Blockchain Based

# DECENTRALIZED LOAN SYSTEM.

Aliasgar Merchant

# Decentralized Loan System

## Table of Contents

## 1. Abstract

**Decentralized Loan System** is designed to address real world problems faced by 'Loan Systems' within a specific institution. The purpose of the paper is to demonstrate the benefits of **Interest Free** loans within a society. While – this kind of system is majorly based on a peer to peer system, we also introduce *'Admin'* or *'Administrator'* for uninterrupted operations within the institution. There are three major actors within the loan system –
*Elite* – They are responsible for donating funds within the institution.
*Needy* – They request for loan as per need.
*Admin* – They overlook the operations performed within the institution.

The paper also explains in depth the concept of **transparency** which is sometimes missing within these institutions. **Decentralized Loan System** is based on a **Blockchain** system to address this issue. Blockchain being **immutable** offers the stake holders the ability to *audit* all the transactions.

Also, the paper describes in detail – the **right to vote** which is bestowed upon the stake holders. This empowers them to take decisions as to *who shall be granted loan, what amount shall be granted and what shall be the repayment term.*

While, the loan is based on a **Interest Free** system – however collateral must be required at all times. The paper also explains, if the collateral offered is not sufficient how can the needy be still benefited from the Loan System.

In the end, the paper also talks about the future scope and benefits of using this kind of system.

**Major Software Modules:**

1. Loan Request / Loan Dispersal / Loan Repayment Smart Contract.
2. Collateral / Mortgage Smart Contract.
3. Election Smart Contract.
4. Audit Smart Contract.

## 2. Introductory notes and Background details

*Decentralized Loan System* specifically considers example of one particular ‘private institution.’
This segment of the paper explicitly tries to explain the working of that institution. Though, it might be presumed at first instance that this is a very specific problem solution catering to very specific audience, but this idea can be well expanded to fit in a number of scenarios.

However, please note given the great use-case of this institution and its ability; it may be used or replicated at a greater extent by any other institution.

## 3. Working of existing system

The ‘private institution’ mentioned above – works in the following ways:

There are three major *actors* in the existing loan process constituted by the private institution.

### Actor 1:

Let’s consider this actor to be “*Administrative Department*” also referred as *admin* throughout the document which consist of the following features.

a. Advertising for donating funds and increasing the pool of funds available.
b. Managing the funds – that is returning the funds to donating parties as and when required.
c. Allocating the funds to borrowers depending on the predefined parameters.

### Actor 2:

Let’s consider this actor to be “Loan Borrower” also referred as *needy* throughout the document which consist of the following features.

a. (S)He will request for certain amount of money depending on the need of the *needy*.
b. Based on the predefined parameters, the allocation is done by the trustees to *needy*.
c. The *needy* are subjected to submit collateral as defined by the *admin* department.

### Actor 3:

Let’s consider this actor to be “Loan Lender” also referred as *elite* throughout the document which consist of the following features.

a. (S)He will respond to the advertising of the fund’s donation.
b. (S)He will donate the funds and agree upon fund withdrawal rules.
c. (S)He will request for the sum of the money as and when required by the *elite*.

## 4. Storyline of a particular implement

### Needy Perspective

The *needy* will request for certain amount of money – depending upon the nature of the need. The *needy* will then submit all the necessary documents which demonstrate the 'actual need' of the *needy*. Then the *needy* is required to submit a detailed plan of how (s)he plans to return the money to the institution within given frame of time. Apart from all these details – the *needy* is required to submit a collateral of the loan. This collateral should be such, that it exceeds the value of the loan being issued. If, for any reason, the *needy* is unable to repay the loan; the collateral would be responsible for loan repayment.
Also, an important perspective to consider – the loan issued by the institution is **100% interest free**. Also, there are no "service" or other "hidden cost." Hence, the repayment of the entire loan amount will be same as the amount borrowed. Also, if the *needy* defaults the loan amount – steps are taken as described by the guidelines of the institution.

### Elite Perspective:

The *elite* will willingly give or offer certain amount of money to the pool to facilitate the institution to lend the money to people in need. While doing so the *elite* agrees to the general conditions laid down by the institution. The *elite* may hold the money with the institution for indefinite amount of time. If needed, the *elite* can request the money from the institution. The institution promises to repay the full amount requested within 3 to 7 business days. Also, since this is no interest scheme, the *elite* gets only the amount he had submitted initially. Hence, this should not be considered as a means of making profits or earning money; but purely as social cause. Also, there is a provision to offer money in this pool recursively every month.

### Admin Perspective

They are responsible for advertising the ideology so more and more people offer money in the pool. Once *needy* request for money, they check all the documents and necessary credentials. This is similar to running a complete credit check. They also manage timely repayment to *elite* as and when they demand. Also, they allocate the funds based on their discretion and availability of collateral.

## Fig: Traditional loan system – Class Diagram

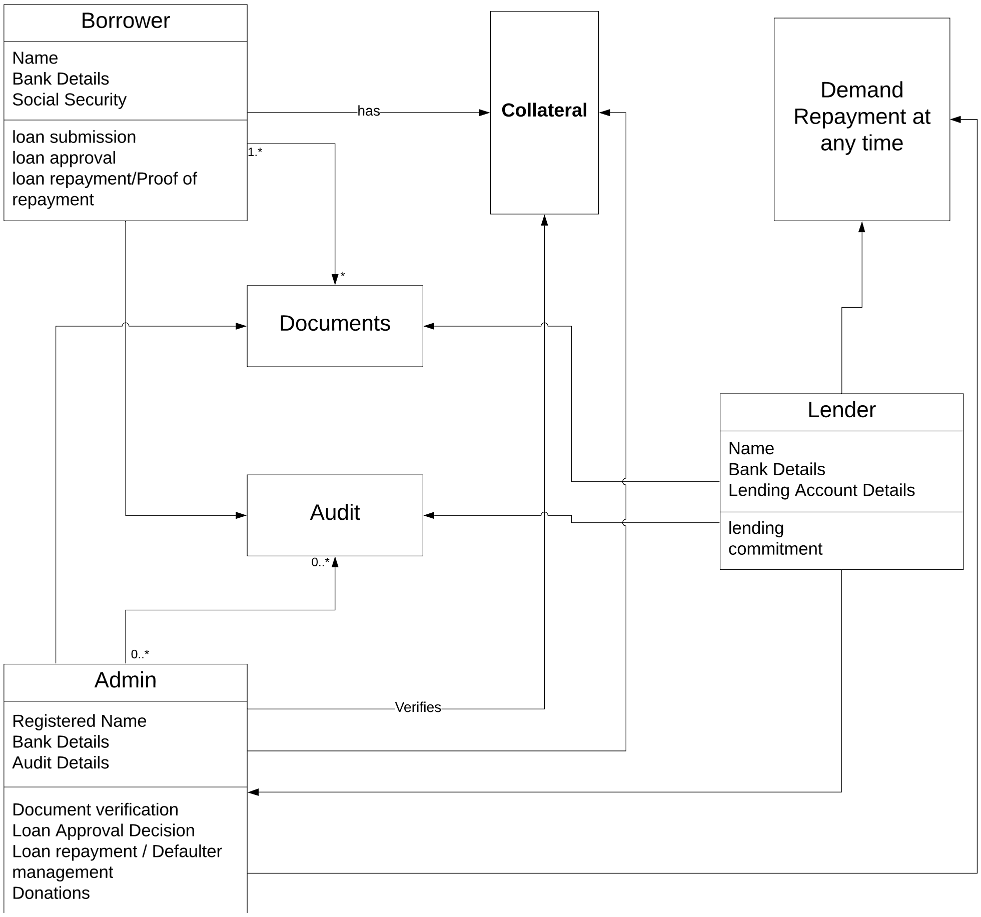
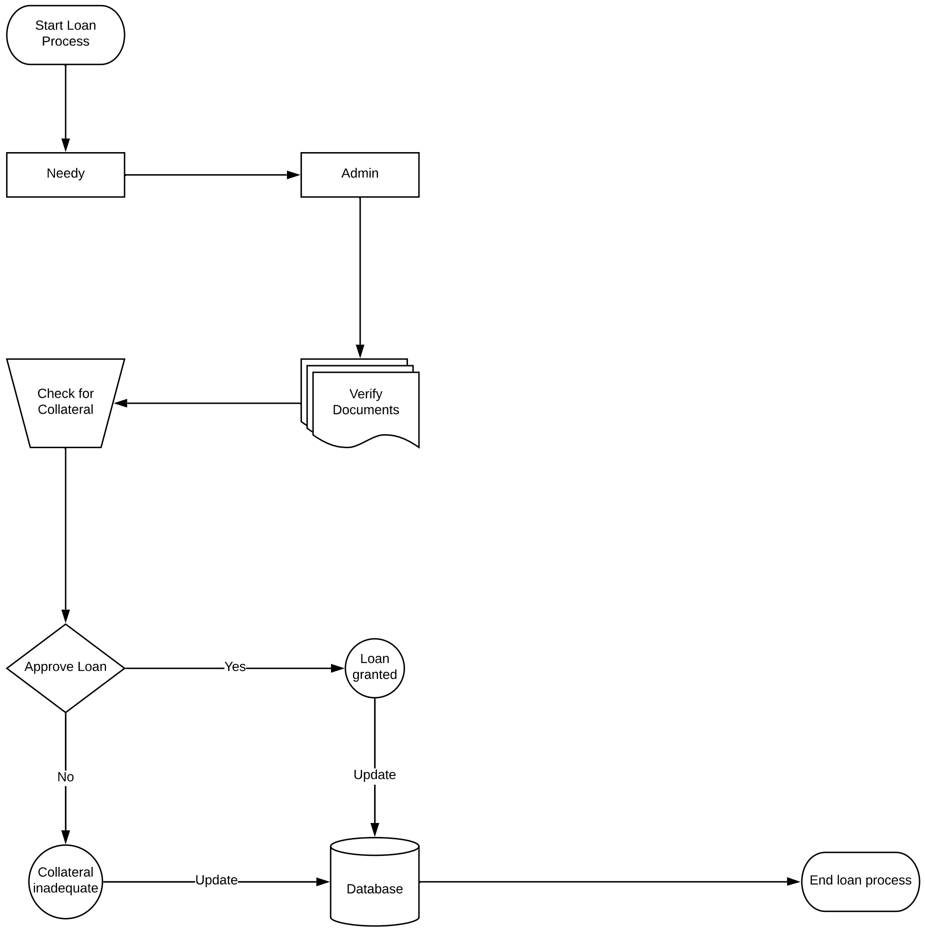


Fig: Traditional loan system – Flow chart



## 5. Area of Improvement

This is a very crucial part of the paper – as most of the parts described here will serve as the **'solution'** to **'special problem.'**

Before we define – the areas where the model can be improved let's consider the factors which influence these 'area of improvement.' We will define two perspective that is – one of the *elite* and other of *needy.* These serve as the foundation to the solution proposed in this paper. However, it should also be noted that – these perspectives only serve as a guideline or suggestion and not all ideas can be implemented in the solution.

### Needy Perspective:

The entire process should be more transparent – especially if the *needy* is being denied the loan amount.
They should be given the option of 'no collateral' loan option – only if the stake holders agree to it.

### Elite Perspective:

They should have 'on demand' transparency. The *elite* should have a right to 'vote' as to where and how the funds are being used. Note – The right to vote is a very subjective feature. Later parts of the paper explain in depth the concept of voting.
Also, the *elite* are able to see on demand every request made to the institution and other details. The transparency is subjective and not everything is revealed to the elite. Note – The paper also explains the reason for subjective transparency in later parts.

## Admin Perspective

The funds collected in existing system is stored, managed and distributed locally. The decision made by committee to allocate these funds have little or no influence from central authority. We try to implement a way in which existing resources can be pooled – especially in same geographic location as country. This will make efficient use of more resources. However, this 'use' is strictly defined using guidelines and standards explained in later parts of paper.

## 6. Solution

In this part of the paper we focus on solving the existing problems defined in 'area of improvement.'
We distribute the definition of solution as described in area of improvement.

The solution we offer is majorly focused on 'Blockchain based solution.' This segment of the paper evaluates how Blockchain can solve all existing concerns raised by respective actors. A supplement to this segment is comparison between Blockchain technology and other technologies and why Blockchain is better apt for this situation.

### Needy Perspective

Blockchain in general makes the entire transaction more transparent and open. One of the major uses at this stage – is for auditing which solves one of the concerns of the needy.
Once – the needy is rejected a loan, he/she can ask for a non-collateral loan from the stake holders.
Here – the stake holders (typically 5 – depends largely on the amount to be dispersed) can vote for dispersing the loan to the needy. Note: if the amount exceeds the defined amount for loan disperse without a collateral - then the Chairman of the committee may be needed to invoke.
If the needy fails to repay the said amount in said time – he/she may be permanently or temporarily banned from using the loan service.
Also, the stake holders who voted for the respective needy may lose the credibility of voting and would have to pay the amount as defined by the management.

