ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

INGEGNERIA DEI SISTEMI SOFTWARE M

USO DI STRUMENTI WEB PER LA REINGEGNERIZZAZIONE PLATFORM INDEPENDENT DI APPLICAZIONI MULTIPIATTAFORMA

CANDIDATO:
Simone Berardi

RELATORE:
Chiar.mo Prof. Antonio Natali

Anno Accademico: 2012/13

Sessione III

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Il mondo delle aziende software è sempre più legato ad una pletora di tecnologie in costante evoluzione. I cambiamenti portati da questo continuo movimento costringe molte aziende a svolgere un pressante lavoro di adeguamento per rimanere competitive sul mercato.

Il lavoro svolto esegue una breve analisi su questo scenario e cerca di porvi rimedio con diversi approcci funzionali e progettuali. In particolare si pone l'obiettivo di ampliare le capacità di comunicazione di software aziendali di modo che possano interfacciarsi anche con l'ambiente web.

Applicativi, anche molto complessi, scritti in linguaggio specifico e fortemente dipendenti da piattaforme operative tendono ad essere inerti a tentativi di cambio di tecnologia. Per evitare i costi ed i tempi di riscrittura integrale di un tale software, è opportuno studiare una soluzione capace di mantenere valido gran parte dell'elaborato. Di particolare interesse è il mantenimento inalterato della *business logic* e della fase di progettazione già svolta.

Il processo di analisi di fattibilità svolto è stato diviso in macro parti di interesse con l'obiettivo di esaminare nei dettagli tutte le possibilità di ammodernamento e tutti i relativi vantaggi e svantaggi. Partendo da una analisi preliminare sui requisiti imposti ad una simile operazione si volgerà velocemente verso il vaglio di differenti soluzioni fino a sviluppare ovvie domande verso possibili sviluppi futuri.

Nella prima parte è presente una breve indagine su software di terze parti, capaci di offrire una condivisione in remoto di applicativi fortemente accoppiati con il sistema operativo sottostante. Nella seconda parte si pone invece l'attenzione sull'analisi e la progettazione di una soluzione software generica, in grado di garantire una complementarità fra applicativo aziendale e nuovi requisiti di comunicazione web. Infine nella terza parte si lascia libero spazio alle riflessioni sui costi ed i tempi

di adeguamento di una tale soluzione a fronte di variazioni software oppure di tecnologia di output. In questa fase vengono anche esplorati approcci progettuali diversi adatti alla generazione automatica di codice.

Nel primo capitolo è presente una breve panoramica sullo scenario in cui si cala la nostra analisi.
Nel secondo capitolo viene descritto il tipo ed il paradigma di funzionamento del software aziendale di interesse.
Nel terzo capitolo è presente una discussione sugli obiettivi che un'azienda si pone durante l'adeguamento software di un applicativo verso l'ambiente web.
Nel quarto capitolo viene eseguita una cernita di tecnologie di terze parti adatte alla condivisione in remoto di interfacce legate a sistemi operativi specifici.
Il quinto capitolo è caratterizzato dall'analisi dettagliata del processo di sviluppo di un software in grado di adattare l'applicativo ad una comunicazione web.
Il sesto capitolo presenta delle considerazioni relative al lavoro svolto prendendone in considerazione le tempistiche e le modalità. Partendo da riflessioni sulla replicazione del procedimento vengono analizzati alcuni strumenti di metamodellazione. Tramite questi strumenti vengono estrapolati i concetti che fungono da linee guida alla progettazione di una tale soluzione. Il settimo capitolo infine contiene alcune riflessioni sul lavoro svolto e qualche spunto per sviluppi futuri.

Capitolo 1

Visione

FIGURA 1.1: Uno squarcio sulle tecnologie web.

In questo capitolo si vedrà quale sia lo scopo della trattazione esplorando problematiche crescenti nel mondo informatico e cercando di porvi rimedio. Partendo da un discorso generale si andrà sempre più nello specifico per inquadrare il campo di interesse all'interno del quale si svolge la trattazione.

Applicazioni e software si rivolgono sempre più ad un utilizzo da parte delle piattaforme più differenti dai luoghi più disparati del globo. Il concetto stesso di

applicativo viene via via separato dall'idea di un suo utilizzo centralizzato tramite strumenti come il desktop computer. Gli utenti vengono sempre più stimolati a richiedere applicazioni in grado di funzionare in maniera omogenea e trasparente sulla maggior parte dei sistemi hardware esistenti in commercio: dai desktop computer, ai laptop computer fino ad arrivare a tablet PC e smartphone per non parlare poi delle smart tv. Alcune richieste, grazie al forte supporto di una struttura ben sviluppata ed estesa come quella di Internet, possono essere evase nelle maniere più fantasiose sfruttando una pletora di tecnologie diverse.

Il problema più pressante per aziende nel settore dell'informatica da diversi anni è proprio quello di modernizzare i propri prodotti, già funzionanti in linguaggio ed ambiente specifico, per poterli rendere disponibili al pubblico tramite un accesso internet, raggiungendo un numero di dispositivi sempre maggiore.

Il desiderio di ogni progettista e programmatore che si trovi ad affrontare un grattacapo simile è quello di avere uno strumento in grado di generare un'implementazione funzionante del software già scritto, che sia accessibile online da qualunque piattaforma. In altri termini serve uno strumento capace di interfacciare una qualunque tecnologia esistente verso la piattaforma web; in questo modo è possibile interagire con qualunque dispositivo in grado di interpretare linguaggi web divenuti oramai "standard de facto" come html, javascript e css.

Grazie ad uno strumento del genere si potrebbe generare un'interfaccia in linguaggio specifico in grado di integrarsi con il software già presente in azienda. Allo stesso modo si disporrebbe dei mezzi per generare una replica web dell'interfaccia grafica di una applicazione tramite un semplice file di configurazione. Una volta istruita correttamente l'interfaccia, tutto il lavoro di traduzione tra business logic ed interfacciamento web risulterebbe completamente automatizzato.

Un generatore del genere potrebbe rivoluzionare il sistema di aggiornamento di vecchi software inizialmente non pensati per garantirne l'utilizzo online, ed ottenere facilmente un'estensione verso il mercato moderno multipiattaforma. In questo modo si può preservare il valore del "know how" aziendale garantendo la necessità di una minima conoscenza di nuove tecnologie web ed allo stesso tempo una massima estendibilità degli applicativi.

In conclusione molte aziende basate su buoni principi di progettazione possono trarre vantaggio dalla progettazione di una utility simile; garantendo infatti la bontà

dei loro progetti a livello di modello, e non solo a livello di codice, potranno estenderli senza la necessità di una reingegnerizzazione completa degli stessi.

Capitolo 2

Obiettivi

Questo capitolo preliminare analizza nei dettagli gli obiettivi imposti durante lo stage affrontato in azienda ponendo l'accento su quelle tematiche che risultano di importanza cruciale nel resto della trattazione.

Durante il lavoro svolto viene affrontato uno squarcio della problematica di distribuzione in formato web; una applicazione deve essere resa accessibile da parte di numerosi dispositivi partendo da una sua versione scritta in codice funzionante su piattaforma specifica.

In particolare si vuole garantire la capacità ad un software gestionale per ristoranti di operare anche online tramite web su dispositivi di tipo tablet. L'azienda ha già sviluppato una soluzione specifica per smartphone con costi medio - elevati in termini di tempo e denaro, dovendo riscrivere gran parte del codice sottostante ed in parte anche il modello del dominio.

Per ovviare a questi inconvenienti, in questo ed altri progetti, l'azienda è interessata a studiare la fattibilità di una remotizzazione dell'interfaccia grafica di questo strumento di gestione tramite un modello flessibile e replicabile. Ovviamente si deve ottenere il risultato rimanendo al passo con le più moderne ed affermate tecnologie web.

2.1 Vocabolario dei termini

Per questione di comodità vengono definiti di seguito i termini maggiormente utilizzati nella trattazione per eliminare ogni ambiguità ed evitare la ripetizione della loro specifica.

(Software) Gestionale: Applicativo software con finalità di assistenza alla gestione di parti sia burocratiche sia amministrative di numerose attività commerciali (hotel, beauty farm, ristoranti, ecc.).

Applicazione client - server: Applicativo software strutturato in base al design pattern client - server che implica la presenza di una parte client di interazione con l'utente in comunicazione con una parte server in grado di fornire dei servizi di diversa natura.

Capitolo 3

Requisiti

Nel seguente capitolo si viene a contatto per la prima volta con i requisiti imposti dagli obiettivi della trattazione. Si comincierà con una visione generale del problema per poi avviare un'analisi approfondita del software in questione e della tipologia di comunicazione fra le parti in gioco.

Il lavoro da svolgere in azienda prevede la modernizzazione di un sistema software progettato e implementato diversi anni fa con una tecnologia basata su framework .NET. Questa peculiarità di creazione rende l'applicazione utilizzabile solo tramite macchine con sistema operativo Windows. L'obiettivo è quello di rendere il software accessibile facilmente anche da tablet ed altri dispositivi, se possibile, senza dover scrivere nuovamente tutto il codice.

L'applicativo in questione è un software gestionale per ristoranti basato su di una architettura client - server, in comunicazione tramite un protocollo di rete socket. Le funzionalità di questo software comprendono, ma non sono limitate a, gestione delle prenotazioni, delle comande e dei fornitori, generazione di bolle e fatture, generazione e stampa di scontrini fiscali.

Spesso viene utilizzato per gestire strutture come ristoranti ed alberghi dando possibilità di lavorare, sia da un punto cassa tramite un computer fisso, sia da palmari in dotazione allo staff per poter effettuare comande oppure notificare modifiche al lavoro svolto e prenotazioni imminenti. I compiti del gestionale non si fermano qui ma per l'analisi in svolgimento sono sufficienti in quanto rendono l'idea della struttura sottostante necessaria per il corretto funzionamento.

Il gestionale deve essere reso disponibile in remoto secondo le specifiche relative al suo uso pratico. In particolare è importante che si possano effettuare numerose connessioni allo stesso gestionale, installato in un particolare ristorante o albergo, ed ognuna di queste connessioni deve essere autenticata da parte dell'utente in connessione remota. E' di vitale importanza infatti poter discriminare tra un amministratore di sistema ed un utente con minori privilegi. Lo scenario pone particolare importanza quindi alla molteplicità delle connessioni simultanee con diverse credenziali di accesso; ovviamente tutto il meccanismo deve permettere una buona scalabilità su quantità di alcune decine di connessioni remote e garantirne la perfetta concorrenza sulla stessa macchina server remota.

Il gestionale, come già accennato, è sviluppato secondo un pattern di tipo client - server e garantisce un buon parallelismo ed un'ottima scalabilità interna. Non è però capace in alcun modo di interfacciarsi tramite strumenti diversi da quelli sviluppati per sistema operativo Windows. La soluzione in analisi deve garantire la possibilità di accedere alla macchina in remoto anche tramite dispositivi di tipo tablet, garantendo una compatibilità ed una retrocompatibilità massima.

3.1 Software aziendale

Il software sviluppato in azienda, come già accennato, è un gestionale. Questo specifico tipo di software si pone come obiettivo l'assistenza di ambienti come ristoranti, beauty farm, hotel oppure piccole e medie attività commercianti. Il supporto viene garantito sotto forma di gestione di comande, prenotazioni e servizi in camera ma anche di gestione magazzino, vendite, bolle, fatture e tanti dettagli economici di un'attività commerciale tramite un unico strumento.

In particolar modo si lavorerà nel corso della tesi su di un gestionale appositamente studiato per l'assistenza a catene di ristoranti; questo dettaglio però non compromette la generalità della soluzione che si pone l'obiettivo di essere valida per una pletora di software simili.

Il sistema è diviso in due parti secondo un'architettura client - server ed entrambe le parti sono state implementate in C# per .NET. Mentre il server non presenta una parte grafica il client si presenta come in Fig. 3.1.

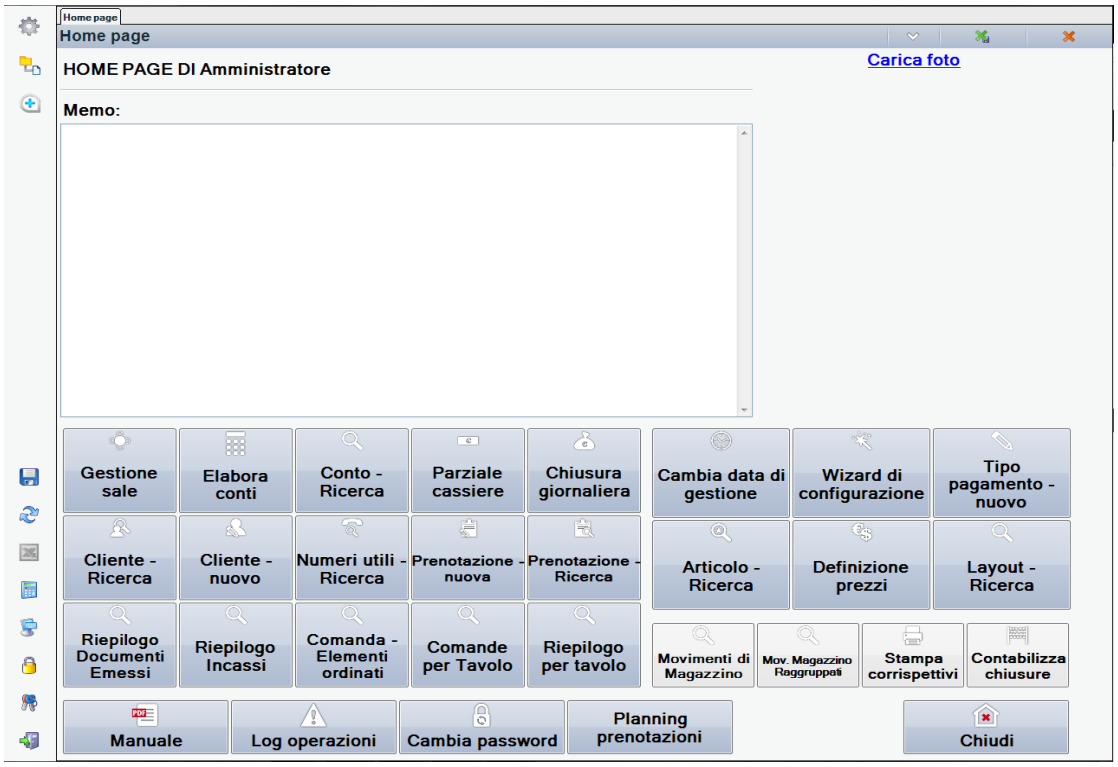

FIGURA 3.1: Pagina principale del software gestionale.

Scendendo più nei dettagli riguardo al funzionamento interno del software possiamo descriverne alcuni meccanismi di funzionamento. Il software possiede una strutturazione fortemente modulare per garantire la manutenibilità migliore possibile come prodotto aziendale. In particolare il progetto si appoggia su alcuni livelli di framework gestiti come progetti a se stanti. Questi forniscono gran parte dell'infrastruttura di comunicazione e funzionalità di base di tutte le applicazioni aziendali basate sugli stessi elementi.

Vediamo come la rappresentazione in Fig. 3.2 aiuti a carpire la complessità interna del codice in analisi per poter effettuare una reingegnerizzazione in un secondo momento. Da una parte è presente il framework responsabile della dichiarazione di tutti i dati e di tutte le operazioni di comunicazione all'interno dell'infrastruttura, dall'altra abbiamo il framework relativo alla definizione dei metadati per la gestione del modello.

Il primo garantisce un punto comune per quanto riguarda client e server dell'applicativo; infatti è gestito come un progetto integrato con entrambi i componenti. In questo modo risulta facile la gestione di operazioni comuni basate su tipi di dati proprietari. Il secondo invece fornisce un meta livello di gestione di dati del modello sottostante; descrive le interfacce sfruttate da parte del client

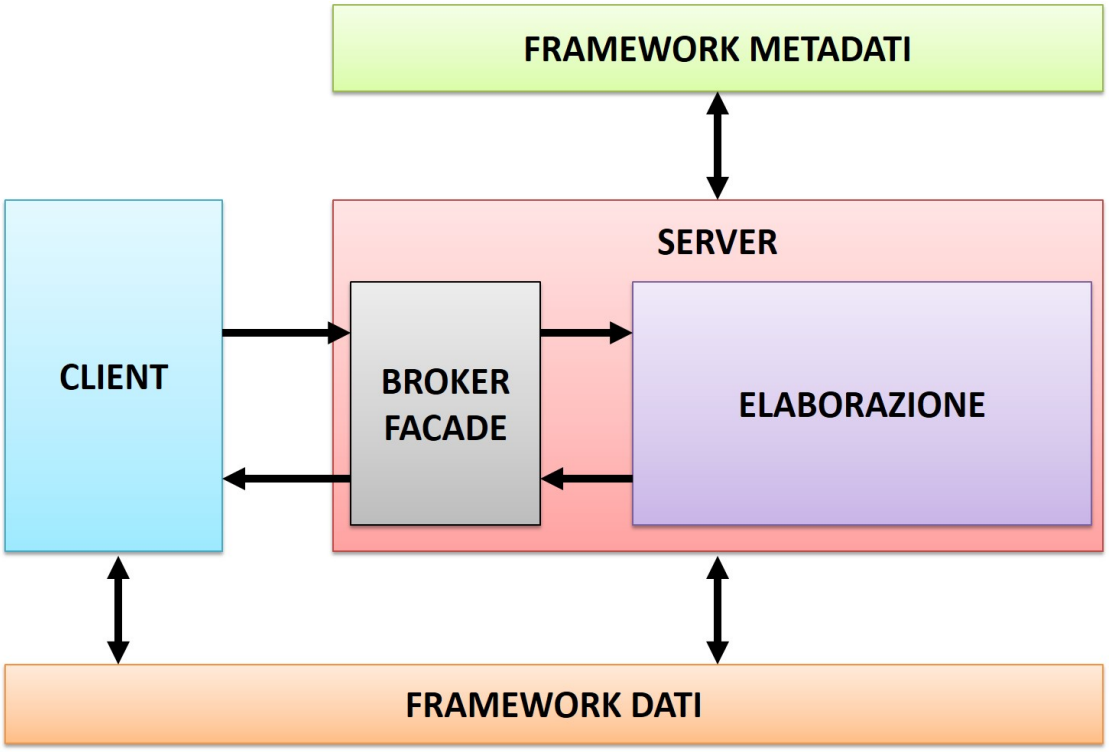

FIGURA 3.2: Architettura gestionale.

del gestionale. La strutturazione interna al gestionale in questione è formata da numerose macro parti che cooperano fra loro per fornire tutte le funzionalità lato server specifiche per questa soluzione aziendale.

Il sistema di comunicazione fra le parti è ciò che veramente desta il nostro interesse per concepire quale sia il funzionamento dell'applicazione. Ogni modifica ed operazione lato client genera una richiesta da spedire e gestire lato server, una volta evasa viene generata una risposta da consegnare al client, in un formato ottimizzato per essere trasferito tramite connessione web (socket C#). Questo specifico formato proprietario risulta un oggetto rappresentante una interfaccia grafica (pagina) nella sua interezza; viene spedito al client contenente il minimo numero di informazioni necessarie per massima performance. Ogni pagina è composta da due componenti: una statica che ne descrive la struttura in mancanza di dati interni, ed una dinamica capace di integrare i valori all'interno delle parti statiche dell'interfaccia.Ogni pagina grafica visualizzata dal client dell'applicazione è corredata da entrambe le parti per essere operativa. Una richiesta di pagina, solitamente, viene evasa lato server con un messaggio corredato dalle proprietà che caratterizzano solamente gli elementi dinamici dell'interfaccia. Se questa particolare interfaccia viene richiesta per la prima volta, allora viene generato anche

un altro messaggio, solitamente salvato in cache all'arrivo, che descrive tutta la struttura statica della pagina.

In sostanza anche se la gestione della parte grafica di interfacciamento con l'utente finale risulta un progetto indipendente (lato client), è in realtà fortemente accoppiato con le informazioni fornite durante l'esecuzione dal server.

La sincronizzazione fra le parti è garantita da una infrastruttura compatta; in sostanza a fronte di una connessione client il server si preoccupa di mantenere il suo stato in memoria. Oltre ad evadere le richieste crea ed aggiorna una interfaccia di meta livello sincronizzata con l'utente remoto. Questo meccanismo fornisce una resistenza maggiore agli errori di comunicazione e malfunzionamento lato client.

Le richieste infine sono divise in due differenti categorie a seconda della loro priorità, esistono richieste accodabili o meno. Nel primo caso vengono messe in attesa mentre nel secondo vengono servite subito dopo aver svuotato e servito la coda delle richieste in attesa fino a quel momento.

3.1.1 Modello dei dati

Il meccanismo di comunicazione, come già accennato, è basato su tecnologia socket ma viene strutturato a livello applicativo come una chiamata RMI [8] (Remote Method Invocation); pertanto è mancante di una parte di gestione esplicita dei messaggi. Nonostante ciò la quantità di dati passati attraverso il canale di comunicazione è di fondamentale importanza mantere prestazioni elevate; il gestionale infatti spesso lavora con diverse decine di utenti nello stesso momento.

Al fine di mantenere il più possibile una struttura dati semplice da passare fra le parti, molti dati ridondanti non vengono replicati e viene sfruttato un formato proprietario adatto allo scopo. In particolare parliamo di un formato costituito da un oggetto serializzabile che può contenere tutte le parti di una singola interfaccia. Nello specifico è composto come un classico oggetto, contenente proprietà e sotto elementi eventuali; può essere reso serializzabile tramite il supporto alla comunicazione socket.

