Università Politecnica delle Marche

Facoltà di Ingegneria

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Tesi di Laurea

Progettazione e implementazione di un modello Java per la rappresentazione del possesso di libri da parte di un insieme di biblioteche

Design and implementation of a Java model for representing the book ownership of a set of libraries

Relatore

Prof. Domenico Ursino

Candidato

Luca Renzi

Anno Accademico 2018-2019

Indice

Elenco delle figure

Elenco dei listati

Introduzione

In questo ultimo periodo di evoluzione tecnologica e digitale, i dati che viaggiano sul web o, semplicemente, all'interno di calcolatori e sistemi hardware prioritari, sono aumentati in maniera esponenziale. Di conseguenza, si è visto il bisogno di trovare metodi e sistemi che permettessero, in primo luogo, di *digitalizzare* questi dati e, successivamente, di memorizzarli e gestirli in maniera appropriata, in base al loro ambiente di competenza.

Questa necessità ha portato gli sviluppatori e i programmatori a cercare di creare sistemi sempre diversi e nuovi, al fine di riuscire a gestire dati, anche in grandi quantità, senza compromettere l'efficienza e migliorandone sempre le funzionalità. Per fare degli esempi, basta pensare alla digitalizzazione delle anagrafiche di un comune di una città, oppure dei documenti medici di un ospedale, quindi cartelle cliniche, liste dei pazienti ricoverati, liste delle varie strutture pubbliche e private, e così via.

Potremmo continuare a fare moltissimi esempi; tuttavia, in questo elaborato, vogliamo soffermarci su un caso particolare, scelto per dimostrare le potenzialità e i vantaggi che si trarrebbero, nella realizzazione di un sistema gestionale; prenderemo, quindi, in esame un sistema per la gestione di un insieme di biblioteche.

Queste ultime, data la loro strutturazione, forniscono un ottimo esempio per riuscire a capire quanto un buon sistema software riesca a rendere molto più facile la gestione di una o più di esse e dei loro contenuti. Parlando di biblioteche, infatti, ci riferiamo a quelle strutture che contengono libri e testi, alcuni dei quali anche molto antichi e di una certa rilevanza, storica e non, che raccontano la storia del nostro mondo e di moltissime discipline studiate dall'uomo.

Prima di questi sistemi, le biblioteche venivano gestite grazie al lavoro costante, e molto dispendioso, di bibliotecari che dedicavano intere giornate a creare degli elenchi, contenenti delle informazioni sui volumi e sui testi che la biblioteca stessa possedeva. Non è difficile immaginare la mole di lavoro che queste persone dovevano gestire, soprattutto nelle biblioteche delle città più grandi, dove si arrivavano ad avere interi schedari pieni di cartelle e fogli, che permettevano di conoscere, se consultati, informazioni sui testi contenuti all'interno della struttura. Allo stesso tempo, però, aumentando il numero dei volumi, aumentavano anche gli errori che si potevano commettere che, in alcuni casi, hanno portato alla perdita di interi testi e

volumi, dimenticati all'interno delle strutture e ritrovati soltanto anni dopo, grazie a lavori di restauro o di riorganizzazione.

L'utilizzo e l'implementazione di sistemi software per gestire questo tipo di dati, hanno contribuito a migliorare sensibilmente la gestione, e anche l'aggiornamento, dei dati, grazie alla semplicità con la quale è possibile reperire informazioni, prima suddivise in diversi posti, mentre ora memorizzate in un unico luogo, sia esso un database online o un supporto di memoria fisico, posto all'interno di un server dedicato, e permetterne la modifica o la consultazione in maniera facile e veloce. Ad esempio, prima non era possibile, se non con un lavoro che consumava molto tempo e spazio, avere elenchi ordinati dei volumi secondo vari criteri, mentre, tramite implementazione software, è possibile avere più di uno di questi elenchi, scegliendo il metodo di ordinamento desiderato; questo, poi, si ripercuote anche nelle ricerche di determinati volumi, per le quali si impiegava molto tempo, mentre ora, le stesse forniscono le informazioni desiderate in pochi secondi.

Un altro esempio, è l'aggiornamento di questi elenchi di volumi, che, nel migliore dei casi, comportava la semplice aggiunta di una nuova cartella o di nuovi fogli, con i dati dei nuovi volumi, che dovevano poi essere catalogati opportunamente; con l'aiuto di un software, invece, in pochi secondi è possibile aggiungere o modificare informazioni su vecchi e nuovi volumi, con il semplice utilizzo di un'interfaccia, lasciando il lavoro di ordinamento e memorizzazione interamente al software gestionale, abbassando notevolmente i tempi richiesti, migliorando l'efficienza e il metodo di lavoro.

In questa tesi viene descritto il lavoro svolto per la creazione di un sistema del tipo descritto, ovvero, un sistema software per la gestione dei possessi di libri, da parte di un insieme di biblioteche, tramite il linguaggio Java. Questo linguaggio è ormai un caposaldo della programmazione mondiale da più di vent'anni; il progetto ha avuto modo di testare le capacità dell'autore nel riuscire ad utilizzare uno strumento a lui non conosciuto, per l'implementazione di funzionalità, più e meno complesse, che hanno dimostrato alcune difficoltà.

Il risultato finale mostra come, anche nei progetti che sembrano più semplici, a volte, si nascondano difficoltà, dovute a svariati aspetti della progettazione, a partire dalla scelta del linguaggio, fino ad arrivare alla complessità delle funzionalità richieste. Allo stesso tempo, però, mostra, anche, come è possibile, col giusto metodo ed i giusti strumenti, riuscire a realizzare un sistema simile, avvalendosi di informazioni facilmente ottenibili da chiunque al giorno d'oggi, adattandolo a uno o più ambiti diversi.

Verranno, di seguito, mostrate e spiegate tutte le attività che sono state effettuate, che vanno dallo studio del sistema in esame, alla scelta degli strumenti per realizzarlo, alla progettazione di un modello e, infine, alla sua implementazione. Verranno, anche, approfonditi alcuni punti che hanno generato difficoltà, sia a livello di progettazione che implementativo, mostrando le soluzioni trovate per risolverli.

L'elaborato è suddiviso in cinque capitoli, ciascuno dei quali tratta argomenti differenti, che riassumiamo di seguito:

- Nel Capitolo 1 viene introdotto il concetto di programmazione ad oggetti ed il linguaggio utilizzato per la realizzazione del sistema, ovvero Java.

- Nel Capitolo 2 vengono, invece, introdotti i modelli utilizzati, concentrandosi maggiormente sul linguaggio UML, utilizzato per la progettazione di questi ultimi.
- Nel Capitolo 3 vengono descritte l'analisi dei requisiti effettuata e la conseguente progettazione dei diagrammi, che sono stati utilizzati per modellare il sistema.
- Nel Capitolo 4 vedremo l'implementazione di tali diagrammi, mostrando il codice derivato da essi e fornendo, alla fine, un manuale utente per capire il funzionamento del sistema.
- Nel Capitolo 5, infine, troveremo le conclusioni sul lavoro svolto, in merito alle conoscenze ottenute e all'analisi dei punti di forza e di debolezza del sistema, con i possibili miglioramenti futuri.

1

Il linguaggio Java

Nell'ambito informatico, un linguaggio di programmazione è un linguaggio formale (come, ad esempio, quello matematico) definito da determinate istruzioni, che hanno un proprio lessico, semantica e sintassi, grazie alle quali è possibile scrivere un programma che permetta di manipolare dei dati in input e produrre dei risultati in output. Nel corso della storia sono stati scritti e sviluppati molteplici linguaggi di programmazione, ciascuno dei quali con le proprie regole e caratteristiche, che ne hanno permesso la diffusione e l'utilizzo in moltissimi ambiti, tecnologici e non. In questo primo capitolo, ci soffermeremo su un particolare linguaggio e sul tipo di programmazione utilizzata da quest'ultimo, ovvero il linguaggio Java e la Programmazione Orientata agli Oggetti, descrivendo le principali caratteristiche e punti di forza che lo rendono, ancora oggi, uno dei linguaggi più usati per la creazione di programmi di vario genere.

1.1 La Programmazione Orientata agli Oggetti

La Programmazione Orientata agli Oggetti (Object Oriented Programming, OOP) è uno dei diversi tipi di paradigmi di programmazione che permette di organizzare il programma secondo oggetti, ovvero dei "contenitori" per i dati, che interagiscono tra di loro grazie a determinate interfacce o modalità definite.

Questi oggetti sono delle istanze di modelli astratti chiamati classi, porzioni di codice che racchiudono le dichiarazioni dei *dati* e le *procedure* per utilizzarli.

Una volta istanziato l'oggetto, questi ultimi vengono chiamati *attributi* e *metodi* dell'oggetto. Gli oggetti possono essere visti come entità che scambiano messaggi per comunicare tra loro e con il codice, o per eseguire istruzioni specifiche, dando così la possibilità di non essere bloccati a pensare un programma come una serie lineare di operazioni e istruzioni da seguire, ma bensì come lo scambio di messaggi da parte di oggetti diversi, che sviluppano istruzioni e interazioni con il codice sorgente.

Lo scambio di messaggi tra gli oggetti avviene attraverso le chiamate ai metodi degli stessi da parte del programma che li gestisce su richiesta.
Per cercare di esemplificare il concetto di Classe e Oggetto, possiamo pensare molto banalmente alla Classe come un insieme matematico con determinate caratteristi-

che, dove gli elementi sono gli Oggetti che si vanno ad istanziare di volta in volta, e che ereditano le caratteristiche dell'insieme di appartenenza.

1.2 Differenze con la Programmazione Orientata ai Processi

Prima della Programmazione Orientata agli Oggetti, il paradigma di programmazione più utilizzato per costruire un programma era la *Programmazione Orientata ai Processi* o *Programmazione Procedurale*, utilizzata ampiamente da linguaggi come il C o il Pascal. La Programmazione Orientata ai Processi, a differenza di quella ad oggetti, organizza il programma attorno al suo codice, che viene utilizzato per accedere e manipolare i dati. Un programma, in questo modo, può essere visto come una serie di passaggi lineari opportunamente codificati, che agiscono direttamente sui dati, senza il bisogno di creare modelli astratti o interfacce, come nel caso delle classi e degli oggetti.
Sebbene, quindi, la programmazione procedurale risulti forse più immediata e intuitiva rispetto a quella ad oggetti, aumentando la complessità dei programmi da codificare, possiamo notare dove i suoi limiti iniziano a mostrarsi.

La programmazione ad oggetti si sviluppa, in parte, con lo scopo di superare alcuni dei limiti imposti dalla programmazione procedurale. Con l'aumentare delle dimensioni e delle generalizzazioni dei programmi, infatti, la programmazione ad oggetti offre molta più semplicità e flessibilità, rispetto a quella procedurale. Basti pensare che, nel caso di ampliamento di un programma qualsiasi, nella programmazione ad oggetti basta aggiungere una classe, costruita ad hoc per quello scopo, senza intaccare il resto del codice, mentre nel caso della programmazione procedurale questo non è possibile, in quanto bisognerebbe ricontrollare il codice e fare le opportune aggiunte e modifiche a seconda dei casi, per implementare la nuova funzionalità. Qualora le dimensioni del programma e del codice sorgente tendono ad aumentare, anche la gestione e le correzioni ne risentono, portando il programmatore a dover spulciare manualmente tra tutto il codice per trovare quello che cerca, piuttosto che accedere al codice della classe interessata e fare degli aggiustamenti.

I *Tre Principi dell'OOP*, che vedremo di seguito, ci permettono di capire quanto, grazie ad essi, un programma ad oggetti sia più sicuro, robusto, modulabile e scalabile rispetto al suo corrispettivo nella programmazione procedurale, ragione per cui, col passare del tempo, si è preferito optare per la programmazione ad oggetti, lasciando quella procedurale a determinati ambiti e ad uso didattico per chi si approccia alla programmazione.

1.3 Principi della Programmazione ad Oggetti

Un linguaggio orientato ad oggetti, per essere tale, deve offrire dei meccanismi che permettano di implementare il modello ad oggetti.

Questi meccanismi fondamentali, spesso chiamati anche Principi, data la loro importanza, sono tre:

- Incapsulamento;
- Ereditarietà;
- Polimorfismo;

Questi tre principi devono essere garantiti per permetterci di definire un linguaggio orientato ad oggetti come tale. Diamo ora una definizione e descrizione per ciascuno di essi.

1.3.1 Incapsulamento

L'incapsulamento è un meccanismo che permette di realizzare due concetti:

- limitare l'accesso diretto ai singoli oggetti;
- collegare il codice e i dati che manipola all'interno della singola classe o di una sua istanza.

Il primo si pone l'obiettivo di evitare che del codice non appartenente all'oggetto possa accedervi in maniera poco controllata, o del tutto incontrollata, solitamente attraverso l'uso di interfacce ben definite. Il secondo, invece, può essere visto come un modo per creare modularità tra le varie classi ed oggetti, semplificando la gestione e la correzione di queste ultime. In questo modo si rende più semplice l'applicazione del primo concetto, interfacciando più facilmente un'unica porzione di codice. Un esempio molto pratico è quello della trasmissione di un'automobile, che racchiude al proprio interno molteplici informazioni sul veicolo, con le quali il guidatore può interagire soltanto attraverso la leva del cambio, che funge da interfaccia per l'accesso ai dati e alle procedure, e che non va a intaccare altre parti dell'automobile, come, ad esempio, lo sterzo o i tergicristalli. Nella Figura 1.1 viene schematizzato il concetto di incapsulamento.

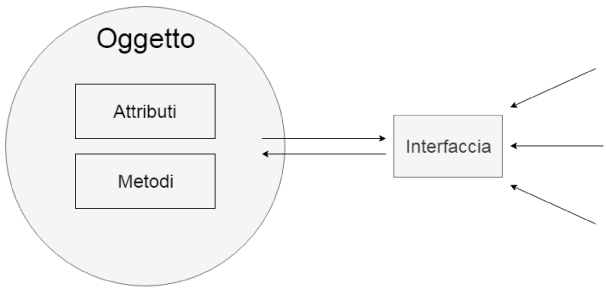


Figura 1.1. Concetto di incapsulamento

L'incapsulamento è forse il principio più forte e importante dei tre, in quanto garantisce la sicurezza sui dati e sulle procedure che li utilizzano, controllandone strettamente accesso e utilizzo, e permettendo di influire fortemente anche sulla scalabilità dei programmi e/o degli applicativi che vi accedono, grazie al riutilizzo di porzioni di codice definite precedentemente.

1.3.2 Ereditarietà

L'ereditarietà è un concetto che mette in relazione due classi o due oggetti. Più nello specifico, essa permette di creare o definire oggetti e/o classi che ereditino le proprietà di altri oggetti e/o classi dalle quali discendono.

La logica segue quella dell'eredità gerarchica, ovvero, definendo un *oggetto figlio*, questo erediterà e potrà utilizzare tutti gli attributi e i metodi dell'*oggetto padre* da cui è stato generato, oltre a quelli definiti soltanto in esso. Il discorso non vale solo in maniera singola ma anche multipla, ovvero un oggetto padre può avere più oggetti figli.
Un esempio che ci fa capire meglio questo concetto è, sicuramente, quello delle classificazioni degli animali: nel regno animale, ad esempio, abbiamo i mammiferi, i rettili o i pesci che, a loro volta, si dividono in categorie diverse con le loro caratteristiche peculiari e specifiche, ma mantengono quelle della classe da cui discendono. In un programma questo può essere utilizzato per andare a definire oggetti particolari e con metodi specifici, che vanno ad estendere i metodi e gli attributi dell'oggetto padre da cui derivano.
Nella Figura 1.2 viene schematizzato il concetto di ereditarietà.

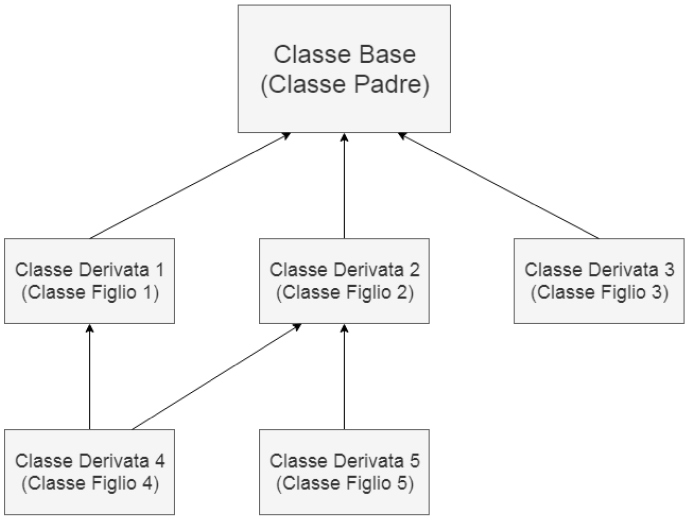


Figura 1.2. Concetto di ereditarietà

Il discorso fatto per gli oggetti si può facilmente estendere alle classi per analogia, creando il concetto di SuperClasse e SottoClasse che vedremo in seguito (Sezione 1.5.2).

1.3.3 Polimorfismo

Il polimorfismo, parola che deriva dal greco e che significa "di molte forme", è un concetto che può essere espresso usando la frase "un'interfaccia, più metodi".

Il polimorfismo, infatti, permette di creare un interfaccia generica che sarà, poi, utilizzata da un insieme di oggetti aventi tutti caratteristiche simili.

Questo aiuta il programma ed il programmatore a ridurre la complessità del codice ed a riutilizzare in modo appropriato e automatico (senza selezione manuale) dei metodi già esistenti ma di valenza "generale". Per fare un esempio possiamo pensare ad un metodo per disegnare una figura di un oggetto base; tale metodo può essere, poi, definito negli oggetti derivati da esso, che possono essere figure specifiche, come il rettangolo, il triangolo o il quadrato, ciascuno col proprio metodo specifico per quel caso.
Grazie al polimorfismo, non occorre richiamare il metodo specifico manualmente, in quanto questo sarà riconosciuto e richiamato automaticamente a seconda dell'oggetto che viene utilizzato in quel momento. In questo modo, si rende il codice flessibile e meno complesso, soprattutto in termini di gestione e ampliamento.
Nella Figura 1.3 viene esemplificato il concetto di polimorfismo.

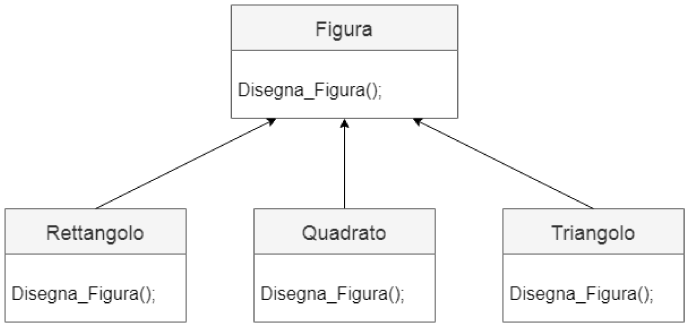


Figura 1.3. Esempio di polimorfismo

Questo tipo di polimorfismo descritto, chiamato *Polimorfismo per Inclusione*, è un concetto utilizzato nella programmazione ad oggetti; nella programmazione generica, invece, si ha soltanto il *Polimorfismo parametrico*, dove il codice del programma può ricevere un tipo come parametro invece che conoscerlo a priori.

1.4 I Principi di Java

Tra i linguaggi ad oggetti, Java è, sicuramente, uno dei più conosciuti e utilizzati ancora oggi; esso è anche il linguaggio scelto per implementare il modello che verrà descritto successivamente.
Java è un linguaggio *ad oggetti, ad alto livello* e a *tipizzazione statica*, che utilizza l'omonima piattaforma e che permette di creare programmi che siano indipendenti dall'architettura hardware su cui vengono eseguiti.

Java è un linguaggio *compilato ed interpretato*; la compilazione avviene in *bytecode*, ovvero una serie di istruzioni appositamente ottimizzate per la compilazione di programmi in Java, che viene, poi, interpretato dalla *Java Virtual Machine (JVM)*, un sistema runtime specifico, che permette di evitare alcuni dei problemi causati

dalla compilazione di codice eseguibile, soprattutto per programmi web-based. Per questo motivo, Java viene, anche, definito come linguaggio *semi-interpretato*.

Java è stato creato con un principio fondamentale alla sua base, espresso dalla sigla *"WORA"*, che significa "Write Once, Run Anywhere", ovvero "scrivi una volta, esegui ovunque". Tale principio ha portato gli sviluppatori a effettuare la compilazione in bytecode, interpretabile da qualsiasi architettura che supporti JVM.

Oltre a questo principio, ci sono altri fattori di cui il team di sviluppo ha tenuto conto nella realizzazione, che possono essere riassunti in alcune "parole chiave":

- semplice, sicuro e portabile;
- a oggetti;
- robusto;
- multithread;
- neutro rispetto l'architettura;
- interpretato e ad alte prestazioni;
- distribuito;
- dinamico.

Alcune di queste parole sono già state descritte o sono di immediata comprensione, per le altre segue una breve descrizione.

Semplice, sicuro e portabile implica che il linguaggio è stato progettato per essere facile da imparare da chiunque abbia già delle basi di programmazione; il linguaggio è sicuro in quanto limitando l'esecuzione all'ambiente Java, gli applicativi o programmi saranno eseguiti in ambiente controllato; infine, il linguaggio è portabile grazie al bytecode, che permette di essere seguito da tutte le architetture.

Ad oggetti, come abbiamo visto, garantisce tutti i benefici della programmazione ad oggetti.

Robusto perché va ad eliminare molte possibilità di errore da parte del programmatore, evidenziando errori nelle prime fasi di programmazione em allo stesso tempo, liberando il programmatore da errori legati all'allocazione delle risorse in memoria.

Multithread, in quanto è stato progettato con l'intento di riuscire a eseguire più operazioni simultaneamente, sincronizzandole accuratamente, requisito ormai basilare per molti programmi sul web o interattivi.

Neutro rispetto all'architettura, come già detto, per riuscire a creare un codice che fosse interpretabile e compilabile a discapito dell'hardware per cui veniva scritto.

Interpretato e ad Alte prestazioni, grazie al bytecode e alla JVM, già citate.

Distribuito, per la possibilità di Java di gestire protocolli TCP/IP su Internet.

Dinamico, infine, poiché le informazioni di tipo runtime, permettono di verificare e risolvere alcuni aspetti del programma a tempo di esecuzione, aggiornando il bytecode anche in esecuzione.

1.5 Principi dell' OOP in Java

Come abbiamo detto più volte, Java è un linguaggio di programmazione ad oggetti e come tale, quindi, è dotato di tutti i vantaggi e di tutte le caratteristiche di questo tipo di linguaggio. In particolare, i tre principi dell'OOP (Sezione 1.3), in Java,

sono definiti da particolari concetti specifici del linguaggio. Questi concetti sono, rispettivamente:

- La Classe, per l'Incapsulamento;
- Le Sottoclassi, per l'Ereditarietà;
- L'Overload e Override, per il Polimorfismo.

1.5.1 Classe

La definizione di Classe in Java non è molto diversa da quella che abbiamo dato per la Programmazione ad Oggetti. Una Classe è una struttura definita da codice e dati, ereditati dagli oggetti che discendono da essa. Anche in questo caso, gli oggetti sono visti come istanze della classe, che possiedono gli stessi dati e le stesse procedure.

I dati e il codice sono definiti come *membri della classe*; in particolare, i dati vengono chiamati *variabili membro o di istanza*, mentre il codice prende il nome di *metodo*. Come già discusso, i metodi forniscono un modo per utilizzare le variabili membro.

Una cosa da notare è che, all'interno di una Classe, variabili o metodi possono essere definiti *pubblici o privati*. Una variabile pubblica è una variabile alla quale anche utenti o codice esterno alla classe possono accedere; se, invece, la variabile è privata può essere utilizzata soltanto dai metodi della classe stessa. Per i metodi pubblici e privati vale un discorso analogo.

Per definizione, variabili e metodi delle Classi in Java sono sempre definiti come *privati*, in modo da garantire sicurezza sia dei dati che del codice.

1.5.2 Sottoclasse

Il concetto di Sottoclasse è essenzialmente derivato dalla definizione di Ereditarietà che abbiamo dato precedentemente (Sezione 1.3.2).
Una Sottoclasse è una *classe derivata* da una classe base, detta *Superclasse*, della quale eredita attributi e metodi, che poi estende con i propri specifici.
Questo concetto è molto legato a quello di Incapsulamento, in quanto i dati e i metodi incapsulati in una Superclasse vengono ereditati, e nuovamente incapsulati, nelle sue Sottoclassi, consentendo una evoluzione lineare, e non esponenziale, degli oggetti.

1.5.3 Overload e Override

Definendo il Polimorfismo (Sezione 1.3.3), si è parlato di *polimorfismo per inclusione*, spiegando come, grazie ad esso, si semplifichi il lavoro del programmatore, il quale non deve più necessariamente effettuare chiamate specifiche a determinati metodi, in quanto verrà fatto in maniera automatica dal codice. Questa definizione e spiegazione che abbiamo fornito si estende al concetto di *Override* o *Overriding*.

L'*Overload*, invece, è una delle caratteristiche di Java che ci permette di andare a "sovraccaricare" uno o più metodi, con diverse sue varianti, cambiando i parametri passati. Per fare un esempio molto semplice, si pensi ad un metodo che calcoli l'area di una figura; con l'Overload è possibile definire più metodi con lo stesso nome, ma

che differiscano solamente per il numero di dati che passati per effettuare il calcolo, ad esempio soltanto il lato per l'area di un quadrato, mentre la base e l'altezza, per l'area di un triangolo.

Non è difficile capire come questo sia un forte mezzo per rendere il codice più dinamico (e anche "pulito" se vogliamo), stando, però, attenti a non cadere in casi di ambiguità dovuti a parametri coincidenti su più metodi.

1.6 Piattaforma Java

Tutto quello che è stato scritto finora riguardo Java, quindi la sua definizione, i principi su cui si fonda, le parole chiave che abbiamo descritto e le sue caratteristiche, contribuiscono a formare quello che è l'ambiente di sviluppo dei programmi in Java, ovvero la sua Piattaforma.

La *Piattaforma Java* altro non è che l'intero ambiente di sviluppo, nel quale vengono eseguiti e compilati i programmi scritti in questo linguaggio. Può essere divisa, a grandi linee, in 2 macro parti:

- la Java Virtual Machine (JVM);
- le API (Application Programming Interface).

La JVM, come abbiamo visto, è la parte che si occupa di eseguire e interpretare il codice del programma, opportunamente compilato in *bytecode*. Le API, invece, non sono altro che delle librerie, che contengono delle istruzioni o delle componenti specifiche per determinati compiti che il programma potrebbe trovarsi a svolgere. Grazie a questa piattaforma, il programmatore è in grado di scrivere un codice in sicurezza, con molteplici possibilità di ampliamento e con la consapevolezza che tale programma sarà, poi, eseguibile a prescindere del sistema fisico sul quale verrà eseguito.
Oltre a questo, la piattaforma offre, anche, altre due importanti funzionalità, ovvero la creazione e utilizzo di *Applet Java* e *Servlet Java*.

