14 . Dal progetto al codice

Sviluppo di Applicazioni Software

Matteo Baldoni

a.a. 2023/24

Università degli Studi di Torino - Dipartimento di Informatica

Attenzione!

Si noti che

questi lucidi sono basati sul libro di testo del corso "C. Larman, *Applicare UML e i Pattern*, Pearson, 2016".

Table of contents

Trasformare i progetti in codice

Dal progetto al codice

- Identificare i requisiti ✓
- Creare il modello di dominio ✓
- Definire i messaggi fra gli oggetti per soddisfare i requisiti Da fare!
- Aggiungere i metodi alle classi appropriate Da fare!

Dal progetto al codice

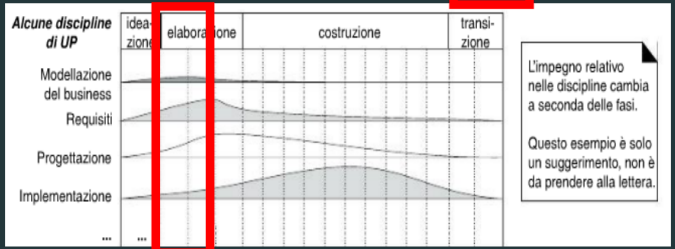
- Identificare i requisiti ✓
- Creare il modello di dominio ✓
- Definire i messaggi fra gli oggetti per soddisfare i requisiti ✓
- Aggiungere i metodi alle classi appropriate ✓

Siamo pronti per la realizzazione del **Modello di Implementazione**, costituito da tutti gli elaborati dell'implementazione, come il codice sorgente, la definizione delle basi di dati, le pagine JSP/XML/HTML, ecc.

UP maps

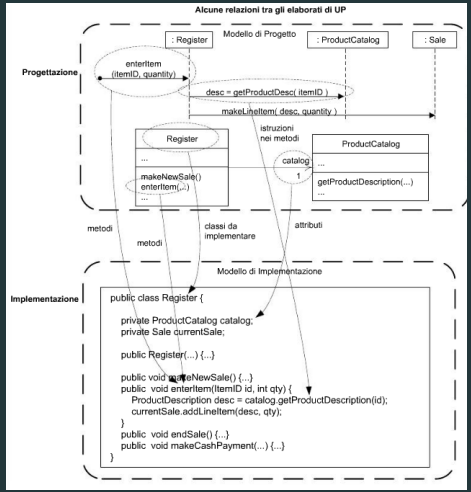
Tabella 2.1 Scenario di Sviluppo di esempio (i – inizio; r – raffinamento)

Disciplina	Pratica	Elaborato Iterazione →	Ideazione I1	Elaboraz. E1..En	Costr. C1..Cn	Transiz. T1..T2
Modellazione del business	modellazione agile workshop requisiti	Modello di Dominio		i		
Requisiti	workshop requisiti esercizio sulla visione votazione a punti	Modello dei Casi d'Uso	i	r		
		Visione	i	r		
		Specifica Supplementare	i	r		
		Glossario	i	r		
Progettazione	modellazione agile sviluppo guidato dai test	Modello di Progetto		i	r	
		Documento dell'Architettura Software		i		
		Modello dei Dati		i	r	
Implementazione	sviluppo guidato dai test programmazione a coppie integrazione continua standard di codifica	…				
Gestione del progetto	gestione del progetto agile riunioni Scrum giornaliere	…				
…						



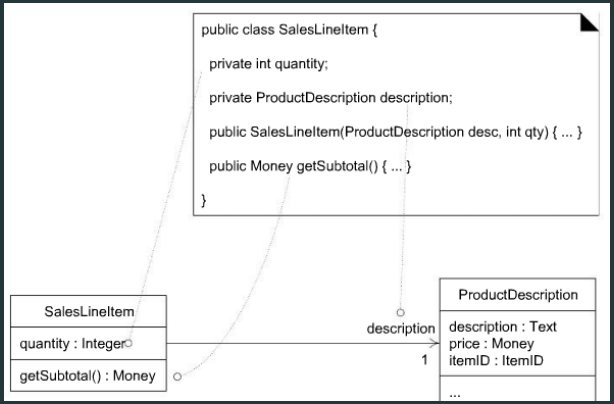
Dagli elaborati della progettazione a quelli dell'implementazione

- Java è solo un linguaggio di esempio! Molti altri linguaggi object-oriented possono essere utilizzati.
- L'implementazione è un processo di traduzione relativamente meccanico. Tuttavia durante la programmazione ci si devono aspettare e si devono pianificare numerosi cambiamenti e deviazioni rispetto al progetto realizzato.
- Questo è un atteggiamento essenziale e pragmatico nei metodi iterativi ed evolutivi.