Come si può vedere ogni interfaccia è incapsulata in un oggetto di tipo *Risposta*. Questa risposta può avere degli attributi ed essere di vari tipi, in particolare ci interesseremo di tutte quelle risposte legate all'interfaccia grafica e di tipo

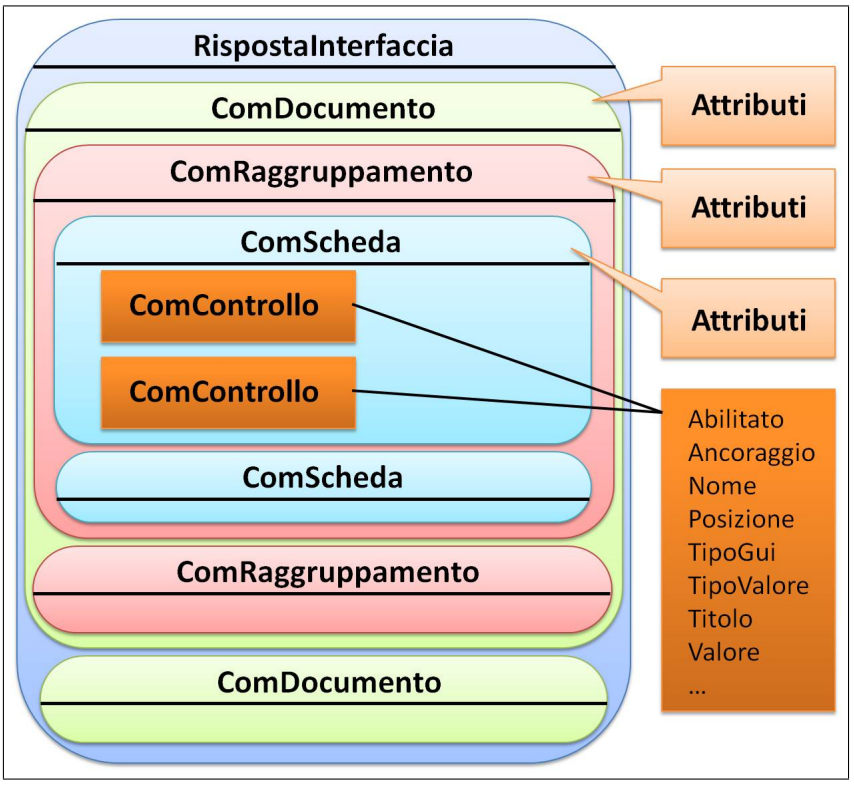

FIGURA 3.3: Messaggio di risposta server.

RispostaInterfaccia. Ognuna di queste possiede informazioni legate a diverse pagine che vengono definite *ComDocumento*; al loro interno è possibile trovare diversi raggruppamenti logici di schede chiamati *ComRaggruppamento*. Ogni risposta può contenere uno o più documenti, allo stesso tempo può contenere tipi di interfacce diverse come finestre di dialogo (*ComFinestraMessaggio*) oppure navigatori (*ComNavigatore*). Le schede si chiamano *ComScheda*, vengono visualizzate una alla volta all'interno di un singolo raggruppamento e possiedono gli elementi grafici di base, denominati a loro volta *ComControllo*. Questi ultimi possono essere di diverso tipo a seconda della loro natura e possiedono attributi variabili.

La caratterizzazione interna del programma è molto ricca e possiede diversi spunti di riflessione per un eventuale riprogettazione.

Capitolo 4

Tecnologie di distribuzione

Nelle pagine seguenti ci si discosta per un momento da un progetto software canonico; in questo modo è possibile analizzare il processo seguito all'interno di un'azienda all'analisi di nuovi requisiti. In particolare il presente capitolo si fa carico di uno studio preliminare di tecnologie e strumenti complementari in grado di raggiungere i nuovi obiettivi nel minor tempo possibile.

Prima di cominciare la progettazione di una soluzione software è stato concordato uno studio di fattibilità sulla remotizzazione dell'applicazione già esistente in azienda. A questo proposito è opportuno ricordare che in un ambiente commerciale il risparmio economico, e principalmente le tempistiche, sono di vitale importanza per l'efficacia di un prodotto. Una soluzione, seppur temporanea, in grado di abbattersi sul mercato molto prima dei concorrenti, risulta vincente nella maggior parte dei casi.

Il primo studio affrontato sarà perciò quello delle tecnologie in grado di remotizzare, in maniera interattiva tramite strumenti web, un'interfaccia grafica prima funzionante solo su di un framework specifico (.NET su Windows).

4.1 Infrastruttura di un SO

Prima di proseguire nella categorizzazione delle soluzioni appetibili, è opportuno fare un passo indietro, ed analizzare nel dettaglio il metodo di funzionamento di un

sistema operativo (SO) quale Windows, a livello di gestione Desktop e protocolli di connessione remota.

Il sistema di connessione remota ad una macchina Windows funziona tramite un protocollo denominato RDP (Remote Desktop Protocol) in grado di far autenticare un utente alla macchina in remoto; la stessa cosa è possibile da parte di più utenti contemporaneamente ma solo previo l'acquisto di una licenza per connessione. I protocolli di comunicazione remota sono due: RFB (Remote FrameBuffer) e X11, entrambi si appoggiano su di un livello di infrastruttura di rete abbastanza basso. Nel primo caso, come suggerisce il nome, il funzionamento si basa sul livello di framebuffer mentre nel secondo si interagisce con display bitmap comuni per sistemi operativi di tipo UNIX. Il protocollo RFB lavora spedendo pixel per pixel l'interfaccia al client remoto con un funzionamento flessibile ma poco efficiente, d'altro canto il protocollo X11 è un sistema in grado di remotizzare interfacce e dispositivi di ingresso, basato sull'idea del desktop remoto ma su finestre singole.

La connessione di client multipli sulla stessa macchina, il cui desktop deve essere reso disponibile in remoto, porta ad analizzare la struttura sottostante la creazione di un Desktop di lavoro. La generazione di un ambiente grafico di lavoro in questo sistema operativo ha una strutturazione piuttosto rigida, che lascia molto poco spazio di manovra.

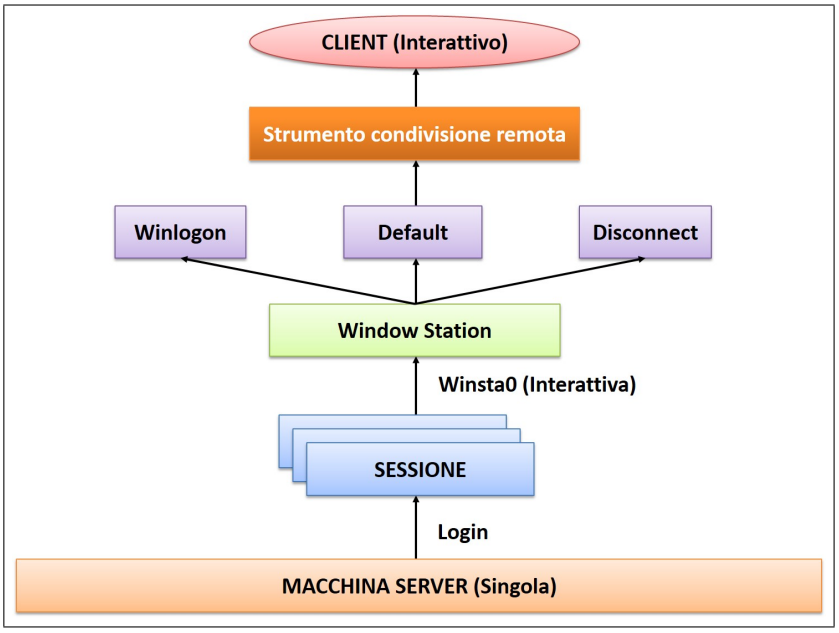

FIGURA 4.1: Struttura sistema operativo Windows.

Come si evince dalla Fig. 4.1 una macchina può possedere diverse *Sessioni* contemporaneamente legate a utenti differenti, delle quali solamente una risulta attiva in ogni momento. Ogni Sessione è in grado di generare più *Window Station* che servono a far interagire l'utente con l'interfaccia grafica; di queste solo la Station di default *Winsta0* è interattiva ed accetta un qualunque tipo di input da parte dell'utente. Ogni *Window Station* possiede diversi *Desktop*, dei quali solamente uno è attivo in un dato momento. Una station alla creazione è composta da tre differenti Desktop: "*Winlogon*", "*Default*" e "*Disconnect*". Di questi usualmente quello di lavoro è sempre il desktop di *Default* appartenente alla *Window Station Winsta0* così da poter raccogliere gli input utente.

4.2 Tecnologie di remotizzazione

Tenendo l'architettura appena analizzata in mente, è possibile vagliare le diverse alternative presenti online; così come sono oppure opportunamente modificate per ottenere un client remoto.

L'idea è quella di non modificare la parte server del gestionale e continuare a farlo funzionare in remoto; a questo punto un'altra macchina che funge da ponte verso il server può lanciare un numero indefinito di client. La stessa macchina può condividere la parte grafica di questi client agli utenti finali tramite connessione remota.

Da questo punto in avanti ci riferiremo a *macchina server* intendendo la macchina ponte in grado di connettere diversi client verso il server originale. La soluzione deve funzionare indipendentemente dal fatto che questa macchina server ponte sia o meno la stessa su cui è installato il server.

Tra le varie proposte disponibili online alcune sono risultate più adatte da subito al nostro scopo, infatti la remotizzazione di un desktop è stata la prima idea presa in considerazione a livello aziendale. La ricerca ha evidenziato alcune interessanti tecnologie, in parte open source, in grado di remotizzare un desktop appartenente ad una macchina con qualunque sistema operativo, anche verso una più accessibile finestra web in formato html5. Questo procedimento sembra il primo punto di attacco valido per interagire in remoto ad una applicazione funzionante su di una

macchina Windows. Si possono vedere di seguito alcune di queste soluzioni descritte in dettaglio:

- **Client-server VNC** - Progetto di tipo open source in grado di visualizzare in remoto un desktop proveniente da una macchina differente scritto completamente in c# e quindi indefinitamente modificabile e manutenibile per un'azienda con alte conoscenze di questo linguaggio. Fin dall'inizio si è rivelata l'alternativa più promettente e sulla quale si stava già lavorando in azienda, purtroppo nonostante la possibilità di generare diverse interfacce da mandare in remoto appartenente a client diversi è risultato un grave problema la condivisione dell'input sulla macchina server.

- **Thin VNC** - Soluzione proprietaria in grado di condividere un solo schermo da parte di un pc in remoto senza possibilità di login multipli da parte di differenti client.

- **Winswitch** - Progetto open source in grado di spedire le finestre singole di applicazioni da un sistema operativo in funzione ad un'altro senza però gestire l'input differenziale da parte di client multipli. Persiste la condivisione del controllo da parte delle diverse applicazioni fatte girare in remoto sulla macchina server.

- **Windows Presentation Foundation** - Soluzione pensata specificatamente per applicazioni scritte in linguaggio c# e funzionanti su tecnologia Windows Forms. La modifica del progetto prevede un passaggio a tecnologia simile basata sulle stesse API ma in grado di essere lanciata anche tramite browser web. Sfortunatamente la soluzione è ancora fortemente cablata sopra chiamate di sistema operativo Microsoft Windows e quindi inutilizzabile da browser diversi da Internet Explorer.

- **Cameyo** - Soluzione proprietaria in grado di impacchettare solamente file di installer di programmi destinati a sistemi operativi specifici da poter poi lanciare in remoto tramite l'installer modificato.

Ognuna di queste tecnologie espone una soluzione interessante basata però in larga parte sulla flessibilità del sistema operativo sottostante. In questo caso il problema più pertinente è risultato sempre quello dell'interazione con il server

da parte degli utenti remoti. Le problematiche da affrontare comprendono sia la creazione di diversi desktop interattivi differenti, ognuno da condividere con un dispositivo client specifico sul web, sia la gestione dell'input condiviso in modo da permettere una buona concorrenza.

Nel primo caso ci scontriamo violentemente con la chiusura applicata dal sistema operativo Microsoft che vuole forzare l'acquisto di molteplici licenze per RDP sulla stessa macchina server. Infatti, nonostante sia facilmente possibile creare molteplici desktop o anche molteplici windows stations, risulta impossibile, per motivi legati alla sicurezza, l'utilizzo contemporaneo di più desktop attivi; allo stesso modo risulta impossibile l'interazione contemporanea verso più window station differenti. Nel secondo caso, anche se non si presentano problemi di infattibilità, ben presto si evince un serio problema di scalabilità. La gestione dell'input concorrente infatti rappresenta un vero e proprio ostacolo, rendendo impossibile l'utilizzo di un singolo desktop per numerose applicazioni client condivise in remoto.

La carenza di flessibilità in questo ambito tecnologico chiude la strada verso una soluzione di replicazione desktop in remoto, ci si trova quindi costretti a riconsiderare un approccio di tipo software.

Capitolo 5

Una soluzione software

In questo capitolo viene affrontato in maniera scientifica il processo di sviluppo di un nuovo software in grado di soddisfare i nuovi requisiti di mercato. Si passerà attraverso le fasi classiche della progettazione partendo dall'analisi dei requisiti, che denotano la nostra analisi del problema, per poi finire nella progettazione di una soluzione specifica.

La prima parte verrà descritta dal punto di vista di un analista con la creazione di un modello, mentre la seconda avrà un accento più pratico rispecchiando le scelte tecniche di un progettista. Seguirà poi una visione dettagliata del codice prodotto per evidenziare come abbiano preso vita le parti del progetto. Infine avremo una breve nota su test, installazione e mantenimento del software appena sviluppato tenendo bene in evidenza quale siano le riflessioni fatte nel Cap. 1.

Una volta scartata l'alternativa di adattare il software gestionale grazie all'ausilio di un componente addizionale è opportuno eseguire un'analisi più approfondita del software stesso. Analizzando il progetto nelle sue parti fondamentali è possibile valutare dove poter integrare il software di supporto capace di adattarne il funzionamento ai nuovi requisiti; inoltre è possibile fare un'analisi dettagliata dei requisiti e del progetto della soluzione prima di passare all'implementazione vera e propria.

5.1 Analisi dei requisiti

I requisiti presentati nel Cap. 3 hanno portato ad una analisi del software approfondita per carpire le basi del funzionamento presente nelle pagine seguenti. Partendo dalla descrizione in linguaggio naturale si passa rapidamente ad una rappresentazione tramite strumenti più adatti alla modellazione per definire in maniera formale i requisiti del problema.

Glossario

Prima di continuare la nostra analisi è bene definire alcuni termini per snellire la trattazione da quì in avanti.

WebInterface: Applicazione web messa in esecuzione tramite browser in grado di comunicare con il Server via WebBroker.

WebBroker: Interfaccia di comunicazione e traduzione interposta fra WebInterface e Server. La parte fondamentale del progetto in analisi.

Server: Parte server dell'architettura del gestionale di cui si sta eseguendo la reingegnerizzazione.

Client: Browser generico in grado di visualizzare pagine scritte in linguaggio web e di metterne in esecuzione i contenuti.

5.1.1 Casi d'uso

Richiesta Pagina: Viene richiesta una pagina web al WebBroker.

Comunicazione Http: Viene stabilita una comunicazione con il WebBroker tramite protocollo http.

Comunicazione WebSocket: Viene stabilita una connessione tra WebInterface e WebBroker tramite protocollo WebSocket.

Gestione Messaggio: Viene gestito un messaggio inoltrato dal Client verso il WebBroker.

Gestione Login: Viene gestito un messaggio di tipo login.

Gestione Evento: Viene gestito un messaggio di tipo evento.

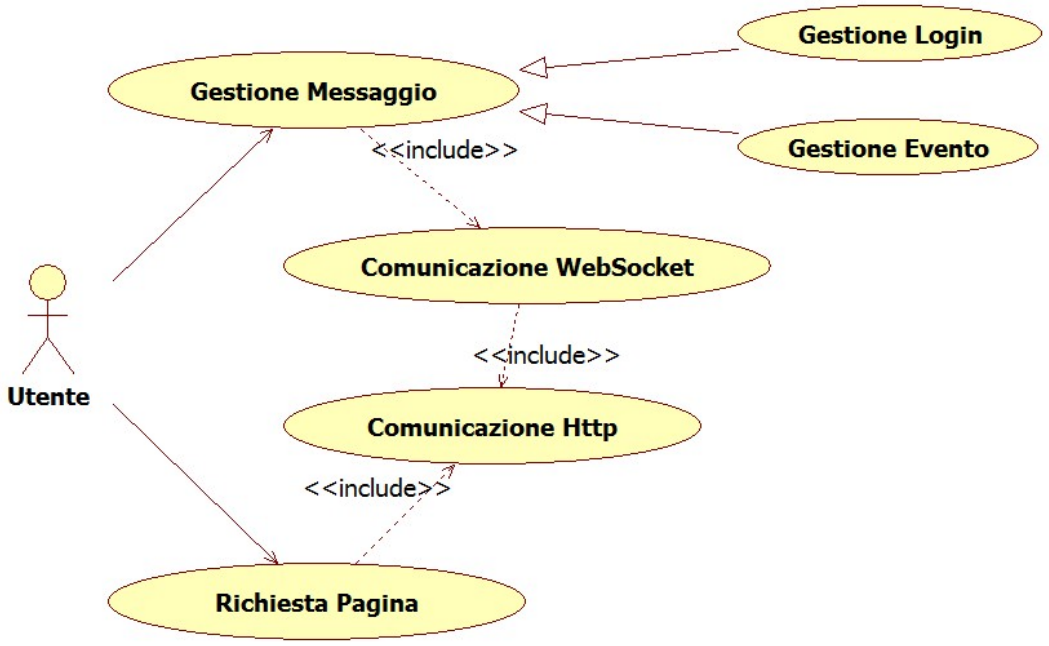

FIGURA 5.1: Casi d'uso.

5.1.2 Scenari

Richiesta Pagina

Attore principale: Utente

Attore secondario: WebBroker

Precondizioni: L'utente è provvisto di una connessione ad Internet.

Sequenza principale:

i L'utente richiede una pagina tramite browser all'indirizzo del WebBroker;

ii Il WebBroker restituisce la pagina richiesta, la prima richiesta restituisce sempre la pagina della WebInterface.

Postcondizioni: L'utente visualizza la schermata di login dell'applicazione.

Sequenza degli eventi alternativa:

i ServerOffline

ii PaginaNonTrovata

Comunicazione Http

Attore principale: Utente

Attore secondario: WebBroker

Precondizioni: L'utente è provvisto di una connessione ad Internet ed il WebBroker è online.

Sequenza principale:

i L'utente effettua una richiesta di comunicazione Http con il WebBroker;

ii Il WebBroker risponde alla richiesta secondo gli standard del protocollo fornendo dei dati in uscita se appropriato.

Postcondizioni: L'utente stabilisce una comunicazione con il WebBroker.

Sequenza degli eventi alternativa:

i PaginaNonTrovata

Comunicazione WebSocket

Attore principale: Utente

Attore secondario: WebBroker

Precondizioni: L'utente ha stabilito una comunicazione Http con il WebBroker e visualizzato la schermata principale della WebInteface.

Sequenza principale:

i L'utente richiede una comunicazione di tipo WebSocket con il WebBroker;

ii Il WebBroker apre una comunicazione di tipo WebSocket con la WebInterface.

Postcondizioni: La WebInterface ed il WebBroker sono ora in comunicazione e possono scambiarsi messaggi.

Sequenza degli eventi alternativa:

i ConnessioneInterrotta

Gestione Login

Attore principale: Utente

Attore secondario: WebBroker, Server

Precondizioni: L'utente ha stabilito una connessione di tipo WebSocket con il WebBroker.

Sequenza principale:

i L'utente fornisce le sue credenziali di login alla WebInterface;

ii La WebInterface spedisce un messaggio di tipo login al WebBroker;

iii Il WebBroker riceve il messaggio, estrapola le informazioni di login e richiede al Server di autenticare l'utente;

iv Il Server autentica l'utente;

v Il WebBroker notifica l'utente e la WebInterface permette all'utente di interagire con l'applicazione.

Postcondizioni: L'utente visualizza la schermata principale dell'applicazione.

Sequenza degli eventi alternativa:

i ConnessioneInterrotta

ii MessaggioNonValido

iii CredenzialiErrate

Gestione Evento

Attore principale: Utente

Attore secondario: WebBroker, Server

Precondizioni: L'utente ha stabilito una connessione di tipo WebSocket con il WebBroker.

Sequenza principale:

i L'utente interagisce con la WebInterface;

ii La WebInterface spedisce un messaggio di tipo evento al WebBroker;

iii Il WebBroker traduce il messaggio in uno dei tipi di eventi riconosciuti dal Server e gli inoltra la richiesta;

iv Il Server gestisce la richiesta e resituisce una risposta al WebBroker;

v Il WebBroker traduce la risposta del Server in un formato interpretabile dalla WebInterface e gli spedisce un messaggio di risposta;

vi La WebInterface interpreta la risposta ricevuta dal WebBroker.

Postcondizioni: La WebInterface aggiorna l'interfaccia coerentemente all'interazione dell'utente.

Sequenza degli eventi alternativa:

i ConnessioneInterrotta

ii MessaggioNonValido

iii EventoNonRiconosciuto

Sequenze degli eventi alternative

ServerOffline

Attore principale: Utente

Attore secondario: Nessuno

Precondizioni: L'utente non ha potuto contattare il WebBroker al suo indirizzo web.

Sequenza principale:

i Inizia al passo 1 di *Richiesta Pagina*.

Postcondizioni: Il browser notifica all'utente un errore di tipo 503 (Service Unavailable) e lo invita a riprovare in un secondo momento.

PaginaNonTrovata

Attore principale: WebBroker

Attore secondario: Nessuno

Precondizioni: Il WebBroker non ha potuto reperire la pagina richiesta.

Sequenza principale:

i Inizia al passo 1 di *Richiesta Pagina*.

Postcondizioni: Il browser notifica all'utente un errore di tipo 404 (Page not found) e lo invita a controllare la richiesta.

ConnessioneInterrotta

Attore principale: WebBroker

Attore secondario: Nessuno

Precondizioni: La connessione WebSocket fra WebInterface e WebBroker è stata interrotta.

Sequenza principale:

i Inizia in qualsiasi momento di *Comunicazione WebSocket*, *Gestione Login* o di *Gestione Evento*.

Postcondizioni: La WebInterface notifica all'utente la perdita di connessione.

MessaggioNonValido

Attore principale: WebBroker

Attore secondario: Nessuno

Precondizioni: Il WebBroker non ha riconosciuto il messaggio spedito dalla WebInterface.

Sequenza principale:

i Inizia al passo 3 di *Gestione Login* o al passo 3 di *Gestione Evento*.

Postcondizioni: Il WebBroker notifica alla WebInterface l'errore e si rimette in attesa di messaggi e questa notifica l'utente dell'errore consentendo però di continuare con l'interazione.

CredenzialiErrate

Attore principale: WebBroker

Attore secondario: Nessuno

Precondizioni: Le credenziali di login inserite dall'utente non sono corrette.

Sequenza principale:

i Inizia al passo 4 di *Gestione Login*.

Postcondizioni: La WebInterface notifica all'utente l'errore e lo invita ad inserire nuovamente le credenziali.

