Lezione 13 08/05/2025

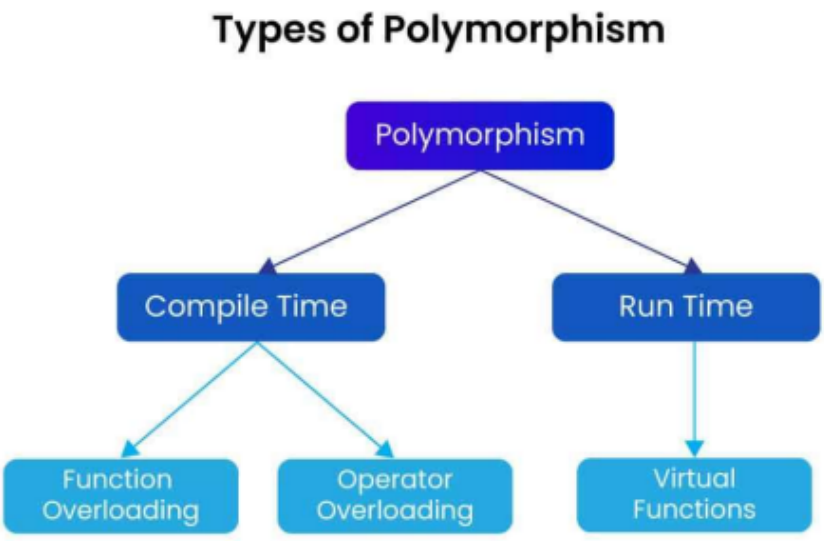
SW Engineering

Buona parte dei componenti del game engine andrebbero implementati da zero. Tutti questi moduli dovrebbero interagire tra loro. Questo fa si che scrivere del codice che sia gestibile e mantenibile in questo progetto enorme è di fondamentale importanza.

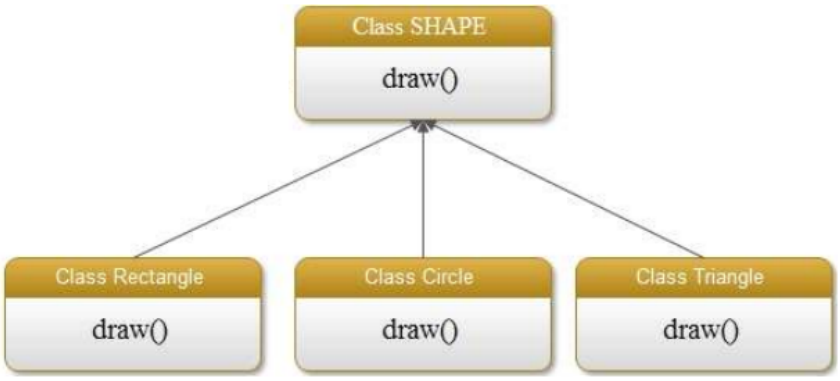
Oggi vediamo alcuni pattern di programmazione.

Il **poliformismo** è una tecnica che permette di modificare il comportamento delle classi andando a definire delle interfacce comuni in una classe padre, con le figlie che vanno a specializzarle.

Quando si fa **overloading di una funzione** significa che definiamo una stessa funzione ma con diversi parametri di input, avendo quindi delle varianti.



Il polimorfismo più interessante è quello attivato a **runtime**, che coinvolge le classi, con una classe padre che dichiara un metodo, non lo implementa, e delega le classi figlie ad implementarlo con gli stessi identici parametri, in modo specifico per la loro natura.



L’idea è che vogliamo usare tutte le istanze di classe figlia non tanto come vere classi figlie, ma come cast alla classe padre. Questo fa si che tutte le figlie possano essere usate con la stessa interfaccia, senza sapere se è un rettangolo, cerchio, triangolo.

Quindi si chiama il metodo dalla classe padre, che chiama il metodo della classe figlia in base da chi è stato creato il cast.

La classe padre avrà quindi **metodi virtuali**, che possono essere definiti ma sovrascritti, oppure neanche definiti. Nel secondo caso si parla di un metodo virtuale puro: è solo dichiarato, non è implementato. Questo rende la classe **astratta**. Cioè non si può istanziare perchè manca un pezzo di codice.

In questo caso si chiama di **override** (non overload) delle funzioni, perchè la stessa identica funzione viene ridefinita in una classe figlia.

