RELAZIONE PROGETTO DI METODOLOGIE DI PROGRAMMAZIONE

Nome: Serena

Cognome: Vannacci

Matricola: 7030223

Mail: serena.vannacci@edu.unifi.it

DESCRIZIONE FUNZIONALITÀ DEL SISTEMA IMPLEMENTATO

Il progetto modella una libreria digitale, ossia un sistema software ad uso personale per la gestione di libri digitali. Il sistema supporta il calcolo della dimensione in MB dei suoi elementi allo scopo di fornire al client un'operazione che lo aiuti a gestire l'occupazione di memoria.

Il sistema supporta inoltre l'organizzazione di tali libri digitali in semplici sezioni o, per un maggiore ordine, in sezioni annidate, ovvero contenenti sotto-sezioni. Il contenuto di quest'ultime può essere cambiato, aggiungendo o rimuovendo elementi. Nella sezione possono essere cercati tramite nome gli elementi desiderati. I client quindi possono organizzare e consultare la propria libreria digitale, gestendo sia i singoli elementi che sezioni composte da più risorse.

La libreria digitale offre anche una simulazione di esportazione, generando una rappresentazione testuale nei formati JSON e XML degli elementi della libreria.

In aggiunta, è stata implementata una funzionalità di condivisione degli elementi della libreria digitale via email, resa possibile attraverso l'adattamento di un modulo esistente per l'invio tramite mail.

ELENCO PATTERN

- Composite
- Visitor
- Adapter
- Template Method

DESCRIZIONE SCELTE DI DESIGN

Il progetto rappresenta una libreria digitale sviluppata utilizzando diversi design pattern per garantire una buona organizzazione, pulizia, manutenibilità del codice e una maggiore estensibilità. L'architettura del sistema è suddivisa in pacchetti per una gestione più chiara delle classi e delle interfacce.

I pacchetti seguono le convezione standard dei progetti java, i nomi sono tutti in minuscolo, senza spazi o caratteri speciali, e sono organizzati in una struttura coerente.

Le classi e le interfacce sono state organizzate secondo il contesto funzionale riflettendo il dominio applicativo.

Il progetto risulta essere strutturato in più moduli:

- *library.model*: contiene le entità del dominio e le loro relazioni. In altre parole, descrivere il modello concettuale della libreria. Include le classi *Ebook*, *LibrarySection* e la loro classe astratta base come prescritto dal pattern strutturale scelto.

- *library.service*: contiene solo l'interfaccia *LibraryVisitor* e rappresenta il punto di estensione astratto per l'applicazione del pattern Visitor alla struttura della libreria digitale, permettendo le implementazioni di servizi applicativi eterogenei.

- *library.service.exporting*: contiene la logica di esportazione dati in formati specifici (come JSON e XML), implementata tramite il pattern Visitor e Template Method. È ben separato dalla logica core del software, favorendo modularità ed estendibilità.

- *library.service.sharing.adapter*: contiene la logica responsabile dell'adattamento di un servizio preesistente di invio mail all'interfaccia richiesta al sistema.

Per rappresentare la struttura gerarchica degli elementi è stato utilizzato il pattern strutturale *Composite*, perché risolve in modo elegante il problema della rappresentazione e gestione di strutture gerarchiche composte da oggetti che possono essere semplici o composti, come nel sistema proposto, in cui è necessario gestire ebook singoli (*EBook*) e sezioni che contengono altri ebook e sotto-sezioni (*LibrarySection*).

È stata adottata la variante *Type-Safe*, in cui le operazioni tipiche del pattern, ovvero quelle sui figli, sono definite esclusivamente nella classe che rappresenta l'elemento composto, in questo caso *LibrarySection*. Tale approccio, pur garantendo maggiore sicurezza a livello di

tipizzazione, sacrifica l'uniformità dell'interfaccia tra elementi semplici e composti, poiché la logica di composizione non è presente nell'elemento foglia *EBook*.

Il supertipo della gerarchia è rappresentato dalla classe astratta *LibraryComponent*, che definisce le operazioni del dominio comuni a tutti gli elementi della libreria.

Al posto del classico metodo *getChild()* previsto dal pattern, si è scelto una versione del metodo più affine al dominio applicativo, ovvero *findComponentByName()*.

