UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Elaborato di Ingegneria del Software

Applicativo in java Interpol

Cosimo Michelagnoli

October 21, 2020

Contents

Introduzione

L'elaborato è stato scritto in java con la finalità di simulare un possibile applicativo gestionale dell'interpol. Questo programma consente la simulazione di un archivio di fedine penali. Gli agenti possono interagire con il sistema inserendo fedine penali, ricercarle o modificarle. La fedina penale (CriminalProfile) è un oggetto complesso che attraverso il pattern builder può essere costruito in più momenti. Grazie al builder infatti, la fedina penale puo essere generata senza che sia neacessario specificare tutti gli attributi, alcuni di questi potrebbero non essere ancora noti e quindi il sistema permettere comunque la creazione della fedina penale. Il pattern builder (CriminalProfileBuilder) viene anche utilizzato dalla classe CriminalProfile delegandogli la crazione di alcuni suoi attributi nel caso non siano stati inizializati. L'archivio possiede un proxy (ProxyRealArchive) che simula dei livelli di accesso per la protezione di dati sensibili e un controllo sull'accesso ad alcune risorse. Alcune delle funzionalità del client richiedono infatti una sorta di autentificazione, il proxy prima di provvedere alla richiesta andrà a verificare il livello dell'agente che sta richiedendo tale funzionalità.

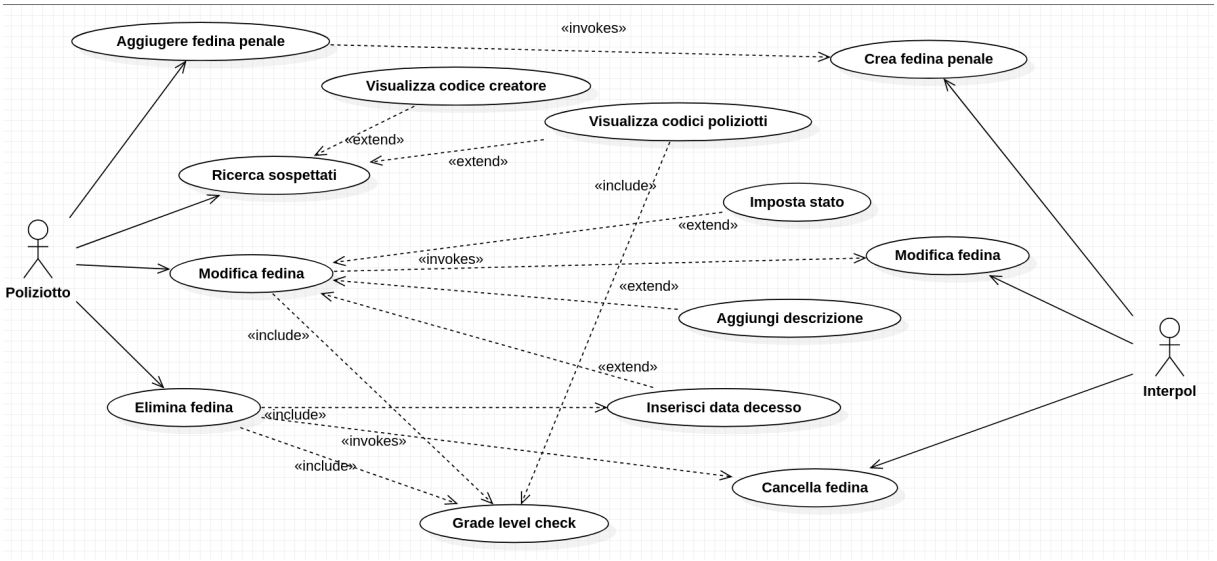
Chapter 1

Progettazione

Per la progettazione dei diagrammi UML ho utilizato il software StarUML. Il codice in java è stato scritto sull'IDE IntelliJ IDEA. Nella fase di progettazione sono stati indentificati i casi d'uso e la logica di dominio, in prospettiva di specifica/implementazione, attraverso un class diagram. Entrambi sono stati successivamente rielaborati durante la fase di implementazione. Una parte fondamentale dello sviluppo è data anche dalla realizzazione di alcune classi di testing (più precisamente di Unit Testing) attraverso il framework JUnit 5 integrato direttamente nell'ambiente di sviluppo. Infine è stato realizzato il Sequence Diagram, per mostrare l'interazione tra le classi durante l'esecuzione del programma per inserire i dati sensibili di una fedina penale ancora non presente nel sistema.

