

**UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II**

*Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica*



Elaborato in Progettazione e Sviluppo di Sistemi Software

Smart Parking Ticket



Anno Accademico 2018/2019

A cura di:

Formisano Raffaele M63000912

Napolano Giuseppe M63000856

Pugliese Nicola Alessandro M63000847

Romito Giuseppe M63000936

Indice

Introduzione	1
1 Processo di sviluppo e requisiti	2
1.1 Requisiti funzionali	2
1.2 Processo di sviluppo	2
2 Analisi	5
2.1 Analisi Testuale	5
2.2 Diagramma dei casi d'uso	6
2.2.1 Descrizione dei casi d'uso	7
2.3 Diagramma di contesto	10
2.4 Diagramma delle classi concettuale	11
2.5 Diagramma delle classi raffinato	12
2.6 Architettura software	12
2.7 Diagramma dei componenti	13
2.8 Casi d'uso implementati	13
2.8.1 Diagramma ad alto livello	15
2.9 Caso d'uso AcquistaTicket	15
3 Progettazione	18
3.1 Diagramma delle classi di Progetto	18
3.1.1 Proxy-Skeleton	20
3.2 Deployment diagram	20
3.3 Diagramma di sequenza AcquistaTicket	21
3.4 Diagramma di sequenza RinnovaTicket	23
3.5 Vista Client Android	26
3.5.1 Diagramma delle classi lato client	26
3.5.2 Diagramma di sequenza AcquistaTicket lato client android	28
Conclusioni e sviluppi futuri	29

Elenco delle figure

1	Parchimetro	1
1.1	Screen utilizzo di Atom(GIT)	3
1.2	Screen utilizzo di Trello	4
2.1	Visual Paradigm: Analisi Testuale	5
2.2	Visual Paradigm: Diagramma dei casi d'uso	6
2.3	Visual Paradigm: Descrizione del caso d'uso Controllo Disponibilità	7
2.4	Visual Paradigm: Descrizione del caso d'uso Aggiungi Auto	8
2.5	Visual Paradigm: Descrizione del caso d'uso carica conto	8
2.6	Visual Paradigm: Descrizione del caso d'uso AcquistaTicket	9
2.7	Visual Paradigm: Descrizione del caso d'uso RinnovaTicket	10
2.8	Diagramma di contesto	10
2.9	Visual Paradigm: Diagramma delle classi concettuale	11
2.10	Visual Paradigm: Diagramma delle classi raffinato	12
2.11	Visual Paradigm: Diagramma delle componenti	13
2.12	Visual Paradigm: Diagramma di sequenza di Aggiungi Auto	14
2.13	Visual Paradigm: Diagramma di attività di AcquistaTicket	15
2.14	Visual Paradigm: Diagramma di attività di RinnovaTicket	15
2.15	Visual Paradigm: Diagramma di sequenza del caso d'uso AcquistaTicket	16
2.16	Visual Paradigm: Diagramma di sequenza del caso d'uso Rinnova Ticket	17
3.1	Visual Paradigm: Diagramma delle classi di progetto	19
3.2	Visual Paradigm: Proxy-Skeleton	20
3.3	Visual Paradigm: Deployment diagram	21
3.4	Visual Paradigm: AcquistaTicket di dettaglio	22
3.5	Visual Paradigm: RinnovaTicket di dettaglio	24
3.6	Visual Paradigm: Arresta Sosta	25
3.7	Visual Paradigm: Vista lato client	27
3.8	Visual Paradigm: Acquista lato-client	28

Introduzione

Smart Parking Ticket è un'applicazione rivolta ai dispositivi mobili con sistema operativo Android, sviluppata nell'ambito del corso di Progettazione e sviluppo dei sistemi software. L'idea nasce dalla necessità di rendere meno snervante l'acquisto di un ticket per il parcheggio di un'auto. Ci sono molte situazioni dove acquistare un ticket si rivela complicato, molte volte ci si dimentica delle monete, altre volte i parchimetri sono guasti, troppo lontani dall'area di sosta o addirittura assenti. Un'altra situazione sgradevole è quando si è in possesso di un ticket prossimo alla scadenza e si è lontani dalla propria auto o impegnati, risulta dunque un problema acquistarne un altro e potremmo essere multati. L'applicazione sviluppata risolve queste problematiche semplificando la vita degli automobilisti e dei vigili addetti al controllo della sosta.



FIGURA 1: Parchimetro

Capitolo 1

Processo di sviluppo e requisiti

1.1 Requisiti funzionali

Il **sistema di smart Parking Ticket** è un'applicazione Android che permette l'acquisto ed il controllo di ticket necessari alla sosta di un'auto. Essa dovrà permettere all'utente di registrarsi e loggarsi al sistema. Un funzionario comunale approva l'account di un utente alla piattaforma. Il sistema permette all'automobilista (una volta effettuato il login) di controllare la disponibilità di posti auto in una determinata area parcheggio e successivamente di acquistare un ticket inserendo il codice dell'area parcheggio in cui si trova, la targa dell'auto e la durata della sosta che influirà sul costo del ticket. Successivamente il sistema permetterà all'automobilista il pagamento del ticket. All'automobilista è garantito un rimborso qualora arresti la sua sosta prima della scadenza. Una volta effettuato il pagamento le informazioni della sosta vengono memorizzate in maniera persistente. Un vigile accede al sistema tramite username e password. Il vigile può visualizzare lo stato di sosta di un veicolo inserendo il numero di targa. Quando il tempo di sosta è prossimo alla scadenza, verrà inviata una notifica all'automobilista che può decidere di rinnovare o meno la copertura. Il sistema permette al vigile di emettere multe ed all'automobilista di pagarle. Inoltre il funzionario può eliminare dal sistema una multa nel caso in cui l'automobilista abbia provveduto al pagamento con un metodo esterno al sistema. Tutti i pagamenti sono effettuati tramite un credito residuo ricaricabile associato all'account dell'automobilista.

1.2 Processo di sviluppo

Dato l'interesse e la curiosità dei membri del team di sviluppo per gli approcci Agile si è scelto di cimentarsi in uno sviluppo Agile di tale applicazione, in particolare si è

utilizzata la metodologia Scrum. Il team di sviluppo dopo un incontro con lo stakeholder in cui si è discusso dei requisiti funzionali ha ottenuto il nulla osta per iniziare la fase di analisi. Dopo le prime riunioni si è generato un Product Backlog contenente le funzionalità da sviluppare e la loro priorità. Da quest'ultimo si è scelto di implementare le due funzionalità prioritarie del nostro sistema cioè l'acquisto ed il rinnovo del ticket. Per la realizzazione di queste ultime però, sono necessarie piccole funzionalità di contorno come l'aggiunta di un'auto nel sistema e l'aggiornamento del conto dell'automobilista. Inizialmente sono stati discussi ed abbozzati in "Daily Scrums" i diagrammi di casi d'uso e delle classi, al contempo ci si è abituati a utilizzare strumenti di sviluppo come **GitHub** ed Atom(IDE di sviluppo creato da GitHub) per incentivare uno sviluppo software collaborativo e diminuire tempi di sincronizzazione tra i membri del team.

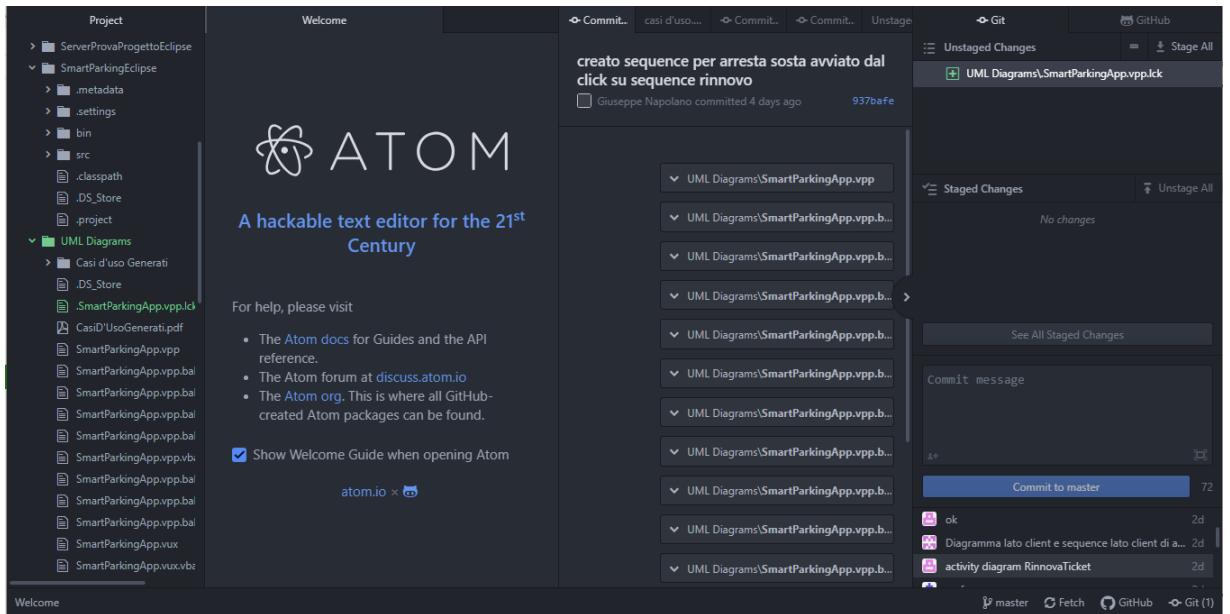


FIGURA 1.1: Screen utilizzo di Atom(GIT)

Quando per motivi di organizzazione non ci si è potuti incontrare si è cominciato ad utilizzare lo strumento di gestione di progetti **Trello**, in esso si è avuta una bacheca di progetto condivisa con la quale si teneva conto di cosa dovesse essere fatto e chi ne aveva il compito. Ulteriore strumento di comunicazione tra i membri del team sono state le videochiamate Skype per ciò che riguarda soprattutto la parte di implementazione.

