# Relazione Progetto: Sistema di Gestione Scolastica (01School)

**Corso:** Generation Italy - Java Full Stack Developer **Studente:** [Nome Studente] **Data:** 07 Gennaio 2026 **Versione Java:** Java 21

---

## 1. INTRODUZIONE

### 1.1 Cosa è il Progetto

Il progetto **01School** è un'applicazione Java SE che simula un sistema di gestione per un istituto scolastico. Il sistema permette di gestire le informazioni anagrafiche e professionali di diverse tipologie di persone presenti nell'ambiente scolastico: studenti, insegnanti e impiegati.

### 1.2 Obiettivi del Progetto

- Implementare una gerarchia di classi che rappresenti le diverse figure del sistema scolastico
- Applicare i principi della Programmazione Orientata agli Oggetti (OOP)
- Utilizzare il pattern architetturale MVC (Model-View-Controller)
- Gestire l'input/output da console e da file
- Applicare il concetto di ereditarietà e polimorfismo
- Implementare l'incapsulamento e la validazione dei dati

---

## 2. ARCHITETTURA DEL PROGETTO

### 2.1 Struttura delle Package

Il progetto è organizzato secondo una struttura gerarchica ben definita:

```
com.generation
  library/              # Classi utility riutilizzabili
      Console.java      # Gestione I/O da console
      FileReader.java   # Lettura file di testo
      FileWriter.java   # Scrittura file di testo
      Template.java     # Caricamento template HTML/TXT

  school/
      model/entities/   # Layer MODEL (MVC)
          Person.java   # Classe base astratta
          Student.java  # Studente (estende Person)
          Employee.java # Impiegato (estende Person)
          Teacher.java  # Insegnante (estende Employee)
```

```
view/             # Layer VIEW (MVC)
    PersonView.java
    StudentView.java
    EmployeeView.java
    TeacherView.java

demo/             # Applicazioni di test
    demoPerson.java
    demoStudent.java
    demoEmployee.java
    demoTeacher.java
```

### 2.2 Pattern Architetturale: MVC (Model-View-Controller)

Il progetto implementa il pattern **MVC** per separare le responsabilità:

- **Model** (`com.generation.school.model.entities`): contiene le entità di business (Person, Student, Employee, Teacher) con la logica di validazione e calcolo
- **View** (`com.generation.school.view`): gestisce la presentazione dei dati utilizzando template esterni
- **Controller** (implicito nelle classi `demo`): coordina l'interazione tra Model e View

**Vantaggi del pattern MVC:** - Separazione delle responsabilità (Separation of Concerns) - Facilità di manutenzione e testing - Riutilizzabilità del codice - Possibilità di modificare la presentazione senza toccare la logica di business

---

## 3. CONCETTI TEORICI IMPLEMENTATI

### 3.1 EREDITARIETÀ (Inheritance)

L'ereditarietà è il pilastro fondamentale del progetto, implementata attraverso una gerarchia di classi a tre livelli:

#### Gerarchia delle Classi

```
Person (classe base)
  Student (estende Person)
  Employee (estende Person)
      Teacher (estende Employee)
```

**Esempio di Implementazione** **Person.java** (classe base):

```
public class Person {
    protected int id;
```

```
    protected LocalDate dateOfBirth;
    protected String name;
    protected String surname;

    public Person(int id, String name, String surname, LocalDate dateOfBirth) {
        this.id = id;
        this.name = name;
        this.surname = surname;
        this.dateOfBirth = dateOfBirth;
    }
}
```

**Student.java** (classe derivata):

```
public class Student extends Person {
    protected int year;
    protected String section;

    public Student(int id, String name, String surname, LocalDate dateOfBirth,
                   int year, String section) {
        super(id, name, surname, dateOfBirth);  // Chiama costruttore della superclasse
        this.year = year;
        this.section = section;
    }
}
```

