Universidad del Valle de Guatemala

Algoritmos y Estructuras de Datos

Profesor: Sebastián Arriola Bethancourt

Proyecto Fase 1

Kevin Alexander
Moreno Martínez
#25730

Daniel Alejandro
Hernández Silvestre
#25054

Iván David
Roblero Mejía
#25238

Diseño e Implementación de la Infraestructura Base del Intérprete

Durante la primera fase del proyecto se establecieron los fundamentos estructurales del intérprete, enfocándose en la construcción de la base sobre la cual se ejecutarán las siguientes funcionalidades del sistema. Esta etapa no se centró en la lógica completa de evaluación, sino en preparar los componentes esenciales que permiten que el intérprete funcione de manera organizada y escalable.

En primer lugar, se definió la arquitectura general del proyecto mediante la separación del código en paquetes específicos. Esta decisión permitió dividir responsabilidades entre los distintos componentes del sistema, evitando dependencias innecesarias y mejorando la claridad estructural. Cada paquete agrupa clases relacionadas según su función, lo que facilita la lectura, el mantenimiento y la ampliación futura del intérprete.

Uno de los elementos principales desarrollados fue la estructura de la pila. Dado que Bitcoin Script opera exclusivamente sobre un modelo basado en stack, era necesario contar con una implementación sólida y eficiente desde el inicio. Para ello se utilizó la interfaz Deque del Java Collections Framework. Esta estructura permite realizar inserciones y extracciones en tiempo constante O(1), lo cual resulta adecuado para un sistema que depende continuamente de operaciones sobre la parte superior de la pila.

La implementación incluyó métodos para insertar datos, retirar elementos, consultar el valor superior y verificar si la pila está vacía. Además, se contemplaron validaciones básicas para evitar estados inconsistentes, como intentar extraer elementos cuando la pila no contiene datos.

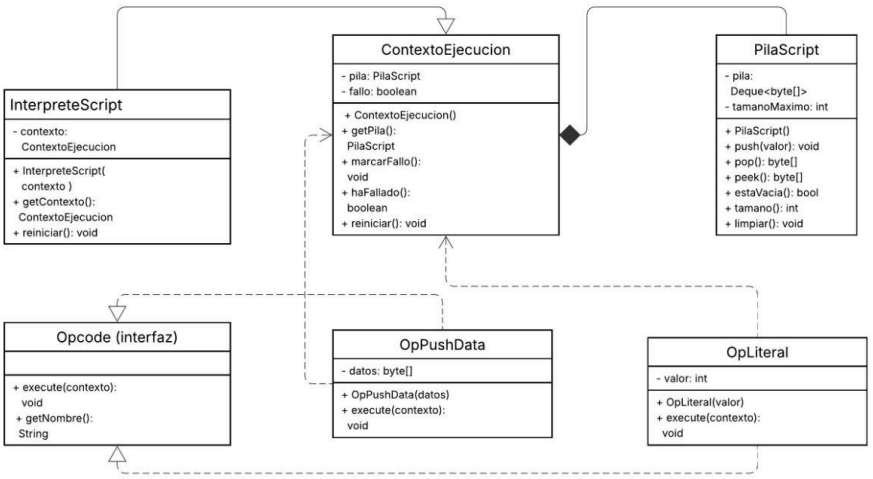
También se diseñó la clase encargada de representar el contexto de ejecución. Esta clase centraliza el estado del programa mientras se encuentra activo, almacenando la pila principal y una variable de control que permite indicar si la ejecución debe considerarse fallida. Separar el estado del resto de componentes ayuda a mantener un diseño más limpio y reduce el acoplamiento entre clases.

Con el objetivo de establecer una estructura uniforme para todas las instrucciones del sistema, se definió una interfaz común para los opcodes. Esta interfaz obliga a que cada instrucción implemente el mismo método de ejecución, garantizando coherencia en la forma en que serán procesadas posteriormente. Esta decisión de diseño favorece la extensibilidad, ya que nuevas instrucciones pueden añadirse sin alterar la base del intérprete.

Como parte del prototipo mínimo requerido en esta fase, se implementaron los opcodes literales, incluyendo OP_0, OP_1 hasta OP_16 y la instrucción PUSHDATA. Estas operaciones permiten insertar valores directamente en la pila, sentando las bases para que el motor de ejecución pueda operar sobre datos reales en etapas posteriores del desarrollo.

Adicionalmente, se elaboraron los diagramas UML correspondientes a esta fase inicial, los cuales describen la relación entre la pila, el contexto de ejecución y la interfaz de

instrucciones. Estos diagramas permitieron visualizar la arquitectura antes de que el sistema estuviera completamente integrado, asegurando coherencia en el diseño general.



