Alma Mater Studiorum - Università di Bologna

CAMPUS DI CESENA
SCUOLA DI SCIENZE
Corso di Laurea in Scienze dell'Informazione

PROGETTAZIONE E SVILUPPO DI SOFTWARE DI SUPPORTO ALLA LOGISTICA DEI TRASPORTI

Relazione finale in
Ricerca Operativa

Relatore:
Ill.mo Prof. Aristide Mingozzi
Correlatore:
Dott. Andrea Valletta

Presentata da:
Luca Canella

III Sessione
Anno Accademico 2012/2013

Ringraziamenti

Vorrei ringraziare il prof. Aristide Mingozzi e il dott. Andrea Valletta, per la costante disponibilità, l'aiuto e le conoscenze che mi hanno trasmesso.

Inoltre vorrei ringraziare il dott. Roberto Roberti, per la disponibilità concessa, e per i consigli condivisi in merito a questa tesi.

Ringrazio **GESP srl**, per l'opportunità accordatami di partecipare allo sviluppo del progetto *NST*, inserendomi in un gruppo di lavoro giovane e preparato, ed avendomi accolto con grande disponibilità e professionalità.

Ringrazio il *Team Trasporti*, cioè la business unit preposta allo sviluppo di *NST*, ed in particolare i miei colleghi Sandra Golfarelli, Federico Foschini e Fabio Colombo.

Un caloroso ringraziamento va a mia madre Silvia e mio padre William, ai miei nonni, ed agli zii, che mi sono stati vicini, credendo in me, con incrollabile pazienza, durante questi anni di studio.

Infine vorrei ringraziare mio fratello Giulio, la mia compagna Mina, ed i miei amici, con i quali il tempo vola, e senza i quali passa troppo lentamente.

Luca Canella

Introduzione

Uno dei problemi più diffusi, nell'ambito della logistica, è rappresentato dai costi di trasporto. La gestione dei flussi merci, l'approvvigionamento dei clienti, e la relativa pianificazione della movimentazione dei veicoli, hanno incidenze notevoli sui costi di gestione aziendali, i quali vengono stimati mediamente nel 45% dei costi logistici. A ragione di questo, sono sempre di più le aziende che ricorrono all'impiego di uffici dedicati alla pianificazione delle consegne e la gestione dei trasporti in generale.

Sebbene le voci di bilancio relative al trasporto raggiungano cifre rilevanti, fino al 4% del fatturato annuo, il tema della pianificazione viene spesso sottovalutato. Infatti la soluzione a problemi di pianificazione e monitoraggio dei costi, è spesso demandata a procedure manuali senza supporto informatico.

Nasce da qui l'esigenza di proporre uno strumento informatico che supporti gli addetti preposti alla pianificazione, sviluppando un sistema che copra esigenze di pianificazione dei viaggi, controllo e consuntivazione dei costi di trasporto, e monitoraggio dei mezzi in tempo reale.

La proposta di **Gesp srl, Geographic Information Systems**, azienda italiana che opera da anni nel campo delle applicazioni software geo-spaziali, prende il nome di **Nuovo Sistema Trasporti**, o più semplicemente, **NST**. In quest'ambito prende corpo questa tesi, la quale si pone l'obiettivo di illustrare le fasi di nascita, analisi, progettazione e sviluppo di un software generico per il supporto alla logistica. Saranno così analizzati:

- le problematiche affrontate nella fase di definizione, e kick-off (avvio), del progetto;

- il problema del routing, o *Vehicle Routing Problem* e le tecniche di Ricerca Operativa che vengono applicate per la sua risoluzione;
- le moderne metodologie di gestione e sviluppo di un software;
- l'architettura e le tecnologie impiegate per la distribuzione dell'applicativo.

Questa tesi si articola in cinque capitoli.

Nel capitolo 1 verrà introdotto il progetto software *NST*, i casi d'uso cui si applica, e la fase di kick-off del progetto.

Il capitolo 2 è dedicato alla formulazione matematica del problema dei trasporti: verranno qui introdotti gli algoritmi risolutivi, con i quali viene affrontato, e ne sarà fornito il relativo stato dell'arte ad oggi.

Nel capitolo 3, saranno introdotte alcune problematiche relative alle fase di approccio alla realizzazione del software. Saranno qui introdotte le metodologie adottate dal team di sviluppo, gli strumenti impiegati e i processi che caratterizzano le fasi di implementazione dell'applicativo.

Il capitolo 4, introduce l'architettura software di *NST*, e le tecniche con le quali vengono approcciati alcuni problemi comuni nello sviluppo del software.

Infine, il capitolo 5, è incentrato sulla progettazione e lo sviluppo dell'interfaccia utente, partendo dalle problematiche di design, fino alla descrizione delle tecnologie impiegate per la sua implementazione in ambito web.

Indice

Elenco delle figure

Capitolo 1

Il software, NST

Il progetto software *NST*, di *GESP srl*, nasce nel 2013 con l'intenzione di proporsi nel mercato dei software per la pianificazione dei trasporti, come prodotto d'avanguardia. Molti sistemi software, che rispondono ad esigenze logistiche, sono infatti basati su principi di sviluppo e tecnologie sorpassati, e, laddove sono state introdotte nuove piattaforme con caratteristiche più moderne, sono state notate mancanze in termini di adattabilità e completezza delle soluzioni fornite. L'acronimo *NST* sta per *Nuovo Sistema Trasporti*, nome che palesa l'obiettivo di fornire un software completo e moderno per il supporto alla logistica dei trasporti.

Nel seguito di questo capitolo, verranno introdotte le problematiche affrontate in fase d'approccio al progetto, introducendo: i casi d'uso principali ai quali *NST* può essere applicato, i principali ambiti aziendali ai quali mira, una descrizione del business plan associato al progetto, ed altri aspetti generali sulle fasi di valutazione preventiva e analisi del progetto.

1.1 Casi d'uso reali

NST nasce come prodotto di supporto alla pianificazione dei trasporti, il cui obiettivo è quello di coprire esigenze relative alla *pianificazione dei viaggi*,

al *controllo e consuntivazione* dei costo di trasporto, e al *monitoraggio dei mezzi di trasporto* in tempo reale.

L'idea di base è quella di un software generico, che si possa adattare a molteplici panorami aziendali, integrandosi con software già presenti (ERP, CRM), e fornendo una ampia gamma di personalizzazioni e moduli aggiuntivi, che facciano fronte a richieste specifiche dei singoli clienti.

NST deve essere basato sul web, al fine di poter offrire una piattaforma condivisa con supporto di multiutenza e minimo impatto sulle infrastrutture tecnologiche dei clienti. Si richiede quindi che il software sia flessibile e facile da utilizzare, dotato di interfacce personalizzabili e semplici.

Le situazioni affrontate dal software, come vedremo in dettaglio nel secondo capitolo, sono riconducibili ad una classe di problemi identificata dal *problema del routing*, o *VRP*. Questo genere di problemi sono applicabili ad una vasta gamma di aziende, ed enti, operanti nell'ambito della logistica, dei quali si riportano tre importanti casi d'uso: aziende di trasporto, aziende di servizi ed enti ambientali.

1.1.1 GDO, grande distribuzione organizzata

Per grande distribuzione organizzata, o *GDO*, si intende una rete di punti vendita al dettaglio, distribuiti sul territorio, operanti sotto un unico marchio. Le aziende della *GDO* sono caratterizzate da politiche di approvvigionamento centralizzate, e quindi, da un'omogeneità nella gestione dei fornitori tra i vari punti vendita. I punti vendita della rete di una *GDO* possono assumere varie forme, dal tradizionale negozio al dettaglio, *all'ipermercato*, allo *specialist drug store*[1], o negozi di *vendita all'ingrosso*.

Gli obiettivi principali riguardano l'ottimizzazione delle operazioni di approvvigionamento dei punti vendita e sono legati all'abbattimento dei costi di trasporto ed alla rapidità nella consegna di certe categorie merceologiche. Considerando inoltre, le ampie aree del territorio sul quale operano tali

[1]Negozi al dettaglio che forniscono principalmente prodotti per la cura della casa e/o della persona

aziende, e l'impiego di differenti tipologie di rami nella rete stradale, possono essere introdotti obiettivi specifici che riducano l'utilizzo di veicoli pesanti in aree urbane, o l'utilizzo di reti distributive a più livelli.

Particolare attenzione va posta nella gestione delle regole applicate ai veicoli della flotta, dovendo gestire ad esempio caratteristiche quali: la capacità dei vani di carico, in genere essere espresse in unità di misura differenti, la possibilità di percorrere o meno certi tratti della rete stradale e le differenti attrezzature disponibili (sponde idrauliche, vani refrigerati, etc.). Spesso sono richiesti servizi di tracciatura e monitoraggio dei mezzi in tempo reale.

I vincoli introdotti possono comprendere regole di urgenza sulla consegna di merci deperibili e tabelle orarie per il carico e scarico merci. Un ulteriore esempio di vincolo, sebbene raramente introdotto, comprende la richiesta di minimizzare il numero di svolte a sinistra (nei paesi con guida a destra) per ridurre il numero di attraversamenti della carreggiata di senso contrario, migliorando così parametri di sicurezza e riducendo i tempi di consegna.

1.1.2 Aziende di servizi

Le aziende che operano nell'ambito dei servizi svolgono interventi di assistenza e manutenzione presso clienti, sparsi su un territorio prevalentemente urbano. Rientrano in questa casistica le aziende che effettuano servizi di manutenzione agli ascensori, alle caldaie nelle abitazioni private, a macchine professionali per ufficio; per le caratteristiche del modello operativo, ricadono in questo caso d'uso anche le aziende che riforniscono clienti di merci poco voluminose, come prodotti consumabili per ufficio o snack per bar e punti commerciali. Gli interventi sono in genere effettuati da personale specializzato che si muove sul territorio con la propria macchina o con un piccolo furgone. In alcune realtà, dove la macchina è concessa come benefit al tecnico che la utilizza anche per uso privato, la giornata di lavoro ha inizio dall'abitazione del tecnico, e non dalla sede aziendale.

In particolare per gli interventi di assistenza tecnica, dove non è prevista la consegna di alcuna merce al destinatario, non è richiesta la preparazio-

ne del veicolo prima della partenza; i veicoli sono già equipaggiati con gli strumenti necessari agli interventi, e non è necessario effettuare alcun carico presso il deposito prima di visitare il cliente. Nel caso di distribuzione di piccoli quantitativi di merce, come il rifornimento di capsule per macchine da caffè, è possibile che sul furgone ci sia una scorta sufficiente sia per le visite programmate, sia per supportare eventuali attività di tentata vendita presso nuovi clienti.

In luogo dei vincoli di capacità dei veicoli, o di compatibilità per dimensioni con il sito del cliente, in ambito servizi sono predominanti le regole di assegnamento sulla base della compatibilità del veicolo (autista) con il cliente. Ad esempio, solo alcuni tecnici sono specializzati nella riparazione di alcune linee di prodotti, e non sono in grado di intervenire su altre. Nell'ambito dei tecnici con abilità compatibili con l'intervento programmato, è preferito il tecnico che abitualmente visita il cliente, per motivi di continuità e fidelizzazione del rapporto.

In genere gli autisti svolgono un solo giro al giorno, dove per giro si intende una sequenza di visite con partenza dal deposito (o dall'abitazione dell'autista). Il giro prevede una sosta per la pausa pranzo, ed un numero di visite che può variare sensibilmente in base ai diversi contesti; mentre la riparazione di una stampante professionale può richiedere un intervento anche di due o tre ore, la consegna di materiale per ufficio è molto rapida. Il numero di visite giornaliere può variare perciò da 2/3 visite a più di 20.

Una volta programmato il piano viaggi, la realizzazione durante lo svolgimento della giornata è molto dinamico. Le aziende di servizi operano prevalentemente in un contesto urbano, dove il fattore traffico è spesso determinante; sulla base della propria esperienza, e delle condizioni di traffico attuali, gli autisti possono variare la sequenze delle visite programmate, aggiungerne di nuove o non riuscire a completare il giro assegnato.

1.1.3 Enti ambientali, la raccolta dei rifiuti

Le terza tipologia di aziende comprende gli enti che operano nell'ambiente operando, ad esempio, raccolta di rifiuti. La raccolta dei rifiuti presenta problemi relativi alla periodicità della raccolta, alla capacità dei mezzi, al numero di risorse disponibili e all'individuazione dei percorsi ottimali verso le idonee aree di stoccaggio. Tali problematiche si articolano in base alla fase di gestione del rifiuto come la raccolta, lo stoccaggio, il trattamento e lo smaltimento. Tutto ciò aggiunge complessità alla modellazione dei depositi, i quali prevedono differenti impianti di gestione dei rifiuti e differente disponibilità di risorse. Il cliente in questi casi prende la forma del punto di raccolta, del quale si necessita il salvataggio di informazioni relative alla quantità di rifiuti prodotta per unità di tempo, la periodicità della raccolta e l'eventuale gestione della raccolta differenziata.

1.2 Business plan

La valutazione del progetto *NST* è iniziata con la stesura di un Business plan, cioé un documento che espone obiettivi, contenuti e caratteristiche del progetto imprenditoriale. Tale documento è di supporto ad ogni fase di sviluppo del software, a partire dall'analisi di fattibilità fino alla pianificazione delle attività di sviluppo.

Il business plan di *NST* comprende:

- una *panoramica sul problema* della gestione trasporti;
- i *settori aziendali* cui si rivolge NST e le loro principali esigenze;
- gli *obiettivi* del software, intesi sia come obiettivi funzionali, che come obiettivi economico/commerciali;
- un survey sulle principali soluzioni proposte dai *competitors*;
- la *struttura* del software, con particolare enfasi sui punti di forza di *NST* rispetto agli altri prodotti;

- un riassunto dei *requisiti funzionali* di base del prodotto;
- i requisiti in termini di *competenze*, richiesti per lo sviluppo del progetto;
- una descrizione dell'*approccio alla fase di sviluppo*;
- alcuni *possibili scenari reali*, all'interno dei quali si andrebbe a collocare *NST*.

A seguito della presentazione del business plan è stato deciso che il progetto sarebbe stato persegubile, dando il via alle successive fasi di project management.

1.3 Deliverable

Terminata la definizione degli obiettivi del progetto *NST*, il processo di pianificazione ha proceduto con l'identificazione delle principali attività di analisi e sviluppo. Ad ognuna di queste attività è stato associato il rilascio, di documenti o prodotti fisici, detti *deliverable*. Lo scopo dei *deliverable* è quello di fornire prova tangibile, e verificabile, delle attività svolte in merito allo sviluppo del progetto.

I principali deliverable prodotti per *NST* riguardano gli ambiti di:

- **Gestione della qualità**:
  - linee guida di scrittura del codice;
  - linee guida di progettazione del database.
- **Gestione della configurazione**:
  - creazione di un repository per il versioning dei sorgenti;
  - strumenti di condivisione dei documenti;
  - configurazione della macchina virtuale di sviluppo.

- **Strategia di comunicazione**.
- **Pianificazione delle attività**.

Il documento delle **linee guida di scrittura del codice**, riporta le direttive cui attenersi per la scrittura del codice, derivato da una una sintesi e adattamento delle linee guida Microsoft. Il documento standardizza aspetti quali la scelta dei nomi delle variabili, le opzioni di casing, la gestione dell'indentazione, e descrive regole più specifiche riguardo la visibilità di metodi, variabili di classe, etc. L'obiettivo è quello di uniformare la struttura dei sorgenti, per facilitare la leggibilità e la comprensione all'interno del gruppo di lavoro.

Le **linee guida di progettazione del database**, sono anch'esse raccolte in un documento, il quale riporta le direttive per la progettazione fisica del database. Qui sono descritte le regole per la scelta dei nomi di tabelle e campi, le convenzioni per le chiavi primarie, la gestione delle relazioni. L'obiettivo del documento è uniformare le strutture contenute nel database, e ricondurle ad una chiave di lettura condivisa. La regolamentazione nella progettazione del database ha ancora maggior importanza per la comprensione rispetto alla scrittura del codice, dove l'inserimento di commenti possono sopperire alla mancata leggibilità.

La creazione di un **repository per il versioning dei sorgenti**, gestito tramite un apposito strumento di controllo, ha lo scopo di fornire una banca dati centralizzata per i sorgenti, abilitare meccanismi di estrazione, modifica e riconciliazione delle versioni, e consentire ai componenti del team di disporre, in qualsiasi momento, della versione più aggiornata del codice.

La **condivisione dei documenti** di progetto di natura tecnica, commericale e di riferimento per la qualità del progetto, avviene attraverso un sito *Sharepoint*. Sharepoint è un *CMS*, cioè Content Management System, fornito da Microsoft, che permette di gestire contenuti web in modo rapido e strutturato, fornendo inoltre meccanismi di autenticazione e profilazione degli utenti. Tali meccanismi di profilazione permettono l'accesso ai documenti, sia a membri di sviluppo, sia al team di management.

La **strategia di comunicazione** impiegata comprende la condivisione degli avanzamenti di progetto all'esterno del team con cadenza di 6 settimane; tramite incontro con il Project Board vengono illustrati l'avanzamento tecnico e le opportunità commerciali raccolte.

Il **Project Plan** è un documento che espone la macro-programmazione delle attività. Qui sono individuati i deliverable specialistici di progetto, i quali sono poi suddivisi in work unit. Per ciascuna work unit è stato definito il perimetro funzionale e fatta una stima in giornate uomo per il completamento.

A fronte dell'impegno complessivo il project board ha autorizzato la formazione di un team dedicato per il delivery della soluzione; il tempo solare per il delivery della soluzione è stato determinato considerando l'effort complessivo di progettazione e la dimensione del team di lavoro.

1.4 Struttura dell'applicativo

NST è progettato come software basato sul web (*web-based*), ciò significa che le sue funzionalità sono distribuite tramite una rete, sia essa una rete aziendale (intranet), o globale (internet). L'accesso tramite network fornisce un supporto naturale alla condivisione dei dati e alla multiutenza, permettendo l'accesso simultaneo alla base dati ad un numero elevato di utenti, i quali possono scambiarsi informazioni in tempo reale.

L'applicativo si basa su un'architettura client-server classica, e su tecnologie largamente usate in ambito web, come *HTTP* e *HTML5*; questo fa sì che la complessità sia incapsulata il più possibile sul lato server, e che le tecnologie impiegate siano largamente supportate. In pratica per accedere al software è sufficiente un browser aggiornato, disponibile ad oggi sulla grande maggioranza di dispositivi connessi alla rete (computer, cellulari, tablet, palmari, etc.).

I principali vantaggi di questo approccio comprendono:

- *impatto estremamente limitato* sull'infrastrtuttura tecnologica del cliente, grazie ai limitati requisiti di hardware;

- *accessibilità* da una grande quantità di device connessi alla rete, anche con caratteristiche differenti;

- *semplificazione delle procedure di distribuzione ed aggiornamento* del software, il quale risiede in un'unica macchina e può essere aggiornato in tempo reale anche da remoto;

- maggior numero di *offerte commerciali* proponibili al cliente, le quali possono comprendere formule SaaS (*Software as a Service*) quali abbonamenti mensili a forfait, tariffazione a tempo *per numero di veicoli*, oppure formule personalizzate che possono comprendere l'installazione dell'applicativo sulla rete intranet aziendale.

1.4.1 Infrastruttura server

La complessità di *NST* è in gran parte racchiusa nel lato server, cioè in quell'insieme di componenti software che vengono gestiti dall'infrastruttura centralizzata. Tale infrastruttura è spesso composta da uno, o più, server e da altri apparati di rete necessari alla propagazione dei dati nel network. In questa infrastruttura sono dispiegati i vari servizi, che comprendono: una base dati, un web-server, un server *DNS*, una connessione ad internet/intranet.

Scopo del web-server è quello di distribuire il software e i dati, tramite protocollo HTTP, in risposta alle richieste dei client. Il client fa la richiesta ad un certo indirizzo web sul quale è pubblicato l'applicativo; a questo punto il server *DNS* risponde traducendo l'indirizzo testuale nell'*indirizzo IP* relativo al server oppotunamente configurato all'erogazione del servizio. La richiesta raggiunge così il web-server, il quale invia in risposta documenti, dati e *script* che verranno rielaborati dal browser del client. Per far fronte ad un numero elevato di richieste si fa uso di una tecnica detta *Round-robin DNS*. Questa tecnica prevede che l'applicativo sia dispiegato su più web-server, i

quali risiedono su diverse macchine, ad ognuna delle quali è assegnato un diverso indirizzo IP. Ogni volta che un client richiede al server *DNS* l'indirizzo dell'applicativo, questo risponde con un indirizzo diverso, bilanciando di fatto, il numero di richieste che raggiungono i differenti server. Un altro vantaggio introdotto dall'utilizzo di questa tecnica è il miglioramento della fault-tolerance del sistema, il quale in caso di malfunzionamento di un web-server, può redirezionare le richieste ai rimanenti web-server funzionanti.

