UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

Elaborato PSSS (Progettazione e Sviluppo Sistemi Software)

di

Lubrano Vincenzo

Piccolo Andrea

Renzi Gianluigi

Tramontin Dario

FEDERICO II

Laurea Magistrale Ingegneria informatica

23 luglio 2018

Indice

Capitolo 1

Introduzione

1.1 Requisiti funzionali

CPS (Connected Personalized Settings) è un'applicazione per dispositivi mobili con sistema operativo Android, sviluppata in collaborazione col gruppo FCA. Essa dovrà permettere all'utente di registrarsi e di selezionare le auto appartenenti al gruppo FCA in suo possesso. Per ciascuna di esse, l'utente potrà settare una o più configurazioni personalizzate inserendo le opportune preferenze (ad esempio l'inclinazione degli specchietti laterali o dello specchietto retrovisore, la posizione del sediolino del guidatore, l'altezza del manubrio, ecc).
Le diverse configurazioni, facilmente trasferibili tra le auto registrate, vengono salvate in Cloud.

1.2 Processo di sviluppo

Per lo sviluppo di CPS si è inizialmente deciso di utilizzare un approccio di tipo Agile, con metodologia Scrum. Inizialmente si è tenuto un incontro tra il team di sviluppo e lo stakeholder per il rilascio dei requisiti funzionali. Successivamente, il team di sviluppo si è riunito più volte in una fase iniziale di pianificazione, in cui sono stati prefissati i principali obiettivi e si è iniziato a pensare ad una possibile architettura su cui costruire il sistema software.

Dopodiché è stato generato un primo Product Backlog, contenente tutte le caratteristiche da sviluppare e le loro priorità (figura: 1.1)
Si è quindi scelto di estrarre dal Product Backlog la funzionalità più complessa che permette ad un utente di associare una configurazione alla propria auto. Quindi, ciascun membro del team di sviluppo ha cominciato ad analizzare il problema e pensare ad una possibile soluzione. Tutte le proposte sono state ampiamente discusse in Daily Scrums, durante i quali sono stati abbozzati dei primi diagrammi dei casi d'uso e diagrammi delle classi, in modo da visualizzare quali fossero le principali entità coinvolte nel sistema e le loro interazioni.
Per motivi esterni non è stato più possibile effettuare riunioni giornaliere, per cui si è continuato lo sviluppo separatamente, restando comunque in contatto ogni giorno tramite strumenti come Skype e Telegram, aggiornando incrementalmente la pianificazione utilizzando lo strumento agile Trello ed effettuando riunioni settimanali. Durante tutto il processo di sviluppo, man mano che sono stati prodotti nuovi artefatti che coinvolgevano determinate scelte di progettazione, si è rimasti in stretto contatto con lo stakeholder, così da ottenere continui feedback e apportare modifiche incrementali, evitando la propagazione di eventuali errori in fasi più avanzate dello sviluppo.

ID	Priority	Type	Item	SP	Status
1	1	User Story	Come utente, voglio creare una configurazione personalizzata	3	Not started
2	1	User Story	Come utente, voglio registrare un'auto	2	Not started
3	6	Epic	Come utente, voglio associare una configurazione ad un'auto	4	Not started
4	5	User Story	Come utente, voglio creare un account	2	Not started
5	4	User Story	Come utente, voglio visualizzare le auto associate al mio account	2	Not Started
6	4	User Story	Come utente, voglio visualizzare le configurazioni associate al mio account	2	Not Started

Figura 1.1

Capitolo 2

Analisi

2.1 Analisi Testuale

In una prima fase di comprensione del dominio, è stata effettuata l'analisi testuale dei requisiti (informali) forniti dallo stakeholder. In particolare sono state estrapolate le principali entità in gioco, cercando di definire come esse interagiscano col sistema e quali funzionalità vengano messe a disposizione. In particolare sono stati individuati gli attori principali, che in questo caso sono Utente ed Auto, e i principali servizi offerti dal sistema per far sì che gli attori possano raggiungere determinati obiettivi.
In 2.1 l'analisi testuale effettuata in Visual Paradigm.

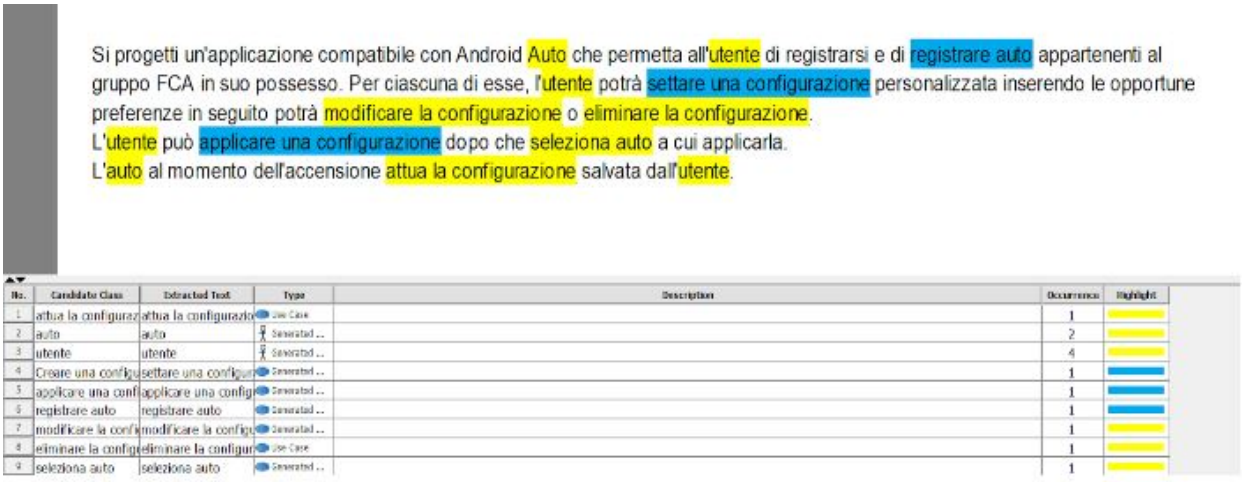

Figura 2.1

2.2 Diagramma dei casi d'uso

A partire dall'analisi testuale, si è pensato di generare un diagramma dei casi d'uso di analisi (figura 2.2), in modo da definire un confine tra il sistema (e tutto ciò che offre) e il mondo esterno con cui esso interagisce. In particolare, si è deciso di "splittare" il generico attore Utente in due attori: UtenteNonRegistrato, il cui unico caso d'uso è Registrati, e Utente, che invece rappresenta un utente già registrato e che quindi può usufruire di tutte le funzionalità messe a disposizione dal sistema. Premesso che qualunque operazione l'utente voglia effettuare deve prima loggare con le proprie credenziali di registrazione, egli avrà la possibilità di inserire ed eliminare auto, gestire le proprie configurazioni ed associare una data configurazione ad una qualunque tra le auto registrate. Invece, l'attore Auto potrà interagire col sistema tramite il caso d'uso AttuaConfigurazione, che gli permetterà di prelevare la configurazione settata dall'utente ed attuarla sui propri componenti.