**Parola Chiave: `extends`**

- `Student extends Person` significa che Student **eredita** tutti gli attributi e metodi di Person
- Gli attributi sono dichiarati `protected` per essere accessibili alle classi figlie
- Il costruttore usa `super()` per richiamare il costruttore della classe genitore

**Parola Chiave: `super`**

- `super(parametri)` invoca il costruttore della classe genitore
- `super.metodo()` invoca un metodo della classe genitore
- Utilizzato in `isValid()` e `toString()` per riutilizzare la logica della classe base

**Esempio di utilizzo di super:**

```
@Override
public boolean isValid() {
    if (!super.isValid()) return false;  // Valida prima i campi della classe base
    if (year < 1 || year > 5) return false;  // Poi valida i campi specifici
```

```
    return true;
}
```

---

### 3.2 POLIMORFISMO E OVERRIDE

**Annotation @Override** Tutti i metodi sovrascritti utilizzano l'annotation `@Override` per: - Documentare esplicitamente l'intenzione di sovrascrivere un metodo - Ottenere controllo del compilatore (errore se il metodo non esiste nella superclasse) - Migliorare la leggibilità del codice

**Metodi Sovrascritti 1. toString() - Override del metodo Object**

Ogni classe sovrascrive `toString()` per fornire una rappresentazione testuale personalizzata:

```java
// Person.java
@Override
public String toString() {
    return "id=" + id + ", name='" + name + "', surname='" + surname +
           "', dateOfBirth=" + dateOfBirth;
}

// Student.java
@Override
public String toString() {
    return super.toString() + ", year=" + year + ", section='" + section + "'";
}
```

**Vantaggi:** - Riutilizzo del codice della superclasse con `super.toString()` - Aggiunta incrementale delle informazioni specifiche - Output formattato e leggibile

**2. isValid() - Validazione a catena**

Il metodo `isValid()` implementa un pattern di validazione gerarchica:

```java
// Person.java
public boolean isValid() {
    if (id < 0) return false;
    if (name == null || name.isEmpty()) return false;
    if (surname == null || surname.isEmpty()) return false;
    if (dateOfBirth == null || dateOfBirth.isAfter(LocalDate.now())) return false;
    return true;
}

// Employee.java
@Override
public boolean isValid() {
```

```
    if (!super.isValid()) return false;  // Validazione della classe base
    if (salary < 0) return false;
    if (role == null || role.trim().isEmpty()) return false;
    return true;
}

// Teacher.java
@Override
public boolean isValid() {
    if (!super.isValid()) return false;  // Validazione Employee + Person
    if (subject == null || subject.isEmpty()) return false;
    if (yearsOfExp < 0) return false;
    return true;
}
```

**Pattern implementato:** - Ogni classe valida i propri attributi specifici - La validazione parte dalla classe base e procede verso le classi derivate - Approccio "fail-fast": se la validazione base fallisce, non si controlla il resto

---

### 3.3 INCAPSULAMENTO (Encapsulation)

**Modificatori di Accesso** Il progetto utilizza correttamente i modificatori di accesso:

**1. Attributi `protected`**

```
protected int id;
protected String name;
protected LocalDate dateOfBirth;
```

**Perché `protected` e non `private`?** - `protected` permette l'accesso alle classi figlie (Student, Employee, Teacher) - Mantiene l'incapsulamento verso l'esterno (altre package) - Consente il riutilizzo diretto degli attributi nelle sottoclassi

**2. Metodi `public`**

```
public int getId() { return id; }
public void setId(int id) { this.id = id; }
```

**Pattern Getter/Setter** Ogni attributo ha il proprio getter e setter seguendo le convenzioni JavaBeans: - `getAttributo()` per la lettura - `setAttributo(tipo valore)` per la scrittura - Permette il controllo futuro dell'accesso (es. validazione nei setter)

