

Progetto

Metodologie di Programmazione

Realizzato da:

Gabriele Bertini - 5793664 - gabriele.bertini3@stud.unifi.it - deve superare la prova scritta completa

Lorenzo Pratesi - 5793319 - lorenzo.pratesi@stud.unifi.it - votazione parziali 23/30

Scopo del Progetto

Lo scopo di questo progetto è di porre le basi per la creazione di un programma che analizzi dati provenienti da più fonti, e che le mostri all'utente secondo le specifiche desiderate. Nel nostro caso, abbiamo preso come esempio i dati provenienti da alcuni siti di previsioni meteo e li abbiamo catalogati in una struttura dati uniforme, che consentisse l'accesso alle stesse informazioni per ogni sito, dando così la possibilità di poter confrontare questi dati tra sé. A scopo esemplificativo abbiamo creato un confronto testuale nella interfaccia grafica, ma da questa base è possibile realizzare anche grafici e statistiche più elaborate.

Struttura del Progetto

Come si evince dai pacchetti, il nostro progetto è principalmente diviso in due parti, una di acquisizione e catalogazione delle informazioni e l'altra di visualizzazione e interfaccia per l'utente.

Le due parti sono messe in connessione da una classe (**InformationManager**), che si occupa di reperire le informazioni richieste dall'interfaccia grafica all'interno della struttura dati. Questo consente una buona indipendenza tra le due strutture, consentendo ad esempio grandi cambiamenti alla struttura dati senza ripercussioni sulla interfaccia grafica e viceversa.



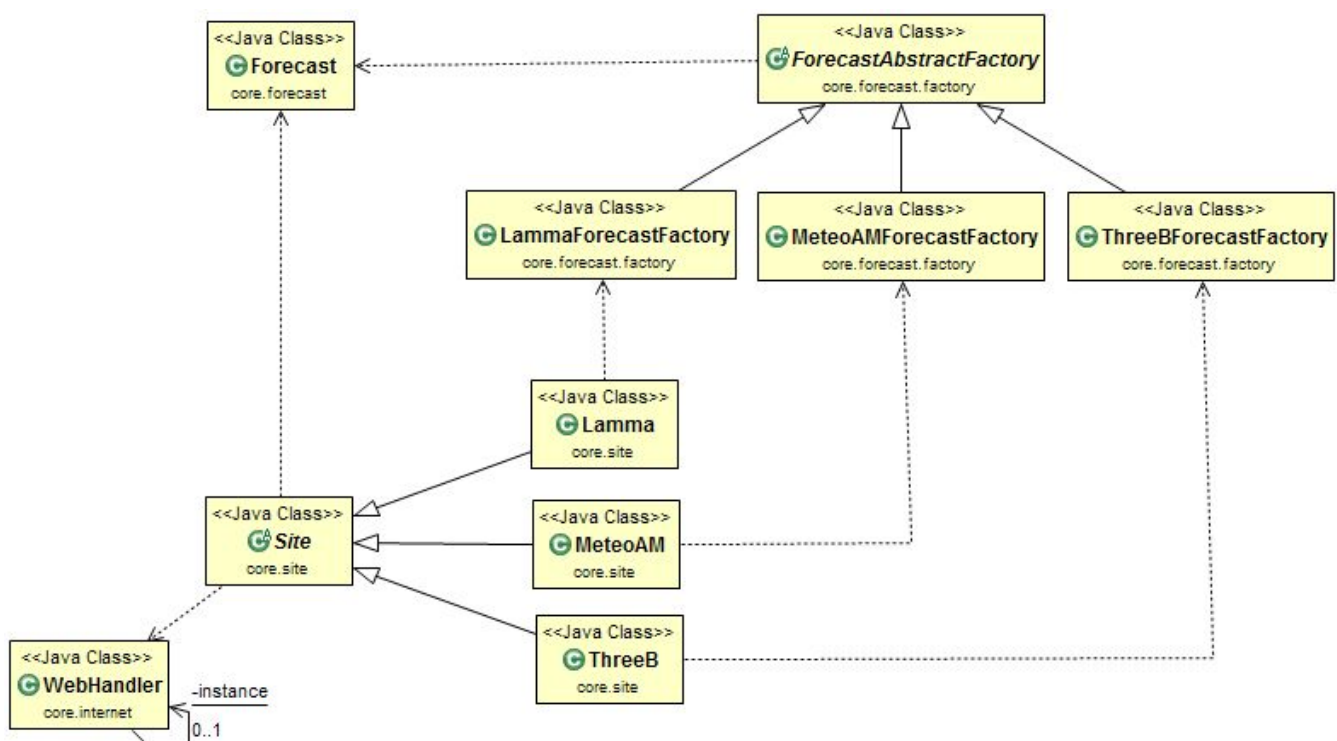
Estensibilità del Progetto e Riutilizzabilità del Codice

Come spiegato nell'introduzione, il progetto si pone lo scopo di raccogliere delle informazioni da fonti dove sono riportate in modi anche piuttosto diversi, per poi catalogarli in una struttura uniforme e renderli disponibili ad una successiva analisi. La struttura delle classi che implementa questo processo consente, con cambiamenti marginali, l'estensione di questo progetto ai più svariati tipi di informazioni.

La struttura dati utilizzata, infatti, è costituita da mappe di Stringhe, il che consente di inserirci informazioni diverse senza necessità di cambiamenti. Poiché il progetto è pensato per reperire informazioni su internet, le classi che implementano queste funzionalità non necessitano modifiche nel caso di cambiamenti nel tipo di informazioni ricercate.

In particolare, l'algoritmo implementato nel metodo template della classe **Site** (*getForecast(.)*), consente di cercare qualunque informazione da qualunque sito, poiché richiede alla classe concreta di fornirgli il proprio url e il proprio costruttore della struttura dati, così che se questa fosse cambiata, basterebbe modificare l'algoritmo del costruttore astratto, mentre, se si aggiunge un sito, basterà creare il relativo costruttore concreto, che cercherà le informazioni in base al sito associato.

