UNIVERSITÀ DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria

Corso di laurea in Ingegneria Informatica

Elaborato di Ingegneria del Software

Pietro Longinetti

"FIRE'S OUT"

Applicazione Java che simula una gestione efficiente dell'estinzione di incendi in una zona monitorata.

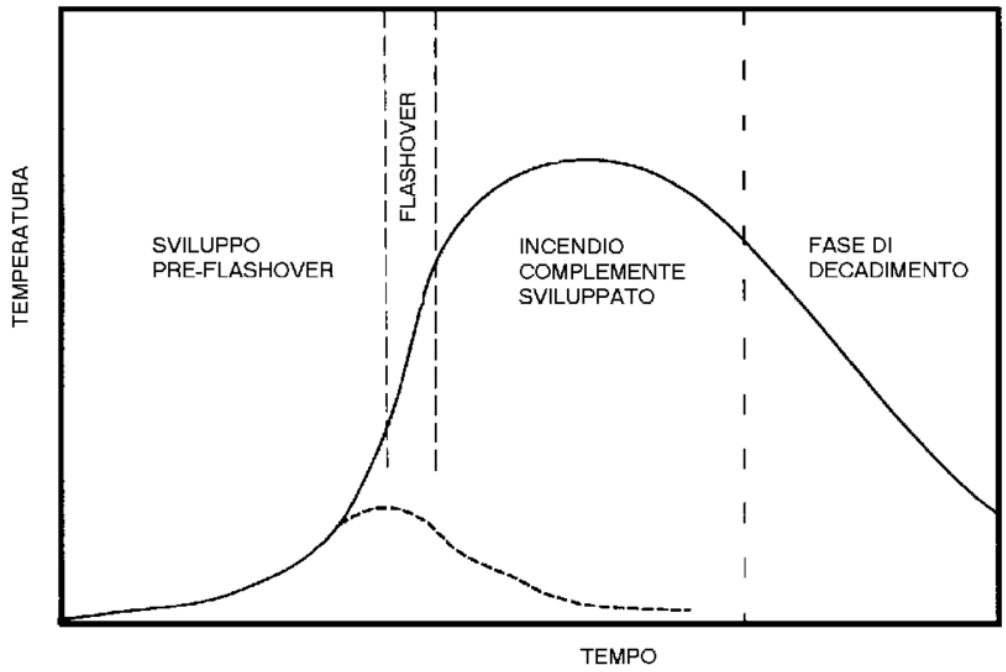
A.A. 2019-202

Contenuti

Capitolo 1 - Introduzione

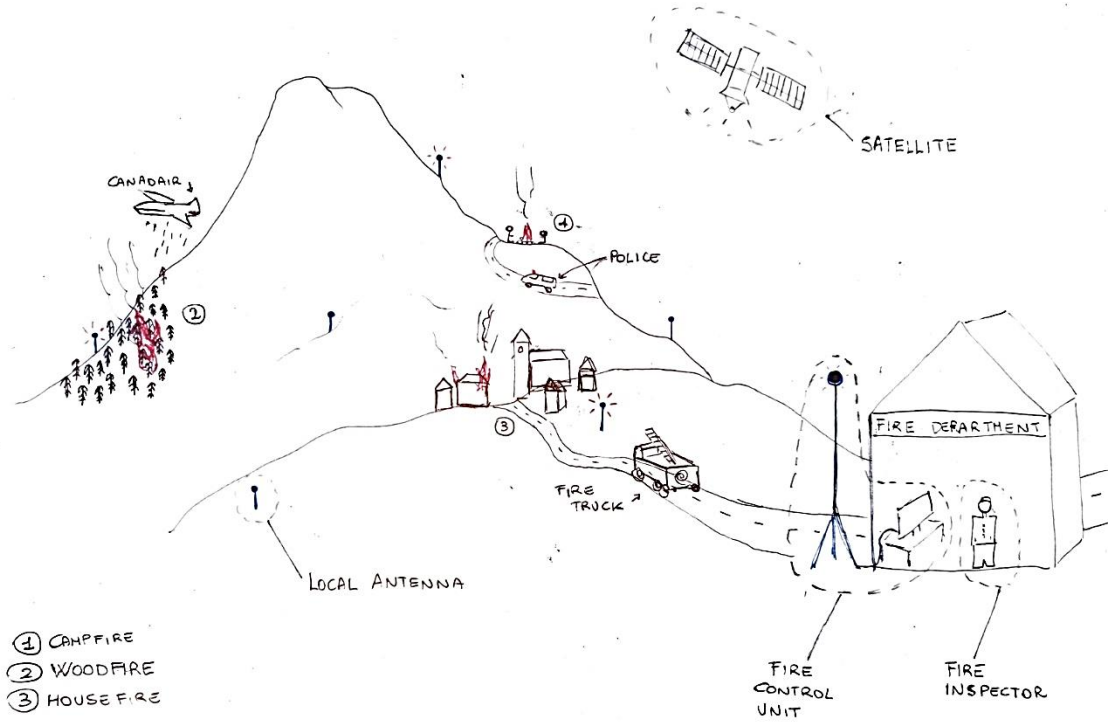
1.1 Motivazioni

Il presente elaborato è incentrato su una possibile soluzione ad un problema molto attuale: il divampare degli incendi dolosi in zone protette o parchi naturali, che, specialmente nella stagione estiva, finisce per deturpare gli habitat e gli ecosistemi colpiti, riducendo a carbone interi ettari di bosco e distese erbose.
Da qui l'idea di progettare una sistema automatizzato ed efficiente che riesca a domare rapidamente un incendio già dai primi momenti in cui si sta sviluppando, cioè durante la fase di **ignizione** o, alla peggio, **propagazione**.
La motivazione di ciò sta nel fatto che tanto prima si riesce ad intervenire, quanto più un incendio è facile da domare. Di fatti, quando un incendio accresce troppo e raggiunge il punto di *"Flash Over"* (punto critico in cui l'incendio diventa pienamente generalizzato), diventa estremamente difficile da estinguere (si vedano i grandi incendi in Australia 2019-2020 o in Amazonia 2019) e così per riuscire a sconfiggerlo nel peggiore dei casi bisogna aspettare l'avvento della stagione piovosa.



