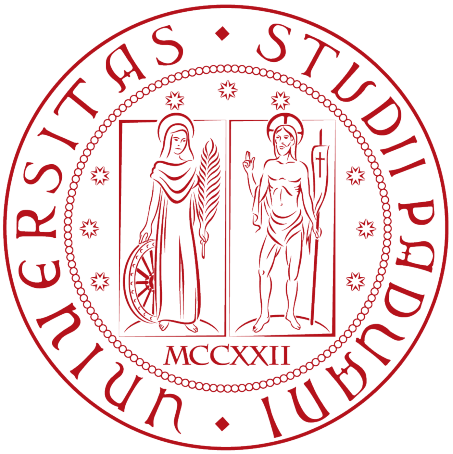
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Containerizzazione ed orchestrazione di soluzioni software applicative .NET tramite utilizzo di Docker & Docker Compose

Tesi di laurea triennale

Relatore

Prof. Massimo Marchiori

Laureando

Edoardo Caregnato

Anno Accademico 2021-2022

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

— Oscar Wilde

Dedicato a ...

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, dal laureando Edoardo Caregnato presso l'azienda PAT - Infinte Solutions. Gli obiettivi da raggiungere, concordati tra Azienda ed Università, al fine di completare con successo l'esperienza di stage erano questi di seguito esplicati.
In primo luogo è stato richiesto lo studio individuale relativo alle differenze architetturali tra Container e Virtual Machine, con relativa discussione ed esposizione di quanto elaborato al Tutor aziendale Ruggero Maffei. In secondo luogo è stato richiesto uno studio individuale di Docker e Docker-Compose e delle relative API di automation. Lo scopo finale dello studio relativo alle due tecnologie appena citate era quello di predisporre un ambiente totalmente compatibile al fine di eseguire con successo i due applicativi di punta dell'Azienda, ovvero HDA e CX studio. Dopo un'attenta analisi sulla fattibilità e tempistiche del progetto, è stato concordato, assieme all'Azienda, di concentrare l'esperienza curricolare sulla containerizzazione dell'applicativo HDA con tutte le sue relative estensioni. Il terzo obiettivo dello stage curricolare è stata la predisposizione dei relativi container atti all'esecuzione dell'applicativo HDA assieme a tutti gli strumenti di monitoraggio richiesti dall'Azienda. Quarto ed ultimo obbiettivo è stato lo studio e la creazione di container atti all'aggiornamento della versione di HDA, con relativo studio della possibilità di automazione di quest'ultimo.

"Life is really simple, but we insist on making it complicated"

— Confucius

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. NomeDelProfessore, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ho desiderio di ringraziare poi i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.

Padova, Dicembre 2022 Edoardo Caregnato

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Introduzione al contesto applicativo.

Esempio di utilizzo di un termine nel glossario Application Program Interface (API).

Esempio di citazione in linea **site:agile-manifesto**.

Esempio di citazione nel pie' di pagina citazione[1]

1.1 L'azienda

Descrizione dell'azienda.

1.2 L'idea

Introduzione all'idea dello stage.

1.3 Organizzazione del testo

Il secondo capitolo descrive ...

Il terzo capitolo approfondisce ...

Il quarto capitolo approfondisce ...

Il quinto capitolo approfondisce ...

Il sesto capitolo approfondisce ...

Nel settimo capitolo descrive ...

[1] **womak:lean-thinking**.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

* gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
* per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola*[g];
* i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Container VS Virtual Machine

Introduzione al capitolo
Nel presente capitolo si esporranno le principali differenze tra un'architettura basata su macchine virtuali ed un'altra basata invece su container.