EventoNonRiconosciuto

Attore principale: WebBroker

Attore secondario: Nessuno

Precondizioni: Il WebBroker non è riuscito a tradurre l'evento in uno dei tipi riconosciuti dal Server.

Sequenza principale:

i Inizia al passo 3 di *Gestione Evento*.

Postcondizioni: Il WebBroker notifica alla WebInterface l'errore e si rimette in attesa di messaggi.

5.1.3 Modello del dominio

Per descrivere efficacemente il modello del dominio in analisi è stata sfruttata la capacità di rappresentazione package del linguaggio UML [14] (Unified Modeling Language), per mettere in evidenza le connessioni fra le varie parti in gioco. Uno strumento come questo può risultare molto efficace per definire la modellazione di un progetto, dall'inizio alla fine. In ambito aziendale viene fortemente sfruttato anche se, come vedremo, non sempre può definire in maniera precisa e non ambigua tutti gli elementi a livello di analisi.

Come si evince facilmente dal diagramma, sono presenti tre macro parti all'interno di questo sistema che rappresentano le entità che modellano il nostro dominio; in

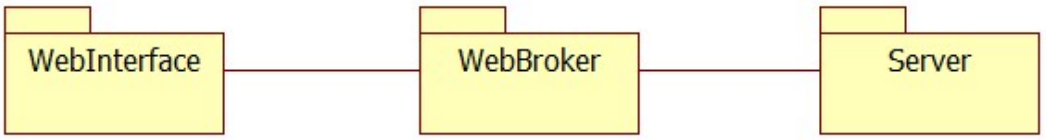

FIGURA 5.2: Modello del dominio.

particolare abbiamo la *WebInterface* come raffigurazione della nuova interfaccia grafica web, il *WebBroker* come raffigurazione del tramite per l'interazione fra *WebInterface* ed il vecchio server, ed il *Server* stesso inalterato dal progetto del gestionale.

Esaminando attentamente il diagramma illustrato, affiora in maniera netta la carenza espressiva a livello di interazione di uno strumento come UML. É necessario cambiare linguaggio per poter descrivere più nei dettagli le interconnessioni fra i vari ambiti del dominio in analisi. A questo scopo si rende utile Contact, uno strumento di modellazione sviluppato dal Prof. Antonio Natali, capace di esprimere in maniera estremamente formale e disambigua il dominio di un sistema sotto forma di struttura, interazione e comportamento.

Nel corso di questa analisi preliminare, è possibile produrre un documento formale che raffina il dominio grezzamente espresso in UML. In questo contesto infatti è possibile dare una definizione anche del protocollo di comunicazione fra le parti.

```
ContactSystem webApplication;

//Contesti
Context ctxWebApplication;
Context ctxWebBroker;

//STRUTTURA
Subject webInterface context ctxWebApplication;
Subject webBroker context ctwWebBroker;
Subject server context ctxWebBroker;

//INTERAZIONE
Request page;
Request message;
```

```
Request webLogin;
Request webEvent;

webInterfaceSendPage: webInterface demand page to webBroker;
webInterfaceSendMessage: webInterface demand message to webBroker;
webBrokerSendWebLogin: webBroker demand webLogin to server;
webBrokerSendWebEvent: webBroker demand webEvent to server;

webBrokerReceivePage: webBroker grant page;
webBrokerReceiveMessage: webBroker grant message;
serverReceiveWebLogin: server grant webLogin;
serverReceiveWebEvent: server grant webEvent;
```

Già in questa fase preliminare dell'analisi, è possibile porsi domande di interesse riguardanti i metodi di interazione fra le vari macro parti messe in evidenza dall'analisi del dominio. Possiamo notare la presenza delle tre macro parti già presentate in Fig. 5.2, anche se, in questo caso, vengono messi in evidenza anche i tipi di interconnessione.

Linguaggio Contact

Questo sistema ha come obiettivo quello di aiutare gli analisti software nel costruire modelli per sistemi distribuiti ad alto livello, utilizzabili come nuova forma di codice sorgente. Il sistema Contact è stato costruito con una serie di linguaggi e metamodelli, così da ridurre significativamente il divario fra modello ed implementazione; potendo ottenere la maggior parte del codice in maniera automatica tramite un ambiente di sviluppo appropriato. Questo linguaggio è stato sviluppato sfruttando la tecnologia Xtext che lega insieme metamodelli con linguaggio di programmazione, fornendo strumenti in grado di creare *parser*, *editor* di sintassi ed una sintassi astratta generati automaticamente da specifiche scritte in grammatica *EBNF*. Per una descrizione completa riguardo alle capacità, implementazioni e sviluppi futuri di questo linguaggio si rimanda al sito del docente [2].

Messaggi web

In questa sede è opportuno soffermarsi anche sulla discussione del tipo di messaggi scambiati fra le parti, per definire gli estremi per una divisione futura del lavoro e spezzare il progetto fra diversi servizi in comunicazione remota fra loro. Questa accortezza permette anche lo sviluppo di alcuni moduli del sistema da parte di terzi in possesso delle sole specifiche di comunicazione.

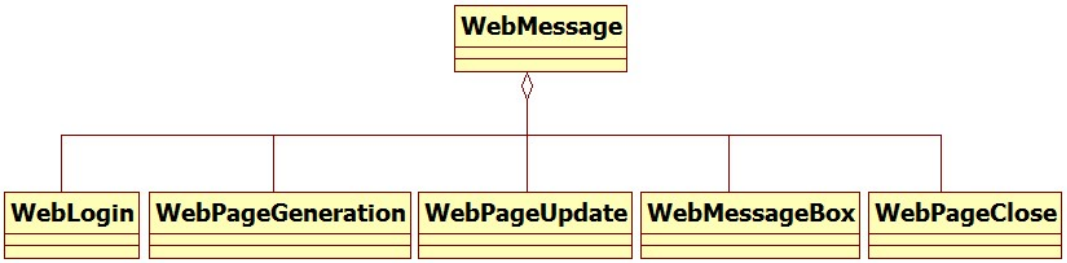

FIGURA 5.3: Tassonomia messaggi web.

Dopo un'attenta analisi dei requisiti è emersa una struttura piuttosto ricca di messaggi da scambiare fra WebBroker e WebInterface; questo per garantire una adeguata interpretazione delle risposte da parte della nostra applicazione.

Ogni messaggio scambiato tramite canale di comunicazione può essere incapsulato in un'unico elemento, il quale può essere composto da uno o più tipologie di messaggi. Ogni elaborazione lato server, infatti, può generare diversi messaggi di risposta che devono essere gestiti opportunamente lato client. Come possiamo notare in Fig. 5.3 è possibile avere: un messaggio di risposta all'operazione di login *WebLogin*, un messaggio di generazione nuova pagina *WebPageGeneration*, un messaggio di aggiornamento pagina esistente *WebPageUpdate*, un messaggio di generazione finestra di dialogo *WebMessageBox* oppure un messaggio di chiusura pagina esistente *WebPageClose*. Infine, come già accennato, viene scambiato solo un elemento tramite connessione web ovvero l'aggregato *WebMessage*.

5.2 Analisi del problema

Si presenta ora lo studio del problema ad un suo prima meta livello. L'obiettivo, posto fin da subito, è quello di ottenere un'alternativa alla parte client del gestionale in analisi.

Studiando il progetto del software pare evidente quale sia il punto di integrazione con l'architettura esistente. E' opportuno, infatti, rendere trasparente lato server quale tipo di client stia facendo una richiesta; sia esso un client di tipo classico Windows Forms oppure un client di tipo web.

I requisiti spingono a considerare una soluzione in grado di garantire la capacità di connessione al server da parte del maggior numero di dispositivi tablet differenti. Allo stesso tempo dovrà garantire una funzionalità completa, o comunque la possibilità di estensione modulare per aggiungere funzionalità in seguito.

In relazione all'analisi del Cap. 5.1 possiamo pensare ad un progetto che si interponga come interfaccia tra il server del gestionale ed un browser web generico. In queste condizioni è opportuno sviluppare una soluzione capace di interagire con l'utente tramite un'interfaccia grafica il più possibile fedele all'originale. Questa può essere generata sfruttando i file di descrizione presenti come meta dati nel client originale.

A fronte dell'interazione dell'utente, vengono inviati appositi dati relativi all'operazione da eseguire al nostro substrato. Questo provvede alla traduzione di suddetti dati in un formato interpretabile dal server effettivo. Una volta consegnata al server l'opportuna richiesta tradotta, possiamo gestire la risposta presa in carico e tradurla nuovamente, questa volta in un linguaggio parlato dal nostro nuovo client web. Questo meccanismo definito a parole fornisce una prima panoramica del funzionamento logico del software che ci prestiamo a progettare.

WebSocket

Una tecnologia sicuramente molto voluta dall'azienda è stata quella della comunicazione tramite Websocket. Questa modalità di comunicazione permette una interazione web paragonabile a quelle delle socket tradizionali; in questo modo è possibile ottenere pagine capaci di aggiornarsi in tempo reale a fronte di una interazione utente.

WebSocket [9] è una tecnologia web in grado di fornire un canale di comunicazione di tipo full-duplex attraverso una singola connessione TCP la cui API è standardizzata dall'IETF (Internet Engineering Task Force) come RFC 6455. Questa tecnologia è stata disegnata sia per utilizzo lato browser che server e

può essere sfruttata da qualunque applicazione client - server. Permette una maggiore interazione tra un browser ed un server e il suo unico legame con l'HTTP è il metodo in cui viene effettuato l'handshake tramite un "upgrade request" della connessione stessa. Uno strumento come quello in analisi è nato dalla necessità sempre crescente di una comunicazione in tempo reale tra browser e server per consentire lo sviluppo di applicazioni online sempre più evolute ed in grado di garantire una interazione fluida con l'utente. La specifica è stata realizzata in concomitanza con l'arrivo di HTML5 in un periodo prospero per quanto riguarda le tecnologie web ed è supportata già da una vasta gamma di browser ampiamente utilizzati. Questo tipo di connessione offre numerosi vantaggi per gli sviluppatori di giochi online portando l'interazione ad un nuovo livello, la nuova tecnologia presenta però numerosi vantaggi anche per applicazioni online di altro interesse come vedremo in breve. Per entrare più nello specifico una websocket è una connessione TCP persistente, bi-direzionale, full-duplex garantita da un sistema di handshaking client-key ed un modello di sicurezza origin-based. Il sistema inoltre maschera le trasmissioni dati per evitare lo sniffing dei pacchetti di testo in chiaro. Questa definizione può essere analizzata più nei dettagli facendo riferimento alla tabella seguente.

L'utilizzo di questa tecnologia risulta molto semplice ed intuitivo nonché in linea con alcuni principi ingegneristici che permettono una interazione event-driven. Nonostante numerosi vantaggi presentati da questa specifica le websocket non possono essere una soluzione universale, il protocollo http riveste ancora un ruolo chiave nella comunicazione tra client e server per trasferimenti di dati di tipo one-time, come caricamenti iniziali. Le richieste http infatti sono in grado di eseguire questo tipo di operazioni in modo più efficiente delle websocket chiudendo le connessioni una volta completate. Inoltre è sempre bene ricordare che non tutti i browser possono sfruttare il potere di questa nuova tecnologia se Javascript non è abilitato.

In conclusione esistono numerose implementazioni websocket lato server ed una vasta gamma di browser in grado di supportare il protocollo e ciò basta a rendere le websocket una tecnologia di vasto interesse per il mercato moderno anche in vista di una sua rapida espansione ed evoluzione.

Termine	Significato
Bi-direzionale	Il client è connesso al server, e il server è connesso al client. Ciascuno può ricevere eventi come collegato e scollegato e possono inviare dati all'altro.
Full duplex	Il server e il client possono inviare dati nello stesso momento senza collisioni.
TCP	Protocollo sottostante tutte le comunicazioni Internet, che fornisce un meccanismo affidabile per trasportare un flusso di byte da un'applicazione all'altra.
Client-key handshake	Il client invia al server una chiave segreta di 16 byte con codifica base64. Il server poi aggiunge una stringa (anche detta "magic string" e specificata nel protocollo come "258EAFA5-E914-47DA-95CA-C5AB0DC85B11") e rimanda il risultato, elaborato con SHA1, al client. In questo modo, il client può essere certo che il server cui aveva inviato la sua chiave è lo stesso che apre la connessione.
Sicurezza origin-based	L'origine della richiesta WebSocket viene verificata dal server per determinare che provenga da un dominio autorizzato. Il server può rifiutare le connessioni socket da domini non attendibili.
Trasmissione dati mascherata	Il client invia una chiave di 4 byte per l'offuscamento nella trama iniziale di ogni messaggio. Essa è utilizzata per effettuare uno XOR bit a bit tra dati e chiave. Ciò aiuta a prevenire lo sniffing di dati, poiché un malintenzionato dovrebbe essere capace di determinare il byte di inizio del messaggio per poterlo decrittare.

TABELLA 5.1: Terminologia WebSocket.

5.2.1 Architettura logica

Il modello presentato di seguito è descritto tramite due strumenti formali di analisi. In primo luogo viene presentato grazie al linguaggio Contact sotto un unico documento analizzato in parti per comodità. In secondo luogo viene sfruttato il più classico linguaggio UML scritto tramite Microsoft Office Visio per meglio supplire a necessità aziendali di documentazione. Nonostante il linguaggio Contact risulti molto più espressivo, formale e puntuale nella descrizione di un modello di progetto, per motivi di retrocompatibilità aziendale è importante mantenere una documentazione uniforme. A questo scopo lo strumento più indicato è Visio in grado di poter importare i suoi diagrammi delle classi e di flusso direttamente in Visual Studio, programma preferenziale di sviluppo in azienda.

Struttura

In prima battuta si può notare come il sistema in analisi sia distribuito; pertanto è opportuno contemplare due contesti separati in cui inserire differenti sottoparti del nostro futuro progetto.

```
Context ctxWebApplication;
Context ctxWebBroker;
```

In particolare ogni utente che vuole interagire con il nostro server disponibile via web lo può fare da un qualunque browser web connesso ad Internet; questo contesto viene rappresentato da *ctxWebApplication*. Le modifiche da apportare al server esistente, e quindi la sua nuova interfaccia web di interazione con client remoti, viene invece rappresentata dal contesto *ctxWebBroker* in ascolto di comunicazioni dall'esterno.

Scendendo nei dettaglio si possono definire le entità in gioco su questi due contesti separati:

```
Subject webInterface context ctxWebApplication;
```

Si vedrà di seguito come le entità presenti sul contesto *ctxWebApplication* siano in realtà molto semplici e di scarso interesse per il progetto. La generalità richiesta dalle specifiche, infatti, abbassa di molto la complessità di realizzazione di questo nodo del sistema. L'obiettivo di poter utilizzare il maggior numero di modalità di accesso al servizio permette di non curarci di questa parte in fase di progetto.

```
Subject server context ctxWebBroker;
```

Allo stesso modo vincoli di massima retrocompatibilità lato server permettono di ignorare completamente l'entità rappresentante il vecchio server in questa fase. L'unico dettaglio di interesse, in fase di progettazione e specialmente di implementazione, è legato al punto di interazione con il server già presente in azienda. In particolare i messaggi da dover scambiare con questo nodo del progetto sono di fondamentale importanza.

```
Subject webBroker context ctxWebBroker;
```

Passando ora al cuore del sistema è possibile vedere come sia posizionato sullo stesso contesto del server, anche se in questa sede continuano a scambiarsi messaggi. Questa architettura è stata usata in fase di analisi per evidenziare le modalità di comunicazione ed interazione con il server già implementato. In fase di progettazione si potrebbe prendere in considerazione sia l'alternativa di porre *WebBroker* e server sulla stessa macchina ed accoppiarli maggiormente o, seppure, di distribuirli e continuare a mantenere lo scambio di messaggi qui definito.

Interazione

Per quanto riguarda l'interazione fra le parti è necessario cominciare dividendo i messaggi scambiati fra client e *webBroker* e tra quest'ultimo e server. Nel primo caso abbiamo due differenti tipologie di messaggio: la richiesta di una pagina web e la richiesta di evasione di un servizio lato client. Nel secondo caso abbiamo due differenti tipologie di messaggio che riguardano il login dell'utente o la richiesta di evasione di un evento generico. Tutta questa infrastruttura è progettata su di un modello ad eventi così da ottenere un applicativo web che sembri vivo, lasciando piena libertà di interazione all'utente. Tutte le richieste infatti sono non bloccanti per il client, e le risposte relative vengono elaborate dall'applicazione web come disaccoppiate dalla richiesta.

```
Request page;
Request message;

webInterfaceSendPage: webInterface demand page to webBroker;
webInterfaceSendMessage: webInterface demand message to webBroker;

webBrokerReceivePage: webBroker grant page;
webBrokerReceiveMessage: webBroker grant message;
```

Come è possibile notare questi sono i messaggi scambiati fra *webInterface* e *webBroker*: il primo rappresenta una richiesta di pagina da visualizzare via browser mentre il secondo rappresenta un messaggio generico. Nell'ambito del messaggio generico possiamo sia scambiare credenziali di login da parte dell'utente, sia scambiare dati relativi ad eventi scatenati dall'utente. Il primo messaggio di richiesta pagina viene evaso direttamente dal *webBroker* mentre il secondo richiede l'intervento del server per elaborare la richiesta.

```
Request webLogin;
Request webEvent;

webBrokerSendWebLogin: webBroker demand webLogin to server;
webBrokerSendWebEvent: webBroker demand webEvent to server;

serverReceiveWebLogin: server grant webLogin;
serverReceiveWebEvent: server grant webEvent;
```

In questa parte è possibile vedere i messaggi scambiati fra *webBroker* e server; questi rappresentano le richieste di login da client remoto e di evasione evento generico. Entrambi vengono processati via server e quest'ultimo differenzia nettamente le due interazioni. A questo stadio dell'analisi è stato preferito lasciare un modello a comunicazione, lasciando così spazio in futuro per un'implementazione di server sotto forma di servizio esterno.

Comportamento

Il comportamento descritto nel seguente paragrafo si limita a dare una panoramica funzionale delle parti in gioco, evidenziando quale sia il flusso di lavoro all'interno del progetto e denotando le macro aree che compongono il sotto progetto *webBroker*.

webInterface

```
BehaviorOf webInterface {

        var msgSent = 0
        var pageName = ""
        var loginInfo = "username+password"
        var eventName = ""
        var eventsNumber = 10

        val elements.Message message = new elements.Message()

        action String compress(elements.Message message)
        action void showPage(elements.Page page)
```

```
        action int increment(int number)

        state webInterfaceInit initial
                showMsg("START -- webInterface")
                goToState requestPage
        endstate

        state requestPage
                set pageNmae = "Homepage"
                showMsg("Requesting page: " + pageName)
                doOutIn webInterfaceSendPage(pageName)
                acquireAnswerFor page
                call showPage(call curReply.msgContent())
                goToState sendLogin
        endstate

        ...
}
```

Questo elemento risulta una specie di *mock object* che replica la capacità di comunicazione di un browser web. Risulta particolarmente interessante la richiesta di una pagina web e l'invio di messaggio di login ed evento.

Molto importante la dichiarazione e l'utilizzo del metodo di compressione dati *compress* che suggerisce ai progettisti la necessità di un protocollo di compressione e/o trasferimento dati via Internet tramite il quale vengono processati i messaggi.

```
BehaviorOf webInterface {

...

        state sendLogin
                call message.setType("login")
                call message.setContent(loginInfo)
                showMsg("Sending login informations...")
                doOutIn webInterfaceSendMessage(exec compress(message))
                goToState sendEvent
        endstate
```

```
        state sendEvent
                set msgSent = exec increment(msgSent)
                set eventName = "event " + msgSent
                call message.setType("event")
                call message.setContent("eventName")
                showMsg("Sending event: " + eventName + " to webBroker")
                doOutIn webInterfaceSendMessage(exec compress(message))
                if{msgSent <= eventsNumber}{goToState sendEvent}
                showMsg("END -- webInterface")
                transitToEnd
        endstate
}
```

Lo pseudo codice visto replica il comportamento di un client web all'atto di comunicare con il nostro *webBroker*: prima richiede la pagina principale, poi effettua il login ed infine spedisce una serie di eventi di interazione.

webBroker

```
BehaviorOf webBroker {

        var messageType = ""

        val elements.Translator translator = new elements.Translator()
        val elements.Generator generator = new elements.Generator()
        val elements.FilesContainer filesContainer = new elements.FilesContainer()
        val elements.Page page = new elements.Page()

        action elements.Message decompress(String stringMessage)

        state webServerInit initial
                showMsg("START -- webServer")
                goToState waitForRequests
        endstate

        state waitForRequests
                onMessage page transitTo webBrokerReceivePage
```

```
            onMessage message transitTo webBrokerReceiveMessage
        endstate

        ...

}
```

É possibile notare già dalla dichiarazione delle variabili interne gli ambiti di interesse. In particolare si evince fin da subito la necessità di una sotto parte in grado di tradurre gli eventi generati dal client (*translator*) e di una sotto parte in grado di generare pagine web a partire dalle risposte del server (*generator*). Inoltre è importante inserire una sotto parte del progetto incaricata di gestire: sia le pagine web generate dalle risposte, sia le pagine relative all'applicazione web lato client da mostrare (*filesContainer*). Infine è interessante notare la dichiarazione di un metodo di decompressione dati provenienti da messaggi client; questo presuppone l'intenzione e la necessità di avere un protocollo di compressione e/o un formato adatto al trasferimento dati via Internet duale a quello sfruttato dalla *webInterface*.

Una volta terminata la dichiarazione di variabili e metodi si passa velocemente dallo stato iniziale a quello di ciclo di attesa richieste facendo diventare l'entità un servizio sempre attivo.

```
BehaviorOf webBroker        {

    ...

        state webBrokerReceivePage
                set pageName = code.curInputMsgContent
                showMsg("Received page request: " + pageName)
                showMsg("Retrieving page...")
                call filesContainer.getPage(pageName)
                goToState waitForRequests
        endstate

        state webBrokerReceiveMessage
                set message = exec decompress(code.curInputMsgContent)
                set messageType = call message.getType()
                showMsg("Received message: " + messageType)
```

```
                if{messageType = "login"}{goToState handleLogin}
                if{messageType = "event"}{goToState handleEvent}
        endstate

        state handleLogin
                showMsg("Logging to server...")
                doOutIn webBrokerSendWebLogin(call message.getContent())
                acquireAnswerFor webLogin
                showMsg("Login result: " + call curReply.msgContent())
                goToState waitForRequests
        endstate

        state handleEvent
                set eventName = call message.getContent()
                showMsg("Translating event...")
                set eventName = call translator.translate(eventName)
                showMsg("Processing event: " + eventName + " ...")
                doOutIn webBrokerSendWebEvent(eventName)
                acquireAnswerFor webEvent
                showMsg("Generating page...")
                call page.setContent(call generator.generate(call
                        curReply.msgContent()))
                showMsg("Page generated.")
                goToState waitForRequests
        endstate
}
```

A seconda del messaggio ricevuto si può transitare in stati diversi per evadere la richiesta opportunamente e riportarsi velocemente allo stato di attesa. In caso di ricezione richiesta di pagina si potrebbe delegare la richiesta alla sotto parte opportuna (*filesContainer*), mentre in caso di messaggio generico si dovrebbe discernere ulteriormente.