Ricapitolando, la struttura dati utilizzata determina l'algoritmo di creazione implementato nell'AbstractFactory, mentre il sito da cui si estraggono le informazioni determina l'implementazione concreta della factory che inserisce le informazioni del sito nella struttura dati. Il cambiamento della struttura dati implica quindi un cambiamento nelle factory (astratta e concrete) ma lascia intatte le classi dei siti, mentre l'aggiunta di un nuovo sito implica solamente la creazione di una classe concreta per il sito e la relativa factory per l'estrazione delle informazioni.



Per quanto riguarda l'interfaccia grafica, la modularità con cui è costruita consente l'aggiunta di nuove componenti o di nuove funzionalità senza cambiamenti al resto del codice. Inoltre, poiché non si relaziona direttamente con la struttura dati, una eventuale modifica di quest'ultima non ha ricadute se non nella classe di raccordo. Come verrà spiegato in un apposito paragrafo, le classi all'interno del pacchetto *gui.utilities* costituiscono a tutti gli effetti un framework (anche se molto semplificato), con cui è possibile realizzare una qualunque interfaccia grafica in modo modulare ed indipendente.

Vista la possibilità di estendere il progetto ad altre tipologie di informazioni con relativa semplicità, ci focalizzeremo da questo punto in poi sulla struttura dati e le informazioni prese da esempio in questo progetto.

Pattern Utilizzati

I pattern utilizzati sono:

❖ Singleton:

- nella classe **WebHandler**, per garantire una solo istanza di questa classe, che si occupa di scaricare le informazioni dai vari siti internet.



❖ Template:

- nella classe **Site**, in cui è implementato l'algoritmo che porta alla costruzione della struttura dati contenente le informazioni del sito. Notare che la costruzione della struttura dati avviene tramite una factory concreta, mantenendo quindi separati il processo di acquisizione delle informazioni da quello di creazione della struttura in cui vengono catalogate le informazioni.

```

public Forecast getForecast(String location, int day) {
    String url = getLocationUrl(location, day);
    WebHandler web = WebHandler.getInstance();
    String siteContent = web.getSite(url);
    Document document = Jsoup.parse(siteContent);
    ForecastAbstractFactory constructor = getForecastConstructor();
    return constructor.createForecast(document, day);
}

public abstract String getLocationUrl(String location, int day);

public abstract ForecastAbstractFactory getForecastConstructor();
  
```

- nella classe **AbstractWeatherListener**, per fornire un unico algoritmo di visualizzazione delle informazioni sul display.

```
@Override
public void actionPerformed(ActionEvent arg0) {
    String location = searchBox.getText();
    display.setText("");
    List<Integer> meteoIDs = getSelectedMeteo();
    if (meteoNotSelected(meteoIDs)) // se non c'è un meteo selezionato interrompo
        return;
    List<Integer> daysIDs = getSelectedDays();
    List<String> timesIDs = getSelectedTimes();
    meteoIDs.forEach(mID -> displayAppend(location, daysIDs, timesIDs, mID));
    cancel.setSelected(true);
}

//Template method -> abs print
private void displayAppend(String location, List<Integer> daysIDs, List<String> timesIDs, int mID) {
    display.append("+++" + getMeteo(mID) + "+++\n\n");
    daysIDs.forEach(dID -> print(location, timesIDs, mID, dID));
    display.append("\n\n");
}

public abstract void print(String location, List<String> timesIDs, int mID, int dID);
```

❖ Abstract Factory:

- nella classe **ForecastAbstractFactory**, per fornire un algoritmo di creazione della struttura dati. Questo è implementato diversamente da ogni factory concreta, che cerca le informazioni all'interno del corpo del sito a cui è associata e le restituisce alla factory astratta per inserirle nella struttura dati. Questo processo consente di uniformare le informazioni contenute in modi diversi nei vari siti in una struttura dati uguale per tutti, consentendo una successiva elaborazione.

```
public Forecast createForecast(Document doc, int day) {
    Elements root = createRoot(doc);
    Map<String, Map<String, String>> forecast = new LinkedHashMap<>();
    forecast.put(ForecastConstants.INFOGIORNO, getInfoGiorno(root, day));
    forecast.put(ForecastConstants.NOTTE, getPrevisioniOrarie(root, day, FactoryConstants.NOTTE));
    forecast.put(ForecastConstants.MATTINA, getPrevisioniOrarie(root, day, FactoryConstants.MATTINA));
    forecast.put(ForecastConstants.POMERIGGIO, getPrevisioniOrarie(root, day, FactoryConstants.POMERIGGIO));
    forecast.put(ForecastConstants.SERA, getPrevisioniOrarie(root, day, FactoryConstants.SERA));
    return new Forecast(forecast);
}

public abstract Elements createRoot(Document doc);

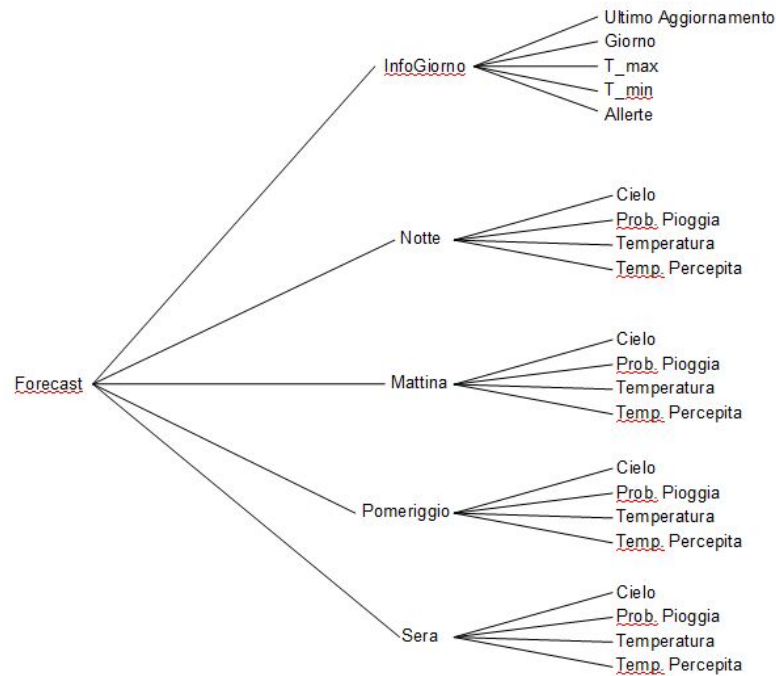
public abstract Map<String, String> getInfoGiorno(Elements root, int day);

public abstract Map<String, String> getPrevisioniOrarie(Elements root, int day, int orario);
```

