5 DE OCTUBRE DE 2018

ANÁLISIS TÉCNICO

EJERCICIO SIETE Y MEDIA

IVÁN CEBALLOS VEGA

1. INTRODUCCIÓN

Este análisis técnico refiere a un conjunto de aplicación de técnicas de ingeniería de software para conseguir un producto que se ajuste al pedido con la máxima calidad posible y que, a la vez, sea mantenible y escalable para un equipo de software.

El plazo del que se dispone es de 1 semana (7 días) en el cual se realizará el análisis, implementación, prueba y entrega.

2. **REQUISITOS**.

2.1. Requisitos técnicos.

- El programa se tiene que poder ejecutar en un ordenador que tenga instalado el lenguaje de programación PHP. Se requiere que la versión de PHP a usar sea 5.6, así que se programará usando ese estándar.
- El programa no contiene ninguna interfaz gráfica y, como se especifica, se usará un entorno de comandos en línea, como por ejemplo BASH en Linux o CMD en Windows.
- En las especificaciones también se indica que no se usará ningún tipo de memoria persistente, esto quiere decir que se almacenarán los datos en memoria. Lo cual deriva a que se tiene que mantener una cantidad no muy elevada de objetos a la vez para no sobrecargar la memoria y a la vez que los datos no se almacenarán.
- Se requiere que el programa sea escalable.

2.2. Requisitos funcionales.

- Se precisa que se puedan realizar varias partidas. Para esto se implementará un bucle infinito que parará sólo cuando el usuario desee.
- Se requiere explícitamente que sea totalmente aleatorio.
- Se indican una serie de reglas para que la lógica del producto funcione de forma correcta.
- Si ambos jugadores quedan empatados en la puntación de 7’5, la máquina gana.

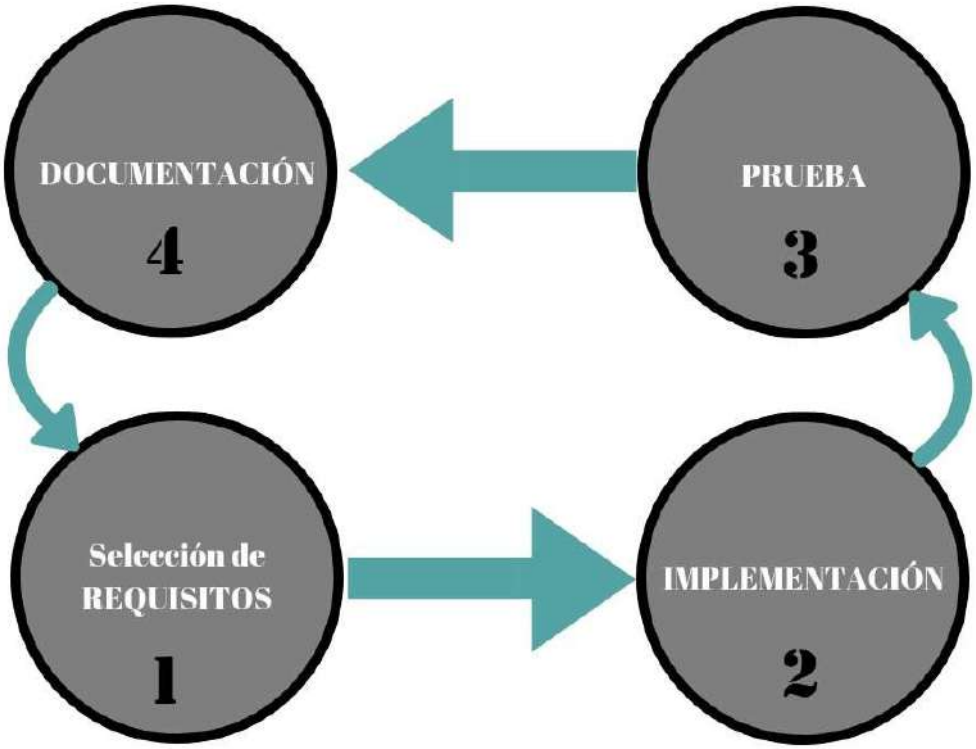
3. MODELO DE DESARROLLO.

Se usará un modelo de desarrollo incremental en espiral para garantizar un programa robusto y que no hay errores en cada iteración.

El proceso constará de 3 iteraciones:

1. Se implementará un programa sencillo en el que funcione correctamente la generación de cartas aleatorias sin repetir para un jugador.
2. Se ampliará la funcionalidad anterior mediante la inclusión de un jugador máquina sin memoria y la creación de la memoria que almacena las cartas que ya han salido y previene que salgan cartas que ya existen.
3. Se implementará una inteligencia al jugador máquina mediante la estadística.

Cada una de estas iteraciones tendrá una duración determinada de un día y cada una de ellas comprenderá el siguiente ciclo:



Así obtenemos que primero se selecciona un conjunto de requisitos, ya mencionados. Seguidamente se implementa el conjunto de requisitos, se prueba y se documenta. Si en el proceso de prueba se necesitara corregir algún elemento del producto se volverá hacia la implementación y luego se volverá a probar corregido en busca de nuevos problemas.

4. EXPLICACIÓN DE LA IMPLEMENTACIÓN.

Se va a usar el Paradigma Orientado a Objetos (POO) dado que permite construir un programa con encapsulación, escalable y fácilmente comprensible.

4.1. Clases y objetos.

Se detectaron los siguientes objetos:

- **Card:** Que almacena la información y operaciones sobre cada carta.
- **Player:** Que será el que tenga la información de cada jugador como sus puntos, y las operaciones relativas al mismo, como la suma de puntos.
- **MachinePlayer:** Es una ampliación de la clase Player que implementa la inteligencia al jugador máquina, basado en la estadística de la probabilidad.
- **Memory:** La memoria fundamental para poder almacenar las cartas que ya han salido y mantener un registro. También sirve a la máquina para implementar su inteligencia.
- **Controller:** Es el que actúa de intermediario entre las clases entre sí y con el usuario.

Todas estas clases se implementarán cada una en un único fichero a fin de localizarlas más rápidamente y para poder realizar una separación física de cada una de las clases.

Se adjunta un esquema UML al final que explica las relaciones

4.2. Patrones de desarrollo.

Dado que se necesita que el programa sea escalable y vemos que existen interacción de elementos aplicaremos el patrón MVC (Modelo Vista Controlador) con la excepción de que obviaremos la vista dado que explícitamente se indicó que no se tendría que aplicar.

Este patrón crea un intermediario llamado Controlador que hará de intermediario entre las clases.

Implementaremos el patrón Memoria (*Memory*) para poder mantener un registro de las cartas que salen y que le servirá a la máquina para implementar cierta inteligencia en sus jugadas.

También usaremos el patrón SOLID, así crearemos un programa que resista a fallos y tenga una estructura concreta. Así tenemos que se implementan los principios:

- SRP (*Single Responsibility Principle*): haciendo que cada clase y función se ocupe de una sola cosa.
- OCP (*Open/Closed Principle):* Haciendo que no se necesite modificar la clase para extender su funcionamiento, sino con la herencia se puede conseguir. Esto se ve en la clase Player y PlayerMachine.
- LSP (*Liskov Substitution Principle*): Esto hace que cada objeto se pueda sustituir por su subclase sin alterar el programa. Esto se puede ver si sustituimos en el controlador la variable del humano por la clase PlayerMachine.
- ISP (*Interface Segregation Pprinciple*): No se aplica porque no es necesario.

- DIP (*Dependency Inversion Principle*): Esto se consigue con el patrón Inyección de Dependencias en la cual las clases no crean las variables, sino que son creadas por el controlador y pasada al método correspondiente.

Implantaremos por último el patrón singleton para asegurarnos de que sólo existe un único objeto Controller.

Estos patrones se reflejan tanto en el esquema de clases o diagrama UML como en el código.

4.3. Casos de uso.

En el documento de requisitos no se aportan los siguientes casos, así que se han tomado las siguientes decisiones:

- Cuando el jugador humano y máquina empaten, pero no sea a 7'5, se toma como que el jugador humano gana la partida.
- En el caso de que el jugador humano exceda los 7'5 puntos, la máquina automáticamente gana la partida porque cualquier puntuación que saque la máquina estará más cerca de la puntuación ganadora sin pasarse.

Estos casos de uso están implementados y documentados en el código.

4.4. Estructura del proyecto.

4.4.1. Estructura física.

El proyecto contiene:

- README.txt: contiene las instrucciones para una correcta ejecución del producto.
- AF.pdf: este archivo.
- Carpeta src: Contiene los archivos del código fuente en PHP. Contiene:
    - main.php: Es el archivo que ejecuta el programa.
    - Carpeta classes: Es la que almacena todas las clases mencionadas en el apartado 4.1.