Una volta decompresso il messaggio si può analizzarne il tipo e, in caso di richiesta di login, spedire l'opportuna *Request* al server; in caso invece di evento si dovrebbe, prima delegarne la traduzione all'apposita sotto parte, e poi spedire la richiesta al server.

server

```
BehaviorOf server {

        action void delay(int milliseconds)

        state serverInit initial
                showMsg("START -- server")
                goToState waitForWebRequests
        endstate

        state waitForWebRequests
                showMsg("Waiting for web requests...")
                onMessage webLogin transitTo serverReceiveWebLogin
                onMessage webEvent transitTo serverReceiveWebEvent
        endstate

        state serverReceiveWebLogin
                set loginInfo = code.curInputMsgContent
                showMsg("Received login request: " + loginInfo)
                showMsg("Attempting login...")
                exec delay(const.2000)
                goToState waitForWebRequests
        endstate

        state serverReceiveWebEvent
                set eventName = code.curInputMsgContent
                showMsg("Received event: " + eventName)
                showMsg("Processing request...")
                exec delay(const.2000)
                goToState waitForWebRequests
        endstate
}
```

É possibile inoltre notare come dallo stato iniziale il server si ponga velocemente in uno stato passivo di attesa di richieste. A questo punto viene replicata la capacità di evasione di richieste di login, oppure di elaborazione eventi, tramite un metodo di comodo *delay* in grado di ritardare la ripresa del programma.

5.2.2 Divario di astrazione

Lo strumento formale Contact solitamente viene in aiuto in questa fase dell'analisi del problema con la sua capacità di generazione automatica di codice. Sfortunatamente l'infrastruttura può generare solo codice per piattaforma Java e lascia perciò inalterato il divario fra il modello appena descritto ed il suo progetto.

É necessario quindi implementare ogni parte del modello riguardante il WebBroker manualmente, compresa la parte di WebInterface da fornire come interfaccia all'utente finale. A questo scopo verrà generato del codice in C# tramite ambiente .NET traendo vantaggio dal "know how" aziendale. In questo modo il progetto può essere inglobato efficacemente con il resto del gestionale una volta completo, essendo in linea con il processo di sviluppo.

5.2.3 Analisi dei rischi

Il tentativo è quello di generare un modello, che non solo possa essere riutilizzato facilmente per ampliare il potenziale di altri software aziendali, ma possa allo stesso tempo gettare le basi per la scrittura di uno strumento di generazione automatica di interfacciamento tecnologico. Come evidenziato nel Cap. 1 infatti è sempre più importante fornire ad un software una finestra sul mondo tramite connesione Internet; uno strumento come questo potrebbe espanderne il potenziale adattandolo velocemente ed in maniera efficace.

5.3 Piano di lavoro

Il lavoro di progetto, legato all'analisi appena presentata, è stato diviso in sottoparti per gestirne la complessità ed ottenere un flusso di lavoro adeguato. In questo modo è possibile garantire un avanzamento delle capacità del nostro progetto in maniera lineare in relazione al tempo impiegato. In pratica, ponendo delle basi solide di funzionamento ed infrastruttura nelle prime fasi di lavoro, è possibile aggiungere funzionalità in un secondo momento senza ulteriori sforzi di progettazione.

A questo scopo il primo passo è quello di tradurre una richiesta elaborata dal server già sviluppato in azienda. In particolare è importante partire da una sua risposta e tradurla in un linguaggio facilmente comprensibile lato client web; sia essa una risposta di generazione nuova pagina oppure di modifica di pagina esistente.

Una volta ottenuta questa capacità, ed aver sviluppato un generatore di risposte web (sotto sistema *generator*), il secondo passo è quello di generare un client web in formato adatto ad interagire in maniera semplice con il server, spedendo richieste in formato semplificato. Allo stesso tempo richiede impegno la selezione di un server web adatto alla gestione di connessioni multiple, sia di richiesta di pagine che di comunicazione via WebSocket.

In seguito è possibile concentrarsi sulla capacità di traduzione lato server dei messaggi spediti da parte del nuovo client web. Questo sfrutterà un formato adatto all'elaborazione (sotto sistema *translator*) considerando tutte le funzionalità principali da implementare: come la richiesta di nuove pagine o la modifica di quelle già visualizzate.

Infine si può perfezionare il lavoro chiudendo l'infrastruttura di comunicazione client - server, garantendo al client la capacità di gestione di messaggi di risposta in grado di modificare l'interfaccia.

Sviluppate queste capacità è possibile concentrarsi su alcuni dettagli di importanza secondaria dal punto di vista del funzionamento. Questi hanno comunque una importanza vitale ai fini del test dell'applicativo in ambiente distribuito, al fine di ottenere un feedback sulle scelte fatte in sede di progetto.

Una volta implementate le funzionalità di base, arricchite da alcune capacità fondamentali si può lasciare spazio ai test; così da saggiare la bontà dell'applicazione. Partendo dai risultati degli stessi si possono auspicare eventuali sviluppi futuri.

5.4 Progetto

Scendendo più nel dettaglio si puà facilmente descrivere la struttura da applicare come modifica al prodotto aziendale.

Volendo avere un supporto a livello web, capace di interoperare con qualsivoglia strumento di tipo tablet e, possibilmente, di tipo smartphone, è necessario sviluppare un nuovo client in grado di funzionare completamente via browser web. Possibilmente il progetto dovrebbe supportare il maggior numero di browser web disponibili per dispositivi mobili in commercio.

Le interfacce del client originale sono descritte per mezzo di oggetti ben strutturati che rappresentano, nella loro interezza, tutte le parti di una interfaccia grafica. Questo risulta un grande vantaggio e costituisce il punto di partenza da cui generare l'applicativo. In particolare, ogni richiesta fatta da parte del client viene gestita dal server tramite una risposta, questa consiste in un oggetto complesso contenente tutta la descrizione dell'interfaccia grafica da implementare lato client.

Il client è formato sia da alcune pagine statiche sia da alcune pagine generate tramite le risposte di cui sopra durante l'esecuzione. Per questo è necessario prevedere sia un meccanismo di mantenimento di file statici sia di un meccanismo di generazione automatica delle pagine richieste in corso di elaborazione.

Per quanto riguarda le pagine generate dinamicamente si deve scardinare il forte accoppiamento fra eventi client e richieste server; è quindi necessario sviluppare una parte in grado di tradurre efficacemente eventi generati sulla nostra interfaccia web in richieste interpretate a livello server. Dualmente poi, è necessario tradurre le risposte server in un formato facilmente leggibile da parte del client.

Per quanto riguarda invece le pagine statiche, da rendere disponibili sotto richiesta di un client in esecuzione, è importante improntare un meccanismo duale di comunicazione lato client. Questo è in grado sia di richiedere risorse statiche, sia di parlare direttamente con il server per spedire eventi e ricevere dati dinamici.

Riesaminando il modello del dominio sotto la luce di queste affermazioni ne è stato generato uno intermedio, capace di mettere in evidenza le sottoparti del progetto (*WebBroker*). Queste possono sia essere viste come parti integranti del progetto, e quindi essere sviluppate in concomitanza al progetto principale, sia come servizi, ed essere così delegati a terzi o ad un diverso gruppo di progettisti. Le parti citate sono le stesse portate alla luce nel Cap. 5.2.1 e dividono la complessità e le aree di lavoro della fase di progetto.

L'analisi dei requisiti fatta al Cap. 5.1 ha negato la necessità di introdurre nuove entità o messaggi, lasciando ai progettisti il compito di scegliere le tecnologie

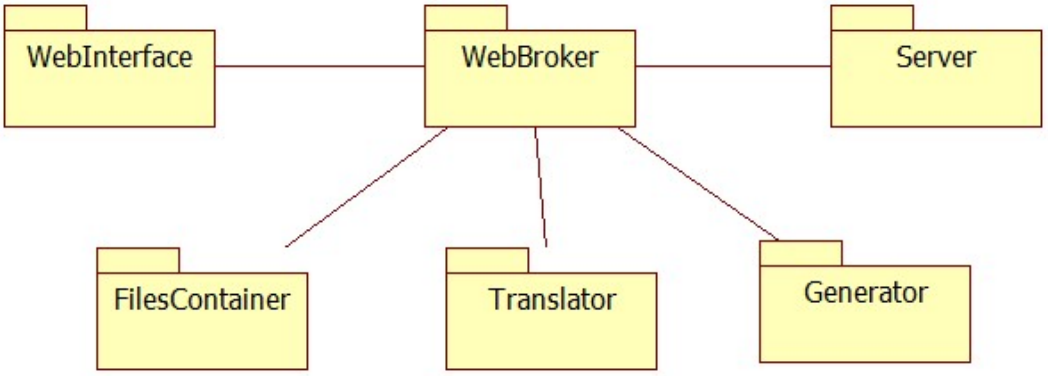

FIGURA 5.4: Sotto parti di progetto.

più adatte alla realizzazione di un primo prototipo. L'attività più complessa ed impegnativa è stata rappresentata dalla replicazione, in linguaggio web, di funzionalità già presenti lato client; questo processo ha lasciato interrogativi aperti sulle tecnologie da sfruttare per la sua realizzazione.

Nel seguito del capitolo sarà presente una breve panoramica sulle problematiche tecnologiche che si sono presentate durante la progettazione. La loro risoluzione ha costituito gran parte dello sforzo implementativo del primo prototipo.

5.4.1 Ambiente di sviluppo

La scelta di un ambiente di sviluppo è risultata abbastanza semplice essendo fortemente vincolata all'ambiente aziendale. Lo sviluppo di software in questo ambito infatti è legato a Visual Studio 2012 come strumento di programmazione, come già messo in evidenza dall'analisi della tecnologia da reingegnerizzare.

É molto importante tenere a mente questa scelta, fin da subito, prima di fare ulteriori analisi tecnologiche approfondite, ulteriori decisioni verranno infatti profondamente modificate dagli strumenti disponibili per questa piattaforma. In particolare verrà utilizzato il più possibile un canale di distribuzione pacchetti privilegiato come *NuGet* in grado di fornire file .dll sviluppati da terze parti. Questi file mettono a disposizione una serie di nuove primitive codificate in C# adatte a colmare necessità specifiche. Spesso i progetti dietro alla distribuzione di questi pacchetti di riferimento sono *open source*, e garantiscono il libero utilizzo del loro codice fornendo i sorgenti nella maggior parte dei casi. Questa peculiarità è risultata fondamentale nella scelta

fra diverse opzioni, in quanto, un'azienda con un forte "know how in linguaggio specifico, può trarre grandi benefici da progetti che danno accesso al codice sorgente. É possibile infatti partire da questi e modificarli, anche marginalmente, per ottenere soluzioni proprietarie che garantiscono il corretto funzionamento aziendale.

In sostanza da qui in avanti la scelta della piattaforma influenzerà notevolmente le decisioni tecnologiche, in modo che risultino facilmente integrabili con ambiente di sviluppo *Visual Studio 2012* tramite distribuzione pacchetti *NuGet*. Infine verrà garantita maggiore importanza a progetti attivi e con codici sorgente aperti al pubblico per eventuali modifiche.

5.4.2 Server web

Per la necessità di gestione pagine online e fornitura servizi di connessione WebSocket remote, si è resa necessaria la presenza di un "oggetto" da interporre fra il nostro applicativo e l'utente finale. L'utente potrà interagire con il server tramite un qualunque browser, a questo scopo si vuole avere un server web [15] con funzionalità di base ed allo stesso tempo in grado di gestire connessioni di tipo WebSocket. Queste funzionalità comprendono principalmente la risposta a richieste di tipo *GET*, espresse tramite protocollo http da un qualunque browser web. Ovviamente è importante mantenere aperta la comunicazione a tutti i tipi di messaggi e richieste che possono essere generati tramite protocollo http, così da poter espandere le funzionalità del progetto in un secondo momento.

In prima battuta è stato considerato uno strumento, già integrato a livello di progetto, responsabile della interazione con dispositivi *smartphone*. Questa soluzione, denominata *MyServerWeb*, è capace di fornire dei servizi di base senza prevedere alcuna implementazione per la comunicazione WebSocket. Per garantire questo tipo di scambio di messaggi è stato considerato un progetto disponibile tra le risorse NuGet di Visual Studio, chiamato *SuperWebSocket* [1], in grado di gestire efficacemente tale protocollo.

Questa soluzione è stata scartata una volta considerati i suoi limiti intrinsechi. Era infatti necessario legare i servizi base di un server web ad una porta, ed i servizi di ascolto WebSocket ad una seconda porta. Una condizione del genere comporta gravi svantaggi dal punto di vista aziendale, in quanto è necessario mantenere operativi un numero maggiore di servizi rispetto ad una soluzione con porta singola.

Per ottenere performance migliori è stato considerato il progetto *websocket-sharp* [4], anch'esso distribuito con libero accesso ai file sorgenti grazie ad una licenza di tipo *MIT*, capace di legare un numero arbitrario di servizi ad una singola porta; fornendo anche protocollo WebSocket. Allo stesso tempo era in grado di riscrivere liberamente funzionalità di comunicazione http base, potendo quindi gestire con una singola porta tutti i tipi di richieste. Ulteriori analisi hanno portato alla luce un buon sistema di *threading pool* interno, in grado di mantenere basso il consumo di cpu in *idle*, mantenendo i riferimenti remoti in memoria principale per poi riassegnare un thread solo a quelli attivi.

5.4.3 Formato Json

Pensando al passaggio di dati fra i due *endpoint* dell'architettura (WenInterface e WebBroker) è importante fare una riflessione sul formato da adottare. Evitando infatti di passare stringhe non formattate sulla rete, incapsulate in semplici pacchetti IP, è possibile pensare a strutture dati più elaborate, magari in grado di interagire con oggetti C#.

A questo fine giunge in aiuto il formato di scambio di dati Json; questo formato è uno dei più utilizzati in comunicazioni web, ed è in grado di replicare lo scambio di oggetti veri e propri su di una connessione internet.

In sostanza viene generata una struttura, spedita previa la traduzione in formato Json [6]. Una volta catturato il messaggio da parte del ricevente, questo può tradurre nuovamente il messaggio in un oggetto, generato automaticamente secondo le specifiche comprese all'interno del messaggio stesso.

Il formato è facilmente sfruttabile grazie a primitive javascript all'interno della WebInterface. Allo stesso tempo però è importante trovare un'implementazione per .NET in grado di fornire un supporto valido alla traduzione.

In questo caso è stato analizzato un pacchetto installabile sotto forma di .dll tramite NuGet denominato Newtonsoft.Json [3]. Le primitive aggiunte da questa estensione garantiscono la capacità di traduzione messaggi, scritti in formato Json, nuovamente in un oggetto; in questo caso l'oggetto può essere un semplice contenitore, con tanti attributi quanti quelli posseduti dalla struttura Json di interesse.

Grazie a questo formato è possibile comunicare tra client web e server C# senza la necessità di entrare nei dettagli di messaggi scritti sul canale di comunicazione; è possibile rendere trasparente tutta la difficoltà a livello di rete, continuando a preoccuparsi solo del livello applicativo.

5.4.4 Tecnologie web

Una importante parte del progetto punta alla scrittura di interfacce grafiche che possano essere interpretabili su di un supporto web. Questo spinge a considerare quali siano le principali tecnologie in questo campo, capaci di garantire un paradigma di programmazione che sia il più possibile flessibile e riutilizzabile.

Essendo passati diversi anni dalla divulgazione ed implementazione di una rete come quella di Internet, alcune tecnologie specifiche sono diventate “standard de facto”. Dapprima tutti i linguaggi e protocolli di comunicazione, come lo standard a livelli ISO/OSI, ed in seguito anche i linguaggi di scrittura pagine web. Al momento sono presenti tre importanti linguaggi di base, sulla quale si basa la maggior parte della programmazione web, aggiungendo estensioni e strumenti di sviluppo. Stiamo parlando dei linguaggio html, css e javascript.

Il linguaggio HTML [12] (HyperText Markup Language) fornisce un modo per descrivere e dare un significato al contenuto di una pagina web. Non si interessa di dettagli legati ad impaginazione e visualizzazione, ma solamente al contenuto del foglio da trattare. In particolare le specifiche sfruttate ai fini di questa trattazione sono quelle contenute nel formato HTML5, sviluppato in una sua prima versione nel 2012, sostituendo lo standard precedente (HTML 4.01 fondato nel 1999) e tutti i relativi linguaggi di estensione (*XHTML* e *HTML DOM Level 2*). La creazione di questo nuovo linguaggio ha reso possibile la descrizione di contenuti sempre più complessi senza l’utilizzo di strumenti addizionali. Infine HTML5 è un linguaggio *cross - platform*, studiato specificamente per funzionare in maniera trasparente su qualunque supporto in grado di connettersi a internet (PC, Tablet, Smartphone o Smart TV).

Il linguaggio di specifica CSS [11] (Cascading Style Sheets o Fogli di stile), invece, punta proprio alla descrizione di tutti quei dettagli di impaginazione e visualizzazione, sfruttati da designer grafici, nello sviluppo di un sito web. Lo standard proposto in questa sede è denominato CSS3, il quale fornisce una

implementazione modulare delle vecchie specifiche con l'aggiunta di numerosi nuovi moduli. Questa versione è nata in concomitanza allo standard HTML5, completamente libero da dettagli di stile, ed è quindi in grado di modellare più efficacemente ogni aspetto grafico di una pagina web.

In conclusione il linguaggio di programmazione web javascript si interessa alla descrizione ed interpretazione di tutte quelle funzionalità aggiuntive che esulano il contenuto e la presentazione di una pagina.

Questo trio di tecnologie può essere trasversalmente visto come una implementazione lassa del pattern architetturale MVC (Model-View-Controller) [7], nel quale i tre ambiti di importanza in un'applicazione vengono divisi per dominarne la complessità e per delimitarne le responsabilità. Allo stesso modo vengono accoppiati fortemente tramite un sistema di comunicazione trasparente ma molto potente come quello ad eventi.

Nel nostro caso, abbiamo già analizzato in fase di analisi dei requisiti nel Cap. 5.1 la necessità ed il grande interesse aziendale per la tecnologia WebSocket. Tornando al parallelo con il pattern MVC possiamo identificare questa come un supporto di comunicazione fortemente *event-driven*, mentre i linguaggio html, css e javascript rappresentano rispettivamente *model*, *view* e *controller*.

Durante l'implementazione del progetto quindi è opportuno sfruttare queste tecnologie in modo da replicare una struttura fortemente interattiva ma modulare allo stesso tempo. Sia la traduzione di interfacce che la scrittura di un'applicazione principale dovranno avere queste come tecnologie target.

Difficoltà di integrazione

Nonostante la scelta delle tecnologie da sfruttare sia evidente a questo stadio del processo di sviluppo sono evidenti da subito alcuni importanti difficoltà.

La traduzione completa di una interfaccia grafica così complessa come quella di un gestionale ad uso aziendale può risultare complicata, specialmente su di un supporto che può sembrare non molto ricco come il web.

Nonostante queste incertezze un'analisi preliminare delle tecnologie ha riscontrato numerose funzionalità, anche grazie all'utilizzo di librerie jquery (javascript cross-browser), in grado di fornire diverse funzionalità avanzate in linguaggio

javascript. Nonostante numerosi strumenti di supporto però, alcuni oggetti più evoluti all'interno del framework aziendale risultano di difficile realizzazione, almeno in un prodotto a livello di prototipo. Mantenendo l'obiettivo del progetto sulla realizzazione di un prototipo funzionante, corredato di un sistema di comunicazione completo, è stato deciso di non implementare alcune delle funzionalità più evolute come griglie e planning eventi. L'azienda infatti possiede delle implementazioni già funzionanti di queste scritte in linguaggio web per altri progetti, quindi saranno facilmente inseribili in un secondo momento.

5.5 Implementazione

La capacità di generazione automatica di codice Java, offerta dal linguaggio di modellazione Contact, non ha potuto assistere lo svolgimento di questo progetto. La codifica è infatti avvenuta tramite ambiente Visual Studio 2012 in linguaggio C# come già anticipato.

Per comodità la trattazione seguente verrà divisa in sezioni, con l'intento di gestire la complessità del lavoro svolto. Dando una breve descrizione sommaria di ogni parte del dominio si potrà poi poi scendere nei dettagli delle sue parti più importanti.