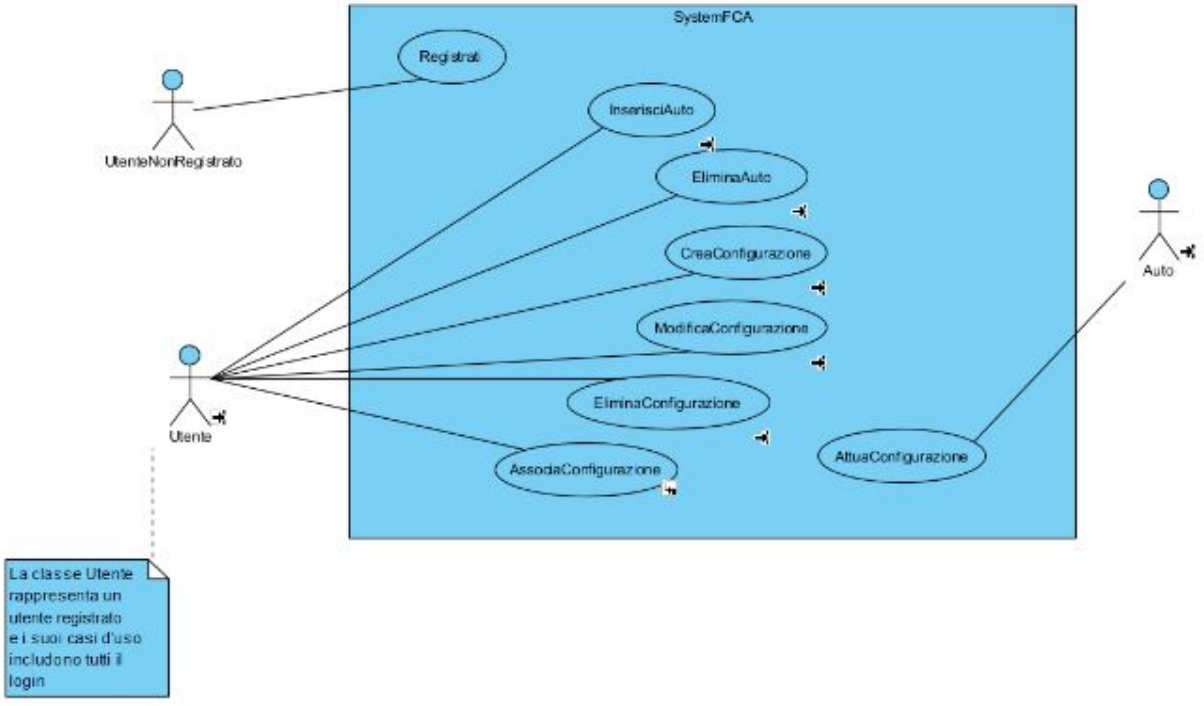

Figura 2.2

2.3 Descrizione dei casi d'uso

Di seguito, vengono riportate delle brevi descrizioni dei singoli casi d'uso, evidenziando gli attori coinvolti e il loro impatto sul sistema.

3. CreaConfigurazione
ID: UC04

Primary Actors	Utente
Level	Obiettivo utente
Complexity	Low
Use Case Status	Initial
Implementation Status	Scheduled
Preconditions	L'utente è registrato e loggato all'interno del sistema
Post-conditions	Una nuova configurazione viene inserita all'interno del sistema ed associata ad un utente
Author	N/A
Assumptions	N/A

Inserisci Configurazione

1. InserisciAuto
ID: UC05

Primary Actors	Utente
Level	Obiettivo utente
Complexity	Low
Use Case Status	Initial
Implementation Status	scheduled
Preconditions	L'utente è registrato e loggato all'interno del sistema
Post-conditions	Una nuova auto è registrata all'interno del sistema ed associata ad un utente
Author	/
Assumptions	N/A

Inserisci auto

2. EliminaAuto
ID: UC02

Primary Actors	Utente
Level	Obiettivo utente
Complexity	Low
Use Case Status	Initial
Implementation Status	Not Scheduled
Preconditions	L'utente è registrato e loggato all'interno del sistema InserisciAuto
Post-conditions	L'auto selezionata è eliminata dal sistema
Author	N/A
Assumptions	N/A

Elimina auto

5. EliminaConfigurazione
ID: UC01

Primary Actors	Utente
Level	Obiettivo utente
Complexity	Basic
Use Case Status	Initial
Implementation Status	Not scheduled
Preconditions	L'utente è registrato e loggato all'interno del sistema CreaConfigurazione
Post-conditions	La configurazione selezionata è eliminata dal sistema
Author	/
Assumptions	N/A

Elimina configurazione

4. ModificaConfigurazione
ID: UC06

Primary Actors	Utente
Level	Obiettivo utente
Complexity	Low
Use Case Status	Initial
Implementation Status	Scheduled
Preconditions	L'utente è registrato e loggato all'interno del sistema CreaConfigurazione
Post-conditions	La configurazione selezionata è stata aggiornata nel sistema
Author	N/A
Assumptions	N/A

Modifica configurazione

6. AssociaConfigurazione
ID: UC08

Primary Actors	Utente
Level	Obiettivo utente
Complexity	High
Use Case Status	Initial
Implementation Status	Not scheduled
Preconditions	L'utente è registrato e loggato all'interno del sistema CreaConfigurazioneInserisciAuto
Post-conditions	La configurazione scelta dall'utente è filtrata in un settaggio apposito per l'auto in questione. Il settaggio è salvato e pronto ad essere attuato dall'auto
Author	/
Assumptions	N/A

Associa configurazione

2.4 Context diagram

Per la comprensione dell'ambiente con cui il sistema interagisce e delle entità in gioco è stato realizzato un context diagram che evidenzia quali device interagiscono con il sistema FCA e con quale molteplicità (figura: 2.3).

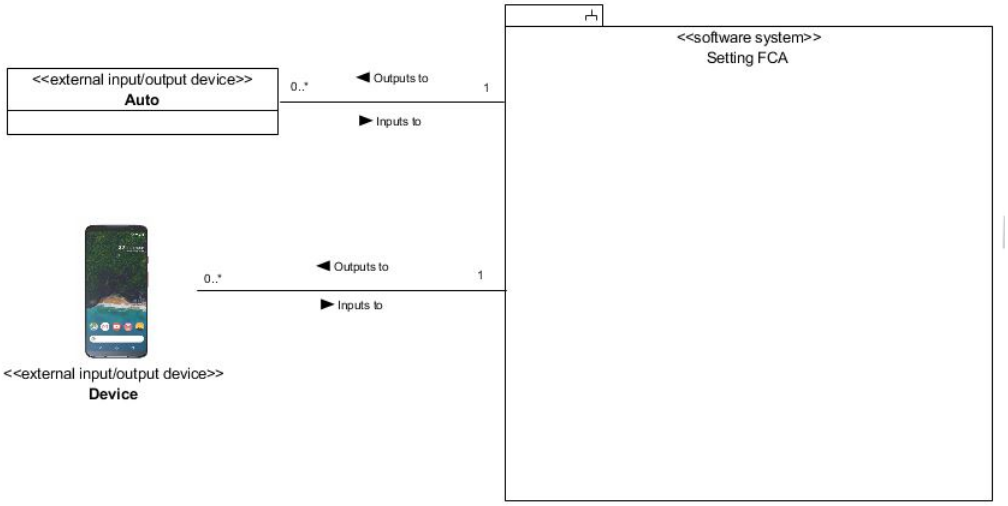

FIGURA 2.3: Caption

Prima di analizzare il componente client android, analizzeremo prima la struttura del sistema Setting FCA da una prospettiva lato server.

