Blockchain for Industrial Engineers: Decentralized Application Development

บล็อกเชนสำหรับวิศวกรอุตสาหการ: การพัฒนาแอปพลิเคชันแบบกระจายศูนย์

Testing smart contract

Why (automated) testing software

- To reduce bugs.
- Tests allow you to make changes in your code quickly.
- Helps break down problems into manageable pieces.
- Let’s you sleep at night (because you actually know that your code works).

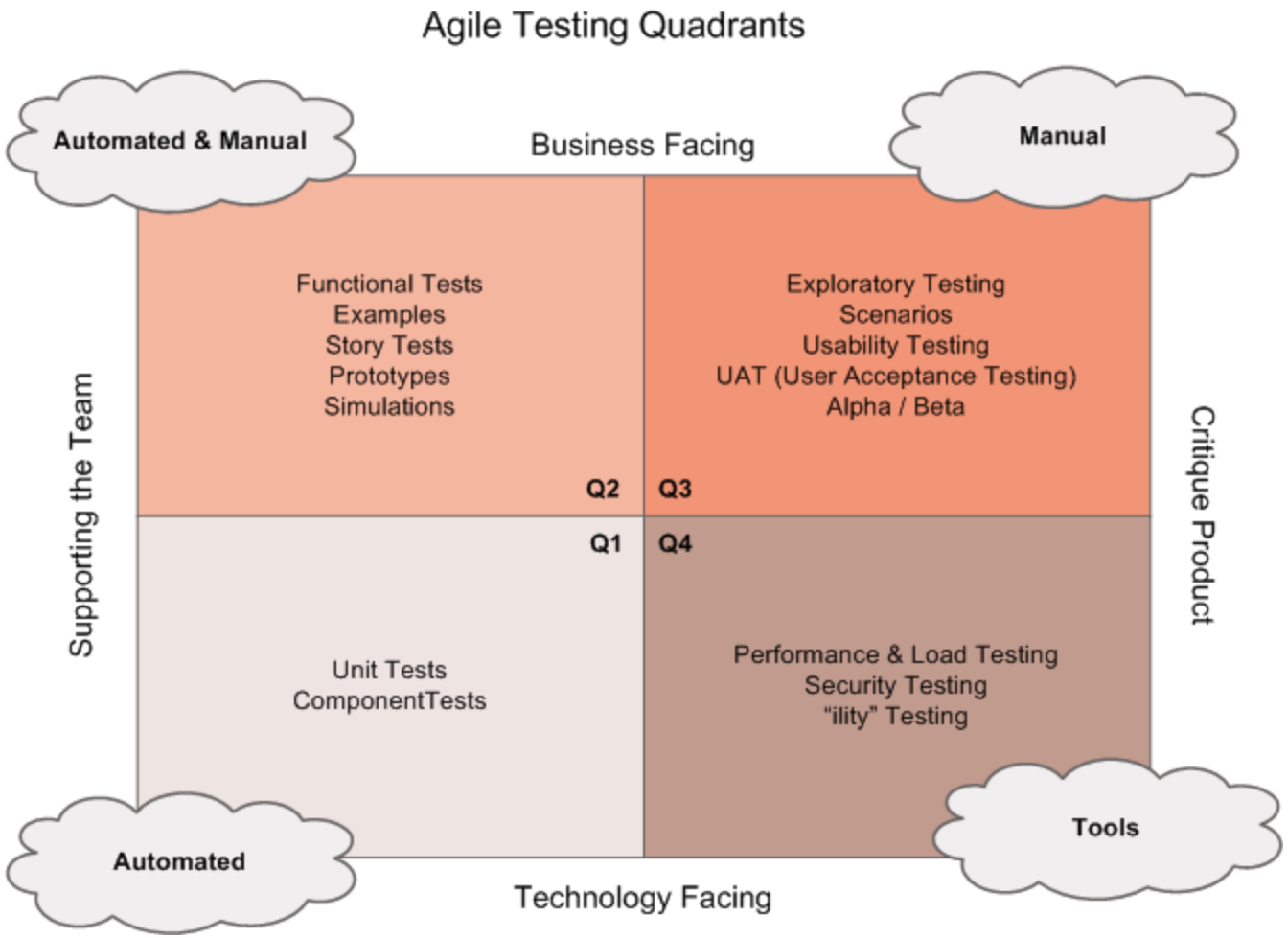
Testing smart contract

- Remember that smart contracts cannot typically be updated after launching.
- You want to make sure that it **works correctly** before deploying.

What do we test in the contract?

- Technical-facing test
  - For programmer
  - Unit test
  - Test Driven Development (TDD)
- Business-facing test
  - For client
  - Integrated test
  - Behavior Driven Development (BDD)

Agile testing quadrant



Story

`Scenario`: double result
`Given`: a variable `x` with value 2
`When`: I multiply `x` by 2
`Then`: `x` should equal 4

Examples

	Technical (TDD)	Business (BDD)
Scenario	Resetting state	Winning
Given	`players` array is not empty	Running lottery
When	calling `pickWinner`	calling `pickWinner`
Then	`players` array is empty	Winner gets money. Others don't.