Poiché la gerarchia è relativamente stabile e non si prevede l'aggiunta frequente di nuove classi concrete nel tempo, è stato possibile applicare con efficacia il pattern comportamentale *Visitor*.

Questo risulta particolarmente vantaggioso in contesti in cui la struttura della gerarchia è ben definita e poco soggetta a variazioni, poiché consente di estendere il comportamento del software senza modificare il codice delle classi che rappresentano la struttura.

In questo progetto, il *Visitor* è stato implementato nella sua variante generica allo scopo di separare la logica della struttura da quella di esportazione.

L'interfaccia *LibraryVisitor<T>* rappresenta il supertipo comune per tutti i visitor applicabili agli elementi della libreria. Essa definisce i metodi *vistEBook* e *visitLibrarySection*, parametrizzati con tipo di ritorno T, in modo da supportare visitor con obiettivi diversi. Ogni metodo visit è definito per un tipo concreto della gerarchia e accetta come unico parametro un'istanza dell'elemento stesso.

Affinché il *Visitor* sia applicabile, tutti i tipi concreti della gerarchia devono condividere un super tipo comune, che nel nostro caso è *LibraryComponent*. Tale classe astratta dichiara il metodo *accept()*, il quale viene ridefinito in tutte le sottoclassi concrete.

La classe astratta *ExportVisitorTemplate* implementa l'interfaccia *LibraryVisitor<String>* e definisce il flusso di esportazione per ciascun tipo di elemento attraverso due metodi *final*, come *visitEBook* e *visitLibrarySection*. Questi metodi seguono uno schema fisso: apertura (*formatHeader*), contenuto (*formatBody*) e chiusura (*formatFooter*), lasciando alle sottoclassi concrete la responsabilità di specificare i dettagli del formato.

Questo approccio garantisce coerenza tra i diversi formati di esportazione, migliora la riusabilità del codice e facilità l'estensione.

Il *JsonExportVisitor* e *XmlExportVisitor* sono due visitor concreti che implementano una logica specifica per esportare i dati in formato JSON e XML corrispettivamente. Entrambi produco un risultato di tipo *String*, una scelta che ne facilita il testing.

Il visitor inoltre sacrifica l'incapsulamento, per cui è stato necessario esporre qualche dettaglio della struttura gerarchica, in modo controllato, tramite i getter.

I visitor concreti sono resi final perché non sono pensati per essere estesi.

Il pattern comportamentale *Template Method* è stato applicato per definire una struttura comune nell'esportazione degli elementi della libreria in diversi formati, definendone il flusso logico.

In questo contesto software l'esportazione della struttura della libreria in formati testuali richiede la generazione di output con una struttura coerente: ogni elemento deve essere rappresentato con una apertura, una parte centrale e una chiusura. Questo schema è sempre presente, ma cambia nel contenuto e nella sintassi a seconda del formato scelto.

I template method sono definiti da tre metodi protetti, in ordine, che rappresentano i punti di estensione destinati ad essere ridefiniti nelle sottoclassi concrete per personalizzare i dettagli specifici del formato.

Il pattern risulta efficace in quanto permette di condividere una struttura comune di esportazione, delegando alle sottoclassi solo le variazione specifiche del formato.

Inoltre il metodo *share()*, nella superclasse *LibraryComponent,* incapsula il flusso di costruzione ed invio, stabilendo una scheletro comune per la condivisione. La personalizzazione è demandata attraverso il metodo *buildShareContent()*, lasciato astratto e protetto. Si tratta quindi di un'applicazione del Template Method Pattern, anche se in forma semplice e compatta.

Nell'ottica di rispettare i principi SOLID, è stato creato un sottopacchetto di *exporting*, denominato *format*, che contiene elementi utili alla formattazione. Per ulteriore modularità e chiarezza quest'ultimo è stato suddiviso in due ulteriori sotto pacchetti, *indentation* e *syntax*, contenenti il primo la logica per la gestione dei livelli di indentazione e il secondo le costanti sintattiche (come simboli e nomi di campo) utilizzati per costruire la struttura dei formati di esportazione. Questo approccio consente di evitare ripetizione di stringhe, facilitando la manutenzione e l'estensione.

