



UNIVERSITÀ DEGLI STUDI DI MESSINA
FACOLTÀ DI SCIENZE
Tesina di Programmazione 2

Dietlytics-APP

Studenti:

Salvatore Bertoncini 445857

Zavettieri Sergio 447265

ANNO ACCADEMICO 2014 – 2015

Sommario

1	Richiesta del cliente	6
2	Introduzione	6
2.1	Definizioni	6
2.1.1	Cosa è il software	6
2.1.2	Prodotti generici vs Prodotti specifici	6
2.1.3	Programma vs prodotto	6
2.1.4	Costi	6
2.1.5	Manutenzione	6
2.1.6	Software engineering	7
2.1.7	Differenza tra software engineering e computer science	7
2.2	Fondamenti dell'ingegneria del software	7
2.2.1	Principi	7
2.2.1	Metodi e metodologie	7
2.2.2	Strumenti, procedure e paradigmi	7
2.2.3	Qualità del software	7
3	Cicli di vita del software	8
3.2	Modello di ciclo di vita del software (CVS)	8
3.3	Modello a cascata	8
3.3.1	Organizzazione sequenziale: <i>fasi alte</i> del processo	8
3.3.3	Pro e contro del modello a cascata	9
3.4	Modello verification e validation (V&V)	9
3.5	Modello trasformatzionale	9
3.6	Modello di sviluppo basato sul riuso	9
3.7	Modello evolutivo (a prototipazione)	9
3.8	Modello incrementale	9
3.9	Modello a spirale	10
3.10	Modelli e valutazione dei rischi	10
3.11	Scopo dell'ingegneria del software	10
4	Project management	11
4.1	Team di sviluppo	11
4.2	Stesura del piano del progetto	11
4.3	Management dei rischi	12
4.3.1	Identificazione	12
4.3.2	Analisi dei rischi	12
4.3.3	Pianificazione dei rischi	12
4.3.4	Monitoraggio dei rischi	12
5	UML (Unified Modeling Language)	13
5.1.1	Diagrammi dei casi d'uso (use case diagrams)	13
5.2	Diagrammi di classi (class diagrams)	13
5.2.1	Associazione	14
5.2.2	Aggregazione	14
5.2.3	Composizione	14
5.2.4	Generalizzazione (o ereditarietà)	14
5.3	Diagrammi sequenziali (sequence diagrams)	14
5.4	Diagramma a stati (state chart)	15
5.5	Diagrammi delle attività (activity diagrams)	15
5.6	Raggruppamento (packages)	15
6	Raccolta dei requisiti (requirements elicitation)	16
6.1	Classificazione dei requisiti	16
6.1.1	Requisiti funzionali	16
6.1.2	Requisiti non funzionali	16

6.2	Validazione dei requisiti	17
6.3	Greenfield engineering, re-engineering, interface engineering	17
6.4	Attività della raccolta dei requisiti	17
6.5	Identificare gli attori	17
6.6	Identificare gli scenari	17
6.7	Identificare i casi d'uso	18
6.8	Raffinare i casi d'uso	18
6.9	Identificare le relazioni tra attori e casi d'uso	18
6.10	Identificare gli oggetti partecipanti	19
6.11	Identificare i requisiti non funzionali	19
6.12	5.4 Gestire la raccolta dei requisiti	19
7	Analisi dei requisiti	20
7.1	Concetti dell'analisi.....	20
7.1.1	Il modello ad oggetti	20
7.1.2	Il modello dinamico	20
7.1.3	Entity, Boundary (oggetti frontiera) e Control object	20
7.1.4	Attività dell'analisi (trasformare un caso d'uso in oggetti).....	20
7.1.5	Identificare gli oggetti entity	21
7.1.6	Identificare gli oggetti boundary	21
7.1.7	Identificare gli oggetti controllo.....	21
7.1.8	Mappare casi d'uso in oggetti con sequence diagrams	22
7.1.9	Identificare le associazioni	22
7.1.10	Identificare le aggregazioni.....	23
7.1.11	Identificare gli attributi.....	23
7.1.12	Modellare il comportamento e gli stati di ogni oggetto.....	23
7.1.13	Rivedere il modello dell'analisi.....	23
8	System design.....	24
8.1	Scopi criteri e architetture.....	24
8.2	Identificare gli obiettivi di design	24
8.3	Decomposizione del sistema in sottosistemi	24
8.3.1	Accoppiamento (coupling)	25
8.3.2	Coesione.....	25
8.3.3	Divisione del sistema con i layer.....	25
8.3.4	Divisione del sistema con le partizioni	25
8.4	Architetture software.....	26
8.4.1	Repository	26
8.4.2	Model/View/Control (MVC).....	26
8.4.3	Client-Server	26
8.4.4	Peer-To-Peer	26
8.4.5	Three-Tier	26
8.4.6	Considerazioni finali	26
8.4.7	Euristiche per scegliere le componenti.....	26
8.4.8	Descrizione delle attività del System Design	27
8.4.9	Mappare i sottosistemi su piattaforme e processori.....	27
8.4.10	Identificare e memorizzare i dati persistenti.....	27
8.4.11	File	27
8.6	Stabilire i controlli di accesso	28
8.6.1	Progettare il flusso di controllo globale.....	28
8.7	Identificare le condizioni limite.....	29
8.7.1	Rivedere il modello del system design	29
8.7.2	Gestione del system design	30
9	Database	31

9.1	Introduzione	31
9.2	Architetture utilizzate.....	31
8.5.1	XAMPP e DBMS	31
9.3	SQL.....	31
9.4	Glossario dei termini.....	32
9.5	Modello concettuale	33
	33
9.6	Trasformazione al modello logico.....	34
9.6.1	Traduzione di un'associazione uno a molti	34
9.6.2	Traduzione di un'associazione molti a molti	34
9.6.3	Traduzione di un'associazione uno a uno.....	34
9.6.4	Traduzione di un'associazione n- aria	34
9.6.5	Traduzione di un'associazione ricorsiva	35
9.7	Modello logico.....	35
	36
9.7.1	Dettagli tabelle.....	36
9.7.1.1	allergy	36
9.7.1.2	associallergia.....	36
9.7.1.3	typediet.....	37
9.7.1.4	diet	37
9.7.1.5	composizionecibo.....	37
9.7.1.6	composizionepasto.....	37
9.7.1.7	food	38
9.7.1.8	pasti	38
9.7.1.9	storico	38
9.7.1.10	lifestyle.....	38
9.7.1.11	sex	38
9.7.1.12	user	39
10	SQL.....	40
10.1.1	Database setup.....	40
10.2	Tabelle	40
10.2.1	allergy	40
	40
10.2.2	associallergia.....	40
	40
10.2.3	typediet.....	41
	41
10.2.4	Diet.....	41
	41
10.2.5	Composizionecibo.....	42
	42
10.2.6	Food	42
	43
10.2.7	Composizionepasto.....	43
	44
10.2.8	Pasti	44
	44
10.2.9	Lifestyle	44
	45
10.2.10	Storico	45
	45
10.2.11	Sex.....	45

.....	46
10.2.12 user	46
.....	46
11 Analisi di implementazione	47
11.1 Generale	47
11.2 Dietlyrics	47
11.3 Model-view-controller	47
11.4 Package	48
11.4.1 Client	49
11.4.2 Comunicazione	50
11.4.3 Debugger	53
11.4.4 Database	53
11.4.5 Server	54
11.4.6 Controller.....	55
11.4.7 dieta	56
11.4.8 Model.....	57
11.4.3.1 Mdiet	57
11.4.3.2 MUser	58
11.4.3.3 Mtypediet	59
11.4.3.4 MUtility	59
11.4.3.5 MCredenziali	60
11.4.3.6 MStorico.....	61
11.4.3.7 MFood	62
12 Screenshot	65
12.4 Login/registrazione	65
.....	65
12.5 Modifica dati personali.....	66
.....	66
12.6 Modifica abitudini alimentary	67
.....	67
12.7 Nuova dieta	68
.....	68
12.8 Ricerca cibo	69
.....	69
12.9 Storico	70
.....	70

1 Richiesta del cliente

Il cliente richiede l'ideazione e l'implementazione di un'applicazione da adottare per la creazione di un qualsiasi tipo di dieta.

Il sistema deve prevedere dunque l'opportunità di informatizzare esclusivamente l'area degli utenti senza prevedere quella amministrativa. Dovrà esistere all'interno del sistema solo una sezione dedicata agli utenti. Si deve dare la possibilità ai clienti al momento della registrazione di inserire oltre a nome, cognome, username, password e email anche la data di nascita, l'altezza, il peso e il sesso e di poterli modificare in futuro, di inserire le proprie abitudini alimentari e di poterle modificare in futuro, di selezionare il tipo di dieta che si vuole seguire tenendo conto di eventuali allergie e dello stile di vita adottato e di potere modificare anche questi. Le singole diete del cliente che verranno adottate attraverso un coefficiente che verrà calcolato in base alle informazioni inserite durante la registrazione, dovranno essere tutte salvate in uno storico e dovranno essere distinte attraverso la data di creazione in modo tale che il cliente le possa visualizzare in futuro. Dovrà essere prevista la stampa di una singola dieta selezionata in xml. Per quanto riguarda l'eliminazione dovrà essere il cliente stesso a potersi autoeliminare poiché non è prevista una sezione amministrativa. E' stato richiesto inoltre di inserire un campo di ricerca nel database in modo tale che un cliente possa visualizzare le singole caratteristiche di uno specifico alimento dal database in base alla quantità in grammi, stabilita di default a 100gr ma è possibile inserire la quantità desiderata.

2 Introduzione

2.1 Definizioni

2.1.1 Cosa è il software

Il software non è solo un insieme di linee di codice ma comprende anche tutta la documentazione, i case test e i manuali.

2.1.2 Prodotti generici vs Prodotti specifici

I prodotti generici sono dei software prodotti da aziende e utilizzati da un ampio bacino di utenza diversificato. I prodotti specifici sono software sviluppati ad hoc per uno specifico cliente, visionato dallo stesso. Il costo dei prodotti generici è maggiore rispetto a quello dei prodotti specifici.

2.1.3 Programma vs prodotto

Un programma è una semplice applicazione sviluppata, testata e usata dallo stesso sviluppatore. Il prodotto software viene sviluppato per terzi, è un software industriale che ha un costo di circa 10 volte superiore ad un normale programma. e deve essere corredato di documentazione, manuali e case test.

2.1.4 Costi

Il costo del software viene calcolato in base alle ore di lavoro, il software e hardware utilizzato e altre risorse di supporto. Il costo della manutenzione è superiore a quello di produzione.

2.1.5 Manutenzione

Il software dopo il suo rilascio, specie se lo stesso ha una vita lunga, ha bisogno di alcune fasi di manutenzione. Per manutenzione intendiamo sia la correzione di eventuali bug, sia l'estensione/modifica di alcune caratteristiche. Il costo della manutenzione è più elevato rispetto a quello di produzione.

2.1.6 Software engineering

Disciplina che cerca di fornire le regole per il processo di produzione del software. Lo scopo dell'ingegneria del software è di pianificare, progettare, sviluppare il software tramite lavoro di gruppo. E' possibile che vengono rilasciate più versioni del prodotto software. Tale attività ha senso per progetti di grosse dimensioni e di notevole complessità ove si rende necessaria la pianificazione.

2.1.7 Differenza tra software engineering e computer science

Mentre la computer science si occupa di creare e ottimizzare algoritmi e si occupa degli aspetti teorici dell'informatica, il software engineering si occupa della pianificazione e della progettazione con la finalità di ottenere un prodotto software.

2.2 Fondamenti dell'ingegneria del software

L'ingegneria del software si occupa principalmente di tre aspetti fondamentali: i principi, i metodi, le metodologie e gli strumenti.

2.2.1 Principi

1. Rigore e formalità
2. Affrontare separatamente le varie problematiche dell'attività
3. Modularità (divide-et-impera)
4. Anticipazione del cambiamento (scalabilità)
5. Generalità (tentare di risolvere il problema nel suo caso generale)
6. Incrementalità (lavorare a fasi di sviluppo, ognuna delle quali viene terminata con il rilascio di una release, anche se piccola).

2.2.1 Metodi e metodologie

Un metodo è una particolare procedimento per risolvere problemi specifici, mentre la metodologia è un'insieme di principi e metodi che serve per garantire la correttezza e l'efficacia della soluzione al problema.

2.2.2 Strumenti, procedure e paradigmi

Uno strumento è un artefatto che viene usato per fare qualcosa in modo migliore. Una procedura è una combinazione di strumenti e metodi finalizzati alla realizzazione di un prodotto.

2.2.3 Qualità del software

La qualità può essere riferita sia al prodotto che al processo applicato per ottenere il risultato finale. Un particolare modello di qualità (modello di McCall) dice che la qualità si basa su i seguenti tre aspetti principali:

1. *Revisione*: manutenibilità, flessibilità e verificabilità (deve rispettare i requisiti del cliente)
2. *Transizione*: portabilità, riusabilità, interoperabilità (capacità del sistema di interagire con altri sistemi esistenti)
3. *Operatività*: correttezza (conformità dello stesso rispetto ai requisiti), affidabilità, efficienza (tempo di risposta o uso della memoria), usabilità, integrità (capacità di sopportare attacchi alla sicurezza).

3 Cicli di vita del software

Un *processo software* è un insieme organizzato di attività finalizzate ad ottenere il prodotto da parte di un team di sviluppo utilizzando metodo, tecniche, metodologie e strumenti.

Il processo viene suddiviso in fasi in base ad uno schema di riferimento (ciclo di vita del software).

3.2 Modello di ciclo di vita del software (CVS)

E' una caratterizzazione descrittiva o prescrittiva di come un sistema software viene sviluppato.

I modelli CVS devono avere le seguenti caratteristiche:

1. Descrizione dell'organizzazione del lavoro nella software house.
2. Linee guida per pianificare, dimensionare il personale, assegnare budget, schedulare e gestire.
3. Definizione e scrittura dei manuali d'uso e diagrammi vari.
4. Determinazione e classificazione dei metodi e strumenti più adatti alle attività da svolgere.

Le fasi principali di un qualsiasi CVS sono le seguenti:

1. **Definizione** (si occupa del cosa).
Determinazione dei requisiti, informazioni da elaborare, comportamento del sistema, criteri di validazione, vincoli progettuali.
2. **Sviluppo** (si occupa del come)
Definizione del progetto, dell'architettura software, traduzione del progetto nel linguaggio di programmazione, collaudi.
3. **Manutenzione** (si occupa delle modifiche)
Miglioramenti, correzioni, prevenzione, adattamenti.

3.3 Modello a cascata

Definisce che il processo segua una progressione sequenziale di fasi senza ricicli, al fine di controllare meglio tempi e costi. Inoltre definisce e separa le varie fasi e attività del processo in modo da minimizzare la sovrapposizione tra di esse. Ad ogni fase viene prodotto un semilavorato con la relativa documentazione e lo stesso viene passato alla fase successiva (*milestone*). I prodotti ottenuti da una fase non possono essere modificati durante il processo di elaborazione delle fasi successive.

3.3.1 Organizzazione sequenziale: *fasi alte* del processo

- **Studio di fattibilità**
Effettua una valutazione preliminare dei costi e dei requisiti in collaborazione con il committente. L'obiettivo è quello di decidere la fattibilità del progetto, valutarne i costi, i tempi necessari e le modalità di sviluppo.
Output: documento di fattibilità.
- **Analisi e specifica dei requisiti**
Vengono analizzate le necessità dell'utente e del dominio d'applicazione del problema.
Output: documento di specifica dei requisiti.
- **Progettazione**
Viene definita la struttura del software e il sistema viene scomposto in componenti e moduli.
Output: definizione dei linguaggi e formalismi.

3.3.2 Organizzazione sequenziale: *fasi basse* del processo

- **Programmazione e test di unità**
Ogni modulo viene codificato nel linguaggio e testato separatamente dagli altri.

- **Integrazione e test di sistema**

I moduli vengono integrati tra loro e vengono testate le loro interazioni. Viene rilasciata una *beta release* (release esterna) oppure una *alpha release* (release interna) per testare al meglio il sistema.

- **Deployment**

Rilascio del prodotto al cliente.

- **Manutenzione**

Gestione dell'evoluzione del software.

3.3.3 Pro e contro del modello a cascata

Pro

- Facile da comprendere e applicare

Contro

- L'interazione con il cliente avviene solo all'inizio e alla fine del ciclo.
- I requisiti dell'utente vengono scoperti solo alla fine.
- Se il prodotto non ha soddisfatto tutti i requisiti, alla fine del ciclo, è necessario iniziare daccapo tutto il processo.

3.4 Modello verification e validation (V&V)

Uguale al modello a cascata, vengono applicati i ricicli, ovvero al completamento di ogni fase viene fatta una verifica ed è possibile tornare alla fase precedente nel caso la stessa non verifica le aspettative.

3.5 Modello trasformatzionale

Basato su un modello matematico che viene trasformato da una rappresentazione formale ad un'altra. Questo modello comporta problemi nel personale in quanto non è facile trovare persone con le conoscenze giuste per poterlo implementare.

3.6 Modello di sviluppo basato sul riuso

E' previsto un repository dove vengono depositate le componenti sviluppate durante le fasi del ciclo di vita. Le componenti vengono prese dal repository e riutilizzate quando necessario.

Questo modello è particolarmente usato per sviluppare software in linguaggi Object Oriented.

3.7 Modello evolutivo (a prototipazione)

In questo modello il cliente è parte integrante del processo di sviluppo del prodotto. Il modello evolutivo si basa su due tipologie di sviluppo basate sui prototipi:

- **Prototipazione evolutiva**

Si inizia a sviluppare le parti del sistema che sono già ben specificate aggiungendo nuove caratteristiche secondo le necessità fornite dal cliente man mano.