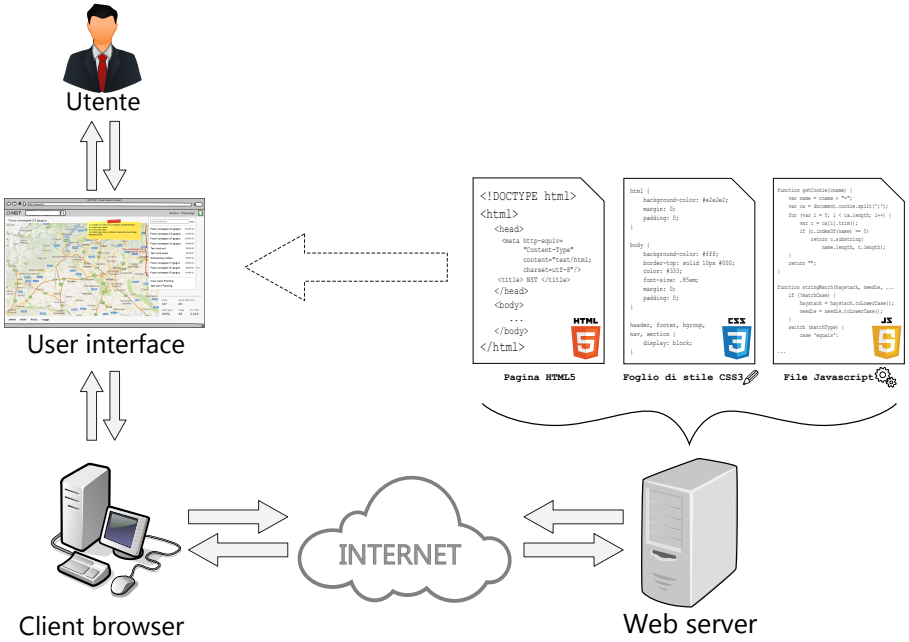
Il database invece è dispiegato su una macchina dedicata, generalmente connessa ai vari web-server tramite segmenti lan ad alta velocità. Il database server risponde alle richieste dati, di tutti i web-server, tramite un *DBMS* (Relational Database Management System) in esecuzione su di esso. Tale macchina deve rispondere a determinati requisiti di prestazioni, affidabilità e sicurezza. La configurazione più diffusa, per questo tipo di macchine, prevede l'impiego di un sistema *RAID 5*.

1.4.2 Il lato client

I client accedono ad *NST* tramite il web utilizzando un semplice browser. Un client può richiedere al server alcuni documenti, dati, fogli di stile, oppure set di istruzioni in formato Javascript, i quali compongono l'interfaccia utente dell'applicativo.

I documenti, o pagine, di cui è composto *NST* sono ipertesti in formato *HTML5*, i quali, corredati da fogli di stile in formato *CSS3*, determinano il layout dell'interfaccia utente. Gli script in formato Javascript, allegati alle pagine ed interpretati dal browser, permettono di implementare dinamiche di interazione non dissimili da quelle fornite da un applicativo desktop. Questi script possono interrogare il server per ottenere dati in tempo reale, i quali vengono inoltrati in formato XML o Json, elaborati e mostrati all'utente.

Figura 1.1: Struttura web-based dell'interfaccia



1.5 Analisi e scelte

L'analisi dei requisiti, svolta al fine di individuare le funzionalità richieste è stata condotta tramite interviste, svolte presso potenziali acquirenti del prodotto finale, le quali hanno permesso di individuare un grande numero di casi d'uso e di servizi richiesti al software. Le tre problematiche principali alle quali il sistema deve rispondere comprendono:

- pianificazione dei viaggi;
- controllo e consuntivazione dei costi di trasporto;
- monitoraggio dei mezzi in tempo reale.

Tali problematiche si articolano a loro volta in diversi punti: la pianificazione comprende l'ottimizzazione dei giri, la possibilità di personalizzarli e il controllo dei vincoli di fattibilità per i giri prodotti. Il monitoraggio dei mezzi di trasporto in tempo reale comprende la possibilità di segnalare agli autisti i viaggi da effettuare in modo automatico tramite l'interfacciamento

con specifici terminali bordo mezzo. A questi punti fanno seguito una serie di problematiche derivate quali:

- gestione delle anagrafiche di veicoli, autisti, clienti, ...
- integrazione con erp aziendali
- integrazione con applicazioni di terze parti
- condivisione di informazioni a livello aziendale

Allo scopo di far fronte a problematiche così varie, *NST* è stato pensato come software web-based generico, adattabile all'occorrenza alle specifiche esigenze dei clienti. Il prodotto in questione deve quindi avere particolari doti di generalità, modularità, scalabilità e incrementalità.

1.6 Domini applicativi del VRP

Nonostante la modellazione delle entità, coinvolte nella risoluzione del VRP, dipenda dalla specifica istanza del problema, è possibile individuare alcuni elementi comuni nei vari domini applicativi. Tra questi abbiamo i veicoli, gli autisti, i depositi, i clienti e la rete stradale, che a meno di piccole variazioni mantengono molte caratteristiche tra le varie istanze del problema.

La rete stradale viene rappresentata da un grafo, nel quale gli archi rappresentano i tratti di strada percorribile e i nodi rappresentano incroci o indirizzi di clienti e depositi. Gli archi del grafo possono essere direzionati o non direzionati, e sono caratterizzati da un costo calcolato in base al tempo di percorrenza dal tratto stradale rappresentato. Sul grafo sono localizzati i punti geografici relativi agli indirizzi dei clienti, dei depositi ed eventuali intersezioni. Per alcune applicazioni è necessario modellare la tipologia di arco correlata al tipo di strada rappresentato (autostrada, superstrada, strada secondaria, ponti, etc.), per avere informazioni sulle dimensioni massime dei veicoli in transito (peso, altezza, larghezza, lunghezza), eventuali limitazioni dovute al codice della strada (divieti, limiti di velocità, etc.) o

pedaggi imposti per il loro attraversamento. Il grafo è utilizzato per calcolare un distanziere, rappresentato da una matrice, nella quale gli indici di riga e colonna rappresentano i punti d'interesse del grafo e ogni elemento della matrice rappresenta il costo, in termini di distanza o di tempo, per l'attraversamento della rete stradale che congiunge tali nodi.

Le caratteristiche tipiche dei *clienti* comprendono:

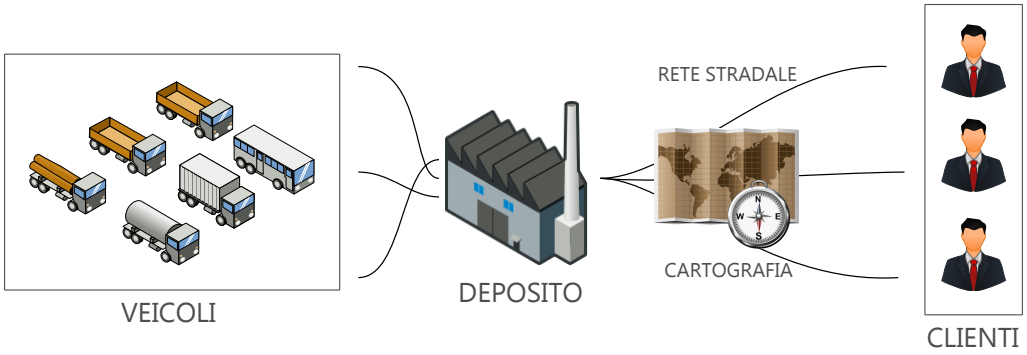
- *informazioni geografiche*, come l'indirizzo o le coordinate geografiche, i quali determinano anche il nodo del grafo stradale dal quale è identificato il cliente;
- *la domanda*, che rappresenta la quantità di merce richiesta;
- *time windows*, o meglio i periodi temporali nei quali il cliente può essere visitato;
- *tempo richiesto*, sia esso un tempo di caricamento o scaricamento di merci oppure il tempo necessario per erogare un servizio richiesto;
- *veicoli ammessi e non ammessi*, determinati ad esempio da problematiche logistiche o relative alla rete stradale.
- *costi o penali* in caso di mancato servizio.

Per gestire vincoli sul caricamento e lo scaricamento delle merci è necessario modellare i *depositi*, anch'essi localizzati geograficamente e quindi rappresentati da un certo nodo del grafo. Ad ogni deposito è associato un insieme di veicoli e una capacità produttiva, intesa come la quantità di merce che può essere servita ai clienti dai veicoli del deposito.

Per ogni veicolo in flotta è necessario modellare le seguenti caratteristiche:

- *posizione di partenza*, che identifica in genere il deposito di partenza o un differente nodo del grafo se previsto dai vincoli del problema;
- *capacità* dei vani di carico, espressa come peso, lunghezza, volume, o numero massimo di colli (o pallet) che il veicolo può trasportare;

Figura 1.2: Il problema del routing



- *insieme di archi* del grafo che il veicolo può attraversare;
- *varianti sul costo di trasporto* per l'utilizzo del veicolo su una certa tratta stradale, espressi per unità di distanza, di tempo o per tipologia di strade attraversate;

Gli autisti sono legati ad una serie di vincoli derivati da norme legali e contrattuali, e dalla necessità di risposo. Si associano quindi ad ogni autista: orari di lavoro, durata massima di guida giornaliera, pause di servizio e disponibilità giornaliera.

1.6.1 L'ottimizzatore

Il componente software che risolve il problema del *VRP*, fornendo come soluzione la pianificazione dei viaggi, è detta **Ottimizzatore**.

L'interfacciamento con l'ottimizzatore è ottenuto tramite la definizione di opportune strutture dati e formati d'interscambio per l'input, e l'output, dei dati tra l'algoritmo e l'applicativo. Le strutture dati fornite come input all'ottimizzatore sono costruite come un elaborato subset dei campi delle entità del dominio applicativo, e forniscono all'algoritmo la base dati necessaria alla risoluzione del problema. L'output fornito dall'ottimizzatore, invece, deve comprendere una serie di informazioni che permettano ad *NST* di ricostruire, dalla soluzione fornita, una serie di oggetti e relazioni che rappresentino la pianificazione ottimizzata.

La necessità di produrre soluzioni con tempo di calcolo ridotto, per problemi complessi, impone l'impiego di metodi veloci di tipo costruttivo, per generare una soluzione ammissibile, e di tecniche di local search per migliorarla (questo aspetto sarà approfondito nel capitolo 2).

Capitolo 2

Introduzione al problema del routing, il VRP

Il Vehicle Routing Problem, o *VRP*, rappresenta una classe di problemi, relativi alla gestione dei trasporti, che coinvolge tutte le aziende attive nel campo della consegna e raccolta di beni o servizi. Si tratta di problemi per i quali, data una flotta di veicoli e un insieme di clienti, si richiede di calcolare un sistema di percorsi minimizzando i costi e rispettando una serie di vincoli di servizio.

Esistono numerose varianti del *VRP*: possono variare tipologia dei vincoli, caratteristiche dei veicoli, possono esserci differenti obiettivi da perseguire e diverse categorie di servizio richieste dai clienti. Gli obiettivi, ad esempio, possono rieferirsi ai tempi di percorrenza, alla lunghezza complessiva dei viaggi, al numero totale di viaggi effettauti o al costo dei viaggi. I veicoli, invece, possono essere caratterizzati da capacità e velocità medie differenti, dalla possibilità di percorrere o meno un certo tratto stradale o dall'insieme di depositi presso i quali si possono approvvigionare (nel caso di problemi multi deposito). Le categorie di servizio richieste dai clienti possono essere di consegna, ritiro, consegna e ritiro oppure erogazione di un servizio (ad esempio servizi di manutenzione presso il cliente). Si possono infine considerare molte tipologie differenti di vincoli quali: vincoli sulla capacità dei

veicoli, lunghezza minima o massima dei viaggi, vincoli di tempo sugli orari degli operatori o sulla disponibilità dei clienti, priorità o precedenze di determinati clienti.

La maggior parte delle formulazioni del *VRP* appartiene all'insieme dei problemi NP-ardui, che sono quei problemi per i quali il calcolo di una soluzione esatta richiede tempo polinomiale utilizzando un algoritmo branch-and-bound intelligente, cioè in grado di valutare ad ogni passo il figlio giusto. I problemi NP-ardui sarebbero pertanto risolvibili, in tempo polinomiale, solamente da una macchina di Touring non deterministica. Nella pratica, secondo la teoria della complessità, si dice che questi problemi hanno tempo di risoluzione che cresce in modo esponenziale con la dimensione dell'input.

Gli algoritmi risolutivi per questa classe di problemi sono suddivisi in due tipologie:

- Algoritmi esatti, che cercano la soluzione ottima al problema impiegando tempo esponenziale.
- Algoritmi euristici, che cercano una soluzione approssimativamente vicina a quella ottima operando in tempo polinomiale.

La scelta di un algoritmo euristico, in luogo di un algoritmo esatto, è spesso dettata dall'impossibilità di ottenere una soluzione ottima in tempi ragionevoli per certe dimensioni del problema.

2.1 Formulazione matematica del VRP

Il *VRP* è un problema NP-arduo che estende il tipico *problema del commesso viaggiatore* (o Travelling Salesman Problem, o TSP) introducendo una flotta di veicoli in luogo di un singolo veicolo.

La formulazione del *TSP* richiede che, dato un grafo G, si trovi un ciclo di costo minimo, il quale visiti tutti i vertici del grafo una ed una sola volta. Come vedremo ciò significa cercare il ciclo hamiltoniano di costo minimo in G.

Introdurremo inizialmente la notazione della teoria dei grafi utile a descrivere il problema in termini matematici; individueremo poi una formulazione del *VRP* in forma base per poi enumerare alcune delle sue varianti più famose.

2.1.1 Notazione della Teoria dei Grafi

Introduciamo la notazione della teoria dei grafi utile a modellare formalmente il grafo della rete stradale usato durante la risoluzione del VRP. Sia $G = (V, A)$ un insieme di elementi di cui $V = v_1, v_2, v_i, ..., v_n$ sono detti vertici e A è l'insieme degli archi del tipo (i, j), che congiungono i nodi i con i nodi j. Un arco si dice orientato se presenta una connessione diretta da i a j, mentre si dice non orientato se tale connessione non specifica un verso di percorrezza.

Individuiamo ora in A i due insiemi D, degli archi orientati, ed E, degli archi non orientati, tali che $A = (D \cup E)$. Il grafo G si dirà quindi orientato se $E = \emptyset$, non orientato se $D = \emptyset$, o misto se $D \neq \emptyset \wedge E \neq \emptyset$. Se ad ogni arco (i, j) è associato un costo c_{ij}, allora il grafo si dice pesato. Diremo inoltre che G è connesso se per ogni coppia (i, j) di nodi esiste un cammino in G che li collega.

Sia G un grafo misto, definiamo ora:

- *stella uscente* del nodo i, indicato con $\Delta^+(i)$, l'insieme di tutti i nodi j tali che $(i, j) \in D$
- *grado uscente* del nodo i, indicato con $d^+(i)$ la cardinalità di $\Delta^+(i)$
- *stella entrante* del nodo i, indicato con $\Delta^-(i)$, l'insieme di tutti i nodi j tali che $(j, i) \in D$
- *grado entrante* del nodo i, indicato con $d^-(i)$ la cardinalità di $\Delta^-(i)$
- *grado* del nodo i, indicato con $d(i)$, la cardinalità dell'insieme $\Delta(i)$ dei nodi j tali che $(i, j) \in E$.

Un viaggio, o ciclo, in un grafo G è una sequenza di nodi i_k e archi a_k del tipo $(i_1, a_1, i_2, a_2, \ldots, i_k, a_k, \ldots, i_n, a_n, i_{n+1})$, dove a_k è l'arco (i_k, i_{k+1}) e $a_1 = a_{n+1}$. Un viaggio è euleriano (*euler tour*) se contiene ogni arco del grafo G esattamente una volta; se il ciclo contiene ogni arco di G almeno una volta allora è detto *postman tour*, ed è detto *covering tour* se copre tutti gli archi del grafo G. Un ciclo si dice *hamiltoniano* se visita una ed una sola volta tutti i vertici del grafo. Diremo che un grafo G è un *grafo euleriano* se possiede un ciclo euleriano e, analogamente, diremo che è un *grafo hamiltoniano* se possiede almeno un ciclo hamiltoniano.

2.1.2 Capacitated VRP

La versione di base del VRP è detta Capacitated Vehicle Routing Problem ($CVRP$). Questa formulazione prevede domande deterministiche e non partizionabili (con corrispondenza univoca tra clienti e consegne), singolo deposito di partenza per ogni consegna e vincoli di capacità sui veicoli. L'obiettivo del problema è quello di minimizzare i costi, in termini di tempo di servizio o distanza effettiva, servendo tutti i clienti. Il problema può essere così descritto:

- Sia $G = (V, A)$ un grafo completo dove V è l'insieme dei vertici e A l'insieme degli archi. I vertici di V $1, \ldots, n$ corrispondono ai clienti e il vertice 0 corrisponde al singolo deposito di partenza.

- Sia c la matrice dei costi, dove c_{ij} è il costo non negativo di percorrenza dell'arco (i, j). L'utilizzo di archi del tipo (i, i) non è permesso associando un costo $c_{ii} = +\infty$ per ognuno dei vertici.

- Ad ogni cliente i è associata una domanda d_i non negativa. Si impone una domanda nulla per il singolo deposito, sia dunque $d_0 = 0$.

- Sia K l'insieme dei veicoli, ognuno dei quali ha capacità C. Assumiamo che $d_i \leq C \forall i = 1, \ldots, n$ e che ogni veicolo possa eseguire al massimo un viaggio. Assumiamo inoltre $K_{min} \subseteq K$

Se il grafo G è orientato, la matrice delle distanze sarà asimmetrica; diremo quindi che il problema è di tipo ACVRP (*Asymmetric* Capacitated VRP). Nella pratica spesso la matrice dei costi soddisfa la *disuguaglianza triangolare*, cioè vale

$$c_{ij} + c_{jk} \geq c_{ik}$$

In altre parole percorrendo l'arco tra i vertici i e k, c_{ij} è il costo minimo affrontabile e non conviene intraprendere altri cammini tra i due vertici. Laddove la matrice c dei costi rappresenta la distanza euclidea tra vertici geolocalizzati, avremo una matrice simmetrica che soddisfa la disuguaglianza trinagolare. Questa proprietà può tuttavia essere violata qualora i valori della matrice c vengano arrotondati all'intero più vicino; in questo caso è preferibile arrotondare tutti gli elementi di c all'intero superiore.

Il valore di K_{min} può essere determinato risolvendo il problema del Bin Packing (BPP) associato al CVRP, che determina il minor numero di contenitori di capacità C richiesto per caricare tutti gli elementi con peso non negativo in un certo insieme. Sebbene il BPP sia NP-arduo è possibile risolverne istanze di centinaia di elementi in modo ottimale in tempi accettabili [MT90].

Definiamo il costo di un viaggio come la somma dei costi degli archi attraversati dallo stesso. Il CVRP consiste nell'individuazione di una serie di K viaggi con costo minimo in modo che:

- ogni viaggio parte dal deposito;
- ogni cliente è visitato da esattamente un circuito;
- la somma delle richieste per vertice (cliente) non ecceda la capacità C del veicolo.

2.1.3 Modello di programmazione lineare per CVRP

Sia x_{ij}^k una variabile binaria così definita:

$$x_{ij}^k := \begin{cases} 1 & \text{se nella soluzione l'arco } (i,j) \text{ è percorso dal veicolo } k \\ 0 & \text{altrimenti} \end{cases} \tag{2.1}$$

formuliamo il problema in modo da minimizzare la funzione obiettivo z così definita,

$$z = \min \sum_{k \subseteq K} \sum_{(i,j) \in A} c_{ij} x_{ij} \tag{2.2}$$

rispettando i seguenti vincoli:

$$\sum_{k \subseteq K} \sum_{j \subseteq V} x_{ij}^k = 1, \qquad \forall i \in V \tag{2.3}$$

$$\sum_{i \in V} q_i \sum_{j \in V} x_{ij}^k \leq C_k, \qquad \forall k \in K \tag{2.4}$$

$$\sum_{j \in V} x^k oj = 1, \qquad \forall k \in K \tag{2.5}$$

$$\sum_{i \in V} x^k ih - \sum_{j \in V} x^k hj = 0, \qquad \forall h \in A, \forall k \in K \tag{2.6}$$

$$\sum_{i \in V} x^k i, n+1 = 1, \qquad \forall k \in K \tag{2.7}$$

$$xij \in \{0,1\}, \qquad \forall i,j \in V \tag{2.8}$$

Il vincolo 2.3 assicura che ogni cliente i sia servito da un singolo veicolo. Il vincolo 2.4 si assicura che la domanda soddisfatta da ogni veicolo non ne ecceda la capacità. Il vincolo 2.5 imposta il deposito come punto di partenza per ogni veicolo. Il vincolo 2.6 impone al veicolo il transito sull'arco generico h. Il vincolo ridondante 2.7 conferma la struttura della route imponendo al veicolo il ritorno al nodo fittizio $n+1$. Infine il vincolo 2.8 definisce la natura binaria delle variabili x_{ij}^k.

