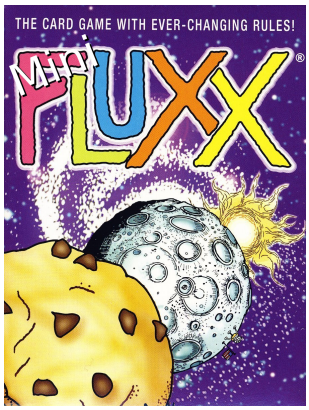
Trabajo Práctico 3
Programación Orientada a Objetos

Paradigmas de Lenguajes de Programación – 1° cuat. 2017

Fecha de entrega: 22 de Junio



Objetivo

El juego se llama MiniFluxx[1], un juego de cartas cuyas reglas fluctúan a medida que transcurre el juego. ¿Su objetivo? En realidad va cambiando durante el desarrollo del juego (en esta versión el objetivo es lo único que cambia).

Elementos

- Lista de jugadores: los turnos son asignados de manera circular.
- Mazo de cartas: contiene cartas que representan Tesoros (TreasureCard) y objetivo (GoalCard). Nadie puede ver el contenido del mazo, solo puede conocerse su tamaño.
- Cartas tesoro (TreasureCard): representan objetos para coleccionar.
- Cartas objetivo (GoalCard): indica el conjunto de tesoros que se debe obtener para ganar el juego. Inicialmente está ausente (nil), pero que puede ser definido (y redefinido) por los jugadores al jugar una GoalCard.
- Mesas: donde los jugadores irán colocando sus tesoros.

[1] El nombre se debe a que es una versión reducida del juego Fluxx

Desarrollo

Cada jugador tiene una mano de cartas y una mesa. Todos pueden ver los tesoros que cada jugador tiene en su mesa, pero solo el propio jugador conoce las cartas que tiene en su mano (los demás solo pueden ver la cantidad de cartas que tiene).

El jugador tiene además una estrategia que le permite, conociendo su mano, su mesa y lo que puede ver del estado del juego, decidir qué carta va a jugar. Si no se definió explícitamente una estrategia, jugará siempre la primera carta que tenga en la mano.

Las estrategias se representan con bloques de la forma `[:mano :mesa :juego | acciones]`.

Cuando un jugador juega una carta hay dos opciones:

- que la carta sea una TreasureCard, entonces la misma pasa a formar parte de su mesa.
- que la carta sea una GoalCard, esta se convierte en el nuevo objetivo del juego, descartando el anterior si lo hubiere.

El Juego

Al iniciar el juego se reparte una cierta cantidad de cartas del mazo a cada jugador. Si no es posible repartir la cantidad de cartas deseada a todos los jugadores, entonces no se reparte ninguna carta y los jugadores empiezan con las manos vacías.

Cada jugador, en su turno, levanta una carta del mazo (la cual pasa a su mano), luego juega una carta de acuerdo con su estrategia, y finalmente pasa el turno al siguiente jugador.

El primer jugador que logre alcanzar el objetivo del momento se convierte en ganador.

Si se acaba el mazo, los jugadores siguen jugando con las cartas que tienen en la mano, y si estas también se acaban, se termina el juego aun si no hay ganador (es lo que se considera un empate).

Se puede asumir que no hay cartas repetidas.

Ejercicio 1. Implementar las clases y métodos necesarios para poder crear y comparar los distintos tipos de cartas, mostrarlas como strings y guardarlas en colecciones.

Las cartas deben poder imprimirse de alguna manera que permita ver su contenido, y deben poder compararse por su igualdad observacional (mensaje `#=`, dos cartas son iguales si son el mismo tipo de carta y tienen el mismo contenido). También se debe poder decidir correctamente su pertenencia a una colección.

Pista: para que las colecciones de cartas funcionen como se espera, se debe implementar el método correspondiente al mensaje `#hash`.

Al finalizar este ejercicio deben poder pasar los tests de la clase `Test01Cards`.

Ejercicio 2. Implementar las clases y métodos necesarios para poder crear un jugador a partir de un nombre. Tener en cuenta que la mesa y la mano comienzan vacías. Dado un jugador se debe poder observar la mesa y cuántas cartas tiene en su mano (pero no cuáles). Además los jugadores deberán poder levantar una carta del mazo, o recibir una determinada cantidad de cartas de un mazo dado. Cada carta que se levanta se agrega al final de la mano.

Finalmente, un objetivo debe ser capaz de responder si una colección de cartas lo cumple. Por ejemplo, la mesa de un jugador para saber si ganó, o un mazo para saber si el objetivo es alcanzable.

Notar que puede ser necesario agregar nuevos métodos y/o variables de instancia a clases ya definidas.

Al finalizar este ejercicio deben poder pasar los tests de la clase `Test02Players`.

Ejercicio 3. Completar la implementación para poder asignar estrategias a los jugadores y poder jugar el juego. Tener en cuenta que un jugador gana cuando alcanza el objetivo con las cartas de su mesa.

Notar que puede ser necesario agregar nuevos métodos y/o variables de instancia a clases ya definidas.

Al finalizar este ejercicio deben poder pasar los tests de la clase `Test03Games`.

Pista: las cartas están hechas específicamente para jugar MiniFluxx, por lo que, dado un juego, pueden saber cómo jugarse (ya sea colocándose en la mesa del jugador actual o convirtiéndose en el nuevo objetivo del juego).

Ejercicio 4. *(Opcional)* Si terminaron y tienen ganas de seguir experimentando, pueden definir nuevas estrategias, o incluso introducir cartas de reglas – que no son tesoros ni objetivos, sino que cambian la cantidad de cartas a levantar y a jugar en cada turno – o cartas de acción – que fuerzan a un jugador a levantar, jugar o incluso descartar una o más cartas, o permiten robar tesoros de la mesa de otro jugador – y desarrollar sus propios tests.

Pautas de entrega

El entregable debe contener:

- un archivo `.st` con todas las clases implementadas
- versión impresa del código, comentado adecuadamente (puede ser el propio .st sin los tests)
- **NO** hace falta entregar un informe sobre el trabajo.

Se espera que el diseño presentado tenga en cuenta los siguientes factores:

- definición adecuada de clases y subclases (si corresponde), con responsabilidades bien distribuidas
- uso de polimorfismo y/o delegación para evitar exceso de condicionales (no usar más de un **ifTrue:**, **ifFalse:** o **ifTrue:ifFalse:**; sí se pueden usar libremente mensajes como **ifNil:**,**ifEmpty:**,**ifNotEmpty:**,**detect:ifNone:**,**detect:ifFound:**,**whileTrue:**, etc.)
- intento de evitar código repetido utilizando las abstracciones que correspondan.

Consulten todo lo que sea necesario.

Consejos y sugerencias generales

- Lean al menos el primer capítulo de *Pharo by example*, en donde se hace una presentación del entorno de desarrollo.
- Explorar la imagen de Pharo suele ser la mejor forma de encontrar lo que uno quiere hacer. En particular tengan en cuenta el buscador (`shift+enter`) para ubicar tanto métodos como clases.
- No se pueden modificar los test entregados, si los hubiere, aunque los instamos a definir todos los tests propios que crean convenientes.

Importación y exportación de paquetes

En Pharo se puede importar un paquete arrastrando el archivo del paquete hacia el intérprete y seleccionando la opción "`FileIn entire file`". Otra forma de hacerlo es desde el "`File Browser`" (botón derecho en el intérprete > `Tools` > `File Browser`, buscar el directorio, botón derecho en el nombre del archivo y elegir "`FileIn entire file`").

Para exportar un paquete, abrir el "`System Browser`", seleccionar el paquete deseado en el primer panel, hacer click con el botón derecho y elegir la opción "`FileOut`". El paquete exportado se guardará en el directorio `Contents/Resources` de la instalación de Pharo (o en donde esté la imágen actualmente en uso).