ADEngine

Ability Driven Engine

Progetto di Ingegneria del Software

A.S. 2008 / 2009

Raimondi Dario Andrea
706021

Riassunto

ADEngine è un motore per avventure grafiche[i].Esso permette di rappresentare una storia e gestirne la progressione in seguito alle decisioni intraprese dal giocatore, il quale deve risolvere enigmi e puzzles per giungere alla conclusione della vicenda.

Descrizione

Gli ambienti in cui si svolge la vicenda vengono detti Scene. Il giocatore ha la possibilità di muovere il proprio alter-ego virtuale in ognuna di queste scene, in modo compatibile con eventuali ostacoli in esse rappresentati. Alcuni di questi ostacoli derivano dalla morfologia della Scena stessa (ad esempio, un albero che costringe il giocatore a girarvi intorno); altri invece sono dipendenti dal progresso del giocatore rispetto alla vicenda complessiva (ad esempio, il possesso o meno di una chiave permette l'apertura di una porta, oppure l'aver parlato con il guardiano del faro ci permette di entrarci). Le Scene sono raggruppate in modo da poter accedere ad una di esse attraversandone altre. È anche possibile realizzare una Scena che, a guisa di cartina, renda possibile il rapido movimento tra un luogo ed un altro. Il raggiungimento di particolari posizioni all'interno una Scena causerà il caricamento di un'altra Scena.
Una Scena è composta da diversi strati, insiemi di immagini che verranno disegnati a schermo in ordine. Uno di questi è lo strato percorribile, rappresentato da una griglia in cui i singoli elementi sono detti “tile”. Il giocatore muoverà su di esso. I tiles possono godere di diverse proprietà, di cui alcune a discrezione dell'ideatore della Storia, mentre la proprietà di essere bloccati, ovvero non attraversabili, deve essere comune a tutti. Deve essere possibile interrogare un tile per ottenerne le proprietà. Anche i Personaggi Non Giocanti si muovono su questa griglia, e devono poter essere in grado di muoversi autonomamente da un tile all'altro in modo algoritmico, per esempio con una variante dell'algoritmo A*[ii].

Il giocatore, all'inizio della sessione di gioco, decide di quali Abilità dotarsi. Le Abilità possono essere di qualunque genere, e vengono definite dallo sviluppatore in funzione della Storia. All'interno dell'insieme di tutte le Abilità disponibili, il giocatore sceglie di favorirne alcune e di penalizzarne altre. Per tenere traccia di ciò, ogni Abilità viene valutata secondo questa scala:

- Superbo
- Ottimo
- Buono
- Normale
- Mediocre
- Scarso
- Pietoso

A titolo di esempio, un giocatore potrebbe decidere che il proprio personaggio abbia queste abilità:

- Conoscenza delle varietà di té: Ottimo
- Tolleranza all'alcool: Mediocre
- Eloquenza: Scarsa

Il titolo di questo progetto, ADEngine, significa “Ability Driven Engine”. È un riferimento alla presenza delle Abilità, un concetto che sotto altre forme è stato presente finora solamente nei Giochi di Ruolo[iii]. All'ideatore della storia viene messo a disposizione questo meccanismo di Abilità al fine di permettergli di far evolvere la storia principale, le storie secondarie o anche solo i dialoghi, in funzione delle diverse Abilità che un giocatore ha scelto per sé. Il progresso nella Storia viene quindi guidato dalle Abilità. La lista delle Abilità deve essere accessibile da parte degli altri elementi del gioco. Si assume che le abilità che il giocatore non decida di migliorare o penalizzare abbiano come valutazione “Normale”.

Un giocatore, durante l'avventura, può venire in possesso di diversi oggetti. Li si possono trovare nelle Scene, oppure vengono consegnati al giocatore dai PNG (personaggi non giocanti[iv]). Il giocatore ha accesso agli oggetti che possiede tramite un inventario. Un oggetto può essere esaminato oppure utilizzato con un altro oggetto o con una particolare locazione di una Scena. Ogni Scena quindi possiede una lista di oggetti presenti al suo interno. È compito della Scena provvedere a disegnare a schermo tali oggetti, e a rendere possibile l'interazione del giocatore con essi quando si trovi nel loro immediato intorno. Gli oggetti devono poter restituire una propria descrizione, e reagire ad un'eventuale interazione da parte del giocatore.

In generale, l'interazione del giocatore con un qualsiasi elemento del gioco avviene mediante due modalità astratte: Esame e Interazione vera e propria. L'Esame consiste nel fornire al giocatore una descrizione dell'elemento corrente, mentre l'Interazione assume modalità diverse a seconda del tipo di elemento e delle intenzioni dell'estensore della storia. Tipi di Interazione possono essere “Premi” per un interruttore, “Mangia” per un biscotto, “Raccogli” per una banconota da 10 €.

I conseguimenti di un giocatore, all'interno dello sviluppo della Storia, vengono detti Pietre Miliari. Servono per tenere traccia del progresso del giocatore. Il giocatore può entrare in possesso di una Pietra Miliare in un qualsiasi punto della storia, a discrezione dell'ideatore della stessa. A differenza degli oggetti, il giocatore non deve sapere di quali Pietre Miliari è entrato in possesso, per evitare che giunga a conclusioni sulla storia prima di averla giocata. Il giocatore mantiene una lista di Pietre Miliari, la quale deve poter essere acceduta ed eventualmente modificata da qualsiasi altro elemento nel gioco. È responsabilità dello sviluppatore della Storia organizzare il flusso di Pietre Miliari in modo coerente: ADEngine deve limitarsi a fornire i meccanismi di base.

Un insieme di Pietre Miliari è detto Stato. Questa astrazione serve per modellare il fatto che ci possono essere Pietre Miliari necessarie per il progresso della storia principale, ed altre non necessarie, ma che servono tuttavia per seguire storie secondarie. La storia viene quindi divisa in Stati, ai quali si accede per accumulo di Pietre Miliari, in modo trasparente per il giocatore.

I PNG sono entità dotate della possibilità di muoversi autonomamente all'interno di una Scena, e di reagire in modo diverso all'interazione con il giocatore. Spetta interamente allo sviluppatore della Storia l'implementazione di queste facoltà: l'ADEngine deve limitarsi a fornire gli strumenti per renderlo possibile. Come per gli oggetti, un PNG deve essere in grado di fornire la propria descrizione e la propria reazione all'interazione con il giocatore. In generale si tratterà di interazioni del tipo “Parla con”.

Il motore di gioco implementa un framework composto da due passi fondamentali:

- **update(int delta)**
- **render**

Ad ogni ciclo (che può avvenire anche diverse decine di volte per secondo), ADEngine esegue in successione il passo **update** seguito dal passo **render**. Durante **update** viene fatto eseguire uno step alla logica del gioco, mentre il passo **render** disegna il tutto a schermo. Diverse librerie per la grafica o per lo sviluppo di videogiochi rendono disponibile, magari con nomi diversi, questo tipo di framework.
L'avventura è divisa in Stati. Ogni Stato deve rendere disponibile il metodo **update(int delta)** ed il metodo **render()**, i quali verranno chiamati ad ogni ciclo del gioco. In ogni istante deve esserci uno ed un solo Stato attivo, che si registrerà presso il framework e quindi intercetterà i due eventi suddetti. Lo Stato ha la responsabilità di chiedere al framework di attivare un altro Stato.
Lo Stato attivo a sua volta ha nozione della Scena attiva in quell'istante. Le chiamate **update(int delta)** e **render()** vengono quindi inoltrate alla Scena attiva, e da lì vengono inoltrate ai PNG contenuti nella Scena.

Il metodo **update(int delta)** accetta come parametro un intero che rappresenta il numero di millisecondi trascorsi dall'ultima chiamata di **update()**, in altre parole il delta del tempo trascorso. Questo delta viene utilizzato per integrare in modo semplice la posizione o qualsiasi altro evento del gioco. Ad esempio, se un PNG si muove con una velocità di 100 pixel al secondo, è possibile rappresentare questo valore con 100 / 1000 pixel al millisecondo. Ad ogni iterazione, dato il delta e la velocità vettoriale, è pertanto possibile ottenere la successiva posizione con $x = x + delta * velocità$ in modo indipendente dalla frequenza di aggiornamento. L'applicazione di questo principio a tutti gli aggiornamenti incrementali del gioco permette di ottenere indipendenza dal framerate e quindi la stessa velocità di esecuzione del programma su macchine diverse, oltre che gestire i casi in cui, sulla stessa macchina, il diverso carico sul sistema operativo causi un'accelerazione o un rallentamento della frequenza di aggiornamento.
L'estensione dell'interfaccia basata su **update(int delta)** e **render()** come interfaccia minima per tutti gli elementi del gioco garantisce uniformità di implementazione nell'aggiornamento della logica di gioco. Inoltre, deve essere possibile mettere in pausa l'aggiornamento del gioco, ad esempio per permettere l'esecuzione di cut-scene o di dialoghi tra il giocatore ed un PNG.

I movimenti del giocatore e l'interazione col mondo possono avvenire con la tastiera o con il mouse. È compito di un modulo separato, unico per tutto l'ADEngine, intercettare, tramite l'opportuno riferimento a librerie di basso livello, i segnali provenienti dalle periferiche di input e inoltrarli allo Stato attivo. Se, ad esempio, la pressione del tasto “freccia sinistra” rappresenta l'intenzione di muovere il giocatore a sinistra, il modulo di gestione dell'input passerà quest'informazione allo Stato attivo, il quale a sua volta farà riferimento alla Scena per valutare la possibilità di muovere effettivamente il giocatore a sinistra di un certo step. Ciò dipende dalla presenza di altri PNG nella direzione desiderata, oppure di un tile bloccato che non è possibile attraversare. In caso di risposta positiva, verrà aggiornata la posizione del giocatore all'interno della scena, ed al successivo render il giocatore sarà visualizzato in modo corretto.

Tutti gli oggetti di ADEngine devono essere dotati di uno Script realizzato in un linguaggio interpretato (ad esempio JavaScript), opportunamente integrato con il motore del gioco. Tramite l'associazione di uno Script ad un PNG, per esempio, è possibile realizzare uno scambio di battute che tenga conto delle Abilità del giocatore. Tutti gli elementi che nel gioco sono suscettibili di evolversi nel tempo devono rendere disponibile l'interfaccia minima illustrata poco sopra. Gli Stati stessi vanno implementati come Script. In altre parole è possibile vedere ADEngine come una collezione di Script, che il framework si preoccupa di chiamare in modo opportuno.
Le motivazioni per questa scelta sono le seguenti:

1. trattandosi di codice scritto in un linguaggio di programmazione interpretato, è possibile modificare uno script senza dover ricompilare il framework, con notevole risparmio di tempo;
2. i linguaggi di scripting hanno in genere una sintassi più semplice rispetto ai linguaggi tradizionali come C o Java (ad esempio, mancano della tipizzazione delle variabili), e ciò permette anche ad un non-programmatore di poter sviluppare una Storia rapidamente.

Pertanto, il framework deve rendere disponibili tutti i metodi necessari per accedere alle proprietà del giocatore (Inventario, Pietre Miliari, Abilità) e a tutti gli elementi del gioco (Tiles, Scene, PNG).

L'interfaccia grafica di ADEngine deve permettere le seguenti cose:

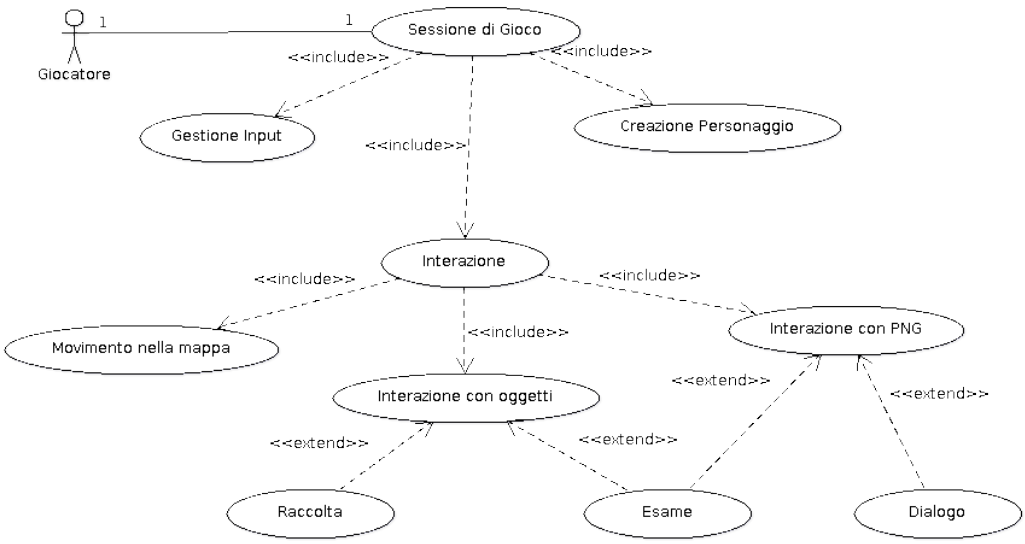
- visualizzazione del giocatore nella Scena;
- accesso all'inventario con possibilità di esaminare o selezionare ogni oggetto in esso presente;
- accesso alla lista delle proprie Abilità;
- visualizzazione di comunicazioni all'utente.

Quest'ultimo punto comprende i seguenti tipi di comunicazione:

1. visualizzazione della descrizione di un oggetto o di un PNG;
2. visualizzazione dell'esito dell'interazione con un oggetto;
3. visualizzazione di uno scambio di battute con un PNG, con eventuale scelta multipla tra le battute da pronunciare;
4. visualizzazione di altri messaggi funzionali alla Storia.

Il framework deve quindi rendere disponibile la visualizzazione di un messaggio di testo. Si può richiedere ad ADEngine di visualizzare tale messaggio in un punto preciso dello schermo, oppure delegare questa responsabilità al motore stesso, il quale sceglierà la posizione più appropriata in funzione degli elementi presenti sulla scena. Il testo deve venir automaticamente formattato e visualizzato, se necessario, in finestre scorrevoli. È possibile istruire ADEngine a far sì che un messaggio rimanga a schermo fino alla pressione di un tasto da parte dell'utente, oppure che scompaia allo scadere di un certo tempo, in modo da rendere possibili dialoghi automatici. ADEngine deve essere in grado di calcolare il tempo minimo in cui un messaggio deve rimanere a schermo, in funzione della lunghezza in caratteri del messaggio stesso.

Diagramma dei Casi d'Uso



Caso d'Uso:Sessione di gioco

Descrizione: Gestione dell'intera sessione di gioco, a partire dalla creazione del personaggio.

Attori: Giocatore

Successo:

1. Il giocatore crea un personaggio
2. Il giocatore gioca la storia
3. Il giocatore termina la sessione di gioco

Caso d'Uso: Gestione Input

Descrizione: Gestione dell'input proveniente da tastiera o mouse, durante l'intera sessione di gioco

Attori: Giocatore

Successo:

1. Il giocatore immette un segnale di input
2. Il segnale viene comunicato all'ADEngine
3. Gli effetti dell'Input si ripercuotono sul mondo virtuale di ADEngine

Caso d'Uso: Creazione Personaggio

Descrizione: Il giocatore sceglie, tra una lista di Abilità, quali favorire e quali inibire, attribuendo loro un punteggio

Attori: Giocatore

Precondizione: il giocatore non deve avere ancora iniziato a giocare

Successo: il giocatore seleziona un certo numero di Abilità cui attribuire una valutazione più alta, e altre cui assegnare una valutazione più bassa

Caso d'Uso: Interazione

Descrizione: Il giocatore decide quale azione eseguire

Precondizione: il giocatore deve aver creato un personaggio

Successo:

1. Il giocatore sceglie un'azione
2. L'azione viene valutata
3. Le conseguenze dell'azione si ripercuotono sul mondo virtuale di ADEngine

Caso d'Uso: Movimento nella mappa

Descrizione: Il giocatore sceglie di eseguire un'azione di movimento

Successo:

1. Il giocatore sceglie un movimento
2. Il movimento viene valutato
3. Lo spostamento del giocatore si ripercuote sul mondo virtuale di ADEngine

Caso d'Uso: Interazione con oggetti

Descrizione: Il giocatore sceglie di eseguire un'azione di interazione con un oggetto

Successo:

1. Il giocatore decide di interagire con un oggetto
2. L'interazione viene valutata dall'oggetto
3. Le conseguenze dell'interazione si ripercuotono sul mondo virtuale di ADEngine

Caso d'Uso: Interazione con personaggi non giocanti

Descrizione: Il giocatore sceglie di eseguire un'azione di interazione con un personaggio non giocante

Successo:

1. Il giocatore decide di interagire con un PNG
2. L'interazione viene valutata dal PNG
3. Le conseguenze dell'interazione si ripercuotono sul mondo virtuale di ADEngine

Caso d'Uso: Raccolta

Descrizione: il giocatore decide di raccogliere un oggetto

Precondizioni:

1. l'oggetto deve essere raccoglibile
2. l'oggetto deve essere a portata del giocatore

Successo: l'oggetto scompare dalla scena e viene inserito nell'inventario del giocatore

Caso d'Uso: Esame

Descrizione: il giocatore decide di esaminare un oggetto o un PNG

Precondizione: il PNG, o l'oggetto, deve essere a portata del giocatore

Successo: la descrizione dell'oggetto o del PNG viene resa disponibile al giocatore

Caso d'Uso: Dialogo

Descrizione: il giocatore decide di interloquire con un PNG

Precondizione: il PNG deve essere a portata del giocatore

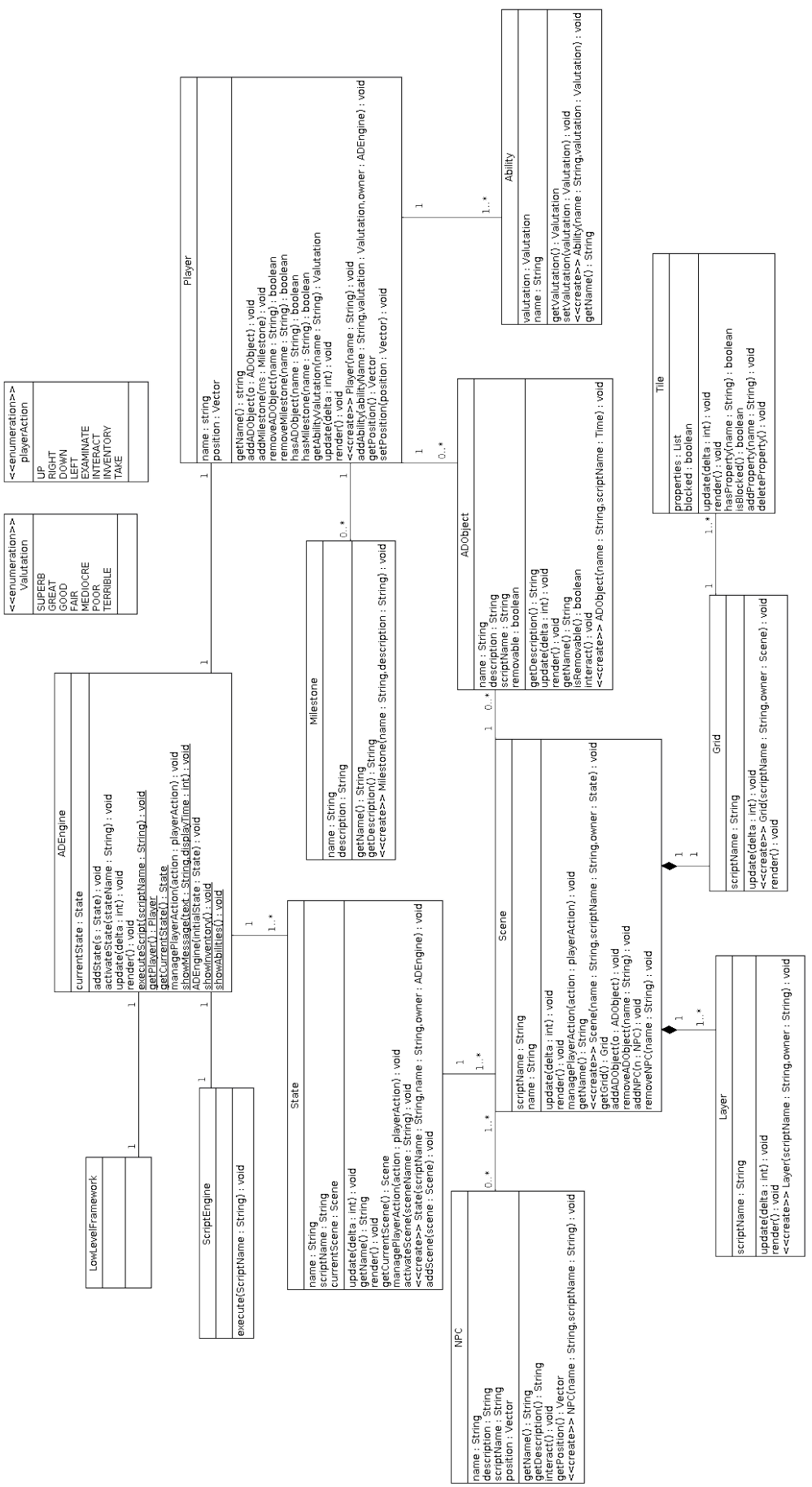
Successo:

1. Il giocatore sceglie quale frase pronunciare
2. Il PNG risponde

I passi 1 e 2 possono venir ripetuti più volte

3. Il dialogo termina

Diagramma delle Classi



Implementazione delle Classi

Dalla lettura ed analisi dei requisiti è stato possibile ottenere la suddivisione delle funzionalità di ADEngine in un certo numero di classi, delle quali si può avere una vista d'insieme nel diagramma alla pagina precedente.

