Exemple d'examen
Assignatura: **Disseny de Software**
Data: *** Curs: **2021/2022**

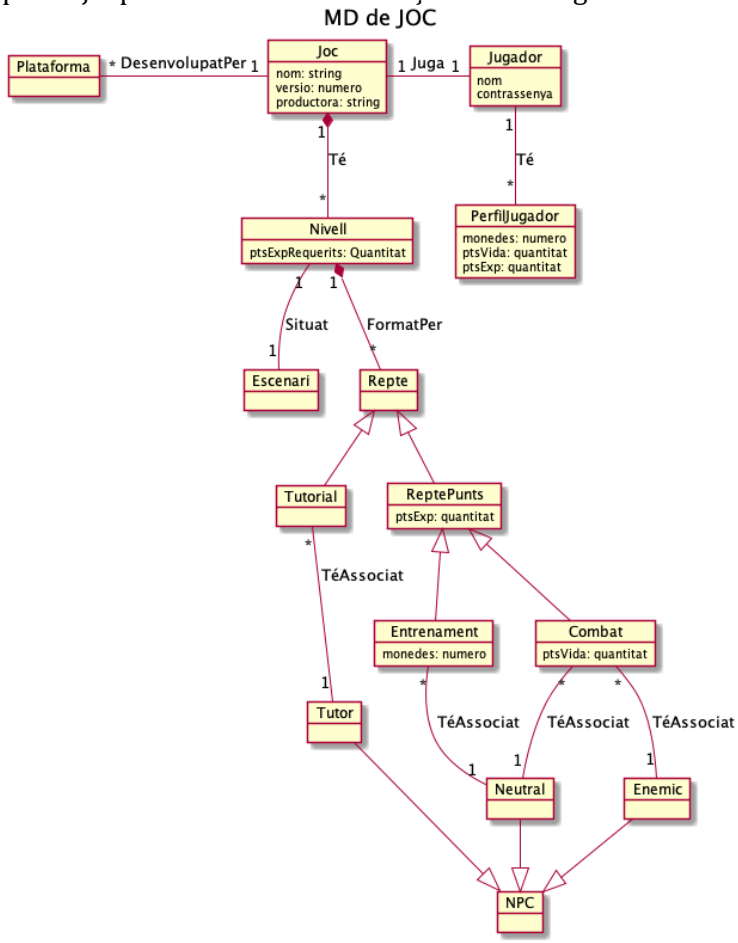
UNIVERSITAT DE BARCELONA

Indicacions:

- L'examen serà presencial, sinó hi ha una contraordre sanitària. En cas que s'hagi de canviar a online, l'examen es realitzarà el mateix dia i a la mateixa hora via Zoom.
- **Dia 11 de gener**: 15:00 a 18:00
- Aules i repartiment entre aules. Cal portar la mascareta en tot moment.
    - B3: ALBARRAN BERLANGA a LEE KIM (ambdós inclosos)
    - B5: LI a ZAMAN SHAHEEN (ambdós inclosos)
- L'examen constarà d'un conjunt de 2 o 3 problemes. Seran problemes obertes on caldrà que realitzeu algun disseny de Model de Domini, algun raonament de patrons GRASP i/o SOLID i l'aplicació d'algun patró de disseny.
- Per l'examen podreu portar un formulari que trobareu al campus amb tots els patrons de disseny. No es pot portar res apuntat en aquest formulari. **Cal que el porteu imprès des de casa.**
- Aquest examen conta 5,5 punts de la nota final de l'assignatura. Les valoracions que es posen en es diferents parts son relatives a puntuacions sobre 10.
- A continuació veieu un possible exemple d'examen

Preguntes obertes. **Temps ESTIMAT 2h-2:30h**

1. (60 min) (4 punts) Suposa que un dissenyador de software, en recollir les especificacions d'un problema plantejat per un usuari ha començat a fer el següent model de Domini per a modelar l'estructura d'un joc.



