TDT4165 Programming Languages Assignment 6

Martin Hegnum Johannessen

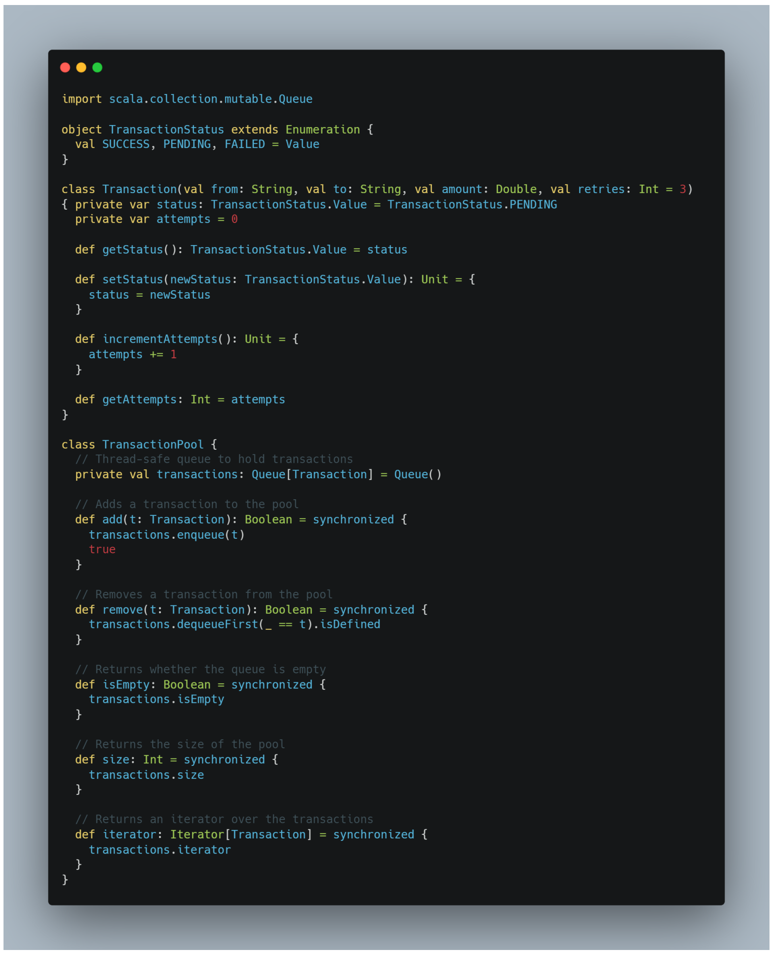
October 9, 2024

Task: 1 Preliminaries

Task: 1.1 Implementation of TransactionPool

The TransactionPool class manages a collection of Transaction objects. The goal is to ensure the class is thread-safe, meaning it can be accessed by multiple threads simultaneously without causing data inconsistencies.

To make it thread-safe, we will use the synchronized keyword to wrap operations that modify or read the shared data structure. In this case, we will use a Queue to hold the transactions. The implemented methods, such as isEmpty, size, and iterator, allow for efficient management of the transaction pool. The implemented code is presented in Figure 1



```scala
import scala.collection.mutable.Queue

object TransactionStatus extends Enumeration {
  val SUCCESS, PENDING, FAILED = Value
}

class Transaction(val from: String, val to: String, val amount: Double, val retries: Int = 3)
{ private var status: TransactionStatus.Value = TransactionStatus.PENDING
  private var attempts = 0

  def getStatus(): TransactionStatus.Value = status

  def setStatus(newStatus: TransactionStatus.Value): Unit = {
    status = newStatus
  }

  def incrementAttempts(): Unit = {
    attempts += 1
  }

  def getAttempts: Int = attempts
}

class TransactionPool {
  // Thread-safe queue to hold transactions
  private val transactions: Queue[Transaction] = Queue()

  // Adds a transaction to the pool
  def add(t: Transaction): Boolean = synchronized {
    transactions.enqueue(t)
    true
  }

  // Removes a transaction from the pool
  def remove(t: Transaction): Boolean = synchronized {
    transactions.dequeueFirst(_ == t).isDefined
  }

  // Returns whether the queue is empty
  def isEmpty: Boolean = synchronized {
    transactions.isEmpty
  }

  // Returns the size of the pool
  def size: Int = synchronized {
    transactions.size
  }

  // Returns an iterator over the transactions
  def iterator: Iterator[Transaction] = synchronized {
    transactions.iterator
  }
}
```

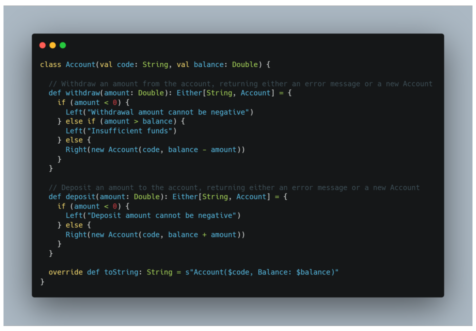
Figure 1: Task 1.1 - Code implementation

Task: 1.2 Account Functions

Two core functions were implemented, withdraw and deposit, for the Account class. The Account class is designed to be immutable, meaning that every transaction (withdrawal or deposit) creates and returns a new Account object with the updated balance instead of modifying the existing object.

