Backend per gestionale di fatture realizzato con Java EE 7

Simone Ricci[1], **Alberto Merciai**[1]

[1]Software Architecture And Methodologies
Laurea Magistrale in Ingegneria Informatica
via di S.Marta, 3 – 50139 – Firenze – Fi – Italia

Abstract. *Questo articolo riporta nel dettaglio le varie fasi adottate per lo sviluppo della logica per un sistema informatico* Java EE 7 *che permette di gestire fatture.*

1. Introduzione

L'obbiettivo del progetto è stato quello di realizzare la logica di un sistema informatico che permetta ad un utente registrato di gestire le fatture che emette verso le proprie aziende compratrici. La realizzazione del prototipo inizia con un'**Analisi dei Requisiti** iniziale, procede con una fase di **Progettazione ed architettura** e si concretizza con una fase di **Test**. Le seguenti sezioni descrivono in dettaglio le varie fasi e riportano le metodologie usate durante lo sviluppo del backend.

2. Analisi dei requisiti

La fase di analisi dei requisiti inizia partendo da una descrizione informale del problema. Da questa sono: estratti i **requisiti** del nostro modello; vengono recuperate quelle che sono le entità e le relazioni coinvolte nel nostro sistema attraverso un **modello concettuale**; costruiti **casi d'uso** con relativi **template**; disegnati dei **mockups**; infine è costruito il **page navigation diagram** che illustra come le diverse pagine interagiscono tra di loro. Le sezioni seguenti descrivono gli step che compongono questa fase nel contesto del problema della gestione di fatture trattato.

2.1. Requisiti

Nella descrizione informale dalla quale sono estratti i requisiti viene indicato come il sistema dovrà interagire con l'utente e sono evidenziate quelle che saranno le funzionalità principali del sistema. Nel contesto della gestione delle fatture il sistema permetterà all'utente di registrarsi ed autenticarsi; in questo modo l'utente potrà gestire i dati relativi alle proprie fatture, ai propri compratori e inoltre al proprio profilo. Da questo documento sono recuperati requisiti di due tipi:

- **data requirements:** requisiti che il sistema deve possedere per soddisfare le necessità legate alla memorizzazione e la manipolazione dei dati. Di seguito è riportato il Data requirements che riguarda l'emittente della fattura:
  - Un Emittente è composto da:
    - dati relativi all'Emittente
      - nome e cognome
      - codice fiscale

- e-mail
- nome
- sede (CAP via...)
- partita iva
- breve descrizione

- **functional requirements:** requisiti legati alle funzionalità che il programma deve offrire per soddisfare le necessità dell'utente, ad esempio:
  - Il sistema deve poter permettere all'Emittente di autenticarsi
    - se già registrato immettere email e password
    - se non registrato riempie i campi legati all'utente da creare

2.2. Modello Concettuale

Il modello concettuale per il sistema gestionale di fatture è costruito sulla base dei requisiti; in particolare le Entità principali previste dal modello concettuale sono 8, in ognuna di queste sono inseriti gli attributi ricavati dai data requirements. La figura 1 sottostante riporta il **modello concettuale** del sistema nel quale sono enfatizzate le relazioni e le entità coinvolte piuttosto che il design del software.

