Progetto di Paradigmi di Programmazione (parte Java) o Programmazione ad Oggetti 2022/2023

Il progetto va svolto in un Gruppo di massimo **3 persone**. Il progetto deve essere consegnato **alcuni giorni prima dell'esame** (vi metterò un contenitore nel sito del corso) e vale 1/3 del voto complessivo (per il corso di paradigmi di programmazione) e 1/2 del voto complessivo (per il corso di programmazione ad oggetti).

Vengono fornite due proposte, una semplice (voto massimo 26) ed una un po' più difficile (voto massimo 32). Ogni gruppo deve svolgere solo una delle due proposte.

Vi ricordo che il voto potrà essere "a scalare" in base ai tentativi di esame: se la prima discussione risulterà insufficiente, a seconda della gravità degli errori la seconda potrà partire da un voto base più basso e così via.

Questo progetto resterà valido per tutto l'anno accademico 22/23 (appello di novembre 2023 incluso).

Requisiti generali

Tutti i progetti presentati **devono** essere **sviluppati secondo la programmazione Object Oriented** e secondo il **pattern MVC** (anche in assenza di GUI: si veda sotto), e seguire questa struttura e requisiti:

a. Netta e totale separazione tra le classi relative al "modello" che risolve il problema affrontato (che deve poter funzionare da codice anche in assenza di vista/e e controllo/i) e quelle relative all'interfaccia con l'utente (pattern MVC). **Non saranno valutati laboratori nei quali il modello non è indipendente da vista e controller**. Come valuto se il mio progetto rispetta i requisiti: devo essere in grado di scrivere un main nel quale inizializzo la/le classe/i principali del modello ed eseguo tutte le operazioni direttamente da codice, SENZA interazione con l'utente.
b. Per quanto riguarda le classi del modello, devono sempre:
    i. ***avere un compito preciso e fare soltanto quello;***
    ii. ***non contenere alcuna lettura/scrittura da console o altra interazione con l'utente;***
    iii. ***lanciare/gestire eccezioni dove appropriato;***
    iv. ***essere dotate di classi di test e/o essere sviluppate secondo la TDD;***
    v. ***essere documentate con Javadoc (almeno per i componenti pubblici).***
c. Cercate sempre di relazionare le classi tra loro ovunque applicabile, facendo uso di **ereditarietà**,
d. **classi astratte** e **interfacce** dove necessario.
e. Cercate sempre di ragionare su quali componenti di una classe dovrebbero essere **pubblici** e quali
f. **privati**/**protetti**, quali **statici** e quali di **istanza**, quali **costanti** e quali no.
g. Cercate di mantenere la **dimensione** dei metodi **contenuta**, spezzandoli ove appropriato, e di rispettare le convenzioni di scrittura codice Java.
h. È sempre fortemente sconsigliato inserire nei progetti elementi fuori dal programma del corso, come database, collegamenti online, etc.

Ricordate sempre che il progetto serve a valutare le vostre competenze in materia di programmazione ad oggetti. Ove le convenzioni e le peculiarità di questo genere di programmazione non saranno usate, il progetto sarà ritenuto insufficiente.

Tutti i progetti **devono avere** almeno:

1. L'estensione di una classe o classe astratta nel modello (sia la superclasse che la sottoclasse devono essere parte della vostra implementazione)
2. Il lancio e la gestione di una eccezione appropriata

3. L'uso appropriato di una classe annidata
4. Il lancio, con la relativa gestione, di eccezioni adeguate

Ogni progetto deve essere accompagnato da una **breve relazione** (2-3 pagine al massimo) che lo descrive. La relazione deve contenere almeno:

a. La lista dei componenti del gruppo
b. Il class diagram delle classi del modello ed una breve descrizione delle classi
c. Una breve descrizione delle funzionalità del programma
d. La seguente tabella compilata (le opzioni fatto/non fatto DEVONO essere tutte "fatto"), potete duplicare le ultime 3 righe se ve ne servono.

