

LÓGICA DE PROGRAMACIÓN 2-ECC-1B

GRACE CARRASCO MORALES



UNIVERSIDAD INTERNACIONAL DEL ECUADOR

TRABAJO AUTÓNOMO 1

DOCENTE:

ESTEFANIA VANESSA HEREDIA JIMENEZ

Quito - Ecuador, Julio 2025

## Tabla de contenido

1.	Investigación:.....	3
1.2	Diagramas de Funcionalidad .....	3
	Ejemplos de Diagramas Funcionales: .....	3
1.2.1	Diagrama de Casos de Uso (Use Case Diagram - UML).....	4
1.2.2	Diagrama de Actividades (Activity Diagram - UML) .....	4
1.2.3	Diagrama de Flujo de Datos (DFD - Data Flow Diagram).....	5
1.2.4	BPMN (Business Process Model and Notation) .....	5
1.2.5	Diagrama SIPOC (Suppliers, Inputs, Process, Outputs, Customers) .....	6
1.2.6	Diagrama de flujo (Flowchart) .....	7
1.2.7	Pseudocódigo .....	8
1.3	Arquitectura de Software .....	9
1.3.1	Arquitectura Monolítica .....	9
1.3.2	Arquitectura Cliente-Servidor.....	10
1.3.3	Arquitectura por Capas (Layered Architecture) .....	10
1.3.4	Arquitectura de Microservicios .....	11
1.3.5	Arquitectura Event-Driven (Basada en Eventos) .....	12
1.3.6	Arquitectura SOA (Service-Oriented Architecture) .....	12
1.3.7	Diagrama de Componentes (Component Diagram - UML) .....	13
1.3.8	Diagrama de Despliegue (Deployment Diagram - UML) .....	13
1.3.9	Diagrama de Arquitectura por Capas (Layered Architecture) .....	13
1.3.10	Diagrama de clases (UML) .....	14
1.3.11	Modelo C4 (Contexto, Contenedores, Componentes, Código) .....	15
2.	Elección de software a desarrollar.....	15
	Juego del Ahorcado .....	15
3.	Fase de análisis: Resolución de problemas. ¿Qué se va a resolver dentro de este problema de programación? .....	15
	Objetivo del sistema.....	15
4.	Fase de diseño de las funcionalidades: detalle de lo que va a ser capaz de hacer el software en función de diagramas de uso .....	16
	Diagrama de Caso de Uso (Use Case Diagram) .....	16
5.	Fase de diseño de arquitectura de la aplicación: detalle a nivel macro de como va a trabajar el software .....	17
	Diagrama de clases (UML) .....	17
6.	Referencias bibliográficas .....	18

## Objetivo:

Selección del Programa a desarrollar / Generación de Diagramas funcionales y Arquitectura de Software

## 1. Investigación:

### 1.2 Diagramas de Funcionalidad

Los diagramas de funcionalidad son representaciones gráficas que muestran qué hace un sistema y cómo interactúan sus partes o sus usuarios para cumplir ciertos requerimientos funcionales. Su enfoque está en las funciones, procesos o casos de uso, no en cómo está construido internamente el sistema.

Su objetivo es entender qué debe hacer el sistema (funcionalidad), sin preocuparse aún por cómo se implementará.

Los diagramas funcionales muestran:

- Las acciones o funciones principales del sistema.
- Los flujos de información o de control entre funciones.
- Los actores o entidades externas que interactúan con el sistema.
- El alcance del sistema y sus requisitos funcionales.

### Ejemplos de Diagramas Funcionales:

1. Diagrama de Casos de Uso (UML):  
Muestra quién usa el sistema y qué funcionalidades ofrece.
2. Diagrama de Actividades (UML):  
Representa los flujos de trabajo o procesos del sistema.
3. Diagrama de Flujo de Datos (DFD):  
Describe cómo fluye la información dentro del sistema.
4. Mapa de procesos o BPMN:  
Representa los procesos de negocio y sus pasos.
5. Diagrama SIPOC (Suppliers-Inputs-Process-Outputs-Customers)  
Representa una visión funcional de alto nivel de un proceso
6. Diagrama de flujo (Flowchart)  
Representa la secuencia lógica de pasos funcionales de un proceso o algoritmo
7. Pseudocódigo  
Es una representación lógica y textual de un algoritmo.