1.2 Funzionamento

L'elaborato in questione presenta un'ipotesi di lavoro molto importante: la zona che si vuole tenere al sicuro da incendi (un parco naturale, una zona ad alto rischio, ecc…) deve essere monitorata costantemente da dei rilevatori di temperatura che tramite un'antenna comunicheranno con una centralina adibita al controllo, la quale a sua volta viene tenuta sotto sorveglianza da un ispettore. Nel momento in cui un incendio viene appiccato, il rilevatore di temperatura invierà il proprio valore di temperatura (anomalo) all'unità di controllo, la quale registrerà il settore da dove è provenuto il segnale "critico", e comunicherà all' ispettore la presenza di possibili incendi. L'ispettore richiederà all'unità di controllo, che a sua volta inoltrerà la richiesta ad un satellite, di fornirgli una fotografia satellitare della zona dove è stata segnalata un'alta temperatura e, dopo aver visualizzato tale fotografia, l'ispettore deciderà quale strategia utilizzare per domare l'incendio:



1.3 Metodi di implementazione

L'applicativo in questione è stato realizzato con il linguaggio Java, attraverso l'uso dell'IDE IntelliJ IDEA.
Sono state implementate sette classi diverse, più la classe *Main* con al suo interno la funzione omonima che ha il compito di inizializzare gli oggetti e quello di simulare le variazioni di temperatura nella zona.
Le classi sono state organizzate nella cartella di default "*src*" e non è stato ritenuto necessario suddividere i pochi file in ulteriori packages. Esse collaborano tra loro tramite l'uso di pattern. In questo progetto ne sono stati utilizzati cinque. Nello specifico:

Pattern	*Tipo*
Observer (Pull)	Comportamentale
Observer (Push)	Comportamentale
Singleton	Creazionale
Proxy	Strutturale
Strategy	Comportamentale

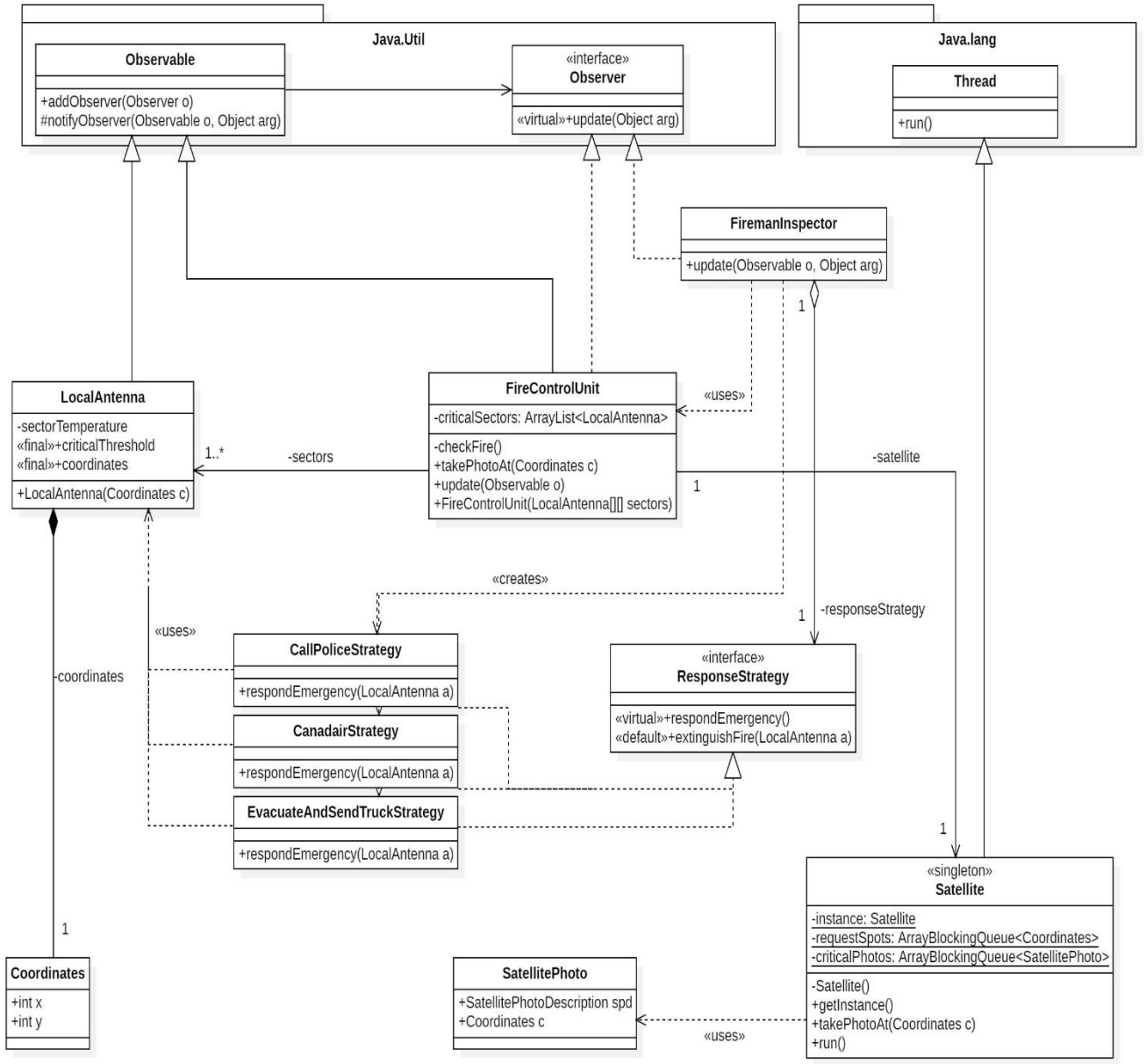
Per illustrare al meglio la logica di dominio e il funzionamento dell'applicazione sono stati realizzati un **Class Diagram** e un **Sequence Diagram** UML entrambi tramite l'utilizzo del software di supporto *StarUML*.

Infine, sono stati realizzati dei **test** utilizzando il framework di unit test *JUnit 5*. Questi sono stati organizzati per classi e sono contenuti nella cartella "*tests*".