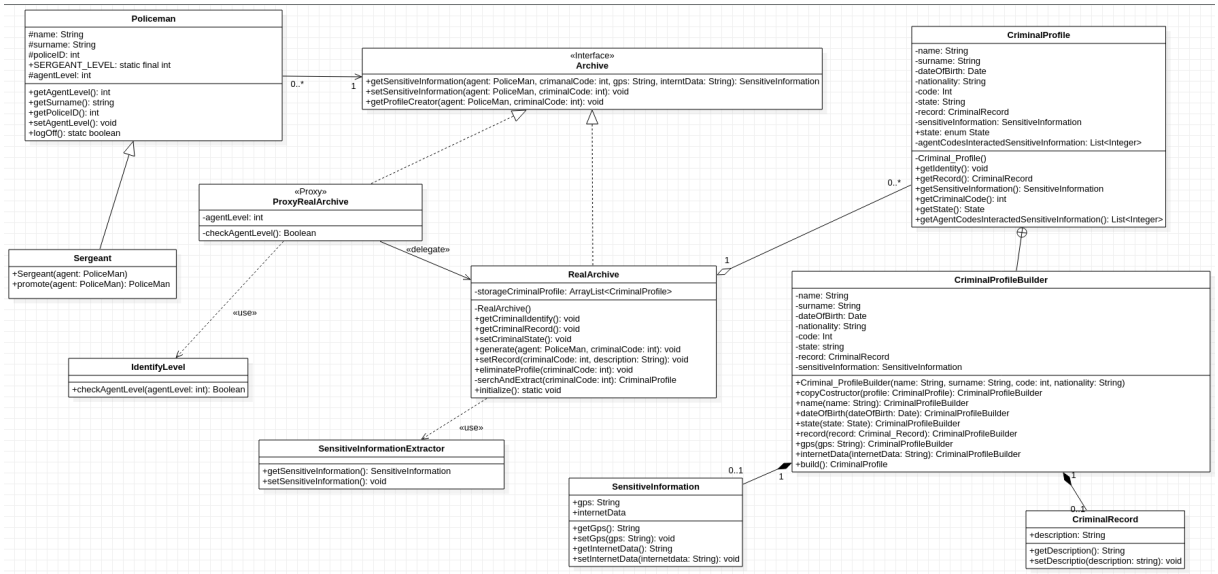
1.1 Casi d'uso

Questo diagramma rappresenta i requisiti e le finalità di questo progetto, mostrando gli attori e come interagiscono.



