BEEMEDIATE

Giorgio Chirico (matr. 1068142)

Project Work per Gestione Dei Sistemi ICT e Testing

Università Degli Studi Di Bergamo

a.a. 2024/2025

Sommario

Introduzione

Il software da realizzare è un componente per le operazioni di e-procurement per CFG Serramenti. Il componente svolgerà il ruolo di middleware di integrazione per il software aziendale di CFG verso il software fornitore GEALAN, con l'obiettivo di automatizzare il flusso informativo, provvedendo alla conversione del formato dati ed alla gestione dello scambio di messaggi tra le piattaforme CRM in uso: rispettivamente, ODOO per CFG e SAP per GEALAN.

CFG utilizza il CRM ODOO per avviare la produzione di determinati prodotti o chiedere una fornitura per il magazzino. Quando uno di questi eventi accade, il database viene popolato con i componenti da richiedere a diverse aziende fornitrici. Tali componenti avranno delle variabili di stato aggiuntive, cosicchè l'operatore ODOO possa segnalare alla piattaforma automatica una richiesta di caricamento, oltre che avere un feedback sullo stato dell'ordine. Agendo con una strategia di polling, il middleware scansionerà periodicamente il database per ricercare i componenti da ordinare. Tali componenti verranno caricati in una coda sul middleware, in attesa di raggiungere un volume batch minimo per essere evacuati. Il volume minimo è anch'esso specificabile dall'operatore ODOO.

Il middleware avrà il compito di generare la struttura dati compatibile con SAP GEALAN e comprensiva di tutti i campi necessari. La comunicazione richiede diverse condizioni:

- Specifici campi obbligatori, alcuni con valori obbligatori, pattuiti col fornitore, o altre condizioni;
- Protocollo FTP con regole sulla gestione delle directory;
- La struttura dati per gli ordini dev'essere XML OpenTrans.

A supporto, il middleware può validare i dati ricevuti da CFG e restituire un feedback, garantendo una minima gestione degli errori nei dati forniti dall'operatore.

Durante la conversione, strutture di supporto come MasterData_IT.xls possono essere utili.

Il middleware è asincrono, non ci sono richieste real-time. A seguito di un ordine, la conferma può arrivare anche diversi giorni dopo.

Nel FTP si può conservare un archivio degli ordini passati.

Requisiti funzionali

Si descrivono gli use case stories degli attori coinvolti nel sistema:

- Operatore ODOO
- fornitore GEALAN

Use Case stories: Operatore ODOO

L'operatore ODOO ha l'incarico di avviare l'Ordine di Acquisto (OdA), generando la Bills Of Materials. Una volta generata la Bills Of Materials, l'operatore seleziona gli elementi della Bills Of Materials da caricare sulla piattaforma middleware. Questi elementi verranno quindi convalidati in un secondo momento dalla piattaforma.

IMPOSTARE OdA COME OaF

- Come operatore ODOO, voglio segnalare al middleware di caricare i miei Ordini di Produzione sulla sua piattaforma, così da poter preparare in automatico le batch di ordini da inviare a GEALAN. Gli OdA sulla piattaforma verranno denominati Ordini a Fornitore (OaF);

CARICAMENTO AUTOMATICO IN CODA BATCH

- Come operatore ODOO, voglio un caricamento automatico degli OaF in una coda sul middleware;

VALIDAZIONE OaF

- Come operatore ODOO, voglio ricevere un feedback sulla validità dei dati degli OaF secondo la configurazione pattuita col fornitore;

INVIO BATCH ORDINI

- Come operatore ODOO, voglio che gli ordini vengano automaticamente inviati a GEALAN al raggiungimento della soglia minima per la dimensione batch;

IMPOSTAZIONE SOGLIA BATCH ORDINI

- Come operatore ODOO, voglio impostare il numero minimo di Ordini a Fornitore per la batch da mandare.

CONTROLLO STATO ORDINE

- Come operatore ODOO, voglio verificare lo stato del singolo OaF fino alla ricezione di conferma OdA da parte del fornitore GEALAN;
- Come operatore ODOO, voglio verificare la presenza di errori nella formattazione dei dati dell'Ordine a Fornitore;

ARCHIVIAZIONE CONFERMA ORDINE

- Come operatore ODOO, voglio un archivio delle conferme d'ordine nel server FTP, così da poter visionare lo storico degli ordini.

ACCESSO AL SISTEMA FTP

- Come operatore ODOO, voglio poter accedere al sistema FTP per poter visualizzare gli ordini archiviati o per altri motivi (ad es. troubleshooting);

Use case stories: Fornitore GEALAN

RICEZIONE BATCH ORDINI

- Come fornitore GEALAN, voglio ricevere le batch di Ordini a Fornitore così da processarli;

NOTIFICA CONFERMA ORDINE

- Come fornitore GEALAN, voglio che CFG riceva la conferma della ricezione degli Ordini a Fornitore.

Requisiti Non Funzionali

Consistenza

I dati inviati devono essere validati rispetto allo standard di configurazione pattuito col fornitore, sia nella struttura che nei valori obbligatori.

Idempotenza

Non dev'essere possibile inviare molteplici volte lo stesso ordine verso il fornitore. In altre parole, non deve presentarsi il caso accidentale in cui un ordine univocamente identificato, in un certo istante nel tempo, viene inviato molteplici volte verso il fornitore, causando equivoci di natura logistica.

Testabilità

Ogni componente deve poter permettere la progettazione, implementazione ed esecuzione di test efficaci, in modo da garantire una massima copertura di requisiti e funzionalità

Modificabilità

Il software potrebbe essere esteso o adattato per rispondere a nuove strategie. Perché ciò accada, bisogna fornire una solida base adattabile alle esigenze future.

Manutenibilità

In caso di errore, il software deve rendere possibile il troubleshooting ed i successivi interventi in tempi minimi, permettendo agli operatori di individuare nel minor tempo possibile le aree in cui agire per apportare correzioni o miglioramenti.

Portabilità

L'esecuzione del software dev'essere platform-independent.

analisi MoSCoW

Must have

- caricamento automatico degli OaF in coda Batch
- validazione OaF
- caricamento ordini sul FTP

Should have

- analisi delle conferme ricevute
- archiviazione delle conferme ricevute

Could have

- aggiornamento stato OaF
- conferma OaF

Won't have

- gestione protocollo FTP
- impostazione soglia batch ordini
- aggiornamento CRM impostazione OdA come OaF

Priorità dei casi d'uso

Lo sviluppo si soffermerà sulle funzionalità della piattaforma middleware. Nelle prime fasi, si procederà con l'assunto che i requisiti lato CRM ODOO siano già stati soddisfatti, coinvolgendo eventualmente dei mock ad es. per il database.

Priorità elevata

Use Case da implementare.

UC1	Caricamento automatico OaF in coda batch
UC2	Validazione OaF
UC3	Invio batch ordini

Priorità media

Use case che possono essere implementati.

UC4	Ricezione conferma ordine
UC5	Archiviazione conferma ordine
UC6	Aggiornamento CRM stato OaF
UC7	Aggiornamento CRM conferma OdA

Priorità bassa

Use case non essenziali per il nucleo del progetto.

UC8	Impostazione soglia batch ordini
UC9	Aggiornamento CRM impostazione OdA come OaF
UC10	Accesso sistema FTP

Use Case Diagram

Di seguito vengono evidenziate in verde i casi d'uso interessanti per il progetto, in rosso le funzionalità ODOO utili al progetto, inserite per favorire una miglior comprensione della panoramica del problema.

Il progetto prioritizzerà la realizzazione del middleware FTP. I requisiti sul CRM ODOO potrebbero inizialmente essere parte di un assunzione ambientale, funzionale allo sviluppo.

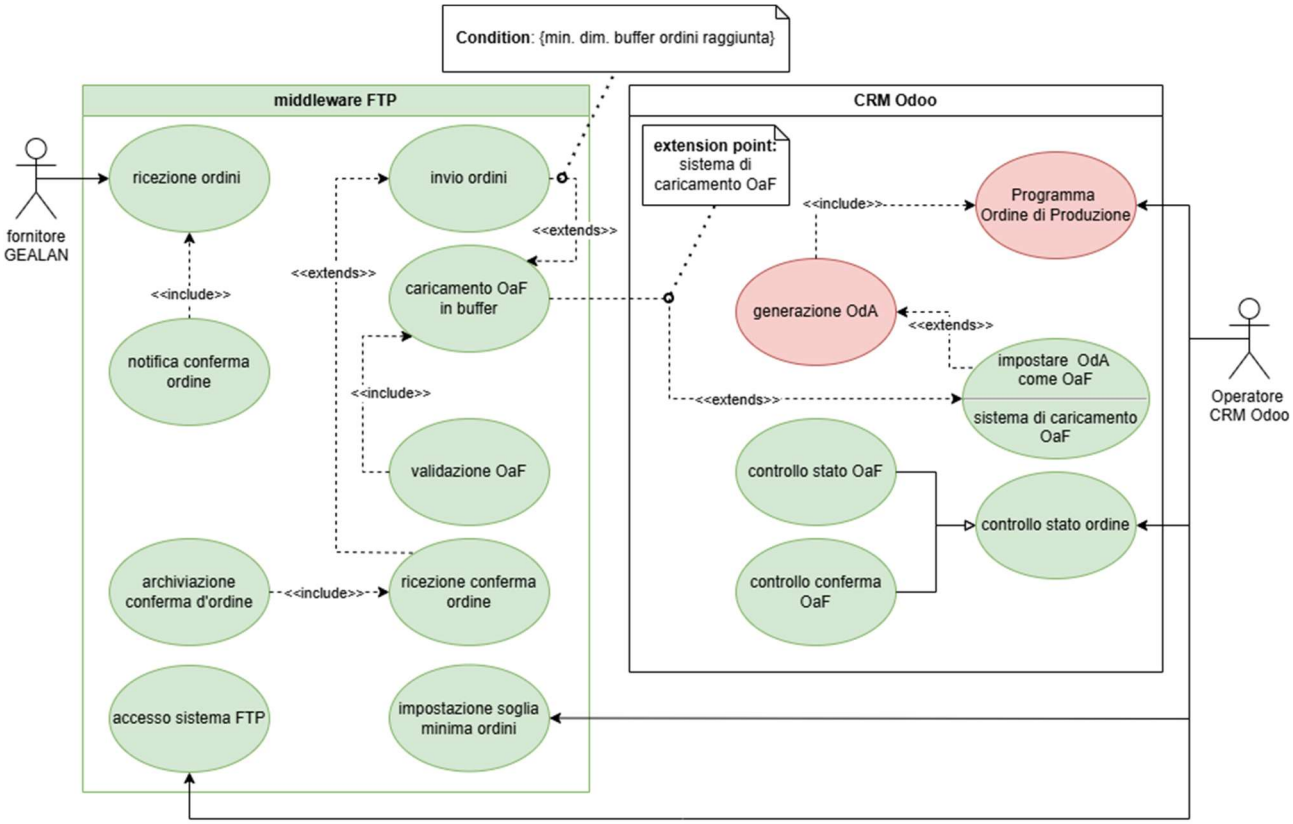


Figure 1 Use Case Diagram

Deployment Diagram

Il middleware sfrutterà un applicativo Java per comunicare con il database di CFG Serramenti ed individuare la tabella dei Bills Of Materials. Stando alla documentazione ODOO, è possibile accedere al database del CRM attraverso le External API, con protocollo XML-RPC.

Si presume che il middleware non darà overhead di tempo e spazio critici. Pertanto, non ci sono esigenze particolari per quanto concerne il nodo d'esecuzione del servizio.

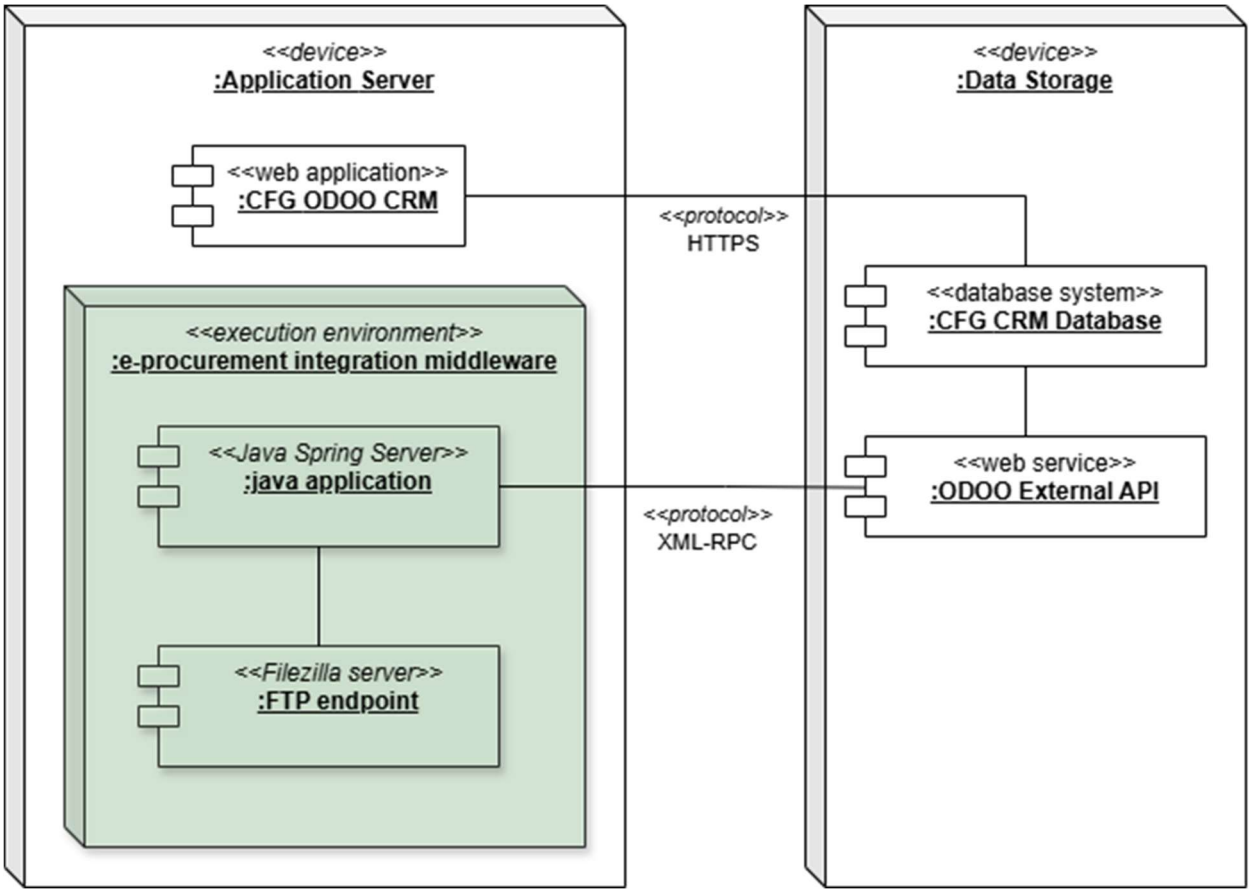


Figure 2 Deployment Diagram

Architettura

Per realizzare il software in risposta alle esigenze non funzionali, si è scelto una architettura esagonale: un design fortemente modularizzato che favorisce testing, estensibilità e modificabilità attraverso una forte separazione delle funzionalità grazie al port-adapter pattern e l'osservazione, più o meno critica, dei principi SOLID.

L'architettura esagonale può presentare overhead del codice sorgente e, come conseguenza, del tempo d'esecuzione. Tuttavia, ciò non costituisce un problema critico per le specifiche di progetto.

Component diagram

Di seguito viene illustrata la progettazione dell'applicativo Java.

Il progetto potrebbe essere modificato iterativamente in risposta a future esigenze di sviluppo.

Unique component

Visualizzazione di alto livello che presenta un componente globale.

Figure 3 Component Diagram Unique Component

First Zoom-in: system

Prima raffinazione che illustra la composizione interna dell'applicativo Java.

Con "interface" si definisce il sottosistema che avrà il compito di monitorare e caricare le informazioni da processare. Con "domain" si definisce il sottosistema contenente le funzionalità core, le quali provvedono alla gestione dei dati, l'organizzazione ed il controllo di questi. Infine, "infrastructure" definisce il sottosistema contenente le funzionalità "attuatori", cioè che hanno lo scopo di manifestare un'azione, una risposta o apportare modifiche.