Capitolo 2 - Progettazione

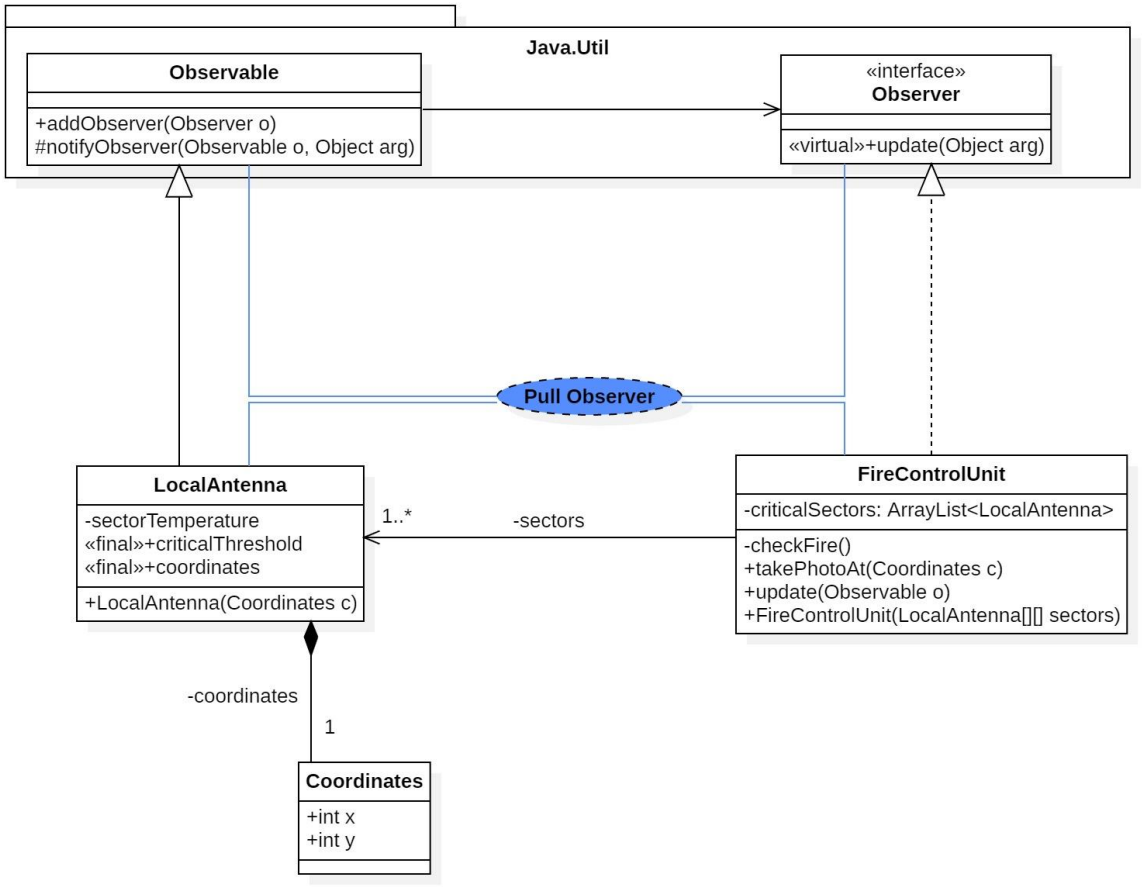
2.1 Class Diagram

Come già anticipato nel paragrafo precedente, si riporta qui sotto la realizzazione del diagramma UML che descrive la logica di dominio in prospettiva di implementazione:



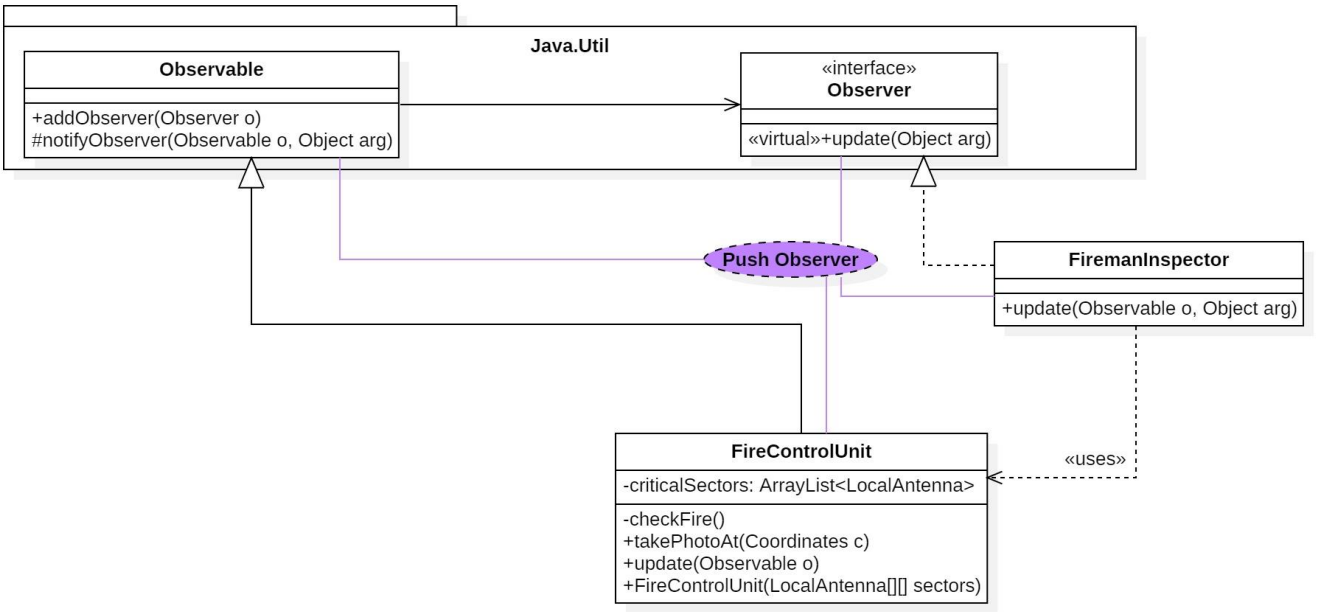
2.2 Pull Observer Pattern

In questa parte dello schema le classi *LocalAntenna* e *FireControlUnit* collaborano secondo uno schema **Observer** di tipo **Pull**: ogni Antenna ha un Observer registrato e gli notifica l'avvenuto cambiamento dello stato di temperatura invocando il metodo *notifyObservers()* della super-classe. L'Unità di Controllo aggiorna il suo stato implementando il metodo *update()* dell'interfaccia *Observer* andando a recuperare i dati di cui ha bisogno attraverso il riferimento a *Observable* passato come parametro *(pulling)*.



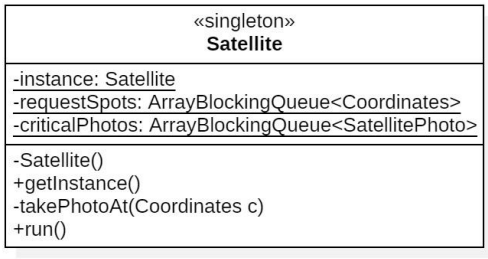
2.3 Push Observer Pattern

Lo schema in questione è quasi interamente identico a quello precedente, con l'unica differenza che il metodo *notifyObservers()* adesso passa come parametro alla classe FiremanInspector un argomento *(arg)* non nullo. Nello specifico questo parametro conterrà una lista di antenne che hanno rilevato una temperatura anomala:



2.4 Singleton Pattern

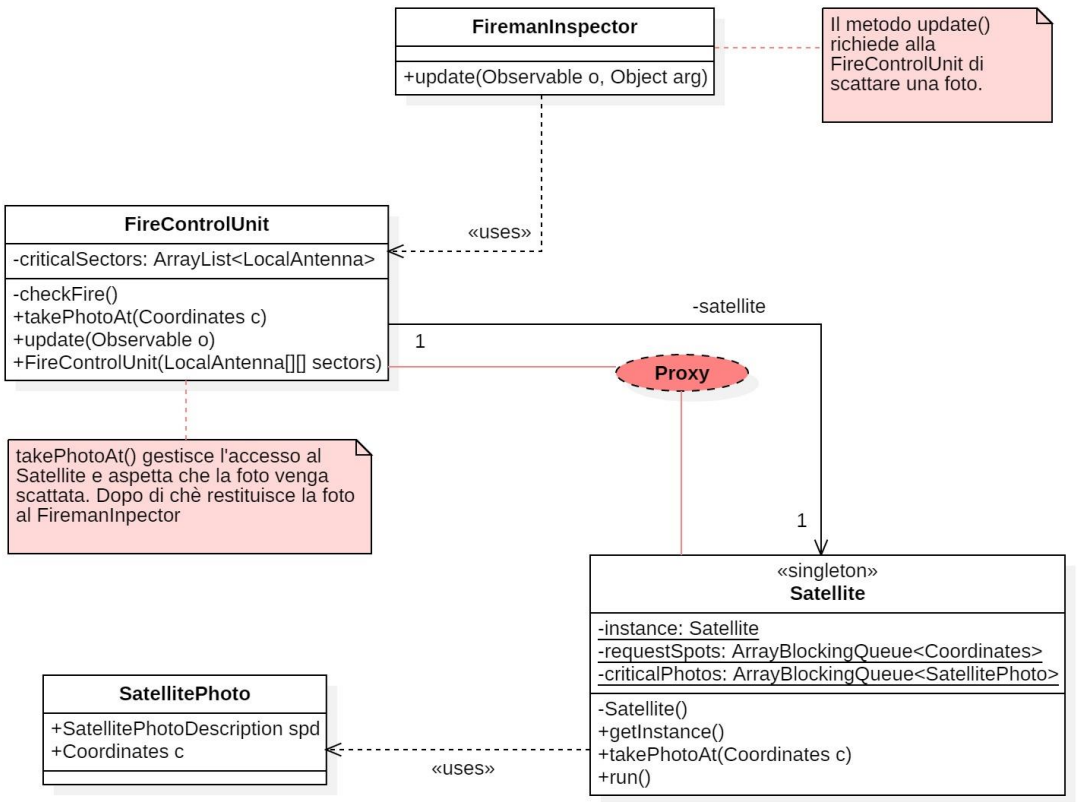
In una concezione realistica dell'elaborato si assume che il software possa comunicare con uno ed un solo satellite e che l'allocazione della sua istanza costituisca una procedura molto pesante da elaborare.
Da qui la scelta di utilizzare il Singleton Pattern con strategia *"Deferred Creation"*.



2.5 Proxy Pattern

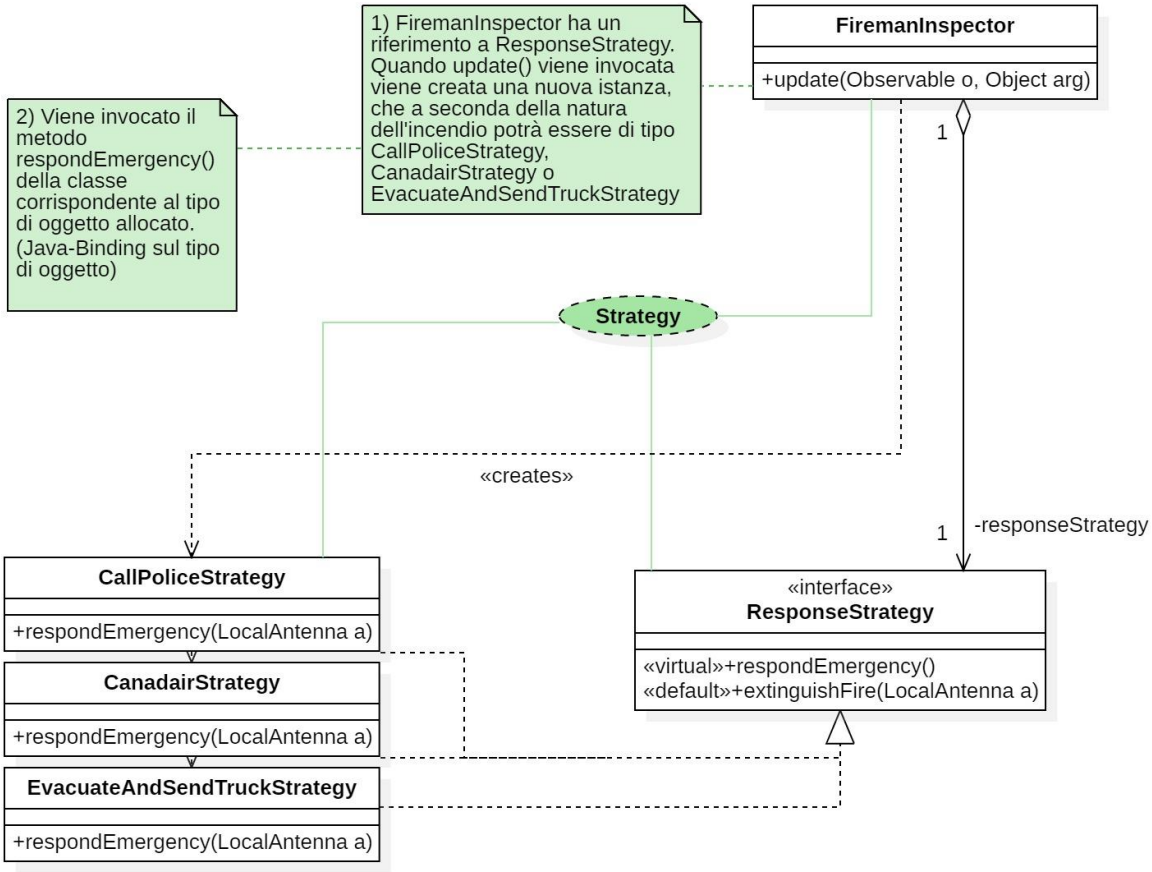
Il satellite inoltre, processa le sue istruzioni in maniera autonoma e di fatto eredita dalla classe *Java.lang.Thread*. Affinché l'ispettore possa ottenere una foto satellitare del settore dove è stata rilevata un'alta temperatura, l'accesso al satellite deve essere gestito dalla *FireControlUnit* secondo un protocollo in cui viene fatta una richiesta, il satellite la processa e restituisce, non appena ha finito di eseguire la sua routine di scatto della foto, il valore di ritorno. Ecco perché l'esigenza di utilizzare uno schema di tipo Proxy.

Secondo tale convenzione il *"client"* che richiede l'accesso è rappresentato dal FiremanInspector, il *"proxy"* è la FireControlUnit, mentre il satellite costituisce il *"subject"* da cui si vuole ottenere il dato.



2.6 Strategy Pattern

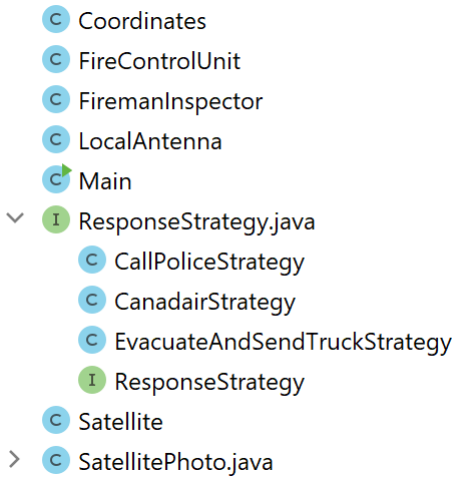
Dopo aver analizzato la foto, l'ispettore decide che strategia adottare per spegnere l'incendio. Alloca quindi una nuova istanza di una delle classi che implementano l'interfaccia *ResponseStrategy*, e successivamente invoca il metodo *respondEmergency()*.