- **Prototipo usa e getta (throw-away)**

Lo scopo di questo tipo di prototipazione è quello di identificare meglio le specifiche richieste dall'utente sviluppando dei prototipi che sono funzionanti. Non appena il prototipo è stato verificato da parte del cliente o da parte degli sviluppatori può essere buttato via.

3.8 Modello incrementale

Utilizzato per la progettazione di grandi software che richiedono tempi ristretti. Vengono rilasciate delle release funzionanti (*deliverables*) anche se non soddisfano pienamente i requisiti del cliente.

Vi sono due tipi di modelli incrementali:

- **Modello ad implementazione incrementale**

Le fasi alte del modello a cascata vengono realizzate e portate a termine, il software viene finito, testato e rilasciato ma non soddisfa tutte le aspettative richieste dall'utente. Le funzionalità non incluse vengono comunque implementate e aggiunte in tempi diversi. Con questo tipo di modello diventa fondamentale la parte di integrazione tra sottosistemi.

- **Modello a sviluppo e consegna incrementale**

E' un particolare modello a cascata in cui ad ogni fase viene applicato il modello ad implementazione incrementale e successivamente le singole fasi vengono sviluppate e integrate con il sistema esistente. Il sistema viene sviluppato seguendo le normali fasi e ad ogni fase l'output viene consegnato al cliente.

3.9 Modello a spirale

Il processo è visto come una spirale dove ogni ciclo viene diviso in quattro fasi:

- Determinazione degli *obiettivi* della fase
- Identificazione e riduzione dei *rischi*, valutazione delle alternative
- *Sviluppo e verifica* della fase
- *Pianificazione* della fase successiva

Una caratteristica importante di questo modello è il fatto che i rischi vengono presi seriamente in considerazione e che ogni fine ciclo produce una *deliverables*. In un certo senso può essere visto come un modello a cascata iterato più volte.

- **Vantaggi**

Rende esplicita la gestione dei rischi, focalizza l'attenzione sul riuso, determina errori in fasi iniziali, aiuta a considerare gli aspetti della qualità e integra sviluppo e manutenzione.

- **Svantaggi**

Richiede un aumento nei tempi di sviluppo, delle persone con capacità di identificare i rischi, una gestione maggiore del team di sviluppo e quindi anche un costo maggiore.

3.10 Modelli e valutazione dei rischi

Il *modello a cascata* genera alti rischi su un progetto mai sviluppato (greenfield engineering) e bassi rischi nello sviluppo di applicazioni familiari con tecnologie già note. Nel *modello a prototipazione* si hanno bassi rischi nelle nuove applicazioni, alti rischi per la mancanza di un processo definito e visibile. Nel *modello trasformatore* si hanno alti rischi a causa delle tecnologie coinvolte e delle professionalità richieste.

3.11 Scopo dell' ingegneria del software

- Migliorare la qualità del prodotto e del processo software
- Portabilità su sistemi legacy
- Eterogeneità
- Velocità di sviluppo

4 Project management

Il project management racchiude le attività necessarie per assicurare che un progetto software venga sviluppato rispettando le scadenze e gli standard.

Le entità fisiche che prendono parte al project management sono:

1. **Business manager:** definiscono i termini economici del progetto
2. **Project manager:** pianificano, motivano, organizzano e controllano lo sviluppo, stimano il costo del progetto, selezionano il team di sviluppo, stendono i rapporti e le presentazioni.
3. **Practitioners:** hanno competenze tecniche per realizzare il sistema
4. **Customers (clienti):** specificano i requisiti del software da sviluppare
5. **End users (utenti):** gli utenti che interagiscono con il sistema

4.1 Team di sviluppo

Esistono vari tipi di team di sviluppo qui di seguito indicati:

- **Democratico decentralizzato**
Assenza di un leader permanente (possono esistere dei leader a rotazione), consenso di gruppo, organizzazione orizzontale.
Vantaggi: individuazione degli errori, adatto a problemi difficili
Svantaggi: difficile da implementare, non è scalabile.
- **Controllato decentralizzato**
Vi è un leader che controlla il lavoro e assegna i problemi ai gruppi a lui sottesi. I sotto gruppi hanno un leader e sono composti di 2 a 5 persone. I leader dei sottogruppi possono comunicare tra loro come anche i membri dei sottogruppi possono comunicare in maniera orizzontale.
- **Controllato centralizzato**
Vi è un leader che decide sulle soluzioni e le organizzazioni dei gruppi. Ogni gruppo ha un proprio leader che assegna e controlla il lavoro dei componenti. I leader dei gruppi non comunicano tra loro ma possono comunicare solo con il loro capo. I membri dei gruppi non comunicano tra loro ma solo con il capo gruppo.

4.2 Stesura del piano del progetto

- **Introduzione**
Viene definita una descrizione di massima del progetto, gli elementi che vengono consegnati con le rispettive date di consegna e vengono pianificati eventuali cambiamenti.
- **Organizzazione del progetto**
Vengono definite le relazioni tra le varie fasi del progetto, la sua struttura organizzativa, le interazioni con entità esterne, le responsabilità di progetto (le principali funzioni e chi sono i responsabili).
- **Processi gestionali**
Si definiscono gli obiettivi e le priorità, le assunzioni, le dipendenze, i vincoli, i rischi con i relativi meccanismi di monitoraggio, pianificazione dello staff.
- **Processi tecnici**
Vanno specificati i sistemi di calcolo, i metodi di sviluppo, la struttura del team, il piano di documentazione del software e viene pianificata la gestione della qualità.
- **Pianificazione del lavoro, delle risorse umane e del budget**
Il progetto viene diviso in task (attività) e a ciascuno assegnata una priorità, le dipendenze, le risorse necessarie e i costi. Le attività devono essere organizzate in modo da produrre risultati valutabili dal management. I risultati possono essere *milestone* o *deliverables*; il primo rappresenta il punto finale di un'attività di processo, il secondo è un risultato fornito al cliente. Ogni *task* è un'unità atomica definita specificando: nome e descrizione del lavoro da svolgere, precondizioni per poter avviare il lavoro, risultato atteso, rischi. I vari task vanno organizzati in modo da ottimizzare la concorrenza e minimizzare la forza lavoro. Lo scopo è quello di

minimizzare la dipendenza tra le mansioni per evitare ritardi dovuti al completamento di altre attività.

Le attività del progetto vengono divise in task che sono caratterizzati dai tempi di inizio e fine, una descrizione, le precondizioni di partenza, i rischi possibili ed i risultati attesi.

4.3 Management dei rischi

Il management dei rischi identifica i rischi possibili e cerca di pianificare per minimizzare il loro effetto sul progetto, pianifica i rischi e li monitorizza.

4.3.1 Identificazione

I rischi da identificare sono di vari tipi tra cui: rischi tecnologici, rischi delle risorse umane, rischi organizzativi, rischi nei tools, rischi relativi ai requisiti, rischi di stima/sottostima.

Le tipologie di rischi sono le seguenti:

- **Tecnologici**
Alcune tecnologie di supporto (database, componenti esterne) non sono abbastanza validi come ci aspettavamo.
- **Risorse umane**
Non è possibile reclutare staff con la competenza richiesta oppure non è possibile fare formazione allo staff.
- **Organizzativi**
Cambi nella struttura organizzativa possono causare ritardi o nello sviluppo del progetto.
Requisiti
Cambiamenti nei requisiti richiedono una revisione del progetto già sviluppato.
- **Stima**
Il tempo richiesto, la dimensione del progetto sono stati sottostimati.

4.3.2 Analisi dei rischi

Ad ogni rischio va assegnata una probabilità che esso si verifichi e vanno valutati gli effetti dello stesso che possono essere: catastrofici, seri, tollerabili, insignificanti.

4.3.3 Pianificazione dei rischi

Viene considerato ciascun rischio e viene sviluppata una strategia per risolverlo. Le strategie che possiamo prendere possono essere:

- Evitare i rischi con una prevenzione
- Minimizzare i rischi
- Gestire i rischi con un piano di contingenza per evitarli

4.3.4 Monitoraggio dei rischi

Ogni rischio viene regolarmente valutato e viene verificato se è diventato meno o più probabile, inoltre i suoi aspetti vanno discussi con il management per valutare meglio i provvedimenti da adottare.

5 UML (Unified Modeling Language)

Il modello UML può essere visto gerarchicamente come un Sistema diviso in uno o più modelli a sua volta divisi in una o più viste. Lo scopo dei modelli UML è quello di semplificare l'astrazione di un progetto software e nascondere i dettagli non necessari alla comprensione della struttura generale.

Prima di visionare i vari tipi di diagrammi, UML definisce alcune convenzioni.

I nomi sottolineati delineano le istanze, mentre nomi non sottolineati denotano tipi (o classi), i diagrammi sono dei grafi, i nodi sono le entità e gli archi sono le interazioni tra di essi, gli attori rappresentano le entità esterne che interagiscono con il sistema.

Esistono vari tipi di diagrammi UML qui di seguito descritti:

5.1.1 Diagrammi dei casi d'uso (use case diagrams)

Serve a rappresentare l'interazione del sistema con uno o più attori e descrive tutti i vari casi possibili. Ha un nome univoco, degli attori partecipanti (almeno 1), una o più condizioni di entrata e di uscita, un flusso di eventi e delle eventuali condizioni eccezionali.

Tra le entità di un use case diagram si possono avere varie relazioni rappresentate dagli archi con la freccia rivolta verso l'entità.

Le relazioni possono essere di vario tipo: l'arco senza descrizione e senza freccia indica che l'attore può eseguire una certa funzionalità, l'arco con linea continua e con freccia indica una generalizzazione (ereditarietà) e punta verso il caso generale, l'arco con la descrizione <<extend>> rappresenta un caso eccezionale o che si verifica di rado e punta verso il caso eccezionale, quello con la descrizione <<include>> è la relazione tra due entità che indica che una determinata funzionalità implica l'esecuzione di un'altra e punta verso quella che viene eseguita per implicazione.

5.2 Diagrammi di classi (class diagrams)

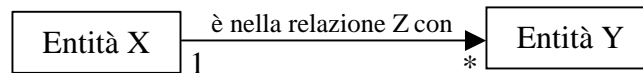
Rappresentano la struttura di un sistema. Vengono usati durante la fase di analisi dei requisiti e il system design di un modello.

È possibile definire diagrammi contenenti classi astratte e altri in cui compaiono istanze delle stesse con i relativi attributi specificati. Ogni nodo del grafo rappresenta una classe o un'istanza con un nome (nel caso di un'istanza il nome è sottolineato) e contiene i suoi attributi e i suoi comportamenti (le operazioni che essa svolge). Ogni attributo ha un tipo e ogni operazione ha una firma.

Le entità del grafo possono essere interconnesse tramite archi con freccia, senza freccia o tratteggiati ed avere le seguenti relazioni:

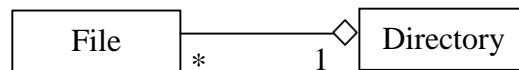
5.2.1 Associazione

L'arco semplice rappresenta una *associazione* che può essere specificata anche da un testo, inoltre è possibile indicare le molteplicità: ad esempio un'entità X può essere associata ad una o più entità Y in tal caso avremo il seguente diagramma:



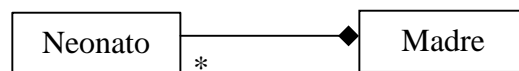
5.2.2 Aggregazione

E' possibile rappresentare chi contiene cosa sotto forma di grafo (o albero) utilizzando la linea con il rombo terminatore come nella figura.



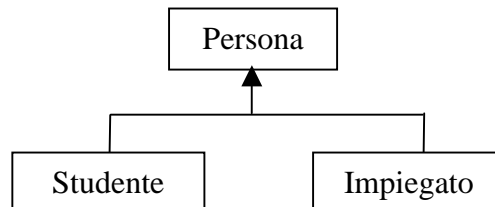
5.2.3 Composizione

Una particolare aggregazione che comporta l'esistenza di un'entità padre data un'entità figlio. In particolare:



5.2.4 Generalizzazione (o ereditarietà)

Indica l'ereditarietà tra entità: la classe figlio eredita attributi e operazioni del padre semplificando il modello ed eliminando la ridondanza.



5.3 Diagrammi sequenziali (sequence diagrams)

Descrivono le sequenze di azioni e le interazioni tra le componenti. Servono a dettagliare gli use case durante la fase di analisi dei requisiti oppure durante il system design per definire le interfacce del sottosistema e per trovare tutti gli oggetti partecipanti al sistema.

5.4 Diagramma a stati (state chart)

Descrivono una sequenza di stati di un oggetto in risposta a determinati eventi. Rappresentano anche le transizioni causate da un evento esterno e l'eventuale stato in cui esso viene portato.

5.5 Diagrammi delle attività (activity diagrams)

E' un particolare diagramma a stati in cui però al posto degli stati vi sono delle funzioni. Gli archi rappresentano la motivazione di esecuzione della funzione a cui punta.

5.6 Raggruppamento (packages)

Si può cercare migliorare la semplicità di un sistema raggruppando elementi del modello in packages. Ad esempio è possibile raggruppare use case o attività.

6 Raccolta dei requisiti (requirements elicitation)

L'ingegneria dei requisiti coinvolge due attività: *raccolta dei requisiti* e *analisi dei requisiti*.

La raccolta dei requisiti richiede la collaborazione tra più gruppi di partecipanti di tipologie e conoscenze diversificate. Gli errori commessi durante questa fase sono difficili da correggere e vengono spesso notati nella fase di consegna. Alcuni errori possono essere: funzionalità non specificate o incorrette o interfacce poco intuitive.

Utenti e sviluppatori devono collaborare per scrivere il **documento di specifica dei requisiti** che è scritto in linguaggio naturale per poi essere successivamente formalizzato e strutturato (in UML o altro) durante la fase di analisi per produrre il **modello di analisi**.

Il primo documento (la specifica dei requisiti) è utile al fine di favorire la comunicazione con il cliente e gli utenti, il documento prodotto nell'analisi è usato dagli sviluppatori.

La raccolta dei requisiti e l'analisi dei requisiti si focalizzano sul punto di vista dell'utente e definiscono i confini del sistema da sviluppare, in particolare vengono specificate:

- Funzionalità del sistema
- Interazione utente-sistema
- Errori che il sistema deve gestire
- Vincoli e condizioni di utilizzo

6.1 Classificazione dei requisiti

Le specifiche dei requisiti sono una sorta di contratto tra il cliente e gli sviluppatori e deve essere curata con attenzione in ogni suo dettaglio. Inoltre le parti del sistema che comportano un maggior rischio devono essere prototipate e provate con simulazioni per controllare la loro funzionalità e ed ottenere un riscontro dall'utente.

Esistono varie tipologie di requisiti qui di seguito specificati:

6.1.1 Requisiti funzionali

Descrivono le interazioni tra il sistema e l'ambiente esterno (utenti e sistemi esterni) indipendentemente dall'implementazione.

6.1.2 Requisiti non funzionali

Descrivono aspetti del sistema che non sono legati direttamente alle funzionalità del sistema. Ad esempio sono requisiti non funzionali dettagli implementativi tipo timeout e altro.

Altri requisiti non funzionali sono parte dello standard **FURPS** e sono di qualità e di vincoli.

- **Qualità**
 - Usabilità (help in linea, documentazione a livello utente)
 - Attendibilità (robustezza, coerenza delle funzionalità richieste)
 - Performance (tempo di risposta, throughput, disponibilità)
 - Supportabilità (manutenzione, portabilità, adattabilità)
- **Vincoli**
 - Implementazione (uso di tool, linguaggi, piattaforma hardware)
 - Interfacce (vincoli imposti da sistemi esterni tra cui sistemi legacy e formato di interscambio di dati)
 - Operativi (vincoli di management e amministrativi)
 - Packaging (riguardano i tools che sono richiesti all'utente al fine del funzionamento del software)
 - Legali (licenza, certificazione e regolamento)

6.2 Validazione dei requisiti

I requisiti devono essere continuamente validati con il cliente e l'utente, la validazione degli stessi è un aspetto molto importante perché ha lo scopo di non tralasciare nessun aspetto.

I requisiti devono rispettare le seguenti caratteristiche:

- Completezza (devono essere presi in considerazione tutti i possibili scenari, inclusi i comportamenti eccezionali)
- Consistenza (non devono contraddire se stessi)
- Non ambiguità (deve essere definito un unico sistema e non deve essere possibile interpretare la specifica in modi differenti)
- Correttezza (deve rappresentare il sistema di cui il cliente ha bisogno con accuratezza)
- Realistiche (se il sistema può essere implementato in tempi ragionevoli)
- Verificabili (se una volta che il sistema è stato implementato è possibile effettuare dei test)
- Tracciabili (se ogni requisito può essere mappato con una corrispondente funzionalità del sistema)

6.3 Greenfield engineering, re-engineering, interface engineering

Altri requisiti possono essere specificati in base alla sorgente delle informazioni. Il *greenfield engineering* avviene quando lo sviluppo di un'applicazione parte da zero, senza alcun sistema preesistente.

Il *re-engineering* è un tipo di raccolta dei requisiti dove c'è un sistema preesistente che deve essere riprogettato a causa di nuove esigenze o nuove tecnologie.

L'*interface engineering* avviene quando è necessario riprogettare un sistema per farlo lavorare in un nuovo ambiente. Un esempio possono essere i sistemi legacy che vengono lasciati inalterati nelle interfacce.