Per ogni classe viene riportata la raccolta dei requisiti ad essa relativi e un'implementazione iniziale della classe, generata automaticamente da ArgoUML e poi corretta manualmente.

Le Classi individuate sono le seguenti:

- ADEngine
- ScriptEngine
- Player
- Ability
- ADObject
- State
- Scene
- Layer
- Grid
- Tile
- NPC

Le **Milestones**, più volte citate nei requisiti, non hanno necessità di una classe che le implementi, in quanto si tratta essenzialmente di una proprietà del **Player**. Il requisito "*Uno Stato è un insieme di Milestones*" è pertanto da considerarsi come un'astrazione del progresso dell'avventura, e come tale non va modellato esplicitamente.

I requisiti che il **LowLevelFramework** deve rispettare si limitano al fatto che esso deve in qualche modo rispettare il pattern **Update-Render** tipico di un'applicazione videoludica. In questo modo è possibile slegare l'ADEngine da una particolare libreria specifica per videogames.

ADEngine

Raccolta dei requisiti

- L'ADEngine aggiorna la logica del gioco
- L'ADEngine viene disegnato a schermo
- L'ADEngine reagisce alle azioni di input
- L'ADEngine visualizza l'inventario del Player
- L'ADEngine visualizza le Ability del Player
- L'ADEngine visualizza messaggi a schermo
- L'ADEngine gestisce l'input
- L'ADEngine inoltra update() e render() allo State attivo e al Player
- L'ADEngine ha un motore di Scripting
- L'ADEngine ha una lista di State
- L'ADEngine ha un Player

Implementazione

```
import java.util.Vector;

public class ADEngine {

  private State currentState;
  private Player myPlayer;
  private Map<String, State>  statesList;
  private LowLevelFramework myLowLevelFramework;
  private ScriptEngine myScriptEngine;
  private static ADEngine mainEngine;

  /**
   *  Aggiunge uno State
   */
  public void addState(State s) {
      statesList.put(s.getName(), s);
  }

  /**
   *  Attiva uno State
   */
  public void activateState(String stateName) {
      if (statesList.containsKey(stateName) {
            currentState = statesList.get(stateName);
      }
  }

  public void update(int delta) {
  }

  public void render() {
  }

  /**
   *  Fa eseguire allo ScriptEngine uno script.
   */
  public static void executeScript(String scriptName) {
      myScriptEngine.execute(scriptName);
  }

  /**
   *  Restituisce un riferimento al Player.
   */
  public static Player getPlayer() {
      return mainEngine.getMyPlayer();
  }

  public Player getMyPlayer() {
      return myPlayer;
  }

  /**
   *  Restituisce un riferimento allo State corrente.
   */
  public static State getCurrentState() {
      return mainEngine.getMyCurrentState;
  }

  public State getMyCurrentState() {
      return currentState;
  }

  /**
   *  Gestisce l'input proveniente dal framework di basso livello.
   */
  public void managePlayerAction(playerAction action) {
```

```
  }

  /**
   *  Visualizza un messaggio di testo a schermo.
   */
  public static void showMessage(String text, int displayTime) {
  }

  /**
   *  Costruttore di ADEngine. initialState è lo State iniziale.
   */
  public void ADEngine(State initialState) {
      statesList.put(initialState.getName(), initialState);
      ADEngine.mainEngine = this;
  }

  /**
   *  Mostra a schermo l'inventario del Player
   */
  public static void showInventory() {
  }

  /**
   *  Mostra a schermo le Ability del Player
   */
  public static void showAbilities() {
  }
}
```

Note di implementazione

Le associazioni con molteplicità pari ad 1, ovvero quelle con **LowLevelFramework**, **ScriptEngine** e **Player** sono state realizzate con variabili private.

L'associazione con State è stata implementata con un Vector<State>. Il metodo **void addState(State s)** aggiunge uno State a tale lista. Lo State corrente viene memorizzato nella variabile privata **currentState**, in modo da renderlo facilmente e velocemente accessibile sia internamente che esternamente ad **ADEngine**.

I metodi void **executeScript(String scriptName)**, **Player getPlayer()**, **State getCurrentState**, **void showInventory()**, **void showAbilities()**, **void showMessage(String text, int displayTime)** sono dichiarati **static** per facilitarne l'accesso, essendo utilizzati in diversi punti del software e anche da parte degli scripts. L'istanza di ADEngine viene salvata nel membro privato statico mainEngine. I metodi **getPlayer()** e **getCurrentState()** sono quindi wrapper statici attorno a **getMyPlayer()** e **getMyCurrentState()**, e sono realizzati sfruttando **mainEngine**.

ScriptEngine

Raccolta dei requisiti

- Lo ScriptEngine esegue uno Script

Implementazione

```
public class ScriptEngine {

  /**
   *  Esegue uno script contenuto in un file.
```

```
   */
  public void execute(String ScriptName) {
  }

  public ScriptEngine() {
  }
}
```

Note di implementazione

Lo ScriptEngine ha lo scopo di istanziare un motore di scripting e di interfacciarlo con il resto del progetto ADEngine. Nei requisiti non si specifica quale linguaggio di Script debba essere implementato. L'unico requisito che si può inferire, relativo ad una implementazione di un linguaggio di Scripting, è che sia in grado di esportare la classe ADEngine all'interno dell'ambiente di Scripting: la maggior parte dei sistemi di Scripting in ambiente Java è in grado di realizzare ciò in modo estremamente semplice. È inoltre da notare che l''interfaccia tra sistema di Scripting e codice Java dipende fortemente dall'implementazione del linguaggio di Script che è stata scelta. Esiste una **Java Scripting API**[v] che fornisce meccanismi standard per collegare un'ambiente di Scripting ad un'applicazione Java, ma vista l'esistenza di numerosi ambienti che non la sfruttano, non sembra il caso di doverne forzare l'adozione.

Player

Raccolta dei requisiti

- Il Player si muove nella scena
- Il Player si muove nella Griglia
- Il Player raccoglie ADObject dalla Scena
- Il Player viene disegnato a schermo
- Il Player interagisce con un ADObject (esame / interazione)
- Il Player interagisce con un PNG (esame / interazione)
- Il Player riceve delle Pietre Miliari (da uno Stato o da un PNG)
- Il Player ha delle Pietre Miliari, interrogabili
- Il Player ha una posizione
- Il Player ha un inventario di ADObject
- Il Player è dotato di Abilità, interrogabili

Implementazione

```
import java.util.Vector;
import java.util.Map;

public class Player {
   /**
    *  Nome del giocatore
    */
  private string name;

  /**
    *  Posizione nella scena corrente
    */
  private Vector position;

    /**
    *
    * @element-type Ability
    */
```

```
  public Map<String, Ability> abilitiesList;

  private ADEngine myADEngine;

    /**
   *
   * @element-type ADObject
   */
  private Map<String, ADObject>  ADObjectsList;
    /**
   *
   * @element-type Milestone
   */
  private Map<String, Milestone>  milestonesList;

  /**
   *  Ritorna il nome del giocatore
   */
  public string getName() {
      return name;
  }

  /**
   *  Aggiunge un ADObject all'inventario del giocatore
   */
  public void addADObject(ADObject o) {
      ADObjectsList.put(o.getName(), o);
  }

  /**
   *  Aggiunge una Milestone all'inventario del giocatore
   */
  public void addMilestone(Milestone ms) {
      milestonesList.put(ms.getName(), ms);
  }

  /**
   *  Rimuove un ADObject dall'inventario del giocatore
   */
  public boolean removeADObject(String name) {
      ADObjectsList.remove(name);
  }

  /**
   *  Rimuove una Milestone dall'inventario del giocatore
   */
  public boolean removeMilestone(String name) {
      milestonesList.remove(name);
  }

  /**
   *  Verifica se il giocatore è in possesso di un ADObject, dato il nome
dell'ADObject
   */
  public boolean hasADObject(String name) {
      return (ADObjectsList.containsKey(name));
  }

  /**
   *  Verifica se il giocatore è in possesso di una Milestone, dato il nome
   */
  public boolean hasMilestone(String name) {
      return (milestonesList.containsKey(name));
  }

  /**
```

```
   *  Ritorna la valutazione di una Ability
   */
  public Valutation getAbilityValutation(String name) {
      return abilitiesList.get(name).getValutation();
  }

  /**
   *  Aggiorna la logica del giocatore (frame di animazione)
   */
  public void update(int delta) {
  }

  /**
   *  Renderizza il giocatore a schermo
   */
  public void render() {
  }

  /**
   *  Costruttore di Player.
   */
  public Player(String name, ADEngine owner) {
      this.name = name;
      myADEngine = owner;
  }

  /**
   *  Aggiunge un'Ability al giocatore, con la sua valutazione
   */
  public void addAbility(String abilityName, Valutation valutation) {
      Ability ability = new Ability(abilityName, valutation);
      abilitiesList.put(abilityName, ability);
  }

  /**
   *  Ritorna la posizione del giocatore nella Scena corrente
   */
  public Vector getPosition() {
      return position;
  }

  /**
   *  Imposta la posizione del giocatore
   */
  public void setPosition(Vector position) {
      this.position = position;
  }
}
```

Note di implementazione

L'associazione di molteplicità 1..1 con **ADEngine** è realizzata tramite una variabile privata. Le associazioni con **Milestone**, **ADObject** e **Ability** sono invece realizzate tramite **Map**. Esistono quindi metodi per modificare tali liste, come specificato nei requisiti. Tali metodi sono:

- **void addADObject(ADObject o)**
- **void addMilestone(Milestone ms)**
- **boolean removeADObject(String name)**
- **boolean removeMilestone(String name)**
- **boolean hasADObject(String name)**

- **boolean hasMilestone(String name)**
- **Valutation getAbilityValutation(String name)**
- **void addAbility(String abilityName, Valutation valutation)**

La realizzazione tramite **java.util.Map** indicizzate per nome rende semplice, da parte delle altre classi e degli scripts, accedere allo stato del **Player** mediante il nome degli oggetti.

Ability

Raccolta dei requisiti

- Le Ability hanno un nome

Implementazione

```
public class Ability {

  /**
   *  Valutazione dell'Ability
   */
  private Valutation valutation;

  /**
   *  Nome dell'Ability
   */
  private String name;

  /**
   *  Ritorna la valutazione dell'Ability
   */
  public Valutation getValutation() {
      return valutation;
  }

  /**
   *  Imposta la valutazione dell'ability
   */
  public void setValutation(Valutation valutation) {
      this.valutation = valutation;
  }

  /**
   *  Costruttore di Ability
   */
  public Ability(String name, Valutation valutation) {
      this.name = name;
      this.valutation = valutation;
  }

  /**
   *  Ritorna il nome dell'Ability
   */
  public String getName() {
      return name;
  }
}
```

Note di implementazione

L'associazione con Player non ha bisogno di essere implementata dal lato di **Ability**, in quanto l'**Ability** è un oggetto meramente passivo.

State

Raccolta dei requisiti

- Lo State consegna Pietre Miliari al Player
- Lo State aggiorna la propria logica
- Lo State viene disegnato a schermo
- Lo State gestisce l'input
- Lo State inoltra l'input alla Scene
- Lo State inoltra le chiamate del framework di basso livello alla Scene
- Lo State ha uno Script
- Lo State è composto da Scene

Implementazione

```
import java.util.Vector;

public class State {

  private String name;

  private String scriptName;

  private Scene currentScene;

  private ADEngine myADEngine;

    /**
   *
   * @element-type Scene
   */
  private Map<String, Scene) scenesList;

  public void update(int delta) {
  }

  /**
   *  Restituisce il nome dello State
   */
  public String getName() {
      return name;
  }

  public void render() {
  }

  /**
   *  Ritorna un riferimento alla Scene corrente
   */
  public Scene getCurrentScene() {
      return currentScene;
  }

  /**
   *  Gestisce l'azione del giocatore
   */
  public void managePlayerAction(playerAction action) {
  }
```

```
  /**
   *  Attiva una Scene
   */
  public void activateScene(String sceneName) {
      currentScene = scenesList.get(sceneName);
  }

  public State(String scriptName, String name, ADEngine owner) {
      myADEngine = owner;
      this.name = name;
      this.scriptName = scriptName;
  }

  /**
   *  Aggiunge una Scene
   */
  public void addScene(Scene scene) {
      scenesList.put(scene.getName(), scene);
  }
}
```

Note di implementazione

L'associazione con **ADEngine** è realizzata tramite una variabile privata. La lista delle Scenes è implementata tramite una **java.util.Map** indicizzata per nome, in modo da poter attivare una Scene in modo semplice, da parte degli scripts. Il metodo void **activateScene(String sceneName)** permette appunto questo.

Scene

Raccolta dei requisiti

- La Scene viene disegnata a schermo
- La Scene aggiorna la propria logica
- La Scene permette di passare ad un'altra Scene
- La Scene ha uno Script
- La Scene è composta da diversi Layer
- La Scene contiene ADObject (eventualmente)

Implementazione

```
import java.util.Vector;

public class Scene {

  private String scriptName;

  private String name;

  private State myState;

    /**
   *
   * @element-type Layer
   */
  private Vector<Layer> layersList;

  private Grid myGrid;

    /**
```

```
    *
    * @element-type ADObject
    */
   private Map<String, ADObject> ADObjectsList;

     /**
    *
    * @element-type NPC
    */
   private Map<String, NPC> NPCList;

   public void update(int delta) {
   }

   public void render() {
   }

   /**
    *  Gestisce l'input del giocatore
    */
   public void managePlayerAction(playerAction action) {
   }

   public String getName() {
       return name;
   }

   /**
    *  Ritorna la Grid di questa Scene
    */
   public Grid getGrid() {
       return myGrid;
   }

   public void addADObject(ADObject o) {
       ADObjectsList.put(o.getName(), o);
   }

   public void removeADObject(String name) {
       ADObjectsList.remove(name);
   }

   public void addNPC(NPC n) {
       NPCList.put(n.getName(), n);
   }

   public void removeNPC(String name) {
       NPCList.remove(name);
   }

   public Scene(String name, String scriptName, State owner) {
       this.name = name;
       this.scriptName = scriptName;
       this.myState = owner;
   }
}
```

Note di implementazione

Il requisito *"La Scene permette di passare ad un'altra Scene"* viene realizzato tramite il riferimento allo State che ha in carico la Scene attuale, tramite il metodo **public void activateScene(String sceneName)**.

La Scene è composta da una serie di Layers e da una Grid. La creazione di tali Layers e della Grid è

affidata allo Script che viene passato come argomento al costruttore della Scene. Questo accorgimento serve per dividere il più possibile l'implementazione dell'Engine dall'implementazione del gioco basato su di esso. Permette infatti di costruire un gioco completamente diverso semplicemente mutando gli Script, senza bisogno di alcuna ricompilazione del progetto.

Layer

Raccolta dei requisiti

- Il Layer viene disegnato a schermo
- Il Layer aggiorna la propria logica
- Il Layer ha uno Script

Implementazione

```
public class Layer {

  private String scriptName;

  private Scene myScene;

  public void update(int delta) {
  }

  public void render() {
  }

  public Layer(String scriptName, Scene owner) {
      this.scriptName = scriptName;
      this.myScene = owner;
  }
}
```

Note di implementazione

Il Layer serve per raggruppare le funzionalità comuni ad una parte passiva della Scena. In particolare, deve essere in grado di disegnare autonomamente una parte del fondale o della zona di primo piano di una Scene.

Grid

Raccolta dei requisiti

- La Grid viene disegnata a schermo
- La Gridaggiorna la propria logica
- La Grid ha uno Script
- La Grid è composta da Tiles

Implementazione

```
import java.util.Vector;

public class Grid extends Layer{
```

```
  private String scriptName;

  private Scene myScene;

   /**
    *
    * @element-type Tile
    */
  private Vector<Tile>  tilesList;

  public void update(int delta) {
  }

  public Grid(String scriptName, Scene owner) {
      this.scriptName = scriptName;
      this.myScene = owner;
  }
}
```

Note di implementazione

La **Grid** deve essere in grado di disegnarsi a schermo in modo passivo, allo stesso modo del **Layer**, ma a differenza di quest'ultimo mantiene una lista di Tiles che vengono utilizzati per il movimento e per il posizionamento degli ADObject nella Scene.

NPC

Raccolta dei requisiti

- L'NPC si muove nella Scena
- L'NPC si muove nella Griglia
- L'NPC consegna ADObject al Player
- L'NPC consegna Pietre Miliari al Player
- L'NPC aggiorna la propria logica
- L'NPC viene disegnato a schermo
- L'NPC ha un nome
- L'NPC ha una descrizione
- L'NPC ha una posizione
- L'NPC ha uno Script

Implementazione

```
import java.util.Vector;

public class NPC {

  private String name;

  private String description;

  private String scriptName;

  private Scene myScene;

  private Vector position;
```

```
  /**
   *  Restituisce il nome dell'NPC
   */
  public String getName() {
      return name;
  }

  /**
   *  Ritorna la descrizione dell'NPC
   */
  public String getDescription() {
      return description;
  }

  /**
   *  Esegue l'interazione relativa a questo NPC
   */
  public void interact() {
  }

  /**
   *  Ritorna la posizione dell'NPC
   */
  public Vector getPosition() {
      return position;
  }

  public NPC(String name, String scriptName) {
      this.name = name;
      this.scriptName = scriptName;
  }
}
```

Note di implementazione

I requisiti *"L'NPC si muove nella Scena"* e *"L'NPC si muove nella Griglia"* vengono implementati tramite la conoscenza della **Scene** cui l'NPC appartiene. Il requisito *"L'NPC si muove con A*"*, invece, può essere implementato in una libreria di Script accessibile in modo globale.

ADObject

Raccolta dei requisiti

- L'ADObject viene disegnato a schermo
- L'ADObject ha un nome
- L'ADObject ha una descrizione

Implementazione

```
public class ADObject {

  private String name;

  private String description;

  private String scriptName;

  private boolean removable;


  /**
```

```
   *  Ritorna la descrizione dell'ADObject
   */
  public String getDescription() {
      return description;
  }

  public void update(int delta) {
  }

  public void render() {
  }

  /**
   *  Ritorna il nome dell'ADObject
   */
  public String getName() {
      return name;
  }

  /**
   *  Verifica se l'ADObject è removibile
   */
  public boolean isRemovable() {
      return removable;
  }

  /**
   *  Esegue l'interazione per questo ADObject
   */
  public void interact() {
  }

  public ADObject(String name, Time scriptName) {
      this.name = name;
      this.scriptName = scriptName;
  }
}
```

Note di implementazione

L'ADObject non deve essere a conoscenza della **Scene** o del **Player** cui appartiene, in quanto questo non ha utilità.

Lo Script il cui nome viene passato come argomento al costruttore provvederà a riempire i membri privati **String description** e **boolean removable**.

Tile

Raccolta dei requisiti

- Il Tile è interrogabile relativamente alle sue proprietà
- Il Tile viene disegnato a schermo
- Il Tile è bloccato o no
- Il Tile ha una lista di proprietà

Implementazione

```
import java.util.List;

public class Tile {
```

```
  private java.util.List<String> properties;

  private boolean blocked;

  private Grid myGrid;

  public void update(int delta) {
  }

  public void render() {
  }

  /**
   *  Verifica se il Tile gode di una certa proprietà
   */
  public boolean hasProperty(String name) {
      return properties.contains(name);
  }

  /**
   *  Verifica se il Tile è percorribile
   */
  public boolean isBlocked() {
      return blocked;
  }

  /**
   *  Imposta la proprietà Blocked del Tile
   */
  public void setBlocked(boolean b) {
      blocked = b;
  }

  /**
   *  Aggiunge una proprietà al Tile
   */
  public void addProperty(String name) {
      properties.add(name);
  }

  public void deleteProperty(String name) {
      properties.remove(name);
  }

  public Tile(Grid owner) {
      this.myGrid = owner;
  }
}
```

Note di implementazione

Le proprietà del **Tile** sono rappresentate come un Vector di String. È possibile aggiungere un numero arbitrario di proprietà, e cancellarle a piacimento. L'unica proprietà di cui deve essere garantita la presenza è **Blocked**. Trattandosi della più importante, in quanto fondamento della possibilità di movimento attraverso una **Grid**, è previsto un membro privato appositamente per essa

Enumerazioni

Sono state individuate due Enumerazioni: **Valutation**, che esprime le valutazioni delle **Abilities**, e **PlayerAction**, che modella quello che il giocatore può fare in termini di interazione con il gioco. In

modo particolare **PlayerAction** permette di astrarre in modo efficace la rilevazione dell'input di basso livello rispetto ai suoi effetti nel gioco.

