Casos d'Ús i Estructures de Dades i Algorismes

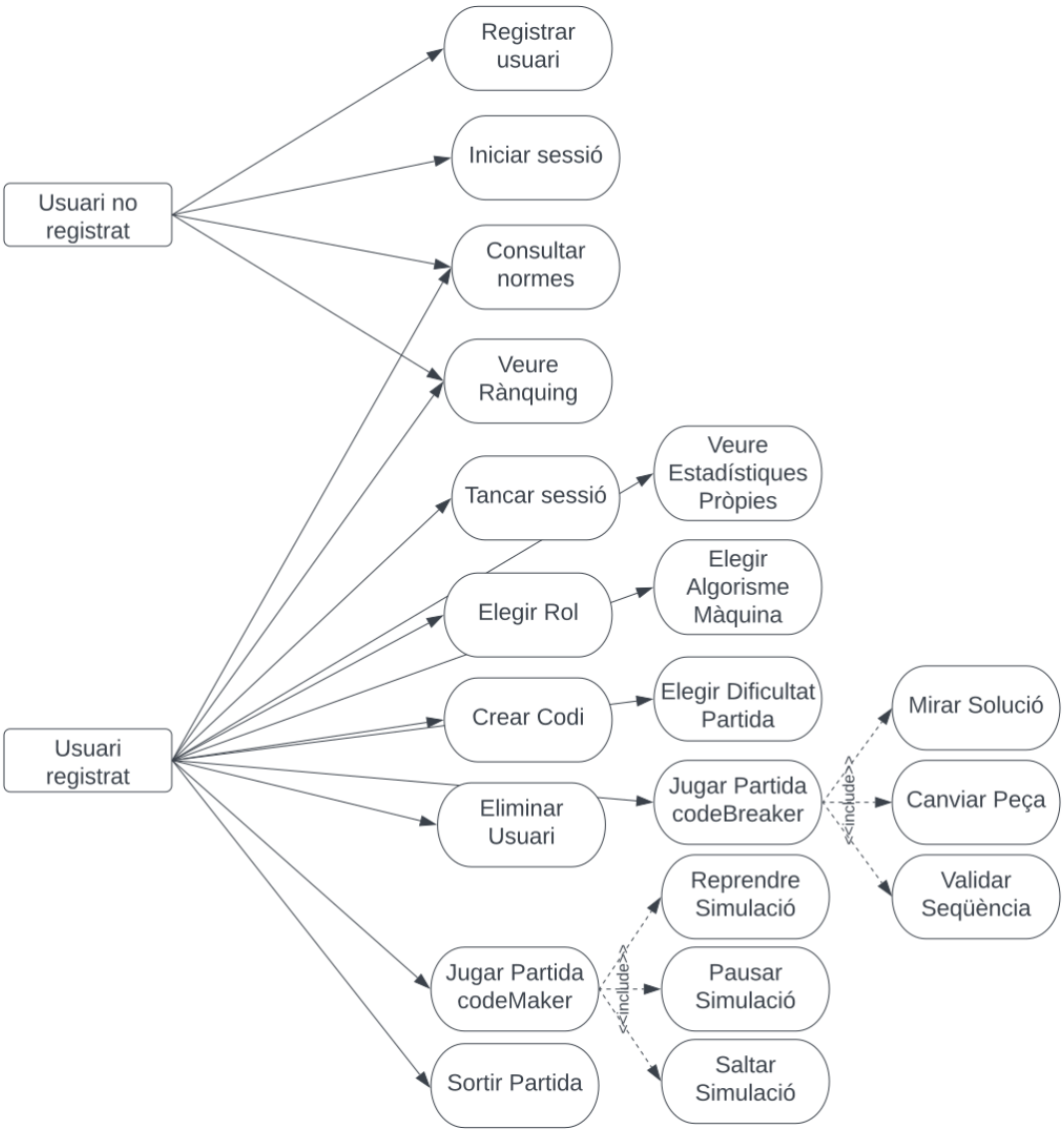
Albert Canales Ros
Mar Gonzàlez Català
Kamil Przybyszewski
Arnau Valls Fusté

