Università degli Studi di Salerno

Corso di Ingegneria del Software
Classe 1 Resto 0
Corso di Laurea in Informatica
A.A. 2022/23

Quiad
Object Design Document

Versione 4.2
15/12/2022

Partecipanti al progetto e scriventi

Nome	Matricola
Di Pasquale Valerio	0512110638
Troisi Vito	0512109807

Revision History

Data	Versione	Descrizione	Autore
30/11/2022	1.0	Prima stesura ODD e Indice	D.P.V. T.V.
02/12/2022	2.0	Sezione introduttiva e vincoli in OCL	D.P.V. T.V.
08/12/2022	2.1	Scheletro sezione Interfaccia delle classi da integrare con modelli ristrutturati	T.V.
09/12/2022	3.0	Trade-offs, organizzazione in directory	T.V.
10/12/2022	4.0	Primi contratti classi solution domain, caching server side, sviluppati CD-OD 1, CD-OD 2, SD-OD 1	D.P.V. T.V.
15/12/2022	4.1	Fine contratti	D.P.V. T.V.
15/12/2022	4.2	Note sul mapping	T.V.

Indice

Overview

Nel presente documento, sarà esplicitata la progettazione di dettaglio del sistema Quiad, per la gestione del proprio albero genealogico.

Lo Object Design Document vuole presentare i trade-offs di progettazione e, per quanto concerne la *slice* del sistema che sarà implementata, la specifica delle interfacce dei moduli e l'organizzazione delle classi dei medesimi. Seguono la presente sezione introduttiva, alcune linee guida utili per la comprensione delle scelte di design e di specifica delle interfacce suddette.

Trade-offs di Object Design

In aggiunta ai trade-offs evidenziati in fase di progettazione architetturale se ne evidenziano alcuni relativi alla progettazione di dettaglio dello stralcio del sistema presentato nei paragrafi seguenti.

- **Spazio occupato VS Aggiornamento continuo**

  Per le medesime ragioni illustrate in fase di progettazione di sistema e viste le scelte illustrate nel RMD sezione 2.1, sarà prevista una cache nella quale conservare i documenti recentemente acceduti. Ciò consente di evitare l'uso della rete per accedere al database e poi al file system per recuperare documenti i quali, si osserva, non sono generalmente soggetti a cambiamenti.

Linee guida per la documentazione delle interfacce

Le interfacce delle classi dei moduli che saranno soggetti ad implementazione, saranno specificate tenendo conto delle seguenti prassi:

- I nomi di attributi, metodi e parametri saranno esplicitati in camel-case.
- I nomi dei metodi getter/setter legati ad un attributo seguiranno la forma "get...()" e "set...()", imponendovi il nome dell'attributo.

- Precondizioni, postcondizioni ed invarianti di classe saranno esplicitati mediante OCL (Object Constraint Language).
- Le classi Route, Service, Controller e Model utilizzate per implementare i servizi dei sottosistemi saranno precedute dal nome della classe del dominio applicativo(od eventualmente da un nome significativo per un insieme di classi) con le quali interagiscono.

Definizioni, acronimi, abbreviazioni

Una lista alfabetizzata di definizioni ed acronimi utili per la lettura della presente:

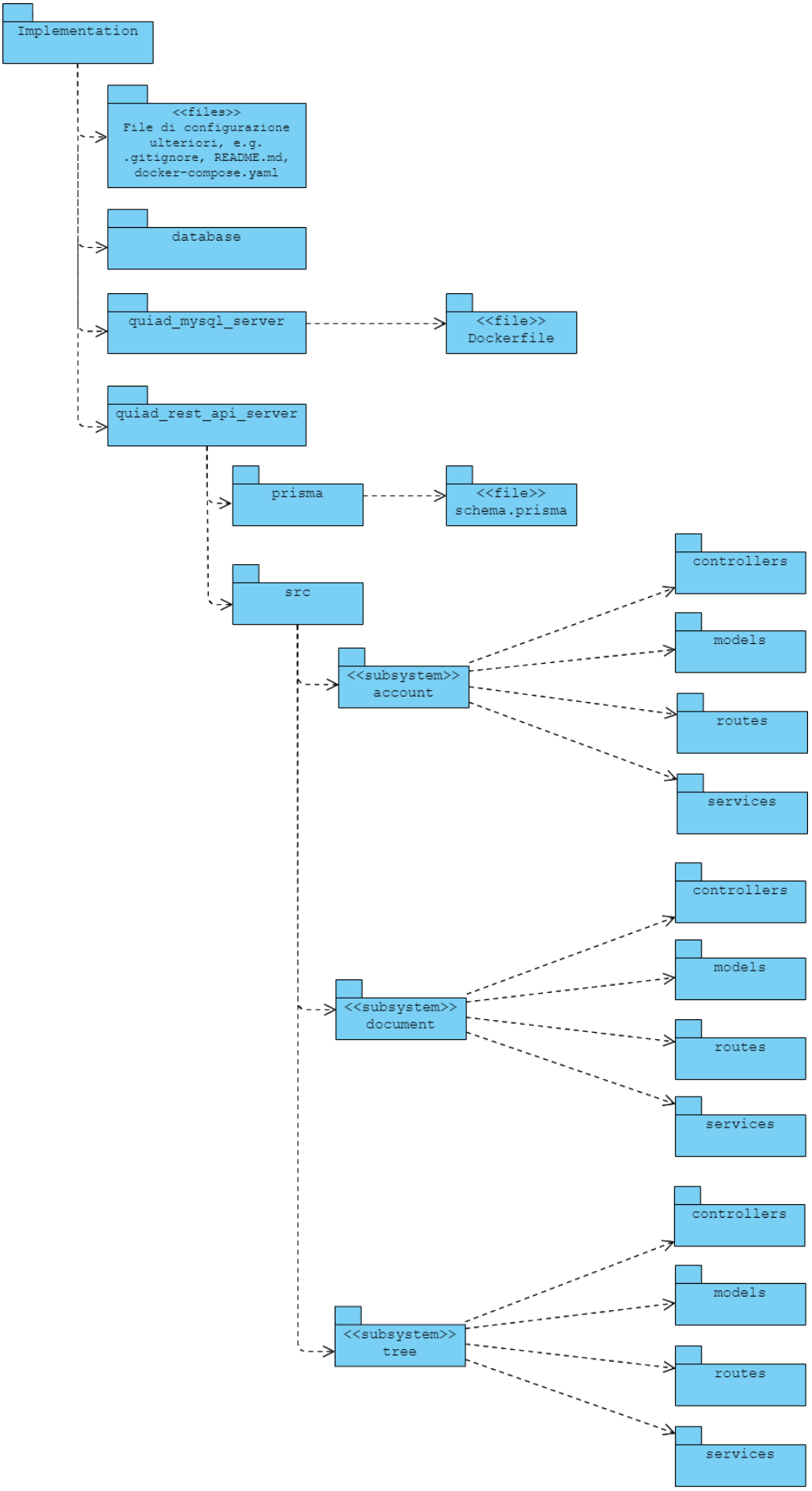
- CD-OD X: Class Diagram [relativo a] Object Design con identificativo X.
- SD-OD X: Sequence Diagram [relativo a] Object Design con identificativo X.
- RAD: Requirements Analysis Document.
- RMD: Rationale Management Document, i.e. il documento in cui è riportato il razionale dietro alcune scelte fatte in fase di progettazione.
- OCL: Object Constraint Language, il quale sarà impiegato per esplicitare i contratti di ogni metodo, e per ciascuna classe.
- ODD: Object Design Document, il presente documento.
- SDD: System Design Document, i.e. il documento di progettazione architetturale del sistema.

Riferimenti

Al fine di garantire una migliore comprensione dello ODD, si invita ad osservare il Requirements Analysis Document, in particolar modo le sezioni 3.5.3 e 3.5.4, rispettivamente legate alla modellazione ad oggetti ed alla modellazione dinamica.