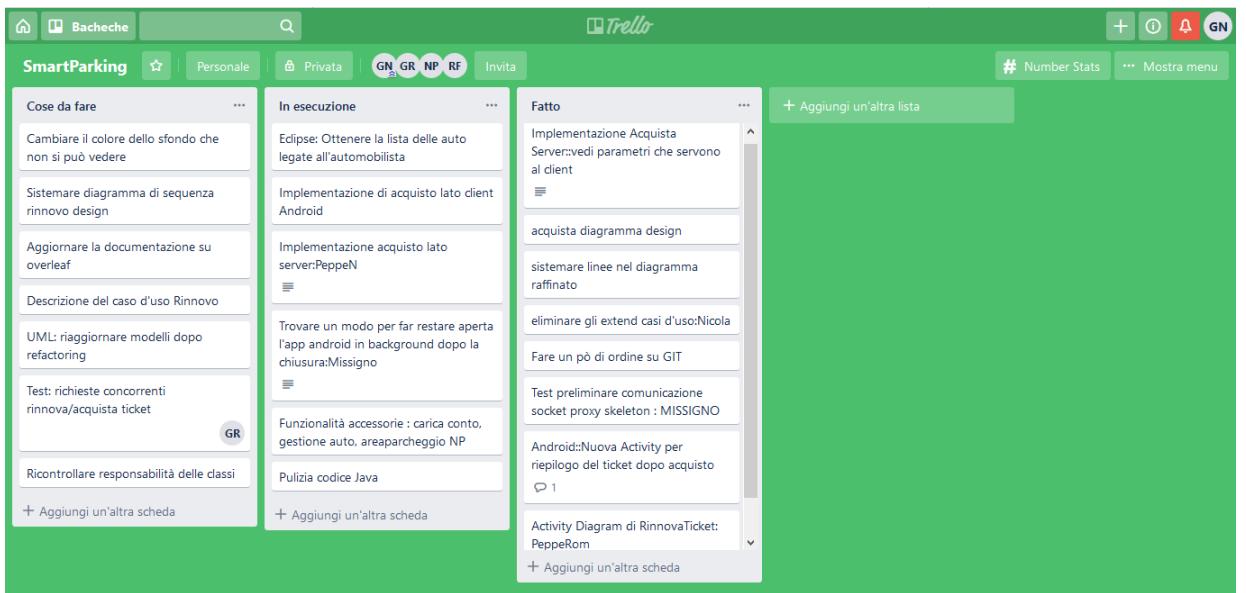


FIGURA 1.2: Screen utilizzo di Trello

Capitolo 2

Analisi

2.1 Analisi Testuale

Ai fini di una precisa comprensione del dominio, è stata effettuata attraverso *Visual Paradigm* l'analisi testuale dei requisiti informali discussi con lo stakeholder. In questa fase sono state determinate le entità del nostro problema e le funzionalità che vengono messe a disposizione. Osserviamo l'individuazione degli attori principali quali automobilista , vigile ,funzionario e l'entità tempo, si sono determinate inoltre i principali servizi del sistema.

No.	Candidate Class	Extracted Text	Type	Description	Occurrences	Highlight
1	controllare la disponibilità	controllare la disponibilità	Use Case		1	
2	terminare la sosta	terminare la sosta	Use Case		1	
3	accede al sistema	accede al sistema	Use Case		1	
4	può eliminare dal sistema uno multiplo eliminare dal sistema	può eliminare dal sistema uno multiplo eliminare dal sistema	Use Case		1	
5	funzionario	funzionario	Actor		2	
6	automobilista	automobilista	Actor		6	
7	utente	utente	Actor		2	
8	tempo	tempo	Actor		1	
9	vigile	vigile	Actor		3	
10	registrazione	registrazione	Use Case		1	
11	approva l'account	approva l'account	Use Case		1	
12	effettua il login	effettua il login	Use Case		1	
13	acquistare un ticket	acquistare un ticket	Use Case		1	
14	accede al portale	accede al portale	Use Case		0	
15	visualizzare lo stato di sosta	visualizzare lo stato di sosta	Use Case		1	
16	notifica	notifica	Use Case		1	
17	innovare le coperture	innovare le coperture	Use Case		1	
18	emettere multa	emettere multa	Use Case		1	

FIGURA 2.1: Visual Paradigm: Analisi Testuale

2.2 Diagramma dei casi d'uso

Successivamente si è creato il diagramma dei casi d'uso, con esso definiamo un confine tra il sistema ed il mondo esterno.

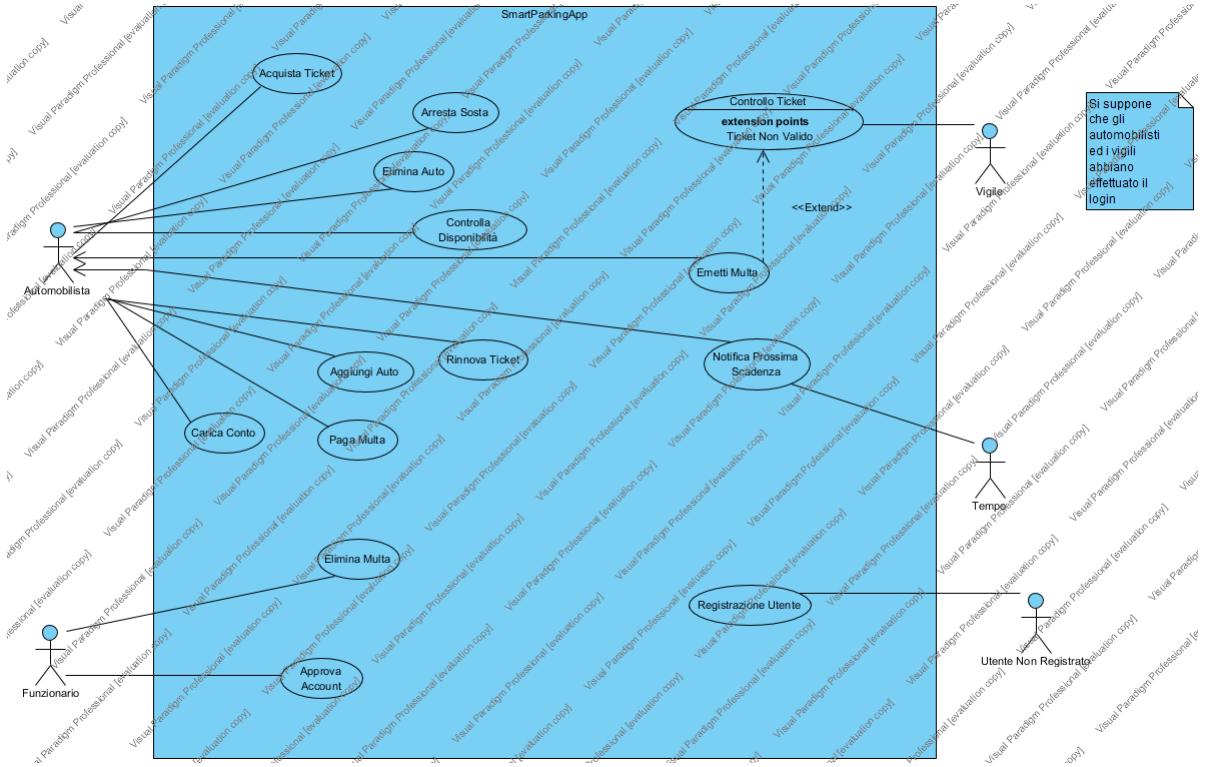


FIGURA 2.2: Visual Paradigm: Diagramma dei casi d'uso

- **Attore Utente non registrato:** L'attore Utente non registrato può registrarsi al sistema, è bene notare che tale funzionalità è utilizzata solo per registrare un utente come automobilista, infatti , il vigile è già registrato al sistema dal comune di provenienza.
- **Attore Automobilista:** L'automobilista una volta loggato può: aggiungere un'auto al sistema, caricare il suo conto, controllare la disponibilità di posti nell'area di parcheggio desiderata, rinnovare o acquistare un ticket informandosi prima sul costo totale della sosta e poi selezionando un'auto dalla lista ad egli associata. Può inoltre arrestare la sua sosta ed ottenere un rimborso per il tempo risparmiato. Se multato l'automobilista può pagare la multa da credito residuo.
- **Attore Vigile:** Un vigile loggato previo il controllo di un ticket può emettere una multa.
- **Attore Tempo :** Viene notificata all'automobilista la scadenza del ticket di uno dei ticket acquistati.

2.2.1 Descrizione dei casi d'uso

Si mostrano brevi descrizioni dei casi d'uso più rilevanti ai fini dell'implementazione di un sottoinsieme di funzionalità del sistema.

8. Controlla Disponibilità

ID: UC26

Primary Actors	♀ Automobilista
Level	User
Complexity	Low
Use Case Status	Complete
Implementation Status	Complete
Preconditions	N/A
Post-conditions	N/A
Author	N/A
Assumptions	N/A

8.1. Scenarios

8.1.1. Scenario

1. L'automobilista decide di verificare se sono presenti posti disponibili in un'area parcheggio
2. L'automobilista inserisce il codice dell'area parcheggio relativo
3. Il sistema fornisce all'automobilista il numero di posti disponibili

8.1.2. Scenario2

1. L'automobilista decide di verificare se sono presenti posti disponibili in un'area parcheggio
2. L'automobilista inserisce un codice area parcheggio errato
3. Il sistema mostra all'automobilista un messaggio di errore riguardo l'inserimento del codice

8.1.3. Scenario3

1. L'automobilista decide di verificare se sono presenti posti disponibili in un'area parcheggio
2. L'automobilista inserisce il codice dell'area parcheggio relativo
3. Il sistema mostra che NON sono presenti posti liberi nell'area parcheggio

FIGURA 2.3: Visual Paradigm: Descrizione del caso d'uso Controllo Disponibilità

●4. Aggiungi Auto

ID: UC20

Primary Actors	♀ Automobilista
Level	User
Complexity	Low
Use Case Status	Complete
Implementation Status	Complete
Preconditions	N/A
Post-conditions	L'auto è associata all'automobilista
Author	N/A
Assumptions	N/A

4.1. Scenarios

4.1.1. Scenario

1. L'automobilista decide di effettuare un inserimento di un auto
2. L'automobilista specifica la targa e il codice fiscale del proprietario
3. Il sistema inserisce l'auto in maniera persistente
4. Il sistema associa l'auto inserita all'automobilista

4.1.2. Scenario2

1. L'automobilista decide di effettuare un inserimento di un auto
2. L'automobilista specifica la targa e il codice fiscale del proprietario
3. Il sistema notifica l'automobilista dell'inserimento errato dei dati

FIGURA 2.4: Visual Paradigm: Descrizione del caso d'uso Aggiungi Auto

●5. Carica Conto

ID: UC03

Primary Actors	♀ Automobilista
Level	User
Complexity	Low
Use Case Status	Complete
Implementation Status	Complete
Preconditions	N/A
Post-conditions	N/A
Author	N/A
Assumptions	N/A

5.1. Scenarios

5.1.1. Scenario

1. L'automobilista seleziona il metodo di pagamento
2. L'automobilista inserisce l'importo da ricaricare
3. Il sistema aggiorna il credito residuo dell'automobilista

5.1.2. Scenario2

1. L'automobilista seleziona il metodo di pagamento
2. L'automobilista inserisce l'importo da ricaricare
3. Il sistema notifica l'automobilista del fallimento della transazione per insufficiente credito bancario