### 1.2.1 Diagrama de Casos de Uso (Use Case Diagram- UML)

Muestra las relaciones entre actores (usuarios u otros sistemas) y los casos de uso (funciones o servicios que el sistema debe realizar). Este diagrama ayuda a definir el alcance funcional del sistema.

Elementos:

- Actores (usuarios, sistemas externos)
- Casos de uso (acciones o servicios del sistema)
- Relaciones (inclusión, extensión, generalización)

Ejemplo:

En un sistema de biblioteca:

- Actor: Bibliotecario
- Caso de uso: Registrar préstamo, buscar libro, generar reporte.

Función: ¿Para qué sirve?

- Definir requisitos funcionales
- Identificar quién hace qué
- Punto de partida para diseñar flujos y procesos

(Booch, G., Rumbaugh, J., & Jacobson, I., 2005)

### 1.2.2 Diagrama de Actividades (Activity Diagram- UML)

Muestra el flujo de actividades o procesos paso a paso dentro de una función, incluyendo decisiones, bifurcaciones, paralelismos y bucles. Este diagrama es ideal para detallar un proceso complejo o flujo funcional y para mostrar ramificaciones y decisiones.

Elementos:

- Actividades
- Decisiones
- Condiciones y bifurcaciones
- Inicio y fin

Ejemplo:

Proceso de compra en línea:

1. Seleccionar producto
2. Agregar al carrito
3. Verificar pago
4. Confirmar pedido

Función: ¿Para qué sirve?

- Documentar flujos de trabajo
- Analizar procesos alternativos
- Identificar cuellos de botella

(Larman, C., 2004)

### 1.2.3 Diagrama de Flujo de Datos (DFD- Data Flow Diagram)

Muestra como fluye la información dentro del sistema entre procesos, almacenes de datos y entidades externas. Sirve para un análisis estructurado, también para identificar entradas, salidas y almacenamiento de datos.

Niveles:

- Nivel 0: Vista general (Diagrama de contexto)
- Nivel 1+: Detalle de procesos internos

Elementos:

- Procesos
- Almacenes de datos
- Entidades externas
- Flujos de datos

Ejemplo:

Sistema bancario: Cliente → Solicitud de retiro → Validación → Saldo → Confirmación

Función: ¿Para qué sirve?

- Comprender cómo se mueve la información
- Modelar sistemas orientados a procesos
- Separar lógica funcional de detalles técnicos

(Pressman, R. S., 2010)

### 1.2.4 BPMN (Business Process Model and Notation)

Muestra procesos de negocio de forma estandarizada, incluyendo tareas humanas, automáticas, eventos, decisiones y subprocessos.

Elementos:

- Actividades (tareas)
- Eventos (inicio, fin, intermedios)
- Puertas de decisión (gateways)

- Swimlanes (carriles por actor/rol)

Ejemplo:

Proceso de atención en clínica:

- Registro de paciente → Evaluación médica → Entrega de receta → Facturación

Función: ¿Para qué sirve?

- Modelar procesos de negocio con precisión
- Comunicar procesos entre áreas (TI + Negocio)
- Automatizar procesos en herramientas en línea o programas.

(Object Management Group (OMG), 2011)

## 1.2.5 Diagrama SIPOC (Suppliers, Inputs, Process, Outputs, Customers)

Muestra un proceso definido en 5 componentes claves: proveedores, entradas, proceso, salidas y clientes.

Elementos:

- Suppliers (Proveedores): Personas o sistemas que suministran entradas.
- Inputs (Entradas): Datos, materiales, información que el proceso necesita.
- Process (Proceso): Serie de pasos que transforman entradas en salidas.
- Outputs (Salidas): Productos, servicios o datos generados.
- Customers (Clientes): Destinatarios o usuarios finales de las salidas.

