RELAZIONE DI BUILDING DEVICE MANAGER

Alfredo Maffi Manuel Peruzzi Simone Lunedei

28/02/15

INDICE

CAPITOLO 1: ANALISI

REQUISITI

Building Devices Manager è un'applicazione di gestione della domotica e, più in generale, di gestione dei devices di un appartamento. Pertanto, il suo scopo principale sarà quello di permettere all'utente di interagire con i suddetti devices, anche attraverso l'ausilio di un simulatore temporale.
Dovrà essere possibile effettuare più operazione di gestione sui devices e garantire inoltre, solo per alcuni, anche la gestione automatizzata.
Per quanto riguarda il simulatore, esso dovrà essere in grado di adattarsi alle scelte dell'utente e fornire una rappresentazione temporale e metereologica prossima alla realtà.
Questo software è stato realizzato come progetto per il corso di Programmazione ad Oggetti della facoltà di Ingegneria e Scienze Informatiche dell'Università di Bologna, Italia (A.A. 2014/2015).

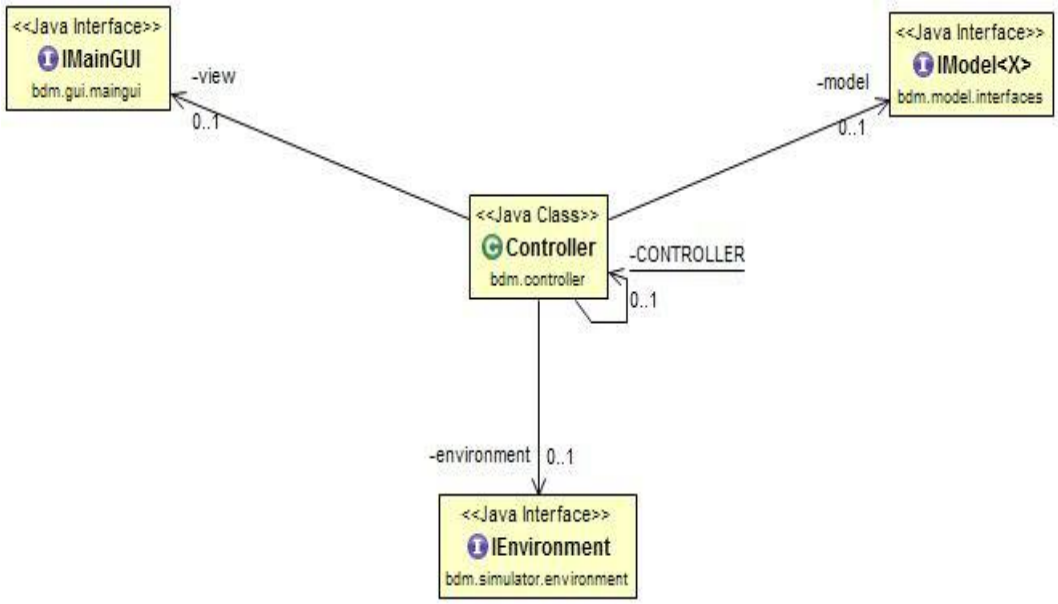
PROBLEMA

Le Problematiche da affrontare sono diverse, legate a diverse parti del progetto. Pertanto si cerca di esporle separatamente.
-MODEL : il modello è il cuore dell'applicazione e dovrà tener conto dello stato interno di tutti i Devices a fronte delle varie modifiche effettuate dall'utente. Ovviamente, non tutti i Devices avranno le stesse funzionalità, quindi vi saranno due categorie di operazioni: quelle effettuabili su tutti i Devices (es. Accensione e Spegnimento) e quelle effettuabili solo su alcuni (Accensione Programmata, Spegnimento Programmato, Automazione).
Inoltre, ci saranno dei particolare Devices (come ad esempio Heater o Lamp) per i quali si dovrà tener traccia del livello di Temperatura/Luminosità.
-VIEW : la user interface dovrà essere strutturata in maniera tale da garantire all'utente una corretta interazione con il modello e il simulatore. Più nello specifico, dovranno essere rappresentati: la piantina di un piano di un edificio, con i relativi Devices e il loro stato corrente; una serie di bottoni per la modifica dello stato interno dei suddetti; le condizioni metereologiche e lo scorrere del tempo in relazione alla data corrente.
La user interface dovrà aggiornarsi ogni qual volta il modello o il simulatore subiscono una modifica e ,ovviamente, adattarsi a qualsiasi risoluzione dello schermo e a qualsiasi ridimensionamento.
-SIMULATOR : il simulatore dovrà occuparsi sostanzialmente di due aspetti cruciali : lo scorrere del tempo e la simulazione , prossima alla realtà, delle previsioni metereologiche in relazione alla data corrente. Più nello specifico, dovrà essere possibile, da parte dell'utente, selezionare una data da cui far partire la simulazione e variare la velocità dello scorrere del tempo.
Infine, dovrà essere presente un file contenente le informazioni metereologiche di un certo lasso temporale (misurato in anni) di una determinata città. Se la data scelta dall'utente ricade all'interno di questo range temporale, allora il simulatore dovrà limitarsi a leggere i dati dal file, altrimenti , questi ultimi, verranno considerati per elaborare previsioni metereologiche accostate alla realtà.
-CONTROLLER: il controller dovrà occuparsi di svolgere il ruolo di intermediario tra le componenti del progetto.

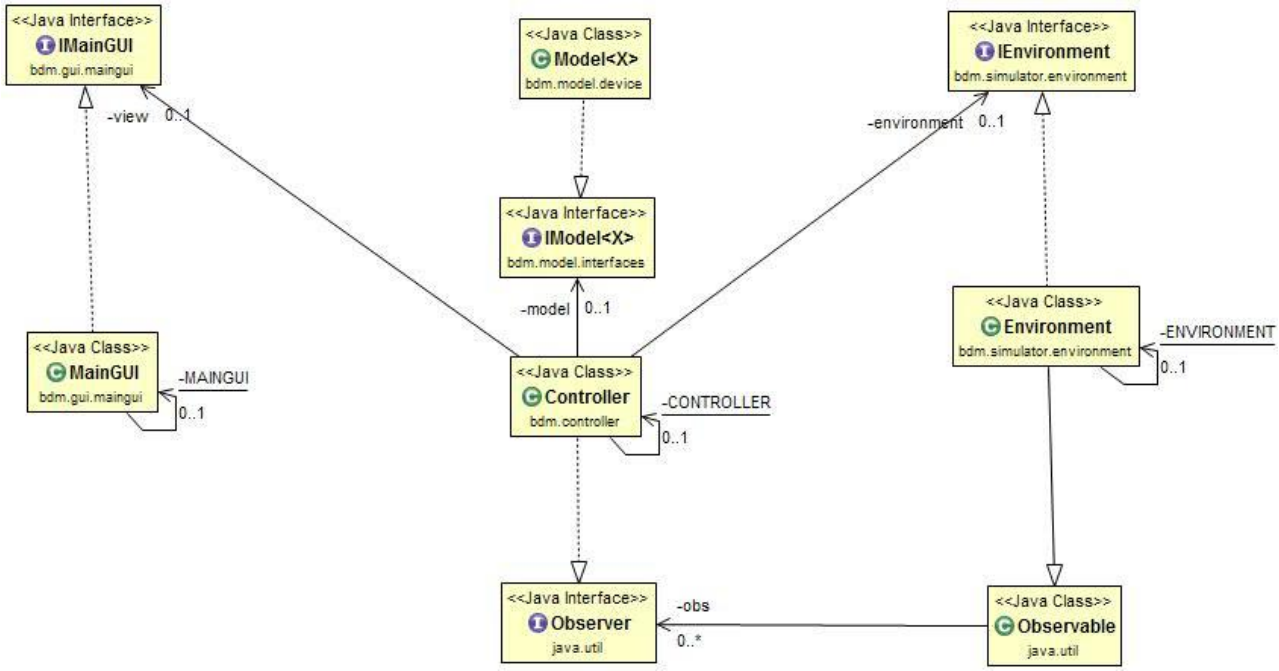
CAPITOLO 2: DESIGN

ARCHITETTURA

Per risolvere tutti i problemi posti in fase di analisi, è stato scelto di seguire il pattern architetturale Model-View-Controller (MVC) e di implementare le varie componenti di conseguenza. In questo modo, Model e View risultato essere completamente indipendenti tra loro e vi è la possibilità di modificare in blocco l'uno o l'altra senza apportare modifiche ad altre parti del progetto (come, ad esempio, modifiche al Controller). E' fornito, di seguito, uno schema UML che rappresenta l'implementazione del suddetto pattern architetturale.



