Relazione Progetto Domo

Corso di Programmazione ad oggetti 2014/15

De Mattia Simone

Falzaresi Stefano

Versari Marco

2 maggio 2015

# Sommario

Il progetto è volto alla creazione di un sistema di gestione della domotica all’interno di un appartamento.

Questo progetto è volto alla creazione di un sistema per la gestione della domotica in un appartamento.

Lo scopo è quello di fornire all’utente un interfaccia facile e semplificata che lo aiuti nella gestione di tutti gli apparati di sicurezza installati all’interno dell’abitazione.

# Indice

1. **Analisi 2**
   1. Requisiti . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   2. Problema . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2. **Design 6**
   1. Architettura . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   2. Design dettagliato . . . . . . . . . . . . . . . . . . . . . . . . . 8

## 3 Sviluppo 16

3.1 Testing automatizzato . . . . . . . . . . . . . . . . . . . . . . 16

3.2 Divisione dei compiti e metodologia di lavoro . . . . . . . . . . 17

3.3 Note di sviluppo . . . . . . . . . . . . . . . . . . . . . . . . . 18

## 4 Commenti finali 20

4.1 Conclusioni e lavori futuri . . . . . . . . . . . . . . . . . . . . 20

4.2 Difficolta` incontrate e commenti per i docenti . . . . . . . . . 20

**A Guida utente 21**

**Capitolo 1**

# Analisi

## 1.1 Requisiti

L’applicazione sviluppata dovrà permettere all’utente la creazione di un appartamento virtuale composto da stanze all’interno delle quali verranno inseriti dei dispositivi di svariate tipologie (ampliabili dinamicamente dall’utente).

L’attivazione di ogni dispositivo inserito potrà essere gestita sia in modo manuale che in modo automatico (attraverso delle schedulazioni).

L’utente sarà agevolato nella gestione da processi di salvataggio e ripristino automatizzati della configurazione creata.

Una interfaccia grafica semplice e intuitiva agevolerà l’utente in tutti questi processi e permetterà di avere una visuale completa dello stato dell’intero appartamento.

## 1.2 Problema

Le principali problematiche che dovranno essere affrontate sono quelle che riguardano la gestione “dinamica” dei dispositivi in quanto si vuol far sì che l’utente ed i futuri implementatori possano gestire in modo autonomo il set di dispositivi da utilizzare all’interno dell’applicazione.

Questa dinamicità va quindi ad impattare anche su tutte le procedure che gestiscono la grafica o il salvataggio e ripristino dei dati.

//inserire problematiche di Simone

**Capitolo 2**

# Design

## 2.1 Architettura

Principalmente si è deciso di utilizzare il pattern MVC (Model, View, Controller) in modo da poter generalizzare l’utilizzo di questo applicativo su dispositivi con diversi tipologie di interazione, grazie all’utilizzo di questo pattern infatti implementatori futuri potranno scegliere di riprogettare l’impatto grafico per permetterne il funzionamento su dispositivi embedded quali ad esempio Raspberry Pi semplicemente riprogrammando la parte View.

Si è scelto poi di utilizzare anche il Factory pattern che attraverso l’uso della Reflection ci permette di caricare le classi dei dispositivi in modo dinamico.

### Elementi positivi

* Si mostrano pochi, mirati schemi UML dai quali si deduce con chiarezza quali sono le parti principali del software e come interagiscono fra loro.
* Si mette in evidenza se e come il pattern architetturale model-viewcontroller `e stato applicato, anche con l’uso di un UML che mostri le interfacce principali ed i rapporti fra loro.
* Gli schemi UML contengono soprattutto (eventualmente anche solo) interfacce.
* Si discute se sia semplice o meno, con l’architettura scelta, sostituire in blocco la view senza toccare minimamente il controller. Va da s´e che, se cambiare la UI si trasforma in un bagno di sangue che impatta controller o modello, il design architetturale non `e stato fatto in modo adeguato.

### Elementi negativi

* MVC non `e applicato in maniera corretta
* L’architettura `e fatta in modo che sia impossibile riusare il modello per un software diverso che affronta lo stesso problema.
* L’architettura `e tale che l’aggiunta di una funzionalita` sul controller impatta pesantemente su view e/o modello.
* L’architettura `e tale che la sostituzione in blocco della view impatta sul controller o, peggio ancora, sul modello.
* Si presentano UML caotici, difficili da leggere.
* Si presentano UML in cui sono mostrati elementi di dettaglio non appartenenti all’architettura, ad esempio includenti campi o con metodi che non interessano la parte di interazione fra le componenti principali del software.
* Si presentano schemi UML con classi (nel senso UML del termine) che “galleggiano” nello schema, non connesse, ossia senza relazioni con il resto degli elementi inseriti.
* Si presentano elementi di design di dettaglio, ad esempio tutte le classi e interfacce del modello o della view.
* Si discutono aspetti implementativi, ad esempio eventuali librerie usate oppure dettagli di codice.

