**Diamond Pattern Implementation for TokenSale Contracts**

**Overview**

This project refactors the original monolithic smart contracts into a Diamond pattern implementation to address the contract size limitations in Ethereum. The Diamond pattern allows us to split functionality into separate "facets" while maintaining a single entry point.

**Problem Addressed**

The original implementation encountered the CreateInitCodeSizeLimit error during deployment because the contracts were too large. The Diamond pattern solves this by:

1. Separating functionality into smaller, focused contracts (facets)
2. Using a proxy pattern to route function calls to the appropriate facet
3. Sharing contract storage across facets using a diamond storage pattern
4. Allowing for upgradability and extensibility

**Architecture**

**Core Diamond Infrastructure**

* **Diamond.sol**: The main proxy contract that routes function calls to facets
* **DiamondCutFacet.sol**: Handles adding, replacing, and removing facets
* **DiamondLoupeFacet.sol**: Provides introspection functions to query facet information
* **OwnershipFacet.sol**: Manages ownership of the diamond

**Token Facets**

Each token type has been refactored into its own facet:

* **BondTokenFacet.sol**: Implements bond token functionality
* **WarrantTokenFacet.sol**: Implements warrant token functionality
* **EquityTokenFacet.sol**: Implements equity token functionality
* **TokenSaleFacet.sol**: Handles token sale functionality

**Storage Patterns**

Instead of inheritance, the Diamond pattern uses a shared storage pattern:

* Each facet defines its own storage structure
* Storage slots are determined by a unique hash to avoid collisions
* Facets access shared storage through accessor functions

**Deployment Process**

The deployment process involves:

1. Deploy Diamond contracts for each token and the token sale
2. Deploy facet contracts
3. Add facets to each Diamond contract with appropriate function selectors
4. Initialize each facet with appropriate parameters
5. Configure cross-contract references

A detailed deployment script is provided in deployment-steps.js.

**Storage Structures**

**Diamond Storage**

struct DiamondState {

// Maps function selector to the facet address and position in facetFunctionSelectors

mapping(bytes4 => FacetAddressAndPosition) selectorToFacetAndPosition;

// Maps facet addresses to function selectors

mapping(address => FacetFunctionSelectors) facetFunctionSelectors;

// Facet addresses

address[] facetAddresses;

}

**Token Storage**

Each token facet defines its own ERC20 storage and token-specific storage:

// ERC20 storage (example)

struct ERC20Storage {

mapping(address => uint256) balances;

mapping(address => mapping(address => uint256)) allowances;

uint256 totalSupply;

string name;

string symbol;

uint8 decimals;

}

// Token-specific storage (Bond example)

struct BondStorage {

uint256 issuanceTimestamp;

uint256 maturityPeriod;

uint256 yieldBasisPoints;

mapping(address => uint256) bondPurchaseTimestamp;

mapping(address => uint256) bondAmounts;

address[] bondHolders;

}

**Interaction Between Diamonds**

For cross-contract interactions, the TokenSaleFacet makes external calls to the token diamonds:

// Example of TokenSaleFacet calling BondTokenFacet

(bool success, ) = ts.bondTokenDiamond.call(

abi.encodeWithSignature("mint(address,uint256)", msg.sender, bondAmount)

);

require(success, "Bond token minting failed");

**Usage Instructions**

1. Deploy the contracts following the deployment guide
2. Interact with the main TokenSale diamond contract, which will route calls appropriately
3. For direct token operations, interact with the respective token diamond contracts

**Advantages of this Implementation**

1. **Contract Size Limitation Solved**: Functionality is split across multiple facets
2. **Upgradability**: New functionality can be added without redeploying everything
3. **Modularity**: Each facet handles a specific part of the functionality
4. **Shared State**: All facets can access the same storage
5. **Single Point of Interaction**: Users still interact with a single contract address

**Security Considerations**

1. Owner access control is handled at the Diamond level
2. Storage access is protected by using unique storage positions
3. Cross-contract calls verify success to ensure atomicity
4. Each diamond maintains its own access control

**Next Steps**

1. Add extensive test suite for each facet
2. Implement event handling and indexing
3. Add administrative dashboard for managing the system
4. Consider implementing EIP-2535 compliant upgrades for future changes