Valutation

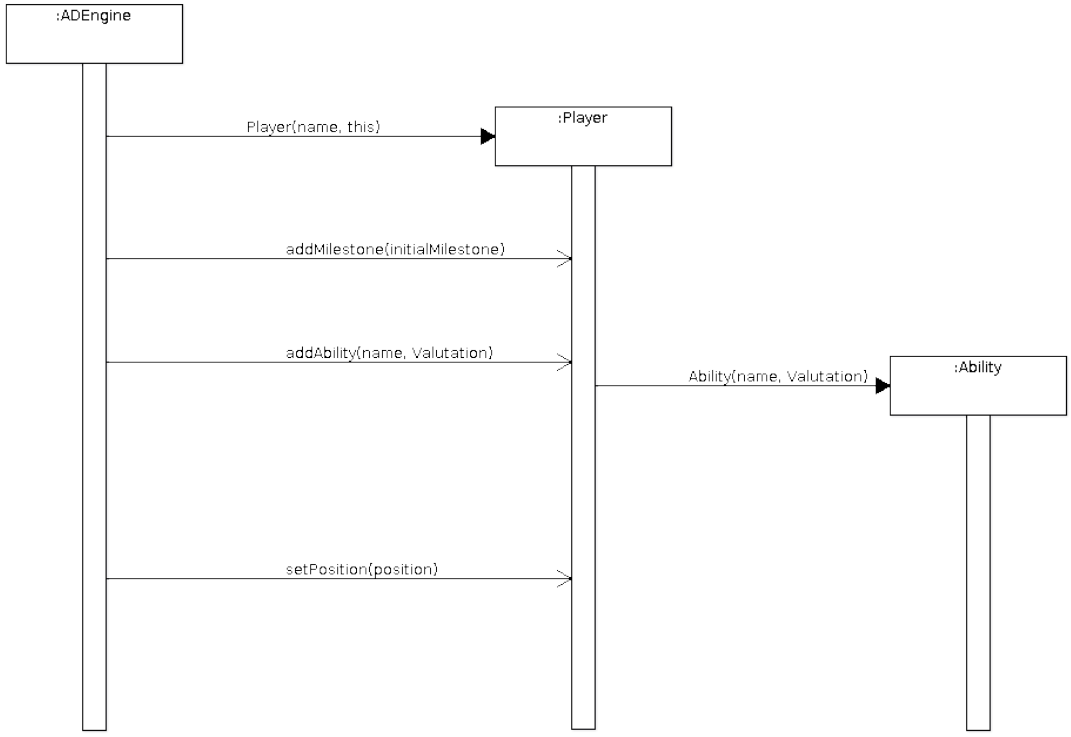
```
public enum Valutation {
      SUPERB, GREAT, GOOD, FAIR, MEDIOCRE, POOR, TERRIBLE
}
```

PlayerAction

```
public enum PlayerAction {
      UP, RIGHT, DOWN, LEFT, EXAMINATE, INTERACT, INVENTORY, TAKE
 }
```

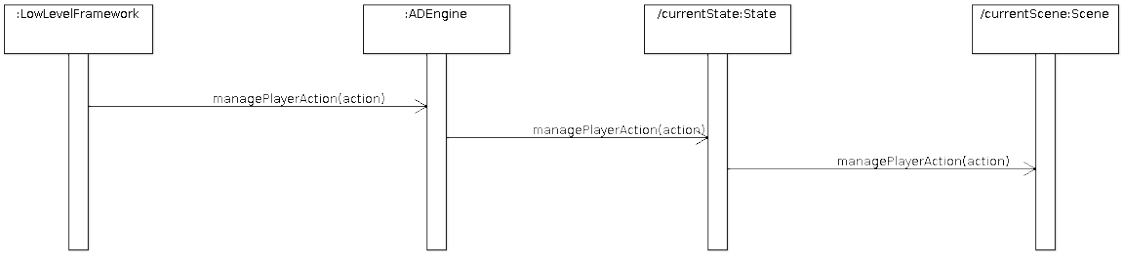
Diagrammi di sequenza

Creazione Personaggio



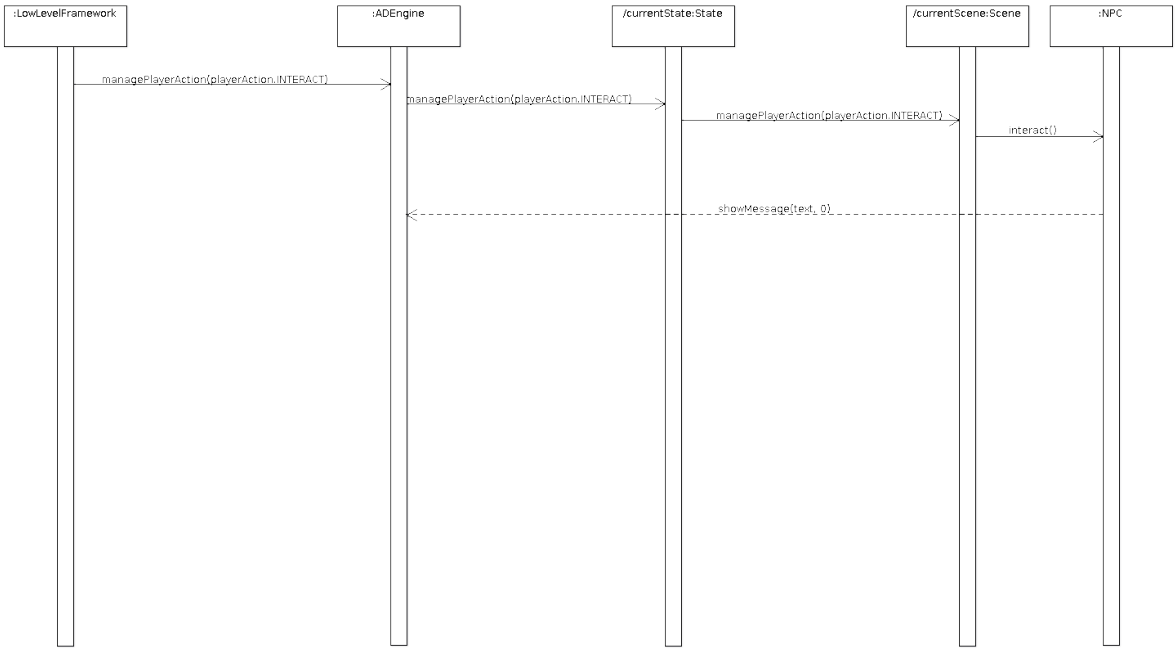
La costruzione del **Player** prende avvio dall'**ADEngine**. Anche le **Ability** vengono aggiunte al **Player** dall'**ADEngine**.