### Esempio di buon diagramma UML dell’Architettura

In Figura 2.1 `e esemplificato un buon diagramma UML architetturale. Le sue caratteristiche positive sono:

* Mostra solo interfacce.
* Le interfacce sono semplificate, e mostrano solo le parti che interessano la parte architetturale.
* Non sono presenti elementi galleggianti, tutti gli elementi sono legati fra loro.
* Lo schema `e minimale, non caotico, facile da leggere.
* Lo schema non mostra elementi implementativi.
* Lo schema non mostra elementi non strettamente correlati all’interazione fra le macro-parti del software, ad esempio di tutto il modello si mostra solo l’entry point.

## 2.2 Design dettagliato

In questa sezione si possono approfondire alcuni elementi del design con maggior dettaglio. Mentre ci attendiamo principalmente (o solo) interfacce negli schemi UML delle sezioni precedenti, in questa sezione `e necessario scendere in maggior dettaglio presentando la struttura di alcune sottoparti rilevanti dell’applicazione. E molto importante che, descrivendo un problema, quan-` do possibile si mostri che non si `e re-inventata la ruota ma si `e applicato un design pattern noto.

### Elementi positivi

* Si mostrano gli aspetti di design piu` rilevanti dell’applicazione, mettendo in luce la maniera in cui si `e costruita la soluzione ai problemi descritti nell’analisi.
* Si ignorano dettagli di poco conto.
* Si mostrano le principali interazioni fra le varie componenti che collaborano alla soluzione di un determinato problema.

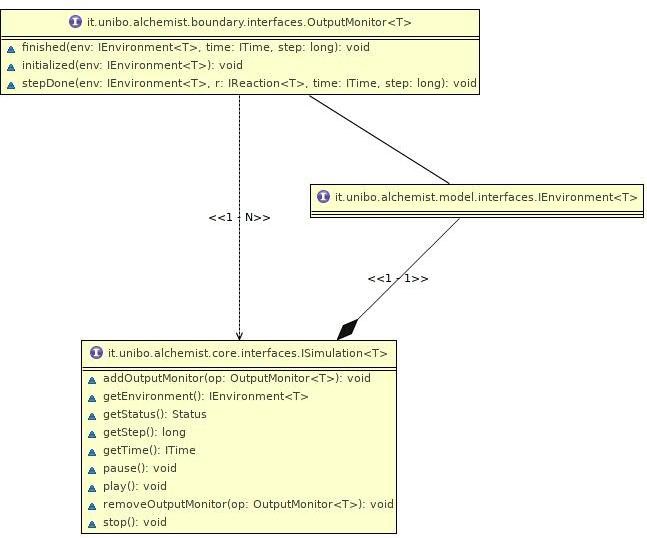


Figura 2.1: Schema UML architetturale. In questo software, una

ISimulation `e associata ad un IEnvironment. Una ISimulation puo` avere uno o piu` OutputMonitor, che possono essere (de)registrati usando i metodi appositi. OutputMonitor dipende da IEnvironment, significa che ha bisogno di sapere come `e fatto quest’ultimo. In questo schema, OutputMonitor rappresenta la UI, che funziona in sola lettura e accede direttamente al model, che ha come punto di ingresso IEnvironment, quando notificata dal controller, rappresentato da ISimulation

* Si tralasciano aspetti strettamente implementativi non rilevanti, non mostrandoli negli schemi UML e non descrivendo aspetti implementativi.
* Si identificano numerosi design pattern.
* Ciascun design pattern identificato presenta una piccola descrizione del problema calato nell’applicazione, uno schema UML che ne mostri la concretizzazione nelle classi del progetto, ed una breve descrizione della motivazione per cui tale pattern `e stato scelto.

### Elementi negativi

* Si tratta in modo prolisso, classe per classe, il software realizzato.
* Non si presentano schemi UML esemplificativi.
* Non si individuano design pattern.
* Si producono schemi UML caotici e difficili da leggere, che comprendono inutili elementi di dettaglio.
* Si presentano schemi UML con classi (nel senso UML del termine) che “galleggiano” nello schema, non connesse, ossia senza relazioni con il resto degli elementi inseriti.
* Il design del modello risulta scorrelato dal problema descritto in analisi.

