---

title: How to mock Solidity smart contracts for testing description: Why you should make fun of your contracts when testing author: Markus Waas lang: en tags: ["solidity", "smart contracts", "testing", "mocking"] skill: intermediate published: 2020-05-02 source: soliditydeveloper.com sourceUrl: https://soliditydeveloper.com/mocking-contracts

---

Mock objects are a common design pattern in object-oriented programming. Coming from the old French word 'mocquer' with the meaning of 'making fun of', it evolved to 'imitating something real' which is actually what we are doing in programming. Please only make fun of your smart contracts if you want to, but mock them whenever you can. It makes your life easier.

Unit-testing contracts with mocks {#unit-testing-contracts-with-mocks}

Mocking a contract essentially means creating a second version of that contract which behaves very similar to the original one, but in a way that can be easily controlled by the developer. You often end up with complex contracts where you only want to unit-test small parts of the contract. The problem is what if testing this small part requires a very specific contract state that is difficult to end up in?

You could write complex test setup logic every time that brings in the contract in the required state or you write a mock. Mocking a contract is easy with inheritance. Simply create a second mock contract that inherits from the original one. Now you can override functions to your mock. Let us see it with an example.

Example: Private ERC20 {#example-private-erc20}

We use an example ERC-20 contract that has an initial private time. The owner can manage private users and only those will be allowed to receive tokens at the beginning. Once a certain time has passed, everyone will be allowed to use the tokens. If you are curious, we are using the `_beforeTokenTransfer` hook from the new OpenZeppelin contracts v3.

```solidity pragma solidity ^0.6.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol"; import "@openzeppelin/contracts/access/Ownable.sol";

contract PrivateERC20 is ERC20, Ownable { mapping (address => bool) public isPrivateUser; uint256 private publicAfterTime;

```
constructor(uint256 privateERC20timeInSec) ERC20("PrivateERC20", "PRIV") public {
    publicAfterTime = now + privateERC20timeInSec;
}

function addUser(address user) external onlyOwner {
    isPrivateUser[user] = true;
}

function isPublic() public view returns (bool) {
    return now >= publicAfterTime;
}

function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual override {
    super._beforeTokenTransfer(from, to, amount);

    require(_validRecipient(to), "PrivateERC20: invalid recipient");
}

function _validRecipient(address to) private view returns (bool) {
    if (isPublic()) {
        return true;
    }

    return isPrivateUser[to];
}
~~~

} ```

And now let's mock it.

```solidity pragma solidity ^0.6.0; import "../PrivateERC20.sol";
~~~

contract PrivateERC20Mock is PrivateERC20 { bool isPublicConfig;

```
constructor() public PrivateERC20(0) {}

function setIsPublic(bool isPublic) external {
    isPublicConfig = isPublic;
}

function isPublic() public view returns (bool) {
    return isPublicConfig;
}
```

} ```

You will get one of the following error messages:

- `PrivateERC20Mock.sol: TypeError: Overriding function is missing "override" specifier.`
- `PrivateERC20.sol: TypeError: Trying to override non-virtual function. Did you forget to add "virtual"?.`

Since we are using the new 0.6 Solidity version, we have to add the `virtual` keyword for functions that can be overridden and override for the overriding function. So let us add those to both `isPublic` functions.

Now in your unit tests, you can use `PrivateERC20Mock` instead. When you want to test the behavior during the private usage time, use `setIsPublic(false)` and likewise `setIsPublic(true)` for testing the public usage time. Of course in our example, we could just use time helpers to change the times accordingly as well. But the idea of mocking should be clear now and you can imagine scenarios where it is not as easy as simply advancing the time.

Mocking many contracts {#mocking-many-contracts}

It can become messy if you have to create another contract for every single mock. If this bothers you, you can take a look at the MockContract library. It allows you to override and change behaviors of contracts on-the-fly. However, it works only for mocking calls to another contract, so it would not work for our example.

Mocking can be even more powerful {#mocking-can-be-even-more-powerful}

The powers of mocking do not end there.

- Adding functions: Not only overriding a specific function is useful, but also just adding additional functions. A good example for tokens is just having an additional `mint` function to allow any user to get new tokens for free.
- Usage in testnets: When you deploy and test your contracts on testnets together with your dapp, consider using a mocked version. Avoid overriding functions unless you really have to. You want to test the real logic after all. But adding for example a reset function can be useful that simply resets the contract state to the beginning, no new deployment required. Obviously you would not want to have that in a Mainnet contract.