View, Model e Simulator, rappresentati rispettivamente dalle interfacce IMainGUI, IModel<X> e IEnvironment, risultano completamente indipendenti l'uno dagli altri.
Infatti non vi è alcuna relazione fra di essi.
La classe che permette l'interfacciamento fra le parti è Controller, la quale mantiene un riferimento ad un oggetto di ciascuna interfaccia.
E' fornito, di seguito, un secondo schema UML che rappresenta in maniera più dettagliata la relazione fra i vari oggetti.



Nello schema sopra riportato sono mostrate anche le classi concrete che implementano le singole interfacce.
Environment estende la classe Observable, e quindi ogni modifica effettuata sul suo stato interno viene identificata e può essere notificata a tutti gli Observers, come applicazione del pattern Observer. Controller implementa l'interfaccia Observer, e quindi può, settandosi come osservatore di Environment, ricevere le notifiche del cambiamento di alcuni suoi oggetti.
In questo modo risulta immediato, per il Controller, aggiornare l'interfaccia grafica e il modello a seconda delle variazioni dello stato interno di Environment, ad esempio è importante che la GUI rappresenti lo scorrere del tempo in modo coerente e reattivo, e ciò è realizzabile se il Controller viene notificato ogni qual volta vi è una variazione dell'ora all'interno di Environment.

DESIGN DETTAGLIATO

Dopo aver esaminato gli aspetti architetturali principali, vengono ora presi in esame singolarmente alcune sottoparti rilevanti dell'applicazione.
Vengono approfonditi, dunque, gli aspetti implementativi rilevanti di GUI, Model e Simulator.

MODEL

L’interfaccia IModel è costituita da tutte le operazioni che è possibile effettuare sui devices. La classa concreta Model contiene al suo interno un serie di IDevices, memorizzati ed identificati attraverso un Generico X. Ovviamente, non tutte le operazioni sono effettuabili su ogni tipo di Device, perciò il Model si occupa di verificare che ogni operazione chiamata dal Controller possa essere effettivamente eseguita sul Device desiderato.

L’interfaccia IDevice rappresenta il contratto base e fornisce i metodi che ogni Device è obbligato ad implementare.

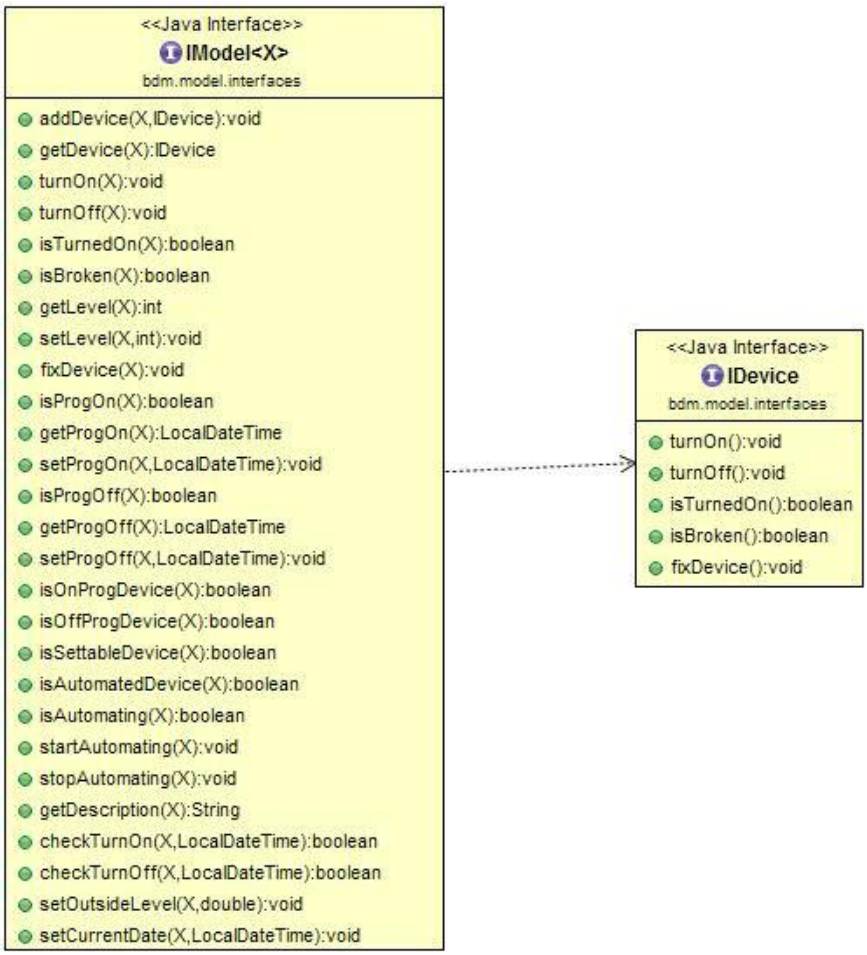


Figura 2.1 : Schema UML che rappresenta l'interfaccia IModel, che opera con IDevice

A seguito di IDevice, è presente una serie di interfacce che rappresentano le varie tipologie di Device presenti nell'applicazione:

- IOnProgDevice : un device la cui accensione può essere programmata ,e quindi settata, ad una data futura.
- IOffProgDevice: un device il cui spegnimento può essere programmato, e quindi settato, ad una data futura.
- IAutomatedDevice: un device che supporta la funzionalità di auto gestione.

Inoltre, per comodità e riutilizzo di codice, sono presenti due interfacce che identificano un particolare tipo di device:

- ILamp: rappresentazione di una Lampadina, che tiene conto della sua luminosità e della luminosità dell'ambiente esterno. Supporta la funzionalità di auto gestione.
- IHeater: rappresenta il sistema di riscaldamento di un appartamento, che tiene conto della temperatura dell'Heater, la temperatura interna e la temperatura esterna. Supporta la funzionalità di auto gestione.