### Esempi di buoni diagrammi UML del design di dettaglio

In Figura 2.2 viene mostrato un UML che appartiene al design di dettaglio, in cui viene mostrato come `e strutturato il model di un’applicazione. Tale model, essendo comunque corposo, viene mostrato solo a livello di interfacce. Se uno schema del problema `e presente nella parte di analisi, allora questo secondo schema dovrebbe essere una sua rivisitazione piu` ricca di dettagli.

Nel caso in cui si vogliano mostrare alcune parti che, se inserite nello schema precedente, renderebbero il tutto difficile da leggere e interpretare, `e preferibile “zoomare” su una parte del proprio progetto come `e stato fatto in Figura 2.3. Esattamente come nessun ingegnere meccanico presenta un solo foglio con l’intero progetto di una vettura di Formula 1, ma molteplici fogli di progetto che mostrano a livelli di dettaglio differenti le varie parti della vettura e le modalit`a di connessione fra le parti, cos`ı ci aspettiamo che voi, futuri ingegneri informatici, ci presentiate prima una visione globale del progetto, e via via siate in grado di entrare nel dettaglio delle specifiche parti, riuscendo a metter da parte le parti del progetto che non interessano la parte specifica. Per continuare il parallelo con la vettura di Formula 1, se nei fogli che mostrano il design delle sospensioni anteriori appaiono pezzi che appartengono al volante o al turbo, vuol dire che c’`e qualche grosso problema di design architetturale.

Infine, in Figura 2.4 viene mostrata una modalita` con cui si puo` mostrare che alcuni problemi non sono stati risolti re-inventando la ruota ma adattando una soluzione precedente. Nel caso della figura, si mostra come il pattern strategy `e stato concretizzato in una applicazione. Per i design pattern che il team di sviluppo riesce ad identificare, ci aspettiamo un simile schema che ce ne mostri la loro applicazione.

### Esempio di pessimo diagramma UML

In Figura 2.5 `e mostrato il modo sbagliato di fare le cose. Questo schema `e fatto male perch´e:

* E caotico.`
* E difficile da leggere e capire.`
* Vi sono troppe classi, e non si capisce bene quali siano i rapporti che intercorrono fra loro.
* Si mostrano elementi implementativi irrilevanti, come i campi e i metodi privati nella classe AbstractEnvironment.
* Se l’intenzione era quella di costruire un diagramma architetturale, allora lo schema `e ancora piu` sbagliato, perch´e mostra pezzi di implementazione.
* Una delle classi, in alto al centro, galleggia nello schema, non connessa a nessuna altra classe, e di fatto costituisce da sola un secondo schema UML scorrelato al resto
* Le interfacce presentano tutti i metodi e non una selezione che aiuti il lettore a capire quale parte del sistema si vuol mostrare.