### Elite Perspective

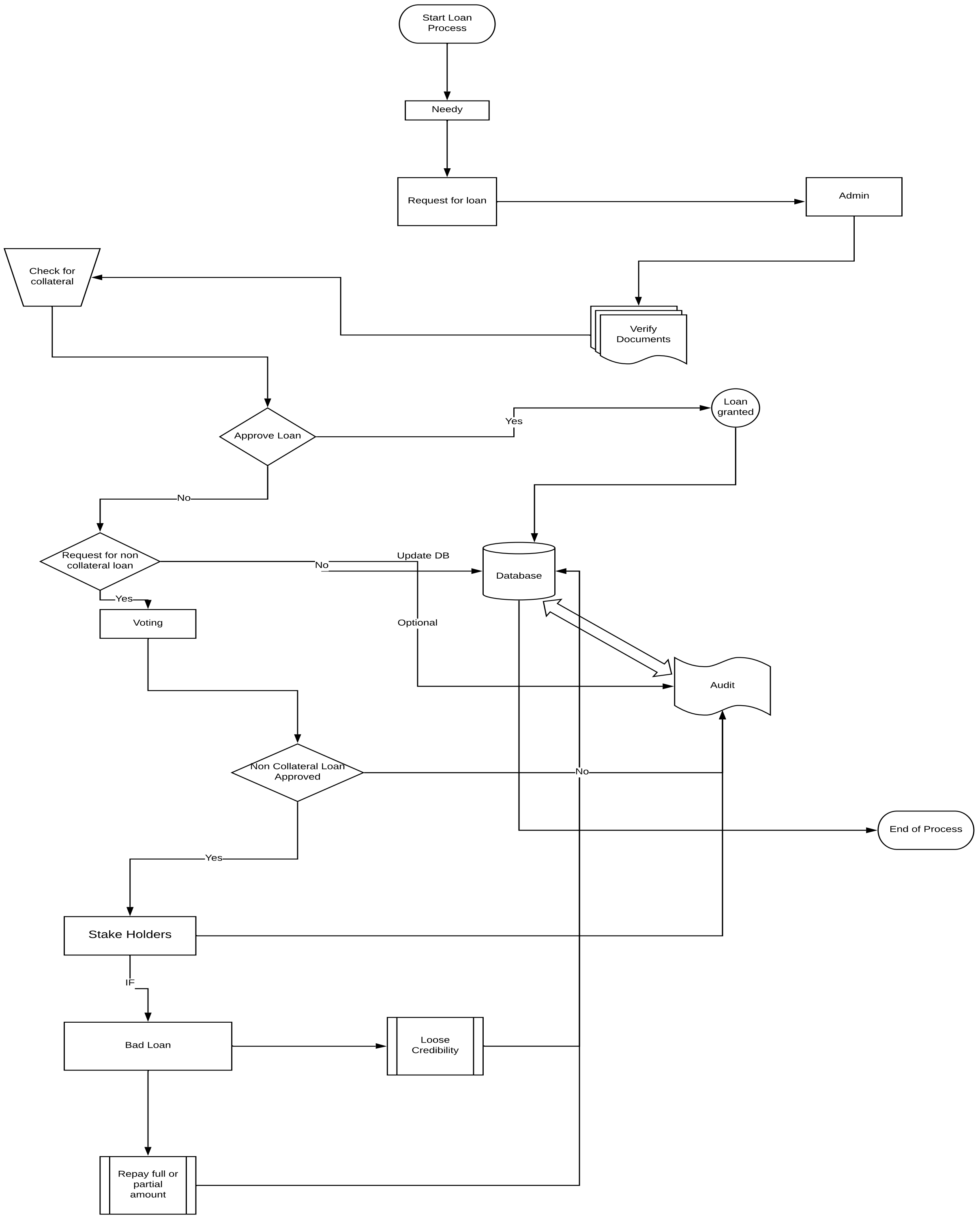
Blockchain solution makes the entire transaction transparent- readily available to view at any given point of time.

Elite has a right to vote – however they might need to hold responsibility for the same. This feature is again subjective – depending on the credibility of user. (Credibility can be defined by member since how long, amount invested, etc.)
The elite can also see – all the request which were made by needy – which we were approved, and which were rejected. However, personal information such as name or SSN are secured.
The elite may also demand – a in detail reason as to why a certain loan was rejected or certain amount was approved.

## Admin Perspective.

The admin may have time to time events in order to promote this kind of system. Also, it may request money from other pools within a geographical location or may also donate to other pools if necessary. This requires necessary approvals. Also, in certain cases it may need approval from central authority.

Fig: Decentralized loan system – Loan request and approval



## Fig: Decentralized loan system – Repayment of loan

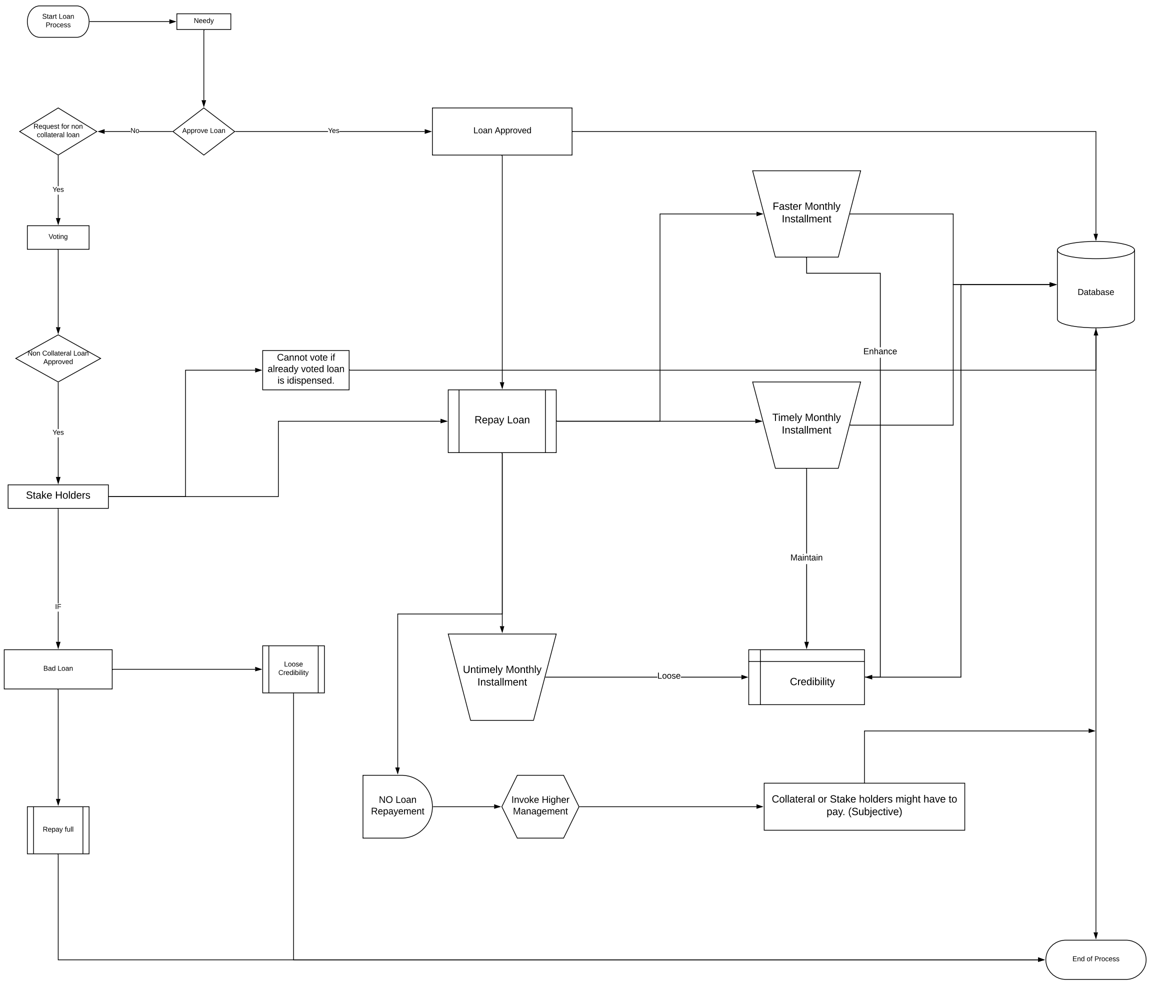
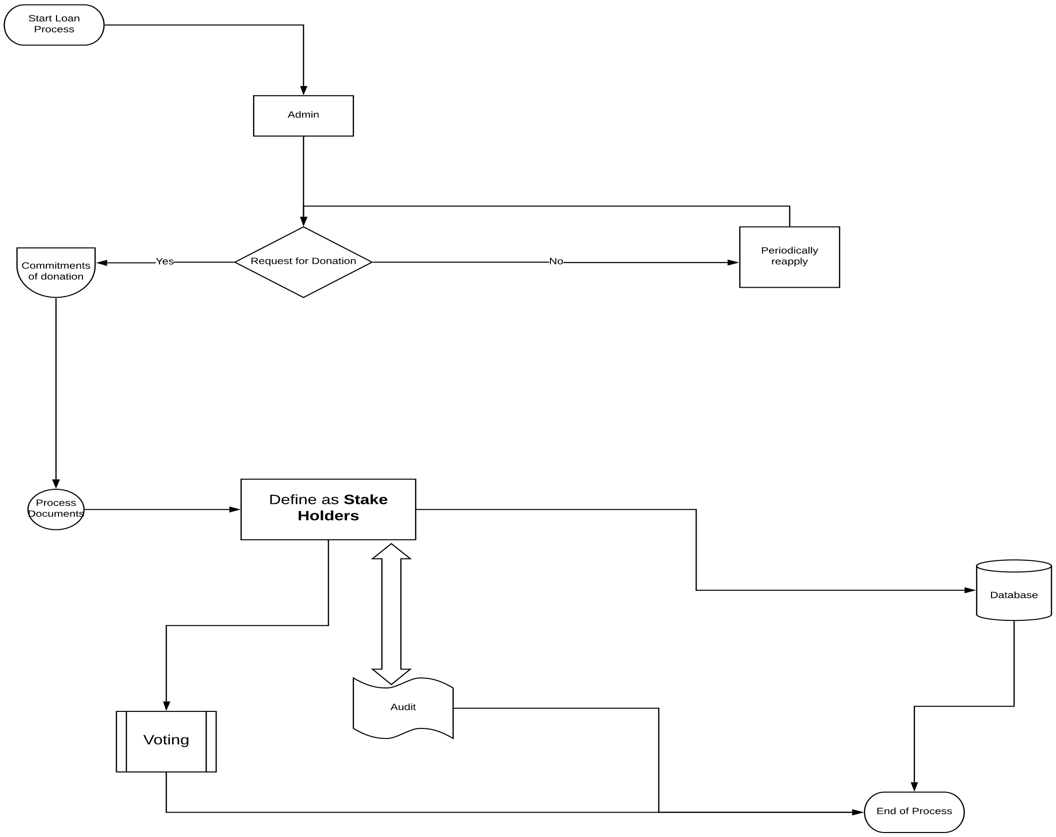
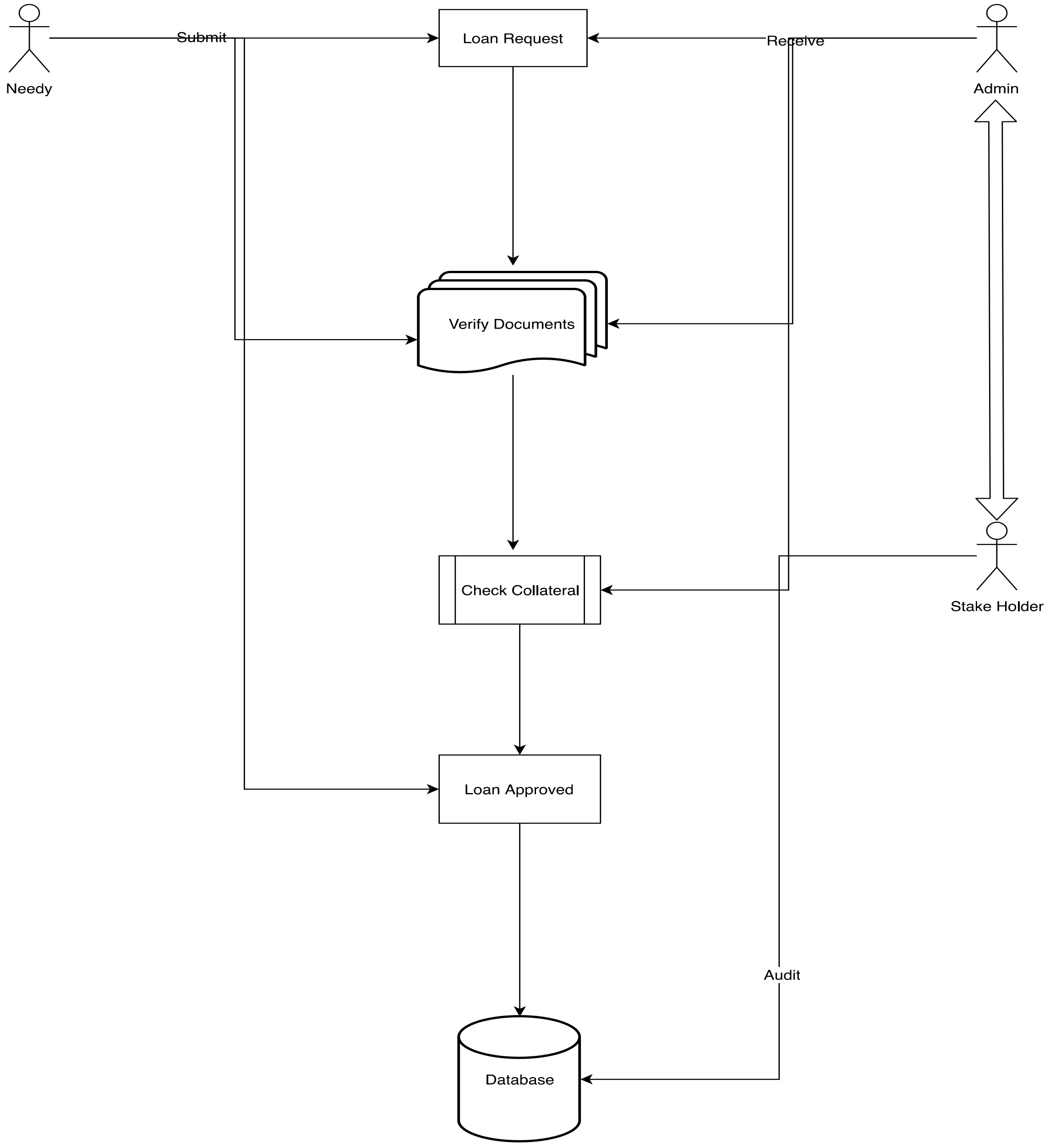