Projectes de Programació

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Contents

1 Casos d'Ús


Usuari no registrat
Registrar usuari
Iniciar sessió
Consultar normes
Veure Rànquing
Usuari registrat
Tancar sessió
Veure Estadístiques Pròpies
Elegir Rol
Elegir Algorisme Màquina
Crear Codi
Elegir Dificultat Partida
Eliminar Usuari
Jugar Partida codeBreaker
<<include>>
Mirar Solució
Canviar Peça
Validar Seqüència
Jugar Partida codeMaker
<<include>>
Reprendre Simulació
Pausar Simulació
Saltar Simulació
Sortir Partida

1.1 Registrar Usuari

Actor: Usuari no registrat

Comportament:

L'actor escull fer un registre d'usuari. Ha d'indicar els seu nom i cognoms, codi d'usuari (*username*) i clau de pas (*password*). L'última ha de de ser inserida dos cops.

Errors possibles i cursos alternatius:

El codi d'usuari ja ha estat assignat en un altre registre: canviar o abandonar. El nom i cognoms, o *username*, o *password* son invàlids: tornar a introduir-les. Les dues contrasenyes donades no coincideixen: tornar a introduir-les.

1.2 Iniciar Sessió

Actor: Usuari no registrat

Comportament:

L'actor demana accedir al seu usuari associat. Indica el seu codi d'usuari (*username*) i la seva clau de pas (*password*).

Errors possibles i cursos alternatius:

La combinació de codi d'usuari i clau de pas no és correcta: tornar a introduir-les. No existeix cap usuari registrat amb aquest codi d'usuari: introduir un codi d'usuari diferent.

1.3 Eliminar usuari

Actor: Usuari registrat

Comportament:

L'actor indica que desitja esborrar l'usuari que té associat de la base de dades. El sistema esborra totes les seves dades personals del sistema i dels rànquings i automàticament tanca la sessió.

1.4 Consultar Normes

Actor: Usuari no registrat, Usuari registrat

Comportament:

L'actor demana veure la normativa del joc. El sistema presenta la normativa del joc.

1.5 Veure Rànquing

Actor: Usuari no registrat, Usuari registrat

Comportament:

L'actor demana veure el rànquing de puntuacions de les partides. El sistema presenta el llistat de les 20 millors partides ordenades per puntuació obtinguda.

Errors possibles i cursos alternatius:

No hi ha partides: s'indica amb un missatge i es presenta una llista buida.

1.6 Veure Estadístiques Pròpies

Actor: Usuari registrat

Comportament:

L'actor demana veure les seves estadístiques generals. El sistema presenta els valors de les estadístiques com a CodeBreaker i com a CodeMaker de l'usuari.

Els valors corresponents a codeBreaker es troben desglossats en cada un dels possibles valors de dificultat triada en les partides, a més d'un còmput total. Són els següents: Temps jugat, nombre de partides guanyades, nombre de partides perdudes, ratxa màxima de victòries, mitjana d'intents en les partides guanyades i mínim d'intents per guanyar.

Els valors correponents a codeMaker es troben desglossats en els possibles valors de l'algorisme triat en les partides, a més d'un còmput total. Són els següents: Mitjana d'intents de la màquina.

Errors possibles i cursos alternatius:

L'usuari no té partides: es presenten estadístiques buides.

1.7 Tancar Sessió

Actor: Usuari registrat

Comportament:

L'actor decideix deixar d'estar associat amb l'usuari indicat prèviament en l'inici de sessió. El sistema li ofereix les opcions d'inici de sessió i de registre d'un nou usuari.