FIGURA 2.5: Visual Paradigm: Descrizione del caso d'uso carica conto

●2. Acquista Ticket

ID: UC04

Primary Actors	Automobilista
Level	User
Complexity	Medium
Use Case Status	Complete
Implementation Status	Complete
Preconditions	L'automobilista deve possedere un credito residuo maggiore del costo del Ticket L'automobilista deve possedere almeno un'auto associata
Post-conditions	Il credito residuo viene aggiornato ed il ticket viene memorizzato in maniera persistente dal sistema
Author	N/A
Assumptions	N/A

2.1. Scenarios

2.1.1. Scenario

1. L'automobilista richiede al sistema la lista di auto associate
2. Il sistema restituisce una lista di auto
3. L'automobilista seleziona una delle auto appartenenti alla lista
4. L'automobilista richiede il costo del ticket specificando il codice dell'area parcheggio e la durata della sosta
5. Il sistema elabora e restituisce il costo del ticket
6. L'automobilista procede all'acquisto del ticket
7. Il credito residuo viene aggiornato ed il ticket viene memorizzato in maniera persistente dal sistema

2.1.2. Scenario2

1. L'automobilista richiede al sistema la lista di auto associate
2. Il sistema restituisce una lista di auto vuota

2.1.3. Scenario3

1. L'automobilista richiede al sistema la lista di auto associate
2. Il sistema restituisce una lista di auto
3. L'automobilista seleziona una delle auto appartenenti alla lista
4. L'automobilista richiede il costo del ticket specificando il codice dell'area parcheggio e la durata della sosta
5. Il sistema elabora e restituisce il costo del ticket
6. L'automobilista decide di non procedere all'acquisto del ticket

2.1.4. Scenario4

1. L'automobilista richiede al sistema la lista di auto associate
2. Il sistema restituisce una lista di auto
3. L'automobilista seleziona una delle auto appartenenti alla lista
4. L'automobilista richiede il costo del ticket specificando il codice dell'area parcheggio e la durata della sosta
5. Il sistema elabora e restituisce il costo del ticket
6. L'automobilista procede all'acquisto del ticket
7. Il sistema avvisa l'automobilista del credito insufficiente per l'acquisto

FIGURA 2.6: Visual Paradigm: Descrizione del caso d'uso AcquistaTicket

3. Rinnova Ticket

ID: UC05

Primary Actors	Automobilista
Level	User
Complexity	Medium
Use Case Status	Complete
Implementation Status	Complete
Preconditions	Esiste un ticket che scadrà entro 10 minuti
Post-conditions	Il credito ed il ticket vengono aggiornati e resi persistenti nel sistema
Author	N/A
Assumptions	N/A

3.1. Scenarios

3.1.1. Scenario

1. Il sistema notifica l'automobilista della prossima scadenza del ticket
2. L'automobilista specifica la durata di rinnovo del ticket
3. Il sistema calcola e restituisce il costo del rinnovo
4. L'automobilista procede al rinnovo del ticket
5. Il credito ed il ticket vengono aggiornati e resi persistenti nel sistema

3.1.2. Scenario2

1. Il sistema notifica l'automobilista della prossima scadenza del ticket
2. L'automobilista specifica la durata di rinnovo del ticket
3. Il sistema calcola e restituisce il costo del rinnovo
4. L'automobilista procede al rinnovo del ticket
5. Il sistema avvisa l'automobilista del credito insufficiente al rinnovo del ticket

FIGURA 2.7: Visual Paradigm: Descrizione del caso d'uso RinnovaTicket

2.3 Diagramma di contesto

Si mostra il diagramma di contesto ,in esso mostriamo una rappresentazione sintetica delle relazioni del sistema di SmartParking con l'ambiente esterno evidenziandone i device con i quali il sistema interagisce e la relativa molteplicità.

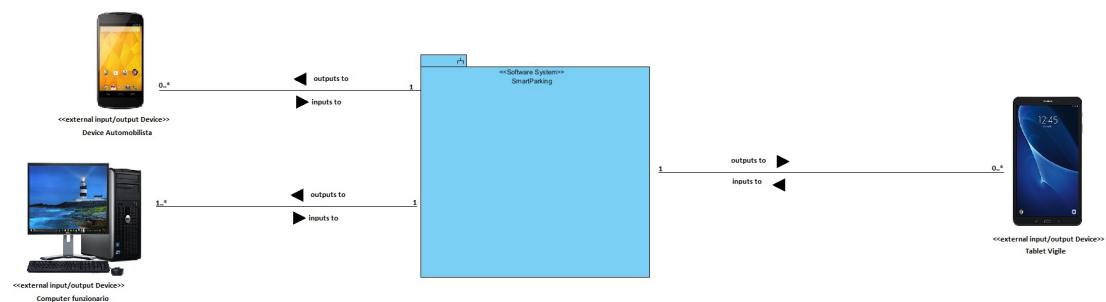


FIGURA 2.8: Diagramma di contesto

2.4 Diagramma delle classi concettuale

Per comprendere meglio le entità del nostro problema si mostra un primo diagramma delle classi. Esso è un diagramma delle classi concettuale che ci permette di catturare il "che cosa si intende realizzare" essendo però ancora lontani dall'implementazione.

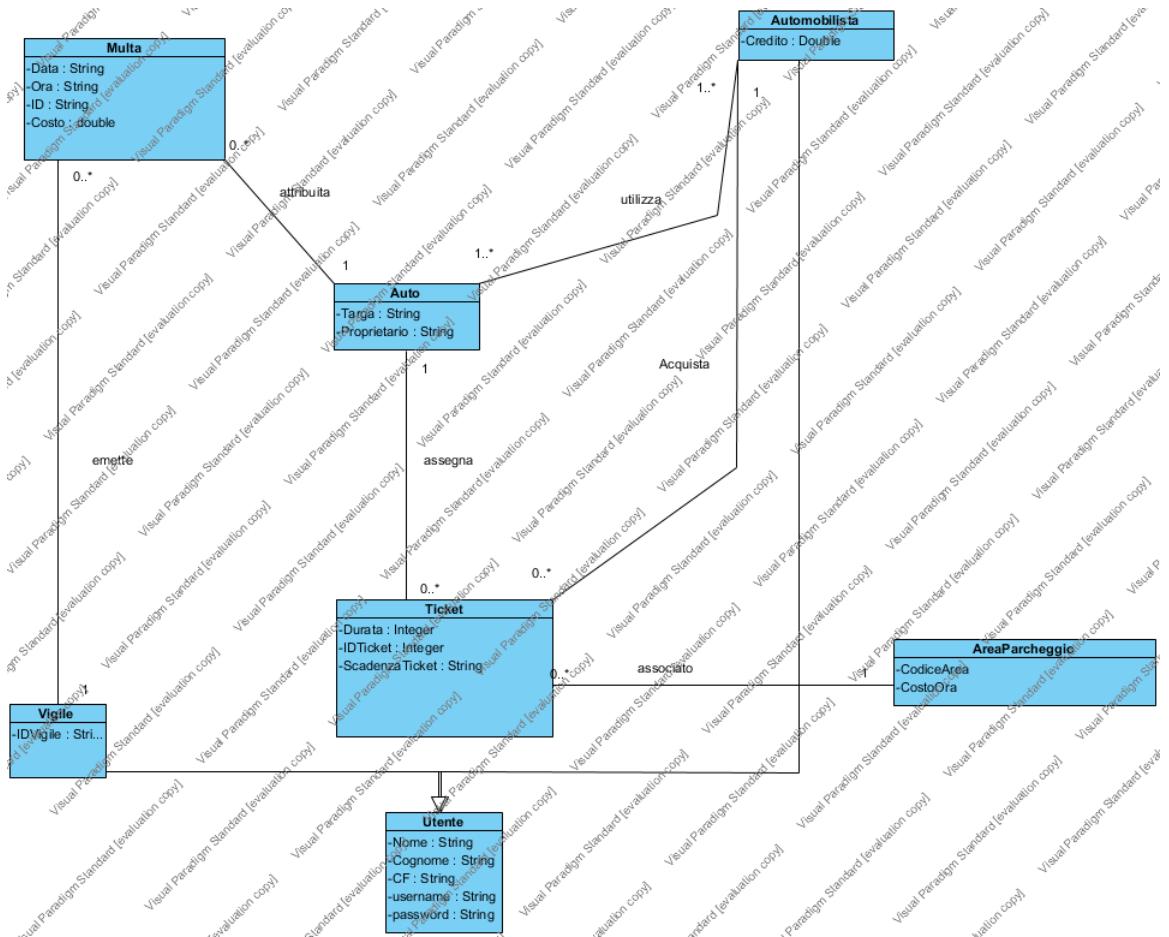


FIGURA 2.9: Visual Paradigm: Diagramma delle classi concettuale

L'utente può essere un automobilista oppure un vigile, ciò è stato mostrato con una gen-spec, si noti come l'automobilista ha un attributo credito e il vigile ha un attributo IDVigile per tener traccia dello specifico vigile che presta servizio. Ogni automobilista può utilizzare più auto, non solo quelle di cui è proprietario. Ogni ticket è relativo ad una sola auto e ad una sola areaParcheggio. Inoltre si tiene traccia dell'automobilista che acquista il Ticket al fine di notificarlo per l'eventuale rinnovo.

2.5 Diagramma delle classi raffinato

A partire dal diagramma precedente, si è effettuato un raffinamento introducendo le classi **Boundary** e **Control**, che rappresentano rispettivamente le interfacce tra gli attori e il sistema, e il responsabile del passaggio delle richieste utente alla logica applicativa.