Capitolo 3 - Implementazione

3.1 Classi ed interfacce realizzate

Come già anticipato nella sezione 1.3 Metodi ed Implementazione, sono state sia definite nuove interfacce e classi, che riutilizzate quelle della libreria standard di Java (e.g classi e interfacce contenute nel package Java.Util o Java.lang):



3.2 *Coordinates*

Una semplice classe progettata per rappresentare le coordinate in una griglia bidimensionale:

```
public class Coordinates {
    final int x;
    final int y;

    public Coordinates(int x, int y) {...}
}
```

3.3 *LocalAntenna*

Classe che rappresenta un'unità di rilevazione della temperatura in un certo settore dell'area monitorata. La classe *LocalAntenna* sarà composta da una coordinata, da un valore di temperatura rilevato e da una soglia di temperatura critica:

```
public class LocalAntenna extends Observable {
    final Coordinates coordinates;
    final static int criticalThreshold = 80;  //degrees
    private int sectorTemperature;  //degrees
    private int increment;      //degrees

    public LocalAntenna(Coordinates coordinates) {
        this.coordinates = coordinates;

        /* Il costruttore genera un valore casuale di temperatura rilevata già al
        momento della creazione */
        Random r = new Random();
        sectorTemperature = 25 + r.nextInt( bound: 10) - r.nextInt( bound: 10);
        setChanged();
        notifyObservers();
    }
    public int getSectorTemperature() {
        return sectorTemperature;
    }
    public int getIncrement() {
        return increment;
    }

    /* Funzione che non dovrebbe poter essere accessibile da classi esterne ma che deve
    necessariamente esserlo per poter simulare una reale variazione della temperatura.*/
    public void setSectorTemperature(int sectorTemperature) {
        this.increment = sectorTemperature - this.sectorTemperature;
        this.sectorTemperature = sectorTemperature;
        setChanged();
        notifyObservers();
    }
}
```

3.4 *FireControlUnit*

Probabilmente la classe più importante del progetto. Si occupa di raccogliere i dati dalle antenne e di avvisare l'ispettore di uno o più eventuali incendi appiccati. Ha inoltre il compito di gestire l'accesso al satellite:

```
public class FireControlUnit extends Observable implements Observer {
    private Satellite satellite;
    private int antennasUpdated;
    private final LocalAntenna[][] sectors;
    private final ArrayList<LocalAntenna> criticalSectors;
```

Metodo di aggiornamento dei valori:

```
public void update(Observable o, Object arg) {
    //O corrisponde alla antenna che ha segnalato una variazione di temperatura
    //Arg è nullo -> Pull Observer
    LocalAntenna s = (LocalAntenna) o;
    if (s.getSectorTemperature() > LocalAntenna.criticalThreshold) {
        criticalSectors.add(s);
    }

    //Gestione dello spegnimento di un incendio
    if (s.getSectorTemperature() - s.getIncrement() > LocalAntenna.criticalThreshold) {
        System.out.println("Incendio domato nel settore " + s.coordinates.x + s.coordinates.y + "\n");
        criticalSectors.remove(s);
        if (criticalSectors.isEmpty()) {
            System.out.println("Tutti gli incendi domati. Attuale situazione: ");
            printGrid();
        }
    }

    //Quando tutti i valori della griglia sono stati aggiornati viene mostrata
    //la griglia di valori aggiornata e il programma controlla se ci sono incendi
    antennasUpdated++;
    if (antennasUpdated == Math.pow(sectors.length, 2)) {
        System.out.println();
        printGrid();
        checkFires();
        antennasUpdated = 0;
    }
}
```

Metodo di notifica all'ispettore (invocato da update):

```
private void checkFires(){
    for (LocalAntenna a:criticalSectors) {
        System.out.println("Incendio rilevato nel settore " + a.coordinates.x + a.coordinates.y);
    }
    if(!criticalSectors.isEmpty()) {
        setChanged();
        notifyObservers(new ArrayList<>(criticalSectors));  //Copia difensiva
    }
```

Metodo Proxy di richiesta di scatto della foto satellitare:

```
public SatellitePhoto takePhotoAt(Coordinates c) throws InterruptedException {
    //Proxy method
    try {...} catch (InvalidParameterException e){...}
    if(satellite == null){
        satellite = Satellite.getInstance();
        satellite.start();
    }
    Satellite.getRequestSpots().offer(c);
    //il metodo take() aspetta che ci sia almeno un elemento nella coda di foto scattate dal satellite
    return Satellite.getCriticalPhotos().take();
}
```

3.5 *FiremanInspector*

La classe che si occupa di gestire i settori critici avviando il procedimento di scatto della foto tramite l'unità di controllo e in seguito sviluppare una strategia per domare gli incendi:

```
public class FiremanInspector implements Observer{
    private ResponseStrategy responseStrategy;
    public ResponseStrategy getResponseStrategy() { return responseStrategy; }

    @Override
    public void update(Observable o, Object arg) {
        //O è la FireControlUnit
        //Arg invece è una lista di antenne che si trovano nei settori con incendio
        for (LocalAntenna spot: (ArrayList<LocalAntenna>)arg) {
            try {
                SatellitePhoto situation = ((FireControlUnit)o).takePhotoAt(spot.coordinates);
                System.out.println("Dalla foto scattata dal satellite si vede chiaramente che l'alta temperatura"+
                        " rilevata nel settore " + situation.spot.x + situation.spot.y +
                        " è dovuta a: " + situation.description);
                switch (situation.description) {
                    case CampFire: {
                        responseStrategy = new CallPoliceStrategy();
                        break;
                    }
                    case WoodFire: {
                        responseStrategy = new CanadairStrategy();
                        break;
                    }
                    case HouseFire: {
                        responseStrategy = new EvacuateAndSendTruckStrategy();
                    }
                }
                responseStrategy.respondEmergency(spot);
            } catch (InterruptedException e) {}
        }
    }
}
```

3.6 *Satellite*

L'unica classe che estende *Java.lang.Thread* e che gode della libertà di avere un thread separato:

```
public class Satellite extends Thread{
    private static ArrayBlockingQueue<Coordinates> requestSpots;
    private static ArrayBlockingQueue<SatellitePhoto> criticalPhotos;
    private static Satellite instance = null;
    private Satellite() {}

    public static Satellite getInstance(){
        if(instance == null) {
            instance = new Satellite();
            requestSpots = new ArrayBlockingQueue<Coordinates>( capacity: 1);
            criticalPhotos = new ArrayBlockingQueue<SatellitePhoto>( capacity: 1);
        }
        return instance;
    }
    public static ArrayBlockingQueue<Coordinates> getRequestSpots() { return requestSpots; }
    public static ArrayBlockingQueue<SatellitePhoto> getCriticalPhotos() { return criticalPhotos; }

    private void takePhotoAt(Coordinates c){
        Random r = new Random();
        int randomPhotoIndex = r.nextInt(SatellitePhotoDescription.values().length);
        criticalPhotos.offer(new SatellitePhoto(SatellitePhotoDescription.values()[randomPhotoIndex], c));
        System.out.println("Il satellite ha scattato una foto nel settore "+ c.x + c.y);
    }

    @Override
    public void run() {
        while(true) {
            try{
                //Quando la coda è vuota il satellite resta in attesa di una "richiesta".
                //La richiesta è rappresentata dagli elementi (Coordinate) presenti nella coda requestSpots
                takePhotoAt(requestSpots.take());
            } catch (InterruptedException e){}
        }
    }
```