Ejemplo:

Sistema de registro de estudiantes:

- Suppliers (Proveedores): Estudiante
- Inputs (Entradas): Datos personales
- Process (Proceso): Ingresar datos → Validar información → Guardar en base de datos
- Outputs (Salidas): Registro confirmado
- Customers (Clientes): Secretaría académica

Función: ¿Para qué sirve?

- Documentar procesos funcionales generales
- En análisis de requerimientos funcionales de negocio

(George, M. L., 2002)

## 1.2.6 Diagrama de flujo (Flowchart)

Este diagrama secuencial muestra la lógica funcional paso a paso de un proceso. Es muy usado en algoritmos y programación.

Elementos:

Los elementos o símbolos más usados se describen en la imagen a continuación.





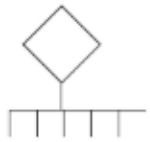









Símbolos principales	Función
	Terminal (representa el comienzo, "inicio", y el final, "fin" de un programa. Puede representar también una parada o interrupción programada que sea necesario realizar en un programa).
	Entrada/Salida (cualquier tipo de introducción de datos en la memoria desde los periféricos, "entrada", o registro de la información procesada en un periférico, "salida").
	Proceso (cualquier tipo de operación que pueda originar cambio de valor, formato o posición de la información almacenada en memoria, operaciones aritméticas, de transferencia, etcétera).
	Decisión (indica operaciones lógicas o de comparación entre datos —normalmente dos— y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa se debe seguir; normalmente tiene dos salidas —respuestas SÍ o NO— pero puede tener tres o más, según los casos).
	Decisión múltiple (en función del resultado de la comparación se seguirá uno de los diferentes caminos de acuerdo con dicho resultado).
	Conector sirve para enlazar dos partes cualesquiera de un organograma a través de un conector en la salida y otro conector en la entrada. Se refiere a la conexión en la misma página del diagrama.
	Indicador de dirección o línea de flujo (indica el sentido de ejecución de las operaciones).
	Línea conectora (sirve de unión entre dos símbolos).
	Conector (conexión entre dos puntos del organograma situado en páginas diferentes).
	Llamada a subrutina o a un proceso predeterminado (una subrutina es un módulo independientemente del programa principal, que recibe una entrada procedente de dicho programa, realiza una tarea determinada y regresa, al terminar, al programa principal).
	Pantalla (se utiliza en ocasiones en lugar del símbolo de E/S).
	Impresora (se utiliza en ocasiones en lugar del símbolo de E/S).
	Teclado (se utiliza en ocasiones en lugar del símbolo de E/S).
	Comentarios (se utiliza para añadir comentarios clasificadores a otros símbolos del diagrama de flujo. Se pueden dibujar a cualquier lado del símbolo).

Tabla de símbolos. (Aguilar, 2020)

Ejemplo:

Calcular el salario bruto y el salario neto de un trabajador (por horas) conociendo el nombre, número de horas trabajadas, impuestos a pagar y salario neto.

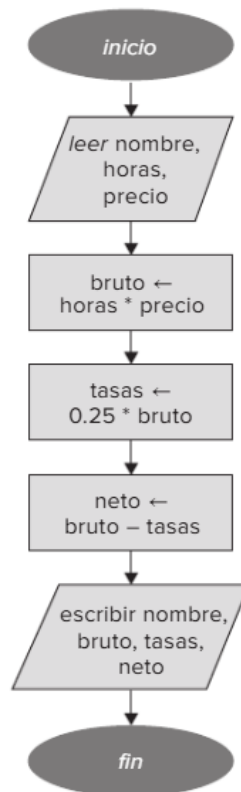


Diagrama de flujo. (Aguilar, 2020)

Función: ¿Para qué sirve?

- Para detallar funciones o procesos que siguen una lógica secuencial.

(Hoffer, J., George, J. & Valacich, J., 2013)

## 1.2.7 Pseudocódigo

Es una representación textual y lógica de un algoritmo, escrita en lenguaje natural o semi-estructurado, que describe qué hace el sistema paso a paso, sin preocuparse por el lenguaje de programación ni por cómo se implementará técnicamente.