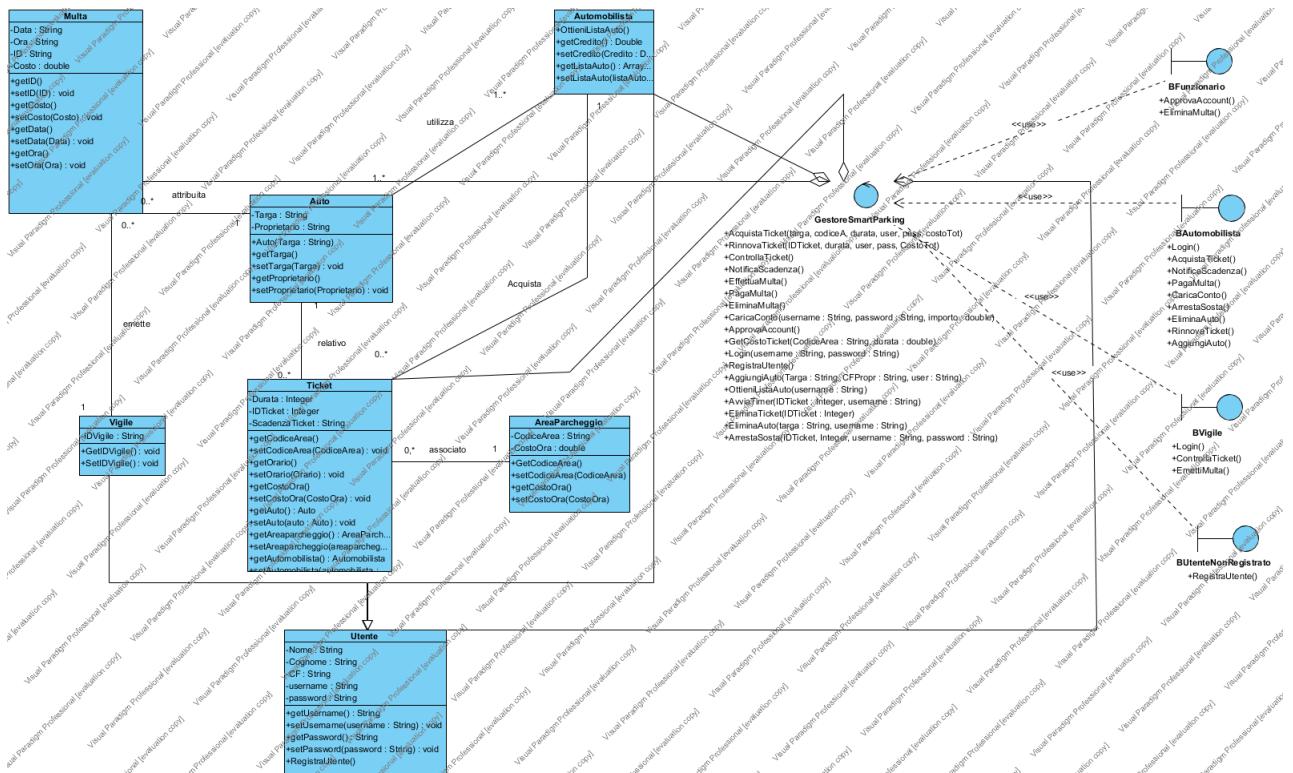


FIGURA 2.10: Visual Paradigm: Diagramma delle classi raffinato

2.6 Architettura software

Questo sistema è stato costruito basandosi sullo stile architetturale *Client/Server*, poichè il client (utente su dispositivo mobile) non fa altro che inviare richieste ad un unico Server remoto restando in attesa di un eventuale risposta. Per realizzare la comunicazione client-server, i client hanno bisogno di conoscere l'indirizzo del Server e non viceversa. In particolare essa è un architettura *two-tier* in cui da un lato vi è Client che ha logica di presentazione e comunicazione (proxy) mentre sul server vi è la business logic e la resource manager.

2.7 Diagramma dei componenti

Tale stile prevede l'utilizzo di due tipi di componenti, che sono appunto i Client e il Server. In particolare, ci sono quattro client: Client android vigili ed automobilista, client utente non registrato che ha a disposizione un interfaccia per effettuare la registrazione ed infine il funzionario che può approvare le richieste di registrazione. In particolare sono stati implementati i client Android automobilista e Utente non registrato, tutti gli altri componenti non sono stati implementati. I connettori invece consistono in *Stream Socket* utilizzate dai Client per accedere alle funzionalità messe a disposizione dal Server. Quest'ultima garantisce una connessione affidabile (utilizzati dal protocollo TCP): il server si mette in ascolto su una porta (connection socket), ad ogni nuova richiesta di collegamento dei client, crea, per ognuno, un “canale virtuale” per la comunicazione tra client e server. Inoltre, è stato aggiunto un componente Database per la gestione e memorizzazione dei dati, legato al Server da un connettore JDBC, che ne consente l'accesso e la gestione della persistenza.

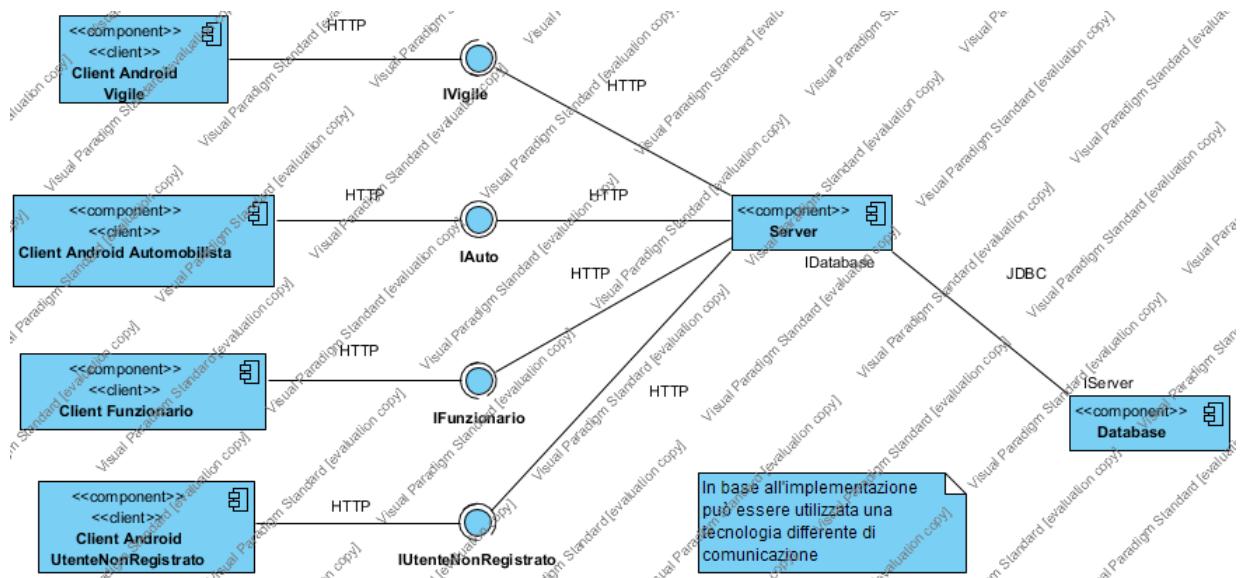


FIGURA 2.11: Visual Paradigm: Diagramma delle componenti

2.8 Casi d'uso implementati

In questa sezione si mostrano i diagrammi di sequenza delle funzionalità che si intende progettare: *AcquistaTicket* e *RinnovaTicket*. Al fine di una corretta esecuzione e di un utilizzo concreto da parte dell'utente si è scelto di analizzare, progettare ed implementare funzionalità di contorno quali:

- **AggiungiAuto:** L'automobilista aggiunge un auto alla lista delle auto che utilizza.

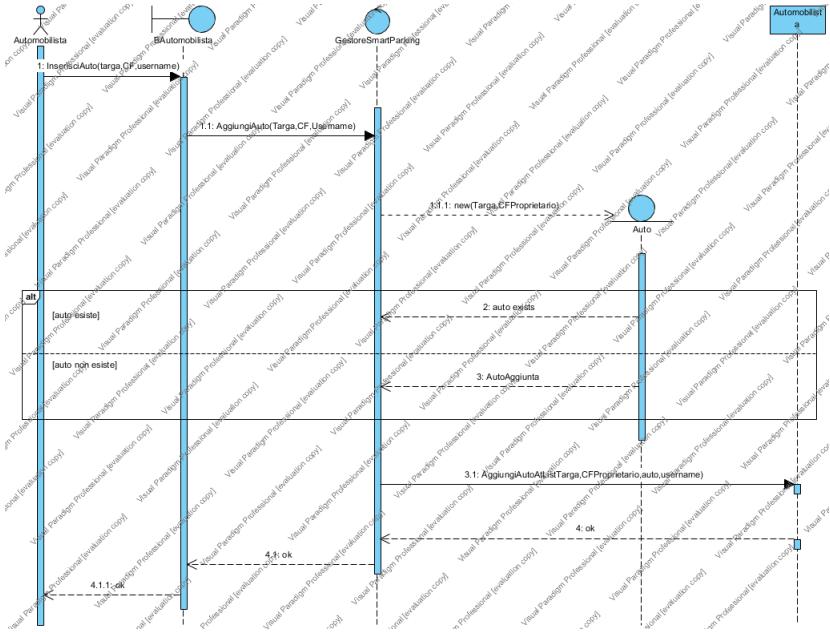


FIGURA 2.12: Visual Paradigm: Diagramma di sequenza di Aggiungi Auto

- **ControlloDisponibilità:** L'automobilista può controllare l'effettiva presenza di posti auto prima di recarsi in loco per effettuare l'acquisto del ticket.
- **ArrestoSosta:** L'automobilista può arrestare la sua sosta prima della scadenza del ticket ottenendo un rimborso calcolato al minuto.
- **CaricaConto:** L'automobilista può ricaricare il conto ad egli associato.

2.8.1 Diagramma ad alto livello

Al fine di una migliore comprensione dei casi d'uso principali si mostrano i diagrammi di attività di AcquistaTicket e RinnovaTicket.

