Indian Institute of Technology, Kharagpur

Department of Computer Science and Engineering

End-Semester Examination, Spring 2013-14

Software Engineering (CS 20006) - Solutions

Students: 130 **Date**: 22-Apr-14 (AN)

Full marks: 100 **Time**: 3 hours

Instructions:

1. Marks for every question is shown with the question.
2. No clarification to any of the questions will be provided. If you have any doubt, please make suitable assumptions and proceed. State your assumptions clearly.

---

1. A private company wants to manage the attendance and leave of its employees through a **Leave Management System (LMS)**. The requirement specifications for LMS and the associated information for the Company are:

   (a) The company has three categories of employees having designations as:

   - *Executive*: Employees who work as individual contributors and report to a Lead.
   - *Lead*: Every Executive reports to a Lead who approves her / his leave. A Lead reports to a Manager.
   - *Manager*: Every Lead reports to a Manager who approves her / his leave. A Manager reports to the Owner. The Manager of the Lead of an Executive is her / his *Reporting Manager*.

   The company has an *Owner*. Every Manager reports to the Owner who approves her / his leave.

   Every employee (and the Owner) is identified by a unique Employee Code and has name, designation, date-of-joining, date-of-birth, gender, address, and other personal details.

   (b) The company has provisions for the following categories of leave associated with the respective leave rules:

   - Each leave (any category) accounts for an absence of a full day from work. There is no half-day leave.
   - *Casual Leave (CL)*:
     - 10 CL's are available in a calendar year (01-Jan to 31-Dec). All CL's are credited to an employee on 01-Jan. For employees joining in the middle of the year, the number of CL's are prorated. CL's cannot be carried over to the next calendar year.
     - More than 2 CL's cannot be availed at a time. CL's cannot be clubbed with other types of leave. Holidays can be prefixed, suffixed or interspersed with CL's. Total period of absence including holidays cannot be more than 4 days. Holidays intervening the absence are not counted as leave.
     - CL's do not need pre-approval; but must be approved within 2 days of its availing.
   - *Earned Leave (EL)*:
     - 15 EL's are available in a calendar year. 1.25 EL's are credited on the completion of a full month's service. EL's can be carried over to the next calendar year and accumulated up to 45 days.
     - Once the EL balance crosses 45 days, then on the completion of the current quarter, 30 EL's are encashed and paid to the employee. Remaining EL's continue in the account. All EL's are encashed when an employee leaves the company.
     - EL's can be availed at a stretch and up to the existing balance. It can be clubbed with other leaves (except CL). All holidays within the leave of absence of EL are counted as EL.
     - In exceptional cases the reporting Manager of an employee or the Owner can approve more EL's than what exists in one's account. Maximum of 15 days' negative EL balance is allowed.
     - All EL's must be pre-approved (at least by a week).

- *Duty Leave (DL)*:
  - When an employee is sent out of station on work, a DL is created.
  - Every DL is on pre-approval basis and has no specific accounting. It is as if being *On Duty*.
  - Only a Lead or a Manager can avail of a DL.
  - Every DL is approved by the Owner.
- *Sick Leave (SL)*:
  - 12 SL's are available in a calendar year. All SL's are credited to an employee on 01-Jan. For employees joining in the middle of the year, the number of SL's are prorated. SL's can be carried over to the next calendar year and accumulated up to 60 days.
  - SL's can be availed at a stretch and up to the existing balance. It can be clubbed with other leaves (except CL). All holidays within the leave of absence of SL are counted as SL.
  - In exceptional cases the reporting Manager of an employee or the Owner can approve more SL's than what exists in one's account. Maximum of 12 days' negative SL balance is allowed.
  - Medical certificates are needed to proceed to and join back from SL's. SL's can be approved post-facto in cases of emergency.
- *Maternity Leave (ML)*:
  - Every female employee is eligible for 4 months' ML when pregnant. It can be clubbed with other leaves (except CL). All holidays within the leave of absence of ML are counted as ML.
  - Medical certificates are needed to proceed to and join back from ML's.
  - All ML's must be pre-approved (at least by a week).
  - A female employee can avail of ML only twice during her employment with the company.
- *Parental Leave (PL)*:
  - Every employee is eligible for 7 days' PL when she / he becomes a parent (biologically, or by adoption). It can be clubbed with other leaves (except CL). All holidays within the leave of absence of PL are counted as PL.
  - A female employee cannot enjoy a PL if she is availing of ML.
  - Parenthood certificates are needed for PL's.
  - All PL's may be pre-approved. However, given the uncertainty of date of birth, PL's can be approved post-facto within 2 days of proceeding on leave.
  - An employee can avail of PL only twice during his / her employment with the company.
- *Leave Without Pay (LWP)*:
  - A employee can avail of a maximum of 180 days' LWP at a stretch where he / she continues to be employed by the company; but is on leave and does not draw any salary.
  - LWP can be clubbed with other leaves (except CL). All holidays within the leave of absence of LWP are counted as LWP.
  - All LWP's must be pre-approved (at least by a week) and are of exceptional nature. Hence LWP's are always approved by the Owner.

