Incremental Testing of Object-Oriented Class Structures

M. Harrold, J. McGregor,
K.Fitzpatrick
1992

Presented by Dave Tahmoush

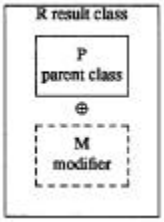
Object Oriented Programming

- Benefit from reuse of information-hiding modules
  - Called classes
  - A class has attributes
    - Data members or instance variables
    - Member functions or methods
  - Classes can be used to define new classes, or subclasses
    - Inheritance allows subclasses to use attributes from parent class
      - May also cancel attributes
      - Redefine attributes
      - Create new attributes
  - Would like to create libraries of tested classes to reuse
    - Completely retesting too expensive

Heirarchical Incremental Class Testing

- Reuse testing information from parent class
- Create testing history
  - Test suites for each attribute
- Incrementally update to guide testing of subclass
  - Inherit testing history, and update it
  - Automatically classify attributes
    - Test or not, or only partially test?
    - Can we reuse test cases?
- Inheritance is guide to testing

Inheritance in Object-Oriented Systems

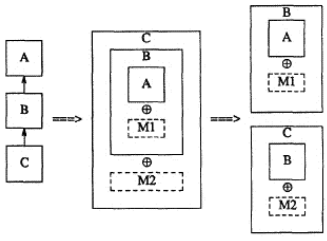


- Modifier M changes the attributes of parent class P to create new class R
- Incremental modification technique
- M contains attributes that alter class R
- Types of attributes in R
  - New, defined in M
  - Recursive, defined in P but available in R
  - Redefined, defined in P and changed in M
  - Virtual of all above types, specified but incomplete
- Examples on next slide

```
class P {                    class R : public P {          R's attributes after the mapping
  private:                     private:                      private:
    int i;                       float i;                      float i;                    //new
    int j;
  public:                      public:                       public:
    P(){}                        R(){}
    void A(int a,int b)                                        void A(int a, int b)
      {i=a; j=a+2*b;}                                            {i=a; j=a+2*b;}          //recursive
                                 void A(int a)                 void A(int a)
                                   {P::A(a,0);}                  {P::A(a,0);}             //new
    virtual int B()              virtual int B()               virtual int B()
      {return i;}                  {return 3*P::B();}            {return 3*P::B();}       //virtual-redefined
    int C()                      int C()                       int C()
      {return j;}                  {return 2*P::C();}            {return 2*P::C();}       //redefined
};                           };
                                                             hidden
                                                               int i;
                                                               int j;
```

Example of inheritance, with P on left, M in the middle, and R on the right.

Note the examples of new, recursive, and redefined attributes



- Inheritance can be thought of as incremental
  - A is parent to B
  - B is parent to C
- Thus only need to determine how to extend testing from parent t o child, and can use recursively to test grandchildren, etc.

Heirarchical Incremental Class Testing

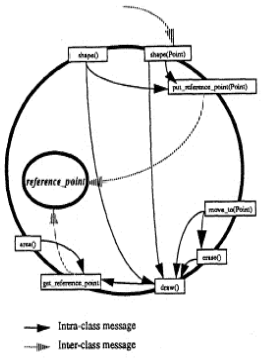
- Test base class
  - Test each member function
  - Test interactions among the member functions
  - Save test cases and execution information in a testing history
- Test subclass
  - Use testing history to avoid retesting when not necessary

Base Class Testing

- Test functions using traditional techniques
  - Specification based, or black box
    - (TS, test?)
  - Program based, or white box
    - (TP, test?)
- Test interactions among the member functions
  - Called integration testing
    - Focuses on interfaces between functions or units
      - IO format, format of entry or exit parameter values
  - Intra-class testing, when the functions are in the same class
    - (TIP, test?) or (TIS, test?)
  - Inter-class testing, when the functions are in different classes
  - Example is class Shape on next slides

```
Class Shape {
Private:        Point reference_point;
Public:         void put_reference_point( Point);  // access to data
                Point get_reference_point();  // access to data
                void move_to(Point); // defined to be erase() and draw()
                void erase();  // calls draw() to overwrite area
        virtual void draw() = 0;  // pure virtual – no implementation
        virtual float area();  // has an initial implementation
                shape(Point);  // constructor
                shape();  // constructor
}
```

```
class Shape {
private:
        Point reference_point;
public:
        void put_reference_point(Point);
        point get_reference_point();
        void move_to(Point);
        void erase();
virtual void draw() = 0;
virtual float area();
        shape(Point);
        shape();
}
```