**Metodi Calcolati getAge() - Logica di business incapsulata**

```
public int getAge() {
    if (dateOfBirth == null) return 0;
```

```
    return Period.between(dateOfBirth, LocalDate.now()).getYears();
}
```

L'età non è memorizzata come attributo ma **calcolata dinamicamente**: - Evita inconsistenze (l'età cambia nel tempo) - Utilizza l'API `java.time` (Period e LocalDate) - Esempio di **computed property**

---

### 3.4 GESTIONE DELLE DATE (java.time API)

Il progetto utilizza l'API moderna `java.time` (introdotta in Java 8):

#### LocalDate

```
protected LocalDate dateOfBirth;
```

**Vantaggi rispetto a Date/Calendar (legacy):** - Immutabile e thread-safe - API più intuitiva e leggibile - Supporto nativo per operazioni temporali

#### Period - Calcolo dell'età

```
public int getAge() {
    return Period.between(dateOfBirth, LocalDate.now()).getYears();
}
```

**Spiegazione:** - `LocalDate.now()` ottiene la data corrente - `Period.between(start, end)` calcola il periodo tra due date - `.getYears()` estrae solo gli anni dal periodo

#### Validazione delle Date

```
if (dateOfBirth == null || dateOfBirth.isAfter(LocalDate.now()))
    return false;
```

Verifica che la data di nascita: - Non sia null - Non sia nel futuro (isAfter confronta con oggi)

---

### 3.5 CLASSI UTILITY E METODI STATIC

#### Console.java - Classe Utility Statica

```
public class Console {
    private static Scanner keyboard = new Scanner(System.in);

    public static void print(Object x) {
        System.out.println(x);
    }
```

```
    public static String readString() {
        return keyboard.nextLine();
    }

    public static int readInt() {
        return Integer.parseInt(keyboard.nextLine());
    }
}
```

**Caratteristiche:** - Tutti i metodi sono `static` (non serve istanziare la classe) - Scanner condiviso e statico (unica istanza per tutta l'applicazione) - Wrapper pattern: semplifica l'uso di Scanner

### Metodi con Validazione Integrata readIntBetween() - Validazione dell'intervallo

```
public static int readIntBetween(String msg, String errMsg, int min, int max) {
    int res;
    do {
        Console.print(msg);
        res = Console.readInt();
        if (res < min || res > max)
            Console.print(errMsg);
    } while (res < min || res > max);
    return res;
}
```

### readStringBetween() - Scelta multipla

```
public static String readStringBetween(String msg, String errMsg, String...values) {
    List<String> v = Arrays.asList(values);
    String res = "";
    do {
        Console.print(msg);
        res = Console.readString();
        if(!v.contains(res))
            Console.print(errMsg);
    } while(!v.contains(res));
    return res;
}
```

**Varargs (`String...values`):** - Permette di passare un numero variabile di parametri - Viene convertito automaticamente in un array - Sintassi: `readStringBetween("Scegli:", "Errore!", "A", "B", "C")`

---

### 3.6 GESTIONE ECCEZIONI (Exception Handling)

**Try-Catch per NumberFormatException**

```
public static int readInt() {
    try {
        return Integer.parseInt(keyboard.nextLine());
    }
    catch(NumberFormatException e) {
        throw new NumberFormatException("Il valore inserito non è un numero.");
    }
}
```

**Spiegazione:** - `Integer.parseInt()` può lanciare `NumberFormatException` se la stringa non è un numero - Il catch intercetta l'eccezione e la rilancia con un messaggio più chiaro - Pattern: **catch and rethrow with custom message**

**Throws Declaration FileReader.java:**

```
public FileReader(String path) throws FileNotFoundException {
    this.file = new File(path);
    this.sc = new Scanner(file);
}
```

**Differenza tra `throws` e `throw`:** - `throws` nella firma del metodo: dichiara che il metodo **può** lanciare un'eccezione - `throw` nel corpo: lancia **effettivamente** un'eccezione - Il chiamante deve gestire l'eccezione con try-catch o propagarla