3.7 *SatellitePhoto*

Una semplice classe che collega la descrizione di una foto con la sua posizione sulla griglia. Al fine di facilitare l'**estensione**, la descrizione di una foto è stata implementata tramite una "*enum*" di possibili situazioni verificabili.

```
enum SatellitePhotoDescription{CampFire, WoodFire, HouseFire}

public class SatellitePhoto {
    public SatellitePhotoDescription description;
    public Coordinates spot;

    public SatellitePhoto(SatellitePhotoDescription description, Coordinates spot) {...}
}
```

3.8 *ResponseStrategy*

L'interfaccia che, attraverso le sue implementazioni, permette tramite il pattern Strategy di scegliere la strategia più adatta per estinguere l'incendio nella sua fase iniziale:

```
public interface ResponseStrategy {
    void respondEmergency(LocalAntenna antenna);
    default void extinguishFire(LocalAntenna a){
        Random r = new Random();
        a.setSectorTemperature(30 + r.nextInt( bound: 5) - r.nextInt( bound: 5));
    }
}

class CallPoliceStrategy implements ResponseStrategy{
    @Override
    public void respondEmergency(LocalAntenna antenna) {
        System.out.println("La polizia viene contattata, una pattuglia sopraggiunge nel settore "
                + antenna.coordinates.x + antenna.coordinates.y +
                " e intima alle persone di spegnere il fuoco e di tornarsene a casa.");
        try {...} catch (InterruptedException e){} //Tempo tecnico per domare l'incendio
        extinguishFire(antenna);
    }
}

class CanadairStrategy implements ResponseStrategy{
    @Override
    public void respondEmergency(LocalAntenna antenna) {
        System.out.println("Un aereo-cisterna decolla e rilascia a pioggia sul settore " + antenna.coordinates.x +
                antenna.coordinates.y + " una quantità d'acqua sufficiente a spegnere un piccolo incendio");
        try {...} catch (InterruptedException e){} //Tempo tecnico per domare l'incendio
        extinguishFire(antenna);
    }
}

class EvacuateAndSendTruckStrategy implements ResponseStrategy{
    @Override
    public void respondEmergency(LocalAntenna antenna) {
        System.out.println("I residenti nel settore " + antenna.coordinates.x + antenna.coordinates.y +
                " vengono contattati e fatti evacuare, nel frattempo i camion-cisterna dei vigili del " +
                "fuoco arrivano sul luogo e domano l'incendio");
        try {...} catch (InterruptedException e){} //Tempo tecnico per domare l'incendio
        extinguishFire(antenna);
    }
}
```

3.9 Output e Sequence Diagram

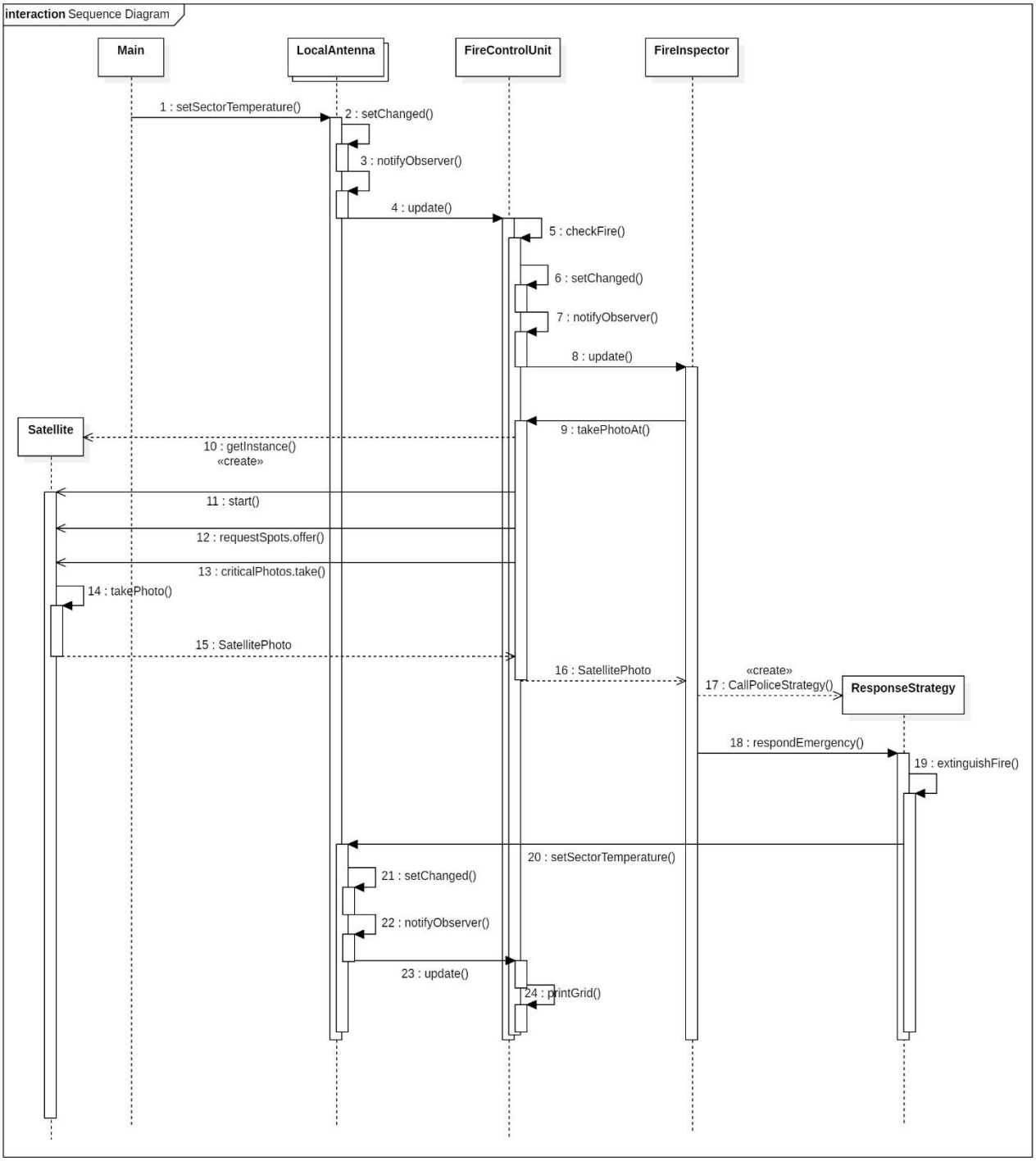
Di sequito viene riportato un l'output e il corrispondente Sequence Diagram di un possibile flusso d'esecuzione del programma. Si ricorda che la funziona Main oltre che ad inizializzare tutte le classi e stampare la configurazione iniziale, gestisce anche le variazioni di temperatura nella maniera illustrata:

```
//Simulazione  delle variazioni di temperatura
while (true) {
    for (int i = 0; i < antennasGrid.length; i++) {
        Random r = new Random();
        for (int j = 0; j < antennasGrid.length; j++) {
            LocalAntenna current = antennasGrid[i][j];
            int dice = r.nextInt( bound: 100);

            if (dice != 0) {
                //Simulazione  di piccole e normali variazioni di temperatura
                current.setSectorTemperature(current.getSectorTemperature() + (r.nextInt() % 3));
            } else /*dice == 0 */ {
                //Simulazione  di un aumento di temperatura corrispondente ad un incendio
                current.setSectorTemperature(current.getSectorTemperature() + 50 + r.nextInt( bound: 100));
            }
        }
    }
    sleep( millis: 2000);
```