WebInterface (resources)

Questa prima parte del dominio rappresenta un browser generico che pone delle richieste al nostro server web. Il progetto ha richiesto la scrittura in tecnologie web di una pagina in grado di fungere da client remoto. Questa pagina, implementata in html5 e css3, è capace di richiedere e mostrare altre sottopagine, che rappresentano l'interfaccia grafica dell'applicazione. Nonostante questa parte di progetto venga utilizzato da browser in maniera remota, è stata sviluppata in concomitanza del progetto principale.

index.html

```
<html lang="it"> <!--manifest="lib/index.mf">-->
<head>
```

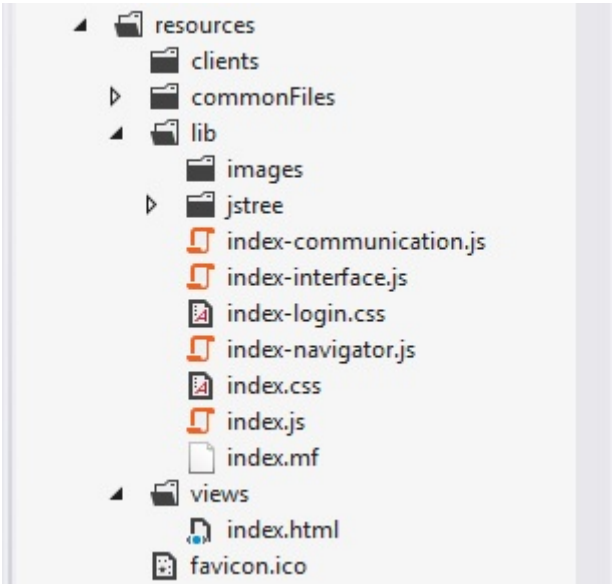

FIGURA 5.5: Interfaccia web.

```
    <meta http-equiv="Content-Type" content="text/html" charset="utf-8" />
    <title>Passepartout Web</title>
    <!-- Caricamento stili -->
    <link href="commonFiles/jquery-ui/jquery-ui.css" rel="styleSheet" />
    <link href="lib/index.css" rel="stylesheet" />
    <link href="lib/index-login.css" rel="stylesheet" />
    <!-- Caricamento scripts -->
    <script src="commonFiles/jquery/jquery-1.10.2.js" type="text/javascript"></script>
    <script src="commonFiles/jquery-ui/jquery-ui.js" type="text/javascript"></script>
    <script src="lib/jstree/jquery.jstree.js"></script>
    <script src="lib/index.js" type="text/javascript"></script>
    <script src="lib/index-communication.js" type="text/javascript"></script>
    <script src="lib/index-interface.js" type="text/javascript"></script>
    <script src="lib/index-navigator.js" type="text/javascript"></script>
</head>
<body>
    <div id="dialog-modal"></div>
    <div id="error-modal"></div>
    <!-- Barra laterale di menu -->
    <aside>
        <input id="menuButton" type="button" title="Apre il menu"
                onclick="OpenMenu(event)" />
        <input id="navigatorButton" type="button" title="Navigatore"
                onclick="OpenNavigator(event)" />
```

```
        <input id="contextButton" type="button" title="Contesto"
                onclick="OpenContext(event)" />
    </aside>
    <!-- Elemento overlay in cui caricare i vari menu -->
    <div id="overlay" hidden>
        <!-- Elemento di log sempre nascosto -->
        <div id="log" class="overlay" hidden></div>
        <div id="menu" class="overlay" hidden></div>
        <div id="tree" class="overlay" hidden></div>
        <div id="context" class="overlay" hidden></div>
    </div>
    <!-- Contenitore delle pagine -->
    <div id="wrapper" hidden>
        <div class="group">
            <div class="tabs">
                <ul>
                </ul>
            </div>
        </div>
    </div>
    <div id="hidden-elements" hidden>
        <!-- Form di login -->
        <div id="login" class="loginForm cf">
            <form name="login" accept-charset="utf-8">
                <ul>
                    <li>
                        <label for="username">Username</label>
                        <input type="text" name="username" placeholder="username"
                                required />
                    </li>
                    <li>
                        <label for="password">Password</label>
                        <input type="password" name="password" placeholder="password"
                                required />
                    </li>
                    <li>
                        <input type="submit" value="Login" />
                    </li>
                </ul>
```

```
            </form>
        </div>
    </div>
</body>
</html>
```

Si può notare fin da subito la grande semplicità del file di specifica della nostra schermata principale. Questa infatti, grazie alle specifiche html5, non è altro che un contenitore di informazioni, in questo caso minime se non assenti, che verrà poi riempito a *runtime*. Le interfacce grafiche verranno caricate in un secondo momento, mentre, per quanto riguarda l'aspetto grafico, possiamo fare affidamento sui fogli di stile.

Questa interfaccia principale possiede numerosi file javascript, uno dei quali fornisce la capacità di istruire una comunicazione di tipo WebSocket con il server, scambiando messaggi di diverso tipo; garantisce inoltre la capacità di gestire i messaggi di risposta da parte del *WebBroker*.

index-communication.js

```
var noSupportMessage = "Your browser cannot support WebSocket!";
var ws;

function ConnectWebSocket() {
    var support = "MozWebSocket" in window ? 'MozWebSocket' :
            ("WebSocket" in window ? 'WebSocket' : null);
    if (support == null) {
        AppendMessage("* " + noSupportMessage + "<br/>");
        return;
    }

    AppendMessage("* Connecting to server ..<br/>");
    ws = new window[support]("ws://localhost:7605/broker");

    ws.onopen = function () {
        OnOpen();
    };
    ws.onmessage = function (input) {
```

```
        OnMessage(input);
    };
    ws.onclose = function () {
        OnClose();
    };
}

function DisconnectWebSocket() {
    if (ws) {
        ws.close();
    }
}

//Operazioni da effettuare in apertura della webSocket
function OnOpen() {
    AppendMessage('* Connection open<br/>');
    SendScreenSize();
    Login();
}

//Operazioni da effettuare alla ricezione di un messaggio
function OnMessage(input) {
    try {
        var message = JSON.parse(input.data);
        AppendMessage("# Ricevuto messaggio:");
        AppendMessage("# - Type: " + message["Type"] + "<br />");
        ProcessMessage(message);
    }
    catch (e) {
        AppendMessage("# Error: " + e.message);
    }
}

//Operazioni da effettuare in chiusura della webSocket
function OnClose() {
    AppendMessage('* Connection closed<br/>');
}

function SendEvent(event) {
```

```
    var message = {
        "Type": "Event",
        "JsonEvent": event
    };
    Send(message);
}
//Spedisco un messaggio di tipo JsonMessage
function Send(message) {
    ws.send(JSON.stringify(message));
}
```

In questo breve file si evince immediatamente il protocollo di comunicazione WebSocket già analizzato nel Cap. 5.2, che propone una gestione ad eventi in grado di notificare l'infrastruttura in apertura, ricezione messaggio e chiusura della WebSocket. Da notare l'utilizzo del formato dati Json in spedizione di messaggi client nei confronti del *WebBroker*.

Mentre l'apertura fornisce informazioni di login la ricezione messaggio fa scattare il protocollo di comunicazione esistente, atto ad interpretare le risposte ricevute.

index.js

```
...

//Gestisco tutti i tipi di messaggi che possono essere spediti dal WebBroker
function ProcessMessage(message) {
    switch (message["Type"]) {
        case "Nothing":
            AppendMessage("Received empty message");
            UnlockClient();
            break;
        case "PageGeneration":
            GeneratePage(message["PageGeneration"]);
            break;
        case "PageUpdate":
            UpdatePage(message["PageUpdate"]);
            break;
        case "PageClose":
```

```
            ClosePage(message["PageClose"]);
            break;
        case "MessageBox":
            MessageBox(message["MessageBox"]);
            break;
        case "Login":
            LoginMessage(message["Login"]);
            break;
        default:
            AppendMessage(message["Content"]);
            break;
    }
}

...
```

Come si può notare vengono gestiti tutti i tipi di messaggio evidenziati in fase di analisi dei requisiti al Cap. 5.1.3. Ovviamente vengono gestiti anche i messaggi non presenti nella nostra tassonomia, ma solo al fine di redigere un file di log esaustivo in caso di anomalie. Analizzando ulteriormente questa parte di progetto, si può vedere come le risposte vengano elaborate in maniera indipendente dalle richieste ricevuto lato server.

```
...

//Gestisco la creazione di un qualunque tipo di pagina
//(Normale, ModaleClient, ModaleSessione)
function GeneratePage(pageGeneration) {
    var senderWindowId = pageGeneration["SenderWindowId"];
    var windowId = pageGeneration["WindowId"];
    var title = pageGeneration["Title"];
    var name = pageGeneration["Name"];
    var url = pageGeneration["Url"];
    pages[name] = windowId;
    switch (pageGeneration["Type"]) {
        case "ModaleClient":
            Popup(windowId, name, url, title, true);
            break;
```

```
        case "ModaleSessione":
            //Provo ad agganciare la sessione modale alla pagina che l'ha generata
            try {
                SelectPage(null, windowId).contentWindow.Popup(url, title, true);
            }
            //Se fallisco significa che la pagina si sta ancora caricando
            //ed aggancio la modale al suo evento loaded
            catch (e) {
                $("body").on("loaded", function (e, innerWindow) {
                    if (innerWindow === SelectPage(null, windowId).contentWindow)
                        innerWindow.Popup(url, title, true);
                });
            }
            break;
        default:
            contexts[windowId] = pageGeneration["Context"];
            AddPage(windowId, title, name, url);

            if (senderWindowId != -1 && senderWindowId != 0)
                SelectPage(null, senderWindowId).contentWindow.HideMask();
            break;
    }
}

//Aggiorno la pagina che ha scatenato un evento
function UpdatePage(pageUpdate) {
    var name = pageUpdate["Name"];
    var senderWindowId = pageUpdate["SenderWindowId"];
    var type = pageUpdate["Type"];

    //Se il messaggio è solo un activate devo attivare la pagina
    if (type == "Activate") {
        ActivatePage(name);
        SelectPage(null, senderWindowId).contentWindow.HideMask();
    }
    else {
        //Posso catturare il messaggio di update per tenere attivo
        //il client se senderWindowId == 0
        if (senderWindowId != 0) {
```

```
            var frame = SelectPage(name, senderWindowId);
            var dialog = $("iframe", dialogClient)[0];
            if (dialog != undefined && dialog != frame)
                dialogClient.dialog("close");
            frame.contentWindow.ManageAnswer(pageUpdate["Content"]);
        }
    }
}

...
```

Server

La parte in questione è rimasta inalterata rispetto al progetto originale. Per aumentare la compatibilità al massimo non sono state apportate modifiche al funzionamento logico del server. Tutte le chiamate ai suoi servizi vengono gestiti esattamente come le chiamate effettuate dal vecchio client .NET, escludendo però la parte di remotizzazione.

WebBroker

Questa ultima sezione si evidenzia come cuore del funzionamento del software. Risulta comodo poter dividere il modulo del *WebBroker* nelle sue sottoparti, così da poterle analizzare dettagliatamente. Nel corso della codifica sono stati generato alcuni namespace di comodo, per gestire la complessità del progetto e garantirne allo stesso tempo una buona manutenzione, anche in sede aziendale. Ai fini di questa analisi il termine cartella e namespace hanno concettualmente lo stesso significato lavorando in ambiente Visual Studio.

Si può notare una cartella contenente la documentazione con diagrammi delle classi e di flusso, nonché il nostro file Contact. Inoltre è facile distinguere le macro parti già emerse nell'analisi dei requisiti formate da *Generators* (*Generator*), *InternalBroker* (*Translator*) e *FilesContainer*. Sono state poi aggiunte alcune sottoparti come *JsonData* e *WebData* per perfezionare lo scambio di messaggi far client web e server C#. Infine è possibile constatare le classi sfruttate per gestire tutte le funzionalità di tipo web server, già discusse in sede di progetto. Scendendo più

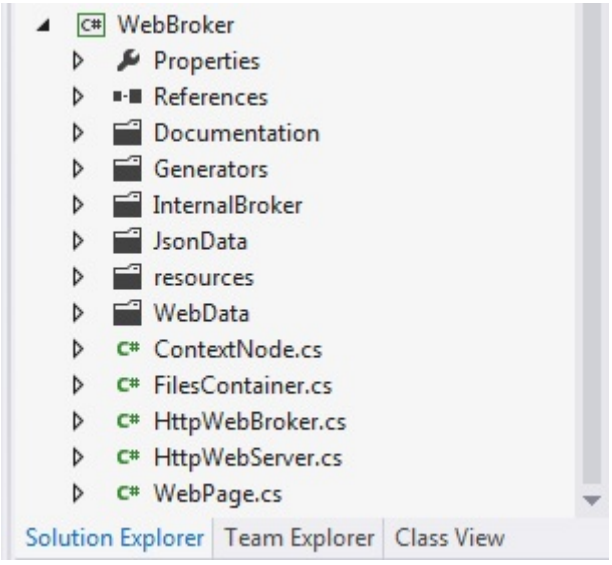

FIGURA 5.6: Progetto WebBroker.

nel dettaglio per ognuna delle sottoparti introdotte è possibile soffermarsi sul codice delle classi più interessanti, soprattutto dal punto di vista delle sfide implementative.

HttpWebServer.cs

```
...

public HttpWebServer(int port, string serverRoot, string clientsPath,
        string resourcesPath, string iconsPath)
{
    FilesContainer.LoadResources(serverRoot);

    mServer = new WebSocketSharp.Server.HttpServer(port);
    mServer.Log.Level = LogLevel.TRACE;

    mServer.RootPath = serverRoot;
    mServer.AddWebSocketService<HttpWebBroker>("/broker",
            () => new HttpWebBroker(clientsPath, resourcesPath, iconsPath));

    mServer.OnGet += (sender, e) =>
    {
        server_OnGet(e);
    };
}
```

```
 //Intercetto le chiamate di tipo get per restituire file
 //dinamici se necessario
private void server_OnGet(HttpRequestEventArgs eventArgs)
{
    var request = eventArgs.Request;
    var response = eventArgs.Response;
    var content = getContent(request.RawUrl);
    if (content != null)
    {
        if (request.RawUrl.Contains(".css"))
            response.ContentType = "text/css";
        else if (request.RawUrl.Contains(".js"))
            response.ContentType = "text/js";
        response.WriteContent(content);
        return;
    }
    response.StatusCode = (int)HttpStatusCode.NotFound;
}

...
```

Partendo dal meccanismo di comunicazione web, gestita dalla classe HttpWebServer, si può prendere nota di come questa rimanga in ascolto, come un servizio online, di qualunque richiesta da parte di browser remoti. Esistono due servizi principali: un servizio di comunicazione tramite WebSocket, gestita dal delegato *HttpWebBroker*, ed un servizio di richiesta pagina tramite protocollo *HTTP GET*, che viene gestita dal metodo server_OnGet, anch'esso gestito internamente come delegato tramite le API *websocket-sharp*.

HttpWebBroker.cs

```
...

private WebClient mClient;

#region OnOpen
```

```
protected override void OnOpen()
{
    Send(new WebMessage("Log", "Connection enstablished!"));
    mClient = new WebClient(mClientsPath, mResourcesPath, mIconsPath, this);
}
#endregion

#region OnMessage
protected override void OnMessage(MessageEventArgs e)
{
    string message = e.Data;
    try
    {
        JsonMessage jsonMessage = JsonConvert.
                DeserializeObject<JsonMessage>(message);
        mClient.ProcessMessage(jsonMessage);
    }
    catch (Exception)
    {
        Console.WriteLine("Invalid message received");
        Send(new WebMessage("ServerError", "Invalid message received."));
    }
}
#endregion

#region OnClose
protected override void OnClose(CloseEventArgs e)
{
    mClient.Logoff();
    Console.WriteLine("Deleted client: " + mClient.Id);
    Send(new WebMessage("Log", "Connection closed."));
}
#endregion

...
```

In questo breve scorcio di codice è possibile rilevare l'implementazione duale del protocollo WebSocket lato server, generato sempre grazie alle primitive fornite

dall'estensione discussa al Cap. 5.4.2. I messaggi vengono passati alla classe interna *WebClient*, la quale si interessa di gestire l'interazione con l'utente finale.

FilesContainer.cs

```
...

public static byte[] Get(string path)
{
    string fullPath = mRootPath + path;
    return mResourceFiles[fullPath];
}

public static byte[] GetClientFile(ulong id, string url)
{
    WebPage page = mClients[id].Find(element => element.Url.Equals(url));
    mClients[id].Remove(page);
    return GetBytes(page.Content);
}

...
```

La classe *FilesContainer*, infine, contiene la gestione dei file di risorse statiche e dinamiche da procurare al browser web, e quindi all'utente. Questo ovviamente vale anche per file generati al bisogno da parte del nostro traduttore interno al progetto; questi file, una volta scaricati lato browser, vengono prontamente cancellati dalla memoria interna del nostro programma.

Namespace JsonData

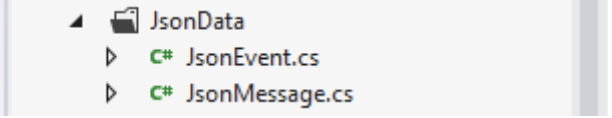

FIGURA 5.7: Cartella JsonData.

La cartella in esame possiede i tipi di messaggi che possono essere generati da parte del client web. Entrambi questi esempi sono una semplice rappresentazione

in oggetto C# del messaggio ricevuto tramite WebSocket, compresso grazie al protocollo Json.

JsonMessage.cs

```
namespace WebBroker.JsonData
{
    class JsonMessage
    {
        public string Type { get; set; }
        public string Data { get; set; }
        public JsonEvent JsonEvent { get; set; }

        public JsonMessage()
        {
        }
    }
}
```

JsonEvent.cs

```
namespace WebBroker.JsonData
{
    class JsonEvent
    {
        public string WindowId { get; set; }
        public string Type { get; set; }
        public string Name { get; set; }
        public bool Taggable { get; set; }
        public string Target { get; set; }
        public string Element { get; set; }
        public Object Value { get; set; }
        public string ValueType { get; set; }

        public JsonEvent()
        {
        }
```

```
    }
}
```

Si può notare come siano solo vuoti contenitori da sfruttare al ricevimento di un messaggio da parte del *WebBroker*, il quale si preoccupa di decomprimere il messaggio in un oggetto da poter utilizzare facilmente. In particolare la classe *JsonMessage* contiene al suo interno una rappresentazione ad oggetto di *JsonEvent*, in caso il messaggio sia del tipo appropriato.

Namespace WebData

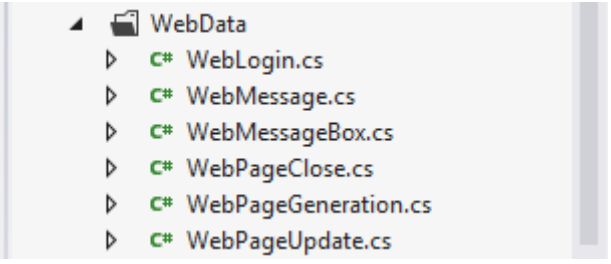

FIGURA 5.8: Cartella WebData.

Questo namespace custodisce tutte le strutture dati che codificano un messaggio di risposta da inviare al client web. La classe principale in questo caso è rappresentata da WebMessage, che racchiude tutte le possibili risposte generate dal server.

WebMessage.cs

```
namespace WebBroker.WebData
{
    public class WebMessage
    {
        public string Type { get; private set; }

        public WebLogin Login { get; set; }
        public WebPageGeneration PageGeneration { get; set; }
        public WebPageUpdate PageUpdate { get; set; }
        public WebMessageBox MessageBox { get; set; }
        public WebPageClose PageClose { get; set; }
```

```
        public string Content { get; private set; }

        ...
        }
}
```

Come si può notare funge semplicemente da involucro per differenti tipi di messaggi, che verranno poi trasmessi sulla connessione WebSocket, previa la codifica Json.

A loro volta tutte le variabili possono contenere dati diversi da passare al client web. I tipi di messaggi che possono essere così codificati sono differenti e rispecchiano la tassonomia vista in sede di progetto al capitolo 5.1.3. Inoltre abbiamo un attributo *Content* che può gestire eventuali messaggi generici ai fini di log.

WebPageGeneration.cs

```
namespace WebBroker.WebData
{
    public class WebPageGeneration
    {
        public string SenderWindowId { get; private set; }
        public string WindowId { get; private set; }
        public string Type { get; private set; }
        public string Title { get; private set; }
        public string Name { get; private set; }
        public string Url { get; private set; }
        public List<ContextNode> Context { get; set; }

        public WebPageGeneration(int senderWindowId, int windowId, string type,
                string title, string name, string url, List<ContextNode> context)
        {
            SenderWindowId = senderWindowId.ToString();
            WindowId = windowId.ToString();
            Type = type;
            Title = title;
            Name = name;
            Url = url;
            Context = context;
```

```
        }
    }
}
```

Analizzando più da vicino una di queste classi emerge come siano solamente un aggregato di attributi; questi rappresentano in maniera efficiente un messaggio del loro tipo.

Namespace InternalBroker

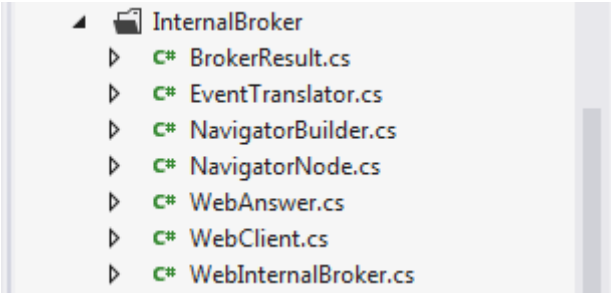

FIGURA 5.9: Cartella InternalBroker.

Il namespace in analisi fornisce tutte le parti relative all'interfacciamento vero e proprio fra la rappresentazione interna del cliente remoto ed il cliente remoto stesso, connesso tramite browser con protocollo WebSocket.

Si può osservare la class *WebClient* che permette la gestione del protocollo di comunicazione con il client in remoto, gestendo e smistando i suoi messaggi.

WebClient.cs

```
...

 public void ProcessMessage(JsonMessage jsonMessage)
        {
            string type = jsonMessage.Type;
            switch (type)
            {
                case "ScreenSize":
                    int width = Convert.ToInt32(jsonMessage.Data.Split('.')[0]);
```

```
                int height = Convert.ToInt32(jsonMessage.Data.Split('.')[1]);
                mWebInternalBroker.ScreenSize = new Coordinate(width, height);
                break;
            case "Login":
                //Recupero i dati dal campo Data qualunque sia il formato
                //in cui li voglio passare
                string username = jsonMessage.Data.Split('.')[0];
                string password = jsonMessage.Data.Split('.')[1];
                Login(username, password);
                break;
            case "Event":
                SubmitRequest(jsonMessage.JsonEvent);
                break;
        }
    }

private void SubmitRequest(JsonEvent jsonEvent)
    {
        int senderWindowId = Convert.ToInt32(jsonEvent.WindowId);

        if (jsonEvent.Taggable)
        {
            mQueuedEvents.Add(jsonEvent);
            //Genero una risposta vuota per sbloccare il client
            //a fronte di una richiesta accodabile
            BrokerResult result = new BrokerResult();
            result.Type = AnswerType.NOTHING;
            ManageAnswer(senderWindowId, result, -1);
        }
        else
        {
            Richiesta richiesta = new Richiesta(Id);

            //Carico tutti gli eventi in coda
            foreach (JsonEvent queuedEvent in mQueuedEvents)
                richiesta.AggiungiRichiesta(mEventTranslator.
                        Translate(queuedEvent));

            richiesta.AggiungiRichiesta(mEventTranslator.Translate(jsonEvent));
```

```
                long idRichiesta = richiesta.IdRichiesta;
                BrokerResult result = mWebInternalBroker.SubmitRequest(richiesta);
                ManageAnswer(senderWindowId, result, idRichiesta);
            }
        }

...
```

Questa classe demanda le operazioni di traduzione eventi ed interfacce grafiche alla classe *WebInternalBroker*, questa possiede la logica di comunicazione vera e propria. In questa sede vengono poi delegate le operazioni di traduzione interfacce grafiche al namespace *Generators* ed operazioni di traduzione eventi alla classe *EventTranslator*.