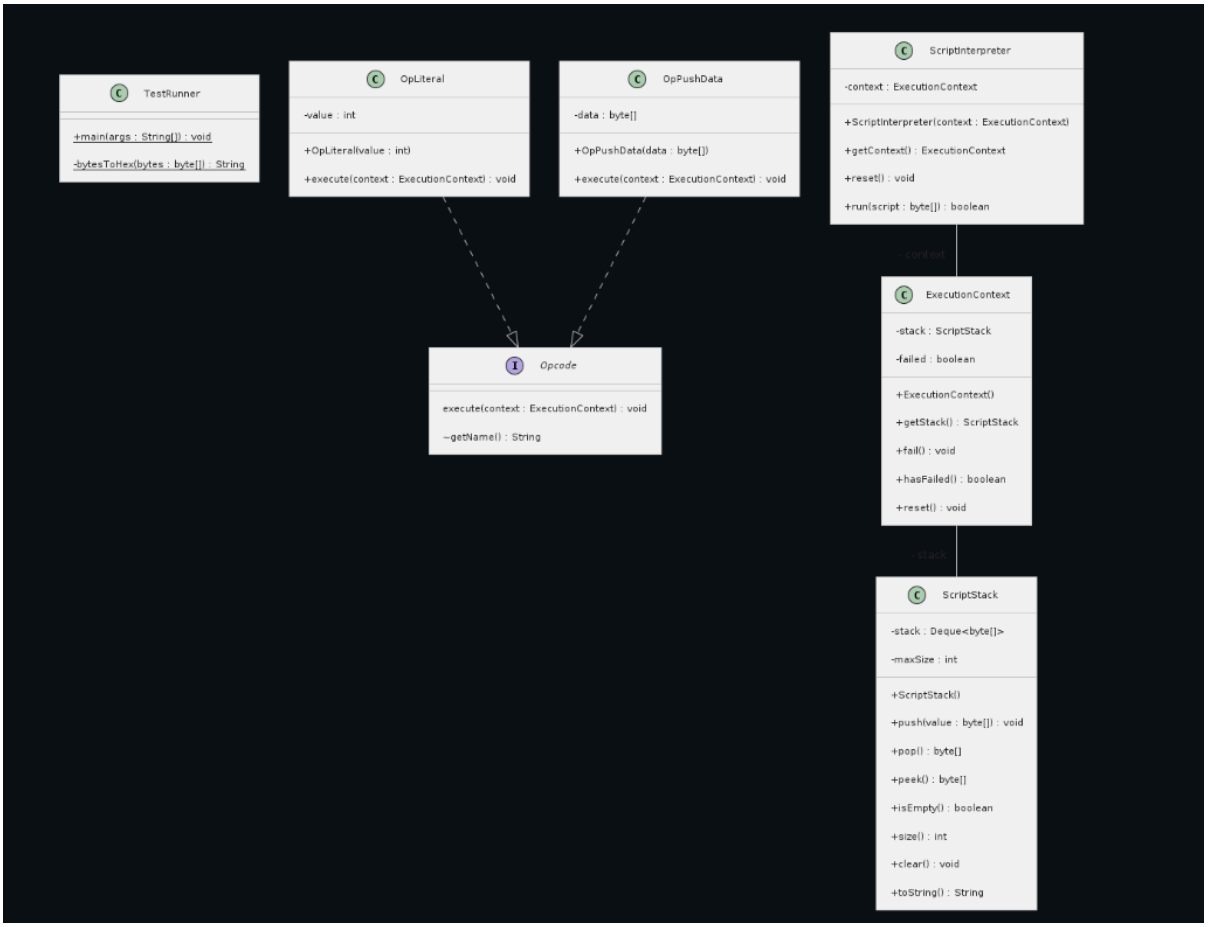
Desarrollo del Motor de Ejecución y Estructura de la Pila:

Se desarrolló el núcleo del intérprete mediante la implementación del motor de ejecución, responsable de procesar y evaluar las instrucciones en un orden secuencial de izquierda a derecha. Para ello, se diseñó y programó el ciclo principal de evaluación, que permite recorrer y ejecutar cada instrucción de manera controlada. Además, se incorporó un manejo de errores básicos, con el objetivo de detectar operaciones inválidas, estados incorrectos de la pila o fallos en las validaciones, garantizando así un comportamiento más robusto del sistema.

En el plano funcional, se implementaron los opcodes de manipulación de pila OP_DUP y OP_DROP, que permiten respectivamente duplicar el elemento superior y eliminarlo de la estructura. Asimismo, se desarrollaron las operaciones OP_EQUAL y OP_EQUALVERIFY, orientadas a realizar comparaciones entre elementos y validar condiciones dentro del flujo de

ejecución. Estas instrucciones fueron integradas directamente al motor, asegurando su correcta interacción con la pila como estructura de datos principal del intérprete. El avance del desarrollo se evidenció mediante commits frecuentes en el repositorio, documentando la implementación progresiva del ciclo de evaluación y de los distintos opcodes.

