Example “Wallet.sol”

Part 1: Spending Limit

In a first step, enable the owner (and only the owner) of the contract to set the spending limit for any address

function setSpendingLimit(address party, uint limit);

Part 2: Spending

Implement the spend function. It sends **value** wei to **destination**. Value has to be within the spending limit, which is decreased afterwards. For the owner the spending limit is ignored.
Also make sure you can deposit into the Wallet.

function spend(address destination, uint value);

Part 3: Requests

The Wallet has a whitelist, controlled by the owner.
Make sure whitelist, owner and spendingLimits are all public!
People on the whitelist can make requests for transfers. A request has a sender, a destination, a value in wei, an ID and a RequestState. The possible states are PENDING, APPROVED, REJECTED, EXECUTED.

function setWhitelisted(address party, bool value);
Can only be called by the owner. Sets the whitelist status of party.

function request(address destination, uint value);
This creates a new request. Fires the RequestAdded event. Id should increase by one. Initial state is PENDING.

function approve(uint id, address destination, uint value);
Can only be called by the owner. Checks destination and value. State changes to APPROVED.

function reject(uint id, address destination, uint value);
Can only be called by the owner. Checks destination and value. State changes to REJECTED.

function execute(uint id);
Can only be called by the sender. State needs to be APPROVED and becomes EXECUTED. Ether are sent out to the destination.

event Deposit(address indexed depositor, uint value);
Fired whenever ether is received.

event Spent(address indexed sender, address indexed destination, uint value);
Fired after every spend()

event RequestAdded(uint id, address indexed sender);
Fired when a when a new request has been added. The only way to get the id.

event RequestUpdate(uint indexed id, RequestState state);
Fired every time when the request state changes (not necessary for “change” to PENDING)

Part 4: Tokens, Inheritance and Time

Reuse the Whitelist contract from the Market example!

A Request takes two additional fields
- **token**: rather than sending ether, it should work with any ERC-20 token, if the token is 0, send ether instead
- **timestamp**: a request can only be executed before its timestamp

Changed function signatures:

function request(address destination, uint value, uint timestamp, address token);
function approve(uint id, address destination, uint value, uint timestamp, address token);
function reject(uint id, address destination, uint value, uint timestamp, address token);

Part 5: JS Testing

Test (using javascript) at least that:

- Spending within the spending limit works
- Executing requests within the timeout works
- Executing requests after the timeout fails. Use **evm_increaseTime** to do so.
- Executing a rejected request does not work

Part 6: Solidity Testing

Test (using Solidity) at least that:

- the owner can spend ether from the wallet (no need to the other accounts)
- the owner can whitelist someone
- other accounts cannot whitelist someone

Part 7: Code as a Library

To reduce deployment cost, move the request related logic to a separate Library.
Make sure not to break ABI! All your tests should continue to work unmodified!

This means
- approve
- reject
- request
- execute

Use the WalletLibrary **for** the **Request[]** type for maximal readability
Checking the owner must still happen inside the main Wallet contract
Your (public) library functions **must not** be internal to avoid copying.
Internal helper functions are OK!

Hint: Events can also be defined in the Library. But make keep them in the main contract as well.

Part 8: Updates

Make the Wallet updatable by separating the code from the data.
Introduce a WalletProxy contract that acts as the entry point and stores the data
A Wallet code instance is supplied as a constructor argument
Calls need to be forwarded to this using **delegatecall**

Implement a **function __upgrade(address next)** where the owner can update the code contract!

In the tests replace **await Wallet.deployed()** by
Wallet.at((await WalletProxy.deployed()).address)
The rest of the tests should remain unchanged!

Hint: If you're getting weird values for your owner field you either
- didn't set it in the proxy constructor
- didn't think of the storage layout