Definition of class Shape and its class graph. Note the Inter-class messages go to class Point

Testing History for *Shape*

attribute	specification-based test suite	program-based test suite
Individual member functions		
put_reference_point	(TS_1,Y)	(TP_1,Y)
get_reference_point	(TS_2,Y)	(TP_2,Y)
move_to	(TS_3,Y)	(TP_3,Y)
erase	(TS_4,Y)	(TP_4,Y)
draw	(TS_5,Y)	$(--)$
area	(TS_6,Y)	(TP_6,Y)
shape	(TS_7,Y)	(TP_7,Y)
shape	(TS_8,Y)	(TP_8,Y)
Interacting member functions		
move_to	(TIS_9,Y)	(TIP_9,Y)
erase	(TIS_{10},Y)	(TIP_{10},Y)

The testing history for class Shape. Note that there are two shape functions but their integration test cases are omitted for brevity.

move_to() calls erase() and draw()

erase() calls draw()

Subclass Testing

- Transform testing history from parent to child
- Modifications are analyzed to transform the testing history
  - New or Virtual-New functions fully tested
    - Use Y for full retest with new test cases
  - Recursive or Virtual-Recursive functions not retested
    - Use N for no retesting
    - Integration tests run if interact with changed code (New or Redefined)
      - Use P for partially retested
  - Redefined or Virtual-Redefined functions fully tested
    - Use Y for retest, may reuse some test cases and build new ones

```
algorithm TestSubclass(HISTORY(P),G(P),M);

input:   HISTORY(P):P's testing history; G(P):P's class graph;
         M:modifier that specifies subclass R;
output:  HISTORY(R):testing history for R indicating what to rerun; G(R):class graph for subclass R;
begin
  HISTORY(R) := HISTORY(P);                          /* initialize R's history to that of P */
  G(R) := G(P);                                      /* initialize R's class graph to that of P */
  foreach attribute A ∈ M do
    case A is NEW or NEW-VIRTUAL:                    /* A is a new/virtual-new attribute */
        Generate TS , TP for A;
        Add {A, (TS, Y), (TP,Y)} to HISTORY(R);
        Integrate A into G(R);
        Generate any new TIS and TIP;
        Add {A, (TIS, Y), (TIP,Y)} to HISTORY(R);
    case A is RECURSIVE or RECURSIVE-VIRTUAL:        /* A is recursive/virtual-recursive */
        if A accesses data in R's scope then
          Identify interface tests to reuse;
          Add (TIS,P) and (TIP,P)} to HISTORY(R);
    case A is REDEFINED or REDEFINED-VIRTUAL:        /* A is redefined /virtual-redefined */
        Generate TP for A;
        Reuse TS from P if it exists or Generate TS for A;
        Add {A, (TS,Y), (TP,Y)} to HISTORY(R);
        Integrate A into G(R);
        Generate TIP for G(R) with respect to A;
        Reuse TIS from P;
        Add {A, (TIS,P), (TIP,P)} to HISTORY(R);
end TestSubclass.
```

Algorithm for transforming a testing history from parent to child

```
class Triangle: public Shape {
    private:
        Point vertex2;
        Point vertex3;
    public:
        Point get_vertex1();          //new
        Point get_vertex2();          //new
        Point get_vertex3();          //new
        void set_vertex1(Point);      //new
        void set_vertex2(Point);      //new
        void set_vertex3(Point);      //new
        void draw();                  //virtual-redefined
        float area();                 //virtual-redefined
        triangle();                   //new
        triangle(Point,Point,Point);  //new
}
```