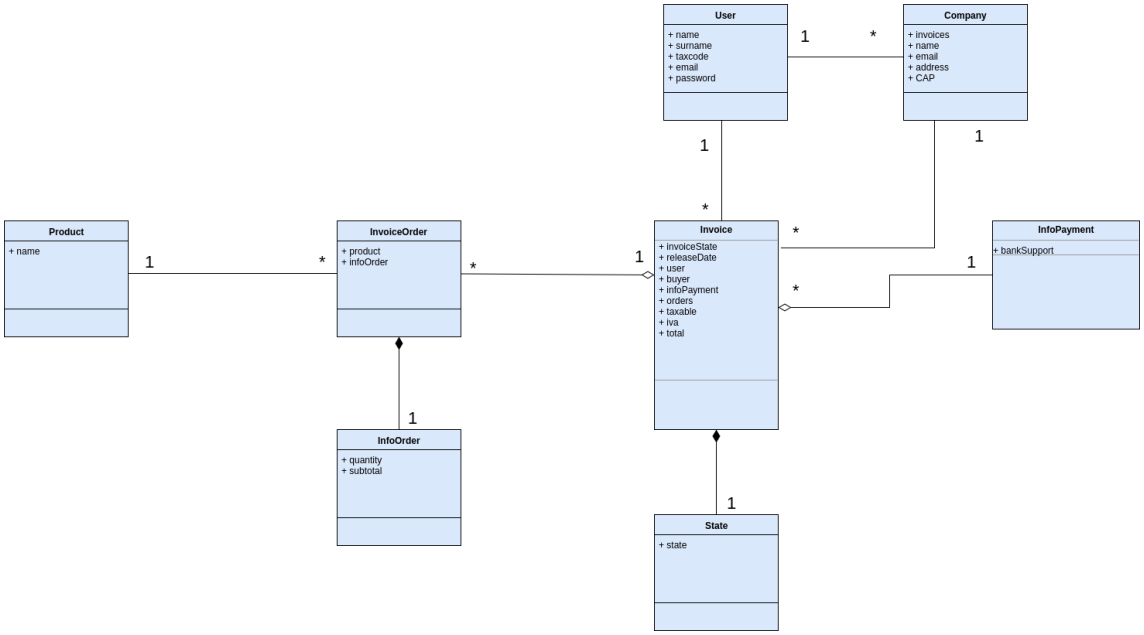

Figura 1. modello concettuale

Da notare in figura 1 la relazione che lega le entità **InvoiceOrder** con **Product**, in particolare la sua cardinalità, che rispecchia la scelta di associare ad ogni ordine di una fattura un solo prodotto. Questo rende indipendenti gli ordini effettuati su prodotti diversi e ed avere una migliore gestione di questi in fase implementativa.

2.3. Casi d'uso

Durante la fase di analisi dei requisiti, dopo aver realizzato il modello concettuale sono disegnati i casi d'uso; questi descrivono come l'utente interagisce con il sistema quando cerca di soddisfare i requisiti funzionali.

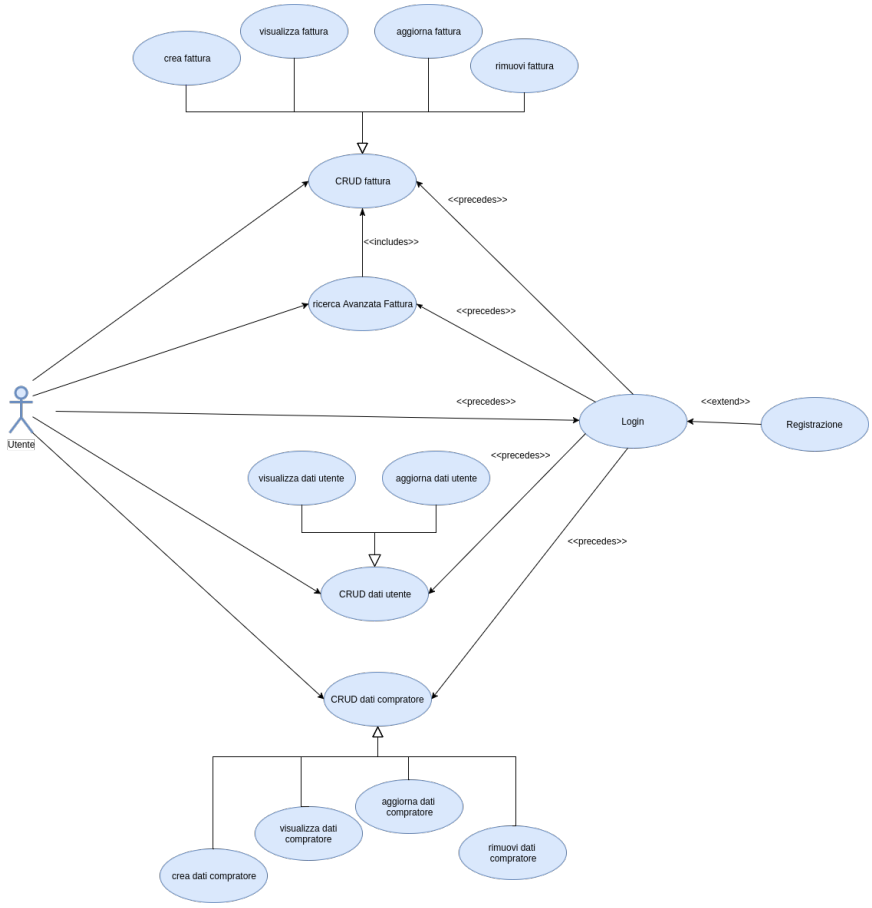

Figura 2. casi d'uso

La figura 2 riporta i casi d'uso disegnati per il sistema in questione. Da porre attenzione sulla sequenza temporale con cui l'utente esercita ogni singolo caso d'uso, infatti prima che questo possa effettuare operazioni sui propri dati, sulle proprie fatture o sulle proprie aziende compratrici deve necessariamente essere registrato al servizio, questo è documentato dallo stereotipo **«precedes»** presente nei corrispondenti casi d'uso. Come documentato in figura 2, in particolare dai casi d'uso CRUD, l'utente una volta registrato al servizio è libero di manipolare qualsiasi dato da lui in possesso. Inoltre il sistema offre un servizio di ricerca avanzata tramite il quale è possibile filtrare le fatture inserite.