2.1 Differenze architetturali tra Container e VM

La **virtualizzazione** è un insieme di software in grado di astrarre componenti hardware, permettendo l'esecuzione, anche simultanea, di più sistemi operativi su un singolo client. Verso la fine degli anni '90, la virtualizzazione ha cominciato ad essere sempre più utilizzata in ambienti *enterprise*, permettendo un aumento di scalabilità e flessibilità dell'infrastruttura informatica aziendale riducendone notevolmente i costi di gestione[1].
I **vantaggi** legati all'utilizzo della virtualizzazione, nello specifico, tramite l'uso di una o più macchine virtuali, comportano una separazione tra il sistema operativo host e guest, fornendo una serie di accessi logici utilizzati da utenti esterni agli applicativi eseguiti in ogni macchina virtuale.
Oltre ad una esecuzione parallela, dal punto di vista dell' *host*, di molteplici applicativi, un'architettura a VM è più facilmente **manutenibile**: una macchina virtuale infatti, può essere facilmente aggiornata, avviata o arrestata in base alle esigenze di carico (ex: *load-balancing*) o aziendali. La virtualizzazione, inoltre, aumenta l'**affidabilità** dell'intero sistema, in quanto garantisce l'**isolamento** di programmi e servizi i quali non andranno in conflitto tra di loro, contenendo, in aggiunta, il numero di server fisici presenti in datacenter nel caso in cui molteplici macchine virtuali vengano eseguite su un singolo *host*, con conseguente notevole riduzione dei costi. Un ulteriore vantaggio della virtualizzazione si rivela nel caso di *disaster recovery*, dove l'intero sistema operativo *guest* può essere facilmente ripristinato su un altro server, indipendentemente dall'*hardware*, riducendo così notevolmente i tempi di indisponibilità di servizio (*downtime*) in caso di guasto favorendo una maggior facile e rapida procedura di data *recovery*.
Esistono diversi tipi di virtualizzazione: ***native*** e ***hosted***. Una virtualizzazione di tipo **native** si appoggia direttamente all'*hardware host*, controllandolo direttamente per garantire tutte le funzionalità della virtualizzazione, come ad esempio **Hyper-V** della Microsoft[2] oppure l'applicativo **Xen** ampiamente utilizzato anche nell'ambiente Cloud di Amazon. La virtualizzazione di tipo **hosted** è invece in esecuzione sul sistema

[1] **fonte: https://www.vmware.com/it/solutions/virtualization.html**.
[2] **questa funzionalita' e' presente solamente nelle versioni Pro e Server di Windows 10**.

operativo *host* senza alcuna interfaccia diretta con l'hardware del computer. Questo tipo di virtualizzazione è molto diffusa, in quanto permette di accedere, in una maniera semplice ed immediata, al sistema operativo *host* e *guest* in maniera simultanea. Gli applicativi più usati in ambito enterprise che usano un tipo di virtualizzazione *hosted* sono, ad esempio, *VMware* o *VirtualBox*.
Nella seguente figura è rappresentato un sistema operativo Windows 10 pro virtualizzato con in esecuzione il browser web "Firefox" in virtualizzazione *native* tramite software Hyper-V: Di seguito, un esempio di sistema operativo Windows 10 Pro virtualizzato con in esecuzione il browser web "Firefox" in virtualizzazione *hosted* tramite le due soluzioni software appena descritte: Nello screenshot sottostante, invece, l'applicativo "Firefox" è eseguito tramite il programma di virtualizzazione "VMware":