1.8 Elegir Rol

Actor: Usuari registrat

Comportament:

L'actor comunica quin rol vol prendre en la següent partida que comenci: codemaker o codebreaker. El sistema registra les actualitzacions.

1.9 Elegir Dificultat Partida

Actor: Usuari registrat

Comportament:

L'actor comunica una la dificultat que tindrà la següent partida que inicï com a codebreaker: això és, el tipus de feedback que rebrà l'usuari actuant com a codebreaker. El sistema registra les actualitzacions. Respecte a la dificultat, hi ha tres valors possibles:

- Fàcil: com el feedback en el Mastermind tradicional, però les posicions de les boles blanques i negres corresponen amb les de l'intent.
- Mitjà (per defecte): com el feedback en el Mastermind tradicional.
- Difícil: com el feedback tradicional en el Mastermind, però eliminant les boles blanques.

1.10 Elegir Algorisme Màquina

Actor: Usuari registrat

Comportament:

L'actor indica quin dels algorismes disponibles vol que la màquina empri per a exercir de CodeBreaker en la pròxima partida que es jugui en aquest rol. Per defecte es pren l'algorisme *Five Guess*. El sistema guarda la informació.

1.11 Crear codi

Actor: Usuari registrat

Comportament:

L'actor indica la seqüència de colors que serà codi ocult en la pròxima partida que es jugui en aquest rol. El sistema guarda la informació, que utilitzarà a posteriori per comprovar si les seqüències proposades pel codebreaker són el codi ocult (i per tant, el joc ha de finalitzar). El codi per defecte és un codi buit.

Errors possibles i cursos alternatius:

La seqüència no és de quatre colors. El sistema informarà a l'actor de l'error i li demanarà de fer una correcció a la seva proposta de seqüència no guardant cap informació fins que se'n proposi una de vàlida.

1.12 Jugar Partida codeBreaker

Actor: Usuari registrat

Comportament:

L'actor indica que vol jugar una partida com a codeBreaker, ja sigui una partida que el sistema tingui guardada (la qual naturalment ha d'existir) o una de nova, per la qual se'n prendrà la dificultat que l'usuari ha indicat anteriorment (se n'ha d'haver indicat una anteriorment).

1.13 Canviar Peça

Actor: Usuari registrat

Comportament:

L'actor indica la peça que voldria inserir i la posició en l'intent en la qual desitjaria fer-ho. El sistema guarda aquesta selecció.

1.14 Validar seqüència

Actor: Usuari registrat

Comportament:

L'actor demana comprovar la semblança entre la seqüència completa que proposa i el codi ocult. Si és la correcta o s'exhaureixen els intents, el sistema dona la partida per guanyada o perduda respectivament i actualitza les estadístiques de l'usuari amb el que s'havia iniciat sessió.

1.15 Mirar Solució

Actor: Usuari registrat

Comportament:

L'actor comunica que vol saber el codi ocult de la partida que està jugant com a codeBreaker i el sistema li facilita. Automàticament es perd la partida.

1.16 Jugar Partida codeMaker

Actor: Usuari registrat

Comportament:

L'actor indica que vol jugar una partida com a codeMaker, ja sigui una partida que el sistema tingui guardada (la qual naturalment ha d'existir) o una de nova, per la qual se'n prendran els valors del codi ocult i l'algorisme que l'actor ha indicat anteriorment (el qual ha d'haver passat).

1.17 Reprendre simulació

Actor: Usuari registrat

Comportament:

L'usuari indica que desitja conèixer els intents de la màquina en la partida que està jugant com a codeMaker. El sistema li anirà mostrant els intents i feedbacks d'aquesta gradualment des de l'última iteració que coneixia l'usuari (la primera si no en coneixia cap).

1.18 Pausar simulació

Actor: Usuari registrat

Comportament:

L'usuari indica que desitja deixar de conèixer gradualment els intents i feedbacks de la màquina en la partida que està jugant com a codeMaker. El sistema deixa de mostrar noves iteracions de la màquina.