2.4. Template

Ogni **use case** è documentato tramite dei **template**, dei moduli testuali che descrivono in dettaglio ogni caso d'uso. Essi includono il livello di astrazione dello **use case**, il flusso principale e flussi alternativi, precondizioni e postcondizioni. Nel contesto del sistema descritto ogni template ha livello di astrazione **user goal** poiché viene descritto il comportamento reale dell'utente nell'esercitare il caso d'uso. Molti **use case** richiedono come condizione iniziale che l'utente abbia effettuato l'accesso al servizio. Un esempio di questo documento è riportato nella figura 3 sottostante.

Use Case: Login

Name: Login
Description: L'utente inserisce le credenziali per accedere al servizio
Primary Actors: Utente
Level: User Goal
Preconditions: L'utente deve essere già registrato nel sistema, altrimenti procede col caso d'uso Registrazione
Main Flow: **1. L'utente seleziona il bottone LOGIN dalla pagina HomePage ed è rediretto alla pagina LoginPage** **2. L'utente inserisce i campi per effettuare il login** **3. l'utente seleziona il bottone LOGIN**
Postconditions:
Alternative Flows: **1. se l'utente non è in possesso delle credenziali occorre effettuare la registrazione premendo dalla pagina LoginPage il bottone REGISTER**

Figura 3. login template

2.5. Page Navigation Diagram

Il penultimo step della fase di analisi dei requisiti è quello che riguarda la costruzione del **page navigation diagram**. In questo documento sono riportate le pagine che compongono l'interfaccia del sistema e le interazioni tra queste. Questo modello è molto utile per capire come un caso d'uso fluisce tra le varie pagine quando viene esercitato dall'utente. Inoltre questo è di fondamentale importanza per l'estrazione di quelli che saranno i **controller** delle nostre pagine nella fase di **progettazione e sviluppo**. La figura 4 riporta nel dettaglio il **page navigation diagram** costruito per il sistema in questione. Come indicato dalla figura 4 l'applicazione prevede 12 pagine, la sottostante lista riassume le principali:

- **HomePage:** Rappresenta la pagina iniziale dell'applicazione tramite la quale è possibile accedere ai servizi di registrazione e login offerti dalla corrispondenti pagine **LoginPage** e **registerPage**. Attraverso questa pagina è inoltre possibile accedere alle pagine di gestione dei dati che riguardano le proprie fatture, i propri compratori e i dati personali.
- **ListInvoicePage, ListBuyerCompaniesPage:** In queste pagine l'utente visualizza fatture o compagnie già precedentemente inserite, quindi può scegliere di inserire nuovi dati o modificare/eliminare quelli già presenti.
- **ViewUserDataPage, ViewInvoicePage, ViewCompanyPage:** A seconda del contesto queste pagine hanno il compito di mostrare all'utente i dettagli riguardo l'informazione selezionata.

- **EditUserDataPage, EditInvoicePage, EditCompanyPage:** A seconda del contesto nel quale l'utente sta lavorando queste pagine permettono al client di modificare o creare nuovi dati.
- **AdvancedSearchPage:** Questa pagina consente all'utente di ricercare fatture sulla base di campi come la data e il nome dell'azienda compratrice