Definire una classe con gli attributi e le firme dei metodi

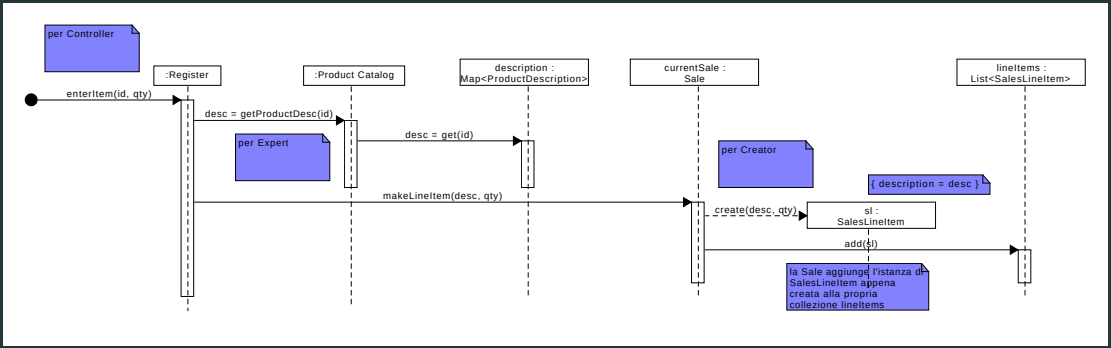
La traduzione in termini di definizione di attributi e di firme di metodi è spesso immediata. Ecco l'esempio della classe *SalesLineItem*.



©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Creare metodi dai diagrammi di interazione

La sequenza dei messaggi in un diagramma di interazione si traduce in una serie di istruzioni nelle definizioni di metodi e costruttori.



Collezioni

Le relazioni *uno-a-molti* sono implementate di solito con l'introduzione di un oggetto collezione.

La definizione dell'attributo *lineItems*.

```
private List<SalesLineItem> lineItems = new ArrayList<>();
```

Collezioni

©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

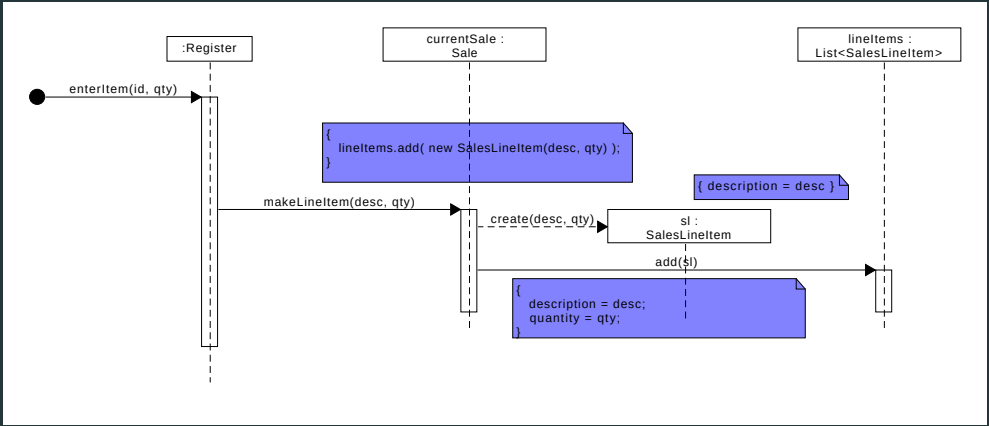
Le relazioni *uno-a-molti* sono implementate di solito con l'introduzione di un oggetto collezione.