Test per le classi del modello	Fatto/non fatto
Javadoc per i componenti pubblici del modello	Fatto/non fatto
Il modello non fa alcuna lettura/scrittura da console o altra interazione con l'utente	Fatto/non fatto
È stata usata l'ereditarietà ovunque appropriato	Fatto/non fatto
È stato riutilizzato il codice ovunque appropriato	Fatto/non fatto
Sono state utilizzate visibilità appropriate	Fatto/non fatto
Sono state lanciate eccezioni ove appropriato	Fatto/non fatto
Sono state usate classi annidate ove appropriato	Fatto/non fatto
Specificare dove/come è stata fatta l'estensione di una classe o classe astratta	
Specificare dove/come è stato usato il lancio/gestione di un'eccezione	
Specificare dove è stata usata una classe annidata	

Infine, per ogni progetto, generate un file JAR eseguibile:

- Generate un file (ad esempio Main.java) con un main che lancia la vostra applicazione.
- Fate click col tasto destro sul vostro progetto, selezionate "Export" e nel menu successivo, come formato di esportazione, cercate "JAR file".
- Nel menu di creazione del JAR file, selezionate tutto il vostro progetto e scegliete dove volete esportare il file JAR come export location.
- Fate click su Next finché non vi chiede la Main Class, e a quel punto scegliere la classe col main creata all'inizio della procedura.
- Ora fate click su Finish: nella export location troverete il file JAR che permette di eseguire il vostro progetto senza aprirlo in Eclipse.

Per la consegna del codice, come spiegato a lezione, potete sfruttare lo stesso menu di esportazione, selezionando "Archive File" invece di "JAR file" al secondo punto.
Mettete il codice, la relazione ed il JAR file in una cartella, compattatela in formato zip o simile, e usate quella per la consegna.

Consegne successive alla prima

Nel caso di riconsegne, nella relazione deve essere presente una sezione che vada a descrivere tutte le modifiche significative applicate.

Valutazione

La discussione del progetto con i singoli studenti verterà sulle modalità di implementazione adottate e sulla padronanza di alcuni dei concetti necessari per realizzare il progetto e/o spiegati a lezione. La valutazione del progetto sarà fatta in base alla: conformità dell'implementazione scelta per risolvere il problema con il paradigma di programmazione a oggetti e alla conformità del codice presentato alle regole di buona programmazione.

Proposta 1 (facile: voto max 26) – PRENOTAZIONE AULE

Progettare e implementare un'applicazione per la gestione delle aule di un'università. In particolare, l'applicazione deve permettere di:

- Inserire, memorizzare e visualizzare **i dati di una serie di aule** (almeno il nome es. "9B", e la capienza)
- Le aule possono essere di **tipi particolari** (es. aula informatica, laboratorio di chimica) oppure no (semplici aule)
- Ogni aula deve avere una serie di **dotazioni** (es. proiettore, lavagna, …). Tipi particolari di aule hanno, di default, determinate dotazioni (es. l'aula informatica avrà sempre il proiettore)
- Sia dotazioni che tipi di aule sono definiti dal sistema: non deve essere possibile avere in dotazione "Topolino". Potete prevederne comunque numeri ristretti (3-4 tipi di dotazioni e 2 tipi di aule, oltre a quelle senza tipo)
- L'applicazione deve permettere di trovare, dati una **data** ed un **orario** (es. 22/12/2021 ore 9-12), e dati una serie di **parametri** (ad esempio, una capienza minima e/o una dotazione), tutte le aule libere in quella data e orario che soddisfano i requisiti richiesti.
  Nota: i requisiti possono anche essere **vuoti**.
- Deve essere possibile **prenotare** un'aula in una data e orario a nome di un **docente**. In tal caso, quell'aula non dovrà più essere prenotabile nella stessa data ed in un orario che si sovrappone anche parzialmente a quello nel quale è stata prenotata.
- L'applicazione deve permettere di gestire (inserire, memorizzare e visualizzare) anche i **docenti**, per i quali basterà memorizzare nome, cognome e matricola (da mantenere **univoca** per tutti i docenti).
- Infine, deve essere possibile selezionare un'aula o un docente, ed estrarre un **report** delle sue **prenotazioni** per una determinata data (es. "report delle prenotazioni di Luca Piovesan il giorno 20/12/2021"). Per aula, il report associerà alle fasce di orario il docente che la occupa. Viceversa, per docente il report associerà alle fasce di orario le aule che occupa. Il report deve essere stampabile **in ordine di fascia di orario**.
  IMPORTANTE: è richiesto che il report venga **memorizzato in un'istanza di una classe**.
- CARICAMENTO/SALVATAGGIO (**facoltativo**): i dati dell'applicazione devono essere salvabili e caricabili da file, ma solo in caso di richiesta dell'utente.
- MVC e GUI: è obbligatorio per tutti che il programma sia strutturato secondo il pattern MVC. La GUI, tuttavia, è **facoltativa** per i gruppi da 1 o 2 membri, ed è **obbligatoria per i gruppi da 3** (almeno per parte del programma: ad esempio la ricerca di un'aula). Nel caso non sviluppiate una GUI, dovete comunque sviluppare una o più classi che facciano il ruolo della view, gestendo l'interazione con l'utente (con stampe/letture da terminale). Ad esempio, tali classi possono **stampare su terminale un menu** e richiedere all'utente di selezionare una voce. È comunque richiesto che anche tali classi di "view non grafica" comunichino con model e controller tramite eventi, come visto a lezione.