Si invita inoltre a consultare il SDD per analizzare le scelte legate alla progettazione di sistema, quale la gestione dei dati persistenti, esplicitata alla sezione 2.4.

Organizzazione delle directory



Per quanto concerne le linee guida generali dell'organizzazione delle classi del progetto in directory, si osservi che ciascun sottosistema (riferirsi al SDD, sezione 2.2) sarà mappato su una precisa directory. Nel dettaglio, la struttura si presenta come mostrato in precedenza. In tale package diagram, ogni package è da intendersi come una directory, ed ogni arco orientato come la relazione "contiene".

Interfacce delle classi

Il Class Diagram presentato nel RAD (paragrafo 3.5.3) richiede una ristrutturazione, al fine di evitare futili ridondanze. Un utente base (o curatore) può infatti essere ben rappresentato dal proprio nodo, con ogni informazione inserita dall'utente stesso in fase di registrazione. Conseguentemente, è stata definita una nuova astrazione, Account, che ristruttura lo schema come presentato di seguito. Al medesimo sono state aggiunte le informazioni circa visibilità e tipi dei membri delle classi.

Si osservi che, per implementazioni future, la classe Curator estenderà User e la classe Supervisor estenderà Account, così da mantenere la consistenza (e la stessa semantica) rispetto al Class Diagram descritto in fase di Object Modeling.

[REFACTORED CD-OD 1]

Inoltre, si osservi che l'albero e l'utente sono coinvolti in un'associazione 1-1. E' possibile accorpare l'albero nell'utente, che si occuperà di gestire la collezione di nodi.

[REFACTORED CD-OD 2]

Seguono i contratti di classi e metodi proposti, specificati mediante OCL.

Classe UtenteBase:

```
context UtenteBase::addNode(n) pre:
     n != null Λ
     n.getFather() != n Λ
     n.getMother() != n Λ
     tree->include(n.getFather()) Λ
     tree->include(n.getMother())
```

```
context UtenteBase::addNode(n) post:
     tree->include(n) Λ
     tree->n.getFather().getDescendants()->asSet()->include(n) Λ
     tree->n.getMother().getDescendants()->asSet()->include(n)
```

```
context UtenteBase::modifyNode(m,n) pre:
     n != null Λ
     m != null Λ
     tree->include(m) Λ
     m.getFather() = n.getFather() Λ
     m.getMother() = n.getMother()
```

```
context UtenteBase::modifyNode(m,n) post:
     !tree->include(m) Λ
     tree->include(n)
```

```
context UtenteBase::deleteNode(n) pre:
     n != null Λ
     tree->include(n) Λ
     n != getUserNode()
```

```
context UtenteBase::deleteNode(n) post:
     !tree->include(n) Λ
     forAll(s|n.getDescendants()) !tree->include(s)
```

Classe Node:

context Node::bindDocument(d) **pre:** d != null Λ !self.getBoundDocuments()->asSet()->include(d)
context Node::bindDocument(d) **post:** self.getBoundDocuments()->asSet()->include(d)
context Node::unbindDocument(d) **pre:** d != null Λ self.getBoundDocuments()->asSet()->include(d)
context Node::unbindDocument(d) **post:** !self.getBoundDocuments()->asSet()->include(d)

Il modello dei dati può poi essere mappato su un DB relazionale, in accordo con quanto esplicitato nel SDD, sezione 2.4.

Mapping classi-relazioni

- Role(**RoleID**, RoleName)
- Operation(**OperationID**, OperationName)
- Capability(Role.***RoleID***, Operation.***OperationID***)
- Category(**CategoryID**, CategoryName, Description)
- Account(**AccountID**, Email, Username, Password)
- Node(**NodeID**, FirstName, LastName, BirthDate, DeathDate, BirthPlace, DeathPlace, Sex, Node.*FatherID*, Node.*MotherID*)
- User(**UserID**, Residence, *AccountID*, Node.*NodeID*)
- Document(**DocumentID**, DocumentPath, RetrievalDate, RetrievalPlace, OriginDate, OriginPlace, DocumentName)
- NodeDocuments(Node.***NodeID***, Document.***DocumentID***)

N.B. Il grassetto indica una chiave primaria ed il corsivo un vincolo di integrità referenziale (i.e. chiave esterna) alla chiave primaria di una data relazione ulteriore.