Start Loan Process
Needy
Request for non collateral loan
No
Approve Loan
Yes
Loan Approved
Yes
Voting
Faster Monthly Installment
Database
Non Collateral Loan Approved
Cannot vote if already voted loan is idispensed.
Enhance
Repay Loan
Timely Monthly Installment
Yes
Stake Holders
Maintain
IF
Bad Loan
Loose Credibility
Untimely Monthly Installment
Loose
Credibility
NO Loan Repayement
Invoke Higher Management
Collateral or Stake holders might have to pay. (Subjective)
Repay full
End of Process

Fig: Decentralized loan system – Donation

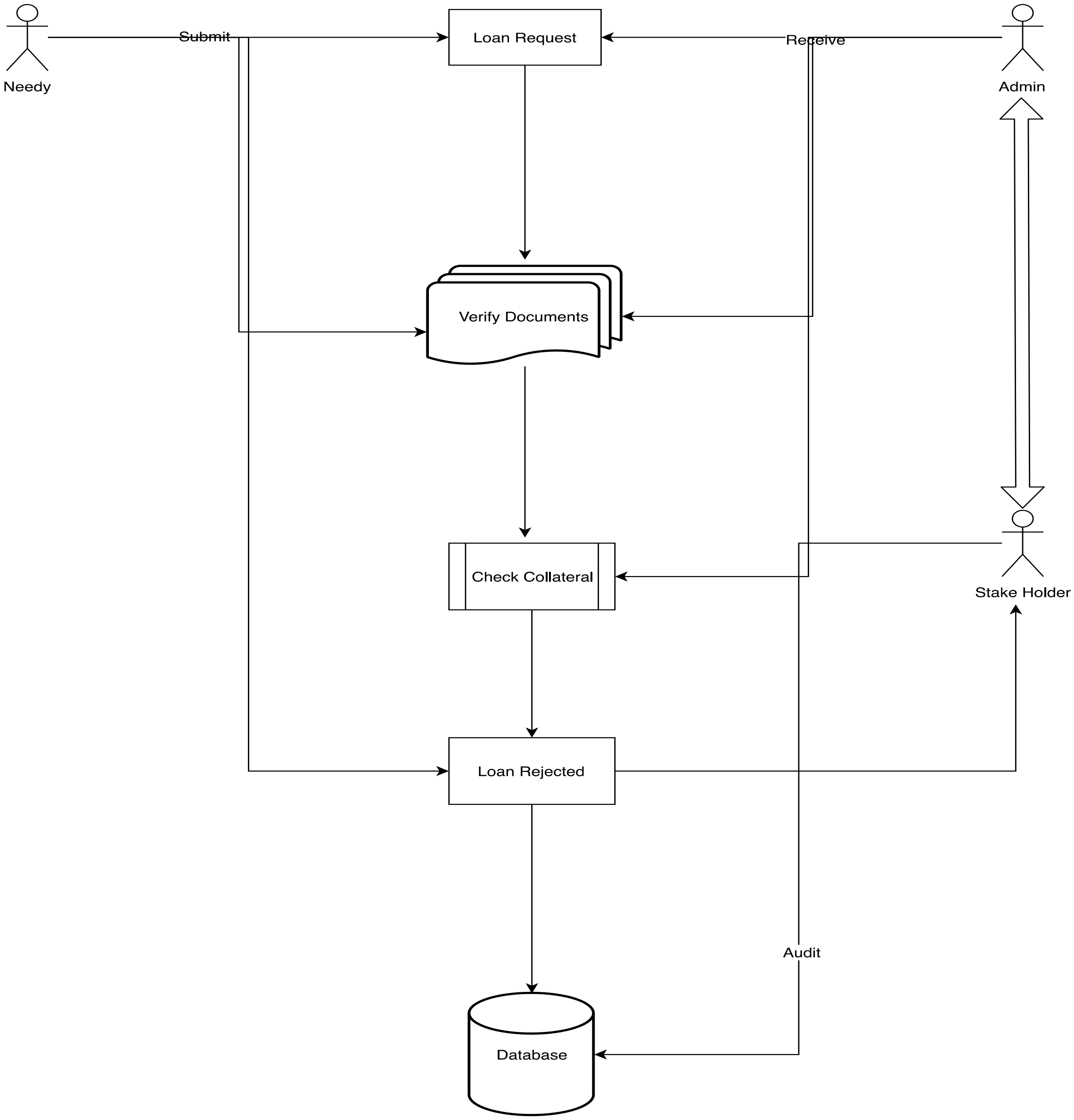


## 7. Use Case

### Case 1: Loan Approved

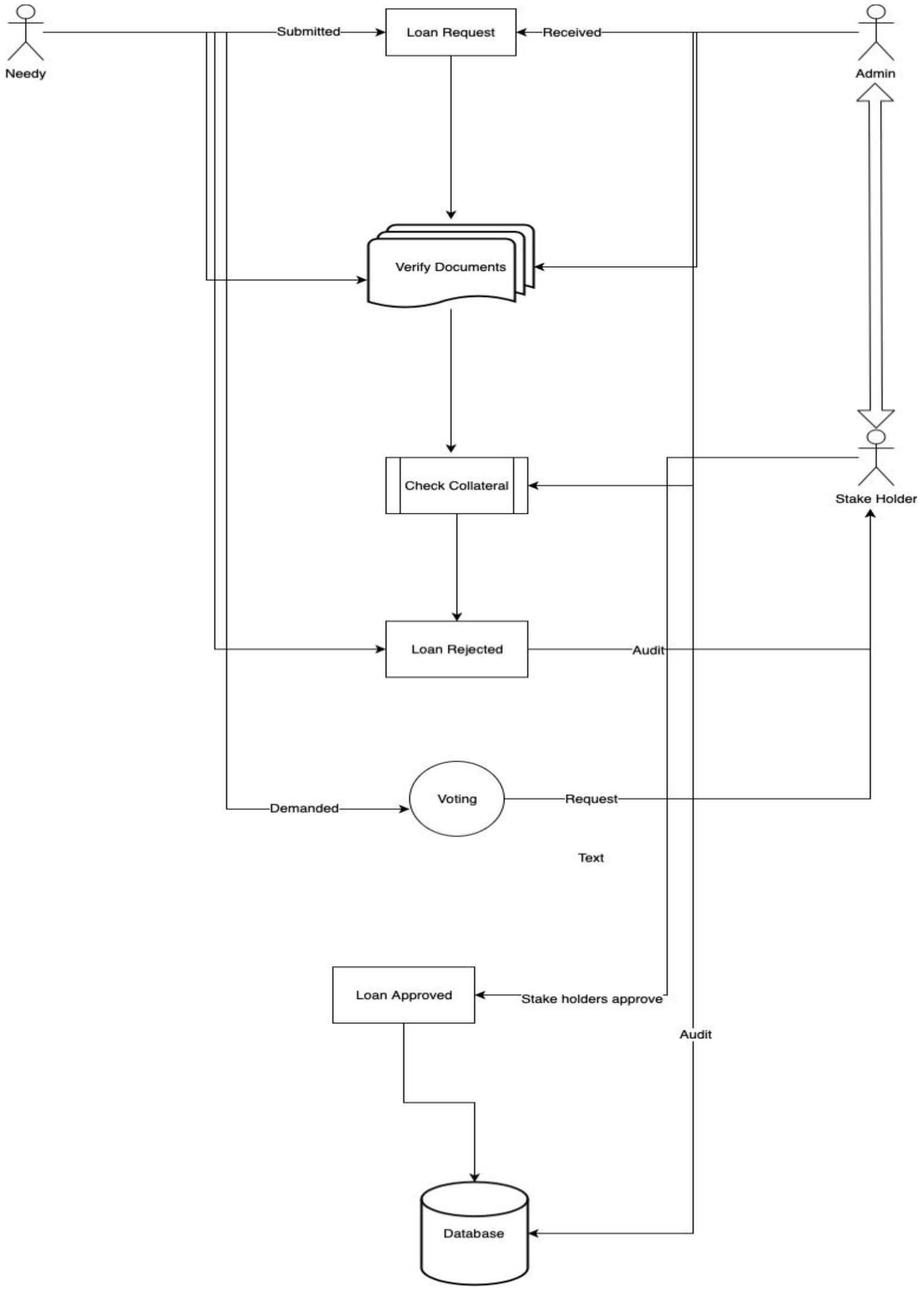


## Case 2: Loan Rejected

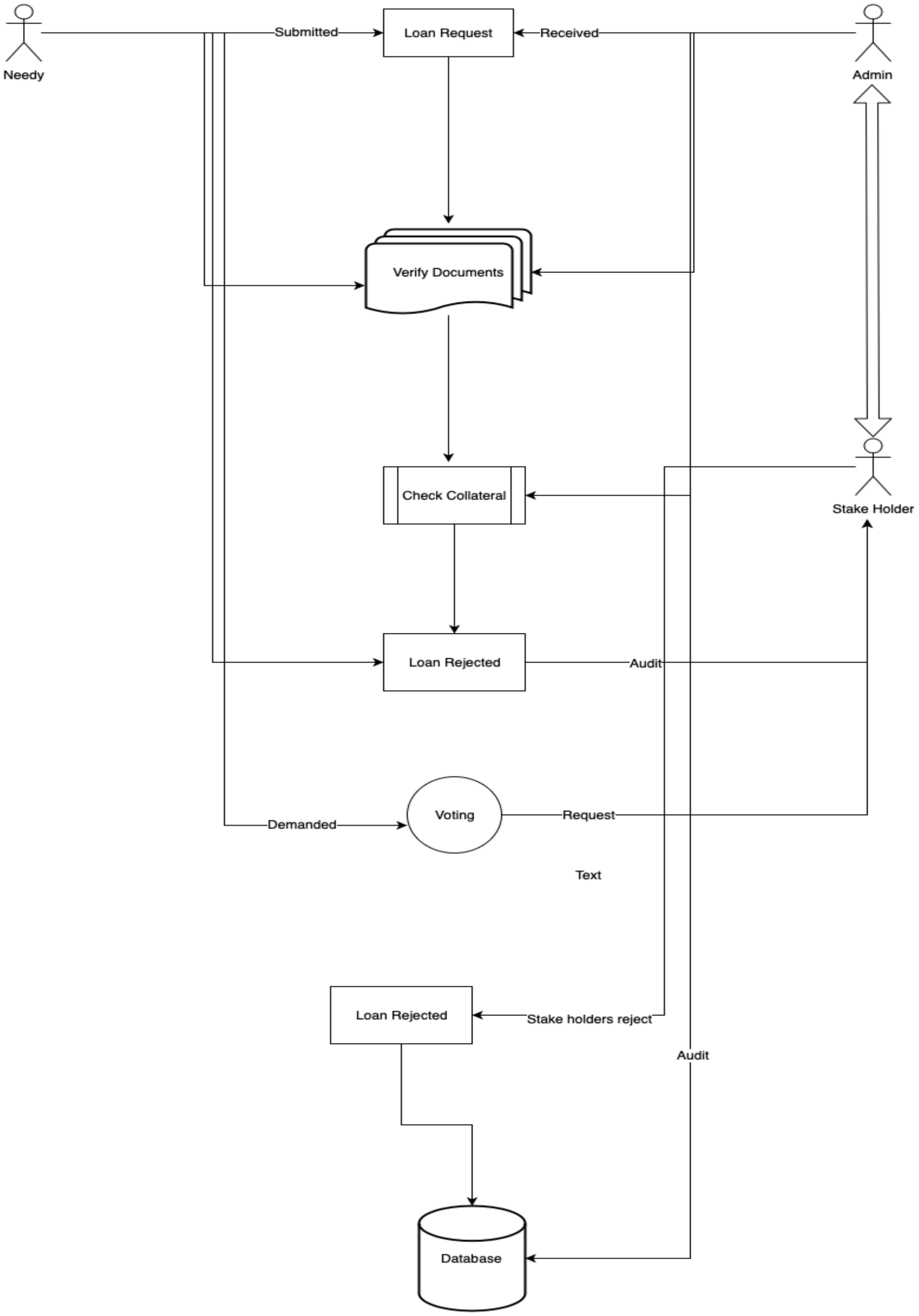


## Case 3: Loan Rejected – Voting Demanded

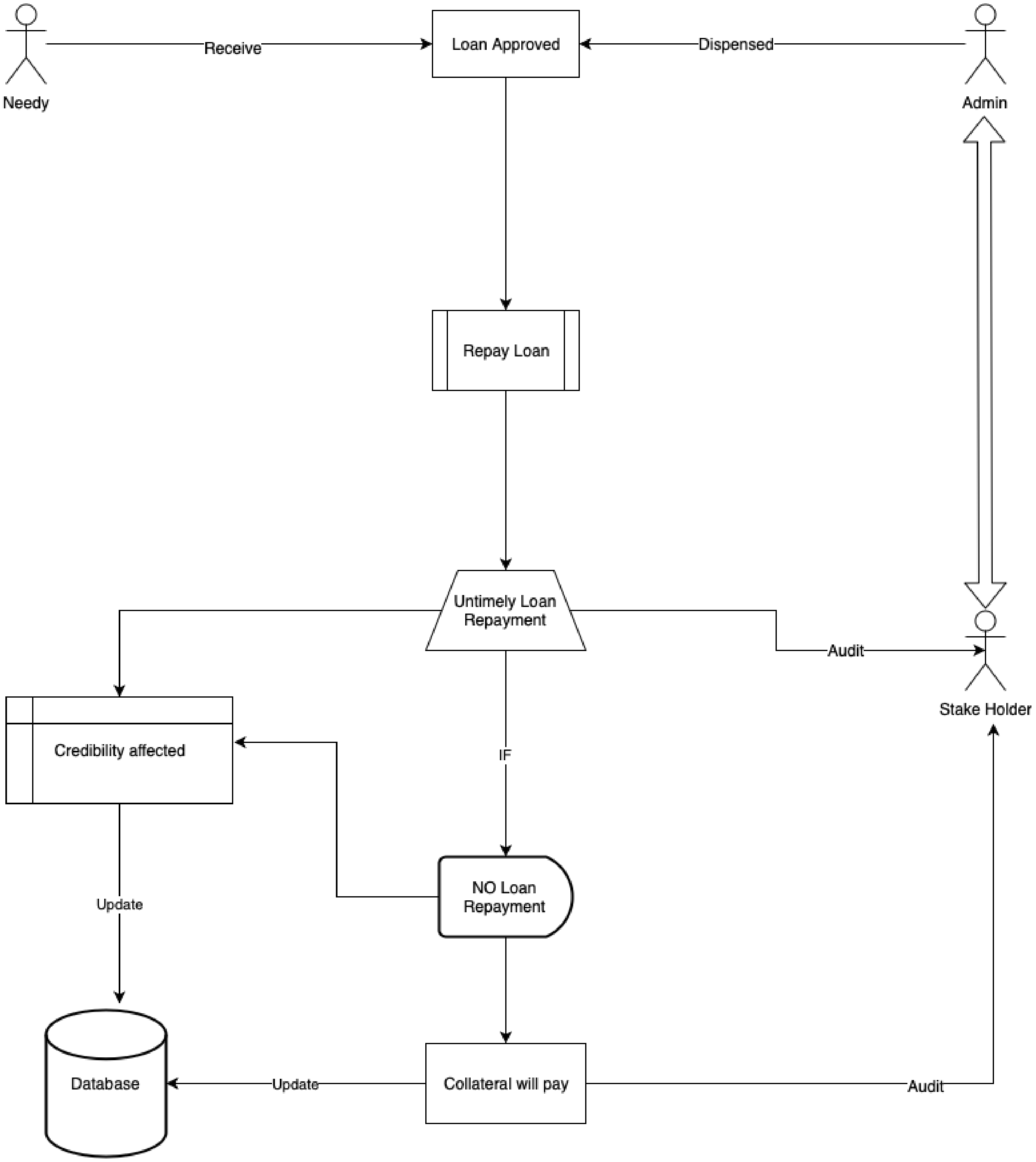