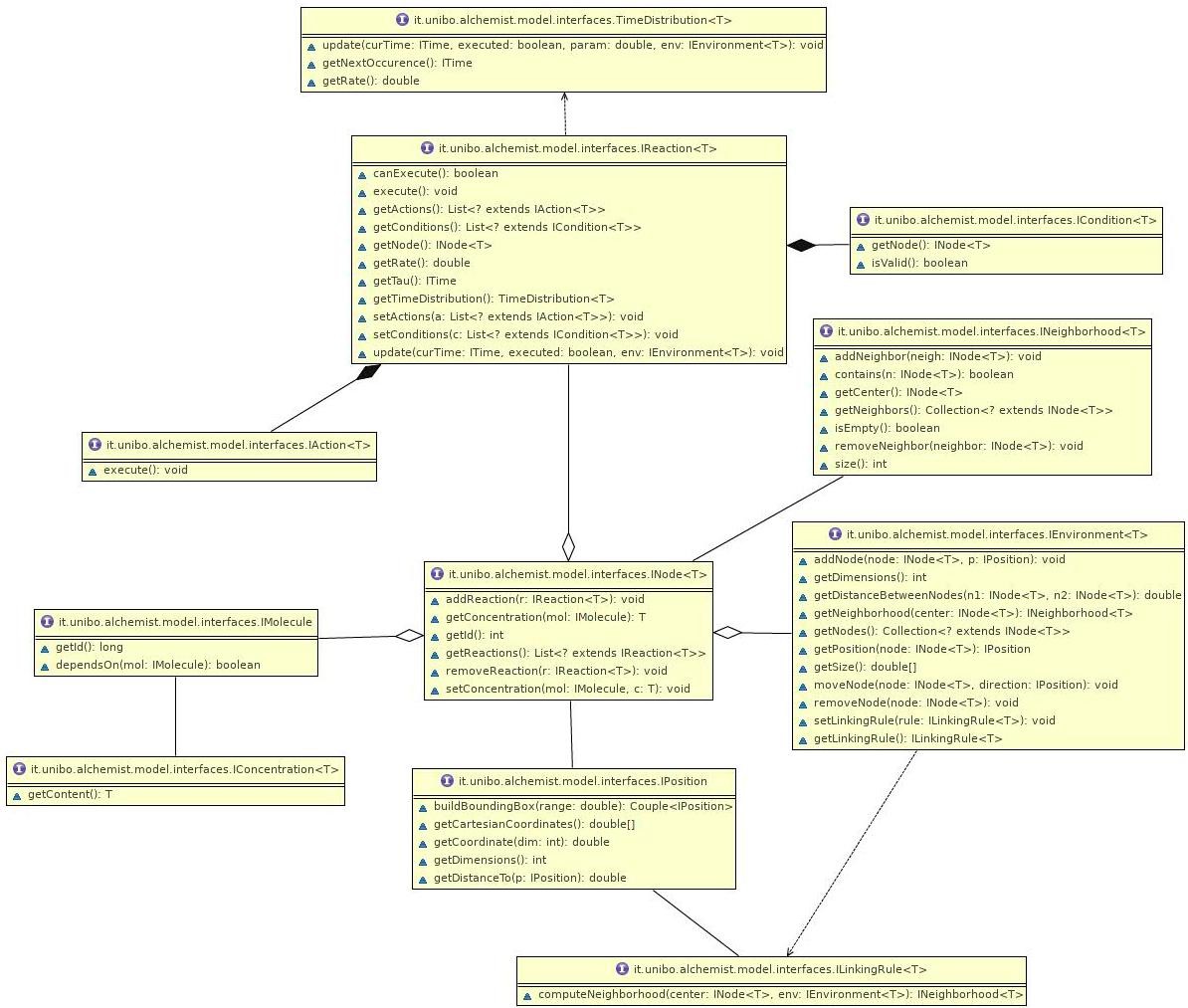


Figura 2.2: Schema UML di dettaglio del model di una applicazione. Dallo schema, si deducono numerose informazioni utili. IEnvironment contiene piu` INode. INode `e associato con IPosition, e IPosition `e associata con LinkingRule. IEnvironment dipende da LinkingRule (probabile pattern Strategy). INode `e associato con INeighborhood Ogni INode puo` contenere delle IMolecule. IMolecule sono associate a IConcentration. Ogni INode puo` contenere delle IReaction. Ogni IReaction `e composta obbligatoriamente da IAction e ICondition, e per funzionare dipende da TimeDistribution (altro probabile pattern Strategy). Da questo schema, `e possibile capire come `e costruito il modello dell’applicazione, in una relazione organica questo dovrebbe essere una estensione e integrazione di quanto mostrato nell’eventuale schema UML descrivente le entit`a del problema.