Al fine di permettere al sistema operativo *host* la virtualizzazione di uno o più sistemi operativi, è necessario installare un *hypervisor*[3], *native* o *hosted*, ovvero uno strato software che si interfacci e gestisca tutte le istanze di macchine virtuali in esecuzione sulla macchina locale.
La virtualizzazione non è priva di svantaggi. Il primo tra tutti, è appunto la necessità di dover *virtualizzare* un intero sistema operativo al fine di eseguire l'applicativo virtuale desiderato. Questo vincolo obbligatorio implica un consumo di memoria RAM e di storage non indifferente anche solo per eseguire il singolo sistema operativo virtualizzato senza alcuna applicazione virtuale in esecuzione. Ne consegue quindi, che un'architettura a macchine virtuali avrà bisogno di uno spazio di *storage* e di un quantitativo di memoria *RAM*[4] installata sul server non indifferente. Anche in termini di consumo *CPU*, la virtualizzazione di molteplici sistemi operativi con le relative applicazioni virtualizzate in esecuzione può comportare grossi carichi prestazionali al server fisico, in quanto la CPU dell'host dovrà servire ed eseguire ogni sistema operativo di ogni istanza di virtualizzazione.
Dal punto di vista della sicurezza, quando si virtualizza un sistema operativo, sia nella virtualizzazione *native* che *hosted*, alcuni registri CPU sono direttamente esposti alla macchina virtuale come, ad esempio, i registri **VT-x** e **VT-d** del processore[5][6]. Questi registri permettono al processore di non rendere accessibile la totalità dei suoi registri all'hypervisor e di controllare le chiamate dirette al *DMA* da parte delle soluzioni software virtualizzate.
Relativamente alla condivisione della rete tra macchine virtuali e host fisico, nel caso in cui si fosse installato un commutatore di rete virtuale di tipo **NAT**, la scheda di rete dell'host e il relativo traffico sarebbe esposta a tutto il set applicativo virtualizzato e viceversa, con conseguente mancante isolamento tra macchine virtuali stesse ed host fisico. Ne conseguirebbe quindi, che eventuali condivisioni di rete, o connessioni applicative, sarebbero disponibili a tutto il set di macchine virtuali. Una possibile soluzione a questo problema potrebbe essere il passaggio da commutatore virtuale *NAT* ad un tipo di commutatore virtuale che riesca ad isolare le singole macchine virtuali tra di esse e l'host fisico, anche, nel caso più estremo, assegnando ad ogni macchina virtuale una propria scheda di rete ed una propria *VLAN* di rete dedicata[7].
Virtualizzare un intero sistema operativo implica, come abbiamo appena analizzato, un elevato consumo di risorse fisiche, specialmente nel caso in cui, per esigenze lavorative, si debba ricorrere ad una multipla virtualizzazione di sistemi operativi dove, in ognuno,

[3] **installabile solamente se il processore supporta la virtualizzazione e se quest'ultima e' abilitata da BIOS**.
[4] **il tipo di RAM "ECC" risulta preferibile ma non obbligatorio**.
[5] **e' necessario abilitare le estensioni di virtualizzazione da BIOS della scheda madre.**.
[6] **nel caso di architettura avente processori Intel**; **IOMMU per architetture basate su processori AMD.**.
[7] **per creare o impostare una VLAN**; **fare riferimento al router/firewall o allo switch di rete**.

viene eseguita una specifica applicazione che deve essere accessibile ad altri *client*. Uno dei principali aspetti positivi di un'architettura a container sta proprio nel poter virtualizzare (o *containerizzare* nel caso appunto di container) una singola e specifica applicazione senza la necessità di inglobare un intero sistema operativo nell'immagine virtuale. L'esecuzione dell'applicativo, nonostante appunto la mancanza di un sistema operativo, sarà comunque possibile grazie a chiamate di sistema al kernel del s.o. host. Ne consegue quindi, che il container applicativo risultante di un'applicazione *containerizzata* è di gran lunga di dimensione inferiore rispetto all' immagine[8] della stessa applicazione *virtualizzata*, causa appunto, in primis, mancanza di sistema operativo.
Un container è quindi una singola unità atomica contenente l'applicativo (il programma *containerizzato*) con i relativi file atti alla sua corretta esecuzione senza l'immagine di un sistema operativo completo. Al momento dell'esecuzione del container, l'applicazione *containerizzata* verrà eseguita immediatamente sopra lo stato del sistema operativo host, attraverso l'aiuto del Docker Engine, senza alcun hypervisor come, ad esempio, nel caso dell'architettura a macchine virtuali. Una rappresentazione grafica del concetto appena descritto è data dalla seguente immagine: Un'architettura a **container** infatti, a differenza dell'architettura a macchine virtuali, garantisce un'esecuzione separata e protetta di ogni singolo applicativo compatibile con il sistema operativo host, indipendentemente dal numero di container presenti nel sistema o dal tipo di interfaccia di rete. E' possibile, inoltre, far coesistere multipli container di uno stesso applicativo in esecuzione nello stesso momento (anche sfruttando il *load-balancing*, come si accennerà nel corso di questa tesi) assegnandoci, esattamente come con le macchine virtuali, eventuali indirizzi IP statici, CPU limit e disk quota. Essendo un container una **sandbox** applicativa indipendente dal sistema operativo, i dati generati dalla sua esecuzione sono destinati a scomparire nell'eventualità in cui il container venisse distrutto. Per ovviare a questo problema, si può ricorrere ad una tecnica di volume-mapping, ovvero una tecnica che permette di esporre il **filesystem** interno al container permettendone quindi la lettura e scrittura direttamente da parte dell'host. La tecnica appena accennata sarà trattata in maniera più approfondita nel corso della lettura di questa tesi. Godendo i container di un approccio standardizzato per la loro costruzione ed esecuzione, è quindi di facile intuizione la facilità di *portabilità* di questi. Un container, infatti, può essere distribuito su una nuova piattaforma in maniera estremamente veloce e senza alcuna modifica allo stesso. Ne consegue, che un eventuale rilascio e distribuzione di un applicativo interno ad un container può essere molto velicizzato rispetto alla stessa operazione svolta invece su un'architettura virtuale. Relativamente sempre allo sviluppo e distribuzione di un applicativo, tramite container è possibile un controllo di versione dell'applicativo stesso più flessibile: è possibile infatti gestire le versioni del codice con le relative dipendenze inglobando il tutto in un container, formando quindi un' unità atomica di più facile manutenzione e distribuzione. Un altro dei vantaggi decisivi di un'architettura a container è la sua facilità di gestione. L'avvio, rimozione o la duplica dei container è un'operazione relativamente meno onerosa rispetto alla controparte nelle macchine virtuali (basti solo pensare al tempo di *boot* del sistema operativo), e può essere facilmente automatizzata e gestita dal Docker Engine. Ne consegue quindi che la scalabilità, ovvero la facilità di modifica dell'infrastruttura per far fronte alle variazioni di mole di informazioni trattate o carichi di lavoro, risulta di gestione più semplice anche per la figura sistemistica interna all'azienda.
La presente immagine ritrae l'applicativo "Firefox" in esecuzione all'interno di un