### Voting Accepted – Loan Approved



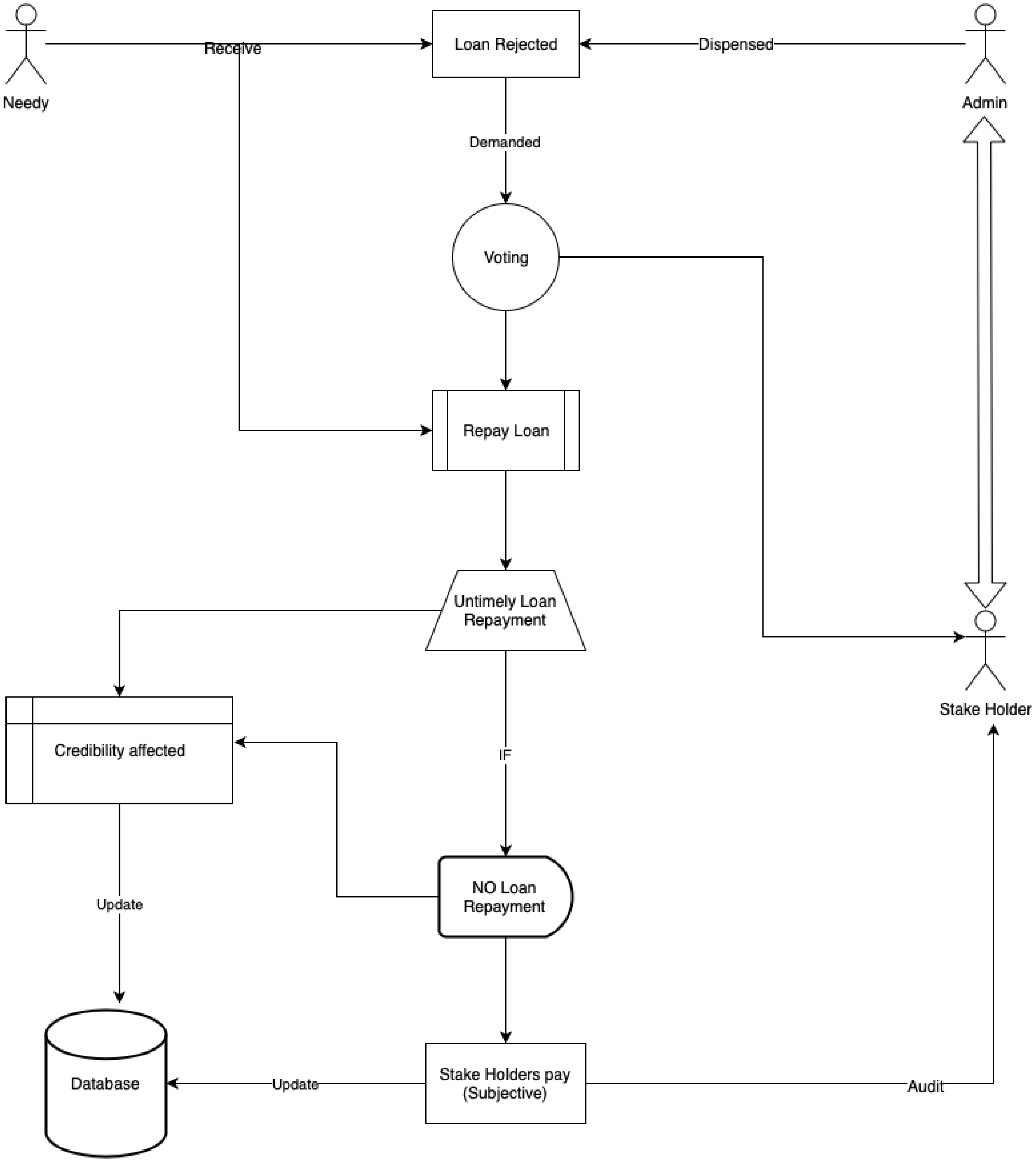
## Voting Accepted – Loan Rejected



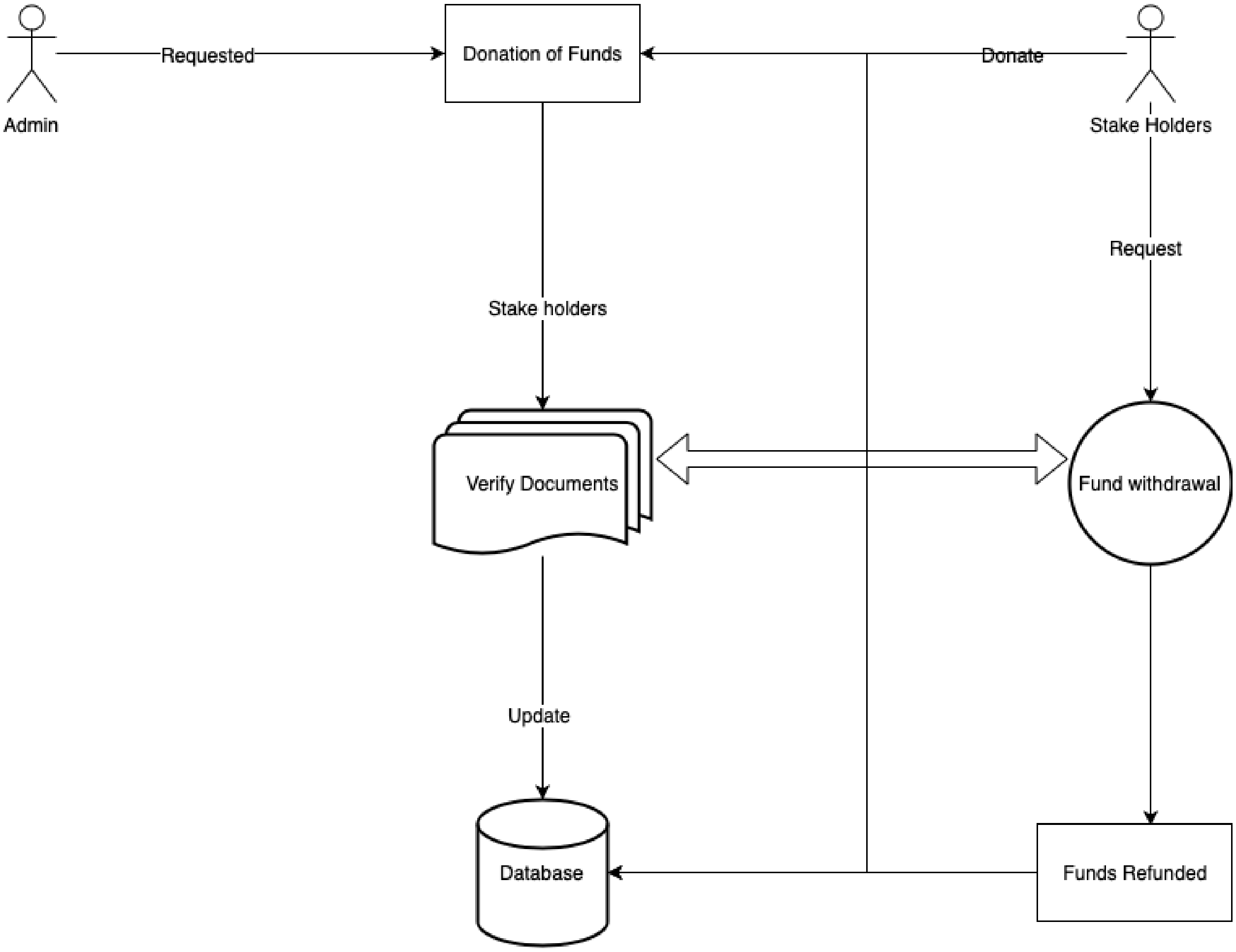
## Case 4: Loan Defaulted



## Case 5: Loan Defaulted after voting

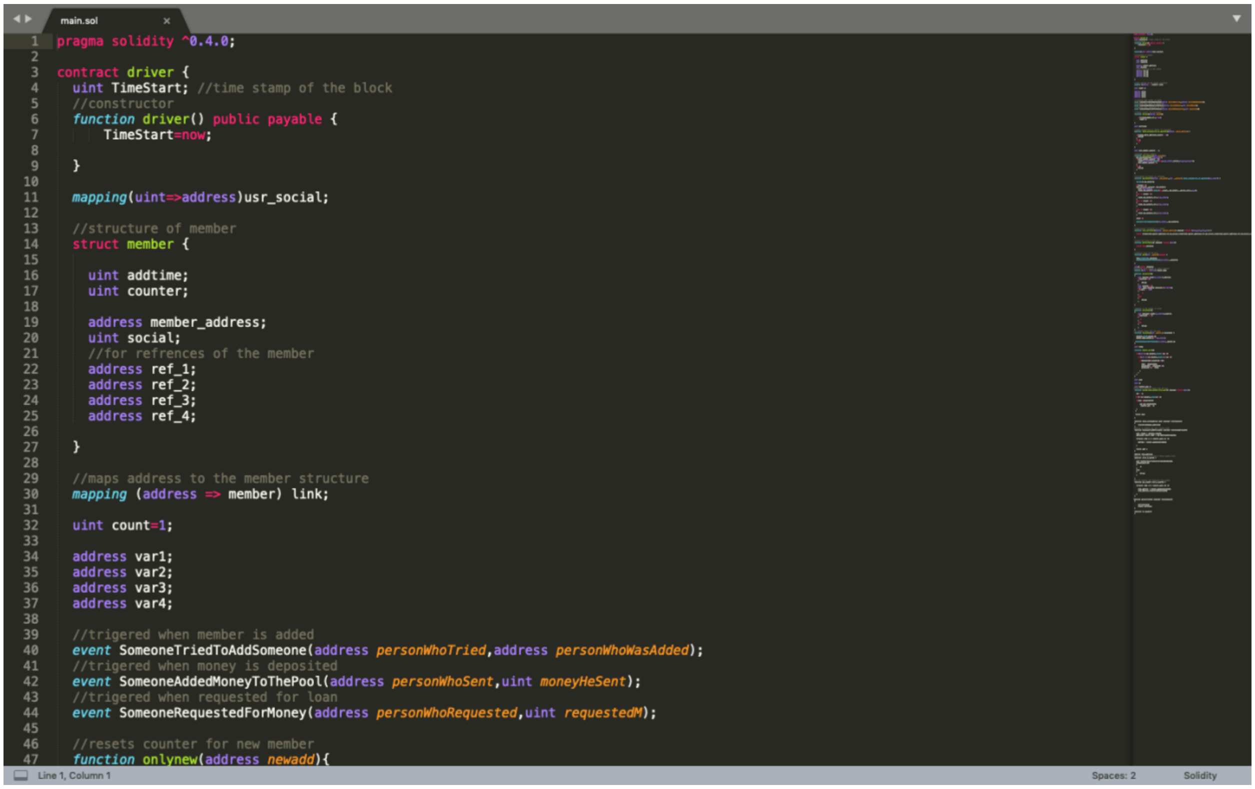


## Case 6: Donation / Withdrawal of funds

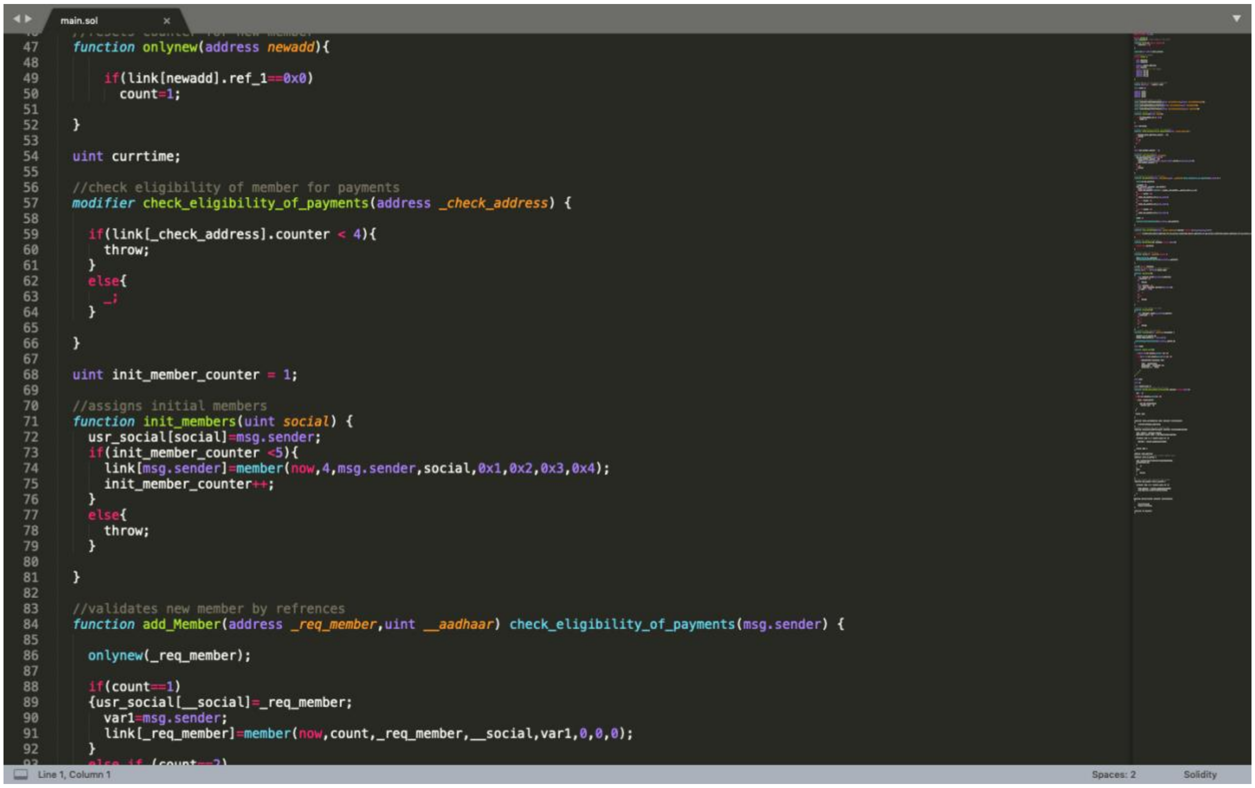


## 8. Software Codes

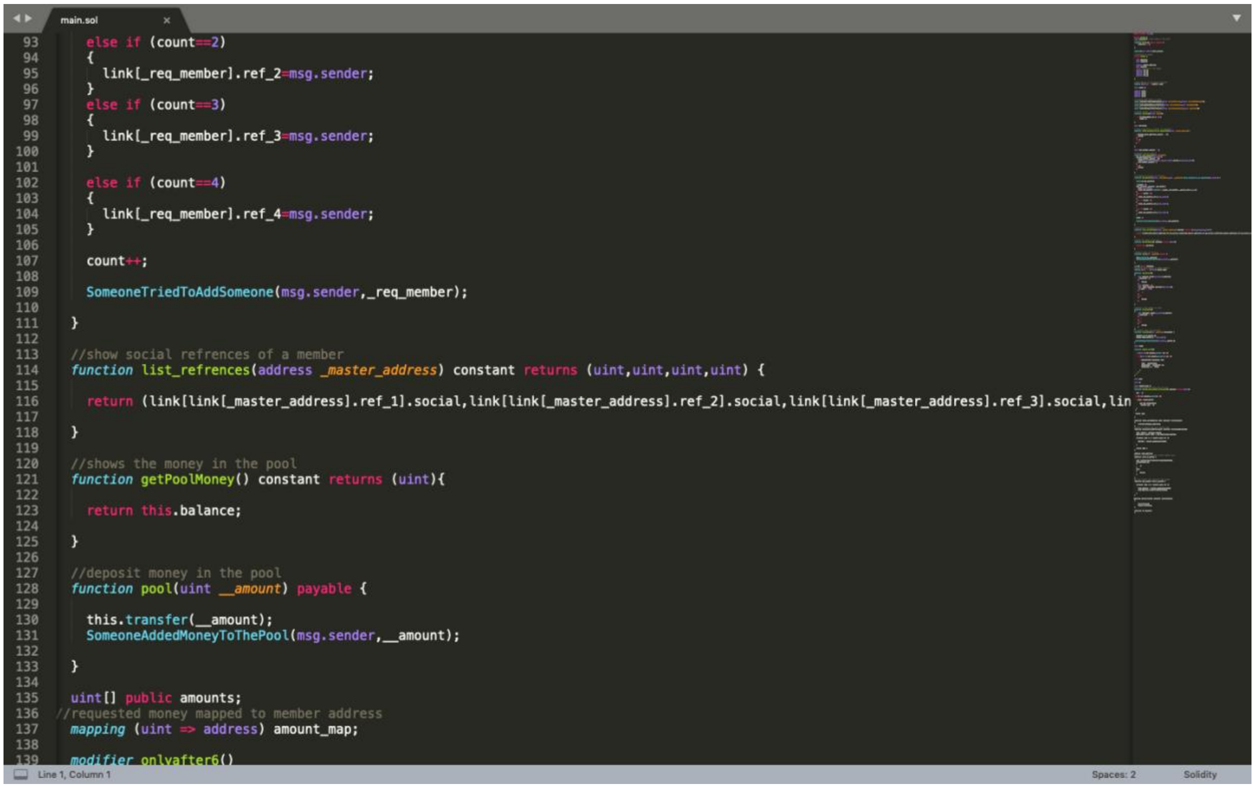
### Main Code – Decentralized Loan System



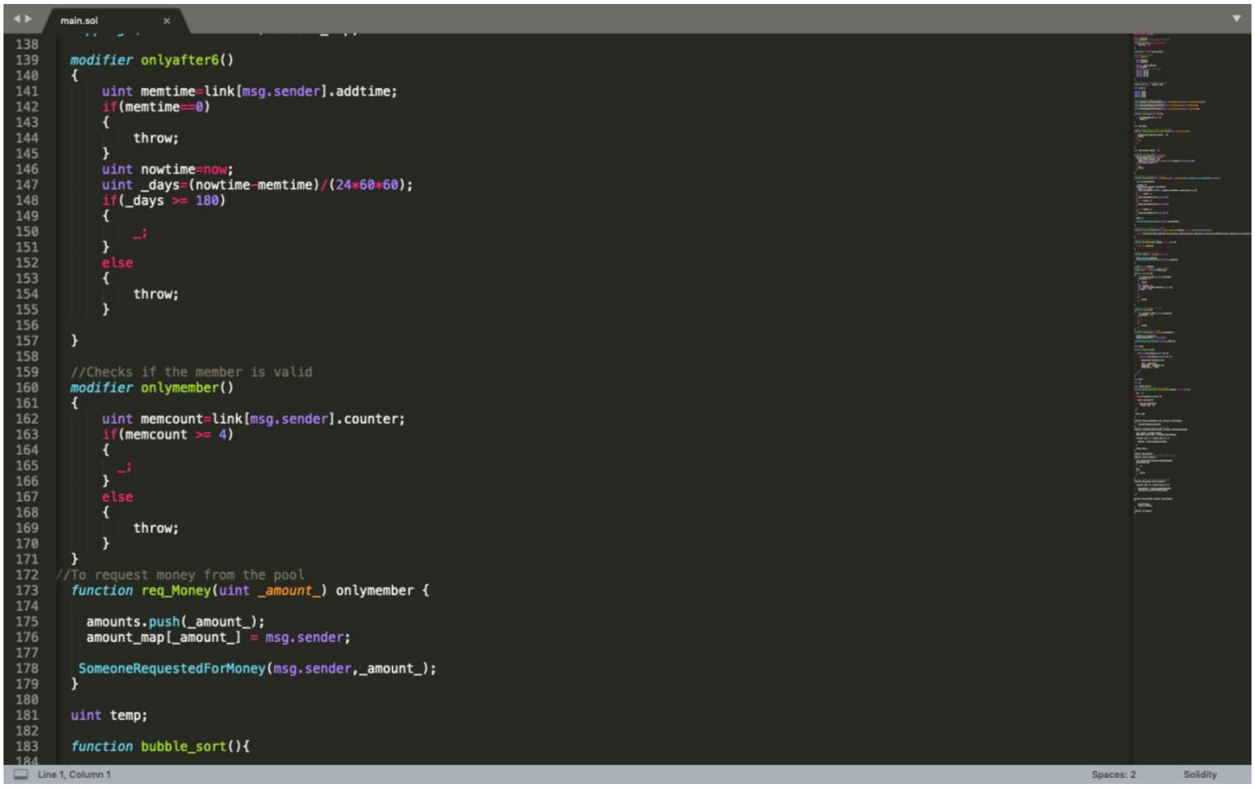
```solidity
pragma solidity ^0.4.0;

contract driver {
  uint TimeStart; //time stamp of the block
  //constructor
  function driver() public payable {
      TimeStart=now;

  }

  mapping(uint=>address)usr_social;

  //structure of member
  struct member {

    uint addtime;
    uint counter;

    address member_address;
    uint social;
    //for refrences of the member
    address ref_1;
    address ref_2;
    address ref_3;
    address ref_4;

  }

  //maps address to the member structure
  mapping (address => member) link;

  uint count=1;

  address var1;
  address var2;
  address var3;
  address var4;

  //trigered when member is added
  event SomeoneTriedToAddSomeone(address personWhoTried,address personWhoWasAdded);
  //trigered when money is deposited
  event SomeoneAddedMoneyToThePool(address personWhoSent,uint moneyHeSent);
  //trigered when requested for loan
  event SomeoneRequestedForMoney(address personWhoRequested,uint requestedM);

  //resets counter for new member
  function onlynew(address newadd){
```