Note:

- Every employee needs to record attendance to office on a daily basis unless it is a declared holiday of the company. If attendance is not recorded on a day, the employee must avail any one of the above kinds of leaves. Any other leave of absence is considered an *Unauthorized Leave (UL)* for which salary is deducted. More than a week's UL warrants disciplinary actions.
- Every leave (excluding the exceptions mentioned above) for an employee is approved by the reporting authority or the Owner.
- Leave is a privilege and not a right. Hence an approved leave can be revoked.
- No daily attendance record is maintained for the Owner. Hence the Owner's absence does not come under the purview of the LMS.

(c) The LMS supports the following core and subsidiary leave actions by the employees or the owner in accordance with the authorities listed in the tables below:

Note: An NA (*Not Applicable*) entry in the tables means that the corresponding action is not applicable for the concerned employee or the owner.

Core Leave Actions	**Employee Designation / Owner**			
	Executive	*Lead*	*Manager*	*Owner*
Request for Leave	For self	For self	For self	NA
Cancel an Approved Leave not yet availed	For self	For self	For self	NA
Approve / Regret Leave request	NA	For Executives reporting to self	For Leads reporting to self	For all
Revoke an Approved Leave	NA	Approved by self	Approved by self or by a Lead reporting to self	Any approved leave

Subsidiary Leave Actions	**Employee Designation / Owner**			
	Executive	*Lead*	*Manager*	*Owner*
Record Daily Attendance	For self	For self	For self	NA
Avail Leave (if approved)	For self	For self	For self	NA
Check / Export Leave Status	For self	For self & for Executives reporting to self	For self & for Leads reporting to self	For all
Credit, Debit and Adjust Leaves	NA	NA	For Leads reporting to self & for Executives reporting to them	For all

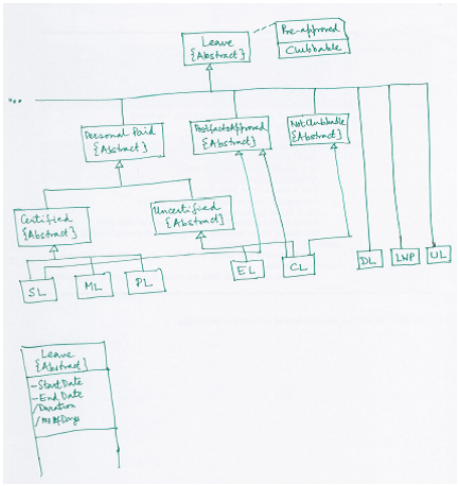
Note: Actions pertaining to sign in to the system, joining & termination of employees, administrative functions of a Manager or the Owner, and batch processing for leave reconciliation, leave credit, encashment etc are certainly required parts of the LMS. However, they have been omitted here for brevity.

You have been assigned as the software engineer for the LMS. You are required to analyse the specifications, design the system and also prepare the test plan. Answer the following questions in this background.