6.4 Attività della raccolta dei requisiti

1. Identificare gli attori
2. Identificare gli scenari
3. Identificare i casi d'uso
4. Raffinare i casi d'uso
5. Identificare le relazioni tra gli attori e i casi d'uso
6. Identificare gli oggetti partecipanti (verranno ripresi nella fase di analisi)
7. Identificare le richieste non funzionali

6.5 Identificare gli attori

Un attore è un'entità esterna che comunica con il sistema e può essere un utente, un sistema esterno o un ambiente fisico.

Ogni attore ha un nome univoco ed una breve descrizione sulle sue funzionalità (es. Teacher: una persona; Satellite GPS: fornisce le coordinate della posizione).

Un modo molto semplice per identificare gli attori di un sistema e porsi le seguenti domande:

- Quali gruppi di utenti sono supportati dal sistema per svolgere il proprio lavoro, quali eseguono le principali funzioni del sistema, quali eseguono le funzioni di amministrazione e mantenimento?
- Con quale sistema hardware o software il sistema interagisce?

6.6 Identificare gli scenari

Uno scenario è una descrizione informale, concreta e focalizzata di una singola caratteristica di un sistema e descrive cosa le persone fanno e sperimentano mentre provano ad usare i sistemi di elaborazione e le applicazioni.

Ogni scenario deve essere caratterizzato da un nome, una lista dei partecipanti e un flusso di eventi.

Esistono vari tipi di scenari:

- **As-is-scenario**
Sono usati per descrivere una situazione corrente. Vengono di solito usati nella raccolta dei requisiti di tipo re-engineering.
- **Visionary-Scenario**
Utilizzato per descrivere funzionalità future del sistema.
- **Evaluation-Scenario**
Descrivono funzioni eseguite dagli utenti rispetto alle quali poi viene testato il sistema.
- **Training-Scenario**
Sono tutorial per introdurre nuovi utenti al sistema.

L'identificazione degli scenari è una fase che avviene in stretta collaborazione con il cliente e l'utente.

Per poter formulare gli scenari bisogna porsi e porre all'utente le seguenti domande:

- Quali sono i compiti primari che l'attore vuole che svolga il sistema?
- Quali dati saranno creati/memorizzati/cambiati/cancellati o aggiunti dall'utente nel sistema?
- Di quali cambiamenti esterni l'attore deve informare il sistema?
- Di quali eventi/cambiamenti deve essere informato l'attore?

6.7 Identificare i casi d'uso

Un caso d'uso descrive una serie di interazioni che avvengono dopo un'inizializzazione da parte di un attore e specifica tutti i possibili scenari per una determinata funzionalità (visto in altri termini uno scenario è un'istanza di un caso d'uso).

Ogni caso d'uso contiene le seguenti informazioni:

- Un *nome* del caso d'uso che dovrebbe includere dei verbi
- I *nomi degli attori* partecipanti che dovrebbero essere sostantivi
- Le *condizioni di ingresso/uscita* da quel caso d'uso.
- Un *flusso di eventi* in linguaggio naturale
- Le *eccezioni* che possono verificarsi quando qualcosa va male descritte in modo distinto e separato
- I *requisiti speciali* che includono i requisiti non funzionali e i vincoli

6.8 Raffinare i casi d'uso

Vengono dettagliati gli elementi che sono manipolati dal sistema, dettagliate le interazioni a basso livello tra l'attore e il sistema, specificati i dettagli su chi può fare cosa, aggiunte eccezioni non presenti, le funzionalità comuni tra i casi d'uso vengono rese distinte.

6.9 Identificare le relazioni tra attori e casi d'uso

Esistono vari tipi di relazioni tra attori e casi d'uso (vedi anche lezione su UML):

- **Comunicazione**
Bisogna distinguere due tipi di relazione di comunicazione tra attori e casi d'uso. La prima detta <<initiate>> viene usata per indicare che un attore può iniziare un caso d'uso, la seconda <<participate>> invece indica che l'attore (che non ha iniziato il caso d'uso) può solo comunicare (es. ottenere informazioni) con lo stesso. In questo modo è possibile specificare già in questa fase dettagli sul controllo di accesso in quando vengono indicate le procedure e i passi per accedere a determinate funzioni del sistema.
- **Extend**
È usato per indicare un caso d'uso eccezionale in cui si viene a finire quando si sta eseguendo un altro caso d'uso. L'arco è tratteggiato con l'etichetta <<extend>> e con la linea rivolta verso il caso eccezionale.
- **Include**
Usata per scomporre un caso d'uso in dei casi d'uso più semplici. La freccia dell'arco è

tratteggiata, etichettata con <<include>> ed è rivolta verso il caso d'uso che viene usato di conseguenza da quelli che puntano.

6.10 Identificare gli oggetti partecipanti

Durante la fase di raccolta dei requisiti utenti e sviluppatori devono creare un glossario di termini usati nei casi d'uso. Si parte dalla terminologia che gli utenti hanno (quella del dominio dell'applicazione) e successivamente si negoziano cambiamenti. Il glossario creato è lo stesso che viene incluso nel manuale utente finale.

I termini possono rappresentare oggetti, procedure, sorgenti di dati, attori e casi d'uso. Ogni termine ha una piccola descrizione e deve avere un nome univoco e non ambiguo.

6.11 Identificare i requisiti non funzionali

6.12 5.4 Gestire la raccolta dei requisiti

Uno dei metodi per negoziare le specifiche con il cliente è il Joint Application Design (**JAD**), sviluppato da IBM, che si compone di cinque attività:

- *Definizione del progetto*: vengono interpellati il cliente e il project manager e vengono determinati gli obiettivi del progetto
- *Ricerca*: vengono interpellati utenti attuali e futuri e vengono raccolte informazioni sul dominio di applicazione e descritti ad alto livello i casi d'uso
- *Preparazione*: si prepara una sessione, un documento di lavoro che è un primo abbozzo del documento finale, un agenda della sessione e ogni altro documento cartaceo utile che rappresenta informazioni raccolte durante la ricerca
- *Sessione*: viene guidato il team nella creazione della specifica dei requisiti, lo stesso definisce e si accorda sugli scenari, i casi d'uso e interfaccia utente mock-up.
- *Documento finale*: viene rivisto il documento lavoro e messe insieme tutte le documentazioni raccolte; il documento rappresenta una completa specificazione del sistema accordato durante l'attività di sessione.

Un altro aspetto importante è la **tracciabilità** del sistema che include la conoscenza della sorgente della richiesta e gli aspetti del sistema e del progetto. Lo scopo della tracciabilità è quello di avere una visione chiara del progetto e rendere meno complessa e lunga un'eventuale fase di modifica ad un aspetto del sistema. A tale supporto è possibile creare dei collegamenti tra i documenti per meglio identificare le dipendenze tra le componenti del sistema.

Il documento dell'analisi dei requisiti (**RAD**) contiene la raccolta dei requisiti e l'analisi dei requisiti ed è il documento finale del progetto, serve come base contrattuale tra il cliente e gli sviluppatori.

7 Analisi dei requisiti

L'analisi dei requisiti è finalizzata a produrre un modello del sistema chiamato modello dell'analisi che deve essere corretto completo consistente e non ambiguo.

La differenza tra la raccolta dei requisiti e l'analisi è nel fatto che gli sviluppatori si occupano di strutturare e formalizzare i requisiti dati dall'utente e trovare gli errori commessi nella fase precedente (raccolta dei requisiti).

L'analisi, rendendo i requisiti più formali, obbliga gli sviluppatori a identificare e risolvere caratteristiche difficili del sistema già in questa fase, il che non avviene di solito.

Il modello dell'analisi è composto da tre modelli individuali:

- Il *modello funzionale* rappresentato da casi d'uso e scenari
- Il *modello ad oggetti* dell'analisi rappresentato da diagrammi di classi e diagrammi ad oggetti
- Il *modello dinamico* rappresentato da diagrammi a stati e sequence diagram

7.1 Concetti dell'analisi

7.1.1 Il modello ad oggetti

Il modello ad oggetti è una parte del modello dell'analisi basato e si focalizza sui concetti del sistema visti individualmente, le loro proprietà e le loro relazioni. Viene rappresentato con un diagramma a classi di UML includendo operazioni, attributi e classi.

7.1.2 Il modello dinamico

Il modello dinamico si focalizza sul comportamento del sistema utilizzando sequence diagram e diagrammi a stati. I sequence diagram rappresentano l'interazioni di un insieme di oggetti nell'ambito di un singolo caso d'uso. I diagrammi a stati rappresentano il comportamento di un singolo oggetto. Lo scopo del modello dinamico è quello di assegnare le responsabilità ad ogni singola classe, identificare nuove classi, nuove associazioni e nuovi attributi.

Nel modello ad oggetti dell'analisi e nel modello dinamico le classi che vengono descritte non sono quelle che in realtà poi verranno implementate nel software ma rappresentano ancora un punto di vista dell'utente. Spesso infatti ogni classe del modello viene mappata con una o più classi del codice sorgente, e gli attributi e tutte le sue caratteristiche sono specificate in modo minimale.

7.1.3 Entity, Boundary (oggetti frontiera) e Control object

Il modello ad oggetti è costituito da oggetti di tipo *entity*, *boundary* e *control*.

Gli oggetti *entity* rappresentano informazioni persistenti tracciate dal sistema; gli oggetti *boundary* rappresentano l'interazione tra attore e sistema; gli oggetti di *controllo* realizzano i casi d'uso.

Gli stereotipi *entity control* e *boundary* possono essere inclusi nei diagrammi e attaccati agli oggetti con le notazioni <<entity>> <<control>> <<boundary>>.

7.1.4 Attività dell'analisi (trasformare un caso d'uso in oggetti)

Le attività che andremo a descrivere sono le seguenti:

- Identificare gli oggetti *entity*
- Identificare gli oggetti *boundary*
- Identificare gli oggetti *control*
- Mappare i casi d'uso in oggetti con i sequence diagram
- Identificare le associazioni
- Identificare le aggregazioni
- Identificare gli attributi
- Modellare il comportamento e gli stati di ogni oggetto
- Rivedere il modello dell'analisi

7.1.5 Identificare gli oggetti entity

Per identificare gli oggetti partecipanti al modello dell'analisi bisogna prendere in considerazione quelli identificati durante la specifica di requisiti.

Visto che il documento della specifica dei requisiti è scritto in linguaggio naturale, è frequente riscontrare imprecisioni nel testo, oppure dei sinonimi sulla notazioni che possono indurre gli sviluppatori a considerare male gli oggetti.

7.1.6 Identificare gli oggetti boundary

Gli oggetti boundary rappresentano l'interfaccia del sistema con l'attore. Ogni attore dovrebbe interagire con almeno un oggetto boundary.

Gli oggetti boundary raccolgono informazioni dall'attore e le traducono in un formato che può essere usato dagli oggetti entity e control.

Essi non descrivono in dettaglio gli aspetti visuali dell'interfaccia utente: ad esempio specificare scroll-bar o menu-item può essere troppo dettagliato.

Per scovare gli oggetti boundary è possibile anche in questo caso usare un'euristica:

- Identificare il controllo dell'interfaccia utente di cui l'utente ha bisogno per iniziare un caso d'uso
- Identificare i moduli di cui gli utenti hanno bisogno per inserire dati nel sistema
- Identificare messaggi e notifiche che il sistema deve fornire all'utente
- Non modellare aspetti visuali dell'interfaccia con oggetti boundary
- Usare sempre il termine utente finale per descrivere le interfacce.

7.1.7 Identificare gli oggetti controllo

Gli oggetti Control sono responsabili del coordinamento tra gli oggetti entity e boundary e lo scopo è quello di prendere informazioni dagli oggetti boundary e inviarli agli oggetti entità. Un oggetto control viene creato all'inizio di un caso d'uso e termina alla fine di questo. Anche questo come gli oggetti entità e boundary si basa su delle euristiche:

- Identificare un oggetto control per ogni caso d'uso

- Identificare un oggetto control per ogni attore nel caso d'uso
- La vita di un oggetto control deve corrispondere alla durata di un caso d'uso o di una sessione utente.

7.1.8 Mappare casi d'uso in oggetti con sequence diagrams

Mappare i casi d'uso in sequence diagram serve per mostrare il comportamento tra gli oggetti partecipanti e ne mostrano l'interazione.

I sequence diagram non sono comprensibili all'utente ma sono uno strumento più preciso di supporto agli sviluppatori.

In un sequence diagram:

- Le colonne rappresentano gli oggetti che partecipano al caso d'uso
- La prima colonna rappresenta l'attore che inizia il caso d'uso
- La seconda colonna è l'oggetto boundary con cui l'attore interagisce per iniziare il caso d'uso
- La terza colonna è l'oggetto control che gestisce il resto del caso d'uso
- Gli oggetti control creano altri oggetti boundary e possono interagire con altri oggetti Control
- Le frecce orizzontali tra le colonne rappresentano messaggi o stimoli inviati da un oggetto ad un altro
- La ricezione di un messaggio determina l'attivazione di un'operazione
- L'attivazione è rappresentata da un rettangolo da cui altri messaggi possono prendere origine
- La lunghezza del rettangolo rappresenta il tempo durante il quale l'operazione è attiva
- Il tempo procede verticalmente dall'alto al basso
- Al top del diagramma si trovano gli oggetti che esistono prima del 1° messaggio inviato
- Oggetti creati durante l'interazione sono illustrati con il messaggio <<create>>
- Oggetti distrutti durante l'interazione sono evidenziati con una croce
- La linea tratteggiata indica il tempo in cui l'oggetto può ricevere messaggi

Mediante i sequence diagram è possibile trovare comportamenti o oggetti mancanti. Nel caso manchi qualche entità è necessario ritornare ai casi d'uso, ridefinire le parti mancanti e tornare a questa fase per ricreare il sequence diagram.

7.1.9 Identificare le associazioni

Un'associazione è una relazione tra due o più oggetti/classi. Ogni associazione ha un nome, un ruolo ad ogni capo dell'arco che identifica la funzione di ogni classe rispetto all'associazione e una molteplicità che indica ad ogni capo il numero di istanze possibili (vedi UML per dettagli).

Un'utile euristica per trovare le associazioni è la seguente:

- Esaminare i verbi nelle frasi
- Nominare in modo preciso i nomi delle associazioni e i ruoli
- Eliminare associazioni che possono essere derivate da altre associazioni
- Troppe associazioni rendono il modello degli oggetti “illeggibile”

7.1.10 Identificare le aggregazioni

Identifica che un oggetto è parte di un altro oggetto o lo contiene (vedi UML per dettagli).

7.1.11 Identificare gli attributi

Gli attributi sono proprietà individuali degli oggetti/classi. Ogni attributo ha un nome, una breve descrizione e un tipo che ne descrive i possibili valori.

L’euristica di Abbott indica che gli attributi possono essere identificati nel linguaggio naturale del documento delle specifiche prendendo in considerazione gli aggettivi.

7.1.12 Modellare il comportamento e gli stati di ogni oggetto

Gli oggetti che hanno un ciclo di vita più lungo dovrebbero essere descritti in base agli stati che essi possono assumere. Per fare ciò vengono usati i diagrammi a stati.

7.1.13 Rivedere il modello dell’analisi

Una volta che il modello dell’analisi non subisce più modifiche o ne subisce raramente è possibile passare alla fase di revisione del modello. La revisione del modello deve essere fatta prima dagli sviluppatori e poi insieme dagli sviluppatori e gli utenti.

L’obiettivo di questa attività di revisione è stabilire che la specifica risulta essere: corretta, completa, consistente e chiara,

Domande da porsi per assicurarsi della correttezza:

- Il glossario è comprensibile per gli utenti?
- Le classi astratte corrispondono a concetti ad alto livello?
- Tutte le descrizioni concordano con le definizioni degli utenti?
- Oggetti Entity e Boundary hanno nomi significativi?
- Oggetti control e use case sono nominati con verbi significativi del dominio?
- Tutti gli errori/eccezioni sono descritti e trattati?

Domande da porsi per assicurarsi della completezza:

- Per ogni oggetto: è necessario per uno use case? In quale use case è creato? modificato? distrutto? Può essere acceduto da un oggetto boundary?
- Per ogni attributo: quando è settato? Quale è il tipo?
- Per ogni associazione: quando è attraversata? Perché ha una data molteplicità?
- Per ogni oggetto control: ha le associazioni necessarie per accedere agli oggetti che partecipano nel corrispondente use case?

Domande da porsi per assicurarsi della consistenza:

- Ci sono classi o use case con lo stesso nome?
- Ci sono entità con nomi simili e che denotano concetti simili?

Domande da porsi per assicurarsi della chiarezza:

- Le richieste di performance specificate sono state assicurate?
- Può essere costruito un prototipo per assicurarsi della fattibilità?

8 System design

8.1 Scopi criteri e architetture

Gli scopi del system design sono quelli di definire gli obiettivi di progettazione del sistema, decomporre il sistema in sottosistemi più piccoli in modo da poterli assegnare a team individuali e selezionare alcune strategie quali:

- Scelte hardware e software
- Gestione dei dati persistenti
- Il flusso di controllo globale
- Le politiche di controllo degli accessi
- La gestione delle condizioni boundary (startup, shutdown, eccezioni)

Il system design si focalizza sul dominio di implementazione, prende in input il modello di analisi e dopo averlo trasformato da in output un modello del sistema.

Le attività del system design globalmente possono essere divise in tre fasi:

8.2 Identificare gli obiettivi di design

In questa fase gli sviluppatori definiscono le priorità delle qualità del sistema. Molti obiettivi possono essere ricavati utilizzando requisiti non funzionali o dal dominio dell'applicazione, altri vengono forniti direttamente dal cliente. Per ottenere gli obiettivi finali vanno seguiti dei criteri di progettazione tenendo presente performance, affidabilità, costi, mantenimento e utente finale.

- **Criteri di performance**

In questi criteri vengono inclusi: tempo di risposta, throughput (quantità di task eseguibili in un determinato periodo di tempo) e memoria.

