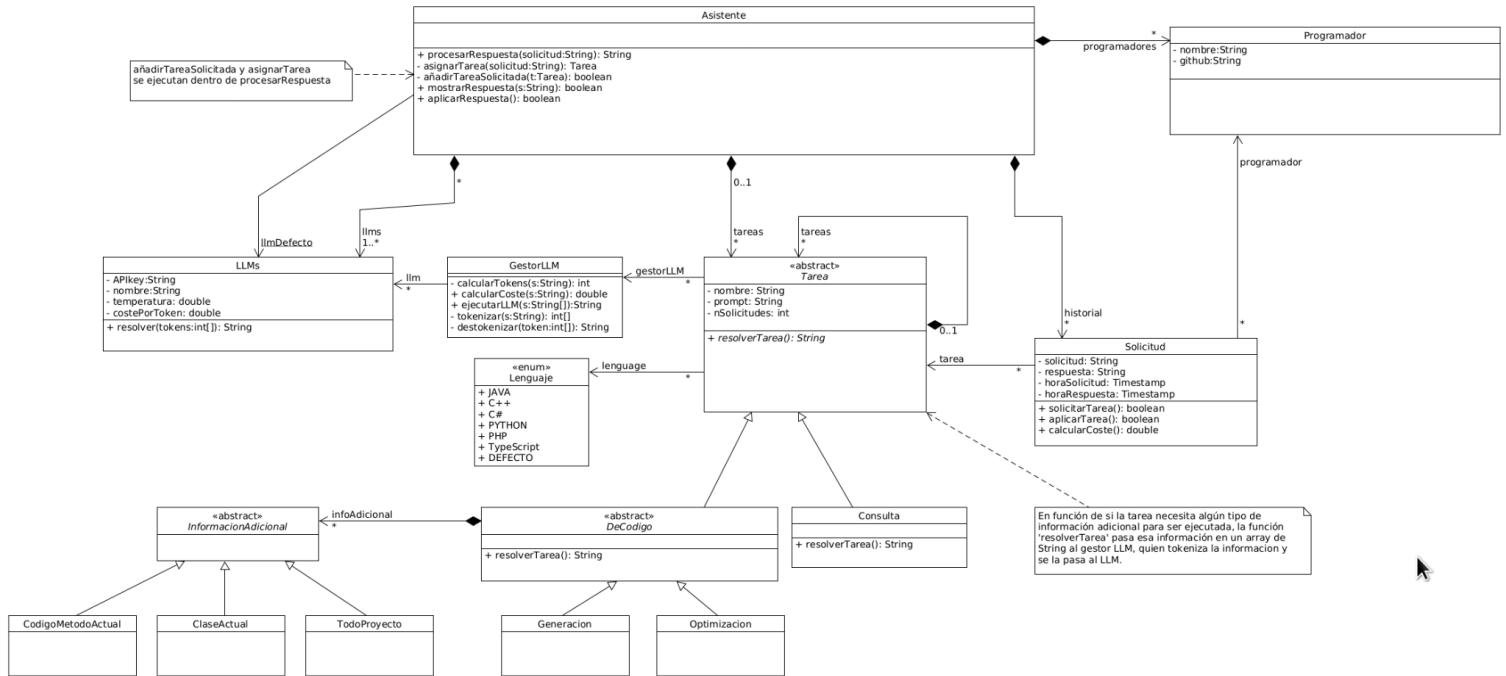


Apartado 2:

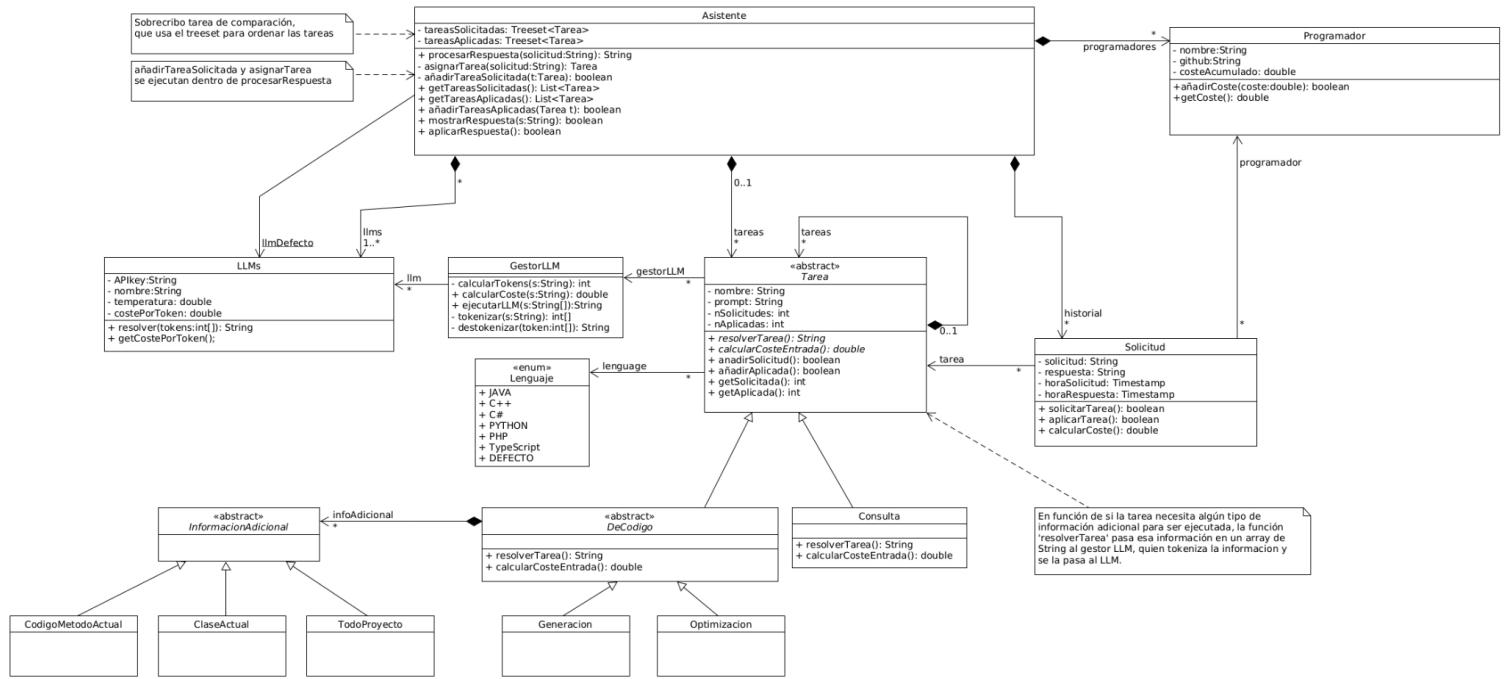
a. Diagrama de clases.



El diseño propuesto para el asistente conversacional se basa en una estructura de herencia y composición que permite gestionar tareas de forma extensible. La clase abstracta **Tarea** centraliza los elementos comunes como el prompt y el lenguaje (gestionado mediante un enum **Lenguaje**), mientras que sus subclases especializan el comportamiento: **DeCodigo** (para generación y optimización) e **InformacionAdicional** para manejar el contexto del proyecto. Destaca la implementación del patrón Composite en la relación recursiva de **Tarea**, que permite crear tareas compuestas.

La clase **Asistente** actúa como núcleo del sistema. La eficiencia en el acceso a los datos se garantiza mediante el uso de estructuras de datos específicas, como podría ser un Treeset para el ordenamiento automático de tareas o un Map para la gestión rápida de suscriptores o programadores. Por último, la clase **Solicitud** vincula a cada programador con la tarea ejecutada y el LLM utilizado, registrando los timestamps y permitiendo el cálculo de costes basado en el consumo de tokens.

b. Diagrama de clases ampliado.



c. Pseudocódigo de todos los métodos en el cálculo del coste de una ejecución de una tarea.

Funciones GestorLLM:

CLASE GestorLLM

```
...  
// Calcula el coste económico en función de la cadena de texto introducida  
FUNCION calcularCoste(texto: String): double
```

```
VARIABLE tokens: entero  
VARIABLE coste: double
```

```
tokens <- llamar calcularTokens(texto)  
coste <- tokens * llm.getCostePorToken()
```

DEVOLVER coste

FIN FUNCION

```
// Algoritmo de tokenización basado en la longitud de las palabras del texto  
FUNCION calcularTokens(texto: String): int
```

```
VARIABLE totalTokens: entero
```

```
// El cálculo se realiza en función de la longitud de las palabras y el String  
totalTokens <- procesarTexto(texto)
```

DEVOLVER totalTokens

FIN FUNCION

...
FIN CLASE

Funciones Tarea:

CLASE ABSTRACTA Tarea

...
// Método abstracto: cada subclase decidirá cómo calcular el coste de su entrada
PROTOTIPO FUNCION calcularCosteEntrada(): double

...
FIN CLASE

CLASE Consulta EXTENDS Tarea

...
// Implementación específica para consultas simples
FUNCION calcularCosteEntrada(): double
 // Solo calculamos el coste del prompt
 DEVOLVER gestorLLM.calcularCoste(getPrompt())
FIN FUNCION

...
FIN CLASE

CLASE ABSTRACTA DeCodigo EXTENDS Tarea

...
// Implementación específica para tareas de código
FUNCION calcularCosteEntrada(): double
 VARIABLE costePrompt, costeInfo: double

 // Coste del prompt
 costePrompt ← gestorLLM.calcularCoste(getPrompt())
 // Coste de la información adicional necesaria
 costeInfo ← gestorLLM.calcularCoste(infoAdicional.getInfo())

 DEVOLVER (costePrompt + costeInfo)
FIN FUNCION

...
FIN CLASE

Funciones Solicitud:

CLASE Solicitud

...
// Calcula el sumatorio de costes de la tarea ejecutada
FUNCION calcularCosteTotal(): REAL
 VARIABLE costeTarea, costeRespuesta: double

 costeInfo <- 0 // Inicialización por defecto
 // Coste de la tarea
 costeTarea <- tarea.calcularCosteEntrada()
 // Coste de la respuesta generada
 costeRespuesta <- tarea.gestorLLM.calcularCoste(respuesta)

 DEVOLVER (costePrompt + costeTarea)
FIN FUNCION

...
FIN CLASE