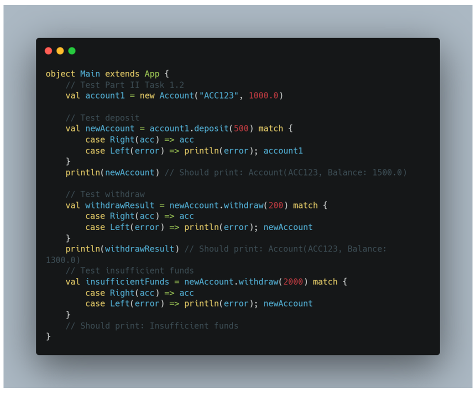
Task: 1.3 Eliminating Exceptions

Using the Either type for error handling allows the program to avoid exceptions and gracefully handle illegal transactions. Figure 2 shows the final implementated code. Meanwhile, Figure 3 displays some easy iterative tests that were made during development. Figure 4 presents the compiled result of those tests.



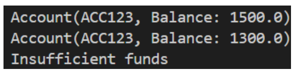
```scala
class Account(val code: String, val balance: Double) {

  // Withdraw an amount from the account, returning either an error message or a new Account
  def withdraw(amount: Double): Either[String, Account] = {
    if (amount < 0) {
      Left("Withdrawal amount cannot be negative")
    } else if (amount > balance) {
      Left("Insufficient funds")
    } else {
      Right(new Account(code, balance - amount))
    }
  }

  // Deposit an amount to the account, returning either an error message or a new Account
  def deposit(amount: Double): Either[String, Account] = {
    if (amount < 0) {
      Left("Deposit amount cannot be negative")
    } else {
      Right(new Account(code, balance + amount))
    }
  }

  override def toString: String = s"Account($code, Balance: $balance)"
}
```

Figure 2: Task 1.2 - Code implementation



```scala
object Main extends App {
    // Test Part II Task 1.2
    val account1 = new Account("ACC123", 1000.0)

    // Test deposit
    val newAccount = account1.deposit(500) match {
        case Right(acc) => acc
        case Left(error) => println(error); account1
    }
    println(newAccount) // Should print: Account(ACC123, Balance: 1500.0)

    // Test withdraw
    val withdrawResult = newAccount.withdraw(200) match {
        case Right(acc) => acc
        case Left(error) => println(error); newAccount
    }
    println(withdrawResult) // Should print: Account(ACC123, Balance:
1300.0)
    // Test insufficient funds
    val insufficientFunds = newAccount.withdraw(2000) match {
        case Right(acc) => acc
        case Left(error) => println(error); newAccount
    }
    // Should print: Insufficient funds
}
```

Figure 3: Task 1.2 - Test code implementation



```
Account(ACC123, Balance: 1500.0)
Account(ACC123, Balance: 1300.0)
Insufficient funds
```

Figure 4: Task 1.2 - Test code compiled

Task: 2 Completing the Bank

The task involved completing the banking system. More specifically, completing the key functions within *Bank.scala*. The full implementation of Bank.scala is presented in Fiugre 5.

First, the *createAccount* function was implemented to create new accounts with a unique identifier (UUID) and an initial balance. These accounts are stored in an internal Map, ensuring they are retrievable through the *getAccount* function, which fetches account details based on their code.

For transaction handling, the transfer function adds new transactions between accounts to a transaction pool (*TransactionPool*). These transactions are processed concurrently by the *processTransactions* function, which creates threads for each pending transaction. The function also manages retries for failed transactions by decrementing their retry count, while successful and exhausted transactions are moved to a completed transaction pool.

In the *Transaction* class, modifications were made to include the ability to track retries and status changes (See Figure 6). The *setStatus* method was added to update the transaction's status, while the retries were adjusted using immutable handling.

Threading was handled through the *processSingleTransaction* method, which spawns threads to process each transaction independently. The immutability of accounts and thread-safety of transactions were maintained throughout the system to avoid data corruption.