2.5 Diagramma delle classi

Per meglio comprendere le entità in gioco, è stato definito un primo diagramma delle classi in figura 2.4.

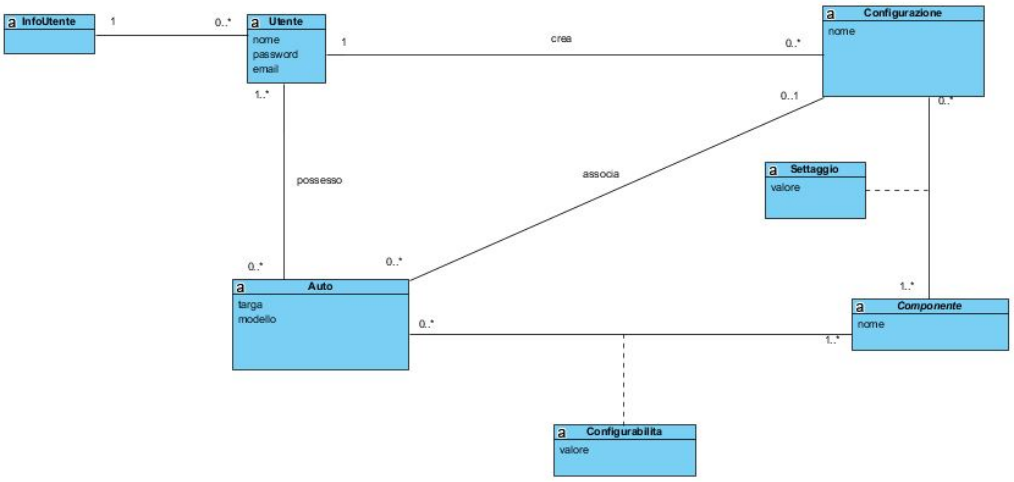

FIGURA 2.4

Le classi Utente, Auto e Configurazione derivano direttamente dai requisiti forniti dal cliente. In aggiunta, si è ritenuto doveroso aggiungere la classe InfoUtente, Information Expert della classe Utente, in modo da avere un oggetto a cui poter assegnare tutte le responsabilità che consentono di ottenere informazioni sugli utenti. Per quanto riguarda la classe Componente, si è rivelata necessaria per poter risolvere il problema relativo alle configurazioni associate alle varie auto. Di fatti, ciascuna auto appartenente al gruppo FCA può avere diversi componenti da poter configurare; per cui non è possibile definire un'unica configurazione universale applicabile su tutte le auto, ma bisogna effettuare un'operazione di filtraggio in modo che un utente possa salvare una o più configurazioni associate al proprio account, che valgano per tutte le possibili auto in suo possesso. Per questo motivo, l'utente ha la possibilità di configurare dei settaggi "completi" sulla base delle proprie esigenze, mentre spetterà al sistema il compito di filtrare le varie configurazioni considerando solo i componenti di interesse per l'auto scelta ed adattarle alle singole auto. A tal proposito, sono state inserite le classi associative Configurabilita, contenente

il valore di ciascun componente di un'auto (se configurabile) e Settaggio, contenente il valore scelto dall'utente per un componente, relativo ad una specifica configurazione.

2.6 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.

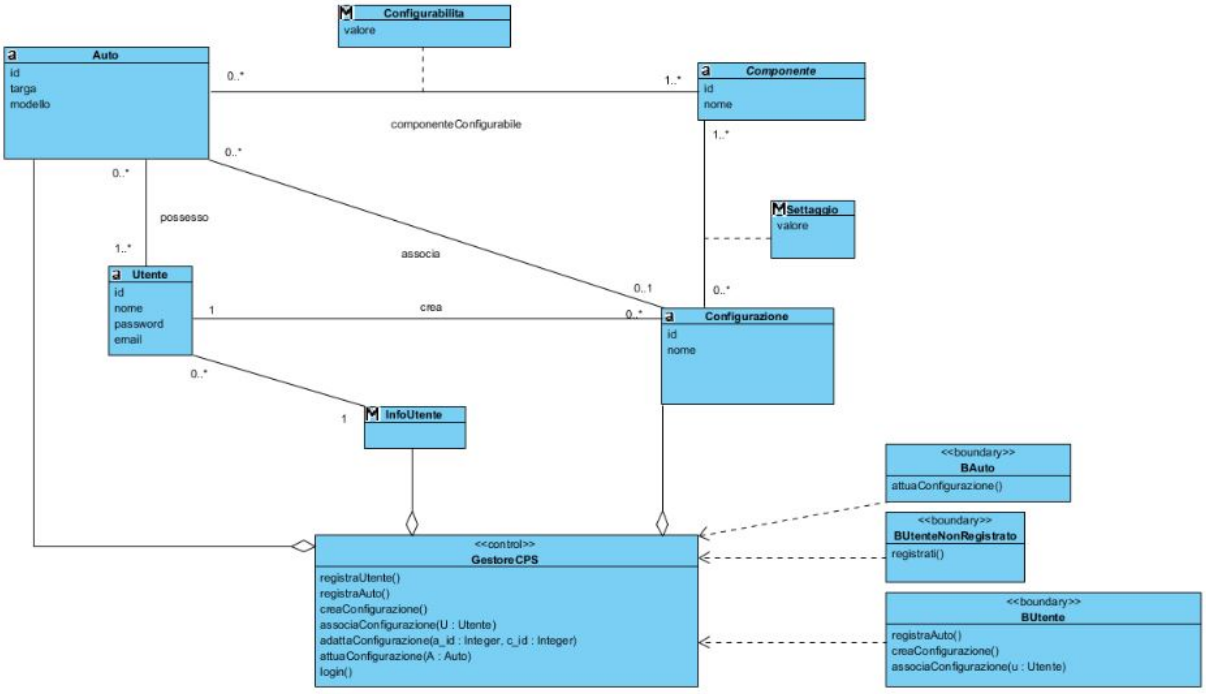

Figura 2.5

2.7 Architettura software

L'architettura su cui è stato costruito il sistema comprende tre figure fondamentali: una, che rappresenta il "gestore" di tutto il sistema, che mette a disposizione i vari servizi; un'altra, che invece rappresenta gli utenti e le auto, che per poter raggiungere determinati obiettivi ha bisogno di utilizzare i servizi offerti dalla prima e infine una terza figura, che si occupa della persistenza e della gestione dei dati necessari per il funzionamento del sistema. Per questo motivo si è pensato di adottare uno stile architetturale *Client/Server* di tipo *three-tiered*, costituito rispettivamente da un Application Server, responsabile della logica funzionale di business, da due tipi di Client,Auto in cui è gestita la presentazione (interfaccia), e infine da un Database Server che contiene la logica di gestione dei dati.

2.8 Diagramma dei componenti

Tale stile prevede l'utilizzo di due tipi di *componenti*, che sono appunto i Client e il Server. In particolare, i Client, rappresentati da Utente ed Auto, utilizzano un'interfaccia per avere accesso al Server ed invocarne i servizi da remoto (quindi i Client devono conoscere l'identità del Server, ma non vale il viceversa). I *connettori* invece consistono in Remote Procedure Calls (RPC) utilizzate dai Client per accedere alle funzionalità messe a disposizione dal 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. È possibile osservare tali relazioni nel Component Diagram in figura 2.6.