(a) Design Use-Case Diagrams for the core and subsidiary leave actions. Identify the actors and highlight various relationships between the use-cases. [**2+2=4**]

(b) Design Class Diagrams for Leaves detailing the attributes and operations with their properties. [**7**]

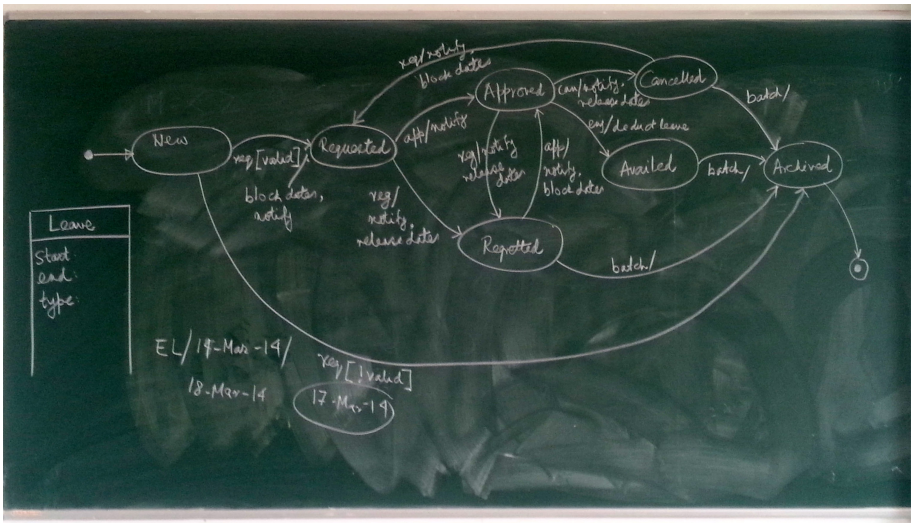
Answer:



(c) Show all other classes / objects (in addition to Question 1b) by brief Class Diagrams (with name and key attributes). For the entire collection of classes (that is, including Leaves) show the associations, aggregations/compositions, generalization/specialization, and abstract/concrete etc. [**2+3=5**]

(d) Design the State-Chart Diagram for Leaves. [**3**]

Answer:



(e) Design Sequence Diagrams for the core leave actions. [**1+1+2+1=5**]

(f) Identify situations for Iterator, Singleton and Command Design Patterns for the LMS design. [**3**]

(g) Outline your implementation plan with reference to the following (justify every choice): [**2+1=3**]

i. Choice of architecture and platform (including servers, if any)

ii. Choice of language and third-party libraries

(h) Prepare a test plan for the LMS to perform black-box tests. Clearly mark the scenarios for Unit Testing, Integration Testing, and System Testing. [**6+2+2=10**]

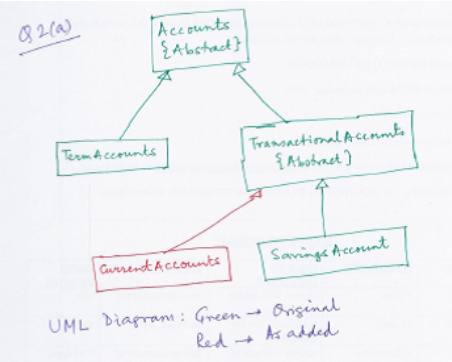
2. Consider the polymorphic hierarchy rooted at `class Accounts`:

```
struct ProhibitedCall { } // Thrown if a prohibited method is called for an object

class Accounts { protected:
    string accountNo_;        // Account number
    string accountHolder_;    // Account holder name
public:
    Accounts(string& no, string& holder): accountNo_(no), accountHolder_(holder) {}
    virtual ~Accounts() {}
    virtual int GetBalance() = 0;  // Compute Balance in the account
    virtual void Passbook() = 0 {      // Print brief account statement
        cout << accountNo_ << ":: " << accountHolder_ << ":: " << GetBalance() << " ";
    }
};
class TransactionalAccounts: public Accounts { protected:
    unsigned int TotalCredit_;   // Total credits to the account
    unsigned int TotalDebit_;    // Total debits from the account
public:
    TransactionalAccounts(string& no, string& holder):
        Accounts(no, holder), TotalCredit_(0), TotalDebit_(0) {}
    int GetBalance() { return TotalCredit_ - TotalDebit_; }
    void Passbook() = 0 { Accounts::Passbook();
        cout << endl << "Credits:: " << TotalCredit_ <<
            " Debits:: " << TotalDebit_ << " ";
    }
};
class SavingsAccount: public TransactionalAccounts {
    unsigned int interestRate_;    // Interest rate for the account
public:
    SavingsAccount(string& no, string& holder):
        TransactionalAccounts(no, holder), interestRate_(10) {}
    void Passbook() { TransactionalAccounts::Passbook();
        cout << endl << "Interest Rate:: " << interestRate_ << " ";
        cout << endl << "Thank you for transacting on your Saving Account" << endl;
    }
};
class TermAccounts: public Accounts {
    unsigned int interestRate_;      // Interest rate for the account
protected:
    unsigned int oneTimeDeposit_;    // Deposit at the time of opening account
    unsigned int currentDeposit_;    // Current amount of deposit in the account
    unsigned int periodMonths_;      // Life period of the account in months
public:
    TermAccounts(string& no, string& holder, unsigned int deposit, unsigned int period):
        Accounts(no, holder), interestRate_(30), oneTimeDeposit_(deposit),
        currentDeposit_(oneTimeDeposit_), periodMonths_(period) {}
    int GetBalance() { return currentDeposit_; }
    void Passbook() { Accounts::Passbook();
        cout << endl << "Deposit:: " << oneTimeDeposit_ <<
            " Period:: " <<  periodMonths_ << " ";
        cout << endl << "Thank you for transacting on your Term Account" << endl;
    }
};
```