EventTranslator.cs

```
namespace WebBroker.InternalBroker
{
class EventTranslator
    {
        private Dictionary<string, Dictionary<string, Enum>> mEvents =
                new Dictionary<string, Dictionary<string, Enum>>();

        public EventTranslator()
        {
            InitializeEvents();
        }

        public RichiestaInterfaccia Translate(JsonEvent jsonEvent)
        {
            RichiestaInterfaccia richiesta = null;
            Dictionary<string, Enum> subEvents;

            subEvents = mEvents[jsonEvent.Type];
            Enum eventType = subEvents[jsonEvent.Name];

            switch (jsonEvent.Type)
```

```
{
    case "EventoScheda":
        richiesta = new RichiestaInterfaccia(eventType,
                jsonEvent.Target);
        break;
    case "EventoFinestraDettaglioEntita":
        if (jsonEvent.Element != null && jsonEvent.Value != null)
        {
            richiesta = new RichiestaInterfaccia(eventType,
                    jsonEvent.Target,
                        Enum.Parse(typeof(TipoCasellaContesto),
                            jsonEvent.Element), jsonEvent.Value);
        }
        else
            richiesta = new RichiestaInterfaccia(eventType,
                    jsonEvent.Target);
        break;
    case "EventoNavigatore":
        Albero.PosizioneNodo node = new Albero.PosizioneNodo();
        Albero albero = new Albero();
        albero.Ordina();
        node.cammino = GetPath(jsonEvent.Value);
        richiesta = new RichiestaInterfaccia(eventType,
                jsonEvent.Target, node);
        break;
    case "EventoCampo":
        if(jsonEvent.ValueType != null &&
                jsonEvent.ValueType.Equals("Lat"))
        {
            jsonEvent.Value = GetPath(jsonEvent.Value);
        }
        int value;
        if (jsonEvent.Value != null &&
                Int32.TryParse(jsonEvent.Value.ToString(), out value))
            jsonEvent.Value = value;
        if (jsonEvent.Element != null)
        {
            switch (jsonEvent.ValueType)
            {
```

```
                            case "ValoreCombo":
                                break;
                            default:
                                richiesta = new RichiestaInterfaccia(eventType,
                                        jsonEvent.Target, jsonEvent.Element,
                                            jsonEvent.Value);
                                break;
                        }
                    }
                    else
                        richiesta = new RichiestaInterfaccia(eventType,
                                jsonEvent.Target, jsonEvent.Value);
                    break;
            }
            return richiesta;
        }

...

        }
}
```

Qui è racchiusa tutta la logica di traduzione da messaggio di tipo evento web del client ad evento server vero e proprio. L'infrastruttura a sostegno di questa operazione è costituita da una serie di dizionari a cascata, questi sono capaci di effettuare una corrispondenza semi automatica di una stringa lato client verso un tipo di evento lato server. Le risposte date dal server vengono restituite secondo un formato denominato *BrokerResult*, che può possedere differenti operazioni svolte nel corso dell'elaborazione di una singola richiesta.

BrokerResult.cs

```
namespace WebBroker.InternalBroker
{
    class BrokerResult
    {
        private AnswerType mType;
```

```
        public AnswerType Type
        {
            get
            {
                return mType;
            }
            set
            {
                if (value > Type)
                    mType = value;
            }
        }
        public LoginState LoginState { get; set; }
        public List<NavigatorNode> Navigator { get; private set; }
        public List<WebPage> GeneratedPages { get; private set; }
        public List<WebPage> GeneratedScripts { get; private set; }
        public List<WebPage> GeneratedMessages { get; private set; }
        public List<WebPage> ClosedPages { get; private set; }

...
```

Si possono notare i differenti tipi di dati forniti dal *WebInternalBroker* in base alla risposta del server, dati che combaciano ancora una volta con il modello del dominio.

Infine le classi *NavigatorBuilder* e *NavigatorNode* offrono metodi di comodo per estrarre una rappresentazione del navigatore dell'applicazione. La classe *WebAnswer* invece funge da mero raccoglitore di messaggi (*WebMessage*) da spedire al client.

Namespace Generators

Il namespace di generazione file in formato web è più complesso degli altri, in quanto possiede molta della logica di traduzione. I punti di accesso alle sue funzionalità sono rappresentati dalle classi *FilesGenerator* e *JavascriptGenerator* mentre *Logger* è una semplice classe di comodo sfruttata in fase di debug.

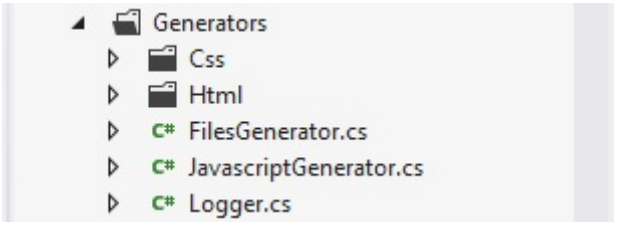

FIGURA 5.10: Cartella Generators.

Il primo punto di accesso, la classe *FilesGenerator*, include tutta la complessità di generazione file html e css per le interfacce grafiche, già presenti e generate sotto forma di dati intermedi.

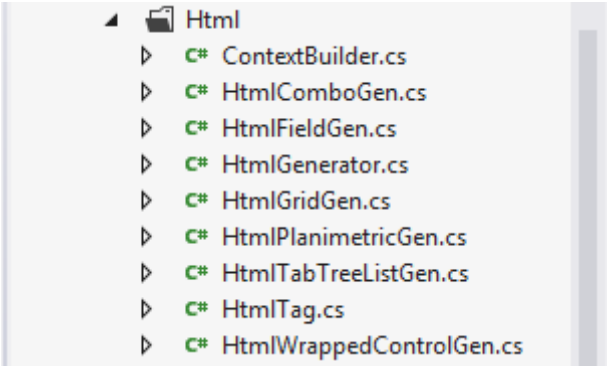

FIGURA 5.11: Cartella Generators.Html.

In questo caso il costrutto *partial* delle classi C# si è rivelato molto utile ai fini di manutenibilità del progetto, potendo dividere le sottoparti di generazione dei costrutti grafici di base da quelli più specifici e complicati.

HtmlGenerator.cs

```
namespace WebBroker.Generators.Html
{
    partial class HtmlGenerator
    {

        public WebPage Generate(ComDocumento document, string name)
        {
            mHtmlWriter = new StringWriter();
            mTitle = "";

            using (HtmlTextWriter writer = new HtmlTextWriter(mHtmlWriter))
```

```
    {
        writer.Indent = 0;

        //<!DOCTYPE html>
        writer.AddAttribute("html", null);
        writer.RenderBeginTag("!DOCTYPE");
        writer.Indent = 0;

        //<html lang="it">
        writer.AddAttribute("lang", "it");
        writer.RenderBeginTag("html");
        writer.Indent = 0;

        GenerateHeader(document, writer, name);
        GenerateBody(document, writer, name);

        //</html>
        writer.WriteEndTag("html");
    }

    string htmlFile = mHtmlWriter.ToString();
    mHtmlWriter.Close();
    ContextBuilder builder = new ContextBuilder();
    List<ContextNode> context = builder.Build(document);

    return new WebPage(GetPageId(document), GetPageType(document),
            mTitle, name, name + ".html", htmlFile, context);
}

private void GenerateHeader(ComDocumento document,
        HtmlTextWriter writer, string name)
...
private void GenerateBody(ComDocumento document, HtmlTextWriter writer,
        string name)
...
private void GenerateGroup(ComRaggruppamento group, HtmlTextWriter writer)
...
private void GenerateBoard(ComScheda board, string boardName,
        HtmlTextWriter writer)
```

```
        ...
        private void GenerateField(ComControllo field, string boardName,
                HtmlTextWriter writer)
        ...
        }
}
```

Si può evincere come sia stato sfruttato il costrutto *partial* come già accennato in precedenza.

HtmlFieldGen.cs

```
namespace WebBroker.Generators.Html
{
    partial class HtmlGenerator
    {
        private void GenerateNormalField(ComControllo field, string boardName,
                HtmlTextWriter writer, TipoGui guiType)
        {
            Dictionary<string, string> attributes =
                    new Dictionary<string, string>();

            string id = boardName + "-" + field.Proprieta[ProprietaCampo.NOME];
            string title = (string)field.Proprieta[ProprietaCampo.TITOLO];
            string tabIndex = Convert.ToString(field.Proprieta[
                    ProprietaCampo.ORDINE]);

            HtmlTag tag = mControls[guiType];

            //Aggiungo attributi generici
            attributes.Add("id", id);
            if (title != null)
                attributes.Add("title", title);
            attributes.Add("tabindex", tabIndex);

            //Aggiungo tutti gli attributi dipendenti dal tag html
            GenerateTagAttributes(tag, attributes);
            //Aggiungo tutti gli attributi dipendenti dal tipo di gui
```

```
            GenerateGuiAttributes(field, guiType, attributes);
            //Inserisco l'attributo per il valore a seconda del suo tipo
            GenerateValue(field, attributes);

            //Inserisco tutti gli attributi nel tag html
            foreach (KeyValuePair<string, string> attribute in attributes)
            {
                writer.AddAttribute(attribute.Key, attribute.Value);
            }

            //<element>
            writer.RenderBeginTag(tag.Name);

            //Inserisco il titolo dell'elemento a seconda del tipo di controllo
            switch (tag.Name)
            {
                case "canvas":
                case "input":
                case "table":
                    break;
                default:
                    writer.Write(title);
                    break;
            }

            //</element>
            writer.RenderEndTag();
            writer.WriteLine();

        }
    }
}
```

Qui si può vedere un esempio pratico di generazione di codice html, che viene sfruttato per tutti gli elementi di base che non necessitano di implementazioni particolari.

Anche in questo caso la scrittura di classi parziali aiuta fortemente a dominare la complessità del progetto e ad aumentare il livello di manutenibilità, circoscrivendo

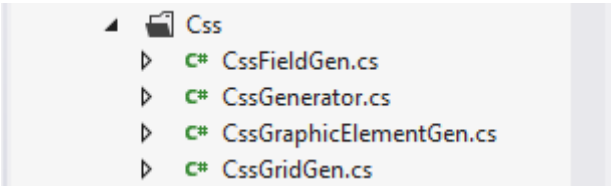

FIGURA 5.12: Cartella Generators.Css.

la provenienza di problemi in caso di malfunzionamento. Il paradigma utilizzato è molto simile a quello già studiato per il generatore di codice html mentre i dettagli sono differenti.

CssFiledGen.cs

```
...

private void WriteCssField(string name, ComControllo field,
        Coordinate positionMargin)
{
    Coordinate position = (Coordinate)field.Proprieta[
            ProprietaCampo.POSIZIONE];
    Coordinate dimension = (Coordinate)field.Proprieta[
            ProprietaCampo.DIMENSIONE];
    TipoAncoraggioCampo anchor = (TipoAncoraggioCampo)field.Proprieta[
            ProprietaCampo.ANCORAGGIO];
    Coordinate adjustedPosition = AdjustCoordinates(SumMargin(position,
            positionMargin));
    Coordinate adjustedDimension = AdjustCoordinates(dimension);

    BeginCssElement(name);
    WriteCssAttribute("position", "absolute");
    WriteCssFieldAnchor(adjustedPosition, adjustedDimension, anchor);
    EndCssElement();
}

private void WriteCssFieldAnchor(Coordinate position, Coordinate dimension,
        TipoAncoraggioCampo anchor)
{
    bool left = false;
```

```
bool top = false;
bool right = false;
bool bottom = false;

if (anchor == TipoAncoraggioCampo.Default)
{
    WriteCssAttribute("left", position.x);
    WriteCssAttribute("top", position.y);
    WriteCssAttribute("width", dimension.x);
    WriteCssAttribute("height", dimension.y);
}
else
{

    if ((TipoAncoraggioCampo.Sinistra & (TipoAncoraggioCampo)anchor) ==
            TipoAncoraggioCampo.Sinistra)
        left = true;
    if ((TipoAncoraggioCampo.Alto & (TipoAncoraggioCampo)anchor) ==
            TipoAncoraggioCampo.Alto)
        top = true;
    if ((TipoAncoraggioCampo.Destra & (TipoAncoraggioCampo)anchor) ==
            TipoAncoraggioCampo.Destra)
        right = true;
    if ((TipoAncoraggioCampo.Basso & (TipoAncoraggioCampo)anchor) ==
            TipoAncoraggioCampo.Basso)
        bottom = true;
    if (left)
    {
        WriteCssAttribute("left", position.x);
        if (right)
            WriteCssAttribute("right", mWindowDimensions.x -
                    position.x -  dimension.x);
    }
    else
        WriteCssAttribute("right", mWindowDimensions.x -
                position.x -  dimension.x);

    if (top)
    {
```

```
                WriteCssAttribute("top", position.y);
                if (bottom)
                    WriteCssAttribute("bottom", mWindowDimensions.y -
                            position.y -  dimension.y);
            }
            else
                WriteCssAttribute("bottom", mWindowDimensions.y -
                        position.y -  dimension.y);

            int leftPos = position.x;
            int rightPos = mWindowDimensions.x - position.x - dimension.x;
            int topPos = position.y;
            int bottomPos = mWindowDimensions.y - position.y - dimension.y;

            if (left && right)
                WriteCssAttribute("width", mScreenSize.x - leftPos - rightPos);
            if (top && bottom)
                WriteCssAttribute("height", mScreenSize.y - topPos - bottomPos);
            if (!(left && right))
                WriteCssAttribute("width", dimension.x);
            if (!(top && bottom))
                WriteCssAttribute("height", dimension.y);
        }
    }

    ...
```

Si può notare come nei file css vengano espressamente assegnati solamente attributi relativi al posizionamento degli elementi, questi possono così replicare un effetto di ancoraggio all'interno delle finestre web. Questi file devono essere coadiuvati dalla presenza di altri dettagli grafici, che sono però inseriti all'interno di una sottoparte della cartelle *resources* come file statici di risorse, da fornire al client una volta richiesti e legati alle pagine come fogli di stile.

generic-script.js

É importante notare come le pagine generate automaticamente da parte del server vengano gestite lato client come degli inner frames (*tag iFrame*) e pertanto possano

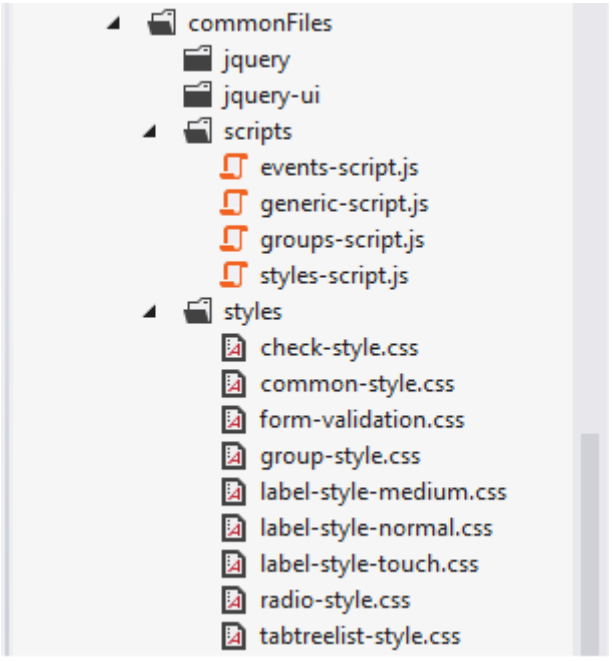

FIGURA 5.13: Risorse statiche comuni.

comunicare solo attraverso specifiche chiamate a funzione. Di particolare interesse il meccanismo che solleva un evento, da una pagina interna ad un frame verso la nostra *index.js*, e quello di evasione risposta di aggiornamento pagina interna, a partire dalla nostra applicazione.

```
...

function InputChanged(input) {
    var input = $(input);
    var id = input.attr('id').replace("controlWrapper-", "");
    var values = id.split("-");
    var target = values[0].replace("_", ",#");
    var element = values[1];
    var value = input.prop('checked');
    var event = {
        "WindowId": windowId,
        "Type": "EventoCampo",
        "Name": "ValoreModificato",
        "Taggable": false,
        "Target": target,
        "Element": element,
        "Value": value,
```

```
        "ValueType": "System.Boolean",
    };
    SendEvent(event);
}

function SendEvent(event) {
    ShowMask();
    var message = {
        "Type": "Event",
        "JsonEvent": event
    };
    parent.Send(message);
}
...
```

In questa prima parte possiamo vedere il meccanismo che permette di scatenare eventi, il quale genera un evento del tipo giusto e lo spedisce tramite una primitiva presente nel suo *parent*. Il file javascript contenente questa funzione risulta essere *index-communication.js*, che provvederà alla spedizione dell'evento in formato Json attraverso la comunicazione WebSocket aperta.

```
...

//Elaboro una richiesta javascript di aggiornamento pagina
function ManageAnswer(script) {
    eval(script);
    HideMask();
}

...
```

In questa seconda parte si evince quanto sia semplice poter mettere in funzione uno script javascript, scritto correttamente, una volta identificata la pagina alla quale è indirizzato.

Infine soffermandosi sulla generazione automatica di codice javascript se ne può descrivere il comportamento generale, questo viene fatto scattare da tutte quelle risposte che vogliono solo modificare interfacce grafiche (pagine web) già esistenti.

Anche in questo caso il meccanismo di aggiornamento delle pagine, già generate e visualizzate lato client, avviene in maniera semplice. Il file javascript viene scritto, includendo al suo interno tutte le variazioni a livello html e css, in modo da incapsulare ogni possibile modifica effettuata all'interfaccia. Questo semplice file viene incluso in un messaggio web (*WebMessage*) ed, una volta ricevuto lato client, viene semplicemente eseguito come script esterno, applicando tutte le modifiche suggerite dal server.

In conclusione l'implementazione data al prototipo rispecchia in pieno il funzionamento descritto in fase di progetto e ci permette una buona manutenibilità del codice. Allo stesso tempo la strutturazione su più livelli delle nostre classi ci permette di aggiungere facilmente delle nuove funzionalità, mantenendo la complessità di traduzione in poche specifiche classi.

5.6 Test

Il progetto non presenta una sua suite di test e pertanto non è possibile accertarne le performance a livello aziendale. D'altro canto il prototipo assemblato nel corso del lavoro è ancora ad uno stato embrionale. Nonostante alcuni problemi legati ad una traduzione tecnologica da strumenti C# a strumenti web l'infrastruttura di comunicazione, definita in sede di analisi, è stata rigorosamente rispettata ed implementata completamente. Questo ha portato alla creazione di un prototipo funzionante nel limite dell sue capacità di base.

Nel proseguire con questo progetto sarebbe opportuno preparare un set di test adeguato. Questi potrebbero essere eseguiti man mano che si aggiungono funzionalità, così da accertare il corretto funzionamento della struttura implementata.

5.7 Installazione

Il prodotto finito risulta ancora un prototipo; pertanto la fase di installazione e di test viene fatta tramite piattaforma Visual Studio, sia in locale che in remoto. É possibile vedere alcune immagini del software in funzione ed elencare le sue caratteristiche principali durante lo svolgimento di una normale sessione.

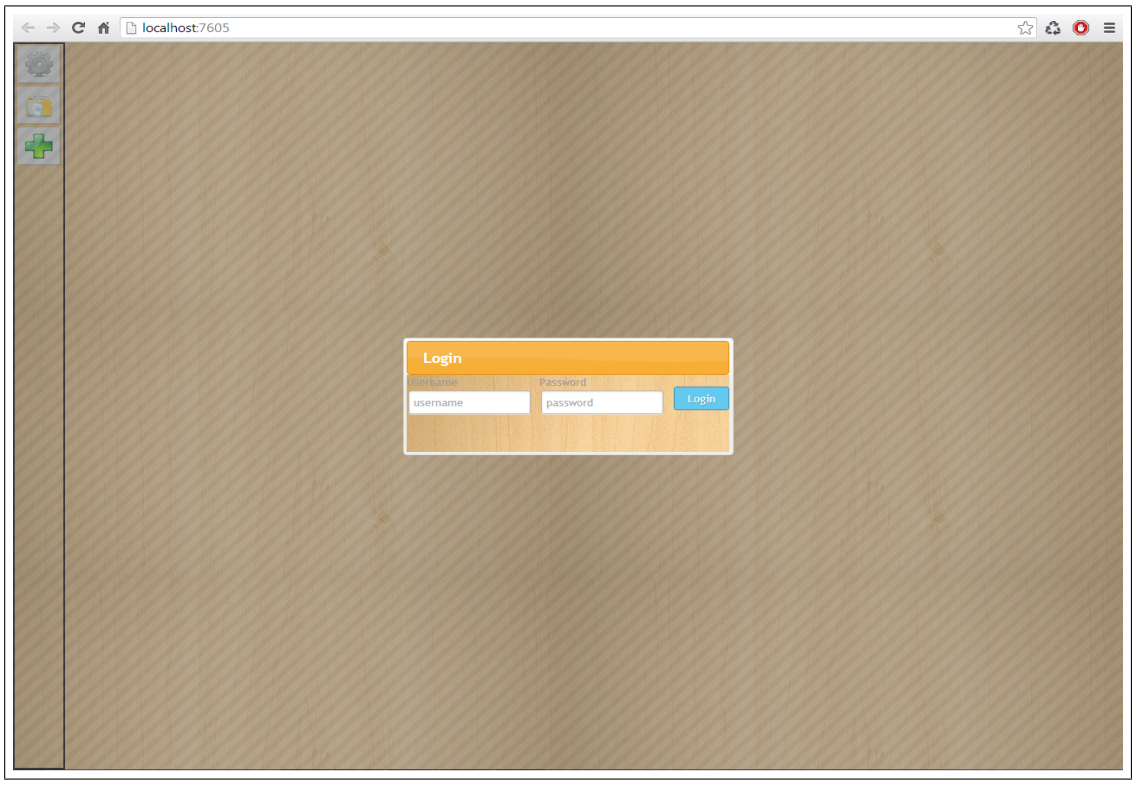

FIGURA 5.14: Schermanta di login.

