
Progettazione Orientata agli Oggetti

Sviluppo del software

- Ciclo di vita del software: comprende tutte le attività dall'analisi iniziale fino all'obsolescenza (sviluppo, aggiornamento, manutenzione)
- Procedimento formale per lo sviluppo del software
 - Descrive le varie fasi da compiere
 - Fornisce le linee guida su come eseguire le fasi
 - Dipartimento della Difesa USA tra i primi a proporlo
- Fasi principali del processo di sviluppo
 - Analisi
 - Progettazione
 - Implementazione
 - Collaudo
 - Installazione

Analisi

- Stabilire **cosa** deve fare il programma finale e **non come** lo deve fare
 - Risultato: documento dei requisiti
 - Descrive cosa il programma sarà in grado di fare quando verrà terminato
 - Manuale utente: descrive come l'utente utilizzerà il programma
 - Criteri per la valutazione delle prestazioni:
 - quanti dati in ingresso il programma deve essere in grado di gestire e in quali tempi
 - il fabbisogno di memoria e spazio su disco
 - etc.
-

Progettazione

- Pianificazione di **come** implementare il sistema
 - Scoprire le strutture per la soluzione del problema
 - Nella progettazione object-oriented
 - è incentrata sulla determinazione delle classi e dei metodi
 - ha come risultato
 - descrizione delle classi e dei metodi
 - determinazione delle relazioni tra classi
-

Implementazione, Collaudo e Installazione

■ Implementazione

- ❑ Scrivere e compilare il codice
- ❑ Il codice implementa le classi e i metodi scoperti nella fase di progettazione
- ❑ Risultato: il programma

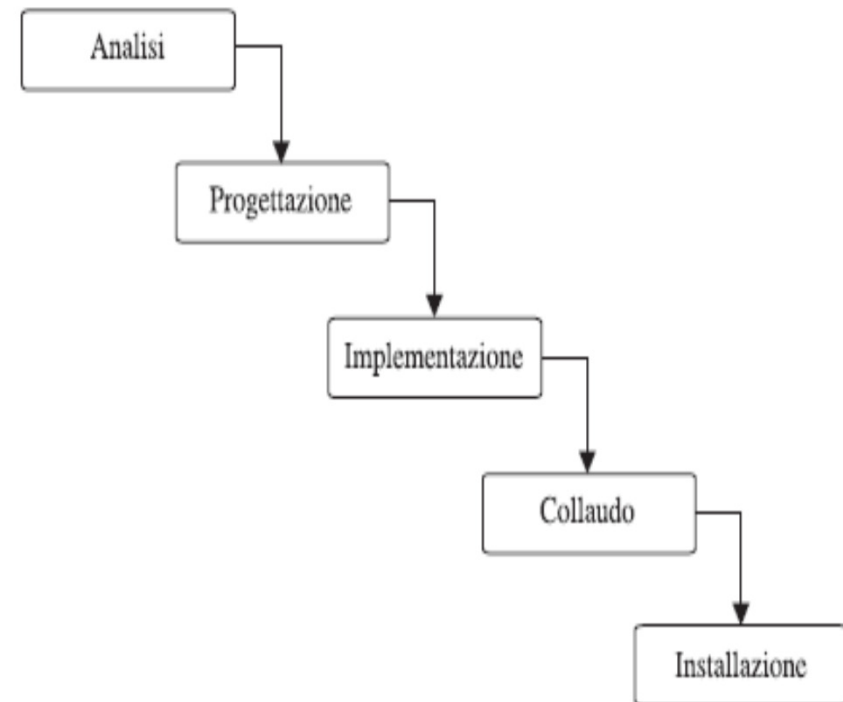
■ Collaudo

- ❑ Verificare che il programma funziona correttamente
- ❑ Risultato: un documento riassuntivo delle operazioni di verifica eseguite e dei relativi risultati

