Progetto MP

Sistema per la gestione di librerie, libri, autori e gli user

Daniil Radchanka

12.06.2024

Contenuti

Introduzione

Avevo deciso di implementare un sistema di gestione di libri, librerie, autori e user.

Per il design system mi sono appoggiato a seguente metodologie, tranne ovviamente le metodologie viste alle lezioni:

- ***Domain Driven Design*** con la gestione di ***Domain Events***
- Architettura alla cipolla (***Onion***)
- ***Test Driven Development*** (tranne la parte di *Application*, perché considererei che questa parte deve essere coperta con i *Integrational Test*, che non fa parte di argomenti di MP)
- ***Command Query Segregation Principle***
- ***Dependency Injection***

Il Sistema informative e composto da 3 stratti:

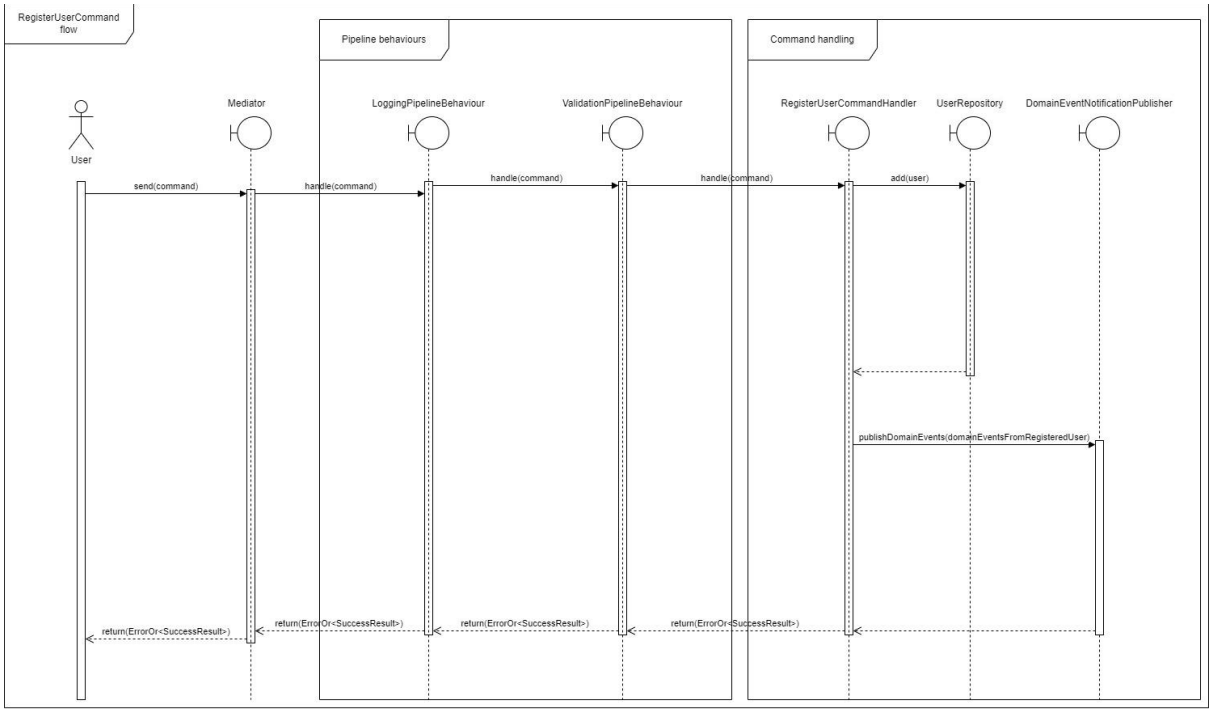
- Domain – dove sono definiti varie aggregazioni, i repository di tali aggregazioni e etc.
- Infrastructure – il stratto dove ci sono le implementazioni di repository definiti dentro Domain
- Application – il stratto principale che definisce i comandi e i query che sono supportati dal sistema. In più contiene anche la gestione di eventi di dominio.

Per rendere il codice riportabile verso altri progetti avevo previsto anche il pacchetto base, dove avevo definito varie implementazioni che sono usati all'interno di tutto l'applicativo come:

- Logger
- Mediator per il supporto di CQRS
- Utils contente la classe con la funzionalità utile e generale
- DDD contente le classi per semplificare lo sviluppo in DDD cioè l'implementazione di Entity e ValueObject.
- Result contente l'implementazione di Result pattern tramite le ***Discriminated Unions***
- Repository contente sia le interfacce di repository sia l'implementazione di un repository in memoria.
- Specification contente la definizione di interfacce e le classi di base usati per definire le specificazioni usati all'interno di repository per fare le ricerche di entità.