In particolare, il contenuto del pacchetto *indentation* è rispetta SRP, poiché separa la logica della esportazione da quella di formattazione, OCP perché qualora si voglia cambiare il modo in cui viene fatta l'indentazione basta fornire una nuova implementazione dell'interfaccia *IndentationFormatter*, lasciando invariata la classe *ExportVisitorTemplate*, che rimane chiusa alla modifica ma aperta all'estensione. Inoltre, *ExportVisitorTemplate* dipende da un'astrazione (*IndentationFormatter*) e non da una classe concreta, come previsto da DIP. La scelta di usare

la *dependency injection* tramite costruttore permette infine di modificare la configurazione anche in fase di runtime.

L'interfaccia *IndentationFormatter* rispetta ISP poiché espone la firma di tre metodi coesi e che hanno senso di stare insieme.

Infine si è implementato il pattern Adapter perché risolve in modo efficace il problema dell'integrazione di componenti software che espongono interfacce incompatibili, ovvero interfacce che non coincidono con quelle attese dal sistema. Nella progettazione di un software capita spesso di voler riutilizzare una componente esistente (in questo caso una classe per l'invio di mail) senza modificarne il codice sorgente, ad esempio perché appartiene a una libreria esterna o a codice preesistente, preservando anche il principio di apertura/chiusura OCP.

Nel contesto software proposto era necessario integrare un sistema esistente di invio mail, rappresentato dalla classe *MailSender*, che fornisce funzionalità per la spedizione di messaggi, ma con una interfaccia non compatibile con le esigenze del sistema di condivisione (*SharingService*).

Per colmare questa incompatibilità, si è introdotto la classe *MailSenderAdapter*, che funge da ponte tra l'interfaccia richiesta dal sistema e quella effettivamente fornita dal componente di invio.

Questo pattern favorisce la manutenibilità del sistema: se in futuro si decidesse di sostituire *MailSender* con un altro provider di servizi email, basterebbe fornire una nuova implementazione dell'adapter, senza toccare il codice esistente.

In generale, il progetto tenta di aderire ai principi *SOLID* e del *Clean Code*, prestando particolare attenzione alle leggibilità, alla manutenibilità e alla coerenza della formattazione del codice.

Sono stati usati nome descrittivi, evitando commenti superflui.

Le eccezioni sono gestite in modo esplicito, con messaggi chiari ed informativi.

I metodi privati sono stati inseriti subito dopo quelli pubblici che li usano per una migliore leggibilità e usati per isolare la logica in modo da creare codice più mantenibile.

Le scelte progettuali sono state orientante anche alla testabilità: il codice è stato strutturato per produrre output facilmente osservabili, favorendo la scrittura di test chiari e affidabili. È stato impiegato il *@Before* per il setup comune (in linea con il principio DRY), mentre non si è ritenuto necessario l'uso di @After, poiché gli oggetti non richiedevano operazioni di pulizia post-test.

CLASSI PARTECIPANTI

- Composite
  - package *progetto.mp.vannacci.serena.library.model*
    - classe astratta *LibraryComponent*:

      La classe astratta *LibraryComponent* definisce le operazioni condivise da tutti gli elementi della libreria, sia semplici che composti.

    - Classe *Ebook*:

      La classe *EBook* è l'elemento foglia della struttura gerarchica implementata tramite il pattern Composite, estende *LibraryComponent*. Definisce i dettagli rilevanti in questo specifico contesto per la costruzione di istanze rappresentanti libri digitali.

    - Classe *LibrarySection*:

      La classe *LibrarySection* implementa l'interfaccia *LibraryComponent* e rappresenta un elemento composto, contenente altri *LibraryComponent.* Fornisce i metodi rilevanti per il pattern.

- Visitor
  - package *progetto.mp.vannacci.serena.library.service*
    - Interfaccia *LibraryVisitor<T>*

      L'interfaccia *LibraryVisitor<T>* rappresenta il supertipo comune per tutti i visitor applicabili agli elementi della libreria. Essa definisce i metodi *vistEBook* e *visitLibrarySection*, parametrizzati con tipo di ritorno T, in modo da supportare visitor con obiettivi diversi.