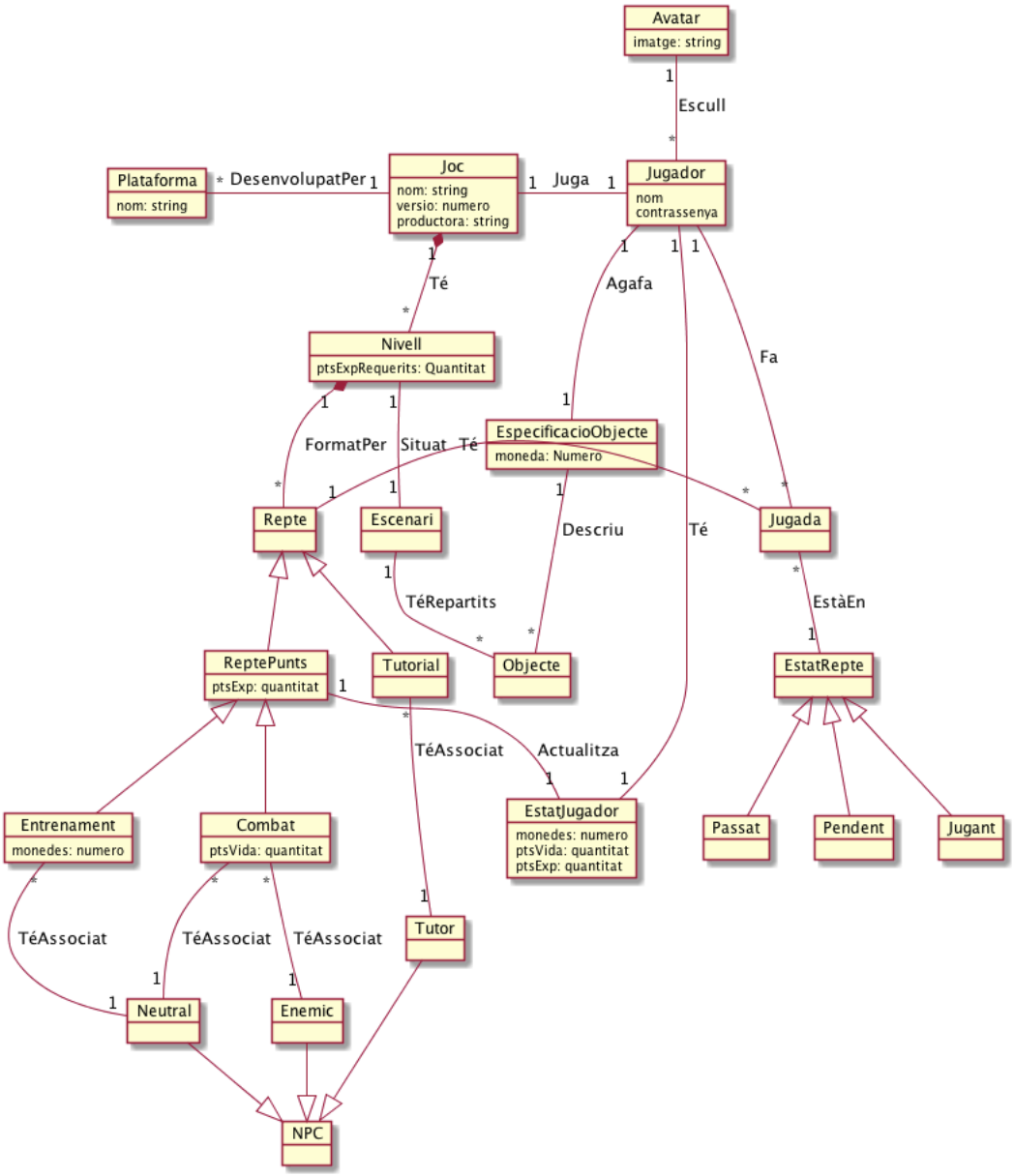
a) Avalua si el Model de Domini és correcta i justificar si trobes que cal fer algun canvi

En principi sembla un Model de Domini correcta, tot i que la cardinalitat de Jugador a Perfil de Jugador hauria de ser 1:1 i no 1:* com està al model, ja eu un Jugador té un únic perfil.