Gestione Input



La **Gestione Input** è un primo esempio della centralizzazione attorno ad **ADEngine** sulla quale verte l'intero progetto. Come sarà visibile anche nei prossimi casi d'uso, l'iniziativa del giocatore viene raccolta dal LowLevelFramework in modalità che, per esigenze di astrazione e di separazione tra meccaniche di gioco e gestione dell'hardware, non riguardano direttamente l'ADEngine.

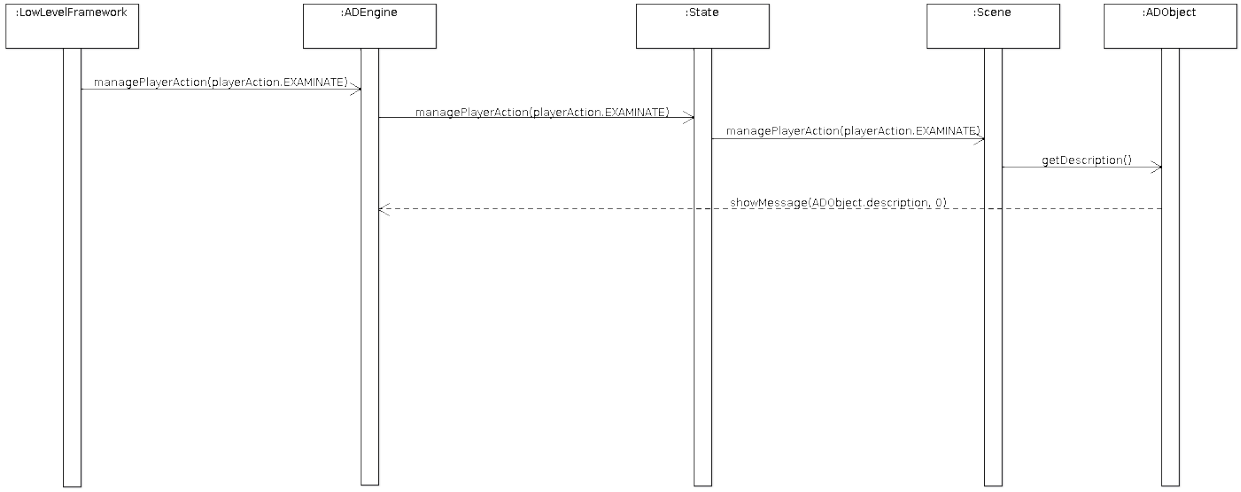
Dialogo



Lo schema già visto in **Gestione Input** qui viene completato. La presenza dei metodi statici richiamabili da qualsiasi punto della gerarchia delle classi permette da un lato di non passare come argomenti ai vari costruttori un eccessivo numero di riferimento ai vari “owner” delle classi, e dall'altro semplifica il ricorso a routine di uso comune.

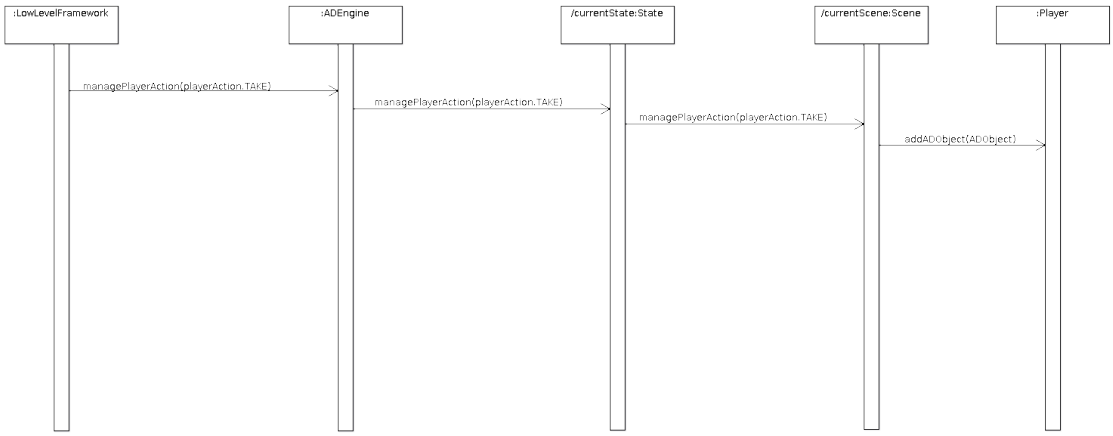
Va inoltre aggiunto che le funzionalità presentate con wrapper statici in **ADEngine** avrebbero potuto far parte di una classe esterna con la funzione appunto di implementare una libreria condivisa. L'esiguo numero di questi metodi, tuttavia, rende superflua una scelta del genere.

Esame



Il caso d'uso **Esame** è un altro esempio della suddivisione delle responsabilità e dell'utilizzo dei metodi statici. In questo grafico si descrive l'esame di un **ADObject**, ma la stessa meccanica viene applicata all'esame di un **NPC**.

Raccolta Oggetti



In questo caso d'uso viene visualizzato un esempio di interazione tra una **Scene** ed il **Player**. Ciò che viene qui esemplificato con una chiamata diretta a **Player** nel codice Java sarebbe in realtà implementato in questo modo:

```
ADEngine.getPlayer().addADObject(obj);
```

Il riferimento a **Player** potrebbe anche essere salvato all'interno di ogni **Scene**, per esempio durante

l'esecuzione del costruttore. È una scelta possibile, che tuttavia non è stata contemplata esplicitamente durante l'esecuzione del seguente progetto per due motivi. Il primo è che non avrebbe introdotto una grande semplificazione nell'accesso al Player. Il secondo è che, dal momento che la maggior parte della logica del gioco sarà implementata mediante Scripts, il riferimento al Player potrebbe essere salvato con maggiore comodità proprio all'interno dell'ambiente di Scripting, come libera scelta da parte di chi deciderà di utilizzare ADEngine per implementare il gioco. Nell'ottica di occuparsi del motore e non delle sue applicazioni, ho deciso di utilizzare una struttura di questo tipo.

Questo ragionamento può essere esteso anche a diversi aspetti dell'ADEngine. In generale, l'idea è che i metodi statici permettano di accedere facilmente alle classi “interessanti” tramite il codice Java, per quanto possano far storcere i nasi a qualche purista della programmazione ad oggetti. Inoltre, è quasi sicuro che nell'ambiente di Script verranno salvati in variabili apposite tutti i riferimenti allo State corrente, alla Scene corrente, alla Grid, al Player e così via, per facilitare ulteriormente l'estensore dei vari Script. Questa certezza deriva da esperienze maturate sul campo, nell'integrazione di Script all'interno di progetti di vaste dimensioni. La scelta di strutturare in questo modo ADEngine mi sembra un buon compromesso tra facilità d'uso e pulizia del codice.

Lo stesso tipo di interazione con il **Player** avviene anche relativamente alla gestione delle **Milestones** e delle **Abilities**, attraverso una rappresentazione sequenziale praticamente identica.

i http://it.wikipedia.org/wiki/Avventura_Grafica
ii http://en.wikipedia.org/wiki/A*_search_algorithm
iii http://it.wikipedia.org/wiki/Gioco_di_ruolo
iv http://it.wikipedia.org/wiki/Personaggio_non_giocante
v http://java.sun.com/javase/6/docs/technotes/guides/scripting/programmer_guide/index.html