Il modello ad oggetti può essere integrato con le classi del dominio delle soluzioni.

Gli oggetti di tipo Route hanno il ruolo di associare i metodi dei servizi ad un *endpoint* della REST API.

Gli oggetti Service traducono le richieste ricevute dai *client* in metodi legati agli oggetti del dominio applicativo. Il collateral effect è la restituzione di una risposta HTTP al client.

I metodi delle classi di tipo Controller, ricevono i parametri tradotti dal Service e sulla base di questi effettuano operazioni sul database per mezzo di oggetti dell'ORM Prisma. Conseguentemente, il collateral effect sarà l'interazione (lettura o scrittura) con il database.

[REFACTORED CD SOLUTION 1]

[SOLUTION DOMAIN SD]

Un'ottimizzazione è d'obbligo circa i cammini di accesso: al fine di ottenere l'insieme dei nodi nell'albero di un utente a partire da un dato nodo in esso inserito, è utile mantenere il riferimento all'albero di appartenenza in ciascun nodo. E' dunque conservato il riferimento al nodo utente in ciascuna istanza di Node.

[REFACTORED CD SOLUTION 2]

Seguono i contratti delle classi del solution domain:

Classe AuthService:

```
context AuthService::login(req) pre:
     req.body.username.length() > 0
     req.body.password.length() > 0
```

```
context AuthService::login(r) post:
     true
```

```
context RegistrationService::register(req) pre:
     req.body.account != null
```

```
context RegistrationService::register(req) post:
     true
```

Classe AccountController:

```
context AccountController inv:
     prismaClient != null
```

```
context AccountController::findByUsername(username) pre:
     username != null
```

```
context AccountController::findByUsername(username) post:
     result.getUsername() = username
```

```
context AccountController::createAccount(account) pre:
     account != null Λ
     account.getUsername().length() > 0 Λ
     account.getPassword().length() > 0 Λ
     account.getEmail().length() > 0 Λ
     account.getUser().getResidence().length() > 0 Λ
```

```
     account.getUser().getNode().getFirstName().length() > 0 Λ
     account.getUser().getNode().getLastName().length() > 0 Λ
     account.getUser().getNode().getBirthDate() != null Λ
     account.getUser().getNode().getBirthPlace().length() > 0 Λ
     account.getUser().getNode().getSex() != null
```

```
context AccountController::createAccount(account) post:
     result != null Λ
     result.getRole().getName() = "UtenteBase"
```

Classe DocumentService:

```
context DocumentService::findDocuments(req) pre:
     true
```

```
context DocumentService::findDocuments(req) post:
     true
```

```
context DocumentService::createDocument(req) pre:
     req.body.document != null
```

```
context DocumentService::createDocument(req) post:
     true
```

Classe DocumentController:

```
context DocumentController inv:
     prismaClient != null
```

```
context DocumentController::findDocuments(filter) pre:
     filter != null
```

```
context DocumentController::findDocuments(filter) post:
     forAll(d|result->asSet())
     d.getRetrievalDate() != null ⇒
          d.getRetrievalDate()= filter.getRetrievalDate() Λ
     d.getRetrievalPlace() != null ⇒
          d.getRetrievalPlace()= filter.getRetrievalPlace() Λ
     d.getOriginDate() != null ⇒
          d.getOriginDate() = filter.getOriginDate() Λ
     d.getOriginPlace() != null ⇒
          d.getOriginPlace() = filter.getOriginPlace()
```

```
context DocumentController::createDocument(document) pre:
     document != null
```

```
context DocumentController::createDocument(document) post:
     result != null
```