---

### 3.7 PATTERN TEMPLATE E FILE I/O

**Template.java - Caricamento Template**

```
public class Template {
    public static String load(String path) {
        FileReader fr = new FileReader(path);
        return fr.readAll();
    }
}
```

**Pattern Template:** - Separa la struttura (template HTML/TXT) dai dati - I placeholder vengono sostituiti dinamicamente - Esempio di **separation of concerns**

**PersonView.java - Rendering con Template**

```
public class PersonView {
    protected String template;
```

```
    public PersonView(String template) {
        this.template = template;
    }

    public String render(Person p) {
        return Template.load(template)
            .replace("[id]", p.getId()+"")
            .replace("[name]", p.getSurname())
            .replace("[surname]", p.getName())
            .replace("[dateOfBirth]", p.getDateOfBirth().toString())
            .replace("[age]", p.getAge()+"");
    }
}
```

**Meccanismo:** 1. Carica il template dal file 2. Sostituisce i placeholder `[campo]` con i valori reali 3. Restituisce la stringa formattata

**Vantaggi:** - Presentazione separata dalla logica - Facile modificare il layout senza toccare il codice Java - Riutilizzabilità dei template

### FileReader.java - Lettura File

```
public class FileReader {
    private File file;
    private Scanner sc;

    public FileReader(String path) throws FileNotFoundException {
        this.file = new File(path);
        this.sc = new Scanner(file);
    }

    public String readAll() {
        String output = "";
        while(sc.hasNextLine()) {
            output += sc.nextLine() + "\n";
        }
        return output;
    }
}
```

**Wrapper Pattern:** - Incapsula la complessità di File e Scanner - Fornisce un'interfaccia semplificata (`readAll()`) - Gestisce le eccezioni FileNotFoundException

### FileWriter.java - Scrittura File

```
public class FileWriter {
    private File file;
```

```
    public FileWriter(String path) throws IOException {
        this.file = new File(path);
        file.createNewFile();
    }

    public boolean write(String text) {
        try (PrintWriter pw = new PrintWriter(file)) {
            pw.println(text);
            return true;
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

**Try-with-resources:** - `try (PrintWriter pw = ...)` garantisce la chiusura automatica del file - Implementa l'interfaccia `AutoCloseable` - Evita resource leak (perdite di risorse)

---

### 3.8 JAVADOC E DOCUMENTAZIONE

Il codice è ampiamente documentato con **Javadoc**, il sistema standard di documentazione Java.

#### Struttura Javadoc

```
/**
 * Calcola l'età della persona in base alla data di nascita
 * Utilizza Period per calcolare la differenza tra la data corrente e quella di nascita
 *
 * @return l'età in anni, o 0 se la data di nascita non è impostata
 */
public int getAge() {
    if (dateOfBirth == null) return 0;
    return Period.between(dateOfBirth, LocalDate.now()).getYears();
}
```

#### Tag Javadoc utilizzati

- `@param nomeParametro` - Descrive un parametro del metodo
- `@return` - Descrive cosa restituisce il metodo
- `/** */` - Commento multilinea per Javadoc
- Descrizione testuale prima dei tag

**Vantaggi:** - Genera automaticamente documentazione HTML - Supporto IDE (tooltip e autocompletamento) - Standard professionale per progetti Java

---

### 3.9 CONVENZIONI DI CODICE (Code Conventions)

**Naming Conventions** **Classi:** PascalCase

```
public class Person { }
public class StudentView { }
```

**Metodi e variabili:** camelCase

```
public int getAge() { }
private LocalDate dateOfBirth;
```

**Costanti:** UPPER_SNAKE_CASE (non presenti nel progetto, ma standard Java)