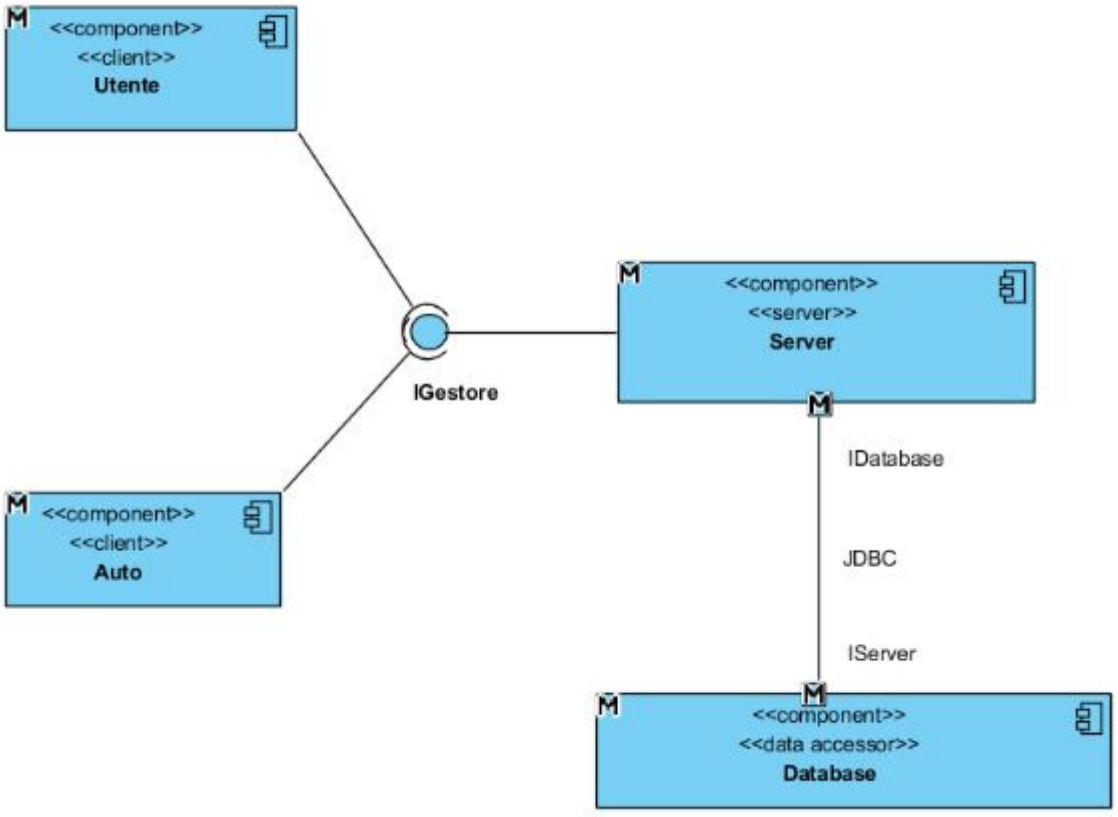

Figura 2.6

2.9 Caso d'uso "Associa configurazione"

In questa prima release dell'applicazione si è scelto di implementare il caso d'uso AssociaConfigurazione. Esso rappresenta la funzionalità offerta dal sistema che permette all'utente di associare una propria configurazione ad una delle auto del gruppo FCA in suo possesso che sono state inserite nel sistema precedentemente. In una prima analisi del caso d'uso si è trascritto un possibile *scenario*, in cui vengono esplicitate le varie interazioni tra l'utente e il sistema che permettono la realizzazione della funzionalità (figura: 2.7).

In particolare, l'utente avvia il caso d'uso, il sistema risponde mostrerà una lista delle configurazioni e delle auto ad egli associate, da cui l'utente deve selezionare quelle di interesse per poter completare il caso d'uso. Il sistema genera un "ack" per notificare l'utente dell'avvenuta associazione.

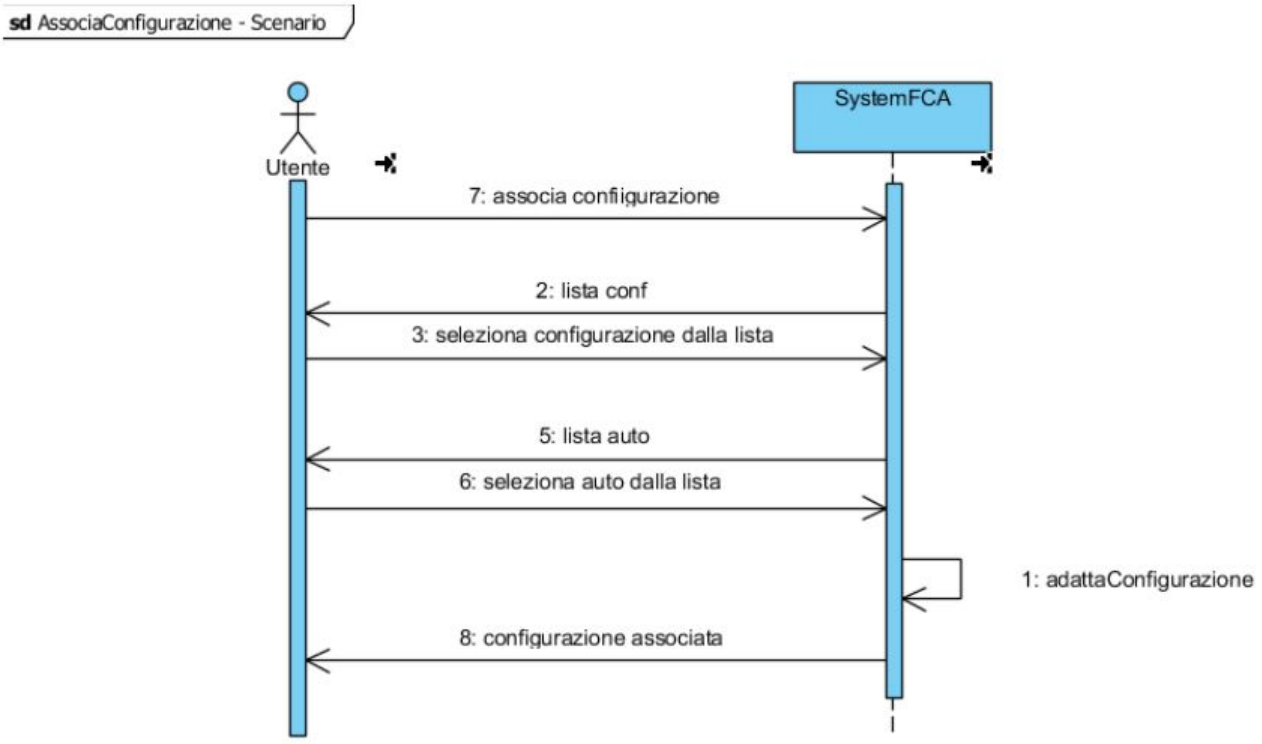

Figura 2.7

2.10 Diagramma di sequenza

A partire da questa base, è stato generato un Sequence Diagram di analisi, che dispiega tali interazioni attraverso scambio di messaggi tra le varie entità in gioco. In particolare, l'attore Utente avvierà il caso d'uso associaConfigurazione tramite la rispettiva classe Boundary, la quale andrà ad inoltrare la richiesta al Gestore. Questo sarà responsabile di reperire le liste delle configurazioni e delle auto associate all'utente, in modo da permettere a quest'ultimo di selezionare le proprie preferenze. Una volta ottenute l'auto e la configurazione di interesse, il Gestore eseguirà l'operazione di filtraggio sui componenti configurabili, adattando la configurazione scelta all'auto selezionata.
Si rimanda al progetto Visual Paradigm per la visione.

