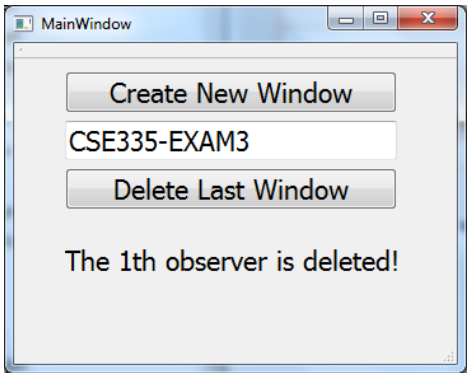
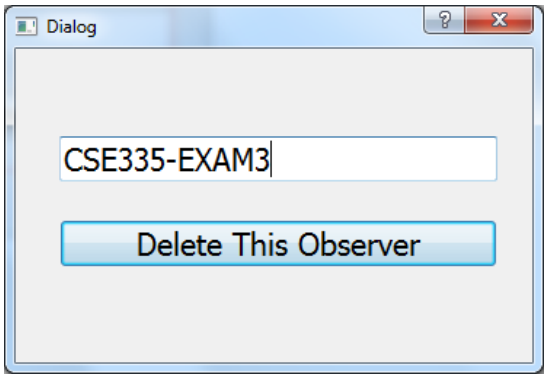
CSE 335 Spring 2016 Exam 3

First Name:_____ Last Name:_____Netid:_____@msu.edu

Question 1.1 [40 points] Observer Pattern

The images below show screen shots of a simple application. The application starts with the main window that contains two push button objects (QPushButton), a text input object (QLineEdit), and a text label (QLabel). When "Create New Window" button is clicked, a dialog window opens as shown below. The dialog window contains a push button object (QPushButton) and a text input object (QLineEdit). When the user changes text in the text input object of dialog window it is immediately reflected in the main window's text input object and vice versa. Clicking the push button in dialog window closes the dialog and clicking the "Delete Last Window" button in main window closes the most recently created dialog. The use can create multiple dialog windows at the same time. When a dialog is deleted, the text label in main window is updated showing which dialog was deleted. Complete the code given below to implement the functionality described above. For your convenience parts of the code are already given to you. Here is a list of QT signals for your reference: *textChanged(QString), valueChanged(QString), textEdited(QString), returnPressed(), editingFinished(), clicked(), pressed(), released(), stateChanged(int)*





Question 1.1.1 [5 points]

Complete the MainWindow.h header by adding slots, signals and/or other necessary data members.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include "observerdialog.h"
namespace Ui {class MainWindow;}
class MainWindow : public QMainWindow{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
```

public slots:
private slots:

```
private:
    Ui::MainWindow *ui;
```


```
};
#endif // MAINWINDOW_H
```

Question 1.1.2 [5 points]
Complete the ObserverDialog.h header by adding slots, signals and/or other necessary data members.

```
#ifndef OBSERVERDIALOG_H
#define OBSERVERDIALOG_H
#include <QDialog>
namespace Ui {class ObserverDialog;}
class ObserverDialog : public QDialog{
    Q_OBJECT
public:
    explicit ObserverDialog(QWidget *parent = 0);
    ~ObserverDialog();
    void setObserverID(int);
```

signals:
public slots:
private slots:

```
private:
    Ui::ObserverDialog *ui;
    unsigned int observerID;
};
#endif // OBSERVERDIALOG_H
```

Question 1.1.3 [10 points]

Complete the ObserverDialog.cpp by writing syntactically correct code including **connect** statement(s) at appropriate places.

```
#include "observerdialog.h"
#include "ui_observerdialog.h"
ObserverDialog::ObserverDialog(QWidget *parent) :
QDialog(parent),ui(new Ui::ObserverDialog){
    ui->setupUi(this);
```

```

```

```
ObserverDialog::~ObserverDialog(){delete ui;}
void ObserverDialog::setObserverID(int id){
    observerID = id;}
```