Testing History for *Triangle*

attribute	specification test suite	program-based test suite
individual member functions		
put_reference_point	(TS_1,N)	(TP_1,N)
get_reference_point	(TS_2,N)	(TP_2,N)
move_to	(TS_3,N)	(TP_3,N)
erase	(TS_4,N)	(TP_4,N)
draw	(TS_5,Y)	(TP_5',Y)
area	(TS_6,Y)	(TP_6',Y)
shape	(TS_7,N)	(TP_7,N)
shape	(TS_8,N)	(TP_8,N)
get_vertex1	(TS_{11},Y)	(TP_{11},Y)
get_vertex2	(TS_{12}',Y)	(TP_{12}',Y)
get_vertex3	(TS_{13}',Y)	(TP_{13}',Y)
put_vertex1	(TS_{14}',Y)	(TP_{14}',Y)
put_vertex2	(TS_{15}',Y)	(TP_{15}',Y)
put_vertex3	(TS_{16}',Y)	(TP_{16}',Y)
triangle	(TS_{17}',Y)	(TP_{17}',Y)
triangle	(TS_{18}',Y)	(TP_{18}',Y)
interacting member functions		
move_to	(TIS_3',P)	(TIP_3',P)
erase	(TIS_{19}',P)	(TIP_{19}',P)
area	(TIS_{19}',Y)	(TIP_{19}',Y)
get_vertex1	(TIS_{20}',Y)	(TIP_{20}',Y)
put_vertex1	(TIS_{21}',Y)	(TIP_{21}',Y)

Definition of subclass Triangle and its testing history.

area(), get_vertex1(), and put_vertex1() call functions get_ and put_reference_point

```
class EquiTriangle: public Triangle{
    public:
        float area();                          //redefined
        equi_triangle(Point,Point,Point);      //new
        equi_triangle();                       //new
}
```

Testing History for *EquiTriangle*

attribute	specification test suite	program-based test suite
individual member functions		
put_reference_point	(TS_1,N)	(TP_1,N)
get_reference_point	(TS_2,N)	(TP_2,N)
move_to	(TS_3,N)	(TP_3,N)
erase	(TS_4,N)	(TP_4,N)
draw	(TS_5,N)	(TP_5,N)
area	(TS_6,Y)	(TP_6',Y)
shape	(TS_7,N)	(TP_7,N)
shape	(TS_8,N)	(TP_8,N)
get_vertex1	(TS_{11},N)	(TP_{11},N)
get_vertex2	(TS_{12},N)	(TP_{12},N)
get_vertex3	(TS_{13},N)	(TP_{13},N)
put_vertex1	(TS_{14},N)	(TP_{14},N)
put_vertex2	(TS_{15},N)	(TP_{15},N)
put_vertex3	(TS_{16},N)	(TP_{16},N)
triangle	(TS_{17},N)	(TP_{17},N)
triangle	(TS_{18},N)	(TP_{18},N)
equi_triangle	(TS_{22},Y)	(TP_{22},Y)
equi_triangle	(TS_{23},Y)	(TP_{23},Y)
interacting member functions		
move_to	(TIS_3',P)	(TIP_3',P)
erase	(TIS_{19}',P)	(TIP_{19}',P)
area	(TIS_{19}',P)	(TIP_{19}',P)

EquiTriangle is a subclass that modifies Triangle

Experimentation

- Determine the savings using this technique
  - Compare the number of attributes to test
- Code to test
  - Base class Interactor has subclass Scene
  - Class Scene has subclass MonoScene
  - Class MonoScene has subclass Dialog
  - Data next slide

Table 1: Interactor Class Hierarchy

class	*lines of code*	*number of attributes of each type*					
		new	*recursive*	*redefined*	*virtual new*	*virtual recursive*	*virtual redefined*
Interactor	908	79	0	0	14	0	0
Scene	195	21	59	0	8	14	1
MonoScene	98	1	73	0	4	16	4
Dialog	84	3	74	0	1	24	0

Table 2: Number of member functions to be tested (specification/program-based)

class	*retest all*	*our technique*	*our method/retest all*
Interactor	93	93	100%
Scene	96	30	31%
MonoScene	99	9	9%
Dialog	103	4	4%

Table 3: Number of member functions to be tested (interaction/interface)

class	*retest all*	*our technique*	*our method/retest all*
Interactor	93	93	100%
Scene	96	36	38%
MonoScene	99	9	9%
Dialog	103	6	6%

Conclusions

- Savings in the amount of testing
- Algorithmic approach may reduce time to analyze classes to determine what must be tested