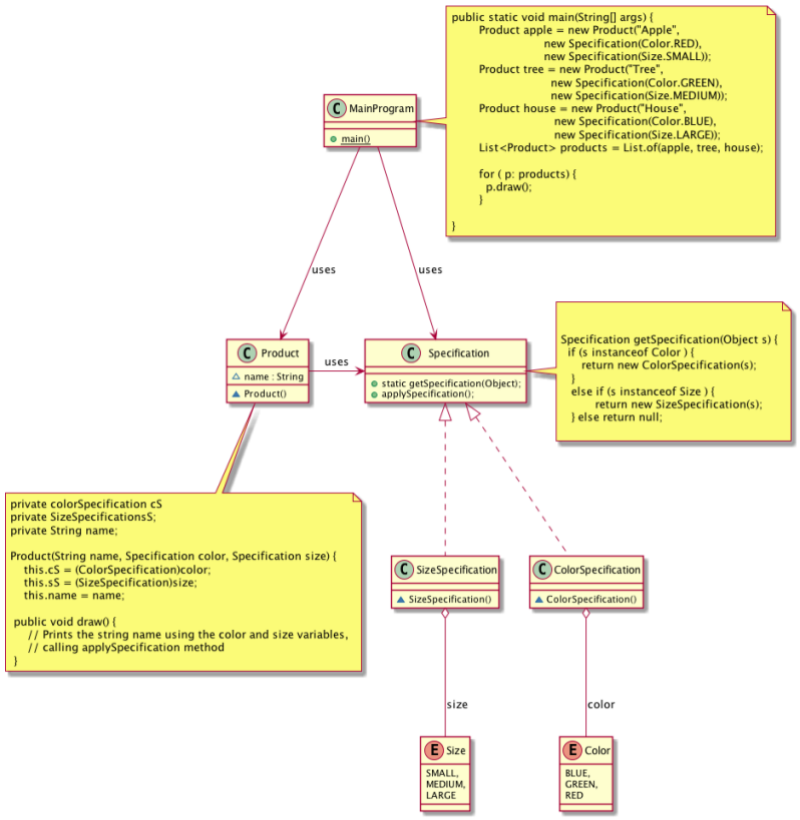
b) Si es volés incloure també la informació de l'evolució del joc per guardar les jugades que fa el jugador, quines classes afegiries/modificaries? les associacions i els atributs del teu Model de Domini que permeten definir els nivells que té el joc i que permeten modelar un moment de la sessió del joc, on el jugador té reptes ja superats, reptes que està jugant I reptes pendents de passar. Raona la teva proposta de canvi.

Per afegir la part dinàmica del Joc, s'afegeix la classe Jugada entre Jugador i Repte, així com la part de l'estat del Repte on es guardaran els diferents estats del Joc

MD de JOC


Avatar
imatge: string
Escull
Plataforma
nom: string
DesenvolupatPer
Joc
nom: string
versio: numero
productora: string
Juga
Jugador
nom
contrassenya
Té
Agafa
Fa
Nivell
ptsExpRequerits: Quantitat
FormatPer
Situat
Té
EspecificacioObjecte
moneda: Numero
Repte
Escenari
Descriu
Té
Jugada
TéRepartits
EstàEn
ReptePunts
ptsExp: quantitat
Tutorial
Objecte
EstatRepte
TéAssociat
Actualitza
Entrenament
monedes: numero
Combat
ptsVida: quantitat
EstatJugador
monedes: numero
ptsVida: quantitat
ptsExp: quantitat
Passat
Pendent
Jugant
TéAssociat
TéAssociat
TéAssociat
Tutor
Neutral
Enemic
NPC

2. (30 min) (2 punts) Donat el següent Diagrama de classes que ha fet un dissenyador de software per poder visualitzar de formes diferents un String



a) Quin/s principis SOLID o GRASP vulnera aquest disseny? Explora cadascun dels principis i indica quins són vulnerats i quins no. Justifica cadascuna de les teves afirmacions