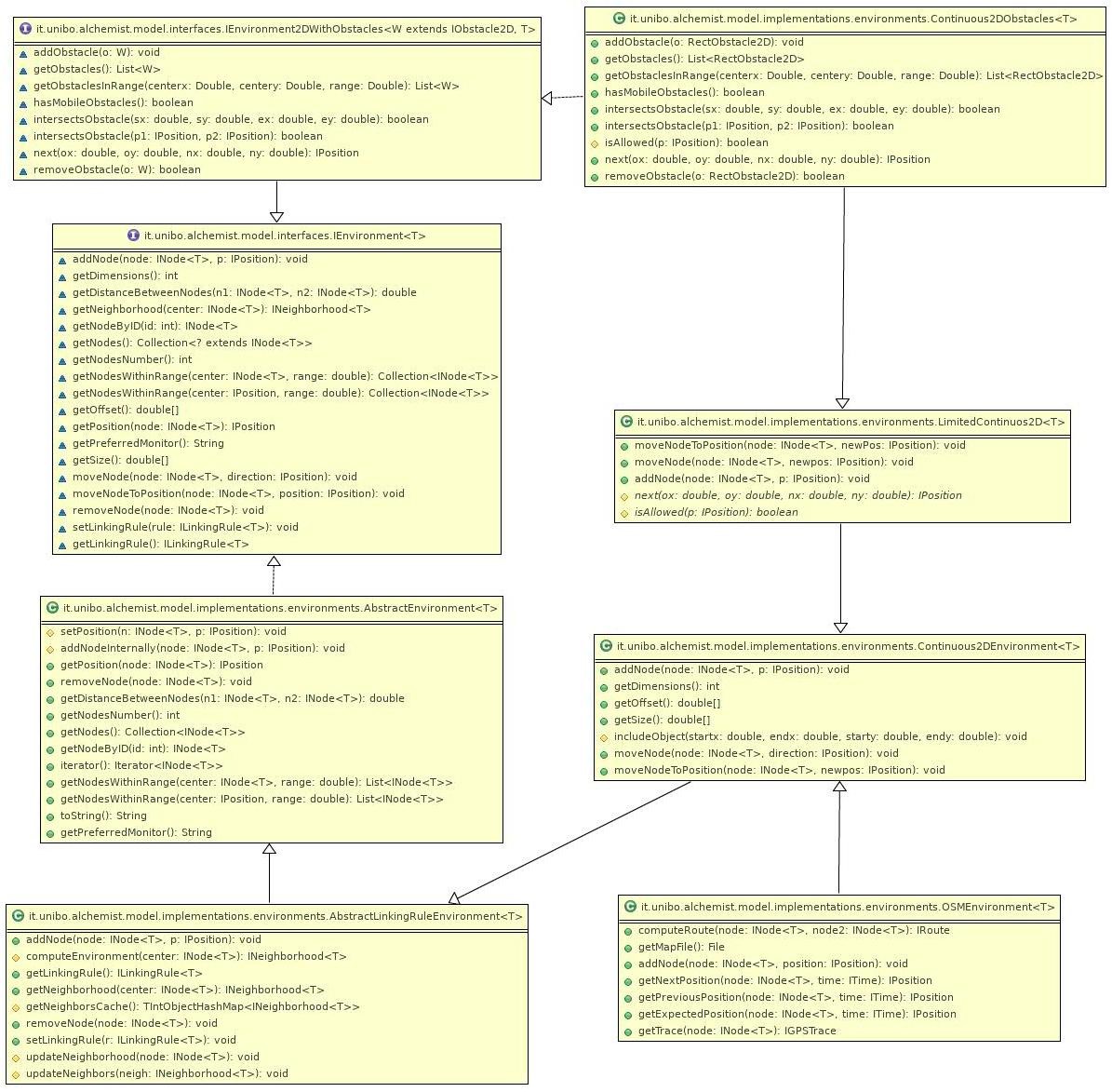


Figura 2.3: Schema UML di dettaglio di una sottoparte del model di una applicazione. Schemi come questo vengono usati per mostrare l’implementazione di pezzi specifici del software, magari particolarmente delicati. Lo schema rappresenta una sorta di “zoom” dentro ad una specifica parte del software, e di conseguenza riesce a mostrare un maggiore livello di dettaglio. Nello schema in oggetto, `e mostrata la gerarchia di un ambiente e di un ambiente con ostacoli. Essi sono descritti da due interfacce, e quella piu` generica viene estesa da due classi astratte, una responsabile di implementare parti del tutto generiche e un’altra responsabile di gestire delle “ILinkingRule”. Da queste due classi astratte eredita una classe concreta che rappresenta un ambiente bidimensionale, dal quale a sua volta viene giu` un ambiente ccon supporto a OpenStreetMaps (si vedono i metodi per calcolare delle route). Dagli ambienti continui bidimensionali derivano ambienti con limitazioni (ad esempio con una dimensione finita), dai quali derivano ambienti con ostacoli, che implementano anche la sotto-interfaccia che aggiunge le funzionalita` per la gestione di ostacoli.

