COMP6451 Assignment 2

Design Report: z5113817

Table of contents

Roles

ChiefOperatingOfficer

The chief operating officer of the university admissions session can deploy one of these contracts and then begin the session.

Properties

Property	Type	Description
owner	address	The owner of the contract (the user acting as chief officer)
tokenFee	uint256	The token cost of each UoC
sessionStarted	bool	A flag representing whether there is a current session in progress

Methods

Method	Arguments	Return Type	Modifiers	Description
startSession	ManagerFactory, UserFactory	-	requiresOwner	Creates an enrolment session
executeRound	-	bool	requiresOwner	Executes the current bidding round. Returns true if a new round was started.
changeOwner	address	-	requiresOwner	Changes the officer in command
getOwner	-	-	-	Returns the owner address
authorizeAdmin(address)	address	address	requiresOwner	Calls the RolesManager to authorize a new admin. Returns the contract address of the newly created Admin contract
setFee	uint256	-	requiresOwner	Sets the token fee
getFee	-	uint256	-	Returns the token fee
getUniversityBalance	-	uint256	-	Returns how many tokens are in the university system

purchaseUoC	address, uint8	bool	requiresSessionStart, requiresStudent	Student contract transfer Wei to this in exchange for tokens. COO approves student to spend its tokens.

Administrator

The administrator contract is deployed by the RolesManager when called by the ChiefOperatingOfficer contract.

Properties

Property	**Type**	**Description**
owner	address	The user address who the admin corresponds to (the only user who can make calls on this contract)

Methods

Method	**Arguments**	**Return Type**	**Modifiers**	**Description**
admitStudent	address	address	requiresOwner	Calls the RolesManager to authorize a new student. Returns the contract address of the newly created Student contract
revokeStudent	address	-	requiresOwner	Calls the RolesManager to authorize the student with the “Revoked” permission
createCourse	Course	-	requiresOwner	Calls the CourseManager to add a new course which students can bid for
setCourseQuota	string, uint16	-	requiresOwner	Calls the CourseManager to alter course quota
setDeadline	timestamp	-	requiresOwner	Calls the SessionManager to set the deadline

Student

The student contract is deployed by the RolesManager when called by the Administrator contract.

Properties

Property	Type	Description
owner	address	The owner of the contract
purchasedUoC	uint8	The number of UoC purchased by the student. This can diverge from the “allowance” as students transfer fees to each other
pendingBids	Bid[]	Bids pending in the current round
enrolled	Enrolment[]	An array of courses that the student has enrolled in for this session

Methods

Method	Arguments	Return Type	Modifiers	Description
getAllowance	-	uint256	requiresOwner	Returns the allowance of the COO that the Student can spend
getEnrolments	-	Enrolment[]	requiresOwner	Returns the successful enrolments of the Student
purchaseUoC	uint8, Wei	uint256	requiresOwner	Calls the TokensManager to purchase a desired amount of UoC. The student should also provide sufficient amounts of Wei.
addBid	string, amount	-	requiresOwner	Calls the SessionManager to get the current RoundManager and add a bid. Does error checking ensuring that the student has enough tokens and that the course exist and transfers allowance to RoundManager.
alterBid	Bid, Bid	-	requiresOwner	Calls the SessionManager

				to get the current RoundManager and alter a bid for a given code.
removeBid	bid	-	requiresOwner	Calls the SessionManager to get the current RoundManager and remove the bid.
transfer	address, uint256	-	requiresOwner	Calls the TokensManager to transfer their tokens.
bidSuccessful	Enrolment	-	requiresRoundManager	Called by RoundManager to notify that the bid was successful and enrol the student in the course.
bidUnsuccessful	Enrolment	-	requiresRoundManager	Called by the RoundManager to notify that the bid was unsuccessful. The student receives their tokens back.