1.19 Saltar simulació

Actor: Usuari registrat

Comportament:

L'usuari indica que desitja conèixer tots els intents i feedbacks de la partida que està jugant com a codeMaker. El sistema els hi indica.

1.20 Sortir Partida

Actor: Usuari registrat

Comportament:

L'actor comunica que vol sortir de la partida i el sistema s'encarrega d'aturar-la.

2 Breu descripció de les estructures de dades i algorismes

En aquesta secció veurem en detall les estructures de dades usades al llarg de l'aplicació i les conseqüències del seu ús.

2.1 Domini

La capa de domini és la més densa en estructures de dades, ja que inclou la major lògica de l'aplicació. Així i tot, no totes les seves classes tenen estructures de dades notables. Entrem en detall en només aquelles que sí que en tenen.

2.1.1 Boles, Nivells de Dificultat i Tipus d'Algorismes

Hem representat aquests tres objectes diferents amb una mateixa estructura, l'*enum*. La raó principal per la qual no hem utilitzat un tipus primitiu com l'enter és per tenir un codi més llegible i mantenible.

A més, com que Java ens permet afegir petits mètodes als *enum*, també hi hem incorporat en el cas de bola un pocs mètodes estàtics que ens permeten evitar la duplicació de codi.

Finalment també és important destacar que l'ús d'aquestes estructures no fa el codi més lent, ja que quan la resta de classes les utilitzen i les guarden, sempre ho fan amb els seus valors numèrics.

2.1.2 Seqüències

Utilitzem la paraula *Seqüència* per referir-nos a una llista ordenada de *Boles*. Tot i que no tingui una classe com a tal ja que no era necessària, aquest concepte apareix repetidament en el codi en forma de:

- Solució. És la seqüència secret a descobrir en la partida.
- Intent. És una seqüència en la qual es pretén endevinar la solució.
- Feedback. És la resposta que el joc retorna al Breaker per calificar un intent.

Sabent que aquests tres conceptes estan representats per un mateix tipus, aquest tipus és una `ArrayList<Integer>`. Per ser més generals en el codi, en canvi, les guardem com a la seva superclasse `List<Integer>`.

La raó per la qual hem optat per una `ArrayList` en comptes de, per exemple, una `LinkedList` és purament per eficiència. Com que és comú fer modificacions en posicions arbitràries, el tipus escollit ho farà en temps constant respecte la llargada de la llista.

A més, tots els tipus de seqüència anomenats tenen una mida fixa (en cas de ser vàlids) que està especificada a la classe *Taulell*, en la variable `NUMBOLES`. Per

temes de l'algorisme, hem elegit que en el nostre joc aquest sigui 4. Això també coincideix amb la capacitat per defecte de `ArrayList`, així que també estem fent un ús eficient de la memòria en aquest aspecte.

2.1.3 Taullel

L'objecte *Taulell* és a nivell d'estructura de dades el més complex, ja que representa el contenidor amb els objectes necessaris per jugar una partida. Conté tres objectes:

- Solució. Seqüència solució de la partida
- Intents. Llista amb els intents de la partida.
- Feedbacks. Llista amb els feedbacks de la partida.

La primera, com s'ha dit abans, és una `List<Integer>`. Les altres dues són `List<List<Integer>>`. A diferència del cas anterior, aquí no modifiquem ni accedim les posicions intermitges de la llista major i augmentem dinàmicament el tamany afegint valors al final de la llista.

Degut a això, ens seria indiferent optar per una `LinkedList` o una `ArrayList`, per comoditat d'implementació, hem seguit elegint una `Arraylist`. Cal notar que les llistes tampoc creixen indefinidament, ja que tenen un màxim de `NUMINTENTS+1` i `NUMINTENTS` respectivament.