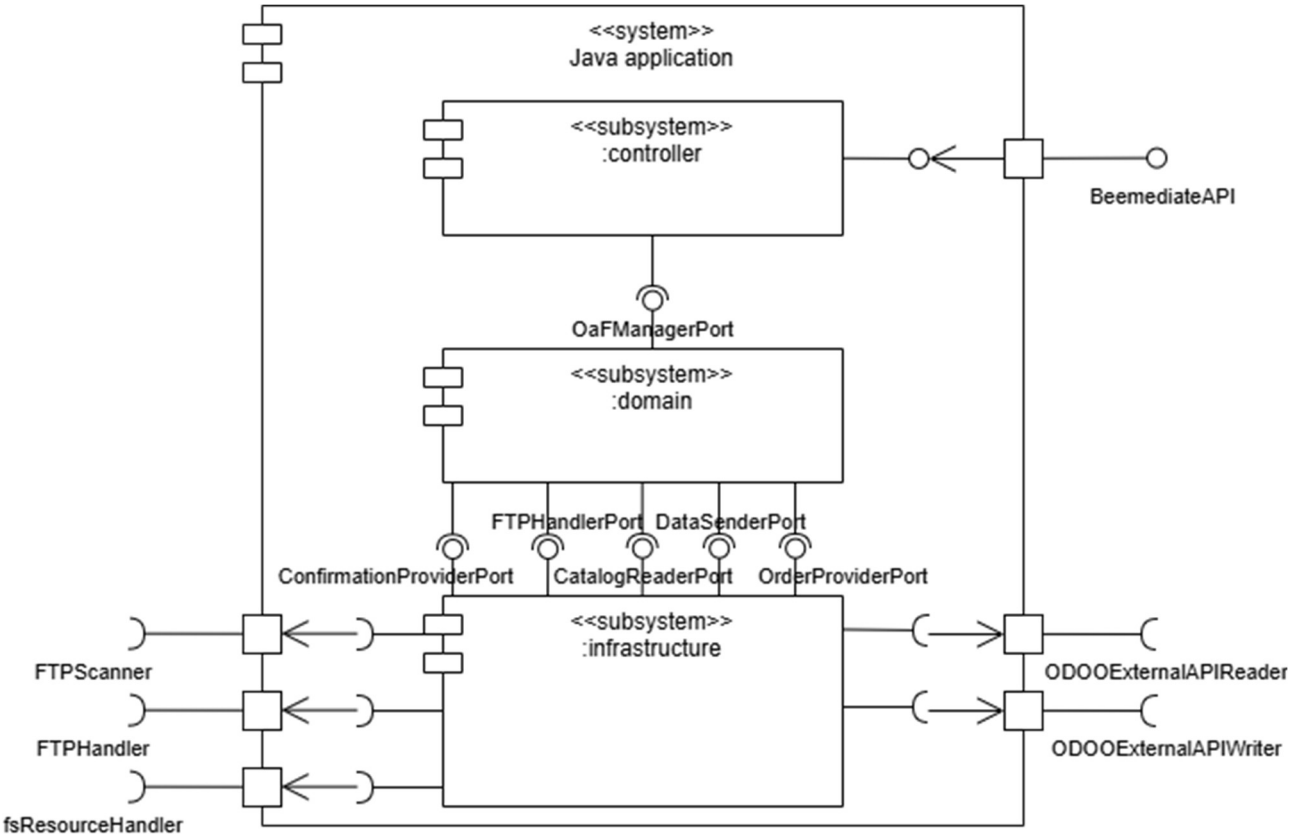


Figure 4 Component Diagram First Zoomin

First zoom-in: subsystem interface

Il sottosistema “Filestystem” contiene ConfirmationBuffer, avente lo scopo di individuare e caricare le conferme d’ordine che si trovano nel FTP. Il sottositema XMLRPC ha lo scopo di interrogare il database del sistema CRM e caricare gli OaF nuovi, non ancora presenti sul middleware.

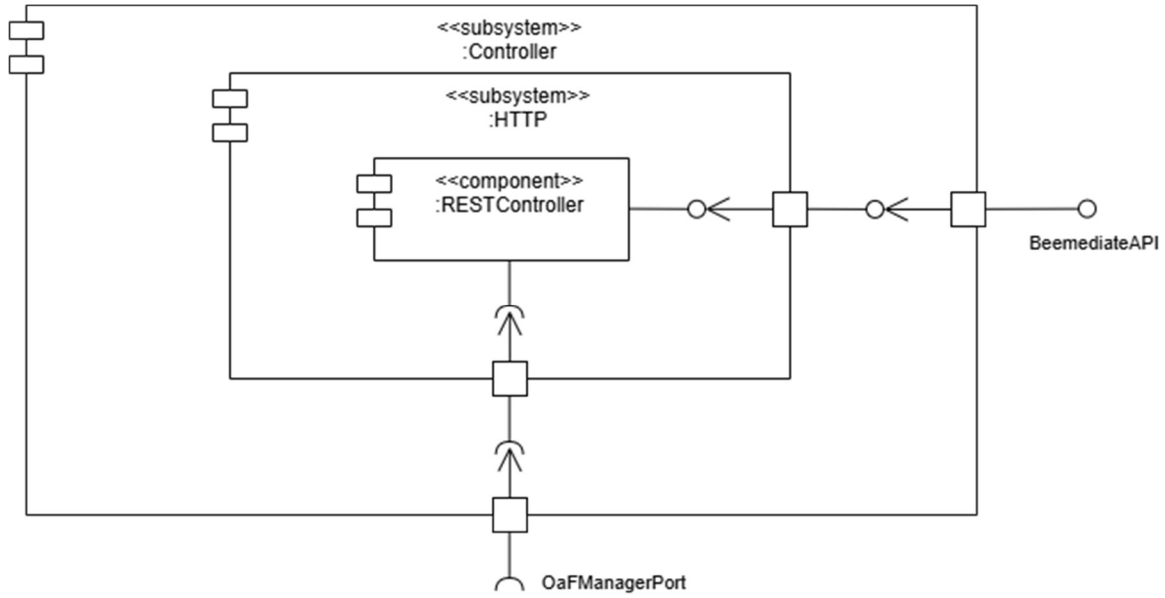


Figure 5 Component Diagram subsystem Controller

First zoom-in: subsystem domain

Il sottosistema presenta i componenti principali per l'esecuzione dell'applicativo.

OaFBatchManager rappresenta il "main component". Esso sfrutterà OaFQueue e l'adattatore di OrderConfirmedEvent per usare gli adattatori di Infrastructure: HandleConfirmation per la gestione delle conferme d'ordine, SendOrder per convertire ed inviare le batch, UpdateOaFState per apportare modifiche alle variabili di stato dei OaF nella banca dati del CRM.

A supporto, OaFBuffer fornirà un'astrazione per la gestione dei OaF, caricati esaminando NewOrdersEvent. In Fase di caricamento, OaFValidator fornirà un framework di supporto per il controllo dell'integrità dati a scopo di garantire compatibilità verso la configurazione stabilita col fornitore.

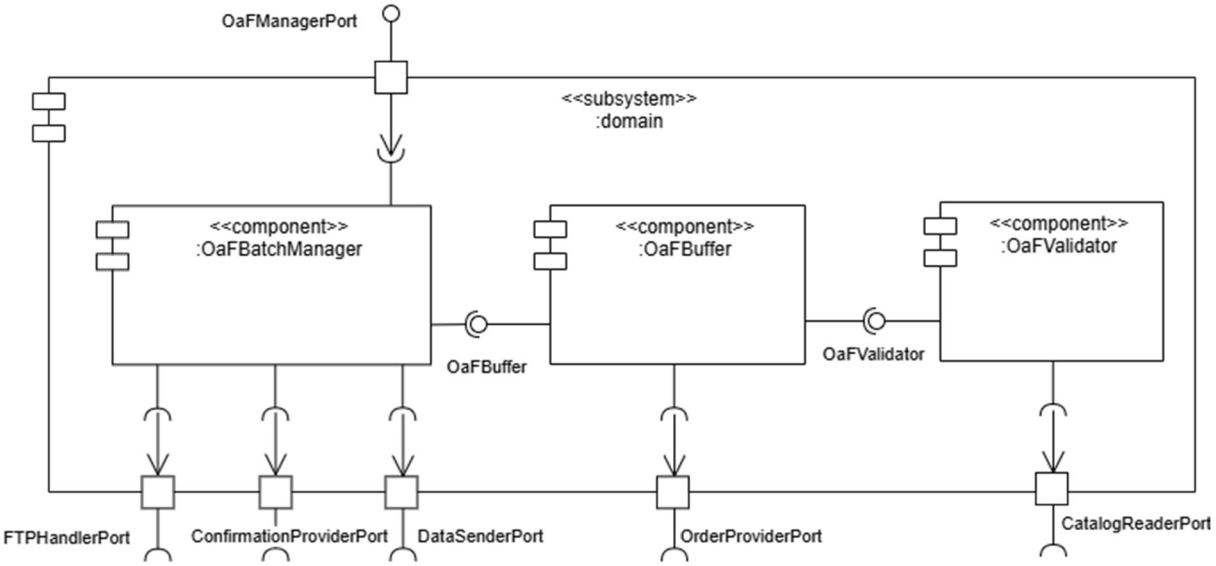


Figure 6 Component Diagram subsystem Domain

First zoom-in: subsystem infrastructure

Il sottosistema presenta gli attuatori che apporteranno modifiche ai sistemi collegati al middleware.

In fase di invio batch, richiesto da livello domain attraverso SendOrder, FTPWriter creerà la struttura XML OpenTrans compatibile con SAP GEALAN e contenente i dati già validati dal livello precedente (domain).

In fase di gestione delle conferme ordine, richiesto dal livello domain attraverso HandleConfirmation, FTPWriter trasferirà la conferma d'ordine nella cartella d'archiviazione, secondo le specifiche pattuite con GEALAN.

Quando verrà richiesto un update degli OaF, ODOOHandler provvederà a comunicare col database la richiesta di update delle variabili di stato.

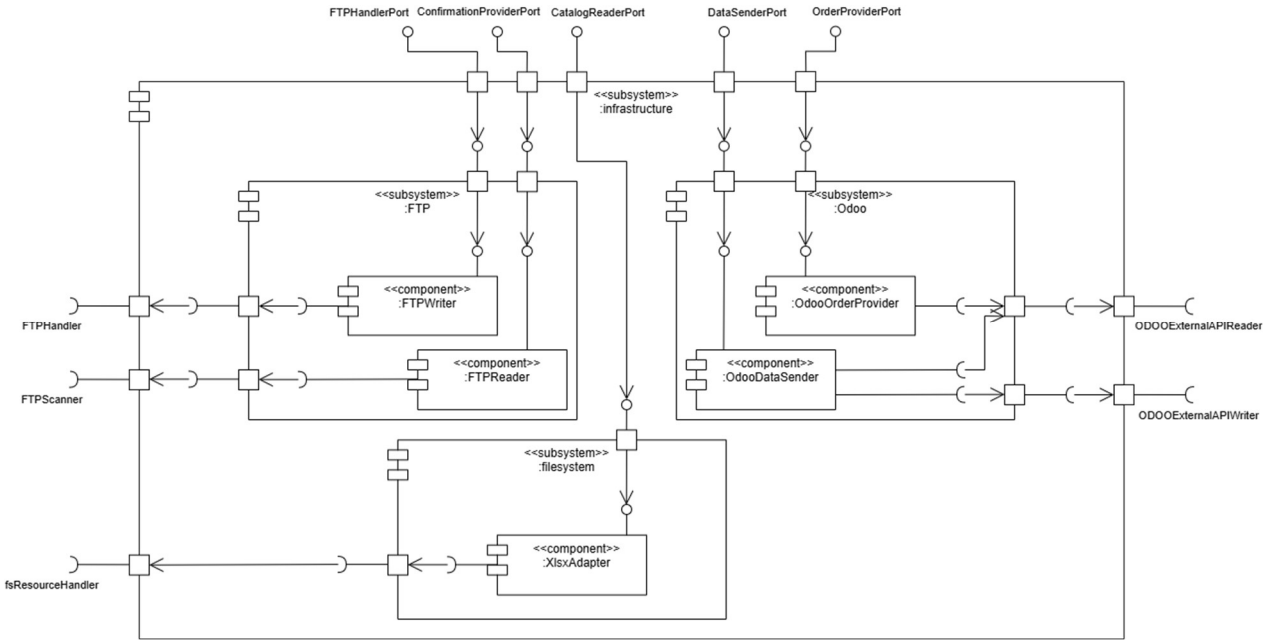


Figure 7 Component Diagram subsystem Infrastructure

Toolchain

Per garantire robustezza, manutenibilità e sicurezza del software, è stata adottata una toolchain integrata che copre l'intero ciclo di sviluppo, dalla codifica all'analisi statica, fino alla generazione automatica dei test e alla documentazione.

1. Sviluppo e Gestione delle Dipendenze

Il cuore applicativo è basato su tecnologie standard e moderne per garantire portabilità e supporto a lungo termine.

- **Linguaggio e Framework:** Java 17 (LTS) e Spring Boot 3.
- **Build Automation:** Apache Maven per la gestione delle dipendenze e il ciclo di vita della build.
- **IDE:** Eclipse IDE, esteso con plugin specifici per l'analisi statica (STAN, SpotBugs).

2. Testing e Generazione Automatica

È stata implementata una strategia di testing ibrida, combinando test scritti manualmente e test generati automaticamente, gestendo la retrocompatibilità tra versioni diverse di JUnit.

- **Framework di Testing:**
    - **JUnit 5 (Jupiter):** Per i test unitari di nuova scrittura.
    - **JUnit 4 & Vintage Engine:** Utilizzati per garantire la compatibilità ed esecuzione dei test generati automaticamente (che spesso producono codice JUnit 4).
- **Generazione Automatica dei Test:**
    - **EvoSuite:** Utilizzato per generare test unitari con l'obiettivo di massimizzare la code coverage tramite algoritmi genetici.
    - **Randoop:** Impiegato per generare test randomizzati finalizzati a trovare casi d'angolo e violazioni di contratti.

3. Analisi Statica, Sicurezza e Verifica Formale

Per garantire la qualità del codice (Clean Code) e la sicurezza, sono stati integrati molteplici livelli di analisi.

- **Qualità e Bug Detection:** PMD e SpotBugs (con plugin **SpotBugs Security** per la sicurezza) per l'analisi del bytecode e del sorgente.
- **Analisi Strutturale:** STAN IDE per il controllo dell'architettura e delle dipendenze tra package.
- **Verifica Formale: OpenJML** per la verifica della correttezza funzionale basata sulle specifiche JML (Java Modeling Language).

- **Sicurezza delle Dipendenze:** Snyk per l'individuazione di vulnerabilità nelle librerie di terze parti.
- **Aggregatore di Qualità: SonarQube** (o SonarCloud) per centralizzare tutte le metriche e visualizzare il debito tecnico.

4. CI/CD e Versionamento

Il processo di integrazione è stato automatizzato per garantire che ogni modifica al codice venga verificata immediatamente.

- **Version Control:** GitHub Repository.
- **Automation: GitHub Actions** per eseguire la pipeline di build, test e analisi statica ad ogni push.

5. Design e Documentazione

La documentazione tecnica e architetturale è stata prodotta utilizzando approcci "Docs-as-Code" e strumenti visuali.

- **Diagrammi UML: PlantUML** per generare diagrammi di classe e sequenza direttamente da descrizioni testuali, facilitando il versionamento dei grafici.
- **Schemi Architetturali:** Draw.io per diagrammi ad alto livello.
- **Redazione e Pubblicazione:** Microsoft Word per la stesura finale della tesi/documentazione e **GitHub Pages** per la pubblicazione della reportistica tecnica (es. site di Maven o report di coverage).

Roadmap

Il modello di sviluppo adottato nel progetto può essere descritto come un approccio ibrido tra ingegneria del software formale e metodologie Agile, con particolare riferimento ad alcune pratiche di eXtreme Programming (XP).

Il processo è stato concepito per conciliare il rigore della verifica formale con la flessibilità iterativa tipica dell'Agile, garantendo al contempo un'elevata affidabilità della logica di dominio e una continua evoluzione del codice.

