamenti al codice già prodotto.

4.1 Sviluppi futuri

Questo software non è stato sviluppato con secondi fini o scopi, ma puramente come attività didattica per il progetto del corso di Programmazione ad Oggetti a.a. 2014/2015 della facoltà di Ingegneria e Scienze Informatiche di Cesena. Per questo motivo non prevediamo futuri sviluppi.

Capitolo 5

Guida Utente

Aperta la schermata si dovrà aspettare qualche secondo che il server si avvii e inizi a generare i Sightings.
In questa fase potrebbe accadere che l'applicazione generi un errore e si chiuda. Questo errore è dato dalla mancata inizializzazione del server network perché ha trovato la porta *8007* occupata.
In seguito si visualizzeranno 3 schede:

- una in cui si possono consultare le statistiche degli Street Observer.
- una in cui possiamo vedere la reale posizione nella mappa.
- una in cui possiamo gestire le macchine rubate.

N.B. Nella mappa è importante notare che il tasto usato per trascianre la mappa non è il sinistro, come siamo abituati tradizionalmente, ma il destro. Lo scroll del mouse funziona invece come di consueto come zoom per la mappa.

L'ultimo pannello è quello più interattivo: si gestisce il database delle macchine rubate. In alto a sinistra è presente una casella di input in cui inserire una targa di cui si vuole fare il track. Per trovarne una valida, basta visualizzare l'ultima targa avvistata da un'osservatore, ed inserirla nel database di auto rubate. Da lì a poco si vedrà il database delle segnalazioni aumentare di dimensione.