2.2 Varianti e metodi risolutivi per il VRP

Il modello matematico del *VRP* può essere adattato in base alle diverse problematiche poste dai clienti, le quali possono introdurre varianti concernenti vincoli, obiettivi, caratteristiche di veicoli, clienti e depositi, o modalità di erogazione del servizio. Per quanto riguarda i servizi distinguiamo tra tre differenti tipologie quali la consegna, il ritiro, o entrambi (*pick-up and delivery*). Se il problema prevede un singolo deposito parleremo di *single depot VRP*, altrimenti si parlerà di *multi depot VRP*. Si possono introdurre vincoli sul numero dei veicoli utilizzabili e sulla tipologia dei singoli veicoli. Le funzioni obiettivo possono dipendere dalla lunghezza dei percorsi o da più parametri quali: numero dei viaggi, numero di veicoli impiegati, tempo di percorrenza, costo fisso di veicoli ed autisti. Alcuni vincoli, ricorrenti tra le varie istanze, comprendono inoltre:

- lunghezza massima dei viaggi;
- introduzione di *time windows*, cioè di intervalli temporali, nei quali è possibile erogare un servizio;
- precedenze o priorità sui clienti;
- penali per parziale o completa inadempienza;
- vincoli sindacali relativi ad orari, pause o straordinari del personale.

2.2.1 VRP con Time Windows e Formulazione Set Partitioning

Il VRPTW, cioè VRP con Time Windows, è, assieme al CVRP, uno dei membri della famiglia del VRP maggiormente studiati. Il VRPTW generalizza il CVRP imponendo, per ogni cliente, alcuni intervalli di tempo all'intero dei quali esso deve essere visitato. Questi intervalli sono chiamati **Time Windows**. Come per il CVRP, anche il VRPTW è un problema NP-arduo.

Il VRPTW è definito su un grafo completo $G = (V', A)$, dove $V' = 0, 1, ..., n$ è un set di $n + 1$ vertici ed A è l'insieme degli archi. Il sottinsieme $V = V'\backslash\{0\}$ corrisponde all'insieme dei clienti, e il vertice 0 rappresenta, come di consueto, il deposito. Ad ogni vertice $i \in V'$ è associata una richiesta q_i, assumendo $q_0 = 0$, e una time window $[e_i, l_i]$, della quale e_i e l_i rappresentano gli estremi dell'intervallo di tempo.

La formulazione matematica, sulla quale sono basati alcuni dei migliori metodi esatti di risoluzione del VRPTW, è la formulazione Set Partitioning, o SP:

Sia $\mathscr{R}$ l'insieme delle routes ammissibili, e $a_{il}, i \in V', l \in \mathscr{R}$ un coefficiente binario uguale a 1 se $i \in V(R_l)$, 0 altrimenti (assumendo che $a_{0l} = 1, \forall l \in \mathscr{R}$). Ogni route $l \in \mathscr{R}$ ha un costo associato c_l. Sia $x_l, l \in \mathscr{R}$, una variabile binaria che vale 1 se e solo se la route l è nella soluzione ottima. La formulazione del VRPTW, basata sul modello SP, è:

$$z(F) = min \sum_{l \in \mathscr{R}} c_l x_l, \tag{2.9}$$

$$s.t. \sum_{l \in \mathscr{R}} a_{il} x_l = 1, \qquad \forall i \in V \tag{2.10}$$

$$\sum_{l \in \mathscr{R}} x_l \leq m \tag{2.11}$$

$$x_l \in \{0, 1\}, \qquad \forall l \in \mathscr{R}. \tag{2.12}$$

I vincoli 2.10 specificano che ogni cliente $i \in V$ deve essere visitato da esattamente una route. Il vincolo 2.11 richiede che al massimo m routes siano selezionate.

2.2.2 Metodi risolutivi esatti

In quanto generalizzazione del *TSP*, il *VRP* è NP-arduo cioè, al crescere delle dimensioni dell'input, il tempo di risoluzione cresce esponenzialmente. Nella pratica il *VRP* (nello specifico la sua formulazione base, il *CVRP*) si dimostra significativamente più complesso da risolvere rispetto al *TSP*. I

migliori algoritmi esatti per il *CVRP* raramente possono affrontare istanze che coinvolgono più di 100 vertici; è noto, invece, che per il *TSP* vengono risolte all'ottimo istanze da migliaia di vertici in tempi accettabili [BL07].

I metodi più diffusi, per risolvere il *VRP* all'esatto, fanno uso di algoritmi basati su tecniche di *branch-and-bound* oppure *branch-and-cut*. La tecnica *branch-and-bound* è una tecnica generale per la risoluzione di problemi di ottimizzazione combinatoria, la quale prevede la scomposizione del problema in sottoproblemi più semplici da risolvere. Le procedure *branch-and-bound* si possono quindi scomporre in procedure di *branching*, cioè di partizione ricorsiva dello spazio di soluzioni ammissibile, e *bounding*, cioè di risoluzione dei sottoproblemi generati in fase di *branching*. Le tecniche di *branch-and-cut* invece sono tecniche miste di *branch-and-bound/piani di taglio*; tali tecniche prevedono che, ad ogni nodo dell'albero decisionale, si generino tagli allo scopo di individuare una soluzione intera, oppure, un bound più stretto. La generazione dei tagli viene interrotta laddove questi diventano poco efficaci, effettuando a questo punto un branching.

Ad oggi gli algoritmi esatti più efficaci, per la **soluzione all'ottimo del CVRP**, sono dovuti a Baldacci et al. (2004), Lysgaard et al. (2004), Fukasawa et al. (2006) e Baldacci et al. (2008). Baldacci et al. (2004) descrivono un algoritmo branch-and-cut (bc) basato su una formulazione matematica con due tipi di variabili di flusso. Lysgaard et al. (2004) hanno proposto un algoritmo bc che è un miglioramento del metodo porposto da Augerat et al. (1995). Il metodo Fukasawa et al. (2006) combina la bc di Lysgaard et al. (2004) con un nuovo algoritmo di tipo branch-and-cut-and-price (BCP), basato su formulazioni a due indici e sulla formulazione Set Partitioning (SP). Il lower bound è calcolato con un metodo di tipo *column and cut generation* che utilizza il rilassamento q-routes, introdotto da Christofides, Mingozzi, Toth (1981), anziché i percorsi elementari e le *disuguaglianze valide*, proposte da Lysgaard et al. (2004). L'algoritmo decide al nodo radice se utilizzare il bc di Lysgaard et al. (2004), o l'algoritmo BVCP. Baldacci et al. (2008) ha presentato un algoritmo basato sul modello

SP rinforzato con disuguaglianze di capacità e clique.

Questi ultimi autori sono stati i primi a calcolare lower bounds basati sull'uso di percorsi elementari. L'algoritmo di Baldacci et al. (2008) non è in grado di risolvere tre problemi risolti da Fukasawa et al. (2006), ma è notevolmente più veloce sui problemi risolti da entrambi i metodi.

Baldacci et al. (2012) descrivono il metodo più efficace, pubblicato in letteratura finora, per risolvere sia il CVRP, che il Vehicle Routing Problem con Time di Windows (VRPTW). Questo metodo migliora quello proposto da Baldacci et al. (2008) per il CVRP, basandosi sul modello SP del problema, ed introducendo un nuovo rilassamento dei percorsi elementari, denominato *ng-route*, utilizzato da diverse euristiche, per calcolare diverse soluzioni duali quasi ottime, del rilassamento lineare della formulazione SP. Essi descrivono un algoritmo column-and-cut-generation per calcolare il lower bound, rafforzato da disuguaglianze valide, che utilizza una nuova strategia per risolvere il problema del pricing (generazione di nuove colonne). Il nuovo rilassamento *ng-route*, e le diverse soluzioni duali ottenute, permettono di generare un problema di SP ridotto, contenente tutte le soluzioni ottime, il quale è risolto da un solutore di programmazione intera. Il metodo proposto risolve 4 dei 5 problemi di VRPTW precedentemente non risolti, e migliora significativamente i tempi di calcolo degli algoritmi esistenti, sia per VRPTW, che per CVRP.

Il **primo algoritmo esatto per il VRPTW**, basato sulla formulazione SP, è stato proposto da Desrochers et al. (1992). Questo metodo è stato in seguito migliorato da Kohl et al. (1999), aggiungendo le disuguaglianze 2-path al rilassamento lineare della formulazione SP. Kohl e Madsen (1997) hanno proposto un algoritmo di branch-and-bound, dove il lower bound viene calcolato con il sottogradiente. Irnich e Villeneuve (2006) descrivono un algoritmo branch-and-price, dove il sottoproblema pricing è risolto utilizzando il metodo dei q-path, con eliminazione di cicli di k vertici consecutivi. Algoritmi basati sul calcolo dei percorsi elementari sono stati proposti dalla Feillet et al. (2004) e Chabrier (2006).

Jepsen et al. (2008) descrivono un bcp, basato sul modello SP rafforzato da nuove disuguaglianze denominate Subset Row(SR). Questo metodo è stato migliorato da Desaulniers et al. (2008), con l'aggiunta di una euristica tabu-search per la generazione rapida di routes di costo ridotto negativo. Questo metodo domina tutti gli altri algoritmi precedentemente pubblicati, riducendo notevolmente il tempo di calcolo, e risolvendo 5 delle 10 istanze non risolte del VRPTW.

In generale qualsiasi algoritmo esatto per la VRPTW può essere facilmente adattato per risolvere il CVRP, semplicemente rilassando i vincoli imposti dalle time windows. Tuttavia, tale semplice adattamento, potrebbe non essere efficace, e nessuno dei metodi finora pubblicati in letteratura per la VRPTW ha dimostrato di risolvere efficacemente il CVRP.

In conclusione l'unico metodo esatto pubblicato ad oggi in letteratura che risolve sia il CVRP ed il VRPTW e che domina tutti gli altri metodi è quello proposto da Baldacci, Mingozzi e Roberti (2012).

2.2.3 Metodi euristici

Gli algoritmi euristici forniscono, per problemi complessi, soluzioni subottime in tempi relativamente brevi. Tali soluzioni, seppur non ottime, sono accettabili e rappresentano, per certe dimensioni dell'input di problemi complessi come il *VRP*, le uniche soluzioni che si possono ottenere con limitati tempi di calcolo. Gli euristici classici, proposti per il problema del *VRP*, si possono suddividere in metodi costruttivi (o *route construction methods*), metodi a due fasi (*two-phase methods*), e metodi migliorativi (*route improvement methods*).

Gli algoritmi **route construction heuristics** partono tipicamente da una soluzione vuota e costruiscono iterativamente nuovi viaggi inserendo uno o più clienti alla volta, fino a che tutti i clienti sono serviti. Gli algoritmi costruttivi sono caratterizzati principalmente da tre caratteristiche: i criteri di inizializzazione, i criteri di selezione dei clienti, e i criteri di selezione della posizione del cliente nella route in costruzione.

Gli algoritmi **two-phase heuristics** si basano sulla scomposizione del processo di risoluzione in due sottoproblemi: *clustering*, cioè la determinazione di sottinsiemi di clienti corrispondenti ad un viaggio, e *routing*, cioè la determinazione della sequenza con la quale i clienti vengono serviti in ogni viaggio. Negli euristici a due fasi che prevedono inizialmente la fase di clustering (metodi *cluster-first-route-second*) la fase di clustering può essere variabile da algoritmo ad algoritmo, mentre la seconda fase si riduce alla risoluzione del *TSP*.

Gli algoritmi **route improvement heuristics** sono algoritmi a ricerca locale, spesso utilizzati per migliorare soluzioni iniziali, non necessariamente ammissibili, generate da altri euristici. Ad ogni iterazione dell'algoritmo vengono apportate semplici modifiche alla soluzione, come lo scambio di archi o lo scambio di clienti tra i viaggi, allo scopo di ottenere soluzioni locali di costo inferiore. Laddove la nuova soluzione elaborata migliora il costo della soluzione iniziale, allora questa diventa la nuova soluzione al problema, altrimenti la soluzione iniziale viene dichiarata minimo locale.

Data la mancanza di garanzie, per quanto riguarda la qualità delle soluzioni fornite da algoritmi euristici, la valutazione delle loro performance è ottenta empiricamente, confrontandone i risultati computazionali ottenuti, su insiemi di problemi di benchmark. Attualmente, la maggior parte dei problemi reali di routing vengono risolti utilizzando algoritmi euristici, a causa della loro velocità e la loro capacità di gestire grandi istanze. Tradizione vuole che, nel risolvere problemi reali, venga impiegata una funzione obiettivo di tipo gerarchico. La prima priorità è di minimizzare il numero dei veicoli utilizzati e la seconda è di minimizzare il costo dei percorsi effettuati. Ciò differisce dagli algoritmi esatti, i quali non considerano il numero di veicoli usati nella funzione obiettivo.

2.2.4 Metodi metaeuristici

Negli ultimi dieci anni la ricerca sui metodi euristici per il CVRP e il VRPTW si è concentrato sulle metodologie metaeuristiche. Gli algoritmi

metaeuristici possono essere considerati un naturale miglioramento degli euristici classici. Gli approci metaeuristici esplorano le regioni più promettenti dello spazio delle soluzioni, spostandosi iterativamente tra i vicini delle soluzioni trovate, finché esse non soddisfano un certo criterio d'interruzione. Alcuni metaeuristici prendono in considerazione anche soluzioni non ammissibili e fasi non migliorative.

La qualità delle soluzioni prodotte da questi algoritmi è generalmente molto più grande di quella ottenuta con le euristiche classiche ma, in alcuni casi, ciò comporta tempi di calcolo più lunghi. Si consideri inoltre che l'utilizzo di metaeuristici è spesso dipendente dal contesto, per questo si richiede una raffinata parametrizzazione delle procedure affinché questi si adattino alle varie situazioni [TV02].

È possibile distinguere tre classi di metaeuristici:

- *Metodi a ricerca locale* come simulated annealing, deterministic annealing, e tabu search; questo tipo di algoritmi prende in considerazione una soluzione iniziale ammissibile e, ad ogni iterazione, passa ad una soluzione appartenente all'insieme dei *neighbors* (vicini) finché non viene soddisfatto un certo criterio di arresto. Soluzioni trovate durante iterazioni successive non sempre sono migliorative, ciò favorisce la diversificazione ed evita lo stallo intorno ai minimi locali.

- *Algoritmi genetici*, o *population search*; questo genere di algoritmi considera ad ogni iterazione una popolazione di soluzioni. Ogni popolazione è derivata dalla precedente tramite ricombinazione delle soluzioni migliori ed eliminazione delle peggiori.

- *Sistemi esperti*, come le reti neurali ed *ant systems*; nelle reti neurali vengono sfruttati meccanismi di auto-regolazione dei coefficienti interni per individuare soluzioni migliori. Gli *ant systems* invece producono, ad ogni iterazione, numerose soluzioni, derivate a loro volta, da informazioni raccolte durante le iterazioni precedenti.

2.2.5 Metaeuristici a ricerca locale

Il concetto di ricerca locale è centrale nella discussione delle metodologie metaeuristiche per la risoluzione del VRP. Queste metodologie prevedono il calcolo di una soluzione iniziale, la quale viene via via migliorata, fino alla soddisfazione di un certo criterio d'arresto.

Sia t una generica iterazione di un algoritmo metaeuristico, e sia x_t la soluzione ad essa associata. Sia $N(x_t)$ l'insieme dei vicini (*neighbors*) di x_t, individuati tramite una funzione *neighborhood* definita. Le funzioni di *neighborhood* possono essere classificate come segue:

- **Intra-route**, in cui viene modificata una singola route della soluzione;
- **Inter-route**, in cui uno o più clienti vengono spostati fra due o più routes;
- **LNS**, cioè Large neighborhood search, per i quali si introducono gli operatori *destroy* e *repair*. L'operatore *destroy* disintegra una soluzione, ad esempio rimuovendo uno o più clienti dalle routes, mentre l'operatore *repair* ricostruisce una soluzione completa, reinserendo i clienti precedentemente rimossi dalle routes. Questi due operatori vengono ripetuti iterativamente per un numero massimo di iterazioni, oppure fino a quando non è possibile migliorare la corrente soluzione.

Individuato l'insieme dei $N(x_t)$, dei *neighbors* della soluzione x_t, si procede con la selezione della soluzione x_{t+1}. A questo scopo si illustreranno tre metodologie che comprendono simulated annealing, deterministic annealing, e tabu search.

Indichiamo con $f(x_t)$ il costo della soluzione x_t, e sia $f(x_{t+1})$ il costo della soluzione $x_{t+1} \in N(x_t)$ calcolata all'iterazione $t+1$.

Nel **simulated annealing** viene selezionata casualmente una soluzione x nell'insieme $N(x_t)$ e, se $f(x) \leq f(x_t)$, allora $x_{t+1} := x$, altrimenti,

$$x_{t+1} := \begin{cases} x & \text{con probabilità } p_t, \\ x_t & \text{con probabilità } 1 - p_t, \end{cases}$$

dove p_t è una funzione di probabilità calcolata come

$$p_t = \exp\left(-\frac{f(x) - f(x_t)}{\theta_t}\right)$$

e θ_t è una funzione di t chiamata temperatura all'iterazione t. Tale funzione è parametrizzata nell'algoritmo e decrescente con t, allo scopo di ridurre la probabilità di selezionare soluzioni peggiorative al crescere di t. Considerando che non vale necessariamente $f(x_{t+1}) < f(x_t)$, gli algoritmi di simulated annealing devono implementare meccanismi per evitare cicli infiniti.

I criteri d'arresto valutati possono comprendere:

- il numero di cicli durante i quali il valore $f(x*)$, cioè il costo della migliore soluzione attuale, non diminuisce;
- il numero di mosse accettate, inteso come il numero di volte in cui la soluzione x_t viene sostituita con la soluzione x trovata casualmente, il quale deve superare una certa percentuale prefissata durante le ultime iterazioni;
- il numero totale di iterazioni eseguite.

Il **deterministic annealing** si differenzia dal *simulated annealing* nei criteri di scelta delle soluzioni, le quali vengono selezionate in modo deterministico anziché casuale. Tali criteri possono essere di tipo:

- *threshold accepting*, per il quale la soluzione x è accettata se $f(x) < f(x_t) + \theta_1$, dove θ_1 è un parametro controllato dall'utente;
- *record to record travel*, per il quale, definita $x*$ come la miglior soluzione trovata, x viene accettata se $f(x) < \theta_2 f(x*)$. Anche in questo caso θ_2 è un parametro controllato dall'utente.

Nel **tabu search**, la ricerca è deterministica e avviene all'interno di un sottinsieme dei vicini $N(x_t)$. Ad ogni soluzione valutata durante un'iterazione, viene assegnato un attributo che ne proibisce la selezione per un certo intervallo di iterazioni; tali soluzioni diventano così soluzioni *tabu*. L'iterazione termina con la selezione della migliore soluzione non proibita in $N(x_t)$. Grazie a questo meccanismo vengono evitati cicli infiniti e stazionamenti intorno a minimi locali.

I migliori algoritmi metaeuristici, che usano le funzioni neighborhood sopra descritte, sono dovuti a Bent and Van Hentenryck [BH04], Mester and Bräysy [MB05], Pisinger and Ropke [PR07], Lim and Zhang [LZ07], Hashimoto, Yagiura and Ibaraki [HHI08], Hashimoto and Yagiura [HY08], Ibaraki et al. [IIN$^+$08], Repoussis, Tarantilis and Ioannou [PRI09], Prescott-Gagnon, Desaulniers and Rousseau [EPGR09], Nagata and Bräysy [NB09], Nagata, Bräysy and Dullaert [YND10], Vidal et al. [TVP11], Blocho and Czech [BC12], CM Cordeau and Maischberger [CM12].

Capitolo 3

Approccio alla fase realizzativa

L'approccio alla fase realizzativa pone problematiche di tipo organizzativo relative alla gestione del team, dei cicli di sviluppo, delle metodologie con le quali questi vengono affrontati, e agli strumenti di sviluppo impiegati.

Saranno così introdotti il framework *Scrum*, tramite il quale vengono organizzati i cicli di lavoro, le pratiche di *TDD* e *CI*, con le quali si articolano le attività di sviluppo, e la suite di strumenti messi a disposizione dei singoli sviluppatori.

3.1 Metodologie di sviluppo, *Scrum*

Per la gestione dei cicli di sviluppo di *NST* è stato scelto di operare secondo una modalità agile, facendo uso del framework lightweight **Scrum**. Scrum consiste in un insieme di pratiche e regole interrelate che ottimizzano l'ambiente di sviluppo, sincronizzando i requisiti emergenti dal cliente con prototipi iterativi, mantenendo ridotto l'overhead organizzativo (da cui il termine lightweight, cioè leggero).