Le Applet sono dei programmi "speciali", realizzati appositamente per essere eseguiti sul web da un qualunque browser che abbia compatibilità con Java. Sono, solitamente, di piccole dimensioni e sono eseguibili su richiesta; quindi non c'è la necessità, per un utente che naviga su un sito o una pagina con una o più Applet, di doverle necessariamente eseguire e appesantire o rallentare la pagina stessa e, di conseguenza, la navigazione. Vengono usate maggiormente per mostrare dati recuperati da un server o per gestire operazioni semplici in maniera locale, senza sfruttare risorse del server (ad esempio, un'operazione di login oppure dei semplici calcoli matematici).

Nel momento della loro creazione, le Applet rappresentarono una grande innovazione nella programmazione web, in quanto davano la possibilità di utilizzare le potenzialità degli oggetti su una pagina web, a richiesta e in maniera dinamica, in modo da non appesantire costantemente lo scambio di dati tra client e server, ma limitandolo ad una chiamata, che poi ne permetteva l'avvio direttamente sul client stesso. In questo modo venivano garantiti la sicurezza dei dati e l'indipendenza dall'architettura.

Le Servlet, invece, hanno rappresentato l'altra faccia della medaglia: se le Applet erano applicazioni lato client sul web, le Servlet sono degli altrettanto piccoli programmi che vengono eseguiti lato server, e permettono di estendere, anch'esse in maniera dinamica, le potenzialità di un web server. Essendo programmi in Java, le Servlet vengono compilate in bytecode ed eseguite su JVM, permettendone l'estensione a qualsiasi sistema server che supporti la piattaforma.

Le Servlet vengono principalmente utilizzate per generare delle informazioni, tramite accesso a database o anche soltanto lato server, in maniera dinamica. Tali informazioni vengono, successivamente, passate al client per essere visualizzate e/o utilizzate. Ciò ha permesso a Java di allargare le sue funzionalità ad entrambi i componenti delle connessioni client/server.

2

Modellazione di scenari tramite Java

Dopo aver introdotto, nel capitolo precedente, il linguaggio di programmazione che verrà utilizzato, passiamo ora a vedere un altro concetto che ci aiuterà a strutturare e costruire il nostro progetto, ovvero il concetto di Modello. Daremo, quindi, una breve definizione e vedremo alcuni dei tipi di modelli utilizzati in ambito scientifico e ingegneristico; infine, concluderemo con il linguaggio scelto per la descrizione dei modelli.

2.1 Modello in ambito scientifico

La parola *"Modello"*, nella nostra lingua, può assumere diversi significati a seconda del contesto nella quale viene utilizzata. Nell'ambito scientifico, quello che ci interessa maggiormente, un *modello* è un'insieme di ipotesi, teorie e informazioni che ci permettono di descrivere un sistema o un fenomeno *reale* in maniera oggettiva, per poi studiarne i vari aspetti, comportamenti e anche evoluzioni, in situazioni o ambienti specifici.

Da questa definizione, si potrebbe arrivare a fare un collegamento con il "Metodo Scientifico", enunciato inizialmente da Galileo Galilei, e poi sempre più utilizzato in fisica e in altri ambiti affini, per quanto riguarda lo studio di un esperimento o di un fenomeno attraverso la sperimentazione matematica e fisica dello stesso. Tale studio può essere visto come la creazione di un modello che lo descrive e ne permette l'analisi.

Questo collegamento, pur non essendo completamente accurato, ci dà la possibilità, però, di avere un'idea più "tangibile" e concreta di questo concetto.

2.2 Tipi di modelli

L'ambito scientifico racchiude al proprio interno molte discipline e materie diverse, che variano dalla fisica alla chimica o dalla matematica all'informatica. Per questo motivo, il concetto di modello tende ad essere diverso a seconda del contesto nel quale viene utilizzato, assumendo significati e definizioni più specifiche e corrette.

Tra i principali tipi di modelli che possiamo incontrare, ci sono:

- il modello matematico;
- il modello fisico;
- il modello chimico;
- il modello economico;
- il modello informatico.

Nei prossimi paragrafi, vedremo in dettaglio separatamente il modello matematico e quello informatico, in quanto il primo è alla base della costruzione degli altri, mentre il secondo è il tipo di modello che utilizzeremo per lo sviluppo del nostro progetto. Gli altri modelli verranno comunque brevemente descritti nel seguito di questo paragrafo, per completezza dell'elaborato.

Il modello chimico, o *modello molecolare*, è un insieme di metodi che ci permettono di rappresentare le composizioni e strutture chimiche, a livello molecolare, per facilitarne lo studio e anche la digitalizzazione. Ad esempio, la struttura molecolare dell'acqua (come mostrato in Figura 2.1) o dello zucchero, sono rappresentati da dei modelli chimici.

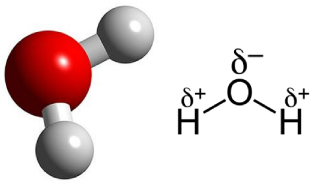


Figura 2.1. Esempio di modello molecolare dell'acqua

Il *modello fisico*, invece, viene usato per rappresentare un fenomeno reale sulla base di ipotesi e leggi fisiche, che ne permettono, poi, lo studio. Grazie ad esso è possibile studiare, modificare e correggere dati o aspetti del fenomeno o dell'oggetto di studio, al fine di migliorarne l'esito o il funzionamento. Possiamo pensare il modello fisico come un "prototipo" più semplice da studiare o con cui interagire, rispetto al suo corrispettivo reale.

Il *modello economico*, infine, sebbene non sia propriamente una disciplina scientifica, permette di semplificare dei processi economici complessi che intercorrono tra variabili economiche e le loro relazioni. Un esempio di modello economico è dato da un modello di spesa o di previsione riguardo una determinata realtà imprenditoriale, sulla base di dati passati e presenti.

2.2.1 Modello matematico

Un *modello matematico* viene utilizzato per descrivere un fenomeno, o dei suoi aspetti, tramite l'utilizzo di formule, equazioni e grafici.
Questa definizione, che si avvicina molto a quella generale della Sezione 2.1, ci permette, inoltre, di affermare che molti dei modelli descritti in questo capitolo (e non solo) si basano in realtà su modelli matematici. Questi ultimi spaziano dal calcolo

delle probabilità alle equazioni, ordinarie e differenziali, permettendone un utilizzo molto ampio e flessibile, soprattutto in ambiti dove si riesce a "matematizzare" fenomeni e studi.

I modelli matematici vengono utilizzati maggiormente per prevedere l'andamento o l'evoluzione di un dato fenomeno, sulla base di dati iniziali forniti e della loro elaborazione, anche grazie all'utilizzo di grafici di vario tipo.
Questo, però, non significa che tali modelli rappresentino in maniera completamente "reale" il fenomeno; essi vengono costruiti sulla realtà, ma con le dovute limitazioni e semplificazioni del caso, che permettono di studiare ed effettuare previsioni sul fenomeno in esame, ma entro suddetti limiti.
Nella costruzione di un modello matematico, infatti, bisogna inizialmente chiedersi quanto di quello che vogliamo studiare sia modellabile e anche in che misura, ovvero se si può modellare interamente una determinata realtà o se occorre effettuare delle semplificazioni. Una volta fatto questo, si formulano equazioni, leggi e sistemi che permettono al modello di prendere forma e di rappresentare il fenomeno; si introducono, poi, i dati iniziali e si studia il modello con modi e metodi opportuni. Ottenute tutte le conclusioni e i dati finali, essi vengono interpretati ed utilizzati per fare le previsioni del caso, prima dell'effettiva verifica reale del fenomeno.

Nella Figura 2.2 viene schematizzata la costruzione di un modello matematico.

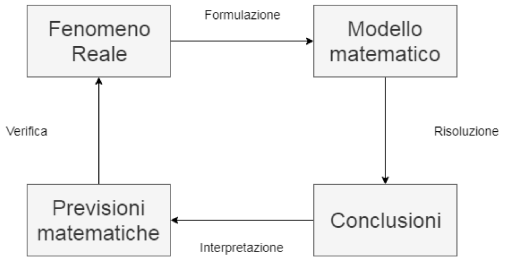


Figura 2.2. Esempio di costruzione di un modello matematico

I modelli matematici, oltre che in base all'ambito di utilizzo, si possono suddividere anche in base al loro comportamento in relazione al tempo ed al tipo di input che ricevono.

Nel primo caso parliamo di modelli *statici e dinamici* in cui abbiamo, rispettivamente, la rappresentazione di un dato fenomeno in un determinato istante temporale e la rappresentazione del comportamento al variare del tempo. Ad esempio, in fisica, possiamo pensare alle leggi della cinematica e della dinamica per la modellazione di un moto qualsiasi, sia esso in un istante di tempo o in un intervallo.

Nell'altro caso, invece, vengono chiamati modelli *deterministici e stocastici*, dove i primi sono caratterizzati da dati in input fissi, che forniscono risultati limitati da questo e da altri vincoli legati al modello; i secondi, invece, vengono caratterizzati da dati variabili in input e forniscono dati in output sulla base delle probabilità che ne conseguono. I modelli stocastici sono, in genere, più complessi di quelli deterministici, in quanto devono tener conto di più fattori e coprire uno spettro di situazioni maggiore, dovuto alla variabilità dei dati.

Quest'ultima suddivisione ci porta, anche, a definire i modelli secondo un'altro concetto, ovvero la linearità[1], ottenendo modelli *lineari e non lineari*, a seconda dei risultati del modello stesso.

Nella Figura 2.3 viene mostrato un esempio di un modello matematico di un motore elettrico.

$$\begin{cases} V_{qs} = R_s I_{qs} + \dfrac{d}{dt}\lambda_{qs} \\ V_{ds} = R_s I_{ds} + \dfrac{d}{dt}\lambda_{ds} \\ V_{0s} = R_s I_{0s} + \dfrac{d}{dt}\lambda_{0s} \end{cases} \qquad \begin{cases} V_{qr} = R'_r I'_{qr} + \dfrac{d}{dt}\lambda'_{qr} + P\omega\lambda'_{dr} \\ V_{qr} = R'_r I'_{dr} + \dfrac{d}{dt}\lambda'_{dr} - P\omega\lambda'_{qr} \\ V_{0s} = R_r 'I'_{0r} + \dfrac{d}{dt}\lambda'_{0r} \end{cases}$$

$$\begin{cases} \lambda'_{qr} = M \cdot I_{qs} + (Ll'r + M) \cdot I'_{qr} \\ \lambda'_{dr} = M \cdot I_{ds} + (Ll'r + M) \cdot I'_{dr} \\ \lambda'_{0r} = Ll'r \cdot I'_{0r} \end{cases} \qquad \begin{cases} \lambda_{qs} = M \cdot I'_{qr} + (Lls + M) \cdot I_{qs} \\ \lambda_{dr} = M \cdot I'_{dr} + (Lls + M) \cdot I'_{ds} \\ \lambda_{0r} = Lls \cdot I_{0s} \end{cases}$$

Figura 2.3. Esempio di un modello matematico di un motore elettrico asincrono

2.3 Modello informatico

Nel mondo dell'informatica la parola "modello" può essere usata per indicare più di un oggetto o concetto.

Pensando ai database, il primo modello che viene in mente è il *modello concettuale o modello E/R*, che viene utilizzato per rappresentare le entità e le relazioni esistenti tra di esse; queste devono caratterizzare la progettazione di una base di dati con un alto livello di astrazione e rappresentazione grafica. Il modello informatico viene utilizzato dai progettisti per "tracciare" le linee guida da seguire nella progettazione, indicando, però, soltanto i concetti e non i dettagli implementativi, dando maggiore potenza al modello, rendendolo più versatile.

Il modello in questione è un modello grafico, che permette un'immediata comprensione per coloro che lo utilizzano e fornisce, inoltre, tutte le informazioni necessarie per progettare le strutture con le quali costruire il database richiesto.

Nella Figura 2.4 si mostra un esempio di un modello E/R di un database.

Pensando alle pagine web, invece, possiamo chiamare "modello" un insieme di campi in un sito, solitamente chiamato *form*, dove l'utente va ad inserire dei dati

[1] Il concetto di linearità qui riportato si rifà maggiormente all'ambito probabilistico e statistico. Tuttavia, come abbiamo detto nei riguardi dei modelli matematici, tale concetto potrebbe essere esteso anche ad ambiti come l'algebra e la logica. Considerato l'ambito generale del progetto e anche la funzione di questo capitolo, si è deciso di evitare di aggiungere anche questo concetto per non appesantire troppo l'elaborato. Si rimanda, quindi, per approfondimenti, ad altri testi o fonti più specifici sull'argomento.

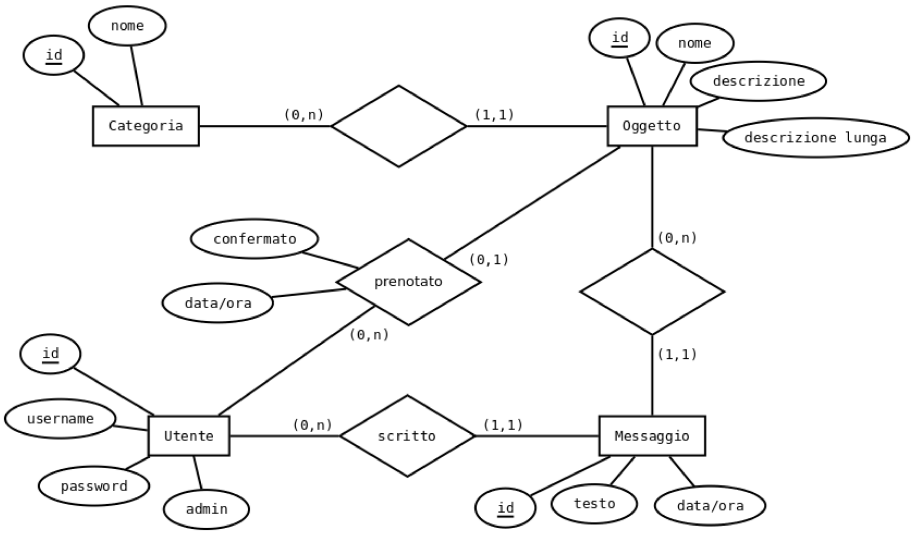


Figura 2.4. Esempio di un modello E/R di un database

per una registrazione o per la modifica di determinate informazioni, o anche, più semplicemente, per il login tramite username e password.

Nelle pagine web, un modello rappresenta una determinata disposizione, già definita, per gli oggetti al proprio interno, di facile personalizzazione in fase di codifica, standardizzabile e utilizzabile su più pagine, anche diverse tra loro. Queste descrizioni ci fanno pensare anche ai fogli di stile, o a file di presentazione, ad esempio del pacchetto Office (fogli di Excel per l'inserimento di dati, presentazioni PowerPoint con un determinato stile grafico, modelli Word da compilare) che rientrano anch'esse in questa definizione.

In questi casi quello che noi chiamiamo "modello" prende il nome di *Template*.

Nella Figura 2.5 viene mostrato un esempio di template web.

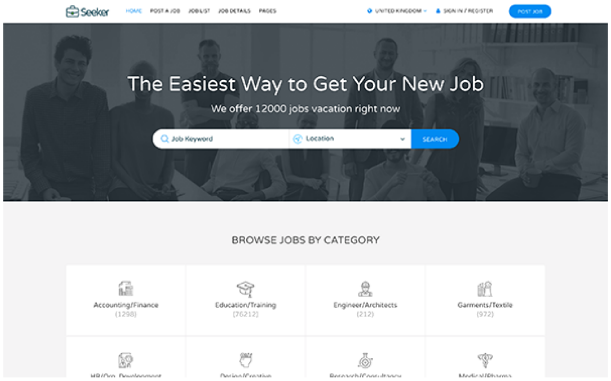


Figura 2.5. Esempio di un template per una pagina HTML per cercare lavoro, acquistabile dal sito wrapbootstrap.com

Oltre che in ambito web, possiamo trovare il termine “template” anche nella programmazione informatica, in C++ e anche in Java, dove viene utilizzato per indicare delle particolari strutture di codice, già definite, che non dipendono dal tipo di dati e che sono riutilizzabili da più programmi. Tali strutture di codice vengono contenute solitamente in una o più librerie che prendono il nome di *Standard Template Library o STL*; queste fanno da contenitore e, se importate o incluse nel programma, permettono l'utilizzo in maniera immediata e semplice.

Un esempio di template in C++ viene mostrato in Figura 2.6.

```
template < typename T > void
funzione (const T & x)
{
  static int num = 10;
  cout << ++num;
  return;
}

int
main ()
{
  funzione < int >(1);      // Stampa a video 11
  cout << endl;
  funzione < int >(3);      // Stampa a video 13
  cout << endl;
  funzione < double >(3.2); // Stampa a video 13
  cout << endl;
  return 0;
}
```

Figura 2.6. Esempio di un Template in C++ di una funzione che aggiunge 10 se richiamata con un valore numerico

I template si suddividono in due tipi, ovvero:

- template per funzioni;
- template per classi.

I *template per funzioni* racchiudono codice che permette al programma di sostituire determinate porzioni di codice con una semplice chiamata ad una funzione, racchiusa nella libreria, che svolge le stesse operazioni, rendendo così il codice meno lungo e più comodo da implementare. Per fare un esempio di template di funzione, possiamo pensare alla funzione matematica che restituisce il valore assoluto di un numero, oppure a quella che confronta due numeri e restituisce il minore o il maggiore, a seconda delle esigenze.

I *template per classi*, invece, sono l'analogo di quelli per funzioni, ma per le classi; essi permettono, infatti, di evitare all'utente di scrivere del codice extra o ridondante per la definizione e l'utilizzo di una classe, contenuta in una libreria o in un file. Basti pensare, ad esempio, al codice della classe di un Array, oppure di uno Stack o Lista e delle funzioni che essi offrono.
In entrambi i casi, la progettazione del codice è la parte più complessa da sviluppare poiché, essendo delle funzioni non legate al tipo di dati che verranno usati, occorre modellarle facendo attenzione a non causare limitazioni involontarie o problemi di

compatibilità. Se, ad esempio, prendiamo la funzione per confrontare il valore tra due numeri, bisogna fare in modo che essa accetti tutti i tipi di dati numerici che si possono utilizzare e non solo, magari, gli interi e i valori a virgola mobile.

2.4 Linguaggi di modellazione

Finora abbiamo parlato sempre di modelli in relazione a oggetti e strutture che ci permettono di rappresentare una realtà, e con la descrizione di alcuni di essi ci siamo accorti che non esiste un unico modo per la loro creazione e il loro sviluppo.

L'esigenza di avere più modi per creare un modello, ha spinto i programmatori e i progettisti a creare dei veri e propri *linguaggi di modellazione*, ovvero linguaggi formali di vario tipo, utilizzati per creare e rappresentare i modelli per diverse realtà. A discapito del tipo, però, ogni linguaggio di modellazione deve offrire all'utente che lo utilizza degli elementi comuni per la creazione di modelli, quali gli *elementi* stessi che permettono di modellare i concetti, la loro *notazione*, sia essa letterale o grafica, e delle *linee guida* per poter creare il modello, usando quello che viene messo a disposizione.

I linguaggi, infine, possono essere suddivisi in macro categorie, a seconda dei loro aspetti caratteristici; possiamo, quindi, avere:

- linguaggi *grafici e testuali*, che utilizzano diagrammi e strutture grafiche specifiche oppure un linguaggio specifico, simile alla programmazione;
- linguaggi *interpretabili e non*, che, quindi, possono essere, o non essere, interpretati da appositi programmi;
- linguaggi *orientati ad oggetto della modellazione*, che vanno a descrivere le strutture o oggetti del programma in diversi modi (ad esempio i linguaggi di data modeling).

Questa classificazione non è rigorosa e non è in grado di rappresentare tutti i tipi; possiamo, infatti, avere linguaggi puramente testuali o grafici, oppure una combinazione dei due, o anche linguaggi non completamente interpretabili.

Concludiamo, infine, facendo degli esempi dei linguaggi più comuni e utilizzati per la modellazione:

- *Diagrammi E/R*, che abbiamo descritto brevemente poco fa (Sezione 2.3).
- *Diagrammi di flusso*, utilizzati per descrivere algoritmi specifici in forma grafica.
- *Business Process Modeling Notation*, uno dei principali linguaggi per processi di business.
- *Unified Modeling Languare, o UML*, un linguaggio ormai divenuto standard per la modellazione di programmi ad oggetti, nonché nostra prima scelta per la modellazione del nostro progetto.
- *Interaction Flow Modeling Language, o IFML*, un linguaggio per descrivere le interazioni di un utente con un software.

2.4.1 Unified Modeling Language (UML)

La sigla *UML*, forma ridotta di *Unified Modeling Language* o Linguaggio di Modellazione Unificato, rappresenta, in informatica, un linguaggio di modellazione utilizzato principalmente per la rappresentazione di programmi o applicativi ad oggetti.

Nasce grazie al consorzio *OMG, Object Management Group*, con l'obiettivo di riuscire a creare uno standard tra i vari metodi precedenti, che contenesse tutte le migliori caratteristiche di ciascuno e si instaurasse come standard unificato, come si legge nel nome. Nella Figura 2.7 vengono mostrati i loghi di OMG e UML.

Figura 2.7. Loghi di OMG e UML

UML è un linguaggio semi-grafico e semi-formale, in quanto i suoi modelli vengono costruiti grazie all'utilizzo di diagrammi grafici, che, però, sono accostati a delle componenti testuali sia libere che formali. Questo insieme ne costituisce un'enorme forza descrittiva, ma, allo stesso tempo, anche facilmente interpretabile, in quanto non occorre una grande conoscenza di concetti di programmazione per comprendere un modello realizzato in questo linguaggio. UML aiuta, inoltre, la scrittura successiva del codice, fornendo una visione di insieme al programmatore e permettendogli di effettuare correzioni o modifiche, qualora ce ne fosse il bisogno, sia durante che dopo la scrittura di quest'ultimo.

Molte persone, per via di queste sue peculiarità, arrivano a definire i modelli UML l'analogo dei "blueprint", ovvero delle planimetrie, nell'ingegneria edile; ciò, insieme ad altri fattori, hanno permesso, inoltre, di accantonare il *Metodo a Cascata*, come metodologia di sviluppo, definendo ed utilizzando il migliore *Visual Modeling*, che permette la visualizzazione del modello attraverso l'uso di diagrammi, opportunamente descritti da UML, ottenendo così, molteplici vantaggi, soprattutto per quanto riguarda il lavoro in team e la suddivisione dei compiti.

Come abbiamo visto, quindi, UML permette di descrivere un sistema software, utilizzando dei diagrammi specifici, in tre aspetti principali:

- il *modello funzionale*, che descrive il comportamento del programma, usando gli *Use Case Diagrams*; questo è l'analogo dell'analisi dei requisiti.
- il *modello ad oggetti*, che descrive la struttura di classi, oggetti e relazioni del programma, usando gli appositi diagrammi; esso rappresenta l'analogo dell'analisi di dominio.
- il *modello dinamico*, infine, che descrive il comportamento e l'evoluzione degli oggetti nel tempo, usando, anche qui, appositi diagrammi che vedremo a breve.

Vediamo, ora, i nove diagrammi principali, utilizzati da UML per modellare i vari aspetti del sistema.

- *Use Case Diagram*, o Diagrammi dei Casi d'Uso; vengono utilizzati per rappresentare tutte le funzioni e scenari d'utilizzo del programma, aiutando sia gli sviluppatori che gli utenti ad avere una visione di quello che il programma dovrà offrire. Un Diagramma dei Casi d'Uso viene rappresentato attraverso l'uso di elementi, che sono *Actor, Use Case* e *System* e relazioni tra i primi due elementi. Gli Actor vengono rappresentati come omini stilizzati e corrispondono alle entità che interagiscono con il programma; essi sono, quindi, utenti finali, hardware o software associati.
  Il termine "Use Case", invece, corrisponde, come dice il nome, ad un caso d'uso o una funzione o comportamento del programma, quando esso viene utilizzato. Esso viene rappresentato da un'ellisse con il relativo nome.
  Il System, infine, è rappresentato da un rettangolo vuoto che racchiude tutti gli elementi del modello che si trovano all'interno del programma preso in esame, evidenziando, anche, altri elementi che sono esterni e possono interagire soltanto in determinati modi.
  Le Relazioni rappresentano delle associazioni tra un Actor e gli Use Case che esso utilizza, e viceversa, e possono legare anche gli Use Case tra loro. In questo ultimo caso, abbiamo due tipi di relazioni, quelle di *Inclusione* e quelle di *Estensione*.
  Le prime vengono rappresentate da una linea tratteggiata con freccia indicante il verso, accostata alla dicitura «`include`»; esse definiscono che la funzione legata da uno dei due Use Case include completamente quella legata all'altro puntato dalla freccia. La freccia, inoltre, indica quale delle due verrà sempre eseguita qualora quella a cui è associata viene richiamata.
  Le seconde, invece, sono rappresentate allo stesso modo, ma con la dicitura «`extend`» al posto della dicitura «`include`» e definiscono che la funzione che "punta" verso uno Use Case può essere estesa alla funzione che viene puntata dalla freccia, qualora ce ne fosse il bisogno.
  Il Diagramma dei Casi d'Uso, infine, viene spesso utilizzato in associazione con gli *Activity Diagrams* o altri diagrammi dinamici. Un esempio di Use Case Diagram viene mostrato nella seguente Figura 2.8.

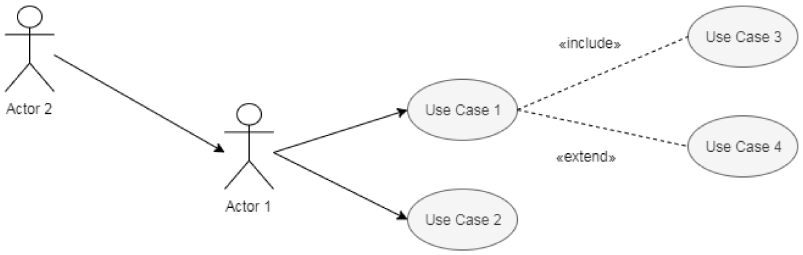


Figura 2.8. Esempio di Use Case Diagram

- *Class Diagram* è il digramma delle classi e, come dice il nome stesso, rappresenta queste ultime e loro eventuali relazioni. È rappresentato da un rettangolo diviso orizzontalmente in tre parti, utilizzate, rispettivamente, per indicare, il nome della classe, la definizione dei suoi attributi e, infine, la dichiarazione dei metodi implementati dalla classe stessa. Le relazioni, invece, vengono rappresentate

attraverso delle linee continue che congiungono due classi legate tra loro; esse possono descrivere, anche, il livello di associazione tra gli elementi (o cardinalità) e la dipendenza tra esse.
Sono i diagrammi più utilizzati nell'ambito di programmi orientati ad oggetti, permettendo di rappresentare qualsiasi livello di astrazione, sia in sede di analisi che di programmazione.

- *Object Diagram* o Diagramma degli Oggetti; anche questo diagramma viene utilizzato maggiormente per programmi orientati ad oggetti; è l'analogo del Class Diagram, ma per gli oggetti, ovvero per le singole istanze della classe.
  La rappresentazione è la stessa che abbiamo descritto per le classi, ma si possono descrivere, anche, utilizzando un singolo rettangolo, con all'interno il nome della classe seguito dal nome del singolo oggetto (ad esempio, "Animale:Cane"). Nella Figura 2.9 vengono mostrati degli esempi di Class Diagram e Object Diagram.