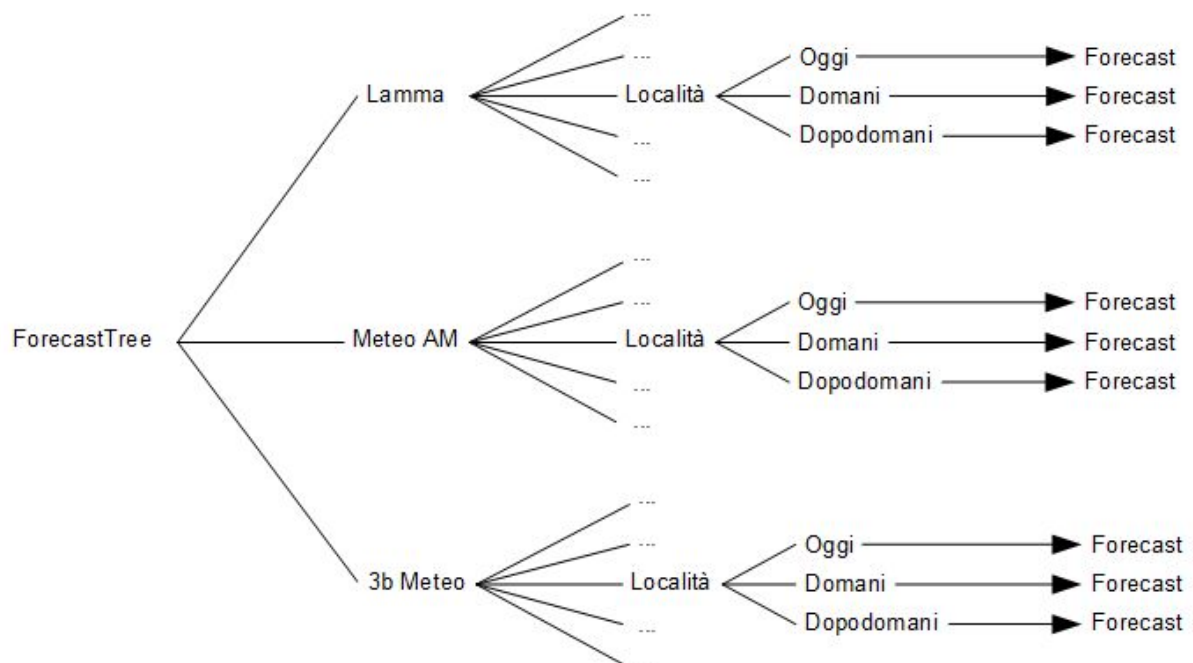
Struttura dati

In questo progetto vengono utilizzate due strutture dati, la prima dove vengono effettivamente catalogate le informazioni di un sito, la seconda dove vengono raccolte queste strutture.

- ❖ La prima struttura, implementata dalla classe **Forecast**, è costruita da una factory associata ad un determinato sito, tramite l'algoritmo fissato nell'abstract factory, ed è così strutturata:

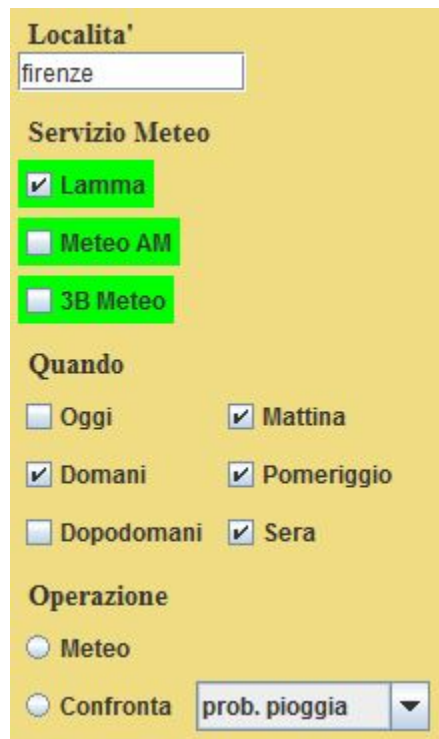


- ❖ La seconda struttura, implementata dalla classe **ForecastTree**, raccoglie varie istanze della prima struttura, organizzate per sito di provenienza, luogo e giorno a cui si riferiscono le informazioni:



Esecuzione Tipo

Riportiamo adesso il funzionamento di una chiamata tipo. Supponiamo di voler



The screenshot shows a web form with a yellow background. It has several sections: 'Localita'' with a text input containing 'firenze'; 'Servizio Meteo' with three checkboxes: 'Lamma' (checked), 'Meteo AM' (unchecked), and '3B Meteo' (unchecked); 'Quando' with four checkboxes: 'Oggi' (unchecked), 'Domani' (checked), 'Dopodomani' (unchecked), 'Mattina' (checked), 'Pomeriggio' (checked), and 'Sera' (checked); 'Operazione' with two radio buttons: 'Meteo' (selected) and 'Confronta' (unchecked); and a dropdown menu next to 'Confronta' showing 'prob. pioggia'.

confrontare le probabilità di pioggia previste per domani su Firenze dal LaMMA. Selezioniamo le informazioni sul pannello di controllo e clicchiamo su "Confronta" per avviare il programma.