Subito dopo la richiesta di comunicazione da parte di un browser, in questo caso chrome, viene visualizzata la schermata di login, che blocca l'utente fino all'inserimento di credenziali corrette per l'autenticazione (Fig. 5.14).

Una volta effettuato l'accesso viene visualizzata la schermata principale dell'applicativo. Questo possiede un menu interattivo a forma di bottoniera e tre semplici menu sul lato destro dello schermo. Con questi è possibile effettuare operazioni semplici oppure navigare all'interno di tutte le funzionalità del programma. É possibile anche richiamare funzionalità contestuali relative alle pagine visualizzate (Fig. 5.15).

Il programma è in grado di gestire: sia le operazioni più semplici di apertura di pagina, sia le operazioni più complesse con tempo di attesa e risposta asincrona. Da notare la riproduzione fedele delle capacità di ancoraggio utilizzate in ambiente .NET.

Appare evidente in Fig. 5.16 come venga gestita la creazione di una pagina sullo sfondo e di una pagina modale associata alla stessa. Una volta completata l'operazione lato server, e spedita la relativa risposta, viene aggiornata la pagina sullo sfondo e viene chiusa automaticamente la modale relativa di attesa.

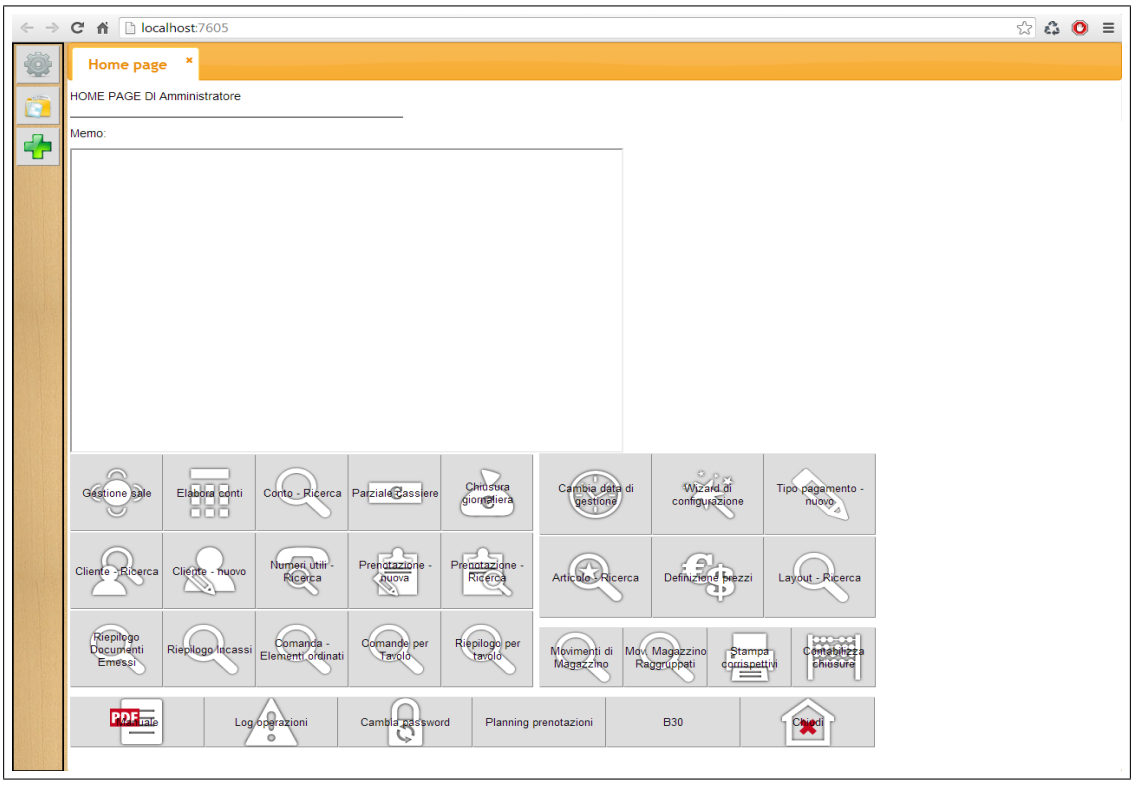

FIGURA 5.15: Schermanta principale.

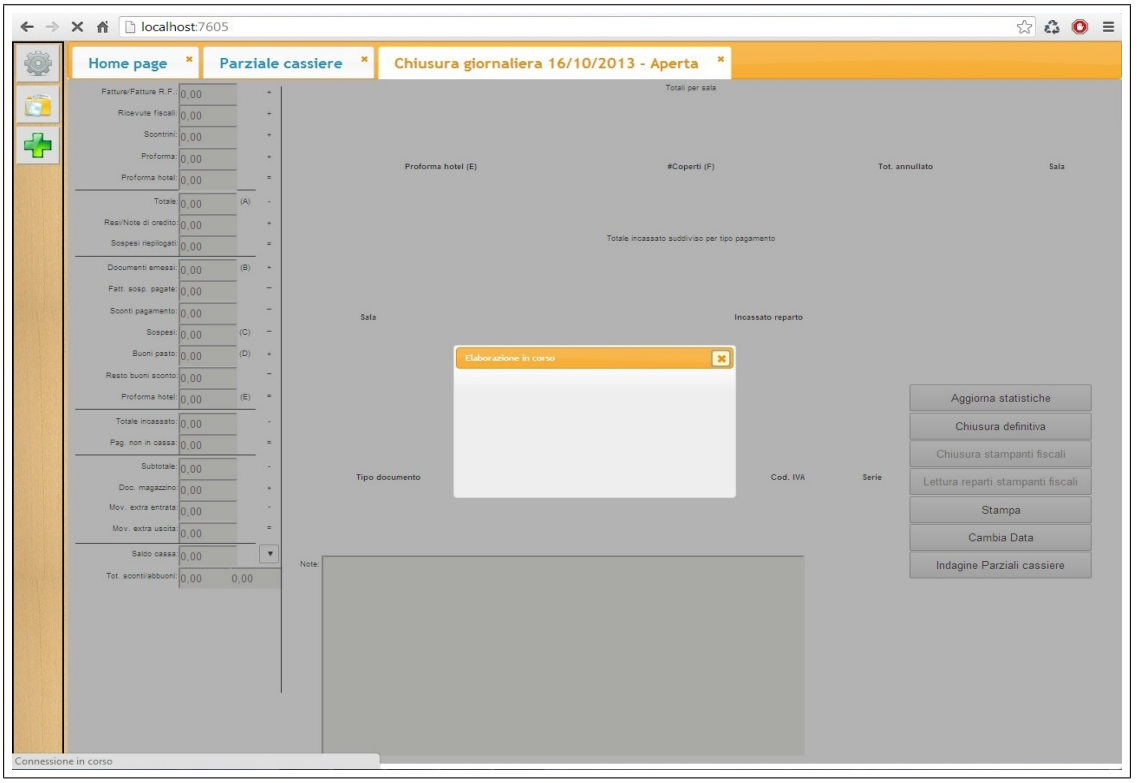

FIGURA 5.16: Nuova pagina con elaborazione.

Oltre alle funzionalità di generazione pagina possiamo notare in Fig. 5.17 la capacità di aggiornamento di quelle schermate già visualizzate con le quali si può interagire liberamente.

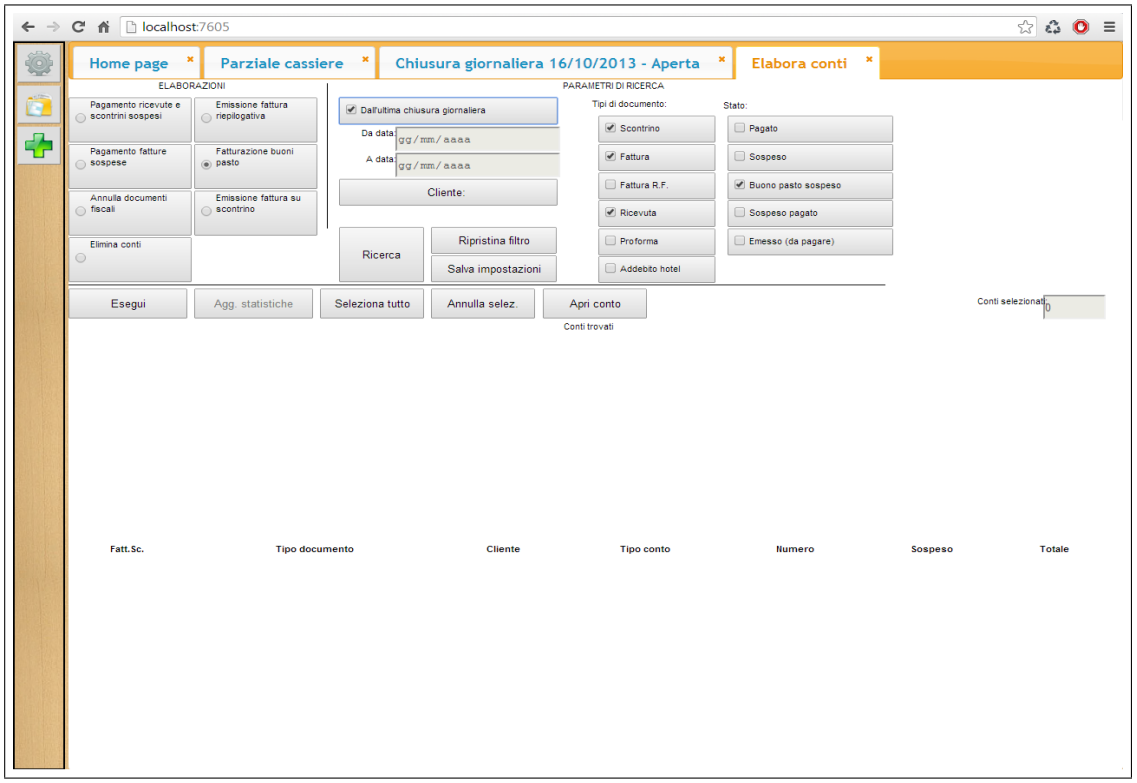

FIGURA 5.17: Aggiornamento pagina in relazione ad interazione utente.

In questo caso l'interfaccia viene aggiornata in maniera automatica grazie all'interpretazione di script javascript di risposta all'interazione utente.

Infine è possibile esaminare anche elaborazioni più complesse come la ricerca di un cliente, la quale genera una nuova interfaccia modale (Fig. 5.18).

Da questa situazione è infatti possibile aprire una pagina di generazione nuovo cliente, chiudendo la modale e sbloccando la pagina che ha generato la prima richiesta (Fig. 5.18).

In conclusione il modello di comunicazione rispetta in pieno la flessibilità del software di partenza. Inoltre la logica di creazione ed aggiornamento pagine non è stata minimamente alterata sfruttando il server presente.

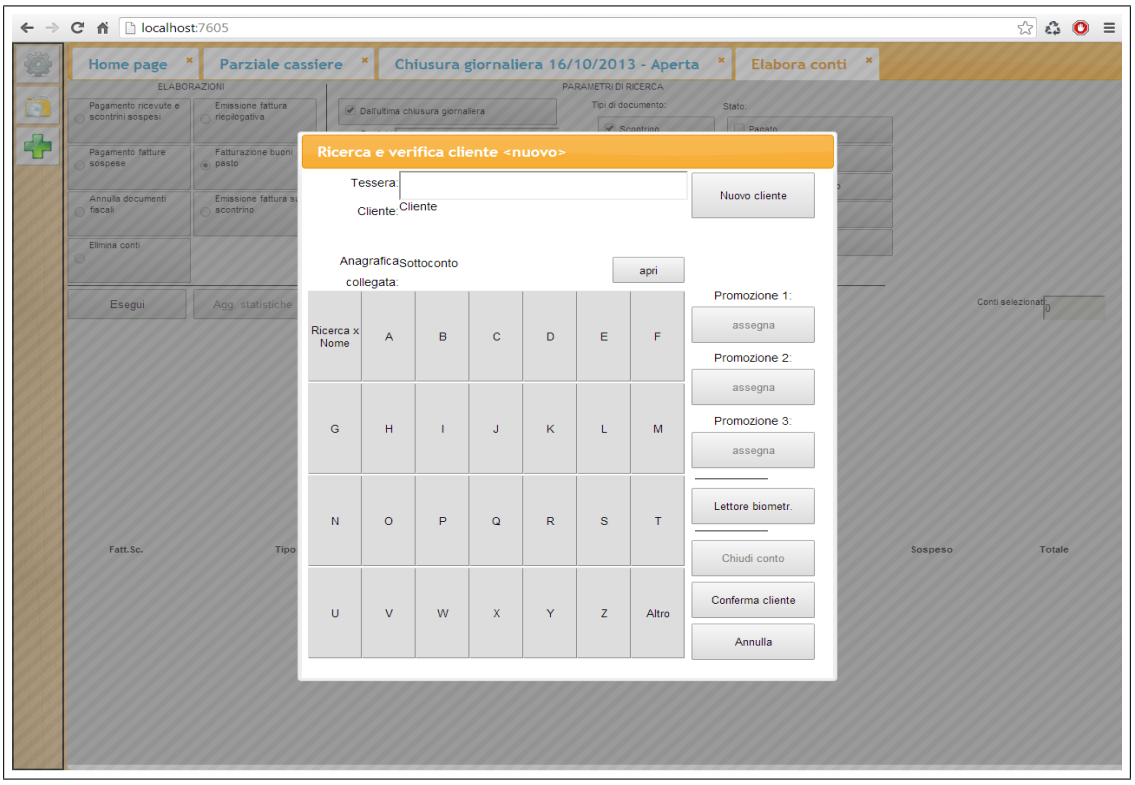

FIGURA 5.18: Ricerca di un cliente esistente.

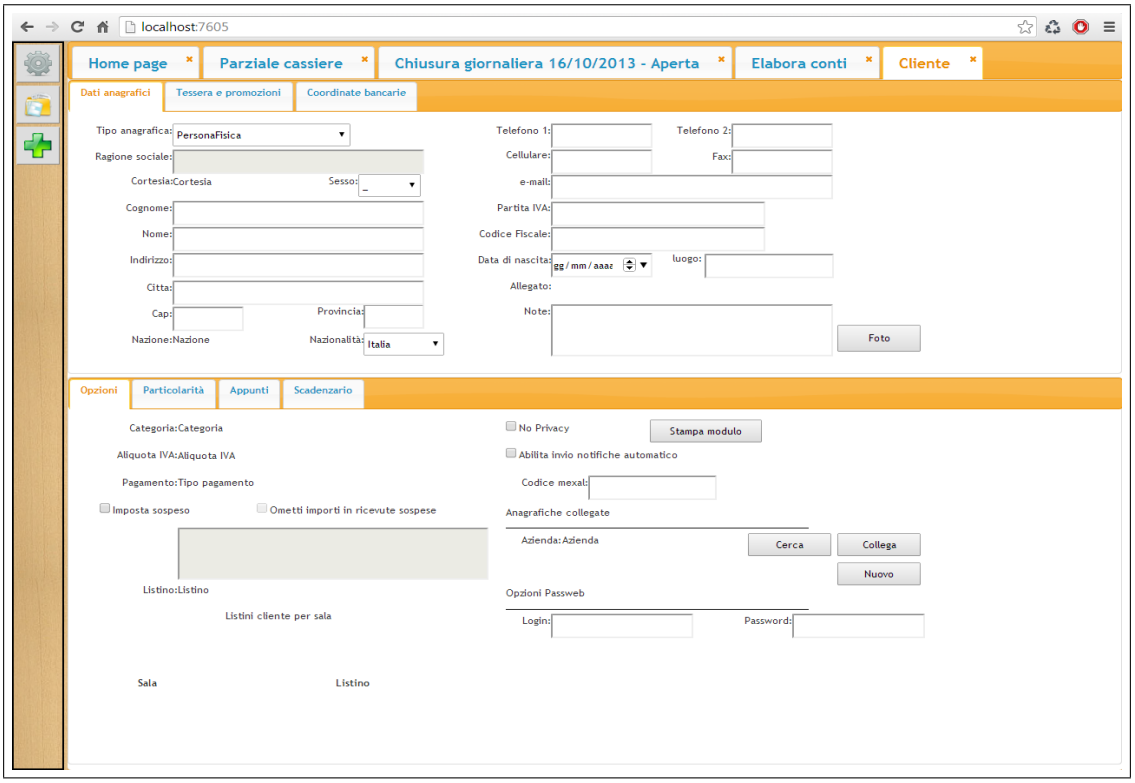

FIGURA 5.19: Generazione di un nuovo cliente.

5.8 Manutenzione

Il progetto ha generato con successo un prototipo funzionante a livello architetturale garantendo le funzionalità chiave del software di partenza. Le operazioni svolte e le fasi affrontante per ottenere questo risultato hanno definito uno sforzo aziendale in termini di tempo e denaro che non si vorrebbe dover ripetere per ogni gestionale.

Il prototipo ci permette di implementare facilmente capacità aggiuntive, previa la loro traduzione in linguaggio web (html, css, js), applicando una breve modifica alle classi di traduzione esistenti.

Nel portare avanti questo progetto, uno degli obiettivi principali, sarebbe quello di integrare le tecnologie web già implementate in azienda all'interno di questo prototipo.

Allo stesso modo, pensando in maniera trasversale rispetto al progetto stesso, risulta di grande interesse la generalizzazione di un metodo di reingegnerizzazione automatizzabile. Si vorrebe poter modernizzare le vecchie tecnologie, siano esse scritte in C# per .NET o in qualunque altro linguaggio, tramite un processo semi automatico, il quale possa essere replicato con i minori costi possibile per l'azienda.

Capitolo 6

Una soluzione progettuale

Nel seguente capitolo si fa un passo avanti rispetto al lavoro di implementazione software, tentando di risolvere alcune problematiche in questo tipo di approccio. In particolare risulta molto interessante la capacità di replicare il processo di reingegnerizzazione, sia per differenti software gestionali, sia per differenti tecnologie di uscita.

Analizzando le varie fasi che hanno portato al completamento di un primo prototipo, possiamo stilare un piccolo elenco delle tempistiche legate ad ogni sottoparte. Il progetto è stato suddivisto in moduli i quali hanno richiesto un impiego di risorse differente ed hanno quindi un peso diverso all'interno del processo di sviluppo stesso:

- `Traduzione eventi`

  Traduzione degli eventi scatenati dall'interfaccia in tecnologia web verso eventi lato server. Ha richiesto circa un mese di tempo.

- `Traduzione costrutti grafici`

  Traduzione degli elementi grafici presenti nel client .NET in parti equivalenti scritte in tecnologia web. Ha richiesto circa un mese di tempo.

- `Implementazione messaggi`

  Implementazione del protocollo di scambio di messaggi (Request / Response) e di richieste previste lato server. Ha richiesto circa due mesi di tempo.

- `Progettazione interfaccia web`

  Progettazione dell'interfaccia web del nuovo client. Ha richiesto circa un mese di tempo.

Nonostante il prodotto finito non sia una implementazione completa di un client in tecnologia web, il tempo di cinque mesi impiegato nella sua realizzazione può dare una visione primitiva della ripartizione di risorse.

Questa statistica ci viene in aiuto, in quanto pone l'accento sulla quantità enorme di tempo che deve essere impiegata in procedure ripetitive di traduzione di eventi e implementazione di messaggi. Allo stesso modo si può notare come si debba impegnare un lasso di tempo notevole, sia nella generazione di una traduzione di costrutti grafici, sia nella progettazione dello scambio di messaggi, sfruttando una differente tecnologia di output.

Volendo evitare di spendere tanto tempo in azioni meccaniche, una volta sviluppato un primo progetto, sono state analizzate tecnologie e strategie per ridurre il più possibile queste tempistiche.

6.1 Metamodello

Tentando quindi di adattare la soluzione web sviluppata ad un diverso tipo di gestionale o, più in generale, ad una diversa applicazione o interfaccia grafica, si dovrà sicuramente impiegare tempo e risorse in una nuova analisi e progettazione.

La traduzione di eventi da formato client a formato server, per esempio, e la traduzione di una risposta server in pagine web, sono operazioni che richiedono, come la generazione di parti grafiche, uno sforzo iniziale di implementazione non trascurabile. Per quanto riguarda la traduzione di risposte da parte del server in linguaggio web, la presenza di sempre nuove funzionalità da esporre lato client, comporta un ulteriore costo di ampliamento. Al contrario una diversa rappresentazione delle risposte da parte del server, implica una completa riscrittura del meccanismo di traduzione tra i due formati. Passando alla traduzione di eventi generati lato web in eventi codificati dal server, nonostante si possa facilmente garantire l'uniformità di eventi lanciati da diversi client, non è possibile conoscere a priori la codifica in sistemi diversi. In sintesi rimane un costo piuttosto elevato nella

generalizzazione della soluzione proposta, questo rende il lavoro meno appetibile dal punto di vista progettuale.

Un altro spunto interessante di analisi è quello che porta ad interessarsi all'interfacciamento verso differenti tecnologie di uscita, nel caso in cui quelle utilizzate diventassero obsolete. Cercando di mantenere un approccio fortemente legato alla "business logic" dell'applicazione, sono stati analizzati gli eventuali costi e tempi necessari alla traduzione di un software in un differente "output" tecnologico. In questo caso, non basterebbe solo ricalibrare i tipi di richieste scambiate fra le parti del software in gioco (client e server), ma sarebbe necessaria anche una complessa operazione di modifica della generazione di pagine.

Per ridurre i tempi sensibilmente e raccogliere tutti queste variabili decisionali effimere, che non possono essere considerate in un progetto classico, si fa ricorso al concetto di metamodello. Questo strumento è in grado di definire le regole secondo le quali descrivere un progetto specifico. In sostanza ci permette di gettare le basi, con le quali poter descrivere un qualunque tipo di interfacciamento software, sia esso gestionale o meno, tramite alcune sue qualità.