El pseudocódigo usa palabras reservadas en inglés, similares a sus homónimas en los lenguajes de programación, para representar las acciones sucesivas, tales como start, end, stop, if-then-else, while-end, repeat-until, etcétera. La escritura de pseudocódigo exige normalmente la indentación (sangría en el margen izquierdo) de diferentes líneas.

Ejemplo:

Problema de cálculo del salario neto de un trabajador:

```

start
//cálculo de impuesto y salarios
read nombre, horas, precio
salario ← horas * precio
tasas ← 0,25 * salario
salario_netto ← salario - tasas
  
```



```
write nombre, salario, tasas, salario  
end
```

Función: ¿Para qué sirve?

- Expresar la lógica funcional de una tarea o proceso.
- Detallar cómo se ejecutan las funciones o casos de uso que luego serán implementados.
- Diseñar, analizar y explicar algoritmos antes de codificarlos.

(Aguilar, 2020)

## 1.3 Arquitectura de Software

La arquitectura de software es la estructura organizativa del sistema que define cómo se va a construir los componentes principales, cómo interactúan entre ellos, dónde se ejecutan, y qué tecnologías o patrones se usarán. Estos diagramas ayudan a visualizar la estructura técnica del software, sus componentes, dependencias y distribución.

Su objetivo es cómo funcionará internamente el sistema, considerando rendimiento, seguridad, mantenibilidad, escalabilidad, etc.

Tipos comunes de Arquitectura:

1. Monolítica
2. Cliente-Servidor
3. Capas (Layered)
4. Microservicios
5. Event-Driven
6. Arquitectura basada en servicios (SOA)

Adicionalmente, incluye diagramas de arquitectura como:

7. Diagrama de Componentes (UML)
8. Diagrama de Despliegue
9. Diagrama de Arquitectura por Capas
10. Diagrama de Clases (UML)
11. C4 Model (Contexto, Contenedores, Componentes, Código)

### 1.3.1 Arquitectura Monolítica

Es una única aplicación en la que todas las funcionalidades están unidas en un solo bloque. Todo se ejecuta junto: interfaz, lógica de negocio, acceso a datos.

Ejemplo:

Aplicaciones antiguas de escritorio como MS Office 2003 o una app web con frontend y backend embebido (PHP + MySQL).

Ventajas:

- Fácil de desarrollar al inicio
- Simplicidad en el despliegue
- Ideal para aplicaciones pequeñas

Desventajas:

- Difícil de escalar por partes
- Cualquier cambio requiere recompilar todo
- Un error puede tumbar todo el sistema

(Len Bass, Paul Clements & Rick Kazman, 2013)

### 1.3.2 Arquitectura Cliente-Servidor

Es la que divide la aplicación en dos partes:

- Cliente: Interfaz con la que interactúa el usuario
- Servidor: Procesa lógica y datos

Ejemplo:

Aplicaciones web tradicionales como Gmail, Outlook Web App, o un sistema bancario con frontend web y backend en Java.

Ventajas:

- Separación clara de roles
- Posibilidad de múltiples clientes (web, móvil)
- Seguridad centralizada

Desventajas:

- Dependencia del servidor
- Escalabilidad limitada comparado con arquitecturas distribuidas

(Sommerville, 2011)

### 1.3.3 Arquitectura por Capas (Layered Architecture)

Es el que organiza el sistema en capas jerárquicas, donde cada una tiene una responsabilidad específica. Las capas superiores usan los servicios de las inferiores.

Capas comunes:

1. Presentación (UI)
2. Lógica de Negocio
3. Acceso a Datos
4. Base de Datos

Ejemplo:

Sistema de facturación ERP o una app con Spring Boot + Thymeleaf + JPA.