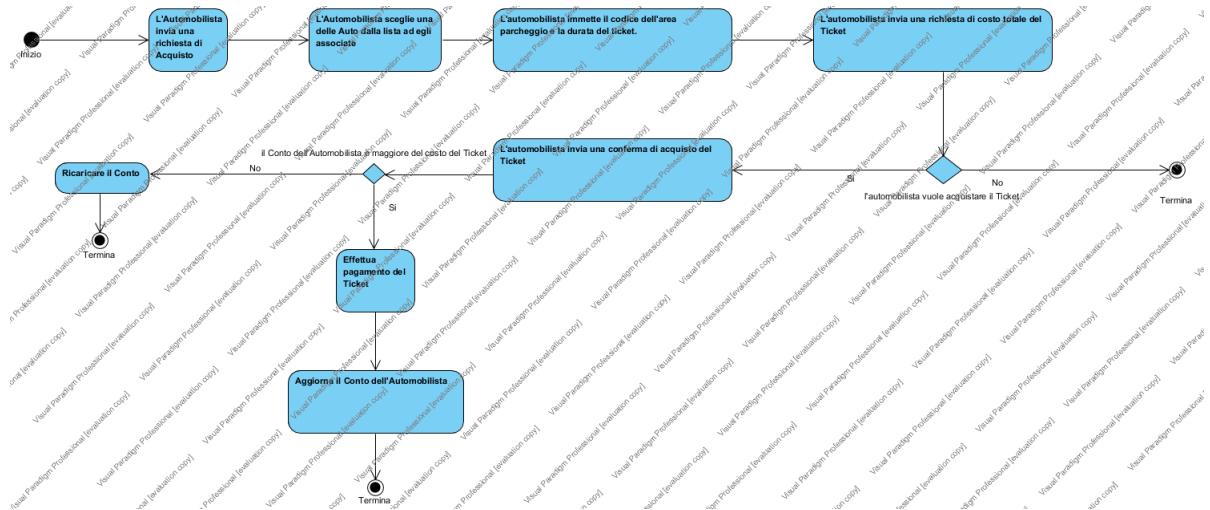


FIGURA 2.13: Visual Paradigm: Diagramma di attività di AcquistaTicket

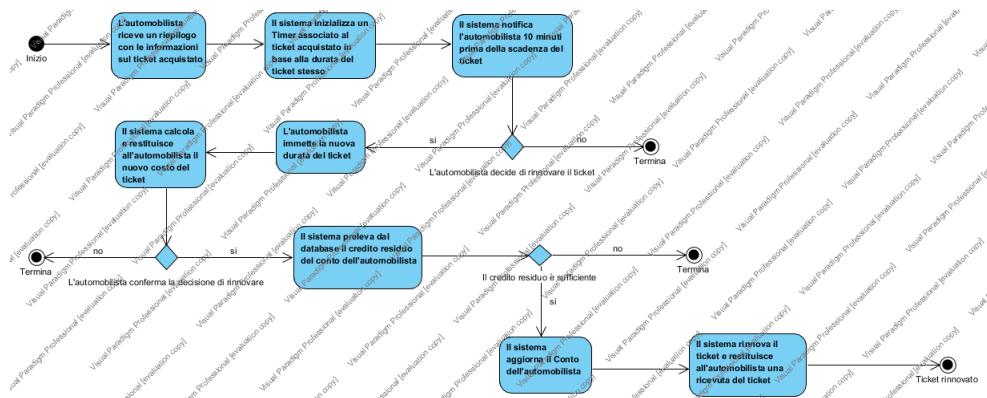


FIGURA 2.14: Visual Paradigm: Diagramma di attività di RinnovaTicket

2.9 Caso d'uso AcquistaTicket

In questo diagramma è descritto il processo con il quale l'automobilista riesce a portare a termine l'acquisto di un ticket. A partire da un interfaccia grafica l'automobilista cliccando sul bottone di Acquisto accede ad un'altra pagina in cui tramite un menù a tendina seleziona l'auto a partire dalle targhe associate a se stesso. Successivamente l'automobilista inserirà il codice dell'area di parcheggio e la durata della sua sosta. Il sistema provvederà a calcolare il costo complessivo del ticket e restituire tale risultato all'automobilista. Qualora il cliente sia ancora

interessato all'acquisto e il suo credito residuo sia sufficiente verrà restituita la ricevuta del ticket, altrimenti verrà esortato l'automobilista a ricaricare il credito.

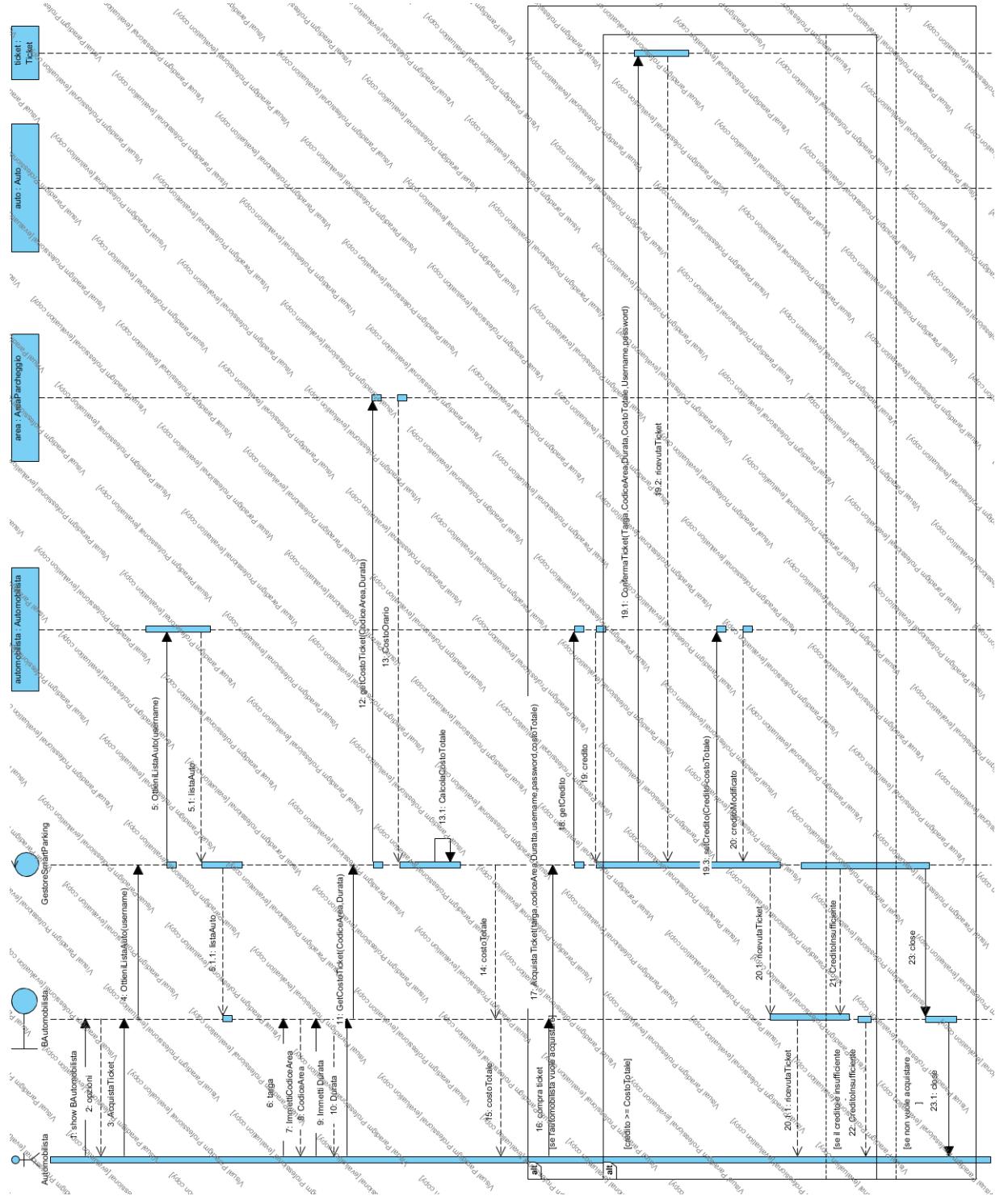


FIGURA 2.15: Visual Paradigm: Diagramma di sequenza del caso d'uso AcquistaTicket

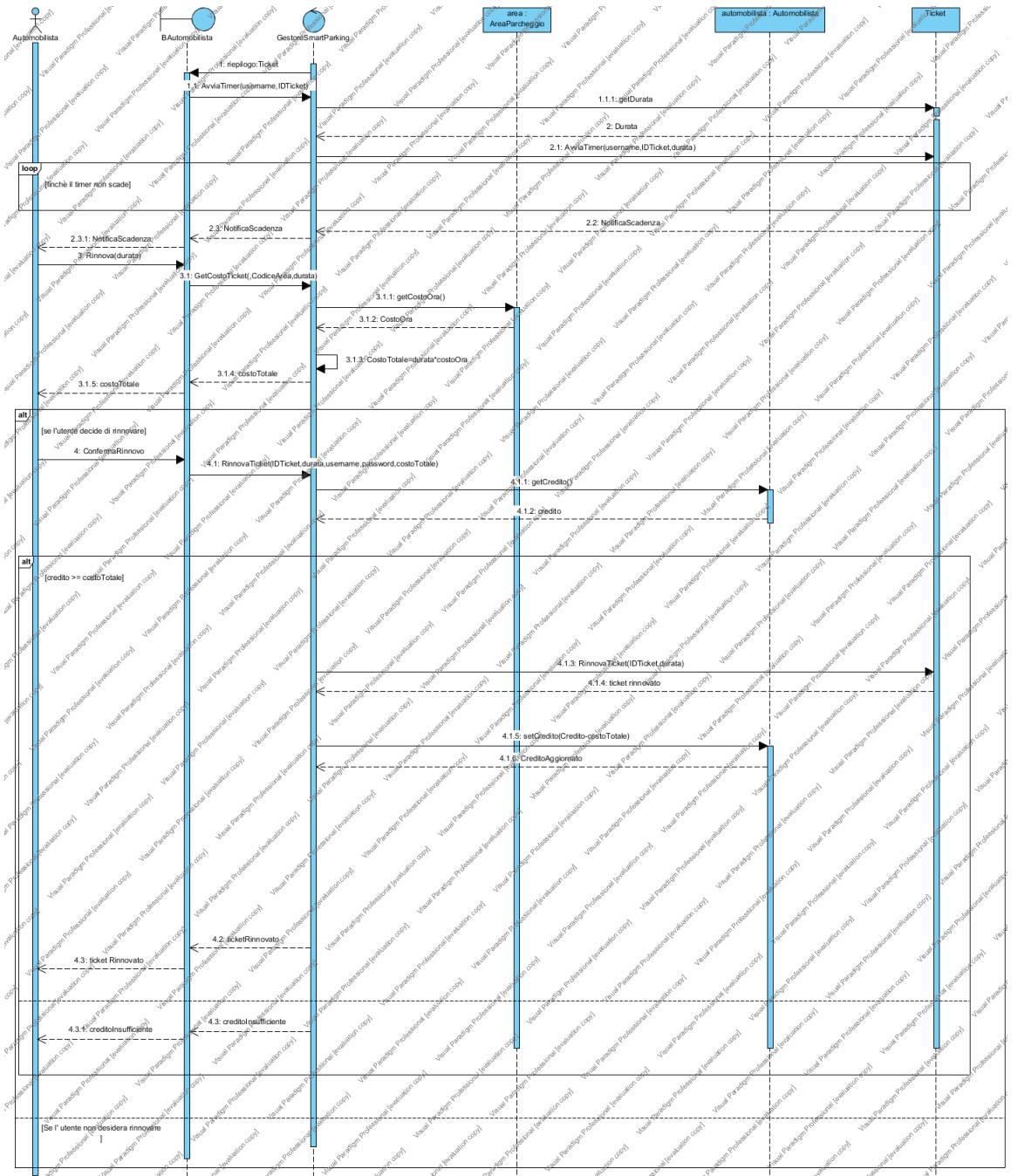


FIGURA 2.16: Visual Paradigm: Diagramma di sequenza del caso d'uso Rinnova Ticket

Terminato l'acquisto il cliente potrà visualizzare un riepilogo contenente le informazioni sul ticket, quest'ultimo avrà un timer associato che scadrà 10 minuti prima del termine della sosta. Allo scadere del ticket verrà notificato il cliente che potrà decidere di rinnovare o meno. Se il cliente vuol rinnovare inserirà una nuova durata ed il sistema provvederà a calcolare il costo totale del ticket. Qualora l'utente dopo aver visualizzato il costo totale voglia ancora rinnovare verrà aggiornato il ticket già presente con una nuova data di scadenza.