■ Installazione

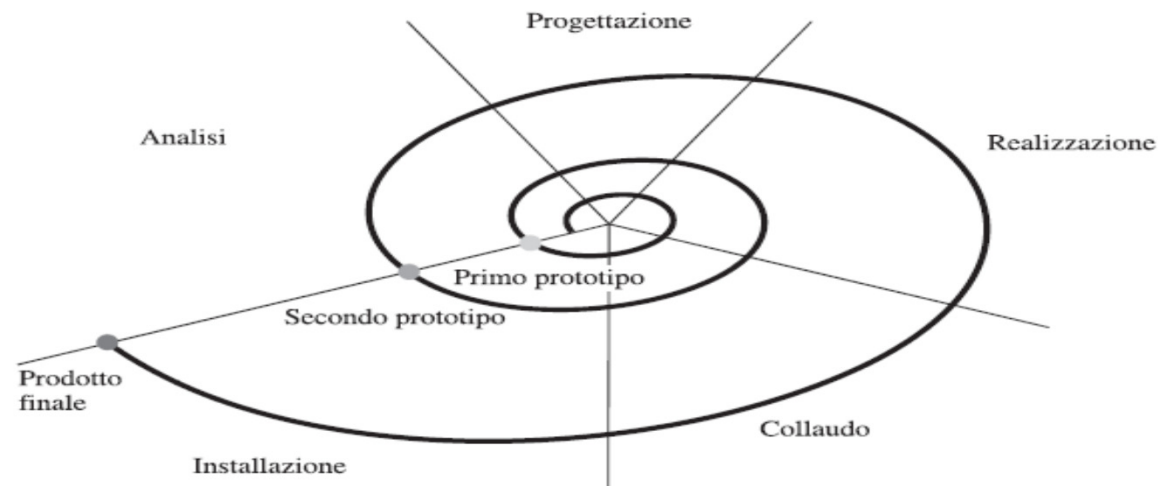
Progettazione formale: modello a cascata (anni settanta)

- Metodo sequenziale di esecuzione delle fasi della progettazione formale
- Se applicato in maniera rigida, questo modello non produce risultati soddisfacenti
 - Spesso è utile ripetere le varie fasi iterativamente raffinando progressivamente il progetto



Il modello a spirale (1988)

- Suddivide il processo di sviluppo in fasi multiple
- Le fasi iniziali si concentrano sulla realizzazione di prototipi del sistema
 - Quello che si impara su un prototipo si usa sul prototipo successivo
- Problema: numero di iterazioni elevato, il processo può richiedere molto tempo per essere completato



Procedimento Razionale Unificato (PRU)

- Procedimento in quattro fasi proposto da sviluppatori UML
- Fase iniziale: (delineare *business case*)
 - comprendere il tipo di mercato di riferimento
 - identificare gli elementi importanti per successo commerciale
- Elaborazione: struttura complessiva del sistema
- Costruzione:
 - maggior parte degli sviluppi
 - produzione prima release del sistema.
- Transizione: (passaggio da ambiente sviluppo a cliente)
 - attività di training degli utenti
 - beta testing del sistema per ulteriore verifica e validazione (se prodotto non conforme ai requisiti individuati nella fase iniziale si ripete l'intero ciclo)

Programmazione Estremizzata (1999)

- Insegue la semplicità
 - Rimuove strutture formali
 - Si concentra sulle migliori regole pratiche
 - Pianificazione realistica
 - Piccoli avanzamenti
 - Metafora
 - Semplicità
 - Collaudo
 - Riprogettazione
 - Coppie programmatori
 - Proprietà collettiva
 - Integrazione continua
 - 40 ore settimanali
 - Clienti a disposizione
 - Standard di codifica
-

Regole programmazione estremizzata

- Pianificazione realistica
 - i clienti prendono le decisioni commerciali
 - i programmatori prendono le decisioni tecniche
 - aggiornare il progetto quando non è realistico
 - Piccoli avanzamenti
 - rilasciare un sistema utile prima possibile
 - rilasciare aggiornamenti di frequente
 - Metafora
 - i programmatori condividono una semplice descrizione (*metafora*) dell'intero sistema
-

Regole programmazione estremizzata

- Semplicità
 - progettare le cose nella maniera più semplice possibile (invece di predisporre il tutto per future complessità)
 - Collaudo
 - programmatori e clienti collaborano alla stesura delle verifiche di collaudo da eseguire
 - eseguire verifiche di continuo
 - Riprogettazione
 - riprogettare il sistema di continuo per migliorare il codice ed eliminare parti duplicate
-