Ventajas:

- Separación de responsabilidades
- Fácil de probar, mantener y extender
- Arquitectura ampliamente adoptada

Desventajas:

- Puede volverse rígida si no se diseña bien
- Las capas deben respetarse estrictamente

(Fowler, 2002)

## 1.3.4 Arquitectura de Microservicios

El sistema se divide en múltiples servicios independientes, cada uno con una función específica y desplegable de manera aislada, es decir, cada uno con su propia lógica y base de datos.

Ejemplo:

Netflix, Amazon, Spotify, Uber (cada microservicio maneja usuarios, pagos, búsquedas, etc.)

Ventajas:

- Escalabilidad independiente
- Despliegue continuo
- Resiliencia (si un servicio falla, el resto sigue funcionando)

Desventajas:

- Complejidad en la orquestación
- Necesita infraestructura avanzada (Docker, Kubernetes, etc.)
- Comunicación entre servicios puede fallar

(Newman, 2015)

### 1.3.5 Arquitectura Event-Driven (Basada en Eventos)

Los componentes se comunican mediante eventos (por ejemplo: "pedido creado", "usuario registrado"). Un componente produce eventos y otros los consumen.

Ejemplo:

Una tienda online que, al recibir un pedido, dispara eventos para facturación, envío, inventario, etc.

Ventajas:

- Desacoplamiento total entre servicios
- Muy flexible y escalable
- Reaccionan en tiempo real

Desventajas:

- Dificultad para depurar errores
- Eventos perdidos pueden causar fallos silenciosos
- No siempre es fácil de testear

(Richards, 2022)

### 1.3.6 Arquitectura SOA (Service-Oriented Architecture)

Es una arquitectura orientada a servicios reutilizables, donde cada uno expone una interfaz estándar (usualmente vía SOAP o REST).

Diferencia con microservicios:  
SOA suele usar servicios más grandes y acoplados, con ESB (Enterprise Service Bus) como middleware.

Microservicios son más pequeños, autónomos y ligeros.

Ejemplo:

Gobiernos o grandes bancos que integran servicios de registro civil, impuestos, pagos, etc.

Ventajas:

- Reutilización de servicios
- Estándares abiertos (WSDL, XML, SOAP)
- Buena integración entre sistemas legados

Desventajas:

- Más pesado que microservicios
- ESB puede ser un punto de falla
- Curva de aprendizaje alta

(Thomas, 2008)

### 1.3.7 Diagrama de Componentes (Component Diagram- UML)

Muestra los módulos o componentes del software (clases, servicios, bibliotecas), y cómo se relacionan entre sí.

Ejemplo:

Sistema web:

- Componente: Módulo de Autenticación
- Componente: Módulo de Reportes
- Relación: Ambos usan Base de Datos

Estructura técnica: ¿Para qué sirve?

- Visualizar modularización
- Definir responsabilidades técnicas
- Ayudar en decisiones de empaquetado y reutilización

(Larman, C., 2004)

### 1.3.8 Diagrama de Despliegue (Deployment Diagram- UML)

Representa la infraestructura física (servidores, nodos, contenedores) donde se ejecutan los componentes del software. Muestra cómo se distribuye el sistema en hardware.

Ejemplo:

App móvil conectada a backend:

- Nodo 1: Servidor web
- Nodo 2: Servidor de base de datos
- Nodo 3: Dispositivo móvil

Estructura técnica: ¿Para qué sirve?

- Visualizar la infraestructura
- Planificar entornos de producción
- Determinar necesidades de red
- Escalabilidad del sistema

(Mark Richards, Neal Ford, 2020)

### 1.3.9 Diagrama de Arquitectura por Capas (Layered Architecture)

Muestra la separación lógica de la aplicación en capas, típicamente:

- Presentación

- Lógica de negocio
- Acceso a datos
- Base de datos

Ejemplo:

Sistema bancario:

- Capa 1: Interfaz gráfica
- Capa 2: Validación y reglas
- Capa 3: Consultas SQL

Estructura técnica: ¿Para qué sirve?

- Dividir el sistema en capas o módulos
- Mantener separación de responsabilidades
- Facilitar pruebas y mantenimiento

(David West, Brett McLaughlin, 2007)

### 1.3.10 Diagrama de clases (UML)

Es un diagrama UML estructural que representa las clases de un sistema de software, junto con sus:

- Atributos (propiedades o variables)
- Métodos (funciones o comportamientos)
- Relaciones entre clases (como herencia, asociación, agregación o composición)

Es como ver el esqueleto lógico del sistema: qué cosas hay, cómo se comportan y cómo se relacionan entre sí.

Estructura técnica: ¿Para qué sirve?

- Diseñar la lógica de negocio: Se puede planificar qué objetos (clases) se necesita, qué deben saber y qué deben hacer.
- Identificar relaciones y responsabilidades: ¿Qué clase se encarga de qué? ¿Qué clases trabajan juntas?
- Documentar el sistema: Sirve para explicar la estructura del sistema a otros desarrolladores, testers, docentes o clientes.
- Base para la codificación: De este diagrama se puede obtener directamente el código en Java, Python, C#, etc.
- Detección temprana de errores: Permite ver errores de diseño antes de escribir código (por ejemplo, clases acopladas o responsabilidades mal distribuidas).

Elementos clave:

Elemento	Significado
Clase	Una entidad que agrupa atributos y métodos. Se representa como un rectángulo dividido en 3 secciones.
Atributos	Características o datos que tiene la clase.
Métodos	Funciones que puede realizar una clase.
Relaciones: 1. Asociación 2. Herencia (Generalización) 3. Composición 4. Agregación	Cómo se conectan las clases: 1. Una clase se relaciona con otra. 2. Una clase hija hereda de una clase padre. 3. Una clase contiene a otra, y si se destruye, también la interna. 4. Similar a la composición, pero más débil (la clase contenida puede vivir por separado).

Tabla de elementos para diagramas de clase.

(Ian Sommerville, 2015)

### 1.3.11 Modelo C4 (Contexto, Contenedores, Componentes, Código)

Muestra una jerarquía de vistas para representar el sistema desde lo general hasta lo específico a través de 4 diagramas:

1. Diagrama de contexto (sistema y su entorno)
2. Diagrama de contenedores (apps, bases de datos, etc.)
3. Diagrama de componentes internos
4. Diagrama de código (opcional, para detalle)

Estructura técnica: ¿Para qué sirve?

- Comunicar la arquitectura a distintos niveles
- Estandarizar la documentación

(Brown, 2014)

## 2. Elección de software a desarrollar

Juego del Ahorcado

### 3. Fase de análisis: Resolución de problemas. ¿Qué se va a resolver dentro de este problema de programación?

Objetivo del sistema

Desarrollar un juego donde el usuario debe adivinar una palabra secreta letra por letra, con un número limitado de intentos.

#### 4. Fase de diseño de las funcionalidades: detalle de lo que va a ser capaz de hacer el software en función de diagramas de uso

##### Diagrama de Caso de Uso (Use Case Diagram)

Con este diagrama se mostrará que puede hacer el usuario y que funcionalidades ofrecerá el sistema. La intención es entender los requisitos funcionales.

Actor principal:

- Jugador (Usuario)

Casos de uso:

1. Iniciar juego
2. Elegir dificultad (por tipos de palabras o número de letras que contiene)
3. Ingresar letra
4. Ver estado del juego (palabra oculta, intentos restantes)
5. Terminar juego (por victoria o derrota)
6. Reiniciar juego (solo si se desea seguir jugando)

Relaciones:

- El jugador interactúa con todos los casos.
- "Ver estado del juego" se actualiza cada vez que se ingresa una letra.