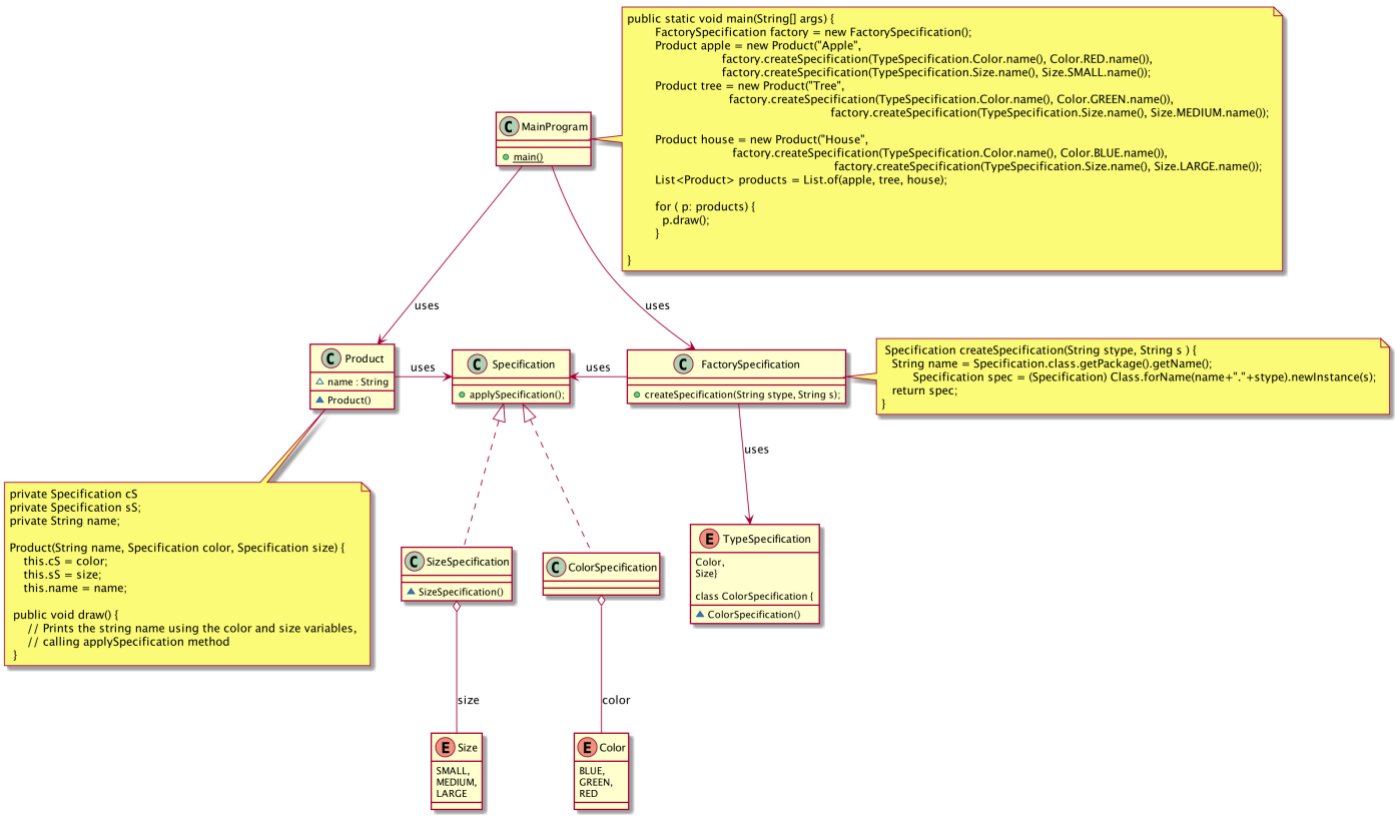
Aqui podríem estar vulnerant els principis Liskov i el OpenClosed. El Liskov a la Classe Specification, ja que es fan accions a la classe mare que no hauran de tenir les filles. Es la típica construcció que s'hauria de delegar a una Factory externa. Tambe es podria pensar que s'està vulnerant el DIP en declarar colorSpecification i SizeSpecification als atributs de la classe Producte

b) En cas que en vulneri algun, proposa un redisseny del Diagrama de Classes per a no vulnerar-lo

Aqui caldria aplicar un patró de Factory Method per treure la construcció de la classe getSpecification i no vulnerar el Liskov. Fer la implementació de Factory usant reflexivitat de Java per evitar el OpenClosed i declarar els atributs cs i sS com Specification.

Una possible solució seria:

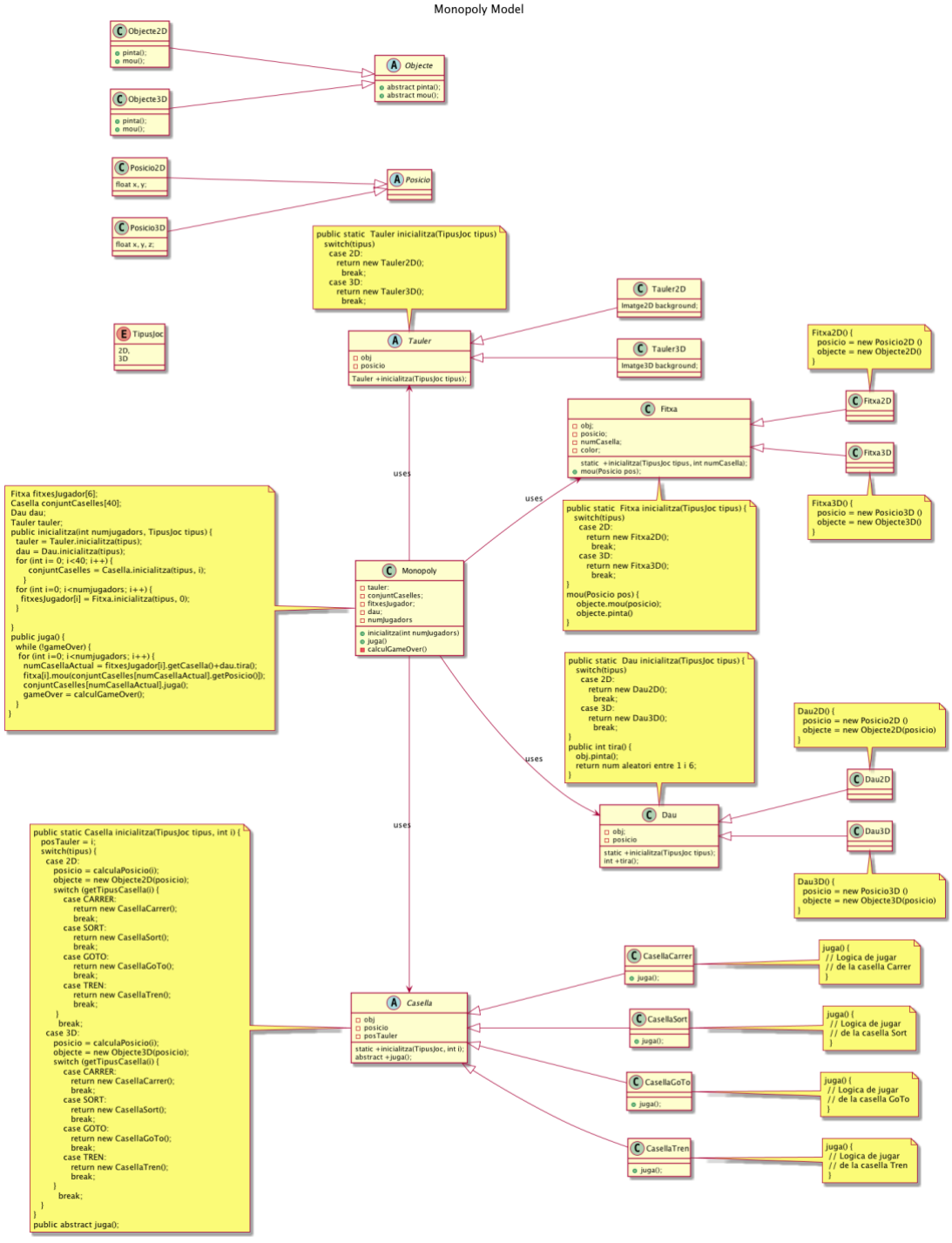
Class Diagram__



3. (60 min) (4 punts) Donat el següent diagrama de classes que vol modelar el Joc del Monopoly de forma que cada casella pot ser de diferents tipus: carrers, casella de la sort, casella de tren, casella d'anar a una altra casella. Cada casella permet fer diferents accions al jugador (comprar, agafar una carta, salta a una altra casella). Es vol pensar en la versió 2D per a tablet i 3D per PC de sobretaula, tot i que no es descarta en un futur fer una versió addicional per a realitat virtual.