```
public static final int MAX_STUDENTS = 30;
```

**Organizzazione del Codice** Ogni classe segue una struttura consistente: 1. **Package declaration** 2. **Import statements** 3. **Javadoc della classe** 4. **Attributi** (con separatori commentati) 5. **Costruttori** 6. **Getter e Setter** 7. **Metodi di business logic** 8. **Override di metodi**

**Esempio:**

```
package com.generation.school.model.entities;

import java.time.LocalDate;

/**
 * Classe che rappresenta uno studente
 */
public class Student extends Person {
    // ==================== ATTRIBUTI ====================
    protected int year;

    // ==================== COSTRUTTORI ====================
    public Student() { }

    // ==================== GETTER E SETTER ====================
    public int getYear() { return year; }

    // ==================== METODI DI VALIDAZIONE ====================
    @Override
    public boolean isValid() { ... }
```

```
    // ==================== OVERRIDE METODI ====================
    @Override
    public String toString() { ... }
}
```

**Commenti Separatori**

```
// ==================== ATTRIBUTI ====================
// ==================== COSTRUTTORI ====================
// ==================== GETTER E SETTER ====================
```

**Scopo:** - Migliorano la leggibilità del codice - Facilitano la navigazione in classi lunghe - Standard nei progetti enterprise

---

## 4. COME FUNZIONA IL PROGETTO

### 4.1 Flusso di Esecuzione Tipico

**Esempio: demoStudent.java**

```
1. Creazione oggetto Student
   ↓
2. Validazione con isValid()
   ↓
3. Se valido: creazione StudentView
   ↓
4. Rendering con template
   ↓
5. Output su console o file
```

### 4.2 Esempio di Utilizzo Completo

```
// 1. Creazione studente
Student s = new Student(
    1,
    "Mario",
    "Rossi",
    LocalDate.of(2005, 3, 15),
    3,
    "A"
);

// 2. Validazione
if (s.isValid()) {
    // 3. Creazione view
    StudentView view = new StudentView("template/student.txt");
```

```
    // 4. Rendering
    String output = view.render(s);