La prima classe ad attivarsi è

CustomWeatherListener, che è l'ascoltatore per questo comando. Una volta attivata, nel metodo *actionPerformed(..)* si occuperà di "leggere" quali informazioni sono state richieste dall'utente e di richiederle alla struttura dati. Chiama quindi il metodo *getPrintableForecastElement(..)*

dell'**InformationManager** per avere queste informazioni. L'**InformationManager** ha al suo interno un albero (di tipo **ForecastTree**) in cui sono catalogate le informazioni, quindi chiede a questo di fornirgli le informazioni richieste dall'ascoltatore tramite il metodo *getDayForecast(..)*. Questo metodo si preoccupa innanzitutto di controllare che nell'albero siano presenti le informazioni di quel meteo per la

località richiesta. In caso affermativo estrae dall'albero quelle richieste e le restituisce all'**InformationManager**. In caso negativo, chiede che vengano create, per poter poi estrarre le informazioni che gli occorrono. Perciò viene chiamato il metodo *createLocationForecast(..)*, in cui viene scelto per quale servizio meteo occorre creare le informazioni e viene chiamato *getFullForecast(..)* per creare ed inserire nell'albero le informazioni per la località richiesta. Poiché le informazioni meteo sono catalogate in una struttura dati giornaliera (**Forecast**), il metodo *getFullForecast(..)* richiede tre di queste strutture (per il meteo di Oggi, Domani e Dopodomani) al corrispondente sito, invocando il metodo *getForecast(..)* per ognuno di questi giorni. La classe **Site**, o meglio nel nostro caso la classe **Lamma** (che è l'implementazione concreta), si occupa di creare le informazioni meteo richieste, seguendo l'algoritmo dettato dal metodo *getForecast(..)* della classe **Site** (*template method*). In particolare la classe **Lamma** si occupa di fornire il proprio url particolare per la località richiesta, così che la classe **Site** possa chiedere al **WebHandler** di scaricarne il contenuto, che verrà parsato dal parser **JSoup**. A questo punto la classe **Lamma** fornisce il proprio costruttore di forecast associato e a questo viene richiesto di elaborare le informazioni date dal parser e di costruire la struttura **Forecast** con le informazioni richieste.

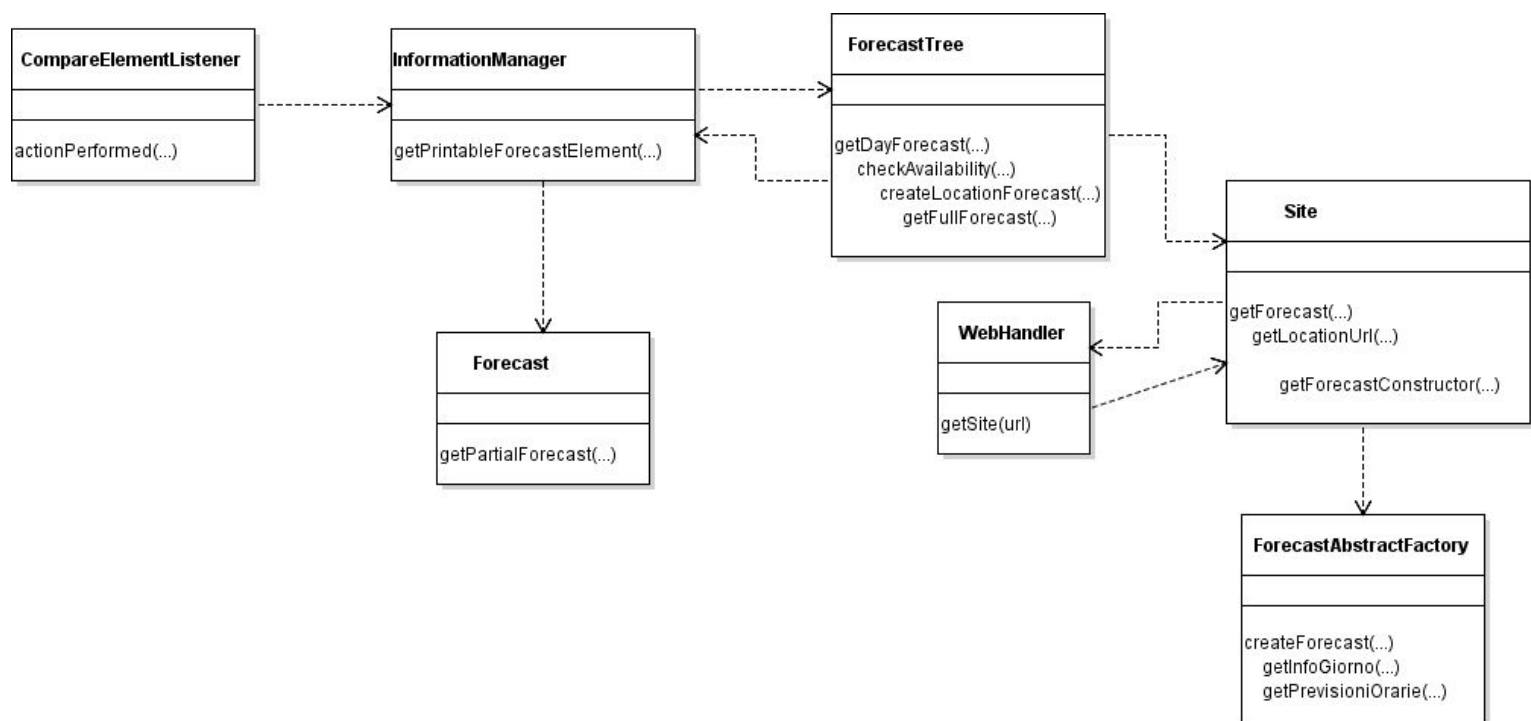
Costruite le 3 strutture **Forecast**, queste vengono inserite nell'albero e il controllo torna al metodo *getDayForecast(..)* che chiede, per ogni giorno selezionato dall'utente, alla relativa **Forecast** di estrarre le informazioni richieste.

Nel nostro caso quindi, poiché il giorno scelto è "Domani", viene richiesto alla **Forecast** con le informazioni meteo di domani di estrarre le informazioni "parziali" ovvero legate ad un solo orario (Mattina, Pomeriggio o Sera), e da queste viene estratto l'elemento richiesto, nel nostro caso le probabilità richieste. Queste informazioni vengono raccolte in un'unica stringa, e passate al richiedente, ovvero il **Listener**, che quindi chiederà al display di stamparle a schermo, ottenendo quanto richiesto dall'utente.

```
+++LAMMA+++

--DOMANI--
*mattina*
prob. pioggia: 5
*pomeriggio*
prob. pioggia: 5
*sera*
prob. pioggia: 5
```

Riportiamo qui uno schema approssimativo delle classi e dei metodi coinvolti:



Java 8

In questo progetto abbiamo scelto di utilizzare alcune funzionalità di Java 8 per la costruzione di metodi cercando, dove possibile, di alleggerire il codice. Questa scelta è stata fatta per mettere in pratica alcuni principi discussi a lezione in aula ed in laboratorio e, soprattutto, per imparare ad utilizzare tali funzionalità.

Note sui test

I test sono stati effettuati sulle classi dei due pacchetti principali (*core* e *gui*), andandone ad analizzare la correttezza. Abbiamo testato solo alcune classi dei rispettivi pacchetti, ovvero quelle di cui abbiamo ritenuto fondamentale assicurare il corretto funzionamento. Vengono testati tutti i metodi pubblici delle relative classi. I test vengono effettuati tramite il framework JUnit 4. Di seguito le classi test:

- Pacchetto core: *ForecastTest*, *ForecastTreeTest*, *InformationManagerTest*, *LammaForecastFactoryTest*, *SiteTest*, *WebHandlerTest*.
- Pacchetto gui: *AbstractWeatherListenerTest*, *ActiveComponentsManagerTest*, *DisplayContainerTest*, *ForecastComparatorMainTest*.

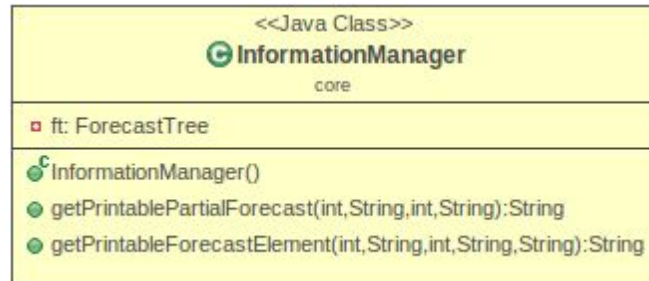
Librerie Esterne

Per il corretto funzionamento del programma ci appoggiamo a due librerie esterne, JUnit 4 come già citato nel paragrafo precedente e JSoup per il parsing del contenuto dei siti, scaricati dal **WebHandler**.

Core

Le classi all'interno del pacchetto core implementano la parte di ricerca e catalogazione delle informazioni.

❖ Package CORE:



- **InformationManager**: classe di raccordo tra la struttura dati e l'interfaccia grafica. Al suo interno ha una istanza di `ForecastTree`, al cui interno sono catalogate le informazioni.

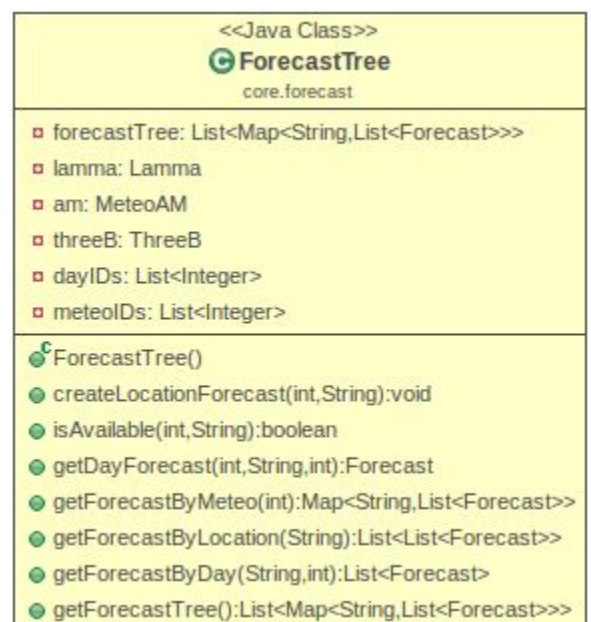
❖ Package core.FORECAST:

- **Forecast**: è la classe che implementa la struttura dati vera e propria. È creata tramite una factory concreta, associata al sito da cui provengono le informazioni.



- **ForecastConstants**: contiene le costanti utilizzate per referenziare i dati nella struttura di `Forecast`.

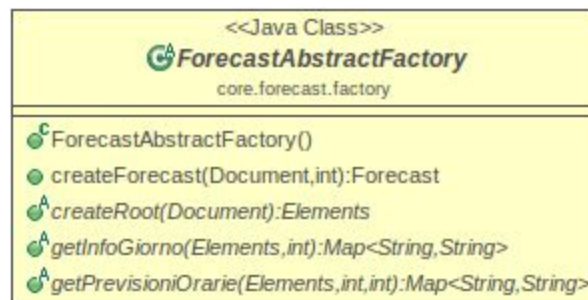
- **ForecastTree**: è la classe che implementa la struttura dati che cataloga le informazioni raccolte.



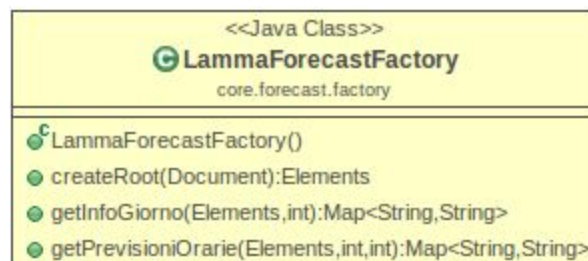
❖ Package core.forecast.FACTORY:

- **FactoryConstants:** contiene le costanti utilizzate per la costruzione della struttura dati Forecast.

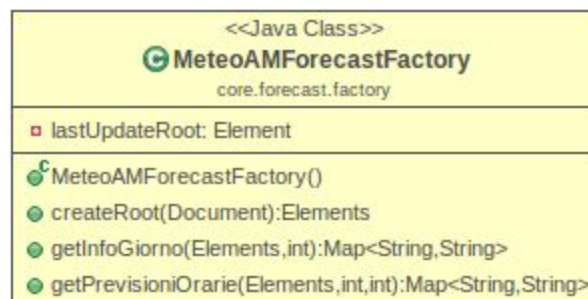
- **ForecastAbstractFactory:** è la classe responsabile della creazione della struttura dati. Il suo metodo principale definisce l'algoritmo attraverso cui la struttura viene creata, appoggiandosi a metodi astratti implementati dalle classi figlie concrete. Queste ultime sviluppano i metodi astratti raccogliendo le informazioni dal sito a cui sono associate.



- **LammaForecastFactory:** factory concreta, implementa i metodi della classe astratta prelevando le informazioni dall'XML fornito dal sito Lamma.

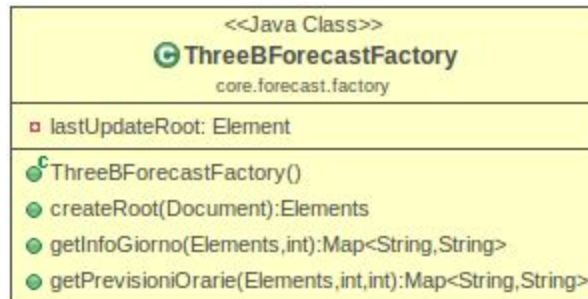


- **MeteoAMForecastFactory:** factory concreta, implementa i metodi della classe astratta prelevando le informazioni dal codice HTML del sito Meteo AM.



- **ThreeBConstants:** contiene costanti utilizzate dalla classe *ThreeBForecastFactory*.

- **ThreeBForecastFactory**: factory concreta, implementa i metodi della classe astratta prelevando le informazioni dal codice HTML del sito 3B Meteo.



❖ Package core.INTERNET:

- **WebHandler**: creata utilizzando il pattern Singleton, tramite il metodo *getSite* si occupa di generare una stringa contenente l'HTML del sito web il cui URL viene passato come parametro (nel caso in cui venga generata un'eccezione dovuta alla mancanza di connessione ad internet, carica l'HTML o l'XML da file con il metodo *loadStringExample* sempre passandogli l'URL come parametro); inoltre può scaricare un'immagine, nel nostro caso un'immagine da satellite utilizzata nell'interfaccia grafica (nel caso in cui venga generata un'eccezione dovuta alla mancanza di connessione ad internet, carica l'immagine da file con in metodo *getSampleImage*).



❖ Package core.SITE:

- **Lamma**: classe concreta che estende Site, inserendo le informazioni specifiche del sito.



- **MeteoAM**: classe concreta che estende Site, inserendo le informazioni specifiche del sito.



- **Site** (abstract): è la classe che si occupa di reperire le informazioni per la creazione della struttura dati. Il metodo `getForecast(..)` fissa l'algoritmo con i passi necessari al reperimento delle informazioni. Inoltre si occupa di creare la struttura di controllo delle località disponibili per un dato sito. Questa è costituita da un array di liste, il cui indice indica la lunghezza delle parole contenute nella lista. Al momento della creazione vengono caricate da file tutte le località disponibili e vengono inserite nella lista all'indice corrispondente alla propria dimensione. Questa struttura consente di controllare in tempi molto rapidi la disponibilità di una certa località per sito, ed è utile per implementare il controllo dinamico che viene sviluppato nell'interfaccia grafica.



- **ThreeB**: classe concreta che estende Site, inserendo le informazioni specifiche del sito.



Interfaccia Grafica

L'interfaccia grafica è stata sviluppata usando le classi fornite dalle librerie awt e swing, utilizzando il **GridBagLayout** per avere maggior controllo sul posizionamento delle componenti.

Al fine di creare una interfaccia con classi coese e indipendenti, sono state create alcune classi che consentono di risolvere le problematiche principali, costituendo quindi un framework, seppure molto semplificato.

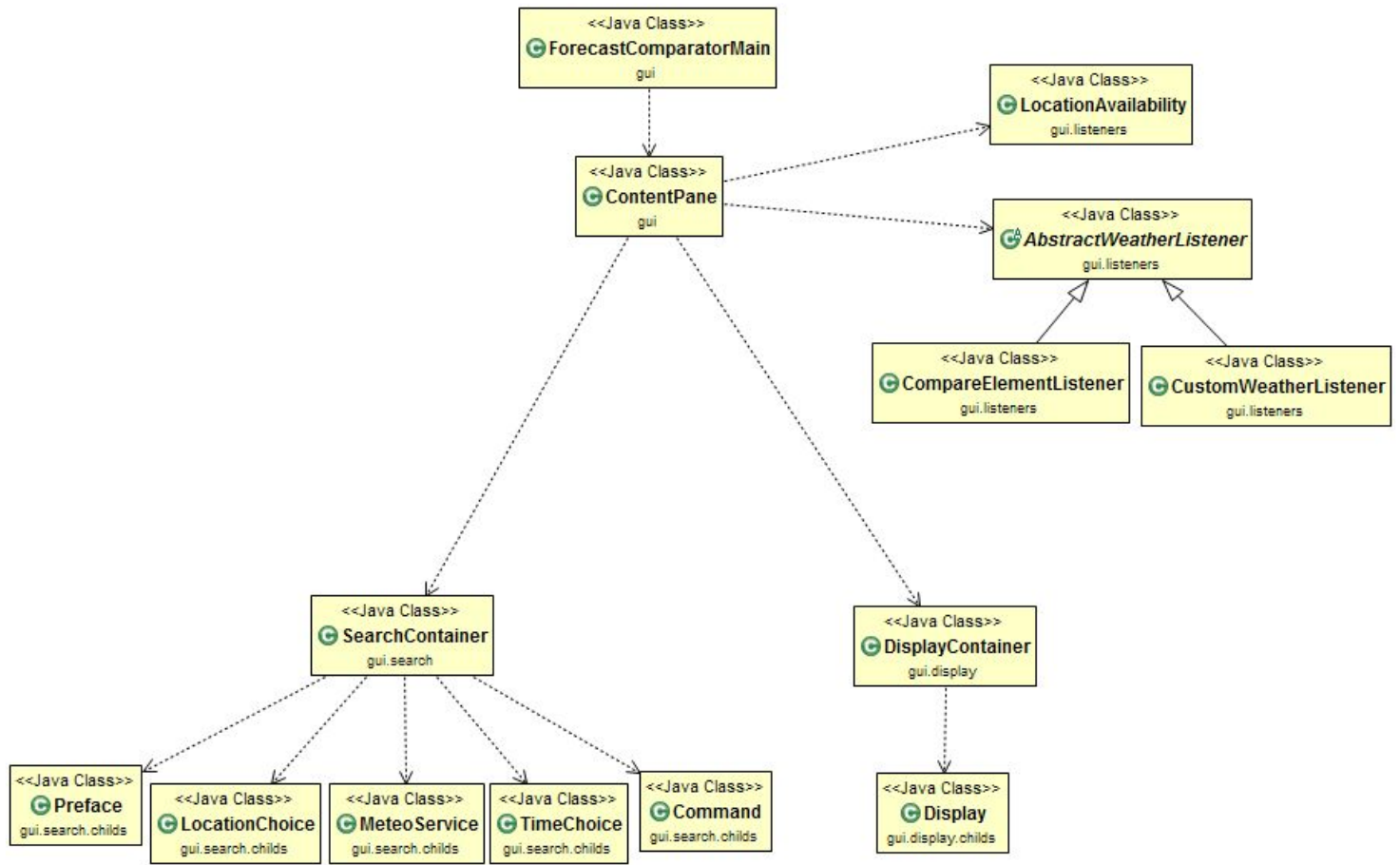
La classe principale di questo framework è **GridBagContainer**. Questa è una estensione astratta della classe **Container**, in cui viene impostato il layout **GridBagLayout** e le limitazioni vengono create tramite la classe **MyConstraints**, che è una versione modificata della classe **GridBagConstraints**.

Inoltre, al suo interno ha un campo di tipo **ActiveComponentsManager**, che implementa una struttura che mantiene in memoria il riferimento alle componenti che vengono utilizzate dai listener per compiere delle azioni.

Questo consente di creare strutture modulari in cui inserire singoli o piccoli gruppi di componenti, mantenendo un codice pulito e conciso, poiché la struttura implementata da **ActiveComponentsManager** consente di far passare i riferimenti alle classi parents fino al **ContentPane**, dove possono essere utilizzati più listeners, ognuno implementato singolarmente.

Se già queste classi, contenute nel pacchetto *gui.utilities*, danno la possibilità di costruire in pochi passi le più svariate interfacce, la struttura adottata per il nostro progetto costituisce una base già pronta all'uso per essere estesa a strutture dati diverse e comandi diversi. Infatti l'interfaccia è suddivisa in due moduli principali, uno che contiene i controlli e l'altro che contiene la parte destinata alla visualizzazione delle informazioni. Questo consente di aggiungere componenti senza modifiche agli altri moduli, caratteristica molto importante, visto che il nostro progetto si propone di essere una base da cui poter sviluppare applicazioni che abbiano a che fare con dati anche molto diversi. In particolare abbiamo lasciato la parte di visualizzazione delle informazioni solo testuale, poiché da questa base si possono realizzare grafici e statistiche diverse, che lasciamo ad un successivo sviluppo.

Per quanto riguarda i **Listener**, che sono le classi responsabili dell'attuazione dei comandi, ovvero dell'elaborazione dei dati secondo le richieste dell'utente, questi sono posizionati nel **ContentPane**, ovvero a monte della divisione delle due gerarchie di moduli, quella dei comandi e quella della visualizzazione. Poiché ogni **Listener** necessita di controllare le richieste dell'utente, questa parte è affidata ad un **Listener** astratto, che si occupa inoltre dell'algoritmo di visualizzazione delle informazioni, mentre è lasciata al **Listener** concreto la richiesta all' **InformationManager** delle informazioni da visualizzare.

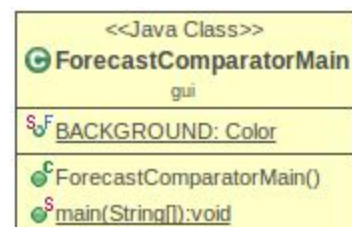


❖ Package GUI:

- **ContentPane:** è il container principale, che racchiude tutti i container e le componenti dell'interfaccia grafica. Viene aggiunto sul JFrame per essere visualizzato.



- **ForecastComparatorMain:** è la classe Main dalla quale si avvia la GUI.



❖ Package gui.DISPLAY:

- **DisplayContainer**: è il container che raccoglie i moduli responsabili della visualizzazione delle informazioni richieste.



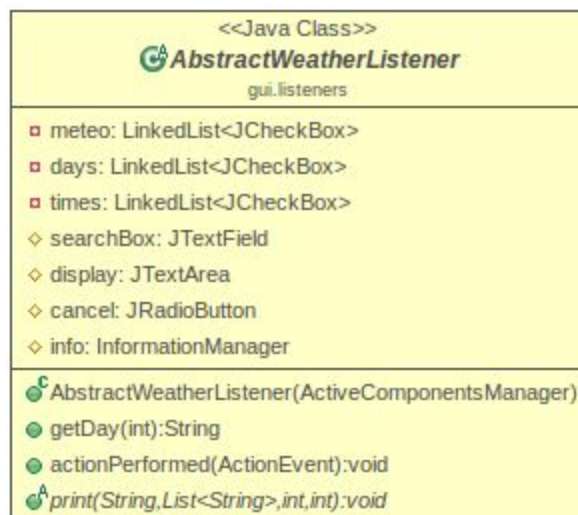
❖ Package gui.display.CHILDS:

- **Display**: è la classe che contiene la JTextArea su cui vengono visualizzate le informazioni.



❖ Package gui.LISTENERS:

- **AbstractWeatherListener**: è la classe che si occupa del controllo di tutte le impostazioni date dall'utente nel modulo dei comandi, necessarie per ogni listener che implementa un comando. Inoltre si occupa di dare un algoritmo di visualizzazione delle informazioni, su cui ogni listener si appoggia per inserire le proprie informazioni.



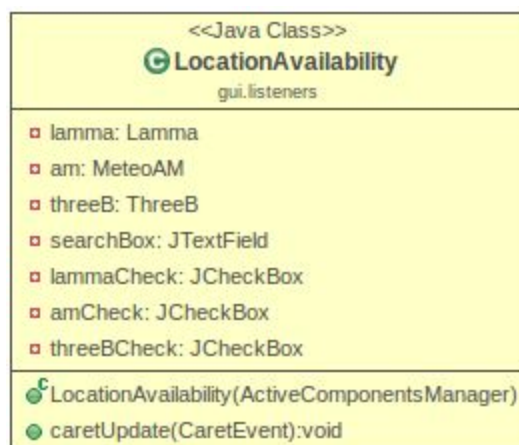
- **CompareElementListener:** è l'ascoltatore che implementa il comando "Confronta". Implementa il metodo print, che fornisce all'algoritmo di visualizzazione le informazioni specifiche da visualizzare.



- **CustomWeatherListener:** è l'ascoltatore che implementa il comando "Meteo".



- **LocationAvailability:** Questo ascoltatore si occupa di implementare il controllo dinamico della località inserita dall'utente, visualizzando per quali servizi meteo è disponibile. Questo controllo avviene sfruttando la struttura delle località generata dalla classe Site, e serve ad impedire che venga cercata una località in un servizio meteo il quale non dispone delle informazioni. Infatti questo ascoltatore viene attivato ogni volta che viene modificato il contenuto del JTextField al cui interno viene inserita la località. All'attivazione viene controllato per ogni sito se la località è disponibile o meno. Se la località risulta disponibile il CheckBox corrispettivo al servizio meteo viene colorato di verde e reso disponibile ad essere selezionato, mentre se la località non è disponibile, viene colorato di rosso e impostato come non selezionabile.



❖ Package gui.SEARCH:

- **SearchContainer:** è il container che raccoglie i moduli responsabili della selezione delle impostazioni e dei comandi.



❖ Package gui.search.CHILDS:

- **Command:** inserisce le componenti per la scelta dell'operazione. Poiché sono RadioButton e il comando viene attivato con la semplice selezione, è stato aggiunto un pulsante "cancel" che non è visualizzato nell'interfaccia, ma viene utilizzato per resettare il pulsante dopo aver eseguito l'operazione.



- **LocationChoice:** inserisce le componenti per l'inserimento della località di cui si vogliono avere informazioni.



- **MeteoService:** aggiunge le componenti per la selezione del servizio meteo.



- **Preface:** inserisce il titolo e l'immagine satellitare.

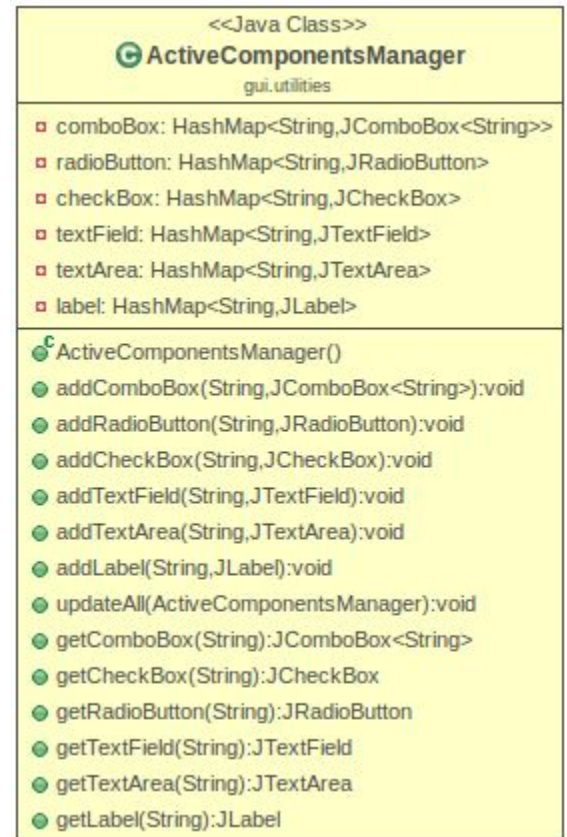


- **TimeChoice:** inserisce le componenti per scegliere alcune impostazioni.

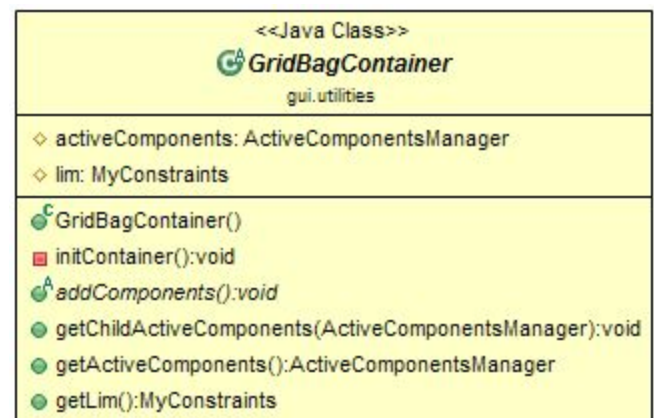


❖ Package gui.UTILITIES:

- **ActiveComponentsManager**: è la classe responsabile di tenere in memoria il riferimento a tutte le componenti "attive" (ovvero che vengono utilizzate dai listener). Per far ciò le varie componenti vengono catalogate per tipo in mappe, in modo che per ogni riferimento ci sia associato un nome significativo con cui cercarlo. Si rende necessaria quindi una mappa per ogni tipo, per evitare downcast successivi. La classe presenta inoltre un metodo per aggiungere il contenuto di un altro ACM alle proprie mappe. Questo si rivela fondamentale per poter mantenere in modo efficace il riferimento alle componenti di un modulo figlio.



- **GridBagContainer** (abstract): come spiegato nell'introduzione, questa classe consente di modularizzare la costruzione dell'interfaccia grafica. I campi sono lasciati protetti, così che le classi che la estendono possano accedervi liberamente.



- **MyConstraints**: è una specializzazione della classe **GridBagConstraints**, in cui sono stati inseriti metodi utili per l'impostazione delle limitazioni. Come esempio è stato inserito anche un metodo che imposta delle limitazioni di default utilizzabile da tutta una categoria di componenti che presentano le stesse necessità.