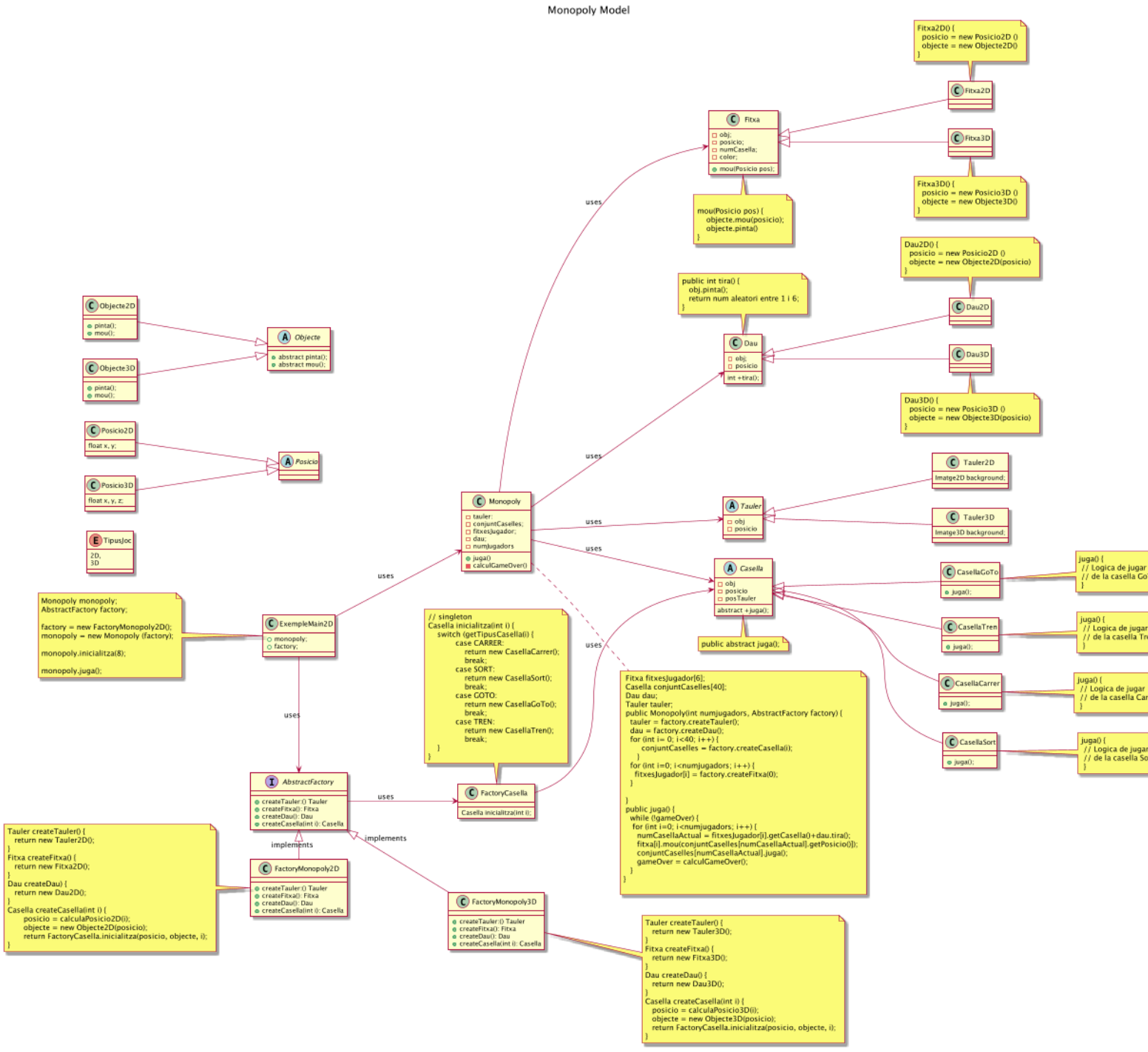
   Un dissenyador ha fet un primer disseny de les classes del Monopoly i de la distribució dels mètodes en les diferents classes a punt per a poder suportar les versions 2D i 3D, sense tenir en compte encara els hotels i les cases, la baralla de cartes o els diners de cada jugador. Quin patró seria més adient aplicar? Per què? Aplica'l per veure com quedaria el Diagrama de classes. Raona la teva resposta.