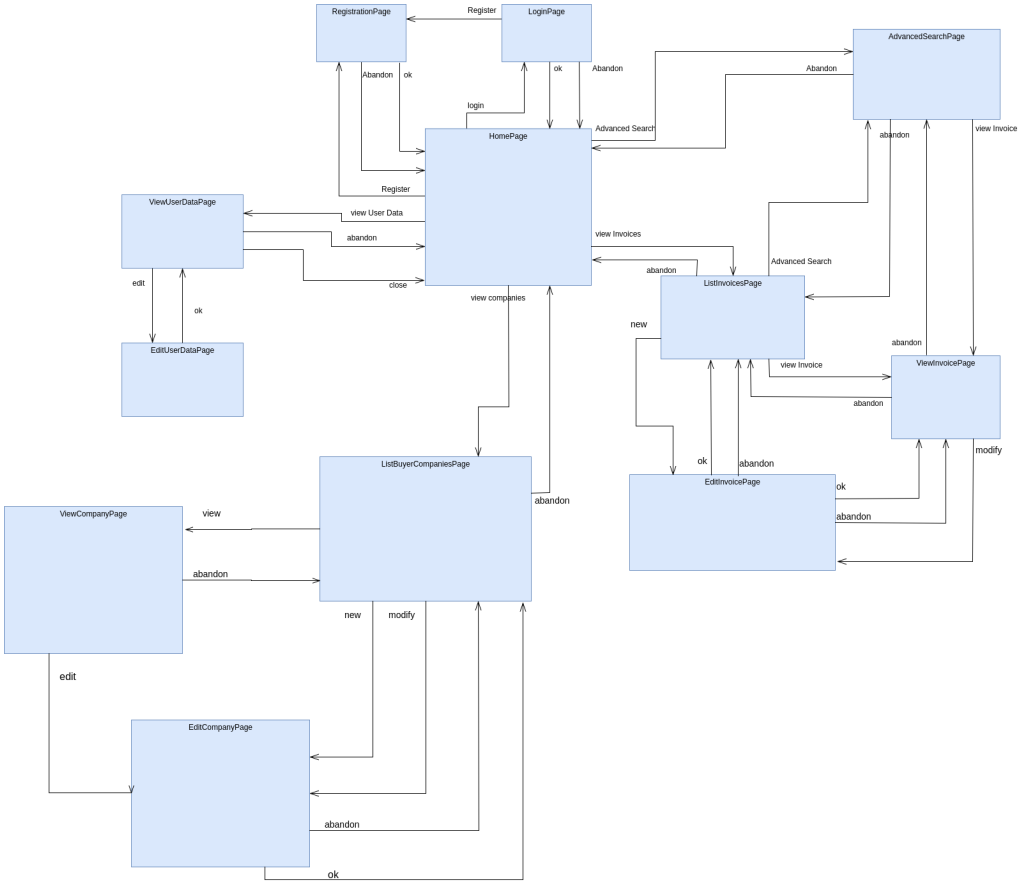

Figura 4. page navigation diagram

2.6. Mockups

La parte conclusiva della fase di analisi dei requisiti è quella relativa ai **mockup**. Sono mostrate le viste di quelle che saranno poi le pagine che compongono il sistema, ponendo maggiore enfasi sui bottoni e le tabelle che le compongono, piuttosto che sul design. La figura 5 riporta alcuni esempi di mockup disegnati per il sistema di gestione delle fatture usati per avere un idea di quello che sarà poi l'interfaccia dell'applicativo. Durante il progetto queste bozze permettono inoltre di semplificare la parte di progettazione dei controller, in quanto consentono di avere un'idea meno astratta di quelle che saranno le varie facciate del sistema, quindi il comportamento di queste.

3. Progettazione e Sviluppo

Nella fase di Progettazione e Sviluppo vengono curati maggiormente quelli che sono i dettagli implementativi del sistema gestionale realizzato tramite Java EE 7 [Oracle].

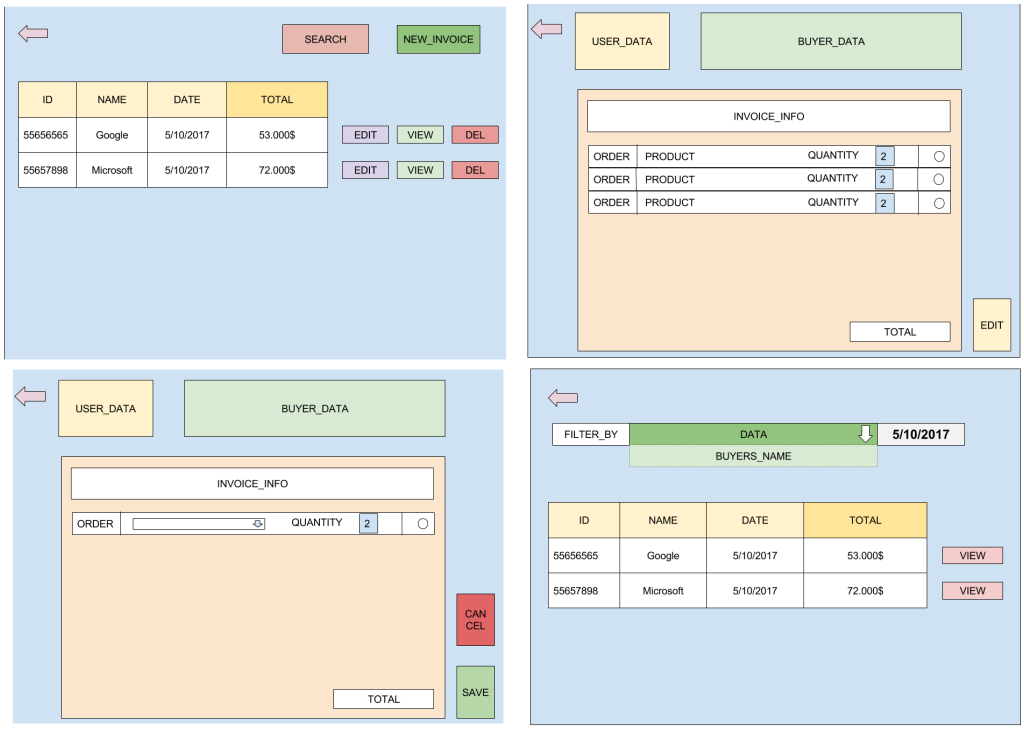

Figura 5. mockup delle pagine riguardanti la gestione delle fatture

In questa parte, infatti, è descritto il modello di dominio, i DAO e l'architettura usata per lo sviluppo del software. Le sezioni sottostanti riportano nel dettaglio i vari step che costituiscono questa fase.