- Altri pacchetti meno importanti, pero che sono usati all'interno di progetto e possono essere facilmente riportati in altri progetti.

Prima di procedere avanti e vedere l'implementazione di vari stratti, pattern implementati in progetto sarà abbastanza utile vedere il flow di questo sistema in modo da capire meglio il suo comportamento. Per dimostrarlo faccio un esempio di RegisterUserCommand.



Come si vede dal diagramma l'iteratore tra user e il sistema sono: mediator e il commando/query che vuole eseguire user.

Prima di passare avanti nel processo di gestione del commando vengono applicati i PipelineBehaviour, che definiscono il comportamento da applicare prima oppure anche dopo la gestione del commando.

E quando finalmente si arriva al momento della gestione di commando usiamo le classi di Domain per gestire la logica di dominio associata al commando.

Dopo di aver gestito il processo di dominio pubblichiamo gli eventi ottenuti per l'entità e li passiamo a DomainEventNotificationPublisher, quale sarà responsabile a pubblicare questi eventi. Idealmente a questo punto deve essere usata una coda *Reliable*, pero nel mio caso, per rendere il progetto più semplice da implementare avevo utilizzato di nuovo il mediator e pubblicavo gli eventi a lui.

Implementazione

Domain

Contiene seguenti gruppi di oggetti, raggruppati per le AggregateRoot:

- Entity
- ValueObject
- Specification
- Repository
- Exception
- DomainEvent

Dal DDD l'entità descrive oggetti che hanno il ciclo di vita e devono essere comparati per la chiave. In più applicando avevo utilizzato gli eventi di dominio per poter sincronizzare alcuni azioni tra diverse aggregazioni, pero potrebbe essere usato anche per altri funzionalità che devono essere definiti fuori dal Domain, ad esempio l'invio di una mail ad un User se la libreria dove lui aveva preso il libro aveva cambiato il suo indirizzo.

Per tale scopo avevo definito Entity, che definisce i metodi usati dalle classi ereditate per controllare il suo flow, quale e comparazione per la chiave e registro/esportazione di eventi di dominio.

Per poter rappresentare l'evento di creazione di una entità viene usato il static factory method, il costruttore di entità viene messo a private e si usa il pattern Prototype. Questa combinazione rende possibile la creazione di entità senza evento solo in caso quando viene restituita dal repository e per il resto del mondo siamo forzati ad 'usare questo evento. Facendo così ci garantiamo la presenza di evento che rappresenta la creazione di entità.

Il ValueObject invece di Entity devono essere comparati per valori. Anche come in caso di entità avevo definito la classe di base ValueObject, che costringe le classi ereditate di restituire l'Iterator sui campi che devono essere usati nella comparazione tra gli oggetti, questo metodo rappresenta anche il Template Method pattern, perché viene usato per definire il equals e i campi su quale viene applicato l'equals sono definite all'interno di classe ereditate.

Nel mio progetto avevo usato il ValueObject per rappresentare l'Id di tutte le entità, in modo da renderlo facilmente modificabile, e per rappresentare l'indirizzo.

Specification e un pattern che viene usato assieme al repository e serve per poter definire il criterio su quali entità devono essere restituiti dal repository.

Per semplificare la definizione di specificazioni avevo usato il pattern Composite in modo da poter combinare logicamente vare specificazioni (usando and, or, not).

Repository e un pattern che serve per astrarre il concetto di accesso/modifiche di dati da un posto persistente, quale può essere ad ‘esempio il DB. In mio caso avevo usato un repository in memoria.

Per rendere questo funzionante il repository in memoria mantengo e restituisco solo le coppie di oggetti. Facendo così mantengo il principio di InformationHiding e rendo non modificabili l’oggetti mantenuti all’interno di repository. Per poter copiare vari oggetti avevo utilizzato quasi da per tutto il pattern Prototype.