In una prima fase, della durata di circa un mese, è stata posta particolare enfasi sulla progettazione UML e sulla definizione di un modello formale della business logic. Tale modello ha costituito la base per l'applicazione di strumenti di verifica statica e inferenza logica (in particolare tramite ASMeta e OpenJML), permettendo di validare la correttezza del comportamento atteso ancora prima dell'implementazione del codice.

Questa scelta metodologica rientra nella filosofia della correctness by design, tipica dei metodi formali e dell'approccio Model-Driven Engineering, e si discosta dal paradigma XP tradizionale, che privilegia la verifica dinamica tramite test unitari e Test-Driven Development (TDD).

Le fasi successive hanno seguito un'impostazione iterativa e incrementale: il lavoro è stato suddiviso in periodi temporali brevi (settimane), ognuno dedicato a specifici sotto-obiettivi (realizzazione di adattatori, pipeline DevSecOps, integrazione e collaudo). A partire dall'introduzione della pipeline DevSecOps, è stato adottato un refactoring continuo a cadenza settimanale, volto a migliorare la qualità architetturale e a mantenere la codebase in uno stato costantemente eseguibile, in linea con le pratiche di integrazione continua e con il principio XP del "working software as the primary measure of progress".

Un aspetto fondamentale del modello è stato il contatto costante con il cliente, soprattutto nelle fasi iniziali, per chiarire e affinare i requisiti, garantendo un allineamento continuo tra le specifiche formali e le esigenze operative. Inoltre, la documentazione tecnica è stata generata e aggiornata periodicamente, supportando la tracciabilità del processo e l'evoluzione architetturale del sistema.

In sintesi, il modello adottato può essere descritto come un approccio Formal-Agile, in cui la rigorosità della modellazione formale e della verifica logica si integra con la flessibilità e iteratività delle metodologie Agili, perseguendo un equilibrio tra affidabilità, correttezza e adattabilità del software.

Iterazione 1

La prima iterazione si sofferma sulla progettazione e realizzazione del nucleo dell'applicativo.

In questa fase si applicano tecniche sofisticate di modellazione per compiere un'analisi ad inferenza logica del comportamento ideale del software. Le specifiche progettuali sono verificate e validate attraverso una Abstract State Machine, mentre l'implementazione delle funzioni principali viene validata attraverso Java Modeling Language.

Realizzazione della Abstract State Machine

La Abstract State Machine è stata realizzata con ASMeta, framework per definire un Metamodello standard basato sui principi del *Model-Driven Engineering (MDE)*.

La simulazione riceve un ordine in formato XML, lo valida secondo regole specifiche (standard "OpenTrans" e regole specifiche del partner "GEALAN"), gestisce gli errori e simula lo scambio di messaggi (FTP) per la conferma degli ordini.

La Abstract State Machine in ogni momento può trovarsi in una delle tre fasi:

1. **NO_ORDER**: Il sistema è in attesa o gestisce operazioni tecniche FTP.
2. **ORDER**: Il sistema sta elaborando un ordine in arrivo.
3. **CONFIRMATION**: Il sistema sta gestendo la conferma di un ordine.

A seconda di queste, la main rule segue un path ben definito.

Il modello distingue due tipi di errori con gravità diversa: OPENTRANSERROR e CONTENTERROR. Il primo si verifica se mancano campi strutturali fondamentali (es. customer_number, delivery_location_number) o se il formato dei numeri non è corretto. Il secondo, meno grave, si verifica se i dati sono logicamente errati, ad esempio una data di consegna nel passato.

Fase NO_ORDER

Il sistema reagisce ai comandi operativi monitorati, ovvero le attività di ftpGEALAN:

- **WITHDRAW_ORDER**: Resetta i messaggi in ingresso (inboundMessages := 0).
- **CONFIRM_ORDER**: Prepara un messaggio di conferma (outboundMessages incrementa).
- **OFFLINE**: Resetta semplicemente i segnali.

Fase ORDER

Vengono eseguite in parallelo

- **Calcolo Errori (r_calculateErrors):** Controlla i campi XML. Se c'è un problema, alza il segnale OPENTRANS_ERROR o CONTENT_ERROR.Invio ordine.

- **Invio Ordine (r_sendOrder)**: Se NON c'è un errore critico (OpenTransError), l'ordine viene accettato e il contatore inboundMessages viene incrementato (+1). Se c'è un errore critico, l'ordine viene ignorato (il contatore non sale).

Fase CONFIRMATION

Viene eseguita la regola r_checkConfirmation:

- Se ci sono messaggi in uscita (outboundMessages > 0), il sistema simula l'archiviazione:
    1. Sposta i messaggi da "in uscita" a "archiviati".
    2. Alza il segnale ODA_CONFIRMATION (Conferma Ordine di Acquisto).
- Se non ci sono messaggi, resetta il segnale di conferma.

Analisi del Model Advisor

Ha un solo tipo di avviso (MP2, ovvero la “completezza delle regole condizionali”), mentre tutte le altre verifiche sono passate con successo.

In logica formale, una regola si dice "completa" se copre tutte le possibilità. Sul file MA.log è possibile notare quali regole hanno un branch per cui la macchina non avrebbe esito.

Verifica della Abstract State Machine

Le specifiche formali (LTL/CTLSPEC) Sono le "promesse" che il sistema deve mantenere e che strumenti come **AsmetaSMV** verificheranno. Per la verifica formale, il modello viene tradotto in NuSMV ed eseguito in quel formato. Sulla repository si può trovare TE.log contenente l’esito dell’esecuzione delle regole formalizzate, le quali sono passate tutte.

1. Accettazione dell'Ordine (Se tutto è OK)

Questa regola verifica che il sistema accetti correttamente un ordine valido.

LTLSPEC g(

(state=ORDER and checkOpenTransError=false and checkOpenTransError=false)

implies

(x(inboundMessages>0))

)

Significato: È sempre vero (g) che se siamo nello stato ORDER e **non** ci sono errori critici (OpenTransError), allora nello stato immediatamente successivo (x) il contatore dei messaggi in ingresso (inboundMessages) deve essere maggiore di 0.

2. Gestione Errore Critico (OpenTrans Error)

Queste regole assicurano che un ordine malformato venga bloccato e segnalato.

LTLSPEC g(

(state=ORDER and checkOpenTransError)

implies

(x(raisedSignal(OPENTRANS_ERROR)))

)

Significato: Se siamo in ORDER e c'è un errore critico, al passo successivo **deve** attivarsi il segnale OPENTRANS_ERROR.

CTLSPEC ag(

(state=ORDER and checkOpenTransError and inboundMessages=0)

implies

ax(raisedSignal(OPENTRANS_ERROR) and inboundMessages=0)

)

Significato: Rinforza la precedente. In tutti i futuri possibili (ag), se c'è un errore critico e la casella di posta è vuota, al passo successivo (ax) ci sarà l'errore E la casella **resterà vuota** (l'ordine non viene accettato).

3. Gestione Errore di Contenuto (Content Error)

Simile alla precedente, ma per errori meno gravi (sui dati, non sulla struttura).

LTLSPEC g(

(state=ORDER and checkContentError)

implies

(x(raisedSignal(CONTENT_ERROR)))

)

Significato: Se c'è un errore di contenuto, al passo successivo deve alzarsi il segnale CONTENT_ERROR.

CTLSPEC ag(

(state=ORDER and checkContentError and inboundMessages=0)

implies

ex((raisedSignal(CONTENT_ERROR)) and inboundMessages>=0)

)

Significato: Esiste almeno un percorso futuro (ex) in cui viene segnalato l'errore, ma i messaggi potrebbero essere accettati (infatti dice inboundMessages >= 0, permettendo che l'ordine passi nonostante l'errore).

4. Persistenza dei Messaggi in Ingresso

Questa regola definisce come vengono svuotati i messaggi ricevuti.

CTLSPEC inboundMessages>0 implies au(inboundMessages>0, ftpGEALAN=WITHDRAW_ORDER)

Significato: Se ci sono messaggi in ingresso (>0), la condizione di "avere messaggi" deve rimanere vera Finché (au = Always Until) non arriva il comando FTP WITHDRAW_ORDER (che li preleva).

5. Generazione della Conferma

Regola il comportamento quando il partner esterno chiede di confermare gli ordini.

LTLSPEC (state=NO_ORDER and ftpGEALAN=CONFIRM_ORDER) implies x(outboundMessages>0)

Significato: Se il sistema riceve il comando FTP CONFIRM_ORDER, nello stato successivo (x) deve generare dei messaggi in uscita (outboundMessages > 0).

6. Coerenza Storica e Archiviazione (Traceability)

Questo gruppo di 3 regole serve a garantire che non si perdano conferme e che lo storico sia coerente.

LTLSPEC (outboundMessages=0 and raisedSignal(ODA_CONFIRMATION)) implies y(state=CONFIRMATION and outboundMessages>0)

Significato: Se ora ho il segnale di conferma attivo ma 0 messaggi in uscita (significa che li ho appena spostati in archivio), allora **IERI** (y = yesterday) dovevo per forza essere nello stato CONFIRMATION e avere messaggi pronti da inviare.

LTLSPEC (archivedOutboundMessages>0 and raisedSignal(ODA_CONFIRMATION)) implies y(state=CONFIRMATION and outboundMessages>0)

Significato: Simile alla precedente, collega l'archivio pieno e il segnale di conferma allo stato precedente.

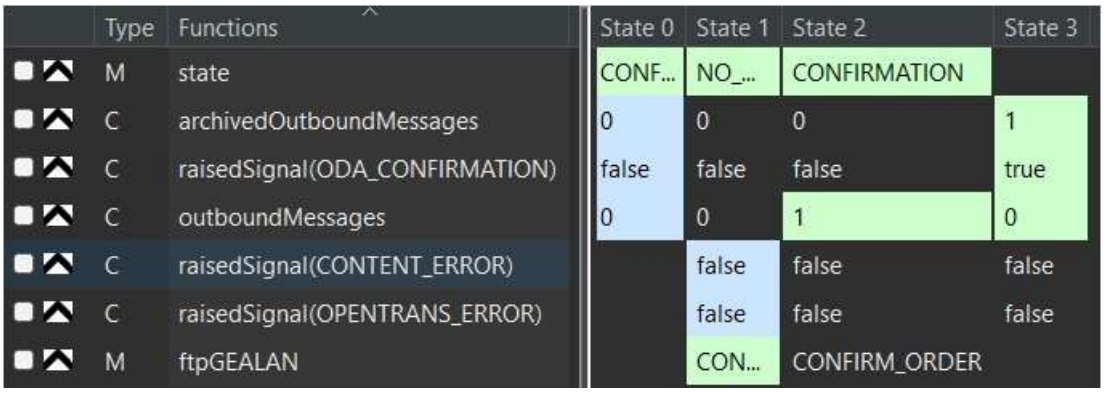
CTLSPEC archivedOutboundMessages>0 implies af(archivedOutboundMessages>0)

Significato: Una volta che un messaggio è archiviato (>0), rimarrà archiviato per sempre (o meglio, in tutti i futuri af sarà ancora vero che l'archivio non è vuoto). Non si cancellano i log.

Validazione della Abstract State Machine

La validazione è stata effettuata tramite l'animazione del modello, permettendo la definizione di scenari idonei al Model-Based Testing del codice implementato.

In particolare, sono stati realizzati tre scenari Avalla: *checkConfirmation.avalla*, *sendOrder.avalla*, *noOrder.avalla*.



	Type	Functions	State 0	State 1	State 2	State 3
	M	state	CONF...	NO_...	CONFIRMATION	
	C	archivedOutboundMessages	0	0	0	1
	C	raisedSignal(ODA_CONFIRMATION)	false	false	false	true
	C	outboundMessages	0	0	1	0
	C	raisedSignal(CONTENT_ERROR)		false	false	false
	C	raisedSignal(OPENTRANS_ERROR)		false	false	false
	M	ftpGEALAN		CON...	CONFIRM_ORDER	

Figure 8 scenario 'checkConfirmation.avalla'

	Type	Functions	State 0	State 1	State 2	State 3	State 4	State 5	State 6	State 7	State 8	State 9	State 10	State 11
	M	state	ORDER	ORD...	ORD...	ORD...	ORD...	ORD...	ORD...	ORD...	ORD...	ORD...	ORDER	
	M	hasRightValue(customer_number)	true	true	false	true	true	true	true	true	true	true	true	
	M	hasRightValue(delivery_location_number)	true	true	true	false	true	true	true	true	true	true	true	
	M	hasContent(delivery_date_content)	true	true	true	true	false	true	true	true	true	true	true	
	M	hasQuantity(quantity)	INTEG...	FLOA...	FLOA...	FLOA...	FLOA...	FLOA...	NAN	FLOA...	FLOA...	FLOA...	FLOAT...	
	M	hasRightValue(article_number)	true	true	true	true	true	true	true	false	true	true	true	
	M	hasRightValue(quantity_measure)	true	true	true	true	true	true	true	true	false	true	true	
	M	hasRightValue(delivery_date)	true	true	true	true	true	true	true	true	true	false	true	
	C	inboundMessages	0	1	2	2	2	2	2	2	3	4	5	6
	C	archivedOutboundMessages	0	0	0	0	0	0	0	0	0	0	0	0
	C	raisedSignal(ODA_CONFIRMATION)	false	false	false	false	false	false	false	false	false	false	false	false
	C	outboundMessages	0	0	0	0	0	0	0	0	0	0	0	0
	C	raisedSignal(CONTENT_ERROR)		false	false	false	false	true	false	false	true	true	true	false
	C	raisedSignal(OPENTRANS_ERROR)		false	false	true	true	true	true	true	false	false	false	false

Figura 9 scenario 'sendOrder.avalla'

Scenario CheckConfirmation.avalla

Questo scenario dimostra che l'archiviazione e la segnalazione avvengono *solo* se c'è effettivamente un messaggio in uscita in attesa. Verifica inoltre che i messaggi di conferma non vengano creati dal nulla, ma seguano un flusso preciso: vengono preparati via FTP e poi archiviati.

- **Step 1:** Imposti lo stato su CONFIRMATION senza aver preparato messaggi.
    - *Risultato:* Non succede nulla (tutto resta a 0). Questo conferma che il sistema non inventa conferme se non ce ne sono in coda.
- **Step 2:** Cambi stato in NO_ORDER e simuli l'arrivo di una richiesta di conferma (ftpGEALAN := CONFIRM_ORDER).
    - *Risultato:* La variabile outboundMessages sale a 1. Il messaggio è pronto per uscire, ma non è ancora archiviato.
- **Step 3:** Torni nello stato CONFIRMATION.
    - *Risultato:* outboundMessages torna a 0 (messaggio inviato), archivedOutboundMessages sale a 1 (messaggio archiviato), il segnale ODA_CONFIRMATION diventa true.

Scenario sendOrder.avalla

Questo scenario testa sistematicamente la differenza tra errori bloccanti (**OpenTrans**) e non bloccanti (**Content**), validando la logica complessa di Validazione XML (stato ORDER).

1. **Happy Path (Step 0-2):** Inserisce dati perfetti (INTEGER) e poi dati validi alternativi (FLOAT_WITH_DOT).
    - *Risultato:* inboundMessages sale a **1** poi a **2**. Nessun errore.
2. **Errori Bloccanti - OpenTrans (Step 2-7):** Inserisce deliberatamente errori strutturali uno alla volta: customer_number mancante, delivery_location errata, contenuto data mancante, quantità con virgola, quantità NAN.
    - *Risultato:* In tutti questi passi si alza OPENTRANS_ERROR. La cosa fondamentale è che **inboundMessages resta fermo a 2**. Il sistema sta rifiutando gli ordini. Qui si alzano *entrambi* gli errori (CONTENT e OPENTRANS). È corretto perché la mancanza di contenuto in un campo obbligatorio viola sia la regola di struttura che quella logica.
3. **Errori Non Bloccanti - Content (Step 7-10):** Inserisce errori logici ma non strutturali: article_number errato, unità di misura sbagliata, data nel passato.

- *Risultato:* Si alza il segnale CONTENT_ERROR. Tuttavia, nota che inboundMessages continua a salire (3, 4, 5). Questo dimostra che il sistema **accetta** l'ordine (perché strutturalmente valido) pur segnalando che il contenuto va controllato manualmente.

4. **Ritorno alla Normalità (Step 10-11):** Tutto corretto di nuovo.
    - *Risultato:* inboundMessages sale a **6**. Segnali spenti.