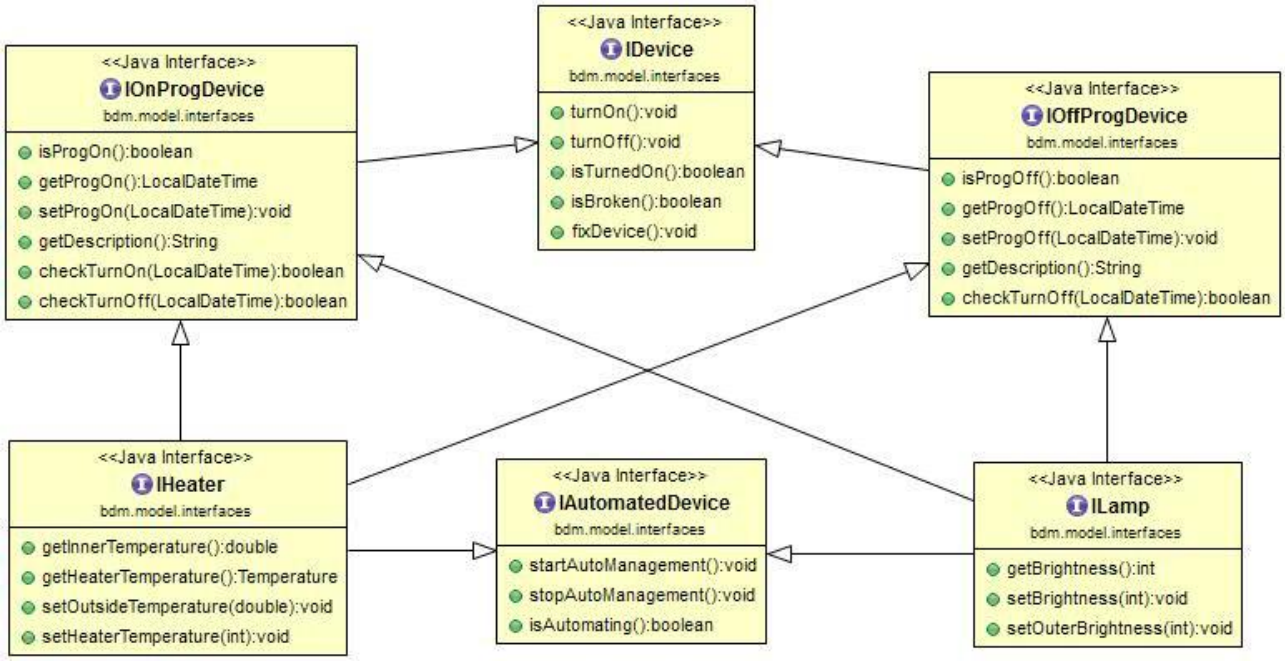


Figura 2.2: Gerarchia di tutte le interfacce dell'applicazione

Di seguito, è presente la classe Device, classe astratta che fornisce una prima implementazione dei metodi di base. E' presente un Template Method (ovvero compute()) che è chiamato per verificare se il Device si rompe o meno (attraverso il metodo astratto getProbability() ogni classe che estende da Device fornisce la probabilità che ha di rompersi ad ogni accensione).Vi sono poi le classi astratte OnProgDevice e OffProgDevice che forniscono una prima implementazione delle operazioni dedite a settaggio, lettura e controllo di , rispettivamente, accensione programmata e spegnimento programmato. Più nello specifico, OnProgDevice rappresenta un dispositivo che, una volta acceso tramite accensione programmata, resta tale per un certo tempo, che dipende dal dispositivo stesso. Per implementare ciò, in OnProgDevice viene tenuto conto , quindi, anche di una data di spegnimento programmato (non accessibile dall'utente) che viene settato internamente, attraverso un Template Method, ogniqualvolta viene settata una data di accensione programmata.
Per rappresentare un Device che supporta entrambe le operazioni sopra citate (interamente gestibili dall'utente) è presente la classe BothProgDevice e , per implementarla, è stata utilizzata una simulazione di ereditarietà multipla, per avere il massimo riutilizzo di codice.
Di seguito, sono presenti le implementazioni concrete di tutti i Devices: ognuno di essi, estende da OnProgDevice, OffProgDevice o BothProgDevice a seconda del tipo.
E' stata fatta un'ulteriore classa astratta per quanto riguarda le lampadine, dove è presente l'implementazione dei metodi per la gestione della luminosità delle stesse e dove è stato ridefinito il Template Method "compute()", siccome diverse lampadine si potranno rompere per motivi diversi.
Inoltre, è presenta la classe Heater, che contiene al suo interno tutte le operazioni per settare la temperatura interna e per tenere traccia di quella esterna. Heater è un SINGLETON ed è stato deciso di renderlo tale perché all'interno di un appartamento, il sistema di riscaldamento è uno e uno solo.

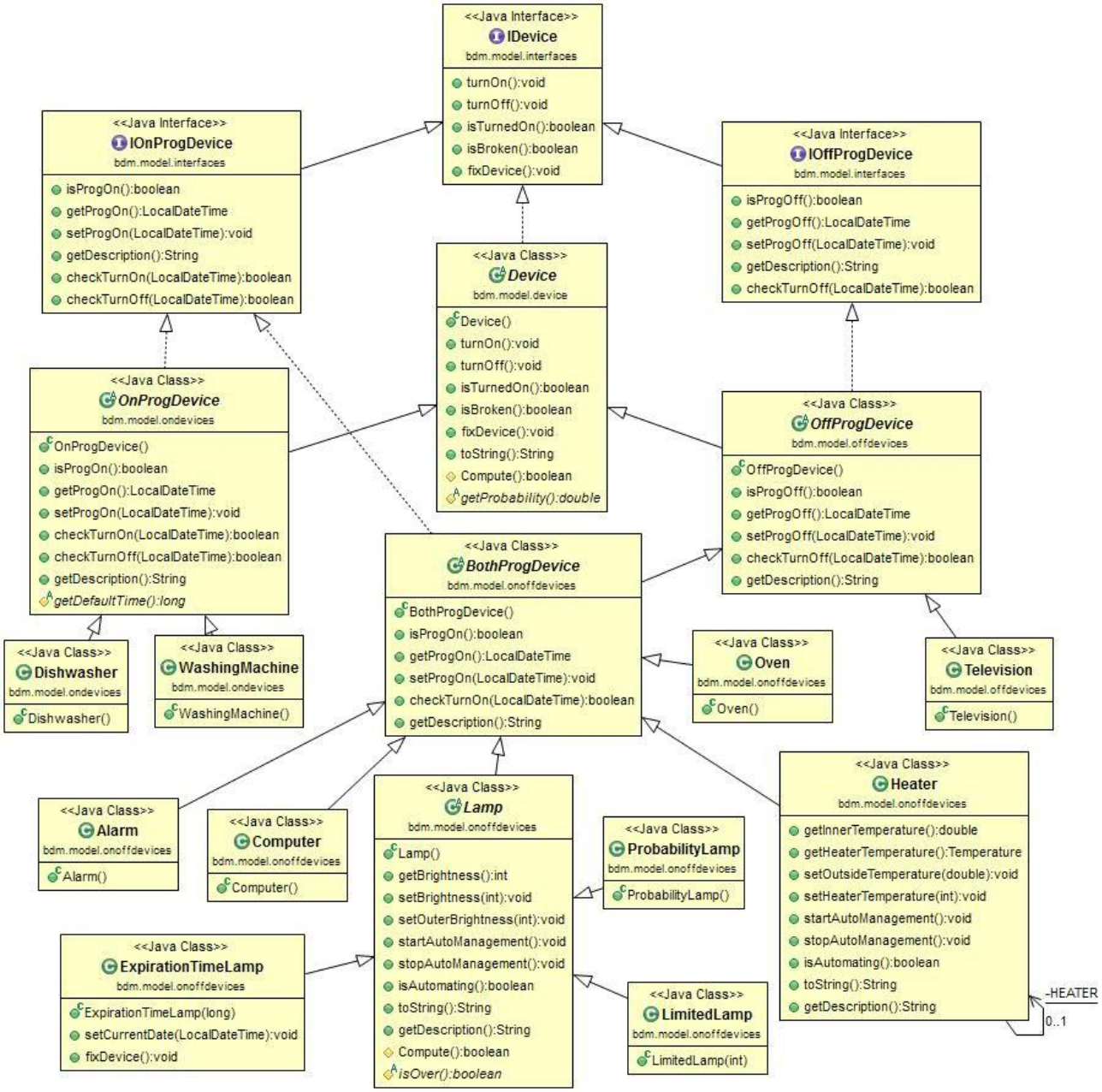


Figura 2.3: UML generale che mostra l'implementazione di tutti i Devices dell'applicazione

Infine, per quanto riguarda la gestione automatizzata, in tutte le classi che la supportano è presente una classe innestata, come mostrato in figura 2.4, che estende da Agent.
Agent, a sua volta, estende da Thread e il metodo run è invocato quanto viene fatta partire la gestione automatizzata (attraverso il metodo pubblico fornito da IAutomatedDevice). Più nello specifico, run è un Template Method, che ,ad ogni secondo, chiama un metodo astratto , implementato nelle classi innestate dei Device. Tale metodo astratto, definisce il comportamento che il Device deve avere quando entra in gestione automatizzata.