Capitolo 3

Progettazione

3.1 Diagramma delle classi di Progetto

Il diagramma delle classi di progetto è ottenuto raffinando il diagramma di analisi. Le classi di progettazione sono state raggruppate logicamente in macro-package come mostrato in figura, identifichiamo un package Boundary per contenere gli elementi appartenenti al client android, un package che contenga invece la business logic del sistema ed un altro per le classi entity. Infine introduciamo un package per contenere le implementazioni delle classi controller, ed uno per le classi DAO. Per migliorare la qualità interna del software si è preferito l'uso di interfacce a classi, in modo da avere dipendenze di servizi , piuttosto che di implementazione. Infatti si sono sviluppati i controller basandoci su interfacce specifiche (GestoreAccount, GestoreTicket, GestoreMulte) che sono state raggruppate in un facade. L'implementazione concreta dei controller è realizzata nel package ControllerImpl. Il facade è composto a runtime grazie ad una classe Builder, in questo modo i controller saranno facilmente sostituibili, possiamo infatti isolarli e sostituirli durante il testing, migliorando la manutenibilità , testabilità e soprattutto l'evolvibilità dato che questa creata è solo una prima release.

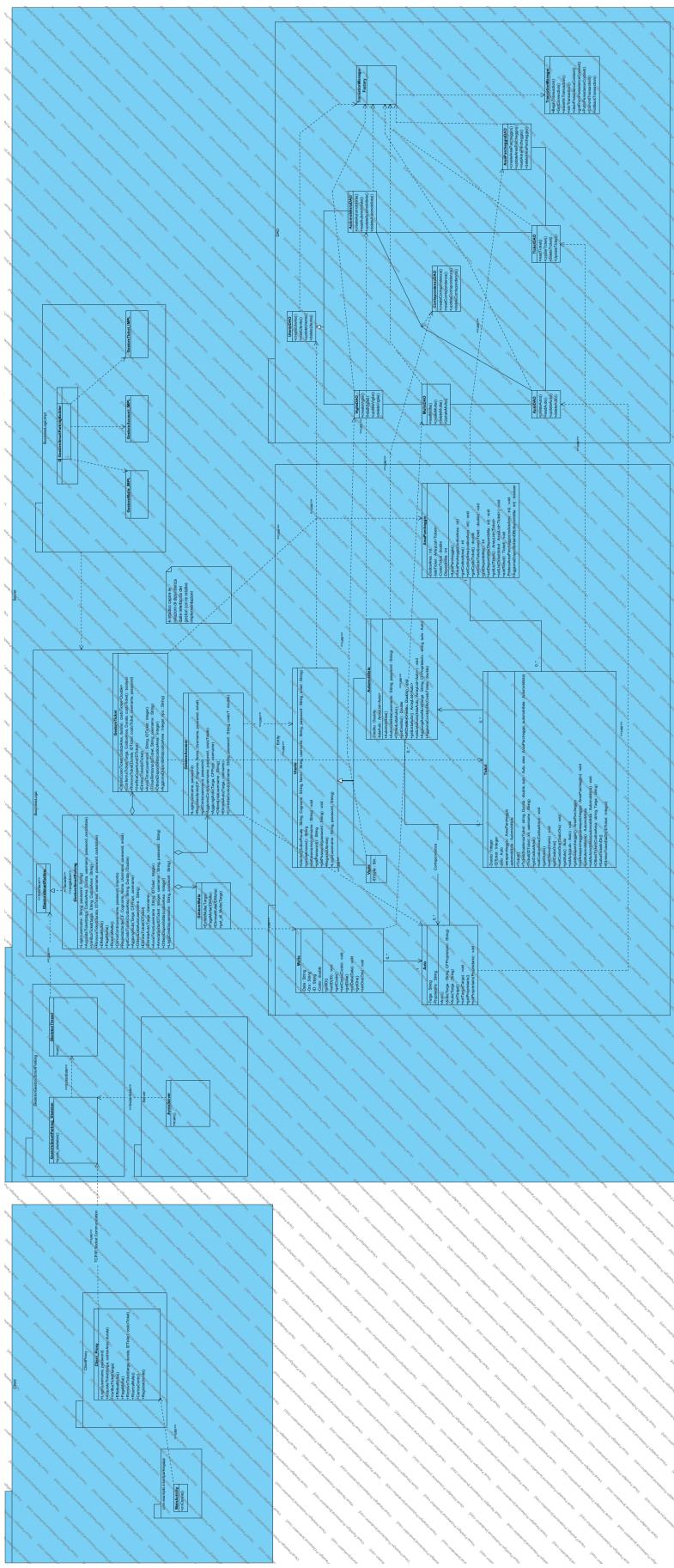


FIGURA 3.1: Visual Paradigm: Diagramma delle classi di progetto

3.1.1 Proxy-Skeleton

Per effettuare la comunicazione da remoto tra Client e Server è stato utilizzato il pattern Proxy. Quest'ultimo fornisce una rappresentazione locale dell'oggetto remoto con cui si vuole interagire. Si osserva infatti dalla figura successiva che il Client interagisce con una sua Proxy che è una rappresentazione in locale del GestoreSmartParking. In tal modo dal client Android sarà possibile inviare dati attraverso le Socket verso il Server. Anche dal lato Server, viene utilizzata la classe GestoreSmartParking Skeleton, in modo da disaccoppiare la logica relativa alla comunicazione da quella relativa all'implementazione delle funzionalità. Il server una volta avviato rimane in attesa di una richiesta di connessione dal client e successivamente creerà un thread per ogni client che intende connettersi.

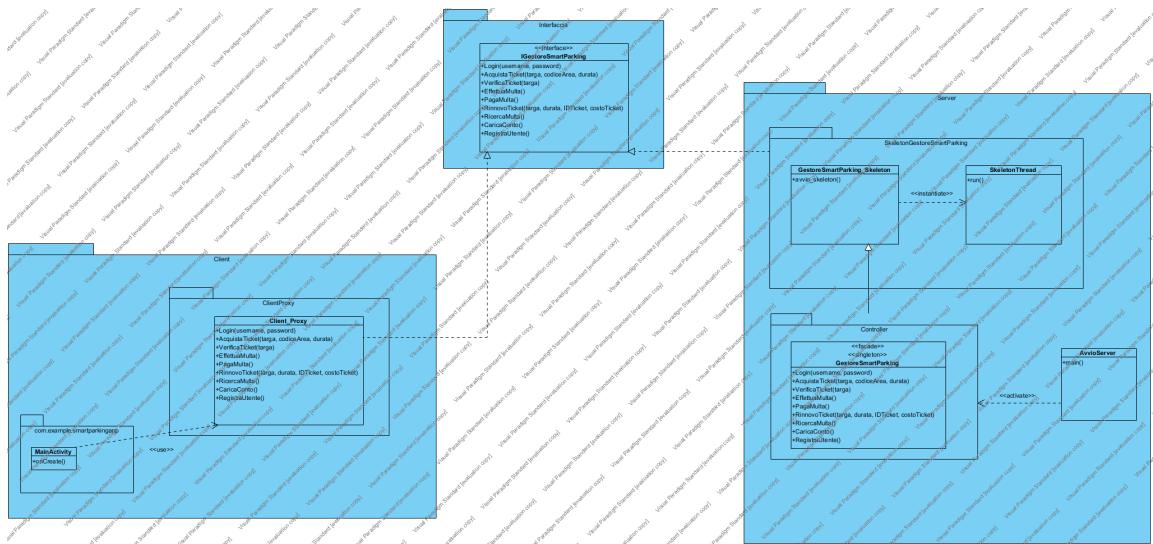


FIGURA 3.2: Visual Paradigm: Proxy-Skeleton

3.2 Deploymemt diagram

Si mostra adesso un diagramma di deployment, in esso mostriamo come gli Artifacts (elementi risultanti dal processo di sviluppo) vengono distribuiti sui nodi di elaborazione. Si può osservare come i client Android Vigili e Automobilista accedano alla stessa applicazione con privilegi diversi. Dunque contrariamente ad Automobilisti e vigili che utilizzano dispositivi mobile, i funzionari hanno una postazione fissa con un sistema operativo Windows/Linux/Mac OS, in quanto rappresentano personale di ufficio ed amministrativo. L'artefatto server.jar 'è distribuito su un ulteriore terminale che comunica con un database mySQL. La comunicazione con il client avviene su rete attraverso l'utilizzo del protocollo TCP/IP.

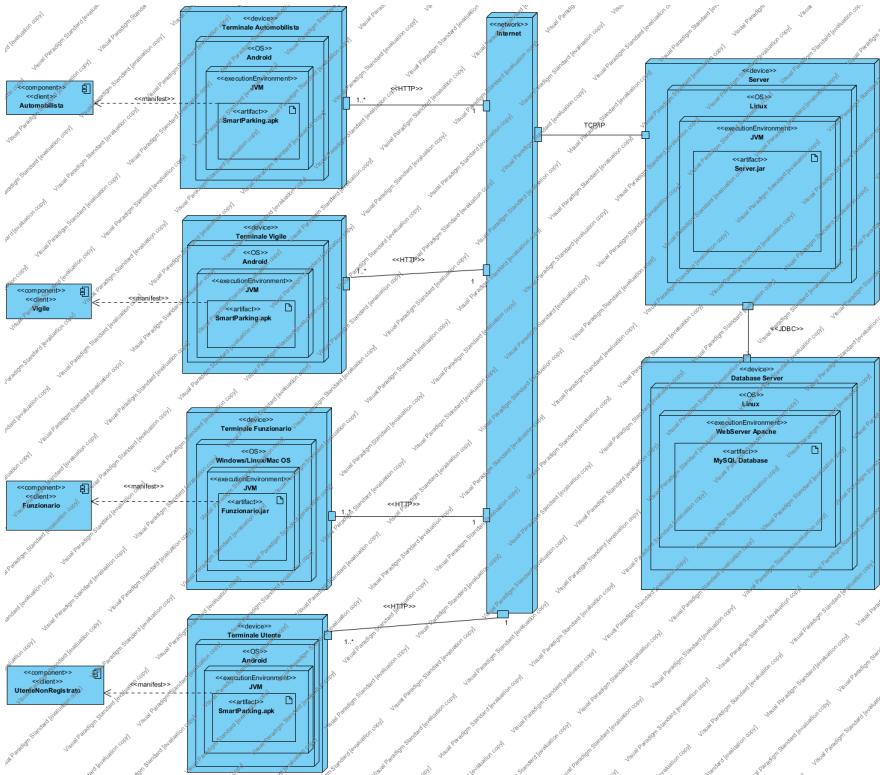


FIGURA 3.3: Visual Paradigm: Deployment diagram

3.3 Diagramma di sequenza AcquistaTicket

Si è raffinato il diagramma di analisi mostrato al capitolo precedente, introducendo qualora necessario le classi DAO per la comunicazione con il database, inoltre si è dettagliata la creazione degli oggetti tramiti appositi costruttori che prelevano le informazioni da DB e popolano gli oggetti. **Per visualizzare meglio il diagramma si rimanda al progetto di visual paradigm.**