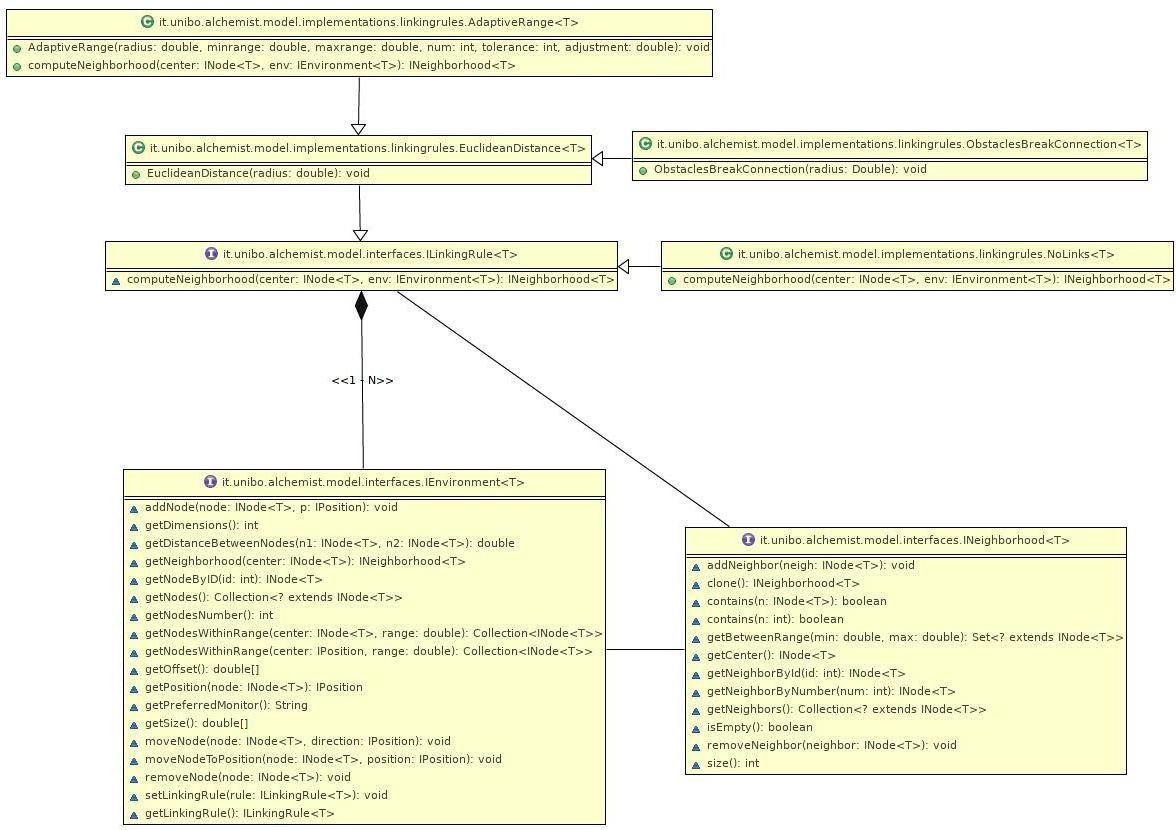


Figura 2.4: Schema UML di dettaglio dell’istanziazione di un design pattern all’interno di una applicazione. In questo caso, si tratta di un pattern strategy, in cui IEnvironment utilizza una ILinkingRule (oggetto strategy) per ottenere un metodo per il calcolo di un INeighborhood. Si puo` anche dedurre che la stessa strategia puo` esser condivisa da piu` ambienti (la molteplicit`a `e 1-N). Nello schema, si mostra anche una gerarchia di strategie implementate: nessun link, distanza euclidea, distanza euclidea con ostacoli che rompono i collegamenti, e una strategia adattativa, probabilmente piu` complicata delle precedenti.

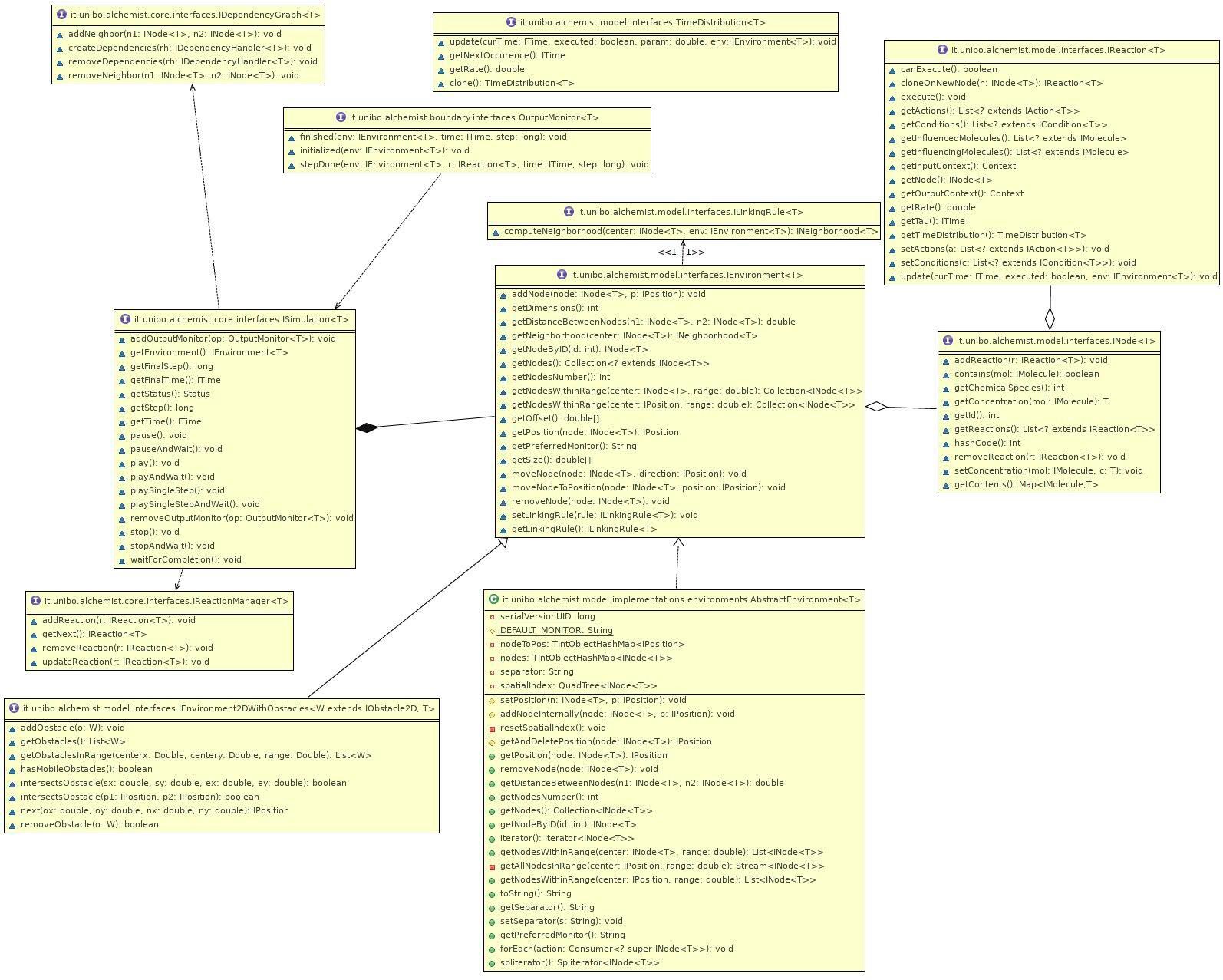


Figura 2.5: Schema UML mal fatto. Don’t try this at home.

**Capitolo 3**

# Sviluppo

## 3.1 Testing automatizzato

### Elementi positivi

* Si descrivono molto brevemente i componenti che si `e deciso di sottoporre a test automatizzato.
* Si utilizzano suite specifiche (e.g. JUnit) per il testing automatico.
* Se sono stati eseguiti test manuali di rilievo, si elencano descrivendo brevemente la ragione per cui non sono stati automatizzati. Ad esempio, se tutto il team sviluppa e testa su uno stesso sistema operativo e si sono svolti test manuali per verificare, ad esempio, il corretto funzionamento dell’interfaccia grafica o di librerie native su altri sistemi operativi, pu`o avere senso menzionare la cosa.

### Elementi negativi

* Non si realizza alcun test automatico.
* Si descrive un testing di tipo manuale in maniera prolissa.
* Si descrivono test effettuati manualmente che sarebbero potuti essere automatizzati, ad esempio descrivendo che si `e usata l’applicazione manualmente.

## 3.2 Divisione dei compiti e metodologia di lavoro

Questa sezione si presentera` ovviamente un po’ piu` ricca nel caso in cui il progetto sia stato realizzato in gruppo. Ci aspettiamo, leggendola, di identificare chi ha realizzato quale parte del software, e come `e stato svolto il lavoro di integrazione. Se il lavoro `e stato svolto da un singolo, alcuni punti ovviamente non potranno essere svolti (ad esempio la descrizione del processo di integrazione), ed altrettanto ovviamente alcuni punti negativi non si applicheranno.

### Elementi positivi

* Si identifica con precisione il ruolo di ciascuno all’interno del gruppo, ossia su quale parte del progetto ciascuno dei componenti si `e concentrato maggiormente.
* La divisione dei compiti `e equa, ossia non vi sono membri del gruppo che hanno svolto molto piu` lavoro di altri
* La divisione dei compiti `e coerente, ossia le dipendenze fra le parti sviluppate siano minime
* Si identifica quale parte del software `e stato sviluppato da tutti i componenti insieme.
* Si spiega in che modo si sono integrate le parti di codice sviluppate separatamente, evidenziando eventuali problemi. Ad esempio, una strategia `e convenire sulle interfacce da usare (ossia, occuparsi insieme di stabilire l’architettura) e quindi procedere indipendentemente allo sviluppo di parti differenti. Una possibile problematica potrebbe essere una dimenticanza in fase di design architetturale che ha costretto ad un cambio e a modifiche in fase di integrazione. Una situazione simile `e la norma nell’ingegneria di un sistema software non banale, ed il processo di progettazione top-down con raffinamento successivo `e il cos`ı detto processo “a spirale”.
* Si descrive in che modo `e stato impiegato il DVCS.

### Elementi negativi

* Non si chiarisce chi ha fatto cosa.
* Tutto il progetto `e stato svolto lavorando insieme invece che assegnando una parte a ciascuno.
* Non viene descritta la metodologia di integrazione delle parti sviluppate indipendentemente.
* Uso superficiale del DVCS.

## 3.3 Note di sviluppo

Questa sezione `e opzionale.

### Elementi positivi

* Si descrivono aspetti particolarmente complicati o rilevanti relativi all’implementazione, ad esempio, in un’applicazione performance critical, un uso particolarmente avanzato di meccanismi di caching, oppure l’implementazione di uno specifico algorigmo.
* Se si `e utilizzato un particolare algoritmo, se ne cita la fonte originale. Ad esempio, se si `e usato Mersenne Twister per la generazione dei numeri random, si cita [1].
* Si identificano eventuali librerie utilizzate, parti di codice prese da altri progetti, dal web, o comunque scritte in forma originale da altre persone. In tal senso, si ricorda che agli ingegneri non `e richiesto di re-inventare la ruota continuamente: se ci cita debitamente la sorgente, non solo `e tollerato ma `e consigliato fare uso di librerie o anche di snippet di codice per risolvere velocemente problemi non banali. Nel caso in cui si usino snippet di codice di qualit`a discutibile, oltre a menzionarne l’autore originale si invitano gli studenti ad adeguare tali parti di codice agli standard e allo stile del progetto.

### Elementi negativi

* Si descrivono aspetti di scarsa rilevanza, o si scende in dettagli inutili.
* Sono presenti parti di codice sviluppate originalmente da altri che non vengono debitamente segnalate. In tal senso, si ricorda agli studenti che i docenti hanno accesso a tutti i progetti degli anni passati, a Stack Overflow, ai principali blog di sviluppatori ed esperti Java e ai blog dedicati allo sviluppo di soluzioni e applicazioni (inclusi blog dedicati ad Android e allo sviluppo di videogame). Conseguentemente, `e *molto* conveniente *citare* una fonte ed usarla invece di tentare di spacciare per proprio il lavoro di altri.

**Capitolo 4**

# Commenti finali

In quest’ultimo capitolo si tirano le somme del lavoro svolto e si delineano eventuali sviluppi futuri.

## 4.1 Conclusioni e lavori futuri

### Cosa scrivere

* Il team si sforza di autovalutare il lavoro svolto, cercandone i punti di forza e di debolezza.
* Nel caso in cui il team pensa che il lavoro possa essere completato o esteso ulteriormente, si descrive brevemente verso che direzione portarlo.

## 4.2 Difficolt`a incontrate e commenti per i docenti

Questa sezione, opzionale, puo` essere utilizzata per segnalare ai docenti eventuali problemi o difficolt`a incontrate nel corso o nello svolgimento del progetto. E possibile che alcuni dei commenti vengano utilizzati per migliorare il` corso in futuro: sebbene non andr`a a vostro beneficio, potreste fare un favore ai vostri futuri colleghi. Ovviamente il contenuto della sezione non impattera` il voto finale.

**Appendice A**

# Guida utente

Capitolo in cui si spiega come utilizzare il software. Nel caso in cui il suo uso sia del tutto banale, tale capitolo puo` essere omesso.

## Elementi positivi

• Si istruisce in modo semplice l’utente sull’uso dell’applicazione, eventualmente facendo uso di schermate e descrizioni.

## Elementi negativi

* Si descrivono in modo eccessivamente minuzioso tutte le caratteristiche, anche minori, del software in oggetto.
* Manca una descrizione che consenta ad un utente qualunque di utilizzare almeno le funzionalita` primarie dell’applicativo.

# Bibliografia

[1] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, Jan. 1998.