[8] **inteso come dimensione in Gb del virtual disk image (*.vdi) dell'immagine virtualizzata**.

container esposto all'host nella porta 5800:

2.2 Analisi di Docker: architettura e funzionalità

Una delle piattaforme di riferimento più utilizzate e supportate in ambito container è appunto Docker. Nato dall'Azienda dotCloud nel 2003 e pubblicato successivamente come progetto open-source, è una piattaforma atta alla gestione dei container. Tramite Docker Desktop[9], è possibile monitorare, avviare, arrestare e gestire eventuali container creati. Come già anticipato nel corso di questa tesi, Docker lavora a stretto contatto con il suo motore, ovvero Docker Engine, che si occupa appunto della creazione e manutenzione dei container nel sistema. Il *Docker Daemon* è il processo principale di Docker, cui compito è appunto la gestione e l'orchestrazione dei container nel sistema operativo dove è installato. Al momento della creazione di un nuovo container, questo processo interroga i *Docker Registries* per verificare la presenza di una immagine già pronta all'uso del programma che si vuole *containerizzare*. I Docker Registries sono un insieme di immagini di container pronte all'uso e scaricabili gratuitamente da qualunque client Docker. Queste immagini sono create e manutenzionate dalla Docker Community, e permettono allo sviluppatore di avere delle immagini (o layer) già pronti, testati e funzionanti al fine di costruire con successo la propria immagine di applicativo virtuale. Il Docker Registries più comune ed utilizzato è "DockerHub".
Come già menzionato nel corso di questo documento, tramite Docker è possibile costruire delle immagini di un applicativo virtuale. Un'immagine è un set di comandi cui scopo è quello di creare, una volta eseguito, un container. Una immagine può basarsi, a sua volta, su altre immagini, ed il set di comandi che espandono l'immagine sorgente compongono, a sua volta, la nuova immagine applicativa. I comandi per costruire con successo un'immagine vengono scritti su un apposito file, chiamato appunto "*dockerfile*". Un **dockerfile** è un documento di testo, senza alcuna estensione, contenente una serie di passi ed istruzioni atti alla corretta creazione di una immagine di un applicativo. Il Docker Daemon, per costruire correttamente un'immagine, leggerà ed eseguirà in maniera sequenziale ogni comando trascritto all'interno del dockerfile. Al termine della lettura del dockerfile, si avrà quindi un'immagine eseguibile di un applicativo. Un **container** è un'istanza di esecuzione di una immagine, a cui è attribuito un nome, arbitrario, dal Docker Daemon.