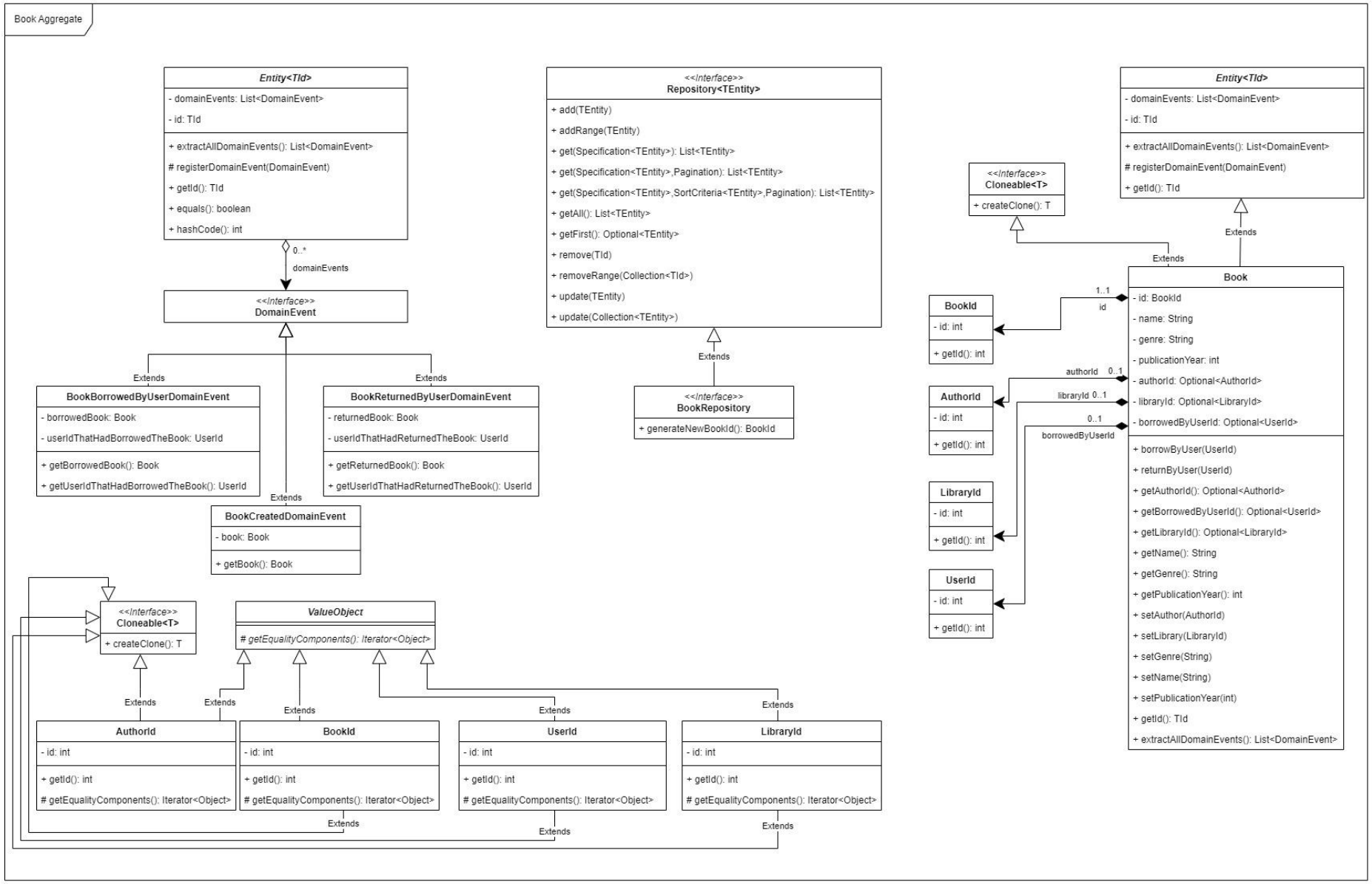
Eccezioni sono definiti solo al livello di Domain perché servono a mostrare i casi che sono non validi per il business, e uno strumento per dire che l’applicativo e arrivato in uno stato non valido in quale lui non deve entrare mai, perché doveva fare i controlli di validità prima di arrivare allo stratto di domain!

DomainEvent viene usato per rappresentare gli eventi di dominio. La gestione di questi eventi ne vedremo avanti nella sezione di Application.

Avevo definito seguenti aggregazioni:

- Book
- User
- Author
- Library

Vediamo l’implementazione di solo un aggregato – book, perché altri hanno l’architettura più semplice ma simile a questa.