Le parti opzionali, qualora siano fatte bene, possono incrementare il voto massimo ottenibile.

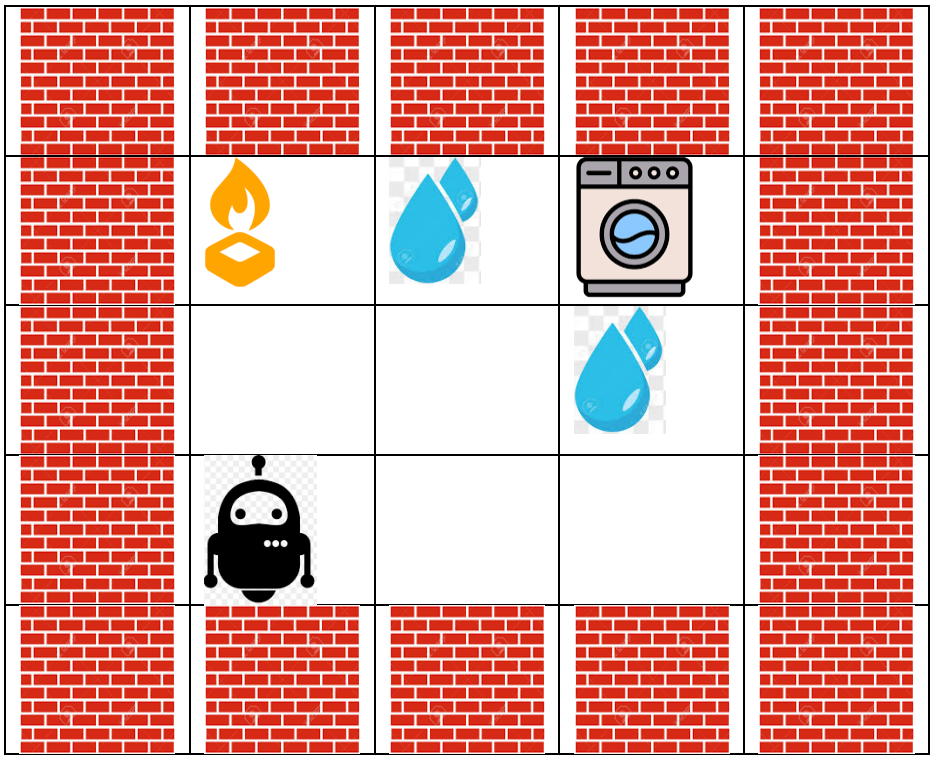
Proposta 2 (un po' meno facile: voto max 32) – MONDO ROBOT

Progettare e implementare un'applicazione per la simulazione di un piccolo robot "manutentore". Prendete come modello un robot come quelli aspirapolvere, che però grazie ad una serie aggiuntiva di sensori ed attuatori, sia in grado di eseguire azioni un po' più complicate.