Il metodo *Scrum* segue un approccio incrementale allo sviluppo del software, organizzando il lavoro in *Sprint*, cioè rapidi cicli di lavoro di 2-4 settimane, dove si crea un incremento di prodotto potenzialmente consegnabile, composto di funzionalità eseguibili e testate. L'idea di base è dunque quella di

apportare continue modifiche al prodotto al fine di farlo evolevere secondo un approccio empirico; l'utilizzo di *Scrum* si rivela particolarmente utile laddove le specifiche del software rendono complicata una pianificazione preventiva delle operazioni di progettazione e sviluppo, come nel caso di *NST*.

La reale durata degli *Sprint* è decisa dal team di lavoro, in base alle attività prese in considerazione per il ciclo di lavoro in esame. L'identificazione delle attività che fanno parte di uno Sprint derivano dalla selezione di una serie di documenti dette *storie*. Le *storie* sono brevi estratti del documento delle specifiche funzionali, stilato durante la fase preliminare di analisi, e descrivono sinteticamente i bisogni del cliente. Le *storie* possono comprendere casi d'uso individuati in fase di analisi, oppure migliorie e interventi incrementali richiesti in corso d'opera. L'insieme delle storie è detto *product backlog*.

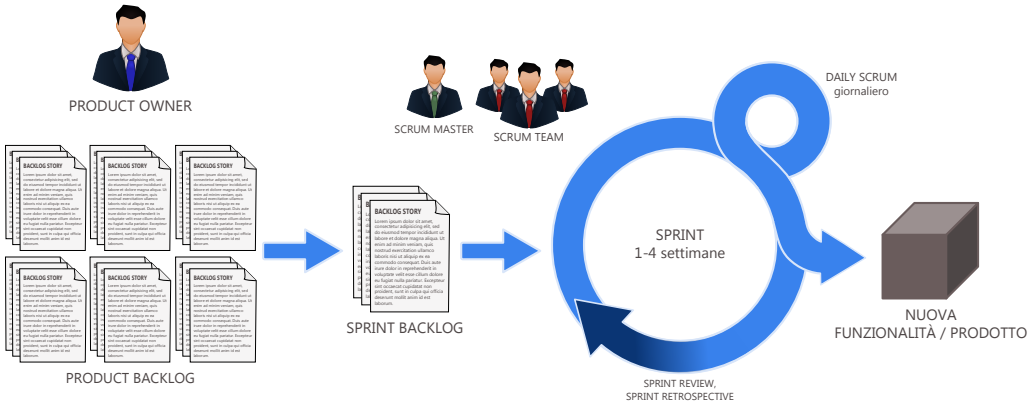
Gli attori principali di questo approccio sono:

- gli sviluppatori, o **scrum team**;
- un coordinatore detto **scrum master**;
- un **product owner** che rappresenta il cliente finale durante i confronti sui vari aspetti dell'analisi dei requisiti.

Compito del *product owner* è anche quello di assegnare alle varie *storie* una priorità in base alle esigenze dei committenti (*stakeholders*), costituendo di fatto il product backlog.

Prima dell'inizio di uno sprint si procede ad uno **sprint planning meeting**, cioè una riunione del team di sviluppo, presieduta dallo *Scrum master* e in presenza del *product owner*. Durante gli *sprint planning meeting* vengono selezionate, dal product backlog, le storie sulle quali organizzare le attività di sviluppo. Tali attività possono comprendere analisi, progettazione o implementazione; in pratica, ogni sprint è composto da fasi di analisi, progettazione ed implementazione, limitate ad un certo subset di specifiche del programma.

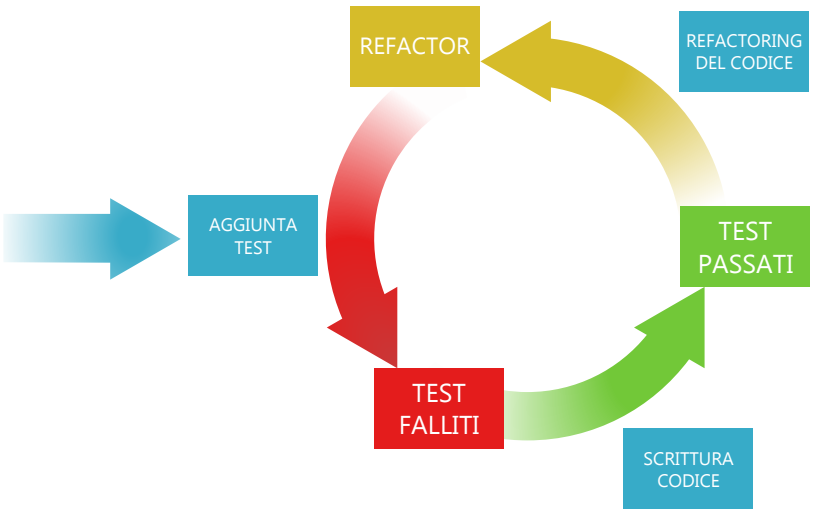
Figura 3.1: Scrum Agile



Al termine di ogni sprint il team si riunisce per una *sprint review*, durante la quale, ogni sviluppatore effettua una demo del prodotto elaborato durante lo sprint. Lo scopo delle review è quello di permettere allo *Scrum master* di individuare eventuali intoppi o impedimenti che influiscono sul lavoro del gruppo o dei singoli membri dello stesso. *Scrum* prevede inoltre alcune retrospettive (*Sprint retrospective*), il cui scopo è quello di individuare potenziali miglioramenti, individuando i punti forti e i punti deboli nello sprint, in un'ottica di continuo miglioramento del processo di delivery del software.

Nel caso di *NST* sono stati individuati due analisti esperti allo scopo di coprire i ruoli di *Scrum master* e *product owner*; per quest'ultimo ruolo in particolare è richiesta una notevole esperienza nel dominio dei trasporti, in quanto l'analista deve poter prevedere le richieste di clienti che operano in campi differenti della logistica. La scelta di un analista interno come *product owner* è dovuta alla natura generica del software; questo, infatti, non è sviluppato in modo specifico, cioè per coprire le esigenze di un singolo cliente, ma piuttosto come prodotto di larga diffusione, mirato cioè, alla risoluzione di problemi riconducibili alla pianificazione dei trasporti ed, eventualmente, adattabile a singoli clienti tramite l'introduzione di moduli verticali.

Figura 3.2: Fasi del Test Driven Development



3.2 Test Driven Development e Continous Integration

Il *Test Driven Development* (abbreviato in *TDD*) è un processo di sviluppo del software, nel quale la stesura delle procedure funzionali, è preceduta (driven) dalla stesura di test automatici. Il TDD si articola in cicli di sviluppo composti da tre fasi:

- **Red**, cioè la prima fase, prevede che lo sviluppatore scriva un test automatico per la funzionalità da implementare. Tale test fallirà necessariamente in quanto creato preventivamente all'implementazione della funzionalità stessa.
- **Green**, durante questa seconda fase lo sviluppatore scrive la quantità minima di codice necessaria al superamento del test.
- **Refactor**, la terza fase prevede il refactoring del codice, cioè la sua ristrutturazione.

I vantaggi apportati dal TDD riguardano l'affidabilità, la solidità e la pulizia del codice. La solidità è garantita dai test stessi che vengono ge-

neralmente ripetuti in seguito ad ogni build, garantendo così che il nuovo codice introdotto non abbia ripercussioni su procedure scritte precedentemente. La necessità di scrivere un test prima della stesura del codice aiuta lo sviluppatore nella comprensione delle specifiche, facendo inoltre emergere preventivamente eventuali errori, incomprensioni o inesattezze, nelle specifiche stesse, migliorando quindi l'affidabilità del codice. Infine i continui refactoring danno luogo ad un codice più pulito.

La **Continous Integration** (abbreviato *CI*) è una pratica, complementare al TDD, che consiste nel frequente allineamento degli ambienti di sviluppo dei singoli sviluppatori, con la mainline, cioè l'ambiente condiviso. Scopo del processo di Continuous Integration è aumentare la qualità del software eseguendo in modo continuo, durante tutto il ciclo di sviluppo, i controllo di qualità e correttezza del codice, che solitamente sono rimandati dopo la fase di rilascio. La realizzazione di un progetto software complesso richiede lo sviluppo di numerosi moduli software, legati tra loro e interdipendenti.

In questo contesto di continua evoluzione è fondamentale mantenere il controllo delle variazioni apportate al codice verificando che:

- siano rispettate le dipendenze con gli altri moduli;
- sia sempre possibile realizzare una build di tutto il sistema;
- i test automatici, associati ai singoli moduli o a livello di sistema, abbiano successo.

La Continuous Integration prevede innanzitutto la centralizzazione del repository dei sorgenti, tramite l'utilizzo di un sistema di versioning dei sorgenti, e l'utilizzo di un sistema, detto **CI server**, che effettui in automatico la compilazione di tutto il codice e l'esecuzione dei test. È richiesto agli sviluppatori di ridurre il lasso di tempo in cui tengono in locale una copia modificata dei sorgenti, per limitare i rischi di conflitti e disallineamenti rispetto la versione del repository centrale. Ad ogni aggiornamento dei sorgenti effettuato dagli sviluppatori, il sistema: effettua una compilazione dell'ultima versione

dei sorgenti, esegue in un ambiente di collaudo tutti i test automatici, unitari, e di integrazione, e notifica agli sviluppatori l'esito della compilazione e dei test.

La scrittura di test sulle singole unità software (*unit tests*) avviene tramite l'utilizzo del Test Framework di Microsoft, ed è effettuata per tutte le principali procedure. Un esempio di unit test è dato dal frammento di codice 3.2, nel quale è possibile notare l'utilizzo di alcune annotazioni che marcano i metodi di test. Tali annotazioni possono influenzare l'ordine o le modalità con le quali i metodi di test vengono richiamati.

```
[TestClass]
public class UserProfileTest
{
  [TestMethod]
  [TestCategory("UILayer"), TestCategory("UserAccount")]
  public void RegisterNewUser()
  {
    //creazione del componente da testare
    AccountController controller = new AccountController();
    //Creazione di modello dati
    RegisterModel rm = new RegisterModel();
    rm.Password = rm.ConfirmPassword = "PasswordComplessa123456789";
    rm.UserName = "Mario Rossi";
    //test del metodo di registrazione utenti
    ViewResult result = controller.Register(rm) as ViewResult;
  }
}
```

3.3 Strumenti di sviluppo

Al fine di fornire ai membri del team un ambiente di sviluppo uniforme, sono stati selezionati una serie di strumenti, per lo svolgimento delle attività di sviluppo del software. Gli strumenti individuati comprendono: macchine virtuali, ambienti di sviluppo integrati (*IDE*), sistemi di versioning, *CI* server, etc.

I principali vantaggi, derivanti dall'utilizzo di questi strumenti, sono:

- **condivisione del know how**

- **interoperabilità**, grazie all'utilizzo di formati dati ben definiti e metodologie di sviluppo comuni a tutti i membri del team;

- **fast recovery**, cioè recupero veloce da eventuali malfunzionamenti, minimizzando il tempo di improduttività.

- **riduzione dei tempi allineamento** per l'eventuale introduzione di nuovi membri nel team di progetto.

3.3.1 Macchina virtuale di sviluppo

Ai membri del team di sviluppo del progetto *NST*, è stata fornita una macchina virtuale, basata su **Microsoft Windows7**, operata tramite l'utilizzo della piattaforma **VMWare player**. VMware Player è un software freeware di virtualizzazione, il quale permette di creare ed eseguire macchine virtuali, allocando risorse della macchina fisica in modo arbitrario. La macchina virtuale è preconfigurata ed ospita già il software di sviluppo, fornendo agli sviluppatori, un'ambiente funzionante e standardizzato per l'approccio al progetto.

I vantaggi addotti comprendono:

- in breve tempo si permette allo sviluppatore di cominciare a lavorare sul prodotto, abbattendo i tempi di installazione degli strumenti di sviluppo; stesso discorso in caso di danneggiamento, e sostituzione, della macchina fisica;

- eventuali malfunzionamenti o infezioni di malware, sulla macchina virtuale, non compromettono la macchina fisica, la quale può ospitare una nuova macchina virtuale funzionante, senza costringere gli sviluppatori a interventi tecnici di ripristino;

- la macchina virtuale non impone l'utilizzo di uno specifico sistema operativo, o particolare hardware, sull'host, adattandosi in pratica ad ogni macchina fisica messa a disposizione degli sviluppatori;

- l'applicazione di pratiche di pair programming (programmazione in coppia) è particolarmente facilitata, in quanto, gli sviluppatori che ricoprono il ruolo di observer (osservatore), riconoscono immediatamente l'ambiente di sviluppo e possono collaborare senza particolari sforzi.

Periodicamente vengono effettuati degli snapshot delle macchine virtuali, cioè ne viene salvato lo stato, i dati e la configurazione hardware, allo scopo di mantenere un backup sempre aggiornato e funzionante.

3.3.2 Versioning del codice sorgente

Il controllo di versione (o **versioning**) è, in informatica, la gestione di versioni multiple di un certo insieme di informazioni, riferito in genere al codice sorgente di un applicativo. Per effettuare il controllo di versione di un progetto è necessario configurare un **Repository**, cioè una banca dati centralizzata, che ne ospita i documenti. Il repository è organizzato in **Revision**, cioè revisioni, o versioni delle informazioni, memorizzate in catena di *modifiche*. I moderni sistemi di versioning prevedono che ogni contributore intervenga su un set di dati proprio, detto **copia locale**, propagando le modifiche effettuate, al repository centralizzato, solo in un secondo momento.

Un sistema di controllo di versione permette di applicare le pratiche di versioning a progetti informatici, fornendo strumenti per effettuare:

- **commit**, cioè propagazione delle modifiche effettuate sui documenti verso il repository centralizzato;

- **update**, cioè la sincronizzazione della copia locale con il repository;

- **merge**, cioè l'integrazione, in una versione unificata, di modifiche provenienti da versioni concorrenti;

- **import**, cioè l'importazione dell'intera copia locale sul repository;
- **export**, cioè l'esportazione, di una versione del progetto, dal repository verso una destinazione esterna;
- individuazione e risoluzione dei **conflitti**, che avviene quando modifiche concorrenti sullo stesso set di dati si sovrappongono;

L'utilizzo di un sistema di versioning è fondamentale per le pratiche di *CI*. Il sistema di versioning impiegato per il progetto *NST* è **Subversion**, o semplicemente *SVN*, il quale permette di effettuare:

- versioning di file di testo e file binari;
- versionamento delle directories, mantenendo traccia della struttura con la quale sono organizzati i sorgenti;
- commits atomici, cioè l'applicazione di un insieme di modifiche in una singola operazione, evitando la generazione di versioni incomplete;
- accesso al repository tramite protocollo HTTP;
- **branching**, cioè la separazione di intere versioni dalla mainline del progetto;
- **etichettamento** o tagging delle versioni, allo scopo di distinguere particolari revisioni;

Il modello di versionamento applicato da *Subversion* è detto **Copy-Modify-Merge**, il quale a differenza del modello *Lock-Modify-Unlock*, prevede che i file in fase di modifica rimangano disponibili anche agli altri sviluppatori, rimandando l'eventuale gestione dei conflitti alla fase di propagazione delle modifiche. Questo approccio, detto anche approccio *ottimistico*, è particolarmente efficace nello sviluppo in team, in quanto non ostacola il lavoro parallelo sui sorgenti.

3.3.3 IDE, ambiente di sviluppo integrato

L'IDE impiegato per lo sviluppo di *NST* è **Microsoft Visual Studio 2012**. Visual Studio fornisce gli strumenti necessari alla stesura del codice, alla compilazione, all'integrazione con librerie esterne, ed all'organizzazione dei sorgenti. La versione *2012* di Visual Studio introduce l'utilizzo dell'interfaccia Metro (5.2), supporto ad *HTML 5* e *CSS 3*, così come per le ultime tecnologie *ASP.NET*, tra le quali *MVC4*, largamente impiegate per lo sviluppo di *NST* (5.4).

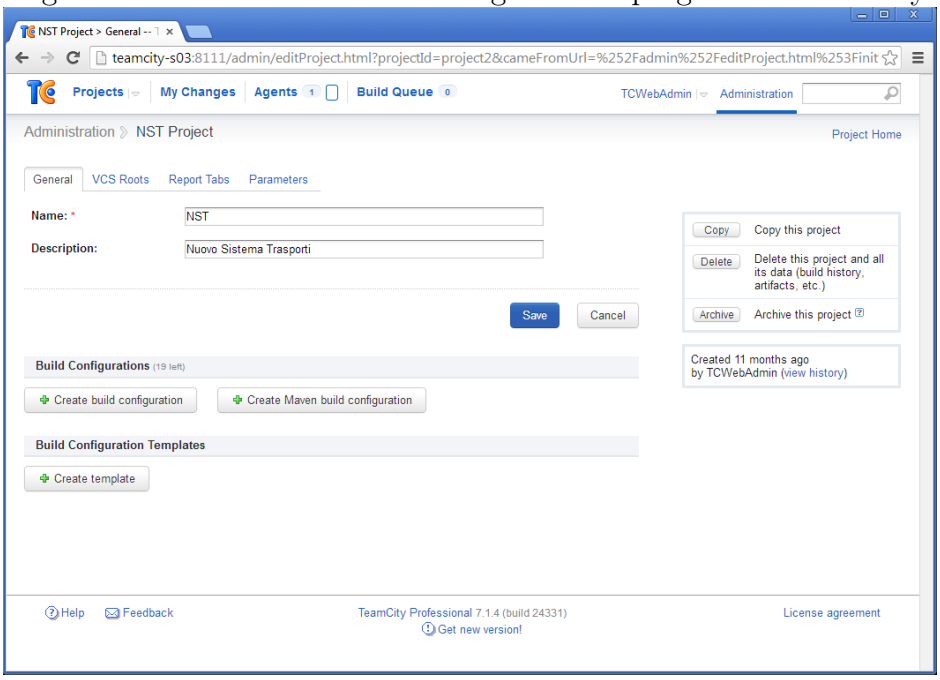
I sorgenti dell'applicazione sono organizzati in una solution, nella quale sono organizzati a loro volta i vari progetti che costituiscono i vari layer applicativi (4.1.1). Visual Studio fornisce è ben integrato con il sistema di versioning del codice *SVN*, permettendo di mantenere rapidamente allineata l'intera soluzione software.

3.3.4 CI server

Il sistema scelto per la gestione del processo di CI in *NST* è *TeamCity* di JetBrains. *TeamCity* è un software scritto in Java che espone un'interfaccia web tramite la quale è possibile configurare i processi di compilazione, test e interfacciamento con sistemi di versioning.

Integrabile con Visual Studio, NuGet (gestore di package per .NET, integrato in Visual Studio) e SVN, *TeamCity* permette di eseguire, in maniera automatizzata, build e test per ogni commit sulla mainline. Il software fornisce inoltre strumenti per l'ispezione del codice e l'individuazione di codice duplicato. In caso di fallimento dei tests, *TeamCity* è in grado di segnalare inviare un report dettagliato agli sviluppatori via e-mail.

Figura 3.3: Una schermata di configurazione progetto di TeamCity



Capitolo 4

Architettura del sistema

Definiamo *architettura* di un software l'organizzazione basilare di un sistema, il quale è rappresentato dalle sue componenti, dalle relazioni che esistono tra di loro e l'ambiente circostante, e dai principi che governano la sua progettazione ed evoluzione [IEE00].

Tra le molte roblematiche da affrontare, nello sviluppo dell'architettura di un'applicazione aziendale, le più importanti sono:

- La *stratificazione* dell'applicazione, cioè la suddivisione delle funzionalità in aree indipendenti e intercomunicanti tramite interfacce;
- La strutturazione della *logica applicativa*;
- *La persistenza* degli oggetti attraverso un database;
- La gestione dello *stato delle sessioni* in un ambiente stateless;
- La strutturazione dell'*interfaccia utente*;
- I principi di *distribuzione*;

Si devono inoltre considerare problemi riguardanti la validazione dei dati, la tipologia di comunicazione sincrona/asincrona, la gestione dei dati di interscambio, la sicurezza, la gestione di errori/eccezioni, la suddivisione dei dati e l'integrazione con applicativi esterni [Fow02].

In questo capitolo si introdurranno alcune delle soluzioni architetturali impiegate e come esse risolvono le problematiche illustrate.

4.1 Stratificazione e Multi-tier architecture

La stratificazione è una tecniche basata sulla filosofia *divide et impera* e viene usata dai software designers per suddividere sistemi software complessi. Le architetture software basate sulla stratificazione sono dette *n-tier*, o *multi-tier*, oppure *multi-strato*. Un esempio pratico, e ampiamente diffuso, di questa pratica è rappresentato dallo standard *ISO/OSI (Open System Interconnection)*, cioè il modello di pila protocollare dalla quale sono derivate le reti moderne. Internet stessa è basata sulla suite di protocolli *TCP/IP*, la quale può a sua volta essere rappresentata tramite il modello OSI.

Questo tipo di suddivisione in strati (altrimenti detti layers) comporta una serie di vantaggi:

- ogni layer rappresenta un insieme coerente di funzionalità, le quali rispondono ad un insieme limitato di problematiche;
- ogni layer incapsula l'implementazione dei servizi forniti, nascondendo la propria complessità agli strati adiacenti;
- le dipendenze tra gli strati sono minimizzate, facendo sì che ogni layer sia intercambiabile, al solo costo di rispettare le interfacce degli strati adiacenti;
- la suddivisione in layer incoraggia la riusabilità, permettendo a volte, di riutilizzare gli strati in altri applicativi con una quantità minima di interventi sul codice.

L'applicazione della stratificazione introduce anche alcune complicazioni inerenti le *prestazioni* del sistema e la possibilità di *modifiche a cascata*. I problemi di prestazioni sono principalmente legati alla necessità di passare da una rappresentazione dei dati, strettamente legata ad uno strato, verso

un'altra, fruibile da uno strato differente. Sebbene tale operazioni spesso si riduce ad una semplice mappatura, tra diverse istanze di oggetti, diventa onerosa al crescere della quantità di dati in transito dallo strato. Il problema delle *modifiche a cascata* dipende, invece, dall'impossibilità di incapsulare tutte le operazioni in modo ottimale. Il risultato che ne deriva è che modifiche, anche semplici e inerenti un singolo strato, si propagano agli strati adiacenti. Un esempio di questo è l'aggiunta, in un'entità della base dati, di un campo che vogliamo mostrare anche nell'interfaccia utente, e che dovremo quindi aggiungere in tutti gli strati intermedi.

4.1.1 La stratificazione di NST

NST presenta un'architettura *multi-strato*, nella quale ogni livello opera indipendentemente dagli altri, incapsulando i dettagli di implementazione del servizio fornito, ed esponendo interfacce, metodi ed oggetti, tramite il quale può essere interrogato dagli altri livelli. Questa pratica favorisce la *separation of concerns*, cioè il principio per cui ogni componente software deve potersi concentrate sul suo obiettivo primario, senza preoccuparsi di come sono implementati i servizi dal quale dipende. Il vantaggio derivante da questa separazione è un incremento della manutenibilità, della modularità, della scalabilità e dell'incrementabilità del software.

L'architettura di *NST* è una generalizzazione della classica architettura a tre strati. L'architettura a tre strati, anche detta *3-tier architecture*, suddivide le funzionalità del software in tre differenti unità:

- interfaccia utente, o *UI layer*, la quale si occupa dell'interazione con l'utente e della presentazione dei dati;
- lo strato logico, o *business layer*, il quale si occupa di incapsulare la logica decisionale, i comportamenti e i calcoli sulle entità del dominio applicativo;
- lo strato di persistenza, o *data layer*, il quale supporta la memorizzazione e il recupero dei dati dell'applicativo.

Il *data layer* di *NST* è composto da un database relazionale, o *RDBMS*, e da un *ORM* (Object Relational Mapping), cioè da un software che favorisce l'integrazione tra un database relazionale e la struttura ad oggetti del sistema. Altro vantaggio indotto dall'utilizzo di un *ORM* è la possibilità di astrarre le caratteristiche implementative dello specifico *RDBMS* utilizzato, rendendo di fatto, la base dati intercambiabile.

Il *business layer*, oltre che fornire servizi relativi alle entità del dominio verso lo strato di interfaccia, incapsula le interfacce di accesso a servizi esterni e le logiche decisionali.

L'*UI layer* comprende tutte le procedure, i modelli dati e i servizi che permettono l'interazione cn l'utente; trattandosi di interfaccia web, questo strato è ulteriormente suddiviso in: server-side, cioè l'insieme di componenti software elaborati dall'infrastruttura server, e client-side, cioè quell'insieme di script, dati e documenti di layouts eseguiti dal client.

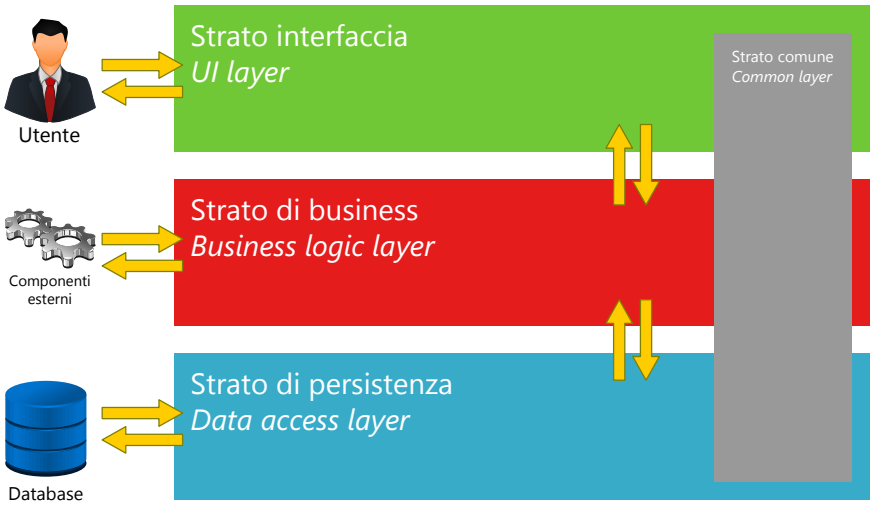
NST comprende infine uno strato verticale, detto *common layer*, che contiene una serie di classi, le quali funzionalità sono condivise tra tutti gli altri layer. Questo strato implementa funzionalità di uso comune come l'infrastruttura di logging, metodi di utilità per la manipolazione dei dati, eccezioni personalizzate, ed una serie di classi per l'integrazione di plugins e moduli aggiuntivi. Il common layer comprende infine un insieme di oggetti leggeri, detti *Data transfer objects* o DTO, utili per il trasferimento di entità complesse all'interno e all'esterno dell'applicativo.

Ognuno di questi layer è compilato come assembly indipendente ed espone agli altri layer solamente i metodi pubblici che gli permettono di interfacciarsi con essi.

4.2 Pattern di sviluppo

In questo capitolo sono presentati i principali *design pattern* impiegati nella progettazione e sviluppo di *NST*. Il concetto di pattern è definito come *una soluzione progettuale generale a un problema ricorrente.* Ogni pattern è

Figura 4.1: Architettura n-tier di NST

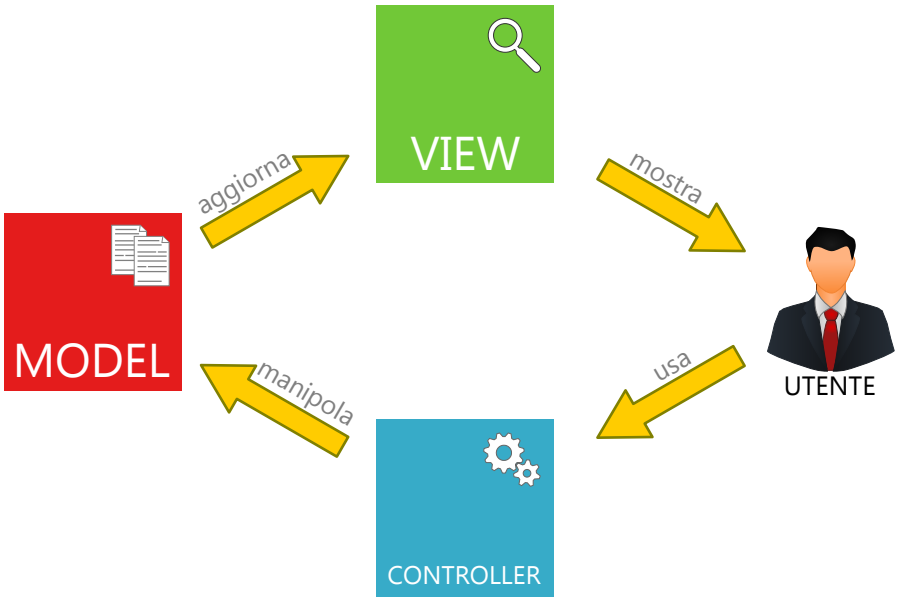


caratterizzato da un *nome* che lo identifica, da una *classe di problemi* ai quali può essere applicato, dalla descrizione della *soluzione* proposta, ed infine, da *vincoli e risultati* introdotti dalla sua applicazione. I pattern di sviluppo si possono suddividere in tre macroaree: *pattern architetturali*, *design pattern* e *pattern di implementazione*, in base all'ambito di applicazione dei pattern che ne fanno parte. I *pattern architetturali* sono pattern di alto livello e operano sulla struttura del sistema, individuandone insiemi di responsabilità omogenee e relazioni di tra esse, allo scopo di partizionarle tra le varie componenti. Un esempio di pattern architetturale è la *n-tier architecture*, esposta in precedenza in questo capitolo. I *design pattern* sono strettamente legati al paradigma object-oriented e si riferiscono alla struttura, e interazione, degli oggetti del sistema. I *pattern di implementazione* sono invece i pattern di basso livello, legati a specifiche tecnologie, che mirano a sfruttare i punti di forza le caratteristiche peculiari di una certa piattaforma [Gol13].

La descrizione del problema affrontato da un pattern comprende una panoramica sulle situazioni cui esso è applicabile, la quale, può essere più o meno specifica e deve comprendere le precondizioni necessarie alla sua applicazione.

La soluzione proposta dal pattern è riassunta dal suo nome e viene de-

Figura 4.2: La struttura Model-View-Controller



scritta come la configurazione degli elementi da esso costituiti affinchè il problema sia risolto. La soluzione proposta deve astrarre il problema, senza addentrarsi nelle specifiche di implementazione.

I vincoli imposti da un pattern sono spesso l'argomento primario per determinarne la scelta, in quanto determinano l'impatto della soluzione prodotta sul resto del sistema.

4.2.1 Model View Controller, MVC

Il pattern *MVC*, Model-View-Controller, è un *pattern architetturale* legato alla programmazione ad oggetti, nato alla fine degli anni 70 come framework per Smalltalk, sviluppato da Trygve Reenskaug. Da allora, il framework *MVC*, ha giocato un ruolo importante nel design e nello sviluppo di framework per interfacce utente [Fow02]. Il problema affrontato da questo pattern è quello di separare le logiche di business dai dati, e dalla loro rappresentazione.

La struttura di *MVC* è caratterizzata da tre tipologie di oggetti:

Figura 4.3: Ripartizione delle funzionalità in MVC

- il **Model**, rappresenta le informazioni del dominio, contiene i dati, e quindi lo stato di entità del sistema;
- la **View**, che mostra i dati del model adattandosi al suo stato;
- il **Controller**, gestisce l'interazione con l'utente, ricevendone i comandi e manipolando il model di conseguenza;

In questo modo la logica di business viene ripartita tra model e controller, model e view si suddividono il problema della rappresentazione delle entità, mentre view e controller compongono l'interfaccia utente gestendo input ed output del sistema.

In *NST* questo pattern è impiegato, tramite il framework *MVC4* di microsoft, per l'implementazione dell'interfaccia lato server, della quale si tratterà in modo più approfondito nel capitolo 5.

4.2.2 Model View View-model, MVVM

Il pattern *MVVM* è una specifica implementazione del pattern *MVC* secondo il paradigma dell'*event-driven-programming*. Questo pattern è nato appositamente per lo sviluppo di interfacce utente in ambienti *.NET*, quali *WPF* e *Silverlight*, ed è stato successivamente impiegato per lo sviluppo di interfacce *HTML5*, con la nascita di librerie *Javascript* quali *KnockoutJS* e *AngularJS* [Wik14b].

Il framework MVVM individua tre componenti principali, dai quali ne deriva anche il nome:

- il **Model**, può avere due forme: secondo l'approccio ad oggetti, esso è rappresentato da un oggetto del domnio mentre, secondo l'approccio centrato sui dati, esso è rappresentato dal data-layer che fornisce i contenuti;

- la **View** rappresenta l'aspetto grafico dei dati, cioè i controlli e i dati che vengono mostrati all'utente;

- il **View model**, anche detto modello della vista, rappresenta il punto di giunzione tra vista e modello. Esso altera la vista tramite i dati del modello e passa i comandi ottenuti dalla vista al modello.

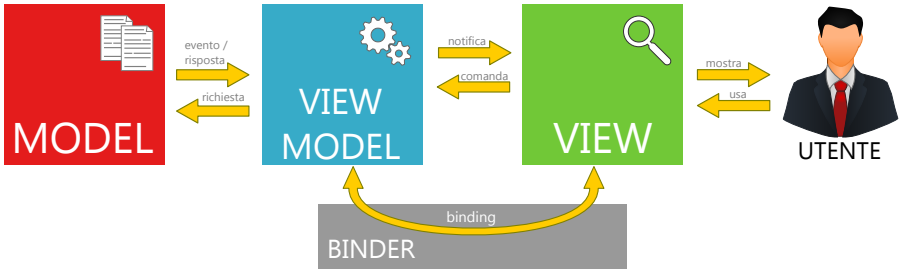
Questo tipo di pattern è implicitamente legato ad un quarto componente detto **Binder**, il quale effettua la connessione dai dati del modello verso la vista, e dai comandi della vista verso il modello. La scopo di questo componente è quello di sincronizzare il *view model* con la *view*, evitando al programmatore di scrivere codice ripetitivo.

In *NST* questo pattern viene impiegato per lo sviluppo dell'interfaccia lato client, implementata tramite l'utilizzo della libreria *KnockoutJS*. I principali vantaggi derivanti dall'utilizzo di *KnockoutJS*, e quindi del pattern *MVVM*, risiedono nella possibilità di mantenere aggiornate interfacce complesse al variare di grandi set di dati, senza dover gestire il binding manualmente. Tale vantaggio deriva dalla possibilità di effettuare un binding dichiarativo, cioè integrando le informazioni di binding direttamente nel markup della vista.

4.2.3 Provider model

Il *provider model* è un *design pattern* formulato da Microsoft per molti *ASP.NET Starter Kits* e formalizzato in *.NET versione 2.0*. Il nome del pattern è derivato dalla descrizione del problema, il quale riguarda la fornitura di funzionalità (dall'inglese *provider*, cioè fornitore) riguardanti una certa API (*Application program interface*) [How04].

Figura 4.4: Struttura di una UI basata su MVVM



Nella pratica, un provider, è un contenitore di oggetti che implementano interfacce comuni fornendo alcune funzionalità. Compito del provider è quello di selezionare quale implementazione impiegare per fornire servizi di business o accesso ai dati. La scelta di tali implementazioni viene effettuata run-time in base alla configurazione dell'applicazione, la quale a sua volta può risiedere in files o su database.

Nel business layer di *NST* è implementato un particolare provider per l'accesso ai servizi, le cui caratteristiche di funzionamento cambiano sotanzialmente tra le varie implementazioni, che li seleziona in base a certi parametri di configurazione modificabili a run-time.

Il provider pattern, inoltre, è largamente impiegato all'interno del framework *MVC*, sul quale si basa lo strato d'interfaccia lato server di *NST*. L'impiego del provider pattern in *MVC4* permette di configurare l'utilizzo di implementazioni personalizzate di servizi di autorizzazione per gli utenti, tra i quali *MembershipProvider* and *RoleProvider*, incaricati di riconoscere gli utenti, che accedono al programma, e i ruoli ai quali essi appartengono.

4.2.4 Adapter, decorator e façade patterns

I patterns *adapter*, *decorator* e *façade*, sono *design patterns* utilizzati nella programmazione ad oggetti. In genere ci si riferisce agli oggetti progettati secondo le soluzioni proposte, con lo stesso nome del pattern che li ha generati.

Un *adapter*, cioè adattatore, è un oggetto che traduce l'interfaccia di un oggetto in un'altra interfaccia compatibile. Per ottenre questo comportamento l'*adapter* espone un'interfaccia e ne implementa i metodi tramite quelli esposti da un'altra interfaccia. L'utilizzo di un *adapter* permette l'interazione tra classi che non potrebbero interagire a causa di incompatibilità di interfaccia.

Un *decorator*, invece, è un oggetto che permette di estendere (quindi decorare) le funzionalità di un oggetto, senza alterarne il funzionamento iniziale. È possibile ottenere un decorator incapsulando un oggetto, caratterizzato da una certa interfaccia, all'interno di un altro oggetto che espone un'interfaccia estesa, nella quale è inclusa l'interfaccia originale.

Il nome *façade* infine, deriva dal termine architetturale facciata, e rispecchia la soluzione proposta, la quale consiste nell'implementazione di un singolo oggetto dall'interfaccia semplice, per l'accesso ad una serie di funzionalità distribuite su più oggetti con interfacce complesse. Il *façade pattern* può essere utilizzato per raggruppare oggetti o metodi, in modo da suddividerli per semplici attività, migliorando così, leggibilità, manutenibilità e semplicità, e riducendo le dipendenze tra i moduli software. Se utilizzato in concomitanza con *adapter* o *decorator*, permette di sfruttare la potenza delle espressioni polimorfiche modificando il comportamento di alcune interfacce a run-time.

In *NST* questi patterns sono impiegati, nello strato di business, per esporre interfacce a servizi complessi, per l'accesso a servizi esterni, e per raggruppare task complessi in metodi semplici. Nel lato client dell'interfaccia utente, i patterns decorator e adapter sono utilizzati per estendere le funzionalità di alcuni view models e renderli intercambiabili e applicabili ad ambienti differenti.

Figura 4.5: Diagramma di un decorator, e un adapter, per la classe User

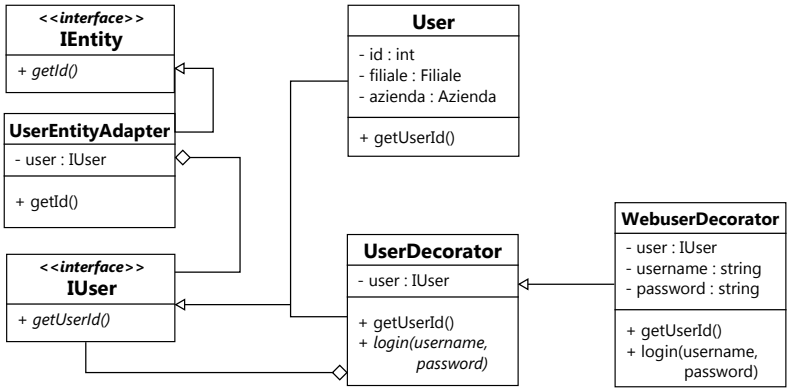
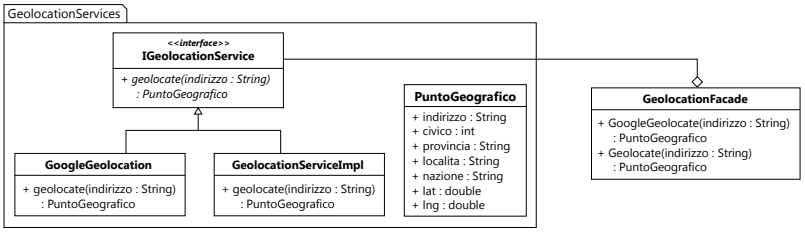


Figura 4.6: Diagramma di una façade per servizi di geolocalizzazione



Capitolo 5

L'interfaccia utente

La gestione dell'interazione con l'utente è un problema molto vasto e complesso, pertanto la progettazione di un'interfaccia utente (in inglese user interface, o UI) diventa un argomento particolarmente delicato. Considerando le caratteristiche web-based di *NST*, e l'obiettivo di trasformarlo in un software ad alta diffusione, la progettazione e lo sviluppo, dello strato di interfaccia, devono tener conto di aspetti relativi a ergonomia e usabilità [ISO08], in particolare:

- efficacia ed efficienza nel raggiungimento degli obiettifi prefissi dall'utente che utilizza il sistema;
- familiarità e facilità di apprendimento, per permettere all'utente di ambientarsi rapidamente al software;
- sicurezza e robustezza all'errore, facendo in modo che eventuali errori abbiano impatto minimo sull'utilizzo del sistema.

Vanno inoltre considerati aspetti relativi alla **portabilità** e alla **mobilità**. Si intende per portabilità la possibilità di utilizzare il software su differenti sistemi, ad esempio palmari, tablet e cellulari. Il problema della mobilità, invece, concerne la possibilità di operare il software in qualsiasi luogo e momento, consentendone la fruizione agli addetti sul campo, così come agli impiegati amministrativi che operano dall'interno del proprio ufficio.

In questo capitolo saranno introdotte le metodologie di progettazione impiegate per l'interfaccia utente di *NST* e, conseguentemente, ne verrano illustrate la struttura e le tecnologie impiegate per lo sviluppo.