3.1. Architettura

L'architettura adottata per lo sviluppo dell'applicazione è di tipo **3-tier**. Il Presentation Layer non è stato implementato. Il layer di Domain Logic (figura 6) comprende:

- **domain model:** in questa parte dell'architettura sono presenti tutte le entità coinvolte nel sistema. A differenza del modello concettuale in questo modello sono curati i dettagli di design del software sia a livello di pattern sia a livello di mapping sul Database. questa parte è implementata utilizzando la specifica JPA tramite il framework **hibernate**. La domain model è usata dai DAOs che creano e manipolano le entità presenti, quando lo necessitano.
- **DAO:** Il DAO è un oggetto per la gestione della persistenza su database delle entità del modello di dominio. Esiste, quindi, un DAO per ogni entità. Questi creano un maggiore livello di astrazione fra domain model e business logic ed una più facile manutenibilità del codice. I metodi del DAO, con le rispettive query JPQL, sono richiamati dalle classi della business logic.
- **business logic:** ha il compito di reagire agli ingressi implementando le corrispondenti azioni e fornire un entry point per il controllo. Nel contesto dell'applicativo discusso, in questa parte architetturale sono presenti tutti i **controller** delle pagine e quindi tutti i metodi che rispondono agli input sollecitati dall'utente del servizio. La business logic si appoggia ai DAOs per mainpolare ed instanziare le classi appartenenti al **modello di dominio**.

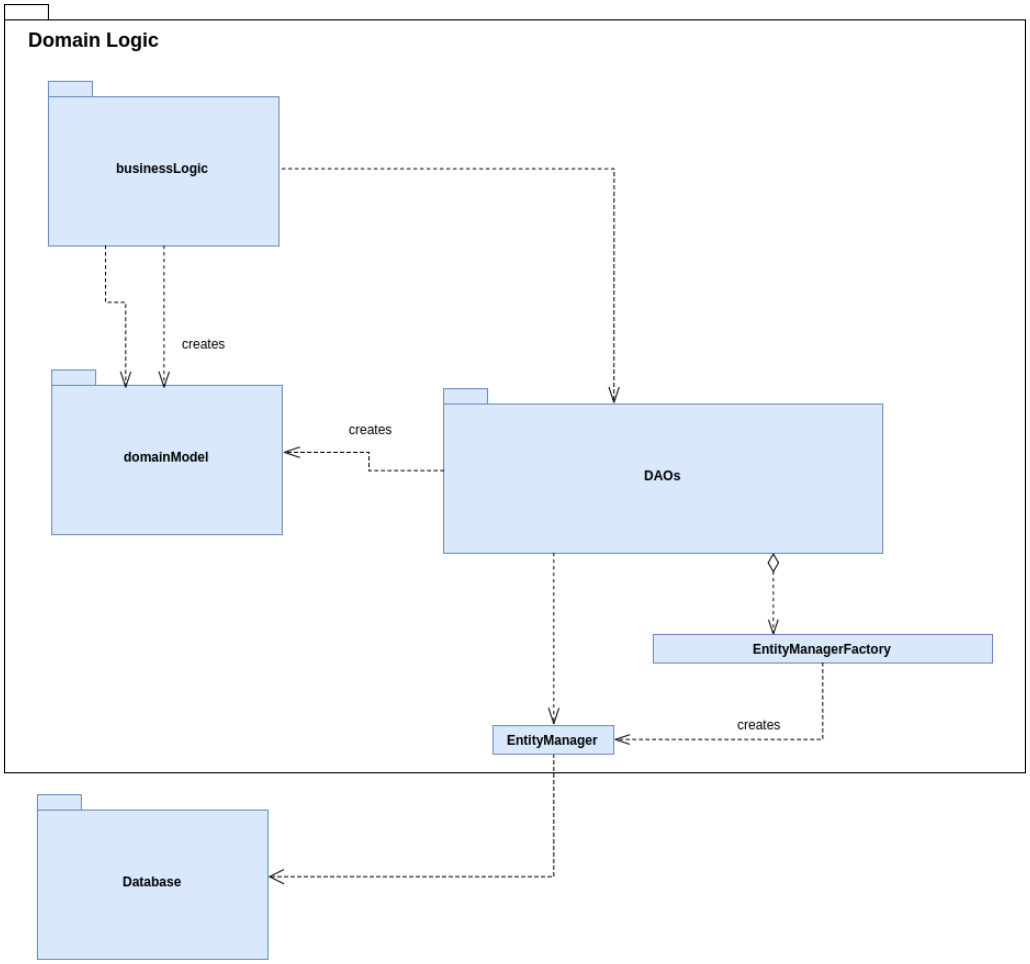

Figura 6. Domain Logic + Database

3.2. Modello di Dominio

Il modello di dominio costruito per l'applicativo è descritto tramite il diagramma UML riportato in figura 7. In questo diagramma sono presenti delle classi value type come **State** e **InfoOrder**, le quali non avendo un propria identità devono ottenerla dall'entità che le contiene. Nel modello è usato il pattern di analisi **Accountability** per ridurre la complessità dovuta alle relazioni binarie tra **User** e **Company**; questo consente di massimizzare l'abilità di evoluzione del sistema introducendo però una maggiore complessità in fase di test. L' **AccountabilityType SellTo** mappa la relazione presente tra l'utente e l'azienda verso la quale vuole emettere una fattura, i **PartyType** consentiti in questa associazione sono User e Company come riportati nel livello di conoscenza del pattern, dove lo User è il responsabile e la Company il commissionario. Per consentire all'utente di filtrare fatture in base ai dati appartenenti al Company è stata introdotta l'associazione 1 a molti tra Company e Invoice. La molteplicità discussa nel modello concettuale tra i Product e gli InfoOrder è mantenuta. Il modello di dominio è stato implementato tramite Java e annotazioni JPA; nello sviluppo è stato fatto uso di una classe astratta chiamata **BaseEntity** che viene estesa da tutte le entità del modello e che permette di evitare di duplicare codice in quanto conserva tutti gli attributi (id, uuid) e metodi (equals)

comuni a tutte le entità. L'ereditarietà presente tra Party e User è mappata sul database tramite una tabella per ogni sottoclasse specificando TABLE_PER_CLASS nell'annotazione **Inheritance**.