Si parla di polimorfismo a runtime perchè la logica con cui si sceglie che metodo eseguire, se decide a runtime.

RAII Pattern

Resource Acquisition Is Initialization.

Se un qualche oggetto, entità, classe, ha bisogno di accedere o generare risorse, allora quella classe è responsabile di chiudere (rimuovere) la risorsa che ha chiesto. Ovvero se ho un oggetto che ha bisogno di allocare memoria, allora la classe allora la memoria ma deve anche de-allocarla. L’oggetto stesso si autogestisce la risorse.

◆ Pros

- Facilitate memory management
- No memory leaks
- When instance no more needed -> destroy and automatic release of the resource

◆ Cons

- None

Si preferisce questo ad un garbage collector perchè il garbage collector non è controllabile, ottimizzabile.

Game Loop Pattern

è un ciclo, è quel ciclo che viene eseguito all'infinito e gestisce gli elementi che fanno parte dell'engine.

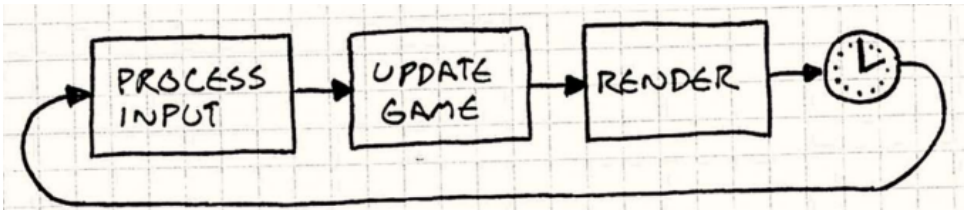
Ogni ciclo del loop, per un singolo frame:

- gestisce l'input dell'utente senza bloccare
- aggiorna lo stato del gioco
- esegue il rendering del gioco

Questo loop viene eseguito alla velocità massima consentita dal sistema. Questo però vuole dire che sistemi diversi fanno girare il gioco a velocità diverse. Questo significa che diventa impossibile giocare a giochi multi-utente.

Come possiamo a forzare una velocità fissa?

- Definire un **famerate target** (in millisecondi per frame)
- Misurare il tempo trascorso per completare le tre operazioni principali:
  - elaborazione input utente
  - aggiornamento dello stato del gioco
  - rendering del frame
- Attendere il tempo rimanente per raggiungere il limite del framerate target



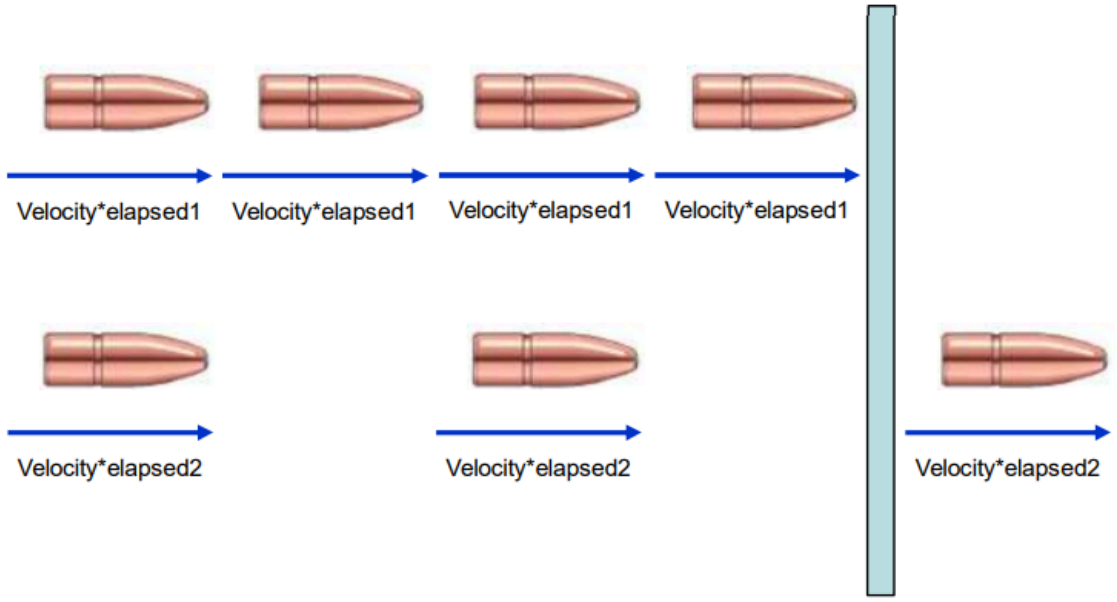
Questi funziona bene se il tempo target è maggiore del tempo di elaborazione effettivo.

Però non è la soluzione ideale, perchè per un tot di tempo il gioco è fermo.

Invece che aggiornare in modo fisso al mondo, posso dire al motore fisico di aggiornarlo di una quantità in base a quanto ci ha messo il loop precedente. Quindi gli dico quanto tempo deve recuperare per ri-allinearsi al tempo reale esterno. Quindi ci sono degli update variabili.

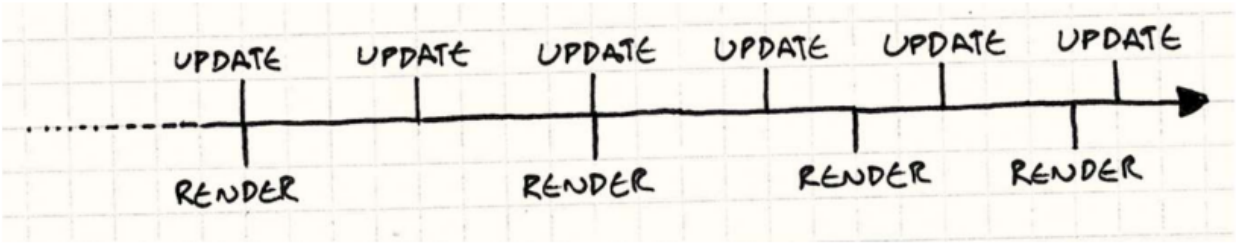
- Pros
  - The game plays at a consistent rate on different hardware.
  - Players with faster machines are rewarded with smoother gameplay.
- Cons
  - The game became non-deterministic and unstable
  - The Physic and AI engine need update at fixed time steps
    - The simulations work this way!
  - Simulations on different systems give different results!
    - Different time steps => different errors

Se il delta time è troppo alto però, ci possono essere degli errori, come delle mancate collisioni.



Una possibile soluzione è quella di:

- Aggiornare lo stato a passi fissi
- Renderizzare il gioco a passi variabili

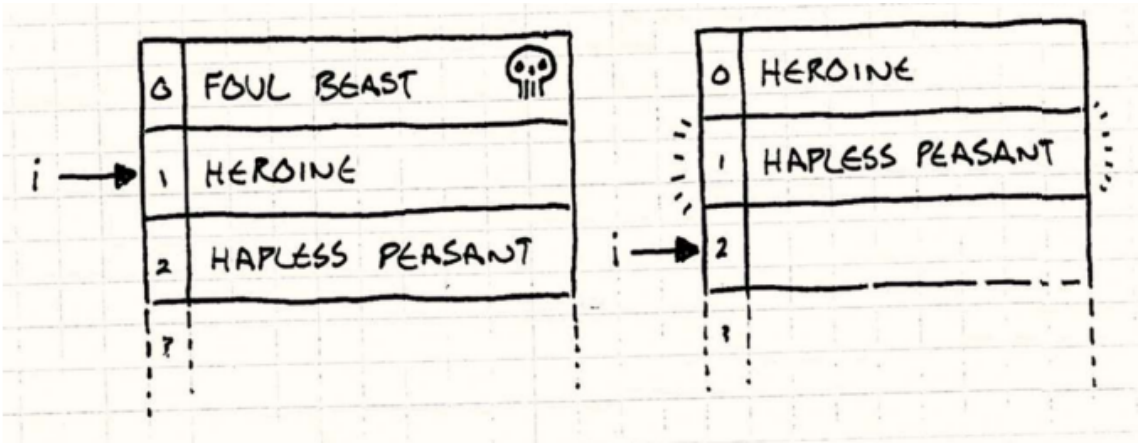


Update Method Pattern

Gli oggetti, entità, classi... hanno un metodo update per poterli aggiornare in qualche modo, in un istante di gioco. Quindi nella fase di update del game loop, il

game engine spammerà ciascun oggetto dicendo di aggiornarsi.

