



You have to implement a simple banking system that only supports the deposit and withdrawal of money. Initially, there are **n** bank accounts, the **ith** of which has **money[i]** money on the balance, and there is a service **IBankService** supporting three operations:

**public interface IBankingService**

**{**

**double GetBalance(int accountId);**

**void Withdraw(int accountId, double amount);**

**void Deposit(int accountId, double amount);**

**}**

* The method **GetBalance** returns the amount of money in the account **accountId**.
* The method **Withdraw** either withdraws **amount** money from the account **accountId** or throws an exception if there is not enough money available in the account.
* The method **Deposit** deposits **amount** money to the account **accountId**.

The methods should also throw an exception if either there is no account with **accountId** or **amount** is not positive. Implement this service as effectively as you can.

Consider this as a TDD task — you are given pre-written unit tests which fail at the beginning, and your task is to implement a solution which makes them pass.

* **[execution time limit] 20 seconds**

**Test Methods to test our code:**

using System;

using SimpleBankingSystem.Service;

using Xunit;

namespace SimpleBankingSystem.Tests

{

public class FunctionalTests

{

public class WithdrawTests

{

private readonly IBankingService \_bankingService;

public WithdrawTests()

{

double[] accounts = {1000.0, 1000.0, 1000.0};

\_bankingService = new BankingService(accounts);

}

[Fact(DisplayName = "Should withdraw when amount is valid and account exists")]

public void ShouldWithdrawSuccessfullyWhenAmountIsValidAndAccountExists()

{

// when

\_bankingService.Withdraw(0, 100);

\_bankingService.Withdraw(1, 200);

\_bankingService.Withdraw(2, 300);

// then

Assert.Equal(900, \_bankingService.GetBalance(0));

Assert.Equal(800, \_bankingService.GetBalance(1));

Assert.Equal(700, \_bankingService.GetBalance(2));

}

[Theory(DisplayName = "Should throw ArgumentException when withdraw from non-existing account")]

[InlineData(-1)]

[InlineData(3)]

public void ShouldThrowArgumentExceptionOnWithdrawalWhenAccountDoesNotExist(int outOfBoundId)

{

Assert.Throws<ArgumentException>(() => \_bankingService.Withdraw(outOfBoundId, 100));

}

[Theory(DisplayName = "Should throw ArgumentException when withdraw wrong amount of money")]

[InlineData(0.0)]

[InlineData(-100.0)]

public void ShouldThrowArgumentExceptionOnWithdrawalWhenAmountIsInvalid(double invalidAmount)

{

Assert.Throws<ArgumentException>(() => \_bankingService.Withdraw(1, invalidAmount));

}

[Fact(DisplayName = "Should throw ArgumentException when withdraw too much money")]

public void ShouldThrowArgumentExceptionOnWithdrawalWhenInsufficientMoney()

{

const double tooMuchMoney = 1500.0;

Assert.Throws<ArgumentException>(() => \_bankingService.Withdraw(1, tooMuchMoney));

}

}

public class DepositTests

{

private readonly IBankingService \_bankingService;

public DepositTests()

{

double[] accounts = {1000, 1000, 1000};

\_bankingService = new BankingService(accounts);

}

[Fact(DisplayName = "Should deposit when amount is valid and account exists")]

public void ShouldDepositSuccessfullyWhenAmountIsValidAndAccountExists()

{

// when

\_bankingService.Deposit(0, 100);

\_bankingService.Deposit(1, 200);

\_bankingService.Deposit(2, 300);

// then

Assert.Equal(1100, \_bankingService.GetBalance(0));

Assert.Equal(1200, \_bankingService.GetBalance(1));

Assert.Equal(1300, \_bankingService.GetBalance(2));

}

[Theory(DisplayName = "Should throw ArgumentException when deposit to non-existing account")]

[InlineData(-1)]

[InlineData(3)]

public void ShouldThrowArgumentExceptionOnDepositWhenAccountDoesNotExist(int outOfBoundId)

{

Assert.Throws<ArgumentException>(() => \_bankingService.Deposit(outOfBoundId, 100));

}

[Theory(DisplayName = "Should throw ArgumentException when deposit wrong amount of money")]

[InlineData(0.0)]

[InlineData(-100.0)]

public void ShouldThrowArgumentExceptionOnDepositWhenAmountIsInvalid(double invalidAmount)

{

Assert.Throws<ArgumentException>(() => \_bankingService.Deposit(1, invalidAmount));

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using SimpleBankingSystem.Service;

using Xunit;

namespace SimpleBankingSystem.Tests

{

public class StressTests

{

private const double InitialAmount = 10\_000\_000.0;

private const int Timeout = 1500;

private static readonly Random Rnd = new Random(10);

[Theory(DisplayName = "Should perform all operations concurrently")]

[InlineData(5, 1000)]

[InlineData(5, 2000)]

[InlineData(5, 5000)]

[InlineData(5, 10000)]

[InlineData(10, 1000)]

[InlineData(10, 10000)]

[InlineData(50, 500)]

[InlineData(50, 1000)]

[InlineData(100, 100)]

[InlineData(100, 500)]

public void ShouldPerformAllOperationsConcurrently(int numberOfAccounts, int operations)

{

var accounts = new double[numberOfAccounts];

Array.Fill(accounts, InitialAmount);

var bankingService = new BankingService(accounts);

var actions = new List<Action>();

var ids = Enumerable.Range(0, numberOfAccounts).ToArray();

foreach (var id in ids)

{

for (var i = 0; i < operations; i++)

{

actions.Add(() => { bankingService.Withdraw(id, 5); });

actions.Add(() => { bankingService.Deposit(id, 4); });

}

}

var shuffledActions = actions.OrderBy(x => Rnd.Next()).ToList();

if (!Task.WaitAll(shuffledActions.Select(Task.Run).ToArray(), Timeout))

{

throw new TimeoutException();

}

foreach (var id in ids)

{

Assert.Equal(InitialAmount - operations, bankingService.GetBalance(id));

}

}

}

}

Code need to write: **// write your code here**

using System;

namespace SimpleBankingSystem.Service

{

public class BankingService : IBankingService

{

public BankingService(double[] accounts)

{

// write your code here

}

public double GetBalance(int accountId)

{

// write your code here

}

public void Withdraw(int accountId, double amount)

{

// write your code here

}

public void Deposit(int accountId, double amount)

{

// write your code here

}

}

}