- **Criteri di affidabilità**

L'affidabilità include criteri di robustezza (capacità di gestire condizioni non previste), attendibilità (non ci deve essere differenza tra il comportamento atteso e quello osservato), disponibilità (tempo in cui il sistema è disponibile per l'utilizzo), tolleranza ai fault (capacità di operare in condizioni di errore), sicurezza, fidatezza (capacità di non danneggiare vite umane).

- **Criteri di costi**

Vanno valutati i costi di sviluppo del sistema, alla sua installazione e al training degli utenti, eventuali costi per convertire i dati del sistema precedente, costi di manutenzione e costi di amministrazione.

- **Criteri di mantenimento**

Tra i criteri di mantenimento troviamo: estendibilità, modificabilità, adattabilità, portabilità, leggibilità e tracciabilità dei requisiti.

- **Criteri di utente finale**

In criteri dell'utente finale da tenere in considerazione sono quelli di utilità (quando bene il sistema dovrà facilitare e supportare il lavoro dell'utente) e quelli di usabilità.

Spesso quando si progetta un sistema non è possibile rispettare tutti i criteri di qualità contemporaneamente, viene quindi utilizzata una strategia di trade-off (compromesso) e data una priorità maggiore ad alcuni criteri tenendo presente scelte manageriali quali scheduling e budget.

8.3 Decomposizione del sistema in sottosistemi

Utilizzando come base i casi d'uso e l'analisi e seguendo una particolare *architettura software* (MVC, Client-Server ecc.) il sistema viene decomposto in sottosistemi.

Lo scopo è quello di poter assegnare a un singolo sviluppatore o ad un team parti software semplici. In questa fase viene descritto come i sottosistemi sono collegati alle classi.

Un sottosistema è caratterizzato dai servizi (insieme di operazioni) che esso offre agli altri sottosistemi. L'insieme di servizi che un sistema espone viene chiamato interfaccia (API: *application programming interface*) che include, per ogni operazione: i parametri, il tipo e i valori di ritorno. Le operazioni che essi svolgono vengono descritte ad alto livello senza entrare troppo nello specifico.

Il sistema va diviso in sottosistemi tenendo presente queste due proprietà:

8.3.1 Accoppiamento (coupling)

Misura quanto un sistema è dipendente da un altro.

Due sistemi si dicono **loosely coupled** (leggermente accoppiati) se una modifica in un sottosistema avrà poco impatto nell'altro sistema, mentre si dicono **strongly coupled** (fortemente accoppiati) se una modifica su uno dei sottosistemi avrà un forte impatto sull'altro.

La condizione ideale di accoppiamento è quella di tipo loosely in quanto richiede meno sforzo quando devono essere modificate delle componenti.

Se ad esempio, tre componenti usano lo stesso servizio esposto da una componente che potrebbe essere modificata spesso, conviene frapporre tra di esse una nuova componente che ci permette di evitare una modifica alle tre componenti che usufruiscono del servizio.

Ovviamente ove non sono presenti componenti che si pensa debbano essere modificate spesso, non conviene utilizzare questa strategia in quanto aggiungerebbe complessità di sviluppo e di calcolo al sistema.

8.3.2 Coesione

Misura la dipendenza tra le classi contenute in un sottosistema.

La coesione è alta se due componenti di un sottosistema realizzano compiti simili o sono collegate l'una con l'altra attraverso associazioni (es. ereditarietà), è invece bassa nel caso contrario.

L'ideale sarebbe quello di avere sottosistemi con coesione interna alta.

La decomposizione del sistema avviene utilizzando layer e/o partizioni.

8.3.3 Divisione del sistema con i layer

Con la decomposizione in layer il sistema viene visto come una gerarchia di sottosistemi. Un layer è un raggruppamento di sottosistemi che forniscono servizi correlati. I layer per implementare un servizio potrebbero usare a sua volta servizi offerti dai layer sottostanti ma non possono usare servizi dei livelli più alti.

Con i layer si possono avere due tipi di **architettura: chiusa e aperta**.

Con l'architettura chiusa un layer può accedere solo alle funzionalità del layer immediatamente a lui sottostante, con quella aperta il layer può accedere alle funzionalità del layer sottostante e di tutti gli altri sotto di esso.

Nel primo caso si ottiene un'alta manutentibilità e portabilità, nel secondo una maggiore efficienza in quanto si risparmia l'overhead delle chiamate in cascata.

8.3.4 Divisione del sistema con le partizioni

Il sistema viene diviso in sottosistemi paritari (peer), ognuno dei quali è responsabile di diverse classi di servizi.

In generale una decomposizione di un sistema avviene utilizzando ambedue le tecniche. Infatti il sistema viene prima diviso in sottosistemi tramite le partizioni e successivamente ogni partizione viene organizzata in layer finché i sottosistemi non sono abbastanza semplici da essere sviluppati da un singolo sviluppatore o team.

8.4 Architetture software

Un architettura software include scelte relative alla decomposizione in sottosistemi, flusso di controllo globale, gestione delle condizioni limite e i protocolli di comunicazione tra i sottosistemi. E' da notare che la decomposizione dei sottosistemi è una fase molto critica in quanto una volta iniziato lo sviluppo con una determinata decomposizione è complesso ed oneroso dover tornare indietro in quanto molte interfacce dei sottosistemi dovrebbero essere modificate.

Alcuni stili architetturali che potrebbero essere usati sono:

8.4.1 Repository

Con questo stile tutti i sottosistemi accedono e modificano i dati tramite un oggetto repository. Il flusso di controllo viene dettato dal repository tramite un cambiamento dei dati oppure dai sottosistemi tramite meccanismi di sincronizzazione o lock.

I vantaggi di questo stile si vedono quando si implementano applicazioni in cui i dati cambiano di frequente poiché si evitano incoerenze. Considerando i problemi che questo stile può dare sicuramente si può notare che il repository può facilmente diventare un collo di bottiglia in termini di prestazioni e inoltre il coupling tra i sottosistemi e il repository è altissimo: una modifica all'API del repository comporta la modifica di tutti i sottosistemi che lo utilizzano.

8.4.2 Model/View/Control (MVC)

Il sistema viene diviso in tre sottosistemi: Model, View e Control. Il sottosistema model implementa la struttura dati centrale, il controller gestisce il flusso di controllo (si occupa di prendere l'input dall'utente e di inviarlo al model), il view è la parte di interazione con l'utente.

Uno dei vantaggi di MVC si vede quando le interfacce utente (view) vengono modificate più di frequente rispetto alla conoscenza del dominio dell'applicazione (model). Per questo motivo MVC è l'ideale per sistemi interattivi e quando il sistema deve avere viste multiple.

8.4.3 Client-Server

Il sottosistema server fornisce servizi ad una serie di istanze di altri sottosistemi detti client i quali si occupano dell'interazione con l'utente. La maggior parte della computazione viene svolta a lato server. Questo stile è spesso usato in sistemi basati su database in quanto è più facile gestire l'integrità e la consistenza dei dati.

8.4.4 Peer-To-Peer

E' una generalizzazione dello stile client-server in cui però client e server possono essere scambiati di ruolo ed ognuno dei due può fornire servizi.

8.4.5 Three-Tier

I sottosistemi vengono organizzati in tre livelli hardware: *interface*, *application* e *storage*. Il primo conterrà tutti gli oggetti boundary di interazione con l'utente, il secondo include gli oggetti relativi al controllo e alle entità, il terzo effettua l'interrogazione e la ricerca di dati persistenti.

8.4.6 Considerazioni finali

Quando si decidono le componenti di un sottosistema bisognerebbe tenere presente che la maggior parte dell'interazione tra le componenti dovrebbe avvenire all'interno di un sottosistema allo scopo di ottenere un'alta coesione.

8.4.7 Euristiche per scegliere le componenti

Le euristiche per scegliere le componenti dei sottosistemi sono le seguenti:

- Gli oggetti identificati in un caso d'uso dovrebbero appartenere ad uno stesso sottosistema.
- Bisogna creare dei sottosistemi che si occupano di trasferire i dati tra i sottosistemi
- Minimizzare il numero di associazioni tra i sottosistemi (devono essere loosely coupled)

- Tutti gli oggetti di un sottosistema dovrebbero essere funzionalmente correlati

8.4.8 Descrizione delle attività del System Design

Le attività del system design sono le seguenti:

- Mappare i sottosistemi su piattaforme e processori
- Identificare e memorizzare informazioni persistenti
- Stabilire i controlli di accesso
- Progettare il flusso di controllo globale
- Identificare le condizioni limite
- Rivedere il modello del system design

8.4.9 Mappare i sottosistemi su piattaforme e processori

Molti sistemi complessi necessitano di lavorare su più di un computer interconnessi da rete. L'uso di più computer può ottimizzare le performance e permettere l'utilizzo del sistema a più utenti distribuiti sulla rete.

In questa fase vanno prese alcune decisioni per quanto riguarda le piattaforme hardware e software su cui il sistema dovrà girare (es Unix vs Windows, Intel vs Sparc etc).

Una volta decise le piattaforme è necessario mappare le componenti su di esse. Questa operazione potrebbe portare all'introduzione di nuove componenti per interfacciare i sottosistemi su diverse piattaforme (es. una libreria per il collegamento ad un DBMS).

Sfortunatamente, da una parte, l'introduzione di nuovi nodi hardware distribuisce la computazione, dall'altro introduce alcune problematiche tra cui la sincronizzazione, la memorizzazione, il trasferimento e la replicazione di informazioni tra sottosistemi.

8.4.10 Identificare e memorizzare i dati persistenti

Il modo in cui i dati vengono memorizzati può influenzare l'architettura del sistema (vedi lo stile architetturale repository) e la scelta di uno specifico database. In questa fase vanno identificati gli oggetti persistenti e scelto il tipo di infrastruttura da usare per memorizzarli (dbms, file o altro).

Gli oggetti entity identificati durante l'analisi dei requisiti sono dei buoni candidati a diventare persistenti. Non è detto però che tutti gli oggetti entità debbano diventare persistenti. In generale i dati sono persistenti se sopravvivono ad una singola esecuzione del sistema. Il sistema dovrà memorizzare i dati persistenti quando questi non servono più e ricaricarli quando necessario.

Una volta decisi gli oggetti dobbiamo decidere come questi oggetti devono essere memorizzati. Principalmente potremmo avere a disposizione tre mezzi: file, dbms relazionale e dbms ad oggetti.

8.4.11 File

La prima tipologia da una parte richiede una logica più complessa per la scrittura e lettura, dall'altra permette un accesso ai dati più efficiente.

8.5 DBMS relazionale

Un DBMS relazionale fornisce un'interfaccia di più alto livello rispetto al file. I dati vengono memorizzati in tabelle ed è possibile utilizzare un linguaggio standard per le operazioni (SQL). Gli oggetti devono essere mappati sulle tabelle per poter essere memorizzati.

8.5.1.1 DBMS ad oggetti

Un database orientato ad oggetti è simile ad un DBMS relazionale con la differenza che non è necessario mappare gli oggetti in tabelle in quanto questi vengono memorizzati così come sono. Questo tipo di database riduce il tempo di setup iniziale (si risparmia sulle decisioni di mapping) ma sono più lenti e le query sono di più difficile comprensione.

8.5.1.2 Considerazioni e trade-offs

La scelta tra una tecnologia o un'altra per la memorizzazione può essere influenzata da vari fattori. In particolare conviene usare un file in questi casi:

- Dimensione elevata dei dati (es. immagini, video ecc.)
- Dati temporanei e logging

Conviene invece usare un DBMS (relazionale e ad oggetti) in casi di:

- Accessi concorrenti (i DBMS effettuano controlli di consistenza e concorrenza bloccando i dati quando necessario)
- Uso dei dati da parte di più piattaforme
- Particolari politiche di accesso a dati

8.6 Stabilire i controlli di accesso

In un sistema multi utenza è necessario fornire delle politiche di accesso alle informazioni. Nell'analisi sono stati associati casi d'uso ad attori, in questa fase vanno definite in modo più preciso le operazioni e le informazioni effettuabili da ogni singolo attore e come questi si autenticano al sistema. E' possibile rappresentare queste politiche tramite una matrice in tre modi:

- **Tabella di accesso globale**
Ogni riga della matrice contiene una tripla (attore, classe, operazione). Se la tupla è presente per una determinata classe e operazioni l'accesso è consentito altrimenti no.
- **Access control list (ACL)**
Ogni classe ha una lista che contiene una tupla (attore, operazione) che specifica se l'attore può accedere a quella determinata operazione della classe a cui la ACL appartiene.
- **Capability**
Una capability è associata ad un attore ed ogni riga della matrice contiene una tupla (classe, operazione) che l'attore a cui è associata può eseguire.

Scegliere una o l'altra soluzione impatta sulle performance del sistema. Ad esempio scegliere una tabella di accesso globale potrebbe far consumare molta memoria. Le altre vanno usate in base al tipo di controllo che vogliamo effettuare: se vogliamo rispondere più velocemente alla domanda "chi può accedere a questa classe?" useremo una ACL, se invece vogliamo rispondere più velocemente alla domanda "a quale operazione può accedere questo attore?" useremo una capability.

8.6.1 Progettare il flusso di controllo globale

Un flusso di controllo è una sequenza di azioni di un sistema. In un sistema Object Oriented una sequenza di azioni include prendere decisioni su quali operazioni eseguire ed in che ordine. Queste decisioni sono basate su eventi esterni generati da attori o causati dal trascorrere del tempo.

Esistono tre tipi di controlli di flusso:

8.6.1.1 Procedure-driven control

Le operazioni rimangono in attesa di un input dell'utente ogni volta che hanno bisogno di elaborare dati. Questo tipo di controllo di flusso è particolarmente usato in sistemi legacy di tipo procedurale.

8.6.1.2 Event-driven control

In questo controllo di flusso un ciclo principale aspetta il verificarsi di un evento esterno. Non appena l'evento diventa disponibile la richiesta viene direzionata all'opportuno oggetto. Questo tipo di controllo ha il vantaggio di centralizzare tutti gli input in un ciclo principale ma ha lo svantaggio di rendere complessa l'implementazione di sequenze di operazioni composte di più passi.

8.6.1.3 Threads

Questo controllo di flusso è una modifica del procedure-driven control che aggiunge la gestione della concorrenza. Il sistema può creare un arbitrario numero di threads (processi leggeri), ognuno assegnato ad un determinato evento.

Se si sceglie di usare un control-flow di tipo threads bisogna stare attenti a gestire situazioni di concorrenza in quando più thread possono accedere contemporaneamente alle stesse risorse e creare situazioni non previste.

8.7 Identificare le condizioni limite

Le condizioni limite del sistema includono lo **startup**, lo **shutdown**, l'inizializzazione e le gestione di fallimenti come corruzione di dati, caduta di connessione e caduta di componenti.

A tale scopo vanno elaborati dei casi d'uso che specificano la sequenza di operazioni in ciascuno dei casi sopra elencati.

In generale *per ogni oggetto persistente*, si esamina in quale caso d'uso viene creato e distrutto. Se l'oggetto non viene creato o distrutto in nessun caso d'uso deve essere aggiunto un caso d'uso invocato dall'amministratore.

Per ogni componente vanno aggiunti tre casi d'uso per l'avvio, lo shutdown e per la configurazione. *Per ogni tipologia di fallimento* delle componenti bisogna specificare come il sistema si accorge di tale situazione, come reagisce e quali sono le conseguenze.

Un'eccezione è un evento o errore che si verifica durante l'esecuzione del sistema. Una situazione del genere può verificarsi in tre casi:

- Un fallimento hardware (dovuto all'invecchiamento dell'hardware)
- Un cambiamento nell'ambiente (interruzione di corrente)
- Un fallimento del software (causato da un errore di progettazione)

Nel caso in cui un errore dipenda da un input errato dell'utente, tale situazione deve essere comunicata all'utente tramite un messaggio così che lo stesso possa correggere l'input e riprovare.

Nel caso di caduta di un collegamento il sistema dovrebbe salvare lo stato del sistema in modo da poter riprendere l'esecuzione non appena il collegamento ritorna.

8.7.1 Rivedere il modello del system design

Un progetto di sistema deve raggiungere degli obiettivi e bisogna assicurarsi che rispetti i seguenti criteri:

- **Correttezza**
Il system design è corretto se il modello di analisi può essere mappato su di esso.
- **Completezza**
La progettazione di un sistema è completa se ogni requisito e ogni caratteristica è stata portata a compimento.
- **Consistenza**
Il system design è consistente se non contiene contraddizioni.

- **Realismo**

Un progetto è realistico se il sistema può essere realizzato ed è possibile rispettare problemi di concorrenza e tecnologie.

- **Leggibilità**

Un system design è leggibile se anche sviluppatori non coinvolti nella progettazione possono comprendere il modello realizzato.

8.7.2 Gestione del system design

La gestione del system design coinvolge le seguenti attività:

- Documentazione del System Design (SDD)
- Assegnazione delle responsabilità
- Iterazione delle attività

La suddivisione in sottosistemi effettuata nel system design influenza le decisioni sulla quantità di personale necessario per portare a termine il progetto e su come dividere i team di sviluppo.

Un team particolare, l'*architecture team* si occupa di suddividere il sistema in sottosistemi e di assegnare le responsabilità ai singoli team o ai singoli sviluppatori.

Un'altra figura importante è quella dell'*architetto* che deve assicurare la consistenza delle decisioni e dello stile delle interfacce.

La stesura del system design è effettuata in modo iterativo. Infatti alla fine di ogni stesura è possibile effettuare delle modifiche al modello relativamente a:

- Decomposizione in sottosistemi
- Interfacce
- Condizioni eccezionali

9 Database

9.1 Introduzione

I database o banche dati o base dati sono collezioni di dati, tra loro correlati, utilizzati per rappresentare una porzione del mondo reale. Sono strutturati in modo tale da consentire la gestione dei dati stessi in termini di inserimento, aggiornamento, ricerca e cancellazione delle informazioni.