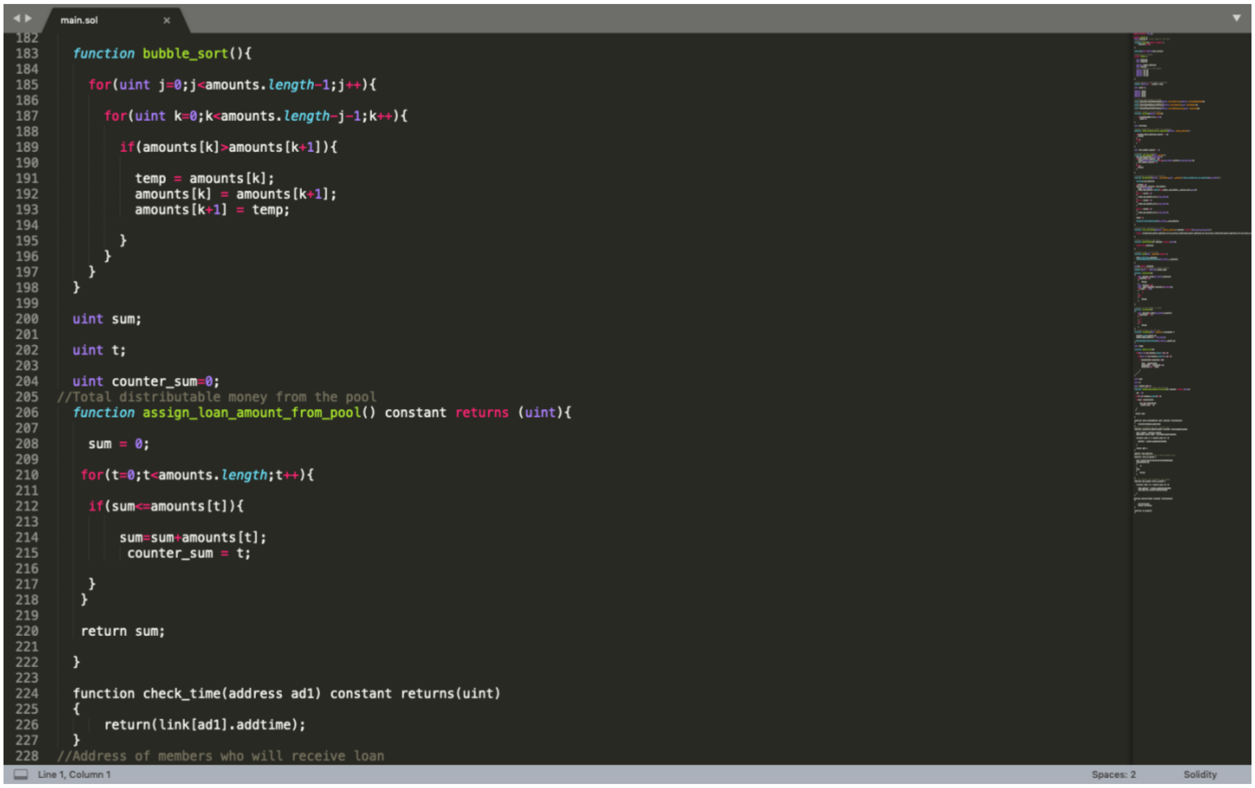
```
  //resets counter for new member
  function onlynew(address newadd){

      if(link[newadd].ref_1==0x0)
        count=1;

  }

  uint currtime;

  //check eligibility of member for payments
  modifier check_eligibility_of_payments(address _check_address) {

    if(link[_check_address].counter < 4){
      throw;
    }
    else{
      _;
    }

  }

  uint init_member_counter = 1;

  //assigns initial members
  function init_members(uint social) {
    usr_social[social]=msg.sender;
    if(init_member_counter <5){
      link[msg.sender]=member(now,4,msg.sender,social,0x1,0x2,0x3,0x4);
      init_member_counter++;
    }
    else{
      throw;
    }

  }

  //validates new member by refrences
  function add_Member(address _req_member,uint __aadhaar) check_eligibility_of_payments(msg.sender) {

    onlynew(_req_member);

    if(count==1)
    {usr_social[__social]=_req_member;
      var1=msg.sender;
      link[_req_member]=member(now,count,_req_member,__social,var1,0,0,0);
    }
    else if (count==2)
```



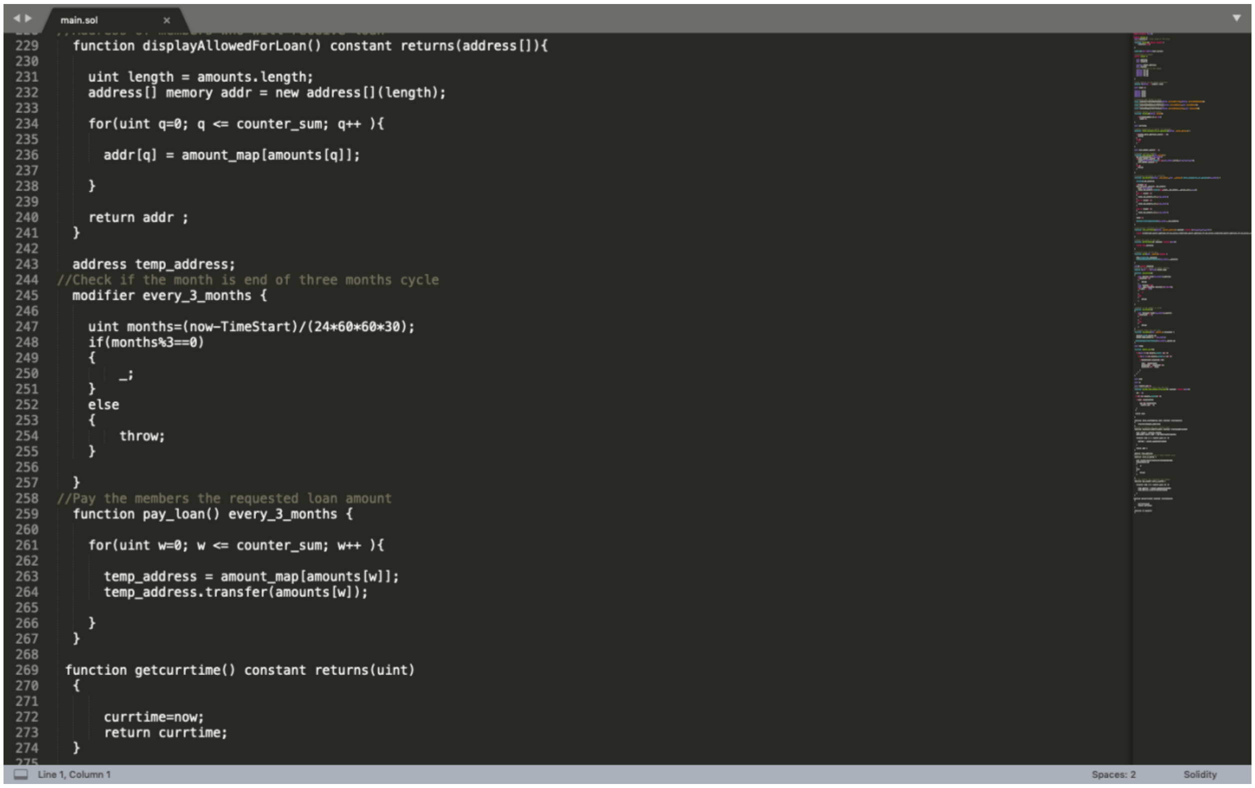
```
    else if (count==2)
    {
      link[_req_member].ref_2=msg.sender;
    }
    else if (count==3)
    {
      link[_req_member].ref_3=msg.sender;
    }

    else if (count==4)
    {
      link[_req_member].ref_4=msg.sender;
    }

    count++;

    SomeoneTriedToAddSomeone(msg.sender,_req_member);

  }

  //show social refrences of a member
  function list_refrences(address _master_address) constant returns (uint,uint,uint,uint) {

    return (link[link[_master_address].ref_1].social,link[link[_master_address].ref_2].social,link[link[_master_address].ref_3].social,lin

  }

  //shows the money in the pool
  function getPoolMoney() constant returns (uint){

    return this.balance;

  }

  //deposit money in the pool
  function pool(uint __amount) payable {

    this.transfer(__amount);
    SomeoneAddedMoneyToThePool(msg.sender,__amount);

  }

  uint[] public amounts;
//requested money mapped to member address
  mapping (uint => address) amount_map;

  modifier onlyafter6()
```



```
  modifier onlyafter6()
  {
      uint memtime=link[msg.sender].addtime;
      if(memtime==0)
      {
          throw;
      }
      uint nowtime=now;
      uint _days=(nowtime-memtime)/(24*60*60);
      if(_days >= 180)
      {
          _;
      }
      else
      {
          throw;
      }

  }

  //Checks if the member is valid
  modifier onlymember()
  {
      uint memcount=link[msg.sender].counter;
      if(memcount >= 4)
      {
        _;
      }
      else
      {
          throw;
      }
  }
//To request money from the pool
  function req_Money(uint _amount_) onlymember {

    amounts.push(_amount_);
    amount_map[_amount_] = msg.sender;

   SomeoneRequestedForMoney(msg.sender,_amount_);
  }

  uint temp;

  function bubble_sort(){
```

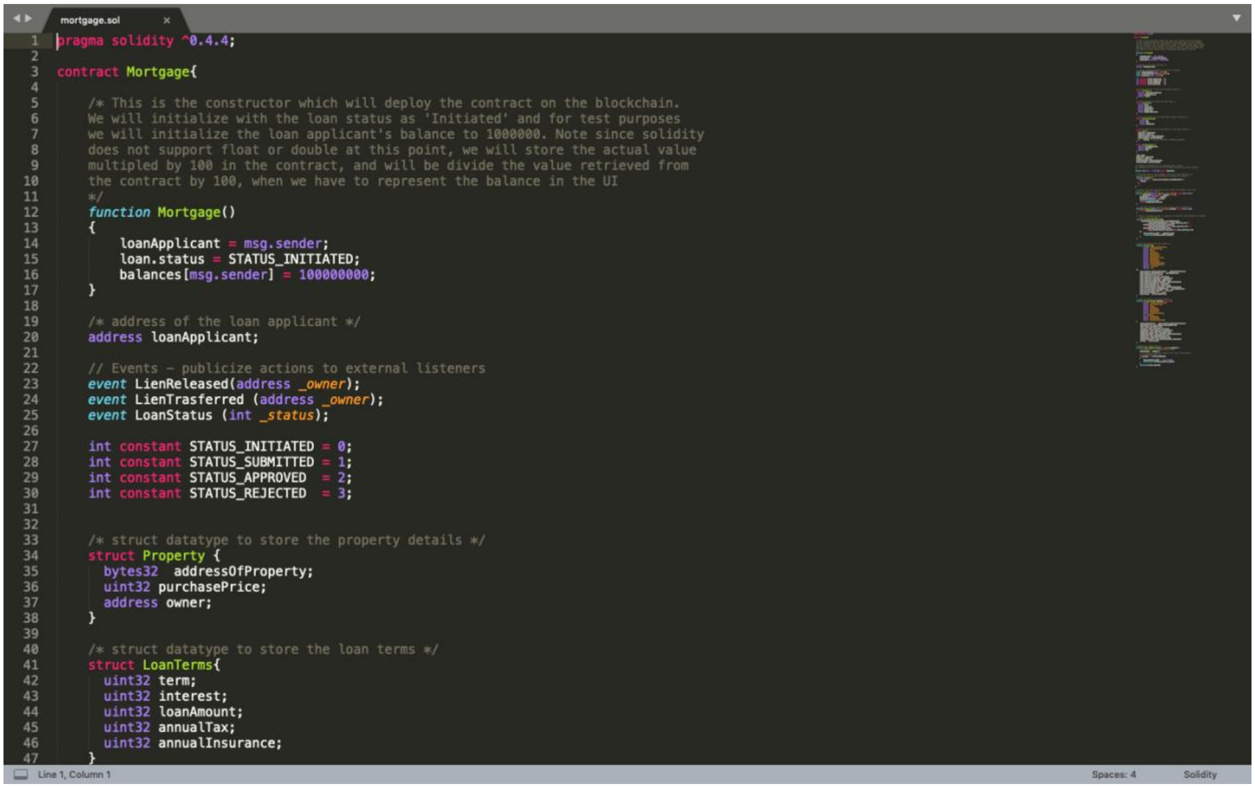


```

  function bubble_sort(){

    for(uint j=0;j<amounts.length-1;j++){

      for(uint k=0;k<amounts.length-j-1;k++){

        if(amounts[k]>amounts[k+1]){

          temp = amounts[k];
          amounts[k] = amounts[k+1];
          amounts[k+1] = temp;

        }
      }
    }
  }

  uint sum;

  uint t;

  uint counter_sum=0;
//Total distributable money from the pool
  function assign_loan_amount_from_pool() constant returns (uint){

    sum = 0;

   for(t=0;t<amounts.length;t++){

    if(sum<=amounts[t]){

        sum=sum+amounts[t];
         counter_sum = t;

    }
   }

   return sum;

  }

  function check_time(address ad1) constant returns(uint)
  {
      return(link[ad1].addtime);
  }
//Address of members who will receive loan
```