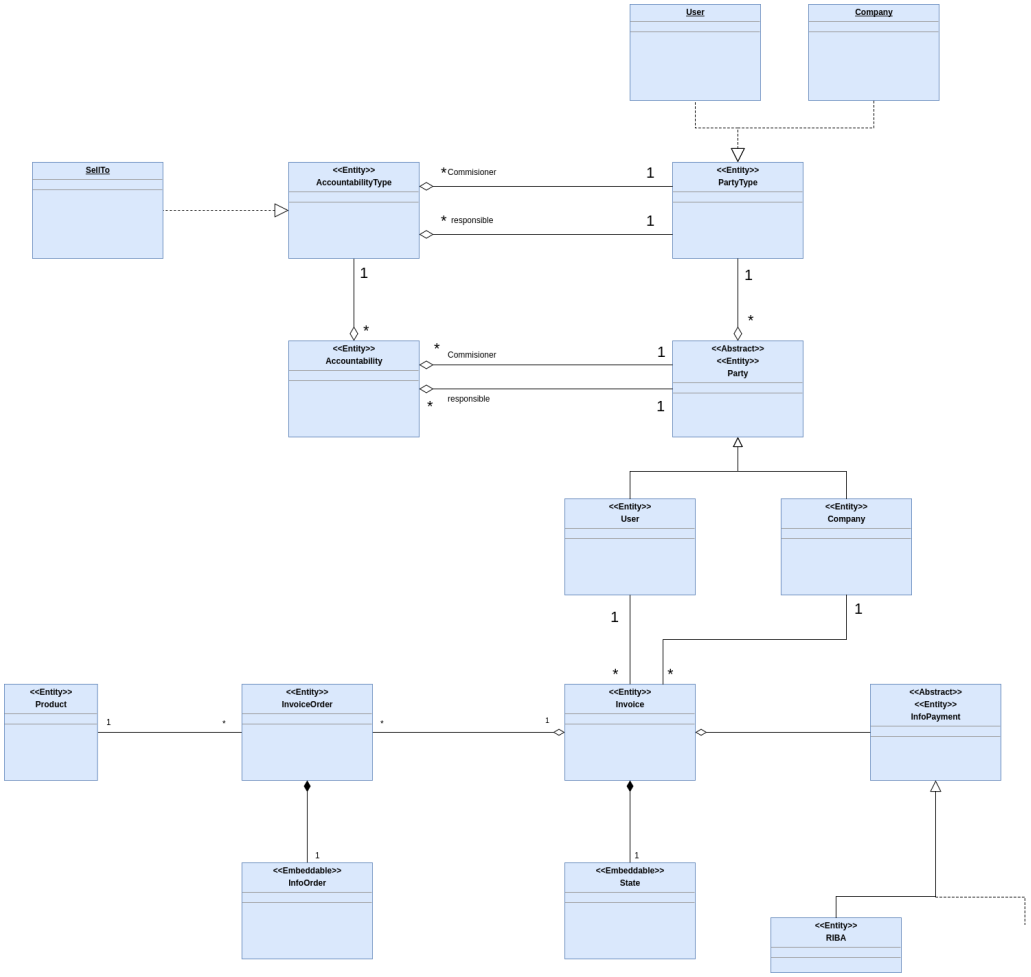

Figura 7. Modello di Dominio

3.3. Dao

I DAO come scritto precedentemente consentono di aumentare il livello di astrazione tra domain model e business logic. Questi permettono di effettuare le operazioni di persistenza sulle entità del domain model mappate sul database servendosi dell'EntityManager. Per la realizzazione dei DAO è stato usato Java, JPQL e annotazioni CDI. Durante lo sviluppo di questi è stata costruita una classe generica chiamata **BaseDao**, successivamente estesa da tutti i DAO, che permette di evitare la duplicazione del codice in quanto conserva i metodi comuni come save, remove e findByID. Per ogni entità è stato implementato un suo particolare DAO.

3.4. business Logic

La business logic racchiude tutti i controller ricavati dai casi d'uso descritti nella sezione 2.3. Ogni pagina è associata ad un controller che risponde ai suoi input. Questa parte dell'architettura usa i DAO per creare le istanze del modello di dominio

e usare queste. Per l'implementazione di questa parte è stato utilizzato Java e le annotazioni CDI. CDI è usato per iniettare le dipendenze nei vari controller; nello specifico è stato usato per iniettare i bean legati alla sessione dell'utente, i DAO ed alcuni parametri passati tramite richieste HTTP dall'interfaccia ai controller. Le annotazioni CDI permettono inoltre di specificare il contesto nel quale un bean rimane in vita; tutti i controller costruiti sono annotati come **Model** ed hanno quindi un tempo di vita e scope di una richiesta; eccetto i controller legati alle pagine di modifica come EditUserDataPageController, EditInvoicePageController, EditCompanyPageController che sono annotati come **ViewScoped** per consentire all'entità recuperata tramite i relativo DAO di restare in vita e quindi riportare le modifiche sul database quando l'utente interagisce con la pagina, la figura 8 mostra la business logic che coinvolge la classe Invoice.

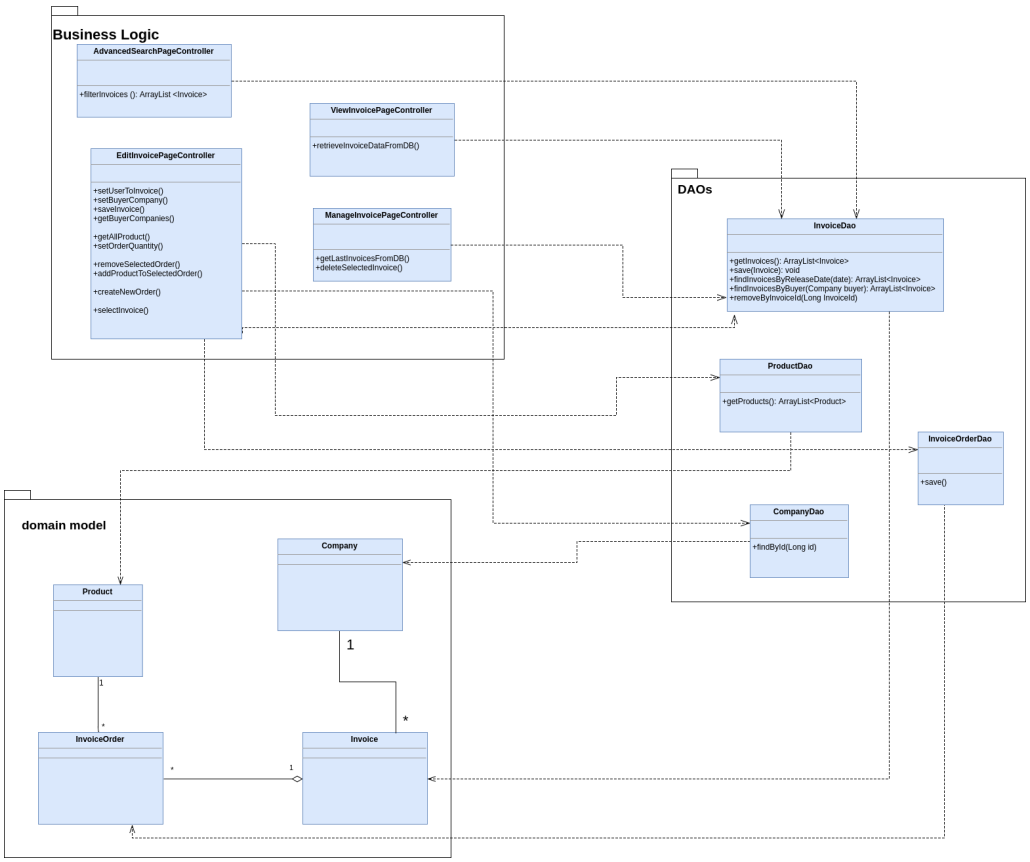

Figura 8. business logic Invoice

4. Test

Anche se inserita come fase finale dello sviluppo dell'applicativo la fase di test è stata eseguita iterativamente durante tutta l'implementazione del software. Infatti dopo che una nuova classe Java è inserita in un package questa viene testata prima di procedere all'implementazione di un'altra. I test applicati sono test di unità. Lo scopo del test di unità è quello di verificare il corretto funzionamento di parti di programma (unità), per esempio una singola classe o un singolo metodo nella programmazione a oggetti. I casi di test adottati sono sia di tipo funzionale che