9.2 Architetture utilizzate

8.5.1 XAMPP e DBMS

E' una semplice e leggera distribuzione **Apache** che rende estremamente semplice a sviluppatori e neofiti creare web server per scopi di test. Tutto ciò di cui si ha bisogno: un server web application (**Apache**), un DBMS (**MySQL**), e un linguaggio di script (**PHP**) sono inclusi in un unico, comodo pacchetto che è stato concepito per un'installazione e un utilizzo intuitivi.

XAMPP è anche **multi piattaforma** (cross-platform). Ciò significa che funziona su ambienti Linux, Mac e Windows. Sul sito ApacheFriends.org ne potete trovare i rispettivi pacchetti.

XAMPP è **open source**. E' una raccolta di software gratuiti (paragonabile a una distribuzione Linux), è totalmente gratuita e la riproduzione è libera.

Inoltre, dato che molti ambienti server utilizzano gli stessi componenti, XAMPP è utile in quanto rende semplice e intuitivo il passaggio da un sistema locale di test a un server web vero e proprio.

I PRINCIPALI COMPONENTI DI XAMPP:

Apache: è il web server application che elabora e risponde le richieste restituendo i contenuti verso il computer richiedente (client). Apache è il web server più popolare al mondo.

MySQL: Ogni applicazione web, si appoggia a un database per memorizzare i dati. MySQL, il più popolare tra i DBMS, è Open Source e gratuito. Usato per piattaforme come Joomla e WordPress, consente la realizzazione di applicazioni professionali.

PHP: acronimo per Hypertext Preprocessor. È un linguaggio di script lato server che è utilizzato in tutto il mondo per realizzare siti di ogni genere. PHP, è il linguaggio con il quale sono scritti Joomla, Drupal e WordPress. Anch'esso è Open Source.

Relativamente semplice da imparare, lavora perfettamente con MySQL, scelta fatta da migliaia di sviluppatori.

Perl: è un linguaggio di programmazione molto potente, ricco di funzionalità, con oltre 27 anni di sviluppo.

XAMPP può contenere inoltre componenti aggiuntivi come **phpMyAdmin**, **OpenSSL**, etc.

In questo caso è stato utilizzato MySQL e il programma con cui è stato amministrato è phpMyAdmin.

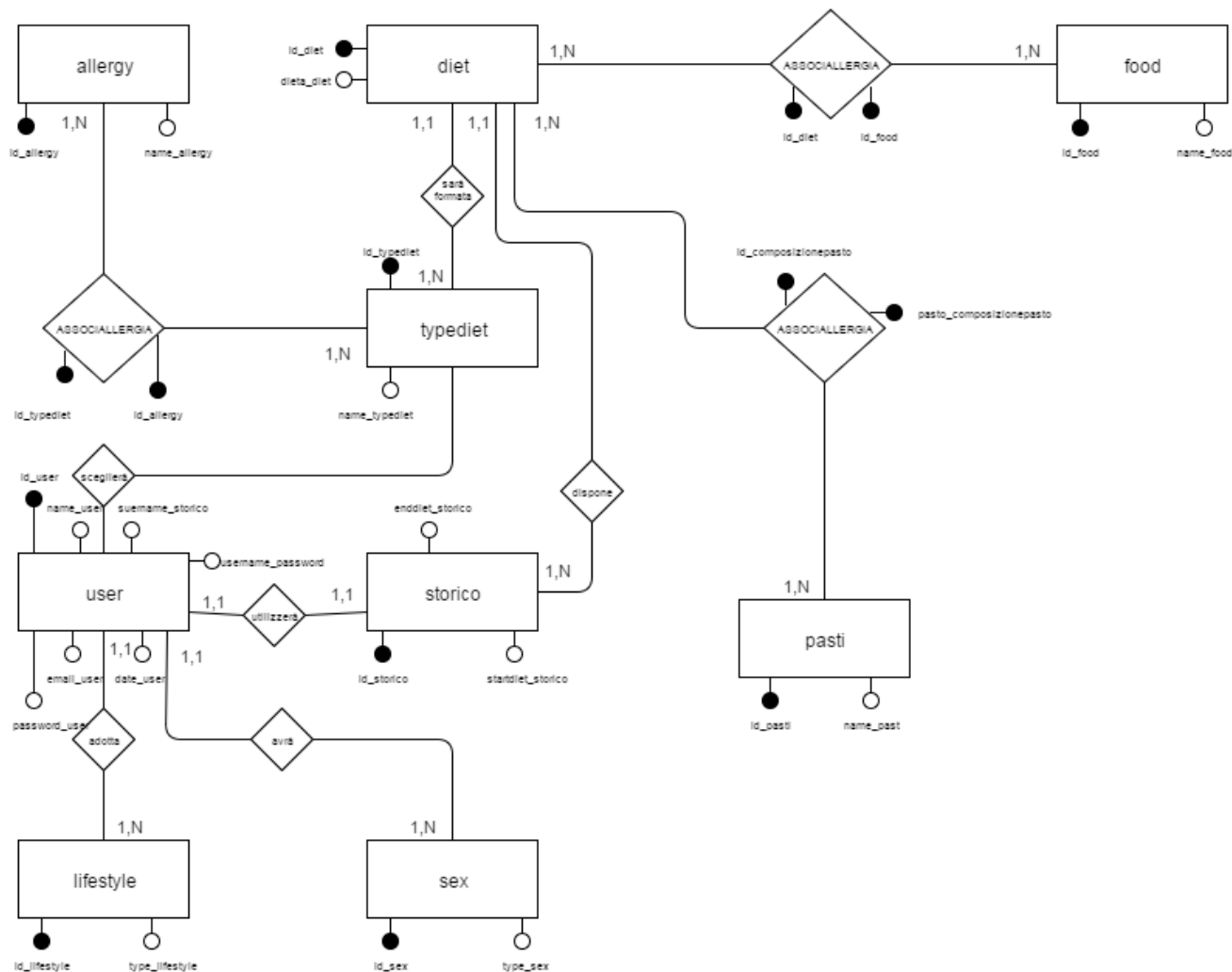
9.3 SQL

SQL (pronunciato SEQUEL) è un acronimo che sta **Structured Query Language** e indica un linguaggio di programmazione per database. Lo SQL è un linguaggio utilizzato per creare, trasformare e recuperare informazioni in un **RDBMS** (*Relational Database Management System*, sistema per la gestione dei database relazionali) e venne progettato e realizzato ad inizio anni '70 nei laboratori IBM. Gestito attraverso applicazioni come SQL Server Express, questo linguaggio di programmazione permette di creare e modificare schemi di database; inserire, modificare e gestire i dati memorizzati; interrogare i dati memorizzati e creare e gestire strumenti di controllo e accesso a questi stessi dati. Insomma, a differenza di quanto potrebbe far credere il nome, lo SQL non è solamente un query language, ma ha anche funzioni di gestione e controllo di database tipiche di altri linguaggi di programmazione. Lo SQL si compone di alcune parti fondamentali: la prima è la Data Definition Language e permette di creare o cancellare database o modificarne la struttura; la seconda è la Data Manipulation Language e permette di inserire, cancellare e modificare dati

9.4 Glossario dei termini

Termini	Descrizione	Collegamenti
Diet	Indica la dieta che è stata calcolata per una determinata persona	Composizionedecibo, composizionedepasto, Typediet,
Composizionedecibo	Collegamento tra dieta e cibo	Diet, Food
Food	Elenco di tutti i cibi	Composizionedecibo
Composizionedepasto	Collegamento tra dieta e pasto	Diet, Pasti
Pasti	Elenco pasti giornalieri	Composizionedepasto
Typediet	Indica il tipo di dieta che si vuole scegliere	Diet, Associallergia, User
Associallergia	Collega l'allergia al tipo di dieta	Typediet, Allergy
Allergy	Elenco tipi di allergie	Associallergia
Storico	Record delle diete calcolate	Diet, User
Lifestyle	Stile di vita prima dell'inizio della dieta	User
Sex	Sesso di una persona	User
User	Utenti registrati	Storico, Sex, Lifestyle, Typediet

9.5 Modello concettuale



Uno strumento utilizzato per costruire un modello concettuale dei dati indipendente dalle applicazioni è il modello entità/associazioni (Entity/Relationship). Lo schema E/R è una rappresentazione grafica che permette di individuare gli elementi del modello concettuale e le associazioni tra essi. Gli elementi di un modello E/R sono:

- entità, oggetti che hanno un significato anche quando vengono considerati in modo isolato e sono di interesse per la realtà che si vuole modellare
- associazioni, legami che si stabiliscono tra entità
- attributi, proprietà di un'entità o di un'associazione. Un'associazione può essere di tre tipi:
 - uno a uno: ad ogni elemento del primo insieme corrisponde uno ed un solo elemento del secondo insieme

- uno a molti: ad ogni elemento del primo insieme possono corrispondere più elementi del secondo insieme ma ad ogni elemento del secondo insieme può corrispondere uno ed un solo elemento del primo
- molti a molti: ad ogni elemento del primo insieme possono corrispondere più elementi del secondo e viceversa

Di seguito viene riportata l'evoluzione del modello concettuale utilizzando una strategia inside-out. Secondo questa strategia si individuano in un primo momento alcuni concetti importanti e si procede a macchia d'olio : prima i più vicini ai concetti iniziali, poi quelli più lontani.

9.6 Trasformazione al modello logico

In generale lo schema logico di un database è la realizzazione dello schema statico del progetto concettuale, eventualmente ottimizzato.

I modelli di schema tradizioni quali gerarchico e reticolare sono stati i primi ad essere sviluppati negli anni '60 ma, a causa della loro scarsa efficienza hanno trovato sempre minore applicazione nei sistemi reali.

Successivamente, si è sviluppato il modello relazionale, il cui scopo è quello di costruire lo schema logico di un database formato sull'uso delle tabelle, secondo le seguenti regole:

- Un tipo di entità, con i suoi attributi, diventa nel modello relazione una relazione e quindi una tabella.
- Un attributo composto diventa nel modello relazionale un gruppo di colonne o una sola colonna

9.6.1 Traduzione di un'associazione uno a molti

- I due tipi di entità dell'associazione diventano due tabelle; gli attributi dei due tipi di entità diventano le colonne delle due tabelle.
- Gli identificatori dei due tipi di entità diventano le chiavi primarie delle due tabelle
- Per mettere in relazione le due tabelle, occorre aggiungere nella tabella dalla parte a molti nuove colonne che contengono la chiave primaria della tabella del lato a uno. La chiave primaria della tabella del lato a uno, aggiunta nella tabella della parte a molti, definisce la sua chiave esterna.

9.6.2 Traduzione di un'associazione molti a molti

- I due tipi d'entità dell'associazione diventano due tabelle; gli attributi dei due tipi di entità diventano le colonne delle due tabelle e gli identificatori dei due tipi di entità diventano le chiavi primarie delle due tabelle
- L'associazione molti a molti diventa una nuova tabella; gli attributi dell'associazione molti a molti, se presenti, diventano le colonne della nuova tabella
- Le due tabelle dei tipi di entità sono in relazione uno a molti con la tabella dell'associazione. Per mettere in relazione le tre tabelle, occorre aggiungere le chiavi primarie delle due tabelle dei tipi di entità nella tabella dell'associazione. Le chiavi primarie formano le chiavi esterne delle due relazioni.
- La chiave primaria composta della tabella dell'associazione è formata dalla chiave primaria delle tabelle che ne derivano dei due tipi di entità e, se necessario, da una o più colonne dell'associazione

9.6.3 Traduzione di un'associazione uno a uno

Per realizzare la relazione uno a uno e l' identificatore esterno si deve aggiungere la chiave primaria di una delle due tabelle nell'altra; in questa tabella, la chiave primaria diventa anche la chiave esterna.

9.6.4 Traduzione di un'associazione n- aria

Un' associazione n-aria dello schema ER diventa una nuova tabella nello schema relazionale. Le n tabelle che provengono dai tipi di entità e la tabella dell'associazione n-aria sono legate tra loro mediante n nuove relazioni, in genere uno a molti.

9.6.5 Traduzione di un'associazione ricorsiva

Un' associazione ricorsiva dello schema ER diventa una nuova tabella nello schema relazionale. La tabella che deriva dall' associazione è legata al tipo di entità mediante due relazioni, in genere uno a molti

9.7 Modello logico

Di seguito viene riportato la trasformazione del modello concettuale sopra trattato nel modello logico rispettando le regole di trasformazione appena descritte:

allergy (id_allergy, name_allergy)

associallergia (id_typediet, id_allergy)

typediet (id_typediet, name_typediet)

diet (id_diet, id_typediet, dieta_diet)

composizionecibo (id_diet, id_food)

food (id_food, name_food)

composizionepasto (id_composizionepasto, id_diet, id_pasti, pasto_composizionepasto)

pasti (id_pasti, name_pasti)

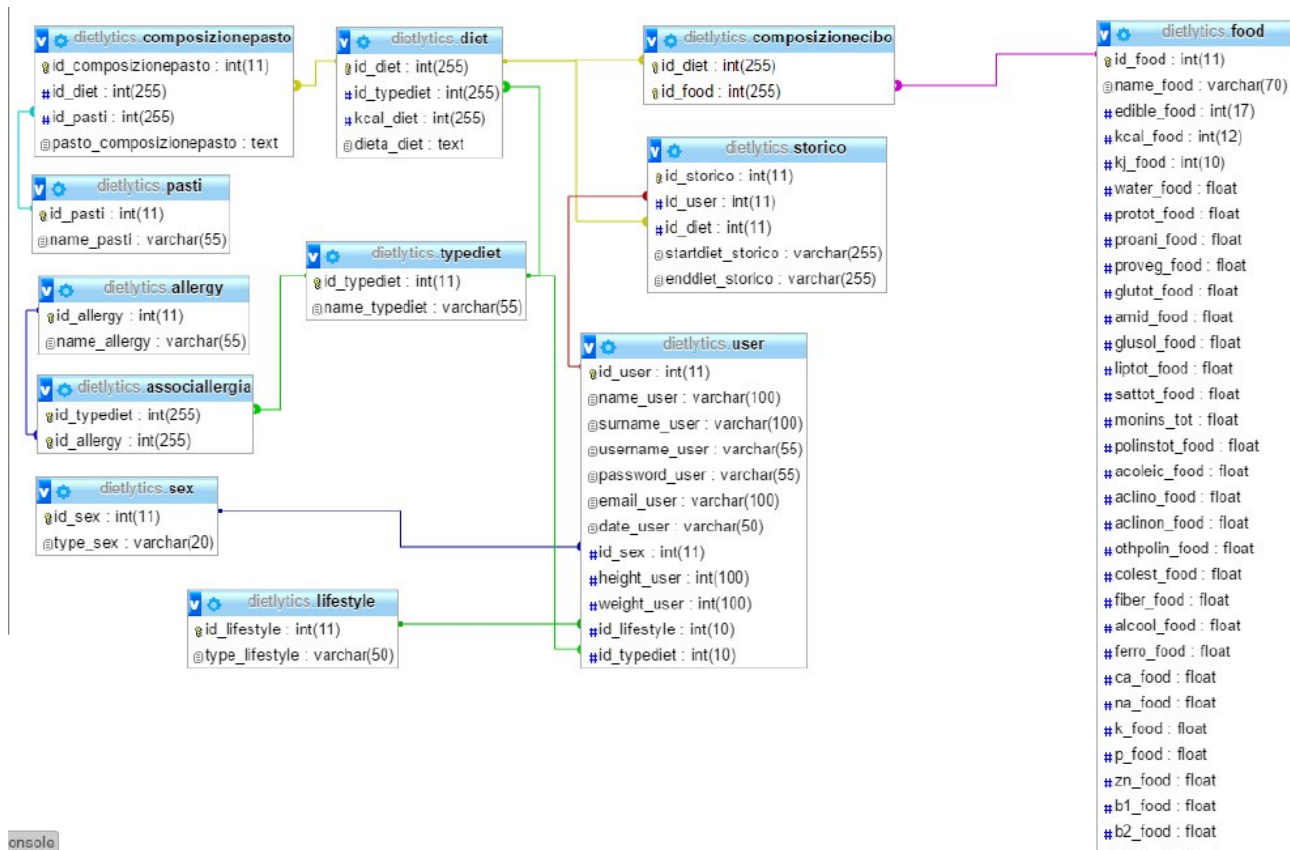
storico (id_storico, id_user, id_diet, startdiet_storico, enddiet_storico)

lifestyle (id_lifestyle, type_lifestyle)

user (id_user, name_user, surname_user, username_user, password_user, email_user, date_user, id_sex, height_user, weight_user, id_lifestyle, id_typediet)

sex (id_sex, type_sex)

Legenda : **relazioni**, chiave primaria, *chiave esterna*



9.7.1 Dettagli tabelle

Di seguito sono riportate le analisi delle tabelle presenti nel modello logico

9.7.1.1 allergy

La tabella “allergy” contiene i tipi di allergie che una persona può avere

I campi della tabella sono:

- **id_allergy**: chiave primaria. E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **name_allergy**: e’ un campo di tipo varchar che contiene i nomi dei tipi di allergia;

9.7.1.2 associallergia

La tabella “associallergia” è una tabella generata dalla relazione N a N di allergy e typediet

I campi della tabella sono:

- **id_typediet**: chiave primaria e esterna; E’ un campo intero che fa riferimento alla chiave primaria della tabella “typediet”;

- **id_allergy:** chiave primaria e esterna; E' un campo intero che fa riferimento alla chiave primaria della tabella "allergy";