(a) Draw the UML class diagram for the hierarchy. [**5**]

Answer:



(b) Identify the abstract base classes in the hierarchy. [**2**]

Answer:

```
class Accounts; // Abstract because ...
    // virtual int GetBalance() = 0;
    // virtual void Passbook() = 0 { ... }

class TransactionalAccounts; // Abstract because ...
    // void Passbook() = 0 { ... } // By inheritance and definition
```

(c) Given that `CurrentAccount` ISA `TransactionalAccounts` extend the hierarchy by adding a concrete `CurrentAccount` class to it. [*Just write the codes that need to be added*]. [**3**]

Answer:

```
class CurrentAccount: public TransactionalAccounts {
public:
    CurrentAccount(string& no, string& holder): TransactionalAccounts(no, holder) {}
    void Passbook() {
        TransactionalAccounts::Passbook();
        // Message optional
        cout << endl << "Thank you for transacting on your Current Account" << endl;
    }
};
```

Note:

- Award **1** mark for `class CurrentAccount: public TransactionalAccounts`.
- `TransactionalAccounts` does not have a default constructor. Hence it must be explicitly constructed. Award **1** mark.
- Since `TransactionalAccounts` is abstract for the virtual function `void Passbook()` (though `TransactionalAccounts` has an implementation for it), it is mandatory to define it for `CurrentAccount`. Award **1** mark.

- The `void Passbook()` method can be implemented in other ways too, like:

```
void Passbook() { } // OR

void Passbook() {
    cout << endl << "Thank you for transacting on your Current Account" << endl;
} // OR

void Passbook() { TransactionalAccounts::Passbook(); } // OR any other
```

  The same will be accepted.

(d) Add an appropriate virtual method to get the `interestRate_` and implement it for every concrete specialization of `Accounts`. Assume that `class CurrentAccount` does not have the `interestRate_` data member. [**3**]

Answer:

```
class Accounts {
    // ...
public:
    // ...
    virtual int GetInterestRate() = 0;
    // ...
};
class TransactionalAccounts: public Accounts {
    // ...
public:
    // ...
};
class SavingsAccount: public TransactionalAccounts {
    unsigned int interestRate_;    // Interest rate for the account
public:
    // ...
    int GetInterestRate() { return interestRate_; }
    // ...
};
class CurrentAccount: public TransactionalAccounts {
    // ...
public:
    // ...
    int GetInterestRate() { throw ProhibitedCall(this); }
    // ...
};
class TermAccounts: public Accounts {
    unsigned int interestRate_;      // Interest rate for the account
public:
    // ...
    int GetInterestRate() { return interestRate_; }
    // ...
};
```

Note:

- **1** mark for the method in `class CurrentAccount` and **0.5** each for the remaining 4 classes.
- Showing `int GetInterestRate() = 0;` in `class TransactionalAccounts` is optional.
- `int GetInterestRate() { }` is not acceptable in `class CurrentAccount`. However an error message like `int GetInterestRate() { cout << "Bad Call"; return -1; }` is.
- Outline implementations would be accepted.