```

```

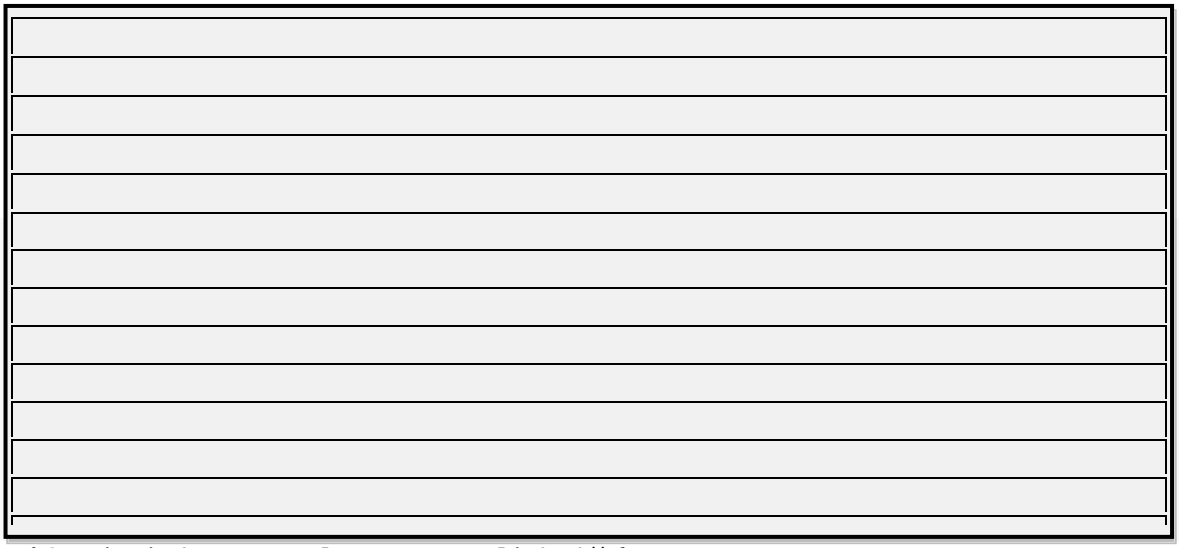
Question 1.1.4 [10 points]

Complete the MainWindow.cpp by writing down syntactically correct code including **connect** statement(s) at appropriate places.

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
MainWindow::MainWindow(QWidget *parent):
QMainWindow(parent),ui(new Ui::MainWindow){
    ui->setupUi(this);
```

```

```

```
MainWindow::~MainWindow(){
    delete ui;
}
void MainWindow::on_CreateButton_clicked(){
```



```
void MainWindow::on_DeleteButton_clicked(){
```

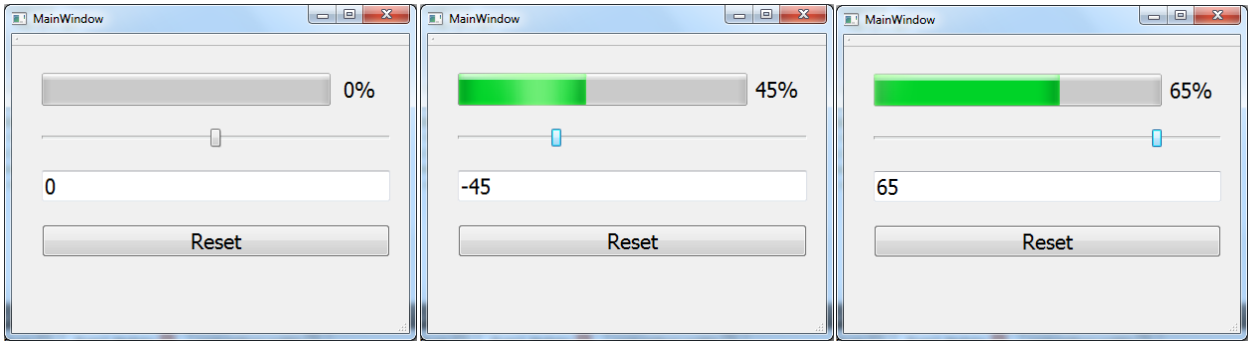
```
void MainWindow::deleteObserver(unsigned int id){
    observerID--;
    ui->label->setText("The "+QString::number(id+1)+"th observer is deleted!");
}
```

Question 1.2 [10 points] Sequence Diagram Draw a sequence diagram that depicts the following scenario: One dialog window is open and a user changes text in the text input of main.

.

Question 2. [30 points] Mediator Pattern

The images below shows screen shots of a simple application. The application starts with the main window that contains the following objects: progress bar, horizontal slider, text input, and push button. The progress bar ranges from 0 to 100. The slider range is from -100 to 100 and its default location is in the middle i.e., zero. The text input shows the slider position, it can also be used to change the slider position. Reset button resets slider position to zero. Complete the code given below to implement the functionality described above. For your convenience parts of the code are already given to you.



MainWindow.h

```
1.  #include <QMainWindow>
2.  namespace Ui {class MainWindow;}
3.  class MainWindow : public QMainWindow{
4.      Q_OBJECT
5.  public:
6.      explicit MainWindow(QWidget *parent = 0);
7.      ~MainWindow();
8.  public slots:
9.      void acceptChange(QObject*);
10. private:
11.     Ui::MainWindow *ui;
12. };
```

Question 2.1 [5 points]

Complete the MyLineEdit.h header by adding slots, signals and/or other necessary data members. Implement slot functions in the header file.

include <QLineEdit>
class MyLineEdit:
Q_OBJECT
public:
MyLineEdit(const QString& qs):QLineEdit(qs){};
MyLineEdit(QWidget* qw):QLineEdit(qw){};
signals:
public slots:
};

Question 2.2 [5 points]

Complete the MySlider.h header by adding slots, signals and/or other necessary data members. Implement slot functions in the header file.

```
#include <QSlider>
class MySlider:
    Q_OBJECT
public:
    MySlider(QWidget* qw):QSlider(qw){}

signals:


public slots:


};
```

Question 2.3 [5 points]

Complete the ResetButton.h header by adding slots, signals and/or other necessary data members. Implement slot functions in the header file.

