UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

Corso di laurea in Informatica
Insegnamento di Basi di Dati I

Anno accademico 2023/2024

Progettazione e sviluppo di una base di dati 

Unina Delivery

Autori:			**Docente:**
Nome	**Matricola**	**E-mail**	
Gianluca Fiorentino	N86/4650	*gianlu.fiorentino@studenti.unina.it*	Prof.Mara Sangiovanni
Luigi Dota	N86/4718	*lui.dota@studenti.unina.it*	

Indice

1 Descrizione del Progetto

1.1 | Analisi del Problema

Descrizione. Unina Delivery è un sistema di gestione della logistica delle spedizioni di merci. Sulla base degli ordini dei clienti, l'operatore può, attraverso l'uso del sistema, pianificare le spedizioni, tenendo conto di fattori come la disponibilità della merce ed il suo peso, la presenza di mezzi di trasporto e corrieri disponibili.

Analisi. Si provvederà alla progettazione ed allo sviluppo di una base di dati dedicata alla descrizione, memorizzazione e gestione del problema descritto. In particolare, si intende procedere nel seguente modo: individuazione delle classi e relativi attributi, realizzazione di schema ER e class diagram UML, ristrutturazione con successivo mapping e schema logico; infine, schema fisico, con trigger e funzioni. Si presuppone che le spedizioni di merci avvengano su **territorio nazionale**; non sono dunque previste spedizioni internazionali.
Per ora, soffermiamoci sugli aspetti più "ad alto livello" del problema, vale a dire individuare le classi (o entità) e gli attributi di ognuna di esse, nonché le relazioni che legano le classi.

1.2 | Individuazione delle classi

Di seguito sono riportate esclusivamente le classi, con una leggera descrizione di ciò che rappresentano all'interno del problema:

- Articolo: la merce presente nel magazzino;
- Ordine: richiesta di acquisto del cliente per specifici articoli;
- Persona: soggetto direttamente coinvolto nel problema;
- Cliente: colui che, eventualmente, effettua ordini;
- Operatore: colui che gestisce gli ordini e si occupa di pianificare le spedizioni;
- Corriere: colui che spedisce l'articolo presso il luogo designato;
- Spedizione: l'atto pratico della consegna al cliente degli articoli ordinati;
- Veicolo: il mezzo di trasporto tramite cui il corriere effettua la spedizione.

1.2.1 Qualche osservazione sulle classi

Vi è una particolare relazione tra alcune delle classi coinvolte nel problema. Ma prima di entrare nel merito della questione, è importante riportare alla mente quanto segue.

Definizione. Per *generalizzazione* s'intende una relazione di ereditarietà tra le classi. Questa relazione indica che una classe, denominata *specializzazione* (o sottoclasse), eredita attributi e comportamenti da un'altra classe, chiamata *generalizzazione* (o superclasse).

In questo caso, vi è una relazione di questo esatto tipo tra le classi: Cliente, Corriere e Operatore. Esse, infatti, sono specializzazioni della superclasse **Persona**.

Si fa inoltre presente che sarebbe stato possibile craeare un sistema per manadare una notifica al cliente qualora l'ordine fosse arrivato a destinazione, ma per ragioni di semplicità non è stata adottata quest'idea.

1.3 | Attributi delle classi

Di seguito saranno elencati gli attributi di ogni entità tramite una tabella autoesplicativa:

Entità	Attributi
Articolo	CodArticolo, Nome, Descrizione, Peso, Quantità, Prezzo
Ordine	CodOrdine, Data, Indirizzo, Città, Cellulare, PrezzoTot, Consegnato
Persona	Nome, Cognome
Cliente	Indirizzo, Cellulare, E-Mail
Operatore	CodFiscale, Username, Password
Corriere	CodFiscale, Cellulare, Disponibilità
Spedizione	CodSpedizione, Data, Orario, Avviata, Completata
Veicolo	Targa, PesoSopportato, Disponibilità

1.3.1 Qualche osservazione sugli attributi

Per quanto riguarda entità come "Operatore" e "Corriere" sarebbe possibile pensare di attribuire loro attributi relativi al tipo di contratto e alla sua durata. Tuttavia, si è scelto di non mettere attributi del genere al fine di evitare complicazioni della base di dati. In questa visione, se, ad esempio, un corriere cessa il suo rapporto di lavoro, verrà semplicemente cancellata l'istanza (o la tupla) che lo riguarda.
Si fa inoltre presente che gli attributi "Indirizzo", "Città" e "Cellulare" dell'entità "Ordine" fanno riferimento a quelli relativi alla consegna dell'ordine stesso, in quanto sono dati che potrebbero differire da quelli del cliente; basti pensare al caso di un cliente che ordina per conto terzi.

1.4 | Relazioni tra le classi

Si riporta alla mente del lettore il significato di relazione e cardinalità della relazione:

Relazione. Si tratta di un legame logico tra due o più entità. Solitamente viene identificata univocamente tramite un verbo. Il numero di entità coinvolte in una relazione prende il nome di **grado** di una relazione.

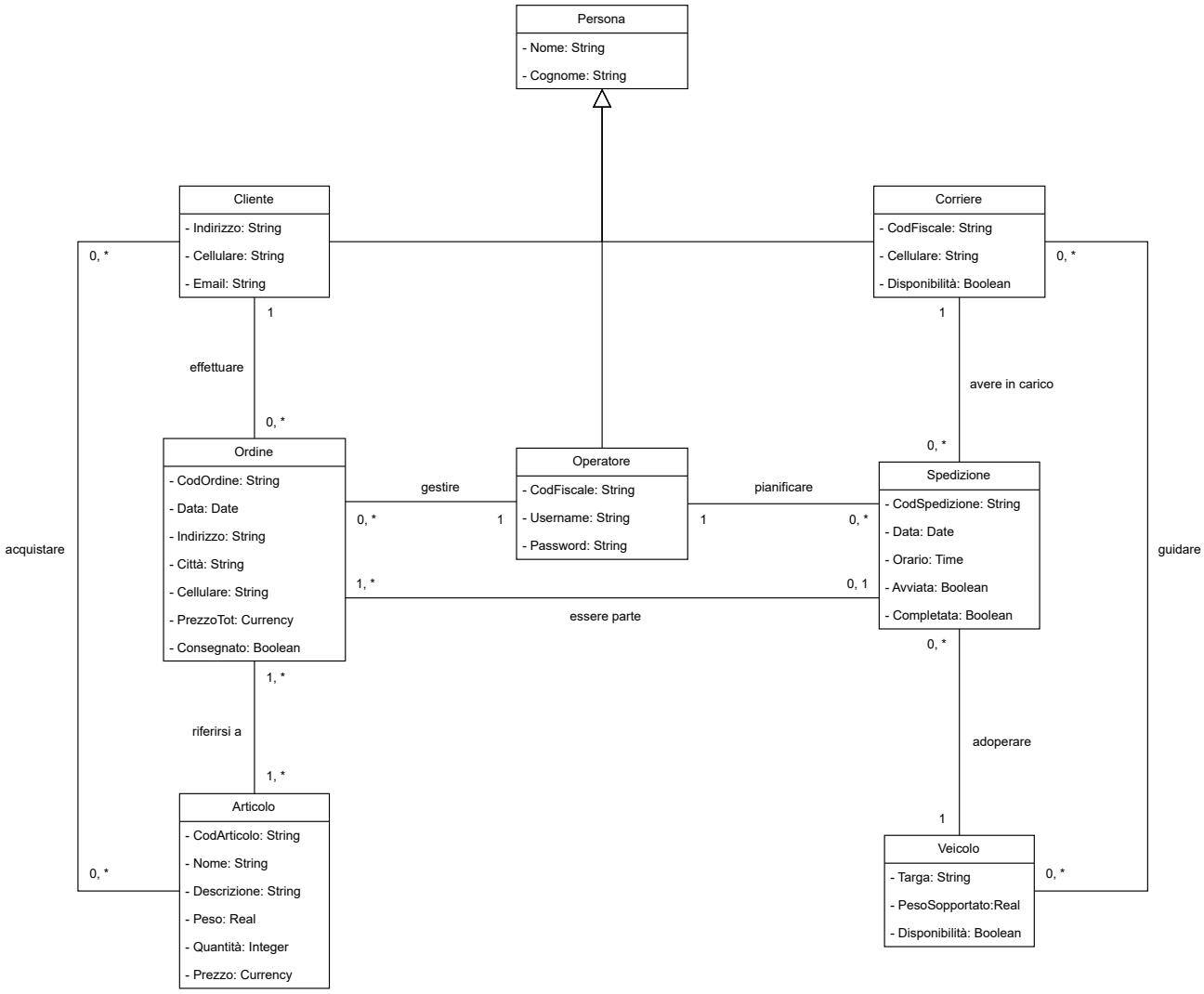
Cardinalità di una relazione. Consiste in una coppia di numeri (min, max), che rispettivamente indicano il numero minimo e il numero massimo di elementi che possono essere associati a un'entità in una relazione. La cardinalità minima può essere 0 (la relazione è facoltativa) oppure 1 (la relazione è obbligatoria); la cardinalità massima è compresa tra 1 e N (dove N sta per un tetto massimo non definito).