(e) Discuss the implications of changing the signature of the destructor of the root class from `virtual ~Accounts()` to `~Accounts()`. [**2**]

Answer: If the destructor `~Accounts()` is not virtual then slicing will happen when an object of some specialized class like `SavingsAccount`, `CurrentAccount` or `TermAccounts` will be (polymorphically) destructed through a pointer to `Accounts`.

3. Consider the polymorphic hierarchy rooted at `class Accounts` (Question 2). Let the collection of different accounts be maintained as a `Portfolio` object:

```
class Portfolio {
    vector<Accounts *> account_;    // Accounts container
public:
    Portfolio() {}
    ~Portfolio() {}
    void AddAccount(Accounts *t) { account_.push_back(t); }
    vector<Accounts *>& GetAccounts() { return account_; }
    unsigned int GetNumberOfAccounts() { return account_.size(); }
};
```

(a) It is required to enforce that there can be *only one* `Portfolio` in the application. Using Singleton Design Pattern, implement the requirements (in `C++`) in the above design. [**5**]

Answer:

In the header file change `class Portfolio` design to: [**3**]

```
// Portfolio of Accounts as a Singleton
class Portfolio {
private:
    vector<Accounts *> account_; // Accounts container
    static Portfolio* instance_; // Singleton Instance

protected:
    Portfolio() {}  // Constructor

public:
    ~Portfolio() {} // Destructor

    static Portfolio* Instance(); // Acquire the Singleton Instance

    // Container Methods - NO CHANGE ...
};
```

Then in the class implementation file introduce: [**1**]

```
Portfolio* Portfolio::instance_ = 0; // Singleton Instance definition
```

Finally in the application file change the use-model to: [**1**]

```
Portfolio& myPortfolio = *Portfolio::Instance(); // Acquire the Singleton Instance
```

(b) Using the Iterator Design Pattern in `vector` (of STL), write a `C++` code snippet that calls the `Passbook()` method for all the different accounts contained in the singleton `Portfolio` object. [**5**]

Answer:

```
Portfolio& myPortfolio = *Portfolio::Instance(); // Acquire Instance

// Iterate on the portfolio
for(vector<Accounts *>::const_iterator pos = myPortfolio.GetAccounts().begin();
    pos != myPortfolio.GetAccounts().end();
    ++pos) {
        (*pos)->Passbook();
}
```

Note:

- `Portfolio` may be used through a pointer. But it is necessary to show its acquisition. Award **1** mark.
- `iterator` may be used in place of `const_iterator`. Similarly `pos++` may be used in place of `++pos`.
- Award **1** mark each for `begin()`, `end()`, `++pos`, and `(*pos)->Passbook()`. Deduct half marks for any kind of syntax error.

4. Consider the polymorphic hierarchy rooted at `class Accounts` (Question 2). It is required to add a polymorphic method `void CreditInterest()` that would update the interest earning of an account given a base type pointer (`Accounts*`) or reference (`Accounts&`) to its instance. The behavior of `void CreditInterest()` for the classes above is given in the code snippets below:

```
void SavingsAccount::CreditInterest() {
    TotalCredit_ += TotalCredit_*interestRate_/100;
}
void CurrentAccount::CreditInterest() {
    throw ProhibitedCall();
}
void TermAccounts::CreditInterest() {
    currentDeposit_ += oneTimeDeposit_*interestRate_/100;
}
```

This addition of method is simple if we are allowed to re-compile the hierarchy. We add `virtual void CreditInterest() = 0;` to `Accounts` and `void CreditInterest();` to all classes that are concrete. However, if we are not allowed to re-compile the hierarchy this cannot be done. So we need to change the design of the hierarchy in a way so that the recompilation of the hierarchy is not needed; yet methods can be added. Using Visitor Design Pattern propose a solution to this situation.