9.7.1.3 typediet

La tabella "typediet" contiene i tipi di dieta che una persona può scegliere

I campi della tabella sono:

- **id_typediet:** chiave primaria; E' un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **name_typediet:** è un campo di tipo varchar che contiene i nomi delle diete da poter scegliere;

9.7.1.4 diet

La tabella "diet" contiene la struttura della dieta da seguire che è stata creata

I campi della tabella sono:

- **id_diet:** chiave primaria e esterna; E' un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **id_typediet:** è una chiave esterna che fa riferimento alla chiave primaria della tabella "typediet".
- **dieta_diet:** è un campo di tipo text che contiene la dieta che è stata creata

9.7.1.5 composizioneecibo

La tabella "composizionecibo" è una tabella generata dalla relazione N a N di diet e food

I campi della tabella sono:

- **id_diet:** chiave primaria e esterna; E' un campo intero che fa riferimento alla chiave primaria della tabella "diet";
- **id_food:** chiave primaria e esterna; E' un campo intero fa riferimento alla chiave primaria della tabella "food";

9.7.1.6 composizioneepasto

La tabella "composizionepasto" è una tabella generata dalla relazione N a N di diet e pasti

I campi della tabella sono:

- **id_composizionepasto:** chiave primaria; E' un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **id_pasti:** è una chiave esterna che fa riferimento alla chiave primaria della tabella "pasti";
- **id_diet:** è una chiave esterna che fa riferimento alla chiave primaria della tabella "diet";
- **pasto_composizionepasto:** è un campo di tipo text che contiene la composizione del pasto;

9.7.1.7 food

La tabella “food” contiene tutti i nomi degli alimenti presenti nel database

Alcuni campi della tabella sono:

- **id_food:** chiave primaria; E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **name_food:** è un campo di tipo varchar che contiene i nomi degli alimenti presenti nel database;

9.7.1.8 pasti

La tabella “pasti” contiene tutti i nomi della suddivisione dei pasti giornalieri

I campi della tabella sono:

- **id_pasti:** chiave primaria; E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **name_pasti:** è un campo di tipo varchar che contiene i nomi della suddivisione dei pasti giornalieri;

9.7.1.9 storico

La tabella “storico” è una tabella che contiene tutte le diete che sono state create per un singolo utente e tiene traccia delle date

I campi della tabella sono:

- **id_storico:** chiave primaria; E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **id_user:** è una chiave esterna che fa riferimento alla chiave primaria della tabella “user”;
- **id_diet:** è una chiave esterna che fa riferimento alla chiave primaria della tabella “diet”;
- **startdiet_storico:** è un campo di tipo varchar che contiene la data di inizio dieta;
- **enddiet_storico:** è un campo di tipo varchar che contiene la data di fine dieta;

9.7.1.10 lifestyle

La tabella “lifestyle” contiene tutti i nomi della suddivisione dei pasti giornalieri

I campi della tabella sono:

- **id_lifestyle:** chiave primaria; E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **type_lifestyle:** è un campo di tipo varchar che contiene i nomi dei tipi di stili di vita;

9.7.1.11 sex

La tabella “sex” contiene tutti i nomi della suddivisione dei pasti giornalieri

I campi della tabella sono:

- **id_sex:** chiave primaria; E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **type_sex:** è un campo di tipo varchar che contiene i nomi dei tipi di sesso;

9.7.1.12 user

La tabella “user” contiene tutti i dati degli utenti che si registrano

I campi della tabella sono:

- **id_user:** chiave primaria; E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento;
- **name_user:** è un campo di tipo varchar che contiene il nome dell’utente;
- **surname_user:** è un campo di tipo varchar che contiene il cognome dell’utente;
- **username_user:** è un campo di tipo varchar che contiene l’username dell’utente;
- **password_user:** è un campo di tipo varchar che contiene la password dell’utente;
- **email_user:** è un campo di tipo varchar che contiene l’email dell’utente;
- **date_user:** è un campo di tipo varchar che contiene la data di nascita dell’utente;
- **height_user:** è un campo di tipo varchar che contiene l’altezza dell’utente;
- **weight_user:** è un campo di tipo varchar che contiene il peso dell’utente;
- **id_sex:** E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento. è una chiave esterna che fa riferimento alla chiave primaria della tabella “sex”;
- **id_typediet:** E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento. è una chiave esterna che fa riferimento alla chiave primaria della tabella “typediet”;
- **id_lifestyle:** E’ un campo intero che autoincrementa il suo valore di una unità a ogni inserimento. è una chiave esterna che fa riferimento alla chiave primaria della tabella “diet”;
- **name_user:** è un campo di tipo varchar che contiene il nome dell’utente;

10 SQL

10.1.1 Database setup

Dietlytics per poter essere utilizzato ha bisogno di essere installato su un'istanza pulita di un server MySQL.

10.2 Tabelle

Una completa installazione richiede la creazione di molte tabelle. Ogni tabella del progetto viene di seguito documentata ed è provvista di spiegazione.

10.2.1 allergy

La seguente tabella viene creata, se già non esiste, per gestire le allergie. Il primo campo è un intero di 11 caratteri non nullo denominato “id_allergy” a cui successivamente viene assegnato il valore di chiave primaria e le viene data la proprietà di auto incrementarsi. Il secondo campo è un varchar di 55 caratteri non nullo denominato “name_allergy” che rappresenta il nome del tipo di allergia che una persona possa avere.

```
-- -----  
--  
-- Struttura della tabella `allergy`  
--  
CREATE TABLE IF NOT EXISTS `allergy` (  
  `id_allergy` int(11) NOT NULL,  
  `name_allergy` varchar(55) NOT NULL  
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;
```

10.2.2 associallergia

La seguente tabella viene creata, se già non esiste, per gestire la relazione tra allergy e typediet. Il primo campo è un intero di 255 caratteri non nullo denominato “id_typediet” a cui successivamente viene assegnato il valore di chiave primaria e esterna che si riferenzia “id_typediet” della tabella “typediet”. Il secondo campo è un int di 255 caratteri non nullo denominato “id_allergy” a cui successivamente viene assegnato il valore di chiave primaria e esterna che si riferenzia a “id_allergy” della tabella “allergy”.

```
-- -----  
--  
-- Struttura della tabella `associallergia`  
--  
CREATE TABLE IF NOT EXISTS `associallergia` (  
  `id_typediet` int(255) NOT NULL DEFAULT '0',  
  `id_allergy` int(255) NOT NULL DEFAULT '0'  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```


10.2.3typediet

La seguente tabella viene creata, se già non esiste, per gestire le allergie. Il primo campo è un intero di 11 caratteri non nullo denominato “id_typediet” a cui successivamente viene assegnato il valore di chiave primaria e le viene data la proprietà di auto incrementarsi. Il secondo campo è un varchar di 55 caratteri non nullo denominato “name_typediet” che rappresenta il nome del tipo di dieta che una persona possa scegliere.

```
-- -----  
--  
-- Struttura della tabella `typediet`  
--  
  
CREATE TABLE IF NOT EXISTS `typediet` (  
  `id_typediet` int(11) NOT NULL,  
  `name_typediet` varchar(55) NOT NULL  
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;
```

10.2.4Diet

La seguente tabella viene creata, se già non esiste, per gestire le diete che vengono create. Il primo campo è un intero di 255 caratteri non nullo denominato “id_diet” a cui successivamente viene assegnato il valore di chiave primaria e le viene data la proprietà di auto incrementarsi. Il secondo campo è un int di 255 caratteri nullo di default denominato “id_typediet” che rappresenta la chiave esterna che referencia “id_typediet” della tabella “typediet”. Il terzo campo è un text non nullo denominato “dieta_diet” che contiene la diete che sono state create.

```
-- -----  
--  
-- Struttura della tabella `diet`  
--  
  
CREATE TABLE IF NOT EXISTS `diet` (  
  `id_diet` int(255) NOT NULL,  
  `id_typediet` int(255) DEFAULT NULL,  
  `kcal_diet` int(255) NOT NULL,  
  `dieta_diet` text NOT NULL  
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8;
```

10.2.5 Composizione cibo

La seguente tabella viene creata, se già non esiste, per gestire la relazione tra diet e food. Il primo campo è un intero di 255 caratteri non nullo denominato “id_diet” a cui successivamente viene assegnato il valore di chiave primaria e esterna che si riferenzia “id_diet” della tabella “diet”. Il secondo campo è un int di 255 caratteri non nullo denominato “id_food” a cui successivamente viene assegnato il valore di chiave primaria e esterna che si riferenzia a “id_food” della tabella “food”.

```
--  
-- Struttura della tabella `composizionedecibo`  
--  
CREATE TABLE IF NOT EXISTS `composizionedecibo` (  
  `id_diet` int(255) NOT NULL DEFAULT '0',  
  `id_food` int(255) NOT NULL DEFAULT '0'  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
--
```

10.2.6 Food

La seguente tabella viene creata, se già non esiste, per gestire gli alimenti. Il primo campo è un intero di 11 caratteri non nullo denominato “id_typediet” a cui successivamente viene assegnato il valore di chiave primaria e le viene data la proprietà di auto incrementarsi. Il secondo campo è un varchar di 70 caratteri nullo denominato “name_food” che rappresenta il nome degli alimenti.

```
--
-- Struttura della tabella `food`
--

CREATE TABLE IF NOT EXISTS `food` (
  `id_food` int(11) NOT NULL,
  `name_food` varchar(70) DEFAULT NULL,
  `edible_food` int(17) DEFAULT NULL,
  `kcal_food` int(12) DEFAULT NULL,
  `kj_food` int(10) DEFAULT NULL,
  `water_food` float DEFAULT NULL,
  `protot_food` float DEFAULT NULL,
  `proani_food` float DEFAULT NULL,
  `proveg_food` float DEFAULT NULL,
  `glutot_food` float DEFAULT NULL,
  `amid_food` float DEFAULT NULL,
  `glusol_food` float DEFAULT NULL,
  `liptot_food` float DEFAULT NULL,
  `sattot_food` float DEFAULT NULL,
  `monins_tot` float DEFAULT NULL,
  `polinstot_food` float DEFAULT NULL,
  `acoleic_food` float DEFAULT NULL,
  `aclino_food` float DEFAULT NULL,
  `aclinon_food` float DEFAULT NULL,
  `othpolin_food` float DEFAULT NULL,
  `colest_food` float DEFAULT NULL,
  `fiber_food` float DEFAULT NULL,
  `alcool_food` float DEFAULT NULL,
  `ferro_food` float DEFAULT NULL,
  `ca_food` float DEFAULT NULL,
  `na_food` float DEFAULT NULL,
  `k_food` float DEFAULT NULL,
  `p_food` float DEFAULT NULL,
  `zn_food` float DEFAULT NULL,
  `b1_food` float DEFAULT NULL,
  `b2_food` float DEFAULT NULL,
  `b3_food` float DEFAULT NULL,
  `c_food` float DEFAULT NULL,
  `b6_food` float DEFAULT NULL,
  `folic_food` float DEFAULT NULL,
  `retin_food` float DEFAULT NULL,
  `betacat_food` float DEFAULT NULL,
  `e_food` float DEFAULT NULL,
  `d_food` float DEFAULT NULL
) ENGINE=InnoDB AUTO_INCREMENT=790 DEFAULT CHARSET=utf8;
```

10.2.7 Composizione pasto

La seguente tabella viene creata, se già non esiste, per gestire la relazione tra diet e pasti. Il primo campo è un intero di 11 caratteri non nullo denominato “id_composizionecibo” a cui successivamente viene assegnato il valore di chiave primaria e le viene data la proprietà di auto incrementarsi. Il secondo campo è un int di 255 caratteri nullo di denominato “id_diet” è una chiave esterna che si riferenzia a “id_diet” della tabella “diet”. Il terzo campo è un int di 255 caratteri nullo di denominato “id_pasti” è una chiave esterna che si riferenzia a “id_pasti” della tabella “pasti”. Il quarto campo è di tipo text denominato “pasto_composizionepasto” non nullo che contiene la composizione dei pasti.

```
--
-- Struttura della tabella `composizionepasto`
--

CREATE TABLE IF NOT EXISTS `composizionepasto` (
  `id_composizionepasto` int(11) NOT NULL,
  `id_diet` int(255) DEFAULT NULL,
  `id_pasti` int(255) DEFAULT NULL,
  `pasto_composizionepasto` text NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-----
```

10.2.8Pasti

La seguente tabella viene creata, se già non esiste, per gestire i pasti. Il primo campo è un intero di 11 caratteri non nullo denominato “id_pasti” a cui successivamente viene assegnato il valore di chiave primaria e le viene data la proprietà di auto incrementarsi. Il secondo campo è un varchar di 55 caratteri non nullo denominato “name_pasti” che rappresenta il nome dei pasti da suddivedere nella giornata.

```
-----

--
-- Struttura della tabella `pasti`
--

CREATE TABLE IF NOT EXISTS `pasti` (
  `id_pasti` int(11) NOT NULL,
  `name_pasti` varchar(55) NOT NULL
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;
```

10.2.9Lifestyle

La seguente tabella viene creata, se già non esiste, per gestire gli stili di vita. Il primo campo è un intero di 11 caratteri non nullo denominato “id_lifestyle” a cui successivamente viene assegnato il valore di chiave primaria. Il secondo campo è un varchar di 50 caratteri non nullo denominato “type_lifestyle” che rappresenta il nome degli stili di vita adottati .

```
-- -----
--
-- Struttura della tabella `lifestyle`
--
CREATE TABLE IF NOT EXISTS `lifestyle` (
  `id_lifestyle` int(11) NOT NULL,
  `type_lifestyle` varchar(50) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

10.2.10 Storico

La seguente tabella viene creata, se già non esiste, per salvare tutte le diete che sono state create per una persona. Il primo campo è un intero di 11 caratteri non nullo denominato “id_storico” a cui successivamente viene assegnato il valore di chiave primaria e le viene data la proprietà di auto incrementarsi. Il secondo campo è un int di 11 caratteri nullo denominato “id_user” è una chiave esterna che si riferenzia a “id_user” della tabella “user”. Il terzo campo è un int di 11 caratteri nullo denominato “id_diet” è una chiave esterna che si riferenzia a “id_diet” della tabella “diet”. Il quarto campo è di tipo varchar di 255 denominato “startdiet_storico” nullo che contiene la data di inizio dieta. Il quinto campo è di tipo varchar di 255 denominato “enddiet_storico” nullo che contiene la data di fine dieta.

```
--
-- Struttura della tabella `storico`
--
CREATE TABLE IF NOT EXISTS `storico` (
  `id_storico` int(11) NOT NULL,
  `id_user` int(11) NOT NULL,
  `id_diet` int(11) NOT NULL,
  `startdiet_storico` varchar(255) DEFAULT NULL,
  `enddiet_storico` varchar(255) DEFAULT NULL
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=latin1;
```

10.2.11 Sex

La seguente tabella viene creata, se già non esiste, per gestire il sesso di una persona. Il primo campo è un intero di 11 caratteri non nullo denominato “id_sex” a cui successivamente viene assegnato il valore di chiave primaria e la possibilità di autoincrementarsi. Il secondo campo è un varchar di 20 caratteri non nullo denominato “type_sex” che rappresenta il nome dei tipi di sesso.

```
--
-- Struttura della tabella `sex`
--

CREATE TABLE IF NOT EXISTS `sex` (
  `id_sex` int(11) NOT NULL,
  `type_sex` varchar(20) NOT NULL
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

--
```

10.2.12 user

La seguente tabella viene creata, se già non esiste, per gestire gli utenti. Il primo campo è un intero di 11 caratteri non nullo denominato “id_user” a cui successivamente viene assegnato il valore di chiave primaria e la possibilità di autoincrementarsi . Il secondo campo è un varchar di 100 caratteri non nullo denominato “name_user” che rappresenta il nome degli utenti. Il terzo campo è un varchar di 100 caratteri non nullo denominato “surname_user” che rappresenta il cognome degli utenti. Il quarto campo è un varchar di 55 caratteri non nullo denominato “username_user” che rappresenta l’username degli utenti. Il quinto campo è un varchar di 55 caratteri non nullo denominato “password_user” che rappresenta la password degli utenti. Il sesto campo è un varchar di 100 caratteri non nullo denominato “email_user” che rappresenta l’email degli utenti. Il settimo campo è un varchar di 50 caratteri nullo denominato “date_user” che rappresenta la data di nascita degli utenti. L’ottavo campo è un int di 11 caratteri non nullo denominato “id_sex” è una chiave esterna che si riferenzia a “id_sex” della tabella “sex”. Il nono campo è un int di 100 caratteri non nullo denominato “height_user” che rappresenta l’altezza degli utenti. Il decimo campo è un int di 100 caratteri non nullo denominato “weight_user” che rappresenta il peso degli utenti. L’undicesimo campo è un int di 10 caratteri non nullo denominato “id_lifestyle” è una chiave esterna che si riferenzia a “id_lifestyle” della tabella “lifestyle”. Il dodicesimo campo è un int di 10 caratteri non nullo denominato “id_typediet” è una chiave esterna che si riferenzia a “id_typediet” della tabella “typediet”.

```
--
-- Struttura della tabella `user`
--

CREATE TABLE IF NOT EXISTS `user` (
  `id_user` int(11) NOT NULL,
  `name_user` varchar(100) NOT NULL,
  `surname_user` varchar(100) NOT NULL,
  `username_user` varchar(55) NOT NULL,
  `password_user` varchar(55) NOT NULL,
  `email_user` varchar(100) NOT NULL,
  `date_user` varchar(50) DEFAULT NULL,
  `id_sex` int(11) NOT NULL,
  `height_user` int(100) NOT NULL,
  `weight_user` int(100) NOT NULL,
  `id_lifestyle` int(10) DEFAULT NULL,
  `id_typediet` int(10) DEFAULT NULL
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
```

11 Analisi di implementazione

11.1 Generale

L'applicazione è stata sviluppata utilizzando Java come linguaggio di programmazione. Come IDE è stato utilizzato Eclipse in versione Mars Release. La versione di JDK (Java Development Kit) che è stata utilizzata è la 1.8.