Book Aggregate
Entity<TId>
- domainEvents: List<DomainEvent>
- id: TId
+ extractAllDomainEvents(): List<DomainEvent>
# registerDomainEvent(DomainEvent)
+ getId(): TId
+ equals(): boolean
+ hashCode(): int
0..*
domainEvents
<<Interface>>
DomainEvent
Extends
BookBorrowedByUserDomainEvent
- borrowedBook: Book
- userIdThatHadBorrowedTheBook: UserId
+ getBorrowedBook(): Book
+ getUserIdThatHadBorrowedTheBook(): UserId
BookReturnedByUserDomainEvent
- returnedBook: Book
- userIdThatHadReturnedTheBook: UserId
+ getReturnedBook(): Book
+ getUserIdThatHadReturnedTheBook(): UserId
BookCreatedDomainEvent
- book: Book
+ getBook(): Book
<<Interface>>
Repository<TEntity>
+ add(TEntity)
+ addRange(TEntity)
+ get(Specification<TEntity>): List<TEntity>
+ get(Specification<TEntity>,Pagination): List<TEntity>
+ get(Specification<TEntity>,SortCriteria<TEntity>,Pagination): List<TEntity>
+ getAll(): List<TEntity>
+ getFirst(): Optional<TEntity>
+ remove(TId)
+ removeRange(Collection<TId>)
+ update(TEntity)
+ update(Collection<TEntity>)
<<Interface>>
BookRepository
+ generateNewBookId(): BookId
<<Interface>>
Cloneable<T>
+ createClone(): T
ValueObject
getEqualityComponents(): Iterator<Object>
AuthorId
- id: int
+ getId(): int
# getEqualityComponents(): Iterator<Object>
BookId
- id: int
+ getId(): int
# getEqualityComponents(): Iterator<Object>
UserId
- id: int
+ getId(): int
# getEqualityComponents(): Iterator<Object>
LibraryId
- id: int
+ getId(): int
# getEqualityComponents(): Iterator<Object>
Entity<TId>
- domainEvents: List<DomainEvent>
- id: TId
+ extractAllDomainEvents(): List<DomainEvent>
# registerDomainEvent(DomainEvent)
+ getId(): TId
<<Interface>>
Cloneable<T>
+ createClone(): T
Book
- id: BookId
- name: String
- genre: String
- publicationYear: int
- authorId: Optional<AuthorId>
- libraryId: Optional<LibraryId>
- borrowedByUserId: Optional<UserId>
+ borrowByUser(UserId)
+ returnByUser(UserId)
+ getAuthorId(): Optional<AuthorId>
+ getBorrowedByUserId(): Optional<UserId>
+ getLibraryId(): Optional<LibraryId>
+ getName(): String
+ getGenre(): String
+ getPublicationYear(): int
+ setAuthor(AuthorId)
+ setLibrary(LibraryId)
+ setGenre(String)
+ setName(String)
+ setPublicationYear(int)
+ getId(): TId
+ extractAllDomainEvents(): List<DomainEvent>
BookId
- id: int
+ getId(): int
AuthorId
- id: int
+ getId(): int
LibraryId
- id: int
+ getId(): int
UserId
- id: int
+ getId(): int
Extends
1..1
id
authorId 0..1
libraryId 0..1
0..1
borrowedByUserId

Infrastructure

In questo stratto vengono messi l'implementazione di repository definiti dentro Domain oppure altri servizi che interagiscono con le robe esterne al nostro applicativo.

Quindi qui avevo messo soltanto l'implementazione di repository in memoria, definendo la classe base InMemoryRepository in modo da non dupplicare il codice.

Importante pero notare come erano implementati Specification, Pagination e SortCriteria.

Per poter gestire la possibilità di definire il sortBy/thenSortBy per un campo specifico e rendere l'interfaccia di repository indipendente da Comparator avevo sviluppato la mia classe SortCriteria, che in poche parole, rappresenta non altro che il pattern Builder per il comparator.
Facendo così sono riuscito ad avere una soluzione che mi permetterà un futuro realizzare un repository che fa l'accesso al DB e usa SortCriteria che applica le trasformazioni ai query SQL, invece di fare il build di un comparator.

Per la Pagination tutto e molto semplice – e un metodo per rappresentare la paginazione rappresentata tramite l'indice e la lunghezza di pagina. Per costruirla avevo usato di nuovo il static factory method.

Per la Specification avevo usato il pattern Composite, per creare le specificazioni composte da altri tramite 'and', 'or' e 'not'.

Application

Nella questa sezione avevo definito tutta la logica applicativa:

- Comandi con i suoi gestori
- Queries con i suoi gestori
- Validatori
- PipelineBehaviours che vengono usati per definire gli step da applicare nella pipeline di esecuzione di un comando
- Gestori di eventi di dominio

Tutti questi argomenti, tranne validatori, si riferiscono a Mediator.