2.3 Creazione di container vs creazione di VM

La creazione di una VM è possibile tramite apposito *hypervisor*. Tuttavia, al momento della creazione, l'utente deve essere a conoscenza della quantità massima di risorse *hardware* da destinare alla stessa. Questa stima, oltre ad includere il costo[10] richiesto dal programma da virtualizzare che si andrà ad installare all'interno della VM, deve includere anche il costo, almeno soddisfacente i requisiti minimi, relativo al sistema operativo. La seguente schermata rappresenta la creazione della macchina virtuale con 8Gb di RAM e 80Gb di disco rigido dedicati:
Terminata la fase di analisi dei costi, sarà necessario installare manualmente il sistema operativo all'interno della macchina virtuale appena creata. Per fare ciò, bisognerà avviare la macchina virtuale da un supporto di **boot**, come ad esempio una immagine

[9] **presente solamente per le versioni Windows**.
[10] **inteso come quantitativo di risorse fisiche da allocare**.

ISO di un sistema operativo, avviabile. Successivamente alla fase di avvio di installazione della VM, si passerà alla fase vera e propria di installazione del sistema operativo, riassunta nelle seguenti immagini: Una volta ottenuto un sistema operativo funzionante ed avviabile, per ottimizzarne al meglio le prestazioni, tramite ad esempio installazione dei relativi driver video virtuali, ed espanderne le funzionalità, come la possibilità di accedere a volume-mapping condivisi con il sistema operativo host, è necessario installare le relative **guest additions**. Una volta fatti i passi sopra-descritti, si avrà un sistema operativo virtualizzato completamente funzionante, pronto per l'installazione di tutto il software applicativo desiderato.
Come precedentemente affermato, un container è un'unità atomica costituita principalmente da una o più applicazioni e dalle librerie di sistema operativo atte al loro corretto funzionamento. Al fine di poter generare ed utilizzare correttamente un container, è necessario prima costruirne il relativo **dockerfile**. Le seguenti immagini mostrano i corretti passi per la costruzione di un container con, all'interno, l'applicativo "Firefox": La creazione di multipli container cloni, ovvero container generati da una singola immagine, è un'operazione di semplice e veloce esecuzione, in quanto il Docker Engine, durante la costruzione di ogni container, adotta un meccanismo di *caching* dei vari *layer* di cui un container è composto. Questo meccanismo permette quindi una costruzione, o aggiornamento, di un container molto più veloce rispetto ad una sua completa ri-creazione, in quanto si useranno, se compatibili, i layer in cache già precedentemente costruiti al fine di costruire il nuovo container contenente l'applicativo, nuovo o aggiornato, desiderato.
Durante tutta la durata dello stage si è utilizzata la piattaforma "Docker" per la creazione delle varie immagini, con relativi container, per gli applicativi specificati nel sommario e nell'introduzione.

Capitolo 3

Descrizione dello stage

Breve introduzione al capitolo

3.1 Introduzione al progetto

3.2 Analisi preventiva dei rischi

Durante la fase di analisi iniziale sono stati individuati alcuni possibili rischi a cui si potrà andare incontro. Si è quindi proceduto a elaborare delle possibili soluzioni per far fronte a tali rischi.

1. Performance del simulatore hardware
Descrizione: le performance del simulatore hardware e la comunicazione con questo potrebbero risultare lenti o non abbastanza buoni da causare il fallimento dei test.
Soluzione: coinvolgimento del responsabile a capo del progetto relativo il simulatore hardware.

3.3 Requisiti e obiettivi

3.4 Pianificazione

Capitolo 4

Analisi dei requisiti

Breve introduzione al capitolo

4.1 Casi d'uso

Per lo studio dei casi di utilizzo del prodotto sono stati creati dei diagrammi. I diagrammi dei casi d'uso (in inglese *Use Case Diagram*) sono diagrammi di tipo Unified Modeling Language (UML) dedicati alla descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso. Essendo il progetto finalizzato alla creazione di un tool per l'automazione di un processo, le interazioni da parte dell'utilizzatore devono essere ovviamente ridotte allo stretto necessario. Per questo motivo i diagrammi d'uso risultano semplici e in numero ridotto.