❖ Template Method

– package *progetto.mp.vannacci.serena.library.service.exporting*

▪ Classe astratta *ExportVisitorTemplate*

La classe *ExportVisitorTemplate* serve per definire uno scheletro riutilizzabile per esportare ebook e sezioni. Facilita la creazione di formati diversi senza duplicare la logica strutturale comune.

▪ Classe *JsonExportVisitor*

La classe concreta *JsonExportVisitor* esporta la libreria in formato JSON.

▪ Classe *XmlExportVisitor*

La classe concreta *XmlExportVisitor* esporta la libreria in formato XML.

❖ Adapter

– package *progetto.mp.vannacci.serena.library.sharing.adapter*

▪ Classe *MailSender*

La classe rappresenta un servizio preesistente per l'invio di email. Fornisce metodi per comporre e spedire messaggi, ma espone un'interfaccia non direttamente compatibile con quella richiesta da *SharingService*.

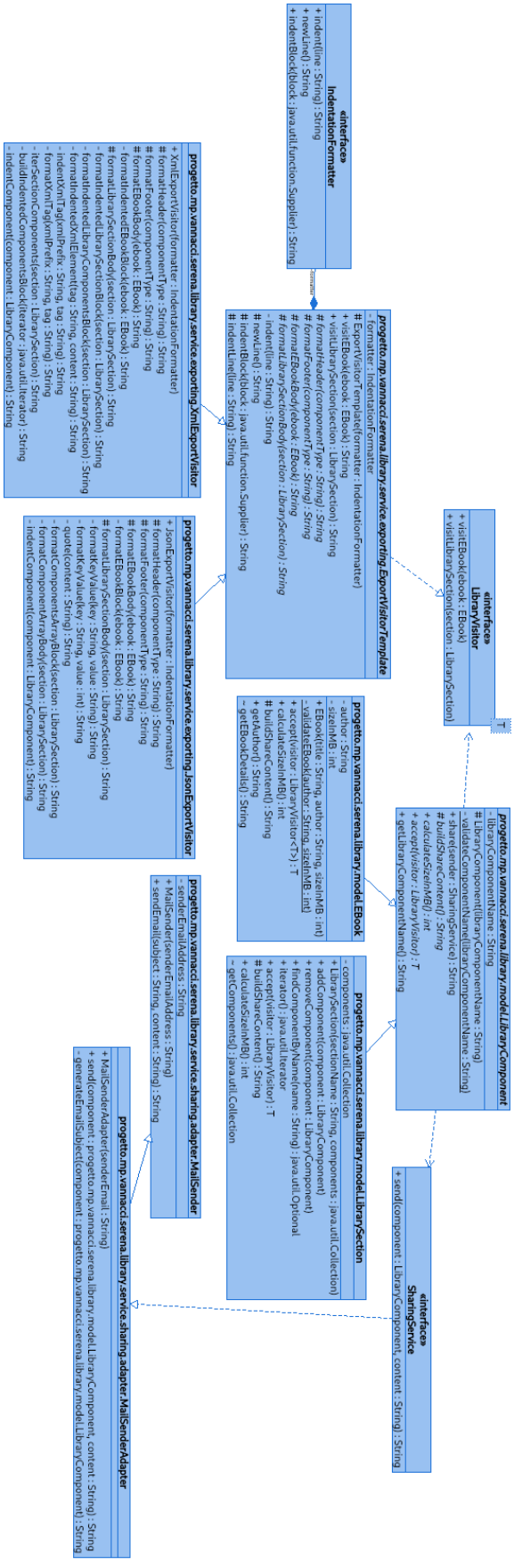
▪ Classe *MailSenderAdapter*

La classe *MailSenderAdapter* implementa l'interfaccia *SharingService* ed estende *MailSender*, fungendo da adattatore verso il servizio di invio mail preesistente.

▪ Interfaccia *SharingService*

L'interfaccia *SharingService* definisce il contratto previsto dal sistema per la condivisione.