5.1 Metodologia collaborativa: Wireframing, mockups e prototyping

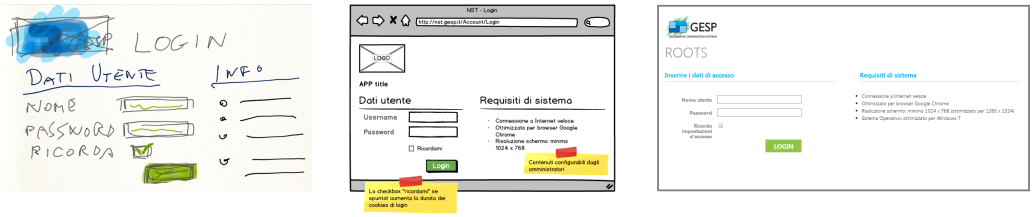
Nell'ambito del *web design* e della programmazione, si indica con **wireframe** la bozza strutturale di un applicativo web. La tecnica del *wireframing*, applicata alla progettazione di interfacce, consiste nella produzione di bozze della UI che ne riassumono gli aspetti principali, cioè:

- i contenuti, cioè cosa è necessario mostrare all'utente;
- la disposizione dei componenti dell'interfaccia;
- una breve descrizione delle interazioni con l'utente;

La produzione di un wireframe è un processo rapido che si presta particolarmente bene al processo collaborativo, fornendo ad ogni membro del team la possibilità di intervenire sulla struttura della UI. Il più delle volte un wireframe è prodotto a matita su un foglio di carta, oppure su una lavagna, e viene rifinito in un secondo momento in caso si renda necessario un confronto con il cliente.

Il passo successivo rappresenta dalla produzione di un **mockup**, cioè una rappresentazione statica, ma esteticamente dettagliata, della versione finale dell'interfaccia. In certi casi un mockup può mostrare già l'aspetto estetico finale dell'interfaccia, dando un'idea più completa del risultato finale. Un mockup permette di rappresentare la struttura delle informazioni, visualizzare alcuni contenuti e dimostrare le funzionalità di base di una UI, senza però implementarle. I mockup possono essere sufficientemente accurati da far parte della documentazione tecnica del prodotto e contengono sufficienti informazioni per la produzione di un **prototipo** funzionante.

Figura 5.1: UI design tramite wireframing, mockup e prototipazione



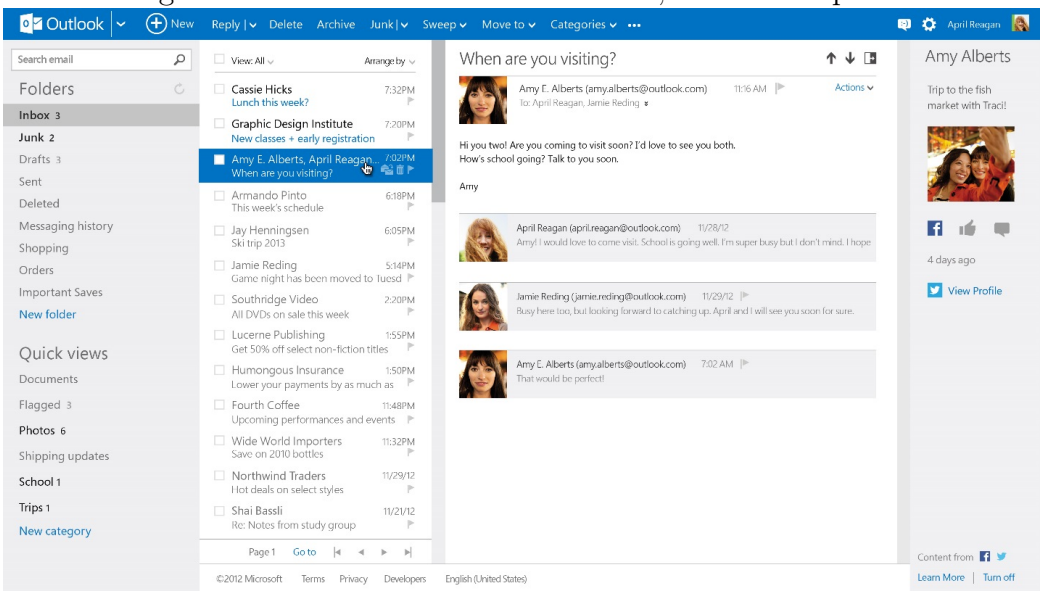
Il **prototipo** è una versione dinamica ed interattiva dell'interfaccia finale. Un prototipo non presenta necessariamente gli aspetti estetici della versione finale, ma deve piuttosto evidenziare dinamicamente i contenuti e le iterazioni della UI in modo simile al prodotto finale. Per ridurre i costi di sviluppo spesso si rimanda la fase di allacciamento del prototipo con il backend applicativo, corredandolo di un set di dati temporanei ma significativi.

5.2 UI guidelines e Metro design language

Steve Krug, information architect e autore del libro Don't make me think, dice che il layout di un sito web dovrebbe essere auto-esplicativo, cioè che non dovrebbe richiedere, all'utente che lo osserva per la prima volta, uno sforzo mentale per individuare ciò che egli si aspetta di trovare nel contesto in cui si trova [Kru06]. Questo principio, sebbene inizialmente diretto ai web-designers, è applicabile anche alle interfacce utente di applicativi web. In pratica Krug ribadisce l'importanza delle caratteristiche che un'interfaccia deve trasmettere all'utente durante il suo utilizzo. Per garantire usabilità è necessario che una UI sia caratterizzata da una certa uniformità all'interno dei vari contesti, siano essi viste di elenco, viste di dettaglio o singoli controlli. È importante inoltre, affinché l'utente possa individuare i contenuti o le funzionalità che cerca in brevissimo tempo, che l'interfaccia presenti aspetti di interazione noti all'utente, senza sorprenderlo.

Allo scopo di aiutare i designer e gli sviluppatori di *NST*, nel mantenere tale uniformità, è stato stilato un documento contenente le linee guida dei

Figura 5.2: Schermata di Outlook.com, con email aperta.



componenti e del layout dell'interfaccia. Questo documento comprende convenzioni estetiche e funzionali, che aiutano nella scelta degli elementi grafici, quali tavolozze dei colori, stili tipografici, utilizzo di bordi e separatori, e direttive per l'utilizzo di componenti per implementare task ricorrenti.

Queste linee guida di *NST* sono ispirate allo stile *Metro*[1] introdotto da Microsoft, a sua volta basato sullo *Stile tipografico internazionale* (o *Stile svizzero*), i cui principi basilari comprendono chiarezza e leggibilità. La scelta di ispirarsi a questo linguaggio di design è stata dettata, innanzitutto, dalla comunanza di obiettivi posti dal team di sviluppo durante la fase di analisi, ed inoltre, dalla volontà di fornire all'utente un'interfaccia che gli fosse familiare, ispirata all'ormai diffusa interfaccia utente di *Windows 8*.

[1] nome in codice del linguaggio di design introdotto da *Microsoft* per l'interfaccia utente di *Windows 8*

5.3 Web applications e web UI

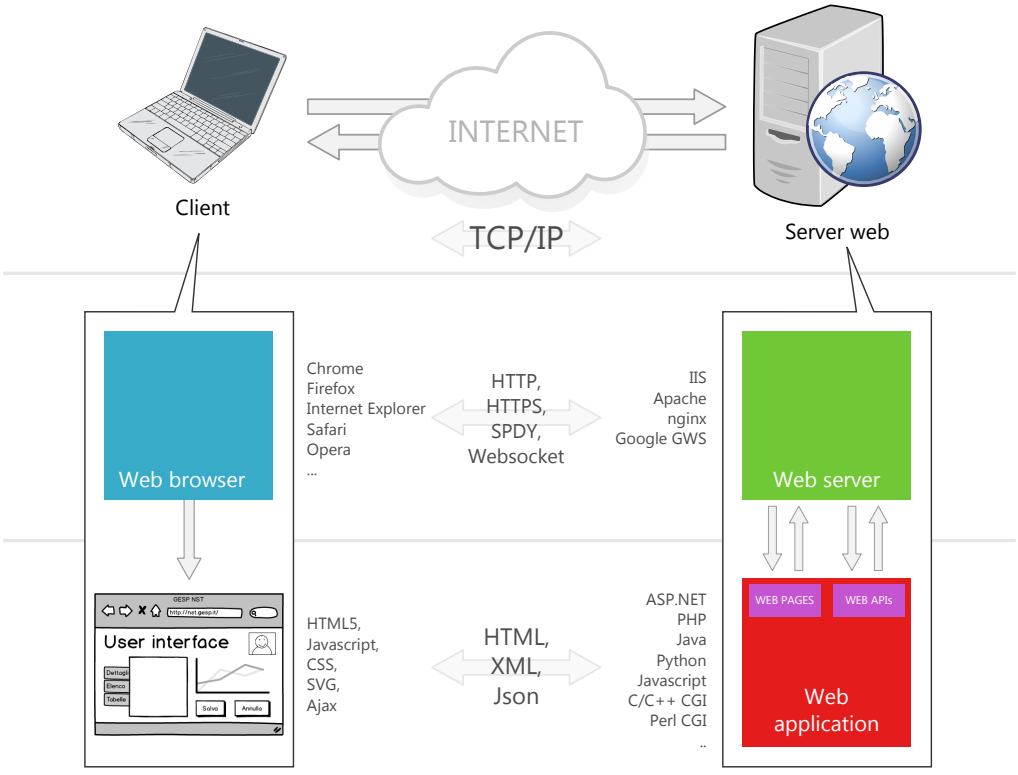
Il processo di standardizzazione del web, iniziato con la fondazione del *W3C* Consortium nell'Ottobre del 1994, ha fornito agli sviluppatori una base solida per lo sviluppo di web applications, cioè applicativi distribuiti tramite il web. La standardizzazione di HTML, CSS, SVG, Javascript, Ajax e altre tecnologie ampiamente impiegate in ambito web, ha permesso di risolvere problematiche legate all'incompatibilità tra browser, permettendo quindi ai client di impiegare un qualsiasi browser standard-compliant per l'accesso ai contenuti online, e trasformando così il web, in un ambiente ideale per la distribuzione di software.

Le web applications sono generalmente basate sul paradigma client-server. Il compito del client è innanzitutto quello di instaurare la comunicazione, inviando richieste tramite appositi protocolli (HTTP, HTTPS, SPDY, Websocket, ...), ed interpretare le risposte attraverso un normale web browser. Il compito del server, invece, è quello di elaborare le richieste ottenute dal client tramite un web-server, elaborando le informazioni ottenute dagli strati applicativi che incapsulano la logica applicativa e i dati, ed inviando al client pagine web opportunamente costruite (pagine web dinamiche). Tali pagine web possono essere corredate di script, i quali vengono eseguiti dal browser del client, allo scopo di renderle interattive, fornendo all'utente un'esperienza assimilabile a quelle fornite da applicativi desktop.

L'insieme di tecnologie disponibili, per l'implementazione e l'utilizzo di applicativi web, è molto vasto, così come sono numerosi i criteri di scelta secondo i quali esse vengono impiegate. Gli aspetti più importanti da considerare durante questa fase comprendono:

- **Sicurezza**, sia lato server che lato client, in quanto ogni applicativo esposto alla rete è potenzialmente un punto d'attacco per malintenzionati. Problematiche relative alla sicurezza possono comprendere:
  - criptatura dei dati sensibili e firma dei contenuti contro *phishing* (falsificazione) e *man in the middle* attacks;

Figura 5.3: Riepilogo della struttura, e delle tecnologie, di una UI basata sul web.



  - politiche di riconoscimento degli utenti (autenticazione) e gestione delle sessioni;
  - gestione dei permessi (autorizzazione) e contromisure per *direct data access and theft* ed *information leakage* (furto d'informazioni);
  - procedure di validazione dei dati e controlli contro *data poisoning*, cioè manipolazione e danneggiamento dei dati;
  - controllo sui files per evitare *malicious file execution*, cioè l'esecuzione di codice malevolo introdotto nel sistema;
  - denial of service, o *DoS*, cioè la negazione del servizio.

- **Affidabilità** della rete, la quale può impedire agli utenti l'utilizzo dell'applicativo; malfunzionamenti della rete possono essere causati da apparati di rete o attacchi *DoS*.

- **Performance**, inteso come tempo di risposta alle richieste degli utenti; problemi di performance possono essere causati dall'errata gestione di grossi insiemi di dati, o codice poco performante.

- **Incompatibilità** tra i devices, o i browsers, e l'applicativo, la quale può essere causata dall'utilizzo di software non standard compliant, o da mancata attenzione agli standards nella programmazione lato server.

[FS04]

Nei prossimi paragrafi verranno analizzate le strutture, e le tecnologie, impiegate per i vari componenti dello strato di interfaccia di *NST*.

5.4 Server side UI, ASP.NET MVC

ASP.NET MVC è un framework per lo sviluppo di applicativi web basato sul pattern *MVC*4.2.1 e sulle tecnologie *ASP.NET* di Microsoft. Per l'implementazione del server side dell'interfaccia di *NST*, è stata impiegata

la versione 4 di *ASP.NET MVC*, detta appunto *MVC4*, codificata tramite *C#*.

ASP.NET MVC è caratterizzato da uno sviluppo fortemente legato al pattern *MVC*, il quale coinvolge principalmente tre ruoli: i models, le view e i controllers. In *ASP.NET MVC (C#)*[2] i models ed i controllers prendono la forma di classi opportunamente annotate, mentre le views sono particolari documenti di markup, caratterizzati da specifiche sintassi basate su HTML, i quali vengono elaborati da un view engine, ed inviati come risposta al client.

MVC4 comprende una serie di componenti per la gestione di problematiche quali: autenticazione e autorizzazione degli utenti che effetuano richieste all'applicativo (Membership management), mappatura degli URLs e instradamento delle richieste verso controllers (url mapping e request routing), e gestione delle sessioni. Questi componenti possono essere rimpiazzati da versioni personalizzate tramite l'utilizzo di Dependency Injection e Inversion of Control. La versione 4 di *ASP.NET MVC* introduce inoltre le *ASP.NET Web API*, cioè un framework per la produzione di servizi RESTful[3].

Il flusso dati di una richiesta, in ambiente ASP.NET MVC, può essere riassunto come segue:

1. IIS riceve la richiesta dal client e la inoltra ad ASP.NET;

2. il sistema di instradamento delle richieste di ASP.NET, esegue un match tra la richiesta e le route disponibili, individuando così, la route corretta;

3. la richiesta raggiunge così MVC il quale, tramite una controller factory, istanzia un oggetto di tipo Controller o ApiController, adatto alla richiesta da gestire;

4. a questo punto MVC individua il metodo specifico del controller da invocare;

[2]progetti ASP.NET MVC codificati con C#

[3]REpresentational State Transfer (REST) è un'architettura software per il world wide web centrata sulla descrizione di risorse, sulla loro localizzazione e sul loro trasferimento

5. prima dell'invocazione del metodo del controller, viene verificato che l'utente sia autorizzato all'esecuzione del metodo stesso;

6. se l'utente è autorizzato il metodo viene eseguito e viene prodotto un oggetto in risposta, eventualmente elaborando un modello e la relativa vista;

7. se non ci sono state eccezioni la risposta viene inviata al client;

Questo flusso rende centrale il ruolo dei controller, i quali hanno il ruolo di coordinatori, interfacciando viste e modelli con il backend applicativo.

5.4.1 Models

I models sono oggetti che modellano le entità del dominio. In *ASP.NET MVC* i models sono implementati tramite semplici classi, corredati all'occorrenza da annotazioni (*Data Annotations*) per specificare regole di validazione, direttive per la visualizzazione e relazioni con altri models. I models vengono utilizzati come base dati delle viste e come input e output per i controllers.

```
public class AziendaModel
{
  [Display(Name = "Id")]
  public int Id { get; set; }

  [Range(0, UInt32.MaxValue,
    ErrorMessageResourceName = "ErrCodiceFiliale",
    ErrorMessageResourceType = typeof(App_GlobalResources.Validation))]
  [Display(Name = "IdFiliale")]
  public int IdFiliale { get; set; }

  public int? IdImmagine { get; set; }

  [Required(AllowEmptyStrings = false,
    ErrorMessageResourceName = "ErrDescrizioneNonValida",
    ErrorMessageResourceType = typeof(App_GlobalResources.Validation))]
  [Display(Name = "Descrizione", ResourceType = typeof(App_GlobalResources↩
      .Archives))]
  public string Descrizione { get; set; }
```

```
    [Display(Name = "Indirizzo", ResourceType = typeof(App_GlobalResources.↩
        Archives))]
    public string Indirizzo { get; set; }

    /*
      ...
     */

    [Required]
    [Display(Name = "CodiceFiscale", ResourceType = typeof(↩
        App_GlobalResources.Archives))]
    public string CodiceFiscale { get; set; }

    [Required]
    [Display(Name = "PartitaIva", ResourceType = typeof(App_GlobalResources.↩
        Archives))]
    public string PartitaIva { get; set; }
}
```

Il codice mostrato rappresenta il model di un'azienda, e mostra l'utilizzo di alcune *data annotations* utilizzate da *ASP.NET MVC* nelle operazioni di validazione, visualizzazione e serializzazione dell'oggetto. Le annotazioni *Range* e *Required* sono relative alla validazione dei dati; *Required*, ad esempio, permette di marcare una proprietà come richiesta, cioè non opzionale, e permette inoltre di specificare eventuali messaggi d'errore da mostrare nel caso la proprietà non venga impostata correttamente. L'annotazione *Display* invece fornisce alcune informazioni di base per la visualizzazione della proprietà in fase di rendering delle viste. *MVC4* prevede altre annotazioni, ad esempio *RegularExpressionAttribute*, per la validazione tramite espressioni regolari, *EnumDataTypeAttribute* per la validazione di tipi enum, *DisplayFormatAttribute* per la formattazione dei dati in fase di rendering, etc.

L'utilizzo delle annotazioni permette di mantenere un approccio dichiarativo durante la codifica dei modelli, ed evita al programmatore di dover gestire compiti ripetitivi di validazione dell'input, ed alla personalizzazione nella fase di rendering dell'output.

5.4.2 Views

Le viste sono rappresentate da templates in formato misto HTML/C#, cioè un documento HTML con inserti codificati tramite C#, i quali vengono elaborati in fase di rendering. Il componente che determina la sintassi di questi templates, e ne effettua il rendering, è detto *view engine*. *ASP.NET MVC* si può interfacciare con diversi tipi di *view engine*, anche personalizzati. Dalla release di *MVC3*, è stato introdotto il view engine *Razor*, impiegato per lo sviluppo delle viste di *NST*.

La sintassi *Razor*, rispetto alle sintassi precedentemente proposte da *Microsoft*, introduce alcuni vantaggi chiave:

- sintassi basata su linguaggi noti, senza particolari aggiunte;
- riduzione del numero di caratteri che compongono una vista;
- supporto di viste di layout, dette layouts;
- supporto ad *IntelliSense* (feature di autocompletamento del codice);
- supporto per Unit testing;

Il supporto ai layouts, e la possibilità di creare viste parziali, permette allo sviluppatore di costituire una gerarchia di viste con caratteristiche estetiche comuni senza ripetere codice, e di creare una serie di componenti grafici ampiamente riutilizzabili.

```
@model Gesp.NST.Models.AziendaModel
<div class="Brief Brief-Azienda unselectable"
   data-bind="click: $parent.SelectItem">
    <div class="Identificazione">
        <span class="Descrizione Title"
      data-bind="text: Descrizione, click: $root.EditItemForm">
      @Model.Descrizione</span>
        <span class="Indirizzo"
      data-bind="trimText: IndirizzoCompleto, trimTextLength: 30">
      @Model.Indirizzo</span>
    </div>
    <div class="Stato">
```

```
        <span data-bind="Stato: Stato">@Model.Stato</span>
        @Html.LabelFor(m => m.Stato)
    </div>
    <div class="Telefono">
        <span data-bind="text: Telefono">@Model.Telefono</span>
        @Html.LabelFor(m => m.Telefono)
    </div>
    <div class="Email">
        <span title="@Model.Email"
            data-bind="trimText: Email, trimTextLength: 16">
      @Model.Email</span>
        @Html.LabelFor(m => m.Email)
    </div>
</div>
```

La vista d'esempio, basata sul model illustrato 5.4.1, mostra l'utilizzo sintassi *Razor*. Il carattere @indica al view engine che quanto segue è da interpretare come codice, e va dunque elaborato prima del rendering. La prima riga del codice, dichiara il modello sul quale è basata la vista; in seguito nel codice ci si riferisce ai dati del modello tramite la sintassi @Model. La vista mostrata è parziale, cioè rappresenta solamente un frammento di pagina HTML, in questo modo è possibile riutilizzare questo codice come componente grafico di viste più ampie, ad esempio elenchi.