Regole programmazione estremizzata

- Programmazione a coppie
 - i programmatori lavorano a coppie e ogni coppia è assegnata ad una postazione per scrivere codice
 - Proprietà collettiva
 - ogni programmatore ha accesso a tutto il codice e può modificarlo quando necessario
 - Integrazione continua
 - non appena un problema è risolto, rimettere insieme l'intero sistema e procedere alla verifica
-

Regole programmazione estremizzata

- 40-ore per settimana
 - non fare piani di lavoro irrealistici riempiendoli di sforzi eroici
 - Disponibilità cliente
 - un rappresentante del cliente deve essere sempre raggiungibile per il team di progettazione
 - Standard di codifica
 - i programmatori devono seguire gli standard di programmazione che enfatizzano la realizzazione di codice auto-documentato
-

Progettazione orientata agli oggetti

1. Individuare le classi
 2. Determinare le responsabilità di ogni classe
 3. Descrivere le relazioni tra le classi individuate
-

Come determinare le classi

- Una classe rappresenta un concetto utile
 - Entità concrete: conti bancari, ellissi, prodotti,...
 - Concetti astratti: flussi, dataSet,...
 - Definire il comportamento di ogni classe
 - Suggerimento:
 - le classi si possono individuare guardando ai nomi usati nella descrizione del lavoro da eseguire
 - i metodi si possono individuare guardando ai verbi utilizzati nella descrizione del lavoro da eseguire
-

Esempio: Fattura (Invoice)

F A T T U R A

Piccoli Elettrodomestici Aldo
via Nuova, 100
Turbigo, MI 20029

```
=====
```

Articolo	Q.tà	Prezzo	Totale
Tostapane	3	€ 29,95	€ 89,85
Asciugacapelli	1	€ 24,95	€ 24,95
Spazzola elettrica	2	€ 19,99	€ 39,98

```
=====
```

IMPORTO DOVUTO: € 154,78

Esempio: Invoice

- **Possibili classi:** Fattura, Articolo, e Cliente
 - Una buona idea è mantenere una lista di candidati da raffinare progressivamente
 - Semplicemente si mettono tutti i nomi di una possibile classe in una lista
 - Come ci si accorge che un elemento della lista non serve o non è adeguato si cancella dalla lista
-

Determinazione delle classi

- Le classi rappresentano insiemi di oggetti con lo stesso comportamento
 - Entità con occorrenze multiple nella descrizione di un problema sono buoni candidati di oggetti
 - Determinate cosa hanno in comune
 - Progettate la classe in modo da catturare gli aspetti comuni
- Rappresentare alcune entità come oggetti, altre come dati primitivi
 - Creiamo una classe Indirizzo o usiamo un oggetto String?
Risposta dipende dall'uso che ne faremo
- Alcune classi possono non essere scoperte nella fase di analisi
- Alcune classi possono essere state già realizzate

Determinazione dei metodi

- Guardare ai verbi nella descrizione dei compiti da eseguire
 - Bisogna stabilire per ogni metodo necessario la classe in cui collocarlo
 - Un metodo utile per svolgere questo compito consiste nel compilare le schede CRC
-

Schede CRC

- CRC: Classe, Responsabilità, Collaboratori
Descrive una classe, le sue responsabilità e i suoi collaboratori
 - Usare una scheda per ogni classe
 - Per ogni metodo (azione, compito), individuare una classe responsabile
 - annotare la responsabilità sulla scheda (può essere implementata attraverso più di un metodo)
 - annotare le altre classi necessarie per assolvere la responsabilità (collaboratori)
-

Scelta responsabilità soddisfacente?