Capitolo 3

Progettazione

3.1 Vista architetturale

Come primo step per la fase di progettazione, si è reputato necessario creare una vista architetturale che esplicitasse le scelte effettuate durante la prima fase di analisi. É stata, dunque, creata la vista architetturale in figura 3.1.
In particolare, vengono evidenziati i due Client (User e Auto) e la loro comunicazione col Server. Dal lato Server si è scelto di utilizzare un pattern façade per l'implementazione di Gestore_CPS, in modo da unificare le interfacce di più basso livello Gestore_Auto, Gestore_Utente e Gestore_Conf fornendo un unico punto di accesso e aumentando il disaccoppiamento tra i Client e il sistema. Per quanto riguarda invece la gestione e la memorizzazione dei dati, tali responsabilità sono delegate alle classi DAO, che si occupano della comunicazione col database.

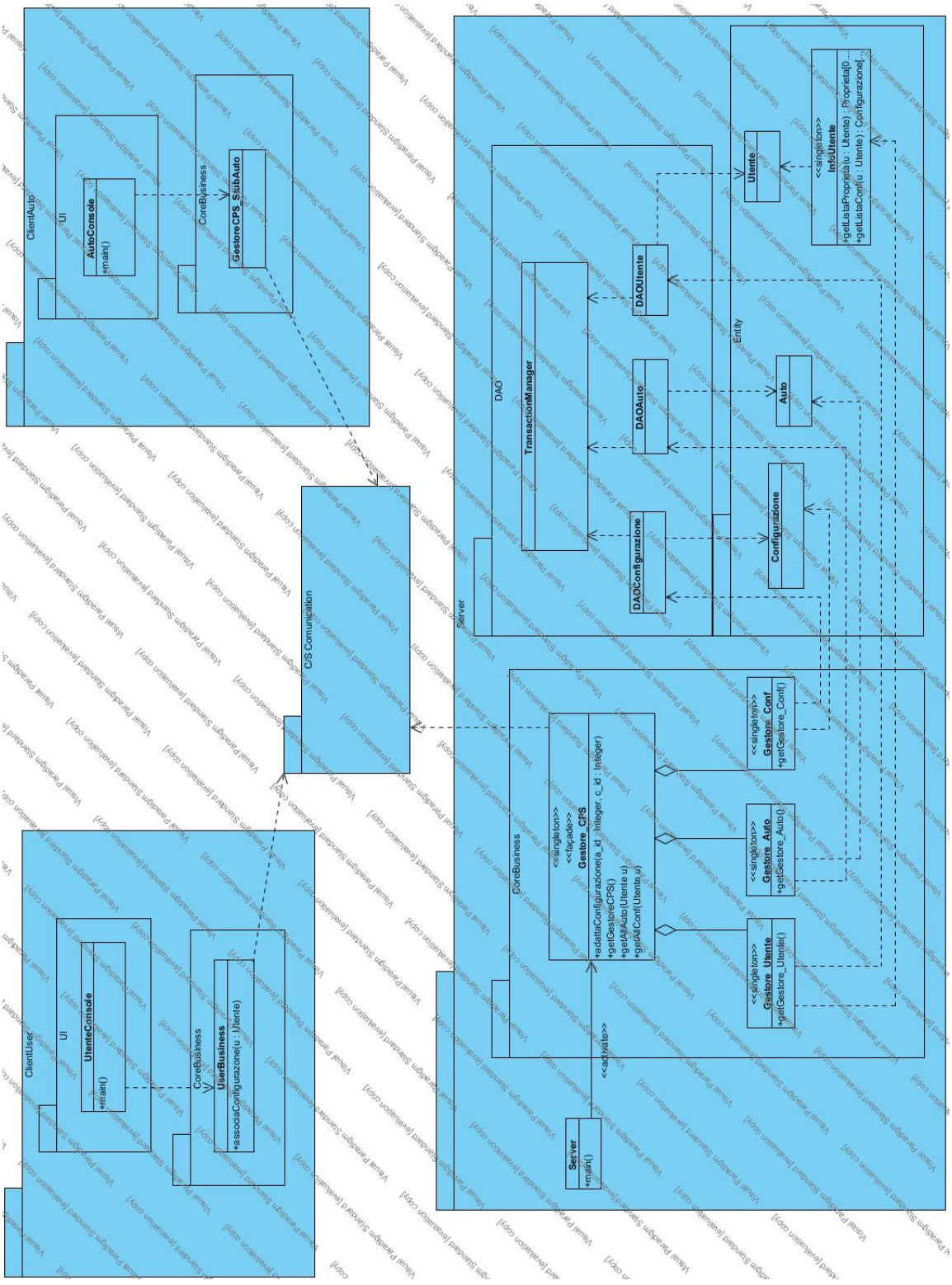
ClientAuto
UI
AutoConsole
+main()
CoreBusiness
GestoreCPS_StubAuto
C/S Comunication
ClientUser
UI
UtenteConsole
+main()
CoreBusiness
UserBusiness
+associaConfigurazione(u : Utente)
Server
DAO
TransactionManager
DAOUtente
DAOAuto
DAOConfigurazione
Entity
Utente
Auto
Configurazione
<<singleton>>
InfoUtente
+getListaPropria(u : Utente) : Proprietà[0....
+getListaConf(u : Utente) : Configurazione[...
CoreBusiness
<<singleton>>
<<facade>>
Gestore_CPS
+adattaConfigurazione(a_id : Integer, c_id : Integer)
+getGestoreCPS()
+getAllAuto(Utente u)
+getAllConf(Utente u)
<<singleton>>
Gestore_Conf
+getGestore_Conf()
<<singleton>>
Gestore_Auto
+getGestore_Auto()
<<singleton>>
Gestore_Utente
+getGestore_Utente()
<<activate>>
Server
+main()

Figura 3.1

3.2 Proxy-Skeleton

Al fine di permettere la comunicazione da remoto tra Client e Server è stato utilizzato il pattern Proxy, il quale fornisce una rappresentazione locale dell'oggetto remoto con cui si vuole interagire. Infatti, analizzando nel dettaglio il package C/S Comunication (figura:3.2), è possibile osservare come il Client interagisca col GestoreCPS_Proxy (rappresentazione in locale di Gestore_CPS), il quale andrà ad utilizzare l'interfaccia IGestoreCPS per richiamare da remoto i metodi offerti da Gestore_CPS. Anche dal lato Server, viene utilizzata la classe GestoreCPS_Skeleton, in modo da disaccoppiare la logica relativa alla comunicazione da quella relativa all'implementazione delle funzionalità.