Per rendere l'implementazione di questo Mediator più estensibile possibile avevo deciso di fare una versione 'trend', quale e un misto di mediator e Observer.

Lo scopo principale di Mediator e di eliminare le dipendenze mutuali da vari componenti di sistemi e far dipendere tali sistemi dal Mediator.

L’Observer in questo caso viene usato in seguente modo:

- avendo un Handler voglio sottoscrivermi a Mediator per ricevere le richieste/notifiche del mio tipo.
- quando il Mediator riceverà la richiesta/notifica del mio tipo mi avviserà passandola a me.

Cioè in poche parole il mediator fara il ruolo di Publisher delle richieste e notifiche a tutti i suoi sottoscrittori che sono interessati a riceverli.

Per supportare la possibilità di sottoscriversi per qualsiasi tipo di comando/notifica dovevo per forza usare il tipo di notifica, per non aggiungere la richiesta di specificare il nome di notifica sono andato dirittamente con l’utilizzo di Class, che pero mi ha forzato di usare il down cast più tardi. Pero tale problema sia risolto semplicemente, perché quando registravo un gestore di comando e il tipo di comando garantivo a compile-time che i tipi saranno corrispondenti tra di sé, pertanto quando facevo il down cast mi era garantito che non avrò nessun errore.

Tramite queste funzionalità vengono definiti comandi/query con i suoi gestori e i gestori di eventi di dominio.

Ma se avendo la funzionalità congiunta tra vari comandi come la validazione del comando ricevuto/logging e etc., come li posso gestire senza dover dupplicare il codice?

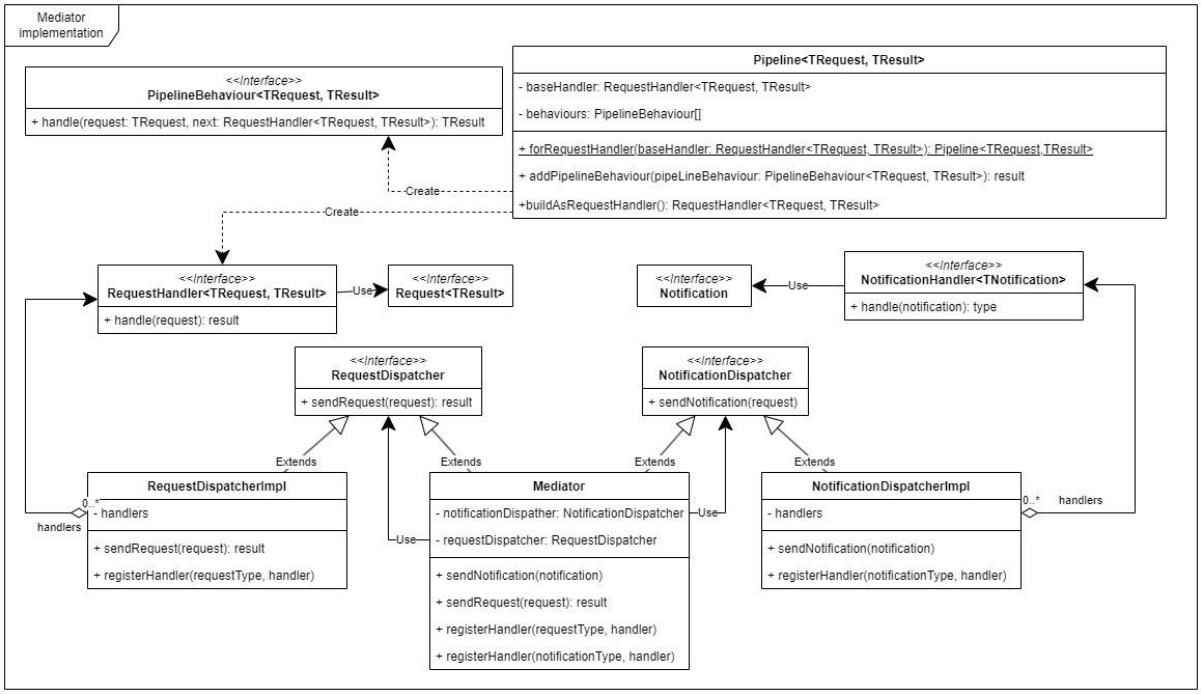
Per risolvere il problema del genere avevo deciso di introdurre un altro pattern Pipeline, che assomiglia al pattern di ChainOfResponsibility.