strutturale, in tutti i test eseguiti sono testati i metodi appartenenti alle varie classi utilizzando il framework **JUnit**[JUnit] e **Mockito**[Faber]. Nella fase finale di testing è stato eseguito il programma **Jacoco** [Jacoco] per valutare la coverage dei nostri test. Come riportato nella figura 9 è stato ottenuta una copertura dell'82.89%; questo numero è giustificato dal fatto che in tutte le classi non sono testati i metodi getter e setter, che però vengono considerati dal software di analisi e questi sono i responsabili della non copertura totale del programma.

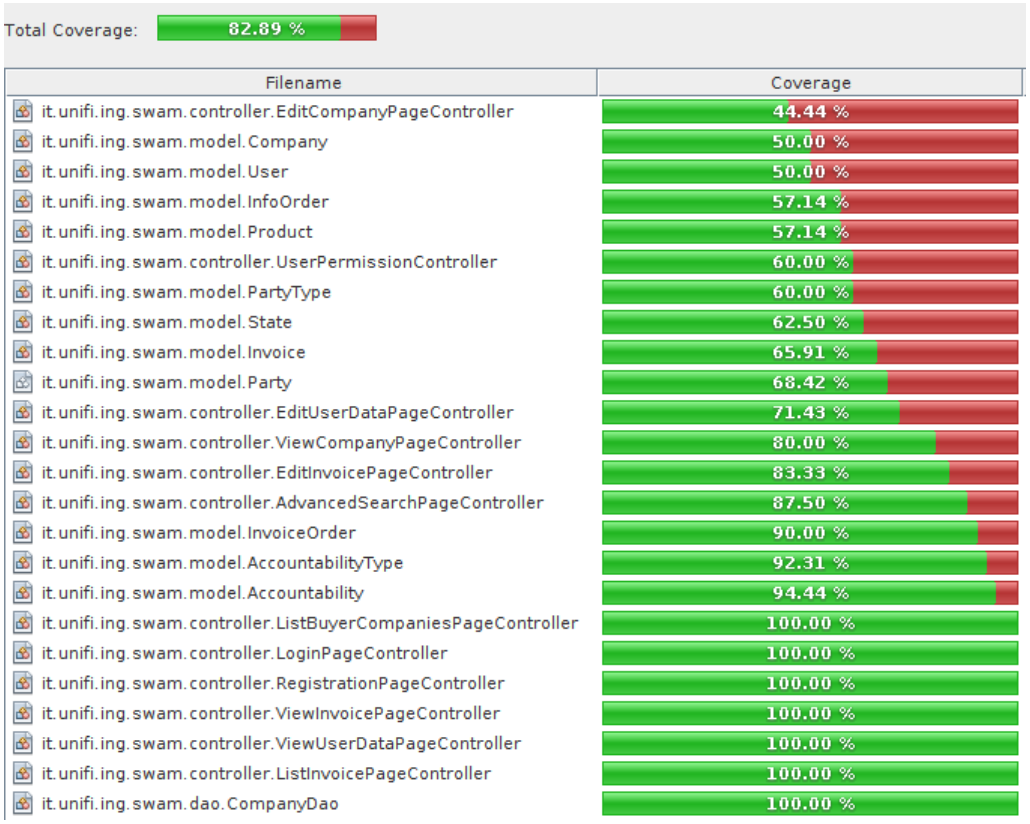

Figura 9. Test coverage

5. Considerazioni finali e sviluppi futuri

Il backend sviluppato seppur privo di interfaccia è progettato e testato in maniera molto curata, durante la fase di progettazione ed implementazione sono utilizzate le tecniche ingegneristiche apprese durante il corso di SOFTWARE ARCHITECTURES AND METHODOLOGIES. Come futuri sviluppi sarebbe interessante aggiungere un layer di presentazione al sistema, successivamente testarlo tramite usability test per verificare come un vero utente interagisce con questo.

Riferimenti bibliografici

Faber, S. Mockito. `http://site.mockito.org/`.

Jacoco. Jacoco. `http://www.jacoco.org/jacoco/`.

JUnit. Junit. `http://junit.org/junit4/`.

Oracle. Java ee 7. `https://docs.oracle.com/javaee/7/tutorial/`.