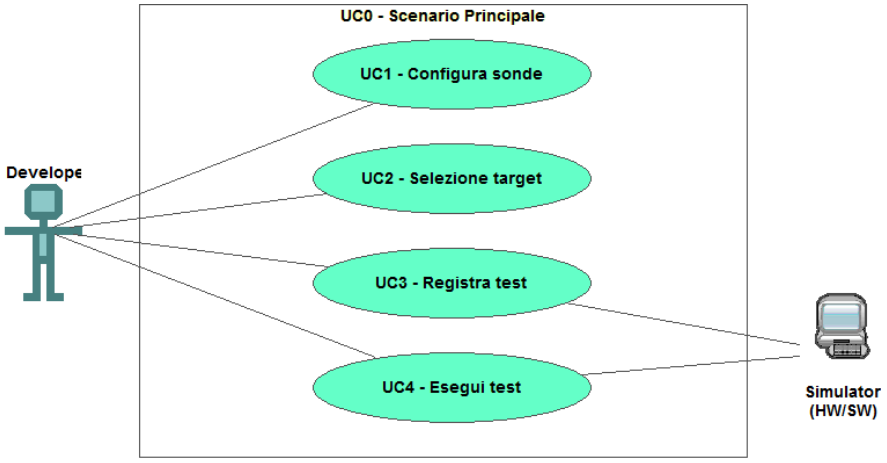


Figura 4.1: Use Case - UC0: Scenario principale

UC0: Scenario principale

Attori Principali: Sviluppatore applicativi.

Precondizioni: Lo sviluppatore è entrato nel plug-in di simulazione all'interno dell'IDE.

Descrizione: La finestra di simulazione mette a disposizione i comandi per configurare, registrare o eseguire un test.

Postcondizioni: Il sistema è pronto per permettere una nuova interazione.

4.2 Tracciamento dei requisiti

Da un'attenta analisi dei requisiti e degli use case effettuata sul progetto è stata stilata la tabella che traccia i requisiti in rapporto agli use case.
Sono stati individuati diversi tipi di requisiti e si è quindi fatto utilizzo di un codice identificativo per distinguerli.
Il codice dei requisiti è così strutturato R(F/Q/V)(N/D/O) dove:

R = requisito

F = funzionale

Q = qualitativo

V = di vincolo

N = obbligatorio (necessario)

D = desiderabile

Z = opzionale

Nelle tabelle 4.1, 4.2 e 4.3 sono riassunti i requisiti e il loro tracciamento con gli use case delineati in fase di analisi.

Tabella 4.1: Tabella del tracciamento dei requisti funzionali

Requisito	Descrizione	Use Case
RFN-1	L'interfaccia permette di configurare il tipo di sonde del test	UC1

Tabella 4.2: Tabella del tracciamento dei requisiti qualitativi

Requisito	Descrizione	Use Case
RQD-1	Le prestazioni del simulatore hardware deve garantire la giusta esecuzione dei test e non la generazione di falsi negativi	-

Tabella 4.3: Tabella del tracciamento dei requisiti di vincolo

Requisito	Descrizione	Use Case
RVO-1	La libreria per l'esecuzione dei test automatici deve essere riutilizzabile	-

Capitolo 5

Progettazione e codifica

Breve introduzione al capitolo

5.1 Tecnologie e strumenti

Di seguito viene data una panoramica delle tecnologie e strumenti utilizzati.

Tecnologia 1

Descrizione Tecnologia 1.

Tecnologia 2

Descrizione Tecnologia 2

5.2 Ciclo di vita del software

5.3 Progettazione

Namespace 1

Descrizione namespace 1.

Classe 1: Descrizione classe 1

Classe 2: Descrizione classe 2

5.4 Design Pattern utilizzati

5.5 Codifica

Capitolo 6

Verifica e validazione

Capitolo 7

Conclusioni

7.1 Consuntivo finale

7.2 Raggiungimento degli obiettivi

7.3 Conoscenze acquisite

7.4 Valutazione personale

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia