99.7 % Citric Liquid

Descripción del proyecto

Profesor: Alexandre Bergel
Auxiliares: Beatriz Graboloza
Ignacio Slater
Tomás Vallejos
Semestre: Otoño 2020

Resumen

Uno de los principales objetivos del curso de *Metodologías de diseño y programación* es que el alumno aprenda a escribir programas de calidad utilizando buenas metodologías de programación. Para lograr este objetivo se le plantea un proyecto semestral dividido en 3 entregas en las que se espera ver aplicados los contenidos enseñados en el curso.

En este caso, el proyecto buscará simular un «caso real» en el que usted deberá programar un juego de acuerdo a las indicaciones de un cliente. Los requisitos específicos serán explicados en el presente documento, pero para guiar el desarrollo de la solución se plantearan tareas periódicas opcionales[1] donde cada entrega del proyecto será la acumulación de las tareas más pequeñas hasta la fecha de entrega. Las mini-tareas que compondrán cada una de las evaluaciones del proyecto se especificarán por el foro cuando sea pertinente.

Para comenzar con el proyecto se le entregará un código base sobre el que deberá construir su solución, teniendo total libertad de cambiar cuanto estime conveniente del código original mientras se mantengan buenas metodologías de diseño.

1. Contexto

El proyecto a realizar será crear un clon (simplificado) del juego *100 % Orange Juice* desarrollado por *Orange_Juice* y distribuido por *Fruitbat Factory*. Este es un *juego de tablero por turnos* de 4 jugadores donde cada uno maneja un personaje distinto con características propias. El objetivo del juego es cumplir una cierta cantidad de requisitos para llegar al nivel más alto antes que los otros jugadores.

Su cliente ya tiene un implementación base de algunas de las características básicas del juego, pero se dio cuenta de que no tiene los conocimientos necesarios para terminarlo así que le encarga a usted la tarea de completarlo, considerando que puede modificar el código original.

1.1. *Norma*

Para ganar una partida un jugador debe alcanzar un *Norma 6* antes que el resto de los participantes. Todos los jugadores comienzan la partida con *Norma 1* y deben cumplir una serie de objetivos específicos para subir su nivel de *Norma*.

Exceptuando el primer nivel, el jugador puede elegir siempre entre 2 objetivos para subir de nivel, estos son, alcanzar una cierta cantidad de *estrellas* o una cierta cantidad de *victorias*, dicho objetivo se elige cada vez

[1] Que las mini-tareas sean opcionales significa que no serán evaluadas individualmente ni se tomará en cuenta la fecha en que sean entregadas, sino que será evaluado que se hayan realizado los requerimientos acumulados de estas en las 3 fechas de entrega del proyecto.

que un jugador sube de nivel y no puede ser cambiado hasta que sea completado (cada jugador tiene un objetivo propio). El cuadro 1 muestra los objetivos que deben alcanzarse para subir de nivel.

Cuando se cumple alguno de los requisitos para ganar una *Norma* el jugador no sube inmediatamente, sino que debe pasar por una etapa llamada *Norma check* que se explicará más adelante. Si el jugador cumple los requisitos de la *Norma* y pasa por el proceso de *Norma check* entonces aumentará su nivel, a esto se le llama *Norma clear*.

1.1.1. *Stars Norma*

Cada jugador cuenta con un número de estrellas que puede aumentar o disminuir en el desarrollo de la partida, donde $stars \in \mathbb{N}$.

Al comienzo del turno de cada jugador éste gana una cantidad de estrellas igual a:

$$\left\lfloor \frac{chapters}{5} \right\rfloor + 1$$

1.1.2. *Wins Norma*

Los jugadores también cuentan con un contador de victorias, este contador solamente puede aumentar (a diferencia de las estrellas).

Las victorias se obtienen al ganar combates (estos se explicarán más adelante), y la cantidad de victorias que se gana al combatir dependerá del tipo de enemigo al que se enfrentó. Las cantidades de victorias dadas por cada tipo de enfrentamiento son:

- *Encounter unit*: 1 victoria
- Otro jugador: 2 victorias
- *Boss unit*: 3 victorias

Esto se explicará en más detalle en la sección 1.3.

Norma	*Stars*	*Wins*
2	10	
3	30	2
4	70	5
5	120	9
6	200	14

Cuadro 1: Objetivos para subir de nivel

1.2. Tablero

El tablero es el lugar donde se juega la partida y se ubican los jugadores. El tablero está formado de paneles conectados entre sí donde:

- cada panel tiene un tipo,
- cada panel puede estar siendo ocupado por uno o más jugadores, y
- cada panel tiene uno o más paneles siguientes.

Al inicio de su turno el jugador debe lanzar un dado de 6 caras (un *D6*), y se moverá esa cantidad de paneles en la dirección que indiquen los paneles. En el caso en que se pase por un panel que tenga más de 1 panel siguiente, entonces el jugador debe elegir cuál camino tomar.

Existen múltiples tipos de paneles, que se especificarán a continuación.

1.2.1. *Neutral panel*

Este tipo de panel no tiene ningún tipo de efecto sobre el jugador, si un jugador cae en un panel de estos entonces termina su turno sin cambios.

1.2.2. *Home panel*

Cada jugador tiene un *Home panel* en el que se ubica cuando comienza el juego, estos paneles se activan de dos formas:

- Si el jugador es dueño del panel puede elegir detenerse en él al pasar sobre este incluso si le quedan movimientos disponibles.
- Si el jugador no es dueño del panel entonces sólo se activa si cae exactamente en este.

En ambos casos el turno termina luego de activar el panel.

Al activarse, este panel tiene 2 efectos: recuperar un punto de salud al personaje y realizar un *Norma check*.

Definición 1 (*Norma check*)**.** Este evento solo puede ser desencadenado por un *Home panel*. En caso de que el jugador cumpla con los requisitos especificados en el cuadro 1 entonces su nivel de *Norma* aumenta en 1.

1.2.3. *Bonus panel*

Estos paneles otorgan estrellas al jugador. Cuando se cae en uno de estos paneles se debe lanzar un dado y de acuerdo al resultado el jugador gana una cantidad de estrellas igual a:

$$\text{mín}\{roll \cdot Norma, 3 \cdot roll\}$$

1.2.4. *Drop panel*

Este tipo de panel hace que el jugador pierda estrellas. Al caer en uno de estos paneles el jugador debe arrojar un dado y pierde $roll \cdot Norma$ estrellas (considerando que el número de estrellas del jugador siempre debe ser mayor o igual a 0).

1.2.5. *Draw panel*

Estos paneles tienen que ver con las cartas que se explican en la sección 1.4. Cuando un jugador cae en este panel roba una carta del mazo y la agrega a su mano.

1.2.6. *Encounter panel*

Al caer en este panel el jugador entra en batalla con un *wild unit* aleatorio. Estos se explican en la sección 1.3.2.

1.2.7. *Boss panel*

Este panel funciona de la misma forma que un *Encounter panel* con la diferencia de que el enemigo al que se enfrenta es un *Boss unit* (véase sección 1.3.3)

1.3. Batallas

Las batallas son eventos que pueden darse entre dos jugadores o entre un jugador y un *wild* o *boss unit*. Las batallas pueden ser desencadenadas cuando un personaje cae en un panel específico o por la activación de una carta.

Todos los personajes tienen una cantidad de *hit points* máximos y una cantidad de *hit points* actuales. Cuando los *hit points* de un jugador llegan a 0, entonces el jugador queda en estado *K.O.* (esto se explicará en la sección 1.5).

Una batalla se realiza de la siguiente forma:

1. Los jugadores eligen si quieren activar alguna carta para la batalla (solamente pueden seleccionar una)
2. El jugador que activó la batalla lanza el dado para decidir cuánto su ataque base.
3. Se le suma al ataque base del jugador la cantidad indicada en su campo *ATK*.
4. El jugador que recibe el ataque debe elegir entre defender o esquivar y luego lanza el dado:
   - Si se escoge **defender** entonces este jugador recibe un daño igual a
   $$\text{máx}\{1,\ roll_{atk} + ATK - (roll_{def} + DEF)\}$$
   - Si se escoge **esquivar** entonces si $roll_{evd} + EVD > roll_{atk} + ATK$ el jugador no recibe daño, pero en caso contrario recibe el daño total del ataque.
5. Si al personaje que recibió el ataque le quedan *hit points* entonces contraataca de la misma forma que en los pasos 2 a 4.
6. La batalla termina.

Las batallas con todos los tipos de personajes (ya sean jugadores o no) siguen el mismo patrón.

1.3.1. Jugador contra jugador

Las batallas entre jugadores pueden comenzarse de dos formas:

- Cuando un jugador pasa por sobre un panel en el que haya otros jugadores puede elegir enfrentarse a ellos:
  - Si decide enfrentarse al otro jugador, entonces se inicia una batalla entre ambos. Una vez que la batalla termina el jugador se queda en el panel en el que se detuvo y termina su turno. Si hay otro jugador en el panel también se le da la opción de enfrentarlo.
  - Si no se comienza una batalla, entonces el jugador sigue avanzando como indicaba el dado.
- Por el efecto de una carta. Lo que suceda luego de una batalla iniciada de esta forma dependerá de lo que especifique la carta.

Si alguno de los jugadores reduce los *hit points* de su oponente a 0, entonces aumenta su contador de victorias en 2 y se le transfieren la mitad (aproximado hacia abajo) de las estrellas del oponente (esto quiere decir que uno pierde y el otro gana estrellas).

1.3.2. *Wild unit*

Este tipo de batalla comienza cuando un jugador cae en un *Encounter panel* o por el efecto de alguna carta.

En estas batallas el jugador siempre ataca primero y el enemigo elige al azar entre defender y esquivar. Si el enemigo derrota al jugador, entonces el jugador pierde la mitad de sus estrellas y el enemigo se queda con

estas. Cuando un jugador vence a un enemigo, todas las estrellas que el enemigo haya acumulado del resto de los jugadores pasan al jugador y se aumenta su contador de victorias en 1.

Luego de las batallas, los enemigos mantienen los *hit points* con los que quedaron al final del combate. Si un *Wild unit* es derrotado, entonces se crea uno nuevo con las mismas características, pero con su salud completa y sin estrellas.

Cada vez que se desencadena una batalla contra un *Wild unit*, se elige al azar entre los 3 personajes que se muestran en el cuadro 2.

Enemigo	HP	ATK	DEF	EVD
Chicken	3	-1	-1	+1
Robo Ball	3	-1	+1	-1
Seagull	3	+1	-1	-1

Cuadro 2: *Wild units*

1.3.3. *Boss unit*

Estos encuentros comienzan cuando un jugador cae en un *Boss panel* **y** alguno de los jugadores de la partida ha alcanzado *Norma 4*, si ningún jugador cumple esa condición entonces al caer en un *Boss panel* se pelea con un *Wild unit*.

Las batallas con este tipo de enemigos son iguales que las descritas en la sección anterior, con la única diferencia siendo que si se derrota a éste el jugador aumenta su contador de victorias en 3.

Durante la partida existirá un solo *Boss unit* que será elegido al azar de los especificados en el cuadro 3. Cuando esta unidad es derrotada todos los *Boss panel* pasaran a desencadenar un encuentro con un *Wild unit* en vez de con un *Boss*.

Enemigo	HP	ATK	DEF	EVD
Store Manager	8	+3	+2	-1
Shifu Robot	7	+2	+3	-2
Flying Castle	10	+2	+1	-3

Cuadro 3: *Boss units*

1.4. Cartas

Las cartas son otro elemento fundamental del juego, estas pueden ser jugadas al comienzo del turno de cada jugador y tienen efectos sobre el juego o los personajes.

Las cartas se almacenan en un mazo de 40 cartas que es común para todos los personajes y que es barajado al comienzo de la partida. Si en cualquier momento de la partida el mazo se vacía, entonces se vuelven a barajar todas las cartas que no estén en las manos de los jugadores.

Un jugador puede tener hasta 3 cartas en la mano al comenzar su turno, pero puede tener más cartas durante su turno. En caso de que un jugador terminara su turno con más de 3 cartas entonces debe elegir cartas para descartar hasta volver a tener 3 antes de que comience el turno del siguiente jugador.

Al comienzo de la partida se le otorga una carta al azar a cada jugador, de modo que todos comienzan con una carta en su mano.

Todas las cartas tienen un precio y una *Norma* mínima. El **precio** de una carta son la cantidad de estrellas que debe pagar un jugador para utilizar una carta (en cuyo caso el jugador pierde tantas estrellas como el

precio de la carta) y la ***Norma* mínima** es la cantidad de *Normas* que debe haber completado el jugador, ambas condiciones deben cumplirse para que la carta pueda ser jugada.

En el juego existen 3 tipos de cartas que tienen distintas restricciones y efectos. Cada vez que una carta es **activada**, esta se descarta.

1.4.1. *Battle cards*

Estas cartas solamente pueden ser jugadas al comienzo de una batalla y su efecto permanece activo solamente durante esa batalla.

Las siguientes son las cartas de este tipo presentes en el juego:

I'm on fire	
Efecto:	Durante esta batalla, gana +1 de ataque y -1 de defensa.
***Norma*:**	1
Precio:	5
Big Magnum	
Efecto:	Paga 1 de HP para jugar esta carta. Durante esta batalla gana +2 de ataque. Esta carta no puede ser usada si el jugador tiene 1 de HP.
***Norma*:**	3
Precio:	20
Sink or Swim	
Efecto:	Gana -1 ATK, -1 DEF y -1 EVD. Si ganas la batalla toma el 75 % de las estrellas de tu oponente.
***Norma*:**	4
Precio:	10

1.4.2. *Boost cards*

Estas cartas solamente pueden ser activadas al comienzo del turno de un jugador y su efecto es instantáneo. La duración del efecto puede variar dependiendo de lo que especifique el efecto de la carta.

A continuación se especifican las cartas a implementar:

Dash!	
Efecto:	Por este turno, lanza el dado 2 veces y avanza la suma de ambos.
***Norma*:**	1
Precio:	3
Saki's Cookie	
Efecto:	Recupera 1 de HP.
***Norma*:**	1
Precio:	0
Pudding	
Efecto:	Recupera todo tu HP.
***Norma*:**	4
Precio:	0
Nice present	
Efecto:	Roba 2 cartas.
***Norma*:**	2
Precio:	10

1.4.3. *Trap cards*

Estas cartas se pueden jugar solamente al comienzo del turno del jugador, pero su efecto no se activa inmediatamente. Al jugar una de estas cartas ésta queda asociada al panel en el que se encuentra el personaje que la jugó.

El efecto de esta carta se activa cuando cualquier jugador cae en el panel en el que se jugó la carta.

Las cartas disponibles son:

Invasion	
Efecto:	Inicia un combate con un *wild unit*. El enemigo ataca primero.
***Norma*:**	1
Precio:	0
Piyopiyo Procession	
Efecto:	El jugador sufre 3 batallas aleatorias con *wild units*. El jugador ataca primero.
***Norma*:**	2
Precio:	0
Sky Restaurant 'Pures'	
Efecto:	Pierde la mitad de tus estrellas y recupera todo tu HP.
***Norma*:**	4
Precio:	0

1.5. Turno

Cada partida del juego se divide en capítulos que a su vez se dividen en 4 turnos cada uno (uno por cada jugador).

Al comienzo de la partida se asigna el orden de los turnos a los jugadores de manera aleatoria y se mantiene ese mismo orden a lo largo de la partida. Cada vez que los 4 jugadores terminan sus respectivos turnos entonces se pasa al siguiente capítulo, que en este caso es solamente un contador.

El turno de cada jugador se estructura de la siguiente manera todos estos ítems hacen referencia al **dueño del turno**:

1. Si está en estado *K.O*, entonces pasa a la fase de *recovery* que se explica más adelante.
2. Recibe la cantidad de estrellas especificada en 1.1.1.
3. Tiene la opción de jugar o no una carta.
4. El jugador lanza el dado y se mueve la cantidad de paneles indicadas por el resultado exceptuando los casos en los que puede detenerse anticipadamente (encuentro con otro jugador, pasar por su *home panel*).

   Si en algún momento se pasa por algún panel que tenga más de un panel siguiente, entonces el jugador debe elegir por cuál camino continuar y avanza por ese camino la cantidad de movimientos restantes
5. Al momento de detenerse en un panel, si hay alguna carta en éste entonces se activa su efecto.
6. Se activa el efecto del panel.
7. Se pasa al turno del siguiente jugador

1.5.1. *Recovery phase*

Cuando un jugador es derrotado y pasa a estado *K.O.* entra a esta fase especial.

Mientras el jugador se encuentra en *recovery* **no puede** jugar su turno. Para salir de esta fase, el jugador debe arrojar el dado y obtener una cantidad mayor o igual a la requerida para recuperarse. Si el jugador logra recuperarse, entonces puede jugar su turno inmediatamente.

La cantidad necesaria para recuperarse comienza en 6 y va disminuyendo en 1 a medida que pasan los capítulos, i.e. $r_1 = 6$, $r_2 = 5$, $r_3 = 4 \ldots$

2. Modelo de la solución

Para guiar la solución de este problema, se plantearan objetivos pequeños en forma de *mini-tareas* que buscarán enfrentar una problemática a la vez. Dichas tareas tendrán fecha de entrega pero será opcional entregar en la fecha señalada y no serán evaluadas con nota. Solamente será evaluado que al momento de entregar la tarea se cumplan todos los objetivos planteados en las *mini-tareas*.

La resolución de este proyecto se hará siguiendo el patrón arquitectónico *Modelo-Vista-Controlador*[2], donde primero se implementará el *modelo*, luego el *controlador* y por último la *vista*. Este patrón se explicará en más detalle en el transcurso del curso, pero en el contexto del proyecto estos componentes serán como se explica a continuación.

Modelo Para la primera parte se le solicitará que cree todas las entidades necesarias que servirán de estructura base del proyecto y las interacciones posibles entre dichas entidades. Las entidades en este caso se refieren a los elementos que componen el juego.

Vista Se le pedirá también que cree una interfaz gráfica simple para el juego que pueda responder al input de un usuario y mostrar toda la información relevante del juego en pantalla.

Controlador Servirá de conexión lógica entre la vista y el modelo, se espera que el controlador pueda ejecutar todas las operaciones que un jugador podría querer efectuar, que entregue los mensajes necesarios a cada objeto del modelo y que guarde la información más importante del estado del juego en cada momento.

3. Código base

El código base se les entregará mediante un link que será proporcionado por el foro.

Este código incluye la implementación de parte del modelo del juego. Además, el código contiene *tests* que prueban su funcionalidad y está documentado para que su comprensión sea más sencilla. Parte de su trabajo será juzgar el diseño del código entregado y, en caso de que se necesitara, modificarlo para que cumpla los estándares de diseño que se enseñan en el curso, **no puede asumir** que el código que se le entrega esté bien diseñado ni que esté totalmente testeado. Lo que sí puede asumir es que los tests escritos son correctos y efectivamente comprueban el funcionamiento de los métodos que se describirán en las secciones siguientes.

El código entregado corresponde a dos de las entidades fundamentales del juego, los paneles y los jugadores.

3.1. Panel

Los paneles se implementan en la clase `Panel` del paquete `com.github.cc3002.citricjuice.model.board`, además en este mismo paquete se encuentra la enumeración `PanelType` que representa los posibles tipos de paneles.

[2] https://www.github.com/CC3002-Metodologias/apunte-y-ejercicios/wiki/Modelo-Vista-Controlador

El constructor de la clase `Panel` recibe un `PanelType` y guarda el tipo como una propiedad de la clase.

```
public Panel(final PanelType type) {
  this.type = type;
}
```

Aquí, la *keyword* `final` indica que el **parámetro** `type` no será sobrescrito (no confundir con la variable de instancia `private final PanelType type`). Además, el constructor implícitamente crea una variable de instancia `private final Set<Panel> nextPanels = new HashSet<>()` que es un conjunto (no ordenado) de los paneles siguientes a éste.

Se incluyen en la clase los *getters* y *setters* correspondientes a los campos de la clase. De estos cabe destacar el de la variable `nextPanels` que retorna una copia del set de paneles siguientes, esto es una buena práctica para impedir que el set de paneles sea cambiado de una forma impredecible o indebida.

```
public Set<Panel> getNextPanels() {
  return Set.copyOf(nextPanels);
}
```

El método principal de la clase es el que define qué hacer dependiendo del tipo de panel que sea.

```
public void activatedBy(final Player player) {
  switch (type) {
    case BONUS:
      applyBonusTo(player);
      break;
    case DROP:
      applyDropTo(player);
      break;
    case HOME:
      applyHealTo(player);
      break;
    case NEUTRAL:
      break;
  }
}
```

Note que los casos en los *boss* y *encounter panels* no están definidos, ni la funcionalidad de *Norma check* al caer en un *home panel*.

Los métodos implementados se preocupan de que se cumpla con las reglas del juego, por ejemplo, al activar un *bonus panel* se verifica que el resultado de lanzar el dado se multiplique a lo más por 3.

```
private static void applyBonusTo(final @NotNull Player player) {
  player.increaseStarsBy(player.roll() * Math.min(player.getNormaLevel(), 3));
}
```

3.2. Jugador

El jugador es representado por la clase `Player` ubicada en el paquete `com.github.cc3002.citricjuice.model`.

El constructor de esta clase recibe 5 parámetros, el nombre que identifica al personaje, sus puntos de vida máximos (que también serán los iniciales), y sus *stats* de ataque, defensa y evasión.

```
public Player(final String name, final int hp, final int atk, final int def,
              final int evd) {
  this.name = name;
```

```
  this.maxHP = currentHP = hp;
  this.atk = atk;
  this.def = def;
  this.evd = evd;
  normaLevel = 1;
  random = new Random();
}
```

Note que además se inicia la *Norma* en 1 y se crea un generador de números aleatorios. Implícitamente se inicia la cantidad de estrellas como 0.

El jugador lleva dos contadores distintos para su *HP*: uno que se mantiene igual durante toda la partida y que son sus puntos de salud máximos, y otro que indica la vida actual del jugador (representado por la variable `currentHP`).

Se incluyen los *getters* y *setters* necesarios para todos los campos tomando en cuenta las restricciones de estos valores, v.g. el código se asegura de que el jugador no pueda tener una cantidad de *HP* negativa:

```
public void setCurrentHP(final int newHP) {
  this.currentHP = Math.max(Math.min(newHP, maxHP), 0);
}
```

Un *setter* en particular que es pertinente destacar es el método `setSeed(long)`. Lo que hace dicho método es quitarle el no-determinismo al generador de números aleatorios, forzando que (para dos generadores con la misma semilla) siempre se retorne la misma secuencia de números. Esto se utilizará para testear los métodos que dependan de valores aleatorios.

El método `roll(): int` es el que simula el lanzamiento de un dado y entrega un número aleatorio con distribución uniforme en $[1, 6]$.

Por último, se incluyen dos métodos adicionales que pueden ser de utilidad al momento de testear a esta clase: `equals(Object): boolean` y `copy(): Player`. El primero de estos métodos comprueba que dos instancias de la clase `Player` sean iguales, mientras que el segundo retorna una **nueva instancia** de la clase con todas sus propiedades iguales a la que invocó el método.

```
@Override
public boolean equals(final Object o) {
  if (this == o) {
    return true;
  }
  if (!(o instanceof Player)) {
    return false;
  }
  final Player player = (Player) o;
  return getMaxHP() == player.getMaxHP() &&
         getAtk() == player.getAtk() &&
         getDef() == player.getDef() &&
         getEvd() == player.getEvd() &&
         getNormaLevel() == player.getNormaLevel() &&
         getStars() == player.getStars() &&
         getCurrentHP() == player.getCurrentHP() &&
         getName().equals(player.getName());
}

public Player copy() {
```

```
  return new Player(name, maxHP, atk, def, evd);
}
```

3.3. *Tests*

Junto con las implementaciones anteriormente mencionadas vienen los *tests* para esas clases.

No se ahondará mucho en el código ya que debiera ser bastante auto explicativo.

Para todos los ejemplos siguientes se tomará en cuenta que las variables de instancia de los tests se inician como:

```
// PlayerTest
@BeforeEach
public void setUp() {
  suguri = new Player(PLAYER_NAME, 4, 1, -1, 2);
}

// PanelTest
@BeforeEach
public void setUp() {
  testBonusPanel = new Panel(PanelType.BONUS);
  testBossPanel = new Panel(PanelType.BOSS);
  testDropPanel = new Panel(PanelType.DROP);
  testEncounterPanel = new Panel(PanelType.ENCOUNTER);
  testHomePanel = new Panel(PanelType.HOME);
  testNeutralPanel = new Panel(PanelType.NEUTRAL);
  testSeed = new Random().nextLong();
  suguri = new Player(PLAYER_NAME, BASE_HP, BASE_ATK, BASE_DEF, BASE_EVD);
}
```

A grandes rasgos, existen 3 tipos de tests.

El primer tipo es el que comprueba que los objetos se creen de manera correcta, para eso puede tomar como ejemplo el método:

```
// PlayerTest
@Test
public void constructorTest() {
  final var expectedSuguri = new Player(PLAYER_NAME, 4, 1, -1, 2);
  assertEquals(expectedSuguri, suguri);
}
```

Note que esto se puede hacer ya que se definió el método `equals(Object): boolean` en la clase `Player` y se sabe que funciona ya que en el mismo archivo del *test* existe un chequeo del funcionamiento de dicho método.

El segundo tipo de *test* es el que comprueba que el programa se comporte de la manera esperada al enviar un mensaje a un objeto (que se cambie algún parámetro, que se retorne algún valor, etc.). Tome como ejemplo el siguiente método:

```
// PanelTest
@Test
public void nextPanelTest() {
  assertTrue(testNeutralPanel.getNextPanels().isEmpty());
  final var expectedPanel1 = new Panel(PanelType.NEUTRAL);
```

```
  final var expectedPanel2 = new Panel(PanelType.NEUTRAL);

  testNeutralPanel.addNextPanel(expectedPanel1);
  assertEquals(1, testNeutralPanel.getNextPanels().size());

  testNeutralPanel.addNextPanel(expectedPanel2);
  assertEquals(2, testNeutralPanel.getNextPanels().size());

  testNeutralPanel.addNextPanel(expectedPanel2);
  assertEquals(2, testNeutralPanel.getNextPanels().size());

  assertEquals(Set.of(expectedPanel1, expectedPanel2),
              testNeutralPanel.getNextPanels());
}
```

Lo primero que hace este test es comprobar que los parámetros iniciales del panel que se utilizará como sujeto de prueba sean los esperados, luego agrega 2 paneles como siguientes y comprueba que se agreguen correctamente y, finalmente, comprueba que el objeto se comporte como debiera en el caso de borde en que se intente agregar el mismo panel dos veces.

Por último, están los ***tests* de consistencia**, estos se encargan de comprobar que se cumplan los invariantes y que el programa se comporte de la manera correcta frente a diversos escenarios, esto es especialmente útil para evaluar secciones de código que tengan un comportamiento no determinista. Tome como ejemplo:

```
// PanelTest
@RepeatedTest(100)
public void bonusPanelConsistencyTest() {
  int expectedStars = 0;
  assertEquals(expectedStars, suguri.getStars());
  final var testRandom = new Random(testSeed);
  suguri.setSeed(testSeed);
  for (int normaLvl = 1; normaLvl <= 6; normaLvl++) {
    final int roll = testRandom.nextInt(6) + 1;
    testBonusPanel.activatedBy(suguri);
    expectedStars += roll * Math.min(3, normaLvl);
    assertEquals(expectedStars, suguri.getStars(),
                "Test failed with seed: " + testSeed);
    suguri.normaClear();
  }
}
```

La anotación `@RepeatedTest` indica que este *test* se correrá múltiples veces (en particular 100). Nuevamente, lo primero que se hace es ver que las condiciones iniciales sean correctas. Luego, se crea un generador de números aleatorios con la misma semilla que se le pasa a la instancia de `Player` (`testSeed` cambia cada vez que se ejecuta una iteración de este *test* ya que se inicializa en el método `setUp()`), esto hará que se pueda predecir el resultado cada vez que el jugador lance el dado. Para terminar, se prueba el efecto de activar el panel para todos los niveles del jugador tomando en cuenta el resultado predicho del dado.

4. Evaluación

4.1. Bonificaciones

Además del puntaje asignado por cumplir con los requisitos anteriores, puede recibir puntos adicionales (*que se sumarán a su nota total*) implementando lo siguiente:

- **Mini-tareas** (0.4 pts.): De acuerdo a las entregas que haya hecho de las mini-tareas puede recibir una bonificación de hasta 0.4 pts. Entregue las mini-tareas solamente si considera que están completas o altamente completas. Entregar mini-tareas demasiado incompletas puede significar un descuento en el puntaje. Para los casos de mini-tareas que estén completas pero contengan errores pueden optar a una fracción del puntaje.
- **Principios SOLID** (0.2 pts.): Si su tarea no rompe ninguno de los principios SOLID recibirán una bonificación.
- **Exclusivamente para la tarea 3:**
  - **Interfaz gráfica avanzada** (0.5 pts.): Si su interfaz gráfica cumple más de los requisitos mínimos podrá recibir una bonificación de hasta 0.5 pts. (esto quedará a criterio del ayudante que le revise).
  - **Manejo de excepciones** (0.3 pts.): Se otorgará puntaje por la correcta utilización de excepciones para manejar casos de borde en el juego. Recuerde que atrapar *runtime exceptions* es una mala práctica, así como arrojar un error de tipo `Exception` (esto último ya que no es un error lo suficientemente descriptivo). Bajo ninguna circunstancia una de estas excepciones debiera llegar al usuario.

4.2. Requisitos adicionales

Que su programa funcione no es suficiente, se espera además que éste presente un buen diseño, siendo esta la característica a la que se le dará **mayor importancia** en la revisión de sus entregas.

Sus programas además deberán estar bien testeados, por lo que **NO SE REVISARÁN** las funcionalidades que no tengan un test correspondiente que pruebe su correcto funcionamiento. Para revisar esto, también es de suma importancia que el trabajo que entregue pueda ejecutarse (que su código compile), en caso contrario no podrá revisarse la funcionalidad y por ende no tendrá puntaje en este aspecto.

Otro aspecto que se tendrá en cuenta al momento de revisar sus tareas es que estas estén bien documentadas, siguiendo los estándares de documentación para *Java* de *Google*.[3]

Todo su trabajo debe ser subido al repositorio privado que se le entregó a través de *Github Classroom*. La modalidad de entrega será mediante un resumen en **formato PDF** entregado mediante *u-cursos* que contenga su nombre, *rut*, un diagrama de clases de su programa y un enlace a su repositorio (no se aceptará código recibido por este medio)[4] Además su repositorio deberá contener un archivo `README.md`[5]que contenga todas las instrucciones necesarias para ejecutar su programa, todos los supuestos que realice y una breve explicación del funcionamiento y la lógica de su programa.

Para la tarea deberá crear su propia rama para trabajar y subir todo su código en esta y, cuando tenga una entrega lista para ser revisada, realizar un ***pull request***[6], esta será la versión que será revisada. Tenga en cuenta que si hace *push* de otros *commits* en esa rama se actualizará el *PR*, por lo que se le recomienda crear una nueva rama para seguir trabajando.

[3] https://github.com/CC3002-Metodologias/apunte-y-ejercicios/wiki/Convenciones
[4] No cumplir con estas condiciones implicará descuento de puntaje.
[5] https://help.github.com/en/articles/basic-writing-and-formatting-syntax
[6] https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/creating-a-pull-request

4.3. Plazos de entrega

No se aceptarán peticiones de extensión de plazo, pero dispondrá de 72 horas a lo largo del semestre para entregar tareas atrasadas sin que se aplique descuento. Esto significa que si la entrega de la primera tarea se hace con 72 horas de atraso, entonces las siguientes tendrán que entregarse sin atraso. No es necesario dar aviso al cuerpo docente para usar las horas, simplemente se revisará el *commit* correspondiente a la entrega de acuerdo a su *Pull request*.

Las horas de atraso serán aproximadas: para esto, se considerará cada media hora de atraso como 1 hora, y menos que eso como 0 horas, exceptuando la primera hora. Entonces, si se entrega la tarea con 1 minuto de atraso, cuenta como 1 hora de atraso; si se entrega con 1 hora y 1 minuto, también contará como 1 hora; si se entrega con 1:30 de atraso se contará como 2 horas, y así.

La entrega de las mini-tareas se hará también mediante *u-cursos*, para esto basta entregar un documento de texto plano con su nombre, *rut*, y un enlace al *commit* que cuente como entrega.

4.4. Evaluación

- **Código fuente (4.0 puntos):** Este ítem se divide en 2:
  - **Funcionalidad (1.5 puntos):** Se analizará que su código provea la funcionalidad pedida. Para esto, se exigirá que testee las funcionalidades que implementó[7]. **Si una funcionalidad no se testea, no se podrá comprobar que funciona y, por lo tanto, NO SE ASIGNARÁ PUNTAJE por ella.**
  - **Diseño (2.5 puntos):** Se analizará que su código provea la funcionalidad implementada utilizando un buen diseño.
- ***Coverage* (1.0 puntos):** Sus casos de prueba deben crear diversos escenarios y contar con un *coverage* de las líneas de al menos 90 % por paquete. No está de más decir que sus tests deben testear algo (es decir, no ser tests vacíos o sin *asserts*).
- ***Javadoc* (0.5 puntos):** Cada clase, interfaz y método público debe ser debidamente documentado. Se descontará por cada falta.
- **Resumen (0.5 puntos):** El resumen mencionado en la sección anterior. Se evaluará que haya incluido el diagrama UML de su proyecto. **En caso de no enviarse el resumen con el link a su repositorio su tarea no será corregida**.[8]

4.5. Recomendaciones

Como se mencionó anteriormente, no es suficiente que su tarea funcione correctamente. Este curso contempla el diseño de su solución y la aplicación de buenas prácticas de programación que ha aprendido en el curso. Dicho esto, no se conforme con el primer diseño que se le venga a la mente, intente ver si puede mejorarlo y hacerlo más extensible.

No comience su tarea a último momento. Esto es lo que se dice en todos los cursos, pero es particularmente importante/cierto en este. Si usted hace la tarea a último minuto lo más seguro es que no tenga tiempo para reflexionar sobre su diseño, y termine entregando un diseño deficiente o sin usar lo enseñado en el curso.

Haga la documentación de su programa en inglés (no es necesario). La documentación de casi cualquier programa *open-source* se encuentra en este idioma. Considere esta oportunidad para practicar su inglés.

[7] Se le recomienda enfáticamente que piense en cuales son los casos de borde de su implementación y en las fallas de las que podría aprovecharse un usuario malicioso.

[8] Porque no tenemos su código.

Se les pide encarecidamente que las consultas referentes a la tarea las hagan por el **foro de U-Cursos**. En caso de no obtener respuesta en un tiempo razonable, pueden hacerle llegar un correo al auxiliar o ayudantes.

Por último, el orden en el que escriben su programa es importante, se le sugiere que para cada funcionalidad que quiera implementar:

1. Cree los *tests* necesarios para verificar la funcionalidad deseada, de esta manera el enfoque está en como debería funcionar ésta, y no en cómo debería implementarse. Esto es muy útil para pensar bien en cuál es el problema que se está buscando resolver y se tengan presentes cuales serían las condiciones de borde que podrían generar problemas para su implementación.
2. Escriba la firma y la documentación del método a implementar, de esta forma se tiene una definición de lo que hará su método incluso antes de implementarlo y se asegura de que su programa esté bien documentado. Además, esto hace más entendible el código no solo para alguna persona que revise su programa, sino que también para el mismo programador[9].
3. Por último implemente la funcionalidad pensando en que debe pasar los tests que escribió anteriormente y piense si estos tests son suficientes para cubrir todos los escenarios posibles para su aplicación, vuelva a los pasos 1 y 2 si es necesario.

[9] Entender un programa mal documentado que haya escrito uno mismo después de varios días de no trabajar en él puede resultar bastante complicado.