La definizione dell'attributo *lineItems*.

```
private List<SalesLineItem> lineItems = new ArrayList<>();
```

Nota: se un oggetto implementa un'interfaccia, si dichiari la variabile in termini dell'interfaccia, non della classe concreta.

Altro esempio: Sale.makeLineItem

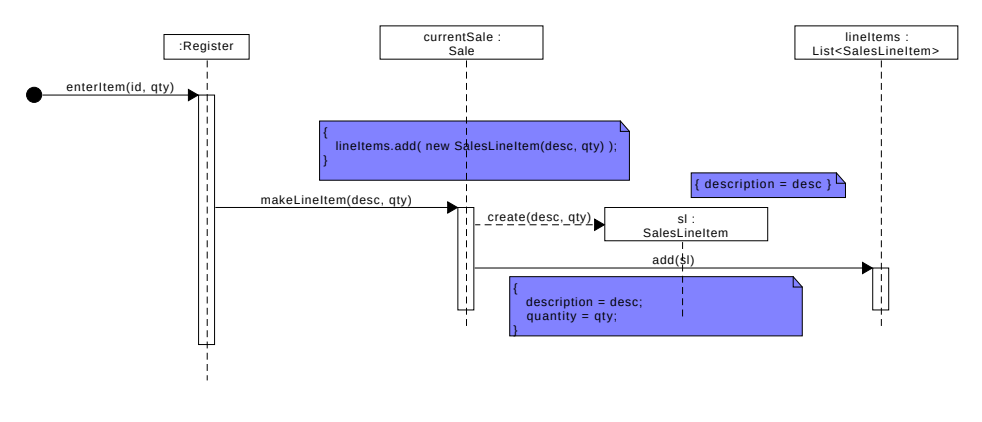


Il metodo *makeLineItem* della classe *Sale* può essere scritto per ispezione del diagramma di collaborazione per *enterItem*.

Costruttore

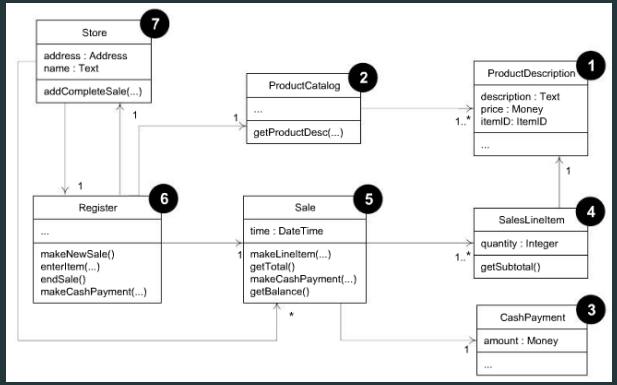
Il *costruttore* della classe *SalesLineItem* è generato, per ispezione, da una versione parziale del diagramma di interazione per *enterItem*.

Costruttore



Il *costruttore* della classe *SalesLineItem* è generato, per ispezione, da una versione parziale del diagramma di interazione per *enterItem*.

Ordine di implementazione



©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Le classi possono essere implementate in modo e in ordine diverso, per esempio dalla meno accoppiata alla più accoppiata.

Si veda libro di testo dalla pagina 428 alla pagina 432 (da 381 a 388 quinta edizione)

Ordine di implementazione

```
package com.foo.nextgen.domain;
```

Classe ProductDescription

```
public class ProductDescription {

   private ItemID id;
   private Money price;
   private String description;
```

©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Ordine di implementazione

```
    public ProductDescription
      ( ItemID id, Money price, String desc ) {
        this.id = id;
        this.price = price;
        this.description = desc;
    }

    public ItemID getItemID() { return id; }
    public Money getPrice() { return price; }
    public String getDescription() { return description; }
}
```

Classe ProductCatalog

```
public class ProductCatalog {

   private Map<ItemID, ProductDescription> descriptions;

   public ProductCatalog() {
      descriptions = new HashMap<>();
      // carica dei dati di esempio
      loadProductDescriptions();
   }

   public ProductDescription getProductDescription(ItemID id) {
      return descriptions.get(id);
   }

   /* carica dei dati di prova */
   private void loadProductDescriptions() {
      ItemID id;
      Money price;
      ProductDescription pd;

      id = new ItemID("100");
      price = new Money(3);
      pd = new ProductDescription(id, price, "pr. #100");
      descriptions.put(id, pd);

      id = new ItemID("200");
      price = new Money(4);
      pd = new ProductDescription(id, price, "pr. #200");
      descriptions.put(id, pd);
   }
}
```