Una cosa a cui prestare attenzione (non solo in questo pattern) è il quando queste entità vanno rimosse dalla lista di oggetti. Queste rimozioni vanno fatte in ordine inverso, per non lasciare buchi nella lista.



Component Pattern

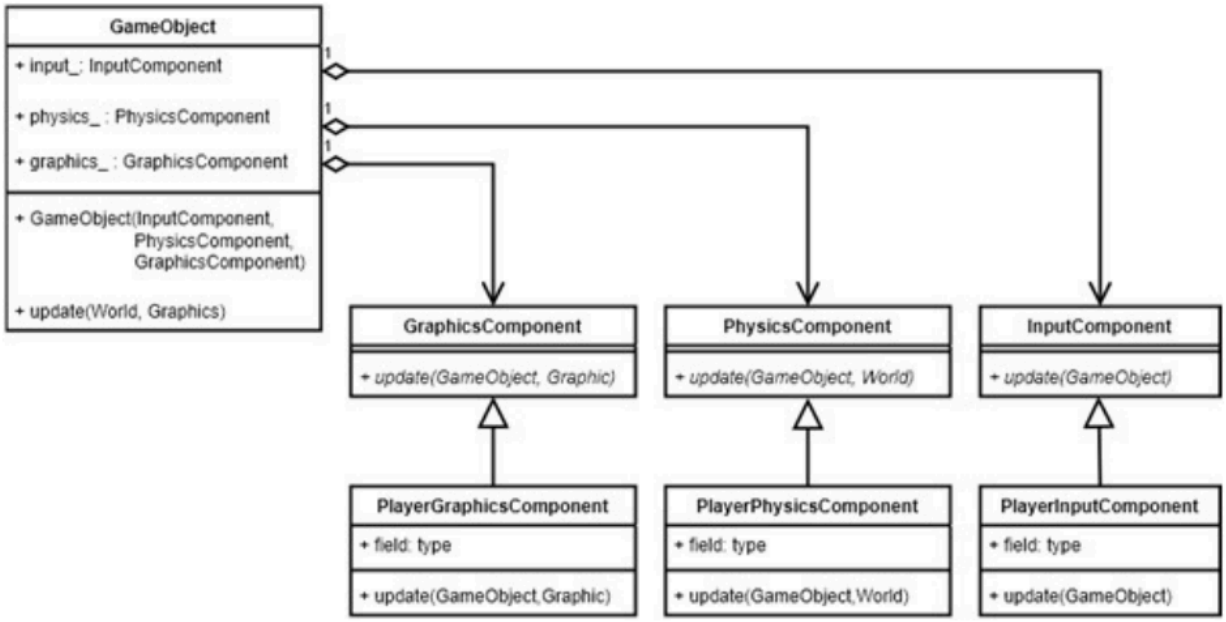
Se lavoriamo con un linguaggio a classi/oggetti, e dobbiamo modellare le varie entità del programma, spesso iniziano a proliferare classi, gerarchie, sottoclassi... questa gerarchia di classi, se è troppo estesa, può rischiare di diventare poco gestibile perchè c'è troppa dipendenza.

Si preferisce estendere le funzionalità tramite composizione.

Se ho un entità nel gioco, che deve modificare il suo stato rispetto ad una certa direttiva del motore fisico, è inutile che prendo una classe entità, eredito dalla classe fisica delle funzionalità, eredito dalla classe rendering altre funzionalità, e altre dalla classe AI... questo crea un accoppiamento molto alto sui dati. Spesso è più comodo gestire queste funzionalità incapsulandole in oggetti separati. Quindi al posto di diventare l'oggetto parte del rendering, andrà a chiedere di usare delle parti del motore fisico.

Questo significa che nell'oggetto, inserisco tanti oggetti quanti ne ho bisogno per gestire le parti del gioco. Quindi non ho una gerarchia di derivazione tra le classi, invece l'oggetto usa altri oggetti.

Avendo questa struttura di classi, potrei anche sostituire un componente con un altro senza dover rompere la gerarchia, ma semplicemente sostituendo l'oggetto.



Double Buffer Pattern

Il double buffer è quel meccanismo che ci consente di creare l'immagine renderizzata in una scena, in un'area di memoria, e nel frattempo che viene generata l'immagine, l'immagine precedente viene mostrata a schermo.

Quindi abbiamo due aree di memoria:

- una usata solo in lettura, presentata a schermo
- una usata solo in scrittura perchè deve essere aggiornato lo stato del fotogramma

Questo va usato quando abbiamo una risorsa usata contemporaneamente in lettura e scrittura in diverse entità. Se si lascia l'accesso libero, ci possono essere problemi di coerenza temporale perchè diversi dati possono essere letti, se qualcuno li sta scrivendo contemporaneamente.

Quindi con questo sistema, quando l'aggiornamento è completato, scambio i due buffer.

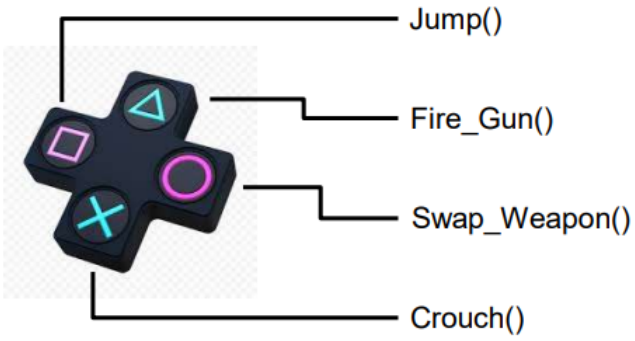
Questo è utile per l'engine fisico o quello AI. Gli attori devono poter interagire l'un l'altro e dovrebbero apparire come se si stessero aggiornando contemporaneamente. Quindi l'aggiornamento è coerente, indipendentemente dal

timestamp dell'aggiornamento, questo svincola anche dall'ordine di aggiornamento degli oggetti perchè tutti vedono lo stato originale.

Command Pattern

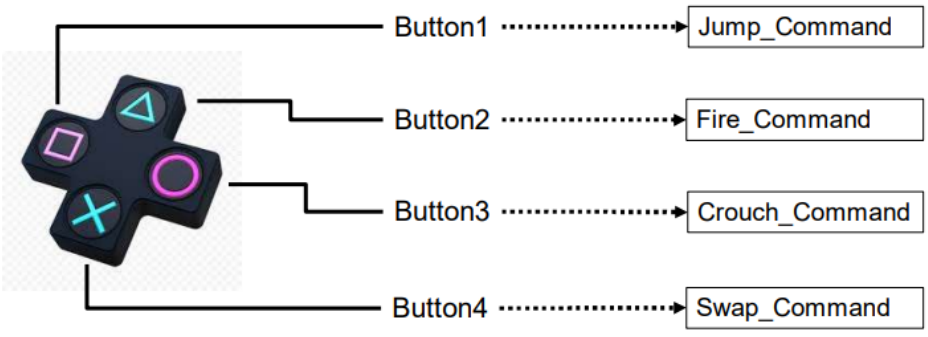
Permette di trasformare gli eventi in oggetti autocontenuti, che sono in grado di gestirsi, contenendo tutte le informazioni necessarie per reagire all'evento stesso.

Per esempio, se dobbiamo gestire questi tasti:

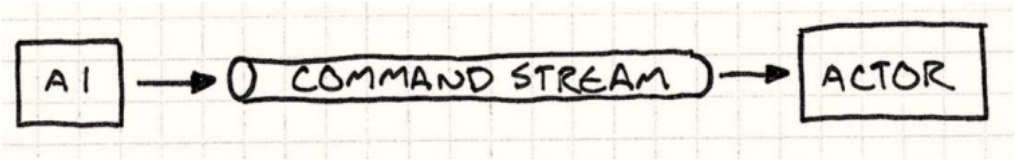


Se voglio cambiare le impostazioni di cosa fanno i tasti, non posso fare un collegamento diretto tra comando e azione.

Tipicamente si usano oggetti intermedi che vengono associati agli input, che attivano dei comandi effettivi. Così posso mantenere l'interfaccia comune e sostituire l'azione.



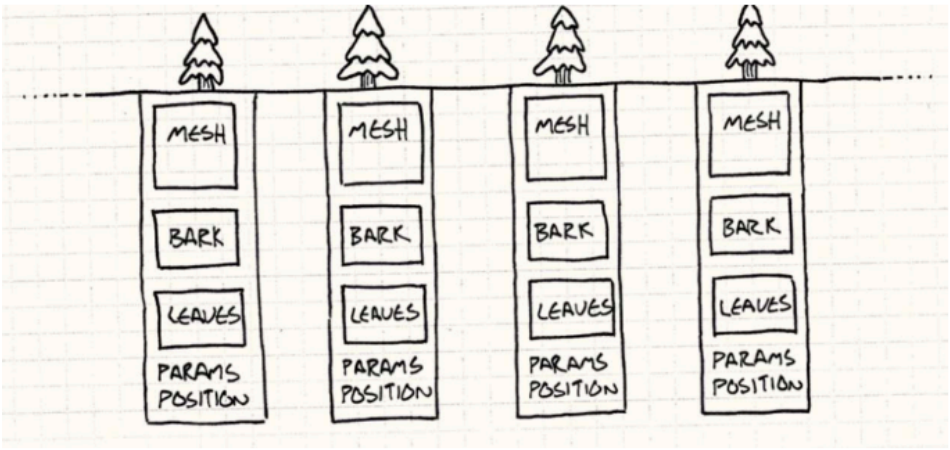
Questo è utile anche quando voglio serializzare una serie di azioni.



Così posso anche mandarli in rete, al server (per esempio per inviare il comando di movimento ad un altro utente).

Flyweight Pattern

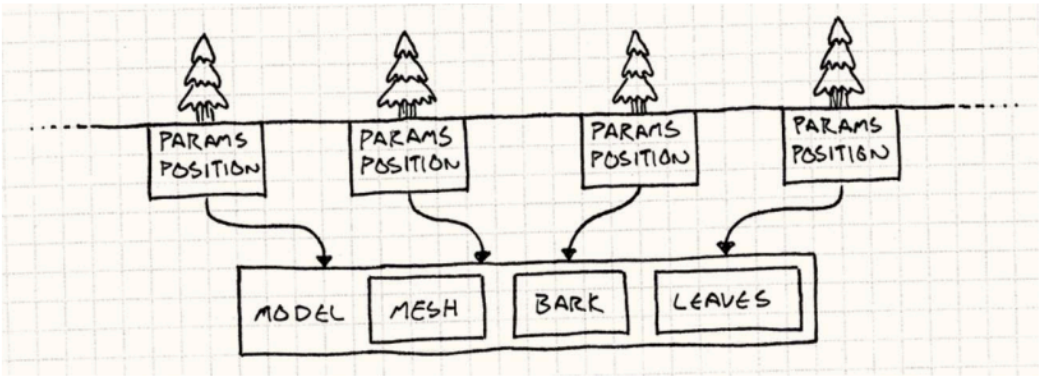
Qual è il problema principale nei giochi? La memoria, la gestione delle risorse. Immaginiamo di avere un gioco dove il campo di gioco è una griglia discreta dove ogni pezzo della griglia può essere roccia, terra... quindi ad ogni quadratino dobbiamo associare delle proprietà. Esempio 2D:

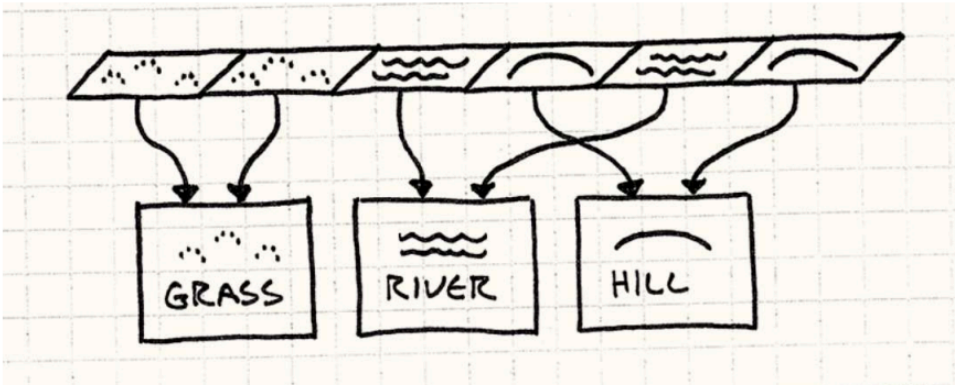


ciascun albero ha delle informazioni associate. Molte istanze condividono delle informazioni. L'unica cosa che cambia magari è la posizione nel mondo.

Includere tutte queste informazioni è uno spreco di risorse.