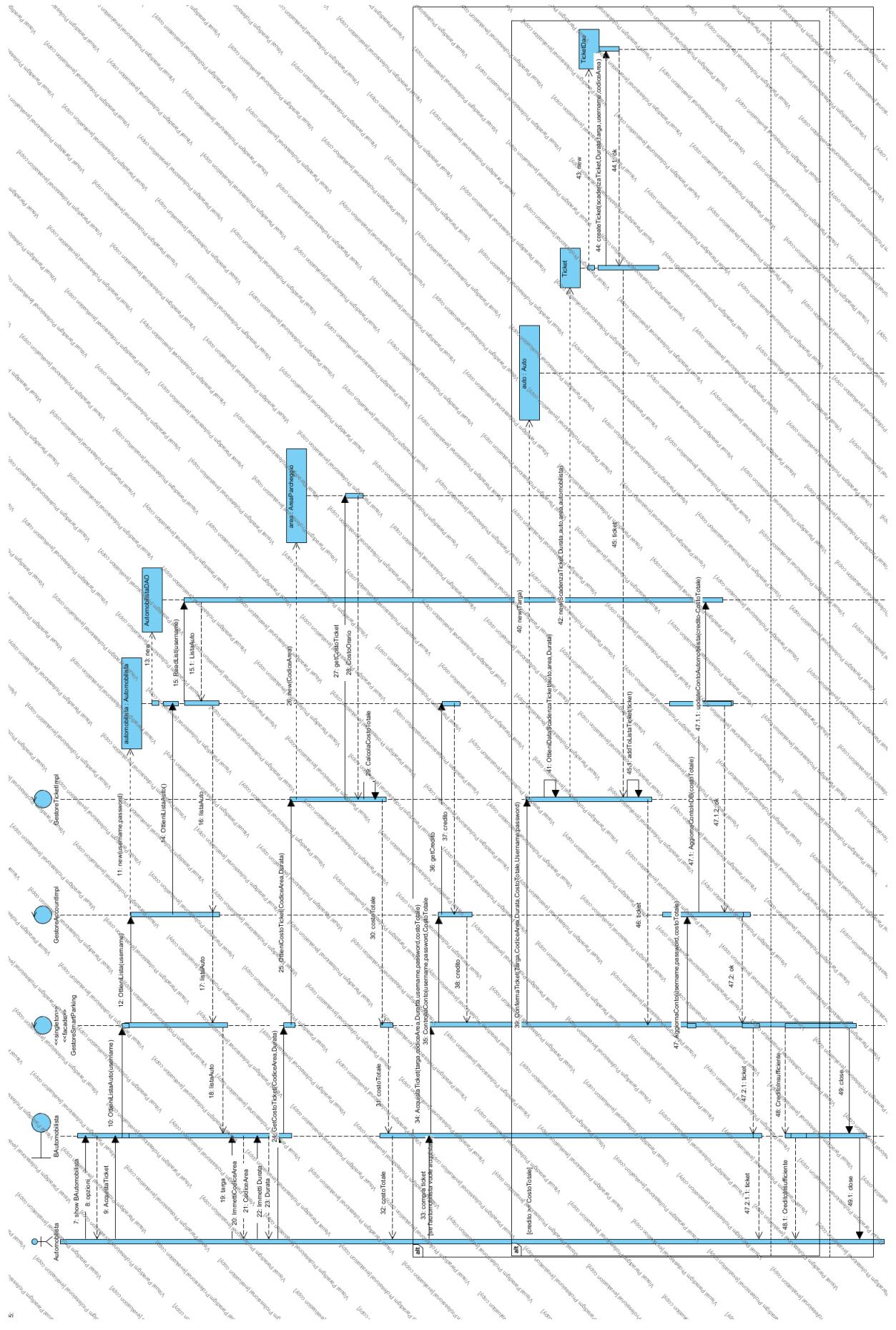


FIGURA 3.4: Visual Paradigm: AcquistaTicket di dettaglio

3.4 Diagramma di sequenza RinnovaTicket

Anche in questo caso si è raffinato il relativo diagramma di analisi. In particolare si mostra come gli appositi gestori gestiscano il rinnovo del ticket. Il gestore relativo ai ticket ha una lista contenente l'insieme dei ticket acquistati. Ad ognuno di essi è associato un timer che scatta al momento dell'acquisto e scade 10 minuti prima del termine della sosta. Come si nota nell'attesa del scadenza del timer l'utente può interrompere l'attesa arrestando la sosta. Si è a tal proposito progettato un sub-diagramma di sequenza che mostri il processo per terminare un ticket e restituire il dovuto rimborso al cliente.