Pipeline al suo posto non richiede che nella catena di gestori dovrà essere solo uno a prendere la responsabilità di gestire la richiesta. Rende anche possibile le modifiche sulla richiesta/risultato dopo oppure prima di aver chiamato il prossimo gestore.

Nella prattica avevo usato il pattern Builder per definire l’elenco di PipelineBehaviour da usare specificando il RequestHandler che sarà presente alla fine della catena di pipeline.

Puoi aggregavo i metodi di PipelineBehaviour con il RequestHandler in modo da costruire il RequestHandler, tale che esegue tutta la catena di metodi rappresentati da pipeline cosi definito.

Dimostro il diagramma di classi usati per il mediator



Altro

Altre cose da menzionare:

- Logging con il supporto di MessageFormat con le date definite dentro java.time.*
- LazyLoading
- Result
- Dependency injection

Avevo implementato il logger perché lo volevo rendere estendibile per vari gestori di log. Il logger all'interno di sé a un oggetto di LogEntryPublisher che serve per poter pubblicare i log da qualche parte.

Il log entry e un oggetto che descrive tutta l'informazione di log effettuato: parametri elencati, messaggio originale, messaggio compilato e etc.

Per rendere il logging più efficiente avevo usato il pattern di LazyLoading, che realizza il calcolo di un campo in maniera 'lazy', cioè alla richiesta, invece di eseguirlo subito.

Un'altra cosa importante e come definire formattare il messaggio di log.

Avevo deciso di andare in una direzione diversa rispetto a C/C++ e usare il MessageFormat, che prevede la formattazione più intelligente con il supporto di varie funzionalità di java (come la Date e etc.).

Pero il MessageFormat non supporta il pacchetto java.time.*. Allora per questo scopo dovevo usare il pattern Decorate in modo da estendere il gruppo di oggetti gestito MessageFormat.

Volevo avere la possibilità di usare direttamente le classi da java.time.*, cioè volevo evitare l'utilizzo forzato di decorazione di java.time.*.

Non potevo neanche usare il pattern Visitor in modo da definire per tutti le classi da java.time.* come tali devono essere convertite a Date di java.utils.*.

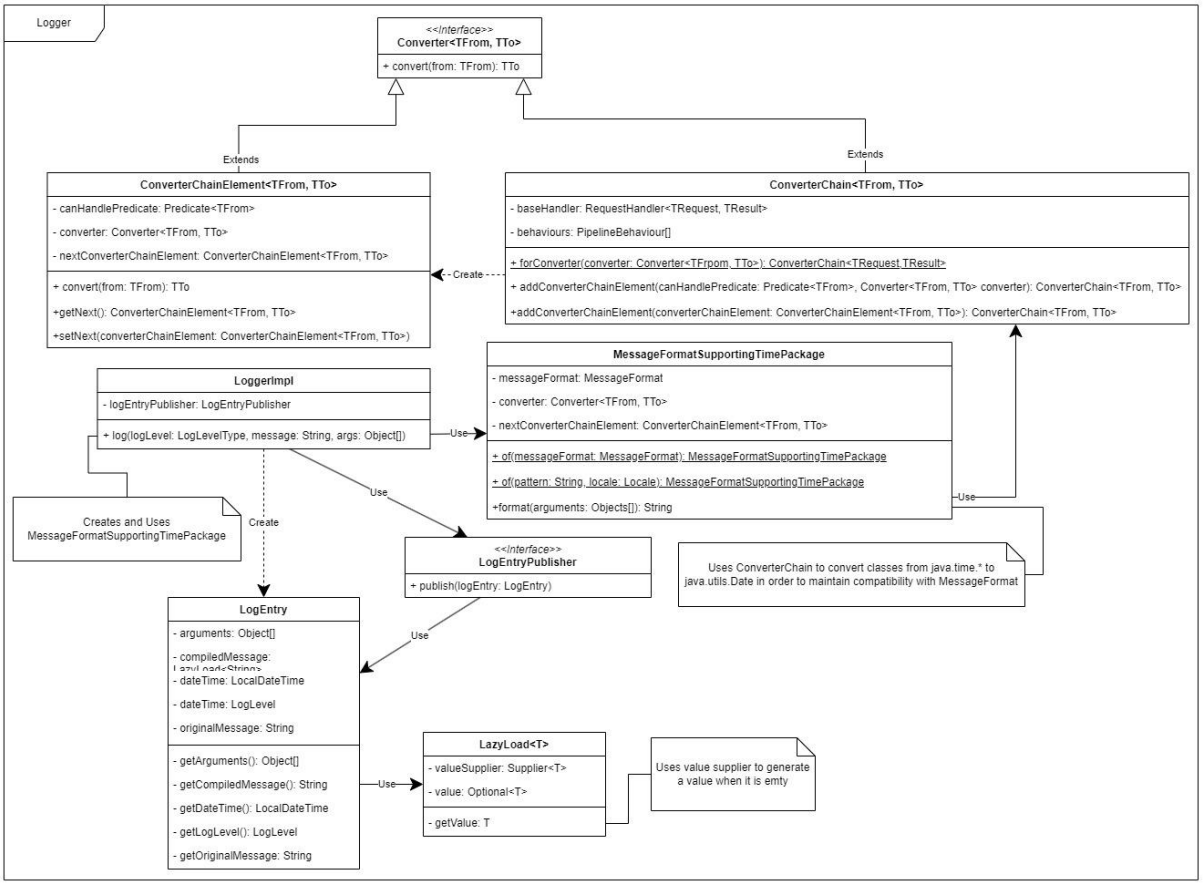
Allora in questa situazione per forza dovevo usare il down cast (usando ovviamente prima il controllo instanceof) sulle classi di java.time.*.