Scenario noOrder.avalla

Verifica correttamente che i comandi FTP agiscano sulle variabili giuste (Inbound vs Outbound) senza interferire tra loro. Questo scenario testa come il sistema reagisce ai comandi esterni quando non sta elaborando un file XML.

- **Step 1 (OFFLINE):** Simula il comando OFFLINE.
    - *Risultato:* Il sistema resetta i segnali ma non cambia i contatori. È lo stato di "riposo".
- **Step 2 (CONFIRM_ORDER):** Simula la richiesta di conferma.
    - *Risultato:* outboundMessages diventa **1**.
- **Step 3 (WITHDRAW_ORDER):** Simula il comando di prelievo/ritiro ordini.
    - *Risultato:* La variabile inboundMessages viene forzata a **0** (simula che GEALAN ha scaricato gli ordini che avevi accumulato). OutboundMessages resta a 1 perché il comando WITHDRAW tocca solo l'entrata (inbound), non l'uscita.

Iterazione 2

Progettazione del dominio

Seguendo i component diagram delineati a inizio progettazione, è stato organizzata e realizzata la prima bozza del package *domain*, costituente la business logic dell'applicativo. Tale parte costituisce il cuore dell'applicativo e riflette il comportamento del modello ASMeta.

- Ricezione e caricamento degli ordini in arrivo;
- Ricezione e caricamento delle conferme in arrivo;
- Analisi dell'ordine e validazione;
- Invio o segnalazione;

Per ordini e conferme vengono delineate delle strutture POJO, di alto livello di astrazione ma fedeli ai dati che dovrebbero contenere.

La business logic userà il pattern port-adapter per comunicare con l'esterno, rispettando i principi di Liskov.

Implementazione con OpenJML ed ESC

L'implementazione della business logic ha aderito rigorosamente ai paradigmi Model-Driven e Design-By-Contract, sfruttando OpenJML per elevare la robustezza strutturale del codice.

Un risultato distintivo di questa fase è stata la verifica integrale del modello di dominio tramite **Extended Static Checking (ESC)**; questa tecnica di analisi statica ha permesso di certificare matematicamente l'assenza di errori a priori, concentrando gli sforzi di validazione sul delicato sistema di analisi degli ordini XML e sulla logica core dell'applicativo, garantendo così una stabilità operativa superiore.

È stata introdotta la classe BoundedBuffer, un buffer per gli ordini a capacità limitata, governato da specifiche formali per le operazioni di push, pop e get, essenziale per una gestione sicura dei flussi di dati. L'ecosistema del codice è stato infine completato da librerie di utilità ottimizzate per la manipolazione delle stringhe e la gestione del formato data, assicurando la massima coerenza nel trattamento delle informazioni.

Class Diagram del dominio

I diagrammi di classe sono consultabili aggiornati sulla repository Github.

Package domain.ports.infrastructure

Contiene le porte che il *domain* userà per comunicare con *infrastructure*.

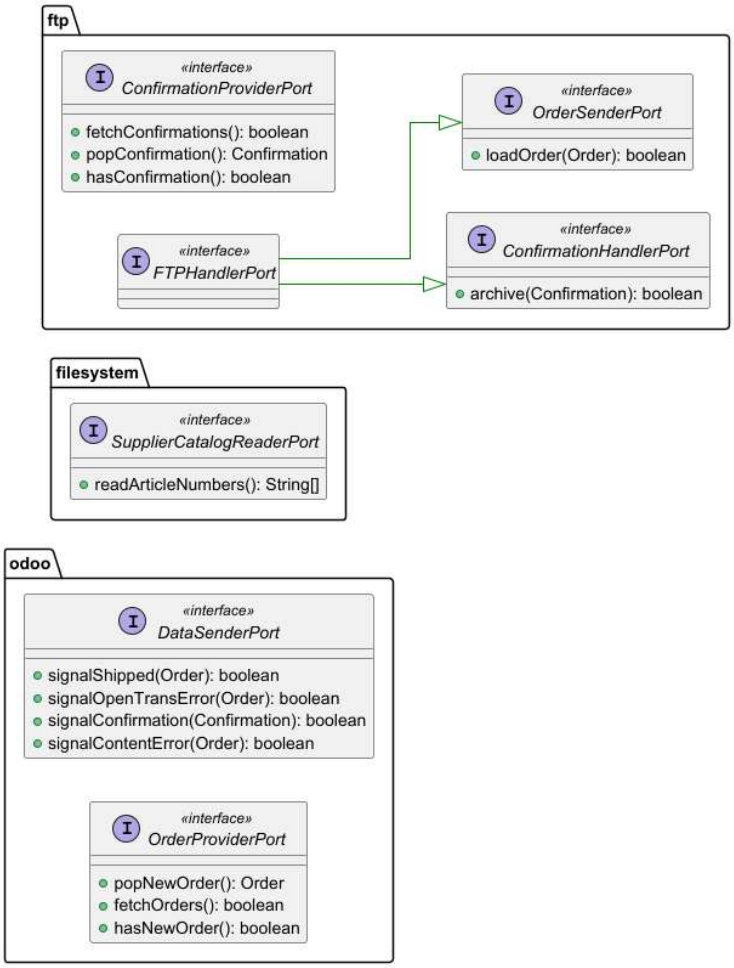


Figura 10 class diagram del package domain.ports.infrastructure

Package domain.ports.controller

Contiene le porte che il *domain* userà per comunicare con *controller*.

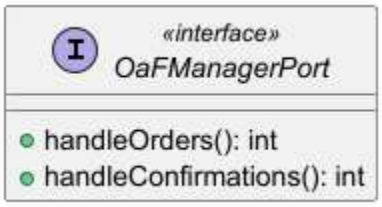


Figura 11 class diagram OaFManagerPort

Package domain.exceptions

Eccezioni specifiche per le attività dei servizi nel dominio.

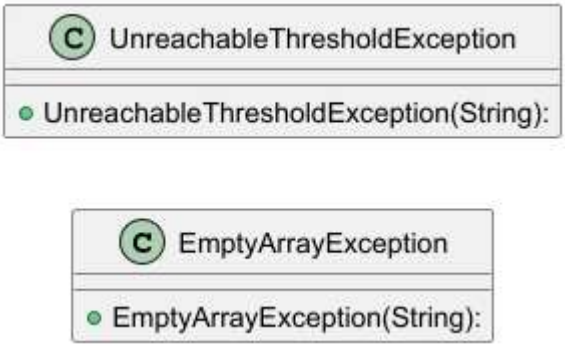


Figura 12 class diagram di UnreachableThresholdException e EmptyArrayException

Pacakge domain.utils

In domain.utils sono definite le strutture di supporto strumentali al funzionamento dei servizi principali.

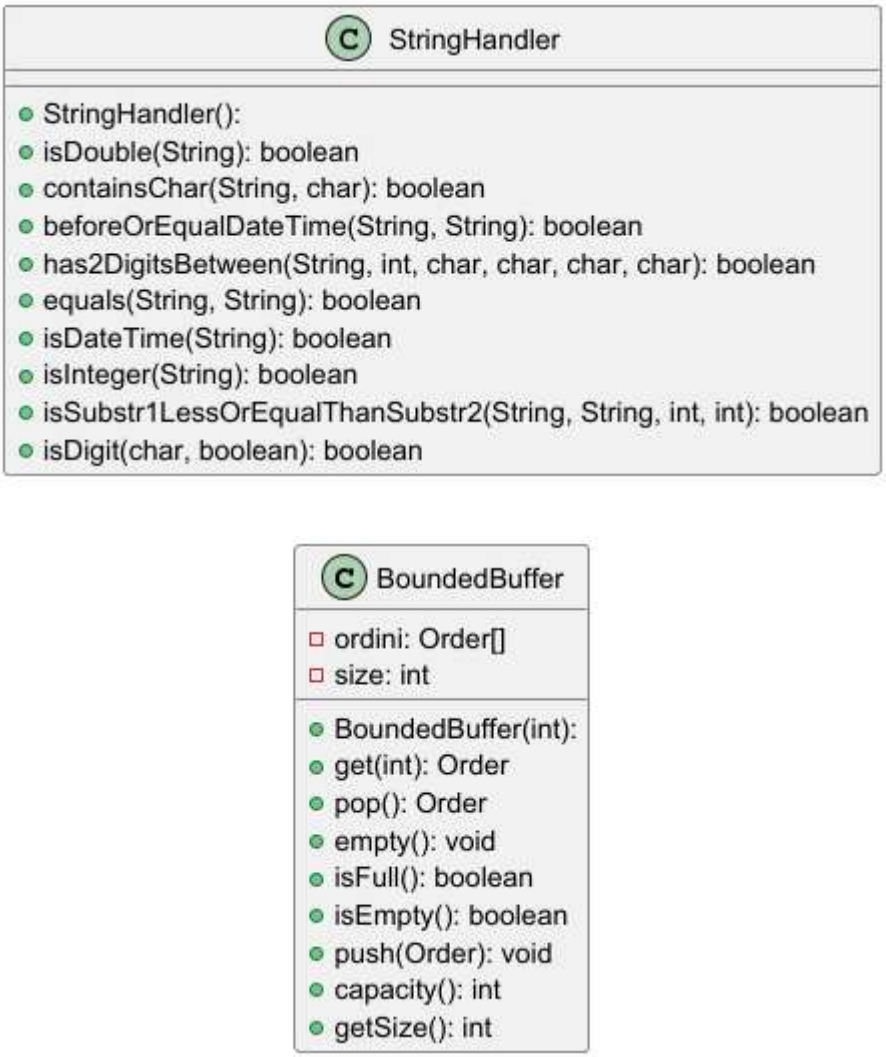


Figura 13 class diagram di domain.utils

Package domain.pojo.confirmation

Contiene le strutture POJO per rappresentare la conferma d'ordine.

Figura 14 class diagram di domain.pojo.confirmation

Package domain.pojo.order

Contiene le strutture POJO per rappresentare l'ordine.

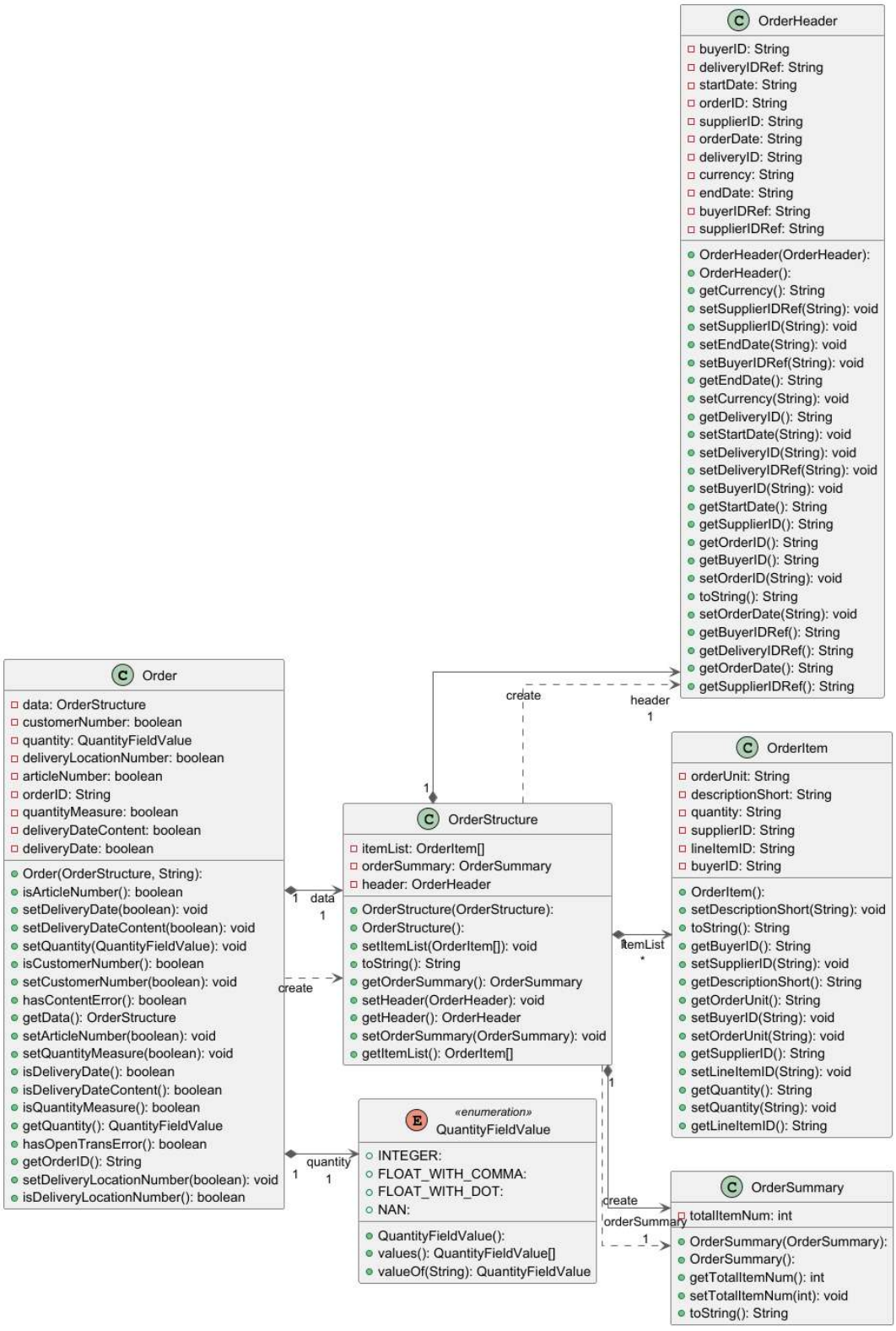


Figura 15 class diagram con le classi di domain.pojo.order

Package domain.service

Rappresenta i servizi principali della business logic.

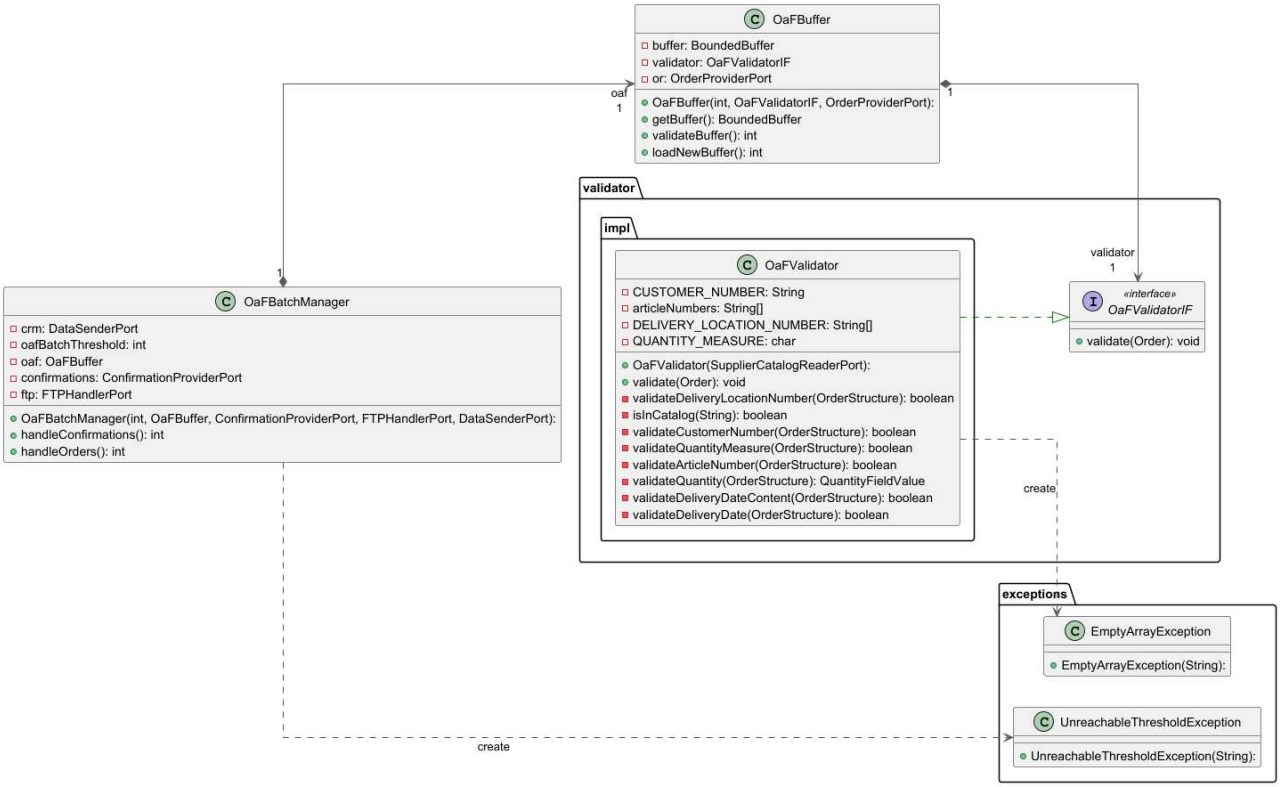


Figure 16 class diagram package domain.service

Iterazione 3

Durante la terza iterazione si è proceduto al completamento dello sviluppo, finalizzando l'implementazione delle restanti componenti del codice. Contestualmente è stata condotta un'approfondita fase di verifica funzionale che ha operato su due livelli distinti. Da un lato sono stati eseguiti test d'unità per validare la logica delle singole classi, dall'altro si è passati alla messa in opera dell'applicativo su un server di test per valutarne il comportamento complessivo in un ambiente integrato. Al fine di assicurare la conformità del progetto ai più elevati standard industriali, è stata istituita una pipeline di Continuous Integration mediante GitHub Actions. Questo flusso di lavoro automatizzato integra strumenti specifici per l'analisi statica e la sicurezza, quali Snyk e SonarCloud, garantendo un controllo continuo sulla qualità del software rilasciato.

In questo contesto, Snyk si configura come una piattaforma essenziale per la sicurezza delle applicazioni, specializzata nell'individuazione di vulnerabilità all'interno del codice proprietario, delle dipendenze open source e delle immagini container. L'integrazione di Snyk nel flusso di lavoro porta il vantaggio tangibile di intercettare le minacce di sicurezza nelle prime fasi dello sviluppo, permettendo al team di correggere preventivamente l'uso di librerie obsolete o compromesse e riducendo drasticamente la superficie di attacco del software finale.

SonarCloud è invece un servizio basato su cloud dedicato all'analisi statica del codice e alla misurazione della qualità del software. Il suo ruolo è quello di scansionare il codice sorgente per rilevare bug, vulnerabilità e porzioni di codice di difficile manutenzione, note come code smells. Il principale beneficio offerto da SonarCloud risiede nella sua capacità di monitorare il debito tecnico nel tempo, fornendo metriche chiare su affidabilità e manutenibilità. Ciò assicura che il codice rimanga pulito, leggibile e conforme alle best practices internazionali, facilitando le future evoluzioni del progetto.

Pipeline CI

La pipeline CI fa uso dei seguenti strumenti e concetti:

- **Maven**: È lo strumento di riferimento per l'automazione della build e la gestione delle dipendenze nei progetti Java. Si occupa di compilare il codice, eseguire i test, gestire le librerie necessarie e creare il pacchetto finale (come JAR o WAR) in modo standardizzato.
- **SpotBugs con SpotBugs Security**: SpotBugs è un tool di analisi statica che esamina il bytecode Java (il codice compilato) alla ricerca di pattern di bug comuni (errori logici, performance, cattive pratiche). SpotBugs Security (precedentemente FindSecBugs) è un plugin specifico che estende queste capacità per individuare vulnerabilità di sicurezza critiche, come quelle della lista OWASP (es. SQL Injection, XSS).

- **SonarCloud con Quality Gate**: SonarCloud è un servizio basato su cloud che analizza il codice sorgente per misurarne la qualità, rilevando bug, vulnerabilità e debito tecnico ("code smells"). Il Quality Gate è la "condizione di passaggio": un insieme di soglie (es. "copertura test > 80%", "zero nuove vulnerabilità") che il codice deve superare obbligatoriamente per essere accettato. Se il Quality Gate fallisce, la pipeline si blocca impedendo il merge.
- **Snyk**: È una piattaforma di sicurezza focalizzata principalmente sulla Software Composition Analysis (SCA). Scansiona il progetto per identificare vulnerabilità note all'interno delle dipendenze open source (librerie esterne) che stai utilizzando, suggerendo spesso la versione "sicura" a cui aggiornare. Può analizzare anche container e codice proprietario.
- **Dependabot**: È un bot nativo di GitHub che monitora i file di configurazione delle dipendenze (come il pom.xml di Maven). Rileva automaticamente se stai usando versioni obsolete o insicure di una libreria e apre autonomamente una Pull Request per aggiornarle alla versione più recente o sicura.
- **CodeQL**: È il motore di analisi semantica sviluppato da GitHub. Tratta il codice come se fosse un dato, permettendo di eseguire "query" per trovare pattern complessi di vulnerabilità. È molto potente perché riesce a tracciare il flusso dei dati attraverso l'applicazione, individuando problemi che una semplice ricerca testuale non vedrebbe.
- **GitHub Advanced Security (GHAS)**: È una suite di funzionalità di sicurezza avanzata integrata in GitHub (solitamente a pagamento per le aziende). Include strumenti come CodeQL per la scansione del codice, il Secret Scanning (per trovare password o chiavi API lasciate per sbaglio nel codice) e la Dependency Review, offrendo una protezione completa direttamente nel flusso di lavoro della repository.

La Workspace locale genera, attraverso il normale goal di maven come '*mvn build',* dei report analizzabili via browser (se HTML) o altri strumenti come BaseX (se XML). Viene inoltre generato un file SARIF, *spotbugs-report.sarif*, utile per l'integrazione dati in Github Advanced Security.

Il lavoro viene salvato prima di tutto su un branch separato dal main, dedicato al codice. Dopo un *git push* dal workspace locale verso il branch, viene effettuata una analisi dei test d'unità con Maven per garantire che il codice mandato sia funzionante.

Appena viene aperta una *pull request* dal branch verso il main, viene fatta un'altra analisi con Maven e, se termina correttamente, vengono avviati SonarCloud e CodeQL per realizzare dei controlli di analisi statica e di sicurezza del codice. In particolare, SonarCloud verificherà se la qualità del codice incontra i requisiti minimi dettati dal profilo impostato dallo sviluppatore sulla piattaforma SonarCloud.

Parallelamente, Snyk analizza il manifest di Maven per segnalare eventuali problemi legati alle dipendenze di progetto, nel caso fossero state aggiunte delle liberie vulnerabili a precisi tipi di attacco. Tramite la sua piattaforma, Snyk illustrerà al dettaglio quali vulnerabilità sono state aggiunte, il loro grado di severity, le possibili mitigazioni e cosa effettivamente comportano, consentendo allo sviluppatore di scegliere la sua strategia.

Appena la pull requrest verrà accettata e unita al main, la pipeline si concentrerà sulla raccolta dati per arricchire le varie piattaforme di nuove informazioni, aggregando Issues e registrando i problemi del codice.

- Snyk registra le dipendenze vulnerabili;
- SonatCloud registra i nuovi issues e le loro posizioni. Se collegato a Eclipse, permette allo sviluppatore di selezionare le issues e raggiungerle sulla IDE “con un click”;
- CodeQL e PMD analizzano il codice e generano dei dati SARIF che, uniti al SARIF di SpotBugs caricato precedentemente, arricchiranno la piattaforma GHAS di nuove segnalazioni.

Periodicamente, Snyk continuerà in maniera indipendente ad effettuare uno scan del codice e proporre tramite pull request dei fix alle dipendenze del codice.

Allo stesso modo, Dependabot controlla periodicamente se le librerie sono aggiornate all’ultima versione, proponendo pull request per sistemare.

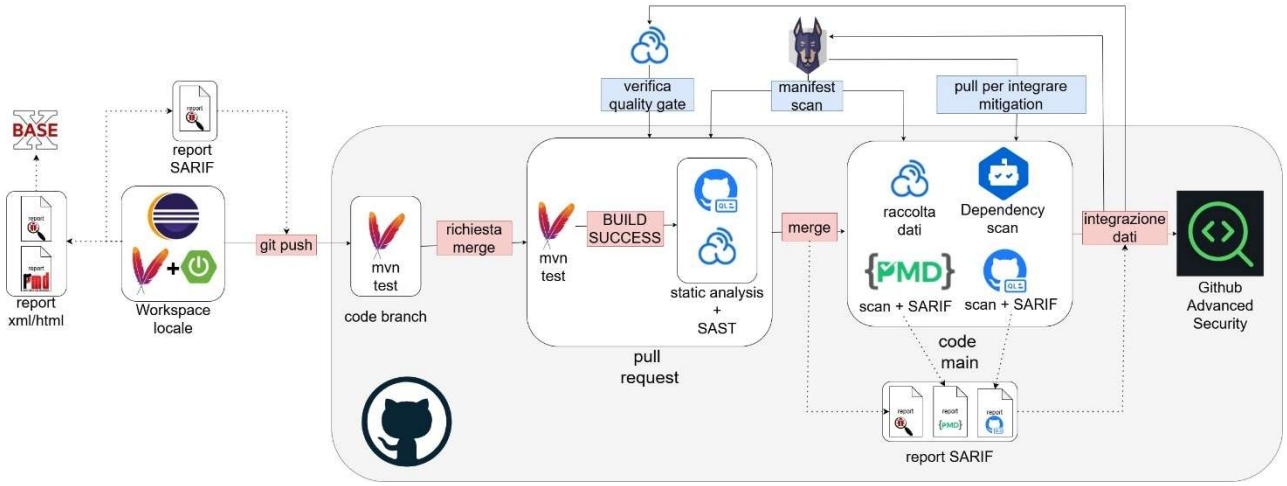


Figura 17 pipeline CI

Class Diagram adattatori

Sulla base delle esigenze della architettura e del materiale necessario per soddisfarle sono stati definiti, progettati ed implementati gli adattatori di servizio alle porte del dominio. Tali adattatori costituiscono l'implementazione "a basso livello" delle funzionalità richieste dal dominio per funzionare. Tali funzionalità sono strettamente legate alla tecnologia specifica con cui si va a trattare.

- Per la funzione di **caricamento ordini** e segnalazione verso il CRM, gli adattatori Odoo dovranno gestire il protocollo XML-RPC, eseguire le richieste specifiche verso il servizio Odoo ed elaborare i dati ricevuti di modo da estrarre le informazioni rilevanti per il dominio;
- Per la **validazione delle strutture dati** d'ordine ricevute, ci sarà bisogno di un adattatore che estragga da una sorgente i dati di riferimento per la validazione. Nel caso specifico, è stato disposto un file *.xlsx* contenente più di 6000 dati di riferimento ai prodotti a catalogo fornitore;
- Per la **scrittura e lettura verso FTP**, è necessario implementare le funzionalità di generazione delle strutture dati XML partendo dai POJO e la lettura delle strutture dati XML per convertirle in POJO. Sarà poi necessario interagire col protocollo FTP per mandare al cliente le strutture dati.

In ultimo, il dominio necessita di essere controllato per poter azionare le procedure di business. Tale controllo potrebbe essere automatizzato oppure manuale, via operatore.

Class Diagram Odoo

Ci sono due gruppi di processi fondamentali nel contatto con Odoo

- Richiesta dei dati necessari, interpretazione dei dati ricevuti, creazione delle strutture POJO per il dominio;
- Invio di segnali contenenti informazioni rilevanti per le attività del CRM.

Il primo gruppo di processi si concretizza in *OdooOrderProvider.java*, mentre il secondo con *OdooDataSender.java*. Entrambi comunicano con lo stesso server e sullo stesso protocollo. Pertanto, *OdooApiConfig.java* incorpora e gestisce i dati necessari alla connessione.

Figure 18 class diagram degli adattatori Odoo

Ricezione ed elaborazione dati a OdooDataSender

Il problema principale nella ricezione dati via le Odoo External API, oltre allo studio dei model e delle API esposte da Odoo per comprendere le richieste da fare, è l'elaborazione delle informazioni ricevute.

Il problema della corretta formulazione delle query si è risolto con uno studio approfondito della documentazione tecnica di Odoo External API, seguito da uno studio delle "successioni di join" tra i model per estrarre l'informazione necessaria.

Tale informazione, però, arriva "grezza" al middleware e necessita di una *evaluation*.

Si è proceduto quindi con la creazione di una serie di strutture di servizio, dette "DTO", che automatizzano il processo di creazione delle entità da rappresentanti i risultati delle richieste

fatte al server. I loro attributi sono degli Optional il cui contenuto è frutto di un'inferenza dei mapper. Se i mapper non colgono una formattazione corretta del dato, attraverso ad esempio una valutazione degli *Object* grezzi via *instanceof*, il valore di tali attributi sarà *Optional.empty()*.

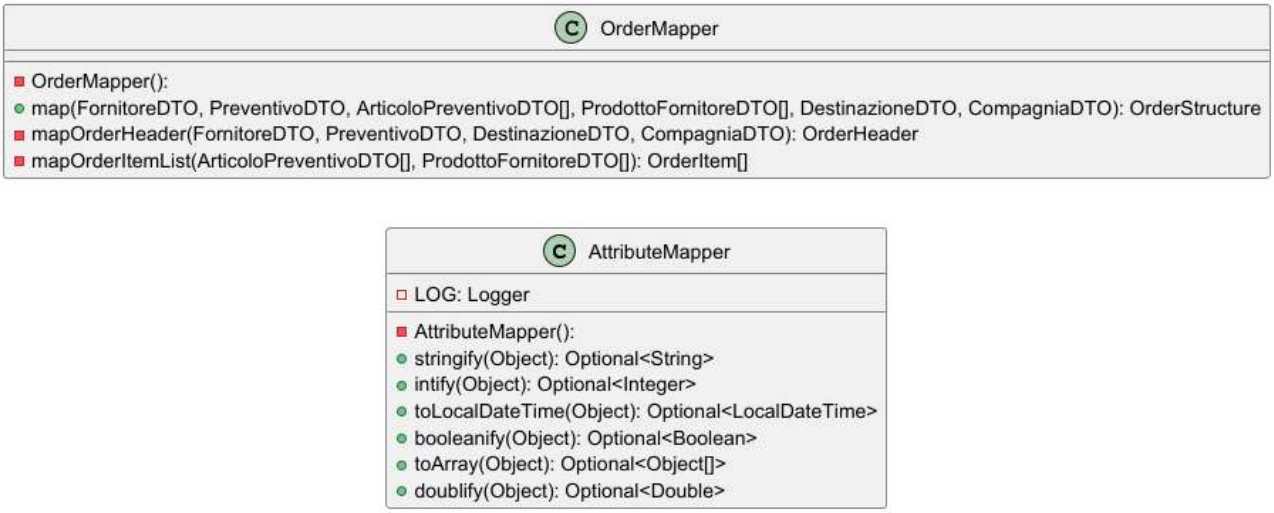


Figure 19 class diagram dei mapper in infrastructure.odoo.mapper

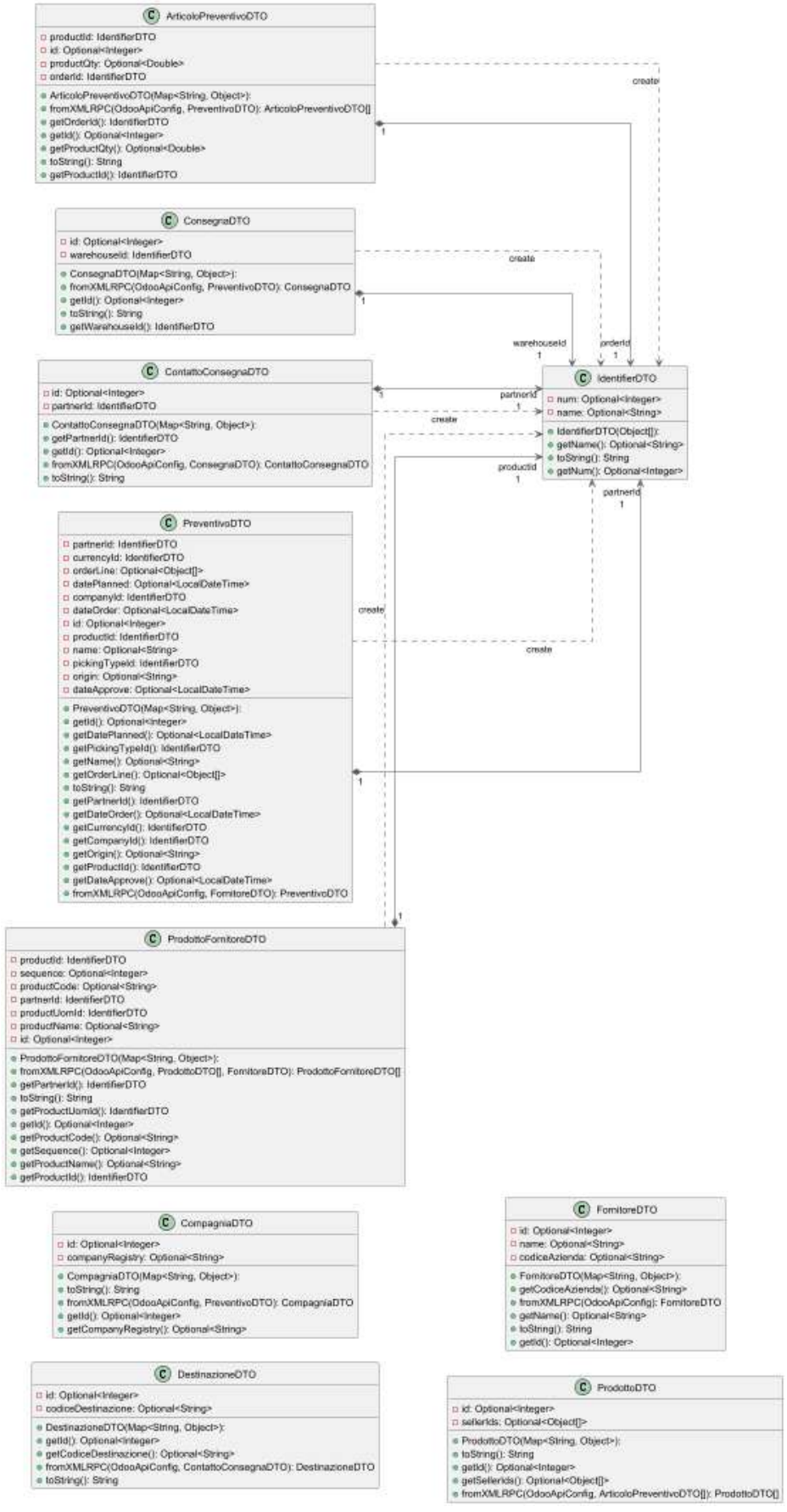


Figure 20 class diagram in infrastructure.odoo.dto

Class Diagram Xlsx

La classe OaFValidator a livello domain necessita delle informazioni del MasterData_IT.xlsx con catalogo fornitore. In particolare, viene richiesta solo l'estrazione della colonna contenente gli identificativi degli articoli. L'oggetto *XlsxAdapter.java* provvede a questo compito.

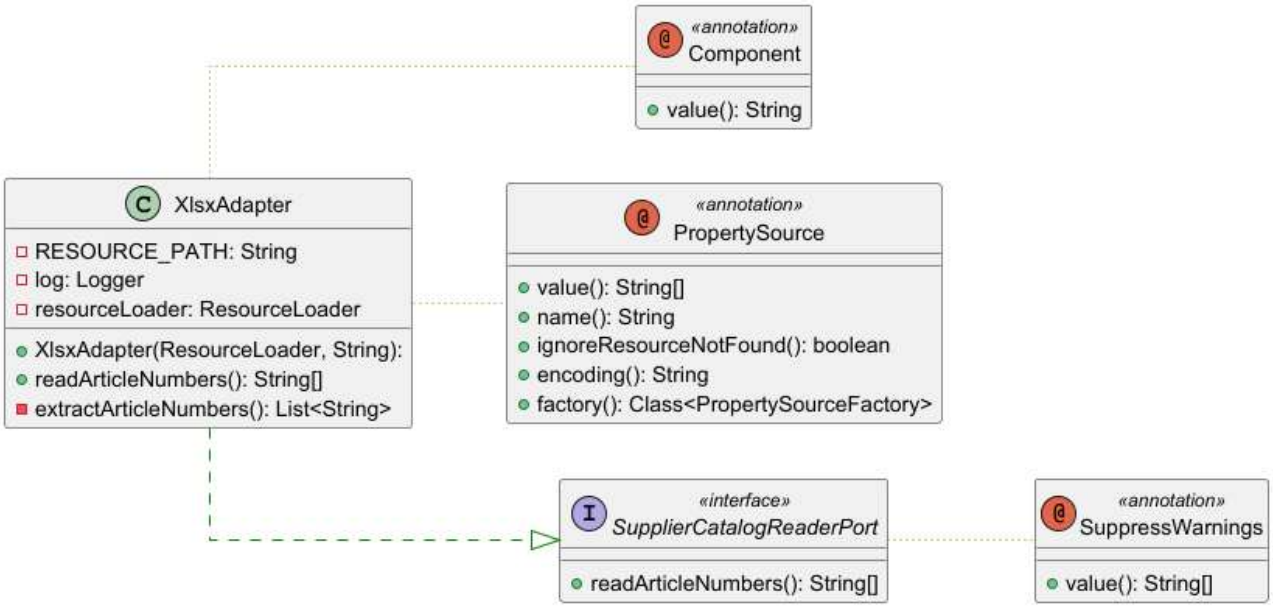


Figure 21 class diagram di infrastructure.filesystem

Class Diagram FTP

Sul filesystem del FTP vanno realizzate le operazioni di

- Conversione dei POJO nei DTO rappersentativi, serializzazione struttura dati ordine, scrittura sul canale INBOUND;
- Scansione del canale OUTBOUND, deserializzazione struttura dati conferma, conversion dei DTO in POJO, spostamento della struttura dati in ARCHIVE.

Il primo compito viene svolto da *FTPwriter.java* mentre il secondo da *FTPReader.java*.

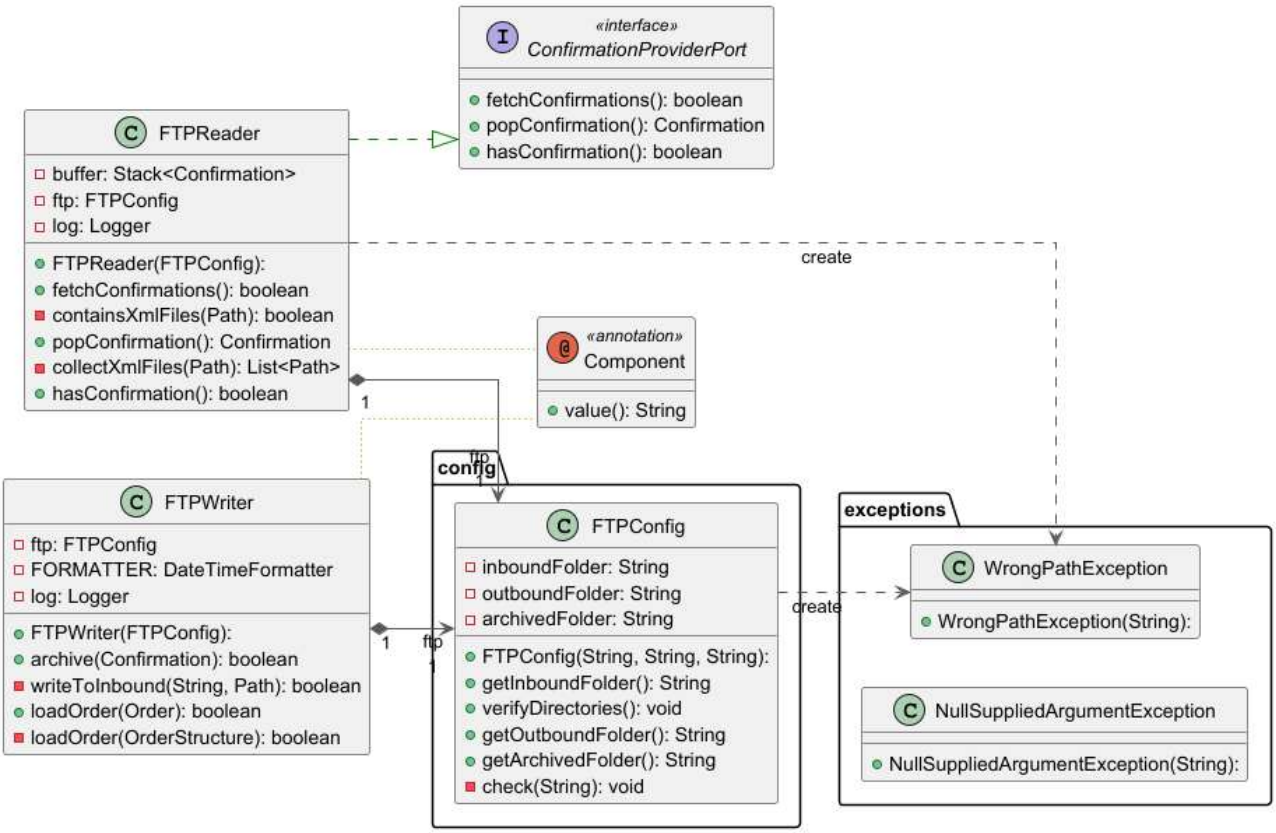


Figure 22 class diagram degli adattatori FTP, con configurazioni ed eccezioni

Figure 23 class diagram di infrastructure.ftp.dto.order

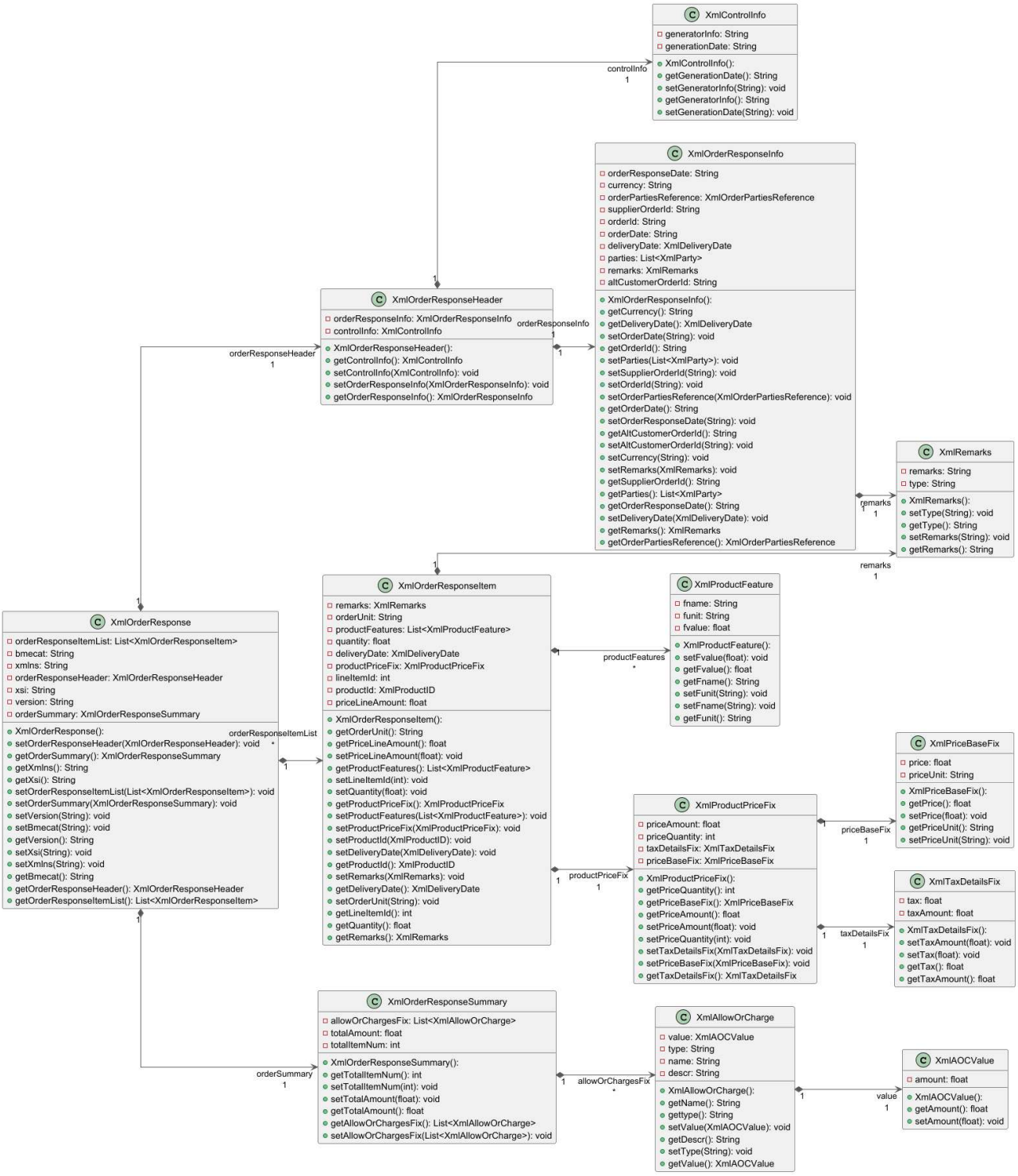


Figure 24 class diagram di infrastructure.dto.confirmation

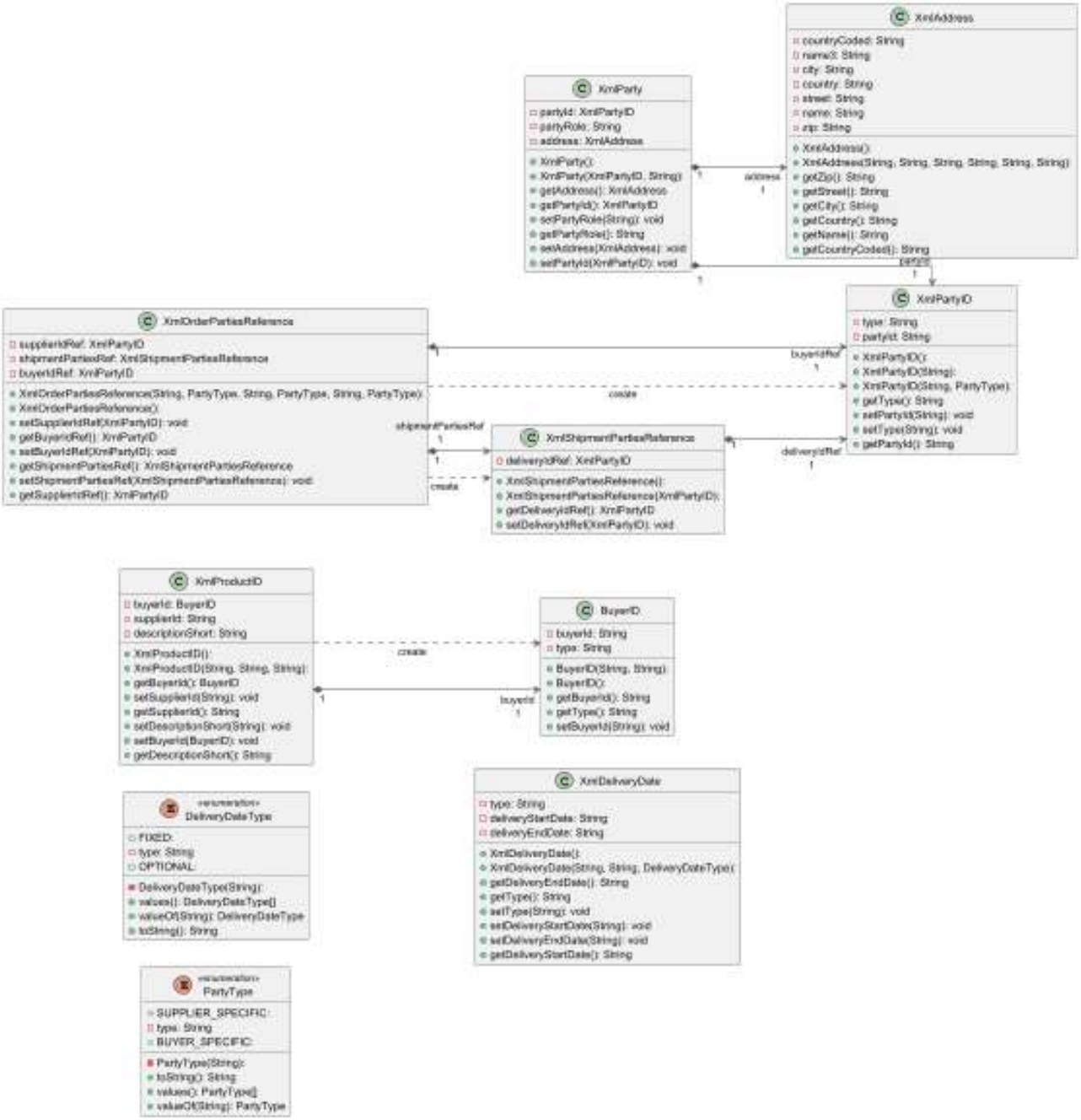


Figure 25 class diagram di infrastructure.ftp.dto.commons

Class Diagram REST controller

Il programma espone delle API per il controllo del dominio. Siccome si è scelto il controllo diretto tramite GUI per l'operatore, sono state disposte delle API apposite per servire la UI di controllo ed effettuare controllo sullo stato di salute del server.