(a) Write all the codes necessary to implement the changed design of the hierarchy. [*You need not copy the whole code - just write the additions, deletions, and changes*]. [**5**]

Answer:

First a `class AccountsVisitor` is designed to visit the hierarchy:

```
class Accounts; // Forward declaration

// Abstract Base Class for Visitor
class AccountsVisitor { public:
    // Visit methods for Abstract classes
    virtual void Visit(Accounts*) = 0;
    virtual void Visit(TransactionalAccounts*) = 0;

    // Visit methods for Concrete classes
    virtual void Visit(SavingsAccount*) = 0;
    virtual void Visit(CurrentAccount*) = 0;
    virtual void Visit(TermAccounts*) = 0;
};

// Implementation of void Accept() method for dispatch from every concrete class
inline void SavingsAccount::Accept(AccountsVisitor& visitor) { visitor.Visit(this); }
inline void CurrentAccount::Accept(AccountsVisitor& visitor) { visitor.Visit(this); }
inline void TermAccounts::Accept(AccountsVisitor& visitor) { visitor.Visit(this); }
```

The change in the design of `class Accounts` hierarchy are as follows (`void Accept(class AccountsVisitor& v)` added):

```
class Accounts {
    // ...
public:
    // ...
    // Accept method for supporting visitor
    virtual void Accept(class AccountsVisitor& v) = 0;
};

class TransactionalAccounts: public Accounts {
    // ...
public:
    // ...
    // Exposed for supporting visitor
    unsigned int GetTotalCredit() { return TotalCredit_; }
    void SetTotalCredit(unsigned int tCredit) { TotalCredit_ = tCredit; }
    unsigned int GetTotalDebit() { return TotalDebit_; }
    void SetTotalDebit(unsigned int tDebit) { TotalDebit_ = tDebit; }
};

class SavingsAccount: public TransactionalAccounts {
    // ...
public:
    // ...
    // Exposed for supporting visitor
    unsigned int GetInterestRate() { return interestRate_; }

    // Accept method for supporting visitor
    void Accept(class AccountsVisitor& v);
};
```

```
class CurrentAccount: public TransactionalAccounts {
public:
    // ...
    // Accept method for supporting visitor
    void Accept(class AccountsVisitor& v);
};

class TermAccounts: public Accounts {
    // ...
public:
    // ...
    // Exposed for supporting visitor
    unsigned int GetInterestRate() { return interestRate_; }
    unsigned int GetOneTimeDeposit() { return oneTimeDeposit_; }
    unsigned int GetCurrentDeposit(){ return currentDeposit_; }
    void SetCurrentDeposit(unsigned int deposit){ currentDeposit_ = deposit; }
    unsigned int GetPeriod() { return periodMonths_; }

    // Accept method for supporting visitor
    void Accept(class AccountsVisitor& v);
};
```

(b) Show how the method `void CreditInterest()` can be added easily. [**3**]

Answer:

We need to create `class CreditInterestVisitor` as a specialization of `class AccountsVisitor` to implement `void CreditInterest();`. Implementations are taken from the given codes.

```
// Visitor for void CreditInterest() method
class CreditInterestVisitor: public AccountsVisitor {
public:
    CreditInterestVisitor() {}

    // Note that these classes are abstract
    // Their Visit()'s are called from concrete classes below
    void Visit(Accounts *acct) { throw ProhibitedCall(acct); }
    void Visit(TransactionalAccounts*acct) { throw ProhibitedCall(acct); }

    // Concrete classes polymorphically dispatch the call on the hierarchy
    // when concrete classes inherit these methods
    void Visit(SavingsAccount *acct) {
        unsigned int tCredit = acct->GetTotalCredit();
        acct->SetTotalCredit(tCredit + tCredit*acct->GetInterestRate()/100);
    }
    void Visit(CurrentAccount *acct) { throw ProhibitedCall(acct); }
    void Visit(TermAccounts *acct) {
        unsigned int deposit = acct->GetOneTimeDeposit();
        acct->SetCurrentDeposit(deposit + deposit*acct->GetInterestRate()/100);
    }
};
```

Given the `CreditInterestVisitor`, the method can be invoked as follows:

```
CreditInterestVisitor civ; // Visitors for dispatch

// Iterate on the portfolio - using methods dispatched through Visitor
for(vector<Accounts *>::const_iterator
    pos = myPortfolio.GetAccounts().begin();
    pos != myPortfolio.GetAccounts().end(); ++pos) {

    try {
        // Visitor-based dispatch - modified void CreditInterest()
        (*pos)->Accept(civ);
    }
    catch (ProhibitedCall& p) {
        cout << "Illegal Function Call for " << p.account_ << endl << endl;
    }
}
```

(c) Explain why you need to break the encapsulation (provide `Get()` / `Set()` methods in public for some of the private or protected members to increase visibility) of some or all of the classes. [**1**]

Answer:

For example to compute `void SavingsAccount::CreditInterest()` as in `void Visit(SavingsAccount *acct)` we need to read (`Get()`) `interestRate_` and read / write (`Get()` / `Set()`) `TotalCredit_` from outside `SavingsAccount` class. Hence the encapsulations need to be compromised by introducing `SavingsAccount::GetInterestRate()` and `TransactionalAccounts::GetTotalCredit()` & `TransactionalAccounts::SetTotalCredit()` methods.

(d) Explain the auxiliary methods needed to make the Visitor Pattern work. [**1**]

Answer:

We need to introduce `virtual void Accept(class AccountsVisitor& v) = 0;` at the root `class Accounts` and `void Accept(class AccountsVisitor& v);` in all concrete classes. Further we need to define `virtual void Visit(X*) = 0;` methods in visitor base class `AccountsVisitor` for all `X` classes of the hierarchy. These methods are then implemented in the concrete visitor class `CreditInterestVisitor` that implements the target visitor method.

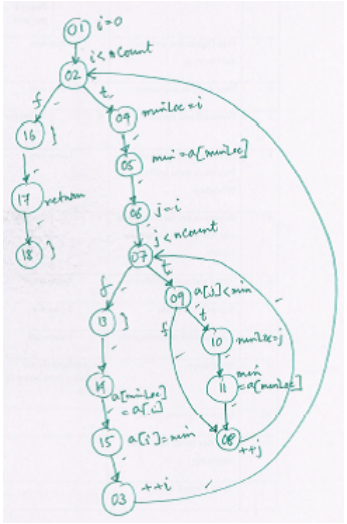
5. You have developed the `SelectionSort` function to sort an array `a` of `nCount` elements in ascending order. Now you need to prepare various white-box tests for the code. For this you would use program analysis techniques and prepare minimal set of test cases for every scenario.

```
    void SelectionSort(int a[], unsigned int nCount) { // Sorts by Selection
        unsigned int i, j; // Indices to run over the array

01:     for(i = 0;
02:         i < nCount;
03:         ++i) {
04:         unsigned int minLoc = i; // Where the min occurs
05:         int min = a[minLoc];     // Start a[i] as min
06:         for(j = i;
07:             j < nCount;
08:             ++j) { // Find the min in the rest of the array
09:             if (a[j] < min) {
10:                 minLoc = j;
11:                 min = a[minLoc];
12:             }
13:         } // Inner for loop
14:         a[minLoc] = a[i]; // Swap min with a[i]
15:         a[i] = min;
16:     } // Outer for loop
17:     return;
18: }
```

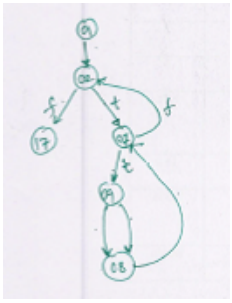
(a) Construct the CFG (Control Flow Graph) for `SelectionSort` using the line numbers as shown. [**5**]

Answer:



(b) Compute the cyclomatic complexity of `SelectionSort` using the CFG in 5a. [**2**]

Answer:



Number of nodes $N = 18$
Number of Edges $E = 10$
Cyclomatic Complexity $V(G) = E - N + 2 = 3$

(c) Compute the Linearly Independent Paths (LIP) in 5a. [**3**]

(d) For every LIP in 5c, design a test case each that forces the control to trace the LIP. [**3**]

(e) Does tests in 5d guarantee 100% line coverage? Justify or identify an uncovered line. [**2**]

(f) Show by a counter example that 100% block coverage in 5a may not guarantee 100% branch coverage. [**2**]

(g) Compute the `DEF` & `USES` sets and `DU`-chains for `minLoc` and `min`. Design test cases to cover these `DU`-chains. [**2+2=4**]

(h) For each of the following mutations (applied separately), check whether the test suite in 5d can kill the mutant. If not, design the killer test. [**2+2=4**]

- Mutant 1:

```
06:          for(j = i;
```

  changed to

```
06:          for(j = 0;
```

- Mutant 2:

```
09:              if (a[j] < min) {
```

  changed to

```
09:              if (a[j] > min) {
```