Output:

```
Inserire il numero di antenne che si hanno a disposizione per il monitoraggio della zona:
20
Configurazione iniziale:
A00:25°(+0)    A01:26°(+0)    A02:22°(+0)    A03:25°(+0)
A10:21°(+0)    A11:22°(+0)    A12:27°(+0)    A13:30°(+0)
A20:31°(+0)    A21:24°(+0)    A22:25°(+0)    A23:25°(+0)
A30:26°(+0)    A31:30°(+0)    A32:24°(+0)    A33:27°(+0)

Avvio del programma di monitoraggio...

A00:24°(-1)    A01:26°(+0)    A02:22°(+0)    A03:23°(-2)
A10:21°(+0)    A11:24°(+2)    A12:28°(+1)    A13:29°(-1)
A20:31°(+0)    A21:26°(+2)    A22:23°(-2)    A23:24°(-1)
A30:26°(+0)    A31:31°(+1)    A32:24°(+0)    A33:29°(+2)

A00:22°(-2)    A01:25°(-1)    A02:23°(+1)    A03:23°(+0)
A10:21°(+0)    A11:26°(+2)    A12:28°(+0)    A13:29°(+0)
A20:30°(-1)    A21:26°(+0)    A22:116°(+93)    A23:22°(-2)
A30:24°(-2)    A31:31°(+0)    A32:26°(+2)    A33:30°(+1)
Incendio rilevato nel settore 22
Il satellite ha scattato una foto nel settore 22
Dalla foto scattata dal satellite si vede chiaramente che l'alta temperatura rilevata nel settore 22 è dovuta a: CampFire
La polizia viene contattata, una pattuglia sopraggiunge nel settore 22 e intima alle persone di spegnere il fuoco e di tornarsene a casa.
Incendio domato nel settore 22

Tutti gli incendi domati. Attuale situazione:
A00:22°(+0)    A01:25°(+0)    A02:23°(+0)    A03:23°(+0)
A10:21°(+0)    A11:26°(+0)    A12:28°(+0)    A13:29°(+0)
A20:30°(+0)    A21:26°(+0)    A22:31°(-85)    A23:22°(+0)
A30:24°(+0)    A31:31°(+0)    A32:26°(+0)    A33:30°(+0)
```

Sequence Diagram:



Capitolo 4 – Unit Testing

Allo scopo di assicurare il corretto funzionamento delle classi e dei loro metodi è stato utilizzato il framework **JUnit 5**, che ha permesso di implementare in modo rapido lo Unit Testing su porzioni del codice.

Per ogni principale classe del progetto sono state create quindi le rispettive classi di Test contenenti i test case relativi ai metodi della classe principale da testare. In JUnit i metodi che effettuano un'operazione di test sono sempre preceduti dall'annotazione *@Test*. L'annotazione *@BeforeEach* permette invece di eseguire una particolare funzione (di solito quella di inizializzazione) prima dell'esecuzione di ogni funzione di test.

Infine, JUnit fornisce all'utente alcuni metodi elementari come:

- *assertEquals*
- *assertTrue*
- *assertNull*

utili per verificare la correttezza di alcune condizioni determinanti per capire se i metodi e le classi funzionano nella maniera in cui dovrebbero.

Le classi di test realizzate nel progetto sono:

1. *LocalAntennaTest*
2. *FireControlUnitTest*
3. *FiremanInspectorTest*
4. *SatelliteTest*
5. *ResponseStrategyTest*

4.1 *LocalAntennaTest*

Per questa classe sono stati effettuati prima i test classici per verificare che al momento della creazione di un oggetto di tipo *LocalAntenna* gli attributi siano stati assegnati in modo corretto. Le ultime due funzioni di test invece verificano che ad un corretto funzionamento del metodo *setTemperatureValue* corrisponda un corretto aggiornamento dei valori di temperatura ed incremento.

```
class LocalAntennaTest {
    LocalAntenna a;

    @BeforeEach
    void setUp(){
        Coordinates c = new Coordinates( x: 0, y: 0);
        a = new LocalAntenna(c);
    }
    @Test
    void getCoordinates(){
        assertEquals( expected: 0, a.coordinates.x);
        assertEquals( expected: 0, a.coordinates.y);
    }
    @Test
    void getCriticalThreshold(){
        assertEquals( expected: 80, LocalAntenna.criticalThreshold);
    }
    @Test
    void hasChangedValuesOnCreation(){
        assertTrue( condition: a.getSectorTemperature() > 0);
        assertEquals( expected: 0, a.getIncrement());
    }
    @Test
    void hasChangedTemperatureValue(){
        int oldTemperature = a.getSectorTemperature();
        a.setSectorTemperature(a.getSectorTemperature() + 5);
        assertEquals( expected: oldTemperature + 5, a.getSectorTemperature());
    }
    @Test
    void hasChangedIncrementValue(){
        int oldTemperature = a.getSectorTemperature();
        a.setSectorTemperature(0);
        assertEquals(-oldTemperature, a.getIncrement());
    }
}
```

4.2 *FireControlUnitTest*

Dal momento che la classe *FireControlUnit* deve gestire molteplici compiti, sono state testate più funzionalità.