11.2 Dietlyrics

Dietlytics è un applicazione client/server per la creazione di una dieta. Il server risulta essere un server multi-thread con la possibilità per più client di potersi connettere cercando di avere delle prestazioni comunque elevate. Al contrario del client per il server non è stata pensata un'interfaccia grafica ma semplicemente è possibile poterlo utilizzarlo da riga di comando. Per quanto riguarda il client, esso è dotato di un interfaccia grafica.

11.3 Model-view-controller

L'architettura dell'applicazione è stata strutturata cercando di massimizzare la modularità e la riusabilità delle classi, utilizzando dunque i principi dell'Object-Oriented-Programming. E' stato utilizzato proprio per questo motivo il Model-View-Controller (Modello-vista-Controllo).

Il Model-View-Controller è un pattern architetturale molto diffuso in grado di separare la logica di presentazione dei dati. Il pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali

- Model
- View
- Controller

Il Model fornisce i metodi per accedere ai dati utili all'applicazione.

Il View visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti.

Il controller riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato degli altri due componenti.

11.4 Package

Per una maggiore manutenibilità e classificazione delle operazioni implementate è stata pensata una struttura con i seguenti package :

- **client**
- **comunicazione**
- **controller**
- **database**
- **debugger**
- **dieta**
- **Model**
- **Server**

11.4.1 Client

Il package client è stato implementato in relazione al package Server.

La classe definita al suo interno è denominata “Client” e serve per dichiarare la socket del client e i canali di comunicazione. Sono stato utilizzati due ObjectOutputStream, il primo per mandare verso il server e il secondo per ricevere dal server.

```
public class Client extends Application{

    public void start(Stage primaryStage) throws Exception {
        Socket clientSocket;
        ObjectOutputStream versoServer;
        ObjectInputStream dalServer;
        try
        {
            clientSocket = new Socket ("127.0.0.1",6001);
            versoServer = new ObjectOutputStream(clientSocket.getOutputStream());
            dalServer = new ObjectInputStream(clientSocket.getInputStream());

            FXMLLoader loader = new F
XMLLoader(getClass().getResource("../view/loginregistration.fxml"));
            scene = new Scene(loader.load());
            Stage = primaryStage;
            Stage.setScene(scene);
            Stage.show();
            InterfacciaIniziale controller =
            loader.<InterfacciaIniziale>getController();
            controller.initializePage(versoServer,dalServer);
        } catch(Exception exc) {
            System.out.println("Errore-InitialieDefaultCartController: " +
            exc.getMessage());
            exc.printStackTrace();
        }
    }
}
```

11.4.2 Comunicazione

Il package Comunicazione è stato implementato per gestire lo scambio dei messaggi tra il client e il server. Troviamo al suo interno quattro classi :

- Richiesta
- Risposta
- TipiRichieste
- TipiRisposte

La classe Richiesta viene utilizzata per poter gestire al meglio le richieste del client. Ogni richiesta conterrà un campo tipo che identifica la richiesta e un oggetto.

```
public class Richiesta implements Serializable {  
  
    public Richiesta(int pTipo, Object pOggetto)  
    {  
        this.Tipo = pTipo;  
        this.Oggetto = pOggetto;  
    }  
  
    public int getTipo()  
    {  
  
    }  
  
    public void setTipo(int tipo)  
    {  
  
    }  
  
    public Object getOggetto()  
    {  
  
    }  
  
    public void setOggetto(Object oggetto)  
    {  
  
    }  
}
```

La classe Risposta viene utilizzata per gestire le risposte da parte del server. E' presente un campo tipo per inviare al richiedente le informazioni che ha richiesto e un oggetto.

```
public class Risposta implements Serializable{  
  
    public Risposta(int pTipo, Object pOggetto)  
    {  
        this.Tipo = pTipo;  
        this.Oggetto = pOggetto;  
    }  
  
    public int getTipo()  
    {  
  
    }  
  
    public void setTipo(int tipo)  
    {  
  
    }  
  
    public Object getOggetto()  
    {  
  
    }  
  
    public void setOggetto(Object oggetto)  
    {  
  
    }  
}
```

Le classi TipiRichieste e TipiRisposte identificano le richieste e le risposte

```
public class TipiRichieste
{
    // MCredenziali
    public final static int Registrazione = 0;

    public final static int Login = 1;

    // MUser
    public final static int UtenteCrea = 2;

    public final static int UtenteModifica = 3;

    public final static int UtenteCancella = 4;

    public final static int AbitudiniUtente = 5;

    public final static int InizializzaUtente = 6;

    public final static int InformazioniUtenteLoggato = 12;

    //MUtility
    public final static int InizializzaMenu = 7;

    //MFood
    public final static int RicercaCibo = 8;

    //MDiet
    public final static int NuovaDieta = 9;

    public final static int InizializzaStorico = 10;

    public final static int StampaDieta = 11;

    // ...
}
```

```
public class TipiRisposte {
    public final static int LoginSuccesso = 0;

    public final static int LoginFallimento = 1;

    public final static int RegistrazioneSuccesso = 2;

    public final static int RegistrazioneFallimento = 3;

    public final static int WelcomeSuccesso = 4;

    public final static int WelcomeFallimento = 5;

    public final static int CreaUtenteSuccesso = 6;

    public final static int CreaUtenteFallimento = 7;

    public final static int ModificaUtenteSuccesso = 8;
```

```
public final static int ModificaUtenteFallimento = 9;
public final static int EliminaUtenteSuccesso = 10;
public final static int EliminaUtenteFallimento = 11;
public final static int RicercaCiboSuccesso = 12;
public final static int RichiestaAbitudiniSuccesso = 13;
public final static int RichiestaAbitudiniFallimento = 14;
public final static int RichiestaInizializzaUtenteSuccesso = 15;
public final static int RichiestaNuovaDietaSuccesso = 16;
public final static int RichiestaInizializzaTpDietaSuccesso = 17;
public final static int RichiestaInizializzaTpDietaFallimento = 18;
public final static int RichiestaInizializzaDietaSuccesso = 19;
public final static int RichiestaInizializzaDietaFallimento = 20;
public final static int RichiestaInformazioniUtenteLoggato = 21;
}
```

11.4.3 Debugger

Il package Debugger è stato implementata in correlazione al package “comunicazione”. La classe definita al suo interno è denominata “DebugRegistration”. E' in questa classe che vengono gestiti i messaggi d'errore o di semplice avviso nelle risposte del server dopouna generica richiesta.

```
public class DebugRegistration {

    public static boolean UsernameCheck(String Username)

    public static boolean EmailCheck(String Email)

    public static boolean EmailOk(String regEmail)

    public static String DataCheck(String YYYYbirthday, String MMbirthday, String
DDbirthday)
```

11.4.4 Database

Il package database è stato implementato per gestire la comunicazione col database. La classe definita al suo interno è denominata “connessionedatabase”. E' in questa classe che viene gestita la connessione al database. Vengono dunque creati in un primo momento gli statement e preparati i driver che verranno utilizzati successivamente.

```
public class ConnessioneDatabase {
    static public Statement cmd;
    static public Connection connessione;
    static String db_name = "Dietlytics";
    static String user_name = "admin";
    static String user_pass = "admin";
    static String driver = "com.mysql.jdbc.Driver";
    static String url = "jdbc:mysql://localhost/";

    public static void Connetti() {
        try {
            Class.forName(driver);
            connessione = DriverManager.getConnection(url + db_name, user_name,
user_pass);
            cmd = connessione.createStatement();
        } catch (SQLException ex1) {
            ex1.printStackTrace();
        } catch (ClassNotFoundException ex1) {
            ex1.printStackTrace();
        }
    }

    public static void Disconnetti() {

    }

}
```

11.4.5 Server

Il package Server è stato implementato per rappresentare la figura del server. Al suo interno troviamo svariate classi che sono state utilizzate e implementate per smistare al meglio le richieste provenienti dal client. Iniziamo con l'analisi della classe principale per poi suddividere le altre in base alla loro attività.

```
class ServerThread extends Thread {
    public boolean Attivo = true;

    MUser utenteLoggato;

    Socket socket;
    ObjectInputStream dalClient;
    ObjectOutputStream versoClient;

    ServerThread(Socket socket) {
        this.socket = socket;
    }

    public void gestisciUtenteCrea(MUser m)

    public void gestisciLogin(MCredenziali m)

    public void gestisciRegistrazione(MCredenziali r)

    public void gestisciInizializzazioneMenu()

    public void inizializzaUtente()

    public void readMatrix()

    public void gestisciAbitudiniUtente()

    public static double calcolaKcal(int userid) {

    public void gestisciNuovaDieta(MDiet r)

    public void gestisciRicercaCibo(MDiet dieta)

    public void gestisciUtenteCancella()

    public void gestisciUtenteModifica(MCredenziali r)

    public void gestisciInizializzaDieta()

    public void gestisciStampaDieta(int r)

    public void gestisciInformazioniUtenteLoggato(MUser utenteTemp)

    public void run()

public class Server{
    public static void main(String[] args) {
```

```

ServerSocket ssock = null;
Socket sock = null;
try {
    ssock = new ServerSocket(6001);
    System.out.println("In attesa del client...");

    while (true) {
        System.out.println("Server Startato: while true threads in
ascolto...");

        sock = ssock.accept();

        ServerThread clientThread = new ServerThread(sock);
        clientThread.start();
    }
} catch (Exception e) {
    System.out.println(e.getMessage() + "->" + ssock);
} finally {
    if (ssock != null) {
        try {
            ssock.close();
        } catch (Exception e) {
        }
    }
}
}
}

```

11.4.6 Controller

Verranno di seguito analizzati solo tre esempi principali in quanto comprendono l'intera varietà dei controller citati.

- InterfacciaIniziale
- InterfacciaNuovaDieta
- InterfacciaStorico

La classe interfacciaIniziale è utilizzata per gestire la grafica di login o registrazione

```

public class InterfacciaIniziale implements Initializable {

    ObjectOutputStream versoServer;
    ObjectInputStream dalServer;
    Scene scene;

    SimpleDateFormat formatter = new SimpleDateFormat("dd-MMM-yyyy");

    public void initialize(URL location, ResourceBundle resources)

    public void Clear()

    public void Login(ActionEvent event)

    public void Registration(ActionEvent event)

```

La classe InterfacciaNuovaDieta è utilizzata per gestire la creazione di una nuova dieta

```
public class InterfacciaNuovaDieta {  
  
    ObjectOutputStream versoServer;  
    ObjectInputStream dalServer;  
    Scene scene;  
  
    public void backhome(MouseEvent Event)  
  
    public void startNewDiet(ActionEvent e)  
  
public void initializePage(ObjectOutputStream versoServer, ObjectInputStream dalServer)
```

La classe InterfacciaStorico viene utilizzata per gestire la visualizzazione di tutte le diete create da un utente

```
public class InterfacciaStorico{  
  
    ObjectOutputStream versoServer;  
    ObjectInputStream dalServer;  
    Scene scene;  
  
public void initializePage(ObjectOutputStream versoServer, ObjectInputStream dalServer)  
  
public void backhome(MouseEvent Event)  
  
public void pane()  
  
public void print() throws TransformerException  
  
public void stampaXML() throws TransformerException  
  
public Document creaDocumentoXml()  
}
```

11.4.7dieta

Il package dieta è stato implementato per creare l'algoritmo che genererà la quantità dei singoli cibi da assumere in una dieta.

Nella classe algoritmo troviamo l'algoritmo che calcola il coefficiente

```
public static double algorithm(){  
  
//prelevo l'anno di nascita dalla data di nascita  
    String split = a.toString();  
    System.out.println(split);  
    String[] items = split.split("-");  
    String YYYYbirthday = items[0];  
    //calcolo l'età  
    age= 2016 - Integer.parseInt(YYYYbirthday);  
  
    sex=Integer.parseInt(s.toString());  
    height=Integer.parseInt(h.toString());  
    weight=Integer.parseInt(w.toString());  
    lifestyle=Float.parseFloat(l.toString());
```



```

        System.out.println(age);
        System.out.println(s);
        System.out.println(h);
        System.out.println(w);
        System.out.println(l);

        if (sex==1)
            MBR = (9.99*weight)+(4.92*height)+(4.92*age)+5;
        else
            MBR = (9.99*weight)+(4.92*height)+(4.92*age)-161;

        TID = MBR*0.10;

        return MBR*lifestyle+TID;
    }
}

```

Nella classe composizioneDieta troviamo tutti i tipi di diete predefinite alle quali l'algoritmo andrà ad assegnare la quantità, l'utente sceglierà quale dieta verrà creata in base ad eventuali allergie e al tipo che vuole adottare

```

public class ComposizioneDieta {

    public static String composizioneDieta(String nomeUtente, int kcal, int typediet,
int allergia)

```

11.4.8 Model

Il package Model è stato implementato per rappresentare le entità del database Dietlytics. E' in questo package che troviamo le model utilizzati nel Model-View- Controller.

Al suo interno troviamo svariate classi che sono state utilizzate e implementate per suddividere al meglio le entità. Procediamo con la loro analisi

11.4.3.1 Mdiet

La seguente classe viene utilizzata per gestire le diete:

```

public class MDiet implements Serializable {

    public MDiet(int id, int id_typediet, int kcal, String dieta) {
        this.id = id;
        this.id_typediet = id_typediet;
        this.kcal = kcal;
        this.dieta = dieta;
    }

    public int getId()

    public void setId(int id)

```

```

    public int getId_typediet()

    public void setId_typediet(int id_typediet)

    public int getKcal()

    public void setKcal(int kcal)

    public String getDieta()

    public void setDieta(String dieta)
}

```

11.4.3.2 MUser

```

public class MUser implements Serializable {

    public MUser(int id, String name, String surname, String username, String password,
String email, String nascita,int altezza, int peso, int sesso, int stiledivita, int
tipodieta, float fattore) {
        this.id = id;
        this.name = name;
        this.surname = surname;
        this.username = username;
        this.password = password;
        this.email = email;
        this.nascita = nascita;
        this.altezza = altezza;
        this.peso = peso;
        this.sesso = sesso;
        this.stiledivita = stiledivita;
        this.tipodieta = tipodieta;
        this.fattore = fattore;
    }
    public int getId()

    public void setId(int id)

    public String getName()

    public void setName(String name)
}

```

```

    public String getSurname()
    public void setSurname(String surname)
    public String getUsername()
    public void setUsername(String username)
    public String getPassword()
    public void setPassword(String password)
    public String getEmail()
    public void setEmail(String email)
    public String getNascita()
    public void setNascita(String nascita)
    public int getAltezza()
    public void setAltezza(int altezza)
    public int getPeso()
    public void setPeso(int peso)
    public int getSesso()
    public void setSesso(int sesso)
    public int getStiledivita()
    public void setStiledivita(int stiledivita)
    public int getTipodieta()
    public void setTipodieta(int tipodieta)
    public float getFattore()
    public void setFattore(float fattore)
}

```

11.4.3.3 Mtypediet

```

public class MTypediet implements Serializable {

    public MTypediet(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId()

    public void setId(int id)

    public String getName()

    public void setName(String name)

}

```

11.4.3.4 MUtility

```

public class MUtility implements Serializable{

```

```

    public MUtility(String welcome) {
        this.Welcome = welcome;
    }
    public String getWelcome()

    public void setWelcome(String welcome)

    private void writeObject(ObjectOutputStream aOutputStream) throws IOException {
        aOutputStream.writeObject(Welcome);
    }

    private void readObject(ObjectInputStream aInputStream) throws
    ClassNotFoundException, IOException {
        Welcome = (String) aInputStream.readObject();
    }
}

```

11.4.3.5 MCredenziali

```

public class MCredenziali implements Serializable{

    public MCredenziali(String pUsername, String pPassword) {
        this.Username = pUsername;
        this.Password = pPassword;
    }
    public String getUsername()

    public void setUsername(String username)

    public String getPassword()

    public void setPassword(String password)

    private void writeObject(ObjectOutputStream aOutputStream) throws IOException {
        aOutputStream.writeObject(Username);
        aOutputStream.writeObject>Password);
    }

    private void readObject(ObjectInputStream aInputStream) throws
    ClassNotFoundException, IOException {

```

```

        Username = (String) aInputStream.readObject();
        Password = (String) aInputStream.readObject();
    }
}

```

11.4.3.6 MStorico

```

public class MStorico implements Serializable {

    public MStorico(int id, int id_user, int id_diet, String start_diet, String
end_diet) {
        this.id = id;
        this.id_user = id_user;
        this.id_diet = id_diet;
        this.start_diet = start_diet;
        this.end_diet = end_diet;
    }

    public int getId()

    public void setId(int id)

    public int getId_user()

    public void setId_user(int id_user)

    public int getId_diet()

    public void setId_diet(int id_diet)

    public String getStart_diet()

    public void setStart_diet(String start_diet)

    public String getEnd_diet()

    public void setEnd_diet(String end_diet)

}

