In se per se il progetto è già realizzato, dato il fatto che la creazione, movimento, eliminazione degli oggetti è già esistente.  
La richiesta quindi ricade sul rendere i comandi scritti interpretabili.

i comandi sono:  
 **new**: classico instanziamento di un nuovo oggetto non collegato ad altro, non necessita dell’undo visto che l’undo naturale è la delete, alla creazione ad ogni oggetto viene allegato il proprio ID unico;

**delete**: classica rimozione di un oggetto esistente, implica infatti che l’oggetto debba esistere precedentemente, rimuove o il singolo oggetto o la completezza del gruppo a cui appartiene. Vengono identificati da un ID quindi in base all’ID viene rimosso un oggetto oppure un gruppo;

**move**: non necessitano dell’undo, visto che il loro UNDO naturale è loro stesso. Unico problema è ricordare la posizione da cui provenivano, quindi, per comodità verso l’utente avrebbe senso inserire un undo che ricordi la vecchia posizione;  
 Ne esistono due versioni:  
 **mv**: sposta l’oggetto identificato dall’id nella posizione scelta, in base all’id sposta un oggetto o un gruppo;  
 **mvoff**: uguale al precedente ma invece che spostare seccamente nella posizione inserita somma la posizione inserita alla posizione attuale;

**scale**: ridimensione l’oggetto con il determinato id ( o il gruppo) del valore insierito, per la stessa logica di comodità attuata nei confronti della move va inserito l’undo

**ls**: non necessita dell’undo dato che legge e non modifica nulla;  
 **ls** **id:** trova l’oggetto con il determinato id e ne restituisce le proprietà a riga di comando;  
 **ls** **oggetto**: restituisce la lista del tipo di oggetti scelti;  
 **ls** **all**: restituisce la lista di tutti gli oggetti;  
 **ls** **groups**: restituisce le lista di tutti i gruppi esistenti;

**problema:** va stampata solo la lista oppure la lista con la descrizione di ogni singolo oggetto? A livello codice non sarebbe un problema andando a richiamare i toString figli, però a livello lettura magari sarebbe orrendo e lento come la merda

**grp**: posso creare un nuovo gruppo attraverso gli ID, POSSO CREARE GRUPPI SOLO DI OGGETTI O ANCHE GRUPPI DI GRUPPI E ANCHE GRUPPI MISTI (OGGETTI E GRUPPI) non necessitano undo, in quano ungrp è l’undo naturale;

**ungrp**: il contrario di grp, “esplode” il gruppo identifica dall’id inserito, ogni oggetto non in un ulteriore gruppo torna ad essere senza gruppo, ogni gruppo torna ad essere un gruppo solitario, forse necessita undo per comodità, un gruppo che contiene molti elementi ed eliminato per sbaglio è problematico da ricreare con grp;

**calcoli**: non necessitano undo, POSSONO ESSERE APPLICATI SU TUTTI I TIPI DI FORME TRANNE LE IMMAGINI  
 **area**: autoesplicativo;  
 **perimetro**: autoesplicativo;  
  
  
Memento Pattern

che è il più ovvio, se non, praticamente richiesto dalla traccia   
il memento si occupa di memorizzare lo stato precedente (o gli stati) in modo tale da poterli ritirare fuori alla necessità.  
Memento si ricollega al prossimo design Pattern.

Command Pattern

Comodissimo perché prima di fare qualcosa memorizza lo stato precedente al suo intervento e il successivo, quindi è sostanzialmente possibile usarlo quasi al posto del memento.  
Comodo anche perché dato che dobbiamo dare comandi a riga, possiamo sdoppiare i comandi, in modo che la prima cosa che faccia sia leggere e la cosa subito dopo sia invocare i metodi magari già esistenti (come le creazioni) con i determinati canoni scelti  
(da aggiungere alla creazione un nuovo costruttore che costruisce nella posizione scelta)

Factory Pattern

Per quanto si vede dal progetto su cui dobbiamo basarci già presente, quindi la metto per questo.  
ovviamente il factory è ormai un metodo standard, quindi presente in quasi tutti i programmi che creano qualcosa, serve a sdoppiare l’oggetto dalla sua creazione, nel senso che viene chiamato privatamente una classe che serve solo a creare l’oggetto, invece che crearlo direttamente in sede.

Composite Pattern

Sono dubbioso su questo, ma leggendolo sembra comodo per creare e gestire i gruppi.  
gestisce il programma come se fosse un albero, quindi potremmo così gestire i possibili gruppi, creando “alberi di alberi” in caso un gruppo ne contenga un altro.  
anche disaccoppiarli secondo la logica degli alberi non dovrebbe essere complesso.

Singleton Pattern

In modo che ci sia in “moto” un’unica istanza del programma ed un’unica istanza costruttrice.

Observer Pattern

Per Gestire il marasma di dati che scorrono ad ogni modifica, in modo che sia tutto esatto e non esploda nulla perché si è modificata una dimensione, così da tenere sempre un “registro aggiornato” ad ogni modifica.