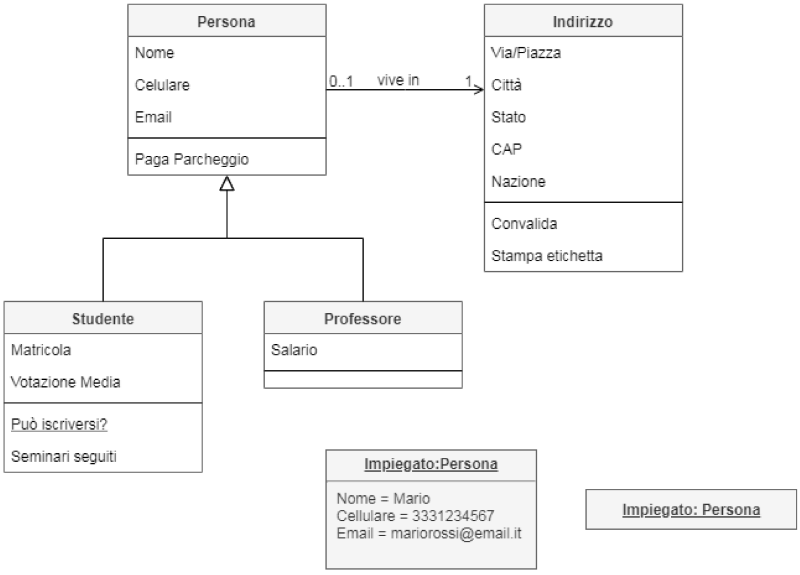


Figura 2.9. Esempi di Class Diagram e Object Diagram

- *Statechart Diagram*, in italiano Diagramma degli Stati, è un diagramma che rappresenta tutti gli stati di un oggetto all'interno del programma. Per capire il concetto di "stato" si pensi ad una lampadina con interruttore: se l'interruttore è su OFF, la lampadina è spenta; se lo si sposta su ON, questa si accende, cambiando il suo stato da Spenta ad Accesa.
  Questo diagramma permette, quindi, di evidenziare il comportamento dei vari oggetti del programma, in relazione ad azioni o altri elementi che esso svolge o con cui interagisce, sia interne che esterne.
  La sua rappresentazione grafica è formata da un rettangolo con angoli smussati, dove vengono inseriti il nome dell'oggetto, le variabili dello stato e, infine, l'azione che viene eseguita qualora l'oggetto entri o esca da tale stato. Questo rettangolo viene, poi, accostato da due frecce. La prima è entrante, parte da un cerchio

nero pieno e indica l'inizio del diagramma. La seconda è uscente, arriva a due cerchi concentrici, con quello interno anch'esso nero pieno, e indica la fine.
Un esempio di Statechart Diagram è mostrato in Figura 2.10.

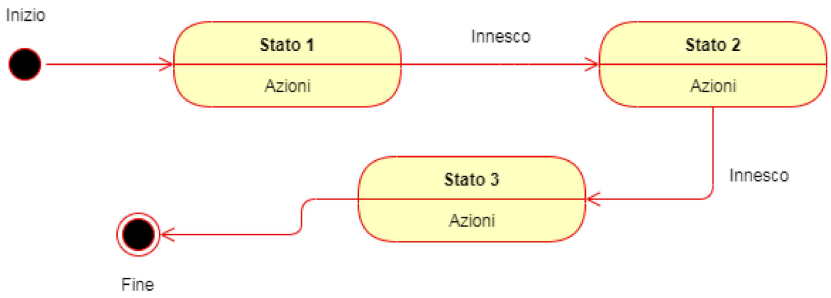


Figura 2.10. Esempio di Statechart Diagram

- *Activity Diagram*, o Diagramma delle Attività, mostra lo svolgimento dei processi, o funzioni, del programma. Gli Activity Diagram ricordano molto i Diagrammi di Flusso, usati per descrivere algoritmi in maniera grafica, e sono costruiti in maniera simile. In questo diagramma, le attività sono rappresentate da rettangoli con angoli smussati, uniti tra loro da delle frecce che ne indicano il flusso e il passaggio da un'attività all'altra. Come negli Statechart Diagram, anche qui un cerchio nero pieno indica l'inizio, mentre due cerchi concentrici, con l'interno nero pieno, indica la fine.
Nella Figura 2.11 viene mostrato un esempio di Activity Diagram.

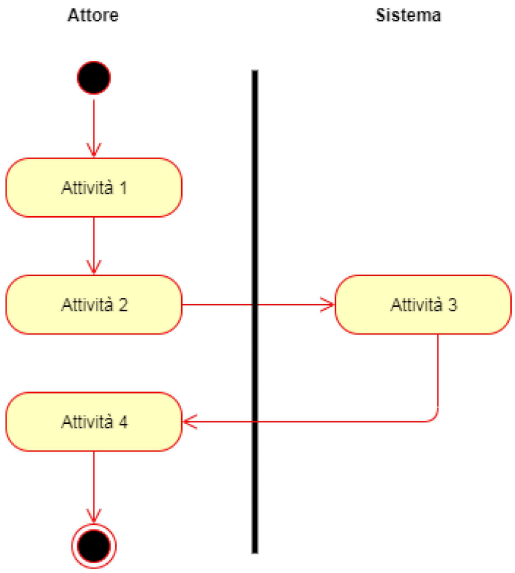


Figura 2.11. Esempio di Activity Diagram

- *Sequence Diagram*, o Diagramma di Sequenza, serve a rappresentare le comunicazioni tra gli oggetti nel tempo. Con il termine "comunicazioni", si intende lo scambio di dati e le azioni che due o più oggetti compiono in uno o più processi; tale scambio e tali azioni vengono rappresentati come messaggi. Solitamente, derivano dagli Activity Diagram, che descrivono le attività, e ne ampliano il dettaglio, concentrandosi sull'evoluzione temporale, e non soltanto funzionale. Gli oggetti sono rappresentati come nei casi precedenti; l'evoluzione temporale è una retta verticale verso il basso che parte dai singoli oggetti; infine, questi ultimi sono rappresentati da frecce che si spostano orizzontalmente tra gli assi temporali dei singoli oggetti.
  Nella Figura 2.12 viene mostrato un esempio di Sequence Diagram.

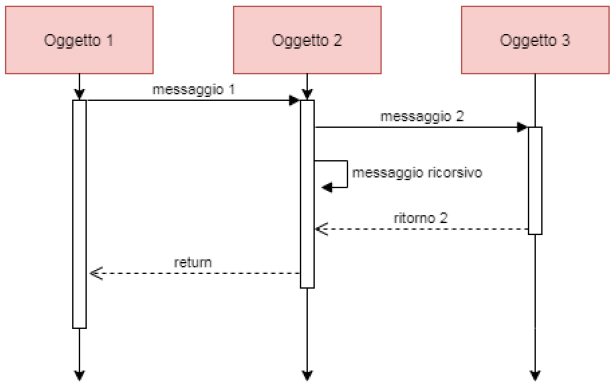


Figura 2.12. Esempio di Sequence Diagram

- *Communication Diagram* o *Collaboration Diagram*, Diagrammi di Comunicazione o Collaborazione, sono molto simili ai Sequence Diagram appena visti, in quanto anche loro rappresentano le comunicazioni tra gli oggetti. Sono anche facilmente scambiabili e convertibili da un tipo all'altro, vista la semantica quasi identica. La differenza principale tra i due, però, è rappresentata dal fatto che i Sequence Diagram modellano in relazione al tempo, mentre i Communication Diagram modellano in relazione allo spazio.
  Sebbene ci sia una evoluzione temporale descritta anche qui (tramite l'uso della numerazione dei messaggi), questo diagramma permette di visualizzare i messaggi scambiati da un oggetto all'altro, rappresentati allo stesso modo dei Sequence Diagram, indicando sia una descrizione di tale messaggio (solitamente l'azione che viene eseguita), sia l'oggetto specifico che esso riceverà.
  Nella Figura 2.13 mostriamo un esempio di Communication Diagram.
- *Component Diagram*, o Diagramma dei Componenti, a differenza di tutti gli altri visti finora, rappresenta i reali componenti software del sistema che si vuole modellare, e non dei concetti, come oggetti o processi. Viene molto spesso utilizzato insieme all'ultimo tipo di diagrammi, il *Deployment Diagram*, che descriveremo a breve. I componenti vengono rappresentati attraverso un rettangolo, con due rettangoli più piccoli attaccati sul lato corto sinistro, dove viene indicato il no-

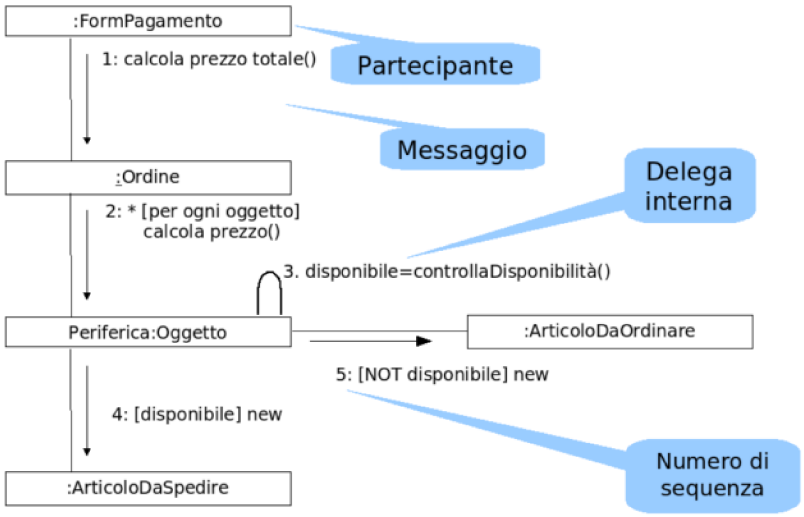


Figura 2.13. Esempio di Comunication Diagram

me del componente e di eventuali package associati. Questi componenti vengono, poi, associati alle proprie interfacce, indicate in due modi diversi ma equivalenti; il primo rappresenta l'interfaccia come un rettangolo, più piccolo, con all'interno il nome, mentre il secondo come un piccolo cerchio a cui viene affiancato il nome. In entrambi i casi, il collegamento viene rappresentato tramite una freccia tratteggiata verso l'interfaccia.

- *Deployment Diagram*, o Diagramma di Dispiegamento, infine, rappresenta i componenti e le risorse hardware del sistema che si sta modellando, insieme alle loro relazioni. Nel diagramma, queste vengono chiamate *nodi* e sono rappresentati da dei cubi, mentre le loro relazioni sono rappresentate da delle semplici linee che li collegano. Come detto precedentemente, vengono spesso usati insieme ai *Component Diagram*.
  Un esempio di Component Diagram e Deployment Diagram viene mostrato in Figura 2.14.

Utilizzati insieme e singolarmente, tutti questi modelli permettono di modellare un qualsiasi sistema, informatico e non, con differenti gradi di precisione e dettaglio. Ad esempio, usando i Class Diagram e gli Use Case Diagram , possiamo rappresentare i vari casi d'uso in relazione alle classi e agli oggetti che abbiamo, oppure, tramite i Component Diagram ed i Deployment Diagram, rappresentare il sistema sul quale un programma dovrà operare.

2.4.2 UML e Java

Come abbiamo visto, UML permette a chiunque, con un minimo di preparazione, di modellare aspetti o realtà in molteplici ambiti e ambienti di sviluppo software. Nel nostro caso, utilizzando Java, UML ci ha aiutato a modellare le classi utilizzate, attraverso l'uso dei Class Diagram, permettendoci una visione d'insieme di tutte

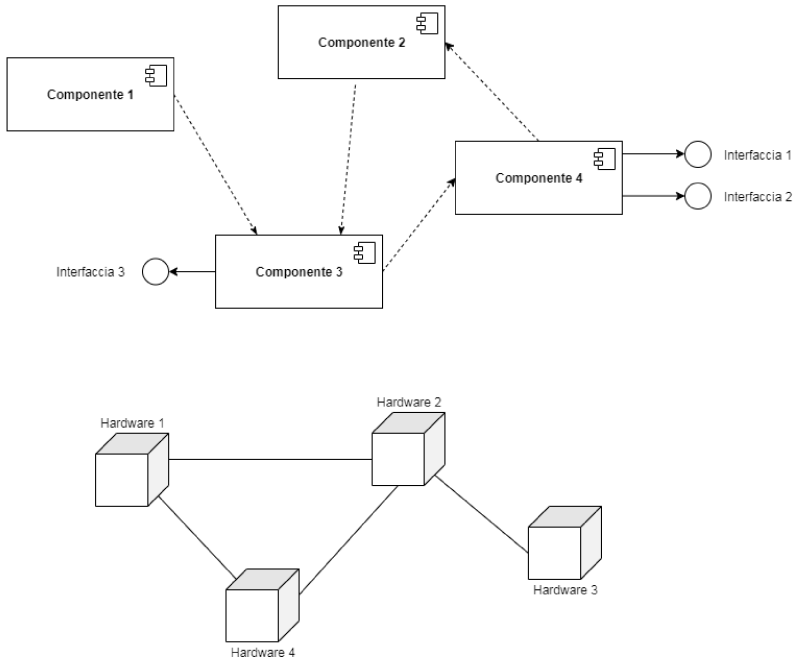


Figura 2.14. Esempi di Component Diagram e Deployment Diagram

le componenti che abbiamo, successivamente, utilizzato. In ambito Java, o della programmazione ad oggetti in generale, ovviamente l'utilizzo di UML non si limita soltanto a questo; si possono, infatti, tracciare modelli delle relazioni tra oggetti, dei casi d'uso, di particolari attività, o anche di stati, qualora ce ne fosse il bisogno.

3

Analisi dei requisiti e Progettazione

Dopo aver introdotto, nei capitoli precedenti, i linguaggi di programmazione e modellazione che useremo, in questo capitolo analizzeremo il nostro progetto e ne tracceremo l'analisi dei requisiti, funzionali e non funzionali; inoltre, mostreremo i diagrammi delle classi e dei casi d'uso utili per modellare i vari elementi presenti, aggiungendo, anche, i diagrammi di flusso utilizzati per implementare le funzioni più complesse.

3.1 Descrizione del Progetto

Il progetto consiste nella costruzione di un sistema basato su Java per la gestione di un insieme di biblioteche.

Il sistema deve presentare all'utente un opportuno menù capace di gestire le seguenti funzionalità:

1. Inserimento di una biblioteca.
2. Rimozione di una biblioteca, dato il suo codice.
3. Modifica del nome di una biblioteca il cui codice è specificato dall'utente.
4. Stampa delle informazioni relative alle varie biblioteche registrate.
5. Ricerca dell'eventuale presenza di una biblioteca, dato il suo codice.
6. Inserimento di un libro.
7. Rimozione di un libro, dato il suo codice.
8. Stampa delle informazioni relative ai vari libri presenti nel sistema.
9. Ricerca dei libri pubblicati in un anno successivo a quello specificato dall'utente.
10. Inserimento dei dati relativi al possesso di un libro da parte di una determinata biblioteca.
11. Rimozione dei dati relativi al possesso di un libro da parte di una biblioteca; a tal fine l'utente dovrà specificare i codici della biblioteca e del libro.
12. Stampa delle informazioni relative ai vari possessi registrati nel sistema.
13. Modifica del numero di copie di un libro in una determinata biblioteca; i codici della biblioteca e del libro vengono specificati dall'utente.
14. Ricerca dei libri pubblicati nell'anno specificato dall'utente che si trovano, con un numero di copie maggiore o uguale a quello specificato dall'utente, in esattamente due biblioteche della città specificata dall'utente.

15. Ricerca dei libri pubblicati in un anno successivo a quello specificato dall'utente che si trovano, esclusivamente, nelle biblioteche di una città specificata dell'utente.
16. Costruzione e visualizzazione della matrice dei possessi; questa è una matrice le cui righe sono associate alle biblioteche e le cui colonne sono associate ai libri; l'elemento (i,j) della matrice rappresenta il numero di copie del libro j-esimo posseduto dalla biblioteca i-esima.
17. Visualizzazione, a partire dalla matrice dei possessi, delle biblioteche che possiedono tutti i libri posteriori ad un anno specificato dall'utente, posseduti dalla biblioteca il cui codice è specificato in input dall'utente.
18. Visualizzazione, a partire dalla matrice dei possessi, delle biblioteche, se esistono, che possiedono il massimo numero di copie di libri pubblicati in un intervallo di tempo specificato dall'utente e, nel contempo, possiedono il minimo numero di libri scritti in un secondo intervallo di tempo specificato dall'utente.
19. Calcolo, mediante una procedura ricorsiva, a partire dalla matrice dei possessi, del numero dei volumi che la biblioteca B1 possiede in più (o in meno) rispetto alla biblioteca B2, dove B1 e B2 sono codici forniti dall'utente.
20. Uscita dal programma.

3.2 Analisi dei requisiti

L'analisi dei requisiti è uno strumento importante nella realizzazione e progettazione di qualsiasi sistema software o progetto in ambito informatico. Grazie ad essa, è possibile definire le funzionalità che il sistema deve offrire, attraverso i requisiti richiesti dalle varie funzioni da implementare.

Durante questa analisi, si definiscono due tipi di requisiti, i *requisiti funzionali* e i *requisiti non funzionali*. I requisiti funzionali sono l'insieme di funzioni e caratteristiche che il sistema software deve implementare. I requisiti non funzionali, invece, sono un insieme di vincoli di vario tipo che il sistema è tenuto a rispettare.

Ad esempio, se consideriamo un programma per la gestione di un magazzino, i requisiti funzionali saranno l'insieme delle funzioni utili a gestire quest'ultimo, come la registrazione di un nuovo prodotto o l'aggiornamento della quantità presente nel magazzino stesso. I requisiti non funzionali, invece, potrebbero essere l'utilizzo di un interfaccia grafica, piuttosto che la console di comando, oppure la necessità di avere limitazioni sulle funzioni operabili, a seconda di chi utilizza il programma, e così via.

Vediamo, quindi, nelle prossime sezioni, i requisiti funzionali e non funzionali relativi al nostro progetto.

3.2.1 Requisiti funzionali

Come abbiamo appena detto, i requisiti funzionali saranno rappresentati dalle funzionalità che il nostro programma dovrà implementare.

Dalla descrizione della lista delle funzionalità che il sistema dovrà garantire, presentate nella Sezione 3.1, è possibile individuare i seguenti requisiti funzionali:

- Inserimento di una nuova biblioteca, di un nuovo libro e/o di un nuovo possesso da parte dell'utente.
- Rimozione dei dati di una biblioteca, di un libro e/o di un possesso, tramite i codici specificati dall'utente.
- Stampa delle informazioni di tutte le biblioteche, di tutti i libri e di tutti i possessi.
- Modifica del nome di una biblioteca, usando il codice inserito dall'utente.
- Modifica del numero di copie di un possesso, usando i codici di biblioteca e del libro inseriti dall'utente.
- Ricerca di una biblioteca nel sistema, dato il suo codice, specificato dall'utente.
- Ricerca dei libri pubblicati dopo un anno, inserito dall'utente.
- Ricerca di libri specifici, che si trovano in due biblioteche di una determinata città specificata dall'utente, pubblicati dopo un anno specificato dall'utente.
- Ricerca dei libri pubblicati dopo un determinato anno, in tutte le biblioteche di una città scelta dell'utente, che hanno un numero di copie pari o superiore a quello scelto sempre dall'utente.
- Costruzione e stampa della matrice dei possessi, come spiegato nella traccia.
- Visualizzazione dei dati delle biblioteche, che possiedono tutti i libri pubblicati prima di un anno specificato dall'utente, posseduti da una biblioteca, il cui codice è specificato dall'utente.
- Visualizzazione dei dati delle biblioteche che possiedono il massimo delle copie di libri pubblicati in un intervallo di date specificato dall'utente e, contemporaneamente, il minor numero di volumi dei libri scritti in un altro intervallo di date specificato.
- Calcolo ricorsivo del numero di volumi di due biblioteche scelte dall'utente, e relativo confronto.
- Operazioni di lettura e scrittura sui file, per la memorizzazione e la modifica dei dati.
- Utilizzo di liste non ordinate per la gestione dei dati di biblioteche, libri e possessi e delle loro operazioni.
- Implementazione di un menù per la gestione delle funzionalità.
- Uscita dal programma.

3.2.2 Requisiti non funzionali

I requisiti non funzionali, o vincoli, che il programma deve rispettare sono, invece, i seguenti:

- *Utilizzo di sottomenù*; il programma consisterà di un menù principale, suddiviso, poi, in diversi sottomenù, ciascuno dei quali legato alla gestione delle biblioteche, dei libri, dei possessi, nonché un menù extra per le operazioni che non ricadono strettamente nei tre precedenti.
- *Controlli sugli inserimenti*; il programma dovrà controllare l'inserimento dei vari dati richiesti all'utente per assicurarsi che non ci siano errori non voluti e, soprattutto, mantenere l'integrità dell'esecuzione.
- *Rimozione Concatenata*; il programma, qualora vengano rimossi una biblioteca o un libro, dovrà rimuovere ogni eventuale possesso associato ad essi in maniera

automatica, senza costringere l'utente ad una eliminazione manuale, in modo da evitare situazioni di ambiguità e possibili errori.
- *Inserimenti multipli*; il programma permette all'utente, tramite una richiesta effettuata dopo la conclusione di un inserimento, di continuare ad inserire altri dati o di fermarsi, in ciascuno dei tre inserimenti descritti.
- *Controllo sui file*; al lancio del programma, quest'ultimo controllerà lo stato dei file per la memorizzazione dei dati; se questi ultimi sono mancanti, essi vengono creati nuovamente; se, invece, sono danneggiati o corrotti, il programma non permetterà all'utente di eseguire alcuna funzione prima della risoluzione del problema.
- *Nessuna distinzione tra utenti*; il programma può essere eseguito da ogni utente senza limitazioni sulle funzionalità utilizzabili.

3.3 Progettazione

Terminata l'analisi e la stesura dei requisiti del sistema software, passiamo, ora, a descrivere, il comportamento del programma con l'utente, e anche parte della sua implementazione in Java, grazie ai diagrammi forniti da UML.

In questa sezione vedremo, quindi, i vari diagrammi che ci hanno aiutato nella progettazione, e successiva implementazione, del programma, iniziando dal diagramma dei casi d'uso, per capire come il programma gestisce l'interazione con l'utente, proseguendo con i diagrammi delle classi, e concludendo con i diagrammi di flusso di alcune delle richieste più complesse da realizzare.

3.3.1 Diagrammi dei casi d'uso

Iniziamo a vedere i diagrammi dei casi d'uso del nostro programma che, come abbiamo visto nel capitolo precedente, ci mostrano le varie funzioni a cui l'utente ha accesso. Essendo il programma suddiviso in menù, analizzeremo quattro diagrammi dei casi d'uso, ciascuno legato ad un menù diverso. I menù del programma sono:

- il menù per la gestione delle biblioteche;
- il menù per la gestione dei libri;
- il menù per la gestione dei possessi;
- il menù extra, che contiene funzionalità che non ricadono strettamente nei primi tre.

Si è scelto di tralasciare il menù generale, che permette la scelta di quale dei quattro menù appena citati utilizzare, in quanto si vuole concentrare l'attenzione sui singoli menù che permettono di utilizzare le funzionalità del programma.

I diagrammi vengono mostrati nelle Figure 3.1, 3.2, 3.3 e 3.4.

Iniziamo col *menù per la gestione delle biblioteche*, che permette all'utente di usare sei funzionalità, legate alla gestione delle biblioteche, presenti nel sistema:

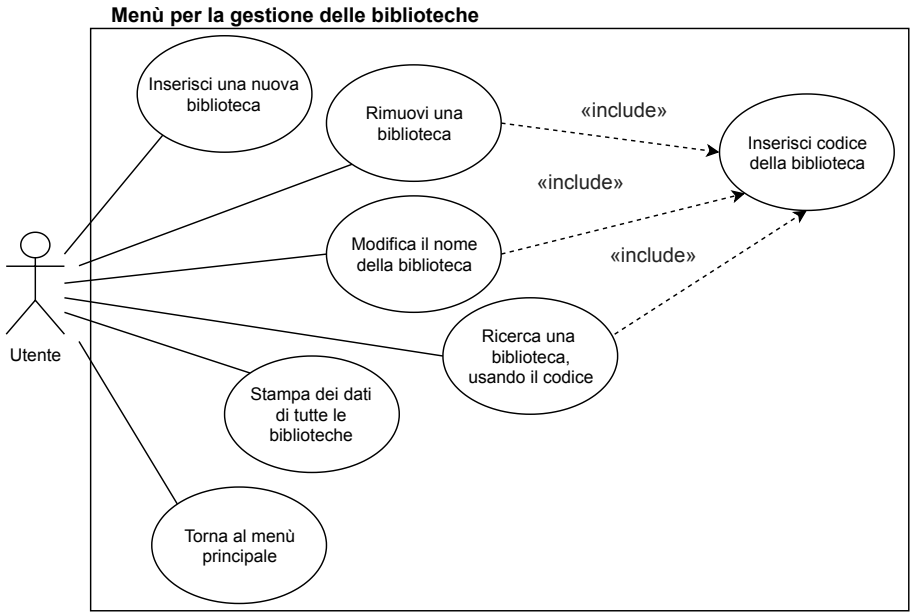


Figura 3.1. Diagrammi dei casi d'uso del menù per la gestione delle biblioteche

1. Inserisci una nuova biblioteca.
2. Rimuovi una biblioteca, che include l'inserimento del codice della biblioteca da eliminare.
3. Modifica il nome della biblioteca; anch'essa include l'inserimento del codice.
4. Ricerca di una biblioteca, usando il codice; include l'inserimento come sopra.
5. Stampa dei dati di tutte le biblioteche presenti nel sistema.
6. Torna al menù principale.

I dettagli delle varie funzioni utilizzate verranno trattati nel paragrafo successivo, relativo ai diagrammi delle classi e, successivamente, nel prossimo capitolo, da un punto di vista implementativo.

Nel *menù per la gestione dei libri*, invece, l'utente può scegliere di usare cinque funzionalità, tutte relative alla gestione dei libri, presenti nel sistema:

1. Inserisci un nuovo libro.
2. Rimuovi un libro, che include l'inserimento del codice del libro da eliminare.
3. Ricerca dei libri successivi ad un anno, che include l'inserimento dell'anno scelto.
4. Stampa dei dati di tutti i libri, presenti nel sistema.
5. Torna al menù principale.

Nel *menù per la gestione dei possessi* l'utente può scegliere di usare, anche qui, cinque funzionalità, tutte relative alla gestione dei possessi, memorizzati nel sistema:

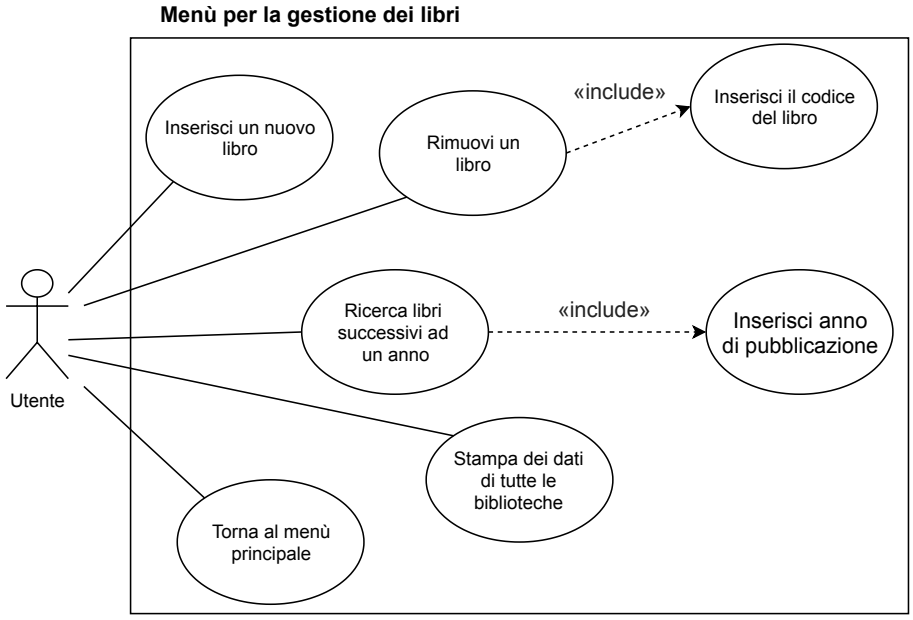


Figura 3.2. Diagrammi dei casi d'uso del menù per la gestione dei libri

1. Inserisci un nuovo possesso.
2. Rimuovi un possesso, che include l'inserimento del codice del libro e della biblioteca, associati al possesso da eliminare.
3. Modifica il numero di copie possedute, che include l'inserimento dei codici del libro e biblioteca, come sopra.
4. Stampa dei dati di tutti i possessi, presenti nel sistema.
5. Torna al menù principale.

Nel *menù per la gestione delle funzionalità extra*, infine, l'utente può selezionare tra sette scelte, legate alle funzionalità più complesse del programma, e che non ricadono strettamente in nessuno dei menù precedenti:

1. Ricerca di libri specifici, in esattamente due biblioteche di una città, che include la ricerca dei libri per anno di pubblicazione e la ricerca delle biblioteche per città.
2. Ricerca dei libri posteriori ad un anno specificato dall'utente, presenti nelle biblioteche di una città specificata dall'utente, che include le medesime funzioni di cui sopra.
3. Costruzione e stampa della matrice dei possessi.
4. Visualizzazione dei dati delle biblioteche, che hanno libri pubblicati prima di un anno specificato dall'utente, posseduti da una biblioteca specificata, usando la matrice dei possessi, che include la ricerca dei libri per anno e la ricerca della biblioteca, usando il codice.

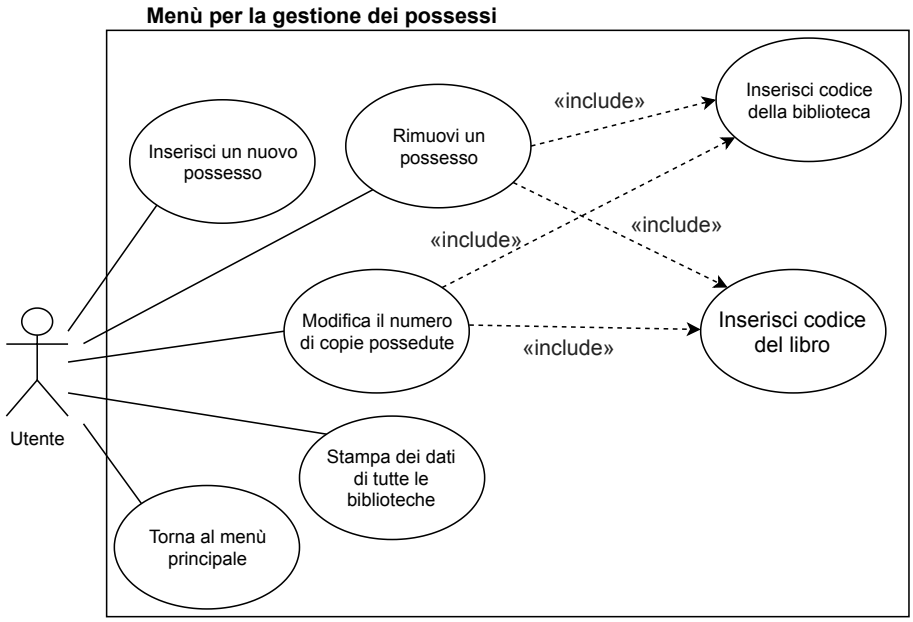


Figura 3.3. Diagrammi dei casi d'uso del menù per la gestione dei possessi

5. Calcolo della differenza dei volumi tra due biblioteche, che include la ricerca della biblioteca tramite codice.
6. Visualizzazione dei dati delle biblioteche, con massimo numero di copie di certi libri e minimo numero di altri, che include la ricerca dei libri per anno.
7. Torna al menù principale.

Queste opzioni, implementano, rispettivamente, le funzionalità 14, 15, 16, 17, 18 e 19, che sono state raggruppate in un menù a parte, anche per non appesantire i menù delle gestioni dei singoli oggetti.

3.3.2 Diagrammi delle classi

Vediamo ora i diagrammi di tutte le classi che abbiamo utilizzato nel nostro programma.

Nel progetto sono state definite e implementate un totale di tredici classi al fine di soddisfare tutti i requisiti funzionali e non funzionali. Per semplicità, di seguito queste verranno raggruppate in quattro macro gruppi che conterranno, rispettivamente, le classi degli *oggetti principali*, ovvero Biblioteca, Libro e Possesso, le classi delle *funzioni base* principali degli oggetti, quali inserimento, rimozione, modifica, ricerca e stampa, le classi delle *gestioni di file*, *liste* e *controlli* e, infine, le classi della *matrice* e del *menù* del programma.

Iniziamo con i diagrammi delle classi degli oggetti, ovvero le *classi Biblioteca, Libro e Possesso*, mostrate in Figura 3.5.

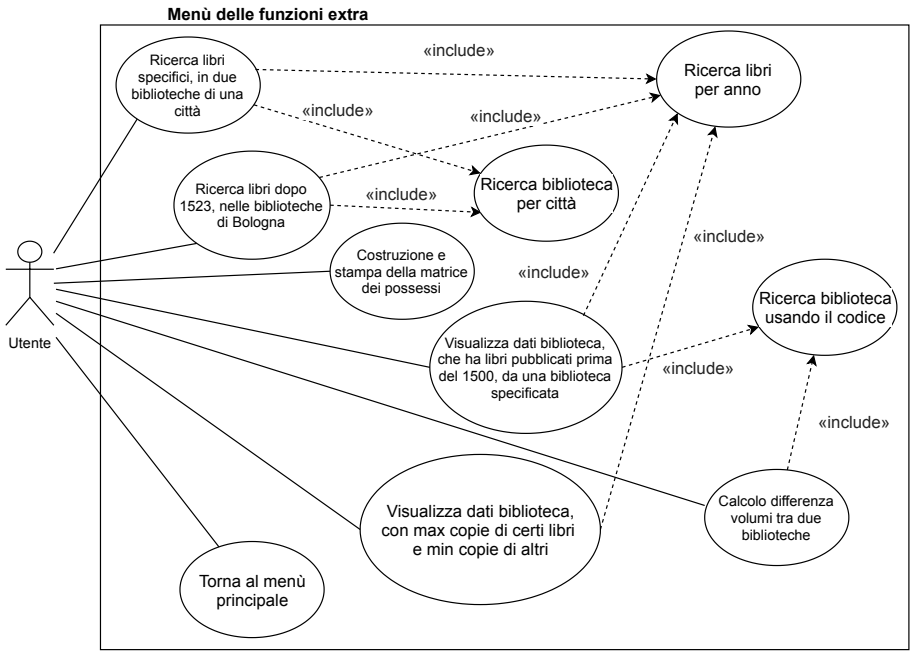


Figura 3.4. Diagrammi dei casi d'uso del menù per le funzionalità extra

Tutte e tre le classi sono simili tra loro; ciascuna presenta gli attributi richiesti dalla traccia (Sezione 3.1), i metodi *set* e *get*, metodi standard per la gestione delle interazioni degli attributi della classe, con classi o codice esterno ad essa. In ogni classe abbiamo tre metodi *get* e tre metodi *set*, legati, a coppie, ai singoli attributi dei vari oggetti. Infine, viene definito, in ciascuna classe, un costruttore, evidenziato con lo stesso nome della classe, ma senza un tipo. In questo specifico progetto, tutti i costruttori delle classi sono dei costruttori vuoti, ovvero non compiono alcuna chiamata di funzione o inizializzazione di variabile quando vengono eseguiti.

Successivamente, abbiamo le classi che si occupano delle operazioni di inserimento, rimozione, modifica, ricerca e stampa dei dati di ciascuno degli oggetti che abbiamo definito. Nelle Figure 3.6 e 3.7 vengono mostrati i diagrammi di tali classi, chiamati, per semplicità, con lo stesso nome delle operazioni che implementano.

Iniziamo a descrivere la *classe Modifica*, in quanto meno densa di contenuti rispetto alle altre.

Notiamo che non abbiamo alcun attributo definito, e soltanto due metodi, che si occupano, rispettivamente, della modifica del nome di una biblioteca, che viene specificata dall'utente tramite il codice, e della modifica del numero di copie legate ad un determinato possesso, anche qui specificato dall'utente tramite i codici del libro e della biblioteca associati. A questi metodi vengono passati, come parametri, le rispettive liste contenenti i dati e un indice che determina il dato sul quale lavorare.

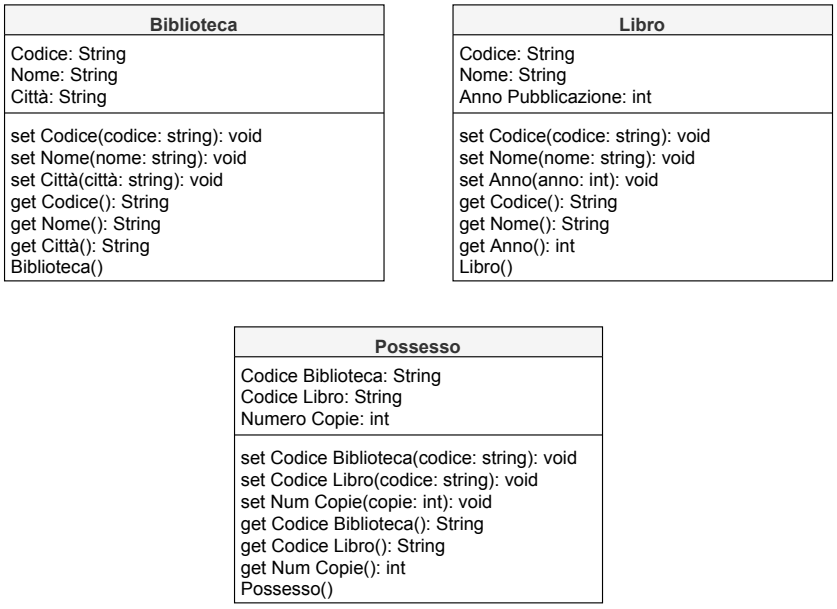


Figura 3.5. Diagrammi delle classi Biblioteca, Libro e Possesso

La modifica di tali dati su file verrà effettuata da un'altra classe, una volta terminato il metodo.

La *classe Inserimento*, invece, gestisce tutti gli inserimenti di dati che vengono richiesti dall'utente, o da altri metodi per ricerche o modifiche. Anche in questa classe non abbiamo attributi ma soltanto metodi. Partendo dai primi tre, questi permettono l'inserimento di un nuovo oggetto, di tipo Biblioteca, Libro o Possesso, all'interno della relativa lista, che viene passata come parametro, insieme ad una variabile di tipo GestioneFile, una classe che vedremo a breve, che consente la scrittura dei nuovi dati sul file.

I quattro metodi rimanenti, invece, sono usati per le gestire le richieste, da parte del programma, di inserimento di codici, anni o numeri di copie, necessari per effettuare ricerche o modifiche di dati già presenti nel sistema. Anche in questo caso, è presente un costruttore vuoto, con lo stesso nome della classe e senza tipo.

La *classe Rimozione* si occupa di cancellare dati di oggetti che non servono o che devono essere rimossi dal sistema. I primi tre metodi, come per l'inserimento, sono legati a ciascuno degli oggetti, di tipo Biblioteca, Libro e Possesso, che, tramite la relativa lista ed un indice, entrambi passati come parametri, permettono la rimozione di quei dati dalla lista, che, successivamente, verrà aggiornata, anche, sul relativo file. Gli altri due metodi, invece, sono stati implementati successivamente, durante la scrittura del programma, e permettono di rimuovere tutti gli eventuali possessi, legati ad una Biblioteca o ad un Libro, che si decide di rimuovere dal sistema. La rimozione avviene su una lista di stringhe, passata come parametro, che contiene tutti i dati dei possessi presenti nel sistema, dove, grazie al codice della biblioteca o del libro, anch'esso passato come parametro, vengono ricercati e rimossi. L'aggiornamento sul file avviene successivamente, grazie al metodo di un'altra

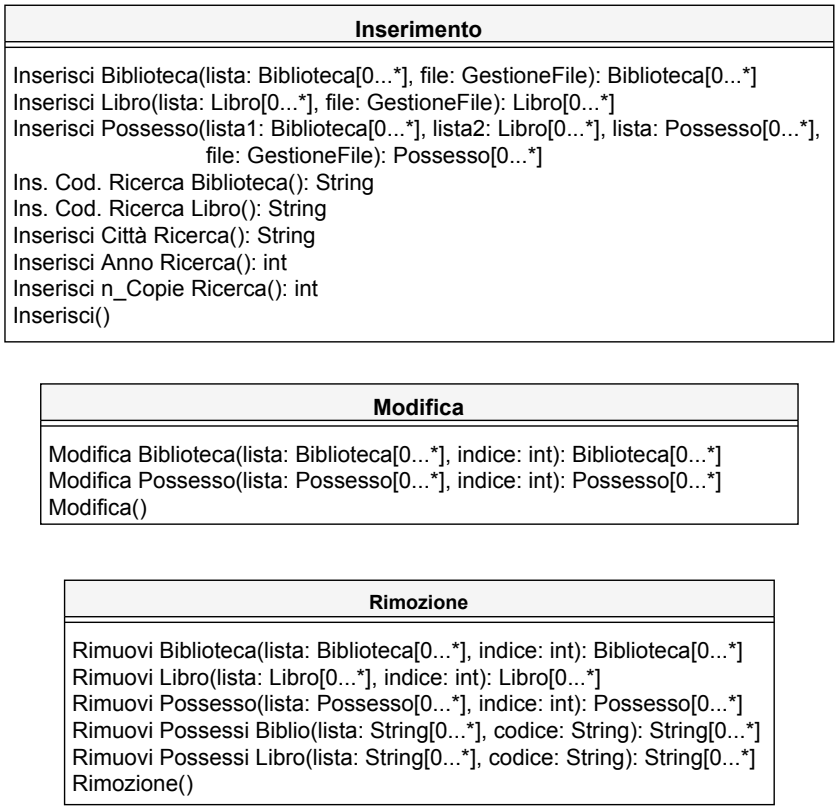


Figura 3.6. Diagrammi delle classi Inserimento, Modifica e Rimozione

classe.

Vediamo, ora, le restanti due classi, citate precedentemente, ovvero le classi Ricerca e Stampa.

La *classe Stampa*, la più semplice delle due, si occupa di mostrare all'utente i dati, presenti nel sistema, dei vari oggetti, oppure il risultato di ricerche o particolari funzioni del programma. I primi tre metodi che vediamo sono legati ai singoli oggetti, di tipo Biblioteca, Libro e Possesso, e mostrano tutti i dati presenti nel sistema legati a ciascuno di essi, grazie alla lista relativa che viene passata come parametro. Il quarto metodo, invece, stampa le informazioni ricavate dalla ricerca di una determinata biblioteca, effettuata precedentemente, alla quale viene passata la lista contenente i dati di tutte le biblioteche nel sistema, e l'indice di quella cercata e, di cui, mostrare i dati. L'ultimo metodo si occupa della stampa della matrice dei possessi, che viene passata come parametro, insieme alle liste delle biblioteche e dei libri, dalle quali ottenere i codici da assegnare alle righe e alle colonne della matrice.

La *classe Ricerca*, infine, si occupa di realizzare tutte le operazioni di ricerca svolte all'interno del programma. La classe definisce due attributi, due liste per essere specifici, utilizzate per contenere i risultati delle ricerche di biblioteche e libri, qualora vengano trovati più dati, da dover poi elaborare o stampare. Riguardo ai metodi, invece, essendoci degli attributi, come nel caso delle classi Biblioteca, Libro

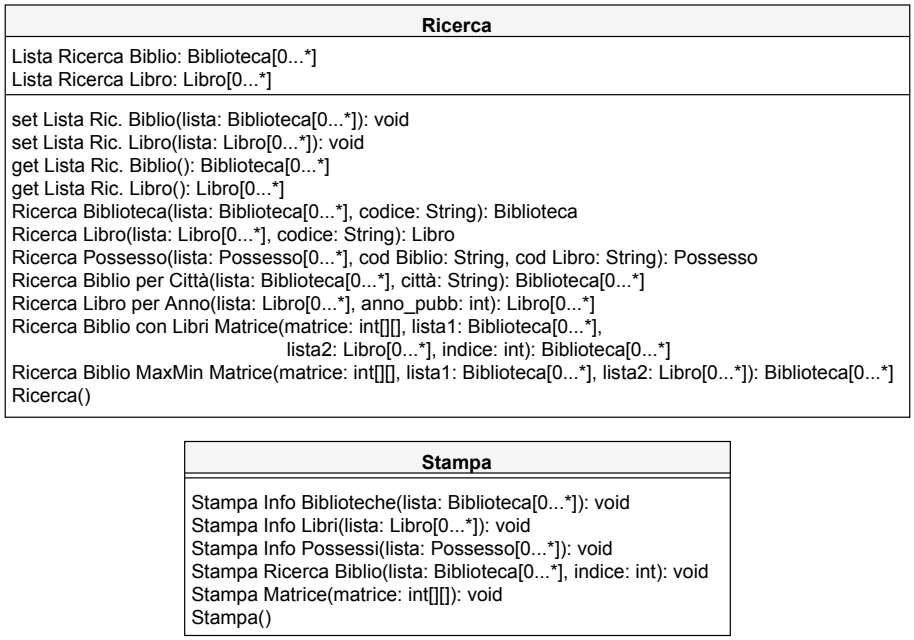


Figura 3.7. Diagrammi delle classi Ricerca e Stampa

e Possesso, abbiamo due coppie di funzioni *get* e *set*, per interazione con le liste sopracitate, con metodi e codice esterno alla classe. Oltre a queste, abbiamo, anche qui, tre metodi separati per effettuare le ricerche di una biblioteca, libro e/o possesso, utilizzando le rispettive liste e i codici, che verranno inseriti dall'utente, passati come parametri. Per soddisfare alcuni requisiti funzionali, la classe dispone, anche, di due funzioni specifiche per la ricerca delle biblioteche per città, e per la ricerca dei libri per anno di pubblicazione; la prima consente tale ricerca, grazie alla lista delle biblioteche e alla città specificata, che vengono passate come parametri al metodo, mentre, alla seconda, oltre alla lista e all'anno di pubblicazione scelto, viene passato anche un parametro chiamato "Verso", usato per determinare se si dovranno cercare libri anteriori o posteriori all'anno scelto, o pubblicati precisamente in quest'ultimo. Gli ultimi due metodi, infine, sono legati a delle operazioni da effettuare sulla matrice dei possessi, e si occupano delle ricerche richieste dalle funzionalità 17 e 18, presenti nella traccia in Sezione 3.1. Ad entrambi i metodi vengono passati come parametri la matrice, le liste di libri e biblioteche per estrarre i dati richiesti, e nel caso della richiesta 17, la ricerca delle biblioteche con libri posteriori all'anno specificato, viene passato anche un indice per identificare la biblioteca inserita dall'utente. La classe termina con la definizione del costruttore, anche qui vuoto.

Proseguiamo col terzo macro gruppo, formato dalle classi di gestione e controllo, quali *GestioneFile*, *GestioneListe* e *Controllo*, di cui vediamo i diagrammi in Figura 3.8.

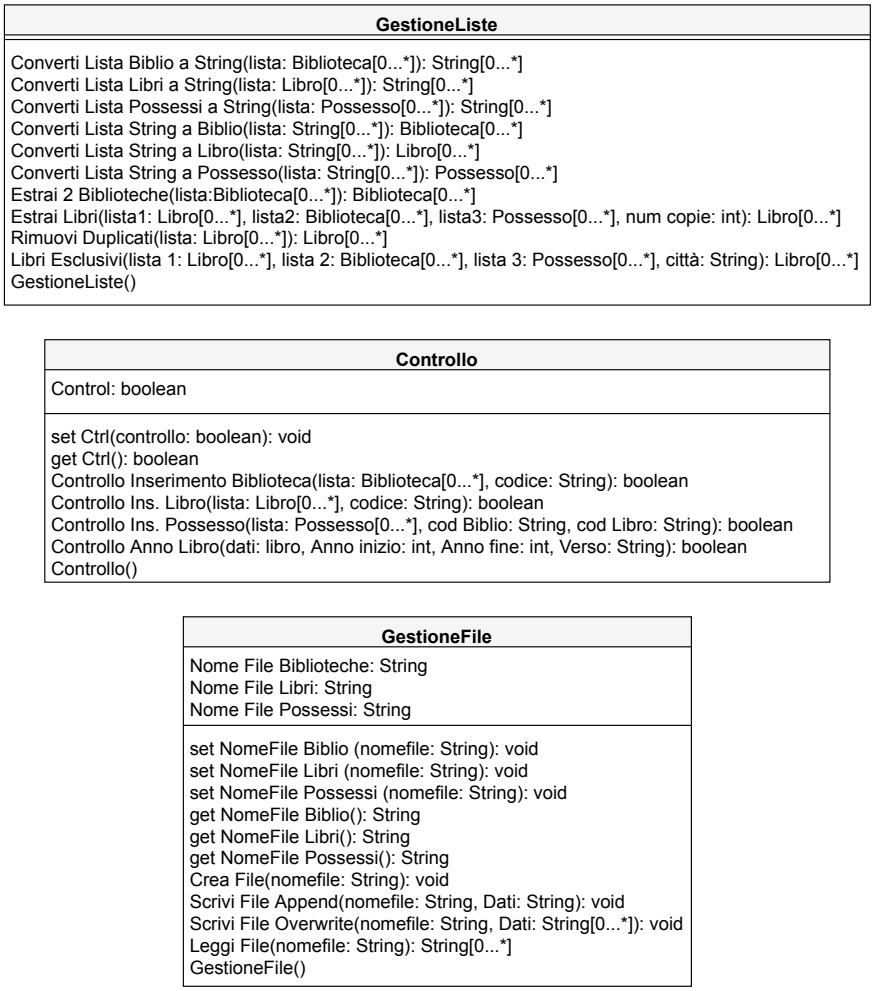


Figura 3.8. Diagrammi delle classi GestioneFile, GestioneListe e Controllo

Tutte queste classi sono state implementate successivamente alla prima progettazione, in quanto, soprattutto la classe Controllo, non erano state prese in considerazione per lo sviluppo dei requisiti funzionali richiesti. In una prima fase, infatti, il controllo degli inserimenti non veniva implementato, così come la gestione dei file, che veniva effettuata direttamente sul codice sorgente, a seconda delle necessità. Con la stesura del codice, però, si è reso necessario andare a raggruppare questi metodi, in classi specifiche, in primis per la comodità d'uso e la scalabilità del programma, e successivamente, anche, per non appesantire il codice di righe superflue, dovute al ripetersi dei medesimi metodi.

Iniziamo con la *classe Controllo*, che gestisce i controlli sugli inserimenti dei dati, per evitare che vengano inseriti duplicati. La classe definisce una variabile di controllo, di tipo booleano, per gestire i risultati delle operazioni de effettuare, e semplificare l'implementazione nel resto del codice, fornendo come risultato soltanto

un successo o un fallimento del controllo stesso. Per questi attributi abbiamo, anche qui, una coppia di metodi *set* e *get*, seguita dai tre metodi per i controlli sui dati di tipo Biblioteca, Libro e Possesso, usati per evitare che vengano inseriti dati di oggetti già presenti nel sistema. A questi metodi vengono passati, come parametri, le rispettive liste con i dati e i codici di biblioteche e libri da inserire, per verificare la presenza di duplicati. Infine, l'ultimo metodo rimasto, oltre al costruttore, effettua il controllo sull'inserimento degli anni di pubblicazione, usati per le ricerche di libri; a tale metodo vengono passati un oggetto di tipo Libro, l'anno iniziale e finale scelti, e il verso secondo il quale effettuare il controllo.

La *classe GestioneFile*, invece, si occupa di gestire tutte le operazioni da eseguire sui file, utilizzati dal programma. Come attributi abbiamo i tre nomi dei file, usati per memorizzare le informazioni delle biblioteche, dei libri e dei possessi nel sistema. Oltre alla dichiarazione, in questo caso gli attributi sono tutti già inizializzati con il nome del file che si dovrà utilizzare. Per quanto riguarda i metodi, abbiamo tre coppie di metodi *get* e *set* per i tre nomi dei file[1], il metodo di creazione del file, qualora questo non sia già presente, grazie al nome passato come parametro; il metodo per la lettura del file, che utilizza, oltre al nome, una variabile per la lettura di stringhe di dati, e due metodi di scrittura, uno per le aggiunte di nuovi dati e uno per la sovrascrittura dei dati modificati, ai quali vengono passati il nome del file e, rispettivamente, una variabile stringa contenente il nuovo dato da aggiungere e una lista di stringhe con tutti i dati da modificare. Anche in questa classe, è presente un costruttore vuoto.

La *classe GestioneListe*, infine, si occupa di gestire le operazioni legate alle liste utilizzate dal programma, in diverse funzionalità. In questa classe non abbiamo alcun attributo. Le liste non vengono dichiarate all'interno della classe, ma direttamente nel corpo del programma stesso, per via del tipo di variabile utilizzato (*ArrayList*, che vedremo nel prossimo capitolo). Di metodi, invece, ne abbiamo diversi, a cominciare dai primi sei che, a coppie, si occupano della conversione delle liste utilizzate. Nel programma si è avuto il bisogno di implementare, oltre alle liste del tipo di dato richiesto, quindi lista di tipo Biblioteca, Libro e Possesso, anche liste di tipo stringa, per facilitare la scrittura e la lettura da file. Per questo motivo, sono stati implementati questi sei metodi, tre per convertire le liste di tipo stringa in liste del tipo di dato richiesto, e tre metodi per l'operazione inversa. Gli altri quattro metodi rimanenti si occupano di gestire alcune operazioni, per realizzare le funzionalità più complesse. In particolare, il metodo per estrarre due biblioteche dalla lista, passata come parametro, e quello per estrarre i libri con un determinato numero di copie, per il quale occorrono tutte e tre le liste principali e, ovviamente, il numero di copie da ricercare, tutti passati come parametri. Abbiamo, poi, un metodo per rimuovere eventuali libri duplicati presenti nelle ricerche, usando la lista di tutti i libri, e un metodo per la ricerca dei libri esclusivi di una biblioteca, anche qui

[1] Nel programma, come si vedrà successivamente, vengono utilizzati soltanto i metodi *get* relativi ai nomi dei file, in quanto un utente non può modificare a piacimento i nomi dei file contenenti i dati presenti nel sistema. I metodi *set* vengono, comunque, definiti nella classe, per future implementazioni, legate alla gestione del programma da parte di un tecnico o di un utente avanzato, che avrà la possibilità di modificare informazioni di questo tipo.

usando tutte le tre liste, e la città della biblioteca richiesta. La classe si conclude, come sempre, con il costruttore vuoto.

Per concludere con i diagrammi delle classi vediamo, ora, quelli delle ultime due classi, ovvero *Matrice e Menu*. I due diagrammi vengono mostrati in Figura 3.9.

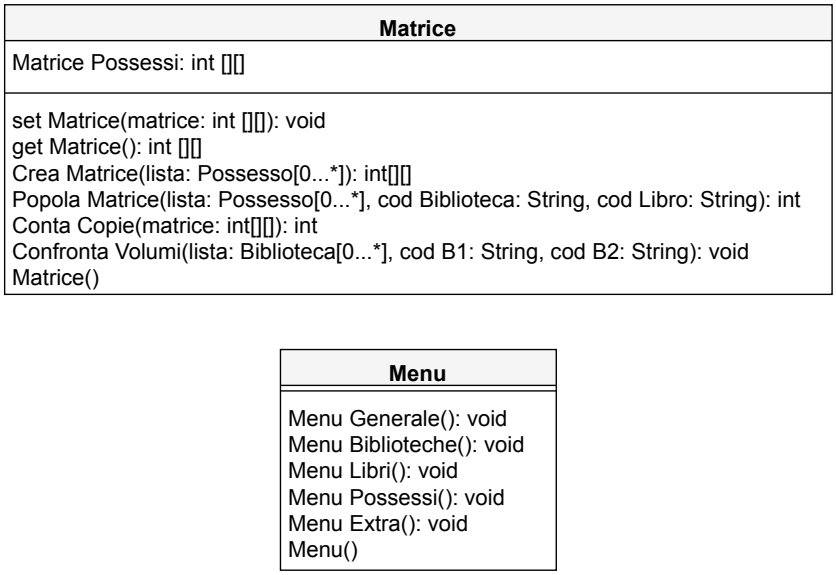


Figura 3.9. Diagrammi delle classi Matrice e Menu

La *classe Menu*, rappresenta i metodi usati per la costruzione e la stampa a video dei menù che vengono utilizzati dal programma. Non viene definito alcun attributo, e i metodi sono tutti senza ritorno. Ciascuno di essi, stampa il menù corrispondente al suo nome, nell'ordine quello generale, che comprende poi quelli per la gestione delle biblioteche, dei libri e dei possessi, e il menù con le funzionalità extra, che non ricadono precisamente in nessuno degli altri, includendo, anche, le funzionalità legate alla matrice dei possessi.

Quest'ultima viene gestita dall'ultima classe rimasta, chiamata *Matrice*, che si occupa di gestire tutte le operazioni, associate a questa struttura di dati, utili al nostro programma. Nella classe viene definita una variabile che rappresenta la matrice di interi, nella quale verranno inseriti i numeri delle copie dei vari libri, in corrispondenza della biblioteca che li possiede. Avendo un attributo, i primi due metodi che incontriamo, come nei casi precedenti, sono i metodi *get* e *set* per la matrice. Successivamente, abbiamo due metodi, uno per la creazione della matrice, che utilizza le tre liste legate alle biblioteche, ai libri e ai possessi, per inserire gli elementi nella matrice, e l'altro che si occupa di popolare gli elementi vuoti della matrice, assegnando le giuste copie al libro corrispondente, usando la lista dei possessi e i codici della biblioteca e dei libri, passati come parametri. Questi due metodi

vengono utilizzati insieme per assicurarsi che non ci siano errori nella composizione dei vari elementi della matrice. Gli ultimi due metodi rimasti, sono stati scritti per implementare la funzionalità 19 del programma, ovvero il confronto del numero di volumi tra due biblioteche, in maniera ricorsiva. Il primo si occupa di produrre il risultato del confronto, specificando quale delle due biblioteche ha più volumi, usando le liste delle biblioteche e dei libri, insieme ai codici delle due biblioteche scelte dall'utente, tutti passati come parametri. Il secondo, invece, è il metodo ricorsivo, che somma il totale dei volumi posseduti dalla biblioteca, sulla base dei libri che essa possiede, utilizzando la matrice dei possessi e tre indici per gestire la ricorsione, anche qui forniti come parametri del metodo. Come nei casi precedenti, è presente il costruttore vuoto.

3.3.3 Diagrammi di flusso

I *diagrammi di flusso* sono una rappresentazione grafica di un algoritmo o una funzione complessa, di un programma o di un sistema software, attraverso l'utilizzo di una simbologia standard e semplice da utilizzare.

Nei diagrammi che vedremo in questo paragrafo, infatti, vengono utilizzati degli elementi grafici per la costruzione, che introdurremo brevemente per farli comprendere più facilmente.

Un diagramma di flusso ha un blocco iniziale e finale, indicati da un rettangolo con gli angoli smussati, con all'interno scritto "Inizio" e "Fine", dove, dal primo, parte una freccia, che, dopo aver attraversato tutti i vari elementi del diagramma, termina nel secondo.

Gli altri elementi da utilizzare sono tre: il primo, per indicare operazioni di I/O o di lettura e scrittura, rappresentato da un trapezio isoscele, con, all'interno, l'azione che viene eseguita; il secondo, che indica le azioni che vengono eseguite, come, ad esempio, somme o assegnazioni, rappresentato da un rettangolo, con l'azione da eseguire scritta all'interno; il terzo, infine, per indicare dei test, con due o più frecce uscenti, indicanti la decisione di quest'ultimo, rappresentato da un rombo, con all'interno la condizione da verificare.

Tutti questi elementi, chiamati *blocchi elementari*, hanno una freccia entrante ed una uscente (ad eccezione del blocco di test, che può avere più frecce uscenti) che li collegano tra loro, secondo questa regola e che partono da un blocco di "Inizio" e terminano in un blocco di "Fine". I diagrammi costruiti per il progetto fungeranno, anche, da esempio per comprendere la costruzione.

Vediamo, quindi, i diagrammi di flusso, relativi alle funzionalità più complesse da implementare nel programma.

I diagrammi successivi, sono relativi alle funzionalità 14, 15, 17 e 18 (Sezione 3.1). Sebbene, anche la funzionalità numero 19 ricadrebbe in quelle più complesse da analizzare, non è opportuno realizzarne un diagramma per via della procedura ricorsiva richiesta.

Iniziamo con il diagramma di flusso della funzionalità numero 14, mostrato in Figura 3.10.

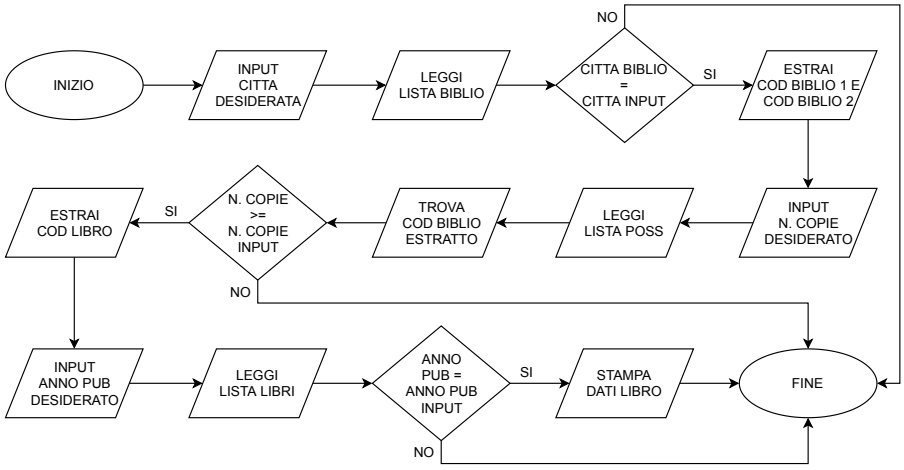


Figura 3.10. Diagrammi di flusso della Funzionalità 14 del programma

Descriviamo, quindi, il diagramma, partendo da "Inizio".

Dopo il blocco iniziale, viene chiesto di inserire la città desiderata per la ricerca, viene letta la lista delle biblioteche, e si verifica se la città inserita corrisponde alla città della biblioteca nella lista; in caso affermativo, la ricerca prosegue; in caso negativo, essa termina, come si vede dalla freccia che va verso il blocco di "Fine", con esito negativo. Si prosegue, quindi, estraendo i dati delle due biblioteche cercate; viene richiesto l'inserimento del numero di copie desiderato; viene, poi, letta la lista dei possessi, vengono cercati i codici delle biblioteche estratte e si controlla se il numero delle copie è maggiore o uguale a quello inserito: se si, la ricerca prosegue, se no, termina, come sopra. Viene, poi, estratto il codice del libro corrispondente, viene chiesto di inserire l'anno di pubblicazione desiderato, si legge la lista dei libri e, si verifica, che l'anno inserito sia lo stesso di quello di pubblicazione: se è uguale, allora si stampano i dati del libro e la ricerca termina, altrimenti termina direttamente, notificando che quest'ultima non ha prodotto risultato.

Come si è visto in questo primo diagramma, la lettura è molto semplice e lineare; basta seguire il verso della freccia e, in caso di verifiche, seguire la freccia indicante l'esito, continuando fino ad arrivare al blocco di "Fine".

Vediamo, ora, il diagramma di flusso della funzionalità numero 15, mostrato in Figura 3.11.

Dopo il blocco iniziale, viene letta la lista delle biblioteche e si verifica che la città della biblioteca letta sia quella specificata dall'utente: se lo è, la ricerca prosegue, altrimenti la ricerca termina con esito negativo. Si estrae, quindi, il codice della biblioteca, si legge la lista dei possessi, si trova il codice estratto al suo interno e si estraggono i codici dei libri posseduti da quest'ultima. Si legge, poi, la lista dei libri e si verifica se l'anno di pubblicazione del libro è maggiore di quello specificato dall'utente: se è così, si stampano i dati del libro e si termina la ricerca, altrimenti si termina la ricerca, con esito negativo.

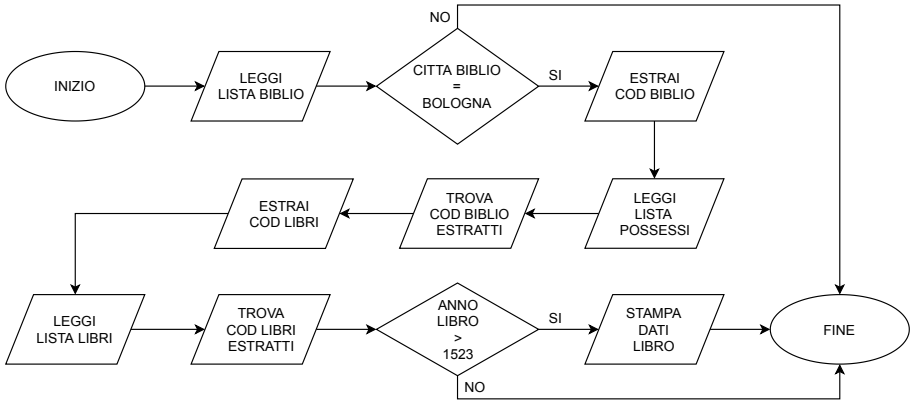


Figura 3.11. Diagrammi di flusso della Funzionalità 15 del programma

Proseguiamo con il diagramma di flusso della funzionalità numero 17, mostrato in Figura 3.12.

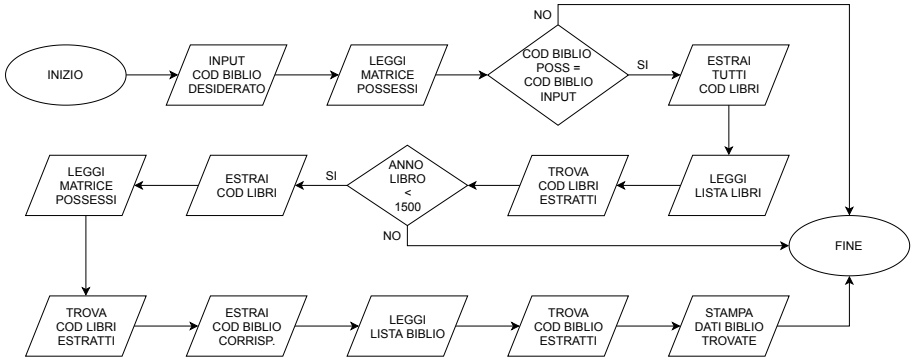


Figura 3.12. Diagrammi di flusso della Funzionalità 17 del programma

Dopo il blocco iniziale, viene chiesto di inserire il codice della biblioteca da cercare, viene letta la matrice dei possessi e si verifica se il codice della biblioteca, legato alla riga della matrice, è lo stesso di quello inserito: se si, la ricerca prosegue; se no, termina con esito negativo. Vengono, quindi, estratti tutti i libri, viene letta la lista dei libri, trovati i codici estratti e si verifica che l'anno è maggiore di quello specificato dall'utente: se lo è, la ricerca continua, altrimenti termina con esito negativo. Si prosegue estraendo i codici dei libri che rispettano la condizione, si legge la matrice dei possessi, si trovano i codici dei libri e si estraggono i codici delle biblioteche che li possiedono. Infine, si rilegge la lista delle biblioteche, si trovano i codici estratti e si stampano le informazioni, terminando la ricerca.

Vediamo, infine, il diagramma di flusso della funzionalità numero 18, mostrato in Figura 3.13.

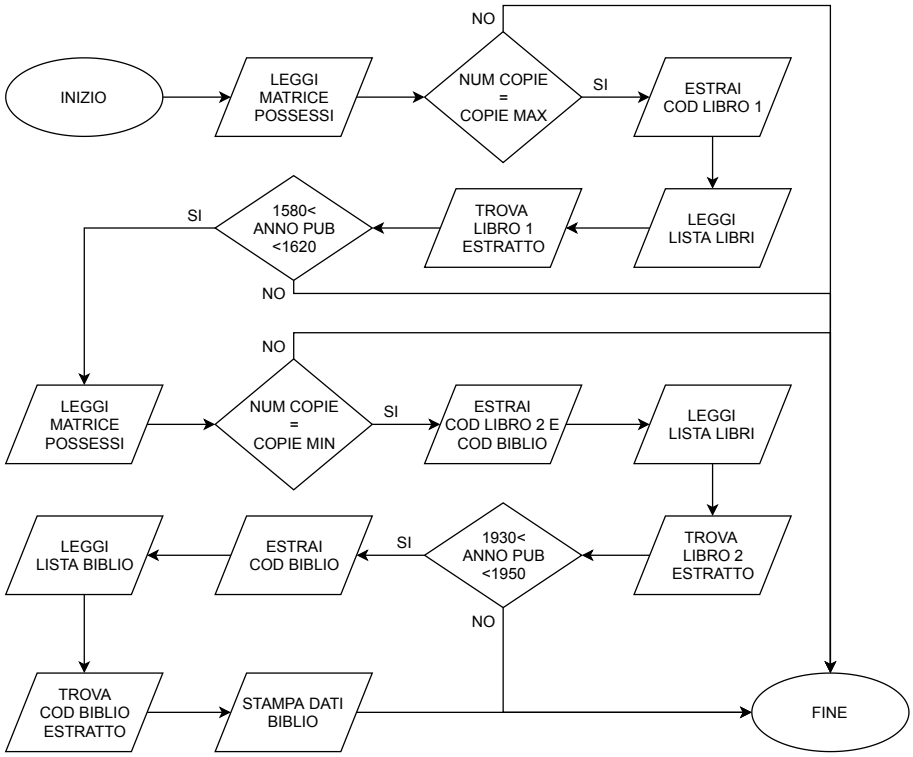


Figura 3.13. Diagrammi di flusso della Funzionalità 18 del programma

Dopo il blocco iniziale, viene letta la matrice dei possessi e si verifica, subito, se il numero delle copie di un libro è uguale al numero massimo consentito: se lo è, la ricerca prosegue, altrimenti termina con esito negativo. Viene, allora, estratto il codice del primo libro, viene letta la lista dei libri, viene trovato il primo libro estratto e viene verificato se l'anno di pubblicazione di quest'ultimo è nel primo intervallo specificato dall'utente: in caso affermativo, si prosegue, altrimenti la ricerca termina, con esito negativo. Si legge, nuovamente, la matrice dei possessi e si verifica se il numero di copie di un altro libro, della stessa riga e, quindi, biblioteca, è pari al minimo numero, ovvero "1": se è così, la ricerca continua, altrimenti termina con esito negativo. Viene, poi, estratto il codice del secondo libro e quello della biblioteca che li possiede entrambi, viene letta la lista dei libri, viene trovato il secondo libro e verificato se l'anno di pubblicazione rientra nel secondo intervallo specificato dall'utente: se lo è, si prosegue, altrimenti la ricerca termina con esito negativo. Infine, si estrae il codice della biblioteca, viene letta la lista delle biblioteche, viene

trovato il codice estratto e vengono stampati i dati relativi alla biblioteca cercata, terminando la ricerca.

Negli ultimi due casi visti, le operazioni da compiere sono da intendersi per una singola riga della matrice; ovviamente, nella funzione implementata nel codice, tale procedimento verrà ripetuto per ogni riga, fino a trovare tutti i risultati che soddisfano le condizioni.

4

Implementazione

In questo capitolo, dopo aver descritto, tramite vari diagrammi, le classi e le funzioni, vedremo come queste ultime sono state implementate all'interno del codice. Non ci soffermeremo su tutto il codice scritto, ma soltanto su alcune porzioni di esso, per non appesantire l'elaborato e la lettura.

4.1 Strumenti utilizzati

L'implementazione di questo progetto, come menzionato nel Capitolo 2, è stata effettuata attraverso il linguaggio Java. La fase di scrittura e di codifica è stata interamente effettuata tramite *Eclipse*, un ambiente di sviluppo integrato, anche detto IDE (Integrated Development Enviroment), creato da un consorzio di grandi società, chiamato *Eclipse Foundation*. Eclipse viene utilizzato per progettare e codificare programmi, non soltanto in Java, ma anche in C++, PHP, XML, JavaScript e altri linguaggi di programmazione. In Figura 4.1 vengono mostrati i loghi del consorzio e di Eclipse stesso.

Figura 4.1. Loghi di Eclipse Foundation e di Eclipse

Essendo scritto in Java, Eclipse, oltre alle sue funzioni di compilatore per il codice, offre la possibilità di aiutare l'utente direttamente durante la scrittura, attraverso l'auto-completamento di determinate chiamate e/o funzioni standard, come, ad esempio, un ciclo `for`, oppure il semplice richiamo di una funzione definita dall'utente, indicando, inoltre, a quest'ultimo eventuali errori di sintassi o errori ortografici presenti nel codice, durante la scrittura stessa e senza dover compilare continuamente.

4.2 Struttura del programma

Il programma è stato strutturato su singoli file sorgente, ciascuno dei quali contenente codice relativo ad una classe diversa, utilizzata nel programma stesso. Grazie ad Eclipse, è stato possibile gestire il progetto direttamente da software, creando una nuova cartella, dove tutti questi file sarebbero stati contenuti.

Eclipse, inoltre, ci ha dato la possibilità di creare e gestire, in maniera semplice e diretta, il package che raggruppa tutte le classi utilizzate dal programma.

Un *package*, in Java, è uno strumento che permette di raggruppare e organizzare le classi legate ad un progetto o programma. Per inserire una classe in un package, il modo più semplice è quello di utilizzare l'istruzione riportata nel Listato 4.1.

```
1   package nome-del-pacchetto;
```

Listato 4.1. Istruzione per includere una classe in un package

Tale istruzione va posta all'inizio della classe stessa.

Nel caso si vogliano importare una o più classi all'interno del programma, basterà scrivere le istruzioni riportate nel Listato 4.2.

```
1   import nome-del-pacchetto.*; \\Importa tutte le classi
2
3   import nome-del-pacchetto.NomeClasse; \\Importa soltanto la classe NomeClasse
```

Listato 4.2. Istruzione per importare un package in una classe

Nelle istruzioni sarà necessario specificare quali classi si vogliono importare; a tal fine si utilizza la sintassi precedentemente descritta.

Grazie a Eclipse, però, questo non è stato necessario, in quanto la gestione del package viene eseguita grazie al software, senza che l'utente si debba preoccupare di specificare, in ogni classe, il package dal quale attingere per usare funzioni di altre classi, inserendo soltanto l'istruzione per includerle.

In Figura 4.2, infine, si può vedere com'è strutturato il programma, una volta che quest'ultimo viene inserito in Eclipse.

Dentro la cartella `src`, contenuta in quella relativa a `Progetto`, sono presenti tutti i file `.java` contenenti il codice sorgente di tutte le classi del programma. Come si può notare, tutte queste classi, sono raggruppate sotto un unico package di nome `progetto`. Gli ultimi tre file che compaiono sul fondo, sono i tre file `.txt`, utilizzati per memorizzare i dati degli oggetti corrispondenti, presenti nel sistema.

Si tiene a far notare, infine, che i piccoli triangoli gialli, con all'interno un punto esclamativo, presenti nell'angolo delle icone dei file contenenti il codice delle classi, non indicano errori, ma "warning", ovvero avvertimenti, dovuti all'utilizzo di un determinato metodo, chiamato `Scanner`, che normalmente dovrebbe essere chiuso una volta terminato il suo utilizzo, ma che, nel nostro caso, non può esserlo, in quanto chiuderlo comporterebbe l'impossibilità di acquisire dati da tastiera successivamente.

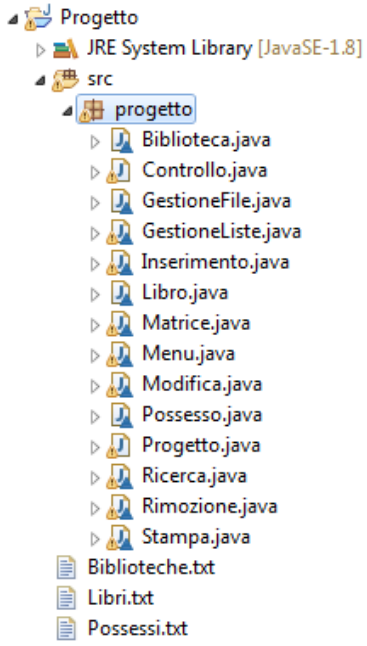


Figura 4.2. Struttura del programma, vista dal software Eclipse

4.3 Implementazione

Dopo aver introdotto anche l'ambiente nel quale è stato sviluppato il programma, passiamo, ora, a vedere del codice implementato in linguaggio Java.

In questo elaborato, non verrà mostrato tutto il codice scritto, ma, in ciascuna delle prossime sezioni, sarà presente una porzione di esso, rappresentante le varie funzioni principali del programma.

Una volta terminato il codice, verrà proposto, anche, un manuale utente, utile a capire, graficamente, come si presenta e come funziona il programma stesso.

4.3.1 Librerie e strutture dati

Iniziamo questa sezione, riportando le librerie utilizzate nel progetto, nonché le strutture dati, ovvero le liste. Oltre alle librerie standard, fornite da Java in maniera automatica, sono state utilizzate due librerie aggiuntive, `java.util` e `java.io`.

La prima, `java.util`, è una libreria che fornisce molteplici funzioni o "utilities" per svariati metodi e classi, come, ad esempio, il formato data, oppure la generazione di numeri casuali. Nel nostro caso, però, tale libreria ci ha permesso di utilizzare il tipo di dato `ArrayList`, per poter gestire e implementare le liste utilizzate, la classe `Scanner`, per gestire gli input da tastiera da parte dell'utente, e la classe `InputMismatchException`, per poter gestire gli errori di inserimento che potrebbero verificarsi.

Queste sono state implementate nel programma grazie alle istruzioni riportate nel Listato 4.3.

```
1   import java.util.ArrayList;
2   import java.util.Scanner;
3   import java.util.InputMismatchException;
```

Listato 4.3. Istruzioni per importare le librerie `java.util` usate

La seconda libreria, invece, che permette di utilizzare classi specifiche per la gestione dei file, è chiamata `java.io`. Tale libreria, come ci dice il nome, fornisce classi e funzioni utili alla gestione delle varie operazioni di input/output. Nel nostro programma, questa libreria è stata inserita all'interno della classe `GestioneFile`, per utilizzare le classi `File`, *FileWriter*, `PrintWriter` e `IOException`.

Le prime tre ci servono per utilizzare il tipo di dato `File` e per poter operare le scritture su di esso. Nello specifico, `FileWriter` è un oggetto che permette la scrittura su file, specificandone il nome e le modalità, mentre `PrintWriter` estende quest'ultimo, gestendo la scrittura come se si utilizzasse un flusso di dati in output, come il `println`, che ci è utile per formattare il file in righe, ciascuna associata ad un singolo oggetto. L'ultima, `IOException`, permette di gestire le eccezioni legate ai file, nel caso in cui si verificassero problemi in fase di operazioni quali apertura, chiusura, lettura o scrittura; essa viene utilizzata nelle istruzioni di `try-catch`, che vedremo successivamente.

Tali librerie vengono importate nel codice come mostrato dnel Listato 4.4.

```
1   import java.io.File;
2   import java.io.FileWriter;
3   import java.io.PrintWriter;
4   import java.io.IOException;
```

Listato 4.4. Istruzioni per imporare le librerie java.io usate

Concludiamo parlando delle strutture dati utilizzate in questo progetto, e già più volte menzionate, ovvero le liste.

Le *liste* sono delle strutture dati dinamiche, utilizzate per contenere un insieme o una collezione di dati omogenei, che permettono molteplici operazioni, tra le quali inserimento, rimozione, ricerca, accesso ad un elemento tramite indice e conteggio degli elementi. Ogni elemento della lista è strutturato in due parti, la prima contenente l'informazione e, la seconda contenente un riferimento all'elemento successivo, usato per scorrere la lista stessa. L'accesso alla lista avviene in maniera sequenziale, partendo dal primo elemento e scorrendo i riferimenti, fino a trovare quello desiderato. A differenza degli array, le liste non hanno una dimensione fissa o stabilita; quest'ultima, infatti, può variare a seconda degli inserimenti e delle rimozioni che vengono effettuate.

Le liste sono strutture molto versatili e comode, per la gestione di un numero anche elevato di elementi, grazie alla loro flessibilità nei tipi di informazioni contenute e alla mancanza di un dimensionamento che ne limita l'utilizzo. La scelta di questo tipo di strutture dati è stata influenzata, in gran parte, proprio da queste loro proprietà.

4.3.2 Classi degli oggetti

Il codice delle classi legate agli oggetti `Biblioteca`, `Libro` e `Possesso`, come abbiamo accennato anche nel capitolo precedente nei diagrammi delle classi (Sezione

```
1   package progetto;
2
3   class Biblioteca {
4   // Attributi della classe
5   String CodBiblio;
6   String NomeBiblio;
7   String CittaBiblio;
8
9   // Metodi della classe
10  void setCod(String Cod) {
11      CodBiblio= Cod;
12  }
13  String getCod() {
14      return CodBiblio;
15  }
16  void setNome(String Nome) {...}
17  String getNome() {...}
18
19  void setCitta(String Citta) {...}
20  String getCitta() {...}
21
22  // Costruttore della classe
23  Biblioteca() {}
24  }
```

Listato 4.5. Codice della classe `Biblioteca`

3.3.2), si limita alla definizione degli attributi e delle funzioni `set` e `get` di questi ultimi. Nel Listato 4.5 riportiamo il codice della classe `Biblioteca`, rappresentante delle altre due per analogia, contenente la dichiarazione degli attributi e di una coppia di funzioni `set` e `get`.

La dichiarazione degli attributi, come si vede nelle prime righe di codice, consiste nella semplice scrittura del tipo di dato, seguito dal nome usato per identificarlo; in questo caso, `String CodBiblio` dichiara una variabile di nome `CodBiblio` e di tipo stringa, e così via. Tutte le altre variabili delle altre classi sono dichiarate in modo analogo.

Per quanto riguarda i metodi, invece, `void setCod(String Cod)` si compone di un'assegnazione, all'attributo `CodBiblio`, della variabile `Cod`, passata come parametro, che contiene l'informazione; mentre, `String getCod()` è un semplice `return CodBiblio` del valore dell'attributo. Gli altri metodi sono analoghi ai due mostrati, e lo stesso vale per i metodi presenti nelle classi `Libro` e `Possesso`.

4.3.3 Inserimento

La classe `Inserimento`, inizia a presentare un codice più denso, e anche più complesso, rispetto a quanto visto il precedenza. Nel Listato 4.6 riportiamo il codice di questa classe, evidenziando, anche qui, l'inserimento di una nuova biblioteca; tutti gli altri inserimenti verranno fatti allo stesso modo.

```
1   package progetto;
2   import java.util.ArrayList;
3   import java.util.InputMismatchException;
4   import java.util.Scanner;
5
6   class Inserimento {
7
8   // Metodi della classe
9   ArrayList<Biblioteca> InsertBiblio(ArrayList<Biblioteca> lista, GestioneFile FileIN) {
10      String sc;
11      String DatiIN;
12      Biblioteca input;
13      Controllo ctrl_cod= new Controllo();
14      Scanner tastiera= new Scanner(System.in);
15      Scanner invio= new Scanner(System.in);
```

```
16      System.out.println("\n Inizio inserimento dati\n");
17      do {
18          input=new Biblioteca();
19          do {
20              System.out.print(" Inserisci il codice della biblioteca: ");
21              input.setCod(tastiera.nextLine());
22              ctrl_cod.setCtrl(ctrl_cod.Ctrl_Input_Biblio(lista, input.getCod()));
23              if(ctrl_cod.getCtrl())
24              System.out.println(" Codice già usato. Scegliere un'altro codice.\n");
25          } while(ctrl_cod.getCtrl());
26          System.out.print("\n Inserisci il nome della biblioteca: ");
27          input.setNome(tastiera.nextLine());
28          System.out.print("\n Inserisci la città della biblioteca: ");
29          String citta= tastiera.nextLine();
30          citta= citta.substring(0,1).toUpperCase() + citta.substring(1).toLowerCase();
31          input.setCitta(citta);
32          DatiIN= input.getCod() + "" + input.getNome() + "" + input.getCitta();
33          FileIN.ScriviFile_app(FileIN.getNomeF_B(),DatiIN);
34          lista.add(input);
35          System.out.println("\n Elemento inserito nel file.");
36          System.out.println(" Vuoi inserire ancora? (S si, N no)");
37          sc= tastiera.nextLine();
38          if(!(sc.equals("S"))&&!(sc.equals("N"))) {
39              do {
40                  System.out.print(" \n Scelta non accettata.\n Inserisci \"S\" per inserire ancora o \"N\" per
                        terminare l'inserimento: ");
41                  sc= tastiera.nextLine();
42                  sc= sc.substring(0).toUpperCase();
43              } while(!(sc.equals("S"))&&!(sc.equals("N")));
44          }
45      } while(sc.equals("S"));
46      System.out.println(" Inserimento dei dati terminato.");
47      System.out.println(" Premi Invio per tornare al menù. . .");
48      invio.nextLine();
49      return lista;
50  }
51  ArrayList<Libro> InsertLibro(ArrayList<Libro> lista, GestioneFile FileIN) {...}
52  ArrayList<Possesso> InsertPossesso(ArrayList<Possesso> listaP, ArrayList<Libro> listaL, ArrayList<Biblioteca>
          listaB, GestioneFile FileIN) {...}
53
54  String InsertCodRicB() {...}
55  String InsertCodRicL() {...}
56  String InsertCittaRic() {...}
57
58  int InsertAnnoRicL() {
59      Scanner tastiera= new Scanner(System.in);
60      int anno;
61      do {
62          anno=0;
63          try {
64              System.out.print("\n Inserisci l'anno di pubblicazione iniziale da cui cercare i libri: ");
65              anno= tastiera.nextInt();
66              if((anno<1453)(anno>2020))
67                  System.out.println(" Anno di pubblicazione non accettato. Inserisci un anno tra il 1453 e il 2020.")
                        ;
68          } catch(InputMismatchException Err) {
69          System.out.println(" Anno non rinosciuto. Inserisci un anno corretto (es. 1789, 1954, 2003).");
70          }
71          tastiera.nextLine();
72      } while((anno<1453)(anno>2020));
73      return anno;
74  }
75  int InsertNCopie() {...}
76
77  Inserimento(){}
78  }
```

Listato 4.6. Codice della classe `Inserimento`

In questa classe non vengono definiti o dichiarati attributi; di conseguenza avremo soltanto metodi. Vediamo il codice relativo al metodo per l'inserimento di una biblioteca, `ArrayList<Biblioteca> InsertBiblio`. A questo metodo vengono passati, come parametri, la lista dei dati e una variabile di tipo `GestioneFile`, per la gestione della scrittura sul file. Nel metodo, tra le variabili dichiarate, notiamo due di tipo `Scanner`, accennate precedentemente e, in questo caso come negli altri visti in precedenza, utilizzate per la gestione degli input da tastiera da parte dell'utente e dell'avanzamento del programma, qualora venga richiesto di premere "Invio" per continuare l'esecuzione.

Tutto l'inserimento è contenuto all'interno di un ciclo `do-while`, che gestisce l'eventuale immissione di più biblioteche, attraverso l'uso della variabile di scelta,

`String Sc`, che viene controllata affinché assuma solo uno di due valori: `S`, come Si, per continuare, oppure `N`, come No, per terminare l'inserimento. In questa immissione, inoltre, viene anche utilizzato il metodo `toUpperCase` delle stringhe, per avere la certezza che `S` e `N` siano inserite, rigorosamente, in maiuscolo.

All'interno di questo ciclo troviamo subito un'altro ciclo `do-while`, che controlla l'inserimento del nuovo codice, tramite la variabile `Controllo ctrl_cod`, che estende l'utilizzo del metodo `Ctrl_Input_Biblio(lista, input.getCod()`, per evitare duplicati. Una volta inseriti tutti gli attributi rimanenti, viene composta, in `String DatiIN`, una stringa formattata nel seguente modo:
`Codice_inserito|Nome_inserito|Città_inserita`

La stringa viene formattata in questo modo per rendere più semplice le operazioni di lettura e scrittura su file, avendo ogni oggetto descritto su quest'ultimo, una riga suddivisa in tre parti, ciascuna legata a un attributo dell'oggetto stesso. Dopo la formattazione, l'oggetto viene inserito nel file e nella lista utilizzata per le operazioni e, subito dopo, viene chiesto se si vuole continuare l'inserimento o meno, ricominciando il ciclo nuovamente o terminandolo, ritornando la lista aggiornata.

Gli altri due inserimenti, quello relativo ad un libro e quello relativo ad un possesso sono analoghi; l'unica precisazione da fare riguarda l'inserimento di un possesso, all'interno del quale viene effettuata un'ulteriore verifica, per controllare che il possesso non sia già presente nella lista, oltre a controllare la presenza dei singoli codici di biblioteca e libro.

I tre metodi successivi, `String InsertCodRicB()`, `InsertCodRicL()` e `InsertCittaRic()` sono dei semplici metodi dove viene richiesto l'inserimento, da parte dell'utente, dei codici da ricercare, effettuando un `return` su tale valore; per questo motivo non sono stati trascritti interamente.

L'ultimo metodo mostrato è `int InsertAnnoRicL()`, utilizzato per mostrare un utilizzo delle istruzioni `try-catch`, per gestire gli errori di formato dell'inserimento. Come si può vedere, infatti, l'inserimento dell'anno di pubblicazione (così come quello del numero di copie), è contenuto tra le istruzioni `try` e `catch` dove, in particolare, viene usato il tipo di dato `InputMismatchException`, per gestire l'eventuale immissione di caratteri letterali in un inserimento di un valore numerico intero, gestendo, quindi, l'eccezione che si crea in fase di esecuzione, richiedendo di inserire nuovamente l'anno e svuotando la variabile `Scanner tastiera` del valore precedentemente acquisito dall'utente.

4.3.4 Rimozione

La prossima classe che vedremo è `Rimozione`, classe omonima al tipo di operazioni che svolge; in questo caso vedremo un esempio di rimozione e anche il metodo per le rimozioni legate ai possessi. Vediamo, nel Listato 4.7, come si presenta il codice.

```
1   package progetto;
2   import java.util.ArrayList;
3   import java.util.Scanner;
4
5   class Rimozione {
6
7   // Metodi della classe
8       boolean Rimuovi_Biblio(ArrayList<Biblioteca> lista, int i){
9       Scanner leggiscelta= new Scanner(System.in);
10      Scanner invio= new Scanner(System.in);
11      String sc;
12      Boolean ctrl=false;
```

```
13      System.out.print(" Rimuovendo la Biblioteca rimuoverai anche i suoi possessi.\n Continuare? (S si, N no)");
14      sc= leggiscelta.nextLine();
15      if(!(sc.equals("S"))&&!(sc.equals("N"))) {
16      do {
17          System.out.print(" \n Scelta non accettata. Inserisci \"S\" per confermare o \"N\" per annullare la
                    rimozione: ");
18          sc= leggiscelta.nextLine();
19          sc=sc.substring(0).toUpperCase();
20      } while(!((sc.equals("S")))&&!((sc.equals("N"))));
21      }
22      if(sc.equals("S")) {
23          ctrl=true;
24          lista.remove(i);
25          System.out.println(" Biblioteca e relativi possessi rimossi.");
26          System.out.println(" Premi invio per continuare. . .");
27          invio.nextLine();
28      }
29      else {
30          ctrl=false;
31          System.out.println(" Operazione annullata. Biblioteca NON rimossa.");
32          System.out.println(" Premi invio per tornare al menù. . .");
33          invio.nextLine();
34      }
35      return ctrl;
36  }
37
38  boolean Rimuovi_Libro(ArrayList<Libro> lista, int i){...}
39  boolean Rimuovi_Possesso(ArrayList<Possesso> lista, int i){...}
40
41  ArrayList<String> RimuoviPoss_Biblio(ArrayList<String> lista, String CodB) {
42      String [] parti;
43      ArrayList<String> listaagg= new ArrayList<String>();
44      for(int i=0;i<lista.size();i++) {
45          parti=lista.get(i).split("\\|");
46          if(!parti[0].equals(CodB)) {
47              listaagg.add(lista.get(i));
48          }
49      }
50      return listaagg;
51  }
52  ArrayList<String> RimuoviPoss_Libro(ArrayList<String> lista, String CodL) {...}
53
54  Rimozione(){}
55  }
```

Listato 4.7. Codice della classe `Rimozione`

Prendiamo, anche qui, la rimozione di una biblioteca come esempio, col metodo `boolean Rimuovi_Biblio(ArrayList<Biblioteca> lista, int i)`.

In questo caso, non viene chiesto alcun input su cosa rimuovere, in quanto al metodo viene passata la variabile `int i`, che è l'indice dell'elemento della lista trovato grazie ad una ricerca, effettuata dalla classe omonima, che vedremo a breve, e che poi verrà eventualmente rimosso grazie ai metodi della classe `ArrayList` utilizzata. Il metodo chiede, poi, conferma sulla conseguente rimozione dei possessi associati alla biblioteca che si vuole rimuovere. Anche in questo caso viene utilizzata una variabile di scelta che può assumere valori `S` o `N`, opportunamente formattati in maiuscolo con il metodo `toUpperCase`. Qualora venga confermata la rimozione, viene eliminata la biblioteca dalla lista opportuna e il metodo termina, ritornando al programma una variabile di tipo booleano, con l'esito della rimozione.

La rimozione dei possessi legati a tale biblioteca rimossa, invece, viene gestita dal metodo `ArrayList<String> RimuoviPoss_Biblio(ArrayList<String> lista, String CodB)`, che, nel corpo del programma, verrà richiamato dopo la rimozione precedente. In questo metodo, quello che succede è semplicemente un ciclo `for`, sulla lista dei possessi in formato stringa e non oggetto[1], dove viene controllata

[1] La scelta della lista in formato `String` è data dalla necessità dell'aggiornamento sul file. Ciò è dovuto al fatto che la lista dei possessi viene letta direttamente da file quando c'è una rimozione, e viene fornita in formato `String` dal metodo di lettura. Si è, quindi, evitato di effettuare due conversioni, da stringa a oggetto `Possesso` e viceversa, per l'aggiornamento, lasciandola in formato `String` per le operazioni.

la corrispondenza del codice della biblioteca eliminata, e viene creata una nuova lista, senza gli elementi che presentano quest'ultimo, da ritornare poi al programma, che provvederà a riscriverla sul file. Gli altri metodi che si vedono sono analoghi ai due descritti; di conseguenza, essi non vengono mostrati per non appesantire l'elaborato.

4.3.5 Modifica

Proseguiamo, vedendo la classe `Modifica`, con i suoi due metodi per la modifica del numero delle copie e del nome di una biblioteca.

Il codice della classe viene riportato nel Listato 4.8.

```
1   package progetto;
2   import java.util.ArrayList;
3   import java.util.InputMismatchException;
4   import java.util.Scanner;
5
6   class Modifica {
7
8   // Metodi della classe
9   ArrayList<Possesso> Mod_Poss(ArrayList<Possesso> lista, int i) {
10      Scanner leggi= new Scanner(System.in);
11      Scanner invio= new Scanner(System.in);
12      Possesso info= new Possesso();
13      System.out.println(" Numero di copie attuale: " +lista.get(i).getN_Copie());
14      do {
15          info.setN_Copie(0);
16          try {
17              System.out.print("\n Inserisci il numero di copie aggiornato (max 50): ");
18              info.setN_Copie(leggi.nextInt());
19              if((info.getN_Copie()<1)(info.getN_Copie()>50))
20              System.out.println(" Numero di copie non valido. Inserisci un numero compreso tra 1 e 50.");
21          } catch(InputMismatchException Err) {
22              System.out.println(" Numero non riconosciuto. Inserisci un numero corretto (es. 1, 2, 5, 10).");
23          }
24          leggi.nextLine();
25      } while((info.getN_Copie()<1)(info.getN_Copie()>50));
26      info.setCodB(lista.get(i).getCodB());
27      info.setCodL(lista.get(i).getCodL());
28      lista.set(i, info);
29      System.out.println(" Numero di copie aggiornato.");
30      System.out.println(" Premi invio per continuare. . .");
31      invio.nextLine();
32      return lista;
33  }
34  ArrayList<Biblioteca> Mod_Biblio(ArrayList<Biblioteca> lista, int i) {...}
35
36  Modifica(){}
37  }
```

Listato 4.8. Codice della classe `Modifica`

Dei due metodi presenti, `ArrayList<Biblioteca> Mod_Biblio(...)` viene tralasciato in quanto analogo a quello che descriveremo ora.

Nel metodo `ArrayList<Possesso> Mod_Poss(ArrayList<Possesso> lista, int i)`, come nel caso della rimozione, non viene chiesto alcun input sul dato da modificare, in quanto questo viene fatto dalla ricerca, nel corpo del programma, che poi viene passato alla classe, tramite la variabile `int i`. Dopo aver mostrato le copie attuali, il metodo chiede l'inserimento del numero di copie aggiornato, gestito con le istruzioni `try-catch`, viste nel caso dell'inserimento (Sezione 4.3.3), per evitare di generare errori nel caso di inserimento di caratteri letterali. Inserito correttamente il numero, vengono estratti dalla lista i codici di biblioteca e libro, legati al possesso cercato, e uniti, all'interno della variabile `Possesso info`, dichiarata nel metodo. Tali codici vengono reinseriti nella lista, col metodo `lista.set(i, info)`, offerto dalla classe `ArrayList`. Infine, viene ritornata la lista aggiornata, che verrà, poi, riscritta sul file al termine delle operazioni.

4.3.6 Ricerca

La classe `Ricerca` contiene, al proprio interno, i metodi dei diversi tipi di ricerche effettuate dal programma nei vari casi. Data la corposità della classe, vedremo soltanto due dei metodi al suo interno, uno di quelli più semplici e uno di quelli più complessi.

Parte del codice di questa classe viene riportato nel Listato 4.9.

```
1   package progetto;
2   import java.util.ArrayList;
3   import java.util.Scanner;
4
5   class Ricerca {
6   // Attributi
7   ArrayList<Biblioteca> listaRicBiblio= new ArrayList<Biblioteca>();
8   ArrayList<Libro> listaRicLibri= new ArrayList<Libro>();
9
10  // Metodi della classe
11  void setListaB(ArrayList<Biblioteca> lista) {...}
12  ArrayList<Biblioteca> getListaB() {...}
13  void setListaL(ArrayList<Libro> lista) {...}
14  ArrayList<Libro> getListaL() {...}
15
16  int Ricerca_Biblio(ArrayList<Biblioteca> lista, String CodRic) {
17      Scanner invio= new Scanner(System.in);
18      int ctrl=0;
19      int win=0;
20      for(int i=0;i<lista.size();i++) {
21          if(lista.get(i).getCod().equals(CodRic)) {
22              win=i;
23              i=lista.size();
24              System.out.println("\n Biblioteca Trovata.");
25              System.out.println(" Premi invio per continuare. . .");
26              invio.nextLine();
27          }
28          else {ctrl++;}
29      }
30      if(ctrl==lista.size()) {
31          win=-1;
32          System.out.println("\n Nessuna Biblioteca Trovata.");
33          System.out.println(" Premi invio per tornare al menù. . ..\n");
34          invio.nextLine();
35      }
36      return win;
37  }
38  int Ricerca_Libro(ArrayList<Libro> lista, String CodRic) {...}
39  int Ricerca_Possesso(ArrayList<Possesso> lista, String CodRicB, String CodRicL) {...}
40  ArrayList<Libro> Ricerca_LibriAnno(ArrayList<Libro> lista, int AnnoRic, String verso) {...}
41  ArrayList<Biblioteca> Ricerca_BiblioCitta(ArrayList<Biblioteca> lista, String CittaRic){...}
42
43  ArrayList<Biblioteca> Ricerca_BiblioconLibri_Matrice(int [][] matrice, ArrayList<Biblioteca> listaB, ArrayList<
        Libro> listaL, int indice_B) {
44      int i, j, k;
45      ArrayList<Integer> indicilibri= new ArrayList<>();
46      Controllo verificaAnno= new Controllo();
47      Scanner invio= new Scanner(System.in);
48      listaRicBiblio.clear();
49      try {
50          if(indice_B!=-1) {
51              for(j=0; j< listaL.size(); j++) {
52                  if(matrice[indice_B][j]!=0) {
53                      if(verificaAnno.Ctrl_Anno_Libro(listaL.get(j), 1500, 0, "<")) {
54                          indicilibri.add(j);
55                      }
56                  }
57              }
58              if(indicilibri.size()==0) {
59                  ...       // Stampa del risultato negativo
60              }
61              else {
62                  ...       // Stampa del risultato positivo
63              // Ricerca delle biliotheche che hanno tutti i libri della biblioteca specificata
64                  for(i=0; i< listaB.size(); i++) {
65                      int count=0;
66                      if(i==indice_B)
67                          i++;
68                      for(j=0; j< listaL.size(); j++) {
69                          if(matrice[i][j]!=0) {
70                              for(k=0; k< indicilibri.size(); k++) {
71                                  if(j==indicilibri.get(k))
72                                      count++;
73                              }
74                          }
75                      }
76                      if(count==indicilibri.size())
77                          listaRicBiblio.add(listaB.get(i));
78                  }
```

```
79                  if(listaRicBiblio.size()==0) {
80                      ...       // Stampa del risultato
81                  }
82                  else {
83                      ...       // Stampa del risultato
84                  }
85              }
86          }
87      } catch(NullPointerException Err) {
88          ...       //Stampa del l'errore
89      }
90      return listaRicBiblio;
91  }
92  ArrayList<Biblioteca> Ricerca_BiblioMaxMin_Matrice(int [][] matrice, ArrayList<Biblioteca> listaB, ArrayList<Libro
        > listaL){...}
93
94  Ricerca(){}
95  }
```

Listato 4.9. Prima parte del codice della classe `Ricerca`

In questa classe, a differenza delle classi precedenti, diverse da quelle relative agli oggetti, abbiamo la dichiarazione di due attributi, `listaRicBiblio` e `listaRicLibri`, entrambi di tipo `ArrayList`, che servono a contenere i dati delle ricerche che si effettueranno. Come nei casi precedenti, se ci sono attributi, vengono sempre definite coppie di funzioni `set` e `get` per la loro interazione fuori dalla classe stessa.

Il metodo "semplice", che abbiamo citato sopra, effettua la ricerca dei dati di un determinato oggetto, sia esso della classe `Biblioteca`, `Libro` o `Possesso`, attraverso il suo codice, specificato dall'utente. Il metodo che vedremo è `int Ricerca_Biblio(ArrayList<Biblioteca> lista, String CodRic)`, mentre, gli altri due sono tralasciati in quanto analoghi.

All'interno del metodo, dopo aver definito e inizializzato due variabili, `int ctrl=0` per il controllo, e `int win=0` per il risultato della ricerca, viene utilizzato un ciclo `for` per scorrere la lista delle biblioteche e, grazie ad un costrutto `if` al suo interno, viene confrontato il codice nella lista con quello contenuto nella variabile `String CodRic`, passato come parametro alla funzione che contiene il codice espresso dall'utente.

Qualora si abbia un riscontro positivo del confronto, essendo i codici univoci, viene salvato in `win` l'indice della lista, contenente i dati della biblioteca cercata. Successivamente con `i=lista.size()`, si fa avanzare l'indice della lista, così da far terminare il ciclo `for`. Qualora, invece, il codice sia differente, nel costrutto `else`, viene aumentato di uno il valore di `ctrl`. Usciti dal ciclo `for`, se tale variabile è pari al numero di elementi presenti nella lista, vorrà dire che la biblioteca cercata non è presente nel sistema. In questo caso, si esegue l'istruzione `win=-1`, che servirà, nel corpo del programma, ad indicare che la ricerca non ha fornito alcun risultato. In caso di riscontro, invece, verrà ritornato l'indice della lista contenente i dati della biblioteca trovata, presente in `win`.

I due metodi `ArrayList<Libro> Ricerca_LibriAnno(...)` e `ArrayList<Biblioteca> Ricerca_BiblioCitta(...)`, sebbene contengano delle funzioni utili in più punti del programma, data la loro somiglianza con gli altri metodi mostrati, sono stati tralasciati, per non appesantire ulteriormente l'elaborato.

L'altro metodo mostrato è legato ad una delle funzionalità "complesse" del programma, ovvero la ricerca delle biblioteche che possiedono tutti i libri, pub-

blicati dopo un determinato anno, posseduti dalla biblioteca specificata dall'utente, utilizzando la matrice dei possessi.

Il metodo, oltre a definire diversi tipi di variabili, tra cui una lista di interi per gli indici e, anche, un oggetto di tipo `Controllo`, altra classe implementata dal programma, è completamente contenuto all'interno di un costrutto `try-catch`, che consente di evitarne l'utilizzo, qualora la matrice non sia stata prima inizializzata, o abbia errori. All'interno di queste istruzioni, troviamo due blocchi principali, caratterizzati da cicli `for`, e intramezzati da costrutti `if` ed `else` per controllare i risultati ottenuti.

Il primo ciclo `for` è responsabile dell'estrazione di tutti i libri della biblioteca specificata dall'utente, in questo caso da `indice_B`, risultato di una ricerca fatta precedentemente, che soddisfano le condizioni dell'anno di pubblicazione, controllate con l'utilizzo dell'oggetto `Controllo verificaAnno` e del suo metodo `Ctrl_Anno_Libro(...)`, all'interno dell'`if`.

Gli indici dei libri che soddisfano le condizioni vengono, quindi, inseriti all'interno di `ArrayList<Integer> indicilibri` e, se ne è stato trovato almeno uno, si passa al secondo ciclo `for` del programma, dove viene fatta scorrere la matrice dei possessi, grazie ad un'altro ciclo `for` interno a quello appena citato, utilizzando le liste delle biblioteche e dei libri per controllare gli indici della matrice. Per ogni elemento di ogni riga della matrice viene, quindi, controllata la corrispondenza tra gli indici, salvati precedentemente, e l'indice del libro nella matrice, aumentando il contatore `int count`, in caso di uguaglianza tra i due.

Terminati tutti i libri, per una singola riga si verifica se il contatore è pari alla dimensione della lista degli indici utilizzata; questo permetterà di confermare che la biblioteca, legata a quella riga, rispetti tutte le condizioni della ricerca, e può quindi essere salvata, all'interno di `listaRicBiblio`, per venire, poi, stampata a video, una volta terminato il metodo.

L'ultimo metodo rimasto nella classe, ovvero `Ricerca_BiblioMaxMin_Matrice(...)`, è simile a quello visto ora, quanto a struttura dei cicli e dei controlli effettuati (anche qui con un oggetto di tipo `Controllo`); di conseguenza, si è scelto di non mostrarlo, per non risultare ripetitivi.

4.3.7 Gestione delle Liste

La prossima classe è quella che si occupa della gestione delle liste; anche per quanto riguarda questa classe mostreremo solo alcuni dei metodi principali.

Iniziamo mostrando il codice della classe `GestioneListe`, riportato nel Listato 4.10.

```
1   package progetto;
2   import java.util.ArrayList;
3   import java.util.Scanner;
4
5   class GestioneListe {
6
7   //Metodi della classe
8   ArrayList<Biblioteca> Converti_Str_a_Bibl(ArrayList<String> listaString) {
9       Biblioteca Info;
10      ArrayList<Biblioteca> listaBiblio= new ArrayList<Biblioteca>();
11      for(int i=0; i< listaString.size(); i++) {
12          String [] parti= listaString.get(i).split("\\|");
13          Info= new Biblioteca();
14          Info.setCod(parti[0]);
15          Info.setNome(parti[1]);
16          Info.setCitta(parti[2]);
```

```
17          listaBiblio.add(Info);
18      }
19      return listaBiblio;
20  }
21  ArrayList<String> Converti_Bibl_a_Str(ArrayList<Biblioteca> listaBiblio) {...}
22  ArrayList<Libro> Converti_Str_a_Lib(ArrayList<String> listaString) {...}
23  ArrayList<String> Converti_Lib_a_Str(ArrayList<Libro> listaLibri) {...}
24  ArrayList<Possesso> Converti_Str_a_Poss(ArrayList<String> listaString) {...}
25  ArrayList<String> Converti_Poss_a_Str(ArrayList<Possesso> listaPossessi) {...}
26
27  ArrayList<Libro> Estrai_Lib_da_Poss(ArrayList<Possesso> listaP, ArrayList<Libro> listaL, ArrayList<Biblioteca>
        listaR_Biblio, int nCopie) {
28      ArrayList<Libro> listalibri= new ArrayList<Libro>();
29      int i, j, k;
30      // Caso generale, senza ricerca per numero di copie
31      if(nCopie==0) {...}
32      //Caso particolare, con ricerca per numero di copie maggiore uguale a un numero specificato
33      if(nCopie>0) {
34          listalibri.clear();
35          for(i=0; i< listaP.size(); i++) { // Scorro la lista dei possessi
36              for(j=0; j< listaR_Biblio.size(); j++) { //Scorro la lista delle biblioteche trovate
37                  if((listaP.get(i).getCodB().equals(listaR_Biblio.get(j).getCod()))&&(listaP.get(i).getN_Copie()
                        >=nCopie)) {
38                      for(k=0; k< listaL.size(); k++) { //Scorro la lista dei libri
39                          if(listaP.get(i).getCodL().equals(listaL.get(k).getCod())) {
40                              listalibri.add(listaL.get(k));
41                          }
42                      }
43                  }
44              }
45          }
46          listalibri= this.Rimuovi_Libri_Duplicati(listalibri);
47          if(listalibri.size()==0)
48              System.out.println("\n Nessun libro trovato nelle biblioteche considerate, con " + nCopie + " o più
                    copie.");
49          else
50              System.out.println("\n Trovati " + listalibri.size() + " libri nelle biblioteche di " + listaR_Biblio.
                    get(0).getCitta() + " con " + nCopie + " o più copie.");
51          System.out.print(" Premi invio per continuare. . .");
52          Scanner invio= new Scanner (System.in);
53          invio.nextLine();
54      }
55      return listalibri;
56  }
57  ArrayList<Libro> Rimuovi_Libri_Duplicati(ArrayList<Libro> lista){...}
58
59  ArrayList<Libro> Libri_Esclusivi(ArrayList<Possesso> listaP, ArrayList<Biblioteca> listaB, ArrayList<Libro>
        listaRicL, String cittaRic){
60      int i, j, k;
61      for(i=0; i<listaP.size(); i++) {
62          for(j=0; j< listaRicL.size(); j++) {
63              if(listaP.get(i).getCodL().equals(listaRicL.get(j).getCod())) {  // Trovo tutti i possessi legati ai
                        singoli libri
64                  for(k=0; k< listaB.size(); k++) {
65                      if(listaP.get(i).getCodB().equals(listaB.get(k).getCod())) { // Trovo i dati della biblioteca
                                legata al possesso
66                          if(!(listaB.get(k).getCitta().equals(cittaRic))) {
67                              listaRicL.remove(j);
68                          }
69                      }
70                  }
71              }
72          }
73      }
74      if(listaRicL.size()==0)
75          System.out.println(" Dei libri trovati, nessuno si trova Esclusivamente nelle biblioteche di " + cittaRic);
76      else
77          System.out.println(" Di tutti i libri trovati, " + listaRicL.size() + " sono Esclusivi delle biblioteche di
                " + cittaRic);
78      System.out.print(" Premi invio per continuare. . .");
79      Scanner invio= new Scanner (System.in);
80      invio.nextLine();
81
82      return listaRicL;
83  }
84  ArrayList<Biblioteca> Estrai_2_Biblio(ArrayList<Biblioteca> lista) {...}
85
86  GestioneListe(){}
87  }
```

Listato 4.10. Codice della classe `GestioneListe`

Questa classe è quella con più metodi, alcuni dei quali specifici per delle funzionalità complesse del programma. I primi metodi che incontriamo sono quelli per la conversione della lista dal tipo oggetto al tipo stringa, e viceversa. Per ciò che riguarda tali metodi, riporteremo soltanto il codice di `ArrayList<Biblioteca> Converti_Str_a_Bibl(ArrayList<String> listaString)` per via delle analogie

tra le coppie.

Il metodo non è molto complesso; infatti, si compone di un ciclo `for`, all'interno del quale viene letta la lista di stringhe, contenente tutti i dati che, grazie al metodo `listaString.get(i).split("\\|")`, vengono separati, in base al carattere | che li divide, per poi essere inseriti in una variabile di tipo oggetto, che viene, infine, inserita nella lista di tipo oggetto che sarà ritornata dalla funzione. Il metodo per la conversione inversa è analogo ed è stato, quindi, tralasciato.

Vediamo, ora, gli altri due metodi mostrati nel codice, ovvero `Estrai_Lib_da_Poss` e `Libri_Esclusivi`, che implementano funzionalità diverse, ma che vengono utilizzate insieme, per una specifica funzionalità. I nomi dei metodi permettono di capire, a grandi linee, le operazioni che svolgono; il primo serve per estrarre tutti i libri dei vari possessi, contenuti nel sistema, tramite i codici delle biblioteche, mentre il secondo serve a trovare i libri di proprietà esclusiva di una biblioteca specifica.

Il primo metodo è diviso, da due costrutti `if`, in due casi di utilizzo, qualora si abbia bisogno del numero di copie o meno. Il codice è pressoché identico, salvo per questo dettaglio delle copie, che abbiamo scelto di mostrare anche nel codice. All'interno del costrutto `if`, abbiamo una serie di cicli `for` annidati, con opportuni altri `if`, per effettuare dei controlli sui dati; in particolare, e nell'ordine, vengono fatte scorrere la lista dei possessi `listaP`, seguita da quella delle biblioteche ricercate `listaR_Biblio`, passata come parametro alla funzione, nella quale vengono confrontati i codici di queste ultime, con quelli presenti nei possessi e controllato il numero di copie, con quello specificato, con l'istruzione: `if((listaP.get(i).getCodB().equals(listaR_Biblio.get(j).getCod()))&& (listaP.get(i).getN_Copie()>=nCopie))`.
Successivamente, se questo confronto risulta positivo, viene fatta scorrere la lista dei libri `listaL`, per estrarre il libro corrispondente e inserirlo in `listalibri`, che sarà poi ritornata dal metodo. Prima di controllare se sono stati trovati libri o meno, con `if(listalibri.size()==0)`, e prima di stampare il risultato, viene chiamato il metodo `this.Rimuovi_Libri_Duplicati(listalibri)`, interno alla classe, che va ad eliminare eventuali libri posseduti da più biblioteche, e quindi duplicati nella ricerca. Il codice di quest'ultimo metodo è stato tralasciato in quanto piuttosto semplice e ritenuto meno interessante.

Concludiamo con l'ultimo metodo mostrato rimasto, ovvero `Libri_Esclusivi`, che notiamo essere molto simile, a livello di struttura, a quello appena descritto. Anch'esso presenta una serie di cicli `for` annidati, dove, dopo aver letto le liste dei possessi, dei libri ricercati e delle biblioteche, controlla, con `if(!(listaB.get(k).getCitta().equals(cittaRic)))`, se la città della biblioteca è diversa da quella specificata dall'utente in `cittaRic`, e, in caso positivo, rimuove il libro corrispondente dalla lista della ricerca, in quanto non soddisfa le condizioni. Il costrutto `if-else` finale, stampa sul video il risultato della ricerca, a seconda che la variabile `listaRicL` presenti elementi al suo interno o meno e, ritornandola, per l'eventuale stampa a video.

4.3.8 Matrice

La classe `Matrice` è l'ultima ad essere stata implementata ma, nonostante questo, contiene alcuni di metodi utili alle ultime funzionalità del nostro sistema, nonché l'unico metodo ricorsivo presente, che ci è sembrato giusto mostrare.

Mostriamo, nel Listato 4.11, il codice della classe e dei metodi che descriveremo a breve.

```
1   package progetto;
2   import java.util.ArrayList;
3   import java.util.Scanner;
4
5   class Matrice {
6   // Attributi della classe
7   int [][] Mat_Poss;
8
9   // Metodi della classe
10  void setMatP(int [][] matrice) {...}
11  int [][] getMatP(){...}
12
13  int [][] Crea_Matrice(ArrayList<Biblioteca> listaB, ArrayList<Libro> listaL, ArrayList<Possesso> listaP){
14      ...
15      Mat_Poss[i][j]=this.Popola_Mat(listaP, listaB.get(i).getCod(), listaL.get(j).getCod());
16      ...
17  }
18
19  int Popola_Mat(ArrayList<Possesso> lista, String CodB, String CodL) {
20      int copie;
21      copie=0;
22      for(int i= 0; i<lista.size(); i++) {
23          if((lista.get(i).getCodB().equals(CodB))&&(lista.get(i).getCodL().equals(CodL))) {
24              copie= lista.get(i).getN_Copie();
25              i=lista.size();
26          }
27      }
28      return copie;
29  }
30  void Confronta_Volumi(ArrayList<Biblioteca> listaB, ArrayList<Libro> listaL, String codB1, String codB2) {...}
31
32  int Conta_Copie(int [][] matrice, int I, int J, int limit_j) {
33          int copie=0;
34          if(J!=limit_j) {
35                  copie=matrice[I][J]+Conta_Copie(matrice, I,(J+1),limit_j);
36                  return copie;
37          }
38          else {
39                  return copie;
40          }
41  }
42
43  Matrice(){}
44  }
```

Listato 4.11. Codice della classe `Matrice`

Dal nome della classe si può capire come al proprio interno, sia presente l'attributo che rappresenta la matrice all'interno del programma, chiamato `Mat_Poss` ad indicare la matrice dei possessi, e, di conseguenza, anche i metodi `set` e `get` relativi a quest'ultimo.

Oltre a questi due, abbiamo altri due metodi su cui vogliamo porre l'attenzione, ovvero `int Popola_Mat(ArrayList<Possesso> lista, String CodB, String CodL)`, per il popolamento degli elementi della matrice, e `int Conta_Copie(int [][] matrice, int I, int J, int limit_j)`, ovvero il metodo ricorsivo per il conteggio di tutti i volumi di una biblioteca.

Iniziamo con `Popola_Mat` che notiamo essere richiamato all'interno del metodo `int [][] Crea_Matrice(...)`, in quanto, entrambi responsabili della costruzione della matrice. Il metodo si occupa di scorrere la lista dei possessi, con un ciclo `for`, e, aiutandosi con il costrutto `if` presente, assegna alla variabile `int copie`, il numero esatto di copie del libro, il cui codice è specificato in `String CodL`, presente nella biblioteca, il cui codice è, invece, specificato in `String CodB`, entrambe variabili

presenti tra i parametri della funzione. Il metodo ritorna il numero di copie, che può essere "0", se la combinazione dei codici libro-biblioteca non è presente nei possessi, oppure l'effettivo numero di copie in possesso; questo valore intero, tramite la chiamata all'interno di `int [][] Crea_Matrice(...)`, verrà inserito all'interno della matrice, nella posizione ottenuta intersecando la riga della biblioteca e la colonna del libro.

Analizziamo, ora, il metodo ricorsivo, iniziando con il dire cos'è una funzione ricorsiva e perché si definisce tale.

Una funzione ricorsiva è una funzione che richiama se stessa al proprio interno e che ha termine grazie a dei parametri che, nel corso dell'esecuzione, cambiano, fino a soddisfare le condizioni di fine. L'esempio più comune, per descrivere la ricorsione, è il fattoriale[2] di un numero naturale n.

Nel nostro caso, all'interno di `int Conta_Copie(...)`, la funzione è richiamata per sommare i numeri delle copie dei vari libri, presenti in `matrice[I][J]`, fino ad arrivare alla fine di questi, indicata dalla variabile `int limit_j` che, rappresenta la condizione di uscita dalla ricorsione, controllata nel costrutto `if` iniziale.

Questo metodo verrà richiamato all'interno di `void Confronta_Volumi(...)`, dove avverrà, anche, la selezione delle biblioteche da confrontare, e verrà, altrimenti, fornito il risultato di tale confronto, espresso come "numero di copie possedute in più".

4.3.9 Classe Progetto

Finora, in questo elaborato, non è mai stata nominata una `classe Progetto`, in quanto quest'ultima contiene soltanto il corpo principale del programma, ovvero il metodo `public static void main(String[] args)`.

In Java non è possibile definire metodi o istruzioni fuori da una classe, così come non è possibile scrivere istruzioni che non siano presenti all'interno delle dichiarazioni dei metodi stessi. Persino nei programmi più semplici, come, ad esempio, il classico `Hello World`, deve essere sempre presente, all'interno di una classe, un metodo specifico che permette al programma, una volta compilato, di avere un punto di partenza, da cui iniziare l'esecuzione, che è rappresentato dal metodo, comunemente chiamato `main`. L'altra particolarità di Java è che, per convenzione, il nome della classe deve coincidere con il nome del file `.java` che ne conterrà il codice e, cosa non meno importante, ad ogni file deve essere associata un'unica classe.

Nasce, così, la classe `Progetto`, che conterrà il `main` del nostro programma, con tutte le chiamate ai vari metodi delle classi, descritte nei precedenti capitoli, che permetterà l'esecuzione del programma in tutti i suoi aspetti.

Il codice della classe è composto unicamente di chiamate ai metodi delle classi, gestiti grazie ai costrutti `if-else` e `switch-case`, per definire le scelte dei menù, oltre alla dichiarazione di tutti gli oggetti, uno per ogni classe, da utilizzare. Di questa classe, non verrà mostrato il codice, ma soltanto alcune righe di esso (Listato 4.12), per comprendere come le chiamate ai metodi sono state concatenate tra loro ed utilizzate, anche in base ai tipi di questi ultimi.

[2] Si lascia la definizione di fattoriale, qualora non la si conoscesse, ad altri testi, che trattano l'analisi numerica e il calcolo delle probabilità.

```
1
2       // Chiamate semplici
3       Sc2=menu_prog.menu_biblio();
4
5       listaBiblio= Input.InsertBiblio(listaBiblio, manageFile);
6
7       manageFile.creafile(manageFile.getNomeF_B());
8
9       // Chiamate annidate, più complesse
10      manageFile.ScriviFile_over(manageFile.getNomeF_B(), manageList.Converti_Bibl_a_Str(listaBiblio));
11
12      Search.setListaL(Search.Ricerca_LibriAnno(listaLibri, Input.InsertAnnoRicL(), ">"));
13
14      ArrayList<Libro> libriconCopie= new ArrayList<Libro>(manageList.Estrai_Lib_da_Poss(listaPossessi, listaLibri,
              Search.getListaB(), Input.InsertNCopie()));
15
16      Search.setListaB(Search.Ricerca_BiblioconLibri_Matrice(matrix.getMatP(), listaBiblio, listaLibri, Search.
              Ricerca_Biblio(listaBiblio, Input.InsertCodRicB())));
17
18      Search.setListaB(Search.Ricerca_BiblioMaxMin_Matrice(matrix.getMatP(), listaBiblio, listaLibri));
```

Listato 4.12. Codice di alcune chiamate effettuate nel metodo `main` del programma

In generale, la chiamata ad un metodo di una classe, attraverso un suo oggetto, è caratterizzata da una forma del seguente tipo:

`Nome-Oggetto.Nome-Metodo(parametro1, parametro2,...);`

In tutti gli esempi riportati sopra, per quanto a primo impatto non sembri, questa forma è sempre rispettata.

Come vediamo nelle chiamate semplici, ad esempio nella prima, abbiamo l'oggetto `menu_prog` che richiama il metodo `menu_biblio()`, assegnando il risultato di quest'ultimo alla variabile `Sc2`. Questa semplice chiamata permette di invocare dalla classe Menù, il metodo per stampare il menù per la gestione delle biblioteche e assegnare la scelta fatta dall'utente alla variabile specificata nel `main`, con cui, poi, gestire lo switch-case associato.

Se, invece, prendiamo in considerazione la prima del secondo gruppo di chiamate, definite annidate, notiamo che l'oggetto `manageFile` richiama il metodo `ScriviFile_Over(...)` dove, al posto dei parametri, come li abbiamo visti finora nei vari codici, troviamo altre chiamate a metodi, che forniranno il nome del file (`manageFile.getNomeF_B()`) e la lista delle biblioteche (`manageList.Converti_Bibl_a_Str(listaBiblio)`); questi saranno poi utilizzati per effettuare la scrittura del file delle biblioteche.

La logica mostrata in questi due esempi è facilmente estendibile anche alle altre chiamate qui mostrate, così come a tutte le altre presenti all'interno del codice

4.4 Manuale Utente

Quest'ultima sezione del capitolo è dedicata a mostrare, grazie all'utilizzo di alcuni screenshot, presi direttamente dall'esecuzione del programma, le varie funzionalità che quest'ultimo propone all'utente, creando una guida all'utilizzo delle funzioni principali offerte.

Prima di iniziare, si tiene a precisare che alcune delle funzionalità di alcuni menù, nello specifico quelle del menù dei Libri e dei Possessi, non vengono mostrate nelle immagini seguenti, in quanto identiche a quelle del menù delle Biblioteche.

Menù Generale

Iniziamo a mostrare, in Figura 4.3, quello che l'utente vede all'inizio dell'esecuzione del programma; continueremo, poi, mostrando, una per volta, tutte le scelte presenti.

```
Biblioteche.txt già presente per l'utilizzo.
Libri.txt già presente per l'utilizzo.
Possessi.txt già presente per l'utilizzo.
 Lettura dei dati da file completata.
 Lettura dei dati da file completata.
 Lettura dei dati da file completata.

   ### Programma Gestionale per Biblioteche ###

  Scegli cosa vuoi gestire

   1. Gestione Biblioteche;
   2. Gestione Libri;
   3. Gestione Possessi;
   4. Altre operazioni;
   0. Esci dal programma;

  Inserisci il numero della tua scelta:
```

Figura 4.3. Inizio dell'esecuzione del programma, menù principale

Partiamo, perciò, dalla scelta "1", mostrata in Figura 4.4.

```
  ### Programma Gestionale per Biblioteche ###

 Scegli cosa vuoi gestire

  1. Gestione Biblioteche;
  2. Gestione Libri;
  3. Gestione Possessi;
  4. Altre operazioni;
  0. Esci dal programma;

Inserisci il numero della tua scelta: 1

  ### Gestione Biblioteche ###

 Scegli cosa vuoi fare

  1. Inserisci una nuova biblioteca nel sistema;
  2. Rimuovi una biblioteca, usando il suo codice;
  3. Modifica il nome di una biblioteca, usando il suo codice;
  4. Ricerca la presenza di una biblioteca, usando il suo codice;
  5. Stampa informazioni di tutte le biblioteche presenti;
  0. Torna al menù precedente;

Inserisci il numero della tua scelta:
```

Figura 4.4. Scelta "1" del menù iniziale

Scelte del menù per la gestione delle Biblioteche

All'interno del menù per la gestione delle Biblioteche troviamo cinque scelte, escludendo la numero "0", che fa tornare al menù precedente. Le videate corrispondenti a tali scelte verranno mostrate nelle Figure 4.5-4.9.

```
Inserisci il numero della tua scelta: 1

  Inizio inserimento dati

Inserisci il codice della biblioteca: 24

Inserisci il nome della biblioteca: Biblioteca civica Romolo Spezioli

Inserisci la città della biblioteca: Fermo
Scrittura completata. Dati aggiunti al file

Elemento inserito nel file.
Vuoi inserire ancora? (S si, N no)
N
Inserimento dei dati terminato.
Premi Invio per tornare al menù. . .
```

Figura 4.5. Scelta "1" del menù per la gestione delle Biblioteche

```
Inserisci il numero della tua scelta: 2
Inserisci il codice della Biblioteca da cercare: 55

Biblioteca Trovata.
Premi invio per continuare. . .

Rimuovendo la Biblioteca rimuoverai anche i suoi possessi.
Continuare? (S si, N no) S
Biblioteca e relativi possessi rimossi.
Premi invio per continuare. . .

Scrittura completata.
Lettura dei dati da file completata.
Scrittura completata.
```

Figura 4.6. Scelta "2" del menù per la gestione delle Biblioteche

```
Inserisci il numero della tua scelta: 3
Inserisci il codice della Biblioteca da cercare: 2

Biblioteca Trovata.
Premi invio per continuare. . .

Nome attuale Biblioteca: Biblioteca del collegio d'abruzzo
Inserisci il nome della biblioteca aggiornato: Biblioteca del Collegio d'Abruzzo

Nome biblioteca aggiornato.
Premi invio per continuare. . .

Scrittura completata.
```

Figura 4.7. Scelta "3" del menù per la gestione delle Biblioteche

```
Inserisci il numero della tua scelta: 4
Inserisci il codice della Biblioteca da cercare: 12


Biblioteca Trovata.
Premi invio per continuare. . .
Dati Biblioteca:
Codice Biblioteca: 12
Nome Biblioteca: Biblioteca Pasquale Albino
Città Biblioteca: Campobasso

Premi invio per tornare al menù. . .
```

Figura 4.8. Scelta “4” del menù per la gestione delle Biblioteche

```
Inserisci il numero della tua scelta: 5

Stampa dei dati delle Biblioteche

Biblioteca 1
Codice : 1
  Nome : Biblioteca comunale Luciano Benincasa
 Città : Ancona
########################################
Biblioteca 2
Codice : 2
  Nome : Biblioteca del Collegio d'Abruzzo
 Città : L'Aquila
########################################
Biblioteca 3
Codice : 3
  Nome : Biblioteca della Casa del Sacro Cuore
 Città : Potenza
########################################
 • • •
Biblioteca 22
Codice : 23
  Nome : Biblioteca comunale Rinaldo Veronesi
 Città : Bologna
########################################
Biblioteca 23
Codice : 55
  Nome : Biblioteca del Seminario maggiore dell'Immacolata
 Città : Como
########################################
Stampa Completata.
Premi invio per continuare. . .
```

Figura 4.9. Scelta “5” del menù per la gestione delle Biblioteche

Torniamo al menù iniziale e vediamo la scelta “2”, che viene mostrata in Figura 4.10.

Scelta del menù per la gestione dei Libri

In questo menù, per la gestione dei Libri, abbiamo quattro scelte, sempre escludendo la numero “0”, che consente di tornare al menù principale. Di queste scelte, però, mostreremo soltanto la numero “3”, in Figura 4.11, poiché, le altre sono identiche a quelle già viste nel menù della gestione delle Biblioteche.

```
  ### Programma Gestionale per Biblioteche ###

 Scegli cosa vuoi gestire

  1. Gestione Biblioteche;
  2. Gestione Libri;
  3. Gestione Possessi;
  4. Altre operazioni;
  0. Esci dal programma;

Inserisci il numero della tua scelta: 2

  ### Gestione dei Libri ###

 Scegli cosa vuoi fare

  1. Inserisci un nuovo libro nel sistema;
  2. Rimuovi un libro, usando il suo codice;
  3. Ricerca dei libri successivi ad un relativo anno;
  4. Stampa informazioni di tutti i libri presenti;
  0. Torna al menù precedente;

Inserisci il numero della tua scelta:
```

Figura 4.10. Scelta "2" del menù iniziale

```
Inserisci il numero della tua scelta: 3

Inserisci l'anno di pubblicazione iniziale da cui cercare i libri: 1970

Trovati 2 libri pubblicati dopo il 1970.
Premi invio per continuare. . .


Stampa dei dati dei Libri

Libro 1
     Codice : 25
       Nome : C'era una volta una fabbrica
Anno Pubbl. : 1999
##########################################
Libro 2
     Codice : 26
       Nome : Pinochio
Anno Pubbl. : 2014
##########################################
Stampa Completata.
Premi invio per continuare. . .
```

Figura 4.11. Scelta "3" del menù per la gestione dei Libri

Scelta del menù per la gestione dei Possessi

Riprendiamo, ancora una volta, il menù iniziale e vediamo, ora, la scelta "3", in Figura 4.12.

Anche in questo caso, come nel menù precedente, e per le stesse motivazione di cui sopra, mostriamo soltanto la scelta "3" tra quelle disponibili, visibile in Figura 4.13.

```
  ### Programma Gestionale per Biblioteche ###

 Scegli cosa vuoi gestire

  1. Gestione Biblioteche;
  2. Gestione Libri;
  3. Gestione Possessi;
  4. Altre operazioni;
  0. Esci dal programma;

Inserisci il numero della tua scelta: 3

  ### Gestione dei possessi delle biblioteche ###

 Scegli cosa vuoi fare

  1. Inserisci il possesso di un nuovo libro da parte di una biblioteca;
  2. Rimuovi un possesso, usando i codici della biblioteca e del libro;
  3. Modifica il numero di copie, specificando i codici della biblioteca e del libro;
  4. Stampa informazioni di tutti i possessi nel sistema;
  0. Torna al menù precedente;

Inserisci il numero della tua scelta:
```

Figura 4.12. Scelta "3" del menù iniziale

```
Inserisci il numero della tua scelta: 3
Inserisci il codice della Biblioteca da cercare: 3
Inserisci il codice del Libro da cercare: 58

Possesso Trovato.
Premi invio per continuare. . .

Numero di copie attuale: 46

Inserisci il numero di copie aggiornato (max 50): 40
Numero di copie aggiornato.
Premi invio per continuare. . .
```

Figura 4.13. Scelta "3" del menù per la gestione dei Possessi

Menù delle operazioni extra

Infine, in Figura 4.14, mostriamo l'ultima scelta del menù iniziale.

```
  ### Programma Gestionale per Biblioteche ###

 Scegli cosa vuoi gestire

  1. Gestione Biblioteche;
  2. Gestione Libri;
  3. Gestione Possessi;
  4. Altre operazioni;
  0. Esci dal programma;

Inserisci il numero della tua scelta: 4

  ### Operazioni extra eseguibili ###

 Scegli cosa vuoi fare

  1. Ricerca dei dati dei libri dell'anno scelto dall'utente che si trovano, con un numero di copie maggiore o uguale a quello specificato,
     in esattamente 2 biblioteche della città inserita dall'utente;
  2. Ricerca dei dati dei libri di un anno successivo al 1523, che si trovano esclusivamente nelle biblioteche di Bologna;
  3. Costruzione e Stampa della Matrice dei Possessi;
  4. Visualizzazione, a partire dalla Matrice dei possessi, delle biblioteche che possiedono tutti i libri posteriori al 1500,
     posseduti dalla biblioteca il cui codice è specificato in input dall'utente;
  5. Visualizzazione, a partire dalla Matrice dei possessi, delle biblioteche, se esistono, che possiedono il massimo numero di copie
     di libri pubblicati tra il 1580 e il 1620 e, nel contempo, possiedono il minimo numero di libri tra il 1930 e il 1950;
  6. Calcolo, a partire dalla Matrice, dei volumi che una Biblioteca possiede in più (o meno) rispetto ad un'altra, fornite dall'utente;
  0. Torna al menù precedente

Inserisci il numero della tua scelta:
```

Figura 4.14. Scelta "4" del menù iniziale

Anche in questo ultimo caso abbiamo un altro menù, contenente tutte le funzionalità che non ricadevano negli altri tre visti, con ben sei scelte, che mostreremo nelle Figure 4.15-4.20.

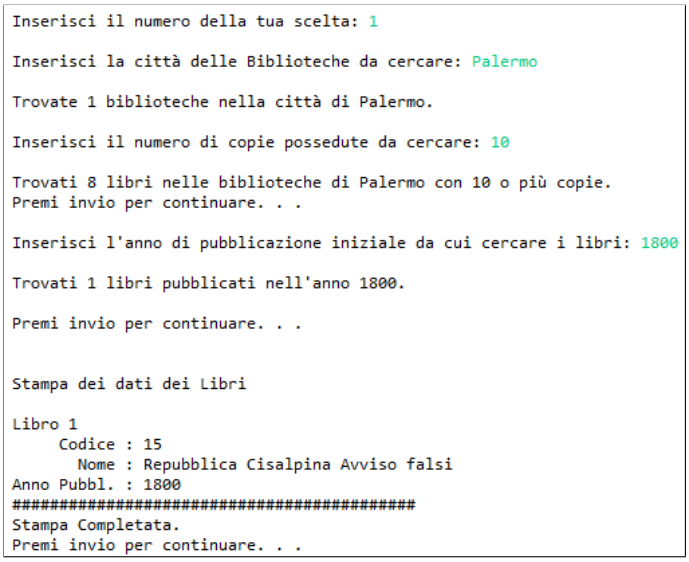


Figura 4.15. Scelta "1" del menù per la gestione delle funzionalità extra

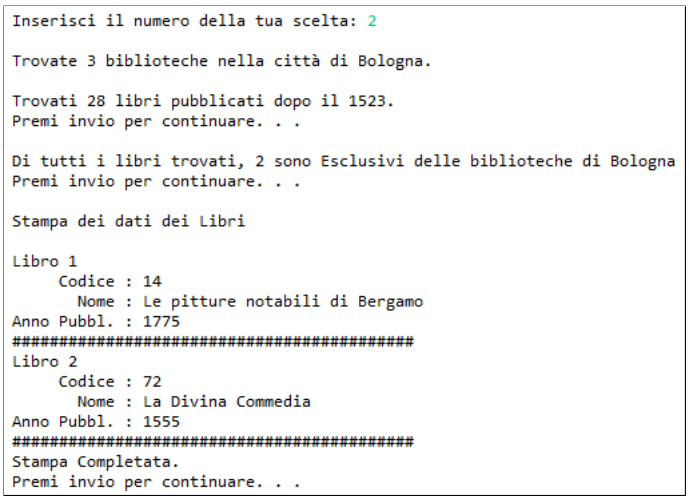
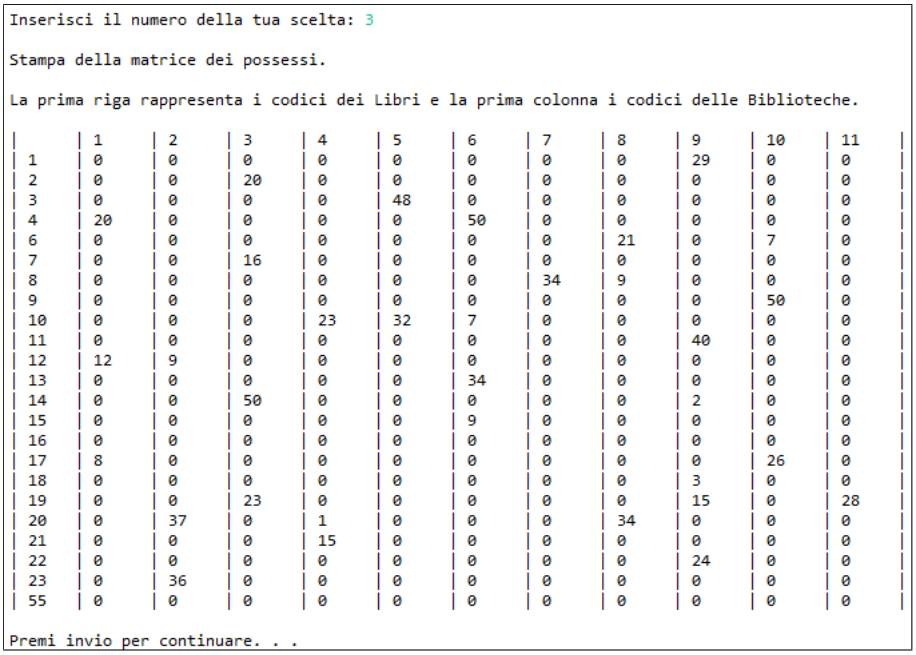


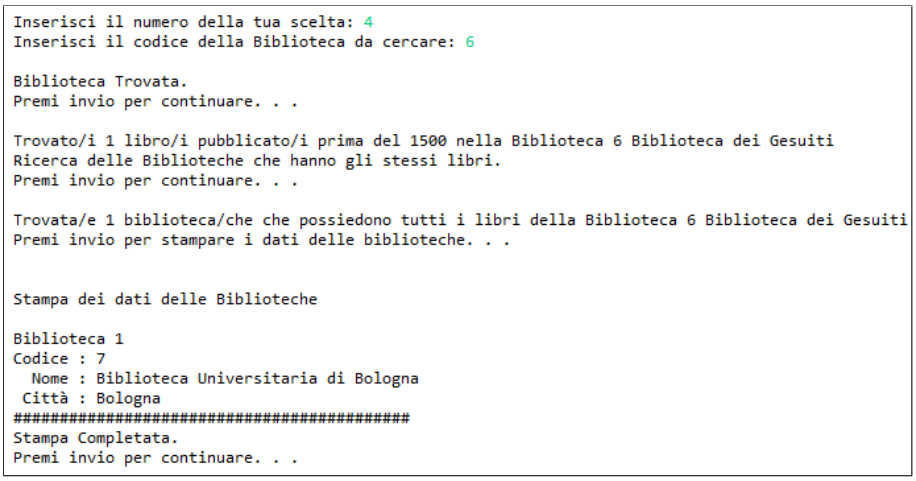
Figura 4.16. Scelta "2" del menù per la gestione delle funzionalità extra

Si vuole, infine, sottolineare che la stampa delle biblioteche è stata volutamente tagliata nell'immagine, per poterla inserire all'interno dell'elaborato, e che,



```
Inserisci il numero della tua scelta: 3

Stampa della matrice dei possessi.

La prima riga rappresenta i codici dei Libri e la prima colonna i codici delle Biblioteche.

|      | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   | 11   |
| 1    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 29   | 0    | 0    |
| 2    | 0    | 0    | 20   | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 3    | 0    | 0    | 0    | 0    | 48   | 0    | 0    | 0    | 0    | 0    | 0    |
| 4    | 20   | 0    | 0    | 0    | 0    | 50   | 0    | 0    | 0    | 0    | 0    |
| 6    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 21   | 0    | 7    | 0    |
| 7    | 0    | 0    | 16   | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 8    | 0    | 0    | 0    | 0    | 0    | 0    | 34   | 9    | 0    | 0    | 0    |
| 9    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 50   | 0    |
| 10   | 0    | 0    | 0    | 23   | 32   | 7    | 0    | 0    | 0    | 0    | 0    |
| 11   | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 40   | 0    | 0    |
| 12   | 12   | 9    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 13   | 0    | 0    | 0    | 0    | 0    | 34   | 0    | 0    | 0    | 0    | 0    |
| 14   | 0    | 0    | 50   | 0    | 0    | 0    | 0    | 0    | 2    | 0    | 0    |
| 15   | 0    | 0    | 0    | 0    | 0    | 9    | 0    | 0    | 0    | 0    | 0    |
| 16   | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 17   | 8    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 26   | 0    |
| 18   | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 3    | 0    | 0    |
| 19   | 0    | 0    | 23   | 0    | 0    | 0    | 0    | 0    | 15   | 0    | 28   |
| 20   | 0    | 37   | 0    | 1    | 0    | 0    | 0    | 34   | 0    | 0    | 0    |
| 21   | 0    | 0    | 0    | 15   | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 22   | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 24   | 0    | 0    |
| 23   | 0    | 36   | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |
| 55   | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |

Premi invio per continuare. . .
```

Figura 4.17. Scelta “3” del menù per la gestione delle funzionalità extra



```
Inserisci il numero della tua scelta: 4
Inserisci il codice della Biblioteca da cercare: 6

Biblioteca Trovata.
Premi invio per continuare. . .

Trovato/i 1 libro/i pubblicato/i prima del 1500 nella Biblioteca 6 Biblioteca dei Gesuiti
Ricerca delle Biblioteche che hanno gli stessi libri.
Premi invio per continuare. . .

Trovata/e 1 biblioteca/che che possiedono tutti i libri della Biblioteca 6 Biblioteca dei Gesuiti
Premi invio per stampare i dati delle biblioteche. . .


Stampa dei dati delle Biblioteche

Biblioteca 1
Codice : 7
  Nome : Biblioteca Universitaria di Bologna
 Città : Bologna
########################################
Stampa Completata.
Premi invio per continuare. . .
```

Figura 4.18. Scelta “4” del menù per la gestione delle funzionalità extra

di conseguenza, nell’esecuzione vengono stampati tutti gli elementi presenti nel sistema.

Per la creazione di queste immagini e delle prove effettuate sul programma stesso, sono state utilizzate un totale di 25 biblioteche e 88 libri, reperiti grazie all’utilizzo dei siti `https://anagrafe.iccu.sbn.it/it/`, per i dati delle biblioteche, e `https://it.wikisource.org/wiki/Categoria:Libri_per_anno_di_pubblicazione`, per i dati dei libri.

```
Inserisci il numero della tua scelta: 5

Trovata/e 1 biblioteca/che che possiedono i requisiti cercati.
Premi invio per stampare i dati delle biblioteche. . .


Stampa dei dati delle Biblioteche

Biblioteca 1
Codice : 4
  Nome : Biblioteca comunale Filippo De Nobili
 Città : Catanzaro
##########################################
Stampa Completata.
Premi invio per continuare. . .
```

Figura 4.19. Scelta “5” del menù per la gestione delle funzionalità extra

```
Inserisci il numero della tua scelta: 6
Inserisci il codice della Biblioteca da cercare: 10
Inserisci il codice della Biblioteca da cercare: 15

Biblioteca 10 ha 170 volumi, mentre Biblioteca 15 ha 272 volumi.

Biblioteca 2 ha 102 volumi in più.
Premi invio per continuare. . .
```

Figura 4.20. Scelta “6” del menù per la gestione delle funzionalità extra

I possessi dei libri, così come i numeri delle copie di questi ultimi, sono stati tutti generati in maniera casuale e non rispecchiano le effettive disponibilità delle biblioteche, inserite nel sistema a fine dimostrativo.

5

Conclusioni e uno sguardo al futuro

Dopo aver visto l'implementazione del codice, in questo capitolo riassumeremo le nozioni che lo sviluppo di questo progetto ci ha portato ad apprendere e conoscere, stilando, anche, un'analisi qualitativa del sistema software creato, sottolineando i suoi punti forza, di debolezza e le aree migliorabili in futuro.

5.1 Nozioni apprese dalla realizzazione

Durante la realizzazione di questo progetto, partendo dalla prima modellazione, fino ad arrivare all'implementazione del codice del programma, c'è stato il bisogno di studiare e apprendere, il più delle volte partendo da zero, molteplici concetti e argomenti, che sono, poi, diventati di fondamentale importanza del progetto stesso. Tra questi, sicuramente, il primo è stato il linguaggio di programmazione Java, seguito, subito dopo, dall'ambiente di sviluppo utilizzato per codificare il programma.

Oltre a questi due "argomenti chiave", come si è visto nel Capitolo 3, sono stati approfonditi anche i linguaggi di modellazione, in particolare UML, e anche l'analisi del progetto, per poter, poi, stilare i requisiti richiesti dal sistema e codificarli in maniera più semplice, grazie ai modelli creati.

5.1.1 Linguaggio Java

Per la realizzazione di questo progetto, sono risultati necessari lo studio e l'apprendimento del linguaggio Java, non conosciuto prima dell'inizio della progettazione, e descritto nel Capitolo 2.

Nonostante questa mancanza, però, l'apprendimento di Java, si è rivelato molto scorrevole e, soprattutto, non complesso, una volta comprese le regole legate alle classi e alle dichiarazioni di metodi e attributi. Una qualsiasi persona, che non sia completamente estranea al mondo della programmazione, non avrà problemi ad approcciarsi al suo studio, date, anche, le similarità con altri linguaggi, come, ad esempio, C o C++. Le librerie di Java, inoltre, aiutano moltissimo il programmatore, ampliando ancora di più lo spettro dei metodi e delle classi già definite, da poter utilizzare per gestire alcune funzionalità specifiche, senza necessariamente codificare una classe completamente da zero.

In questo progetto si è avuto modo di approfondire, in primis, la struttura di un programma in Java, con la definizione di molteplici classi e metodi per lo svolgimento delle funzionalità e, successivamente, l'utilizzo di altre classi, derivate da diverse librerie, per svolgere diversi task, tra i quali, anche, la gestione delle stesse liste dei dati.

5.1.2 Ambiente di sviluppo Eclipse

Per poter utilizzare il linguaggio appreso, si è deciso di appoggiarsi ad un ambiente di sviluppo che offrisse, oltre alle funzioni di interprete e compilatore, anche un aiuto per la scrittura e la comprensione del codice.

L'IDE Eclipse, offre all'utente un forte alleato nella programmazione, grazie a tutti i "tool", o strumenti, che mette a disposizione per facilitare la scrittura del codice. Si è, infatti, fatto ampio uso, nel progetto, della funzione per la generazione automatica della struttura standard delle classi, della funzione di controllo degli errori presenti nel codice, in fase di scrittura e non di compilazione, e, inoltre, della funzione di auto completamento delle chiamate ai metodi, utili sia per i metodi standard, presenti nelle librerie di Java, sia per i metodi delle classi definite dall'utente stesso, ricordando eventuali parametri da inserire e il loro ordine.

5.2 Analisi del sistema realizzato

In questa seconda parte del capitolo, faremo una breve analisi dei punti di forza e di debolezza del sistema. Parleremo, inoltre, dei margini di miglioramento, in termini di efficienza del codice e di ulteriori funzionalità.

5.2.1 Punti di Forza

Iniziamo questa analisi partendo dai punti di forza, che il nostro sistema offre. Tali punti possono essere così specificati:

- La presenza dei menù per la selezione delle funzionalità semplifica l'utilizzo del sistema, suddividendo queste ultime in base al tipo di elemento che si vuole gestire, e cercando di rendere la scelta, e la conseguente esecuzione, quanto più immediata possibile.
- L'utilizzo delle liste, invece, permette di non essere legati ad avere un numero finito di elementi presenti nel sistema, in quanto, non essendo dimensionate, le liste permettono di gestire anche cospicue quantità di dati, senza dover modificare il codice.
- L'utilizzo di semplici file di testo, opportunamente formattati, permette, inoltre, di gestire sistemi diversi, ad esempio, dividendo i dati in base alla regione di appartenenza e creando, quindi, tre file per ogni regione, rendendo il programma facilmente scalabile e riutilizzabile.
- Gli inserimenti e le rimozioni, soprattutto dei possessi legati ad un libro o una biblioteca nel secondo caso, vengono automaticamente rimossi dal sistema, evitando possibili problemi con i dati stessi, possono avvenire in modo estremamente semplice.

- I controlli implementati riescono a garantire, nella quasi totalità delle situazioni, una gestione corretta degli inserimenti da parte dell'utente, rispettando i tipi dei dati ed evitando errori in fase di esecuzione.

5.2.2 Punti di Debolezza

Proseguiamo la nostra analisi con i punti di debolezza, mostrati dal programma in fase di testing.

- Manca un'interfaccia grafica, con la quale il programma sarebbe stato reso più intuitivo, anche a livello visivo.
- I dati dell'utente vengono salvati unicamente su file, non utilizzando alcun tipo di database o supporto che ne permetta una copia in cloud o su un dispositivo esterno, dove poterli esportare o salvare alternativamente.
- Il programmatore non può agire su alcune funzionalità del programma, in fase di esecuzione. Egli è, pertanto, costretto a lavorare unicamente sul codice per modificare determinati elementi, come, ad esempio, il nome dei file da utilizzare.
- Vengono utilizzate, in questo specifico sistema[1], alcune funzionalità "statiche", ovvero non modificabili dall'utente, attraverso la scelta di biblioteche, libri o anni desiderati.

5.2.3 Margini di miglioramento

Dalle sezioni precedenti, soprattutto da quella legata ai punti di debolezza, emerge che il programma non è, ovviamente, perfetto e che, di conseguenza, presenta molteplici punti migliorabili, primi tra tutti quelli che emergono dai punti di debolezza.

- Grazie ad uno studio più approfondito, infatti, non sarebbe complesso creare un'interfaccia grafica per interfacciare il programma e renderlo più semplificato nell'utilizzo, cosa che, però, non è stata possibile implementare per motivi di tempo.
- Per la sicurezza dei dati, l'idea più semplice e immediata sarebbe l'utilizzo di una base di dati che, in combinazione con le già descritte liste, permetterebbe ancora più libertà nelle dimensioni dei sistemi da utilizzare.
- La scelta di dedicare un menù, specifico per il programmatore o per eventuali tecnici, utile a gestire in fase di esecuzione la modifica di determinate impostazioni, aiuterebbe a evitare di mettere mano al codice più del dovuto (ovviamente, il menù non sarebbe accessibile all'utente finale, in modo da evitare possibili problemi).
- La "staticità" di alcuni metodi, come già accennato, è facilmente aggirabile e migliorabile, agendo su di essi, modificandone le chiamate e permettendo l'ampliamento delle scelte dell'utente.

[1] In realtà, con l'utilizzo dei metodi già presenti nelle classi, questo problema sarebbe facilmente correggibile. N.d.A.

- La classe `Controllo`, vista nel Capitolo 3, può essere migliorabile, definendo, al proprio interno, tutti i controlli che vengono effettuati nel programma, e non soltanto quelli più importanti. In questo modo, essa sarebbe scalabile e riutilizzabile anche in altri progetti.
- A seconda dell'utilizzo, potrebbe essere più comodo per l'utente, modificare gli attuali legami tra i tre elementi principali del programma, ovvero le biblioteche, i libri ed i loro possessi, collegando questi ultimi alla relativa biblioteca, e gestendo, poi, diverse liste dei possessi, ciascuna legata ad una singola biblioteca.

5.2.4 Potenziali implementazioni future

Come già detto in precedenza, il sistema creato si rivela molto più che incline a possibili miglioramenti futuri.

Basti pensare all'idea del menù per i programmatori, che potrebbe tradursi, successivamente, in un sistema di login, per separare le funzionalità del programma in base al grado di competenza dell'utente, da semplice "Utente comune", a "Superuser", terminando con "tecnico/programmatore".

Oltre a questo, sebbene sia una miglioria minore, sarebbe possibile estendere le funzionalità di modifica, in primis alla classe `Libro`, che ne è sprovvista. La modifica potrebbe essere estesa anche ai nomi dei libri e delle biblioteche, permettendo una modifica completa dei dati presenti.

Parlando di dati, si potrebbero aumentare il numero di dati gestiti, aggiungendo ulteriori informazioni legate alle biblioteche o ai libri, come gli autori di questi ultimi.

Per ciò che concerne le funzionalità, invece, queste sarebbero estensibili, permettendo all'utente di effettuare ulteriori ricerche e di utilizzare i risultati per lo studio statistico dei libri o dei possessi di ogni biblioteca, magari considerando l'importanza di determinati testi o libri.

Un'altra implementazione, inoltre, sarebbe la creazione di un'interfaccia web, tramite un sito o una web application, che permetta di gestire questo sistema in rete, consentendo, anche, la multi-utenza, attraverso l'uso di un database e dei già accennati sistemi di login. Una volta sul web, si possono aprire ancora più opzioni a cui attingere, per rendere l'esperienza d'uso piacevole e stimolante, magari inserendo dei collegamenti a determinati autori o libri, permettendone la scoperta e lo studio.

Riferimenti bibliografici

1. Anagrafe delle Biblioteche Italiane. https://anagrafe.iccu.sbn.it/it/, 2020.
2. Analisi dei requisiti. https://it.wikipedia.org/wiki/Analisi_dei_requisiti, 2020.
3. Analisi dei Requisiti, Porfirio Tramontana ≪ Ingegneria del Software ≪ Ingegneria ≪ Federica e-Learning. http://www.federica.unina.it/ingegneria/ingegneria-del-software-ingegneria/analisi-requisiti/, 2020.
4. ArrayList (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html, 2020.
5. Bytecode. https://it.wikipedia.org/wiki/Bytecode, 2020.
6. Categoria:Libri per anno di pubblicazione - WikiSource. https://it.wikisource.org/wiki/Categoria:Libri_per_anno_di_pubblicazione, 2020.
7. Class Diagram. https://it.wikipedia.org/wiki/Class_diagram, 2020.
8. Communication Diagram. https://it.wikipedia.org/wiki/Communication_Diagram, 2020.
9. Component Diagram. https://it.wikipedia.org/wiki/Component_Diagram, 2020.
10. Deployment Diagram. https://it.wikipedia.org/wiki/Deployment_Diagram, 2020.
11. Diagramma di attività. https://it.wikipedia.org/wiki/Diagramma_di_attivit\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{a\global\mathchardef\accent@spacefactor\spacefactor}\accent18a\egroup\spacefactor\accent@spacefactor, 2020.
12. Eclipse (informatica). https://it.wikipedia.org/wiki/Eclipse_(informatica), 2020.
13. Ereditarietà (informatica). https://it.wikipedia.org/wiki/Ereditariet\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{a\global\mathchardef\accent@spacefactor\spacefactor}\accent18a\egroup\spacefactor\accent@spacefactor_(informatica), 2020.
14. File (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/io/File.html, 2020.
15. FileReader (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html, 2020.
16. FileWriter (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html, 2020.
17. Guida programmazione orientata agli oggetti — HTML.it. https://www.html.it/guide/guida-programmazione-orientata-agli-oggetti/, 2020.
18. Incapsulamento (informatica). https://it.wikipedia.org/wiki/Incapsulamento_(informatica), 2020.

19. Introduzione ad UML: Gli Use Case Diagrams. http://www.federica.unina.it/smfn/ingegneria-del-software-smfn/uml-use-case-diagrams/, 2020.
20. Java ArrayList. https://www.w3schools.com/java/java_arraylist.asp, 2020.
21. Java (linguaggio di programmazione). https://it.wikipedia.org/wiki/Java_(linguaggio_di_programmazione), 2020.
22. java.io (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html, 2020.
23. java.util (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html, 2020.
24. LaTeX. https://it.wikipedia.org/wiki/LaTeX, 2020.
25. Macchina virtuale Java. https://it.wikipedia.org/wiki/Macchina_virtuale_Java, 2020.
26. Modello concettuale (informatica). https://it.wikipedia.org/wiki/Modello_concettuale_(informatica), 2020.
27. Modello (logica matematica). https://it.wikipedia.org/wiki/Modello_(logica_matematica), 2020.
28. Object Diagram. https://it.wikipedia.org/wiki/Object_diagram, 2020.
29. Piattaforma Java. https://it.wikipedia.org/wiki/Piattaforma_Java, 2020.
30. Polimorfismo (informatica). https://it.wikipedia.org/wiki/Polimorfismo_(informatica), 2020.
31. PrintWriter (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html, 2020.
32. Programmare in Java, Guida — HTML.it. https://www.html.it/guide/guida-java/, 2020.
33. Programmazione orientata agli oggetti. https://it.wikipedia.org/wiki/Programmazione_orientata_agli_oggetti, 2020.
34. Scanner (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html, 2020.
35. Sequence Diagram. https://it.wikipedia.org/wiki/Sequence_Diagram, 2020.
36. Statechart Diagram. https://it.wikipedia.org/wiki/Statechart_Diagram, 2020.
37. String (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/lang/String.html, 2020.
38. Template. https://it.wikipedia.org/wiki/Template, 2020.
39. UML: Guida pratica — HTML.it. https://www.html.it/guide/guida-uml/, 2020.
40. Unified Modeling Language. https://it.wikipedia.org/wiki/Unified_Modeling_Language, 2020.
41. Use Case Diagram. https://it.wikipedia.org/wiki/Use_Case_Diagram, 2020.
42. Dr. Annalisa Appice. Introduzione ad Eclipse. http://www.di.uniba.it/~appice/courses/1415/map/teoria/eclipse.pdf, 2015.
43. Alessandro Cannarsi and Marc Baudoin. Impara LaTeX! (...e mettilo da parte). https://users.dimi.uniud.it/~gianluca.gorni/TeX/itTeXdoc/impara_latex.pdf, 2020.
44. E. Frontoni and A. Mancini. *Programmazione ad oggetti per l'Ingegneria Informatica*. McGraw-Hill Education, 2019.
45. Walter Savitch. *Programmazione di base e avanzata con Java*. Pearson, 2018.

Ringraziamenti

Prima di concludere questa tesi, ci tenevo a fare dei ringraziamenti, per quello che è stato il mio viaggio fino questo punto.

Volevo, quindi, ringraziare sicuramente i miei genitori, che mi hanno dato la possibilità di iniziare, continuare e finire questo percorso universitario, supportandomi al meglio delle loro possibilità.

Un altro ringraziamento va, sicuramente, a quelle persone che ho incontrato durante questo percorso universitario, compagni di corso e di studio, vecchi e nuovi, con cui ho condiviso momenti di svago, ma soprattutto progetti didattici e aiuti nello studio e nella preparazione degli esami.

Ringrazio, anche, i miei amici, fuori dall'università, che mi hanno aiutato ad alleggerire la tensione, tra un esame e l'altro, durante questi anni e mi hanno aiutato a superare alcuni momenti dove le cose non andavano come volevo.

Non posso, poi, non ringraziare la mia ragazza Deasy, che mi è sempre stata vicina durante tutto questo mio percorso, incoraggiandomi e credendo in me, anche nei momenti in cui volevo smettere di studiare, non facendomi perdere di vista l'obiettivo e dandomi una ragione in più per impegnarmi; grazie davvero di cuore.

Ringrazio, infine, ma non per importanza, il mio relatore, Professor Domenico Ursino, per l'aiuto che mi ha dato nella realizzazione di questa tesi e anche per avermi fatto riflettere sul progetto che fosse più giusto realizzare, date le mie conoscenze; ringrazio anche il suo dottorando Luca Virgili, che mi ha assistito durante tutta la fase di progettazione e implementazione, dandomi parecchi consigli e chiarendo molti dei miei dubbi.

Il ringraziamento più grande, però, credo vada a me stesso, per il non aver mollato finora e per essermi dimostrato molto più tenace di quanto io stesso credessi, riuscendo a stringere i denti e sopportare situazioni, per me spiacevoli, superandole e continuando ad avanzare, nonostante gli sbagli e gli "Stop" trovati sul percorso.