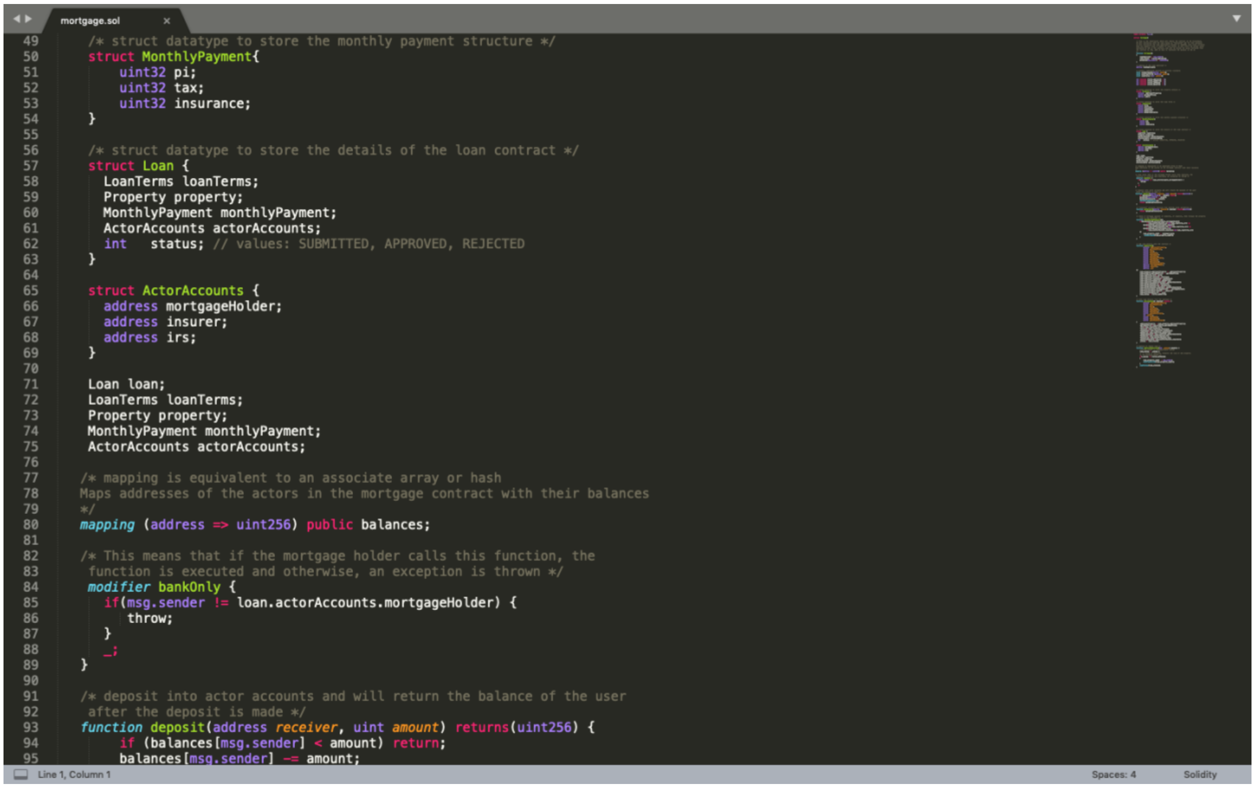


```
  function displayAllowedForLoan() constant returns(address[]){

    uint length = amounts.length;
    address[] memory addr = new address[](length);

    for(uint q=0; q <= counter_sum; q++ ){

      addr[q] = amount_map[amounts[q]];

    }

    return addr ;
  }

  address temp_address;
//Check if the month is end of three months cycle
  modifier every_3_months {

    uint months=(now-TimeStart)/(24*60*60*30);
    if(months%3==0)
    {
        _;
    }
    else
    {
        throw;
    }

  }
//Pay the members the requested loan amount
  function pay_loan() every_3_months {

    for(uint w=0; w <= counter_sum; w++ ){

      temp_address = amount_map[amounts[w]];
      temp_address.transfer(amounts[w]);

    }
  }

 function getcurrtime() constant returns(uint)
  {

      currtime=now;
      return currtime;
  }
```

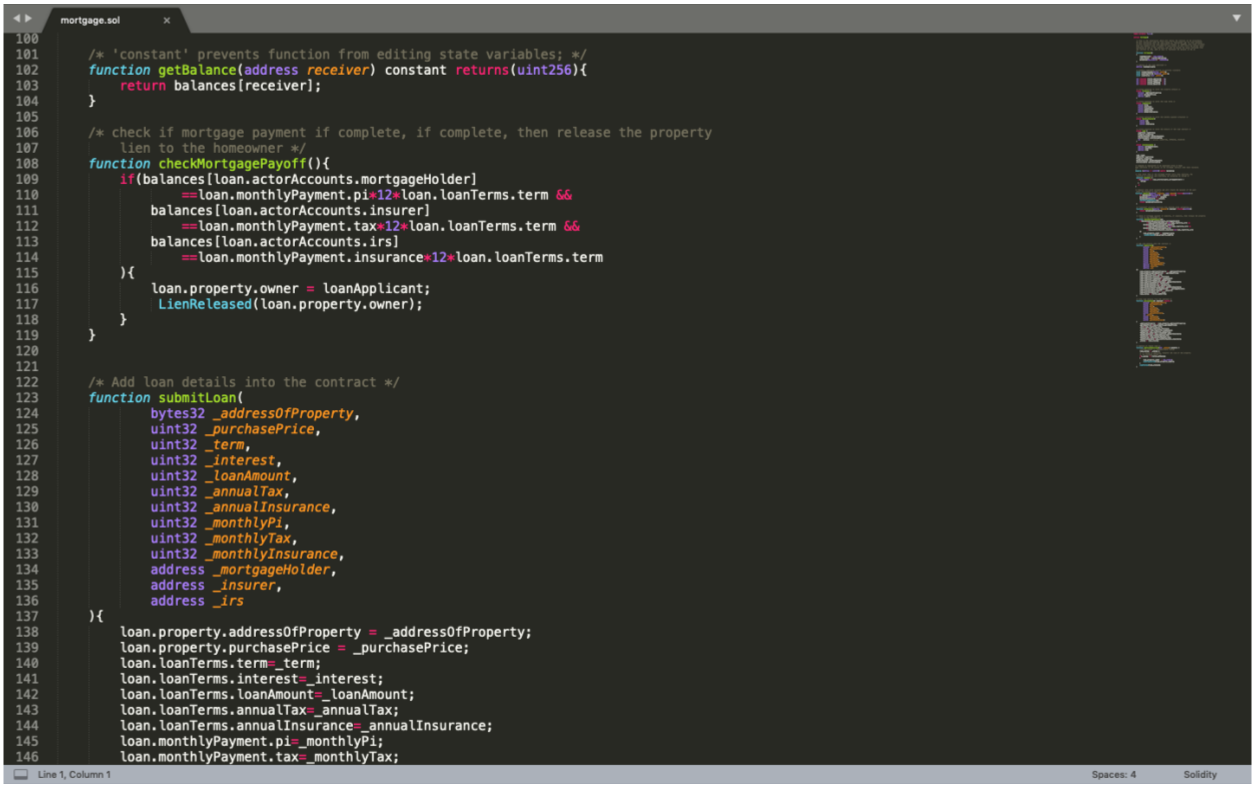
## Mortgage Code (Collateral) – Decentralized Loan System



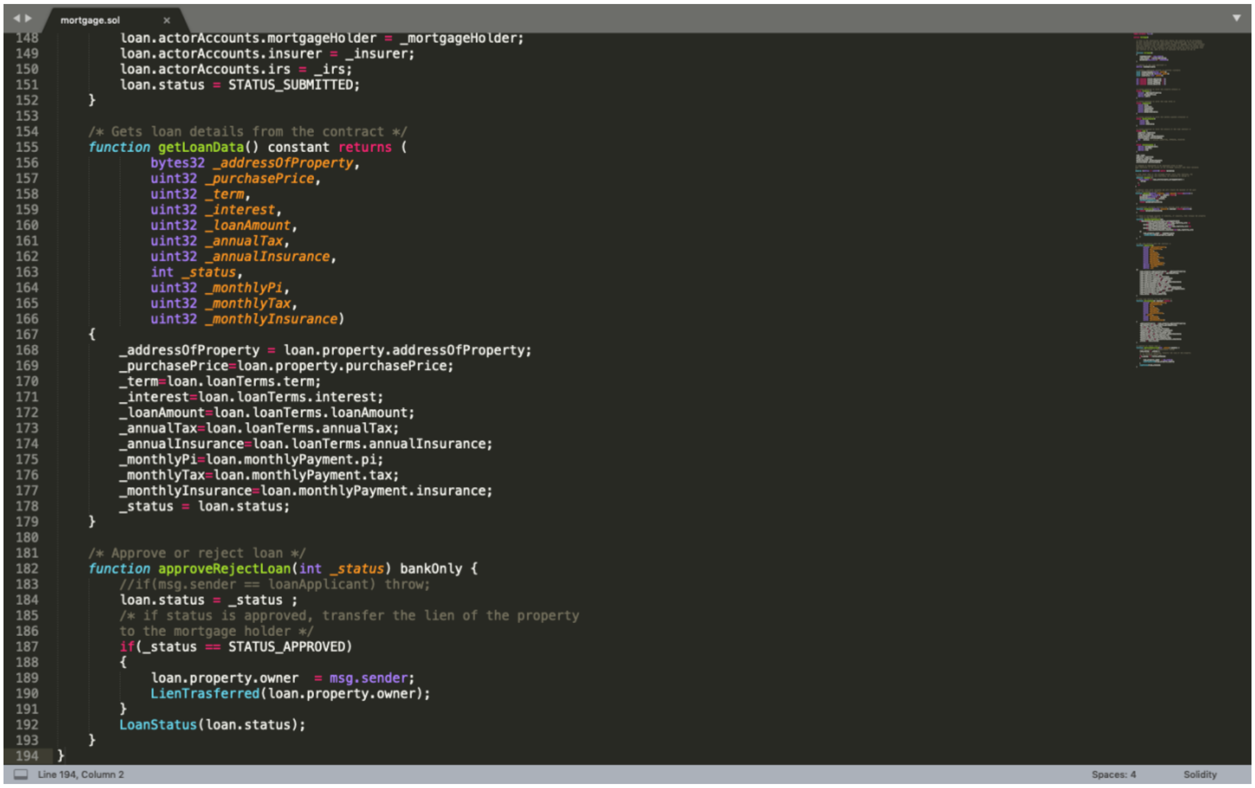
```
pragma solidity ^0.4.4;

contract Mortgage{

    /* This is the constructor which will deploy the contract on the blockchain.
    We will initialize with the loan status as 'Initiated' and for test purposes
    we will initialize the loan applicant's balance to 1000000. Note since solidity
    does not support float or double at this point, we will store the actual value
    multipled by 100 in the contract, and will be divide the value retrieved from
    the contract by 100, when we have to represent the balance in the UI
    */
    function Mortgage()
    {
        loanApplicant = msg.sender;
        loan.status = STATUS_INITIATED;
        balances[msg.sender] = 100000000;
    }

    /* address of the loan applicant */
    address loanApplicant;

    // Events - publicize actions to external listeners
    event LienReleased(address _owner);
    event LienTrasferred (address _owner);
    event LoanStatus (int _status);

    int constant STATUS_INITIATED = 0;
    int constant STATUS_SUBMITTED = 1;
    int constant STATUS_APPROVED  = 2;
    int constant STATUS_REJECTED  = 3;


    /* struct datatype to store the property details */
    struct Property {
      bytes32  addressOfProperty;
      uint32 purchasePrice;
      address owner;
    }

    /* struct datatype to store the loan terms */
    struct LoanTerms{
      uint32 term;
      uint32 interest;
      uint32 loanAmount;
      uint32 annualTax;
      uint32 annualInsurance;
    }
```



```
    /* struct datatype to store the monthly payment structure */
    struct MonthlyPayment{
        uint32 pi;
        uint32 tax;
        uint32 insurance;
    }

    /* struct datatype to store the details of the loan contract */
    struct Loan {
      LoanTerms loanTerms;
      Property property;
      MonthlyPayment monthlyPayment;
      ActorAccounts actorAccounts;
      int   status; // values: SUBMITTED, APPROVED, REJECTED
    }

    struct ActorAccounts {
      address mortgageHolder;
      address insurer;
      address irs;
    }

    Loan loan;
    LoanTerms loanTerms;
    Property property;
    MonthlyPayment monthlyPayment;
    ActorAccounts actorAccounts;

   /* mapping is equivalent to an associate array or hash
   Maps addresses of the actors in the mortgage contract with their balances
   */
   mapping (address => uint256) public balances;

   /* This means that if the mortgage holder calls this function, the
    function is executed and otherwise, an exception is thrown */
    modifier bankOnly {
      if(msg.sender != loan.actorAccounts.mortgageHolder) {
         throw;
      }
      _;
   }

   /* deposit into actor accounts and will return the balance of the user
    after the deposit is made */
   function deposit(address receiver, uint amount) returns(uint256) {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
```

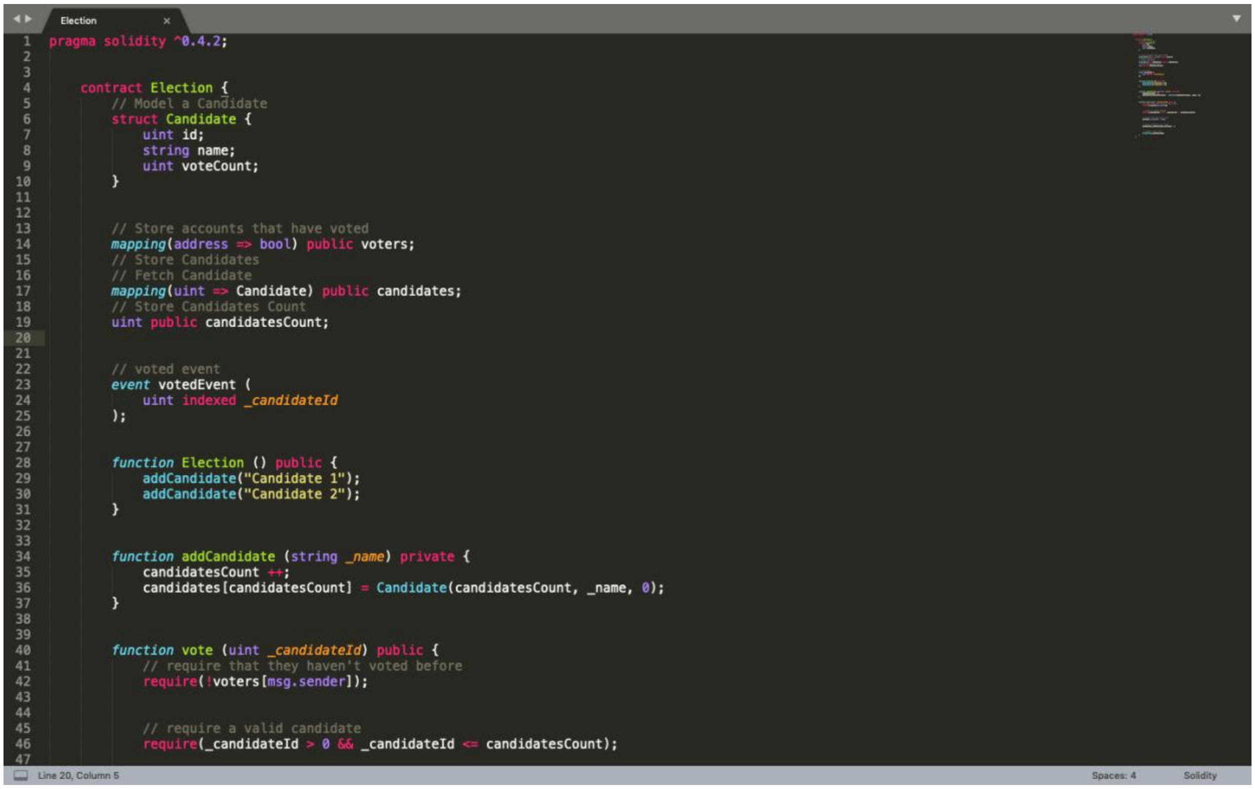


```

    /* 'constant' prevents function from editing state variables; */
    function getBalance(address receiver) constant returns(uint256){
        return balances[receiver];
    }

    /* check if mortgage payment if complete, if complete, then release the property
        lien to the homeowner */
    function checkMortgagePayoff(){
        if(balances[loan.actorAccounts.mortgageHolder]
                ==loan.monthlyPayment.pi*12*loan.loanTerms.term &&
            balances[loan.actorAccounts.insurer]
                ==loan.monthlyPayment.tax*12*loan.loanTerms.term &&
            balances[loan.actorAccounts.irs]
                ==loan.monthlyPayment.insurance*12*loan.loanTerms.term
        ){
            loan.property.owner = loanApplicant;
             LienReleased(loan.property.owner);
        }
    }


    /* Add loan details into the contract */
    function submitLoan(
            bytes32 _addressOfProperty,
            uint32 _purchasePrice,
            uint32 _term,
            uint32 _interest,
            uint32 _loanAmount,
            uint32 _annualTax,
            uint32 _annualInsurance,
            uint32 _monthlyPi,
            uint32 _monthlyTax,
            uint32 _monthlyInsurance,
            address _mortgageHolder,
            address _insurer,
            address _irs
    ){
        loan.property.addressOfProperty = _addressOfProperty;
        loan.property.purchasePrice = _purchasePrice;
        loan.loanTerms.term=_term;
        loan.loanTerms.interest=_interest;
        loan.loanTerms.loanAmount=_loanAmount;
        loan.loanTerms.annualTax=_annualTax;
        loan.loanTerms.annualInsurance=_annualInsurance;
        loan.monthlyPayment.pi=_monthlyPi;
        loan.monthlyPayment.tax=_monthlyTax;
```



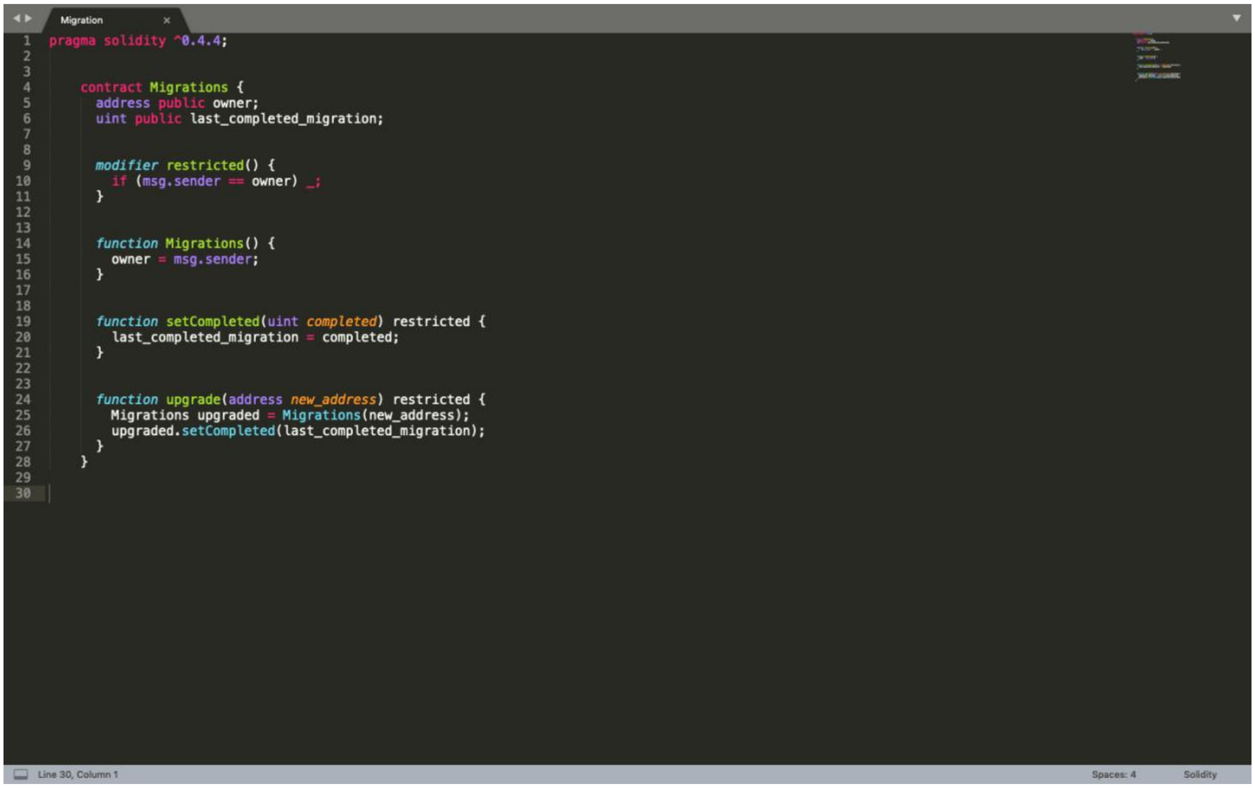
```
        loan.actorAccounts.mortgageHolder = _mortgageHolder;
        loan.actorAccounts.insurer = _insurer;
        loan.actorAccounts.irs = _irs;
        loan.status = STATUS_SUBMITTED;
    }

    /* Gets loan details from the contract */
    function getLoanData() constant returns (
            bytes32 _addressOfProperty,
            uint32 _purchasePrice,
            uint32 _term,
            uint32 _interest,
            uint32 _loanAmount,
            uint32 _annualTax,
            uint32 _annualInsurance,
            int _status,
            uint32 _monthlyPi,
            uint32 _monthlyTax,
            uint32 _monthlyInsurance)
    {
        _addressOfProperty = loan.property.addressOfProperty;
        _purchasePrice=loan.property.purchasePrice;
        _term=loan.loanTerms.term;
        _interest=loan.loanTerms.interest;
        _loanAmount=loan.loanTerms.loanAmount;
        _annualTax=loan.loanTerms.annualTax;
        _annualInsurance=loan.loanTerms.annualInsurance;
        _monthlyPi=loan.monthlyPayment.pi;
        _monthlyTax=loan.monthlyPayment.tax;
        _monthlyInsurance=loan.monthlyPayment.insurance;
        _status = loan.status;
    }

    /* Approve or reject loan */
    function approveRejectLoan(int _status) bankOnly {
        //if(msg.sender == loanApplicant) throw;
        loan.status = _status ;
        /* if status is approved, transfer the lien of the property
        to the mortgage holder */
        if(_status == STATUS_APPROVED)
        {
            loan.property.owner  = msg.sender;
            LienTrasferred(loan.property.owner);
        }
        LoanStatus(loan.status);
    }
}
```

## Election Code – Decentralized Loan System



```
pragma solidity ^0.4.2;


    contract Election {
        // Model a Candidate
        struct Candidate {
            uint id;
            string name;
            uint voteCount;
        }


        // Store accounts that have voted
        mapping(address => bool) public voters;
        // Store Candidates
        // Fetch Candidate
        mapping(uint => Candidate) public candidates;
        // Store Candidates Count
        uint public candidatesCount;


        // voted event
        event votedEvent (
            uint indexed _candidateId
        );


        function Election () public {
            addCandidate("Candidate 1");
            addCandidate("Candidate 2");
        }


        function addCandidate (string _name) private {
            candidatesCount ++;
            candidates[candidatesCount] = Candidate(candidatesCount, _name, 0);
        }


        function vote (uint _candidateId) public {
            // require that they haven't voted before
            require(!voters[msg.sender]);


            // require a valid candidate
            require(_candidateId > 0 && _candidateId <= candidatesCount);

```

## Migration Code – Decentralized Loan System



```
pragma solidity ^0.4.4;


    contract Migrations {
      address public owner;
      uint public last_completed_migration;


      modifier restricted() {
        if (msg.sender == owner) _;
      }


      function Migrations() {
        owner = msg.sender;
      }


      function setCompleted(uint completed) restricted {
        last_completed_migration = completed;
      }


      function upgrade(address new_address) restricted {
        Migrations upgraded = Migrations(new_address);
        upgraded.setCompleted(last_completed_migration);
      }
    }

```

## Future Scope

There are two improvements which can be made in the near future which will facilitate more and more participants to take benefit of the system.

1. Increase the trust between stake holders so that more and more participants contribute money into the pool.

2. Stake holders should be incentivized so that they keep money for much longer period of time.

Consider a scenario which may be implemented in future.

If person A – donates money in the pool recursively over a long period of time. At the end of the tenure the amount is now $1,00,000. Now person A requires the said amount for his child (B's) education. However instead of withdrawing the money from the pool – person A takes a loan of the said amount from organization. Now, the said amount of person A acts like a collateral for the loan. However, while the loan is not completely repaid – person A does not enjoy the right to vote. After B completes his study – he can repay the loan as defined by the guidelines of the institution. After the completion of the tenure – the lock upon A's amount is withdrawn and A can enjoy his amount.
Hence, at the end – B is able to pursue education without utilizing his father's money.

This example can be applied to number of different scenarios. Loan against the existing amount within the system may serve to be a great way of retaining funds and utilizing the money.

# Appendix

## Definitions

*1. Decentralized Loan System*

This is a Blockchain based application of the loan system which tries to migrate traditional systems on to Blockchain.

*2. Existing System*

This typically defines the ‘non-profit’ organization for which this system was defined.

*3. Administrative Department / Admin*

They are responsible for allocating loan, managing funds and requesting funds.

*4. Loan Borrower / Needy*

They are the people who request for the loan from the system.

*5. Loan Lender / Elite*

They are the people who donate funds into the system.

*6. Blockchain v/s Other System*

The paper makes it very much clear as to why Blockchain based solution is used as compared to any other technology. Following are the most notable reasons –

a. Blockchain is immutable – hence cannot be altered or manipulated.

b. There is on demand transparency – which can be leveraged as and when required.

c. Stake holders can vote for or against approval of a loan.

d. Managing of funds is relatively easy.

e. There is a trust among the stake holders – which attracts more donations.

*7. Vote*

Stake holders may enjoy the power of voting. However, it has certain restrictions which are defined here.

a. The power of the vote is directly proportional to the amount of funds currently secured with the system by the stake holder.

b. The power of the vote is also dependent upon – credibility. If there is a bad loan or delayed repayments – the credibility of the stake holder is affected.

c. Once a stake holder cast his vote – he cannot re-vote until the said amount has been completely repaid. (Certain exceptions applied.)

Note: The feature of vote may be superseded by the chairman of the committee at any time.

*8. Partial Transparency*

Although the participants of this system enjoy transparency at all times – it should be noted that personal information which may reveal the identity of the participant will be not be shared with any member. However, a few members of the admin department may be allowed to view the identity of a person. However, it is mandatory on them to not disclose the information.

## Bibliography

a. https://medium.com/trivial-co/lending-and-borrowing-on-the-blockchain-should-banks-be-scared-e0a01c857c43

b. https://hackernoon.com/decentralized-credit-lending-on-the-blockchain-fa988b1ec7b4

c. https://lendoit.com

d. https://github.com/bhaumik-choksi/Loan

e. https://github.com/ripio/rcn-network

f. https://github.com/SatoshiNextTechLab/0xSHG

g. https://github.com/rajivjc/mortgage-blockchain-demo

h. https://github.com/anshulshah96/CrowdBank

i. https://github.com/piy0999/CreditSense