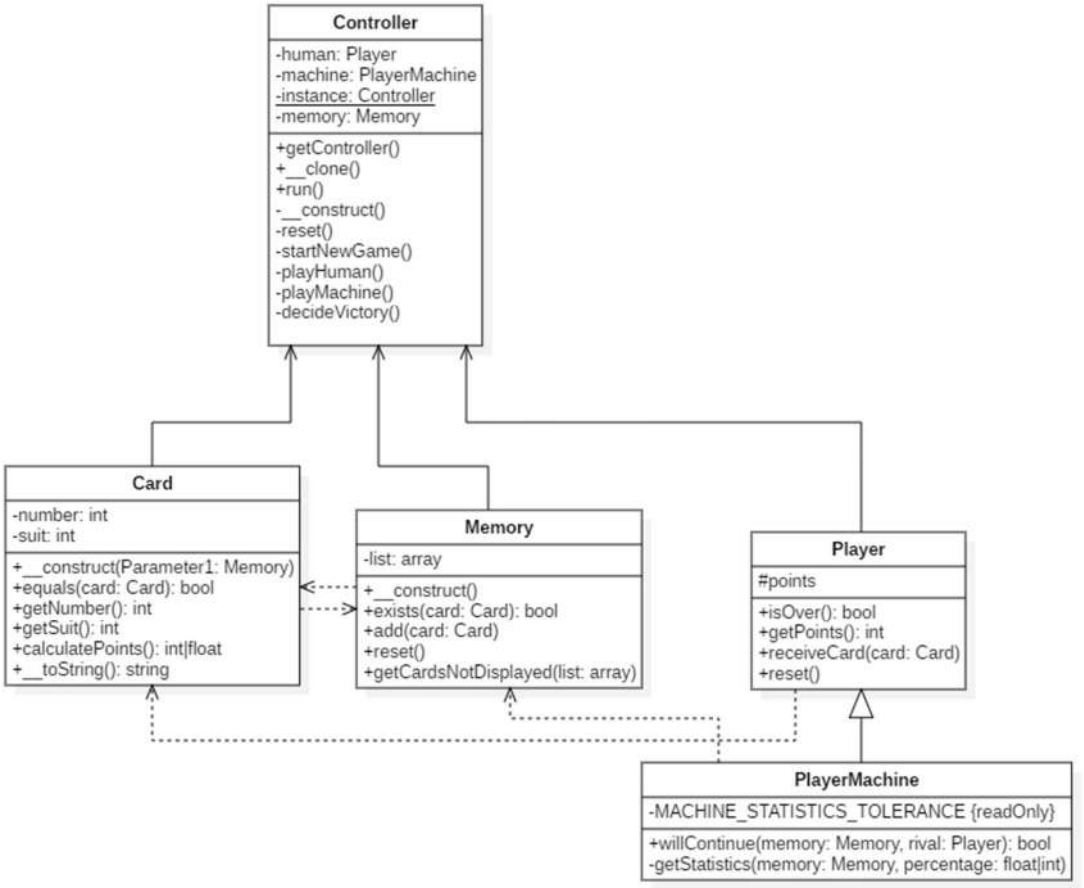
4.4.2. Estructura funcional o lógica.

Todas las clases son independientes entre sí excepto por el tipo de las clases indicadas en los métodos correspondientes. La clase Controller es el que se encarga de generar los objetos e inyectarlos en los métodos correspondientes.

5. **CONCLUSIÓN**.

A partir de este análisis técnico se construirá el programa intentando mantener la limpieza y robustez en el código y aportar la mayor documentación posible.

ANEXOS


Controller
-human: Player
-machine: PlayerMachine
-instance: Controller
-memory: Memory
+getController()
+__clone()
+run()
-__construct()
-reset()
-startNewGame()
-playHuman()
-playMachine()
-decideVictory()
Card
-number: int
-suit: int
+__construct(Parameter1: Memory)
+equals(card: Card): bool
+getNumber(): int
+getSuit(): int
+calculatePoints(): int|float
+__toString(): string
Memory
-list: array
+__construct()
+exists(card: Card): bool
+add(card: Card)
+reset()
+getCardsNotDisplayed(list: array)
Player
#points
+isOver(): bool
+getPoints(): int
+receiveCard(card: Card)
+reset()
PlayerMachine
-MACHINE_STATISTICS_TOLERANCE {readOnly}
+willContinue(memory: Memory, rival: Player): bool
-getStatistics(memory: Memory, percentage: float|int)