2.1.4 User

L'objecte *User* emmagatzema les dades relacionades amb l'usuari que actualment ha iniciat sessió. Aquestes dades ja estan sempre disponibles des de Persistència, es copien a Domini en l'inici de sessió per una qüestió de eficiència.

L'objectiu de la classe és que després de l'inici de sessió les estadístiques pròpies de l'usuari no s'hagin de consultar de Persistència quan siguin requerides per Presentació, sinó que les ofereixi directament domini.

Per això cal guardar les estadístiques pròpies. Moltes d'aquestes (les que són llistes) tenen un valor per cada NivellDificultat o TipusAlgorisme possible. Per facilitar les crides, les dades es reben amb llistes, que un cop més són `Arraylists` ja que ens interessa fer un accés i modificació en una posició aleatòria.

2.1.5 Rànquing

L'objecte *Rànquing* emmagatzema les dades de les partides de tots els usuaris en una llista ordenada per puntuació. Com amb *User*, aquestes dades també es troben a persistència, però es copien a domini per facilitar i agilitzar el seu ús.

Les dades estan estructurades en una `List` de `List<List<String>>`. Aquesta `List` té únicament 3 elements, un per dificultat. Cada `List<List<String>>`representa el rànquing per la respectiva dificultat.

Tots els rànquings comparteixen format. Cada `List<String>`que hi pertany codifica una partida guanyada en la corresponent dificultat. La codificació per cada partida consta de 3 elements: [username, intents, temps], on username pertany a l'usuari que va jugar la partida, i intents i temps son les estadístiques de la partida. Dins els rànquings les partides s'ordenen per la seva puntuació, calculada a partir dels atributs esmentats -la puntuació és millor quants menys intents hagin calgut, i en cas d'empat, quant menys temps s'hagi trigat-.

Els atributs es guarden com `String`, tot i representar valors numèrics, i per tant les partides es codifiquen en `List<String>`i els rànquings com `List<List<Str␣ing>>`. Els atributs es guarden tots així per no barrejar diferents tipus de dades en una mateixa `List`, i per poder escriure les dades a través de Presentació amb un processament mínim.

Estudiem ara els costos de les operacions efectuades amb els rànquings. Sigui N la mida d'un rànquing concret, és a dir, el nombre de partides emmagatzemades en ell. Portar un rànquing de Persistència a Domini requereix la creació d'un nou objecte, i per tant la de la llista que representa el rànquing. Així doncs, crear un objecte *Rànquing* a partir de les dades d'un existent té cost $O(N)$, és a dir, cost lineal. Inserir una nova partida dins un rànquing requereix comparar la puntuació de les partides ja presents al rànquing amb la puntuació de la partida a inserir per determinar el seu lloc. Comparar puntuacions és constant, i en cas pitjor es pot haver de comparar amb totes les partides del rànquing. A més, inserir un element en una posició aleatòria d'una `List` té cost lineal sobre la seva mida. Per tant inserir una partida al rànquing té cost $O(N)$. Finalment, es permet la funcionalitat d'esborrar un usuari del rànquing. Esborrar un element d'una `List` té cost lineal sobre la seva mida, i un sol usuari pot arribar a tenir totes les partides d'un rànquing. En conseqüència, esborrar un usuari d'un rànquing té cost $O(N^2)$.

2.1.6 Dificultat

La classe *Dificultat*, de la qual hereten les respectives classes per cada dificultat, és, mitjançant les seves subclasses, l'encarregada de calcular el feedback pels intents d'una partida.

No utilitzen cap estructura de dades més enllà dels `List<Integer>`amb els quals rep la seqüència solució i la seqüència intents que ha de comparar.

Respecte els algorismes que utilitzen per calcular el feedback, diferent per cada dificultat, tots tenen cost lineal sobre la mida de les seqüències. La raó és que tots els algorismes de feedback recorren les seqüències un nombre constant de cops. Com a màxim: una passada per comptar el nombre de boles de cada color (lineal gràcies a que el conjunt de possibles colors és finit), una passada per calcular posicions amb colors coincident -fitxes negres-, i una tercera passada per calcular colors coincidents però en posicions diferents -fitxes blanques-.

2.1.7 BotBreaker

L'objecte BotBreaker agrupa tota aquella informació rellevant sobre una partida en la qual la màquina juga en rol BotBreaker. L'objectiu de la classe és agrupar aquelles funcionalitats que són comunes en l'algorisme FiveGuess i Genetic.

L'estructura de dades que ambdós algorismes empren, i que cal inicialitzar, és el conjunt de possibles solucions. L'estructura que hem utilitzat en aquest cas és un `HashMap<Integer>`perquè és una estructura que es recorre molt ràpidament (en general cada consulta es realitza en temps constant però el cas pitjor és lineal), una funcionalitat que utilitzem constantment. Això passa perquè, en el cas de *FiveGuess*, *minimax* ho requereix i en el cas de *Genetic*, ens cal per a generar la població inicial en cada intent. A més, també ens ofereix la possibilitat de treure elements fet que tant l'algorisme de *FiveGuess* com el de *Genetic* exigeix.

2.1.8 FiveGuess

L'algorisme FiveGuess aconsegueix mitjançant l'ús de la tècnica minimax endevinar la seqüència oculta en cinc intents o menys. S'assumeix el feedback del mastermind clàssic: número de boles coincidents en color i posició -fitxes negres- i número de només coincidents en color -fitxes blanques-, donat un intent comparat amb la solució. També es pressuposa que el joc és amb seqüències de quatre boles i de sis colors possibles.

L'estratègia de l'algorisme per escollir el següent intent a enviar al CodeMaker es basa en la tècnica *minimax*. Per a cada possible seqüència, analitza de entre seqüències que encara romanen com possibles solucions -donats els feedbacks del CodeMaker previs- quantes compartirien cada possible feedback, és a dir, si enviéssim aquell intent i ens donés un cert feedback, a quant es reduiria la pool de possibles solucions. Guardem pel feedback que deixaria una pool de possibles solucions major, el tamany de la respectiva potencial pool. Com a següent intent selecciona la seqüència amb menor màxima pool dels feedbacks -d'aquí la terminologia *minimax*-. Amb aquest criteri es pot demostrar que arriba a la solució en com a màxim cinc intents.

Estructures de dades en la implementació del FiveGuess:

- Seqüència solució i seqüències intents (actual i finals): externament estan implementades com una `List<Integer>`perquè això ens demanava l'enunciat. Internament vam decidir implementar-les com una `ArrayList<Integer>`perquè la funció `E set(int index, E element)` ens permet fer l'equivalent a "col·locar una bola d'un color determinat" amb cost constant, cosa que necessitem fer de forma repetida internament en tot el projecte i en especial a la classe *FiveGuess*.
- Conjunt d'intents: externament està implementat com una `List<List<Integer>>`perquè això ens demanava l'enunciat. Internament són una `ArrayList<ArrayList<Integer>>`perquè aquesta estructura és coherent amb el que representa que és: un conjunt d'intents.

2.1.9 Genetic

L'algorisme Genetic utilitza una metaheurística basada en el procés de selecció natural per tal d'endevinar la seqüència oculta en el mínim d'intents possibles. S'assumeix el feedback del mastermind clàssic: número de boles coincidents en color i posició -fitxes negres- i número de només coincidents en color -fitxes blanques-, donat un intent comparat amb la solució. També es pressuposa que el joc és amb seqüències de quatre boles i de sis colors possibles.

L'estratègia de l'algorisme per escollir el següent intent a enviar al CodeMaker es basa en combinar solucions amb un bon fitness per obtenir un intent que a priori tindrà un bon fitness per ser combinació de seqüències que tenen aquesta propietat. Concretament, per cada intent, el primer que fa és prendre com a població inicial les seqüències que, segons el feedback rebut amb anterioritat, encara romanen com possibles solucions. Aquestes són les que donats els intents anteriors haguessin donat el mateix feedback. Entre les seqüències que conformen la població inicial fem parelles de seqüències (a,b) que combinem per l'obtenció d'una tercera (c) seqüència sent aquesta la conjunció de les dues primeres boles de a i les dues segones boles de b. Per evitar que l'algorisme enviï una mateixa seqüència com a intent múltiples vegades prenem la dràstica decisió de prendre la seqüència a en comptes de la c combinada en cas que aquesta darrera ja hagi estat un intent abans. Les seqüències c resultants de l'aparellament són la nostra segona generació. Realitzem aquest procés seqüencialment fins a obtenir una sola seqüència que esdevé el nostre intent. És important remarcar que de cada intent n'aprofitem el feedback per als configuració de la població inicial que, com és intuïtiu, es va reduint en mida.

Estructures de dades en la implementació del Genetic:

- Seqüència solució i seqüències intents (actual i finals): externament estan implementades com una `List<Integer>`perquè això ens demanava l'enunciat. Internament vam decidir implementar-les com una `ArrayLis t<Integer>`perquè la funció `E set(int index, E element)` ens permet fer l'equivalent a "col·locar una bola d'un color determinat" amb cost constant, cosa que necessitem fer de forma repetida internament en tot el projecte i en especial a la classe *Genetic*.
- Conjunt d'intents: externament està implementat com una `List<List <Integer>>`perquè això ens demanava l'enunciat. Internament són una `ArrayList<ArrayList<Integer>>`perquè aquesta estructura és coherent amb el que representa que és: un conjunt d'intents.
- Població inicial per a cada intent del *Genetic*: l'estructura que hem utilitzat en aquest cas és un `HashMap<Integer>`perquè és una estructura que es recorre molt ràpidament (el cas pitjor és lineal però estant ben implementada, la majoria de consultes es realitzen amb cost constant), funcionalitat que emprem constantment. El més important però és que ens ofereix la possibilitat de treure elements fet que l'algorisme de *Genetic*

exigeix per tal de passar d'una generació a la següent. De nou, el cost de treure un element és en general constant però cal mencionar que el cas pitjor és lineal.

- Seqüències intentades: hem emprat un `HashMap<Integer>`perquè de nou ens interessa comprovar si una seqüència concreta hi és o no ràpidament (operació que es realitza en general en temps constant encara que el cas pitjor sigui lineal). Per a fer-ho utilitzem la funció *contains*. També, quan afegim un nou intent, ens és molt útil la funció *add*, de nou una operació que podem considerar constant encara que a nivell teòric pugui ser lineal en algun cas. Cal destacar que en aquesta estructura de dades hi emmagatzem funció redundant (és la mateixa que ja figura en conjunt d'intents) però el format i la manera de guardar-ho ens compensa de sobres computacionalment com per poder assumir guardar un `Hashmap` amb menys de deu enters emmagatzemats.

2.2 Presentació

La capa de presentació és la més lleugera a nivell d'estructures de dades. De fet, moltes classes no contenen directament cap estructura de dades o algorisme notable per mencionar, així que no seran indicades.

2.2.1 BolaColor

És l'equivalent a *Bola* però en capa de presentació. A diferència de domini, aquí per a cada valor possible de la bola també guardem, a part del seu valor numèric, el color que el representa.

Aquesta representació ens permet lligar aquests dos conceptes (valor numèric i color corresponent) d'una forma eficient, ja que en tenir el valor de l'*enum* es poden obtenir els dos valors sense necessitat de buscar en cap conjunt ni estructura de dades similar.

Pels casos on volem obtenir el valor de l'*enum* només a partir del valor numèric (com és el cas dels valors que rebem de domini), tenim un mètode addicional que permet fer-ho en temps lineal respecte el nombre de boles.