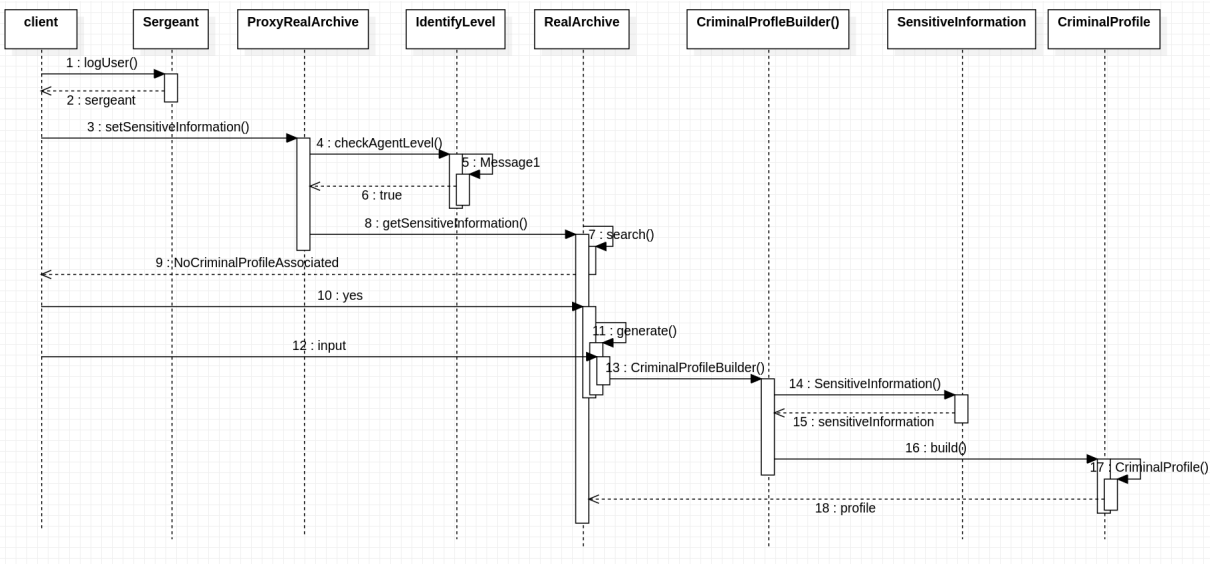
1.2 Class Diagram

Il seguente diagramma UML schematizza la logica di dominio del programma.



1.3 Sequence Diagram

Il seguente diagramma UML schematizza un possibile flusso d' esecuzione del programma se un Sergeant prova a inserire i dati sensibili di una fedina penale che non è presente nell'archivio.



Chapter 2

Implementazione

L'implementazione è stata la fase successiva alla progettazione. Questa fase ha permesso di definire meglio le finalità e le modalità di funzionamento del programma, obbligando alcune modifice ai diagrammi UML.

2.1 Classi ed interfacce

Le classi dell'applicativo sono nel package **com.interpol**:

- PoliceMan
  - Sergeant
- CriminalProfile
- CriminalRecord
- SensitiveInformation
- Archive (*interface*)
  - RealArchive
- IdentifyLevel

Oltre a queste classe sono state anche utilizzate classi ed interfacce della libreria standard di Java. Nel package **com.interpol** è stata definita anche una classe per gestire le eccezioni nel caso in cui venga richiesta un fedina penale non presente in archivio.

2.1.1 PoliceMan

Questa classe definisce il client che interagisce con l'archivio e può richedere di visionare e modificare le fedine penali.

```
package com.interpol;
import javax.swing.*;

public class PoliceMan {
    public static final int SERGEANT_LEVEL = 5;
    protected final String name;
    protected final String surname;
    protected int policeID;
    protected int agentLevel;

    public PoliceMan(String name, String surname, int policeID) {
        this.name = name;
        this.surname = surname;
        this.policeID = policeID;
    }
    public  int getAgentLevel(){ return agentLevel;}

    public String getName() { return name; }

    public String getSurname() { return surname; }

    public int getPoliceID() { return policeID; }

    public void setAgentLevel(int agentLevel) { this.agentLevel = agentLevel; }

    public static boolean logOff(PoliceMan agent) {

        JOptionPane.showMessageDialog( parentComponent: null,  message: "EXIT");

        return true;
    }
    }
```

Figure 2.1: Frammento di codice della classe PoliceMan

2.1.2 Sergeant

Sergeant è una classe che eredita da PoliceMan.

```java
package com.interpol;

public class Sergeant extends PoliceMan {

    public Sergeant(String name, String surname, int policeID) {
        super(name, surname, policeID);
        this.agentLevel = 5;
    }

    public Sergeant(PoliceMan agent) { this(agent.getName(),agent.getSurname(),agent.getAgentLevel()); }

    public  int getAgentLevel(){ return agentLevel;}

    public int getPoliceID() { return this.policeID; }

    public PoliceMan promote(PoliceMan agent) {
        int agentLevel = agent.getAgentLevel() + 1;
        agent.setAgentLevel(agentLevel);
        return agentLevel == PoliceMan.SERGEANT_LEVEL ? new Sergeant(agent) : agent;
    }
}
```