Testing tools

- Framework
  - Mocha
- Assertion Library
  - Tools to verify that things are correct
  - Chai
  - Chai-Matcher
- JS blockchain library
  - Ether.js

Test-code structure

```
describe("Set 1", function () {
  describe("1-1", function () {
    it("does something 1", function () {});
    it("does something 2", function () {});
    it("does something 3", function () {});
  });

  describe("1-2", function () {
    it("does something 4", function () {});
    it("does something 5", function () {});
  });
});

describe("Set 2", function () {
  describe("2-1", function () {
    it("does something 6", function () {});
    it("does something 7", function () {});
    it("does something 8", function () {});
  });
});
```

Running test

- `npx hardhat test`
- `npx hardhat test --grep 5`
  - Only run tests that matche the pattern.

expect

```
import { expect } from "chai";

function double(a: number) {
  return a * 2;
}

describe("Set 3", function () {
  it("equals 1", function () {
    expect(double(1)).to.be.equal(2);
  });
  it("is greater than 1", function () {
    expect(double(2)).to.be.greaterThan(3);
  });
  it("is not equal to itself", function () {
    expect(double(10)).to.not.be.equal(10);
  });
});
```

Let's test a smart contract

`./contracts/MySecret.sol`

```
// SPDX-License-Identifier: GPL-3.0
import "../node_modules/hardhat/console.sol";
pragma solidity >=0.7.0 <0.9.0;
contract MySecret {
    string public secret;
    constructor(string memory _secret) {
        secret = _secret;
    }
    function changeSecret(string memory _secret) public {
        secret = _secret;
    }
}
```

Run `npx hardhat compile`

Test structure

```
import { expect } from "chai";
import { ethers } from "hardhat";

describe("Set 4", function () {
  // ...
  // ...
});
```

Check initial message

```
//  ...
it("checks initial message", async function () {
  const [owner] = await ethers.getSigners();
  const MySecret = await ethers.getContractFactory("MySecret");
  const mySecret = await MySecret.connect(owner).deploy("Super Secret");

  const message = await mySecret.secret();
  expect(message).to.be.equal("Super Secret");
});
//  ...
```

Check changing message

```
// ...
it("checks that message can be changed", async function () {
  const [owner, other] = await ethers.getSigners();
  const MySecret = await ethers.getContractFactory("MySecret");
  const mySecret = await MySecret.connect(owner).deploy("Super Secret");

  await mySecret.connect(other).changeSecret("Changed Secret");
  expect(await mySecret.secret()).to.be.equal("Changed Secret");
});
// ...
```

Use of `loadFixture`

- Notice that we always have to repeatedly deploy the contract in every test.
- We can use `loadFixture` to avoid repeating the same process.
- `loadFixture` will run the setup the first time, and quickly return to that state in the other tests.

Setup `loadFixture`

```
import { loadFixture } from "@nomicfoundation/hardhat-network-helpers";

async function deployMySecret() {
  const [owner, other] = await ethers.getSigners();
  const MySecret = await ethers.getContractFactory("MySecret");
  const mySecret = await MySecret.connect(owner).deploy("Super Secret");
  return { owner, other, mySecret };
}
```

Use `loadFixture`

```
it("checks initial message", async function () {
  const { mySecret } = await loadFixture(deployMySecret);
  const message = await mySecret.secret();
  expect(message).to.be.equal("Super Secret");
});
```

Use `loadFixture`

```
it("checks that message can be changed", async function () {
  const { other, mySecret } = await loadFixture(deployMySecret);
  await mySecret.connect(other).changeSecret("Changed Secret");
  expect(await mySecret.secret()).to.be.equal("Changed Secret");
});
```

BigNumber

- Many operations in Ethereum operate on numbers which are outside the range of safe values to use in JavaScript.
- A BigNumber is an object which safely allows mathematical operations on numbers of any magnitude.
- Most operations which need to return a value will return a BigNumber and parameters which accept values will generally accept them.

Experimenting with BigNumber

```
import { expect } from "chai";
import { ethers } from "hardhat";
const { BigNumber } = ethers;

describe("Set 5", function () {
  it("tests BigNumber", async function () {
    const big1 = BigNumber.from(100);
    const big2 = BigNumber.from(10);
    expect(big1).to.be.equal(100);
    expect(big1.add(big2)).to.be.equal(110);
    expect(big1.sub(big2)).to.be.equal(90);
    expect(big1.mul(big2)).to.be.equal(1000);
  });
});
```

Testing remaning balance

```
describe("Set 6", function () {
  it("checks gas", async function () {
    const { other, mySecret } = await loadFixture(deployMySecret);
    const balanceBefore = await other.getBalance();
    const tx = await mySecret.connect(other).changeSecret("Changed Secret");
    const receipt = await tx.wait();
    const gas = receipt.gasUsed.mul(receipt.effectiveGasPrice);
    const balanceAfter = await other.getBalance();
    expect(balanceAfter).to.be.equal(balanceBefore.sub(gas));
  });
});
```