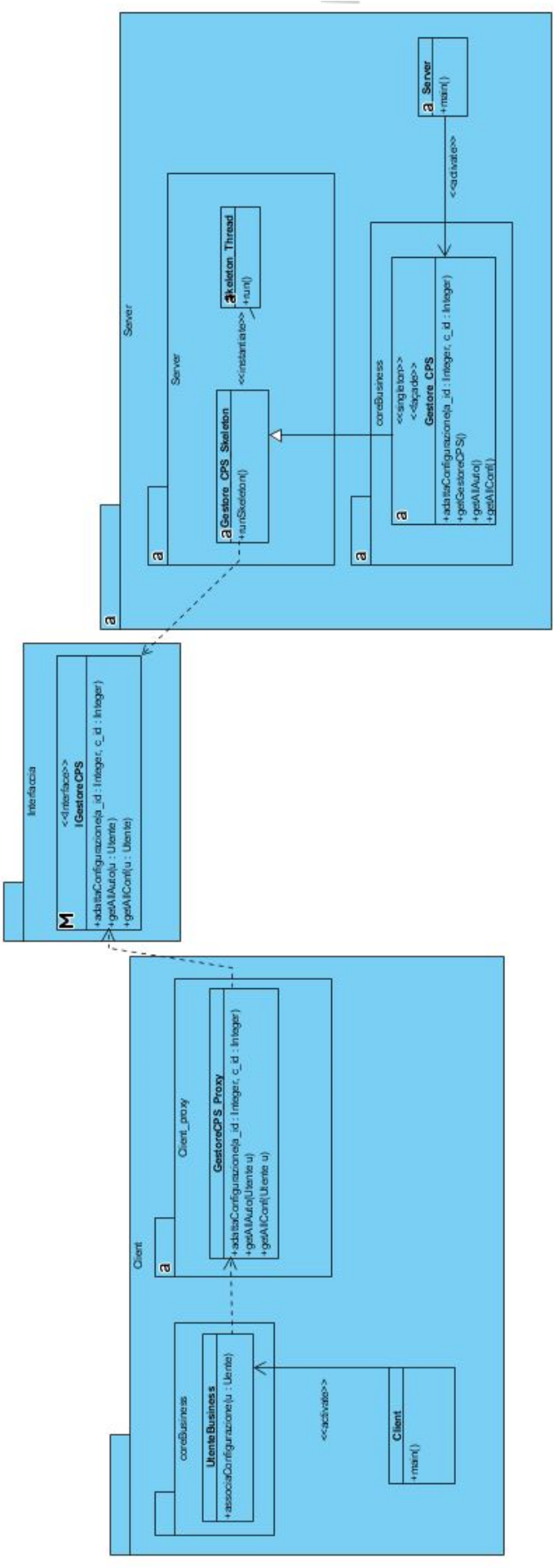
Server
Server
Skeleton_Thread
+run()
<<instantiate>>
Gestore_CPS_Skeleton
+runSkeleton()
coreBusiness
<<singleton>>
<<façade>>
Gestore_CPS
+aggiungiConfigurazione(a_id : Integer, c_id : Integer)
+getGestoreCPS()
+getAllAuto()
+getAllConf()
Server
+main()
<<activate>>
Interfaccia
<<interface>>
IGestoreCPS
+aggiungiConfigurazione(a_id : Integer, c_id : Integer)
+getAllAuto(u : Utente)
+getAllConf(u : Utente)
Client
Client_proxy
GestoreCPS_Proxy
+aggiungiConfigurazione(a_id : Integer, c_id : Integer)
+getAllAuto(Utente u)
+getAllConf(Utente u)
coreBusiness
UtenteBusiness
+associaConfigurazione(u : Utente)
<<activate>>
Client
+main()

Figura 3.2

3.3 Vista BCE

Successivamente è stato creato un diagramma BCE (Boundary, Control, Entity) reputato molto utile per la descrizione della logica del prodotto software, separando le responsabilità degli elementi del sistema. Tale diagramma rappresenta un raffinamento del class diagram di analisi; infatti, sono state esplicitate tutte le relazioni tra le classi in gioco. In particolar modo sono state raffinate le entity e tradotte le associazioni risolvendo tutte le molteplicità e definendo tutte le relazioni di aggregazione e composizione, specificandone la navigabilità. Sono stati inoltre aggiunti tutti gli attributi e i metodi necessari per ogni classe, coi relativi tipi e visibilità. **Si rimanda al progetto Visual Paradigm per la visione del diagramma**.

3.4 Sequence diagram Associa configurazione

La fase successiva di progettazione ha visto la stesura di un sequence diagram che seguisse le linee guida del corrispettivo sequence di analisi del caso d'uso Associa Configurazione. Anche qui, essendo il diagramma molto articolato, **si rimanda al progetto Visual Paradigm**.

3.5 Sequence diagram client side (Associa Configurazione

È stato invece ritenuto interessante evidenziare la parte relativa alla comunicazione lato client (3.3).