2.2.2 BolaButton

És una classe que encapsula un botó amb una bola. Naturalment la bola tindrà associat un *BolaColor* que representi el seu color.

Aquesta classe no té cap estructura notable.

2.2.3 SequenciaPanel

De nou, aquesta classe és l'equivalent al concepte de *Seqüència* discutit a domini. Similarment, consisteix en essència un vector. A diferència de domini, aquí el vector és de **BolaButtons**, ja que és l'element que es vol mostrar en la UI.

Hem usat un vector ja que així podem assignar propietats de les boles de forma eficient tant seqüencial com arbitràriament. Aquest és l'ús principal de la classe, i conté mètodes per facilitar aquests accessos.

2.2.4 TaullelPanel

Aquesta classe, com les anteriors, té un equivalent en domini. Per aquesta classe és *Taulell*.

Per tant, de la mateixa manera que en *Taulell* teníem *Sequencies* per representar la solució, els feedbacks i els intents, aquí tindrem *SequenciaPanels*.

Tal com en la secció anterior, guardem els intents i feedbacks en un vector, ja que és una estructura de dades en la qual tenim un ordre definit fàcilment i que és constant respecte a la llargada de la llista per a accessos aleatoris, els quals farem repetidament durant el joc.

Més concretament tenim que, en utilitzar vectors tant per la llista d'intents com per les boles dins de l'intent en si, quan ho tenim tot en compte veiem que l'accés a una bola arbitrària del taulell segueix tenint un cost constant.

2.2.5 BolaPalettePanel

Aquesta classe no té un equivalent en domini. Es un panell que representa una paleta de colors dels quals se'n pot seleccionar un.

Tal com en el cas de *SequenciaPanel*, com que ens interessa fer accessos aleatoris, utilitzem un vector per fer-ho en temps constant sobre el nombre de colors.

2.2.6 PartidaBreakerView

Aquesta classe no té estructures de dades pròpies, ja que aquestes són emmagatzemades completament en *TaulellView* a presentació i *Taulell* a domini.

2.2.7 PartidaMakerView

Aquesta classe, a diferència de l'anterior, sí que té estructures de dades pròpies. Això és necessari per una qüestió d'optimització.

Com que la màquina soluciona tota la partida de cop, en *Taulell* a domini es guardarà tots els intents i feedbacks. Així i tot, volem que l'usuari vegi les boles de *TaulellPanel* d'intent en intent.

Podríem demanar els intents i feedbacks a domini cada cop que volguéssim mostrar la següent iteració, però això no seria gaire eficient.

Per aquesta raó, tenim les mateixes estructures per guardar intents i feedbacks en la pròpia *PartidaMakerView*. Aquestes estructures es carreguen només a l'inici de la vista en comptes de a cada nova iteració.

Considerem N la mida de les boles i M el nombre d'intents. Amb aquesta implementació, tot i que en obrir la vista tenim un cost $O(NM)$, per a cada iteració tindrem un cost $O(N)$ en comptes de $O(NM)$ (amb la implementació més directa).

2.3 Persistència

2.3.1 GestorCSVFile

Al utilitzar la llibreria *OpenCSV* hi ha dues maneres de llegir les línies d'un fitxer: llegir línia per línia en forma d'`String[]` fins trobar el que necessitem o bé llegir totes les línies de cop en forma de `List<String[]>`.

Al llegir línia per línia necessitem avançar l'iterador, així que per simplificar les operacions hem escollit llegir i escriure tot el fitxer excepte quan volem afegir una línia al final, ja que podem aprofitar la funcionalitat *append* de la classe *File*.

Finalment, aquesta classe conté algunes funcions comunes utilizades per *GestorPartidesActuals*, *GestorRanquing* i *GestorUsuaris* que ajuden a convertir diferents tipus de llistes en `String[]` (la operació inversa també), que com ja hem dit és el format en el que acabarem escrivint al CSV.