- Per ciascun compito del software in progettazione, chiedetevi come potrebbe essere materialmente svolta utilizzando soltanto le responsabilità che avete scritto sulle varie schede
 - Le responsabilità elencate sulla scheda sono *ad alto livello*:
 - a volte una singola responsabilità dovrà essere realizzata con due o più metodi Java
 - Alcuni ricercatori sostengono che una scheda CRC non dovrebbe avere più di tre diverse responsabilità.
-

Relazioni tra classi

- Ereditarietà
 - Associazione/Aggregazione
 - Dipendenza
-

Ereditarietà

- Relazione del tipo “*è un*”
- Relazione tra una classe generale (*superclasse*) e una classe specializzata (*sottoclasse*)
- Esempi:
 - Ogni conto di deposito è un conto bancario
 - Ogni cerchio è un ellisse (con ampiezza e altezza uguali)
- Qualche volta questa relazione viene abusata
 - La classe `Pneumatico` deve essere una sottoclasse della classe `Cerchio`?
 - In questo caso un altro tipo di relazione (*ha un*) sarebbe più appropriata

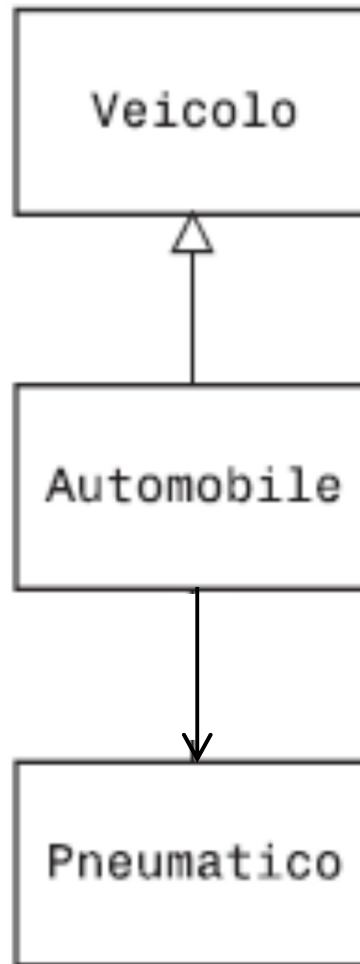
Associazione/Aggregazione

- Relazione del tipo “*ha un*”
- Gli oggetti di una classe contengono riferimenti ad oggetti di un'altra classe (nelle variabili di istanza)
 - Uno pneumatico *ha un* cerchio come sua frontiera:
 - Ogni auto *ha uno* pneumatico (4 più uno di scorta)

```
class Tyre
{
    . . .
    private String modello;
    private Circle frontiera;
}
```

```
class Car extends Vehicle
{
    . . .
    private Tyre[] tyres;
}
```

Esempio: Notazione UML



Dipendenza

- Relazione “*usa*”
 - Es.: molte delle applicazioni che abbiamo visto dipendono da Scanner per l’input
 - Associazione è una forma più forte di dipendenza
 - Per avere dipendenza basta passare un oggetto come parametro di un metodo
 - Per avere associazione occorre che l’oggetto sia referenziato da una variabile d’istanza
-

Notazione UML

Relazione	Simbolo	Tratto	Punta della freccia
Ereditarietà		Continuo	Triangolare
Implementazione di interfaccia		Tratteggio	Triangolare
Aggregazione		Continuo	Aperta
Dipendenza		Tratteggio	Aperta

Metodi e attributi nei diagrammi UML

Per attributo si intende una proprietà dell'oggetto osservabile dall'esterno (non necessariamente una variabile di istanza)



Procedura di sviluppo programma

1. Stilare l'elenco dei requisiti
 2. Utilizzare le schede CRC per classi, responsabilità e collaboratori
 3. Utilizzare diagrammi UML per registrare le relazioni tra classi
 4. Utilizzare `javadoc` per documentare il comportamento dei metodi
 5. Implementare il programma
-

Stampa fattura – Requisiti

- Problema: stampare una fattura
- Fattura (Invoice): descrive i costi per un insieme di prodotti ognuno in una quantità specifica
- Omettiamo alcuni dettagli
 - Date, tasse, e numero di fattura
- Vogliamo sulla fattura
 - Indirizzo del cliente, gli articoli venduti, l'importo dovuto
- Per ogni articolo vogliamo
 - Descrizione, prezzo unitario, quantità ordinata, prezzo totale
- Per semplicità non consideriamo un'interfaccia utente
- Programma di collaudo: aggiunge articoli ad una fattura e la stampa

Esempio di fattura

I N V O I C E

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Description	Price	Qty	Total
Toaster	29.95	3	89.85
Hair dryer	24.95	1	24.95
Car vacuum	19.99	2	39.98