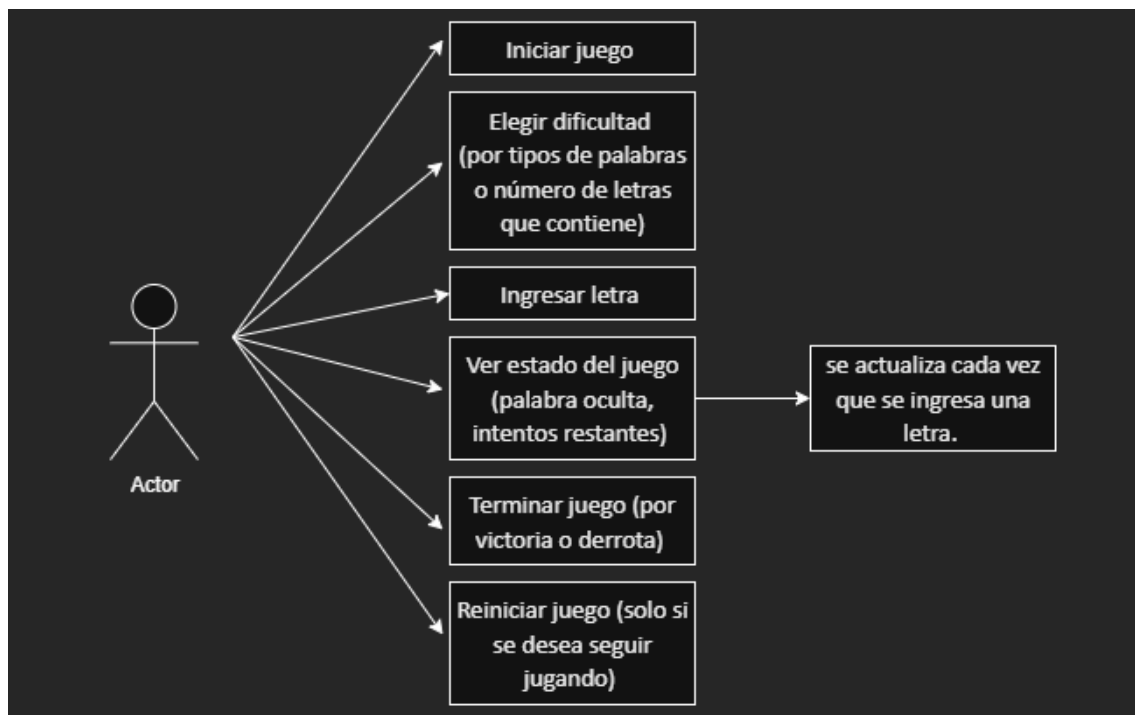


Diagrama de caso de uso. Elaboración propia.



## 5. Fase de diseño de arquitectura de la aplicación: detalle a nivel macro de como va a trabajar el software

### Diagrama de clases (UML)

La idea con este diagrama permite modelar las estructuras de datos y clases del juego, incluyendo atributos y métodos. Es fundamental para definir objetos como:

- Juego
- Jugador
- Palabra
- Letra
- Interfaz

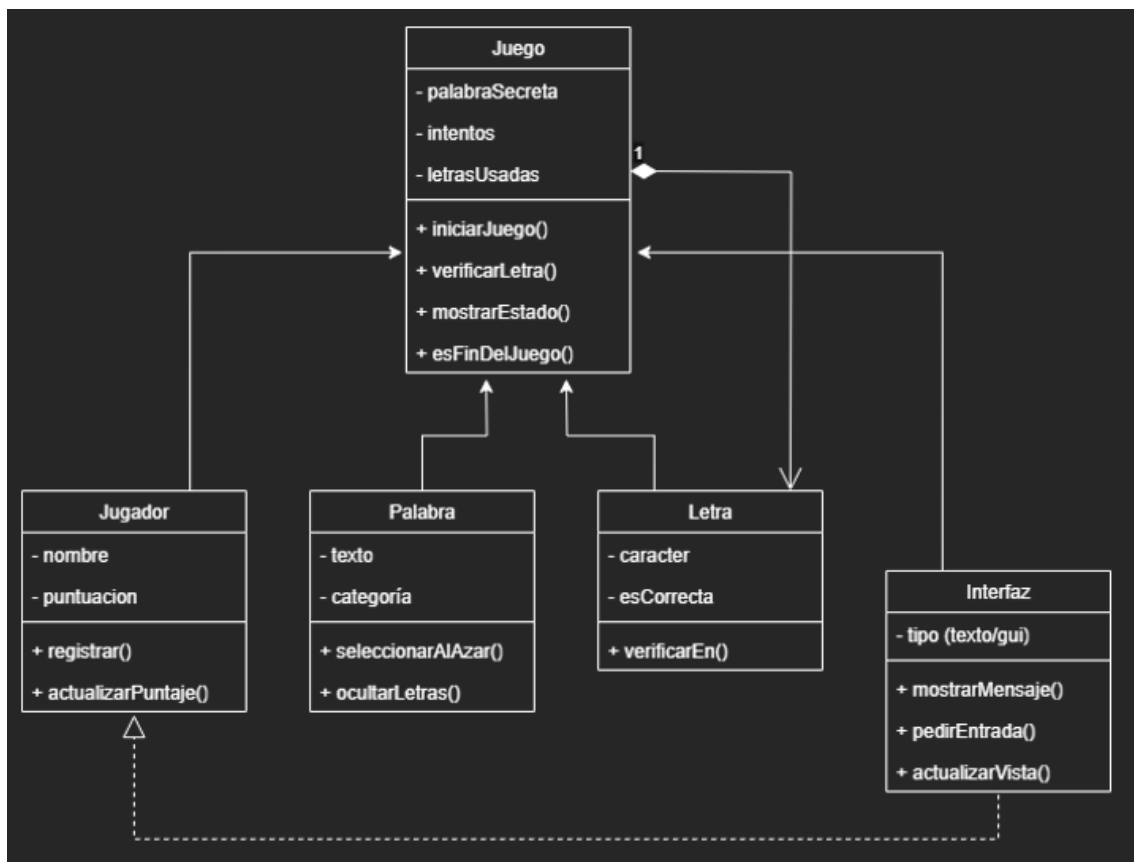


Diagrama de clase. Elaboración propia.

Relaciones:

- Juego usa Jugador, Palabra, Letra e Interfaz.
- Juego contiene la lógica principal y coordina el resto de las clases.
- Interfaz se encarga de la entrada/salida (puede ser consola o gráfica).
- Letra representa cada intento del jugador.
- Palabra maneja la palabra secreta y su procesamiento.

## 6. Referencias bibliográficas

- Aguilar, L. J. (2020). Fundamentos de programación: Algoritmos, estructura de datos y objetos. McGraw Hill Interamericana.
- Booch, G., Rumbaugh, J., & Jacobson, I. (2005). The Unified Modeling Language User Guide (2da ed.). Addison-Wesley.
- Brown, S. (2014). *Software Architecture for Developers*.
- David West, Brett McLaughlin. (2007). *Head First Object-Oriented Analysis and Design*. O'reilly & Associates.
- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Signature Series.
- George, M. L. (2002). Lean Six Sigma: Combining Six Sigma Quality with Lean Speed. McGraw-Hill.
- Hoffer, J., George, J. & Valacich, J. (2013). Modern Systems Analysis and Design. Pearson.
- Ian Sommerville. (2015). *Ingeniería de Software (décima edición)*. Pearson.
- Larman, C. (2004). Applying UML and Patterns (3ra ed.). Prentice Hall.
- Len Bass, Paul Clements & Rick Kazman. (2013). *Software Architecture in Practice (3ª edición)*.
- Mark Richards, Neal Ford. (2020). *Fundamentals of Software Architecture*. O'Reilly Media.
- Newman, S. (2015). *Building Microservices*. O'Reilly Media.
- Object Management Group (OMG). (2011). *BPMN 2.0 Specification*.
- Pressman, R. S. (2010). Software Engineering: A Practitioner's Approach (7ma ed.). McGraw-Hill.
- Richards, M. (2022). *Software Architecture Patterns, 2nd Edition*. O'Reilly Media.
- Sommerville, I. (2011). *Ingeniería del Software (novena edición)*. Pearson.
- Thomas, E. (2008). *SOA: Principles of Service Design*. Prentice Hall, Upper Saddle River, NJ.