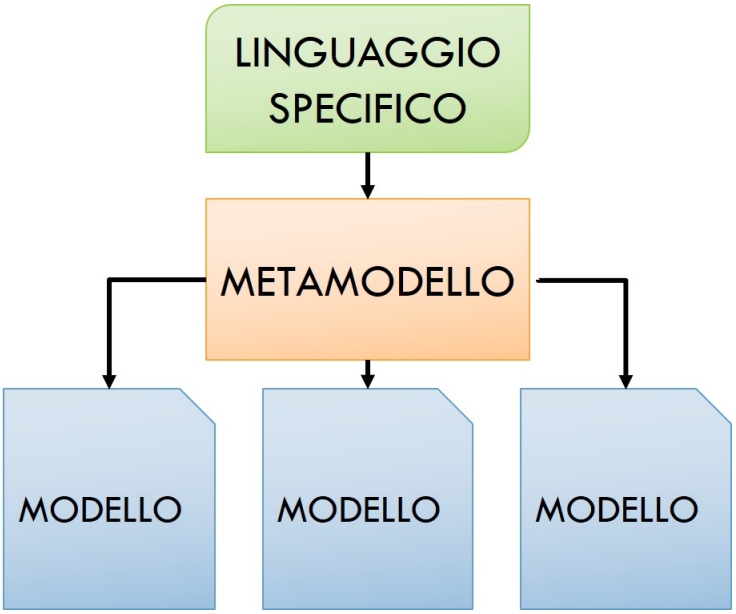

FIGURA 6.1: Metamodello e linguaggio specifico.

Sfruttando un concetto progettuale così evoluto è possibile utilizzare le stesse regole per definire una pletora di interfacciamenti software, anche sostanzialmente differenti. Definendo infatti poche regole di base, in grado di specificare le funzionalità esposte lato server, si può imbastire una infrastruttura di base in maniera automatica fra software esistente ed un client web.

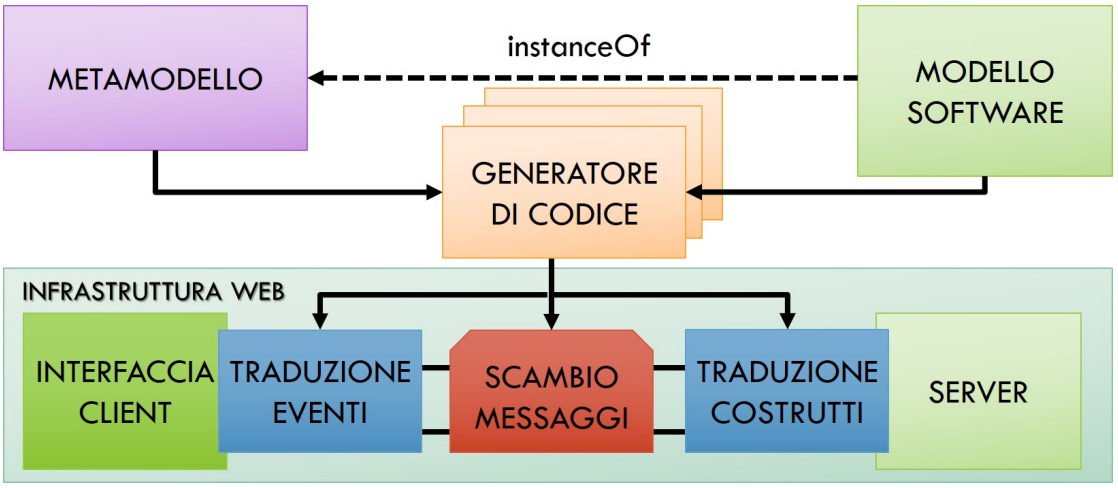

FIGURA 6.2: Riduzione tempi tramite metamodellazione.

Grazie ad una prima implementazione di metamodello, adatto a questo scopo, è possibile abbattere facilmente gran parte dei costi legati all'implementazione dei messaggi, il quale rappresenta circa il venti percento del tempo totale impiegato.

6.1.1 Linguaggio specifico

Un linguaggio definito con caratteristiche peculiari in relazione ad un determinato dominio viene denominato "Domain Specific Language" (DSL) [5]. Questo tipo di linguaggio può ritornare molto utile per definire le specifiche generali, alle quali si devono adeguare tutti i modelli di un qualsivoglia software gestionale.

Queste regole grammaticali vengono in aiuto nella definizione di un cosiddetto metamodello, specificato tramite un linguaggio di questo tipo, da sfruttare per definire le specifiche del modello in questione, dichiarando quali siano i tipi di richieste che un server specifico debba soddisfare.

In primo luogo quindi è possibile definire un metamodello tramite un DSL, il quale definisce le generalità di un qualunque gestionale da adattare ad una differente tecnologia. In questo caso si avrebbe sempre una interazione di tipo Request / Response, come già puntualizzato più volte in fase di progetto al Cap. 5.4, mentre ogni differente applicativo avrebbe una serie unica di richieste da esporre lato server.

In pratica è possibile definire tutte le operazioni che vengono esposte da un server generico e di conseguenza tutti i tipi di richieste che possono essere spedite tra

client e server. Ognuna di queste funzionalità è corredata da un evento scatenato lato client, la sua controparte server ed una struttura dati adatta a scambiare le informazioni necessarie fra le parti.

Questa caratterizzazione, specifica per ogni tipo di software interessato ad un aggiornamento tecnologico di sorta, viene definita sotto forma di modello specifico, derivato direttamente da un unico metamodello capace di adattarsi ad un qualsivoglia utilizzo. La capacità metamorfica di questa tecnica ci permette di definire un unico insieme di regole di linguaggio, fornendo un progetto di tutti i tipi di software da poter interfacciare ad una differente tecnologia a meta livello.

Allo scopo di ottenere questo automatismo risulta di notevole aiuto uno strumento come Xtext, che non solo permette la definizione dei cosiddetti DSL, ma consente anche la generazione di codice automatica in concomitanza con la specifica di un dominio relativo.

Questo tipo di approccio funzionale permette di limitare le modifiche al solo ampliamento delle operazioni di traduzione tecnologica. I tempi ed i costi di una modifica di software di input o di tecnologia di output vengono ridotti drasticamente, infine la business logic non viene formalmente più cambiata una volta definita in sede di analisi.

6.1.2 Xtext

Xtext [10] si cataloga fra gli strumenti che forniscono assistenza nella scrittura di linguaggi di programmazione o linguaggi di dominio specifici.

In particolare Xtext è un vero e proprio framework di sviluppo per questi linguaggi specifici, in grado di coprire tutti gli aspetti di una infrastruttura completa. É capace di fornire implementazioni di default per aspetti come *parser*, *linker*, compilatori o interpreti e può essere efficacemente impostato per soddisfare le necessità più differenti.

Il progetto è open - source ed, oltre a fornire gli strumenti di analisi e lavoro attorno a linguaggio di dominio specifico, fornisce anche un modello delle classi per l'*Abstract Syntax Tree* (AST) ed un ambiente di sviluppo configurabile, integrato completamente in Eclipse.

Per specificare un linguaggio in questo modo un utente deve scrivere una grammatica nel linguaggio specifico Xtext. Da questa definizione un generatore di codice crea un parser ANTLR e le classi per il modello a oggetti. L'infrastruttura fornisce differenti funzionalità di comodo per la generazione di grammatiche di dominio, rimanendo altamente configurabile.

I risultati possono essere facilmente integrati in ambienti di sviluppo codice come linguaggio Java mappando i concetti del linguaggio specifico ad artefatti Java per ottenere un'integrazione olistica Java. Questa funzionalità è presente per differenti linguaggio tramite interpreti alternativi.

Possiamo vedere di seguito un esempio di quanto sia facile definire una grammatica per un linguaggio, questa è capace di definire le richieste che possono essere scambiate fra le parti ed evase dal server.

```
grammar org.webbrowser.metamodel.Metamodel with org.eclipse.xtext.common.Terminals

generate metamodel "http://www.webbrowser.org/metamodel/Metamodel"

MetaModel :
  (elements += Type)*
;

Type:
        DataType | Request
;

DataType:
        'datatype' name=ID
;

Request:
  'request' name=ID ':' type = RequestType
;

RequestType:
        ('login' | 'update' | 'datarequest') '{'
                (datas += Data)+
        '}'
```

```
;

Data:
        (many ?= 'many')? name=ID ':' type = [DataType]
;
```

Come si può notare bastano poche righe per definire una grammatica nella sua interezza; la modifica di altre parti dell'infrastruttura, presente per ogni progetto Xtext, permette la generazione automatica di codice, relativo agli elementi definiti in sede di specifica di modello. La stesura di questo file, definito di metamodello, permette di lanciare un ulteriore IDE Eclipse capace di elaborare come linguaggio specifico quello appena definito, permettendo all'utente la definizione di modelli legati ad applicazioni di qualunque tipo.

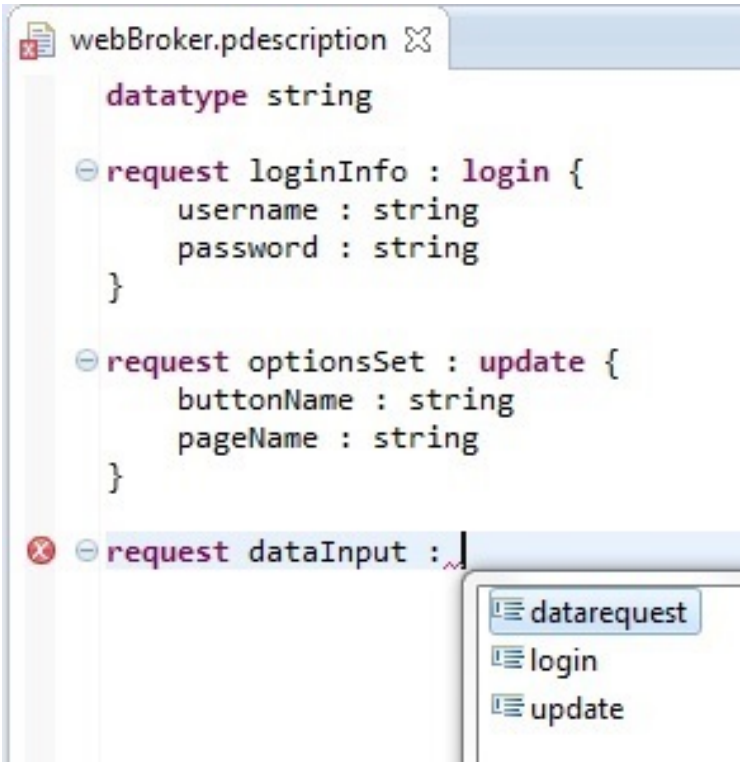

FIGURA 6.3: Eclipse - IDE linguaggio specifico.

Come visibile in Fig. 6.3 l'ambiente di lavoro si basa completamente sulla definizione del nostro linguaggio personalizzato, ed è capace di fornire suggerimenti sull'elaborazione stessa di un file di specifica.

```
datatype string
```

```
request loginInfo : login {
        username : string
        password : string
}

request optionsSet : update {
        buttonName : string
        pageName : string
}

request dataInput : datarequest {
        fieldName : string
        value : string
}
```

Dopo una prima fase di configurazione e specifica del linguaggio, lo strumento permette di descrivere senza ambiguità, e con una notevole espressività, un qualunque tipo di software applicativo. Un ulteriore passo di interesse è quello di modificare la specifica perché possa integrare tutte le parti che risultano un automatismo: come la traduzione di eventi da linguaggio client a linguaggio server o come la traduzione di interfacce server in linguaggio client. Anche in questo caso poche righe di codice permettono la generazione di una interfaccia ricca ed auto contenuta, già pronta per interagire con il server esistente, e quindi, con una qualsivoglia piattaforma tecnologica decisa a priori.

Capitolo 7

Riflessioni e sviluppi futuri

Il seguente capitolo si sofferma brevemente a considerare vantaggi e svantaggi relativi alla soluzione software implementata al Cap. 5. Vengono effettuate alcune riflessioni riguardo al lavoro svolto, inoltre vengono considerati alcuni miglioramenti e sviluppi futuri legati ad alcune parti fondamentali del progetto.

La capacità di sfruttare la maggior parte del codice già scritto comporta un forte vantaggio; allo stesso tempo i costi di adeguamento verso scenari differenti hanno evidenziato punti di debolezza della soluzione sviluppata. Questa infatti risulta fortemente legata al progetto specifico del gestionale modificato.

Il costo di una eventuale replicazione del processo è fondamentale dal punto di vista aziendale. Sono le tempistiche infatti a determinare il successo o il fallimento di un progetto sviluppato in fase preliminare.

7.1 Tecnologia server

Una prima riflessione riguarda la tecnologia sfruttata per l'implementazione del server web, tramite il quale viene esposto il servizio web del gestionale. Si può notare come la disponibilità di un tale servizio sia fortemente legata alle capacità della sua realizzazione tecnica. La soluzione trattata al Cap. 5.4.2 si basa fortemente su tecnologie legate all'ambiente di sviluppo, utilizzato per ragioni di compatibilità e adattabilità. Ovviamente una possibile richiesta di servizi da parte di numerosi client potrebbe recare forti disagi ad un server web poco performante.

Una soluzione a questo diffuso problema tecnico è stata recentemente sviluppata tramite l'utilizzo di tecnologie alternative a quella del *threading*; queste infatti sono basate su di un *pool* di thread riservato a livello di sistema. Un linguaggio, espressivamente più potente, come Javascript ha reso possibile l'eliminazione completa di processi multipli concorrenti.

7.1.1 node.js

Una tecnologia di spicco per quanto riguarda l'implementazione di tali tecnologie server web tramite javascript è sicuramente quella rappresentata da node.js [13].

Questa è una piattaforma usata per costruire reti scalabili, specialmente per applicazioni server. Sfrutta Javascript come linguaggio di scripting e contiene una libreria server integrata, permettendo l'utilizzo di web server senza la necessità di software esterno di controllo.

Il suo concetto si trova in contrasto con le tecnologie più largamente utilizzate su internet costituite da strumenti server che gestiscono un largo pacchetto di *thread*, lanciati per ogni richiesta. In questo caso non esistono processi separati in quanto il funzionamento è in linea con quello del codice Javascript. Per intendersi viene simulata la gestione di processi secondari, che avviene nei browser come chrome, alla chiamata di una callback javascript. Così facendo esiste un singolo *loop* in funzione, che per altro non è in grado di generare altri processi concorrenti, eliminando così la necessità di semafori nella gestione delle risorse computazionali. Questa modalità di gestione delle risorse elimina anche completamente la possibilità di *deadlock*, assistendo grandemente anche programmatori non esperti nella scrittura di server web complessi e performanti.

7.2 Generalità del modello

Una seconda analisi sulla definizione di un metamodello specifico, per generare progetti di interfacciamento software, porta in luce la possibilità di abbattere maggiormente i costi in termini di tempo. In particolare è interessante la capacità di generazione di codice automatica da parte dello strumento Xtext. Tramite questa peculiarità è facile eliminare i costilegati alla traduzione di eventi e di costrutti

grafici, i quali risultano intrinsechi alla generazione di un modello basato sulle regole grammaticali.

7.2.1 Xtext

Tramite questo strumento è quindi facile anche la generazione di un modello abile a gestire l'interfacciamento verso una tecnologia arbitraria di output. La capacità di generazione di codice automatica, infatti, permette di cambiare facilmente il codice generato in relazione ad un modello specifico. In questo modo possiamo definire il comportamento e le operazioni, rese disponibili da parte di un server qualunque, tramite un documento capace di definire il modello sottostante; questo può generare una infrastruttura completa, adatta all'interfacciamento di tale applicativo verso una nuova tecnologia.

In questo scenario un cambio di infrastruttura di uscita implica la sola riscrittura del linguaggio di output di tale strumento; mentre un cambio di software applicativo viene gestito tramite la stesura di un nuovo documento di definizione del modello, sfruttando il potere espressivo del linguaggio specifico a livello di metamodello.

Conclusioni

Durante lo svolgimento del lavoro di tesi è stato affrontata la complessa problematica dell'adeguamento tecnologico di un'applicativo. In particolare è stato analizzato un tipo di software strettamente legato al sistema operativo sul quale è stato sviluppato. L'obiettivo di rendere utilizzabile questa applicazione anche tramite supporti tecnologici quali tablet, senza dover riscrivere interamente il progetto, ha presentato diverse difficoltà ed alcuni spunti di riflessione.

Lo studio di soluzioni di terze parti, capaci di condividere in remoto l'applicativo in maniera interattiva, non hanno raggiunto i risultati sperati; mentre una soluzione di natura software è stata capace di manipolare adeguatamente la parte client del nostro gestionale.

Una riflessione riguardo alla possibile replica del processo di produzione, verso un differente tipo di software oppure verso una differente tecnologia di uscita, ha aperto la strada verso uno studio più generale. Tale studio ha evidenziato l'incapacità di una soluzione tradizionale di apportare cambiamenti a livello tecnologico ad un progetto fortemente accoppiato con la sua implementazione.

La seconda parte di analisi progettuale ha portato quindi alla luce la possibilità di definire un metamodello per replicare il processo di adeguamento. Tramite un primo utilizzo di questo strumento è stato possibile eliminare una buona parte dei costi di produzione. Lo sviluppo di un metamodello tramite strumenti Xtext infine ha posto le linee guide anche per eventuali sviluppi futuri, legati ad un generatore pressoché automatico di codice, capace di adeguare un qualunque software verso una qualunque tecnologia di uscita.

Il lavoro svolto in questa direzione rappresenta solo pochi passi verso la progettazione di un modello di metalivello capace di descrivere, tramite apposito DSL, un modello apposito per ogni tipologia di applicativo di interesse. Questa

capacità, in concomitanza alla potenza della generazione automatizzata di codice, può ridurre di molto i costi ed i tempi legati ad una tale operazione di modernizzazione.

Bibliografia

[1] Kerry Jiang. Superwebsocket, a .net websocket server, 2014. URL http://superwebsocket.codeplex.com/. [Online; in data 21-gennaio-2014].

[2] Antonio Natali. Contact - first frame page, 2013. URL //http://www-natali.deis.unibo.it/. [Online; in data 20-gennaio-2014].

[3] James Newton-King. Json.net 5.0.8, 2014. URL http://www.nuget.org/packages/newtonsoft.json/. [Online; in data 21-gennaio-2014].

[4] sta (http://sta blockhead.blogspot.it/). websocket-sharp a c# implementation of the websocket protocol client and server, 2014. URL http://www.nuget.org/packages/WebSocketSharp. [Online; in data 21-gennaio-2014].

[5] Wikipedia. Domain-specific language — wikipedia, l'enciclopedia libera, 2013. URL http://it.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=63238873. [Online; in data 13-febbraio-2014].

[6] Wikipedia. Json — wikipedia, l'enciclopedia libera, 2013. URL http://it.wikipedia.org/w/index.php?title=JSON&oldid=62282429. [Online; in data 21-gennaio-2014].

[7] Wikipedia. Model-view-controller — wikipedia, l'enciclopedia libera, 2013. URL http://it.wikipedia.org/w/index.php?title=Model-View-Controller&oldid=63301147. [Online; in data 21-gennaio-2014].

[8] Wikipedia. Remote method invocation — wikipedia, l'enciclopedia libera, 2013. URL http://it.wikipedia.org/w/index.php?title=Remote_Method_Invocation&oldid=57919093. [Online; in data 21-gennaio-2014].

[9] Wikipedia. Websocket — wikipedia, l'enciclopedia libera, 2013. URL http://it.wikipedia.org/w/index.php?title=WebSocket&oldid=62599254. [Online; in data 20-gennaio-2014].

[10] Wikipedia. Xtext — wikipedia, the free encyclopedia, 2013. URL http://en.wikipedia.org/w/index.php?title=Xtext&oldid=586422485. [Online; accessed 21-January-2014].

[11] Wikipedia. Css — wikipedia, l'enciclopedia libera, 2014. URL http://it.wikipedia.org/w/index.php?title=CSS&oldid=63453175. [Online; in data 21-gennaio-2014].

[12] Wikipedia. Html — wikipedia, l'enciclopedia libera, 2014. URL http://it.wikipedia.org/w/index.php?title=HTML&oldid=63583356. [Online; in data 21-gennaio-2014].

[13] Wikipedia. Node.js — wikipedia, the free encyclopedia, 2014. URL http://en.wikipedia.org/w/index.php?title=Node.js&oldid=591755490. [Online; accessed 22-January-2014].

[14] Wikipedia. Unified modeling language — wikipedia, l'enciclopedia libera, 2014. URL http://it.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=63375432. [Online; in data 21-gennaio-2014].

[15] Wikipedia. Server web — wikipedia, l'enciclopedia libera, 2014. URL http://it.wikipedia.org/w/index.php?title=Server_web&oldid=63549442. [Online; in data 21-gennaio-2014].

Ringraziamenti

Il raggiungimento di questo traguardo scandisce la fine di una intensa fase di crescita e maturazione; l'impegno personale ed il duro lavoro lo hanno reso possibile ma sono le persone, con le loro esperienze e parole, ad averlo reso un viaggio unico.

Desidero perciò ricordare tutti coloro che mi hanno aiutato ed assistito nel corso di questi anni di studio e nella stesura stessa della tesi con suggerimenti, critiche e tanta pazienza: a tutti loro va la mia gratitudine.

Ringrazio anzitutto il professor Antonio Natali per il supporto e l'assistenza garantita durante tutta la preparazione dell'elaborato.

Proseguo ringraziando il personale dell'azienda in cui ho avuto il privilegio di svolgere il mio periodo di stage, in particolare il mio responsabile Rudy Ricci, per i preziosi consigli e l'assistenza nella preparazione del prototipo.

Un ringraziamento speciale va ai colleghi ed amici che mi hanno sempre incoraggiato ed accompagnato nel corso di questi anni di fatiche e duro lavoro.
Ringrazio Ivan e Stefano per aver condiviso numerose sofferenze, sia durante le lezioni e gli esami più spaventosi, sia durante le serate online più disastrose.
Ringrazio Lorenzo e Nicola per essere stati così bravi a sopportarmi durante le giornate più difficili ed aver affrontato, con impavido coraggio, le sfide poste sul cammino dei loro sofferenti personaggi fantasy.

Un ringraziamento assertivo invece è indirizzato a tutti i ragazzi del gruppo Ronin Academy A.K.S. Rimini, in particolare al Sensei Francesco, per avermi mostrato la via del samurai ed avermi accompagnato durante la mia crescita spirituale. Oss.

Vorrei inoltre ringraziare le persone a me più care: la mia famiglia, in particolare mia madre, per avermi sempre spronato ed assecondato durante tutto questo tempo

nonostante le cadute, mio fratello Gianluca per aver ascoltato i miei deliri ed aver aggiunto un pizzico di follia alle giornate noiose ed infine la mia ragazza Lucia per essermi sempre stata vicina nei momenti di ordinaria pazzia ed in quelli di lucida rassegnazione.

In conclusione vorrei estendere un pensiero speciale a mio padre, venuto a mancare prematuramente, a cui questo lavoro è dedicato. La sua visione della vita ha indirizzato il mio cammino e mi ha spinto a non arrendermi mai di fronte alle difficoltà.