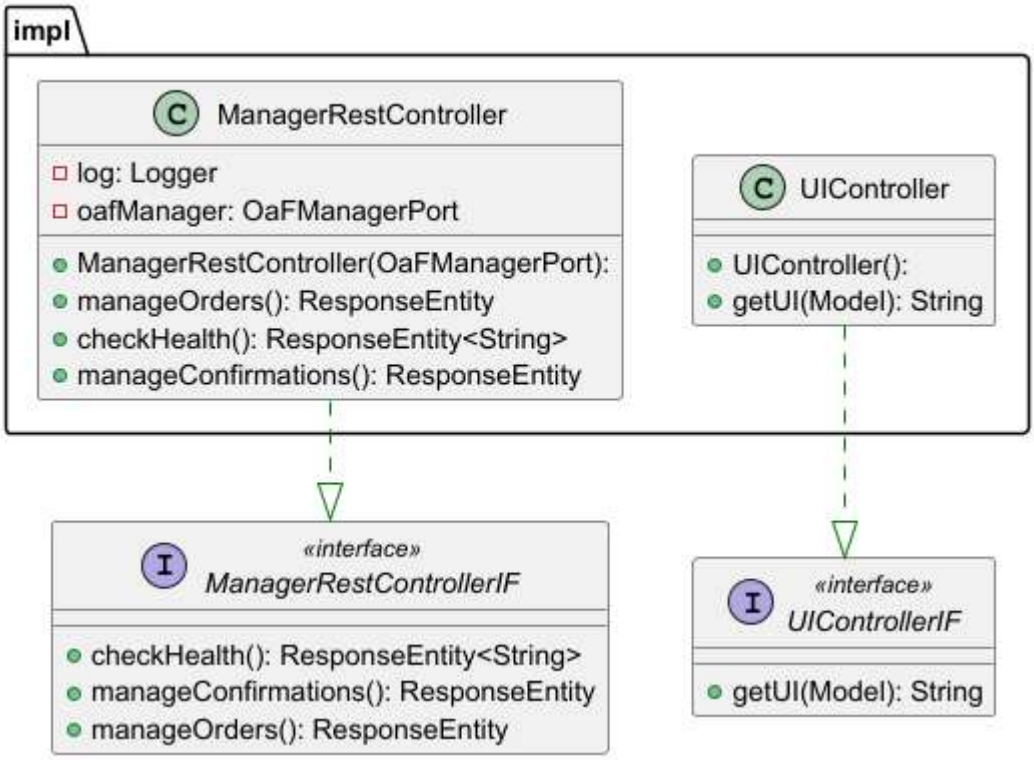


Figure 26 class diagram del package controller.http

Realizzazione adattatori

Uno schema intuitivo della mappatura per ottenere l'ordine a fornitore è visualizzabile qui. Da destra a sinistra, si passa dai model di Odoo in rosso ai DTO del livello infrastructure lato Odoo, per poi ottenere la struttura dati conforme a XML-OpenTrans dai DTO del livello infrastructure lato FTP.

Realizzazione adattatori Odoo

A inizio realizzazione, le classi di questo package si presentavano come delle "fat", specie per quanto riguarda il codice per il contatto col server. Nel corso del tempo, numerosi refactoring hanno scomposto quanto possibile le funzionalità in classi distinte, portando al raffinamento della struttura delle classi come visto sopra. Rimangono ancora comunque dei problemi di "classi Tangled" in questo package.

L'adattatore OdooOrderProvider crea sequenzialmente le strutture DTO di interesse, di modo da verificare, valutare ed usare i dati necessari sia all'estrazione di altre strutture dati che alla realizzazione del POJO per il dominio. Una rappresentazione della successione con la quale vengono effettuate le chiamate è rappresentato sulla repository attraverso il seguente activity diagram. OdooOrderProvider chiama il static factory method "fromXMLRPC" dell'oggetto DTO

da estrarre, passandogli l'oggetto di configurazione necessario ad effettuare le chiamate. L'adattatore risulta così molto semplice da comprendere e "spoglio" di codice di basso livello. A loro volta, le chiamate XMLRPC sono astratte attraverso l'oggetto OdooApiConfig, rendendole più semplici da gestire anche all'interno dei metodi chiamanti.

Grazie ad OdooApiConfig, anche l'adattatore OdooDataSender vede una semplificazione delle sue operazioni, che a basso livello seguono le stesse chiamate di OdooDataSender. Questo adattatore, in particolare, provvede alla segnalazione dello stato dell'ordine (SHIPPED, CONFIRMED, OPENTRANSERROR, CONTENTERROR).

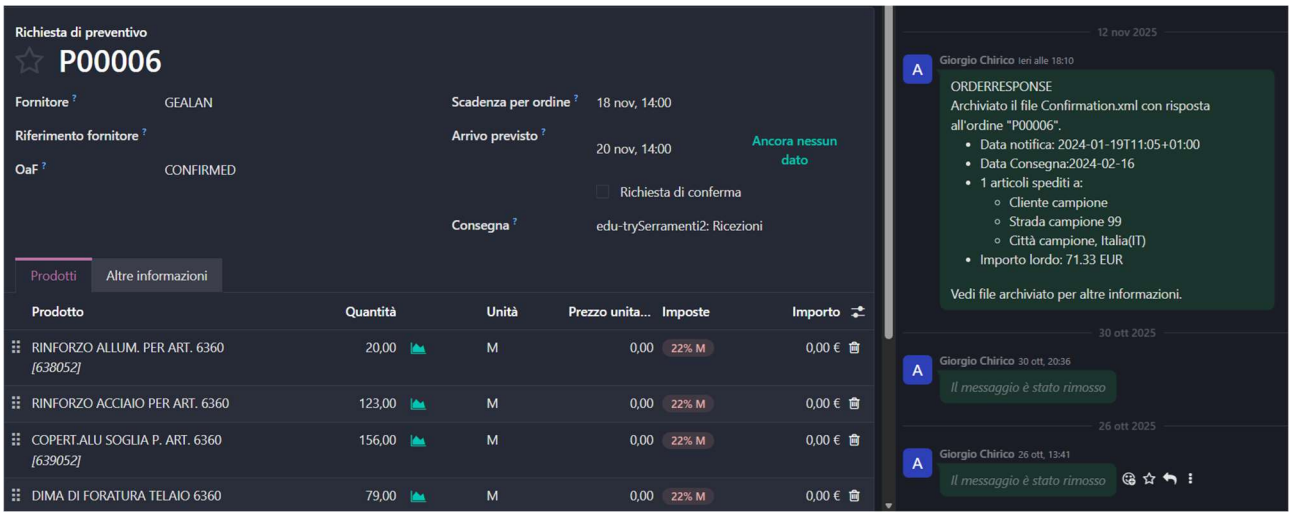


Figure 27 messaggio sul workflow e stato CONFIRMED inviato da OdooDataSender.java

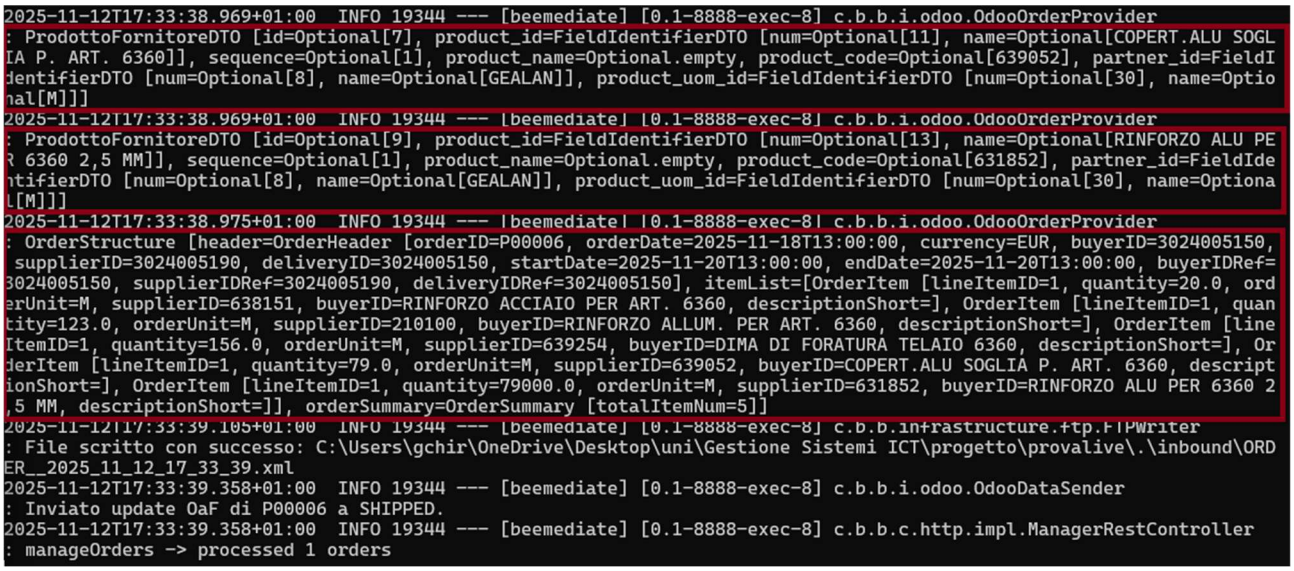


Figure 28 log INFO dei toString dei DTO generati e del POJO OrderStructure

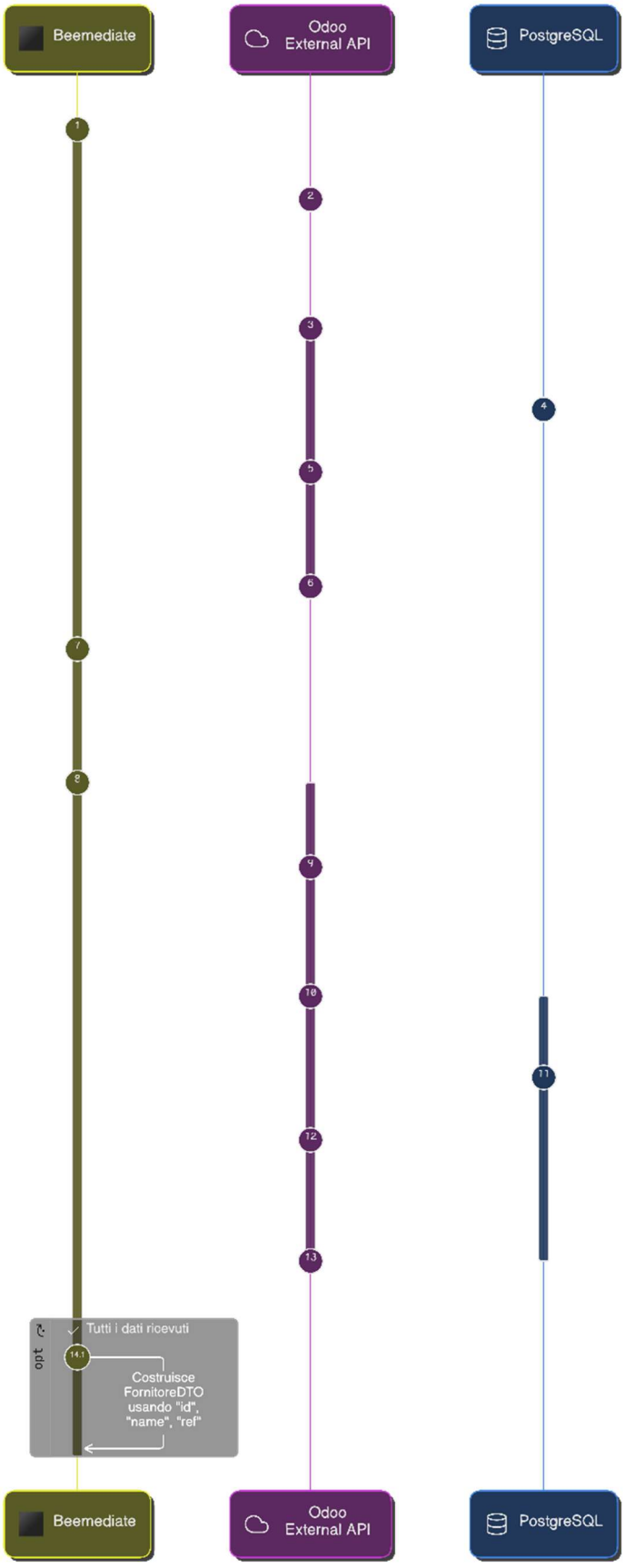


Figure 29 sequence diagram con esempio di procedura di estrazione dati

Realizzazione adattatore Xlsx

Trasforma la colonna degli identificativi articolo a catalogo fornitore in un semplice array di stringhe (String[]) contenente i codici articolo, pronto per essere elaborato dalla business logic.

1. **Carica la Risorsa:** Recupera il file Excel (es. MasterData_IT.xlsx) dal percorso specificato nel file di configurazione (datasource.path).
2. **Estrazione Dati:** Apre il file e legge esclusivamente la **prima colonna** del primo foglio di lavoro.
3. **Logica di Lettura:** Salta la prima riga (considerata come intestazione/header), converte il contenuto delle celle in testo (Stringhe), colleziona tutti i valori trovati in una lista di "Numeri Articolo".
4. **Validazione Rigida:** Include un controllo di integrità forte. Se durante la scansione delle righe trova una cella vuota o nulla, **interrompe l'intero processo** lanciando un errore (IllegalArgumentException), invece di saltarla.

Realizzazione adattatori FTP

Si è scelto di delegare la gestione low-level del protococollo FTP(S) a FileZilla. L'operatore utilizzerà quindi Beemediate per gli stessi obiettivi iniziali, ma ora focalizzati e limitati alla gestione su filesystem, mentre userà FileZilla per gestire la spedizione sicura via FTP(S) sul web.

Per tale ragione, sia FTPWriter che FTPreader lavorano esclusivamente a livello di percorsi (Path) locali, assumendo che le cartelle configurate siano quelle utilizzate per lo scambio dati con i partner commerciali.

L'implementazione di FTPWriter è così riassumibile nella seguente procedura:

1. **Esporta Ordini (Scrittura):**

- Riceve un oggetto di dominio Order.
- o converte (serializza) in formato **XML-OpenTrans** tramite un DataMapper.
- Crea un file fisico nella cartella *Inbound* (ingresso del sistema esterno) con un nome univoco basato su un timestamp (ORDER__yyyy_MM_dd...xml).

2. Archivia Conferme (Spostamento):

- Riceve una Confirmation già elaborata.
- Sposta fisicamente il file associato dalla cartella *Outbound* (uscita) alla cartella *Archived* (storico).
- Utilizza uno spostamento atomico (ATOMIC_MOVE) per garantire l'integrità dell'operazione.

L’implementazione di FTPReader è così riassumibile nella seguente procedura:

1. **Fase di Caricamento (fetchConfirmations):**

- **Scansione:** Esamina la cartella *Outbound* configurata alla ricerca di file .xml.
- **Reset:** Svuota il buffer interno esistente (buffer.clear()).
- **Deserializzazione:** Legge il contenuto dei file e lo trasforma da stringa XML a oggetti di dominio Confirmation (tramite DataMapper).
- **Popolamento:** Riempie il buffer con gli oggetti appena creati.

2. Fase di Consumo (popConfirmation):

- Agisce come un serbatoio di dati per il dominio.
- Fornisce le conferme una alla volta estraendole dal buffer (seguendo una logica dichiarata come LIFO - Last In First Out).

Anche in questa parte viene fatto uso di un mapper per traduzione (statica e finale), chiamato *DataMapper.java*. Il suo compito è isolare la logica di conversione tra il modello dati interno (Domain POJO) e il formato di interscambio esterno (XML, specificamente lo standard OpenTrans).