5.4.3 Controllers ed API controllers

I controllers hanno il compito di elaborare le richieste utente e coordinare modelli e viste allo scopo di produrre l'output desiderato. Data la struttura stratificata di *NST*, ricade sul controller anche la responsabilità di interfacciarsi con lo strato della logica applicativa. Un particolare tipo di controller è rappresentato dall'*APIController*, il quale, anziché renderizzare viste, si limita a serializzare i dati in un formato di interscambio, come Json o XML, ed inviarli in risposta al client. L'operazione di serializzazione avviene in modo automatico, attraverso il framework *MVC4*, in base alle direttive dichiarate tramite annotazioni e al tipo di formato richiesto dal client.

```
[Authorize(Roles = "Amministratore")]
```

```
public class AmministrazioneController : Controller
{
  /// <summary>
  /// Richiesta vista parziale per azienda.
  /// </summary>
  /// <param name="id">Identificativo numerico univico azienda</param>
  /// <returns>Vista brief azienda</returns>
  public ActionResult BriefAzienda(int idAzienda)
  {
    //istanzio modello della vista
    AziendaModel model;

    /*
      inoltro richiesta verso Business Layer per ottenere
      azienda con id "idAzienda" e compilazione del modello vista
     */

     //rendering del model attraverso vista parziale
    return PartialView("~/Views/Amministrazione/Aziende/_Brief.cshtml", ↩
        model);
  }
}
```

Il metodo *BriefAzienda*, del controller mostrato, si interfaccia con lo strato di business e richiede i dati dell'azienda identificata da un certo intero (*idAzienda*), per poi compilare il modello dati ed inviarlo alla vista *_Brief.cshtml* per il rendering.

Il seguente controller di tipo *ApiController* è analogo a quello mostrato in precedenza ma, anziché renderizzare l'azienda in una vista, ritorna il model a *MVC4* per la serializzazione.

```
[Authorize(Roles = "Amministratore")]
public class AmministrazioneApiController : ApiController
{
  public Azienda GetAzienda(int idAzienda)
  {
    //istanzio modello della vista
    AziendaModel model;

    /*
      inoltro richiesta verso Business Layer per ottenere
      azienda con id "idAzienda" e compilazione del modello vista
     */
```

```
    //ritorno il modello dati
    return model;
  }
}
```

Entrambi i controller mostrati sono annotati con *AuthorizeAnnotation*, annotazione che permette ad *MVC4* di identificare gli utenti che possono accedere ai metodi del controller. In questo caso la proprietà *Roles* dell'annotazione *Authorize* permette l'accesso ai controller solamente agli utenti del gruppo Amministratore.

5.5 Client side UI

Il client side di un'interfaccia web è composto da pagine web interattive. Queste pagine vengono elaborate dal client tramite un web browser, e costituiscono il punto d'accesso vero e proprio per il sistema. I sorgenti dei quali necessita il browser, per realizzare l'interfaccia, sono costituiti da:

- documenti di markup, codificati in HTML, forniscono la struttura delle pagine, e dunque dell'interfaccia, e contengono riferimenti agli altri elementi necessari all'interfaccia;
- fogli di stile, cioè documenti codificati in CSS (Cascading Style Sheets), contenenti direttive di layout che governano l'aspetto grafico all'interfaccia;
- script, codificati in linguaggio Javascript, interpretati ed eseguiti dal browser per aggiungere aspetti di interattività e dinamicità all'interfaccia.

Questi componenti devono rispettare il più possibile gli standard imposti dal consorzio *W3C*, affinché l'interfaccia sia eseguibile sul maggior numero di browser possibile, e sia quindi indipendente dall'architettura hardware del device che accede all'applicativo.

Nei prossimi paragrafi verranno analizzate alcune delle tecnologie impiegate per lo sviluppo dell'interfaccia lato client di *NST*.

5.5.1 HTML5, CSS3 e Javascript

L'interfaccia utente di *NST* è basata sugli standard *HTML5*, *CSS3* e *Javascript*.

Il termine *HTML5* è spesso usato in riferimento ad un insieme di tecnologie del web moderno, molte delle quali sono sviluppate dal *WHATWG*, cioè *Web Hypertext Application Technology Working Group*, talvolta in collaborazione con il consorzio *W3C* e l'*IETF* (Internet Engineering Task Force)[Gro14]. Lo sviluppo di *HTML5*, è iniziato nel 2004 ad opera del *WHATWG*, allo scopo di progettare specifiche per lo sviluppo di applicazioni web, ed ha visto l'intervento del *W3C* solo nel 2007 [Wik14a]. Nel 2012 però, il *W3C*, ha deciso nuovamente di serpararsi dal processo di standardizzazione, perseguendo una propria versione di *HTML5*, che al momento è presentato in due versioni: la versione *HTML living standard*, supportata dal *WHATWG*, e la versione del *W3C*, che corrisponderà ad uno snapshot della versione del *WHATWG*.

HTML5 introduce alcune importanti novità rispetto alla versione 4:

- maggiore disaccoppiamento tra la struttura del documento, gli stili grafici (fonts, colori, bordi, etc.) e il contenuto della pagina;
- supporto di contesti grafici, rappresentati da elementi di tipo *canvas*, che permettono di utilizzare Javascript per creare grafica raster;
- supporto alla geolocalizzazione per i device che lo consentono;
- introduzione del *WebStorage*, in sostituzione ai cookies, per la memorizzazione di dati nel browser senza spreco di banda;
- introduzione dei *WebWorkers*, cioè script in formato Javascript, eseguiti in background, indipendentemente dagli script sui quali è eseguita la pagina.

La tecnologia di supporto ad *HTML5* scelta per *NST*, per quanto riguarda le direttive di stile, è la terza versione di CSS, cioè *CSS3*. *CSS* è un linguaggio per la formattazione di documenti HTML, XHTML e XML, introdotto a partire dal 1996 dal W3C, per migliorare la manutenibilità e la riusabilità del codice delle pagine web. Il codice *CSS* è strutturato in regole, a loro volta suddivise in selettori, proprità e valori. Il selettore permette di indicare al browser quali elementi sono soggetti ad una certa regola; proprietà e valori sono raggruppate per selettori, e indicano le proprietà da assegnare agli elementi riferiti dai selettori che le comprendono.

```
/* Questo e' un commento. */

/* Il seguente selettore applica le regole a tutti gli
   elementi di tipo "a" */
a {
  /* foreground dell'elemento colore 00 00 00 (nero)*/
  color: #000000;
}

/* Il seguente selettore applica le regole a tutti gli elementi
   il cui attributo "class" contiene la stringa "blue-fg" */
.blue-fg {
  color: #0000FF;
}

/* Il seguente selettore applica le regole all'elemento
   identificato dall'attributo "id" con valore "TitoloGrande" */
#TitoloGrande {
  /* colore bianco del foreground */
  color: #FFFFFF;
  /* colore nero per background */
  background-color: #000000;
  /* padding (scostamento) di 5px su tutti i lati*/
  padding: 5px;
  /* Imposta una dimensione del testo al 200% del normale */
  font-size: 200%;
}
```

CSS3 è al momento lo standard più recente per i fogli di stile e, pur mantenendo completa retrocompatibilità, introduce numerose features quali: selettori più evoluti, box models più elastici, alterazione del contenuto tramite

direttive, effetti al testo, trasformazioni 2D e 3D, layout multi-colonna, etc.

L'interattività delle pagine, come di consueto nel web, è implementata tramite l'utilizzo di *Javascript*. *Javascript* è un linguaggio interpretato, orientato agli oggetti, e caratterizzato da una sintassi simile al Java (dal quale prende il nome). A dispetto del nome però, le differenze con Java sono molteplici: mentre Java è fortemente tipizzato, e fortemente orientato agli oggetti, *Javascript* è debolmente tipizzato, e gli oggetti sono strutturati più come array associativi, che come oggetti Java o C++. Sebbene venga utilizzato anche lato server (ad esempio da piattaforme come *NodeJS*), JavaScript è nato come linguaggio lato client per il web, cioè per essere eseguito direttamente sul browser del client, pertanto per lo scopo di questa trattazione, ci riferiremo a Javascript in riferimento al codice lato client.

```
/**
 * Semplice classe JavaScript che implementa le funzionalità
 * di un cronometro.
 */
function Cronometro()
{
  //riferimento privato all'oggetto
  var self = this;
  //attributo privato
  var interval = null;
  //attributo pubblico
  self.ticks = 0;

  //metodo pubblico
  self.reset = function () {
    self.ticks = 0;
  };
  //avvio
  self.avvia = function () {
    //impostazione di chiamata temporizzata
    interval = setInterval(
      function () { self.ticks++; }, //funzione richiamata
      Cronometro.SECOND_IN_MILLISECONDS //intervallo
    );
  };
  //stop
  self.ferma = function () {
```

```
    //blocca l'esecuzione temporizzata
    if(interval)
      clearInterval(interval);
  };
  //mostra all'utente quanti ticks sono passati
  self.alarm = function () {
    alert("Tempo passato: "+self.ticks+" secondi");
  };
}

/**
 * Variabile statica (usata come costante) della classe Cronometro
 */
Cronometro.SECOND_IN_MILLISECONDS = 1000;

//istanziazione di un oggetto
var stopWatch = new Cronometro();
//invocazione di un metodo pubblico
stopWatch.avvia();
//temporizzazione di chiamata a funzione dopo circa 3 secondi
setTimeout( function(){
  stopWatch.ferma();
  stopWatch.alarm();
  }, 3001 );
```

5.5.2 Il DOM, Document Object Model

Il *Document Object Model*, o *DOM*, è un'interfaccia per la manipolazione dinamica della struttura, dei contenuti e degli stili di un documento[W3C09], i cui standard sono definiti dal *W3C*.

In sostanza il *Document Object Model*, tradotto in *modello ad oggetti del documento*, è la rappresentazione di un documento secondo il modello ad oggetti, che permette ai programmi di gestirne gli elementi in tempo reale. Tale rappresentazione è indipendente da piattaforme o linguaggi, ed è impiegata da programmi lato server, quanto da script lato client. In particolare, questa rappresentazione ad oggetti, è supportata nativamente dai browser web, che la impiegano in fase di interpretazione dei documenti HTML, costruendo una struttura ad albero manipolabile, e stilizzabile, tramite JavaScript e CSS.

Il *DOM*, nei browser, prende tipicamente la forma di una struttura dati ad albero, i cui nodi sono rappresentati da elementi relativi a tags specifici, ad esempio tabelle, forms, collegamenti esterni, immagini, oppure ad attributi e nodi testuali. Il codice 5.5.2 rappresenta una semplice pagina web in HTML, mentre la figura 5.5.2 ne riassume l'interpretazione, e la relativa struttura del *DOM*.

Listing 5.1: Un semplice documento HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>titolo documento</title>
    <script type="text/javascript">
      /* ... */
    </script>
  </head>
  <body>
    <h1>Intestazione 1</h1>
    <p>testo del paragrafo...</p>
    <a href="http://nst.gesp.it/">testo del link</a>
  </body>
</html>
```

Una volta che il browser ha intepretato il documento HTML, il modello risultante sarà simile a quello rappresentato in figura 5.5.2. La manipolazione del modello costruito può avvenire tramite script, i quali possono essere inclusi nel documento stesso sotto forma di codice JavaScript, oppure possono essere referenziati come risorse esterne. Il codice 5.5.2 mostra un esempio di manipolazione del *DOM*: inserito nel tag script dell'esempio 5.5.2, il codice proposto permette di eliminare dal documento tutti i tag di tipo *p*.

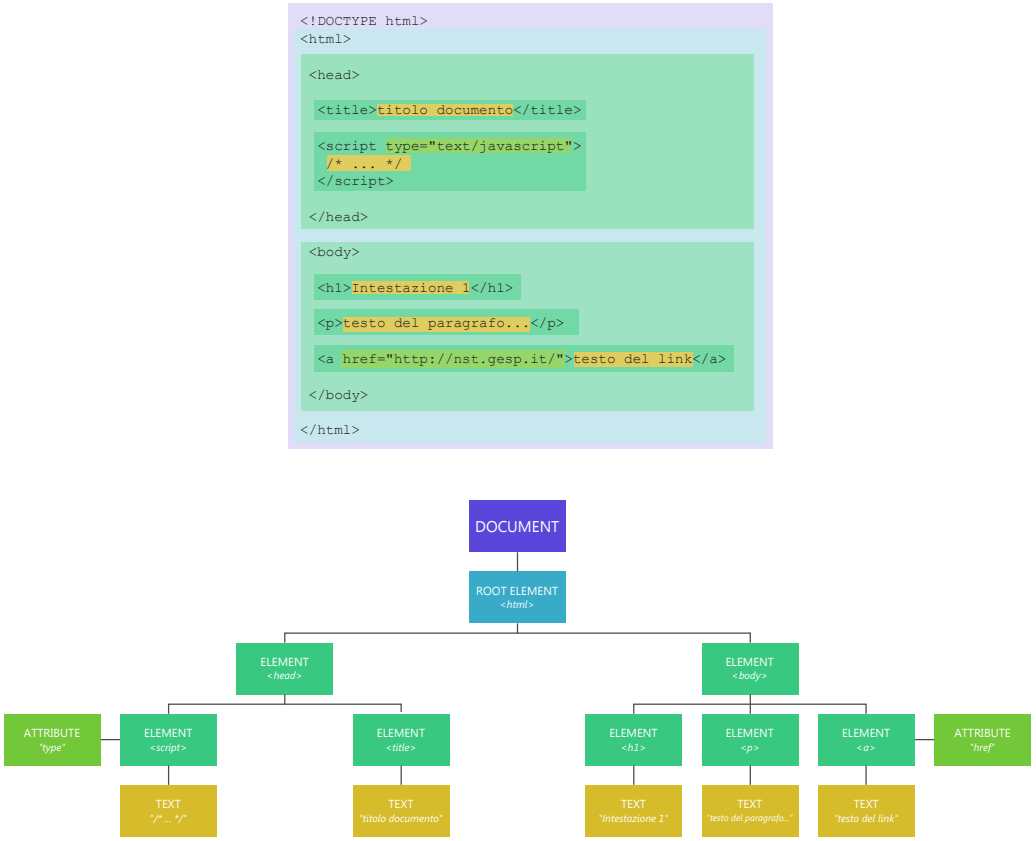
5.5.3 AJAX, accesso asincrono ai dati

Il funzionamento delle prime web applications, legate al funzionamento dei primi web browser, prevedeva che ogni interazione tra utente ed applicazione (quale l'invio di dati, la richiesta di risorse o una semplice operazione di

Listing 5.2: Esempio di manipolazione del DOM tramite JavaScript

```
//recupera dal DOM (oggetto document) tutti i tags di tipo "p"
var paragrafi = document.getElementsByTagName("p");
//finché ci sono paragrafi da eliminare...
while(paragrafi.length > 0) {
  //prendi il primo paragrafo dall'array di paragrafi
  var paragrafo = paragrafi[0];
  //ottieni l'elemento padre
  var parent = paragrafo.parentNode;
  //rimuovi il paragrafo dal dom attraverso il parent
  parent.removeChild(paragrafo);
}
```

Figura 5.4: Il DOM, Document Object Model, di una semplice pagina HTML



login), fosse seguito da un nuovo ciclo di richiesta, risposta e rendering della pagina, durante il quale l'interazione dell'utente con l'interfaccia web è bloccata. Questo comportamento viene comunemente detto *refresh* ed è derivato dalla struttura dell'ipertesto, la quale prevede semplici documenti referenziati tramite collegamenti, sul quale si è basato il web fin dal principio. Col tempo, e con l'evoluzione delle web applications, questo comportamento è diventato particolarmente controproducente: l'utente è infatti costantemente interrotto durante la sua interazione, dovendo attendere che il browser proceda ad effettuare le singole richieste sui dati, rompendone la concentrazione ed eliminando di fatto il parallelismo intrinseco in certe operazioni di manipolazione dei dati.

Supponiamo ad esempio di voler eliminare da un elenco di elementi, una serie di elementi selezionati secondo criteri arbitrari. Secondo il suddetto modello di funzionamento le procedure coinvolte nell'eliminazione di un singolo elemento comprendono:

1. l'utente richiede, attraverso il browser, l'elenco degli elementi e attende una risposta;
2. il webserver risponde inviando un documento contenente l'elenco dati;
3. il browser dell'utente riceve il documento e ne effettua il rendering, presentando i dati all'utente;
4. l'utente individua uno o più elementi, da eliminare, ed invia la richiesta di eliminazione tramite il browser;
5. il browser blocca la pagina, bloccando di fatto l'interazione dell'utente, per inviare la richiesta al server;
6. il server riceve la richiesta, effettua l'eliminazione e invia nuovamente tutto l'elenco di elementi al client;
7. il browser ripete il rendering della pagina, sui dati ottenuti dalla risposta, e l'utente può tornare a consultare e manipolare i dati.

Questa procedura, oltre agli svantaggi già citati, comporta uno spreco di banda, in quanto ad ogni richiesta il server è costretto a re-inviare l'intero elenco di oggetti al client.

Per ovviare a questi problemi vengono impiegate tecniche **AJAX**, cioè JavaScript asincrono e XML (*Asynchronous JavaScript and XML*). Il termine asincrono si riferisce al fatto per cui le richieste avvengono *in background*, cioè senza interrompere l'interazione utente, tramite alcuni script javascript, e possono essere effettuate anche in parallelo. L'utilizzo di XML, invece, è opzionale. Le tecniche AJAX prevedono diverse fasi, cioè: la richiesta asincrona, la deserializzazione dei dati ottenuti in risposta, la manipolazione della struttura del documento. A queste fasi si può aggiungere la fase di serializzazione, nel caso in cui rappresentazioni di oggetti complesse vengano inviate tramite richieste *HTTP POST*, secondo un certo formato di interscambio.

La procedura illustrata per la manipolazione dei dati (5.5.3), viene percioò sostanzialmente modificata tramite l'impiego di tecniche ajax:

1. l'utente richiede, attraverso il browser, l'elenco degli elementi e attende una risposta;

2. il webserver risponde inviando un documento contenente l'elenco dati, il quale è corredato di opportuni script che implementano tecniche AJAX;

3. il browser dell'utente riceve il documento, ne effettua il rendering, ed interpreta gli script allegati;

4. l'utente individua uno o più elementi, da eliminare, e seleziona il comando di eliminazione dall'interfaccia;

5. gli script AJAX vengono eseguiti, tramite il browser, e la richiesta viene inviata in background, senza interrompere l'utente, il quale può continuare a selezionare altri elementi da eliminare oppure consultare diversamente i dati;

6. il server riceve la richiesta, effettua l'eliminazione e invia una semplice conferma dell'eliminazione;

7. il browser riceve la conferma dell'eliminazione e può alterare la pagina già renderizzata, confermando eventualmente all'utente l'esito affermativo delle azioni compiute.

Grazie a queste tecniche viene reintrodotto il parallelismo nelle operazioni, in quanto l'utente può continuare ad utilizzare l'applicativo, come se si trattasse di una normale interfaccia desktop, oppure effettuare più operazioni sui dati contemporaneamente.

I formati di scambio più utilizzati sono *XML*, *HTML* e *JSON*. Per *NST* è stato scelto *JSON*, principalmente per le sue doti di leggerezza e semplicità. L'acronimo **JSON** significa *JavaScript Object Notation*, in quanto tale formato è basato sullo standard *ECMA* per Javascript. *JSON*, a differenza di XML, non è un linguaggio di marcatura e la sua sintassi è interpretabile rapidamente tramite javascript, permettendo una rapida deserializzazione in oggetti, tramite l'operatore *eval()*. *JSON* è inoltre supportato dal framework *MVC4* per quanto riguarda l'interfaccia lato server. I tipi di dato, supportati dal formato *JSON* sono booleani, numeri interi e a virgola mobile, stringhe, array e array associativi, ed il tipo di dato nullo.

```
{
  "status":200,
  "reason":"Ok",
  "result":[
      {
        "Id":1,
        "Descrizione":"GESP srl",
        "Indirizzo":"Viale Scarampo, 47"
        "Cap":"20418",
        "Localita":"Milano",
        "Provincia":"MI",
        "Nazione":"IT",
        "Stato":"Attivo"
      }
    ]
  "headers":null
}
```

Il frammento di codice proposto rappresenta una tipica risposta in formato *JSON*, e ne mostra le doti di leggibilità. I campi degli oggetti sono facilmente identificabili dal proprio nome, e sono seguiti dal loro valore, sempre espresso in formato leggibile, secondo la formattazione richiesta dal tipo di dato rappresentato. L'oggetto rappresentato è complesso, ed è composto dai dati di un'azienda, incapsulati in un oggetto che trasporta informazioni di stato riguardanti l'esito della richiesta. Si può notare l'utilizzo di alcuni dei tipi di dati supportati da JSON: i campi *status* e il campo *Id* dell'azienda, assumono valori interi; il loro valore è rappresentato da cifre espresse in caratteri, senza conversioni in binario o cambiamenti di base. Campi quali *Descrizione*, *indirizzo*, o il campo *reason*, sono campi di tipo stringa, il cui valore è delimitato da doppi apici. Il campo *result* è un array, i cui elementi sono racchiusi tra parentesi quadre e delimiatati da virgole. Infine l'oggetto azienda, compreso nell'array *result*, è espresso come un array associativo, e il campo *headers* presenta un valore nullo.

5.5.4 MVVM con KnockoutJS

I dati della UI, scambiati col server tramite richieste AJAX, devono poi essere elaborati, strutturati e mostrati all'utente, sotto forma di elementi del documento HTML. Questa operazione di collegamento, tra i dati e la loro rappresentazione, è ottenuta nei client di *NST*, tramite l'utilizzo della libreria *KnockoutJS*, la quale opera secondo i principi del pattern *MVVM*. *KnockoutJS* fornisce una serie di utilities, e classi, che possono essere utilizzate per effettuare il binding tra i model e le viste, senza doverlo operare manualmente, bensì utilizzando una tecnica di *binding dichiarativo*, che sfrutta informazioni incluse nelle viste per associarvi il view model. L'esempio 5.5.4 mostra un viewmodel associato ad un'azienda.