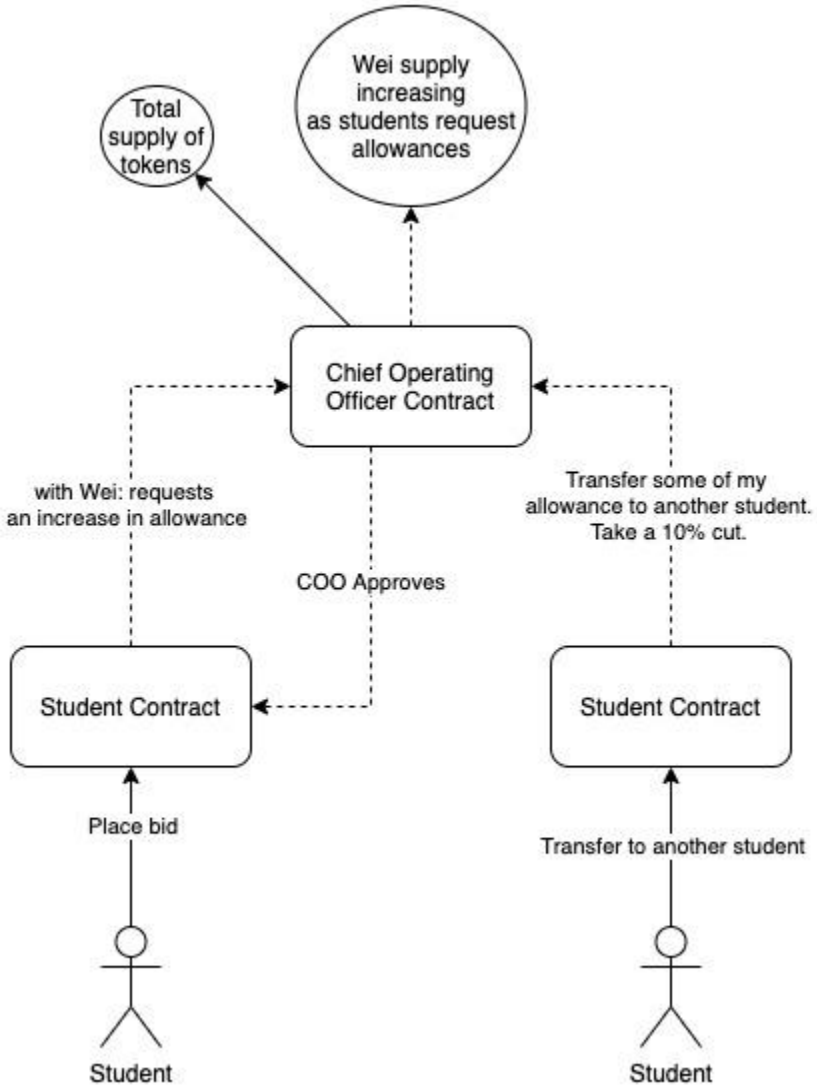
Managers

All of the managers are created when the Chief Operating Officer decides to create a session.

TokensManager

The TokensManager is responsible for managing student payments and allocating them tokens to bid on course enrolments. The TokensManager implements inherits from the ERC20 contract, although only the methods that I intend to use will be shown in the below tables.

The payment structure takes use of the “allowance” mechanism provided in an ERC20 contract. Tokens aren’t transferred between students and the COO always owns the tokens. Instead, students are given permission to spend the COO’s tokens (forming a sort of allowance) and can use these transfer rights for bidding or granting other students.



Properties

Property	Type	Description
totalSupply	uint256	The total supply of tokens in the system

Methods

Method	Arguments	Return Type	Modifiers	Description
approve	address, uint8	bool	requiresCOO	Approves a student allowance.
transferToStudent	address,uint256	bool	requiresStudent	Checks that both the sender and receiver are students and then calls transferFro, taking a 10% fee.
destroyTokens	uint256	-	requiresRoundManager	Removes token total supply to fit “destroy tokens” requirement

RolesManager

The RolesManager is responsible for handling the permissions associated with each role. The RolesManager keeps track of which contracts have what role.

Properties

Property	Type	Description
roles	address => Roles	A mapping of contract addresses to the Roles enum.

Methods

Method	Arguments	Return Type	Modifiers	Description
authorize	address, Roles	address	Conditional depending on the Roles value.	Authorizes the newly created contract address and returns its address. The owner of this contract is set to be the user supplied in the arguments.
hasRole	address, Role	boolean	-	Returns true if the supplied contract address has the supplied Role.

SessionManager

The SessionManager is responsible for managing rounds for the current session.

Properties

Property	Type	Description
currRound	address	The contract address to the current round
deadline	timestamp	The deadline of the current round. By default this is one day after the currRound was deployed.

Methods

Method	Arguments	Return Type	Modifiers	Description
executeRound	-	-	requiresChiefOperatingOfficer	Calls the RoundManager to execute the round. Will start a new round if

				required.
setDeadline	timestamp	-	requiresAdmin	Sets a new deadline
getCurrRound	-	address	requiresStudent	Returns the address of the current round to a student.

RoundManager

The RoundManager is responsible for managing bids for the current round.

Properties

Property	**Type**	**Description**
bidsPerStudent	address=>Bid[]	A mapping from a student contract address to an array of bids, representing all the bids that student currently has.
bidsPerCourse	string=>Bid[]	A mapping of the Course to an array of bids. This array of bids will be in descending order of the Bid.amount field. Even though this mapping captures the same information as “bidsPerStudent”, it is a more efficient way to represent the information when it comes to executing the round.

Methods

Method	**Arguments**	**Return Type**	**Modifiers**	**Description**
executeRound	-	boolean	requiresSessionManager	Enrols students in courses and pays back tokens. Returns true if a new round should be started.
addBid	Bid	-	requiresStudent	Adds a bid to the mappings
alterBid	string, Bid	-	requiresStudent	Finds the students bid and changes the bid properties
removeBid	string	-	requiresStudent	Removes the bid for the given course code
seeAllBids	-	Bid[]	requiresStudent	Returns all bids
kill	-	-	requiresSessionManager	Destroys this contract

CourseManager

The CourseManager is responsible for handling the existing courses in the system.

Properties

Property	Type	Description
courses	string=>Course	A mapping from course codes to Course.

Methods

Method	Arguments	Return Type	Modifiers	Description
addCourse	Course	-	requiresAdmin	Adds a course to the courses mapping
setQuota	string, uint16	-	requiresAdmin	Updates the supplied quota for the supplied course code
setEnrolment	string, address[]	-	requiresRoundManager	When the roundManager executes the round, it calls this to update the enrolled students
courseExists	string	bool	-	Returns true if the course code provided exists
getCourse	string	code	-	Returns the course, will fail if course does not exist

Factories

The factory contracts are responsible for handling the creation of other contracts. These factories are deployed by the COO and must be passed into the COO contract's "startSession" method before a session can begin.

ManagerFactory

The Manager Factory is contained in most of the managers so that they can grab the relevant contracts they need to execute their logic.

This doesn't really follow a factory pattern since the manager contracts have to be set in the contract. Only the COO can set these Managers. This was not the initial design however Ethereuem has a contract bytecode size limit which was exceeded when this contract was made with "creator" methods.

Methods

Getters and Setters for all Managers.

UserFactory

The User Factory is used by the RolesManager to create the User contracts.

Methods

Method	**Arguments**	**Return Type**	**Modifiers**	**Description**
createAdmin	address	address	requiresRolesManager	Creates a new Admin contract and sets the owner to the address supplied. This function returns the Admin contract address.
createStudent	address	address	requiresRolesManager	Creates a new Student contract and sets the owner to the address supplied. This function returns the Student contract address.

Data Structures

Roles

This is an enum consisting of “Admin”, “Student” and “Revoked”.

Bid

This is a Struct data type with the following fields.

Field	Type	Description
amount	uint256	The amount of tokens associated with the bid
course	Course	The course associated with the bid
student	address	The contract address associated with this bid
updated	timestamp	The last time the bid was updated by the student

Course

This is a Struct data type with the following fields.

Field	Type	Description
code	string	The course code e.g “COMP6451”. This is assumed to be the unique identifier of the course.
name	string	A more descriptive title for the course.
quota	uint16	The amount of students who can enrol. Maximum value is 65,535.
enrolled	address[]	The students who have enrolled.
UoC	uint8	The units of credits associated with this course. Maximum value is 255.

Enrolment

This is a Struct data type with the following fields.

Field	Type	Description
code	string	The course code e.g “COMP6451”. This is assumed to be the unique identifier of the course.
UoC	uin8	The number of UoC associated with this course.

Requirements

The prototype should be able to manage enrollment for one session, with an arbitrary number of bidding rounds. Management of multiple sessions is not required at this time.

The Chief Operating Officer can use the corresponding contract to create a session. The SessionsManager contract handles multiple rounds.

People using the system should be identified by means of an anonymous Ethereum address - the system should not store personal information on the blockchain.

People using the system are either the COO, an admin or a student. All of the functionality for this is handled by the respective contracts, which are controlled by an anonymous Ethereum address.

The system should support the following user roles: (many) students, (many) university administrators, and (one) university chief operating officer.

The RolesManager contract supports all of these roles.

Only the chief operating officer has the ability to control which users act in the roles of the university administrators. They may authorize users to act in these roles, and revoke a user's authority to act in a given role. They may also transfer their chief operating officer powers to another user.

The RolesManager contract's "authorizeAdmin" method can only be called by the ChiefOperatingOfficer contract. Only the owner of the ChiefOperatingOfficer contract can call methods to authorize an admin, revoke admin rights and transfer powers to another user.

Only the chief operating officer should have the ability to

- **– set the amount of fees (in Wei) per unit of credit**
- **– transfer money out of the admissions system.**

Only the owner of the ChiefOperatingOfficer can call the methods for setting fees and transferring money out of the admissions system.

Only university administrators should have the ability to:

- **– admit a student to the university**
- **– create a new course, setting its admissions quota, and number of UOC**
- **– change the quota on a course**
- **– set a bidding round deadline**

The RolesManager contract's method "authorizeStudent", the CourseManager contract's "addCourse" and "alterCourseQuota" and the SessionManager contract's "setDeadline" method all can only be called by an admin contract. Only the owner of an admin contract, who had to be authorized by the COO, can call these methods on the contract.

Students should be able to do the following:

- **– pay fees (in Wei) for a given number of units of credit**
- **– bid some of their admission tokens for admission to a course**
- **– change their bid before the round deadline**
- **– transfer some of their tokens to another student**

The Student contract has methods for all of these requirements.

Only students who have been admitted to the university may pay student fees and receive admission tokens.

The TokenManager contract's "approve" method can only be called by a student contract. Only the owner of a student contract, who had to be previously authorized by a university admin, can call the Student contract's "payFees" method. This ensures only authorized students can pay fees.

Students should not be able to bid more tokens in any round than they own at the time of that round.

This logic will be handled in the "addBid" method in the Student contract.

In any round, students may not bid for courses totaling more units of credit than they have paid for. (For example, if Alice has paid for 18 UOC, she may not bid for 3 courses with 12, 6, and 6 UOC, respectively, since the total UOC for these courses is 24 and there would be a problem if she is accepted to all these courses.)

This logic will be handled in the "addBid" method in the Student contract, using the "purchasedUoC" internal variable.

When a student is admitted to a course at the end of a round, the tokens that they have bid for the course in that round are destroyed.

The RoundManager contract's method "executeRound" will use the TokenManager to set the allowance of the student to the new calculated amount and destroy the tokens.

Students should not be able to enroll in a larger number of units of credit than they have paid for.

Because of restrictions in the "addBid" method, students will not be able to make bids for more UoC than they have purchased. Since the RoundManager is the only contract that can call the Student's "enrol" method, this means students will not be able to enrol in more courses then they have paid for.

It should not be possible for a student to transfer tokens without the university receiving the transfer payment.

The only way for students to transfer tokens is through the TokensManager contract's "transferFrom" method. This method takes a 10% fee before transferring to the desired student.