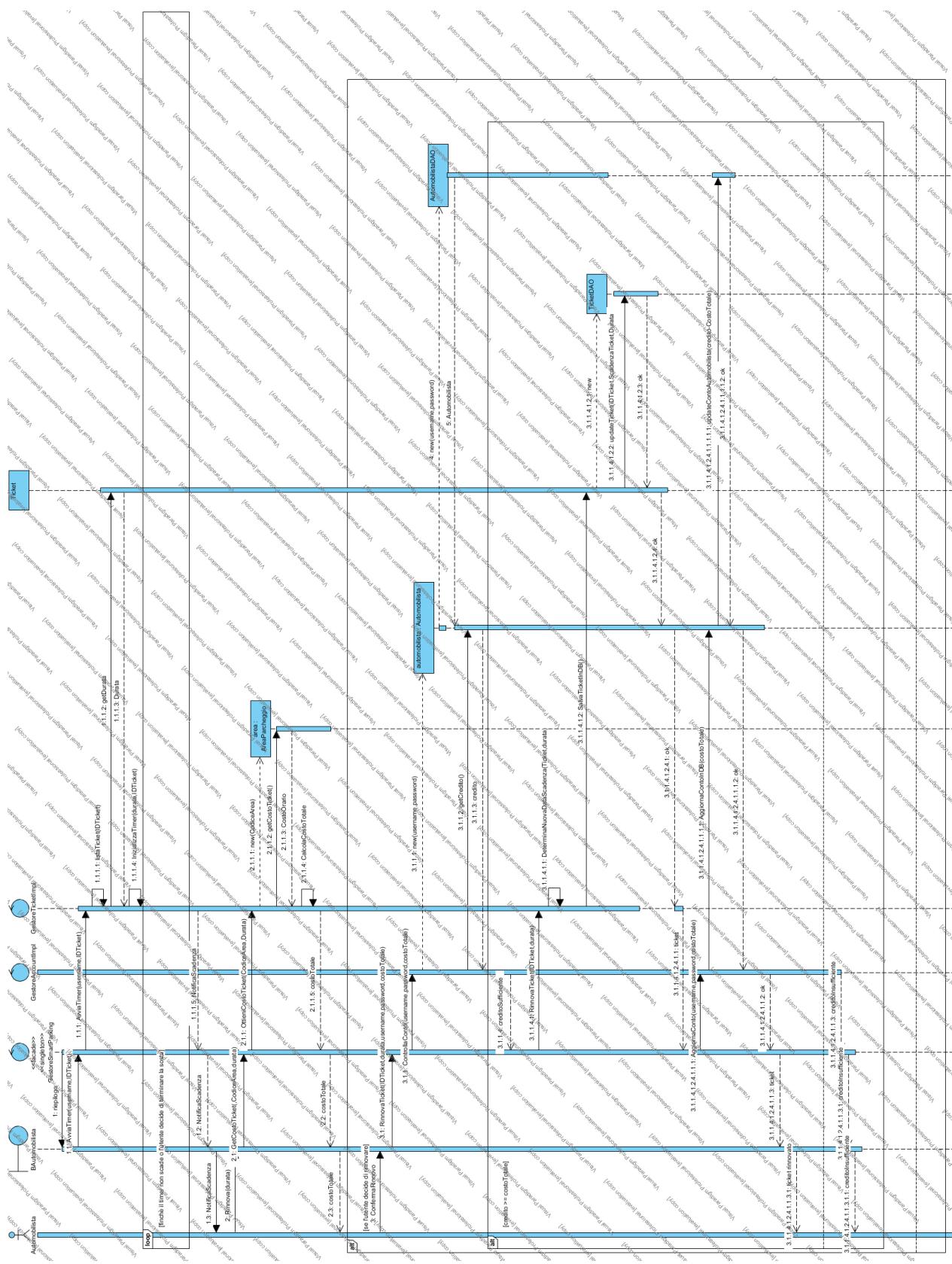


FIGURA 3.5: Visual Paradigm: RinnovaTicket di dettaglio

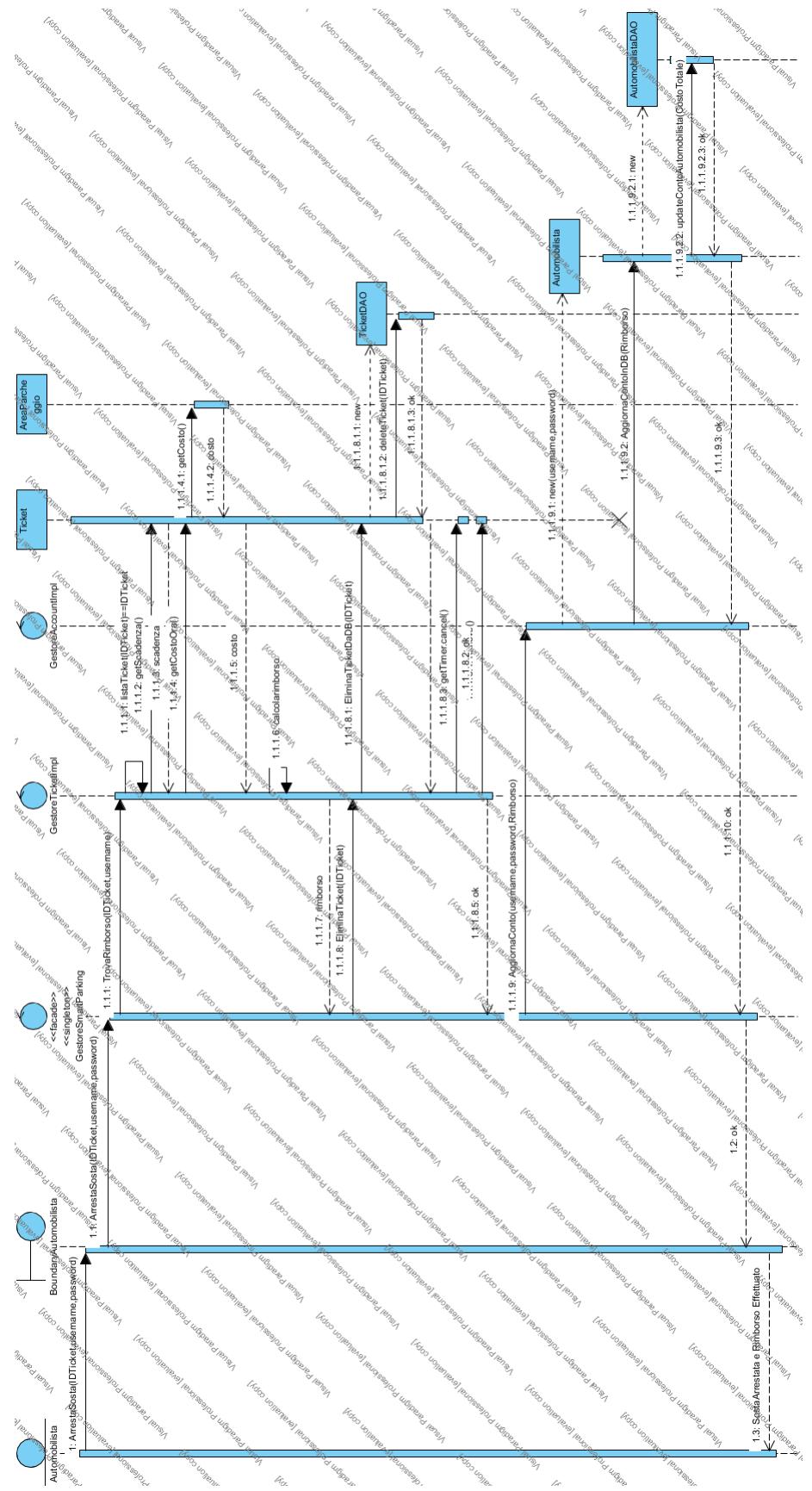


FIGURA 3.6: Visual Paradigm: Arresta Sostan

3.5 Vista Client Android

In questa sezione si mostra la struttura della logica interna al client ed un diagramma di sequenza che mostra il comportamento ai fini della funzionalità di acquisto del ticket.

3.5.1 Diagramma delle classi lato client

In questo diagramma viene mostrata la struttura dell'applicazione android realizzata attraverso l'uso dell'IDE *Android Studio*. Nel package *res/layout* sono presenti le interfacce grafiche in formato **XML** che realizzano il layout di ogni activity presente. Le varie activity attraverso il metodo **onCreate** realizzano la grafica XML associata. A loro volta per effettuare una comunicazione si affidano al proxy il quale gestisce la comunicazione lato client verso il server. Infatti una classe **SocketHandler** mantiene i riferimenti della connessione TCP/IP con il server (Socket,InputStream e OutputStream). Per semplicità visiva sono state mostrate soltanto una parte delle classi e dei metodi dell'applicazione. Infine il package ClientServerCommunication viene utilizzato per riferirsi ad un ulteriore diagramma nel quale è implementata la logica di ricezione da parte del server.

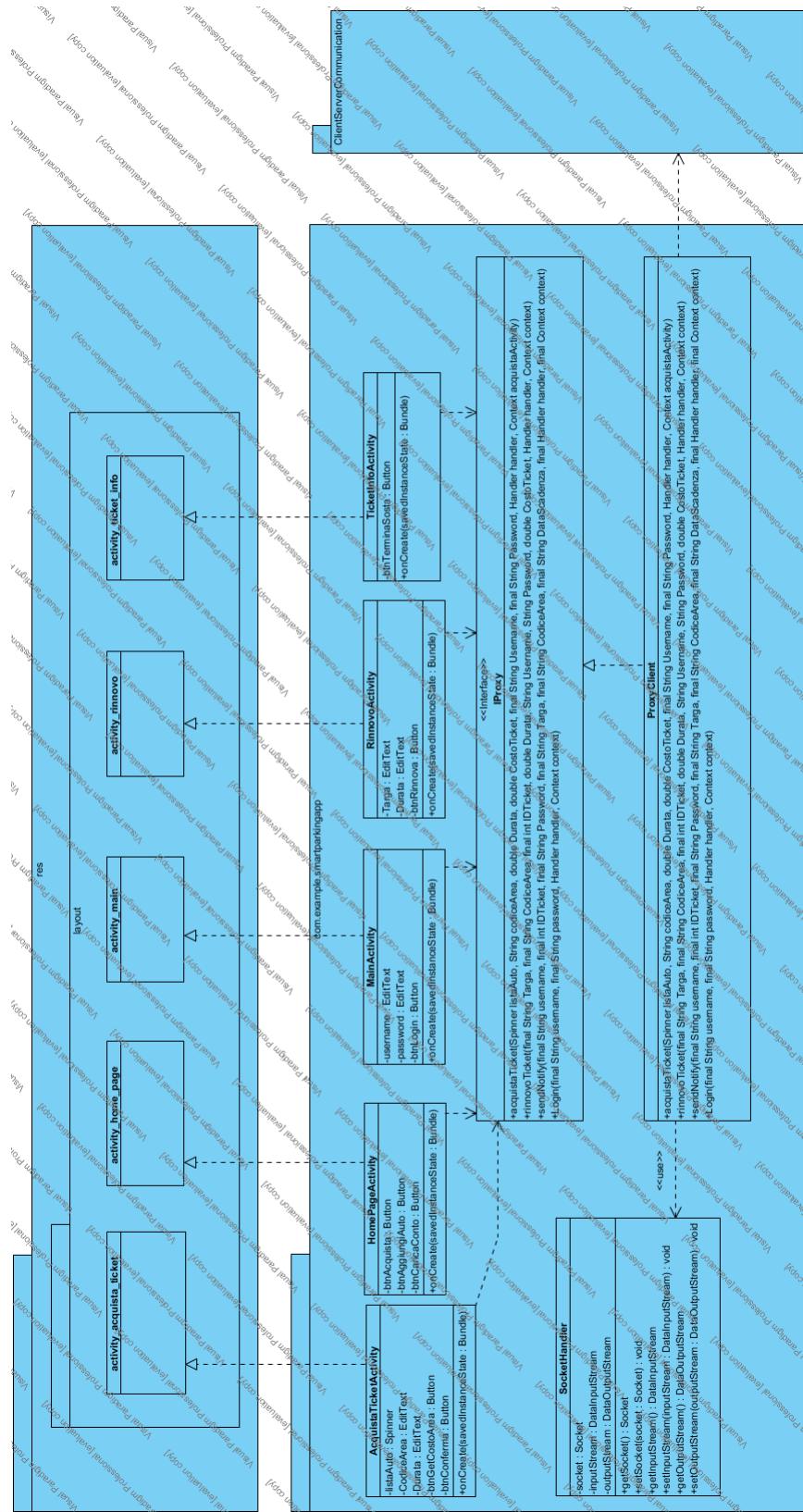


FIGURA 3.7: Visual Paradigm: Vista lato client

3.5.2 Diagramma di sequenza AcquistaTicket lato client android

Nel seguente sequence diagram viene mostrata come avviene una richiesta di acquisto del ticket lato client.

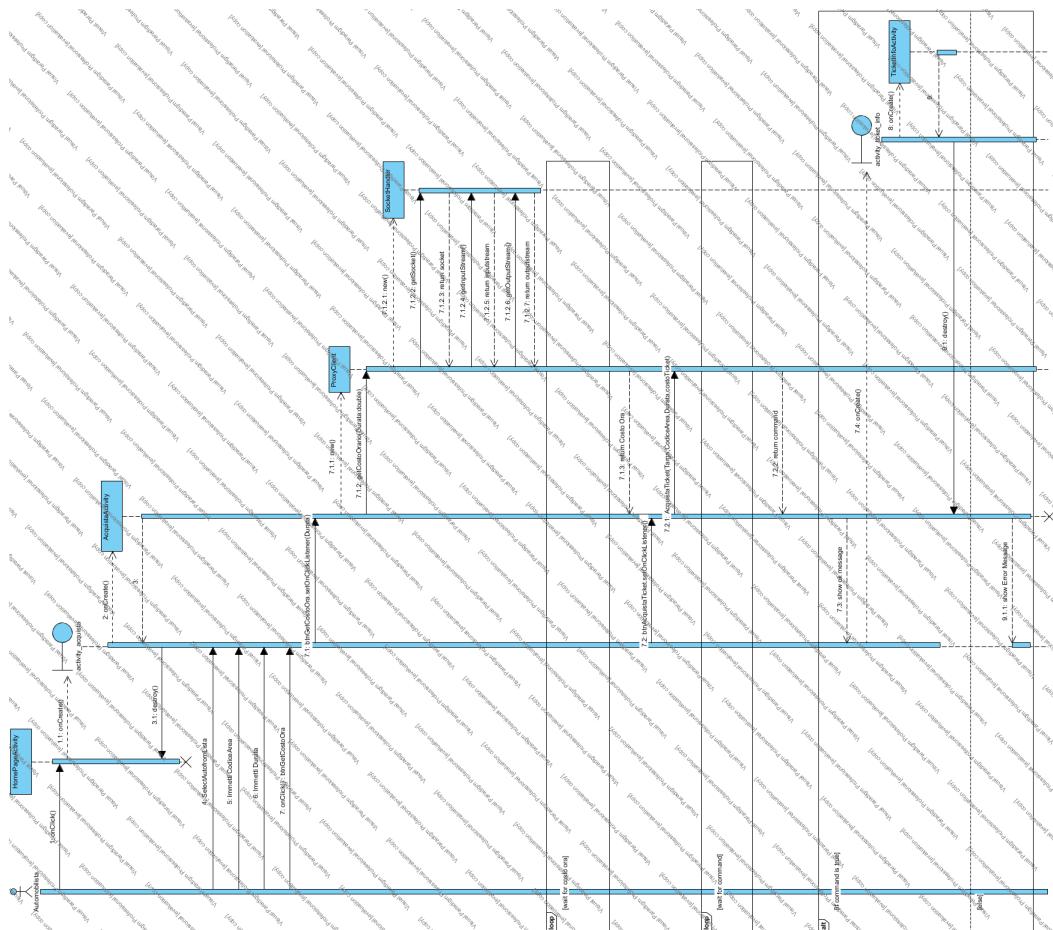


FIGURA 3.8: Visual Paradigm: Acquista lato-client

Conclusioni e sviluppi futuri

Del software trattato ne è stata creata solo una prima release che implementa un sottoinsieme delle funzionalità di tutto il sistema. Si ipotizza di poter sviluppare in futuro le altre funzionalità relative all'utente vigile ed all'utente funzionario. Inoltre per il completamento del lato automobilista si prevede di implementare la funzionalità di pagamento di una multa.