```
/**
 * ViewModel per Azienda.
 */
function AziendaViewModel() {
```

```
  //riferimento intero oggetto
  var self = this;
  //id intero univoco
  self.Id = ko.observable(0);
  //altri dati
  self.Descrizione = ko.observable("");
  self.Indirizzo = ko.observable("");
  self.Cap = ko.observable("");
  self.Localita = ko.observable("");
  self.Provincia = ko.observable("");
  self.Nazione = ko.observable("");
  self.Stato = ko.observable("");

  //campo calcolato,
  self.IndirizzoCompleto = ko.computed(function () {
    return self.Indirizzo() + ", " + self.Cap()
      + " " + self.Localita() + " - " + self.Provincia()
      + " - " + self.Nazione();
  }, self);

  //popola il ViewModel in base ai dati azienda
  //ottenuti dal server.
  self.fromJSon = function (aziendaJson) {
    self.Id(aziendaJson.Id);
    self.Descrizione(aziendaJson.Descrizione);
    self.Indirizzo(aziendaJson.Indirizzo);
    self.Cap(aziendaJson.Cap);
    self.Localita(aziendaJson.Localita);
    self.Provincia(aziendaJson.Provincia);
    self.Nazione(aziendaJson.Nazione);
    self.Stato(aziendaJson.Stato);
  };

}
```

Il ViewModel proposto utilizza le classi di *KnockoutJS* per gestire variazioni sui dati del modello, ed eventualmente, modificare la vista di conseguenza. I campi di tipo *ko.computed* hanno valori calcolati, e possono essere derivati da altri campi dell'oggetto, come nell'esempio relativo all'indirizzo dell'azienda. Il metodo *fromJson* viene richiamato dal client quando ottiene un nuovo modello dati dal server, allo scopo di popolare i dati del ViewModel con i dati ottenuti, per poi legarli alla vista. Un esempio di vista, legata al ViewModel proposto, è rappresentato dal frammento di codice 5.4.2. Il bin-

ding dichiarativo si effettua tramite l'utilizzo degli attributi *data-bind*, che vengono letti ed interpretati da *KnockoutJS* in fase di rendering.

La libreria *KnockoutJS* fornisce anche supporto per templating, tramite particolari direttive di binding di cui si fornisce un esempio con il frammento di codice 5.5.4.

```
<!-- Elemento che contiene una lista di aziende
   renderizzate secondo il template TmplAzienda -->
<ul id="ListaAziende"
   data-bind="foreach: ListaAziende">
  <li data-bind="template: { name: TmplAzienda }"></li>
</ul>

<!-- Template per la visualizzazione in lista di
   aziende; comprende descrizione e indirizzo -->
<script type="text/html" id="TmplAzienda">
  <li class="WrapperAzienda" data-bind="css: Stato()">
    <input type="hidden" data-bind="value: Id" value="" />
    <h3 data-bind="text: Descrizione"></h3>
    <p class="Indirizzo" data-bind="text: IndirizzoCompleto">
    </p>
  </li>
</script>
```

L'esempio 5.5.4 mostra diverse direttive di binding:

- *foreach*, indica che il rendering va effettuato per ogni elemento della collection, in questo caso l'array *ListaAziende*;
- *template*, indica che il rendering degli elementi va effettuato utilizzando un template, il quale è contenuto in un tag *script*, identificato dal nome TmplAzienda;
- *text* è probabilmente la direttiva più comune, in quanto permette di impostare il valore di un tag html con il valore del campo collegato;
- *value*, associa il valore di un campo all'attributo value; molto usata in template relativi a moduli di inserimento (*form*);

- *css*, permette di aggiungere un certo stile, in questo caso associato allo stato dell'azienda.

Conclusioni

Lo sviluppo di *NST* è tuttora in corso, ma è possibile affermare che questo processo ha finora avuto esito positivo, secondo l'impiego delle metodologie, e delle tecnologie, esposte in questa tesi, seppur con qualche eccezione: Scrum è stato applicato, ed ha dato degli ottimi benefici nella selezione delle attività secondo priorità e importanza funzionale; tuttavia le demo a volte non sono state efficaci, e non è stato utilizzato il controllo della velocità previsto dal metodo. TDD e CI sono stati improntati ma non usati fino in fondo; in alcuni casi le funzionalità hanno anticipato i test, e le build automatiche non sono state effettuate.

Il team ha seguito le linee guida impostate ad inizio progetto (separazione in layer, linee guida di scrittura codice e progettazione db, progettazione di GUI tramite Mockups), e di questo ne ha beneficiato la qualità finale del software.

L'applicativo è al momento interfacciato con un ottimizzatore, dotato di algoritmi all'avanguardia, capace di risolvere in poco tempo, complessi problemi di pianificazione logistica. Durante la fase *pre-alfa*, *NST* è stato interfacciato con *ERP* aziendali e piattaforme di monitoraggio mezzi, e sarà presto avviata la fase di *beta* per un modulo di monitoraggio della temperatura dei vani di carico, di una flotta di mezzi atti al trasporto di merci di consumo per la *GDO*. L'applicativo può già essere utilizzato tramite interfaccia web mediante diversi tipi di dispositivi, a partire dai personal computer fino ai cellulari di ultima generazione.

Con questo studio sono stati esposti quei temi, selezionati come i più

importanti, necessari per introdurre il lettore al vasto problema della progettazione e sviluppo di software per il supporto alla logistica, illustrando il percorso sostenuto e i mezzi con i quali è stato affrontato.

Bibliografia

[ABB+95] P. Augerat, J.M. Belenguer, E. Benavent, A. Corberán, D. Naddef, and G. Rinaldi. Computational results with a branch and cut code for the capacitated vehicle routing problem. Technical Report 1 RR949-M, ARTEMIS-IMAG, Grenoble France, 1995.

[AGL+04] R. K. Ahuja, J. Goodstein, J. Liu, A. Mukherjee, and J. B. Orlin. A neighborhood search algorithm for the combined through and fleet assignment model with time windows. *Networks*, 44(2):160–171, 2004.

[Aug95] P. Augerat. *Approche polyédrale du problème de tournées de véhicules*. PhD thesis, Institut National Polytechnique de Grenoble, 1995.

[BBM06] R. Baldacci, L. D. Bodin, and A. Mingozzi. The multiple disposal facilities and multiple inventory locations rollon-rolloff vehicle routing problem. *Computers & Operations Research*, 33:2667–2702, 2006.

[BBMR10] R. Baldacci, E. Bartolini, A. Mingozzi, and R. Roberti. An exact solution framework for a broad class of vehicle routing problems. *Computational Management Science*, pages –, 2010.

[BC12] M. Blocho and Z.J. Czech. A parallel memetic algorithm for the vehicle routing problem with time windows. *Parallel Computing*, 2012.

[BCM08] R. Baldacci, N. Christofides, and A. Mingozzi. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*, 115(2):351–385, 2008.

[BDG04] O. Braysy, W. Dullaert, and M. Gendreau. Evolutionary algorithms for the vehicle routing problem with time windows. *Journal of Heuristics*, 10(6):587–611, 2004.

[BG05a] O. Braysy and M. Gendreau. Vehicle routing problem with time windows, part I: route construction and local search algorithms. *Transportation Science*, 39(1):104–118, 2005.

[BG05b] O. Braysy and M. Gendreau. Vehicle routing problem with time windows, part II: metaheuristics. *Transportation Science*, 39(1):119–139, 2005.

[BH04] R. Bent and P. Van Hentenryck. A two-stage hybrid local search for the vehicle routing problem with time windows. *Transportation Science*, 38(4):515–530, 2004.

[BHM04] R. Baldacci, E. Hadjiconstantinou, and A. Mingozzi. An exact algorithm for the capacitated vehicle routing problem based on a two-commodity network flow formulation. *Operations Research*, 52(5):723–738, 2004.

[BKM07] N. Boland, B. Kallehauge, and O. B. G. Madsen. Path inequalities for the vehicle routing problem with time windows. *Networks*, 49(4):273–293, 2007.

[BKY02] J. F. Bard, G. Kontoravdis, and G. Yu. A branch-and-cut procedure for the vehicle routing problem with time windows. *Transportation Science*, 36(2):250–269, 2002.

[BL07] C. Barnhart and G. Laporte. *Handbook in OR & MS, Vol. 14*. Elsevier B.V., 2007.

[BMR08] M.A. Boschetti, A. Mingozzi, and S. Ricciardelli. A dual ascent procedure for the set partitioning problem. *Discrete Optimization*, 5(4):735–747, 2008.

[BMR11] R. Baldacci, A. Mingozzi, and R. Roberti. New route relaxation and pricing strategies for the vehicle routing problem. *Operations Research*, 59(5), 2011.

[BP76] E. Balas and M. W. Padberg. Set partitioning: a survey. *SIAM Review*, 18:710–760, 1976.

[BS97] J. Bramel and D. SimchiLevi. On the effectiveness of set covering formulations for the vehicle routing problem with time windows. *Operations Research*, 45(2):295–301, 1997.

[CE69] N. Christofides and S. Eilon. An algorithm for the vehicle dispatching problem. *Operational Research Quarterly*, 20:309–318, 1969.

[Cha05] A. Chabrier. Vehicle routing problem with elementary shortest path based column generation. *Computers & Operations Research*, 33:2972–2990, 2005.

[CLM01] J.-F. Cordeau, G. Laporte, and A. Mercer. A unified tabu search heuristic for vehicle routing problems with time windows. *Journal of the Operational Research Society*, 52(8):928–936, 2001.

[CLR07] J.-F. Cordeau, G. Laporte, and S. Ropke. Models and branch-and-cut algorithms for pickup and delivery problems with time windows. *Networks*, 49(4):258–272, 2007.

[CM12] J.-F. Cordeau and M. Maischberger. A parallel iterated tabu search heuristic for vehicle routing problems. *Computers & Operations Research*, 39(9):2033–2050, 2012.

[CMT79] N. Christofides, A. Mingozzi, and P. Toth. The vehicle routing problem. In N.Christofides, A. Mingozzi, P. Toth, and C. Sandi, editors, *Combinatorial Optimization*, pages 315 – 338. Wiley, Chichester, 1979.

[CMT81] N. Christofides, A. Mingozzi, and P. Toth. Exact algorithms for the vehicle routing problem based on spanning tree and shortest path relaxation. *Mathematical Programming*, 10:255–280, 1981.

[CPL09] CPLEX. *IBM ILOG CPLEX 12.1 callable library.* ILOG, 2009.

[DDS84] M. Desrochers, J. Desrosiers, and F. Soumis. Routing with time windows by column generation. *Networks*, 14(4):545–565, 1984.

[DDS92] M. Desrochers, J. Desrosiers, and M. M. Solomon. A new optimization algorithm for the Vehicle-Routing problem with time windows. *Operations Research*, 40(2):342–354, 1992.

[DHL08] G. Desaulniers, A. Hadjar, and F. Lessard. Tabu search, partial elementarity, and generalized k-Path inequalities for the vehicle routing problem with time windows. *Transportation Science*, 42(3):387–404, 2008.

[DKM+99] M. Desrochers, N. Kohl, O. B. G. Madsen, M. M. Solomon, and F. Soumis. 2-path cuts for the vehicle routing problem with time windows. *Transportation Science*, 33(1):101–116, 1999.

[DLP05] E. Danna and C. Le Pape. *Column Generation*, chapter Accelerating branch-and-price with local search: A case study on the vehicle routing problem with time windows, pages 90–130. Desaulniers, G., Desrosiers, J., Solomon, M.M., New York, springer edition, 2005.

[EPGR09] G. Desaulniers E. Prescott-Gagnon and L.-M. Rousseau. A branch-and-price based large neighborhood search algorithm

for the vehicle routing problem with time windows. *Networks*, 54(4):190–204, 2009.

[FJM97] M. L. Fisher, K. O. Jornsten, and O. B. G. Madsen. Vehicle routing with time windows: two optimization algorithms. *Operations Research*, 45(3):488–492, 1997.

[FLL+06] R. Fukasawa, H. Longo, J. Lysgaard, M. Poggi de Aragão, M. Reis, E. Uchoa, and R.F. Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical Programming*, 106:491–511, 2006.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wisley, 2002.

[FS04] Susan Fowler and Victor Stanwick. *WEB APPLICATION DESIGN HANDBOOK, Best Practices for Web-Based Software*. Morgan Kaufmann, 2004.

[GHL94] M. Gendreau, A. Hertz, and G. Laporte. A tabu search heuristic for the vehicle routing problem. *Management Science*, 40(10):1276–1290, 1994.

[Gol13] Riccardo Golia. Introduzione ai design pattern. `http://msdn.microsoft.com/it-it/library/cc185081.aspx`, 2013.

[Gro14] Web Hypertext Application Technology Working Group. Html, living standard. `http://www.whatwg.org/specs/web-apps/current-work/multipage/introduction.html`, Feb 2014.

[HHI08] M. Yagiura H. Hashimoto and T. Ibaraki. An iterated local search algorithm for the time-dependent vehicle routing problem with time windows. *Discrete Optimization*, 5:434–456, 2008.

[How04] Rob Howard. Provider model design pattern and specification. http://msdn.microsoft.com/en-us/library/ms972319.aspx, March 2004.

[HP93] K. Hoffman and M. W. Padberg. Solving airline crew scheduling problems by branch and cut. *Management Science*, 39:657–682, 1993.

[HY08] H. Hashimoto and M. Yagiura. A path relinking approach with an adaptive mechanism to control parameters for the vehicle routing problem with time windows. *Lecture Notes in Computer Science*, 4972:254–265, 2008.

[IEE00] Ieee recommended practice for architectural description of software-intensive systems. Software Engineering Standards Committee, Sep 2000.

[IIN+08] T. Ibaraki, S. Imahori, K. Nonobe, K. Sobue, T. Uno, and M. Yagiura. An iterated local search algorithm for the vehicle routing problem with convex time penalty functions. *Discrete Applied Mathematics*, 156:2050–2069, 2008.

[Int14] Internazionalizzazione e localizzazione. http://it.wikipedia.org/wiki/Internazionalizzazione_e_localizzazione, February 2014.

[ISO08] Ergonomic requirements for office work with visual display terminals. http://www.iso.org/iso/catalogue_detail.htm?csnumber=16883, 2008.

[IV06] S. Irnich and D. Villeneuve. The Shortest-Path problem with resource constraints and k-Cycle elimination for k>=3. *INFORMS Journal on Computing*, 18(3):391–406, 2006.

[JPSP08] M. Jepsen, B. Petersen, S. Spoorendonk, and D. Pisinger. Subset-Row inequalities applied to the Vehicle-Routing problem with time windows. *Operations Research*, 56(2):497–511, 2008.

[Kal08] B. Kallehauge. Formulations and exact algorithms for the vehicle routing problem with time windows. *Computers & Operations Research*, 35(7):2307–2330, 2008.

[KLM06] B. Kallehauge, J. Larsen, and O. B. G. Madsen. Lagrangian duality applied to the vehicle routing problem with time windows. *Computers & Operations Research*, 33(5):1464–1487, 2006.

[KLMS05] B. Kallehauge, J. Larsen, O. B. G. Madsen, and M. M. Solomon. *Column Generation*, chapter Vehicle Routing Problem with Time Windows, pages 67–98. GERAD 25th Anniversary Series. Desaulniers, G., Desrosiers, J., Solomon, M.M., New York, springer edition, 2005.

[KM97] N. Kohl and O. B. G. Madsen. An optimization algorithm for the vehicle routing problem with time windows based on lagrangian relaxation. *Operations Research*, 45(3):395–406, 1997.

[Kru06] Steve Krug. *Don't Make Me Think*. New Riders, 2006.

[LLE04] J. Lysgaard, A. N. Letchford, and R. W. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming Ser. A*, 100:423–445, 2004.

[LPR08] N. Labadi, C. Prins, and M. Reghioui. A memetic algorithm for the vehicle routing problem with time windows. *RAIRO - Operations Research*, 42(3), 2008.

[LST03] H. C. Lau, M. Sim, and K. M. Teo. Vehicle routing problem with time windows and a limited number of vehicles. *European Journal of Operational Research*, 148(3):559–569, 2003.

[Lys06] J. Lysgaard. Reachability cuts for the vehicle routing problem with time windows. *European Journal of Operational Research*, 175(1):210–223, 2006.

[LZ07] A. Lim and X. Zhang. A two-stage heuristic with ejection pools and generalized ejection chains for the vehicle routing problem with time windows. *INFORMS Journal on Computing*, 19(3):443–457, 2007.

[MB05] D. Mester and O. Bräysy. Active guided evolution strategies for large-scale vehicle routing problems with time windows. *Computers & Operations Research*, 32:1593–1614, 2005.

[MBRB99a] A. Mingozzi, M. A. Boschetti, S. Ricciardelli, and L. Bianco. A set partitioning approach to the crew scheduling problem. *Operations Research*, 47(6):873–888, 1999.

[MBRB99b] A. Mingozzi, M.A. Boschetti, S. Ricciardelli, and L. Bianco. A set partitioning approach to the crew scheduling problem. *Operations Research*, 47:873–888, 1999.

[MCH94] A. Mingozzi, N. Christofides, and E. A. Hadjiconstantinou. An exact algorithm for the vehicle routing problem based on the set partitioning formulation. Technical report, University of Bologna, 1994.

[Mic06] Guidelines for test-driven development. `http://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx`, May 2006.

[MT90] Silvano Martello and Paolo Toth. An exact algorithm for large unbounded knapsack problems. *Operations Research Letters*, 9, 1990.

[NB09] Y. Nagata and O. Bräysy. Edge assembly based memetic algorithm for the capacitated vehicle routing problem. *Networks*, 54(4):205–215, 2009.

[PR07] D. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. *Computers & Operations Research*, 34:2403–2435, 2007.

[PRI09] C.D. Tarantilis P.P. Repoussis and G. Ioannou. Arc-guided evolutionary algorithm for the vehicle routing problem with time windows. *EEE Transactions on evolutionary computation*, 13(3):624–647, 2009.

[Rop05] S. Ropke. *Heuristic and exact algorithms for vehicle routing problems*. PhD thesis, Computer science department at the University of Copenhagen (DIKU), 2005.

[Rop10] S. Ropke. Private communication. 2010.

[RS06] G. Righini and M. Salani. Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. *Discrete Optimization*, 3(3):255–273, September 2006.

[RS08] G. Righini and M. Salani. New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Networks*, 51(3):155–170, May 2008.

[RS09] G. Righini and M. Salani. Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic program-

ming. *Computers & Operations Research*, 36(4):1191–1203, April 2009.

[Sol86] M. M. Solomon. On the worst-case performance of some heuristics for the vehicle routing and scheduling problem with time window constraints. *Networks*, 16(2):161–174, 1986.

[Sol87] M. M. Solomon. Algorithms for the vehicle routing and scheduling problem with time window constraints. *Operations Research*, 35:234–265, 1987.

[spe05] SPEC: standard performance evaluation corporation. www.spec.org, 2005.

[Tsi92] J. N. Tsitsiklis. Special cases of traveling salesman and repairman problems with time windows. *Networks*, 22(3):263–282, 1992.

[TV02] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*. Society for Industrial and Applied Mathematics, 2002.

[TVP11] M. Gendreau T. Vidal, T.G. Crainic and C. Prins. A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time windows. Technical report, CIRRELT, Oct 2011.

[W3C09] The World Wide Web Consortium W3C. Document object model (dom). `http://www.w3.org/DOM/`, Jan 2009.

[Wik14a] Html5. `http://it.wikipedia.org/wiki/HTML5`, February 2014.

[Wik14b] Model view viewmodel. `http://en.wikipedia.org/wiki/Model_View_ViewModel`, February 2014.

[Wik14c] Problema del commesso viaggiatore. `http://it.wikipedia.org/wiki/Problema_del_commesso_viaggiatore`, February 2014.

[YND10] O. Bräysy Y. Nagata and W. Dullaert. A penalty-based edge assembly memetic algorithm for the vehicle routing problem with time windows. *Computers & Operations Research*, 7:724–737, 2010.