Classe TreeService:

```
context TreeService::getNodes(req) pre:
     req.params.owner != null
```

```
context TreeService::getNodes(req) post:
     true
```

```
context TreeService::createNode(req) pre:
     req.body.node != null
```

```
context TreeService::createNode(req) post:
     true
```

```
context TreeService::updateNode(req) pre:
     req.params.id != null Λ
     req.body.node != null
```

```
context TreeService::updateNode(req) post:
     true
```

```
context TreeService::deleteNode(req) pre:
     req.params.id != null
```

```
context TreeService::deleteNode(req) post:
     true
```

```
context TreeService::bindDocument(req) pre:
     req.params.nodeID != null Λ
     req.params.documentID != null
```

```
context TreeService::bindDocument(req) post:
     true
```

```
context TreeService::unbindDocument(req) pre:
     req.params.nodeID != null Λ
     req.params.documentID != null
```

```
context TreeService::unbindDocument(req) post:
     true
```

Classe NodeController:

```
context NodeController inv:
     prismaClient != null
```

```
context NodeController::getNodes(owner) pre:
     owner!= null
```

```
context NodeController::getNodes(owner) post:
     forAll(n|result->asSet()) n.getOwner() = owner
```

```
context NodeController::createNode(node) pre:
     node != null Λ
     node.getOwner() != null
```

```
context NodeController::createNode(node) post:
     result != null
```

```
context NodeController::updateNode(id, node) pre:
     id != null Λ
     node != null
```

```
context NodeController::updateNode(id, node) post:
     node.getFirstName() != null ⇒
          result.getFirstName()= node.getFirstName() Λ
     node.getLastName() != null ⇒
          result.getLastName()= node.getLastName() Λ
     node.getBirthDate() != null ⇒
          result.getBirthDate()= node.getBirthDate() Λ
     node.getDeathDate() != null ⇒
          result.getDeathDate()= node.getDeathDate() Λ
     node.getBirthPlace() != null ⇒
          result.getBirthPlace()= node.getBirthPlace() Λ
     node.getDeathPlace() != null ⇒
          result.getDeathPlace()= node.getDeathPlace() Λ
```

```
     node.getSex() != null ⇒
          result.getSex()= node.getSex()
```

```
context NodeController::deleteNode(id) pre:
     id != null
```

```
context NodeController::deleteNode(id) post:
     result != null
```

```
context NodeController::bindDocument(nodeID, documentID) pre:
     nodeID != null Λ
     documentID != null
```

```
context NodeController::bindDocument(nodeID, documentID) post:
     result.getDocument().getID() = documentID
```

```
context NodeController::unbindDocument(nodeID, documentID) pre:
     nodeID != null Λ
     documentID != null
```

```
context NodeController::unbindDocument(nodeID, documentID) post:
     result.getDocument() = null
```

[Descrivere i design pattern utilizzati (proxy) e i trade-offs che comportano]

Note sul mapping dei modelli e delle scelte sul codice

In precedenza sono state inevitabilmente introdotte alcune ottimizzazioni e ristrutturazioni, implicitamente legate al

mapping dei modelli sul codice. Linee guida e strategie ulteriori sono descritte nella sezione presente.

Al fine di evitare la ripetizione di letture da database e file system dei documenti, e come esplicitato nel paragrafo 2.1 del RMD caching è gestito:

- Server side: mediante API-cache, i documenti recuperati dal database sono mantenuti in una cache per 1 ora. All'invocazione dell'operazione findDocuments() del servizio DocumentService, DocumentRoute interpone un metodo che verifica la presenza dei documenti nella cache prima dell'invocazione stessa.

- Client side: il modulo Node.js NgHttpCaching consente di porre in cache le richieste HTTP: nel caso esaminato ciò sarà effettuato relativamente alle richieste di ricerca documenti.

Le associazioni riportate nel Class Diagram saranno mappate traducendo:

- Le associazioni 1-1 come variabili di istanza riferite all'istanza associata;
- Le associazioni 1-N imponendo una collezione Set (o List, se ordinata).