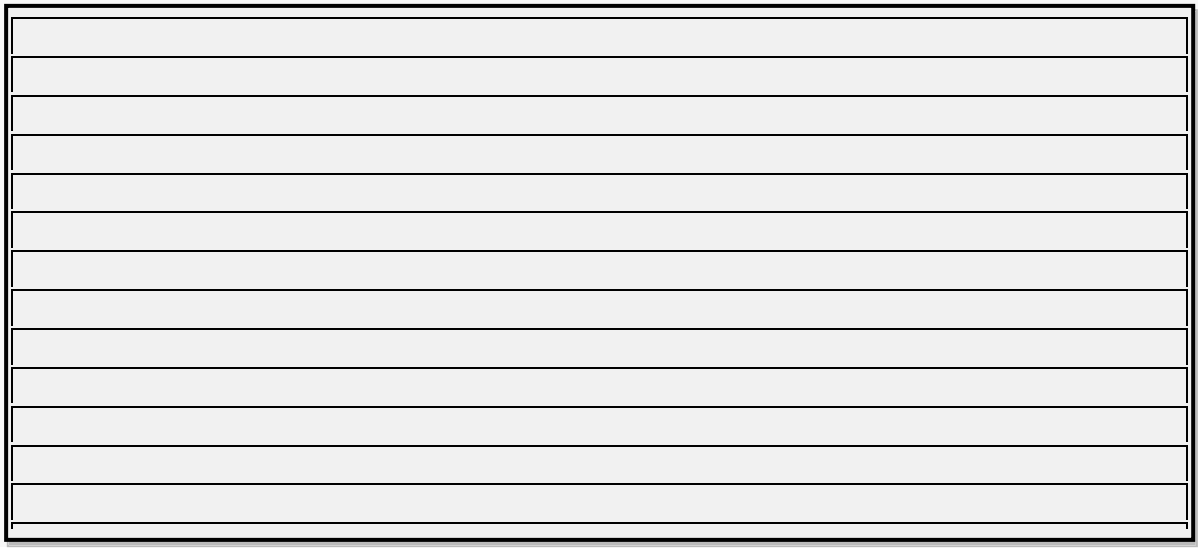
```
#include<QPushButton>
class ResetButton:
    Q_OBJECT
public:
    ResetButton(QWidget* qw):QPushButton(qw){}

signals:


private slots:


};
```

Question 2.4 [10 points]

Complete the MainWindow.cpp by writing down syntactically correct code including **connect** statements at appropriate places.

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
MainWindow::MainWindow(QWidget *parent) :
QMainWindow(parent),ui(new Ui::MainWindow){
    ui->setupUi(this);
```

```
    ui->horizontalSlider->setMinimum(-100);
    ui->horizontalSlider->setMaximum(100);
    ui->horizontalSlider->setValue(0);
    ui->progressBar->setMinimum(0);
    ui->progressBar->setMaximum(100);
    ui->progressBar->setValue(0);
    ui->lineEdit->setText("0");
```



```
}
MainWindow::~MainWindow(){delete ui;}
void MainWindow::acceptChange(QObject* sender){
    if(sender==ui->horizontalSlider){
        int val=ui->horizontalSlider->value();
        ui->lineEdit->setText(QString::number(val));
        if (val<0)val=-1*val;
        ui->progressBar->setValue(val);
    }
    else if(sender==ui->lineEdit){
        int val=ui->lineEdit->text().toInt();
        ui->horizontalSlider->setValue(val);
    }
    else if(sender==ui->pushButton){
        ui->horizontalSlider->setValue(0);
        ui->lineEdit->setText("0");
    }
}
```

Question 3 [30 points] Builder Pattern

The following code takes as input a logical expression as a string, parses the logical expression and prints out elements of the expression. A logical expression is similar to an arithmetic expression; the binary operators are replaced by logical operators. In this program we consider four local operators: AND (&), OR (|), NOR(~), and XOR (^). Consider the following logical expression: `(((1&0)~0)|(1^0))` and complete the following code to implement the parsing and printing functionality for the logical expression. For your convenience parts of the code are already given to you. Your solution must show appropriate headers included, inheritance relationships, member variables, and member functions.

Node.h