Questo pattern dice di unire le informazioni comuni in un oggetto a parte:





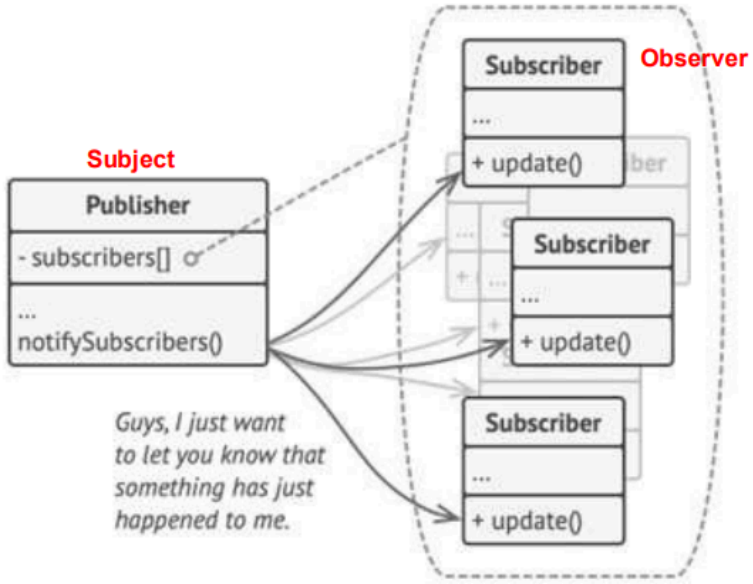
Observer Pattern

Nei giochi abbiamo altre situazioni dove ci sono oggetti che interagiscono tra di loro.

Immaginiamo di avere un sistema di achievement, dove deve comparire il pop up dell’achievement.

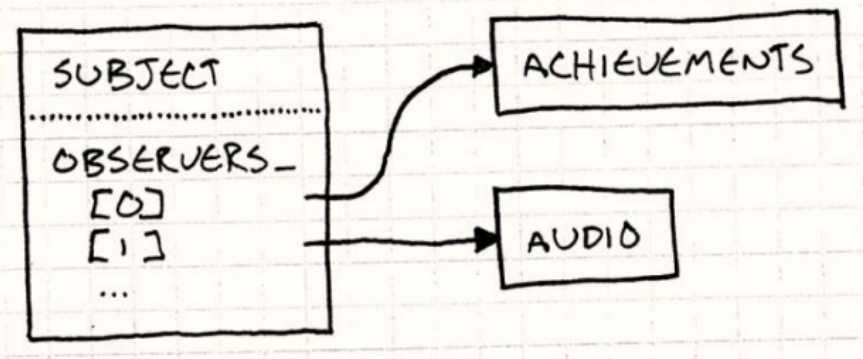
Dove metto la logica?

Le regole che consentano di identificate l’obiettivo raggiunto dovrebbero essere nascoste da chi deve visualizzarlo a schermo. L’idea è quella di avere un’entità (in questo caso il modulo responsabile di gestire tutti gli achievement) che viene costruita come classe Observer, che sta in ascolto perchè qualcosa succeda, pronta a reagire nel modo opportuno. Poi c’è l’entità publisher che si occupa di capire se è successo qualcosa.



Quindi il modulo di combattimento dice “potrei generare degli achievement”, quindi si interfaccia con un publisher dicendo “guarda potrei generare degli achievement, gestiscili”, quindi possiamo avere diversi oggetti dispersi nel gioco che potrebbero generare un achievement, tramite un meccanismo di subscribing al publisher.