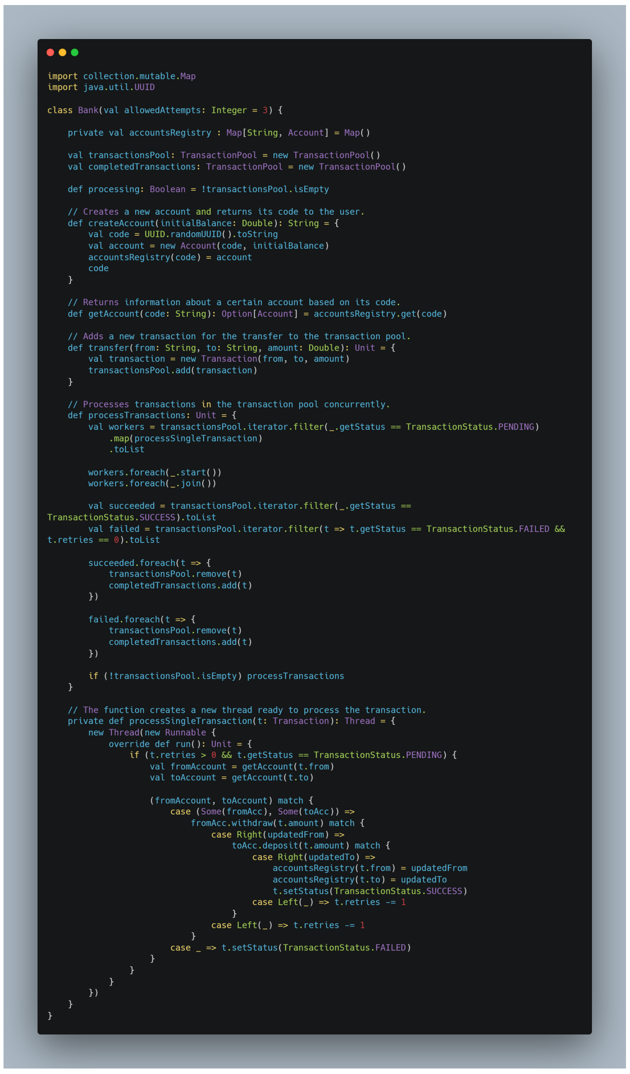
Task: 3 Explain how the code works

Task: 3.1 How does the code of the bank system work?

The banking system is designed to handle real-time transactions between accounts. It does so by composing the components: accounts, transactions, and transaction processing. Accounts are stored in a registry (Map), and transactions are placed in a TransactionPool for processing. Transactions can be pending, successful, or failed. The bank processes transactions concurrently by creating threads to handle each pending transaction. Failed transactions are retried until the allowed attempts are exhausted, after which they are marked as failed and moved to a completed transaction pool.

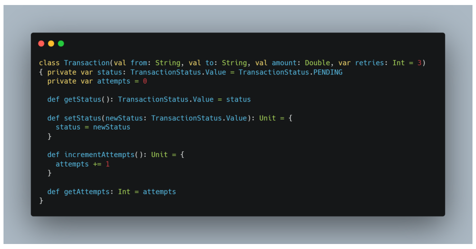
Task: 3.2 Which features of the system were the easiest to implement and why?