GRAFICO UML


«interface»
IndentationFormatter
+ indent(line : String) : String
+ newLine() : String
+ indentBlock(block : java.util.function.Supplier) : String
progetto.mp.vannacci.serena.library.service.exporting.ExportVisitorTemplate
- formatter : IndentationFormatter
ExportVisitorTemplate(formatter : IndentationFormatter)
+ visitEBook(ebook : EBook) : String
+ visitLibrarySection(section : LibrarySection) : String
# formatHeader(componentType : String) : String
# formatFooter(componentType : String) : String
# formatEBookBody(ebook : EBook) : String
# formatLibrarySectionBody(section : LibrarySection) : String
- indent(line : String) : String
newLine() : String
indentBlock(block : java.util.function.Supplier) : String
indentLine(line : String) : String
«interface»
LibraryVisitor
+ visitEBook(ebook : EBook)
+ visitLibrarySection(section : LibrarySection)
T
progetto.mp.vannacci.serena.library.model.LibraryComponent
- libraryComponentName : String
LibraryComponent(libraryComponentName : String)
- validateComponentName(libraryComponentName : String)
+ share(sender : SharingService) : String
# buildShareContent() : String
+ calculateSizeInMB() : int
+ accept(visitor : LibraryVisitor) : T
+ getLibraryComponentName() : String
progetto.mp.vannacci.serena.library.model.EBook
- author : String
- sizeInMB : int
+ EBook(title : String, author : String, sizeInMB : int)
- validateEBook(author : String, sizeInMB : int)
+ accept(visitor : LibraryVisitor<T>) : T
+ calculateSizeInMB() : int
# buildShareContent() : String
+ getAuthor() : String
+ getEBookDetails() : String
progetto.mp.vannacci.serena.library.model.LibrarySection
- components : java.util.Collection
+ LibrarySection(sectionName : String, components : java.util.Collection)
+ addComponent(component : LibraryComponent)
+ removeComponent(component : LibraryComponent)
+ findComponentByName(name : String) : java.util.Optional
+ iterator() : java.util.Iterator
+ accept(visitor : LibraryVisitor) : T
# buildShareContent() : String
+ calculateSizeInMB() : int
- getComponents() : java.util.Collection
«interface»
SharingService
+ send(component : LibraryComponent, content : String) : String
progetto.mp.vannacci.serena.library.service.exporting.XmlExportVisitor
+ XmlExportVisitor(formatter : IndentationFormatter)
# formatHeader(componentType : String) : String
# formatFooter(componentType : String) : String
# formatEBookBody(ebook : EBook) : String
- formatIndentedEBookBlock(ebook : EBook) : String
formatLibrarySectionBody(section : LibrarySection) : String
- formatIndentedLibrarySectionBlock(section : LibrarySection) : String
- formatIndentedLibraryComponentBlock(section : LibrarySection) : String
- formatIndentedXmlElement(tag : String, content : String) : String
- indentXmlTag(xmlPrefix : String, tag : String) : String
- formatXmlTag(xmlPrefix : String, tag : String) : String
- iterSectionComponents(section : LibrarySection) : String
- buildIndentedComponentsBlock(iterator : java.util.Iterator) : String
- indentComponent(component : LibraryComponent) : String
progetto.mp.vannacci.serena.library.service.exporting.JsonExportVisitor
+ JsonExportVisitor(formatter : IndentationFormatter)
# formatHeader(componentType : String) : String
# formatFooter(componentType : String) : String
# formatEBookBody(ebook : EBook) : String
- formatEBookBlock(ebook : EBook) : String
formatLibrarySectionBody(section : LibrarySection) : String
- formatKeyValue(key : String, value : String) : String
- formatKeyValue(key : String, value : int) : String
- quote(content : String) : String
- formatComponentsArrayBlock(section : LibrarySection) : String
- formatComponentArrayBody(section : LibrarySection) : String
- indentComponent(component : LibraryComponent) : String
progetto.mp.vannacci.serena.library.service.sharing.adapter.MailSender
- senderEmailAddress : String
+ MailSender(senderEmailAddress : String)
+ sendEmail(subject : String, content : String) : String
progetto.mp.vannacci.serena.library.service.sharing.adapter.MailSenderAdapter
+ MailSenderAdapter(senderEmail : String)
+ send(component : progetto.mp.vannacci.serena.library.model.LibraryComponent, content : String) : String
- generateEmailSubject(component : progetto.mp.vannacci.serena.library.model.LibraryComponent) : String