   Fixa't que les classes Posicio i Objecte s'usen des de diferents classes, però per a deixar més clar el disseny no s'han inclòs les associacions entre elles.



- Si exploro els principis SOLID, trobo que:
    - S: classe Monopoly: crea les dades i juga. Es poden considerar dues responsabilitats diferents
    - O: Clarament es vulnera en molts classes Tauler, Fitxa, Dau, Casella + enum TipusJoc que no són tancades a extensions ja que si s'afegeix un nou tipus de representació, caldria modificar els inicialitza
    - L: Vulnera el fet que la classe mare de Tauler, Dau, Fitxa, Casella esta coneixent les classes derivades en el seu mètode static.
    - I: No es vulnera. No hi ha cap interfície
    - D: Es podria pensar que en fer el new a les classes mare s'estan vulnarant algun tipus de dependency
- Patró que cal aplicar: Abstract Factory i Factory Method
- Aplicació:

  Aquesta es una possible solució, però en podien haver-ne d'altres també vàlides.



- Com usaria el patró des de la classe Monopoly?

```
public Monopoly(int numjugadors, AbstractFactory factory) {
  tauler = factory.createTauler();
  dau = factory.createDau();
  for (int i= 0; i<40; i++) {
      conjuntCaselles = factory.createCasella(i);
    }
  for (int i=0; i<numjugadors; i++) {
    fitxesJugador[i] = factory.createFitxa(0);
  }
}
```

- Anàlisi del patró aplicat en relació als principis S.O.L.I.D.

S: classe Monopoly: crea les dades i juga. Es poden considerar dues responsabilitats diferents. Ara la creacio es delega a la Factory que s'injecta a la classe Monopoly, tot i que segueix sent la responsable
O: S'ha encapsulat a la classe CasellaFactory. Tot i que es podria acabar treient si es fa un codi com

```
public enum CasellaFactory {
        INSTANCE;
        public Casella createCasella(String CasellaType) throws Exception {
                Casella Casella = null;
                String name = Casella.class.getPackage().getName();
                try {
                        Casella = (Casella) Class.forName(name + "." + CasellaType).newInstance();
                        return Casella;
                } catch (InstantiationException e) {
                        throw new Exception("The Casella type is not valid as a object!");
                } catch (IllegalAccessException e) {
                        throw new Exception("The Casella type is not found!");
                } catch (ClassNotFoundException e) {
                        throw new Exception("The Casella class is unknown!");
                }
        }
}
```

L: Ja no es vulnera
I: No es vulnera. Les calsses que implementen la interficie AbstractFactory realitzen tots els mètodes.
D: solucionat passant l'abstractFactory a monopoly per paràmetre

S'ha fet un FactoryMethod de tipus enum (singleton) per a la Factory de Caselles.

Per a complementar la solució es podria pensar en un Strategy per a les jugades a les caselles de forma que en una casella es podria injectar el seu comportament i fer una factory de comportaments més que de Caselles en si.

També s'ha donat per bo fer un una AbstractFactory a Casella pel tema de tenir una Posicio i un Objecte.