Amount Due: \$154.78

Scelta classi

- I nomi della descrizione sono classi potenziali

```
Invoice
Address
LineItem // Mantiene il prodotto e la quantità
Product
Description // Campo della classe Product
Price // Campo della classe Product
Quantity // Non è un attributo di Product
Total // Viene calcolato
Amount Due // Viene calcolato
```

- Rimangono

```
Invoice
Address
LineItem
Product
```

Schede CRC

- Aggiungere i collaboratori a `Invoice`:

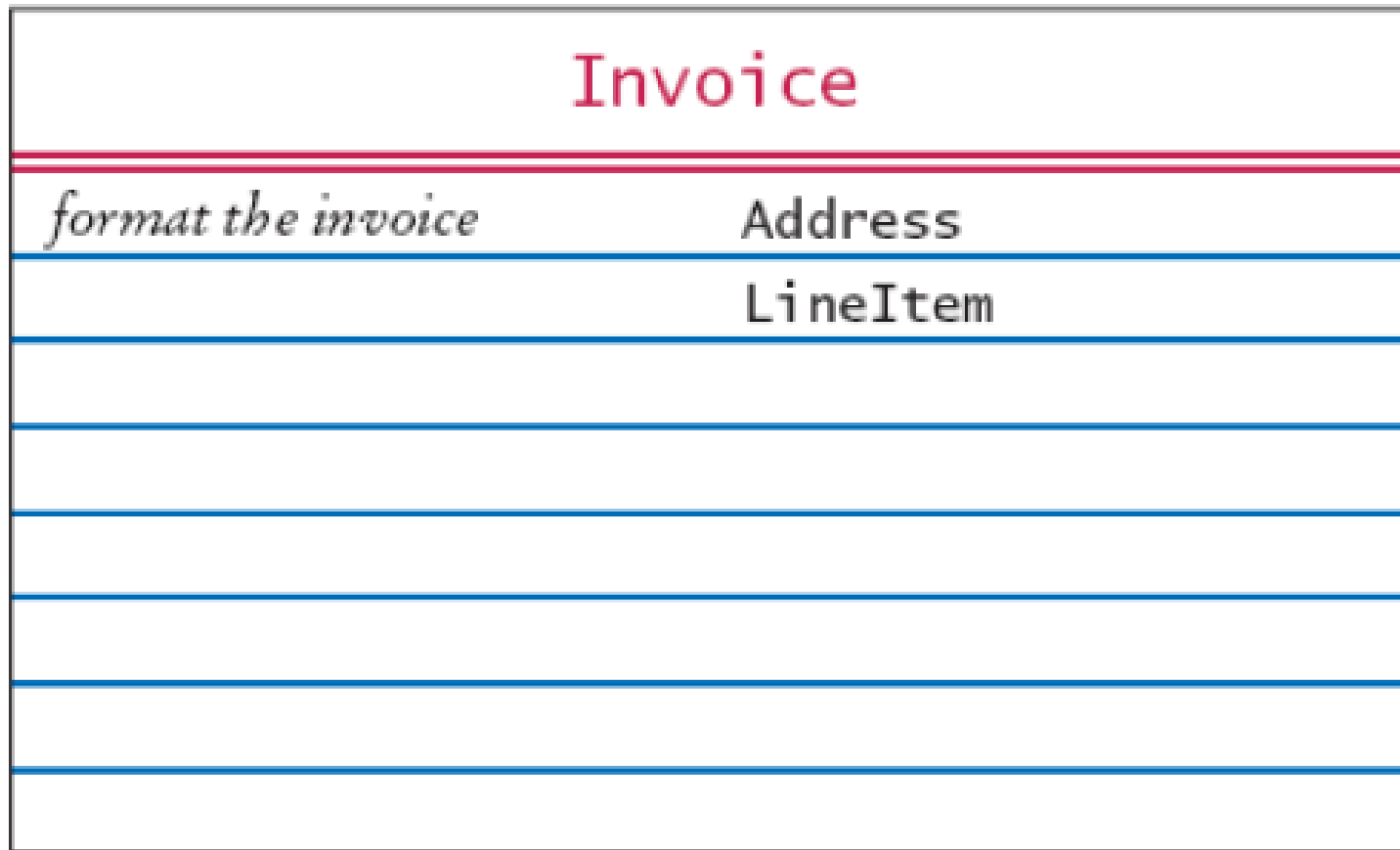
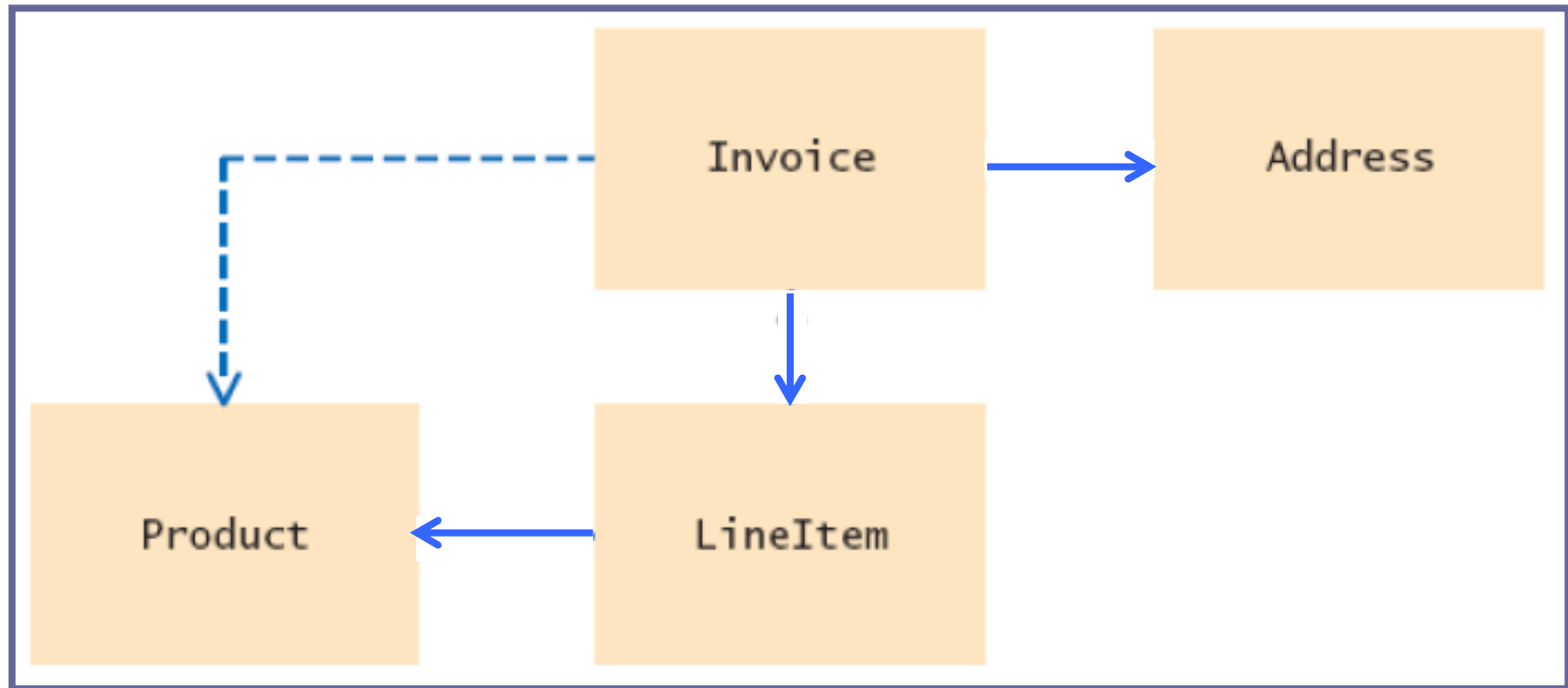


Diagramma UML



Documentazione

- Si può usare `javadoc` per generare la documentazione
 - In prima istanza si può lasciare il corpo dei metodi vuoto
 - Lanciando `javadoc` si ottiene una documentazione in formato HTML
 - Vantaggi:
 - Condividere documentazione HTML con gli altri membri del team
 - Le classi Java sono già predisposte
 - Si danno i commenti dei metodi chiave
-