    // 5. Visualizzazione
    Console.print(output);
}
```

---

## 5. PERCHÉ QUESTE SCELTE TECNICHE

### 5.1 Ereditarietà vs Composizione

**Scelta:** Ereditarietà (Student **è una** Person)

**Motivazione:** - Relazione IS-A naturale (uno studente è una persona) - Riutilizzo del codice (attributi comuni) - Polimorfismo (trattare Student come Person quando necessario)

### 5.2 Protected vs Private

**Scelta:** Attributi `protected`

**Motivazione:** - Accesso diretto nelle classi derivate (no getter/setter interni) - Mantiene incapsulamento verso l'esterno - Migliora leggibilità (no `this.getName()` ma diretto `name`)

### 5.3 LocalDate vs Date

**Scelta:** `LocalDate` (Java 8+ Time API)

**Motivazione:** - Immutabilità (thread-safe) - API moderna e intuitiva - Metodi utili (isAfter, isBefore, Period) - Date è deprecato e problematico

### 5.4 Pattern MVC

**Scelta:** Separazione Model-View

**Motivazione:** - Testabilità (testare logica senza UI) - Manutenibilità (modifiche isolate) - Riutilizzabilità (stesso Model, diverse View) - Standard professionale

### 5.5 Template esterni vs Hardcoded

**Scelta:** Template su file esterni

**Motivazione:** - Separazione presentation/logic - Modificabile da non-programmatori (designer) - Riutilizzabilità (stesso template per più oggetti) - Facilita l'internazionalizzazione

---

## 6. ARGOMENTI TEORICI CORRELATI

### 6.1 Principi OOP Applicati

1. **Incapsulamento:**
   - Attributi protected/private
   - Getter/Setter per controllo accessi
   - Validazione interna (isValid)
2. **Ereditarietà:**
   - Gerarchia Person → Student/Employee → Teacher
   - Riutilizzo codice con `extends`
   - Chiamate a superclasse con `super`
3. **Polimorfismo:**
   - Override di toString() e isValid()
   - Annotation @Override
   - Binding dinamico (metodo chiamato in base al tipo runtime)
4. **Astrazione:**
   - Person come classe base generica
   - Specializzazioni concrete (Student, Teacher)
   - Metodi comuni definiti in Person

### 6.2 Design Patterns

1. **MVC (Model-View-Controller)**
   - Model: entities package
   - View: view package
   - Controller: demo classes
2. **Template Method Pattern**
   - Template esterni con placeholder
   - Sostituzione dinamica con replace()
3. **Wrapper/Adapter Pattern**
   - Console wrappa Scanner
   - FileReader wrappa File e Scanner
   - Semplifica API complesse
4. **Factory Pattern (implicito)**
   - Template.load() crea istanze FileReader
   - Centralizza creazione oggetti

### 6.3 Best Practices Java

1. **Convenzioni di codice**
   - Naming (PascalCase, camelCase)
   - Organizzazione package
   - Struttura classi consistente
2. **Gestione risorse**

- Try-with-resources per file I/O
- Chiusura automatica Scanner

3. **Gestione eccezioni**
   - Try-catch per parsing
   - Throws per I/O
   - Messaggi di errore chiari
4. **Documentazione**
   - Javadoc completo
   - Commenti esplicativi
   - Separatori sezioni

---

# 7. POSSIBILI EVOLUZIONI

### 7.1 Miglioramenti Tecnici

1. **Persistenza Dati:**
   - Aggiungere JPA/Hibernate annotations (@Entity, @Table)
   - Database integration (MySQL, PostgreSQL)
   - Repository pattern
2. **Validazione:**
   - Bean Validation (JSR-303) con @NotNull, @Size, @Min, @Max
   - Sostituire isValid() manuale con annotations
   - Validator framework
3. **Dependency Injection:**
   - Spring Framework
   - Autowiring
   - Configuration management
4. **Testing:**
   - JUnit 5 per unit testing
   - Mockito per mock objects
   - Test coverage

### 7.2 Funzionalità Aggiuntive

1. **CRUD Operations:**
   - Create, Read, Update, Delete per tutte le entità
   - Gestione collezioni (List)
   - Ricerca e filtri
2. **Relazioni tra Entità:**
   - Teacher  Subject (Many-to-Many)
   - Student  Classroom (Many-to-One)
   - Course enrollment system
3. **Business Logic:**
   - Calcolo medie voti
   - Gestione presenze

- Report generazione

---

## 8. CONCLUSIONI

Il progetto **01School** dimostra una solida comprensione dei fondamenti della programmazione orientata agli oggetti in Java. L'implementazione copre tutti gli argomenti chiave del corso Generation Italy:

**Concetti teorici padroneggiati:** - Ereditarietà e gerarchia di classi - Incapsulamento e modificatori di accesso - Polimorfismo e override - Gestione eccezioni - File I/O - Pattern MVC - Java Time API - Javadoc e documentazione

**Qualità del codice:** - Struttura ben organizzata con package logici - Naming conventions rispettate - Codice commentato e documentato - Separazione delle responsabilità - Riutilizzabilità e manutenibilità

**Applicabilità pratica:** - Base solida per un sistema gestionale reale - Facilmente estendibile con nuove entità - Pronto per integrazione database (JPA) - Adatto per evoluzione in applicazione web (Spring Boot)

Il progetto costituisce un'eccellente base di partenza per applicazioni enterprise più complesse, dimostrando la capacità di applicare i principi teorici della programmazione Java in un contesto pratico e realistico.

---

## RIFERIMENTI


- Oracle Java Documentation: https://docs.oracle.com/en/java/
- Java Time API: https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html
- Design Patterns (Gang of Four)
- Clean Code (Robert C. Martin)
- Effective Java (Joshua Bloch)


---

**Fine Relazione**