

# Software Design

## Session 2

Prof. Barry Dwolatzky

Director: Joburg Centre for Software Engineering (JCSE)

Wits University



26 October 2011

JCSE at Wits University



Page - 1

# Fundamental Principles

- Abstraction
- Modularity
- Cohesion
- Coupling
- Design Options
- Maintainability
- Reuse



# Achieving the fundamental principles

- Object Oriented Design
  - Objects
  - Classes
  - Inheritance
  - Late-binding
  - Polymorphism



2011/10/26

JCSE at Wits University



Page - 3

# Functional Decomposition



2011/10/26

JCSE at Wits University



Page - 4

```
enum functionType {linear, quadratic, invalid};
```

```
int main()
```

```
{
```

```
    functionType func;
```

```
    double vals[10];
```

```
    func = getFunctionType() ;
```

```
    getValues(func, vals);
```

```
    displayRoots(func, vals);
```

```
    cout << "That is it!" << endl;
```

```
    return 0;
```

```
}
```



```
functionType getFunctionType()
{
    char s;
    cout << "Linear or quadratic? [l or q]: ";
    cin >> s;
    switch (s)
    {
        case 'l': return linear;
            break;
        case 'q': return quadratic;
            break;
        default: return invalid;
    }
    return invalid;
}
```



```
void getValues(functionType f, double a[ ])
{
    switch (f)
    {
        case linear:
            cout << "Linear coefficients: m = ";
            cin >> a[0];
            cout << "c = ";
            cin >> a[1];
            break;
        case quadratic:
            cout << "Quadratic coefficients: a = ";
            cin >> a[0];
            cout << "b = ";
            cin >> a[1];
            cout << "c = ";
            cin >> a[2];
            break;
        case invalid:
            break;
    }
    return;
}
```



```
void displayRoots(functionType f, double coefs[ ])
{
    double m,c, root;
    double a,b,
    discr,root1,root2;
    switch (f)
    {
        case linear:
            m = coefs[0];
            c = coefs[1];
            root = -c/m;
            cout << "Linear root = " << root << endl;
            break;
```



```

case quadratic:
    a = coefs[0];
    b = coefs[1];
    c = coefs[2];
    discr = a*a - 4*a*c;
    if (discr < 0.0)
    {
        cout << "Quadratic roots imaginary" << endl;
        break;
    }
    root1 = (-b + sqrtf(discr))/(2*a);
    root2 = (-b - sqrtf(discr))/(2*a);
    cout << "Quadratic roots: " << root1 << " and "
        << root2 << endl;
    break;
case invalid:
    break;
}
return;
}

```



# OO Decomposition



2011/10/26

JCSE at Wits University



Page - 10

```
int main()
{
    function *f1;
    controller c;
    f1 = c.getFunctionType();
    if (f1 == NULL)
    {
        cout << "Illegal choice!" << endl;
        return 1;
    }
    f1->getValues( );
    f1->findRoots( );
    f1->displayRoots();
    delete f1;
    cout << "This is the end" << endl;
    return 0;
}
```



```
class function
{
    public:
        function( );
        virtual void getValues( );
        virtual void findRoots( );
        virtual void displayRoots( );
};
```

```
class controller
{
    public:
        function* getFunctionType( );
};
```



```
class quadratic: public function
{
    public:
        quadratic( );
        virtual void getValues ( );
        virtual void findRoots( );
        virtual void displayRoots( );
    private:
        double a,b,c, x1, x2;
        bool imag;
};
```

```
class linear: public function
{
    public:
        linear( );
        virtual void getValues ( );
        virtual void findRoots( );
        virtual void displayRoots( );
    private:
        double m, c, x;
};
```

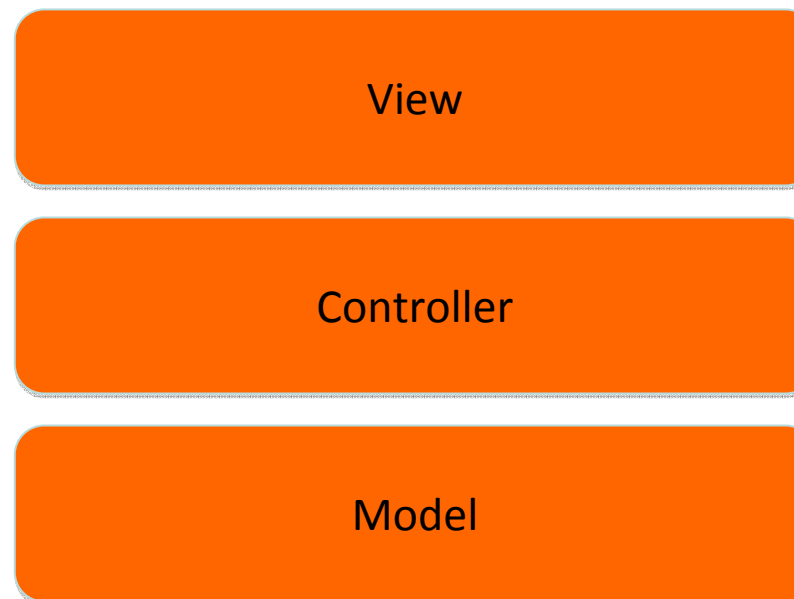


```
function* controller::getFunctionType()
{
    function* f;
    char s;
    cout << "Linear or quadratic? [l or q]: ";
    cin >> s;
    switch (s)
    {
        case 'l': f = new linear;
                break;
        case 'q': f = new quadratic;
                break;
        default: return NULL;
    }
    return f;
}
```



# Layered Architecture

- Model – View – Controller (MVC) is an important architectural pattern for mobile apps



# MVC

- The **model** manages the behaviour and data of the application domain
- The **view** renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model
- The **controller** accepts inputs and instructs the model and view to perform actions based on that input



# MVC

- The control flow in the MVC is generally as follows:
  - User interacts with the UI in some way (eg. pressing a button)
  - The controller handles an input event from the UI and converts it into an appropriate action understandable by the model
  - The controller notifies the model of the user action, possibly resulting in a change in the model's state (eg. Adds something to user's shopping cart)
  - A view queries the model in order to generate an appropriate change in the UI. In some implementations the controller may issue an update instruction to the view. In others the view is automatically notified by the model of changes in state (observer)
  - The UI waits for further user interactions.