Ordine di implementazione

Classe Register

```
public class Register {

   private Store store;
   private ProductCatalog catalog;
   private Sale currentSale;

   public Register(Store store, ProductCatalog catalog) {
       this.store = store;
       this.catalog = catalog;
       this.currentSale = null;
   }

   public void makeNewSale() {
       currentSale = new Sale();
   }
   public void enterItem(ItemID id, int quantity) {
       ProductDescription desc =
          catalog.getProductDescription(id);
       currentSale.makeLineItem(desc, quantity);
   }
   public void endSale() {
       // niente da fare
   }
   public void makeCashPayment(Money cashTendered) {
       currentSale.makeCashPayment(cashTendered);
       store.addSale(currentSale);
   }
}
```

Classe Sale

```
public class Sale {

   private List<SalesLineItem> lineItems;
   private Date date;
   private CashPayment payment;

   public Sale() {
       lineItems = new ArrayList<>();
       date = new Date();
       payment = null;
   }

   public void makeLineItem(ProductDescription desc, int qty) {
       lineItems.add( new SalesLineItem(desc, qty) );
   }
```

Ordine di implementazione

```
    public Money getTotal() {
        Money total = new Money(0);
        Money subtotal;
        for (SalesLineItem lineItem : lineItems) {
            subtotal = lineItem.getSubtotal();
            total = total.add(subtotal);
        }
        return total;
    }
    public void makeCashPayment(Money cashTendered) {
        payment = new CashPayment( cashTendered );
    }
    public Money getBalance() {
        return payment.getAmount().minus(getTotal());
    }
}
```

Classe SalesLineItem

```
public class SalesLineItem {

    private int quantity;
    private ProductDescription description;

    public SalesLineItem(ProductDescription desc, int qty) {
        this.description = desc;
        this.quantity = qty;
    }

    public Money getSubtotal() {
        return description.getPrice().times(quantity);
    }
}
```

Classe CashPayment

```
public class CashPayment {

    private Money amount;

    public CashPayment(Money cashTendered) {
        this.amount = cashTendered;
    }

    public Money getAmount() { return amount; }
}
```

Ordine di implementazione

Classe Store

```
public class Store {

   private ProductCatalog catalog;
   private Register register;
   private List<Sale> completedSales;

   public Store() {
      catalog = new ProductCatalog();
      register = new Register(this, catalog);
      completedSales = new ArrayList<>();
   }

   public Register getRegister() {  return register;  }

   public void addSale(Sale s) {
      completedSales.add(s);
   }
}
```

©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Sviluppo guidato dai test e refactoring

Test e refactoring

Extreme Programming (XP) e test

Extreme Programming ha promosso la pratica dei *test*: scrivere i test per *primi*.

Extreme Programming (XP) e refactoring

Extreme Programming ha inoltre promosso il *refactoring continuo* del codice per migliorare la qualità: meno duplicazioni, maggiore chiarezza, ecc.

Sviluppo guidato dai test

Test-Driven Development (TDD)

Una pratica promossa dal metodo iterativo e agile XP (applicabile a UP) è lo sviluppo guidato dai test, noto come sviluppo preceduto dai test.

Il codice dei test è scritto prima del codice da verificare, *immaginando* che il codice da testare sia scritto.

Vantaggi dello sviluppo guidato dai test

- I test unitari (ovvero i test relativi a singole classe e metodi) vengono effettivamente scritti
- La soddisfazione del programmatore porta a una scrittura più coerente dei test[1]
- Chiarimento dell'interfaccia e del comportamento dettagliati
- Verifica dimostrabile, ripetibile e automatica
- Fiducia nei cambiamenti

[1] Lo *sviluppo seguito dai test* è noto anche come *sviluppo solo per questa volta eviterò di scrivere i test*...