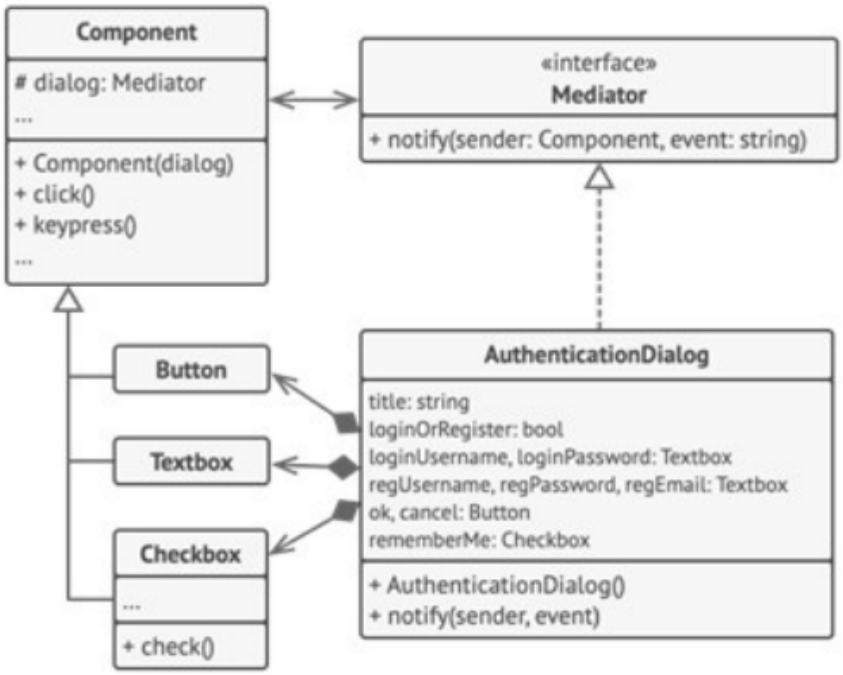
L’observer è qualcosa che gestisce degli eventi. L’achievement è un observer, è una classe specializzata per gestire gli achievement. Avrà della logica per esempio “è stato ucciso un nemico, e questo evento è legato al fatto che è stato usato un pollo di gomma? allora abilito l’achievement.”.



Quindi abbiamo un’entità che può generare degli eventi, e una che li può gestire.

Mediator Pattern

Abbiamo delle entità che non si vogliono o non si devono parlare direttamente, e lo fanno attraverso un mediatore.

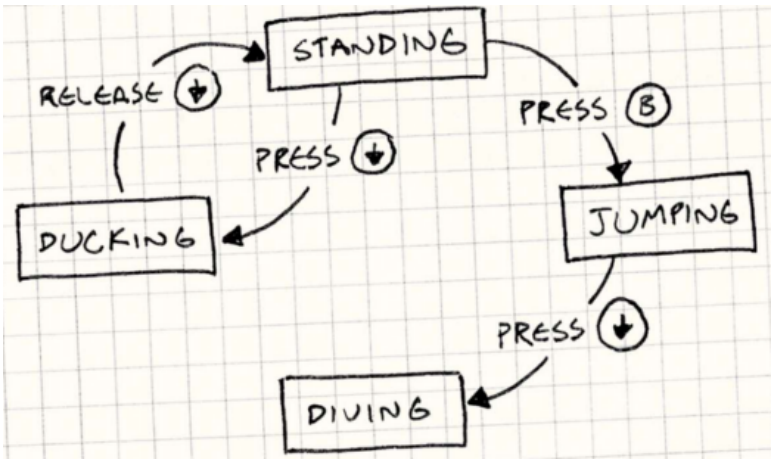


Qui per esempio il mediator è il bottone del form, che applica dei cambiamenti in base a quanto inserito nei campi, non sono i campi che direttamente cambiano qualcosa.

State Pattern

Ci permette di modellare l'automa a stati finiti.

L'idea è supponiamo di avere un personaggio che nel suo stato normale è in piedi...



L’idea di questo pattern è che posso prendere ciascuno stato come un’oggetto a se stante, in grado di autogestirsi. Quindi quando si passa ad un nuovo stato, si crea un nuovo oggetto che farà quello che deve fare, e gestirà gli eventi per cambiare stato.

Quindi in pratica ciascuno stato è una classe a parte che gestisce l’azione. Ho una classe base di stato e poi definisco gli stati.

Service Locator Pattern

Se abbiamo un oggetto che rappresenta un cane, e abbiamo bisogno che questo oggetto inizi ad abbaiare, allora bisogna per esempio recuperare l’audio e la libreria che permette di riprodurlo. Come vengono gestite?

Il service locator ha composto da:

- una classe service che definisce un’interfaccia astratta con delle operazioni comuni
- una classe figlia concreta che implementa specificatamente quell’interfaccia (per esempio se devo renderizzare una scena usando OpenGL o Vulkan, questa saranno due classi figlie concrete, la classe padre sarà un renderer generico).
- il vero e proprio service locator, ovvero una classe terza che sa che ci sono dei service provider e lì da a chi li chiede.