The creation of accounts and the retrieval of account information were relatively easy to implement. These functions mainly involve storing and re-



```scala
import collection.mutable.Map
import java.util.UUID

class Bank(val allowedAttempts: Integer = 3) {

    private val accountsRegistry : Map[String, Account] = Map()

    val transactionsPool: TransactionPool = new TransactionPool()
    val completedTransactions: TransactionPool = new TransactionPool()

    def processing: Boolean = !transactionsPool.isEmpty

    // Creates a new account and returns its code to the user.
    def createAccount(initialBalance: Double): String = {
        val code = UUID.randomUUID().toString
        val account = new Account(code, initialBalance)
        accountsRegistry(code) = account
        code
    }

    // Returns information about a certain account based on its code.
    def getAccount(code: String): Option[Account] = accountsRegistry.get(code)

    // Adds a new transaction for the transfer to the transaction pool.
    def transfer(from: String, to: String, amount: Double): Unit = {
        val transaction = new Transaction(from, to, amount)
        transactionsPool.add(transaction)
    }

    // Processes transactions in the transaction pool concurrently.
    def processTransactions: Unit = {
        val workers = transactionsPool.iterator.filter(_.getStatus == TransactionStatus.PENDING)
            .map(processSingleTransaction)
            .toList

        workers.foreach(_.start())
        workers.foreach(_.join())

        val succeeded = transactionsPool.iterator.filter(_.getStatus ==
TransactionStatus.SUCCESS).toList
        val failed = transactionsPool.iterator.filter(t => t.getStatus == TransactionStatus.FAILED &&
t.retries == 0).toList

        succeeded.foreach(t => {
            transactionsPool.remove(t)
            completedTransactions.add(t)
        })

        failed.foreach(t => {
            transactionsPool.remove(t)
            completedTransactions.add(t)
        })

        if (!transactionsPool.isEmpty) processTransactions
    }

    // The function creates a new thread ready to process the transaction.
    private def processSingleTransaction(t: Transaction): Thread = {
        new Thread(new Runnable {
            override def run(): Unit = {
                if (t.retries > 0 && t.getStatus == TransactionStatus.PENDING) {
                    val fromAccount = getAccount(t.from)
                    val toAccount = getAccount(t.to)

                    (fromAccount, toAccount) match {
                        case (Some(fromAcc), Some(toAcc)) =>
                            fromAcc.withdraw(t.amount) match {
                                case Right(updatedFrom) =>
                                    toAcc.deposit(t.amount) match {
                                        case Right(updatedTo) =>
                                            accountsRegistry(t.from) = updatedFrom
                                            accountsRegistry(t.to) = updatedTo
                                            t.setStatus(TransactionStatus.SUCCESS)
                                        case Left(_) => t.retries -= 1
                                    }
                                case Left(_) => t.retries -= 1
                            }
                        case _ => t.setStatus(TransactionStatus.FAILED)
                    }
                }
            }
        })
    }
}
```

Figure 5: Task 2 - Bank.scala



```scala
class Transaction(val from: String, val to: String, val amount: Double, var retries: Int = 3)
{ private var status: TransactionStatus.Value = TransactionStatus.PENDING
  private var attempts = 0

  def getStatus(): TransactionStatus.Value = status

  def setStatus(newStatus: TransactionStatus.Value): Unit = {
    status = newStatus
  }

  def incrementAttempts(): Unit = {
    attempts += 1
  }

  def getAttempts: Int = attempts
}
```

Figure 6: Task 2 - Transaction.scala

trieving data from a key-value Map, which is a straightforward operation in Scala.

Task: 3.3 Which features of the system were the most challenging to implement and why?

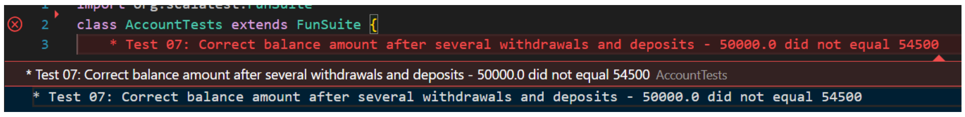
The most challenging part was implementing the concurrent transaction processing with retries. Handling concurrency in a thread-safe manner requires careful synchronization to avoid data races and inconsistent states.

Task: 3.4 Write a short summary of what you have implemented to make tests pass

In order to make the tests pass, I encountered issues primarily with Test 7, Test 11, and Test 12, which required changes to address concurrency problems and thread synchronization.

For Test 7, I initially synchronized the withdraw and deposit methods to prevent race conditions. However, issues still arose due to the way these methods were being called concurrently in the test code. To resolve this, I modified the Account class so that the balance field could be updated directly within the same instance instead of creating new Account instances for each withdrawal or deposit. Though, I still did not manage to make this pass the test (See Figure 7).

In Test 11 and Test 12, which involved multiple transfers between accounts, I ensured that the Bank class properly handled concurrent transactions. This included ensuring that all transfers were processed correctly and no transfers were skipped or failed due to concurrent access issues. Ultimately, I managed to make these tests pass (See Figure 8).



```
class AccountTests extends FunSuite {
    * Test 07: Correct balance amount after several withdrawals and deposits - 50000.0 did not equal 54500
```

* Test 07: Correct balance amount after several withdrawals and deposits - 50000.0 did not equal 54500 AccountTests

```
* Test 07: Correct balance amount after several withdrawals and deposits - 50000.0 did not equal 54500
```

Figure 7: AccountTests (1-7)

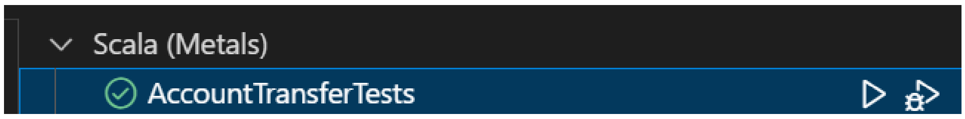


Figure 8: AccountTransferTests (8-12)