Il robot si troverà dentro ad una **mappa di una casa**, che per semplicità discretizzeremo in una mappa di caselle nxn, con n (la dimensione) decidibile dall'utente e con la mappa che è generata automaticamente ad ogni "partita". La mappa potrà contenere, oltre al robot, almeno:

- Caselle occupate da **muri**: queste non possono essere attraversate (se il robot cerca di attraversarle, riceve un segnale di "bump") e non è possibile interagire con esse.
- Caselle **vuote**, attraversabili. Trovandosi sopra ad una casella vuota, il robot può sapere se questa è bagnata o no.
- **Fornelli**: caselle non attraversabili. Il robot, trovandosi in una casella adiacente ad un fornello, può richiederne lo stato (acceso/spento) oppure cambiarne lo stato (può solo spegnerlo).
- **Lavatrici** e **Rubinetti**: caselle non attraversabili. Sia lavatrici che rubinetti possono rompersi casualmente durante il gioco, ed iniziare a perdere acqua nelle caselle adiacenti. La perdita di acqua si espande di una casella ogni tot azioni del robot, con le lavatrici che perdono di più ed i rubinetti di meno (ad es. il rubinetto può perdere ogni 15 azioni del robot). Il robot, trovandosi di fianco ad una lavatrice/rubinetto, può chiudere l'acqua ed interrompere la perdita.
- **Animali domestici**: la casella che contiene un animale domestico non è attraversabile (se il robot cerca di attraversarla, riceve un segnale di "bump") e dopo ogni turno del robot l'animale domestico si può muovere casualmente in una casella adiacente o restare in quella corrente.

Un esempio di mappa può essere questo (voi però considerate solo mappe di dimensioni almeno 10x10):



Nel caso in cui implementiate il caricamento da file (vedi sotto), le mappe possono essere definite "a mano" sui file e poi caricate a scelta dell'utente. Altrimenti, va bene creare una mappa casuale ogni volta che il gioco parte (es. tutte le celle del perimetro saranno muri, poi aggiungete qualche oggetto – fornelli, lavatrici, … - a caso nella mappa ed il robot in una posizione casuale).

Il robot deve essere controllabile dall'utente, che ad ogni turno può decidere se:

- Farlo **girare** a destra o a sinistra di 90 gradi
- Farlo **avanzare** di una casella davanti a lui
- Fargli eseguire un'azione di **interazione con gli oggetti**, come quelle elencate sopra (es. spegni un fornello).

Le azioni possono **fallire**, se non eseguite nelle condizioni corrette (es. lo spegnimento di un fornello fallisce se il robot non è di fianco ad esso) e dovete gestire i fallimenti secondo la OOP.

Il robot inizia la partita in una determinata posizione della casa: a quel punto sarà in grado soltanto di avere informazioni sulle caselle ad esso adiacenti e su quella nella quale si trova. Dopo ogni azione, sarà in grado di **raccogliere informazioni** sulle cose ad esso adiacenti.

L'applicazione deve avere due GUI (**implementate con classi diverse**):

- una è quella di gioco, che visualizza le cose man mano che il robot le scopre,
- una è quella di "controllo" (se volete potete nasconderla di default, ma deve essere visualizzabile) che vi fa vedere la posizione e lo stato di TUTTI gli elementi della mappa

PER GRUPPI DA 3 A SCELTA ALMENO UNA DELLE SEGUENTI DUE FUNZIONALITA' (altrimenti sono facoltative):

- Il gioco deve essere salvabile/caricabile su file in modalità "testo in chiaro modificabile" (no serializzazione)
- Le lavatrici/rubinetti non perdono in base al numero di azioni del robot, ma in base allo scorrere del tempo (es. il rubinetto perde ogni 10 secondi)

Importante: GUI

Mi preme sottolineare, anche per non farvi lavorare troppo su parti non importanti, che la "bellezza" delle GUI non è oggetto di valutazione. Mi interessa che "funzionino" e che comunichino correttamente con il modello ed il controller tramite EDP, ma non andrò nel dettaglio della loro estetica, né della loro implementazione.

Importante: uso di DIR

Chi avesse dubbi sul laboratorio può (e deve) **usare il forum su DIR**, controllando prima che non siano già state fatte domande simili. Questo permetterà a tutti di avere le stesse informazioni.