En cuanto al diseño del sistema, se elaboró el diagrama UML de clases que define la arquitectura del intérprete y las relaciones entre sus componentes. Este trabajo permitió establecer una estructura clara y modular, facilitando la comprensión del funcionamiento interno y promoviendo una mejor organización del código. Además, se participó en la revisión del diseño junto al equipo, contribuyendo a optimizar la coherencia y mantenibilidad de la solución propuesta.



Opcodes Criptográficos Simulados

OP_HASH160

Lo que hace este opcode es sacar el elemento de hasta arriba de la pila, le aplica un hash y mete el resultado de vuelta. En Bitcoin real se usa RIPEMD-160 sobre SHA-256, eso da un resultado de 20 bytes que es básicamente la dirección de Bitcoin.

El problema es que Java no trae RIPEMD-160, así que lo que hice fue usar SHA-256 que sí viene con MessageDigest y agarré los primeros 20 bytes del resultado. No es el hash real pero sirve para lo que necesitamos: dado el mismo input siempre da el mismo output de 20 bytes, y eso es lo que importa para que después la comparación con OP_EQUALVERIFY funcione bien.

El método simulatedHash160() lo dejé como público y estático porque lo necesitamos desde el TestRunner para calcular de antemano el hash de la clave pública y ponerlo en el scriptPubKey.

OP_CHECKSIG

Este opcode saca dos cosas de la pila: primero la clave pública (que está en la cima) y luego la firma (que queda abajo). En Bitcoin real verifica la firma digital con ECDSA contra el hash de la transacción.

Como no tenemos transacciones reales, hice una verificación mock: si la firma y la clave pública tienen al menos un byte (no están vacías), se pone un 1 en la pila. Si alguna está vacía se pone un 0. Es simple pero sirve para probar todo el flujo de P2PKH sin complicarnos con criptografía.

Algo que me costó entender al principio es el orden del pop: primero sale la clave pública y después la firma. Esto es porque en el script la firma se empuja primero (queda abajo) y la pubKey después (queda arriba).

Prueba de pago P2PKH

P2PKH (Pay-to-Public-Key-Hash) es el tipo de transacción más común en Bitcoin. El script completo es la unión del scriptSig y el scriptPubKey:

<firma> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

Caso correcto

En el TestRunner armamos una prueba con datos simulados: una firma de 4 bytes y una pubKey de 5 bytes. Calculamos el hash de la pubKey con simulatedHash160() y lo puse como el pubKeyHash del script. Cuando se ejecuta, los hashes coinciden, EQUALVERIFY pasa, CHECKSIG verifica que no estén vacíos y devuelve 1. Transacción válida.

Caso incorrecto

También hicimos una prueba con un hash equivocado. Cuando se ejecuta, al llegar a OP_EQUALVERIFY los hashes no coinciden y el script falla de inmediato. No se llega ni a ejecutar OP_CHECKSIG. Transacción inválida.

Diagrama de secuencia: evaluación P2PKH

El flujo de ejecución va así:

1. El TestRunner crea el script en bytes y se lo pasa a ScriptInterpreter
2. ScriptInterpreter lee byte por byte de izquierda a derecha
3. Por cada byte identifica qué opcode es por su código hexadecimal
4. Crea la instancia del opcode correspondiente (ej: new OpDup())
5. Le pasa el ExecutionContext al método execute()
6. El opcode manipula la pila que está dentro del ExecutionContext
7. Si algún opcode falla (como EQUALVERIFY con hashes distintos), marca context.fail() y el intérprete se detiene
8. Al terminar, se revisa si la cima de la pila es verdadera con validateResult()

Descripción de la arquitectura

El código está dividido en tres paquetes:

- stack: tiene la clase ScriptStack que es básicamente un wrapper de ArrayDeque. Tiene push, pop, peek, y valida cosas como que la pila no esté vacía antes de hacer pop o que no se pase de 1000 elementos.
- opcodes: tiene la interfaz Opcode con el método execute(ExecutionContext) y una clase por cada operación. Yo hice OpHash160 y OpCheckSig, mi compañero hizo OpDup, OpDrop, OpEqual y OpEqualVerify. Cada clase es independiente, lo cual hace que agregar opcodes nuevos en la Fase 2 sea fácil.
- interpreter: tiene ExecutionContext que guarda la pila y una bandera de si el script falló, ScriptInterpreter que es el loop principal que lee los bytes y va llamando a cada opcode, y TestRunner que tiene las pruebas.