```
#include <string>
#include <iostream>
class Node{
protected:
    string m_value;
    Node* m_leftChild;
    Node* m_rightChild;
public:
    Node(string value):m_value(value), m_leftChild(NULL), m_rightChild(NULL){}
    virtual ~Node(){
        if(!m_leftChild) delete m_leftChild;
        if(!m_rightChild) delete m_rightChild;
    }
    string getValue(){ return m_value;}
    void setLeftChild(Node* left){ m_leftChild = left;}
    void setRightChild(Node* left){ m_rightChild = left;}
    Node* getLeftChild() const { return m_leftChild;}
    Node* getRightChild() const { return m_rightChild;}
    void print(){
        if(m_leftChild){
            cout<<"(";
            m_leftChild->print();
        }
        cout<<m_value;
        if(m_rightChild){
            m_rightChild->print();
            cout<<")";
        }
    }
};
```

Question 3.1 [10 points]
Write down syntactically correct C++ code for the Builder.h header file. Required header files are already included. If necessary show appropriate inheritance relationship, member variables, and member functions.

```
#include "Node.h"
#include <string>

class Builder{
public:
```

Question 3.2 [10 points]
Complete the ExpBuilder.h class by showing appropriate member variables. Also provide the implementation for the function `addRightParenthesis()`.

```
#include <stack>
#include <iostream>
#include "Node.h"
#include "Builder.h"
class ExpBuilder: public Builder{
```

```
public:
    virtual void addOperand(string operand){
        Node* newOperand = new Node(operand);
        expStack.push(newOperand);
    }
    virtual void addLiteral(string literal){
        Node* newOperand = new Node(literal);
        expStack.push(newOperand);
    }
    virtual void addLeftParenthesis(){}
```

```
virtual void addRightParenthesis(){
```


```
}
    virtual Node* getExpression(){
        Node* root = expStack.top();
        expStack.pop();
        return root;
    }
};
```

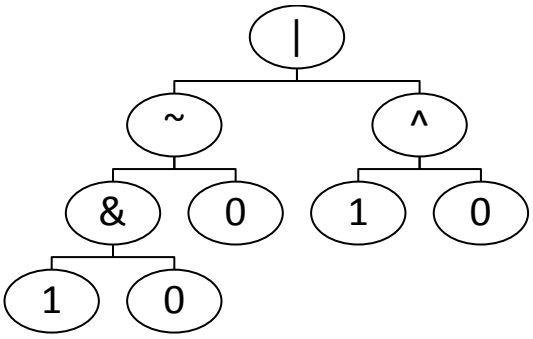
Question 3.3 [5 points]
Complete the ExpParser.h class by showing appropriate member variables and member functions.

#include <string>
#include <iostream>
#include "Builder.h"
class ExpParser{
} ;

```
    void parse(string exp){
        string literal;
        for(int i=0; i<exp.size(); i++){
            switch(exp[i]){
                case '(':
                    if(literal.size() > 0){
                        m_expBuilder->addLiteral(literal);
```

```
                        literal.clear();
                    }
                    m_expBuilder->addLeftParenthesis();
                    break;
                case ')':
                    if(literal.size() > 0){
                        m_expBuilder->addLiteral(literal);
                        literal.clear();
                    }
                    m_expBuilder->addRightParenthesis();
                    break;
                case '&':
                    if(literal.size() > 0){
                        m_expBuilder->addLiteral(literal);
                        literal.clear();
                    }
                    m_expBuilder->addOperand("&");
                    break;
                case '|':
                    if(literal.size() > 0){
                        m_expBuilder->addLiteral(literal);
                        literal.clear();
                    }
                    m_expBuilder->addOperand("|");
                    break;
                case '~':
                    if(literal.size() > 0){
                        m_expBuilder->addLiteral(literal);
                        cout<<literal<<endl;
                        literal.clear();
                    }
                    m_expBuilder->addOperand("~");
                    break;
                case '^':
                    if(literal.size() > 0){
                        m_expBuilder->addLiteral(literal);
                        literal.clear();
                    }
                    m_expBuilder->addOperand("^");
                    break;
                default:
                    literal.push_back(exp[i]);
                    break;
            }
        }
    }
};
```

Question 3.4 [5 points]
Complete main.cpp so that it prints out the nodes of the logical expression in an order shown in the tree.




```
using namespace std;
#include <cstdlib>
#include <string>
#include <iostream>
#include "Node.h"
#include "Builder.h"
#include "ExpBuilder.h"
#include "ExpParser.h"
int main(int argc, char** argv) {
```

root->print();
return 0;
}

Question 4 [12 points]
How do the following six design patterns allow us to avoid changing classes?

1- Composite Pattern:
2- Visitor Pattern:
3- Abstract Factory Pattern:
4- Template Method Pattern:
5- Adaptor Pattern:
6- Observer Pattern: