Indice

1 Analisi
1.1 Requisiti _____2
1.2 Analisi e modello del dominio _____3

2 Design
2.1 Architettura _____
2.2 Design dettagliato _____

3 Sviluppo
3.1 Testing automatizzato _____
3.2 Metodologia di lavoro _____
3.3 Note di sviluppo._____

4 Commenti finali
4.1 Autovalutazione e lavori futuri _____
4.2 Difficoltà incontrate e commenti per i docenti ____

A Guida utente _____

Capitolo 1

Analisi

1.1 Requisiti

Il software,realizzato per un soddisfare un' esigenza di uno dei componenti del gruppo, mira a realizzare un software adibito alla gestione di uno o più magazzini di una casa editrice. Si presuppone che l'utente sia l'amministratore di uno o più magazzini, che contengono i libri di una casa editrice. Ciò significa che tutti i libri sono della stessa casa editrice e che, se ci fossero due o più magazzini, questi non debbano agire l'uno indipendentemente dall'altro.

Requisiti concordati

- Gestione delle attività di uno o più magazzini, le quali prevedono
  - Inizializzazione di un magazzino. Questo prevederà a sua volta funzioni di inserimento libri, o pacchetti di libri.
  - Funzioni di trasferimento libri(in entrata o in uscita), verso più entità trasferenti(possono essere altri magazzini, persone, librerie, azienda che stampa libri).
  - Modifica dei suddetti ordini in entrata o in uscita(p.e. una libreria ha sbagliato la quantità di un certo tipo di libro per un ordine).
  - Ricerca nel database dei magazzini posseduti, di un libro. Saranno fornite informazioni circa la quantità, e circa i campi opzionali posseduti(p.e. collana). Sarà prevista la funzione di autocompletamento.
  - Ricerca nel database dei magazzini posseduti, di tutti i libri che possiedono una o più certe caratteristiche(ricerca filtrata).
- Creazione di file che rappresentino il resoconto relativo al saldo, e ralativo al contenuto, di un dato periodo di tempo(mensile,annuale) di uno o più magazzini.
- Rappresentazione statistica dei dati utili presenti in database(p.e. percentuale libri con prezzo maggiore € 10).

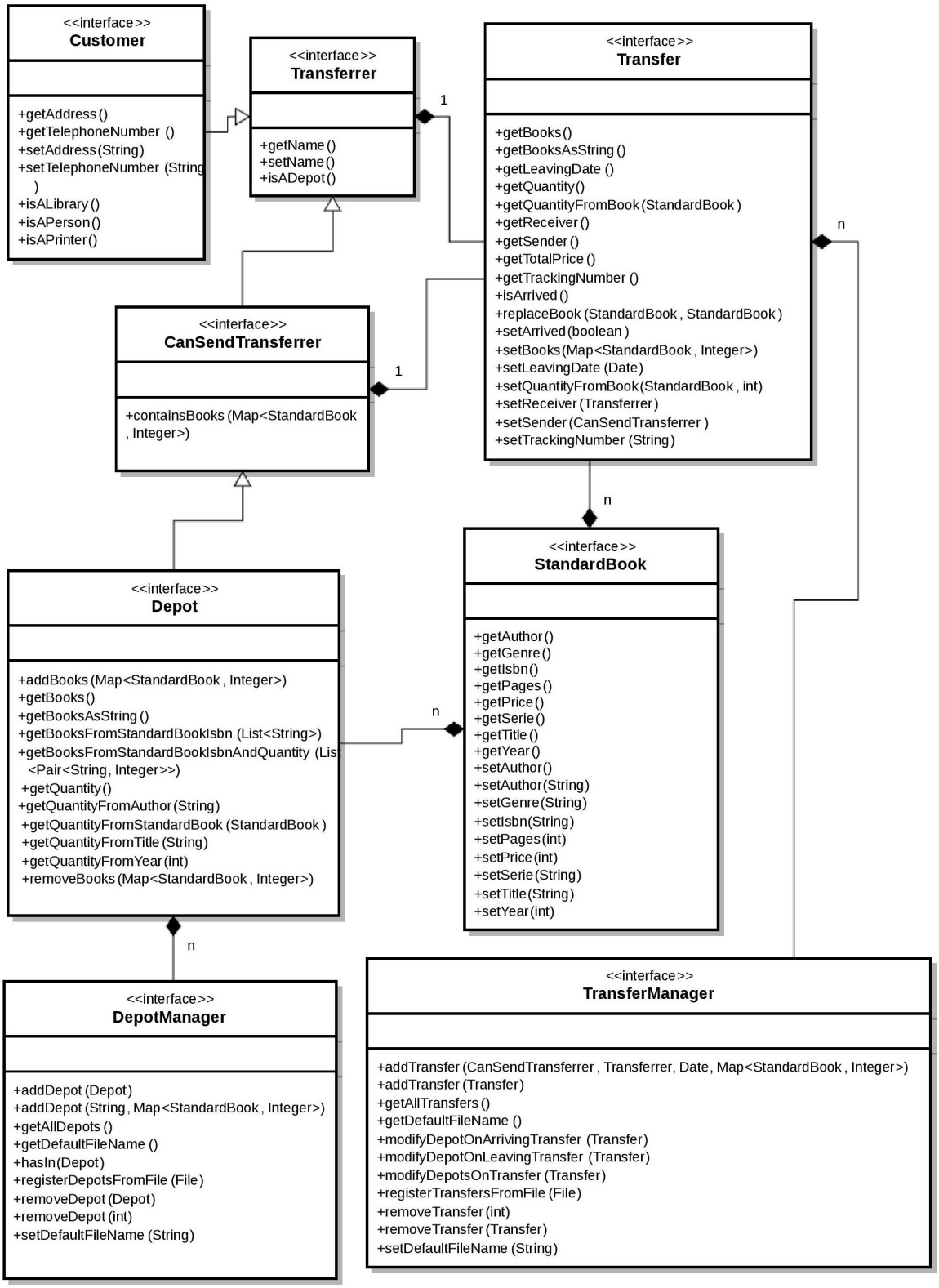
Requisiti opzionali

- Lettura di un file da utilizzare come input per registrazione multipla

di movimenti .
- Creazione di grafici statistici a partire dai dati raccolti .
- Visualizzazione posizione corriere espresso per i libri spediti attraverso API corriere/RSS .
- Invio di mail relative ai resoconti direttamente dal software verso un ipotetico commercialista .

1.2 Analisi e modello del dominio

MedusaBookDepot dovrà essere in grado di accedere e controllare uno o più magazzini(probabilmente nella versione demo ce ne saranno due di default), nelle loro operazioni interne e verso l'esterno. Dovrà essere infatti capace di reperire informazioni relativi ai magazzini(come libri contenuti o nome del magazzino) ed effettuare trasferimenti verso altre entità capaci di trasferire libri(stampa, private persone, librerie, altri magazzini). Naturalmente alcune di queste saranno capaci solo di ricevere, altre solo di inviare, e altre avranno entrambe le funzioni. MedusaBookDepot dovrà inoltre gestire i trasferimeti, stabilendo quindi la data di partenza, la data di arrivo, oltre ai vari campi(tracking number, libri contenuti ecc.). Dovrà anche in questo caso reperire informazioni relative ai trasferimenti(passati e futuri). Una difficoltà quindi sarà quella di trovare un modo per ottenere le informazioni da un input, e scrivere le informazioni su un output, di modo che riavviata l'applicazioni tutte le informazioni relative ai trasferimenti e ai depositi non vadano perse. Gli elementi costitutivi il dominio sono sintetizzati nella figura sottostante.


<<interface>>
Customer
+getAddress()
+getTelephoneNumber ()
+setAddress(String)
+setTelephoneNumber (String
)
+isALibrary()
+isAPerson()
+isAPrinter()
<<interface>>
Transferrer
+getName()
+setName()
+isADepot()
1
<<interface>>
Transfer
+getBooks()
+getBooksAsString()
+getLeavingDate ()
+getQuantity()
+getQuantityFromBook(StandardBook)
+getReceiver()
+getSender()
+getTotalPrice()
+getTrackingNumber ()
+isArrived()
+replaceBook (StandardBook, StandardBook)
+setArrived(boolean)
+setBooks(Map<StandardBook, Integer>)
+setLeavingDate (Date)
+setQuantityFromBook(StandardBook, int)
+setReceiver(Transferrer)
+setSender(CanSendTransferrer)
+setTrackingNumber (String)
n
<<interface>>
CanSendTransferrer
+containsBooks(Map<StandardBook
, Integer>)
1
n
<<interface>>
StandardBook
+getAuthor()
+getGenre()
+getIsbn()
+getPages()
+getPrice()
+getSerie()
+getTitle()
+getYear()
+setAuthor()
+setAuthor(String)
+setGenre(String)
+setIsbn(String)
+setPages(int)
+setPrice(int)
+setSerie(String)
+setTitle(String)
+setYear(int)
<<interface>>
Depot
+addBooks(Map<StandardBook, Integer>)
+getBooks()
+getBooksAsString()
+getBooksFromStandardBookIsbn (List<String>)
+getBooksFromStandardBookIsbnAndQuantity (Lis
<Pair<String, Integer>>)
+getQuantity()
+getQuantityFromAuthor(String)
+getQuantityFromStandardBook (StandardBook)
+getQuantityFromTitle (String)
+getQuantityFromYear(int)
+removeBooks(Map<StandardBook, Integer>)
n
n
<<interface>>
DepotManager
+addDepot (Depot)
+addDepot (String, Map<StandardBook, Integer>)
+getAllDepots ()
+getDefaultFileName ()
+hasIn(Depot)
+registerDepotsFromFile (File)
+removeDepot (Depot)
+removeDepot (int)
+setDefaultFileName (String)
<<interface>>
TransferManager
+addTransfer (CanSendTransferrer, Transferrer, Date, Map<StandardBook, Integer>)
+addTransfer (Transfer)
+getAllTransfers ()
+getDefaultFileName ()
+modifyDepotOnArrivingTransfer (Transfer)
+modifyDepotOnLeavingTransfer (Transfer)
+modifyDepotsOnTransfer (Transfer)
+registerTransfersFromFile (File)
+removeTransfer (int)
+removeTransfer (Transfer)
+setDefaultFileName (String)