Figure 2.2: Frammento di codice della classe Sergeant

2.1.3 CriminalProfile

La classe **CriminalProfile** rappresenta la fedina penale di un soggetto inserito nell'archivio. La fedina presenta oltre all'anagrafica anche descrizioni del reato (report) e dati sensbili sul soggetto.

```java
package com.interpol;

import javax.swing.*;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class CriminalProfile {
    public enum State{
        RELEASED,
        WANTED,
        CONVICT,
        ESCAPEE
    }
    private final String name;
    private final String surname;
    private final Date dateOfBirth;
    private final String nationality;
    private final int criminalCode;
    private State state;
    private CriminalRecord record;
    private SensitiveInformation sensitiveInformation;
    private int creatorCode;
    private List<Integer> agentCodesInteractedSensitiveInformation;

    public CriminalProfile(String name, String surname, Date dateOfBirth, String nationality, int criminalCode,
                           State state, CriminalRecord record, SensitiveInformation sensitiveInformation, int creatorCode) {...}

    public String getName() { return name; }

    public String getSurname() { return surname; }
```

Figure 2.3: Frammento di codice della classe CriminalProfile

2.1.4 CriminalRecord

La classe **CriminalRecord** rappresenta un possibile testo associato alla fedina penale con la descrizione dei reati.

```java
package com.interpol;

public class CriminalRecord {
    private String description;

    public String getDescription() { return description; }

    public void setDescription(String description) { this.description = description; }
}
```

Figure 2.4: Frammento di codice della classe CriminalRecord

2.1.5 SensitiveInformation

La classe **SensitiveInformation** rappresenta il dato sensibile della fedina penale visionabile solo da oggetti Sergeant.

```java
package com.interpol;

public class SensitiveInformation {
    private String gps;
    private String internetData;

    public String getGps() { return gps; }

    public void setGps(String gps) { this.gps = gps; }

    public String getInternetData() { return internetData; }

    public void setInternetData(String internetData) { this.internetData = internetData; }
}
```

Figure 2.5: Frammento di codice della classe SensitiveInformation

2.1.6 Archive (interface)

L'interfaccia *Archive* viene utilizzata per definire su quali azioni verà verificato il livello di accesso dell'oggetto che richiede tali azioni.

```java
package com.interpol;

public interface Archive{
    public void setSensitiveInformation(PoliceMan agent, int criminalCode, String gps, String internetData);
    public SensitiveInformation getSensitiveInformation(PoliceMan agent, int criminalCode);
    public void getProfileCreator(PoliceMan agent,int criminalCode);
}
```

Figure 2.6: Frammento di codice dell' interfaccia Archive

2.1.7 RealArchive

Questa classe **RealArchive** implementa l'interfaccia e rappresenta l'archivio dell'interpol.

```java
package com.interpol;

import javax.swing.*;
import java.io.Serializable;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.*;

public class RealArchive implements Archive {
    private static final RealArchive instance;
    private ArrayList<CriminalProfile> storageCriminalProfile;
    //private List<Profile.com.interpol.CriminalProfile> mostWantedList;

    public RealArchive() { storageCriminalProfile = new ArrayList<>(); }

    static {
        instance = new RealArchive();
    }

    public static RealArchive getInstance() { return instance; }

    @Override
    public void setSensitiveInformation(PoliceMan agent, int criminalCode, String gps, String internetData) {...}

    @Override
    public SensitiveInformation getSensitiveInformation(PoliceMan agent, int criminalCode) {...}

    @Override
    public void getProfileCreator(PoliceMan agent, int criminalCode) {...}
```

Figure 2.7: Frammento di codice della classe RealArchive

2.1.8 IdentifyLevel

La classe **IdentifyLevel** esegue il controllo per verificare il livello di autentificazione per avere accesso ai dati sensibili.

```java
package com.interpol;

public class IdentifyLevel {
    public static boolean checkAgentLevel(int agentLevel) { return agentLevel == PoliceMan.SERGEANT_LEVEL; }
}
```

Figure 2.8: Frammento di codice della classe IdentifyLevel

2.2 Design Patterns

Nel progetto sono stati utilizzati i seguenti design patterns:

- Singleton
- Builder
- Protection Proxy

2.2.1 Singleton

Il singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza. In questo progetto il pattern singleton ha l'intento di fornire un punto di accesso globale all'oggetto di RealAchive e per impedire che vengano creati più istanze di RealArchive e ProxyRealArchive.

```
    private RealArchive() {
        storageCriminalProfile = new ArrayList<>();
    }

    static {
        instance = new RealArchive();
    }

    public static RealArchive getInstance() {
        return instance;
    }
```

Figure 2.9: Frammento di codice del pattern Singleton

2.2.2 Builder

Il design pattern Builder, nella programmazione ad oggetti, separa la costruzione di un oggetto complesso dalla sua rappresentazione cosicché il processo di costruzione stesso possa creare diverse rappresentazioni. Questo design pattern, definito nella classe CriminalProfileBuilder, è stato utilizzato per la creazione delle istanze di CriminalProfile che è un oggetto complesso che ha come attributi altri oggetti. Per rendere possibile l'aggiunta di attributi a una fedina già esistente è stato creato un costruttore di copia nella classe CriminalProfileBuilder.

```java
package com.interpol;

import java.util.Date;
import java.util.List;

public final class CriminalProfileBuilder {
    private  String name;
    private  String surname;
    private  Date dateOfBirth;
    private  String nationality;
    private final int criminalCode;
    private int creatorCode;
    private CriminalProfile.State profileState;
    private List<Integer> agentCodesInteractedSensitiveInformation;
    private CriminalRecord record;
    private SensitiveInformation sensitiveInformation;

    public CriminalProfileBuilder(int criminalCode) { this.criminalCode = criminalCode; }

    public CriminalProfile build() {
        return new CriminalProfile(name, surname, dateOfBirth, nationality,
                criminalCode, profileState, record, sensitiveInformation, creatorCode);
    }
    public CriminalProfileBuilder copyCostructor(CriminalProfile profile){
        this.name = profile.getName();
        this.surname = profile.getSurname();
        this.dateOfBirth = profile.getDateOfBirth();
        this.nationality = profile.getNationality();
        this.profileState = profile.getState();
        this.record = profile.getRecord();
        this.agentCodesInteractedSensitiveInformation = profile.getAgentCodesInteractedSensitiveInformation();
        return this;
    }
```

Figure 2.10: Frammento di codice della classe CriminalProfileBuilder

2.2.3 Protection Proxy

Nella sua forma più generale, un proxy è una classe che funziona come interfaccia per qualcos'altro. In questo programma il proxy ha al'intento di filtrare le chiamate a RealArchive e permettere solo ad alcune di queste di interagire con l'oggetto effettivo. Il controllo è effettuato dalla classe ProxyRealArchive sulla base del livello dell'agente che effettua la chiamata. In questo progetto per poter accedere ai dati sensibili è richiesto che l'agente abbia un livello pari a quello del Sergeant. Le chiamate effettuate da un agente di tipo PoliceMan non avranno successo.

```
package com.interpol;
import javax.swing.*;

public class ProxyArchive implements Archive {
    private int agentLevel;
    private static final ProxyArchive instance;

    static {
        instance = new ProxyArchive();
    }

    public static ProxyArchive getInstance() { return instance; }

    @Override
    public SensitiveInformation getSensitiveInformation(PoliceMan agent, int criminalCode) {
        this.agentLevel = agent.getAgentLevel();
        if(checkAgentLevel())
            return RealArchive.getInstance().getSensitiveInformation(agent, criminalCode);
        else JOptionPane.showMessageDialog ( parentComponent: null,  message: "YOU ARE NOT ALLOW..");
        return null;
    }

    @Override
    public void getProfileCreator(PoliceMan agent, int criminalCode) {
        this.agentLevel = agent.getAgentLevel();
        if(checkAgentLevel())
            RealArchive.getInstance().getProfileCreator(agent, criminalCode);
        else JOptionPane.showMessageDialog ( parentComponent: null,  message: "YOU ARE NOT ALLOW..");

    }

    @Override
    public void setSensitiveInformation(PoliceMan agent, int criminalCode, String gps, String internetData) {
        this.agentLevel = agent.getAgentLevel();
        if(checkAgentLevel())
            RealArchive.getInstance().setSensitiveInformation(agent, criminalCode, gps, internetData);
        else JOptionPane.showMessageDialog ( parentComponent: null,  message: "YOU ARE NOT ALLOW..");
    }

    private boolean checkAgentLevel() { return IdentifyLevel.checkAgentLevel(agentLevel); }
}
```

Figure 2.11: Frammento di codice della classe ProxyRealArchive

Chapter 3

Testing

3.1 Unit Testing

Quando si parla di Unit Testing si intende la verifica di singole porzioni di codice, nel caso dell'Object-Oriented Programming (OOP) tipicamente saranno le singole classi o i singoli metodi. Una volta individuate le varie sezioni di codice si potrà procedere con i test che vengono detti test cases. Nel progetto è stato utilizzato il framework JUnit nella versione 5.0. In JUnit i test-case sono dei metodi anteposti dall'annotazione @Test (o eventualmente @BeforeEach oppure @After). Per ogni classe principale del progetto sono state create le rispettive classi di Test contenenti i test case relativi ai metodi della classe principale da testare. In ogni metodo di test vengono verificate delle asserzioni (asserts) elementari attraverso vari metodi offerti da JUnit stesso: assertEquals, assertTrue, assertFalse, assertNull etc. Le classi di Test realizzate nel progetto sono:

- RealArchiveTest
- PoliceManTest
- SergeantTest
- CriminalProfileTest
- CriminalRecordTest
- SensitiveInformationTest

3.1.1 RealArchiveTest

```
@BeforeEach
void setUp(){
    policeMan = new PoliceMan( name: "Mario", surname: "Rossi",  policeID: 1);
    sergeant = new Sergeant( name: "Paolo", surname: "Bianchi",  policeID: 2);
    RealArchive.getInstance().generate(policeMan, criminalCode: 99, name: "francesco", surname: "verdi",new Date(), nationality: "ital
            CriminalProfile.State.valueOf("WANTED"), gps: "parigi", internetData: "twitter",  creatorCode: 1);
}

@Test
void setSensitiveInformation() {
    ProxyArchive.getInstance().setSensitiveInformation(policeMan, criminalCode: 99, gps: "florence", internetData: "youtube");
    assertNull(ProxyArchive.getInstance().getSensitiveInformation(policeMan,  criminalCode: 99));
    ProxyArchive.getInstance().setSensitiveInformation(sergeant, criminalCode: 99, gps: "rome", internetData: "google");
    assertEquals(ProxyArchive.getInstance().getSensitiveInformation(sergeant,  criminalCode: 99).getGps(), actual: "rome");
    assertEquals(ProxyArchive.getInstance().getSensitiveInformation(sergeant,  criminalCode: 99).getInternetData(), actual: "goog
}

@Test
void getSensitiveInformation() {
    assertNull(ProxyArchive.getInstance().getSensitiveInformation(policeMan, criminalCode: 99));
    assertEquals(ProxyArchive.getInstance().getSensitiveInformation(sergeant, criminalCode: 99).getGps(), actual: "parigi");
    assertEquals(ProxyArchive.getInstance().getSensitiveInformation(sergeant,  criminalCode: 99).getInternetData(),  actual: "twi
}

@Test
void getProfileCreator() throws NoCriminalProfileAssociated {
    assertEquals(RealArchive.getInstance().search( criminalCode: 99).getCreatorCode(), actual: 1);
}

@Test
void setRecord() throws NoCriminalProfileAssociated {
    RealArchive.getInstance().setRecord( criminalCode: 99, description: "He is free");
    assertEquals(RealArchive.getInstance().search( criminalCode: 99).getRecord().getDescription(), actual: "He is free");
}

@Test
void getCriminalIdentity() throws NoCriminalProfileAssociated {
    assertEquals(RealArchive.getInstance().search( criminalCode: 99).getName(), actual: "francesco");
    assertEquals(RealArchive.getInstance().search( criminalCode: 99).getSurname(), actual: "verdi");
    assertNotEquals(RealArchive.getInstance().search( criminalCode: 99).getDateOfBirth(),new Date());
    assertEquals(RealArchive.getInstance().search( criminalCode: 99).getNationality(), actual: "italiano");
```

Figure 3.1: Frammento di codice della classe RealArchiveTest

3.1.2 PoliceManTest

```java
package com.interpol;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotEquals;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class PoliceManTest {
    PoliceMan agent;

    @BeforeEach
    void setUp() { agent = new PoliceMan( name: "Mario",  surname: "Rossi",  policeID: 89);

    @Test
    void getAgentLevel() { assertEquals(agent.getAgentLevel(),  actual: 0); }

    @Test
    void getName() { assertEquals(agent.getName(), actual: "Mario"); }

    @Test
    void getSurname() { assertEquals(agent.getSurname(),  actual: "Rossi"); }

    @Test
    void getPoliceID() { assertEquals(agent.getPoliceID(),  actual: 89); }

    @Test
    void setAgentLevel() {...}
}
```

Figure 3.2: Frammento di codice della classe PoliceManTest

3.1.3 SergeantTest

```java
package com.interpol;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotEquals;

class SergeantTest {
    PoliceMan agent;

    @BeforeEach
    void setUp() { agent = new Sergeant( name: "Mario", surname: "Rossi", policeID: 22); }

    @Test
    void getAgentLevel() { assertEquals(agent.getAgentLevel(), actual: 5); }

    @Test
    void getPoliceID() { assertEquals(agent.getPoliceID(), actual: 22); }

    @Test
    void setAgentLevel(){...}
```

Figure 3.3: Frammento di codice della classe SergeantTest

3.1.4 CriminalProfileTest

```java
package com.interpol;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotEquals;

import java.util.Date;

class CriminalProfileTest {
    CriminalProfile profile;
    @BeforeEach
    void setUp(){
        profile = new CriminalProfileBuilder( criminalCode: 1).name("Mario").surname("Rossi")
                .nationality("italiano").dateOfBirth(new Date()).setCreatorCode(99).profileState(CriminalProfile.State.valueOf("WANTED"))
    }

    @Test
    void getCreatorCode() { assertNotEquals(profile.getCreatorCode(), actual: 50); }

    @Test
    void getIdentity() {
        assertEquals(profile.getName(), actual: "Mario");
        assertEquals(profile.getSurname(), actual: "Rossi");
        assertEquals(profile.getNationality(), actual: "italiano");
        assertNotEquals(profile.getDateOfBirth(), new Date());
        assertEquals(profile.getCriminalCode(), actual: 1);
    }

    @Test
    void getCriminalCode() { assertEquals(profile.getCriminalCode(), actual: 1); }

    @Test
    void getState() { assertEquals(profile.getState().toString(), actual: "WANTED"); }

    @Test
    void setCreatorCode() {
        profile.setCreatorCode(9);
        assertEquals(profile.getCreatorCode(), actual: 9);
    }
```

Figure 3.4: Frammento di codice della classe CriminalProfileTest

3.1.5 CriminalRecordTest

```java
package com.interpol;

import ...

class CriminalRecordTest {

    CriminalRecord record;

    @BeforeEach
    void setUp(){
        record = new CriminalRecord();
        record.setDescription("He made a crime");
    }

    @Test
    void getDescription() { assertEquals(record.getDescription(), actual: "He made a crime

    @Test
    void setDescription() {
        record.setDescription("He is free");
        assertEquals(record.getDescription(), actual: "He is free");
    }
}
```

Figure 3.5: Frammento di codice della classe CriminalRecordTest

3.1.6 SensitiveInformationTest

```
package com.interpol;

import ...

class SensitiveInformationTest {
    SensitiveInformation information;

    @BeforeEach
    void setUp(){
        information = new SensitiveInformation();
        information.setGps("florence");
        information.setInternetData("youtube");
    }

    @Test
    void getGps() { assertEquals(information.getGps(), actual: "florence"); }

    @Test
    void setGps() {
        information.setGps("rome");
        assertNotEquals(information.getGps(), actual: "florence");
    }

    @Test
    void getInternetData() { assertEquals(information.getInternetData(), actual: "youtub

    @Test
    void setInternetData() {
        information.setInternetData("google");
        assertNotEquals(information.getInternetData(), actual: "youtube");
```

Figure 3.6: Frammento di codice della classe SensitiveInformationTest