```

11.4.3.7 MFood

```
public class MFood implements Serializable {
    public MFood(int id, String name, int edible, int kcal, int kj, int water, int
    protot, int proani, int proveg, int glucot, int amid, int glucos, int liptot, int
    sattot, int monins, int polinstot, int acoleic, int aclino, int aclinon, int
    othpolin, int colest, int fiber, int alcool, int ferro, int ca, int na, int k,
    int p, int zn, int b1, int b2, int b3, int c, int b6, int folic, int retin, int
    betacat, int e, int d) {
        this.id = id;
        this.name = name;
        this.edible = edible;
        this.kcal = kcal;
        this.kj = kj;
        this.water = water;
        this.protot = protot;
        this.proani = proani;
        this.proveg = proveg;
        this.glucot = glucot;
        this.amid = amid;
        this.glucos = glucos;
        this.liptot = liptot;
        this.sattot = sattot;
        this.monins = monins;
        this.polinstot = polinstot;
        this.acoleic = acoleic;
        this.aclino = aclino;
        this.aclinon = aclinon;
        this.othpolin = othpolin;
        this.colest = colest;
        this.fiber = fiber;
        this.alcool = alcool;
        this.ferro = ferro;
        this.ca = ca;
        this.na = na;
        this.k = k;
        this.p = p;
        this.zn = zn;
        this.b1 = b1;
        this.b2 = b2;
        this.b3 = b3;
        this.c = c;
        this.b6 = b6;
        this.folic = folic;
        this.retin = retin;
        this.betacat = betacat;
        this.e = e;
        this.d = d;
    }

    public int getId()
    public void setId(int id)
    public String getName()
    public void setName(String name)
    public int getEdible()
    public void setEdible(int edible)
```

```

public int getKcal()
public void setKcal(int kcal)
public int getKj()
public void setKj(int kj)
public int getWater()
public void setWater(int water)
public int getProtot()
public void setProtot(int protot)
public int getProani()
public void setProani(int proani)
public int getProveg()
public void setProveg(int proveg)
public int getGlucot()
public void setGlucot(int glucot)
public int getAmid()
public void setAmid(int amid)
public int getGlucos()
public void setGlucos(int glucos)
public int getLiptot()
public void setLiptot(int liptot)
public int getSattot()
public void setSattot(int sattot)
public int getMonins()
public void setMonins(int monins)
public int getPolinstot()
public void setPolinstot(int polinstot)
public int getAcoleic()
public void setAcoleic(int acoleic)
public int getAclino()
public void setAclino(int aclino)
public int getAclinon()
public void setAclinon(int aclinon)
public int getOthpolin()
public void setOthpolin(int othpolin)
public int getColest()
public void setColest(int colest)
public int getFiber()
public void setFiber(int fiber)
public int getAlcool()
public void setAlcool(int alcool)
public int getFerro()
public void setFerro(int ferro)
public int getCa()
public void setCa(int ca)
public int getNa()
public void setNa(int na)
public int getK()
public void setK(int k)
public int getP()
public void setP(int p)
public int getZn()
public void setZn(int zn)
public int getB1()
public void setB1(int b1)
public int getB2()
public void setB2(int b2)
public int getB3()
public void setB3(int b3)
public int getC()
public void setC(int c)
public int getB6()
public void setB6(int b6)

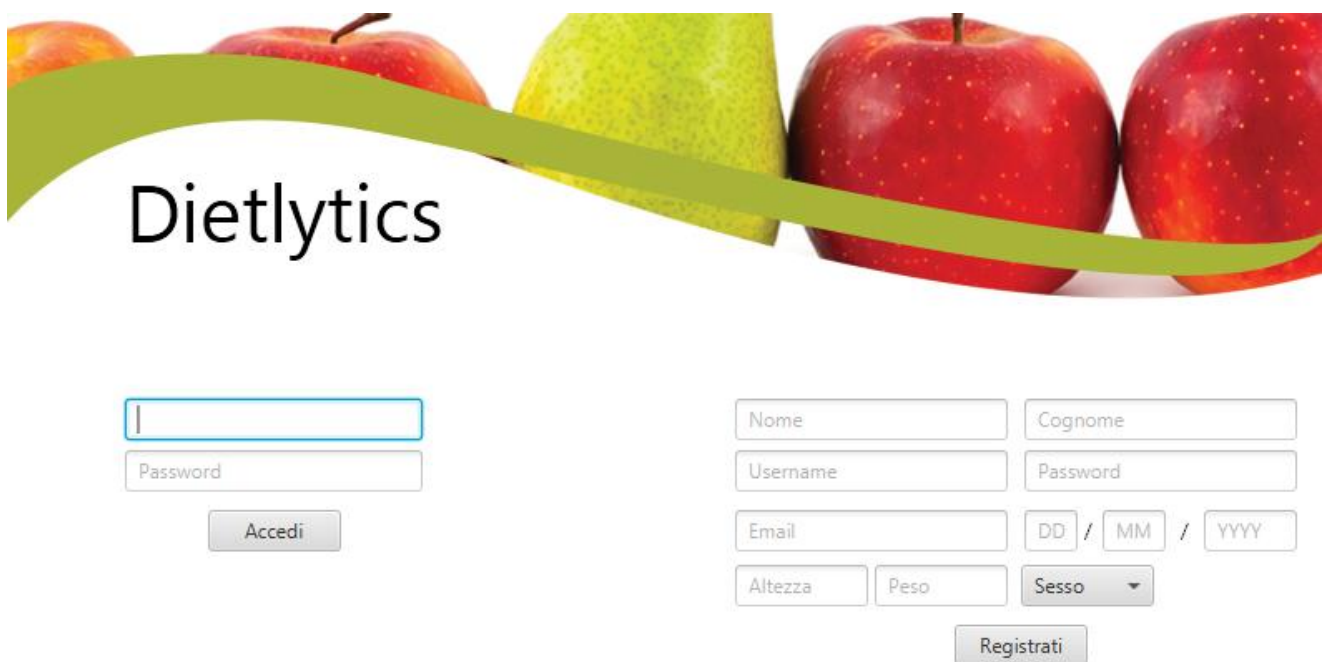
```

```
public int getFolic()
public void setFolic(int folic)
public int getRetin()
public void setRetin(int retin)
public int getBetacat()
public void setBetacat(int betacat)
public int getE()
public void setE(int e)
public int getD()
public void setD(int d)
```

```
}
```


12 Screenshot

12.4 Login/registrazione



The image shows a web interface for 'Dietlytics' with a header featuring a row of fruit (apples and a pear) and a green wavy line. Below the header, there are two main sections: a login form on the left and a registration form on the right.

Header: Dietlytics

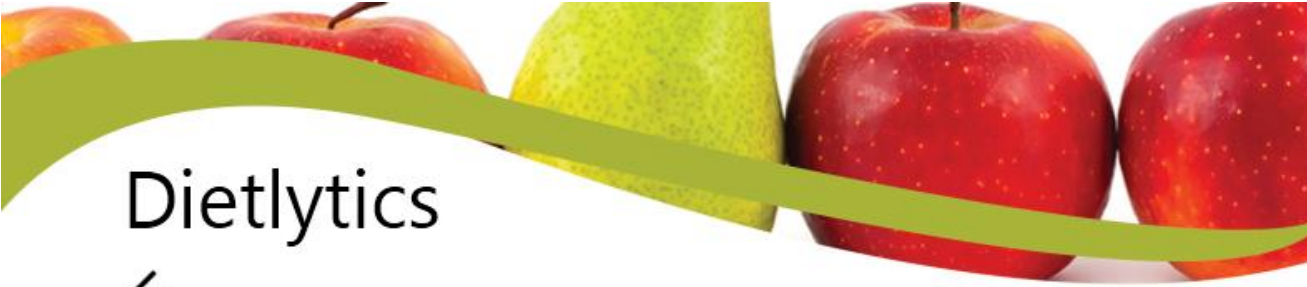
Login Form:

- Input field for Username
- Input field for Password
- Button: Accedi

Registration Form:

- Input field: Nome
- Input field: Cognome
- Input field: Username
- Input field: Password
- Input field: Email
- Input field: DD / MM / YYYY
- Input field: Altezza
- Input field: Peso
- Dropdown menu: Sesso
- Button: Registrati

12.5 Modifica dati personali

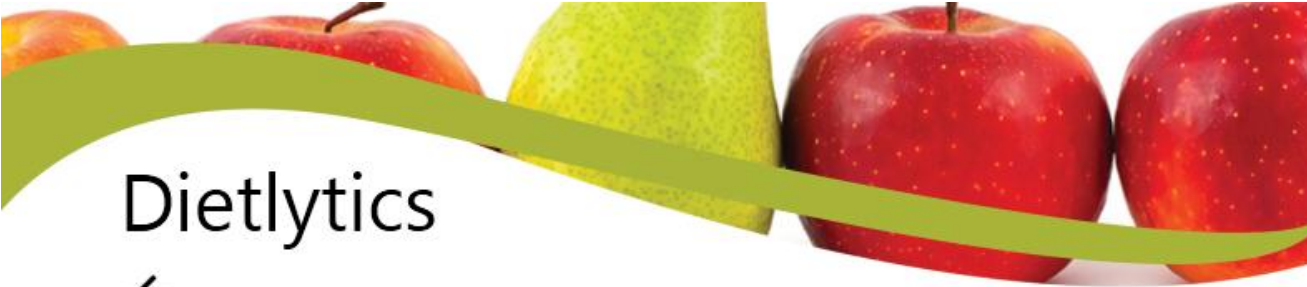


Dietlytics

←

<input data-bbox="805 936 1082 981" type="text" value="zack"/>	<input data-bbox="1109 936 1385 981" type="text" value="zack"/>
<input data-bbox="805 1003 1082 1041" type="text" value="zack"/>	<input data-bbox="1109 1003 1161 1041" type="text" value="18"/> / <input data-bbox="1193 1003 1241 1041" type="text" value="2"/> / <input data-bbox="1273 1003 1385 1041" type="text" value="1994"/>
<input data-bbox="805 1064 941 1102" type="text" value="170"/>	<input data-bbox="949 1064 1082 1102" type="text" value="60"/> <input data-bbox="1109 1064 1241 1102" type="text" value="M"/>

12.6 Modifica abitudini alimentary



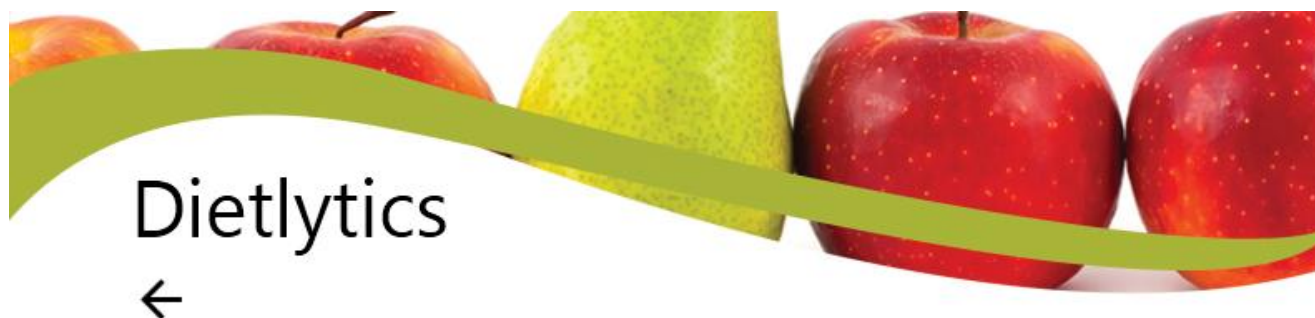
Dietlytics

←

Colazione	<input type="text" value="Spremuta/Succo"/>	<input type="text" value="Cornetto/Sfoglia"/>	<input type="text" value="Cornetto/Sfoglia"/>
1° Spuntino	<input type="text" value="Birra"/>	<input type="text" value="Yogurt"/>	<input type="text" value="Biscotti/grissini"/>
Pranzo	<input type="text" value="Birra"/>	<input type="text" value="Contorno"/>	<input type="text" value="Insalata"/>
2° Spuntino	<input type="text" value="Acqua"/>	<input type="text" value="Frutta/frullato"/>	<input type="text" value="Cracker"/>
Cena	<input type="text" value="Vino"/>	<input type="text" value="Insalata"/>	<input type="text" value="Contorno"/>

Salva

12.7 Nuova dieta



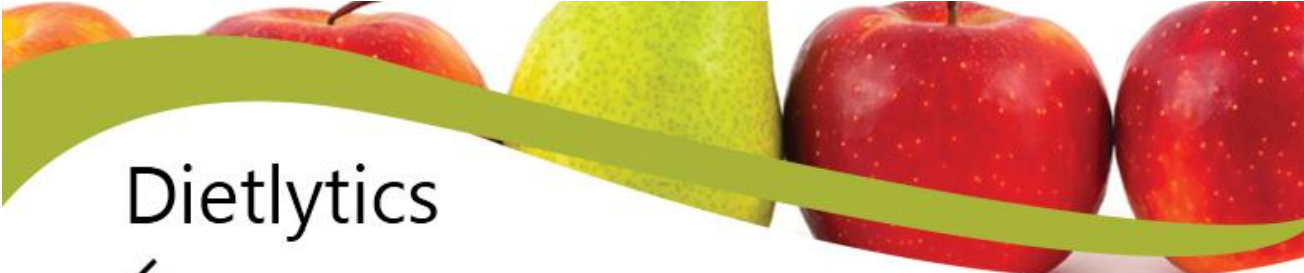
Tipi Allergie/Intolleranze ▼

Tipo di Dieta ▼

Stile di Vita adottato ▼

Inizia Nuova Dieta

12.8 Ricerca cibo



Dietlytics

←

Inserisci Cibo

Inserisci quantità in gr.

Cerca

Nome: PANE AL LATTE KCAL: 347.0 KJ: 1452.0 proteine: 8.6 colesterolo: 11.0 fibre: 3.1 alcool: 0.0 ferro: 1.7

Nome: PANE ALLE OLIVE KCAL: 286.0 KJ: 1197.0 proteine: 7.0 colesterolo: 0.0 fibre: 3.8 alcool: 0.0 ferro: 1.8

Nome: PANE COMUNE, pezzatura da g KCAL: 279.0 KJ: 1167.0 proteine: 8.1 colesterolo: 0.0 fibre: 2.8 alcool: 0.0 ferro: 1.4

Nome: PANE CONDITO KCAL: 284.0 KJ: 1188.0 proteine: 13.3 colesterolo: 7.0 fibre: 7.3 alcool: 0.0 ferro: 1.4

Nome: PANE DI GRANO DURO KCAL: 245.0 KJ: 1025.0 proteine: 10.1 colesterolo: 0.0 fibre: 9.8 alcool: 0.0 ferro: 2.8

Nome: PANE DI GRANO E SEGALE KCAL: 226.0 KJ: 946.0 proteine: 6.2 colesterolo: 0.0 fibre: 4.6 alcool: 0.0 ferro: 1.7

Nome: PANE DI SEGALE KCAL: 219.0 KJ: 916.0 proteine: 8.3 colesterolo: 0.0 fibre: 5.8 alcool: 0.0 ferro: 2.5

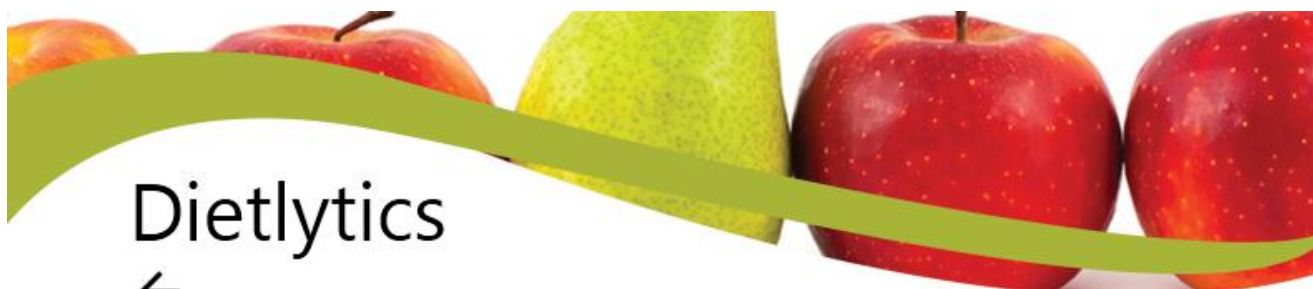
Nome: PANE DI SEGALE BISCOTTATO KCAL: 284.0 KJ: 1188.0 proteine: 9.4 colesterolo: 0.0 fibre: 6.4 alcool: 0.0 ferro: 2.2

Nome: PANE DI SOIA KCAL: 276.0 KJ: 1155.0 proteine: 12.0 colesterolo: 0.0 fibre: 4.5 alcool: 0.0 ferro: 2.2

Nome: PANE GRATTUGIATO KCAL: 354.0 KJ: 1481.0 proteine: 11.6 colesterolo: 0.0 fibre: 3.4 alcool: 0.0 ferro: 2.8

Nome: PANE INTEGRALE KCAL: 243.0 KJ: 1017.0 proteine: 7.5 colesterolo: 0.0 fibre: 5.7 alcool: 0.0 ferro: 2.5

12.9 Storico



▼ 17-12-2015	10-11-2015
▼ 10-11-2015	DIETA PER: zack
▼ 21-10-2015	COLAZIONE:
	Latte 418.2gr
	marmellata prugne 41.82gr o altre marmellate146.37001gr
	Fetta di Torta o Crostata 62.730003gr o Fette Biscottate Integrali 41.82gr o cerea
	Caffè o Tè FACOLTATIVI
	PRIMOSPUNTINO:
	Yogurt 271.83002gr
	Albicocche 209.1gr o banana 41.82gr
	PRANZO:
	Pasta 209.1gr
	olio 5gr
	Parmigiano 209.1gr o Bresaola 200gr
	Tonno 104.55gr o Vitello 125.46001gr o Pollo 167.28gr
	Spinaci 418.2gr o Broccoli 418.2gr
	SECONDOSPUNTINO:
	Mela 418.2gr o Banana 146.37001gr
	CFNA