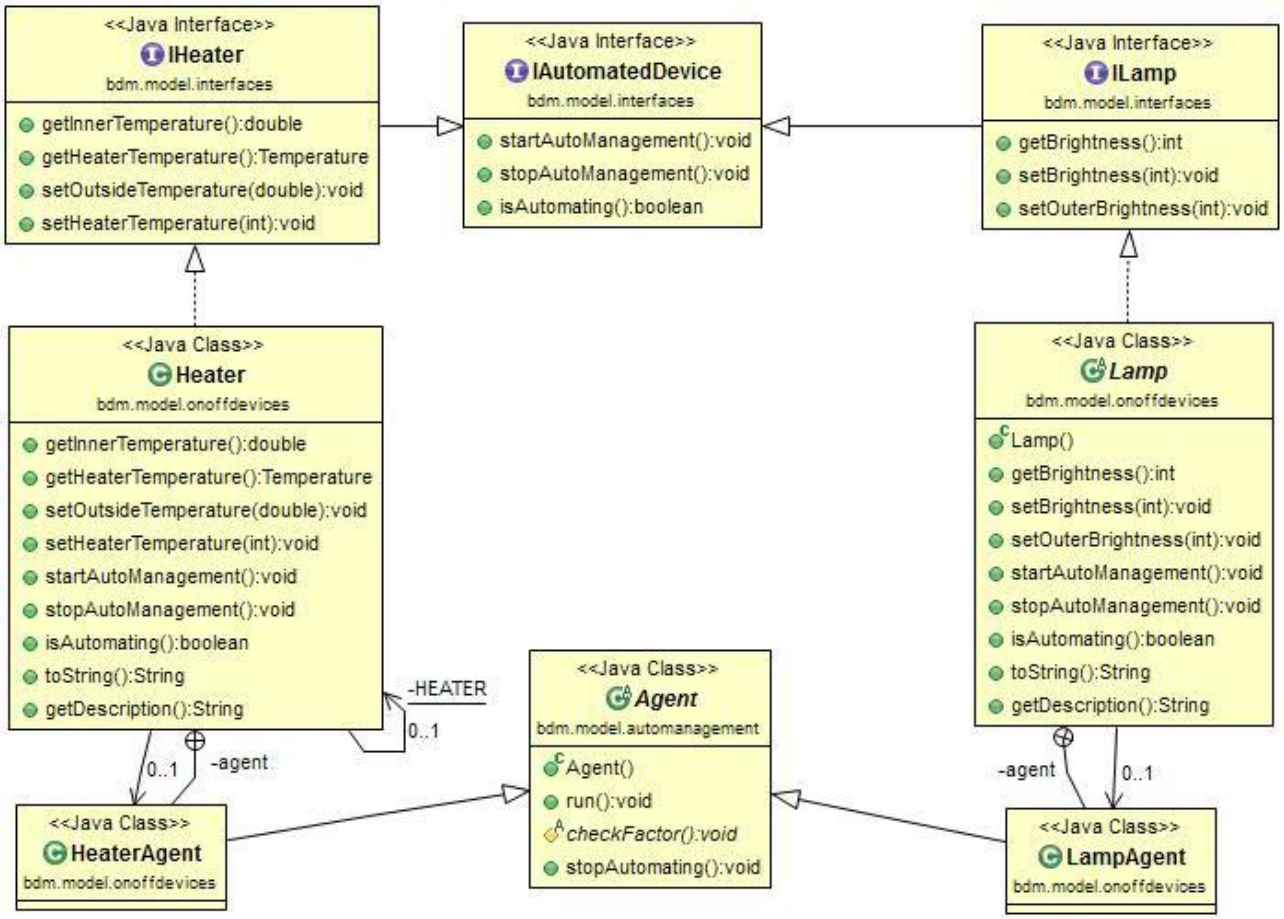
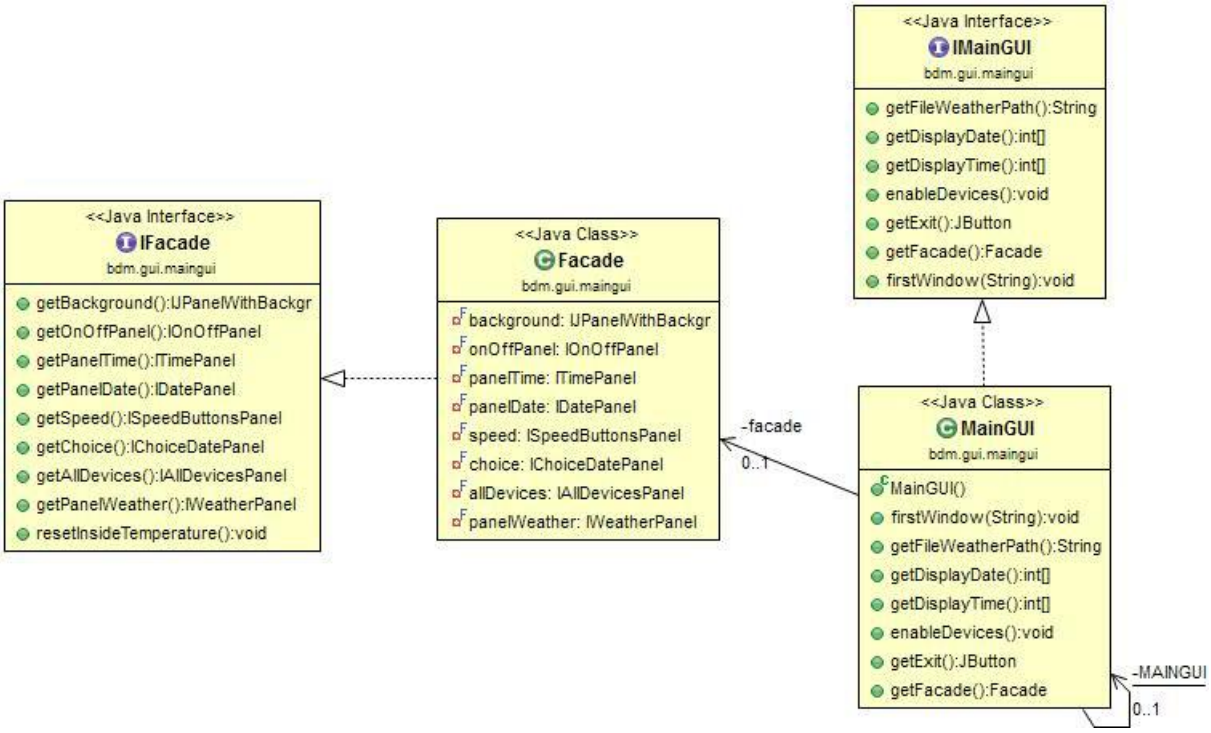


Figura 2.4: schema UML che espone l'implementazione della gestione automatizzata

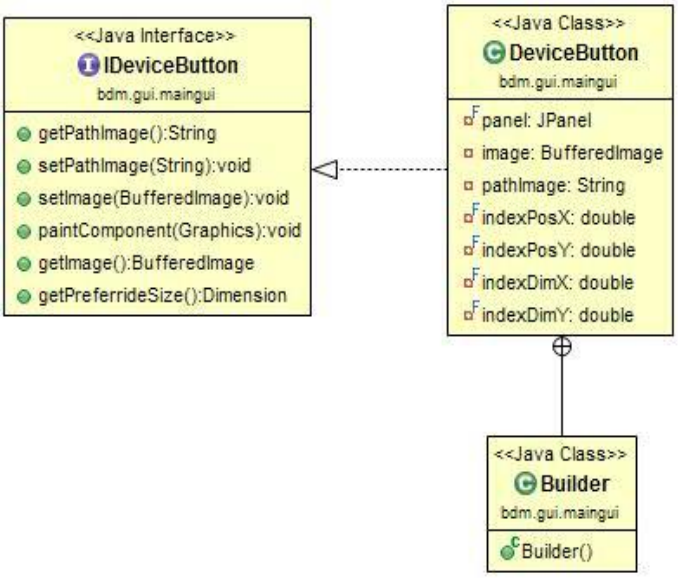
VIEW

La realizzazione dell'interfaccia grafica avviene attraverso l'implementazione dell'Interfaccia IMainGUI. Tale interfaccia richiede l'implementazione di diversi metodi getters; tra questi è presente il metodo getFacade() che restituisce un oggetto Facade;Vuole ora essere posta l'attenzione su quest'ultimo, in quanto attraverso l'implementazione dell'interfaccia IFacade è stato possibile realizzare l'interfaccia IMainGUI con un numero di metodi piuttosto limitato, rendendola più semplice e ordinata.



Si tratta quindi dell'implementazione del pattern Facade, in quanto la quasi totalità dei metodi che vengono richiamati dal controller sono stati definiti all'interno dell'oggetto Facade.
Tale oggetto è composto dai seguenti campi (che sono a loro volta delle interfacce):

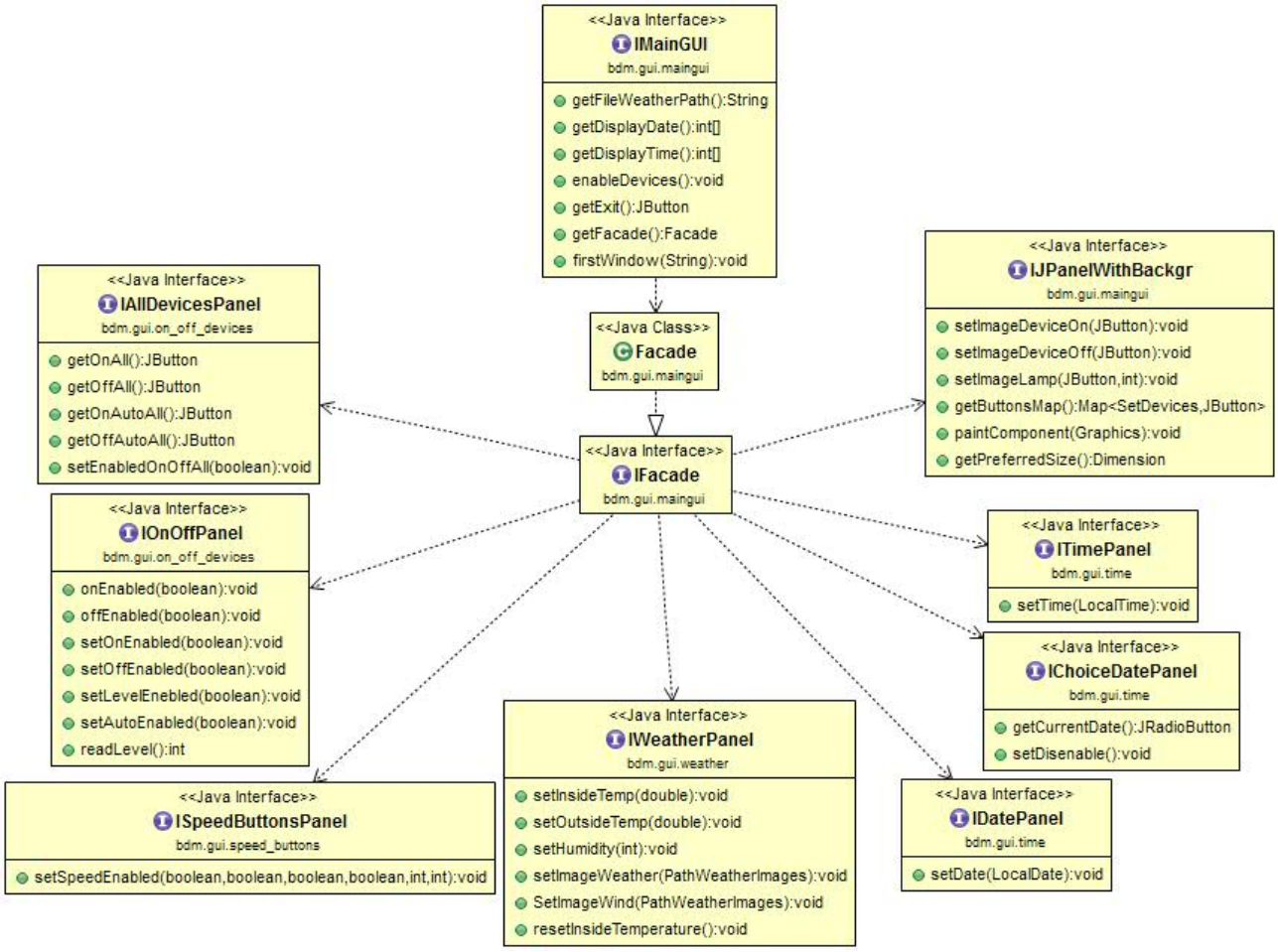
- IJPanelWithBack: si tratta dell'interfaccia che si occupa di posizionare all'interno del frame principale la mappa del piano dell'appartamento e tutti i suoi dispositivi, il cui stato (device on/ device off) è rappresentato attraverso lo switch di immagini di device distinte solo dal colore (device giallo per indicare che il device è acceso, rosso per indicare che il dispositivo è spento). Pertanto all'interno di tale interfaccia sono definiti i metodi per simulare l'accensione e lo spegnimento di ogni singolo device. Per la creazione di ogni singolo button Device è stato implementato il patter Builder, ottenendo in questo modo la creazione dell'oggetto mediante una chiamata a costruttore ordinata.



- IOnOffPanel: Tale interfaccia svolge il compito di definire il comportamento dei dispositivi in seguito alla pressione dei bottoni di accensione / spegnimento (istantanei o programmati), di settaggio del livello di stato (per i devices Lamp e Heater) e di inizio e fine gestione automatizzata;
- IPanelTime: utilizzata per l'eventuale settaggio iniziale dell'ora definita dall'utente;
- IPanelDate: utilizzata per l'eventuale settaggio iniziale della data definita dall'utente;
- ISpeedButton: i metodi di tale interfaccia devono permettere all'utente di selezionare la velocità della simulazione;
- IAllDevicePanel: usata per consentire di effettuare operazioni su un un raggruppamento di devices. (accensione e spegnimento istantanei o automatizzati);
- IWeatherPanel che permette di rappresentare le condizioni metereologiche in relazione allo sviluppo della simulazione.

Come già descritto in fase di analisi del problema, si è voluto realizzare un'interfaccia grafica che si adatti alla risoluzione del monitor e all'eventuale ridimensionamento del frame, una volta che l'applicazione è stata lanciata. Per questo motivo all'interno delle interfacce "DeviceButton", "IPanelImageWeather", "IJPanelWithBackgr" vengono definiti i metodi getPreferrideSize() e paintComponent(Graphics g), che permettono di modificare la dimensione dei button device. Inoltre per permettere l'adattamento dei vari pannelli alla dimensione dell'interfaccia grafica, per la quasi totalità dei pannelli che compongono la GUI è stato necessario settare il GridLayout come Layout di questi ultimi.

Qui di seguito viene riportato uno schema UML che rappresenta l'implementazione del Facade pattern.

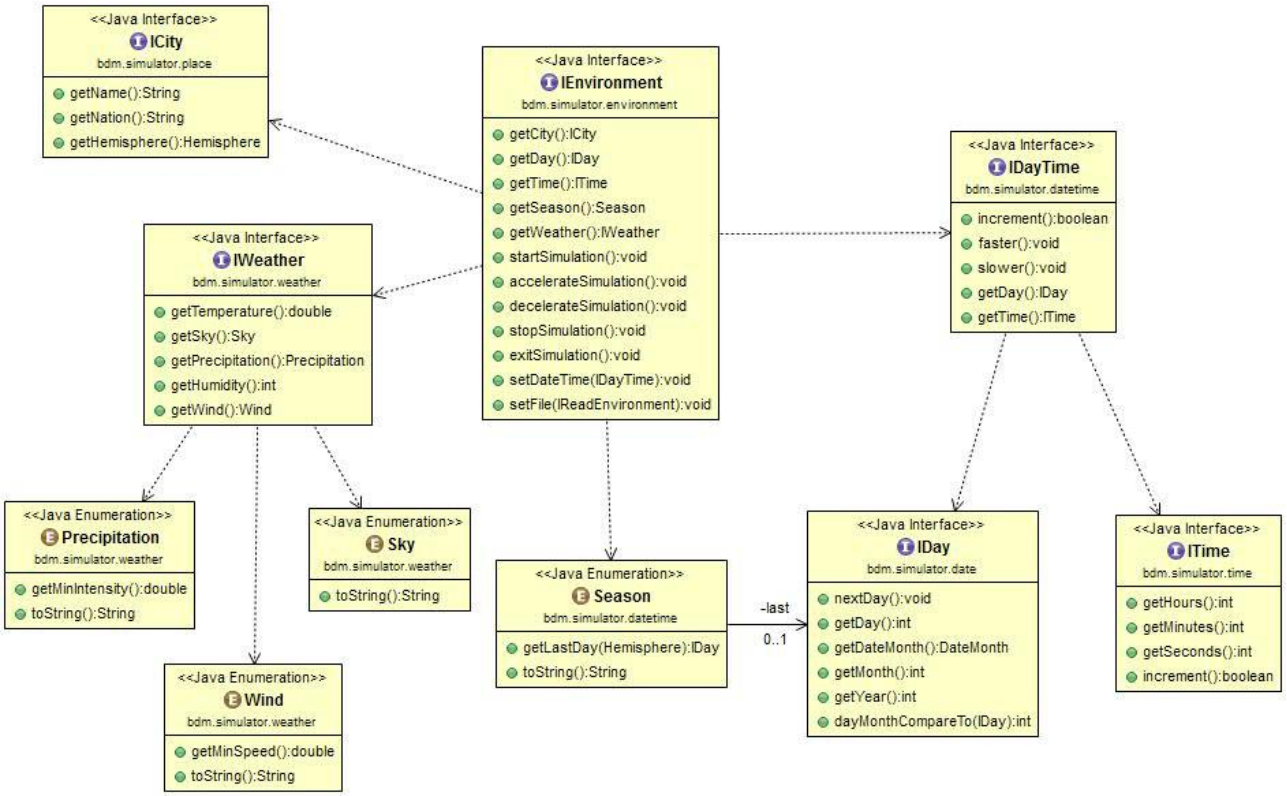


Da questo schema si evince che per utilizzare varie funzionalità dell'interfaccia è necessario richiamare prima il metodo getFacade. Tuttavia questa struttura implementativa consente di rendere più comprensibile e ordinata l'interfaccia IMainGUI.

SIMULATOR

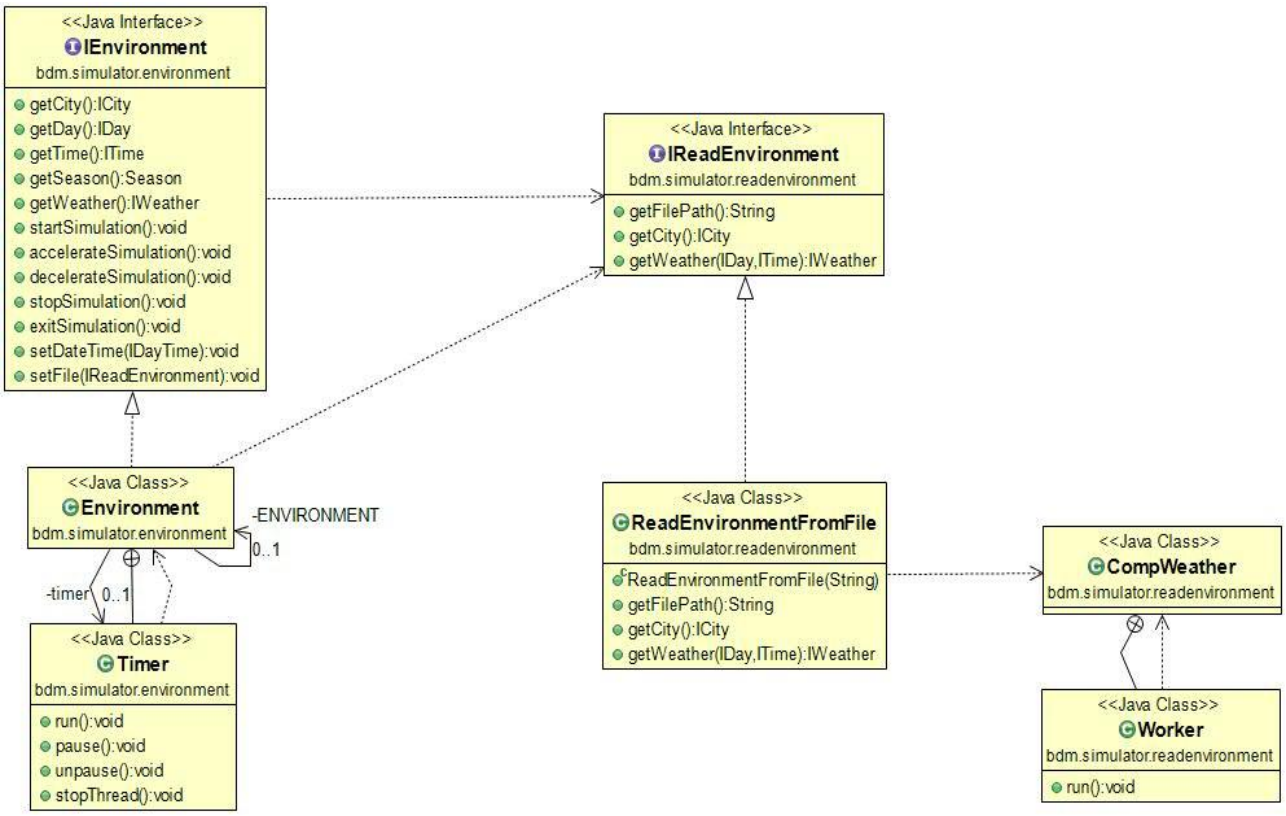
Il simulatore ha il compito della gestione dell'ambiente durante lo scorrere più o meno veloce del tempo. E' composto da un'interfaccia IEnvironment, che deve essere in grado di restituire in ogni momento lo stato dell'ambiente scelto, e modificarlo internamente attraverso simulazioni quanto più possibile realistiche.

La sua struttura interna di IEnvironment è la seguente:



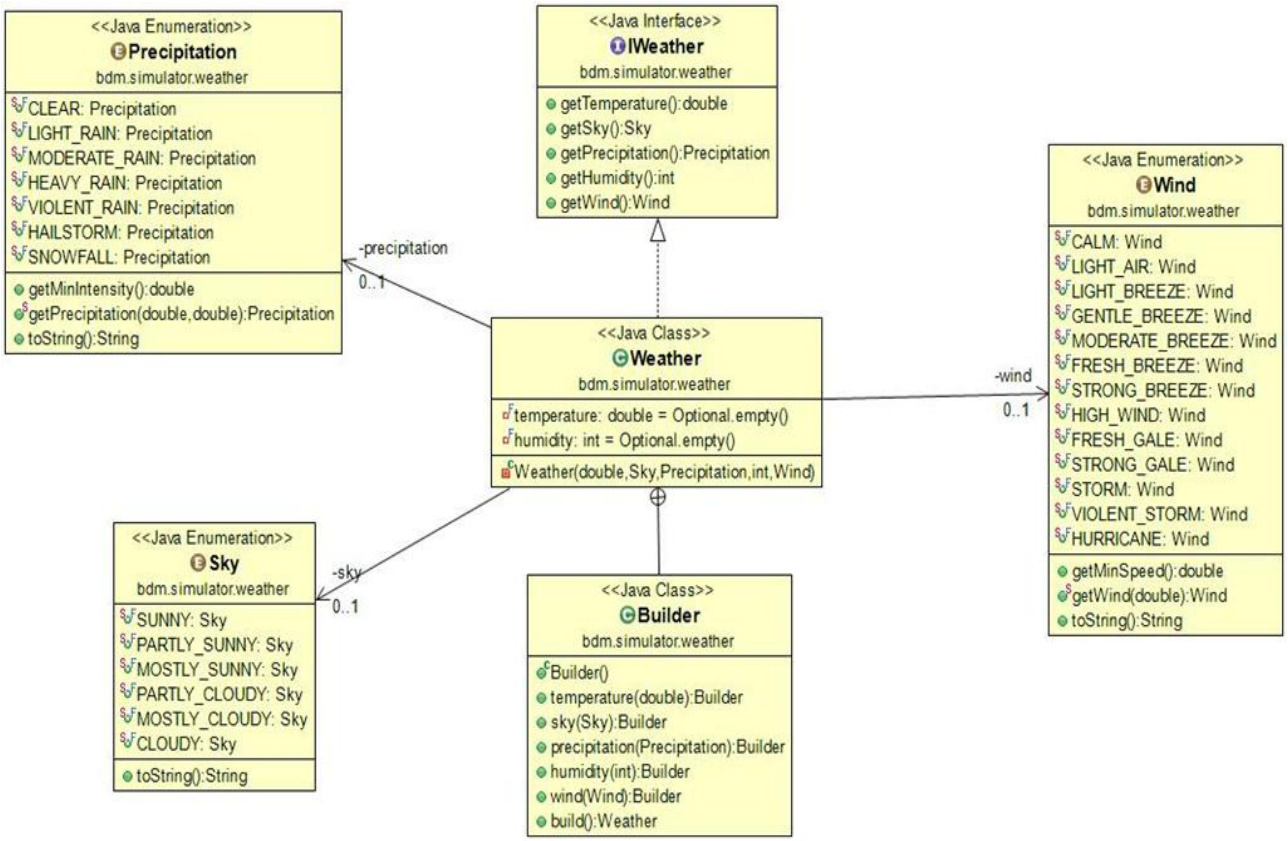
IEnvironment deve, dunque, tener traccia dell'ambiente che rappresenta. Deve essere in grado in ogni momento di restituire dati quali l'ora corrente dell'ambiente, la città nella quale ci si pone e una serie di variabili che rappresentano lo stato metereorologico corrente del sistema, oltre alla stagione corrente, facilmente ricavabile grazie alla data.

Non è però sufficiente che IEnvironment si limiti a fornire lo stato dell'ambiente in un preciso istante, ma deve anche essere in grado di modificare quell'ambiente in modo verosimile, tenendo conto dello scorrere del tempo e del luogo nel quale è situato. Per permettere ciò vi è la classe Environment, che non è altro che una concreta implementazione dell'interfaccia IEnvironment. Per tenere conto dello scorrere incessante del tempo sarà necessario l'ausilio di un apposito thread che andrà a modificare l'ora corrente dell'ambiente. Mentre per la variazione realistica del meteo viene utilizzato un oggetto dell'interfaccia IReadEnvironment che si preoccupa di recuperare i dati necessari a Environment. Questo processo può esere riassunto nello schema seguente:



La classe Environment è stata realizzata mediante il pattern Singleton, in quanto è ragionevole supporre come non ci possano essere più istanze dell'ambiente.
La classe Environment utilizza un oggetto di tipo Timer, una classe innestata che estende da Thread, per gestire l'avanzamento del tempo (esempio: ogni secondo Timer incrementa il tempo corrente dell'ambiente).
Si compone, inoltre, di un oggetto di tipo IReadEnvironment, che è incaricato di fornire all'Environment stesso il luogo di partenza (una certa città di una determinata nazione) e i dati meteo di cui Environment ha bisogno.
Nel nostro caso viene utilizzato un oggetto di tipo ReadEnvironmentFromFile che implementa l'interfaccia IReadEnvironment ed esegue una lettura da file dei dati riguardanti città di partenza e meteo. Il file in questione dovrà obbligatoriamente essere strutturato in un certo modo, con le prime 3 righe che serviranno per identificare il luogo iniziale mediante nome della città, nazione ed emisfero, e le restanti che indicano invece le condizioni meteorologiche di tutti i giorni in un certo range a piacere.
E' necessario però gestire anche il caso in cui l'utente inserisca una data per la quale non sono disponibili dati meteo (potrebbero essere inserite anche date future), in questo caso l'oggetto dovrà essere in grado di ricavare delle condizioni ambientali realistiche e per farlo utilizza un metodo statico fornito dalla classe CompWeather. Tale metodo scorre tutto il file passato in input elaborando i dati storici presenti e restituendoli alla classe ReadEnvironmentFromFile, che a questo punto potrà, attraverso delle simulazioni interne, generare il meteo di una certa data.

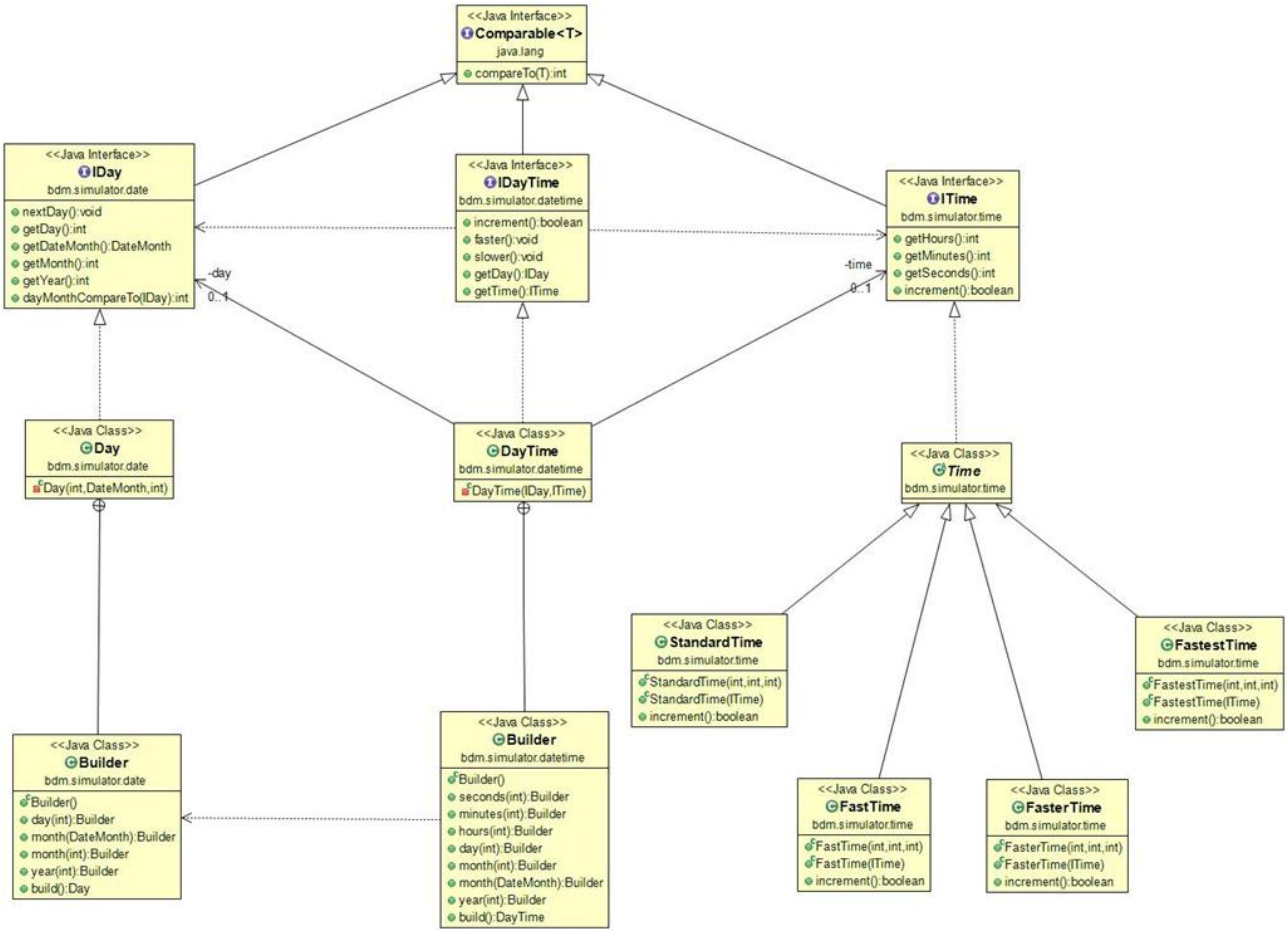
Le condizioni meteorologiche saranno organizzate in un oggetto comune restituito mediante un'interfaccia IWeather, strutturata come illustrato nello schema seguente:



La classe Weather è un'implementazione dell'interfaccia IWeather, costruita mediante l'uso di un pattern creazionale, il Builder. La costruzione di un oggetto di tipo Weather è, in questo modo, resa più fluida e intuitiva, oltre ad essere meno soggetta ad errori di errato passaggio di parametri. Weather si compone di vari oggetti che definiscono in un certo momento le condizioni meteorologiche principali, quali temperatura, situazione del cielo (sereno, nuvoloso, …), l'intensità delle precipitazioni come pioggia e neve (se presenti), l'umidità dell'ambiente e l'intensità del vento.

Per tener conto dell'avanzamento in del tempo in Environment è invece utilizzato un oggetto di tipo IDayTime, che tiene traccia sia della data corrente che dell'ora e può essere utilizzato per simulare in modo efficacie l'avanzamento di tempo e data nello stesso momento.

IDayTime è strutturata nel modo seguente:



L'interfaccia IDayTime è implementata attraverso una classe DayTime costruita mediante l'utilizzo di un Builder. Grazie a questo pattern creazionale è possibile gestire in modo più semplice eventuali errori nel passaggio dei parametri, effettuando dei controlli di validità prima di creare una nuova istanza dell'oggetto. Inoltre la creazione di un nuovo oggetto risulta più semplice e intuitiva, evitando all'utente di passare molti parametri al costruttore con l'obbligo di dover rispettare l'ordine di inserimento.
La classe DayTime si compone di un oggetto di un interfaccia IDay (implementato in una classe Day costruita sempre con l'ausilio di un Builder) e di un oggetto di tipo ITime.
L'interfaccia ITime è specificata in una classe astratta Time che fattorizza il codice comune alle classi StandardTime, FastTime, FasterTime e FastestTime. Ognuna di queste classi ha un concetto diverso di scorrimento del tempo, ad esempio StandardTime è la rappresentazione di un comune orologio che incrementa il tempo di un secondo alla volta, mentre FasterTime incrementa il tempo di un ora ogni volta.

IDayTime, IDay e ITime implementano l'interfaccia Comparable<T> e possono quindi essere confrontati senza problemi attraverso il metodo compareTo(T): int.

CAPITOLO 3: SVILUPPO

TESTING AUTOMATIZZATO

Sia le classi del modello che del simulatore sono state sottoposte a JUnit Test per verificarne il corretto funzionamento. Più nello specifico, sono state sottoposte a JUnit Test tutte le classi che si occupano della gestione del tempo e delle previsioni metereologiche, oltre alla verifica del corretto funzionamento generale (per quanto riguarda il simulatore) e tutte le classi che identificano i dispositivi e il loro funzionamento (per quanto riguarda il modello). Per quanto riguarda la GUI, invece, non è stato possibile effettuare JUnit Test. Pertanto, sono stati eseguiti accurati test manuali per verificarne la correttezza. I suddetti sono stati eseguiti nel seguente modo: test per l'accensione e lo spegnimento (istantanei o programmati) e per l'automazione, verificando il cambio d'immagine dei dispositivi; test per la correttezza dello scorrere del tempo e delle conseguenze che esso deve determinare (ovvero, aggiornamento di ora/data settate sulla gui e delle immagini rappresentanti le condizioni metereologiche); test per il ridimensionamento dell'interfaccia e , quindi, di tutti i componenti.

DIVISIONE DEI COMPITI E METODOLOGIA DI LAVORO

Per quanto riguarda la divisione dei compiti, le varie parti sono state implementate come segue:
-Manuel Peruzzi: implementazione del simulatore e di tutte le sue funzionalità;
-Simone Lunedei: implementazione dell'interfaccia grafica e di tutte le sue funzionalità;
-Alfredo Maffi: implementazione del modello e ,quindi, di tutti i Device e delle loro funzionalità;
A nostro parere la divisione dei compiti è stata piuttosto equa e coerente, siccome non vi sono dipendenze tra le componenti (grazie all'utilizzo del pattern architetturale MVC).
La parte di interfacciamento tra i blocchi individuali è stata svolta in comune , e ha preso il nome di Controller.
Il Controller mantiene un riferimento all'interfaccia del Model, un riferimento all'interfaccia della View e un riferimento all'interfaccia del Simulatore. Attraverso i suddetti, esso gestisce le scelte da parte dell'utente e modifica View o Model anche in base ai dati forniti dal Simulatore.
Purtroppo, durante l'interfacciamento, si è presentata la necessità di implementare un'interfaccia generale per il modello che in precedenza non era stata ritenuta necessaria. In ogni caso, tale implementazione non ha creato grossi problemi e ha contribuito a garantire la corretta implementazione del pattern MVC.
Infine, è stato utilizzato, come DVCS, Mercurial (utilizzo semplificato grazie al plugin "HGE" di Eclipse) e abbiamo fatto largo uso del servizio di hosting offerto da BitBucket per la condivisione del progetto.

NOTE DI SVILUPPO

Durante l'implementazione del progetto si sono presentati diversi aspetti critici:
Per quanto riguarda il modello, la rottura della maggior parte dei dispositivi è determinata da una formula che genera un numero random (in dipendenza dal numero di volte che il dispositivo è stato acceso) e lo confronta con una costante. Se il primo numero è maggiore del secondo, allora il device risulta rotto. Detto questo, la formula non è ideale e non siamo riusciti a trovarne una migliore. Inoltre, nel dispositivo "Heater" , per calcolare la temperatura interna viene fatta una media pesata tra quella dell'Heater stesso e quella esterna. Come detto sopra, anche qui la formula non è ideale, ma , comunque, risulta approssimativamente buona.
Per quanto riguarda la view, il problema principale è stato il ridimensionamento automatico della view stessa e di tutti i suoi componenti (immagini , bottoni e pannelli).
Per quanto riguarda il simulatore, il problema principale è stato la simulazione delle condizioni metereologiche, in quanto ha richiesto la creazione di appositi metodi in grado
di effettuare un'estrazione che permettesse, dato un intervallo, di privilegiare alcuni range di valori rispetto ad altri.
Inoltre, durante l'implementazione delle singole parti è stato ritenuto conveniente l'utilizzo di codice preesistente, dichiarato come segue:
-Model: utilizzo delle classi LimitedLamp e ExpirationTimeLamp del prof. Mirko Viroli (anche se poi leggermente modificate)
-View: utilizzo della classe ControllerGUI del prof. Danilo Pianini (anche se in seguito rimodellata)

CAPITOLO 4: COMMENTI FINALI

CONCLUSIONE

Siamo complessivamente soddisfatti del lavoro svolto e, grazie a questo progetto, abbiamo avuto l'opportunità di migliorare le nostre competenze nel lavoro di gruppo e le nostre conoscenze in generale. Per quanto riguarda il monte ore, riteniamo di aver rispettato il tempo a disposizione imposto dal docente.

L'applicazione è reattiva e sempre a disposizione dell'utente; viene fornita una corretta simulazione dell'andamento temporale e atmosferico; l'applicazione è facilmente estendibile e riutilizzabile: è possibile ,infatti , utilizzare un qualunque file contenente informazioni metereologiche in modo tale che questa vengano lette e utilizzate dal simulatore; è possibile settare l'elaborazione dei dati atmosferici per ora anziché per giorno, mediante una lieve modifica ad un metodo; risulta non difficoltoso cambiare il modo di lettura dei dati, ad esempio sarebbe sufficiente creare una classe ReadEnvironmentFromDatabase che implementi l'interfaccia IReadEnvironment per leggere i dati da un database piuttosto che da un file, senza dover apportare alcuna modifica a Environment; è possibile creare nuovi tipi di Devices, utilizzando le interfacce fornite per il massimo riutilizzo di codice, così come è possibile aggiungere nuove immagini purché vengano inseriti i rispettivi path relativi nelle apposite Enum.

Purtroppo, durante ogni ridimensionamento effettuato dall'utente, le immagini dei devices si muovono senza un apparente senso logico all'interno della GUI. Ovviamente però, una volta concluso il ridimensionamento, le immagini risultano adattate e situate correttamente in proporzione alla GUI;

APPENDICE A

GUIDA UTENTE

All'avvio dell'applicazione, è necessario selezionare un file di testo adatto per la lettura delle condizioni metereologiche (ne viene fornito uno valido assieme al progetto). Una volta selezionato il file, è possibile settare come data corrente quella attuale oppure sceglierne una a piacere. A questo punto l'utente è inviato a premere il pulsante "play" per avviare la simulazione (una volta premuto, la data non sarà più modificabile).
Ora l'utente può interagire con l'intera interfaccia: può velocizzare lo scorrimento del tempo attraverso gli appositi SpeedButtons e può interagire con ogni singolo Device.
Per operare su un determinato Device, occorre selezionarlo, cliccandoci sopra, e quindi selezionare un'azione a piacere nel pannello posizionato sotto la piantina.
Ogniqualvolta che un device viene selezionato, il simulatore si ferma per dare tempo all'utente di effettuare le operazioni desiderate sul device stesso.
Attraverso "Set Prog On" è possibile settare l'accensione programmata del Device selezionato ad una data futura, così come per lo spegnimento programmato mediante "Set Prog Off". Mediante il pulsante "Auto Management" è possibile attivare e disattivare la gestione automatizzata del device selezionato.
I pulsanti "Turn On All" e "Turn Off All" , rispettivamente, accendono e spengono tutti i devices, mentre i pulsanti "Set On Auto All" e "Set Off Auto All", rispettivamente, attivano e disattivano la gestione automatizzata di tutti i devices che la supportano.
Se un device si rompe, sarà fornita la possibilità di ripararlo.