1. Configurazione del Parser:

- Inizializza un motore di parsing XML (basato su *Jackson* e *Woodstox*).
- **Nota Importante:** Disabilita intenzionalmente la gestione dei **Namespace XML** (IS_NAMESPACE_AWARE = false). Questo approccio "pragmatico" permette di leggere i file XML ignorando la complessità degli schemi XSD e dei prefissi dei tag, rendendo l'integrazione più flessibile ma meno rigorosa.

2.Flusso in Uscita (Serializzazione Ordini):

- Traduce l'oggetto di dominio OrderStructure in una gerarchia di oggetti DTO intermedi (XmlOrder, XmlItem...).
- Converte questi DTO in una stringa XML grezza, pronta per essere scritta su file dal FTPWriter.

3. **Flusso in Entrata (Deserializzazione Conferme):**

- Legge la stringa XML di una conferma d'ordine.
- La trasforma in oggetti DTO (XmlOrderResponse).
- Estrae e mappa solo i campi essenziali (date, indirizzi, totali) in un oggetto ConfirmationStructure pulito, scartando la complessità della struttura XML originale.

Infine, la classe classe FTPConfig opera come centralizzatore delle configurazioni per il file system, caricando i percorsi delle cartelle di lavoro (ingresso, uscita e archivio) direttamente dal file ftpconfig.properties.

Oltre a esporre questi percorsi, esegue una validazione bloccante all'avvio dell'applicazione: verifica che le directory fisiche esistano realmente e, in caso contrario, impedisce l'avvio del sistema lanciando un'eccezione, prevenendo così errori di scrittura o lettura durante l'esecuzione.

Realizzazione API e Web UI

L'interfaccia operatore è una pagina HTML statica ("beemediate-ui.html") servita direttamente dal backend tramite lo UIController. Il frontend è stilizzato con Bootstrap 5 e utilizza JavaScript nativo per gestire la comunicazione asincrona con il server.

Il backend espone le funzionalità tramite il ManagerRestController, che funge da adattatore primario per i comandi HTTP:

- **Health Check:** Il frontend esegue un polling automatico ogni 5 secondi verso l'endpoint di controllo (*/manage/healthcheck*) per verificare che il backend sia attivo e aggiornare l'indicatore di stato visivo.
- **Gestione Processi:** I pulsanti dell'interfaccia invocano gli endpoint REST */manage/orders* e */manage/confirmations*. Questi delegano l'esecuzione alla logica di business (OaFManagerPort) e restituiscono all'utente un feedback testuale con il numero di elementi processati o un messaggio di errore specifico in caso di fallimento.

Figure 30 beemediate-ui.html

Unit Testing

I dettagli dei test sono stati specificati nelle testapi, consultabili su Github Pages.

Il panorama di testing del progetto Beemedaite si caratterizza per un approccio fortemente stratificato, dove ogni livello architetturale utilizza strumenti specifici in base alla propria responsabilità.

Nel cuore del sistema, il **dominio** e le sue utility, si concentrano le tecniche più sofisticate di verifica logica e matematica. Per i servizi vengono impiegati approcci formali come il Model-Based Testing con Avalla e il calcolo della copertura MCDC, affiancati da test parametrici per validare diverse condizioni di input. Le classi di utilità si distinguono per l'uso di generazione automatica dei test tramite EvoSuite e Randoop, oltre a tecniche combinatorie come il PairWise, garantendo robustezza sugli algoritmi di base.

Spostandosi verso l'**infrastruttura** (Odoo e FTP), la strategia cambia radicalmente puntando sull'isolamento dai sistemi esterni. Qui domina l'uso di Mockito e MockedStatic per simulare le dipendenze e i metodi statici, mentre per i test su file system vengono sfruttate directory temporanee dedicate. I numerosi oggetti di trasporto dati (DTO), specialmente nel pacchetto Odoo, seguono uno standard rigoroso che combina test parametrici, input-based sui costruttori e analisi MCDC sui metodi factory.

Infine, il livello più esterno dei **controller** gestisce l'interazione utente e di rete attraverso strumenti di simulazione ad alto livello: Selenium su Chromium viene utilizzato per l'automazione browser, mentre WireMock permette di testare le chiamate HTTP senza coinvolgere server reali. Tutto il codice è trasversalmente monitorato attraverso metriche di copertura strutturale, come la Statement e la Branch Coverage, per assicurare che ogni linea e ramificazione logica venga eseguita.

Test dominio

Le attività di testing per il package *com.beemediate.beemediate.domain* e i suoi sottopackage (service, utils, pojo) sono suddivise come segue:

1. Domain Service (com.beemediate.beemediate.domain.service)

In questo package sono stati testati tre servizi principali utilizzando diverse tecniche, tra cui il testing parametrico, strutturale e model-based:

- **OaFBuffer**
    - È stato applicato il **test parametrico** sui metodi loadBuffer, validateBuffer e sul costruttore.
    - È stata utilizzata la tecnica **MCDC** (Modified Condition/Decision Coverage) per il metodo loadBuffer.

    - È stata verificata la **Statement Coverage** per il metodo getBuffer.
    - È stato impiegato **Mockito** per la gestione delle dipendenze.
- **OaFValidator**
    - È stato utilizzato il **Partition Testing** per il costruttore.
    - È stato applicato il **Model-based testing con Avalla** per il metodo validate.
    - Anche qui è stato utilizzato **Mockito**.
- **OaFBatchManager**
    - È stato applicato il **Model-based testing con Avalla** sui metodi handleOrders e handleConfirmations.
    - È stato utilizzato **Mockito**.

2. Domain Utils (com.beemediate.beemediate.domain.utils)

E' stata applicata un'ampia varietà di tecniche, inclusa la generazione automatica dei test e test combinatori:

- **StringHandler**
    - **Generazione Automatica:**
        - **EvoSuite** è stato usato per i metodi isDigit e has2DigitsBetween.
        - **Randoop** è stato usato per isSubstrLessOrEqualThanSubstr2, substrCompare e containsChar.
    - **Test Combinatorio:** È stato applicato il **PairWise con IPO e condizioni** sul metodo beforeOrEqualDateTime.
    - **Copertura Strutturale:**
        - **Statement Coverage** per isSubstrLessOrEqualThanSubstr2, substrCompare, containsChar e isDateTime.
        - **MCDC** per i metodi equals, isDouble e isInteger.
    - **Partition Testing** applicato al metodo isDigit.
- **BoundedBuffer**
    - Sono stati applicati sia il **test parametrico** che il **Partition Testing**.

3. Domain POJO (com.beemediate.beemediate.domain.pojo)

Per le classi semplici di tipo POJO (Plain Old Java Object):

- **ConfirmationStructure**

- È stata verificata esclusivamente la **Statement Coverage**.

Test adattatori Odoo

Le attività di testing per il package *com.beemediate.beemediate.infrastructure.odoo* e i suoi sottopackage sono suddivise come segue:

1. Adattatori Odoo

In questo livello principale, i test si concentrano sulla gestione dell'invio dati e del recupero ordini, con un forte uso di mock per isolare le dipendenze esterne:

- **OdooDataSender**
  - Utilizza **Mockito** per la simulazione delle dipendenze.
  - Applica la tecnica **MCDC** (Modified Condition/Decision Coverage).
- **OdooOrderProvider**
  - Utilizza **Mockito** e **MockedStatic** (per metodi statici).
  - Verifica sia la **Branch Coverage** che la **Statement Coverage**.

2. Infrastructure OdooApiConfig

Qui vengono testate le configurazioni e gli stati:

- **OafStatus**
  - Testato tramite approccio **Parametrico**.
  - Verifica la **Statement Coverage**.
- **OdooApiConfig**
  - Utilizza **Mockito**.
  - Verifica sia la **Branch Coverage** che la **Statement Coverage**.

3. Infrastructure Odoo DTO (...infrastructure.odoo.dto)

Questo package contiene numerosi Data Transfer Object. Tutti seguono uno schema di testing identico e molto rigoroso che copre costruttori, conversioni stringa e metodi factory statici.

Le classi incluse sono: ArticoloPreventivoDTO, CompagniaDTO, ConsegnaDTO, ContattoConsegnaDTO, DestinazioneDTO, IdentifierDTO, PreventivoDTO, ProdottoDTO e ProdottoFornitoreDTO.

Per **tutte** queste classi sono stati applicati:

- **Test Parametrico**.

- **Input-Based testing** specificamente per il **costruttore**.
- **Statement coverage** per il metodo toString.
- **MCDC con Mockito** per i metodi static factory XML-RPC.

4. Infrastructure Odoo Mapper

I mapper, responsabili della trasformazione dei dati, sono testati per garantire la corretta logica di mappatura:

- **OrderMapper** e **AttributeMapper**
    - Entrambi verificano la **Branch + Statement coverage**.
    - Utilizzano l'**Input-based testing**.
    - Fanno uso di **Mockito** per gestire le dipendenze durante il mapping.

Test adattatore Xlsx

In questo package è presente una sola classe testata, che si occupa della lettura di file Excel:

- **XmlsxAdapter**
    - È stato applicato il **Test Parametrico**.
    - È stata utilizzata la tecnica del **Partition Testing**.
    - È stato verificato specificamente l'**uso di file in input** per i test.

Test adattatori FTP

Le attività di testing per il package *com.beemediate.beemediate.infrastructure.ftp* e i suoi sottopackage sono suddivise come segue:

1. Adattatori

In questo package principale, le classi responsabili della lettura e scrittura via FTP sono state testate con un focus sulla gestione del file system temporaneo e dei metodi statici:

- **FTPReader**
    - Utilizza **MockedStatic** (per il mocking di metodi statici).
    - Fa uso di **TempDir** (per la gestione di directory temporanee nei test).
    - Verifica la **Branch + Statement Coverage**.
- **FTPWriter**

    - Presenta le stesse tecniche del Reader: **MockedStatic**, **TempDir** e **Branch + Statement Coverage**.

2. Infrastructure FTPConfig

- **FTPConfig**
    - È stata applicata la tecnica del **Partition Testing**.
    - Vengono testate le **Temporary directories**.

3. Infrastructure FTP DTO

Per gli oggetti di trasferimento dati relativi all'XML, i test si concentrano sulla struttura e sulla copertura del codice base:

- **XmlOrderResponse**
    - Verifica la **Statement Coverage**.
    - Controlla la **Struttura di riferimento**.
- **XmlOrder**
    - Anche qui viene verificata la **Statement Coverage** e la **Struttura di riferimento**.

Test interfaccia Web

ManagerRestController e UIController Vengono testati utilizzando **Selenium** con **Chromium**, tecnica indicata solitamente per test end-to-end o di interfaccia grafica automatizzati tramite browser.

Inoltre, viene utilizzato **WireMock**, uno strumento specifico per simulare servizi HTTP (stubbing di API), utile per testare come i controller reagiscono alle risposte di rete senza effettuare chiamate reali.

Copertura e analisi statica

Eccetto STAN, I report di analisi statica sono pubblici e consultabili via le risorse collegate al Progetto come il maven site su Github Pages.

Qualità del codice e Bug

PMD segnala poche regole per lo più di priorità 4 (medio-bassa) ed una di priorità 1 (ora fixata). SpotBugs segnala alert perlopiù low e qualcuno medium. SonarCloud segnala 64 Issues legati a manutenibilità di cui 1 alto: la complessità cognitiva di OrderMapper.java è 19 ma il threshold stabilito è 15.

Sicurezza

Snyk segnala le seguenti dipendenze ad alto rischio

- 2 vulnerabilità“critiche”, legate a xmlrpc-client e cmlrpc-common per “Deserialization of Untrusted Data”, non fixabile;
- 3 vulnerabilità alte legate a spring-boot-starter (consente path-traversal e Incorrect Authorization) e poi-ooxml (causa Uncontrolled Recursion)

Architettura

STAN segnala

- Pollution=0.55 significa “spaghetti code” (componenti troppo legati tra loro)
- Cc=1.46 indica un buon livello di complessità: metodi puliti e semplici, “fanno una cosa sola”
- Fat=2 significa che ci sono 2 classi “tuttofare”
- D=-1 per diversi pacchetti (Zone of Pain, codice troppo rigido, troppe dipendenze)
- Il package infrastructure.ftp.config ha Tangled =11.05% : c’è un problema legato alle dipendenze cicliche

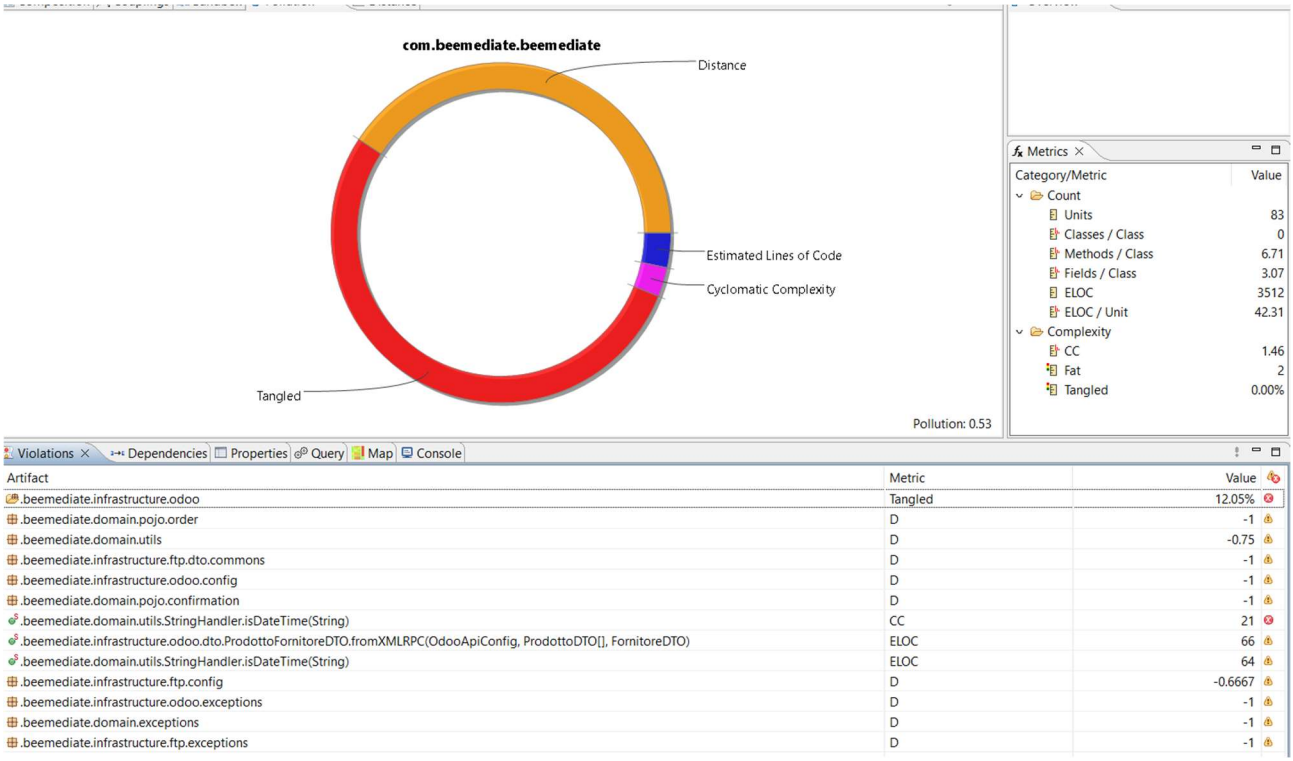


Figure 31 analisi STAN

Coverage

JaCoCo indica una copertura di circa il 97% del codice.

Aggregazione e monitoraggio continuo

Il grafico "Risk" di SonarCloud evidenzia un buono stato di salute del codice sorgente:

- **Qualità e Sicurezza:** La totalità dei componenti analizzati presenta un rating **"A" (Verde)**, indicando l'assenza di Bug e Vulnerabilità critiche rilevate.
- **Manutenibilità:** La concentrazione dei file nell'angolo inferiore sinistro dimostra un **Debito Tecnico trascurabile** (max 35 min per file) associato a un'**alta copertura dei test** (>80%).
- **Conclusioni:** Il codice risulta conforme ai principi di Clean Code, con classi di dimensioni contenute e ben testate, riducendo drasticamente il rischio di regressioni future.

Figure 32 Grafico SonarCloud per rischio e Technical Debt