Vantaggi dello sviluppo guidato dai test

In generale il TDD prevede l'utilizzo di diversi tipi di test:

- **Test unitari**: hanno lo scopo di verificare il funzionamento delle piccole parti (unità) del sistema ma non di verificare il sistema nel suo complesso
- **Test di integrazione**: per verificare la comunicazione tra specifiche parti (elementi strutturali) del sistema
- **Test end-to-end**: per verificare il collegamento complessivo tra tutti gli elementi del sistema
- **Test di accettazione**: hanno lo scopo di verificare il funzionamento complessivo del sistema, considerato a scatola nera e dal punto di vista dell'utente, ovvero con riferimento a scenari di casi d'uso del sistema

Test unitari

Un metodo di test unitario è logicamente composto da quattro parti:

- **Preparazione**: crea l'oggetto (o il gruppo di oggetti) da verificare (chiamato anche la **fixture**) e prepara altri oggetti e/o risorse necessari per l'esecuzione del test
- **Esecuzione**: fa fare qualcosa alla fixture (per esempio, eseguire delle operazioni), viene richiesto lo specifico comportamento da verificare
- **Verifica**: valuta che i risultati ottenuti corrispondano a quelli previsti
- **Rilascio**: opzionalmente rilascia o ripulisce gli oggetti e/o le risorse utilizzate nel test (per evitare che altri test vengano corrotti)

Esempio di test per il metodo Sale.makeLineItem

```
import static org.junit.Assert.*;
import org.junit.*;

import com.foo.nextgen.domain.*;

public class SaleTest {

   @Test
   // test per il metodo Sale.makeLineItem
   public void testMakeLineItem() {
      // PREPARAZIONE
      // - crea la fixture, ovvero l'oggetto da testare
      // - un possibile idioma è chiamarlo 'fixture'
      // - spesso è definito come una variabile d'istanza
      //   anziché come una variabile locale
      Sale sale = new Sale();
      // - crea degli oggetti di supporto per il test
      ProductDescription tofu =
            new ProductDescription( new ItemID( "1" ),
                                    new Money( 2.50 ),
                                    "Tofu" );
      ProductDescription burger =
            new ProductDescription( new ItemID( "2" ),
                                    new Money( 1.50 ),
                                    "VeggieBurger" );

      // ESECUZIONE
      // questo codice viene scritto **immaginando** che
      // ci sia già un metodo makeLineItem.
      // Questo atto di immaginazione mentre viene scritto
      // il test tende a migliorare o a chiarire la nostra
      // comprensione dell'interfaccia dettagliata dell'oggetto.
      // Pertanto il TDD ha il vantaggio collaterale di
      // chiarire la progettazione a oggetti dettagliata.
      sale.makeLineItem( tofu, 2 );
      sale.makeLineItem( burger, 1 );

      // VERIFICA
      // confronta il risultato atteso con quello effettivo
      assertEquals( new Money(6.50), sale.getTotal() );
   }
}
```

Refactoring

Refactoring

Il **refactoring** è un metodo *strutturato* e *disciplinato* per scrivere o ristrutturare del codice esistente senza però modificare il comportamento esterno, applicando piccoli passi di trasformazione in combinazione con la ripetizione dei test ad ogni passo.

Il refactoring continuo del codice è un'altra pratica di XP applicabile a tutti i metodi iterativi (compreso UP).

Refactoring e test

L'essenza del refactoring è applicare piccole trasformazioni che preservano il comportamento.

Refactoring e test

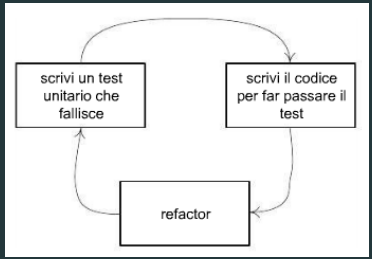
Dopo ciascuna trasformazione, i test unitari vengono eseguiti nuovamente per dimostrare che il refactoring non abbia provocato una *regressione* (un *fallimento*).

C'è una relazione tra il refactoring e il TDD: tutti i test unitari sostengono il processo di refactoring.