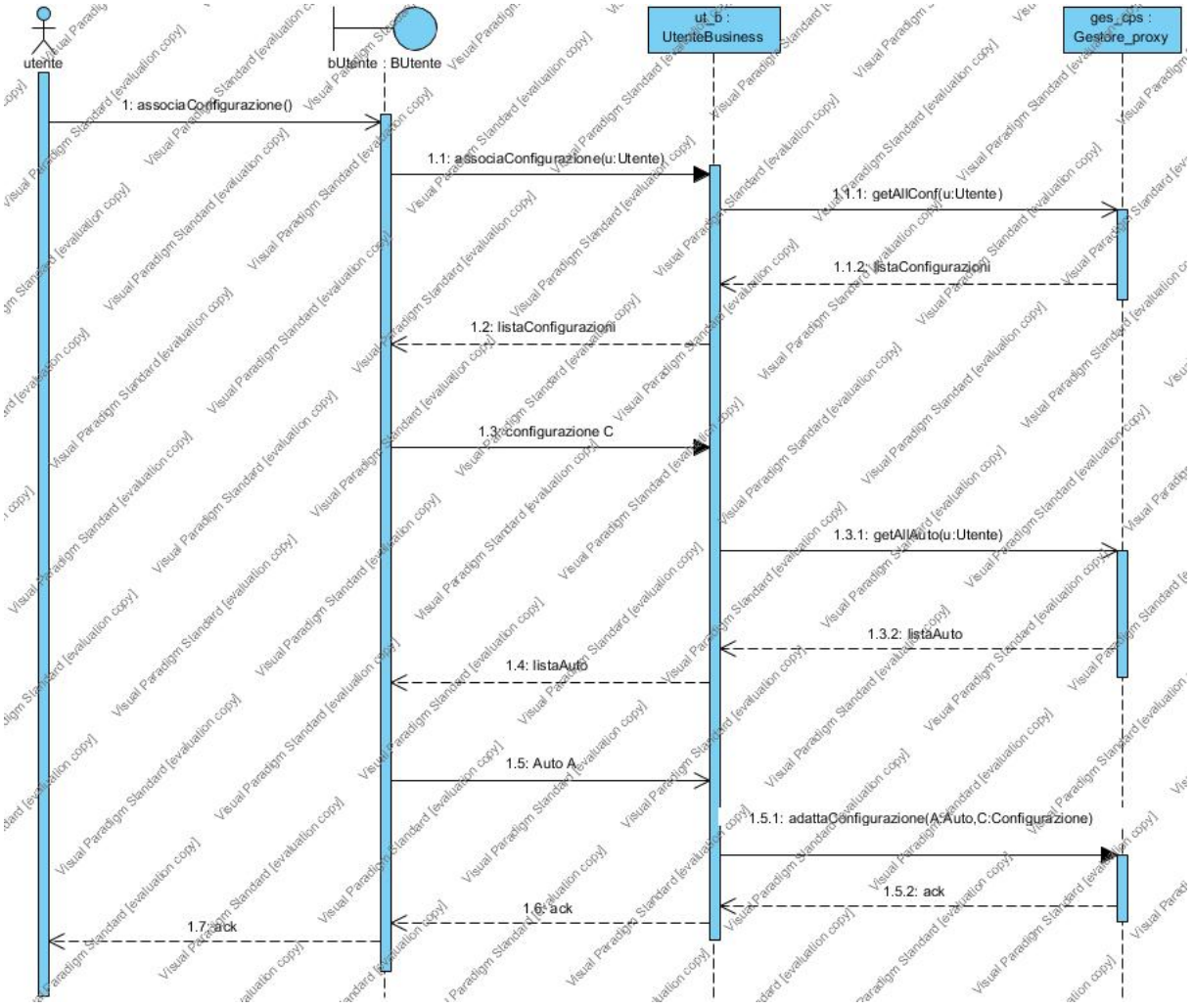

Figura 3.3

3.6 Deployment diagram

Per fornire una vista di come gli Artifacts (gli elementi risultanti dal processo di sviluppo) vengono distribuiti sui Nodes, ossia hardware o ambienti di sviluppo software, è stato sviluppato un diagramma di deployment (figura: 3.4).

In questo specifico caso è possibile osservare come gli artefatti utente.jar e auto.jar siano distribuiti su due nodi, che consistono nei rispettivi dispositivi "Terminale utente" (dispositivo Android) e "Terminale auto" (dispositivo con Android Auto). Mentre l'artefatto server.jar è distribuito su un terzo terminale che comunica con il database mySQL. La comunicazione avviene su rete attraverso l'utilizzo del protocollo TCP/IP.

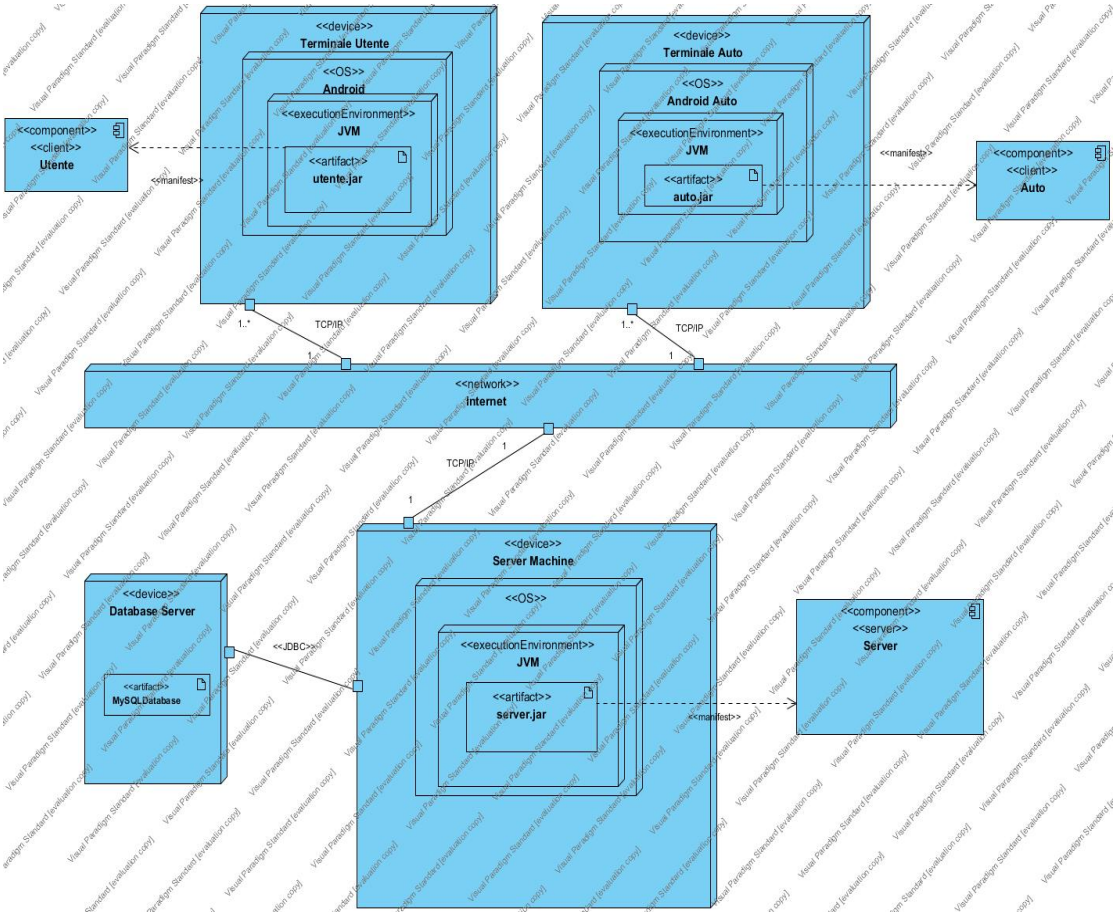

Figura 3.4

3.7 Viste client Android

Per la progettazione Android sono state apportate modifiche considerevoli al lato client dell'applicazione, dunque si è reso necessario costruire una prima vista della struttura logica interna al client (figura 3.5).