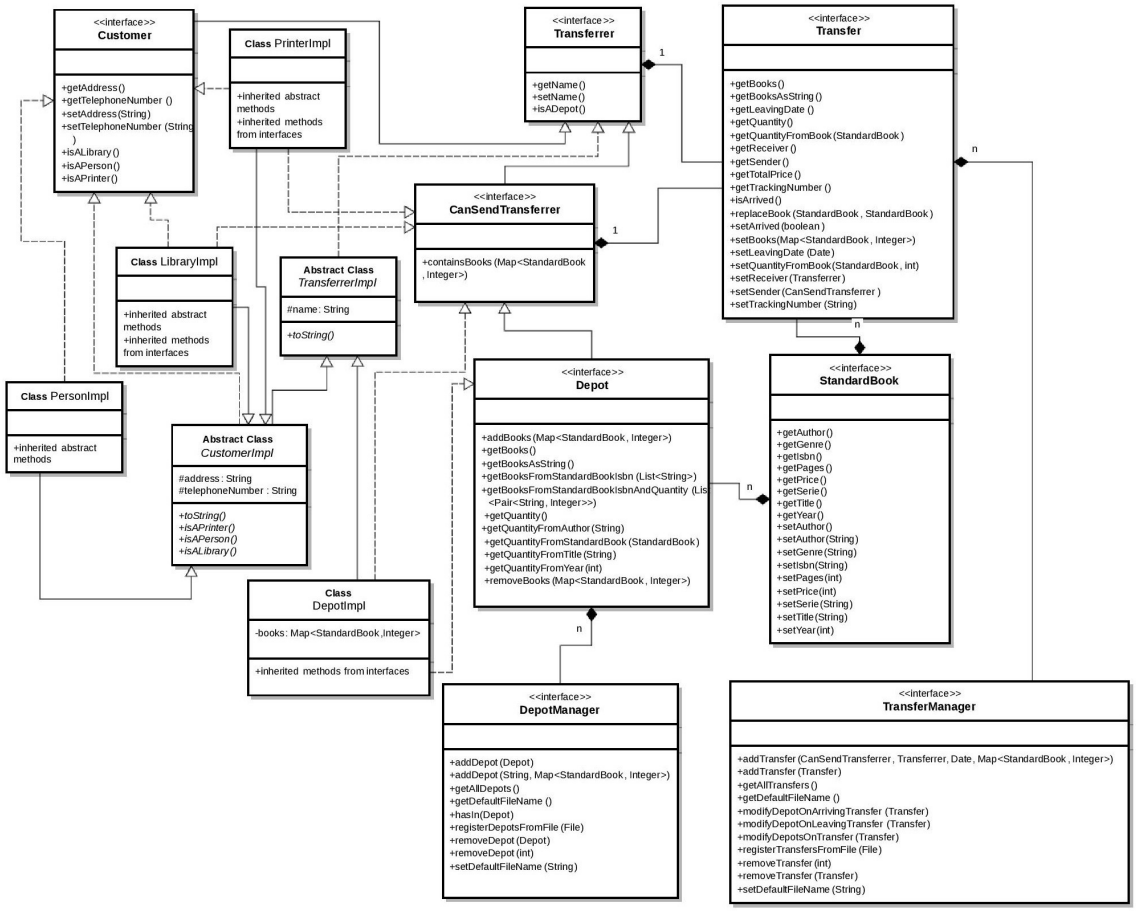
Capitolo 2

Design

2.1 Architettura

2.1 Design dettagliato

2.1.Model Marcello Feroce



Nel model le interfacce fondamentali(con le loro relative classi di implementazione non raffigurate nell'uml qui sopra) sono DepotManager,e TransferManager. La prima interfaccia consente di gestire i depositi, con operazioni base. La seconda invece consente di gestire trasferimenti, e di modificare i depositi coinvolti nel trasferimento. Si noti che quest'ultima interfaccia fa uso dell'interfaccia DepotManager, quando si vuole fare

un'operazione di modifica su un deposito. Le rispettive classi che implementano queste due interfacce, possono avere una sola istanza, è stato usato infatti il pattern creazionale singleton. Non ha infatti senso crearsi più oggetti di questo tipo, con proprietà diverse, ne con proprietà uguali. Questo perché ogni l'oggetto di una di queste classi deve tenere traccia di un insieme particolare di proprietà della sessione del programma(non ha senso creare due oggetti del tipo DepotManager, che lavorano su diversi depositi!).
Quando si lavora con i trasferimenti, necessariamente si dovrà lavorare con entità trasferenti come persone private, magazzini(depot), librerie, Azienda di stampa. Tutti gli oggetti di questo tipo saranno di un tipo quindi che estende una classe astratta TransferrerImpl, la quale a sua volta implenta l'interfaccia Transferrer. Il vantaggio di avere TransferrerImpl astratta è che si può sfruttare il fatto che ogni entità trasferente possiede una proprietà comune, il nome. I metodi relativi al nome infatti sono gestiti in TransferrerImpl. Ho quindi sfruttato qui il pattern Strutturale Template Method. A sua volta possiamo suddividere le entità trasferenti in due gruppi, i magazzini(depot), e i “clienti”(customer, scusate la poca fantasia). Per i depot ho creato una classe a parte che implementa la sua interfaccia Depot, e estende TransferrerImpl,così da poter implementare i modo diverso da altre classi i metodi astratti della classe padre. Ho fatto lo stesso lavoro usando lo stesso pattern per la classe astratta CustomerImpl, che a sua volta estende TransferrerImpl. Ho poi usato il pattern strutturale Strategy, per creare classi che implementino Customer e estendino CustomerImpl; è il caso delle Classi PrinterImpl, LibraryImpl(indica una libreria), e PersonImpl.
Inoltre ho stabilito che ogni oggetto di tipo Transfer deve possedere necessariamente un campo di un tipo particolare, di tipo CanSendTransferrer. Questa interfaccia è implementata solo da classi che identificano entità che possono inviare libri. Questa interfaccia contiene infatti metodi necessari solo da queste classi. Questa moltitudine di classi astratte e interfacce mi consente di fare poche modifiche al codice se voglio cambiare la struttura di ogni oggetto, andando a modificare solo la classe più generale. Nel model ho avuto la necessità di creare due diversi packages, uno per le interfacce ed uno per le classi.