Si invita il gentile lettore ad osservare i diagrammi riportati di seguito per osservare le relazioni e le relative cardinalità appena descritte poc'anzi.

2 Progettazione concettuale

A seguito dell'analisi affrontata nella sezione precedente, presentiamo di seguito due rappresentazioni che descrivono il problema: class diagram UML e schema ER.

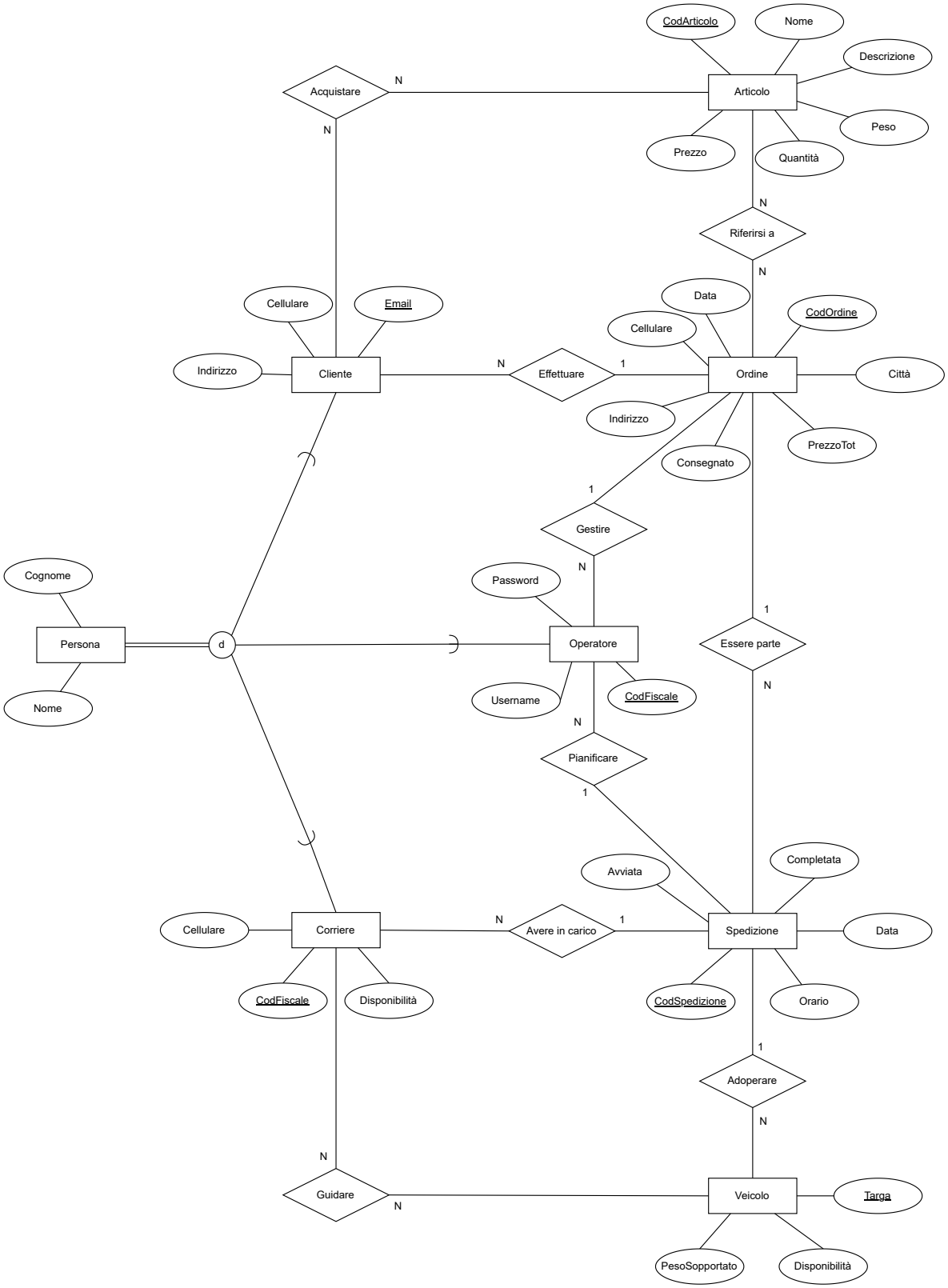
2.1 | Class Diagram UML



2.1.1 Note al Class Diagram UML

Tra gli aspetti che si possono notare solo nel Class Diagram, ricordiamo che vi sono la visibilità (in questo caso, ***private*** per tutti gli attributi), e il tipo per ogni attributo di ogni entità.

2.2 | Schema ER



2.2.1 Note allo schema ER

Invece, per quanto riguarda lo schema ER, si possono notare aspetti quali la ***primary key***, tipica del modello relazionale, per le entità che ne possiedono una, nonché informazioni aggiuntive sulla generalizzazione, che affronteremo, però, in un altro momento.

2.3 | Ristrutturazione

Prima di procedere con gli argomenti che affronteremo in questa sezione, è bene ricordare la **scelta** del modello logico di rappresentazione dei dati del database. Nel caso specifico, infatti, si procederà con il modello relazionale. Se si fosse optato per un modello diverso, allora si sarebbero dovute fare considerazioni diverse.

In particolare, ci occuperemo di:

1. eliminare eventuali attributi strutturati o multivalore;
2. eliminare le generalizzazioni/specializzazioni;
3. scegliere una primary key per ogni entità;
4. analizzare eventuali ridondanze.

2.3.1 Attributi strutturati o multivalore

Al fine di individuare attributi strutturati e attributi multivalore, ricordiamo prima, come di consueto, di cosa stiamo parlando:

Attributo strutturato. Si tratta di un attributo che contiene un insieme di dati, e pertanto può essere suddiviso in parti più piccole, ognuna delle quali ha un significato specifico. Si pensi, ad esempio, al caso di un numero cellulare, che ha un prefisso oltre all'effettivo numero. Il prefisso, infatti, è un dato a sé stante, tramite il quale è possibile risalire alla nazionalità del numero di riferimento.

Attributo multivalore. Si tratta di un attributo che può contenere più valori per una stessa istanza. Ad esempio, l'attributo "Colore" nell'entità "Maglia" è potenzialmente multivalore, a meno che non si voglia descrivere una realtà un po' anacronistica in cui tutte le maglie sono monocolore.

Da un'osservazione un po' più attenta, è facile notare che, nel caso specifico, non vi siano attributi multivalore. Non vi si può dire lo stesso, però, per quanto riguarda gli attrbuti strutturati: di fatti, "Indirizzo" è proprio un attributo strutturato, e contiene informazioni quali "Via", e quindi il nome della via, "Civico", cioè il numero civico, e "CAP". Stesso discorso vale per "Cellulare", "Data" e "Orario".
Mentre per l'indirizzo potrebbe rivelarsi una saggia scelta trattare i 3 attributi in maniera distinta - si pensi all'ipotesi di ordinare una tabella per CAP -, per gli altri attributi la natura strutturata rappresenta, in un certo modo, un vantaggio. Essi, infatti sono definiti con un formato standard (YYYY-MM-DD per "Data" e HH:MM per "Orario") che permette in modo univoco ed efficiente la gestione stessa dei dati. Per "Cellulare" potremmo considerare del tutto inutile il prefisso, in quanto, come già specificato qui, si tratta di una base di dati contenente informazioni su base nazionale e, pertanto, non sarebbe così irrealistico pensare che clienti e corrieri abbiano una SIM italiana (o, comunque, del Paese di chi ha commissionato il database). Al netto di quanto detto poc'anzi, ***verrà trattato unicamente "Indirizzo"***.

2.3.2 Generalizzazioni/Specializzazioni

Abbiamo già affrontato precedentemente il significato di generalizzazione e conseguentemente anche di specializzazione. Inoltre, dallo schema ER si può evincere che l'unica generalizzazione presente è "totale" e "disgiunta". Perché questa scelta? È presto detto:

- **totale**, perché, all'interno del minimondo che siamo intenti a descrivere, una persona può essere un operatore, un cliente oppure un corriere, ma necessariamente sarà almeno uno di loro;
- **disgiunta**, perché si presuppone che se una persona è un operatore, allora **non** può essere anche un corriere, e viceversa; se una persona è un operatore o un corriere però, in determinate circostanze, potrebbe essere anche cliente. Tuttavia, ciò dipende meramente dalle politiche della ditta di logistica, senza contare che, da un punto di vista contabile, anche se un operatore o un corriere della ditta acquistasse un articolo e se lo facesse spedire 'in proprio', l'operatore o il corriere in questione **non** sarebbe considerato cliente.

Pertanto, una persona può essere un operatore, un cliente o un corriere, ma esattamente uno ed uno solo di loro. In quanto il tipo di gerarchia è totale disgiunta, si è optato per ***accorpare la classe generale nelle sottoclassi***.

2.3.3 Individuazione chiavi primarie

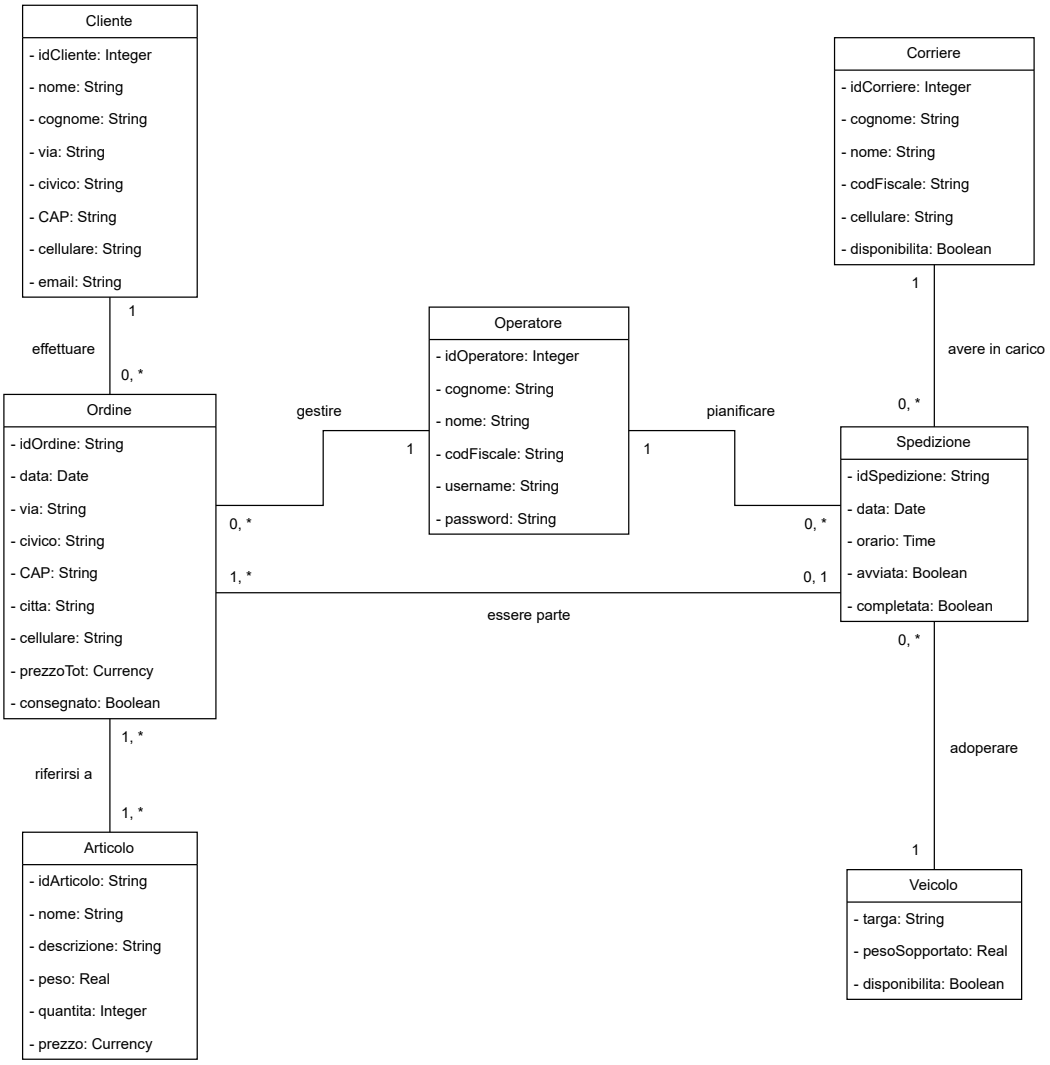
Le chiavi primarie, in realtà, sono state già individuate, ed è possibile leggere quali esse sono dallo schema ER, come precedentemente anticipato. Possiamo, allora, discutere di quanto alcune chiavi siano più o meno efficaci. Ad esempio, di un "Veicolo" so per certo che ci sarà una targa, univoca per ogni mezzo, e che potrò usarla come primary key. Discorso un po' diverso per "Cliente", però, di cui magari ci interessa salvare un'istanza nel database anche senza conoscerne l'e-mail. Allora, per entità quali "Cliente", "Operatore" e "Corriere" potrebbe risultare opportuno ideare una chiave surrogata, che, per convenzione, chiameremo con il prefisso *Id* seguito dal nome della classe.
Per coerenza, anche PK quali "CodArticolo" avranno il prefisso cambiano in Id.

2.3.4 Analisi delle ridondanze

Quelli con un occhio un po' più attento, avranno sicuramente notato che i due diagrammi hanno qualche aspetto probabilmente superfluo, ed è effettivamente così. Infatti, la relazione <*Corriere guida veicolo*>, in un certo modo, è già codificata attraverso la classe "Spedizione". Infatti, di una spedizione avremo informazioni sia di chi la ha a carico (un corriere) sia del mezzo utilizzato per effettuarla (veicolo); conseguentemente, per ogni spedizione conosciamo già quale corriere ha guidato quale veicolo, anche senza la specifica relazione.
Discorso del tutto equivalente vale per <*Cliente acquista articolo*>, la cui relazione risulta già codificata dalla classe "Ordine".
Per tale ragione, ***le due relazioni di cui sopra saranno eliminate***.

`Nota:` Non si faccia l'errore di credere che sia ridondante anche la relazione <*Ordine è parte di spedizione*> perché già codificata dalla classe "Operatore": infatti, le relazioni coinvolte specificano aspetti diversi della realtà, motivo per cui non vi sono ridondanze.

2.4 | Class Diagram UML Ristrutturato



Per praticità, ci limiteremo alla presentazione del class diagram UML ristrutturato, senza riproporre anche lo schema ER.

2.5 | Dizionario

2.5.1 Dizionario delle classi

Classe	Attributo		Descrizione
	nome	**cardinalità**	
Cliente	*idCliente*	totale	Classe contenente tutte le informazioni relative ai possibili acquirenti.
	nome	totale	
	cognome	totale	
	via	totale	
	civico	totale	
	CAP	totale	
	cellulare	parziale	
	e-mail	parziale	
Operatore	*idOperatore*	totale	Classe contenente tutte le informazioni relative agli operatori. Gli attributi username e password (crittografata) sono necessari per permettere all'operatore di accedere al database.
	nome	totale	
	cognome	totale	
	codFiscale	totale	
	username	totale	
	password	totale	
Corriere	*idCorriere*	totale	Classe contenente tutte le informazioni relative ai corrieri.
	nome	totale	
	cognome	totale	
	codFiscale	totale	
	cellulare	totale	
	disponibilità	totale	
Ordine	*idOrdine*	totale	Classe contenente i dettagli relativi agli ordini effettuati dai clienti. Si prevede la possibilità di spedire ad un indirizzo diverso dal proprio.
	data	totale	
	via	totale	
	civico	totale	
	CAP	totale	
	città	totale	
	cellulare	totale	
	prezzoTot	totale	
	consegnato	totale	
Articolo	*idArticolo*	totale	Classe contenente le informazioni relative alla merce presente in magazzino.
	nome	totale	
	descrizione	totale	
	peso	totale	
	quantità	totale	
	prezzo	totale	
Spedizione	*idSpedizione*	totale	Classe contenente le informazioni relative alle spedizioni effettuate o da effettuare.
	data	totale	
	orario	totale	
	avviata	totale	
	completata	totale	
Veicolo	*targa*	totale	Classe contenente tutte le informazioni relative ai mezzi di trasporto con cui vengono effettuate le spedizioni.
	pesoSopportato	totale	
	disponibilità	totale	

2.5.2 Dizionario delle associazioni

Associazione	Classi coinvolte	Descrizione
Effettuare	Ordine - Cliente	Un cliente può effettuare zero o più ordini [0, *]. Un ordine può essere effettuato da uno ed un sol cliente [1].
Riferirsi a	Ordine - Articolo	Un ordine fa riferimento ad uno o più articoli [1, *]. Ad un articolo fanno riferimento uno o più ordini [1, *]. Tale relazione genera una tabella ponte che comprende, oltre alle due foreign key, anche gli attributi quantità, peso e prezzo.
Gestire	Ordine - Operatore	Un operatore può gestire zero o più ordini [0, *]. Un ordine può essere gestito da uno ed un sol operatore [1].
Pianificare	Spedizione - Operatore	Un operatore può pianificare zero o più spedizioni [0, *]. Una spedizione può essere pianificata da uno ed un sol operatore [1].
Essere parte	Spedizione - Ordine	Un ordine può essere parte di zero o al più una spedizione [0, 1]. Di una spedizione sono parte uno o più ordini [1, *].
Avere in carico	Spedizione - Corriere	Un corriere può avere in carico zero o più spedizioni [0, *]. Una spedizione può essere ad in carico di uno ed un sol corriere [1].
Adoperare	Spedizione - Veicolo	Un veicolo viene adoperato per zero o più spedizioni [0, *]. Una spedizione si può adoperare di uno ed un sol veicolo [1].

2.5.3 Dizionario dei vincoli

Vincolo	Tipo	Attributi coinvolti	Descrizione
Regex nomi	D	Nome, cognome, città	La stringa può contenere solo lettere dell'alfabeto italiano.
Regex codFisc	D	CodFiscale	La stringa deve essere del formato tipico italiano.
Regex cellulare	D	Cellulare	La stringa può contenere solo numeri.
Regex e-mail	D	Email	La stringa deve essere del formato [parola]@[parola][.it/.com].
Regex username and password	D	Username, password	"username" deve contenere almeno 4 caratteri; "password" deve contenere almeno 8 caratteri. Inoltre, i due attributi possono contenere solo lettere o numeri.
Date possibili	D	ORDINE(data)	La data della tabella ORDINE deve essere precedente al momento in cui la si sta inserendo.
Positive and derivable values	D	Peso, pesoSopportato, quantità prezzo, prezzoTot, prezzoPar, pesoPar	Gli attributi "prezzo" e "peso" devono essere > 0; "quantità" >= 0. PrezzoTot si ottiene sommando prezzo x quantità di tutti gli articoli dell'ordine. Stesso vale per Prezzo e Peso (ma con peso x quantità) della tabella ponte.
Nullable values	D	CLIENTE(cellulare, email) ORDINE(idOperatore, idSpedizione)	Gli attributi descritti possono essere NULL; tutti gli altri sono NOT NULL.
Default values	D	CORRIERE/VEICOLO(disponibilità) ORDINE(prezzoTot, consegnato) SPEDIZIONE(data, orario, avviata, completata)	disponibilità=TRUE; prezzoTot=0, consegnato=FALSE; data=CURRENT, orario=CURRENT, avviata=FALSE, completata=FALSE.
Rintracciamento	N	CLIENTE(cellulare, email)	Al fine (teorico) di rintracciare il cliente, cellulare ed e-mail possono essere NULL, ma solo uno dei due, non entrambi.
Sped completata	N	Avviata, completata	Completata può essere TRUE solo se Avviata era stata aggiornata a TRUE.
Cliente unique	IR	CLIENTE(cellulare, email)	Gli attributi devono essere unique singolarmente.
Corriere unique	IR	CORRIERE(codFiscale, cellulare)	Gli attributi devono essere unique singolarmente.
Operatore unique	IR	OPERATORE(codFiscale, username, password)	Gli attributi devono essere unique singolarmente.
CompOrdine unique	IR	COMPORDINE(idordine, idarticolo)	Gli attributi devono essere unique in coppia.
Codici fiscali diversi	IRR	CORRIERE(codFiscale) OPERATORE(codFiscale)	L'attributo codFiscale deve essere diverso anche tra tutte le istanze di corriere e quelle di operatore. Vincolo da ristrutturazione.
Cellulari diversi	IRR	CLIENTE(cellulare) CORRIERE(cellulare)	Come sopra, anche cellulare deve essere diverso. Altro vincolo da ristrutturazione.
Pianifica Spedizione 1	IRR	CORRIERE(idCorriere, disponibilità)	Per una spedizione, posso scegliere unicamente un corriere che risulta essere disponibile.
Pianifica Spedizione 2	IRR	ARTICOLO(peso) VEICOLO(targa, pesoSopportato, disp.)	Per una spedizione, posso scegliere unicamente un veicolo che risulta essere disponibile e che sia in grado di trasportare le merci degli ordini da spedire, in base al pesoSopportato.
Componi Ordine	IRR	ARTICOLO(idArticolo, quantità) COMPORDINE(idArticolo, quantità)	Un ordine non può essere ultimato se composto da articoli la cui presenza non è certificata in magazzino alla giusta quantità.
Componi Sped	IRR	SPEDIZIONE(avviata) ORDINE(idSpedizione)	Se la spedizione è partita, allora non è possibile aggiungere a quella spedizione altri ordini.
Aggiorna scorte	IRR	ARTICOLO(idArticolo, quantità) COMPORDINE(idArticolo, quantità)	Quando di un ordine viene specificato quali articoli ne fanno parte, allora si aggiornano le scorte di articoli in magazzino.
Aggiorna ordine	IRR	ORDINE(idOrdine, prezzoTot) COMPORDINE(prezzoPar, pesoPar) ARTICOLO(idArticolo, quantità, prezzo, peso)	Man mano che vengono inseriti i dettagli dell'ordine, vengono aggiornati gli attributi della tabella ponte e, consequentemente, quelli di Ordine.
Possible ordine consegnato	IRR	SPEDIZIONE(avviata) ORDINE(consegnato)	Se la spedizione non è partita, gli ordini non possono essere stati consegnati.
Spedizione completata 1	IRR	SPEDIZIONE(completata) ORDINE(consegnato)	Se la spedizione è completata, allora tutti gli ordini sono stati consegnati.
Spedizione completata 2	IRR	SPEDIZIONE(completata) ORDINE(consegnato)	Se tutti gli ordini sono stati consegnati, allora la spedizione può dirsi completata.
Corriere e Veicolo disponibilità 1	IRR	SPEDIZIONE(completata) idCorriere, targa, disponibilità	Se la spedizione non è completata, allora corriere e veicolo associati non sono disponibili per altre spedizioni.
Corriere e Veicolo disponibilità 2	IRR	SPEDIZIONE(completata) idCorriere, targa, disponibilità	Se la spedizione è completata, allora corriere e veicolo associati tornano ad essere disponibili per altre spedizioni.

`Legenda:`

D → Dominio
N → N-pla
IR → Intrarelazionale
IRR → Interrelazionali

3 Progettazione logica

3.1 | Mapping delle associazioni

Di seguito, è riportato il mapping delle associazioni, preceduto da una breve descrizione. Come si può notare, non è stato riportato il caso Uno-A-Uno (1:1) in quanto del tutto assente nel class diagram.

3.1.1 Associazioni Uno-A-Molti (1:N)

Dopo aver individuato le chiavi primarie, bisogna individuare il lato debole della realazione. Per capire ciò basta osservare le cardinalità. La chiave primaria (PK) della classe forte (1) migra nella classe debole (N).
Nel problema in questione abbiamo:

- ***Cliente - effettua - Ordine***: la chiave primaria di Cliente (*idCliente*) migrerà verso Ordine come chiave esterna (FK);
- ***Operatore - gestisce - Ordine***: la chiave primaria di Operatore (*idOperatore*) migrerà verso Ordine come chiave esterna.
- ***Operatore - pianifica - Spedizione***: la chiave primaria di Operatore (*idOperatore*) migrerà verso Spedizione come chiave esterna.
- ***Ordine - è parte di - Spedizione***: la chiave primaria di Ordine (*idOrdine*) migrerà verso Spedizione come chiave esterna.
- ***Corriere - ha in carica - Spedizione***: la chiave primaria di Corriere (*idCorriere*) migrerà verso Spedizione come chiave esterna.
- ***Spedizione - si adopera - Veicolo***: la chiave primaria di Spedizione (*idSpedizione*) migrerà verso Veicolo come chiave esterna.

3.1.2 Associazioni Molti-A-Molti (N:N)

Dopo l'individuazione delle chiavi primarie, bisogna creare una **tabella ponte** che contenga in sé le primary key di entrambe le classi sottoforma di foreign key.
Nel problema in questione abbiamo solamente:

- ***Ordine - Riferirsi a - Articolo***: il risultato sarà una tabella ponte che prenderà il nome di "CompOrdine" (composizione ordine) e che conterrà al suo interno idArticolo, chiave primaria di Articolo, e idOrdine, chiave primaria di Ordine, entrambe chiavi esterne. Inoltre, saranno presenti nella tabella anche attributi quali quantità, pesoPar, prezzoPar (dove il suffisso "Par" sta per "parziale").

3.2 | Modello Logico

Di seguito sono riportate le tabelle nella forma: [nome tabella] ([attributo 1], [attributo 2], ...).

Cliente	(idCliente, nome, cognome, via, civico, CAP, cellulare, email)
Operatore	(idOperatore, nome, cognome, codFiscale, username, password)
Corriere	(idCorriere, nome, cognome, codFiscale, cellulare, disponibilita)
Ordine	(idOrdine, data, via, civico, CAP, citta, cellulare, prezzoTot, consegnato, idCliente, idOperatore, idSpedizione)
CompOrdine	(idOrdine, idArticolo, quantita, pesoPar, prezzoPar)
Articolo	(idArticolo, nome, descrizione, peso, quantita, prezzo)
Spedizione	(idSpedizione, data, orario, avviata, completata, idOperatore idCorriere, targa)
Veicolo	(targa, pesoSopportato, disponibilita)

`Notazione:`
Per indicare la ***primary key*** abbiamo utilizzato la sottolineatura.
Per indicare la ***foreign key*** abbiamo utilizzato la doppia sottolineatura.

4 Progettazione fisica

4.1 | Definizioni tabelle SQL

```sql
-- TABLE: Clienti
CREATE TABLE clienti
(
    idCliente    INTEGER          PRIMARY KEY,
    nome         VARCHAR(20)      NOT NULL  CHECK (nome ~ '^[A-Za-z]+$'),
    cognome      VARCHAR(20)      NOT NULL  CHECK (cognome ~ '^[A-Za-z]+$'),
    via          VARCHAR(100)     NOT NULL  CHECK
                                      (via ~ '^[A-Za-z][A-Za-z0-9\s.,-]*$'),
    civico       VARCHAR(3)       NOT NULL  CHECK (civico ~ '^[0-9]{1,3}$'),
    CAP          CHAR(5)          NOT NULL  CHECK (CAP ~ '^[0-9]{5}$'),
    cellulare    CHAR(10)         UNIQUE    CHECK (cellulare ~ '^[0-9]{10}$'),
    email        VARCHAR(50)      UNIQUE    CHECK
                 (email ~ '^[a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z]{2,}$'),
    /*sezione costraint*/
    CONSTRAINT cell_or_email CHECK
                 (cellulare IS NOT NULL OR email IS NOT NULL)
);

/*  Per la PK, si crea poi una sequenza ed un trigger che la sfrutta prima
    dell'inserimento.
    Tale ragionamento si ripete per le tabelle che lo richiedono.   */
```

```sql
--TABLE: Operatori
CREATE TABLE operatori
(
    idoperatore INTEGER      PRIMARY KEY,
    nome        VARCHAR(20)  NOT NULL    CHECK (nome ~ '^[A-Za-z]+$'),
    cognome     VARCHAR(20)  NOT NULL    CHECK (cognome ~ '^[A-Za-z]+$'),
    codFiscale  CHAR(16)     NOT NULL    UNIQUE  CHECK
                (codFiscale ~ '^[A-Z]{6}\d{2}[A-Z]\d{2}[A-Z]\d{3}[A-Z]$'),
    username    VARCHAR(50)  NOT NULL    UNIQUE  CHECK
                    (LENGTH(username) >= 4 AND username ~ '^[a-zA-Z0-9]+$'),
    pass        VARCHAR(50)  NOT NULL    UNIQUE  CHECK
                    (LENGTH(pass) >= 8 AND pass ~ '^[a-zA-Z0-9]+$')
);
```

```sql
--TABLE: Corrieri
CREATE TABLE corrieri
(
    idcorriere      INTEGER      PRIMARY KEY,
    nome            VARCHAR(20)  NOT NULL    CHECK (nome ~ '^[A-Za-z]+$'),
    cognome         VARCHAR(20)  NOT NULL    CHECK (cognome ~ '^[A-Za-z]+$'),
    codFiscale      CHAR(16)     NOT NULL    UNIQUE  CHECK
                (codFiscale ~ '^[A-Z]{6}\d{2}[A-Z]\d{2}[A-Z]\d{3}[A-Z]$'),
    cellulare       CHAR(10)     NOT NULL    UNIQUE  CHECK
                                                (cellulare ~ '^[0-9]{10}$'),
    disponibilita   BOOLEAN      NOT NULL    DEFAULT TRUE
);
```

```sql
--TABLE: Veicoli
CREATE TABLE veicoli
(
    targa           CHAR(7)         PRIMARY KEY CHECK
                    (targa ~ '^[A-Z]{2}\d{3}[A-Z]{2}$'),
    pesoSupportato  DECIMAL(6,2)    NOT NULL    CHECK (pesoSupportato > 0),
    disponibilita   BOOLEAN         NOT NULL    DEFAULT TRUE
);
```

```sql
--TABLE: Articoli
CREATE TABLE articoli
(
    idarticolo  VARCHAR(15)     PRIMARY KEY,
    nome        VARCHAR(50)     NOT NULL,
    descrizione VARCHAR(200)    NOT NULL,
    quantita    INTEGER         NOT NULL    CHECK (quantita >= 0),
    peso        DECIMAL(5,2)    NOT NULL    CHECK (peso > 0),
    prezzo      DECIMAL(8,2)    NOT NULL    CHECK (prezzo > 0)
);
```

```sql
--TABLE: Spedizioni
CREATE TABLE spedizioni
(
    idspedizione    VARCHAR(15) PRIMARY KEY,
    data_sped       DATE        NOT NULL    DEFAULT CURRENT_DATE,
    orario          TIME        NOT NULL    DEFAULT CURRENT_TIME,
    avviata         BOOLEAN     NOT NULL    DEFAULT FALSE,
    completata      BOOLEAN     NOT NULL    DEFAULT FALSE,
    idoperatore     INTEGER     NOT NULL,
    idcorriere      INTEGER     NOT NULL,
    targa           CHAR(8)     NOT NULL,
    /*sezione costraint*/
    CONSTRAINT fk_operatore_in_spedizione   FOREIGN KEY (idoperatore)
    REFERENCES operatori(idoperatore) ON DELETE CASCADE ON UPDATE CASCADE,

    CONSTRAINT fk_corriere_in_spedizione    FOREIGN KEY (idcorriere)
    REFERENCES corrieri(idcorriere)   ON DELETE CASCADE ON UPDATE CASCADE,

    CONSTRAINT fk_targa_in_spedizione       FOREIGN KEY (targa)
    REFERENCES veicoli(targa)         ON DELETE CASCADE ON UPDATE CASCADE,

    CONSTRAINT completata_se_avviata CHECK (completata IS NOT TRUE OR
                                            avviata IS TRUE)
);
```

```sql
--TABLE: Ordini
CREATE TABLE ordini
(
    idordine        VARCHAR(15)     PRIMARY KEY,
    data_ord        DATE            NOT NULL    CHECK
                                                (data_ord <= CURRENT_DATE),
    via             VARCHAR(100)    NOT NULL    CHECK
                                    (via ~ '^[A-Za-z][A-Za-z0-9\s.,-]*$'),
    civico          VARCHAR(3)      NOT NULL    CHECK
                                                (civico ~ '^[0-9]{1,3}$'),
    CAP             CHAR(5)         NOT NULL    CHECK
                                                    (CAP ~ '^[0-9]{5}$'),
    citta           VARCHAR(30)     NOT NULL    CHECK
                                        (citta ~ '^[A-Za-z][A-Za-z\s]*$'),
    cellulare       CHAR(10)        NOT NULL    CHECK
                                                (cellulare ~ '^[0-9]{10}$'),
    prezzoTot       DECIMAL(12,2)   NOT NULL    DEFAULT 0   CHECK
                                                        (prezzoTot >= 0),
    consegnato      BOOLEAN         NOT NULL    DEFAULT FALSE,
    idcliente       INTEGER         NOT NULL,
    idoperatore     INTEGER,
    idspedizione    VARCHAR(15),
    /*sezione costraint*/
    CONSTRAINT fk_cliente_in_ordine     FOREIGN KEY (idcliente)
    REFERENCES Clienti(idcliente)       ON DELETE CASCADE ON UPDATE CASCADE,

    CONSTRAINT fk_operatore_in_ordine   FOREIGN KEY (idoperatore)
    REFERENCES operatori(idoperatore)   ON DELETE SET NULL ON UPDATE CASCADE
    ,

    CONSTRAINT fk_spedizione_in_ordine  FOREIGN KEY (idspedizione)
    REFERENCES spedizioni(idspedizione) ON DELETE SET NULL ON UPDATE CASCADE
);
```

```sql
--TABLE: CompOrdine (tabella ponte, COMPosizione ordine)
CREATE TABLE compOrdine
(
    idordine    VARCHAR(15)     NOT NULL,
    idarticolo  VARCHAR(15)     NOT NULL,
    quantita    INTEGER         NOT NULL    CHECK (quantita > 0),
    pesoPar     DECIMAL(6,2)    NOT NULL    CHECK (pesoPar > 0),
    prezzoPar   DECIMAL(12,2)   NOT NULL    CHECK (prezzoPar > 0),
    /*sezione costraint*/
    CONSTRAINT fk_ordine_in_compOrdine      FOREIGN KEY (idordine)
    REFERENCES ordini(idordine)     ON DELETE CASCADE ON UPDATE CASCADE,

    CONSTRAINT fk_articolo_in_compOrdine    FOREIGN KEY (idarticolo)
    REFERENCES articoli(idarticolo) ON DELETE CASCADE ON UPDATE CASCADE,

    CONSTRAINT unique_compOrdine    UNIQUE (idordine, idarticolo)
);
```

4.2 | Trigger e procedure

Qui, di seguito, un esempio di come creare una **sequenza** per gestire la PK di una tabella. Questo esempio è la base anche per le PK delle tabelle Operatori e Corrieri che, pertanto, evitiamo di scrivere un'altra volta; risulterebbe sufficiente, infatti, cambiare i nomi di sequenza, funzione e trigger.

```
-- Creazione della sequenza per autoincrement PK(cliente)
CREATE SEQUENCE idcliente_seq
    START 1
    INCREMENT 1;

-- Funzione per ottenere un nuovo valore dalla sequenza
CREATE OR REPLACE FUNCTION nextval_idcliente()
RETURNS INTEGER AS $$
DECLARE
    next_id INTEGER;
BEGIN
    SELECT nextval('idcliente_seq') INTO next_id;

    RETURN next_id;
END;
$$ LANGUAGE plpgsql;

-- Trigger per utilizzare la sequenza prima dell'inserimento nella tabella
CREATE OR REPLACE FUNCTION before_insert_cliente()
RETURNS TRIGGER AS $$
BEGIN
    NEW.idcliente = nextval_idcliente();

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER clienti_insert
BEFORE INSERT ON clienti
FOR EACH ROW
EXECUTE FUNCTION before_insert_cliente();
```

Per quanto riguarda le restanti tabelle, la cui PK si è deciso essere di tipo string, si è preferito optare per un'operazione leggermente diversa, che prevede un prefisso ad una stringa di cifre. Di seguito l'esempio per la tabella Articoli:

```
-- Creazione della sequenza per PK(articoli)
CREATE SEQUENCE idarticolo_seq
    START 1000
    INCREMENT 1;

-- Funzione per ottenere un nuovo valore dalla sequenza
CREATE OR REPLACE FUNCTION nextval_idarticolo()
RETURNS VARCHAR(15) AS $$
DECLARE
    next_id INTEGER;
    result_id VARCHAR(15);
BEGIN
    SELECT nextval('idarticolo_seq') INTO next_id;
     --concateno A col prossimo valore della sequenza
    result_id := 'A' || next_id::VARCHAR;

    RETURN result_id;
END;
$$ LANGUAGE plpgsql;

-- Trigger per utilizzare la sequenza prima dell'inserimento nella tabella
CREATE OR REPLACE FUNCTION before_insert_articolo()
RETURNS TRIGGER AS $$
BEGIN
    NEW.idarticolo = nextval_idarticolo();

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER articoli_insert
BEFORE INSERT ON articoli
FOR EACH ROW
EXECUTE FUNCTION before_insert_articolo();
```

Questo, invece, è un **trigger** utile a crittografare le password, ma è alquanto banale: non fa altro che sfruttare il cifrario di Cesare. Si tratta di un algoritmo ben lontano da quelli realmente usati dalle aziende di big data, ma lo usiamo per un semplice fine teorico.

```
-- TRIGGER PER CIFRARE LA PASSWORD DI OPERATORE --
CREATE OR REPLACE FUNCTION cesare_plus_uno()
RETURNS TRIGGER AS $$
BEGIN
    NEW.pass := calcola_cesare_plus_uno(NEW.pass);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION calcola_cesare_plus_uno(pass VARCHAR(50))
RETURNS VARCHAR(50) AS $$
DECLARE
    result VARCHAR(50) := '';
    char_code INTEGER;
BEGIN
    FOR i IN 1..LENGTH(pass) LOOP
        char_code := ASCII(SUBSTRING(pass FROM i FOR 1));
        -- Controlla se il carattere e' una lettera dell'alfabeto
        IF char_code BETWEEN ASCII('a') AND ASCII('z') THEN
            char_code := ((char_code - ASCII('a') + 1) % 26) + ASCII('a');
        ELSIF char_code BETWEEN ASCII('A') AND ASCII('Z') THEN
            char_code := ((char_code - ASCII('A') + 1) % 26) + ASCII('A');
        ELSIF char_code BETWEEN ASCII('0') AND ASCII('9') THEN
            char_code := ((char_code - ASCII('0') + 1) % 10) + ASCII('0');
        END IF;

        result := result || CHR(char_code);
    END LOOP;

    RETURN result;
END;
$$ LANGUAGE plpgsql;

-- Crea il trigger
CREATE TRIGGER before_insert_pass
BEFORE INSERT ON operatori
FOR EACH ROW
EXECUTE FUNCTION cesare_plus_uno();
```

Trattasi questo di un **trigger** per assicurarsi che uno dei vincoli dovuti alla ristrutturazione venga rispettato. I restanti, molto simili a questo, non verranno riproposti in questo file.

```
-- CodFisc di operatore diverso da quelli di corriere
CREATE OR REPLACE FUNCTION is_operatore_valid()
RETURNS TRIGGER AS $$
DECLARE
    num_codfisc_uguali INTEGER := 0;
BEGIN
    -- Conto le righe di corrieri che hanno lo stesso codFiscale
    -- dell'operatore inserito. Mi restituisce al piu' una riga
    SELECT COUNT(*)
    INTO num_codfisc_uguali
    FROM corrieri AS C
    WHERE C.codFiscale = NEW.codFiscale;

    IF num_codfisc_uguali = 1 THEN
        RAISE EXCEPTION 'L''operatore non puo'' essere anche un '
        'corriere, pertanto i codici fiscali devono essere diversi.';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Controllo necessario sia al primo inserimento...
CREATE TRIGGER check_codfiscale_operatori
BEFORE INSERT ON operatori
FOR EACH ROW
EXECUTE FUNCTION is_operatore_valid();

-- ...sia in caso di modifica del dato
CREATE TRIGGER check_update_codfiscale_operatori
BEFORE UPDATE ON operatori
FOR EACH ROW
WHEN (OLD.codFiscale <> NEW.codFiscale)
EXECUTE FUNCTION is_operatore_valid();
```

Il prossimo è un **trigger** necessario affinché quello successivo funzioni correttamente in ogni caso. Si occupa di assicurarsi che una spedizione venga prima ideata, e solo dopo avvenga effettivamente; sono contemplati, però, anche i casi di una spedizione avvenuta in passato.

```
CREATE OR REPLACE FUNCTION is_sped_valid()
RETURNS TRIGGER AS $$
BEGIN
    NEW.avviata := FALSE;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER ricalibra_sped
BEFORE INSERT ON spedizioni
FOR EACH ROW
WHEN ((NEW.avviata = TRUE) AND (NEW.completata = FALSE))
EXECUTE FUNCTION is_sped_valid();
```

Iniziamo a vedere una **funzione** che useremo nel trigger successivo:

```
-- Peso totale delle merci che compongono un ordine
CREATE OR REPLACE FUNCTION calcola_peso_ordine(idordineIN VARCHAR(15))
RETURNS DECIMAL(6,2) AS $$
DECLARE
    peso_par CURSOR FOR
        SELECT pesoPar
        FROM compOrdine
        WHERE idordine = idordineIN;

    exists_compOrdine INTEGER := 0;
    peso_ordine DECIMAL(6,2) := 0;
BEGIN
    -- Conto per verificare che esista almeno una riga in
    -- compOrdine associata
    SELECT COUNT(*)
    INTO exists_compOrdine
    FROM compOrdine
    WHERE idordine = idordineIN;

    -- Se l'ordine non prevede alcun articolo
    IF exists_compOrdine = 0 THEN
        RAISE EXCEPTION 'L''ordine % non prevede l''acquisto di alcun '
        'articolo. Si prega di aggiornarlo o eliminarlo.', idordineIN;
    END IF;

    -- Incremento man mano il peso dell'ordine
    FOR riga IN peso_par
    LOOP
        peso_ordine = peso_ordine + riga.pesoPar;
    END LOOP;

    RETURN peso_ordine;
END;
$$ LANGUAGE plpgsql;
```

Ed ecco un'altra **funzione**, ancora una volta utile al trigger:

```
-- Peso totale della merce degli ordini che compongono una spedizione
CREATE OR REPLACE FUNCTION calcola_peso_sped(idspedizioneIN VARCHAR(15))
RETURNS DECIMAL(6,2) AS $$
DECLARE
    codici_ordini CURSOR FOR
        SELECT idordine
        FROM ordini
        WHERE idspedizione = idspedizioneIN;

    pesoTot DECIMAL(6,2) := 0;
    exists_ordine INTEGER := 0;
BEGIN
    -- Conto per verificare che esista almeno una riga in Ordini associata
    SELECT COUNT(*)
    INTO exists_ordine
    FROM ordini
    WHERE idspedizione = idspedizioneIN;

    -- Se la spedizione non prevede di consegnare nulla
    IF exists_ordine = 0 THEN
        RAISE EXCEPTION 'La spedizione non prevede di consegnare nulla. Si'
        ' prega di aggiornare prima quali ordini avranno tale spedizione.';
    END IF;

    -- Ottenuto il peso di ogni ordine, procedo a sommarlo nel peso totale
    FOR riga_cod IN codici_ordini
    LOOP
        pesoTot = pesoTot + calcola_peso_ordine(riga_cod.idordine);
    END LOOP;

    RETURN pesoTot;
END;
$$ LANGUAGE plpgsql;
```

Infine il **trigger**, che ha premura di controllare se il corriere inserito è disponibile e se il veicolo è non solo anch'esso disponibile, ma anche adeguato al trasporto.

```
-- Pianifica spedizione 1-2, corriere e veicolo disponibilita' 1
CREATE OR REPLACE FUNCTION are_corriere_veicolo_validi()
RETURNS TRIGGER AS $$
DECLARE
    disp_corriere BOOLEAN;
    disp_veicolo BOOLEAN;
    peso_da_sostenere DECIMAL(6,2) := 0;
    capacita_di_peso DECIMAL(6,2);
BEGIN
    -- Seleziono e salvo la disponibilita' del corriere incaricato
    SELECT disponibilita
    INTO disp_corriere
    FROM corrieri
    WHERE idcorriere = NEW.idcorriere;

    -- Seleziono la disponibilita' e peso supportato del veicolo adoperato
    SELECT disponibilita, pesoSupportato
    INTO disp_veicolo, capacita_di_peso
    FROM veicoli
    WHERE targa = NEW.targa;

    peso_da_sostenere = calcola_peso_sped(NEW.idspedizione);

    -- Controlla se il corriere non e' disponibile
    IF disp_corriere = FALSE THEN
        RAISE EXCEPTION 'Il corriere deve risultare disponibile per poterlo'
        ' incaricare della spedizione.';
    -- Controlla se il veicolo non e' disponibile
    ELSIF disp_veicolo = FALSE THEN
        RAISE EXCEPTION 'Il veicolo deve risultare disponibile per poterlo'
        ' adoperare nella spedizione.';
    -- Controlla se il veicolo regge il peso della merce
    ELSIF capacita_di_peso < peso_da_sostenere THEN
        RAISE EXCEPTION 'Si prega di selezionare un veicolo adatto al peso'
        ' della spedizione, oppure di ripartire diversamente gli ordini.';
    END IF;
    -- Se tutto e' ok, aggiorno i dati
    NEW.data_sped := CURRENT_DATE;
    NEW.orario := CURRENT_TIME;

    UPDATE corrieri
    SET disponibilita = FALSE
    WHERE idcorriere = NEW.idcorriere;

    UPDATE veicoli
    SET disponibilita = FALSE
    WHERE targa = NEW.targa;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER pianifica_sped
BEFORE UPDATE ON spedizioni
FOR EACH ROW
WHEN ((OLD.avviata = FALSE) AND (NEW.avviata = TRUE))
EXECUTE FUNCTION are_corriere_veicolo_validi();
```

Ora è necessario considerare i casi in cui una spedizione inserita venga cancellata per qualche motivo oppure torni ad essere avviata = false:

```
-- Consequenza del precedente trigger, aggiorna disponibilita'
-- di corriere e veicolo precedentemente coinvolti
CREATE OR REPLACE FUNCTION update_disponibilita_true()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE corrieri
    SET disponibilita = TRUE
    WHERE idcorriere = OLD.idcorriere;

    UPDATE veicoli
    SET disponibilita = TRUE
    WHERE targa = OLD.targa;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER aggiorna_disponibilita_sped_cancellata
BEFORE DELETE ON spedizioni
FOR EACH ROW
WHEN ((OLD.avviata = TRUE) AND (OLD.completata = FALSE))
EXECUTE FUNCTION update_disponibilita_true();

CREATE TRIGGER aggiorna_disponibilita_sped_cambiata
BEFORE UPDATE ON spedizioni
FOR EACH ROW
WHEN (((OLD.avviata = TRUE) AND (OLD.completata = FALSE))
                                AND (NEW.avviata = FALSE))
EXECUTE FUNCTION update_disponibilita_true();
```

Il seguente **trigger** si occupa di vedere se un ordine può o meno essere parte di una spedizione:

```
-- Componi spedizione
CREATE OR REPLACE FUNCTION is_possible_ordine_in_spedizione()
RETURNS TRIGGER AS $$
DECLARE
    sped_avviata BOOLEAN;
    sped_completata BOOLEAN;
BEGIN
    -- Seleziono e salvo gli attributi che descrivono se la spedizione
    -- e' partita (ed eventualmente finita)
    SELECT avviata, completata
    INTO sped_avviata, sped_completata
    FROM spedizioni
    WHERE idspedizione = NEW.idspedizione;

    -- Se si tratta di una spedizione che sta avvenendo
    IF sped_avviata = TRUE AND sped_completata = FALSE THEN
        RAISE EXCEPTION 'Non e'' possibile aggiungere un ordine '
                        'ad una spedizione gia'' avviata.';
    END IF;
    /* Nota: nel caso di sped_completata = TRUE, e' possibile inserire
        degli ordini con il riferimento a tale spedizione per permettere
        all'utente di inserire, qualora volesse, spedizioni che si sono
        verificate in passato, senza pero' perdere il riferimento negli
        ordini. (prima creo tupla in spedizioni, poi aggiorno gli ordini
        con il giusto idspedizione).                                    */

    -- Se tutto e' ok, permetto il nuovo idspedizione
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER componi_sped
BEFORE UPDATE ON ordini
FOR EACH ROW
WHEN ((OLD.idspedizione IS NULL) OR (OLD.idspedizione <> NEW.idspedizione))
EXECUTE FUNCTION is_possible_ordine_in_spedizione();
```

Simile al caso precedente, stavolta vogliamo verificare che un ordine possa essere composto da articoli presenti nella giusta quantità, dopodiché si aggiornano le scorte in magazzino:

```
-- Componi ordine e aggiorna scorte e ordine (prezzoTot, pesoPar, prezzoPar)
CREATE OR REPLACE FUNCTION is_possible_articolo_in_ordine()
RETURNS TRIGGER AS $$
DECLARE
    quantita_in_magazzino INTEGER;
    peso_articolo DECIMAL(5,2);
    prezzo_articolo DECIMAL(8,2);
BEGIN
   -- Seleziono e salvo la quantita' di uno specifico articolo in magazzino
    SELECT quantita, peso, prezzo
    INTO quantita_in_magazzino, peso_articolo, prezzo_articolo
    FROM articoli
    WHERE idarticolo = NEW.idarticolo;

    -- Se la quantita' dell'ordine e' maggiore di quella in magazzino
    IF quantita_in_magazzino < NEW.quantita THEN
        RAISE EXCEPTION 'Non e'' possibile ultimare l''ordine;'
        ' articolo non disponibile al momento nelle quantita'' richieste.';
    END IF;

    -- Se tutto e' ok, permetto l'inserimento in compOrdine,
    -- ma non prima di aggiornare le scorte in magazzino
    UPDATE articoli
    SET quantita = quantita - NEW.quantita
    WHERE idarticolo = NEW.idarticolo;

    -- Devo aggiornare anche gli attributi derivabili
    NEW.pesoPar := NEW.quantita * peso_articolo;
    NEW.prezzoPar := NEW.quantita * prezzo_articolo;

    UPDATE ordini
    SET prezzoTot = prezzoTot + NEW.prezzoPar
    WHERE idordine = NEW.idordine;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER componi_ordine_poi_aggiorna
BEFORE INSERT ON CompOrdine
FOR EACH ROW
EXECUTE FUNCTION is_possible_articolo_in_ordine();

-- Non prevedo la possibilita' di update in compOrdine perche' non
-- rientra nei poteri dell'utente
```

Nel caso in cui una componente dell'ordine venga cancellata perché inserita per errore magari, va fatto il ragionamento opposto, e bisogna incrementare le scorte precedentemente diminuite, oltre ad aggiornare il prezzoTot dell'ordine in questione:

```
-- Aggiorna scorte se compOrdine cancellato
CREATE OR REPLACE FUNCTION update_scorte()
RETURNS TRIGGER AS $$
DECLARE
    sped_avviata BOOLEAN;
BEGIN
    -- Seleziono e salvo se l'ordine fa parte di una spedizione gia' avviata
    SELECT avviata
    INTO sped_avviata
    FROM spedizioni NATURAL JOIN ordini
    WHERE idordine = NEW.idordine;

    IF sped_avviata = TRUE THEN
        RAISE EXCEPTION 'Non e'' possibile cancellare la composizione'
                ' di un ordine gia'' in spedizione.';
    END IF;

    UPDATE articoli
    SET quantita = quantita + OLD.quantita
    WHERE idarticolo = OLD.idarticolo;

    -- Sottraggo una componente che prima era stata sommata,
    -- ergo non ho problemi di dominio su prezzoTot
    UPDATE ordini
    SET prezzoTot = prezzoTot - OLD.prezzoPar
    WHERE idordine = OLD.idordine;

    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER aggiorna_scorte_non_piu_in_ordine
BEFORE DELETE ON CompOrdine
FOR EACH ROW
EXECUTE FUNCTION update_scorte();
```

Gli ordini possono risultare consegnati solamente se fanno parte di una spedizione e quest'ultima è stata avviata. Vediamo come poter implementare tale vincolo:

```
-- Quando gli ordini possono risultare consegnati?
-- Quando fanno parte di una spedizione ed essa e' stata avviata
CREATE OR REPLACE FUNCTION is_possible_ordine_consegnato()
RETURNS TRIGGER AS $$
DECLARE
    spedizione_avviata BOOLEAN;
BEGIN
    IF NEW.idspedizione IS NULL THEN
        RAISE EXCEPTION 'L''ordine non e'' associato a nessuna spedizione;'
                        ' pertanto, non puo'' risultare consegnato.';
    END IF;

    SELECT avviata
    INTO spedizione_avviata
    FROM spedizioni
    WHERE idspedizione = NEW.idspedizione;

    IF spedizione_avviata = FALSE THEN
        RAISE EXCEPTION 'L''ordine e'' associato ad una spedizione che'
        ' risulta non essere ancora avvenuta, quindi non puo'' essere'
        ' stato consegnato.';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Possibile ordine consegnato
CREATE TRIGGER update_ordine_consegnabile
BEFORE UPDATE ON ordini
FOR EACH ROW
WHEN (NEW.consegnato = TRUE)
EXECUTE FUNCTION is_possible_ordine_consegnato();

CREATE TRIGGER ordine_consegnabile_con_insert
BEFORE INSERT ON ordini
FOR EACH ROW
WHEN (NEW.consegnato = TRUE)
EXECUTE FUNCTION is_possible_ordine_consegnato();
```

Una spedizione risulta completata se e solo se tutti gli ordini sono stati consegnati:

```
-- Spedizione completata 1 and Corriere e veicolo disponibilita' 2
CREATE OR REPLACE FUNCTION update_ordini_consegnato()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE ordini
    SET consegnato = TRUE
    WHERE idspedizione = NEW.idspedizione;

    UPDATE corrieri
    SET disponibilita = TRUE
    WHERE idcorriere = OLD.idcorriere;

    UPDATE veicoli
    SET disponibilita = TRUE
    WHERE targa = OLD.targa;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER spedizione_completata_then_ordini_consegnati
AFTER UPDATE ON spedizioni
FOR EACH ROW
WHEN (NEW.completata = TRUE)
EXECUTE FUNCTION update_ordini_consegnato();
```

Implicazione da destra verso sinistra dello stesso vincolo di pagina precedente:

```
-- Spedizione completata 2
CREATE OR REPLACE FUNCTION update_sped_completata()
RETURNS TRIGGER AS $$
DECLARE
    ordini_consegnato CURSOR FOR
        SELECT consegnato
        FROM ordini
        WHERE idspedizione = NEW.idspedizione;

    ordini_non_consegnati INTEGER := 0;
    sped_completata BOOLEAN;
BEGIN
    SELECT completata
    INTO sped_completata
    FROM spedizioni
    WHERE idspedizione = NEW.idspedizione;

    -- Lo stack esplode causa loop se non eseguo questo check
    IF sped_completata <> TRUE THEN

        FOR riga IN ordini_consegnato
        LOOP
            IF riga.consegnato = FALSE THEN
                ordini_non_consegnati = ordini_non_consegnati + 1;
            END IF;
        END LOOP;

        IF ordini_non_consegnati = 0 THEN
            UPDATE spedizioni
            SET completata = TRUE
            WHERE idspedizione = NEW.idspedizione;
        END IF;

    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER ordini_consegnati_then_sped_completata
AFTER UPDATE ON ordini
FOR EACH ROW
WHEN (NEW.consegnato = TRUE)
EXECUTE FUNCTION update_sped_completata();
```