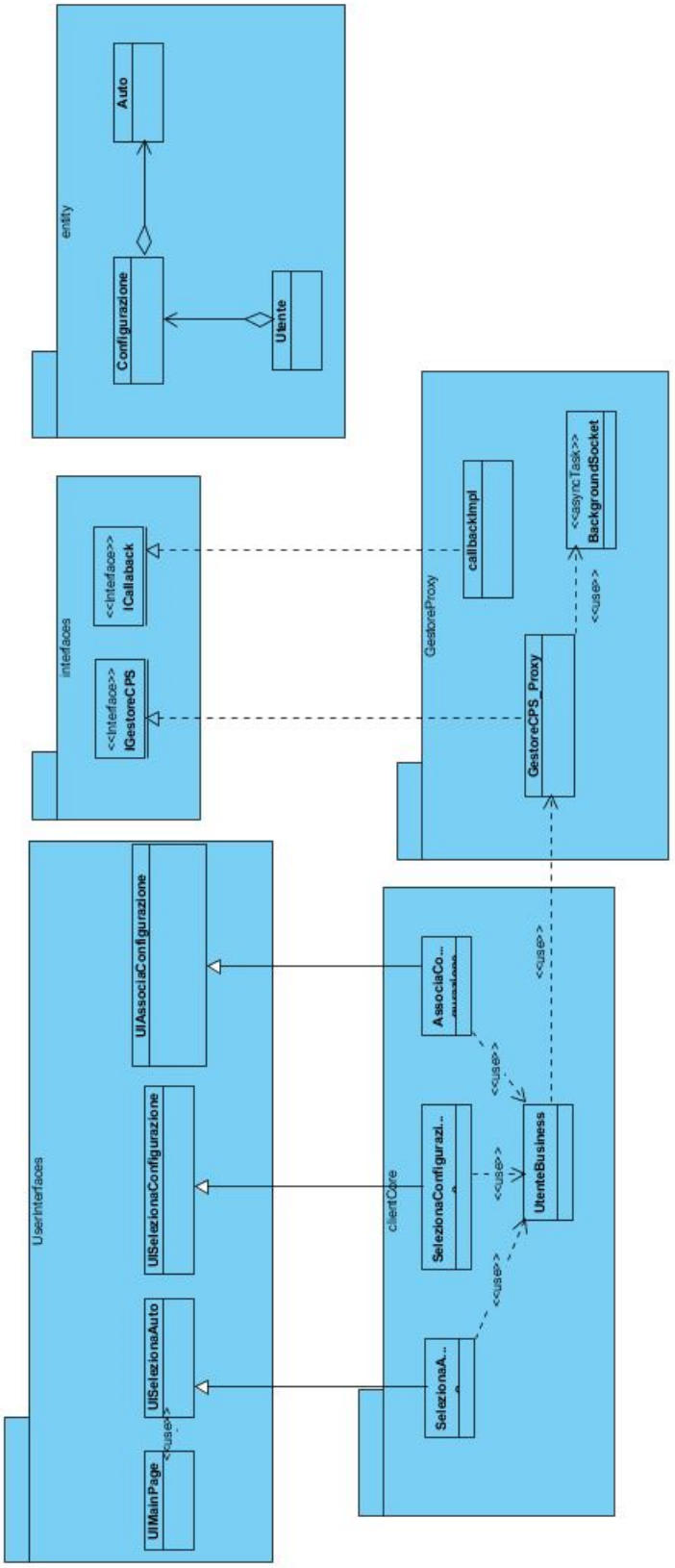
entity
Auto
Configurazione
Utente
interfaces
<<Interfaces>>
ICallaback
<<Interfaces>>
IGestoreCPS
GestoreProxy
callbackImpl
<<asyncTask>>
BackgroundSocket
<<use>>
GestoreCPS_Proxy
UserInterfaces
UIAssociaConfigurazione
UISelezionaConfigurazione
UISelezionaAuto
UIMainPage
<<use>>
clientCore
AssociaCo...
SelezionaConfigurazi...
SelezionaA...
UtenteBusiness
<<use>>

Figura 3.5

3.8 Proxy-Skeleton

Al fine di descrivere la comunicazione Client-Server si è deciso di creare un class diagram che evidenzi il ruolo del pattern Proxy per la comunicazione da remoto.
Si noti principalmente come, a differenza del corrispettivo diagramma per l'applicazione java, il Client utilizzi molteplici activy racchiuse nel package "userInterfaces". Inoltre, il gestore_Proxy non comunica direttamente con il lato server ma usufruisce di una Task asincrona di nome "backgroundSocket" che ha il ruolo di instaurare una connessione socket in background per favorire la comunicazione asincrona. **Per la visione si rimanda al progetto Visual Paradigm.**

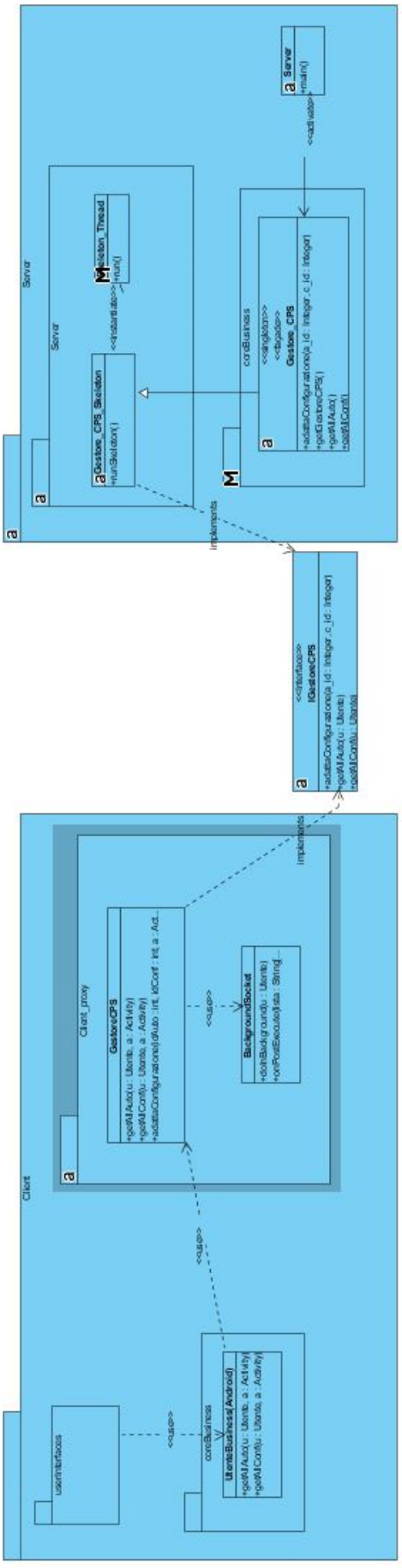
Server
Server
Server
main()
<<activate>>
Sessione_Thread
run()
<<instantiate>>
Gestore_CPS_Skeleton
runSkeleton()
coreBusiness
<<singleton>>
<<bound>>
Gestore_CPS
aggiornaConfigurazione(a_id : Integer, c_id : Integer)
getStatoInCPS()
getAllAuto()
getAllConfig()
implements
<<interface>>
IGestoreCPS
aggiornaConfigurazione(a_id : Integer, c_id : Integer)
getAllAuto(u : Utente)
getAllConfig(u : Utente)
implements
Client_proxy
GestoreCPS
getAllAuto(u : Utente, a : Activity)
getAllConfig(u : Utente, a : Activity)
aggiornaConfigurazione(idAuto : int, idConf : int, a : Act...
<<use>>
BackgroundSocket
doInBackground(u : Utente)
onPostExecute(lista : String[...
Client
<<use>>
coreBusiness
UtenteBusiness(Android)
getAllAuto(u : Utente, a : Activity)
getAllConfig(u : Utente, a : Activity)
<<use>>
userInterfaces

FIGURA 3.6

3.9 Communication diagram

Per spiegare a pieno il funzionamento dell'applicazione lato client si è reso necessario costruire un communication diagram che espliciti le varie azioni che avvengono per portare a termine il caso d'uso AssociaConfigurazione.
Il diagramma è utile per specificare come l'utente interagisce con le varie interfacce e quali sono le azioni che conseguono agli input dell'utente per ognuna delle interfacce grafiche.
Per prendere visione, si rimanda al progetto Visual Paradigm.

3.10 Sequence diagram lato client (Associa Configurazione)

Per esplicitare la sequenza completa di azioni che avvengono lato cliente è stato costruito un apposito diagramma di sequenza. Quest'ultimo diagramma spiega al meglio i vari passaggi che occorrono per associare una configurazione ad una specifica auto.
Essendo il sequence composto da un numero elevato di passaggi non è stato possibile riportarlo all'interno della documentazione, pertanto **si rimanda al progetto Visual Paradigm per prenderne visione**.