Documentazione per Stampa fattura

```
/**
 * Describes an invoice for a set of purchased products.
 */
public class Invoice
{
    /**
     * Adds a charge for a product to this invoice.
     * @param aProduct the product that the customer ordered
     * @param quantity the quantity of the product
     */
    public void add(Product aProduct, int quantity)
    {
    }

    /**
     * Formats the invoice.
     * @return the formatted invoice
     */
    public String format()
    {
    }
}
```

Documentazione: LineItem

```
/**
 * Describes a quantity of an article to purchase and its price.
 */
public class LineItem
{
    /**
     * Computes the total cost of this line item.
     * @return the total price
     */
    public double getTotalPrice()
    {
    }

    /**
     * Formats this item.
     * @return a formatted string of this line item
     */
    public String format()
    {
    }
}
```

Documentazione: Product

```
/**
 * Describes a product with a description and a price.
 */
public class Product
{
    /**
     * Gets the product description.
     * @return the description
     */
    public String getDescription()
    {
    }

    /**
     * Gets the product price.
     * @return the unit price
     */
    public double getPrice()
    {
    }
}
```

Documentazione: Address

```
/**
 * Describes a mailing address.
 */
public class Address
{
    /**
     * Formats the address.
     * @return the address as a string with three lines
     */
    public String format()
    {
    }
}
```

Implementazione

- I diagrammi UML danno le variabili di istanza
- Alcune variabili sono date attraverso le associazioni
- Invoice **associa** Address e LineItem
 - Ogni fattura ha un indirizzo a cui inviare la fattura
 - Una fattura può avere diversi articoli (LineItem)

```
public class Invoice
{
    . . .
    private Address billingAddress;
    private ArrayList<LineItem> items;
}
```

Implementazione

- Un `LineItem` deve contenere un oggetto `Product` e una quantità:

```
public class LineItem
{
    . . .
    private int quantity;
    private Product theProduct;
}
```



Implementazione

- Il codice dei metodi è molto semplice
- Es.:
 - `getTotalPrice` di `LineItem` prende il prezzo unitario del prodotto e lo moltiplica per la quantità

```
/**
 * Computes the total cost of this line item.
 * @return the total price
 */
public double getTotalPrice()
{
    return theProduct.getPrice() * quantity;
}
```

File InvoiceTester.java

```
01: /**
02:     This program tests the invoice classes by printing
03:     a sample invoice.
04: */
05: public class InvoiceTester
06: {
07:     public static void main(String[] args)
08:     {
09:         Address samsAddress
10:             = new Address("Sam's Small Appliances",
11:                 "100 Main Street", "Anytown", "CA", "98765");
12:
13:         Invoice samsInvoice = new Invoice(samsAddress);
14:         samsInvoice.add(new Product("Toaster", 29.95), 3);
15:         samsInvoice.add(new Product("Hair dryer", 24.95), 1);
16:         samsInvoice.add(new Product("Car vacuum", 19.99), 2);
17:
18:         System.out.println(samsInvoice.format());
19:     }
20: }
21:
22:
23:
```


File Invoice.java

```
01: import java.util.ArrayList;
02:
03: /**
04:     Describes an invoice for a set of purchased products.
05: */
06: public class Invoice
07: {
08:     /**
09:         Constructs an invoice.
10:         @param anAddress the billing address
11:     */
12:     public Invoice(Address anAddress)
13:     {
14:         items = new ArrayList<LineItem>();
15:         billingAddress = anAddress;
16:     }
17:
```

File Invoice.java

```
18:     /**
19:         Adds a charge for a product to this invoice.
20:         @param aProduct the product that the customer ordered
21:         @param quantity the quantity of the product
22:     */
23:     public void add(Product aProduct, int quantity)
24:     {
25:         LineItem anItem = new LineItem(aProduct, quantity);
26:         items.add(anItem);
27:     }
28:
29:     /**
30:         Formats the invoice.
31:         @return the formatted invoice
32:     */
33:     public String format()
34:     {
```