Refectoring e test

Regole per il TDD e refactoring:

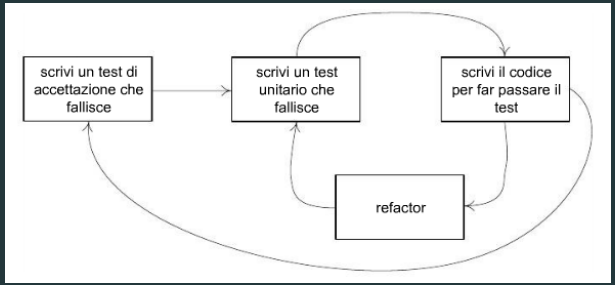
- Scrivi un test unitario che fallisce, per dimostrare la mancanza di una funzionalità o di codice
- Scrivi il codice più semplice possibile per far passare il test
- Riscrivi o ristruttura (refactor) il codice, migliorandolo, oppure passa a scrivere il prossimo test unitario



©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Refectoring e test

Estensione del ciclo di base del TDD con un doppio ciclo, più ampio. Il ciclo più ampio è relativo ai test di accettazione, ad esempio per un'intera esecuzione di uno specifico caso d'uso.



©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Obiettivi del refactoring

Gli obiettivi del refactoring sono gli obiettivi e le attività di una buona programmazione:

- Eliminare il codice duplicato
- Migliorare la chiarezza
- Abbreviare i metodi lunghi
- Eliminare l'uso dei letterali costanti hard-coded
- altro...

Alcuni refactoring

Refactoring	Descrizione
Rename	Per cambiare il nome di una classe, un metodo o un campo, per rendere più evidente il suo scopo. Semplice ma estremamente utile.
Extract Method	Trasforma un metodo lungo in uno più breve, estraendone una parte in un metodo di supporto.
Extract Class	Crea una nuova classe e vi muove alcuni campi e metodi da un'altra classe.
Extract Constant	Sostituisce un letterale costante con una variabile costante.
Move Method	Crea un nuovo metodo, con un corpo simile, nella classe che lo usa di più.
Introduce Explaining Variable	Mette il risultato dell'espressione, o di una parte dell'espressione, in una variabile temporanea con un nome che ne spiega lo scopo.
Replace Constructor Call with Factory Method	In Java, per esempio, sostituisce l'uso dell'operazione *new* e la chiamata di un costruttore con l'invocazione di un metodo di supporto che crea l'oggetto (nascondendo i dettagli).

©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Esempio: refactoring con Extract Method

Prima.

```
public class Player {

    private Piece piece;
    private Board board;
    private Die[] dice;
    // …

    public void takeTurn() {
        int rollTotal = 0;
        for (int i=0; i<dice.length; i++) {
            dice[i].roll();
            rollTotal += dice[i].getFaceValue();
        }
        Square oldLoc = piece.getLocation();
        Square newLoc = board.getSquare(oldLoc, rollTotal);
        piece.setLocation(newLoc);
    }
}
```

©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Dopo.

```
public class Player {

   private Piece piece;
   private Board board;
   private Die[] dice;
   // …

   public void takeTurn() {
       // chiama il metodo ottenuto dal refactoring
       int rollTotal = rollDice();

       Square oldLoc = piece.getLocation();
       Square newLoc = board.getSquare(oldLoc, rollTotal);
       piece.setLocation(newLoc);
   }

    // metodo di supporto estratto dal refactoring
    private int rollDice() {
        int rollTotal = 0;
        for (int i=0; i<dice.length; i++) {
            dice[i].roll();
            rollTotal += dice[i].getFaceValue();
        }
        return rollTotal;
    }
}
```

©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Esempio: refactoging con Introduce Explaining Variable

Prima.

```
// buon nome del metodo, ma la logica del corpo non è chiara
public boolean isLeapYear( int year ) {
    return ( year % 400 ) == 0 ||
           ( ( year % 4 ) == 0 && ( year % 100 ) != 0 );
}
```

©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Dopo.

```
// cosi va meglio
public boolean isLeapYear( int year ) {
    boolean isFourthYear = ( year % 4 ) == 0;
    boolean isHundrethYear = ( year % 100 ) == 0;
    boolean is4HundrethYear = ( year % 400 ) == 0;

    return is4HundrethYear || ( isFourthYear && ! isHundrethYear );
}
```

©C. Larman. Applicare UML e i Pattern. Pearson, 2016.