Your design should facilitate the secure sale of enrollment tokens for cryptocurrency, so that students cannot cheat each other when selling some of their tokens. However, the university does not wish to have legal liability for such transactions, and does not want to be a party to the actual transfer of money when a student sells some of their tokens to another student. Money paid by a student for tokens that they are buying should not pass through any smart contract controlled by the university. The university merely wishes to ensure that they will collect fees for the transfer.

There is no support in the contracts to exchange off-chain money for tokens. Only the token side of the transaction is recorded by the system.

The system should be cost effective for the university to run. To the largest extent possible, costs for running of the system should be borne by students rather than by the university.

Most computation comes from executing the round. Extra mappings were created in the RoundManager contract so that executing a round is O(n) which scales with the number of bids.

Cost Analysis

When running locally on a ganache instance, the gas price was set to 0 and the gas limit was the highest value (16777215), so cost was never an issue in development. This is however a consideration on Ethereum mainnet.

According to Etherscan, the average gas price is **310 Gwei**. As of 18th April 2021, 1 Ether is $2,705.61 AUD, meaning one Gwei (0.000000001 Ether) costs **$0.00000270561** AUD. These numbers will be referenced in my assumptions below.

Cost for Deployment

The total gas used for completing the initial setup described in the tests which involves deploying all of the Manager, Factory and COO contracts was **19,600,868** gas.

Using the assumptions above, this would cost **$16,440.014** AUD for deployment alone!

Cost for Student

The cost of a student is mainly in how much they bid. Even though students can perform other actions like transfer to other students, these costs are negligible compared to the storage cost of all the bids and executing a round.

Assuming a student places an average of 5 bids, the gas cost for this was **687,821** gas per student, or **$576.90** per student.

Summary

Using UNSW's reported numbers of 59,000 students and assuming every student enrols and has an average amount of activity, the total cost would be **$340,387,440.01** per session.

Discussion

Suitability

I don't believe my estimates are totally accurate and there were definitely some assumptions made about the average student activity. Even if, however, my estimate was 100 times larger than the real value, that's still around $3,000,000 the university will have to spend per trimester.

Because of the cost, I do not believe Ethereum mainnet would be a suitable application to host this application.

That being said, I spent very little time optimising my code meaning there could be some ways to drastically reduce this cost. In addition, the university could host its own instance of the Ethereum blockchain to get around these costs. Although there is a trade-off between privacy and decentralisation with cost, UNSW knows all student details and enrolment activity anyway.

Security

Assuming these contracts are open source and published somewhere, there could be exploitable bugs in my code. It would be a disaster if any student could call the COO contract to transfer all Wei to themselves, or automatically mark their bids as successful. I did my best to prevent this but didn't test extensively.

Design Considerations

If I had more time, I would make my contracts a lot smaller. Most of the gas cost comes from deploying new contracts, and contracts have a size limit. Because of this, Solidity contracts very strictly follow the "Single Responsibility" design principle.

I was forced to re-work my initial design of the COO contract as having the COO deploy all the managers when it's initialised made the contract bytecode way too large. This made me move to a Factory Pattern of sorts.

One area I would change is I would have an "EnrolmentManager" which is responsible for knowing all of the student's enrolments (rather than the student handling this). The RoundManager could then talk to this contract rather than the students directly. This would improve the design as well as reduce contract size.

Tests

All of the code is on github: https://github.com/william-coulter/university_admissions_system

The README.md contains information on how to run tests but I will copy here:

Dependencies (and versions)

- `npm`: 6.14.4
- `node`: v14.16.1
- `make`: 3.81
- `truffle`: `npm install -g truffle@v5.1.65`: v5.1.65
- `ganache-cli`: `npm install -g ganache-cli`: v6.12.2

Tests

First deploy the chain: `make deploy-chain`. This will deploy a `ganache` instance listening on `localhost:8545`.

Run the tests with: `make test` This will perform an `npm install` and make sure all of the contracts have been compiled and deployed. All the tests in the `/test` directory will run.

See `Makefile` for more commands.

No stretch goal was attempted.