File Invoice.java

```
35:         String r = "                               I N V O I C E\n\n"
36:             + billingAddress.format()
37:             + String.format("\n\n%-30s%8s%5s%8s\n",
38:                 "Description", "Price", "Qty", "Total");
39:
40:         for (LineItem i : items)
41:         {
42:             r = r + i.format() + "\n";
43:         }
44:
45:         r = r + String.format("\nAMOUNT DUE: $%8.2f",
46:             getAmountDue());
47:         return r;
48:     }
49:
```

File Invoice.java

```
50:     /**
51:         Computes the total amount due.
52:         @return the amount due
53:     */
54:     public double getAmountDue()
55:     {
56:         double amountDue = 0;
57:         for (LineItem i : items)
58:         {
59:             amountDue = amountDue + i.getTotalPrice();
60:         }
61:         return amountDue;
62:     }
63:
64:     private Address billingAddress;
65:     private ArrayList<LineItem> items;
66: }
```

File LineItem.java

```
01: /**
02:     Describes a quantity an article to purchase.
03: */
04: public class LineItem
05: {
06:     /**
07:         Constructs an item from the product and quantity.
08:         @param aProduct the product
09:         @param aQuantity the item quantity
10:     */
11:     public LineItem(Product aProduct, int aQuantity)
12:     {
13:         theProduct = aProduct;
14:         quantity = aQuantity;
15:     }
16:
```

File LineItem.java

```
17:     /**
18:         Computes the total cost of this line item.
19:         @return the total price
20:     */
21:     public double getTotalPrice()
22:     {
23:         return theProduct.getPrice() * quantity;
24:     }
25:
26:     /**
27:         Formats this item.
28:         @return a formatted string of this item
29:     */
30:     public String format()
```

File LineItem.java

```
31:     {
32:         return String.format("%-30s%8.2f%5d%8.2f",
33:             theProduct.getDescription(),
34:             theProduct.getPrice(),
35:             quantity, getTotalPrice());
36:     }
37:     private int quantity;
38:     private Product theProduct;
39: }
```

File Product.java

```
01: /**
02:     Describes a product with a description and a price.
03: */
04: public class Product
05: {
06:     /**
07:         Constructs a product from a description and a price.
08:         @param aDescription the product description
09:         @param aPrice the product price
10:     */
11:     public Product(String aDescription, double aPrice)
12:     {
13:         description = aDescription;
14:         price = aPrice;
15:     }
16:
```


File Product.java

```
17:     /**
18:         Gets the product description.
19:         @return the description
20:     */
21:     public String getDescription()
22:     {
23:         return description;
24:     }
25:
26:     /**
27:         Gets the product price.
28:         @return the unit price
29:     */
30:     public double getPrice()
31:     {
32:         return price;
33:     }
34:
35:     private String description;
36:     private double price;
37: }
```

File Address.java

```
01: /**
02:     Describes a mailing address.
03: */
04: public class Address
05: {
06:     /**
07:         Constructs a mailing address.
08:         @param aName the recipient name
09:         @param aStreet the street
10:         @param aCity the city
11:         @param aState the two-letter state code
12:         @param aZip the ZIP postal code
13:     */
14:     public Address(String aName, String aStreet,
15:                   String aCity, String aState, String aZip)
16:     {
```

File Address.java

```
17:     name = aName;
18:     street = aStreet;
19:     city = aCity;
20:     state = aState;
21:     zip = aZip;
22: }
23:
24: /**
25:     Formats the address.
26:     @return the address as a string with three lines
27: */
28: public String format()
29: {
30:     return name + "\n" + street + "\n"
31:         + city + ", " + state + " " + zip;
32: }
33:
```

File Address.java

```
34:     private String name;  
35:     private String street;  
36:     private String city;  
37:     private String state;  
38:     private String zip;  
39: }  
40:
```

Commenti

- La progettazione per la stampa di una fattura può essere migliorata
 - Problema: Invoice e LineItem si occupano sia di calcolare dati che di formattare l'output.
 - In genere, è meglio scindere questi aspetti
 - Invoice deve sapere come LineItem formatta l'output per etichettare in maniera opportuna le colonne
 - Esercizio:
 - Migliorare la progettazione della stampa di una fattura utilizzando una classe InvoiceFormatter che si occupa dell'impaginazione della fattura
-