Avendo deciso che sono costretto di usare la combinazione di instanceof-downcast per tutte le classi che volevo convertire da java.time.* a java.Utils.Date ho cercato di migliorare la qualità del codice.

Siccome la logica di conversione tra varie implementazioni eseguita in seguente modo: avendo trovato la corrispondenza per il tipo, faccio il downcast e restituisco il risultato di conversione verso il Date; Avevo notato che questa logica assomiglia abbastanza al pattern di ChainOfResponsibility, dove solo un if-instanceof gestisce la conversione. Cosi sono arrivato all'implementazione di ConverterChain.

Il ConverterChain ha un converter di base, che garantisce che almeno lui gestirà la conversione in caso quando i predecessori non la avevano fatto. E puoi iniziavo a definire la catena dall'inizio verso la fine con il converter di base.

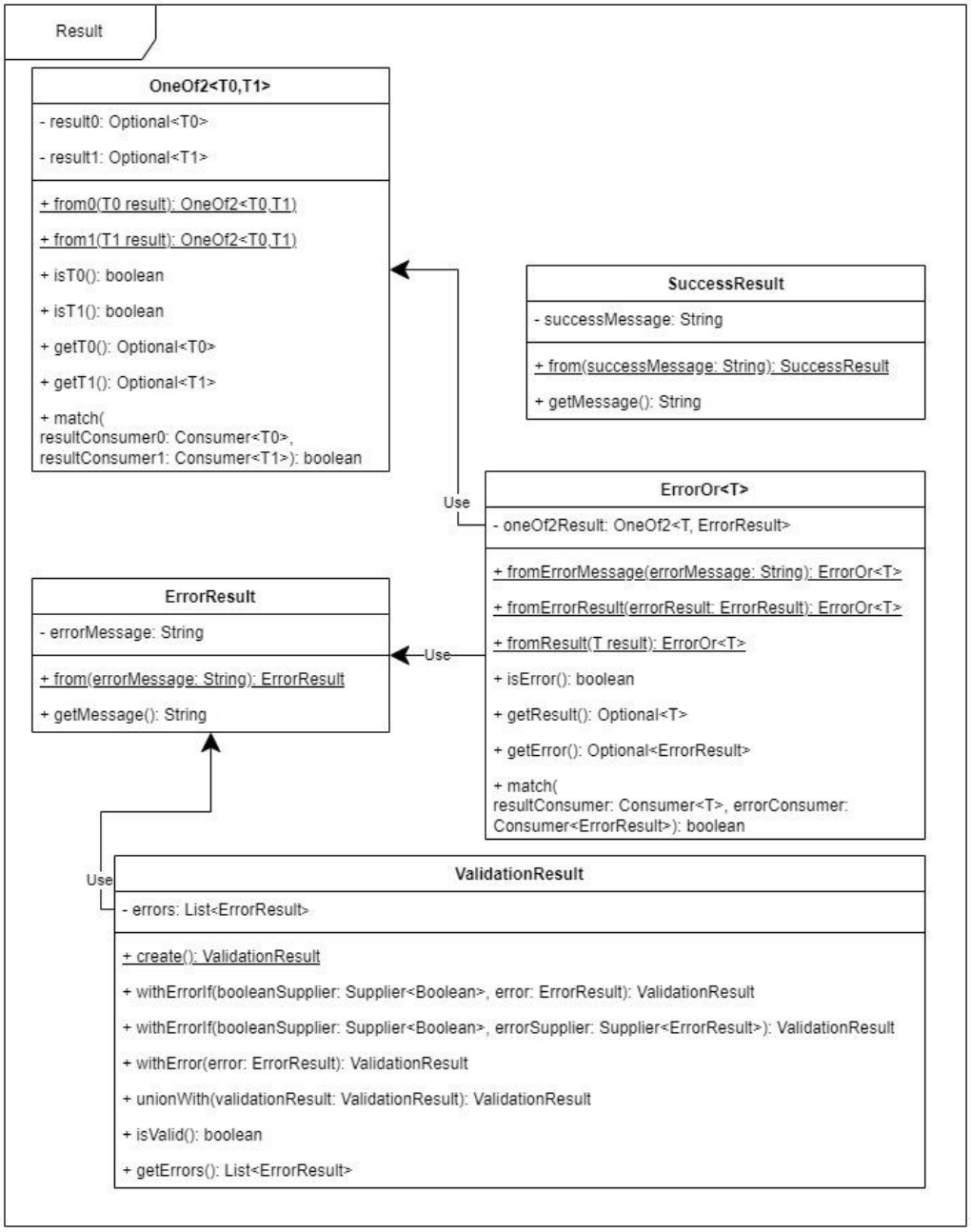
Dimostro il diagramma per l'implementazione di Logger:



Un altro pattern importante che avevo utilizzato era Result. Lui serve per poter evitare ad 'utilizzare il try-catch come il flow di applicazione.

L'implementazione di questo pattern in progetto è SuccessResult, ErrorResult che hanno il messaggio descrivente il risultato, OneOf2<T1, T2>/ErrorOr<T> che riportano due possibili risultati di operazione, l'ErrorOr che usa all'interno di sé OneOf2 con il ErrorResult come uno tra i possibili risultati. ValidationResult che rappresenta il risultato di validazione, cioè dice se l'entità e valida oppure non valida con la lista di ErrorResult.

Dimostro il diagramma per l'implementazione di Result pattern:


Result
OneOf2<T0,T1>
- result0: Optional<T0>
- result1: Optional<T1>
+ from0(T0 result): OneOf2<T0,T1>
+ from1(T1 result): OneOf2<T0,T1>
+ isT0(): boolean
+ isT1(): boolean
+ getT0(): Optional<T0>
+ getT1(): Optional<T1>
+ match(
resultConsumer0: Consumer<T0>,
resultConsumer1: Consumer<T1>): boolean
SuccessResult
- successMessage: String
+ from(successMessage: String): SuccessResult
+ getMessage(): String
Use
ErrorOr<T>
- oneOf2Result: OneOf2<T, ErrorResult>
+ fromErrorMessage(errorMessage: String): ErrorOr<T>
+ fromErrorResult(errorResult: ErrorResult): ErrorOr<T>
+ fromResult(T result): ErrorOr<T>
+ isError(): boolean
+ getResult(): Optional<T>
+ getError(): Optional<ErrorResult>
+ match(
resultConsumer: Consumer<T>, errorConsumer:
Consumer<ErrorResult>): boolean
ErrorResult
- errorMessage: String
+ from(errorMessage: String): ErrorResult
+ getMessage(): String
Use
Use
ValidationResult
- errors: List<ErrorResult>
+ create(): ValidationResult
+ withErrorIf(booleanSupplier: Supplier<Boolean>, error: ErrorResult): ValidationResult
+ withErrorIf(booleanSupplier: Supplier<Boolean>, errorSupplier: Supplier<ErrorResult>): ValidationResult
+ withError(error: ErrorResult): ValidationResult
+ unionWith(validationResult: ValidationResult): ValidationResult
+ isValid(): boolean
+ getErrors(): List<ErrorResult>

Conclusione

Pattern utilizzati nel progetto:

- Builder
- Static Factory method
- Template method
- Chain of responsibility
- Decorator
- Composite
- Prototype
- Repository
- Mediator
- Pipeline
- Lazy loading
- Result

Metodologie applicate:

- SOLID
- TDD (tranne lo stratto Application, dove teoreticamente devono essere usati i IntegrationTest che non fanno parte del mio progetto)
- DDD (Domain Driven Design) con la gestione di eventi di dominio (DomainEvents)
- CQRS (Command Query Responsibility Segregation)