È stata verificata la corretta inizializzazione di tutti gli attributi al momento della creazione dell'istanza:

```
class FireControlUnitTest {
    FireControlUnit fcu;

    @BeforeEach
    void setUp(){...}
    @Test
    void correctlyInitialized(){
        assertNotNull(fcu.getSectors());
        assertNotNull(fcu.getCriticalSectors());
        for (int i = 0; i < fcu.getSectors().length; i++) {
            for (int j = 0; j < fcu.getSectors().length; j++) {
                assertTrue( condition: fcu.getSectors()[i][j].getSectorTemperature()>0);
            }
        }
        assertTrue(fcu.getCriticalSectors().isEmpty());
    }
```

È stato verificato che l'unità di controllo riesce ad identificare e inserire nella lista *criticalSectors* uno o più settori critici dove è stata rilevata un'alta temperatura:

```
@Test
void hasSpottedFire(){
    for (int i = 0; i < fcu.getSectors().length; i++) {
        for (int j = 0; j < fcu.getSectors().length; j++) {
            if(i == 0 && j == 0)
                //La prima antenna simula un incendio
                fcu.getSectors()[i][j].setSectorTemperature(81);
            else fcu.getSectors()[i][j].setSectorTemperature(fcu.getSectors()[i][j].getSectorTemperature());
        }
    }
    assertEquals( expected: 1, fcu.getCriticalSectors().size());
}
@Test
void hasSpottedMultipleFires(){
    for (int i = 0; i < fcu.getSectors().length; i++) {
        for (int j = 0; j < fcu.getSectors().length; j++) {
            fcu.getSectors()[i][j].setSectorTemperature(81);
        }
    }
    assertEquals(Math.pow(fcu.getSectors().length, 2), fcu.getCriticalSectors().size());
}
```

È stato testato il corretto funzionamento del meccanismo di acquisizione di fotografie satellitari. Inoltre, è stato eseguito un "crash test" in modo da verificare che quando si chiede al satellite di scattare una foto in una posizione che non esiste sulla griglia viene lanciata un'eccezione e la fotografia non viene acquisita:

```
@Test
void hasTakenPhotoAtLegalPosition() throws InterruptedException{
    Random r = new Random();
    Coordinates c = new Coordinates(r.nextInt(fcu.getSectors().length),r.nextInt(fcu.getSectors().length));
    fcu.getSectors()[c.x][c.y].setSectorTemperature(81);
    SatellitePhoto sp = fcu.takePhotoAt(c);
    assertNotNull(sp);
    assertNotNull(sp.description);
    assertNotNull(sp.spot);
}
@Test
void hasFailedTakingPhotoAtIllegalPosition() throws InterruptedException{
    SatellitePhoto sp = fcu.takePhotoAt(new Coordinates( x: -1, y: 0));
    assertNull(sp);
    sp = fcu.takePhotoAt(new Coordinates( x: 0, y: -1));
    assertNull(sp);
    sp = fcu.takePhotoAt(new Coordinates( x: fcu.getSectors().length +1, y: 0));
    assertNull(sp);
    sp = fcu.takePhotoAt(new Coordinates( x: 0, y: fcu.getSectors().length +1));
    assertNull(sp);
}
```

4.3 *FiremanInspectorTest*

La classe *FiremanInspectorTest* verifica con il primo test che al momento della allocazione di un oggetto di tipo *FiremanInspector* nessuna strategia è ancora stata creata. Col secondo invece, verifica che una strategia viene creata dopo che lo scoppio di un incendio viene notificato:

```
class FiremanInspectorTest {
    FireControlUnit fcu;
    FiremanInspector fi;

    @BeforeEach
    void setUp(){...}

    @Test
    void noStrategyOnCreation() { assertNull(fi.getResponseStrategy()); }

    @Test
    void elaboratesStrategyWhenFiresNotified(){
        Random r = new Random();
        Coordinates c = new Coordinates(r.nextInt(fcu.getSectors().length), r.nextInt(fcu.getSectors().length));
        for (int i = 0; i < fcu.getSectors().length; i++) {
            for (int j = 0; j < fcu.getSectors().length; j++) {
                if(i == c.x && j == c.y)
                    //La antenna alle coordinate c simula un incendio
                    fcu.getSectors()[i][j].setSectorTemperature(81);
                else fcu.getSectors()[i][j].setSectorTemperature(fcu.getSectors()[i][j].getSectorTemperature());
            }
        }
        assertNotNull(fi.getResponseStrategy());
    }
```

4.4 *SatelliteTest*

Per classe *SatelliteTest* sono state nuovamente testate le funzioni che verificano la corretta inizializzazione di un oggetto istanziato di tipo *Satellite*:

```
class SatelliteTest{
    Satellite satellite;

    @BeforeEach
    void setUp() { satellite = Satellite.getInstance(); }
    @Test
    void initialized(){
        assertNotNull(satellite);
        assertNotNull(Satellite.getRequestSpots());
        assertNotNull(Satellite.getCriticalPhotos());
    }
    @Test
    void emptyQueuesOnCreation(){
        assertTrue(Satellite.getRequestSpots().isEmpty());
        assertTrue(Satellite.getCriticalPhotos().isEmpty());
    }
```

E in seguito è stato testato il meccanismo di richiesta, scatto e acquisizione della fotografia satellitare. Il meccanismo fa uso di una *ArrayBlockingQueue,* una classe contenuta nel pacchetto Java.util che permette di gestire con facilità l'accesso ad una coda di dati nella programmazione concorrente:

- Il metodo *offer(Object o)* permette di inserire un elemento nella coda nel caso in cui non sia piena.
- Il metodo *take()* invece permette di acquisire un elemento in testa alla coda. Ma se la coda è vuota, il thread che ha invocato tale metodo resta in attesa che un elemento venga inserito.

```
@Test
void photoShootingMechanism(){
    Coordinates c = new Coordinates( x: 0, y: 0);
    Satellite.getRequestSpots().offer(c);

    assertEquals( expected: 1, Satellite.getRequestSpots().size());
    assertEquals( expected: 0, Satellite.getCriticalPhotos().size());

    satellite.start();
    try {
        sleep( millis: 1000);

        assertEquals( expected: 0, Satellite.getRequestSpots().size());
        assertEquals( expected: 1, Satellite.getCriticalPhotos().size());

        Satellite.getCriticalPhotos().take();

        assertEquals( expected: 0, Satellite.getRequestSpots().size());
        assertEquals( expected: 0, Satellite.getCriticalPhotos().size());

    }catch (InterruptedException e){}
}
```

4.5 *ResponseStrategyTest*

Dal momento che le classi che implementano l'interfaccia ResponseStrategy non hanno attributi, la classe ResponseStrategyTest testa soltanto che il meccanismo di spegnimento degli incendi vada a buon fine:

```
class ResponseStrategyTest {
    ResponseStrategy rs;
    LocalAntenna a;

    @BeforeEach
    void setUp() { a = new LocalAntenna(new Coordinates( x: 0, y: 0)); }
    @Test
    void callPoliceEmergencyResponse(){
        rs = new CallPoliceStrategy();
        a.setSectorTemperature(81);
        rs.respondEmergency(a);

        assertTrue( condition: a.getSectorTemperature() < LocalAntenna.criticalThreshold);
    }
    @Test
    void canadairEmergencyResponse(){
        rs = new CanadairStrategy();
        a.setSectorTemperature(81);
        rs.respondEmergency(a);

        assertTrue( condition: a.getSectorTemperature() < LocalAntenna.criticalThreshold);
    }
    @Test
    void evacuateAndSendtruckEmergencyResponse(){
        rs = new EvacuateAndSendTruckStrategy();
        a.setSectorTemperature(81);
        rs.respondEmergency(a);

        assertTrue( condition: a.getSectorTemperature() < LocalAntenna.criticalThreshold);
    }
}
```